Computer Graphics

CS602

Mid term Preparation

Virtual University of Pakistan

Knowledge beyond the boundaries

Table of Contents

Masters

Lecture No.1 Introduction to Computer Graphics

1.1 Definition

Computers accept process, transform and present information.

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

1.2 Why Study Computer Graphics?

There are certain important reasons to study computer graphics. We will discuss them under certain heads:

Visualization
I like to see what I am doing. Many a times it happens that you perform certain tasks which you cannot visualize; for example as a student of data structures, you implement trees, graphs and other Abstract Data Types (ADTs) but you cannot visualize them whereas you must be having an inner quest to see what these actually look like.

I like to show people what I am doing. Similarly at certain times you would be performing certain tasks which you know but it would be difficult for others to understand them so there is very important requirement of showing the things in order to make them understandable.

Graphics is interesting
We are visual creatures and for us a picture is worth a thousand words. If we can get rid of text based static screen and get some graphics; it's always interesting to see things in colours and motion on the screen. Therefore graphics is interesting because it involves simulation, algorithm, and architecture.

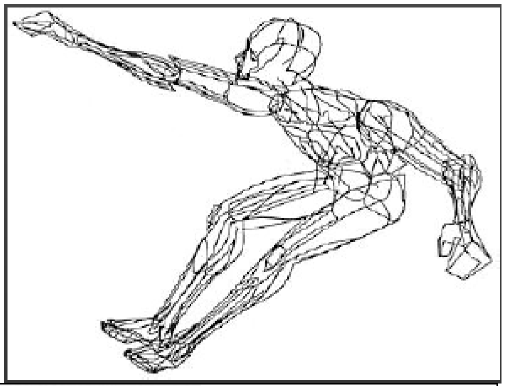
Requirement
Well there are certain areas which require use of computer graphics heavily. One example is drawing of machines. It is required to prepare drawing of a machine before the actual production. The other heavy requirement is for architects as they have to prepare a complete blue print of the building they have to build long before the actual construction work gets underway. AutoCAD and other applications of the kind are heavily used today for building architecture. MCQs

Entertainment
Merely a couple of decades back, the idea of a 24 hours Cartoons Network was really a far fetched one. That was the time when one would wait for a whole week long before getting an entertainment of mere 15 minutes. Well thanks to computer graphics that have enabled us to entertain ourselves with animated movies, cartoons etc.

MCQs

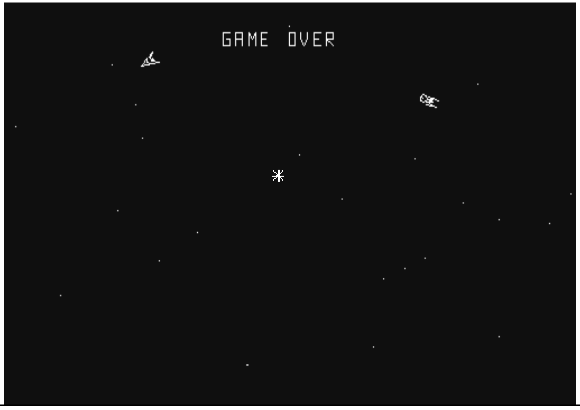
1.3 Some History

The term "computer graphics" was coined in 1960 by William Fetter to describe the 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.

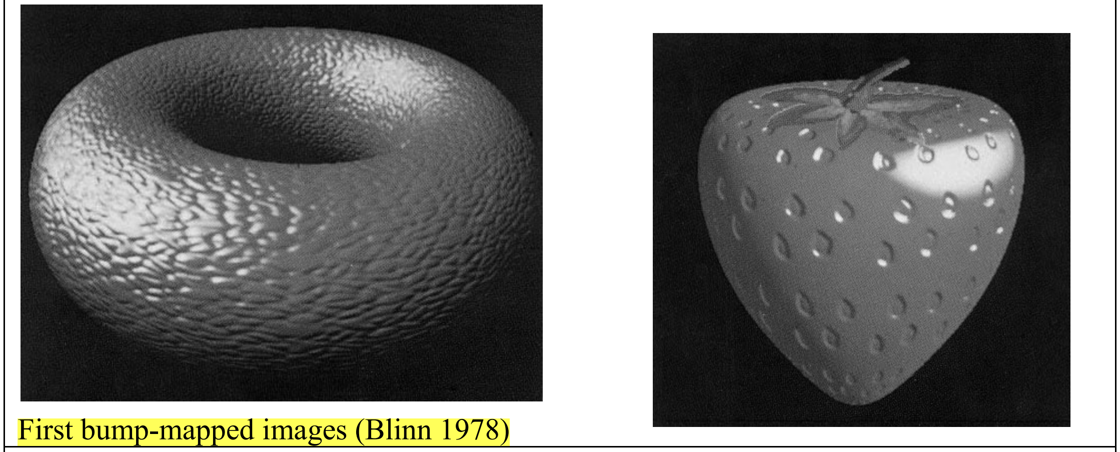


Whirlwind: early graphics using Vector Scope (1951)

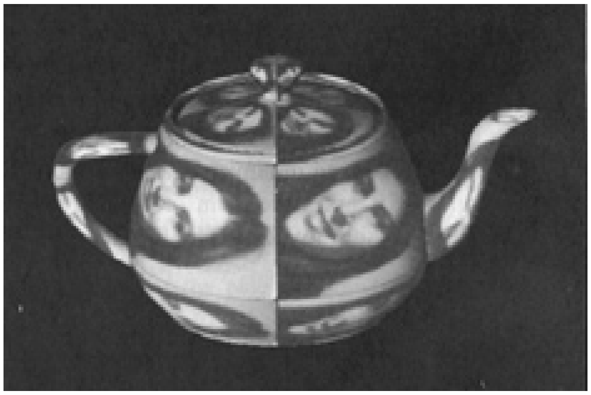
Spacewars: first computer graphics game (MIT 1961)



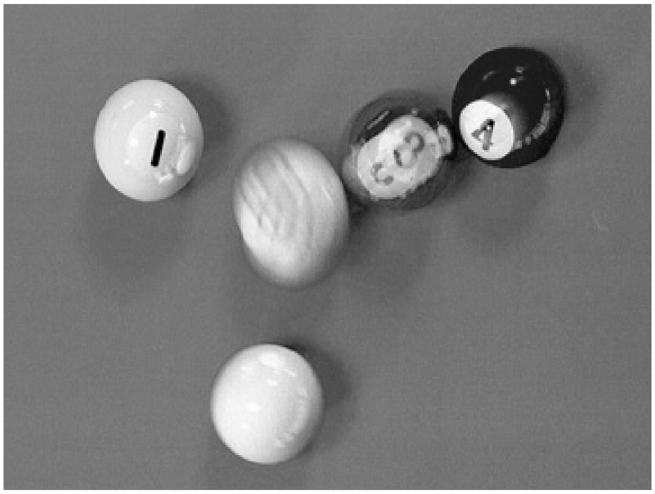
First CAD system (IBM 1959)



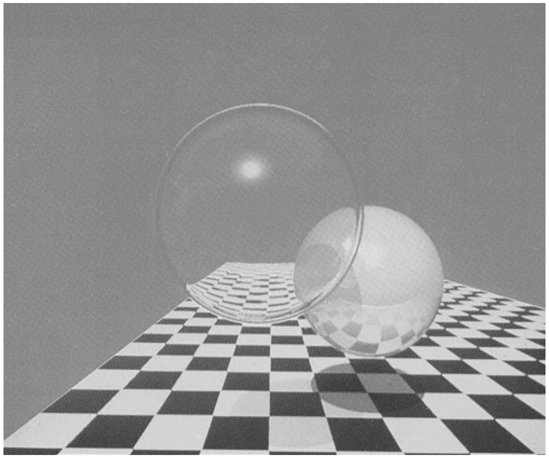
First bump-mapped images (Blinn 1978)



Early texture-mapped image (Catmull 1974)



First distributed ray traced image (Cook 1984)



First ray traced image (Whitted 1980)

1.4 Graphics Applications

Due to rapid growth in the field of computing, now computer is used as an economical and efficient tool for the production of pictures. Computer graphics applications are found in almost all areas. Here we will discuss some of the important areas including:

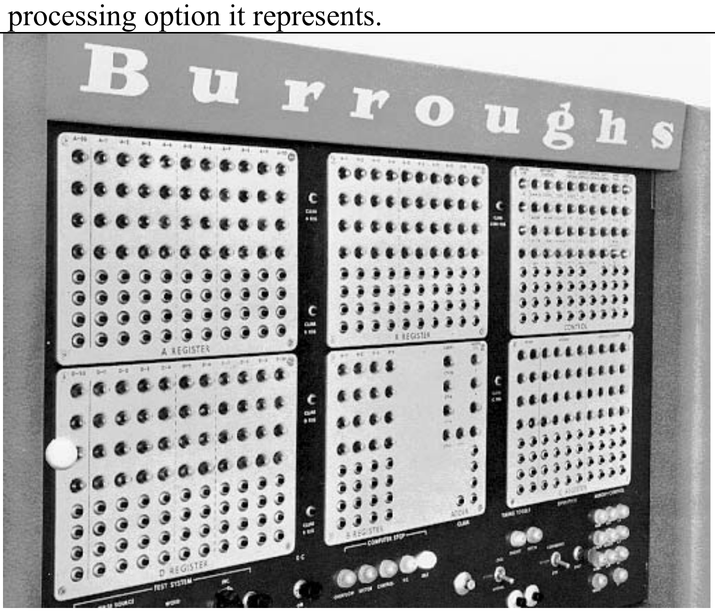
i. User Interfaces
ii. Layout and Design
iii. Scientific Visualization and Analysis
iv. Art and Design
v. Medicine and Virtual Surgery
vi. Layout Design & Architectural Simulations
vii. History and cultural heritage
viii. Entertainment
ix. Simulations
x. Games

User Interfaces

MCQs

Almost all the software packages provide a graphical interface. A major component of graphical interface is a window manager that allows a user to display multiple windows like areas on the screen at the same time. Each window can contain a different process that can contain graphical or non-graphical display. In order to make a particular window active, we simply have to click in that window using an interactive pointing device.

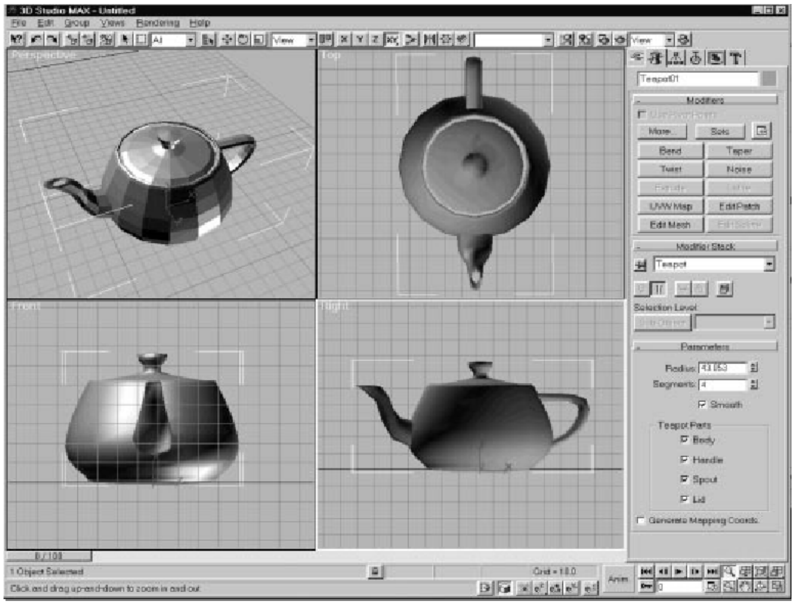
Graphical Interface also includes menus and icons for fast selection of programs, processing operations or parameter values. An icon is a graphical symbol that is designed to look like the processing option it represents.



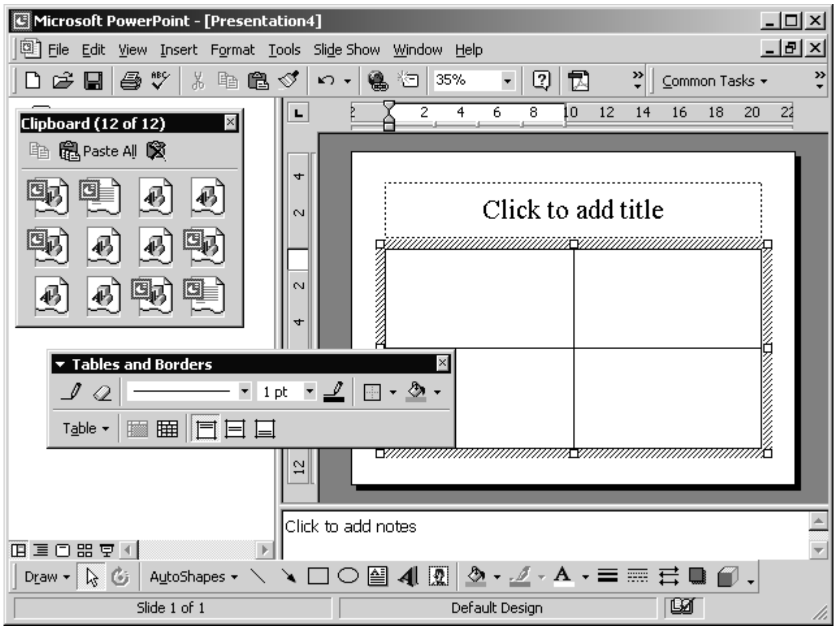
B205 Control Console (1960)

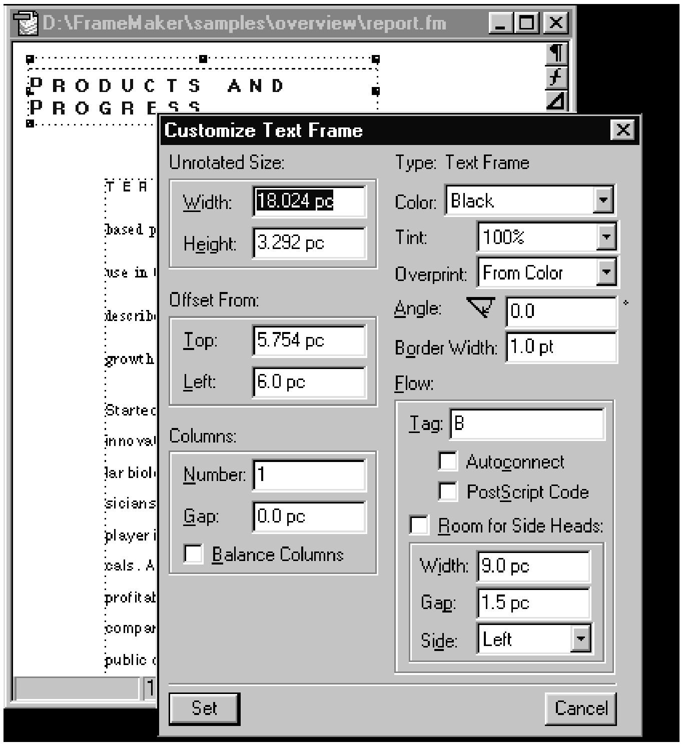
Impressive and Interactive 3D environment

3D Studio MAX



Layout and Design





Customize Text Frame

Unrotated Size:

Width: 18.024 pc

Height: 3.292 pc

Offset From:

Top: 5.754 pc

Left: 6.0 pc

Columns:

Number: 1

Gap: 0.0 pc

□ Balance Columns

Type: Text Frame

Color: Black

Tint: 100%

Overprint: From Color

Angle: 0.0 °

Border Width: 1.0 pt

Flow:

Tag: B

□ Autoconnect

□ PostScript Code

□ Room for Side Heads:

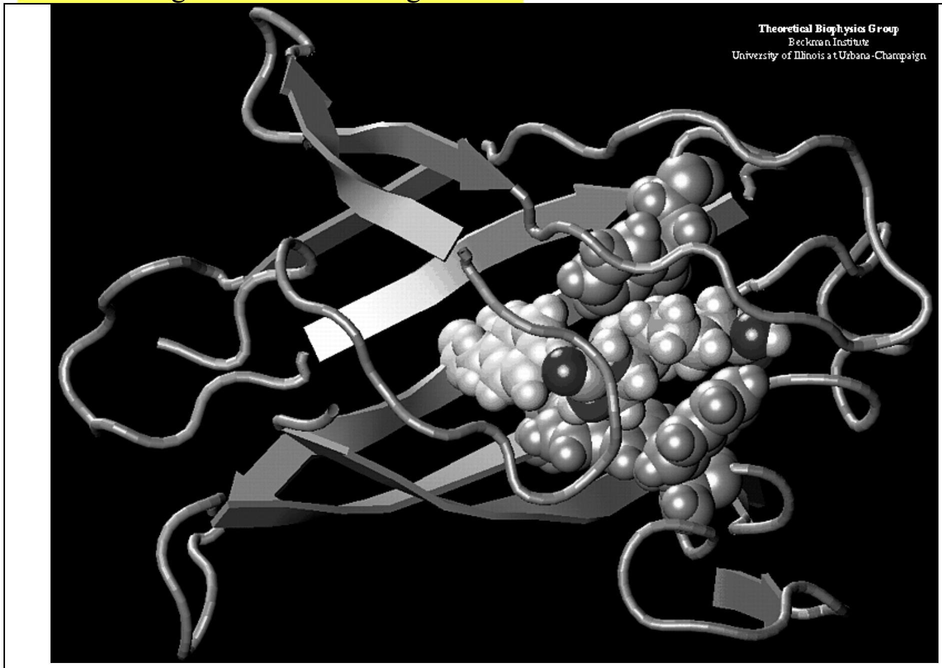
Width: 9.0 pc

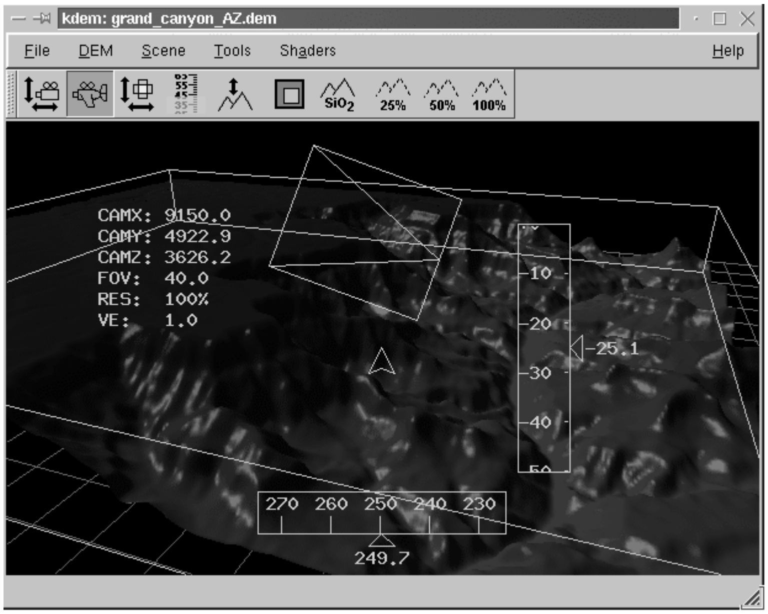
Gap: 1.5 pc

Side: Left

Scientific Visualization and Analysis

Computer graphics is very helpful in producing graphical representations for scientific visualization and analysis especially in the field of engineering and medicine. It helps a lot in drawing charts and creating models.


Theoretical Biophysics Group
Beckman Institute
University of Illinois at Urbana-Champaign



ART AND DESIGN

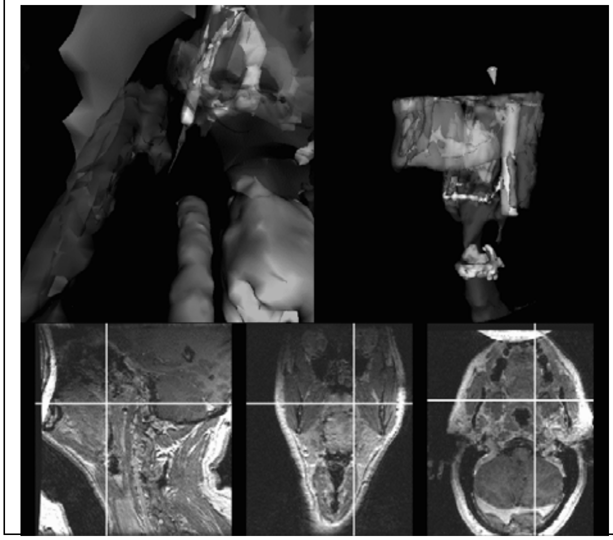
Computer graphics is widely used in Fine Arts as well as commercial arts for producing better as well as cost effective pictures. Artists use a variety of programs in their work, provided by computer graphics. Some of the most frequently used packages include:

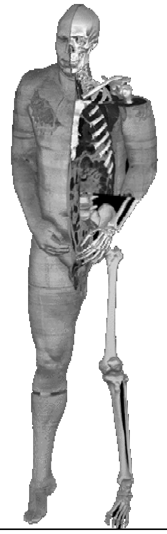
Artist's paintbrush
Pixel paint
Super paint

Medicine and Virtual Surgery

Computer graphics has extensive use in tomography and simulations of operations. Tomography is the technique that allows cross-sectional views of physiological systems in X-rays photography. Moreover, recent advancement is to make model and study physical functions to design artificial limbs and even plan and practice surgery.

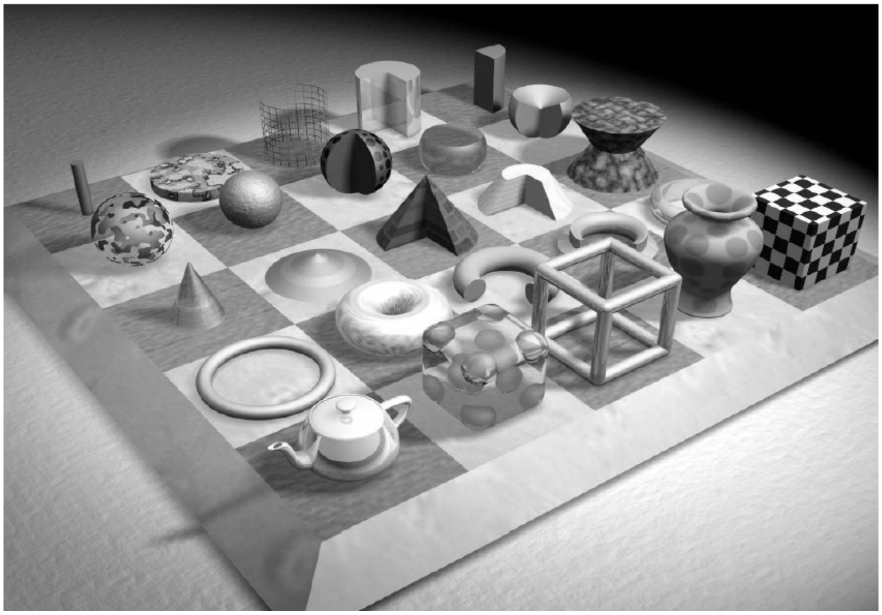
Computer-aided surgery is currently a hot topic.





Room Layout Design and Architectural Simulations

Layout Design & Architectural Simulations



Today: realistic image synthesis
How much light leaving the luminaire reaches the viewer? If there is only one reflection this is the LOCAL ILUMINATION problem.
Review: phong shading
N
light
if angles a and b are the same we will see a highlight

History and cultural heritage

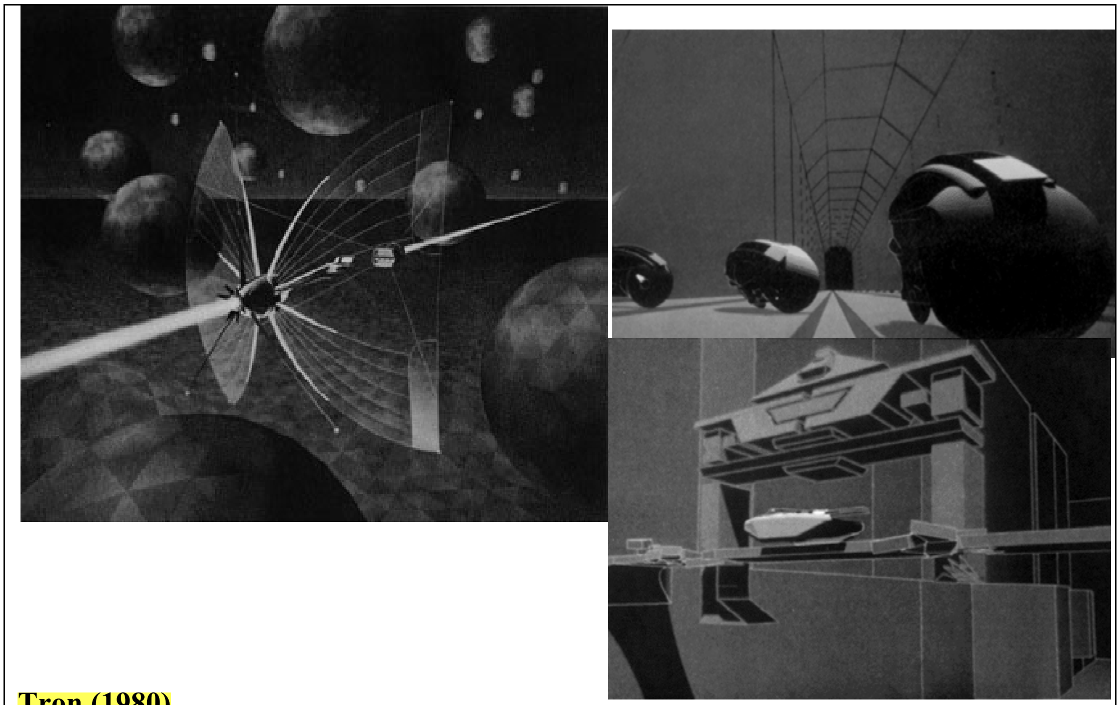
Another important application of computer graphics is in the field of history and cultural heritage. A lot of work is done in this area to preserve history and cultural heritage. The features so for provide are:

- **Innovative graphics presentations developed for cultural heritage applications**
- **Interactive computer techniques for education in art history and archeology**
- **New analytical tools designed for art historians**
- **Computer simulations of different classes of artistic media**

Movies

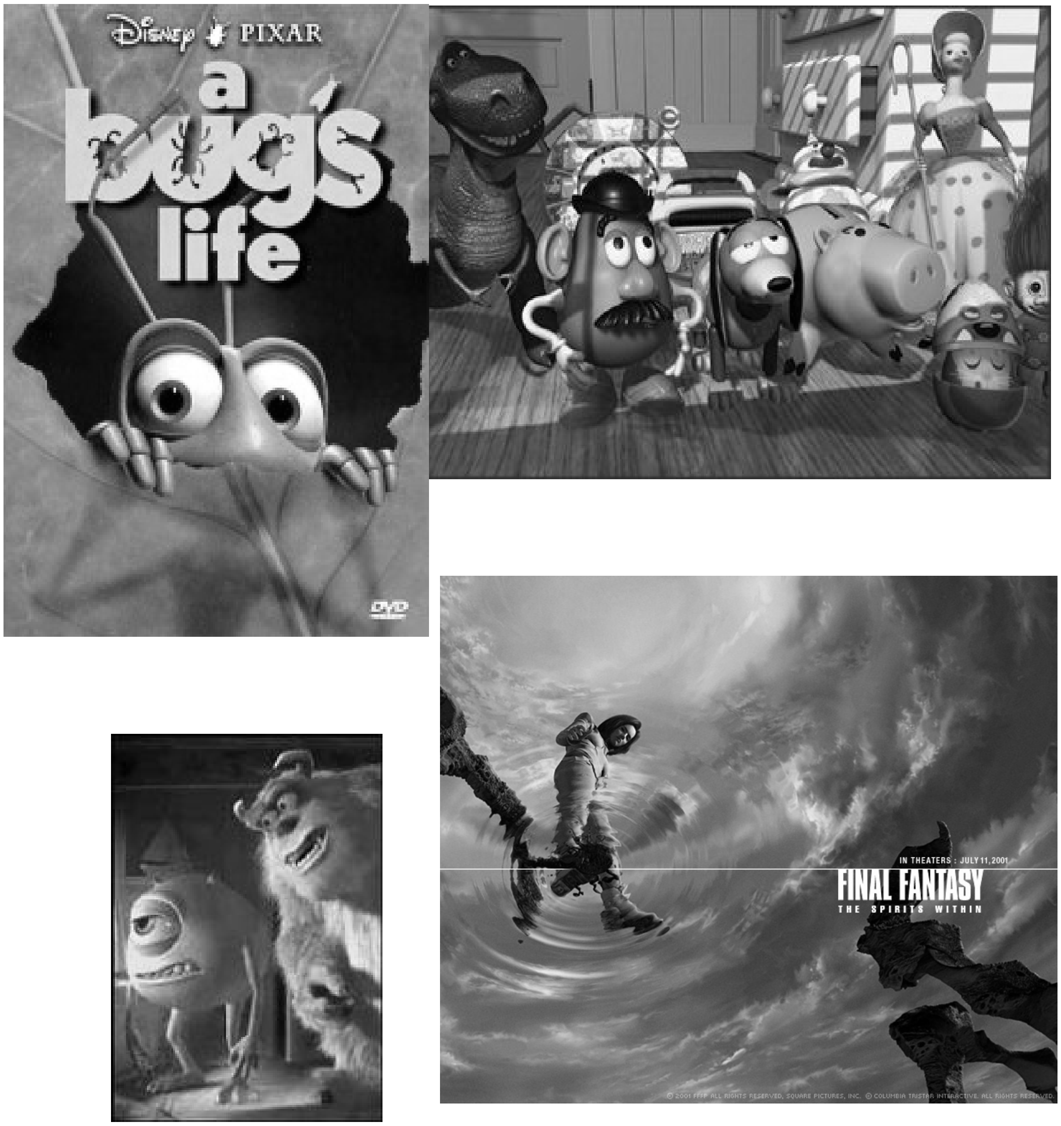
Computer graphics methods are now commonly used in making motions pictures, music videos and television shows. Sometimes the graphics scenes are displayed by themselves and sometimes graphics objects are combined with the actors and live scenes. A number of hit movies and shows are made using computer graphics technology. Some of them are:

Star Trek- The Wrath of Khan
Deep Space Nine
Stay Tuned
Reds Dreams
She's Mad

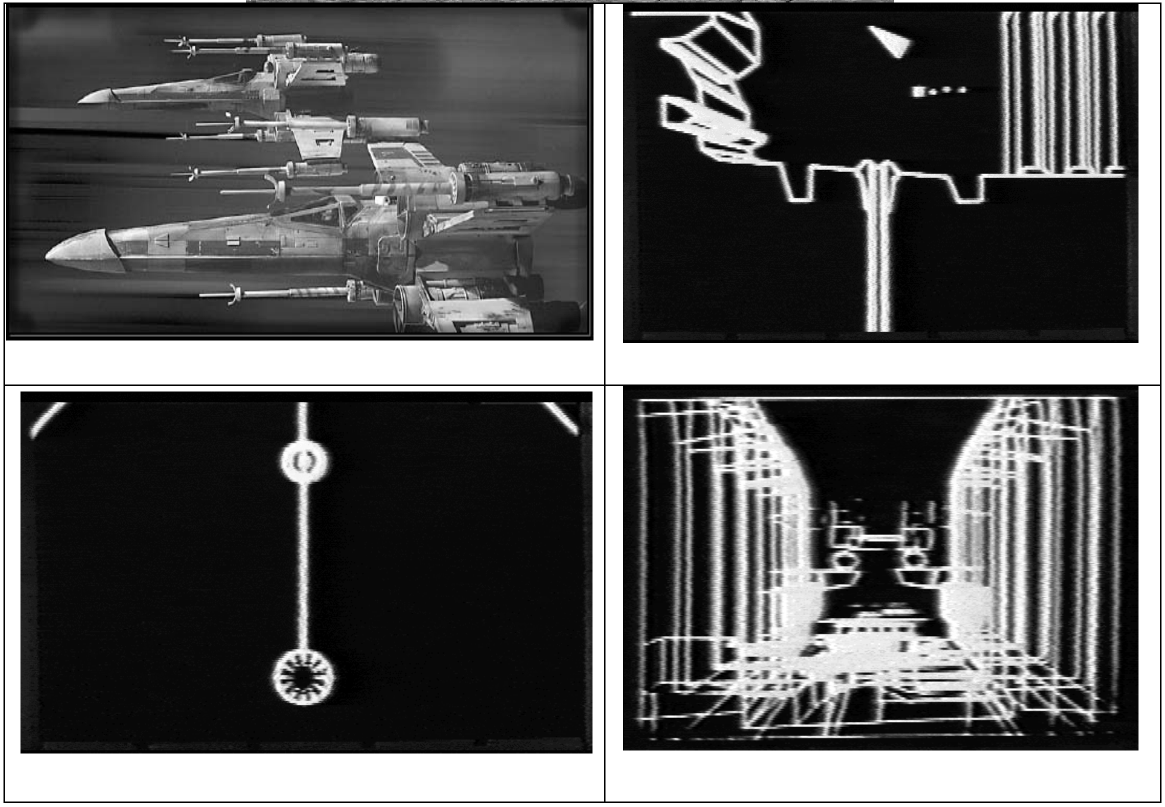


Tron (1980)

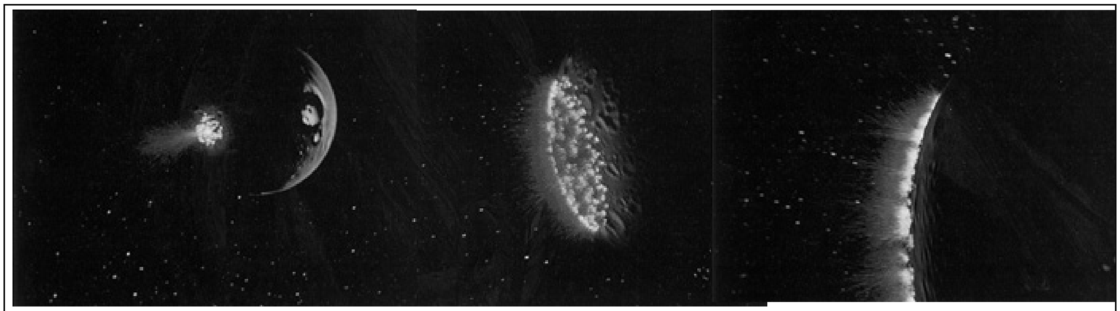
First time computer graphics were used for live action sequences.


Disney PIXAR
a bug's life
DVD
IN THEATERS : JULY 11,2001
FINAL FANTASY
THE SPIRITS WITHIN
© 2001 FFFP ALL RIGHTS RESERVED, SQUARE PICTURES, INC. © COLUMBIA TRISTAR INTERACTIVE. ALL RIGHTS RESERVED.

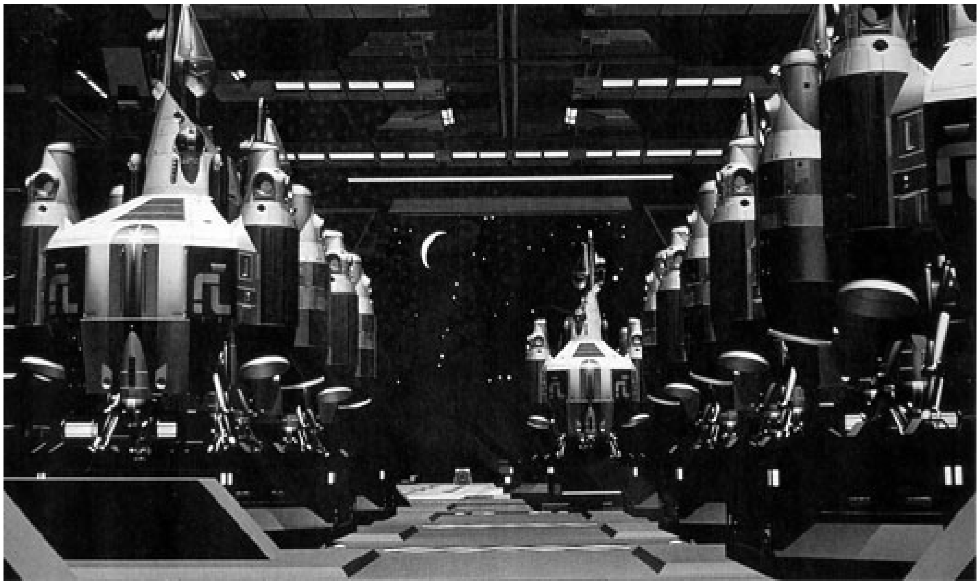
Fully computer generated animated features



Star Wars (1977)

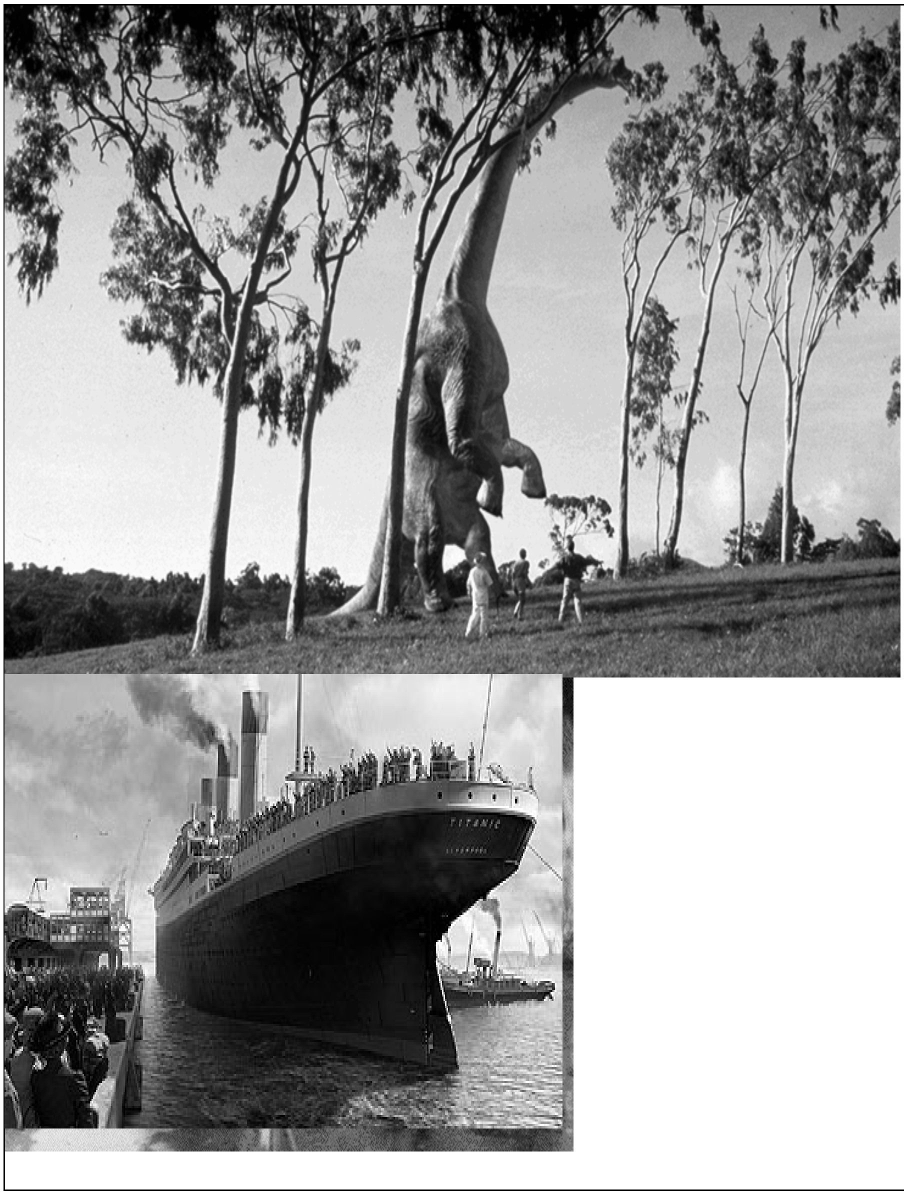


Star Trek II: The Wrath of Khan, genesis



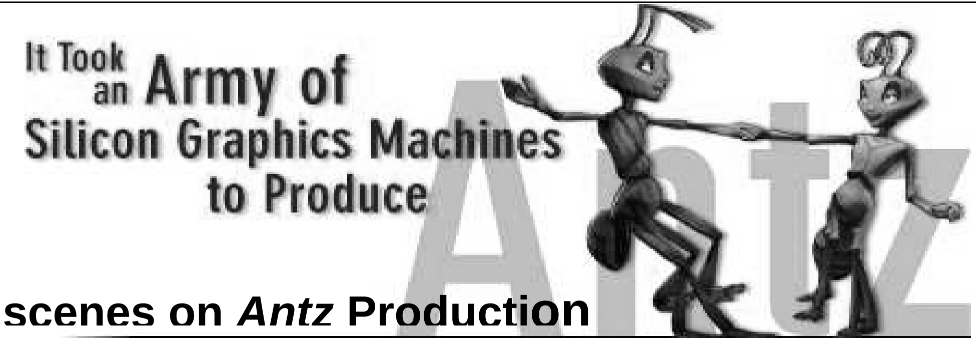
The Last Starfighter (15 minutes) (1982)

The Last Starfighter (15 minutes) (1982)



Special Effects… in Live Action Cinema

"Traditional" Animated Features… **Some examples:** **• Automating Keyframing in many Disney-type animations** **• The flocking behaviour of the wild beast in Lion King** **•Non photorealistic rendering: 3D effects in Futurama**	



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**
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**
Time it would have taken to render this movie on 1 processor	**54 yrs., 222 days, 15 mins., 36**
Amount of storage required for the movie	**3.2 TB**
Amount of frames kept online at any given time	**75000 frames**
Time to re-film out final cut beginning to end	**41.5 days (997 hrs.)**

Simulations

Simulation by all means is a very helpful tool to show the idea you have or the work you are doing or to see the results of your work. Given below is the picture in which you can see wave's ripples on water; no doubt looking like original but is simply a simulation. A number of software packages are used for simulation including:

Crackerjack Computer Skills
Keen Artistic Eye
Flash
Maya

© Arete Image Software 1996

© Arete Image Software 1996

Game

Thanks to computer graphics, real time games are now possible. Now game programming itself has become an independent field and game programmers are in high demand. Some of the famous games are:

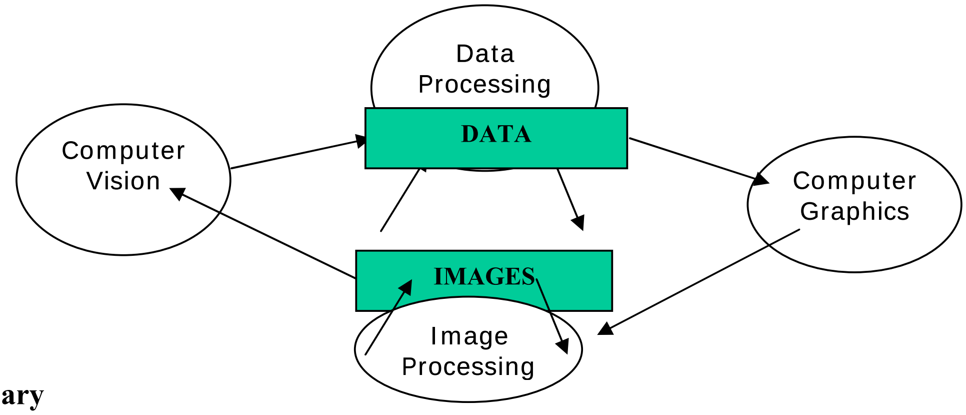
- Quake
- Dooms
- Need For Speed
- Commandos

- Entertainment: Games

id: Quake II

Cyan: Riven

Related Disciplines



Interdisciplinary

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

Lecture No.2 Graphics Systems I

Introduction of Graphics Systems

With the massive development in the field of computer graphics a broad range of graphics hardware and software systems is available. Graphics capabilities for both two-dimensional and three-dimensional applications are now common on general-purpose computers, including many hand-held calculators. On personal computers there is usage of a variety of interactive input devices and graphics software packages; whereas, for higher-quality applications some special-purpose graphics hardware systems and technologies are employed.

VIDEO DISPLAY DEVICES

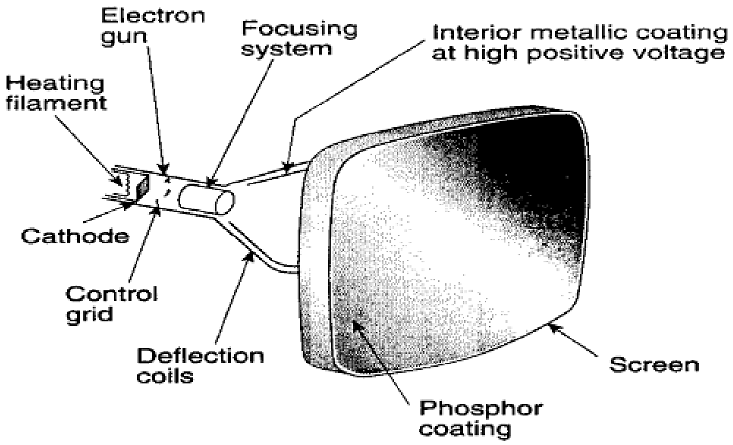
The primary output device in a graphics system is a video monitor. The operation of most video monitors is based on the standard cathode-ray-tube (CRT) design, but several other technologies exist and solid-state monitors may eventually predominate.

Refresh Cathode-Ray Tubes

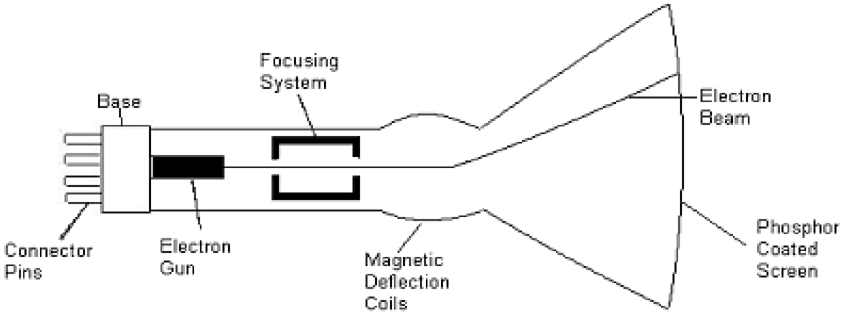
Following figures illustrate the basic operation of a CRT. A **beam** of electrons (cathode rays) emitted by an **electron gun,** passes through **focusing and deflection systems** that direct the beam toward specified positions on the **phosphor-coated screen.** The phosphor then emits a small spot of light at each position contacted by the electron beam.

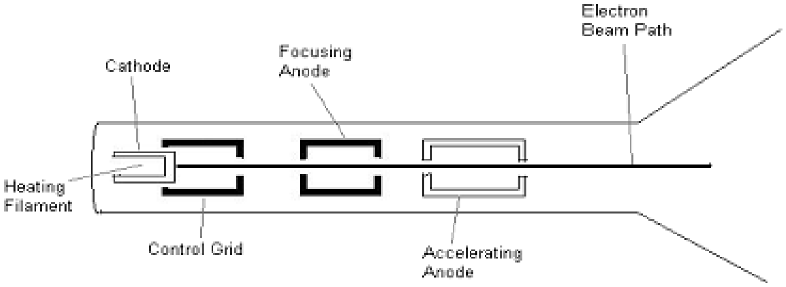
The light emitted by the phosphor fades very rapidly therefore to keep the picture it is necessary to keep the phosphor glowing. This is achieved through **redrawing** the picture repeatedly by quickly directing the electron beam back over the same points and the display using this technique is called **refresh CRT**.

The primary components of an electron gun in a CRT are the **heated metal cathode** and a **control grid**. **Heat** is supplied to the cathode by directing a current through **filament** (a coil of wire), inside the cylindrical cathode structure. Heating causes electrons to be boiled off the hot cathode surface. In the vacuum inside the CRT envelope, the free, negatively charged electrons are then **accelerated** toward the phosphor coating by a high positive voltage.



The **accelerating voltage** can be generated with a **positively charged metal** coating on the inside of the **CRT envelope** near the phosphor screen an **accelerating anode** can be used.





MCQs

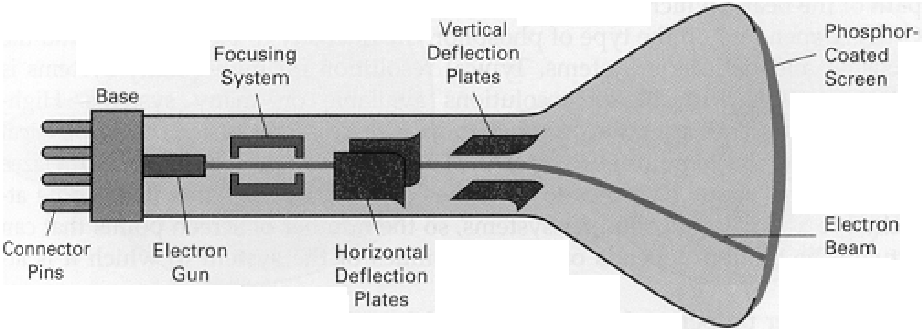
Intensity of the electron beam is controlled by setting voltage levels on the control grid, a metal cylinder that fits over the cathode. A high negative voltage applied to the control grid will shut off the beam by repelling electrons and stopping them from passing through the small hole at the end of the control grid structure. A smaller negative voltage on the control grid simply decreases the number of electrons striking the phosphor coating on the screen.

It is the responsibility of **focusing system** to converge electron beam to a small spot where it strikes the phosphor. Otherwise the electrons will repel each other and the beam would disperse. This focusing is achieved through **electric or magnetic fields**.

In **electrostatic focusing** the electron beam passes through a positively charged metal cylinder that forms an electrostatic lens. Then electrostatic lens focuses the electron beam at the center of the screen. Similar task can be achieved with a **magnetic field** setup by a coil mounted around the outside of the CRT envelope. Magnetic lens focusing produces the smallest spot size on the screen and is used in special purpose devices.

The distance that the electron beam must travel from gun to the exact location of the screen that is small spot is different from the distance to the center of the screen in most CRTs because of the curvature therefore some **additional focusing hardware** is required in high precision systems to take beam to all positions of the screen. This procedure is achieved in two steps in first step beam is conveyed through the exact center of the screen and then additional focusing system adjust the focusing according to the screen position of the beam.

Cathode-ray tubes are now commonly **constructed** with magnetic deflection coils mounted on the outside of the CRT envelope. Two pairs of coils are used, with the coils in each pair mounted on opposite sides of the neck of the CRT envelope. One pair is mounted on the top and bottom of the neck and the other pair is mounted on opposite sides of the neck. The magnetic field produced by each pair of coils results in a traverse deflection force that is perpendicular both to the direction of the magnetic field and to the direction of travel of the electron beam. **Horizontal deflection** is achieved with one pair of coils, and **vertical deflection** by the other pair. The proper deflection amounts are attained by adjusting the current through the coils. When electrostatic deflection is used, two pairs of parallel plates are mounted inside the CRT envelope. One pair of plates is mounted horizontally to control the vertical deflection, and the other pair is mounted vertical to control horizontal deflection.



MCQs **Phosphor** is available in different kinds. One variety is available in color but a major issue is their **persistence**. Persistence is defined as the time it takes the emitted light from the phosphor to decay to one-tenth of its original intensity. Lower persistence phosphors require higher refresh rates to maintain a picture on the screen without flicker. A phosphor with low persistence is useful for displaying highly complex, static pictures. Monitors normally come with persistence in the range from 10 to 60 microseconds.

The maximum number of points (that can be uniquely identified) on a CRT is referred to as the **resolution**. A more precise definition of resolution is the number of points per centimeter that can be plotted horizontally and vertically, although it is often simply stated as the total number of points in each direction.

Finally **aspect ratio**; is the ratio of vertical points to horizontal points necessary to produce equal-length lines in both directions on the screen. An aspect ratio of 3/4 means that a vertical line plotted with three points has the same length as a horizontal line plotted with four points. MCQs

RASTER-SCAN SYSTEMS

Raster scan is the most common type of monitors using CRT. In raster scan picture is stored in the area called **refresh buffer or frame buffer**. First of all why information is stored; because picture have to be refreshed again and again for this very reason it is stored. Second is how it is stored; so picture is stored in a two dimensional matrix where each element corresponds to each **pixel** on the screen. If there arise a question what is a pixel? The very simple answer is a pixel (short for picture element) represents the shortest possible unique position/ element that can be displayed on the monitor without overlapping.

The frame buffer stores information in a two dimensional matrix; the question is that how many bits are required for each pixel or element. If there is black and white picture then there is only one bit required to store '0' for black or 1 for white and in this case buffer will be referred as **bitmap**. In colour pictures obviously multiple bits are required for each pixel position depending on the possible number of colours for example to show 256 colours 8 bits will be required for each pixel and in case if multiple bits are used for one pixel frame buffer will be referred as **pixmap**.

Now with the information in frame buffer, let us see how an image is drawn. The drawing is done in a line-by-line fashion. After drawing each line from left to right it reaches at the left end of the next line to draw next line; which is called **horizontal retrace**. Similarly after completing all lines in horizontal fashion it again reaches the top left corner to start redrawing the image (that is for refreshing) and this is called **vertical retrace**. Normally each vertical retrace takes 1/60th of a second to avoid flickering.

There are two further methods to scan the image: **interlaced** and **non-interlaced**. In interlaced display beam completes scanning in two passes. In one pass only odd lines are drawn and in the second pass even lines are drawn. Interlacing provides effect of double refresh rate by completing half of the lines in half of the time. Therefore, in systems with low refresh rates interlacing helps avoid flickering.

RANDOM-SCAN Displays

In random-scan displays a portion of the screen can be displayed. Random-scan displays draw a picture one line at a time and are also called vector displays (or stroke-writing or calligraphic displays). In these systems image consists of a set of line drawing commands referred to as **Refresh Display File**. Random-scan can refresh the screen in any fashion by repeating line drawing mechanism.

Random-scan displays are designed to draw all the component lines of a picture 30 to 60 times each second. High-quality vector systems are capable of handling approximately 100,000 short lines at this refresh rate. When a small set of lines is to be displayed, each refresh cycle is delayed to avoid refresh rates greater than 60 frames per second. Otherwise, faster refreshing of the set of lines could burn out the phosphor.

Random-scan displays are designed for line-drawing applications and cannot display complex pictures. The lines drawn in vector displays are smoother whereas in raster-scan lines often become jagged.

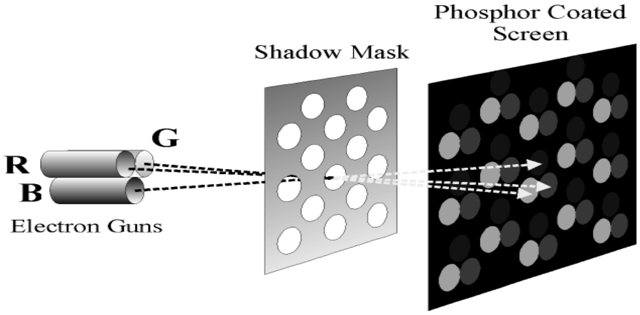
Color CRT Monitors

A CRT monitor displays colour pictures by using a combination of phosphors that emit different coloured light. With the combination of phosphor a range of colours can be displayed. There are two techniques used in colour CRT monitors:

- Beam Penetration Method
- Shadow Mask Method

In **beam penetration** method two layers of phosphor, usually coated onto the inside of the CRT screen, and the displayed colour depend on how far the electron beam penetrates into the phosphor layers. At intermediate beam speeds, combinations of red and green light are emitted to show two additional colours, orange and yellow. Beam penetration is an inexpensive way to produce colours as only a few colours are possible and the quality of picture is also not impressive.

Important



Shadow mask methods can display a wide range of colours. In this technique each pixel position is made up of three phosphor dots called triads as shown in the following figure. Three phosphor dots have different colors i.e. red, green and blue and the display colour is made by the combination of all three dots. Three guns are used to throw beam at the three dots of the same pixel. By varying intensity at each dot a wide range of colours can be generated.

A **shadow-mask** is used which has holes aligned with the dots so that each gun can fire beam to corresponding dot only.

Important

CRT Displays

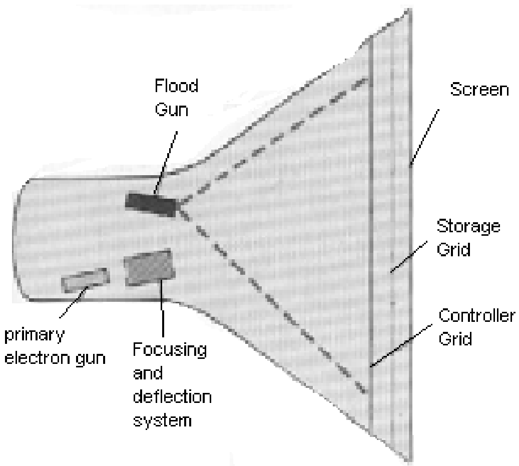
Advantages

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



A direct view storage tube stores the picture information as a charge distribution just behind the phosphor-coated screen. Two electron guns are used in this system as shown in the following figure. They are:

- Primary Gun
- Flood Gun

Primary gun is used to store the picture pattern whereas **flood gun** maintains the picture display.

DVST has advantage that no refresh is required so very complex pictures can be displayed at very high resolutions without flicker. Whereas, it has disadvantage that ordinarily no colors can be displayed and that selected parts of a picture cannot be erased. To eliminate a picture section, the entire screen must be erased and the modified picture redrawn. The erasing and redrawing process can take several seconds for a complex picture.

Flat-Panel Displays

This is emerging technology slowly replacing CRT monitors. The flat-panel displays have following properties:

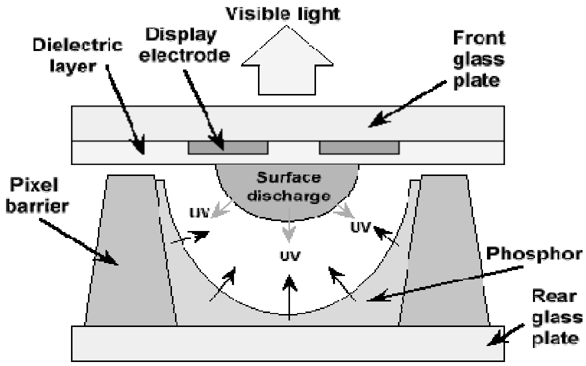
- Little Volume
- Light Weight
- Lesser Power consumption

Flat panels are used in calculators, pocket video games and laptop computers.

There are two categories of flat panel displays:

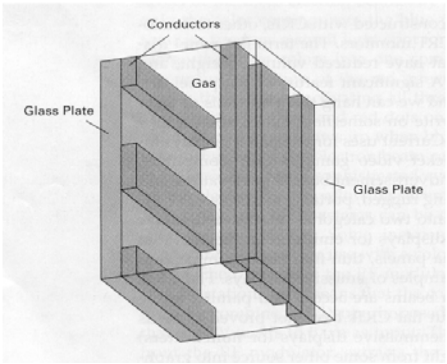
- Emissive Display (Plasma Panels)
- Non-Emissive Display (Liquid Crystal Display)

The **emissive displays** (emitters) are devices that convert electrical energy into light. Plasma panels, thin-film electro-luminescent displays, and light-emitting diodes are examples of emissive displays. **Non-emissive** displays (non-emitters) use optical effects to convert sunlight or light from some other source into graphics patterns. The most important example of a non-emissive flat-panel display is a liquid-crystal device.



Plasma-panel Displays

Plasma panels also called gas-discharge displays are constructed by filling the region between two glass plates with a mixture of gases that usually includes neon. A series of vertical conducting ribbons is placed on one glass panel, and a set of horizontal ribbons is built into the other glass panel. Firing voltages applied to a pair of horizontal and vertical conductors cause the gas at the intersection of the two conductors to break down into glowing plasma of electrons and ions. Picture definition is stored in a refresh buffer, and the firing voltages are applied to refresh the pixel positions 60 times per second.



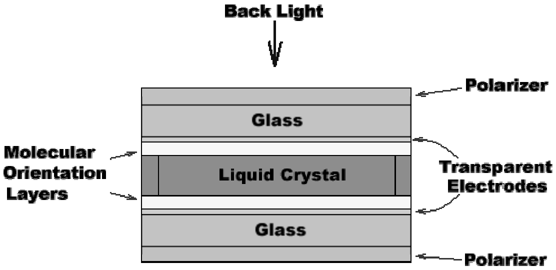
Advantages Important

–Large viewing angle

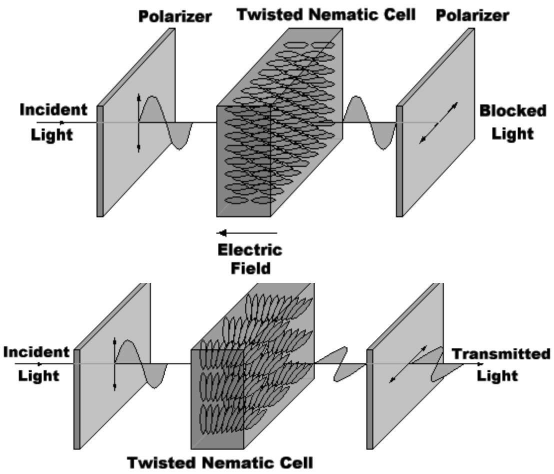
–Good for large-format displays
–Fairly bright
Disadvantages
–Expensive
–Large pixels (~1 mm versus ~0.2 mm)
–Phosphors gradually deplete
–Less bright as compared to CRTs, using more power

Liquid Crystal Displays

Liquid crystal refers to the fact that these compounds have a crystalline arrangement of molecules, yet they flow like a liquid. Flat panel displays use nematic liquid crystal, as demonstrated in the following figures.



Two glass plates, each containing a light polarizer at right angles to the other plate, sandwich the liquid-crystal material. Rows of horizontal transparent conductors are built into one glass plate, and columns of vertical conductors are put into the other plate. The intersection of two conductors defines a pixel position. Polarized light passing through the material is twisted so that it will pass through the opposite polarizer. The light is then reflected back to the viewer. To turn off the pixel, we apply a voltage to the two intersecting conductors to align the molecules so that the light is not twisted.



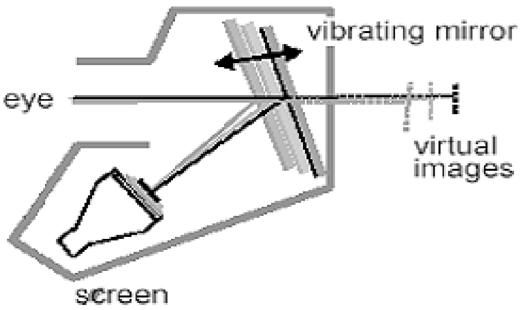
LCD Displays

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

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

Three-Dimensional Viewing Devices

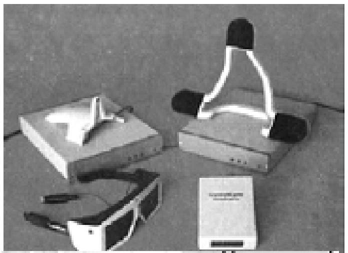
Graphics monitors for the display of three-dimensional scenes have been devised using a technique that reflects a CRT image from a vibrating, flexible mirror. In this system when varifocal mirror vibrates it changes focal length. These vibrations are synchronized with the display of an object on a CRT so that each point on the object is reflected from the mirror into spatial position corresponding to the distance of that point from a specified viewing position. This allows user to walk around an object or scene and view it from different sides.



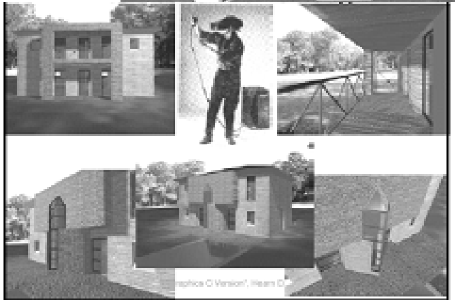
Virtual Reality Devices

Virtual reality system enables users to move and react in a computer-simulated environment. Various types of devices allow users to sense and manipulate virtual objects much as they would real objects. This natural style of interaction gives participants the feeling of being immersed in the simulated world. Virtual reality simulations differ from other computer simulations in that they require special interface devices that transmit the sights, sounds, and sensations of the simulated world to the user. These devices also record and send the speech and movements of the participants to the simulation program.

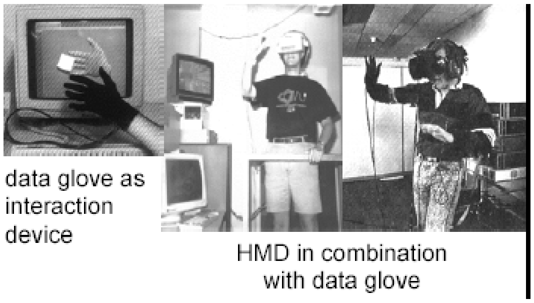
To see in the virtual world, the user wears a head-mounted display (HMD) with screens directed at each eye. The HMD contains a position tracker to monitor the location of the user's head and the direction in which the user is looking. Using this information, a computer recalculates images of the virtual world to match the direction in which the user is looking and displays these images on the HMD.



Users hear sounds in the virtual world through earphones in the HMD. The hepatic interface, which relays the sense of touch and other physical sensations in the virtual world, is the least developed feature. Currently, with the use of a glove and position tracker, the user can reach into the virtual world and handle objects but cannot actually feel them.



Another interesting simulation is **interactive walk through.** A sensing system in the headset keeps track of the viewer's opposition, so that the front and back of objects can be seen as the viewer walks and interacts with the displays. Similarly given below is a figure using a headset and a **data glove** worn on the right hand?

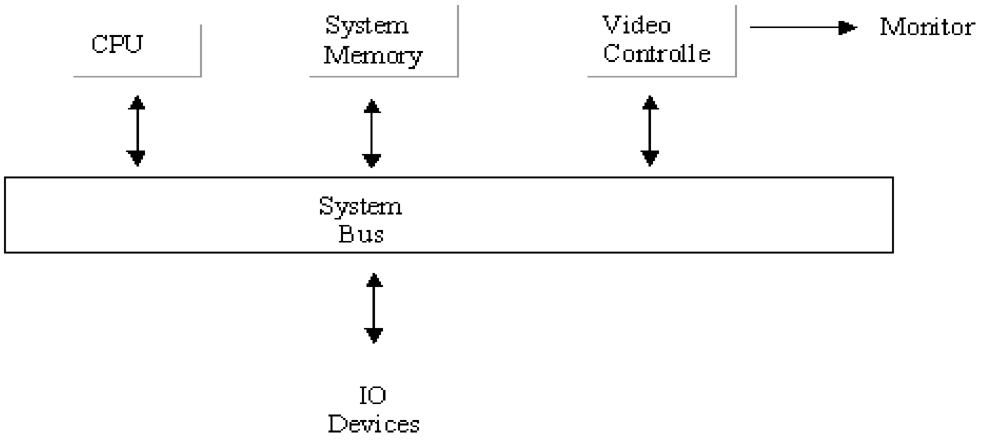

data glove as interaction device
HMD in combination with data glove

Lecture No.3 Graphics Systems II

Raster-Scan Systems

Interactive raster graphics systems typically employ several processing units. In addition to the CPU, a special purpose processor, called the **video controller** or **display controller** is used to control the operation of the display device.

Organization of a simple raster system is shown in following figure. Here the frame buffer can be anywhere in the system memory, and the video controller accesses the frame buffer to refresh the screen.



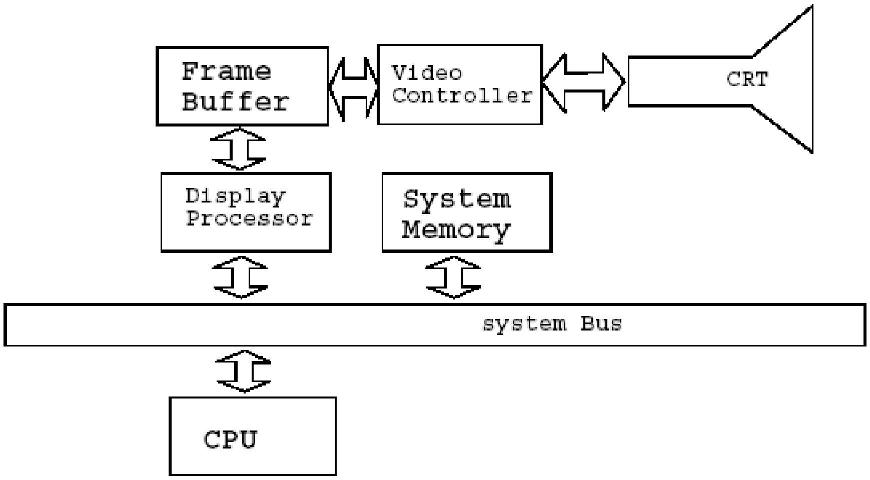
Architecture of a simple raster graphics system

In addition to the video controller more sophisticated raster systems employ other processors as coprocessors and accelerators to implement various graphics operations.

Video Controller

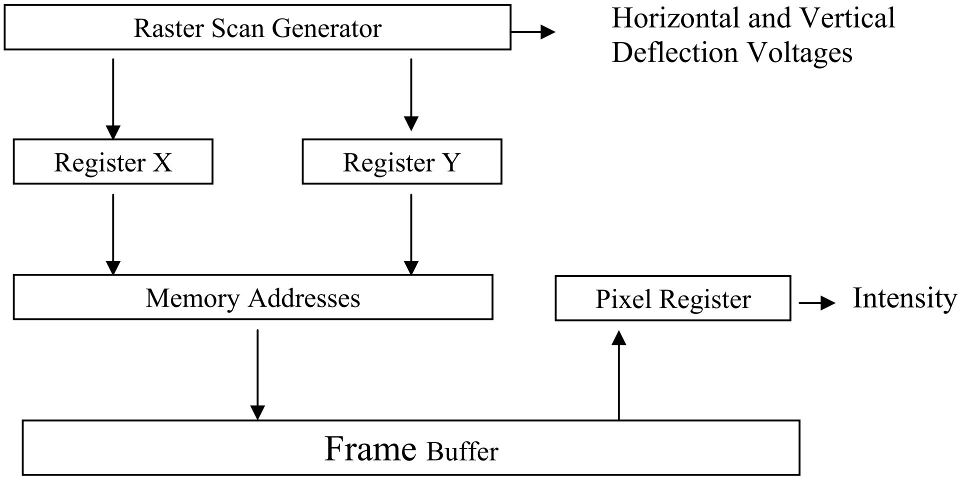
Following figure shows a commonly used organization for raster systems. A fixed area of the system memory is reserved for the frame buffer, and the video controller is given direct access to the frame-buffer memory.

Frame-buffer locations, and the corresponding screen positions, are referenced in Cartesian coordinates.



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

In the following figure the basic refresh operations of the video controller are diagrammed. Two registers are used to store the coordinates of the screen pixels. Initially, the x register is set to 0 and the y register is set to y_{max}. The value stored in the frame buffer for this pixel position is then retrieved and used to set the intensity of the CRT beam. Then the x register is incremented by 1, and the process repeated for the next pixel on the top scan line. This procedure is repeated for each pixel along the next line by resetting x register to 0 and decrementing the y register by 1. Pixels along this scan line are then processed in turn, and the procedure is repeated for each successive scan line. After cycling through all pixels along the bottom scan line y=0, the video controller resets to the first pixel position on the top scan line and the refresh process starts over.



Basic Video Controller Refresh Operations

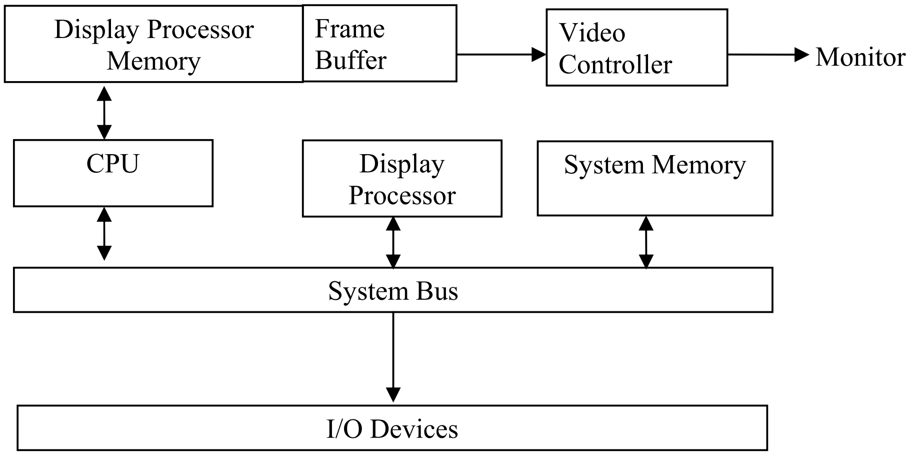
Since the screen must be refreshed at the rate of 60 frames per second, the simple procedure illustrated in above figure cannot be accommodated by typical RAM chips. The cycle time is too large making the process very slow. To speed up pixel processing, video controllers can retrieve multiple pixel values from the refresh buffer on each pass.

The multiple pixel intensities are then stored in a separate register and used to control the CRT beam intensity for a group of adjacent pixels. When that group of pixels has been processed, the next block of pixel values is retrieved from the frame buffer.

Raster Scan Display Processor
Following figure shows one way to setup the organization of a raster system containing a separate display processor, sometimes referred to as a graphics controller or a display coprocessor. The purpose of the display processor is to free the CPU from the graphics chores. In addition to the system memory, a separate display processor memory area can also be provided.

Important

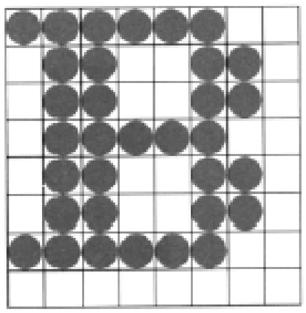
A major task of the display processor is **digitizing** a picture definition given in an application program into a set of **pixel-intensity values** for storage in the frame buffer. This digitization process is called **scan conversion**.



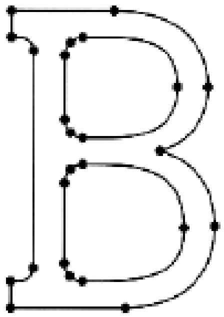
Architecture of a raster graphics system with a display processor

Raster-Scan Characters
Graphics commands specifying straight lines and other geometric objects are scan converted into a set of discrete intensity points. Scan converting a straight-line segment, for example, means that we have to locate the pixel positions closest to the line path and store the intensity for each position in the frame buffer. Similar methods are used for scan converting curved lines and polygon outlines.

Characters can be defined with rectangular grids, as shown in following figure, or they can be defined with curved outlines shown in the right hand side figure given below. The array size for character grids can vary from about 5 by 7 to 9 by 12 or more for higher-quality displays. A character grid is displayed by superimposing the rectangular grid pattern into the frame buffer at a specified coordinate position. With characters that are defined as curve outlines, character shapes are scan converted into the frame buffer.



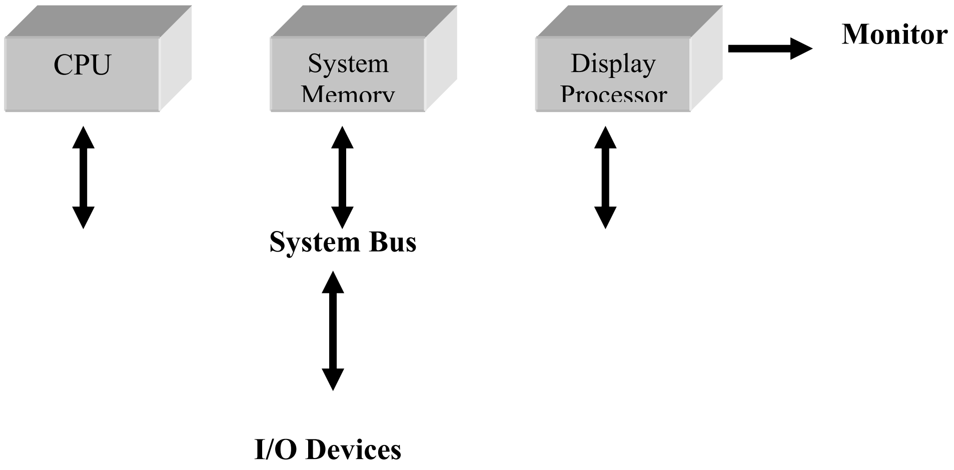
Defined as a grid of pixel positions



Defined as a curve outline

Random-Scan Systems

The organization of a simple random scan system is shown in following figure. An application program is input and stored in the system memory along with a graphics package. Graphics commands in the application program are translated by the graphics package into a display file stored in the system memory. This display file is then accessed by the display processor to refresh the screen. The display processor cycles through each command in the display file program once during every refresh cycle. Sometimes the display processor in a random scan system is referred to as a **display processing unit** or **graphics controller**.



Architecture of a simple random scan system

Graphics Card or Display Adapters

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

A video display adapter 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 adapter is simply referred as a video card.

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.

A number of display adapters are available with varying capabilities especially Intel systems support following adapters:

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

Monochrome Adapter

The simplest and the first available adapter is MA. This adapter can display only text in single color and has no graphics displaying capability. Originally this drawback only prevented the users from playing video games, but today, even the most serious business software uses graphics and color to great advantage. Hence, MA is no longer suitable, though it offers clarity and high resolution.

Hercules Adapter

The Hercules card emulates the monochrome adapter but also operates in a graphics mode. Having graphics capabilities the Hercules card became somewhat of a standard for monochrome systems.

Color Graphics Adapter

This adapter can display text as well as graphics. In text mode it operates in 25 rows by 80 column mode with 16 colors. In graphics mode two resolutions are available:

- Medium resolution graphics mode 320 * 200 with 4 colors available from palette of 16 colors
- and 640 * 200 with 2 colors

One drawback of CGA card is that it produces flicker and snow. **Flicker** is the annoying tendency of the text to flash as it moves up or down. **Snow** is the flurry of bright dots that can appear anywhere on the screen. Important

Enhanced Graphics Adapter

The EGA was introduced by IBM in 1984 as alternative to CGA card. The EGA could emulate most of the functions and all the display modes of CGA and MA. The EGA offered high resolution and was not plagued with the snow and flicker problems of CGA. In addition EGA is designed to use the enhanced color monitor capable of displaying 640 * 350 in 16 colors from a palette of 64.

The EGA card has several internal registers. A serious limitation of the EGA card is that it supports write operations to most of its internal registers, but no read operation. The result is it is not possible for software to detect and preserve the state of the adapter, which makes EGA unsuited for memory resident application or for multitasking like windows and OS/2.

Multicolor Graphics Adapter

The MCGA was designed to emulate the CGA card and to maintain compatibility with all the CGA modes. In addition to the text and graphics modes of the CGA, MCGA has two new graphics modes:

640 * 480 with 2 colors
320 * 200 in with 256 colors

Video Graphics Adapter

The VGA supports all the display modes of MA, CGA and MCGA. In addition VGA supports a graphics mode of 640 * 480 with 16 colors.

Super Video Graphics Adapter

The SVGA designation refers to enhancements to the VGA standard by independent vendors. Unlike display adapters discussed earlier SVGA does not refer to a card that meets a particular specification but to a group of cards that have different capabilities. For example one card may have resolutions 800 * 600 and 1024 * 768, whereas, another card may have same resolution but more colors. These cards have different capabilities, but still both of them are classified as SVGA. Since each SVGA card has different capabilities, you need special device driver programs for driving them. This means that unlike VGA cards which can have a single driver that works with all VGA cards, regardless of the vendor, each SVGA card must have a corresponding driver.

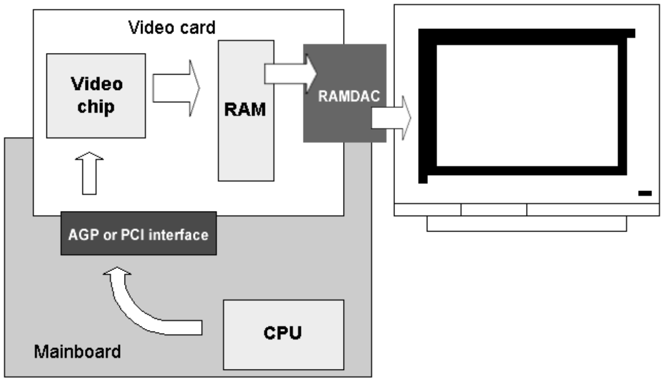
Extended Graphics Adapter

The XGA evolved from the VGA and provides greater resolution, more colors and much better performance. The XGA has a graphics processor bus mastering. Being a bus master adapter means that the XGA can take control of the system as though it were the mother board. In essence, a bus master is an adapter of the mother board. The XGA offers 2 new modes:

640 * 480 with 16 bit colors (65536 colors)
1024 * 768 with 8 bit colors (256 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

Video cards always have a certain amount of RAM. This RAM is also called the frame buffer. Today video cards hold plenty of RAM, but earlier it was more important:

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

Video card RAM is necessary to keep the entire screen image in memory. The CPU sends its data to the video card. The video processor forms a picture of the screen image and stores it in the frame buffer. This picture is a large bit map. It is used to continually update the screen image.

3D - lots of RAM

Supporting the demand for high quality 3D performance many new cards come with a frame buffer of 16 or 32 MB RAM and they use the AGP interface for better bandwidth and access to the main memory.

VRAM

Briefly, in principle all common RAM types can be used on the video card. 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. This was a RAM type, which only was used on video cards. 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 also costs twice as much. The smart feature is that the double cell allows the video processor to simultaneously read old and write new data on the same RAM address. Thus, VRAM has two gates which can be active at the same time. Therefore, it works significantly faster.

With VRAM you will not gain speed improvements increasing the amount of RAM on the graphics controller. VRAM is already capable of reading and writing simultaneously due to the dual port design.

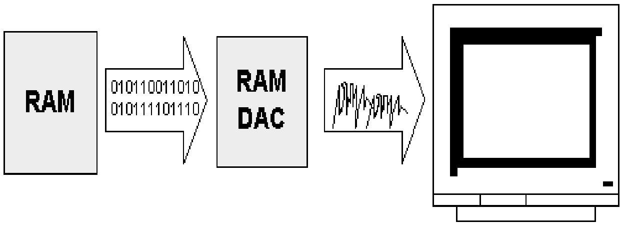
UMA and DVMT

On some older motherboards the video controller was integrated. Using SMBA (Shared Memory Buffer Architecture) or UMA (Unified Memory Architecture) in which parts of the system RAM were allocated and used as frame buffer. But sharing the memory was very slow and the standards never became very popular.

A newer version of this is found in Intel chip set 810 and the better 815, which also integrates the graphics controller and use parts of the system RAM as frame buffer. Here the system is called Dynamic Video Memory Technology (D.V.M.T.).

The RAMDAC

All traditional graphics cards have a RAMDAC chip converting the signals from digital to analog form. CRT monitors work on analog signals. The PC works with digital data which are sent to the graphics adapter. Before these signals are sent to the monitor they have to be converted into analog output and this is processed in the RAMDAC:



The recommendation on a good RAMDAC goes like this:

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

Heavy Data Transport

The original VGA cards were said to be "flat." They were unintelligent. They received signals and data from the CPU and forwarded them to the screen, nothing else. The CPU had to make all necessary calculations to create the screen image.

As each screen image was a large bit map, the CPU had to move a lot of data from RAM to the video card for each new screen image.

The graphic interfaces, like Windows, gained popularity in the early nineties. That marked the end of the "flat" VGA cards. The PC became incredibly slow, when the CPU had to use all its energy to produce screen images. You can try to calculate the required amount of data.

A screen image in 1024 x 768 in 16 bit color is a 1.5 MB bit map. That is calculated as 1024 x 768 x 2 bytes. Each image change (with a refresh rate of 75 HZ there is 75 of them each second) requires the movement of 1.5 MB data. That zaps the PC energy, especially when we talk about games with continual image changes.

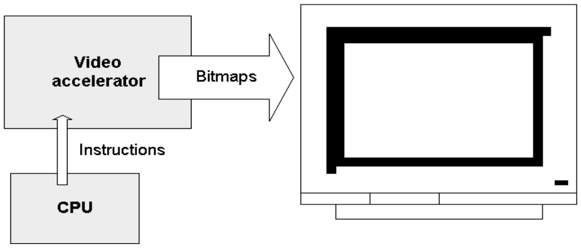
Furthermore, screen data have to be moved across the I/O bus. In the early nineties, we did not have the PCI and AGP buses, which could move large volumes of data. The transfer took place through the ISA bus, which has a very limited width. Additionally the CPUs were 386’s and early 486’s, which also had limited power.

Accelerator Cards

In the early nineties the accelerator video cards appeared. Today all cards are accelerated and they are connected to the CPU through high speed buses like PCI and AGP.

With accelerated video chips, Windows (and with that the CPU) need not calculate and design the entire bit map from image to image. The video card is programmed to draw lines, Windows and other image elements.

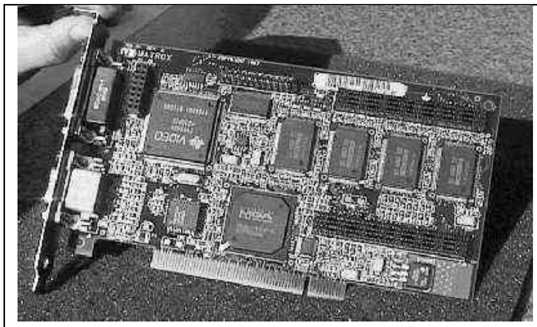
The CPU can, in a brief code, transmit which image elements have changed since the last transmission. This saves the CPU a lot of work in creating screen images. The video chip set carries the heavy load:

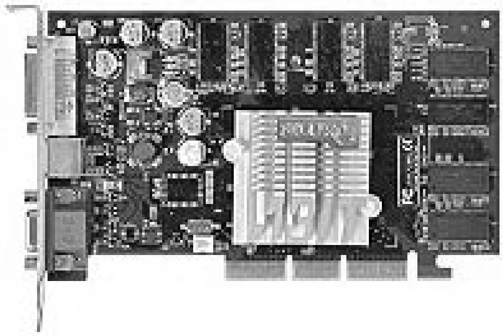


All video cards are connected to the PCI or the AGP bus, this way providing maximum data transmission. The AGP bus is an expanded and improved version of the PCI bus - used for video cards only.

Modern video cards made for 3D gaming use expensive high-end RAM to secure a sufficient bandwidth. If you for example want to see a game in a resolution of 1280 x 1024 at 80 Hz, you may need to move 400 MB of data each second - that is quite a lot. The calculation goes like this:

1280 X 1024 pixels x 32 bit (color depth) x 80 = 419,430,400 bytes
419,430,400 bytes = 409,600 kilobytes = 400 megabytes.





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 in video hardware.

Hardware manufacturers give support in hardware for libraries. Famous manufacturers include SIS, NVIDIA, ATI, INTEL etc.

Graphics Software

There is a lot of 2D and 3D software 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, CorelDraw, image viewer, paintbrush etc.

Lecture No.4 Point

Pixel: The smallest dot illuminated that can be seen on screen.

Picture: Composition of pixels makes picture that forms on whole screen

Resolution

We know that Graphics images on the screen are built up from tiny dots called picture elements or pixels. The display resolution is defined by the number of rows from top to bottom, and number of pixels from left to right on each scan line.

Since each mode uses a particular resolution. For example mode 19 uses a resolution of 200 scan lines, each containing 320 pixels across. This is often referred to as 320*200 resolution.

In general, higher the resolution, more pleasing is the picture. Higher resolution means a sharper, clearer picture, with less pronounced 'staircase' effect on lines drawn diagonally and better looking text characters. On the other hand, higher resolution also means more memory requirement for the display.

Text and Graphics Modes

We discussed different video hardware devices that include VGA cards and monitors. Video cards are responsible to send picture data to monitor each time it refresh itself. Video cards support both different text and graphics modes. Modes consist of their own refresh rate, number of colors and resolutions (number of rows multiply by number of columns). The following famous video modes that we can set in today's VGA cards on different refresh rate:

25 * 80 with 16 colors support (text mode)
320 * 200 with 8 bit colors support (graphics mode)
640 * 480 with 16 colors support (graphics mode)
640 * 480 with 8, 16, 24, 32 bit color support (graphics mode)
800 * 600 with 8, 16, 24, 32 bit color support (graphics mode)

Text and Graphics

All modes are fundamentally of two types, text or graphics. Some modes display only text and some are made only for graphics. As seen earlier, the display adapter continuously dumps the contents of the VDU (video display unit) memory on the screen.

The amount of memory required representing a character on screen in text mode and a pixel in graphics mode varies from mode to mode.

Important

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

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, only two colors can be used with each pixel.

How text displays

As seen previously text modes need two bytes in VDU memory to represent one character on screen; of these two bytes, the first byte contains the ASCII value of the character being displayed, whereas the second byte is the attribute byte. The attribute byte controls the color in which the character is being displayed.

MCQs

The ASCII value present in VDU memory must be translated into a character and drawn on the screen. This drawing is done by a character generator this is part of the display adapter or in VBIOS. The CGA has a character generator that uses 8 scan lines and 8 pixels in each of these scan lines to produce a character on screen; whereas the MA's character generator uses 9 scan lines and 14 pixels in each of these scan lines to produce a character. This larger format of MA makes the characters generated by MA much sharper and hence easier to read.

On older display adapters like MA and CGA, the character generator is located in ROM (Read Only Memory). EGA and VGA do not have a character generator ROM. Instead, character generator data is loaded into plane 2 of display RAM. This feature makes it easy for custom character set to be loaded. Multiple character sets (up to 4 for EGA and up to 8 for VGA) may reside in RAM simultaneously.

A set of BIOS services is available for easy loading of character sets. Each character set can contain 256 characters. Either one or two character sets may be active giving these adapters on the screen simultaneously. When two character sets are active, a bit in each character attribute byte selects which character set will be used for that character.

Using a ROM-BIOS service we can select the active character set. Each character in the standard character set provided with the EGA is 8 pixels wide and 14 pixels tall. Since VGA has higher resolution, it provides a 9 pixel wide by 16 pixels tall character set. Custom character set can also be loaded using BIOS VDU services.

The graphics modes can also display characters, but they are produced quite differently. The graphics modes can only store information bit by bit. The big advantage of this method is that you design characters of desired style, shape and size.

Text mode colors

In mode 3, for each character on screen there are two bytes in VDU memory, one containing the ACCII value of the character and other containing its attribute. The attribute byte controls the color of the character. The attribute byte contains three components: the foreground color (color of the character itself), the background color (color of the area not covered by the character) and the blinking component of the character. The next slide shows the breakup of the attribute byte.

Bits

7	6	5		4	3	2	1	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

So far we have seen how to set color in text modes. Setting color in graphics modes is quite different. In the graphics mode each pixel on the screen has a color associated with it. There are important differences here as compared to setting color in text mode. First, the pixels cannot blink. Second, each pixel is a discrete dot of color, there is no foreground and background. Each pixel is simply one color or another. The number of colors that each adapter can support and the way each adapter generates these colors is drastically different. But we will only discuss here colors in VGA.

Colors in VGA

IBM first introduced the VGA card in April 1987. VGA has 4 color planes – red, green, blue and intensity, with one bit from each of these planes contributing towards 1 pixel value.

There are lots of ways that you can write pixel on screen. You can write pixel on screen by using one of the following methods:

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

As we have discussed three ways to write pixel on screen. Here we will discuss all these ways practically and see how the pixel is displayed on screen. For that we will have to write code in Assembly and C languages. So get ready with these languages

Writing pixel Using Video BIOS
The following steps are involved to write 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

Below are the three lines written in assembly language that can set graphics mode 19(13h). You can use this for assembler or you can embed this code in C language using ‘***asm***’ keyword

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

To insert in C language above code will be inserted with key word asm and curly braces.

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

Description

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

The following code can be used 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 from 0 to 255 to all 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 mean default page.
Line#4: column number will be used in CX register
Line#5: row number will be used in DX register
Line#6: BIOS interrupt number 10h

Writing pixel by accessing memory directly

So far we used BIOS to draw pixel. Here we will draw pixel by accessing direct pointer to the video memory and write color value. The following steps are involved to write direct pixel without using BIOS:

Set video mode by using video BIOS routine as discussed earlier
Set any pointer to the video graphics memory address 0x0A0000.
Now write any color value in the video memory addressing

Direct Graphics Memory Access Code

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

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 direct text by setting any text mode using BIOS service and then setting direct pointer at text memory address 0x0b8000.

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

```
Mov ax,0b8000h
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. These graphics library functions then use BIOS routines or use direct memory access drivers to draw pixel on screen.

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

Steps in C language
First call Initgraph() function
and then call putpixel() function to draw pixel on screen. It takes row, column and color value as parameters.
after drawing pixel use closegraph() function to close the graphics routines provided by built in driver by Borland.

Discussion on pixel drawing methods

BIOS routines are standard routines built in VGA cards but these routines are very much slow. You will use pixel to draw filled triangle, rectangles and circles and these all will be much slower than direct memory access method. Direct memory access method allows you to write pixel directly by passing the complex BIOS routines. It is easy and faster but its programming is only convenient in mode 13h. Library functions are easier to use and even faster because these are optimized and provided with special drivers by different companies.

Drawing pixel in Microsoft Windows

So far we have been discussing writing pixel in DOS. Here we will discuss briefly how to write pixel in Microsoft Windows. Microsoft windows are a complete graphical operating system but it does not allow you to access BIOS or direct memory easily. It provides library functions (APIs) that can be used to write graphics.

By working in graphics in windows one must have knowledge about Windows GDI (graphics device interface) system.

Windows GDI functions

Here are some windows GDI functions that can be used to draw pixel e.g SetPixel and SetPixelV. Both are used to draw pixel on screen. The example and source code of writing pixel in windows will be available.

<u>Window Code Example:</u>

```
// a.cpp : Defines the entry point for the application.
//

#include "stdafx.h"
#include "resource.h"

#define MAX_LOADSTRING 100

// Global Variables:
HINSTANCE hInst;                                          //      current
instance
TCHAR szTitle[MAX_LOADSTRING];
        // The title bar text
TCHAR szWindowClass[MAX_LOADSTRING];
                // The title bar text

// Foward declarations of functions included in this code module:
ATOM                    MyRegisterClass(HINSTANCE hInstance);
BOOL                    InitInstance(HINSTANCE, int);
LRESULT CALLBACK        WndProc(HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK        About(HWND, UINT, WPARAM, LPARAM);

int APIENTRY WinMain(HINSTANCE hInstance,
              HINSTANCE hPrevInstance,
              LPSTR     lpCmdLine,
              int       nCmdShow)
```

```
{
        // TODO: Place code here.
        MSG msg;
        HACCEL hAccelTable;

        // Initialize global strings
        LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
        LoadString(hInstance, IDC_A, szWindowClass, MAX_LOADSTRING);
        MyRegisterClass(hInstance);

        // Perform application initialization:
        if (!InitInstance (hInstance, nCmdShow))
        {
                return FALSE;
        }

        hAccelTable = LoadAccelerators(hInstance, (LPCTSTR)IDC_A);

        // Main message loop:
        while (GetMessage(&msg, NULL, 0, 0))
        {
                if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
                {
                        TranslateMessage(&msg);
                        DispatchMessage(&msg);
                }
        }

        return msg.wParam;
}

//  FUNCTION: MyRegisterClass()
//
//  PURPOSE: Registers the window class.
//
//  COMMENTS:
//
//    This function and its usage is only necessary if you want this code
//    to be compatible with Win32 systems prior to the 'RegisterClassEx'
//    function that was added to Windows 95. It is important to call this function
//    so that the application will get 'well formed' small icons associated
//    with it.
//
ATOM MyRegisterClass(HINSTANCE hInstance)
{
        WNDCLASSEX wcex;

        wcex.cbSize = sizeof(WNDCLASSEX);

        wcex.style = CS_HREDRAW | CS_VREDRAW;
```

```
	wcex.lpfnWndProc	= (WNDPROC)WndProc;
	wcex.cbClsExtra		= 0;
	wcex.cbWndExtra		= 0;
	wcex.hInstance		= hInstance;
	wcex.hIcon			= LoadIcon(hInstance, (LPCTSTR)IDI_A);
	wcex.hCursor		= LoadCursor(NULL, IDC_ARROW);
	wcex.hbrBackground	= (HBRUSH)(COLOR_WINDOW+1);
	wcex.lpszMenuName	= (LPCSTR)IDC_A;
	wcex.lpszClassName	= szWindowClass;
	wcex.hIconSm		= LoadIcon(wcex.hInstance, (LPCTSTR)IDI_SMALL);

	return RegisterClassEx(&wcex);
}

//
//   FUNCTION: InitInstance(HANDLE, int)
//
//   PURPOSE: Saves instance handle and creates main window
//
//   COMMENTS:
//
//        In this function, we save the instance handle in a global variable and
//        create and display the main program window.
//
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
   HWND hWnd;

   hInst = hInstance; // Store instance handle in our global variable

   hWnd = CreateWindow(szWindowClass, szTitle, WS_OVERLAPPEDWINDOW,
      CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);

   if (!hWnd)
   {
      return FALSE;
   }

   ShowWindow(hWnd, nCmdShow);
   UpdateWindow(hWnd);

   return TRUE;
}

//
//  FUNCTION: WndProc(HWND, unsigned, WORD, LONG)
//
//  PURPOSE:  Processes messages for the main window.
//
//  WM_COMMAND - process the application menu
//  WM_PAINT          - Paint the main window
//  WM_DESTROY    - post a quit message and return
//
//
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam,
LPARAM lParam)
{
    int wmId, wmEvent;
    PAINTSTRUCT ps;
    HDC hdc;
    TCHAR szHello[MAX_LOADSTRING];
    LoadString(hInst, IDS_HELLO, szHello, MAX_LOADSTRING);

    switch (message)
    {
        case WM_COMMAND:
            wmId    = LOWORD(wParam);
            wmEvent = HIWORD(wParam);
            // Parse the menu selections:
            switch (wmId)
            {
                case IDM_ABOUT:
                  DialogBox(hInst, (LPCTSTR)IDD_ABOUTBOX, hWnd,
(DLGPROC)About);
                  break;
                case IDM_EXIT:
                  DestroyWindow(hWnd);
                  break;
                default:
                  return    DefWindowProc(hWnd,    message,    wParam,
lParam);
            }
            break;
        case WM_PAINT:
            {
            hdc = BeginPaint(hWnd, &ps);
            // TODO: Add any drawing code here...
            RECT rt;
            GetClientRect(hWnd, &rt);
            int j=0;
            //To draw some pixels of RED colour on the screen
            for(int i=0;i<100;i++)
            {
                SetPixel(hdc,i+j,10,RGB(255,0,0));
                j+=6;
            }

            EndPaint(hWnd, &ps);
            }
            break;
```

```
                case WM_DESTROY:
                        PostQuitMessage(0);
                        break;
                default:
                        return DefWindowProc(hWnd, message, wParam, lParam);
  }
  return 0;
}
// Mesage handler for about box.
LRESULT CALLBACK About(HWND hDlg, UINT message, WPARAM wParam,
LPARAM lParam)
{
        switch (message)
        {
                case WM_INITDIALOG:
                                return TRUE;

                case WM_COMMAND:
                        if (LOWORD(wParam) == IDOK || LOWORD(wParam) ==
IDCANCEL)
                        {
                                EndDialog(hDlg, LOWORD(wParam));
                                return TRUE;
                        }
                        break;
        }
   return FALSE;
}
```

Lecture No.5 Line Drawing Techniques

Line

A **line**, or **straight line**, is, roughly speaking, an (infinitely) thin, (infinitely) long, straight geometrical object, i.e. a curve that is long and straight. Given two points, in Euclidean geometry, one can always find exactly one line that passes through the two points; this line provides the shortest connection between the points and is called a straight line. Three or more points that lie on the same line are called ***collinear***. Two different lines can intersect in at most one point; whereas two different planes can intersect in at most one line. This intuitive concept of a line can be formalized in various ways.

A line may have three forms with respect to slope i.e. it may have slope = 1 as shown in following figure (a), or may have slope < 1 as shown in figure (b) or it may have slope > 1 as shown in figure (c). Now if a line has slope = 1 it is very easy to draw the line by simply starting form one point and go on incrementing the x and y coordinates till they reach the second point. So that is a simple case but if slope < 1 or is > 1 then there will be some problem.

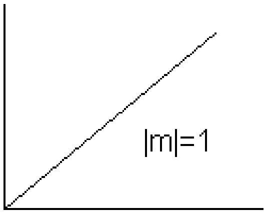


figure (a)

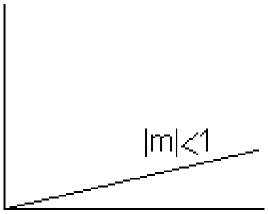


figure (b)

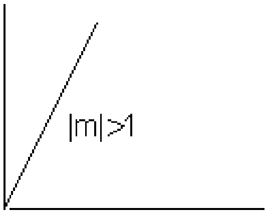


figure (c)

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

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

Incremental line algorithm

This algorithm exploits simple line equation $y = m\,x + b$
Where $m = dy / dx$
and $b = y - m\,x$

slop less X increment
slop greater than 1 y increment

Now check if $|m| < 1$ then starting at the first point, simply increment x by 1 (unit increment) till it reaches ending point; whereas calculate y point by the equation for each x and conversely if $|m|>1$ then increment y by 1 till it reaches ending point; whereas calculate x point corresponding to each y, by the equation.

Now before moving ahead let us discuss why these two cases are tested. First if |m| is less than 1 then it means that for every subsequent pixel on the line there will be unit increment in x direction and there will be less than 1 increment in y direction and vice versa for slope greater than 1. Let us clarify this with the help of an **example**:

Suppose a line has two points p1 (10, 10) and p2 (20, 18)
Now difference between y coordinates that is dy = y2 – y1 = 18 – 10 = 8
Whereas difference between x coordinates is dx = x2 – x1 = 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.

Consider the case of another line with points p1 (10, 10) and p2 (16, 20)
Now difference between y coordinates that is dy = y2 – y1 = 20 – 10 = 10
Whereas difference between x coordinates is dx = x2 – x1 = 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 having discussed this concept at length let us learns the algorithm to draw a line using above technique, called incremental line algorithm:

```
Incremental_Line (Point p1, Point p2)
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:
Well above algorithm is quite simple and easy but firstly it involves lot of mathematical calculations that is for calculating coordinate using equation each time secondly it works only in incremental direction.

We have another algorithm that works fine in all directions and involving less calculation mostly only addition; which will be discussed in next topic.

Digital Differential Analyzer (DDA) Algorithm:
DDA abbreviated for digital differential analyzer has very simple technique. Find difference dx and dy between x coordinates and y coordinates respectively ending points of a line. If |dx| is greater than |dy|, than |dx| will be step and otherwise |dy| will be step.

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

step = |dy|

Now very simple to say that step is the total number of pixel required for a line. Next step is to divide dx and dy by step to get xIncrement and yIncrement that is the increment required in each step to find next pixel value.

xIncrement = dx/step
yIncrement = dy/step

Next a loop is required that will run step times. In the loop drawPixel and add xIncrement in x1 by and yIncrement in y1.

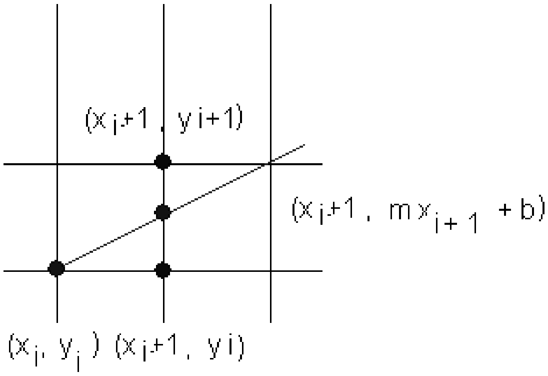
To sum-up all above in the algorithm, we will get,

```
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:

There is serious criticism on the algorithm that is use of floating point calculation. They say that when we have to draw points that should have integers as coordinates then why to use floating point calculation, which requires more space as well as they have more computational cost.

Therefore there is need to develop an algorithm which would be based on integer type calculations. Therefore, work is done and finally we will come up with an algorithm “Bresenham Line Drawing algorithm” which will be discussed next.



Bresenham's Line 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.

If the current point is (x_i, y_i), the next point can be either (x_i+1,y_i) or (x_i+1,y_i+1) . The actual position on the line is $(x_i+1, m(x_i+1)+c)$. Calculating the distance between the true point, and the two alternative pixel positions available gives:

$$
\begin{aligned}
d_1 &= y - y_i \\
&= m * (x+1)+b-y_i \\
d_2 &= y_i + 1 - y \\
&= y_i + 1 - m (x_i + 1) - b
\end{aligned}
$$

Let us magically define a decision function p, to determine which distance is closer to the true point. By taking the difference between the distances, the decision function will be positive if d1 is larger, and negative otherwise. A positive scaling factor is added to ensure that no division is necessary, and only integer math need be used.

$$
\begin{aligned}
p_i &= dx\,(d_1-d_2) \\
p_i &= dx\,(2m * (x_i+1) + 2b - 2y_i-1) \\
p_i &= 2\,dy\,(x_i+1) - 2\,dx\,y_i + dx\,(2b-1) \text{ ------------------ (i)} \\
p_i &= 2\,dy\,x_i - 2\,dx\,y_i + k \quad \text{------------------ (ii)}
\end{aligned}
$$

where $k=2\,dy + dx\,(2b-1)$

Then we can calculate pi+1 in terms of pi without any x_i , y_i or k .

$$
\begin{aligned}
p_{i+1} &= 2\,dy\,x_{i+1} - 2\,dx\,y_{i+1} + k \\
p_{i+1} &= 2\,dy\,(x_i + 1) - 2\,dx\,y_{i+1} + k \qquad \text{since } x_{i+1}= x_i + 1 \\
p_{i+1} &= 2\,dy\,x_i + 2\,dy- 2\,dx\,y_{i+1} + k \quad \text{------------------ (iii)}
\end{aligned}
$$

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

$$
\begin{aligned}
d_1<d_2 &\Rightarrow d_1-d_2<0 \\
&\Rightarrow p_i<0 \\
&\Rightarrow \mathbf{p_i+1= p_i + 2\,dy}
\end{aligned}
$$

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

$$
\begin{aligned}
d_1>d_2 &\Rightarrow d_1-d_2>0 \\
&\Rightarrow p_i>0 \\
&\Rightarrow \mathbf{p_i+1= p_i + 2\,dy - 2\,dx}
\end{aligned}
$$

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 how to calculate initial value of p_i. For that use equation (i) and put values (x_1, y_1)

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

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

$$p_i = 2\,dy\,x_1 + 2\,dy - 2\,dx\,y_i + dx\,(\,2\,(y_1 - mx_1)\,-1\,)$$
$$p_i = 2\,dy\,x_1 + 2\,dy - 2\,dx\,y_i + 2\,dx\,y_1 - 2\,dy\,x_1\ - dx$$
$$p_i = 2\,dy\,x_1 + 2\,dy - 2\,dx\,y_i + 2\,dx\,y1 - 2\,dy\,x1\ - dx$$

there are certain figures will cancel each other shown in same different colour

$$\mathbf{p_i = 2\,dy\ - dx}$$

Thus Bresenham's line drawing algorithm is as follows:

```
$dx = x_2 - x_1$
$dy = y_2 - y_1$
$p = 2dy - dx$
$c1 = 2dy$
$c2 = 2(dy - dx)$
$x = x_1$
$y = y_1$
plot (x,y,colour)
while ($x < x_2$ )
    x++;
    if ($p < 0$)
        $p = p + c_1$
    else
        $p = p + c_2$
    y++
    plot (x,y,colour)
```

Again, this algorithm can be easily generalized to other arrangements of the end points of the line segment, and for different ranges of the slope of the line.

Improving performance

Several techniques can be used to improve the performance of line-drawing procedures. These are important because line drawing is one of the fundamental primitives used by most of the other rendering applications. An improvement in the speed of line-drawing will result in an overall improvement of most graphical applications.

Removing procedure calls using macros or inline code can produce improvements. Unrolling loops also may produce longer pieces of code, but these may run faster.

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 replaced:

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

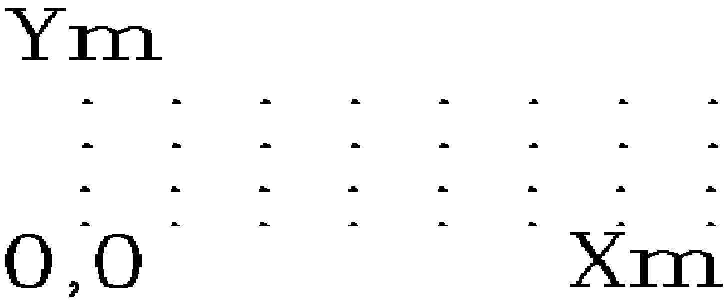
Addition in fixed point representation occurs by adding fractional and integer components separately, and only transferring any carry-over from the fractional result to the integer result. The sequence could be implemented using the following two integer additions: ADD Yfrac,Mfrac ; ADC Yint,Mint

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 - 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.



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

addr(X, Y) = addr(0,0) + Y rows * (Xm + 1) + X (all in bytes)

addr(X,Y) = the memory address of pixel (X,Y)

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.

Example:

For a system with 640 × 480 pixel resolution, find the address of pixel X = 340, Y = 150

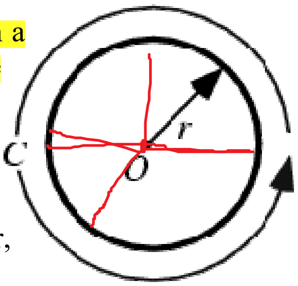
addr(340, 150) = addr(0,0) + 150 * 640 (bytes/row) + 340

= base + 96,340 is the byte location

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

Lecture No.6 Circle Drawing Techniques

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 $d = 2r$. The angle a circle subtends from its center is a full angle, equal to 360° or 2π radians.

MCQs

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

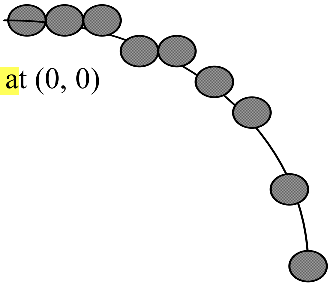
The perimeter C of a circle is called the circumference, and is given by

$$C = 2\pi r$$

Circle Drawing Techniques

First of all for circle drawing we need to know what the input will be. Well the input will be one center point (x, y) and radius r. Therefore, using these two inputs there are a number of ways to draw a circle. They involve understanding level very simple to complex and reversely time complexity inefficient to efficient. We see them one by one giving comparative study.

Circle drawing using Cartesian coordinates



This technique uses the equation for a circle on radius r centered at (0, 0) given as:

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

an obvious choice is to plot

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

Obviously in most of the cases the circle is not centered at (0, 0), rather there is a center point (x_c, y_c); other than (0, 0). Therefore the equation of the circle having center at point (x_c, y_c):

$$(x - x_c)^2 + (y - y_c)^2 = r^2,$$

this implies that ,

$$y = y_c \pm \sqrt{r^2 - (x - x_c)^2}$$

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

Circle1 (xcenter, ycenter, radius)

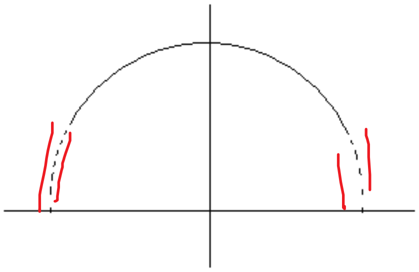
for x = radius - xcenter to radius + xcenter

$y = x_c + \sqrt{r^2 - (x - x_c)^2}$

drawPixel (x, y)

$y = x_c - \sqrt{r^2 - (x - x_c)^2}$

drawPixel (x, y)



This works, but is inefficient because of the multiplications and square root operations. It also creates large gaps in the circle for values of x close to r as shown in the above figure.

Circle drawing using Polar coordinates

A better approach, to eliminate unequal spacing as shown in above figure is to 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

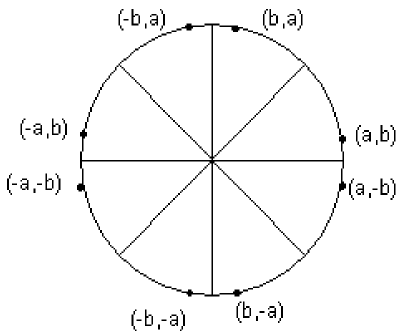
$$x = x_c + r\cos\theta$$
$$y = y_c + r\sin\theta$$

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. The step size chosen for θ depends on the application and the display device. Larger angular separations along the circumference can be connected with straight-line segments to approximate the circular path. For a more continuous boundary on a raster display, we can set the step size at 1/r. This plots pixel positions that are approximately one unit apart. MC

Now let us see how this technique can be sum up in algorithmic form.

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)
```



Again this is very simple technique and also solves problem of unequal space but unfortunately this technique is still inefficient in terms of calculations involves especially floating point calculations.

Calculations can be reduced by considering the symmetry of circles. The shape of circle is similar in each quadrant. We can generate the circle section in the second quadrant of the xy-plane by noting that the two circle sections are symmetric with respect to the y axis and circle sections in the third an fourth quadrants can be obtained from sections in the first and second quadrants by considering symmetry about the x axis. We can take this one step further and note that there is also symmetry between octants. Circle sections in adjacent octants within one quadrant are symmetric with respect to the 45° line dividing the two octants. These symmetry conditions are illustrated in above figure.

Therefore above algorithm can be optimized by using symmetric octants. Let's see:

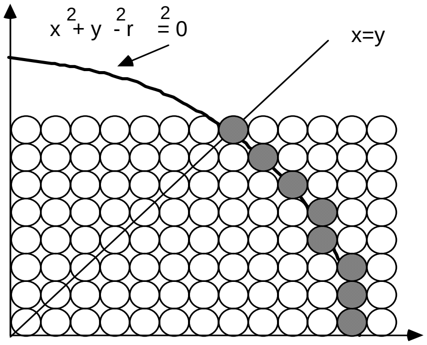
```
Circle2 (xcenter, ycenter, radius)
for $\theta = 0$ to $\pi / 4$ step 1/r
        $x = x_c + r * \cos\theta$
        $y = y_c + r * \sin\theta$
        DrawSymmetricPoints (xcenter, ycenter, x, y)
```

```
DrawSymmeticPoints (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)
```

Hence we have reduced half the calculations by considering symmetric octants of the circle but as we discussed earlier inefficiency is still there and that is due to the use of floating point calculations. In next algorithm we will try to remove this problem.

Midpoint circle Algorithm

As in the Bresenham line drawing algorithm we derive a decision parameter that helps us to determine whether or not to increment in the y coordinate against increment of x coordinate or vice versa for slope > 1. Similarly here we will try to derive decision parameter which can give us closest pixel position.



Let us consider only the first octant of a circle of radius r centred on the origin. We begin by plotting point (r, 0) and end when x < y.

The decision at each step is whether to choose the pixel directly above the current pixel or the pixel; which is above and to the left (8-way stepping).

Assume:

$P_i = (x_i, y_i)$ is the current pixel.

$T_i = (x_i, y_i + 1)$ is the pixel directly above

$S_i = (x_i - 1, y_i + 1)$ is the pixel above and to the left.

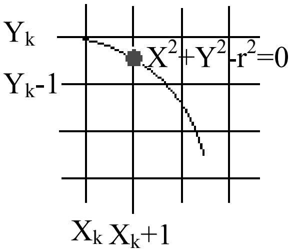
To apply the midpoint method, we define a circle function:

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

Therefore following relations can be observed:

mcqs in quiz

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



The circle function tests given above are performed for the midpoints between pixels near the circle path at each sampling step. Thus, the circle function is the decision parameter in the midpoint algorithm, and we can set up incremental calculations for this function as we did in the line algorithm.

Figure given above shows the midpoint between the two candidate pixels at sampling position x_k+1. Assuming we have just plotted the pixel at (x_k, y_k), we next need to determine whether the pixel at position (x_k + 1, y_k), we next need to determine whether the pixel at position (x_k+1, y_k) or the one at position (x_k+1, y_k-1) is closer to the circle. Our decision parameter is the circle function evaluated at the midpoint between these two pixels:

$$P_k = f_{\text{circle}}(x_k + 1, y_k - ½)$$
$$P_k = (x_k + 1)^2 + (y_k - ½)^2 - r^2 \qquad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots(1)$$

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.

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 $x_{k+2}= x_{k+1}+1=x_k+1+1 = x_k+2$:

$$P_{k+1} = f_{\text{circle}}(x_{k+1} + 1,\ y_{k+1} - ½)$$
$$P_{k+1} = [(x_k + 1) + 1]^2 + (y_{k+1} - ½)^2 - r^2 \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots(2)$$

Subtracting (1) from (2), we get

$$P_{k+1} - P_k = [(x_k + 1) + 1]^2 + (y_{k+1} - ½)^2 - r^2 - (x_k + 1)^2 - (y_k - ½)^2 + r^2$$

or

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

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:

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

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

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

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

Now a similar case that we observe in line algorithm is that how would starting P_k be evaluated. For this at the start pixel position will be (0, r). Therefore, putting this value is equation , we get

$$P_0 = (0 + 1)^2 + (r - ½)^2 – r^2$$
$$P_0 = 1 + r^2 - r + ¼ – r^2$$
$$P_0 = 5/4 – r$$

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 sum up all in the algorithm:

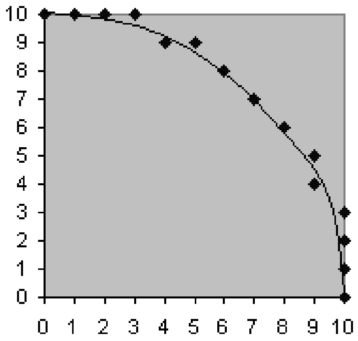
MidpointCircle (xcenter, ycenter, radius)

```
y = r;
x = 0;
p = 1 - r;
do
        DrawSymmetricPoints (xcenter,  ycenter, x, y)
        x = x + 1
        If p < 0 Then
                p = p + 2 * ( x + 1 ) + 1
        else
                y = y - 1
                p = p + 2 * ( x + 1 ) – 2 * ( y  - 1 ) + 1
while ( x  <  y )
```

Now let us consider an example to calculate first octant of the circle using above algorithm; while one quarter is displayed where you can observe that exact circle is passing between the points calculated in a raster circle.

Example:

xcenter= 0 ycenter= 0 radius= 10

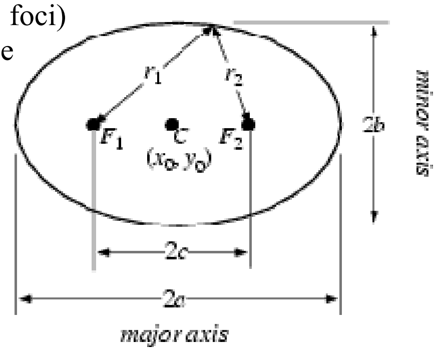


p	x	Y
-9	0	10
-6	1	10
-1	2	10
6	3	10
-3	4	9
8	5	9
5	6	8
6	7	7

Lecture No.7 Ellipse and Other Curves

Ellipse

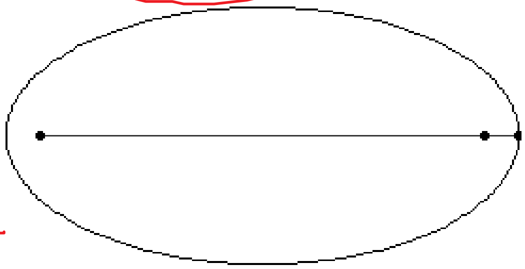
An ellipse is a curve that is the locus of all points in the plane the sum of whose distances r1 and r2 from two fixed points F1 and F2, (the foci) separated by a distance of $2c$ is a given positive constant $2a$. This results in the two-center bipolar coordinate equation:



r1 + r2 = 2a ✓

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.

MCQs The ellipse was first studied by Menaechmus, investigated by Euclid, and named by Apollonius. The focus and conic section directrix of an ellipse were considered by Pappus. In 1602, Kepler believed that the orbit of Mars was oval; he later discovered that it was an ellipse with the Sun at one focus. In fact, Kepler introduced the word "focus" and published his discovery in 1609. In 1705 Halley showed that the comet now named after him moved in an elliptical orbit around the Sun (MacTutor Archive). An ellipse rotated about its minor axis gives an oblate spheroid, while an ellipse rotated about its major axis gives a prolate spheroid.



Let an ellipse lie along the *x*-axis and find the equation of the figure given above where F1 and F2 are at (-c, 0) and (c, 0). In Cartesian coordinates,

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

✓ ✓

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

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

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,

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

Grouping the *x* terms then gives

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

this can be written in the simple form

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

Defining a new constant

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

puts the equation in the particularly simple form

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

The parameter *b* is called the semi-minor axis by analogy with the parameter *a*, which is called the semi-major axis (assuming *b* < *a*). The fact that *b* as defined at right is actually the semi-minor axis is easily shown by letting r1 and r2 be equal. Then two right triangles are produced, each with hypotenuse *a*, base *c*, and height b = $\sqrt{a^2 - c^2}$. Since the largest distance along the minor axis will be achieved at this point, *b* is indeed the semi-minor axis.

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

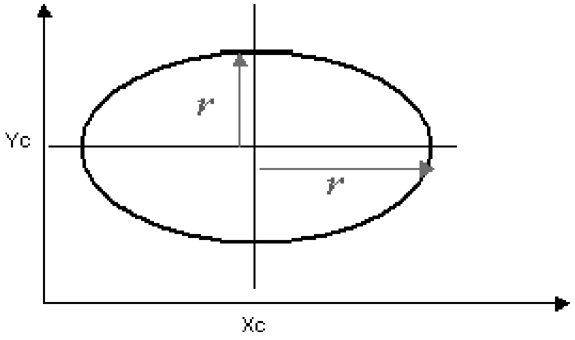
$$\boxed{\frac{(x - x_0)^2}{a^2} + \frac{(y - y_0)^2}{b^2} = 1.}$$

Ellipse Drawing Techniques

Now we already understand circle drawing techniques. One way to draw ellipse is to use the following equation:

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

where x_0 may be replaced by x_c in case of center other than origin and same in case of y.

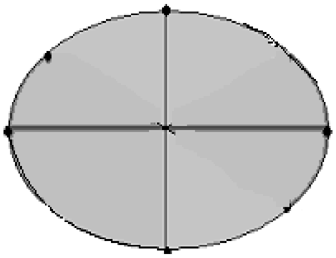


Another way is to use polar coordinates r and θ, for that we have parametric equations:

$x = x_c + r_x \cos\theta$
$y = y_c + r_y \sin\theta$

Four-way symmetry

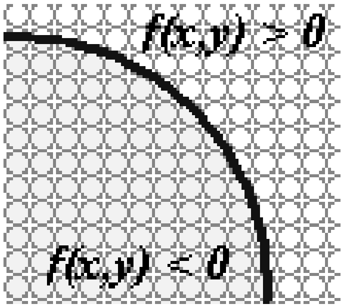
Symmetric considerations can be had to further reduce computations. An ellipse in standard position is symmetric between quadrants, but unlike a circle, it is not symmetric between the two octants of a quadrant. Thus, we must calculate pixel positions along the elliptical arc throughout one quadrant, and then we obtain positions in the remaining three quadrants by symmetry as shown in at right figure.



Midpoint ellipse algorithm

Consider an ellipse centered at the origin:

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



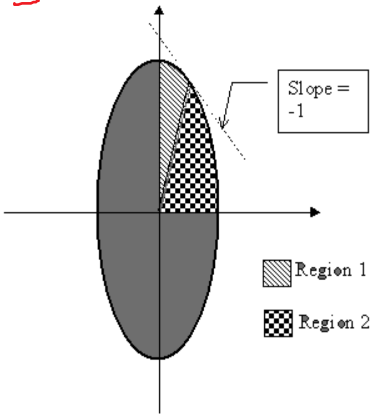
To apply the midpoint method, we define an ellipse function:

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

Therefore following relations can be observed:

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

Now as you have some idea that ellipse is different from circle. Therefore, a similar approach that is applied in circle can be applied here using some different sampling direction as shown in the figure at right. There are two regions separated in one octant.



Therefore, idea is that in region 1 sampling will be at x direction; whereas y coordinate will be related to decision parameter. In region 2 sampling will be at y direction; whereas x coordinate will be related to decision parameter.

So consider first region 1. We will start at $(0, r_y)$; we take unit steps in the x direction until we reach the boundary between region 1 and region 2. Then we switch to unit steps in the y direction over the remainder of the curve in the first quadrant. At each step, we need to test the value of the slope of the curve. The ellipse slope is calculated from following equation:

$$dy / dx = -2\ r_y^2x^2 / 2\ r_x^2y^2$$

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

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

Therefore, we move out of region 1 whenever

$$2\ r_y^2x^2 >= 2\ r_x^2y^2$$

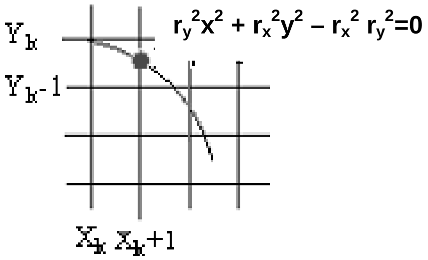


Figure at right shows the midpoint between the two candidate pixels at sampling position $x_k + 1$ in the first region. Assuming position (x_k, y_k) has been selected at the previous step; we determine the next position along the ellipse path by evaluating the decision parameter at this midpoint:

$$P1k = f_{ellipse} (x_k + 1, y_k – ½)$$

$$f_{ellipse} (x_k +1, y_k – ½) = r^2{}_y (x_k + 1)^2 + r_x^2 (y_k – ½)^2 – r_x^2\ r_y^2 \text{-------}(1)$$

If $p_k < 0$, this midpoint is inside the ellipse and the pixel on scan line y_k is closer to the ellipse boundary. Otherwise, the mid position is outside or on the ellipse boundary, 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=x_k+2:

$$f_{ellipse} (x_{k+1} +1, y_{k+1} – ½) = r_y^2 [(x_k + 1) + 1]^2 + r_x^2 (y_{k+1} – ½)^2 – r_x^2\ r_y^2 \text{ ---}(2)$$

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

$$P_{k+1}1 = P_k1 + 2\ r_y^2 (x_k + 1) + r_x^2 (y_{k+1}^2– y_k^2) - r_x^2 (y_{k+1} - y_k) + r_y^2$$

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:

$$P1_{k+1} = P_k1 + 2\ r_y^2 (x_k + 1) + r_x^2 (y\ k^2 – y_k^2) - r_x^2 (y_k - y_k) + r_y^2$$

$$P_{k+1}1 = P_k1 + 2\ r_y^2 (x_k + 1) + r_y^2$$

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

$$P_{k+1}1 = P_k1 + 2\, r_y^2\, (x_k + 1) + r^2_x\, ((y_k - 1)^2 - y_k^2) - r_x^2\, (y_k - 1 - y_k) + r_y^2$$
$$P1_{k+1} = P1_k + 2\, r_y^2 (x_k + 1) + r_x^2\, (-2\, y_k + 1) - r_x^2 (-1) + r_y^2$$
$$P1_{k+1} = P1_k + 2\, r_y^2 (x_k + 1) - 2\, r_x^2\, y_k + r^2_x + r_x^2 + r_y^2$$
$$P1_{k+1} = P1_k + 2\, r_y^2 (x_k + 1) - 2\, r_x^2 (y_k - 1) + r_y^2$$

Now a similar case that we observe in line algorithm is from where starting P_k will evaluate. For this at the start pixel position will by (0, r_y). Therefore, putting this value is equation , we get

$$P1_0 = r_y^2 (0 + 1)^2 + r_x^2 (r_y - ½)^2 - r_x^2\, r_y^2$$
$$P1_0 = r^2_y + r_x^2\, r_y^2 - r^2_x\, r_y + ¼\, r_x^2 - r_x^2\, r_y^2$$
$$P1_0 = r^2_y - r_x^2\, r_y + ¼\, r_x^2$$

Similarly same procedure will be adapted for region 2 and decision parameter will be calculated, here we are giving decision parameter and there derivation is left as an exercise for the students.

$$P_{k+1}2 = P_k2 - 2\, r_x^2\, (y_k + 1) + r_x^2 \qquad ,$$
if $p_k2 > 0$

$$P_{k+1}2 = P_k2 + 2\, r_y^2\, (x_k + 1) - 2\, r_x^2\, y_k + r_x^2$$
otherwise

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

$$P_02 = r_y^2\, (x_0 + ½) + r_x^2\, (y_0 - 1)^2 - r_x^2 r_y^2$$

Since all increments are integer. Finally sum up all in the algorithm:

MidpointEllipse (xcenter, ycenter, r $_x$, r $_y$)

```
x =0
x =0
    y = $r_y$
    do
        DrawSymmetricPoints (xcenter, ycenter, x, y)
        $P_01 = r_y^2 - r_x^2\, r_y + ¼\, r_x^2$          x = x +1

    If $p1_k < 0$
        $P_{k+1}1 = P_k1 + 2\, r_y^2\, (x_k + 1) + r_y^2$ else
        $P_{k+1}1 = P_k1 + 2\, r_y^2\, (x_k + 1) - 2\, r_x^2\, (y_k - 1) + r^2_y$
         y = y -1

        $P_02 = r_y^2\, (x_0 + ½) + r_x^2\, (y_0 - 1)^2 - r_x^2\, r_y^2$      y = y -1
    If $p2_k > 0$
```

$P_{k+1}2 = P_k2 - 2\, r_x^2\, (y_k + 1) + r_x^2$ else

$P_{k+1}2 = P_k2 + 2\, r_y^2\, (x_k + 1) - 2\, r_x^2\, y_k + r_x^2$ $x = x + 1$

while ($2\, r_y^2 x^2 >= 2\, r_x^2 y^2$)

Other Curves

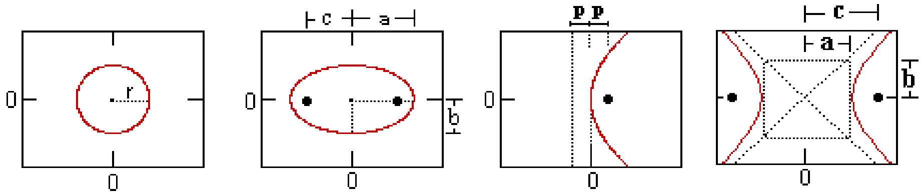
Various curve functions are useful in object modeling, animation path specifications, data, function graphing, and other graphics applications. Commonly encountered curves include conics, trigonometric and exponential functions, probability distributions, general polynomials, and spline functions. 3

Displays of these curves can be generated with methods similar to those discussed for the circle and ellipse. We can obtain positions along curve paths directly from explicit representations y = f(x) or from parametric forms. Alternatively, we could apply the incremental midpoint method to plot curves described with implicit functions f(x,y) = 0.

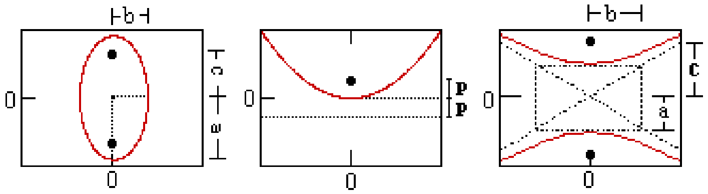
Conic Sections

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

Circle Ellipse (h)Parabola (h)Hyperbola (h)



Ellipse (v)Parabola (v)Hyperbola (v)



The general equation for a conic section:
$Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0$

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

If B^2 - 4AC is then the curve is a... MCQs

< 0 ellipse, circle, point or no curve.
= 0 parabola, 2 parallel lines, 1 line or no curve.
> 0 hyperbola or 2 intersecting lines.

For any of the below with a center (j, k) instead of (0, 0), replace each x term with (x-j) and each y term with (y-k).

Important

MCQs

	Circle	**Ellipse**	**Parabola**	**Hyperbola**
Equation (horiz. vertex):	$x^2 + y^2 = r^2$	$x^2 / a^2 + y^2 / b^2 = 1$	$4px = y^2$	$x^2 / a^2 - y^2 / b^2 = 1$
Equations of Asymptotes:				$y = \pm (b/a)x$
Equation (vert. vertex):	$x^2 + y^2 = r^2$	$y^2 / a^2 + x^2 / b^2 = 1$	$4py = x^2$	$y^2 / a^2 - x^2 / b^2 = 1$
Equations of Asymptotes:				$x = \pm (b/a)y$

Variables:	r = circle radius	a = major radius (= 1/2 length major axis)
		b = minor radius (= 1/2 length minor axis)
		c = distance center to focus
			p = distance from vertex to focus (or directrix)	a = 1/2 length major axis
				b = 1/2 length minor axis
				c = distance center to focus

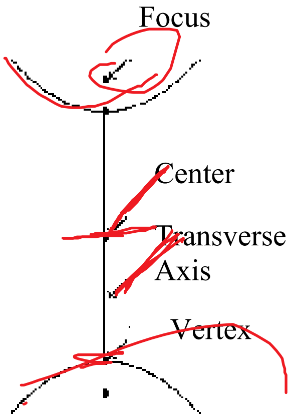
Eccentricity:	0		c/a	c/a
Relation to Focus:	p = 0	$a^2 - b^2 = c^2$	p = p	$a^2 + b^2 = c^2$
Definition: is the locus of all points which meet the condition...	distance to the origin is constant	sum of distances to each focus is constant	distance to focus = distance to directrix	difference between distances to each foci is constant

Hyperbola

We begin this section with the definition of a 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**.

MCQs

Each hyperbola consists of two **branches**. The line segment; which connects the two foci intersects the hyperbola at two points, called the **vertices**. The line segment; which ends at these vertices is called the **transverse axis** and the midpoint of this line is called the **center of the hyperbola**. See figure at right for a sketch of a hyperbola with these pieces identified.



Note that, as in the case of the ellipse, a hyperbola can have a vertical or horizontal orientation.

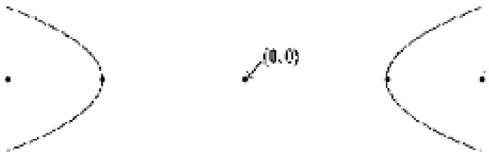
We now turn our attention to the standard equation of a hyperbola. We say that the standard equation of a hyperbola centered at the origin is given by

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

if the transverse axis is horizontal, or

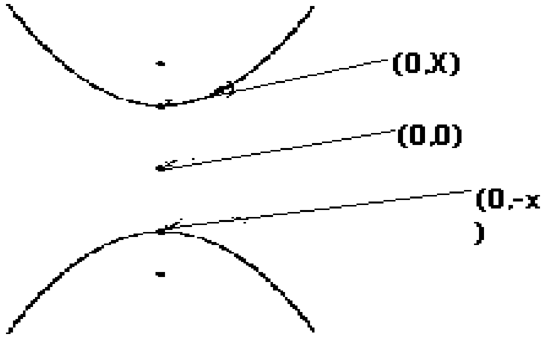
$$\frac{y^2}{a^2} - \frac{x^2}{b^2} = 1$$

if the transverse axis is vertical.



Notice a very important difference in the notation of the equation of a hyperbola compared to that of the ellipse. We see that a always corresponds to the positive term in the equation of the ellipse. The relationship of a and b does not determine the orientation of the hyperbola. (Recall that the size of a and b was used in the section on the ellipse to determine the orientation of the ellipse.) In the case of the hyperbola, the variable in the ``positive" term of the equation determines the orientation of the hyperbola. Hence, if the variable x is in the positive term of the equation, as it is in the equation

$x^2/a^{2-}y^2/b^2=1$,



$$\frac{x^2}{a^2} - \frac{y^2}{b^2} = 1,$$

then the hyperbola is oriented as follows:

If the variable **y** is in the positive term of the equation, as it is in the equation

$$\frac{y^2}{a^2} - \frac{x^2}{b^2} - 1,$$

then we see the following type of hyperbola:

Note that the vertices are always a units from the center of the hyperbola, and the distance c of the foci from the center of the hyperbola can be determined using a, b, and the following equality:

$$b^2 = c^2 - a^2$$

We will use this relationship often, so keep it in mind.

The next question you might ask is this: ``what happens to the equation if the center of the hyperbola is not (0, 0)?" As in the case of the ellipse, if the center of the hyperbola is (h, k), then the equation of the hyperbola becomes

$$\frac{(x-h)^2}{a^2} - \frac{(y-k)^2}{b^2} = 1$$

if the transverse axis is horizontal, or

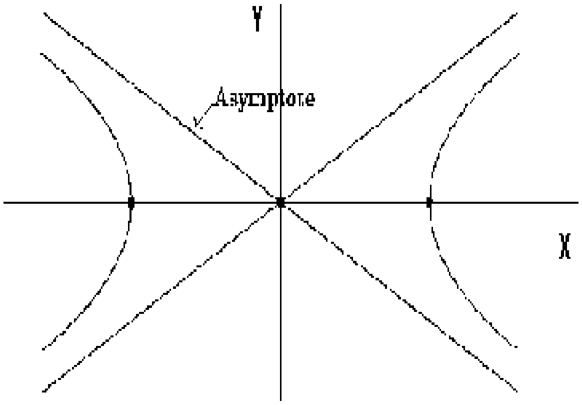
$$\frac{(y-k)^2}{a^2} - \frac{(x-h)^2}{b^2} = 1$$

if the transverse axis is vertical.

A few more terms should be mentioned here before we move to some examples. First, as in the case of an ellipse, we say that the eccentricity of a hyperbola, denoted by e, is given by

$$e = \frac{c}{a},$$

or we say that the eccentricity of a hyperbola is given by the ratio of the distance between the foci to the distance between the vertices. Now in the case of a hyperbola, the distance between the foci is greater than the distance between the vertices. Hence, in the case of a hyperbola, $e > 1$.



Recall that for the ellipse,

$$0 \leq e < 1.$$

Two final terms that we must mention are asymptotes and the conjugate axis. The two branches of a hyperbola are “bounded by” two straight lines, known as asymptotes. These asymptotes are easily drawn once one plots the vertices and the points (h, k+b) and (h, k-b) and draws the rectangle which goes through these four points. The line segment joining (h, k+b) and (h, k-b) is called the conjugate axis. The asymptotes then are simply the lines which go through the corners of the rectangle.

But what are the actual equations of these asymptotes? Note that if the hyperbola is oriented horizontally, then the corners of this rectangle have the following coordinates:

$$(h+a, k+b), (h-a, k-b),$$

and

$$(h-a, k+b), (h+a, k-b).$$

Here I have paired these points in such a way that each asymptote goes through one pair of the points. Consider the first pair of points:

$$(h+a, k+b), (h-a, k-b)$$

Given two points, we can find the equation of the unique line going through the points using the point--slope form of the line. First, let us determine the slope of our line. We find this as ``change in y over change in x" or ``rise over run". In this case, we see that this slope is equal to

$$\frac{2b}{2a}$$

or simply

$$\frac{b}{a}.$$

Then, we also know that the line goes through the center (h, k). Hence, by the point--slope form of a line, we know that the equation of this asymptote is

$$y-k=\frac{b}{a}(x-h)$$

or

$$y=k+\frac{b}{a}(x-h).$$

The other asymptote in this case has a negative slope; which is given by

$$-\frac{b}{a}.$$

Using the same argument, we see that this asymptote has equation

$$y=k-\frac{b}{a}(x-h).$$

What if the hyperbola is vertically oriented? Then one of the asymptote will go through the “corners” of the rectangle given by

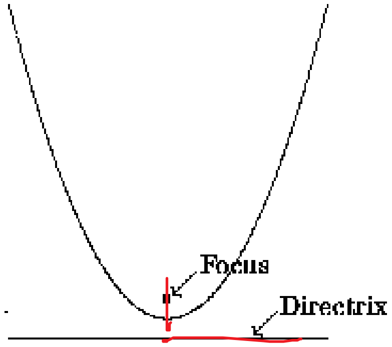
$$(h+b, k+a), (h-b, k-a).$$

Then the slope in this case will not be b/a but will be a/b. Hence, analogous to the work we just performed, we can show that the asymptotes of a vertically oriented hyperbola are determined by

$$y = k + \frac{a}{b}(x-h)$$

and

$$y = k - \frac{a}{b}(x-h).$$

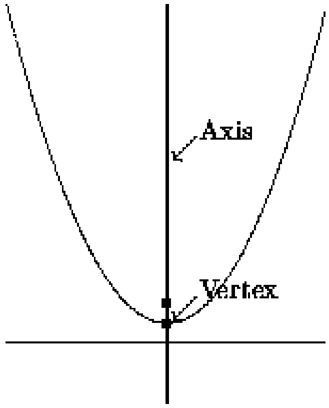


Important

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.

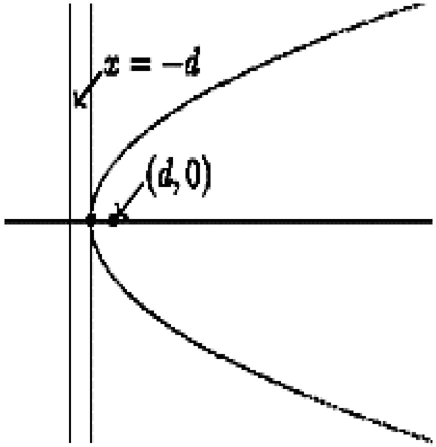
Note that the graph of a parabola is similar to one branch of a hyperbola. However, you should realize that a parabola is **not** simply one branch of a hyperbola. Indeed, the branches of a hyperbola approach linear asymptotes, while a parabola does not do so.

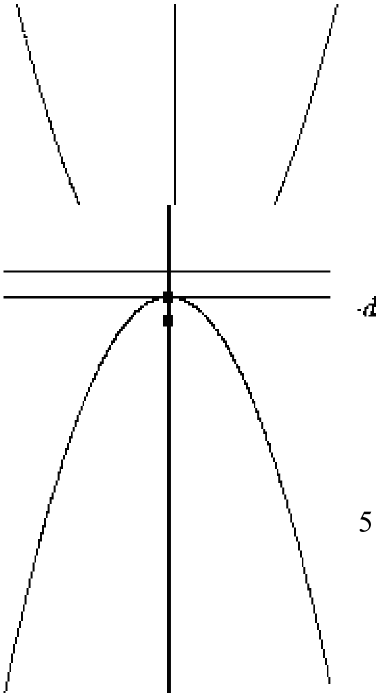


Several other terms exist which are associated with a parabola. The midpoint between the focus and directrix of the parabola is called the **vertex** and the line passing through the focus and vertex is called the **axis** of the parabola. (This is similar to the major axis of the ellipse and the transverse axis of the hyperbola.) See figure at right.

Now let's move to the standard algebraic equations for parabolas and note the four types of parabolas that exist. As we discuss the four types, you should notice the differences in the equations that are related to each of the four parabolas.

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





if

if

ε

—

n

5

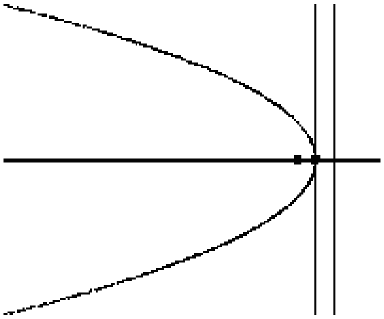
example with vertical axis and figure below for an example with horizontal axis.

Note here that we have assumed that

$$d > 0.$$

It is also the case that d could be negative, which flips the orientation of the parabola. (See Figures)

Thus, we see that there are four different orientations of parabolas, which depend on a) which variable is squared (x or y) and b) whether d is positive or negative.



One last comment before going to some examples; if the vertex of the parabola is at (h, k), then the equation of the parabola does change slightly. The equation of a parabola with vertex at (h, k) is given by

$$(x-h)^2 = 4d(y-k)$$

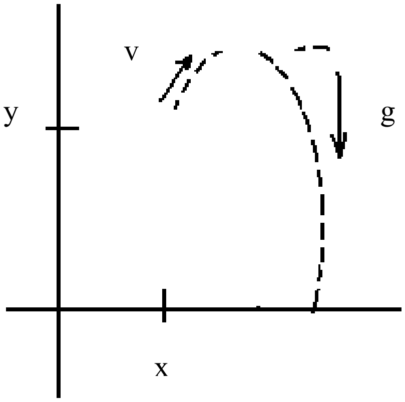
if the axis is vertical and

$$(y-k)^2 = 4d(x-h)$$

Rotation of Axes

Note that in the sections at right dealing with the ellipse, hyperbola, and the parabola, the algebraic equations that appeared did not contain a term of the form **xy**. However, in our "Algebraic View of the Conic Sections," we stated that every conic section is of the form

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



where **A**, **B**, **C**, **D**, **E**, and **F** are constants. In essence, all of the equations that we have studied have had **B=0**. So the question arises: ``what role, if any, does the **xy** term play in conic sections? If it were present, how would that change the geometric figure?"

First of all, the answer is NOT that the conic changes from one type to another. That is to say, if we introduce a **xy** term, the conic does NOT change from an ellipse to a hyperbola. If we start with the standard equation of an ellipse and insert an extra term, a **xy** term, we still have an ellipse.

So 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.

Important

Animated Applications

Ellipses, hyperbolas, and parabolas are particularly useful in certain animation applications. These curves describe orbital and other motions for objects subjected to gravitational, electromagnetic, or nuclear forces. Planetary orbits in the solar system, for example, are ellipses; and an object projected into a uniform gravitational field travels along a parabolic trajectory.

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

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

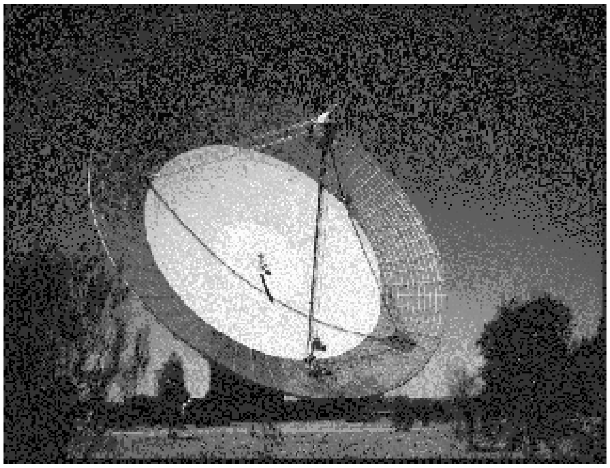
With constants a and b determined by the initial velocity v_0 of the object and the acceleration g due to the uniform gravitational force. We can also describe such parabolic motions with parametric equations using a time parameter t, measured in seconds from the initial projection point:

$$x = x_0 + v_{x0}\,t$$
$$y = y_0 + v_{y0}\,t - \frac{1}{2}\,g\,t^{2}$$

Here v_{x0} and v_{y0} are the initial velocity components, and the value of g near the surface of the earth is approximately 980 cm/ sec2. Object positions along the parabolic path are then calculated at selected time steps.

Some related real world applications are given below.

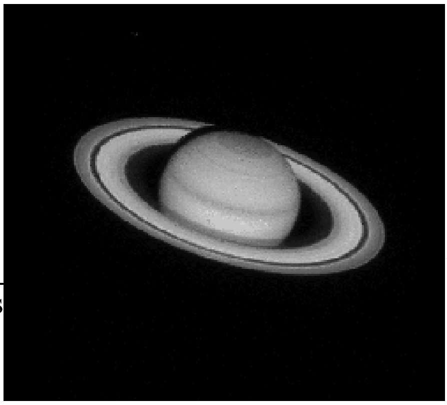
Parabolic Reflectors



One of the ``real--world" applications of parabolas involves the concept of a 3-dimensional parabolic reflector in which a parabola is revolved about its axis (the line segment joining the vertex and focus). The shape of car headlights, mirrors in reflecting telescopes, and television and radio antennae (such as the one at right) all utilize this property.

Important

In terms of a car headlight, this property is used to reflect the light rays emanating from the focus of the parabola (where the actual light bulb is located) in parallel rays.



This property is used in a converse fashion when one considers parabolic antennae. Here, all incoming rays parallel to the axis of the parabola are reflected through the focus.

Elliptical Orbits

At one time, it was thought that the planets in our solar system revolve around the sun in a circular orbit. It was later discovered, however, that the orbits are not circular, but were actually very round elliptical shapes. (Recall the discussion of the eccentricity of an ellipse mentioned at right.) The eccentricity of the orbit of the Earth around the sun is approximately 0.0167, a fairly small number. Pluto's orbit has the highest eccentricity of all the planets in our solar system at 0.2481. Still, this is not a very large value.

As a matter of fact, the sun acts as one of the foci in the ellipse. This phenomenon was first noted by Apollonius in the second century B.C. Kepler later studied this in a more rigorous fashion and developed the scientific view of planetary motion.

Important

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 and has been used by many in the design of special rooms. In particular, St. Paul's Cathedral and one of the rooms at the United States Capitol were built with this in mind.

Polynomials and Spline Curves

A polynomial function of nth degree in x is defined as

$$y = \sum_{k=0}^{n} a_k x^k$$

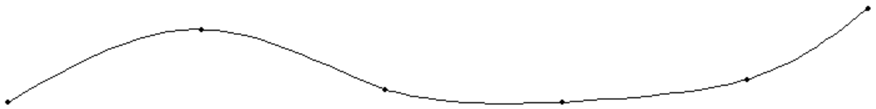
$$y = a_0 x^0 + a_1 x^1 + \text{----------------} + a_{n-1} x^{n-1} + a_n x^n$$

Where n is a nonnegative integer and the a_k are constants, with a_n not equal to 0. We get a quadratic when n = 2; a cubic polynomial when n = 3; a quadratic when n = 4; and so forth. And obviously a straight line when n = 1. Polynomials are useful in a number of graphics applications, including the design of object shapes, the specifications of animation paths, and the graphing of data trends in a discrete set of data points.

Designing object shapes or motion paths is typically done by specifying a few points to define the general curve contour, then fitting the selected points with a polynomial. One way to accomplish the curve fitting is to construct a cubic polynomial curve section between each pair of specified points. 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 to1. A curve is shown below calculated using at right equations.



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

Lecture No.8 Filled-Area Primitives-I

So far we have covered some output primitives that is drawing primitives like point, line, circle, ellipse and some other variations of curves. Also we can draw certain other shapes with the combinations of lines like triangle, rectangle, square and other polygons (we will have some discussion on polygons coming ahead). Also we can draw some shapes using mixture of lines and curves or circles etc. So we are able to draw outline/ sketch of certain models but need is there to make a solid model.
Therefore, in this section we will see what are filled area primitives and what are the different issues related to them. There are two basic approaches to area filling on raster systems. One way is to draw straight lines between the edges of polygon called scan-line polygon filling. As said earlier there are several issues related to scan line polygon, which we will discuss in detail. Second way is to start from an interior point and paint outward from this point till we reach the boundary called boundary-fill. A slight variation of this technique is used to fill an area specified by cluster (having no specific boundary). The technique is called flood-fill and having almost same strategy that is to start from an interior point and start painting outward from this point till the end of cluster.
Now having an idea we will try to see each of these one by one, starting from scan-line polygon filling.

Scan-line Polygon Fill

Before we actually start discussion on scan-line polygon filling technique, it is useful to discuss what is polygon? Besides polygon definition we will discuss the following topics one by one to have a good understanding of the concept and its implementation.

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

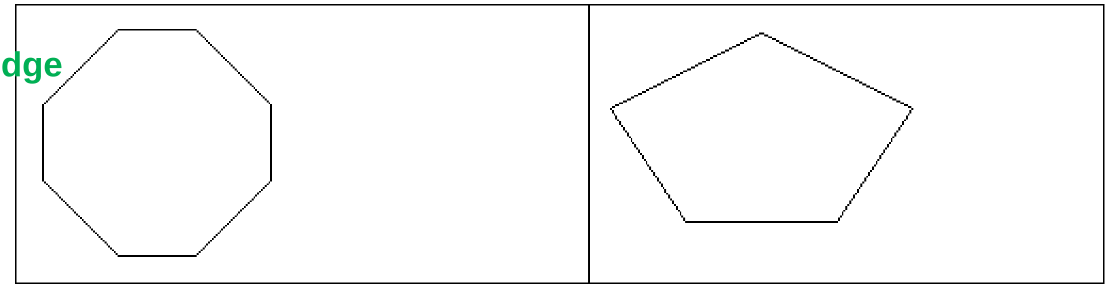
Polygon

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

I. **Convex** polygons are the simplest type of polygon to fill. To determine whether or not a polygon is convex, ask the following question:

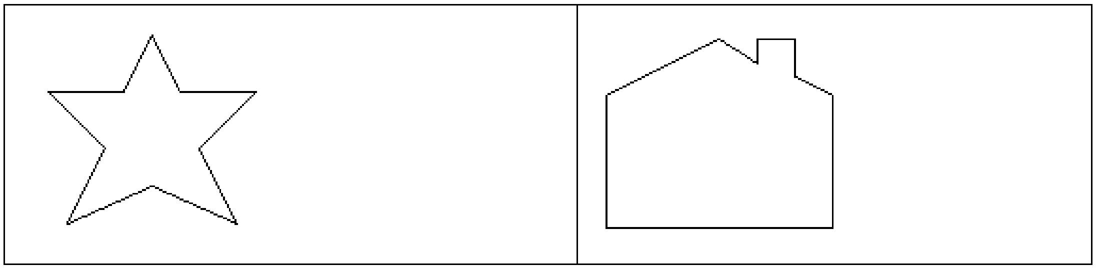
Does a straight line connecting ANY two points that are inside the polygon intersect any edges of the polygon?

in convex no edge will intersect

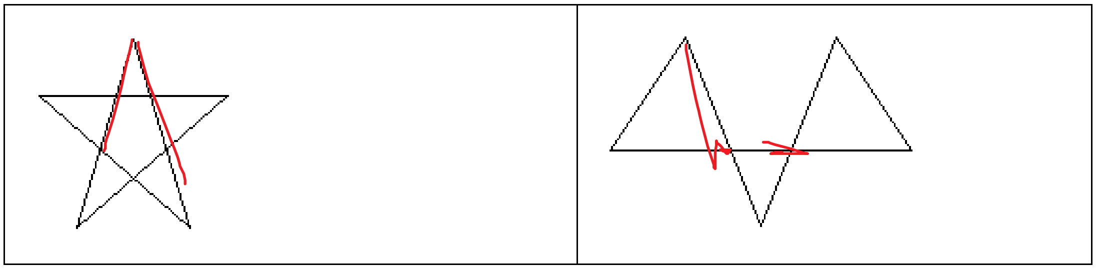


If the answer is no, the polygon is convex. This means that for any scan-line, the scan-line will cross at most two polygon edges (not counting any horizontal edges). Convex polygon edges also do not intersect each other.

II. **Concave** polygons are a superset of convex polygons, having fewer restrictions than convex polygons. The line connecting any two points that lie inside the polygon may intersect more than two edges of the polygon. Thus, more than two edges may intersect any scan line that passes through the polygon. The polygon edges may also touch each other, but they may not cross one another.



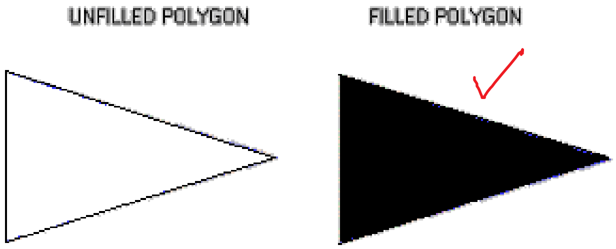
Complex polygons are just what their name suggests: complex. Complex polygons are basically concave polygons that may have self-intersecting edges. The complexity arises from distinguishing which side is inside the polygon when filling it.



Difference between Filled and Unfilled Polygon

When an unfilled polygon is rendered, only the points on the perimeter of the polygon are drawn. Examples of unfilled polygons are shown in the next page.

However, 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. Here, we deal only with solid colors. The following figure shows the difference between filled and unfilled polygons.



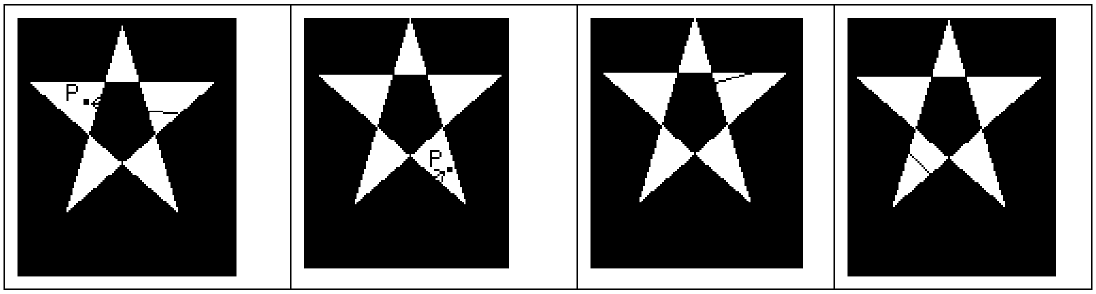
In order to determine which pixels are inside the polygon, the odd-parity rule is used within the scan-line polygon fill algorithm. This is discussed next.

Parity

What is 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.

The Underlying Principle: Conceptually, the odd parity test entails drawing a line segment from any point that lies outside the polygon to a point P that we wish to determine whether it is inside or outside of the polygon. Count the number of edges that the line crosses. If the number of polygon edges crossed is **odd**, then P lies within the polygon. Similarly, if the number of edges is even, then P lies outside of the polygon. There are special ways of counting the edges when the line crosses a vertex. This will be discussed in the algorithm section. Examples of counting parity can be seen in the following demonstration.

MCQs



Using the Odd Parity Test in the Polygon Fill Algorithm

The odd parity method creates a problem: How do we determine whether a pixel lies outside of the polygon to test for an inside one, if we cannot determine whether one lies within or outside of the polygon in the first place? If we assume our polygon lies entirely within our scene, then the edge of our drawing surface lies outside of the polygon.

Furthermore, it would not be very efficient to check each point on our drawing surface to see if it lies within the polygon and, therefore, needs to be colored.

So, we can take advantage of the fact that for each scan-line we begin with even parity; we have NOT crossed any polygon edges yet. Then as we go from left to right across our scan line, we will continue to have even parity (i.e., will not use the fill color) until we cross the first polygon edge. Now our parity has changed to odd and we will start using the fill color.

How long will we continue to use the fill color? Well, our parity won't change until we cross the next edge. Therefore, we want to color all of the pixels from when we crossed the first edge until we cross the next one. Then the parity will become even again.

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 from the first x to the second, the third to the forth and so on.

Polygon Filling

In order to fill a polygon, we do not want to have to determine the type of polygon that we are filling. The easiest way to avoid this situation is to use an algorithm that works for all three types of polygons. Since both convex and concave polygons are subsets of the complex type, using an algorithm that will work for complex polygon filling should be sufficient for all three types. The scan-line polygon fill algorithm, which employs the odd/even parity concept previously discussed, works for complex polygon filling.

Reminder: The basic concept of the scan-line algorithm is to draw points from edges of odd parity to even parity on each scan-line.

b) What is a scan-line?

A scan-line is a line of constant y value, i.e., y=c, where c lies within our drawing region, e.g., the window on our computer screen.

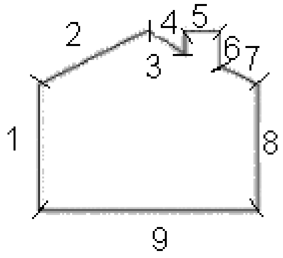
The **scan-line algorithm** is outlined next.

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. Initializing All of the Edges:

The first thing that needs to be done is determine how the polygon's vertices are related. The all_edges table will hold this information.



Each adjacent set of vertices (the first and second, second and third, similarly last and first) defines an edge. In above figure vertices are shown by small lines and edges are numbered from 1 to 9 each between successive vertices.

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

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:

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

For example, the edge values may be kept as follows, where N is equal to the total number of edges - 1 (starting from 0) and each index into the all_edges array contains a pointer to the array of edge values.

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. 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. Since we will fill the edges from bottom to top and left to right. To do this, the global edge table should be inserted with edges grouped by increasing minimum y values. Edges with the same minimum y values are sorted on minimum x values as follows:

1. Place the first edge with a slope that is not equal to zero in the global edge table.
2. If the slope of the edge is zero, do not add that edge to the global edge table.
3. 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.

Next, Increase the index to the global edge table once each time the current edge's x value is greater than and the y value is less than or equal to that of the edge at the current index in the global edge table.

If the index, at any time, is equal to the number of edges currently in the global edge table, do not increase the index.

Place the edge information for minimum y value, maximum y value, x value, and 1/m in the global edge table at the index.

The global edge table should now contain all of the edge information necessary to fill the polygon in order of increasing minimum y and x values.

3. Initializing 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. Since the global edge table is sorted, the scan-line is the minimum y value of the first entry in this table.

5. 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:

Since the global edge table is ordered on minimum y and x values, search, in order, through the global edge table and, for 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. Do this until an edge is found with a minimum y value greater than the scan line value. The active edge table will now contain ordered edges of those edges that are being filled as such:

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

Active

6. 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. Increase 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.

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.)

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

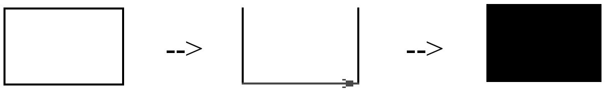
There are some special cases, the scan-line polygon fill algorithm covers these cases, but you may not understand how or why. The following will explain the handling of special cases to the algorithm.

1. Horizontal Edges:

Here we follow the minimum y value rule during scan-line polygon fill. If the edge is at the minimum y value for all edges, it is drawn. Otherwise, if the edge is at the maximum y value for any edge, we do not draw it. (See the next section containing information about top vs. bottom edges.)

This is easily done in the scan-line polygon fill implementation. Horizontal edges are removed from the edge table completely.

Question arises that how horizontal lines are are filled then? Since each horizontal line meets exactly two other edge end-points on the scan-line, the algorithm will allow a fill of the pixels between those two end-point vertices when filling on the scan-line which the horizontal line is on, if it meets the top vs. bottom edge criteria.

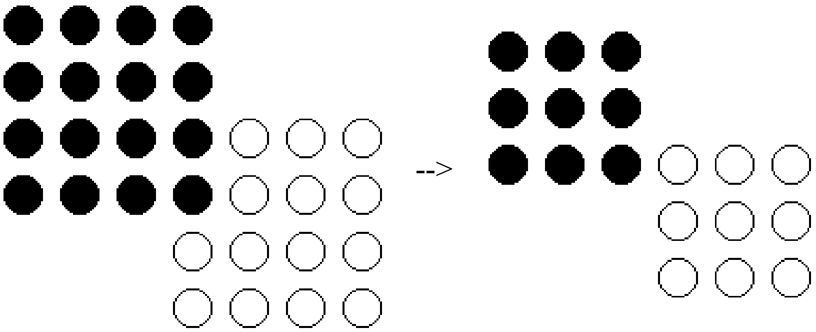


As can be seen above, if we start with a polygon with horizontal edges, we can remove the horizontal edges from the global edge table. The two endpoints of the edge will still exist and a line will be drawn between the lower edges following the scan-line polygon fill algorithm. (The blue arrowed line is indicating the scan-line for the bottom horizontal edge.)

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

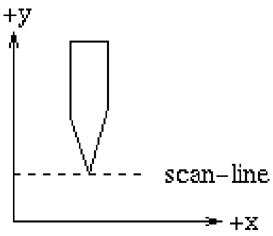
If polygons, having at least one overlapping edge the other, were filled completely from edge to edge, these polygons would appear to overlap and/or be distorted. This would be especially noticeable for polygons in which edges have limited space between them.

In order to correct for this phenomenon, our algorithm does not allow fills of the right or top edges of polygons. This distortion problem could also be corrected by not drawing either the left or right edges and not drawing either the top or bottom edges of the polygon. Either way, a consistent method should be used with all polygons. If some polygons are filled with the left and bottom edges and others with the bottom and right edges, this problem will still occur.



As can be seen above, if we remove the right and top edges from both polygons, the polygons no longer appear to be different shapes. For polygons with more overlap than just one edge, the polygons will still appear to overlap as was meant to happen.

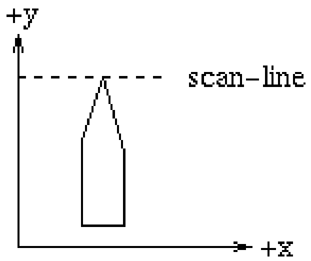
3. How do we deal with two edges meeting at a vertex when counting parity? This is a scenario which needs to be accounted for in one of the following ways:



1.

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

Essentially, the following occurs. In the scan-line polygon fill algorithm, the 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. The parity is increased once for the first edge and again for the second edge.

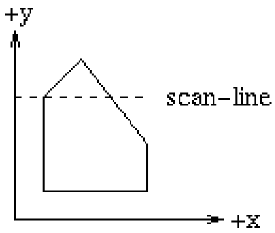


2.

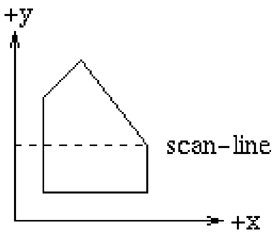
When dealing with two edges; which meet at a vertex and for both edges the vertex is the maximum point, the pixel is **not drawn** and is **counted twice for parity**.

Basically, this occurs because the vertex is not drawn for the first edge, since it is a maximum point for that edge, and parity is increased. The vertex is then not drawn for the second edge, since it is a right edge, and parity is The point should not be drawn since maximum y values for edges are not drawn in the scan-line polygon fill implementation.

3. When dealing with two edges; which meet at a vertex and for one edge the vertex is the maximum point and for the other edge the vertex is the minimum point, we must also consider whether the edges are left or right edges. Two edges meeting in such a way can be thought of as one edge; which is "bent".



If the edges are on the **left** side of the polygon, the pixel is **drawn** and is **counted once for parity** purposes. This is due to the fact that left edges are drawn in the scan-line polygon fill implementation. The vertex is drawn just once for the edge; which has this vertex as its minimum point. Parity is incremented just once for this "bent edge".



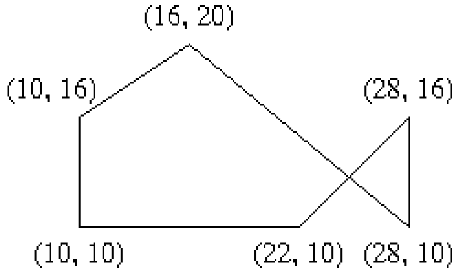
4.

If 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.

A Simple Example

Just to reiterate the algorithm, the following simple example of scan-line polygon filling will be outlined. Initially, each vertex of the polygon is given in the form of (x, y) and is in an ordered array as such:

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



Ordered Vertices

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

1. Initializing All of the Edges:

We want to determine the minimum y value, maximum y value, x value, and 1/m for each edge and keep them in the all_edges table. We determine these values for the first edge as follows:

Y-min: Since the first edge consists of the first and second vertex in the array, we use the y values of those vertices to choose the lesser y value. In this case it is 10.

Y-max: In the first edge, the greatest y value is 16.

X-val: Since the x value associated with the vertex with the highest y value is 10, 10 is the x value for this edge.

1/m: Using the given formula, we get (10-10)/ (16-10) for 1/m.

The edge value results are in the form of Y-min, Y-max, X-val, and Slope for each edge array pointed to in the all_edges table. As a result of calculating all edge values, we get the following in the all_edges table.

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

Table: All_edges

2. Initializing the Global Edge Table:

We want to place all the edges in the global edge table in increasing y and x values, as long as slope is not equal to zero. For the first edge, the slope is not zero so it is placed in the global edge table at index=0.

Index		**Y-min**	**Y-max**	**X-val**	**1/m**
0	→	10	16	10	0

Table: global

For the second edge, the slope is not zero and the minimum y value is greater than that at zero, so it is placed in the global edge table at index=1.

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

Table: global

For the third edge, the slope is not zero and the minimum y value is equal the edge's at index zero and the x value is greater than that at index 0, so the index is increased to 1. Since the third edge has a lesser minimum y value than the edge at index 2 of the global edge table, the index for the third edge is not increased again. The third edge is placed in the global edge table at index=1.

Index		**Y-min**	**Y-max**	**X-val**	**1/m**
0	→	10	16	10	0
1	→	10	20	28	-1.2
2	→	16	20	10	1.5

Table: global

We continue this process until we have the following:

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

Table: global

Notice that the global edge table has only five edges and the all_edges table has six. This is due to the fact that the last edge has a slope of zero and, therefore, is not placed in the global edge table.

3. Initializing 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

Since our scan-line value is 10, we choose all edges which have a minimum y value of 10 to move to our active edge table. This results in the following.

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

Table: active

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

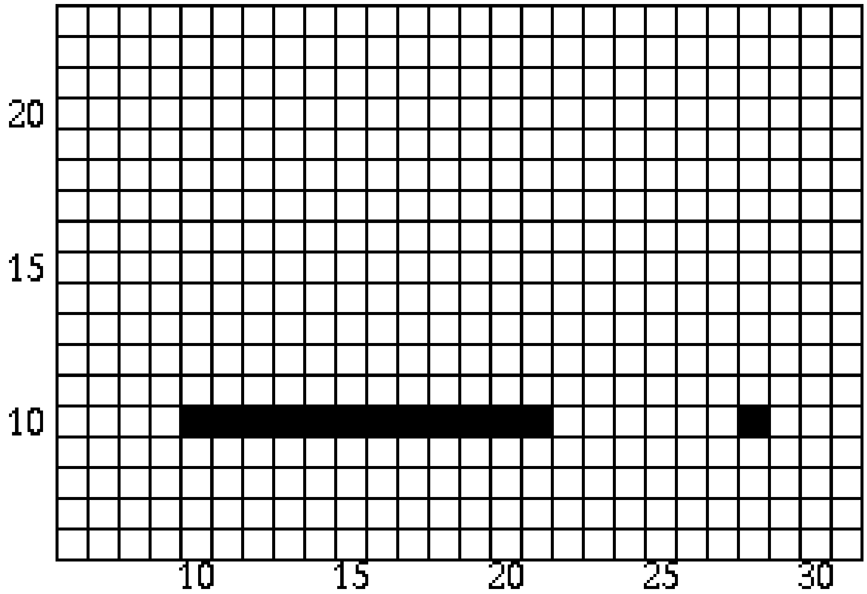
Table: global

6. Filling the Polygon

Starting at the point (0, 10), which is on our scan-line and outside of the polygon, will want to decide which points to draw for each scan-line.

1. Scan-line = 10:

Once the first edge is encountered at x=10, parity = odd. All points are drawn from this point until the next edge is encountered at x=22. Parity is then changed to even. The next edge is reached at x=28 and the point is drawn once on this scan-line due to the special parity case. We are now done with this scan-line. The polygon is now filled as follows:



First, we update the x values in the active edge table using the formula $x_1 = x_0 + 1/m$ to get the following:

Index	**Y-max**	**X-val**	**1/m**		**Index**		**Y-min**	**Y-max**	**X-val**	**1/m**
0 →	16	10	0		**0** →		16	20	10	1.5
1 →	16	23	1							
2 →	16	28	0							
3 →	20	26.8	-1.2							

Table: active **Table: global**

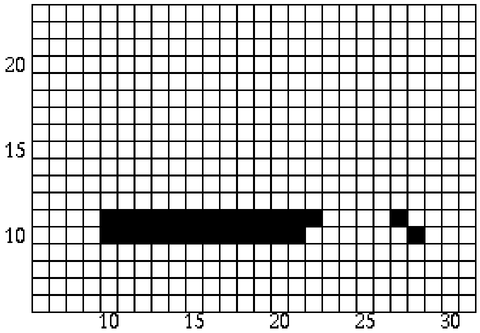
The edges then need to be reordered since the edge at index 3 of the active edge table has a lesser x value than that of the edge at index 2. Upon reordering, we get:

Index	**Y-max**	**X-val**	**1/m**		**Index**		**Y-min**	**Y-max**	**X-val**	**1/m**
0 →	16	10	0		**0** →		16	20	10	1.5
1 →	16	23	1							
2 →	16	26.8	-1.2							
3 →	20	28	0							

Table: active **Table: global**

2. Scan-line = 11:

Once the first edge is encountered at x=10, parity = odd. All points are drawn from this point until the next edge is encountered at x=23. Parity is then changed to even. The next edge is reached at x=27 and parity is changed to odd. The points are then drawn until the next edge is reached at x=28. We are now done with this scan-line. The polygon is now filled as follows:



Upon updating the x values, the edge tables are as follows:

Index		Y-max	X-val	1/m	
0	→	16	10	0	
1	→	16	24	1	
2	→	16	25.6	-1.2	
3	→	20	28	0	

Table: active

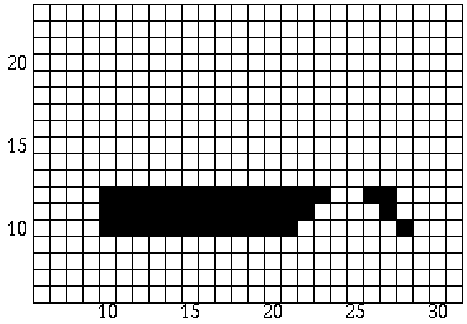
Index		Y-min	Y-max	X-val	1/m
0	→	16	20	10	1.5

Table: global

It can be seen that no reordering of edges is needed at this time.

3. Scan-line = 12:

Once the first edge is encountered at x=10, parity = odd. All points are drawn from this point until the next edge is encountered at x=24. Parity is then changed to even. The next edge is reached at x=26 and parity is changed to odd. The points are then drawn until the next edge is reached at x=28. We are now done with this scan-line. The polygon is now filled as follows:



Updating the x values in the active edge table gives us:

Index		Y-max	X-val	1/m	
0	→	16	10	0	
1	→	16	25	1	
2	→	16	24.4	-1.2	
3	→	20	28	0	

Table: active

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

Table: global

We can see that the active edges need to be reordered since the x value of 24.4 at index 2 is less than the x value of 25 at index 1. Reordering produces the following:

Index		Y-max	X-val	1/m	
0	→	16	10	0	
1	→	16	24.4	1	
2	→	16	25	0	
3	→	20	28	-1.2	

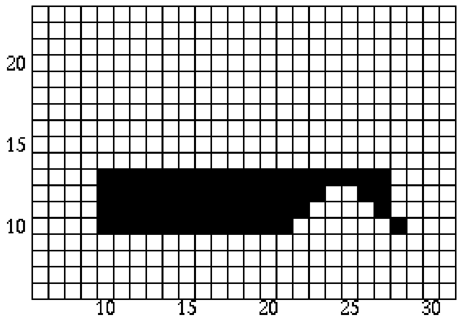
Table: active

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

Table: global

4. Scan-line = 13:

Once the first edge is encountered at x=10, parity = odd. All points are drawn from this point until the next edge is encountered at x=25 Parity is then changed to even. The next edge is reached at x=25 and parity is changed to odd. The points are then drawn until the next edge is reached at x=28. We are now done with this scan-line. The polygon is now filled as follows:



Upon updating the x values for the active edge table, we can see that the edges do not need to be reordered.

Index		Y-max	X-val	1/m	
0	→	16	10	0	
1	→	16	23.2	1	
2	→	16	26	0	
3	→	20	28	-1.2	

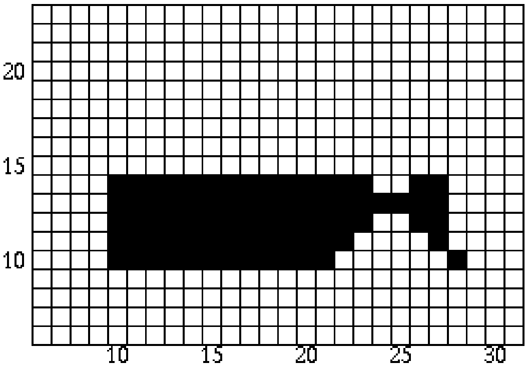
Table: active

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

Table: global

5. Scan-line = 14:

Once the first edge is encountered at x=10, parity = odd. All points are drawn from this point until the next edge is encountered at x=24. Parity is then changed to even. The next edge is reached at x=26 and parity is changed to odd. The points are then drawn until the next edge is reached at x=28. We are now done with this scan-line. The polygon is now filled as follows:



Upon updating the x values for the active edge table, we can see that the edges still do not need to be reordered.

Index	Y-max	X-val	1/m	
0 →	16	10	0	
1 →	16	22	-1.2	
2 →	16	27	1	
3 →	20	28	0	

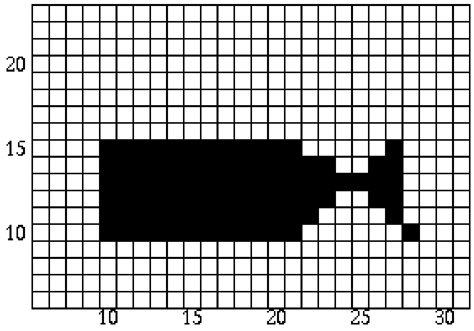
Table: active

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

Table: global

6. Scan-line = 15:

Once the first edge is encountered at x=10, parity = odd. All points are drawn from this point until the next edge is encountered at x=22. Parity is then changed to even. The next edge is reached at x=27 and parity is changed to odd. The points are then drawn until the next edge is reached at x=28. We are now done with this scan-line. The polygon is now filled as follows:



Since the maximum y value is equal to the next scan-line for the edges at indices 0, 2, and 3, we remove them from the active edge table. This leaves us with the following:

Index	Y-max	X-val	1/m
0	20	22	-1.2

Table: active

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

Table: global

We then need to update the x values for all remaining edges.

Index	Y-max	X-val	1/m
0	20	20.8	-1.2

Table: active

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

Table: global

Now we can add the last edge from the global edge table to the active edge table since its minimum y value is equal to the next scan-line. The active edge table now looks as follows (the global edge table is now empty):

Index	Y-max	X-val	1/m
0	20	20.8	-1.2
1	20	10	1.5

Table: active

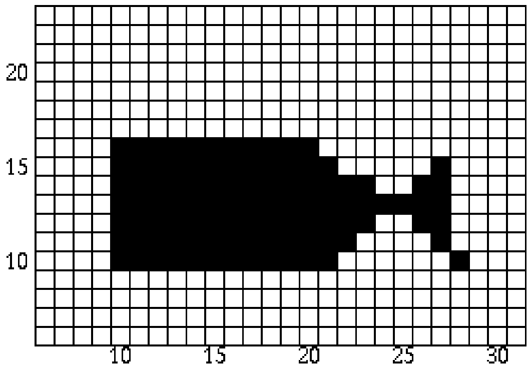
These edges obviously need to be reordered. After reordering, the active edge table contains the following:

Index	Y-max	X-val	1/m
0	20	10	1.5
1	20	20.8	-1.2

Table: active

7. Scan-line = 16:

Once the first edge is encountered at x=10, parity = odd. All points are drawn from this point until the next edge is reached at x=21. We are now done with this scan-line. The polygon is now filled as follows:



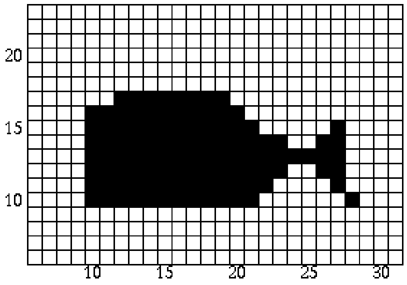
The x values are updated and the following is obtained:

Index	Y-max	X-val	1/m
0	20	11.5	1.5
1	20	19.6	-1.2

Table: active

8. Scan-line = 17:

Once the first edge is encountered at x=12, parity = odd. All points are drawn from this point until the next edge is reached at x=20. We are now done with this scan-line. The polygon is now filled as follows:



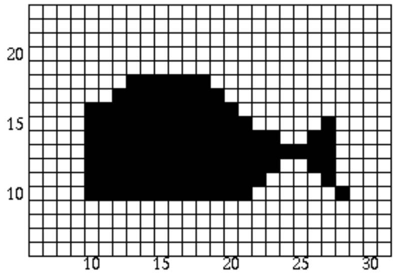
We update the x values and obtain:

Index	Y-max	X-val	1/m
0	20	13	1.5
1	20	18.4	-1.2

Table: active

9. Scan-line = 18:

Once the first edge is encountered at x=13, parity = odd. All points are drawn from this point until the next edge is reached at x=19. We are now done with this scan-line. The polygon is now filled as follows:



Upon updating the x values we get:

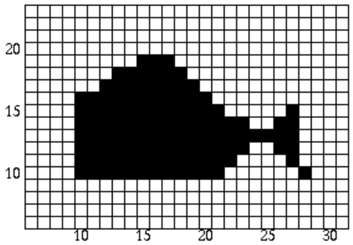
Index	Y-max	X-val	1/m
0	20	14.5	1.5
1	20	17.2	-1.2

Table: active

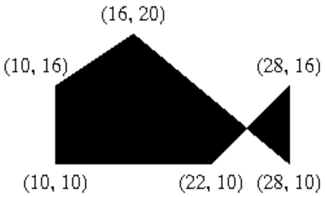
10. Scan-line = 19:

Once the first edge is encountered at x=15, parity = odd. All points are drawn from this point until the next edge is reached at x=18. We are now done with this scan-line. Since the maximum y value for both edges in the active edge table is equal to the next scan-line, we remove them. The active edge table is now empty and we are now done.

The polygon is now filled as follows:



Now that we have filled the polygon, let's see what it looks like to the human eye:



Lecture No.9 Filled-Area Primitives-II

Boundary fill

Another important class of area-filling algorithms starts at a point known to be inside a figure and starts filling in the figure outward from the point. Using these algorithms a graphic artist may sketch the outline of a figure and then select a color or pattern with which to fill it. The actual filling process begins when a point inside the figure is selected. These routines are like the *paint-scan function* seen in common interactive paint packages.

The first such method that we will discuss is called the *boundary-fill algorithm.* The boundary-fill method requires the coordinates of a starting point, a fill color, and a boundary color as arguments.

Boundary fill algorithm: Important

The Boundary fill algorithm performs the following steps:

Check the pixel for boundary color

Check the pixel for fill color

Set the pixel in fill color

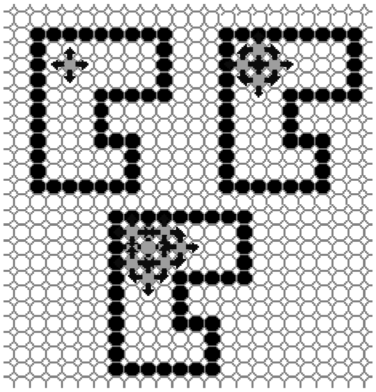
Run the process for neighbors

The pseudo code for Boundary fill algorithm can be written as:

```
boundaryFill (x, y, fillColor, boundaryColor)
        if ((x < 0) || (x >= width))
                                    return
        if ((y < 0) || (y >= height))
                                    return
        current = GetPixel(x, y)
        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)
```

Note that this is a **recursive routine**. Each invocation of *boundaryFill ()* may call itself four more times.

The logic of this routine is very simple. If we are not either on a boundary or already filled we first fill our point, and then tell our neighbors to fill themselves.



Process of Boundary Fill Algorithm

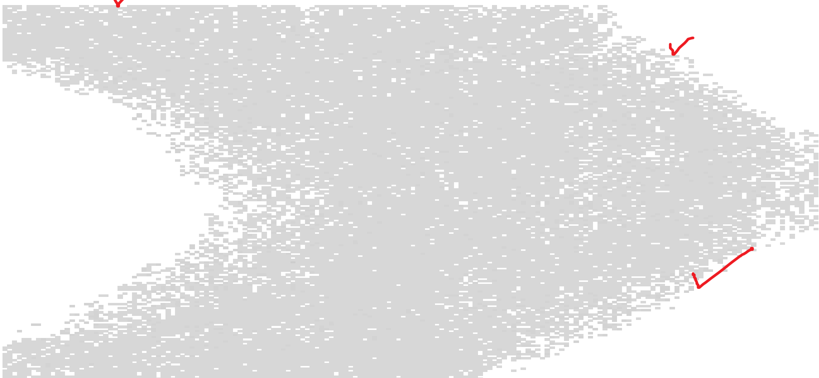
By the way, sometimes the boundary fill algorithm doesn't work. Can you think of such a case?

Flood Fill

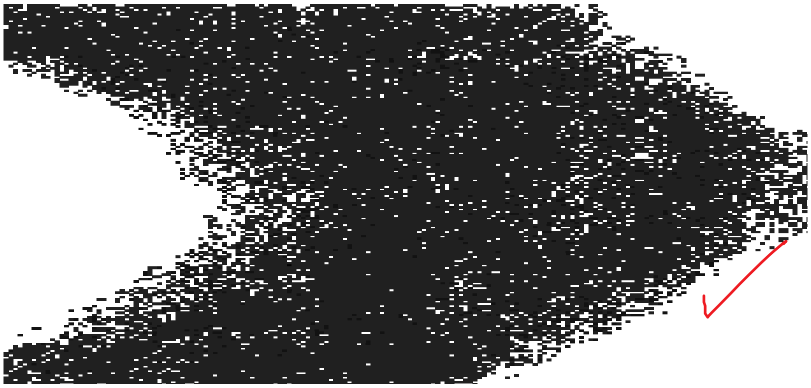
Sometimes we need an area fill algorithm that replaces all *connected* pixels of a selected color with a fill color.
The ***flood-fill algorithm*** does exactly that.

Flood-fill algorithm

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



Before Applying Flood-fill algorithm (Light color)



After Applying Flood-fill algorithm (Dark color)

Flood-fill algorithm in action

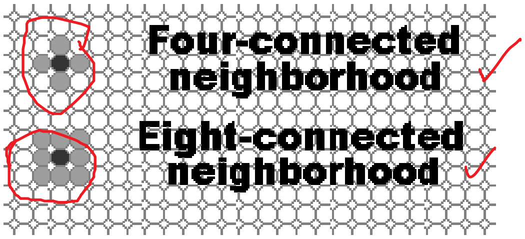
The pseudo code for Flood fill algorithm can be written as:

```
  public void floodFill(x, y, fillColor, oldColor)

      if ((x < 0) || (x >= width))
                                          return
      if ((y < 0) || (y >= height))
                                          return
      if ( getPixel (x, y) == oldColor)

          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)
```

It's a little awkward to kick off a flood fill algorithm because it requires that the old color must be read before it is invoked. The following implementation overcomes this limitation, and it is also somewhat faster, a little bit longer. The additional speed comes from only pushing three directions onto the stack each time instead of four.

```
  fillFast (x, y, fillColor)
      if ((x < 0) || (x >=width)) return
      if ((y < 0) || (y >=height)) return
      int oldColor = getPixel (x, y)
      if ( oldColor == fill ) return
      setPixel (fillColor, x, y)
      fillEast (x+1, y, fillColor, oldColor)
      fillSouth (x, y+1, fillColor, oldColor)
      fillWest (x-1, y, fillColor, oldColor)
      fillNorth (x, y-1, fillColor, oldColor)

  fillEast (x, y, fillColor, oldColor)
      if (x >= width) return
      if ( getPixel(x, y) == oldColor)
          setPixel( fillColor, x, y)
          fillEast (x+1, y, fillColor, oldColor)
          fillSouth (x, y+1, fillColor, oldColor)
          fillNorth (x, y-1, fillColor, oldColor)

  fillSouth(x, y, fillColor, oldColor)
      if (y >=height) return
      if (getPixel (x, y) == oldColor)
          setPixel (fillColor, x, y)
          fillEast (x+1, y, fillColor, oldColor)
          fillSouth (x, y+1, fillColor, oldColor)
          fillWest (x-1, y, fillColor, oldColor)

  fillWest(x, y, fillColor, oldColor)
  {
      if (x < 0) return
      if (getPixel (x, y) == oldColor)
          setPixel (fillColor, x, y)
          fillSouth (x, y+1, fillColor, oldColor)
          fillWest (x-1, y, fillColor, oldColor)
          fillNorth (x, y-1, fillColor, oldColor)

  fillNorth (x, y, fill, old)
      if (y < 0) return
      if (getPixel (x, y) == oldColor)
          setPixel (fill, x, y)
          fillEast (x+1, y, fillColor, oldColor)
          fillWest (x-1, y, fillColor, oldColor)
          fillNorth (x, y-1, fillColor, oldColor)
```

A final consideration when writing an area-fill algorithm is the size and connectivity of the neighborhood around a given pixel.



The eight-connected neighborhood is able to get into nooks and crannies that an algorithm based on a four-connected neighborhood cannot.

Here's the code for an ***eight-connected flood fill***.

```
floodFill8 (x, y, fill, old)
    if ((x < 0) || (x >=width)) return
    if ((y < 0) || (y >=height)) return
    if (getPixel (x, y) == oldColor)
        setPixel (fill, x, y);
        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)
```

Important Lecture No.10 Mathematics Fundamentals

Matrices and Simple Matrix Operations

In many fields matrices are used to represent objects and operations on those objects. In computer graphics matrices are heavily used especially their major role is in case of transformations (we will discuss in very next lecture), but not only transformation there are many areas where we use matrices and we will see in what way matrices help us. Anyhow today we are going to discuss matrix and their operation so that we will not face any problem using matrices in coming lectures and in later lectures. Today we will cover following 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 of Matrix

A matrix is a collection of numbers arranged into a fixed number of rows and columns. Usually the numbers are real numbers. In general, matrices can contain complex numbers but we won't see those here. Here is an example of a matrix with three rows and three columns:

$$\begin{array}{cc} & \begin{array}{ccc}\text{col 1 ......} & & \end{array} \\ \begin{array}{c}\text{row 1 ......} \\ \\ \\ \end{array} & \begin{pmatrix} 1 & -2 & 3 \\ 0 & 8 & 4.6 \\ 4 & -1 & 0 \end{pmatrix} \end{array}$$

The top row is row 1. The leftmost column is column 1. This matrix is a 3x3 matrix because it has three rows and three columns. In describing matrices, the format is:

rows X columns

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

The upper left corner of the matrix is row 1 column 1. In the above matrix the element at row 1 column 1 is the value 1. The element at row 2 columns 3 is the value 4.6.

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:

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

Sometimes the dimensions are written off to the side of the matrix, as in the above matrix. But this is just a little reminder and not actually part of the matrix. Here is a matrix with different dimensions. It has two rows and three columns. This is a different "data type" than the previous matrix.

$$\begin{pmatrix} 5.12 \\ -4.08 \\ 0.0 \\ 1.0 \end{pmatrix}_{4x1}$$

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

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

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

Answer: The number of rows == the number of 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.

A **column matrix** consists of a single column. It is a N x 1 matrix. These notes, and most computer graphics texts, use column matrices to represent geometrical vectors. At left is a 4 x 1 column matrix. A **row matrix** consists of a single row.

A column matrix is also called **column vector** and call a row matrix a **row vector**.

Question: What are square matrices used for?

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

Names for Matrices

Try to remember that matrix starts from rows never from columns so if order of matrix is 3*2 that means there are three rows and two columns. A matrix can be given a name. In printed text, the name for a matrix is 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 $\mathbf{B}_{3x3}$. The elements of a matrix also have names, usually a lowercase letter the same as the matrix name, with the position of the element written as a subscript. So, for example, the 3x3 matrix **A** might be written as:

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

Sometimes you write **A** = [a_{ij}] to say that the elements of matrix **A** are named a_{ij}.
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.

same dimensions
same values
same positions then matrix are equal

Matrix Equality

For two matrices to be equal, they must have
The same dimensions.
Corresponding elements must be equal.
In other words, say that $\mathbf{A}_{n \times m} = [a_{ij}]$ and that $\mathbf{B}_{p \times q} = [b_{ij}]$.
Then **A** = **B** if and only if n=p, m=q, and $a_{ij}=b_{ij}$ for all i and j in range.

$$\begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix} =?= \begin{pmatrix} 6 & 4 \\ 5 & 2 \\ 1 & 3 \end{pmatrix}$$

Here are two matrices which are not equal even though they have the same elements.

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

Matrix Addition

If two matrices have the same number of rows and same number of columns, then the *matrix sum* can be computed:

If **A** is an MxN matrix, and **B** is also an MxN matrix, then their sum is an MxN matrix formed by adding corresponding elements of **A** and **B**
Here is an example of this:

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

Of course, in most practical situations the elements of the matrices are real numbers with decimal fractions, not the small integers often used in examples.

Question: What 3x2 matrix could be added to a second 3x2 matrix without changing that second matrix?

Answer: The 3x2 matrix that has all its elements zero.

Zero Matrix

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

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

The name of a zero matrix is a boldface zero: **0**, although sometimes people forget to make it bold face. Here is an interesting problem:

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

Question: Form the above sum. No electronic calculators allowed!
Answer: Of course, the sum is the same as the non-zero matrix.

Rules for Matrix Addition

You should be happy with the following rules of matrix addition. In each rule, the matrices are assumed to all have the same dimensions.

A + **B** = **B** + **A**
A + **0** = **0** + **A** = **A**
0 + **0** = **0**

These look the same as some rules for addition of real numbers. (**Warning!!** Not all rules for matrix math look the same as for real number math.)
The first rule says that matrix addition is *commutative*. This is because ordinary addition is being done on the corresponding elements of the two matrices, and ordinary (real) addition is commutative:

$$\begin{pmatrix} 1 & 3 & 5 \\ 7 & 9 & 11 \\ 13 & 15 & 17 \end{pmatrix} + \begin{pmatrix} 0 & 2 & 4 \\ 6 & 8 & 10 \\ 12 & 14 & 16 \end{pmatrix} = \begin{pmatrix} 1+0 & 3+2 & 5+4 \\ 7+6 & 9+8 & 11+10 \\ 13+12 & 15+14 & 17+16 \end{pmatrix}$$

$$= \begin{pmatrix} 0+1 & 2+3 & 4+5 \\ 6+7 & 8+9 & 10+11 \\ 12+13 & 14+15 & 16+17 \end{pmatrix} = \begin{pmatrix} 0 & 2 & 4 \\ 6 & 8 & 10 \\ 12 & 14 & 16 \end{pmatrix} + \begin{pmatrix} 1 & 3 & 5 \\ 7 & 9 & 11 \\ 13 & 15 & 17 \end{pmatrix}$$

Question: Do you think that (**A** + **B**) + **C** = **A** + (**B** + **C**)
Answer: Yes — this is another rule that works like real number math.

Practice with Matrix Addition

Here is another matrix addition problem. Mentally form the sum (or use a scrap of paper):

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

Hint: this problem is not as tedious as it might at first seem.
Question: What is the sum?
Answer: Each element of the 3x3 result is 10.

Multiplication of a Matrix by a Scalar

A matrix can be multiplied by a scalar (by a real number) as follows:
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.)

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

Question: Show the result if the scalar *a* in the above is the value -1.
Answer: Each element in the result is the negative of the original, as seen below.

Negative of a Matrix

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

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

So, for example:

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

It will not surprise you that **A** + **(-A)** = **0**
Question: Look at the above fact. Can you think of a way to define *matrix subtraction*?
Answer: It seems like subtraction could be defined as adding a negation of a matrix.

Matrix Subtraction

If **A** and **B** have the same number of rows and columns, then **A** - **B** is defined as **A** + **(-B)**.
Usually you think of this as:
To form **A** - **B**, from each element of **A** subtract the corresponding element of **B**.
Here is a partly finished example:

Important

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

Notice in particular the elements in the first row of the answer. The way the result was calculated for the elements in row 1 column 2 is sometimes confusion.
Question: Mentally fill in the two question marks.
Answer:

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

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). Here is a matrix and its transpose:

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

The superscript "T" means "transpose". Another way to look at the transpose is that the element at row r column c if the original is placed at row c column r of the transpose. We will usually work with square matrices, and it is usually square matrices that will be transposed. However, non-square matrices can be transposed, as well:

$$\begin{pmatrix} 5 & 4 \\ 4 & 0 \\ 7 & 10 \\ -1 & 8 \end{pmatrix}^{T}_{4\times 2} = \begin{pmatrix} 5 & 4 & 7 & -1 \\ 4 & 0 & 10 & 8 \end{pmatrix}_{2\times 4}$$

Question: What is the transpose of:

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

Answer:

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

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 **$(A^{T})^{T} = A$.**

Question: What is the transpose of:

$$\begin{pmatrix} 1 & 2 & 3 & 4 \end{pmatrix}^{T} = ?$$

Answer:

$$\begin{pmatrix} 1 & 2 & 3 & 4 \end{pmatrix}^{I} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}$$

The transpose of a row matrix is a column matrix. And the transpose of a column matrix is a row matrix.

✓**Rule Summary**

Here are some rules that cover what has been discussed. You should check that they seem reasonable, rather than memorize them. For each rule the matrices have the same number of rows and columns. long question in 2023 ✓

$\mathbf{A} + \mathbf{0} = \mathbf{A}$?	$\mathbf{A} + \mathbf{B} = \mathbf{B} + \mathbf{A}$?	$\mathbf{0} + \mathbf{0} = \mathbf{0}$
$\mathbf{A} + (\mathbf{B} + \mathbf{C}) = (\mathbf{A} + \mathbf{B}) + \mathbf{C}$	$(ab)\mathbf{A} = a(b\mathbf{A})$	$a(\mathbf{A} + \mathbf{B}) = a\mathbf{B} + a\mathbf{A}$
$a\mathbf{0} = \mathbf{0}$?	$(-1)\mathbf{A} = -\mathbf{A}$ ✓	$\mathbf{A} - \mathbf{A} = \mathbf{0}$ ✓
$(\mathbf{A}^T)^T = \mathbf{A}$?	$\mathbf{0}^T = \mathbf{0}$	

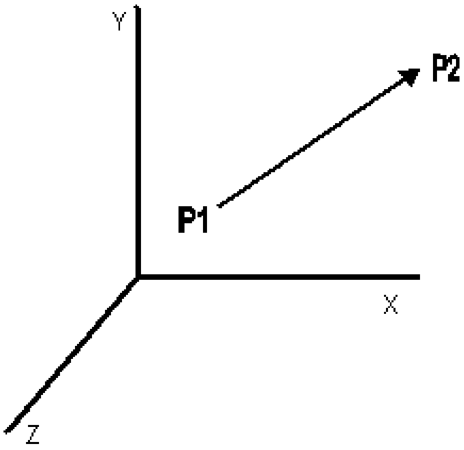
In the above, a and b are scalars (real numbers). **A** and **B** are matrices, and **0** is the zero matrix of appropriate dimension.

Question: If **A** = **B** and **B** = **C**, then does **A** = **C**?

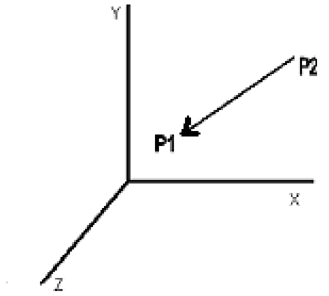
Answer: Yes A=B=C

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



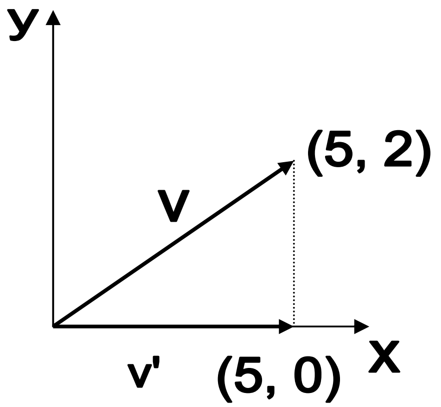
and if $P_2 = (x_2, y_2, z_2)$ is the starting point and $P1 = (x_1, y_1, z_1)$ is the ending point, then the vector $V = (x_1 - x_2, y_1 - y_2, z_1 - z_2)$



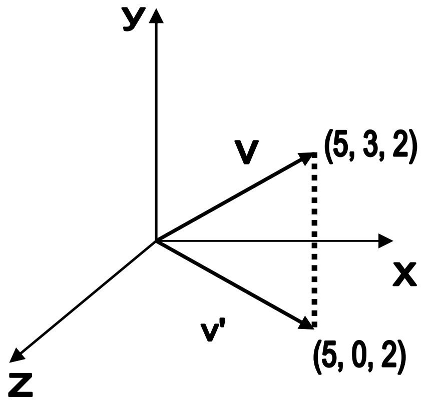
This just defines *length and direction*, but *not position.*

Vector Projections

Projection of v onto the x-axis



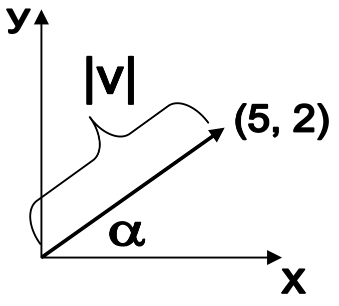
Projection of v onto the xz plane



2D Magnitude and Direction

The magnitude (length) of a vector:

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



The equation is derived from the Pythagorean theorem.

The direction of a vector:

MCQs $\tan \alpha = V_y / V_x$

$\alpha = \tan^{-1} (V_y / V_x)$ ✓

Where α 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. Particularly it needs 2 angles to fully describe direction. Latitude/ longitude is a real-world example.

Direction Cosines are often used:

- α, β, and γ are the positive angles that the vector makes with each positive coordinate axes x, y, and z, respectively

$\cos \alpha = V_x / |V|$
$\cos \beta = V_y / |V|$
$\cos \gamma = V_z / |V|$

Important

Vector Normalization ✓

"Normalizing" a vector means shrinking or stretching it so its magnitude is 1. A simple way is normalize by dividing by its magnitude:

$|V|$

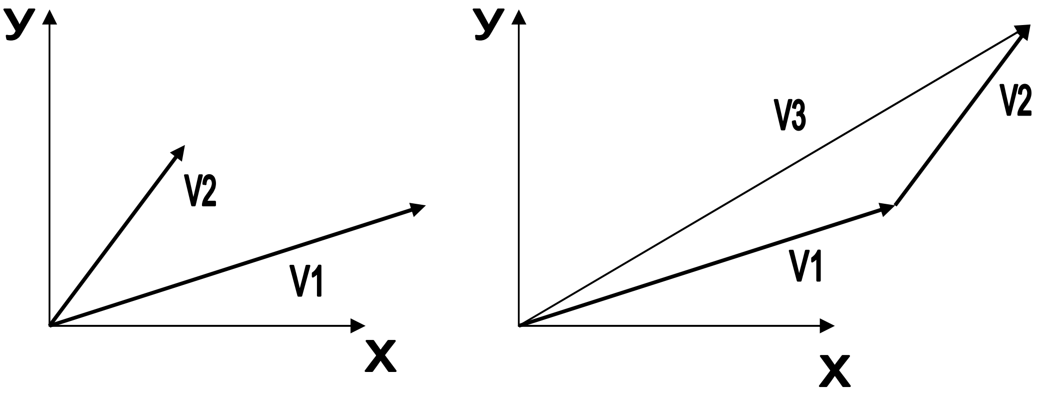
V = (1, 2, 3)
$|V| = \text{sqrt}(1^2 + 2^2 + 3^2) = \text{sqrt}(14) + 3.74$ ✓
$V_{norm} = V / |V| = (1, 2, 3) / 3.74 =$
(1 / 3.74, 2 / 3.74, 3 / 3.74) = (.27, .53, .80) ✓

$|V| = \sqrt{x^2 + y^2}$

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

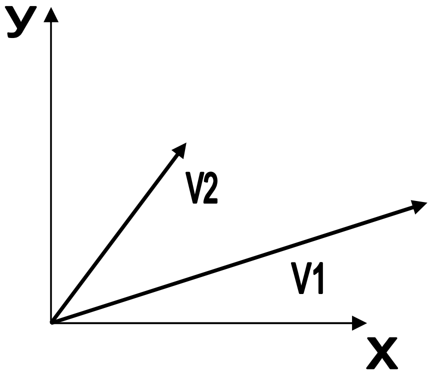
Note that the last calculation doesn't come out to exactly 1. This is because of the error introduced by using only 2 decimal places in the calculations above.

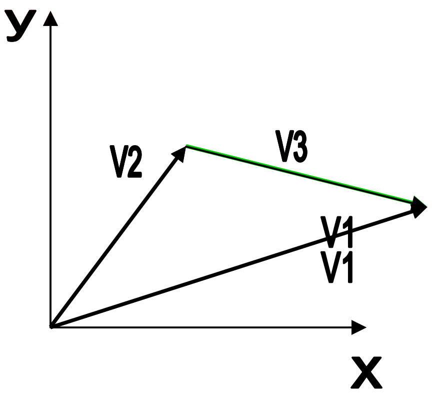
Vector Addition
Equation:
$V_3 = V_1 + V_2 = (V_{1x} + V_{2x} , V_{1y} + V_{2y} , V_{1z} + V_{2z})$



Vector Subtraction

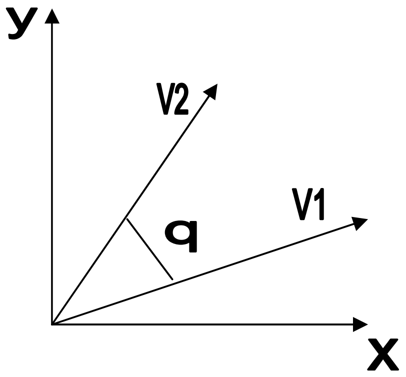
Equation:
$V_3 = V_1 - V_2 = (V_{1x} - V_{2x} , V_{1y} - V_{2y} , V_{1z} - V_{2z})$





Dot Product

The dot product of 2 vectors is a scalar ✓ MCQs



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

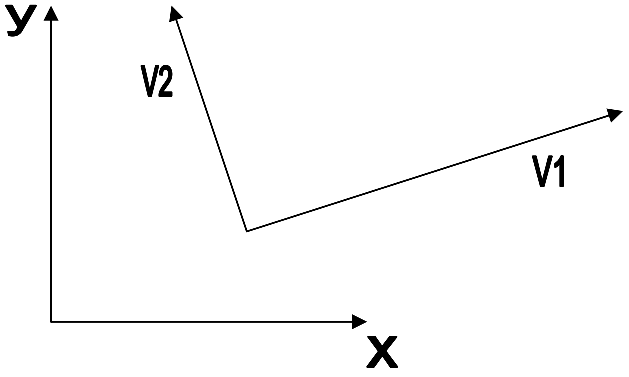
Or, perhaps more importantly for graphics:

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

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

Why is dot product important for graphics?

It is zero if and only if 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 sqrts.

Cross Product

The cross product of 2 vectors is a vector ✓

$$\begin{aligned} V_1 {}^{x} V_2 = (& V_{1y} V_{2z} - V_{1z} V_{2y}, \\ & V_{1z} V_{2x} - V_{1x} V_{2z}, \\ & V_{1x} V_{2y} - V_{1y} V_{2x}) \end{aligned}$$

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

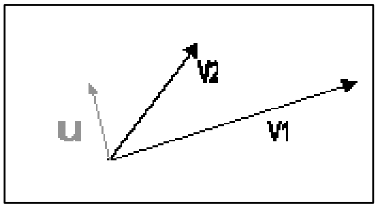
Again, just as with the dot product, there is a more graphical definition:

$V_1 {}^{x} V_2 = u\,|V_1|\ |V_2| \sin(\theta)$

where θ is the angle between the 2 vectors and θ is in the range $0 \leq \theta \leq \Pi$ and u is the unit vector that is perpendicular to both vectors

Why u?

$|V_1|\ |V_2|\ \sin(\theta)$ 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 à 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.

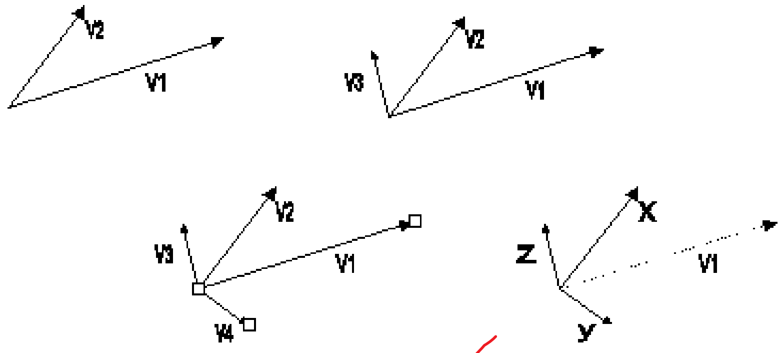
1) Cross V1 and V2 to form V3.

V_3 is now perpendicular to both V_1 and V_2

2) Cross V2 and V3 to form V4

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 ✓

Lecture No.11 2D Transformations I

In the previous lectures so far we have discussed output primitive as well as filling primitives. With the help of them we can draw an attractive 2D drawing but that will be static whereas in most of the cases we require moving pictures for example games, animation, and different model; where we show certain objects moving or rotating or changing their size.

Therefore, changes in orientation that is displacement, rotation or change in size is called geometric transformation. Here, we have certain basic transformations and some special transformation. We start with basic transformation.

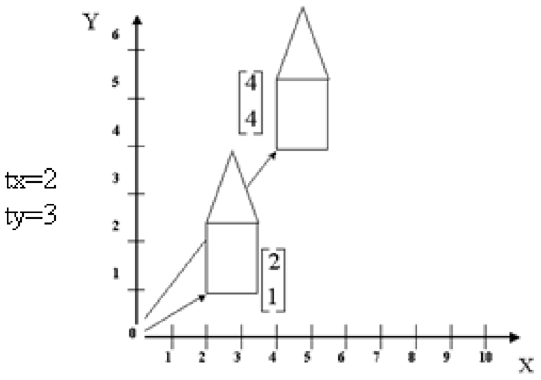
Basic Transformations *Important*

Translation
Rotation
Scaling

Above are three basic transformations. Where translation is independent of others whereas rotation and scaling depends on translation in most of cases. We will see how in their respective sections but here we will start with translation.

Translation

A translation is displacement from original place. This displacement happens to be along a straight line; where two distances involves one is along x-axis that is t_x and second is along y-axis that is t_y. The same is shown in the figure also we can express it with following equation as well as by matrix:



$x' = x + t_x$, $y' = y + t_y$

Here (t_x, t_y) is translation vector or shift vector. We can express above equations as a single matrix equation by using column vectors to represent coordinate positions and the translation vector:

$$P' = P + T$$

$$\text{Where } P = \begin{bmatrix} x \\ y \end{bmatrix} \quad P' = \begin{bmatrix} x' \\ y' \end{bmatrix} \quad T = \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

Translation is a rigid-body transformation that moves objects without deformation. That is, every point on the object is translated by the same amount.

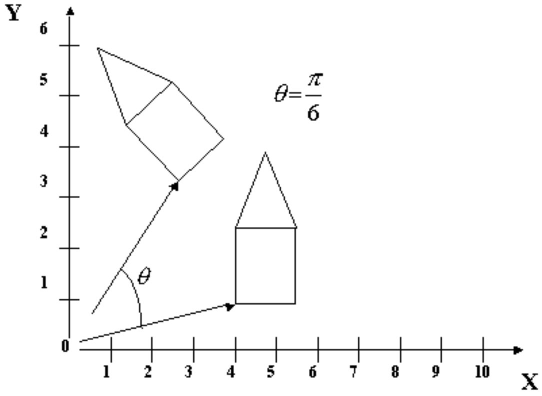
A straight line can be translated by applying the above transformation equation to each of the line endpoints and redrawing the line between the new coordinates. Similarly a polygon can be translated by applying the above transformation equation to each vertices of the polygon and redrawing the polygon with new coordinates. Similarly curved objects can be translated. For example to translate circle or ellipse, we translate the center point and redraw the same using new center point.

Rotation

A two dimensional rotation is applied to an object by repositioning it along a circular path in the xy plane. To rotate a point, its coordinates and rotation angle is required. Rotation is performed around a fixed point called pivot point. In start we will assume pivot point to be the origin or in other words we will find rotation equations for the rotation of object with respect to origin, however later we will see if we change our pivot point what should be done with the same equations.

Another thing is to be noted that for a positive angle the rotation will be anti-clockwise where for negative angle rotation will be clockwise.

Now for the rotation around the origin as shown in the above figure we required original position/ coordinates which in our case is P(x,y) and rotation angle θ. Now using polar coordinates assume point is already making angle Φ from origin and distance of point from origin is r, therefore we can represent x and y in the form:



x = r cosΦ and y = r sinΦ

Now if we want to rotate point by an angle θ, we have new angle that is (Φ+ θ), therefore now point P′(x′,y′) can be represented as:

x′ = r cos(Φ + θ) = r cosΦ cosθ – r sinΦ sinθ
and
y′ = r sin(Φ + θ) = r cosΦ sinθ + r sinΦ cosθ

Now replacing r cosΦ = x and r sinΦ = y in above equations we get:

x′ = x cosθ – y sinθ and y′= x sinθ + y cosθ

Again we can represent above equations with the help of column vectors:

Important

P′= R . P

Where

$$P' = \begin{bmatrix} x' \\ y' \end{bmatrix} \quad R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \quad P = \begin{bmatrix} x \\ y \end{bmatrix}$$

When coordinate positions are represented as row vectors instead of column vectors, the matrix product in rotation equation is transposed so that the transformed row coordinate vector [x′,y′] is calculated as:

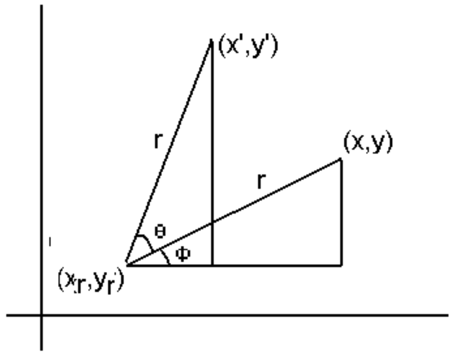
$$\begin{aligned} P'^{T} &= (R \,.\, P)^{T} \\ &= P^{T} \,.\, R^{T} \end{aligned}$$

Important

Where P^T and the other transpose matrix can be obtained by interchanging rows and columns. Also, for rotation matrix, the transpose is obtained by simply changing the sign of the sine terms.

Rotation about an Arbitrary Pivot Point:
As we discussed above that pivot point may be any point as shown in the above figure, however for the sake of simplicity we assume above that pivot point is at origin.

Anyhow, the situation can be dealt easily as we have equations of rotation with respect to origin. We can simply involve another transformation already read that is translation so simply translate pivot point to origin. By translation, now points will make angle with origin, therefore apply the same rotation equations and what next? Simply retranslate the pivot point to its original position that is if we subtract x_r,y_r now add them therefore we get following equations:

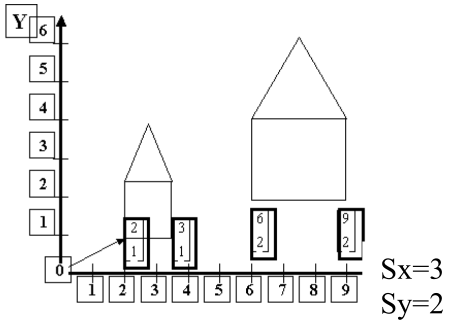


$$x' = x_r + (x - x_r)\cos\theta - (y - y_r)\sin\theta$$
$$y' = y_r + (x - x_r)\sin\theta - (y - y_r)\cos\theta$$

As it is discussed in translation rotation is also rigid-body transformation that moves the object along a circular path. Now if we want to rotate a point we already achieved it. But what if we want to move a line along its one end point very simple treat that end point as pivot point and perform rotation on the other end point as discussed above. Similarly we can rotate any polygon with taking some pivot point and recalculating vertices and then redrawing the polygon.

Scaling

A scaling transformation changes the size of an object. Scaling may be in any terms means either increasing the original size or decreasing the original size. An exemplary scaling is shown in the above figure where scaling factors used Sx=3 and Sy=2. So, what are these scaling factors and how they work very simple, simply we multiply each coordinate with its respective scaling factor.



Therefore, scaling with respect to origin is achieved by multiplying x coordinate with factor S_x and y coordinate with factor S_y. Therefore, following equations can be expressed:

$$x' = x.S_x$$
$$y' = y.S_y$$

In matrix form it can be expressed as:

$$P' = S.P$$

Important
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} Sx & 0 \\ 0 & Sy \end{bmatrix} . \begin{bmatrix} x \\ y \end{bmatrix}$$

Now we may have different values for scaling factor. Therefore, as it is multiplying factor therefore, if we have scaling factor > 1 then the object size will be increased than original size; whereas; in reverse case that is scaling factor < 1 the object size will be decreased than original size and obviously there will be no change occur in size for scaling factor equal 1.

Two variations are possible in scaling that is having scaling factors to be kept same that is to keep original shape; which is called uniform scaling having Sx factor equal Sy factor. Other possibility is to keep Sx and Sy factor unequal that is called differential scaling and that will alter the original shape that is a square will no more remain square.

Now above equation of scaling can be applied to any line, circle and polygon etc. However, as in case of line and polygon we will scale ending points or vertices then redraw the object but in circle or ellipse we will scale the radius.

Now coming to the point when scaling with respect to any point other then origin, then same methodology will work that is to apply translation before scaling and retranslation after scaling. So here if we consider fixed/ pivot point (x_f,y_f), then following equations will be achieved:

$$x' = x_f + (x - x_f)S_x$$
$$y' = y_f + (y - y_f)S_y$$

These can be rewritten as:

$$x' = x.\ S_x + x_f\,(1 - S_x)$$
$$y' = y.\ S_y\ + y_f\,(1 - S_y)$$

Where the terms $x_f\,(1 - S_x)$ and $y_f\,(1 - S_y)$ are constant for all points in the object.

Lecture No.12 2D Transformations II

Before starting our next lecture just recall equations of three basic transformations i.e. translation, rotation and scaling:

Important

Translation: P′= P + T

Rotation: P′= R. P

Scaling: P′= S. P

In many cases of computer graphics applications we require sequence of transformations. For example in animation on each next move we may have object to be translated than scaled. Similarly in games an object in a particular moment may have to be rotated as well as translated. That means we have to perform sequence of matrix operations but the matrix we have seen in the previous lecture have order which restrict them to be operated in sequence. However, with slight reformulated we can bring them into the form where they can easily be operated in any sequence thus efficiency can be achieved.

Homogeneous Coordinates

Again considering our previous lecture all the three basic transformations covered in that lecture can be expressed by following equation:

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

With coordinate positions P and P' represented as column vectors. Matrix M1 is a 2 by 2 array containing multiplicative factors, and M_2 is a two-element column matrix containing translation terms. For translation, M_1 is a the identity matrix, For rotation or scaling, M_2 contains the translational terms associated with the pivot point or scaling fixed point. To produce a sequence of transformations with these equations, such as scaling followed by rotation then translation, we must calculate the transformed coordinate's one step at a time. First, coordinate positions are scaled, then these scaled coordinates are rotated, and finally the rotated coordinates are translated.

Now the question is 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

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

We can choose the h as any non-zero value. However, a convenient choice is 1, thus (x, y) has homogeneous coordinates as (x, y, 1). Expressing positions in homogeneous coordinates allows us to represent all geometric transformation equations as matrix multiplications. Coordinates are represented with three-element column vectors, and transformation operations are written as 3 by 3 matrices.

Translation with Homogeneous Coordinates

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 (tx, ty) . P Important

Rotation with Homogeneous Coordinates

The rotation can now be expressed using homogeneous coordinates as:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Important

Abbreviated as:

P′ = R (θ) . P

Scaling with Homogeneous Coordinates

The scaling can now be expressed using homogeneous coordinates as:

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

Important

Abbreviated as:

P′ = S (Sx, Sy) . P

Matrix representations are standard methods for implementing transformations in graphics systems. In many systems, rotation and scaling functions produce transformations with respect to the coordinate origin as expressed in the equation above. Rotations and scaling relative to other reference positions are then handled as a succession of transformation operations.

Composite Transformations

As in the previous section we achieved homogenous matrices for each of the basic transformation, we can find a matrix for any sequence of transformation as a composite transformation matrix by calculating the matrix product of the individual transformations.

Translations

If two successive translations vectors (tx_1, ty_1) and (tx_2, ty_2) are applied to a coordinate position P, the final transformed location P is calculated as

$$P' = T(tx_2,ty_2) \,.\, \{T(tx_1,ty_1) \,.\, P\}$$
$$= \{T(tx_2,ty_2) \,.\, T(tx_1,ty_1)\} \,.\, P$$

where P and P′ are represented as homogeneous-coordinate column vectors. The composite transformation matrix for this sequence of translations is

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

or

$$T(tx_2,ty_2) \,.\, T(tx_1,ty_1) = T(tx_1 + tx_2 \,,\, ty_1 + ty_2)$$

Which means that two successive translations are additive. Hence,

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

Composite Rotations

Two successive Rotations applied to a point P produce the transformed position

$$P' = R(\theta_2) \,.\, \{R(\theta_1) \,.\, P\}$$
$$= \{R(\theta_2) \,.\, R(\theta_1)\} \,.\, P$$

By multiplying the two rotation matrices, we can verify that two successive rotations are additive:

$$R(\theta_2) \,.\, R(\theta_1) = R(\theta_1 + \theta_2)$$

so that the final rotated coordinates can be calculated with the composite rotation matrix as

$$P' = R(\theta_1 + \theta_2) \,.\, P$$

Composite Scaling

Concatenating transformation matrices for two successive scaling operations produces the following composite scaling matrix:

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

or

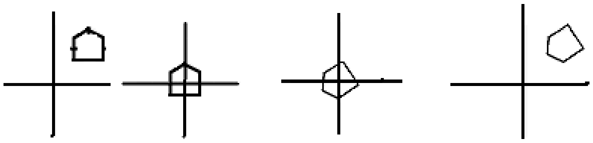
$S(sx_2,sy_2).S(sx_1,sy_1) = S(sx_1.sx_2, sy_1.sy_2)$

The resulting matrix in the case indicates that successive scaling operations are multiplicative. That is, if we were to triple the size of an object twice in succession, the final size would be nine times that of the original.

General Pivot Point Rotation

With a graphics package that only provides a rotate function for revolving object about the coordinate origin, we can generate rotations about any selected pivot point (x_r, y_r) by performing the following sequence of translate-rotate-translate operations:

Translate the object so that the pivot-point positions is moved to the coordinate origin
Rotate the object about the coordinate origin
Translate the object so that the pivot point is returned to its original position



$$\begin{bmatrix} 1 & 0 & x_r \\ 0 & 1 & y_r \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} 1 & 0 & -x_r \\ 0 & 1 & -y_r \\ 0 & 0 & 1 \end{bmatrix}$$

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

which can be expressed in the form

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

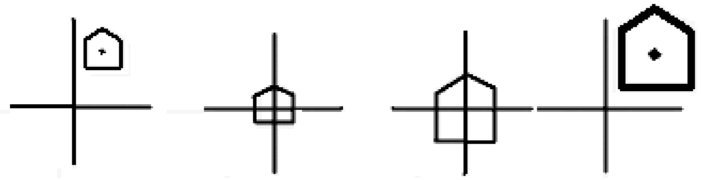
where $T(-x_r , -y_r) = T\text{-}1(x_r , y_r)$.

General Fixed Point Scaling

Following figure is showing a transformation sequence to produce scaling with respect to a selected fixed point (x_f, y_f) using a scaling function that can only scale relative to the coordinate origin.

Translate object so that the fixed point coincides with the coordinate origin
Scale the object with respect to the coordinate origin
Use the inverse translation of step 1 to return the object to its original position

Concatenating the matrices for these three operations produces the required scaling matrix.



$$\begin{bmatrix} 1 & 0 & x_f \\ 0 & 1 & y_f \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} 1 & 0 & -x_f \\ 0 & 1 & -y_f \\ 0 & 0 & 1 \end{bmatrix}$$

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

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

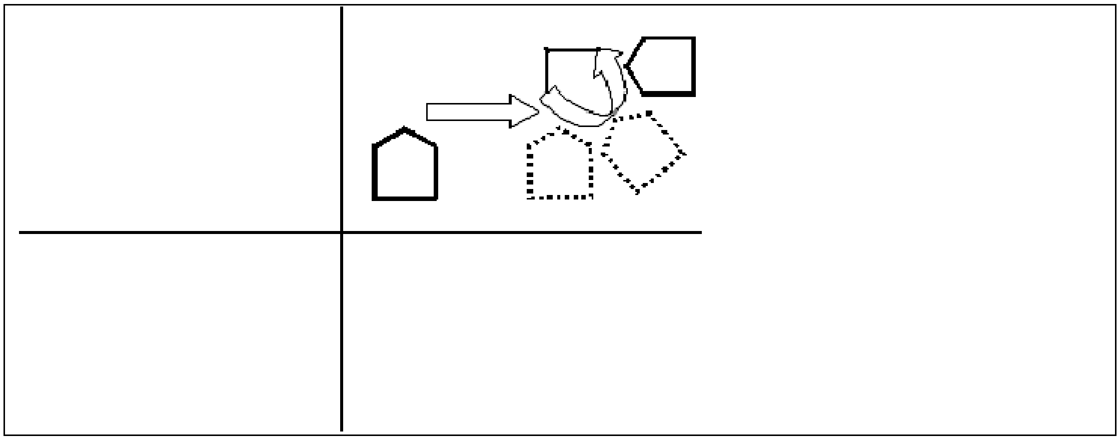
This transformation is automatically generated on systems that provide a scale function that accepts coordinates for the fixed point.

Concatenation Properties

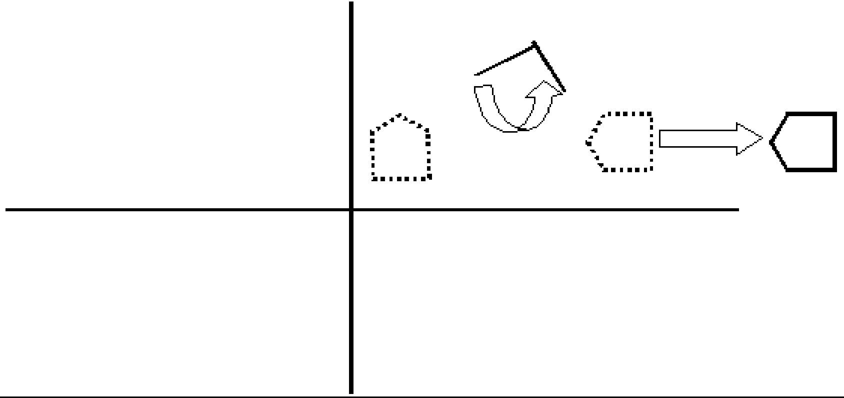
Matrix multiplication is associative. For any three matrices A, B and C, the matrix product A. B. C can b3e performed by first multiplying A and B or by first multiplying B and C:

A . B . C = (A . B) . C = A . (B . C)

Therefore, we can evaluate matrix products using a left-to-right or a right-to-left associative grouping. On the other hand, transformation products may not be commutative. The matrix product A. B is not equal to B. A, in general. This means that if we want to translate and rotate an object, we must be careful about the order in which the composite matrix is evaluated as show in following figure.



Reversing the order in which a sequence of transformations is performed may affect the transformed position of an object. In above figure an object is first translated, and then rotated. Whereas, in this figure an object is rotated first, then translated.



For some special cases, such as a sequence of transformations all of same kind, the multiplication of transformation matrices is commutative. As an example, two successive rotations could be performed in either order and the final position would be the same. This commutative property holds also for two successive translations or two successive scalings. Another commutative pair of operation is rotation and uniform scaling ($S_x = S_y$).

General Composite Transformations and Computational Efficiency

A general two-dimensional transformation, representing a combination of translations, rotations, and scaling, can be expressed as

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

The four elements rs_{ij} are the multiplicative rotation-scaling terms in the transformation that involve only rotating angles and scaling factors. Elements trs_x and trs_y are the translational terms containing combinations of translation distances, pivot-point and fixed-point coordinates, and rotation angles and scaling parameters. For example, if an object is to be scaled and rotated about its centroid coordinates (x_c, y_c) and then translated, the values for the elements of the composite transformation matrix are

$T\ (t_x,t_y)\ .\ R(x_c,y_c,\theta)\ .\ S(x_c,y_c,s_x,s_y)$

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

Although matrix given before above matrix requires nine multiplications and six additions, the explicit calculations for the transformed coordinates are

$x' = x.rs_{xx} + y.rs_{xy} + trs_x$
$y' = x.rs_{yx} + y.rs_{yy} + trs_y$

Thus, we only need to perform four multiplications and four additions to transform coordinate positions. This is the maximum number of computations required for any transformation sequence, once the individual matrices have been concatenated and the elements of the composite matrix evaluated. Without concatenation, the individual transformations would be applied one at a time and the number of calculations could be significantly increased. An efficient implementation for the transformation operations, therefore, is to formulate transformation matrices, concatenate any transformation sequence, and calculate transformed coordinates using above equations.

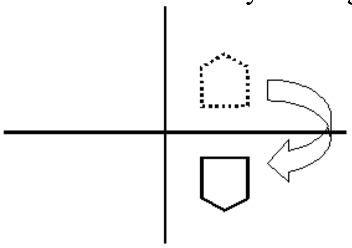
Other Transformations

Basic transformations such as translation, rotation, and scaling are included in most graphics packages. Some packages provide a few additional transformations that are useful in certain applications. Two such transformations are reflection and shear.

Reflection

A reflection is a transformation that produces a mirror image of an object. The mirror image for a two-dimensional reflection is generated relative to an axis of reflection by rotating the object 180^o about the reflection axis. We can choose an axis of reflection in the xy plane or perpendicular to the xy plane. When the reflection axis is a line in the xy plane; the rotation path about this axis is in a plane perpendicular to the xy plane. For reflection axes that are perpendicular to the xy plane, the rotation path in the xy plane. Following are examples of some common reflections.

Reflection about the line y=0, the x-axis, relative to axis of reflection can be achieved by rotating the object about axis of reflection by 180o.



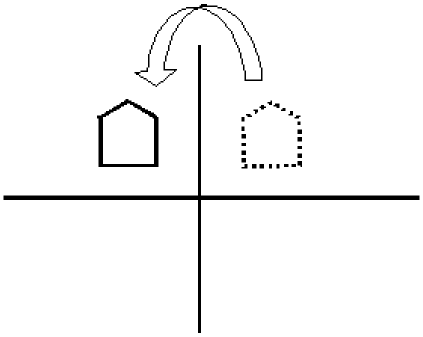
$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The transformation matrix is

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

Similarly in case of reflection about y-axis the transformation matrix will be, also the reflection is shown in following figure:

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



Shear

A transformation that distorts the shape of an object such that the transformed shape appears as if the object were composed of internal layers that had been caused to slide over each other is called a shear. Two common shearing transformations are those that shift coordinate x values and those that shift y values.

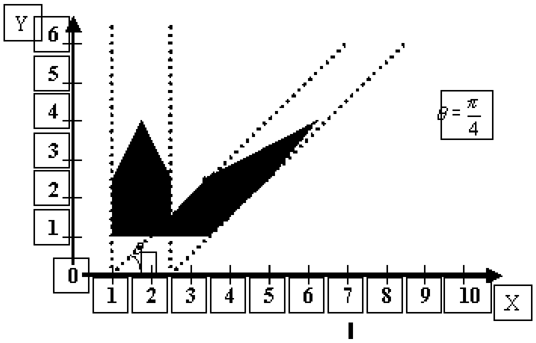
An x direction shear relative to the x axis is produced with the transformation matrix

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

which transforms coordinate position as

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

Any real number can be assigned to the shear parameter sh_x. A coordinate position (x,y) is then shifted horizontally by an amount proportional to its distance (y value) from the x axis (y=0). Setting sh_x to 2, for example, changes the square in following figure into a parallelogram. Negative values for sh_x shift coordinate positions to the left.



Similarly y-direction shear relative to the y-axis is produced with the transformation matrix

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

and coordinate positions transformed as

$x' = x$

$y' = sh_y \cdot x + y$

Another similar transformation may be in x and y direction shear, where matrix will be

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

and coordinate positions transformed as

$x' = x + sh_x \cdot y$

$y' = sh_y \cdot x + y$

Lecture No.13 Drawing Example

Let us now learn some of the implementation techniques. So far we already have done with learning drawing primitives including output primitives as well as filling primitives. Also we have studied transformations. So we should be in position to make use of them in two-dimensional drawing. Though most of you will think that they can draw two-dimensional drawing very easily yet it may not be true due to lack of knowledge of some implementation techniques, which are very useful in drawing as well as in transformation. So we will cover this with some examples.

Drawing Table

First of all we are going to draw a very simple drawing that is a “table”. Yes, a simple rectangular table with four legs. So, in order to draw such table we have to draw “table top” plus four legs connecting four edges of the rectangle.

Design

Here we will first design the table like an ordinary student. So, what we will do we will see the location of the table. For example assume that our screen has dimensions 640*480 and initially we want to draw table right in the middle of the center. Also, another factor is important that is y axis travels from top to bottom. That is y=0 will be the top edge of the screen and 480 will be the lower edge of the screen. Another thing is the dimension of the table we want to draw. Therefore, we have table that has width 20, length 14 and height 10.

Therefore, we have to find out four vertices that make the corners of the table. So, first of all consider x coordinate. Left edge of the table will be 10 less from the center of the screen that is 320. Therefore, x1 and x4 values will be 310. Similarly right edge of the table will be 10 plus center of the screen. Therefore, x2 and x3 values will be 330. Similarly, top edge of the table will be 10 less from the center of the screen that is 240. Therefore, y1 and y2 values will be 233 and y3 and y4, which are lying on the lower edge, will be 247. Finally, last parameter is required to define the length of legs. Having length of legs we have to simply draw vertical lines of that length starting from each corner respectively. Therefore, following code will be required to draw such a table:

```
void translate(int tx, int ty)
{
        xc+=tx;
        yc+=ty;
        x1+=tx;
        x2+=tx;
        x3+=tx;
        x4+=tx;
        y1+=ty;
        y2+=ty;
        y3+=ty;
        y4+=ty;
}
```

```
        void rotate (float angle)
        {
                int tempx=x1;
                x1=xc+(tempx-xc)*cos(angle)-(y1-yc)*sin(angle);
                y1=yc+(tempx-xc)*sin(angle)+(y1-yc)*cos(angle);
                tempx=x2;
                x2=xc+(tempx-xc)*cos(angle)-(y2-yc)*sin(angle);
                y2=yc+(tempx-xc)*sin(angle)+(y2-yc)*cos(angle);
                tempx=x3;
                x3=xc+(tempx-xc)*cos(angle)-(y3-yc)*sin(angle);
                y3=yc+(tempx-xc)*sin(angle)+(y3-yc)*cos(angle);
                tempx=x4;
                x4=xc+(tempx-xc)*cos(angle)-(y4-yc)*sin(angle);
                y4=yc+(tempx-xc)*sin(angle)+(y4-yc)*cos(angle);
        }

        void scale(int sx, int sy)
        {
                x1=xc+(x1-xc)*sx;
                x2=xc+(x2-xc)*sx;
                x3=xc+(x3-xc)*sx;
                x4=xc+(x4-xc)*sx;
                y1=yc+(y1-yc)*sy;
                y2=yc+(y2-yc)*sy;
                y3=yc+(y3-yc)*sy;
                y4=yc+(y4-yc)*sy;
                legLength*=sy;
        }

x1=310, x2=330, x3=330, x4=310;
y1=233, y2=233, y3=247, y4=247;
legLength=10;
```

So, what I want that you should observe the issue. Now consider first of all translation. In translation you have to translate all the points one by one and redraw the picture. Instruction to translate the table will be of the form:

*Now in this design drawing is pretty simple. We have to draw 4 lines between corner points. That is from (x1, y1) to (x2, y2), from (x2, y2) to (x3, y3), from (x3, y3) to (x4, y4) and from (x4, y4) to (x1, y1). That will suffice our table top. Next simply draw four lines each starting from one of the corner of the table in the vertical direction having length 10. Now let us see the simple code of drawing such tab*_**le:**_

```
//Table Top
line (x1, y1, x2, y2);
line (x2, y2, x3, y3);
line (x3, y3, x4, y4);
line (x4, y4, x1, y1);
//Table Legs
```

```
        line (x1, y1, x1, y1+legLength);
        line (x2, y2, x2, y2+legLength);
        line (x3, y3, x3, y3+legLength);
        line (x4, y4, x4, y4+legLength);
```

Now is not that easy to draw table in the same manner? We will discuss the problem after take a bit look at basic transformations (translation, rotation, scaling). The code is:

```
        void translate(int tx, int ty)
        {
                xc+=tx;
                yc+=ty;
                x1+=tx;
                x2+=tx;
                x3+=tx;
                x4+=tx;
                y1+=ty;
                y2+=ty;
                y3+=ty;
                y4+=ty;
        }
```

Now seeing the code you can easily understand the idea. In translation we have to translate all points one by one and then redrawing the table at new calculated points. Later you will see in the other method that translation will only involve one line.

Anyhow next we will move to other transformation (rotation). Having pivot point at the center of the screen, we have to perform translation in three steps. That is translation then rotation and then translation. So take a look at the code:

```
        void rotate (float angle)
        {
                int tempx=x1;
                x1=xc+(tempx-xc)*cos(angle)-(y1-yc)*sin(angle);
                y1=yc+(tempx-xc)*sin(angle)+(y1-yc)*cos(angle);
                tempx=x2;
                x2=xc+(tempx-xc)*cos(angle)-(y2-yc)*sin(angle);
                y2=yc+(tempx-xc)*sin(angle)+(y2-yc)*cos(angle);
                tempx=x3;
                x3=xc+(tempx-xc)*cos(angle)-(y3-yc)*sin(angle);
                y3=yc+(tempx-xc)*sin(angle)+(y3-yc)*cos(angle);
                tempx=x4;
                x4=xc+(tempx-xc)*cos(angle)-(y4-yc)*sin(angle);
                y4=yc+(tempx-xc)*sin(angle)+(y4-yc)*cos(angle);
        }
```

So, here calculations required each time and for each pixel; whereas; you will observe that we can make that rotation pretty simple. A similar problem is lying in the case of scaling that is to perform three steps; translation then scaling and then translation. So look at the code given below:

```
	void scale(int sx, int sy)
	{
		x1=xc+(x1-xc)*sx;
		x2=xc+(x2-xc)*sx;
		x3=xc+(x3-xc)*sx;
		x4=xc+(x4-xc)*sx;
		y1=yc+(y1-yc)*sy;
		y2=yc+(y2-yc)*sy;
		y3=yc+(y3-yc)*sy;
		y4=yc+(y4-yc)*sy;
		legLength*=sy;
	}
```

Therefore, same heavy calculations involves in scaling. So, here we will conclude our first method and will start next method so that we can judge how calculations become simple. Now having all discussion on table drawing, let us now consider the complete implementation of class Table:

```
/*****************************************************************
Table is designed without considering pivot point simply taking points according to the requirement.

Therefore, translation of table involves translation of all points.

Scaling and Rotation will be done after translation.
*****************************************************************/

#include <graphics.h>
#include <iostream.h>
#include <conio.h>
#include <math.h>

float round(float x)
{
	return x+0.5;
}

class Table
{
	private:
		int xc, yc;//Center of the figure
		int xp, yp;//Pivot point for this figure
		int x1, x2, x3, x4;
		int y1, y2, y3, y4;
		int legLength;

	public:
		Table()
		{
    xc=320, yc=240;//Center of the figure
    xp=0; yp=0;//Pivot point for this figure
    x1=310, x2=330, x3=330, x4=310;
    y1=233, y2=233, y3=247, y4=247;
    legLength=10;
}

void translate(int tx, int ty)
{
    xc+=tx;
    yc+=ty;
    x1+=tx;
    x2+=tx;
    x3+=tx;
    x4+=tx;
    y1+=ty;
    y2+=ty;
    y3+=ty;
    y4+=ty;
}

void rotate (float angle)
{
    int tempx=x1;
    x1=xc+(tempx-xc)*cos(angle)-(y1-yc)*sin(angle);
    y1=yc+(tempx-xc)*sin(angle)+(y1-yc)*cos(angle);
    tempx=x2;
    x2=xc+(tempx-xc)*cos(angle)-(y2-yc)*sin(angle);
    y2=yc+(tempx-xc)*sin(angle)+(y2-yc)*cos(angle);
    tempx=x3;
    x3=xc+(tempx-xc)*cos(angle)-(y3-yc)*sin(angle);
    y3=yc+(tempx-xc)*sin(angle)+(y3-yc)*cos(angle);
    tempx=x4;
    x4=xc+(tempx-xc)*cos(angle)-(y4-yc)*sin(angle);
    y4=yc+(tempx-xc)*sin(angle)+(y4-yc)*cos(angle);
}

void scale(int sx, int sy)
{
    x1=xc+(x1-xc)*sx;
    x2=xc+(x2-xc)*sx;
    x3=xc+(x3-xc)*sx;
    x4=xc+(x4-xc)*sx;
    y1=yc+(y1-yc)*sy;
    y2=yc+(y2-yc)*sy;
    y3=yc+(y3-yc)*sy;
    y4=yc+(y4-yc)*sy;
    legLength*=sy;
}
```

```
                void draw()
                {
                        line (x1, y1, x2, y2);
                        line (x2, y2, x3, y3);
                        line (x3, y3, x4, y4);
                        line (x4, y4, x1, y1);
                        line (x1, y1, x1, y1+legLength);
                        line (x2, y2, x2, y2+legLength);
                        line (x3, y3, x3, y3+legLength);
                        line (x4, y4, x4, y4+legLength);
                }
};


void main()
{
        clrscr();
        int gdriver = DETECT, gmode, errorcode;
        initgraph(&gdriver, &gmode, "c:\\tc\\bgi");
        Table table;
        table.draw();
        setcolor(CYAN);
        table.translate(15, 25);
        table.draw();
        table.translate(50, 75);
        table.scale(3,2);
        table.draw();
        table.translate(-100, 75);
        table.rotate(3.14/4);
        table.draw();
        getch();
        closegraph();
}
```

Second Method

Well now we will design our table by considering pivot point. That is we will decide our pivot point and next all points will be taken according to that pivot point. Similarly you will see that this consideration will do a little effect on drawing portion of the code; otherwise all other things will become simpler.

Table Design

So let us start with designing the table, that is to calculate parameters of the table. That is 4 corners plus length of legs. First of all we assume that our pivot point is lying on the center of the screen and initially that is (0, 0). So having pivot point we will calculate other points with respect to that point.

So having length 20 units, left edge of the table will be ten digits away from the pivot point and away on the left side; therefore; x1 and x4 will be –10; and similarly right edge of the table will be ten digits away from the pivot point on the right side. Therefore, x2 and x3 will be 10 (yes, positive ten).

Now consider the top and lower edges of the table they will be 7 points away from the pivot point in each direction; therefore value of y on the upper edge will be –7 and on the lower edge it will be 7 (yes, positive seven). Now finally y1, y2 will be –7 and y3, y4 will be 7. Well, length of the leg will be simple ten. Therefore, now take a look at the parameters in this design:

```
xc=320, yc=240;//Center of the figure
xp=0; yp=0;//Pivot point for this figure
x1=-10, x2=10, x3=10, x4=-10;
y1=-7, y2=-7, y3=7, y4=7;
legLength=10;
```

Table Drawing

So, points x1, x2, x3, and x4 are not having the value at which they will appear on the screen rather they are at the relative distance from the pivot point. Here, we are also using xc, yc that is center on the screen that will keep pivot point align. Now having vertices defined in this fashion our drawing method will be differ from the older one. That is while drawing lines we will add center of the screen and pivot point in each vertex. That will take us to the exact position of the screen. Let us look at the drawing code:

```
int xc=this->xc+xp;
int yc=this->yc+yp;
line (xc+x1, yc+y1, xc+x2, yc+y2);
line (xc+x2, yc+y2, xc+x3, yc+y3);
line (xc+x3, yc+y3, xc+x4, yc+y4);
line (xc+x4, yc+y4, xc+x1, yc+y1);
line (xc+x1, yc+y1, xc+x1, yc+y1+legLength);
line (xc+x2, yc+y2, xc+x2, yc+y2+legLength);
line (xc+x3, yc+y3, xc+x3, yc+y3+legLength);
line (xc+x4, yc+y4, xc+x4, yc+y4+legLength);
```

So, in the above code first we have added xp to xc in order to reduce some of the calculations required in each line drawing command. Next, we have added that calculated figure to all line drawing commands in order to draw them exactly at the position where it should be appear in the screen.

Now having a bit difficulty while drawing there are many more facilities that we will enjoy in especially transformation.

Table Transformation

So, first of all consider Translation. In this technique translation is quite simple that is simply add translation vector in the pivot point. All other points will be calculated accordingly. Now look at the very simple code of translation:

```
void translate(int tx, int ty)
{
        xp+=tx;
        yp+=ty;
    }
```

So how simple add tx to xp and ty to yp. Similarly next consider rotation that is again very simple; no need of translation. Let us look at the code:

```
    void rotate (float angle)
    {
        int tempx=x1;
        x1=tempx*cos(angle)-y1*sin(angle);
        y1=tempx*sin(angle)+y1*cos(angle);
        tempx=x2;
        x2=tempx*cos(angle)-y2*sin(angle);
        y2=tempx*sin(angle)+y2*cos(angle);
        tempx=x3;
        x3=tempx*cos(angle)-y3*sin(angle);
        y3=tempx*sin(angle)+y3*cos(angle);
        tempx=x4;
        x4=tempx*cos(angle)-y4*sin(angle);
        y4=tempx*sin(angle)+y4*cos(angle);
    }
```

So, here you can check that there is no extra calculation simply rotated points are calculated using formula that is used to rotate a point around the origin. Now similarly given below you can see calculations of scaling.

```
    void scale(int sx, int sy)
    {
        x1=x1*sx;
        x2=x2*sx;
        x3=x3*sx;
        x4=x4*sx;
        y1=y1*sy;
        y2=y2*sy;
        y3=y3*sy;
        y4=y4*sy;
        legLength=legLength*sy;
    }
```

So very simple calculation is done again that is to multiply scaling factor with old vertices and new vertices will be obtained.

So, now we look at the class Table in which table is designed considering pivot point and taking all other points accordingly.

```
/*****************************************************************
Table is designed with considering pivot point and taking all other
points with respect to that pivot point.

Therefore, translation of table involves translation of only pivot
point, all other points will change respectively.
```

```
Scaling and Rotation will be done directly no translation or other
transformation is required.
*****************************************************************/


#include <graphics.h>
#include <iostream.h>
#include <conio.h>
#include <math.h>

float round(float x)
{
        return x+0.5;
}

class Table
{
        private:
                int xc, yc;//Center of the figure
                int xp, yp;//Pivot point for this figure
                int x1, x2, x3, x4;
                int y1, y2, y3, y4;
                int legLength;
                int sfx, sfy;                   //Scaling factor
        public:
        Table()
        {
                xc=320, yc=240;//Center of the figure
                xp=0; yp=0;//Pivot point for this figure
                x1=-10, x2=10, x3=10, x4=-10;
                y1=-7, y2=-7, y3=7, y4=7;
                legLength=10;
                sfx=1, sfy=1;
        }

        void translate(int tx, int ty)
        {
                xp+=tx;
                yp+=ty;
        }

        void rotate (float angle)
        {
                int tempx=x1;
                x1=tempx*cos(angle)-y1*sin(angle);
                y1=tempx*sin(angle)+y1*cos(angle);
                tempx=x2;
                x2=tempx*cos(angle)-y2*sin(angle);
                y2=tempx*sin(angle)+y2*cos(angle);
                tempx=x3;
                x3=tempx*cos(angle)-y3*sin(angle);
                y3=tempx*sin(angle)+y3*cos(angle);
                tempx=x4;
                x4=tempx*cos(angle)-y4*sin(angle);
                y4=tempx*sin(angle)+y4*cos(angle);
        }
        void scale(int sx, int sy)
        {
                x1=x1*sx;
                x2=x2*sx;
                x3=x3*sx;
                x4=x4*sx;
                y1=y1*sy;
                y2=y2*sy;
                y3=y3*sy;
                y4=y4*sy;
                legLength=legLength*sy;
        }
        void draw()
        {
                int xc=this->xc+xp;
                int yc=this->yc+yp;
                line (xc+x1, yc+y1, xc+x2, yc+y2);
                line (xc+x2, yc+y2, xc+x3, yc+y3);
                line (xc+x3, yc+y3, xc+x4, yc+y4);
                line (xc+x4, yc+y4, xc+x1, yc+y1);
                line (xc+x1, yc+y1, xc+x1, yc+y1+legLength);
                line (xc+x2, yc+y2, xc+x2, yc+y2+legLength);
                line (xc+x3, yc+y3, xc+x3, yc+y3+legLength);
                line (xc+x4, yc+y4, xc+x4, yc+y4+legLength);
        }};
void main()
{
        clrscr();
        int gdriver = DETECT, gmode, errorcode;
        initgraph(&gdriver, &gmode, "c:\\tc\\bgi");
        Table table;
        table.draw();
        setcolor(CYAN);
        table.translate(15, 25);
        table.draw();
        table.translate(50, 0);
        table.scale(3,2);
        table.draw();
        table.translate(-100, 0);
        table.rotate(3.14/4);
        table.draw();
        getch();
        closegraph();
}
```

Lecture No.14 Clipping-I

Concept

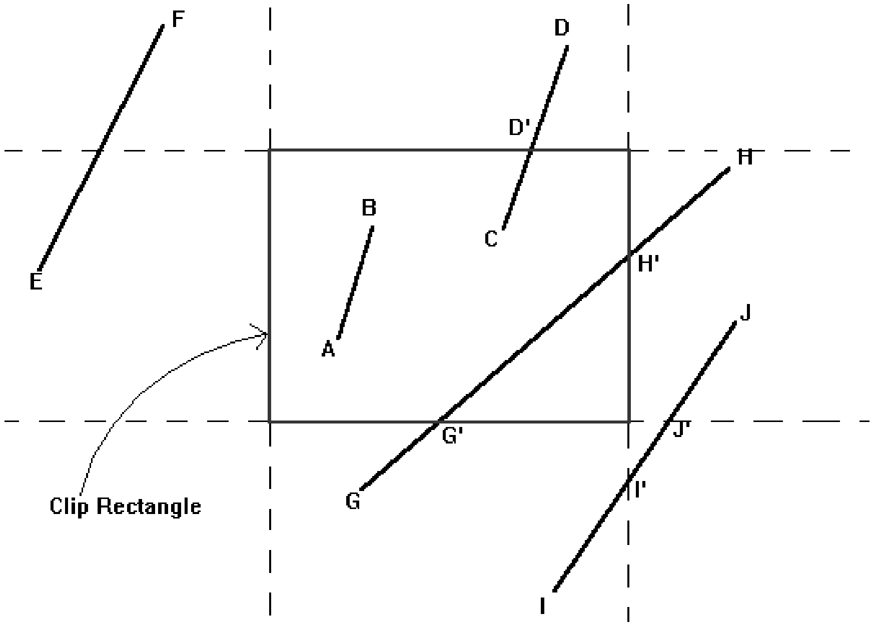
It is desirable to restrict the effect of graphics primitives to a sub-region of the canvas, to protect other portions of the canvas. All primitives are clipped to the boundaries of this **clipping rectangle**; that is, primitives lying outside the clip rectangle are not drawn.

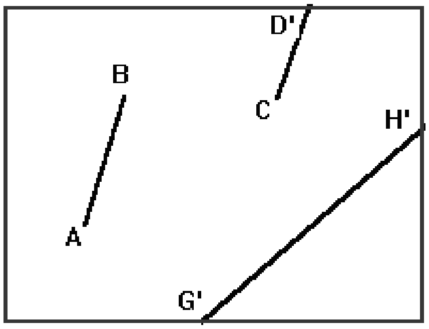
The default clipping rectangle is the full canvas (the screen), and it is obvious that we cannot see any graphics primitives outside the screen.

A simple example of line clipping can illustrate this idea:

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.

The red box is the clipping rectangle we will use later, and the dotted line is the extension of the four edges of the clipping rectangle.





Point Clipping

Assuming a rectangular clip window, point clipping is easy. we save the point if:

$x_{min} <= x <= x_{max}$
$y_{min} <= y <= y_{max}$

Line Clipping
This section treats clipping of lines against rectangles. Although there are specialized algorithms for rectangle and polygon clipping, it is important to note that other graphic primitives can be clipped by repeated application of the line clipper.

Clipping Individual Points
Before we discuss clipping lines, let's look at the simpler problem of clipping individual points.
If the x coordinate boundaries of the clipping rectangle are Xmin and Xmax, and the y coordinate boundaries are Ymin and Ymax, then the following inequalities must be satisfied for a point at (X, Y) to be inside the clipping rectangle:

Xmin < X < Xmax

and

Ymin < Y < Ymax

If any of the four inequalities does not hold, the point is outside the clipping rectangle.
MCQs 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 of a line:

x = x1 + u (x2 - x1)
y = y1 + u (y2 - y1), and 0 <= u <= 1.
If the value of u for an intersection with a clipping edge is outside the range 0 to 1, then the line does not enter the interior of the window at that boundary. If the value of u is within this range, then the line does enter the interior of the window at that boundary.

Solve Simultaneous Equations
To clip a line, we need to consider only its endpoints, not its infinitely many interior points. If both endpoints of a line lie inside the clip rectangle (eg AB, refer to the first example), the entire line lies inside the clip rectangle and can be trivially accepted. If one endpoint lies inside and one outside(eg CD), the line intersects the clip rectangle and we must compute the intersection point. If both endpoints are outside the clip rectaangle, the line may or may not intersect with the clip rectangle (EF, GH, and IJ), and we need to perform further calculations to determine whether there are any intersections.
The brute-force approach to clipping a line that cannot be trivially accepted is to intersect that line with each of the four clip-rectangle edges to see whether any intersection points lie on those 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. Next, we 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. In the first example, intersection points G' and H' are interior, but I' and J' are not.

The Cohen-Sutherland Line-Clipping Algorithm

The more efficient Cohen-Sutherland Algorithm performs initial tests on a line to determine whether intersection calculations can be avoided.

Steps for Cohen-Sutherland algorithm

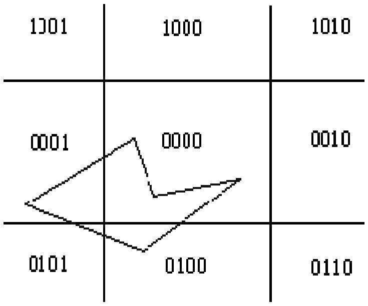
End-points pairs of the line are checked for trivial acceptance or trivial reject using outcode.

If not trivial-acceptance or trivial-reject, the line is divided into two segments at a clip edge.

Line is iteratively clipped by testing trivial-acceptance or trivial-rejected, and divided into two segments until completely inside or trivial-rejected.

Trivial acceptance/reject test

To perform trivial accept and reject tests, we extend the edges of the clip rectangle to divide the plane of the clip rectangle into nine regions. Each region is assigned a 4-bit code determined by where the region lies with respect to the outside halfplanes of the clip-rectangle edges. Each bit in the outcode is set to either 1 (true) or 0 (false); the 4 bits in the code correspond to the following conditions:



MCQs Bit 1: outside halfplane of top edge, above top edge Y > Ymax

Bit 2: outside halfplane of bottom edge, below bottom edge Y < Ymin

Bit 3: outside halfplane of right edge, to the right of right edge X > Xmax

Bit 4: outside halfplane of left edge, to the left of left edge X < Xmin

Conclusion

In summary, the Cohen-Sutherland algorithm is efficient when out-code testing can be done cheaply (for example, by doing bit-wise operations in assembly language) and trivial acceptance or rejection is applicable to the majority of line segments. (For example, large windows - everything is inside, or small windows - everything is outside).

Liang-Barsky Algorithm

Faster line clippers have been developed that are based on analysis of the parametric equation of a line segment, which we can write in the form:

$$x = x_1 + u\,\Delta x$$
$$y = y_1 + u\,\Delta y, \text{ where } 0 <= u <= 1$$

Where $\Delta x = x_2 - x_1$ and $\Delta y = y_2 - y_1$. Using these parametric equations, Cryus and Beck developed an algorithm that is generally more efficient than the Cohen-Sutherland algorithm. Later, Liang and Barsky independently devised an even faster parametric line-clipping algorithm. Following the Liang-Barsky approach, we first write the point-clipping in a parametric way:

$$x_{min} <= x_1 + u\,\Delta x <= x_{max}$$
$$y_{min} <= y_1 + u\,\Delta y <= y_{max}$$

of these four inequalities can be expressed as

$u * p_k <= q_k$, for k = 1, 2, 3, 4

Where parameters p and q are defined as:

$p_1 = -\Delta x,\qquad q_1 = x_1 - x_{min}$
$p_2 = -\Delta x,\qquad q_2 = x_{max} - x_1$
$p_3 = -\Delta y,\qquad q_3 = y_1 - y_{min}$
$p_4 = -\Delta y,\qquad q_4 = y_{max} - y_1$

Any line that is parallel to one of the clipping boundaries has $p_k = 0$ for the value of k corresponding to that boundary (k = 1, 2, 3, 4 correspond to the left, bottom, and top boundaries, respectively). If, for that value of k, we also find $q_k >= 0$, the line is inside the parallel clipping boundary.

When $p_k < 0$, the infinite extension of the line proceeds from the outside to the inside of the infitite extension of the particular clipping boundary. If $p_k > 0$, the line proceeds from the inside to the outside. For a nonzero value of $p_k = 0$, 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 that defines that part of the line that 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 o and the various values of r. Conversely, the value of u_2 is determined by examining the boundaries for which the line proceeds from inside to outside (p > o). 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.

This algorithm is presented in the following procedure. 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 by the function clipTest to determine whether the line can be rejected or whether the intersection parameters are to be adjusted.

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 *Important*

In general, the Liang-Barsky algorithm is more efficient than the Cohen Sutherland algorithm, since 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 have computed. In contrast, the 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. Both the Cohen Sutherland and the Liang Barsky algorithms can be extended to three-dimensional clipping.

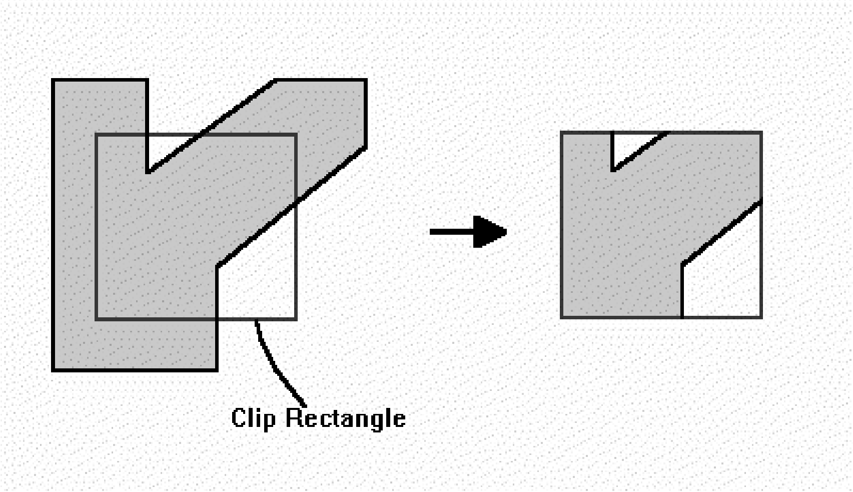
Lecture No.15 Clipping-II

Polygon Clipping

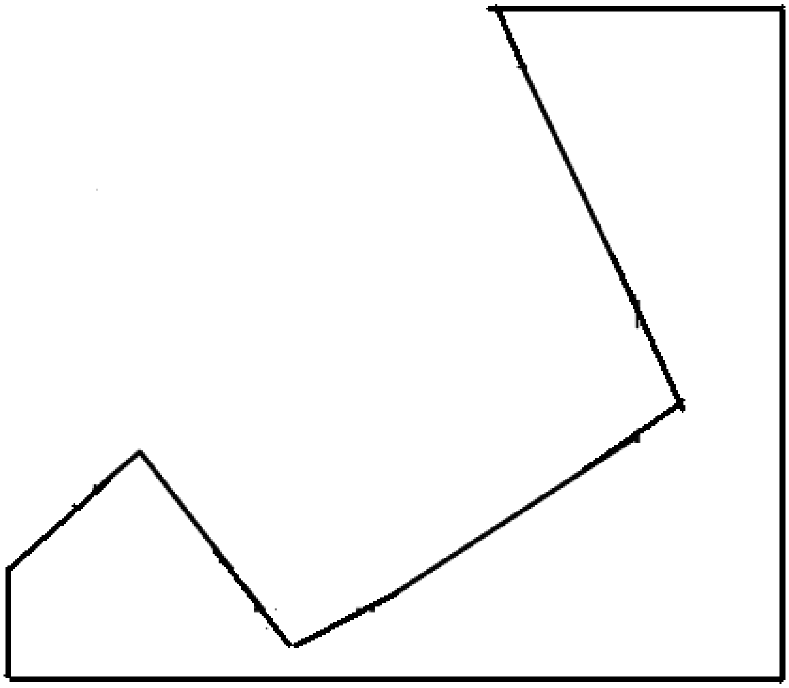
A polygon is usually defined by a sequence of vertices and edges. If the polygons are unfilled, line-clipping techniques are sufficient however, if the polygons are filled, the process in more complicated. A polygon may be fragmented into several polygons in the clipping process, and the original colour associated with each one. The Sutherland-Hodgeman clipping algorithm clips any polygon against a convex clip polygon. The Weiler-Atherton clipping algorithm will clip any polygon against any clip polygon. The polygons may even have holes.

An algorithm that clips a polygon must deal with many different cases. The case is particularly note worthy in that the concave polygon is clipped into two separate polygons. All in all, the task of clipping seems rather complex. Each edge of the polygon must be tested against each edge of the clip rectangle; new edges must be added, and existing edges must be discarded, retained, or divided. Multiple polygons may result from clipping a single polygon. We need an organized way to deal with all these cases.

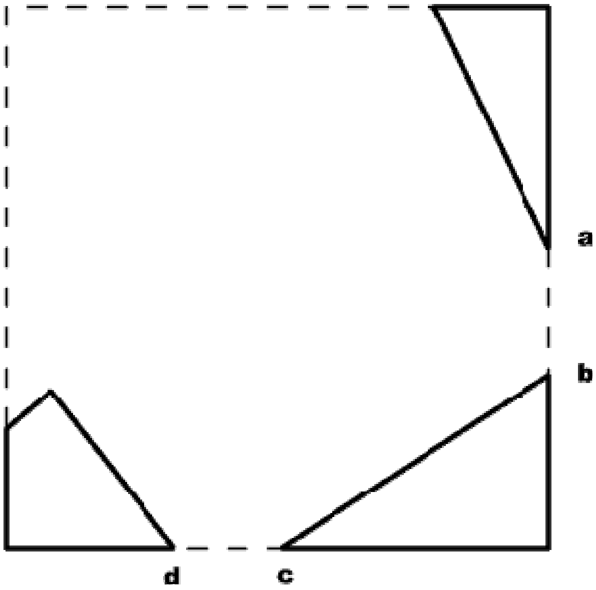
The following example illustrates a simple case of polygon clipping.



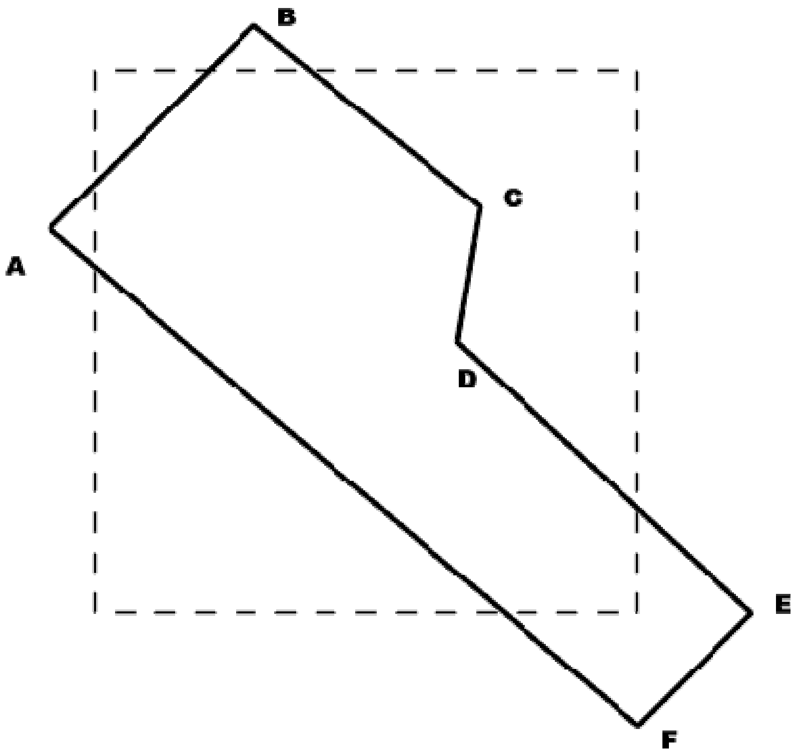
Given below are some examples to elaborate further.



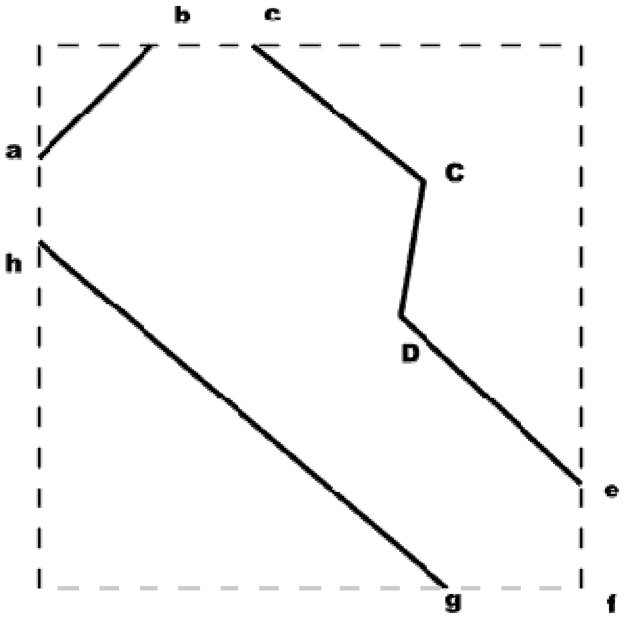
Polygon clipping - disjoint polygons.



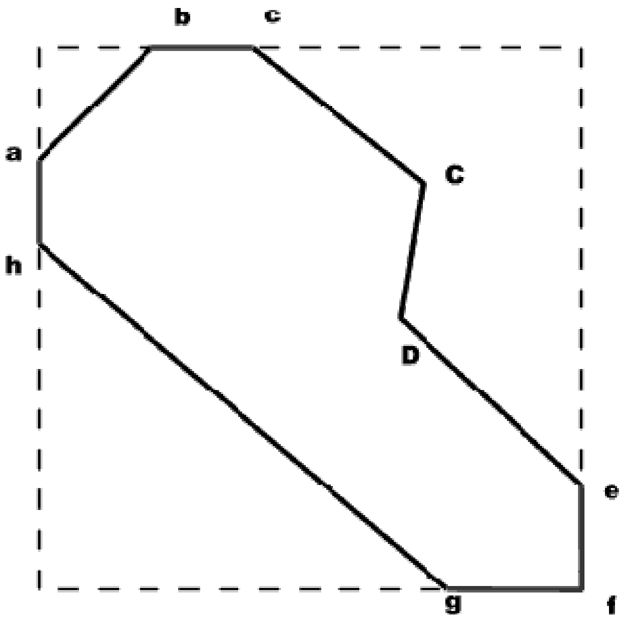
Polygon clipping - disjoint polygons.



Polygon clipping - open polygons.



Polygon clipping - open polygons.



Polygon clipping - open polygons.

Sutherland and Hodgman's polygon-clipping algorithm

Sutherland and Hodgman's polygon-clipping algorithm uses a divide-and-conquer strategy: It solves a series of simple and identical problems that, when combined, solve the overall problem. The simple problem is to 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.

Note the difference between this strategy for a polygon and the Cohen-Sutherland algorithm for clipping a line: The polygon clipper clips against four edges in succession, whereas the line clipper tests outcode to see which edge is crossed, and clips only when necessary.

Steps of Sutherland-Hodgman's polygon-clipping algorithm

- Polygons can be clipped against each edge of the window one at a time. Windows/edge intersections, if any, are easy to find since the X or Y coordinates are already known.
- 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.

Here is a STEP-BY-STEP example of polygon clipping.

Important

Four Cases of polygon clipping against one edge

The clip boundary determines a visible and invisible region. The edges from vertex i to vertex i+1 can be one of four types:

Case 1 : Wholly inside visible region - save endpoint

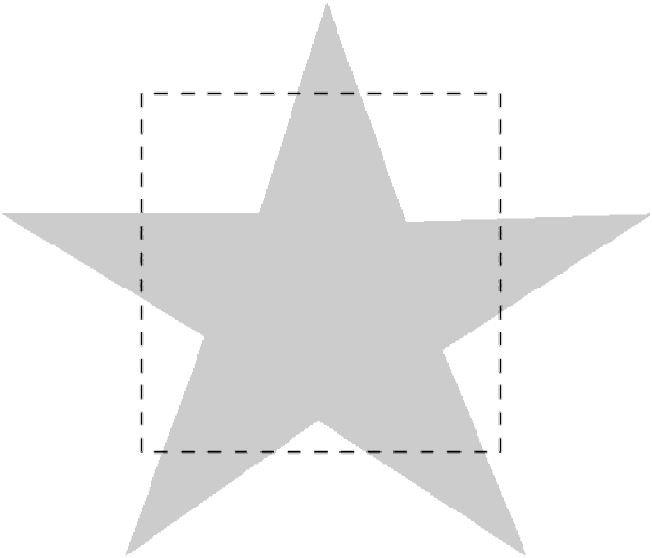
Case 2 : Exit visible region - save the intersection

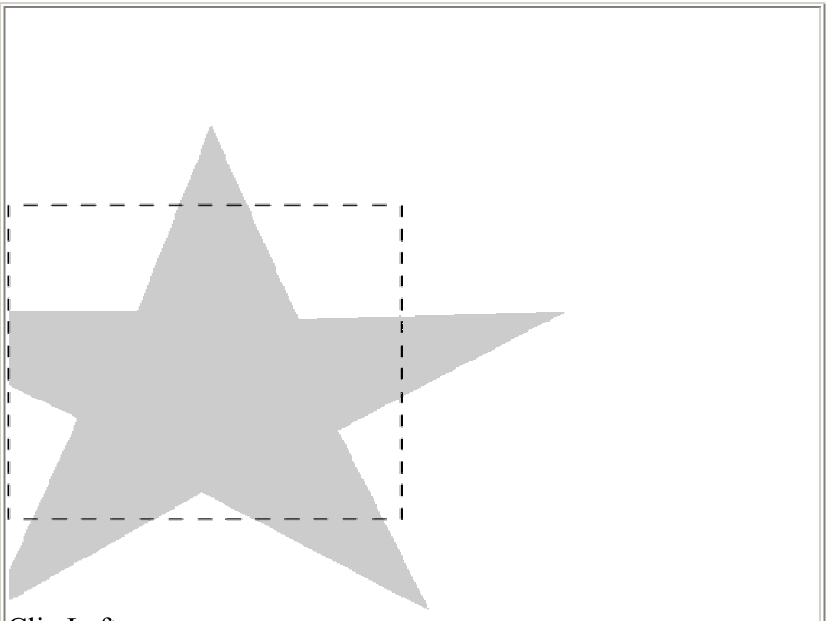
MCQs

Case 3 : Wholly outside visible region - save nothing

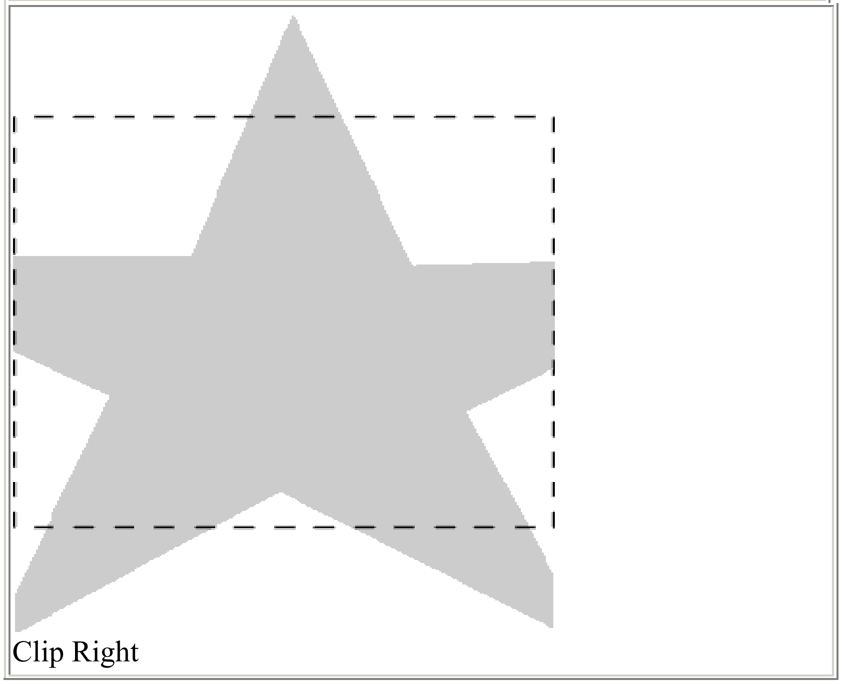
Case 4 : Enter visible region - save intersection and endpoint

Because clipping against one edge is independent of all others, it is possible to arrange the clipping stages in a pipeline. The input polygon is clipped against one edge and any points that are kept are passed on as input to the next stage of the pipeline. In this way four polygons can be at different stages of the clipping process simultaneously. This is often implemented in hardware.

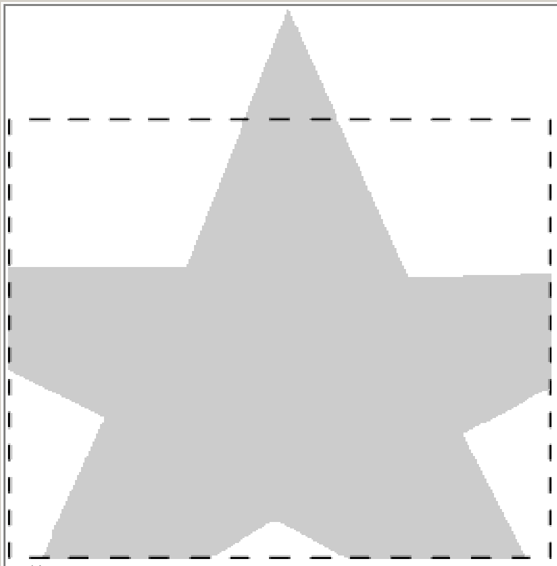
Example No # 1 Clipping a Polygon
 Original polygon



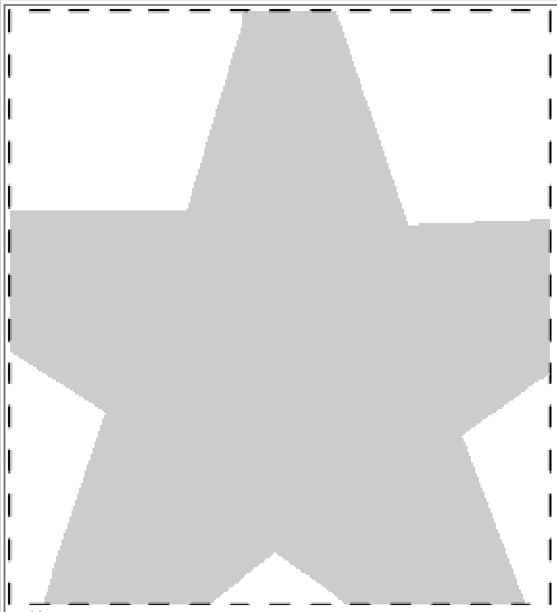
Clip Left



Clip Right



Clip Bottom

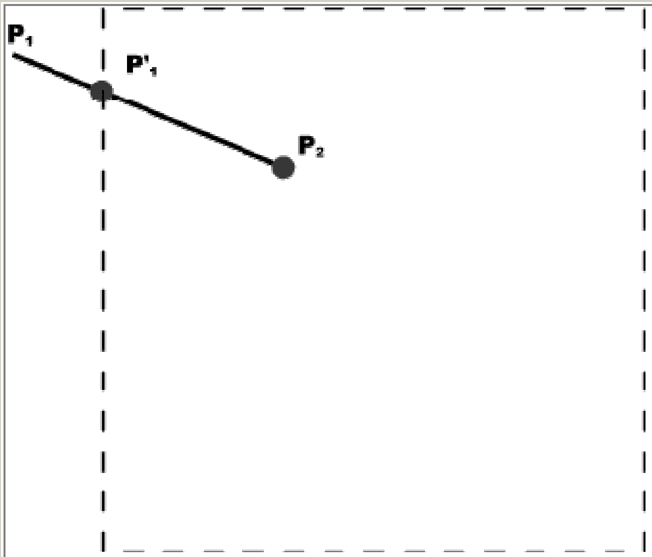


Clip Top

Example No #2 Clipping a Rectangle

If Clipping Rectangle is denoted by dashed lines and Line is defined by using points P1 and P2

Case i

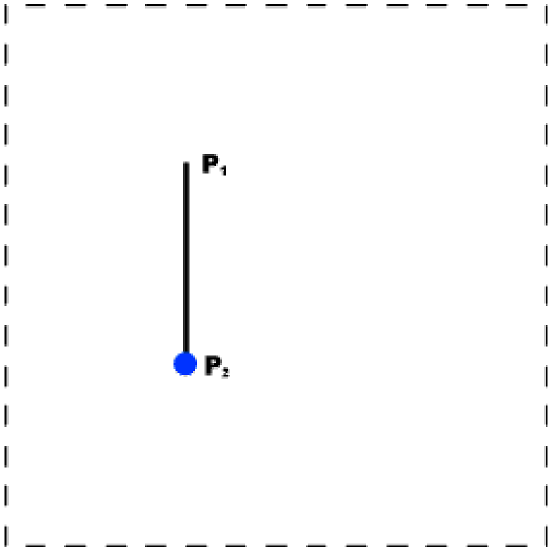


For each boundary b in [L(Left), R(Right), T(Top), B(Bottom)]
If P_1 outside and P_2 inside.

Output:

intersection Point (**P1'**)
Point $\mathbf{P_2}$

Case ii

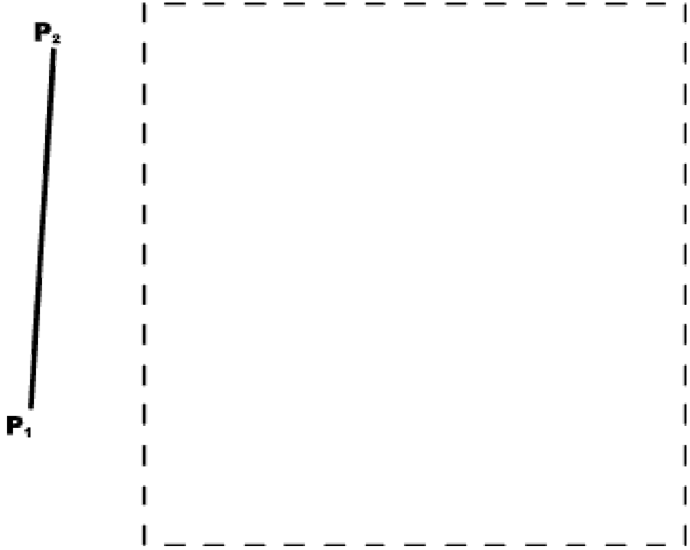


For each boundary b in [L(Left), R(Right), T(Top), B(Bottom)]
If P_1 inside and P_2 inside

Output:

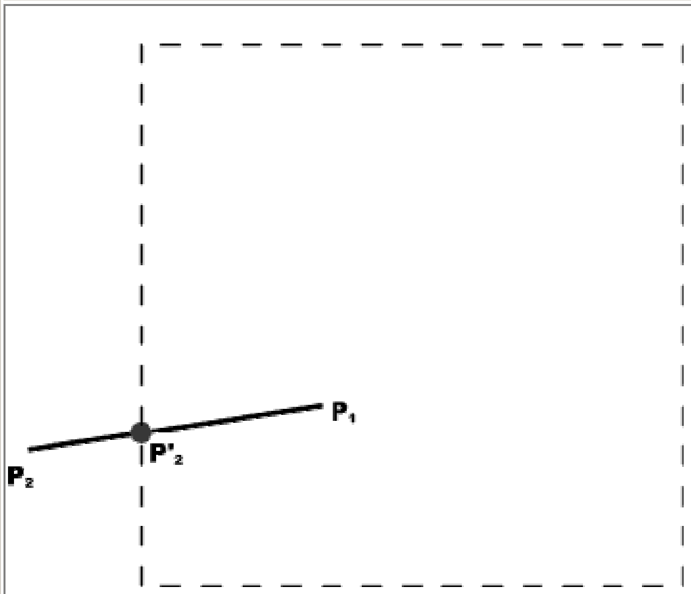
Point $\mathbf{P_2}$

Case iii



For each boundary b in [L(Left), R(Right), T(Top), B(Bottom)]
If P1 outside and P2 outside
Do nothing

Case iv



For each boundary b in [L(Left), R(Right), T(Top), B(Bottom)]
If P_1 inside and P_2 outside (We are going from P1 to P2)
Output:
Point of intersection (**P2’**) only.

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 *clipPoint* 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 *firstPoint* stores for each window boundary the first point flipped 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 Sutherlands -Hodgeman Algorithm

Convex polygons are correctly clipped by the Sutherland-Hodegeman algorithm, but concave polygons may be displayed with extraneous lines. This occurs when the clipped polygon should have two or more separate sections. But since there is only one output vertex list, the last vertex in the list is always joined to the first vertex. There are several things we could do to correct display concave polygons. For one, we could split the concave polygon into two or more convex polygons and process each convex polygon separately.

Another approach to check the final vertex list for multiple vertex points along any clip window boundary and correctly join pairs of vertices. Finally, we could use a more general polygon clipper, such as wither the Weiler-Atherton algorithm or the Weiler algorithm described in the next section.

Weiler-Atherton Polygon Clipping

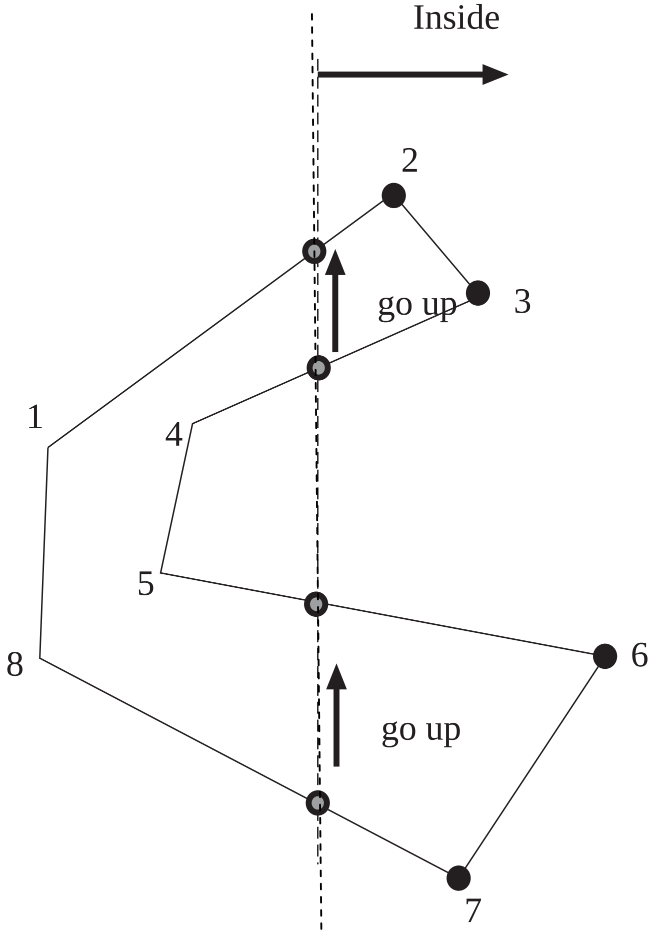
In this technique, the vertex-processing procedures for window boundaries are modified so that concave polygons are displayed correctly. This clipping procedure was developed as a method for identifying visible surfaces, and so it can be applied with arbitrary polygon-clipping regions.

The basic idea in this algorithm is that instead of always proceeding around the polygon edges as vertices are processed, we sometimes want to follow the window boundaries. Which path we follow depends on the polygon-processing direction(clockwise or counterclockwise) and whether the pair of polygon vertices currently being processed represents an outside-to-inside pair or an inside-to-outside pair. For clockwise processing of polygon vertices, we use the following rules:

MCQs

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

In following figure, the processing direction in the Wieler-Atherton algorithm and the resulting clipped polygon is shown for a rectangular clipping window.

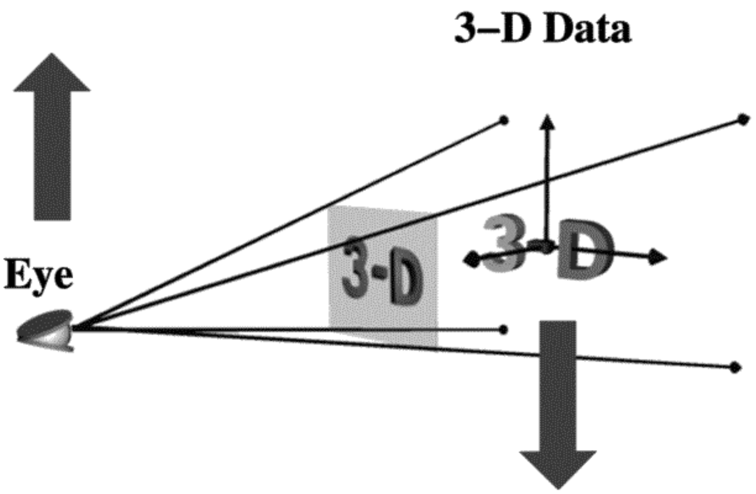

Inside
2
go up
3
1
4
5
8
6
go up
7

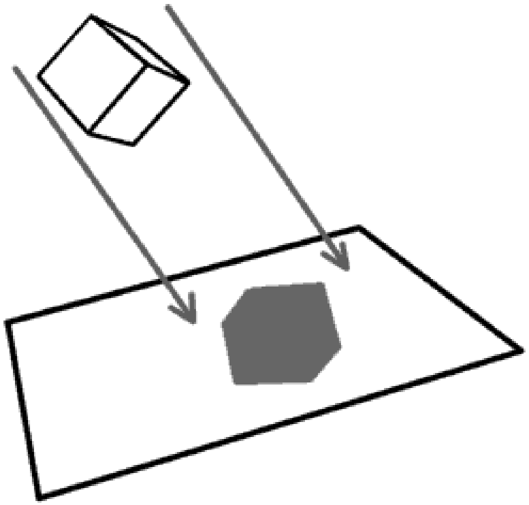
Lecture No.16 3D Concepts

Welcome! You are about to embark on a journey into the wondrous world of three-dimensional computer graphics. Before we take the plunge into esoteric 3D jargon and mathematical principles (as we will in the next lectures), let’s have a look at what the buzzword “3D” actually means.

We have heard the term “3D” applied to everything from games to the World Wide Web to Microsoft’s new look for Windows XP. The term 3D is often confusing because games (and other applications) which claim to be 3D, are not really 3D. In a 3D medium, each of our eyes views the scene from slightly different angles. This is the way we perceive the real world. Obviously, the flat monitors most of us use when playing 3D games 3D applications can’t do this. However, some Virtual Reality (VR) glasses have this capability by using a separate TV-like screen for each eye. These VR glasses may become common place some years from now, but today, they are not the norm. Thus, for present-day usage, we can define “3D to mean “something using a three-dimensional coordinate system.”

A three-dimensional coordinate system is just a fancy term for a system that measures objects with width, height, and depth (just like the real world). Similarly, 2-dimensional coordinate systems measure objects with width and height --- ignoring depth properties (so unlike the real world).





Shadow of a 3D object on paper

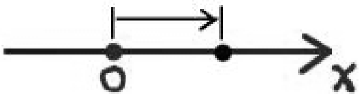
16.1 Coordinate Systems

Coordinate systems are the measured frames of reference within which geometry is defined, manipulated and viewed. In this system, you have a well-known point that 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).

With the Cartesian coordinate system, you can define any point in space by saying how far along each of the three axes you need to travel in order to reach the point if you start at the origin.

Following are three types of the coordinate systems.

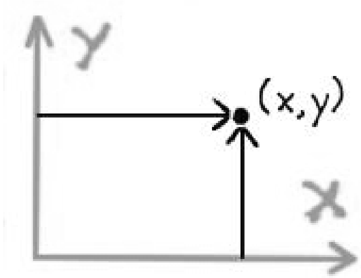
a) 1-D Coordinate Systems:



This system has the following characteristics:

- Direction and magnitude along a single axis, with reference to an 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

b) 2-D Coordinate Systems:

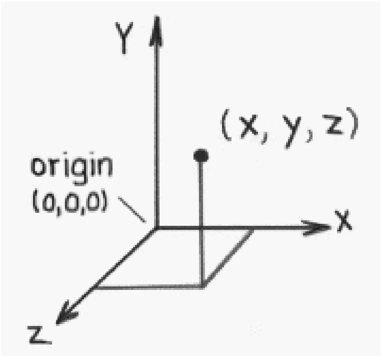


- Direction and magnitude along two axes, with reference to an origin
- Locations are defined by x, y coordinate pairs

Important

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

c) 3-D Coordinate Systems:

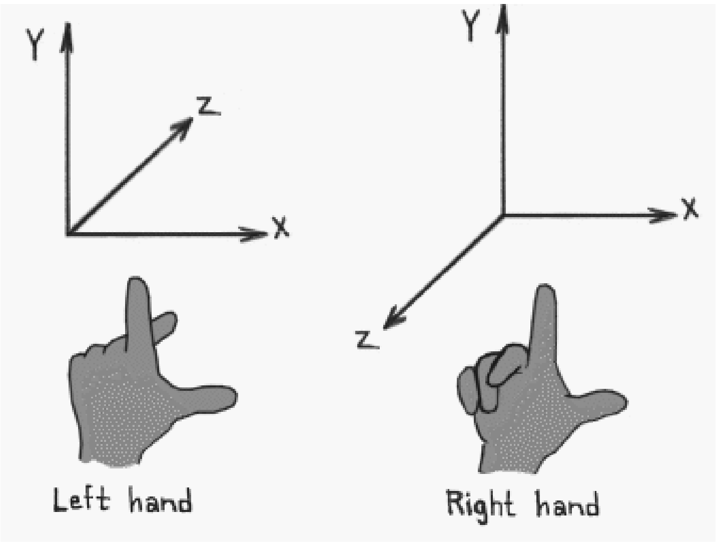


- 3D Cartesian coordinate systems
- Direction and magnitude along three axes, with reference to an origin
- Locations are defined by x, y, z triples

Important

- 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

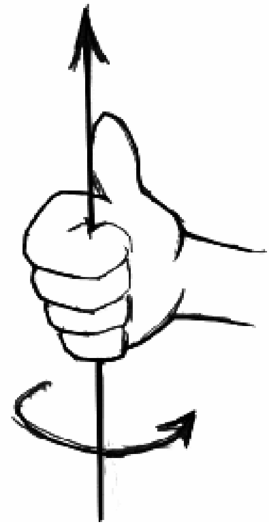
16.2 Left-handed versus Right-handed



- Determines 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

a) Right Handed Rule:

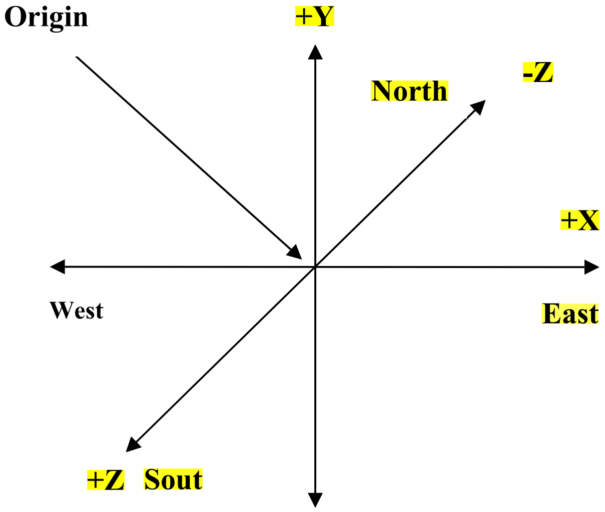
"Right Hand Rule" for rotations: grasp axis with right hand with thumb oriented in positive direction, fingers will then curl in direction of positive rotation for that axis.



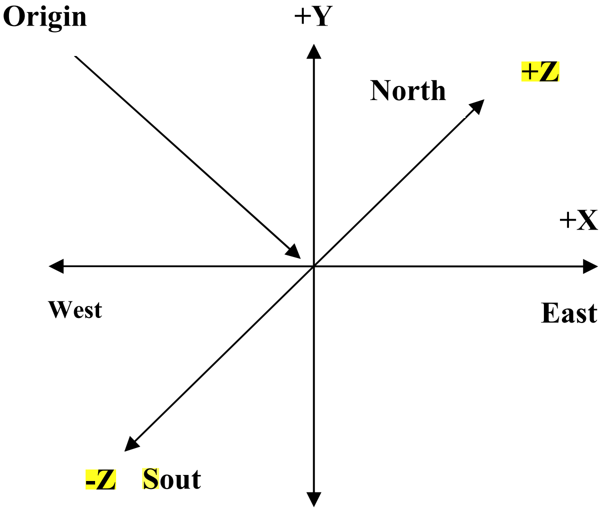
Right handed Cartesian coordinate system describes the relationship of the X,Y, and Z in the following manner:

Important

- 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.



b) Left Handed Rule:



Left handed Cartesian coordinate system describes the relationship of the X, Y and Z in the following manner:

Important

- 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.

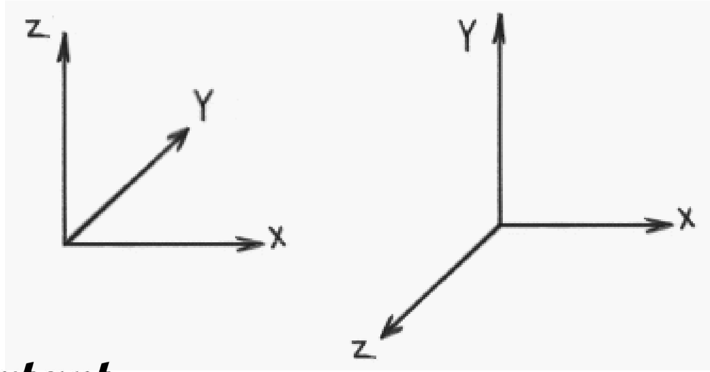
Defining 3D points in mathematical notations

3D points can be described using simple mathematical notations

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).

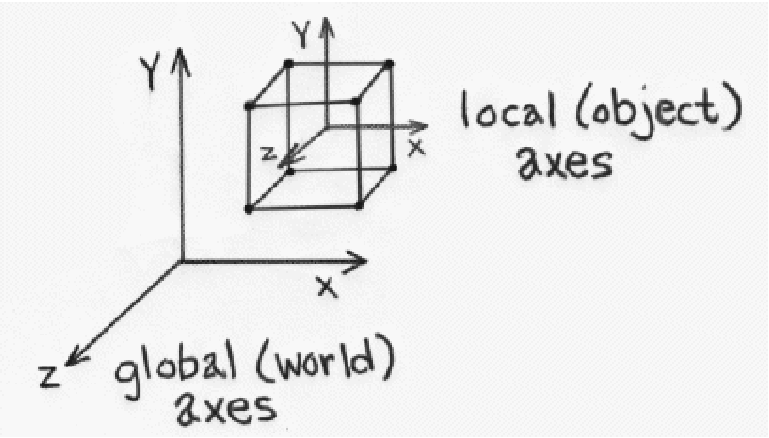
Y-up versus Z-up:



Important

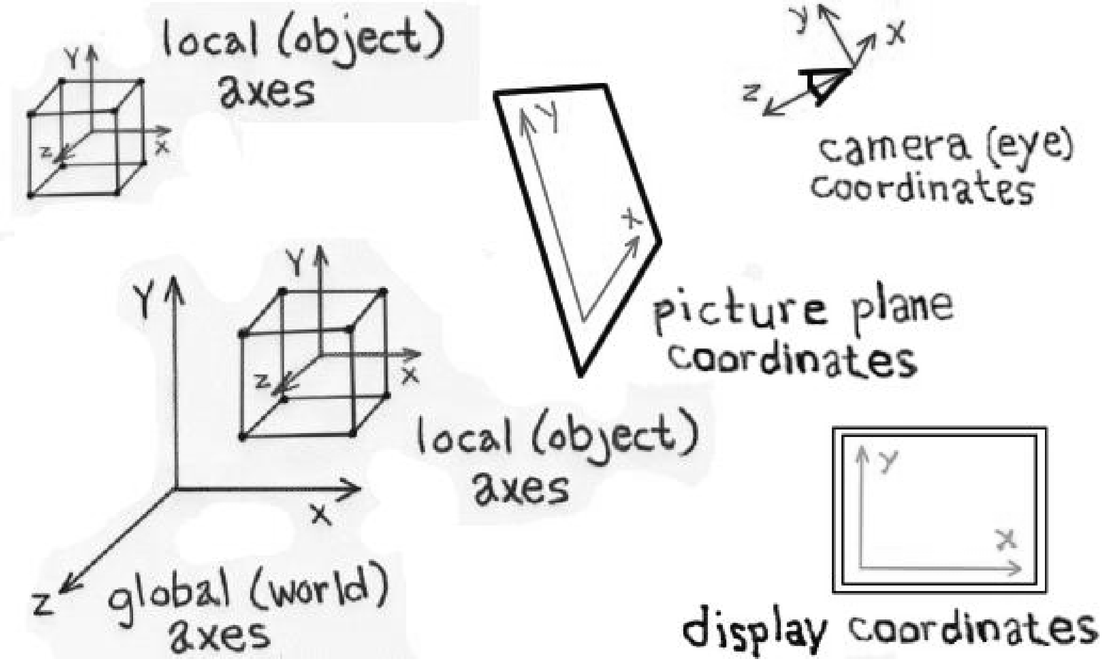
- z-up typically used by designers
- y-up typically used by animators
- orientation by profession supposedly derives from past work habits
- often handled differently when moving from application to application

16.3 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 another.

16.4 Multiple Frames of Reference in a 3-D Scene:



- In fact, there usually are multiple coordinate systems within any 3-D screen

- Application data will be transformed among the various coordinate systems, depending on what's to be accomplished during program execution

- Individual coordinate systems often are hierarchically linked within the scene

16.5 Defining points in C language structure

You can now define any point in the 3D by saying how far east, up, and north it is from your origin. The center of your computer screen ? it would be at a point such as "1.5 feet east, 4.0 feet up, 7.2 feet north." Obviously, you will want a data structure to represent these points. An example of such a structure is shown in this code snippet:

```
typedef struct _POINT3D
{
    float x;
    float y;
    float z;
}POINT3D;

POINT3D screenCenter = {1.5, 4.0, 7.2};
```

16.6 The Polar Coordinate System

Cartesian systems are not the only ones we can use. We could have also described the object position in this way: "starting at the origin, looking east, rotate 38 degrees northward, 65 degrees upward, and travel 7.47 feet along this line. "As you can see, this is less intuitive in a real world setting. And if you try to work out the math, it is harder to manipulate (when we get to the sections that move points around). Because such polar coordinates are difficult to control, they are generally not used in 3D graphics.

16.7 Using Multiple Coordinate Systems

As we start working with 3D objects, you may find that it is more efficient to work with groups of points instead of individual single points. For example, if you want to model your computer, you may want to store it in a structure such as that shown in this code snippet:

```
typedef struct _CPU{

        POINT3D center;       // the center of the CPU, in World coordinates
        POINT3D coord[8];   // the 8 corners of the CPU box relative to the center point

}CPU;
```

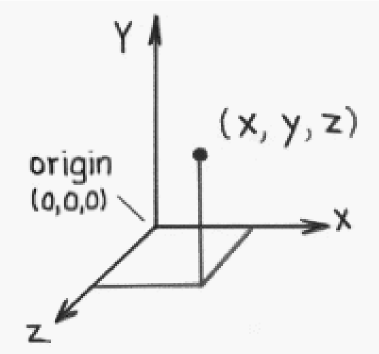
In next lectures we will learn how we can show 3D point on 2D computer screen.

16.8 Defining Geometry in 3-D

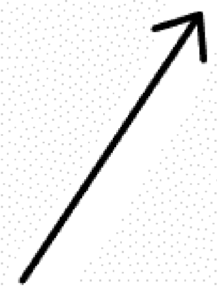
Here are some definitions of the technical names that will be used in 3D lectures.

Modeling: is the process of describing an object or scene so that we can construct an image of it.

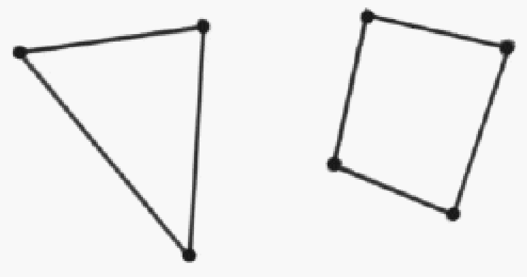
Points & Polygons:



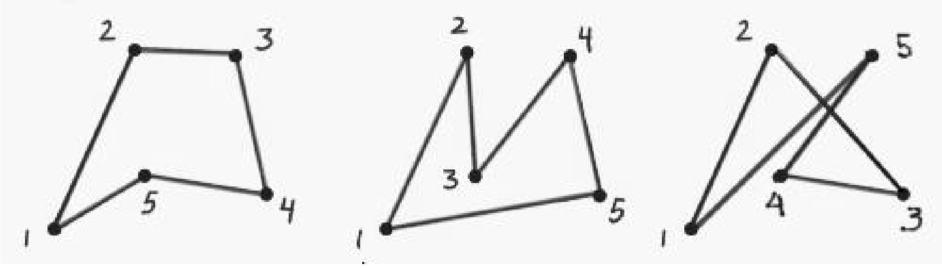
- Points: three-dimensional locations (or coordinate triples)



- Vectors: - have direction and magnitude; can also be thought of as displacement



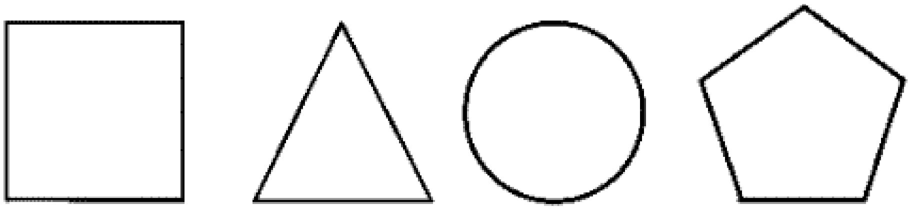
- Polygons: - sequences of "correctly" co-planar points; or an initial point and a sequence of vectors



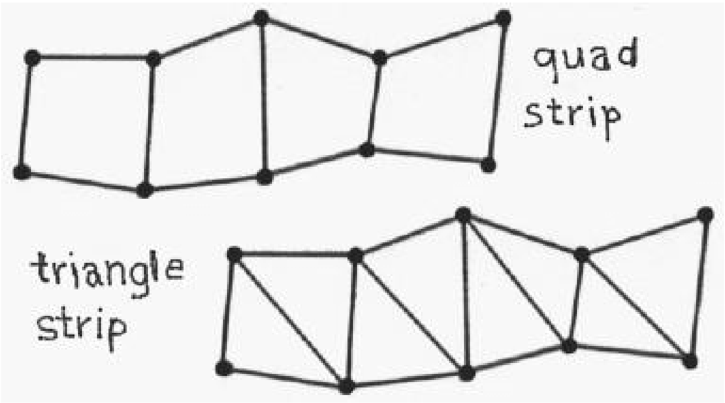
Primitives

Primitives are the fundamental geometric entities within a given data structure.

- We have already touched on point, vector and polygon primitives



- Regular Polygon Primitives - square, triangle, circle, n-polygon, etc.



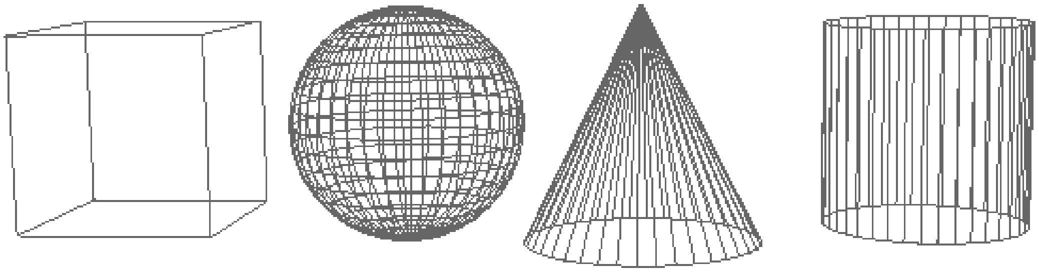
- Polygon strips or meshes

- Meshes provide a more economical description than multiple individual polygons

  For example, 100 individual triangles, each requiring 3 vertices, would require 100 x 3 or 300 vertex definitions to be stored in the 3-D database.

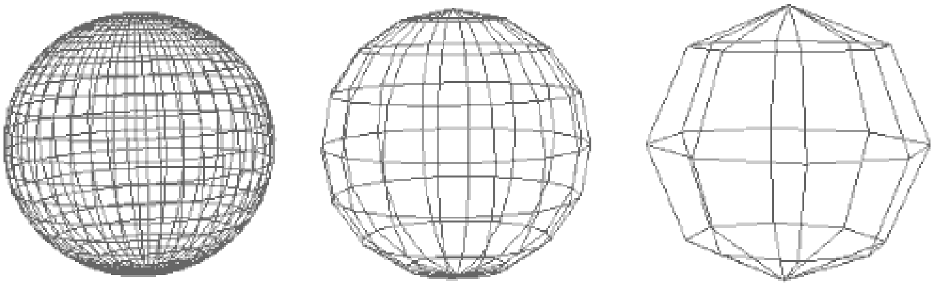
  By contrast, triangle strips require n + 2 vertex definitions for any n number or triangles in the strip. Hence, a 100 triangle strip requires only 102 unique vertex definitions.

- 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.

Important

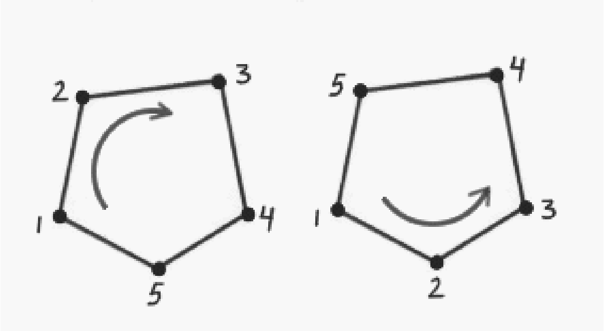
Rendering - The process of computing a two dimensional image using a combination of a three-dimensional database, scene characteristics, and viewing transformations. Various algorithms can be employed for rendering, depending on the needs of the application.

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.

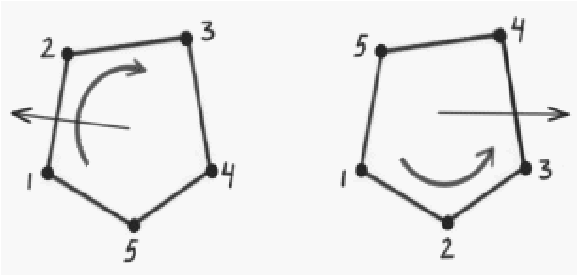
Sampling - The process of selecting a representative but finite number of values along a continuous function sufficient to render a reasonable approximation of the function for the task at hand.

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.

Polygons and rendering

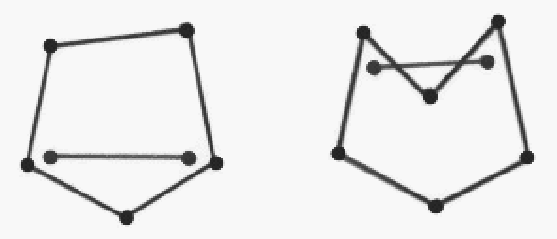


- Clockwise versus counterclockwise

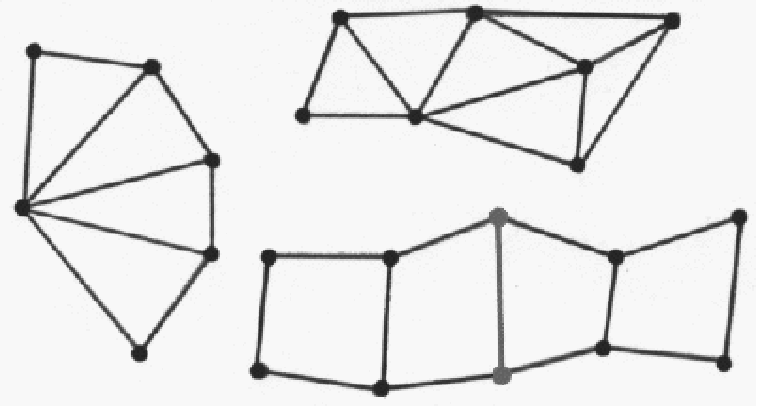


Surface normal - a vector that is perpendicular to a surface and "outward" facing

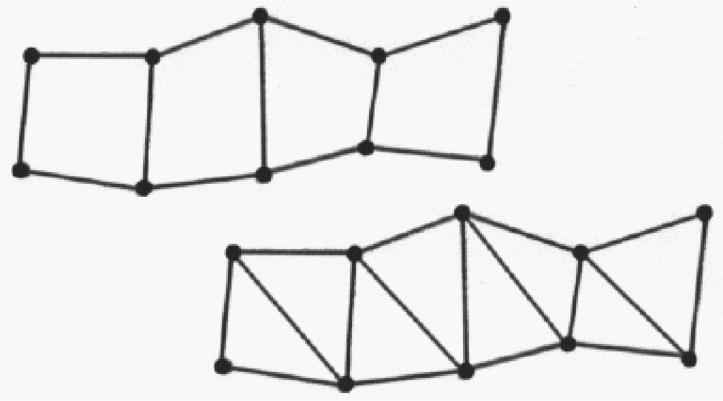
- Surface normals are used to determine visibility and in the calculation of shading values (among other things)



- Convex versus concave
  - A shape is convex if any two points within the shape can be connected with a straight line that never goes out of the shape. If not, the shape is concave.
  - Concave polygons can cause problems during rendering (e.g. tears, etc., in apparent surface).



- Polygon meshes and shared vertices



- Polygons consisting of non-co-planar vertices can cause problems when rendering (e.g. visible tearing of the surface, etc.)

- With quad meshes, for example, vertices within polygons can be inadvertently transformed into non-co-planer positions during modeling or animation transformations.

- With triangle meshes, all polygons are triangles and therefore all vertices within any given polygon will be coplanar.

With polygonal databases:

- Explicit, low-level descriptions of geometry tend to be employed

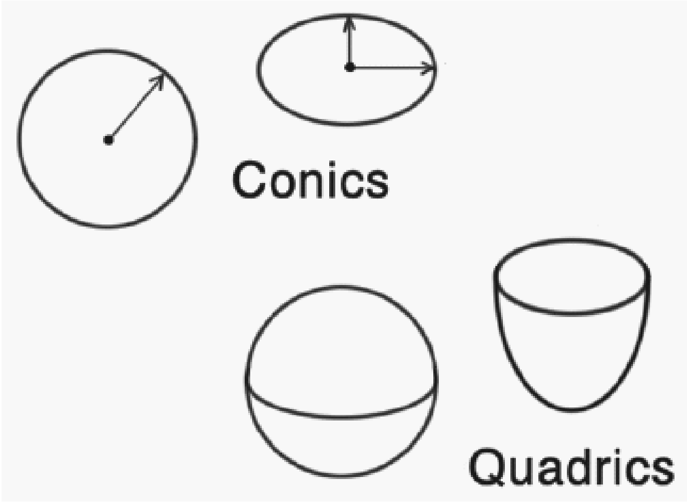
- Object database files can become very large relative to more economical, higher order descriptions.

- Organic forms or free-form surfaces can be difficult to model.

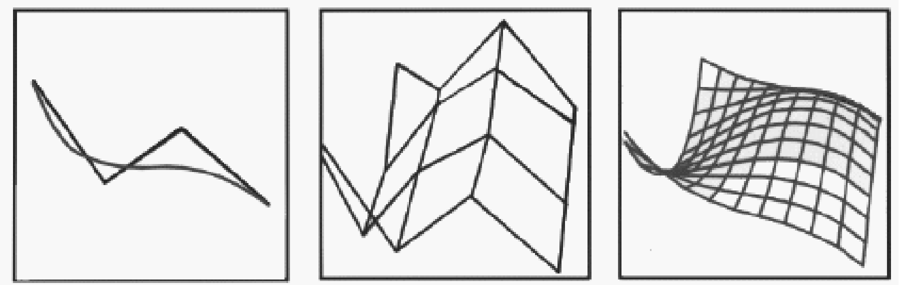
16.9 **Surface models**

Here is brief over view of surface models:

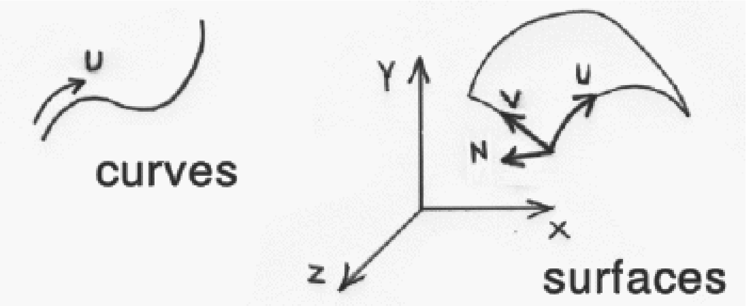
- Surfaces can be constructed from mathematical descriptions
- Resolution independent - surfaces can be tessellated at rendering with an appropriate level of approximation for current display devices and/or viewing parameters
- Tessellation can be adaptive to the local degree of curvature of a surface.



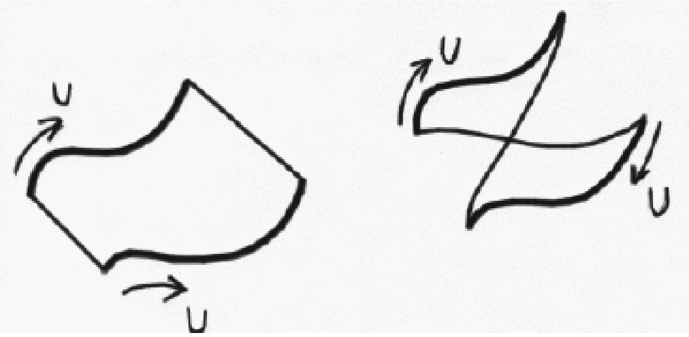
- Primitives



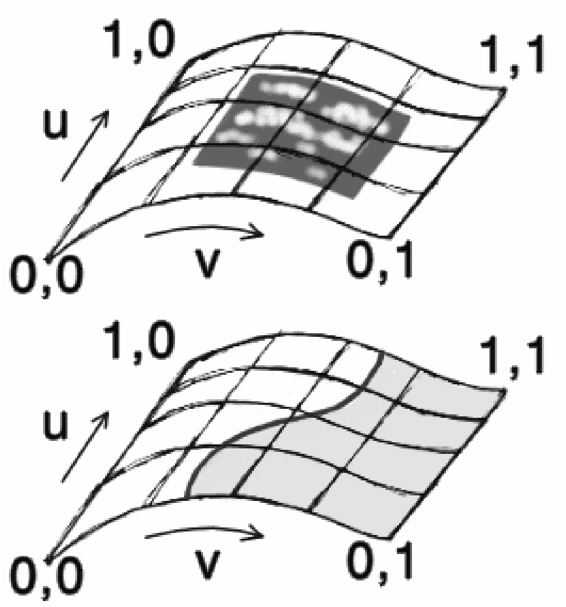
- Free-form surfaces can be built from curves
- Construction history, while also used in polygonal modeling, can be particularly useful with curve and surface modeling techniques.



- Parameterization



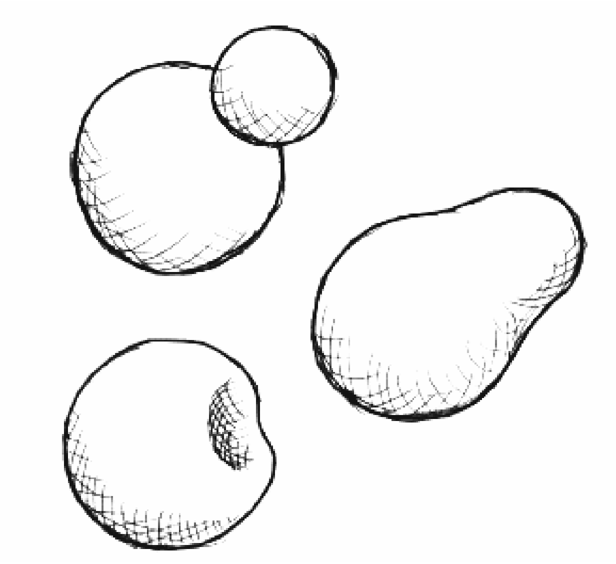
- Curve direction and surface construction



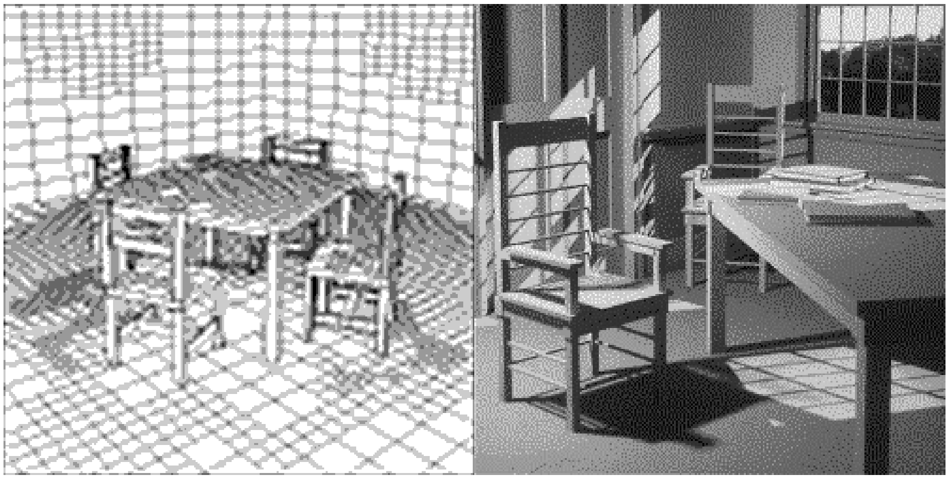
- Surface parameterization (u, v, w) are used
    - For placing texture maps, etc.
    - For locating trimming curves, etc.

Metaballs (blobby surfaces)

- Potential functions (usually radially symmetric Gaussian functions) are used to define surfaces surrounding points



Lighting Effects



Texture Mapping:

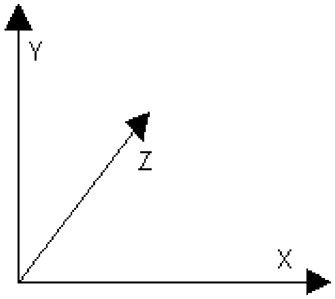
The texture mapping is of the following types that we will be studying in our coming lectures on 3D:

1. Perfect Mapping:
2. Affine Mapping
3. Area Subdivision
4. Scan-line Subdivision
5. Parabolic Mapping
6. Hyperbolic Mapping
7. Constant-Z Mapping

Lecture No.17 3D Transformations I

Definition of a 3D Point

A point is similar to its 2D counterpart; we simply add an extra component, Z, for the 3rd axis:



Points are now represented with 3 numbers: <x, y, z>. This particular method of representing 3D space is the "left-handed" coordinate system. In the left-handed system the x axis increases going to the right, the y axis increases going up, and the z axis increases going into the page/screen. The right-handed system is the same but with the z-axis pointing in the opposite direction.

Distance between Two 3D Points

The distance between two points <Ax,Ay,Az> and <Bx,By,Bz> can be found by again using the Pythagoras theorem:

dx = Ax-Bx
dy = Ay-By
dz = Az-Bz
distance = sqrt(dx*dx + dy*dy + dz*dz)

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>.

3D Vector addition and subtraction is virtually identical to the 2D case. You can add a 3D vector <vx,vy,vz> to a 3D point <x,y,z> to get the new point <x',y',z'> like so:

x' = x + vx
y' = y + vy
z' = z + vz

Vectors themselves can be added by adding each of their components, or they can be multiplied (scaled) by multiplying each component by some constant k (where k <> 0). Scaling a vector by 2 (say) will still cause the vector to point in the same direction, but it will now be twice as long. Of course you can also divide the vector by k (where k <> 0) to get a similar result.

To calculate the length of a vector we simply calculate the distance between the origin and the point at <x, y, z>:

Length = | <x,y,z> - <0,0,0> |
= sqrt((x-0)*(x-0) + (y-0)*(y-0) + (z-0)*(z-0))
= sqrt(x*x + y*y + z*z)

Unit Vector

Often in 3D computer graphics you need to convert a vector to a unit vector, ie a vector that points in the same direction but has a length of 1.

This is done by simply dividing each component by the length:

Let <x,y,z> be our vector, length = sqrt(x*x + y*y + z*z)

$$\textbf{Unit vector} = \frac{<x,y,z>}{\textbf{length}} = \left| \frac{x}{\textbf{length}}, \frac{y}{\textbf{length}}, \frac{z}{\textbf{length}} \right|$$

(Where 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.

Definition of a Line

As in 2D, we can represent a line by it's endpoints (P1 and P2) or by the parametric equation:

P = P1 + k * (P2-P1)

Where k is some scalar value between 0 and 1

Transformations:

A static set of 3D points or other geometric shapes on screen is not very interesting. You could just use a paint program to produce one of these. To make your program interesting, you will want a dynamic landscape on the screen. You want the points to move in the world coordinate system, and you even want the point-of-view (POV) to move. In short, you want to model the real world. *The process of moving points in space is called transformation*, and can be divided into *translation, rotation and other kind of transformations.*

Translation

Translation is used to move a point, or a set of points, linearly in space, for example, you may want to move a point “3 meters east, -2 meters up, and 4 meters north.” Looking at this textual description, you might think that this looks very much like a Point3D, and you would be close. But the above does not require one critical piece of information: it does not reference the origin. The above only encapsulates direction and distance, not an absolute point in space. This called a vector and can be represented in a structure identical to Point3D:

```
struct Vector3D
        float x;        distance along x axes
        float y;        distance along y axes
        float z;        distance along z axes
end struct
```

Vector Addition

You translate a point by adding a vector to it; you add points and vectors by adding the components piecewise:

```
Point3D point = {0, 0, 0}
Vector3D vector = {10, -3, 2.5 }
```

Adding vector to point

```
point.x = point.x + vector.x;
point.y = point.y + vector.y;
point.z = point.z + vector.z;
```

Point will be now at the absolute point < 10,-3 2.5>. you could move it again:

```
point.x = point.x + vector.x;
point.y = point.y + vector.y;
point.z = point.z + vector.z;
```

And point would now be at the absolute point <20, -6, 5>.

In pure mathematical sense, you cannot add two points together – such an operation makes no sense (what is Lahore plus Karachi?). However, you can subtract a point from another in order to uncover the vector that would have to be added to the first to translate it into the second:

```
Point3D p1,p2
Vector3D v;
```

Set p1 and p2 to the desired points

```
v.x = p2.x – p1.x
v.y = p2.y – p1.y
v.z = p2.z – p1.z
```

Now you can add v to p1, you would translate it into the point p2.

The following lists the operations you can do between points and vectors:

point – point => vector
point + point = point - (- point) => vector
vector – vector => vector
vector + vector => vector
point – vector = point + (-vector) => point
point + vector => point

Multiplying: Scalar Multiplication

Multiplying a vector by a scalar (a number with no units), and could be coded with:

```
Vector.x  =    Vector.x  * scalarValue
Vector.y  =    Vector.y  * scalarValue
Vector.z  =    Vector.z  * scalarValue
```

If you had a vector with a length of 4 and multiplied it by 2.5, you would end up with a vector of length 10 that points in the same direction the original vector pointed. If you multiplied by -2.5 instead, you would still end up with a vector of length 10; but now it would be pointing in the opposite direction of the original vector.

Multiplying: Vector Multiplication

You can multiply with vectors two other ways; both involve multiplying a vector by a vector.

Dot Product

The dot product of two vectors is defined by the formula:
Vector A, B

A * B = A.x * B.x + A.y * B.y + A.z * B.z

The result of a dot product is a number and has units of A's units times B's units. Thus, if you calculate the dot product for two vectors that both use feet for units, your answer will be in square feet. However, in 3D graphics we usually ignore the units and just treat it like a scalar.
Consider the following definition of the dot product that is used by physicists (instead of mathematicians):

A * B = |A| * |B| * cos(theta)

Where theta is the angle between the two vectors

Remember that |v| represents the length of vector V and is a non-negative number; we can replace the vector lengths above and end up with:

K = |A| * |B| (therefore k > = 0)

A * B = K * cos (theta)

Therefore:

A * B => cos(theta)

Where "=>" means "directly correlates to." Now, if you remember, the cos(theta) function has the following properties:

Important

cos(theta) > 0 iff theta is less than 90 degrees or greater than 270 degrees
cos(theta) < iff theta is greater than 90 degrees and less than 270 degrees
cos(theta) = 0 iff theta is 90 degrees or 270 degrees

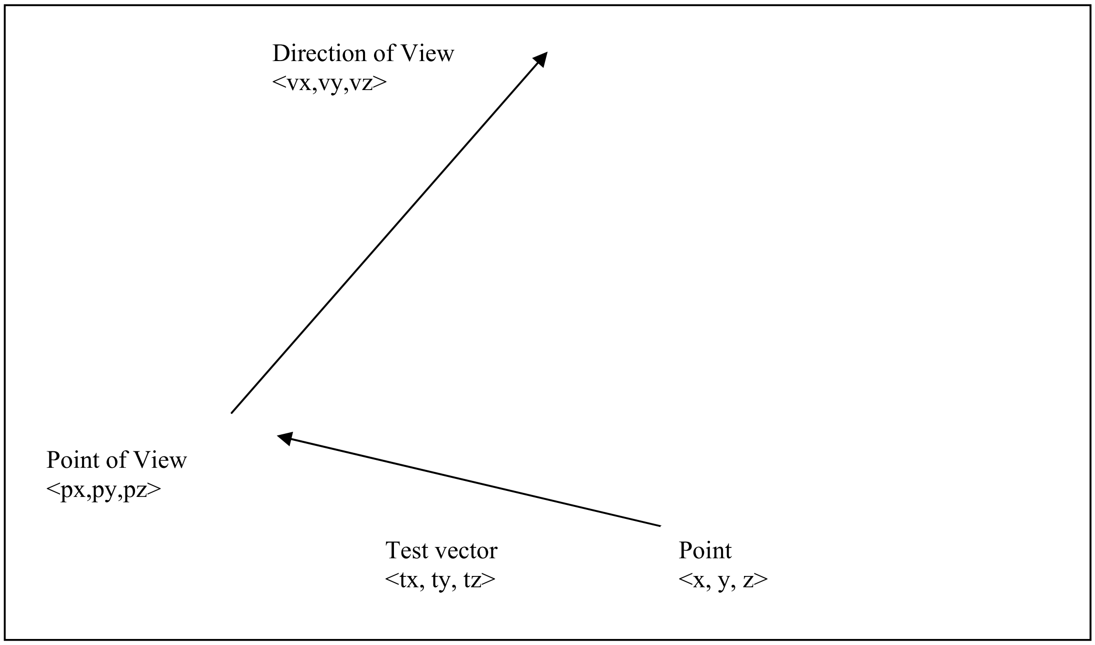
We can extend this to the dot product of two vectors, since it directly correlates to the angle between the two vectors:

Important

A*B > 0 iff the angle between them is less than 90 or greater than 270 degrees
A*B < 0 iff the angle between them is greater than 90 and less than 270 degrees
A*B = 0 iff the angle between them is 90 or 270 degrees (they are orthogonal).

Use of Dot Product

Assume you have a point of view at < px,py,pz>. It is looking along the vector <vx,vy,vz>, and you have a point in space <x,y,z> you want to know if the point–of-view can possible see the point, of if the point is "behind " the POV, as shown in figure.



```
Point3D pov;
Vector3D povDir;
Point3D test;
Vector3D vTest
float dotProduct;
vTest.x = pov.x – test.x;
vTest.y = pov.y – test.y;
vTest.z = pov.z – test.z;
```

```
dotProduct == vTest.x*povDir.x  + vTest.y*povDir.y + vTest.z * povDir.z;

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

Another kind of multiplication that you can do with vectors is called the cross product this is defined as:
Vector A, B

A X 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 >

For physicists:

|A x B| = |A| * |B| sin(theta)

Where theta is the angle between the two vectors.

The above formula for A x B came from the determinate of order 3 of the matrix:

```
|  X      Y      Z |
|A.x    A.y    A.z|
|B.x    B.y    B.z|
```

Important

Transformations

The process of moving points in space is called transformation.

Types of Transformation

There are various types of transformations as we have seen in case of 2D transformations. These include:

a) Translation
b) Rotation
c) Scaling
d) Reflection
e) Shearing

Translation

Translation is used to move a point, or a set of points, linearly in space. Since now we are talking about 3D, therefore each point has 3 coordinates i.e. x, y and z. similarly, the translation distances can also be specified in any of the 3 dimensions. These Translation Distances are given by tx, ty and tz.

For any point P(x,y,z) after translation we have P′(x′,y′,z′) where

$x' = x + tx$,
$y' = y + ty$,

z′ = z + tz
and (tx, ty , tz) is Translation vector

Now this can be expressed as a single matrix equation:

$$P' = P + T$$

Where:

$$P = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \qquad P' = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} \qquad T = \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$$

3D Translation Example
We may want to move a point "3 meters east, -2 meters up, and 4 meters north." What would be done in such event?

Steps for Translation
Given a point in 3D and a translation vector, it can be translated as follows:

```
        Point3D point = (0, 0, 0)
        Vector3D vector = (10, -3, 2.5)
Adding vector to point
        point.x = point.x + vector.x;
        point.y = point.y + vector.y;
        point.z = point.z + vector.z;
And finally we have translated point.
```

Homogeneous Coordinates

Analogous to their 2D Counterpart, the homogeneous coordinates for 3D translation can be expressed as :

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

Abbreviated as:

$$P' = T\,(tx, ty, tz).\ P$$

On solving the RHS of the matrix equation, we get:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ z + t_z \\ 1 \end{bmatrix}$$

Which shows that each of the 3 coordinates gets translated by the corresponding translation distance.

Lecture No.18 3D Transformations II

Rotation

Rotation is the process of moving a point in space in a non-linear manner. More particularly, it involves moving the point from one position on a sphere whose center is at the origin to another position on the sphere. Why would you want to do something like this? As we will show in later section, allowing the point of view to move around is only an illusion – projection requires that the POV be at the origin. When the user thinks the POV is moving, you are actually translating all your points in the opposite direction; and when the user thinks the POV is looking down a new vector, you are actually rotating all the points in the opposite direction; and when the user thinks the POV is looking down a new vector, you are actually rotating all the points in the opposite direction.

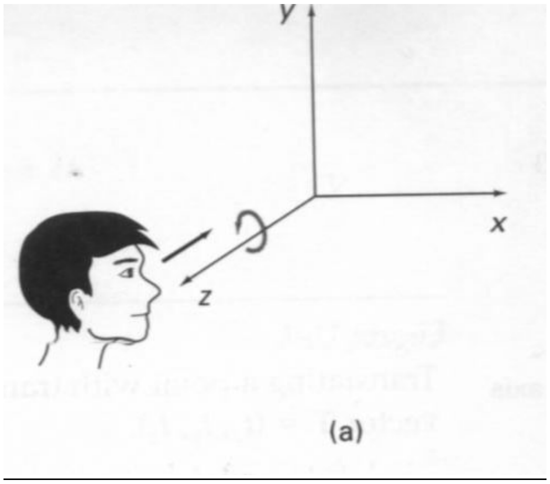
Normalization: Note that this process of moving your points so that your POV is at the origin looking down the +Z axis is called normalization.

Rotation a point requires that you know
the coordinates for the point, and
That you know the rotation angles.

You 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 plane, or "yaw"); and how far to rotate around the Z axis (XY rotation, or "roll"). Conceptually, you do the three rotations separately. First, you rotate around one axis, followed by another, then the last. The order of rotations is important when you cascade rotations; we will rotate first around the Z axis, then around the X axis, and finally around the Y axis.

To show how the rotation formulas are derived, let's rotate the point <x,y,z> around the Z axis with an angle of θ degrees.

ROLL:-



If you look closely, you should note that when we rotate around the Z axis, the Z element of the point does not change. In fact, we can just ignore the Z – we already know what it will be after the rotation. If we ignore the Z element, then we have the same case as if we were rotating the two-dimensional point <x,y> through the angle θ.

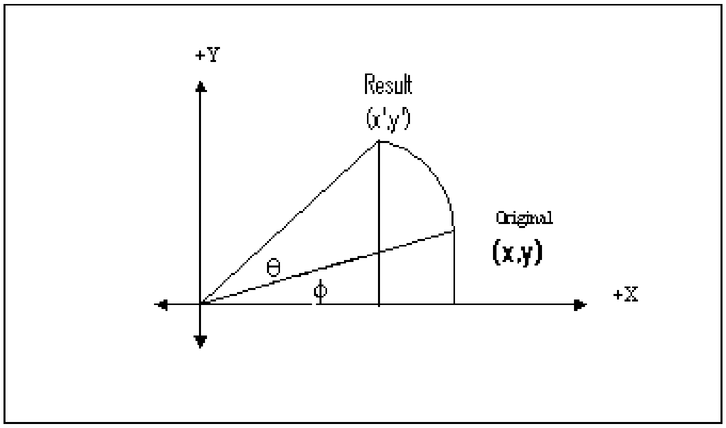
This is the way to rotate a 2-D point. For simplicity, consider the pivot at origin and rotate point P (x,y) where $x = r \cos\Phi$ and $y = r \sin\Phi$

If rotated by θ then:

$x' = r\cos(\Phi + \theta)$
$\quad = r\cos\Phi\cos\theta - r\sin\Phi\sin\theta$

and

$y' = r\sin(\Phi + \theta)$
$\quad = r\cos\Phi\sin\theta + r\sin\Phi\cos\theta$



Replacing $r\cos\Phi$ with x and $r\sin\Phi$ with y, we have:

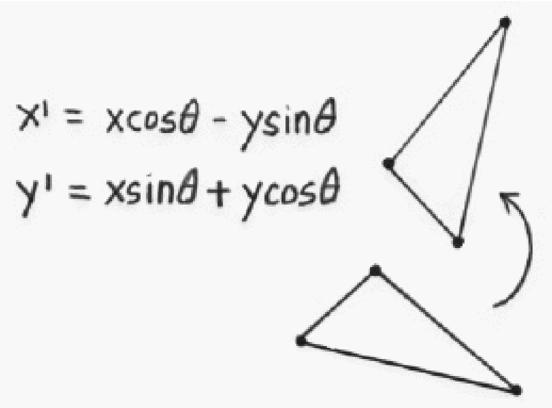
$x' = x\cos\theta - y\sin\theta$

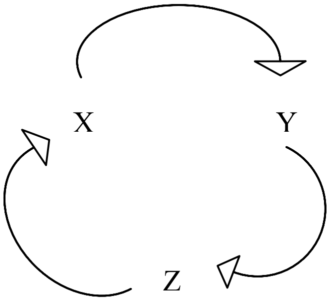
and

$y' = x\sin\theta + y\cos\theta$

and

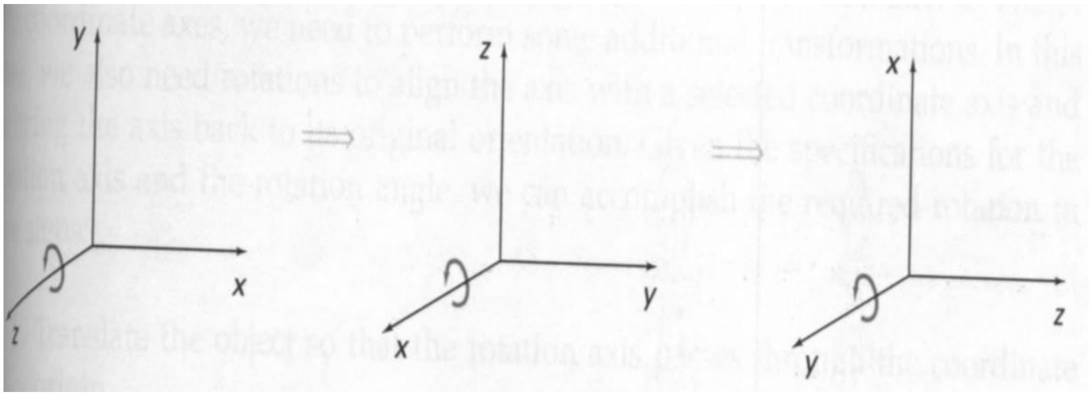
$z' = z$ (as it does not change when rotating around z-axis)





Now for rotation around other axes, cyclic permutation helps form the equations for yaw and pitch as well:

In the above equations replacing x with y and y with z gives equations for rotation around x-axis. Now in the modified equations if we replace y with z and z with x then we get the equations for rotation around y-axis.



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):

$x' = z \sin\theta + x \cos\theta$
$y' = y$
$z' = z \cos\theta - x \sin\theta$

Using Matrices to create 3D

A matrix is usually defined as a two-dimensional array of numbers. However, I think you will find it much more useful to think of a matrix as an array of vectors. When we talk about vectors, what it really mean is an ordered set of numbers (a tuple in mathematics terms). We can use 3D graphics vectors and points interchangeably for this, since they are both 3-tuples (or triples).

In general we work with “square” matrices. This means that the number of vectors in the matrix is the same as the number of elements in the vectors that comprise it. Mathematically, we show a matrix as a 2-D array of numbers surrounded by vertical lines. For example:

$$\begin{vmatrix} x1 & y1 & z1 \\ x2 & y2 & z2 \\ x3 & y3 & z3 \end{vmatrix}$$

we designate this as a 3*3 matrix (the first 3 is the number of rows, and the second 3 is the number of columns).

The “rows” of the matrix are the horizontal vectors that make it up; in this case, <x1, y1,z1>, <x2,y2,z2>, and <x3,y3,z3>. In mathematics, we call the vertical vectors “columns.” In this case they are < x1,x2,x3>, <y1,y2,y3> and <z1,z2,z3>.

The most important thing we do with a matrix is to multiply it by a vector or another matrix. We follow one simple rule when multiplying something by a matrix: multiply each column by a multiplicand and store this as an element in the result. Now as I said earlier, you can consider each column to be a vector, so when we multiply by a matrix, we are just doing a bunch of vector multiplies. So which vector multiply do you use-the dot product, or the crosss product? You use the dot product.

We also follow on simple rule when multiplying a matrix by something: mubliply each ro by the multiplier. Again, rows are just vectors, and the type of ultiplicaiton is the dot product.

Let's look at some examples. First, let's assume that I have a matrix M, and I want to multiply it by a point < x,y,z>, the first ting I know is that the vector rows of the matrix must contain three elements (in other words, three columns). Why ? because I have to multiply those rows by my point using a dot product, and to do that, the two vectors must have the same number of element. Since I am going to get dot product for each row in M, I will end up with a tuple that has one element for each row in M. as I stated earlier, we work almost exclusively with square matrices, since I must have three columns, M will also have three rows. Lets see:

$$< x,y,z> * \begin{vmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{vmatrix} =\{<x,y,z>*< 1,0,0> ,<x,y,z><0,1,0>,<x,y,z> *<0,0,1>\}=\{ x,y,z\}$$

Using Matrices for Rotation Important

Roll (rotate about the Z axis):

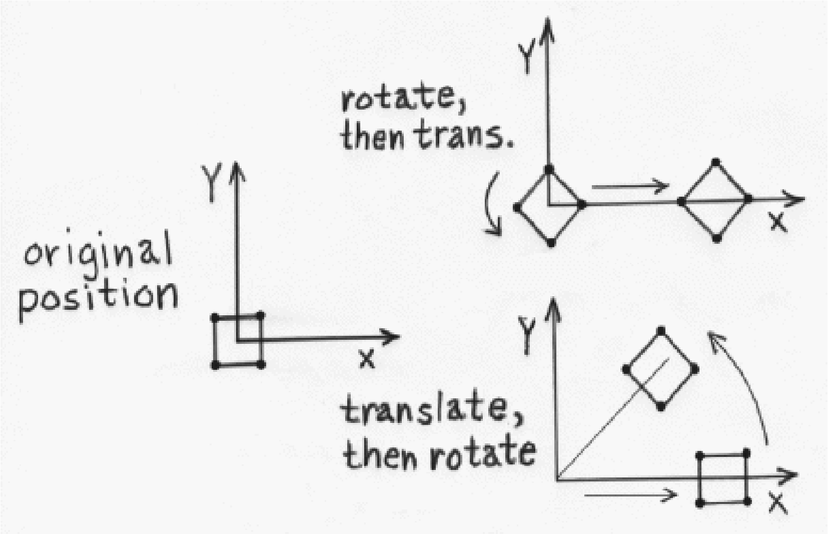
$$\begin{bmatrix}x'\\ y'\\ z'\\ 1\end{bmatrix} = \begin{bmatrix}\cos\theta & -\sin\theta & 0 & 0\\ \sin\theta & \cos\theta & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix} . \begin{bmatrix}x\\ y\\ z\\ 1\end{bmatrix}$$

Pitch (rotate about the X axis):

$$\begin{bmatrix}x'\\ y'\\ z'\\ 1\end{bmatrix} = \begin{bmatrix}1 & 0 & 0 & 0\\ 0 & \cos\theta & -\sin\theta & 0\\ 0 & \sin\theta & \cos\theta & 0\\ 0 & 0 & 0 & 1\end{bmatrix} . \begin{bmatrix}x\\ y\\ z\\ 1\end{bmatrix}$$

Yaw (rotate about the Y axis): Important

$$\begin{bmatrix}x'\\ y'\\ z'\\ 1\end{bmatrix} = \begin{bmatrix}\cos\theta & 0 & \sin\theta & 0\\ 0 & 1 & 0 & 0\\ -\sin\theta & 0 & \cos\theta & 0\\ 0 & 0 & 0 & 1\end{bmatrix} . \begin{bmatrix}x\\ y\\ z\\ 1\end{bmatrix}$$



Example:

To show this happening, let's manually rotate the point <2,0,0> 45 degrees clockwise about the z axis.

$$\mathbf{v}' = R_z(45)\mathbf{v}$$

$$\mathbf{v}' = \begin{bmatrix} 0.707 & 0.707 & 0 \\ -0.707 & 0.707 & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 2 \\ 0 \\ 0 \end{bmatrix}$$

$$\mathbf{v}' = \begin{bmatrix} 2\times0.707+0\times0.707+0\times0 \\ 2\times-0.707+0\times0.707+0\times0 \\ 2\times0+0\times0+0\times0 \end{bmatrix}$$

$$\mathbf{v}' = \begin{bmatrix} 1.414 \\ -1.414 \\ 0 \end{bmatrix}$$

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:

$$\mathbf{v} = \mathbf{v}T(-\mathbf{p})$$

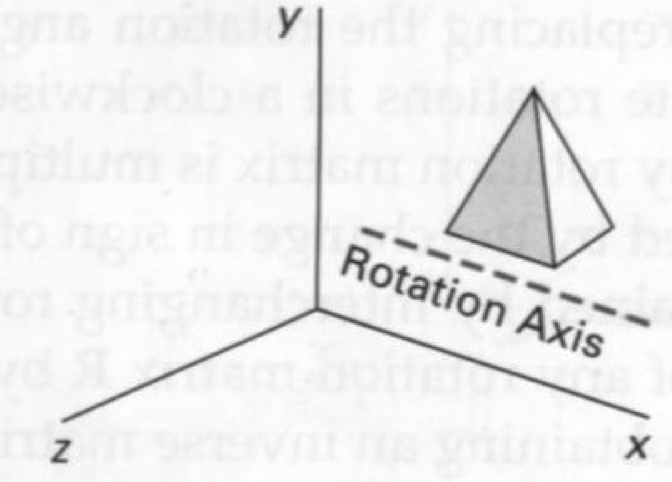
$$\mathbf{v} = \mathbf{v}R_z\left(\frac{\pi}{2}\right)$$

$$\mathbf{v} = \mathbf{v}T(\mathbf{p})$$

The first transformation moves a point such that it is situated about the world origin instead of being situated about the point **p**. The next one rotates it (remember, you can only rotate about the origin, not arbitrary points in space). Finally, after the point is rotated, you want to move it back so that it is situated about **p**. The final translation accomplishes this.

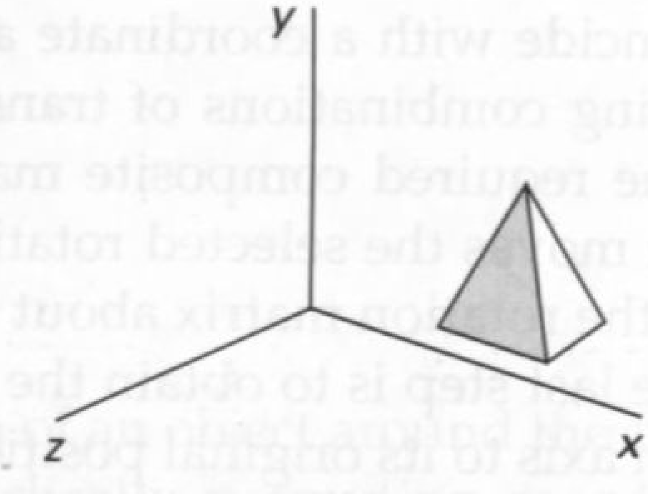
Rotation w.r.t. Arbitrary Axis:

If an object is required to be rotated with respect to a line acting as an axis of rotation, arbitrarily, then the problem is addressed using multiple transformations. Let us assume that such an arbitrary axis is parallel to one of the coordinate axes, say x-axis.



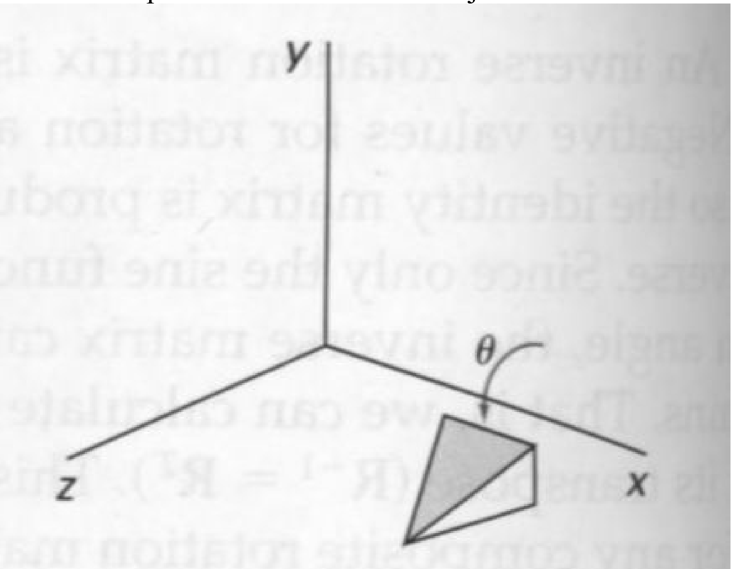
(a)
Original Position of Object

The first step in such case would be to translate the object such that the arbitrary axis coincides with the x-axis.



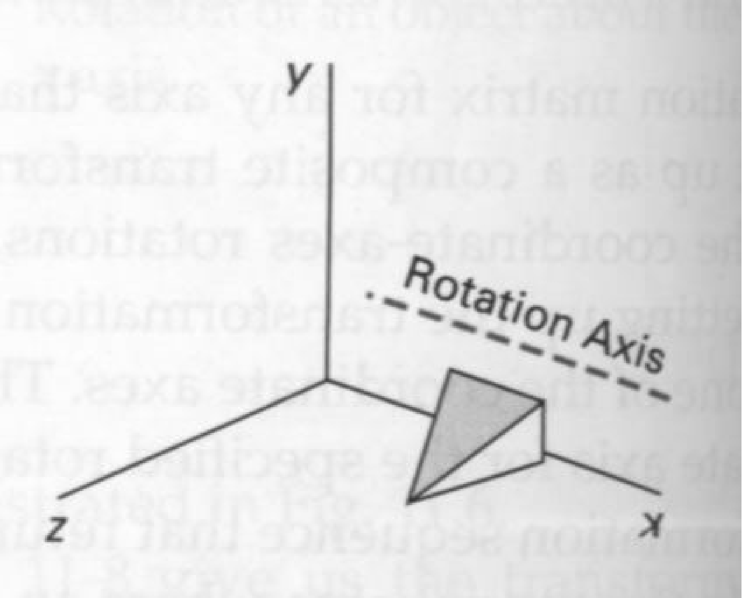
(b)
Translate Rotation Axis onto *x* Axis

The next step would be to rotate the object w.r.t. x-axis through angle θ.



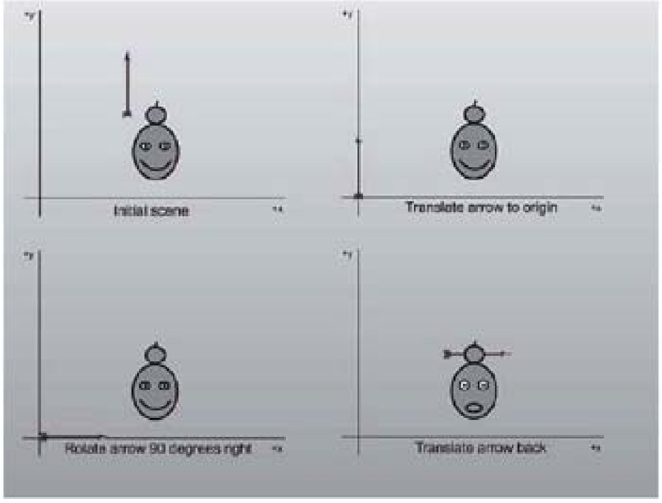
(c)
Rotate Object Through Angle θ

Then the object is translated such that the arbitrary axis gets back to its original position.

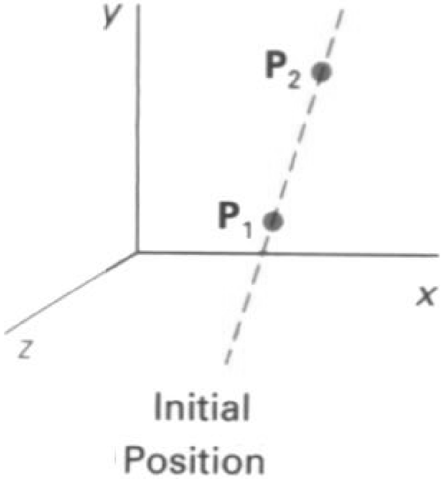


(d)
Translate Rotation
Axis to Original Position

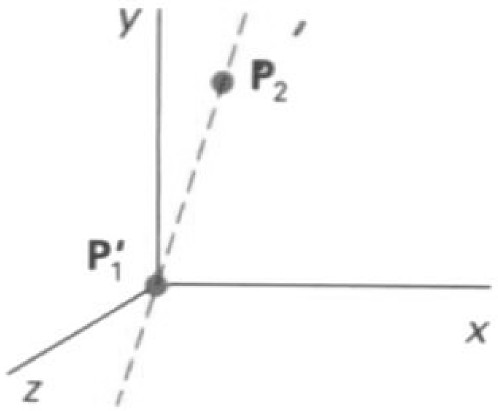
And thus the job is done.
An interesting usage of compound transformations:-



Now, if the arbitrary axis is not parallel to any of the coordinate axes, then the problem is slightly more difficult. It only adds to the number of steps required to get the job done. Let P1, P2 be the line arbitrary axis.

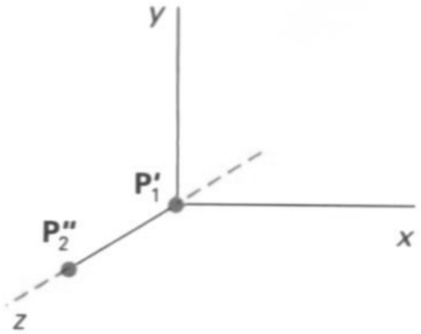


In the first step, the translation takes place that coincides the point P1 to the origin. Points after this step are P1' and P2'.

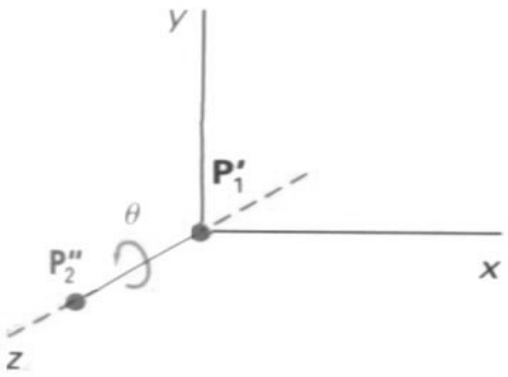


Step 1
Translate
$\mathbf{P}_1$ to the Origin

Now the arbitrary axis is rotated such that the point P2’ rotates to become P2’’ and lies on the z-axis.

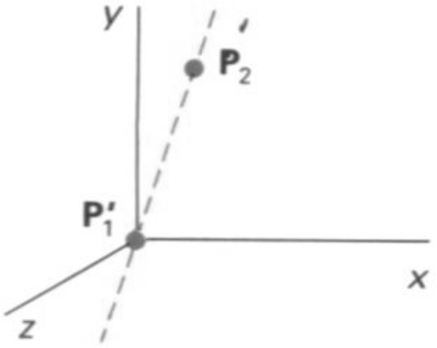


Step 2
Rotate $\mathbf{P}'_2$
onto the z Axis



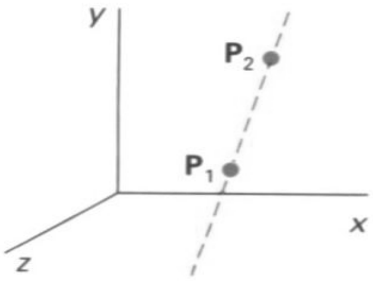
Step 3
Rotate the
Object Around the
z Axis

In the next step the object of interest is rotated around z-axis.



Step 4
Rotate the Axis
to the Original
Orientation

Now the object of interest is rotated about origin such that the arbitrary axis is poised like in above figure. Point P2'' gets back to its previous position P2'.



Step 5
Translate the
Rotation Axis
to the Original
Position

Finally the translation takes place to position the arbitrary axis back to its original position.

Scaling

Coordinate transformations for scaling relative to the origin are

X` = X . Sx

Y` = Y. Sy

Important

Z` = Z. Sz

Scaling an object with transformation changes the size of the object and reposition the object relative to the coordinate origin. If the transformation parameters are not all equal, relative dimensions in the object are changed.

Uniform Scaling : We preserve the original shape of an object with a uniform scaling (Sx = Sy = Sz)

Differential Scaling : We do not preserve the original shape of an object with a differential scaling (Sx <> Sy <> Sz)

Scaling relative to the coordinate Origin:

Scaling transformation of a position P = (x, y, z) relative to the coordinate origin can be written as

$$\begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scaling with respect to a selected fixed position:

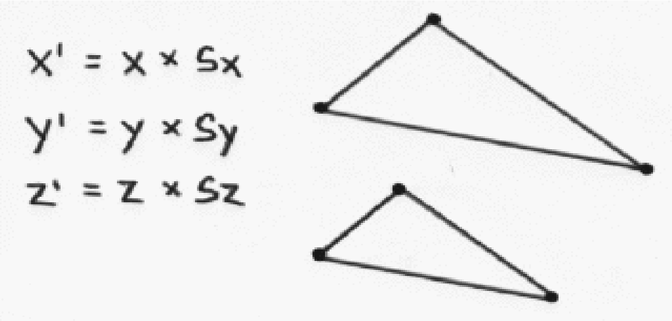
Scaling with respect to a selected fixed position (X_f,Y_f,Z_f) can be represented with the following transformation sequence:

Translate the fixed point to the origin.
Scale the object relative to the coordinate origin
Translate the fixed point back to its original position

For these three transformations we can have composite transformation matrix by multiplying three matrices into one

$$\begin{bmatrix} 1 & 0 & 0 & X_f \\ 0 & 1 & 0 & Y_f \\ 0 & 0 & 1 & Z_f \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & -X_f \\ 0 & 1 & 0 & -Y_f \\ 0 & 0 & 1 & -Z_f \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} S_x & 0 & 0 & (1-S_x)X_f \\ 0 & S_y & 0 & (1-S_y)Y_f \\ 0 & 0 & S_z & (1-S_z)Z_f \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Reflection

A three-dimensional 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.

The matrix representation for this reflection of points relative to the X axis

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

The matrix representation for this reflection of points relative to the Y axis

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

The matrix representation for this reflection of points relative to the xy plane is

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

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:

$$\begin{bmatrix} 1 & 0 & a & 0 \\ 0 & 1 & b & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Parameters a and b can be assigned and real values. The effect of this transformation matrix is to alter x and y- coordinate values by an amount that is proportional to the z value, while leaving the z coordinate unchanged.

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 $\begin{bmatrix} 1 & 0 & 0 & 0 \\ b & 1 & 0 & 0 \\ c & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$