

www.vuCTn.com
CS602 Complete MERGED
PPT Slides 1 TO 45

** Missing Slide (43,44) * LMS*

Regards : Zarva Chaudhary

Admin:

** Zarva Chaudhary **

Merged File date

31 -07 -2021

****Chaudhary Moazzam****

**** Laiba Maki ****

Computer Graphics

Lecture 01

Introduction

Taqdees A. Siddiqi

taqdees@vu.edu.pk

Definition

Computers...

accept, process, transform and present information.

Computer Graphics...

involve technology to accept, process, transform and present information in a visual form that also concerns with producing images (or animations) using a computer.

Why Study Computer Graphics?

- Visualization
 - I like to see what I'm doing
 - I like to show people what I'm doing
- Graphics is interesting
 - Involves simulation, algorithms, architecture...
- Requirement
 - Just ask Intel...
- Entertainment
 - Roll the video...

TEXT BOOK

Introduction to Computer Graphics
using Open GL
2nd Edition

by Donald Hearn and M. Pauline Baker
published by Prentice Hall

Course Outline

- ◆ Introduction
- ◆ Hardware
- ◆ Point
- ◆ Line
- ◆ Circle
- ◆ Mathematics
- ◆ Filling Algorithms I
- ◆ Filling Algorithms II
- ◆ Filling Attributes
- ◆ Transformation
- ◆ Types of Transformation
- ◆ Drawings in 2d
- ◆ 3d Concepts
- ◆ 3d Mathematics
- ◆ Projections

Course Outline

- ◆ Parallel Projection
- ◆ Types of Parallel Projection
- ◆ Perspective Projection
- ◆ Perspective Detail
- ◆ Hidden Surface Removal
- ◆ Rasterization
- ◆ Clipping
- ◆ Light And Light Sources
- ◆ Light Models
- ◆ Flat Shading
- ◆ Gouraud Shading
- ◆ Phong Shading
- ◆ Recursive Subdivision
- ◆ Introduction to Curves and Surfaces, Parametric Curves

Course Outline

- ◆ Cubic Curves and Interpolation
- ◆ Hermite Curves and Surfaces
- ◆ Bezier Curves and Surfaces
- ◆ Cubic B-Splines Curves and Surfaces
- ◆ General B-Splines
- ◆ Rendering of Curves and Surfaces
- ◆ Buffers and Mappings, Texture Mapping
- ◆ Environment Maps, Bump Maps
- ◆ Animation
- ◆ Recursive Methods and Fractals
- ◆ Ray Tracing I
- ◆ Ray Tracing II
- ◆ Radiosity

Course Organization

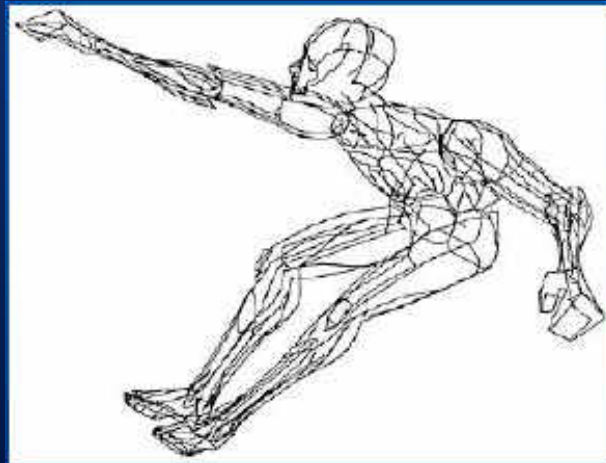
- ◆ Assignments: 8 15%
- ◆ Mid Term Exam: 1 35%
- ◆ Final Term Exam: 1 50%

Some History

History

The term “computer graphics” was coined in 1960 by William Fetter to describe new design methods that he was developing at Boeing.

He created a series of widely reproduced images on a plotter exploring cockpit design using a 3D model of a human body

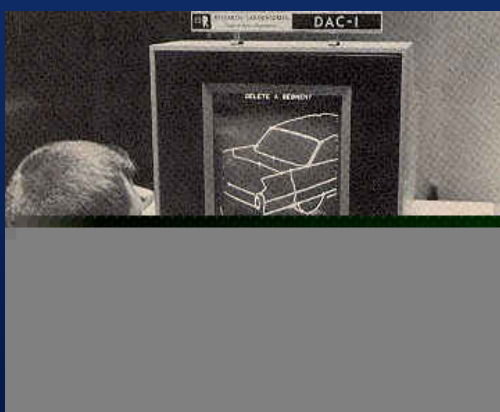


History

Whirlwind: early graphics using VectorScope (1951)



Spacewars: first computer graphics game (MIT 1961)

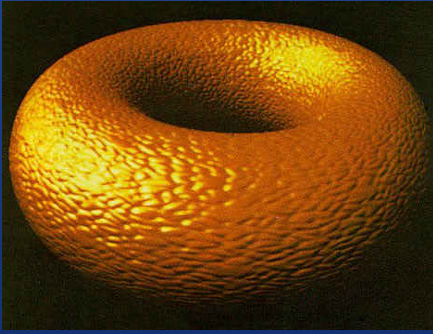


first CAD system (IBM 1959)



SketchPad: first interactive graphics (1961)

History



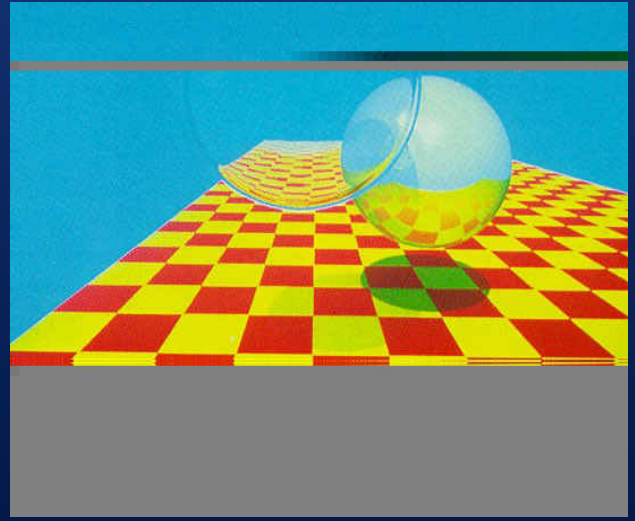
First bump-mapped images (Blinn 1978)



Early texture-mapped image (Catmull 1974)



First distributed ray traced image (Cook 1984)



First ray traced image (Whitted 1980)

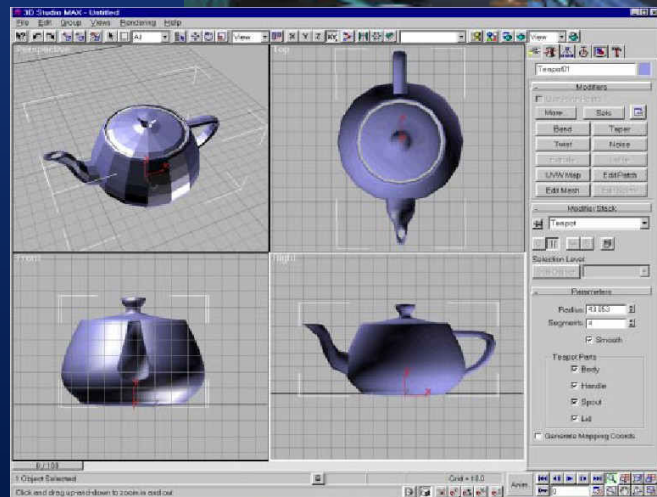
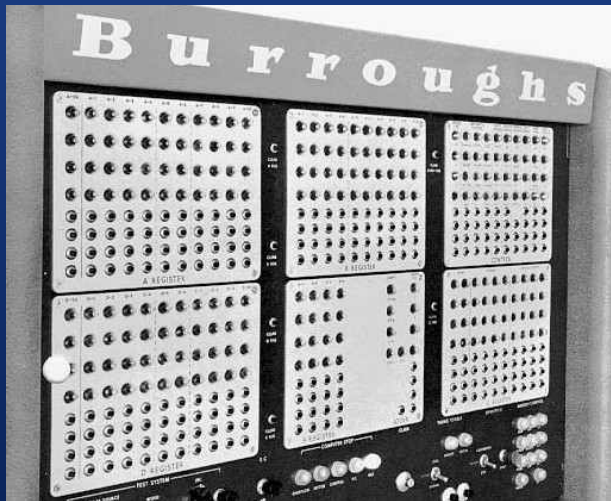
Graphics Applications

- User Interfaces
- Layout and Design
- Scientific Visualization and Analysis
- Art and Design
- Medicine and Virtual Surgery
- Layout Design & Architectural Simulations
- History and cultural heritage
- Entertainment
- Simulations
- Games

User Interfaces

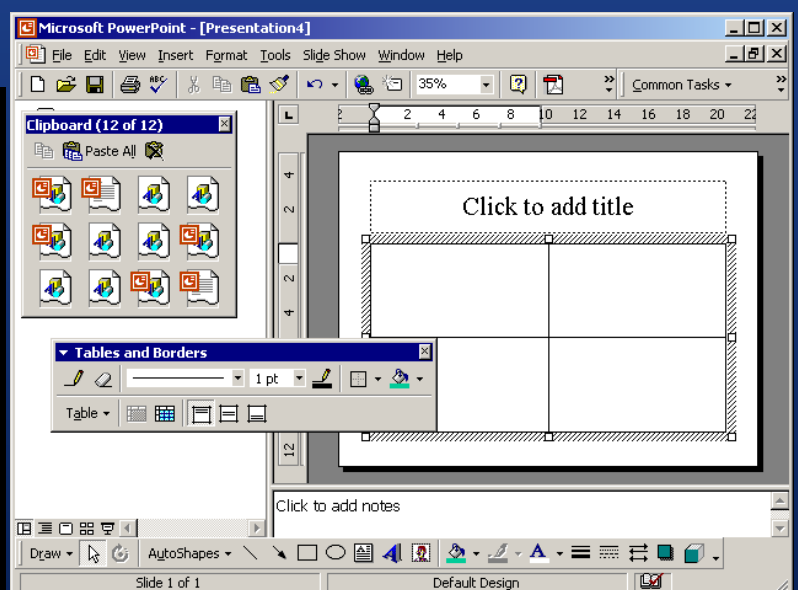
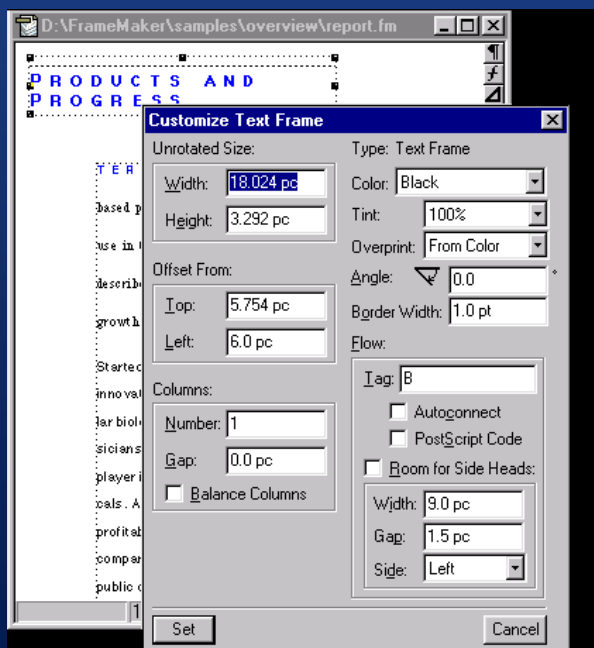
CAVE: Immersive and Interactive 3D environment

B205 Control Console (1960)

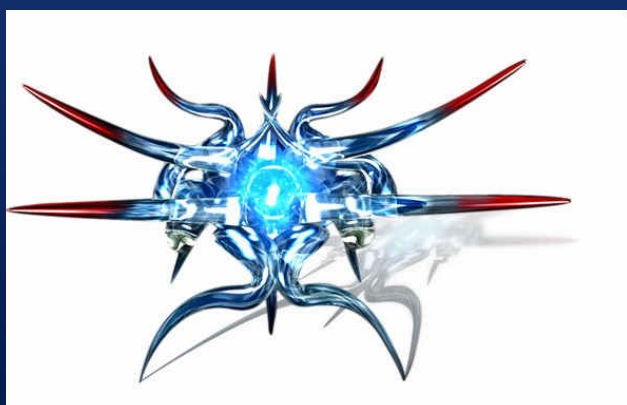
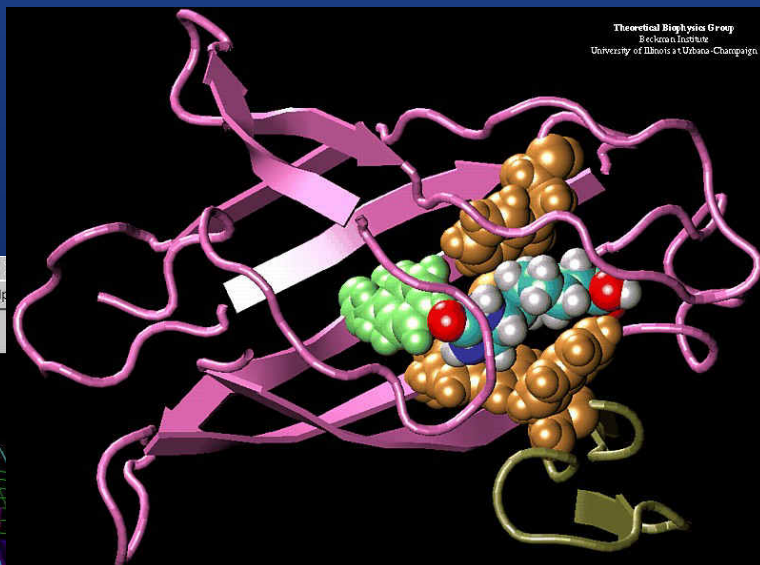
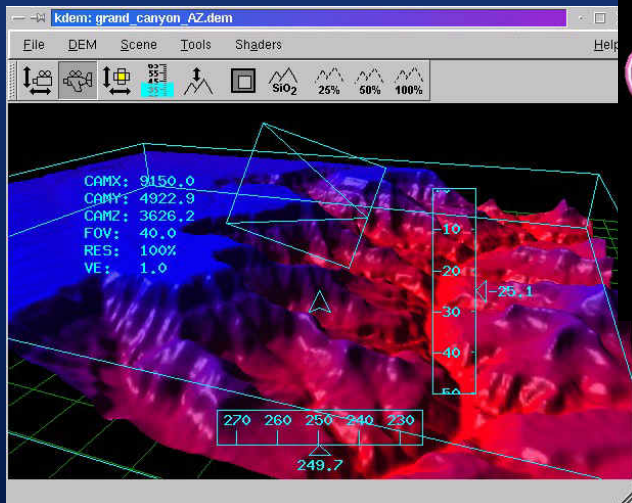


3D Studio MAX

Layout and Design

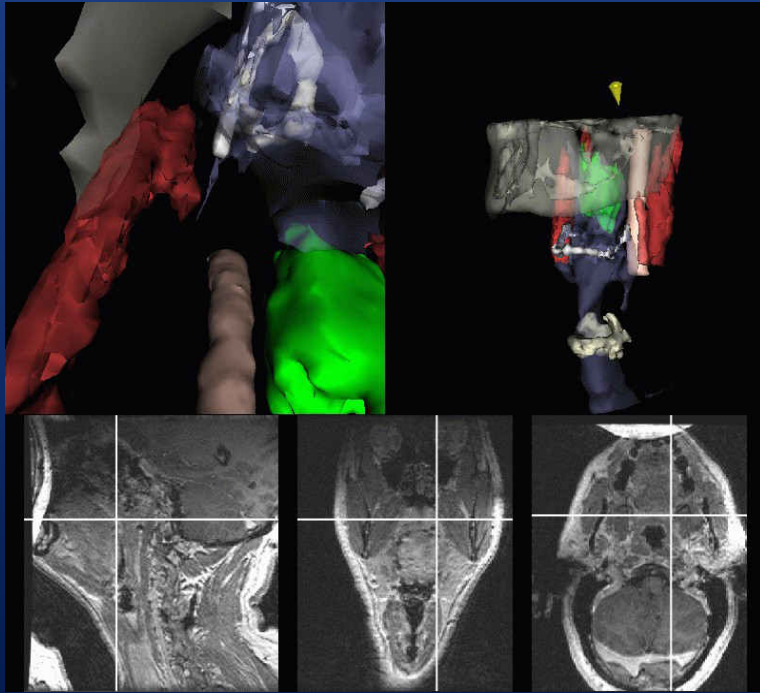


Scientific Visualization and Analysis



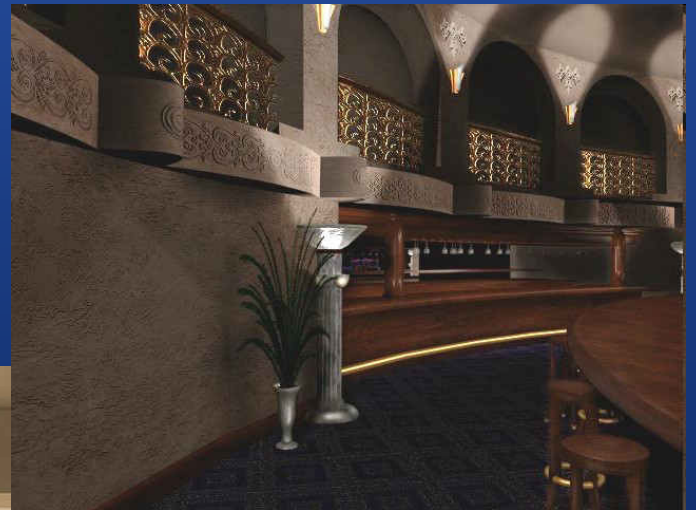
**ART AND
DESIGN**

Medicine and Virtual Surgery



The Visible Human project

Room Layout Design and Architectural Simulations

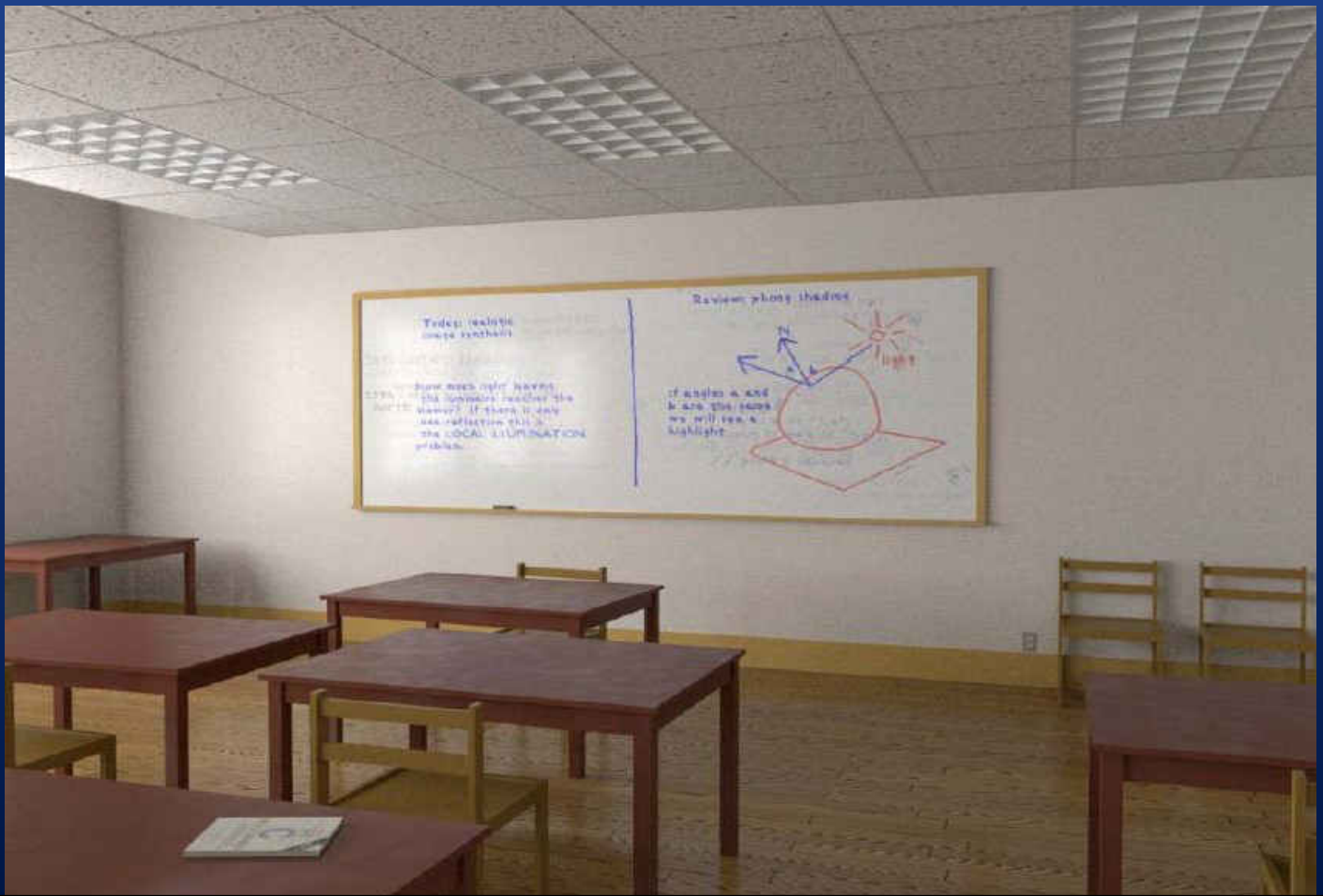


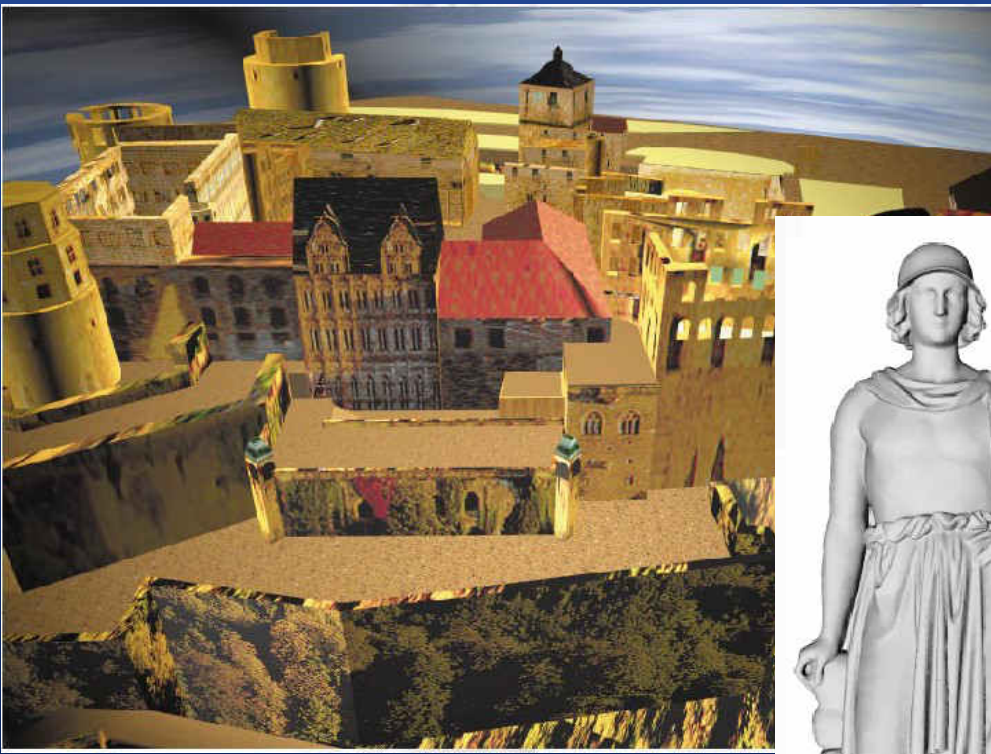
A 3D rendering of a chessboard with various geometric and organic shapes placed on it. The shapes include a wireframe cylinder, a yellow cylinder, a brown rectangular prism, a teal hexagonal prism, a blue fluted vase, a red sphere with black spots, a brown sphere, a yellow L-shaped block, a green and blue patterned sphere, a grey sphere, a pink cylinder, a yellow and black patterned sphere, a blue and red concentric circle, a white cone, a white torus, a yellow ring, a white teapot, a transparent cube with internal spheres, a yellow cube, a green and red polka-dot vase, and a black and white checkered cube. The chessboard is set on a grey surface.

Layout Design & Architectural Simulations



Layout Design & Architectural Simulations

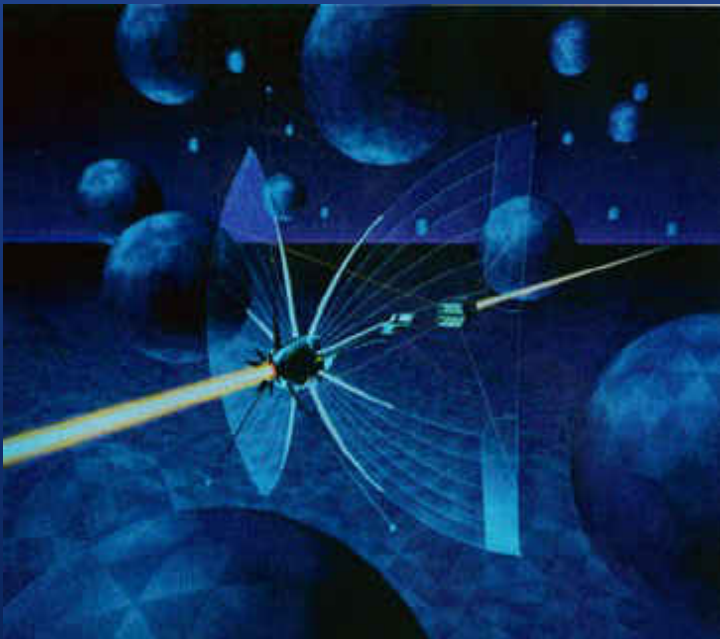




History and cultural heritage

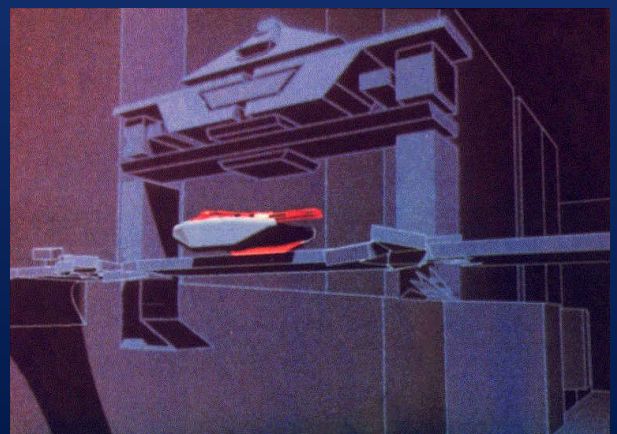
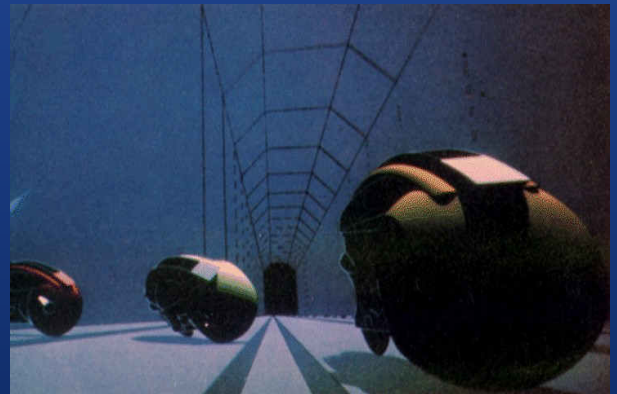


Movies



Tron (1980)

First time computer graphics were used for live action sequences.

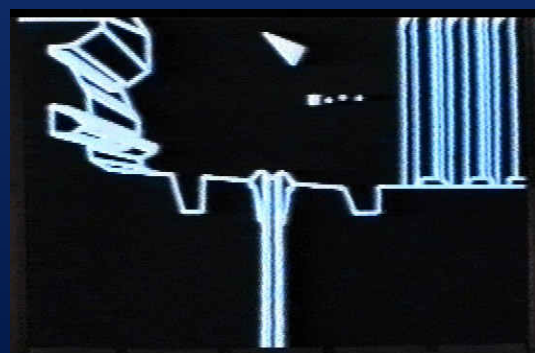
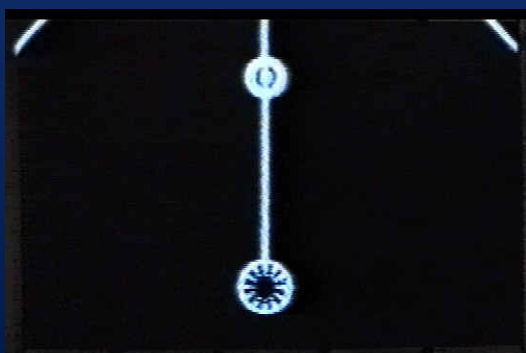
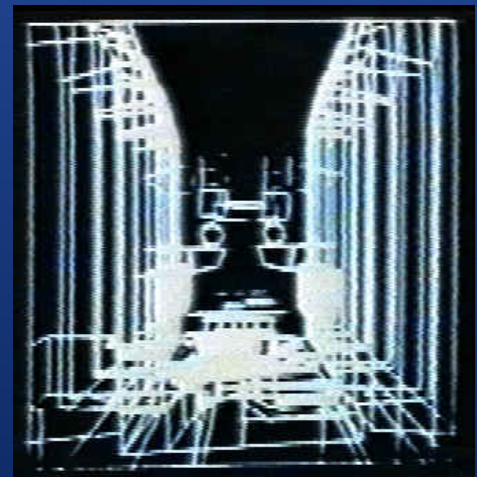


Movies

Fully computer generated
animated features



Movies

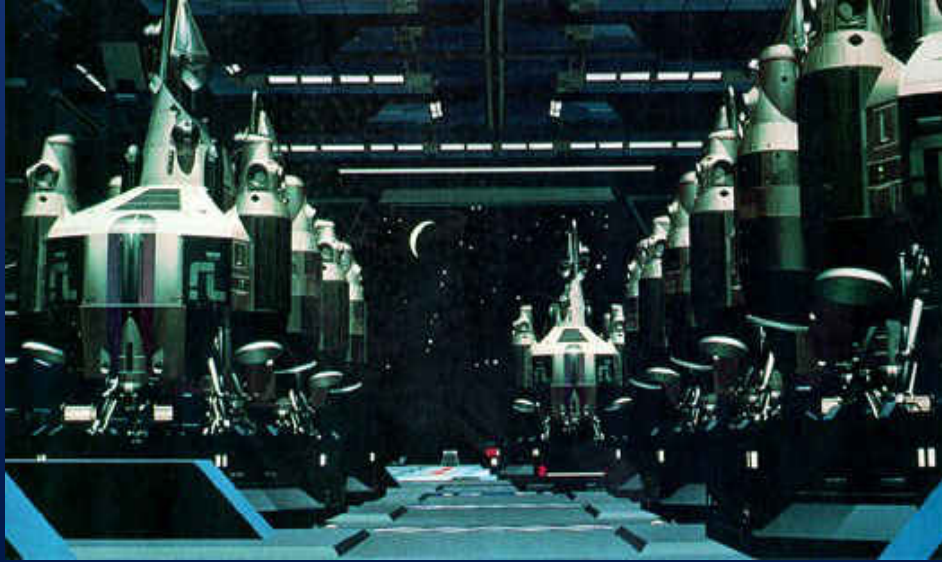


Star Wars (1977)

Movies



Star Trek II: The Wrath of Khan, genesis effect (1983)

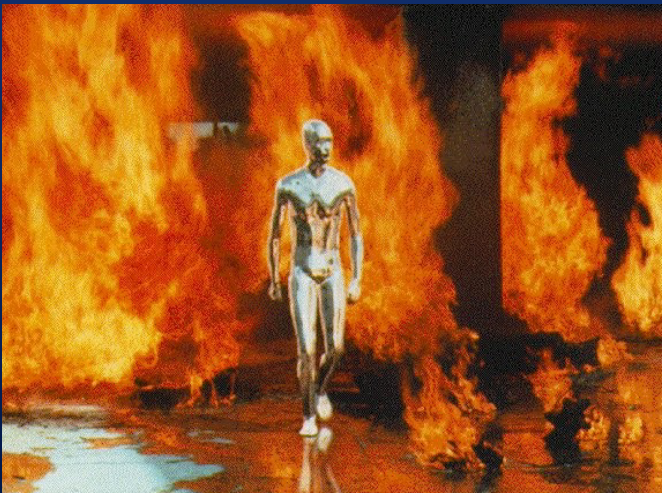


The Last Starfighter (15 minutes) (1982)

Movies

Special Effects... in
Live Action Cinema

Jurassic park, Titanic
and T2

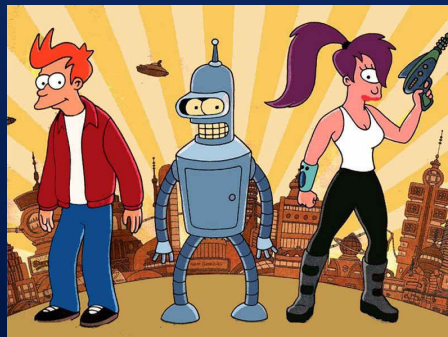


Movies

“Traditional” Animated Features...

Some examples:

- Automating Key-framing in many Disney-type animations
- The flocking behaviour of the wild-beast in Lion King
- Non photorealistic rendering: 3D effects in Futurama



Movies



Behind the scenes on *Antz* Production

Number of frames in the movie 119,592

Number of times the movie was rendered during production 15 (approx.)

Number of feet of approved animation produced in a week 107 ft.

Total number of hours of rendering per week 275,000 hrs.

Average size of the frame rendered 6 MB



Behind the scenes on *Antz* Production



Total number of Silicon Graphics servers used for rendering	270
Number of desktop systems used in production	166
Total Number of processors used for rendering	700
Average amount of memory per processor	256 MB

Simulations



Game

- Entertainment: Games

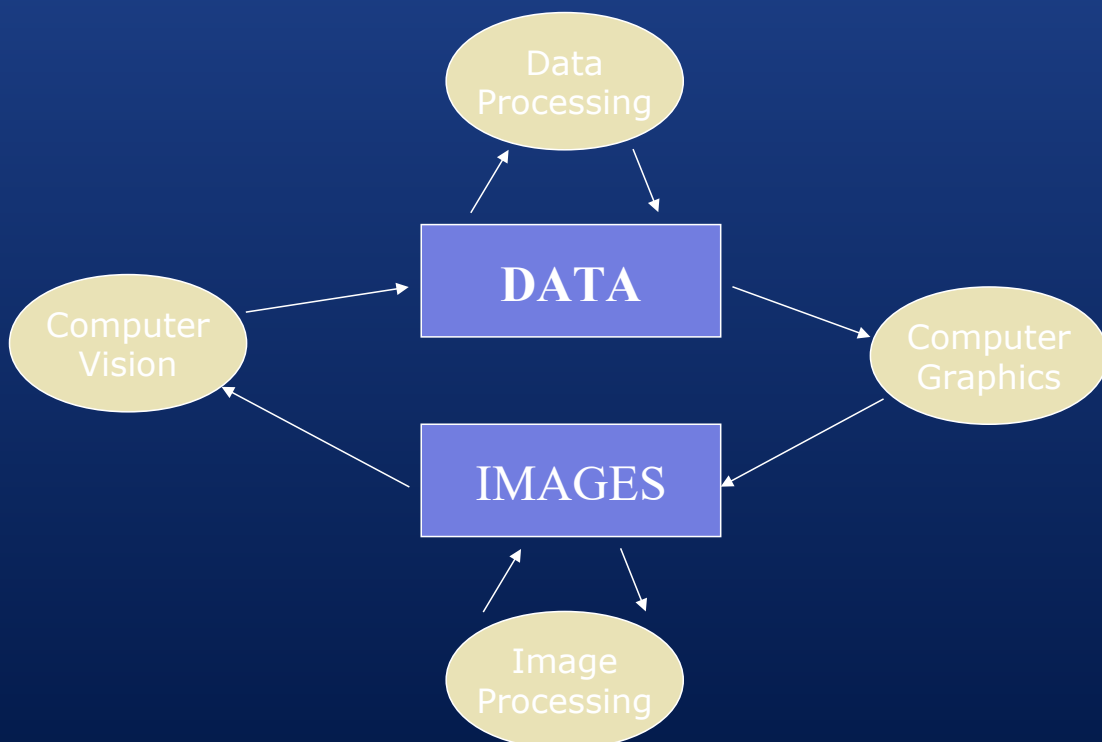


Cyan: Riven



Id: Quake II

Related Disciplines



Interdisciplinary

- ♦ Science
 - Physics: light, color, appearance, behaviour
 - Mathematics: Curves and Surfaces, Geometry and Perspective
- ♦ Engineering
 - Hardware: graphics media and processors, input and output devices
 - Software: graphics libraries, windows systems
- ♦ Art, Perception and Aesthetics
 - Color, Composition, Lighting, Realism

Computer Graphics

Lecture 01

Introduction

Taqdees A. Siddiqi

taqdees@vu.edu.pk

Movies



Average amount of memory per processor	256MB
Time it would have taken to render this movie on 1 processor	54 yrs., 222 days, 15 mins., 36 secs.
Amount of storage required for the movie	3.2 TB
Amount of frames kept online at any Given time	75000 frames

Computer Graphics

Lecture 02

Graphics Systems

Taqdees A. Siddiqi
taqdees@vu.edu.pk

Introduction to Graphics Systems

With the massive development in the field of computer graphics a broad range of graphics hardware and software systems are available.

With personal computers... usage of variety of interactive input devices and graphics software packages

Higher-quality applications... usage of special-purpose graphics hardware systems and technologies

Graphics Systems

- VIDEO DISPLAY DEVICES
- RASTER-SCAN SYSTEMS
- RANDOM-SCAN SYSTEMS
- GRAPHICS MONITORS AND WORKSTATIONS
- INPUT DEVICES
- HARD-COPY DEVICES
- GRAPHICS SOFTWARE

Video Display Devices

- Refresh Cathode-Ray Tubes
- Raster-Scan Displays
- Random-Scan Displays
- Color CRT Monitors
- Direct-View Storage Tubes
- Flat-Panel Displays
- Three-Dimensional Viewing Devices

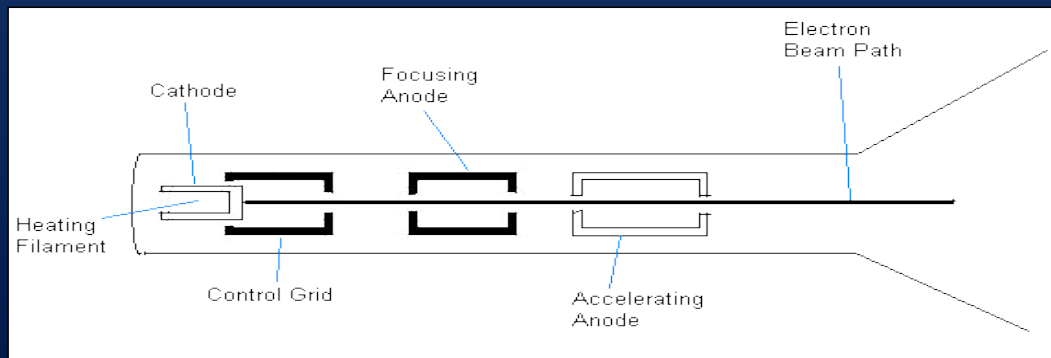
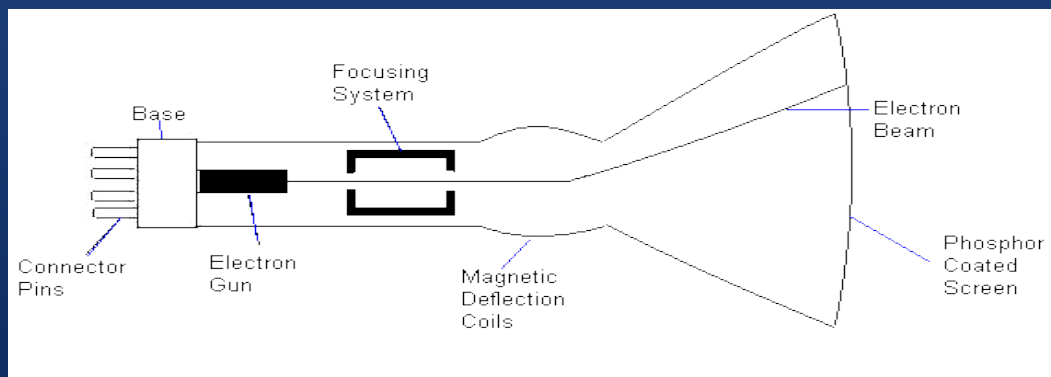
Refresh Cathode-Ray Tubes

Refresh CRT

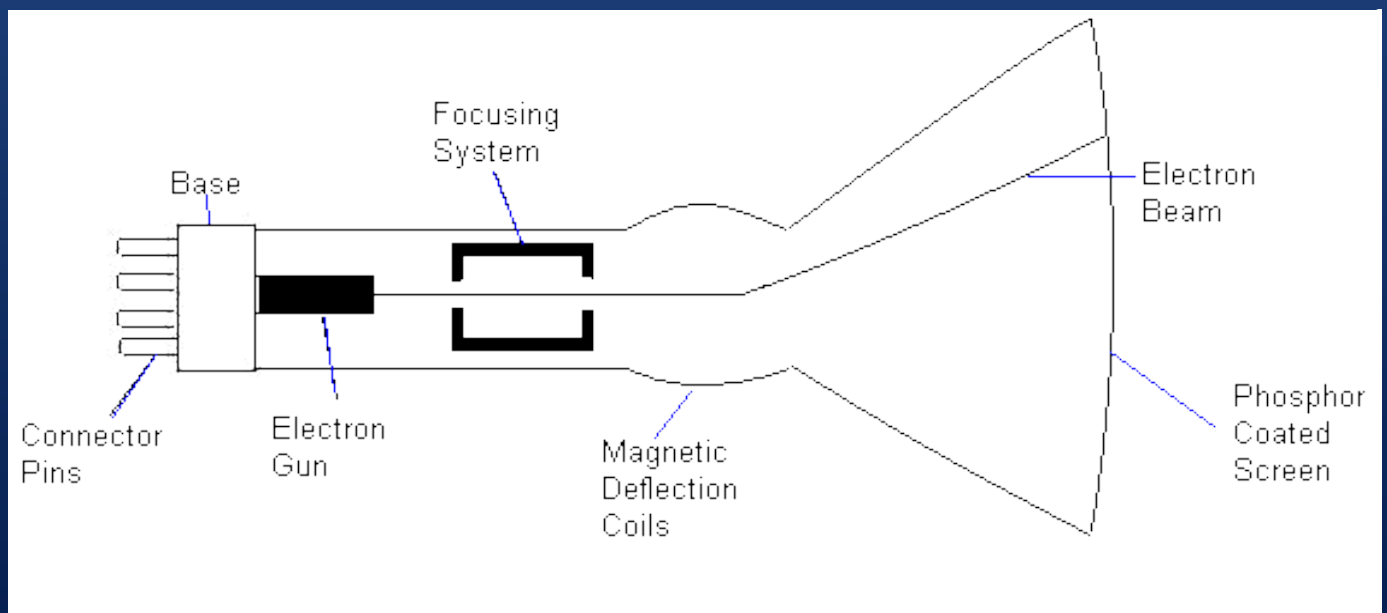
- Control Grid
- Focusing System
- Focusing in High Precision System
- Construction of Cathode-Ray Tubes
- Persistence
- Resolution
- Aspect Ratio



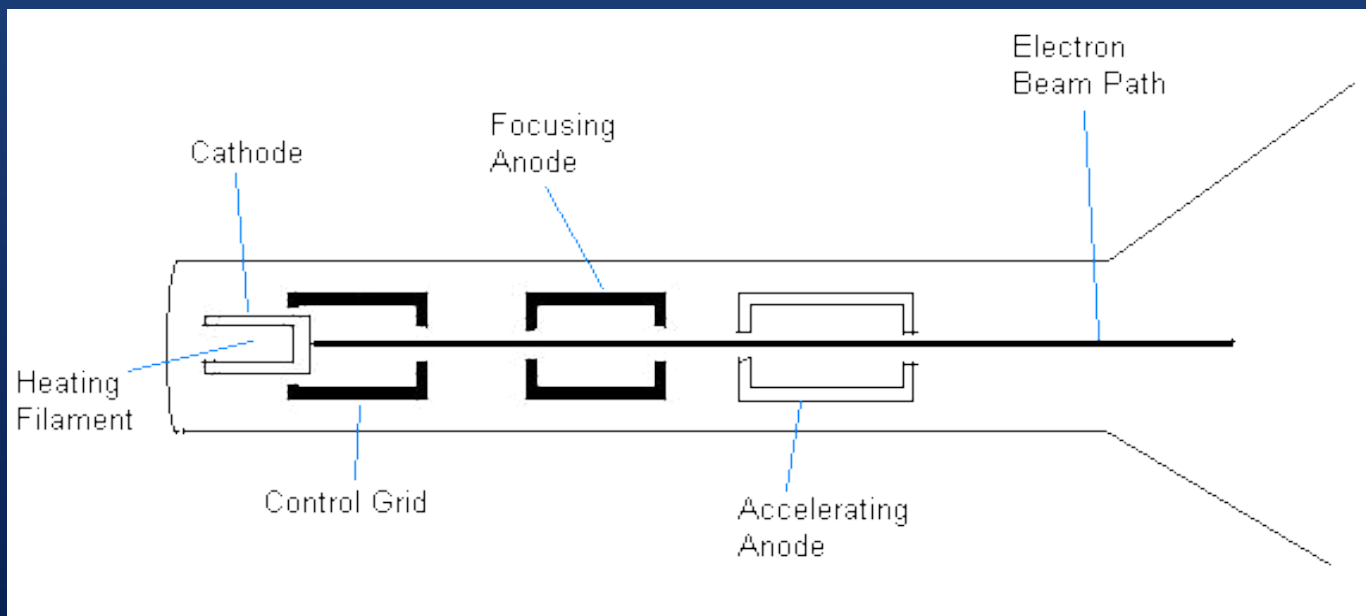
Refresh Cathode-Ray Tubes



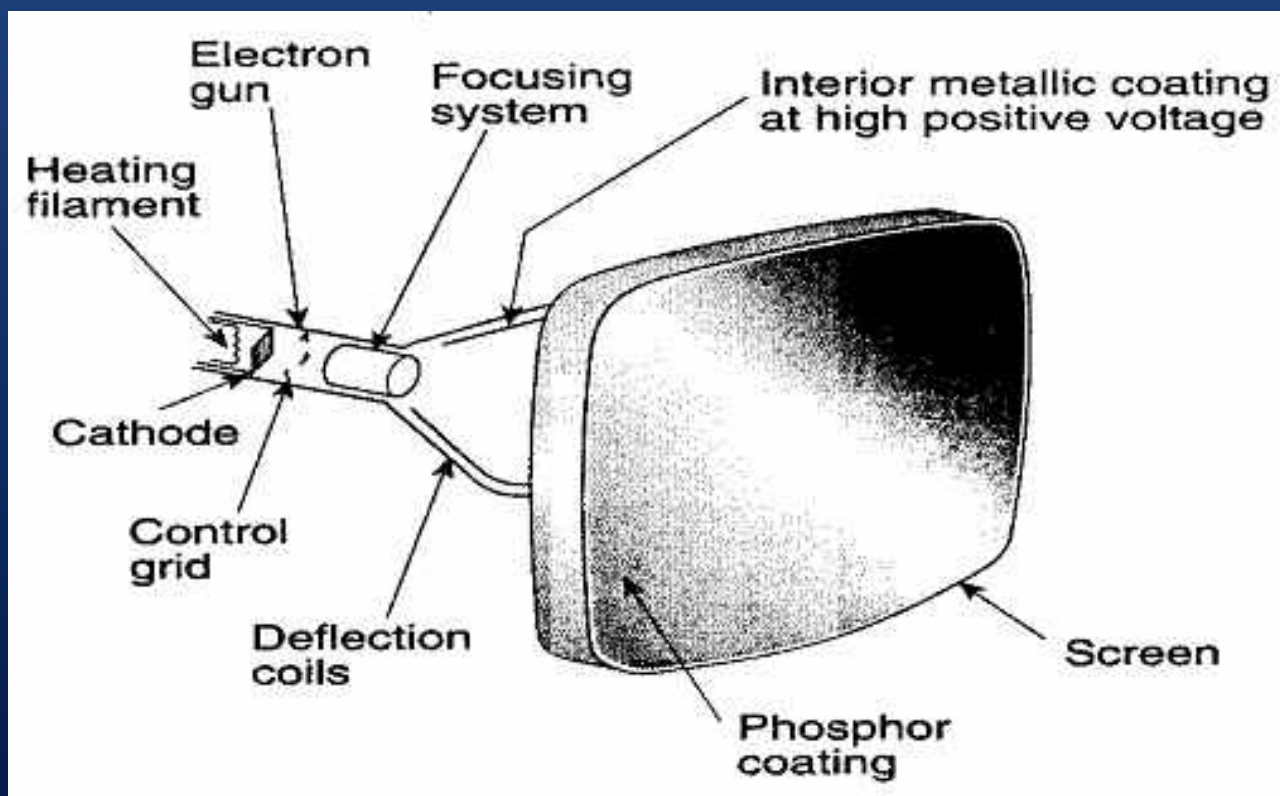
Refresh Cathode-Ray Tubes



Refresh Cathode-Ray Tubes



Refresh Cathode-Ray Tubes Cont..



Refresh Cathode-Ray Tubes

Refresh CRT

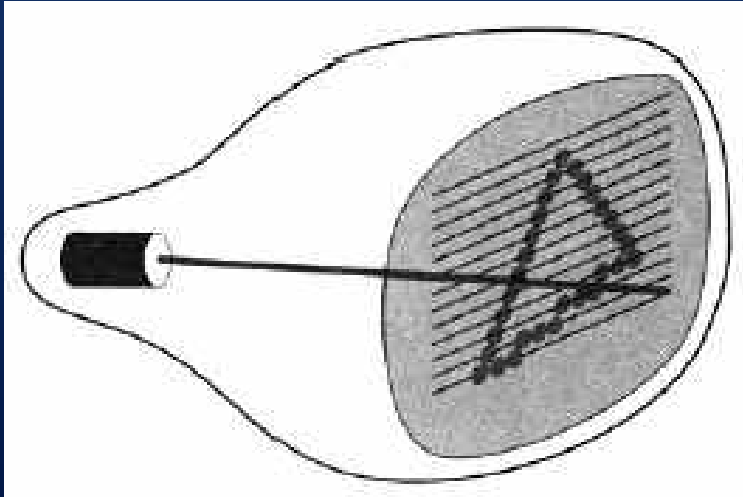
- Control Grid
- Focusing System
- Focusing in High Precision System
- Construction of Cathode-Ray Tubes
- Persistence
- Resolution
- Aspect Ratio

Refresh Cathode-Ray Tubes



Raster-Scan Displays

- Refresh Buffer or Frame Buffer
- Pixel
- Bitmap vs. Pixmap
- Horizontal Retrace & Vertical Retrace
- Interlaced vs. Non-interlaced Display

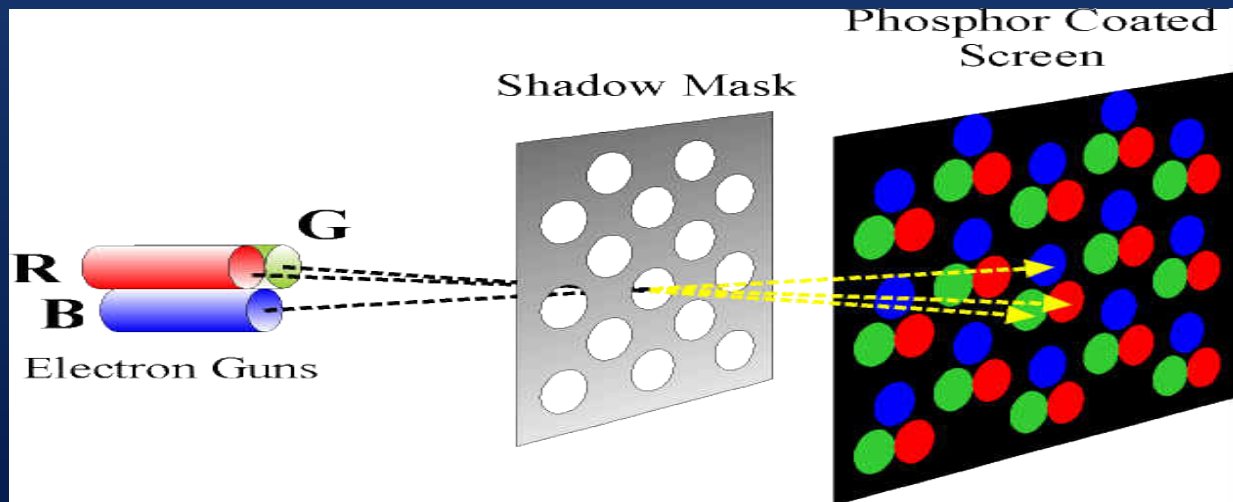


Random-Scan Displays

- ◆ Vector Display
- ◆ Refresh Display File
- ◆ Random-Scan vs. Raster-Scan

Color CRT Monitors

- Beam Penetration Method
- Shadow Mask Method



CRT Displays

Advantages

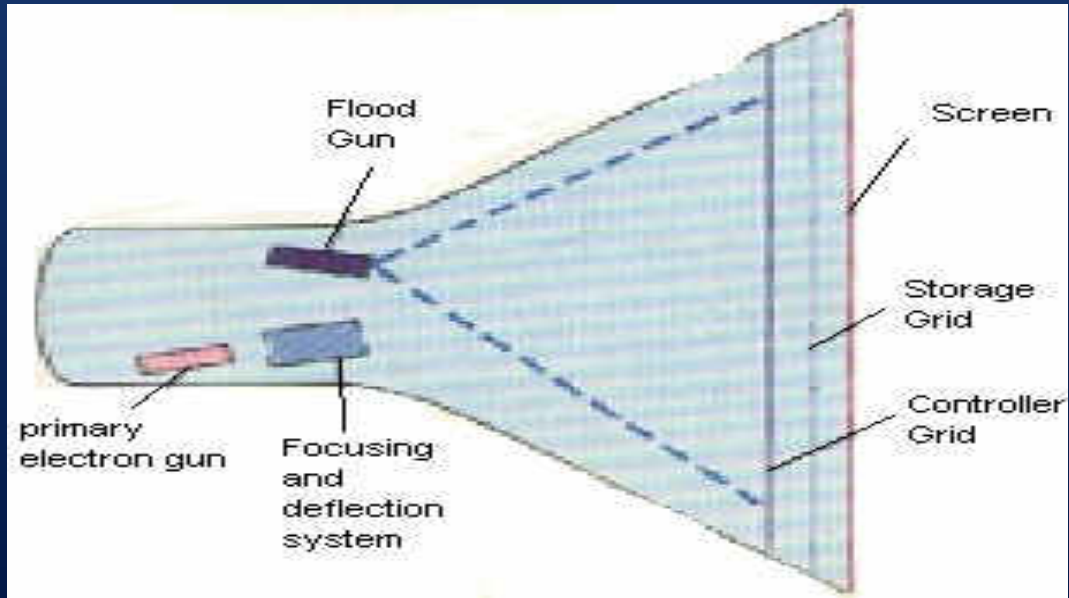
- ♦ Fast response (high resolution possible)
- ♦ Full colour (large modulation depth of E-beam)
- ♦ Saturated and natural colours
- ♦ Inexpensive, matured technology
- ♦ Wide angle, high contrast and brightness

Disadvantages

- ♦ Large and heavy (typ. 70x70 cm, 15 kg)
- ♦ High power consumption (typ. 140W)
- ♦ Harmful DC and AC electric and magnetic fields
- ♦ Flickering at 50-80 Hz (no memory effect)
- ♦ Geometrical errors at edges

Direct View Storage Devices

- Primary Gun
- Flood Gun
- Advantages & Disadvantages



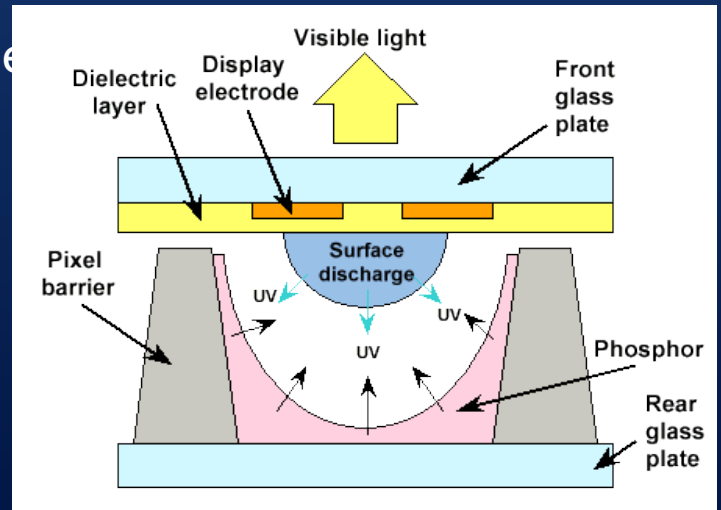
Flat-Panel Displays

- Properties
Volume, Weight, Power
- Emissive Display
Plasma Panels
- Non-Emissive Display
Liquid Crystal Display



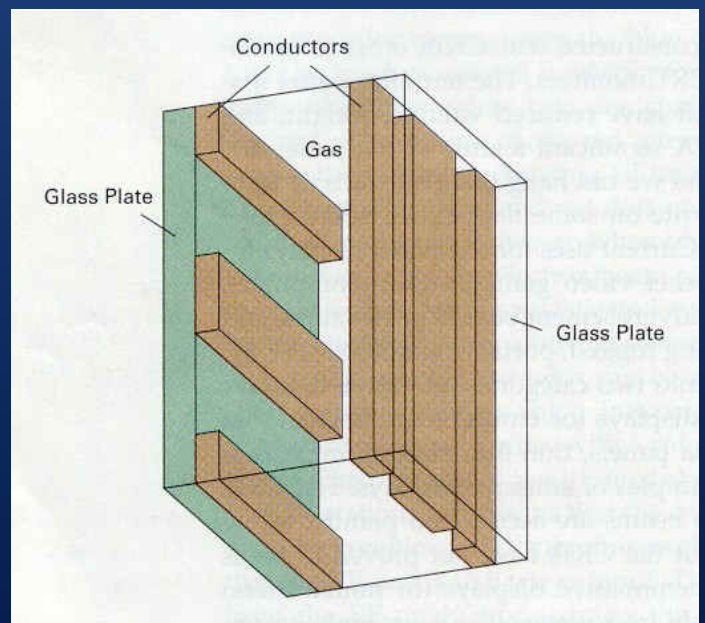
Plasma-panel Displays

- Similar in principle to fluorescent light tubes
- Small gas-filled capsules are excited by electric field, emits UV light
- UV excites phosphor
- Phosphor relaxes, emits some other color



Plasma-panel Displays

- Two glass plates separated by neon gas
- Vertical and horizontal conductors
- Very rugged
- No refreshing
- Screens can be transparent and overlay documents
- Used to be available in monochrome (Orange)
- Now available in colour



Plasma-panel Displays

Advantages

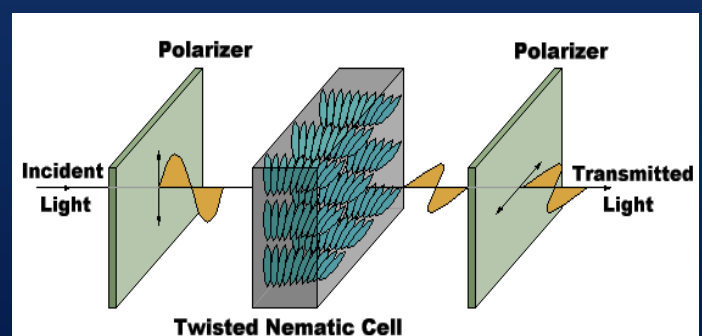
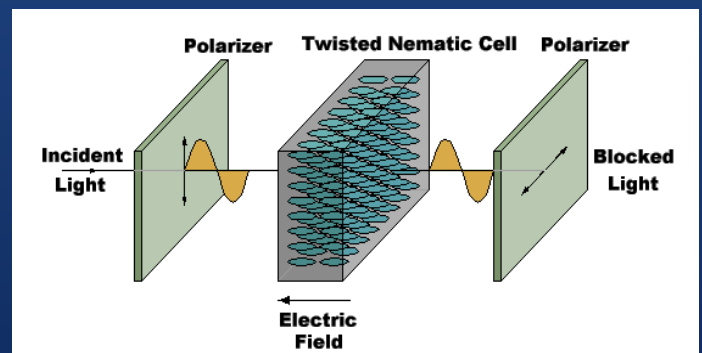
- Large viewing angle
- Good for large-format displays
- Fairly bright

Disadvantages

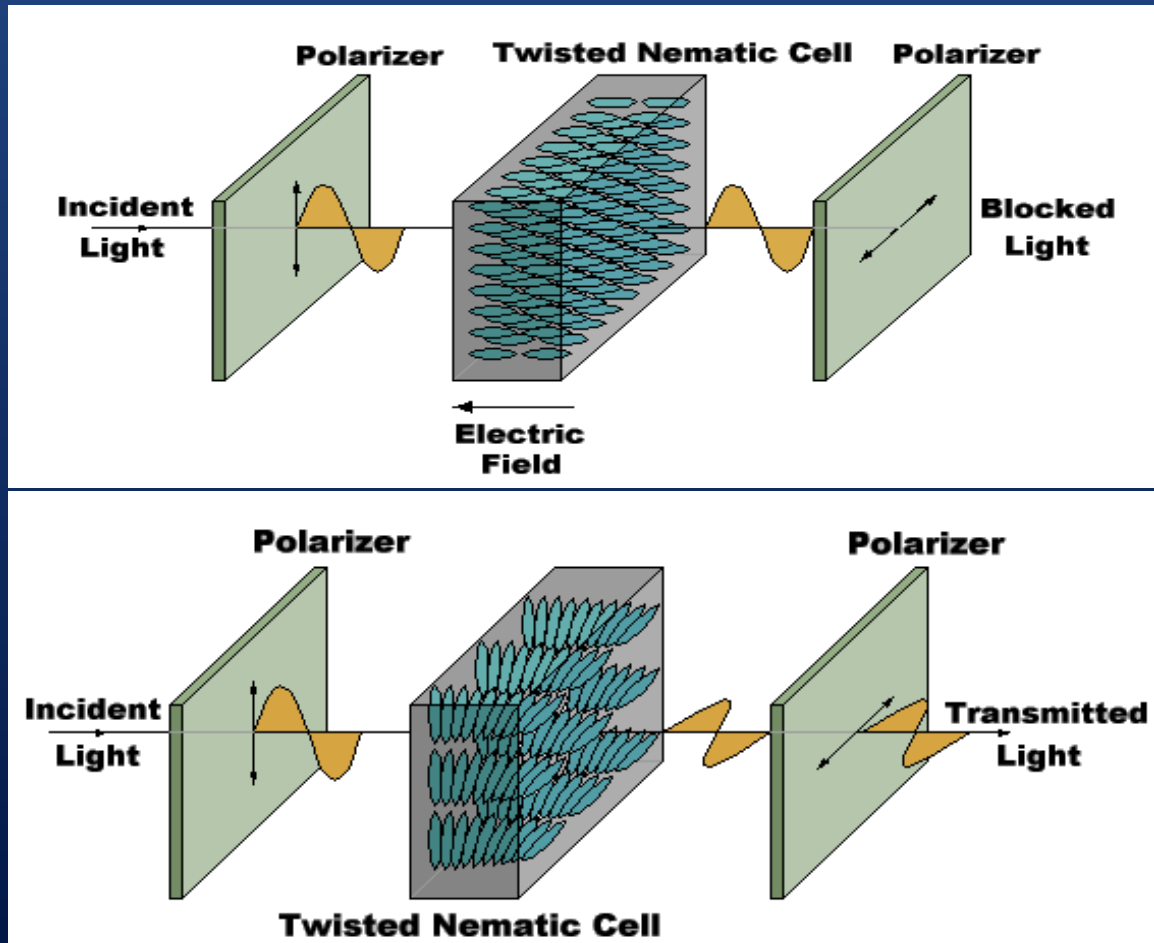
- Expensive
- Large pixels (~ 1 mm versus ~ 0.2 mm)
- Phosphors gradually deplete
- Less bright than CRTs, using more power

Liquid Crystal Displays

- Liquid Crystal Display
- A transmissive technology
- Works by letting varying amounts of a fixed-intensity white backlight through an active filter
- Organic crystals that align themselves together
- When external force is applied they realign themselves
- This is used to change polarisation and filter light

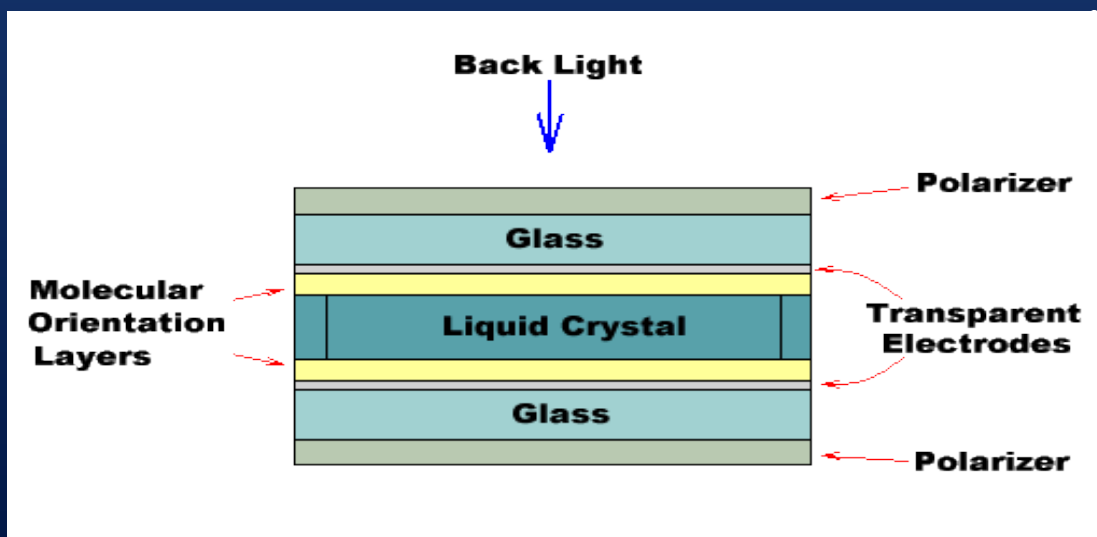


Liquid Crystal Displays



Liquid Crystal Displays

- **Transmissive & reflective LCDs:**
 - LCDs act as light valves, not light emitters, and thus rely on an external light source.
 - Laptop screen: backlit, *transmissive display*
 - Palm Pilot/Game Boy: *reflective display*



Liquid Crystal Displays

Advantages

- ◆ Small footprint (approx 1/6 of CRT)
- ◆ Light weight (typ. 1/5 of CRT)
- ◆ Low power consumption (typ. 1/4 of CRT)
- ◆ Completely flat screen - no geometrical errors
- ◆ Crisp pictures - digital and uniform colours
- ◆ No electromagnetic emission
- ◆ Fully digital signal processing possible
- ◆ Large screens (>20 inch) on desktops

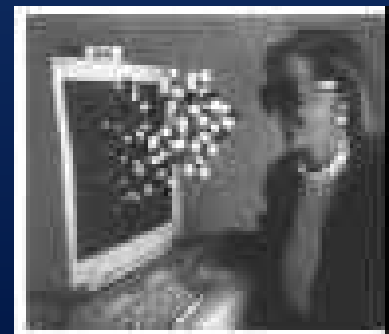
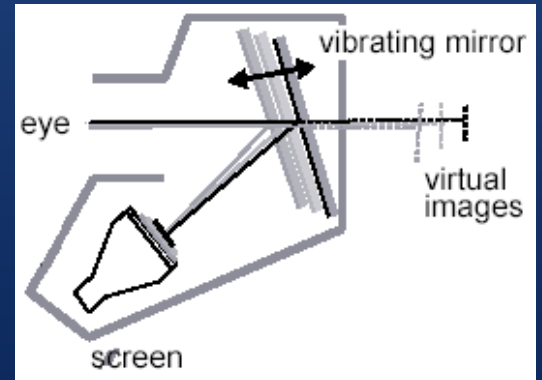
Liquid Crystal Displays

Disadvantages

- ◆ High price (presently 3x CRT)
- ◆ Poor viewing angle (type. +/- 50 degrees)
- ◆ Low contrast and luminance (type. 1:100)
- ◆ Low luminance (Natural light) (type. 200 cd/m²)

Three-Dimensional Viewing Devices

- **Vibrating flexible mirror**
- **Stereoscopic and virtual**
 - reality systems
 - shutter glasses (with tracking)
 - headset with tracking
 - other VR devices
- **Holograms**
- **NC-mills + soft material**
- **Stereo-lithographic apparatus (SLA)**



Virtual Reality Devices

Virtual reality = stimulation of the human senses to ***simulate a fictive environment***

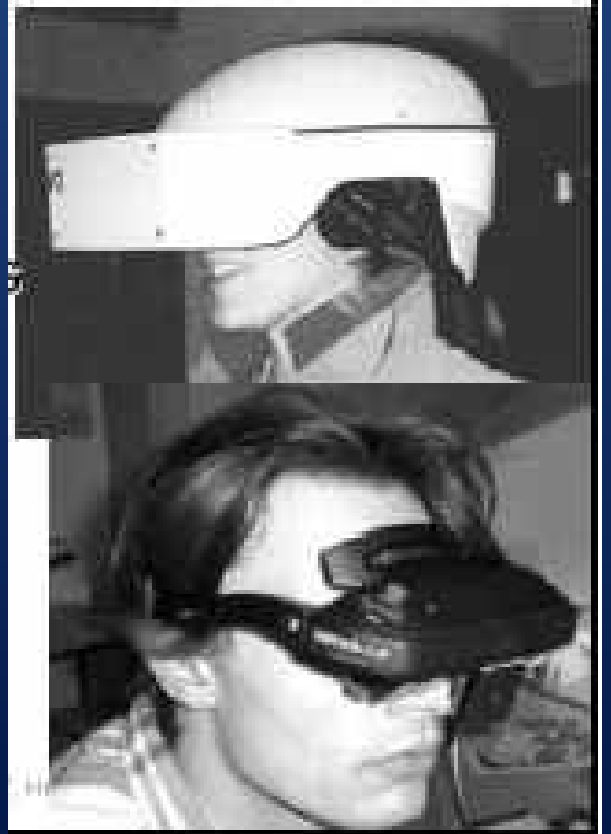
eyes: private eye, head mounted displays (hmd)

ears: headphones, loudspeakers in hmd

tactile sense: data glove, data suit, treadmill

Head Mounted Display

- 2 small screens display
- images for both eyes:
- high immersion degree
- head movement controls
- virtual camera
- artificial image only
- low resolution (e.g. 256 x 210)
- very expensive
- narrow view-angle



Light Weight HMD

ultrasound tracking device used with stereoscopic glasses to track head position



Interactive Walkthrough



Data Glove



data glove as
interaction
device

HMD in combination
with data glove

Computer Graphics

Lecture 02

Graphics Systems

Taqdees A. Siddiqi

taqdees@vu.edu.pk

Computer Graphics

Lecture 03

Graphics Systems Cont...

Taqdees A. Siddiqi

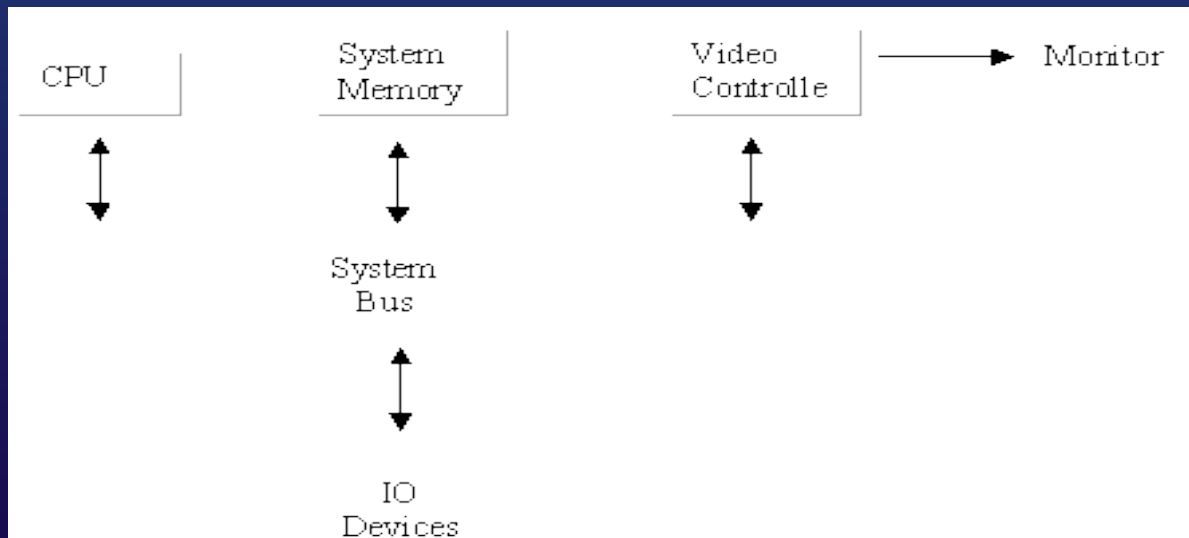
taqdees@vu.edu.pk

Graphics Systems Cont...

- RASTER-SCAN SYSTEMS
- RANDOM-SCAN SYSTEMS
- GRAPHICS CARDS OR DISPLAY ADAPTORS
- GRAPHICS SOFTWARE

Raster-Scan Systems

- In simple organization; frame buffer can be anywhere in the system memory
- Video controller accesses the frame buffer to refresh the screen

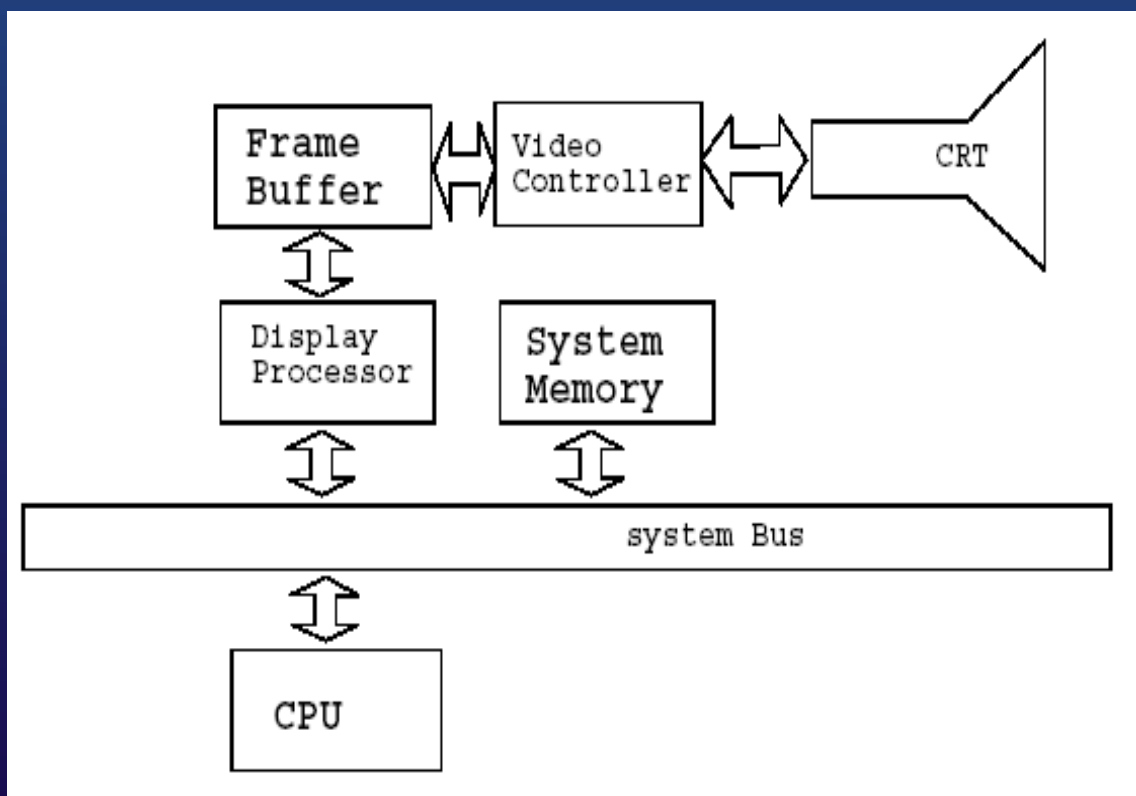


Architecture of a simple raster graphics system

Raster-Scan Systems

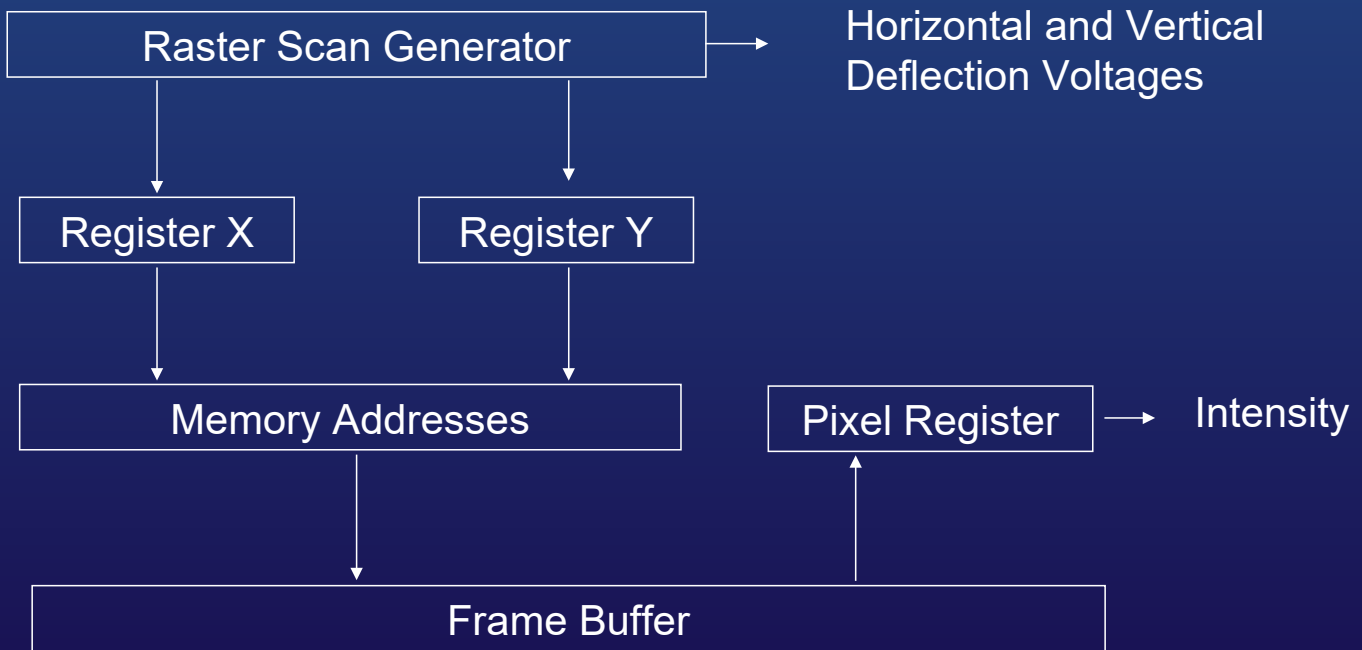
- Video controller or display controller
- Raster scan display processor

Video Controller



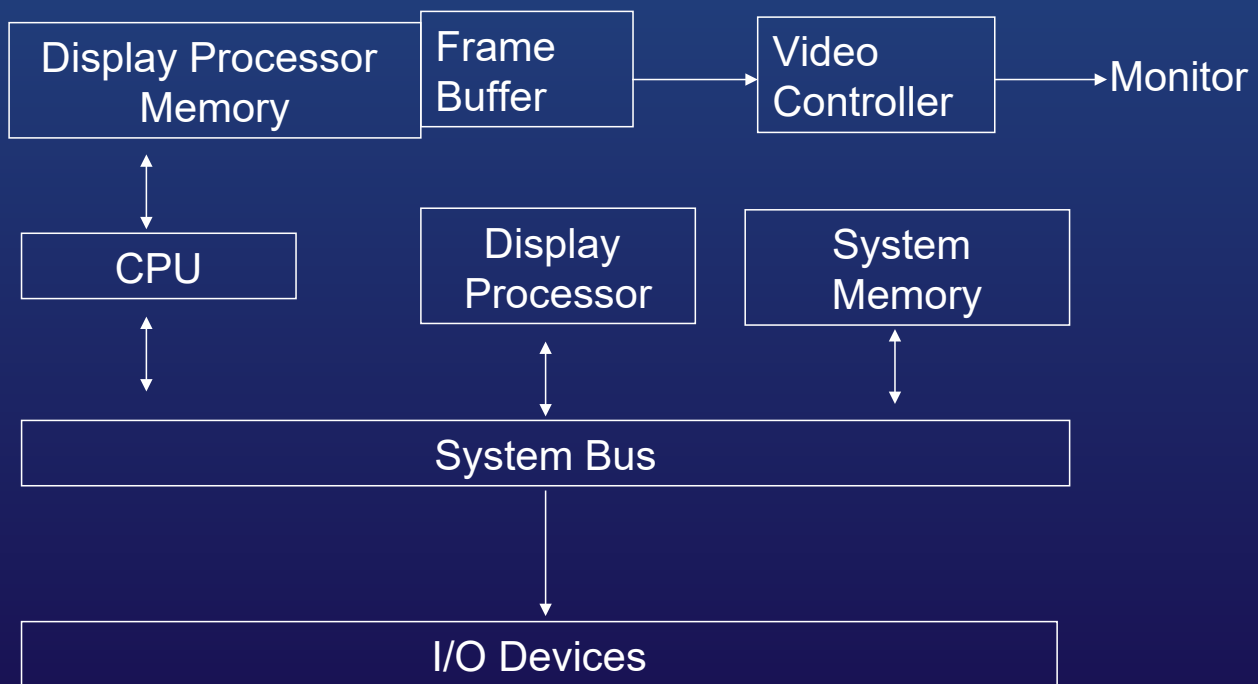
Architecture of a raster system with a fixed portion of system memory reserved for the frame buffer.

Video Controller



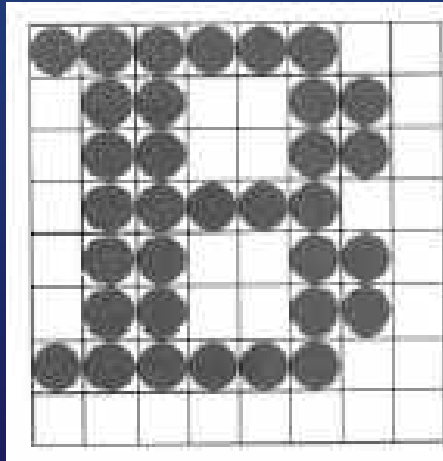
Basic Video Controller Refresh Operations

Raster Scan Display Processor

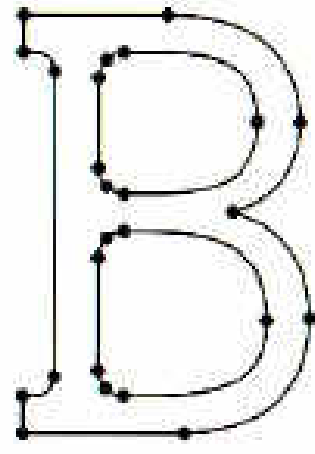


Architecture of a raster graphics systems with a display processor

Raster-Scan Characters

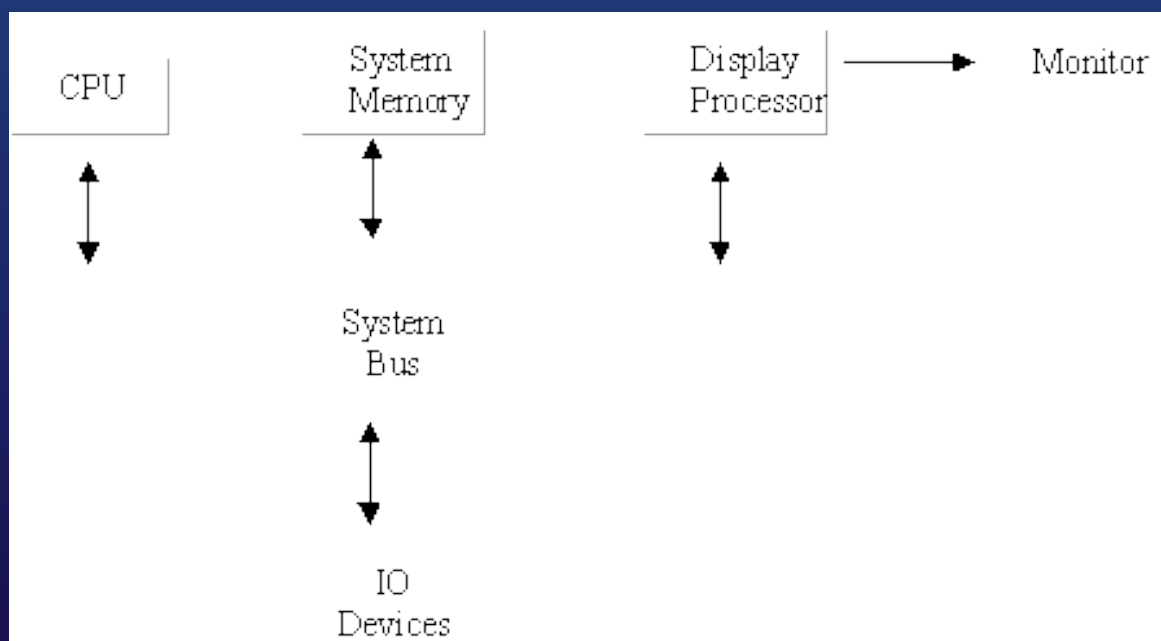


Defined as a grid
of pixel positions



Defined as a
curve outline

Random-Scan Systems



Architecture of a simple random scan system

Graphics Card or Display Adaptors

A video card is typically an adaptor, a removable expansion card in the PC. Thus, it can be replaced!

A video display Adaptor which is the special printed circuit board that plugs into one of the several expansion slots present on the mother board of the computer. A video display Adaptor is referred to as a video card as well.

The video card can also be an integral part of the system board; this is the case in certain brands of PCs and is always the case in laptops and clear preference for the replaceable video card in some PCs.

Graphics Card or Display Adaptors

A number of display Adaptors are available with varying capabilities specially Intel systems support following Adaptors:

- Monochrome Adaptor (MA)
- Hercules Adaptor (HA)
- Color Graphics Adaptor (CGA)
- Enhanced Graphics Adaptor (EGA)
- Multicolor Graphics Adaptor (MCGA)
- Video Graphics Adaptor (VGA)
- Super Video Graphics Adaptor (SVGA)
- Extended Graphics Adaptor (XGA)

Monochrome Adaptor

- Simplest and the first available Adaptor
- Can display only text and that too in single color

Hercules Adaptor

- Emulates the Monochrome Adaptor
- Can also operate in a graphics mode

Color Graphics Adaptor

- Can display text and graphics in more than one color
- Text mode is supported with 25 rows by 80 columns
- Two resolutions available:
 - 320 * 200 with 4 colors from a palette of 16
 - 640 * 200 with 2 colors

Color Graphics Adaptor

- One drawback is that it produces:
 - Flicker
 - Snow

Enhanced Graphics Adaptor

- Introduced by IBM in 1984 as alternative to CGA
- EGA could emulate all functions of CGA and MA
- palette of 64 colors

Enhanced Graphics Adaptor

- Designed to avoid:
 - Snow
 - Flicker
- A serious limitation is that it supports only write operation and no read operations

Multicolor Graphics Adaptor

- Designed to emulate the CGA
- To maintain compatibility with all the CGA modes
- Provides two new graphics modes
 - 640 * 480 in 2 colors
 - 320 * 200 in 256 colors

Video Graphics Adaptor

- Supports all the display modes of MA, CGA, MCGA
- Provides graphics mode
 - 640 * 480 in 16 colors

Super Video Graphics Adaptor

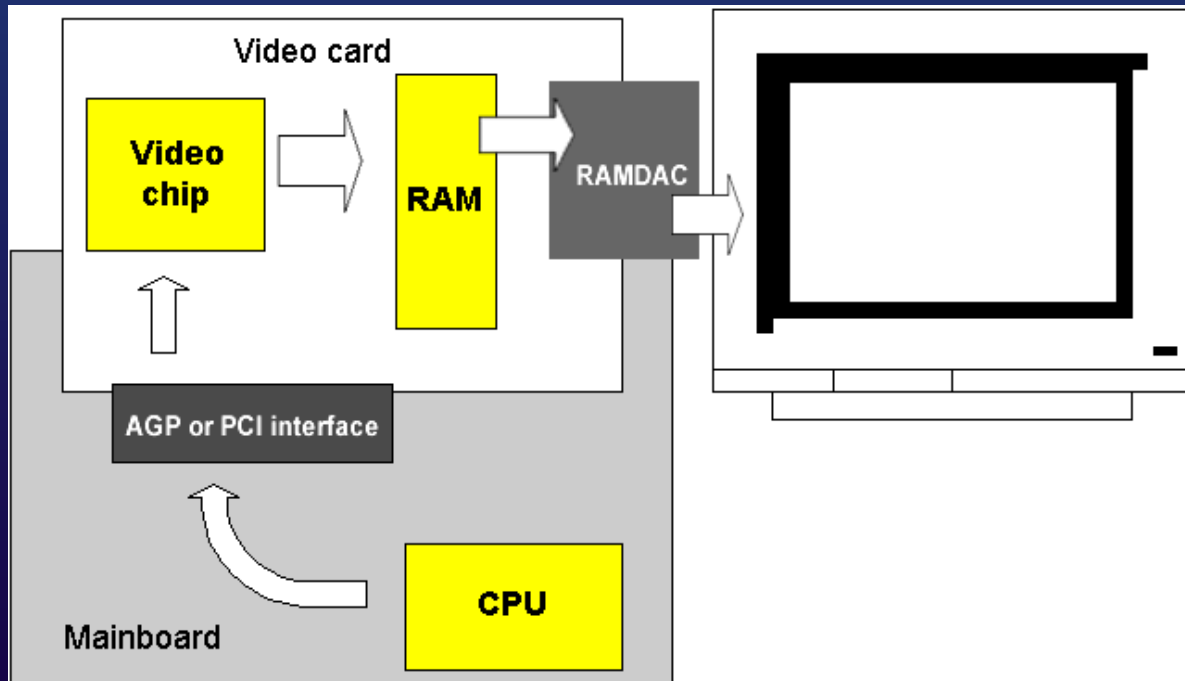
- SVGA refers to enhancements to the VGA
- SVGA has different capabilities on different cards for example:
 - 800 * 600 and 1024 * 768
 - Another may have same resolution but more colors
- SVGA required different driver for different cards

Extended Graphics Adaptor

- XGA evolved from the VGA provides:
 - greater resolution
 - More colors
 - Much better performance
- XGA has its own graphics processor and bus mastering
- XGA offers 2 new modes:
 - 640 * 480 with 16 bit colors
 - 1024 * 768 with 8 bit colors

Video Card Supports the CPU

The video card provides a support function for the CPU. It is a processor like the CPU. However it is especially designed to control screen images.



RAM on the Video Card

- How much RAM? That is significant for color depth at the highest resolutions
- Which type of RAM? This is significant for card speed

3D - lots of RAM

- To support the demand for high quality 3D performance new cards are coming with a frame buffer of 16 or 32 MB RAM
- They use the AGP interface for:
 - better bandwidth
 - better access to the main memory

VRAM

Most cards use very fast editions of ordinary RAM (SDRAM or DDR)

Some high end cards (like Matrox Millennium II) earlier used special VRAM (Video RAM) chips.

In principle, a VRAM cell is made up of two ordinary RAM cells, which are "glued" together. Therefore, you use twice as much RAM than otherwise.

VRAM

VRAM has features:

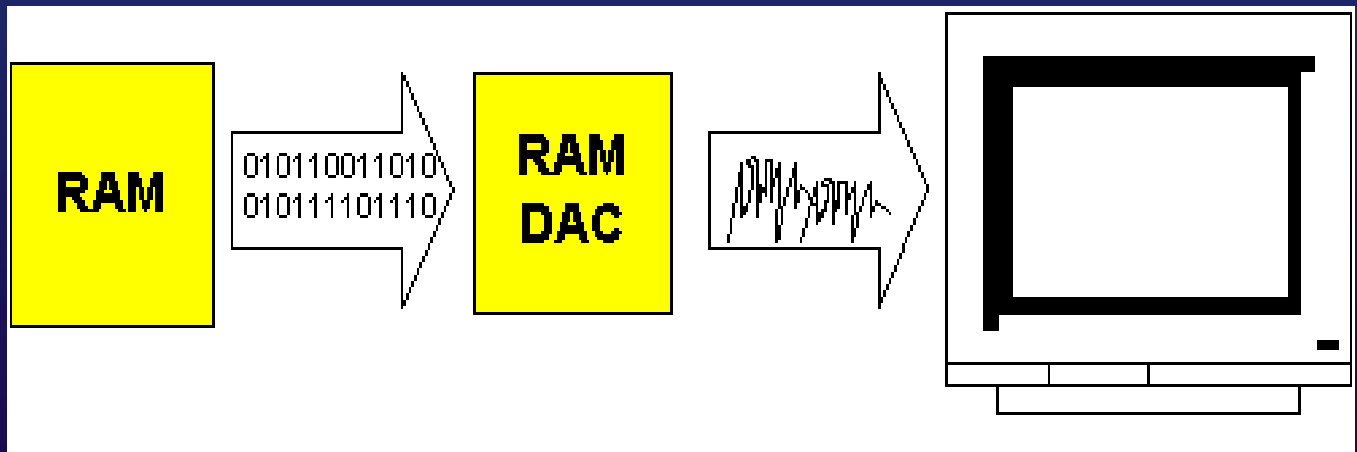
- Costs twice
- Double cell allows the video processor to simultaneously read old and write new data on the same RAM address
- VRAM is capable of reading and writing simultaneously due to the dual port design.

UMA and DVMT

- Unified memory architecture (UMA) was very slow
- Shared Memory Buffer Architecture (SMBA)
- DVMT, Dynamic Video Memory Technology, found on Intel chip set 810 and 815
- DVMT integrates the graphics controller and uses part of the system RAM as frame buffer

The RAMDAC

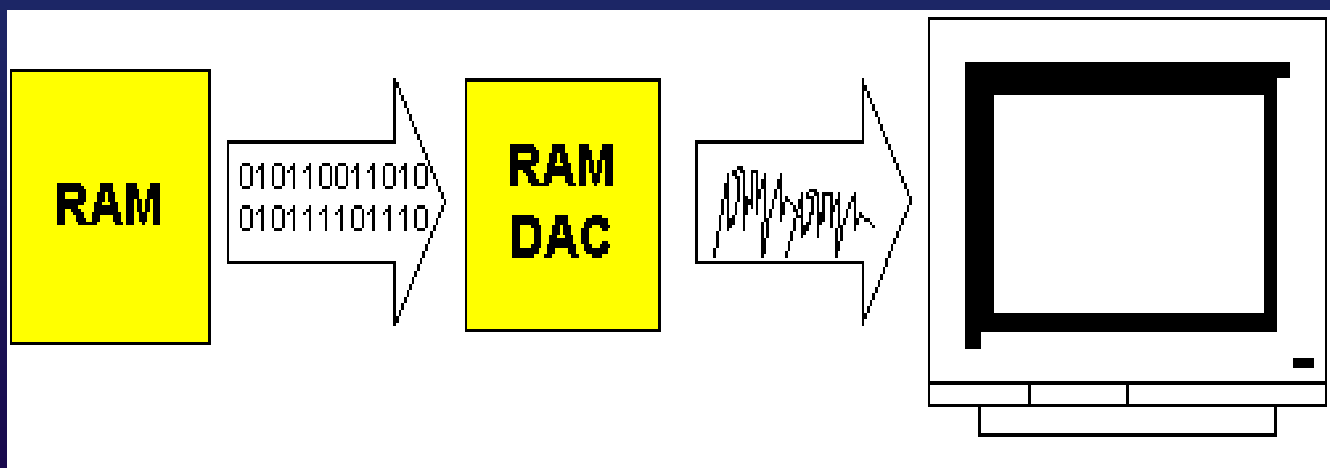
- All traditional graphics cards have a RAMDAC chip
- RAMDAC converts the signals from digital to analog form



The RAMDAC

The recommendation on a good RAMDAC go like this:

- External chip not integrated in the VGA chip
- Clock speed 250 - 360 MHz



Heavy Data Transport

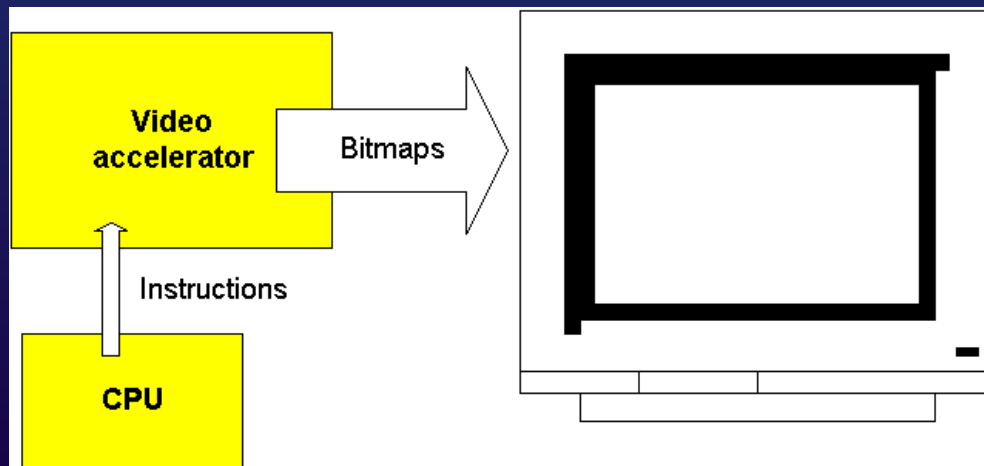
- The original VGA cards were unintelligent
- CPU had to make all necessary calculations to create the screen image
- Screen image of 1024*768 in 16 bit color is a 1.5 MB bit map
1024 x 768 x 2 bytes
refresh rate of 75 HZ
Zaps the PC energy
- Transfer went through the ISA bus (limited width)

Accelerator Cards

- Appeared in the early nineties
- Now all cards are accelerated and they are connected to the CPU through high speed buses like PCI and AGP.
- No need to calculate and design the entire bit map from image to image
- Built in algorithms for drawing lines, Windows, and other image elements

Accelerator Cards

- The AGP bus is an expanded and improved version of the PCI bus - used for video cards only
- Modern video cards made for 3D gaming use expensive high-end RAM to secure a sufficient bandwidth



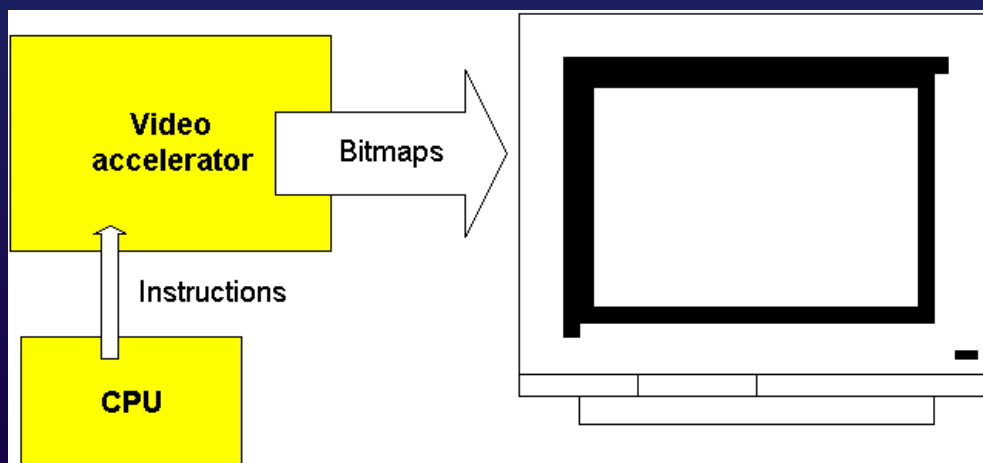
Accelerator Cards

A game in a resolution of 1280 x 1024 at 80 Hz may need to move 400 MB of data each second

The calculation goes like this:

$1280 \times 1024 \text{ pixels} \times 32 \text{ bit (color depth)} \times 80$

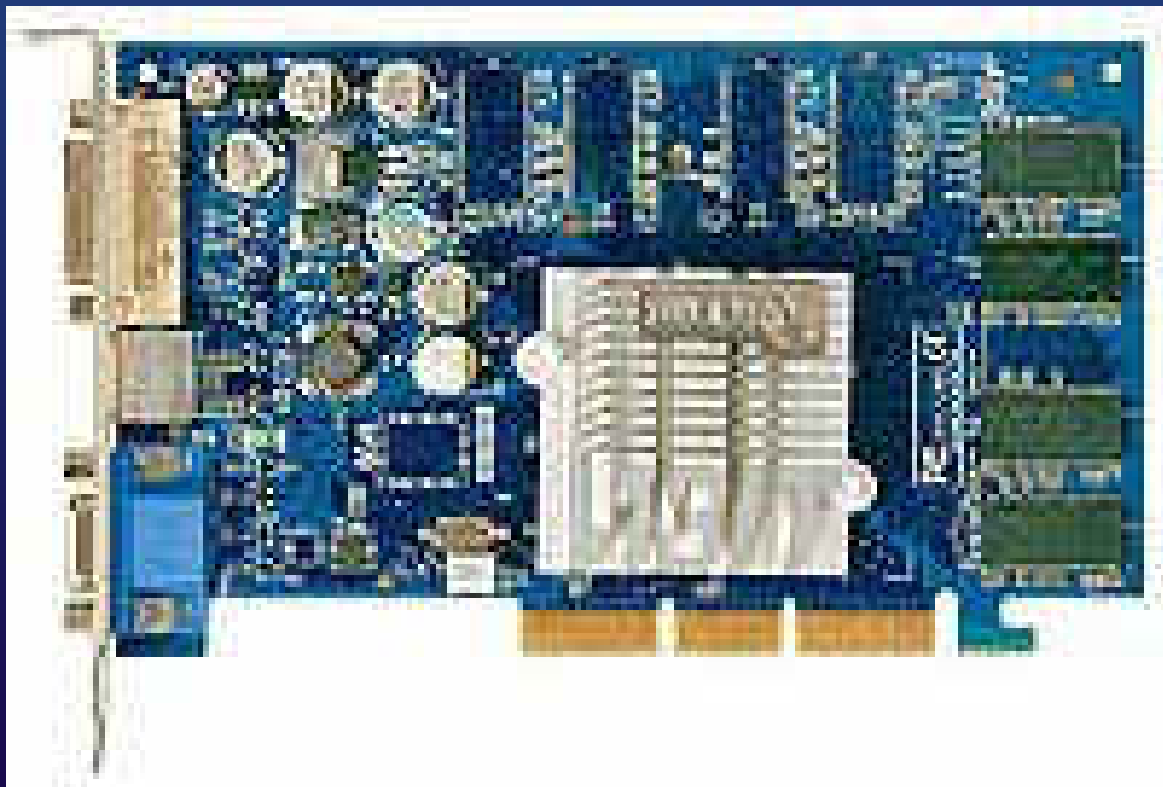
$= 419,430,400 \text{ bytes} = 409,600 \text{ kilobytes} = 400 \text{ megabytes.}$



Graphics Card



Graphics Card



Graphics Libraries

Graphics developers some time use 2D or 3D libraries to create graphics rapidly and efficiently. These developers include game developers, animators, designers etc.

The following libraries are commonly used among developers:

- FastGL
- OpenGL
- DirectX
- Others

Advantages of Graphics Libraries

These libraries help developers to create fast and optimized animations and also help to access features that are available on video hardware.

Hardware manufacturers give support in hardware for libraries

Famous manufacturers includes.. SIS, NVIDIA, ATI, INTEL etc.

Graphics Software

There are lot of 2D and 3D software are available in the market. These software provide visual interface for creation of 2D and 3D animation/ models, image creation. These tools are under use of movie makers professional animators and designers.

These tools are flash, maya, 3D studio max, adobe photo shop, corel draw, image viewer, paintbrush etc.

Computer Graphics

Lecture 03

Graphics Systems Cont...

Taqdees A. Siddiqi

taqdees@vu.edu.pk

Computer Graphics

Lecture 04

Point

Taqdees A. Siddiqi
taqdees@vu.edu.pk

Pixel

The smallest dot illuminated that can be seen on screen

Picture

Composition of pixels makes picture that forms on whole screen

Resolution

Resolution:

- Number of rows & number of columns

Each mode uses a particular resolution

For example:

- mode 19 uses 200 rows and 320 columns i.e. 320*200 resolution

Resolution

Higher resolution:

- sharper, clearer picture, with less pronounced 'staircase' effect on lines drawn diagonally and better looking text characters
- more memory requirement for the display

Text and Graphics Modes

- ◆ Video cards are responsible to send picture data
- ◆ Video cards support different text and graphics modes
- ◆ Every Mode has its own resolution, number of colors and refresh rate

Text and Graphics Modes

The following famous video modes that we can set in today's video cards on different refresh rates

- 25 * 80 with 16 colors support (text mode)
- 320 * 200 with 8 bit colors support (graphics mode)

Text and Graphics Modes

- 640 * 480 with 16 colors support (graphics mode)
- 640 * 480 with 8, 16, 24, 32 bit colors support (graphics mode)
- 800 * 600 with 8, 16, 24, 32 bit colors support (graphics mode)

Text and Graphics Modes Cont...

VDU

Amount of memory with respect to mode

Text and Graphics Modes Cont...

Mode No.	Type	Resolution	Memory Required
3	Text	80 x 25	2 bytes per char
6	Graphics	640 x 200	1 bit per pixel
7	Text	80 x 25	2 bytes per char
18	Graphics	640 x 480	1 bit per pixel
19	Graphics	320 x 200	1 byte per pixel

Mode 6

In mode 6 each pixel displayed on the screen occupies one bit in VDU memory. Since this bit can take only two values, either 0 or 1, each pixel can only be displayed in one of two colors.

How Text is Displayed

Text modes need two bytes in VDU memory to represent each character on screen

- ASCII value in one byte
- Attributes in second byte

How Text is Displayed

- Character information in VBIOS
- Character support in different types of adapters
- Characters in graphics mode

Text Mode Colors

- Character in mode 3 (two bytes in VDU memory)
 - one containing the ASCII value of the character
 - and other containing its attribute

Attribute Byte Configuration

Bits

7	6	5	4	3	2	1	0	Purpose
x	x	x	x	x	x	x	1	Blue component of foreground color
x	x	x	x	x	x	1	x	Green component of foreground color
x	x	x	x	x	1	x	x	Red component of foreground color
x	x	x	x	1	x	x	x	Intensity component of foreground color
x	x	x	1	x	x	x	x	Blue component of background color
x	x	1	x	x	x	x	x	Green component of background color
x	1	x	x	x	x	x	x	Red component of background color
1	x	x	x	x	x	x	x	Blinking component

Graphics Mode Colors

- In the graphics mode each pixel on the screen has a color associated with it
- Difference between text mode and graphics mode

Colors in VGA

- The VGA card was first introduced by IBM in April 1987
- VGA has 4 color planes:
 - red, green, blue and intensity
 - one bit from each of these planes contributing towards one pixel value

Writing Pixel on Screen

- Following methods can be used to write pixel on screen:
 - Using video bios services to write pixel
 - Accessing memory and registers directly to write pixel on screen.
 - Using library functions to write pixel on screen

Practical Approach to Write Pixel on Screen

The language used to write codes for graphics is Assembly and C

Writing Pixel Using Video BIOS

- The following are the steps involved in writing a pixel using video BIOS services:
 - Setting desired video mode
 - Using BIOS service to set color of a screen pixel
 - Calling BIOS interrupt to execute the process of writing pixel

Source Code

Code in assembly language that can set graphics mode 19 (13h)

```
MOV AH,0
MOV AL,13h ;mode number from 0-19
INT 10h
```

Source Code Cont..

You can use this for assembler or you can embed this code in C language using 'asm' keyword

```
asm {
    MOV AH,0
    MOV AL,13h;;mode number from 0-19
    INT 10h
}
```


Description of Source Code

- Line #1: mov AH,0 is the service number for setting video mode that is in register AH
- Line #2: mov AL,13h is the mode number that is in register AL
- Line #3: INT 10h is the video BIOS interrupt number that will set mode 13h

Source Code for Writing Pixel

Code to write pixel using video BIOS interrupt 10h and service number 0ch.

```
MOV AH,0Ch
MOV AL,COLOR_NUM
MOV BH,0
MOV CX,ROW_NUM
MOV DX,COLUMN_NUM
INT 10h
```

Description

- Line#1: service number in register AH
- Line#2: color value, since it is 13h mode so it has 0-255 colors range. You can assign any color number between 0 to 255 to AL register. Color will be selected from default palette setting against the number you have used
- Line#3: page number in BH register. This mode supports only one page. So 0 is used in BH register. 0 means default page

Description

- Line#4: column number in CX register
- Line#5: row number in DX register
- Line#6: BIOS interrupt number 10h

Writing pixel by accessing Video Memory directly

- So far we used BIOS to draw pixel.
- Now we will draw pixel by accessing direct pointer to the video memory and write color value

Writing pixel by accessing Video Memory directly

- Steps to write pixel directly (without using BIOS):
 - Set video mode by using video BIOS routine as discussed earlier
 - Set any pointer to the video graphics memory address 0x0A000
 - Now write any color value in the video memory address

Direct Graphics Memory Access

Example:

```
Mov ax,0a000h
Mov ds,ax      ;;segment address changed
Mov si,10      ;; column number
Mov [si],COLOR_NUM
```

Direct Graphics Memory Access

Work to do:

- Write pixel at 12th row and 15th column
- Hint: use formula (row * 320 + column) in si register

Writing Character Directly on Screen

You can also write text directly:

- by setting any text mode using BIOS service
- and then setting direct pointer at text memory address 0x0b800

Example Writing Character Directly on Screen

Set mode Number 3 using BIOS service and then use this code to write character

```
Mov ax,0b800h
Mov ds,ax
Mov si,10      ;;column number
Mov [si],'a'   ;;character to write
```

Using Library Functions

- While working in C language, you can use graphics library functions to write pixel on screen
- Graphics library functions use BIOS routines or use direct memory access drivers to draw pixel on screen

Using Library Functions

```
initgraph(&gdriver, &gmode, "");  
/* read result of initialization */  
errorcode = graphresult();  
if (errorcode != grOk)  
/* an error occurred */  
{  
  
    printf("Graphics error: %s\n", getch());  
    exit(1);  
/* return with error code */  
}
```

Using Library Functions

```
/* draw a pixel on 10th row and 10 column */  
putpixel(10, 10, BLUE);  
/* clean up */  
closegraph();
```

Steps, Example in C language

- First call Initgraph() function
- then call putpixel() function to draw pixel on screen
- then use closegraph() function

Discussion on Pixel drawing Methods

- BIOS routines are quite slow
- Direct memory access method is easy and faster
 - its programming is only convenient in mode 13h
- Library functions are even faster

Drawing Pixel in Microsoft Windows

- how to write pixel in Microsoft Windows
- Microsoft windows provides library functions (APIs)
- Knowledge about Windows GDI (graphics device interface) system

Windows GDI Functions

- SetPixel and SetPixelV
- Both are used to draw pixel on screen
- Sample code is provided in the notes

Computer Graphics Lecture 04 Point

Taqdees A. Siddiqi
taqdees@vu.edu.pk

Computer Graphics

Lecture 05

Line Drawing Techniques

Taqdees A. Siddiqi
taqdees@vu.edu.pk

What is Line

A line, or straight line, is, roughly speaking, (infinitely) thin, (infinitely) long, straight geometrical object, i.e. a curve that is long and straight.

Line Cont...

Given two points, in Euclidean geometry, one can always find exactly one line that passes through the two points.

Line Cont...

A line may have three forms with respect to slope:

- slope = 1
- slope > 1
- slope < 1

Line Cont...

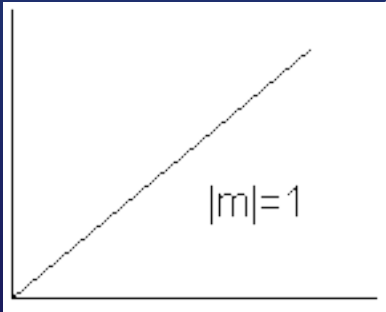


figure (a)

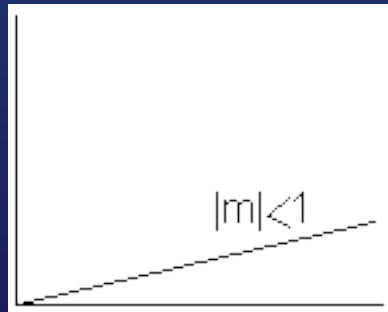


figure (b)

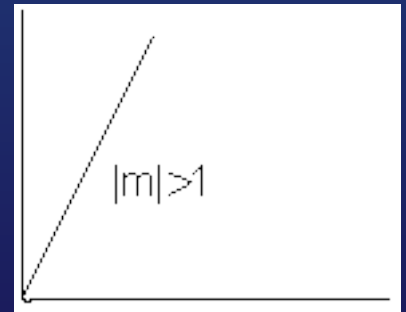


figure (c)

Line Drawing Techniques

There are three techniques to be discussed to draw a line involving different time complexities that will be discussed along. These techniques are:

- Incremental line algorithm
- DDA line algorithm
- Bresenham line algorithm

Incremental Line Algorithm

This algorithm exploits simple line equation

$$y = m x + b$$

$$\text{where } m = dy / dx$$

$$\text{and } b = y - m x$$

now check if

$$|m| < 1 \text{ then}$$

$$x = x + 1 \quad \text{whereas}$$

$$y = m x + b$$

why to check $|m|$

suppose a line has points

$$p_1 (10, 10) - p_2 (20, 18)$$

$$dy = y_2 - y_1 = 18 - 10 = 8$$

$$dx = x_2 - x_1 = 20 - 10 = 10$$

This means that there will be 10 pixels on the line in which for x-axis there will be distance of 1 between each pixel and for y-axis the distance will be 0.8.

Now consider the reverse case

suppose a line has points

$$p_1 (10, 10) , p_2 (16, 20)$$

$$dy = y_2 - y_1 = 20 - 10 = 10$$

$$dx = x_2 - x_1 = 16 - 10 = 6$$

This means that there will be 10 pixels on the line in which for x-axis there will be distance of 0.6 between each pixel and for y-axis the distance will be 1.

Now sum-up all discussion in algorithm to fully understand.

The algorithm will take two points P1 and P2 and draw line between them whereas each point consists of x,y coordinates.

$$dx = p2.x - p1.x$$

$$dy = p2.y - p1.y$$

$$m = dy / dx$$

$$x = p1.x$$

$$y = p1.y$$

$$b = y - m * x$$

```
if  $|m| < 1$ 
  for counter = p1.x to p2.x
    drawPixel (x, y)
     $x = x + 1$ 
     $y = m * x + b$ 
else
  for counter = p1.y to p2.y
    drawPixel (x, y)
     $y = y + 1$ 
     $x = (y - b) / m$ 
```

Discussion on algorithm:

- ◆ quite simple and easy to understand
- ◆ but involves a lot of mathematical calculations 24

We have another algorithm that works fine in all directions and involves less calculations; mostly only additions.

DDA Algorithm

DDA abbreviated for digital differential analyzer has a very simple technique.

Find difference, dx and dy as:

$$dy = y_2 - y_1$$

$$dx = x_2 - x_1$$

```
if  $|dx| > |dy|$  then  
    step =  $|dx|$   
else  
    step =  $|dy|$ 
```

Now very simple to say that step is the total number of pixels required for a line.

Next step is to find xIncrement and yIncrement:

$$xIncrement = dx/step$$

$$yIncrement = dy/step$$

Next a loop is required that will run 'step' times.

In the loop drawPixel and add xIncrement to x1 and yIncrement to y1.

Now sum-up all above in the algorithm:

DDA_Line (Point p1, Point p2)

$dx = p2.x - p1.x$

$dy = p2.y - p1.y$

$x1 = p1.x$

$y1 = p1.y$

if $|dx| > |dy|$ then

$step = |dx|$

else

$step = |dy|$

```
xIncrement = dx/step  
yIncrement = dy/step  
for counter = 1 to step  
    drawPixel (x1, y1)  
    x1 = x1 + xIncrement  
    y1 = y1 + yIncrement
```

Criticism on Algorithm:

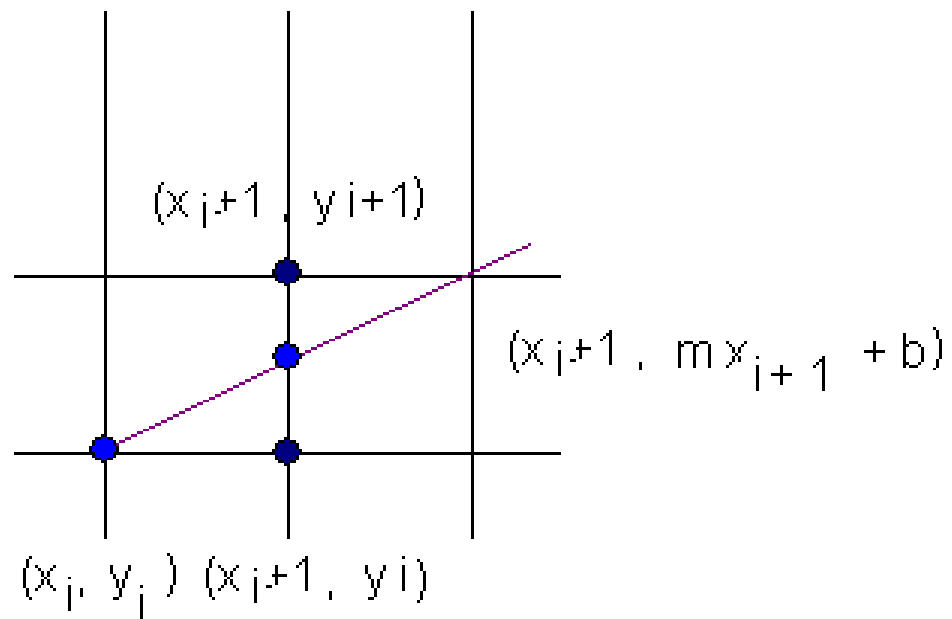
Use of floating point calculation

An algorithm based on integer type calculations is likely to be more efficient

Therefore, after some effort, finally we came up with an algorithm called “Bresenham Line Drawing Algorithm” which would be discussed next.

Bresenham Algorithm

Bresenham's algorithm finds the closest integer coordinates to the actual line, using only integer math. Assuming that the slope is positive and less than 1, moving 1 step in the x direction, y either stays the same, or increases by 1. A decision function is required to resolve this choice.



$$\begin{aligned} d_1 &= y - y_i \\ &= m * (x_{i+1}) + b - y_i \end{aligned}$$

$$\begin{aligned} d_2 &= y_{i+1} - y \\ &= y_{i+1} - m (x_{i+1}) - b \end{aligned}$$

$$p_i = d\mathbf{x} (d_1 - d_2)$$

$$p_i = d\mathbf{x} (2m * (x_i + 1) + 2b - 2y_i - 1)$$

$$p_i = 2dy (x_i + 1) - 2dx y_i + dx (2b - 1) \text{ ----- (i)}$$

$$p_i = 2 dy x_i - 2 dx y_i + k \text{ ----- (ii)}$$

$$\text{where } k = 2 dy + dx (2b - 1)$$

Then we can calculate p_{i+1} in terms of p_i without any x_i , y_i or k .

$$p_{i+1} = 2 dy x_{i+1} - 2 dx y_{i+1} + k$$

$$p_{i+1} = 2 dy (x_i + 1) - 2 dx y_{i+1} + k \quad \text{since } x_{i+1} = x_i + 1$$

$$p_{i+1} = 2 dy x_i + 2 dy - 2 dx y_{i+1} + k \text{ --- (iii)}$$

Now subtracting (ii) from (iii), we get

$$\begin{aligned} p_{i+1} - p_i &= 2 \, dy - 2 \, dx \, (y_{i+1} - y_i) \\ p_{i+1} &= p_i + 2 \, dy - 2 \, dx \, (y_{i+1} - y_i) \end{aligned}$$

If the next point is: (x_i+1, y_i) then

$$d_1 < d_2 \quad \Rightarrow \quad d_1 - d_2 < 0$$

$$\Rightarrow p_i < 0$$

$$\Rightarrow p_{i+1} = p_i + 2 \, dy$$

If the next point is: (x_i+1, y_i+1) then

$$d_1 > d_2 \quad \Rightarrow \quad d_1 - d_2 > 0$$

$$\Rightarrow \quad p_i > 0$$

$$\Rightarrow \quad \mathbf{p_{i+1} = p_i + 2 \, dy - 2 \, dx}$$

The p_i is our decision variable, and calculated using integer arithmetic from pre-computed constants and its previous value. Now a question is remaining; i.e. how to calculate initial value of p_i ? For that, we use equation (i) and put values (x_1, y_1)

$$p_i = 2 \, dy \, (x_1+1) - 2 \, dx \, y_1 + dx \, (2b - 1)$$

where $b = y - m \, x$ implies that

$$p_i = 2 \, dy \, x_1 + 2 \, dy - 2 \, dx \, y_1 + dx \, (2 \, (y_1 - mx_1) - 1)$$

$$p_i = 2 \, dy \, x_1 + 2 \, dy - 2 \, dx \, y_1 + 2 \, dx \, y_1 - 2 \, dy \, x_1 - dx$$

$$p_i = 2 \, dy \, x_1 + 2 \, dy - 2 \, dx \, y_1 + 2 \, dx \, y_1 - 2 \, dy \, x_1 - dx$$

there are certain figures that will cancel each other (shown in pairs of different colour)

$$p_i = 2 \, dy - dx$$

```
dx = x2-x1  
dy = y2-y1  
p = 2dy-dx  
c1 = 2dy  
c2 = 2(dy-dx)  
x = x1  
y = y1  
plot (x, y, colour)
```

```
while (x < x2)  
    x++  
    if (p < 0)  
        p = p + c1  
    else  
        p = p + c2  
    y++  
    plot (x,y,colour)
```

Improving Performance

The use of separate x and y coordinates can be discarded in favour of direct frame buffer addressing. Most algorithms can be adapted to calculate only the initial frame buffer address corresponding to the starting point and to replace:

x++ with Addr++

y++ with Addr+=XResolution

Fixed point representation allows a method for performing calculations using only integer arithmetic, but still obtaining the accuracy of floating point values. In fixed point, the fraction part of a value is stored separately, in another integer:

M = Mint.Mfrac

Mint = Int(M)

Mfrac = Frac(M) × MaxInt

Addition in fixed point representation occurs by adding fractional and integer components separately, and only transferring any carry-over from the fractional result to the integer result. The sequence could be implemented using the following two integer additions: ADD Yfrac,Mfrac ; ADC Yint,Mint

Improved versions of these algorithms exist. For example the following variations exist on Bresenham's original algorithm:

Symmetry (forward and backward simultaneously)

Segmentation (divide into smaller identical segments - $\text{GCD}(D_x, D_y)$)

Double step, triple step, n step

Setting a Pixel

Initial Task: Turning on a pixel (loading the frame buffer/bit-map). Assume the simplest case, i.e., an 8-bit, non-interlaced graphics system. Then each byte in the frame buffer corresponds to a pixel in the output display.

Y_m



$0,0$

X_m

$\text{addr}(0,0)$ = the memory address of the initial pixel (0,0)

Number of rows = number of raster lines.

Number of columns = number of pixels/raster line.

To find the address of a particular pixel (X,Y) we use the following formula:

$$\begin{aligned} \text{addr}(X, Y) = & \text{addr}(0,0) \\ & + Y \text{ rows} * (X_m + 1) \\ & + X \quad \quad \quad (\text{all in bytes}) \end{aligned}$$

$\text{addr}(X,Y)$ = the memory address of pixel (X,Y)

Example:

For a system with 640×480 pixel resolution, find the address of pixel

($X = 340$, $Y = 150$)

$$\text{addr}(340, 150) = \text{addr}(0,0) + 150 * 640 + 340$$

= base + 96,340 is the byte location

Graphics systems usually have a command such as `set_pixel(x, y)` where x, y are integers.

Lets call it a day!

Computer Graphics

Lecture 05

Line Drawing Techniques

Taqdees A. Siddiqi
taqdees@vu.edu.pk

Computer Graphics

Lecture 06

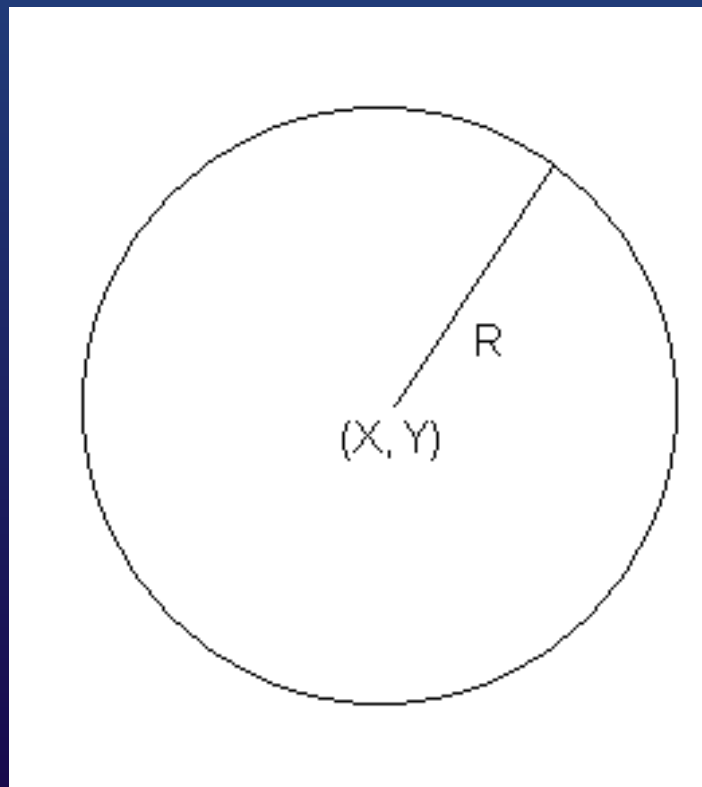
Circle Drawing Techniques

Taqdees A. Siddiqi
cs602@vu.edu.pk

What is Circle

A circle is the set of points in a plane that are equidistant from a given point O. The distance r from the center is called the radius, and the point O is called the center. Twice the radius is known as the diameter. The angle a circle subtends from its center is a full angle, equal to 360° or 2π radians.

Circle Cont..



Circle Cont..

A circle has the maximum possible area for a given perimeter, and the minimum possible perimeter for a given area.

The perimeter C of a circle is called the circumference, and is given by

$$C = 2 \pi r$$

Circle Drawing Techniques

- Input for circle drawing
 - . one center point (x_c, y_c) and
 - . radius r

Now, using these two inputs there are a number of ways to draw a circle.

Circle Drawing Techniques

These techniques have:

- Understanding curve
 - . very simple to
 - . complex
- Time complexity
 - . inefficient to
 - . efficient

Circle Drawing Using Cartesian Coordinates

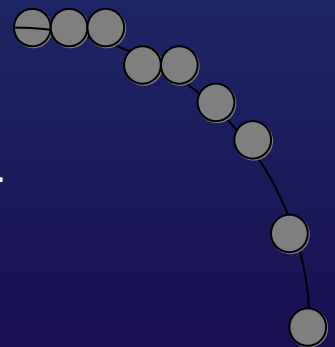
This technique uses the equation for a circle with radius r centered at $(0,0)$:

$$x^2 + y^2 = r^2,$$

an obvious choice is to plot

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

against different values of x .



Circle Drawing Using Cartesian Coordinates

Using above equation a circle can be easily drawn. The value of x varies from $r-x_c$ to $r+x_c$, and y is calculated using above formula. Using this technique a simple algorithm will be:

Circle Drawing Using Cartesian Coordinates

```
for x= radius-xcenter to radius+xcenter
     $y = y_c + \sqrt{r^2 - (x - x_c)^2}$ 
    drawPixel (x, y)
     $y = y_c - \sqrt{r^2 - (x - x_c)^2}$ 
    drawPixel (x, y)
```

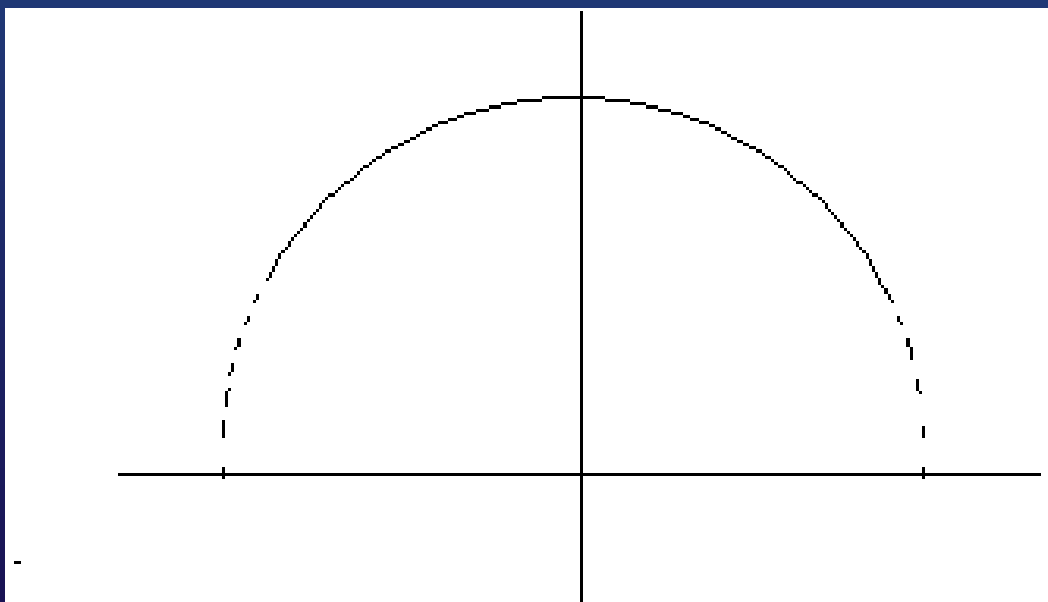

Drawbacks/ Shortcomings

This works, but ...

- . is inefficient
- . multiplications & square root
- . large gaps in the circle for values of x close to r

(as shown in the next figure)

Drawbacks/ Shortcomings



Circle Drawing Using Polar Coordinates

Polar Coordinates:

Radius r

Angle θ

calculate points along the circular boundary

using polar coordinates r and θ

Expressing the circle equation in parametric polar form yields the pair of equations:

Circle Drawing Using Polar Coordinates Cont...

$$x = x_c + r \cos \theta$$

$$y = y_c + r \sin \theta$$

Using above equation circle can be plotted by calculating x and y coordinates as θ takes values from 0 to 360 degrees or 0 to 2π radians.

Circle Drawing Using Polar Coordinates Cont...

The step size for θ depends on:

- . application and
- . display device

Larger angular separations along the circumference can be connected with straight-line segments to approximate the circular path.

Step size at $1/r$ gives continuous boundary

This plots pixel positions that are approximately one unit apart.

Polar Coordinates Algorithm

Circle2 (xcenter, ycenter, radius)

for $\theta = 0$ to 2π step $1/r$

$$x = x_c + r * \cos \theta$$

$$y = y_c + r * \sin \theta$$

drawPixel (x, y)

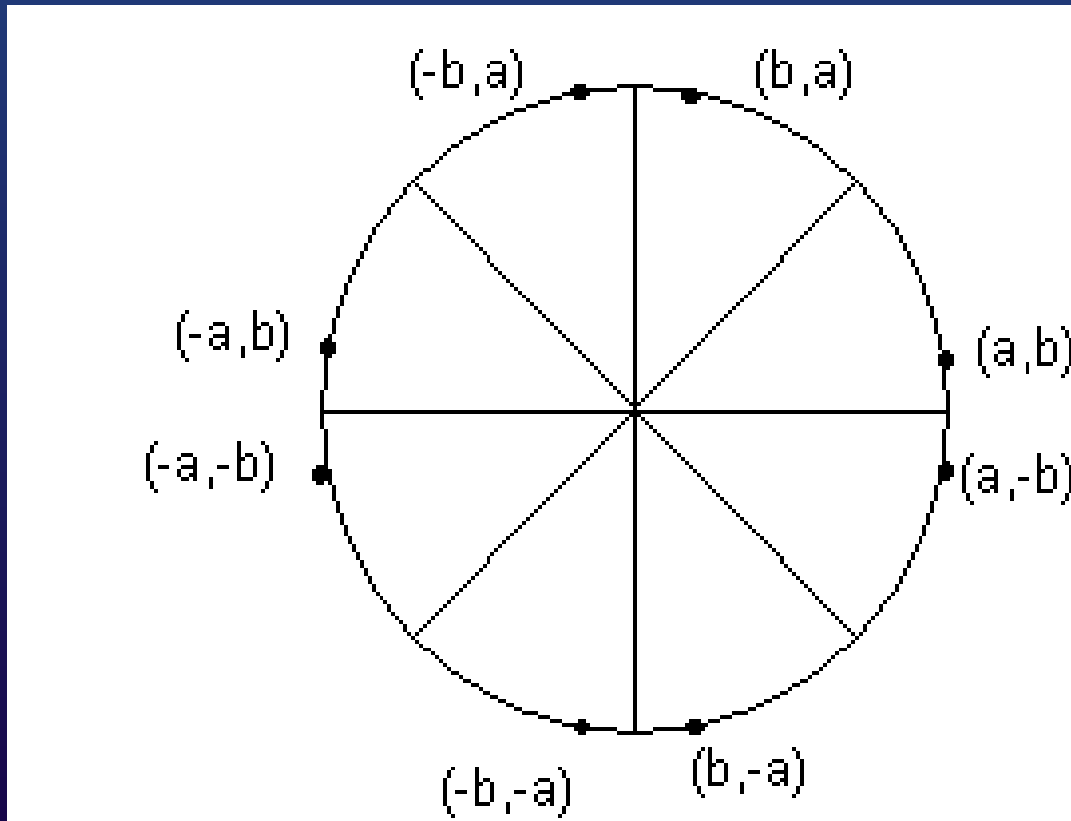
Discussion

- very simple technique
- solves problem of unequal space
- is inefficient in terms of calculations
- involves floating point calculations

Reduction in Calculations

- Symmetry in octants
- Upper half circle symmetric to lower half circle
- Left half circle symmetric to right half circle
- Finally two halves of the same quarters are symmetric to each other

Eight Octants Symmetry



Optimizing the Algorithm

Now this algorithm can be optimized by using symmetric octants as:

Circle2 (xcenter, ycenter, radius)

for $\theta = 0$ to $\pi/4$ step $1/r$

$x = x_c + r * \cos \theta$

$y = y_c + r * \sin \theta$

DrawSymmetricPoints(xcenter, ycenter, x,y)

Optimized Algorithm

DrawSymmetricPoints (xcenter, ycenter, x, y)

Plot ($x + xcenter$, $y + ycenter$)

Plot ($y + xcenter$, $x + ycenter$)

Plot ($y + xcenter$, $-x + ycenter$)

Plot ($x + xcenter$, $-y + ycenter$)

Plot ($-x + xcenter$, $-y + ycenter$)

Plot ($-y + xcenter$, $-x + ycenter$)

Plot ($-y + xcenter$, $x + ycenter$)

Plot ($-x + xcenter$, $y + ycenter$)

Inefficiency Still Prevails

- Reduction in calculations by exploiting symmetric octants but...
- Floating point calculations still involved

Midpoint Circle Algorithm

- Derivation of decision parameter
- Decrement decision in the y coordinate against increment of x coordinate

Midpoint Circle Algorithm

Consider only the first octant of a circle of radius r centered on the origin.

We begin by plotting point $(0, r)$ and end when $x = y$.

Midpoint Circle Algorithm

The decision at each step is whether to choose :
the pixel to the right of the current pixel

or

the pixel which is to the right and below the current pixel
(8-way stepping)

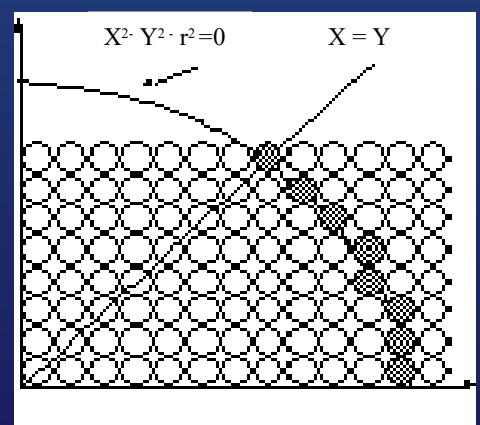
Midpoint Circle Algorithm

Assume:

$P = (x_k, y_k)$ is the current pixel.

$Q = (x_k+1, y_k)$ is the pixel to the right

$R = (x_k+1, y_k -1)$ is the pixel to the right and below.



Midpoint Circle Algorithm

To apply the midpoint method, we define a circle function:

$$f_{\text{circle}}(x, y) = x^2 + y^2 - r^2$$

The following relations can be observed:

$f_{\text{circle}}(x, y) < 0$, if (x, y) is inside the circle boundary

$f_{\text{circle}}(x, y) = 0$, if (x, y) is on the circle boundary

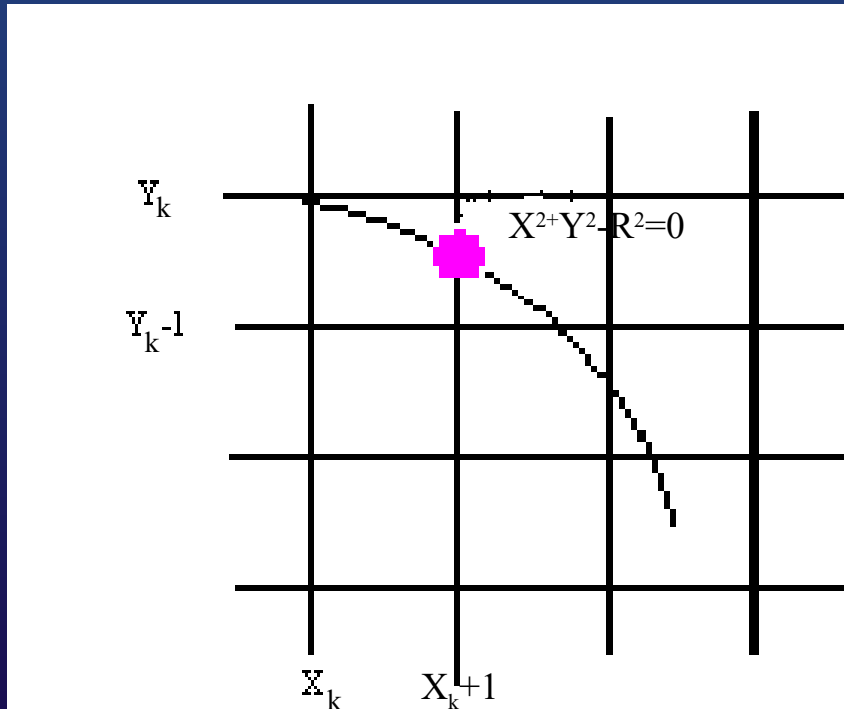
$f_{\text{circle}}(x, y) > 0$, if (x, y) is outside the circle

boundary

Midpoint Circle Algorithm

Circle function tests are performed for the midpoints between pixels near the circle path at each sampling step.

Midpoint Circle Algorithm



Midpoint Circle Algorithm

Figure given in previous slide shows the midpoint between the two candidate pixels at sampling position x_k+1 .

Our decision parameter is the circle function evaluated at the midpoint between these two pixels:

Decision Parameter

$$P_k = f_{\text{circle}}(x_k + 1, y_k - \frac{1}{2})$$

$$P_k = (x_k + 1)^2 + (y_k - \frac{1}{2})^2 - r^2 \dots\dots\dots(1)$$

Decision Parameter

If $p_k < 0$, this midpoint is inside the circle and the pixel on scan line y_k is closer to the circle boundary. Otherwise, the mid position is outside or on the circle boundary, and we select the pixel on scan-line y_{k-1} .

Decision Parameter

Successive decision parameters are obtained using incremental calculations. We obtain a recursive expression for the next decision parameter by evaluating the circle function at sampling position $x_{k+1}+1 = x_k+2$:

Decision Parameter

$$P_{k+1} = f_{\text{circle}}(x_{k+1} + 1, y_{k+1} - \frac{1}{2})$$

$$P_{k+1} = [(x_k + 1) + 1]^2 + (y_{k+1} - \frac{1}{2})^2 - r^2 \quad \dots\dots\dots(2)$$

Decision Parameter

Subtracting (1) from (2), we get:

$$P_{k+1} - P_k = [(x_k + 1) + 1]^2 + (y_{k+1} - \frac{1}{2})^2 - r^2 - (x_k + 1)^2 - (y_k - \frac{1}{2})^2 + r^2$$

or

$$P_{k+1} = P_k + 2(x_k + 1) + (y_{k+1}^2 - y_k^2) - (y_{k+1} - y_k) + 1$$

Decision Parameter

Where y_{k+1} is either y_k or $y_k - 1$, depending on the sign of P_k .
Therefore, if $P_k < 0$ or negative then y_{k+1} will be y_k and the formula to calculate P_{k+1} will be:

Decision Parameter

$$P_{k+1} = P_k + 2(x_k + 1) + (y_k^2 - y_{k-1}^2) - (y_k - y_{k-1}) + 1$$

$$P_{k+1} = P_k + 2(x_k + 1) + 1$$

Decision Parameter

Otherwise, if $P_k > 0$ or positive then y_{k+1} will be y_{k-1} and the formula to calculate P_{k+1} will be:

Decision Parameter

$$P_{k+1} = P_k + 2(x_k + 1) + [(y_k - 1)^2 - y_k^2] \\ - (y_k - 1 - y_k) + 1$$

$$P_{k+1} = P_k + 2(x_k + 1) + (y_k^2 - 2y_k + 1 - y_k^2) \\ - (y_k - 1 - y_k) + 1$$

$$P_{k+1} = P_k + 2(x_k + 1) - 2y_k + 1 + 1 + 1$$

$$P_{k+1} = P_k + 2(x_k + 1) - 2y_k + 2 + 1$$

$$P_{k+1} = P_k + 2(x_k + 1) - 2(y_k - 1) + 1$$

Decision Parameter

Now a similar case that we observed in line algorithm is that how would starting P_k be calculated.

For this, the starting pixel position will be $(0, r)$.

Therefore, putting this value in equation, we get

Decision Parameter

$$P_0 = (0 + 1)^2 + (r - \frac{1}{2})^2 - r^2$$

$$P_0 = 1 + r^2 - r + \frac{1}{4} - r^2$$

$$P_0 = \frac{5}{4} - r$$

If radius r is specified as an integer, we can simply round p_0 to:

$$P_0 = 1 - r$$

Since all increments are integer. Finally the algorithm can be summed up as :

Algorithm

MidpointCircle (xcenter, ycenter, radius)

$x = 0;$

$y = r;$

$p = 1 - r;$

do

DrawSymmetricPoints (xcenter, ycenter, x , y)

$x = x + 1$

If $p < 0$ Then

$p = p + 2 * (y + 1) + 1$

Algorithm Cont...

else

$$y = y - 1$$

$$p = p + 2 * (y + 1) - 2 * (x - 1) + 1$$

while ($x < y$)

Example to calculate first octant of the circle using above algorithm.

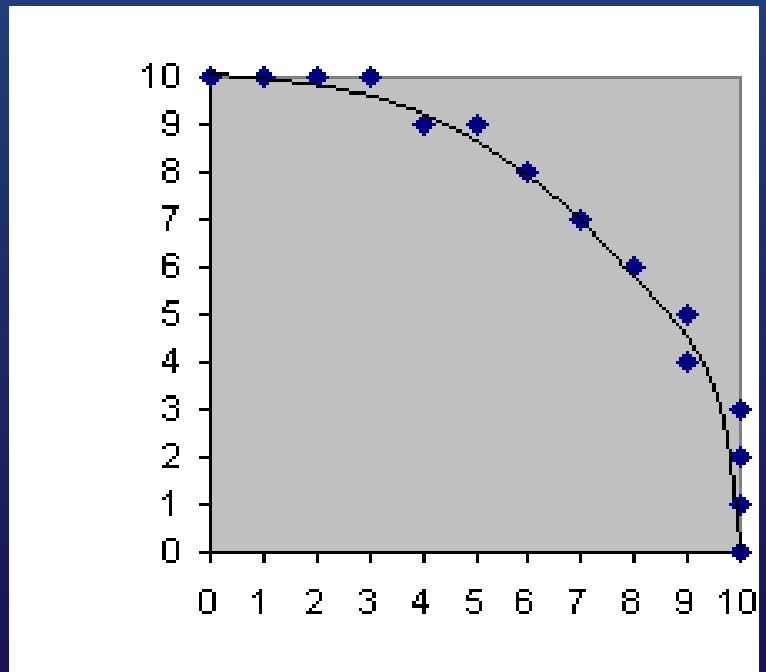
Example

center (0, 0) radius = 10

p	x	y
-9	0	10
-6	1	10
-1	2	10
6	3	10
-3	4	9
8	5	9
5	6	8
6	7	7

Example Cont...

Behaviour of
calculated points
around the circle is
observable



Computer Graphics

Lecture 06

Circle Drawing Techniques

Taqdees A. Siddiqi
cs602@vu.edu.pk

Computer Graphics

Lecture 07

Ellipse and Other Curves

Taqdees A. Siddiqi
cs602@vu.edu.pk

Ellipse

An ellipse is a curve that is the locus of all points in the plane, the sum of whose distances r_1 and r_2 from two fixed points F_1 and F_2 (the foci) separated by a distance of $2c$ is a given positive constant $2a$. This results in the two-center bipolar coordinate equation:

$$r_1 + r_2 = 2a$$

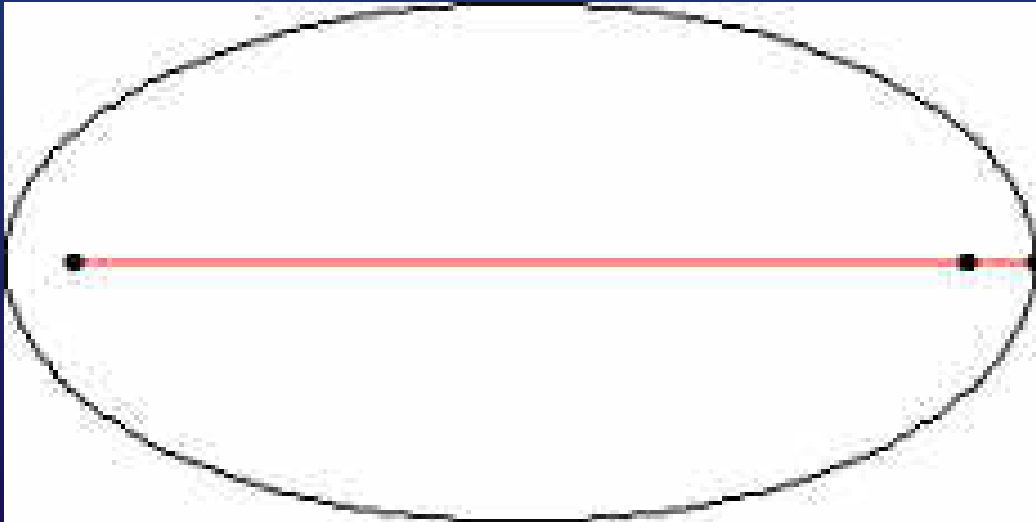
Ellipse Cont..



Ellipse Cont..

where a is the semi-major axis and the origin of the coordinate system is at one of the foci. The corresponding parameter b is known as the semi-minor axis.

Ellipse Cont..



Ellipse Cont..

Let an ellipse lie along the x -axis and find the equation of the figure given above where F_1 and F_2 are at $(-c, 0)$ and $(c, 0)$. In Cartesian coordinates,

Bring the second term to the right side and square both sides,

$$\sqrt{(x + c)^2 + y^2} + \sqrt{(x - c)^2 + y^2} = 2a.$$

$$(x + c)^2 + y^2 = 4a^2 - 4a\sqrt{(x - c)^2 + y^2} + (x - c)^2 + y^2.$$

Ellipse Cont..

Now solve for the square root term and simplify

$$\begin{aligned}\sqrt{(x-c)^2 + y^2} &= -\frac{1}{4a}(x^2 + 2xc + c^2 + y^2 - 4a^2 - x^2 + 2xc - c^2 - y^2) \\ &= -\frac{1}{4a}(4xc - 4a^2) = a - \frac{c}{a}x.\end{aligned}$$

Square one final time to clear the remaining square root,

$$x^2 - 2xc + c^2 + y^2 = a^2 - 2cx + \frac{c^2}{a^2}x^2.$$

Ellipse Cont..

Grouping the x terms then gives

$$x^2 \frac{a^2 - c^2}{a^2} + y^2 = a^2 - c^2,$$

which can be written in the simple form

$$\frac{x^2}{a^2} + \frac{y^2}{a^2 - c^2} = 1.$$

defining a new constant

$$b^2 \equiv a^2 - c^2$$

Ellipse Cont..

puts the equation in the particularly simple form

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1.$$

Parameter b (semi-minor axis)

Parameter a (semi-major axis)

Ellipse Cont..

If, instead of being centered at (0, 0), the center of the ellipse is at (x_0, y_0) , above equation becomes:

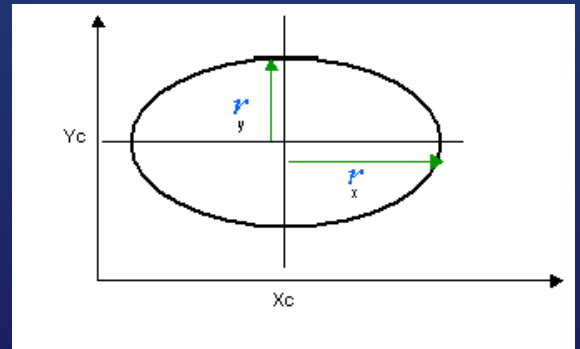
$$\frac{(x - x_0)^2}{a^2} + \frac{(y - y_0)^2}{b^2} = 1.$$

Ellipse Drawing Techniques

Input for drawing Ellipse:
one center point,
radius rx and
radius ry

Now, using these three inputs there are a number of ways to draw an ellipse.

Keeping in view that you already understand circle drawing techniques, One way to draw ellipse is to use the following equation:



$$\frac{(x - x_0)^2}{a^2} + \frac{(y - y_0)^2}{b^2} = 1.$$

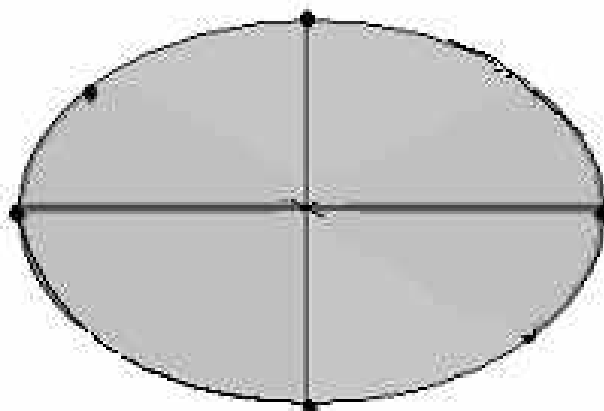
Ellipse Drawing Using Polar Coordinates

Another way is to use polar coordinates r and θ , for that we have parametric equations:

$$x = x_c + r_x \cos \theta$$

$$y = y_c + r_y \sin \theta$$

Four-way Symmetry



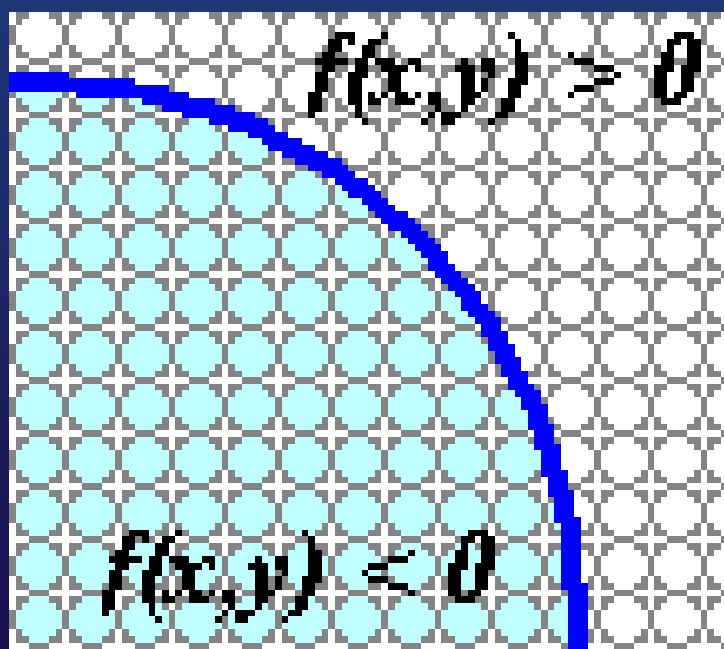
Midpoint Ellipse Algorithm

Consider an ellipse centered at the origin:

To apply the midpoint method, we define an ellipse function:

$$f_{\text{ellipse}}(x, y) = r_y^2 x^2 + r_x^2 y^2 - r_x^2 r_y^2$$

$$\frac{(x - x_0)^2}{a^2} + \frac{(y - y_0)^2}{b^2} = 1$$



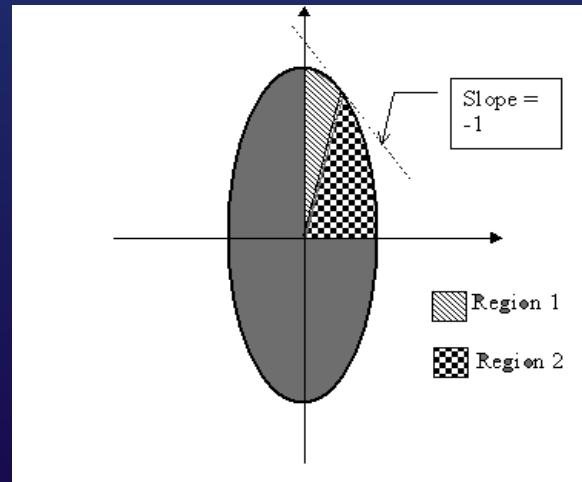
To apply the midpoint method, we define an ellipse function:

$$f_{\text{ellipse}}(x, y) = r_y^2 x^2 + r_x^2 y^2 - r_x^2 r_y^2$$

The following relations can be observed:

$$\begin{aligned} f_{\text{ellipse}}(x, y) &< 0, && \text{if } (x, y) \text{ is inside the ellipse boundary} \\ f_{\text{ellipse}}(x, y) &= 0, && \text{if } (x, y) \text{ is on the ellipse boundary} \\ f_{\text{ellipse}}(x, y) &> 0, && \text{if } (x, y) \text{ is outside the ellipse boundary} \end{aligned}$$

Two regions in one octant.



Region 1: sampling will be in x direction; whereas y coordinate will be related to decision parameter.

Region 2 sampling will be in y direction; whereas x coordinate will be related to decision parameter.

Consider first region 1:

start at $(0, r_y)$

At each step, we need to test the value of the slope of the curve. The ellipse slope is calculated from the following equation:

$$\frac{d_y}{d_x} = - \frac{2 r_y^2 x^2}{2 r_x^2 y^2}$$

At the boundary of region 1 and region 2, $dy / dx = -1$ and

$2 r_x^2 y^2 = 2 r_y^2 x^2$, Therefore, we move out of region 1 whenever

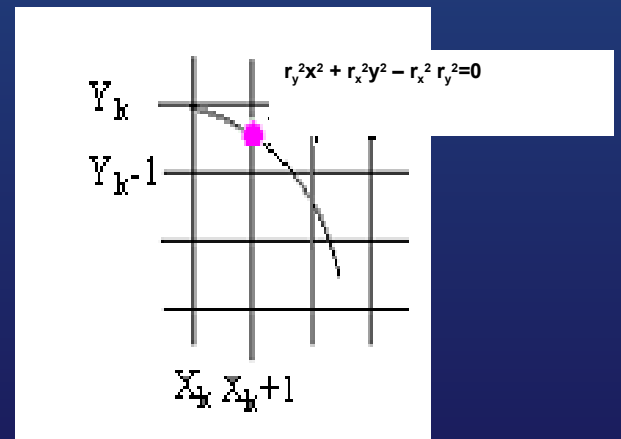
$$r_y^2 x^2 + r_x^2 y^2 - r_x^2 r_y^2 = 0$$

$$2 r_y^2 x^2 \geq 2 r_x^2 y^2$$

In the 1st region, two candidate pixels for next value of x i.e. $x_k + 1$, are :

$$(x_k + 1, y_k)$$

$$(x_k + 1, y_k - 1)$$



$$P1_k = f_{\text{ellipse}}(x_k + 1, y_k - \frac{1}{2})$$

$$f_{\text{ellipse}}(x_k + 1, y_k - \frac{1}{2}) = r_y^2 (x_k + 1)^2 + r_x^2 (y_k - \frac{1}{2})^2 - r_x^2 r_y^2 \text{ ————— } (1)$$

If $p1_k < 0$, point is inside the ellipse and the pixel on scan line y_k is closer to the ellipse boundary. Otherwise, the mid position is outside and we select the pixel on scan-line $y_k - 1$.

Successive decision parameters are obtained using incremental calculations. We obtain a recursive expression for the next decision parameter by evaluating the ellipse function at sampling position $x_{k+1}+1=x_k+1+1=x_k+2$:

$$f_{\text{ellipse}}(x_k+1+1, y_{k+1} - \frac{1}{2}) = r_y^2 [(x_k + 1) + 1]^2 + r_x^2 (y_{k+1} - \frac{1}{2})^2 - r_x^2 r_y^2 \quad (2)$$

Subtracting (1) from (2), and by simplification, we get:

$$\begin{aligned} P1_{k+1} = P1_k &+ 2r_y^2(x_k+1) \\ &+ r_x^2(y_{k+1}^2 - y_k^2) \\ &- r_x^2(y_{k+1} - y_k) + r_y^2 \end{aligned}$$

Where y_{k+1} is either y_k or y_k-1 , depending on the sign of $P1_k$. Therefore, if $P1_k < 0$ or negative then y_{k+1} will be y_k and the formula to calculate $P1_{k+1}$ will be:

$$P1_{k+1} = P1_k + 2 r_y^2 (x_k + 1) + r_x^2 (y_k^2 - y_k^2) - r_x^2 (y_k - y_k) + r_y^2$$

$$P1_{k+1} = P1_k + 2 r_y^2 (x_k + 1) + r_y^2$$

Otherwise, if $P1_k > 0$ or positive then y_{k+1} will be y_k-1 and the formula to calculate $P1_{k+1}$ will be:

$$P1_{k+1} = P1_k + 2 r_y^2 (x_k + 1) + r_x^2 ((y_k - 1)^2 - y_k^2) - r_x^2 (y_k - 1 - y_k) + r_y^2$$

$$P1_{k+1} = P1_k + 2 r_y^2 (x_k + 1) - 2 r_x^2 (y_k - 1) + r_y^2$$

The starting $P1_k$ will be evaluated at the pixel position (0, r_y)

$$P1_0 = r_y^2 (0 + 1)^2 + r_x^2 (r_y - 1/2)^2 - r_x^2 r_y^2$$

$$P1_0 = r_y^2 + r_x^2 r_y^2 - r_x^2 r_y + 1/4 r_x^2 - r_x^2 r_y^2$$

$$P1_0 = r_y^2 - r_x^2 r_y + 1/4 r_x^2$$

In Region 2, we increment y in unit steps and decide about the x coordinate using the decision parameter:

$$P2_k = f_{\text{ellipse}} (x_k + 1/2, y_k - 1)$$

$$f_{\text{ellipse}} (x_k + 1/2, y_k - 1) = r_y^2 (x_k + 1/2)^2 + r_x^2 (y_k - 1)^2 - r_x^2 r_y^2 \text{ ————— (3)}$$

$$P2_{k+1} = f_{\text{ellipse}} (x_{k+1} + 1/2, y_{k+1} - 1)$$

Now since $y_{k+1} = y_k - 1$ therefore

$$P2_{k+1} = r_y^2 (x_{k+1} + 1/2)^2 + r_x^2 [(y_k - 1) - 1]^2 - r_x^2 r_y^2 \text{ ——— (3)}$$

$$P2_{k+1} = P2_k - 2 r_x^2 (y_k - 1) + r_x^2 + r_y^2 [(x_{k+1} + 1/2)^2 - (x_k + 1/2)^2]$$

Where x_{k+1} is either set to x_k or $x_k + 1$ depending upon sign of $p2_k$.

The initial parameter for region 2 will be calculated by following formula using the last point calculated in region 1 as:

$$P2_0 = r_y^2 (x_0 + 1/2) + r_x^2 (y_0 - 1)^2 - r_x^2 r_y^2$$

Algorithm

MidpointEllipse (xcenter, ycenter, r_x, r_y)

$x = 0$

$y = r_y$

Perform

DrawSymmetricPoints(xcenter, ycenter, x, y)

$$P1_0 = r_y^2 - r_x^2 r_y + \frac{1}{4} r_x^2$$

$x = x + 1$

Algorithm Cont..

If $p1_k < 0$

$$P1_{k+1} = P1_k + 2 r_y^2 (x_k + 1) + r_y^2$$

else

$$P1_{k+1} = P1_k + 2 r_y^2 (x_k + 1) - 2 r_x^2 (y_k - 1) + r_y^2$$

$y = y - 1$

until ($2 r_y^2 x \geq 2 r_x^2 y$)

Algorithm Cont..

In region 2, using the last point in region 1 as (x_0, y_0) :

$$P2_0 = r_y^2 (x_0 + \frac{1}{2}) + r_x^2 (y_0 - 1)^2 - r_x^2 r_y^2$$

$$y = y - 1$$

If $p2_k > 0$

$$P2_{k+1} = P2_k - 2 r_x^2 (y_k + 1) + r_x^2$$

else

$$P2_{k+1} = P2_k + 2 r_y^2 (x_k + 1) - 2 r_x^2 y_k + r_x^2$$

$$x = x + 1$$

Repeat this until $y = 0$

Other Curves

Various curve functions are useful in:

- object modeling animation path specifications

- data and function graphing

- and other graphics applications

Curves Cont...

Commonly encountered curves include:

- conics

- trigonometric functions

- exponential functions

- probability distributions

- general polynomials

- spline functions

Curves Generation

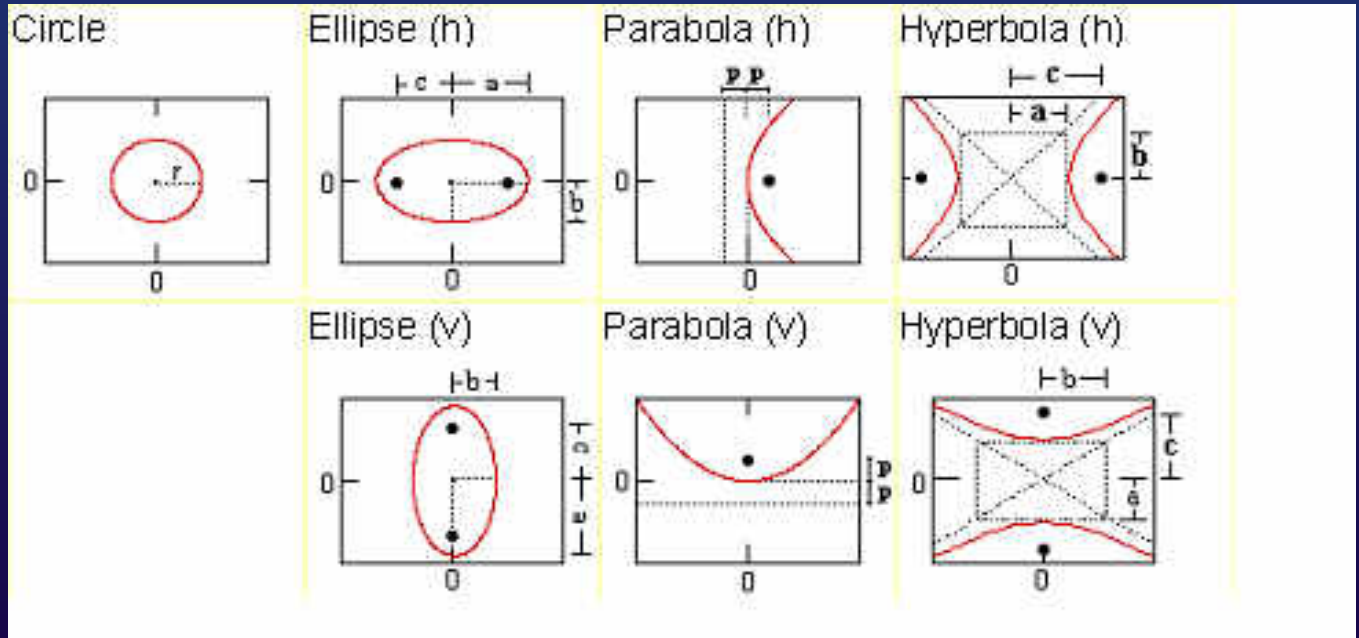
With methods similar to those discussed for the circle and ellipse.

Explicit representations $y = f(x)$ or from parametric forms.

Alternatively, incremental midpoint method to plot curves described with implicit functions $f(x,y) = 0$.

Conic Sections

A conic section is the intersection of a plane and a cone.



The general equation for a conic section:

$$Ax^2 + By^2 + Cxy + Dx + Ey + F = 0$$

The type of section can be found from the sign of : $B^2 - 4AC$

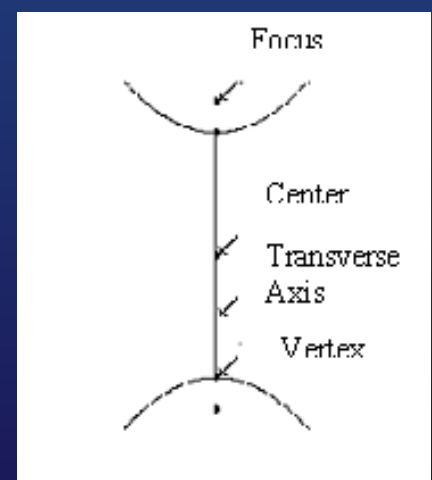
If $B^2 - 4AC$ is < 0 then the curve is a...
ellipse, circle, point or no curve.

If $B^2 - 4AC$ is $= 0$ then the curve is a...
parabola, 2 parallel lines, 1 line or no curve.

If $B^2 - 4AC$ is > 0 then the curve is a...
hyperbola or 2 intersecting lines.

Hyperbola

A hyperbola is the set of all points (x, y) in the plane the difference of whose distances from two fixed points is some constant. The two fixed points are called the foci.

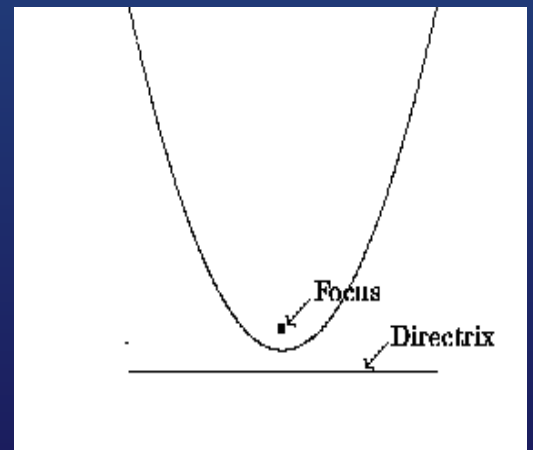


We now turn our attention to the standard equation of a hyperbola. We say that the standard equation of a hyperbola centered at the origin is given by:

if the transverse axis is horizontal, or
if the transverse axis is vertical.

Parabola

A parabola is the set of all points (x, y) that are the same distance from a fixed line (called the directrix) and a fixed point (focus) not on the directrix. See figure for the view of a parabola and its related focus and directrix.



The standard form of the equation of the parabola with vertex at (0,0) with the focus lying d units from the vertex is given by $x^2 = 4dy$

If the axis is vertical and $y^2 = 4dx$

Rotation of Axes

In conic section we stated that every conic section is of the form

where A , B , C , D , E , and F are constants

$$Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0$$

What does the xy term do? The xy term actually rotates the graph in the plane. For example, in the case of an ellipse, the major axis is no longer parallel to the x -axis or y -axis. Rather, depending on the constant in front of the xy term, we now have the major axis rotated.

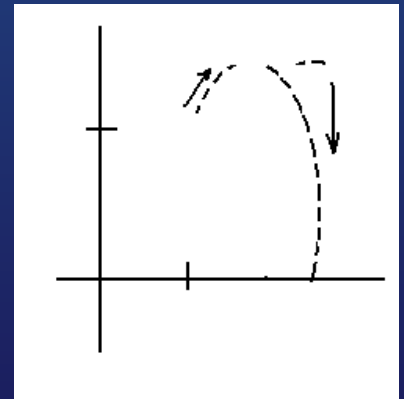
Animated Applications

Ellipses, hyperbolas, and parabolas are particularly useful in certain animation applications.

Orbital and other motions for objects

Planetary orbits in the solar system

Figure shows a parabolic path in standard position for a gravitational field acting in the negative y direction. The explicit equation for the parabolic trajectory of the object shown can be written as:



$$y = y_0 + a (x - x_0)^2 + b (x - x_0)$$

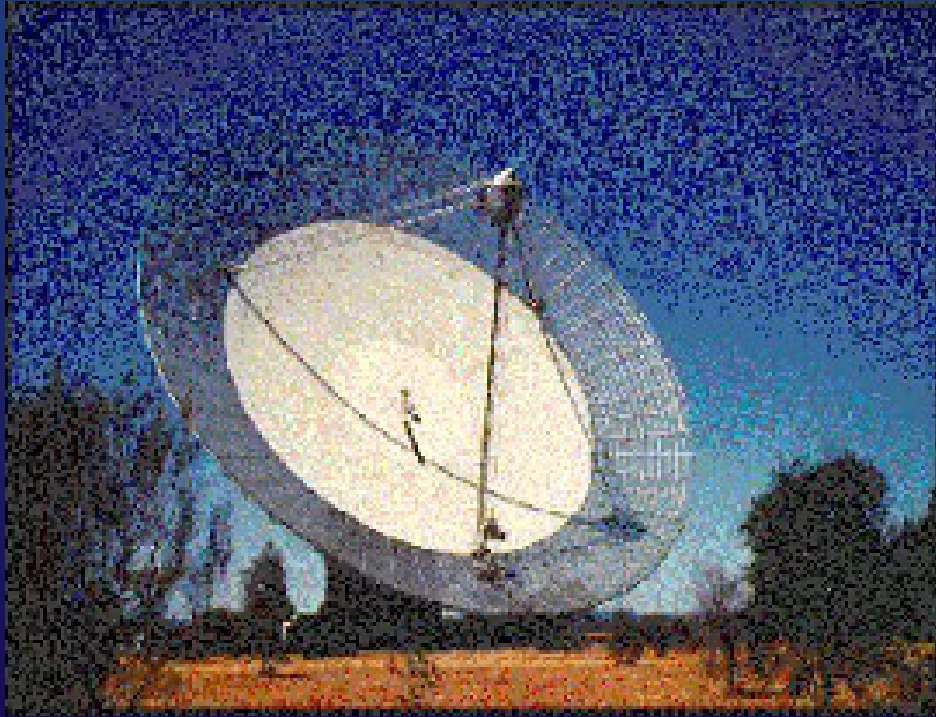
constants a and b determined by v_0 , g

Parabolic motions with parametric equations using a time parameter t:

$$x = x_0 + v_{x0} t$$

$$y = y_0 + v_{y0} t - \frac{1}{2} g t^2$$

Parabolic Reflectors



Elliptical Orbits



Whispering Galleries

In rooms whose ceilings are elliptical, a sound made at one focus of the ellipse will be reflected to the other focus (across the room), allowing people standing at the two foci to hear one another very clearly. This has been called the "whispering gallery" effect.

Polynomials and Spline Curves

A polynomial function of n th degree in x is defined as

$$y = a_k x^k$$

$$y = a_0 x^0 + a_1 x^1 + \text{-----} \\ + a_{n-1} x^{n-1} + a_n x^n$$

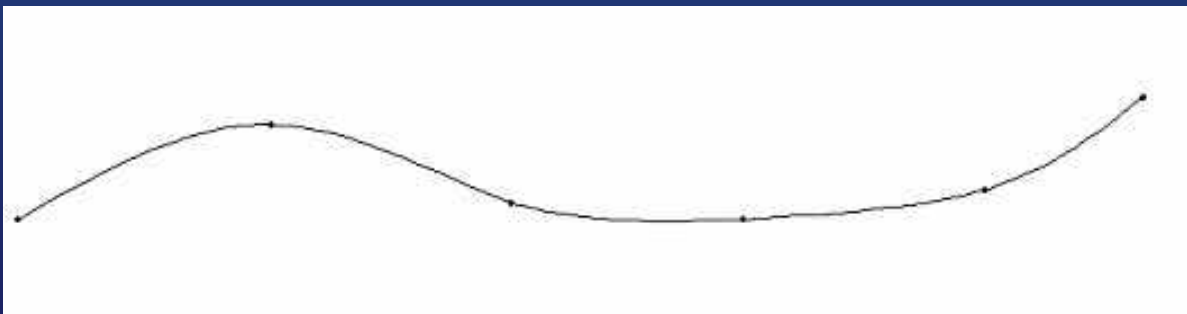
Where n is a nonnegative integer and the a_k are constants, with a_n not equal to 0.

Each curve section is then described in parametric form as

$$x = a_{x0} + a_{x1}u + a_{x2}u^2 + a_{x3}u^3$$

$$y = a_{y0} + a_{y1}u + a_{y2}u^2 + a_{y3}u^3$$

Where parameter u varies over the interval 0 to 1. A curve is shown below calculated using at right equations.



Continuous curves that are formed with polynomial pieces are called called spline curves, or simply splines. Spline is a detailed topic; which will be discussed later in 3 dimensions.

Computer Graphics

Filled Area Primitives I

Lecture 08

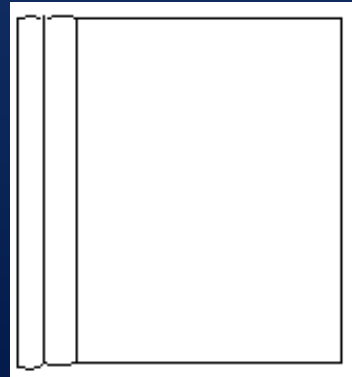
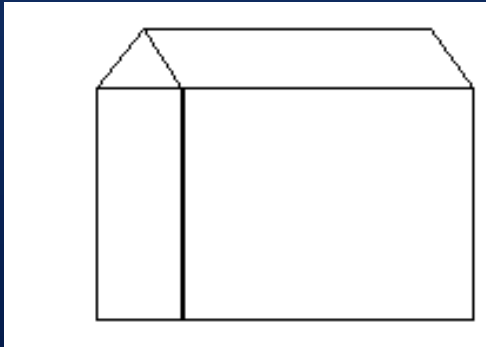
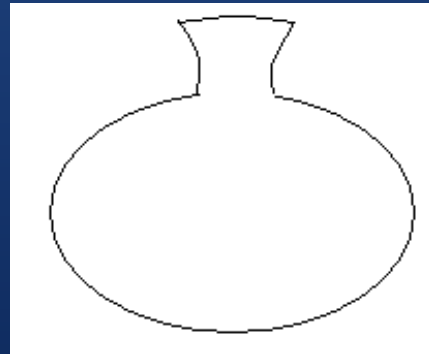
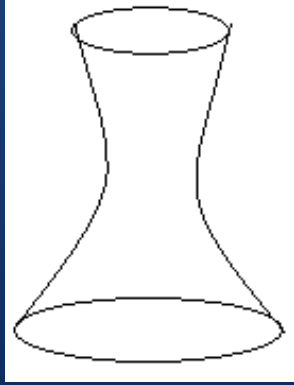
Taqdees A. Siddiqi

cs602@vu.edu.pk

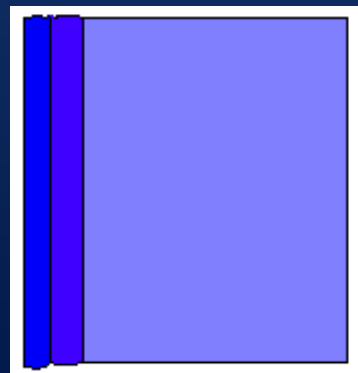
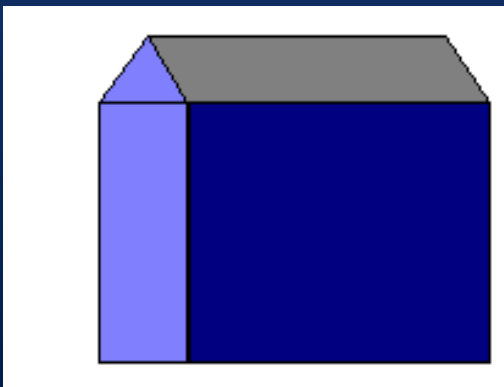
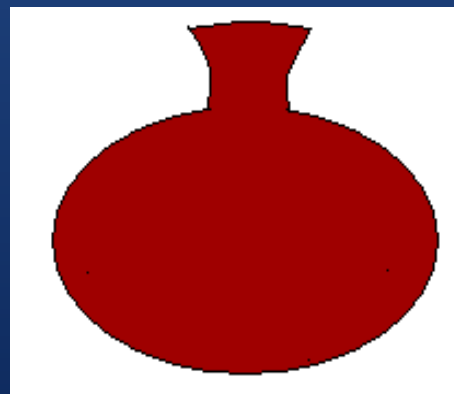
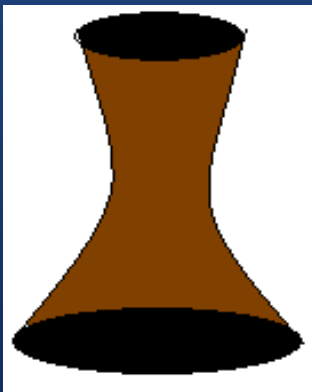
Drawing Primitives

- ◆ Point
- ◆ Line
- ◆ Circle
- ◆ Ellipse
- ◆ Curves

Figures Using Primitives



Solid Models



Object Filling

Two Basic Approaches

- ◆ Scan-line Between Boundaries
- ◆ Select interior point - Paint outwards

Scan-line Polygon Filling

Draw straight lines between the edges of polygon called scan-line polygon filling.

Polygon ?

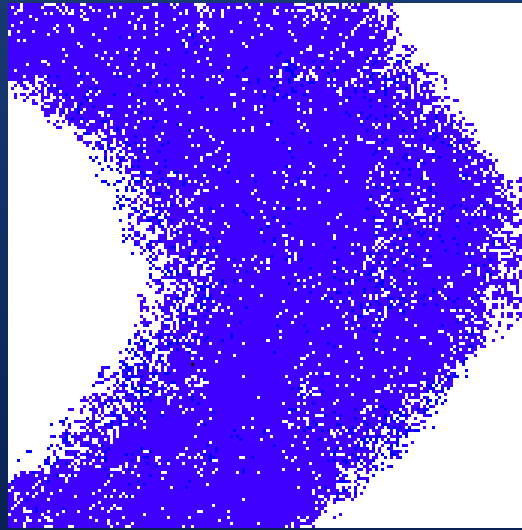
Boundary-fill

Start from an interior point and paint outward from this point till the boundary is reached; called boundary-fill.

Flood-fill

Start from an interior point and start painting outward from this point till the end of cluster.

Flood-fill



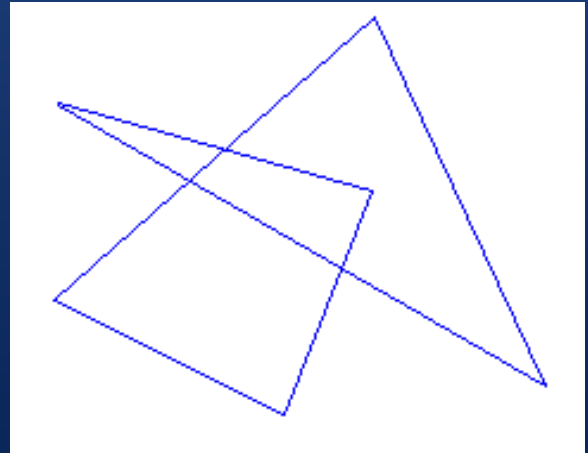
Scan-line Polygon Filling

For having a good understanding of the concept and its implementation we will discuss the following topics:

- ◆ Polygon Definition
- ◆ Filled vs. Unfilled Polygons
- ◆ Parity Definition
- ◆ Scan-Line Polygon Fill Algorithm
- ◆ Special Cases Handled By the Fill
- ◆ Polygon Fill Example

Polygon

A **polygon** can be defined as a shape that is formed by line segments that are placed end to end, creating a continuous closed path.

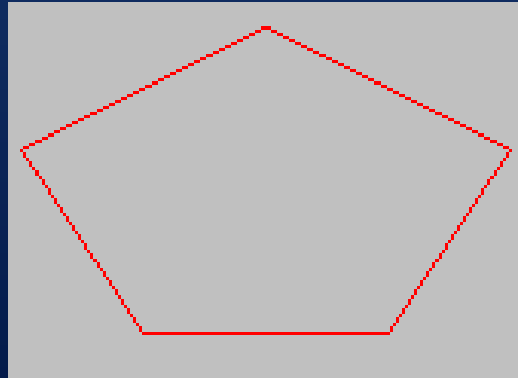
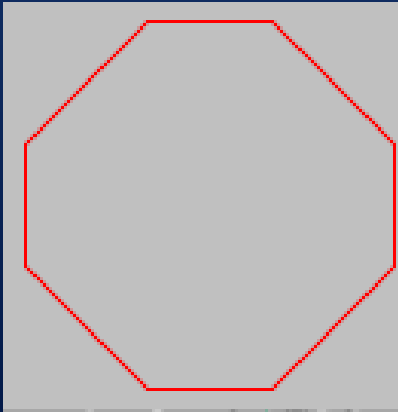


Polygon Types

Polygons can be divided into three basic types: **convex**, **concave**, and **complex**.

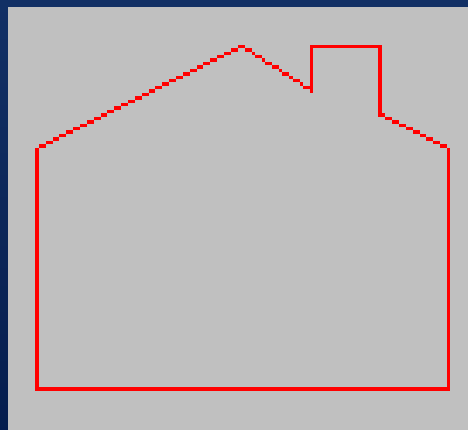
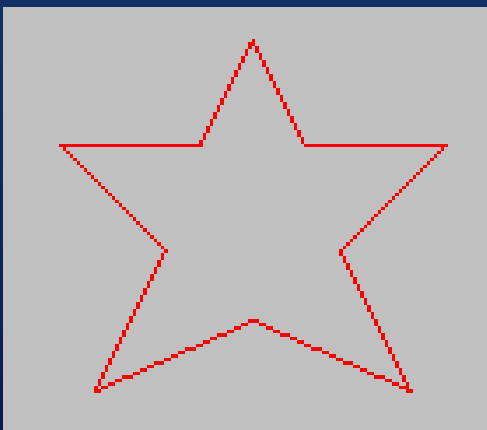
Convex

Does a straight line connecting ANY two points that are inside the polygon intersect any edges of the polygon?



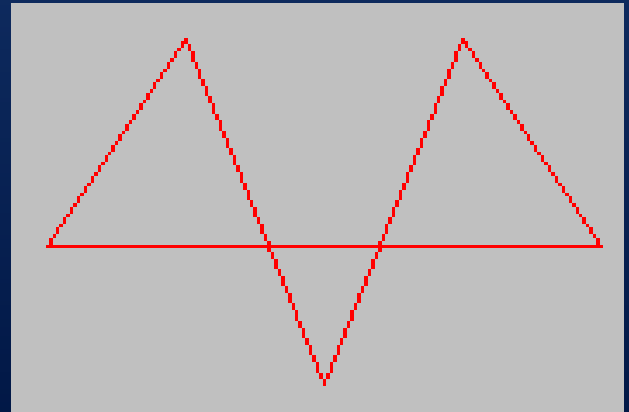
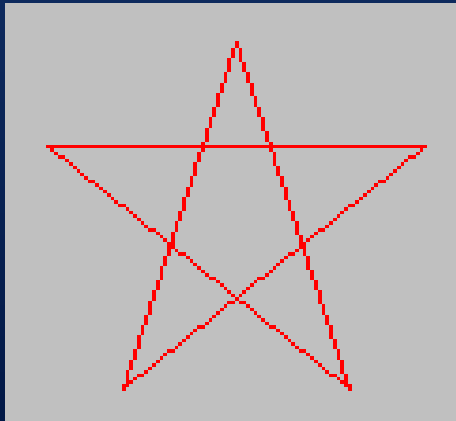
Concave

The polygon edges may also touch each other, but they may not cross one another.



Complex

Complex polygons are basically concave polygons that may have self-intersecting edges. The complexity arises while distinguishing which side is inside the polygon when filling it.

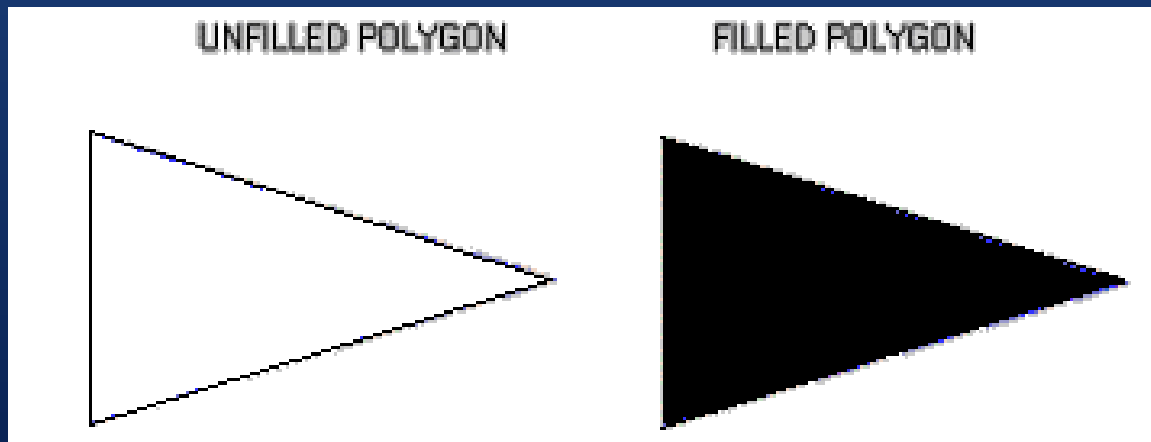


Difference Between Filled and Unfilled Polygon

When an unfilled polygon is rendered, only the points on the perimeter of the polygon are drawn.

When a polygon is filled, the interior of the polygon must be considered. All of the pixels within the boundaries of the polygon must be set to the specified color or pattern.

Difference Between Filled and Unfilled Polygon



Parity Requirement

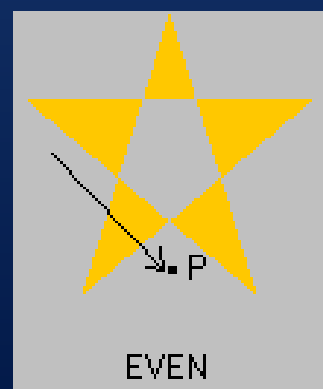
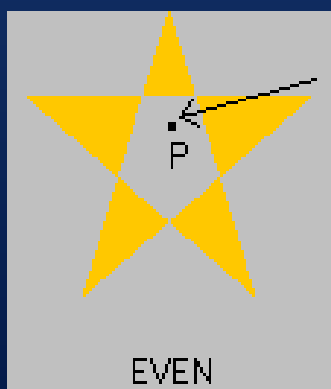
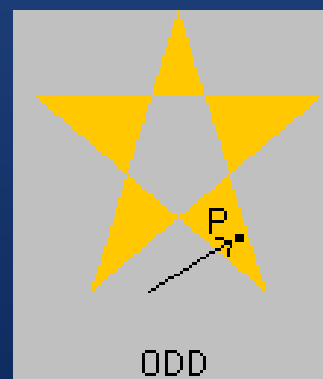
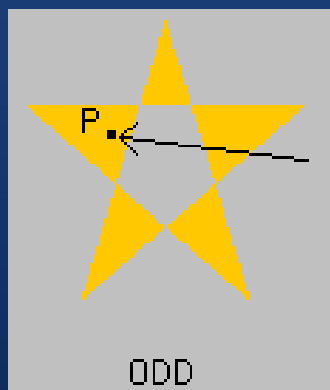
In order to determine which pixels are inside the polygon, the odd-parity rule is used within the scan-line polygon fill algorithm.

Parity

Draw a line segment from any point that lies outside the polygon to a point P.

Count the number of edges that the line crosses. If the number of polygon edges crossed is **odd**, then P lies within the polygon. Whereas, if the number of edges is even, then P lies outside the polygon.

Parity Examples



Parity Test in Algorithm

How do we determine whether a pixel lies outside of the polygon to test for an inside one?

(is the polygon completely visible in scene?)

In start parity is even, on crossing first edge parity becomes odd, so start colouring then crossing second edge parity becomes even, so stop colouring and so on.

Parity Test in Algorithm

So, you can see if we have a sorted list of x-intersections of all of the polygon edges with the scan line, we can simply draw/fill from the first x to the second, the third to the forth and so on.

Polygon Filling

- Algorithm independent of polygon type
- Complex polygon is superset of all polygons
- Algorithm for complex polygon should work for all

Algorithm

When filling a polygon, you will most likely just have a set of vertices, indicating the x and y Cartesian coordinates of each vertex of the polygon. The following steps should be taken to turn your set of vertices into a filled polygon.

Steps

1. Initializing All of the Edges
2. Initializing the Global Edge Table
3. Initializing Parity
4. Initializing the Scan-Line
5. Initializing the Active Edge Table
6. Filling the Polygon

Initializing All of the Edges

all_edges table holds:

- The minimum y value of the two vertices
- The maximum y value of the two vertices
- The x value associated with the minimum y value
- The slope of the edge

Initializing All of the Edges

The slope of the edge can be calculated from the formula for a line:

$$y = mx + b;$$

where m = slope, b = y-intercept,

y_0 = maximum y value,

y_1 = minimum y value,

Initializing All of the Edges

x_0 = maximum x value,

x_1 = minimum x value

The formula for the slope is as follows:

$$m = (y_0 - y_1) / (x_0 - x_1)$$

Initializing All of the Edges

Index		Y-min	Y-max	X-val	1/m
0	→	10	16	10	0
1	→	16	20	10	1.5
		-	-	-	-
		-	-	-	-
N	→	10	16	28	0

Table: All_edges

Initializing the Global Edge Table

The global edge table will be used to keep track of the edges that are still needed to complete the polygon.

Filling the edges from bottom to top and left to right.

Global edge table should have edges sorted on minimum value of y .

Initializing the Global Edge Table

Edges with the same minimum y values are sorted on minimum x values as follows:

- ◆ Place the first edge with a slope that is not equal to zero in the global edge table.

- ◆ If the slope of the edge is zero, do not add that edge to the global edge table.
- ◆ For every other edge, start at index 0 and increase the index of the global edge table once each time the current edge's y value is greater than that of the edge at the current index in the global edge table.

Initializing Parity

The initial parity is even since no edges have been crossed yet.

Initializing the Scan-Line

The initial scan-line is equal to the lowest y value for all of the global edges.

Initializing the Active Edge Table

The active edge table will be used to keep track of the edges that are intersected by the current scan-line. This should also contain ordered edges.

This is initially set up as follows:

Each edge found having a minimum y value equal to the current scan-line, append the edge information for the maximum y value, x value, and $1/m$ to the active edge table.

Initializing the Active Edge Table

Index		Y-max	X-val	1/m
0	→	16	10	0
1	→	20	10	1.5
		-	-	-
		-	-	-
N	→	16	28	0

Active

Filling the Polygon

Filling the polygon involves deciding whether or not to draw pixels, adding to and removing edges from the active edge table, and updating x values for the next scan-line.

Starting with the initial scan-line, until the active edge table is empty, do the following:

1. Draw all pixels from the x value of odd to the x value of even parity edge pairs
2. Increment the scan-line by 1
3. Remove any edges from the active edge table for which the maximum y value is equal to the scan line

Filling the Polygon

4. Update the x value for each edge in the active edge table using the formula $x_1 = x_0 + 1/m$.

(This is based on the line equation and the fact that the next scan-line equals the old scan-line plus one.)

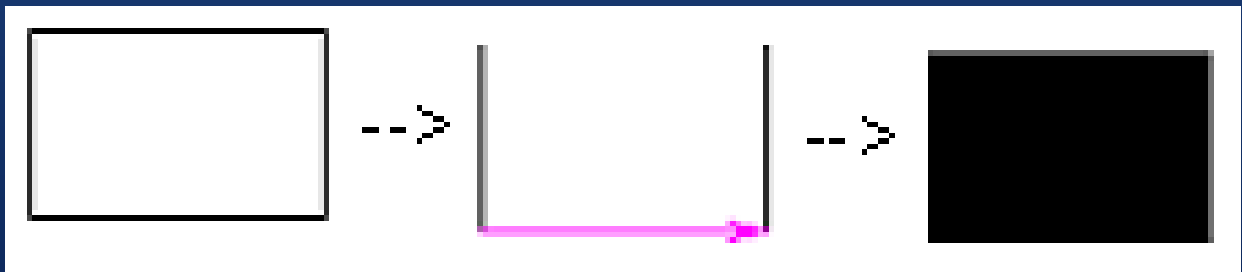
5. Remove any edges from the global edge table for which the minimum y value is equal to the scan-line and place them in the active edge table
6. Reorder the edges in the active edge table according to increasing x value. This is done in case edges have crossed

Special Cases

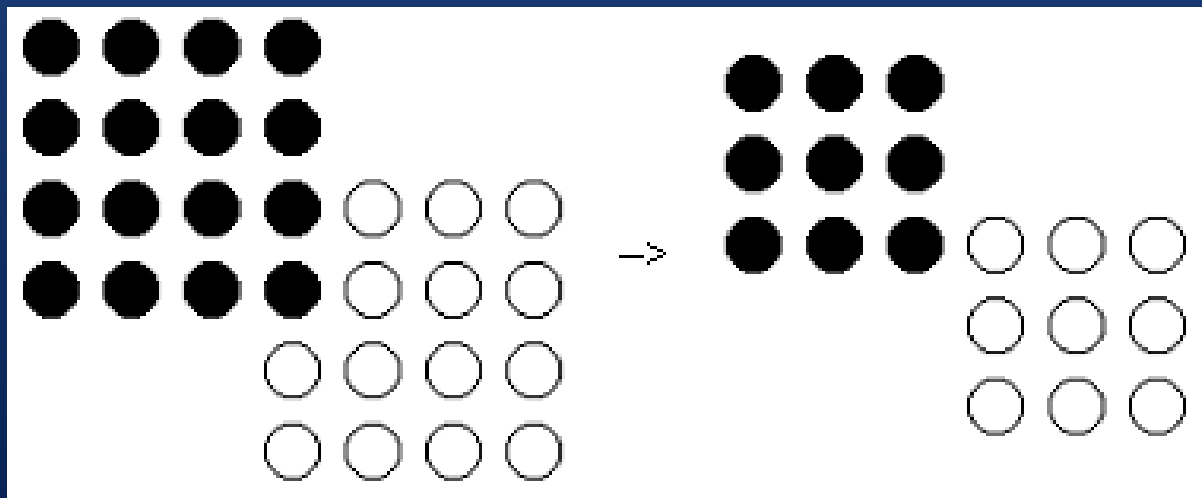
There are some special cases, the scan-line polygon fill algorithm covers these cases, but you may not understand how or why.

The following will explain the handling of special cases by the algorithm.

1. Horizontal Edges

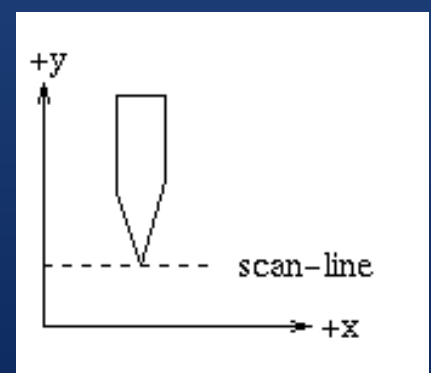


2. Bottom and Left Edges vs. Top and Right Edges

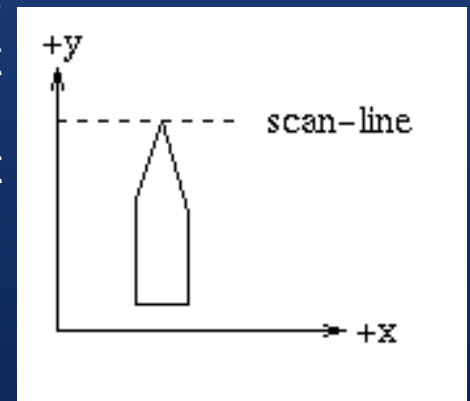


3. Edges Meeting at a Vertex

When dealing with two edges meeting at a vertex and for both edges the vertex is the minimum point the pixel is **drawn** and is **counted twice for parity**.

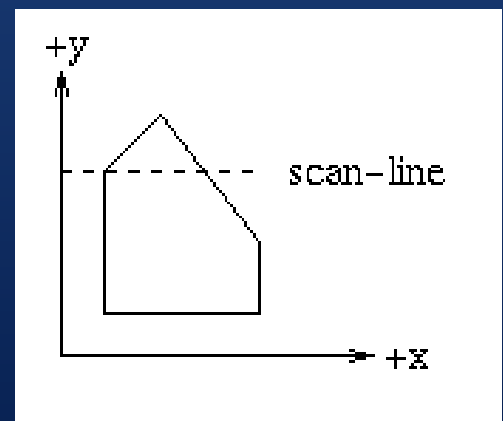


Vertex is drawn for the first edge, since it is a minimum value for that edge, but not for the second edge, since it is a right edge and right edges are not drawn in the scan-line fill algorithm

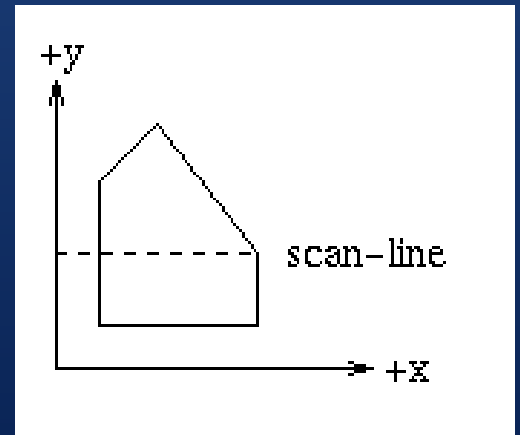


3. Edges Meeting at a Vertex

A vertex and for one edge the vertex is the maximum point and for the other edge the vertex is the minimum point



Both edges are on the **right**, the pixel is **not drawn** and is **counted just once for parity** purposes. This is due to the fact that right edges are not drawn in the scan-line polygon fill implementation.



Computer Graphics

Filled Area Primitives I

Lecture 08

Taqdees A. Siddiqi

cs601@vu.edu.pk

Computer Graphics

Filled Area Primitives II

Lecture 09

Taqdees A. Siddiqi
cs602@vu.edu.pk

Filled Area Primitives

- There are two basic approaches to area filling on raster systems.
- scan-line polygon filling
- Boundary-fill and Flood-fill

Polygon

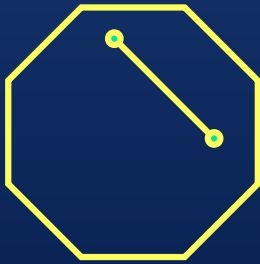
- A polygon can be defined as a shape that is formed by line segments that are placed end to end, creating a continuous closed path. Polygons can be divided into three basic types:
- convex
- concave
- complex

Polygon types

- Does a straight line connecting ANY two points that are inside the polygon intersect any edges of the polygon?

Convex polygons

- If the answer is no then polygon is convex



Concave polygons

- Otherwise the polygon is concave



Complex polygons

- Complex polygons are basically concave polygons that may have self-intersecting edges



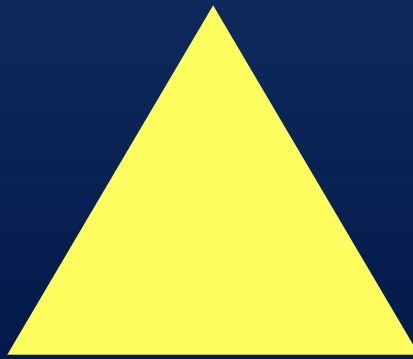
Unfilled polygons

- When an unfilled polygon is rendered, only the points on the perimeter of the polygon are drawn.



Filled Polygons

- When a polygon is filled, the interior of the polygon must be considered. All of the pixels within the boundaries of the polygon must be set to the specified color or pattern.

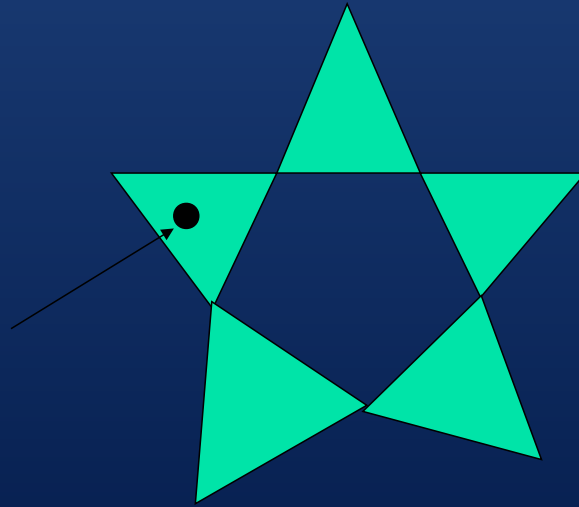


Parity

- Parity is a concept used to determine which pixels lie within a polygon, i.e. which pixels should be filled for a given polygon.

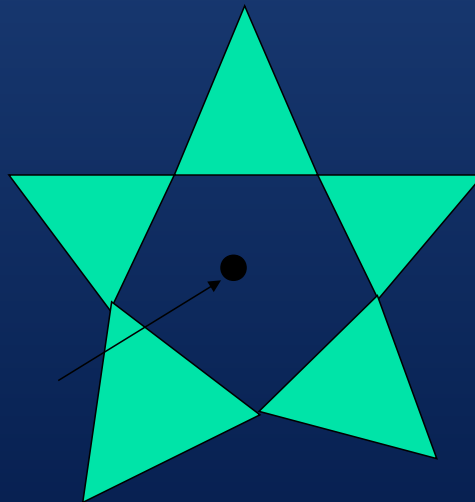
ODD number of edges

- Point is inside the polygon filled area



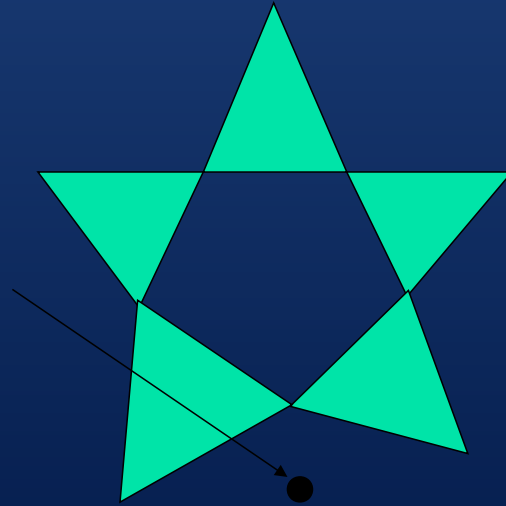
Even number of edges

- Point lies outside the polygon



Even number of edges II

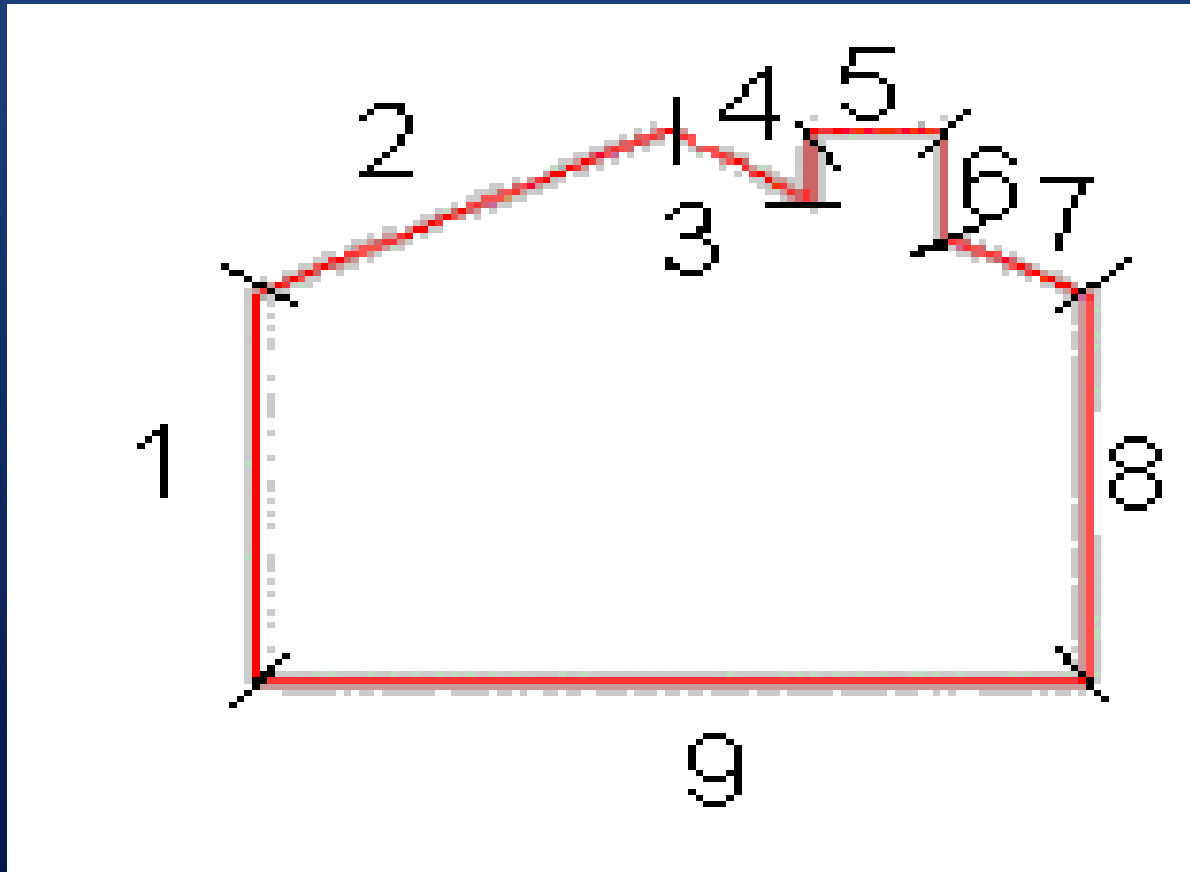
- Point lies outside the circle



Polygon Filling Algorithm

- When filling a polygon, you will most likely just have a set of vertices, indicating the x and y Cartesian coordinates of each vertex of the polygon. The following steps should be taken to turn your set of vertices into a filled polygon.

1. Initialize all of the edges



- For each edge, the following information needs to be kept in a table:
 1. The minimum y value of the two vertices
 2. The maximum y value of the two vertices
 3. The x value associated with the minimum y value
 4. The slope of the edge

Cont...

- The slope of the edge can be calculated from the formula for a line

$$y = mx + b$$

where

m = slope

b = y-intercept,

y_0 = maximum y value,

y_1 = minimum y value,

x_0 = maximum x value,

x_1 = minimum x value

The formula for the slope is as follows:

$$m = (y_0 - y_1) / (x_0 - x_1).$$

Index		Y-min	Y-max	X-val	1/m
0	→	10	16	10	0
1	→	16	20	10	1.5
		-	-	-	-
		-	-	-	-
N	→	10	16	28	0

Table: All edges

2. Initialize the Global edge Table

- Edges with the same minimum y values are sorted on minimum x values as follows:
 1. First edge with non zero slope.
 2. If the slope of an edge is zero, do not add it.
 3. For every other edge, start at index 0 and look for an appropriate position for the edge, in ascending order of minimum y-values

3. Initialize Parity

- The initial parity is even since no edges have been crossed yet.

4. Initializing the Scan-Line

- The initial scan-line is equal to the lowest y value for all of the global edges.

5. Initializing the Active Edge Table

Index		Y-max	X-val	1/m
0	→	16	10	0
1	→	20	10	1.5
		-	-	-
		-	-	-
N	→	16	28	0

6. Filling the Polygon

- Starting with the initial scan-line, until the active edge table is empty, do the following steps:

Step 1

- Draw all pixels from the x value of odd to the x value of even parity edge pairs.

Step 2

- Increase the scan-line by 1.

Step 3

- Remove any edges from the active edge table for which the maximum y value is equal to the scan line.

Step 4

- Update the x value for each edge in the active edge table using the formula
 - $x_1 = x_0 + 1/m$
 - (This is based on the line formula and the fact that the next scan-line equals the old scan-line plus one.)

Step 5

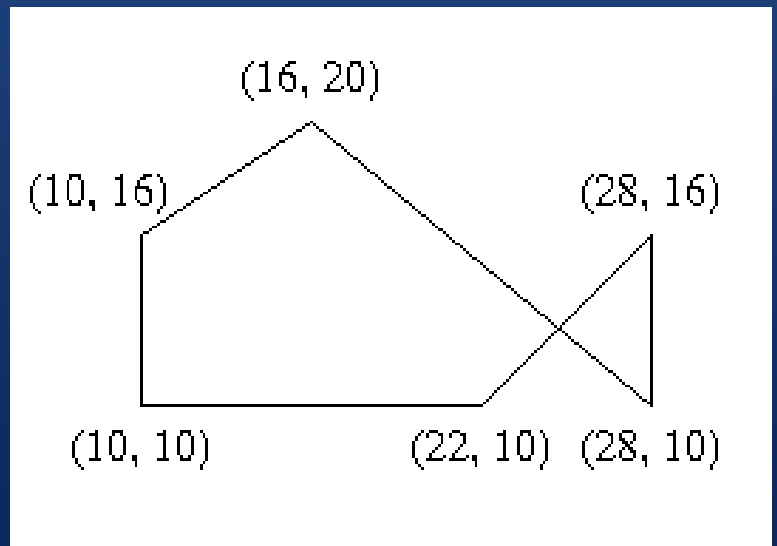
- Remove any edges from the
global edge table
for which the
minimum y value = the scan-line
and place them in the
active edge table.

Step 6

- Reorder the edges in the active edge table according to increasing x value. This is done in case edges have crossed.

An Example

0	(10, 10)
1	(10, 16)
2	(16, 20)
3	(28, 10)
4	(28, 16)
5	(22, 10)



Steps involved

- We will now walk through the steps of the algorithm to fill in the polygon:

1. Initializing All of the Edges

Index		Y-min	Y-max	X-val	1/m
0		10	16	10	0
1		16	20	10	1.5
2		10	20	28	-1.2
3		10	16	28	0
4		10	16	22	1
5		10	10	10	Inf

2. Initializing the Global Edge Table:

Index		Y-min	Y-max	X-val	1/m
0		10	16	10	0
1		10	16	22	1
2		10	16	28	0
3		10	20	28	-1.2
4		16	20	10	1.5

3. Initialize Parity

- Parity is initially set to Even

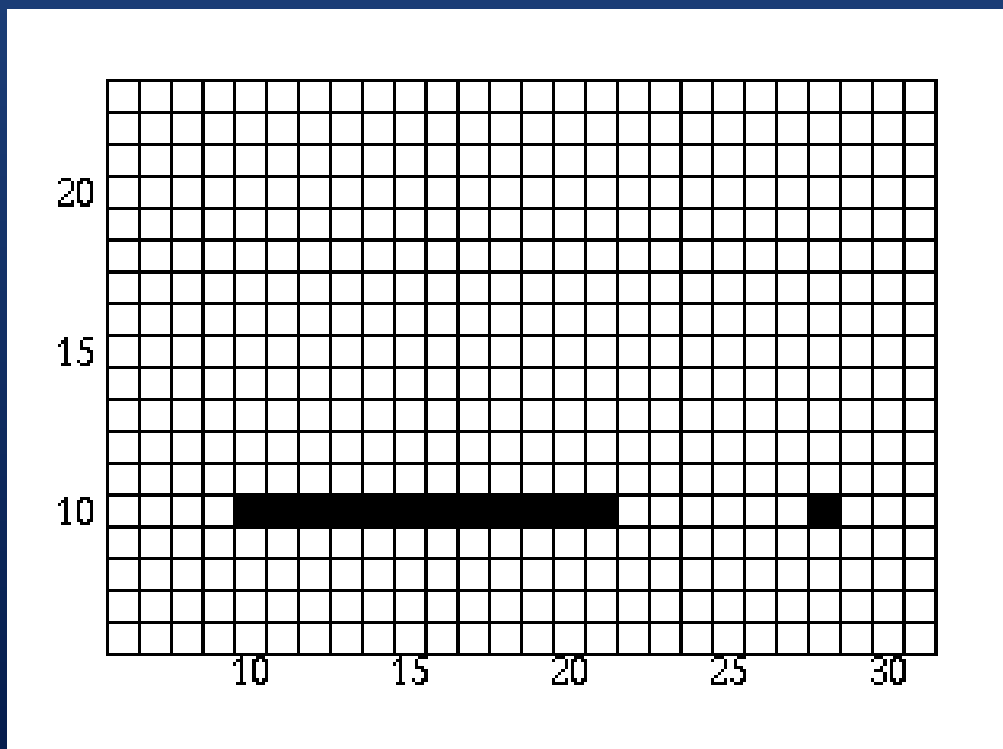
4. Initializing the Scan-Line

- Since the lowest y value in the global edge table is 10, we can safely choose 10 as our initial scan-line.

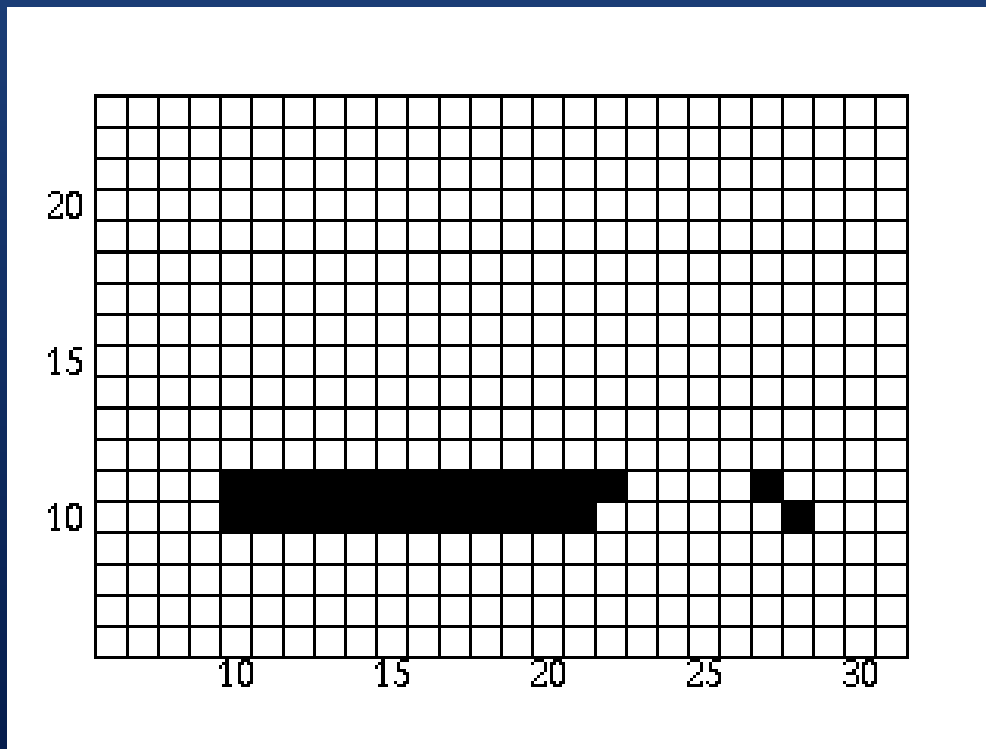
5. Initializing the Active Edge Table

Index		Y-max	X-val	1/m
0	→	16	10	0
1	→	16	22	1
2	→	16	28	0
3	→	20	28	-1.2
4	→	20	10	1.5

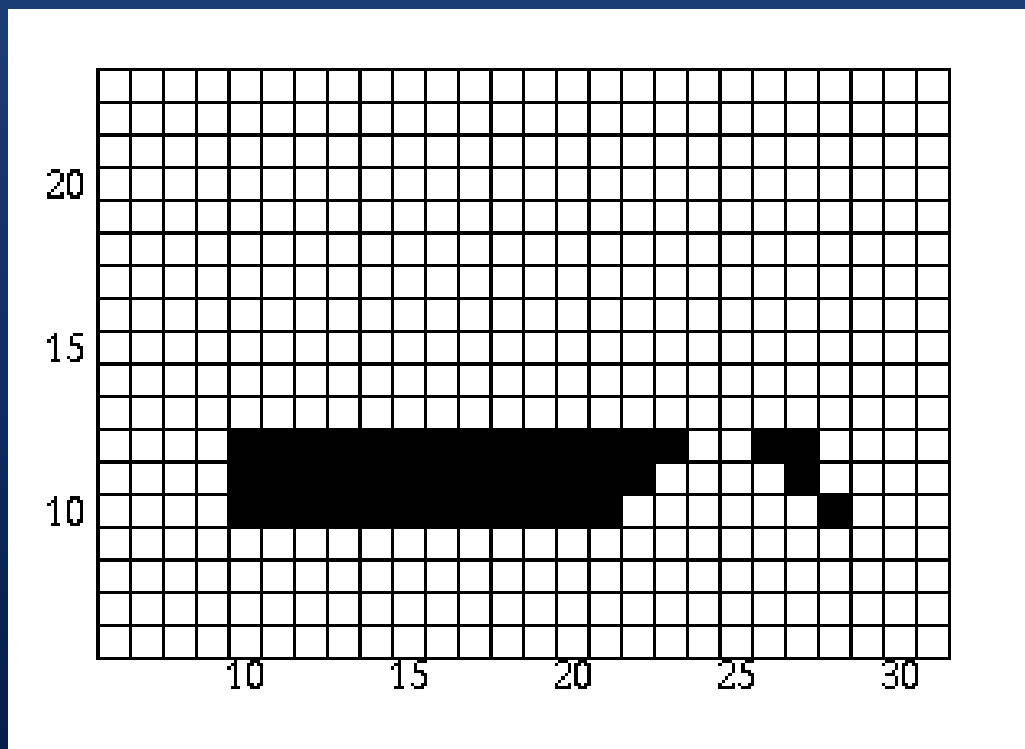
6. Filling the polygon



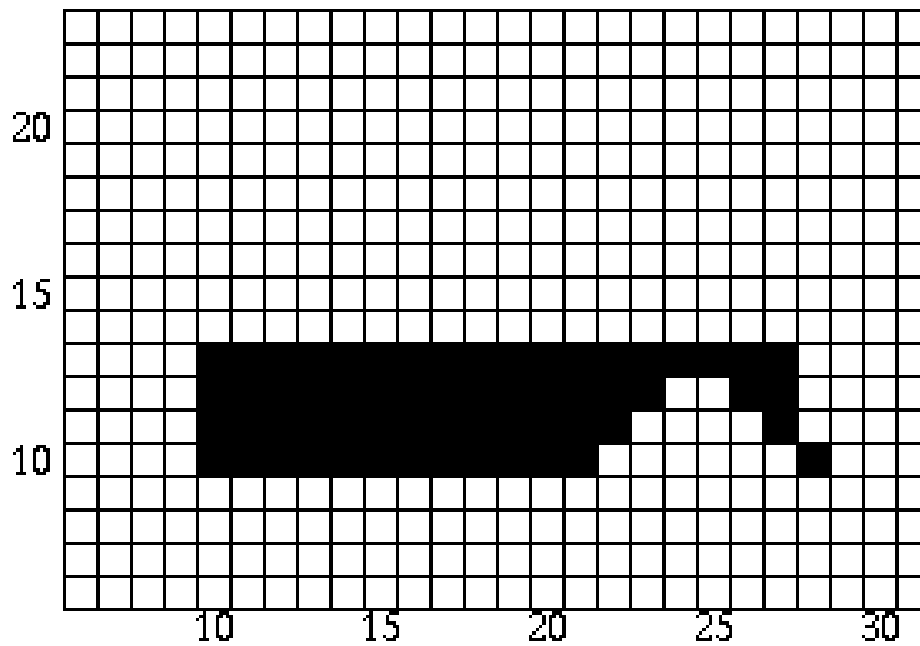
6. Filling the polygon



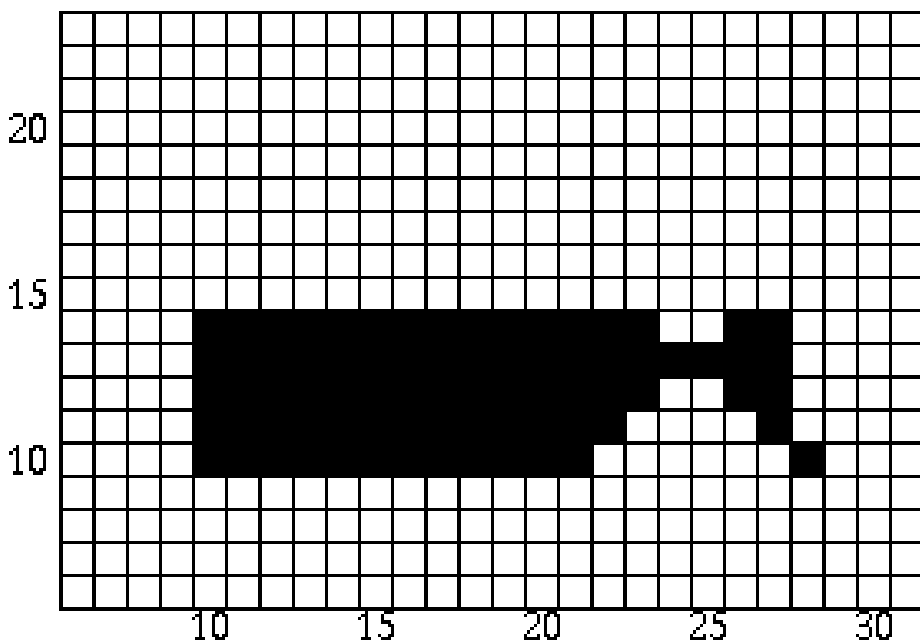
6. Filling the polygon



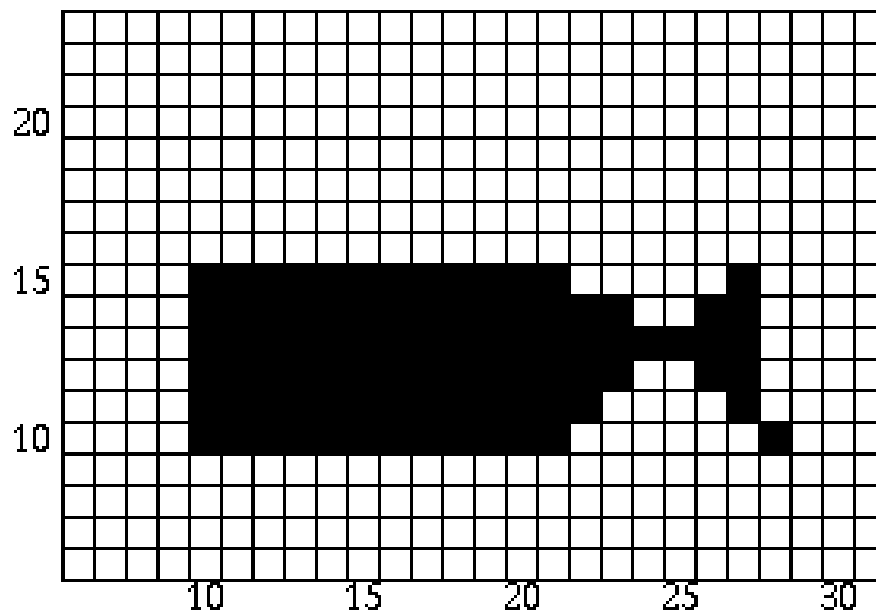
6. Filling the polygon



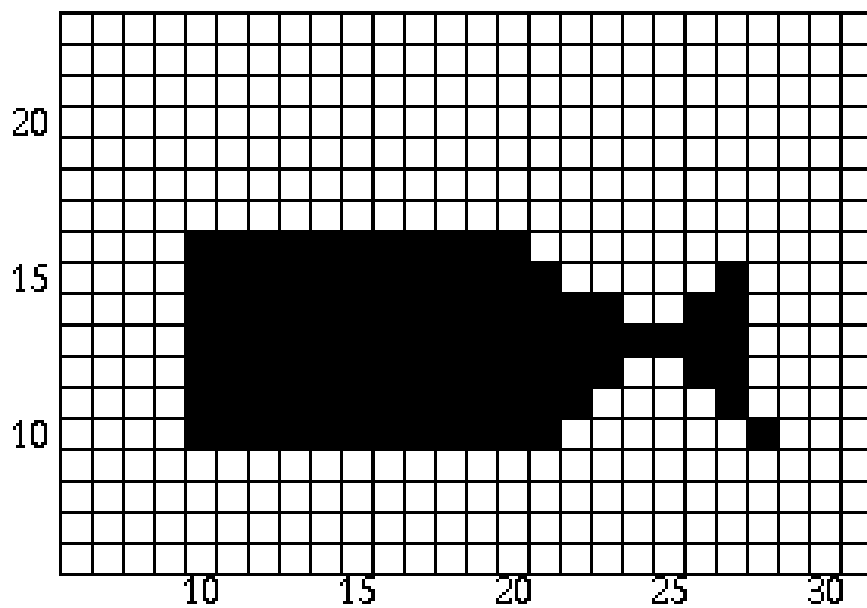
6. Filling the polygon



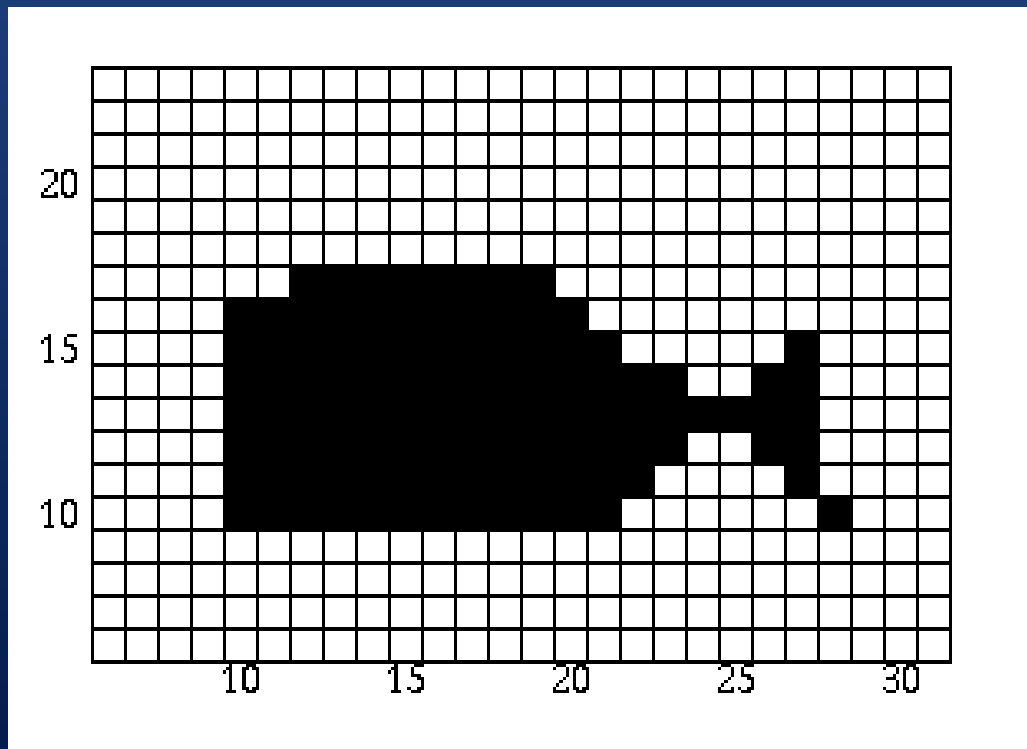
6. Filling the polygon



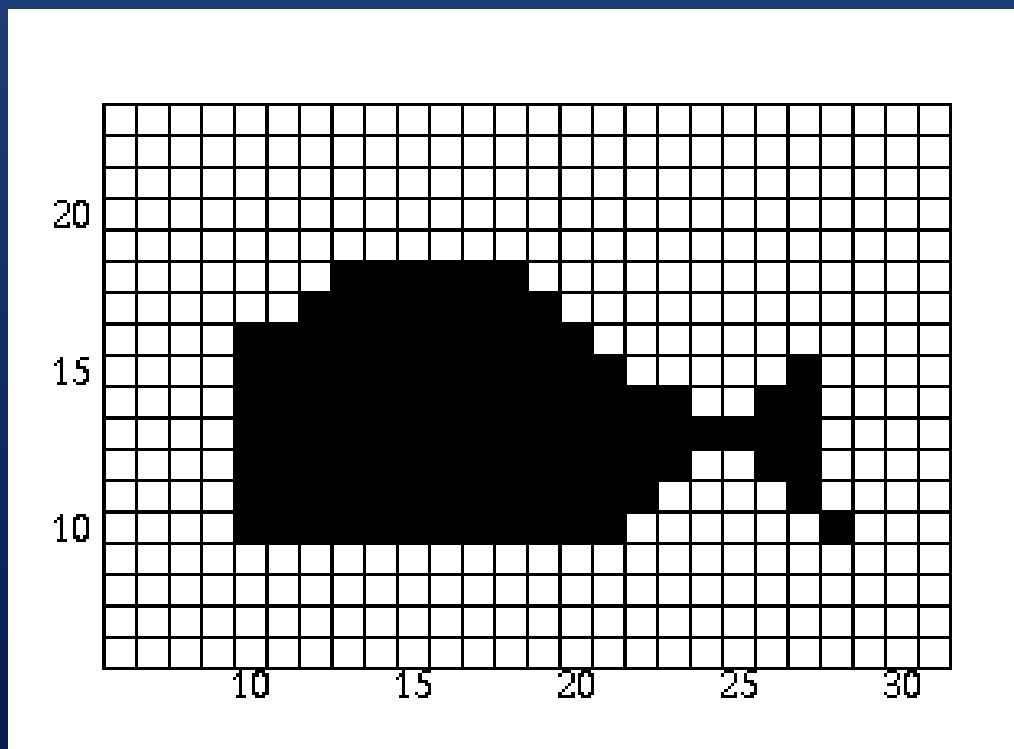
6. Filling the polygon



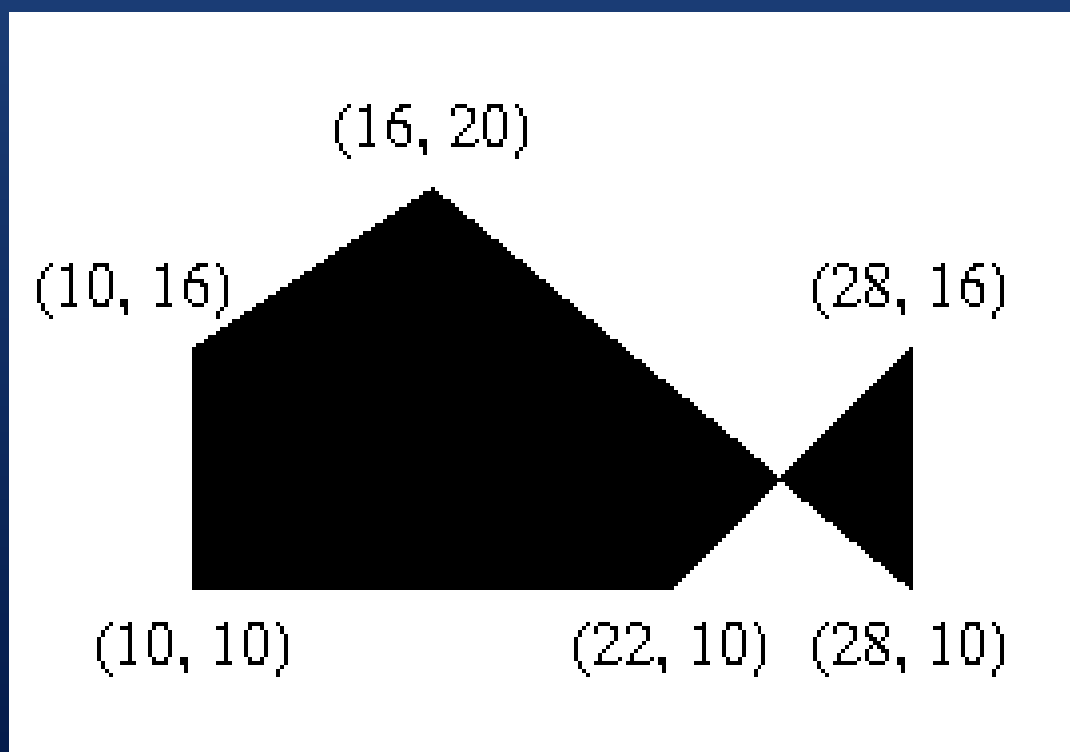
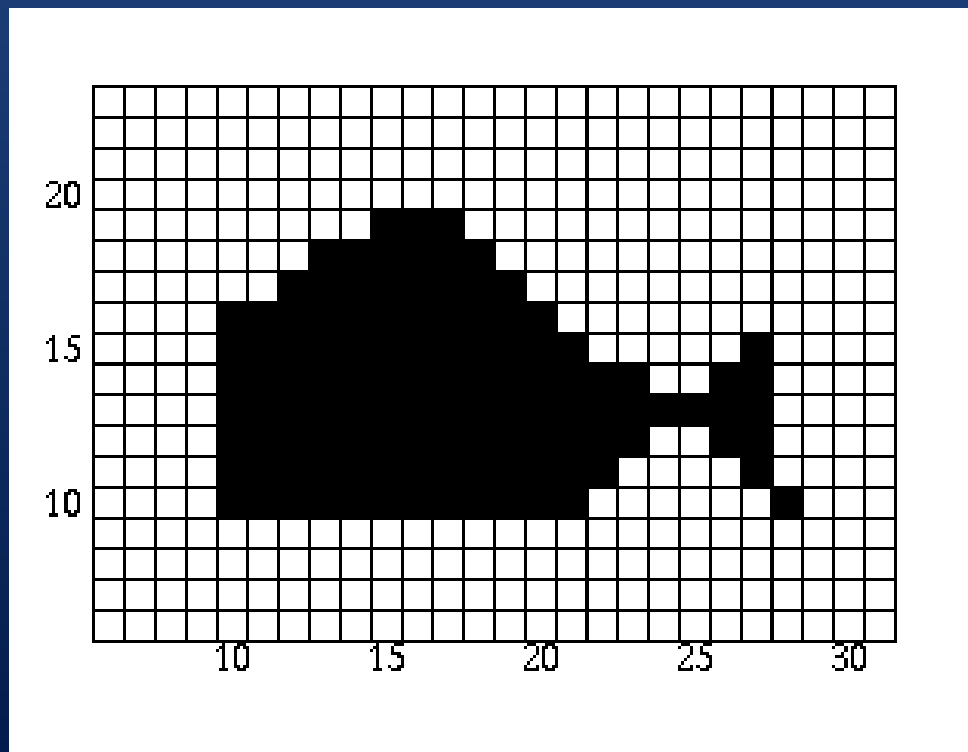
6. Filling the polygon



6. Filling the polygon



6. Filling the polygon



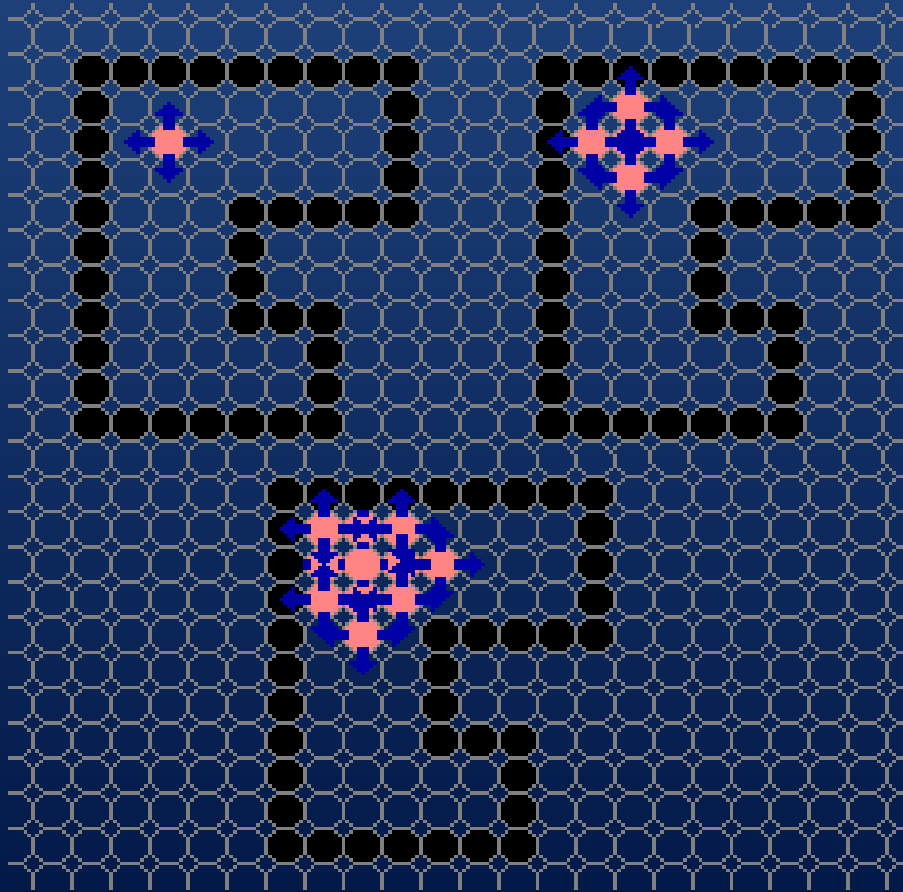
Boundary fill

- Check the pixel for boundary color
- Check the pixel for fill color
- Set the pixel in fill color
- Run the process for neighbors

Boundary fill

- Boundary fill is pixel checking and setting method that involves following steps:
 1. Check the width and height of area
 2. Get the current pixel
 3. Apply boundary checking
 4. Set the pixel
 5. Call this function recursively and set all neighboring pixels.

Process



Algorithm

- BoundaryFill (x, y, fillColor ,
 boundaryColor)
- if ((x < 0) || (x >= width)) then return
- if ((y < 0) || (y >= height)) then return
- current = GetPixel(x, y)

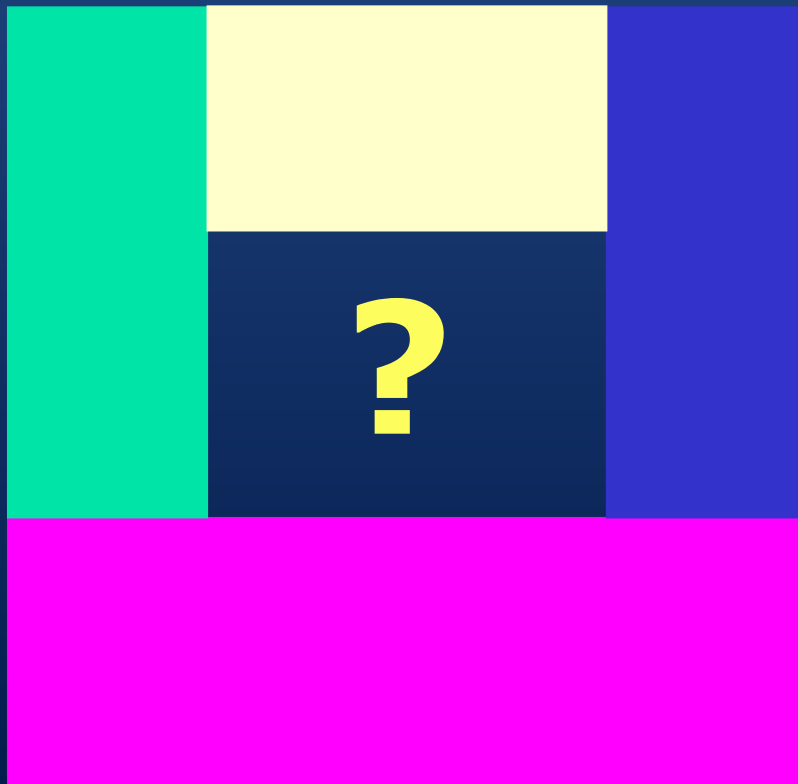
Algorithm

- ```

 if ((current != boundaryColor) &&
 (current != fillColor))
 setPixel(fillColor, x, y)
 BoundaryFill(x+1,y,fillColor,boundaryColor)
 BoundaryFill(x,y+1,fillColor,boundaryColor)
 BoundaryFill (x-1,y,fillColor,boundaryColor)
 BoundaryFill (x,y-1,fillColor,boundaryColor)

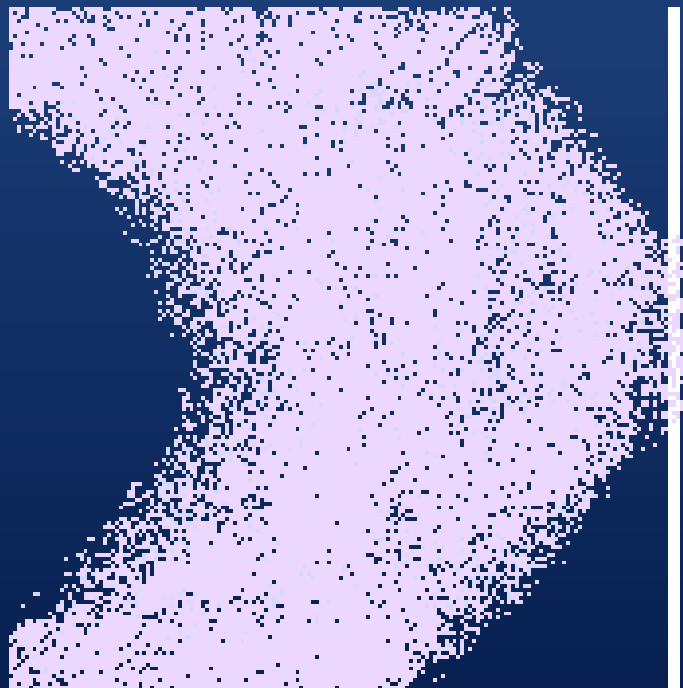
```

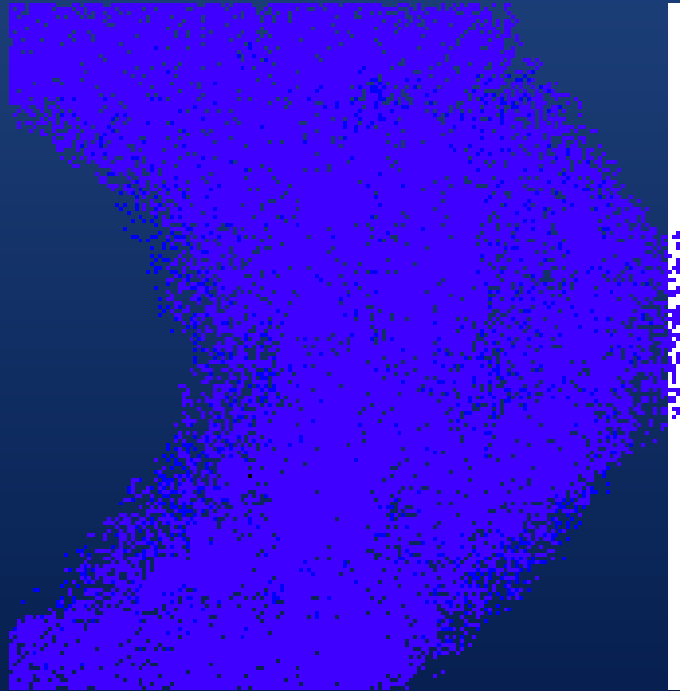
# How about this?



# Flood Fill

- An area fill algorithm that replaces all connected pixels of a selected color with a fill color.





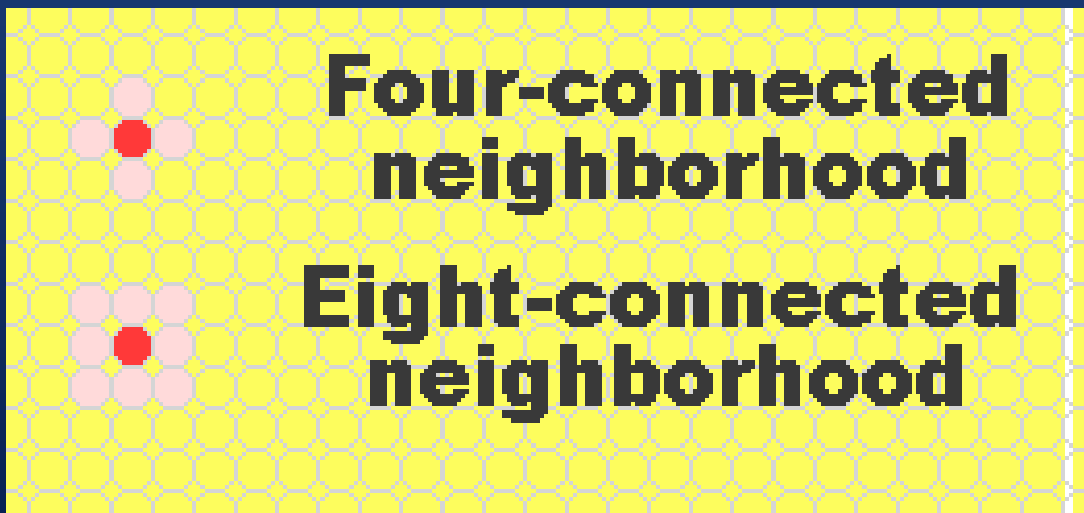
## Algorithm

- `floodFill(x, y, fillColor, oldColor)`
- `if ((x < 0) || (x >= width))` then return
- `if ((y < 0) || (y >= height))` then return

# Algorithm

- if ( `getPixel (x, y) == oldColor`) then
  - `setPixel (fillColor, x, y)`
  - `floodFill (x+1, y, fillColor, oldColor)`
  - `floodFill (x, y+1, fillColor, oldColor)`
  - `floodFill (x-1, y, fillColor, oldColor)`
  - `floodFill (x, y-1, fillColor, oldColor)`

## Neighborhood



## 4-connected

- floodFill4 (x+1, y, fillColor, oldColor)
- floodFill4 (x, y+1, fillColor, oldColor)
- floodFill4 (x-1, y, fillColor, oldColor)
- floodFill4 (x, y-1, fillColor, oldColor)

## 8-connected

- floodFill8 (x+1, y, fillColor, oldColor)
- floodFill8 (x, y+1, fillColor, oldColor)
- floodFill8 (x-1, y, fillColor, oldColor)
- floodFill8 (x, y-1, fillColor, oldColor)
- floodFill8 (x+1, y+1, fillColor, oldColor)
- floodFill8 (x-1, y+1, fillColor, oldColor)
- floodFill8 (x-1, y-1, fillColor, oldColor)
- floodFill8 (x+1, y-1, fillColor, oldColor)

# Recursion

- Recursive calls to functions...
- When can it cause problems?
- Why?
- What can be done to prevent the problem?

Computer Graphics

Filled Area Primitives II

Lecture 09

Taqdees A. Siddiqi  
cs602@vu.edu.pk



# Computer Graphics

## Mathematical Fundamentals

### Lecture 10

Taqdees A. Siddiqi

cs602@vu.edu.pk

## Matrices and Simple Matrix Operations

- In many fields matrices are used to represent objects and operations on those objects.

# Topics

What a Matrix is?

Dimensions of a Matrix

Elements of a Matrix

Matrix Addition

Zero Matrix

Matrix Negation

Matrix Subtraction

Scalar multiplication of a matrix

The transpose of a matrix

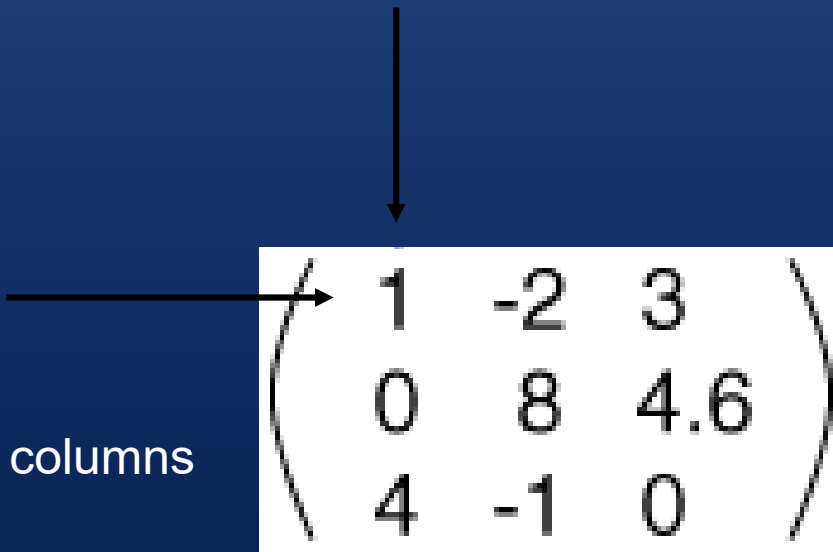
## Definition

A matrix is a collection of numbers arranged into a fixed number of rows and columns.

Example    Col 1

Row 1

rows X columns



A diagram illustrating a 3x3 matrix. The matrix is represented as  $\begin{pmatrix} 1 & -2 & 3 \\ 0 & 8 & 4.6 \\ 4 & -1 & 0 \end{pmatrix}$ . A vertical arrow points from the text 'Col 1' to the first column of the matrix. A horizontal arrow points from the text 'Row 1' to the first row of the matrix. The text 'rows X columns' is positioned to the left of the matrix.

## Element

Each number that makes up a matrix is called an **element** of the matrix. The elements in a matrix have specific locations.

## Matrix Dimensions

The numbers of rows and columns of a matrix are called its dimensions. Here is a matrix with three rows and two columns: (a 3 x 2 matrix)

$$\begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}_{3 \times 2}$$

4 x 1 matrix

$$\begin{pmatrix} 5.12 \\ -4.08 \\ 0.0 \\ 1.0 \end{pmatrix}_{4 \times 1}$$

2 x 3 matrix

$$\begin{pmatrix} -1 & 0 & 1 \\ 5 & 3 & 4 \end{pmatrix}_{2 \times 3}$$

Question: What do you think, a **square matrix** is? Here is an example:

$$\begin{pmatrix} 5 & 4 & 3 \\ -4 & 0 & 4 \\ 7 & 10 & 3 \end{pmatrix}$$

|

Answer: A matrix with the number of  
Rows == Columns

## Square Matrix

In a square matrix the number of rows equals the number of columns. In computer graphics, square matrices are used for transformations.

Column Matrix

Row Matrix

Column Vector and Row Vector

**Question: What are square matrices used for?**

**Answer:** Square matrices are used (in computer graphics) to represent geometric transformations.

## Names for Matrices

Usually a capital letter in bold face, like **A** or **M**.

Sometimes as a reminder the dimensions are written to the right of the letter, as in **B**<sub>3x3</sub>.

## Elements of a matrix

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

Question: (Thought Question:) If two matrices contain the same numbers as elements, are the two matrices equal to each other?

Answer: No, to be equal, two matrices must have the same dimensions, and must have the same values in the same positions.

## **Matrix Equality**

For two matrices to be equal, they must have:

1. The same dimensions.
2. Corresponding elements must be equal



In other words, say that  $A_{n \times m} = [a_{ij}]$  and that  $B_{p \times q} = [b_{ij}]$

Then  $A = B$  if and only if  $n=p$ ,  $m=q$ , and  $a_{ij}=b_{ij}$  for all  $i$  and  $j$  in range.

## Matrix Addition

Order

If **A** is an  $M \times N$  matrix, and **B** is also an  $M \times N$  matrix, then their sum is an  $M \times N$  matrix formed by adding corresponding elements of **A** and **B**

## Example

$$\begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix} + \begin{pmatrix} 0 & 2 \\ -1 & 2 \\ 1 & -6 \end{pmatrix} = \begin{pmatrix} 1 & 6 \\ 1 & 7 \\ 4 & 0 \end{pmatrix}$$

## Zero Matrix

A zero matrix is one; which has all its elements zero.  
Here is a 3x3 zero matrix

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}_{3 \times 3} = 0$$

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 1 & 10 & 7.5 \\ 2 & 9 & -3.2 \\ -2 & 0 & 5 \end{pmatrix} = ?$$

Question: What would be the result of above addition?

Answer: Of course, the sum is the same as the non-zero matrix.

## Rules for Matrix Addition

$$A + B = B + A$$

$$A + 0 = 0 + A = A$$

$$0 + 0 = 0$$

Question: Do you think that  $(A + B) + C = A + (B + C)$

Answer: Yes — this is another rule that works like real number math.

### Multiplication of a Matrix by a Scalar

To multiply a matrix by a scalar, multiply each element of the matrix by the scalar. Here is an example of this. (In this example, the variable  $a$  is a scalar.)

$$a \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} = \begin{pmatrix} 1a & 2a & 3a \\ 4a & 5a & 6a \\ 7a & 8a & 9a \end{pmatrix}$$

## Negative of a Matrix

The negation of a matrix is formed by negating each element of the matrix:

$$-\mathbf{A} = -1\mathbf{A}$$

Example

$$-1 \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} = \begin{pmatrix} -1 & -2 & -3 \\ -4 & -5 & -6 \\ -7 & -8 & -9 \end{pmatrix}$$

## Matrix Subtraction

If  $\mathbf{A}$  and  $\mathbf{B}$  have the same number of rows and columns, then  $\mathbf{A} - \mathbf{B}$  is defined as  $\mathbf{A} + (-\mathbf{B})$ . Usually you think of this as:

To form  $\mathbf{A} - \mathbf{B}$ , from each element of  $\mathbf{A}$  subtract the corresponding element of  $\mathbf{B}$ .

## Example

$$\begin{pmatrix} 5 & 4 & 3 \\ 4 & 0 & 4 \\ 7 & 10 & 3 \end{pmatrix} - \begin{pmatrix} 1 & -2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} = \begin{pmatrix} 4 & 6 & 0 \\ ? & -5 & -2 \\ 0 & 2 & ? \end{pmatrix}$$

## Transpose

The **transpose** of a matrix is a new matrix whose rows are the columns of the original (which makes its columns the rows of the original).

## Example

$$\begin{pmatrix} 5 & 4 & 3 \\ 4 & 0 & 4 \\ 7 & 10 & 3 \end{pmatrix}^T = \begin{pmatrix} 5 & 4 & 7 \\ 4 & 0 & 10 \\ 3 & 4 & 3 \end{pmatrix}$$

## A Rule for Transpose

If a transposed matrix is itself transposed, you get the original back:

$$\left( \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}^T \right)^T = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}^T = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

This illustrates the rule  $(\mathbf{A}^T)^T = \mathbf{A}$ .

Question: What is the transpose of:

$$(1 \quad 2 \quad 3 \quad 4)^T = ?$$

Answer:

$$(1 \quad 2 \quad 3 \quad 4)^T = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}$$

## Rule Summary

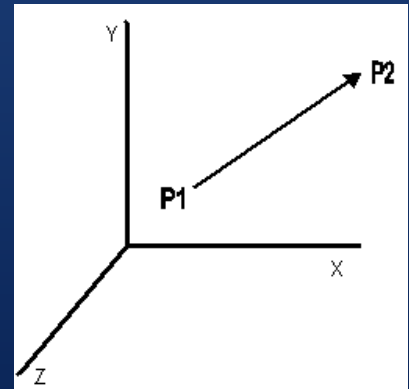
|                             |                                                     |                                                          |
|-----------------------------|-----------------------------------------------------|----------------------------------------------------------|
| $A + 0 = A$                 | $A + B = B + A$                                     | $0 + 0 = 0$                                              |
| $A + (B + C) = (A + B) + C$ | $(\underline{ab})A = \underline{a}(\underline{b}A)$ | $\underline{a}(A + B) = \underline{a}B + \underline{a}A$ |
| $\underline{a}0 = 0$        | $(\underline{-1})A = -A$                            | $A - A = 0$                                              |
| $(\underline{A^T})^T = A$   | $0^T = 0$                                           |                                                          |



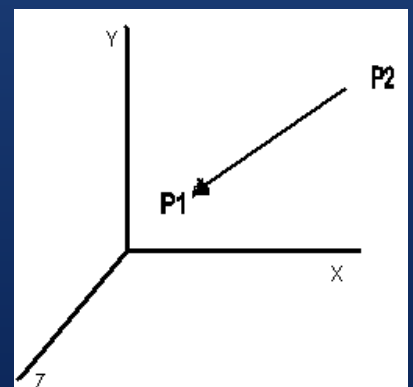
# Vectors

- Another important mathematical concept used in graphics is the Vector
- If  $P_1 = (x_1, y_1, z_1)$  is the starting point and  $P_2 = (x_2, y_2, z_2)$  is the ending point, then the vector

$$V = (x_2 - x_1, y_2 - y_1, z_2 - z_1)$$

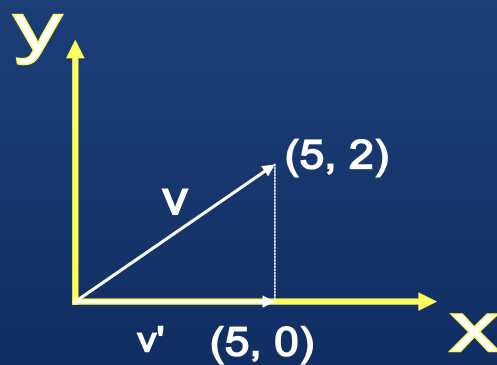


- and if  $P_2 = (x_2, y_2, z_2)$  is the starting point and  $P_1 = (x_1, y_1, z_1)$  is the ending point, then the vector  $V = (x_1 - x_2, y_1 - y_2, z_1 - z_2)$

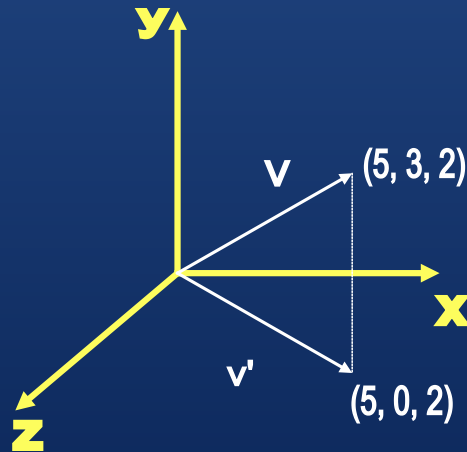


- This just defines length and direction, but not position

## Vector Projections



- Projection of  $\mathbf{v}$  onto the x-axis



- Projection of  $v$  onto the  $xz$  plane

## 2D Magnitude and Direction

- The magnitude (length) of a vector:

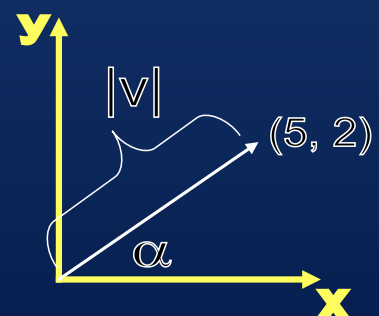
$$|V| = \text{sqrt}(V_x^2 + V_y^2)$$

- Derived from the Pythagorean theorem

- The direction of a vector:

$$\alpha = \tan^{-1}(V_y / V_x)$$

- a.k.a. arctan or atan
- atan vs. atan2
- $\alpha$  is angular displacement from the  $x$ -axis



# 3D Magnitude and Direction

- 3D magnitude is a simple extension of 2D

$$|V| = \sqrt{V_x^2 + V_y^2 + V_z^2}$$

- 3D direction is a bit harder than in 2D
  - Need 2 angles to fully describe direction
    - Latitude/longitude is a real-world example

- Direction Cosines are often used:

- $\alpha$ ,  $\beta$ , and  $\gamma$  are the positive angles that the vector makes with each of the positive coordinate axes x, y, and z, respectively

$$\cos \alpha = V_x / |V|$$

$$\cos \beta = V_y / |V|$$
$$= V_z / |V|$$

$$\cos \gamma$$

# Vector Normalization

- “Normalizing” a vector means shrinking or stretching it so its magnitude is 1
  - a.k.a. creating unit vectors
  - Note that this does not change the direction
- Normalize by dividing by its magnitude;  $V = (1, 2, 3)$   
 $|V| = \text{sqrt}(1^2 + 2^2 + 3^2)$   
 $= \text{sqrt}(14)$   
 $= 3.74$

# Vector Normalization

$$\begin{aligned} V_{\text{norm}} &= V / |V| = (1, 2, 3) / 3.74 \\ &= (1 / 3.74, 2 / 3.74, 3 / 3.74) \\ &= (.27, .53, .80) \end{aligned}$$

$$\begin{aligned} |V_{\text{norm}}| &= \text{sqrt}(.27^2 + .53^2 + .80^2) \\ &= \text{sqrt}(.9) = .95 \end{aligned}$$

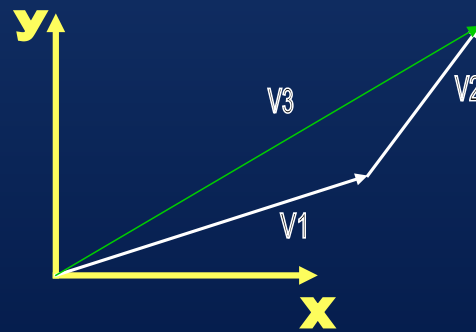
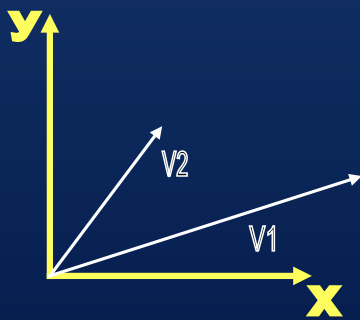
- Note that the last calculation doesn't come out to exactly 1. This is because of the error introduced by using only 2 decimal places in the calculations above.

# Vector Addition

- Equation:

$$\mathbf{V}_3 = \mathbf{V}_1 + \mathbf{V}_2 = (\mathbf{V}_{1x} + \mathbf{V}_{2x}, \mathbf{V}_{1y} + \mathbf{V}_{2y}, \mathbf{V}_{1z} + \mathbf{V}_{2z})$$

- Visually:

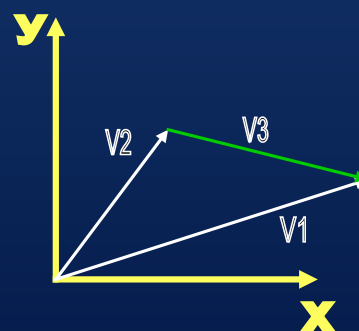
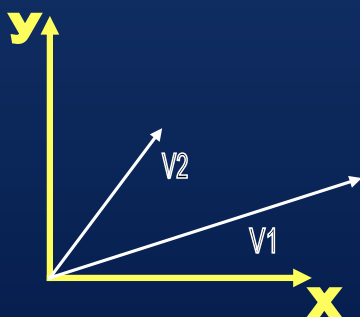


# Vector Subtraction

- Equation:

$$\mathbf{V}_3 = \mathbf{V}_1 - \mathbf{V}_2 = (\mathbf{V}_{1x} - \mathbf{V}_{2x}, \mathbf{V}_{1y} - \mathbf{V}_{2y}, \mathbf{V}_{1z} - \mathbf{V}_{2z})$$

- Visually:



# Dot Product

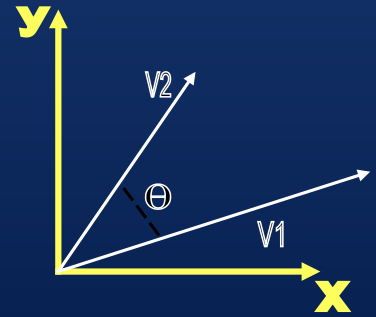
- The dot product of 2 vectors is a scalar

$$V_1 \cdot V_2 = (V_{1x} V_{2x}) + (V_{1y} V_{2y}) + (V_{1z} V_{2z})$$

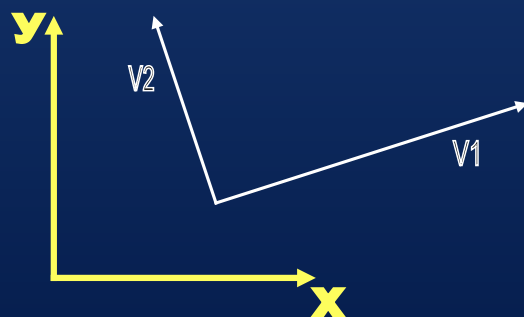
- Or, perhaps more importantly for graphics:

$$V_1 \cdot V_2 = |V_1| |V_2| \cos(\theta)$$

where  $\theta$  is the angle  
between the 2 vectors  
and  $\theta$  is in the range  
 $0 \leq \theta \leq \Pi$



- Why is dot product important for graphics?
  - It is zero iff the 2 vectors are perpendicular
    - $\cos(90) = 0$



- The Dot Product computation can be simplified when it is known that the vectors are unit vectors

$$\mathbf{V}_1 \cdot \mathbf{V}_2 = \cos(\theta)$$

because  $|\mathbf{V}_1|$  and  $|\mathbf{V}_2|$  are both 1

- Saves 6 squares, 4 additions, and 2 square roots

## Cross Product

- The cross product of 2 vectors is a vector

$$\mathbf{V}_1 \times \mathbf{V}_2 = ( \mathbf{V}_{1y} \mathbf{V}_{2z} - \mathbf{V}_{1z} \mathbf{V}_{2y} , \\ \mathbf{V}_{1z} \mathbf{V}_{2x} - \mathbf{V}_{1x} \mathbf{V}_{2z} , \\ \mathbf{V}_{1x} \mathbf{V}_{2y} - \mathbf{V}_{1y} \mathbf{V}_{2x} )$$

- Note that if you are into linear algebra there is also a way to do the cross product calculation using matrices and determinants



- Again, just as with the dot product, there is a more graphical definition:

$$\mathbf{V}_1 \times \mathbf{V}_2 = u |\mathbf{V}_1| |\mathbf{V}_2| \sin(\theta)$$

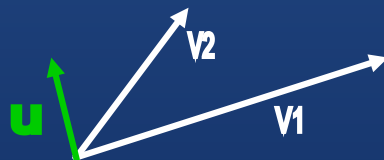
where  $\theta$  is the angle between the 2 vectors

and  $\theta$  is in the range  $0 \leq \theta \leq \Pi$

and  $u$  is the unit vector that is perpendicular to both vectors

– Why  $u$ ?

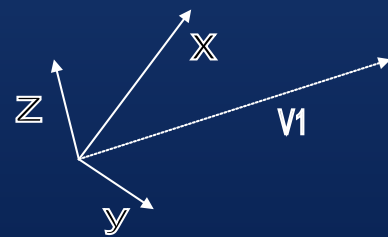
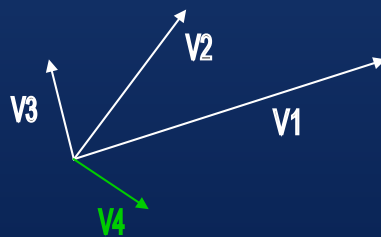
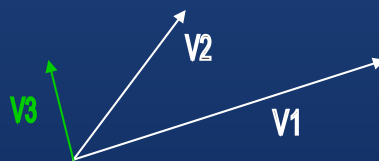
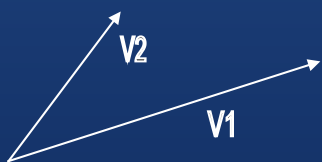
- $|\mathbf{V}_1| |\mathbf{V}_2| \sin(\theta)$  produces a scalar and the result needs to be a vector



- The direction of  $u$  is determined by the right hand rule
  - The perpendicular definition leaves an ambiguity in terms of the direction of  $u$
- Note that you can't take the cross product of 2 vectors that are parallel to each other
  - $\sin(0) = \sin(180) = 0 \rightarrow$  produces the vector  $(0, 0, 0)$

# Forming Coordinate Systems

- Cross products are great for forming coordinate system frames (3 vectors that are perpendicular to each other) from 2 random vectors
  - Cross  $V_1$  and  $V_2$  to form  $V_3$
  - $V_3$  is now perpendicular to both  $V_1$  and  $V_2$
  - Cross  $V_2$  and  $V_3$  to form  $V_4$
  - $V_4$  is now perpendicular to both  $V_2$  and  $V_3$
  - Then  $V_2$ ,  $V_4$ , and  $V_3$  form your new frame



- $V_1$  and  $V_2$  are in the new  $xy$  plane

# Computer Graphics

## Mathematical Fundamentals

### Lecture 10

Taqdees A. Siddiqi

cs602@vu.edu.pk

# Computer Graphics

## Lecture 11

### 2D Transformations I

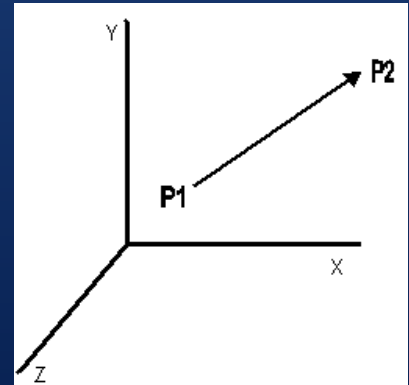
Taqdees A. Siddiqi

cs602@vu.edu.pk

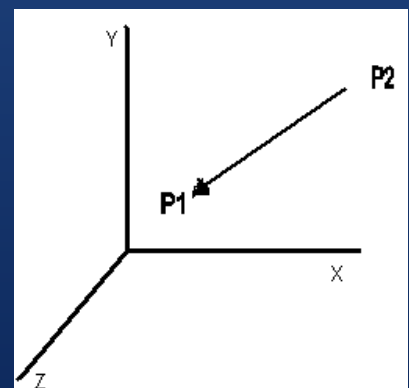
# Vectors

- ◆ Another important mathematical concept used in graphics is the Vector
- ◆ If  $P_1 = (x_1, y_1, z_1)$  is the starting point and  $P_2 = (x_2, y_2, z_2)$  is the ending point, then the vector

$$V = (x_2 - x_1, y_2 - y_1, z_2 - z_1)$$

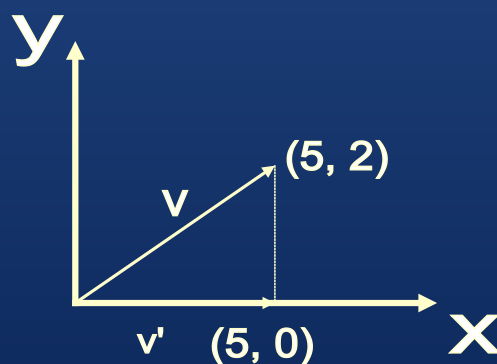


- ◆ and if  $P_2 = (x_2, y_2, z_2)$  is the starting point and  $P_1 = (x_1, y_1, z_1)$  is the ending point, then the vector  $V = (x_1 - x_2, y_1 - y_2, z_1 - z_2)$

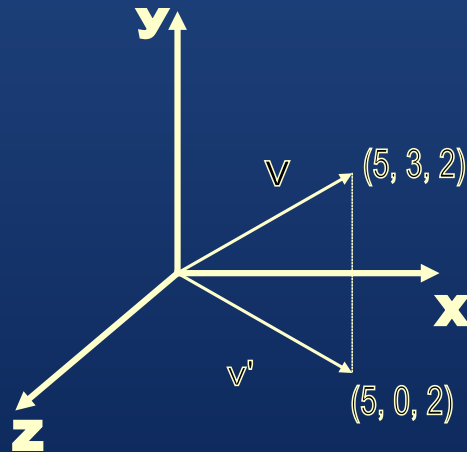


- ◆ This just defines *length and direction*, but *not position*

## Vector Projections



- ◆ Projection of  $\mathbf{v}$  onto the x-axis



- ◆ Projection of  $v$  onto the  $xz$  plane

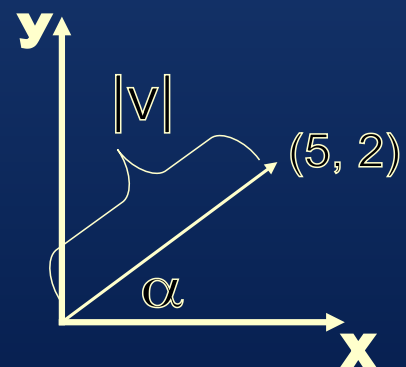
## 2D Magnitude and Direction

- ◆ The magnitude (length) of a vector:
  - Derived from the Pythagorean theorem
- ◆ The direction of a vector:

$$|v| = \sqrt{V_x^2 + V_y^2}$$

$$\alpha = \tan^{-1} (V_y / V_x)$$

$\forall \alpha$  is angular displacement  
from the  $x$ -axis



# 3D Magnitude and Direction

- ◆ 3D magnitude is a simple extension of 2D

$$|V| = \text{sqrt}(V_x^2 + V_y^2 + V_z^2)$$

- ◆ 3D direction is a bit harder than in 2D
  - Need 2 angles to fully describe direction
    - Latitude/longitude is a real-world example

- Direction Cosines are often used:

- $\alpha$ ,  $\beta$ , and  $\gamma$  are the positive angles that the vector makes with each of the positive coordinate axes x, y, and z, respectively

$$\cos \alpha = V_x / |V|$$

$$\cos \beta = V_y / |V|$$

$$\cos \gamma = V_z / |V|$$

# Vector Normalization

- ◆ “Normalizing” a vector means shrinking or stretching it so its magnitude is 1
  - creating unit vectors
  - Note that this does not change the direction
- ◆ Normalize by dividing by its magnitude;  $V = (1, 2, 3)$ 
$$\begin{aligned} |V| &= \text{sqrt}(1^2 + 2^2 + 3^2) \\ &= \text{sqrt}(14) \\ &= 3.74 \end{aligned}$$

## Vector Normalization

$$\begin{aligned} V_{\text{norm}} &= V / |V| = (1, 2, 3) / 3.74 \\ &= (1 / 3.74, 2 / 3.74, 3 / 3.74) \\ &= (.27, .53, .80) \end{aligned}$$

$$\begin{aligned} |V_{\text{norm}}| &= \text{sqrt}(.27^2 + .53^2 + .80^2) \\ &= \text{sqrt}(.9) = .95 \end{aligned}$$

- ◆ Note that the last calculation doesn't come out to exactly 1. This is because of the error introduced by using only 2 decimal places in the calculations above.

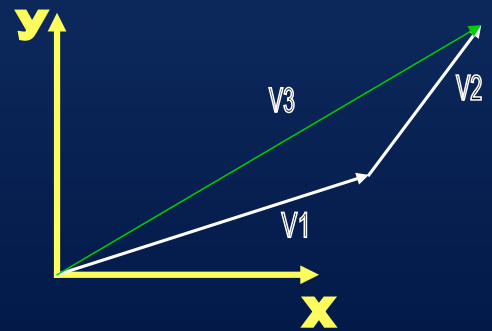
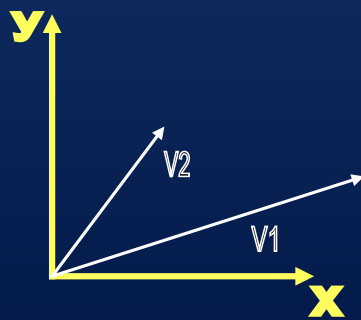


# Vector Addition

- Equation:

$$\begin{aligned} \mathbf{V}_3 &= \mathbf{V}_1 + \mathbf{V}_2 \\ &= (V_{1x} + V_{2x}, V_{1y} + V_{2y}, V_{1z} + V_{2z}) \end{aligned}$$

- Visually:

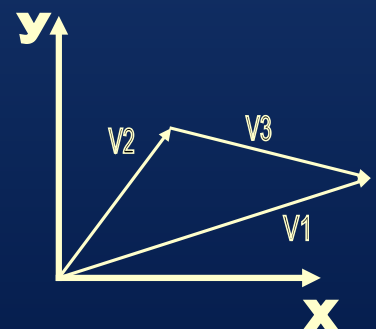
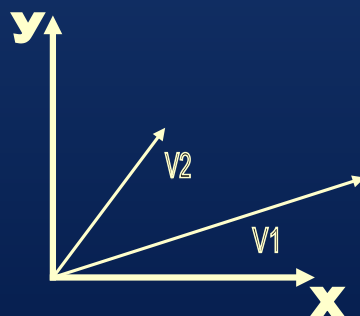


# Vector Subtraction

- Equation:

$$\begin{aligned} \mathbf{V}_3 &= \mathbf{V}_1 - \mathbf{V}_2 \\ &= (V_{1x} - V_{2x}, V_{1y} - V_{2y}, V_{1z} - V_{2z}) \end{aligned}$$

- Visually:

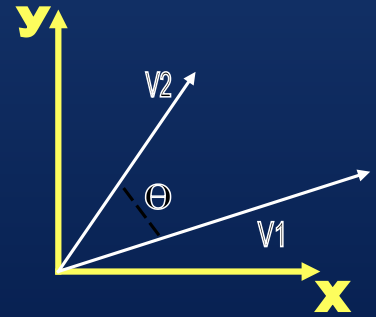


# Dot Product

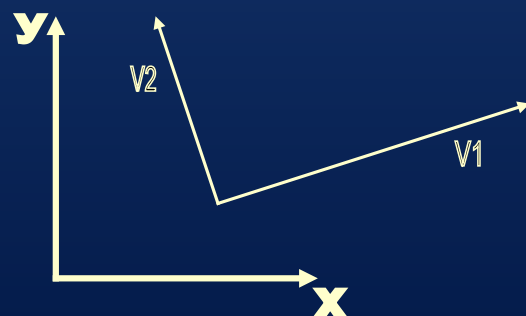
- ◆ The dot product of 2 vectors is a scalar
$$V_1 \cdot V_2 = (V_{1x} V_{2x}) + (V_{1y} V_{2y}) + (V_{1z} V_{2z})$$
- ◆ Or, perhaps more importantly for graphics:

$$V_1 \cdot V_2 = |V_1| |V_2| \cos(\theta)$$

where  $\theta$  is the angle  
between the 2 vectors  
and  $\theta$  is in the range  
 $0 \leq \theta \leq \Pi$



- ◆ Why is dot product important for graphics?
  - It is zero iff the 2 vectors are perpendicular
    - $\cos(90) = 0$



- ◆ The Dot Product computation can be simplified when it is known that the vectors are unit vectors

$$V_1 \cdot V_2 = \cos(\theta)$$

because  $|V_1|$  and  $|V_2|$  are both 1

- Saves 6 squares, 4 additions, and 2 square roots

## Cross Product

- ◆ The cross product of 2 vectors is a vector

$$V_1 \times V_2 = ( V_{1y} V_{2z} - V_{1z} V_{2y}, \\ V_{1z} V_{2x} - V_{1x} V_{2z}, \\ V_{1x} V_{2y} - V_{1y} V_{2x} )$$

- Note that if you are into linear algebra there is also a way to do the cross product calculation using matrices and determinants

- ◆ Again, just as with the dot product, there is a more graphical definition:

$$\mathbf{V}_1 \times \mathbf{V}_2 = u |\mathbf{V}_1| |\mathbf{V}_2| \sin(\theta)$$

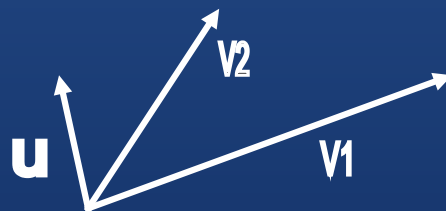
where  $\theta$  is the angle between the 2 vectors

and  $\theta$  is in the range  $0 \leq \theta \leq \Pi$

and  $u$  is the unit vector that is perpendicular to both vectors

– Why  $u$ ?

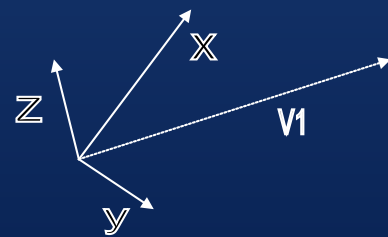
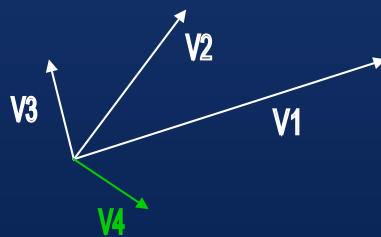
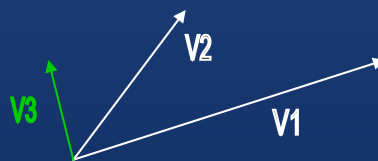
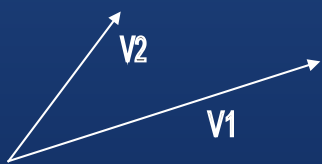
- $|\mathbf{V}_1| |\mathbf{V}_2| \sin(\theta)$  produces a scalar and the result needs to be a vector



- ◆ The direction of  $u$  is determined by the right hand rule
  - The perpendicular definition leaves an ambiguity in terms of the direction of  $u$
- ◆ Note that you can't take the cross product of 2 vectors that are parallel to each other
  - $\sin(0) = \sin(180) = 0 \rightarrow$  produces the vector  $(0, 0, 0)$

# Forming Coordinate Systems

- ◆ Cross products are great for forming coordinate system frames (3 vectors that are perpendicular to each other) from 2 random vectors
  - Cross  $V_1$  and  $V_2$  to form  $V_3$
  - $V_3$  is now perpendicular to both  $V_1$  and  $V_2$
  - Cross  $V_2$  and  $V_3$  to form  $V_4$
  - $V_4$  is now perpendicular to both  $V_2$  and  $V_3$
  - Then  $V_2$ ,  $V_4$ , and  $V_3$  form your new frame



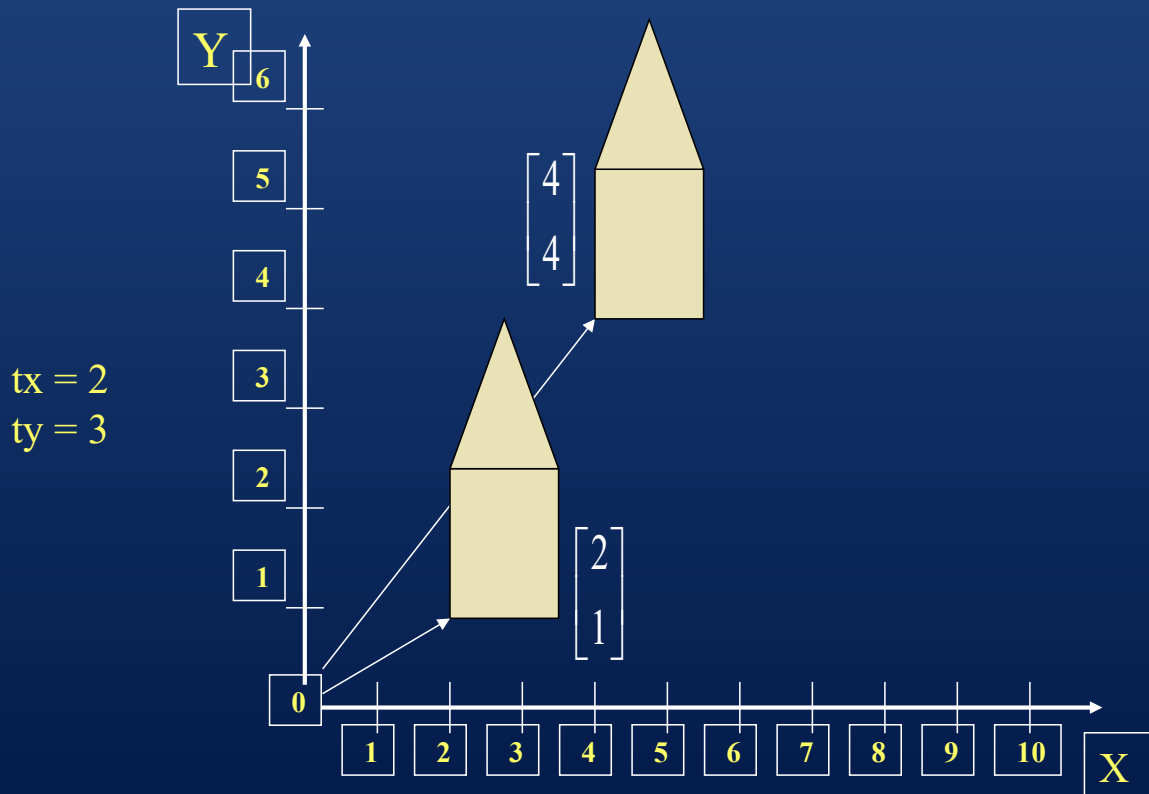
- ◆  $V_1$  and  $V_2$  are in the new  $xy$  plane

# Basic Transformations

- ◆ Translation
- ◆ Rotation
- ◆ Scaling

- ◆ Translation
  - Straight line movement
- ◆ Rotation
  - Circular movement w.r.t pivot
- ◆ Scaling
  - Change of size

# Translation



- ◆ Translation Distances  $t_x, t_y$
- ◆ For point  $P(x,y)$  after translation we have  $P'(x',y')$
- ◆  $x' = x + t_x, \quad y' = y + t_y$
- ◆  $(t_x, t_y)$  is Translation vector

Now  $x' = x + t_x$  ,  $y' = y + t_y$  can be expressed as a single matrix equation:

$$P' = P + T$$

Where  $P =$

$$P' = \quad T =$$

$$\begin{bmatrix} x \\ y \end{bmatrix}$$

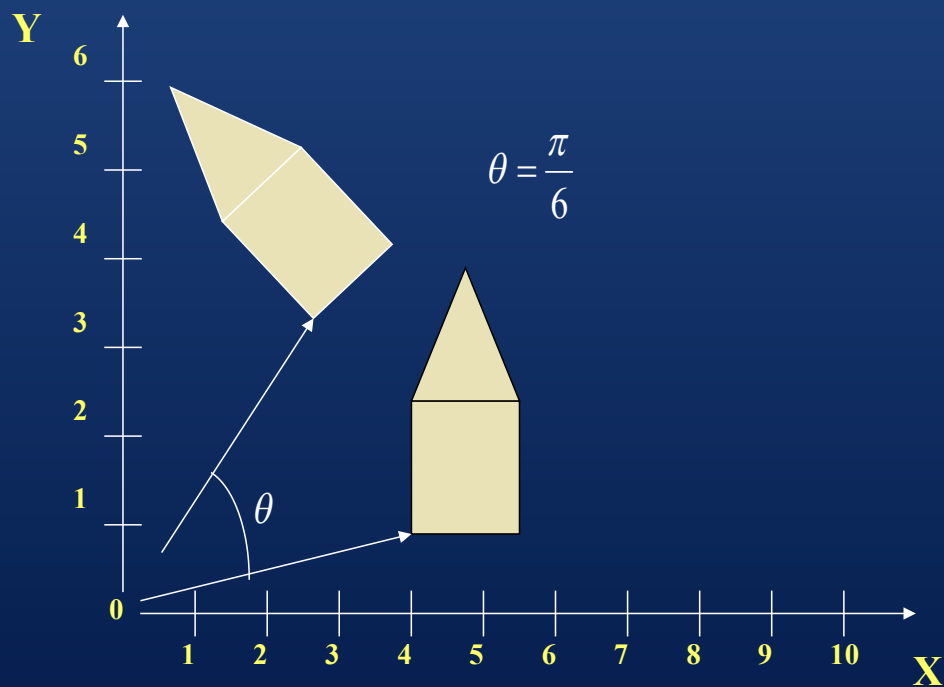
$$\begin{bmatrix} x' \\ y' \end{bmatrix}$$

$$\begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

- ◆ Rigid Body Transformation
- ◆ Objects are moved without deformation
- ◆ Every point on the object is translated by the same amount
- ◆ Typically all the endpoints are translated and object is redrawn using new endpoint positions



# Rotation



- ◆ Repositions an object along a circular path in xy-plane
- ◆ Rotation Angle  $\theta$
- ◆ Rotation Point  $(x_r, y_r)$
- ◆  $\theta$  is +ve  $\rightarrow$  counterclockwise rotation
- ◆  $\theta$  is -ve  $\rightarrow$  clockwise rotation
- ◆  $\theta$  is zero  $\rightarrow$  ?

- ◆ For simplicity, consider the pivot at origin and rotate point  $P(x, y)$   
where  $x = r \cos\Phi$  and  $y = r \sin\Phi$
- ◆ If rotated by  $\theta$  then:
- ◆  $x' = r \cos(\Phi + \theta)$   
 $= r \cos\Phi \cos\theta - r \sin\Phi \sin\theta$   
and
- ◆  $y' = r \sin(\Phi + \theta)$   
 $= r \cos\Phi \sin\theta + r \sin\Phi \cos\theta$

- ◆ Replacing  $r \cos\Phi$  with  $x$  and  $r \sin\Phi$  with  $y$ , we have:
- ◆  $x' = x \cos\theta - y \sin\theta$   
and
- ◆  $y' = x \sin\theta + y \cos\theta$

- ◆ Column vector representation:

$$P' = R \cdot P$$

- ◆ Where

$$P' = \begin{bmatrix} x' \\ y' \end{bmatrix}$$

$$R = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}$$

$$P = \begin{bmatrix} x \\ y \end{bmatrix}$$

- ◆ For Row vector representation of P and P', in order to get the same results:

- ◆ Which is actually transpose of original R, therefore:

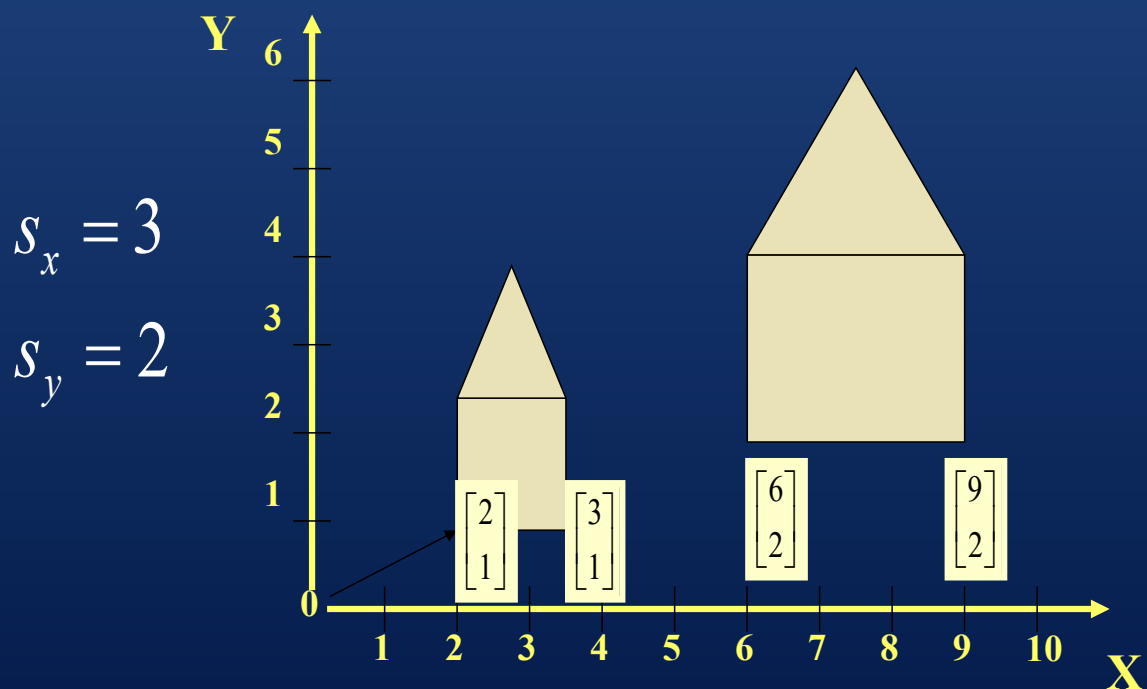
$$\begin{aligned} P'^T &= (R \cdot P)^T \\ &= P^T \cdot R^T \end{aligned}$$

$$R = \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix}$$

- ◆ When rotating about  $(x_r, y_r)$
- ◆  $x' = x_r + (x - x_r) \cos\theta - (y - y_r) \sin\theta$
- ◆  $y' = y_r + (x - x_r) \sin\theta - (y - y_r) \cos\theta$
- ◆ A better way of expressing these as matrix operations would be discussed later

- ◆ Rigid Body Transformation
- ◆ Objects are rotated without deformation
- ◆ Every point on the object is rotated by the same angle
- ◆ Typically all the endpoints are rotated and object is redrawn using new endpoint positions

# Scaling



# Scaling

- ◆ Scaling alters the size of an object
- ◆ Scaling factors  $S_x$  and  $S_y$  are used
- ◆ For polygons, coordinates of each vertex may be multiplied by  $S_x$  &  $S_y$  to produce the transformed coordinates:

$$x' = x.S_x$$

$$y' = y.S_y$$

In matrix form it can be expressed as :

$$P' = S.P$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} S_x & 0 \\ 0 & S_y \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

Scaling factor  $> 1$

Scaling factor  $< 1$

Scaling factor  $= 1$

Same valued scaling factors  $\rightarrow$  uniform scaling in x and y

Different valued scaling factors  $\rightarrow$  differential scaling in x and y

When scaling w.r.t. origin, both the size and distance from origin get scaled.

Size and distance from origin:

for Scaling factor 2

(double the size and double the distance from origin.)

for Scaling factor 0.5

(reduce the size to 50% and also the distance from origin to 50%.

This can be controlled by scaling w.r.t. a fixed point  $(x_f, y_f)$

- $x' = x_f + (x - x_f)S_x$

- $y' = y_f + (y - y_f)S_y$

- ◆ These can be rewritten as:

- $x' = x \cdot S_x + x_f (1 - S_x)$

- $y' = y \cdot S_y + y_f (1 - S_y)$

- Where the terms

$x_f (1 - S_x)$  and  $y_f (1 - S_y)$  are constant for all points in the object

# Computer Graphics

## Lecture 11

### 2D Transformations I

Taqdees A. Siddiqi  
cs602@vu.edu.pk

# Computer Graphics

## Lecture 12

### 2D Transformations II

Taqdees A. Siddiqi  
cs602@vu.edu.pk



# Homogeneous Coordinates

Translation:

$$P' = P + T$$

Rotation:

$$P' = R . P$$

Scaling:

$$P' = S . P$$

## Homogeneous Coordinates

From our last discussion we know:

$$P' = M_1 . P + M_2$$

With coordinate positions  $P$  and  $P'$  represented as column vectors

Matrix  $M_1$  is a 2x2 array containing multiplicative factors

$M_2$  is a 2 element column matrix containing translation factors

Can we find a way to eliminate the matrix addition associated with translation?

Yes, we can. But for that  $M_1$  will have to be rewritten as a 3x3 matrix and also the coordinate positions will have to be expressed as a homogeneous coordinate triple:  $(x, y)$  as  $(x_h, y_h, h)$  where

$$x = \frac{x_h}{h}, \quad y = \frac{y_h}{h}$$

For simplicity, 1 is used as value for  $h$  in affine transformations, thus

$(x, y)$  has homogeneous coordinates as  $(x, y, 1)$

# Translation with Homogeneous Coordinates

The translation can now be expressed using homogeneous coordinates as:

Abbreviated as

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

# Rotation with Homogeneous Coordinates

The rotation can now be expressed using homogeneous coordinates as:

Abbreviated as:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$
$$P' = R(\theta) \cdot P$$

# Scaling with Homogeneous Coordinates

The scaling can now be expressed using homogeneous coordinates as:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Abbreviated as:

$$P' = S(S_x, S_y) \cdot P$$

## Composite Transformations

Two successive translations

Translation vectors  $(t_{x1}, t_{y1})$  and  $(t_{x2}, t_{y2})$

$$\begin{aligned} P' &= T(t_{x2}, t_{y2}) \cdot \{T(t_{x1}, t_{y1}) \cdot P\} \\ &= \{T(t_{x2}, t_{y2}) \cdot T(t_{x1}, t_{y1})\} \cdot P \end{aligned}$$

Where P and P' are homogeneous column vectors.

$$\begin{bmatrix} 1 & 0 & t_{x2} \\ 0 & 1 & t_{y2} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & t_{x1} \\ 0 & 1 & t_{y1} \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_{x1} + t_{x2} \\ 0 & 1 & t_{y1} + t_{y2} \\ 0 & 0 & 1 \end{bmatrix}$$

# Composite Translations

Which means that

$$T(t_{x2}, t_{y2}) \cdot T(t_{x1}, t_{y1}) = T(t_{x1} + t_{x2}, t_{y1} + t_{y2})$$

hence

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_{x1} + t_{x2} \\ 0 & 1 & t_{y1} + t_{y2} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

# Composite Rotations

Two successive Rotations applied to a point P produce:

$$\begin{aligned} P' &= R(\theta_2) \cdot \{R(\theta_1) \cdot P\} \\ &= \{R(\theta_2) \cdot R(\theta_1)\} \cdot P \end{aligned}$$

But since,

$$R(\theta_2) \cdot R(\theta_1) = R(\theta_1 + \theta_2)$$

Therefore,

$$P' = R(\theta_1 + \theta_2) \cdot P$$

# Composite Scalings

Two successive scaling operations would produce:

$$\begin{bmatrix} S_{x2} & 0 & 0 \\ 0 & S_{y2} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} S_{x1} & 0 & 0 \\ 0 & S_{y1} & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} S_{x1} \cdot S_{x2} & 0 & 0 \\ 0 & S_{y1} \cdot S_{y2} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

## General Pivot Point Rotation

- General pivot point  $(x_r, y_r)$
- Translate-Rotate-Translate
- Translate the object to coincide the pivot with the origin
- Rotate about the origin
- Translate back to the original pivot position

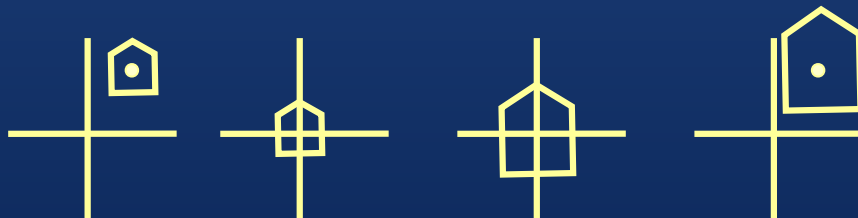


$$\begin{aligned}
 & \begin{bmatrix} 1 & 0 & x_r \\ 0 & 1 & y_r \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_r \\ 0 & 1 & -y_r \\ 0 & 0 & 1 \end{bmatrix} \\
 &= \begin{bmatrix} \cos \theta & -\sin \theta & x_r(1 - \cos \theta) + y_r \sin \theta \\ \sin \theta & \cos \theta & y_r(1 - \cos \theta) - x_r \sin \theta \\ 0 & 0 & 1 \end{bmatrix}
 \end{aligned}$$

$$T(x_r, y_r) \cdot R(\theta) \cdot T(-x_r, -y_r) = R(x_r, y_r, \theta)$$

# General Fixed Point Scaling

- ◆ General fixed point  $(x_f, y_f)$
- ◆ Translate-Scale-Translate
- ◆ Translate the object to coincide the fixed point with the origin
- ◆ Scale w.r.t. origin
- ◆ Translate the object back to the original position





$$\begin{bmatrix} 1 & 0 & x_f \\ 0 & 1 & y_f \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_f \\ 0 & 1 & -y_f \\ 0 & 0 & 1 \end{bmatrix} \\
 = \begin{bmatrix} S_x & 0 & x_f(1-S_x) \\ 0 & S_y & y_f(1-S_y) \\ 0 & 0 & 1 \end{bmatrix}$$

$$T(x_f, y_f) \cdot S(s_x, s_y) \cdot T(-x_f, -y_f) = S(x_f, y_f, s_x, s_y)$$

## General Composite Transformations and Computational Efficiency

- ◆ Translations-Rotations-Scalings

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} rS_{xx} & rS_{xy} & trs_x \\ rS_{yx} & rS_{yy} & trs_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

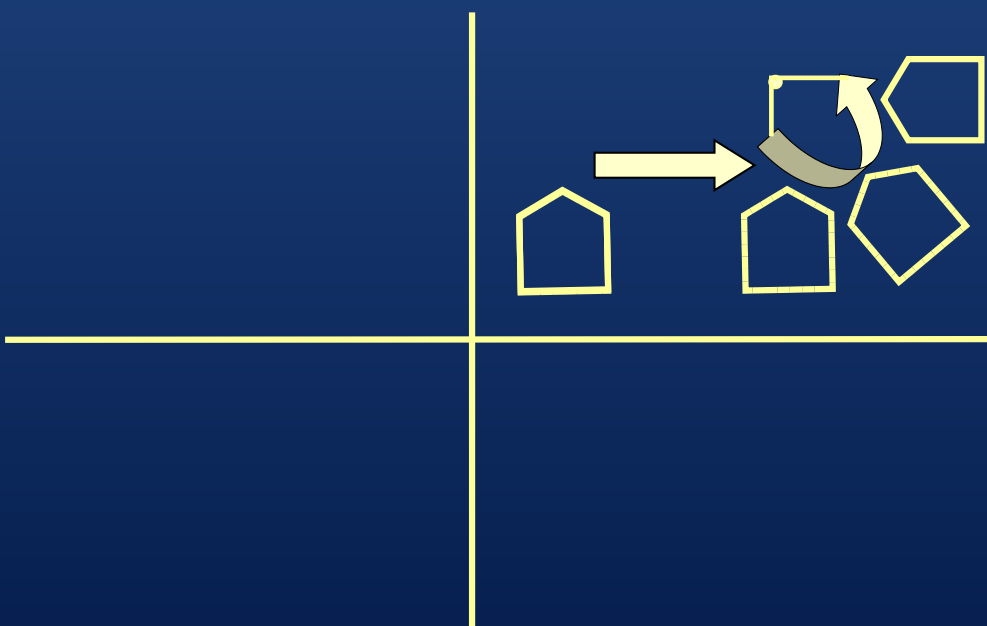
- ◆  $rs_{ij}$  are Rotation-Scaling terms
- ◆  $trs_i$  are the translational terms
- ◆ It takes 9 multiplications and 6 additions to evaluate but explicit calculations for transformed-coords:
- ◆  $x' = x.rs_{xx} + y.rs_{xy} + trs_x$
- ◆  $y' = x.rs_{yx} + y.rs_{yy} + trs_y$
- ◆ i.e. 4 multiplications and 4 additions

- ◆ If an object is to be scaled and rotated about its centroid coordinates  $(x_c, y_c)$  and then translated, the composite transformation matrix will be:
- ◆  $T(t_x, t_y) \cdot R(x_c, y_c) \cdot S(x_c, y_c, s_x, s_y)$

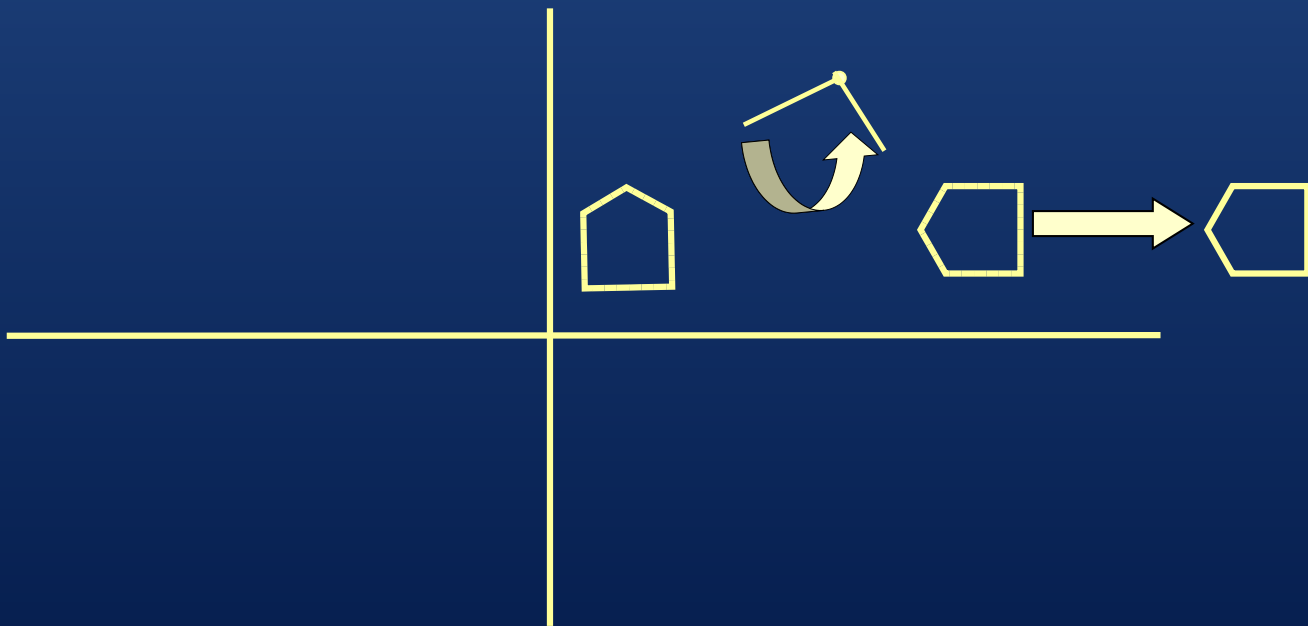
◆  $T(t_x, t_y) \cdot R(x_c, y_c) \cdot S(x_c, y_c, s_x, s_y)$

$$= \begin{bmatrix} S_x \cos \theta & -S_y \sin \theta & x_c(1 - S_x \cos \theta) + y_c S_y \sin \theta + t_x \\ S_x \sin \theta & S_y \cos \theta & y_c(1 - S_y \cos \theta) - x_c S_x \sin \theta + t_y \\ 0 & 0 & 1 \end{bmatrix}$$

**Order is Important**



# Reverse Order – Changed Position

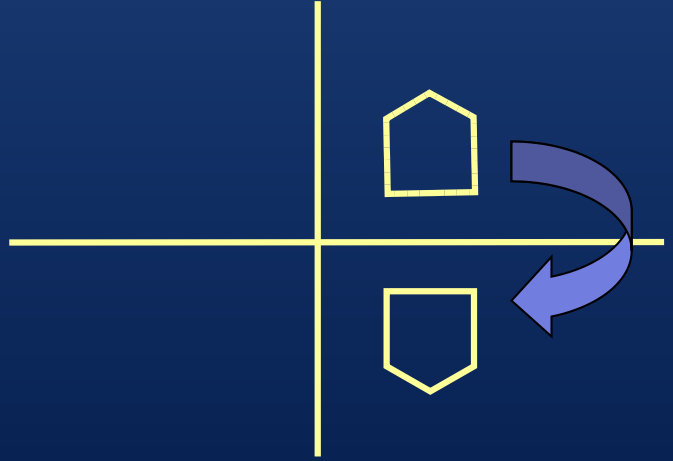


## Reflection

- ◆ Mirror image of object relative to axis of reflection
- ◆ By rotating the object about axis of reflection by  $180^\circ$

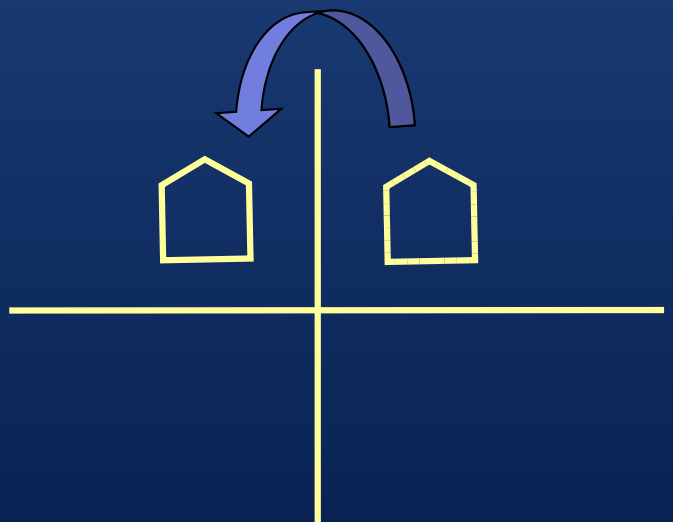
## Reflection about x-axis

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



## Reflection about y-axis

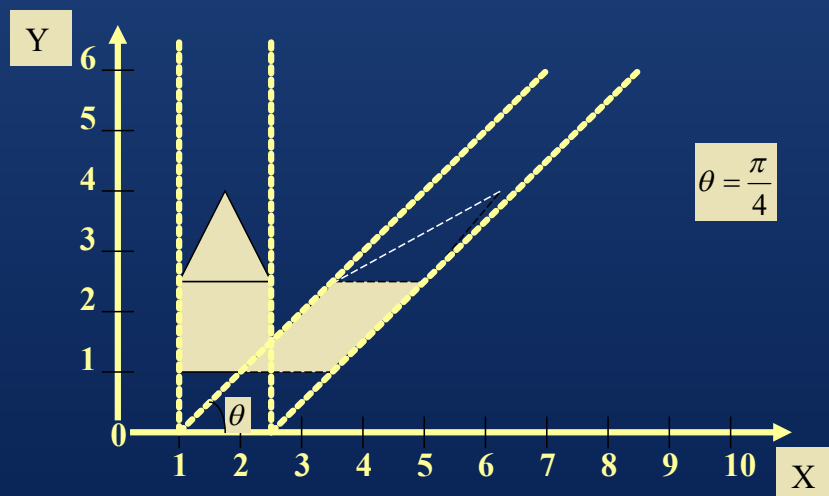
$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



# Shear

- ◆ Distorts the shape
- ◆ Effect is, of layers sliding off each other
- ◆ x direction shear
- ◆ y direction shear

## Shear



## x direction Shear

$$\begin{bmatrix} 1 & sh_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- ◆  $x' = x + sh_x \cdot y$
- ◆  $y' = y$

## y direction Shear

$$\begin{bmatrix} 1 & 0 & 0 \\ sh_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- ◆  $x' = x$
- ◆  $y' = sh_y \cdot x + y$

## x and y direction Shear

$$\begin{bmatrix} 1 & sh_x & 0 \\ sh_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- ◆  $x' = x + sh_x \cdot y$
- ◆  $y' = sh_y \cdot x + y$

## Computer Graphics

### Lecture 12

## 2D Transformations II

Taqdees A. Siddiqi  
[cs602@vu.edu.pk](mailto:cs602@vu.edu.pk)



# Computer Graphics

## Lecture 13

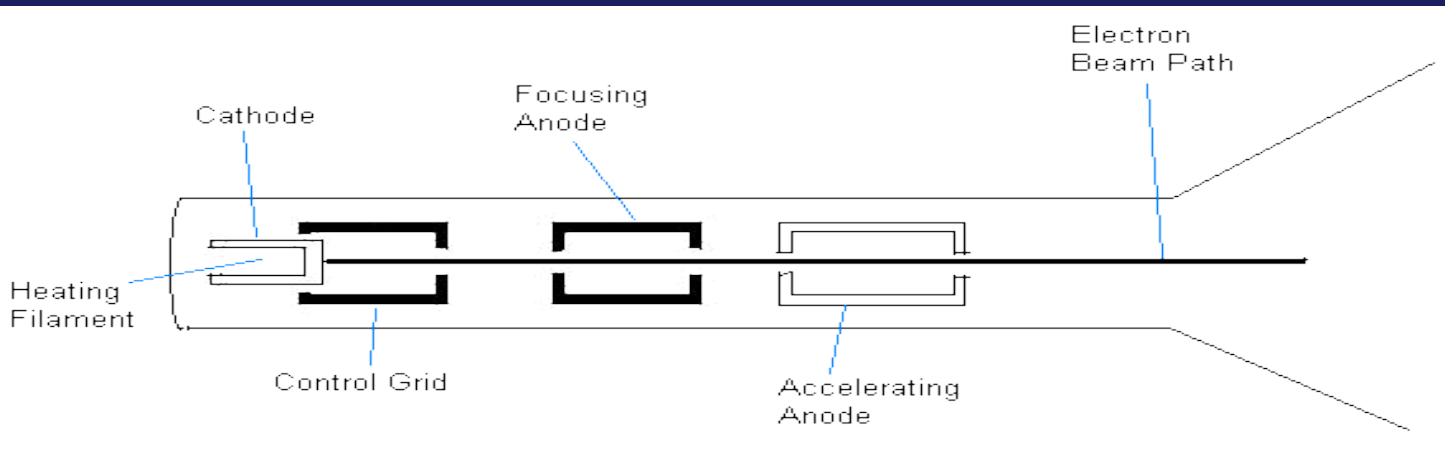
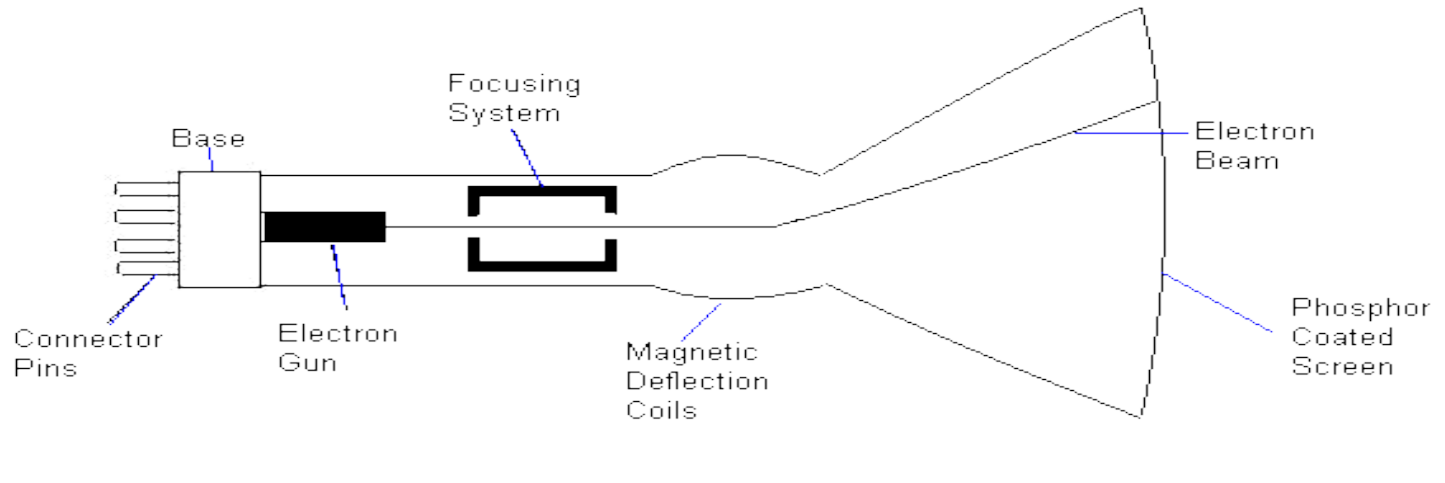
### Graphics Systems

Taqdees A. Siddiqi  
[taqdees@vu.edu.pk](mailto:taqdees@vu.edu.pk)

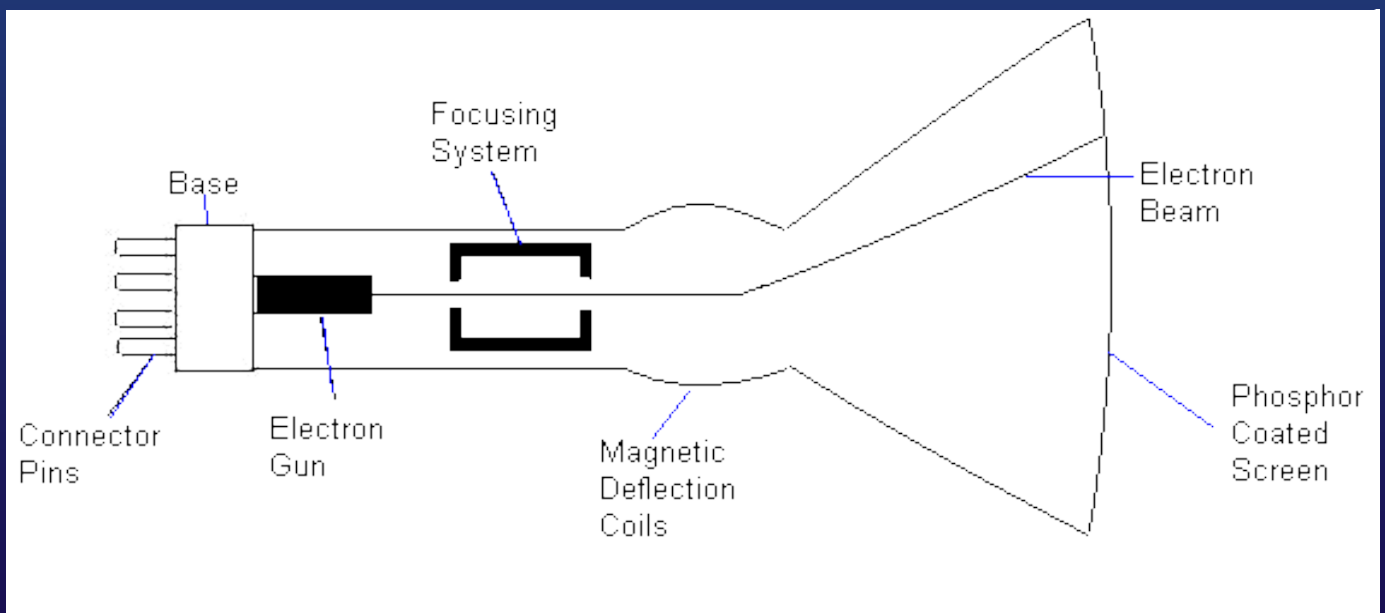
## Video Display Devices

- ◆ Refresh Cathode-Ray Tubes
- ◆ Raster-Scan Displays
- ◆ Random-Scan Displays
- ◆ Color CRT Monitors
- ◆ Direct-View Storage Tubes
- ◆ Flat-Panel Displays
- ◆ Three-Dimensional Viewing Devices

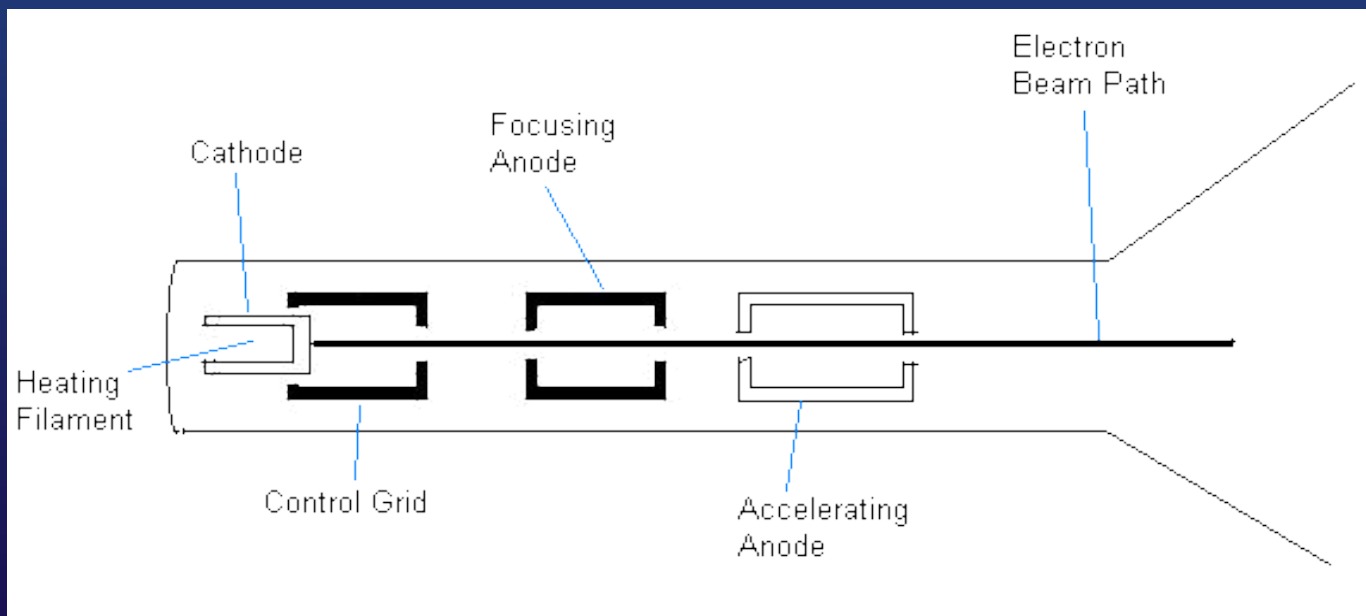
# Refresh Cathode-Ray Tubes



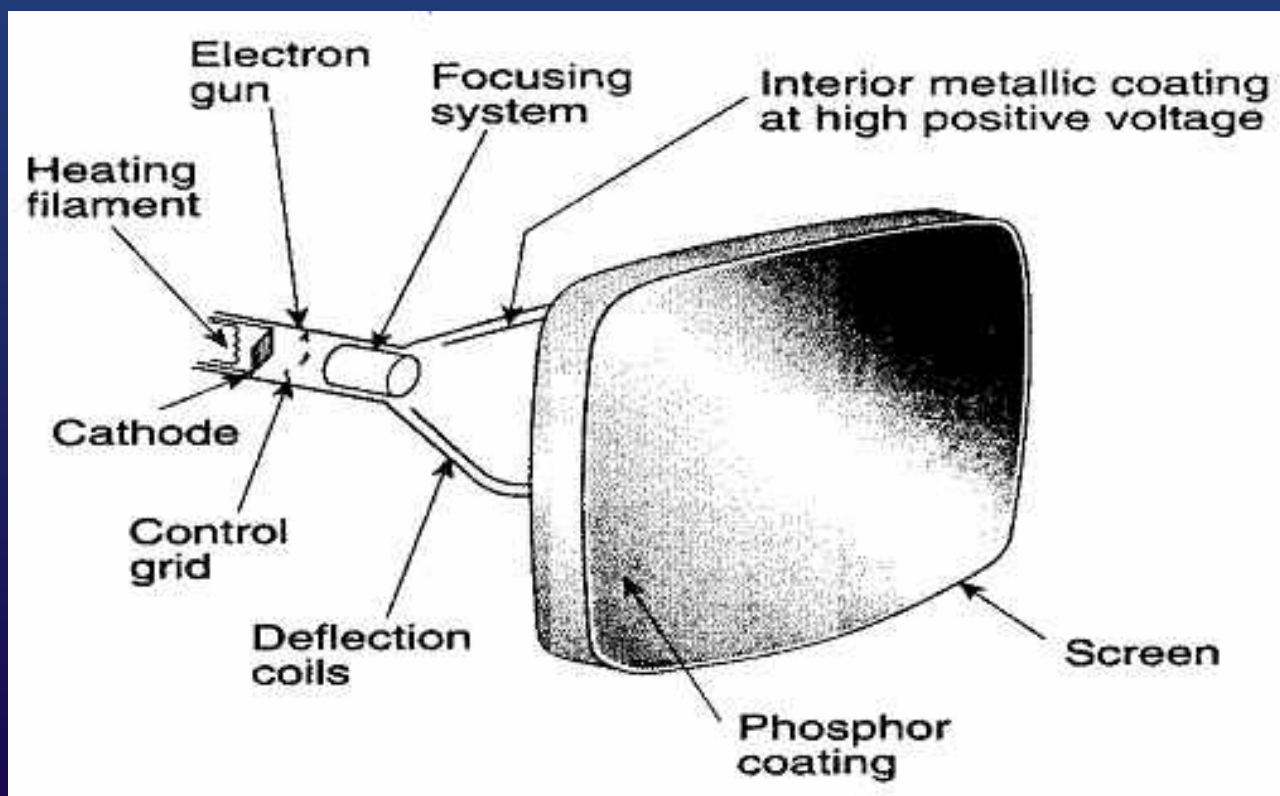
# Refresh Cathode-Ray Tubes



# Refresh Cathode-Ray Tubes



## Refresh Cathode-Ray Tubes Cont..



# Refresh Cathode-Ray Tubes

- ◆ Refresh CRT
- ◆ Control Grid
- ◆ Focusing System
- ◆ Focusing in High Precision System
- ◆ Construction of Cathode-Ray Tubes
- ◆ Persistence
- ◆ Resolution
- ◆ Aspect Ratio

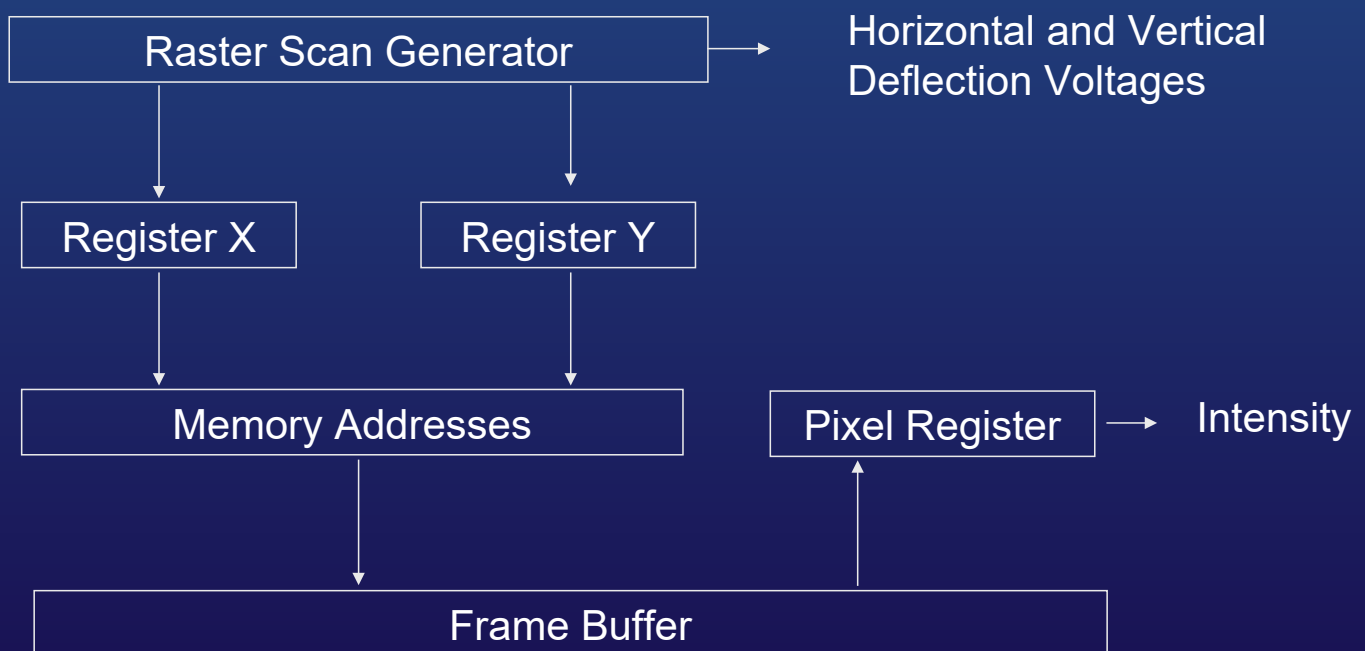
## Refresh Cathode-Ray Tubes



# Area Filling

- ◆ Scan\_line Fill
- ◆ Boundary Fill
- ◆ Flood Fill

## Video Controller

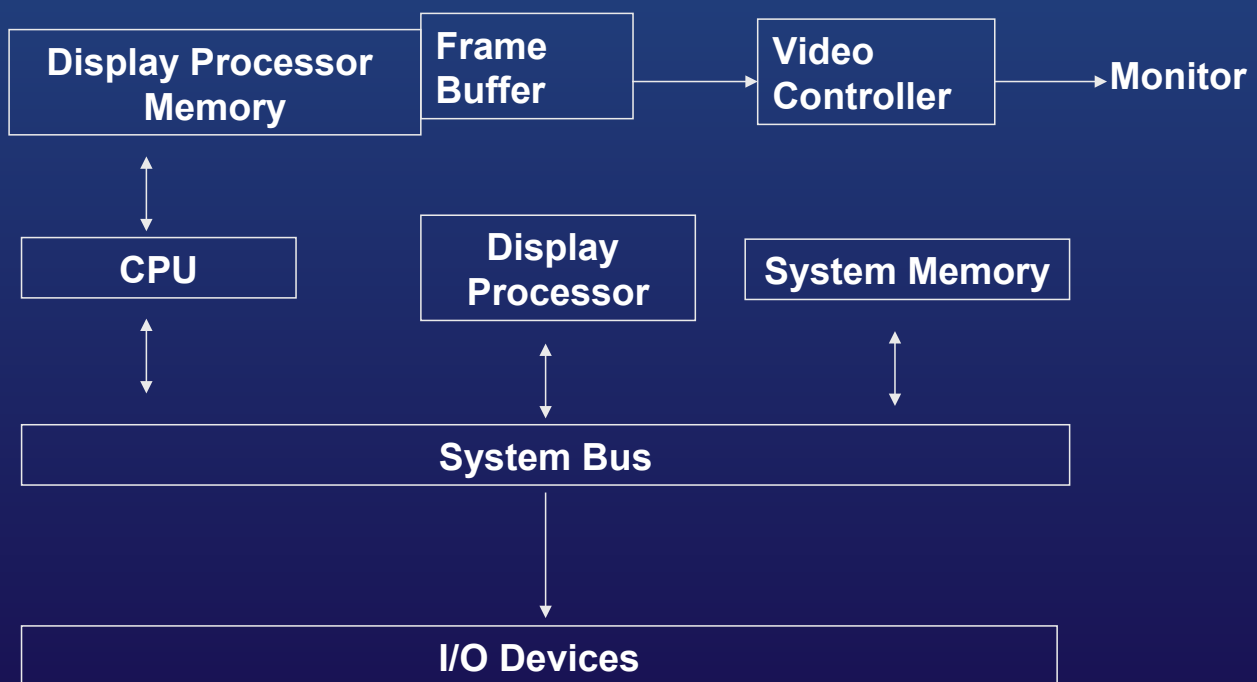


Basic Video Controller Refresh  
Operations

# Graphics Card or Display Adaptors

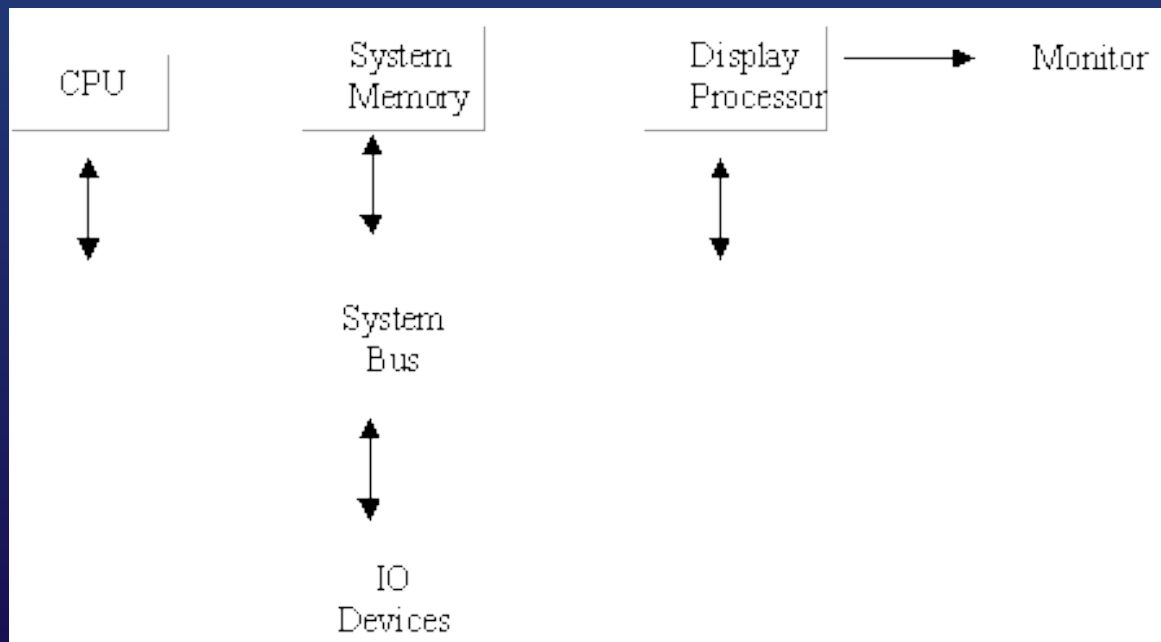
- ♦ A number of display Adaptors are available with varying capabilities specially Intel systems support following Adaptors:
- ♦ Monochrome Adaptor (MA)
- ♦ Hercules Adaptor (HA)
- ♦ Color Graphics Adaptor (CGA)
- ♦ Enhanced Graphics Adaptor (EGA)
- ♦ Multicolor Graphics Adaptor (MCGA)
- ♦ Video Graphics Adaptor (VGA)
- ♦ Super Video Graphics Adaptor (SVGA)
- ♦ Extended Graphics Adaptor (XGA)

## Raster Scan Display Processor



Architecture of a raster graphics systems with a display processor

# Random-Scan Systems



Architecture of a simple random scan system

## Graphics Card or Display Adaptors

- ◆ A number of display Adaptors are available with varying capabilities specially Intel systems support following Adaptors:
- ◆ Monochrome Adaptor (MA)
- ◆ Hercules Adaptor (HA)
- ◆ Color Graphics Adaptor (CGA)
- ◆ Enhanced Graphics Adaptor (EGA)
- ◆ Multicolor Graphics Adaptor (MCGA)
- ◆ Video Graphics Adaptor (VGA)
- ◆ Super Video Graphics Adaptor (SVGA)
- ◆ Extended Graphics Adaptor (XGA)

# Graphics Card or Display Adaptors

A video card is typically an adaptor, a removable expansion card in the PC. Thus, it can be replaced!

A video display Adaptor which is the special printed circuit board that plugs into one of the several expansion slots present on the mother board of the computer. A video display Adaptor is referred to as a video card as well.

The video card can also be an integral part of the system board; this is the case in certain brands of PCs and is always the case in laptops and clear preference for the replaceable video card in some PCs.

## Source Code

Code in assembly language that can set graphics mode 19 (13h)

```
MOV AH,0
```

```
MOV AL,13h ;mode number from 0-19
```

```
INT 10h
```



## Source Code Cont..

You can use this for assembler or you can embed this code in C language using 'asm' keyword

```
asm {
 MOV AH,0
 MOV AL,13h;;mode number from 0-19
 INT 10h
}
```

## Source Code

Code in assembly language that can set graphics mode 19 (13h)

```
MOV AH,0
MOV AL,13h ;mode number from 0-19
INT 10h
```

# Source Code for Writing Pixel

Code to write pixel using video BIOS interrupt 10h and service number 0ch.

```
MOV AH,0Ch
MOV AL,COLOR_NUM
MOV BH,0
MOV CX,ROW_NUM
MOV DX,COLUMN_NUM
INT 10h
```

## Incremental Line Algorithm

This algorithm exploits simple line equation

$$y = m x + b$$

where  $m = dy / dx$

and  $b = y - m x$

# DDA Algorithm

DDA abbreviated for digital differential analyzer has a very simple technique.

Find difference, dx and dy as:

$$dy = y_2 - y_1$$

$$dx = x_2 - x_1$$

if  $|dx| > |dy|$  then

$step = |dx|$

else

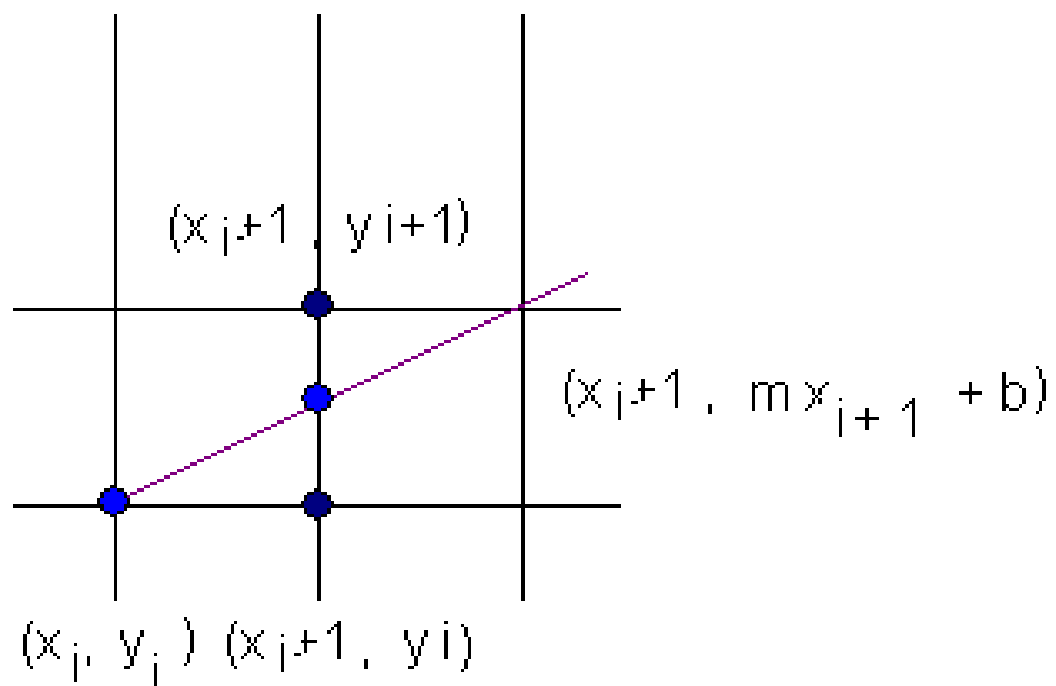
$step = |dy|$

Now very simple to say that step is the total number of pixels required for a line.

Next step is to find xIncrement and yIncrement:

$$\text{xIncrement} = dx/\text{step}$$

$$\text{yIncrement} = dy/\text{step}$$



# Circle Drawing Using Cartesian Coordinates

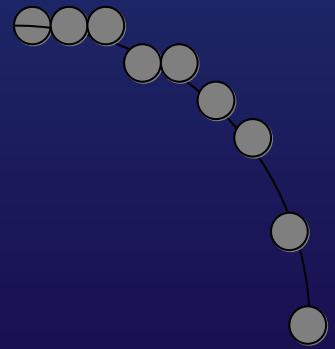
This technique uses the equation for a circle with radius  $r$  centered at  $(0,0)$ :

$$x^2 + y^2 = r^2,$$

an obvious choice is to plot

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

against different values of  $x$ .



## Polar Coordinates Algorithm

Circle2 (xcenter, ycenter, radius)

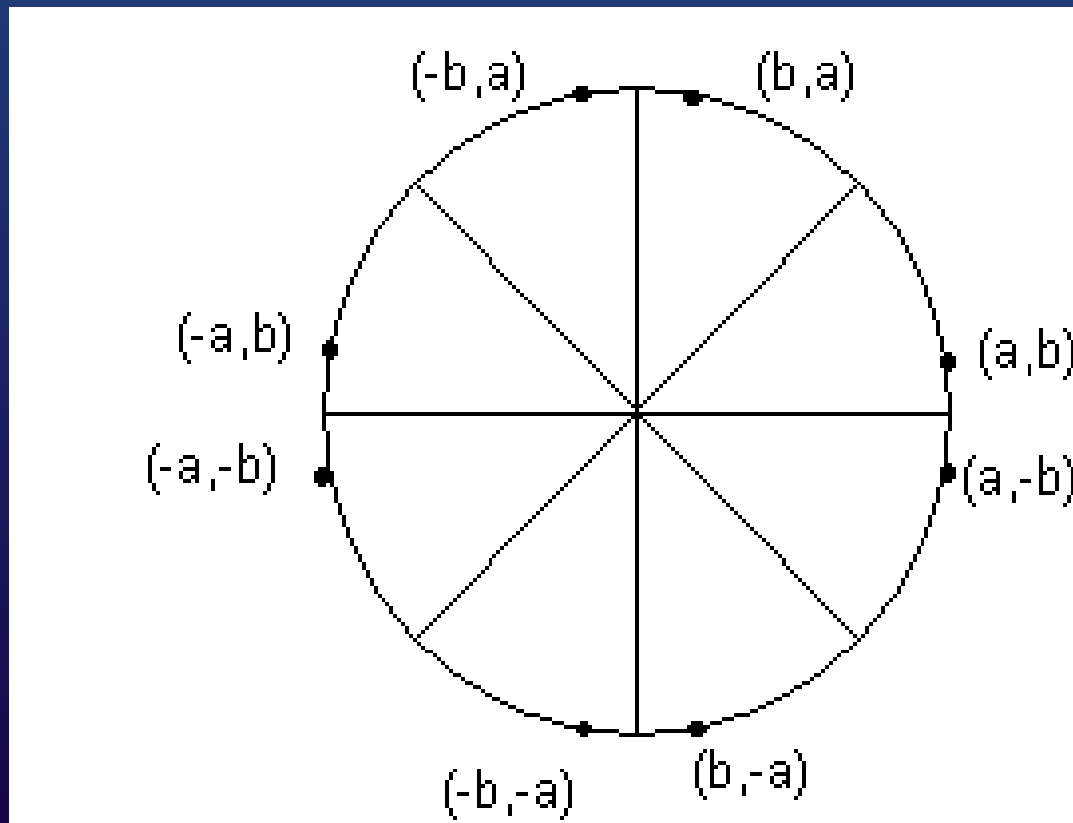
for  $\theta = 0$  to  $2\pi$  step  $1/r$

$$x = x_c + r * \cos \theta$$

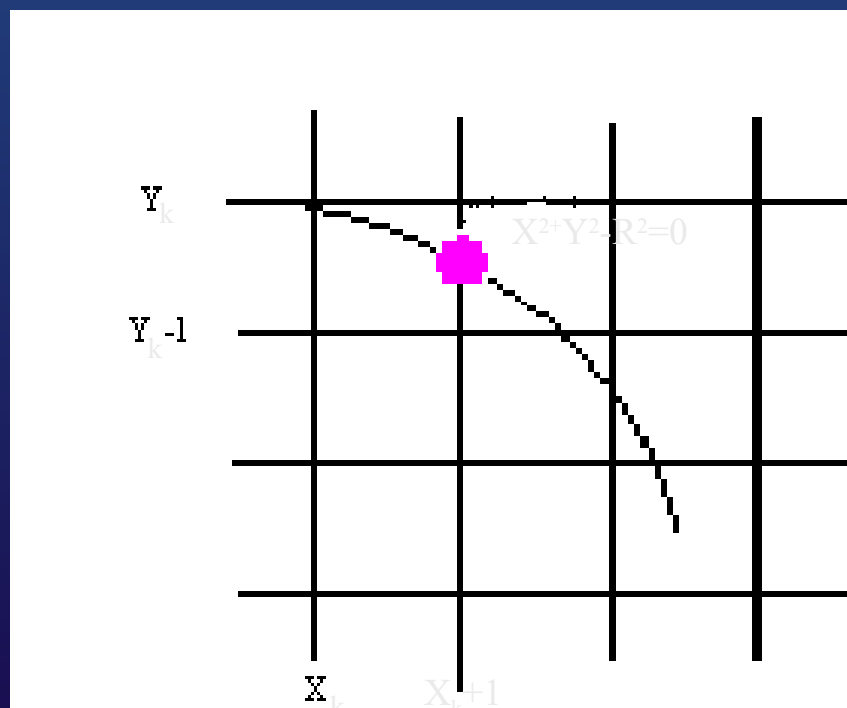
$$y = y_c + r * \sin \theta$$

drawPixel (x, y)

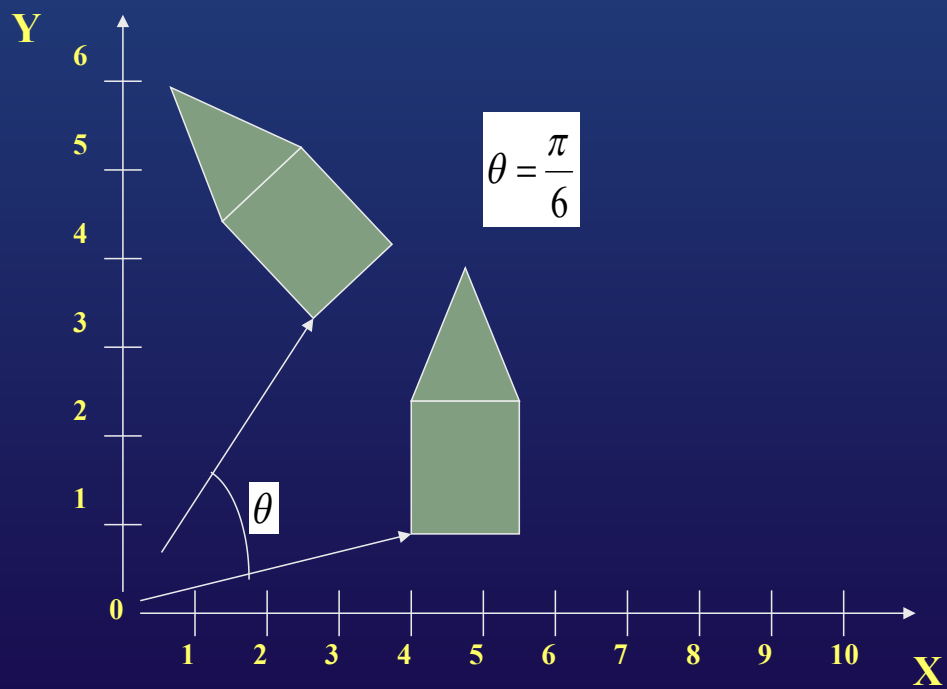
# Eight Octants Symmetry



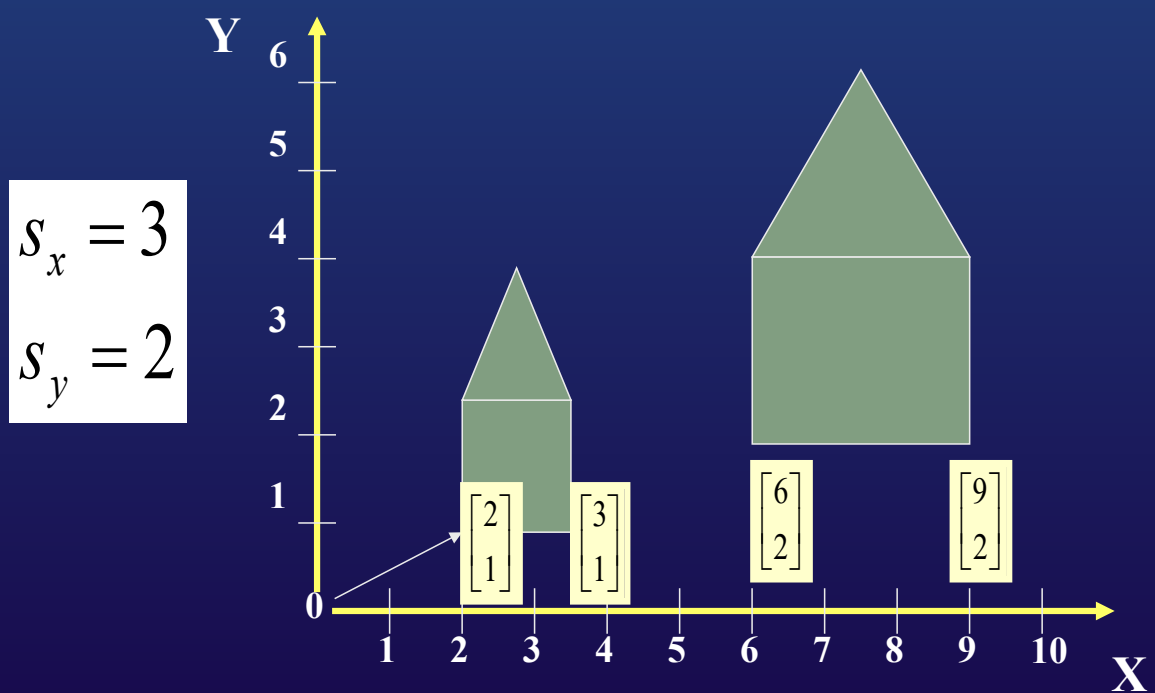
# Midpoint Circle Algorithm



# Rotation



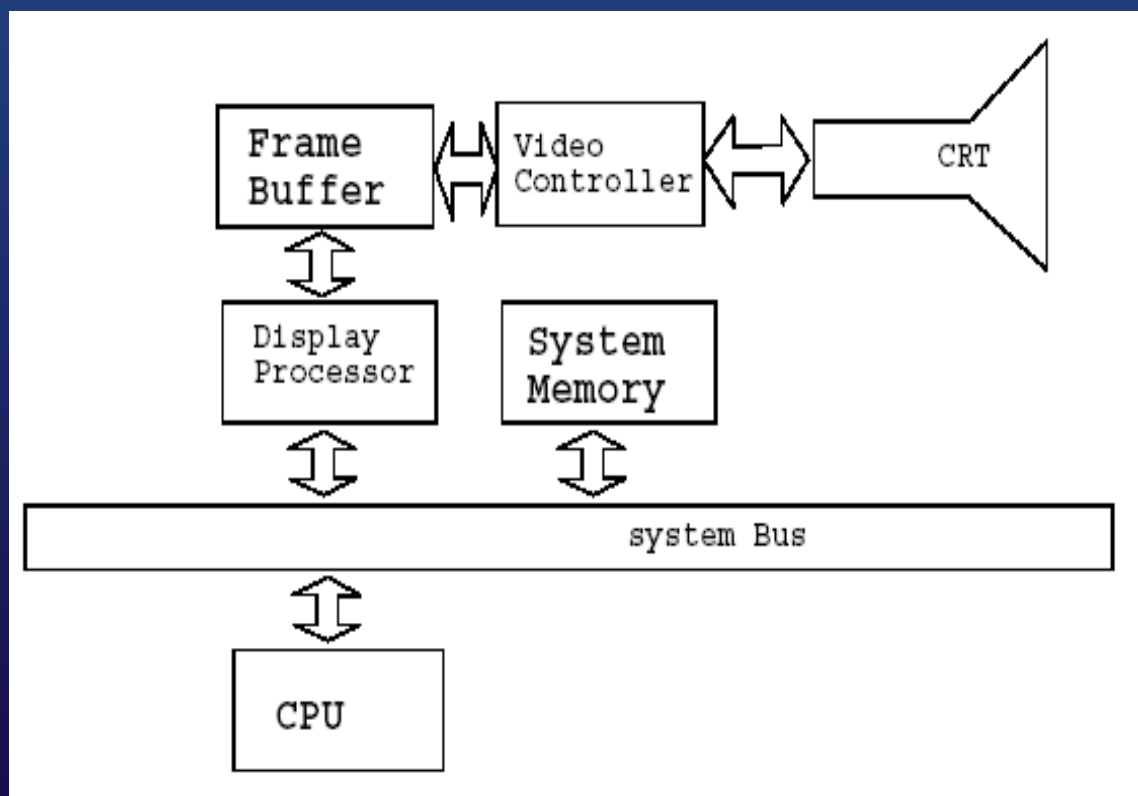
# Scaling



# Basic Transformations

- ♦ Transformation
- ♦ Rotation
- ♦ Scaling

## Video Controller

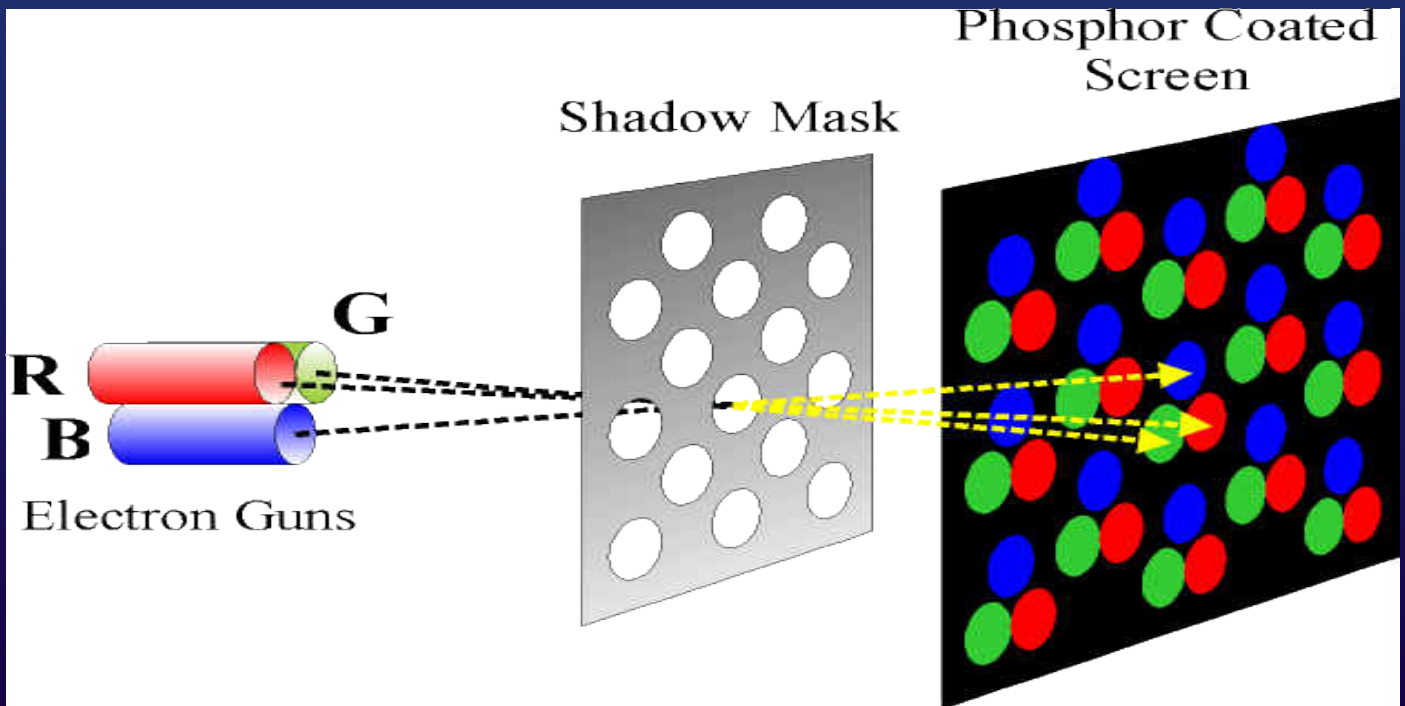


Architecture of a raster system with a fixed portion of system memory reserved for the frame buffer.

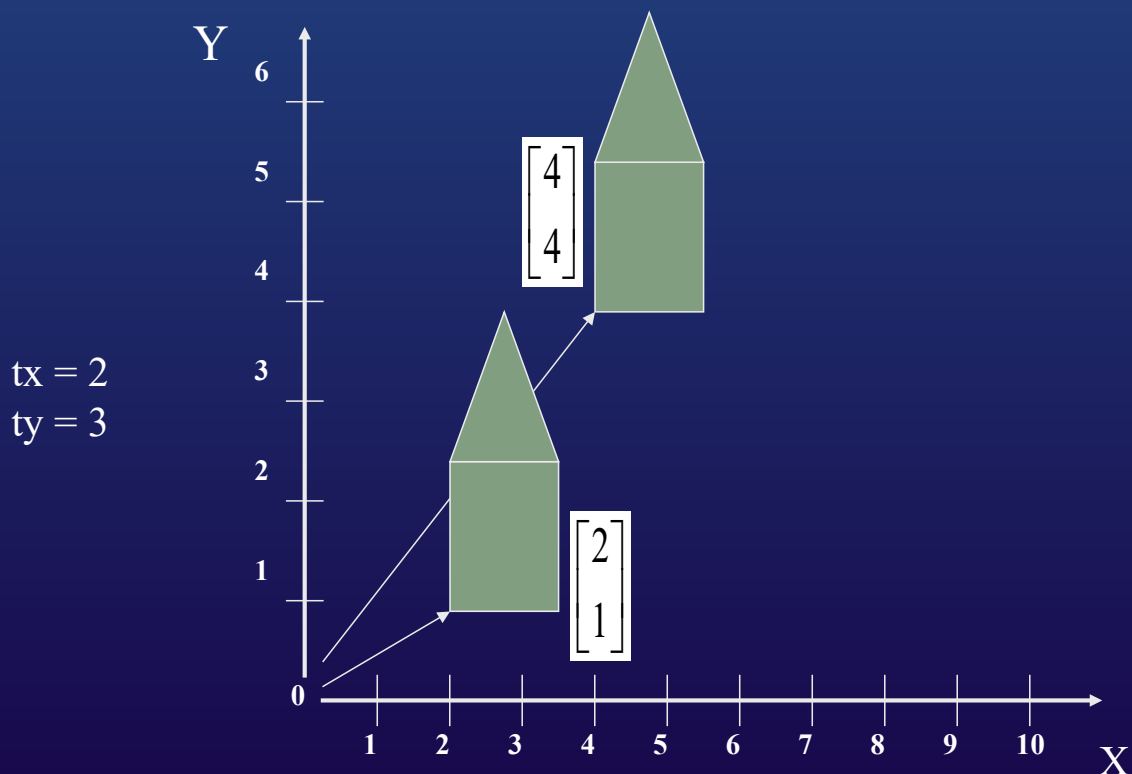


# Color CRT Monitors

- ◆ Beam Penetration Method
- ◆ Shadow Mask Method



## Translation



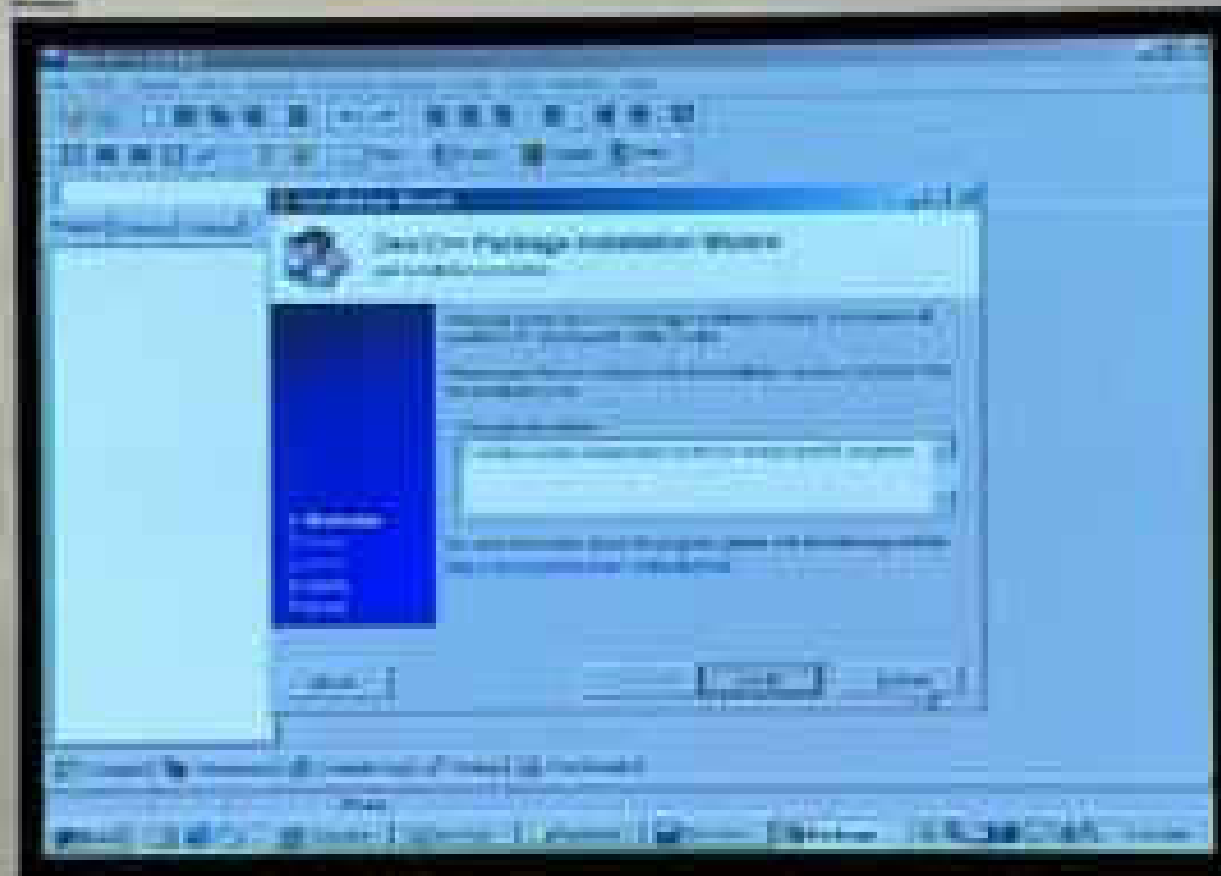
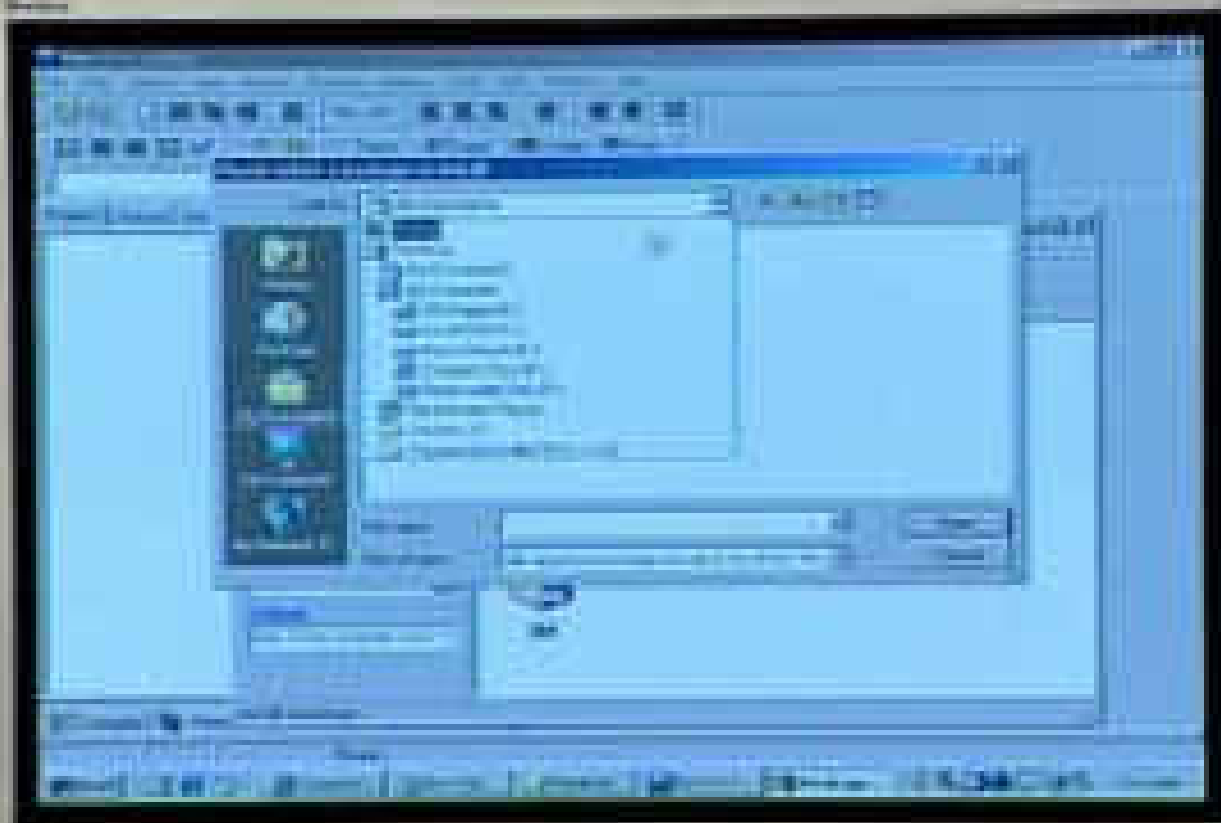
# Color CRT Monitors

- ◆ Beam Penetration Method
- ◆ Shadow Mask Method

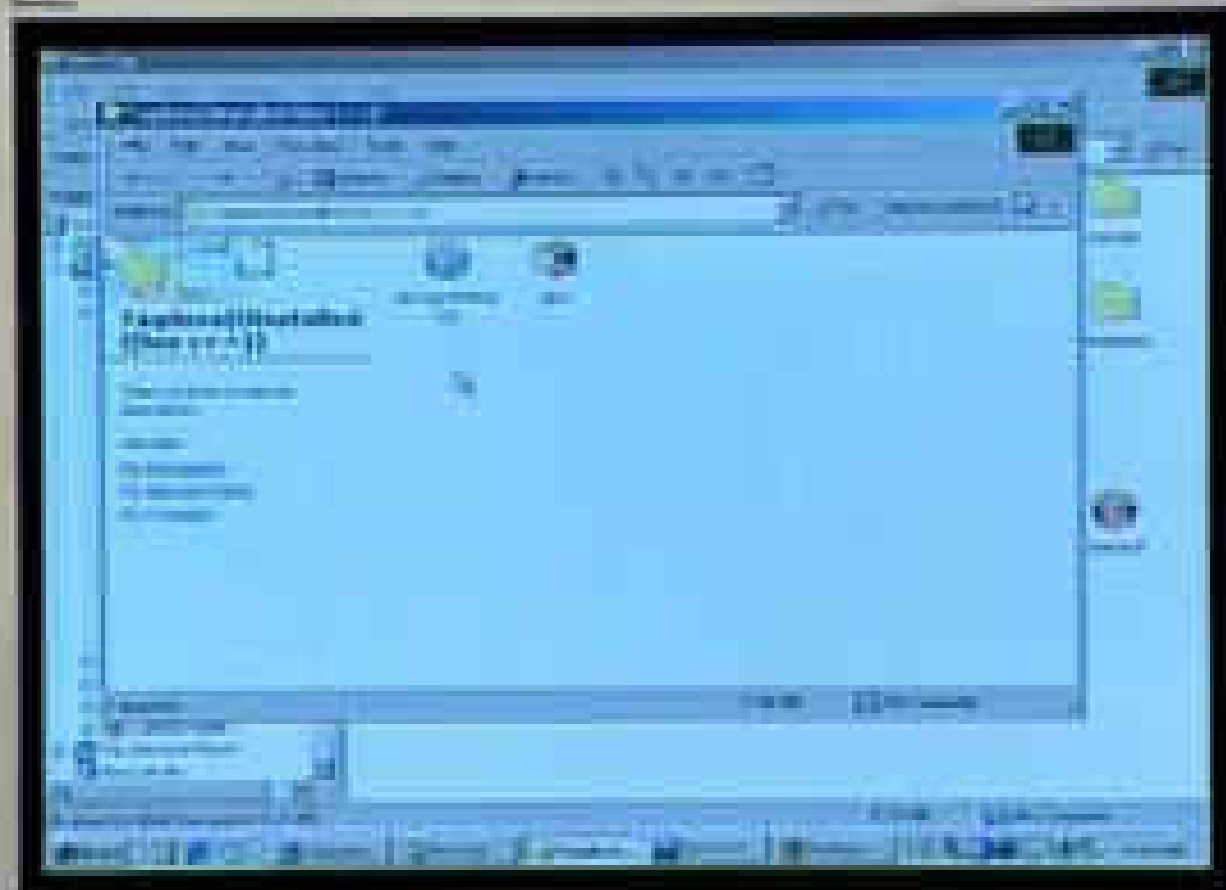












The translation can now be expressed using homogeneous coordinates as:

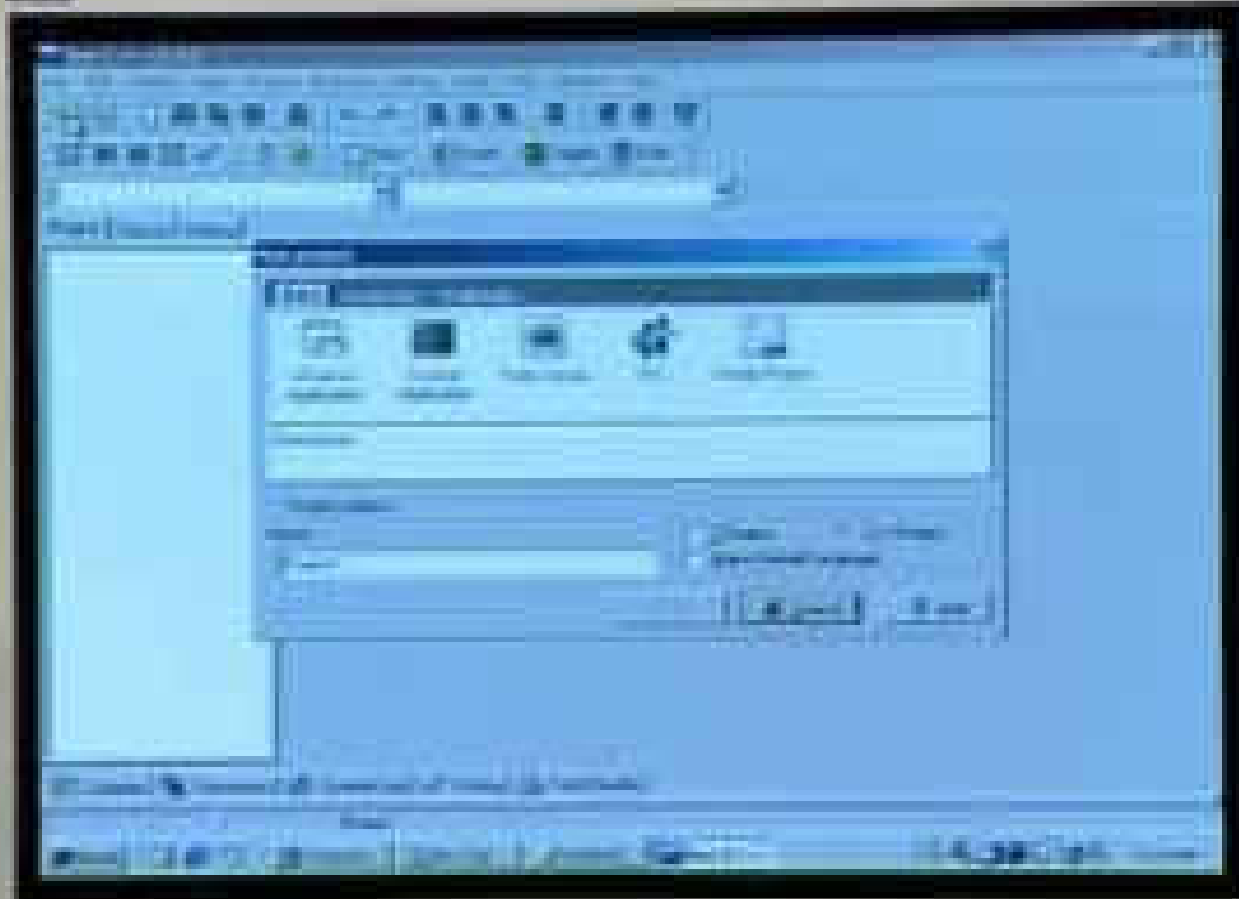
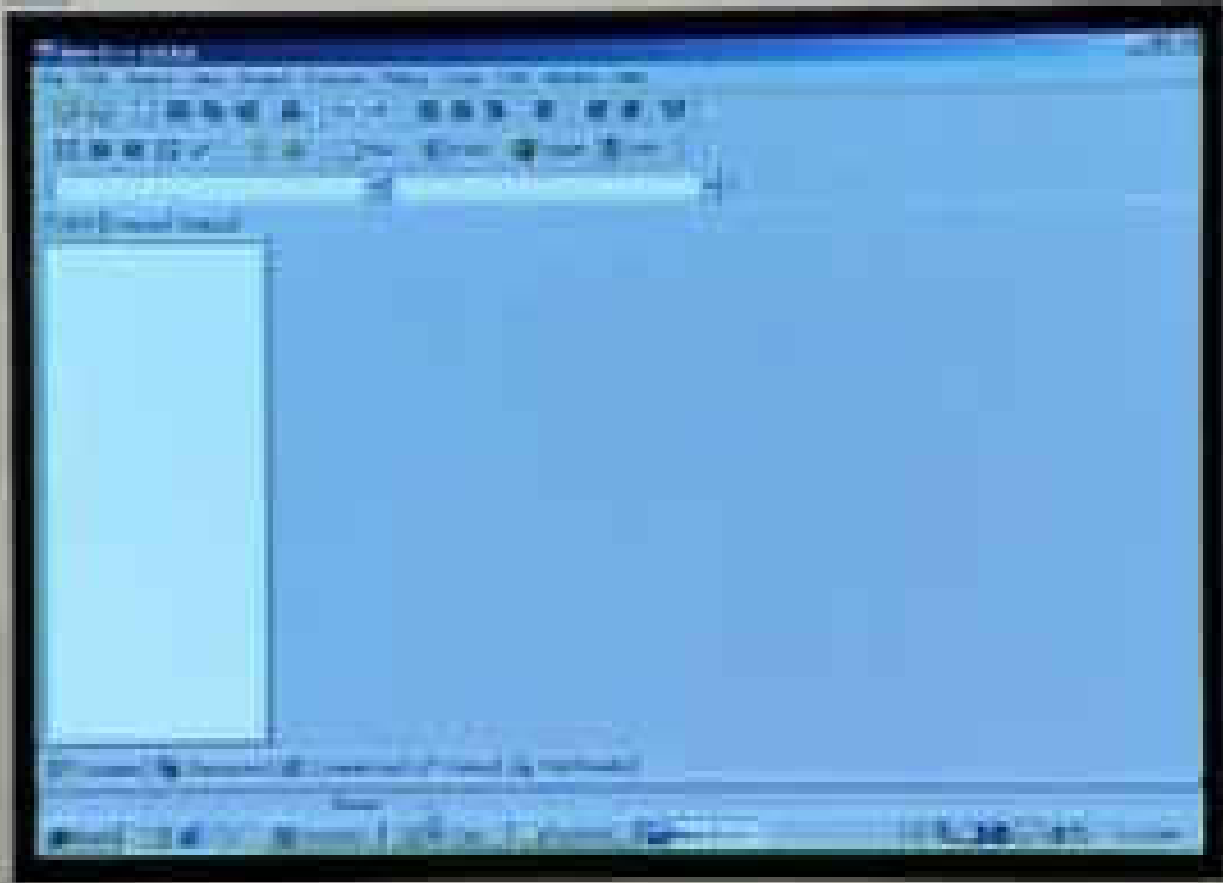
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Abbreviated as:

$$P' = T(t_x, t_y) \cdot P$$

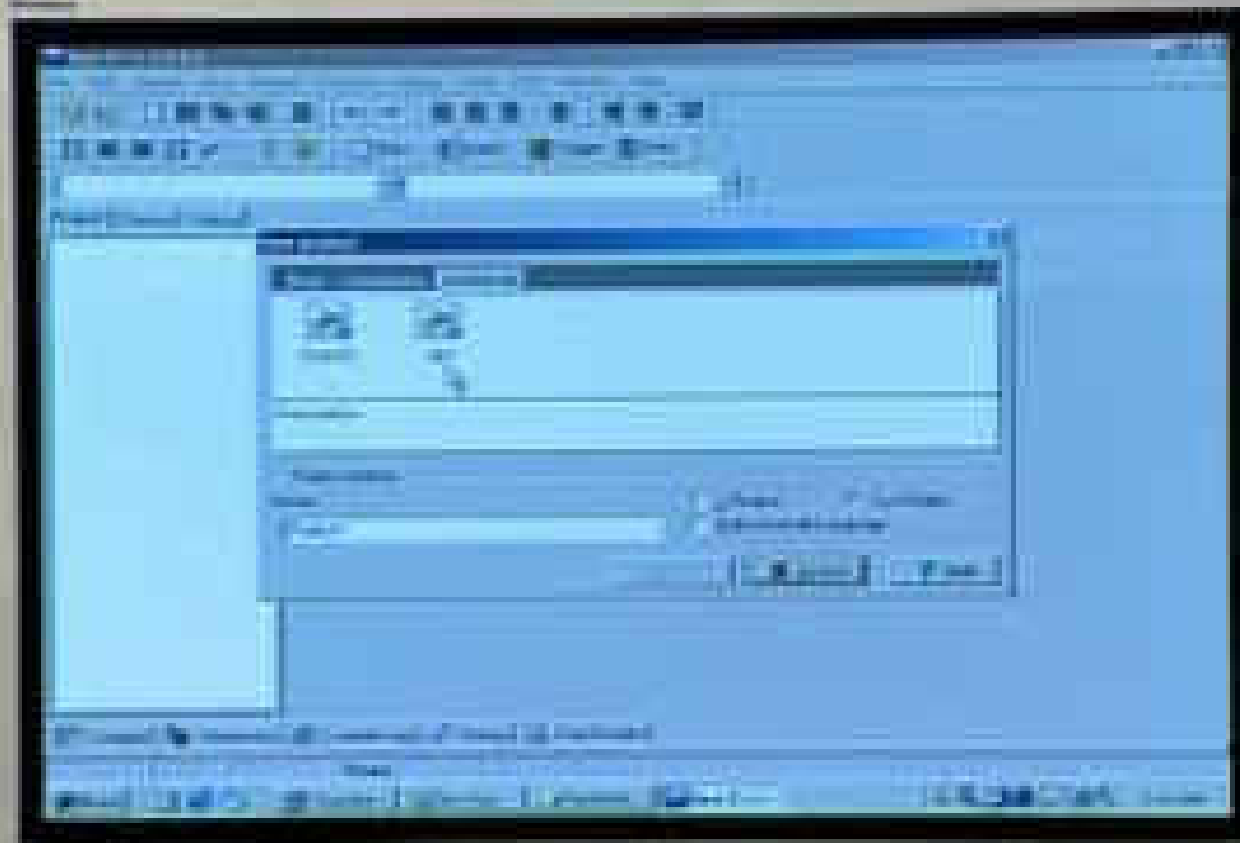
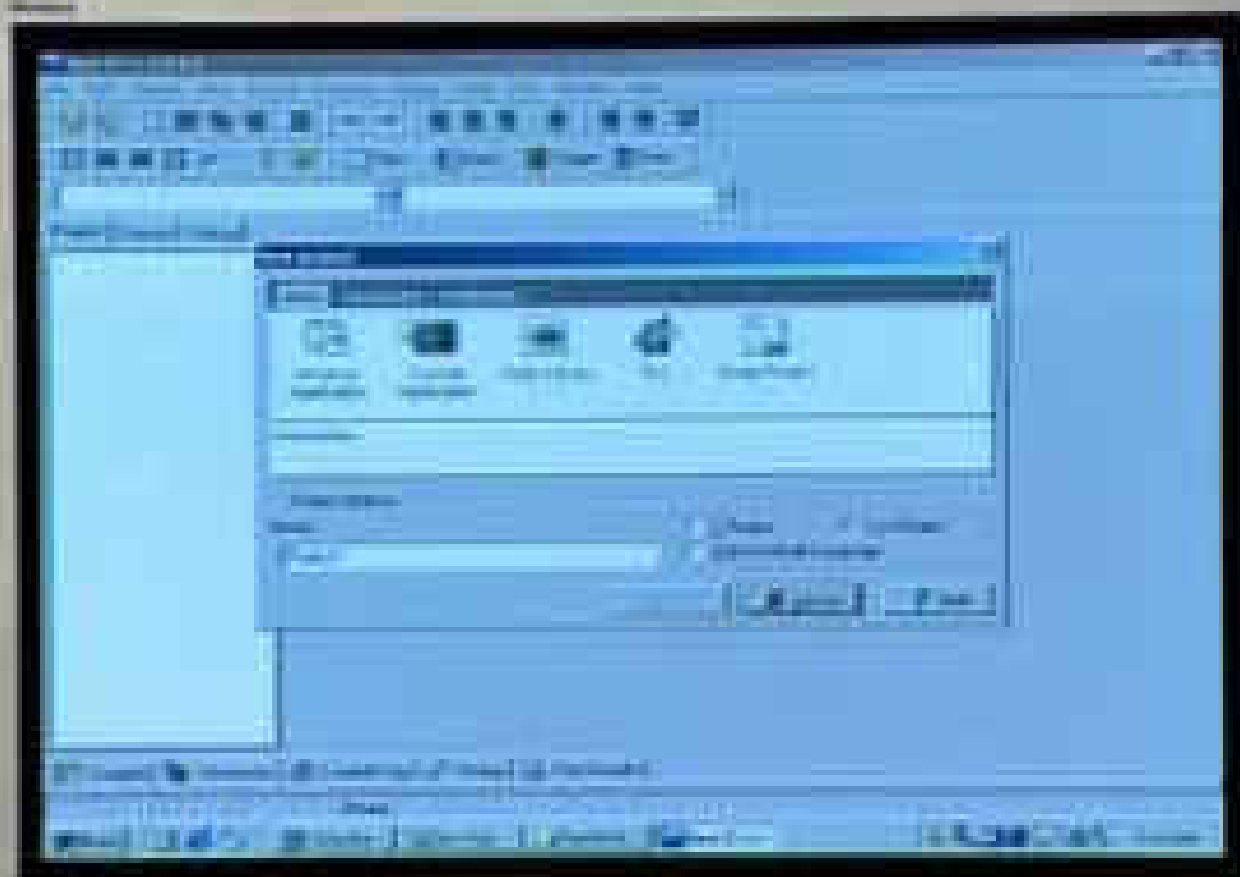


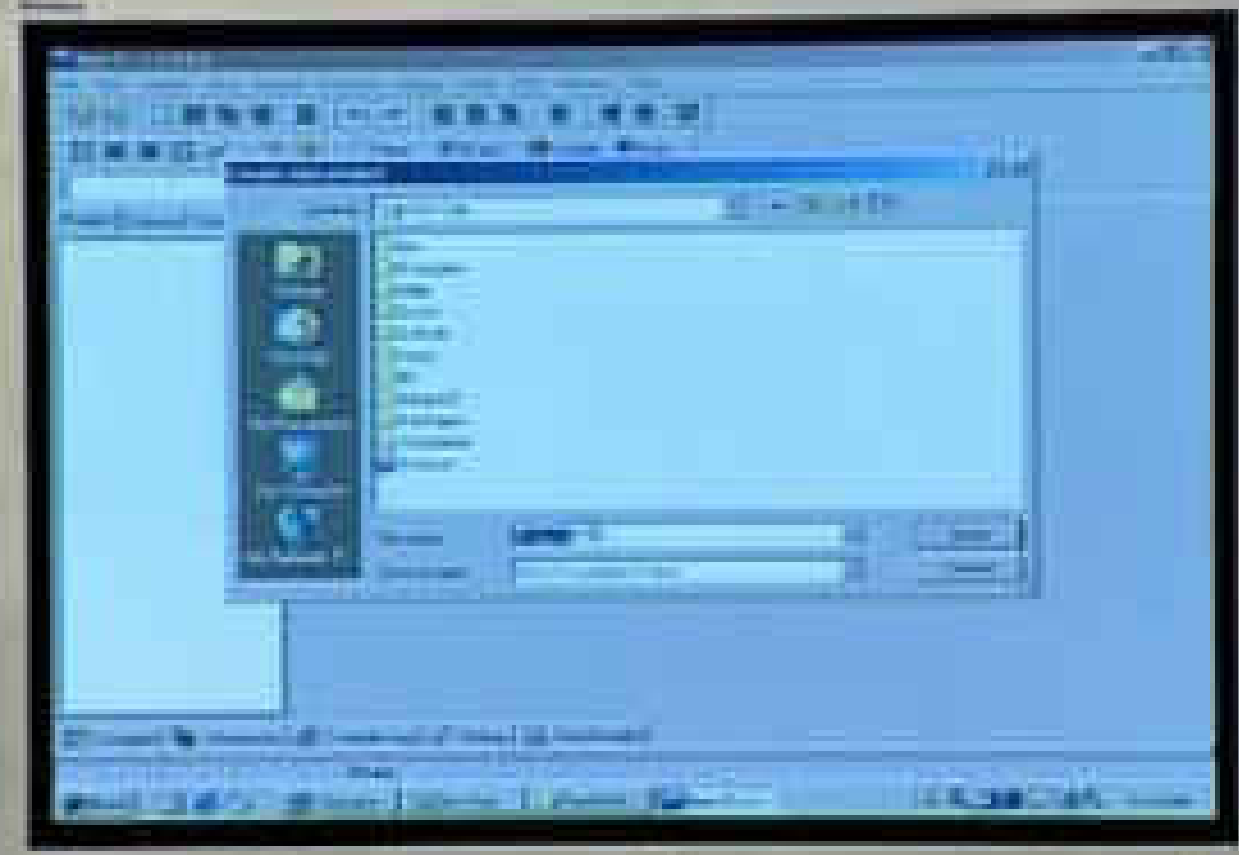
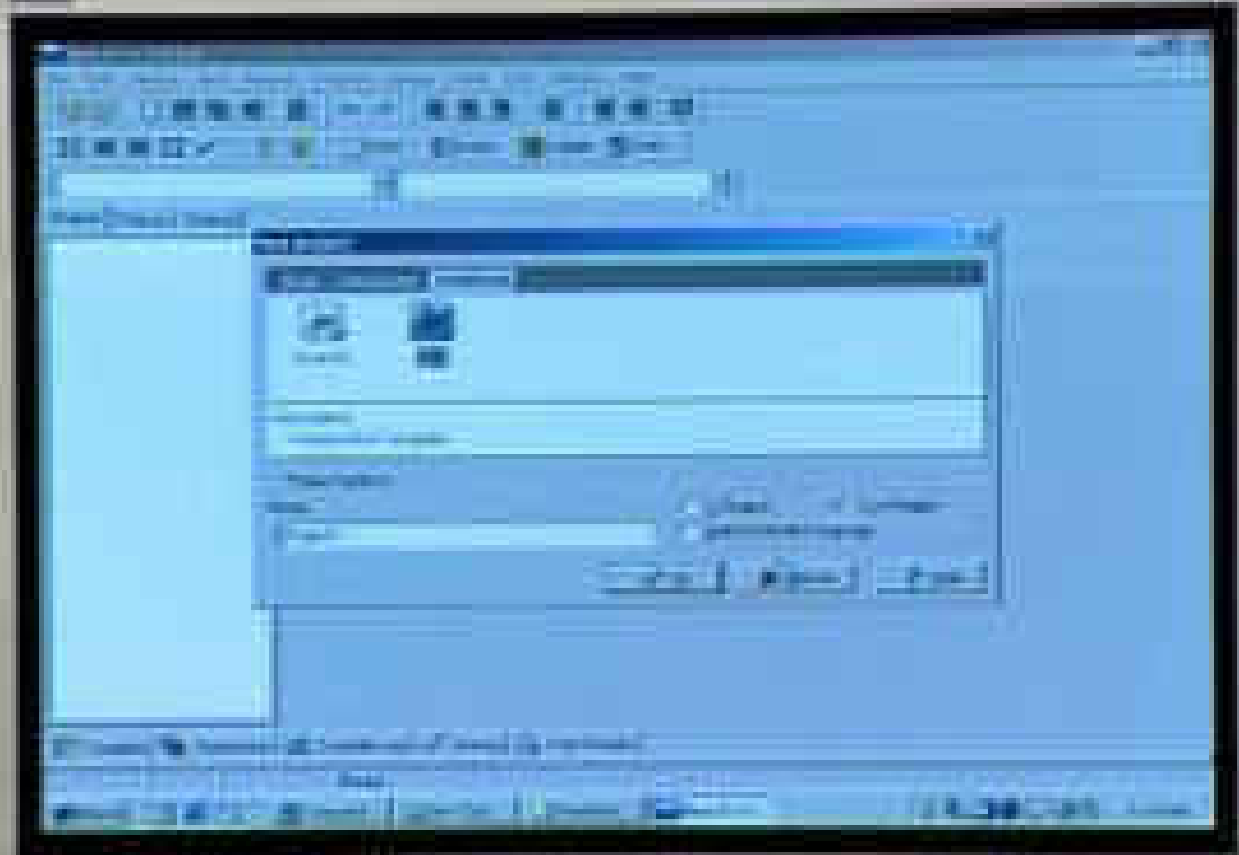




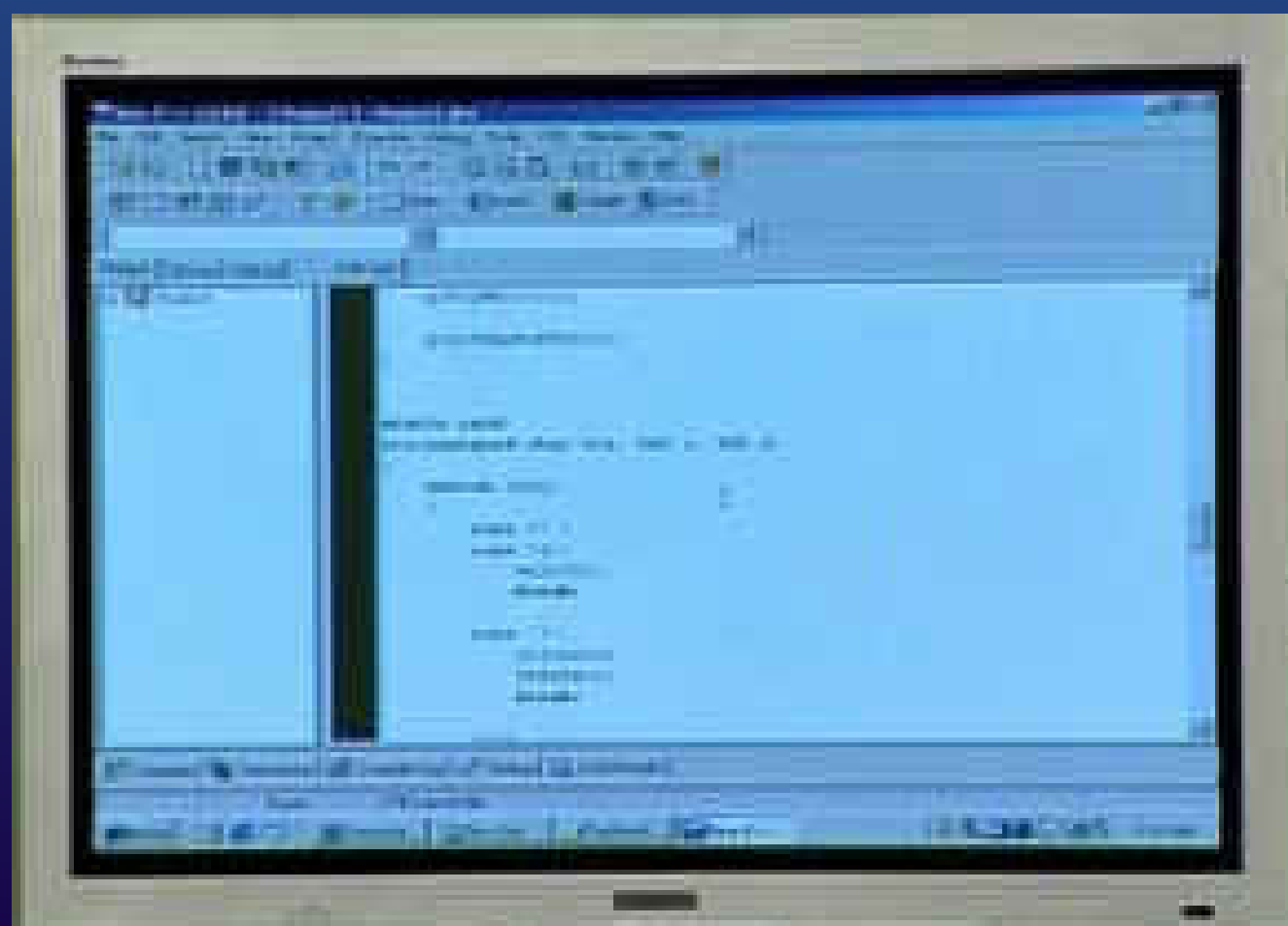
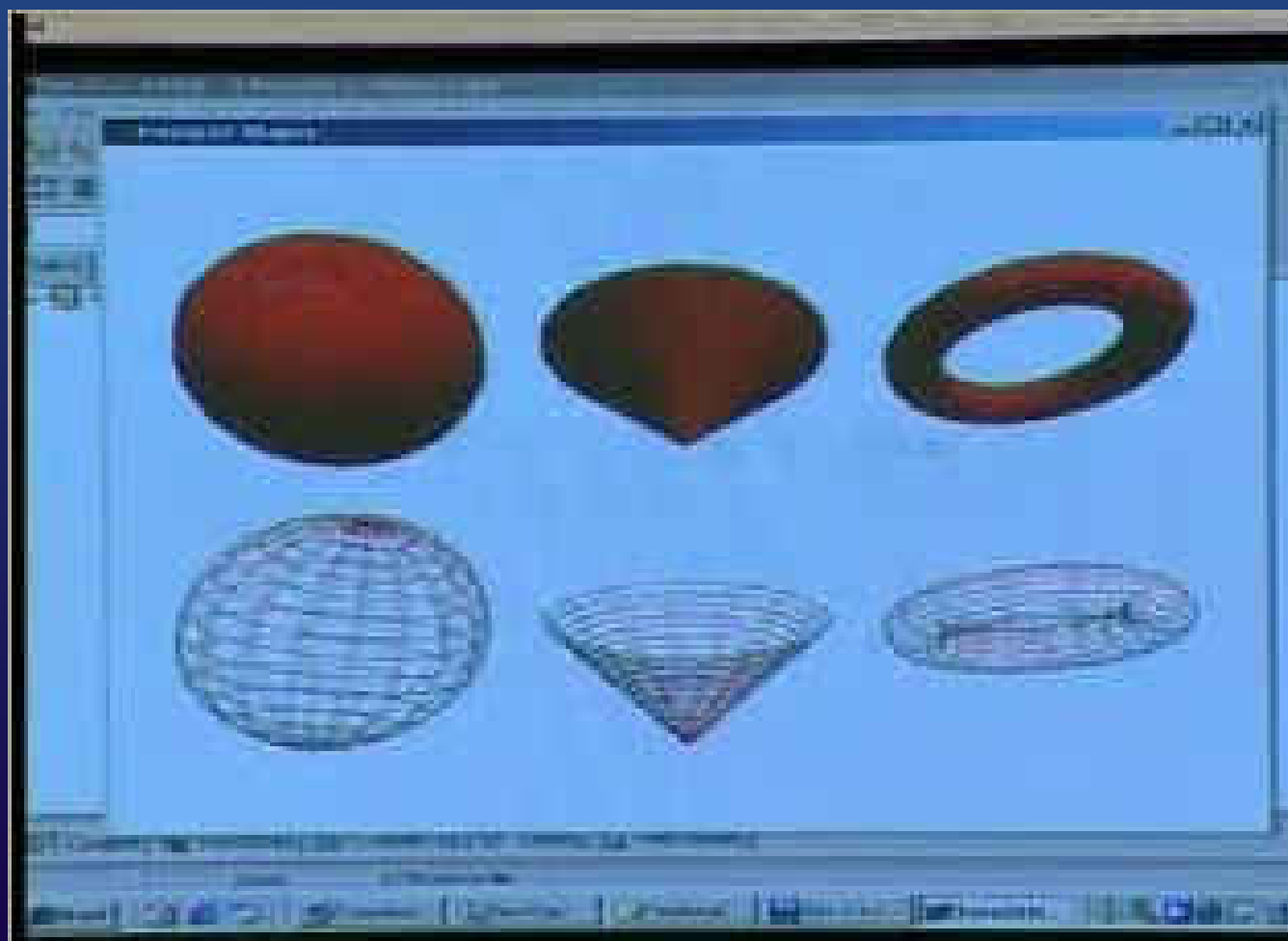


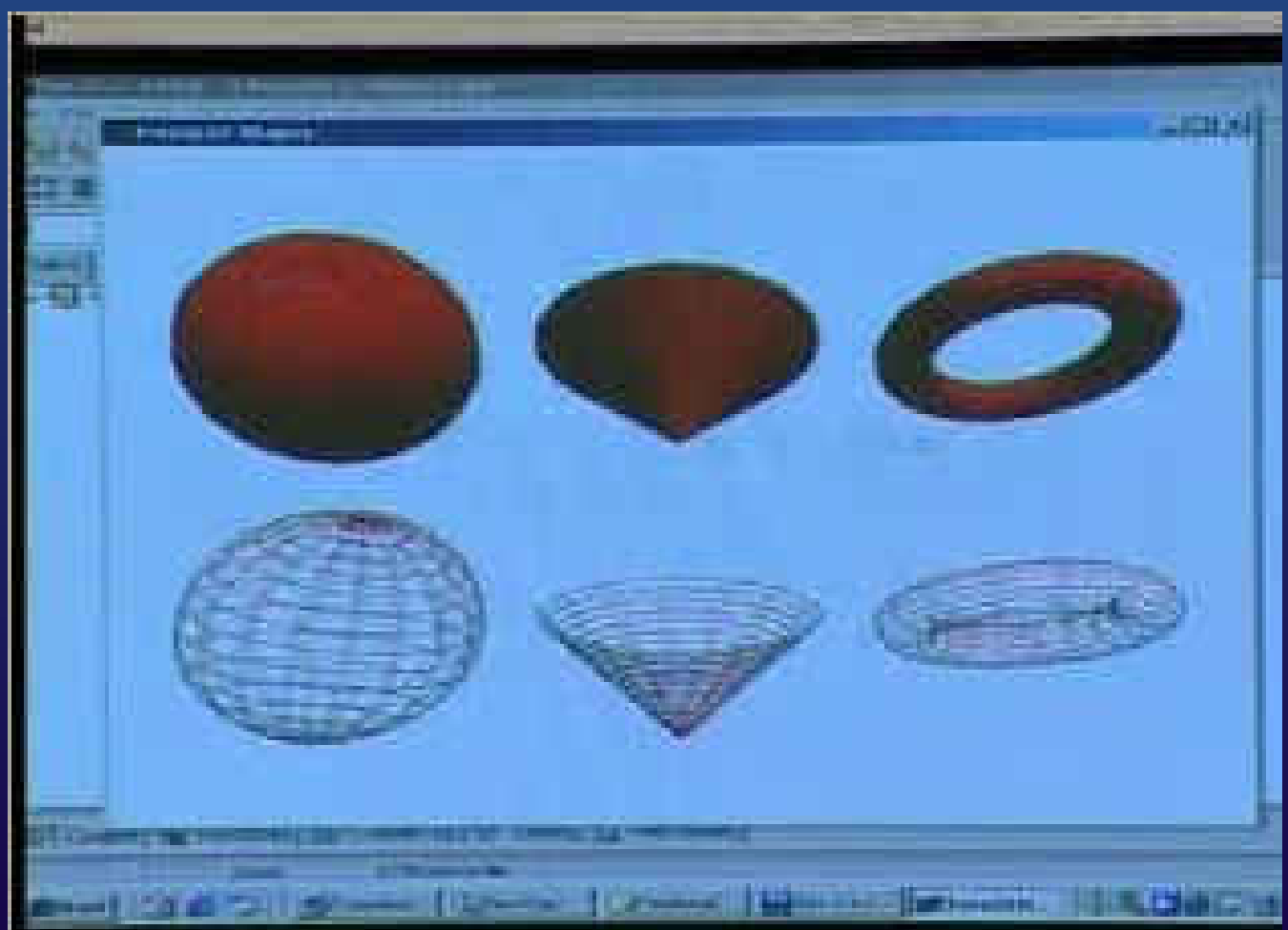
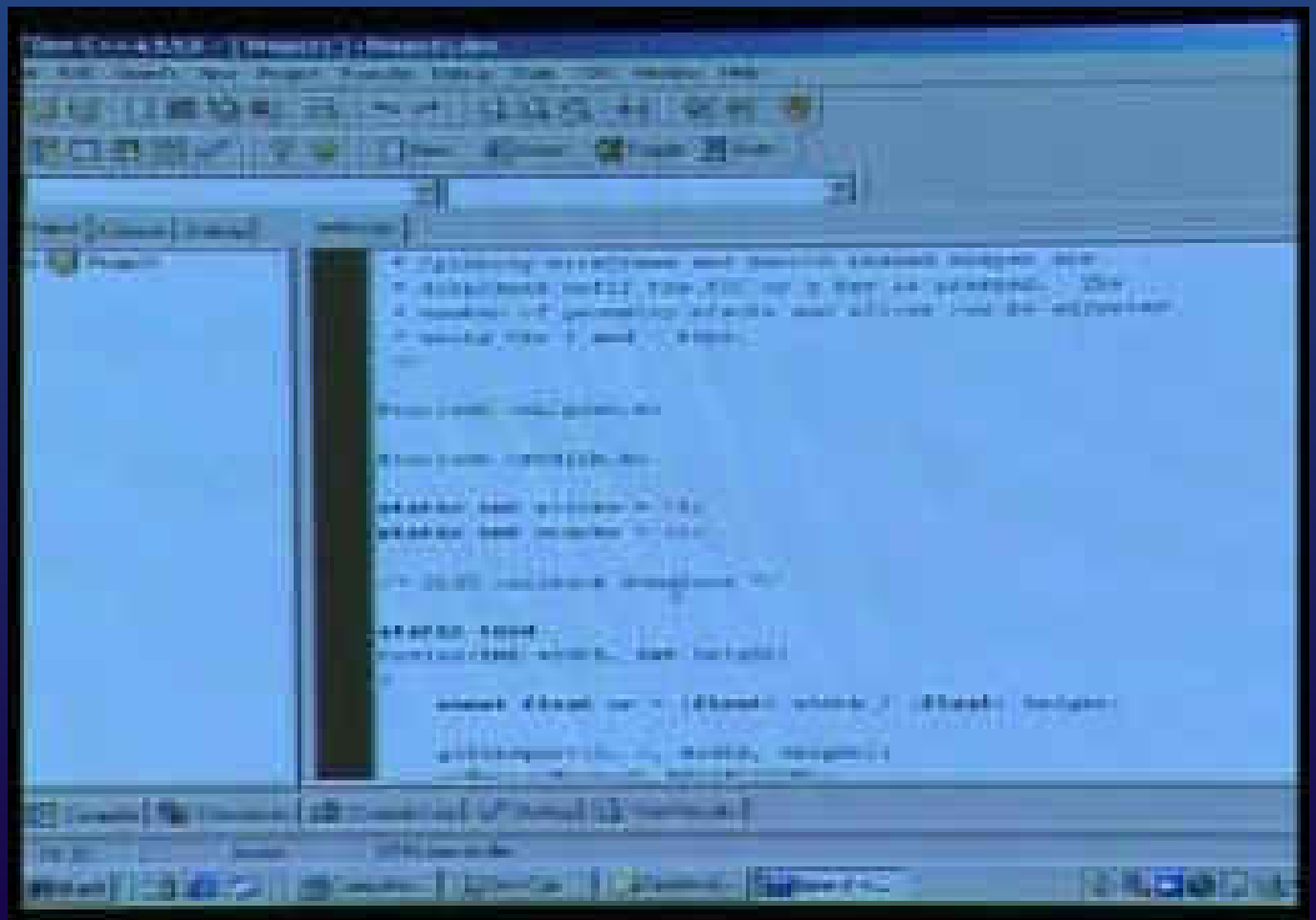




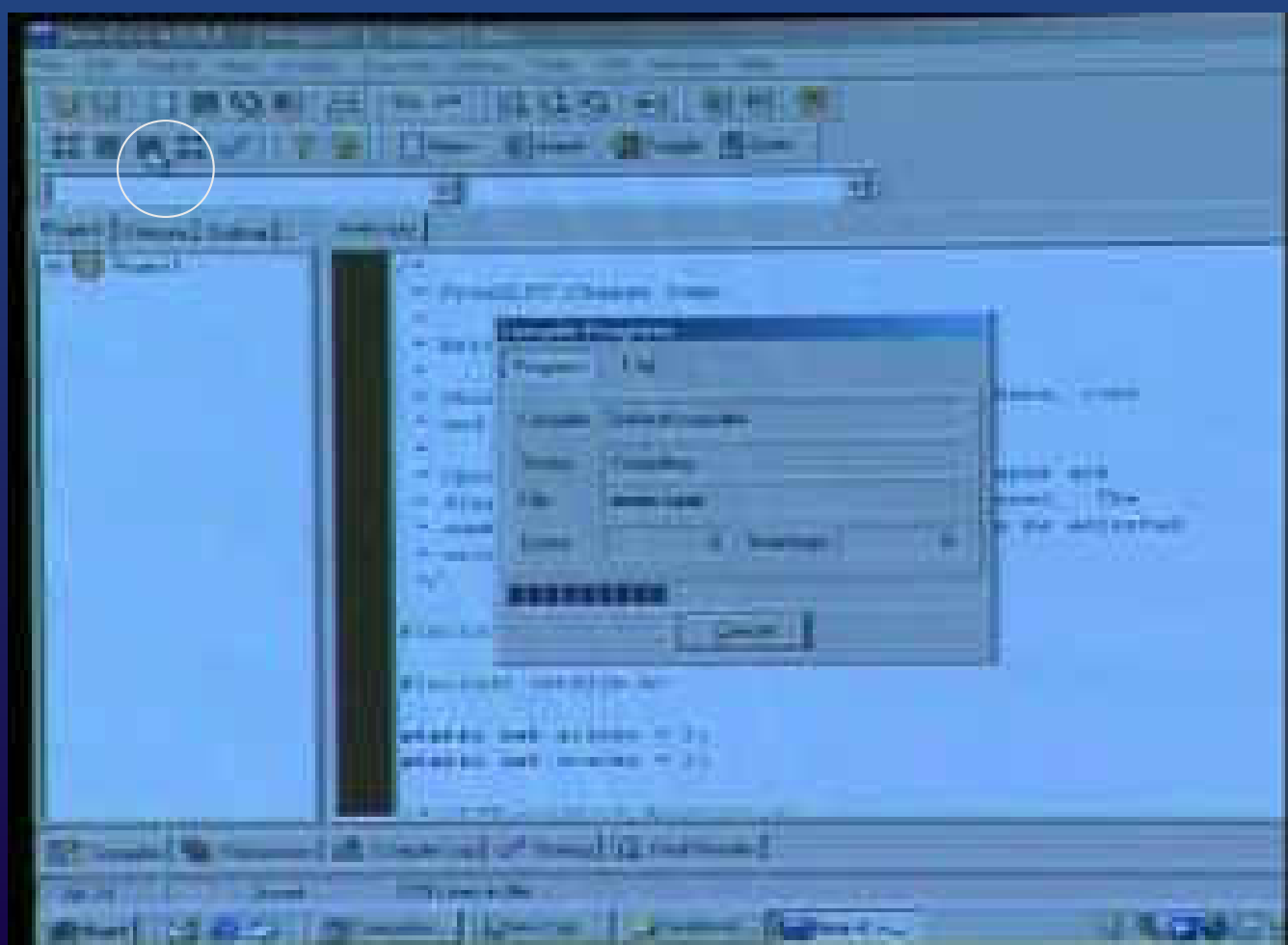
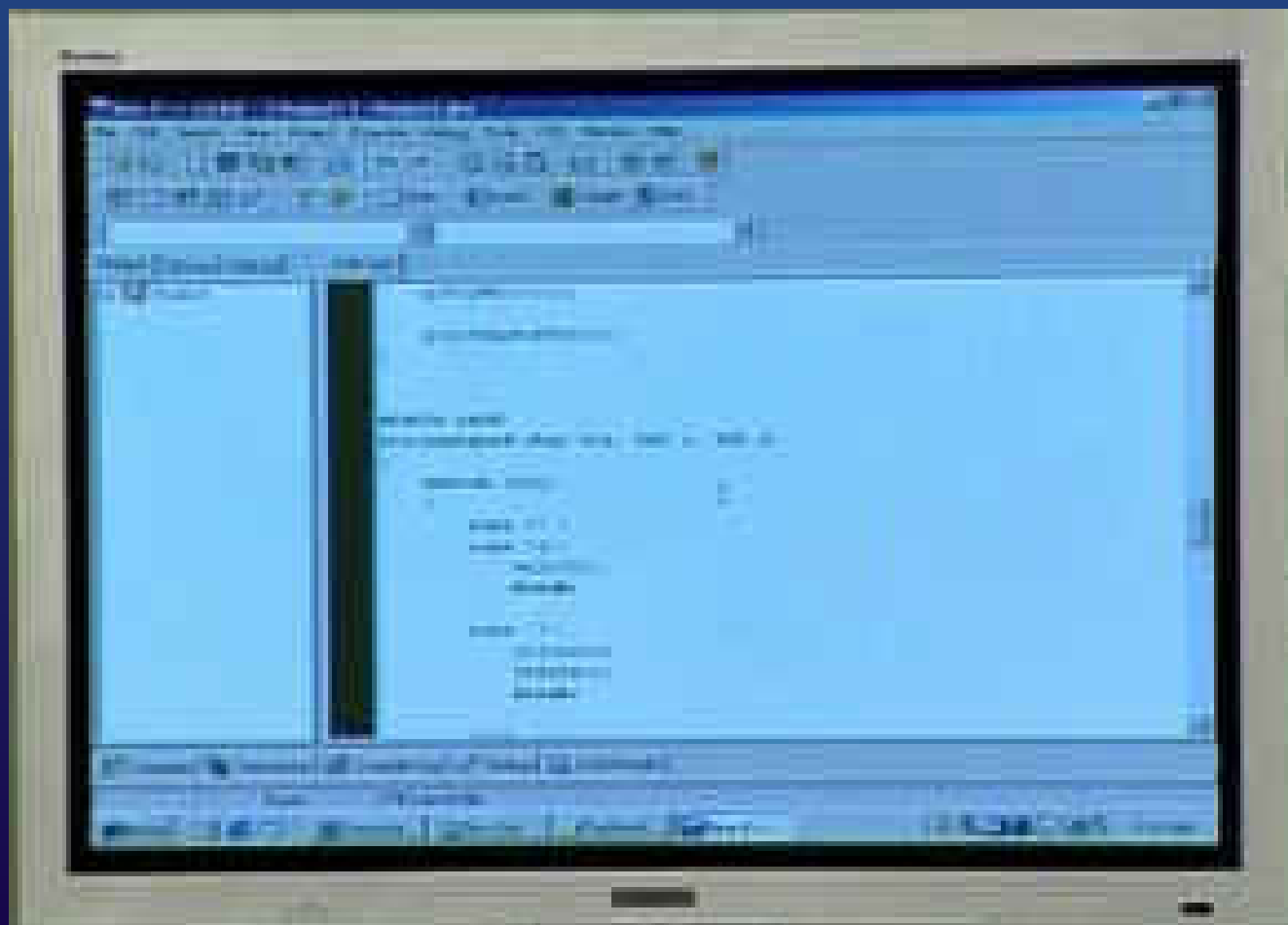


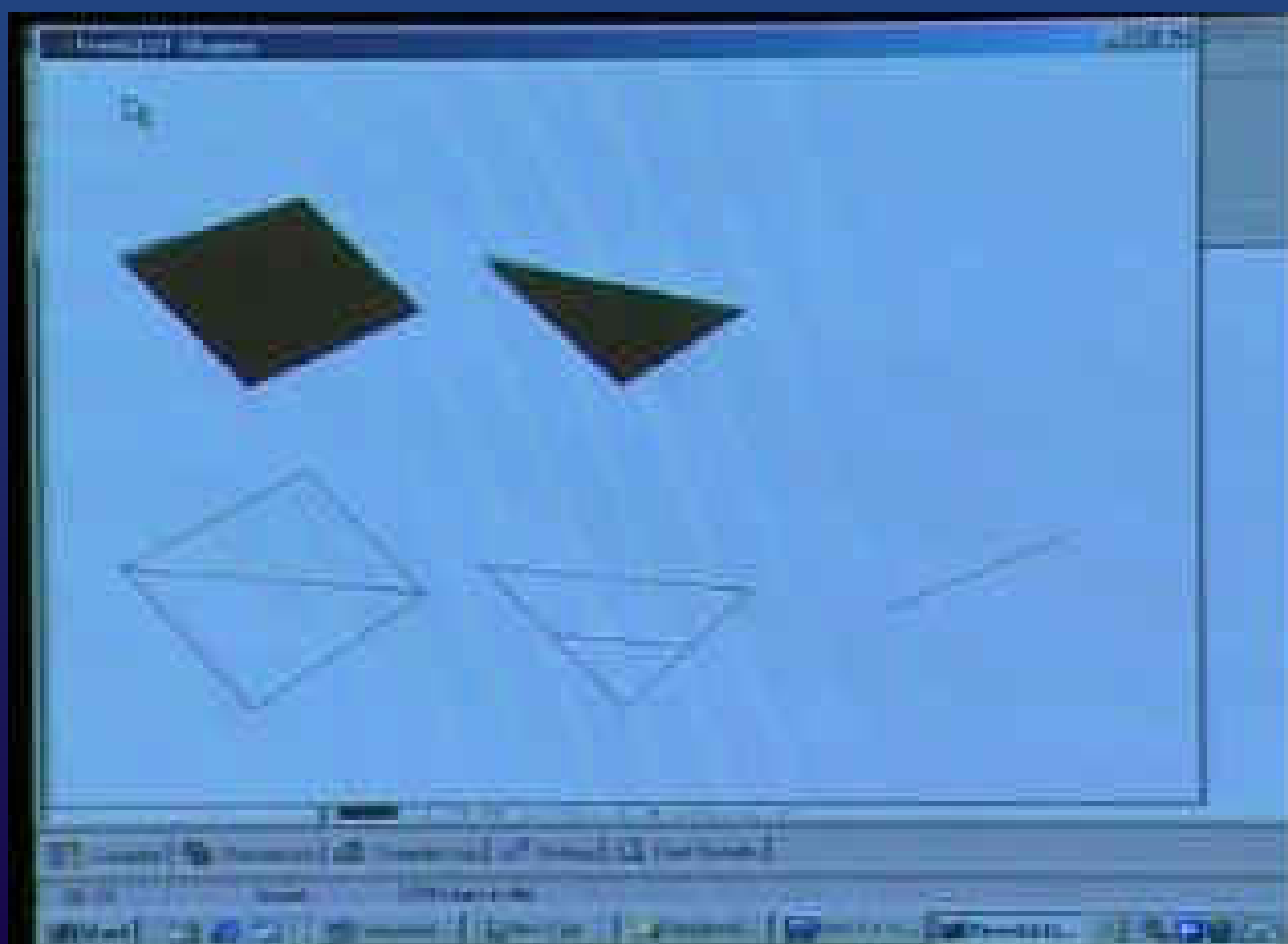
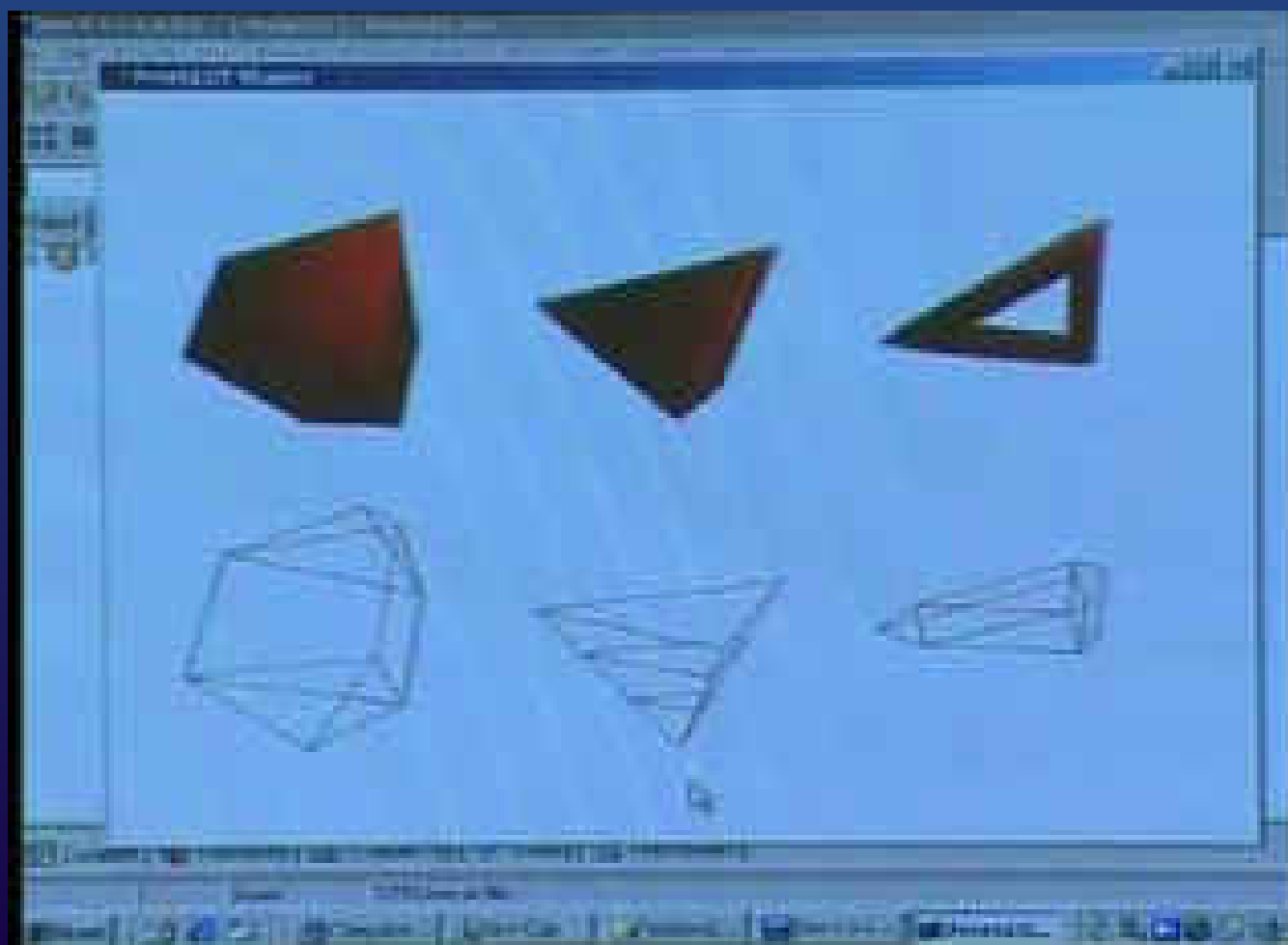


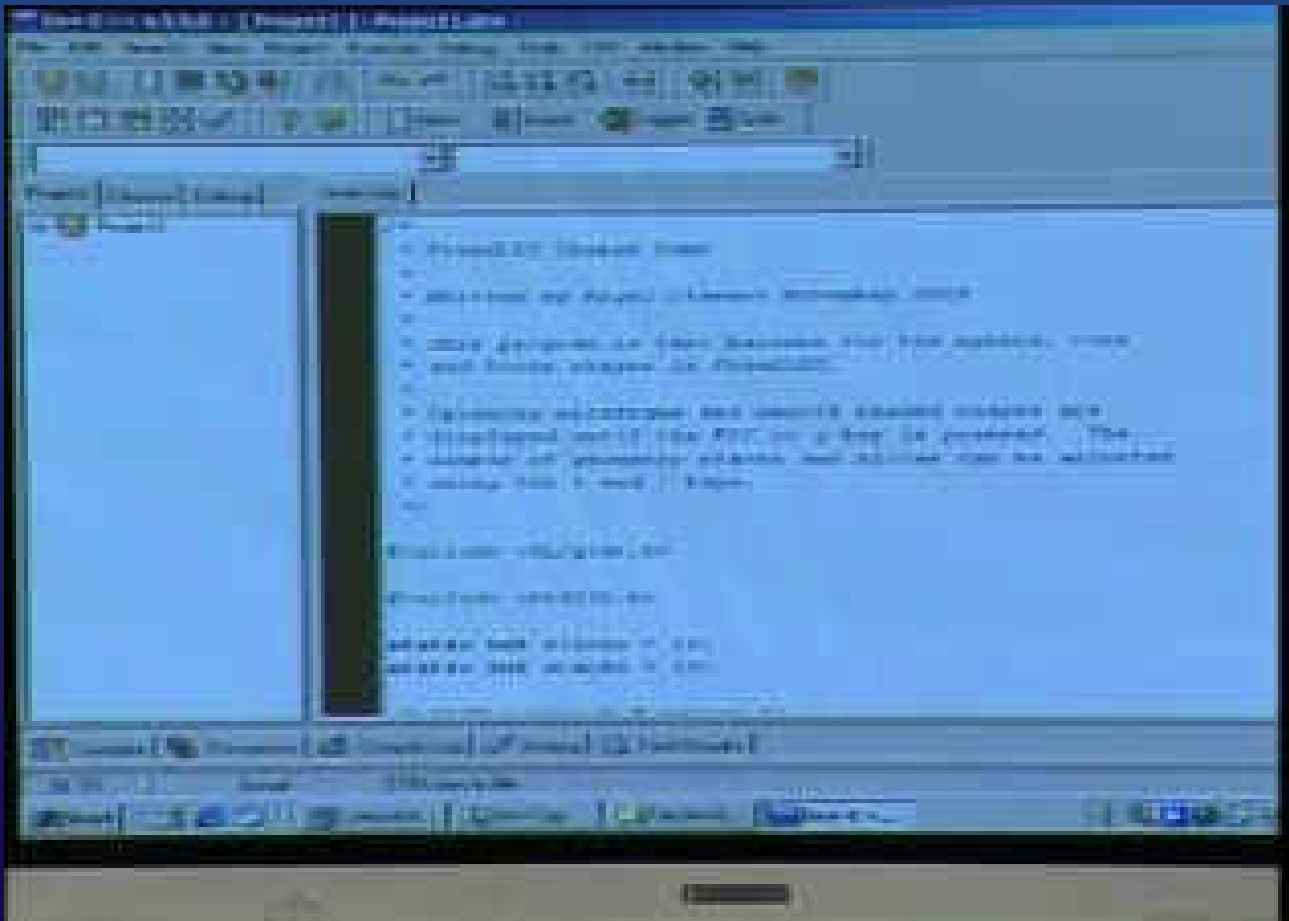












# Computer Graphics

## Lecture 14

### CLIPPING I

Taqdees A. Siddiqi  
cs602@vu.edu.pk

# CLIPPING I

Taqdees A. Siddiqi  
cs602@vu.edu.pk

## Concept

Restricting the effect of graphics primitives to a sub-region of the canvas

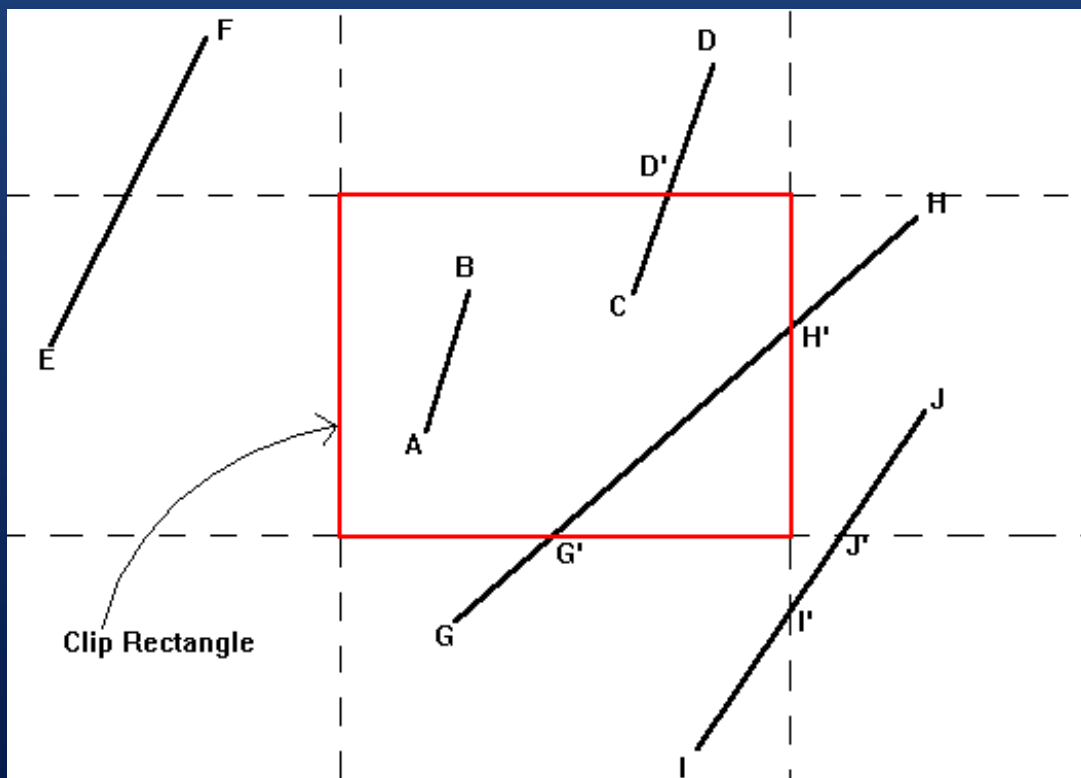
Protecting other portions of the canvas

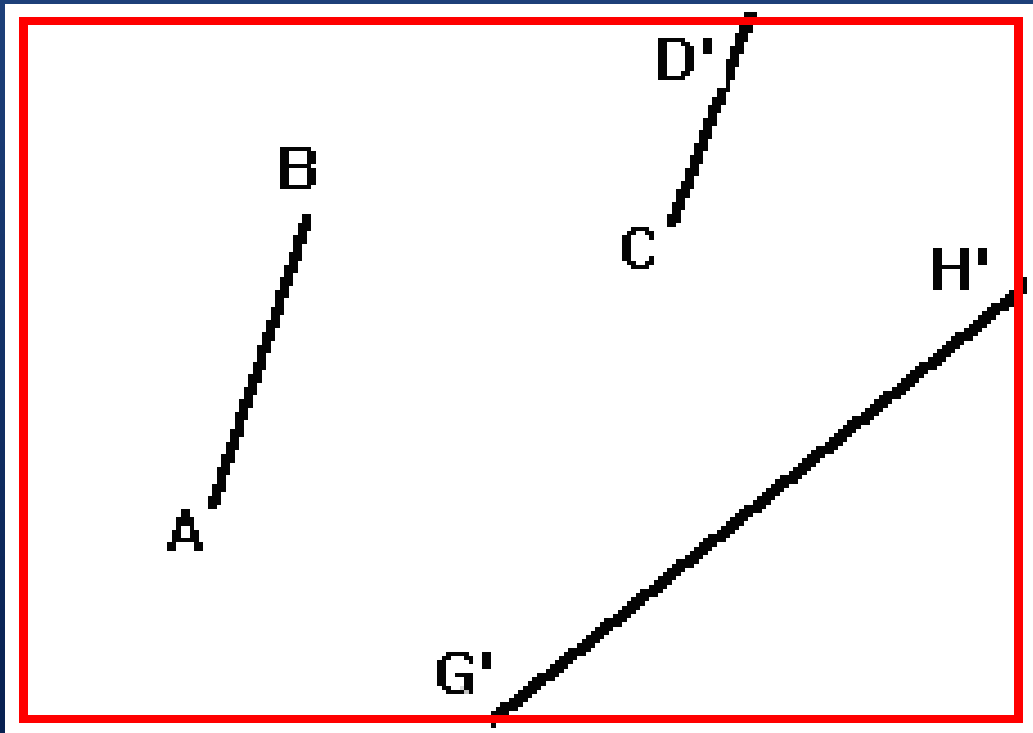
All primitives are clipped to the boundaries of this clipping rectangle;  
Primitives lying outside the clip rectangle are not drawn.

The default clipping rectangle is the full canvas (the screen); We cannot see any graphics primitives outside the screen

# Example

This is a simple example of line clipping:  
the display window is the canvas and also the default  
clipping rectangle, thus all line segments inside the  
canvas are drawn.





## Point Clipping

$$x_{\min} \leq x \leq x_{\max}$$

$$y_{\min} \leq y \leq y_{\max}$$

# Line Clipping

Clipping of lines against rectangles

## Clipping Individual Points

Simpler problem

$X_{min} < X < X_{max}$

And

$Y_{min} < Y < Y_{max}$



Trivial Accept –

save a line with both endpoints inside all clipping boundaries

Trivial Reject –

discard a line with both endpoints outside the clipping boundaries

For all other lines –

compute intersections of line with clipping boundaries

## Parametric Representation Line

$$x = x_1 + u (x_2 - x_1)$$

$$y = y_1 + u (y_2 - y_1)$$

$$\text{and } 0 \leq u \leq 1$$

If  $0 \leq u \leq 1$  then the line does enter the interior of the window at that boundary  
otherwise, the line does not enter the interior of the window at that boundary.

## Solve Simultaneous Equations

Need to consider only endpoints of a line:

If both endpoints lie inside the clip rectangle; it can be trivially accepted.

If one endpoint lies inside and one outside, the line intersects the clip rectangle and we must compute the intersection point.

If both endpoints are outside the clip rectangle, we need to perform further calculations to determine whether there are any intersections.

# The Brute Force Approach

Intersect that line with each of the four clip rectangle edges. if so, the line cuts the clip rectangle and is partially inside.

For each line and clip-rectangle edge, we therefore take the two mathematically infinite lines that contain them and intersect them.

Test whether this intersection point is "interior" -- that is, whether it lies within both the clip rectangle edge and the line.

If so, there is an intersection with the clip rectangle.

# Cohen Sutherland Line Clipping Algorithm

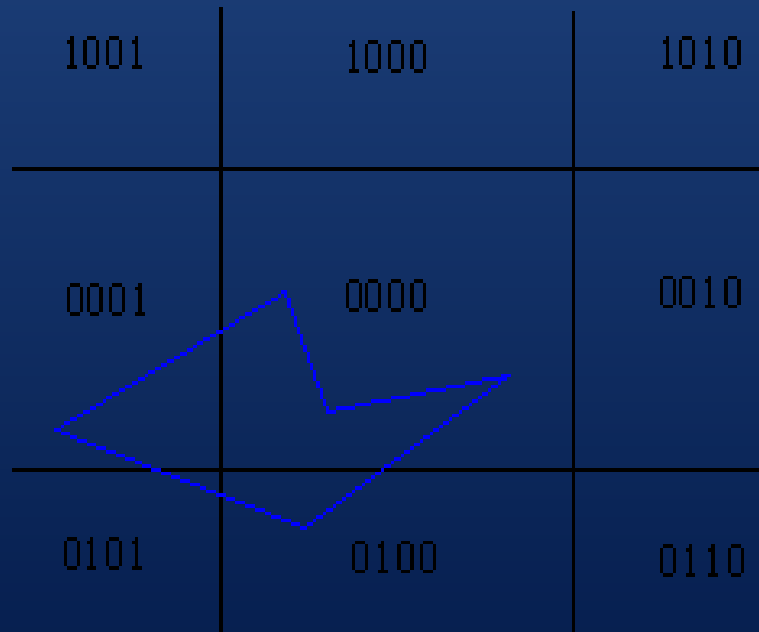
More efficient;

Performs initial tests on a line to determine whether intersection calculations can be avoided.

End-point pairs are checked using the out-code.

If not divided into two segments at a clip edge, iteratively clipped by testing trivial-acceptance or trivial-rejection.

# Trivial Accept / Reject Test



Each bit in the out-code is set to either  
1 (true) or 0 (false);

The 4 bits in the code correspond to the following  
conditions:



Bit 1: outside half-plane of top edge, above top edge  
 $Y > Y_{\max}$

Bit 2: outside half-plane of bottom edge, below  
bottom edge  $Y < Y_{\min}$

Bit 3: outside half-plane of right edge, to the right of  
right edge  $X > X_{\max}$

Bit 4: outside half-plane of left edge, to the left of left  
edge  $X < X_{\min}$

# Conclusion

Algorithm is efficient when out-code testing can be done cheaply

## Liang-Barsky Algorithm

Faster line clippers based on analysis of the parametric equation of a line segment:

$$x = x_1 + u \Delta x$$

$$y = y_1 + u \Delta y,$$

where  $0 \leq u \leq 1$ ,

$$\Delta x = x_2 - x_1 \text{ and}$$

$$\Delta y = y_2 - y_1$$

Following the Liang-Barsky approach, we first write the point-clipping in a parameteric way:

$$x_{\min} \leq x_1 + u \Delta x \leq x_{\max}$$

$$y_{\min} \leq y_1 + u \Delta y \leq y_{\max}$$

These four inequalities can be expressed as

$$u \cdot p_k \leq q_k, \text{ for } k = 1, 2, 3, 4$$

Where parameters  $p$  and  $q$  are defined as:

$$p_1 = -\Delta x, \quad q_1 = x_1 - x_{\min}$$

$$p_2 = -\Delta x, \quad q_2 = x_{\max} - x_1$$

$$p_3 = -\Delta y, \quad q_3 = y_1 - y_{\min}$$

$$p_4 = -\Delta y, \quad q_4 = y_{\max} - y_1$$

Any line that is parallel to one of the clipping boundaries having  $p_k = 0$ . If, for that value of  $k$ , we also find  $q_k \geq 0$ , the line is inside the parallel clipping boundary.

When  $p_k < 0$ , the infinite extension of the line proceeds from outside to the inside of the infinite extension of the particular clipping boundary.

If  $p_k > 0$ , the line proceeds from the inside to the outside.

For a nonzero value of  $p_k$ , we can calculate the value of  $u$  that corresponds to the point where the infinitely extended line intersects the extension of boundary  $k$  as:

$$u = q_k / p_k$$

For each line, we can calculate values for parameters  $u_1$  and  $u_2$  which define that part of the line which lies within the clip rectangle.

The value of  $u_1$  is determined by looking at the rectangle edges for which the line proceeds from the outer-side to the inner-side ( $p < 0$ ). For these edges we calculate  $r_k = q_k / p_k$ .

The value of  $u_1$  is taken as the largest of the set consisting of 0 and the various values of  $r$ .

The value of  $u_2$  is determined by examining the boundaries for which the line proceeds from inside to outside  
( $p > 0$ ).



# Algorithm

Line intersection parameters are initialized to the values:

$u_1 = 0$  and  $u_2 = 1$ .

For each clipping boundary, the appropriate values for  $p$  and  $q$  are calculated and used to determine whether the line can be rejected or the intersection parameters are to be adjusted.

A value of  $r_k$  is calculated for each of these boundaries and the value of  $u_2$  is the minimum of the set consisting of 1 and the calculated  $r$  values.

If  $u_1 > u_2$ , the line is completely outside the clip window and it can be rejected.

Otherwise, the end points of the clipped line are calculated from the two values of parameter  $u$ .

When  $p < 0$ , the parameter  $r$  is used to update  $u_1$ ;

When  $p > 0$ , the parameter  $r$  is used to update  $u_2$ .

If updating  $u_1$  or  $u_2$  results in  $u_1 > u_2$ , we reject the line.

Otherwise, we update the appropriate  $u$  parameter only if the new value results in a shortening of the line.

When  $p = 0$  and  $q < 0$ , we can discard the line since it is parallel to and outside of this boundary.

If the line has not been rejected after all four values of  $p$  and  $q$  have been tested, the endpoints of the clipped line are determined from values of  $u_1$  and  $u_2$ .

# Conclusion

More efficient; intersection calculations are reduced.

Each update of parameters  $u_1$  and  $u_2$  requires only one division; and window intersections of the line are computed only once, when the final values of  $u_1$  and  $u_2$  are computed.

Cohen-Sutherland algorithm can repeatedly calculate intersections along a line path, even though the line may be completely outside the clip window, and, each intersection calculation requires both a division and a multiplication.

# Computer Graphics

## Lecture 14

# Computer Graphics

## Lecture 15

### CLIPPING II

Taqdees A. Siddiqi  
cs602@vu.edu.pk

### Polygon Clipping

A polygon is usually defined by a sequence of vertices and edges. If the polygons are un-filled, line clipping techniques are sufficient however, if the polygons are filled, the process is more complicated.



# **Sutherland-Hodgeman clipping algorithm**

Clips any polygon against a convex clip polygon.

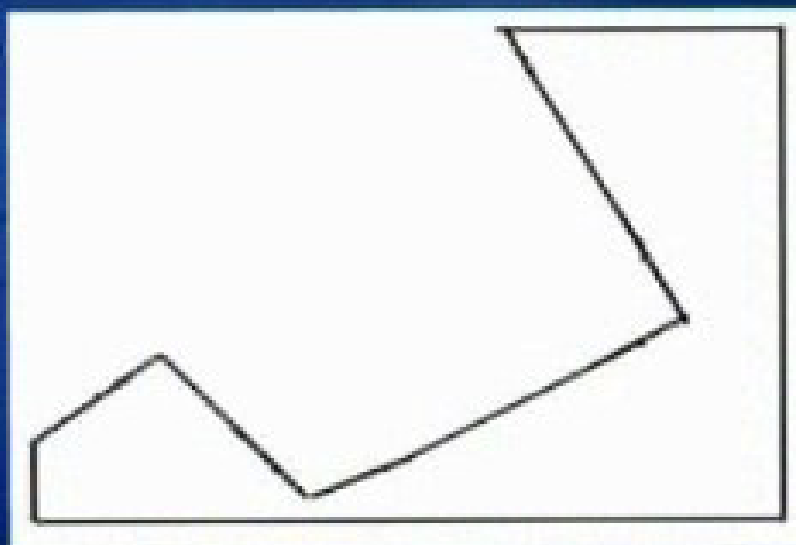
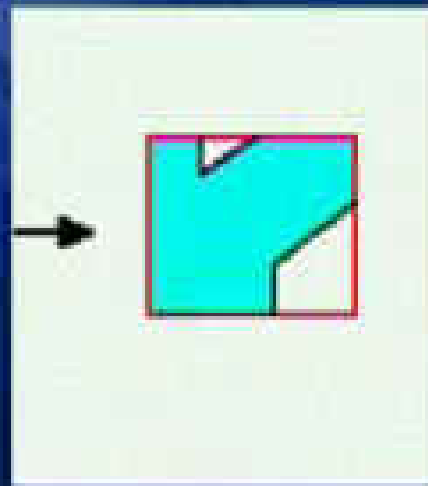
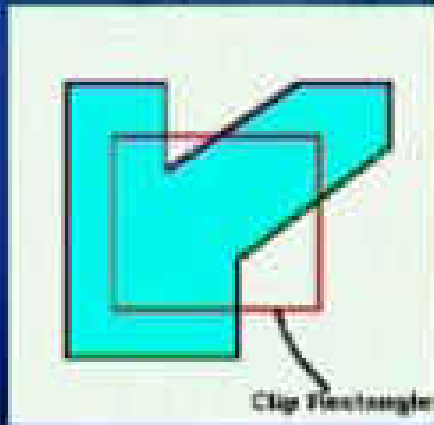
# **Weiler-Atherton clipping algorithm**

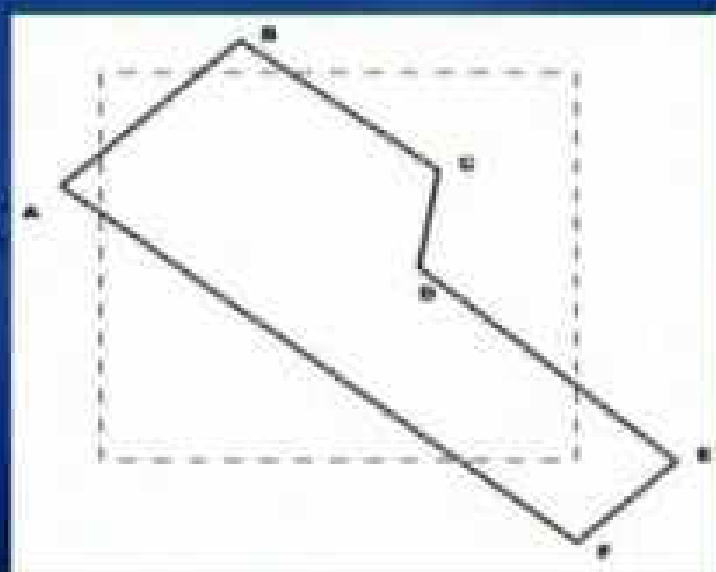
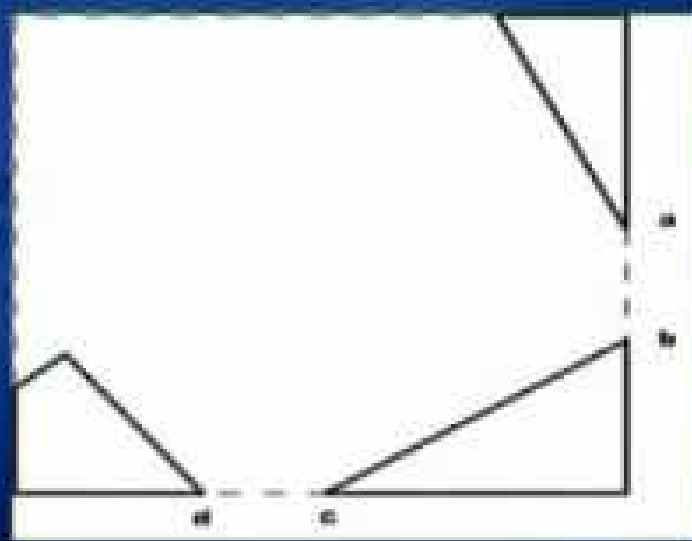
Weiler-Atherton clipping algorithm clips any polygon against any clip polygon.

The polygons may even have holes.

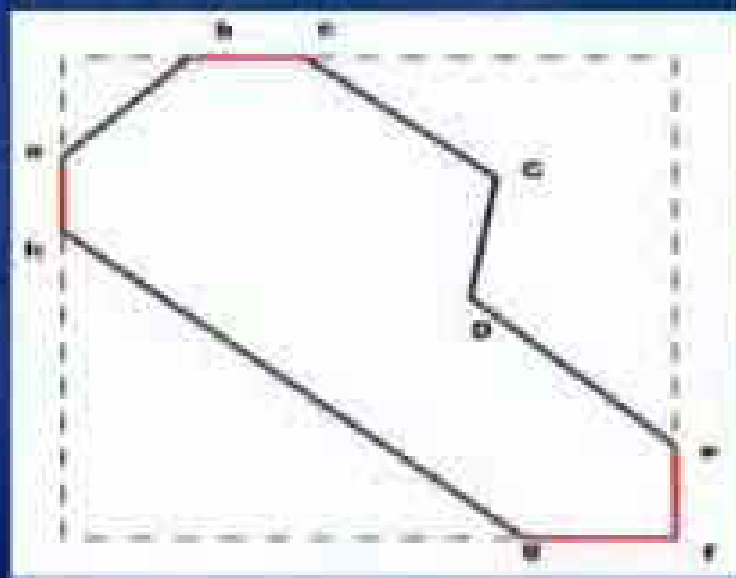
# Example

Example





**Polygon clipping- open polygons**



Polygon clipping - open polygons.

## Sutherland-Hodgman's Polygon Clipping Algorithm

Uses a divide-and-conquer strategy.

Clip a polygon against a single infinite clip edge.

Four clip edges, each defining one boundary of the clip rectangle, successively clip a polygon against a clip rectangle.

## Steps of Algorithm

Polygons can be clipped against each edge of the window one at a time.

Vertices which are kept after clipping against one window edge are saved for clipping against the remaining edges.

Note that the number of vertices usually changes and will often increase.

We are using the Divide and Conquer approach.

## **Four Cases of Polygon Clipping Against One Edge**

Case 1: Wholly inside visible region save endpoint

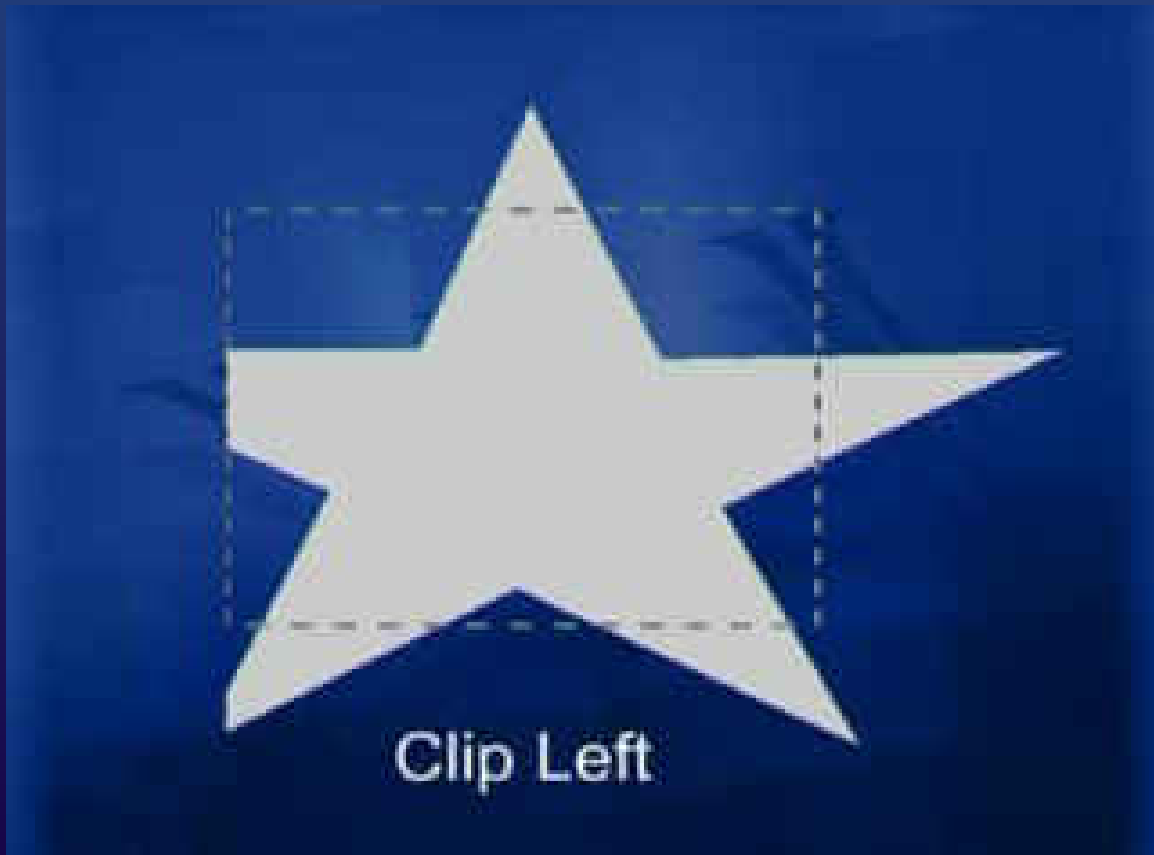
Case 2: Exit visible region save the intersection point

Case 3: Wholly outside visible region-save nothing

Case 4: Enter visible region-save intersection and endpoint



## Clip Left



## Clip Right



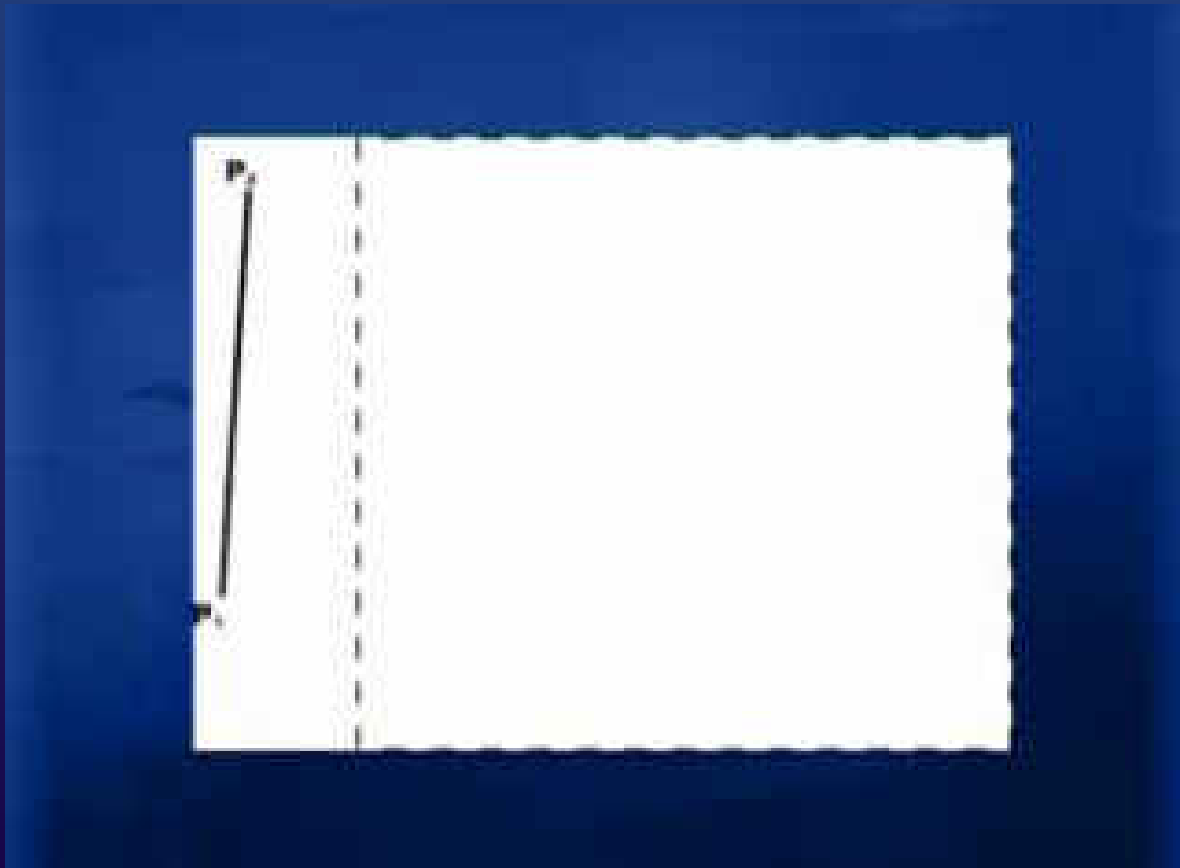


## Clip Bottom

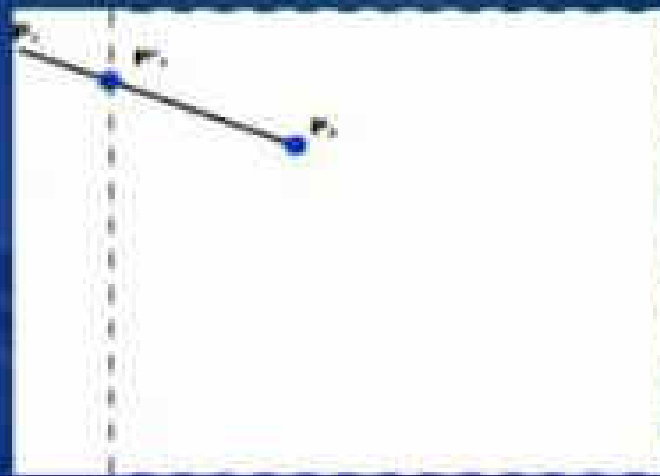


## Clip Top

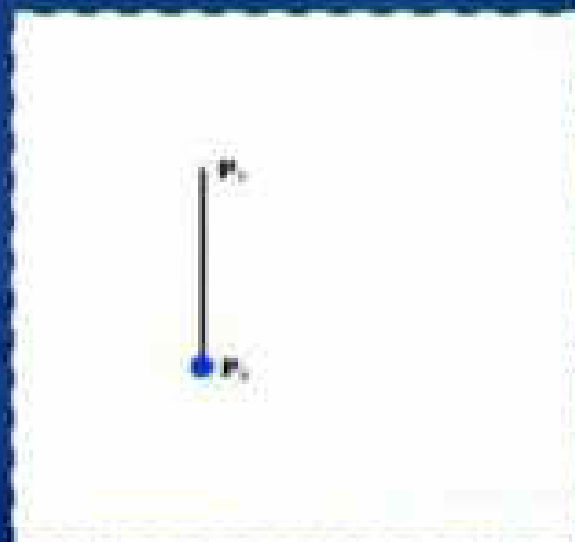




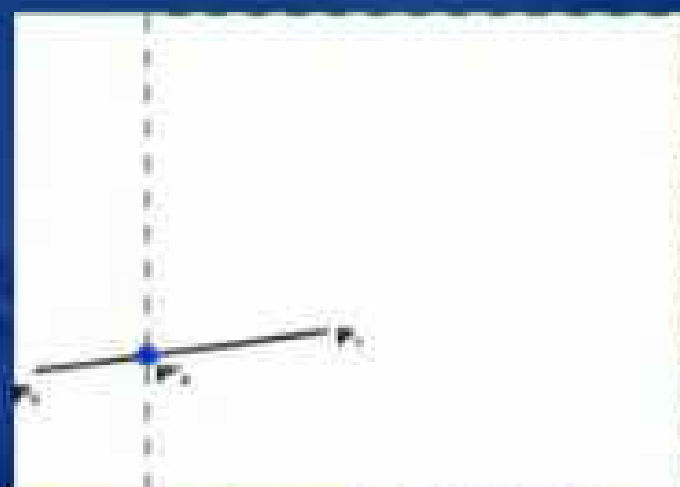
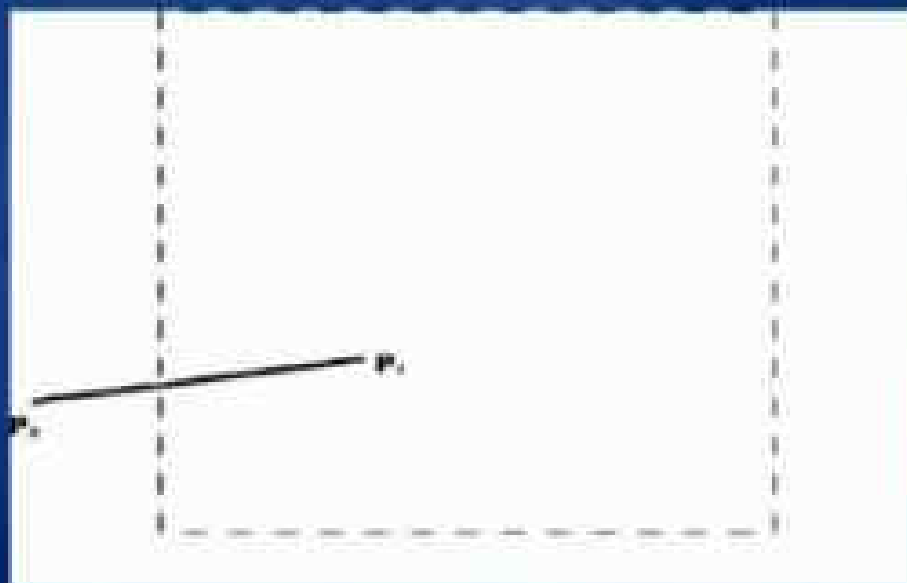
For each boundary if P1 and P2 both are outside,  
reject whole line, output null



For each boundary if  $P_1$  outside  
 $P_2$  in side output intersection  
 point and  $P_2$



For each boundary if  $P_1$  and  
 $P_2$  both are inside, output  $P_2$



For each boundary if  $P_1$  inside and  $P_2$  outside, output intersection point

# Pipeline Clipping Approach

An array,  $s$ , records the most recent point that was clipped for each clip-window boundary.

The main routine passes each vertex  $p$  to the clip Point routine for clipping against the first window boundary.

If the line defined by endpoints  $p$  and  $s$  (boundary) crosses this window boundary, the intersection is calculated and passed to the next clipping stage.

If  $p$  is inside the window, it is passed to the next clipping stage.

Any point that survives clipping against all window boundaries is then entered into the output array of points.

The array first Point stores for each window boundary the first point clipped against that boundary.

After all polygon vertices have been processed, a closing routine clips lines defined by the first and last points clipped against each boundary.

# Shortcoming of Sutherland-Hodgeman Algorithm

Concave polygons may be displayed with extensors lines

## Things to correct concave Polygon

Split the concave polygon into two or more convex polygons and process each convex polygon separetley

Check the final vertex list for multiple vertex points  
along any clip window boundary and correctly join  
pairs

Use a more general polygon clipper

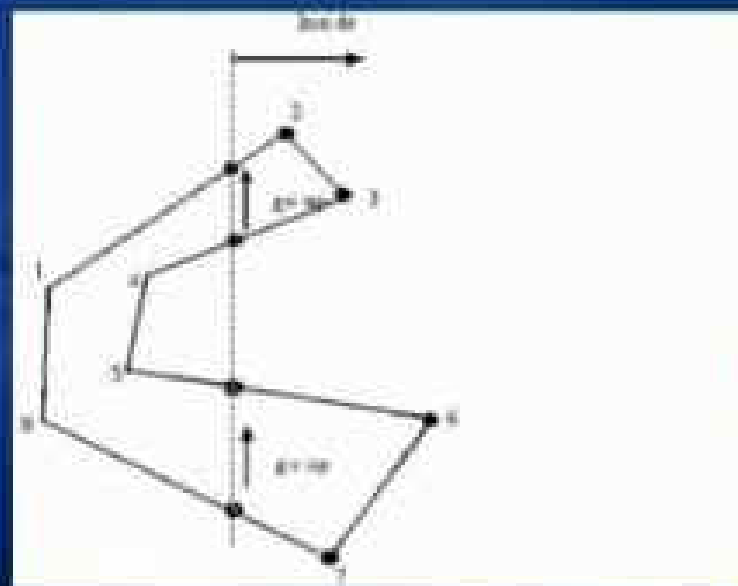
## **Weiler-Atherton Polygon Clipping**

Sometimes follow the window boundaries instead of  
polygon boundary.



# Weiler-Atherton Polygon Clipping

For an inside-to-outside pair of vertices, follow the window boundary in a clockwise direction



# Computer Graphics

## Lecture 15

[www.vuCTn.com](http://www.vuCTn.com)  
**CS602 Complete MERGED**  
**PPT Slides 1 TO 45**

Regards : Zarva Chaudhary

Admin:

\* Zarva Chaudhary \*

\* Laiba Maki \*

Merged File date  
31 -07 -2021

\*Chaudhary Moazzam\*

# Computer Graphics

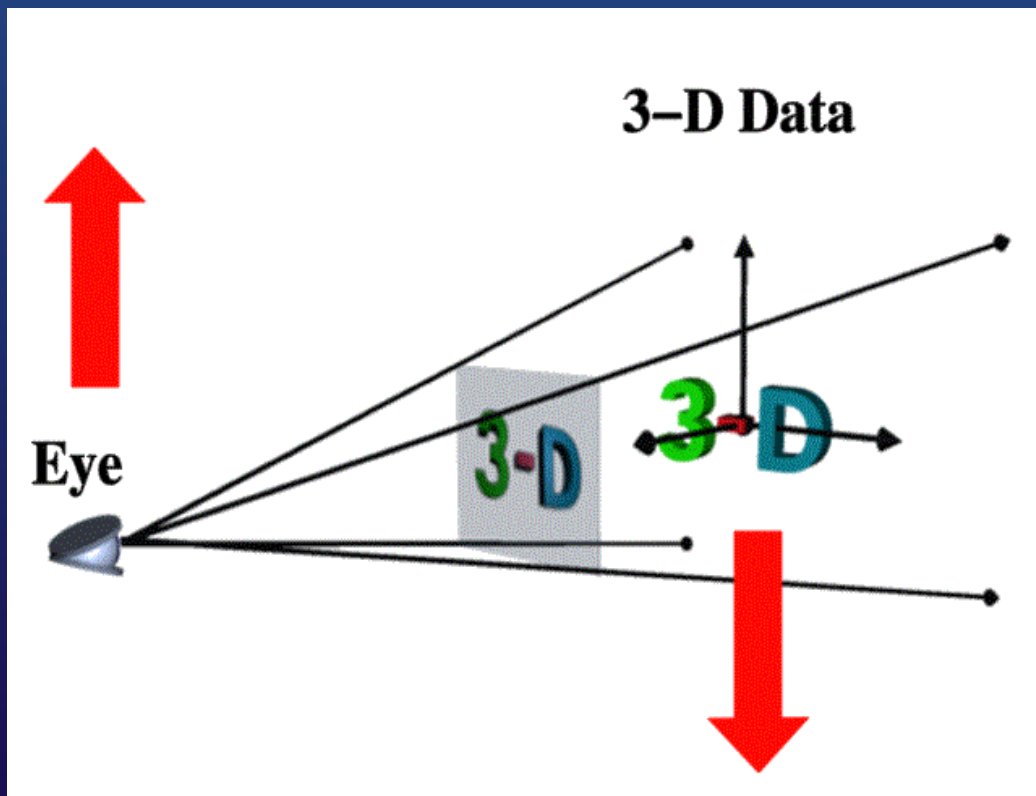
## Lecture 16

### 3-D Concepts

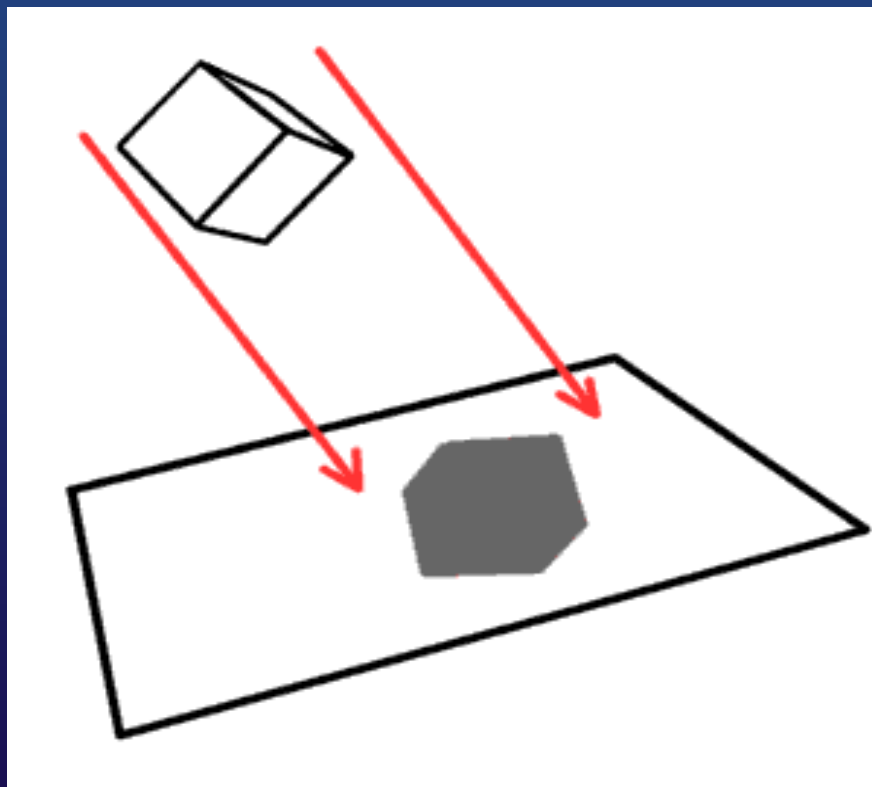
Taqdees A. Siddiqi  
cs602@vu.edu.pk

### 3-Dimensions

In a 3D medium, each of our eyes views the scene from slightly different angles. This is the way we perceive the real world.



**3D Viewing**



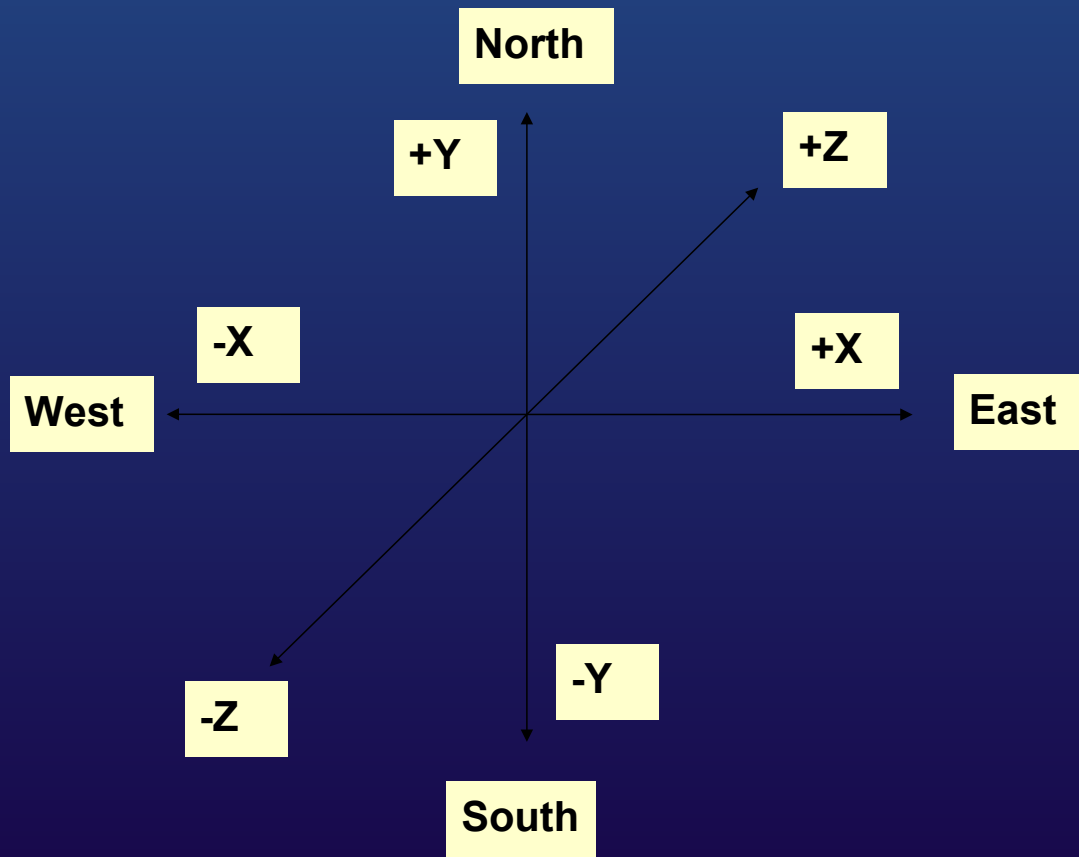
**Casting the shadow of a 3D image**

# Coordinate Systems

Coordinate systems are the measured frames of reference within which geometry is defined, manipulated and viewed.

# Coordinate Systems

In this system, a point serves as the origin (reference point), and three lines (axes) that pass through this point and are orthogonal to each other (at right angles – 90 degrees).



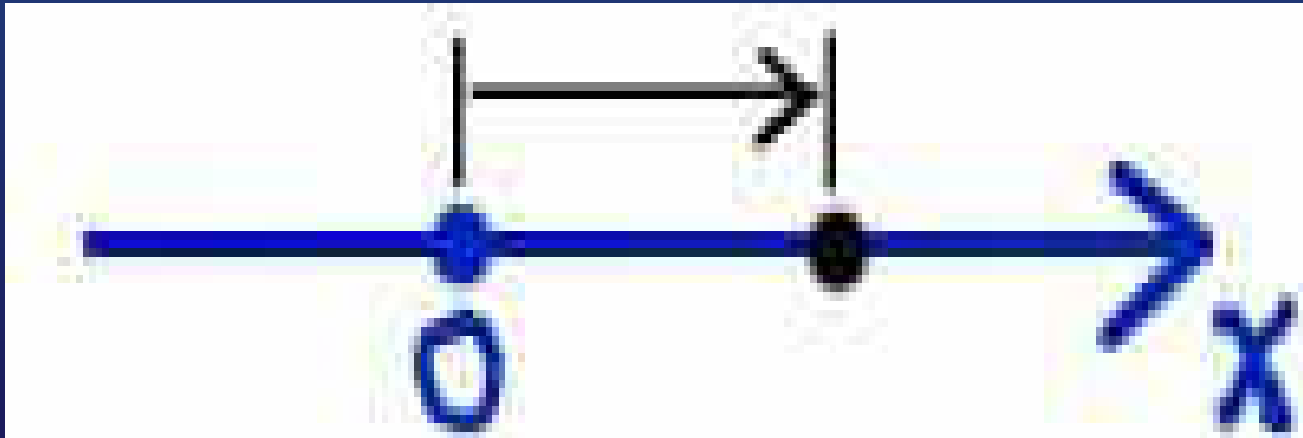
## Types of the coordinate systems

- ◆ Following are three types of the coordinate systems
  - 1-D Coordinate System
  - 2-D Coordinate System
  - 3-D Coordinate System

# 1-D Coordinate System

- ◆ Direction and magnitude along a single axis, with reference to origin
- ◆ Locations are defined by a single coordinate

- ◆ Can define points, segments, lines, rays
- ◆ Can have multiple origins (frames of reference) and transform coordinates among them



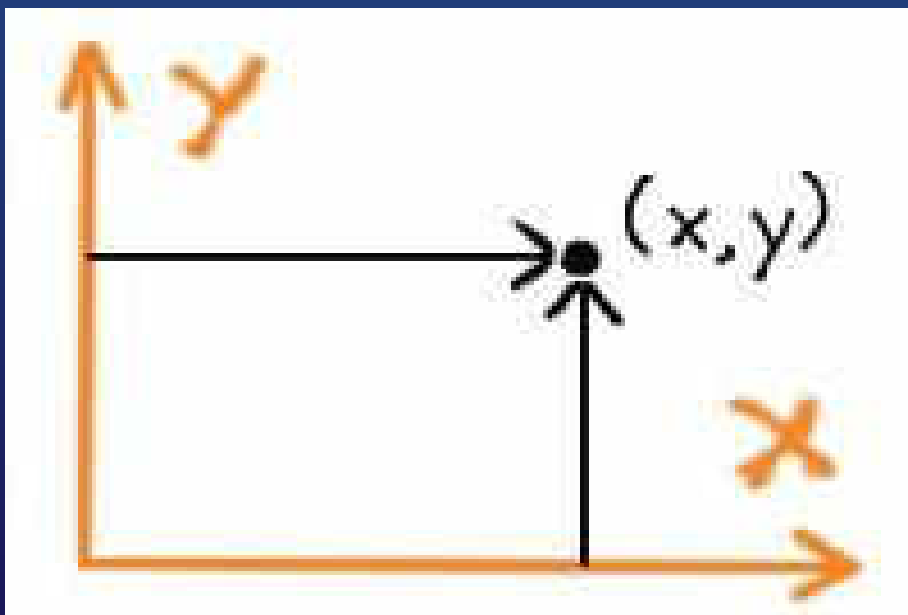
## 1-D Coordinate System

## 2-D Coordinate System

- ◆ Direction and magnitude along two axes, with reference to an origin
- ◆ Locations are defined by x, y coordinate pairs



- ♦ Can define points, segments, lines, rays, curves, polygons, (any planar geometry)
- ♦ Can have multiple origins (frames of reference) and transform coordinates among them

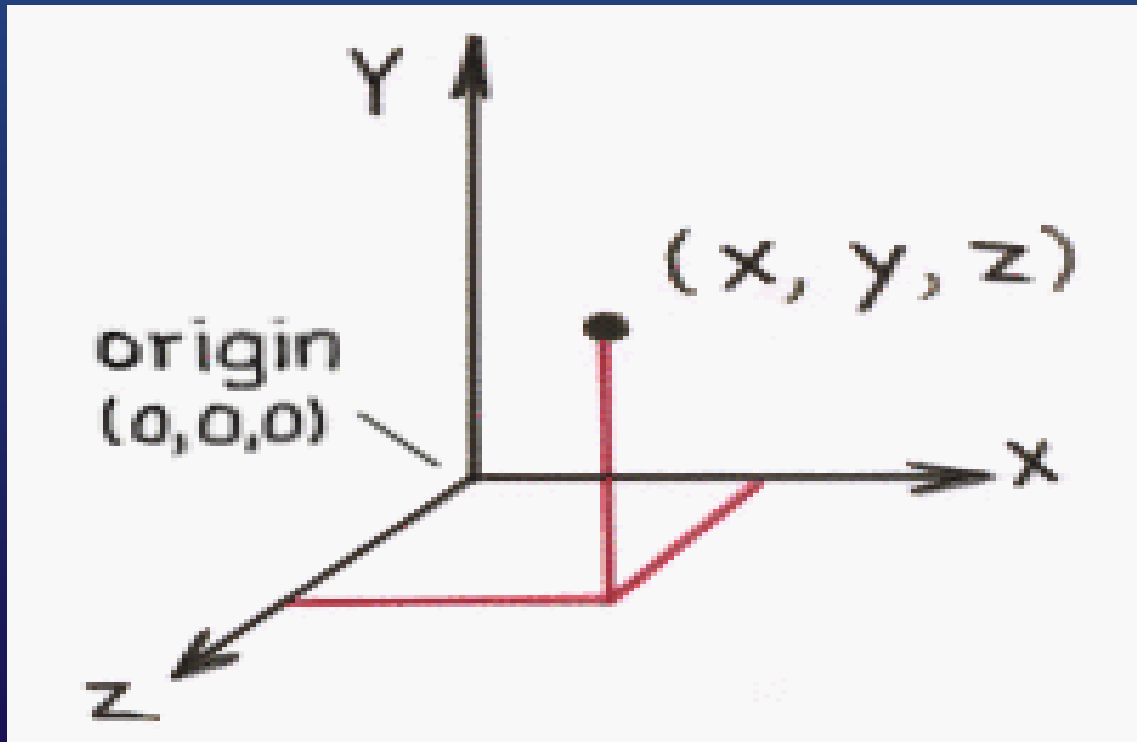


## 2-D Coordinate System

## 3-D Coordinate System

- ◆ It is 3D Cartesian coordinate system
- ◆ Direction and magnitude along three axes, with reference to an origin
- ◆ Locations are defined by  $x, y, z$  triples

- ◆ Can define cubes, cones, spheres, etc., (volumes in space) in addition to all one- and two-dimensional entities
- ◆ Can have multiple origins (frames of reference) and transform coordinates among them



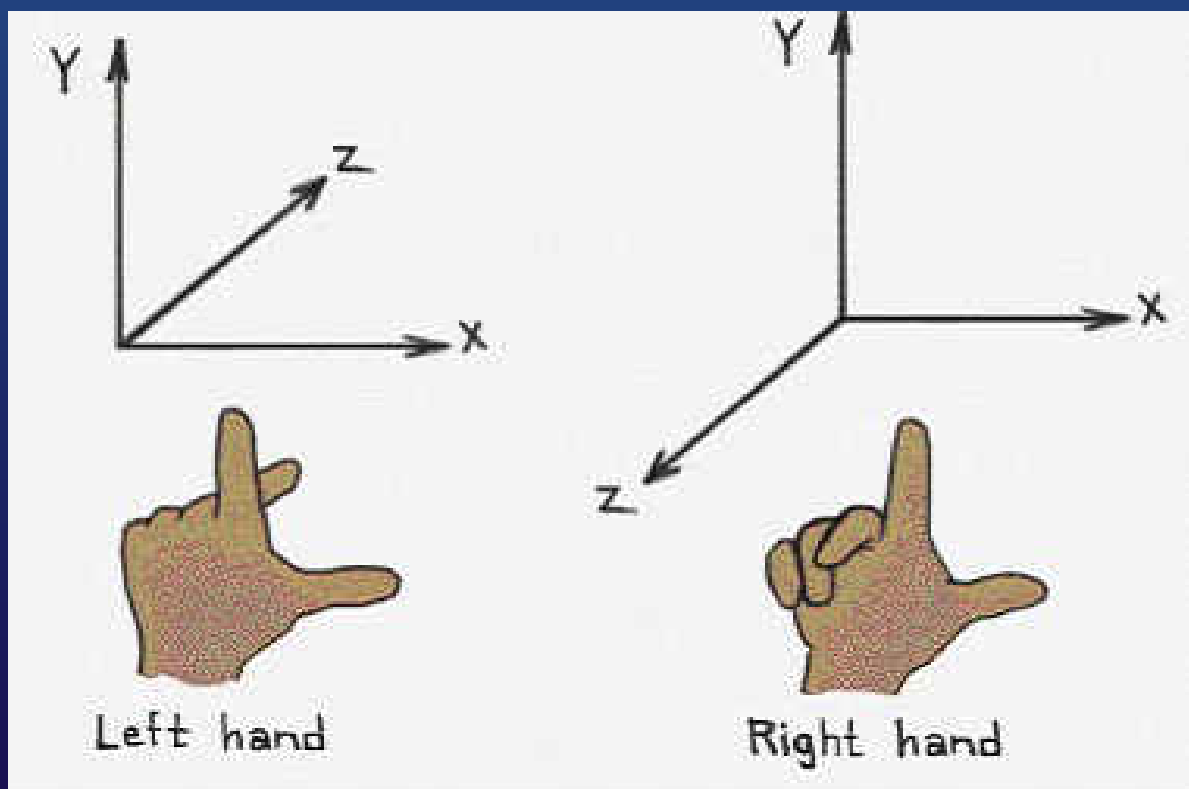
## 3-D Coordinate System

### Left Handed versus Right Handed Rules

These rules determine orientation of axes and direction of rotations

- Thumb = pos x
- Index up = pos y
- Middle out = pos z

- ◆ Most world and object axes tend to be right handed
- ◆ Left handed axes often are used for cameras



**Left handed versus Right Handed Rules**

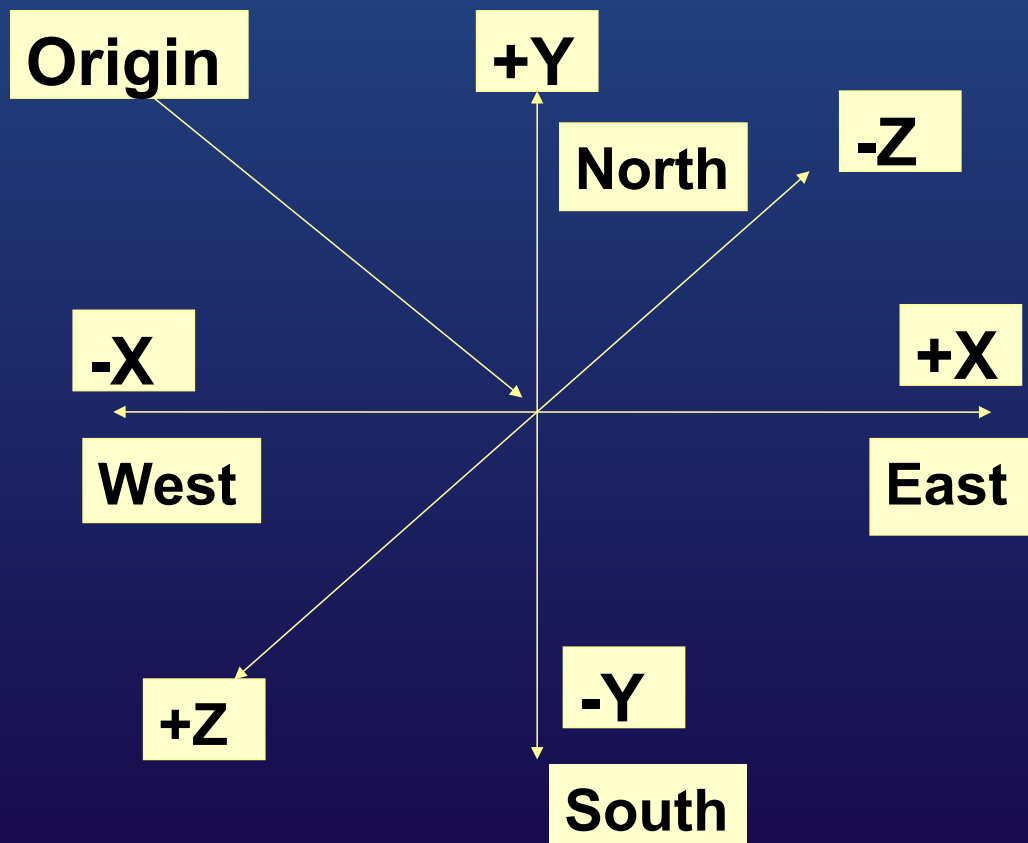
“Right Hand Rule” for rotations: grasp axis with right hand with thumb oriented in positive direction, fingers will then curl in direction of positive rotation for that axis.



## Defining right handed rules

Right handed Cartesian coordinate system describes the relationship of the X, Y and Z in the following manner:

- X is positive to the right of the origin, and negative to the left.
- Y is positive above the origin, and negative below it.
- Z is *negative* beyond the origin, and *positive* behind it.

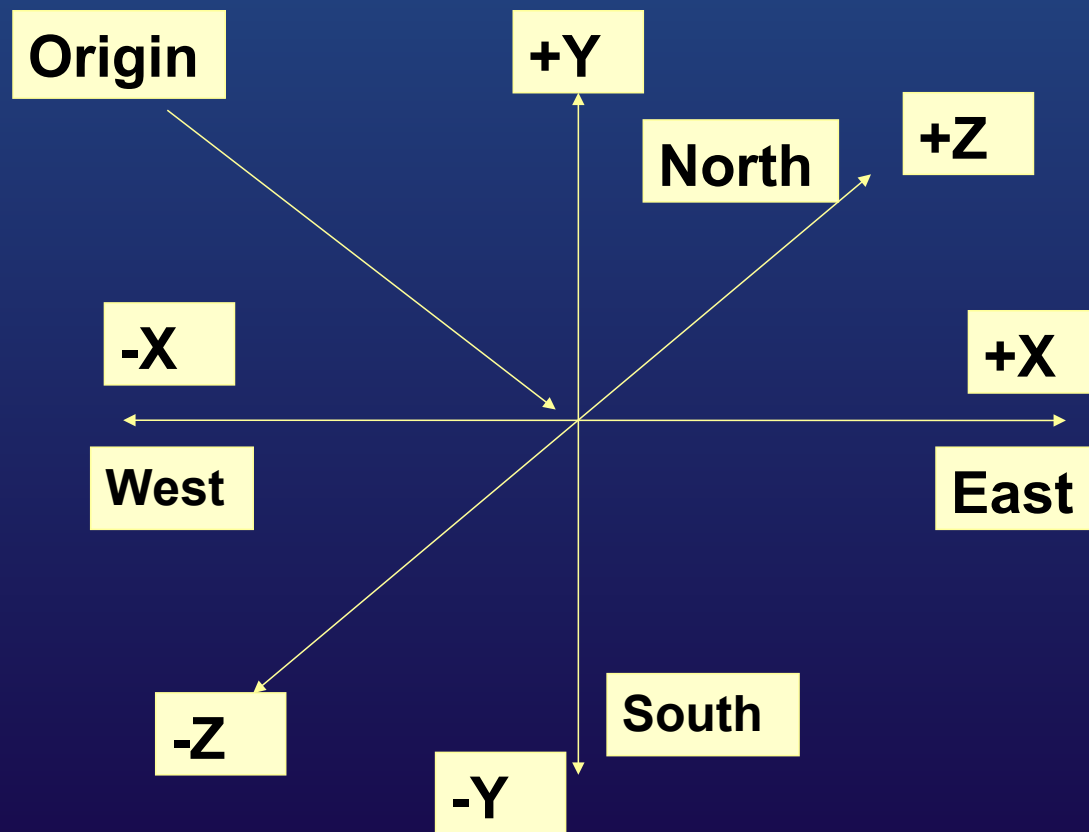


Right hand rule

# Defining Left hand rules

Left handed Cartesian coordinate system describes the relationship of the X, Y and Z in the following manner:

- X is positive to the right of the origin, and negative to the left.
- Y is positive above the origin, and negative below it.
- Z is positive beyond the origin, and negative behind it.



Left hand rule

## Defining 3D points in mathematical notations

- ♦ 3D points can be described using simple mathematical notation

$$P = (x, y, z)$$



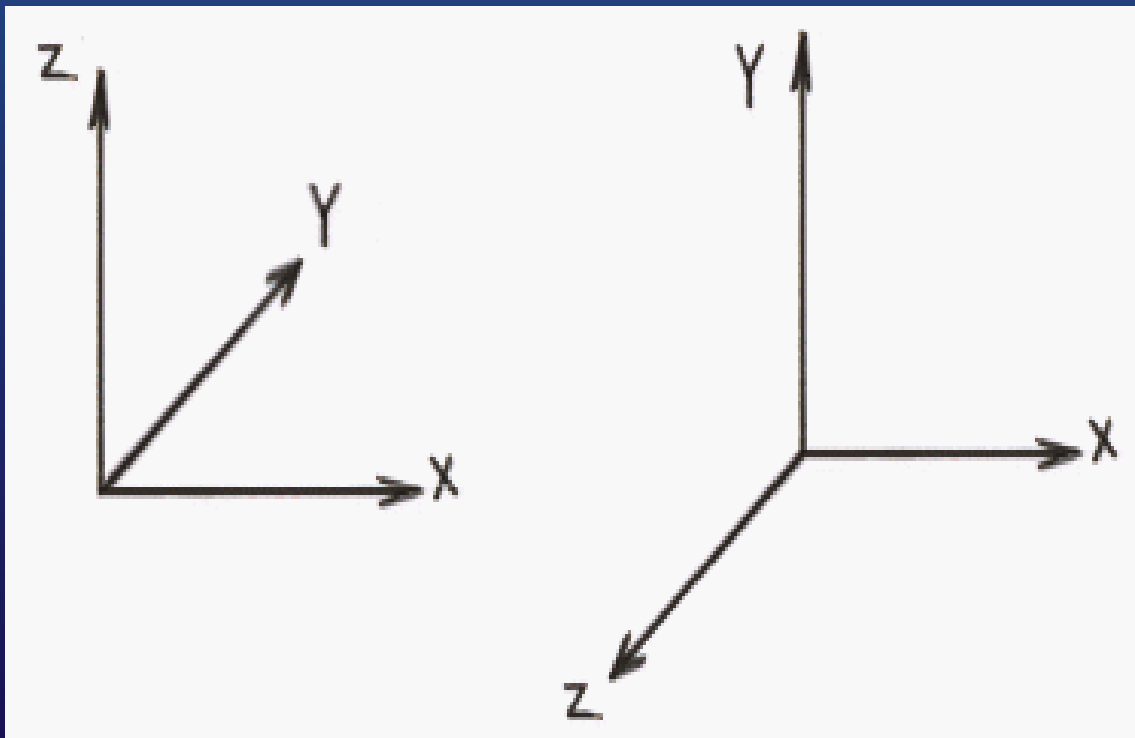
- ◆ Thus the origin of the Coordinate system is located at point  $(0,0,0)$
- ◆ While five units to the right of that position might be located at point  $(5,0,0)$ .

## Y-up versus Z-up

- ◆ z-up typically used by designers
- ◆ y-up typically used by animators

# Y-up versus Z-up

- ♦ orientation by profession supposedly is derived from past work habits
- ♦ often handled differently when moving from application to application

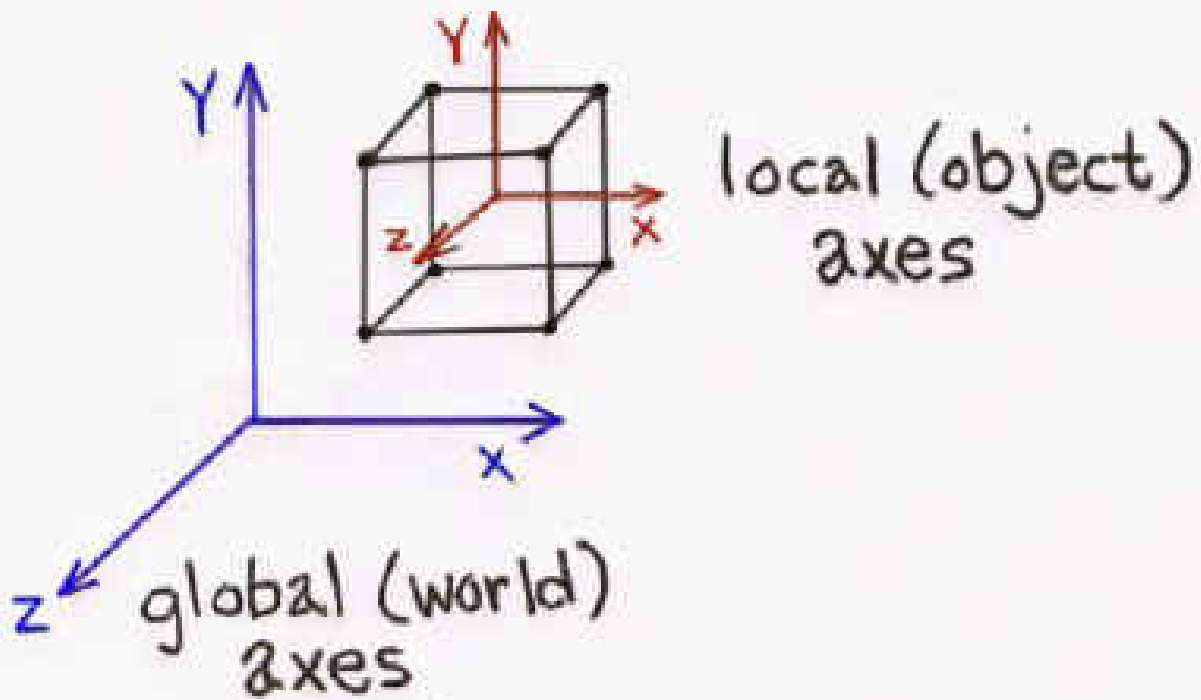


Y-up versus Z-up

# Global and Local Coordinate Systems

- ◆ Local coordinate systems can be defined with respect to global coordinate system

- ◆ Locations can be relative to any of these coordinate systems
- ◆ Locations can be translated or "transformed" from one coordinate system to the other.

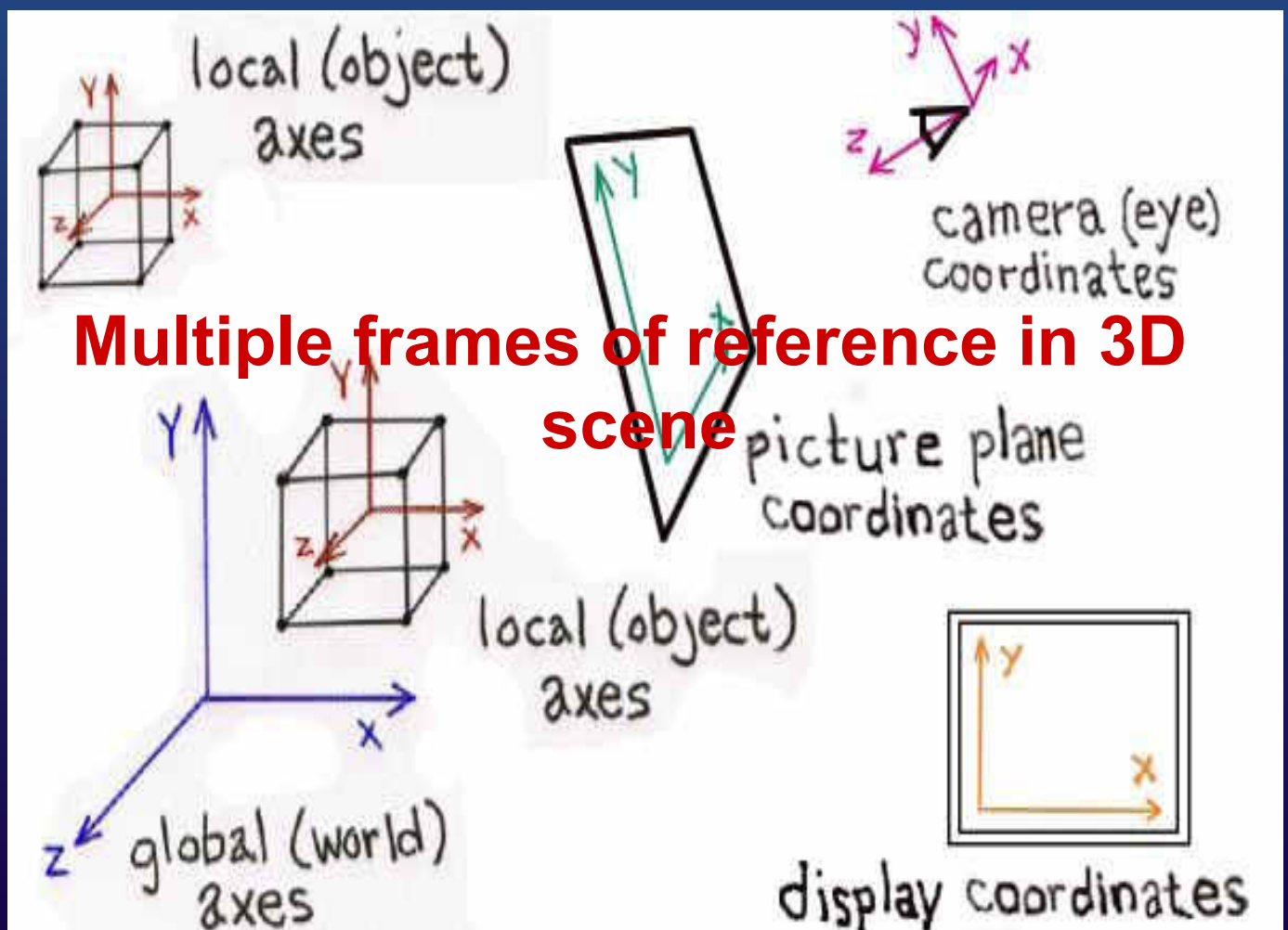


## Global and Local Coordinate systems

### Multiple Frames of Reference in a 3-D Scene

- ◆ Multiple Coordinate Systems within any 3-D scene

- ◆ Application data will be transformed among the various coordinate systems, depending on what's to be accomplished
- ◆ Individual coordinate systems often are hierarchically linked within the scene



# Defining points in C language structure

Define a point in 3D

How far East, Up, and North it is from origin.

The center of your computer screen?

If it would be at a point such as “1.5 feet east, 4.0 feet up, 7.2 feet north.”

A data structure is required to hold these values.

# POINT3D structure

- ◆ typedef struct \_POINT3D{
- ◆ float x;
- ◆ float y;
- ◆ float z;
- ◆ } POINT3D;
- ◆ POINT3D screenCenter = {1.5, 4.0, 7.2};

## The Polar Coordinate System

Is Cartesian system the only choice? No. We could also described the object position in this way: “starting at the origin, looking east, rotate 38° northward, 65° upward, and travel 7.47 feet along this line”.

This is less intuitive in a real world setting.

If you try to work out the math, it is harder to manipulate (transformation).

Because such polar coordinates are difficult to control.

## Defining Geometry in 3-D

- ◆ Here are some definitions of the technical terms that will be used in 3D.

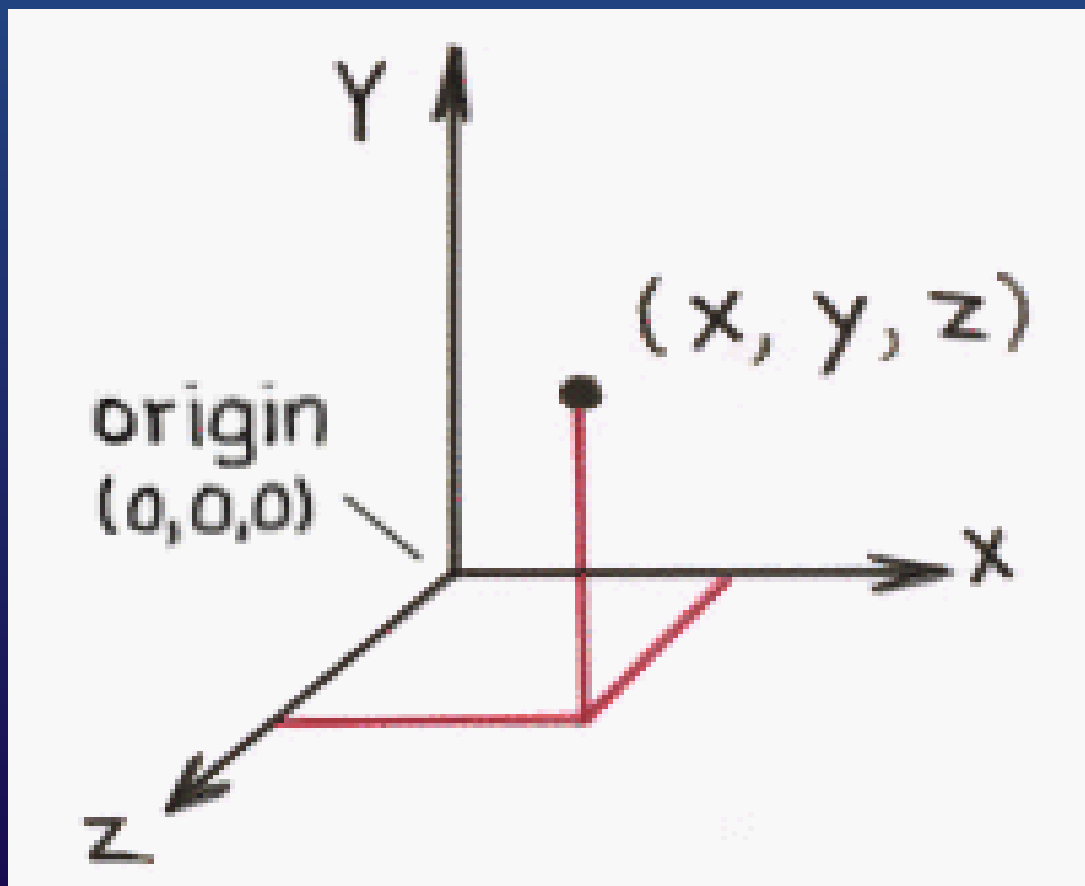


# Modeling

- ◆ Modeling is the process of describing an object or scene so that we can construct an image of it

## Points

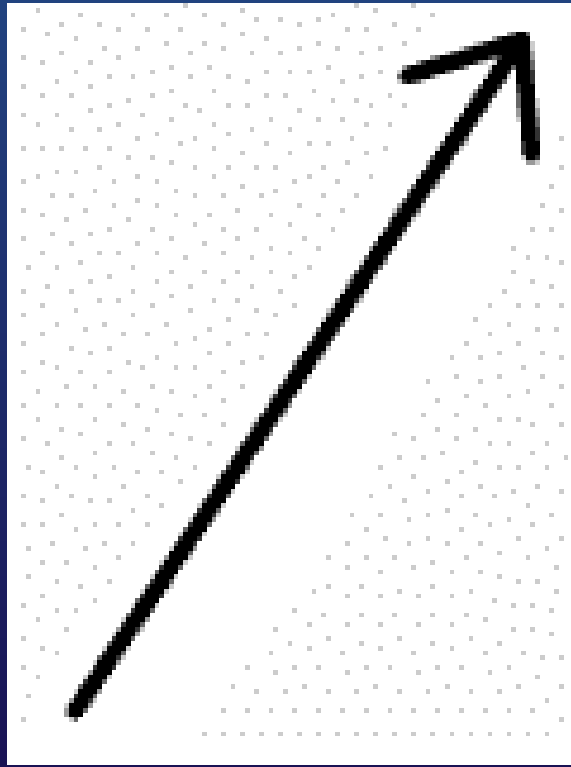
- Points are the three-dimensional locations (or coordinate triples)



Point (X,Y,Z) in Cartesian Coordinate System

## Vectors

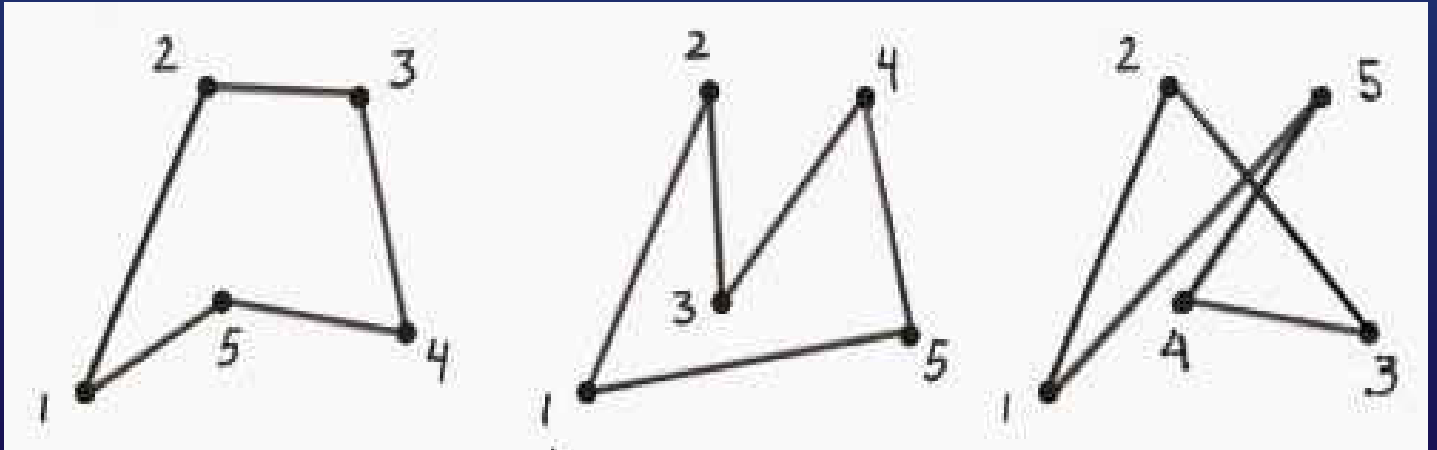
- Vectors have direction and magnitude; can also be thought of as displacement



Vector

## Polygons

- sequences of “correctly” co-planar points; or an initial point and a sequence of vectors



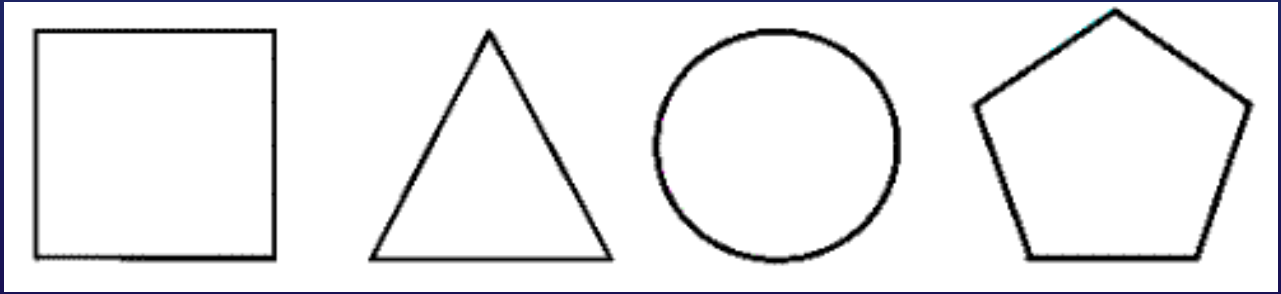
## Polygons

## Primitives

- ◆ Primitives are the fundamental geometric entities within a given data structure.
- ◆ We have already touched on point, vector and polygon primitives

# Primitives

- Regular Polygon Primitives - square, triangle, circle, n-polygon, etc.

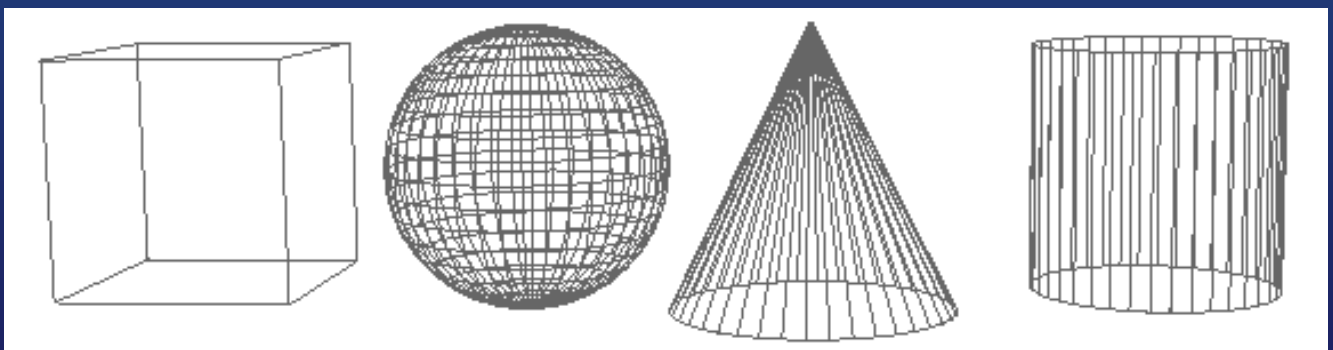


- ◆ Polygon strips or meshes
- ◆ Meshes provide a more economical description than multiple individual polygons

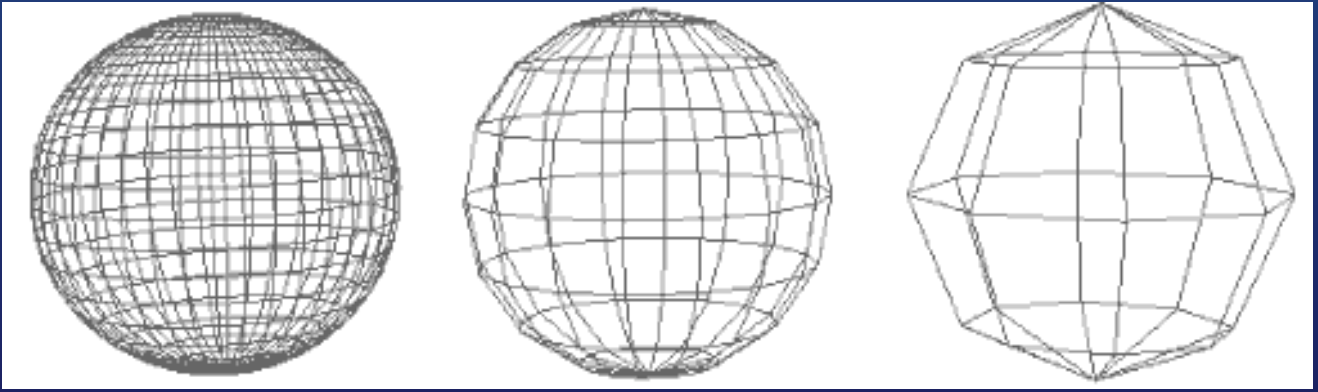
- ◆ 100 individual triangles, each having 3 vertices, would require 100 x 3 or 300 vertex definitions to be stored

- ◆ In contrast, triangle strips require  $n + 2$  vertex definitions for  $n$  number of triangles in the strip. Hence, a 100 triangles strip requires only 102 unique vertex definitions.

Meshes also provide continuity across surfaces  
which is important for shading calculations



3D primitives in a polygonal database



3D shapes are represented by polygonal meshes that define or approximate geometric surfaces.

- ◆ With curved surfaces, the accuracy of the approximation is directly proportional to the number of polygons used in the representation.



- ◆ More polygons (when well used) yield a better approximation.
- ◆ But more polygons also exact greater computational overhead, thereby degrading interactive performance, increasing render times, etc.

## Rendering

The process of computing a 2-D image using a combination of a 3-D database, scene characteristics, and viewing transformations.

Various algorithms according to the needs of the application.

# Tessellation

- ◆ The subdivision of an entity or surface into one or more non-overlapping primitives. Typically, renderers decompose surfaces into triangles as part of the rendering process.

# Sampling

The process of selecting a representative but finite number of values along a continuous function sufficient to render a reasonable approximation of the function for the task at hand.

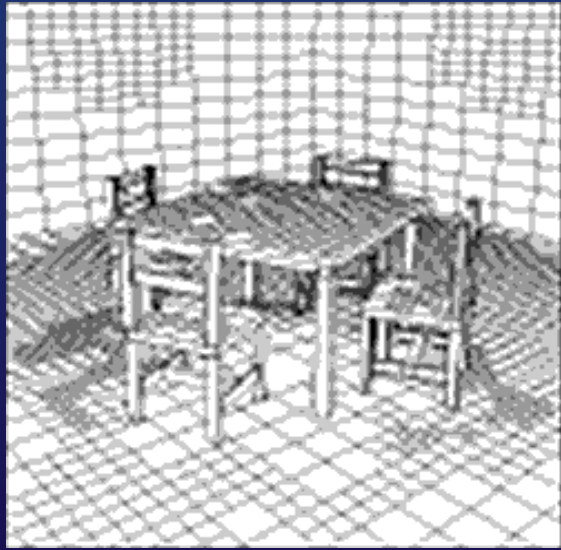
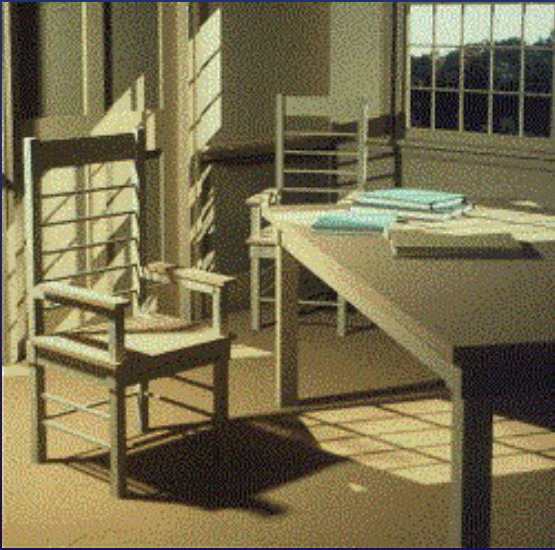
# Level of Detail (LOD)

To improve rendering efficiency when dynamically viewing a scene, more or less detailed versions of a model may be swapped in and out of the scene database depending on the importance (usually determined by image size) of the object in the current view.

## Surface normal

- ◆ A vector perpendicular to a surface and “outward” facing
- ◆ Surface normal are used to determine visibility and also in the calculation of shading values

# 3D Model and Lighting Effects



With and Without lighting

Computer Graphics

Lecture 16

# Computer Graphics

## Lecture 17

### 3-D Transformations-I

Taqdees A. Siddiqi  
cs602@vu.edu.pk

# Computer Graphics

## Lecture 17

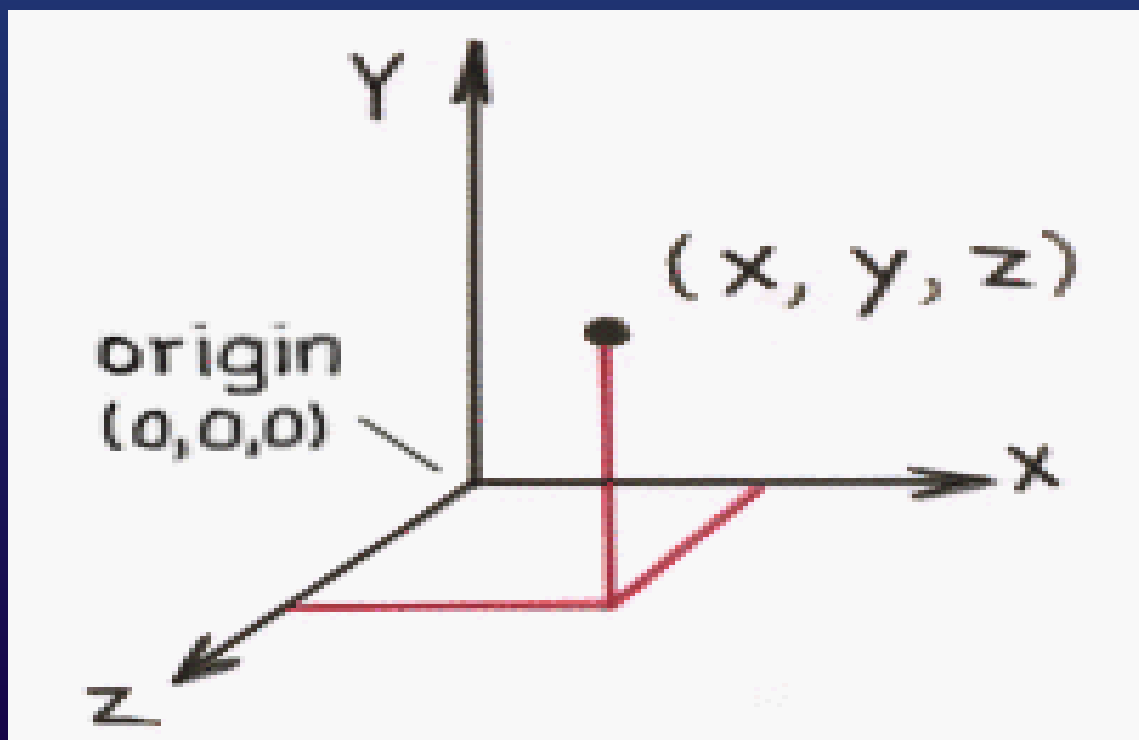
### 3-D Transformations-I

Taqdees A. Siddiqi  
cs602@vu.edu.pk

# Definition of a 3D Point

- ◆ A 3-D point is similar to its 2D counterpart; we simply add an extra component, Z, for the 3rd axis:
- ◆  $P = (x,y,z)$

## 3D Point



# Distance between Two 3D Points

- ◆ The distance between two points  $(A_x, A_y, A_z)$  and  $(B_x, B_y, B_z)$  can be found by again using the pythagorus theorem

## Pythagorus Theorem

- ◆ Distance between A and B =  
 $\text{sqrt}(dx^2 + dy^2 + dz^2)$   
where
  - $dx = A_x - B_x$
  - $dy = A_y - B_y$
  - $dz = A_z - B_z$

# Definition of a 3D Vector

- ◆ Like it's 2D counterpart, a vector can be thought of in two ways: either a point at  $(x,y,z)$  or a line going from the origin  $(0,0,0)$  to the point  $(x,y,z)$ .

## Length of vector

- ◆ To calculate the length of a vector we simply calculate the distance between the origin and the point at  $(x,y, z)$  :



# Length of vector

$$\begin{aligned}\text{length} &= | (x,y,z) - (0,0,0) | \\ &= \text{sqrt}( (x-0)*(x-0) + \\ &\quad (y-0)*(y-0) + \\ &\quad (z-0)*(z-0) ) \\ &= \text{sqrt}(x*x + y*y + z*z)\end{aligned}$$

## Unit Vector

- ◆ Often in 3D computer graphics you need to convert a vector to a unit vector, i.e. a vector that points in the same direction but has a length of 1.

# Finding Unit Vector

- ◆ This is done by simply dividing each component by the length:

# Finding Unit Vector

- ◆ Let  $(x,y,z)$  be our vector
- ◆  $\text{length} = \sqrt{x^2 + y^2 + z^2}$
- ◆ Unit vector =

$$\left| \frac{x}{\text{length}}, \frac{y}{\text{length}}, \frac{z}{\text{length}} \right|$$

Where  $\text{length} = |(x,y,z)|$

- ◆ Note that if the vector is already a unit vector then the length will be 1, and the new values will be the same as the old ones.

## 3D Vector Addition

Adding vector to point

`point.x = point.x + vector.x;`

`point.y = point.y + vector.y;`

`point.z = point.z + vector.z;`

# Operations between Points and Vectors

- ◆  $\text{point} - \text{point} \Rightarrow \text{vector}$
- ◆  $\text{point} + \text{point} =$   
 $\text{point} - (-\text{point}) \Rightarrow \text{vector}$
- ◆  $\text{vector} - \text{vector} \Rightarrow \text{vector}$

# Operations between Points and Vectors

- ◆  $\text{vector} + \text{vector} \Rightarrow \text{vector}$
- ◆  $\text{point} + \text{vector} \Rightarrow \text{point}$
- ◆  $\text{point} - \text{vector} =$   
 $\text{point} + (-\text{vector}) \Rightarrow \text{point}$

# Multiplying: Scalar Multiplication

- ◆ Multiplying a vector by a scalar ( a number with no units) :
- ◆  $\text{Vector.x} = \text{Vector.x} * \text{Scalar}$
- ◆  $\text{Vector.y} = \text{Vector.y} * \text{Scalar}$
- ◆  $\text{Vector.z} = \text{Vector.z} * \text{Scalar}$

## Example

- ◆ A vector of length 4
- ◆ if multiplied by 2.5
- ◆ Yields a vector of length 10
- ◆ in the same direction as original

# Example

- ◆ A vector of length 4
- ◆ If multiplied by -2.5 instead,
- ◆ Yields a vector of length 10;
- ◆ but in the direction opposite to original

## Vector Multiplication

1. Vectors can be multiplied in two ways:
  1. Dot Product
  2. Cross Product

# Dot Product

- ◆ The dot product of two vectors A & B is defined by the formula:
- ◆  $A \cdot B = A.x \cdot B.x + A.y \cdot B.y + A.z \cdot B.z$

## Another definition of Dot Product

$$A \cdot B = |A| \cdot |B| \cdot \cos(\theta)$$

Where  $\theta$  is the angle between the two vectors

# Properties of Cos( $\theta$ )

$$\cos(\theta) > 0$$

if  $\theta < 90^\circ$  or  $\theta > 270^\circ$

$$\cos(\theta) < 0$$

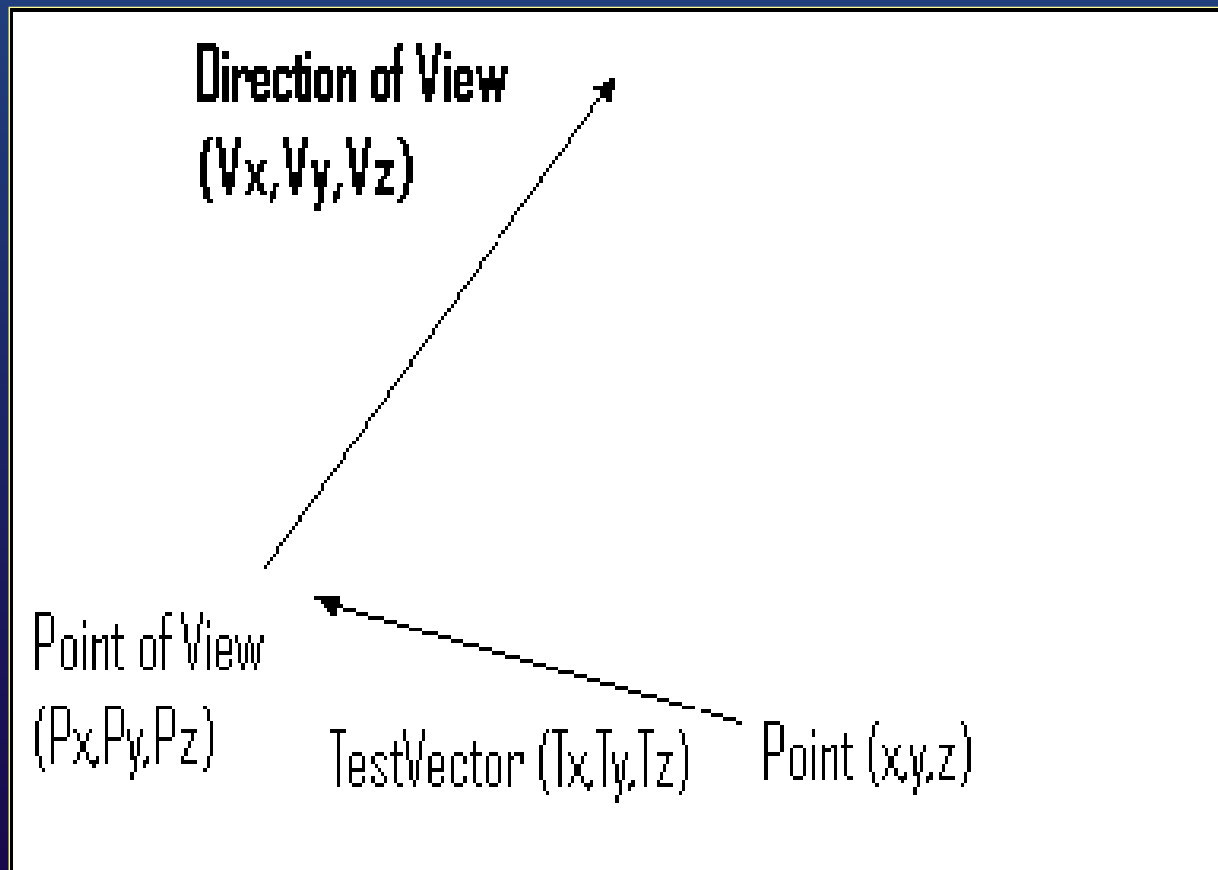
if  $\theta$  is  $> 90^\circ$  and  $\theta < 270^\circ$

$$\cos(\theta) = 0 \quad \text{if } \theta = 90^\circ \text{ or } \theta = 270^\circ$$

## Use of Dot Product

- ◆ If point of view is at  $(p_x, p_y, p_z)$ .
- ◆ and is looking along the vector  $(v_x, v_y, v_z)$ ,
- ◆ we have a point P  $(x, y, z)$  in space.





- ◆ we want to know if  $P$  is visible from the point-of-view (POV), or if  $P$  is “behind” the POV

# Dot Product - Example

- ◆ Point3D pov;
- ◆ Vector3D povDir;
- ◆ Point3D test;
- ◆ Vector3D vTest
- ◆ float dotProduct;

## Example cont...

- ◆  $vTest.x = pov.x - test.x;$
- ◆  $vTest.y = pov.y - test.y;$
- ◆  $vTest.z = pov.z - test.z;$

## Example cont...

dotProduct =

$$\begin{aligned} & \text{vTest.x} * \text{povDir.x} + & \text{vTest.y} * \text{povDir.y} + \\ & \text{vTest.z} * \text{povDir.z} \end{aligned}$$

if(dotProduct > 0)

- ◆ point is “in front of “ POV
- ◆ else if (dotProduct < 0)
- ◆ point is “behind” POV
- ◆ else point is orthogonal to the POV direction

# Cross Product

- ♦ for Vectors A, B
- ♦  $A \times B = (A.y * B.z - A.z * B.y, A.z * B.x - A.x * B.z, A.x * B.y - A.y * B.x)$

- ♦ This formula for  $A \times B$  came from the determinate of order 3 of the matrix:

$$\begin{vmatrix} X & Y & Z \\ A.x & A.y & A.z \\ B.x & B.y & B.z \end{vmatrix}$$

# Cross Product for physicists

- ◆  $|A \times B| = |A| * |B| \sin(\theta)$
- ◆ Where  $\theta$  is the angle between the two vectors

## A Question

- ◆ A static set of 3D points or other geometric shapes on screen are not very interesting, are they ?

# Transformations

## Transformations

- ◆ The process of moving points in space is called transformation

# Types of Transformation

- ◆ Translation
- ◆ Rotation
- ◆ Scaling
- ◆ Reflection
- ◆ Shearing

## Translation

- ◆ Translation is used to move a point, or a set of points, linearly in space

- ◆ Translation Distances  $t_x, t_y, t_z$
- ◆ For point  $P(x,y,z)$  after translation we have  $P'(x',y',z')$
- ◆  $x' = x + t_x, y' = y + t_y, z' = z + t_z$  ( $t_x, t_y, t_z$ ) is Translation vector

Now  $x' = x + t_x, y' = y + t_y$

and  $z' = z + t_z$  can be expressed as a single matrix equation:

$$P' = P + T$$

Where  $P =$

$P' =$      $T =$

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix}$$

$$\begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$$



## 3D Translation Example

- ◆ you may want to move a point “3 meters east, -2 meters up, and 4 meters north.”

## Steps for Translation

- ◆ Point3D point = (0, 0, 0)
- ◆ Vector3D vector =  
(10, -3, 2.5)
- ◆ *Adding vector to point*

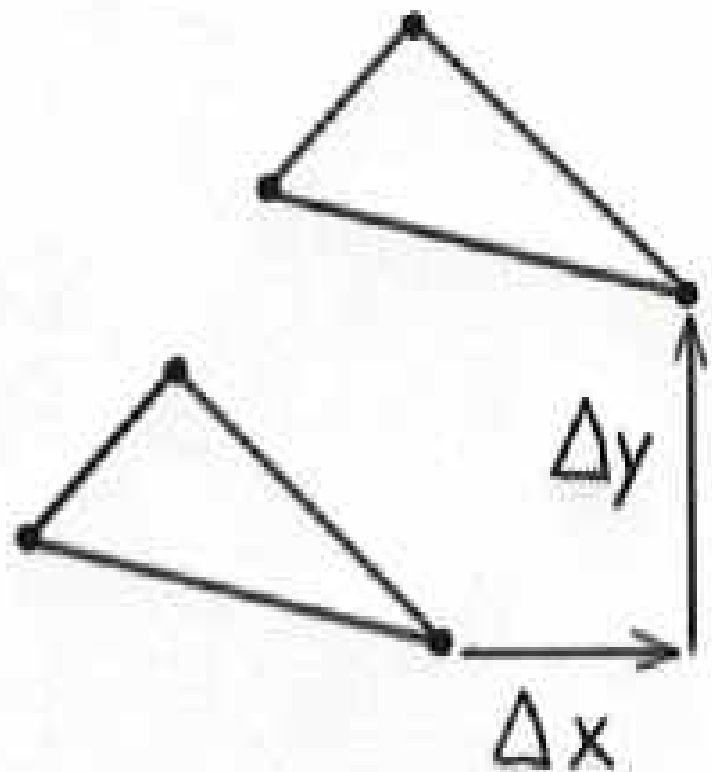
# Steps for Translation

- ◆ `point.x = point.x + vector.x;`
- ◆ `point.y = point.y + vector.y;`
- ◆ `point.z = point.z + vector.z;`
- ◆ And finally we have translated point

$$x' = x + \Delta x$$

$$y' = y + \Delta y$$

$$z' = z + \Delta z$$



# Homogeneous Coordinates

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Abbreviated as:

$$P' = T(t_x, t_y, t_z) \cdot P$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ z + t_z \\ 1 \end{bmatrix}$$

# Computer Graphics

## Lecture 17

# Computer Graphics

## Lecture 18

## 3-D Transformations-II

Taqdees A. Siddiqi  
cs602@vu.edu.pk

# Rotation

Rotation is the process of moving a point in space in a non-linear manner

# Normalization

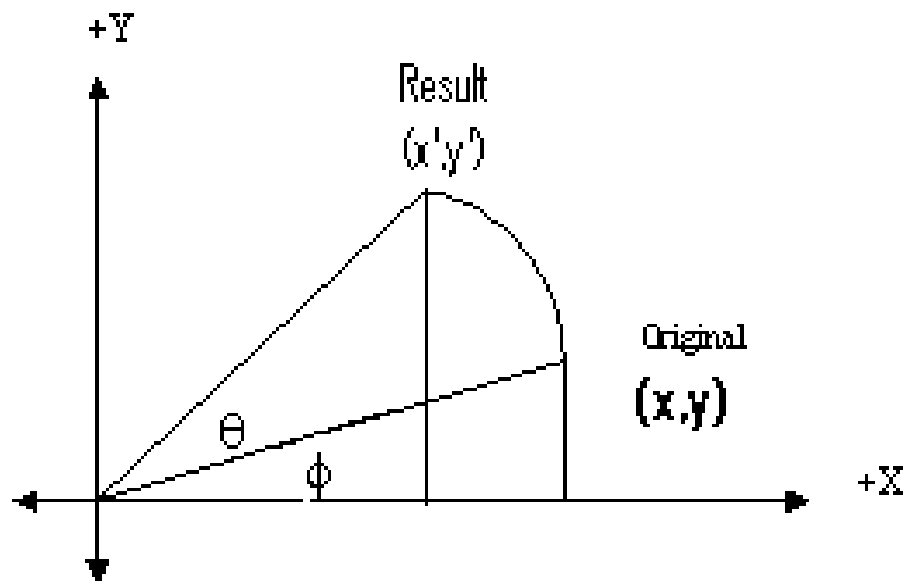
Note that the process of moving the points so that the POV is at the origin looking down the +Z axis is called normalization.

# For Rotation

- Rotating a point requires that we know:
  - 1) the coordinates for the point, and
  - 2) the rotation angles

- ◆ We need to know three different angles:
- ◆ How far to rotate around the X axis (YZ rotation, or “pitch”)
- ◆ How far to rotate around the Y axis (XZ rotation, or “yaw”)
- ◆ How far to rotate around the Z axis (XY rotation, or “roll”)

# Rotating a Point



- ◆ Repositions an object along a circular path in  $xy$ -plane
- ◆ Rotation Angle  $\theta$
- ◆ Rotation Point  $(x_r, y_r)$
- ◆  $\theta$  is +ve  $\rightarrow$  counterclockwise rotation
- ◆  $\theta$  is -ve  $\rightarrow$  clockwise rotation
- ◆  $\theta$  is zero  $\rightarrow$  ?

- ◆ For simplicity, consider the pivot at origin and rotate point  $P(x, y)$   
where  $x = r \cos\Phi$  and  $y = r \sin\Phi$
- ◆ If rotated by  $\theta$  then:
- ◆  $x' = r \cos(\Phi + \theta)$   
 $= r \cos\Phi \cos\theta - r \sin\Phi \sin\theta$   
and
- ◆  $y' = r \sin(\Phi + \theta)$   
 $= r \cos\Phi \sin\theta + r \sin\Phi \cos\theta$

- ◆ Replacing  $r \cos\Phi$  with  $x$  and  $r \sin\Phi$  with  $y$ , we have:
- ◆  $x' = x \cos\theta - y \sin\theta$   
and
- ◆  $y' = x \sin\theta + y \cos\theta$



- ◆ Column vector representation:

$$P' = R \cdot P$$

- ◆ Where

$$P' = \begin{bmatrix} x' \\ y' \end{bmatrix}$$

$$R = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}$$

$$P = \begin{bmatrix} x \\ y \end{bmatrix}$$

## Rotation: Homogeneous Coordinates

The rotation can now be expressed using homogeneous coordinates as:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

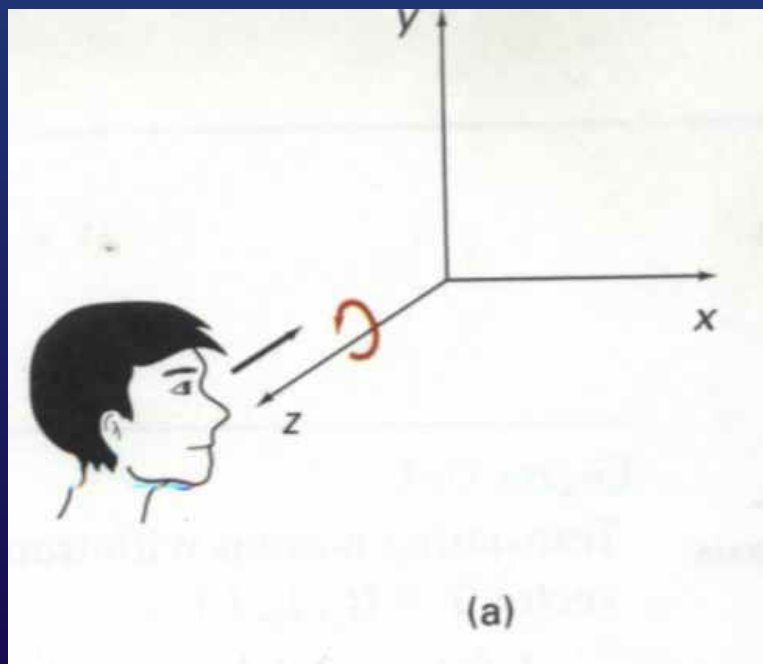
Abbreviated as:

$$P' = R(\theta) \cdot P$$

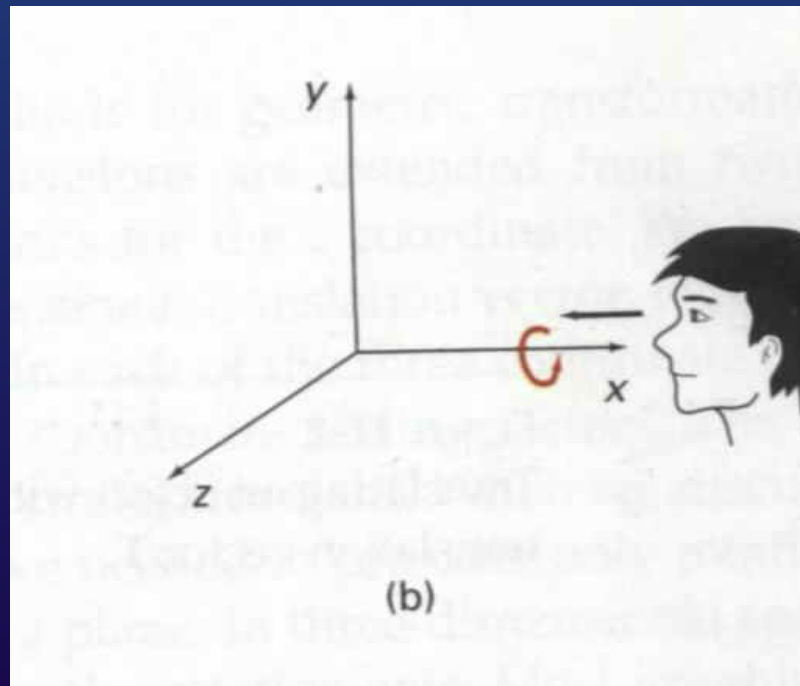
## ... Now in 3D

- ♦ Rotation can be about any of the three axes:
- ♦ About z-axis (i.e. in xy plane)
- ♦ About x-axis (i.e. in yz plane)
- ♦ About y-axis (i.e. in xz plane)

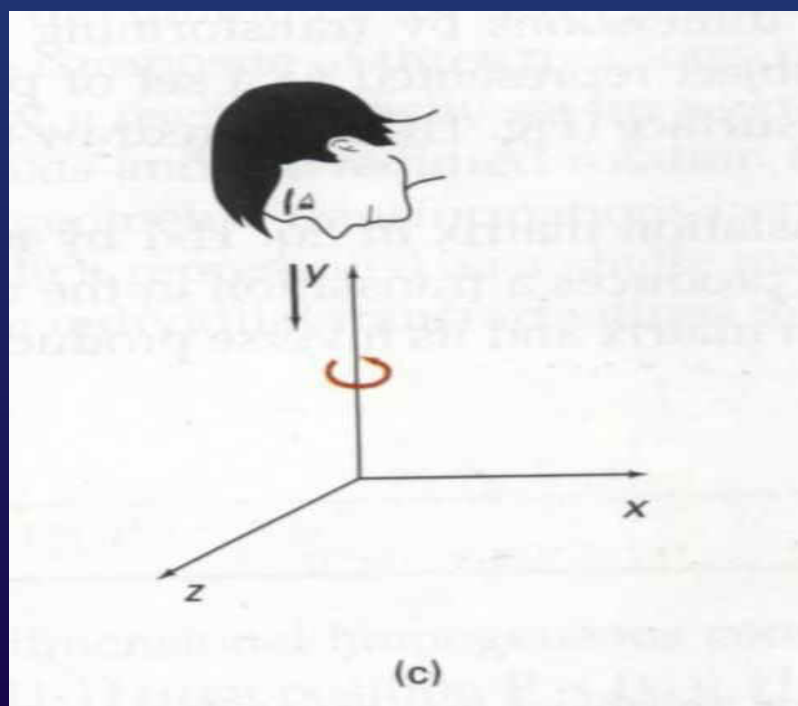
### Roll : around z-axis



## Pitch: around x-axis



## Yaw: around y-axis



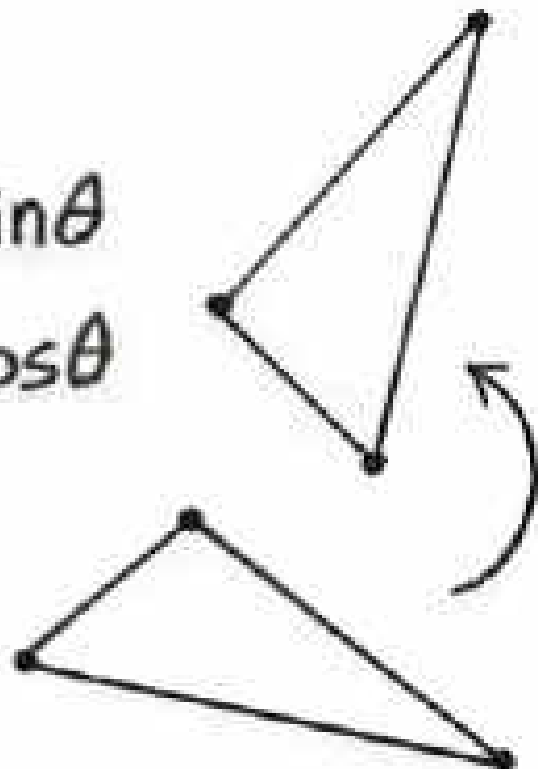
♦ Rotation about z-axis

(i.e. in xy plane):

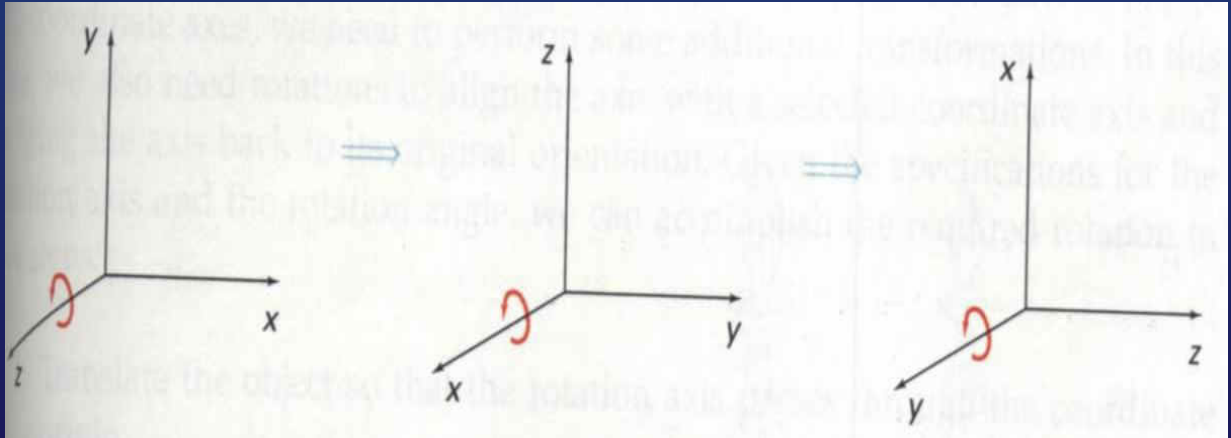
- $x' = x \cos\theta - y \sin\theta$
- $y' = x \sin\theta + y \cos\theta$
- $z' = z$

$$x' = x \cos\theta - y \sin\theta$$

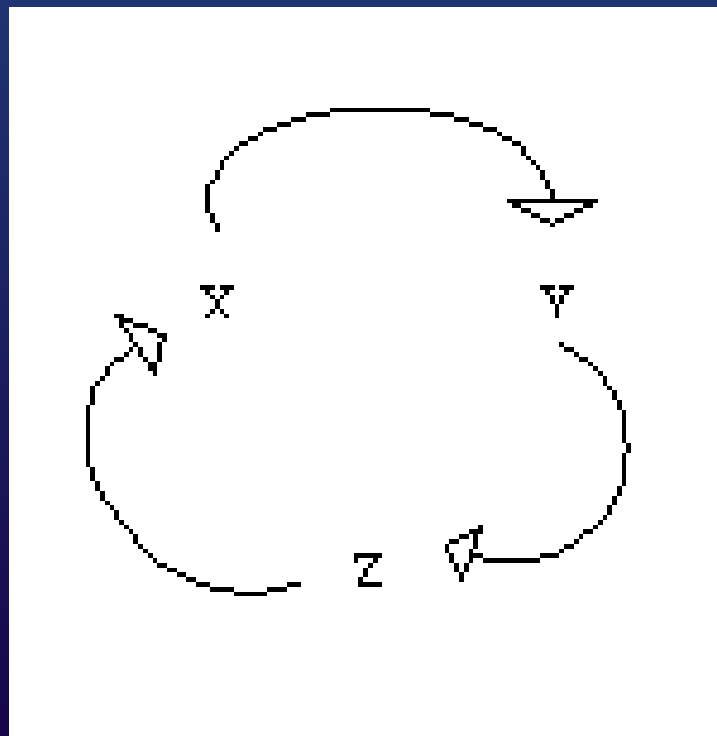
$$y' = x \sin\theta + y \cos\theta$$



Alright, but what about rotations w.r.t. other axes?



## Cyclic permutation



Rotation about x-axis

(i.e. in yz plane):

$$x' = x$$

$$y' = y \cos\theta - z \sin\theta$$

$$z' = y \sin\theta + z \cos\theta$$

Rotation about y-axis

(i.e. in xz plane):

$$x' = z \sin\theta + x \cos\theta$$

$$y' = y$$

$$z' = z \cos\theta - x \sin\theta$$

# Matrix Representations

## Rotation about z-axis

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$P' = R_z(\theta) \cdot P$$

## Rotation about x-axis

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$P' = R_x(\theta) \cdot P$$

## Rotation about y-axis

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$P' = R_y(\theta) \cdot P$$



# Example

- Let's manually rotate the point (2,0,0) 45 degrees clockwise about the z axis

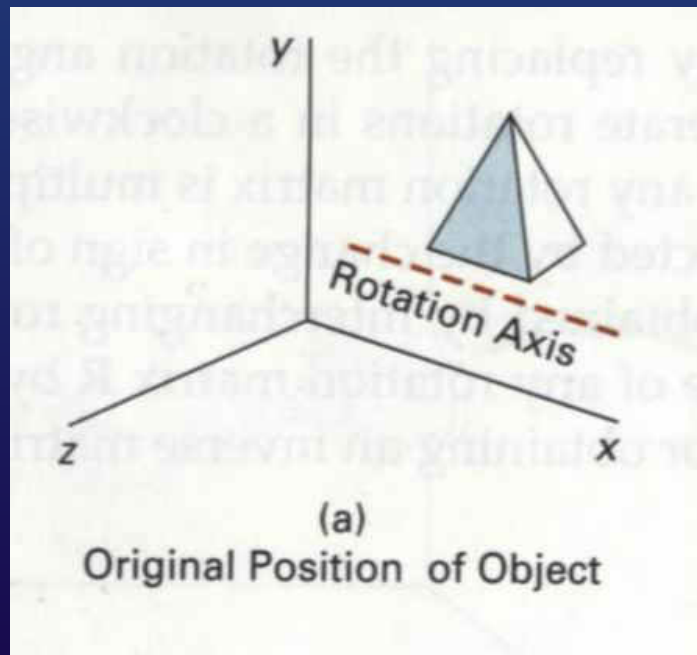
$$\mathbf{v}' = \mathbf{R}_z(45)\mathbf{v}$$

$$\mathbf{v}' = \begin{bmatrix} 0.707 & 0.707 & 0 \\ -0.707 & 0.707 & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 2 \\ 0 \\ 0 \end{bmatrix}$$

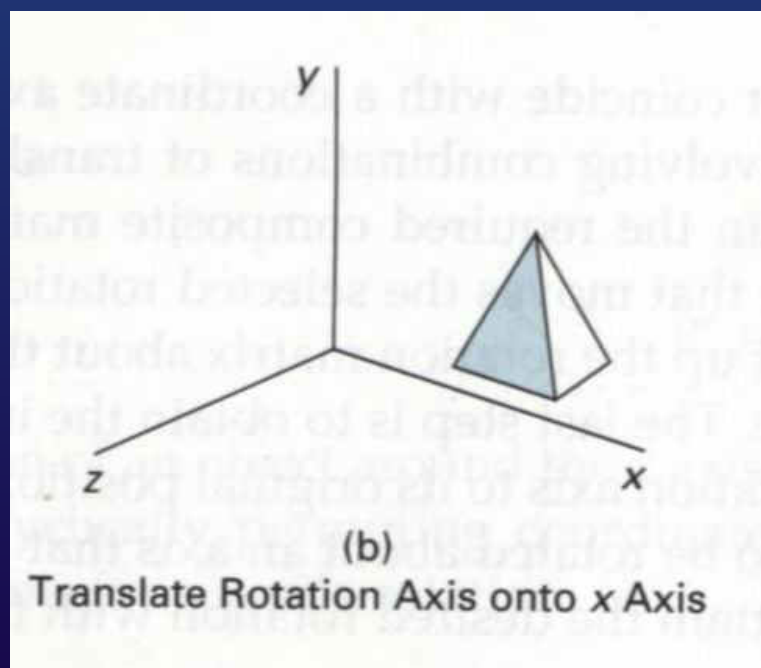
$$\mathbf{v}' = \begin{bmatrix} 2 \times 0.707 + 0 \times 0.707 + 0 \times 0 \\ 2 \times -0.707 + 0 \times 0.707 + 0 \times 0 \\ 2 \times 0 + 0 \times 0 + 0 \times 0 \end{bmatrix}$$

$$\mathbf{v}' = \begin{bmatrix} 1.414 \\ -1.414 \\ 0 \end{bmatrix}$$

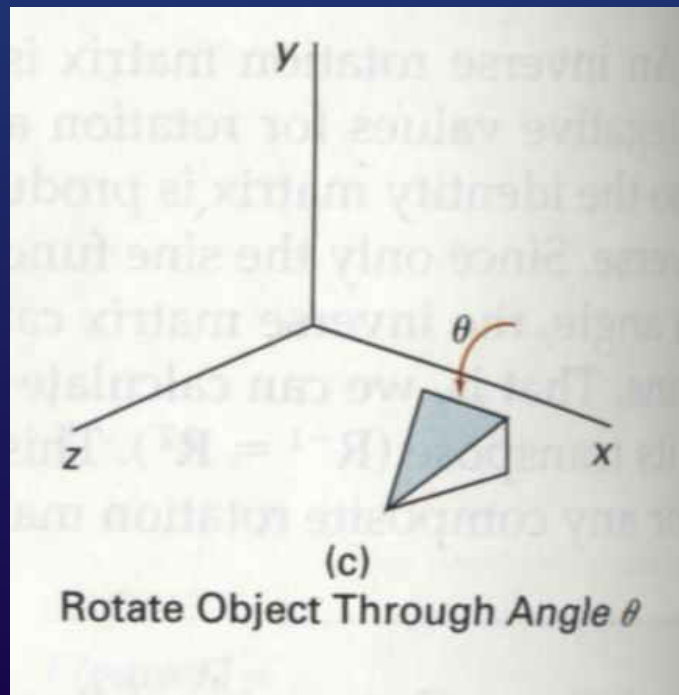
## Around arbitrary axis



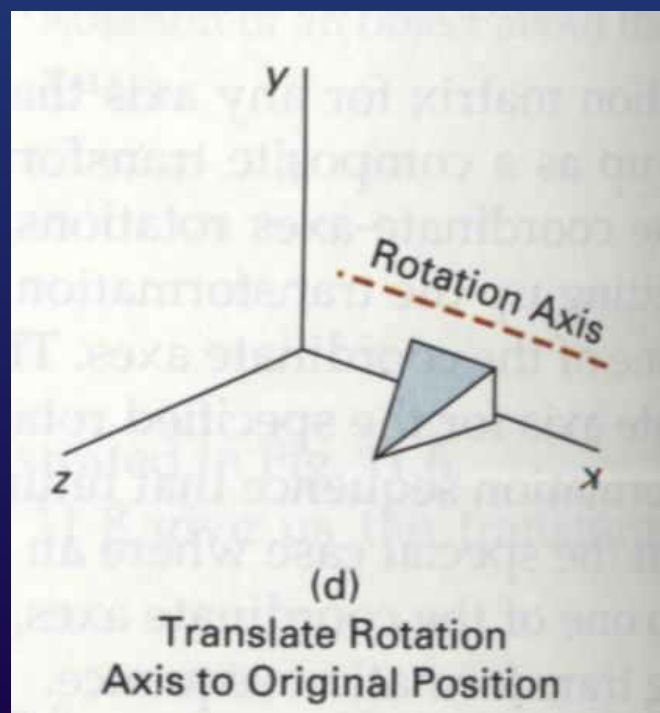
## Around arbitrary axis



## Around arbitrary axis



## Around arbitrary axis



# Composite Transformation

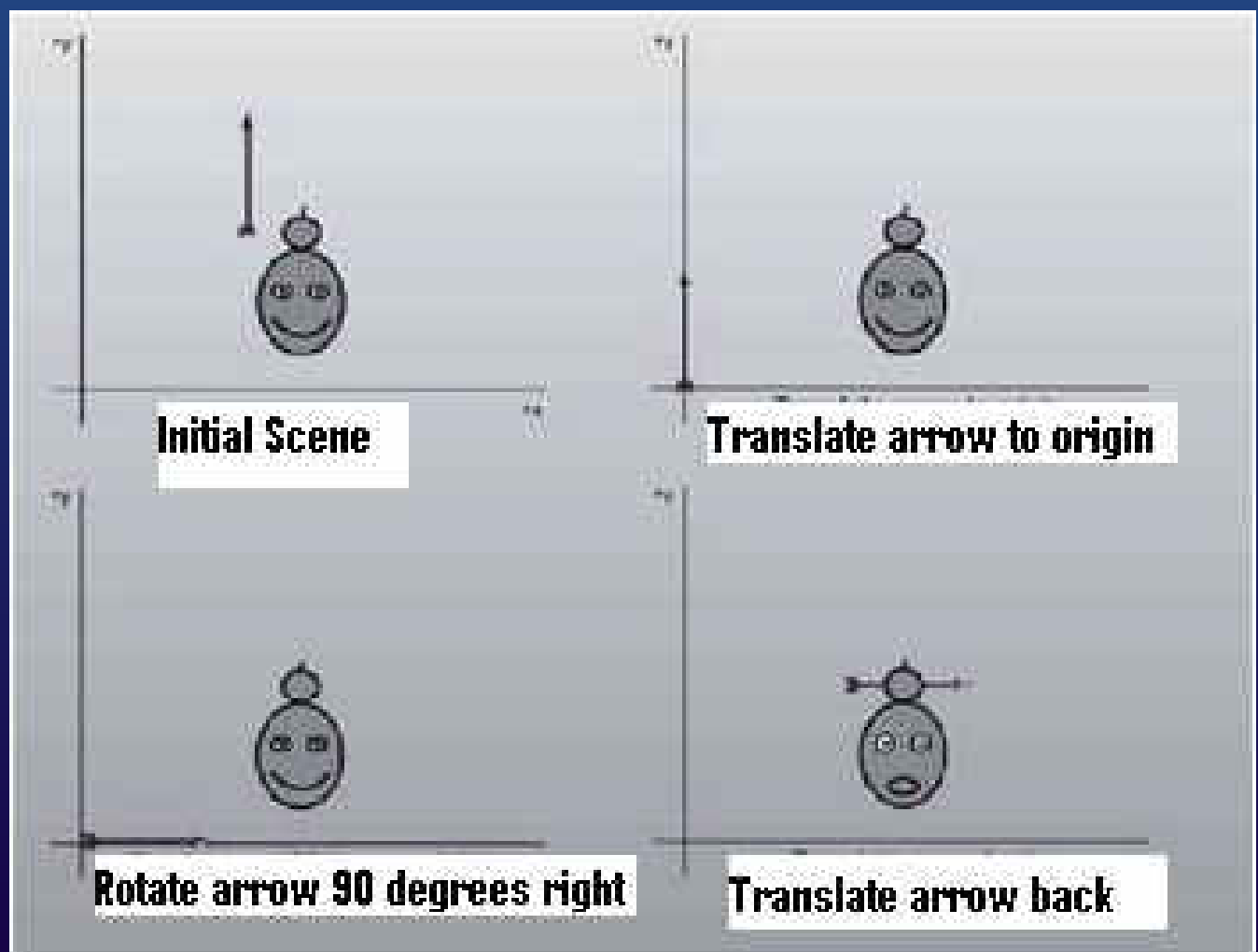
Now you can take an object and apply a sequence of transformations to it to make it do whatever you want. All you need to do is figure out the sequence of transformations needed and then apply the sequence to each of the points in the model.

As an example, let's say you want to rotate an object sitting at a certain point  $p$  around its  $z$  axis. You would perform the following sequence of transformations to achieve this

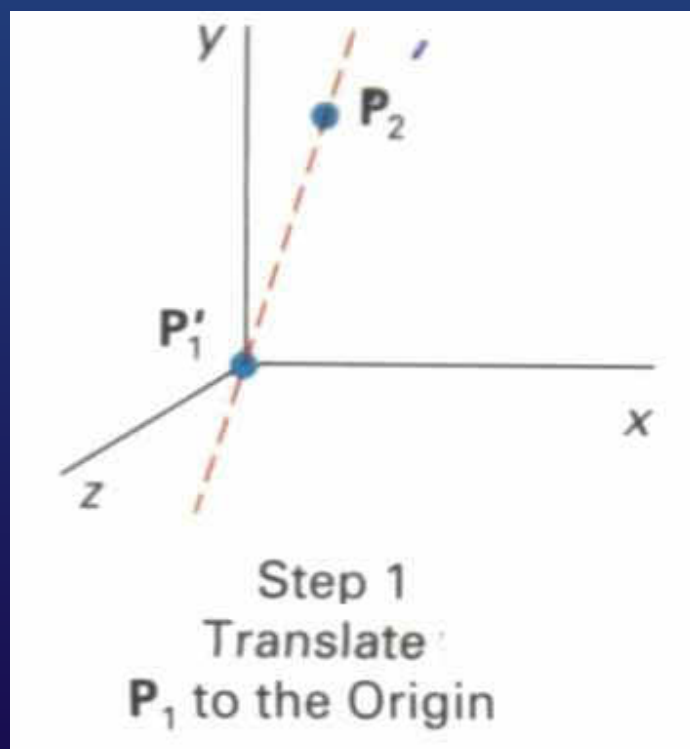
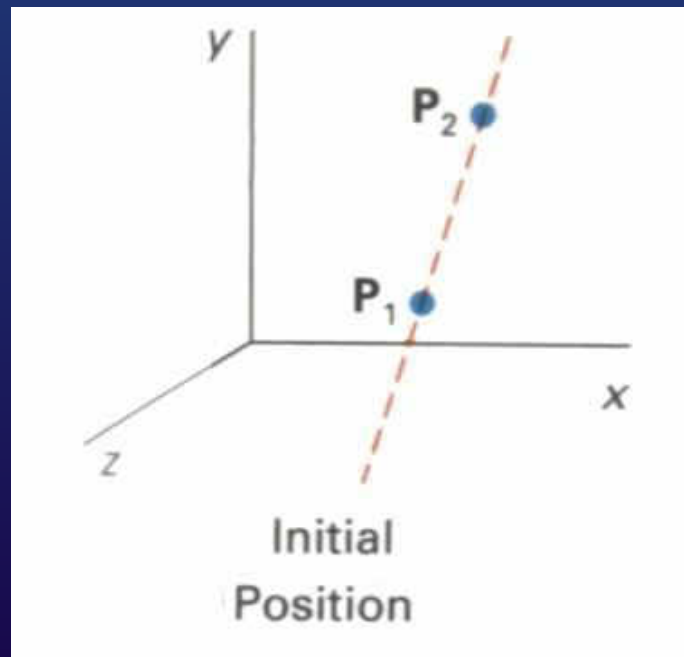
$$v = vT(-p)$$

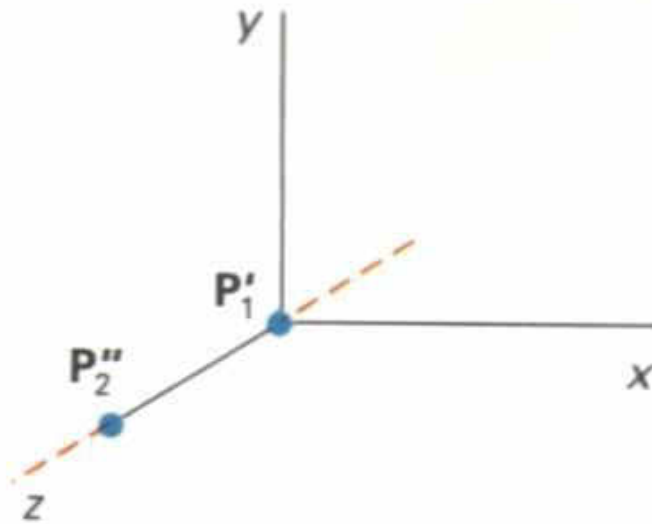
$$v = vR_z\left(\frac{\pi}{2}\right)$$

$$v = vT(p)$$

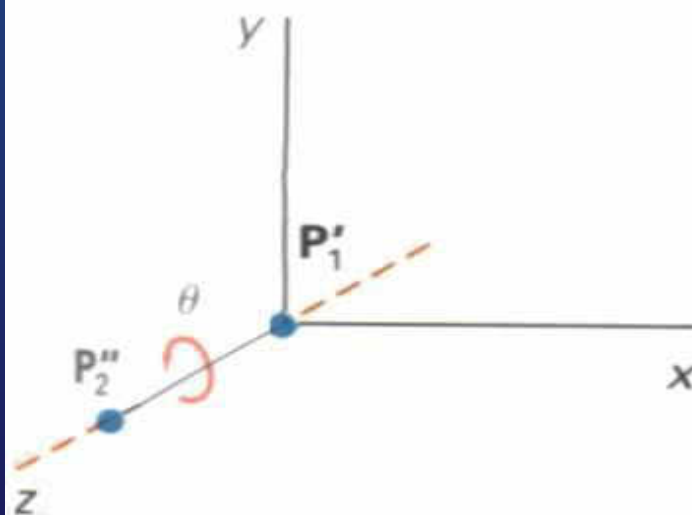


# Around arbitrary axis

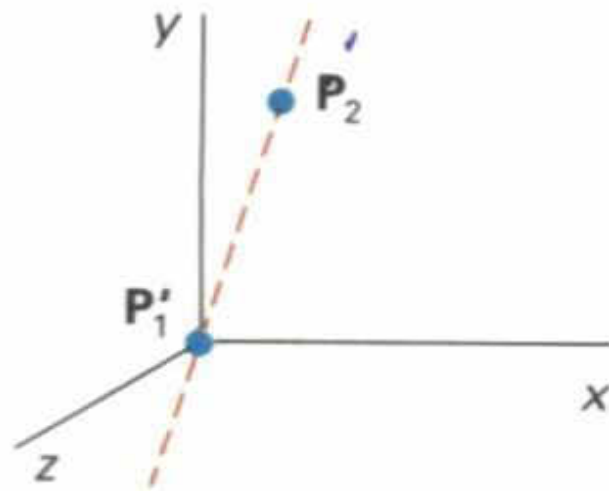




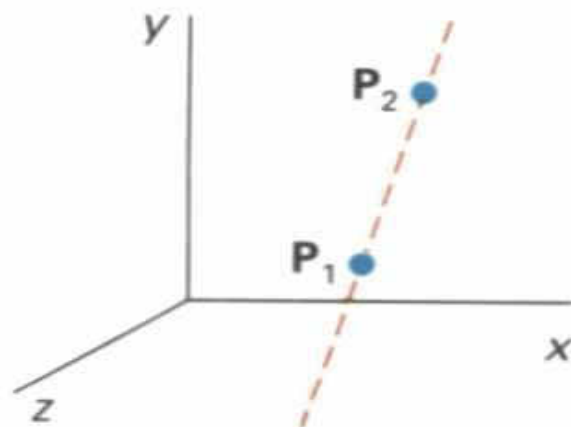
Step 2  
Rotate  $P'_2$   
onto the  $z$  Axis



Step 3  
Rotate the  
Object Around the  
 $z$  Axis



Step 4  
Rotate the Axis  
to the Original  
Orientation



Step 5  
Translate the  
Rotation Axis  
to the Original  
Position



# Scaling

Scaling an object changes its size and repositions the object relative to the coordinate origin. If the transformation parameters are not all equal, relative dimensions in the object are changed

- ◆ Coordinate transformations for scaling relative to the origin are

- ◆  $X' = X \cdot S_x$

- ◆  $Y' = Y \cdot S_y$

- ◆  $Z' = Z \cdot S_z$

- ◆ 1) Uniform Scaling
- ◆ 2) Differential Scaling

# Uniform Scaling

- ◆ We preserve the original shape of an object with a uniform scaling
- ◆ (  $S_x = S_y = S_z$  )

# Differential Scaling

- ◆ We do not preserve the original shape of an object with a differential scaling
- ◆ (  $S_x \neq S_y \neq S_z$  )

## Scaling w.r.t. Origin

$$\begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

## Scaling w.r.t. a fixed position

Scaling with respect to a selected fixed position (Xf,Yf,Zf) can be represented with the transformation sequence:

# Transformation sequence

1. Translate the fixed point to the origin.
2. Scale the object relative to the coordinate origin
3. Translate the fixed point back to its original position

## Composite Transformation

$$\begin{bmatrix} 1 & 0 & 0 & X_f \\ 0 & 1 & 0 & Y_f \\ 0 & 0 & 1 & Z_f \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & -X_f \\ 0 & 1 & 0 & -Y_f \\ 0 & 0 & 1 & -Z_f \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

# Composite Transformation

$$\begin{bmatrix} S_x & 0 & 0 & (1-S_x)X_f \\ 0 & S_y & 0 & (1-S_y)Y_f \\ 0 & 0 & S_z & (1-S_z)Z_f \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

## Reflection

A reflection can be performed relative to a selected reflection axis or with respect to a selected reflection plane.

In general, three-dimensional reflection matrices are set up similarly to those for two dimensions. Reflections relative to a given axis are equivalent to 180 degree rotations.

## Reflection of points relative to the X axis

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

## Reflection of points relative to the Y axis

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

## Reflection of points relative to the xy plane

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



# Shears

- ◆ Shearing transformations can be used to modify object shapes

- ♦ As an example of three-dimensional shearing, the following transformation produces a z-axis shear:

$$\begin{bmatrix} 1 & 0 & a & 0 \\ 0 & 1 & b & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

## Y-axis shear

$$\begin{bmatrix} 1 & a & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & c & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

## X-axis shear

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ b & 1 & 0 & 0 \\ c & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Computer Graphics  
Lecture 18

**Rotation w.r.t. y-axis:**

$$P' = R_y(\theta) \cdot P$$

- ◆ From the drawing and elementary trigonometry, we can say:

$$\cos(q) = X/D \text{ or}$$

$$D = X / \cos(q)$$

$$\sin(q) = Y/D \text{ or}$$

$$D = Y / \sin(q)$$

where  $D$  = hypotenuse

= length of vector  
from origin to  $(x,y)$

To rotate to our new point, we just add  $m$  to  $q$

For our new point, we know that:

$$\cos (q + m) = X' / D$$

$$\sin (q + m) = Y' / D$$

$$X' = D * \cos (q + m)$$

$$Y' = D * \sin (q + m)$$

- ◆ We can expand these using the addition formulas

$$\cos ( q + m ) = \cos ( q ) * \cos ( m ) - \sin ( q ) * \sin ( m )$$

$$X' = D * \cos (q + m)$$

$$= D * [\cos (q) * \cos (m) - \sin (q) * \sin (m)]$$

$$= D * \cos (q) * \cos (m) - D * \sin (q) * \sin (m)$$

- ◆ But we noted that
- ◆  $D = X / \cos(q)$
- ◆  $D = Y / \sin(q)$
- ◆ so we can substitute them in :

- ◆  $X' = [X / \cos(q)] * \cos(q) * \cos(m) - [Y / \sin(q)] * \sin(q) * \sin(m)$
- ◆  $= X * \cos(m) - Y * \sin(m)$

- ◆ And we can follow a similar approach to find  $Y'$



- ◆  $Y' = D \sin(q+m)$   
 $= D [\sin(q) \cos(m) + \cos(q) \sin(m)]$   
 $= D \sin(q) \cos(m) + D \cos(q) \sin(m)$

- ◆  $Y' = [X / \cos(m)] \sin(q) \cos(m) + [Y / \sin(m)] \cos(q) \sin(m)$
- ◆  $= X \sin(q) + Y \cos(q)$

- ◆ So to rotate the point  $(x,y,z)$  around the  $Z$  axis through an angle of  $M$  degrees, we use:
- ◆  $X' = x \cos(M) - y \sin(M)$
- ◆  $Y' = x \sin(M) + y \cos(M)$
- ◆  $Z' = z$

- ◆ You see, in the above, we rotate  $M$  degrees around the  $Z$  axis (“roll”), moving from the  $+X$  axis to the  $+Y$  axis
- ◆ But if you look down on the  $XY$  plane from the  $+Z$  axis, this is a  $-M$  degree rotation!

- ◆ you can go through a similar process to calculate the rotation formula for rotating around the X axis (“pitch”) and around the Y axis (“yaw”), ending up with:

- ◆ Roll ( rotate around Z axis):
- ◆  $x' = x * \cos(m) + y * \sin(m)$
- ◆  $y' = y * \cos(m) - x * \sin(m)$
- ◆  $z' = z$

- ◆ Pitch ( rotate around the X axis):
- ◆  $x' = x$
- ◆  $y' = y * \cos(m) + z * \sin(m)$
- ◆  $z' = z * \cos(m) - y * \sin(m)$

- ◆ Yaw ( rotate about the Y axis):
- ◆  $x' = x * \cos(m) - z * \sin(m)$
- ◆  $y' = y;$
- ◆  $z' = x * \sin(m) + z * \cos(m)$

# Using Matrices for Rotation

## Roll (rotate about the Z axis)

$$\begin{bmatrix} \cos(m) & \sin(m) & 0 & 0 \\ -\sin(m) & \cos(m) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

or

$$P' = R_z(m) \cdot P$$

## Pitch (rotate about the X axis)

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(m) & \sin(m) & 0 \\ 0 & -\sin(m) & \cos(m) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

or

$$P' = R_x(m) \cdot P$$

## Yaw (rotate about the Y axis)

$$\begin{bmatrix} \cos(m) & 0 & -\sin(m) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(m) & 0 & \cos(m) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

or

$$P' = R_y(m) \cdot P$$

# Computer Graphics

## Lecture 19

### PROJECTIONS I

Taqdees A. Siddiqi  
cs602@vu.edu.pk

### PROJECTIONS I

Taqdees A. Siddiqi  
cs602@vu.edu.pk

# Reflection

A reflection can be performed relative to a selected reflection axis or with respect to a selected reflection plane.

In general, three-dimensional reflection matrices are set up similarly to those for two dimensions. Reflections relative to a given axis are equivalent to 180 degree rotations.



## Reflection of points relative to the X axis

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

## Reflection of points relative to the Y axis

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

# Reflection of points relative to the xy plane

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

- ◆ Shearing transformations can be used to modify object shapes

- ♦ As an example of three-dimensional shearing, the following transformation produces a z-axis shear:

$$\begin{bmatrix} 1 & 0 & a & 0 \\ 0 & 1 & b & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

## Y-axis shear

$$\begin{bmatrix} 1 & a & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & c & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

## X-axis shear

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ b & 1 & 0 & 0 \\ c & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

How can we display 3D point on 2D Screen?

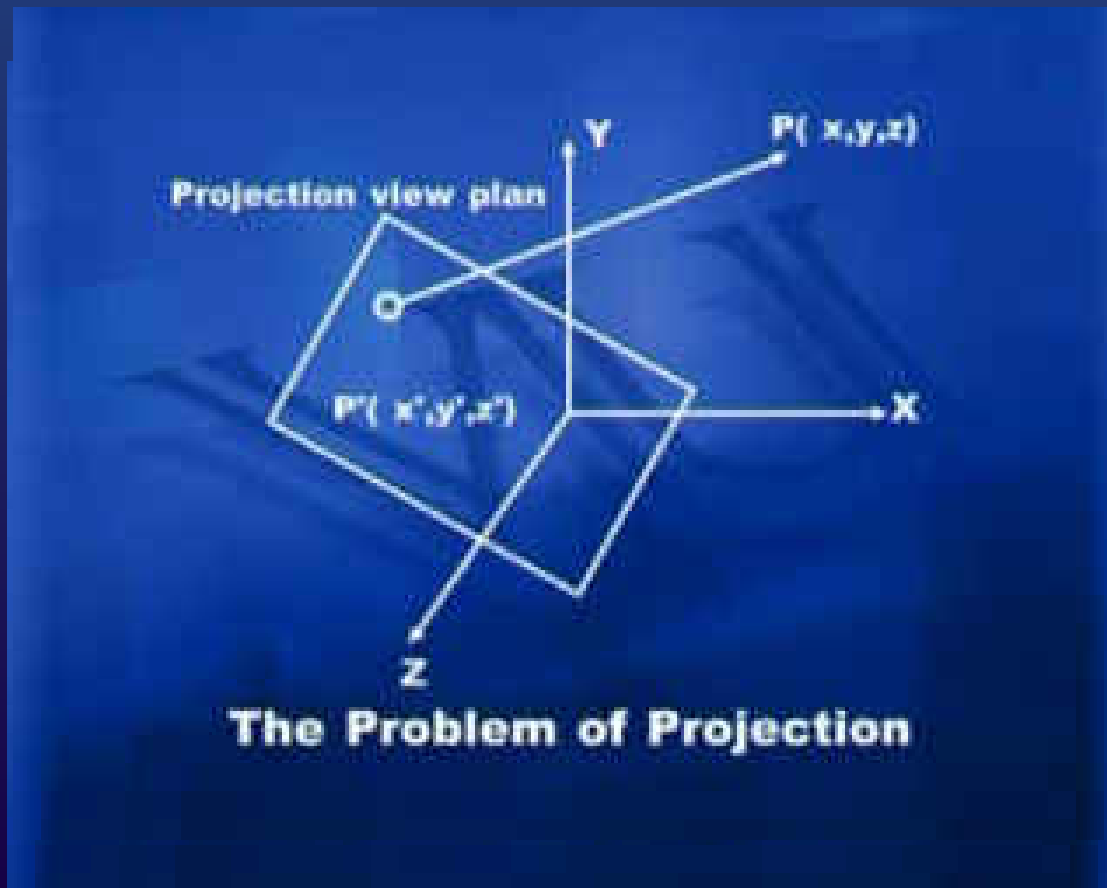
# PROJECTION

Projection can be defined as a mapping of points  $p(x,y,z)$  onto its image  $p'(x',y',z')$  in the projection plane or view plane, which constitutes the display surface

## What is mapping?

The mapping is determined by a projection line called the projector that passes through  $P$  and intersects the view plane

# What is mapping?

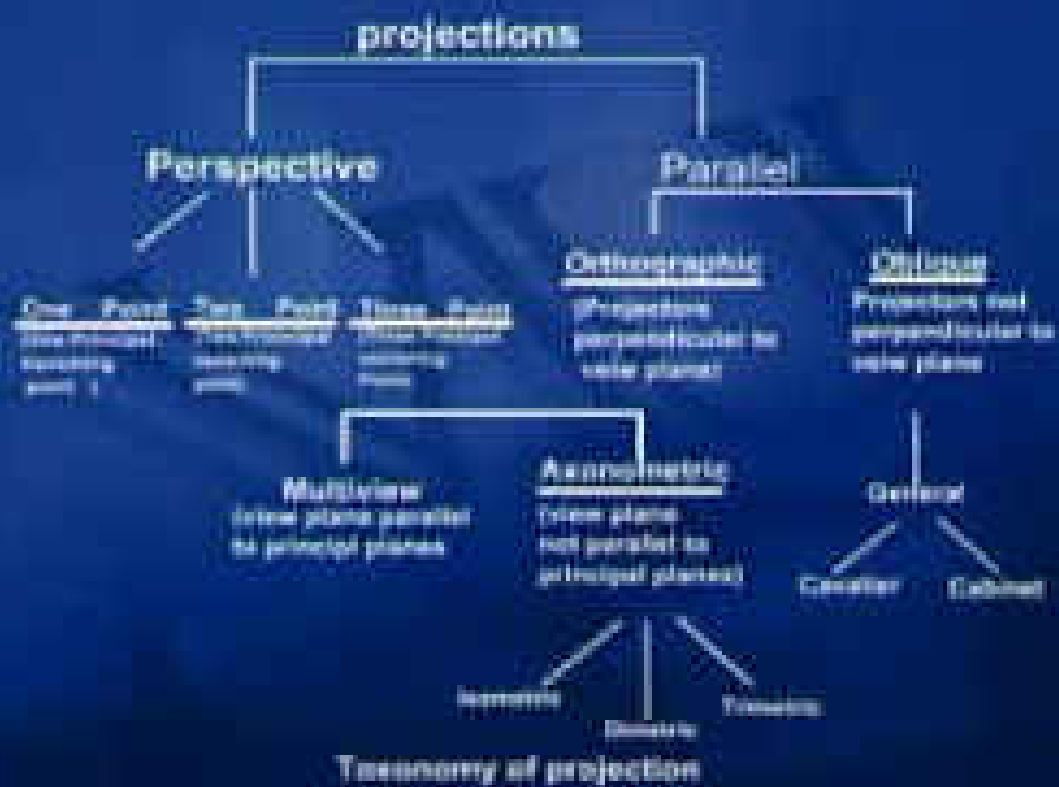


## Methods of Projection

1. Parallel Projection
2. Perspective Projection

These methods are used to solve the basic problems of pictorial representations

- ◆ We can construct different projections according to the view that is desired



## Parallel Projection

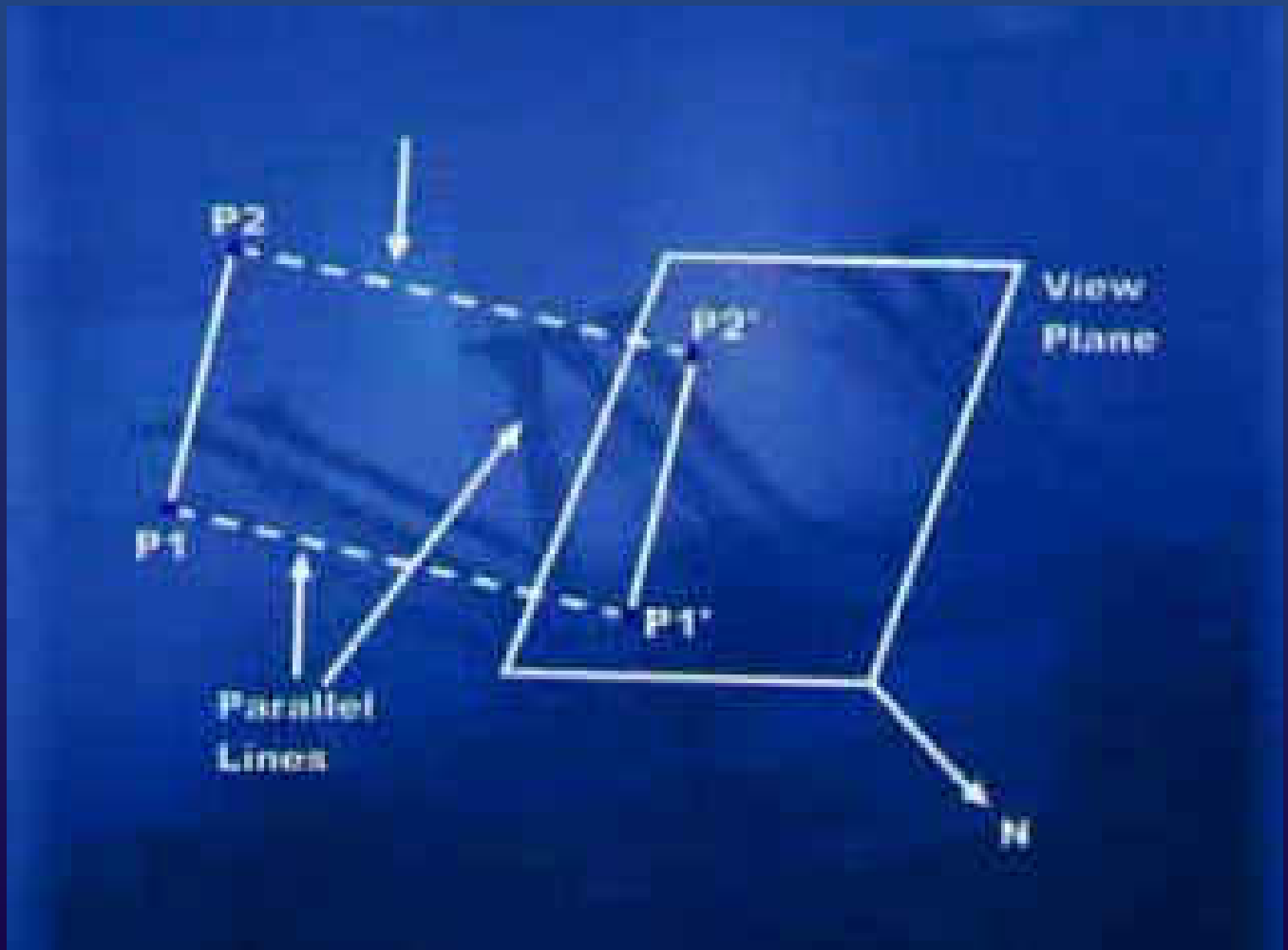


In parallel projection, image points are found as the intersection of the view plane with a projector drawn from the object point and having fixed direction. The direction of projection is the prescribed direction for all projections

Parallel projection methods are used by drafters and engineers to create working drawings of an object which preserve its scale and shape

# Mathematical Description of a Parallel Projection

Projection rays (projectors) emanate from a point called Center of Projection (COP) and intersect Projection Plane (PP). The COP for parallel projectors is at infinity. The length of a line on the projection plane is the same as the “true Length”



These are two different types of parallel projections

1. Orthographic
2. Oblique

If the direction of projection is (at  $90^{\circ}$ ) perpendicular to the projection plane then it is an orthographic projection.

# Projection plane

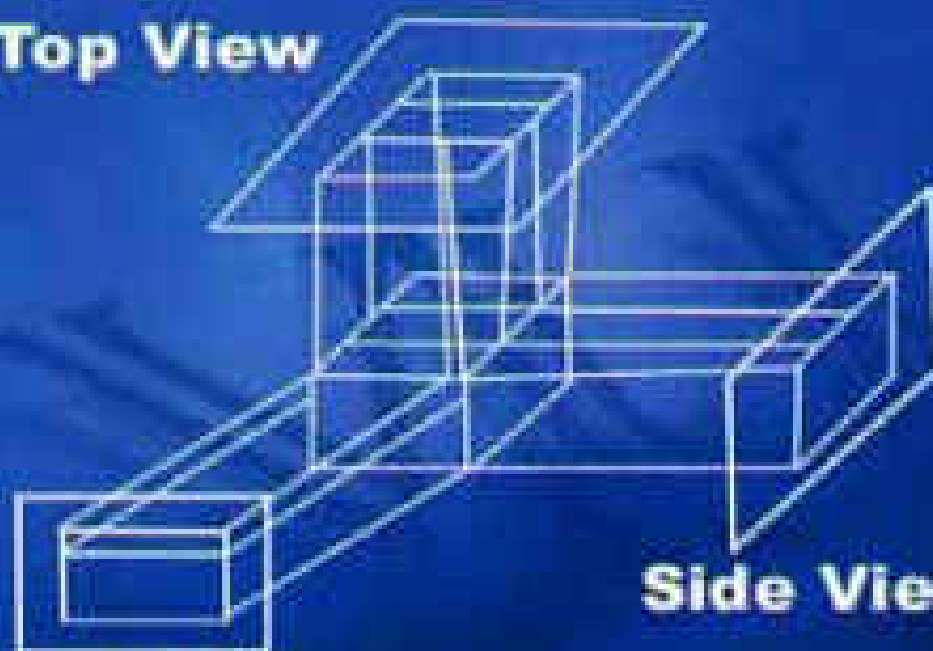


Orthographic Projection

## Axonometric Orthographic Projection

Orthographic projections that show more than one side of an object are called axonometric orthographic projections.

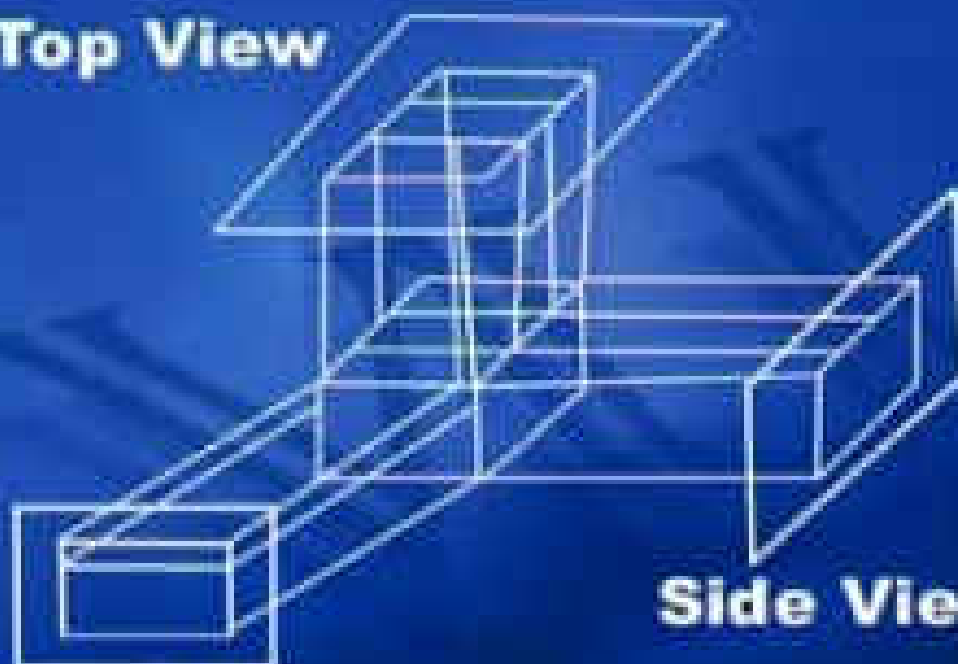
**Top View**



**Side View**

**Front View**

**Top View**



**Side View**

**Front View**

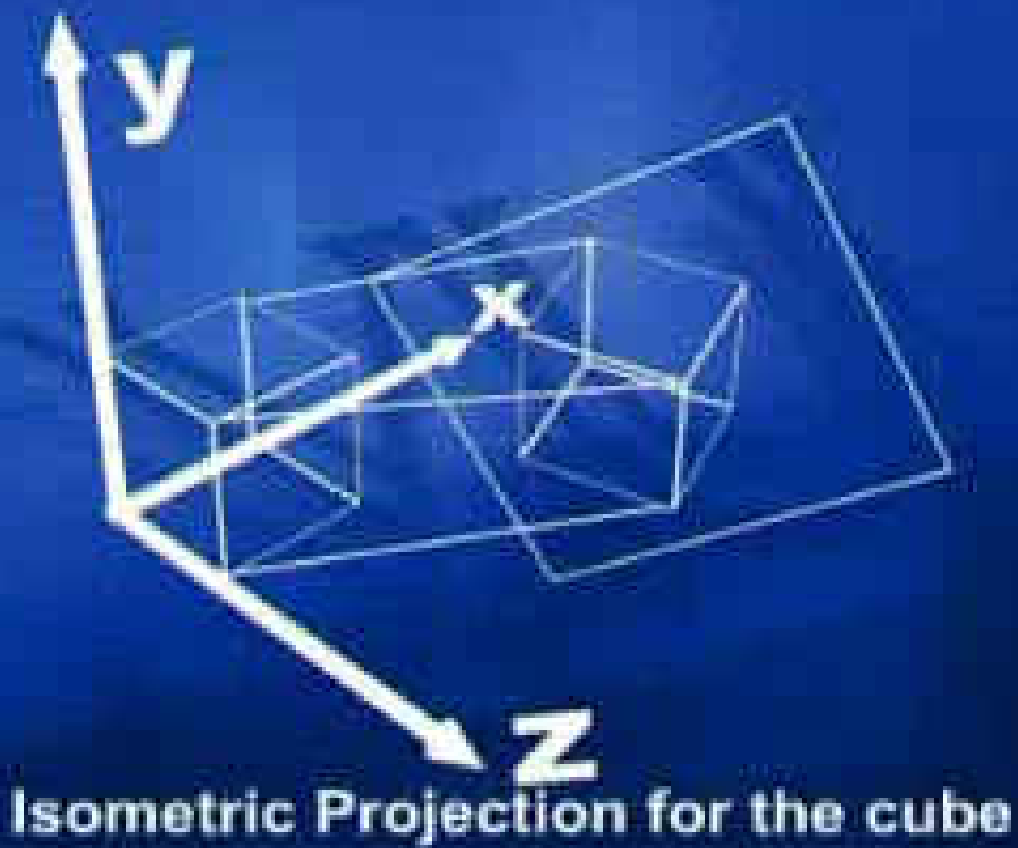
Here are three orthographic views of  
an object.

# There are three axonometric projections

1. Isometric
2. Dimetric
3. Trimetric

## Isometric

The projection plane intersects each coordinate axis in the model coordinate system at an equal distance or the direction of projection makes equal angles with all of the three principal axes



## Dimetric .2

The direction of projection make equal angles with exactly two of the principal axis



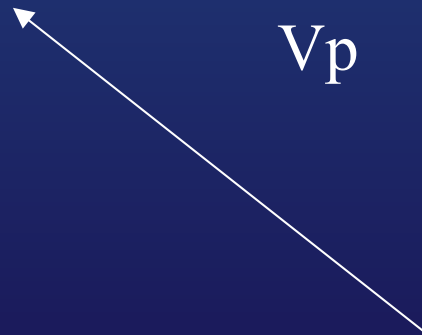
### **3. Trimetric**

The direction of projection makes unequal angles with the three principal axes

### **Oblique Projection**

If the direction of projection is not perpendicular to the projection plane then it is an oblique projection

# Projection Plane



Oblique Projection

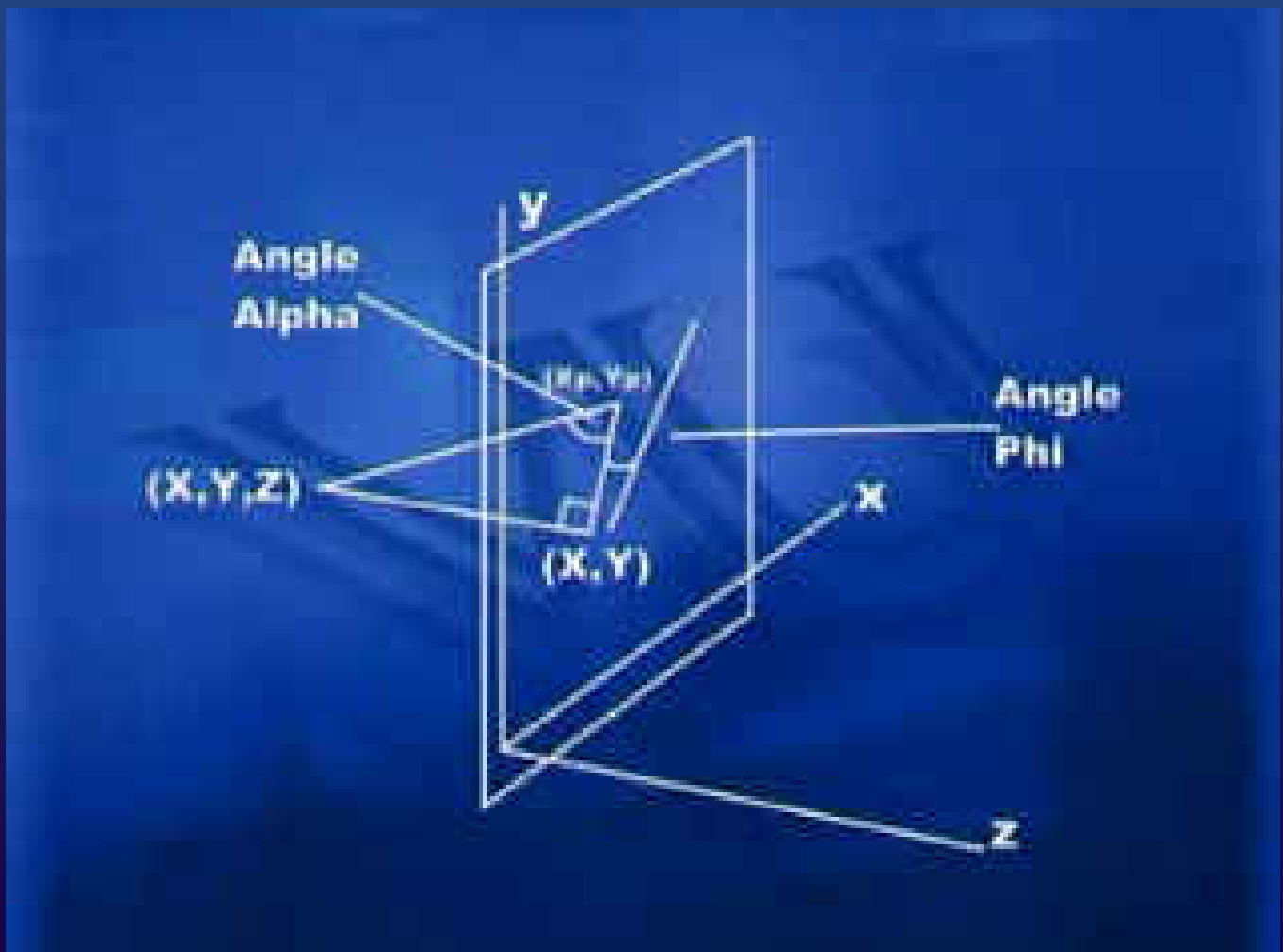
The projectors are not perpendicular to the projection plane but are parallel from the object to the projection plane

# Transformation equations for an orthographic prallel projection

- ◆ If the view plane is placed at position  $Z_{vp}$  along the  $Z$  axis. Then any point  $(x,y,z)$  in viewing coordinates is transformed to projection coordinates as :
- ◆  $X_p = x$
- ◆  $Y_p = y$

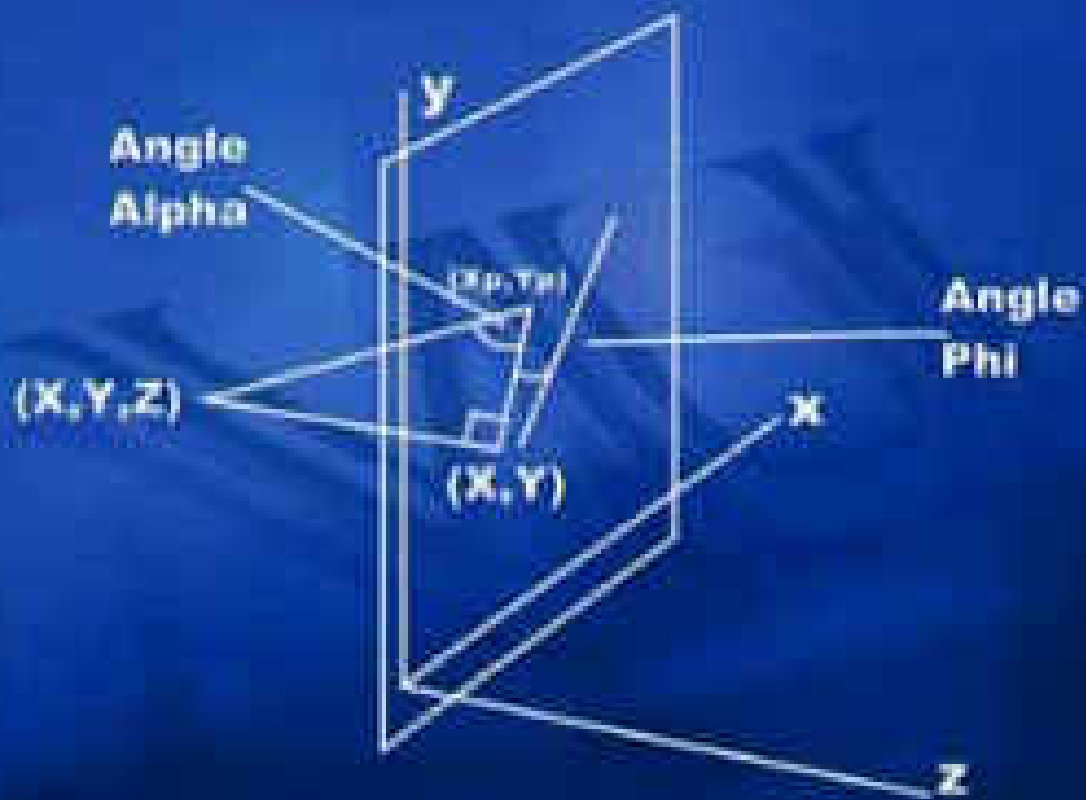
- ◆ Where the original Z-coordinate value is preserved for the depth information needed in depth cueing and visible surface determination procedures.

- ◆ Point  $(x, y, p)$  is projected to position  $(X_p, Y_p)$  on the view plane
- ◆ Orthographic projection coordinates on the plane are  $(x, y)$



$\alpha$

- ◆ The oblique projection line from  $(x, y, z)$  to  $(X_p, Y_p)$  makes an angle ' $\alpha$ ' with the line on the projection plane that joins  $(X_p, Y_p)$  and  $(x, y)$ .
- ◆ This line, of length  $L$ , is at an angle  $\phi$  with the horizontal direction in the projection plane



- ◆ We can express the projection coordinates in terms of  $x, y, L$ , and  $\phi$

- ◆  $\cos(\phi) = (X_p - x) / L$
- ◆  $\sin(\phi) = (Y_p - y) / L$

- ◆  $X_p = x + L \cos(\phi)$
- ◆  $Y_p = y + L \sin(\phi)$

- ◆ Length L depends on the angle ' $\alpha$ ' and the z coordinate of the point to be projected:

- ◆  $\tan(\alpha) = z / L$

thus,

$$L = z * 1 / \tan(\alpha)$$

- ◆  $L = z * L1$

- ◆ Where L1 is the inverse of  $\tan \alpha$ , which is also the value of L when  $z = 1$



- ◆ We can then write the oblique projection equations as:

- ◆  $X_p = x + z (L1 \cos (\phi) )$

- ◆  $Y_p = y + z (L1 \sin (\phi) )$

# The transformation matrix for parallel projection

- ◆ The transformation matrix for producing any parallel projection onto the xy plane can be written as

$$\textit{parallel } M = \begin{bmatrix} 1 & 0 & L1 \cos(\text{Phi}) & 0 \\ 0 & 1 & L1 \cos(\text{Phi}) & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- ◆ Now if  $\alpha = 90$  (projection line is perpendicular to Projection Plane) then
- ◆  $\tan(\alpha) = \text{infinity} \rightarrow L1 = 0$ , so we have an orthographic projection

# Two special cases of oblique projection:

1. Cavalier
2. Cabinet

## 1. Cavalier

- ◆  $\alpha = 45$
- ◆  $\tan(\alpha) = 1 \rightarrow L1 = 1$
- ◆ This is a Cavalier projection such that all lines perpendicular to the projection plane are projected with no change in length.

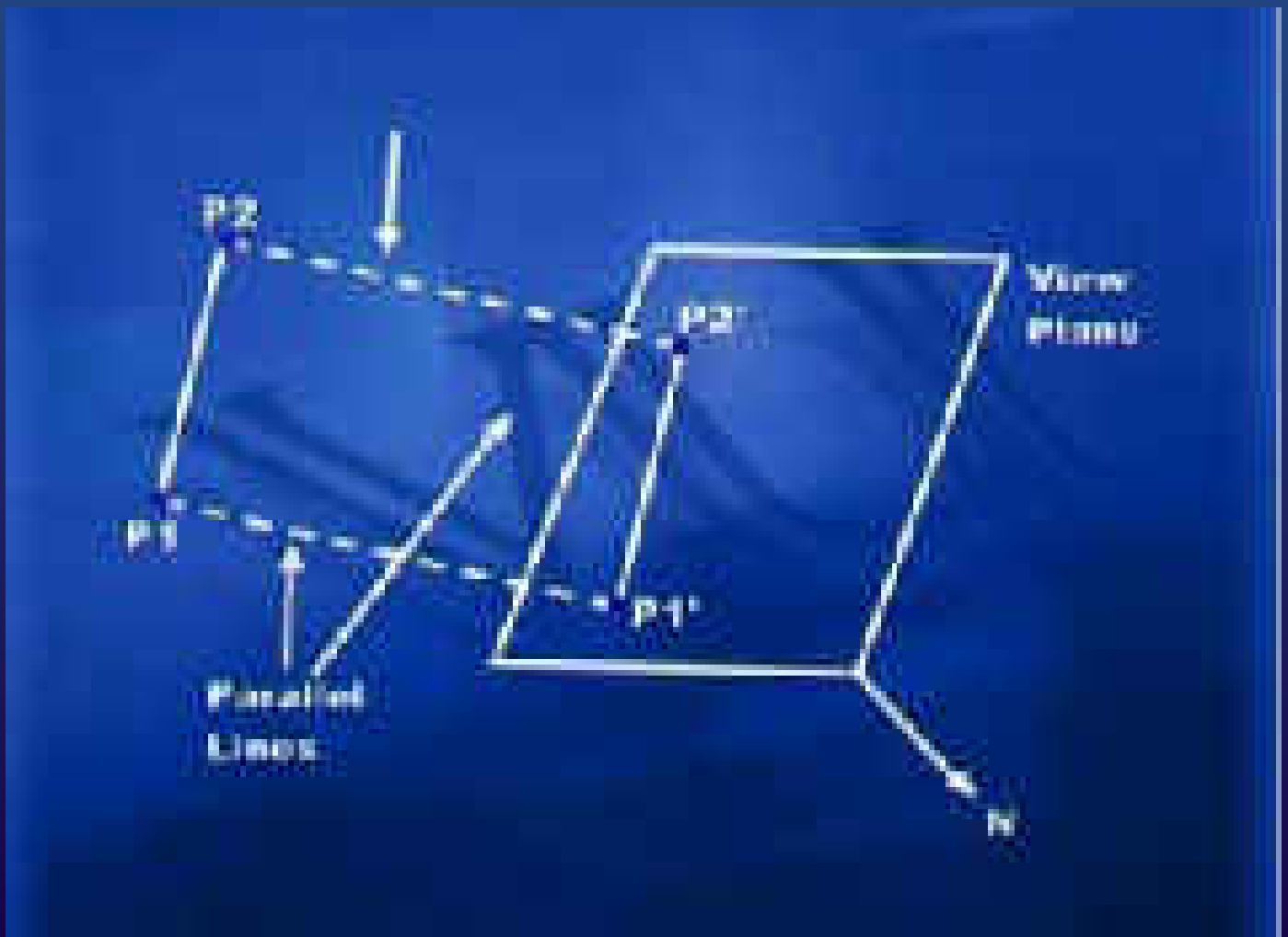
## $L_1 = L_2$ Two Cubs w Cavalier



- ◆  $\tan(\alpha) = 2$
- ◆  $\alpha = 63.40^\circ$ ,  $L_1 = \frac{1}{2}$
- ◆ Lines which are perpendicular to the projection plane are projected at  $\frac{1}{2}$  length. This is a Cabinet projection.

$$L_1 = \frac{1}{2} L_2$$





# Computer Graphics

## Lecture 20

# Computer Graphics

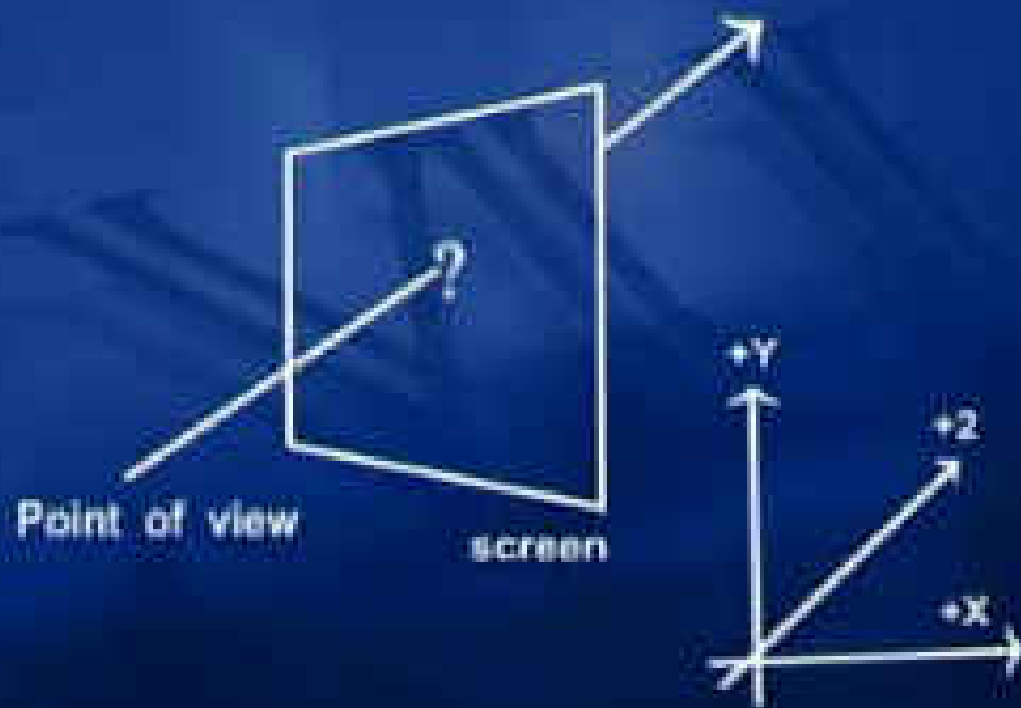
## Lecture 20 PROJECTIONS II

Taqdees A. Siddiqi  
cs602@vu.edu.pk

### Prospective Projection

How do you calculate the corresponding screen pixel?





This is a mathematical task as pictured. However , we can make it much simpler if we impose the following requirements:

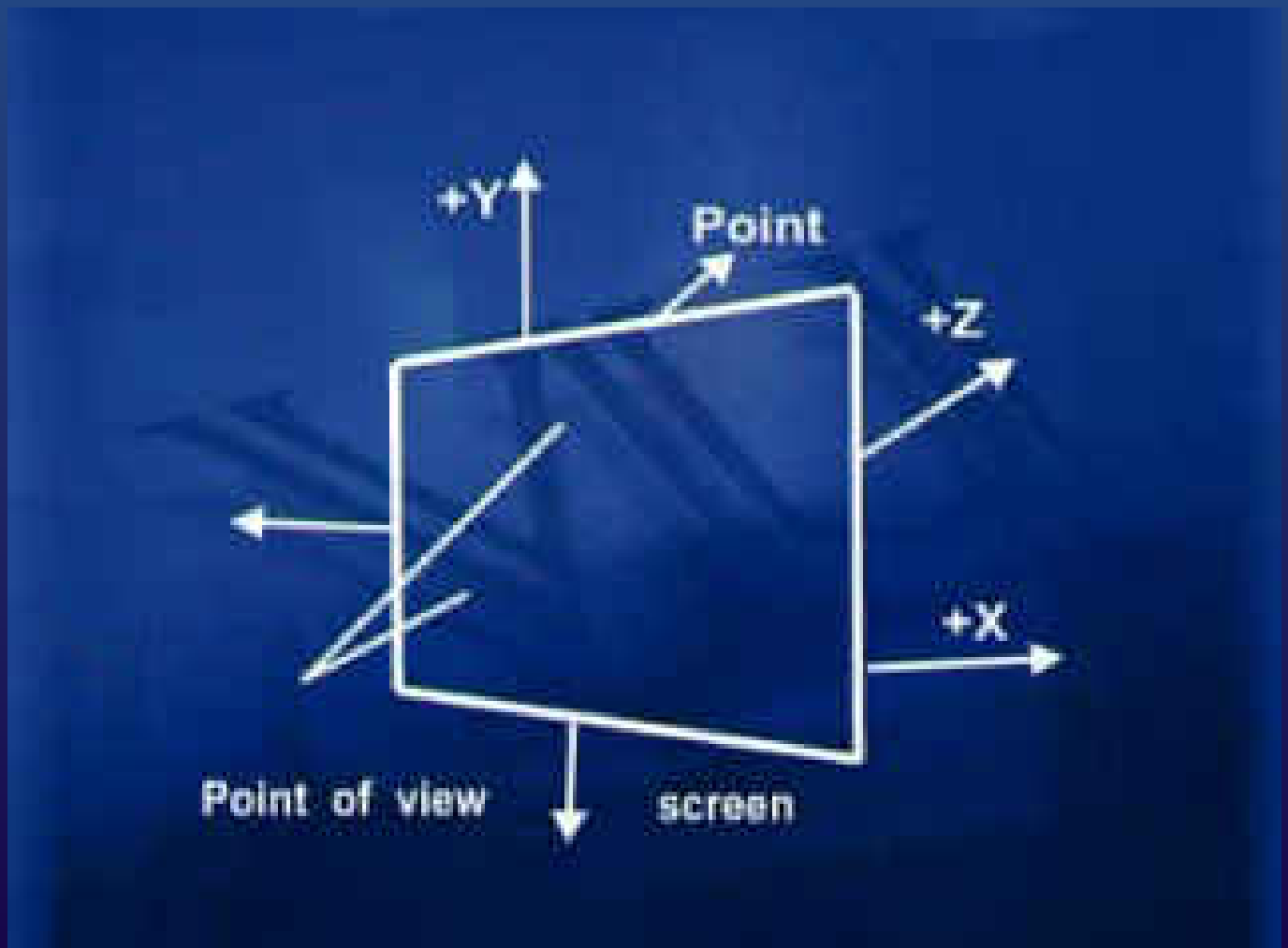
The point of View(POV) must lie on the Z axis, The screen plane must be parallel to the X,Y plane, with the left and right edges of the screen parallel to the Y-axis, And the top and bottom edges of the screen parallel to the X-axis

For our view to come out correctly, we will also want the Z axis to pass through the middle of the screen.

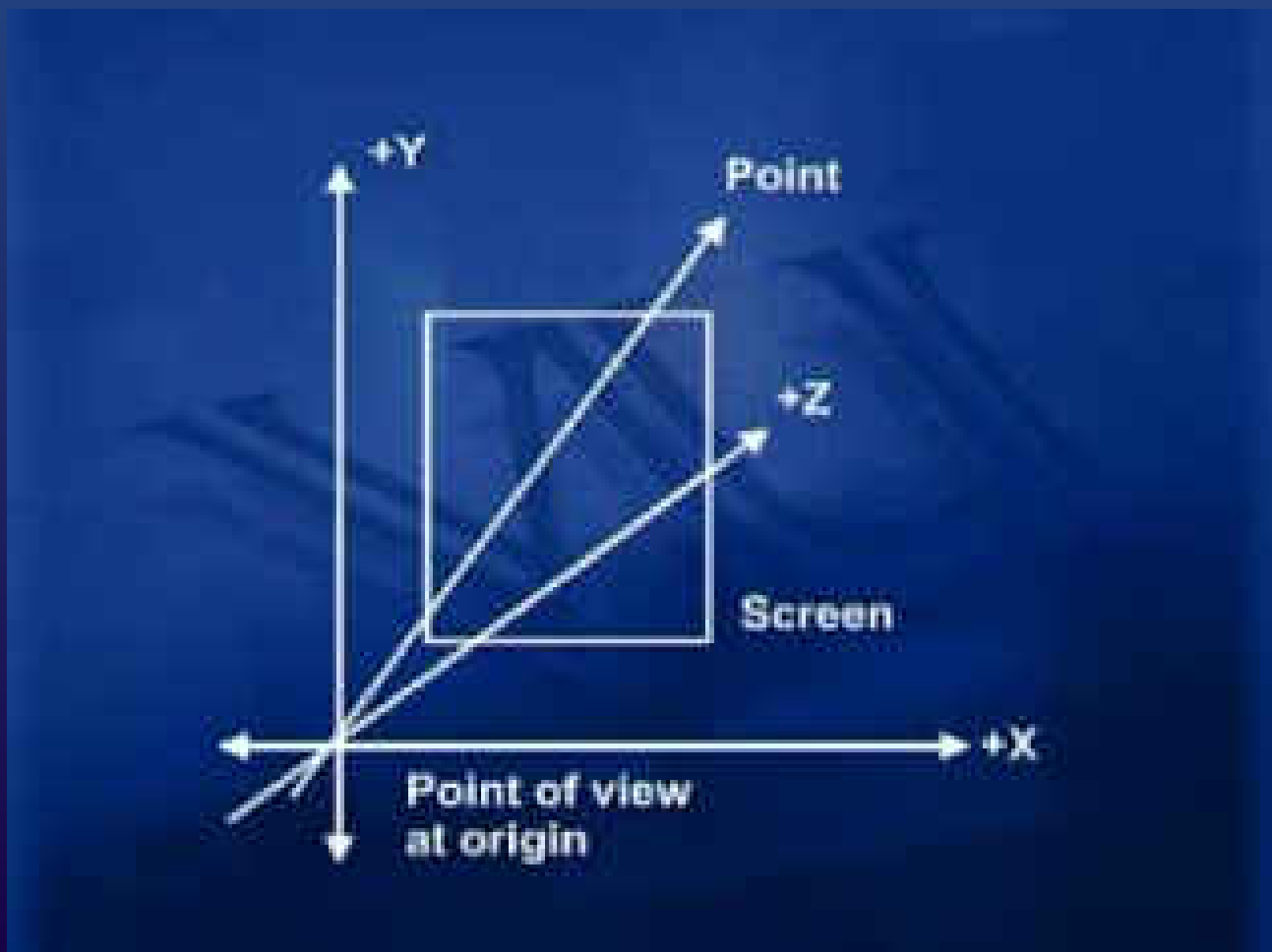
Why ?

Two common approaches are used with this:

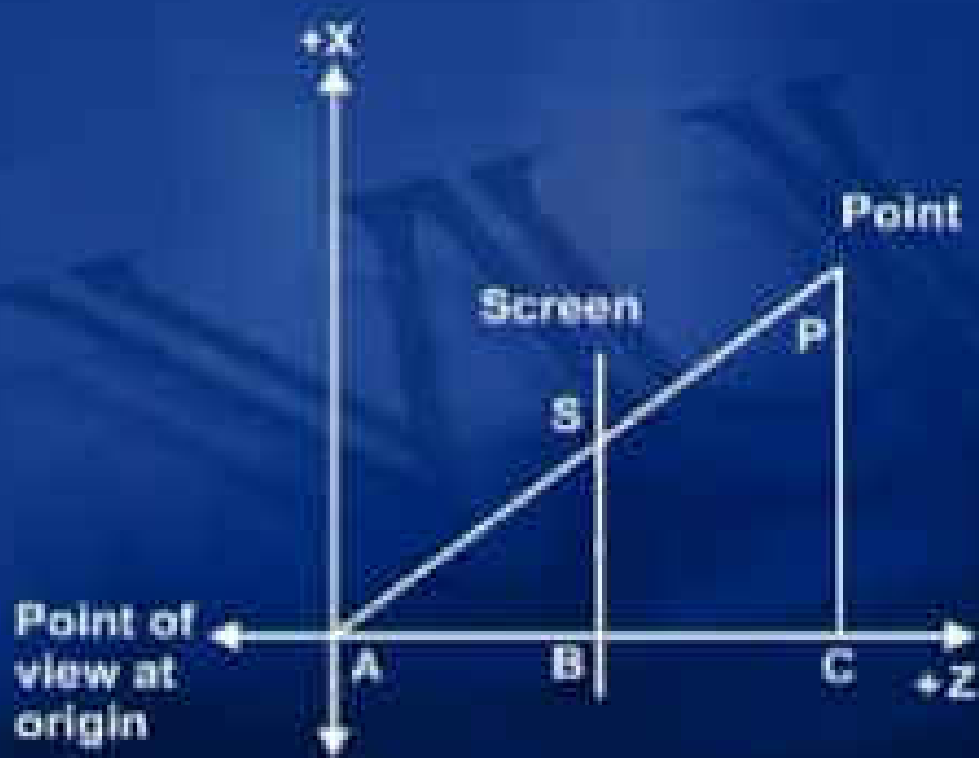
1. The first is where the POV is at same point  $(0,0,-z)$  and the screen lies on the X-Y plane, graphically, this looks like figure:



1. The second is where the POV lies at the origin, and the screen lies on a plane at some  $+Z$  coordinate, as shown in figure given below:



- ♦ Calculating the screen pixel that correlates to a 3D point is now a matter of simple geometry. From a viewpoint above the screen and POV (looking at the X-Z plane), the geometry appears like the one shown in this figure:



In geometric terms, we say that the triangle from A to B to S is similar to the triangle from A to C to P because the three angles that make up the triangles are the same; the angle from AB to AS is the same as the angle from AC to AP.

- ◆ The two right angles i.e.  $90^{\circ}$ , and therefore the remaining two angles are the same (the sum of the angles in a triangle is always 180 degrees).

What also holds true from similar triangles is that the ratio of two sides also holds between the similar triangles; this means that the ratio of BS to AB is the same as the ratio of CP to AC.

But we know what AB is –it is Screen.z! and

But we know what AB is –it is Screen.z! and  
we know what AC is – it is point.z! and

But we know what AB is –it is Screen.z! and  
we know what AC is – it is point.z! and  
we know what CP is – it is point.x!  
Therefore:



$$|BS| / |AB| = |CP| / |AC|$$

$$|BS| = |AB| * |CP| / |AC|$$

$$|BS| = \text{Screen.z} *$$

$$\text{point.x} / \text{point.z}$$

Note that :

- ◆ Screen.z is the distance d from the point of view at origin or the scaling factor.

- ◆ There is a final transformation that the points must go through in this transformation process.

This transformation maps 3D points defined with respect to the view origin (in view space) and turns them into 2D points that can be drawn on the display

- ◆ After transforming and clipping the polygons that make up the scene such that they are visible on the screen, the final step is to move them into 2D coordinates

- ◆ Since in order to actually draw things on the screen we need to have absolute x,y coordinates

- ◆ The way this used to be done was without matrices, just as an explicit projection calculation. The point  $(x,y,z)$  would be mapped to  $(x',y')$  using the following equations:

$$x' = scale \frac{x}{z} + xCenter$$

$$y' = height - (scale \frac{y}{z} + yCenter)$$

## The perspective Projection Matrix

- ◆ The aspect ratio of screen (in terms of width and height), is calculated as:

$$aspect = \frac{height}{width}$$

$$w = aspect \frac{\cos(fov)}{\sin(fov)}$$

$$h = \frac{\cos(fov)}{\sin(fov)}$$

$$q = \frac{Z_{far}}{Z_{far} - Z_{nert}}$$

- ◆ With these parameters, the following projection matrix can be made:

$$\begin{bmatrix} w & 0 & 0 & 0 \\ 0 & h & 0 & 0 \\ 0 & 0 & q & 1 \\ 0 & 0 & -q(Z_{near}) & 0 \end{bmatrix}$$

- ◆ Just for a sanity check, check out the result of this matrix multiplication:

$$\begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} w & 0 & 0 & 0 \\ 0 & h & 0 & 0 \\ 0 & 0 & q & 1 \\ 0 & 0 & -q(Z_{near}) & 0 \end{bmatrix} = \begin{bmatrix} wx & hy & qz - q(Z_{near})z \end{bmatrix}$$

- ◆ Since, after the multiplication, 'w' is set to z (which can be any value)
- ◆ All four components need to be divided by z to get normalized.
- ◆ This gives the following Cartesian coordinates:

$$\begin{bmatrix} \frac{wx}{z} & \frac{hy}{z} & q \left[ 1 - \frac{(Znear)}{Z} \right] & 1 \end{bmatrix}$$

- ◆ As you can see, this is exactly what was wanted. The width and height are still scaled by values as in the above equation and they are still divided by z



- ◆ We then multiply the world, view, and projection matrices together (in that order) to get a total matrix that transforms point from object space to screen space

$V_{world} = V_{local} M_{world}$

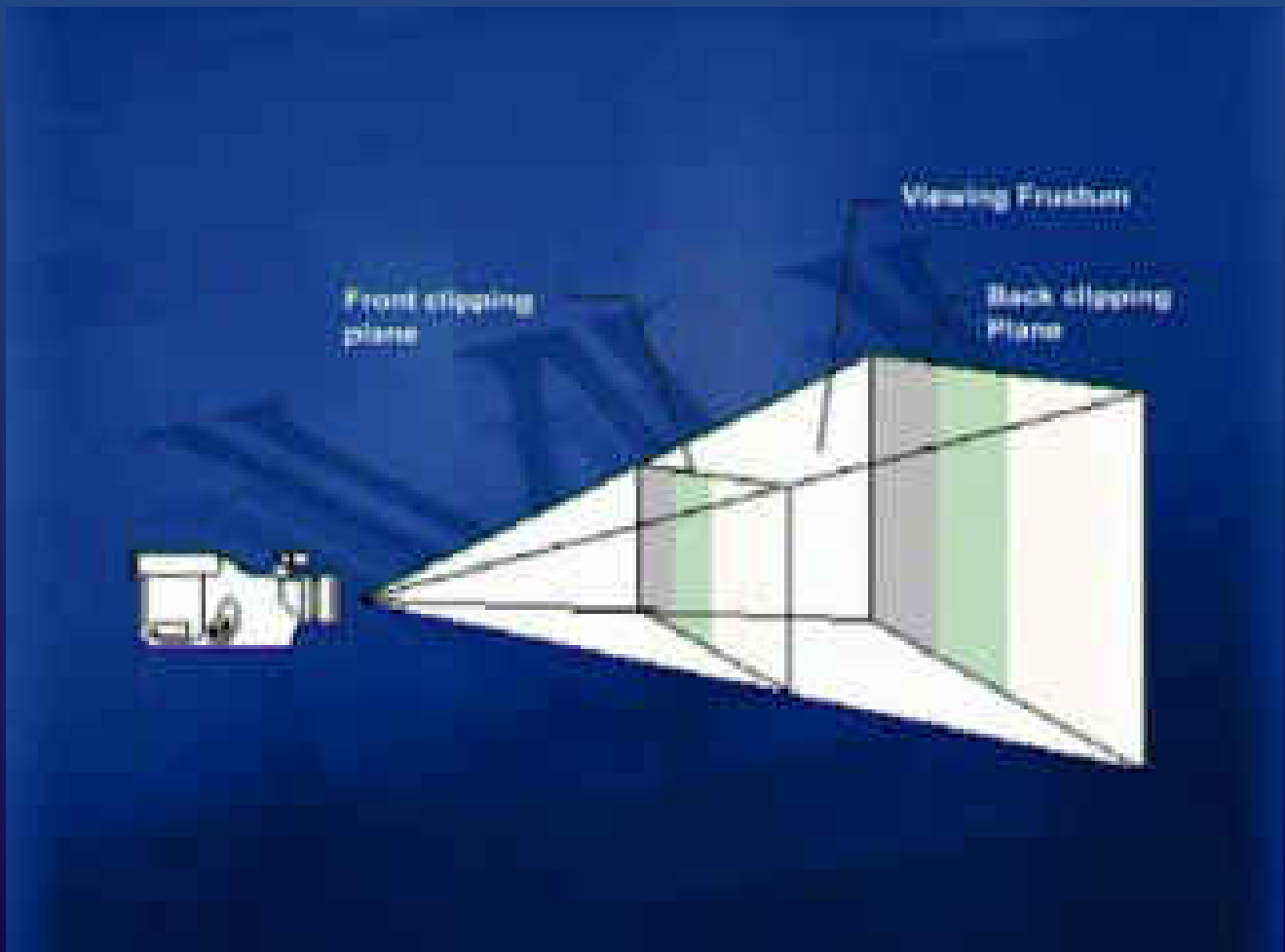
$V_{view} = V_{world} M_{view}$

$V_{screen} = V_{view} M_{projection}$

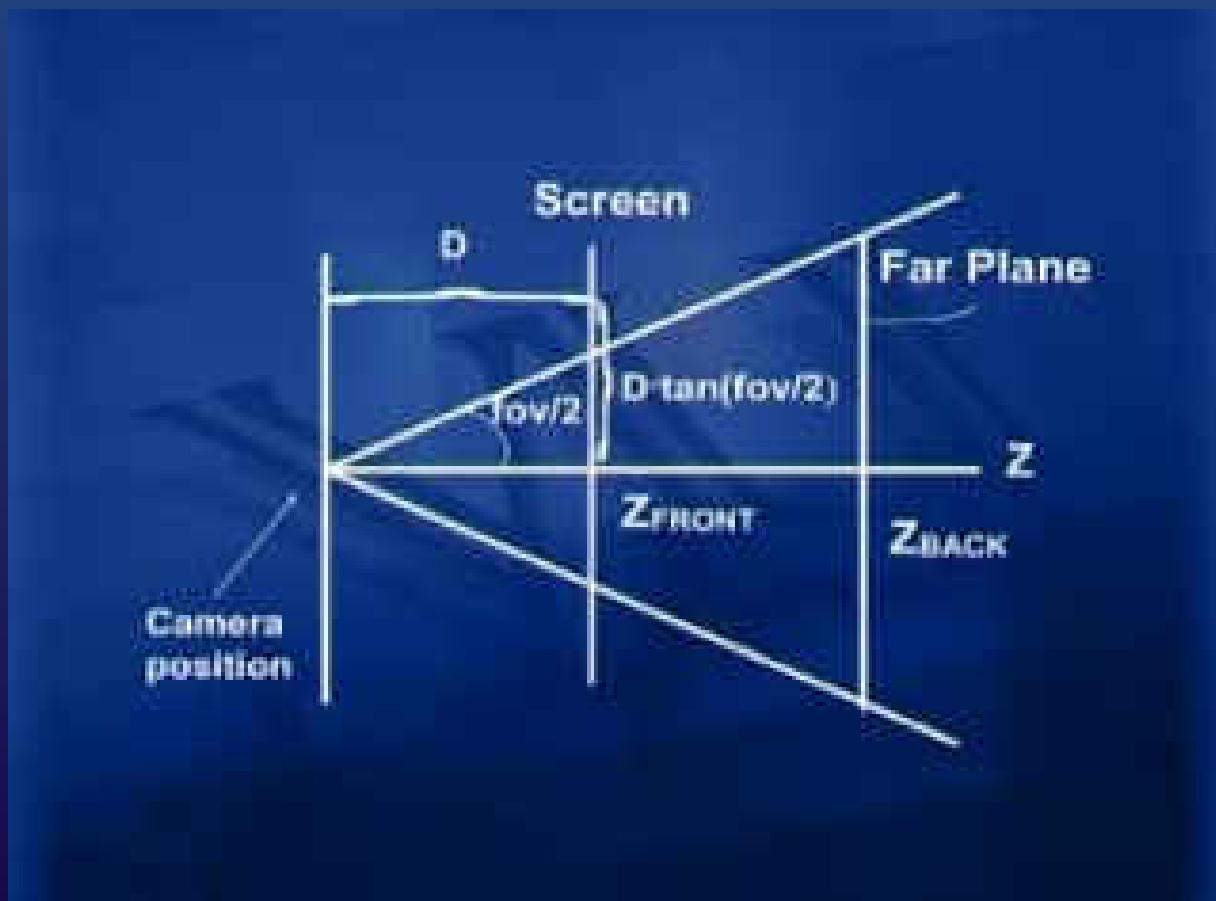
$V_{screen} = V_{local} (M_{world} M_{view} M_{projection})$

# The Viewing Frustum

- ◆ The volume within the pyramid between the front and back clipping planes is the viewing frustum. Objects are visible only when they are in this volume.



- ◆ The viewing frustum is defined by fov (field of view) front and back clipping planes, specified in z-coordinates.

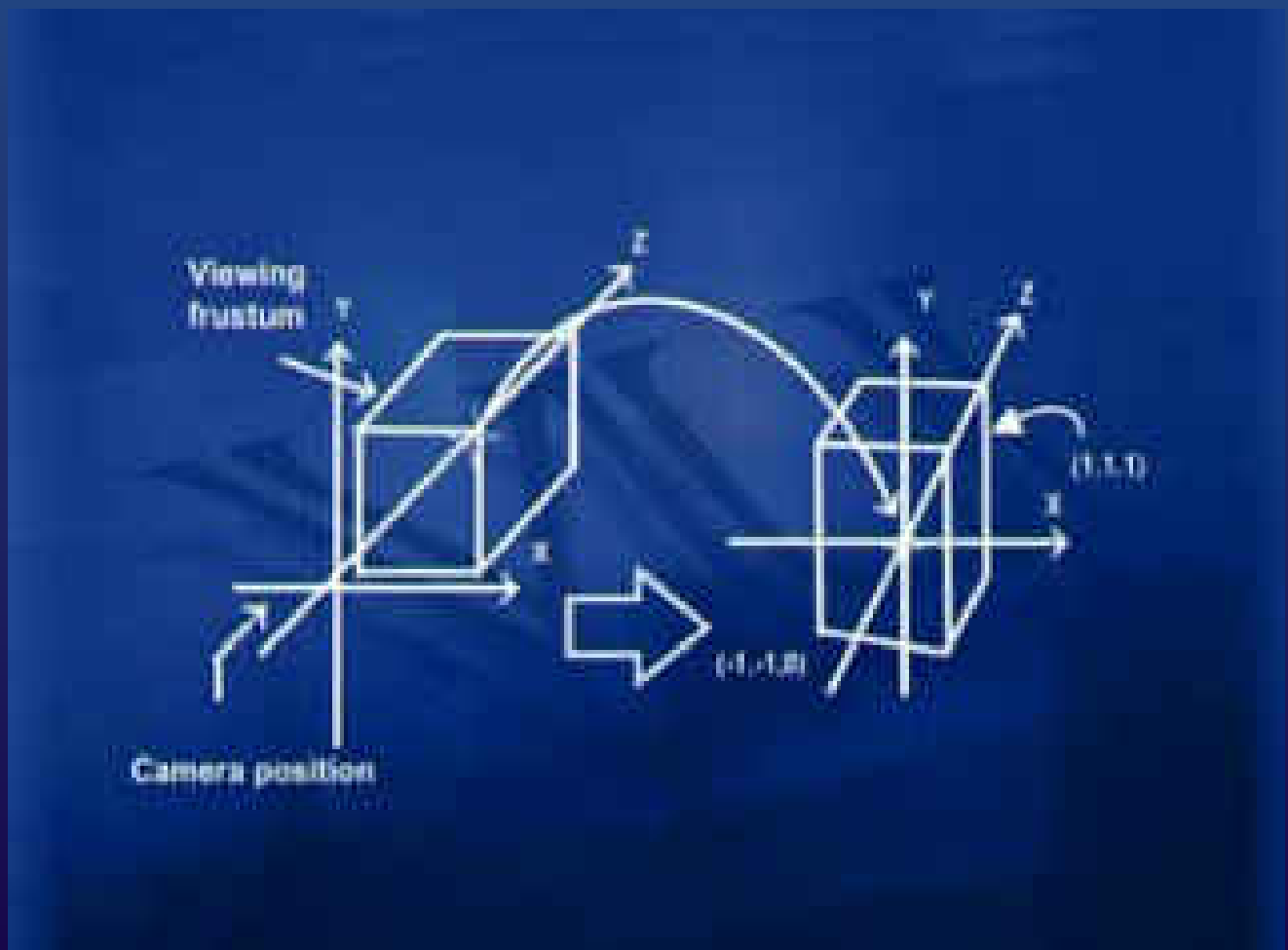


$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1/D \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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

$$T * P$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1/D \\ 0 & 0 & -D & 0 \end{bmatrix}$$

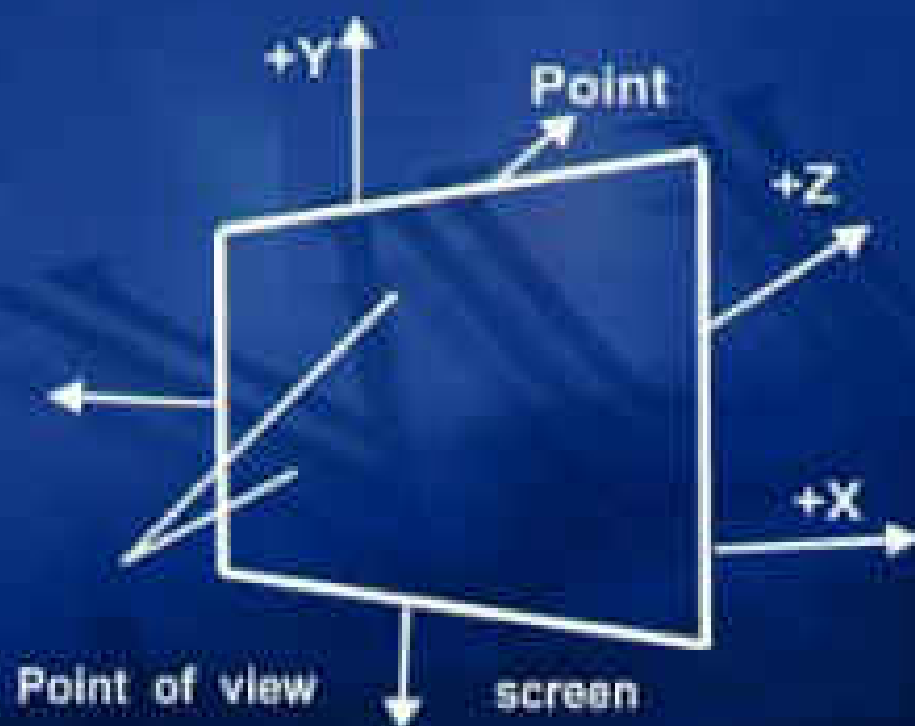
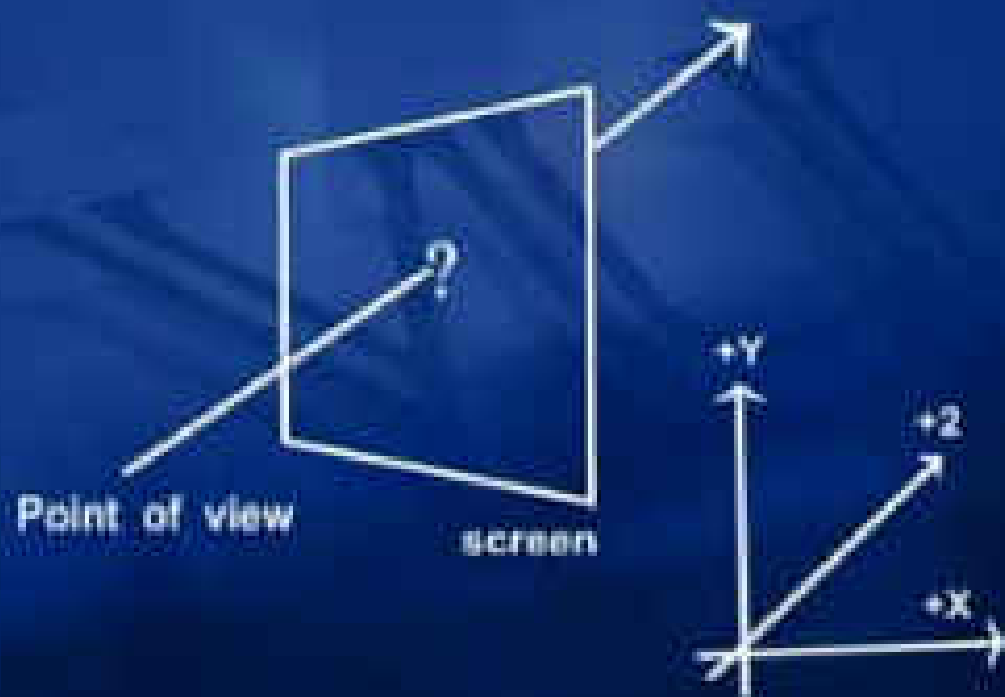


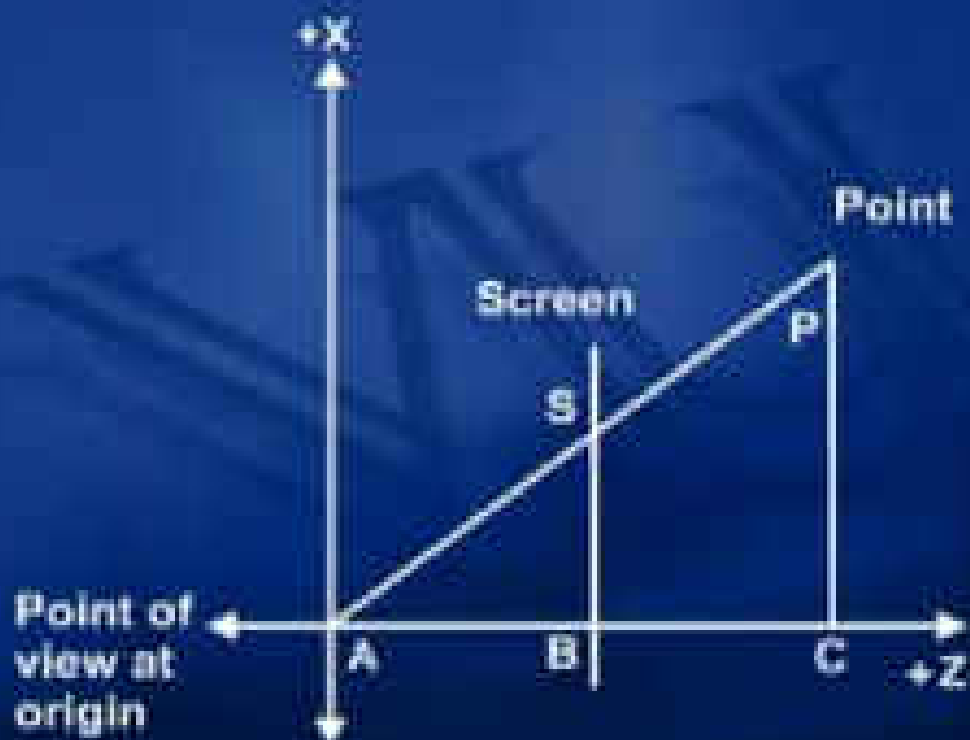
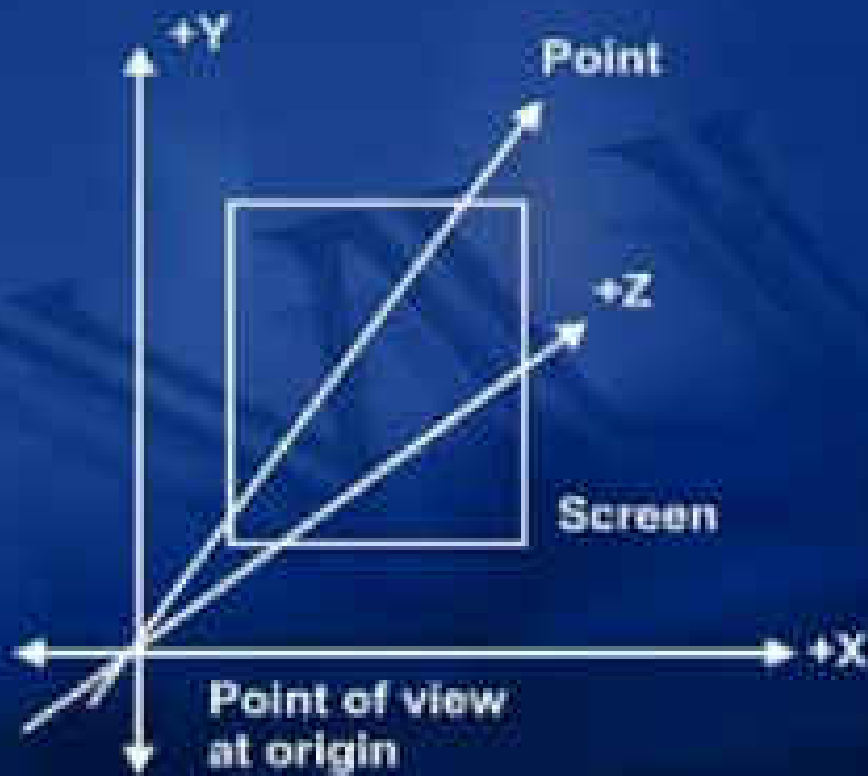
$$\begin{bmatrix} w & 0 & 0 & 0 \\ 0 & h & 0 & 0 \\ 0 & 0 & Q & 1 \\ 0 & 0 & -QZ_n & 0 \end{bmatrix}$$

$$w = \cot \left[ \frac{fov_w}{2} \right]$$

$$h = \cot \left[ \frac{fov_h}{2} \right]$$

$$Q = \frac{Z_f}{Z_f - Z_n}$$







# Computer Graphics

## Lecture 21

### Triangles and Planes

Taqdees A. Siddiqi  
cs602@vu.edu.pk

### Triangles

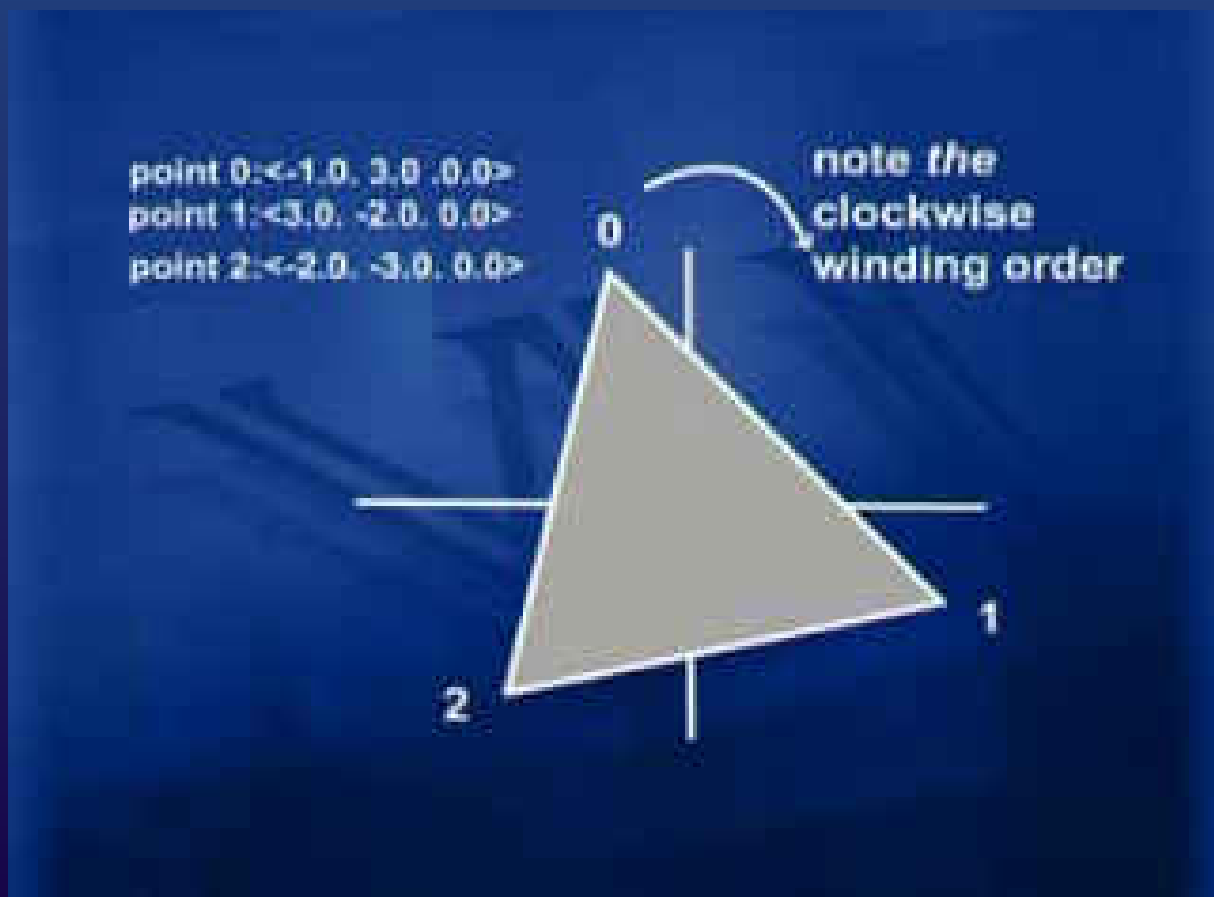
- ◆ Triangles are to 3D graphics what pixels are to 2D graphics
- ◆ Every PC hardware accelerator under the sun uses triangles as the fundamental drawing primitive

- ◆ When we draw a polygon..
- ◆ Hardware devices really use a fan of triangles

- ◆ Triangles “flesh out” a 3D object, connecting them together to form a skin or mesh that defines the boundary surface of an object

- ◆ The ordering of the vertices goes clockwise around the triangle

## Triangles



- ◆ It is impossible to see triangles that face away from us. (we can find this out by computing the triangle plane normal and performing a dot product with a vector from the camera location to a location on the plane)

- ◆ The code that defines triangle with the help of structure and public constructor:

- ◆ Struct triangle

Type v[3]; //Array access useful for loops

Ttriangle() { //nothing }

Triangle(type v0, type v1, type v2)

{

V[0] = v0; v[1]; v[2] = v2;

}

## Triangle Strips and Fans

- ◆ Lists of triangles are generally represented in one of three ways:

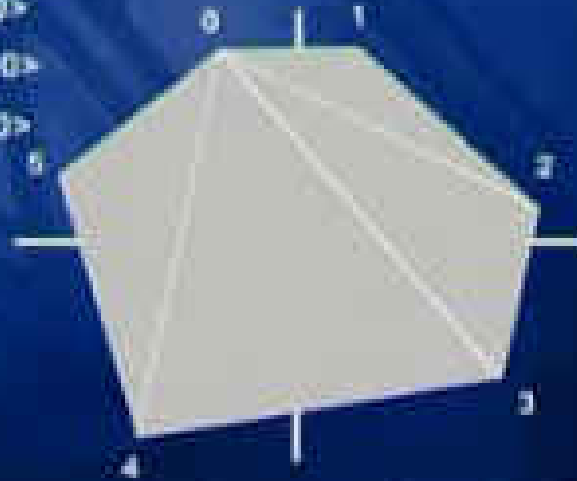
- ◆ The first is an explicit that or array of triangles. Where every three elements represent a new triangle.

- ◆ However there are two additional representation design to solve bandwidth while sending triangle to dedicated hardware to draw them. They are called triangle strips and triangle fans.

## Triangle Fans

- ◆ Triangles fans, conceptually, look like the folding fans you see in souvenir shops. They are a list of triangles that all share a common point.

point 0: <-1.0, 3.0, 0.0>  
point 1: <1.0, 3.0, 0.0>  
point 2: <3.5, 0.5, 0.0>  
point 3: <3.0, -2.0, 0.0>  
point 4: <-2.0, -3.0, 0.0>  
point 5: <-3.0, 1.0, 0.0>



**A list of points composing a triangle fan**

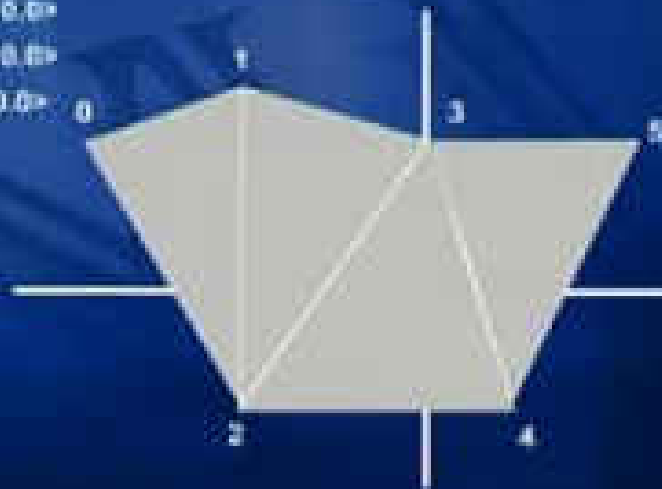
- ◆ An N-sided polygon can be represented efficiently using a triangle fan



- ◆ Triangles in a triangle strip, instead of sharing a common element like a fan, only share elements with the triangle immediately preceding them.

- ◆ The first three elements defines the first triangle. then each subsequent element is combined with the two elements before it, in clockwise order, to create a new triangle

point 0: <-3.0, 1.0, 0.0>  
point 1: <-2.0, 1.5, 0.0>  
point 2: <-2.0, -1.5, 0.0>  
point 3: <0.0, -1.5, 0.0>  
point 4: <0.5, -2.5, 0.0>  
point 5: <2.5, 1.0, 0.0>



**A list of points composing a triangle strip**

**Plane**

- ◆ Planes are to 3D what lines are to 2D

- ◆ Planes are  $n-1$  dimensional hyper planes that can help us accomplish various tasks.

- ◆ Planes are defined as infinitely large, infinitely thin slices of space, like big pieces of paper.

- ◆ Each of the triangles that make up our model, exist in their own plane

- ◆ When we have a plane that represents a slice of 3D space, we can perform operations like classification of points, polygons and clipping.

- ◆ How do we represent planes?

$$Ax + by + cz + d = 0$$

Equation that defines a plane in 3D

$$Ax + By + Cz + d = 0$$

The triplet (a, b , c) represents what is called the normal of the plane

# Normal

- ◆ A normal is a unit vector that, conceptually speaking, sticks directly out of a plane. A stronger mathematical definition would be that the normal is a vector that is perpendicular to all of the points that lie in the plane.

- ◆ The  $d$  component in the equation represents the distance from the plane to the origin. The distance is computed by tracing a line towards the plane until you hit it.

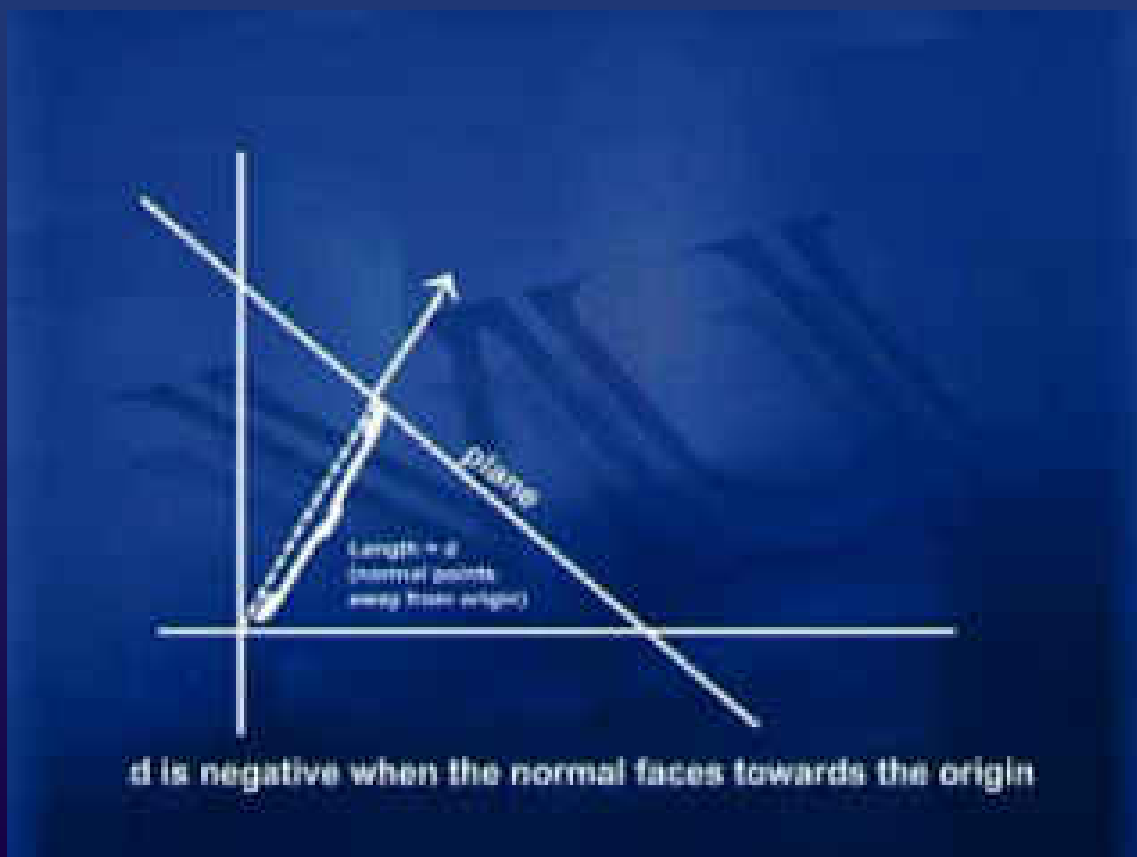
- ◆ Finally the triplet  $(x,y,z)$  is any point that satisfies the equation

- ◆ Finally the triplet  $(x,y,z)$  is any point that satisfies the equation
- ◆ The set of all points  $(x,y,z)$  that solve the equation is exactly all the points that lie in the plane

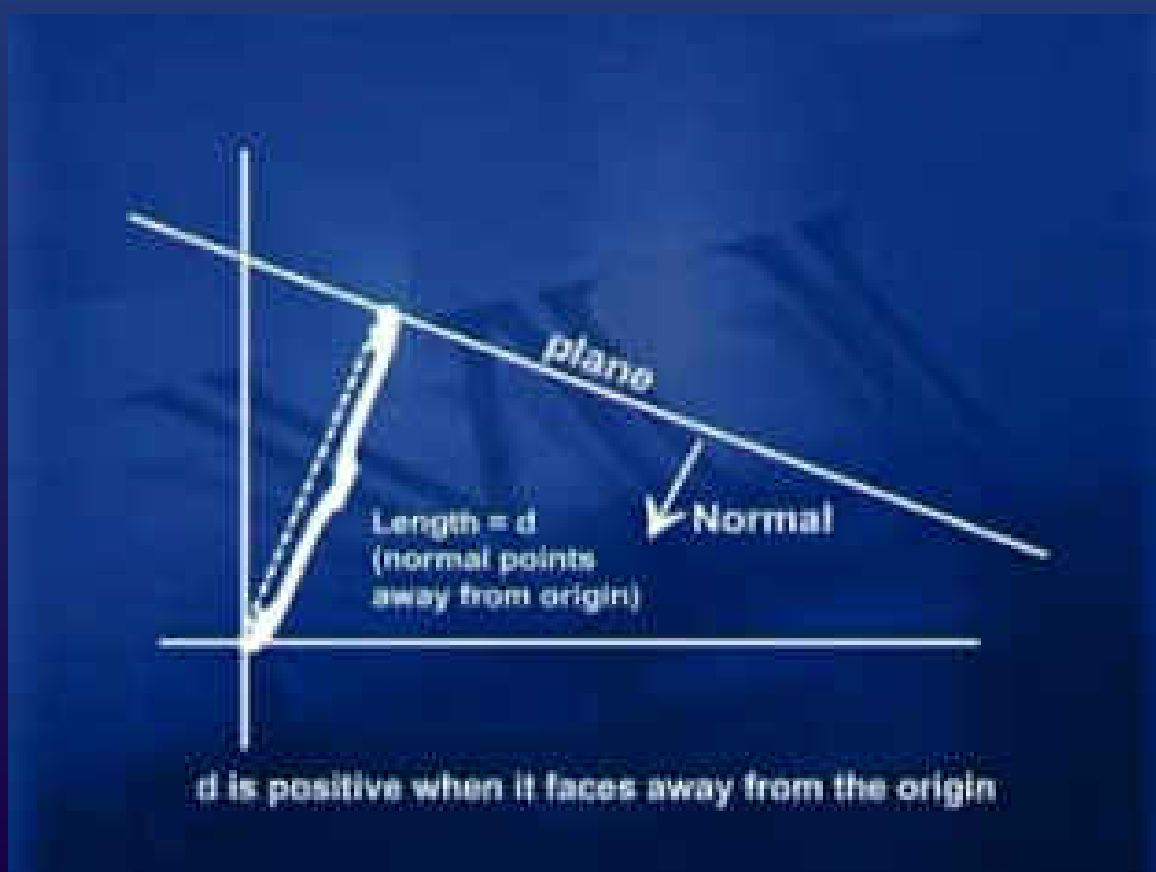


# Examples of planes

- ◆ Following are two examples of planes. The first has the normal pointing away from the origin, which causes  $d$  to be negative
- ◆ Try some sample values for yourself if this doesn't make sense

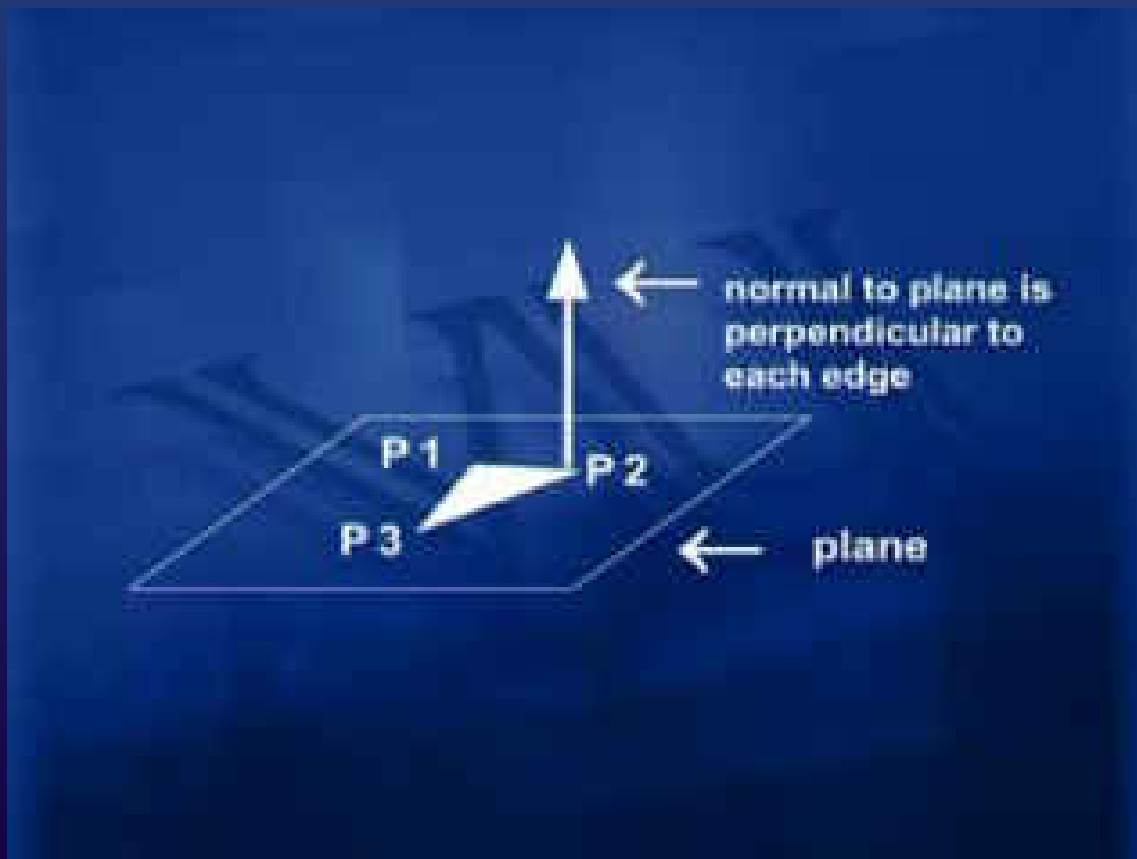


- ◆ The second has the normal pointing towards the origin, so  $d$  is positive. Of course, if the plane goes through the origin,  $d$  is zero (the distance from the plane to the origin is zero)



- ◆ It's important to notice that technically the normal  $(x,y,z)$  does not have to be unit-length for it to have a valid plane equation. But since things end up nicer if the normal is unit-length

- ◆ We just perform a cross product between the two vectors made up by the three points



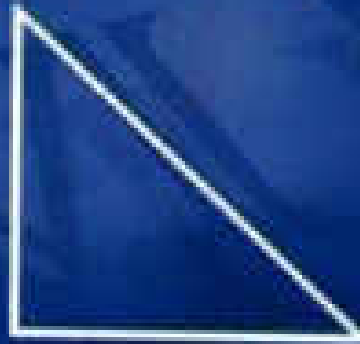
- ◆ Normal =  $(p1-p2) \times (p3-p2)$
- ◆  $K = \text{normal} * p1$

- ♦ Note that it is extremely important to keep track of which direction your points are stored in. let's take 3 points stored in clockwise direction in the x/y plane:

$P2 = \langle 0.1.0 \rangle$

$P1 = \langle 0.0.0 \rangle$

$P3 = \langle 1.0.0 \rangle$



- ◆ The normal to the plane these 3 points can be defined as

$$\begin{aligned}\text{Normal} &= (p_1 - p_2) \times (p_3 - p_2) \\ &= (0, -1, 0) \times (1, -1, 0) \\ &= \langle (-1) \cdot 0 - 0 \cdot (-1), \\ &\quad 0 \cdot 1 - 0 \cdot 0, \\ &\quad 0 \cdot (-1) - (-1) \cdot 1 \rangle \\ &= \langle 0, 0, 1 \rangle\end{aligned}$$

- ◆ The z axis. If we were to store the points counter-clockwise the normal calculated would be  $\langle 0, 0, -1 \rangle$ , which is still the z axis but in the “opposite” direction. It’s important to keep track of these things since we often need plane equations to be correct in order to determine which side of a polygon an object (such as the view point) is on.

## Constructing a plane from three points on the plane

- ◆ Normal vector =  $n$
- ◆  $n = \text{cross product } (b-a), (c-a)$
- ◆ Normalize ( $n$ ); find a unit vector
- ◆  $D = -\text{dot product } (n, a)$

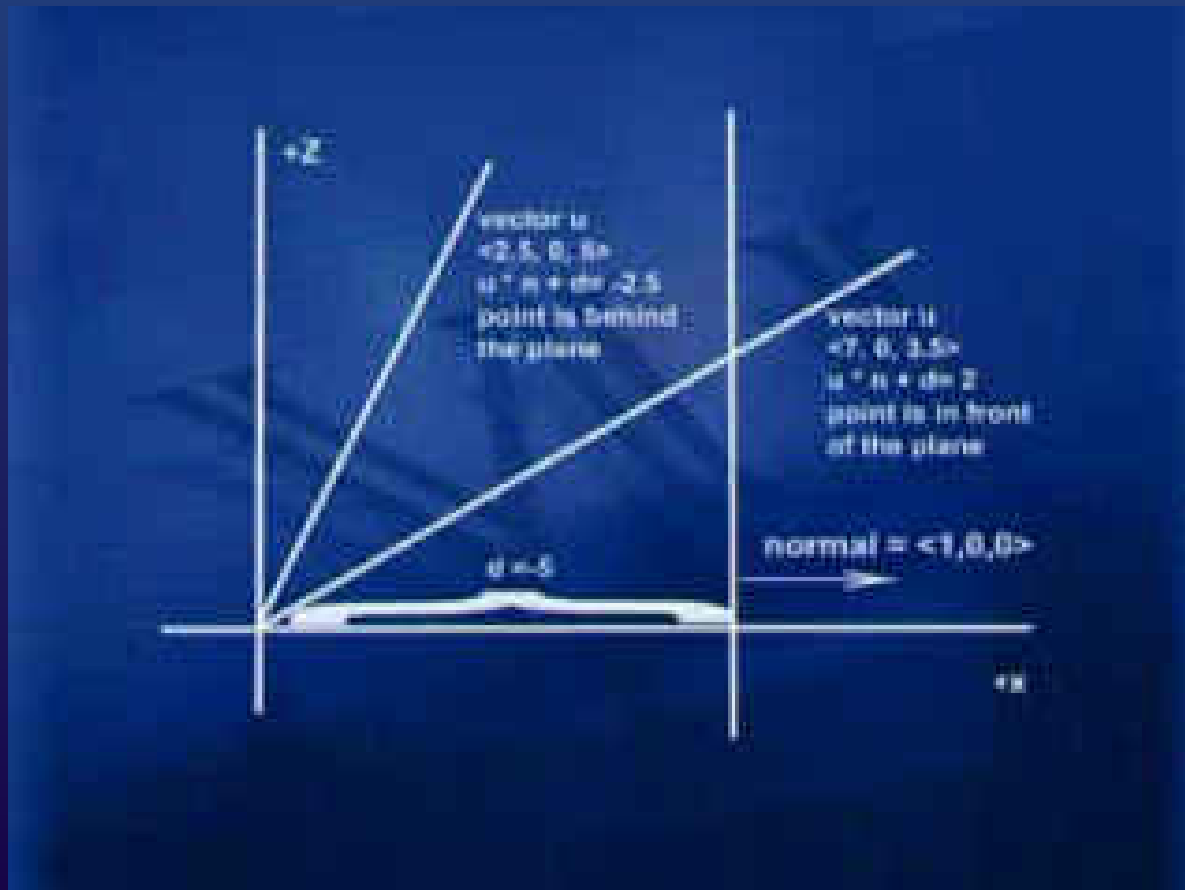
## Constructing a plane from a normal and a point on the plane

- ◆ Normalize (n); find a unit vector
- ◆  $d = -\text{dot product } (n, a)$

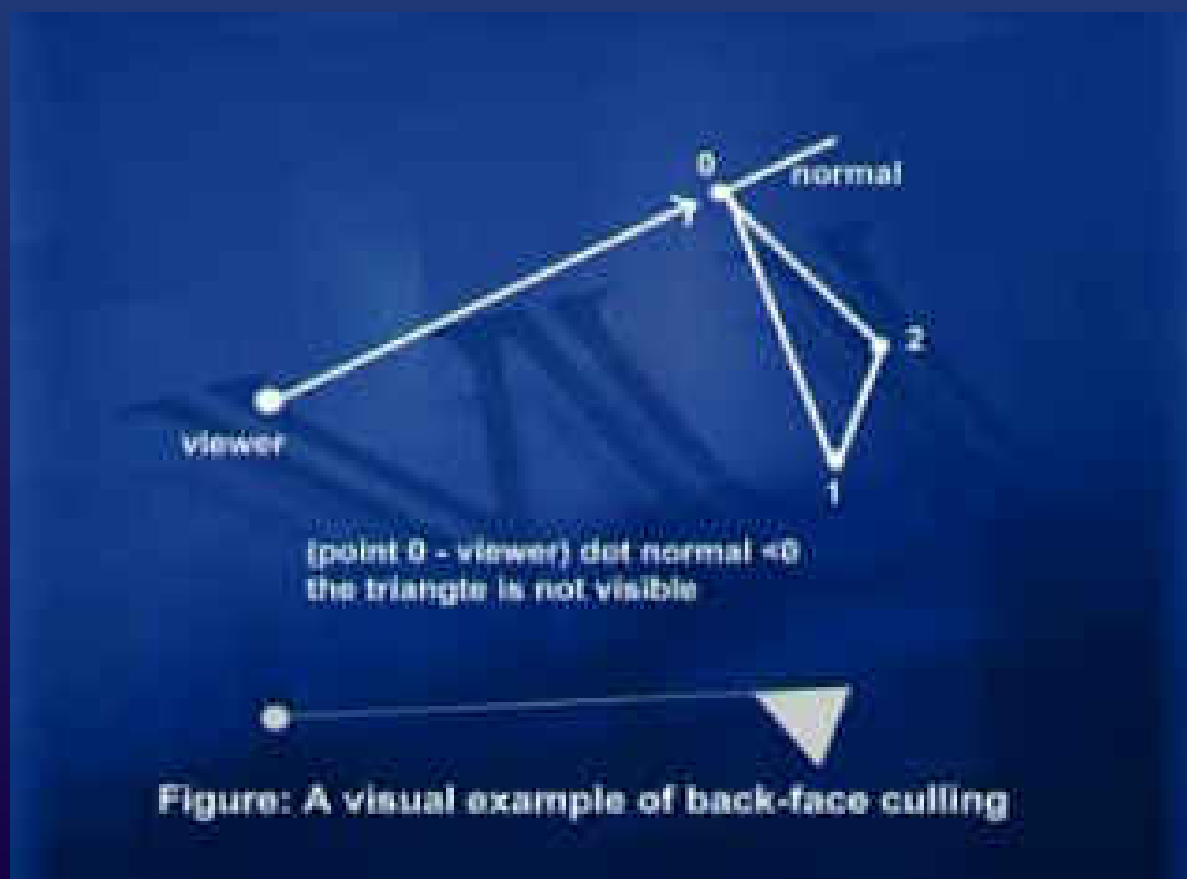


# Defining Locality with Relation to a Plane

- ◆ One of the most important operations planes let you perform is defining the location of a point with respect to a plane. If you drop a point into the equation, it can be classified into three cases: in front of the plane, in back of the plane, or coplanar with the plane



- ◆ To perform the back-face cull, just subtract one of the triangle's points from the camera location and perform a dot product with the resultant vector and the normal. If the result of the dot product is greater than zero, then the view point was in front of the triangle



## Intersection Between a line and a plane

This occurs at the point which satisfies both the line and the plane equations

- ◆ Line equation

$$p = \text{org} + u * \text{dir} \quad (1)$$

Plane equation

$$p * \text{normal} - k = 0. \quad (2)$$

Substituting (1) into (2) and rearranging we get :

- ◆  $(\text{org} + u * \text{dir}) * \text{normal} - k = 0$
- ◆ i.e.,  $u * \text{dir} * \text{normal} = k - \text{org} * \text{normal}$
- ◆ i.e.,  $u = (k - \text{org} * \text{normal}) / (\text{dir} * \text{normal})$

- ◆ If  $(d * \text{normal}) = 0$  then the line runs parallel to the plane and no intersection occurs. The exact point at which intersection does occur can be found by plugging the value of parameter  $u$  back into the line equation in (1)

# Computer Graphics

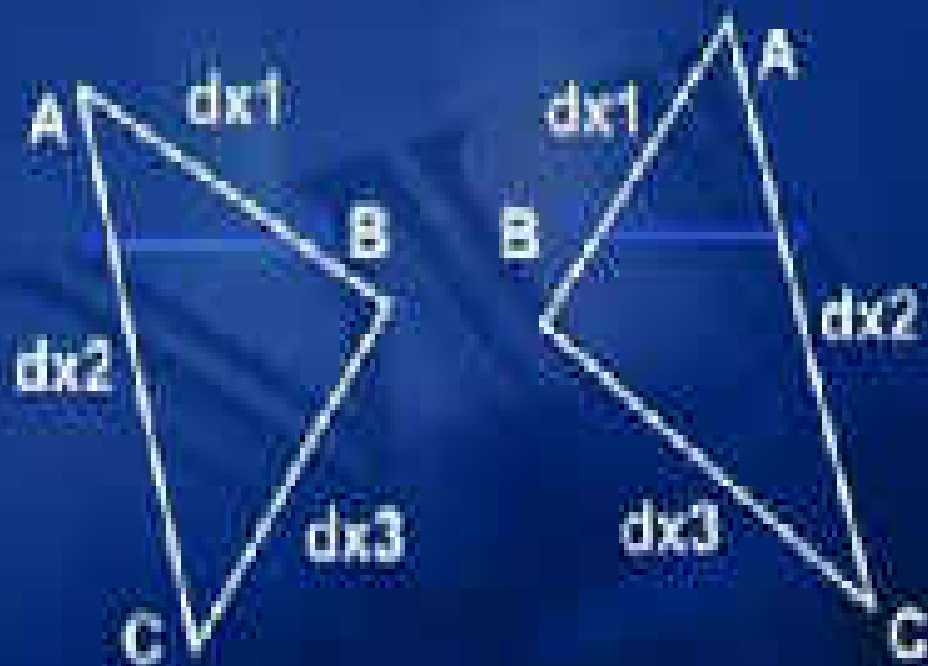
## Lecture 22 Triangles and Planes

Taqdees A. Siddiqi  
cs602@vu.edu.pk

- ◆ High performance triangle pasteurization is a very important topic in computer Graphics in today's world.
- ◆ Triangles are the foundation of modern real time graphics, and are by far the most popular rendering primitive

For each scan line (horizontal line on the screen), find the points of intersection with the edges of the triangle. Then, draw a horizontal line between intersections and do this for all scan lines.

**First filling Triangle**



- But how can we find these points quickly ?
- Using linear interpolation!



- We have 3 vertices and we want to find coordinates of all points belonging to segments determined by these vertices

- Assume we have segment given by points:
  - $(x_a, y_a)$  and  $(x_b, y_b)$

- Our task is to find points:

$(x_c, y_a+1), (x_d, y_a+2), \dots,$

$(x_m, y_b-1), (x_n, y_b)$

- Notice that  $x_a$  changes to  $x_b$  in  $(y_b-y_a)$  steps.

- We also have:

- $X_a = x_a + 0 * (x_b-x_a)/(y_b-y_a),$

- $X_b = x_a + (y_b-y_a) * (x_b-x_a) / (y_b-y_a)$

- and, in general,  $x_i = x_a + (y_i-y_a) * \text{delta},$

- where  $\text{delta} = (x_b - x_a) / (y_b - y_a)$

- The general function for linear interpolation is:

- $f(X) = A + X * ((B-A) / \text{steps})$   
where we slide from A to B  
in steps

- The coordinates of vertices are

- (A.x, A.y), (B.x, B.y),  
(C.x, C.y);

**we assume that**

$$A.y \leq B.y \leq C.y$$

**we should sort them  
first**

- **dx1, dx2, dx3 are deltas  
used in interpolation**

- Horizontal line draws horizontal segment with coordinates

- $(S.x, Y), (E.x, Y)$

- $S.x, E.x$  are left and right x-coordinates of the segment we have to draw

- $S = A$  means that
- $S.x = A.x; S.y = A.y;$

```
if (B.y - A.y > 0)
 dx1 = (B.x - A.x) / (B.y - A.y)
else
 dx1 = 0;
```

```
if (C.y - A.y > 0)
 dx2 = (C.x - A.x) / (C.y - A.y)
else
 dx2 = 0;
```

```
if (C.y - B.y > 0)
 dx3 = (C.x - B.x) / (C.y - B.y)
else
 dx3=0;
```

- $S = E = A$

- if(dx1 > dx2) Then  
for( ; S.y <= B.y; S.y++,  
E.y++, S.x += dx2, E.x +=  
dx1)  
horizontal line (S.x, E.x, S.y,  
E.x, color);



```
E=B;
for(;S.y<=C.y;S.y++,E.y+
+,S.x+=dx2,E.x+=dx3)
horizontal line
(S.x,E.x,S.y,E.x,color);
END IF
```

```
E=B;
for(;S.y<=C.y;S.y++,E.y+
+,S.x+=dx2,E.x+=dx3)
horizontal line
(S.x,E.x,S.y,E.x,color);
END IF
```

■ else

```
for(;S.y<=B.y;S.y++,E.y++
+,S.x+=dx1,E.x+=dx2)
```

horizontal line

```
(S.x,E.x,S.y, E.x,color);
```

```
S=B;
```

```
for(;S.y<=C.y;S.y++,E.y++
+,S.x+=dx3,E.x+=dx2)
```

horizontal line

```
(S.x,E.x,S.y,E.x,color);
```

■ End else

## Gouraud Shading Triangle

- The idea of gouraud and flat triangle is nearly the same. Gouraud takes only three parameters more (the color value of each of the vertices), and the routine just interpolates among them drawing a beautiful, shaded triangle.

- You can use 256-colors mode, in which vertices' colors are simply indices to palette or hi-color mode (recommended).

- Flat triangle interpolated only one value (x in connection with y), 256 colors gouraud needs three (x related to y, color related to y, and color related to x),

- hi-color gouraud needs seven (x related to y, red, green and blue components of color related to y, and color related to x (also three components))

- Drawing a gouraud triangle, we add only two parts to the flat triangle routine. The horizontal line routine gets a bit more complicated due to the interpolation of the color value related to x but the main routine itself remains nearly the same.

the coordinates of  
vertices are  $(A.x, A.y)$ ,  
 $(B.x, B.y)$ ,  $(C.x, C.y)$

we assume that

$$A.y \leq B.y \leq C.y$$

you should sort them  
first

vertex A has color

$(A.r, A.g, A.b)$ ,

B  $(B.r, B.g, B.b)$ ,

C  $(C.r, C.g, C.b)$ ,

where  $X.r$  is color's red component,  
 $X.g$  is color's green component and  
 $X.b$  is color's blue component

$dx1$ ,  $dx2$ ,  $dx3$  are deltas used in interpolation of x-coordinate

*dr1, dr2, dr3, dg1, dg2, dg3.*

*db1, db2, db3* are deltas used in interpolation of color's components

*putpixel(P)* plots a pixel with coordinates  $(P.x, P.y)$  and color  $(P.r, P.g, P.b)$



$S=A$

means that

$S.x=A.x; S.y=A.y; S.r=A.r;$   
 $S.g=A.g; S.b=A.b;$

if (B.y-A.y > 0) {

dx1=(B.x-A.x) / (B.y-A.y);

dr1=(B.r-A.r)/(B.y-A.y);

dg1=(B.g-A.g)/(B.y-A.y);

db2=(C.b-A.b)/(C.y-A.y);}

else

dx3=dr3=dg3=db3=0;

```
if (C.y-A.y > 0) {
 dx2=(C.x-A.x)/(C.y-A.y);
 dr2=(C.r-A.r)/(C.y-A.y);
 dg2=(C.g-A.g)/(C.y-A.y);
 db2=(C.b-A.b)/(C.y-A.y);}
else
 dx2=dr2=dg2=db2=0;
```

```
S=E=A;
if(dx1 > dx2) {
 for(;S.y<=B.y;S.y++,E.y++)
 {
 if(E.x-S.x > 0) {
 dr=(E.r-S.r)/(E.x-S.x);
 dg=(E.g-S.g)/(E.x-S.x);
 db=(E.b-S.b)/(E.x-S.x);}
 }
```

```
else
 dr=dg=db=0;
 P=S;
 for(;P.x < E.x;P.x++){
 putpixel(P);
 P.r+=dr;
 P.g+=dg;
 P.b+=db; }
```

```
S.x+=dx2; S.r+=dr2;
S.g+=dg2; S.b+=db2;
E.x+=dx1; E.r+=dr1;
E.g+=dg1; E.b+=db1;
}
```

```
E=B;
```

```
for(;S.y<=C.y;S.y++,E.y++) {
 if(E.x-S.x > 0) {
 dr=(E.r-S.r)/(E.x-S.x);
 dg=(E.g-S.g)/(E.x-S.x);
 db=(E.b-S.b)/(E.x-S.x);
 }
}
```

```
else
```

```
 dr=dg=db=0;
 P=S;
 for(;P.x < E.x;P.x++) {
 putpixel(P);
 P.r+=dr;
 P.g+=dg;
 P.b+=db;
 }
```

```
else {
 for(;S.y<=B.y;S.y++,E.y++){
 if(E.x-S.x > 0) {
 dr=(E.r-S.r)/(E.x-S.x);
 dg=(E.g-S.g)/(E.x-S.x);
 db=(E.b-S.b)/(E.x-S.x);
 }
 }
```

```
else
 dr=dg=db=0;
P=S;
for(;P.x < E.x;P.x++) {
 putpixel(P);
 P.r+=dr;
 P.g+=dg;
 P.b+=db;
}
```

# **Computer Graphics**

## **Lecture 22**

# **Computer Graphics**

## **Lecture 23** **Lightning I**

**Taqdees A. Siddiqi**  
**cs602@vu.edu.pk**

# Textured Triangle

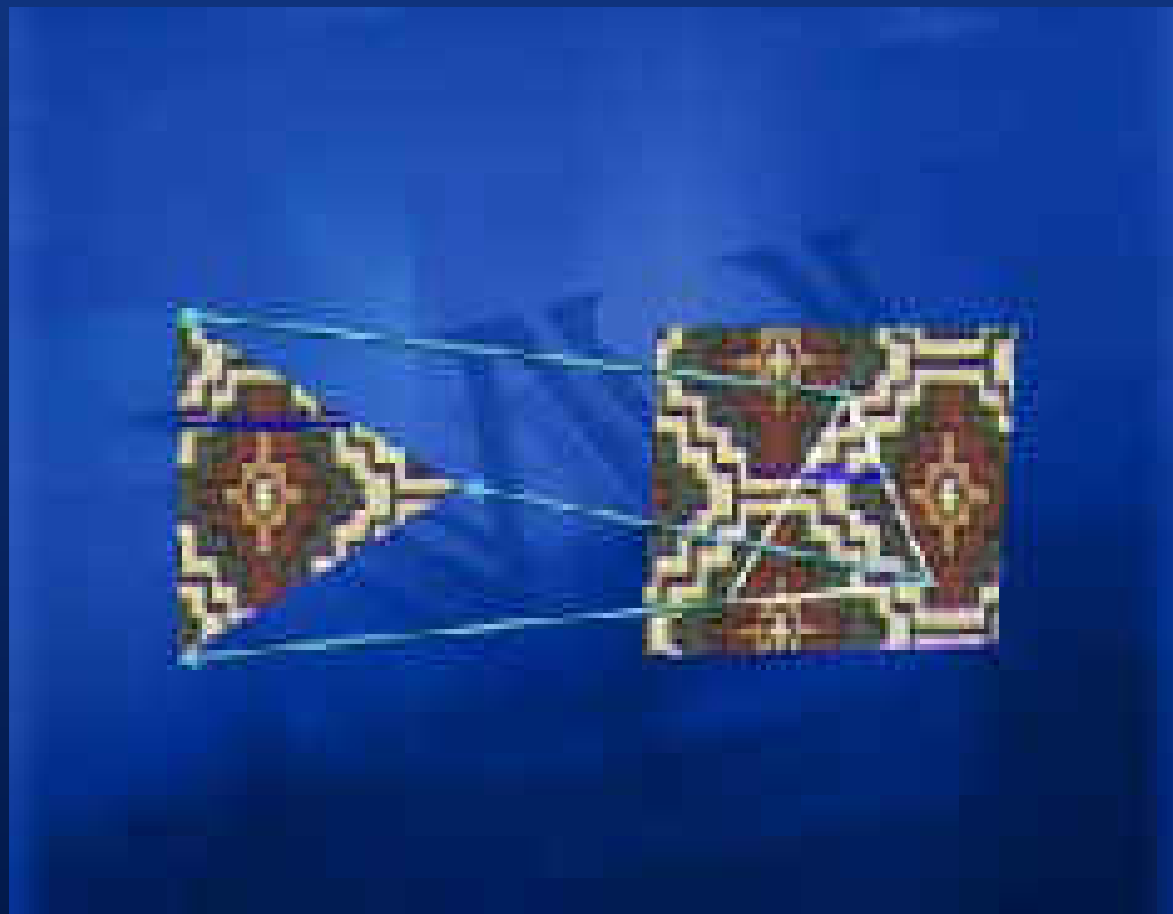
We can also apply any bitmap on triangle for filling it.



Again we're using the idea of interpolation: now we'll code a texture triangle filler. And again the idea is perfectly the same, only two more values to interpolate, that is five values total

In texture mapping, we interpolate  $x$ ,  $u$ , and  $v$  related to  $y$ , and  $u$  and  $v$  related to  $x$  ( $u$  and  $v$  are coordinates in the 2D bitmap space)





The left triangle is the triangle which is drawn onto the screen. There's a single scanline (one call to the horizontal line routine) pointed out as an example.

The triangle on the right is the same triangle in the bitmap space, and there's the same scanline drawn from another point of view into it, too..

So we need just to interpolate, interpolate, and once more interpolate in a texture filler

- $du = (Eu - Su) / (Ex - Sx).$

- we do like this in the setup part of the polygon routine:
- We know that

- $S.x = A.x + (B.y - A.y) * dx1,$

- $S.u = A.u + (B.y - A.y) * du1,$   
 $E.x = B.x = A.x + (B.y - A.y) * dx2,$   
 $E.u = B.u = A.u + (B.y - A.y) * du2,$

- When  $y = B.y$  (when  $y$  is the  $y$ -coordinate of the second vertex).
- When we place the values of the variables  $S.u, E.u, S.x$  and  $E.x$  (above) to the  $u$  delta statement,

we get the following  
statement as a result:

$$du = \frac{(A.u+(B.y-A.y)^2 du2) - (A.u+(B.y-A.y)^2 du1)}{(A.x+(B.y-A.y)^2 dx2) - (A.x+(B.y-A.y)^2 dx1)}$$

$$du = \frac{(B.y-A.y)^2 (A.u-A.u+du2-du1)}{(B.y-A.y)^2 (A.x-A.x+dx2-dx1)}$$

$$du = \frac{du_2 - du_1}{dx_2 - dx_1}$$

$$\text{innerUdelta} = \frac{\text{outerUdelta2} - \text{outerUdelta1}}{\text{outerXdelta2} - \text{outerXdelta1}}$$

- Nice! But what if  $dx_2 = dx_1$ ?  
This of course means that the polygon is just one line, so  $dx$  doesn't need any specific value; zero does the job very well.

## Lighting 1

### COLOR

- It is important to understand how color is represented in computer graphics so that we can manipulate it effectively

- A color is usually represented in the graphics pipeline by a three-element vector representing the intensities of the red, green, and blue components



- Or for a more complex object, by a four-element vector containing an additional value called the alpha component that represents the opacity of the color

- Thus we can talk about RGB or RGBA colors and mean a color that's made up of either three or four elements. There are many different ways of representing the intensity of a particular color element.

- Colors can also be represented as floating point values in the range  $[0,1]$

A 16-bit display is named since each pixel in a 16-bit image is taken up by 16 bits (2 bytes):

- 5 bits of red information,
- 6 bits of green information,
- and 5 bits of blue information

Incidentally, the extra bit (and therefore twice as much color resolution) is given to green because our eyes are more sensitive to green

- A 24-bit display, of course, uses 24 bits, or 3 bytes per pixel, for color information. This gives 1 byte, or 256 distinct values each, for red, green, and blue

- This is generally called true colors, because  $(256)^3$  (i.e. 16.7 million) colors is about as much as your eyes can discern, so more color resolution really isn't necessary, at least for computer monitors

Finally, there is 32-bit color, something seen on most new graphics cards. Many 3D accelerators keep 8 extra bits per pixel around to store transparency information, which is generally referred to as the alpha channel

And therefore take up 4 bytes, or 32 bits, of storage per pixel. Rather than re-implement the display logic on 2D displays that don't need alpha information, these 8 bits are usually just wasted.

## Representing Color

- Before we can go about giving color to anything in a scene, we need to know how to represent color!

- Usually we use the same red, green, and blue (RGB) channels discussed before

- Practically we will use two structures to ease the color duties:
  - color3 and
  - color4

Define unname union to store 4 values

union:

struct:

float r, g, b, a  
float c[4];

Define constructor:  
`color4( const color3& in,  
float alpha = 1.f )`  
    `r = in.r`  
    `g = in.g`  
    `b = in.b`  
    `a = alpha`

Define another constructor  
`color4(unsigned long color)`  
    `b = (float)(color&255) / 255.f;`  
    `color >>= 8;`  
    `g = (float)(color&255) / 255.f;`



```
color >>= 8;
r = (float)(color&255) / 255.f;
color >>= 8;
a = (float)(color&255) / 255.f;
```

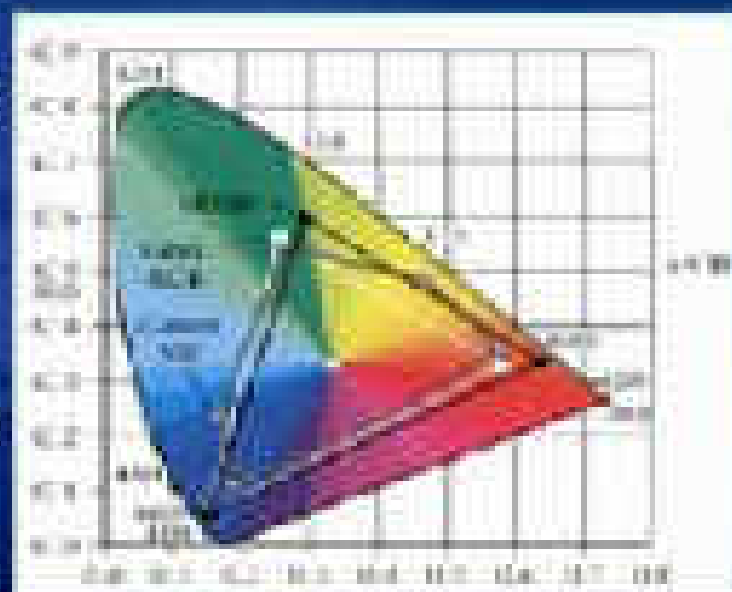
- Now make a function that converts back to double word (unsigned long) value

```
unsigned long MakeDWord()
unsigned long iA = (int)(a*255.f) << 24;
unsigned long iR = (int)(r*255.f) << 16;
unsigned long iG = (int)(g*255.f) << 8;
unsigned long iB = (int)(b*255.f);
return iA | iR | iG | iB;
```

```
void Saturation()
if(r > 1) r = 1.f;
if(g > 1) g = 1.f;
if(b > 1) b = 1.f;
if(a > 1) a = 1.f;
```

```
if(r < 0.f) r = 0.f;
if(g < 0.f) g = 0.f;
if(b < 0.f) b = 0.f;
if(a < 0.f) a = 0.f;
```

The 1931 CIE  
diagram  
shows the  
gamut of the  
eye and the  
lesser gamut  
of output  
devices.



# Multiplying Color Values

- vectors where the individual color elements are limited to the  $[0,1]$  range. Operations on colors are done piecewise – that is, even though we represent colors as RGB vectors, they aren't really vectors in the mathematical sense.

- Vector multiplication is different from the operation we perform to multiply color. We'll use special symbol to indicate such piecewise multiplication.

- Colors are multiplied to describe the interaction between a surface and a light source

- The colors of each are multiplied together to estimate the reflected light color—this is the color of the light that this particular light reflects off this surface

- The problem with the standard RGB model is just that we're simulating the entire visible spectrum by three colors with a limited range.

- Let's start with a simple example of using reflected colors

- This intensity of our light is represented by, say, a nice lime green color

light color  $l_s = [0.34765, 0.92578, 0.24609]$

- Let's say we shine this light on a nice magenta surface given by  $c_s$ .

surface color  $c_s = [0.86719, 0.00000, 0.98828]$

- So, to calculate the color contribution of this surface from this particular light, we perform a piecewise multiplication of the color values



$$\begin{aligned}
 \vec{I}_L \otimes \vec{C}_S &= [0.34765, 0.92578, 0.24609] \otimes [0.86719, 0.00000, 0.98828] \\
 &= [(0.34765)(0.86719), (0.92578)(0), (0.24609)(0.98828)] \\
 &= [0.30148, 0.00000, 0.24320]
 \end{aligned}$$

- This gives us the dark plum color shown in figure below:



Light color



Surface color



Final color

- We should note that since the surface has no green component, that no matter what value we used for the light color, there would never be any green component from the resulting calculation.

Thus a pure green light would provide no contribution to the intensity of a surface if that surface contained a zero value for its green intensity.

- Thus it's possible to illuminate a surface with a bright light and get little or no illumination from that light. We should also note that using anything other than a full-bright white light  $[1,1,1]$



Multiplying (modulating) color values results in a color equal to or less than (darker) the original two.

Given that using a colored light in a scene makes the scene darker, how do we make the scene brighter?

**Dealing with Saturated Colors**

While the colors of lights are modulated by the color of the surface, each light source that illuminates the surface is added to the final color. All these colors are summed up to calculate the final color.

We'll start with summing the reflected colors off a surface from two lights. The first light is an orange color and has RGB values  $[1.0, 0.49, 0.0]$ , and the second light is a nice light green with RGB values  $[0.0, 1.0, 0.49]$ .

- Summing these two colors yields  $[1.0, 1.49, 0.49]$ , which we can't display because of the values larger than one figure below shows



Adding colors can result in colors that are outside the displayable range.

- So, what can be done when color values exceed the range that the hardware can display? It turns out that there are three common approaches

- The first is clamping the color values is implemented in hardware

- The second most common approach is to scale the colors by the largest component. This maintains the color but reduces the overall intensity of the color.

- The third is to try to maintain the intensity of the color by shifting (or clipping) the color toward pure bright white by reducing the colors that are too bright while increasing the other colors and maintaining the overall intensity





**The results of three strategies  
for dealing with the same  
oversaturated color.**

# Computer Graphics

## Lecture 24

### Lightning II

Taqdees A. Siddiqi  
cs602@vu.edu.pk



**Clamping Color  
Values**

Any value that's greater than one is clamped to one, and any value less than zero are clamped to zero.

```
if(r > 1) r = 1.f;
if(g > 1) g = 1.f;
if(b > 1) b = 1.f;
if(a > 1) a = 1.f;
```

```
if(r < 0.f) r = 0.f;
if(g < 0.f) g = 0.f;
if(b < 0.f) b = 0.f;
if(a < 0.f) a = 0.f;
```

Scaling Color Values by Intensity

Instead of clamping, we might want to scale the color by dividing by the largest color value, thus scaling the *rgb* values into the  $[0,1]$  range.

Using this approach, we divide each element by 1.49, yielding a scaled color of  $[0.671, 1.0, 0.329]$ .

Thus any values greater than one are scaled to one, while any other values are also scaled by the same amount. This maintains the hue and saturation but loses the intensity.

This might not be acceptable because the contrast with other colors is lost, since contrast perception is nonlinear and we're applying a linear scaling

One problem with clamping or scaling colors is that they get darker (lose saturation). An alternative to scaling is to maintain saturation by shifting color values. This technique is called clipping,

- and it's a bit more complicated than color scaling or clamping. The idea is to create a gray-scale vector that runs along the black-white axis of the color cube

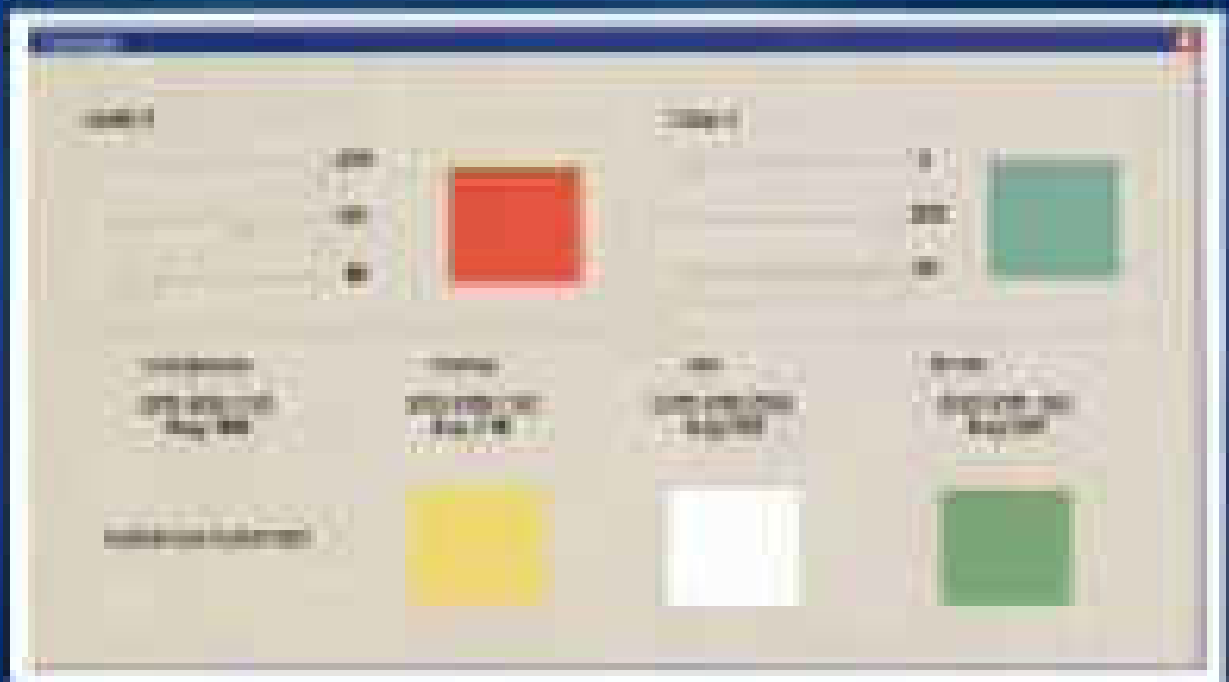
that's got the same brightness as the original color and then to draw a ray at right angles to this vector that intersects (i.e., clips) the original color's vector.

We need to check to make sure that the grayscale vector is itself within the  $[0, 1]$  range and then to check the sign of the ray elements to see if the color elements need to be increased or decreased



- The ColorSpace tool is a handy tool that we can use to interactively add two colors together to see the effects of the various strategies for handling oversaturated colors.

- We simply use the sliders to select the rgb color values for each color. The four displays in the next Figure show the composite, unmodified values of the resulting color (with no color square)



Negative Colors and Dark lights

- We may be wondering, if we can have color values greater than the range in intermediate calculations, can we have negative values? Yes, we can!

- They are called "darklights" after their description in an article [GLASSNER 1992] in *Graphic Gems III*.

- Darklights are nothing more than lights in which one or more of the color values are negative.

- Thus instead of contributing to the overall lighting in a scene, we can specify a light that diminishes the overall lighting in a scene.

- Darklights are used to eliminate bright areas when we're happy with all the lighting in your scene *except* for an overly bright area.

- Darklights can also be used to affect a scene if we want to filter out a specific rgb color

- If we wanted to get a night vision effect, we could use a darklight with negative red and blue values, for example, which would just leave the green channel.

- Now, we may finally learn its dark secrets. A lot of power is hidden away inside the alpha component.

- The alpha component of the RGBA quad represents the opaqueness of a surface.

- An alpha value of 0xFF (255) means the color is completely opaque, and an alpha value of 0x00 (0) means the color is completely transparent.

- Of course, the value of the alpha component is fairly meaningless unless we actually activate the alpha blending step.

- That pixel is combined using blending factors to the pixel that is currently in the frame buffer.



- We can add the two pixels together, multiply them together, linearly combine them using the alpha component, and so forth.

- The name "alpha blending" comes from the fact that generally the blending factors used are either the alpha or the inverse of the alpha.

# The Alpha Blending Equation

- The equation that governs the behavior of the blending is defined as follows:

$$\begin{aligned} \text{Final Color} = & \text{SOURCE} * \\ & \text{SOURCE BLEND FACTOR} \\ & + \\ & \text{DESTINATION} * \\ & \text{DESTINATION BLEND} \\ & \text{FACTOR} \end{aligned}$$

- **Final color** is the color that goes to the frame buffer after the blending operation.

- **Source** is the pixel we are attempting to draw to the frame buffer, generally one of the many pixels in a triangle we have to draw.

- Destination is the pixel that already exists in the frame buffer before we attempt to draw a new one.

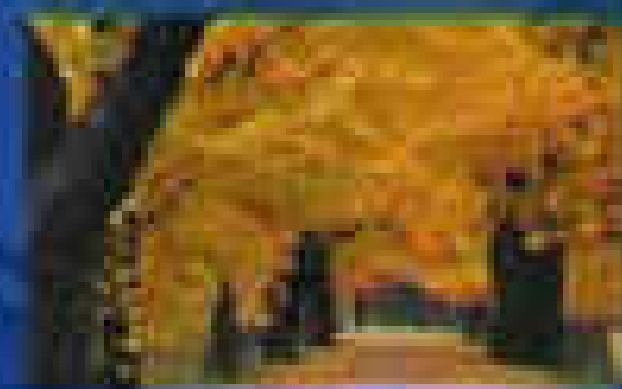
- The source and destination blend factors are variables that modify how the colors are combined together.

The blend factors are the components we have control over in the equation

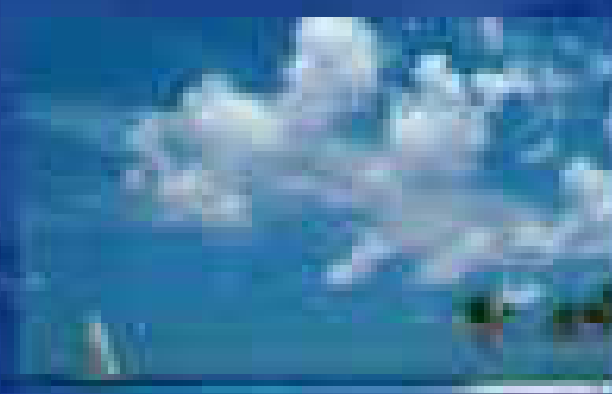
$$\begin{aligned}\text{FINAL COLOR} = \\ \text{SOURCE} * 1.0 + \\ \text{DESTINATION} * 0.0\end{aligned}$$

- As we can see, the destination-blending factor is 0 and the source-blending factor is 1. This reduces the equation to:
- **FINAL COLOR=SOURCE**

- The source code provides example of Alpha blending.
- initially we have two images and we blend first image into second and result is as:



**First Image**



## Second Image

```
struct COLOR3{
 BYTE b;
 BYTE g;
 BYTE r;
};
```

//Assume the two bitmaps have  
the same size

```
COLOR3 *p1; //pointer to 1st bitmap
COLOR3 *p2; //pointer to 2nd bitmap
```



```
int k=0;
now we compute the blending
factor av.
float av=(float)(alpha&255)/255;
/* alphaValue in floating point
compute the new pixel by using
the alpha blending formula. */
```

```
for(int i=firstBMP.Height; i>0; i--)
{
 for(int j=0; j<firstBMP.Width; j++)
 {
 // get the red color
 p1->r =(BYTE)((p1->r * (av))
 + p2->r * (1.0f-(av)));
 // get the green color
 p1->g =(BYTE)((p1->g * (av))
 + p2->g*(1.0f-(av)));
```

```

// get the blue color
p1->b = (BYTE)((p1->b * (av))
+ p2->b*(1.0f-(av)));
p1++;
p2++;
}
}
send the new bitmap to display device
BlitData(displaydeviceContext,0,0,
firstimage.Width,firstimage.Height);

```





# Computer Graphics

## Lecture 25

### Mathematics of Lightning and Shading Part I

Taqdees A. Siddiqi  
[cs602@vu.edu.pk](mailto:cs602@vu.edu.pk)

# Light and Materials

In order to understand how an object's color is determined, we'll need to understand the parts that come into play to create the final color.

First, we need a source of illumination, typically in the form of a light source in our scene.

Lights can also have attenuation, which means that their intensity is a function of the distance from the light to the surface.

Lights can additionally be given other properties such as a shape (e.g., spotlights) and position (local or directional),

but that's more in the implementation rather than the mathematics of lighting effects.

**Two types of phenomena  
are important in lighting  
calculations:**

**The first is the interaction  
of light with the surface  
boundary**

and the second is the effect of light as it gets absorbed, transmitted, and scattered by interacting with the actual material itself

Materials are typically richer in their descriptions in an effort to mimic the effects seen in real life, light – Material surface interactions.



Materials are typically described using two to four separate colors in an effort to catch the nuances of real-world light-material surface interactions.

These colors are the ambient, diffuse, specular, and emissive colors, with ambient and specular frequently grouped together

and emissive specified only for objects that generate light the mselves.

The reason there are different colors is to give different effects arising from different environmental causes. The most common lights are discussed next.



## Ambient lighting

This is the color of the object when there's no particular light, just the general environmental illumination.

The ambient light is an approximation for the global illumination in the environment, and relies upon no light in the scene.

## Defuse Light

The diffuse light is the light of the surface if the surface were perfectly matte.

The diffuse light is reflected in all directions from the surface and depends only on the angle of the light to the surface normal.

## Specular Lighting

The specular light mimics the shininess of a surface, and its intensity is a function of the light's reflection angle off the surface.

# Emissive Light

When we need an object to "glow" in a scene, we can do this with an emissive light.

This is just an additional color source added to the final light of the object.

Because we are simulating an object giving off its own light; we still add a real "light" to get an effect on objects in a scene.

Before we get into exactly what these types of lighting are, let's put it in perspective for our purpose of writing shader code.

Shading is simply calculating the color reflected off a surface (which is pretty much what shaders do).

When a light reflects off a surface, the light colors are modulated by the surface color (typically, the diffuse or ambient surface color).



Modulation means multiplication, and for colors, since we are using rgb values, this means component-by-component multiplication.

So for light source '*l*' with color (*r<sub>l</sub>*,*g<sub>l</sub>*,*b<sub>l</sub>*) shining on surface '*s*' with color (*r<sub>s</sub>*,*g<sub>s</sub>*,*b<sub>s</sub>*), the resulting color *r* would be:

$$a = l \otimes s$$

or, multiplying it out, we get

$$r = [r r_s, g g_s, b b_s]$$

Where the resulting rgb values of the light and surface are multiplied out to get the final color's rgb values.

The final step after calculating all the lighting contributions is to add together all the lights to get the final color.

So a shader might typically do the following:

1. Calculate the overall ambient light on a surface.
2. For each light in a scene, calculate the diffuse and specular contribution for each light.

3. Calculate any emissive light for a surface.
4. Add all these lights together to calculate the final color value.

In the real world, we get some sort of interaction (reflection, etc.) when a photon interacts with a surface boundary

Thus we see the effects not only when we have a transparent—opaque boundary (like air-plastic), but also a transparent—transparent boundary (like air-water)

The key feature here is that we get some visual effect when a photon interacts with some boundary between two different materials.

The conductivity of the materials directly affects how the photon is reflected. At the surface of a conductor (metals, etc.), the light is mostly reflected.

For dielectrics (that is nonconductors), there is usually more penetration and transmittance of the light. For both kinds of materials, the dispersion of the light is a function of the roughness of the surface...

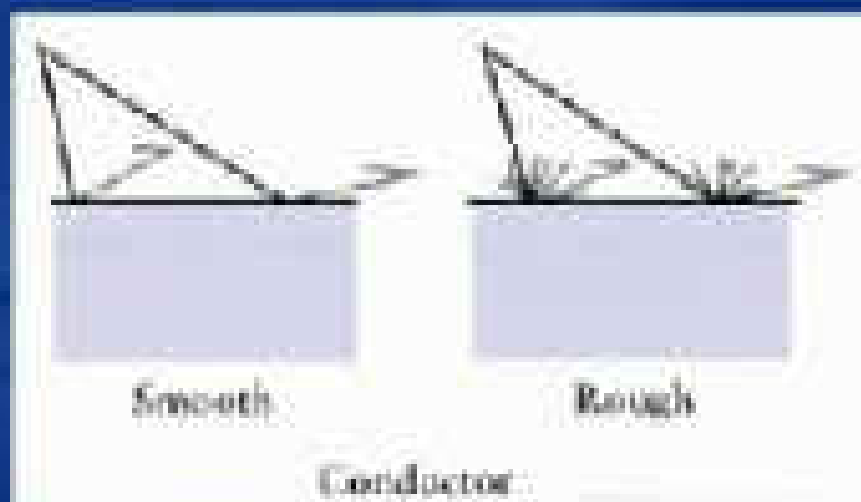


**Figure 1: Light reflecting from a rough and smooth surface of a conductor.**

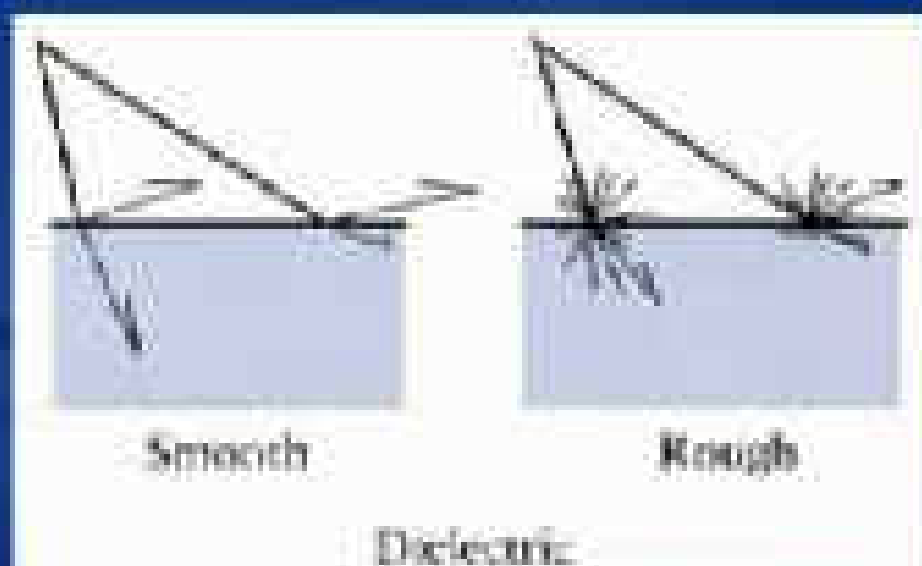


**Figure 2: Light reflecting from a rough and smooth surface of a dielectric showing some penetration.**

The simplest model assumes that the roughness of the surface is so fine that light is dispersed equally in all directions as it was shown in Figure 1

Consider a thick colored lacquer surface. The lacquer itself is transparent, but suspended in the lacquer are reflective pigment off of which light gets reflected, bounced, split, shifted or altered before perhaps reemerging from the surface.





Figure 3: Subsurface scattering typical of pigment-saturated translucent coatings.

Metallic paint, brushed metal, velvet, etc. are all materials for which we need to examine better models to try to represent these surfaces.

Figure 4 shows what you get when you use multiple broad specular terms for multiple base colors combined with a more traditional shiny specular term

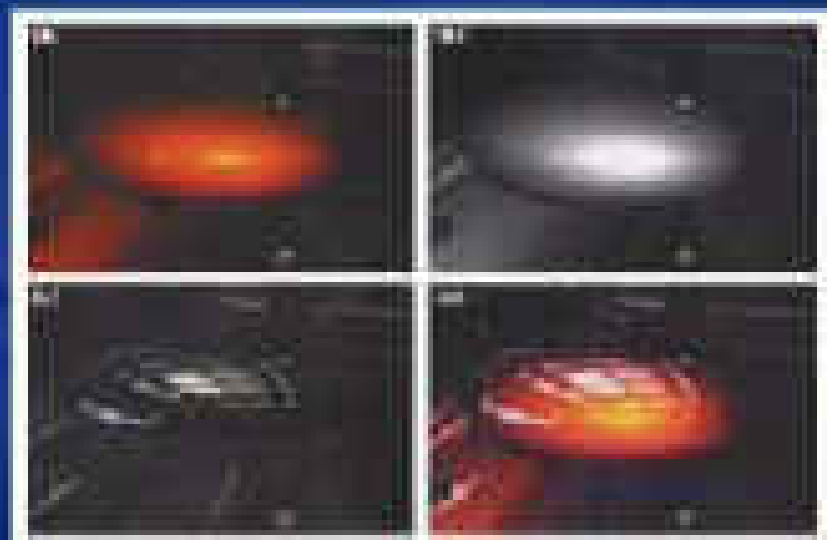


Figure 4: A simple shader to simulate metallic paint:

There's also a high-frequency normal perturbation that simulates the sparkle from a metallic flake pigment. As we can see, we can get something that looks particularly striking with a fairly simple model.

4 colors used to describe object surfaces.

- Ambient
- Diffuse
- Specular
- emissive

1. Calculate the overall ambient light on a surface.
2. For each light in a scene, calculate the diffuse and specular contribution for each light.

- The diffuse light is reflected in all directions from the surface and depends only on the angle of the light to the surface normal.

# Computer Graphics

## Lecture 26

### Computer Graphics

### Lecture 26

### Mathematics of Lighting and Shading

### Part II

Taqdees A. Siddiqi  
CS602@vu.edu.pk

# Light Types and Shading Models

## Light Types

Now that we have a way to find the light hitting a surface, we're going to need some lights!

There are three types of lights we are going to discuss

- 1) Parallel Lights (or Directional Lights)
- 2) Point Lights
- 3) Spotlights

Parallel Lights (or  
Directional Lights)



They represent light that comes from an infinitely far away light source.

All of the light rays that reach the object are parallel (hence the name).

The standard use of parallel lights is to simulate the sun.

While it's not infinitely far away, 93 million miles is good enough!

The incoming light vector for calculation of the diffuse reflection factor is the same for all considered points,

whereas point lights and spotlights involve vector subtractions and a normalization per vertex.

Parallel light sources are the easiest and therefore fastest to process.

# Point Lights

One step better than directional lights are point lights.

They represent infinitesimally small points that emit light.

Light scatters out equally in all directions.

we can have the intensity falloff based on the inverse squared distance from the light, which is how real lights work.

$$\text{attenuation\_factor} = \frac{k}{|\text{surface\_location} - \text{light\_location}|^2}$$

The light direction is different for each surface location (otherwise the point light would look just like a directional light). The equation for it is:

$$\text{light\_direction} = \frac{\text{surface\_location} - \text{light\_location}}{|\text{surface\_location} - \text{light\_location}|}$$

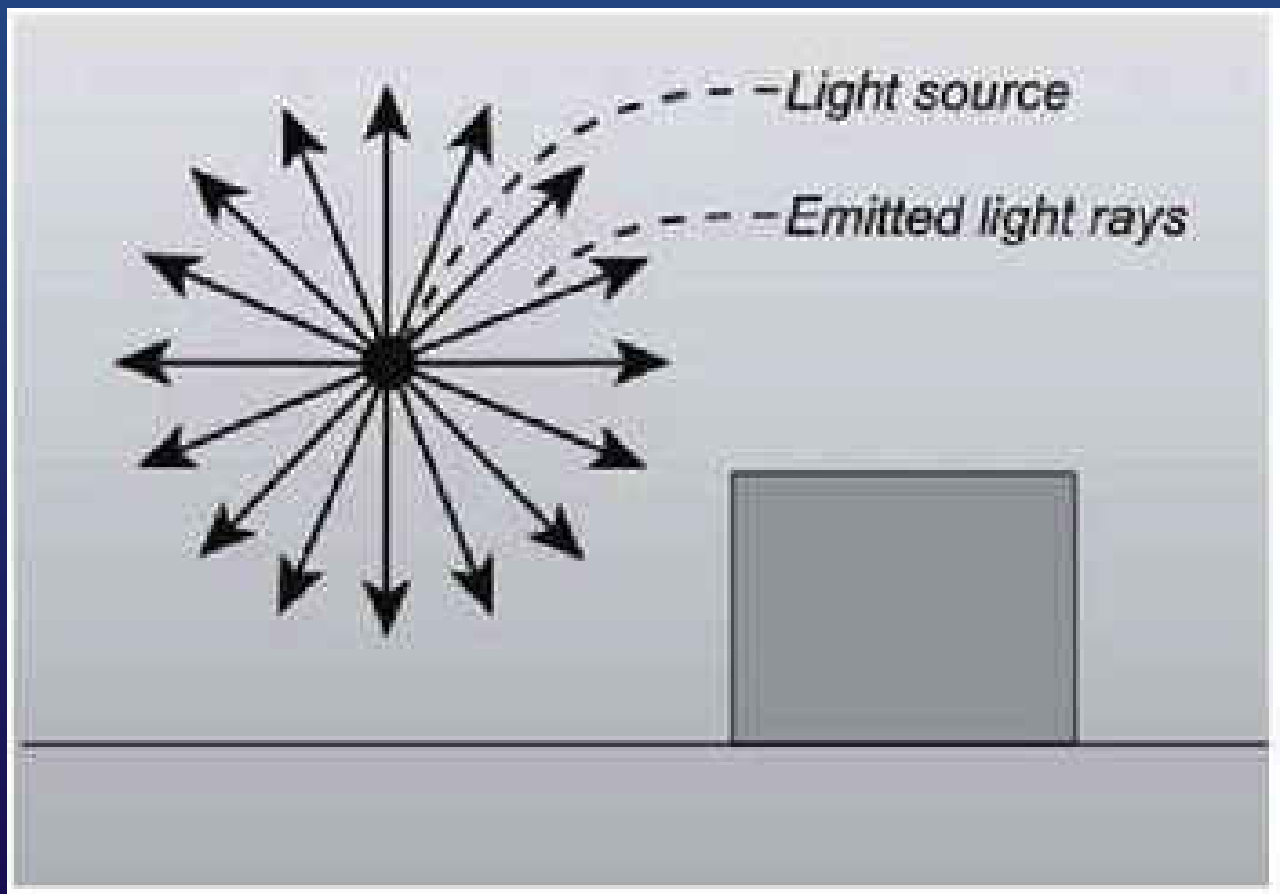


**Figure 1: Point light sources**

Spotlights



Spotlights are the most expensive type of light.

It is normally avoided to use because it is not for real time environment.

Spotlight is the type type we would see in a theatrical production.

They are point lights, but light only leaves the point in a particular direction, spreading out based on the aperture of the light.

Spotlights have two angles associated with them.

One is the internal cone whose angle is generally referred to as theta ( $\theta$ ).

Points within the internal cone receive all of the light of the spotlight; the attenuation is the same as it would be if point lights were used.

There is also an angle that defines the outer cone; the angle is referred to as  $\phi$ .

Points outside the outer cone receive no light.

Points outside the inner cone but inside the outer cone receive light,

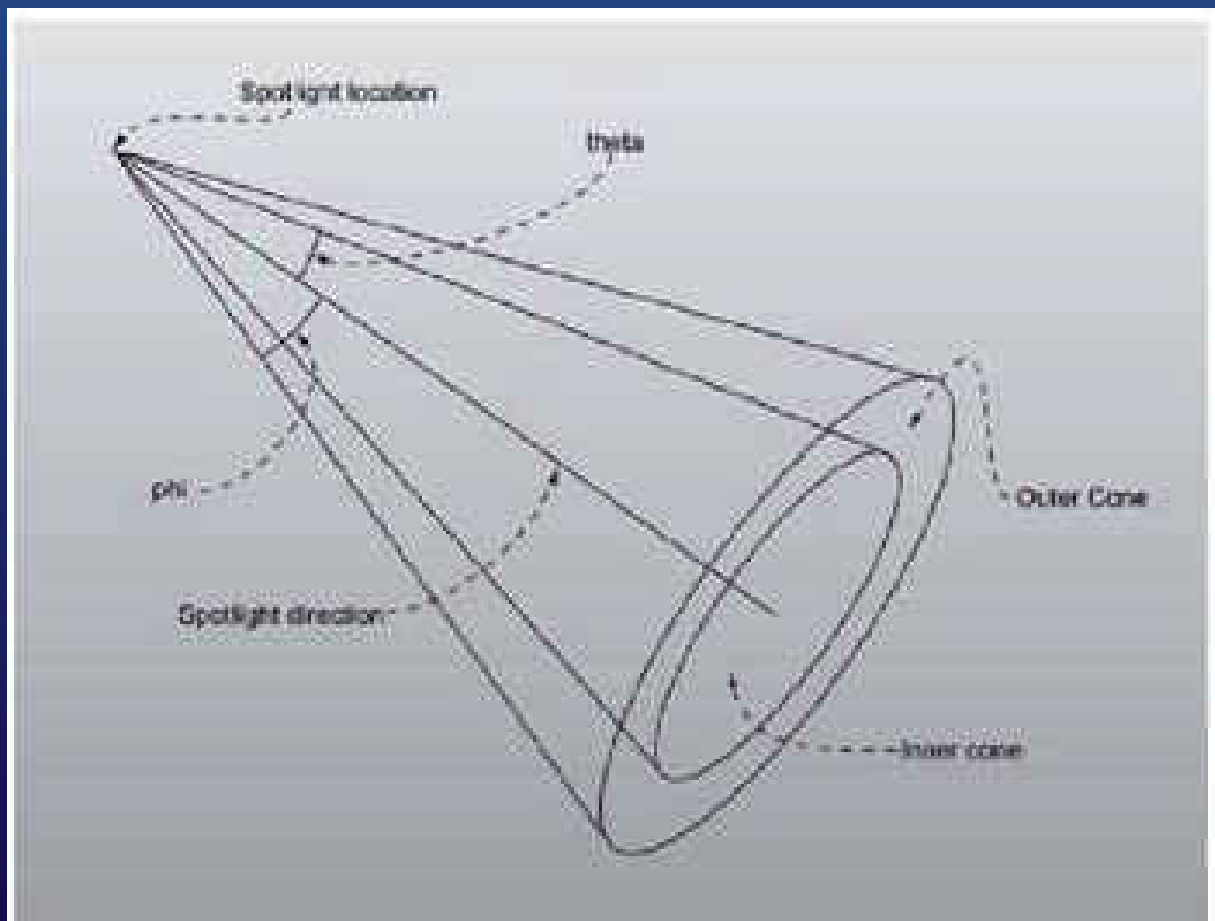


**Figure 2: A spotlight**

OpenGL and DirectX Implements the lighting math behind the scene

Lighting is extremely expensive and can slow down our graphics application a great deal.

So we will have to figure out a line between performance and aesthetics.

## Shading Models

Once we've found basic lighting information, we need to know how to draw the triangles with the supplied information. There are currently three ways to do this;

- 1) Lambertian Shading
- 2) Gouraud Shading
- 3) Phong Shading



# Lambertian Shading

Triangles that use Lambertian shading are painted with one solid color instead of using a gradient.

Typically each triangle is lit using that triangle's normal.  
The resulting object looks very angular and sharp.

Lambertian shading was used mostly back when computers weren't fast enough to do Gouraud shading in real time.

To light a triangle, you compute the lighting equations using the triangle's normal and any of the three vertices of the triangle.



**Figure 3:** Flat shaded view of our polygon mesh

# Gouraud Shading

Gouraud (pronounced *garrow*) shading is the current de facto shading standard in accelerated 3D hardware.

Instead of specifying one color to use for the entire triangle, each vertex has its own separate color.

The color values are linearly interpolated across the triangle, creating a smooth transition between the vertex color values.

To calculate the lighting for a vertex, we use the position of the vertex and a vertex normal.



Figure 4: Gouraud shaded view of our polygon mesh

One problem with Gouraud shading is that the triangles' intensities can never be greater than the intensities at the edges.

So if there is a spotlight shining directly into the center of a large triangle, Gouraud shading will interpolate the intensities at the three dark corners, resulting in an incorrectly dark triangle.

# Phong Shading

Phong shading is the most realistic shading model We are going to talk about, and also the most computationally expensive.



It tries to solve several problems that arise when we use Gouraud shading.

First of all, Gouraud shading uses a linear gradient.

Many objects in real life have sharp highlights, such as the shiny spot on an apple.

The Phong does by interpolating the normal across the triangle face, not the color value, and the lighting equation is solved individually for each pixel.



**Figure 5:** Phong shaded view of a polygon mesh

Phong Shading is not technically supported in hardware but we can program and many other special effects, using shaders, a hot new technology.

## Computer Graphics Lecture 26

# **Computer Graphics Lecture 27**

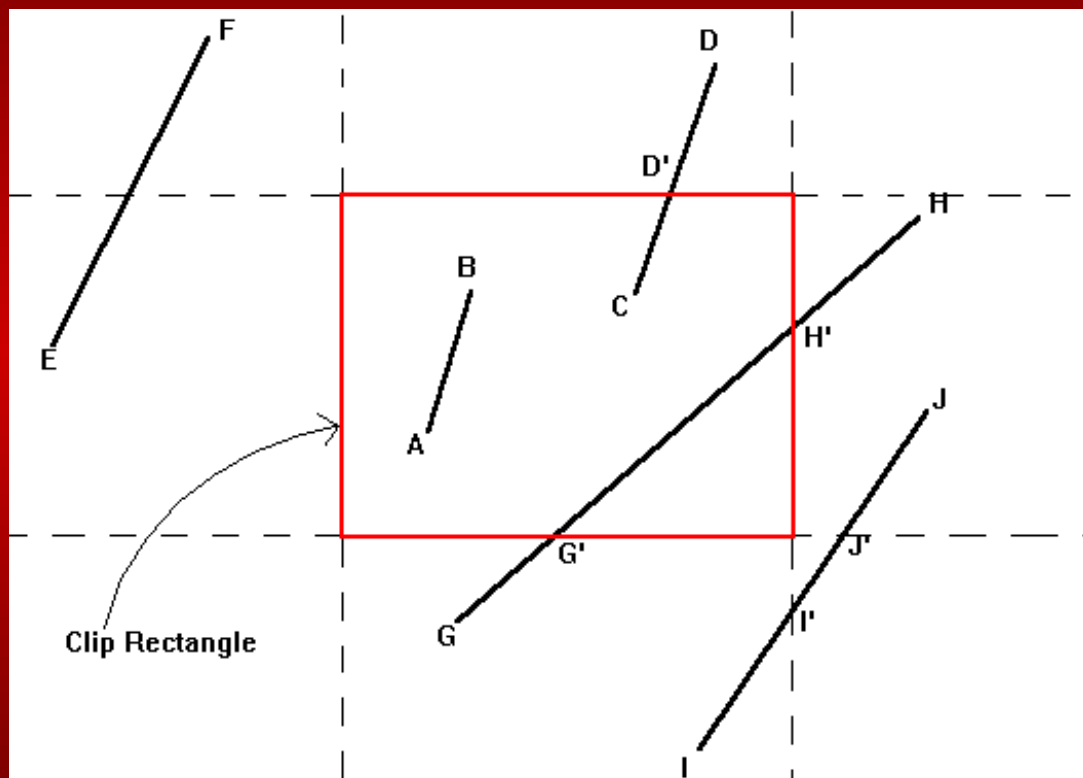
## **REVIEW II**

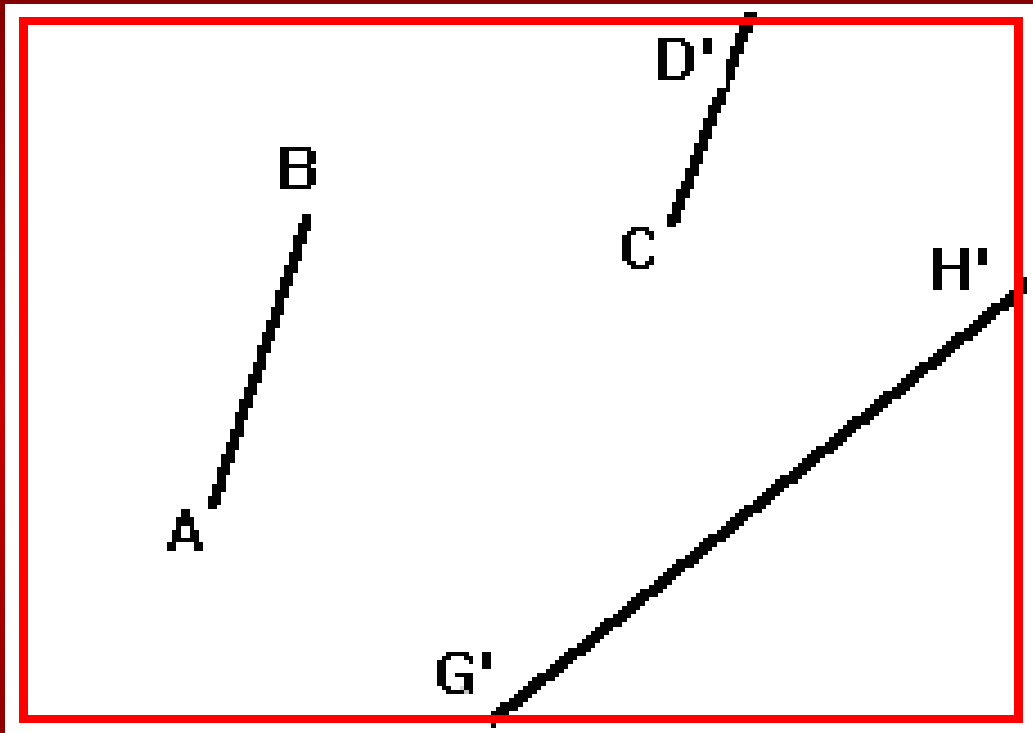
**Taqdees A. Siddiqi**  
**cs602@vu.edu.pk**

# CLIPPING

Restricting the effect of graphics primitives to a sub-region of the canvas

Protecting other portions of the canvas





# Point Clipping

$$x_{\min} \leq x \leq x_{\max}$$

$$y_{\min} \leq y \leq y_{\max}$$

# Line Clipping

Clipping of lines against rectangles:

- ◆ Trivial Accept
- ◆ Trivial Reject
- ◆ Intersecting Lines

## The Brute Force Approach

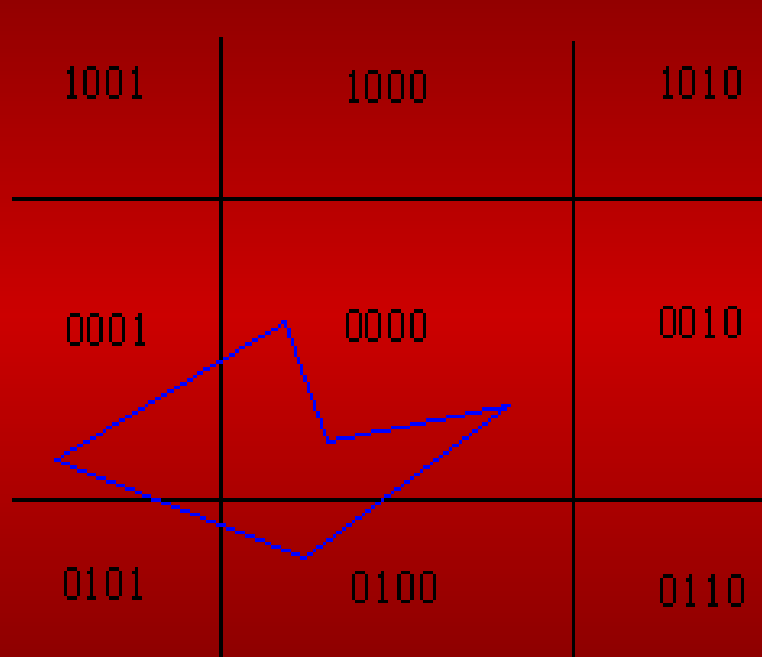
Intersect that line with each of the four clip rectangle edges. if so, the line cuts the clip rectangle and is partially inside.

# Cohen Sutherland

## Line Clipping Algorithm

More efficient;  
Performs initial tests on a line to determine whether intersection calculations can be avoided.

## Trivial Accept / Reject Test





# **Liang-Barsky Algorithm**

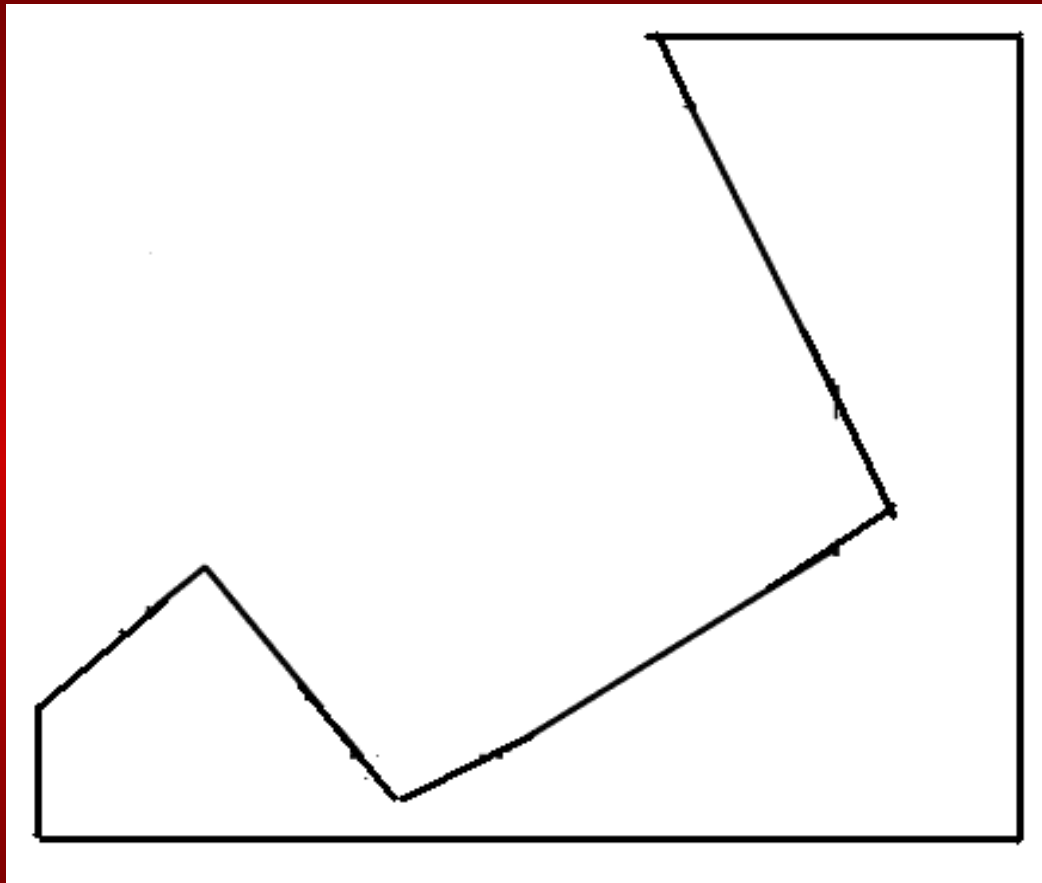
**Faster line clippers  
based on analysis of  
the parametric  
equation of a line  
segment:**

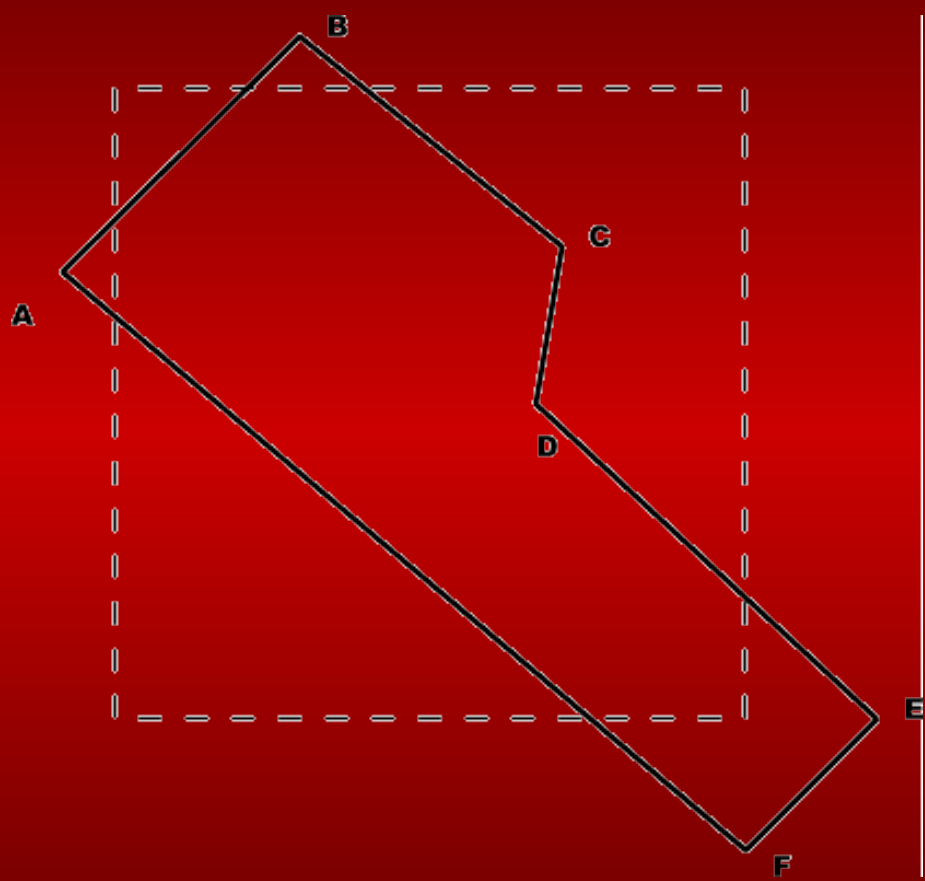
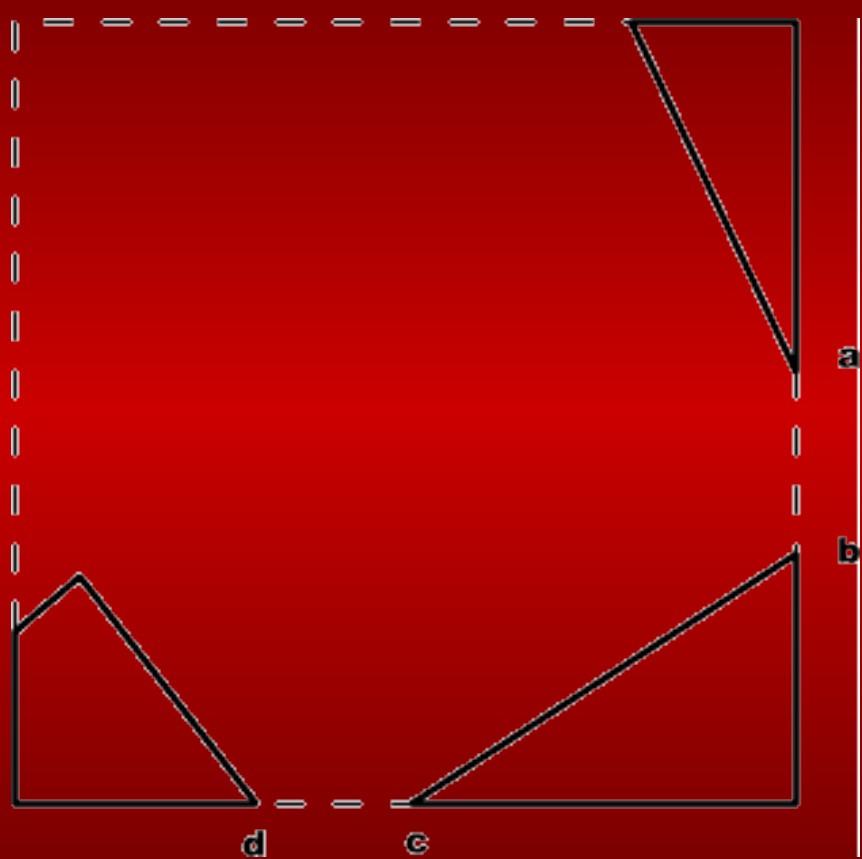
## **Conclusion**

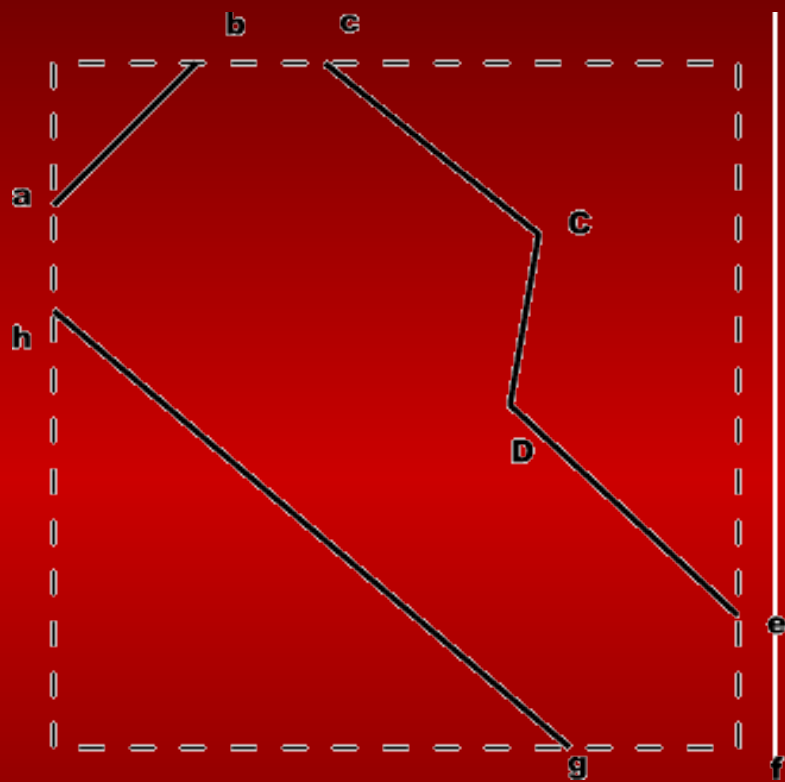
**More efficient;  
intersection calculations  
are reduced.**

# Polygon Clipping

A polygon is usually defined by a sequence of vertices and edges. If the polygons are un-filled, line-clipping techniques are sufficient however, if the polygons are filled, the process is more complicated.

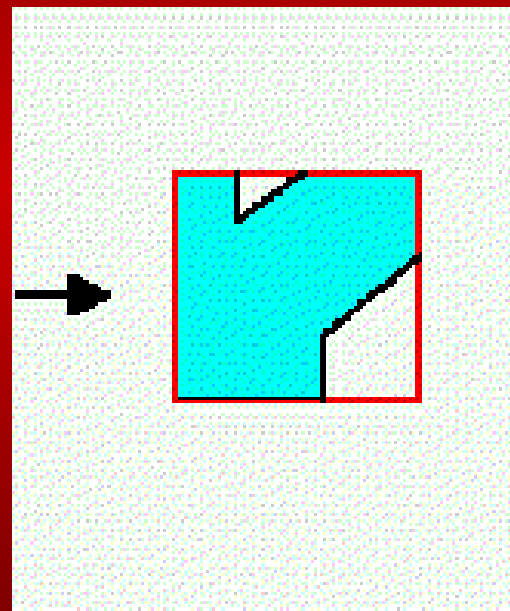
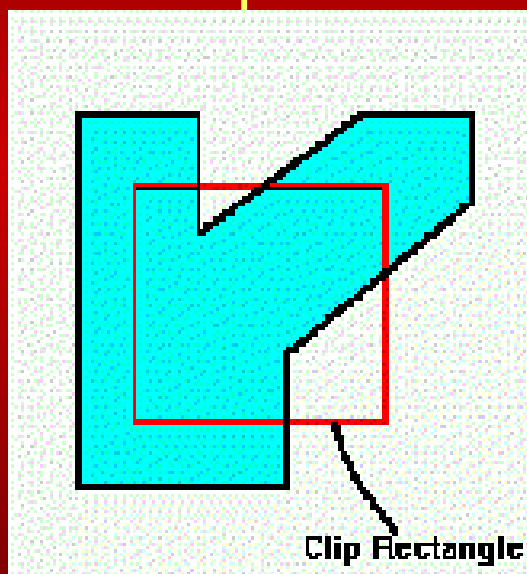






Polygon clipping - open polygons.

## Example



# **Sutherland-Hodgeman clipping algorithm**

**clips any polygon against a  
convex clip polygon.**

# **Weiler-Atherton clipping algorithm**

**Weiler-Atherton clipping  
algorithm clips any polygon  
against any clip polygon. The  
polygons may even have  
holes.**

# **Four Cases of Polygon Clipping Against One Edge**

**Case 1 : Wholly inside visible region save endpoint**

**Case 2 : Exit visible region - save the intersection point**

**Case 3 : Wholly outside visible region - save nothing**

**Case 4 : Enter visible region - save intersection and endpoint**

# **Weiler-Atherton Polygon Clipping**

**Sometimes follow the window boundaries instead of polygon boundary.**

## **3-Dimensions**

**In a 3D medium, each of our eyes views the scene from slightly different angles. This is the way we perceive the real world.**

# Coordinate Systems

Coordinate systems are the measured frames of reference within which geometry is defined, manipulated and viewed.

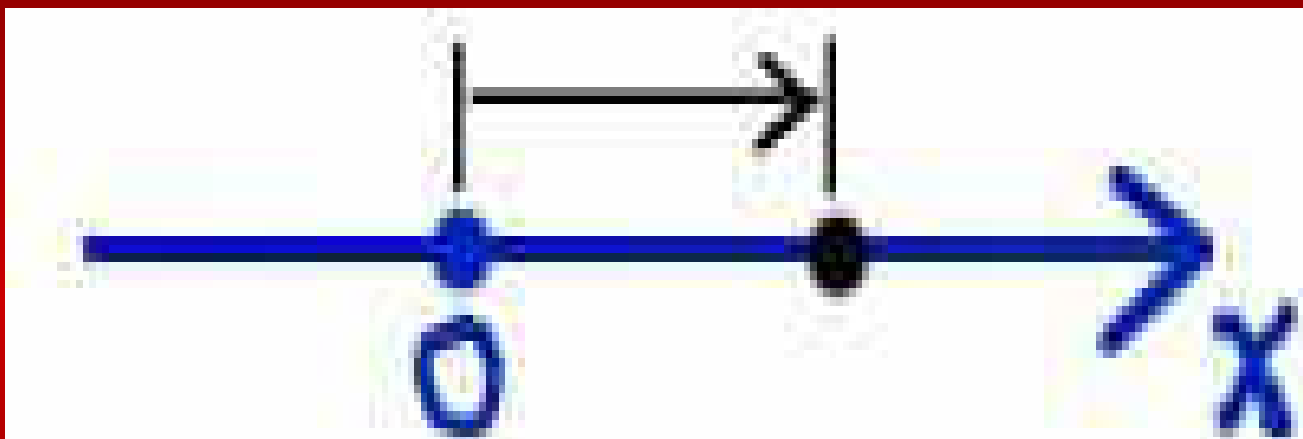
## Coordinate Systems

In this system, a point serves as the origin (reference point), and three lines (axes) that pass through this point and are orthogonal to each other (at right angles – 90 degrees).

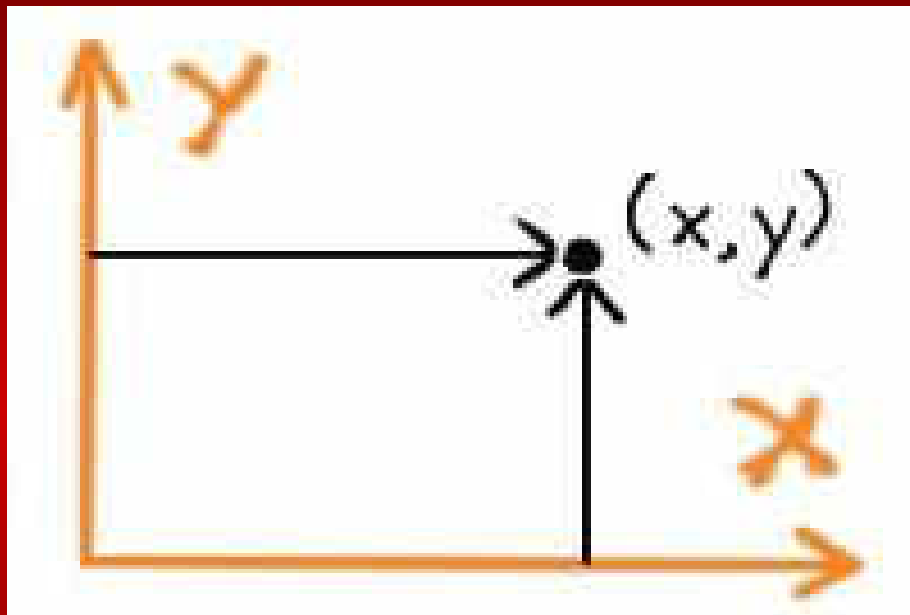


# Types of the coordinate systems

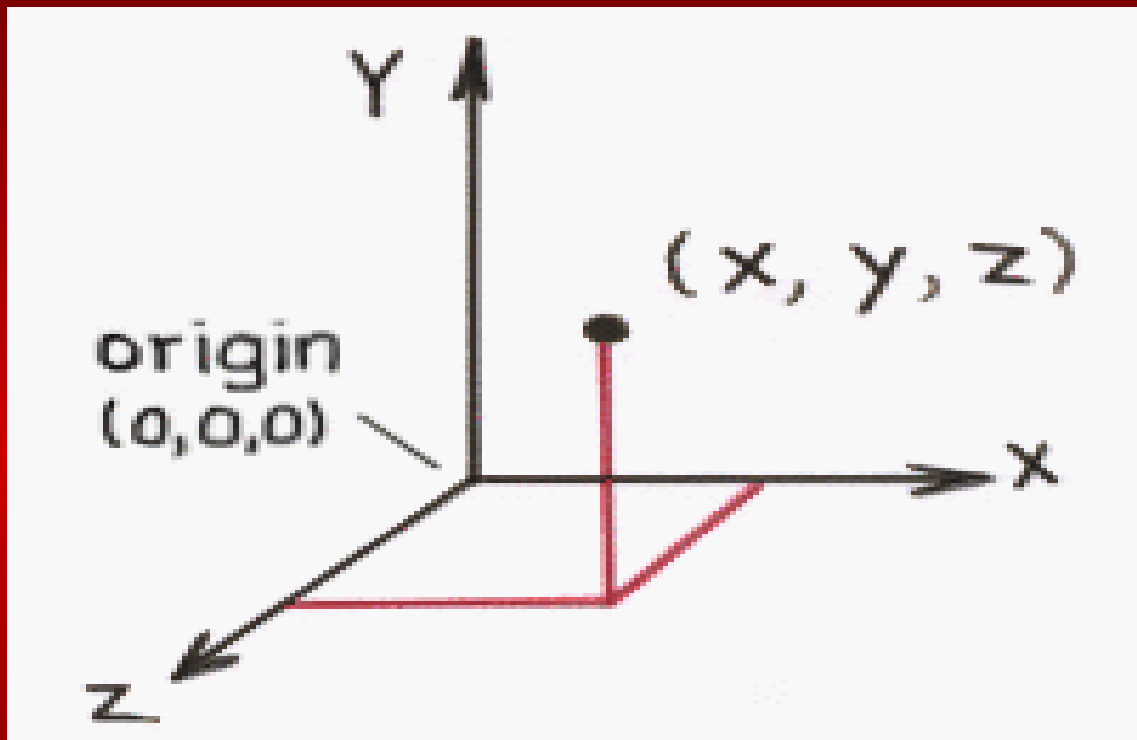
- ◆ Following are three types of the coordinate systems
  - 1-D Coordinate System
  - 2-D Coordinate System
  - 3-D Coordinate System



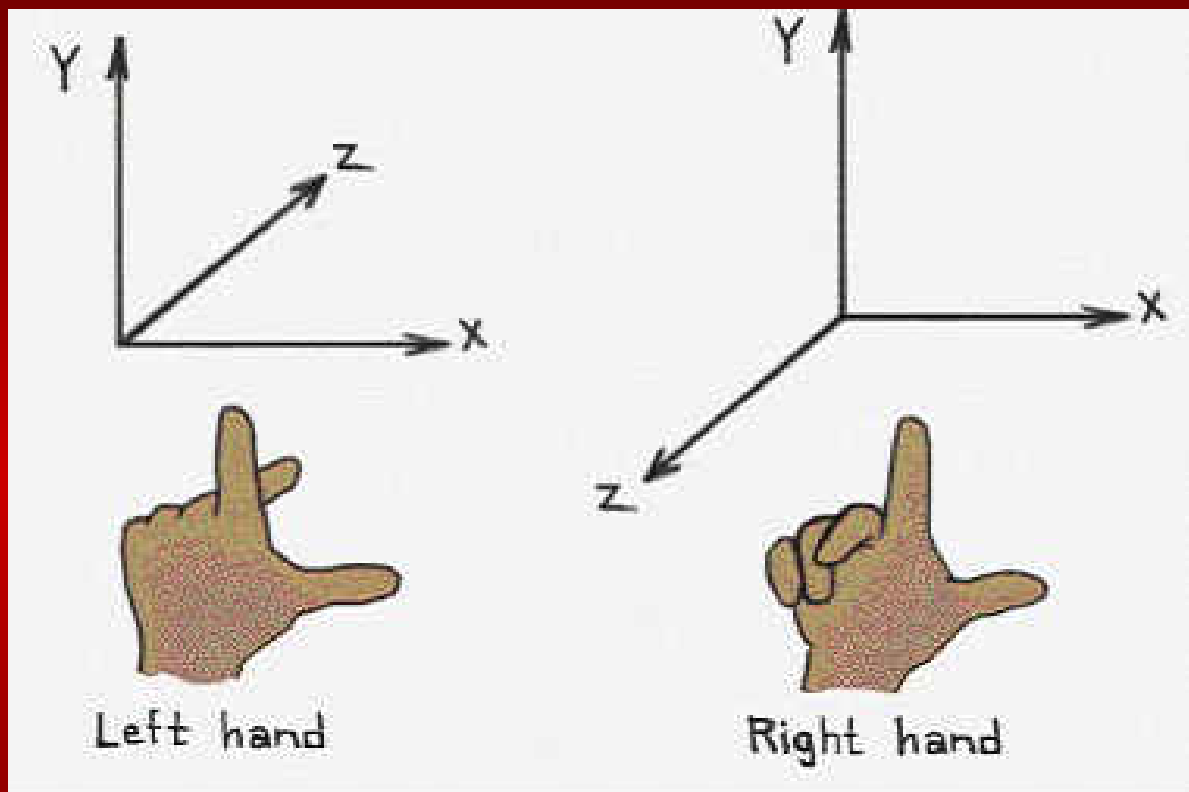
**1-D Coordinate System**



**2-D Coordinate System**



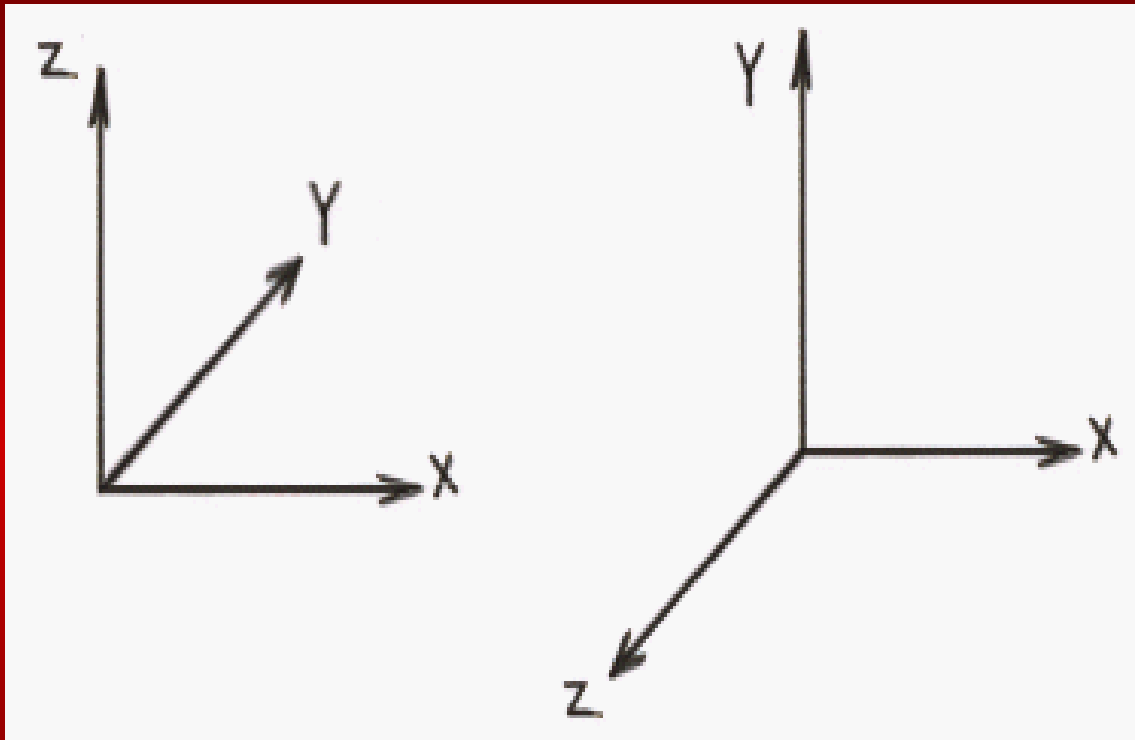
**3-D Coordinate System**



## Left handed versus Right Handed Rules

**“Right Hand Rule”**  
for rotations: grasp  
axis with right hand  
with thumb oriented  
in positive direction,  
fingers will then curl  
in direction of  
positive rotation for  
that axis.





## Y-up versus Z-up

# The Polar Coordinate System

Is Cartesian system the only choice? No. We could also describe the object position in this way: “starting at the origin, looking east, rotate  $38^\circ$  northward,  $65^\circ$  upward, and travel 7.47 feet along this line”.

# Modeling

- ◆ **Modeling is the process of describing an object or scene so that we can construct an image of it**

## Vectors

- **Vectors have direction and magnitude; can also be thought of as displacement**

# Rendering

**The process of computing a 2-D image using a combination of a 3-D database, scene characteristics, and viewing transformations.**

**Various algorithms according to the needs of the application.**

# Tessellation

- ◆ **The subdivision of an entity or surface into one or more non-overlapping primitives.**

**Typically, renderers decompose surfaces into triangles as part of the rendering process.**

# Level of Detail (LOD)

To improve rendering efficiency when dynamically viewing a scene, more or less detailed versions of a model may be swapped in and out of the scene database depending on the importance (usually determined by image size) of the object in the current view.

## Surface normal

- ◆ A vector perpendicular to a surface and “outward” facing
- ◆ Surface normal are used to determine visibility and also in the calculation of shading values

# Types of Transformation

- ◆ Translation
- ◆ Rotation
- ◆ Scaling
- ◆ Reflection
- ◆ Shearing

Now  $x' = x + t_x$  ,  $y' = y + t_y$

and  $z' = z + t_z$  can be expressed as a single matrix equation:

$$P' = P + T$$

Where  $P =$

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$=$

$$T \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix}$$

$$\begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$$



# Homogeneous Coordinates

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Abbreviated as:

$$P' = T(t_x, t_y, t_z) \cdot P$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ z + t_z \\ 1 \end{bmatrix}$$

# Rotation

**Rotation is the process of moving a point in space in a non-linear manner**

- ◆ **We need to know three different angles:**
- ◆ **How far to rotate around the X axis (YZ rotation, or “pitch”)**
- ◆ **How far to rotate around the Y axis (XZ rotation, or “yaw”)**
- ◆ **How far to rotate around the Z axis (XY rotation, or “roll”)**

- ◆ Column vector representation:

$$P' = R \cdot P$$

- ◆ Where

$$P' = \begin{bmatrix} x' \\ y' \end{bmatrix}$$

$$R = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}$$

$$P = \begin{bmatrix} x \\ y \end{bmatrix}$$

## Rotation: Homogeneous Coordinates

The rotation can now be expressed using homogeneous coordinates as:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

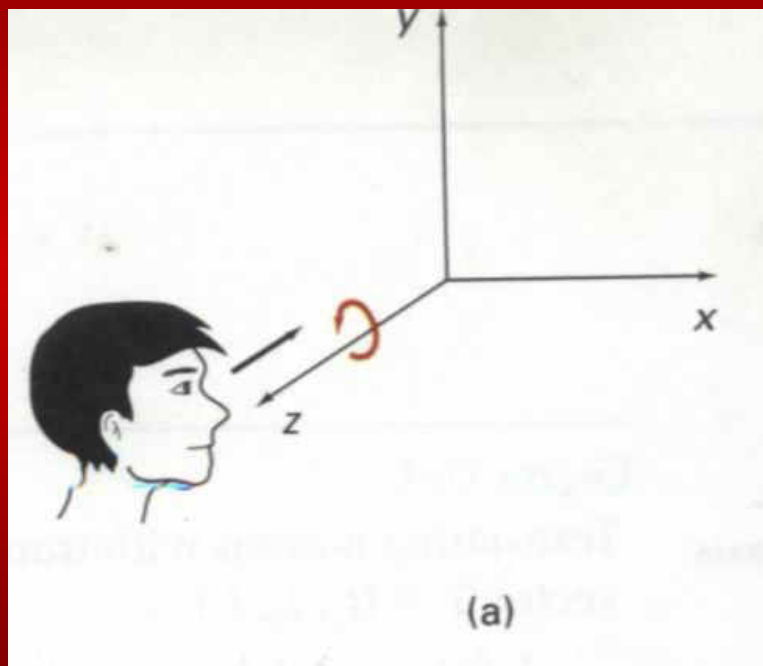
Abbreviated as:

$$P' = R(\theta) \cdot P$$

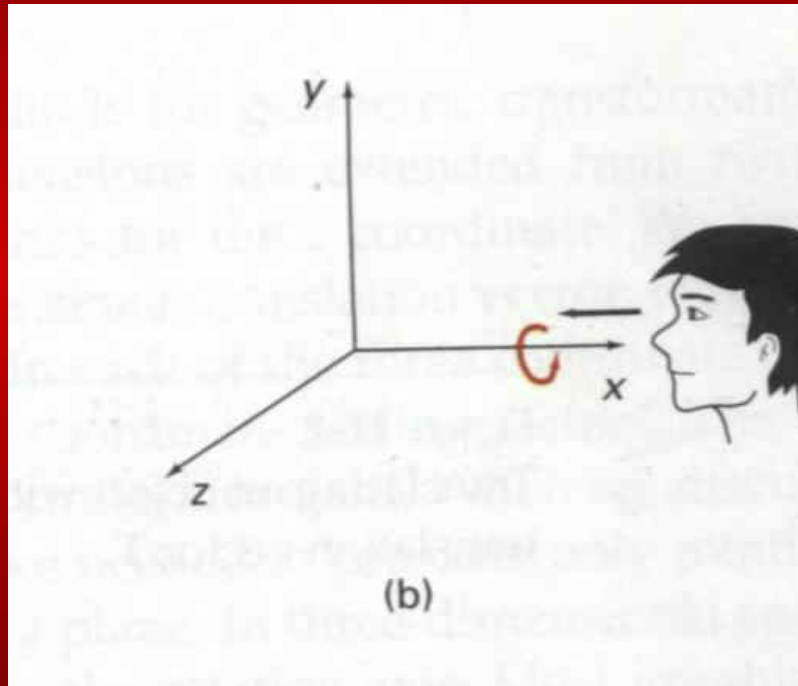
## ... Now in 3D

- ♦ Rotation can be about any of the three axes:
- ♦ About z-axis (i.e. in xy plane)
- ♦ About x-axis (i.e. in yz plane)
- ♦ About y-axis (i.e. in xz plane)

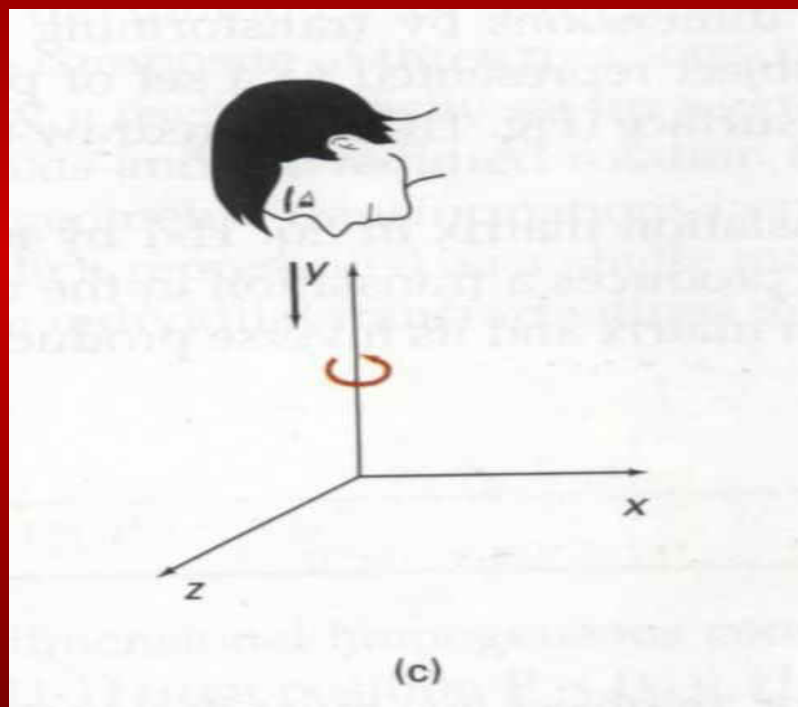
## Roll : around z-axis



# Pitch: around x-axis



# Yaw: around y-axis



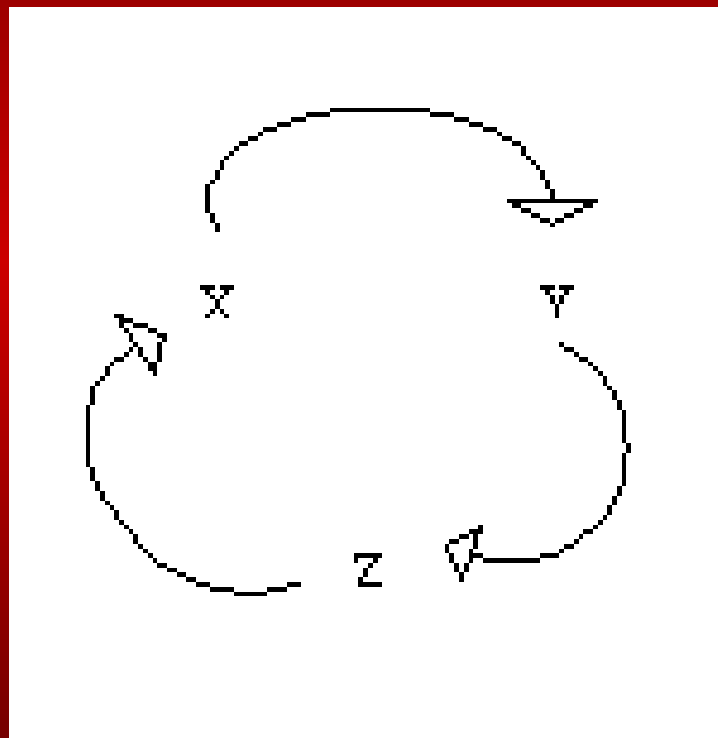
◆ **Rotation about z-axis**  
(i.e. in xy plane):

- $x' = x \cos\theta - y \sin\theta$

- $y' = x \sin\theta + y \cos\theta$

- $z' = z$

## Cyclic permutation



**Rotation about x-axis  
(i.e. in yz plane):**

$$x' = x$$

$$y' = y \cos\theta - z \sin\theta$$

$$z' = y \sin\theta + z \cos\theta$$

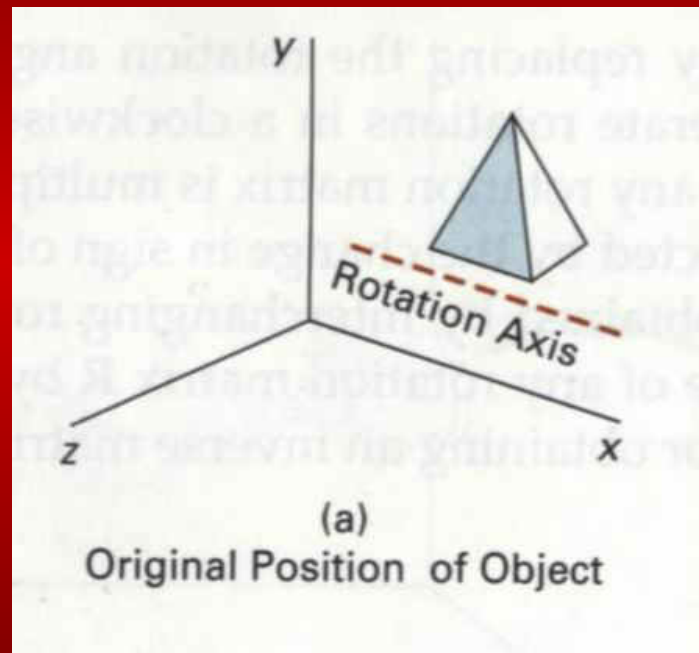
**Rotation about y-axis  
(i.e. in xz plane):**

$$x' = z \sin\theta + x \cos\theta$$

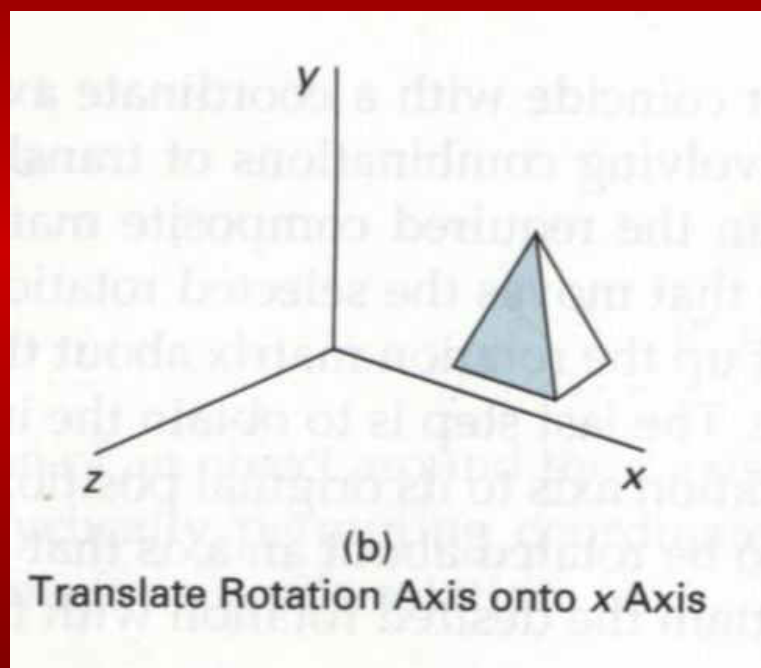
$$y' = y$$

$$z' = z \cos\theta - x \sin\theta$$

# Around arbitrary axis

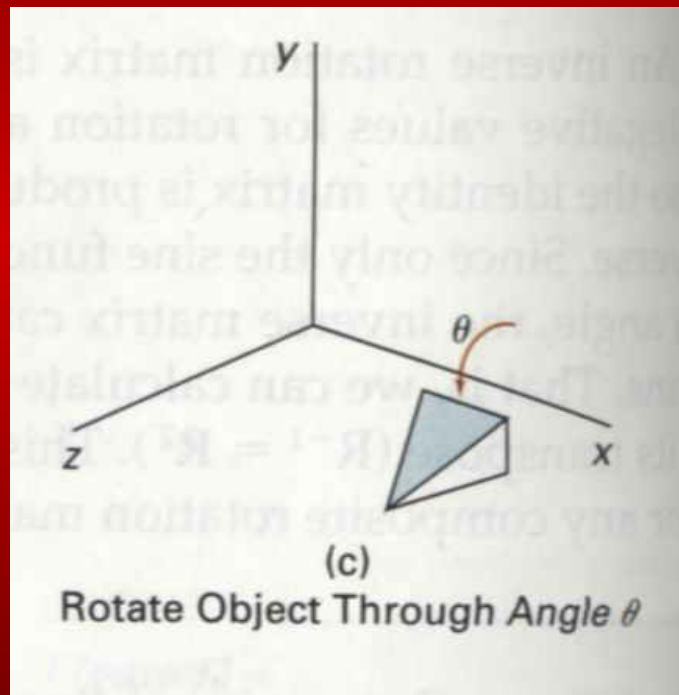


# Around arbitrary axis

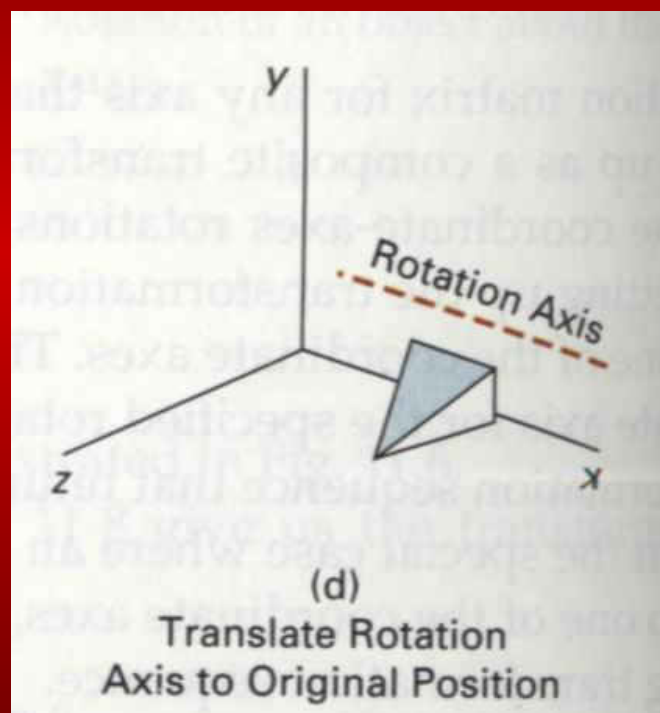




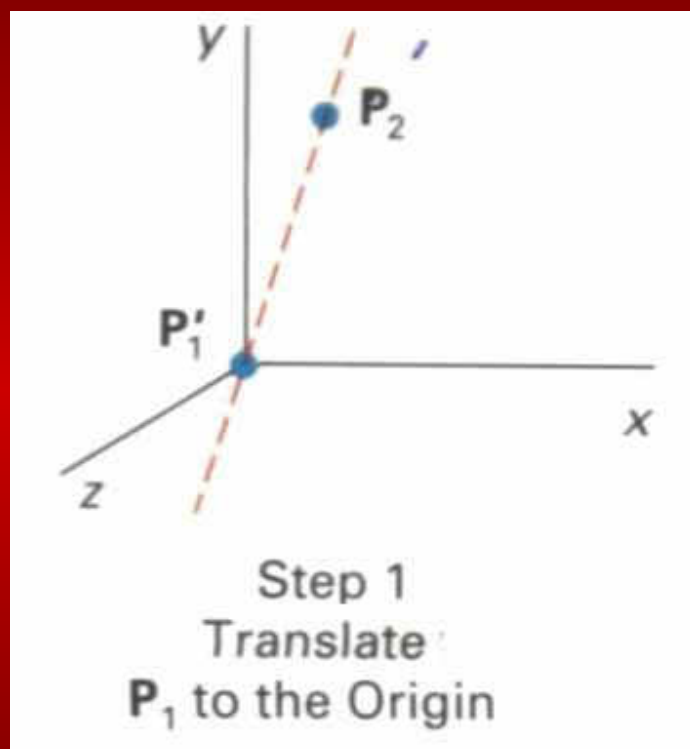
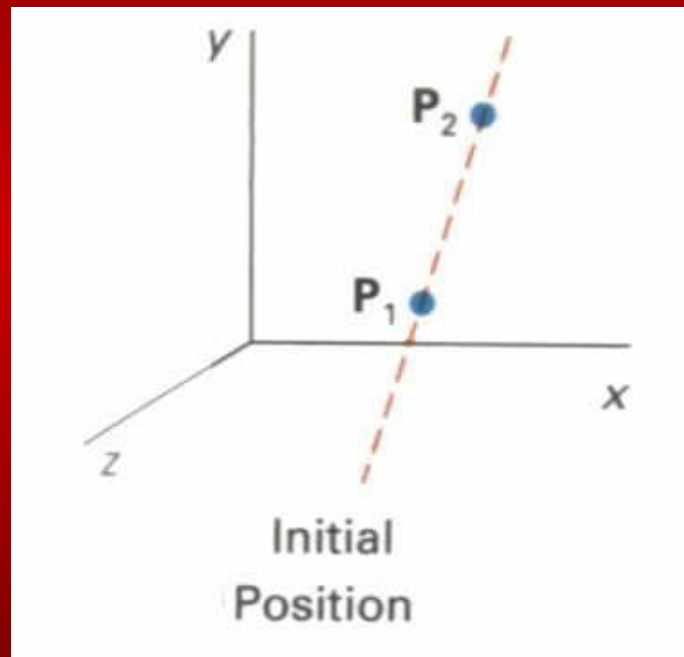
# Around arbitrary axis

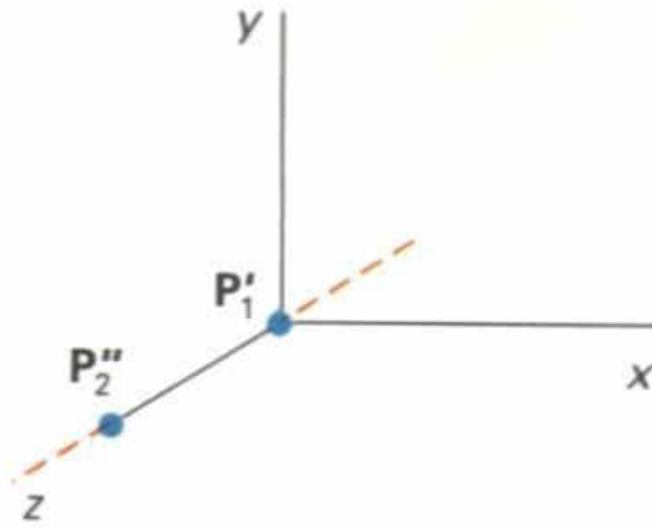


# Around arbitrary axis

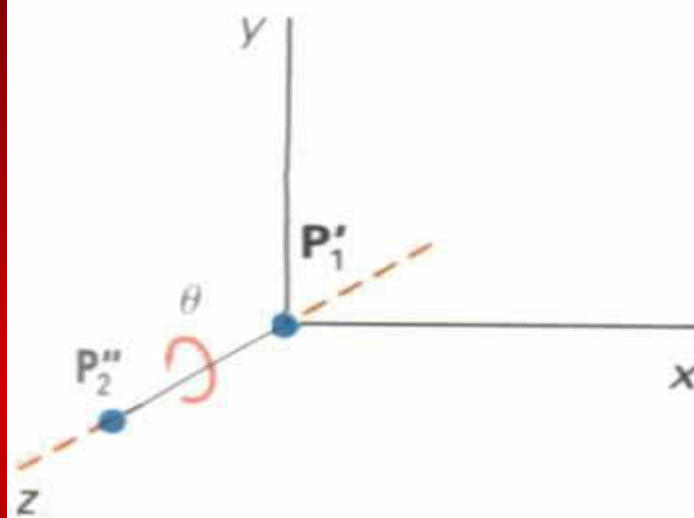


# Around arbitrary axis

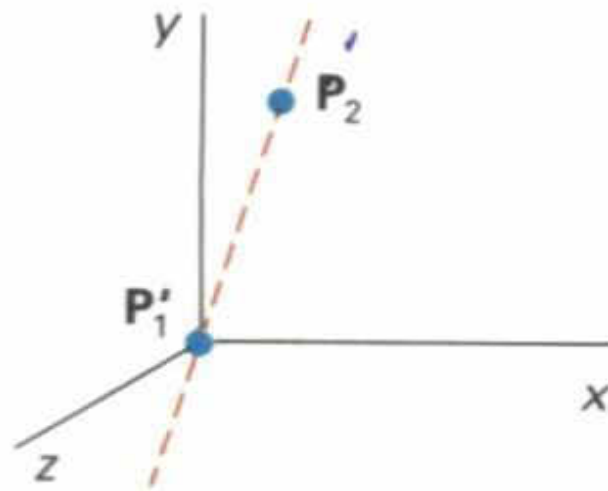




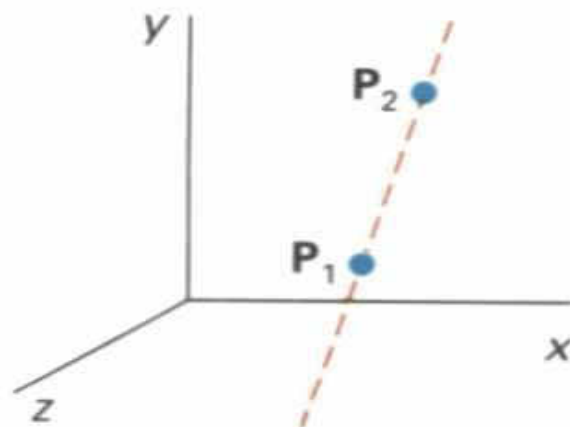
Step 2  
Rotate  $P'_2$   
onto the  $z$  Axis



Step 3  
Rotate the  
Object Around the  
 $z$  Axis



Step 4  
Rotate the Axis  
to the Original  
Orientation



Step 5  
Translate the  
Rotation Axis  
to the Original  
Position

- ◆ **Coordinate transformations for scaling relative to the origin are**
- ◆  **$X' = X \cdot S_x$**
- ◆  **$Y' = Y \cdot S_y$**
- ◆  **$Z' = Z \cdot S_z$**

## **Uniform Scaling**

- ◆ **We preserve the original shape of an object with a uniform scaling**
- ◆ **(  $S_x = S_y = S_z$  )**

# Differential Scaling

- ◆ We do not preserve the original shape of an object with a differential scaling
- ◆ (  $S_x \neq S_y \neq S_z$  )

## Scaling w.r.t. Origin

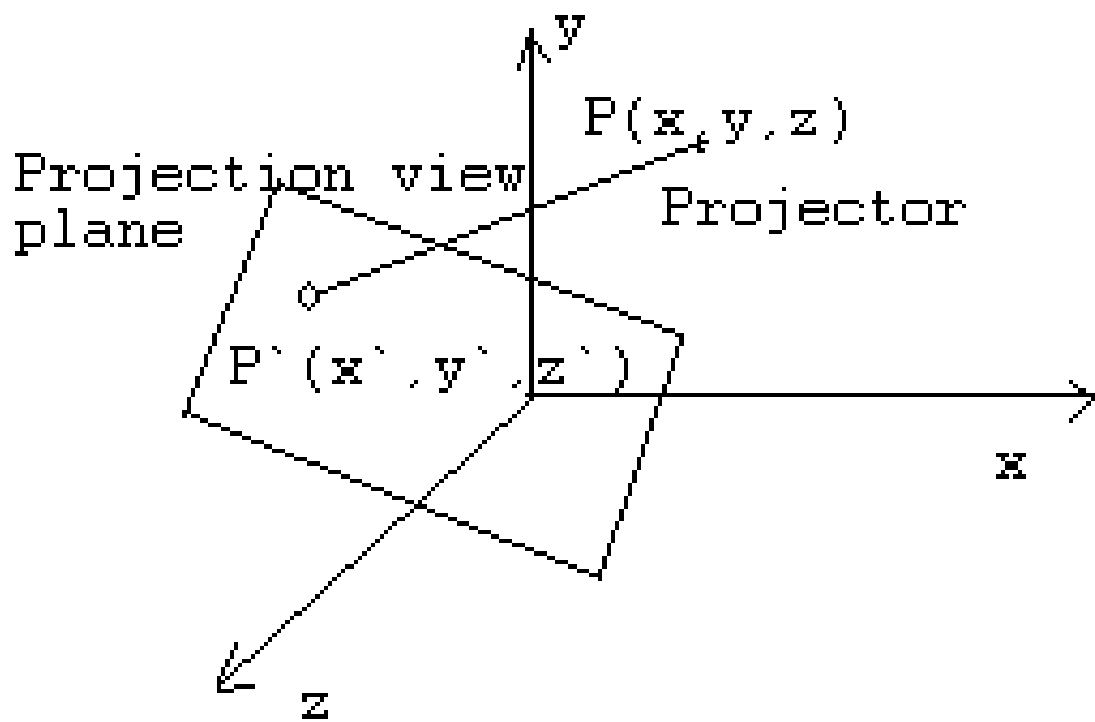
$$\begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

# Scaling w.r.t. a fixed position

Scaling with respect to a selected fixed position  $(X_f, Y_f, Z_f)$  can be represented with the transformation sequence:

## PROJECTION

Projection can be defined as a mapping of point  $P(x, y, z)$  onto its image  $P'(x', y', z')$  in the projection plane or view plane, which constitutes the display surface



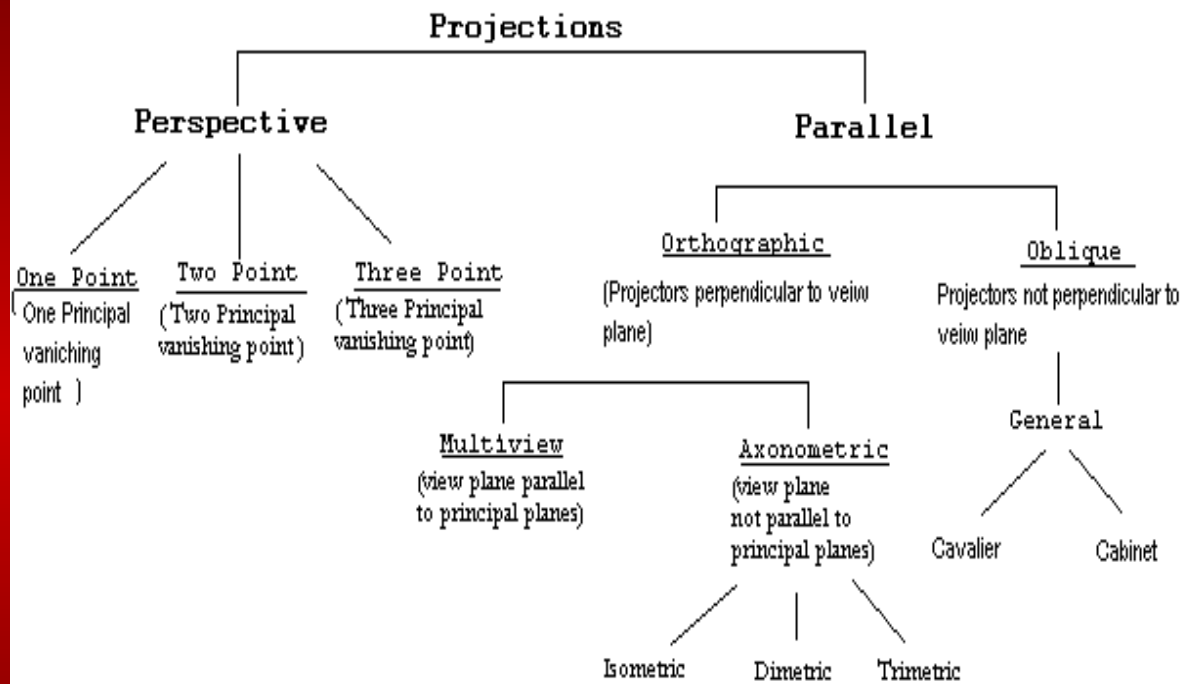
The Problem of Projection

## Methods of Projection

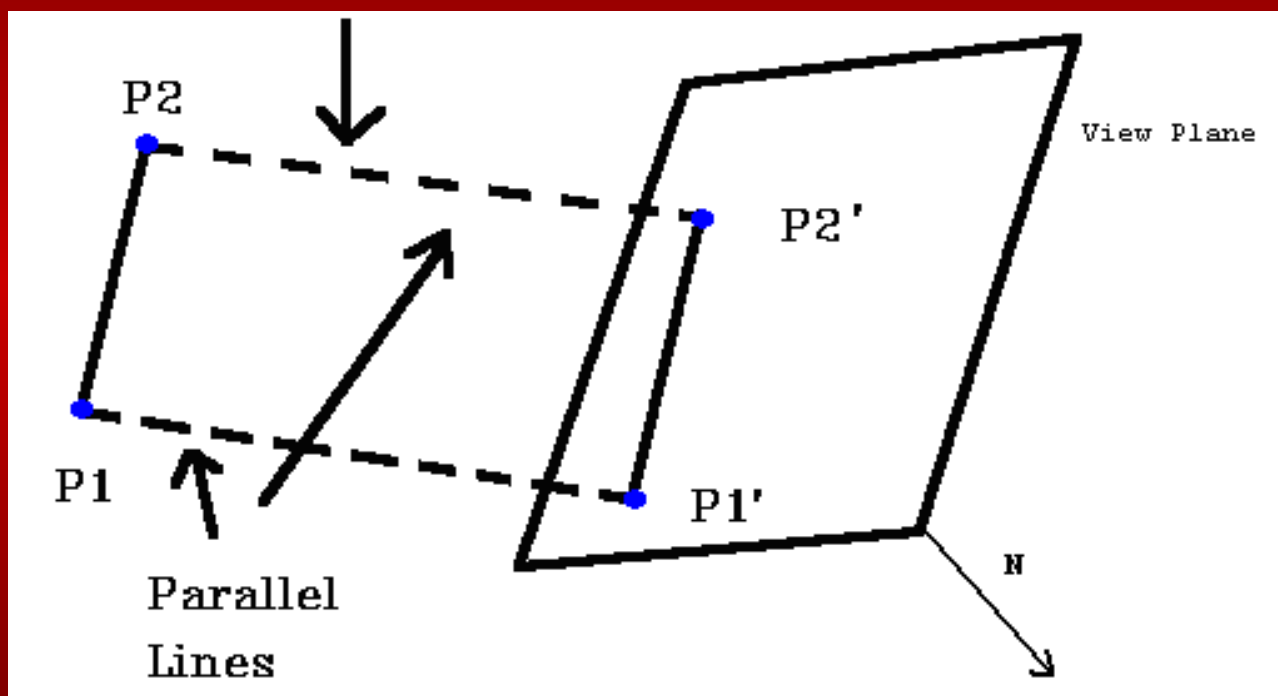
**1. Parallel Projection**

**2. Perspective Projection**

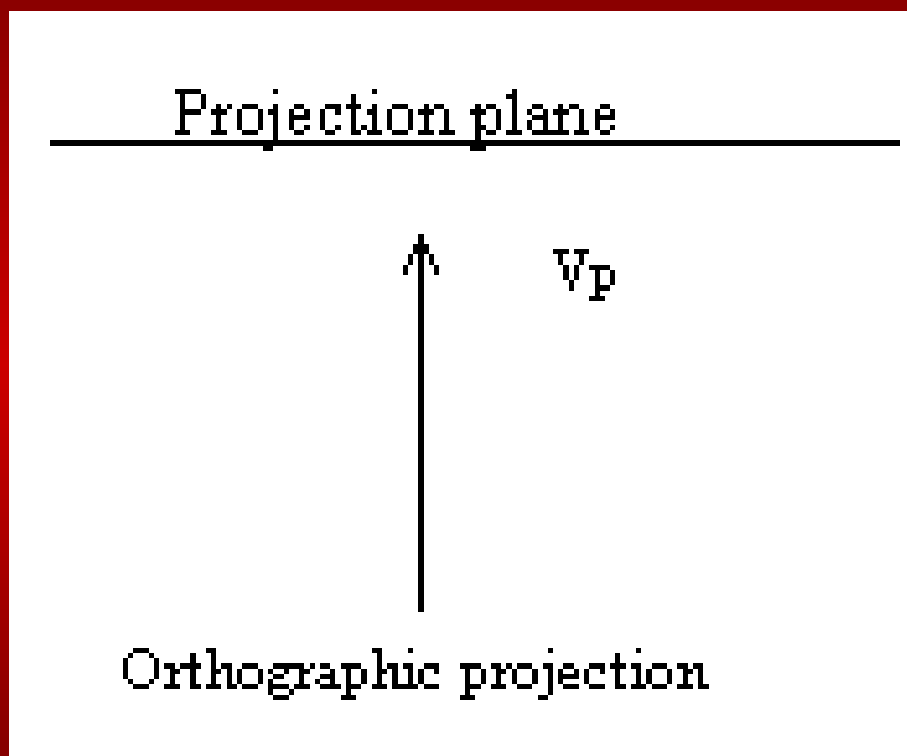


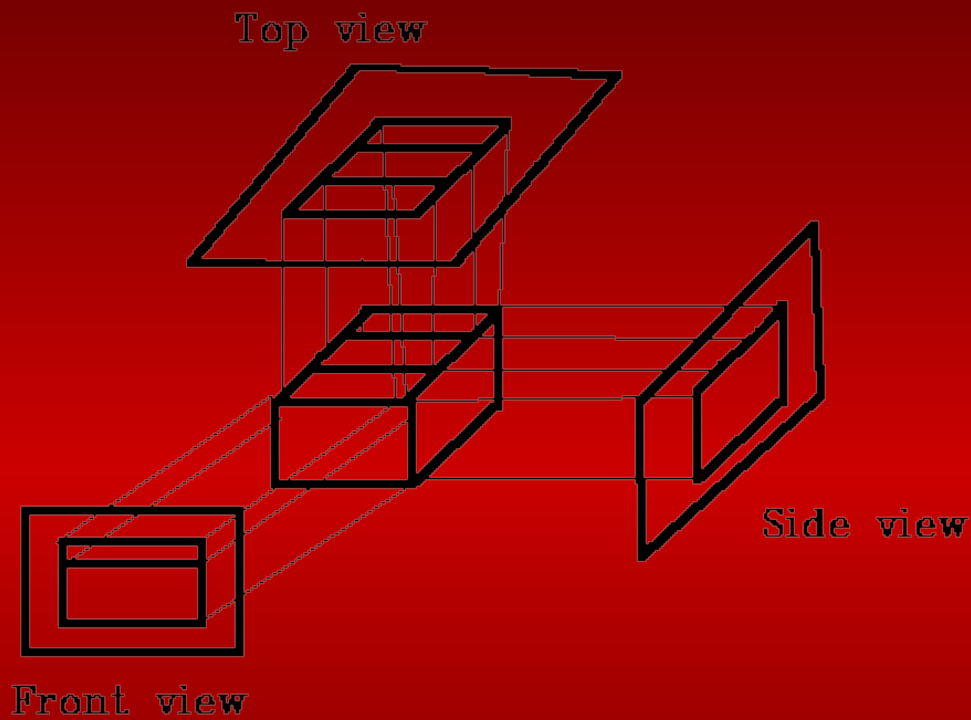


Toxonomy of Projection



1. Orthographic
2. Oblique





**Here are three orthographic views of an object.**

**There are three axonometric projections:**

- 1. Isometric**
- 2. Dimetric**
- 3. Trimetric**

# 1. Isometric

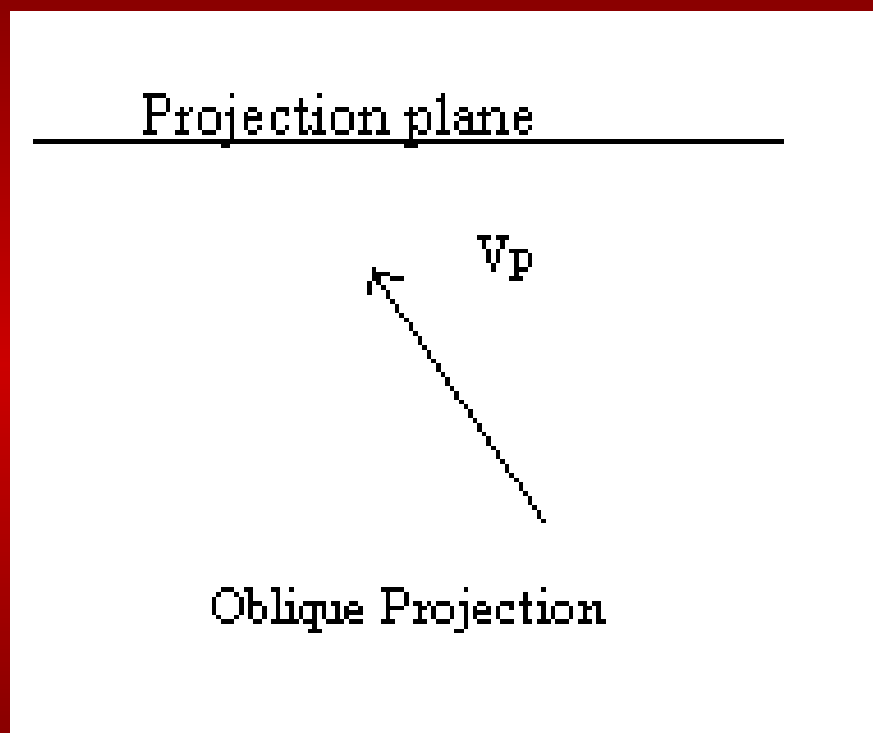
The projection plane intersects each coordinate axis in the model coordinate system at an equal distance or the direction of projection makes equal angles with all of the three principal axes

# 2. Dimetric

The direction of projection makes equal angles with exactly two of the principal axes

### 3. Trimetric

The direction of projection makes unequal angles with the three principal axes



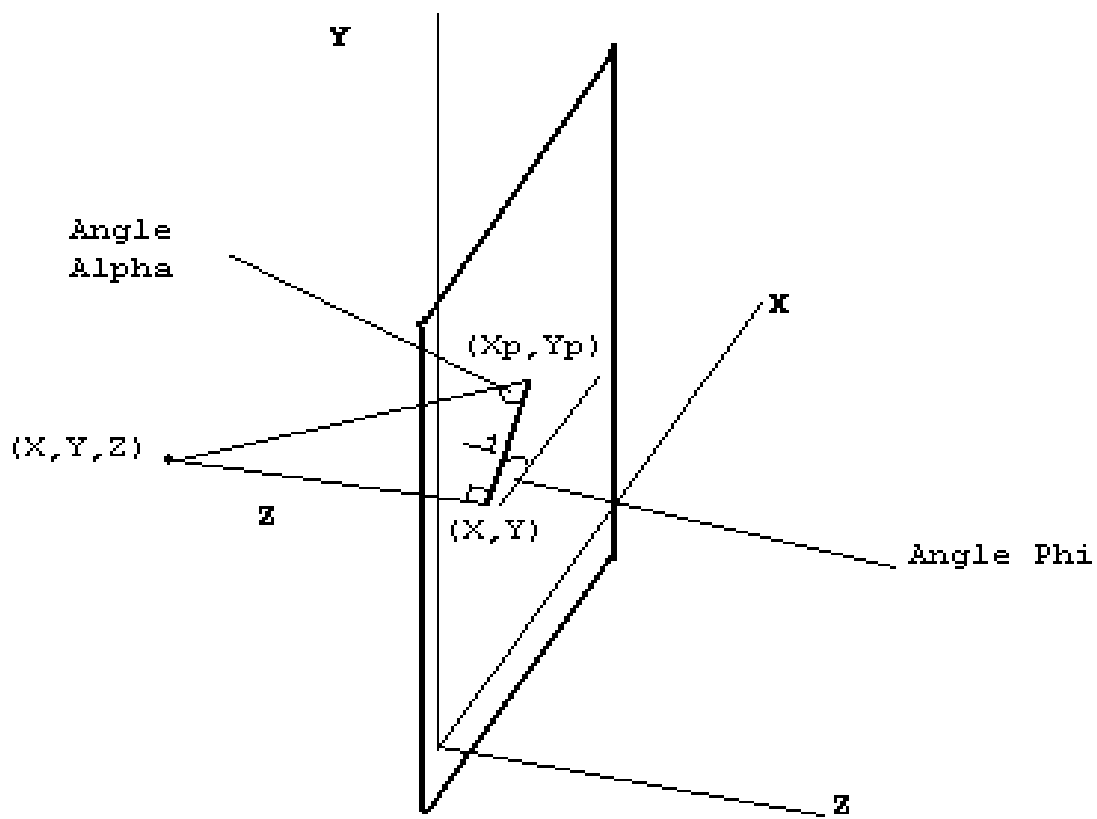


Figure: Oblique Projection of coordinate position  $(x, y, z)$  to position  $(X_p, Y_p)$  on the view plane

- ◆  $X_p = x + z (L1 \cos (\Phi) )$
- ◆  $Y_p = y + z (L1 \sin (\Phi) )$

# Computer Graphics Lecture 27

Computer Graphics

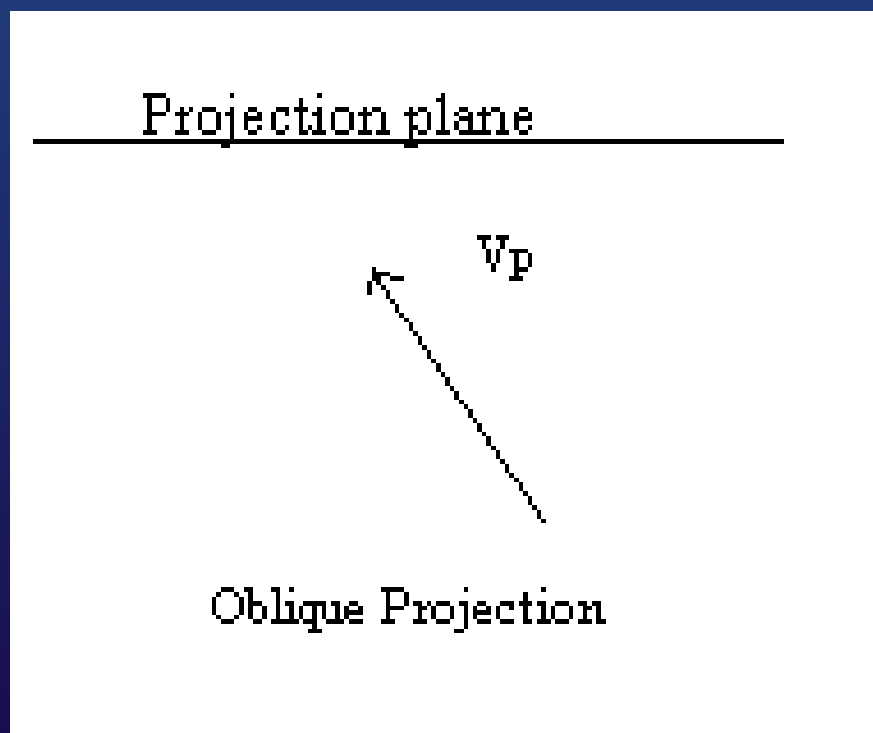
Lecture 28

**REVIEW III**

Taqdees A. Siddiqi  
cs602@vu.edu.pk

REVIEW III  
Taqdees A. Siddiqi  
cs602@vu.edu.pk

## Oblique Projection





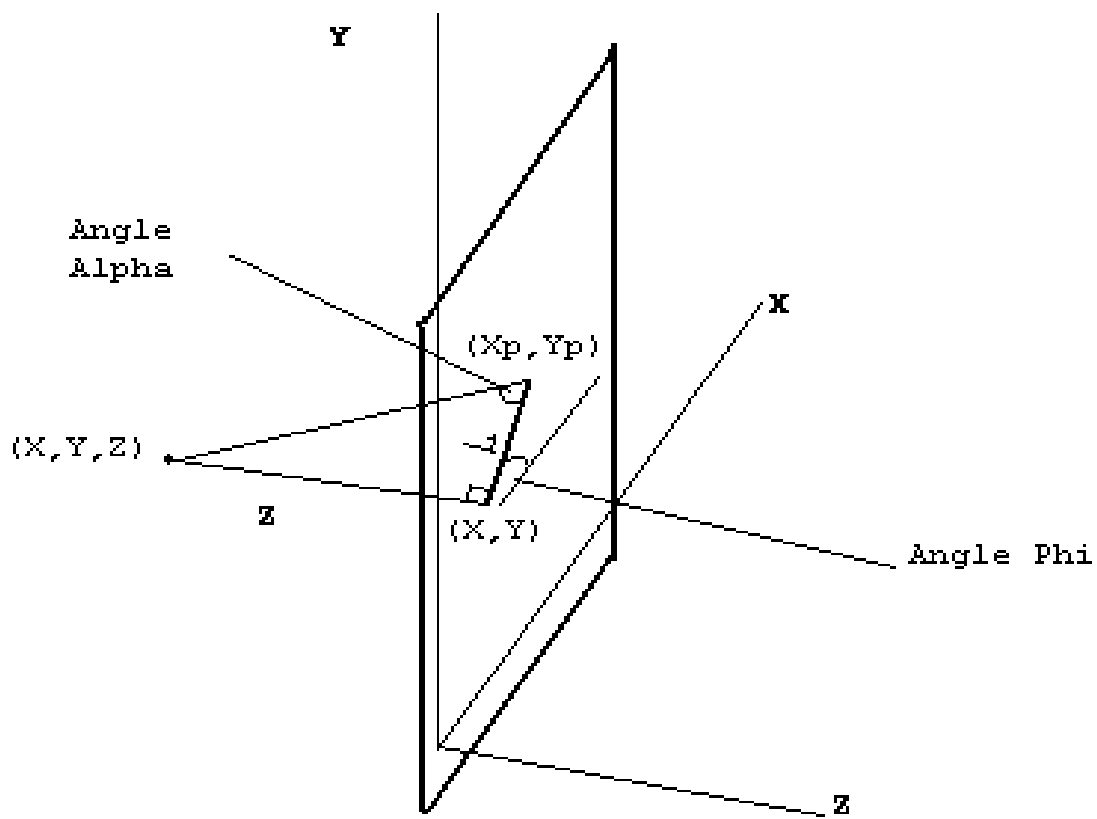
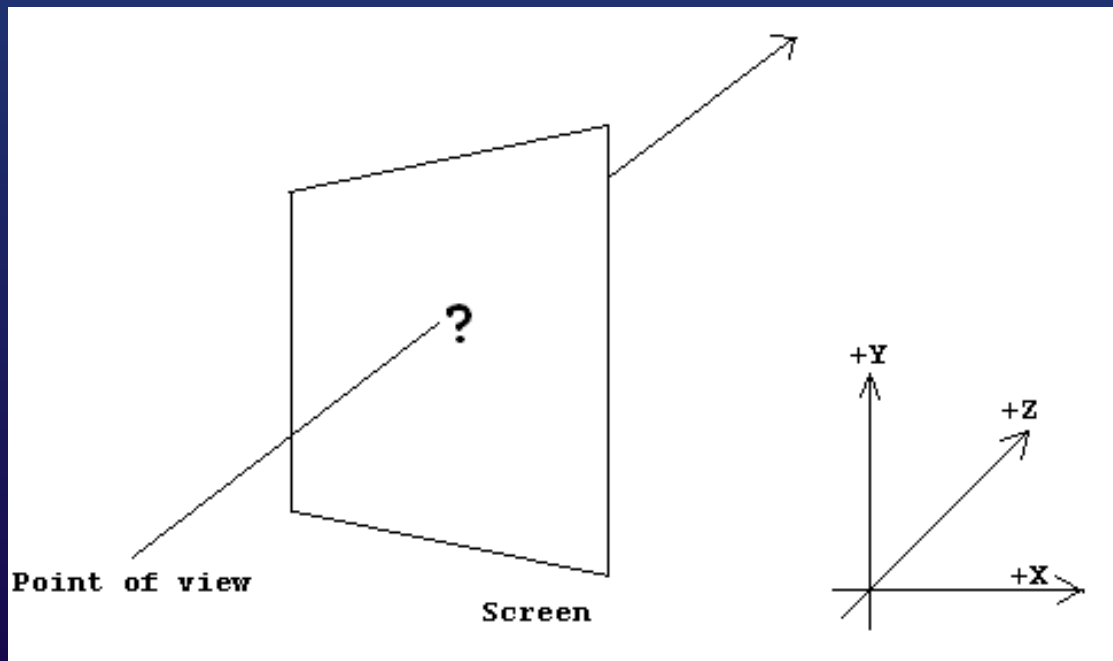


Figure: Oblique Projection of coordinate position  $(x, y, z)$  to position  $(X_p, Y_p)$  on the view plane

- ◆  $X_p = x + z (L1 \cos (\Phi) )$
- ◆  $Y_p = y + z (L1 \sin (\Phi) )$

# Perspective Projection



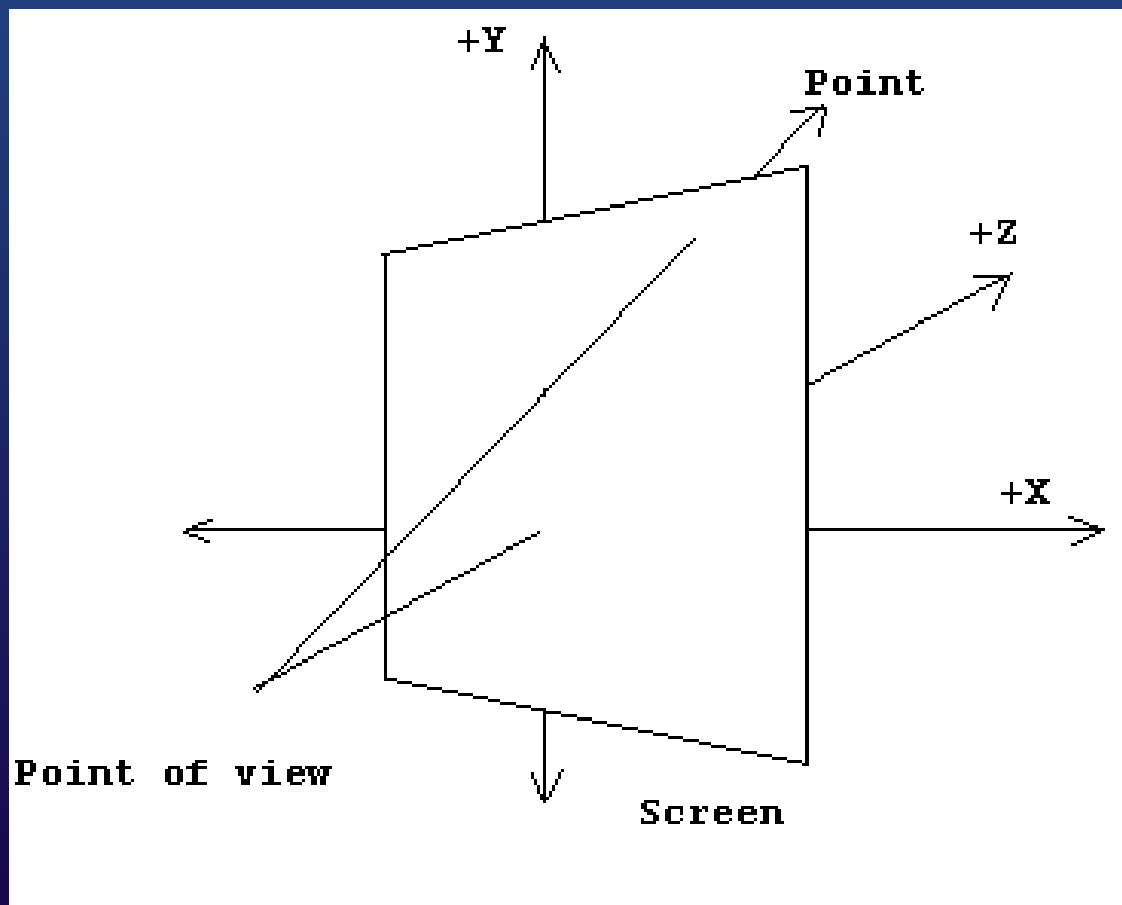
## Presumptions For Simplification

The point of View (POV) must lie on the Z axis,  
The screen plane must be parallel to the X-Y plane,  
with the left and right edges of the screen parallel to  
the Y-axis,  
and the top and bottom edges of the screen  
parallel to the X-axis

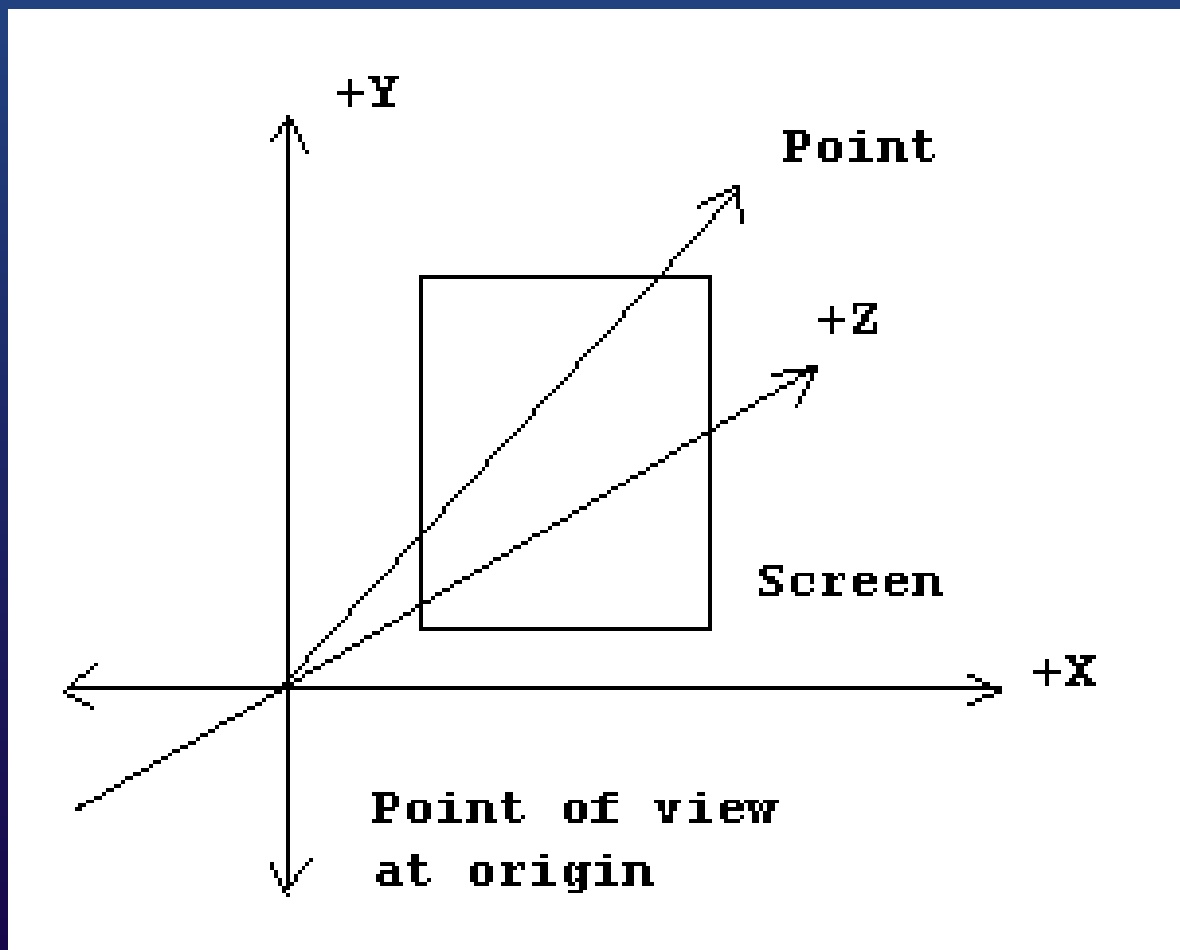
Because the POV represents the viewer's eye, and we presume that the viewer will be behind the center of the screen.

Two common approaches are used with this:

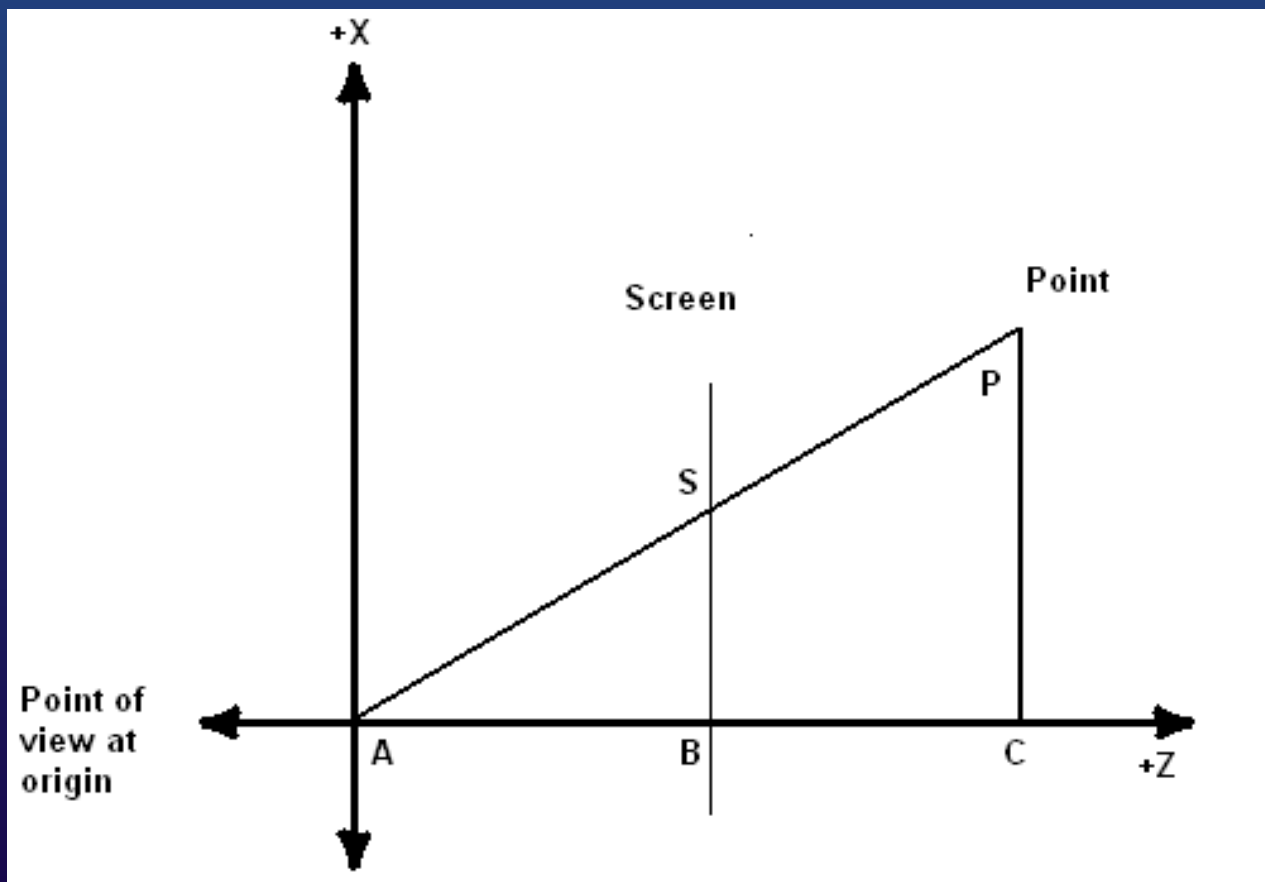
1. The first is where the POV is at some point  $(0, 0, -z)$  and the screen lies on the X-Y plane, graphically, this looks like figure:



- 
2. The second is where the POV lies at the origin, and the screen lies on a plane at some +Z coordinate, as shown in figure given below:



- ♦ Calculating the screen pixel that correlates to a 3D point is now a matter of simple geometry. From a viewpoint above the screen and POV (looking at the X-Z plane), the geometry appears like the one shown in figure below:



In geometric terms, we say that the triangle from A to B to S is similar to the triangle from A to C to P because the three angles that make up the triangles are the same; the angle from AB to AS is the same as the angle from AC to AP.

- ◆ The two right angles i.e.  $90^\circ$ , and therefore the remaining two angles are the same (the sum of the angles in a triangle is always 180 degrees).

What also holds true from similar triangles is that the ratio of two sides also holds between the similar triangles; this means that the ratio of BS to AB is the same as the ratio of CP to AC.

But we know what AB is - it is Screen.z! and  
we know what AC is - it is point.z! and  
we know what CP is - it is point.x!  
therefore:

$$|BS| / |AB| = |CP| / |AC|$$

$$|BS| = |AB| * |CP| / |AC|$$

$$|BS| = \text{Screen.z} * \text{point.x} / \text{point.z}$$

Note that:

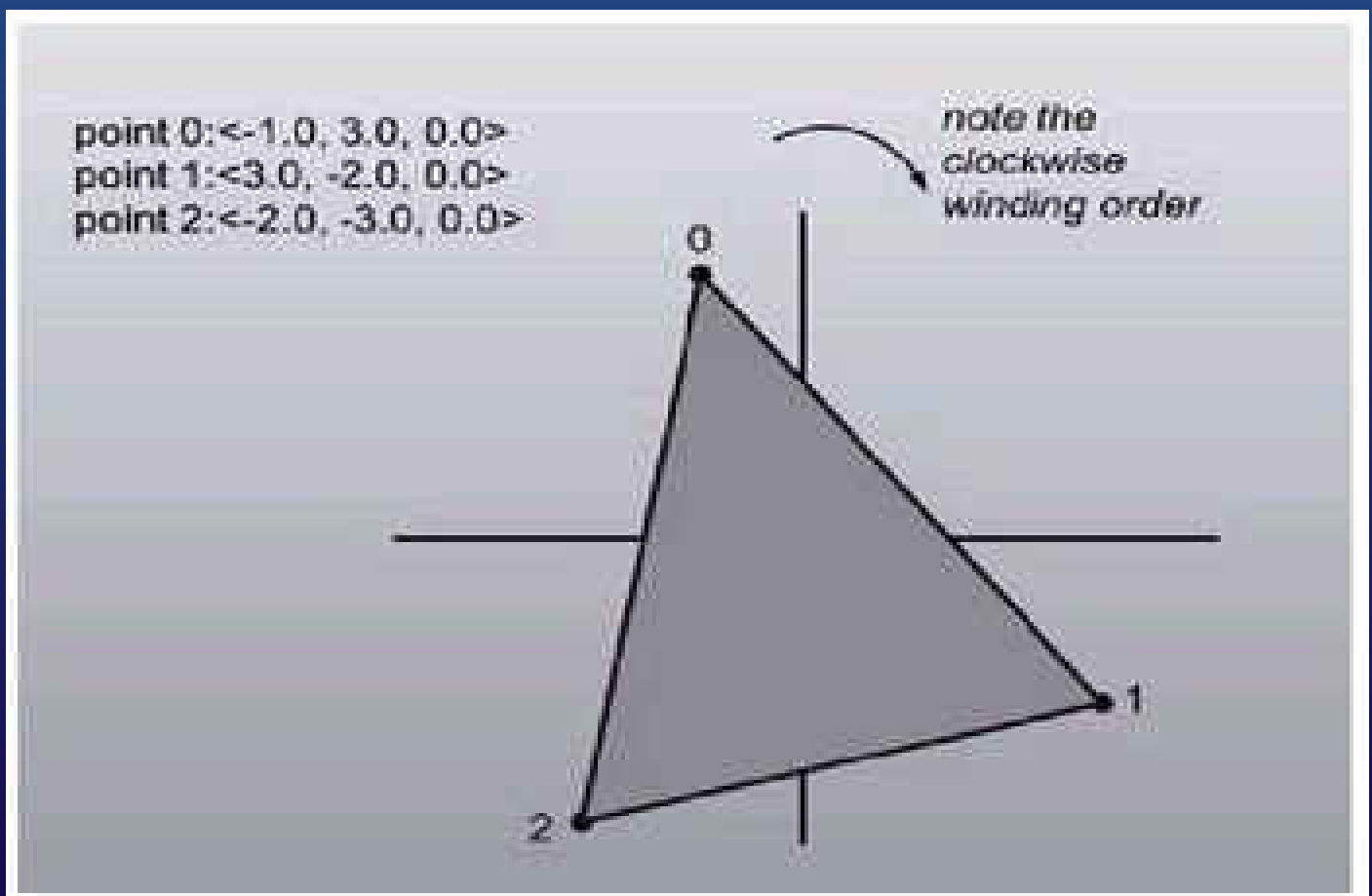
- ◆ Screen.z is the distance d from the point of view at origin or the scaling factor.



# Triangles

- ◆ Triangles "flesh out" a 3D object, connecting them together to form a skin or mesh that defines the boundary surface of an object

- ♦ The ordering of the vertices goes *clockwise* around the triangle



**Three points in space, and the triangle connecting them**

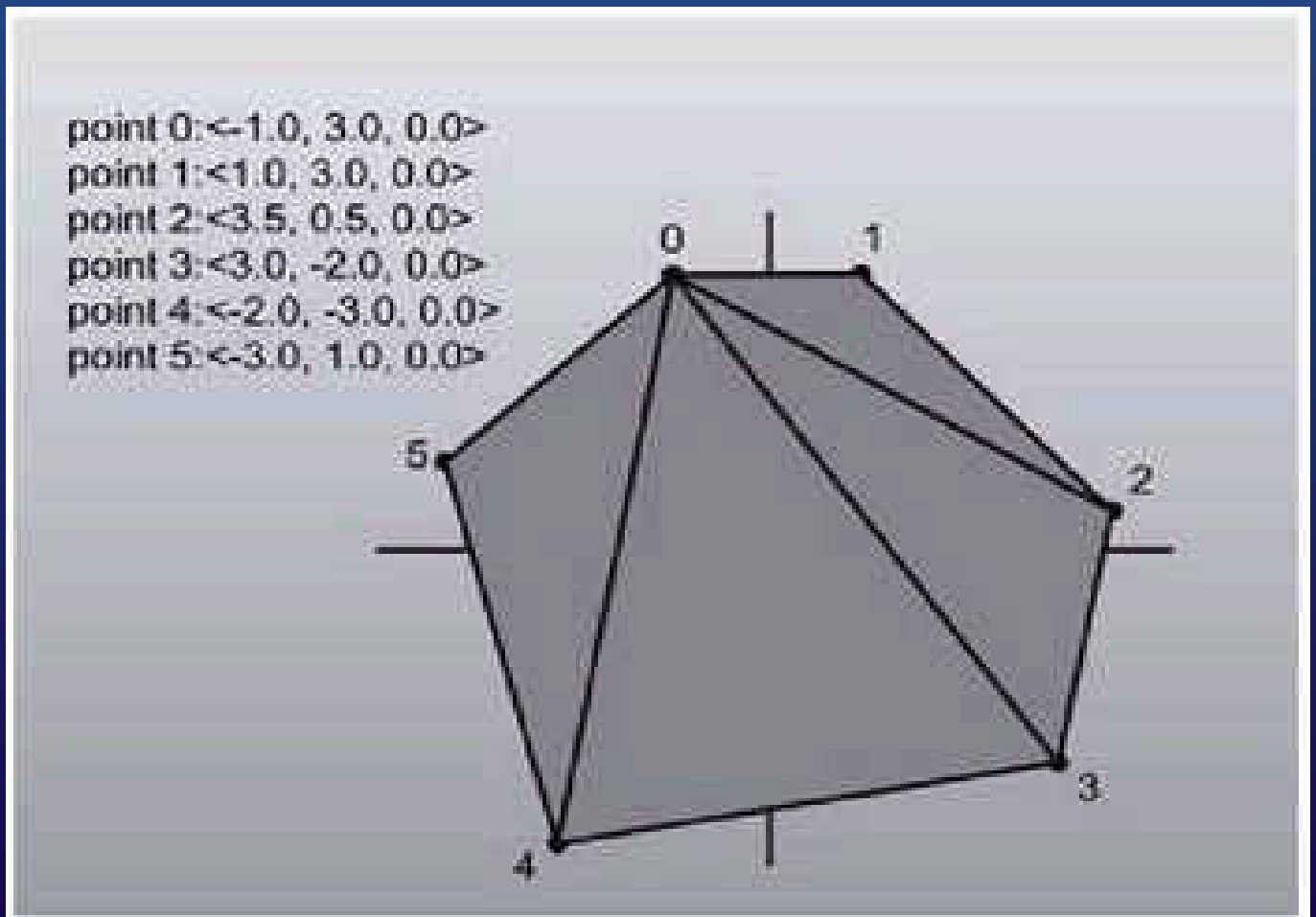
# Triangle Strips and Fans

## Triangle List

- ◆ The first is an explicit list or array of triangles, where every three elements represent a new triangle.

# Triangle Fan

- Triangle fans, conceptually, look like the folding fans you see in souvenir shops. They are a list of triangles that all share a common point.



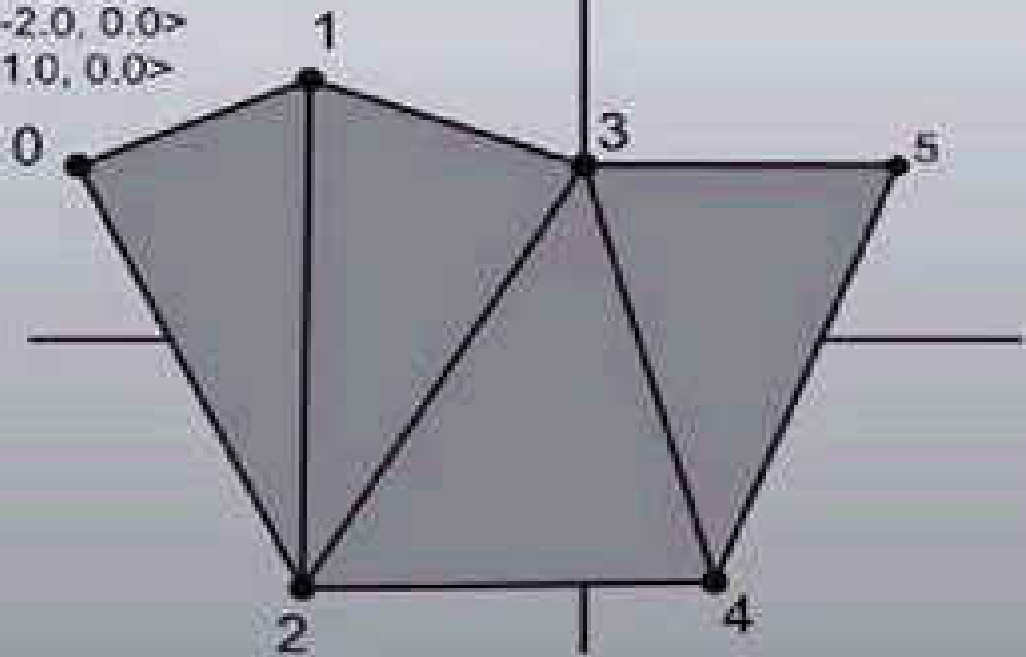
**A list of points composing a triangle fan**

# Triangle Strip

- ◆ Triangles in a triangle strip, instead of sharing a common element with all other triangles like a fan, only share elements with the triangle immediately preceding them.

- ◆ The first three elements define the first triangle. Then each subsequent element is combined with the two elements before it, in clockwise order, to create a new triangle

point 0:<-3.0, 1.0, 0.0>  
point 1:<-2.0, 1.5, 0.0>  
point 2:<-2.0, -1.5, 0.0>  
point 3:<0.0, 1.0, 0.0>  
point 4:<0.5, -2.0, 0.0>  
point 5:<2.0, 1.0, 0.0>



**A list of points composing a triangle strip**

## Plane

- ◆ Planes are defined as infinitely large, infinitely thin slices of space, like big pieces of paper.

- ◆ Each of the triangles that make up our model, exists in its own plane

Equation of plane in 3D

$$Ax + By + Cz + d = 0$$

$$Ax + By + Cz + d = 0$$

- ♦ The triplet  $(a, b, c)$  represents what is called the normal of the plane

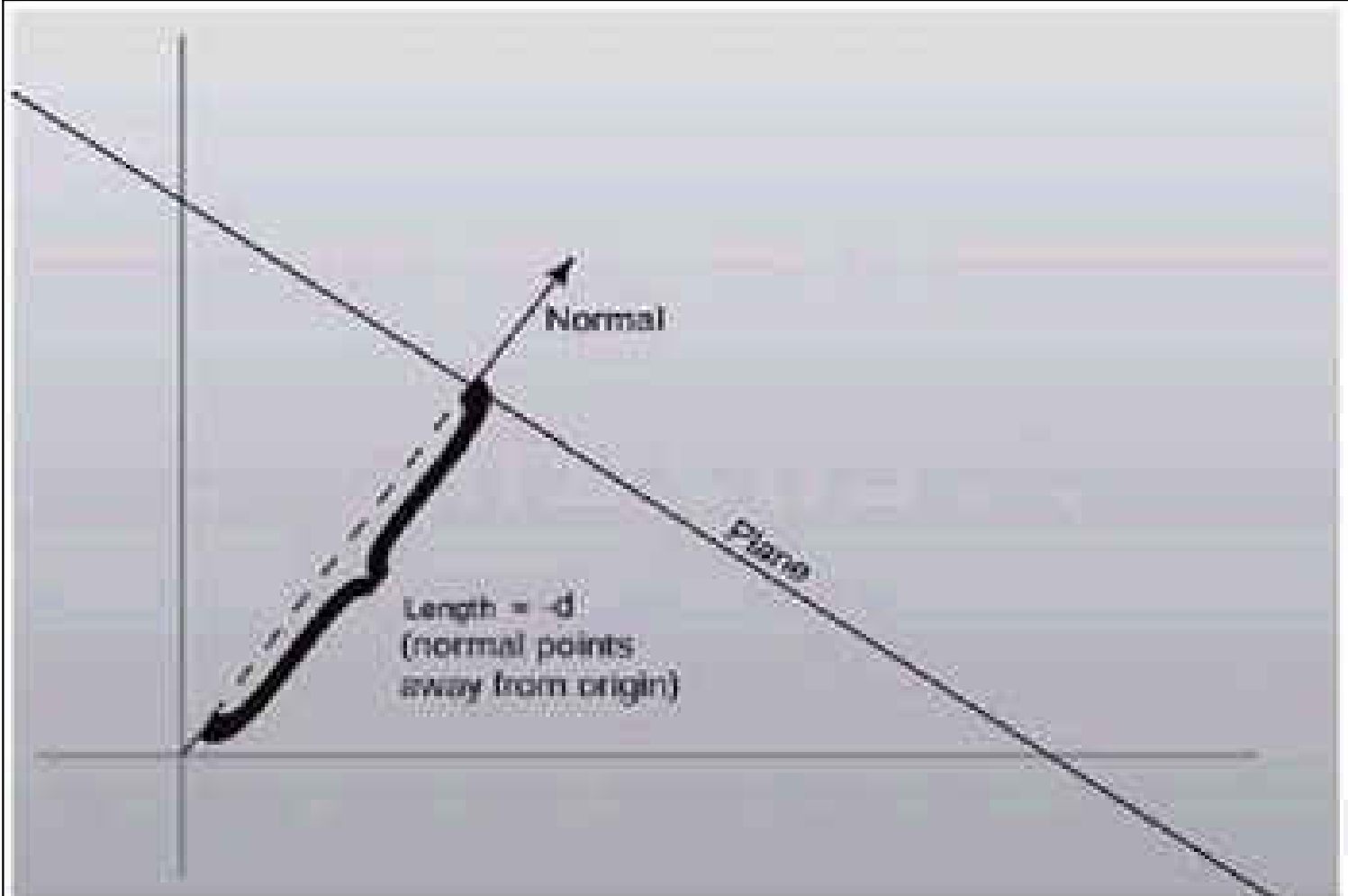
$$Ax + By + Cz + d = 0$$

- ♦ The  $d$  component in the equation represents the distance from the plane to the origin. The distance is computed by tracing a line towards the plane until you hit it.

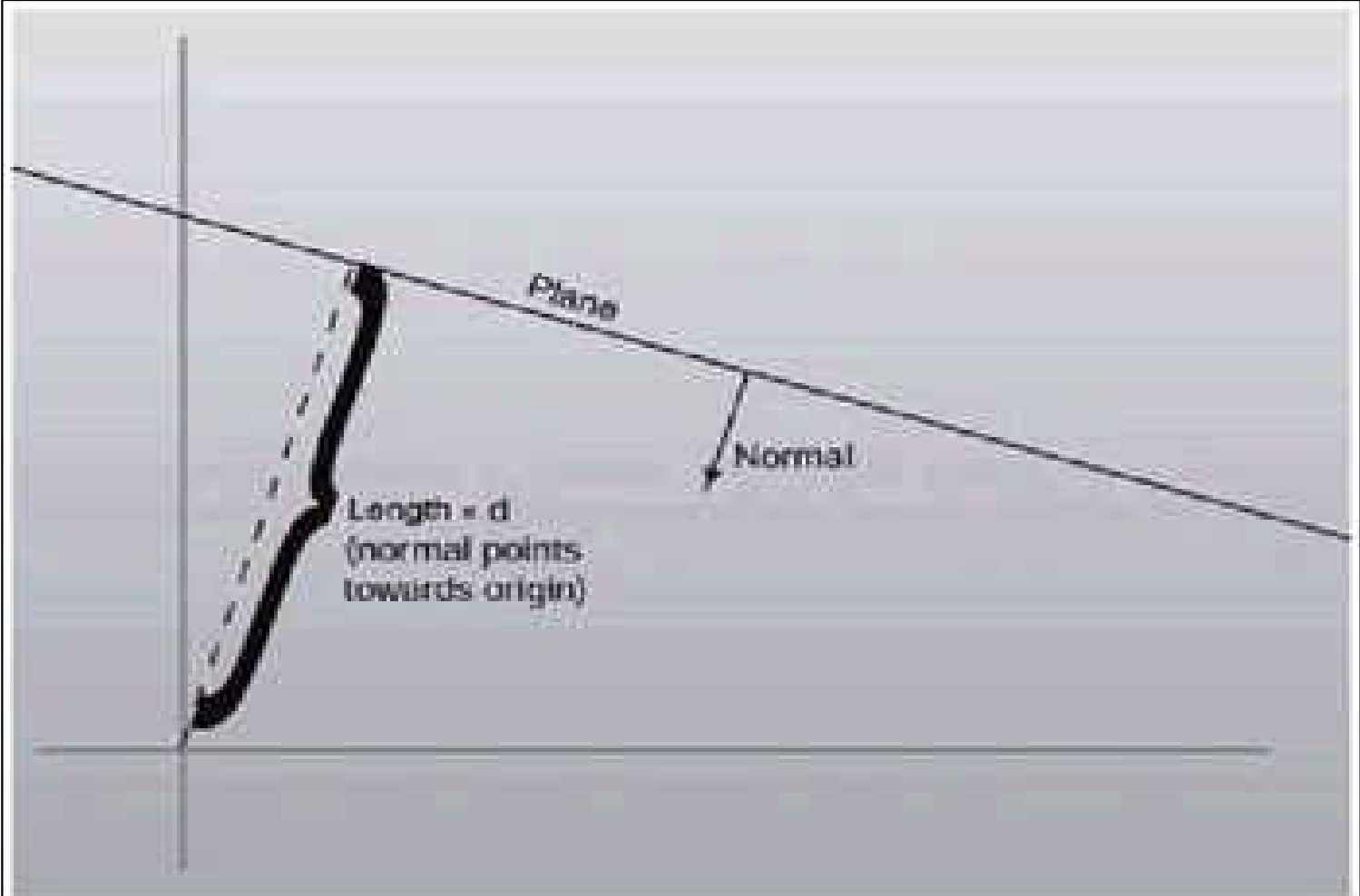


$$Ax + By + Cz + d = 0$$

- ◆ Finally the triplet  $(x, y, z)$  is any point that satisfies the equation
- ◆ The set of all points  $(x, y, z)$  that solve the equation is exactly all the points that lie in the plane



'd' is -ve when the normal faces away from origin



'd' is +ve when the normal faces towards origin

- ♦ It's important to notice that technically the normal  $(x, y, z)$  does not have to be unit-length for it to have a valid plane equation. But since things end up nicer if the normal is unit-length.

# Back-face Culling

## Back-face Culling

- ◆ To perform the back-face cull, just subtract one of the triangle's points from the camera location and perform a dot product of the resultant vector and the normal. If the result of the dot product is greater than zero, then the view point was in front of the triangle

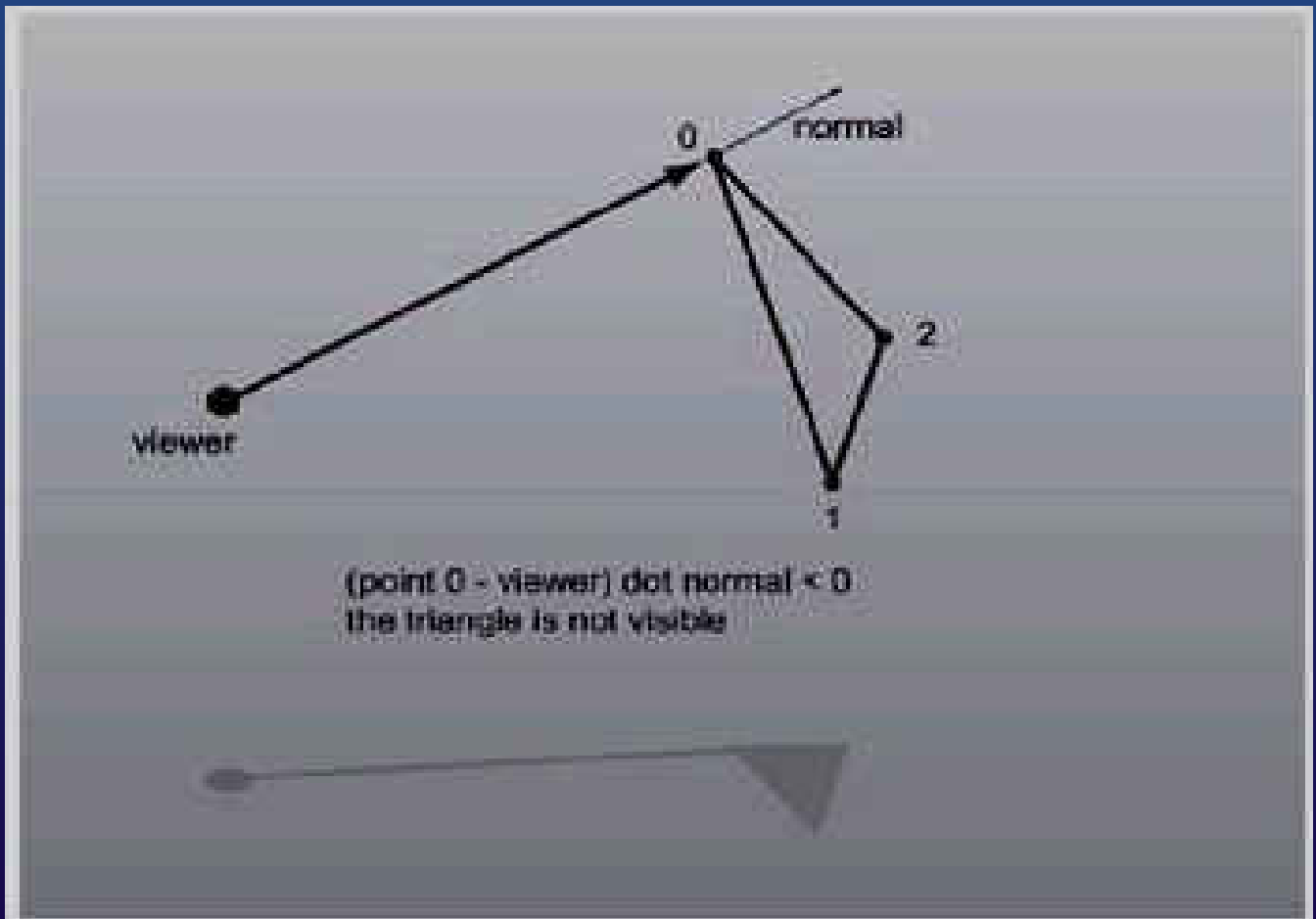
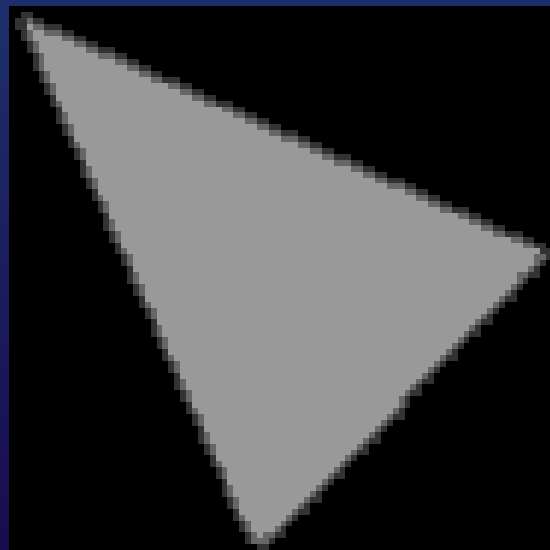
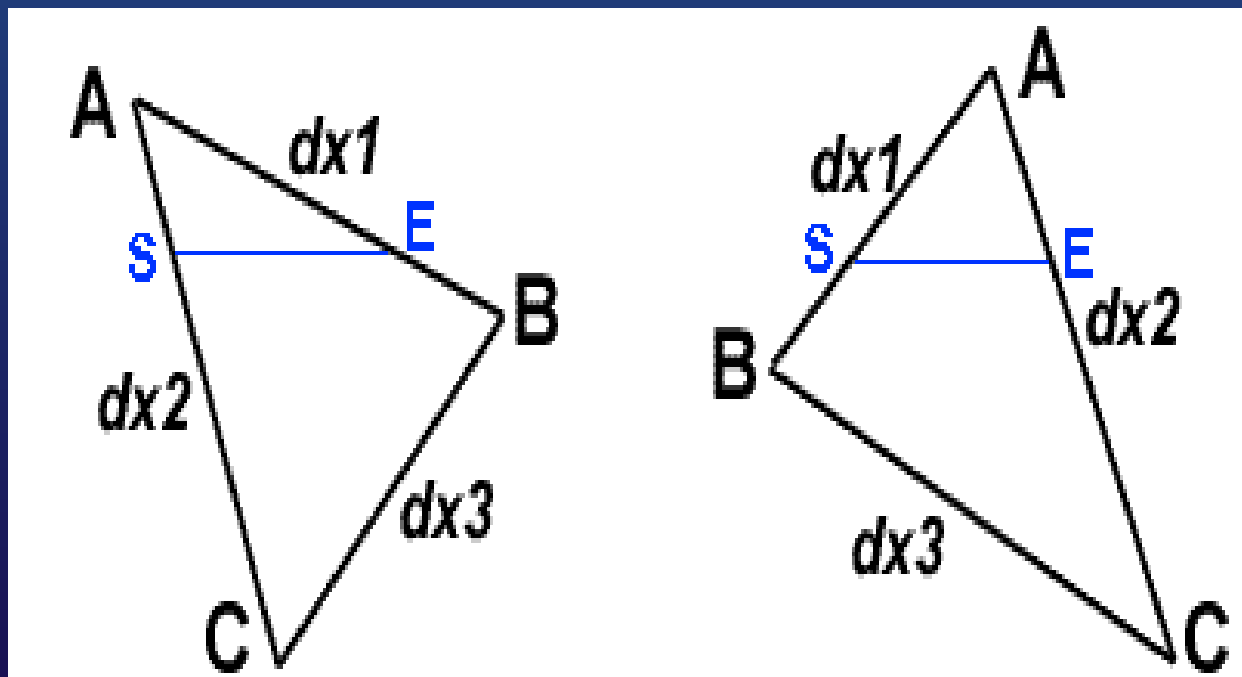


**Figure: A visual example of back-face culling**

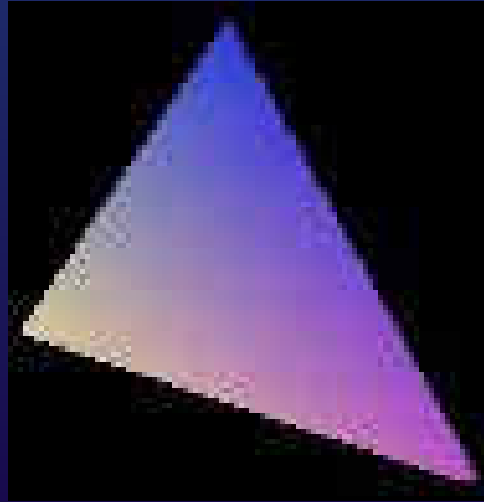
## Triangle Rasterization Flat Filling Triangles



- ◆ For each scan line (horizontal line on the screen), find the points of intersection with the edges of the triangle. Then, draw a horizontal line between intersections and do this for all scan lines.



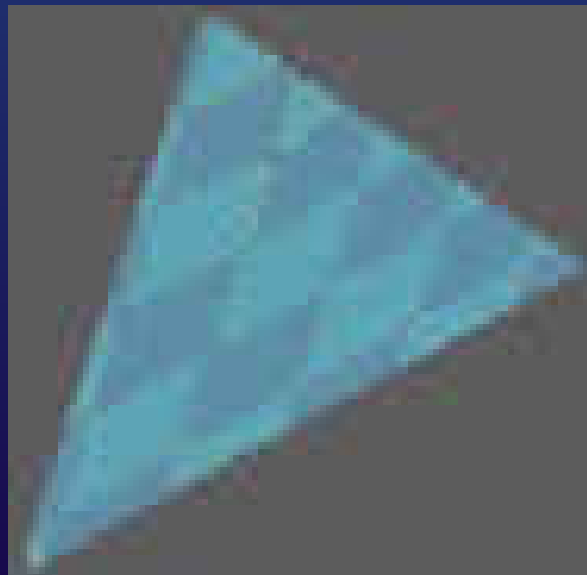
# Gouraud Shading Triangles



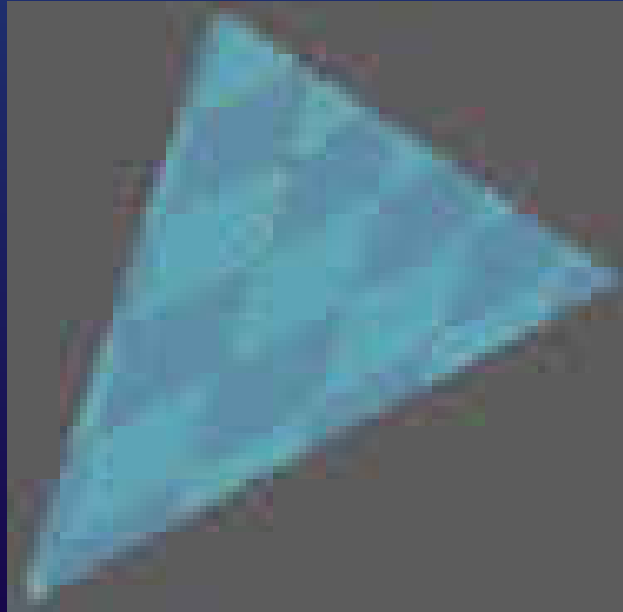
- ◆ Flat triangle interpolated only one value (x in connection with y), 256 colors gouraud needs three (x related to y, color related to y, and color related to x),

- ◆ hi-color gouraud needs seven parameters (i.e. x related to y, red, green and blue components of color related to y, and color related to x (also three components))

## Textured Triangles



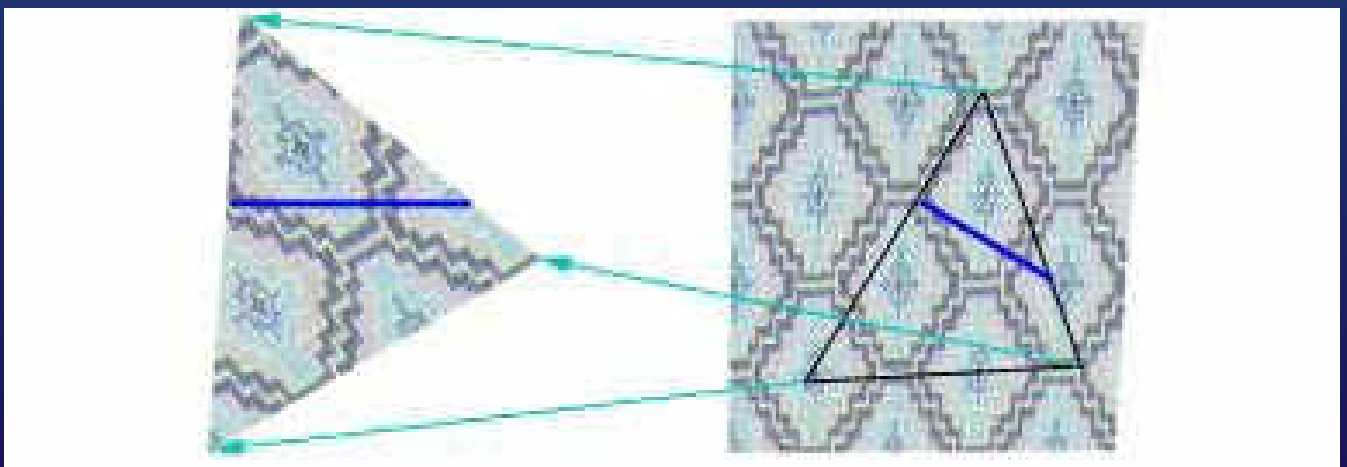
- ◆ We can also apply any bitmap on triangle for filling it.



- ◆ Again we're using the idea of interpolation; And again the idea is perfectly the same, only two more values to interpolate, that is five values in all



- ◆ In texture mapping, we interpolate  $x$ ,  $u$ , and  $v$  related to  $y$ , and  $u$  and  $v$  related to  $x$  ( $u$  and  $v$  are coordinates in the 2D bitmap space)



The left triangle is the triangle which is drawn onto the screen. There's a single scanline (one call to the horizontal line routine) pointed out as an example.

The triangle on the right is the same triangle in the bitmap space, and there's the same scanline drawn from another point of view into it, too..

# COLOR

- ◆ A color is usually represented in the graphics pipeline by a three-element vector representing the intensities of the red, green, and blue components
- ◆ Or for a more complex object, by a four-element vector containing an additional value called the alpha component that represents the opacity of the color

- ◆ Colors can also be represented as floating point values in the range  $[0,1]$

A 16-bit display is named since each pixel in a 16-bit image is taken up by 16 bits (2 bytes):

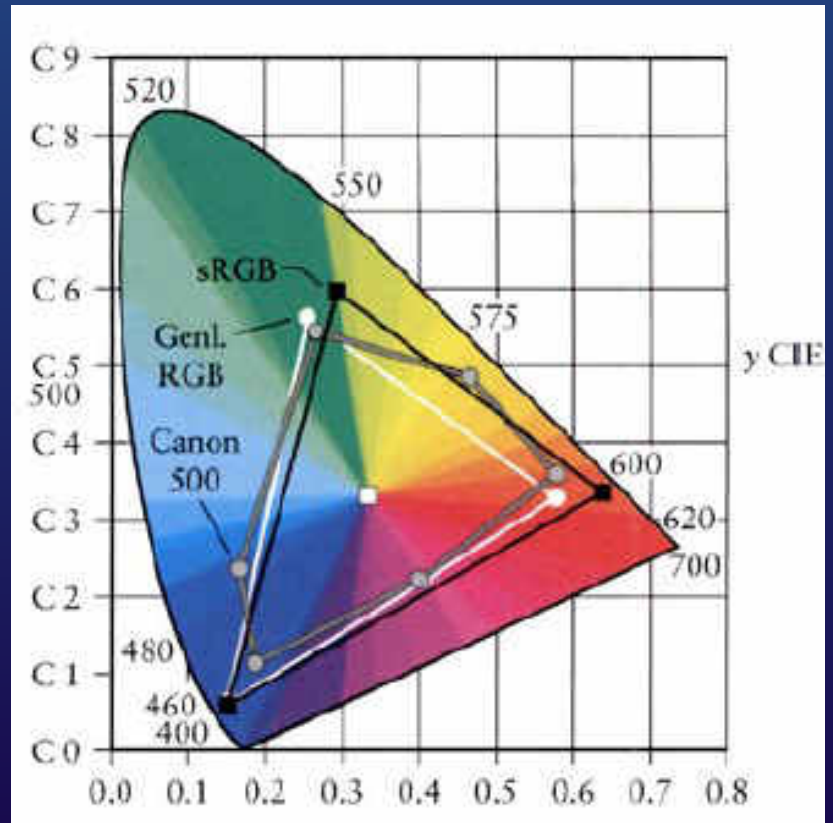
- ◆ 5 bits of red information,
- ◆ 6 bits of green information,
- ◆ and 5 bits of blue information

- ◆ A 24-bit display, of course, uses 24 bits, or 3 bytes per pixel, for color information. This gives 1 byte, or 256 distinct values each, for red, green, and blue

Finally, there is 32-bit color, something seen on most new graphics cards. Many 3D accelerators keep 8 extra bits per pixel around to store transparency information, which is generally referred to as the alpha channel

# GAMUT

The 1931 CIE diagram shows the gamut of the eye and the lesser gamut of output devices.



## Multiplication of Colors

- ◆ Colors are multiplied to describe the interaction between a surface and a light source
- ◆ The colors of each are multiplied together to estimate the reflected light color—this is the color of the light that this particular light reflects off this surface

- This intensity of our light is represented by, say, a nice lime green color

$$\text{light color } i_l = [0.34765, 0.92578, 0.24609]$$

- Let's say we shine this light on a nice magenta surface given by  $c_s$ .

$$\text{surface color } c_s = [0.86719, 0.00000, 0.98828]$$

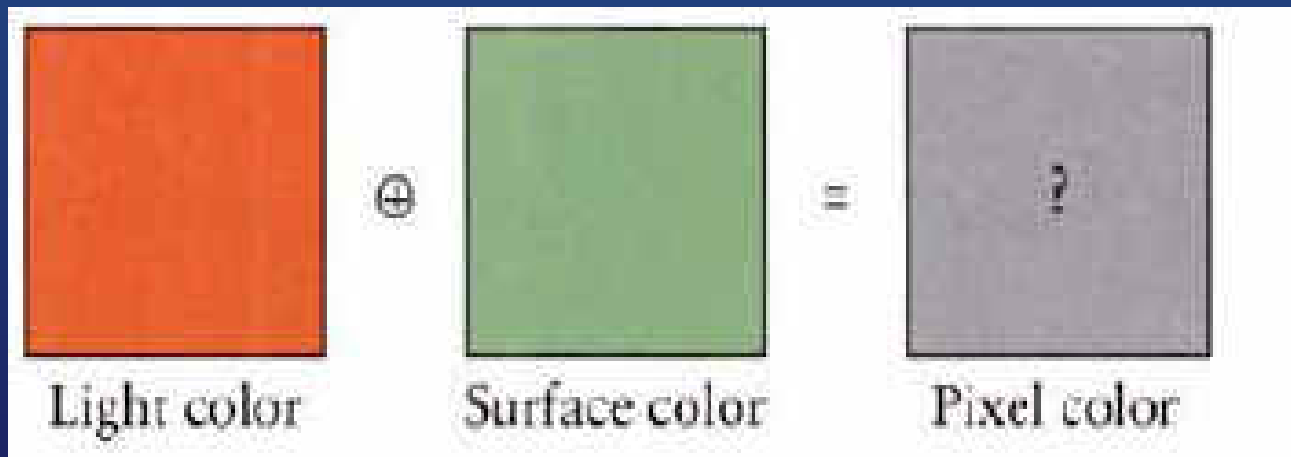
$$\begin{aligned} i_l \otimes c_s &= [0.34765, 0.92578, 0.24609] \otimes [0.86719, 0.00000, 0.98828] \\ &= [(0.34765)(0.86719), (0.92578)(0), (0.24609)(0.98828)] \\ &= [0.30148, 0.00000, 0.243210] \end{aligned}$$



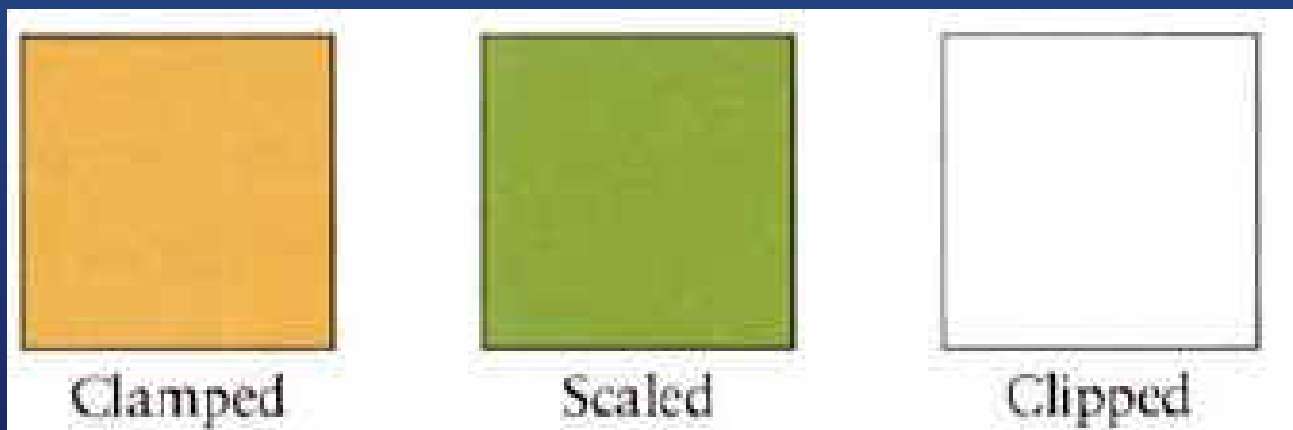
Multiplying (modulating) color values results in a color equal to or less than (darker) the original two.

## Dealing with Saturated Colors





Adding colors can result in colors that are outside the displayable range.



The results of three strategies for dealing with the same oversaturated color.

# Clamping Color Values

Any value that's greater than one is clamped to one, and any value less than zero are clamped to zero.

## Scaling Color Values

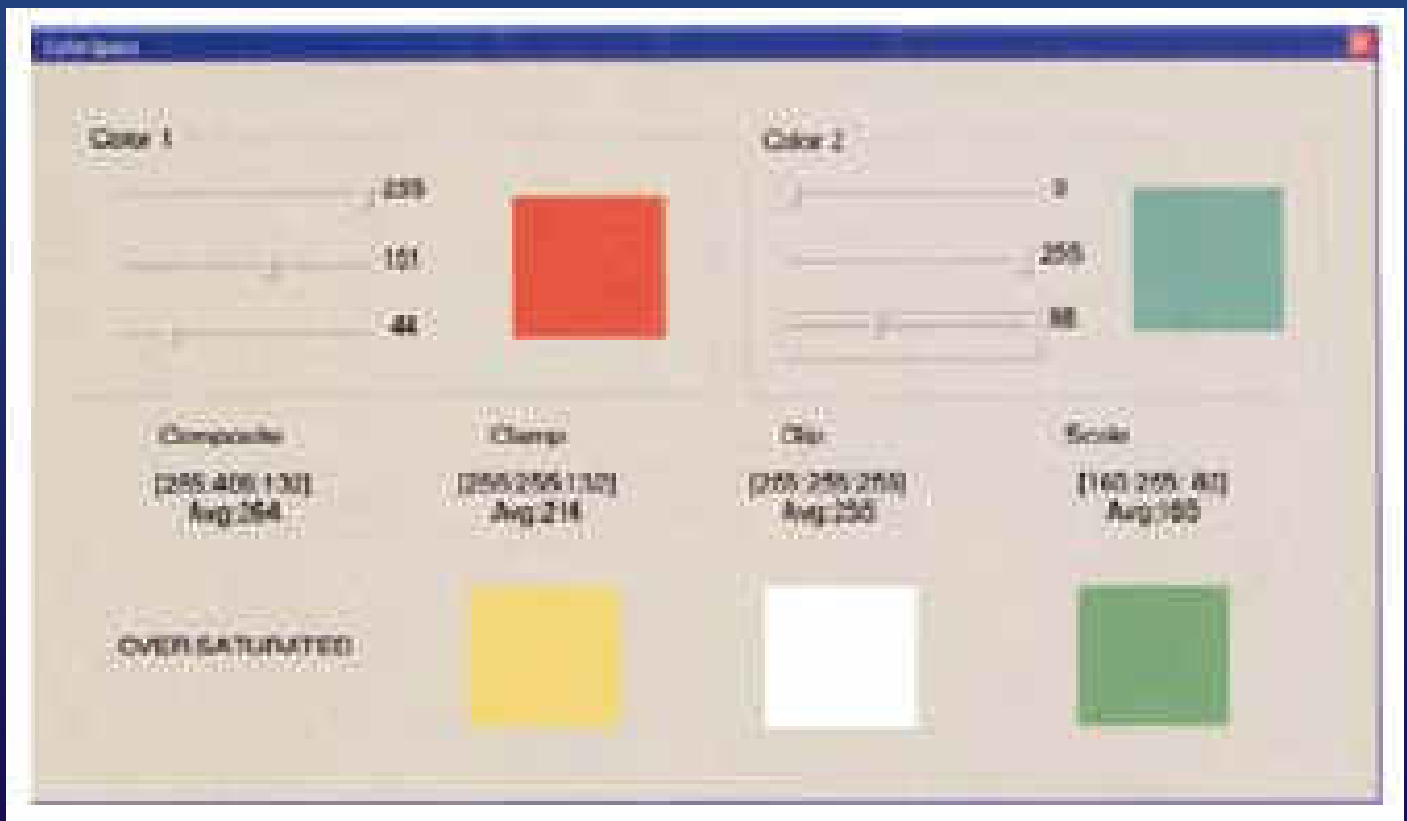
Instead of clamping, we might want to scale the color by dividing by the largest color value, thus scaling the *rgb* values into the  $[0,1]$  range.

# Clipping Color Values

One problem with clamping or scaling colors is that they get darker (lose saturation). An alternative to scaling is to maintain saturation by shifting color values. This technique is called clipping,

## Color Space Tool

- ◆ The ColorSpace tool is a handy tool that we can use to interactively add two colors together to see the effects of the various strategies for handling oversaturated colors.



## Negative Colors and Darklights

- ◆ Darklights are nothing more than lights in which one or more of the color values are negative.

# Uses of Darklights

- ◆ Eliminate bright areas of a scene
- ◆ Filter out a specific RGB color
- ◆ Night Vision – to eliminate Red and Blue components

## Alpha Blending

- ◆ An alpha value of 0xFF (255) means the color is completely opaque, and an alpha value of 0x00 (0) means the color is completely transparent.

- ◆ The name "alpha blending" comes from the fact that generally the blending factors used are either the alpha or the inverse of the alpha.

$$\begin{aligned} \text{Final Color} = & \\ & \text{SOURCE} * \\ & \text{SOURCE BLEND FACTOR} \\ & + \\ & \text{DESTINATION} * \\ & \text{DESTINATION BLEND FACTOR} \end{aligned}$$

FINAL COLOR = SOURCE \* 1.0 + DESTINATION \* 0.0



**First Image**



**Second Image**



**Image after blending**



# Computer Graphics

## Lecture 28

# Computer Graphics

## Lecture 29

# Mathematics of Lighting and Shading - III

Taqdees A. Siddiqi

[cs602@vu.edu.pk](mailto:cs602@vu.edu.pk)

# Mathematics of Lighting and Shading - III

Taqdees A. Siddiqi  
cs602@vu.edu.pk

**Traditional 3D  
Hardware-Accelerated Lighting  
Models**

We will now take a look at the traditional method of calculating lighting in hardware

A method that we'll find is sufficient for most of our needs

The traditional approach in real-time computer graphics has been to calculate lighting at a vertex as a sum of the ambient, diffuse, and specular light.

In the simplest form (used by OpenGL and Direct3D), the function is simply the sum of these lighting components (clamped to a maximum color value).

we have an ambient term and then a sum of all the light from the light sources

$$i_{\text{total}} = k_a i_a + \sum_i (k_d i_d + k_s i_s)$$

where *itotal* is the intensity of light (as an rgb value) from the sum of the intensity of the global ambient value and the diffuse and specular components of the light from the light sources

This is called a local lighting model since the only light on a vertex is from a light source, not from other objects.

That is, lights are lights, not objects!

we've included the *reflection coefficients* for each term,  $k$  for completeness.

The reflection coefficients are in the  $[0,1]$  range and are specified as part of the material property

This equation does fail to take into account any gross roughness or anything other than perfect isotropic reflection

the surface is treated as being perfectly smooth and equally reflective in all directions

Thus this equation is really not good at modeling the illumination of objects that have anything other than a smooth surface (like fur or sand)



or a surface that doesn't really reflect light uniformly in all directions (like brushed metal, hair, or skin).

However, with liberal use of texture maps to add detail, this model has served pretty well and can still be used for a majority of the lighting processing to create a realistic environment in real time

Let's take a look at the individual parts of the traditional lighting pipeline.

## **Ambient Light**

Ambient light is the light that comes from all directions — thus all surfaces are illuminated equally regardless of orientation



Figure 1: Ambient light provides illumination, but no surface details.

We can see that it's difficult to make out details or depth information with just ambient light.

Ambient lighting is our friend. With it we make our scene seem more realistic than it is.

A world without ambient light is one filled with sharp edges, of bright objects surrounded by sharp, dark, harsh shadows.

A world with too much ambient light looks washed out and dull.

# Diffuse Light

Diffuse light is the light that is absorbed by a surface and is reflected in all directions

In the traditional model, this is *ideal* diffuse reflection

This is good for rough surfaces where the reflected intensity is constant across the surface and is independent of viewpoint but depends only upon the direction of the light source to the surface

The regardless of the direction from which we view an object with a stationary diffuse light source on it, the brightness of any point on the surface will remain the same

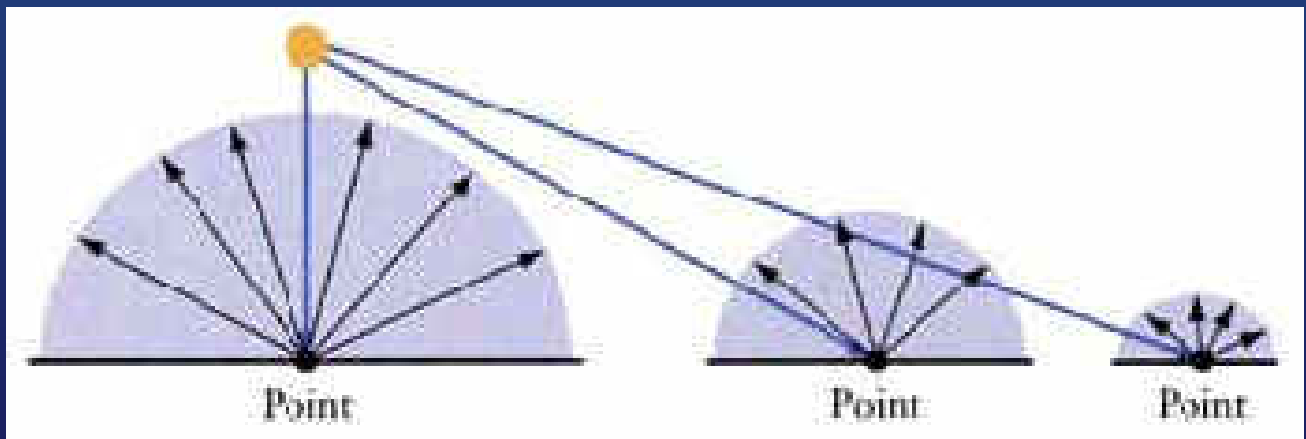
Unlike ambient light, the intensity of diffuse light is directional and is a function of the angle of the incoming light and the surface



This type of shading is called *Lambertian shading* after Lambert's cosine law,

The law states that the intensity of the light reflected from an ideal diffuse surface is proportional to the cosine of the direction of the light to the vertex normal

Since we're dealing with vertices here and not surfaces, each vertex has a normal associated with it



Diffuse light decreases as the angle between the light vector and the surface normal increases.

Figure 2 shows the intensity of reflected light as a function of the angle between the vertex normal and the light direction.

$$\vec{r}_d = (\hat{n} \cdot \hat{l})(m_d \otimes s_d)$$

The equation for calculating diffuse lighting is

Diffuse light term is now multiplied by the dot product of the unit normal of the vertex and the unit direction vector *to* the light from the vertex (not the direction *from* the light)



Diffuse light shading brings out some surface details

The problem with just diffuse lighting is that it's independent of the viewer's direction

Thus as we change the viewing angle to a vertex, the vertex's diffuse light value never changes

We have to rotate the object (change the normal direction) or move the light (change the light direction) to get a change in the diffuse lighting of the object

However, when we combine the ambient and diffuse, as in Figure 4, we can see that the two types of light give a much more realistic representation than either does alone



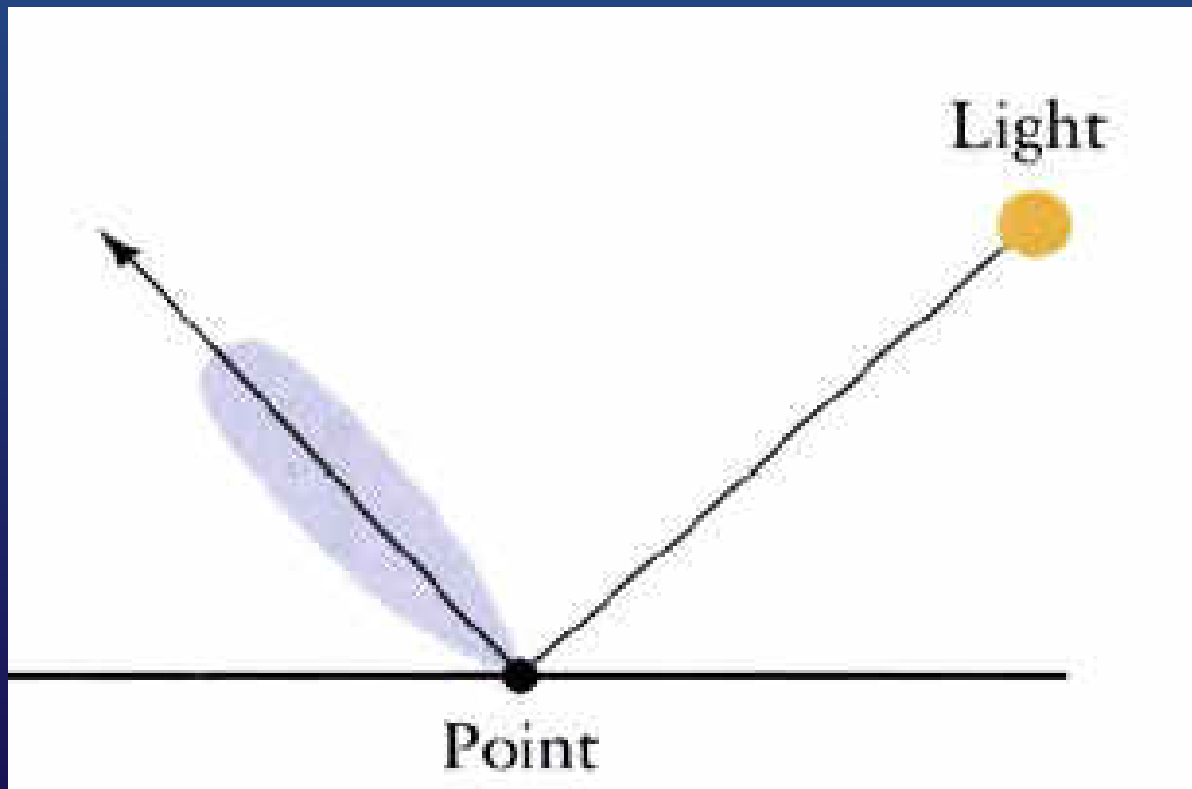
Ambient and Diffuse light together give a much better **shading**

Figure 4: When diffuse and ambient terms are combined, you get more detail and a more natural-looking scene. The final color is the combination of the ambient and diffuse colors.

# Specular Light

Specular light is the light from a light source that is reflected by a surface and is reflected in such a manner that it's both a function of the light's vector and the viewer's direction.





Specular light's intensity follows the reflection vector.

Most discussions of lighting (including this one) start with Phong's lighting equation

# Phong's Specular Light Equation

Warnock [WARNOCK 1969] and Romney [ROMNEY 1969] were the first to try to simulate highlights using a  $\cos^n(\theta)$  term.

But it wasn't until Phong Bui-Tong [BUI 1998] reformulated this into a more general model that formalized the power value as a measure of surface roughness that we approach the terms used today for specular highlights

**Phong's equation for specular lighting is**

$$I_s = (m_s \otimes s_s) (\hat{p} \bullet \hat{v})^{m_s} \quad (\text{Phong})$$

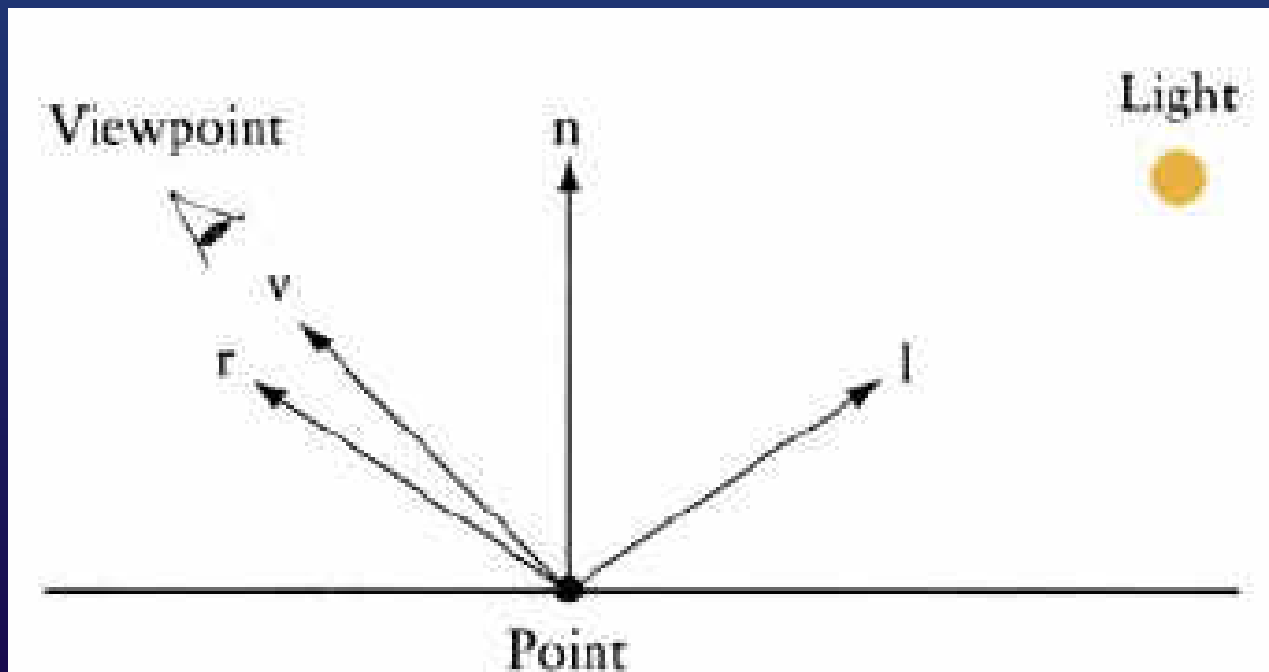


Figure 6: The relationship between the normal  $n$ , the light vector  $l$ , the view direction  $v$ , and the reflection vector  $r$ .

## RECAP

The traditional approach in real-time computer graphics has been to calculate lighting at a vertex as a sum of the ambient, diffuse, and specular light.

$$i_{\text{total}} = k_a i_a + \sum (k_d i_d + k_s i_s)$$

This is called a local lighting model since the only light on a vertex is from a light source, not from other objects.

That is, lights are lights, not objects!

The reflection coefficients are in the  $[0,1]$  range and are specified as part of the material property

Let's take a look at the individual parts of the traditional lighting pipeline.

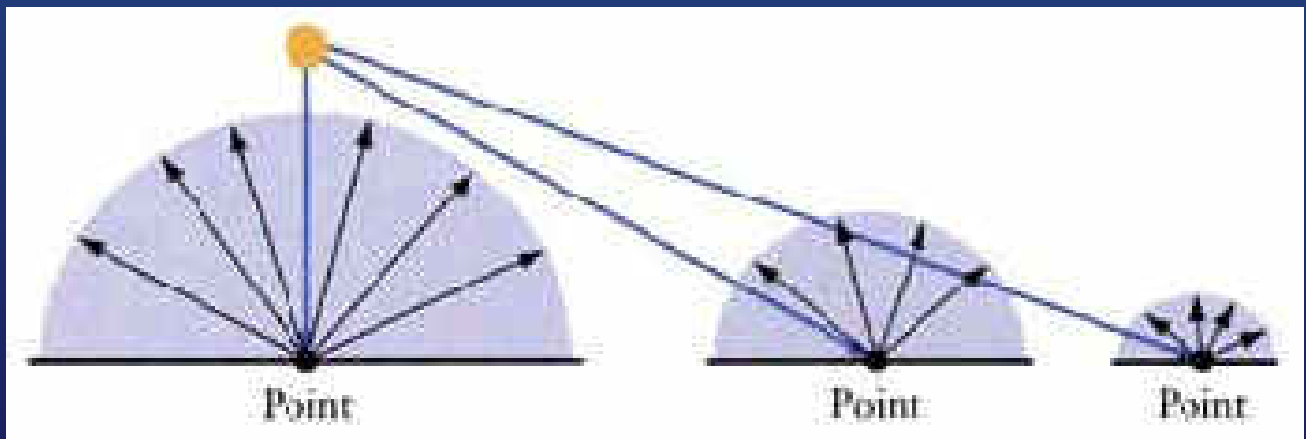
## **Ambient Light**



Figure 1: Ambient light provides illumination, but no surface details.

## Diffuse Light





Diffuse light decreases as the angle between the light vector and the surface normal increases.

$$\vec{r}_d = (\hat{n} \cdot \hat{l})(m_d \otimes s_d)$$

The equation for calculating diffuse lighting is

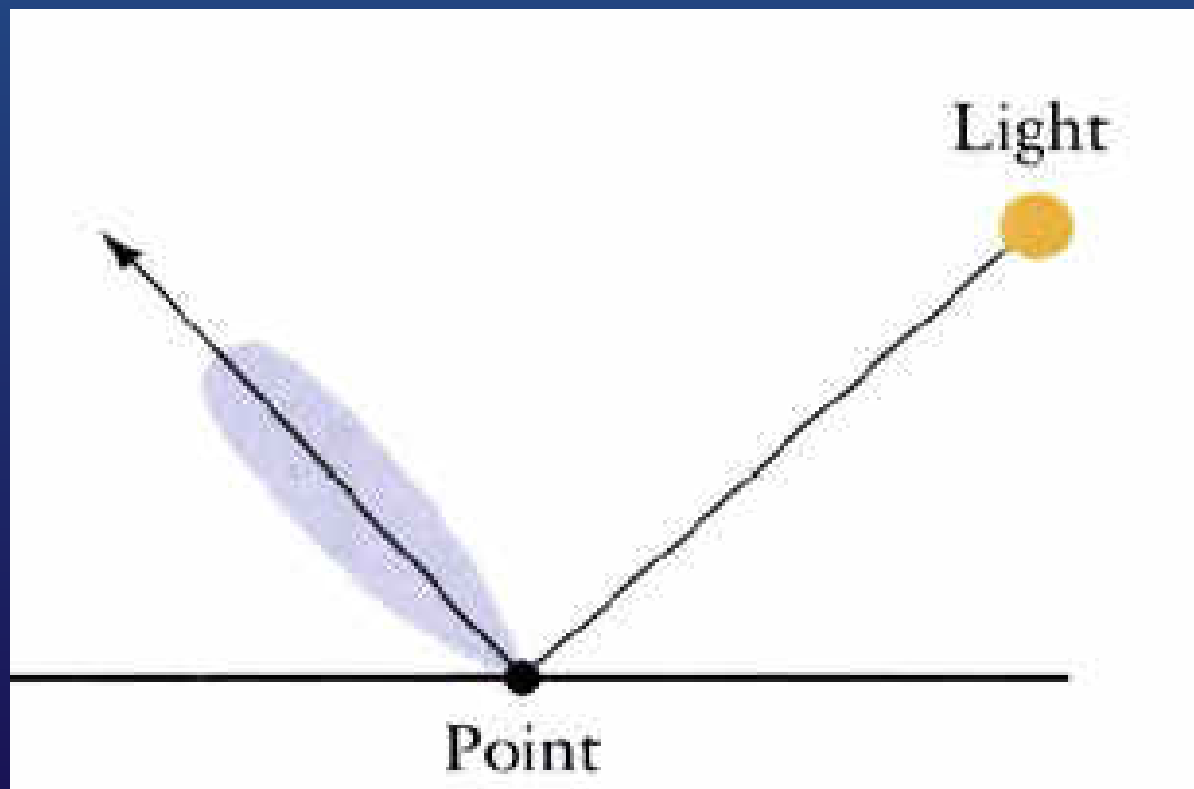


Diffuse light shading brings out some surface details



Ambient and Diffuse light together give a much better shading

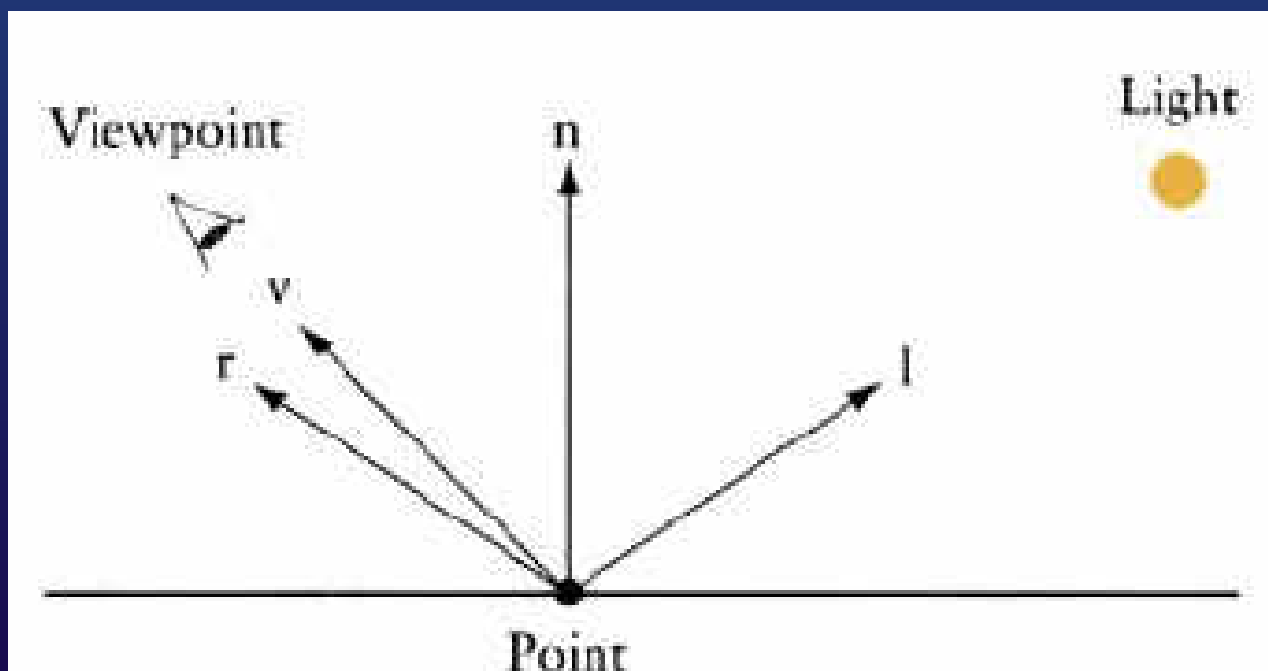
# Specular Light



Specular light's intensity follows the reflection vector.

Phong's equation for specular lighting is

$$I_s = (m_s \otimes s_s)(\hat{r} \bullet \hat{v})^{m_s} \quad (\text{Phong})$$



# Computer Graphics

## Lecture 29

# Computer Graphics

## Lecture 30

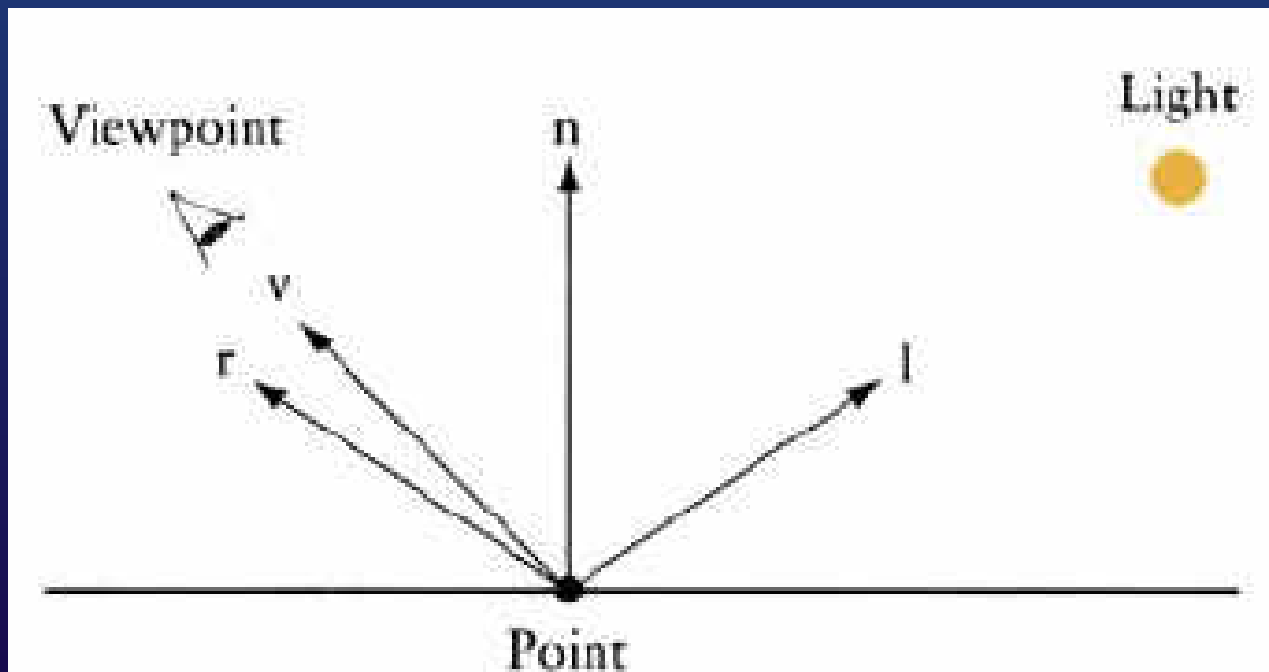
# Mathematics of Lighting and Shading - IV

Taqdees A. Siddiqi  
cs602@vu.edu.pk

# The Phong's Specular Light Equation

Phong's equation for specular lighting is

$$I_s = (m_s \otimes s_s) (\hat{r} \bullet \hat{v})^{m_s}$$



The more the view direction,  $v$ , is aligned with the reflection direction,  $r$ , the brighter the specular light will be.

Note the introduction of the  $ms$  term (shininess value), which is a power term that attempts to approximate the distribution of specular light reflection.

This can be seen in the next Figure, which shows values of  $(\hat{\mathbf{r}} \cdot \hat{\mathbf{v}})^m$  for values of 'm' ranging from 1 to 128. As we can see, the specular highlights get narrower for higher values, but they don't get any brighter.

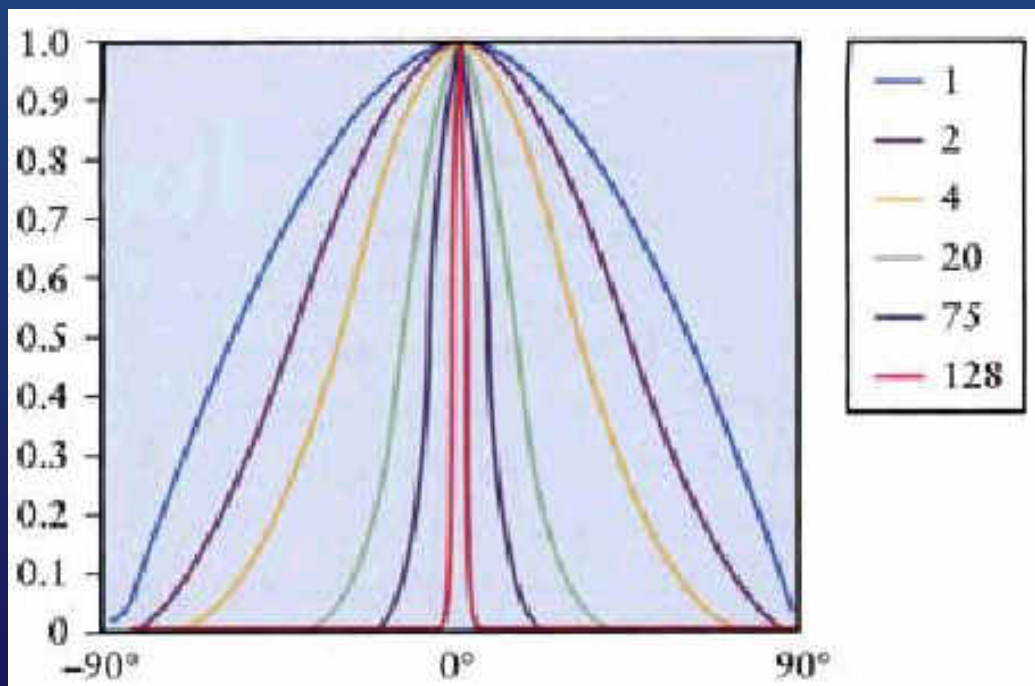


Figure: Phong's specular term for various values of the "shininess" term. Note that the values never get above 1.



To calculate  $r$  we can use the following equation:

$$r = \frac{2(n \bullet l)n - l}{|n|^2}$$

If  $l$  and  $n$  are normalized, then the resulting  $r$  is normalized and the equation can be simplified as:

$$\hat{r} = 2(\hat{n} \bullet \hat{l})\hat{n} - \hat{l}$$

And just as we did for diffuse lighting, if the dot product is negative, then the term is ignored.

$$i_s = \text{MAX}\left(0, (\hat{r} \bullet \hat{v})^{m_s} (m_s \otimes s_s)\right)$$



Figure: A specular term just shows the highlights.

## The Lighting Equation

Now that we have computed the various light contributions to our final color value, we can add them up to get the final color value.

Note that the final color values will have to be made to fit in the  $[0,1]$  range for the final rgb values.

$$\hat{I}_{\text{total}} = I_a + \sum (r_d + r_s)$$

Our final scene with ambient, diffuse, and specular light contributions (with one white light above and to the left of the viewer) looks like Figure 1.



Figure 1: A combination of ambient, diffuse, and specular illumination.

Next we will discuss some refinements and alternative ways of calculating the various coefficients of the lighting equation.

## **Light Attenuation**

Light in the real world loses its intensity as the inverse square of the distance from the light source to the surface being illuminated

An empirical model was developed that seems to give satisfactory results. This is the attenuation model that's used in OpenGL and DirectX

The attenuation factor is calculated by the following equation:

$$f_{\text{atten}} = 1 / (k_c + k_l d + k_q d^2)$$

The fatten factor is the attenuation factor. The distance  $d$  between the light and the vertex is always positive.

where the  $k_c$ ,  $k_l$ , and  $k_q$  parameters are the constant, linear, and quadratic attenuation constants, respectively

The lighting equation with the attenuation factor looks like this.

$$I_{\text{total}} = I_a + \sum f_{\text{atten}}(I_d + I_s)$$



Figure 2 shows a sample of what attenuation looks like. This image is the same as the one shown in Figure 1, but with light attenuation added.



Figure 2: A scene with light attenuation. The white sphere is the light position.

## Schlick's Simplification for the Specular Exponential Term

Real-time graphics programmers are always looking for simplifications. We've probably gathered that there's no such thing as the "correct" lighting equation, just a series of hacks to make things look right with as little computational effort as possible

Schlick [SCHLICK 1994] suggested a replacement for the exponential term since that's a fairly expensive operation. If we define part of our specular light term as follows:

$$(S)^{m_s}$$

where  $S$  is either the Phong or Blinn-Phong flavor of the specular lighting equation, then Schlick's simplification is to replace the preceding part of the specular equation with:

$$\frac{S}{m_s - m_s S + S}$$

which eliminates the need for an exponential term.

At first glance, a plot of Schlick's function looks very similar to the exponential equation (Figure 3).

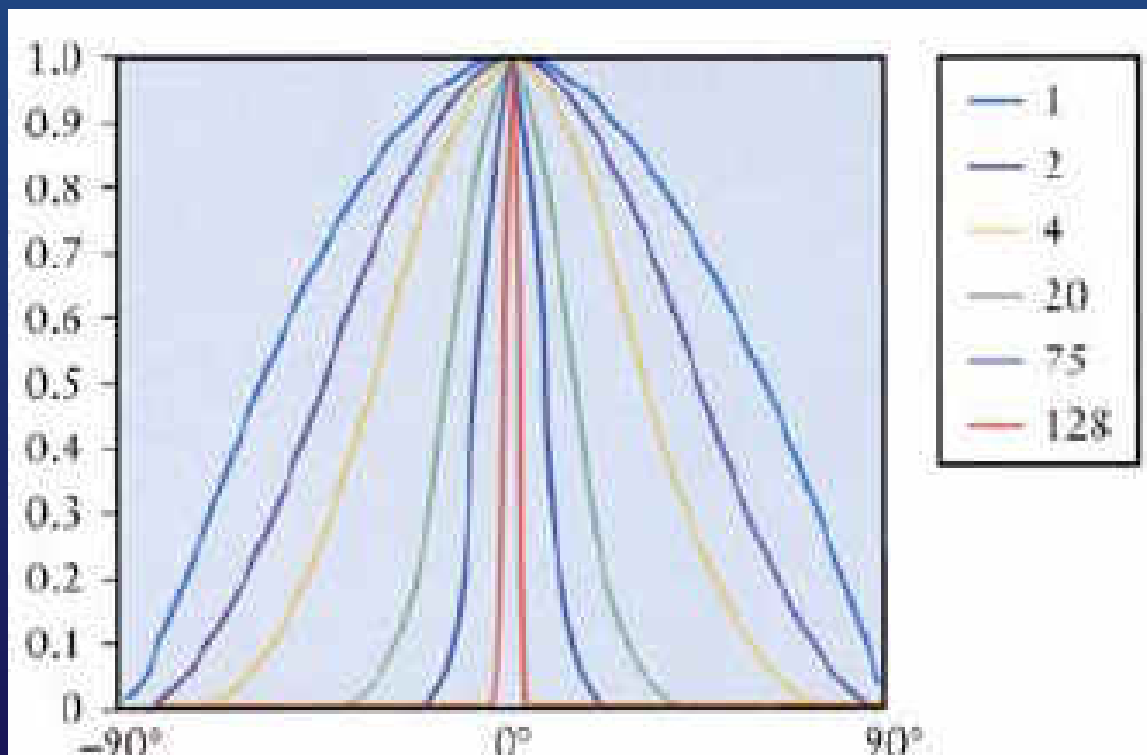


Figure 3: Schlick's term for specular looks very much like the more expensive Phong term.

Schlick's simplification works. The blue values are Schlick's, and the red are the exponential plot. As the view and light angles get closer (i.e., get closer to zero on the x axis), we can see that the values of the curves are quite close.

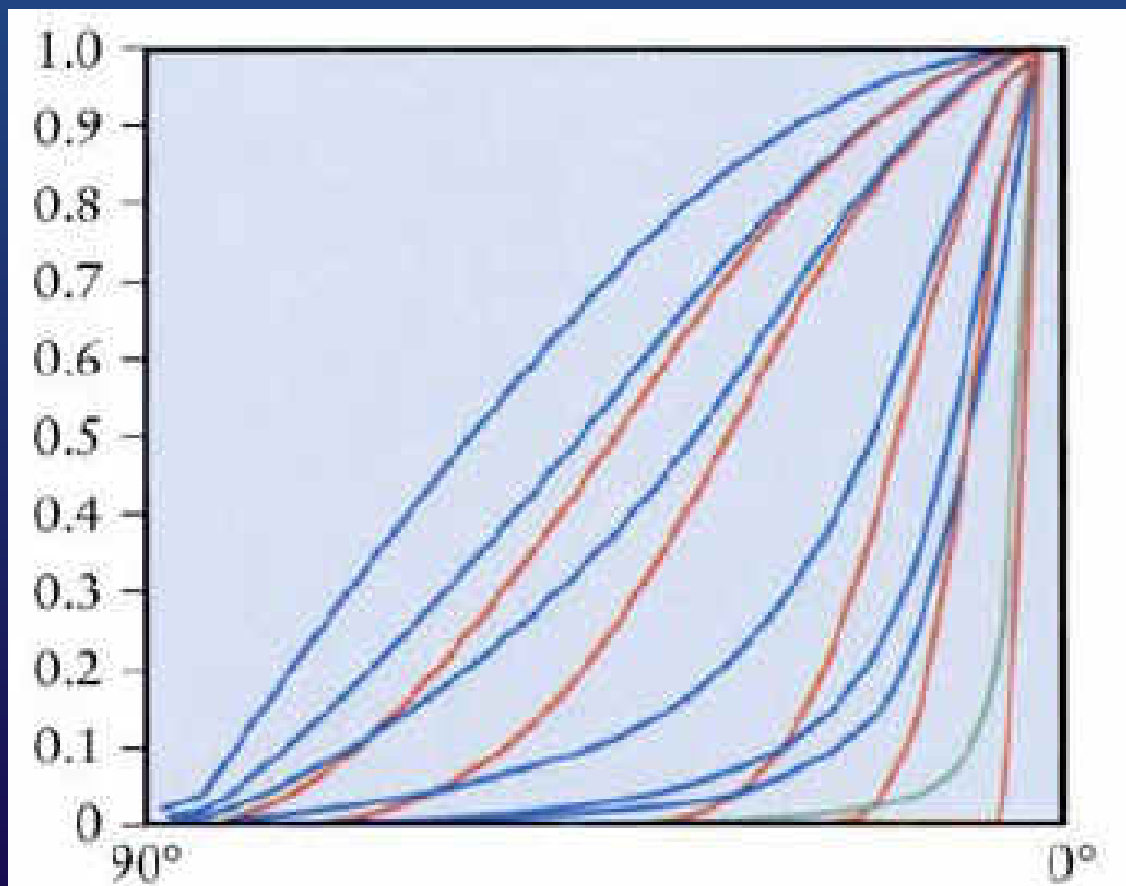


Figure 4: Schlick's vs. Phong's specular terms.

## Oren—Nayar Diffuse Reflection

One of the problems of the standard Lambertian model is that it considers the surface as a smooth diffuse surface.

The classic example of this is a full moon.

it's pretty obvious that this doesn't follow the Lambertian distribution—if it did, the edges of the moon would be in near darkness



Figure 5: The full moon is an good example of something that doesn't show Lambertian diffuse shading.



The effect we're looking at is called backscattering.

Backscattering is when a rough surface bounces around a light ray and then reflects the ray in the direction the light originally came from

Note that there is a similar but different effect called retroreflection

Retroreflection is the effect of reflecting light towards the direction from which it came, no matter the orientation of the surface

This is the same effect that we see on bicycle reflectors. However, this is due to the design of the surface features (made up of vshaped or spherical reflectors) rather than a scattering effect.

when the light direction is closer to the view direction,  
we get the effect of forward scattering

Forward scattering is just backscattering from a different direction

Figures 6 and 7 show the same surfaces demonstrating backscattering and forward scattering.

Both the dirt field in Figure 6 and the soybean field in Figure 7 can be considered rough diffuse reflecting surfaces.



Figure 6: The same dirt field showing widely differing reflection properties.



Figure 7: A soybean field showing differing reflection properties.

Notice how the backscattering image shows a near uniform diffuse illumination, whereas the forward scattering image shows a uniform dull diffuse illumination.

The Oren—Nayar diffuse shading model looks like this.

$$i_d = \frac{\rho}{\pi} E_{\Omega} \cos(\theta_i) (A + B \max[0, \cos(\phi_r - \phi_i)] \sin(\alpha) \tan(\beta))$$

Where

$$A = 1 - 0.5 \frac{\sigma^2}{\sigma^2 + 0.33}$$

$$B = 0.45 \frac{\sigma^2}{\sigma^2 + 0.09}$$

$\rho/\pi$  is a surface reflectivity property, which we can replace with our surface diffuse color.  $E_0$  is a light input energy term, which we can replace with our light diffuse color.

And the  $\theta_i$  term is just our familiar angle between the vertex normal and the light direction. Making these exchanges gives us

$$i_d = (m_d \otimes s_d)(\hat{n} \cdot \hat{l})(A + B \max[0, \cos(\phi_r - \phi_l)] \sin(\alpha) \tan(\beta))$$



which looks a lot more like the equations we've used.  
There are still some parameters to explain.

$\sigma$  is the surface roughness parameter. It's the standard deviation in radians of the angle of distribution of the microfacets in the surface roughness model. The larger the value, the rougher the surface.

$\theta_r$  is the angle between the vertex normal and the view direction.

$\varphi_r - \varphi_i$  is the circular angle (about the vertex normal) between the light vector and the view vector.

$\alpha$  is  $\max(\theta_i, \theta_r)$ .

$\beta$  is  $\min(\theta_i, \theta_r)$ .

Note that if the roughness value is zero, the model is the same as the Lambertian diffuse model. Oren and Nayar also noted that we can replace the value 0.33 in coefficient  $A$  with 0.57 to better account for surface interreflection.

## RECAP

Phong's equation for specular lighting is

$$I_s = (m_s \otimes s_s) (\hat{r} \bullet \hat{v})^{m_s}$$

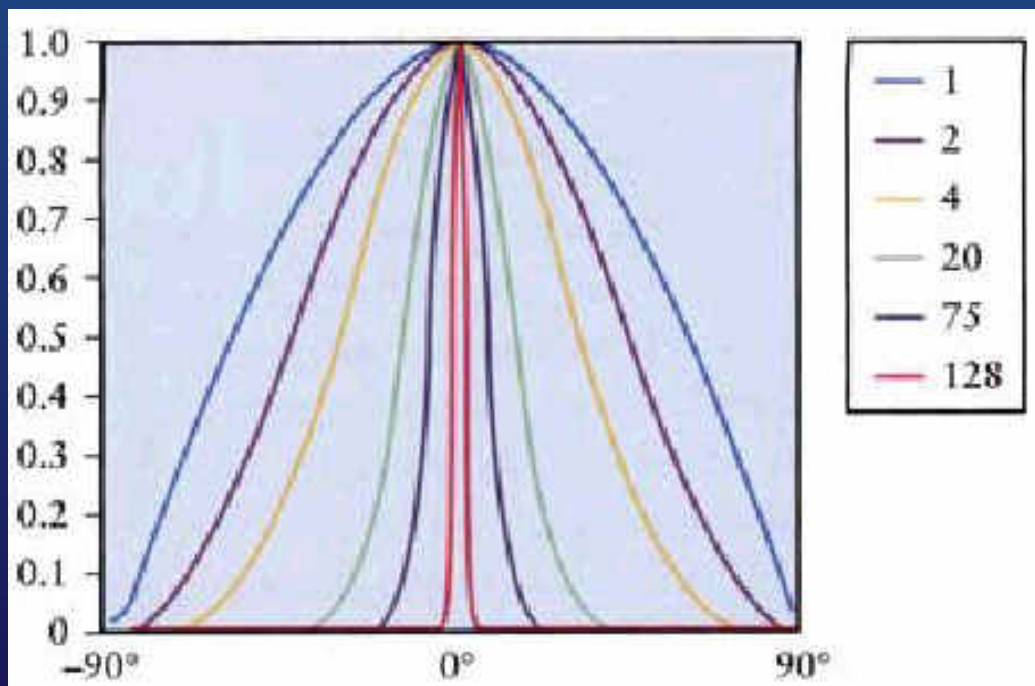


Figure: Phong's specular term for various values of the "shininess" term. Note that the values never get above 1.

To calculate  $r$  we can use the following equation:

$$r = \frac{2(n \bullet l)n - l}{|n|^2}$$

If  $l$  and  $n$  are normalized, then the resulting  $r$  is normalized and the equation can be simplified as:

$$\hat{r} = 2(\hat{n} \bullet \hat{l})\hat{n} - \hat{l}$$

And just as we did for diffuse lighting, if the dot product is negative, then the term is ignored.

$$i_s = \text{MAX}\left(0, (\hat{r} \bullet \hat{v})^{m_s} (m_s \otimes s_s)\right)$$



Figure: A specular term just shows the highlights.

Note that the final color values will have to be made to fit in the  $[0,1]$  range for the final rgb values.

$$\hat{I}_{\text{total}} = I_a + \sum (r_d + r_s)$$



Figure 1: A combination of ambient, diffuse, and specular illumination.

## Light Attenuation

Light in the real world loses its intensity as the inverse square of the distance from the light source to the surface being illuminated

The attenuation factor is calculated by the following equation:

$$f_{\text{atten}} = 1 / (k_c + k_l d + k_q d^2)$$



The fatten factor is the attenuation factor. The distance  $d$  between the light and the vertex is always positive.

where the  $k_c$ ,  $k_l$ , and  $k_q$  parameters are the constant, linear, and quadratic attenuation constants, respectively

The lighting equation with the attenuation factor looks like this.

$$I_{\text{total}} = I_a + \sum f_{\text{atten}}(I_d + I_s)$$



Figure 2: A scene with light attenuation. The white sphere is the light position.

Schlick [SCHLICK 1994] suggested a replacement for the exponential term since that's a fairly expensive operation. If we define part of our specular light term as follows:

$$(S)^{n_s}$$

where  $S$  is either the Phong or Blinn-Phong flavor of the specular lighting equation, then Schlick's simplification is to replace the preceding part of the specular equation with:

$$\frac{S}{n_s - n_s S + S}$$

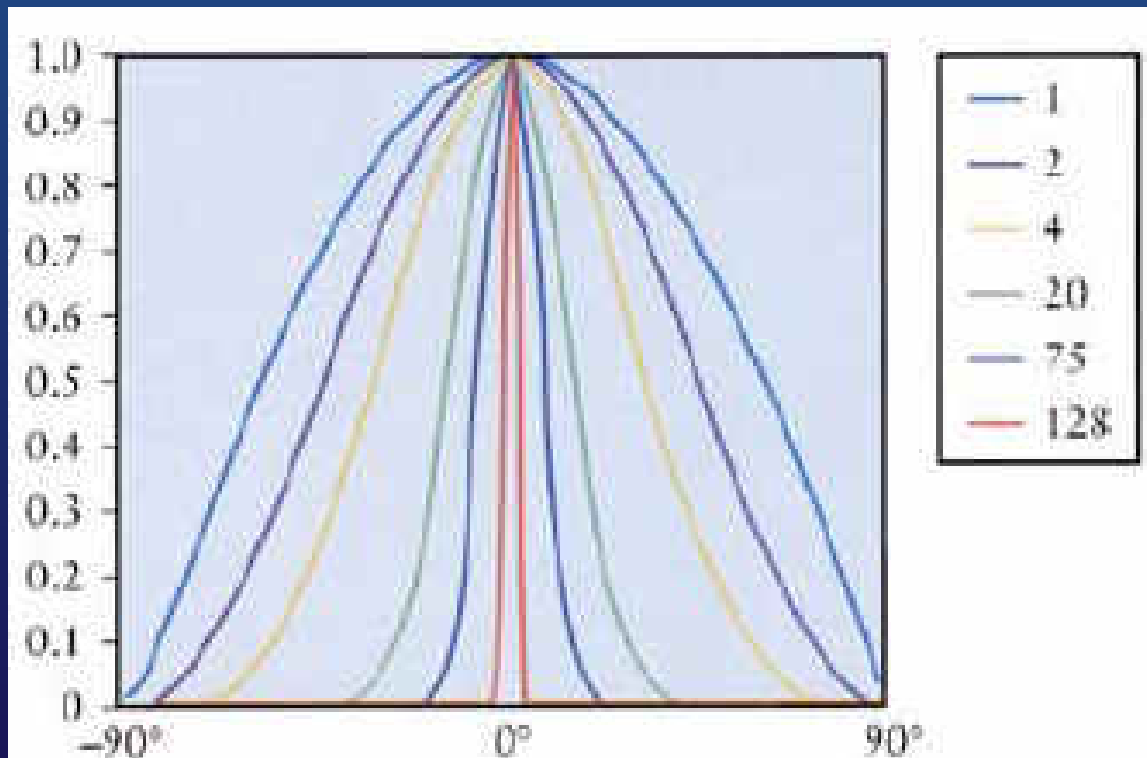


Figure 3: Schlick's term for specular looks very much like the more expensive Phong term.

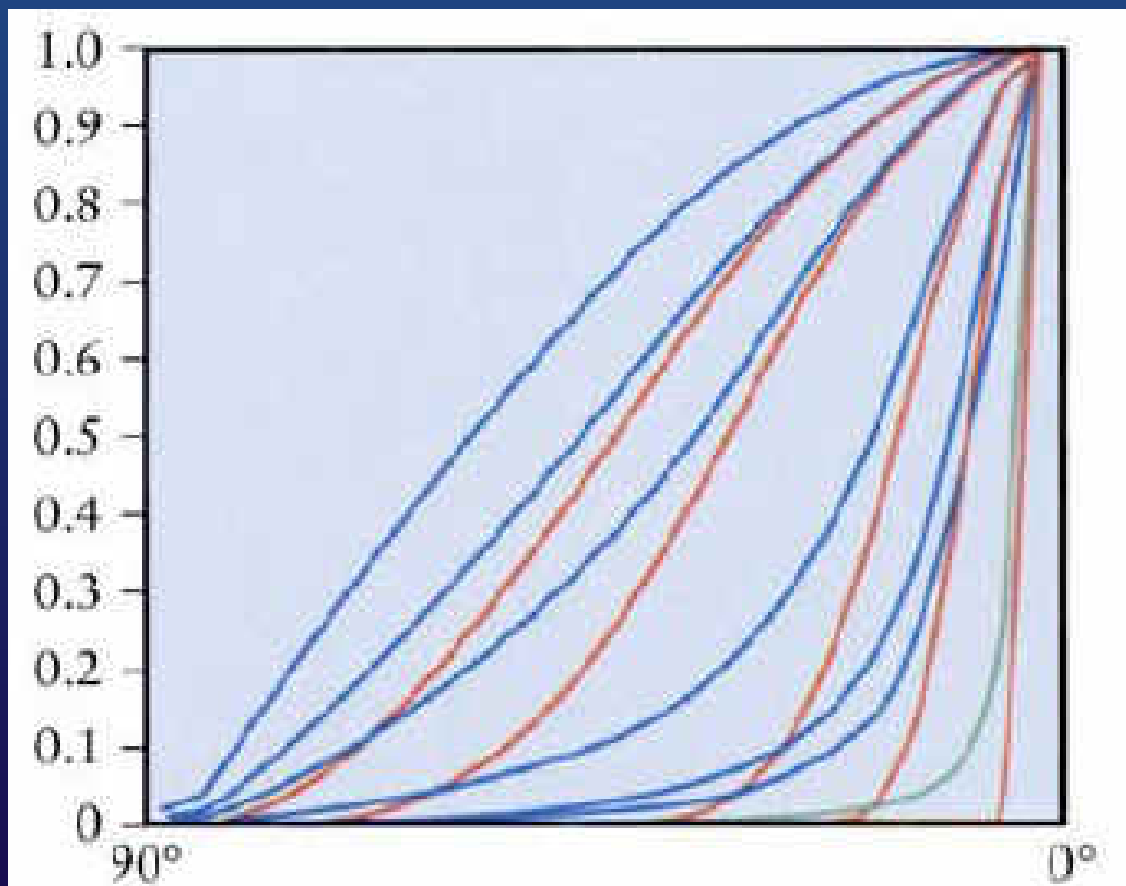


Figure 4: Schlick's vs. Phong's specular terms.

# Oren—Nayar Diffuse Reflection

One of the problems of the standard Lambertian model is that it considers the surface as a smooth diffuse surface.



Backscattering is when a rough surface bounces around a light ray and then reflects the ray in the direction the light originally came from

Retroreflection is the effect of reflecting light towards the direction from which it came, no matter the orientation of the surface

when the light direction is closer to the view direction, we get the effect of forward scattering



Figure 6: The same dirt field showing widely differing reflection properties.



Figure 7: A soybean field showing differing reflection properties.



$$i_d = \frac{\rho}{\pi} E_0 \cos(\theta_i) (A + B \max[0, \cos(\phi_r - \phi_i)] \sin(\alpha) \tan(\beta))$$

Where

$$A = 1 - 0.5 \frac{\sigma^2}{\sigma^2 + 0.33}$$

$$B = 0.45 \frac{\sigma^2}{\sigma^2 + 0.09}$$

$\rho/\pi$  is a surface reflectivity property, which we can replace with our surface diffuse color.  $E_0$  is a light input energy term, which we can replace with our light diffuse color.

And the  $\theta_i$  term is just our familiar angle between the vertex normal and the light direction. Making these exchanges gives us

$$i_d = (m_d \otimes s_d)(\hat{n} \cdot \hat{l})(A + B \max[0, \cos(\phi_r - \phi_l)] \sin(\alpha) \tan(\beta))$$

which looks a lot more like the equations we've used. There are still some parameters to explain.

$\sigma$  is the surface roughness parameter. It's the standard deviation in radians of the angle of distribution of the microfacets in the surface roughness model. The larger the value, the rougher the surface.

$\theta_r$  is the angle between the vertex normal and the view direction.

$\varphi_r - \varphi_i$  is the circular angle (about the vertex normal)  
between the light vector and the view vector.

$\alpha$  is  $\max(\theta_i, \theta_r)$ .

$\beta$  is  $\min(\theta_i, \theta_r)$ .

**Computer Graphics**  
**Lecture 30**

**Computer**  
**Graphics**  
**Lecture 31**

# Mathematics of Lighting and Shading - V

Taqdees A. Siddiqi  
[cs602@vu.edu.pk](mailto:cs602@vu.edu.pk)

**Physically Based  
Illumination**

**In order to get a more realistic representation of lighting, we need to move away from the simplistic models that are found hard coded in most graphics pipelines.**

**and move to something that is based more in a physical representation of light as a wave with properties of its own that can interact with its environment**

**To do this, we'll need to understand how light passes through a medium and how hitting the boundary layer at the intersection of two media can affect light's properties**

**In Figure 1, there's an incident light hitting a surface. At the boundary of the two media (in this case, air and glass), there are two resulting rays of light.**



The reflected ray is the one that we've already discussed to some extent, and the other ray is the refracted or transmitted ray.

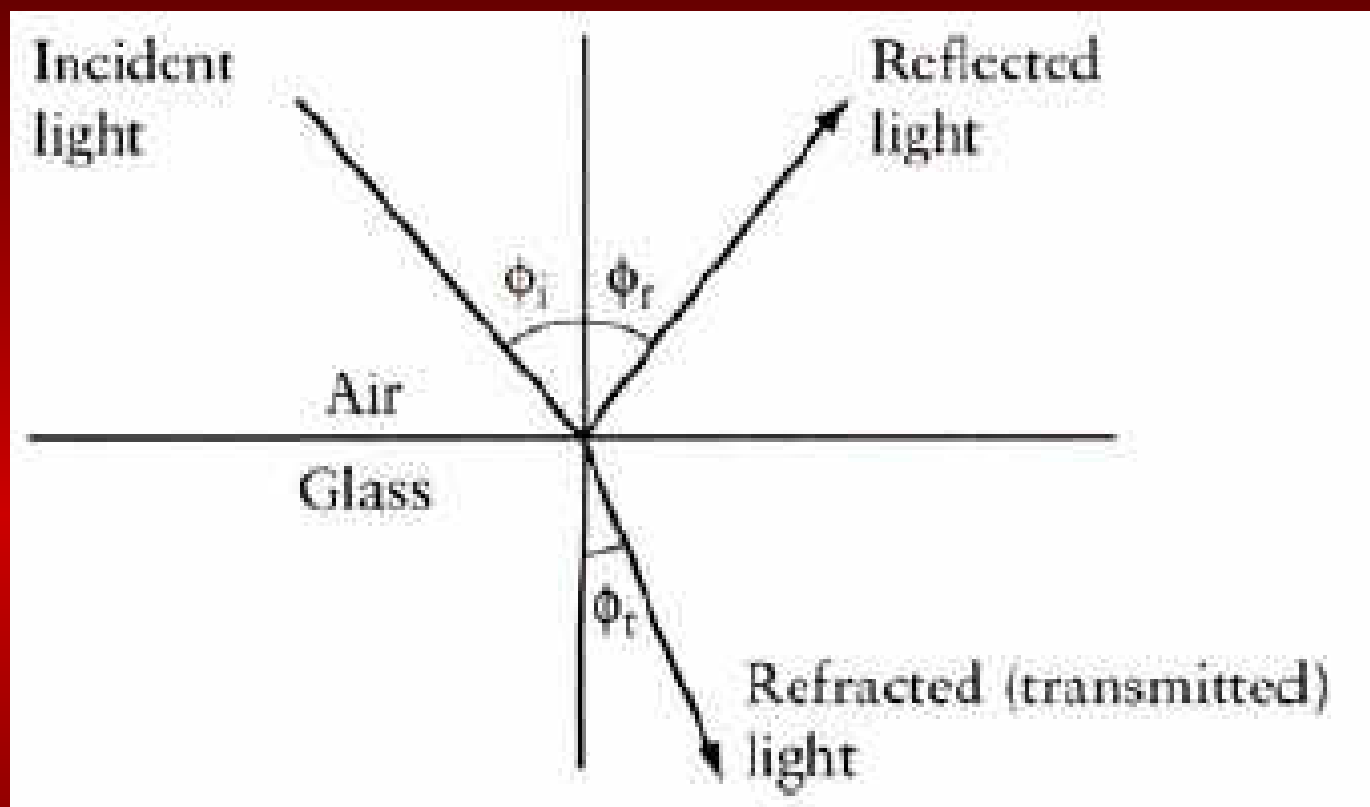


Figure 1: Light being reflected and refracted through a boundary.

**Until now, we've been treating our surfaces as perfectly smooth and uniform this prevents us from getting some interesting effects**

**In addition to examining the interaction of light with the surface boundary, we need a better description of real surface geometries**

**let's look at the physics of  
light interacting at a  
material boundary.**

## **Reflection**

**Reflection of a light wave is the change in direction of the light ray when it bounces off the boundary between two media**

**light wave reflecting off a perfectly smooth surface**

$$\phi_{\text{incident}} = \phi_{\text{reflected}}$$

**Using a lighting model based upon the Blinn—Phong model means that we'll always get a uniform specular highlight based upon the color of the reflecting light and material**

**In order to get a more interesting and realistic lighting model, we need to add in some nonlinear elements to our calculations**

# Refraction

**Refraction happens when  
a light wave goes from  
one medium into  
another.**

**Because of the difference in the speed of light of the media, light bends when it crosses the boundary. Snell's law gives the change in angles.**

$$n_a \sin(\phi_a) = n_b \sin(\phi_b)$$

where the  $n$ 's are the material's index of refraction

Snell's law states that when light refracts through a surface, the refracted angle is shifted by a function of the ratio of the two material's indices of refraction



**The index of refraction of vacuum is 1, and all other material's indices of refraction are greater than 1**

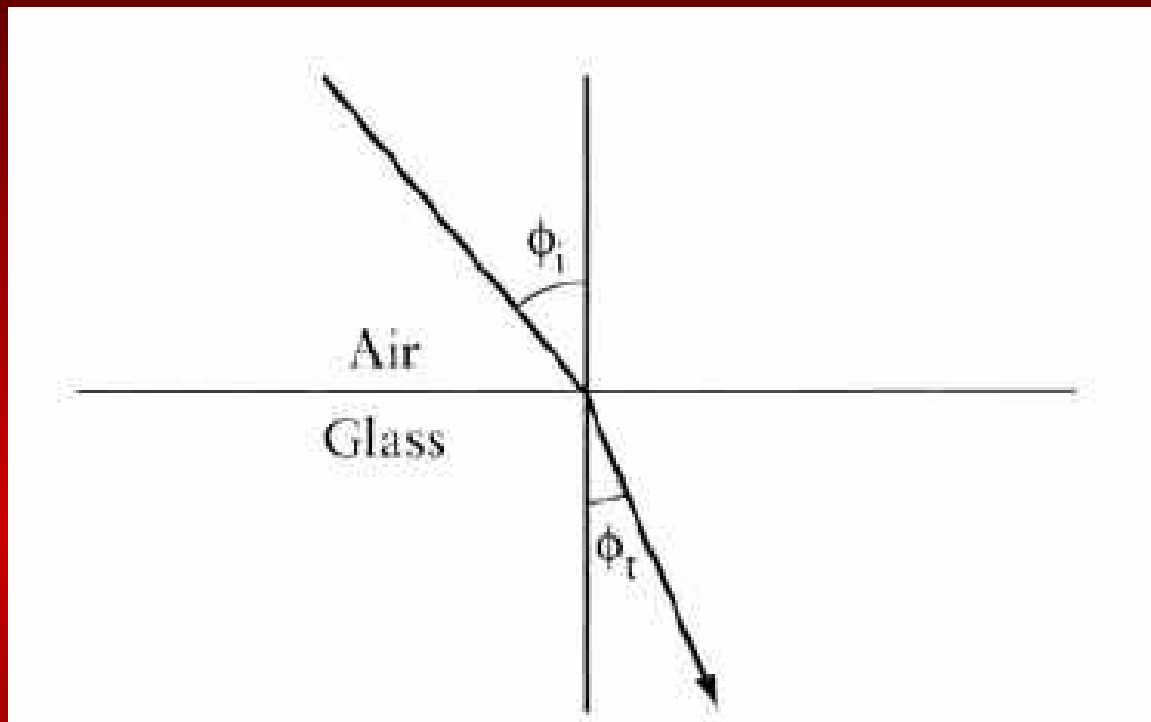


**Figure 2: The refracted ray's angle is less than the incoming ray's.**

The angle of incidence is the angle between the incoming vector and the surface. Rearranging Snell's law, we can solve for the refracted angle.

$$\left(\frac{n_{\text{air}}}{n_{\text{glass}}}\right) \sin(\phi_{\text{air}}) = \sin(\phi_{\text{glass}})$$

Inserting the values give us

$$\left(\frac{1.0}{1.5}\right) \sin(45^\circ) = \sin(\phi_{\text{glass}})$$

$$\phi_{\text{glass}} = 28.1^\circ$$

If we use Snell's law for light going from water to air, and plug in  $90^\circ$  for the refracted angle, we get  $41.8^\circ$  for the incident angle. This is called the critical angle at which we observe the phenomenon of *total internal reflection*

At any angle greater than this, light will not pass through a boundary but will be reflected internally

**One place that we get interesting visual properties is in the diamond—air interface.**

**The refractive index of a diamond is fairly high, 2.24, which means that it's got a very low critical angle, just  $24.4^\circ$ .**

**This means that a good portion of the light entering a diamond will bounce around the inside of the diamond hitting a number of air—diamond boundaries, and as long as the angle is  $24.4^\circ$  or greater, it will keep reflecting internally**

**This is why diamonds are cut to be relatively flatish on the top but with many faceted sides, so that light entering in one spot will bounce around and exit at another, giving rise to the sparkle normally associated with diamonds.**

**Another place where a small change in the indices of refraction occurs is on a road heated by the sun when viewed from far away**

**The hot air at the road's surface has a slightly smaller index of refraction than the denser, cooler air above it.**

**This is why we get the effect of a road looking as though it were covered with water and reflecting the image above it—the light waves are actually reflected off the warm air—cold air interface.**

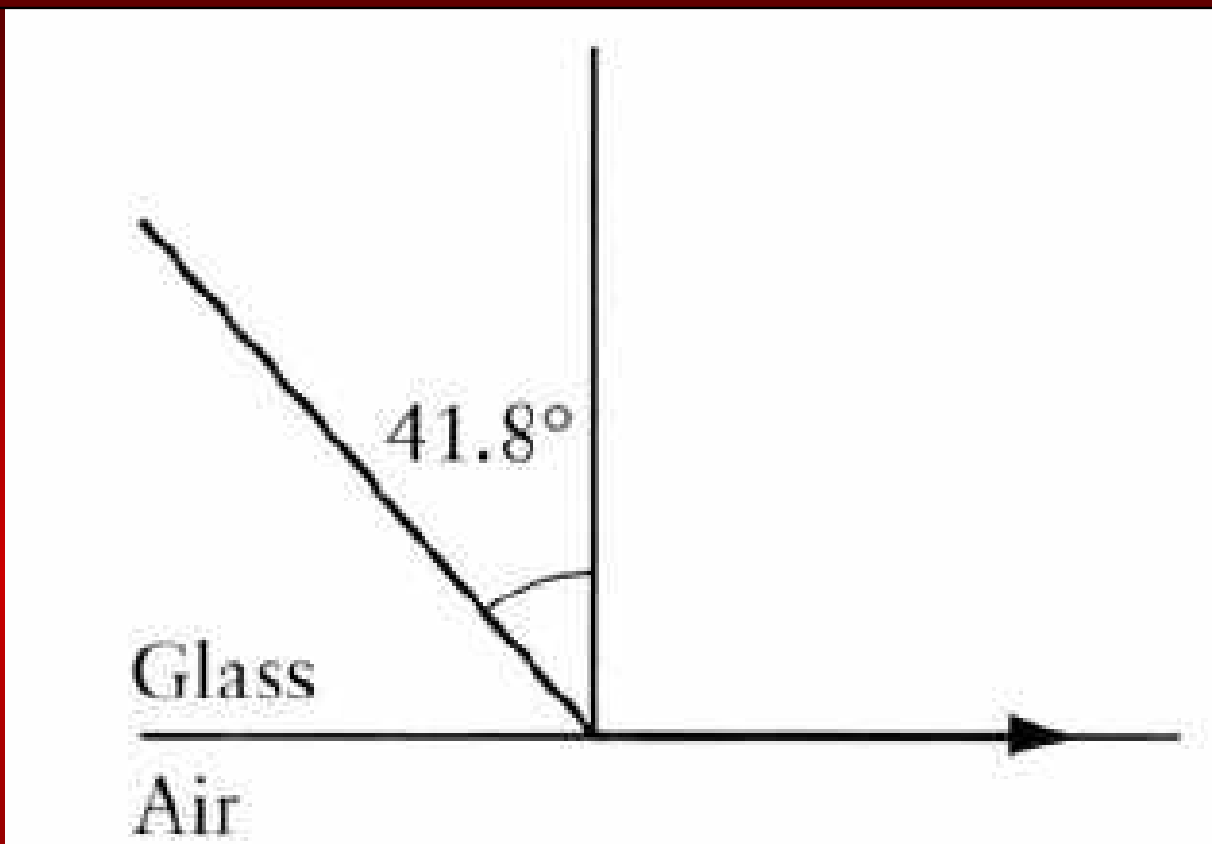


**Figure 3: The critical angle.**

**What makes this really challenging to model is that the index of refraction for most materials *is a function of the wavelength of the light***

**This means that not only is there a shift in the angle of refraction, but that the shift is different for differing wavelengths of light**



**Figure 4 and 5 show the index of refraction for fused quartz and sapphire plotted against the wavelength.**

**We can see the general trend that shorter wavelength light (bluish) tends to bend more than the longer (reddish) wavelengths.**

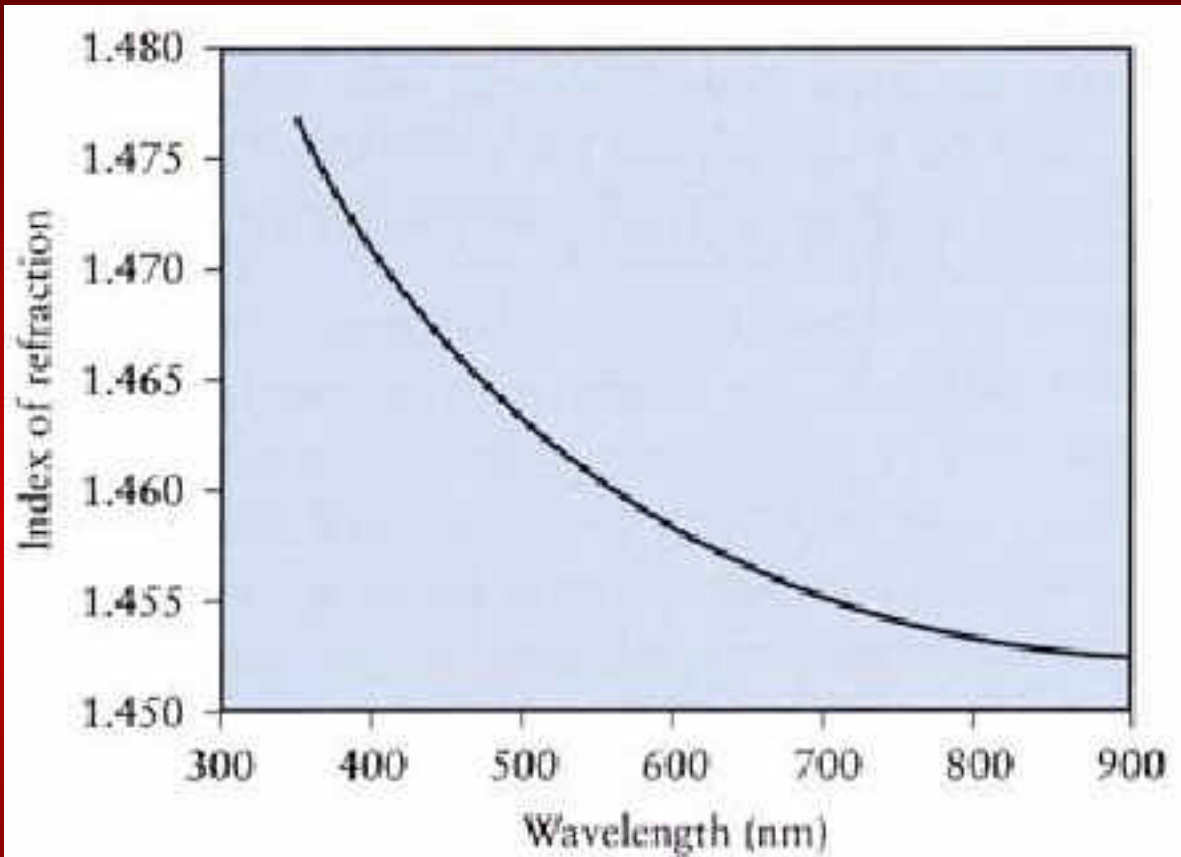


**Figure 4: Index of refraction as a function of wavelength for quartz.**

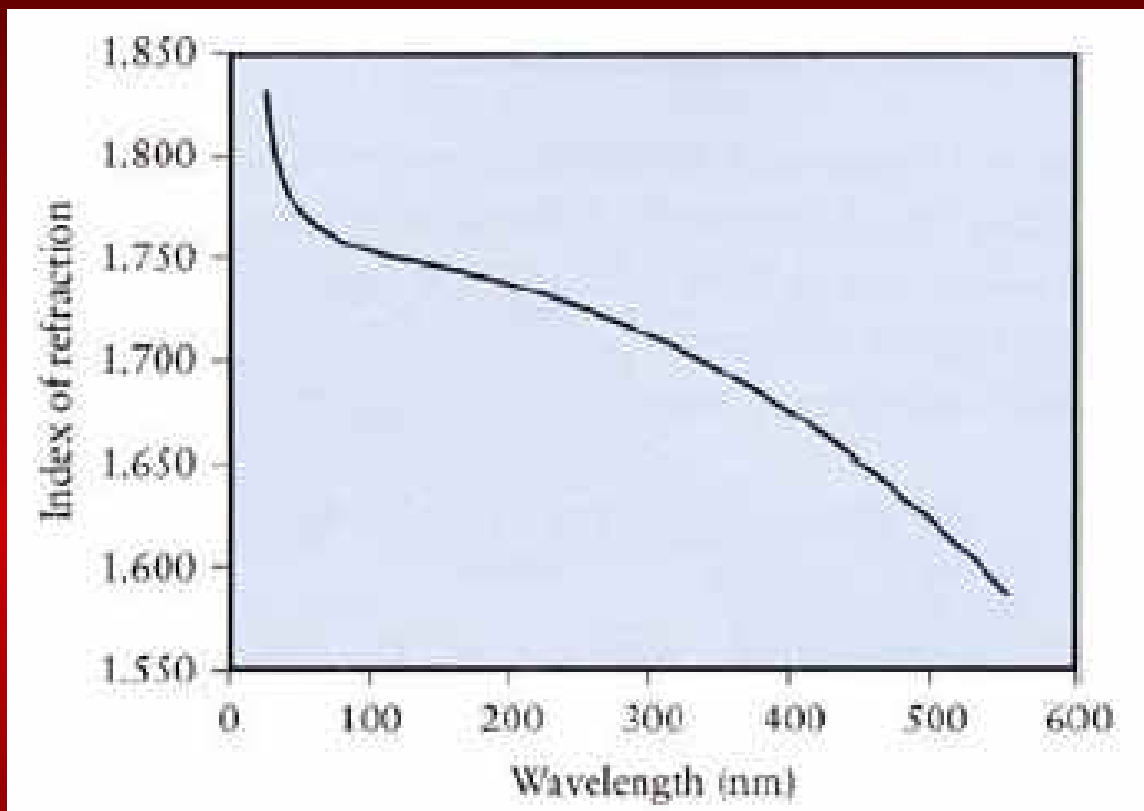


**Figure 5: Index of refraction as a function of wavelength for sapphire.**

**This is the phenomenon that's responsible for the spectrum that can be seen when white light is passed through a prism (Figure 6)**

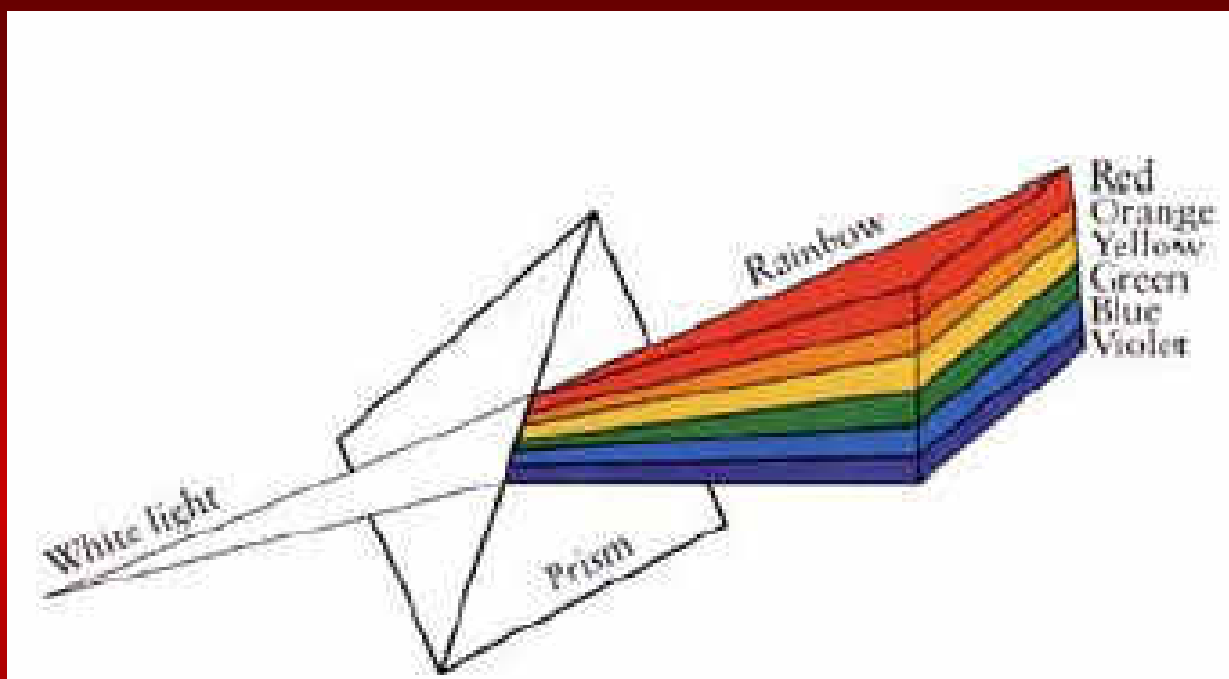


**Figure 6: The wavelength dependence of the index of refraction in action.**

**It's refraction that will break apart a light source into its component colors, not reflection.**

**This is one area where our simplistic model of light breaks down since we're not computing an entire spectrum of light waves, but we're limited to three primary colors.**

**For reference, the rgb values can be assigned to a range of wavelengths as follows:**

blue = 492 – 455nm

green = 577 – 492nm

red = 780 – 622nm

**While the spectrum spreading effect of refraction is interesting in itself, the rgb nature of computer color representation precludes performing this spreading directly—we can't break up a color value into multiple color values.**

**However, with some work, we can compute the shade of the color for a particular angle of refraction and then use that as the material color to influence the refracted color.**

# Computer Graphics Lecture 31

# Computer Graphics Lecture 32

# **Introduction to OpenGL**

**Taqdees A. Siddiqi**  
**cs602@vu.edu.pk**

- ◆ **Most Widely Adopted Graphics Standard**
- ◆ **Inception – 1992**
- ◆ **Multi-platform**
- ◆ **Rendering, texture mapping, special effects, and other powerful visualization functions**



# ◆ High-quality high-performance

## Markets:

- ◆ Broadcasting,
- ◆ CAD/CAM/CAE,
- ◆ Entertainment,
- ◆ Medical imaging, and
- ◆ Virtual reality

## Developer-Driven Advantages

- ◆ Industry standard
  - OpenGL Architecture Review Board (ARB)
  - Broad industry support,
  - The only truly open, vendor-neutral, multiplatform graphics standard
- ◆ Stable
  - Available for more than a decade
  - on a wide variety of platforms
  - Well controlled Additions to the specification,
  - Proposed updates announced in time
  - Backward compatibility

## Developer-Driven Advantages

- ◆ **Reliable and portable**
  - Consistent visual display results on any OpenGL API-compliant hardware regardless of operating system or windowing system
- ◆ **Evolving**
  - Thorough and forward-looking design,
  - Allows new hardware innovations to be accessible through the API (OpenGL extension mechanism)

## Developer-Driven Advantages

- ◆ **Scalable**
  - OpenGL API-based applications can run on :
    - consumer electronics
    - PCs
    - Workstations
    - Supercomputers.
  - As a result, applications can scale to any class of machine that the developer chooses to target.

## Developer-Driven Advantages

- ◆ **Easy to use**
  - Well structured
  - Intuitive design
  - Logical commands
  - Efficient routines
  - Compact programs/code
  - Hardware features independent coding possible
- ◆ **Well-documented**
  - Many 3<sup>rd</sup> party books
  - A lot of sample code

**As a software interface for graphics hardware, OpenGL renders multidimensional objects into a framebuffer.**

**OpenGL is industry standard graphics software with which programmers can create high-quality still and animated three-dimensional color images.**

## **Where Applicable:**

**OpenGL is built for compatibility across hardware and operating systems. This architecture makes it easy to port OpenGL programs from one system to another.**

# **Developer Audience:**

## **C/C++ programmers**

### **Run Time Requirements**

#### **OpenGL can run on:**

- **Linux**
- **All 32 bit versions of Microsoft Windows**

# Main purpose of OpenGL

Purpose of OpenGL is to render two- and three-dimensional objects into a framebuffer.

These objects are described as sequences of vertices (that define geometric objects) or pixels (that define images).

**OpenGL performs several processes on this data to convert it to pixels to form the final desired image in the framebuffer.**

**How OpenGL Works**

**The following topics present a global view of how OpenGL works:**

- **Primitives and Commands**
- **OpenGL Graphic Control**
- **Execution Model**
- **Basic OpenGL Operations**

**Primitives and  
Commands**



# **OpenGL draws primitive:**

- **points**
- **line segments**
- **polygons**

**OpenGL draw these primitives subject to several selectable modes.**

**User can control modes independently of one another.**

**Primitives are defined by a group of one or more *vertices*. A vertex defines:**

- **a point**
- **an end point of a line or**
- **a corner of a polygon where two edges meet**

**Following data is associated with a vertex:**

- **its coordinates**
- **colour**
- **normal**
- **texture coordinates and flags**

**Each vertex and its data is processed independently and in the same way.**

**Exceptions are cases in which the group of vertices must be clipped so that a particular primitive fits within a specified region.**

**In this case, vertex data may be modified and new vertices created.**

# Commands

**Commands are always processed in order. Though there may be an indeterminate delay before a command takes effect.**

**Each primitive is completed before next command**

**Therefore state-querying commands return inconsistent data.**

# **OpenGL Graphic Control**

**OpenGL provides you with fairly direct control over the fundamental operations of two- and three-dimensional graphics.**

**This includes specification of such parameters as:**

- **transformation matrices**
- **lighting equation coefficients**
- **antialiasing methods and**
- **pixel-update operators**

**OpenGL commands specify how a certain result should be produced (The procedure to be followed).**

**OpenGL is fundamentally procedural rather than descriptive.**

# **Execution Model**

## **Client/server model**

**Application code issues commands and these are processed by OpenGL.**

**OpenGL is network-transparent.**

**A server can maintain several OpenGL contexts. A client can connect to any context.**

**Network protocol may be an already existing protocol (as X Window System) or an independent protocol.**

**No OpenGL commands are provided for obtaining user input.**



**The window system that allocates framebuffer resources ultimately controls the effects of OpenGL commands on the framebuffer.**

**The window system:**

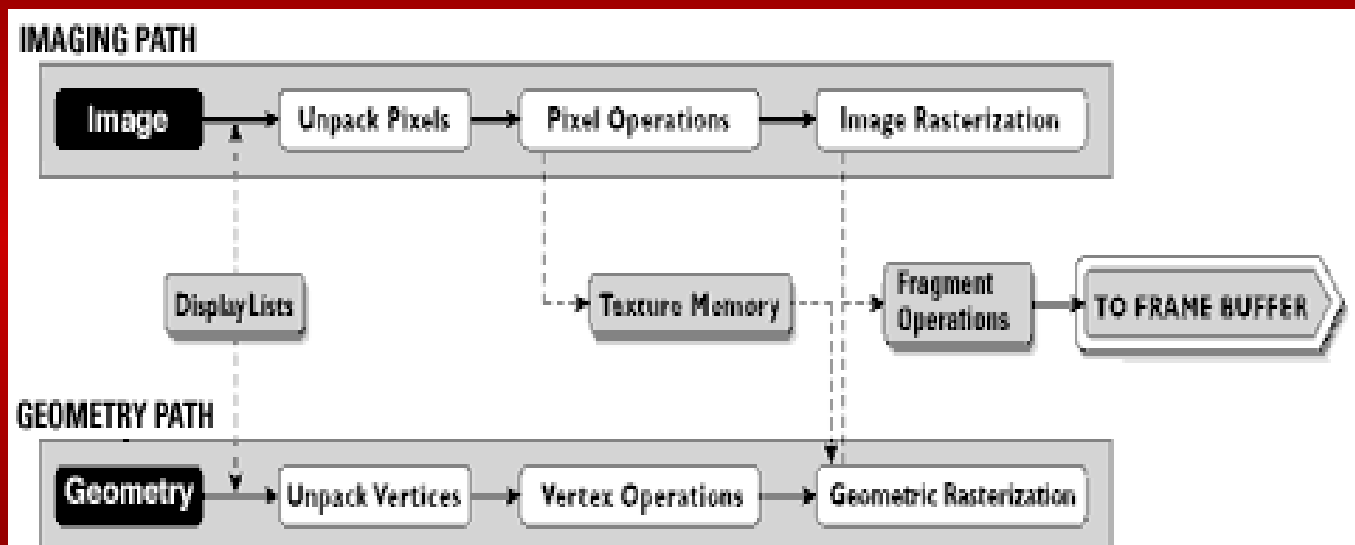
- ◆ **determines the portions to be accessed at any given time**
- ◆ **communicates to OpenGL how those portions are structured**

**No OpenGL commands to configure the framebuffer or initialize OpenGL.**

**Configuration is done outside of OpenGL in conjunction with the window system.**

## **Basic OpenGL Operation**

# The OpenGL Visualization Programming Pipeline



*OpenGL operates on image data as well as geometric primitives*

## Display list

Rather than having all commands proceed immediately through the pipeline, you can choose to accumulate some of them in a display list for processing later.

# Evaluator

**Provides an efficient way to approximate curve and surface geometry by evaluating polynomial commands of input values.**

**Per-vertex operations and primitive assembly**

**OpenGL processes geometric primitives—points.**

**Vertices are transformed and lit, and primitives are clipped to the viewport in preparation for rasterization.**

# Rasterization

The rasterization stage produces a series of frame-buffer addresses and associated values using a two-dimensional description of a point, line segment, or polygon.

## Rasterization

Each fragment so produced is fed into the last stage, per-fragment operations

# **Per-fragment operations**

**These are the final operations performed on the data before it's stored as pixels in the framebuffer.**

**Per-fragment operations include conditional updates to the framebuffer based on incoming and previously stored z values.**

**for z buffering and blending of incoming pixel colors with stored colors, as well as masking and other logical operations on pixel values**

**Data can be input in the form of pixels rather than vertices.**

**Following pixel operations, the pixel data is either:**

- **Stored as texture memory, for use in the rasterization stage.**

**Resulting fragments merged into the framebuffer just as if they were generated from geometric data.**

## **OpenGL Processing Pipeline**



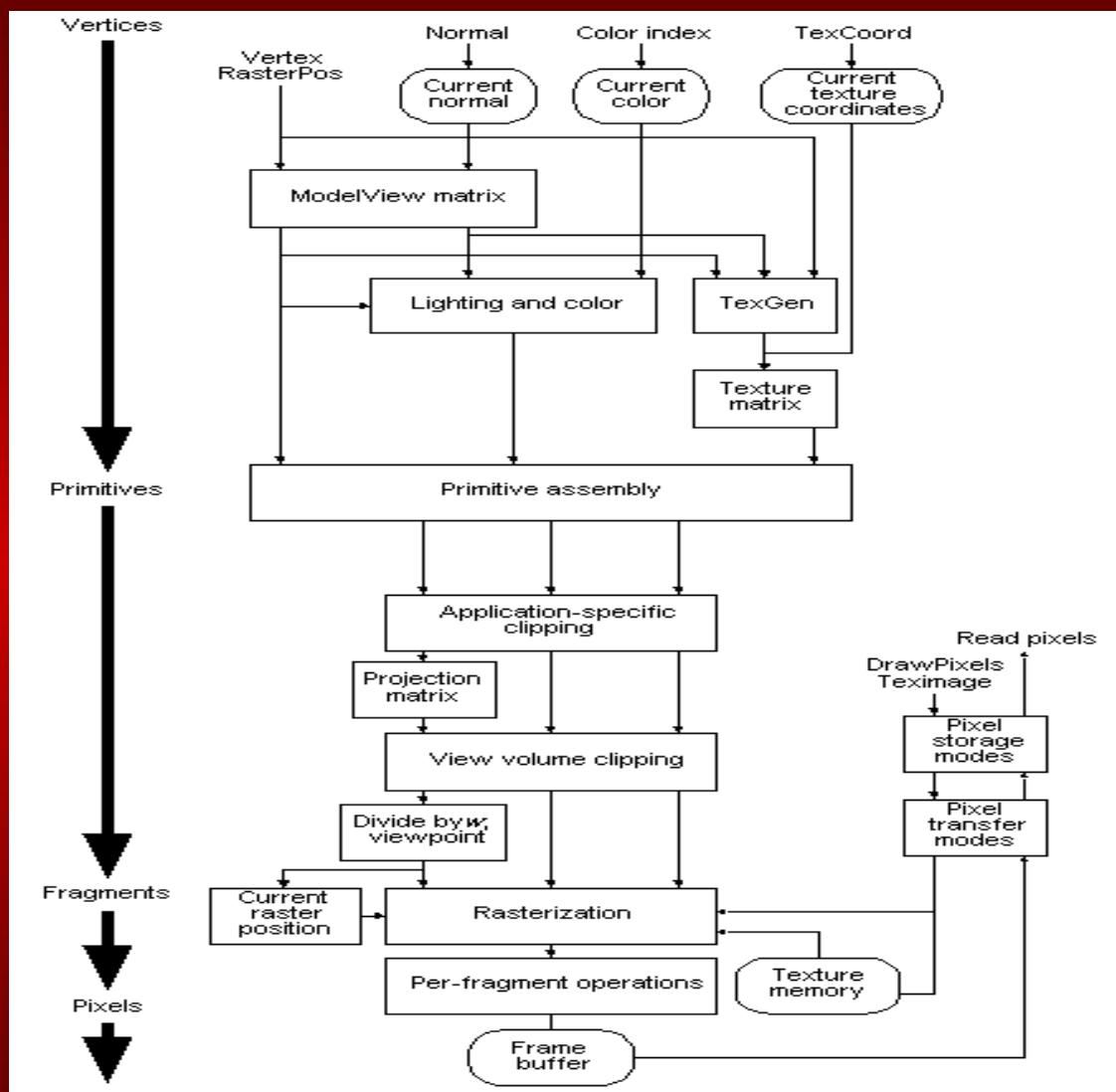
**Many OpenGL functions are used specifically for drawing objects such as points, lines, polygons, and bitmaps.**

**Some functions control the way that some of this drawing occurs (such as those that enable antialiasing or texturing). Other functions are specifically concerned with framebuffer manipulation.**

The topics discussed here describe how all of the OpenGL functions work together to create the OpenGL processing pipeline. Here we also take a closer look at the stages in which data is actually processed, and tie these stages to OpenGL functions.

The following diagram details the OpenGL processing pipeline. For most of the pipeline, you can see three vertical arrows between the major stages.

These arrows represent vertices and the two primary types of data that can be associated with vertices: color values and texture coordinates.



# Computer Graphics Lecture 32

# Computer Graphics Lecture 33

# **OpenGL Programming**

**Taqdees A. Siddiqi**  
**cs602@vu.edu.pk**

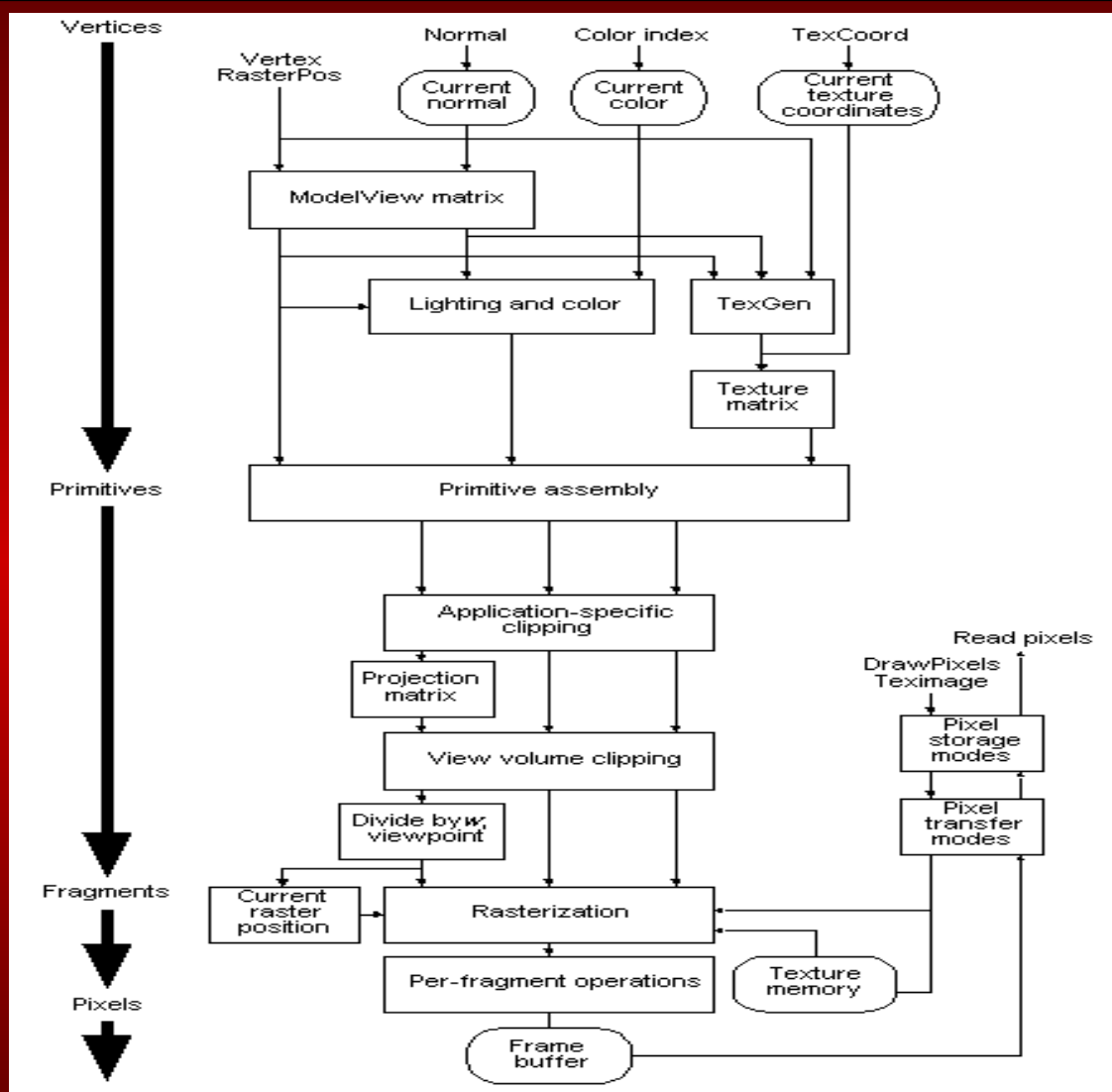
**Many OpenGL functions are used specifically for drawing objects such as points, lines, polygons, and bitmaps.**

**Some functions control the way that some of this drawing occurs (such as those that enable antialiasing or texturing). Other functions are specifically concerned with framebuffer manipulation.**

**The topics discussed here describe how all of the OpenGL functions work together to create the OpenGL processing pipeline. Here we also take a closer look at the stages in which data is actually processed, and tie these stages to OpenGL functions.**

**The following diagram details the OpenGL processing pipeline. For most of the pipeline, you can see three vertical arrows between the major stages.**

**These arrows represent vertices and the two primary types of data that can be associated with vertices: color values and texture coordinates.**



**Writing a program, using OpenGL, for any of the operating system we can use OpenGL Utility Library name glut.**



**glut is platform independent.  
glut can:**

- **create window**
- **get keyboard input and**
- **run an event handler or**
- **message loop of application**

**Functions start with the prefix “gl” are the core OpenGL functions and those start with “glu” or “glut” are the OpenGL utility library functions.**

**Let's write a program using “glut” and also OpenGL function to create graphics.**

```
#include <GL/glut.h>
int main()
{
 glutCreateWindow("first
 graphics window");
}
```

**glutCreateWindow**  
creates a top-level  
window.

**String** argument in the  
window is used as  
window name.

**Implicitly,** the *current*  
*window* is set to the newly  
created window.

**Each** created window has a  
unique associated OpenGL  
context.

window's *display state* is not actually acted upon until `glutMainLoop` is entered.

This means until `glutMainLoop` is called, rendering to a created window is ineffective.

The `int` value returned is a unique identifier for the window.

identifiers starts from one, identifier can be used for calling `glutSetWindow`.

**Now we will use more features of glut library in a sample program provided you in the notes; here we will discuss them in details.**

**glutInitDisplayMode**

**glutInitDisplayMode** sets the *initial display mode*.

```
void glutInitDisplayMode (
 unsigned int mode);
```

Display mode, normally the bitwise *OR*-ing of GLUT display mode bit masks.

**Modes have values:**

**GLUT\_RGBA** Bit mask to select an RGBA mode window. This is the default if neither GLUT\_RGBA nor GLUT\_INDEX are specified.

**GLUT\_RGB** An alias for **GLUT\_RGBA**.

**GLUT\_INDEX** Bit mask to select a color index mode window. This overrides **GLUT\_RGBA**.

**GLUT\_SINGLE** Bit mask to select a single buffered window. This is the default.

**GLUT\_DOUBLE** Bit mask to select a double buffered window. This overrides **GLUT\_SINGLE** if it is also specified.

**GLUT\_ACCUM** Bit mask to select a window with an accumulation buffer.

**GLUT\_ALPHA** Bit mask to select a window with an alpha component to the color buffer(s).

**GLUT\_DEPTH** Bit mask to select a window with a depth buffer.

**GLUT\_STENCIL** Bit mask to select a window with a stencil buffer.



**GLUT\_MULTISAMPLE** Bit mask to select a window with multisampling support. If multisampling is not available, a non-multisampling window will automatically be chosen.

**GLUT\_STEREO** Bit mask to select a stereo window.

**GLUT\_LUMINANCE** Bit mask to select a window with a "luminance" color model.

# **glutReshapeWindow**

**glutReshapeWindow** requests a change to the size.

```
void glutReshapeWindow(int
width, int height);
```

**New width and height of  
window in pixels.**

This requests a change in the size of the *current window*. The width and height parameters are size extents in pixels. The width and height must be positive values.

The requests by `glutReshapeWindow` is executed after returning to the main event loop.

This allows multiple `glutReshapeWindow`, `glutPositionWindow`, and `glutFullScreen` requests with same window be coalesced.

# glutKeyboardFunc

**glutKeyboardFunc** sets the keyboard callback for the *current window*.

```
void glutKeyboardFunc(void
 (*func)(unsigned char
 key, int x, int y)); func
```

**The new keyboard callback function.**

**glutKeyboardFunc** sets the keyboard callback for the *current window*. When a user types into the window, each key press generating an ASCII character will generate a keyboard callback.

The **x** and **y** callback parameters indicate the mouse location in window relative coordinates when the key was pressed.

# glutDisplayFunc

**glutDisplayFunc** sets the display callback for the *current window*.

```
void glutDisplayFunc(void
 (*func)(void)); func
```

**The new display callback function.**

**glutDisplayFunc** sets the display callback for the *current window*. When GLUT determines that the normal plane for the window needs to be redisplayed, the display callback for the window is called.

GLUT determines when the display callback should be triggered based on the window's redisplay state. The redisplay state for a window can be either set explicitly by calling **glutPostRedisplay** or implicitly as the result of window damage reported by the window system.

# glutReshapeFunc

**glutReshapeFunc** sets the reshape callback for the *current window*.

```
void glutReshapeFunc(void
 (*func)(int width,
 int height)); func
```

The new reshape callback function.



**glutReshapeFunc** sets the reshape callback for the *current window*; triggered when a window is reshaped.

Also triggered immediately before a window's first display callback after a window is created or whenever an overlay for the window is established.

The width and height parameters of the callback specify the new window size in pixels. Before the callback, the *current window* is set to the window that has been reshaped.

**This default callback will simply call**

**glViewport(0,0,width,height) on the normal plane.**

**If an overlay is established for the window, a single reshape callback is generated. If a top-level window is reshaped, subwindows are not reshaped.**

**glutIdleFunc**

**glutIdleFunc sets the global idle callback.**

```
void glutIdleFunc(void
 (*func)(void));
```

**glutIdleFunc sets the global idle callback to be func so a GLUT program can perform background processing tasks or continuous animation when window system events are not being received.**

If enabled, the idle callback is continuously called when events are not being received.

Programs with multiple windows and/or menus should explicitly set the *current window* and/or *current menu* and not rely on its current setting.

In general, not more than a single frame of rendering should be done in an idle callback.

Passing NULL to glutIdleFunc disables the generation of the idle callback

# **glutMainLoop**

**glutMainLoop** enters the GLUT event processing loop.

```
void glutMainLoop(void);
```

**glutMainLoop** enters the GLUT event processing loop. This routine should be called at most once in a GLUT program.

# glutSwapBuffers

**glutSwapBuffers** swaps the buffers of the *current window* if double buffered.

```
void glutSwapBuffers(void);
```

**Performs a buffer swap on the *layer in use* for the *current window*.**

Promotes the contents of the back buffer of the *layer in use* of the *current window* to become the contents of the front buffer.

The contents of the back buffer then become undefined. The update typically takes place during the vertical retrace of the monitor.

An implicit `glFlush` is done by `glutSwapBuffers` before it returns. Subsequent OpenGL commands are not executed until the buffer exchange.

If the *layer in use* is not double buffered, `glutSwapBuffers` has no effect.

# **Computer Graphics Lecture 34**

## **OpenGL Programming II**

**Taqdees A. Siddiqi  
cs602@vu.edu.pk**



**We will start with a simple program; source code is provided in notes; here we will discuss it to get a better understanding of OpenGL programming.**

**This program first sets perspective projection matrix and then renders a triangle in the “idle” function.**

# glMatrixMode

**glMatrixMode** function specifies which matrix is the current matrix.

```
void glMatrixMode(
GLenum mode);
```

The matrix stack that is the target for resulting matrix operations. The *mode* parameter can:

| Value                | Meaning                                                                     |
|----------------------|-----------------------------------------------------------------------------|
| <b>GL_MODELVIEW</b>  | <b>Applies subsequent matrix</b>                                            |
| <b>GL_PROJECTION</b> | <b>Applies subsequent matrix operations to the projection matrix stack.</b> |
| <b>GL_TEXTURE</b>    | <b>Applies subsequent matrix operations to the texture matrix stack.</b>    |

**glMatrixMode** function sets the current matrix mode.

**The following function retrieves information related to glMatrixMode:**

**Following are the error codes generated and conditions.**

| <b>Error code</b>           | <b>Condition</b>                                                                             |
|-----------------------------|----------------------------------------------------------------------------------------------|
| <b>GL_INVALID_ENUM</b>      | <b>mode was not an accepted value</b>                                                        |
| <b>GL_INVALID_OPERATION</b> | <b>glMatrixMode was called between a call to glBegin and the corresponding call to glEnd</b> |

**glLoadIdentity**

```
void glLoadIdentity(void);
```

**glLoadIdentity** function replaces the current matrix with the identity matrix. It is semantically equivalent to calling **glLoadMatrix** with the identity matrix.

|   |   |   |   |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 |

**The following is the error code and its condition.**

| Error code                  | Condition                                                                                       |
|-----------------------------|-------------------------------------------------------------------------------------------------|
| <b>GL_INVALID_OPERATION</b> | <b>glLoadIdentity was called between a call to glBegin and the corresponding call to glEnd.</b> |

**glTranslated,  
glTranslatef**

**glTranslated** and **glTranslatef** functions multiply the current matrix by a translation matrix.

```
void glTranslated(GLdouble,
 GLdouble y, GLdouble z);
```

```
void glTranslatef(GLfloat x,
 GLfloat y, GLfloat z);
```

**glTranslate** function produces the translation specified by  $(x, y, z)$ . The translation vector is used to compute a 4x4 translation matrix:

$$\begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

**The current matrix is multiplied by translation matrix, with the product replacing the current matrix.**

**M•T**



**If the matrix mode is either `GL_MODELVIEW` or `GL_PROJECTION`, all objects drawn after `glTranslate` is called are translated.**

**Use `glPushMatrix` and `glPopMatrix` to save and restore the untranslated coordinate system.**

**The following is the error code and its condition.**

| Error code                        | Condition                                                                                                                     |
|-----------------------------------|-------------------------------------------------------------------------------------------------------------------------------|
| <code>GL_INVALID_OPERATION</code> | <code>glTranslate</code> was called between a call to <code>glBegin</code> and the corresponding call to <code>glEnd</code> . |

# **gluPerspective**

**The gluPerspective is the function of gl utility library. This function sets up a perspective projection matrix.**

```
void gluPerspective(
 GLdouble fovy,
 GLdouble aspect,
 GLdouble zNear,
 GLdouble zFar);
```

**The field of view angle, in degrees, in the y-direction.**

**The aspect ratio that determines the field of view in the x-direction. The aspect ratio is the ratio of x (width) to y (height).**

**The distance from the viewer to the near clipping plane (always positive).**

**The distance from the viewer to the far clipping plane (always positive).**

**The `gluPerspective` function specifies a viewing frustum into the world coordinate system. In general, the aspect ratio in `gluPerspective` should match the aspect ratio of the associated viewport.**

For example, *aspect* = 2.0 means the viewer's angle of view is twice as wide in *x* as it is in *y*. If the viewport is twice as wide as it is tall, it displays the image without distortion.

The matrix generated by `gluPerspective` is multiplied by the current matrix, just as if `glMultMatrix` were called. To load the perspective matrix onto the current matrix stack instead, precede the call to `gluPerspective` with a call to `glLoadIdentity`.

# **glRotated, glRotatef**

**The glRotated and glRotatef functions multiply the current matrix by a rotation matrix.**

```
void glRotated(
 GLdouble angle,
 GLdouble x, GLdouble y,
 GLdouble z);
```

```
void glRotatef(
 GLfloat angle, GLfloat x,
 GLfloat y, GLfloat z);
```

**The glRotate function computes a matrix that performs a counter-clockwise rotation of *angle* degrees about the vector from the origin through the point.**

**The current matrix is multiplied by rotation matrix, with the product replacing the current matrix.**

**M•R**

**The following is the error code and its condition.**

| <b>Error code</b>           | <b>Condition</b>                                                                          |
|-----------------------------|-------------------------------------------------------------------------------------------|
| <b>GL_INVALID_OPERATION</b> | <b>glRotate was called between a call to glBegin and the corresponding call to glEnd.</b> |



# glClearColor

The `glClearColor` function specifies clear values for the color buffers.

```
void glClearColor(
 GLclampf red,
 GLclampf green,
 GLclampf blue,
 GLclampf alpha);
```

**The `glClearColor` function specifies the red, green, blue, and alpha values used by `glClear` to clear the color buffers. Values specified by `glClearColor` are clamped to the range `[0,1]`.**

**The following is the error code generated and its condition.**

| Error code                               | Condition                                                                                                                            |
|------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------|
| <b><code>GL_INVALID_OPERATION</code></b> | <b><code>glClearColor</code> was called between a call to <code>glBegin</code> and the corresponding call to <code>glEnd</code>.</b> |

# glColor

There are variety of color functions some of them given below, however complete list is provided in the notes.

glColor3b, glColor3d, glColor3f,  
glColor3i, glColor3s, glColor3ub,  
glColor3ui, glColor3us,  
glColor4b, glColor4d.

```
void glColor3b(GLbyte red,
GLbyte green, GLbyte blue);
```

```
void glColor3f(GLfloat red,
GLfloat green, GLfloat blue);
```

**A new alpha value for the current color. Included only in the four-argument glColor4 function.**

**OpenGL stores both a current single-valued color index and a current four-valued RGBA color. The glColor function sets a new four-valued RGBA color.**

**There are two major variants to glColor:**

- **glColor3**
- **glColor4**

**The glColor3b, glColor4b, glColor3s, glColor4s, glColor3i, and glColor4i functions take three or four signed byte, short, or long integers as arguments. When you append v to the name, the color functions can take a pointer to an array of such values.**

**Current color values stored in floating-point format, with unspecified mantissa and exponent sizes.**

**Unsigned integer color components are linearly mapped to floating-point values such that the largest value maps to 1.0 and zero maps to 0.0**

**Signed integer color components are linearly mapped to floating-point values such that the most positive value maps to 1.0 and the most negative value maps to 0.0.**

**Neither floating-point nor signed integer values are clamped to the range  $[0,1]$  before updating the current color.**

**However, color components are clamped to this range before interpolation.**

**You can update the current color at any time. In particular, you can call `glColor` between a call to `glBegin` and the corresponding call to `glEnd`.**

# glClear

The glClear function clears buffers to preset values.

```
void glClear(GLbitfield mask);
```

Bitwise OR operators of masks that indicate the buffers to be cleared. The four masks are as follows.



| Mask                         | Buffer to be Cleared                             |
|------------------------------|--------------------------------------------------|
| <b>GL_COLOR_BUFFER_BIT</b>   | The buffers currently enabled for color writing. |
| <b>GL_DEPTH_BUFFER_BIT</b>   | The depth buffer.                                |
| <b>GL_ACCUM_BUFFER_BIT</b>   | The accumulation buffer.                         |
| <b>GL_STENCIL_BUFFER_BIT</b> | The stencil buffer.                              |

**The glClear function sets the bitplane area of the window to values previously selected by glClearColor, glClearIndex, glClearDepth, glClearStencil, and glClearAccum.**

**You can clear multiple color buffers simultaneously using glDrawBuffer.**

**The glClear function ignores the alpha function, blend function, logical operation, stencil, texture mapping, and z-buffering.**

**The glClear function takes a single argument (mask) that is the bitwise OR of several values indicating which buffer is to be cleared.**

**If a buffer is not present, a glClear call directed at that buffer has no effect.**

**Following are the error codes generated and their conditions.**

| Error code       | Condition                                                         |
|------------------|-------------------------------------------------------------------|
| GL_INVALID_VALUE | Any bit other than the four defined bits was set in <i>mask</i> . |

| Error code                  | Condition                                                                                |
|-----------------------------|------------------------------------------------------------------------------------------|
| <b>GL_INVALID_OPERATION</b> | <b>glClear was called between a call to glBegin and the corresponding call to glEnd.</b> |

**glBegin, glEnd**

**The glBegin and glEnd functions delimit the vertices of a primitive or a group of like primitives.**

```
void glBegin(GLenum mode);
```

```
void glEnd(void);
```

**The following are accepted symbolic constants and their meanings:**

**GL\_POINTS: treats each vertex as a single point. Vertex n: defines point n. N points are drawn.**

**GL\_LINES** treats each pair of vertices as an independent line segment. Vertices  $2n-1$  and  $2n$  define line  $n$ .  $N/2$  lines are drawn.

**GL\_LINE\_STRIP** draws a connected group of line segments from the first vertex to the last. Vertices  $n$  and  $n+1$  define line  $n$ .  $N-1$  lines are drawn.

**GL\_LINE\_LOOP** draws a connected group of line segments from the first vertex to the last, then back to the first. Vertices  $n$  and  $n+1$  define line  $n$ . The last line, however, is defined by vertices  $N$  and  $1$ .  $N$  lines are drawn.

**GL\_TRIANGLES** treats each triplet of vertices as an independent triangle. Vertices  $3n-2$ ,  $3n-1$ , and  $3n$  define triangle  $n$ .  $N/3$  triangles are drawn.

**GL\_TRIANGLE\_STRIP** draws a connected group of triangles. One triangle is defined for each vertex presented after the first two vertices. For odd  $n$ , vertices  $n$ ,  $n + 1$ , and  $n + 2$  define triangle  $n$ . For even  $n$ , vertices  $n + 1$ ,  $n$ , and  $n + 2$  define triangle  $n$ .  $N-2$  triangles are drawn.

## **GL\_TRIANGLE\_FAN**

**draws a connected group of triangles. One triangle is defined for each vertex presented after the first two vertices. Vertices 1,  $n + 1$ , and  $n + 2$  define triangle  $n$ .  $N-2$  triangles are drawn.**

**GL\_QUADS treats each group of four vertices as an independent quadrilateral. Vertices  $4n-3$ ,  $4n-2$ ,  $4n-1$ , and  $4n$  define quadrilateral  $n$ .  $N/4$  quadrilaterals are drawn.**



**GL\_QUAD\_STRIP** draws a connected group of quadrilaterals. One quadrilateral is defined for each pair of vertices presented after the first pair. Vertices  $2n-1$ ,  $2n$ ,  $2n+1$ , and  $2n+2$  define quadrilateral  $n$ .  $N$  quadrilaterals are drawn.

**GL\_POLYGON** draws a single, convex polygon. Vertices  $1$  through  $N$  define this polygon.

**The `glBegin` and `glEnd` functions delimit the vertices that define a primitive or a group of like primitives.**

**Taking  $n$  as an integer count starting at one, and  $N$  as the total number of vertices specified, the interpretations are as follows:**

**You can use only a subset of OpenGL functions between `glBegin` and `glEnd`. The functions you can use are:**

**`glVertex`  
`glColor`  
`glIndex`  
`glNormal`  
`glMaterial`**

**You can also use `glCallList` or `glCallLists` to execute display lists that include only the preceding functions. If any other OpenGL function is called between `glBegin` and `glEnd`, the error flag is set and the function is ignored.**

**Regardless of the value chosen for *mode* in `glBegin`, there is no limit to the number of vertices you can define between `glBegin` and `glEnd`. Lines, triangles, quadrilaterals, and polygons that are incompletely specified are not drawn.**

**The minimum specification of vertices for each primitive is:**

| <b>Minimum number of vertices</b> | <b>Type of primitive</b> |
|-----------------------------------|--------------------------|
| <b>1</b>                          | <b>Point</b>             |
| <b>2</b>                          | <b>Line</b>              |
| <b>3</b>                          | <b>Triangle</b>          |
| <b>4</b>                          | <b>Quadrilateral</b>     |
| <b>3</b>                          | <b>Polygon</b>           |

**Modes that require a certain multiple of vertices are GL\_LINES (2), GL\_TRIANGLES (3), GL\_QUADS (4), and GL\_QUAD\_STRIP (2).**

**The following are the error codes generated and their conditions.**

| <b>Error code</b>           | <b>Condition</b>                                                                                                                                                                                                    |
|-----------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>GL_INVALID_ENUM</b>      | <b>mode was set to an unaccepted value.</b>                                                                                                                                                                         |
| <b>GL_INVALID_OPERATION</b> | <b>A function other than glVertex, glColor, glIndex, glNormal, glTexCoord, glEvalCoord, glEvalPoint, glMaterial, glEdgeFlag, glCallList, or glCallLists was called between glBegin and the corresponding glEnd.</b> |

**The following are the error codes generated and their conditions.**

| <b>Error code</b>           | <b>Condition</b>                                                                                                                         |
|-----------------------------|------------------------------------------------------------------------------------------------------------------------------------------|
| <b>GL_INVALID_OPERATION</b> | <b>The function glEnd was called before the corresponding glBegin was called, or glBegin was called within a glBegin/glEnd sequence.</b> |

# glVertex

There are variety of vertex functions some of them given below, however complete list is provided in the notes.

**glVertex2d, glVertex2f,  
glVertex2i, glVertex2s,  
glVertex3d, glVertex3f,  
glVertex3i, glVertex3s,  
glVertex4d.**

```
void glVertex3f(
 GLfloat x,
 GLfloat y,
 GLfloat z);
```

**Consult notes to see for the other functions of the same nature.**

**The glVertex function commands are used within glBegin/glEnd pairs to specify point, line, and polygon vertices. The current color, normal, and texture coordinates are associated with the vertex when glVertex is called.**

**When only x and y are specified, z defaults to 0.0 and w defaults to 1.0. When x, y, and z are specified, w defaults to 1.0.**

**Invoking glVertex outside of a glBegin/glEnd pair results in undefined behavior.**

# **Computer Graphics Lecture 34**



# **Computer Graphics Lecture 35**

## **CURVES**

**Taqdees A. Siddiqi**  
**cs602@vu.edu.pk**

# Curves

**We all know what a curve is.**

**Today we will explore the  
mathematical definition of a  
curve in a form that is very  
useful to geometric  
modeling and other  
computer graphics  
applications**

**The mathematics of  
parametric equations is  
the basis for Bezier,  
NURBS (Non Uniform  
Rational Beta Splines), and  
Hermite curves**

## **Parametric Equations of a Curve**

**A parametric curve is one whose defining equations are given in terms of a single, common, independent variable called the parametric variable.**

**We have already encountered parametric variables in earlier discussions of vectors, lines, and planes.**

**In three-dimensional space  
each point on curve has  
unique set of coordinates:  
each coordinate is  
controlled by a separate  
parametric equation,  
whose general form looks  
like**

$$x = x(u), \quad y = y(u), \quad z = z(u) \quad \dots\dots\dots(1)$$

where  $x(u)$  stands for some as yet unspecified function in which  $u$  is the independent variable;

for example,  
 $x(u) = au^2 + bu + c$ , and similarly for  $y(u)$  and  $z(u)$ . It is important to understand that each of these is an independent expression

**The dependent variables are the  $x, y$ , and  $z$  coordinates themselves, because their values depend on the value of the parametric variable  $u$**

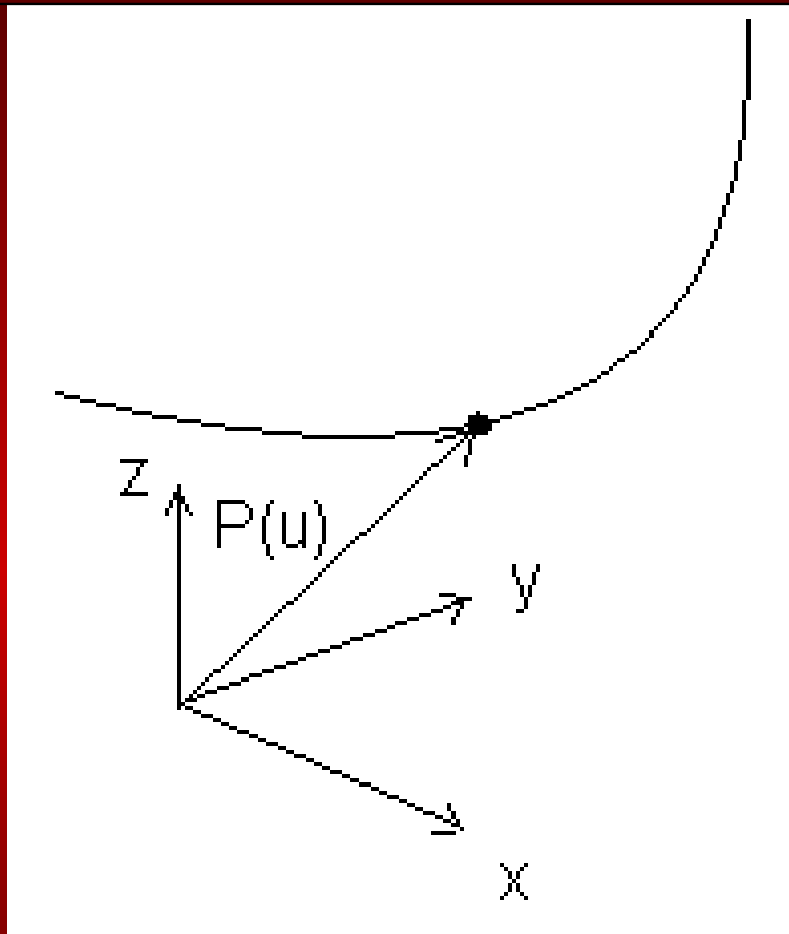


**Figure 1: point of a curve defined by a vector.**

Each point on a curve is defined by a vector  $p$  (figure 1).

The components of this vector are  $x(u)$ ,  $y(u)$ , and  $z(u)$ . we express this as

$$p = p(u) \dots\dots\dots(2)$$

which says that the vector  $p$  is a function of the parametric variable  $u$ .



**There is a lot of information  
in equation 2. when we  
expand it into component  
form, it becomes**

$$p(u) = [x(u) \quad y(u) \quad z(u)] \quad \dots\dots\dots(3)$$

**There are only a few  
simple rules that we  
must follow to design  
these mathematical  
functions**

- 1) define each component by a single, common parametric variable, and**
- 2) make sure that each point on the curve corresponds to a unique value of the parametric variable.**

## **Plane Curves**

**To define plane curves, we use parametric functions that are second-degree polynomials:**

$$x(u) = a_x u^2 + b_x u + c_x$$

$$y(u) = a_y u^2 + b_y u + c_y \quad \dots\dots\dots(4)$$

$$z(u) = a_z u^2 + b_z u + c_z$$

- where the  $a$ ,  $b$ , and  $c$  terms are constant coefficients.
- We can combine  $x(u)$ ,  $y(u)$ ,  $z(u)$ , and their respective coefficients into an equivalent, more concise, vector equation:

$$P(u) = au^2 + bu + c \quad \dots\dots\dots(5)$$

**We allow the parametric variable to take on values only in the interval**

$$0 \leq u \leq 1.$$

**It can generate conic curves but it cannot generate curve with an inflection point, like an S-shaped curve, no matter what values we select for the coefficients  $a$ ,  $b$ ,  $c$ . to do this requires a cubic polynomial.**

To one of the end points we assign  $u = 0$ , and to the other  $u = 1$ . To the intermediate point, we arbitrarily assign  $u = 0.5$ . We can write these points as

- $P_0 = [x_0 \quad y_0 \quad z_0]$
- $P_{0.5} = [x_{0.5} \quad y_{0.5} \quad z_{0.5}] \dots (6)$
- $P_1 = [x_1 \quad y_1 \quad z_1]$

- Where the subscripts indicate the value of the parametric variable at each point.
- Now we solve equations 4 for the  $a_x$ ,  $b_x$ , ...,  $c_z$  coefficients in terms of these points.  
Thus, for  $x$  at  $u = 0$ ,  $u = 0.5$ , and  $u = 1$ , we have

- $x_0 = c_x$
- $x_{0.5} = 0.25a_x + 0.5b_x + c_x \dots(7)$
- $x_1 = a_x + b_x + c_x$
- with similar equations for  $y$ , and  $z$ .

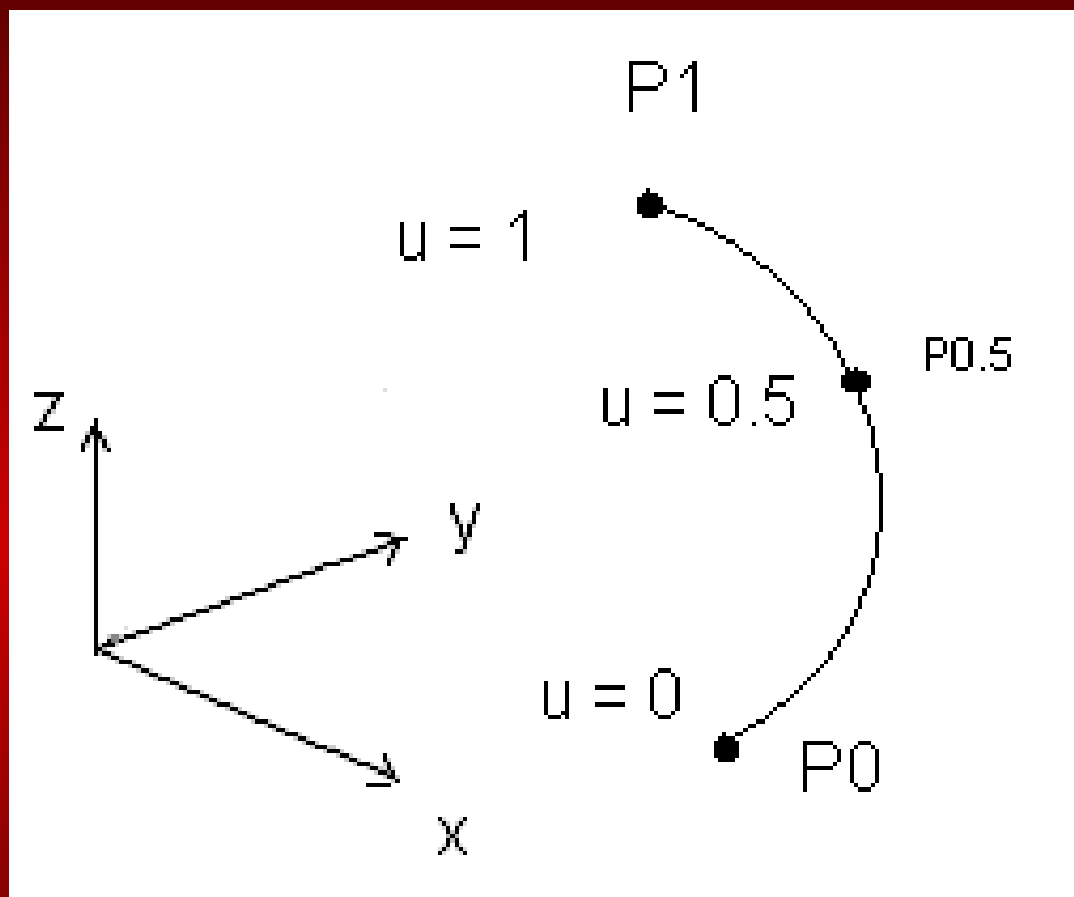


**Figure 2: A plane curve defined by three points**

- Next we solve these three equations in three unknowns for  $a_x$ ,  $b_x$ , and  $c_x$ , finding
- $a_x = 2x_0 - 4x_{0.5} + 2x_1$
- $b_x = -3x_0 + 4x_{0.5} + x_1 \dots (8)$
- $c_x = x_0$



- **Substituting this result in to equation 4 yields**

- $$x(u) = (2x_0 - 4x_{0.5} + 2x_1) u^2 + (-3x_0 + 4x_{0.5} - x_1) u + x_0 \quad \dots(9)$$

- **Again, there are equivalent expressions for  $y(u)$  and  $z(u)$ .**
- **We rewrite equation 9 as follows:**

$$\begin{aligned} \mathbf{x}(u) = & (2u^2 - 3u + 1) \mathbf{x}_0 + \\ & (-4u^2 + 4u) \mathbf{x}_{0.5} + \\ & (2u^2 - u) \mathbf{x}_1 \quad \dots (10) \end{aligned}$$

- Using this result and equivalent expressions for  $y(u)$  and  $z(u)$ , we combine them into a single vector equation:

$$\begin{aligned} P(u) = & (2u^2 - 3u + 1) P_0 + \\ & (-4u^2 + 4u) P_{0.5} + \\ & (2u^2 - u) P_1 \quad \dots(11) \end{aligned}$$

**Equation 11 produces the same curve as Equation 5.**

**The curve will always lie in a plane no matter what three points we choose.**

Furthermore, it is interesting to note that the point  $P_{0.5}$  which is on the curve at  $u = 0.5$ , is not necessarily half way along the length of the curve between  $p_0$  and  $p_1$ .

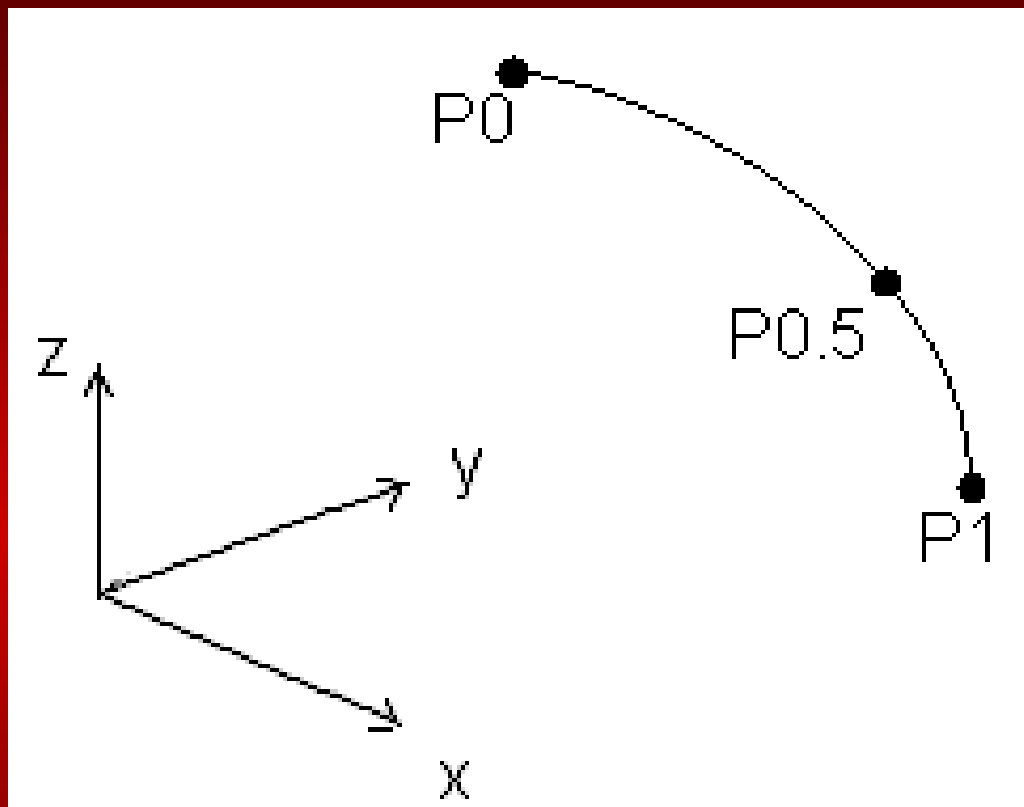


**Figure 3: Curve defined by three non-uniformly spaced points.**

**Equation 5 is the algebraic form and equation 11 is the geometric form. Each of these equations can be written more compactly with matrices**

**So now we rewrite equation 5 using the following substitutions:**

$$\begin{bmatrix} u^2 & u & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = au^2 + bu + c \quad \dots\dots\dots(12)$$

$$U = \begin{bmatrix} u^2 & u & 1 \end{bmatrix} \quad \dots\dots\dots(13)$$

$$A = \begin{bmatrix} a & b & c \end{bmatrix}^T \quad \dots\dots\dots(14)$$

**And finally we obtain**

$$p(u) = UA \quad \dots\dots\dots(15)$$

**Remember that  $A$  is actually a matrix of vectors, so that**

$$A = \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \end{bmatrix} \quad \dots\dots\dots(16)$$

- The nine terms on the right are called the algebraic coefficients.
- Next, we convert equation 11 into matrix form. The right-hand side looks like the product of two matrices

$$[(2u^2 - 3u + 1) \quad (-4u^2 + 4u) \quad (2u^2 - u)]$$

and

$$[p_0 \quad p_{0.5} \quad p_1]$$

This means that



$$p(u) = [(2u^2 - 3u + 1) \quad (-4u^2 + 4u) \quad (2u^2 - u)] \begin{bmatrix} p_0 \\ p_{0.5} \\ p_1 \end{bmatrix} \dots\dots\dots(17)$$

**Using the following substitutions**

$$F = [(2u^2 - 3u + 1) \quad (-4u^2 + 4u) \quad (2u^2 - u)] \dots\dots\dots(18)$$

**and**

$$p = \begin{bmatrix} p_0 \\ p_{0.5} \\ p_1 \end{bmatrix} = \begin{bmatrix} x_0 & y_0 & z_0 \\ x_{0.5} & y_{0.5} & z_{0.5} \\ x_1 & y_1 & z_1 \end{bmatrix} \dots\dots\dots(19)$$

where  $P$  is the control point matrix and the nine terms on the right are its elements or the geometric coefficients, we can now write

$$p(u) = FP \dots\dots\dots(20)$$

- This is the matrix version of the geometric form.
- Because it is the same curve in algebraic form,  $p(u)=UA$ , or geometric form,  $p(u)=FP$ , we can write

$$FP = UA \quad \dots\dots\dots(21)$$

The F matrix is itself the product of two other matrices

$$F = \begin{bmatrix} u^2 & u & 1 \end{bmatrix} \begin{bmatrix} 2 & -4 & 2 \\ -3 & 4 & -1 \\ 1 & 0 & 0 \end{bmatrix} \dots\dots\dots(22)$$

**The matrix on the left we recognize as U, and we can denote the other matrix as**

$$M = \begin{bmatrix} 2 & -4 & 2 \\ -3 & 4 & -1 \\ 1 & 0 & 0 \end{bmatrix} \dots\dots\dots(23)$$

**This means that**

$$F = UM \dots\dots\dots(24)$$

Using this we substitute appropriately to find

$$UMP = UA \quad \dots\dots\dots(25)$$

Pre-multiplying each side of this equation by  $1/U$  yields

$$MP = A \quad \dots\dots\dots(26)$$

**This expresses a simple relationship  
between the algebraic and geometric  
coefficients**

$$A = MP \quad \dots\dots\dots(27)$$

**Or**

$$P = M^{-1} A \quad \dots\dots\dots(28)$$

The matrix  $M$  is called a basis transformation matrix, and  $F$  is called a blending function matrix.

# Computer Graphics Lecture 35

# Computer Graphics

## Lecture 36

### CURVES II

Taqdees A. Siddiqi  
cs602@vu.edu.pk

### CURVES II

Taqdees A. Siddiqi  
cs602@vu.edu.pk



# **RECAP of last lecture: Parametric Equations of a Curve**

A parametric curve is one whose defining equations are given in terms of a single, common, independent variable called the parametric variable.

$$x = x(u), \quad y = y(u), \quad z = z(u) \quad \dots\dots\dots(1)$$

for example,

$x(u) = au^2 + bu + c$ , and similarly for  $y(u)$  and  $z(u)$ . It is important to understand that each of these is an independent expression

Each point on a curve is defined by a vector  $p$  (figure 1).  
The components of this vector are  $x(u)$ ,  $y(u)$ , and  $z(u)$ .  
we express this as

$$\mathbf{p} = \mathbf{p}(u) \quad \dots\dots\dots (2)$$

which says that the vector  $p$  is a function of the  
parametric variable  $u$ .

There is a lot of information in equation 2. when we expand it into component form, it becomes

$$p(u) = [x(u) \quad y(u) \quad z(u)] \quad \dots\dots\dots(3)$$

define each component by a single, common parametric variable, and  
make sure that each point on the curve corresponds to a unique value of the parametric variable.

# Plane Curves

To define plane curves, we use parametric functions that are second-degree polynomials:

$$\begin{aligned}x(u) &= a_x u^2 + b_x u + c_x \\y(u) &= a_y u^2 + b_y u + c_y \quad \dots\dots\dots(4) \\z(u) &= a_z u^2 + b_z u + c_z\end{aligned}$$

where the a, b, and c terms are constant coefficients.

We can combine x(u), y(u), z(u), and their respective coefficients into an equivalent, more concise, vector equation:

$$P(u) = au^2 + bu + c \quad \dots\dots\dots(5)$$

We allow the parametric variable to take on values only in the interval

$$0 \leq u \leq 1.$$

To one of the end points we assign  $u = 0$ , and to the other  $u = 1$ . To the intermediate point, we arbitrarily assign  $u = 0.5$ . We can write these points as

$$P_0 = [x_0 \quad y_0 \quad z_0]$$

$$P_{0.5} = [x_{0.5} \quad y_{0.5} \quad z_{0.5}] \dots (6)$$

$$P_1 = [x_1 \quad y_1 \quad z_1]$$



$$x_0 = cx$$

$$x_{0.5} = 0.25ax + 0.5bx + cx \dots(7)$$

$$x_1 = ax + bx + cx$$

with similar equations for y, and z.

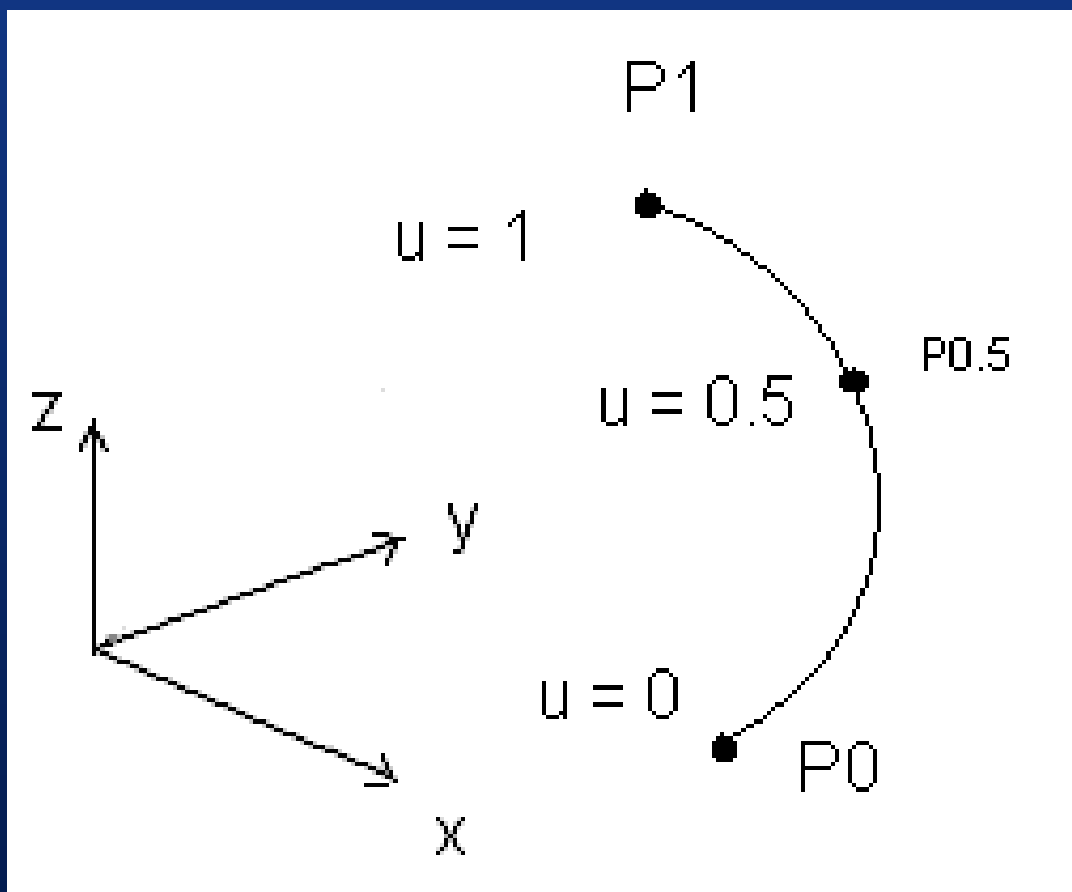


Figure 2: A plane curve defined by three points

Next we solve these three equations in three unknowns for  $a_x$ ,  $b_x$ , and  $c_x$ , finding

$$a_x = 2x_0 - 4x_{0.5} + 2x_1$$

$$b_x = -3x_0 + 4x_{0.5} + x_1 \dots(8)$$

$$c_x = x_0$$

Substituting this result in to equation 4 yields

$$x(u) = (2x_0 - 4x_{0.5} + 2x_1) u^2 + (-3x_0 + 4x_{0.5} - x_1) u + x_0 \dots(9)$$

Again, there are equivalent expressions for  $y(u)$  and  $z(u)$ .

We rewrite equation 9 as follows:

$$\begin{aligned} x(u) = & (2u^2 - 3u + 1) x_0 + \\ & (-4u^2 + 4u) x_{0.5} + \\ & (2u^2 - u) x_1 \quad \dots (10) \end{aligned}$$

$$\begin{aligned}
 P(u) = & (2u^2 - 3u + 1) P_0 + \\
 & (-4u^2 + 4u) P_{0.5} + \\
 & (2u^2 - u) P_1 \dots (11)
 \end{aligned}$$

Equation 5 is the algebraic form and equation 11 is the geometric form. Each of these equations can be written more compactly with matrices

So now we rewrite equation 5 using the following substitutions:

$$\begin{bmatrix} u^2 & u & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = au^2 + bu + c \quad \dots\dots\dots(12)$$

$$U = [u^2 \quad u \quad 1] \quad \dots\dots\dots(13)$$

$$A = [a \quad b \quad c]^T \quad \dots\dots\dots(14)$$

And finally we obtain

$$p(u) = UA \quad \dots\dots\dots(15)$$

Remember that A is actually a matrix of vectors, so that

$$A = \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \end{bmatrix} \dots\dots\dots(16)$$

The nine terms on the right are called the algebraic coefficients.

Next, we convert equation 11 into matrix form. The right-hand side looks like the product of two matrices

$$[(2u^2 - 3u + 1) \quad (-4u^2 + 4u) \quad (2u^2 - u)]$$

and

$$[p_0 \quad p_{0.5} \quad p_1]$$

This means that

$$p(u) = [(2u^2 - 3u + 1) \quad (-4u^2 + 4u) \quad (2u^2 - u)] \begin{bmatrix} p_0 \\ p_{0.5} \\ p_1 \end{bmatrix} \dots\dots\dots(17)$$

Using the following substitutions

$$F = [(2u^2 - 3u + 1) \quad (-4u^2 + 4u) \quad (2u^2 - u)] \dots\dots\dots(18)$$



and

$$p = \begin{bmatrix} p_0 \\ p_{0.5} \\ p_1 \end{bmatrix} = \begin{bmatrix} x_0 & y_0 & z_0 \\ x_{0.5} & y_{0.5} & z_{0.5} \\ x_1 & y_1 & z_1 \end{bmatrix} \dots\dots\dots(19)$$

$$p(u) = FP \dots\dots\dots(20)$$

This is the matrix version of the geometric form.

Because it is the same curve in algebraic form,  
 $p(u)=UA$ , or geometric form,  $p(u)=FP$ , we can write

$$FP = UA \quad \dots\dots\dots(21)$$

The F matrix is itself the product of two other matrices

$$F = \begin{bmatrix} u^2 & u & 1 \end{bmatrix} \begin{bmatrix} 2 & -4 & 2 \\ -3 & 4 & -1 \\ 1 & 0 & 0 \end{bmatrix} \dots\dots\dots(22)$$

The matrix on the left we recognize as U, and we can denote the other matrix as

$$M = \begin{bmatrix} 2 & -4 & 2 \\ -3 & 4 & -1 \\ 1 & 0 & 0 \end{bmatrix} \dots\dots\dots(23)$$

This means that

$$F = UM \dots\dots\dots(24)$$

Using this we substitute appropriately to find

$$UMP = UA \quad \dots\dots\dots(25)$$

Pre-multiplying each side of this equation by 1/U yields

$$MP = A \quad \dots\dots\dots(26)$$

Or

$$P = M^{-1} A \quad \dots\dots\dots (28)$$

**Space Curves**

A space curve is not confined to a plane. It is free to twist through space.

To define a space curve we must use parametric functions that are cubic polynomials. For  $x(u)$  we write:

$$x(u) = a_x u^3 + b_x u^2 + c_x u + d_x$$

**Equation (1)**

we combine the  $x(u)$ ,  $y(u)$  and  $z(u)$  expressions into a single vector equation :

$$p(u) = au^3 + bu^2 + cu + d$$

**Equation (2)**

If  $a = 0$ , then this equation is identical to Equation discussed in plane curves

$$p(u) = bu^2 + cu + d$$

$$p(u) = au^2 + bu + c$$

To define a specific curve in space, we use the same approach as we did for a plane curve. This time, though, there are 12 coefficients to be determined

We specify four points through which we want the curve to pass, which provides all the information we need to determine  $a$ ,  $b$ ,  $c$ , and  $d$ .

Which four points?



we can have the four points we need: Points at

$$u = 0$$

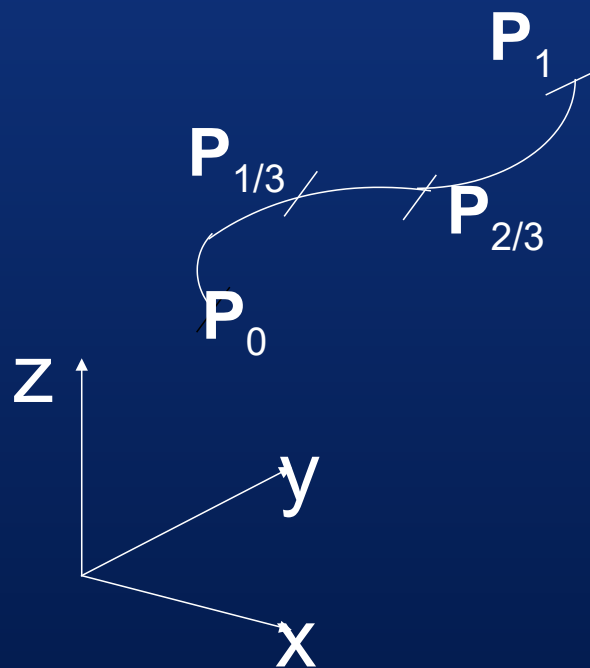
$$u = 1$$

it turns out to be advantageous to use two intermediate points that we assign parametric values of

$$u = \frac{1}{3}$$

$$u = \frac{2}{3}$$

So we now have the four points we need.



$$p(0)$$

$$p\left(\frac{1}{3}\right)$$

$$p\left(\frac{2}{3}\right)$$

$$p(1)$$

which we can rewrite as the more convenient  $p_1$ ,  $p_2$ ,  $p_3$ , and  $p_4$  (Figure 1).

Substituting each of the values of the parametric variable into Equation 1, we obtain the following four equations in four unknowns:

$$x_1 = d_x$$

$$x_2 = \frac{1}{27}a_x + \frac{1}{9}b_x + \frac{1}{3}c_x + d_x$$

$$x_3 = \frac{8}{27}a_x + \frac{4}{9}b_x + \frac{2}{3}c_x + d_x$$

$$x_4 = a_x + b_x + c_x + d_x$$

**Equation (3)**

Now we can express  $a_x$ ,  $b_x$ ,  $c_x$  and  $d_x$  in terms of  $x_1$ ,  $x_2$ ,  $x_3$  and  $x_4$ . After doing the necessary algebra, we obtain

$$a_x = -\frac{9}{2}x_1 + \frac{27}{2}x_2 - \frac{27}{2}x_3 + \frac{9}{2}x_4$$

$$b_x = 9x_1 - \frac{45}{2}x_2 + 18x_3 - \frac{9}{2}x_4$$

$$c_x = -\frac{11}{2}x_1 + 9x_2 - \frac{9}{2}x_3 + x_4$$

$$d_x = x_1$$

**Equation (4)**

We substitute these results into Equation 1, producing

$$x(u) = \left( -\frac{9}{2}x_1 + \frac{27}{2}x_2 - \frac{27}{2}x_3 + \frac{9}{2}x_4 \right)u^3 + \left( 9x_1 - \frac{45}{2}x_2 + 18x_3 - \frac{9}{2}x_4 \right)u^2 + \left( -\frac{11}{2}x_1 + 9x_2 - \frac{9}{2}x_3 + x_4 \right)u + x_1$$

**Equation (5)**

Rewriting Equation 5 as follows:

$$\begin{aligned} x(u) = & \left( -\frac{9}{2}u^3 + 9u^2 - \frac{11}{2}u + 1 \right) x_1 + \left( \frac{27}{2}u^3 - \frac{45}{2}u^2 + 9u \right) x_2 \\ & + \left( -\frac{27}{2}u^3 + 18u^2 - 9u \right) x_3 + \left( \frac{9}{2}u^3 - \frac{9}{2}u^2 + u \right) x_4 \end{aligned}$$

## Equation (6)

Using equivalent expressions for  $y(u)$  and  $z(u)$ , we can summarize them with a single vector equation:

$$p(u) = \left(-\frac{9}{2}u^3 + 9u^2 - \frac{11}{2}u + 1\right)p_1 + \left(\frac{27}{2}u^3 - \frac{45}{2}u^2 + 9u\right)p_2 \\ + \left(-\frac{27}{2}u^3 + 18u^2 - 9u\right)p_3 + \left(\frac{9}{2}u^3 - \frac{9}{2}u^2 + u\right)p_4$$

## Equation (7)

This means that, given four point assigned successive values of  $u$  (in this case at  $u=0$ ,  $1/3$ ,  $2/3$  &  $1$ ), equation 7 produces a curve that starts at  $p_1$ , passes through  $p_2$  and  $p_3$ , and ends at  $p_4$ .

Now let's take one more step towards a more compact notation. Using the four parametric functions appearing in Equation 7, we define a new matrix, where

$$G_1 = \left( -\frac{9}{2}u^3 + 9u^2 - \frac{11}{2}u + 1 \right)$$

$$G_2 = \left( \frac{27}{2}u^3 - \frac{45}{2}u^2 + 9u \right)$$

$$G_3 = \left( -\frac{27}{2}u^3 + 18u^2 - 9u \right)$$

$$G_4 = \left( \frac{9}{2}u^3 - \frac{9}{2}u^2 + u \right)$$

**Equation (8)**



And then define a matrix  $P$  containing the control points,

$$P = [P_1 \quad P_2 \quad P_3 \quad P_4]^T$$

- so that

$$P(u) = GP$$

**Equation (9)**

The matrix  $G$  is the product of two other matrices,  $U$  and  $N$ :

$$G = UN$$

**Equation (10)**

Where

$$U = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix}^T$$

And

$$N = \begin{bmatrix} -\frac{9}{2} & -\frac{27}{2} & -\frac{27}{2} & \frac{9}{2} \\ 9 & -\frac{45}{2} & -\frac{9}{2} & 1 \\ -\frac{11}{2} & 9 & -\frac{9}{2} & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \quad \text{Equation (11)}$$

(Note that N is another example of a basis transformation matrix.)

Now we let

$$A = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \\ d_x & d_y & d_z \end{bmatrix}$$

**Equation (12)**

Using matrices, Equation 2 becomes

$$P(u) = UA$$

**Equation (13)**

To convert the information in the  $A$  matrix into that required for the  $P$  matrix, we do some simple matrix algebra, using Equations 9, 10 and 13. First we have

And then

$$GP = UNP$$

**Equation (14)**

And then

$$UA = UNP$$

**Equation (15)**

Or more simply

$$A = NP$$

Equation (16)

Computer Graphics

Lecture 36

# **Computer Graphics Lecture 37**

## **CURVES III**

**Taqdees A. Siddiqi**  
**cs602@vu.edu.pk**

# The Tangent Vector

- Another way to define a space curve does not use intermediate points. It uses the tangents at each end of a curve, instead

- Every point on a curve has a straight line associated with it called the tangent line, which is related to the first derivation of the Parametric functions  $x(u)$ ,  $y(u)$ , and  $z(u)$

$$\frac{d}{du}x(u), \quad \frac{d}{du}y(u), \quad \text{and} \quad \frac{d}{du}z(u)$$

**Equation (1)**



- From elementary calculus, we can compute, for example,

$$\frac{dy}{du} = \frac{dy(u) / du}{dx(u) / du}$$

**Equation (2)**

- We can treat as components of a vector along the tangent line to the curve. We call this the *tangent vector*, and define it as

$$P^u(u) = \left[ \frac{d}{du} x(u)i \quad \frac{d}{du} y(u)j \quad \frac{d}{du} z(u)k \right]$$

Equation (3)

- Or more simply as

$$P^u = \begin{bmatrix} x^u & y^u & z^u \end{bmatrix}$$

Equation (4)

- (Here the superscript  $u$  indicates the first derivative operation with respect to the independent variable  $u$ ). This is a very powerful idea, and we will now see how to use it to define a curve

- We will still use the two end points, but instead of two intermediate points, we will use the tangent vectors at each end to supply the information we need to define a curve

- By manipulating these tangent vectors, we can control the slope at each end. The set of vectors  $\mathbf{p}_0$ ,  $\mathbf{p}_1$ , and  $\mathbf{p}_2$  are called the *boundary conditions*

- This method itself is called the *cubic Hermite interpolation*, after C. Hermite (1822-1901) the French mathematician who made significant contributions to our understanding of cubic and quadratic polynomials.

- We differentiate to obtain the  $x$  component of the tangent vector:

$$\frac{d}{du} dx(u) = x^u = 3a_x u^2 + 2b_x u + c_x$$

Equation (5)

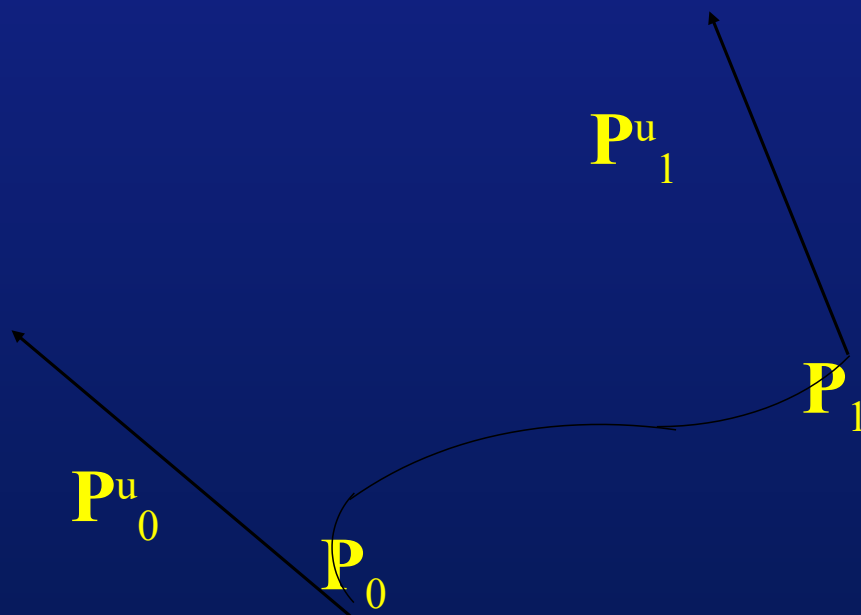


Figure (1)

$$x(u) = a_x u^3 + b_x u^2 + c_x u + d_x$$

**Equation (1A)**

- **Evaluating (1A) and Equation 5 at  $u = 0$ ,  $u = 1$ , yields**

$$x_0 = d_x$$

$$x_1 = a_x + b_x + c_x + d_x$$

$$x_0^u = c_x$$

$$x_1^u = 3a_x + 2b_x + c_x$$

### Equation (6)

- Using these four equations in four unknowns, we solve for  $a_x$ ,  $b_x$ ,  $c_x$  and  $d_x$  in terms of the boundary conditions



$$\begin{aligned}
 a_x &= 2(x_0 - x_1) + x_0^u + x_1^u \\
 b_x &= 3(-x_0 + x_1) - 2x_0^u - x_1^u \\
 c_x &= x_0^u \\
 d_x &= x_0
 \end{aligned}$$

**Equation (7)**

- **Substituting the result into Equation (1A), yields**

$$x(u) = (2x_0 - 2x_1 + x_0^u + x_1^u)u^3 + (-3x_0 + 3x_1 - 2x_0^u - x_1^u)x_0^u u^2 + x_0^u + x_0$$

**Equation (8)**

- **Rearranging terms we can rewrite this as**

$$x(u) = (2u^3 - 3u^2 + 1)x_0 + (-2u^3 + 3u^2)x_1 + (u^3 - 2u^2 + u)x_0^u + (u^3 - u^2)x_1^u$$

## **Equation (9)**

- **Because  $y(u)$  and  $z(u)$  have equivalent forms, we can include them by rewriting Equation 9 in vector form:**

$$p(u) = (2u^3 - 3u^2 + 1)p_0 + (-2u^3 + 3u^2)p_1 + (u^3 - 2u^2 + u)p_0^u + (u^3 - u^2)p_1^u$$

## Equation (10)

- To express Equation 14.54 in matrix notation, we first define a blending function matrix  $F = \begin{bmatrix} F_1 & F_2 & F_3 & F_4 \end{bmatrix}$

Where

$$F_1 = 2u^3 - 3u^2 + 1$$

$$F_2 = -2u^3 + 3u^2$$

$$F_3 = u^3 - 2u^2 + u$$

$$F_4 = u^3 - u^2$$

## Equation (11)

- These matrix elements are the polynomial coefficients of the vectors which we rewrite as:

$$p(u) = F_1 p_0 + F_2 p_1 + F_3 p_0^u + F_4 p_1^u$$

**Equation (12)**

- **If we assemble the vectors representing the boundary conditions into a matrix B,**

$$B = \begin{bmatrix} p_0 & p_1 & p_0^u & p_1^u \end{bmatrix}^T$$

**Equation (13)**

**Then**

$$p(u) = FB$$

**Equation (14)**

- Here again we write the matrix  $F$  as the product of two matrices,  $U$  and  $M$ , so that

$$F = UM$$

Equation (15)

- where

$$U = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix}$$

**Equation (16)**

- And

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

**Equation (17)**



- Rewriting Equation 14 using these substitutions, we obtain

$$P(u) = UMB$$

Equation (18)

- It is easy to show the relationship between the algebraic and geometric coefficients for a space curve. Since

$$P(u) = UA$$

**Equation (19)**

- the relationship between A and B is, again,

$$A = MB$$

**Equation (20)**

- The magnitude of the tangent vector is also necessary and contributes to the shape of the curve. In fact, we can write  $p_0^u$  and  $p_1^u$  as

$$p_0^u = m_0 t_0$$

Equation (21)

And

$$p_1^u = m_1 t_1$$

Equation (22)

- Clearly,  $m_0$ , and  $m_1$  are the magnitudes of and .
- Using these relationships, we modify Equation 10 as follows:

$$p(u) = (2u^3 - 3u^2 + 1)p_0 + (-2u^3 + 3u^2)p_1 + (u^3 - 2u^2 + u)m_0t_0 + (u^3 - u^2)m_1t_1$$

Equation (23)

# Computer Graphics Lecture 37

- To define a space curve we must use parametric functions that are cubic polynomials. For  $x(u)$  we write:

$$x(u) = a_x u^3 + b_x u^2 + c_x u + d_x$$

Equation (1)

- A space curve is not confined to a plane. It is free to twist through space.

# Space Curves

- we combine the  $x(u)$ ,  $y(u)$  and  $z(u)$  expressions into a single vector equation :

$$p(u) = au^3 + bu^2 + cu + d$$

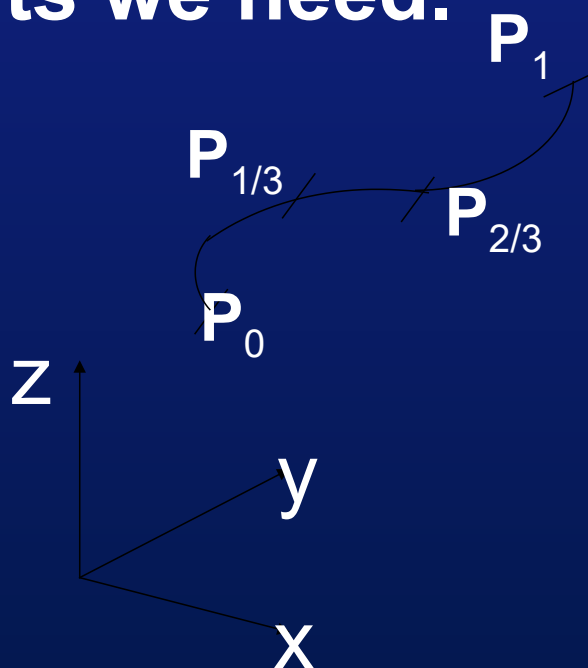
Equation (2)

- If  $a = 0$ , then this equation is identical to Equation discussed in plane curves

$$p(u) = bu^2 + cu + d$$

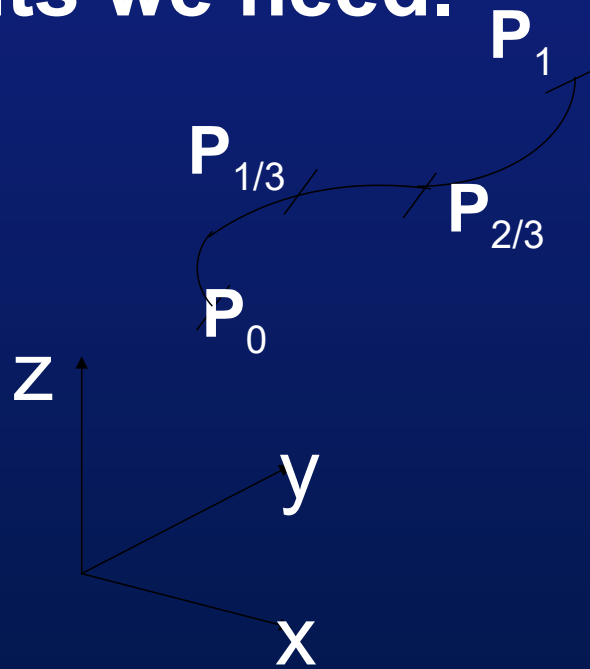
$$p(u) = au^2 + bu + c$$

- So we now have the four points we need.





- So we now have the four points we need.



$$x_1 = d_x$$

$$x_2 = \frac{1}{27}a_x + \frac{1}{9}b_x + \frac{1}{3}c_x + d_x$$

$$x_3 = \frac{8}{27}a_x + \frac{4}{9}b_x + \frac{2}{3}c_x + d_x$$

$$x_4 = a_x + b_x + c_x + d_x$$

**Equation (3)**

$$a_x = -\frac{9}{2}x_1 + \frac{27}{2}x_2 - \frac{27}{2}x_3 + \frac{9}{2}x_4$$

$$b_x = 9x_1 - \frac{45}{2}x_2 + 18x_3 - \frac{9}{2}x_4$$

$$c_x = -\frac{11}{2}x_1 + 9x_2 - \frac{9}{2}x_3 + x_4$$

$$d_x = x_1$$

**Equation (4)**

$$x(u) = \left(-\frac{9}{2}x_1 + \frac{27}{2}x_2 - \frac{27}{2}x_3 + \frac{9}{2}x_4\right)u^3 + \left(9x_1 - \frac{45}{2}x_2 + 18x_3 - \frac{9}{2}x_4\right)u^2 + \left(-\frac{11}{2}x_1 + 9x_2 - \frac{9}{2}x_3 + x_4\right)u + x_1$$

**Equation (5)**

- **Rewriting Equation 5 as follows:**

$$x(u) = \left(-\frac{9}{2}u^3 + 9u^2 - \frac{11}{2}u + 1\right)x_1 + \left(\frac{27}{2}u^3 - \frac{45}{2}u^2 + 9u\right)x_2 \\ + \left(-\frac{27}{2}u^3 + 18u^2 - 9u\right)x_3 + \left(\frac{9}{2}u^3 - \frac{9}{2}u^2 + u\right)x_4$$

**Equation (6)**

$$p(u) = \left(-\frac{9}{2}u^3 + 9u^2 - \frac{11}{2}u + 1\right)p_1 + \left(\frac{27}{2}u^3 - \frac{45}{2}u^2 + 9u\right)p_2 \\ + \left(-\frac{27}{2}u^3 + 18u^2 - 9u\right)p_3 + \left(\frac{9}{2}u^3 - \frac{9}{2}u^2 + u\right)p_4$$

**Equation (7)**

- This means that, given four point assigned successive values of  $u$  (in this case at  $u=0, 1/3, 2/3$  &  $1$ ), equation 7 produces a curve that starts at  $p_1$ , passes through  $p_2$  and  $p_3$ , and ends at  $p_4$ .

- Now let's take one more step towards a more compact notation. Using the four parametric functions appearing in Equation 7, we define a new matrix, where

$$G_1 = \left( -\frac{9}{2}u^3 + 9u^2 - \frac{11}{2}u + 1 \right)$$

$$G_2 = \left( \frac{27}{2}u^3 - \frac{45}{2}u^2 + 9u \right)$$

$$G_3 = \left( -\frac{27}{2}u^3 + 18u^2 - 9u \right)$$

$$G_4 = \left( \frac{9}{2}u^3 - \frac{9}{2}u^2 + u \right)$$

**Equation (8)**

- **And then define a matrix P containing the control points,**

$$P = [P_1 \quad P_2 \quad P_3 \quad P_4]^T$$

- **so that**

$$P(u) = GP$$

**Equation (9)**

- The matrix  $G$  is the product of two other matrices,  $U$  and  $N$ :

$$G = UN$$

Equation (10)

- Where

$$U = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix}^T$$

- And

$$N = \begin{bmatrix} -\frac{9}{2} & -\frac{27}{2} & -\frac{27}{2} & \frac{9}{2} \\ 9 & -\frac{45}{2} & -\frac{9}{2} & 1 \\ -\frac{11}{2} & 9 & -\frac{9}{2} & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Equation (11)

$$A = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \\ d_x & d_y & d_z \end{bmatrix}$$

**Equation (12)**

- **Using matrices, Equation 2 becomes**

$$P(u) = UA$$

**Equation (13)**

- And then

$$GP = UNP$$

Equation (14)

- And then

$$UA = UNP$$

Equation (15)

- Or more simply

$$A = NP$$

Equation (16)



$$p(0) \quad p\left(\frac{1}{3}\right) \quad p\left(\frac{2}{3}\right) \quad p(1)$$

- which we can rewrite as the more convenient  $p_1$ ,  $p_2$ ,  $p_3$ , and  $p_4$  (Figure 1).

# Computer Graphics Lecture 38

# **CURVES IV**

**Taqdees A. Siddiqi**  
**cs602@vu.edu.pk**

## **Bezier Curves**

**The Bezier curve is an important part of almost every computer-graphics illustration program and computer-aided design system in use today**

**It is used in many ways, from designing the curves and surfaces of automobiles to defining the shape of letters in type fonts.**

**Because it is numerically the most stable of all the polynomial-based curves used in these applications, the Bezier curve is the ideal standard for representing the more complex piecewise polynomial curves.**

**In the early 1960s, Peter Bezier began looking for a better way to define curves and surfaces one that would be useful to a design engineer**

# A Geometric Construction

We can draw a Bezier curve using a simple recursive geometric construction. Let's begin by constructing a second-degree curve.

We select three points A, B, C so that line AB is a tangent to the curve at A, and BC is tangent at C. The curve begins at A and ends at C.

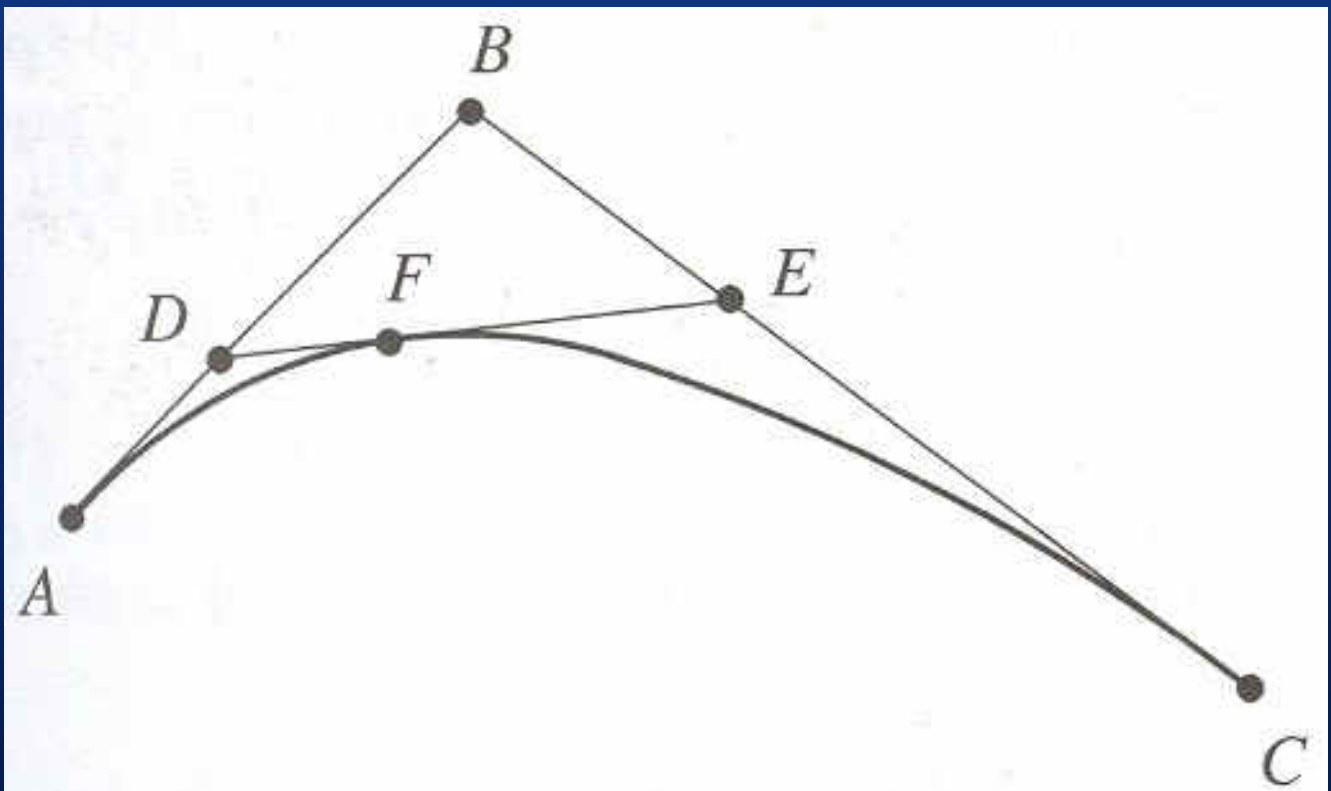


Figure 1: *Geometric Construction of a second-degree Bezier curve.*

For any ratio,  $u_i$ , we construct points D and E such that:

$$\frac{AD}{AB} = \frac{BE}{BC} = u_i$$

Equation 1

On DE we construct F so that  $DF/DE = u_j$ . Point F is on the curve.

Repeating this process for other values of  $u_i$ , we produce a series of points on a Bezier curve.

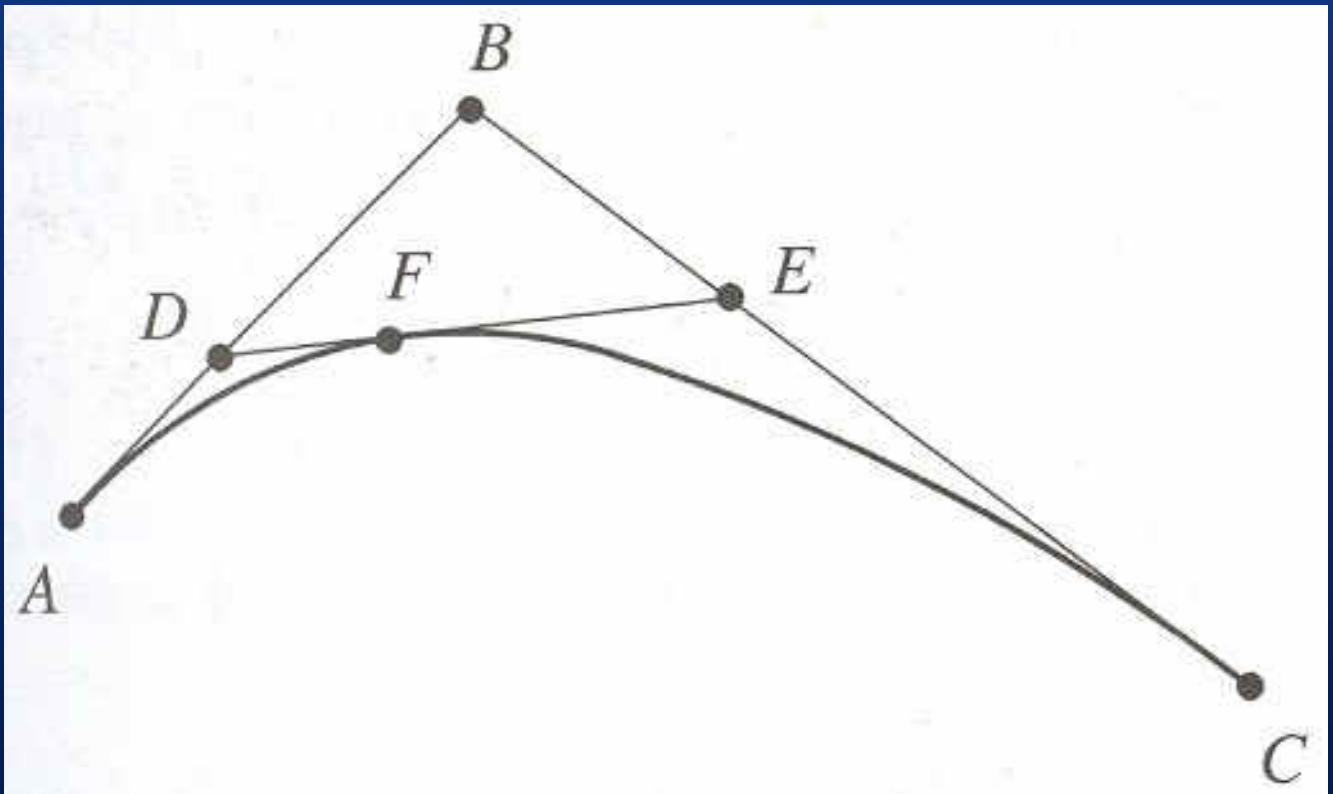


Figure 1: *Geometric Construction of a second-degree Bezier curve.*

**Note that we must be consistent in the order in which we sub-divide AB and BC.**

**For example here first,  
AB is divided into AD & DB  
And then BC into BE & EC  
And finally DE into DF & FE**



To define this curve in a coordinate system, let point

$$A = (x_A, y_A),$$

$$B = (x_B, y_B)$$

Then coordinates of points D and E for some value of  $u_i$  are

$$\begin{aligned} x_D &= x_A + u_i (x_B - x_A) \\ y_D &= y_A + u_i (y_B - y_A) \end{aligned}$$

**Equation 2**

**And**

$$x_E = x_B + u_i(x_C - x_B)$$

$$y_E = y_B + u_i(y_C - y_B)$$

**Equation 3**

**The coordinates of point F  
for some value of  $u_i$  are**

$$x_F = x_D + u_i(x_E - x_D)$$

$$y_F = y_D + u_i(y_E - y_D)$$

**Equation 4**

To obtain  $x_F$  and  $y_F$  in terms of the coordinates of points A, B and C, for any value of  $u_i$  in the unit interval, we substitute appropriately from Equation 2 and 3 into equation 4 of plane curve. After rearranging terms to simplify, we find

$$\begin{aligned}x_F &= (1 - u_i)^2 x_A + 2u_i(1 - u_i)x_B + u_i^2 x_C \\y_F &= (1 - u_i)^2 y_A + 2u_i(1 - u_i)y_B + u_i^2 y_C\end{aligned}$$

**Equation 5**

**We generalize this set of equations for any point on the curve using the following substitutions:**

$$x(u) = x_F$$

$$y(u) = y_F$$

**Equation 6**

**And we let**

$$\begin{aligned}x_0 &= x_A & x_1 &= x_B & x_2 &= x_C \\y_0 &= y_A & y_1 &= y_B & y_2 &= y_C\end{aligned}$$

**Equation 7**

**Now we can rewrite  
Equation 5 as**

$$\begin{aligned}x(u) &= (1-u)^2 x_0 + 2u(1-u)x_1 + u^2 x_2 \\y(u) &= (1-u)^2 y_0 + 2u(1-u)y_1 + u^2 y_2\end{aligned}$$

**Equation 8**

**This is the set of second-degree equations for the coordinates of points on Bezier curve based on our construction**

**We express this construction process and Equations 8 in terms of vectors with the following substitutions.**

Let the vector  $p_0$  represent point A,  $p_1$  point B, and  $p_2$  point C.

From vector geometry we have  $D \equiv p_0 + u(p_1 - p_0)$ , and

$E \equiv p_1 + u(p_2 - p_1)$ , if we let  $F \equiv p(u)$  we see that

$$p(u) = p_0 + u(p_1 - p_0) + u[p_1 + u(p_2 - p_1) - p_0 + u(p_1 - p_0)]$$

## Equation 9

**We rearrange terms to obtain a more compact vector equation of a second degree Bezier curve:**



$$p(u) = (1-u)^2 p_0 + 2u(1-u) p_1 + u^2 p_2$$

## Equation 10

Where the points  $p_0$ ,  $p_1$  and  $p_2$  are called control points, and  $p$  is the point on the curve. The ratio  $u$  is the parametric variable that takes its value in the range  $[0,1]$

Later we will see that this equation is an example of a *Bernstein polynomial*.

Similar constructions apply to Bezier curves of any degree. In fact the degree of a Bezier curve is equal to  $n-1$ , where  $n$  is the number of control points.

Figure 2 shows the construction of point on a cubic Bezier curve, which requires four control points A, B, C, and D to define it. The curve begins at point A tangent to the, AB, and ends at D and tangent to CD. We construct points E, F, and G such that

$$\frac{AE}{AB} = \frac{BF}{BC} = \frac{CG}{CD} = u_i$$

Equation 11

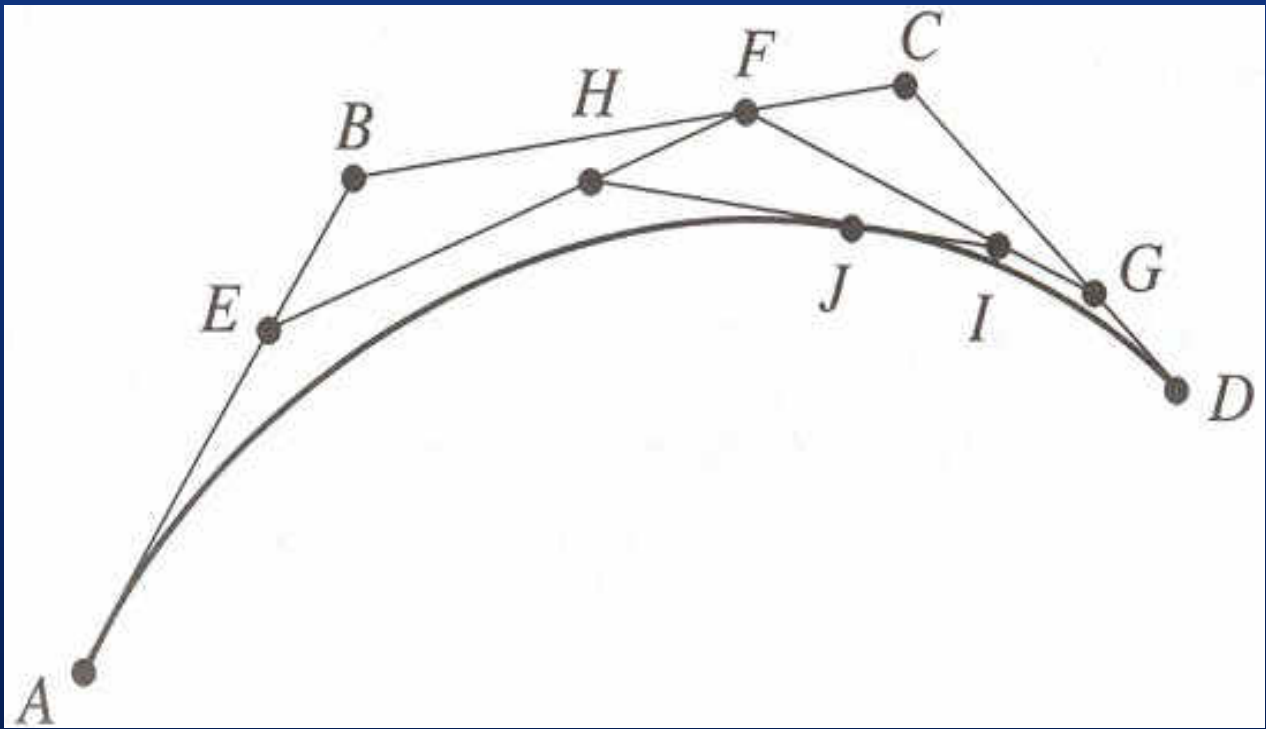


Figure 2: Geometric Construction of a cubic Bezier curve.

On EF and FG we locate H and I, respectively, such that

$$\frac{EH}{EF} = \frac{FI}{FG} = u_i$$

Equation 12

Finally on HI we locate J  
such that

$$\frac{HJ}{HI} = u_i$$

Equation 13

We can make no more subdivisions, which means that point J is on the curve. If we continue this process for a sequence of points, then their locus defines the curve.

**If points A, B, C, and D are represented by the vectors  $p_0$ ,  $p_1$ ,  $p_2$  and  $p_3$  respectively, then expressing the construction of the intermediate points E, F, G, H, and I in terms of these vectors to produce point J, or  $P(u)$ , yields**

$$\begin{aligned}
 p(u) = & p_0 + u(p_1 - p_0) + u[p_1 + u(p_2 - p_1) - p_0 + u(p_1 - p_0)] \\
 & + u\{p_1 + u(p_2 - p_1) + u[p_2 + u(p_3 - p_2) - p_1 + u(p_2 - p_0)] - p_0 \\
 & - u(p_1 - p_0) + u[p_1 + u(p_2 - p_1) - p_0 - u(p_1 - p_0)]\}
 \end{aligned}$$

**Equation 14**

**This awkward expression  
simplifies nicely to**

$$p(u) = (1-u)^3 p_0 + 3u(1-u)^2 p_1 + 3u^2(1-u) p_2 + u^3 p_3$$

## **Equation 15**

**The geometric construction  
of a Bezier curve shows  
how the control points  
influence its shape. The  
curve begins at the first  
point and ends at the last  
point.**

It is tangent to the lines connecting the first two points and the last two points. The curve is always contained within the *convex hull* of the control points.

## An Algebraic Definition



**Bezier began with the idea that any point  $p(u)$  on a curve segment should be given by an equation such as the following:**

$$p(u) = \sum_{i=0}^n p_i f_i(u)$$

**Equation 16**

where  $0 \leq u \leq 1$  and vectors  $p_i$  are the control points (Figure 3).

Equation 16 is a compact way to express the sum of several similar terms because what it says is this:

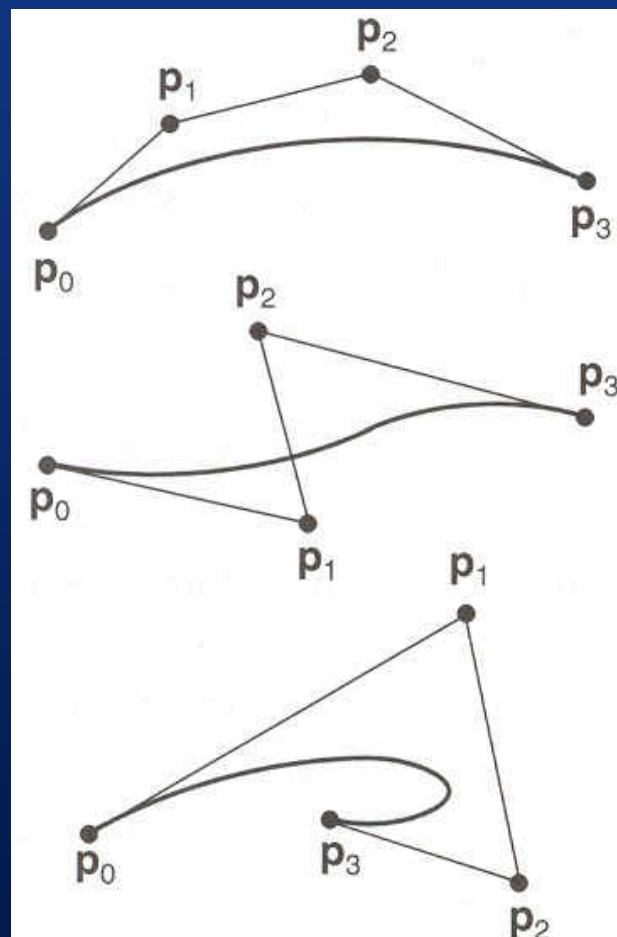


Figure 3: *Bezier curves and their control points.*

$$p(u) = p_0 f_0(u) + p_1 f_1(u) + \dots + p_n f_n(u)$$

## Equation 17

Of which Equation 10 and 15 are specific examples, for  $n=2$  and  $n=3$ , respectively.

The  $n+1$  functions, that is the  $f_i(u)$  must produce a cure that has certain well-defined characteristics. Here are some of the most important ones:

1. The curve must start on the first control point,  $p_0$ , and end on the last,  $p_n$ . Mathematically, we say that the functions must interpolate these two points.
2. The curve must be tangent to the line given by  $p_1 - p_0$  at  $p_0$  and to  $p_n - p_{n-1}$  at  $p_n$ .

3. The functions  $f_i(u)$  must be symmetric with respect to  $u$  and  $(1-u)$ . This lets us reverse the sequence of control points without changing the shape of the curve.

# Some Characteristics of Bezier Curves:

- End Point Interpolation
- Affine Invariance (transform ctrl points and redraw – curve keeps shape)
- Convex Hull Property
- Linear Precision (ctrl points in st. line)
- Variation Diminishing

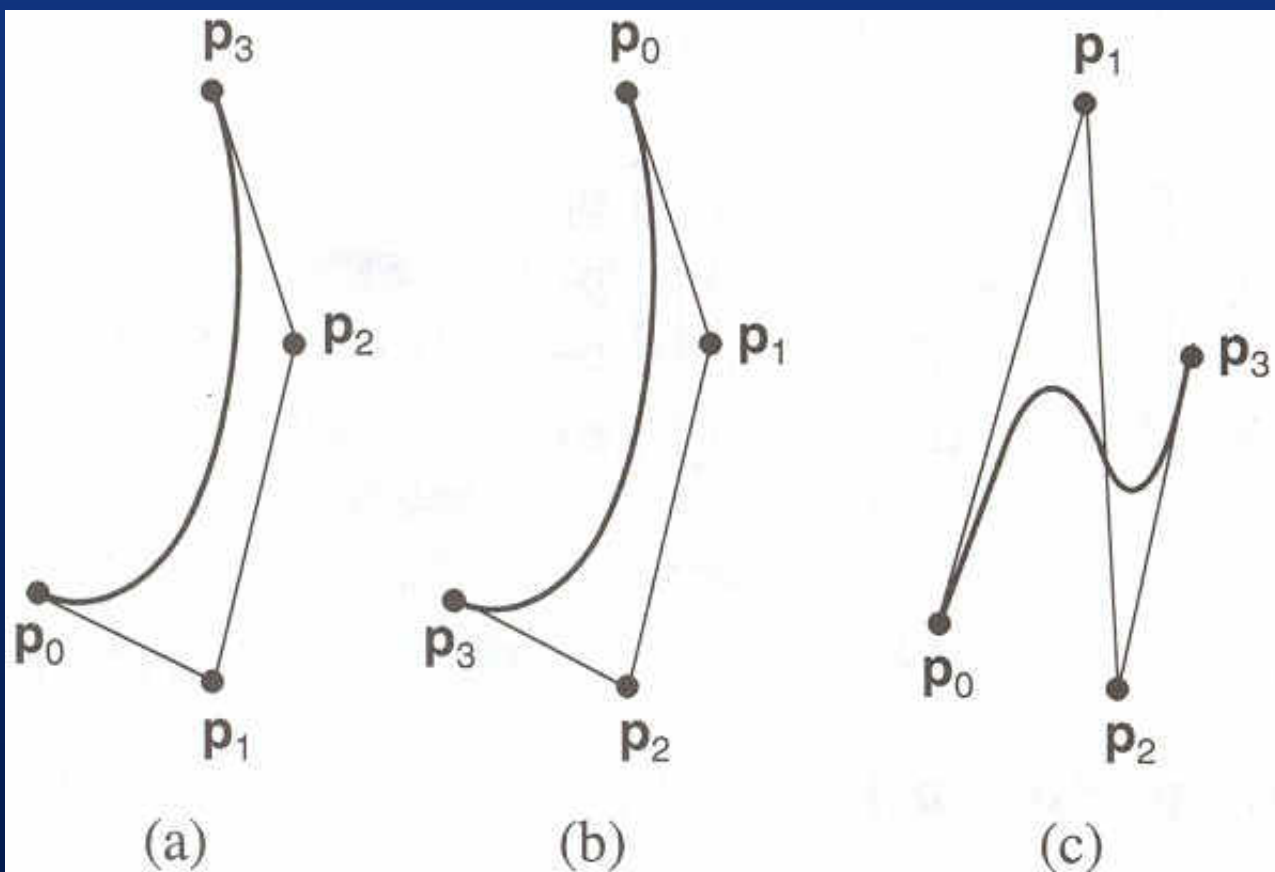


Figure 4: Three different sequences of four control points.

- Problem of Local Control

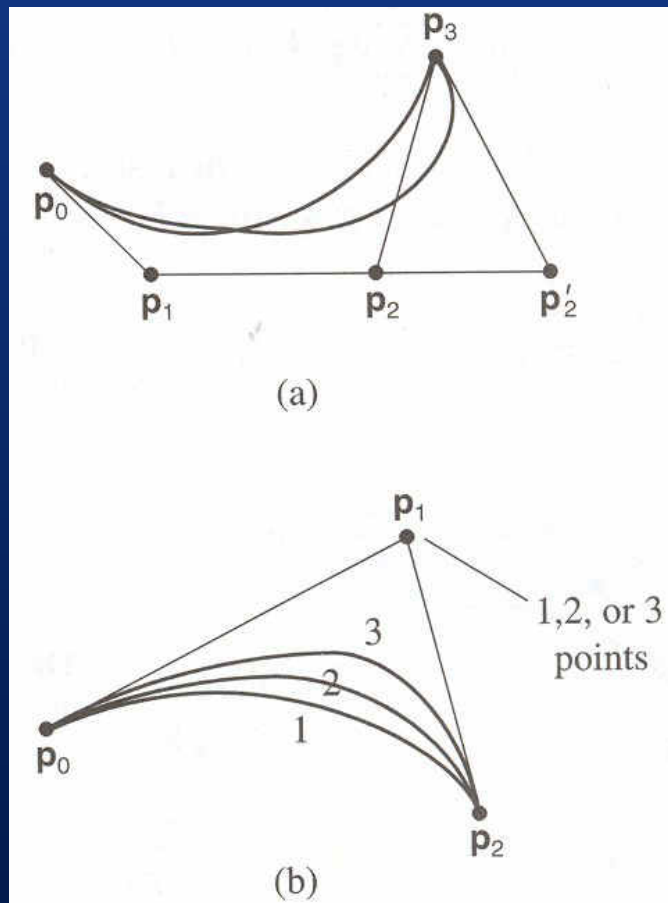


Figure 5: *Modifying the shape of a Bezier curve*

A family of functions called *Bernstein polynomials* satisfies these requirements. They are the *basis functions* of the Bezier curve (Other curves, such as the NURBS curves, use different, but related, basis functions).

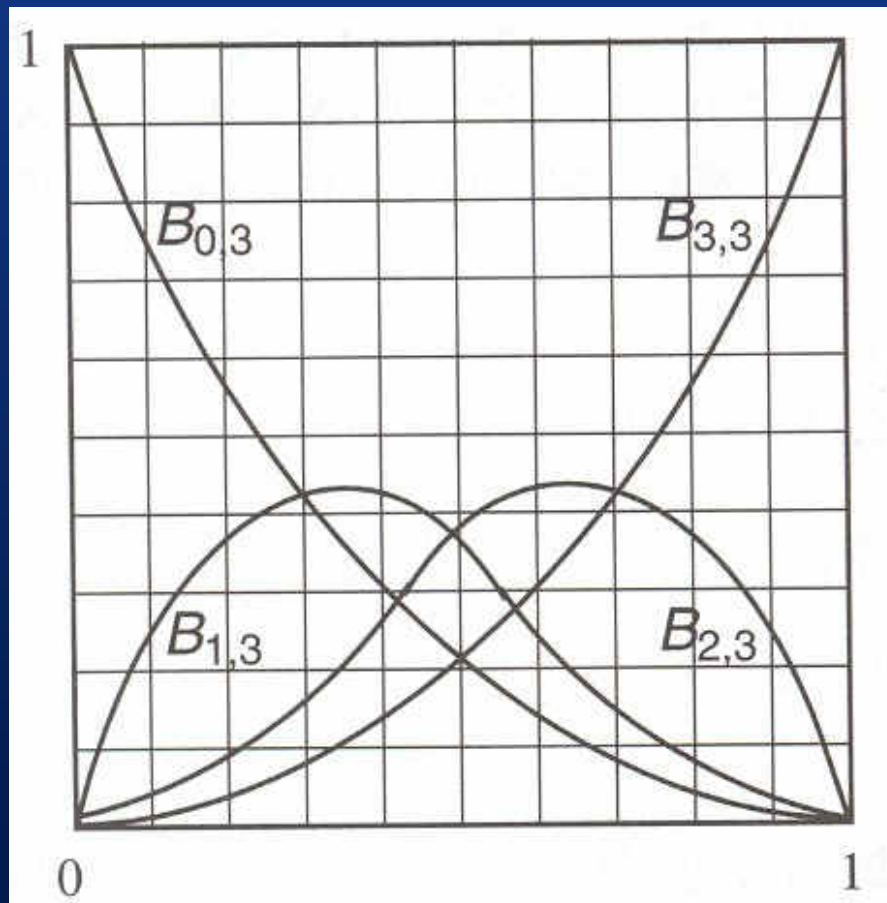


Figure 6: *Bezier curve basis functions*

**We rewrite Equation 16  
using these basis  
functions as:**

$$p(u) = \sum_{i=0}^n p_i B_{i,n}(u)$$

**Equation 18**

**Were the basis functions  
are given as:**

$$B_{i,n}(u) = \binom{n}{i} u^i (1-u)^{n-i}$$

**Equation 19**

**The term  $\binom{n}{i}$  is the binomial  
coefficient function from  
probability theory and  
statistics, defined as**



$$\binom{n}{i} = \frac{n!}{i!(n-i)!}$$

## Equation 20

The symbol ! is the factorial operator. For example,  $3! = 3 \times 2 \times 1$ ,  $5! = 5 \times 4 \times 3 \times 2 \times 1$ , so forth. We use the following conventions when evaluating Equation 20; If  $i$  and  $u$  equal zero, then  $u_i = 1$  and  $0! = 1$ .

**We see that for  $n+1$  control points, the basis functions produce an  $n$ th-degree polynomial.**

**Expanding Equation 18 for a second degree Bezier curve (When  $n=2$  and there are three control points) produces**

$$p(u) = p_0 B_{0,2}(u) + p_1 B_{1,2}(u) + p_2 B_{2,2}(u)$$

**Equation 21**

**From Equation 20, we find**

$$B_{0,2}(u) = (1 - u)^2$$

**Equation 22**

$$B_{1,2}(u) = 2u(1-u)$$

**Equation 23**

**and**

$$B_{2,2}(u) = u^2$$

**Equation 24**

**These are the basis functions for a second-degree Bezier curve. Substituting them into Equation 21 and rearranging terms, we find**

$$p(u) = (1-u)^2 p_0 + 2u(1-u) p_1 + u^2 p_2$$

## Equation 25

**This is the same expression we found from the geometric construction, Equation 10. The variable  $u$  is now called the parametric variable.**

**Now, let's expand Equation 18 for a cubic Bezier curve, where  $n=3$ :**

$$p(u) = p_0 B_{0,3}(u) + p_1 B_{1,3}(u) + p_2 B_{2,3}(u) + p_3 B_{3,3}(u)$$

**Equation 26**

**And from Equation 20 we find**

$$B_{0,3}(u) = (1 - u)^3$$

**Equation 27**

$$B_{1,3}(u) = 3u(1-u)^2$$

**Equation 28**

$$B_{2,3}(u) = 3u^2(1-u)$$

**Equation 29**

$$B_{3,3}(u) = u^3$$

**Equation 30**

**Substituting  
Equation  
rearranging  
produces**

**these  
26**

**into  
and  
terms**



$$p(u) = (1-u)^3 p_0 + 3u(1-u)^2 p_1 + 3u^2(1-u) p_2 + u^3 p_3$$

## Equation 31

**Bezier curve equations are well suited for expression in matrix form. We can expand the cubic parametric functions and rewrite Equation 31 as**

$$p(u) = \begin{bmatrix} (1 - 3u + 3u^2 - u^3) \\ (3u - 6u^2 + 3u^3) \\ (3u^2 - 3u^3) \\ u^3 \end{bmatrix}^T \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{bmatrix}$$

**Equation 32**

$$p(u) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{bmatrix}$$

**Equation 33**

**If we let**

$$U = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix}$$

**Equation 34**

$$P = \begin{bmatrix} p_0 & p_1 & p_2 & p_3 \end{bmatrix}^T$$

**Equation 35**

$$M = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

**Equation 36**

**Then we can write Equation 33 even more compactly as**

$$p(u) = UMP$$

**Equation 37**

**Note that the composition of the matrices  $U$ ,  $M$ , and  $P$  varies according to the number of control points (that is, the degree of the Bernstein polynomial basis functions).**

# **Computer Graphics Lecture 38**

# Computer Graphics

## Lecture 39

### Building Polygonal Models of Surfaces

Taqdees A. Siddiqi  
cs602@vu.edu.pk

Constructing polygonal approximations to surfaces is an art, and there is no substitute for experience.

A few points would be of help:

Keep polygon orientations consistent.

Make sure that when viewed from the outside, all the polygons on the surface are oriented in the same direction.

Try to get this right the first time, since it's awfully painful to fix the problem later.

(If you use `glScale*()` to reflect geometry around some axis of symmetry, you might change the orientation with `glFrontFace()` to keep the orientations consistent.)

When you subdivide a surface, watch out for any non-triangular polygons.

The three vertices of a triangle are guaranteed to lie on a plane; any polygon with four or more vertices might not.



Non-planar polygons can be viewed from some orientation such that the edges cross each other, and OpenGL might not render such polygons correctly.

There's always a trade-off between the display speed and the quality of the image.

Ideally, you can provide a parameter to the subdivision routines that indicates how fine a subdivision you want, and if the object is farther from the eye, you can use a coarser subdivision.

Also, when you sub-divide, use large polygons where the surface is relatively flat, and small polygons in regions of high curvature

For high-quality images, it's a good idea to subdivide more on the silhouette edges than in the interior.

Silhouette edges occur where the normal vectors are perpendicular to the vector from the surface to the viewpoint - that is, when their vector dot product is zero.

Try to avoid T-intersections in your models shown in next figure.

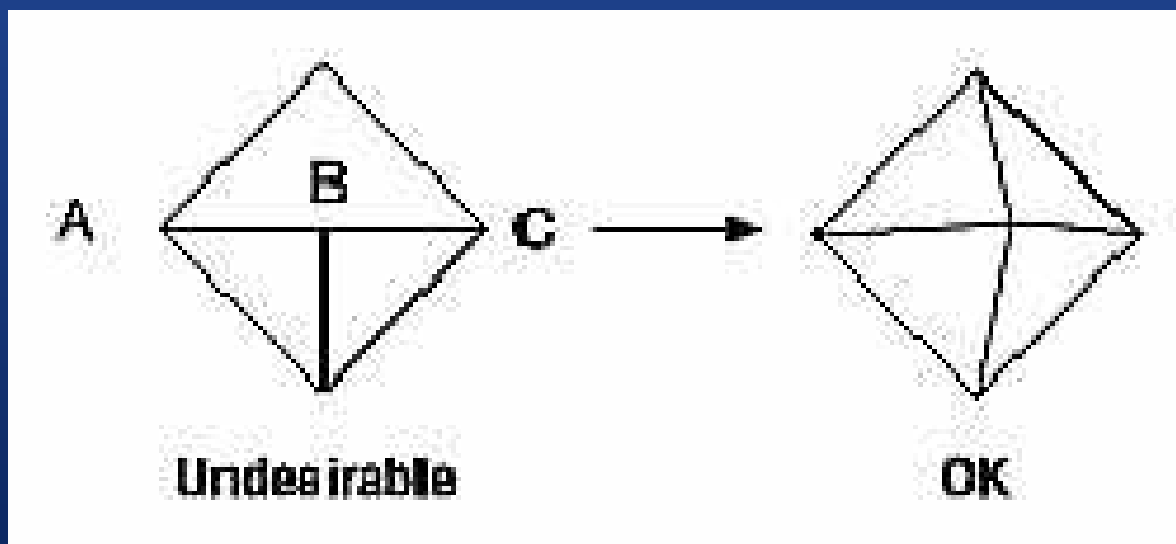


Figure 1: Modifying an undesirable T-intersection

As shown, there's no guarantee that the line segments AB and BC lie on exactly the same pixels as the segment AC.

Sometimes they do, and sometimes they don't, depending on the transformations and orientation. This can cause cracks to appear intermittently in the surface.

If you're constructing a closed surface, make sure to use exactly the same number for coordinates at the beginning and end of a closed loop, or you can get gaps and cracks due to numerical round-off.

Here's a two-dimensional example of bad code:

```
#define PI 3.14159265
#define EDGES 30
```

```
/* draw a circle */
glBegin(GL_LINE_STRIP);
for (i = 0; i <= EDGES; i++)
 glVertex2f (
 cos((2*PI*i) / EDGES),
 sin((2*PI*i) / EDGES));
glEnd();
```

The edges meet exactly only if your machine manages to calculate the sine and cosine of 0 and of  $(2*PI*EDGES/EDGES)$  and gets exactly the same values.

To correct the code, make sure that when  $i == \text{EDGES}$ , you use 0 for the sine and cosine instead of  $(2 * \text{PI} * \text{EDGES} / \text{EDGES})$ .

Or use `GL_LINE_LOOP` instead of `GL_LINE_STRIP`, and change the loop termination condition to  $i < \text{EDGES}$ .

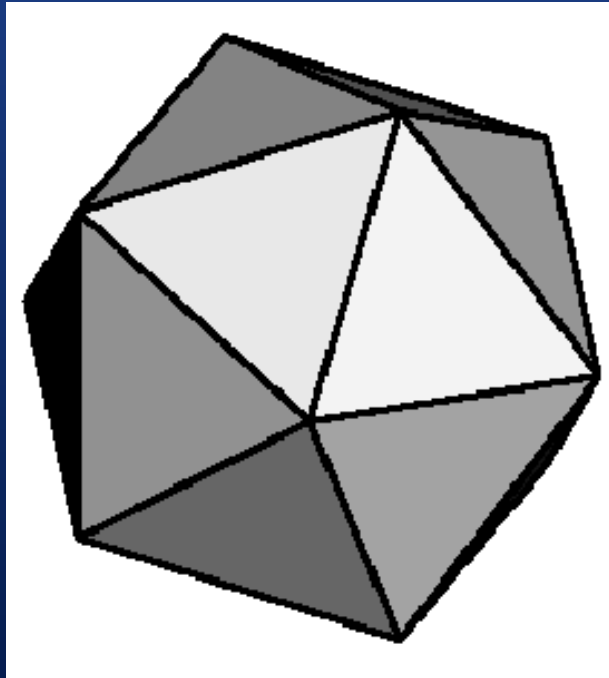


## Example: Icosahedron

This code concerns the vertices of a regular icosahedron (which is a Platonic solid composed of twenty faces that span twelve vertices, each face of which is an equilateral triangle).

An icosahedron can be considered a rough approximation for a sphere. Example 1 defines the vertices and triangles making up an icosahedron and then draws the icosahedron.

## An icosahedron



```
#define X 0.525731112119133606
#define Z 0.850650808352039932
static GLfloat vdata[12][3] = { {-X, 0.0, Z}, {X, 0.0, Z}, {-
 X, 0.0, -Z}, {X, 0.0, -Z}, {0.0, Z, X}, {0.0, Z, -X}, {0.0, -
 Z, X}, {0.0, -Z, -X}, {Z, X, 0.0}, {-Z, X, 0.0}, {Z, -X,
 0.0}, {-Z, -X, 0.0} };
```

```
static GLuint tindices[20][3] = { {0,4,1}, {0,9,4}, {9,5,4},
 {4,5,8}, {4,8,1}, {8,10,1}, {8,3,10}, {5,3,8}, {5,2,3},
 {2,7,3}, {7,10,3}, {7,6,10}, {7,11,6}, {11,0,6}, {0,1,6},
 {6,1,10}, {9,0,11}, {9,11,2}, {9,2,5}, {7,2,11} };
```

```
int i;
glBegin(GL_TRIANGLES);
for (i = 0; i < 20; i++) {
 /* color information here */
 glVertex3fv(&vdata[tindices[i][0]][0]);
 glVertex3fv(&vdata[tindices[i][1]][0]);
 glVertex3fv(&vdata[tindices[i][2]][0]);
}
glEnd();
```

The strange numbers  $X$  and  $Z$  are chosen so that the distance from the origin to any of the vertices of the icosahedron is 1.0.

`vdata[ ][ ]` contains 12 vertices where zeroth vertex is  $\{-X, 0.0, Z\}$ , the first is  $\{X, 0.0, Z\}$ , and so on  
The array `tindices[ ][ ]` tells how to link the vertices to make triangles

For example, the first triangle is made from the zeroth, fourth, and first vertex. If you take the vertices for triangles in the order given, all the triangles have the same orientation.

The line that mentions color information should be replaced by a command that sets the color of the  $i$ th face.

Note: In all the examples, you should probably save the calculated vertex and normal coordinates to avoid repetition. This can be done using your own data structures or by constructing display lists.

### Calculating Normal Vectors for a Surface:

If a surface is to be lit, you need to supply the vector normal to the surface. Calculating the normalized cross product of two vectors on that surface provides normal vector.

With the flat surfaces of an icosahedron, all three vertices defining a surface have the same normal vector. In this case, the normal needs to be specified only once.

## Example 2: Generating Normal Vectors for a Surface

```
GLfloat d1[3], d2[3], norm[3];
for (j = 0; j < 3; j++) {
 d1[j] = vdata[tindices[i][0]][j] - vdata[tindices[i]
[1]][j];
 d2[j] = vdata[tindices[i][1]][j] -
vdata[tindices[i][2]][j];
}
```

```
normcrossprod(d1, d2,
 norm);
glNormal3fv(norm);
```

The function `normcrossprod()` produces the normalized cross product of two vectors, as shown in Example 3.

Example 3 : Calculating the Normalized Cross Product of Two Vectors



```
void normalize(float v[3]) {
 GLfloat d = sqrt(v[0]*v[0]
 +v[1]*v[1]+v[2]*v[2]);
 if (d == 0.0) {
 error("zero length vector");
 return;}
 v[0]/=d; v[1]/=d; v[2]/=d;
}
```

```
void normcrossprod(float v1[3], float v2[3], float out[3]){
 GLint i, j;
 GLfloat length;
 out[0] = v1[1]*v2[2]-v1[2]*v2[1];
 out[1] = v1[2]*v2[0]-v1[0]*v2[2];
 out[2] = v1[0]*v2[1]-v1[1]*v2[0];
 normalize(out);
}
```

If you're using an icosahedron as an approximation for a shaded sphere, you'll want to use normal vectors that are perpendicular to the true surface of the sphere, rather than being perpendicular to the faces.

Since the icosahedron vertex data is for an icosahedron of radius 1, the normal and vertex data is identical. Here is the code that would draw an icosahedral approximation of a smoothly shaded sphere.

```
glBegin(GL_TRIANGLES);
 for (i = 0; i < 20; i++) {
 glNormal3fv(&vdata
 [tindices[i][0]][0]);
 glVertex3fv(&vdata
 [tindices[i][0]][0]);
 glNormal3fv(&vdata
 [tindices[i][1]][0]);
```

```
 glVertex3fv(&vdata
 [tindices[i][1]][0]);
 glNormal3fv(&vdata
 [tindices[i][2]][0]);
 glVertex3fv(&vdata
 [tindices[i][2]][0]);
 }
glEnd();
```

## Improving the Model

A twenty-sided approximation to a sphere doesn't look good unless the image of the sphere on the screen is quite small, but there's an easy way to increase the accuracy of the approximation.

Imagine the icosahedron inscribed in a sphere, and subdivide the triangles as shown in next Figure.

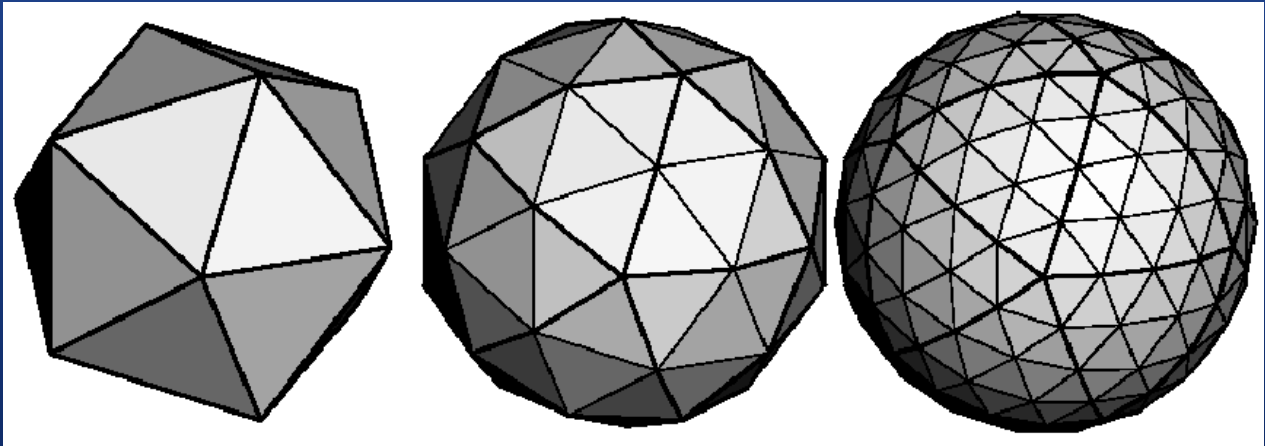


Figure 2: Subdividing to Improve a Polygonal Approximation to a Surface

New vertices lie slightly inside the sphere, so push them to the surface by normalizing them. The three objects shown in Figure 2 use 20, 80, and 320 approximating triangles, respectively.

Example 4 performs a single subdivision, creating an 80-sided spherical approximation.

```
void drawtriangle (float *v1, float *v2, float
 *v3){
 glBegin(GL_TRIANGLES);
 glNormal3fv(v1); glVertex3fv(v1);
 glNormal3fv(v2); glVertex3fv(v2);
 glNormal3fv(v3); glVertex3fv(v3);
 glEnd();}
```

```
void subdivide(float *v1,
 float *v2, float *v3) {
 GLfloat v12[3], v23[3], v31[3];
 GLint i;
 for (i = 0; i < 3; i++) {
 v12[i] = v1[i]+v2[i];
 v23[i] = v2[i]+v3[i];
 v31[i] = v3[i]+v1[i];}
```

```
 normalize(v12);
 normalize(v23);
 normalize(v31);
 drawtriangle(v1, v12, v31);
 drawtriangle(v2, v23, v12);
 drawtriangle(v3, v31, v23);
 drawtriangle(v12, v23, v31);
}
```

```
for (i = 0; i < 20; i++) {
 subdivide(
 &vdata[tindices[i][0]][0],
 &vdata[tindices[i][1]][0],
 &vdata[tindices[i][2]][0]
);
}
```

Example 5 is a slight modification of Example 4 which recursively subdivides the triangles to the proper depth.



If the depth value is 0, no subdivisions are performed, and the triangle is drawn as is. If the depth is 1, a single subdivision is performed, and so on.

```
void subdivide(float *v1, float *v2, float *v3, long depth)
{
 GLfloat v12[3], v23[3], v31[3];
 GLint i;
 if (depth == 0) {
 drawtriangle(v1, v2, v3);
 return;}
}
```

```
for (i = 0; i < 3; i++) {
 v12[i] = v1[i]+v2[i];
 v23[i] = v2[i]+v3[i];
 v31[i] = v3[i]+v1[i];
}
```

```
normalize(v12);
normalize(v23);
normalize(v31);
subdivide(v1,v12,v31,depth-1);
subdivide(v2,v23,v12,depth-1);
subdivide(v3,v31,v23,depth-1);
subdivide(v12,v23,v31,depth-1);
}
```

## Generalized Subdivision

A recursive subdivision technique such as the one described in Example 5 can be used for other types of surfaces.

Typically, the recursion ends either when a certain depth is reached or when some condition on the curvature is satisfied (highly curved parts of surfaces look better with more subdivision).

To look at a more general solution to the problem of subdivision, consider an arbitrary surface parameterized by two variables  $u[0]$  and  $u[1]$ . Suppose that two routines are provided:

```
void surf(GLfloat u[2], GLfloat vertex[3],
 GLfloat normal[3]);
```

```
float curv(GLfloat u[2]);
```

If `surf()` is passed `u[]`, the corresponding three-dimensional vertex and normal vectors (of length 1) are returned. If `u[]` is passed to `curv()`, the curvature of the surface at that point is calculated and returned.

Example 6 shows the recursive routine that subdivides a triangle either until the maximum depth is reached or until the maximum curvature at the three vertices is less than some cutoff.

```
void subdivide(float u1[2], float u2[2], float u3[2],
 float cutoff, long depth) {
 GLfloat v1[3], v2[3], v3[3],
 n1[3], n2[3], n3[3];
 GLfloat u12[2], u23[2],
 u32[2];
 GLint i;
```

```
 if (depth == maxdepth || (curv(u1) < cutoff &&
 curv(u2) < cutoff && curv(u3) < cutoff)) {
 surf(u1, v1, n1); surf(u2, v2, n2); surf(u3, v3, n3);
 glBegin(GL_POLYGON);
```

```
 glNormal3fv(n1); glVertex3fv(v1);
 glNormal3fv(n2); glVertex3fv(v2);
 glNormal3fv(n3); glVertex3fv(v3);
 glEnd();
 return;}

```

```
for (i = 0; i < 2; i++) {
 u12[i] = (u1[i] + u2[i])/2.0;
 u23[i] = (u2[i] + u3[i])/2.0;
 u31[i] = (u3[i] + u1[i])/2.0;
}
```

```
subdivide(u1, u12, u31, cutoff, depth+1);
subdivide(u2, u23, u12, cutoff, depth+1);
subdivide(u3, u31, u23, cutoff, depth+1);
subdivide(u12, u23, u31, cutoff, depth+1);}
```

# Computer Graphics

## Lecture 39



# Computer Graphics

## Lecture 40 Fractals

Taqdees A. Siddiqi  
cs602@vu.edu.pk

Fractal are geometric patterns that is repeated at ever smaller scales to produce irregular shapes and surfaces that can not be represented by classical geometry.

Fractals are used in computer modeling of irregular patterns and structure in nature.

According to Webster's Dictionary a fractal is defined as being "derived from the Latin fractus meaning broken, uneven: any of various extremely irregular curves or shape that repeat themselves at any scale on which they are examined."

Mandelbrot, the discoverer of fractals gives two definitions:

"I coined fractal from the Latin adjective fractus. The corresponding Latin verb frangere means 'to break:' to create irregular fragments"

Every set with a non-integer (Hausdorff-Besicovitch) dimension ( $D$ ) is a fractal. However not every fractal has an integer  $D$ . A fractal is by definition a set for which  $D$  strictly exceeds the topological dimension ( $D^{\wedge}$ ).

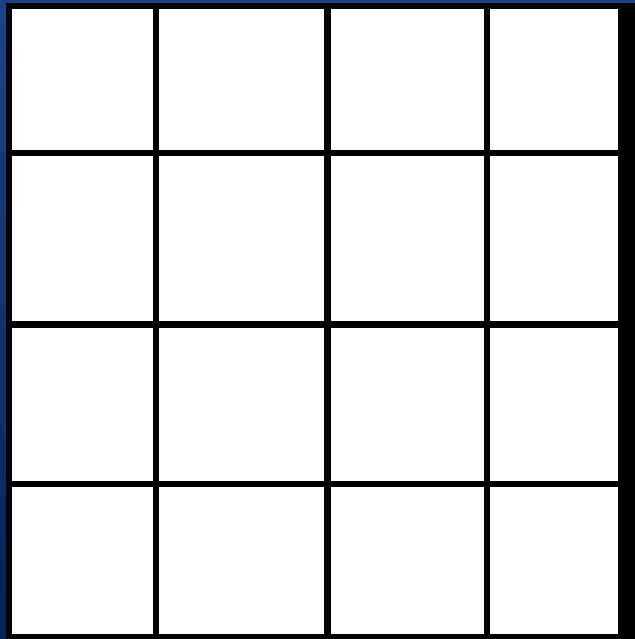
# Hausdorff-Besicovitch(Fractal Dimension)

To understand the second definition we need to be able to understand the fractal dimension. So first we have to look at understanding how to calculate the dimension of an object. Below we have three different objects.

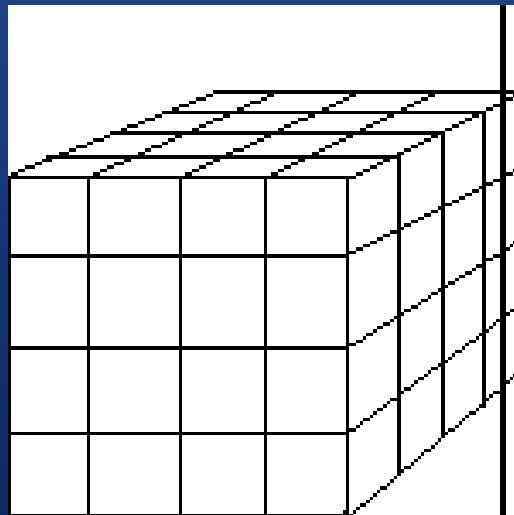
As you can see the line is broken into 4 smaller lines. Each of these lines is similar to the original line, but they are all  $\frac{1}{4}$  the scale. This is the idea of self similarity.



The square ahead is also broken into smaller pieces. Each of which is  $\frac{1}{4}$ th the size of the original. In this case it takes 16 of the smaller pieces to create the original.



As with the others the cube is also broken down into smaller cubes of  $\frac{1}{4}$  the size of the original. It takes 64 of these smaller cubes to create the the original cube.



By looking at this we begin to see a pattern:

$$4 = 4^1$$

$$16 = 4^2$$

$$64 = 4^3$$

This gives us the equation:

$$N = S^D$$

Where  $N$  is the number of small pieces that go into the larger one,  $S$  is the scale to which the smaller pieces compare to the larger one and  $D$  is the dimension.

Now solve for  $D$  in the previous equation; we find:

$$D = \log N / \log S$$

This dimension is the Hausdorff-Besicovitch dimension.

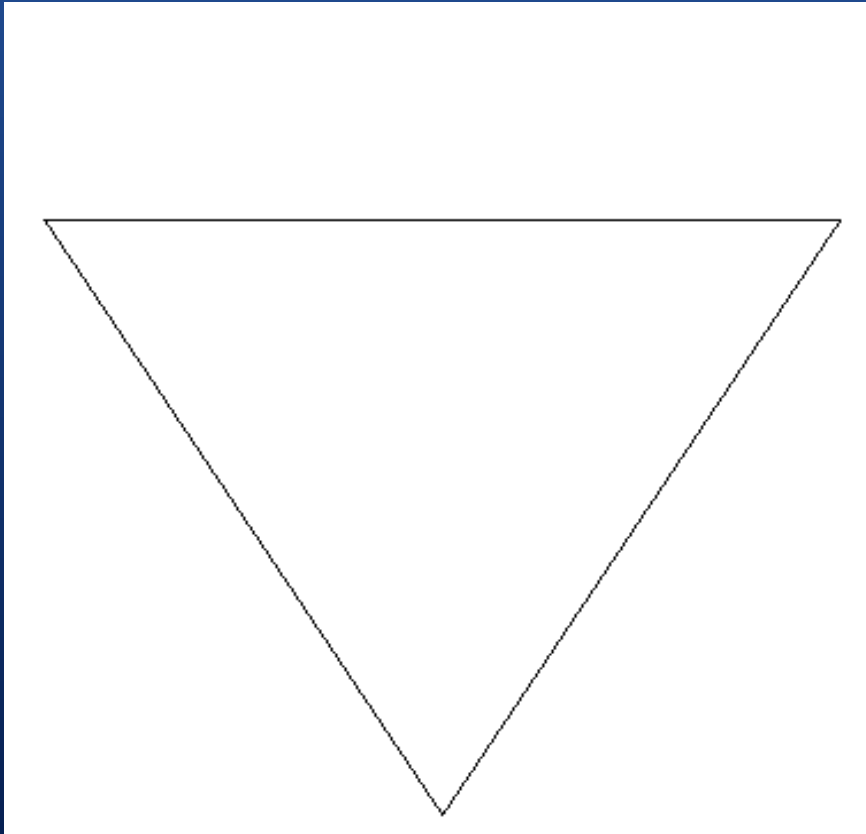


# Koch Curve

Euclidean Geometry is the geometry of lines, planes, circles etc. It's simple and it works, and for a long time, mathematicians thought it was a reasonable representation of nature.

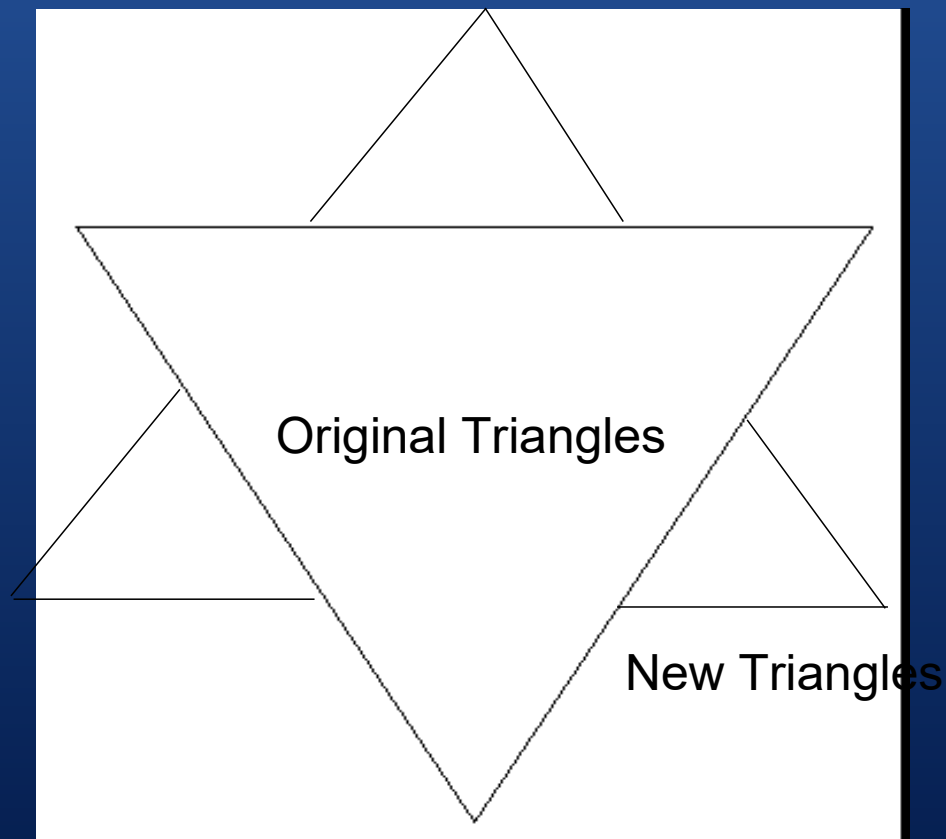
However, people soon discovered that they could draw (or at least begin to draw) certain curves and surfaces that could not be described by the classical geometry.

How hard can it be to draw a curve? Let us attempt to describe. This is the Koch curve:

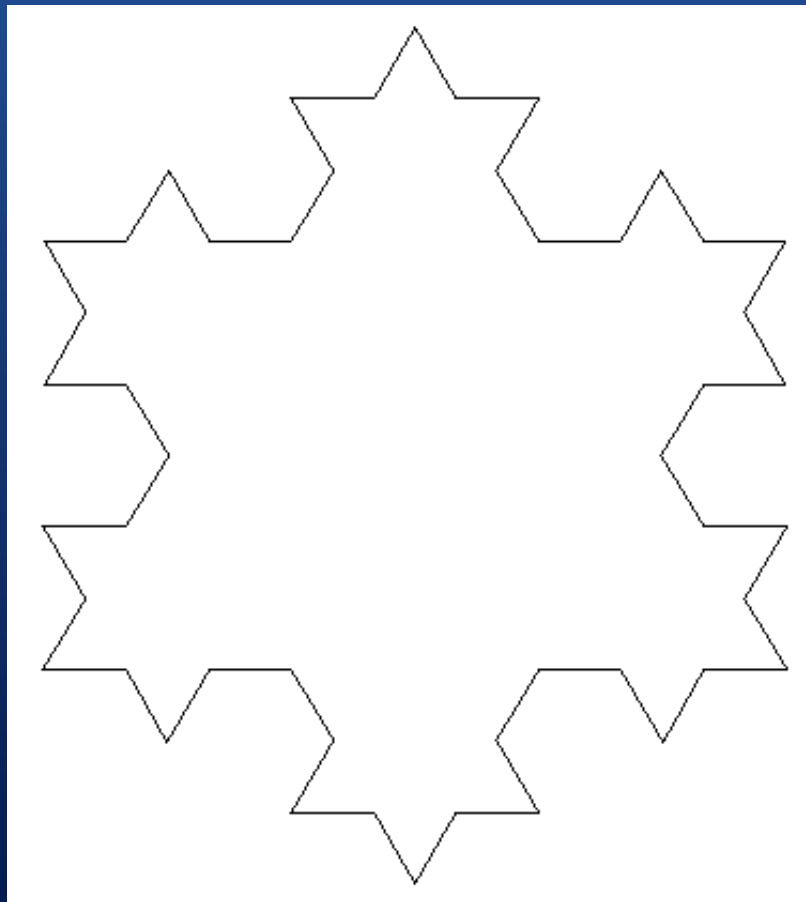


Draw a triangle.

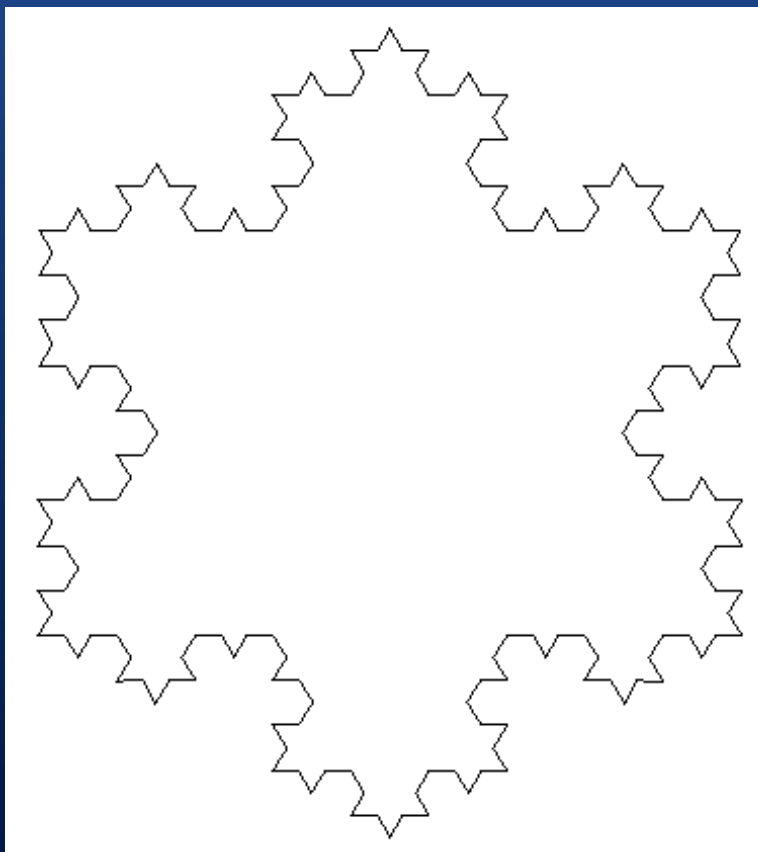
If we say that each line is of length 1, then the total length of the curve is 3.



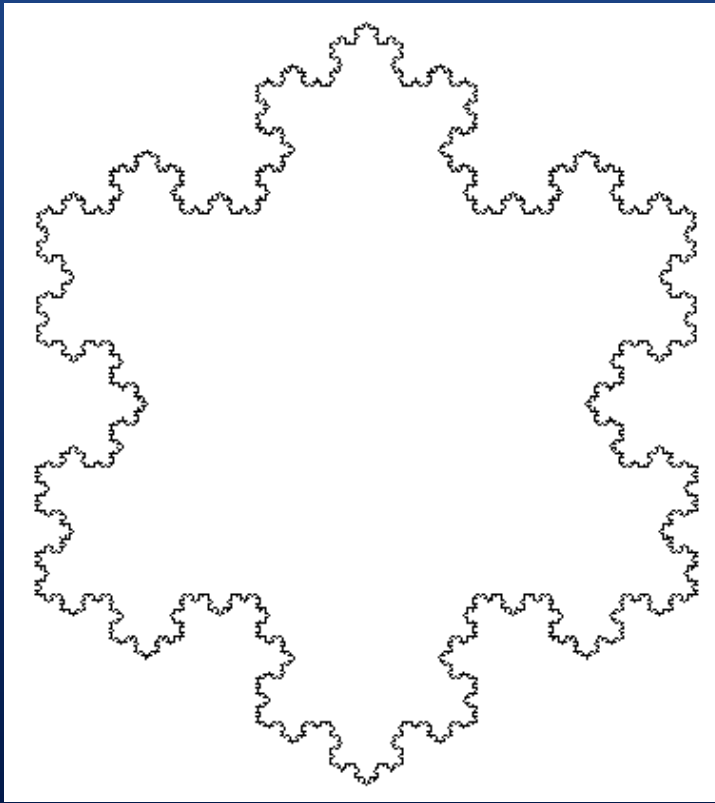
Now take each edge in turn and add another triangle, a third of the size. So now there are 12 edges and 12 points. The length of the curve is now 4.  
Repeat the process again, and again, forever.



length = 5.3333



length = 7.1111



length = 12.6420

As we continue adding edges, the length of the curve increases. If we add edges forever, then length of the curve reaches infinity, but the whole curve nevertheless covers a finite area.

The curve is infinitely detailed. No matter how closely we zoom into the image, it always shows up more detail.

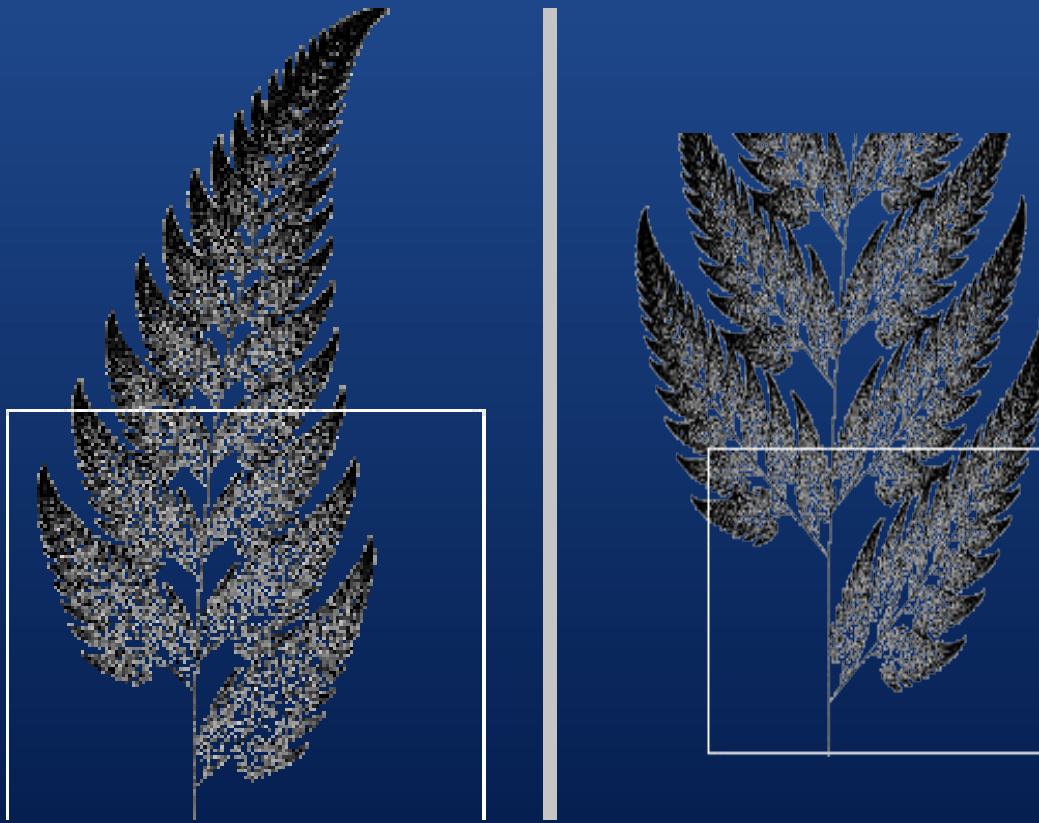
### Self Similarity

So what do these mathematical curiosities have to do with the real world? Well, everything as it turns out. Such objects turn up all the time in the natural world.

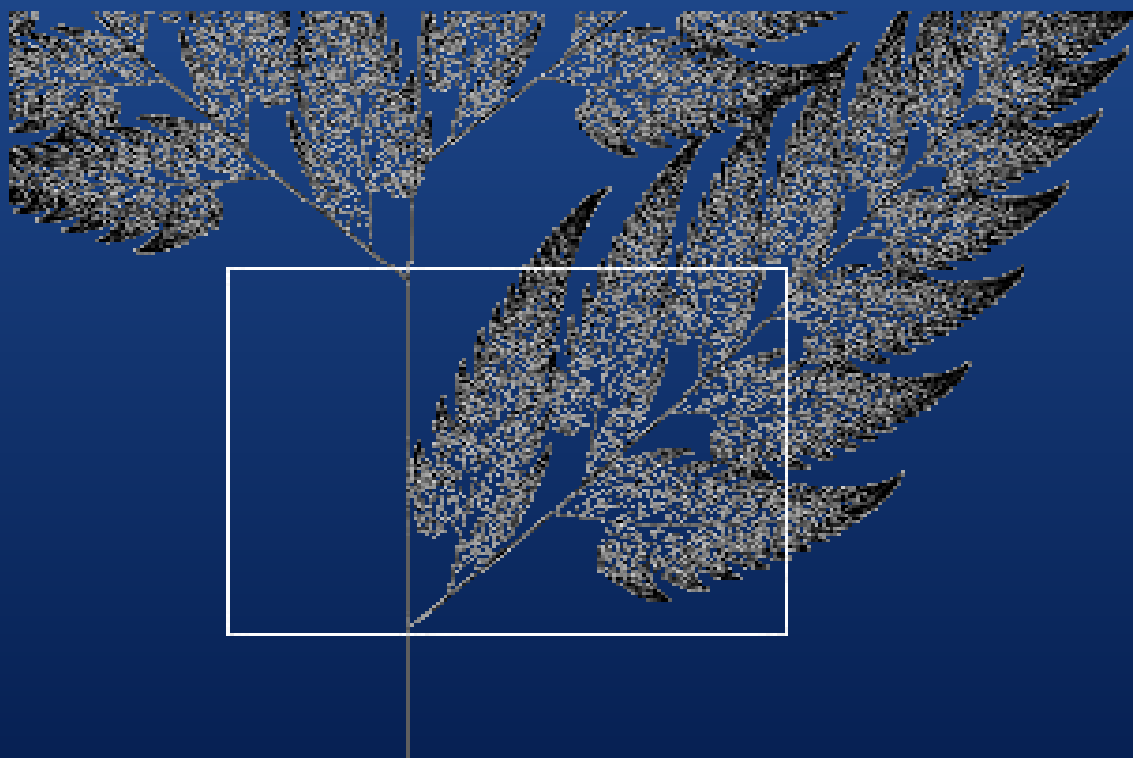
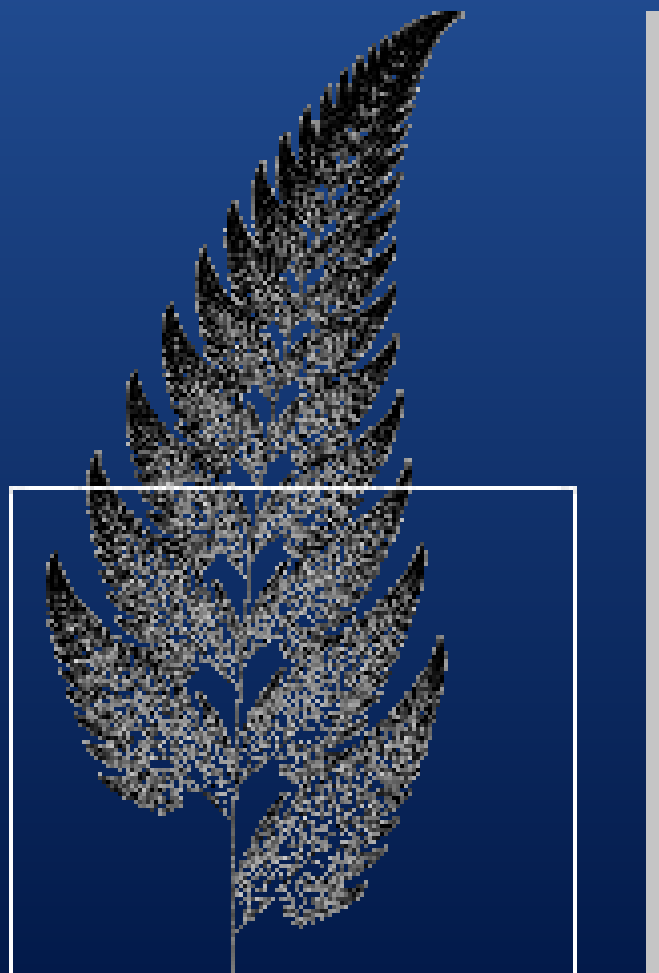
Lets take a look at a common plant, the fern. A fern consists of a leaf, which is made up from many similar, but smaller leaves, each of which, in turn, is made from even smaller leaves.

The following figure is a standard fern. We will see the overall theme of repeating leaves. Each smaller leaf looks similar to the larger leaf.





Looking a little closer, we can see that those small leaves are made up from even smaller leaves.



Of course, in reality, a fern does have a smallest leaf, though we're sure every fern aspires to be like that one. What is interesting is that the program to generate this image is only a few lines long.

## Fractal Geometry

Almost all geometric forms used for building man made objects belong to Euclidean geometry, they are comprised of lines, planes, rectangular volumes, arcs, cylinders, spheres, etc. These elements can be classified as belonging to an integer dimension, either 1, 2, or 3.

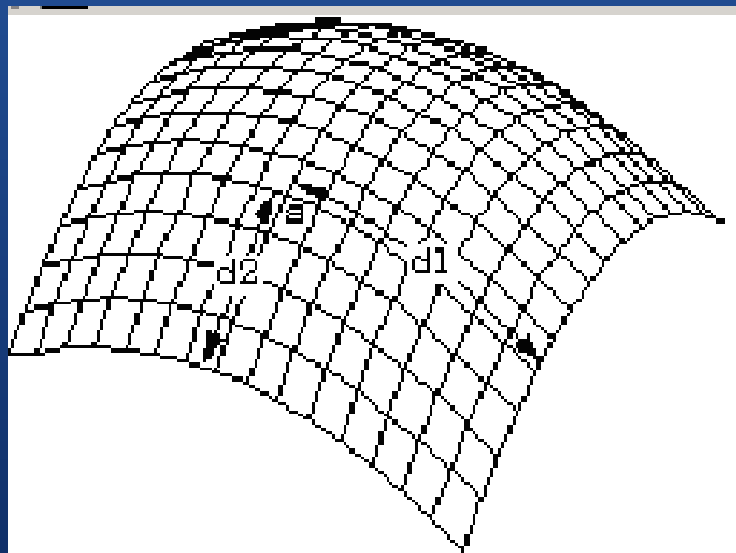
This concept of dimension can be described both intuitively and mathematically. Intuitively we say that a line is one dimensional because it only takes 1 number to uniquely define any point on it.

That one number could be the distance from the start of the line. This applies equally well to the circumference of a circle, a curve, or the boundary of any object.



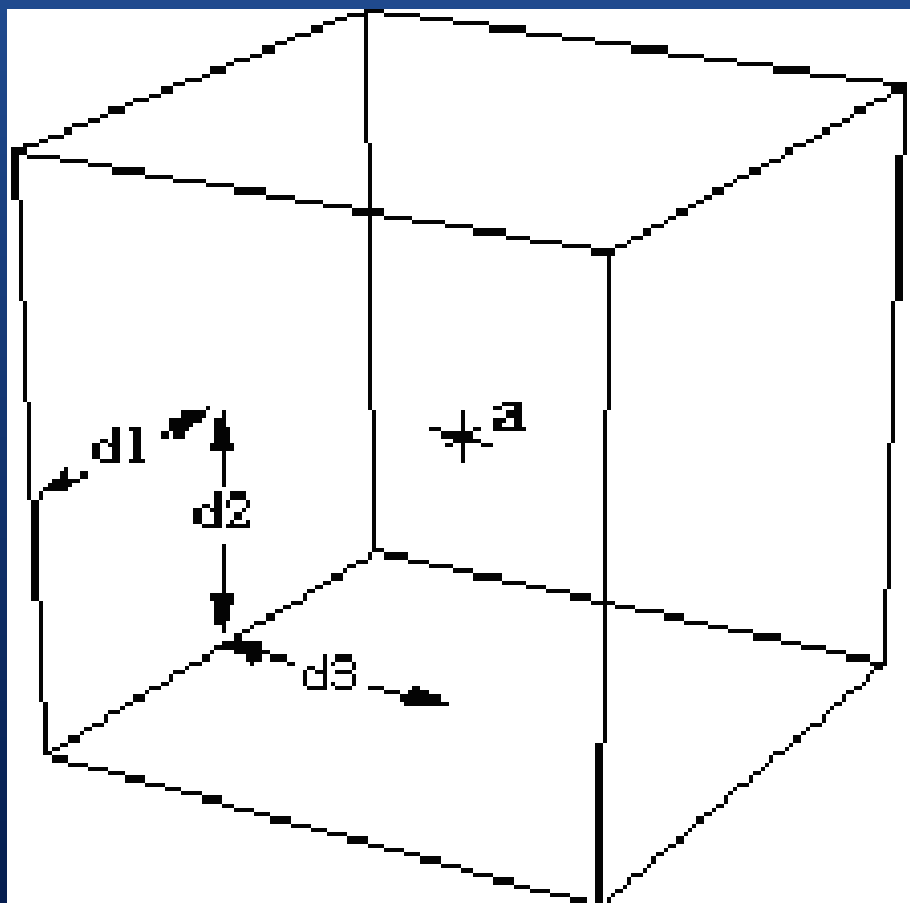
Any point “a” on one dimension curve can be represented by one number, the distance  $d$  from the start point.

A plane is two dimensional since in order to uniquely define any point on its surface we require two numbers. There are many ways to arrange the definition of these two numbers but we normally create an orthogonal coordinate system.



One of the two possible methods is to grid the surface and measure two distances along the grid lines.

The volume of some solid object is 3 dimensional on the same basis as above, it takes three numbers to uniquely define any point within the object.



Any point “a” on three dimensions can be uniquely represented by three numbers. Typically these three numbers are the coordinates of the point using an orthogonal coordinate system.

A more mathematical description of dimension is based on how the "size" of an object behaves as the linear dimension increases. In one dimension consider a line segment.



If the linear dimension of the line segment is doubled then obviously the length (characteristic size) of the line has doubled. In two dimensions, if the linear dimensions of a rectangle for example is doubled then the characteristic size, the area, increases by a factor of 4.

In three dimensions if the linear dimension of a box are doubled then the volume increases by a factor of 8. This relationship between D, L and the resulting increase in size S can be generalised and written as

$$S = L^D$$

This is just telling us mathematically what we know from everyday experience. If we scale a two dimensional object for example then the area increases by the square of the scaling.

If we scale a 3-dimensional object the volume increases by the cube of the scale factor. Rearranging the above gives an expression for dimension depending on how the size changes as a function of linear scaling, namely

$$D = \log(S)/\log(L)$$

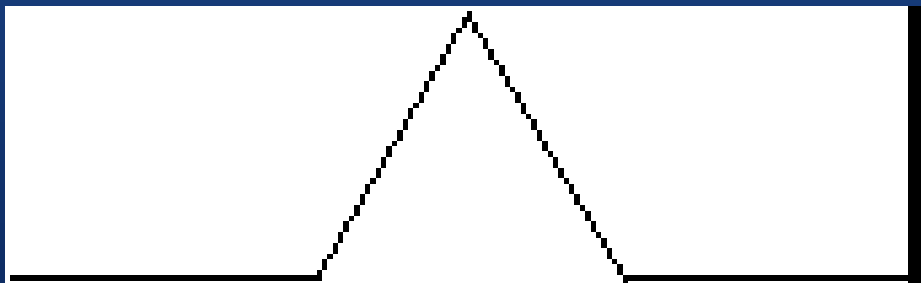
In the examples above the value of  $D$  is an integer, either 1, 2, or 3, depending on the dimension of the geometry. This relationship holds for all Euclidean shapes.

There are however many shapes which do not conform to the integer based idea of dimension given above in both the intuitive and mathematical descriptions.

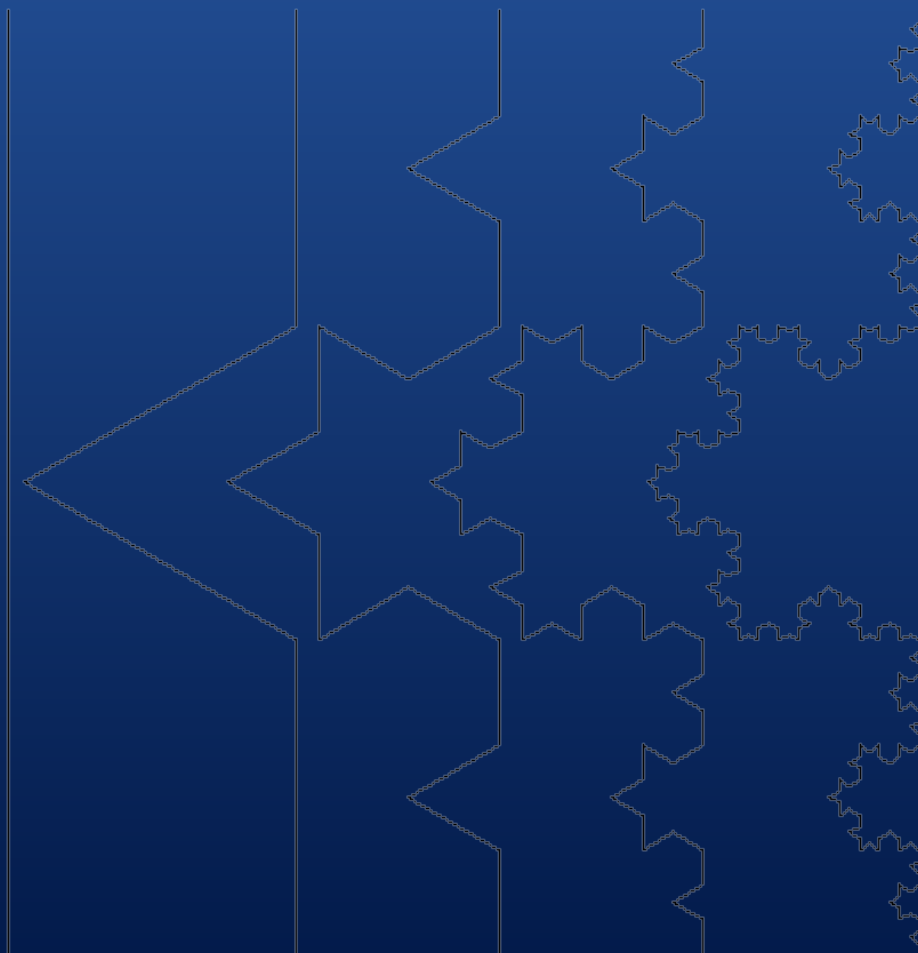
That is, there are objects which appear to be curves for example but which a point on the curve cannot be uniquely described with just one number. If the earlier scaling formulation for dimension is applied the formula does not yield an integer.

There are shapes that lie in a plane but if they are linearly scaled by a factor  $L$ , the area does not increase by  $L$  squared but by some non integer amount. These geometries are called fractals!

One of the simpler fractal shapes is the von Koch snowflake. The method of creating this shape is to repeatedly replace each line segment with the following 4 line segments.



The process starts with a single line segment and continues for ever. The first few iterations of this procedure are shown ahead.



This demonstrates how a very simple generation rule for this shape can generate some unusual (fractal) properties. Unlike Euclidean shapes this object has detail at all levels.

If one magnifies an Euclidean shape such as the circumference of a circle it becomes a different shape, namely a straight line. If we magnify this fractal more and more detail is uncovered, the detail is self similar or rather it is exactly self similar.

Note also that the "curve" on the right is not a fractal but only an approximation of one. This is no different from when one draws a circle, it is only an approximation to a perfect circle. At each iteration the length of the curve increases by a factor of  $4/3$ .

Thus the limiting curve is of infinite length and indeed the length between any two points of the curve is infinite. This curve manages to compress an infinite length into a finite area of the plane without intersecting itself!



Considering the intuitive notion of 1 dimensional shapes, although this object appears to be a curve with one starting point and one end point, it is not possible to uniquely specify any position along the curve with one number as we expect to be able to do with Euclidean curves which are 1 dimensional.

Although the method of creating this curve is straightforward, there is no algebraic formula that describes the points on the curve. Some of the major differences between fractal and Euclidean geometry are outlined in the following table.

| Fractal                                                                                                          | Euclidean                                                                                                                                |
|------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------|
| Modern Invention<br>No specific size or scale<br>Appropriate for geometry in nature<br>Described by an algorithm | Traditional<br>Based on a characteristic size or scale<br>Suits description of man made objects<br>Described by a usually simple formula |

Firstly the recognition of fractal is very modern, formally been studied in the last 10 years compared to Euclidean geometry back over 2000 years. Secondly whereas Euclidean shapes normally have a few characteristic sizes or length scales fractals have so characteristic sizes.

Fractal shapes are self similar and independent of size or scaling. Third, Euclidean geometry provides a good description of man made objects whereas fractals are required for a representation of naturally occurring geometries.

It is likely that this limitation of our traditional language of shape is responsible for the striking difference between mass produced objects and natural shapes. Finally, Euclidean geometries are defined by algebraic formulae, for example

# Computer Graphics

## Lecture 41 Viewing Using OpenGL

Taqdees A. Siddiqi  
cs602@vu.edu.pk

Today we will practically implement viewing a geometric model in any orientation by transforming it in three-dimensional space and control the location in three-dimensional space (POV) from where the model is viewed.

Also we will see:

Clipping undesired portions

Modeling transformation

Projecting the model and

Combining transformations

We will also discuss on how to instruct OpenGL to draw the geometric models. Now we must decide how we want to position the models in the scene, and we must choose a vantage point from where to view the scene.

Use the default positioning and vantage point,  
Or specify position and vantage point  
Choose a viewpoint

If we want to look at the corner of the room containing a globe, then we must decide:

- how far away from the scene is the viewer?
- and where exactly should the viewer be?

We would like to ensure that the final image of the scene contains a good view:

that a portion of the floor is visible

all the objects in the scene are visible

objects are presented in an interesting arrangement.

Now how to use OpenGL to accomplish these tasks:

how to position and orient models in three-dimensional space

how to establish the location in three-dimensional space of the viewpoint

All of these factors help determine exactly what image appears on the screen.

Although ultimately a 2-D image of 3-D models is drawn;

yet we need to think in 3-D while making many of the decisions that determine what gets drawn on the screen.



A series of three computer operations converts an object's 3-D coordinates to pixel positions on the screen. Transformations, which are represented by matrix multiplication, include modeling, viewing, and projection operations.

Such operations include:

1. rotation,
2. translation,
3. scaling,
4. reflection,
5. orthographic projection, and
6. perspective projection.

Generally, we use a combination of several transformations to draw a scene.

Since the scene is rendered on a rectangular window, objects (or parts of objects) that lie outside the window must be clipped. In three-dimensional computer graphics, clipping occurs by throwing out objects on one side of a clipping plane.

Finally, a correspondence must be established between the transformed coordinates and screen pixels. This is known as a viewport transformation.

# Overview: The Camera Analogy

The transformation process to produce the desired scene for viewing is analogous to taking a photograph with a camera. As shown in Figure 1 ahead, the steps might be the following:

0. Set up our tripod, pointing the camera towards the scene  
(viewing transformation).

1. Arrange the scene to be photographed into the desired composition (modeling transformation)
2. Choose a camera lens or adjust the zoom (projection transformation)

3. Determine how large we want the final photograph to be - we might want it enlarged (viewport transformation)
4. After these steps are performed, the picture can be snapped or the scene can be drawn

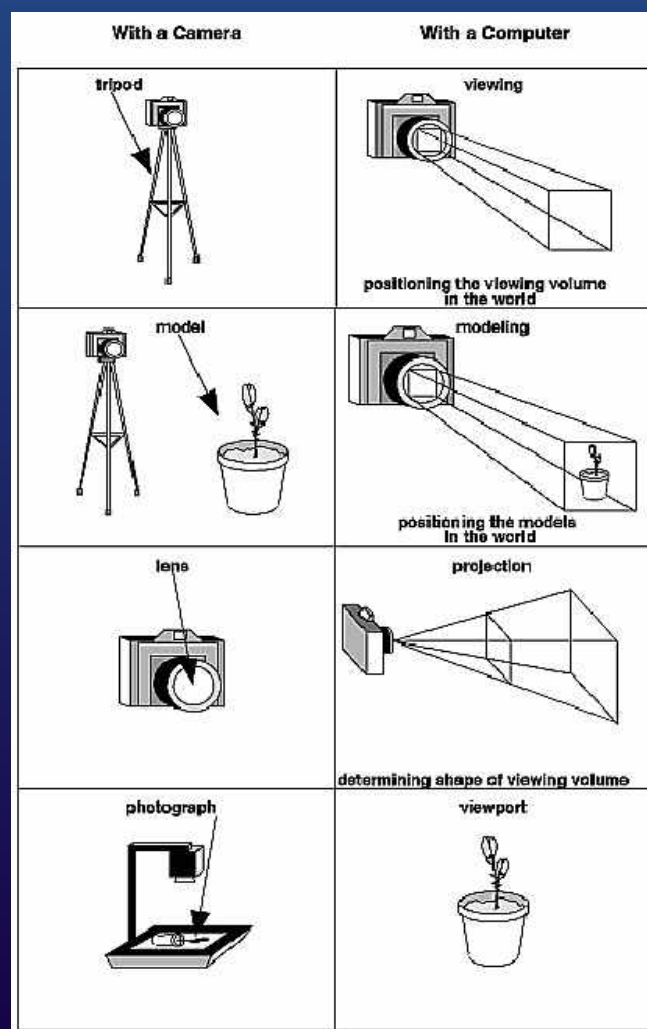


Figure 1: The Camera Analogy

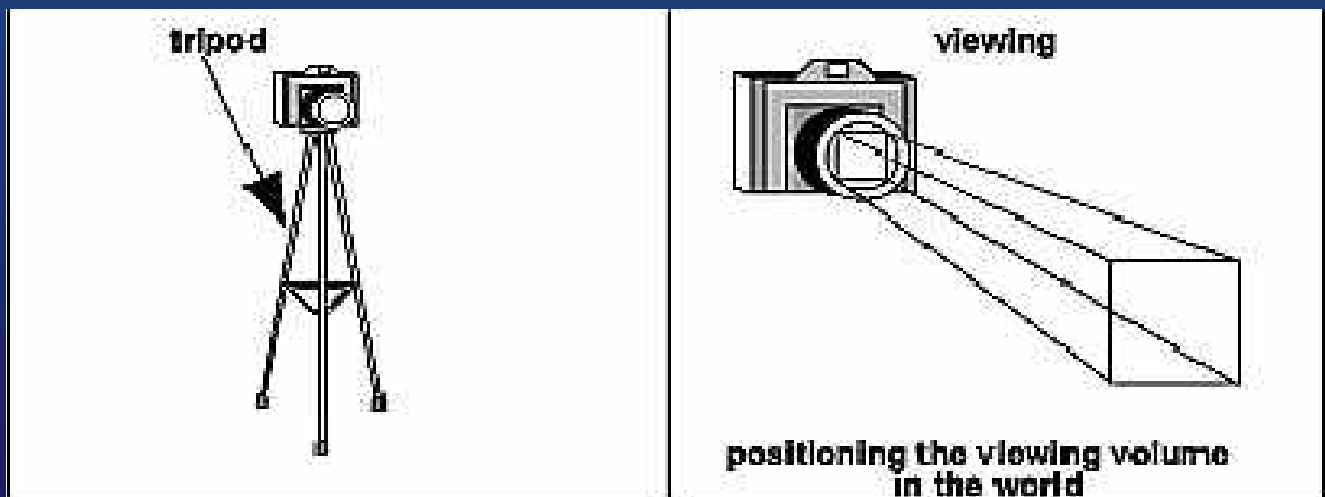


Figure 1-a : The Camera Analogy

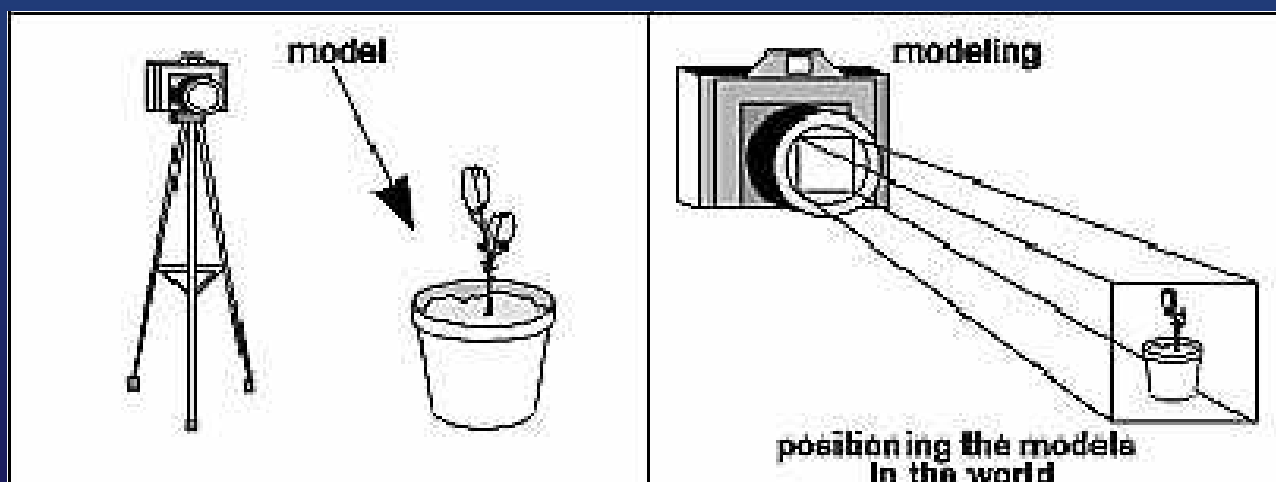


Figure 1-b : The Camera Analogy

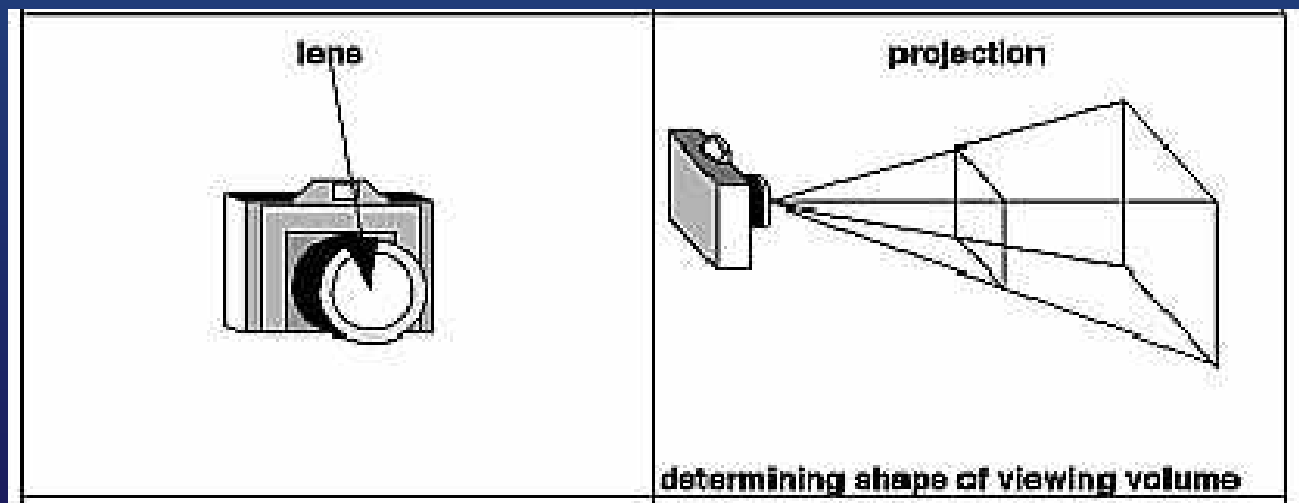


Figure 1-c : The Camera Analogy

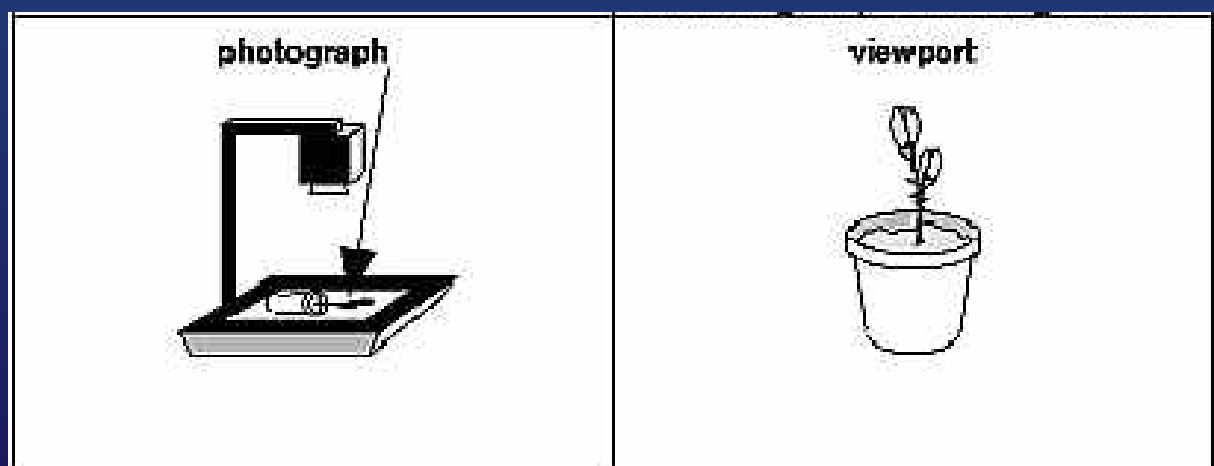


Figure 1-d : The Camera Analogy

Note that these steps correspond to the order in which we specify the desired transformations in our program, not necessarily the order in which the relevant mathematical operations are performed on an object's vertices.

The viewing transformations must precede the modeling transformations in our code, but we can specify the projection and viewport transformations at any point before drawing occurs. Figure 2 ahead shows the order in which these operations occur on our computer.



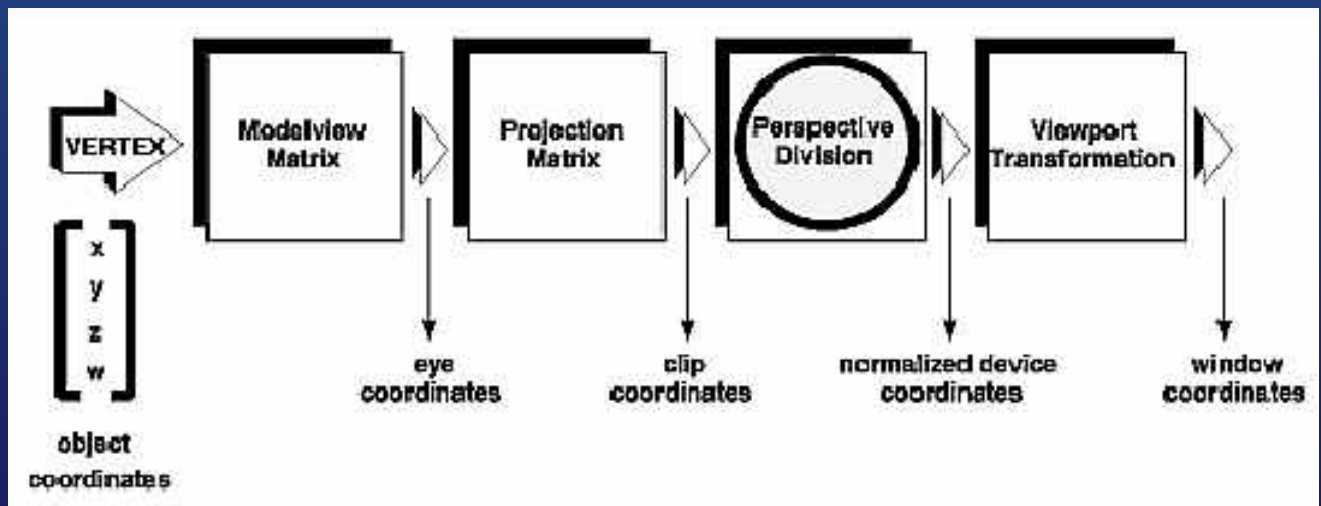


Figure 2: Stages of Vertex Transformation

To specify viewing, modeling, and projection transformations, we construct a  $4 \times 4$  matrix  $M$ , which is then multiplied by the coordinates of each vertex  $v$  in the scene to accomplish the transformation  $v' = Mv$

(Remember that vertices always have four coordinates (x, y, z, w), though in most cases w is 1 and for two-dimensional data z is 0.) Note that viewing and modeling transformations are automatically applied to surface normal vectors, in addition to vertices.

(Normal vectors are used only in eye coordinates.) This ensures that the normal vector's relationship to the vertex data is properly preserved.

The viewing and modeling transformations we specify are combined to form the modelview matrix, which is applied to the incoming object coordinates to yield eye coordinates.

Next, if we've specified additional clipping planes to remove certain objects from the scene or to provide cutaway views of objects, these clipping planes are applied.

After that, OpenGL applies the projection matrix to yield clip coordinates. This transformation defines a viewing volume; objects outside this volume are clipped so that they're not drawn in the final scene.

After this point, the perspective division is performed by dividing coordinate values by  $w$ , to produce normalized device coordinates.

Finally, the transformed coordinates are converted to window coordinates by applying the viewport transformation. We can manipulate the dimensions of the viewport to cause the final image to be enlarged, shrunk, or stretched.

We might correctly suppose that the  $x$  and  $y$  coordinates are sufficient to determine which pixels need to be drawn on the screen.

However, all the transformations are performed on the z coordinates as well. This way, at the end of this transformation process, the z values correctly reflect the depth of a given vertex.

One use for this depth value is to eliminate unnecessary drawing. OpenGL can use this information to determine which surfaces are obscured by other surfaces and can then avoid drawing the hidden surfaces.

As we've probably guessed by now, we need to know a few things about matrix mathematics to get the most out of this lecture as we have learnt from previous lectures.

## **A Simple Example: Drawing a Cube**

Example 1 draws a cube that's scaled by a modeling transformation see Figure 3 ahead.

The viewing transformation, `gluLookAt()`, positions and aims the camera towards where the cube is drawn.

A projection transformation and a viewport transformation are also specified. The rest of this section explains the transformation commands it uses.



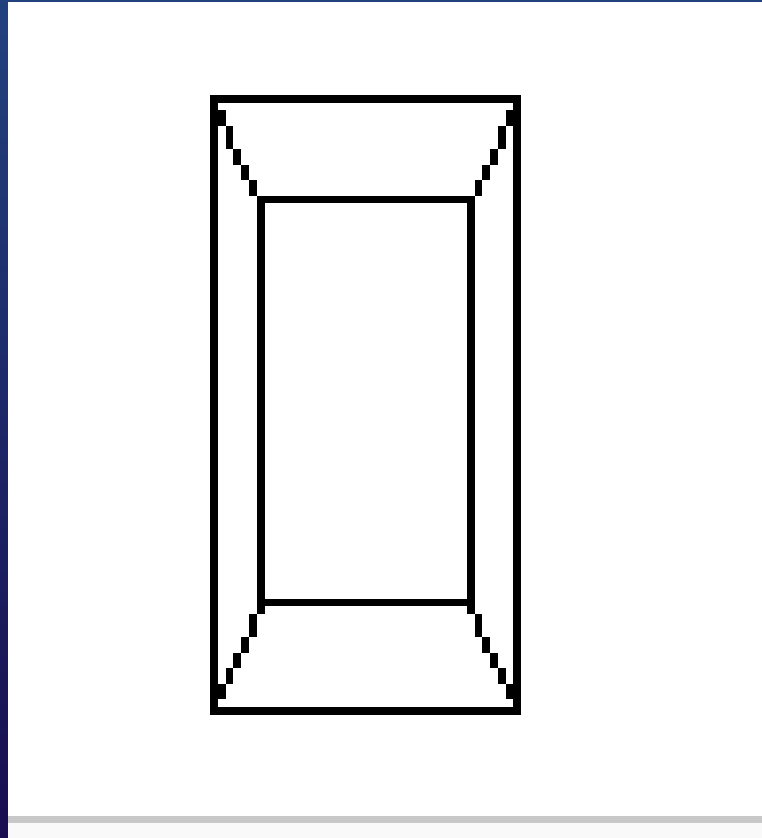


Figure 3: Transformed Cube

### Example 1 : Transformed Cube

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
void init(void){
 glClearColor (0.0, 0.0, 0.0, 0.0);
 glShadeModel (GL_FLAT);
}
```

```
void display(void){
 glClear (GL_COLOR_BUFFER_BIT);
 glColor3f (1.0, 1.0, 1.0);
 glLoadIdentity();
 gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
 glScalef (1.0, 2.0, 1.0);
```

```
 glutWireCube (1.0);
 glFlush ();
}
```

```
void reshape (int w, int h){glViewport (0, 0, (GLsizei) w,
 (GLsizei) h);glMatrixMode
 (GL_PROJECTION);glLoadIdentity ();
 glFrustum (-1.0, 1.0, -1.0, 1.0, 1.5, 20.0);
 glMatrixMode (GL_MODELVIEW);
}
```

```
int main(int argc, char** argv){glutInit(&argc, argv);
 glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
 glutInitWindowSize (500, 500);
 glutInitWindowPosition (100, 100);
 glutCreateWindow (argv[0]);
```

```
init ();
glutDisplayFunc(display);
glutReshapeFunc(reshape);
glutMainLoop();
return 0;
}
```

## The Viewing Transformation

Recall that the viewing transformation is analogous to positioning and aiming a camera. In this code example, before the viewing transformation can be specified, the current matrix is set to the identity matrix with `glLoadIdentity()`.

This step is necessary. If we don't clear the current matrix by loading it with the identity matrix, we continue to combine previous transformation matrices with the new one we supply.

In some cases, we do want to perform such combinations, but we also need to clear the matrix sometimes. In Example 1, after the matrix is initialized, the viewing transformation is specified with `gluLookAt()`.

The arguments for this command indicate where the camera (or eye position) is placed, where it is aimed, and which way is up.

The arguments used here place the camera at  $(0, 0, 5)$ , aim the camera lens towards  $(0, 0, 0)$ , and specify the up-vector as  $(0, 1, 0)$ . The up-vector defines a unique orientation for the camera.

If `gluLookAt()` was not called, the camera has a default position and orientation. By default, the camera is situated at the origin, points down negative z-axis, and has an up-vector of  $(0, 1, 0)$ . In Example 1, the overall effect is that `gluLookAt()` moves the camera 5 units along the z-axis.

# The Modeling Transformation

We use the modeling transformation to position and orient the model. For example, we can rotate, translate, or scale the model or perform some combination of these operations.



In Example 1, `glScalef()` is the modeling transformation used. The arguments for this command specify how scaling occurs along the axes. If all arguments are 1.0, command has no effect. In Example 1, the cube is drawn twice as large in the y direction.

Thus, if one corner of the cube had originally been at (3.0, 3.0, 3.0), that corner would wind up being drawn at (3.0, 6.0, 3.0). The effect of this modeling transformation is to transform the cube so that it isn't a cube but a rectangular box.

Now change the `gluLookAt()` call in Example 1 to the modeling transformation `glTranslatef()` with parameters (0.0, 0.0, -5.0). The result should look exactly the same as when we used `gluLookAt()`. Why are the effects of these two commands similar?

Note that instead of moving the camera (with a viewing transformation) so that the cube could be viewed, we could have moved the cube.

This duality in the nature of viewing and modeling transformations is the reason why we need to think about the effect of both types of transformations simultaneously.

It doesn't make sense to try to separate the effects, but sometimes it's easier to think about them one way rather than the other. This is also why modeling and viewing transformations are combined into the modelview matrix before the transformations are applied.

Also note that the modeling and viewing transformations are included in the `display()` routine, along with the call that's used to draw the cube, `glutWireCube()`.

This way, `display()` can be used repeatedly to draw the contents of the window if, for example, the window is moved or uncovered, and we've ensured that each time, the cube is drawn in the desired way, with the appropriate transformations.

The potential repeated use of `display()` underscores the need to load the identity matrix before performing the viewing and modeling transformations, especially when other transformations might be performed between calls to `display()`.

## **The Projection Transformation**

Specifying the projection transformation is like choosing a lens for a camera. We can think of this as determining what the field of view is and therefore what objects are inside it and to some extent how they look.

This is equivalent to choosing among wide-angle, normal, and telephoto lenses, for example, with a wide-angle lens, we can include a wider scene in the final photograph than with a telephoto lens.

But a telephoto lens allows us to photograph objects as though they're closer to us than they actually are.

In computer graphics, we don't have to pay \$10,000 for a 2000-millimeter telephoto lens; once we've bought our graphics workstation, all we need to do is use a smaller number for our field of view.

In addition to the field-of-view considerations, the projection transformation determines how objects are projected onto the screen, as its name suggests.

Two basic types of projections are provided by OpenGL, along with several corresponding commands for describing the relevant parameters. One type is the perspective projection, how we see things in daily life.



For example, railroad tracks.

If we're trying to make realistic pictures, we'll want to choose perspective projection, which is specified with the `glFrustum()` command in this code example.

Orthographic projection is used in architectural and computer-aided design applications.

Before `glFrustum()` can be called to set the projection transformation, some preparation needs to happen. As shown in the `reshape()` routine in Example 1, the command called `glMatrixMode()` is used first, with the argument `GL_PROJECTION`.

This indicates that the current matrix specifies the projection transformation; the following transformation calls then affect the projection matrix. As we can see, a few lines later `glMatrixMode()` is called again, this time with `GL_MODELVIEW` as the argument.

This indicates that succeeding transformations now affect the modelview matrix instead of the projection matrix.

Note that `glLoadIdentity()` is used to initialize the current projection matrix so that only the specified projection transformation has an effect.

Now `glFrustum()` can be called, with arguments that define the parameters of the projection transformation.

In this example, both the projection transformation and the viewport transformation are contained in the reshape() routine, which is called when the window is first created and whenever the window is moved or reshaped.

This makes sense, since both projecting (the width to height aspect ratio of the projection viewing volume) and applying the viewport relate directly to the screen, and specifically to the size or aspect ratio of the window on the screen.

Change the `glFrustum()` call in Example 1 to the more commonly used Utility Library routine `gluPerspective()` with parameters (60.0, 1.0, 1.5, 20.0). Then experiment with different values, especially for fov(field of view ), near and far plane.

# Computer Graphics

## Lecture 41

# **Computer Graphics**

## **Lecture 42**

### **Viewing Using OpenGL II**

**Taqdees A. Siddiqi**  
**cs602@vu.edu.pk**

**Viewing and Modeling  
Transformations**

Viewing and modeling transformations are related in OpenGL and are combined into a single modelview matrix. One of the toughest problems newcomers to computer graphics face is understanding the effects of combined three-dimensional transformations.

As we've already seen, there are alternative ways to think about transformations - do we want to move the camera in one direction, or move the object in the opposite direction?

Each way of thinking about transformations has advantages and disadvantages, but in some cases one way more naturally matches the effect of the intended transformation.

If we can find a natural approach for our particular application, it's easier to visualize the necessary transformations and then write the corresponding code to specify the matrix manipulations.



The first part of this section discusses how to think about transformations; later, specific commands are presented. For now, we use only the matrix-manipulation commands we've already seen.

Finally, keep in mind that we must call `glMatrixMode()` with `GL_MODELVIEW` as its argument prior to performing modeling or viewing transformations.

# Thinking about Transformations

Let's start with a simple case of two transformations: a 45 degree counterclockwise rotation about the origin around the z-axis, and a translation down the x-axis.

Suppose that the object we're drawing is small compared to the translation (so that we can see the effect of the translation), and that it's originally located at the origin.

If we rotate the object first and then translate it, the rotated object appears on the x-axis. If we translate it down the x-axis first and then rotate about the origin, the object is on the line  $y=x$ , as shown in Figure.

In general, the order of transformations is critical. If we do transformation A and then transformation B, we almost always get something different than if we do them in the opposite order.

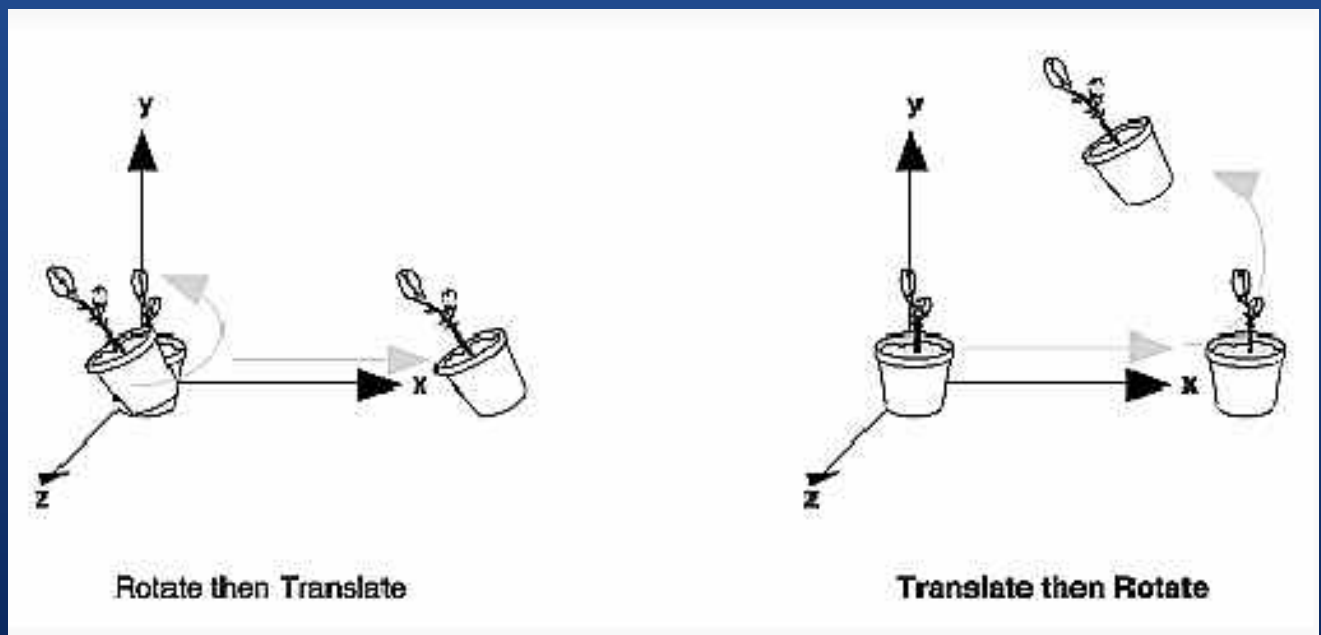


Figure 4: Rotating First or Translating First

All viewing and modeling transformations are represented as  $4 \times 4$  matrices. Each successive `glMultMatrix*()` or transformation command multiplies a new  $4 \times 4$  matrix  $M$  by the current modelview matrix  $C$  to yield  $CM$ .

Finally, vertices  $v$  are multiplied by the current modelview matrix. This process means that the last transformation command called in our program is actually the first one applied to the vertices:  $CMv$ .

Thus, one way of looking at it is to say that we have to specify the matrices in the reverse order.

Like many other things, however, once we've gotten used to thinking about this correctly, backward will seem like forward. Consider the following code sequence, which draws a single point using three transformations:

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glMultMatrixf(N);
glMultMatrixf(M);
glMultMatrixf(L);
glBegin(GL_POINTS);
 glVertex3f(v);
glEnd();
```

In code, the modelview matrix successively contains I, N, NM, and finally NML, where I is identity matrix. The transformed vertex is  $NMLv$ . The vertex transformation is  $N(M(Lv))$ .  $v$  is multiplied by L, resulting  $Lv$  is multiplied by M, and resulting  $MLv$  is multiplied by N.

Notice that the transformations to vertex  $v$  effectively occur in the opposite order than they were specified. (Actually, only a single multiplication of a vertex by the modelview matrix occurs.

In this example, the  $N$ ,  $M$ , and  $L$  matrices are already multiplied into a single matrix before it's applied to  $v$ .)



# Grand, Fixed Coordinate System

Thus, if we like to think in terms of a grand, fixed coordinate system. We have to think of the multiplications as occurring in the opposite order from how they appear in the code.

Using the simple example shown in the left half of Figure 4 (a rotation about the origin and a translation along the x-axis), if we want the object to appear on the axis after the operations, the rotation must occur first;

followed by the translation. To do this, we'll need to reverse the order of operations, so the code looks something like this (where  $R$  is the rotation matrix and  $T$  is the translation matrix):

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glMultMatrixf(T);
glMultMatrixf(R);
draw_the_object();
```

## Moving a Local Coordinate System

Another way to view matrix multiplications is to forget grand, fixed coordinate system in which our model is transformed and instead imagine that a local coordinate system is tied to the object we're drawing. All operations occur relative to this changing coordinate system.

With this approach, the matrix multiplications now appear in the natural order in the code.

To see this in the translation-rotation example begin by visualizing the object with a coordinate system tied to it.

The translation operation moves the object and its coordinate system down the x-axis. Then, the rotation occurs about the (now-translated) origin, so the object rotates in place in its position on the axis.

This approach is what we should use for applications such as articulated robot arms, where there are joints at the shoulder, elbow, and wrist, and on each of the fingers.

To figure out where the tips of the fingers go relative to the body, we'd like to start at the shoulder, go down to the wrist, and so on, applying the appropriate rotations and translations at each joint.

Thinking about it in reverse would be far more confusing. This second approach can be problematic, however, in cases where scaling occurs, and especially so when the scaling is nonuniform.

After uniform scaling, translations move a vertex by a multiple of what they did before, since the co-ordinate system is stretched. Nonuniform scaling mixed with rotations may make the axes of the local coordinate system non-perpendicular.

As mentioned earlier, we normally issue viewing transformation commands in our program before any modeling transformations. This way, a vertex in a model is first transformed into the desired orientation and then transformed by the viewing operation.

Since the matrix multi-plications must be specified in reverse order, the viewing commands need to come first. Note, however, that we don't need to specify either viewing or modeling transformations if we're satisfied with the default conditions.

If there's no viewing transformation, the "camera" is left in the default position at the origin, pointed toward the negative z-axis; if there's no modeling transformation, the model isn't moved, and it retains its specified position, orientation, and size.



Since the commands for performing modeling transformations can be used to perform viewing transformations, modeling transformations are discussed first, even if viewing transformations are actually issued first.

Often, programmers write all the code, which involves transformations to position and orient objects correctly relative to each other. Next, they decide where they want the viewpoint to be relative to the scene they've composed, and then they write the viewing transformations accordingly.

# Modeling Transformations

The three OpenGL routines for modeling transformations are `glTranslate*`(), `glRotate*`(), and `glScale*`(). As we might suspect, these routines transform an object (or coordinate system, if we're thinking of it that way) by moving, rotating, stretching, shrinking, or reflecting it.

All three commands are equivalent to producing an appropriate translation, rotation, or scaling matrix, and then calling `glMultMatrix*()` with that matrix as the argument. However, these three routines might be faster than using `glMultMatrix*()`.

OpenGL automatically computes the matrices for us. In the command summaries that follow, each matrix multiplication is described in terms of what it does to the vertices of a geometric object using the fixed coordinate system approach.

# Translate

```
void glTranslate{fd}(TYPEEx, TYPEy, TYPEz);
```

Multiplies the current matrix by a matrix that moves (translates) an object by the given x, y, and z values (or moves the local coordinate system by the same amounts).

Figure 5 next shows the effect of `glTranslate*`( ).

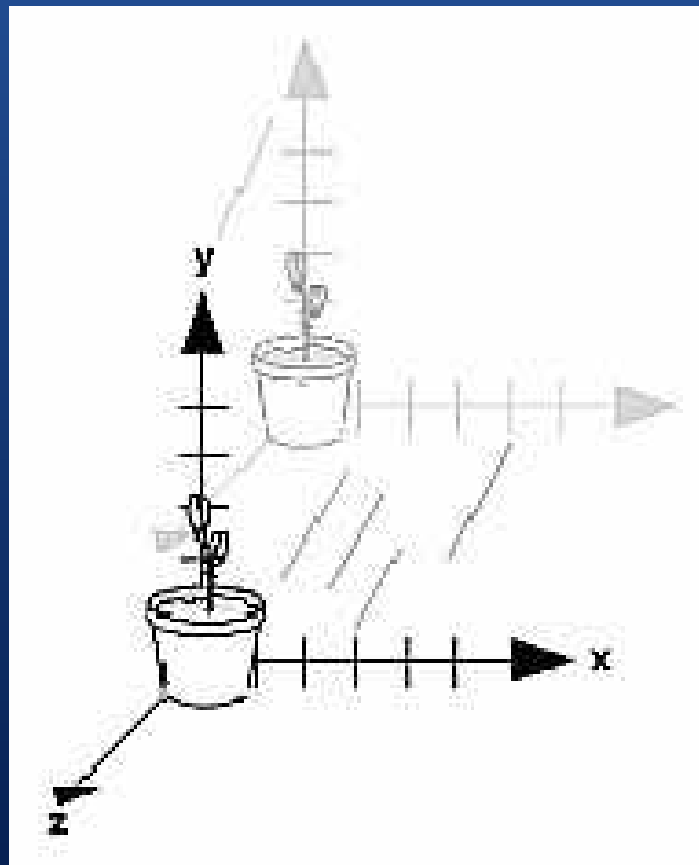


Figure 5 : Translating an Object

Note that using  $(0.0, 0.0, 0.0)$  as the argument for `glTranslate*()` is the identity operation - that is, it has no effect on an object or its local coordinate system.

# Rotate

```
void glRotate{fd}(TYPE angle, TYPE x, TYPE y, TYPE z);
```

Multiplies the current matrix by a matrix that rotates an object (or the local coordinate system) in a counterclockwise direction about the ray from the origin through the point (x, y, z). The angle parameter specifies the angle of rotation in degrees.

The effect of `glRotatef(45.0, 0.0, 0.0, 1.0)`, which is a rotation of 45 degrees about the z-axis, is shown in Figure 6.

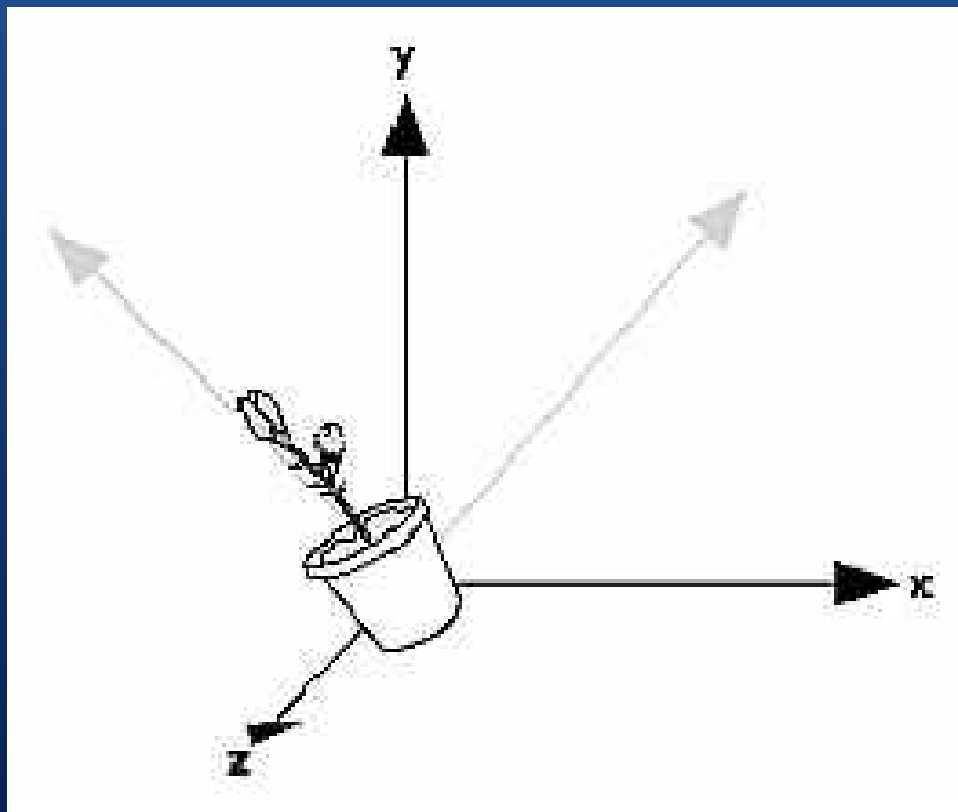


Figure 6 : Rotating an Object

Note that an object that lies farther from the axis of rotation is more dramatically rotated (has a larger orbit) than an object drawn near the axis. Also, if the angle argument is zero, the `glRotate*()` command has no effect.

## Scale



```
void glScale{fd}(TYPE x, TYPE y, TYPE z);
```

Multiplies the current matrix by a matrix that stretches, shrinks, or reflects an object along the axes. Each x, y, and z coordinate of every point in the object is multiplied by the corresponding argument x, y, or z.

With the local coordinate system approach, the local coordinate axes are stretched, shrunk, or reflected by the x, y, and z factors, and the associated object is transformed with them.

Figure 7 shows the effect of `glScalef(2.0, -0.5, 1.0)`.

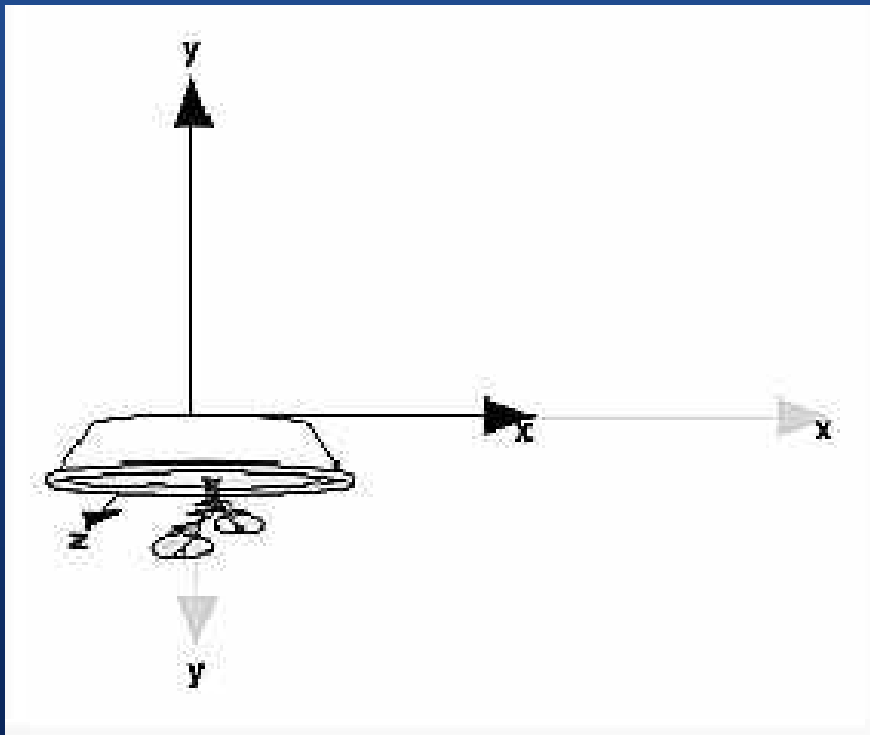


Figure 7 : Scaling and Reflecting an Object

`glScale*()` is the only one of the three modeling transformations that changes the apparent size of an object: Scaling with values greater than 1.0 stretches an object, and using values less than 1.0 shrinks it. Scaling with a -1.0 value reflects an object across an axis.

The identity values for scaling are (1.0, 1.0, 1.0). In general, we should limit `glScale*()` to cases where necessary. Using `glScale*()` decreases the performance of lighting calculations, because the normal vectors have to be renormalized after transformation.

Note: A scale value of zero collapses all object coordinates along that axis to zero. It's usually not a good idea to do this, because such an operation cannot be undone.

Sometimes collapsing co-ordinates does make sense, however, the calculation of shadows on a planar surface is a typical application. In general, if a coordinate system is to be collapsed, the projection matrix should be used.

## **A Modeling Transformation Code Example**

Example 2 is a portion of a program that renders a triangle four times, as shown in Figure 8. These are the four transformed triangles:

A solid wireframe triangle is drawn with no modeling transformation

The same triangle is drawn again, but with a dashed line stipple and translated

A triangle is drawn with a long dashed line stipple, with its height (y-axis) halved and its width (x-axis) increased by 50%

A rotated triangle, made of dotted lines, is drawn

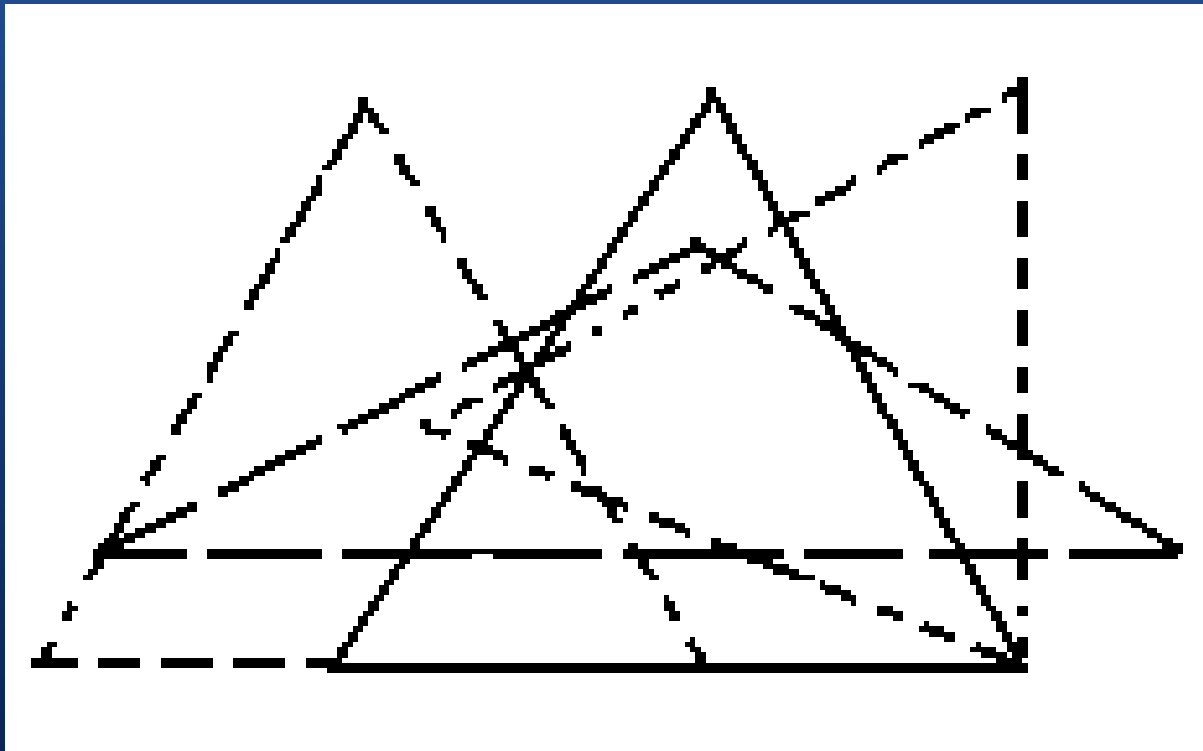


Figure 8 : Modeling Transformation Example

#### Example 2 : Using Modeling Transformations

```
glLoadIdentity();
glColor3f(1.0, 1.0, 1.0);
draw_triangle();
glEnable(GL_LINE_STIPPLE);
glLineStipple(1, 0xF0F0);
```

```
glLoadIdentity();
glTranslatef(-20.0, 0.0, 0.0);
draw_triangle();
glLineStipple(1, 0xF00F);
glLoadIdentity();
glScalef(1.5, 0.5, 1.0);
```

```
draw_triangle();
glLineStipple(1, 0x8888);
glLoadIdentity();
glRotatef (90.0, 0.0, 0.0, 1.0);
draw_triangle ();
glDisable (GL_LINE_STIPPLE);
```

Note the use of `glLoadIdentity()` to isolate the effects of modeling transformations; initializing the matrix values prevents successive transformations from having a cumulative effect.

Even though using `glLoadIdentity()` repeatedly has the desired effect, it may be inefficient, because we may have to re specify viewing or modeling transformations.



Note: Sometimes, programmers who want a continuously rotating object attempt to achieve this by repeatedly applying a rotation matrix that has small values.

The problem with this technique is that because of round-off errors, the product of thousands of tiny rotations gradually drifts away from the value we really want (it might even become something that isn't a rotation).

Instead of using this technique, increment the angle and issue a new rotation command with the new angle at each update step.

## Viewing Transformations

A viewing transformation changes the position and orientation of the viewpoint. If we recall the camera analogy, the viewing transformation positions the camera tripod, pointing the camera towards the model.

Just as we move the camera to some position and rotate it until it points in the desired direction, viewing transformations are generally composed of translations and rotations. To achieve a certain scene composition in the final image, we can either move the camera or move all the objects in the opposite direction.

Thus, a modeling transformation that rotates an object counterclockwise is equivalent to a viewing transformation that rotates the camera clockwise, for example.

Finally, the viewing transformation commands must be called before any modeling transformations are performed, so that the modeling transformations take effect on the objects first.

We can manufacture a viewing transformation in any of several ways, as described next. we can also choose to use the default location and orientation of the viewpoint, which is at the origin, looking down the negative z-axis:

Use one or more modeling transformation commands that is, `glTranslate*`() and `glRotate*`(). We can think of the effect of these transformations as moving the camera position or as moving all the objects in the world, relative to a stationary camera

Use the Utility Library routine `gluLookAt()` to define a line of sight. This routine encapsulates a series of rotation and translation commands

Create your own utility routine that encapsulates rotations and translations.

Some applications might require custom routines that allow to specify the viewing transformation in a convenient way. For example, we might want to specify the roll, pitch, and heading rotation angles of a plane in flight, or to specify a transformation for a camera that's orbiting around an object.

# Using `glTranslate*()` and `glRotate*()`

When we use modeling transformation commands to emulate viewing transformations, we're trying to move the viewpoint in a desired way while keeping the objects in the world stationary.

Since the viewpoint is initially located at the origin and since objects are often most easily constructed there as well (see Figure 9), in general we have to perform some transformation so that the objects can be viewed.

Note that, as shown in the figure, the camera initially points down the negative z-axis. (we're seeing the back of the camera.)



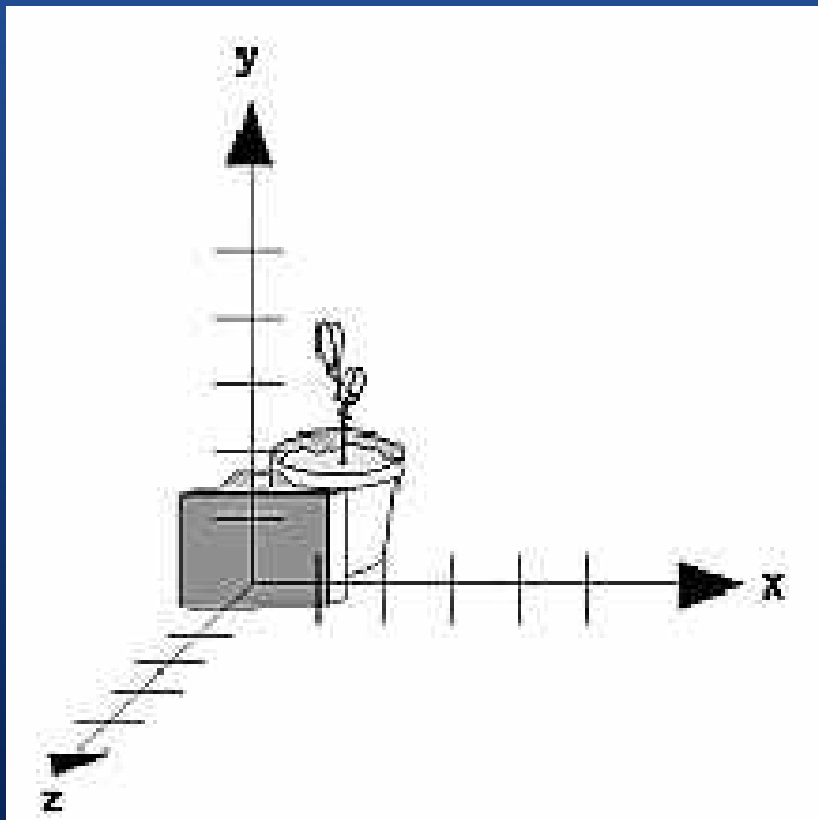


Figure 9 : Object and Viewpoint at the Origin

In the simplest case, we can move the viewpoint backward, away from the objects; this has the same effect as moving the objects forward, or away from the viewpoint.

Remember by default forward is down the negative z-axis; if we rotate the viewpoint, forward has a different meaning. So, to put 5 units of distance between the viewpoint and the objects by moving the viewpoint, as shown in Figure 10, use

```
glTranslatef(0.0, 0.0, -5.0);
```

This routine moves the objects in the scene -5 units along the z axis. This is also equivalent to moving the camera +5 units along the z axis.

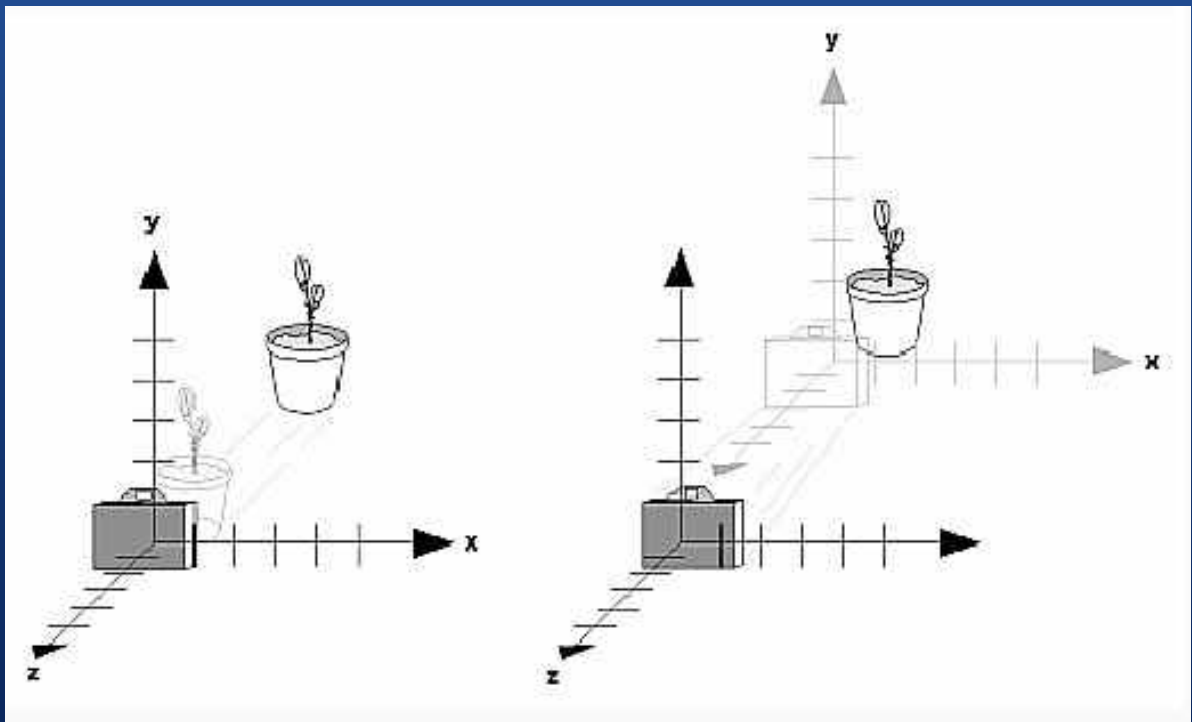


Figure 10 : Separating the Viewpoint and the Object

Now suppose we want to view the objects from the side. Should we issue a rotate command before or after the translate command? If we're thinking in terms of a grand, fixed coordinate system, first imagine both the object and the camera at the origin.

We could rotate the object first and then move it away from the camera so that the desired side is visible.

Since we know that with the fixed coordinate system approach, commands have to be issued in the opposite order in which they should take effect, we know that we need to write the translate command first in our code and follow it with the rotate command.

Now let's use the local coordinate system approach. In this case, think about moving the object and its local coordinate system away from the origin; then, the rotate command is carried out using the now-translated coordinate system.

With this approach, commands are issued in the order in which they're applied, so once again the translate command comes first. Thus, the sequence of transformation commands to produce the desired result is

```
glTranslatef(0.0, 0.0, -5.0);
glRotatef(90.0, 0.0, 1.0, 0.0);
```

If we're having trouble keeping track of the effect of successive matrix multiplications, try using both the fixed and local coordinate system approaches and see whether one makes more sense to us.

Note that with the fixed coordinate system, rotations always occur about the grand origin, whereas with the local coordinate system, rotations occur about the origin of the local system. We might also try using the `gluLookAt()` utility routine.

# **Computer Graphics**

## **Lecture 42**

# **Computer Graphics**

## **Lecture 45 Animation**

**Taqdees A. Siddiqi**  
**cs602@vu.edu.pk**

The passage of time – a fascination for artists, scientists and theologians

They have attributed to it different interpretations, implications & conclusions.

A general agreement on one aspect of time –

- we are all conditioned by it and
- there is a time space into which we have to fit



Einstein's theory of relativity maintains that space and time are merely different aspects of the same thing.

An inference by physicists - - objects can be moved backward and forward in space, but nothing can be moved back in time.

Another method of describing the concept of time is through the 'three arrows of time'.

Thermodynamic

Historical 'arrow'

Cosmological

## Film Time

- Vital concept to understand and to use in Animation.
- An essential raw material which can be compressed or expanded and
- used for effects and moods in a highly creative way.

## How time can be applied to animation?

- the animator can creatively manipulate time
- an action must be timed prior to carrying out the actual physical work on a film.

How the audience will react?

Time sense is just as important as colour sense and skill of drawing or craftsmanship in film animation.

A performance on stage and on the screen requires a basic understanding of how timing works.

However, we will consider hand drawn animation.

# Timing for TV series

- Animation is expensive
- TV series use simple minimum animation-Limited animation

plots are carried along by means of dialogue

prerecorded blocks of dialogue

The director has room to maneuver sections:

Available Time = Total Time – Dialogue Time

It can be split and distributed throughout the film

**Limited Animation**

24 frames per second required but:

No more than 6 drawings used for one second of animation.

Hold is lengthened

Animator skill required:

for Limited animation = for Full animation

# Full animation

Large number of drawings per second of action.

May use 24 drawings per second of animation

Time and money not spared on animation.

As a rule only TV-commercials and feature-length animated films can afford this luxury.

Animation is expensive and time-consuming.

It is not economically possible to animate more than is needed and edit the scenes later, as it is in live-action films.



In cartoons the director carefully pre-times every action so that the animator works within exact limits and does no more drawings than necessary.

Ideally, the director should be able to view line test loops of the film as it progresses and so have a chance to make adjustments. But often there is no time to make corrections in limited animation.

# Timing for Animation in General

Timing in animation is an elusive subject. It only exists whilst the film is being projected, in the same way that a melody only exists when it is being played. A melody is more easily appreciated by listening to it than by trying to explain it in words.

So with cartoon timing, it is difficult to avoid using a lot of words to explain what may seem fairly simple when seen on the screen.

Timing is also a dangerous factor.

The only real criterion for timing is: if it works effectively on the screen it is good, if it doesn't, it isn't.

# **Animation Principles**

**What is good timing?**

# GRAVITY

Forces working against gravity  
Arrangement of Jointed body and muscles  
Psychological reason or motivation for action  
Live vs. lifeless objects

## The Storyboard

A smooth visual flow is the major objective in any film, especially if it is an animated one. Good continuity depends on coordinating the action of the character, choreography, scene changes and camera movement.

The storyboard should serve as a blueprint for any film project

No production should proceed until a satisfactory storyboard is achieved

Creative & Technical issues must be addressed in advance

A rough guideline is approximately 100 storyboard sketches for each minute of film.

For technically complex films, it may double

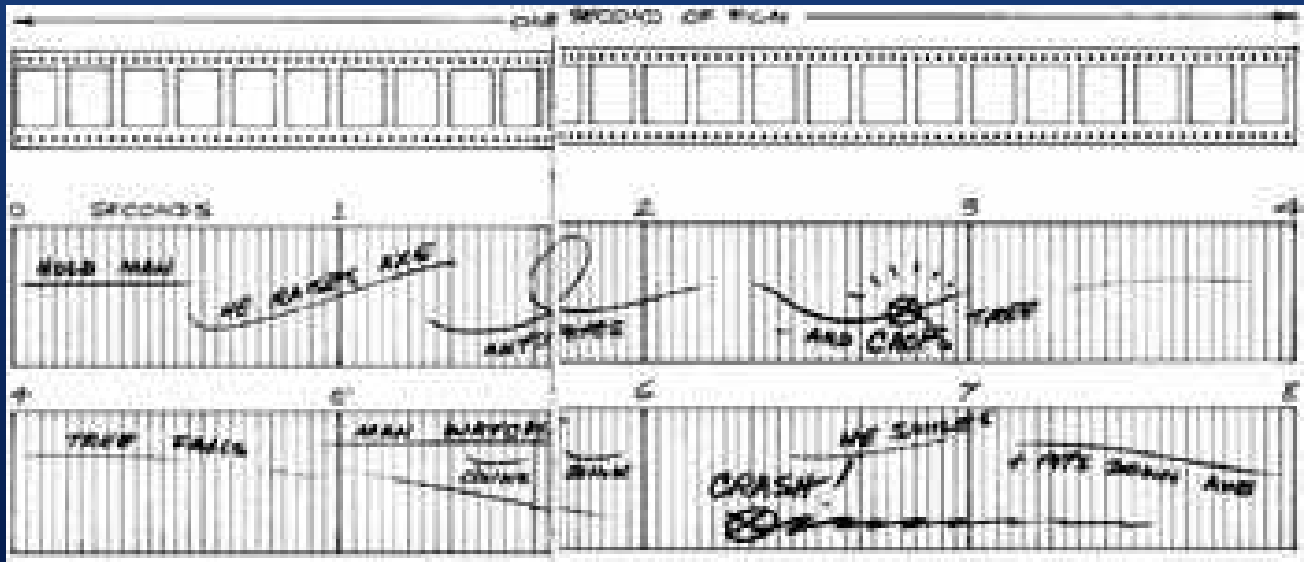
## **The Basic Unit of Time in Animation**

The basis of timing in animation is the fixed projection speed of 24 frames per second (fps) for film and video. On television and video this becomes 25 frames per second (PAL) or 30 fps (NTSC), but the difference is usually imperceptible.

If an action on the screen takes one second it covers 24 frames of film, and if it takes half a second it covers 12 frames. This fixed number of frames provides the basis on which all actions are planned and timed by the director.



# “The basic unit of time in animation”



For single frame animation, a second of action needs 24 drawings. If the same action is animated on double frames, 12 drawings are necessary the speed of the action would be the same in both cases.

The unit of time within which an animator works is, therefore,  $\frac{1}{24}$  sec,  $\frac{1}{25}$  sec or  $\frac{1}{30}$  sec and an important part of the skill, which the animator has to learn is what this specific timing 'feels' like on screen. With practice the animator also learns what multiples of this unit look like — 4 frames, 8 frames, 12 frames and so on.

**Animation and properties  
of  
matter**

The basic question which an animator is continually asking himself is: 'What will happen to this object when a force acts upon it?'.  
And the success of his animation largely depends on how well he answers this question.

Objects have their own weight, construction and degree of flexibility,  
different behaviour on application of force  
This behaviour - a combination of position and timing – is the basis of animation

Animation consists of drawings:

- no weight

- no forces acting

- are moving patterns

- have to be made to look live

Animators must consider Newton's laws of motion.

Not verbal definitions but actual realization of these laws in action.

Nothing moves automatically.

Nothing stops suddenly.

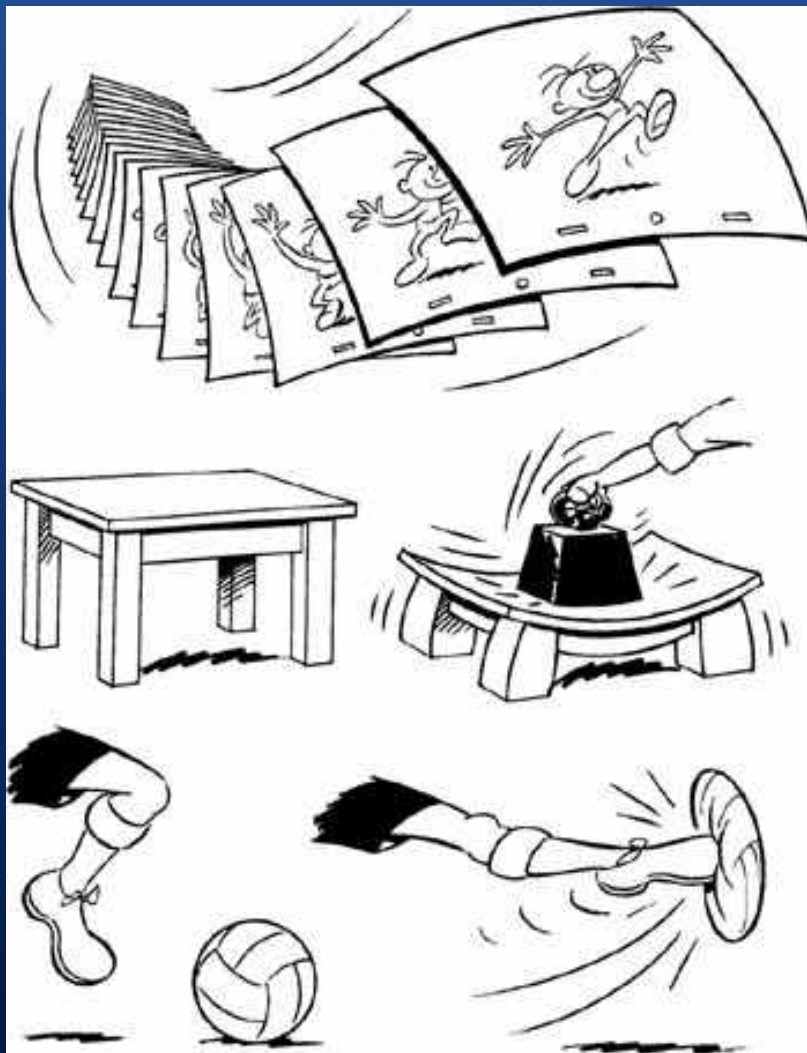
The timing of a scene for animation has two aspects:

- The timing of lifeless objects
- The timing of the movement of a living character

With lifeless objects the problems are straightforward dynamics. 'How long does a door take to slam?', 'How quickly does a cloud drift across the sky?'.

With living characters the same kind of problems occur because a character is a piece of flesh and a structure of jointed bones which has to be moved around by the action of forces on it.

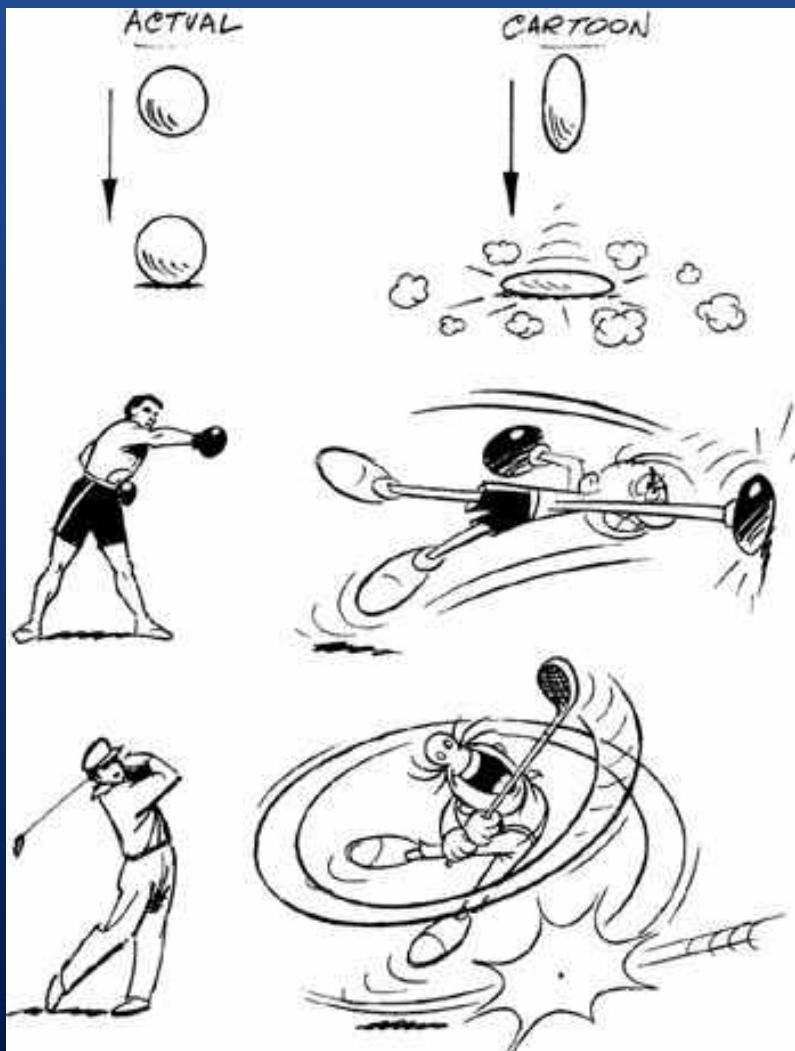
In addition, time must be allowed for the mental operation of the character. He must appear to be thinking his way through his actions, making decisions and finally moving his body around under the influence of his own will power and muscle.



## **“Animation and properties of matter”**

Animation consists of sequences of weightless drawings. The success of animation on the screen depends largely on how well these drawings give the impression of reacting in an exaggerated way when weight and forces are made to act upon them.

# Movement and Caricature



**“Movement  
and  
caricature”**



Cartoon is a medium of caricature.  
Naturalistic action looks weak in animation.  
Look at what actually happens,  
simplify down to the essentials of the movement  
and exaggerate these to the extreme.

## **Cause and Effect**

The strength of the forces pulling the rope

The flexibility or rigidity of the object being squeezed. Exaggerate the tendency

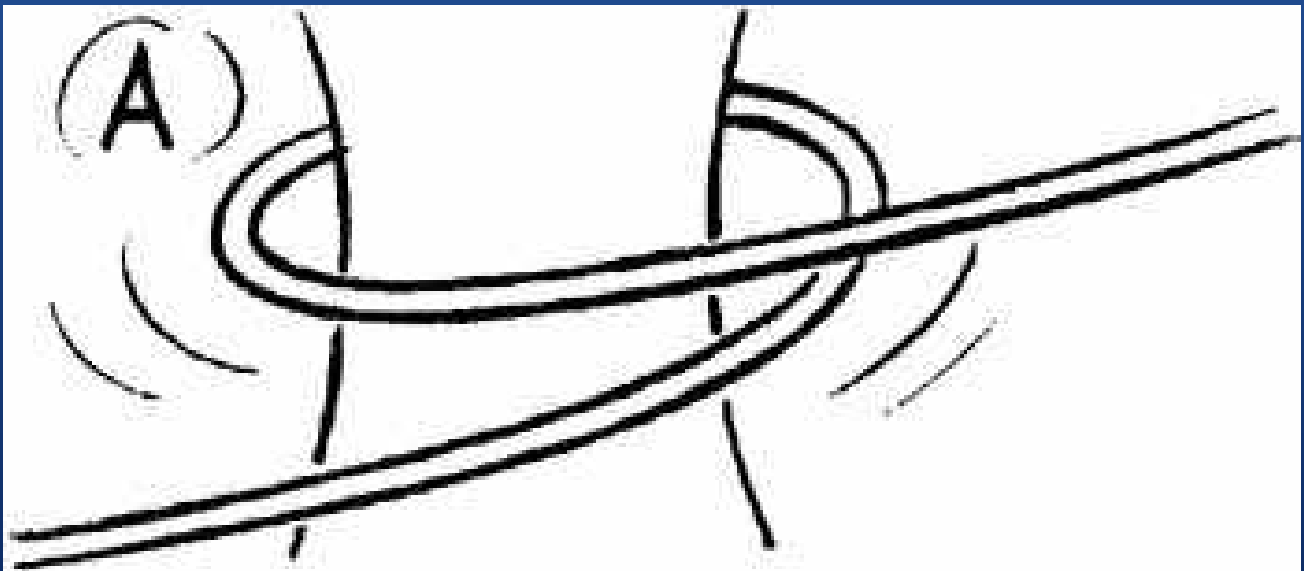


Fig A: A rope wrapped around something

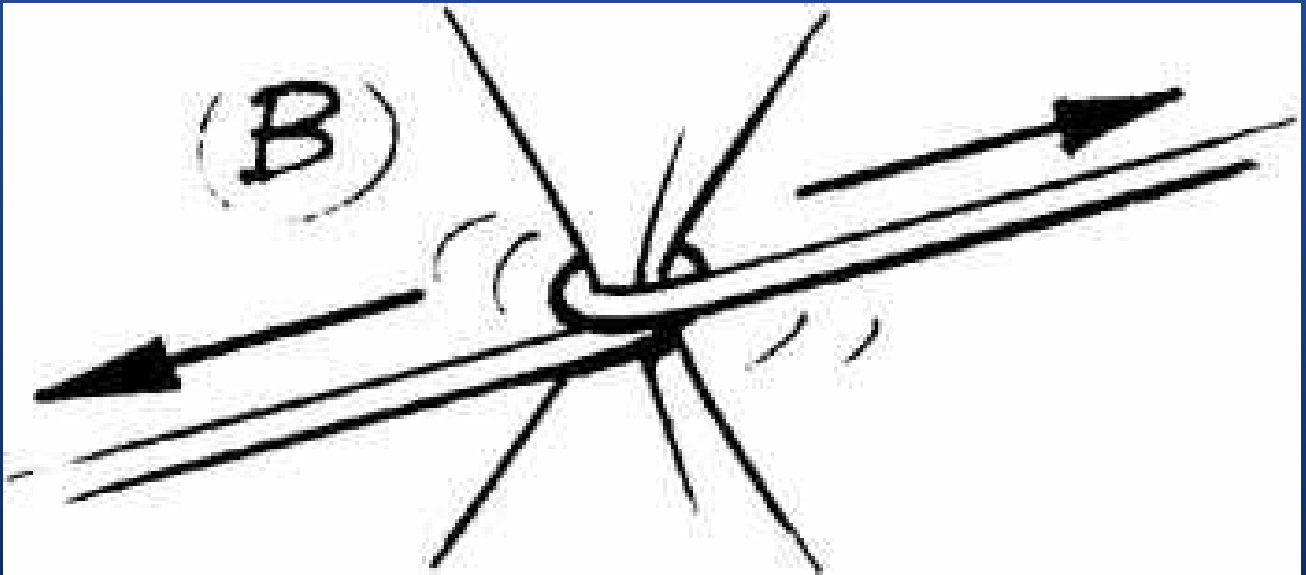


Fig B: The ends of the rope are pulled tight

## Looking back

History

Devices

Drawing Primitives

- Point
- Line
- Circle
- Ellipse

## Area Filling

- Polygon Filling
- Scanline Fill
- Boundary Fill
- Flood Fill
- 4-Connected vs. 8-Connected

## Mathematics

- Matrices
- Vectors
- Matrix Operations
- Vector Operations

## 2-D Transformations

- Translation
- Rotation
- Scaling
- Reflection
- Shear
- Homogeneous Coordinates

## Clipping

- Point
- Line
- Polygon
- Filled Polygon

3D Concepts

3D Transformations

- Translation
- Rotation
- Scaling
- Reflection
- Shear
- Homogeneous Coordinates

Projection

Parallel Projection

Perspective Projection

Triangles and Planes

Triangle Rasterization

Light

Color

Lighting and Shading

Types of Light

Lighting Equation

Lighting Models

Light

Color

Lighting and Shading

Types of Light

Directional

Positional

## Components of Light

- Ambient

- Diffuse

- Specular

- Back-Scattering

## OpenGL

- Introduction

- Architecture

- Advantages

- GLUT

- OpenGL Programming Examples



## Curves

- Plane Curves

- Space Curves

- Bezier Curves

## Surfaces and Subdivision

- Icosahedron

## Fractals

- OpenGL – Lighting

- OpenGL – Viewing

- OpenGL – Modeling

- OpenGL – Curves

- OpenGL – Surfaces

- OpenGL – Evaluators

Fractals

OpenGL – Lighting

OpenGL – Viewing

OpenGL – Modeling

OpenGL – Curves

OpenGL – Surfaces

OpenGL – Evaluators

# Computer Graphics

## Lecture 45

[www.vuCtn.com](http://www.vuCtn.com)

**Good Luck**

Regards : Zarva Chaudhary

**Admin:**

\* Zarva Chaudhary \*

\* Chaudhary Moazzam \*

\* Laiba Maki \*