COMPUTER GRAPHICS

ANUPAMA NAMBURU
ASSOCIATE. PROFESSOR
SCHOOL OF CSE
ROOM 328 G FACULTY AREA 10
NAMBURI.ANUPAMA@VITAP.AC.IN

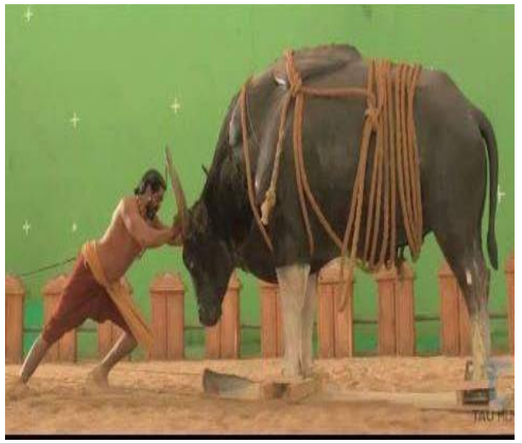
JURASSIC WORLD

WHAT IS COMPUTER GRAPHICS ?

- Computer Graphics involves display, manipulation and storage of pictures and experimental data for proper visualization using a computer.

COMPUTER GENERATED SHOTS

ORIGINAL SHOTS



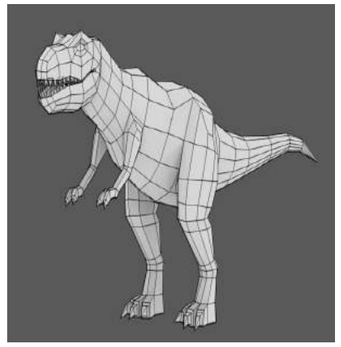
COMPUTER GENERATED SHOTS

ORIGINAL SHOTS

GREEN SCREEN TECHNOLOGY

BACKGROUND??? GREEN SCREEN???

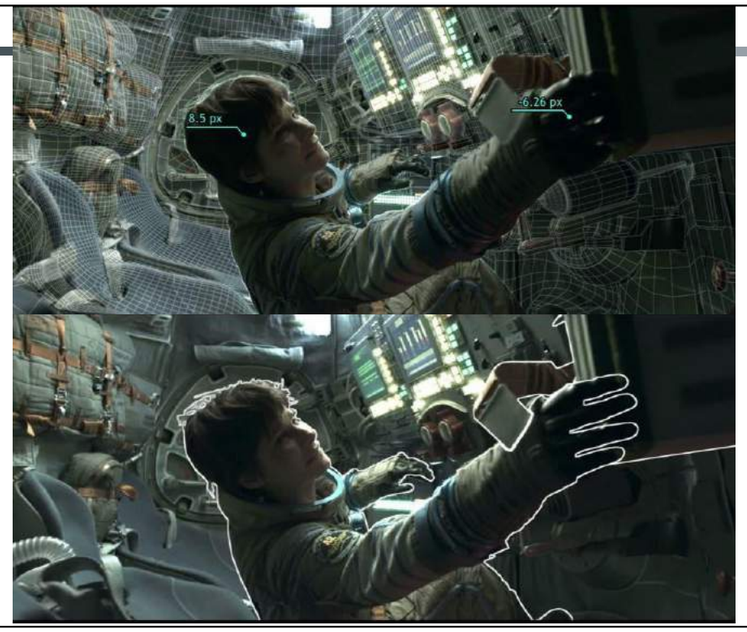
CHARACTER MODELLING



3 DIMENSIONAL VIEW

REALITY?

ROTOSCOPE



COURSE OBJECTIVE

- Principles of 2D and 3D Computer graphics.
- To utilize a computer system and methods to implement the algorithms and techniques necessary to produce 2D and 3D illustrations.
- Understand transformational geometry
- Utilize transforms to positioning and manipulate objects in 2D and 3D dimensional space.
- Determine how a surface should be shaded to produce realistic illustrations.

COMPUTER GRAPHICS – TEXT BOOK

- **Donald D. Hearn, M. Pauline Baker and Warren Carithers, "Computer Graphics with Open GL", Pearson, Fourth Edition, 2013.**
- F.S. Hill, "Computer Graphics Using OpenGL", Pearson, Third Edition, 2006.
- Edward Angel, "Interactive Computer Graphics: A Top-Down Approach", Pearson, Sixth Edition, 2011.
- J. D. Foley, A. Van Dam, S. K. Feiner, and J. F. Hughes "Computer Graphics: Principles and Practice", Addison Wesley Third Edition, 2013.
- D F Rogers , "Mathematical Elements for Computer Graphics", McGraw Hill, 2nd Edition, 2002.

Module No. 1	**Introduction**	**6 hours**
Introduction -What is Computer Graphics? - History of computer graphics, applications, Graphics primitives – graphics pipeline, physical and synthetic images, synthetic camera, modelling, animation, rendering, relation to computer vision and image processing, review of basic mathematical objects		
Module No. 2	**Geometric Manipulation**	**8 hours**
Homogeneous coordinates, affine transformations (translation, rotation, scaling, shear), concatenation, matrix stacks and use of model view matrix in OpenGL for these operations		
Module No 3	**Elementary 3D Graphics**	**8 hours**
Plane projections, Vanishing points, Specification of a 3D view. Camera Models; Viewing classical three-dimensional viewing, computer viewing, specifying views, parallel and perspective projective transformations		
Module No. 4	**Visibility**	**7 hours**
Image and object precision; z-buffer algorithms; area based algorithms BSP trees, Open-GL culling, hidden-surface algorithms		
Module No. 5	**Basic Raster Graphics**	**8 hours**
Scan conversion; filling; and clipping Light sources, illumination model, Gouraud and Phong shading for polygons. Rasterization- Line segment and polygon clipping, 3D clipping, scan conversion, polygonal fill, Bresenham's algorithm		
Module No. 6	**Rendering**	**8 hours**
Lighting; Radiosity; Raytracing texture mapping, compositing, textures in OpenGL; Ray Tracing-Recursive ray tracer, ray-sphere intersection Bezier curves and surfaces, B-splines, visualization, interpolation, marching squares algorithm		

GRAPHICS APPLICATIONS

- Entertainment
- Computer-aided design
- Scientific visualization
- Training
- Education
- E-commerce
- Computer art

GRAPHICS APPLICATIONS

- **Entertainment**
- Computer-aided design
- Scientific visualization
- Training
- Education
- E-commerce
- Computer art

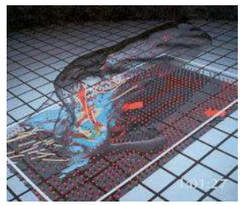
Toy Story

Jurasic Park

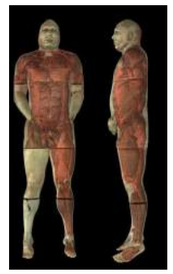
Quake

GRAPHICS APPLICATIONS

- Entertainment
- Computer-aided design
- **Scientific visualization**
- Training
- Education
- E-commerce
- Computer art



Airflow inside a thunderstorm



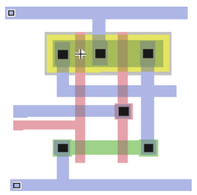
Visible human

GRAPHICS APPLICATIONS

- Entertainment
- **Computer-aided design**
- Scientific visualization
- Training
- Education
- E-commerce
- Computer art

Gear Shaft Design

Los Angeles Airport

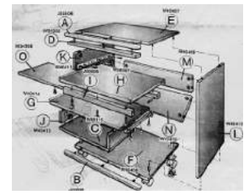


CMOS Circuit Design

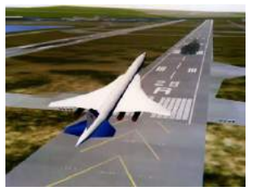
GRAPHICS APPLICATIONS

- Entertainment
- Computer-aided design
- Scientific visualization
- **Training**
- Education
- E-commerce
- Computer art

Driving Simulation



Desk Assembly



Flight Simulation

GRAPHICS APPLICATIONS

- Entertainment
- Computer-aided design
- Scientific visualization
- Training
- ➢ **Education**
- E-commerce
- Computer art

Human Skeleton

GRAPHICS APPLICATIONS

- Entertainment
- Computer-aided design
- Scientific visualization
- Training
- Education
- ➢ **E-commerce**
- Computer art

Virtual Phone Store

Interactive kitchen planner

GRAPHICS APPLICATIONS

- Entertainment
- Computer-aided design
- Scientific visualization
- Training
- Education
- E-commerce
- ➢ **Computer art**

Fantasyartdesign.com

MANY MORE AREAS..

- Entertainment(movie, TV Advt., Games etc.)
- Simulation studies
- Cartography
- Virtual reality
- Process Monitoring
- Digital Image Processing
- Education and Training
- Simulators
- Multimedia
- Desktop publishing

FOR LAB

- Processing Sketch 3 (java/python)
  - https://processing.org/
- OpenGL
  - www.opengl.org

1 . PROCESSING 3

- Processing is a flexible software sketchbook and a language for learning how to code within the context of the visual arts.
- There are tens of thousands of students, artists, designers, researchers, and hobbyists who use Processing for learning and prototyping.

» Free to download and open source

» Interactive programs with 2D, 3D, PDF, or SVG output

» OpenGL integration for accelerated 2D and 3D

» For GNU/Linux, Mac OS X, Windows, Android, and ARM

» Over 100 libraries extend the core software

» Well documented, with many books available

2. OPENGL REFERENCES

- **www.opengl.org**
- **The Red Book** :

The OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 2 (5th Edition) Addison-Wesley 2004.

http://fly.cc.fer.hr/~unreal/theredbook/

- **The Blue Book** :

The OpenGL Reference Manual : The Official Reference Document to OpenGL, Version 1.4 (4th Edition), Addison-Wesley 2004.

http://www.glprogramming.com/blue/

WHAT IS OPENGL

- OpenGL is a graphics library for developing portable, interactive 2D and 3D graphics applications.
- Most Widely Adopted Graphics Standard
- Open, vendor-neutral, multiplatform
- Can run OpenGL on any system
  - Windows
  - Linux
  - Mac
- Get GLUT from web if needed (Lab TA will explain further)

COMPUTER GRAPHICS

Introduction

We will cover…

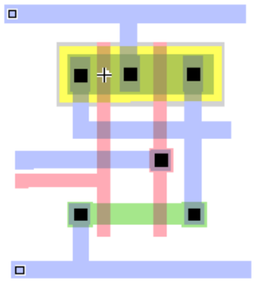
- Graphics programming and algorithms
- Graphics data structures
- Colour
- Applied geometry, modelling and rendering

COMPUTER GRAPHICS IS ABOUT ALL ASPECTS OF PICTURE AND IMAGE GENERATED WITH COMPUTER

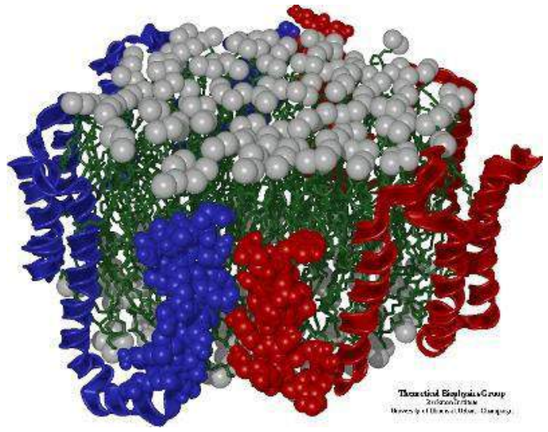
GAMES ARE VERY IMPORTANT IN COMPUTER GRAPHICS

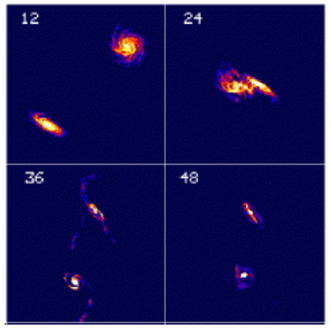
MEDICAL IMAGING IS ANOTHER DRIVING FORCE

COMPUTER AIDED DESIGN TOO



SCIENTIFIC VISUALISATION





FIRST LECTURE

The graphics processes
- What we will cover on this course

Some definitions
- Fundamental units we use in these processes

First Practical

OVERVIEW OF THE COURSE

Graphics Pipeline (Today)

Modelling

Surface / Curve modelling

Local lighting effects

Illumination, lighting, shading, mirroring, shadowing

Rasterization (creating the image using the 3D scene)

Ray tracing

Global illumination

Curves and Surfaces

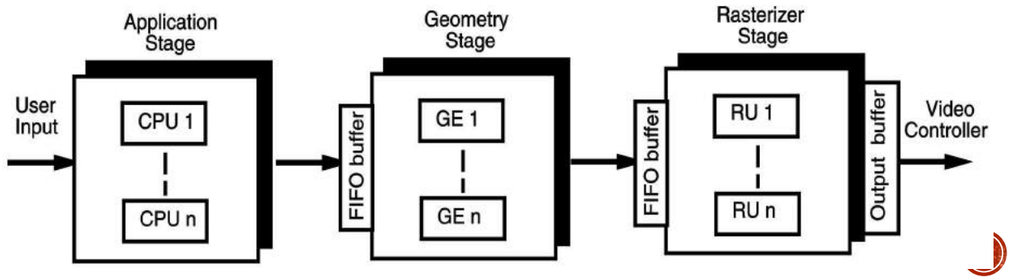
GRAPHICS / RENDERING PIPELINE

There are three stages

Application Stage

Geometry Stage

Rasterization Stage



GRAPHICS/RENDERING PIPELINE

Graphics processes generally execute sequentially

Pipelining the process means dividing it into stages

Especially when rendering in real-time, different hardware resources are assigned for each stage

APPLICATION STAGE

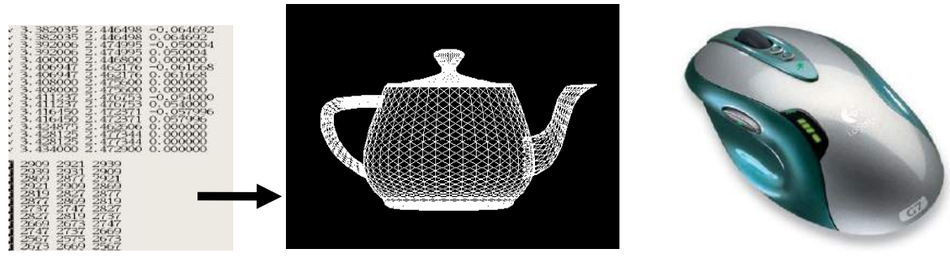
Entirely done in software by the CPU

Read Data

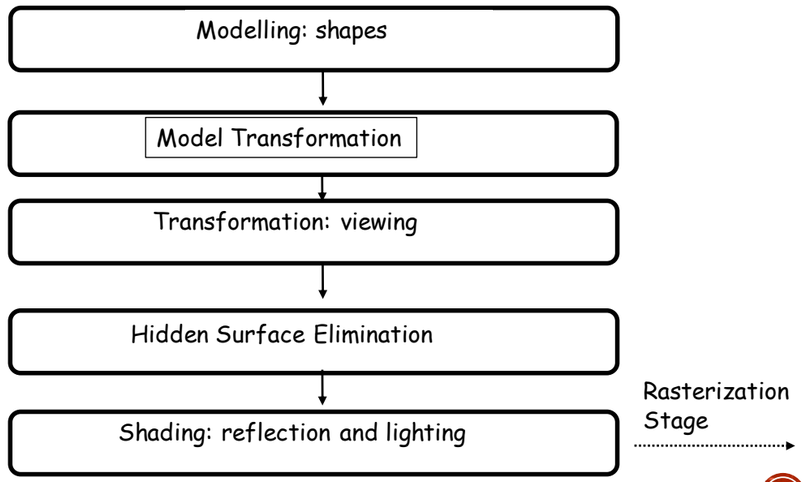
the world geometry database,

User's input by mice, trackballs, trackers, or sensing gloves

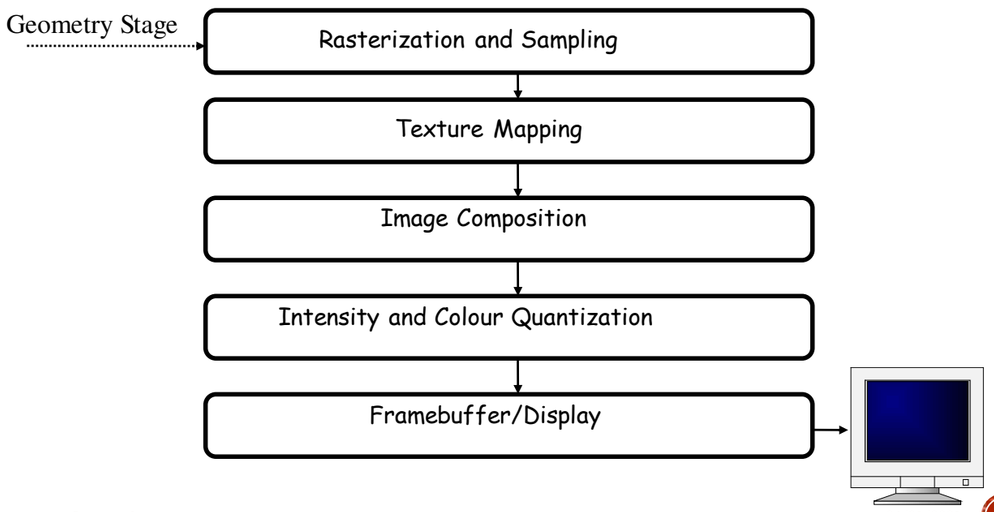
In response to the user's input, the application stage change the view or scene



GEOMETRY STAGE

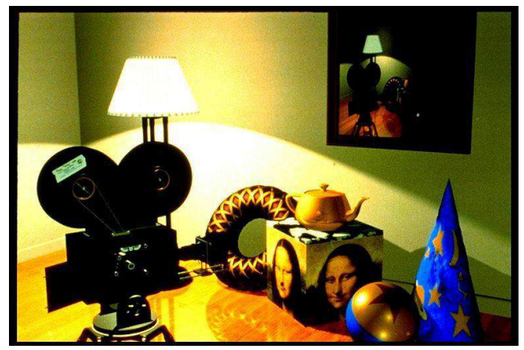


RASTERIZATION STAGE

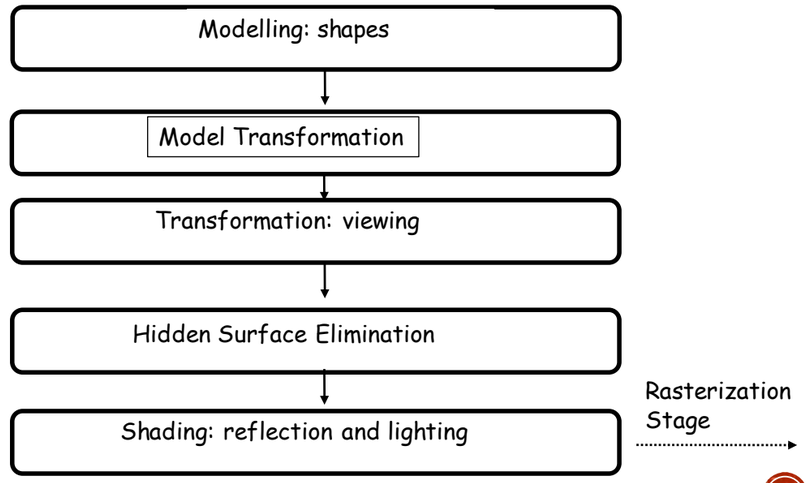


AN EXAMPLE THRO' THE PIPELINE...

The scene we are trying to represent:

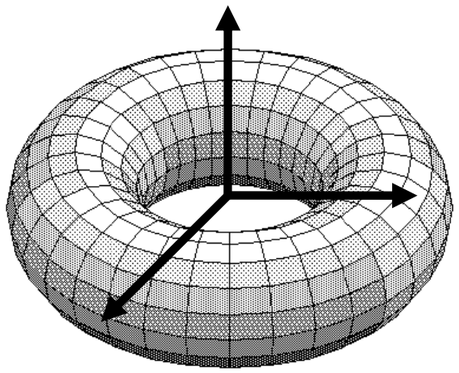


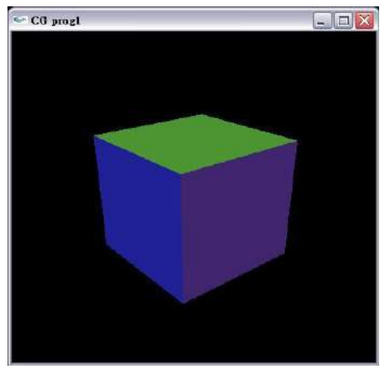
GEOMETRY STAGE



PREPARING SHAPE MODELS

- Designed by polygons, parametric curves/surfaces, implicit surfaces etc
- Defined in its own coordinate system



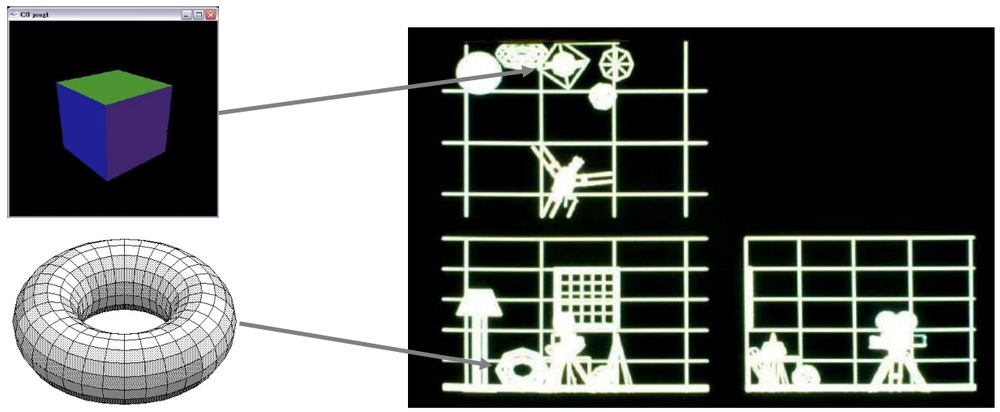


MODEL TRANSFORMATION

Objects put into the scene by applying translation, scaling and rotation

Linear transformation called homogeneous transformation is used

The location of all the vertices are updated by this transformation

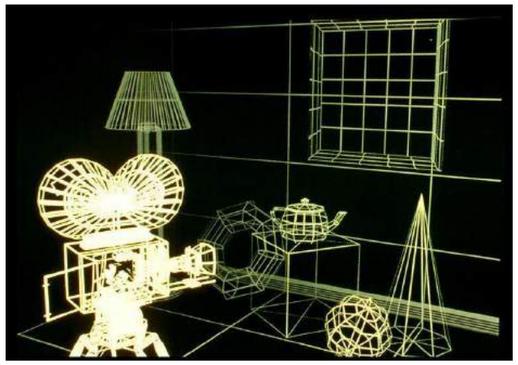


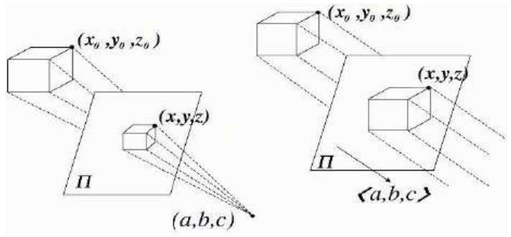
PERSPECTIVE PROJECTION/VIEWING

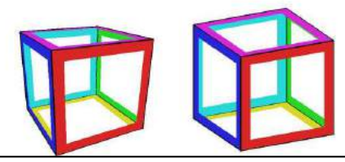
We want to create a picture of the scene viewed from the camera

We apply a perspective transformation to convert the 3D coordinates to 2D coordinates of the screen

Objects far away appear smaller, closer objects appear bigger

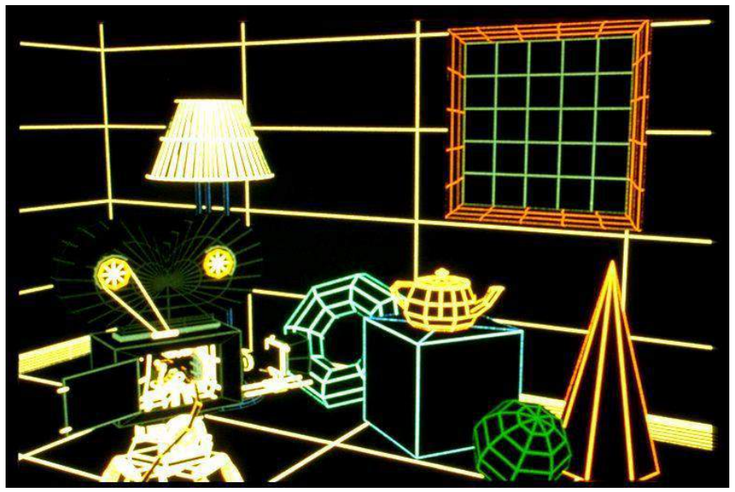






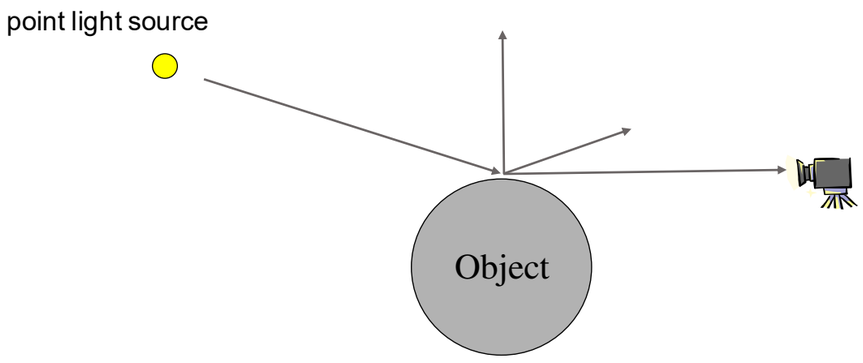
HIDDEN SURFACE REMOVAL

Objects occluded by other objects must not be drawn



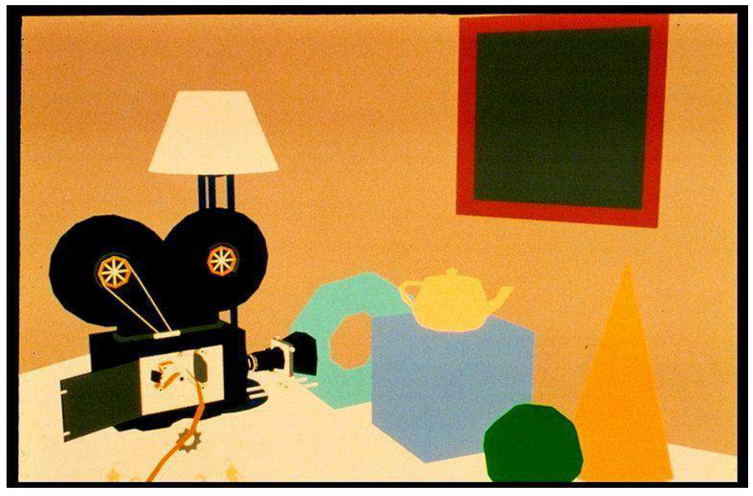
SHADING

Now we need to decide the colour of each pixels taking into account the object's colour, lighting condition and the camera position



SHADING : CONSTANT SHADING - AMBIENT

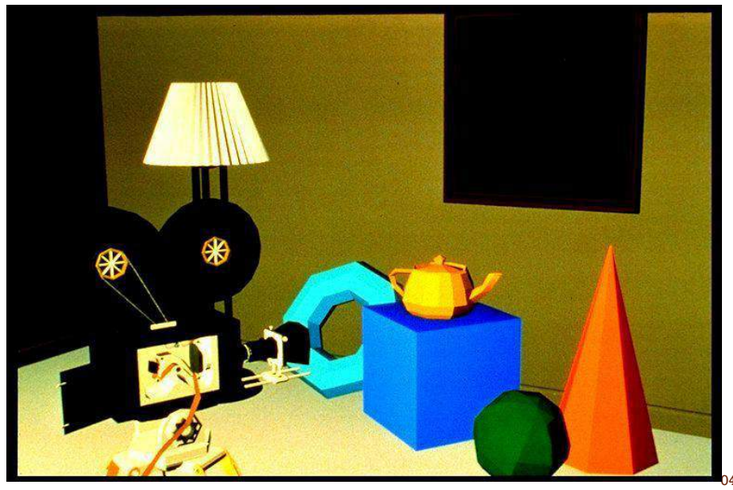
Objects colours by its own colour



SHADING – FLAT SHADING

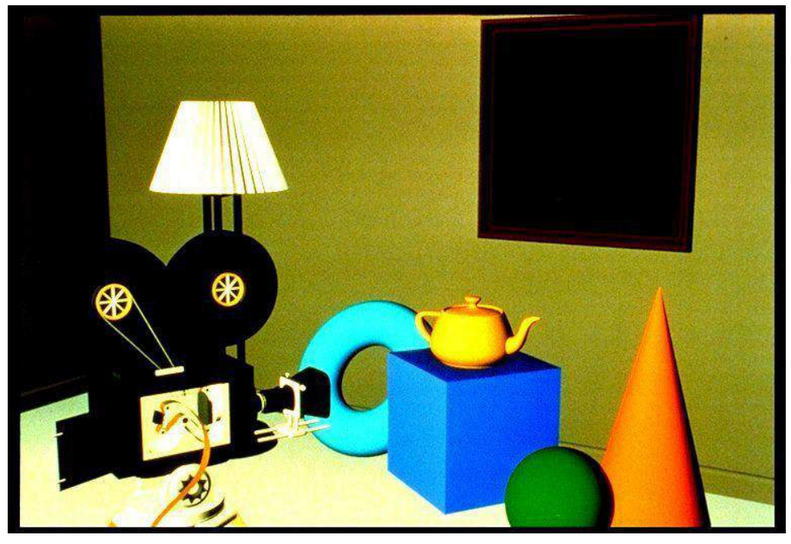
Objects coloured based on its own colour and the lighting condition

One colour for one face

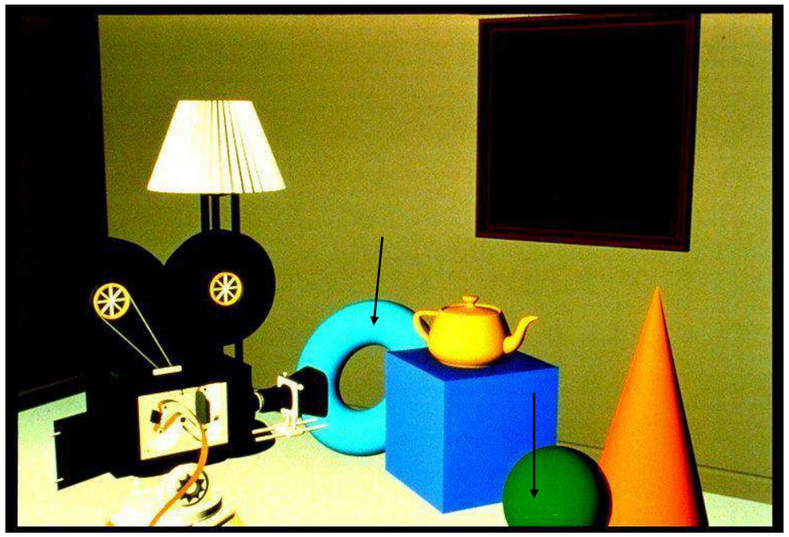


GOURAUD SHADING, NO SPECULAR HIGHLIGHTS

Lighting calculation per vertex

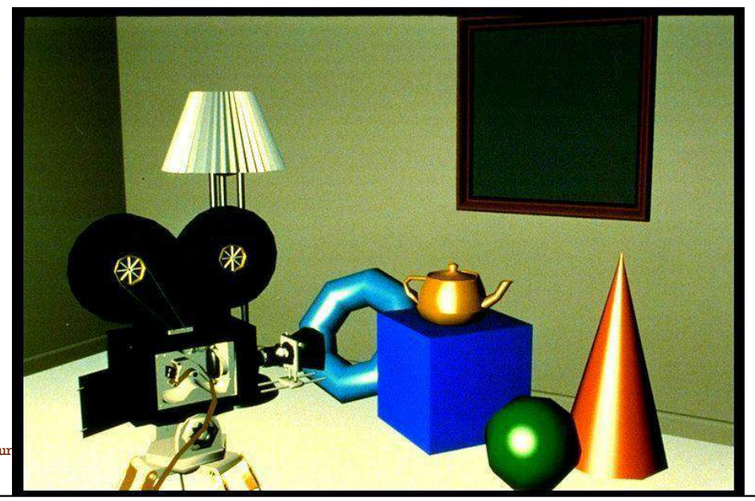


SHAPES BY POLYNOMIAL SURFACES

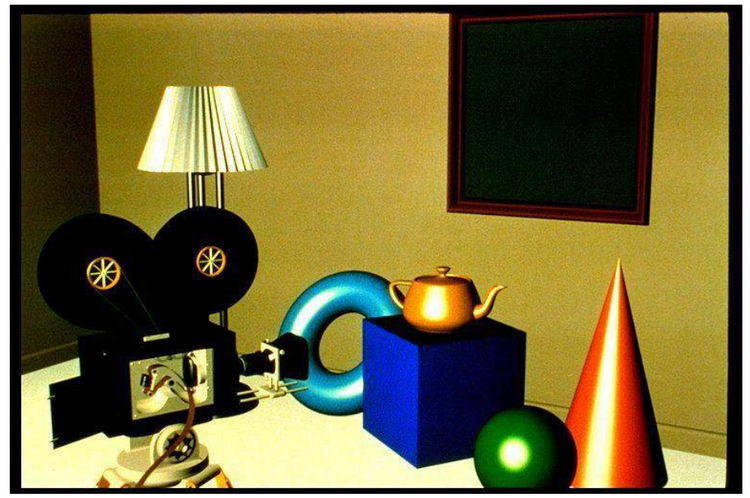


SPECULAR HIGHLIGHTS ADDED

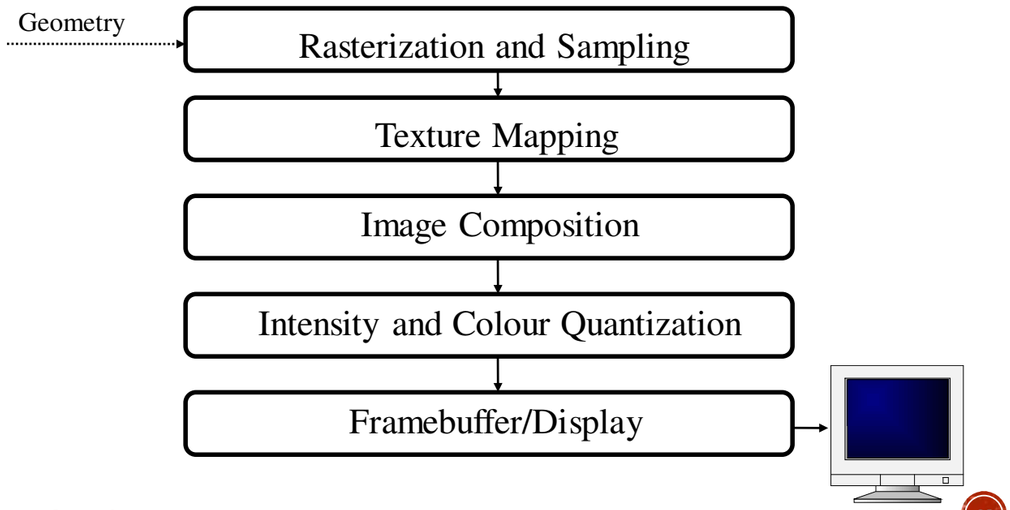
Light perfectly reflected in a mirror-like way



PHONG SHADING



NEXT, THE IMAGING PIPELINE

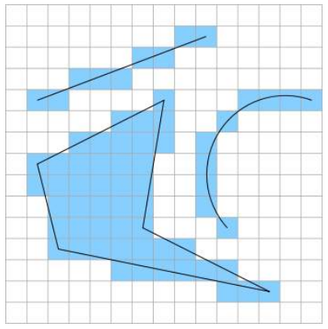


RASTERIZATION

Converts the vertex information output by the geometry pipeline into pixel information needed by the video display

Aliasing: distortion artifacts produced when representing a high-resolution signal at a lower resolution.

Anti-aliasing : technique to remove aliasing

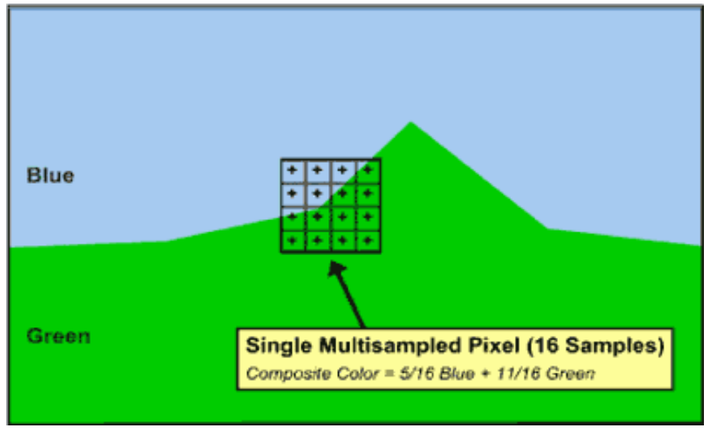


ANTI-ALIASING

Aliased polygons (jagged edges)

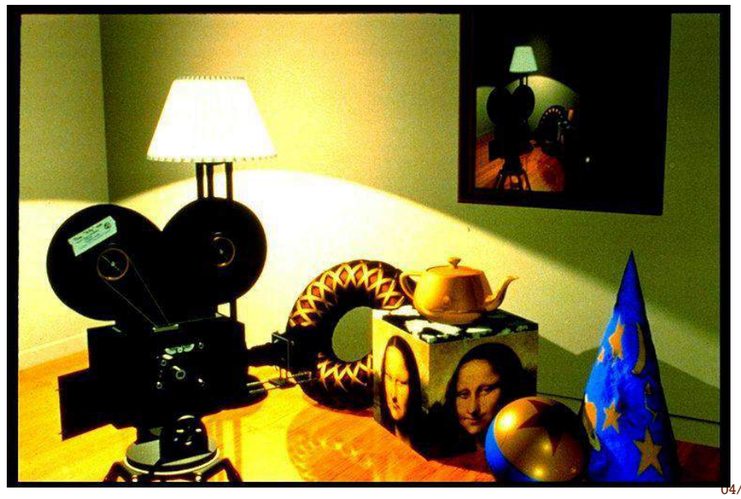
Anti-aliased polygons

✓**How is *anti-aliasing* done? Each pixel is subdivided (sub-sampled) in n regions, and each sub-pixel has a color;**
✓**Compute the average color value**



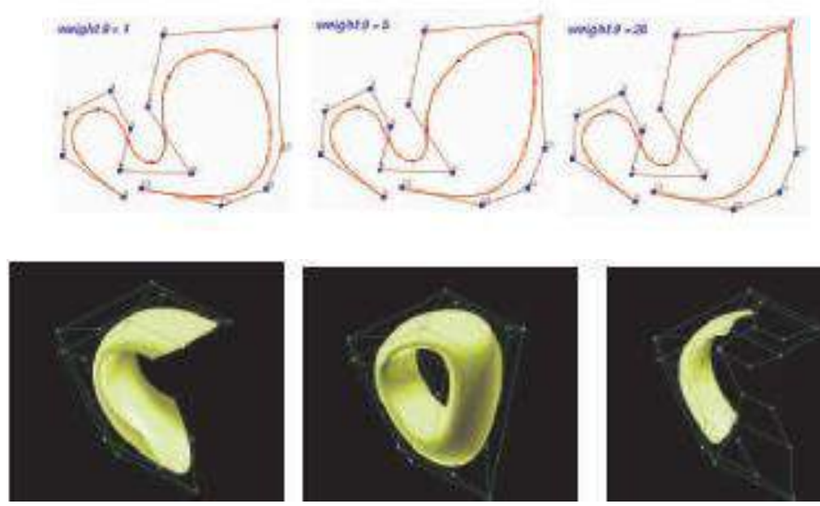
TEXTURE MAPPING

OTHER COVERED TOPICS: REFLECTIONS, SHADOWS & BUMP MAPPING



OTHER COVERED TOPICS: GLOBAL ILLUMINATION

POLYNOMIAL CURVES, SURFACES



GRAPHICS DEFINITIONS

Point
- a location in space, 2D or 3D
- sometimes denotes one pixel

Line
- straight path connecting two points
- infinitesimal width, consistent density
- beginning and end on points

GRAPHICS DEFINITIONS

Vertex
- point in 3D

Edge
- line in 3D connecting two vertices

Polygon/Face/Facet
- arbitrary shape formed by connected vertices
- fundamental unit of 3D computer graphics

Mesh
- set of connected polygons forming a surface (or object)
- :

GRAPHICS DEFINITIONS

Rendering : process of generating an image from the model

Framebuffer : a video output device that drives a video display from a memory containing the color for every pixel

SUMMARY

The course is about algorithms, not applications
- Lots of mathematics

Graphics execution is a pipelined approach

Basic definitions presented

Some support resources indicated

Graphics primitives

Agenda

- History of computer graphics
- Graphics pipeline(already covered)
- Physical and synthetic images
- Synthetic camera

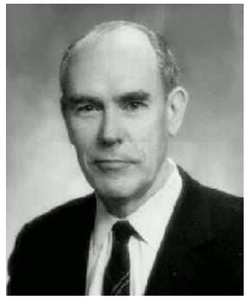
Pioneer：Ivan Sutherland

- First truly interactive graphics system, Sketchpad, pioneered at MIT by Ivan Sutherland for his 1963 Ph.D. thesis

Sketchpad in 1963. Note use of a CRT monitor, light pen and function-key panel.

The History of Computer Graphics

Pioneer: Ivan Sutherland
1963: Sutherland's PhD Thesis：“Sketchpad: A Man-machine Graphical Communications System.”, MIT,1963



First time used “Computer Graphics”. CG started to be a novel and independent scientific branch.

The History of Computer Graphics

- **Poineer: Ivan Sutherland**
- **1962: Pierre Bezier put forward “Bezier curve” for the representation of space curve**
- **1967: Wylie added lighting effect in objects representation**
- **1969: Xerox developed GUI (Graphic User Interface)**
- **1973: Richard Shoup invented Raster-Scan Display**
- **The great improvement of graphic techniques**
  - **Phong lighting model(1973)； Texture mapping(1974)； Ray tracing(1980)； Radiosity(1984)…**

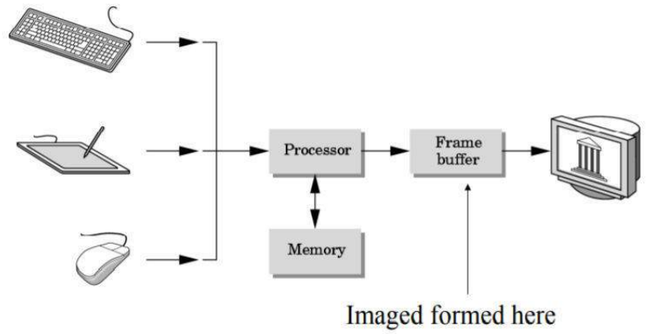
The History of Computer Graphics

- Industry
  - ILM(Industrial Light and Magic): an Academy Award winning motion picture visual effects company, 1975
  - SGI (*Silicon Graphics, Inc*): 1982
  - Pixar 1986
  - AutoDesk, Adobe
- Display card
  - 1994: the first PC display card --- by 3DLabs
  - 1995.11: Voodoo --- by 3DFx
  - 1999: Geforce256, the first GPU --- by nVidia and ATI
  - Geforce 8800、Radeon HD 2900 XT

The History of Computer Graphics

- Graphics Standard
  - GKS(Germany, 1970's); PHIGS(ISO, 1986); GKS-3D(1988)
  - OpenGL(SGI, 1992); DirectX(Microsoft); Java3D(Sun)
- Graphics Application Software
  - 3DS Max, Maya, LightWave 3D
  - Renderman
  - AutoCAD, Solid Work
  - Blender

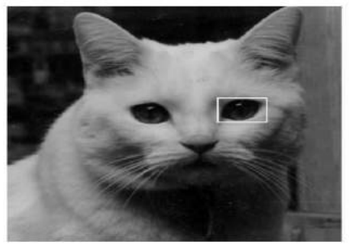
A Typical Graphics System



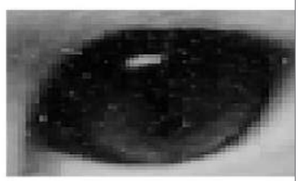
Pixels and the Frame Buffer

- All graphics systems are raster-based.
- A picture is produced as an array – the raster – of picture elements, pixels.
- Each pixel corresponds to a small area of the image

Pixels are stored in a part of memory called the **frame buffer**.



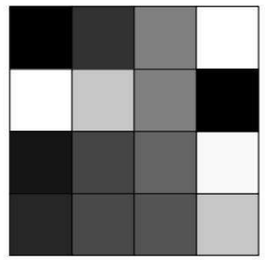
A) Image of Yeti the cat.



B) Detail of area around one eye showing individual pixels.

A sample Image in Pixel_Gray_Map (PGM) Format

```
P2
# test
4 4
255
0     50    127   255
255   200   127   0
10    60    100   250
20    70    80    200
```



```
This is a 4_by_4 8 bit image, i.e., 255
   represents white and 0 represents black,
   other numbers are used for different
   shades of gray.
```

Pixels and the Frame Buffer

- The depth of the frame buffer, defines the number of bits that are used for each pixel and defines the number of colors.
- The resolution is the number of pixels in the frame buffer and determines the detail that you can see in the image.
- The conversion of geometric entities to pixel assignments in the frame buffer is known as rasterization, or scan conversion.

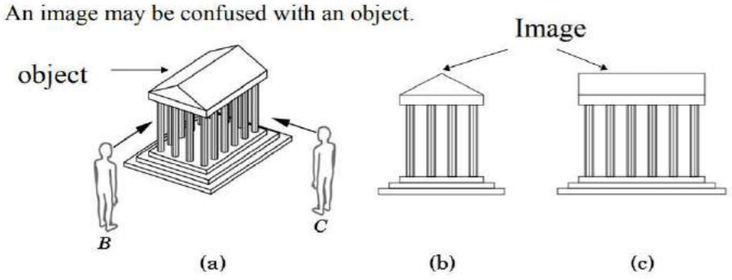
Physical and Synthetic Images

- A computer generated image is a synthetic or artificial,
- In the sense that the object being imaged does not exist physically.
- In order to understand how synthetic images are generated, we first look into the ways traditional imaging systems such as cameras form images.

Physical imaging systems

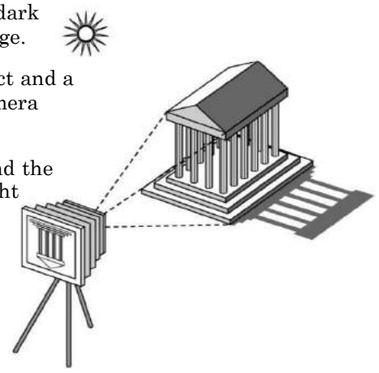
- To form an image, we must have someone, or something, that is viewing our objects, be it a human, a camera, or a digitizer.
- It is the viewer that forms the image of our objects.
- In human visual system, the image is formed on the back of the eye, on the retina. In a camera, the image is formed on the film plane.

An image may be confused with an object.

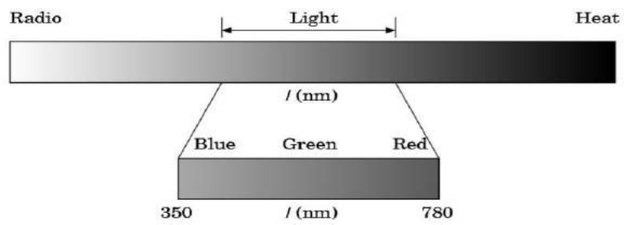


Light and Images

- If there is no light source, an object would be dark and there won't be anything visible of the image.
- Light usually strikes various parts of the object and a portion of the reflected light will enter the camera through the lens.
- The details of the interaction between light and the surfaces of the object determine how much light enters the camera.
- Light is formed of electromagnetic radiation characterized by its wavelength or frequency.



- The electromagnetic spectrum includes radio waves, infrared (heat), and a portion that causes a response in our visual systems, visible light spectrum.

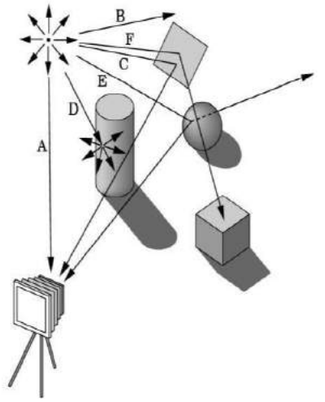


The color of light source is determined by the energy that it emits at various wavelengths.

In graphics, we use the geometric optics which models light sources as emitters of light energy that have a fixed rate or intensity

Ray tracing

- We can model an image by following light from a source. The light that reaches the viewer, will determine the image-formation.
- Ray tracing is an image-formation technique used as the basis for producing computer-generated images.
- A ray is a semi-infinite line that emanates from a point and travels to infinity in a particular direction.
- Light travels in straight line, thus only a portion the light will get to the viewer.
- The light from the light source can interact with the surface of the object differently, depending on the orientation and the luminosity of the surface.



The Human Visual System

Lights enters the eye through Cornea and Lens.

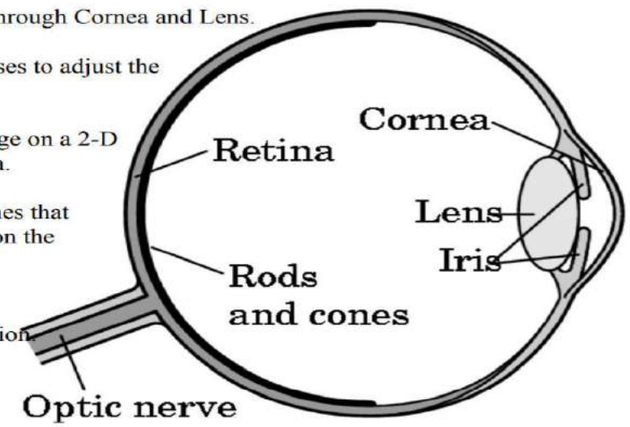
The Iris opens and closes to adjust the amount of light.

The lens forms an image on a 2-D structure called Retina.

There are rods and cones that Act like light sensors on the Retina.

Rods are low-level light sensors, night vision

Cones are responsible for our day vision.

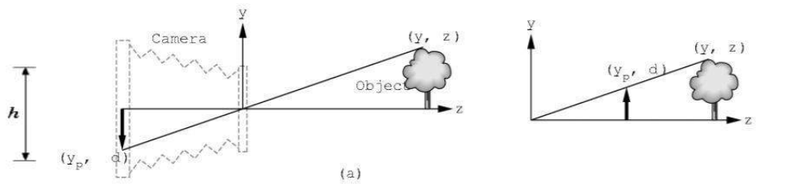


Synthetic Imaging

- Synthetic images are computer-generated image that are similar to forming an image using an optical system.
- This paradigm is known as synthetic-camera model.
- The image is formed on the back of the camera, so we can emulate this process to create artificial images
- We can compute the image using simple trigonometric calculations.

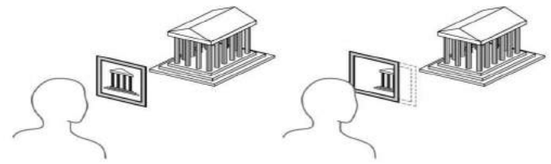
$$\tan(\theta) = \frac{h/2}{d}$$

$$\theta = 2\tan^{-1}\frac{h}{2d}$$



Synthetic Imaging

- We must consider the limited size of the image. Not all objects can be imaged onto the camera film plane. The angle of view expresses this limitation.
- In our synthetic camera, we place this limitation by placing a clipping rectangle or clipping window in the projection plane.

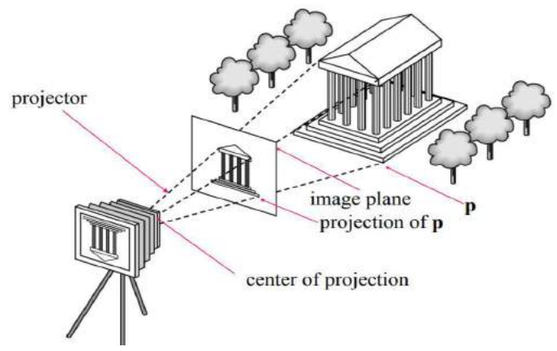


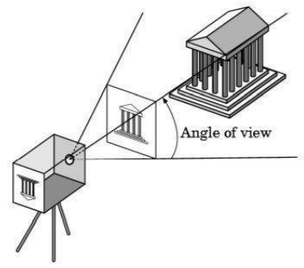
(a) (b)

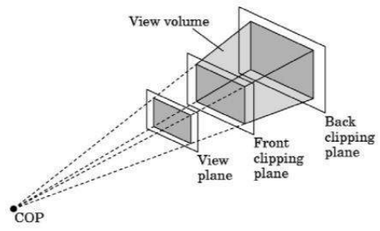
Given the location of the center of the projection, the location and orientation of the projection plane, and the size of the clipping rectangle, we can determine which objects will appear in the image.

Synthetic Imaging

- We find the image of a point on the object by drawing a line, projector, from the point to the center of the lens, or the center of projection. In our synthetic camera, the film plane is moved in front of the lens and is called Projection plane.



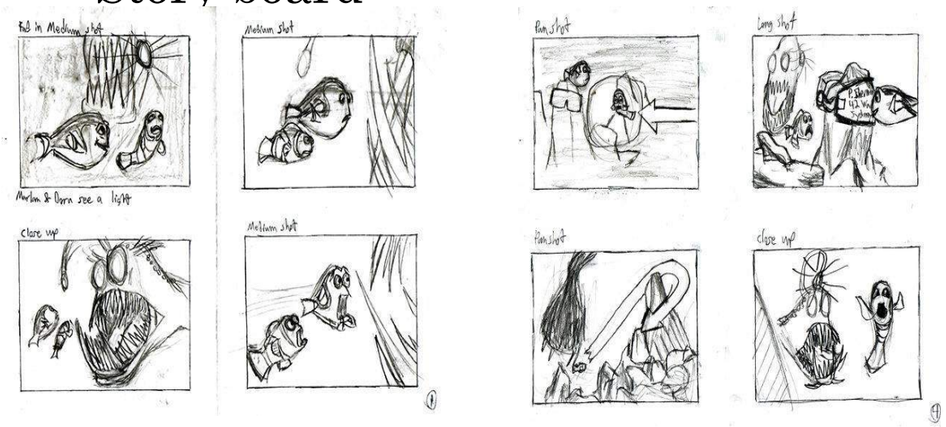




Animation

- Process used for generating animated images
- Application
  - Video games
  - Cartoons/movies
  - Mobile applications

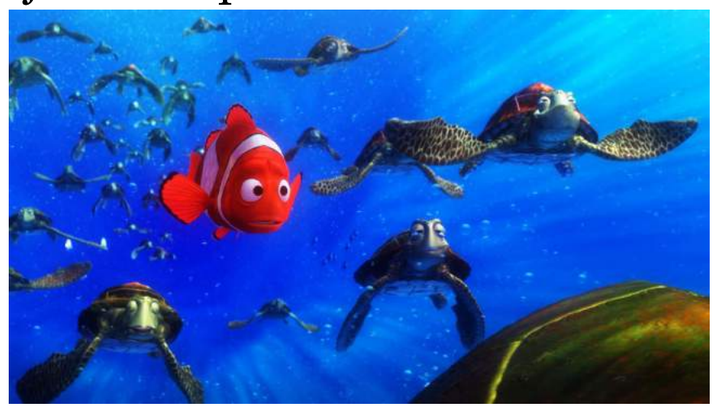
Story board



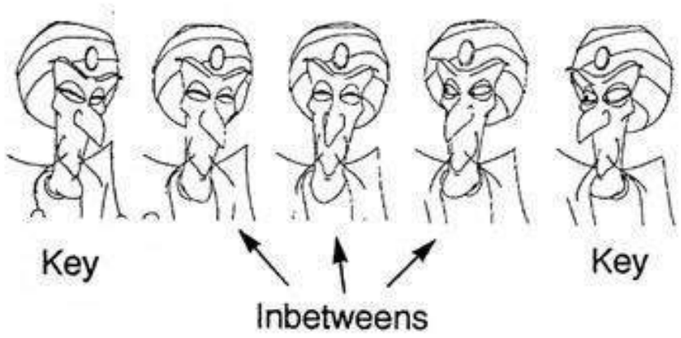
Designing animation sequence

- Story board layout
- Object and path definition
- Key frame specification
- Generation of in-between frames

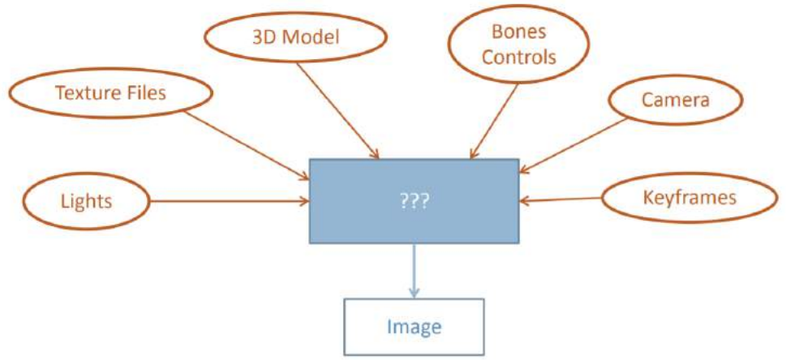
Object and path definition



Key frames and in-between frames generation



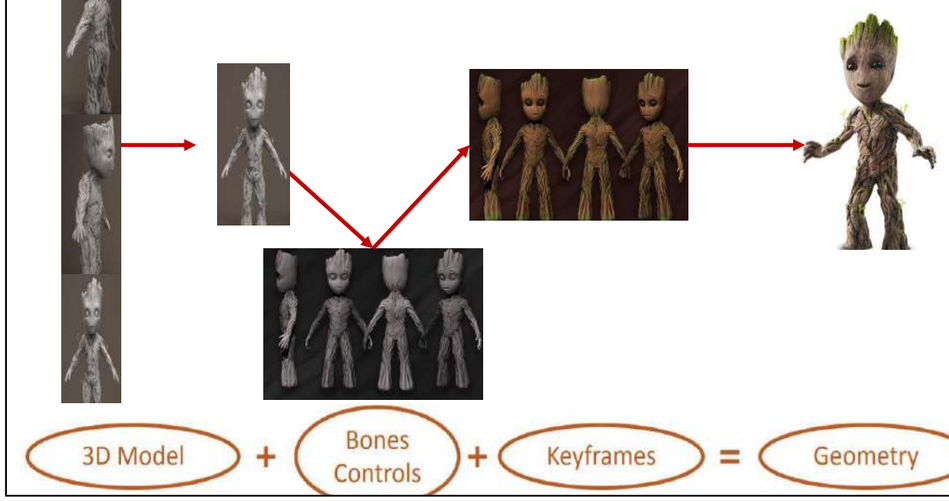
Rendering



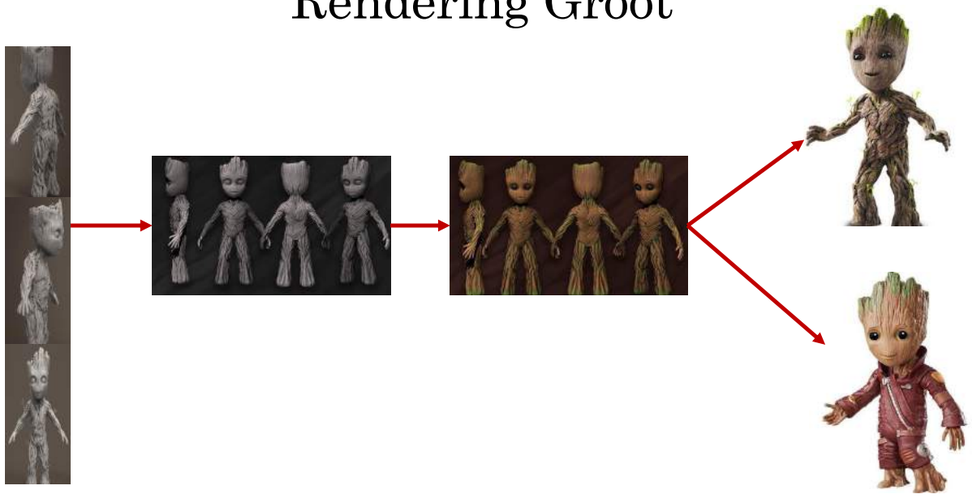
Sibi Chakkaravarthy Sethuraman, CSE, VIT-AP

Animation

Making of Groot



Rendering Groot



Rendering - Factors to be considered

- Projection
- Occlusion (technique used to calculate how each point in a scene is exposed to lighting)
- Color / Texture
- Lighting
- Shadows
- Reflections / Refractions (reflected rays/ transmitted rays)
- Indirect illumination (techniques used to add more realistic lighting to 3D scenes)
- Sampling / Antialiasing (technique used to reduce the visual defects that occur when high-resolution images are presented in a lower resolution)

Computer vision vs Image processing

Computer Vision:

- **Input**: Images
  Output: Knowledge of the scene (recognize objects, people, activity happening there, distance of the object from camera and each other, ...)
  Methods: Image processing, machine learning, ...

Image Processing:

- **Input**: Images
  Output: Images (Might be in different formats, for example compressed images). No knowledge of the scene is given.
  Methods: Different filtering, FFT,

Mathematical object models – A review

- **Algebra and Trigonometry (**Vectors and matrix)
- **Linear Algebra (**numerical representations of geometry)
- **Calculus/ Differential Geometry (**smooth curves and surfaces**)**
- **Numerical Methods (**represent and manipulate numbers**)**
- **Sampling Theory and Signal Processing (**Image processing(2D/3D)**)**
- **Physics (**animation/particles/model dynamics**)**
- **Optimization (**gaming**)**

Graphics Primitives

Two kinds of computer graphics

- Raster Graphics
- Vector graphics

Raster graphics vs Vector graphics

- What is a raster graphics?
  - Raster graphics are more commonly called **bitmap images**.
  - A **bitmap** image uses a grid of individual pixels where each pixel can be a different color or shade.
  - Bitmaps are composed of pixels

Raster graphics vs Vector graphics

- What is a vector graphics?
  - Composed of points and paths
  - Uses mathematical relationships between points and paths, connecting them to describe an image.

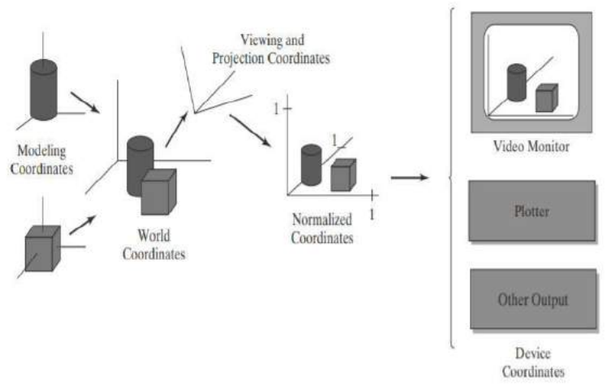
Vector vs Raster

Modelling coordinates

- To generate the picture, geometric descriptions of the objects that are to be displayed are needed.
- These descriptions determine the locations and shapes of the objects.
- Defines the shapes of individual objects, such as trees or furniture, within a separate reference frame.
- These reference frames are called modeling coordinates, or sometimes local coordinates or master coordinates

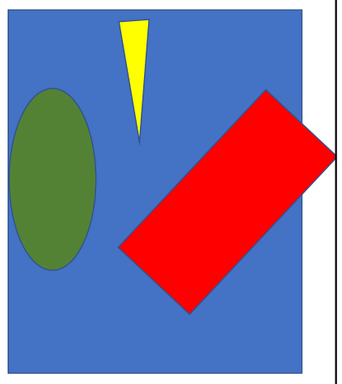
Coordinate Systems

Model coordinates
World coordinates
Viewing coordinates
Normalized coordinates
Device coordinates



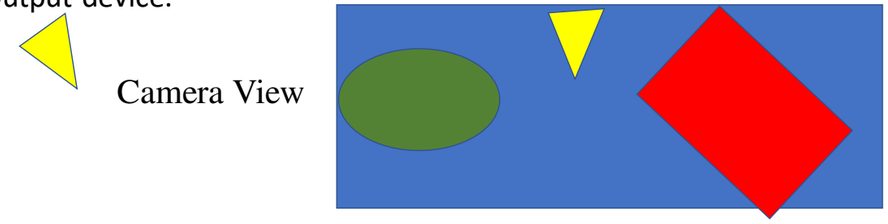
World coordinates

- Once the individual object shapes have been specified a scene is constructed by placing the objects into appropriate locations within a scene reference frame called world coordinates.
- This step involves the transformation of the individual modeling-coordinate frames to specified positions and orientations within the world-coordinate frame.
- For a repeating object only one modelling coordinate is created.
- Translate,rotate,scale..etc



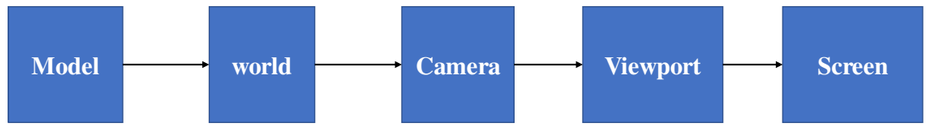
Viewing Coordinates

- World coordinate positions are first converted to viewing coordinates corresponding to the view we want of a scene, based on the position and orientation of a hypothetical camera.
- Then object locations are transformed to a two-dimensional (2D) projection of the scene, which corresponds to what we will see on the output device.



Device coordinates

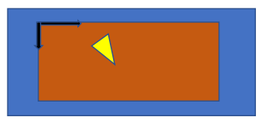
- Finally, the picture is scan-converted into the refresh buffer of a raster system for display.
- The coordinate systems for display devices are generally called device coordinates, or screen coordinates in the case of a video monitor.



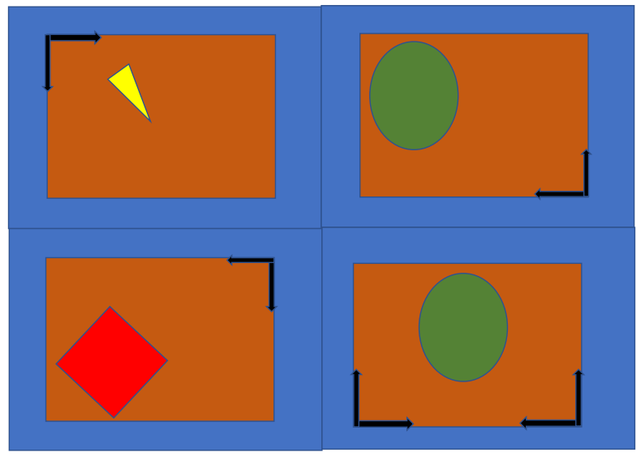
(xmc , ymc , zmc) → (xwc , ywc , zwc) → (xvc , yvc , zvc) → (xpc , ypc , zpc) → (xnc , ync , znc) → (xdc , ydc)

Normalized coordinates

- The scene is then stored in normalized coordinates, where each coordinate value is in the range from −1 to 1 or in the range from 0 to 1, depending on the system.
- Normalized coordinates are also referred to as normalized device coordinates, since using this representation makes a graphics package independent of the coordinate range for any specific output device
- We also need to identify visible surfaces and eliminate picture parts outside the bounds for the view we want to show on the display device.



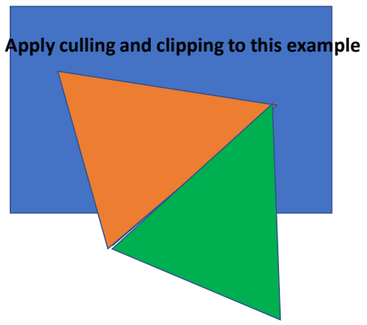
Screen



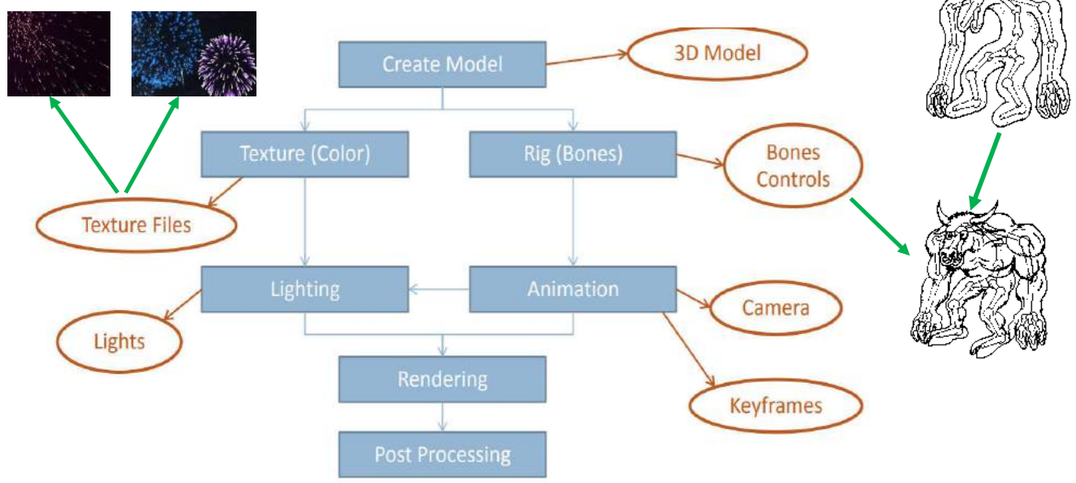
Common properties

- Multi cameras in the world.
- Multi view ports in the screen space.
- Culling
  - Act of excluding entire object from the pipeline.
- Clipping
  - Act of cutting out a portion of an object.

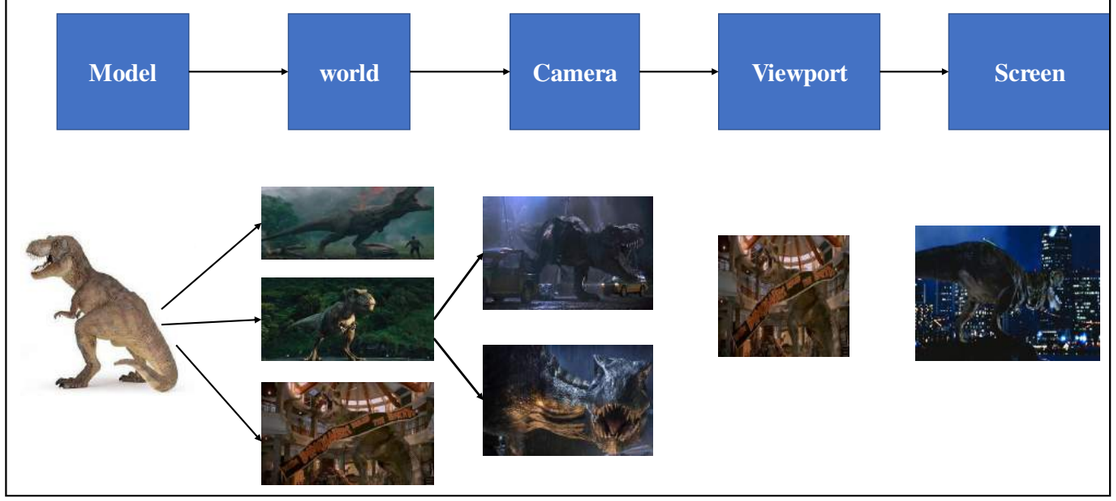
- **Note:**
  - Clipping and culling takes place in world space



Modelling – 3D



T-rex in Real world



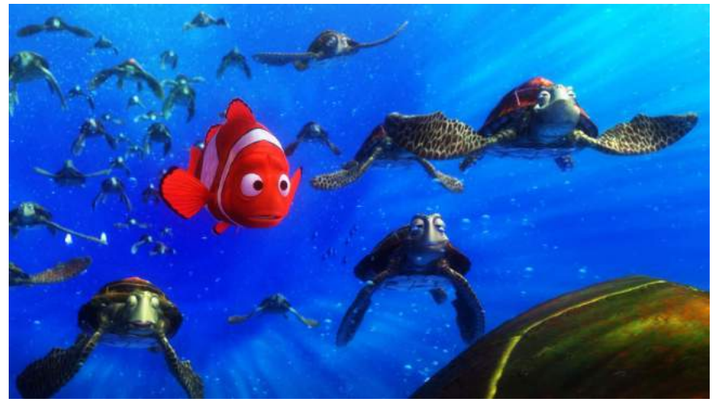
Animation

- Process used for generating animated images
- Application
  - Video games
  - Cartoons/movies
  - Mobile applications

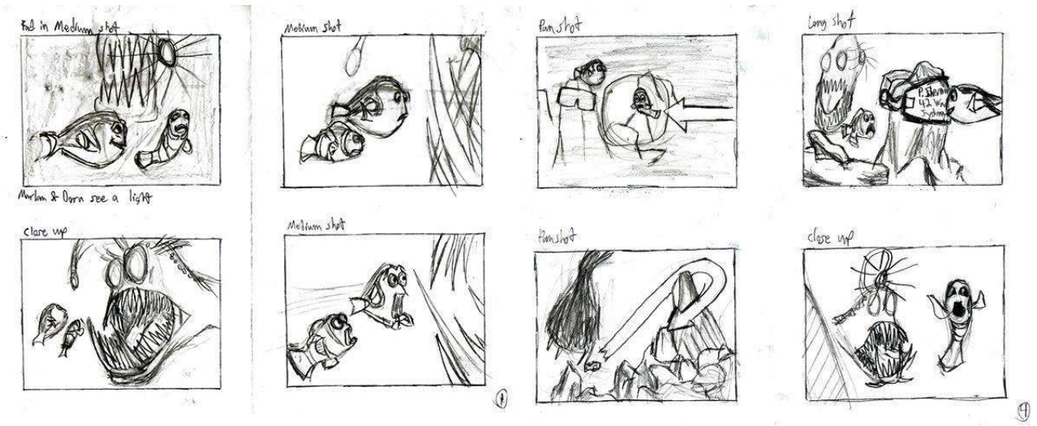
Designing animation sequence

- Story board layout
- Object and path definition
- Key frame specification
- Generation of in-between frames

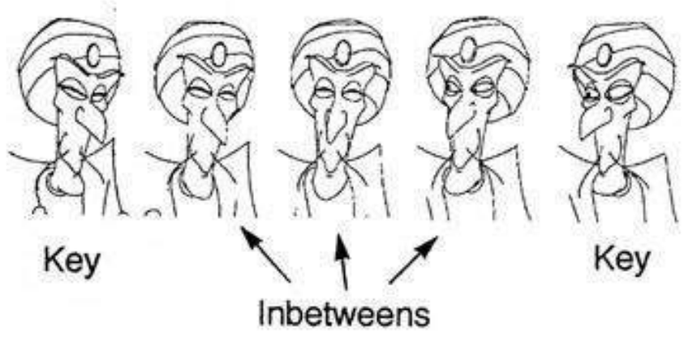
Object and path definition



Story board

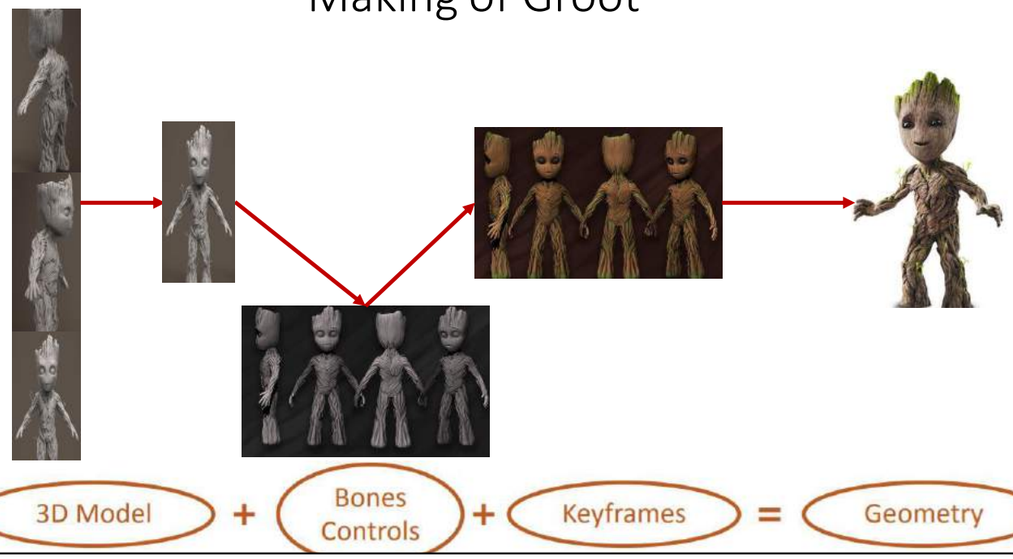


Key frames and in-between frames generation

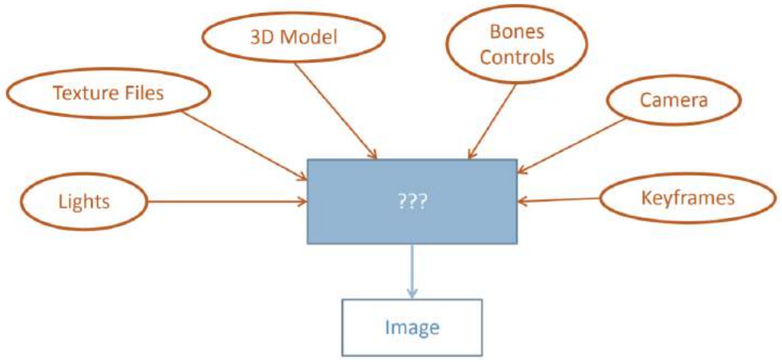


Animation

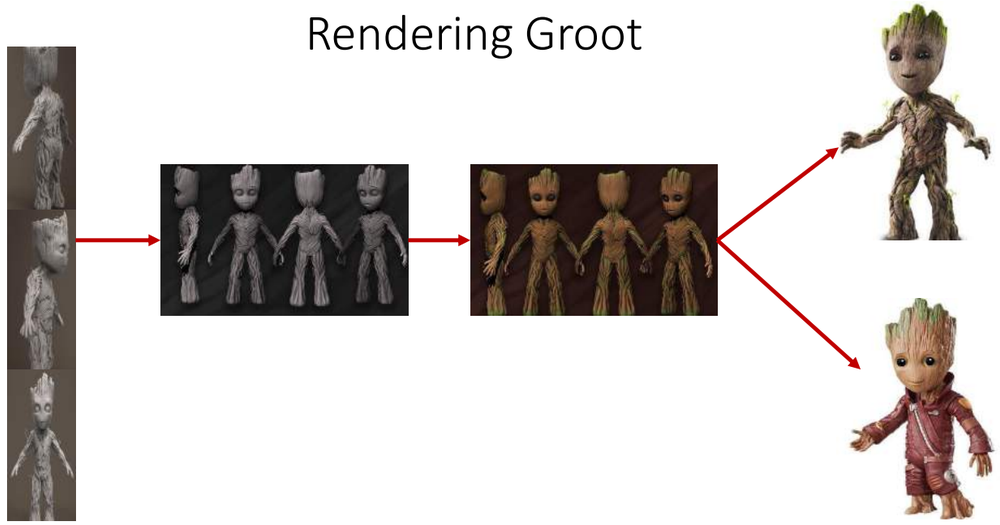
Making of Groot



Rendering



Rendering Groot



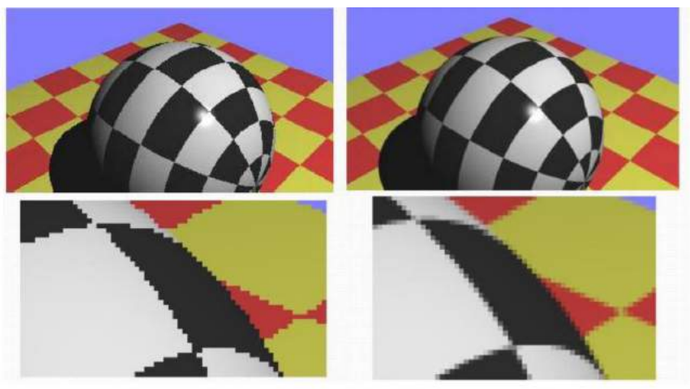
Rendering - Factors to be considered

- Projection
- Occlusion (technique used to calculate how each point in a scene is exposed to lighting)
- Color / Texture
- Lighting
- Shadows
- Reflections / Refractions (reflected rays/ transmitted rays)
- Indirect illumination (techniques used to add more realistic lighting to 3D scenes)
- Sampling / Antialiasing (technique used to reduce the visual defects that occur when high-resolution images are presented in a lower resolution)

Computer vision vs Image processing

- ***Computer Vision:***
- **Input**: Images
  Output: Knowledge of the scene (recognize objects, people, activity happening there, distance of the object from camera and each other, ...)
  Methods: Image processing, machine learning, ...
- ***Image Processing:***
- **Input**: Images
  Output: Images (Might be in different formats, for example compressed images). No knowledge of the scene is given.
  Methods: Different filtering, FFT,

Sampling / Antialiasing



Mathematical object models – A review

- **Algebra and Trigonometry (**Vectors and matrix)
- **Linear Algebra (**numerical representations of geometry)
- **Calculus/ Differential Geometry (**smooth curves and surfaces**)**
- **Numerical Methods (**represent and manipulate numbers**)**
- **Sampling Theory and Signal Processing (**Image processing(2D/3D)**)**
- **Physics (**animation/particles/model dynamics**)**
- **Optimization (**gaming**)**

Basic shapes

P5.js/preprocessor

- Point
- Single location in the raster or image

```
function setup()
{
createCanvas(400, 400);
}
function draw()
{
 background(220);
 set(50,100,color(255,10,10))
 updatePixels();
}
```

Basic shapes/Primitives

- Point
- Line
- Circle
- Ellipse

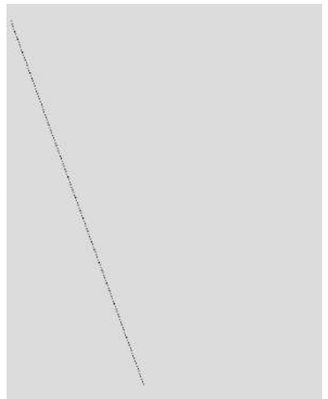
Line

- Y=mx+c is the Line equation
- c is the intercept on y axis.
- It means x=0, Y=m*0+c, it means c is starting of y
- Given two point x1,y1 and x2,y2 how do we draw the line?
- m=y2-y1/x2-x1 and known c=y1 then

Y=m*(x-x1)+c (x-x1 current x position)

P5.js program

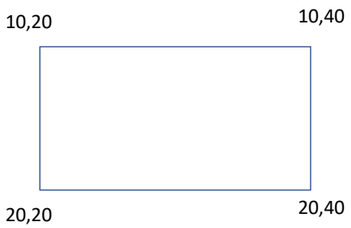
```
function setup()
{    createCanvas(400, 400);
}
 //my own function for drawing line
 function myLine(startX,startY,endX,endY){
  var slope = (endY - startY) / (endX - startX);
  var x = startX;                    //solving y=mx+c by making x as 0 so y=c means intercept is startY
  while (x <= endX)
 {   var y = slope * (x - startX) + startY;
     set(x, y,color(0));
      x += 1;
 }
 }
 function draw()
 {
  background(220);
  myLine(10,20,150,300);
  updatePixels();
 }
```



```
function setup() {
  // Create Canvas of given size
  createCanvas(400, 300);
}

function draw() {
  background(220);
  // Use color() function
  let c = color('green');
  // Use fill() function to fill color
  fill(c);
  // Draw a rectangle
  rect(50, 50, 300, 200);
}
```

Rectangle with Lines

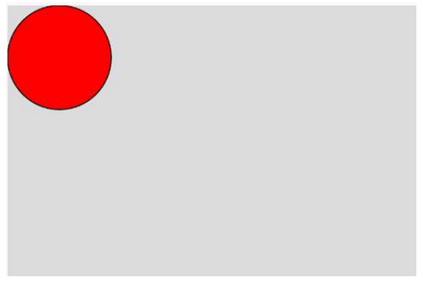
```
function setup() {
  createCanvas(400, 400);
}

function draw() {
  background(220);
  stroke(0);
  line(10,20,10,40);
  line(20,20,20,40);
  line(10,20,20,20);
  line(10,40, 20,40);

}
```



Circle

circle(x,y,d)

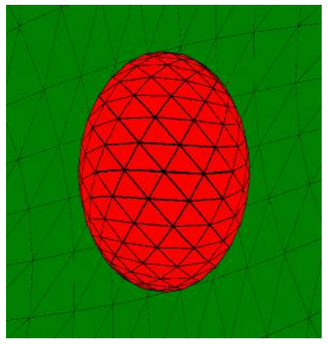
```
function setup() {
  // Create Canvas of given size
  createCanvas(400, 300);
}

function draw() {
  background(220);
  // Use color() function
  let c = color('red');
  // Use fill() function to fill color
  fill(c);
  // Draw a rectangle
  recirclect(50, 50, 100);
}
```



Ellipse

- ellipse(50, 50, 100,50);
- Sphere(100);

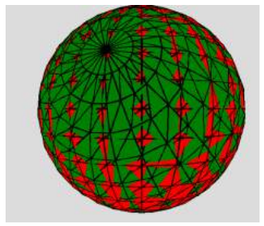
Animation
Scaling and rotation



```
function setup() {
   // Create Canvas of given size
   createCanvas(400, 300,WEBGL);
}

function draw() {
   background(220);
   // Use color() function
   let c = color('red');
   let c1=color('green');
   // Use fill() function to fill color
   fill(c);
   rotate(frameCount*0.03);
   sphere(100);

   fill(c1);
  // Rotate
   scale(frameCount * 0.01);
   rotate(frameCount*0.03);
   sphere(100);

}
```

Simple Animation with two spheres

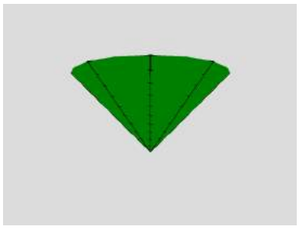
```
function setup() {
   // Create Canvas of given size
   createCanvas(400, 300,WEBGL);
}

function draw() {
   background(220);
   // Use color() function
   let c = color('red');
   let c1=color('green');
   // Use fill() function to fill color
   fill(c);
   sphere(100);
   fill(c1);
  // Rotate
   rotateX(frameCount * 0.01);
   rotate(frameCount*0.03);
    sphere(100);
}
```

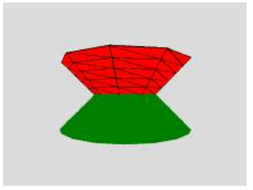


Cones

- `cone( radius, height, detailX, detailY, cap )`
- Last three parameters are optional. (try executing by // rotate(PI); and rotate(PI); see the difference)

```
function setup() {
   // Create Canvas of given size
   createCanvas(400, 300,WEBGL);
}

function draw() {

   background(220);

   // Use color() function
   let c = color('red');
   let c1=color('green');
   // Use fill() function to fill color

 fill(c);
 rotateY(frameCount*0.03);
 cone(100, 100, 10, 10, true);
 fill(c1);
 rotateY(frameCount*0.03);
// rotate(PI);
 cone(100, 100, 20, 20, true);

}
```





Geometric Manipulation

Anupama Namburu
Department of CSE
VIT-AP University, Amaravati

What do you understand in this video?

https://www.youtube.com/watch?v=lR0kyOdT86E&feature=youtu.be

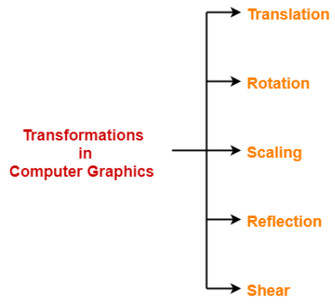
- Introduction to geometric transformations (T,R,S)
- Sets & Staging
- Compositing
- Camera movement
- Computer animation

Module 2

Module No. 2	Geometric Manipulation	8 hours
• Homogeneous coordinates • Affine transformations (translation, rotation, scaling, shear) • Concatenation • Matrix stacks and • Use of model view matrix in OpenGL for these operations		

Geometric Transformation

- **Transformation** is a process of modifying and re-positioning the existing graphics.
- **Geometric Transformation**: The object itself is transformed relative to the coordinate system or background.
- 2D and 3D transformations



Transformation

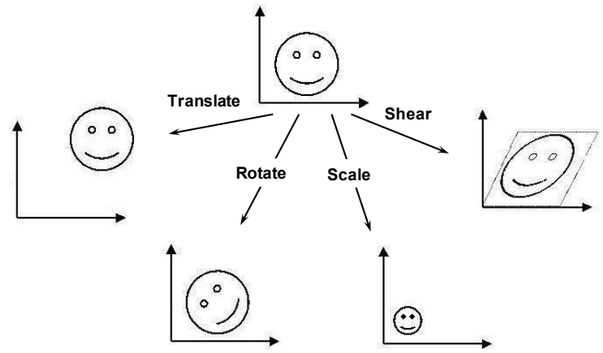
- Transformation plays a major role in graphics
- All the transformation can be represented as matrix



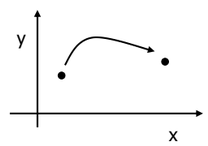
2D transformation

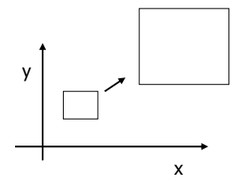
- Considering a 2D object , transformation is to change the object's
  - Positioning(Translation)
  - Orientation(rotation)
  - Size(Scaling)
  - Shapes(Shearing)
  - Mirroring(Reflection)

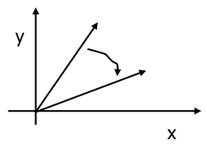
2D Geometrical Transformations



2D Transformations







OpenGl transformations:

glTranslatef (tx, ty, tz);

glRotatef (theta, vx, vy, vz)

glScalef (sx,sy,sz)

.....

P5.js

Translate()

Rotate()

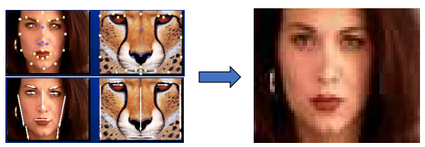
Scale()

...

Applications of 2D Transformations

- 2D geometric transformations

- Animation
- Image warping
- Face morphing
- Cricket



Translation

- A translation moves an object to a different position on the screen. You can translate a point in 2D by adding translation coordinate (tx, ty) to the original coordinate X,Y to get the new coordinate X',Y'.

- We translate objects in the (x, y) plane to new positions by adding translation vectors to the coordinates of their vertices.

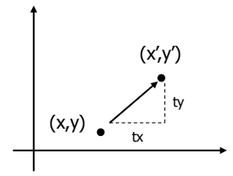
1. Translation

- Re-position a point along a straight line
- Given a point (x,y), and the translation distance (tx,ty)

The new point: (x′, y′)

$x' = x + tx$

$y' = y + ty$

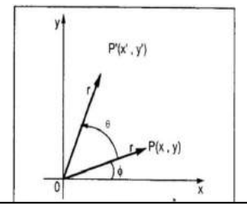


OR $P' = P + T$ where $P' = \begin{vmatrix} x' \\ y' \end{vmatrix}$ $p = \begin{vmatrix} x \\ y \end{vmatrix}$ $T = \begin{vmatrix} tx \\ ty \end{vmatrix}$

2. Rotation

- In rotation, we rotate the object at particular angle θ theta from its origin. From the following figure, we can see that the point P X,Y is located at angle ϕ from the horizontal X coordinate with distance r from the origin.

- Let us suppose you want to rotate it at the angle θ. After rotating it to a new location, you will get a new point P' X',Y'.



- Using standard trigonometric the original coordinate of point P(X,Y) can be represented as (Clockwise - -ve && anti cloak +ve)
- X=rcosφ.....(1)
- Y=rsinφ.....(2)

Same way we can represent the point P' X',Y' as –

- x'=rcos(φ+θ)=rcosφcosθ−rsinφsinθ.....(3)
- y'=rsin(φ+θ)=rcosφsinθ+rsinφcosθ.....(4)

Substituting equation 1 & 2 in 3 & 4 respectively, we will get

- x'=xcosθ−ysinθ
- y'=xsinθ+ycosθ

- Representing the above equation in matrix form,

$$[X'Y'] = [XY]\begin{bmatrix} cos\theta & sin\theta \\ -sin\theta & cos\theta \end{bmatrix} OR$$

P' = P . R

Where R is the rotation matrix

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

3.Scaling

- To change the size of an object, scaling transformation is used. In the scaling process, you either expand or compress the dimensions of the object.
- Scaling can be achieved by multiplying the original coordinates of the object with the scaling factor to get the desired result.

- Let us assume that the original coordinates are X,Y, the scaling factors are (S_X, S_Y) and the produced coordinates are X',Y'. This can be mathematically represented as shown below –

$$X' = X \cdot S_X \quad \text{and} \quad Y' = Y \cdot S_Y$$

The scaling factor S_X, S_Y scales the object in X and Y direction respectively. The above equations can also be represented in matrix form as below –

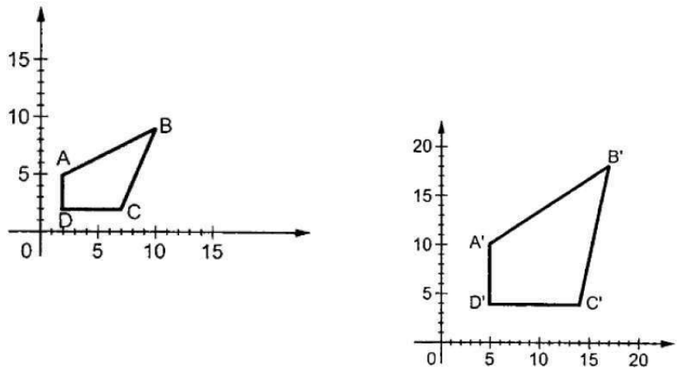
$$\begin{pmatrix} X' \\ Y' \end{pmatrix} = \begin{pmatrix} X \\ Y \end{pmatrix}\begin{bmatrix} S_x & 0 \\ 0 & S_y \end{bmatrix}$$

OR

P' = P . S

Where S is the scaling matrix. The scaling process is shown in the following figure.

The scaling process is shown in the following figur



If we provide values less than 1 to the scaling factor S, then we can reduce the size of the object. If we provide values greater than 1, then we can increase the size of the object.

Review…

- Translate: P’ = P+T
- Scale: P’ = S.P
- Rotate: P’ = R.P

- Spot the odd one out…
  - Multiplying versus adding matrix…
  - Ideally, all transformations would be the same..
    - easier to code
- Solution: Homogeneous Coordinates

Put it all together

- Translation:

$$\begin{vmatrix} x' \\ y' \end{vmatrix} = \begin{vmatrix} x \\ y \end{vmatrix} + \begin{vmatrix} t_x \\ t_y \end{vmatrix}$$

- Rotation:

$$\begin{vmatrix} x' \\ y' \end{vmatrix} = \begin{vmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{vmatrix} \cdot \begin{vmatrix} t_x \\ t_y \end{vmatrix}$$

- Scaling:

$$\begin{vmatrix} x' \\ y' \end{vmatrix} = \begin{vmatrix} s_x & 0 \\ 0 & s_y \end{vmatrix} \cdot \begin{vmatrix} t_x \\ t_y \end{vmatrix}$$

Points to reminder

- Though we are converting 2d projection coordinates into 3d coordinates, the formulae of Rotation, Translation, Scaling should not be changed.

3x3 2D Translation Matrix

$$\begin{vmatrix} x' \\ y' \end{vmatrix} = \begin{vmatrix} x \\ y \end{vmatrix} + \begin{vmatrix} tx \\ ty \end{vmatrix}$$

Use 3 x 1 vector

$$\begin{vmatrix} x' \\ y' \\ 1 \end{vmatrix} = \begin{vmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{vmatrix} * \begin{vmatrix} x \\ y \\ 1 \end{vmatrix}$$

- Note that now it becomes a matrix-vector multiplication
- We are converting to homogenous coordinate system.

All the transformation should be treated in a consistent way

- Translation → addition
- Rotation and Scaling → multiplication

Homogenous Coordinates

- If you want to perform sequence of transformation then you have to use 3 X 3 matrix. For.ex. we need to follow a sequential process –
  - Translate the coordinates,
  - Rotate the translated coordinates, and then
  - Scale the rotated coordinates to complete the composite transformation.
- So convert 2X2 tranf.matrix to 3X3 tranf.matric by adding one extra dummy coordinate w.
- In this way, we can represent the point by 3 numbers instead of 2 numbers, which is called Homogenous Coordinate system.
- The homogeneous coordinates representation of (X, Y) is (X, Y, 1).

All the transformation should be treated in a consistent way

- This can only happen, if all the points are expressed in homogeneous coordinates.
- Finally, in homogeneous coordinates, all the three transformation can be treated as multiplication.

Homogeneous coordinates

- Represents coordinates in 2 dimensions with a vector

$$\begin{bmatrix} x \\ y \end{bmatrix} \Rightarrow \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \longrightarrow \begin{bmatrix} x \\ y \end{bmatrix} \Rightarrow \begin{bmatrix} x \\ y \\ 1w \end{bmatrix}$$

- Basically we are adding a 3rd coordinate to every 2d point
- Now (x,y) become (x,y,w)

3x3 2D Translation Matrix

$$\begin{vmatrix} x' \\ y' \end{vmatrix} = \begin{vmatrix} x \\ y \end{vmatrix} + \begin{vmatrix} tx \\ ty \end{vmatrix}$$

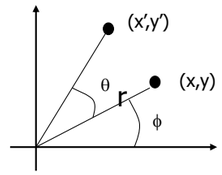
Use 3 x 1 vector

$$\begin{vmatrix} x' \\ y' \\ 1 \end{vmatrix} = \begin{vmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{vmatrix} * \begin{vmatrix} x \\ y \\ 1 \end{vmatrix}$$

- Note that now it becomes a matrix-vector multiplication
- We are converting to homogenous coordinate system.

3x3 2D Rotation Matrix

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



$$\begin{vmatrix} x' \\ y' \\ 1 \end{vmatrix} = \begin{vmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} x \\ y \\ 1 \end{vmatrix}$$

3x3 2D Scaling Matrix

$$\begin{vmatrix} x' \\ y' \end{vmatrix} = \begin{vmatrix} Sx & 0 \\ 0 & Sy \end{vmatrix} \begin{vmatrix} x \\ y \end{vmatrix}$$

$$\begin{vmatrix} x' \\ y' \\ 1 \end{vmatrix} = \begin{vmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{vmatrix} * \begin{vmatrix} x \\ y \\ 1 \end{vmatrix}$$

Why Use 3x3 Matrices?

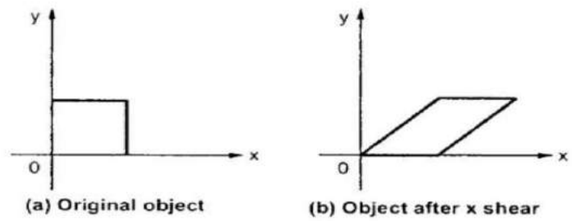
- So that we can perform all transformations using matrix/vector multiplications
- This allows us to *pre-multiply* all the matrices together
- The point (x,y) needs to be represented as (x,y,1) -> this is called Homogeneous coordinates!

4.Shearing

- A transformation that slants the shape of an object is called the shear transformation. There are two shear transformations X-Shear and Y-Shear.
- One shifts X coordinates values and other shifts Y coordinate values. However; in both the cases only one coordinate changes its coordinates and other preserves its values. Shearing is also termed as Skewing.

Types

- X-Shear
- The X-Shear preserves the Y coordinate and changes are made to X coordinates, which causes the vertical



(a) Original object (b) Object after x shear

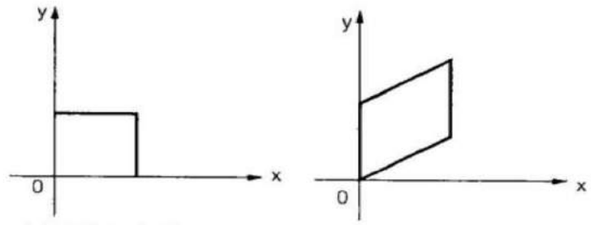
The transformation matrix for X-Shear can be represented as –

$$X_{sh} = \begin{bmatrix} 1 & shx & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$X' = X + Sh_x \cdot Y$

$Y' = Y$

- Y-Shear
- The Y-Shear preserves the y coordinates and changes the x coordinates which causes the horizontal lines to t
s



(a) Original object (b) Object after y shear

The Y-Shear can be represented in matrix from as –

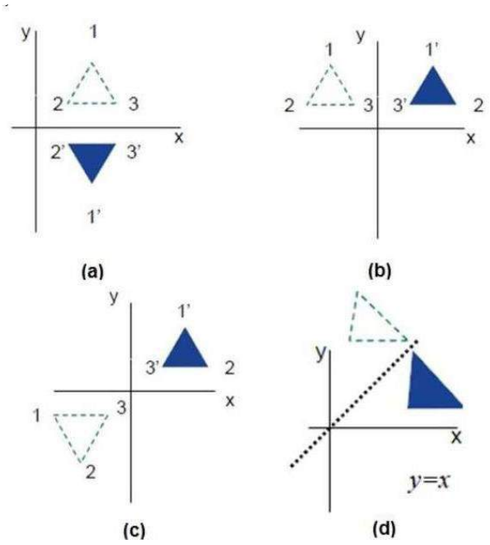
$$Y_{sh} \begin{bmatrix} 1 & 0 & 0 \\ shy & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$Y' = Y + Sh_y \cdot X$

$X' = X$

5.Reflections

- Reflection is the mirror image of original object. In other words, we can say that it is a rotation operation with 180°. In reflection transformation, the size of the object does not change.
- The following figures show reflections with respect to X and Y axes, and about the origin respectively.

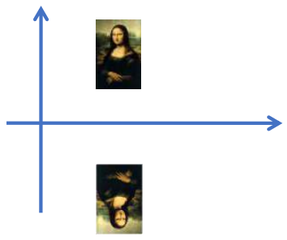


Reflection ex.

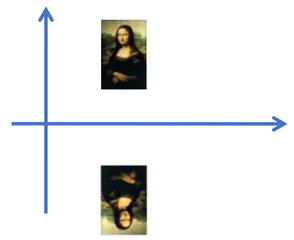
Reflection

Reflection

Reflection about X-axis

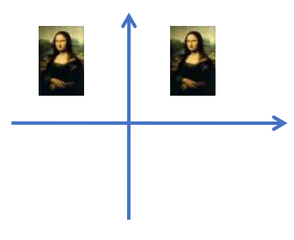


Reflection about X-axis

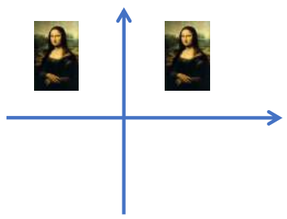


$$\begin{vmatrix} x \\ y \\ 1 \end{vmatrix} = \begin{vmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{vmatrix} * \begin{vmatrix} x \\ y \\ 1 \end{vmatrix}$$

Reflection about Y-axis



Reflection about Y-axis



$$\begin{vmatrix} x \\ y \\ 1 \end{vmatrix} = \begin{vmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix} * \begin{vmatrix} x \\ y \\ 1 \end{vmatrix}$$

Refer the link for problems

- http://csis.pace.edu/~marchese/CG/Lect6/cg_l6_part1.htm

Probelms

- **Problem-01:**
- Given a circle C with radius 10 and center coordinates (1, 4). Apply the translation with distance 5 towards X axis and 1 towards Y axis. Obtain the new coordinates of C without changing its radius.

Solution-

Given-

- Old center coordinates of C = $(X_{old}, Y_{old}) = (1, 4)$
- Translation vector = $(T_x, T_y) = (5, 1)$

Let the new center coordinates of C = (X_{new}, Y_{new}).

Applying the translation equations, we have-

- $X_{new} = X_{old} + T_x = 1 + 5 = 6$
- $Y_{new} = Y_{old} + T_y = 4 + 1 = 5$

Thus, New center coordinates of C = (6, 5).

In matrix form, the new center coordinates of C after translation may be obtained as-

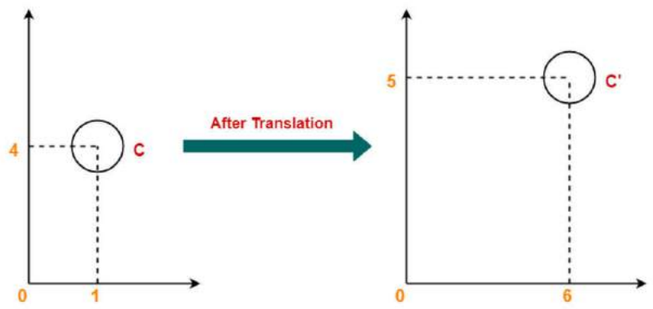
$$\begin{bmatrix} X_{new} \\ Y_{new} \end{bmatrix} = \begin{bmatrix} X_{old} \\ Y_{old} \end{bmatrix} + \begin{bmatrix} T_x \\ T_y \end{bmatrix}$$

$$\begin{bmatrix} X_{new} \\ Y_{new} \end{bmatrix} = \begin{bmatrix} 1 \\ 4 \end{bmatrix} + \begin{bmatrix} 5 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} X_{new} \\ Y_{new} \end{bmatrix} = \begin{bmatrix} 6 \\ 5 \end{bmatrix}$$

Thus, New center coordinates of C = (6, 5).

Thus, New center coordinates of C = (6, 5).



Problem-02:

- Given a square with coordinate points A(0, 3), B(3, 3), C(3, 0), D(0, 0). Apply the translation with distance 1 towards X axis and 1 towards Y axis. Obtain the new coordinates of the square.

Given-

- Old coordinates of the square = A (0, 3), B(3, 3), C(3, 0), D(0, 0)
- Translation vector = $(T_x, T_y) = (1, 1)$

For Coordinates A(0, 3)

Let the new coordinates of corner A = (X_{new}, Y_{new}).

Applying the translation equations, we have-

- $X_{new} = X_{old} + T_x = 0 + 1 = 1$
- $Y_{new} = Y_{old} + T_y = 3 + 1 = 4$

Thus, New coordinates of corner A = (1, 4).

For Coordinates B(3, 3)

Let the new coordinates of corner B = (X_{new}, Y_{new}).

Applying the translation equations, we have-

- $X_{new} = X_{old} + T_x = 3 + 1 = 4$
- $Y_{new} = Y_{old} + T_y = 3 + 1 = 4$

Thus, New coordinates of corner B = (4, 4).

For Coordinates C(3, 0)

Let the new coordinates of corner C = (X_{new}, Y_{new}).

Applying the translation equations, we have-

- $X_{new} = X_{old} + T_x = 3 + 1 = 4$
- $Y_{new} = Y_{old} + T_y = 0 + 1 = 1$

Thus, New coordinates of corner C = (4, 1).

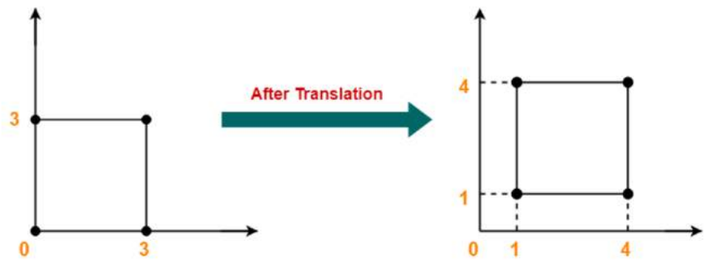
For Coordinates D(0, 0)

Let the new coordinates of corner D = (X_{new}, Y_{new}).

Applying the translation equations, we have-

- $X_{new} = X_{old} + T_x = 0 + 1 = 1$
- $Y_{new} = Y_{old} + T_y = 0 + 1 = 1$

Thus, New coordinates of corner D = (1, 1).



Problem-03:

- Given a line segment with starting point as (0, 0) and ending point as (4, 4). Apply 30 degree rotation anticlockwise direction on the line segment and find out the new coordinates of the line.

Given-

- Old ending coordinates of the line = $(X_{old}, Y_{old}) = (4, 4)$
- Rotation angle = $\theta = 30^o$

We rotate a straight line by its end points with the same angle. Then, we re-draw a line between the new end points.

- Let new ending coordinates of the line after rotation = (X_{new}, Y_{new}).
- Applying the rotation equations, we have-

$$\begin{bmatrix} X_{new} \\ Y_{new} \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \times \begin{bmatrix} X_{old} \\ Y_{old} \end{bmatrix}$$

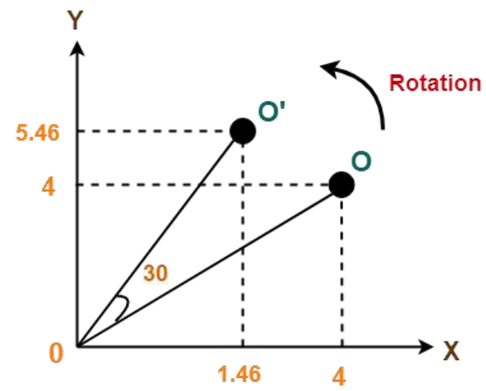
$$\begin{bmatrix} X_{new} \\ Y_{new} \end{bmatrix} = \begin{bmatrix} \cos 30 & -\sin 30 \\ \sin 30 & \cos 30 \end{bmatrix} \times \begin{bmatrix} 4 \\ 4 \end{bmatrix}$$

$$\begin{bmatrix} X_{new} \\ Y_{new} \end{bmatrix} = \begin{bmatrix} 4 \times \cos 30 - 4 \times \sin 30 \\ 4 \times \sin 30 + 4 \times \cos 30 \end{bmatrix}$$

$$\begin{bmatrix} X_{new} \\ Y_{new} \end{bmatrix} = \begin{bmatrix} 4 \times \cos 30 - 4 \times \sin 30 \\ 4 \times \sin 30 + 4 \times \cos 30 \end{bmatrix}$$

$$\begin{bmatrix} X_{new} \\ Y_{new} \end{bmatrix} = \begin{bmatrix} 1.46 \\ 5.46 \end{bmatrix}$$

Thus, New ending coordinates of the line after rotation = (1.46, 5.46).



Problem 04:

Given a triangle with corner coordinates (0, 0), (1, 0) and (1, 1). Rotate the triangle by 90 degree anticlockwise direction and find out the new coordinates.

Given:

We rotate a polygon by rotating each vertex of it with the same rotation angle.

Given-

- Old corner coordinates of the triangle = A (0, 0), B(1, 0), C(1, 1)
- Rotation angle = θ = 90°

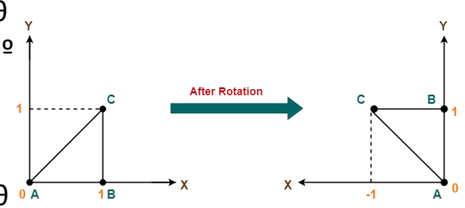
- $x' = x\cos\theta - y\sin\theta$
- $y' = x\sin\theta + y\cos\theta$

- **For Coordinates A(0, 0)**
- Let the new coordinates of corner A after rotation = (X_{new}, Y_{new}).
- Applying the rotation equations, we have-
- X_{new}
- = X_{old} x cosθ – Y_{old} x sinθ
- = 0 x cos90º – 0 x sin90º
- = 0
- Y_{new}
- = X_{old} x sinθ + Y_{old} x cosθ
- = 0 x sin90º + 0 x cos90º
- = 0
- Thus, New coordinates of corner A after rotation = (0, 0).

- **For Coordinates B(1, 0)**
- Let the new coordinates of corner B after rotation = (X_{new}, Y_{new}).
- X_{new}
- = X_{old} x cosθ – Y_{old} x sinθ
- = 1 x cos90° – 0 x sin90°
- = 0
- Y_{new}
- = X_{old} x sinθ + Y_{old} x cosθ
- = 1 x sin90° + 0 x cos90°
- = 1 + 0
- = 1
- Thus, New coordinates of corner B after rotation = (0, 1).

- **For Coordinates C(1, 1)**
- Let the new coordinates of corner C after rotation = (X_{new}, Y_{new}).
- X_{new}
- = X_{old} x cosθ – Y_{old} x sinθ
- = 1 x cos90º – 1 x sin90º
- = 0 – 1
- = -1
- Y_{new}
- = X_{old} x sinθ + Y_{old} x cosθ
- = 1 x sin90º + 1 x cos90º
- = 1 + 0
- = 1
- Thus, New coordinates of corner C after rotation = (-1, 1).

- Thus, New coordinates of the triangle after rotation = A (0, 0), B(0, 1), C(-1, 1).



Problem-05:

- Given a square object with coordinate points A(0, 3), B(3, 3), C(3, 0), D(0, 0). Apply the scaling parameter 2 towards X axis and 3 towards Y axis and obtain the new coordinates of the object.

Solution-

Given-

- Old corner coordinates of the square = A (0, 3), B(3, 3), C(3, 0), D(0, 0)
- Scaling factor along X axis = 2
- Scaling factor along Y axis = 3

For Coordinates A(0, 3)

Let the new coordinates of corner A after scaling = (X_{new}, Y_{new}).

Applying the scaling equations, we have-

- $X_{new} = X_{old} \times S_x = 0 \times 2 = 0$
- $Y_{new} = Y_{old} \times S_y = 3 \times 3 = 9$

Thus, New coordinates of corner A after scaling = (0, 9).

For Coordinates B(3, 3)

Let the new coordinates of corner B after scaling = (X_{new}, Y_{new}).

Applying the scaling equations, we have-

- $X_{new} = X_{old} \times S_x = 3 \times 2 = 6$
- $Y_{new} = Y_{old} \times S_y = 3 \times 3 = 9$

Thus, New coordinates of corner B after scaling = (6, 9).

For Coordinates C(3, 0)

Let the new coordinates of corner C after scaling = (X_{new}, Y_{new}).

Applying the scaling equations, we have-

- $X_{new} = X_{old} \times S_x = 3 \times 2 = 6$
- $Y_{new} = Y_{old} \times S_y = 0 \times 3 = 0$

Thus, New coordinates of corner C after scaling = (6, 0).

For Coordinates D(0, 0)

Let the new coordinates of corner D after scaling = (X_{new}, Y_{new}).

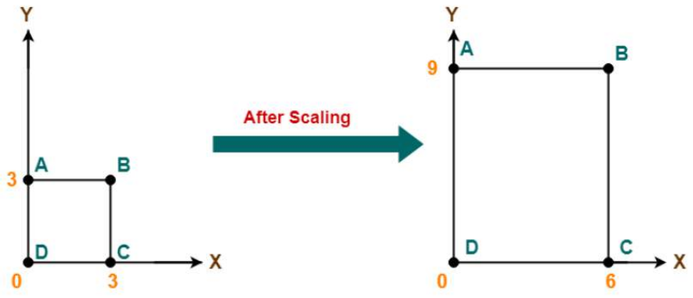
Applying the scaling equations, we have-

- $X_{new} = X_{old} \times S_x = 0 \times 2 = 0$
- $Y_{new} = Y_{old} \times S_y = 0 \times 3 = 0$

Thus, New coordinates of corner D after scaling = (0, 0).

Thus, New coordinates of the square after scaling = A (0, 9), B(6, 9), C(6, 0), D(0, 0).

Thus, New coordinates of the square after scaling = A (0, 9), B(6, 9), C(6, 0), D(0, 0).



Reflection

- Let-
- Initial coordinates of the object O = (X_{old}, Y_{old})
- New coordinates of the reflected object O after reflection = (X_{new}, Y_{new})
- **Reflection On X-Axis:**
- This reflection is achieved by using the following reflection equations-
- $X_{new} = X_{old}$
- $Y_{new} = -Y_{old}$

In Matrix form, the above reflection equations may be represented as-

$$\begin{bmatrix} X_{new} \\ Y_{new} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \times \begin{bmatrix} X_{old} \\ Y_{old} \end{bmatrix}$$

Reflection Matrix

(Reflection Along X Axis)

For homogeneous coordinates, the above reflection matrix may be represented as a 3 x 3 matrix as-

$$\begin{bmatrix} X_{new} \\ Y_{new} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} X_{old} \\ Y_{old} \\ 1 \end{bmatrix}$$

Reflection Matrix

(Reflection Along X Axis)

(Homogeneous Coordinates Representation)

Reflection On Y-Axis:

- This reflection is achieved by using the following reflection equations-
- Xnew = -Xold
- Ynew = Yold
- In Matrix form, the above reflection equations may be represented as-

$$\begin{bmatrix} X_{new} \\ Y_{new} \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} X_{old} \\ Y_{old} \end{bmatrix}$$

Reflection Matrix
(Reflection Along Y Axis)

For homogeneous coordinates, the above reflection matrix may be represented as a 3 x 3 matrix as-

$$\begin{bmatrix} X_{new} \\ Y_{new} \\ 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} X_{old} \\ Y_{old} \\ 1 \end{bmatrix}$$

Reflection Matrix
(Reflection Along Y Axis)
(Homogeneous Coordinates Representation)

Problem-06:

- Given a triangle with coordinate points A(3, 4), B(6, 4), C(5, 6). Apply the reflection on the X axis and obtain the new coordinates of the object.
- **Solution-**

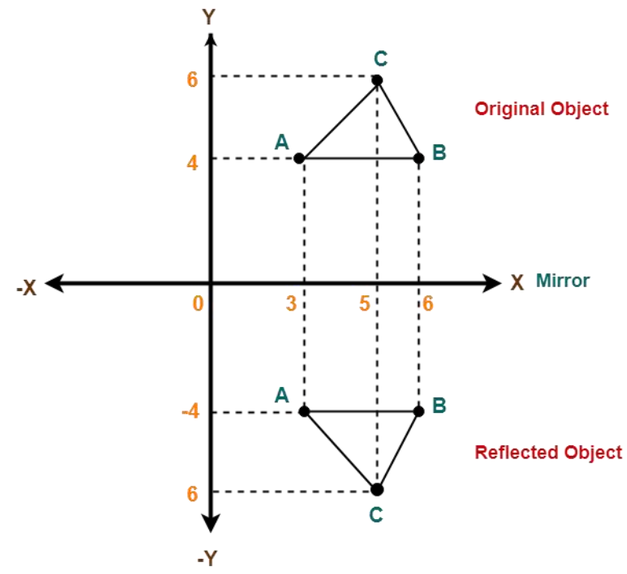
Given-

- Old corner coordinates of the triangle = A (3, 4), B(6, 4), C(5, 6)
- Reflection has to be taken on the X axis

- **For Coordinates A(3, 4)**
- Let the new coordinates of corner A after reflection = (X_{new}, Y_{new}).
- Applying the reflection equations, we have-
- $X_{new} = X_{old} = 3$
- $Y_{new} = -Y_{old} = -4$
- Thus, New coordinates of corner A after reflection = (3, -4).

- **For Coordinates B(6, 4)**
- Let the new coordinates of corner B after reflection = (X_{new}, Y_{new}).
- Applying the reflection equations, we have-
- $X_{new} = X_{old} = 6$
- $Y_{new} = -Y_{old} = -4$
- Thus, New coordinates of corner B after reflection = (6, -4).

- **For Coordinates C(5, 6)**
- Let the new coordinates of corner C after reflection = (X_{new}, Y_{new}).
- Applying the reflection equations, we have-
- $X_{new} = X_{old} = 5$
- $Y_{new} = -Y_{old} = -6$
- Thus, New coordinates of corner C after reflection = (5, -6).
- Thus, New coordinates of the triangle after reflection = A (3, -4), B(6, -4), C(5, -6).

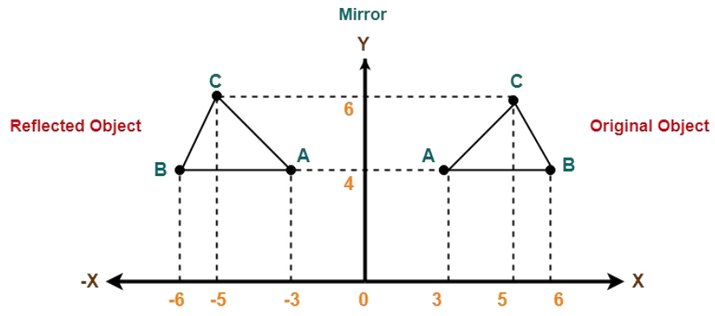


Problem-07:

- Given a triangle with coordinate points A(3, 4), B(6, 4), C(5, 6). Apply the reflection on the Y axis and obtain the new coordinates of the object.

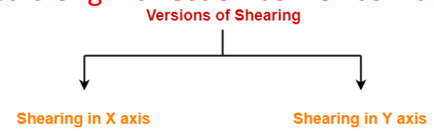
- **Solution-**
- Given-
- Old corner coordinates of the triangle = A (3, 4), B(6, 4), C(5, 6)
- Reflection has to be taken on the Y axis

- Thus, New coordinates of the triangle after reflection = A (-3, 4), B(-6, 4), C(-5, 6).



Shearing

- 2D Shearing is an ideal technique to change the shape of an existing object in a two dimensional plane. In a two dimensional plane, the object size can be changed along X direction as well as Y direction.



- Consider a point object O has to be sheared in a 2D plane.
- Let-
- Initial coordinates of the object O = (X_{old}, Y_{old})
- Shearing parameter towards X direction = Sh_x
- Shearing parameter towards Y direction = Sh_y
- New coordinates of the object O after shearing = (X_{new}, Y_{new})

Shearing in X Axis-

- Shearing in X axis is achieved by using the following shearing equations-
- $X_{new} = X_{old} + Sh_x \times Y_{old}$
- $Y_{new} = Y_{old}$
- In Matrix for ations may be represented a

$$\begin{bmatrix} X_{new} \\ Y_{new} \end{bmatrix} = \begin{bmatrix} 1 & Sh_x \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} X_{old} \\ Y_{old} \end{bmatrix}$$

Shearing Matrix

(In X axis)

For homogeneous coordinates, the above shearing matrix may be represented as a 3 x 3 matrix as-

$$\begin{bmatrix} X_{new} \\ Y_{new} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ Sh_x & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} X_{old} \\ Y_{old} \\ 1 \end{bmatrix}$$

Shearing Matrix

(In X axis)

(Homogeneous Coordinates Representation)

Shearing in Y Axis-

- Shearing in Y axis is achieved by using the following shearing equations-
- $X_{new} = X_{old}$
- $Y_{new} = Y_{old} +$
- In Matrix fo uations may be represented

$$\begin{bmatrix} X_{new} \\ Y_{new} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ Sh_y & 1 \end{bmatrix} \times \begin{bmatrix} X_{old} \\ Y_{old} \end{bmatrix}$$

Shearing Matrix

(In Y axis)

For homogeneous coordinates, the above shearing matrix may be represented as a 3 x 3 matrix as-

$$\begin{bmatrix} X_{new} \\ Y_{new} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & Sh_y & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} X_{old} \\ Y_{old} \\ 1 \end{bmatrix}$$

Shearing Matrix

(In Y axis)

(Homogeneous Coordinates Representation)

Problem-08:

- Given a triangle with points (1, 1), (0, 0) and (1, 0). Apply shear parameter 2 on X axis and 2 on Y axis and find out the new coordinates of the object.

- **Solution-**

Given-

- Old corner coordinates of the triangle = A (1, 1), B(0, 0), C(1, 0)
- Shearing parameter towards X direction (Sh_x) = 2
- Shearing parameter towards Y direction (Sh_y) = 2

Shearing in X Axis-

For Coordinates A(1, 1)

Let the new coordinates of corner A after shearing = (X_{new}, Y_{new}).

Applying the shearing equations, we have-

- $X_{new} = X_{old} + Sh_x \times Y_{old} = 1 + 2 \times 1 = 3$
- $Y_{new} = Y_{old} = 1$

Thus, New coordinates of corner A after shearing = (3, 1).

For Coordinates B(0, 0)

Let the new coordinates of corner B after shearing = (X_{new}, Y_{new}).

Applying the shearing equations, we have-

- $X_{new} = X_{old} + Sh_x \times Y_{old} = 0 + 2 \times 0 = 0$
- $Y_{new} = Y_{old} = 0$

Thus, New coordinates of corner B after shearing = (0, 0).

For Coordinates C(1, 0)

Let the new coordinates of corner C after shearing = (X_{new}, Y_{new}).

Applying the shearing equations, we have-

- $X_{new} = X_{old} + Sh_x \times Y_{old} = 1 + 2 \times 0 = 1$
- $Y_{new} = Y_{old} = 0$

Thus, New coordinates of corner C after shearing = (1, 0).

Thus, New coordinates of the triangle after shearing in X axis = A (3, 1), B(0, 0), C(1, 0).

Shearing in Y Axis-

For Coordinates A(1, 1)

Let the new coordinates of corner A after shearing = (X_{new}, Y_{new}).

Applying the shearing equations, we have-

- $X_{new} = X_{old} = 1$
- $Y_{new} = Y_{old} + Sh_y \times X_{old} = 1 + 2 \times 1 = 3$

Thus, New coordinates of corner A after shearing = (1, 3).

For Coordinates B(0, 0)

Let the new coordinates of corner B after shearing = (X_{new}, Y_{new}).

Applying the shearing equations, we have-

- $X_{new} = X_{old} = 0$
- $Y_{new} = Y_{old} + Sh_y \times X_{old} = 0 + 2 \times 0 = 0$

Thus, New coordinates of corner B after shearing = (0, 0).

For Coordinates C(1, 0)

Let the new coordinates of corner C after shearing = (X_{new}, Y_{new}).

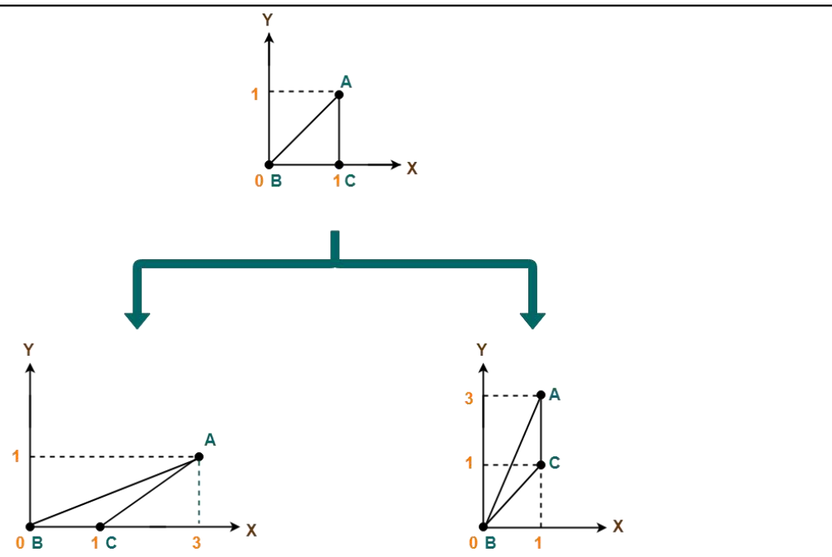
Applying the shearing equations, we have-

- $X_{new} = X_{old} = 1$
- $Y_{new} = Y_{old} + Sh_y \times X_{old} = 0 + 2 \times 1 = 2$

Thus, New coordinates of corner C after shearing = (1, 2).

Thus, New coordinates of the triangle after shearing in Y axis = A (1, 3), B(0, 0), C(1, 2).

74



Shearing in X Axis

Shearing in Y Axis

75

3D transformation

Anupama Namburu

From 2D to 3D

- Much +/- the same:
  - Translation, scaling
  - Homogeneous vectors: one extra coordinate
  - Matrices: 4x4
- Rotation more complex

3D Translation

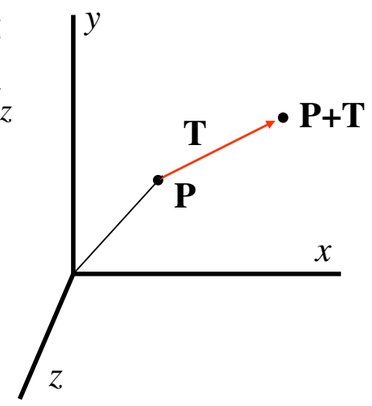
Translate over vector (t_x, t_y, t_z):

$x' = x + t_x,\ y' = y + t_y,\ z' = z + t_z$

or

$\mathbf{P}' = \mathbf{P} + \mathbf{T}$, met

$$\mathbf{P}' = \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix}, \mathbf{P} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \text{ en } \mathbf{T} = \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix}$$

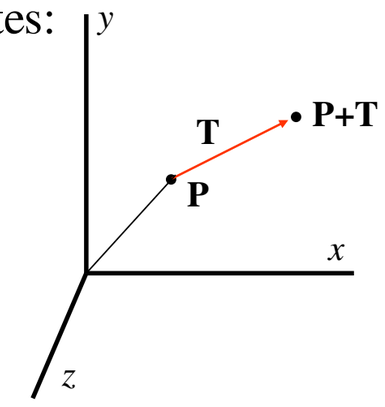


3D Translation

In 4D homogeneous coordinates:

$\mathbf{P}' = \mathbf{M}\mathbf{P}$, or

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$



3D Rotation

Rotate over angle α around z - as:

$x' = x\cos\alpha - y\sin\alpha$

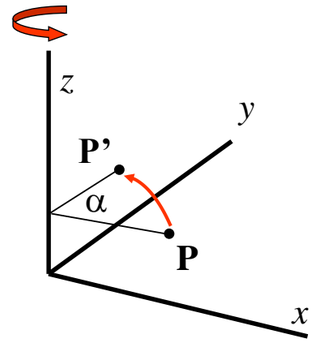
$y' = x\sin\alpha + y\cos\alpha$

$z' = z$

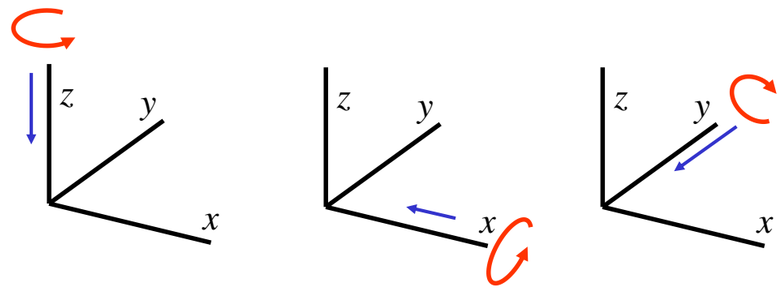
Or

$\mathbf{P}' = \mathbf{R}_z(\alpha)\mathbf{P}$, with

$$\mathbf{R}_z(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

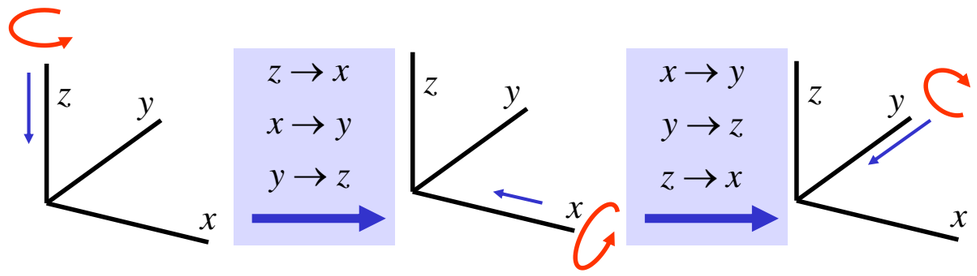


3D Rotation



Rotation around axis:
- Counterclockwise, viewed from rotation axis

3D Rotation



Rotation around axes:
Cyclic permutation coordinate axes

$x \to y \to z \to x$

Rotate over angle α around x - as:

$y' = y\cos\alpha - z\sin\alpha$

$z' = y\sin\alpha + z\cos\alpha$

$x' = x$

Or

$x \to y$
$y \to z$
$z \to x$

$\mathbf{P'} = \mathbf{R}_x(\alpha)\mathbf{P}$, with

$$\mathbf{R}_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

3D Rotation

Rotate over angle α around z - as:

$x' = x\cos\alpha - y\sin\alpha$

$y' = x\sin\alpha + y\cos\alpha$

$z' = z$

Or

$\mathbf{P'} = \mathbf{R}_z(\alpha)\mathbf{P}$, with

$$\mathbf{R}_z(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$z \to x$
$x \to y$
$y \to z$

Rotate over angle α around x - as:

$y' = y\cos\alpha - z\sin\alpha$

$z' = y\sin\alpha + z\cos\alpha$

$x' = x$

Or

$\mathbf{P'} = \mathbf{R}_x(\alpha)\mathbf{P}$, with

$$\mathbf{R}_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

3D Rotation around parallel axis

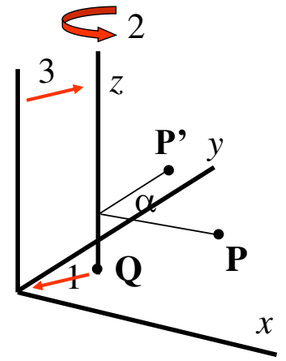
Rotation around axis, parallel to coordinate axis, through point **Q**.

For example. the z - as. Similar as 2D rotation :

1. Translate over -**Q**;
2. Rotate around z - axis;
3. Translate back over **Q**.

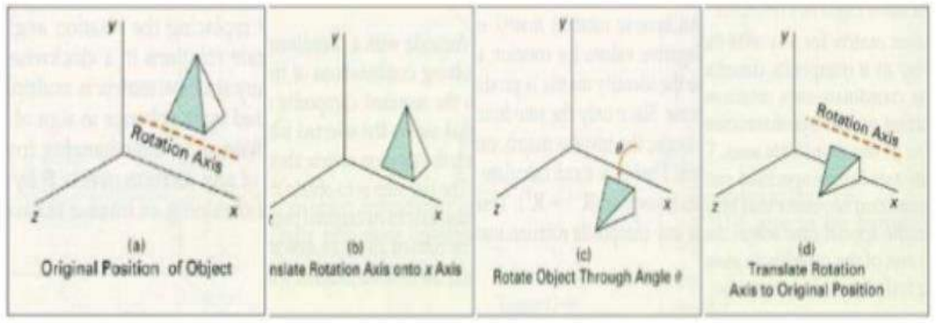
Or :

$\mathbf{P'} = \mathbf{T(Q)R_z}(\alpha)\mathbf{T(-Q)P}$



Rotation around parallel axis

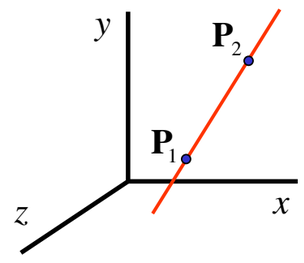
- Translate to coincide with one axis.
- Rotate along that axis.
- Inverse translate



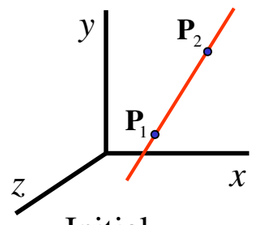
3D Rotation around arbitrary axis

Rotation around axis through two points $\mathbf{P}_1$ and $\mathbf{P}_2$.
More complex:

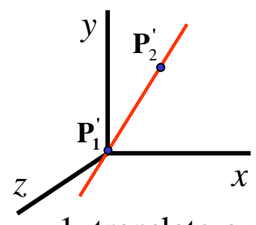
1. Translate such that axis goes through origin;
2. Rotate…
3. Translate back again.



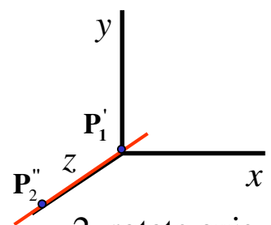
3D Rotation around arbitrary axis



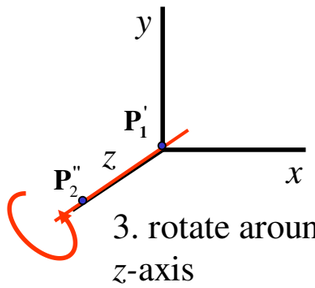
Initial



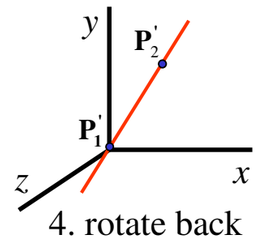
1. translate axis



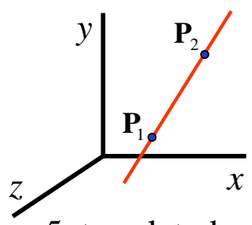
2. rotate axis



3. rotate around z-axis

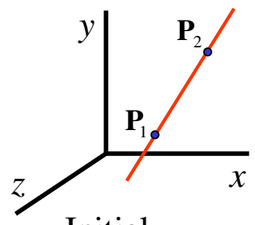


4. rotate back

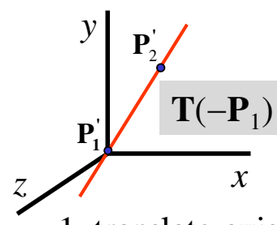


5. translate back

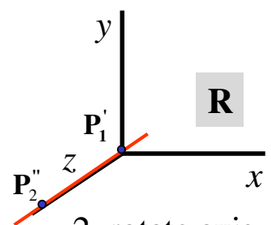
3D Rotation around arbitrary axis



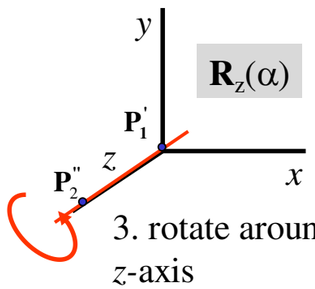
Initial



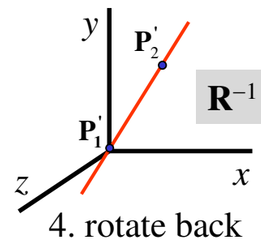
1. translate axis



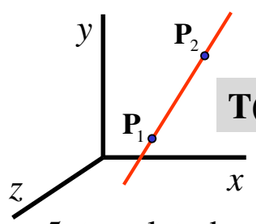
2. rotate axis



3. rotate around z-axis



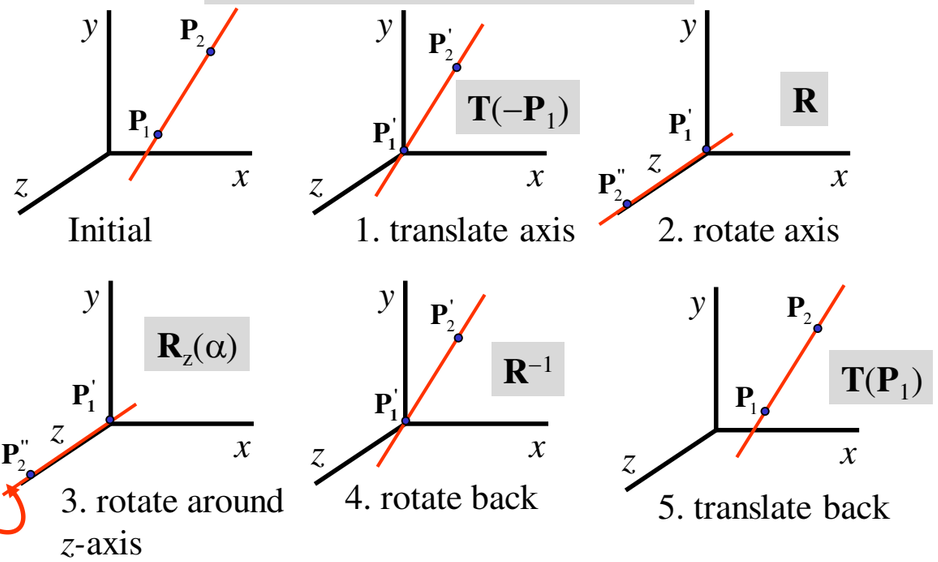
4. rotate back



5. translate back

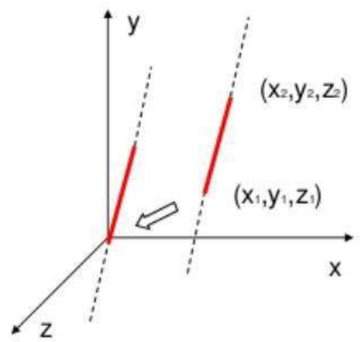
3D Rotation around arbitrary axis

$$\mathbf{M} = \mathbf{T}(\mathbf{P}_1)\,\mathbf{R}^{-1}\mathbf{R}_z(\alpha)\,\mathbf{R}\mathbf{T}(-\mathbf{P}_1)$$



Step 1:T(-P)

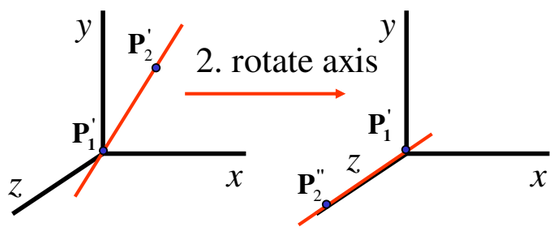
- Step 1. Translation



$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & -x_1 \\ 0 & 1 & 0 & -y_1 \\ 0 & 0 & 1 & -z_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Step 2: Rotate axis

step 2:rotate axis **R**



Find rotation such that rotation axis aligns with z-axis.
Two options:
1.Step by step =Rotate along X and Y to align to Z
2. Direct evaluation

Direct derivation-step2

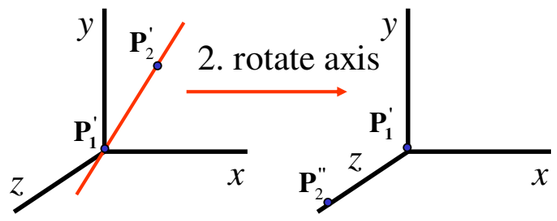
- $$R=\begin{bmatrix} \lambda/|V| & 0 & a/|V| & 0 \\ -ab/|V| & c/\lambda & b/|V| & 0 \\ -ac/|V| & -b/\lambda & c/|V| & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$V=ax+by+cz \quad ; \quad |v|=\sqrt{(a^2+b^2+c^2)}$

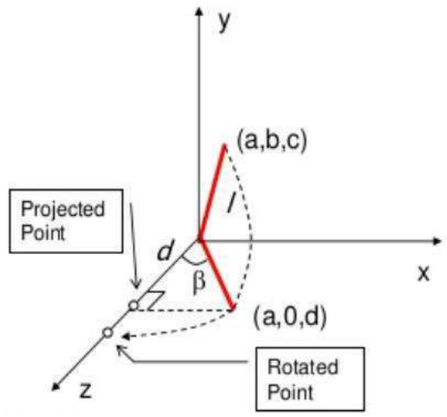
$\lambda=\sqrt{(b^2+c^2)}$

Step by step derivation of Step2

- Rotate along X to move point to XY plane
- Rotate along Y to align the point to Z plane



Rotate about *y* axis by ϕ



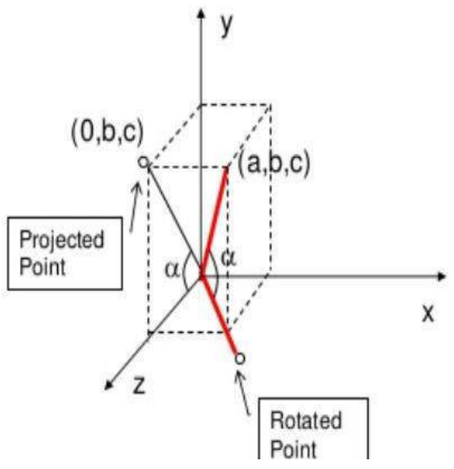
$$\sin\beta = \frac{a}{l}, \quad \cos\beta = \frac{d}{l}$$

$$l^2 = a^2 + b^2 + c^2 = a^2 + d^2$$

$$d = \sqrt{b^2 + c^2}$$

$$\mathbf{R}_y(\beta) = \begin{bmatrix} \cos\beta & 0 & -\sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} d/l & 0 & -a/l & 0 \\ 0 & 1 & 0 & 0 \\ a/l & 0 & d/l & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Establish $[\,T_R\,]^{\alpha}_{x}$ *x* axis



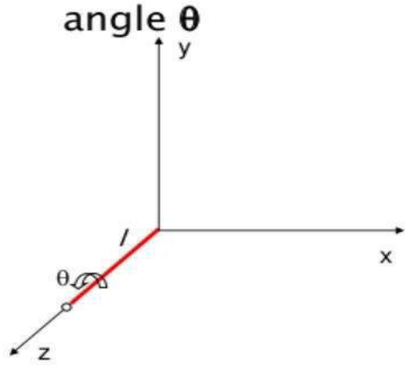
$$\sin\alpha = \frac{b}{\sqrt{b^2 + c^2}} = \frac{b}{d}$$

$$\cos\alpha = \frac{c}{\sqrt{b^2 + c^2}} = \frac{c}{d}$$

$$\mathbf{R}_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c/d & -b/d & 0 \\ 0 & b/d & c/d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Step 3:Rotate along Z-axis

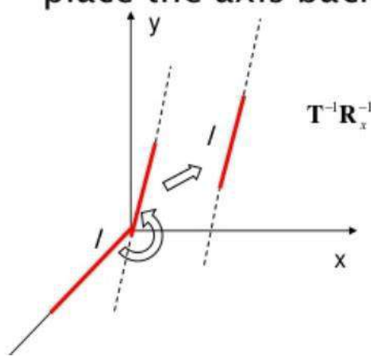
Rotate about *z* axis by the desired angle θ



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

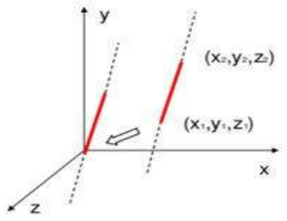
Step 4: rotate back

Apply the reverse transformation to place the axis back in its initial position



$$\mathbf{T}^{-1}\mathbf{R}_x^{-1}(\alpha)\mathbf{R}_y^{-1}(\beta) = \begin{bmatrix} 1 & 0 & 0 & -x_1 \\ 0 & 1 & 0 & -y_1 \\ 0 & 0 & 1 & -z_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha & 0 \\ 0 & -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\beta & 0 & \sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Step 5-Translate back



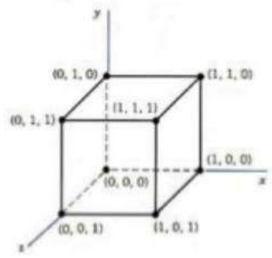
$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & x_1 \\ 0 & 1 & 0 & y_1 \\ 0 & 0 & 1 & z_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation along arbitrary axix is given by

$$\mathbf{M} = \mathbf{T}(\mathbf{P}_1)\,\mathbf{R}^{-1}\mathbf{R}_z(\alpha)\,\mathbf{RT}(-\mathbf{P}_1)$$

Example

Find the new coordinates of a unit cube 90º-rotated about an axis defined by its endpoints A(2,1,0) and B(3,3,1).

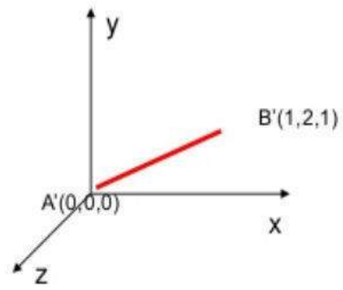


$$\begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

-
-

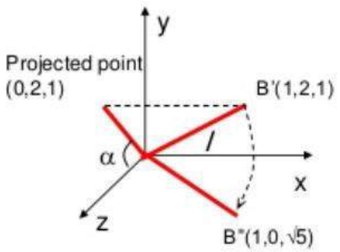
points

Translate point A to the origin



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

Step 2. Rotate axis *A'B'* about the x axis by and angle α, until it lies on the *xz* plane.



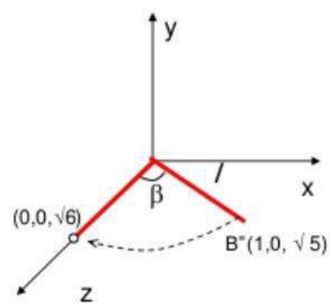
$$\sin\alpha = \frac{2}{\sqrt{2^2+1^2}} = \frac{2}{\sqrt{5}} = \frac{2\sqrt{5}}{5}$$

$$\cos\alpha = \frac{1}{\sqrt{5}} = \frac{\sqrt{5}}{5}$$

$$l = \sqrt{1^2+2^2+1^2} = \sqrt{6}$$

$$\mathbf{R}_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{\sqrt{5}}{5} & -\frac{2\sqrt{5}}{5} & 0 \\ 0 & \frac{2\sqrt{5}}{5} & \frac{\sqrt{5}}{5} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Step 3. Rotate axis *A'B''* about the y axis by and angle ϕ, until it coincides with the z axis.



$$\sin\beta = \frac{1}{\sqrt{6}} = \frac{\sqrt{6}}{6}$$

$$\cos\beta = \frac{\sqrt{5}}{\sqrt{6}} = \frac{\sqrt{30}}{6}$$

$$\mathbf{R}_y(\beta) = \begin{bmatrix} \frac{\sqrt{30}}{6} & 0 & -\frac{\sqrt{6}}{6} & 0 \\ 0 & 1 & 0 & 0 \\ \frac{\sqrt{6}}{6} & 0 & \frac{\sqrt{30}}{6} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Step 4. Rotate the cube 90° about the z axis

$$\mathbf{R}_z(90°) = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

  - Finally, the concatenated rotation matrix about the arbitrary axis AB becomes,

$$\mathbf{R}(\theta) = \mathbf{T}^{-1}\mathbf{R}_x^{-1}(\alpha)\mathbf{R}_y^{-1}(\beta)\mathbf{R}_z(90°)\mathbf{R}_y(\beta)\mathbf{R}_x(\alpha)\mathbf{T}$$

$$\mathbf{R}(\theta) = \begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{\sqrt{5}}{5} & \frac{2\sqrt{5}}{5} & 0 \\ 0 & -\frac{2\sqrt{5}}{5} & \frac{\sqrt{5}}{5} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{\sqrt{30}}{6} & 0 & \frac{\sqrt{6}}{6} & 0 \\ 0 & 1 & 0 & 0 \\ -\frac{\sqrt{6}}{6} & 0 & \frac{\sqrt{30}}{6} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} \frac{\sqrt{30}}{6} & 0 & -\frac{\sqrt{6}}{6} & 0 \\ 0 & 1 & 0 & 0 \\ \frac{\sqrt{6}}{6} & 0 & \frac{\sqrt{30}}{6} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{\sqrt{5}}{5} & -\frac{2\sqrt{5}}{5} & 0 \\ 0 & \frac{2\sqrt{5}}{5} & \frac{\sqrt{5}}{5} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -2 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 0.166 & -0.075 & 0.983 & 1.742 \\ 0.742 & 0.667 & 0.075 & -1.151 \\ -0.650 & 0.741 & 0.167 & 0.560 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Multiplying $\mathbf{R}(\theta)$ by the point matrix of the original cube

$$[\mathbf{P}'] = \mathbf{R}(\theta) \cdot [\mathbf{P}]$$

$$[\mathbf{P}'] = \begin{bmatrix} 0.166 & -0.075 & 0.983 & 1.742 \\ 0.742 & 0.667 & 0.075 & -1.151 \\ -0.650 & 0.741 & 0.167 & 0.560 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 2.650 & 1.667 & 1.834 & 2.816 & 2.725 & 1.742 & 1.909 & 2.891 \\ -0.558 & -0.484 & 0.258 & 0.184 & -1.225 & -1.151 & -0.409 & -0.483 \\ 1.467 & 1.301 & 0.650 & 0.817 & 0.726 & 0.560 & -0.091 & 0.076 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

More 3D transformations

+/- same as in 2D:

- Matrix concatenation by multiplication
- Reflection
- Shearing
- Transformations between coordinate systems

3D scaling

Scale with factors s_x, s_y, s_z:

$x' = s_x x$, $y' = s_y y$, $z' = s_z z$

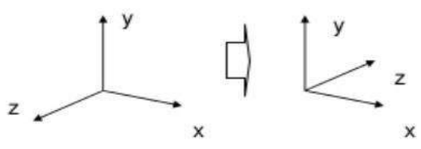
or

$\mathbf{P}' = \mathbf{S}\mathbf{P}$, or

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

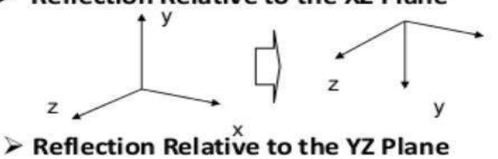
3D Reflection

- Reflection Relative to the XY Plane



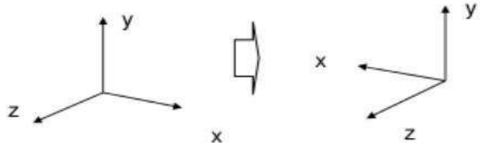
$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- Reflection Relative to the XZ Plane



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

- Reflection Relative to the YZ Plane

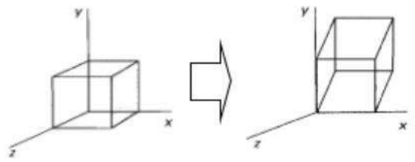


$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Shear

- **Z-axis 3-D Shear transformation**

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



- The effect of this transformation matrix is to alter the x and y co-ordinate values by an amount that is proportional to the z-value, while leaving z co-ordinate unchanged. Boundaries of the plane that are perpendicular to z-axis are thus shifted proportional to z-value.

Affine transformations 1

Generic name for these transformations:
affine transformations

$$x' = a_{xx}x + a_{xy}y + a_{xz}z + b_x$$

$$y' = a_{yx}x + a_{yy}y + a_{yz}z + b_y$$

$$z' = a_{zx}x + a_{zy}y + a_{zz}z + b_z$$

Shear

X-axis 3-D Shear transformation

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

Y-axis 3-D Shear transformation

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

Affine transformations

Properties:

1. Transformed coordinates *x',y'* and *z'* are *linearly* dependent on original coordinates *x, y* and *z.*
2. Parameters a_{ij} and b_k are constant and determine type of transformation;
3. Examples: translation, rotation, scaling, reflection
4. Parallel lines remain parallel
5. Only translation, rotation reflection: angles and lengths are maintained

Concatenation

- Combining two or more transformation is known as Concatenation.
- The combined transformation formed is known as Composite transformation.
- The order of performing the transformation is important.

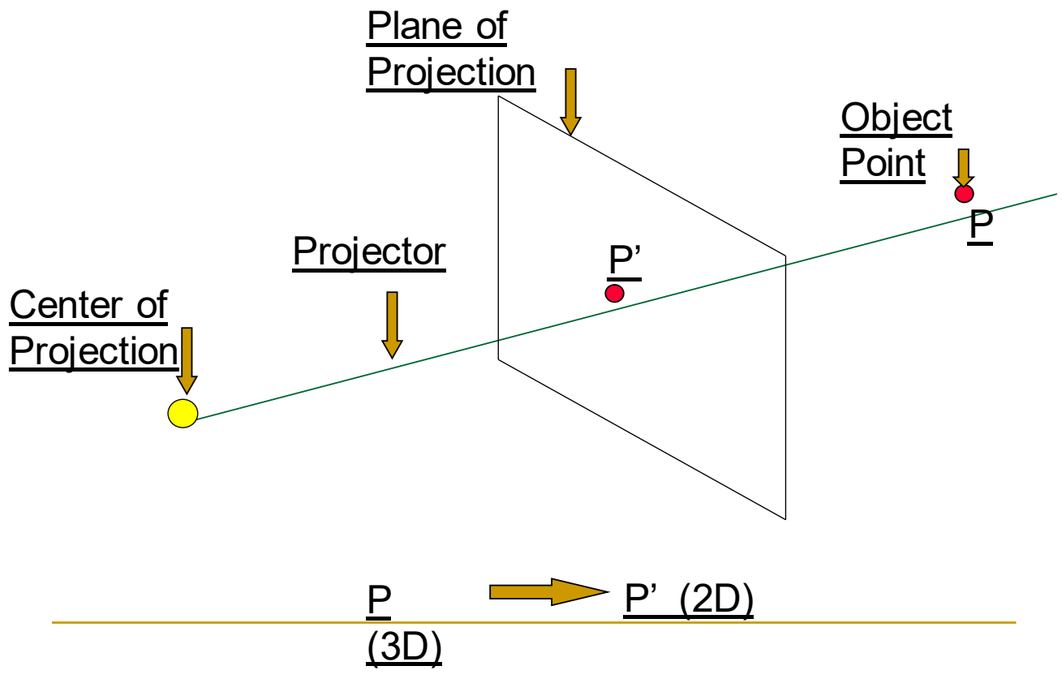
3D Elementary Graphics

Anupama Namburu

Module 3

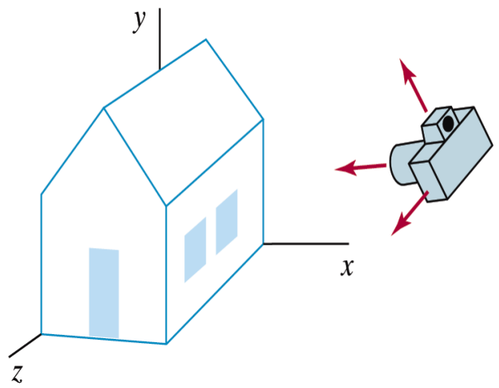
Module No 3	Elementary 3D Graphics	8 hours
Plane projections, Vanishing points, Specification of a 3D view. Camera Models; Viewing classical three-dimensional viewing, computer viewing, specifying views, parallel and perspective projective transformations		

Projection Geometry



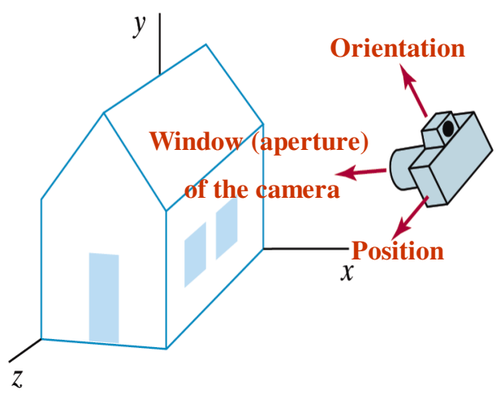
3D Viewing

- The steps for computer generation of a view of a three dimensional scene are somewhat analogous to the processes involved in taking a photograph.



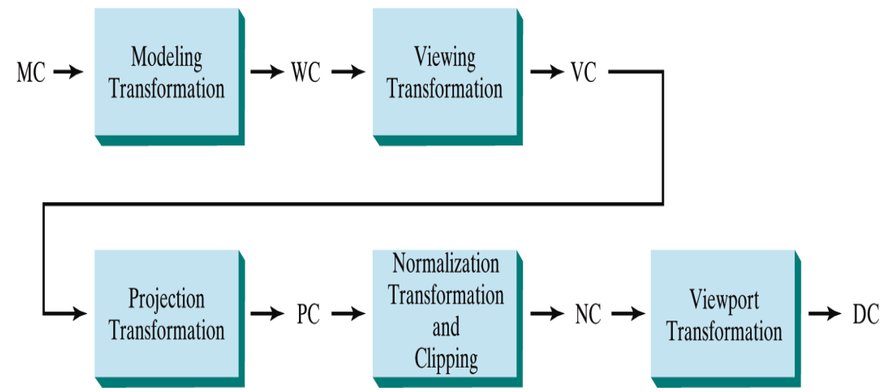
Camera Analogy

1. Viewing position
2. Camera orientation
3. Size of clipping window



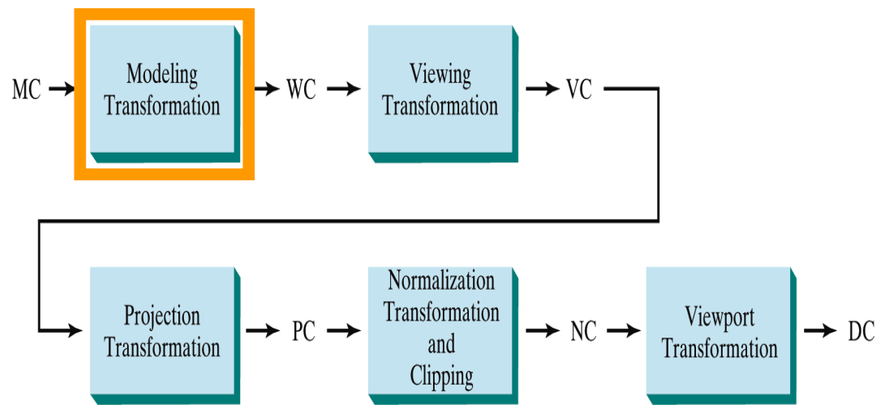
Viewing Pipeline

- The general processing steps for modeling and converting a world coordinate description of a scene to device coordinates:



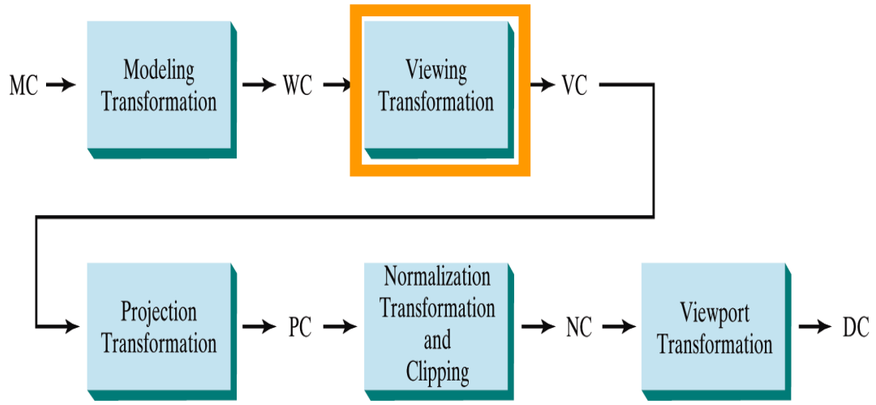
Viewing Pipeline

1. Construct the shape of individual objects in a scene within modeling coordinate, and place the objects into appropriate positions within the scene (world coordinate).



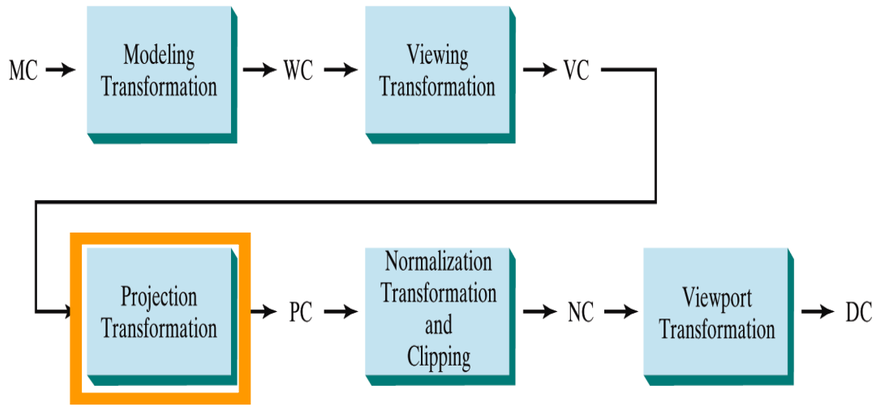
Viewing Pipeline

2. World coordinate positions are converted to viewing coordinates.



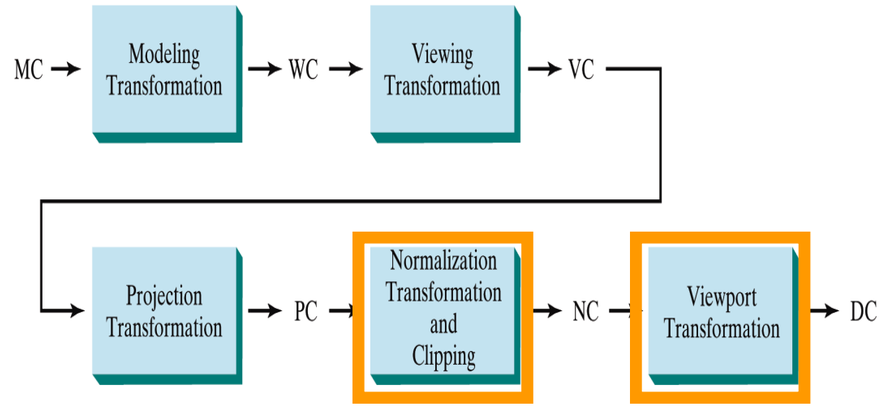
Viewing Pipeline

3. Convert the viewing coordinate description of the scene to coordinate positions on the projection plane.

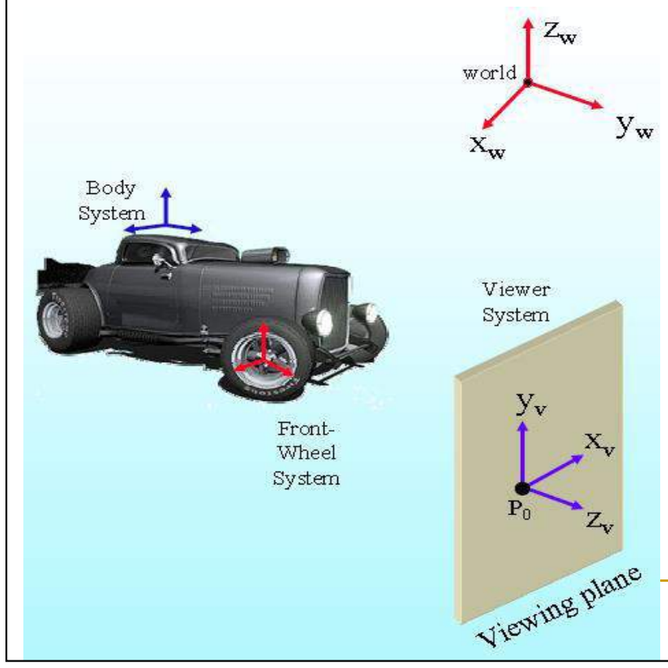


Viewing Pipeline

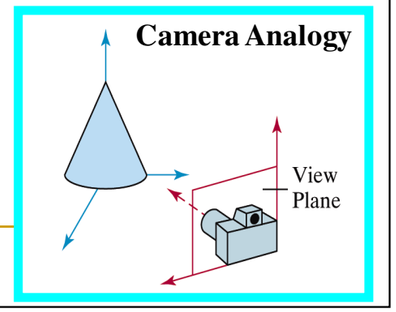
4. Positions on the projection plane, will then mapped to the Normalized coordinate and output device.



Viewing Coordinates

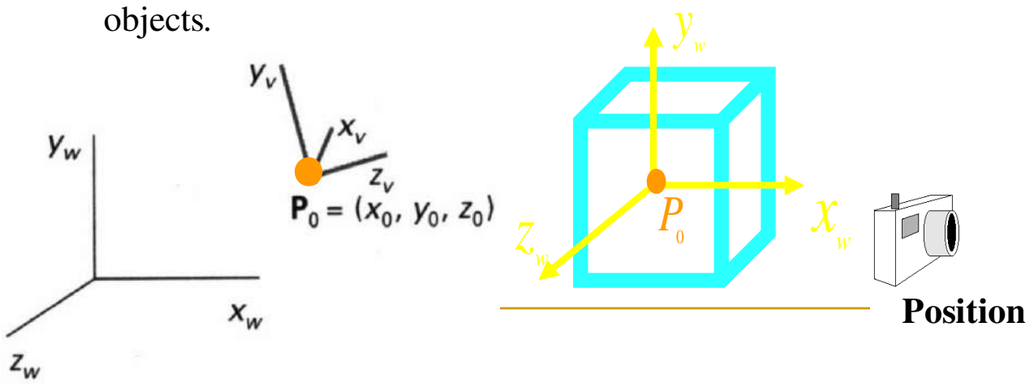


- Viewing coordinates system described 3D objects with respect to a viewer.
- A Viewing (Projector) plane is set up perpendicular to z_v and aligned with (x_v,y_v).



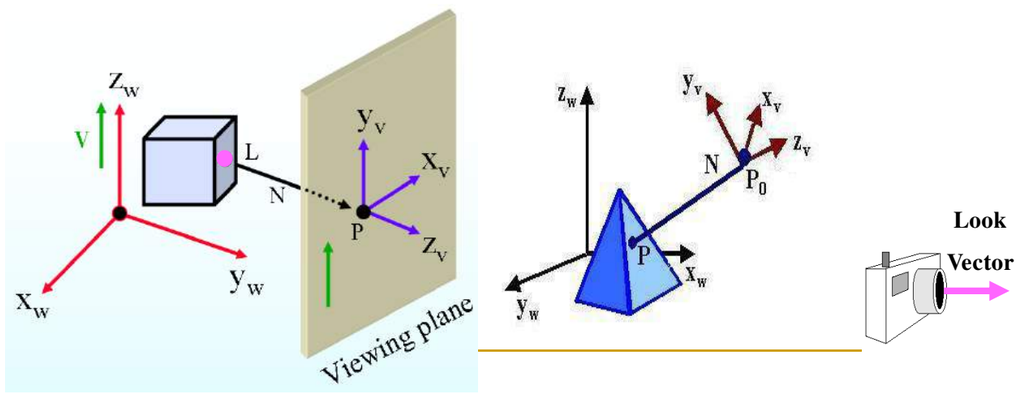
Specifying the Viewing Coordinate System (View Reference Point)

- We first pick a world coordinate position called **view reference point** (origin of our viewing coordinate system).
- P_0 is a point where a camera is located.
- The view reference point is often chosen to be close to or on the surface of some object, or at the center of a group of objects.



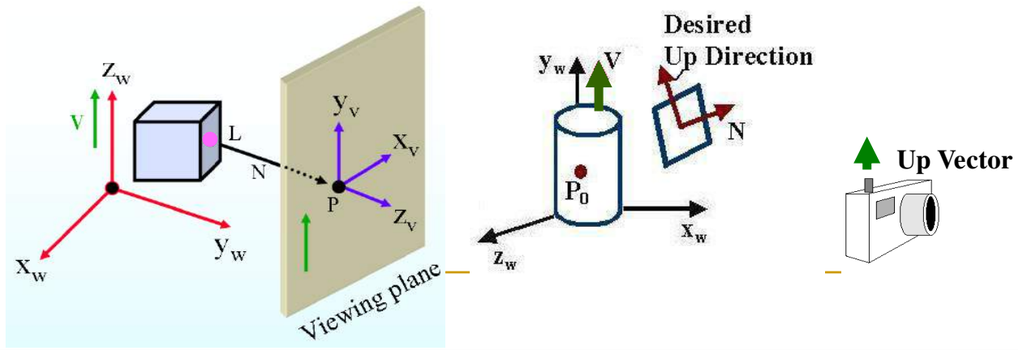
Specifying the Viewing Coordinate System (Z_v Axis)

- Next, we select the positive direction for the viewing z_v axis, by specifying the **view plane normal vector**, **N**.
- The direction of **N**, is from the **look at point** (L) to the view reference point.



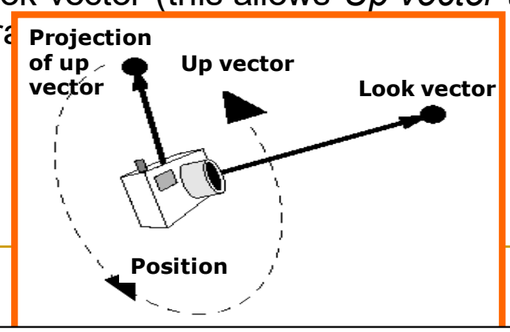
Specifying the Viewing Coordinate System (y_v Axis)

- Finally, we choose the ***up direction*** for the view by specifying a vector ***V***, called the ***view up vector***.
- This vector is used to establish the positive direction for the y_v axis.
- ***V*** is projected into a plane that is perpendicular to the normal vector.



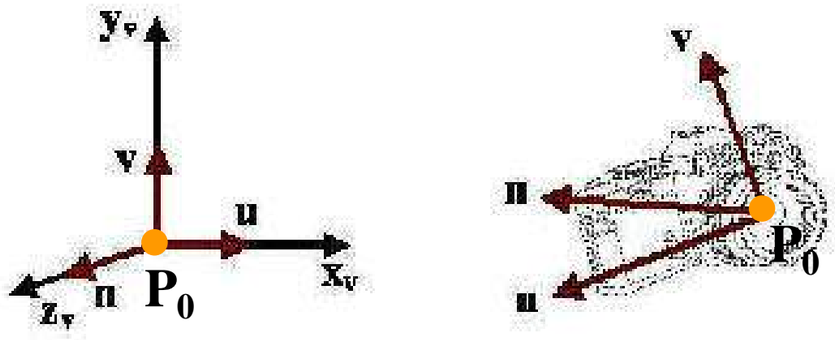
Look and Up Vectors

- ***Look Vector***
  - the direction the camera is pointing
  - three degrees of freedom; can be any vector in 3-space
- ***Up Vector***
  - determines how the camera is rotated around the *Look vector*
  - for example, whether you're holding the camera horizontally or vertically (or in between)
  - projection of *Up vector* must be in the plane perpendicular to the look vector (this allows *Up vector* to be specified at an arbitra



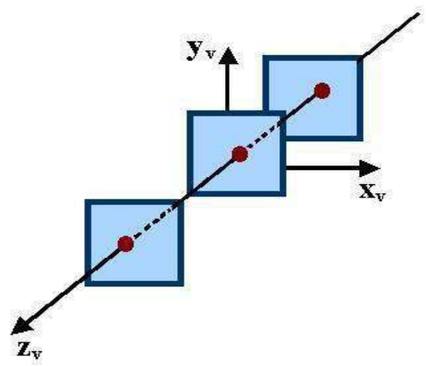
Specifying the Viewing Coordinate System (x_v Axis)

- Using vectors **N** and **V**, the graphics package computer can compute a third vector **U**, perpendicular to both **N** and **V**, to define the direction for the $\mathbf{x_v}$ axis.



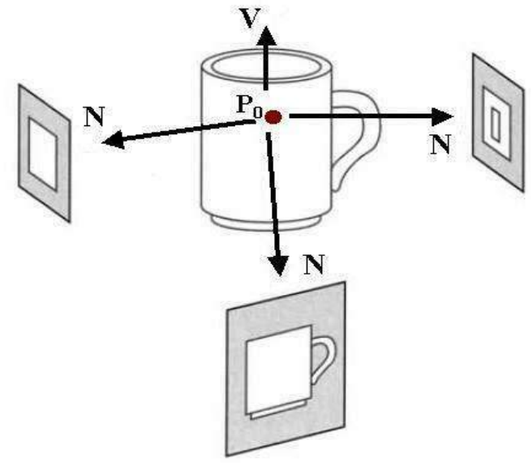
The View Plane

- Graphics package allow users to choose the position of the view plane along the z_v axis by specifying the **view plane distance** from the viewing origin.
- The view plane is always parallel to the $x_v y_v$ plane.



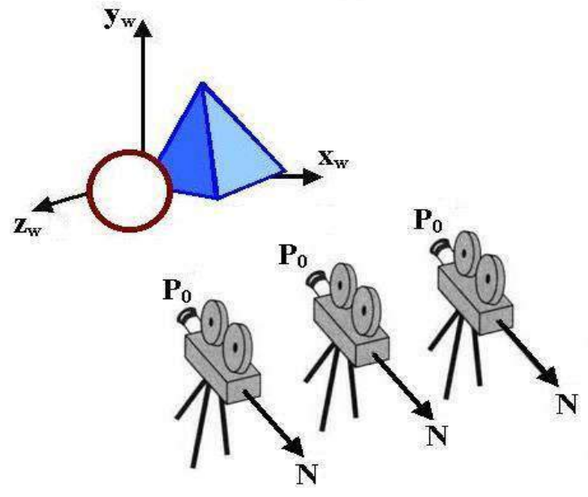
Obtain a Series of View

- To obtain a series of view of a scene, we can keep the view reference point fixed and **change** the direction of **N**.



Simulate Camera Motion

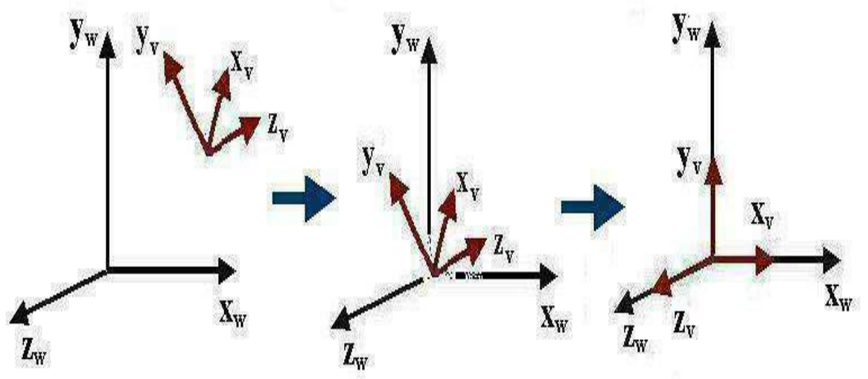
- To simulate camera motion through a scene, we can keep **N fixed** and **move** the view reference **point** around.



Transformation from World to Viewing Coordinates

Transformation from World to Viewing Coordinates

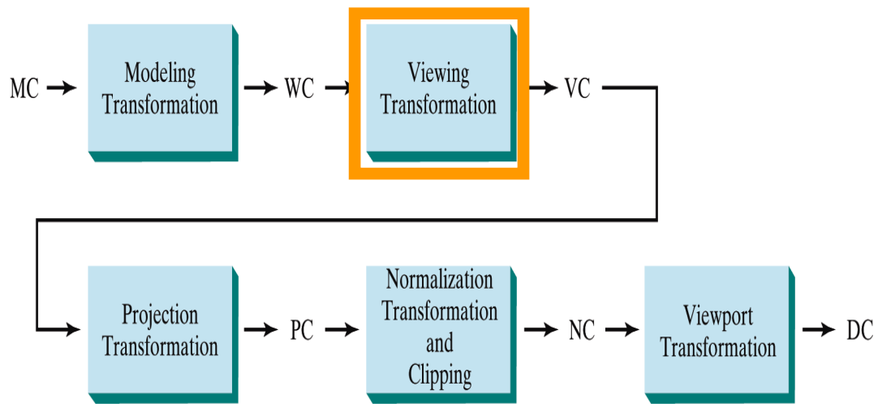
- Transformation sequence from world to viewing coordinates:



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

Viewing Pipeline

- Before object description can be projected to the view plane, they must be transferred to viewing coordinates.
- World coordinate positions are converted to viewing coordinates.



Transformation from World to Viewing Coordinates

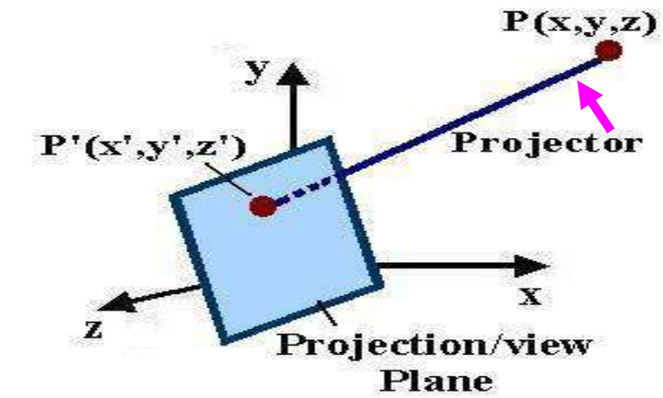
- Another Method for generating the rotation-transformation matrix is to calculate unit **uvn** vectors and form the composite rotation matrix directly:

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

Projection

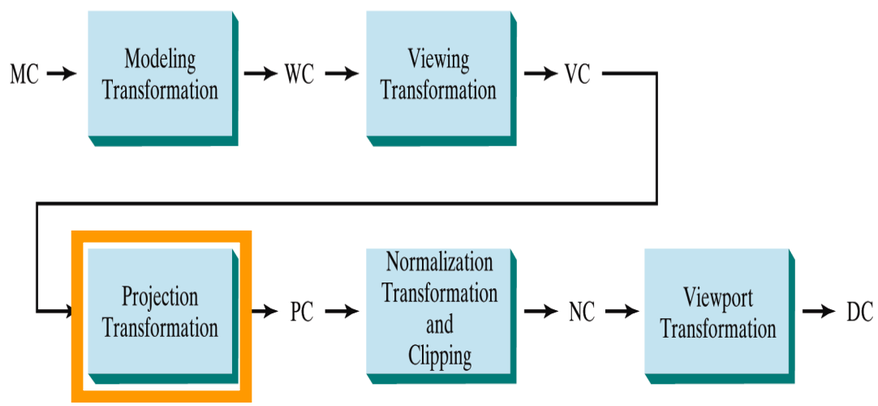
Projection

- **Projection** can be defined as a mapping of point P(x,y,z) onto its image in the projection plane.
- The mapping is determined by a ***projector*** that passes through P and intersects the view plane ( ).



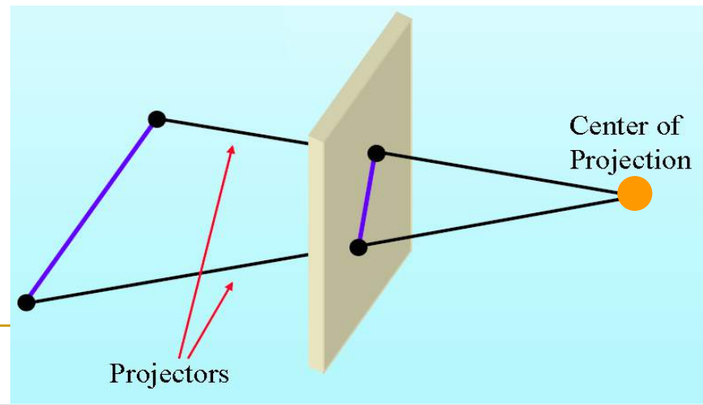
Viewing Pipeline

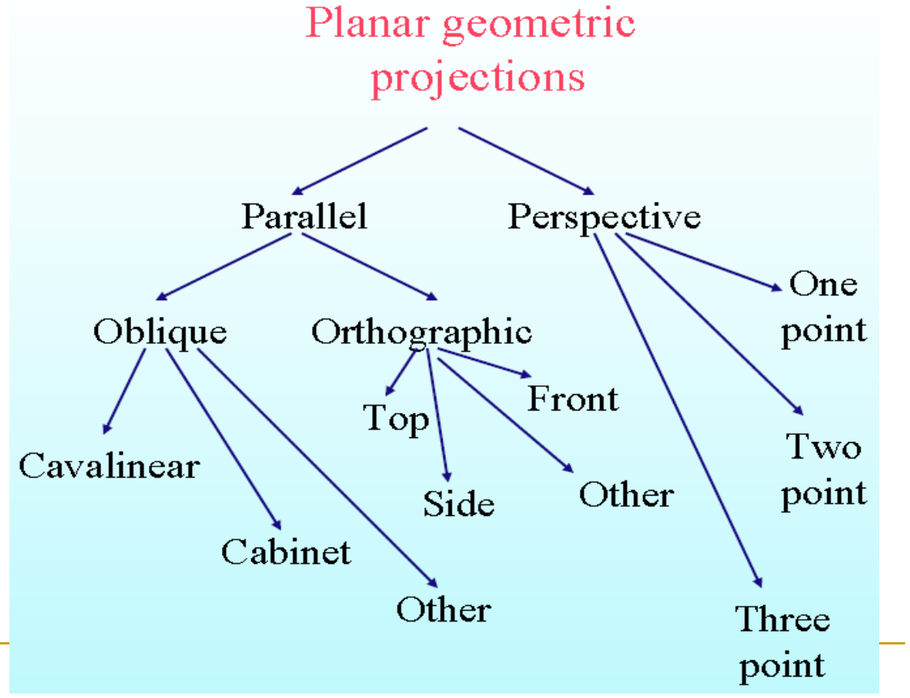
- Convert the viewing coordinate description of the scene to coordinate positions on the projection plane.
- Viewing 3D objects on a 2D display requires a mapping from 3D to 2D.



Projection

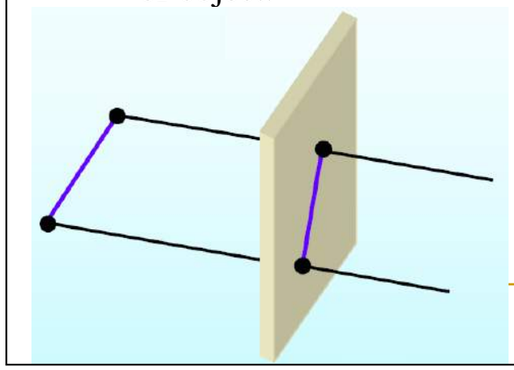
- Projectors are lines from **center (reference) of projection** through each point in the object.
- The result of projecting an object is dependent on the spatial relationship among the projectors and the view plane.

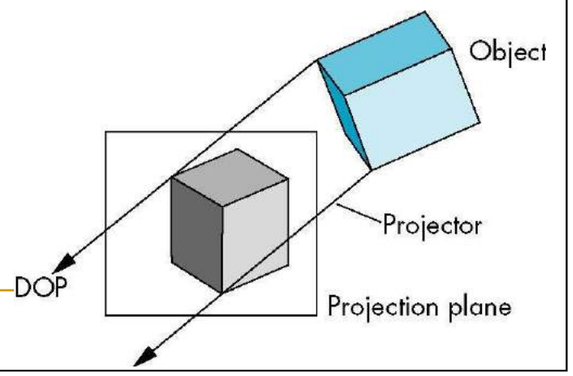




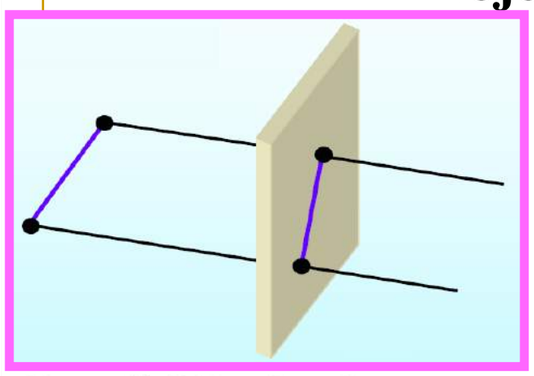
Parallel Projection

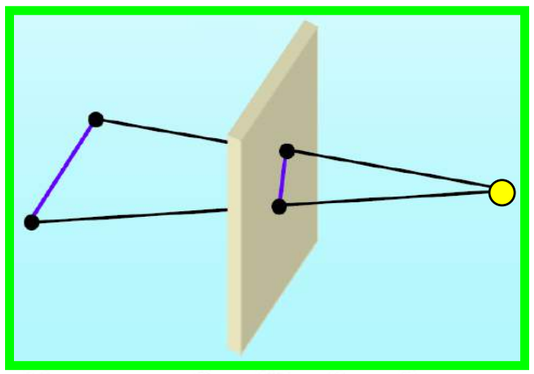
- Coordinate position are transformed to the view plane along parallel lines.
- Center of projection at infinity results with a parallel projection.
- A parallel projection preserves relative proportion of objects, but dose not give us a realistic representation of the appearance of object.





Projection



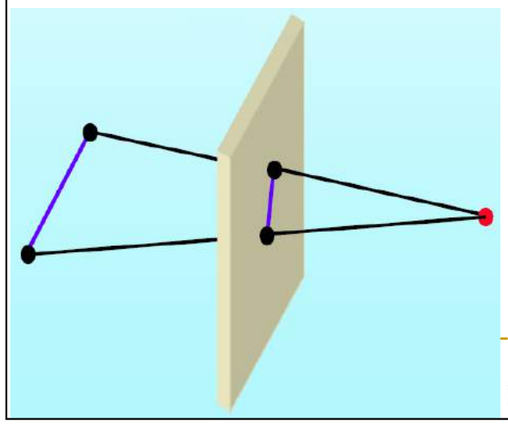


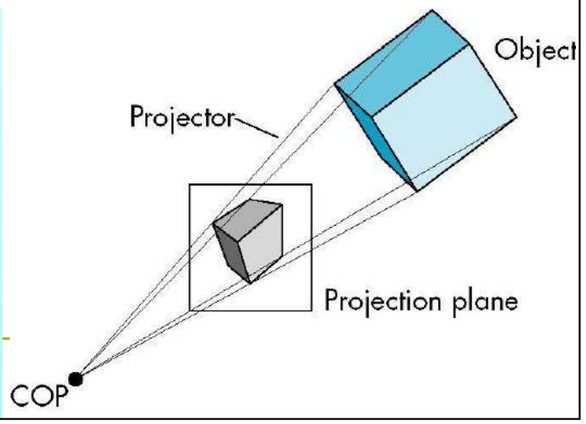
Parallel Projection :
Coordinate position are transformed to the view plane along **parallel lines**.

Perspective Projection:
Object positions are transformed to the view plane along lines that converge to the **projection reference (center) point**.

Perspective Projection

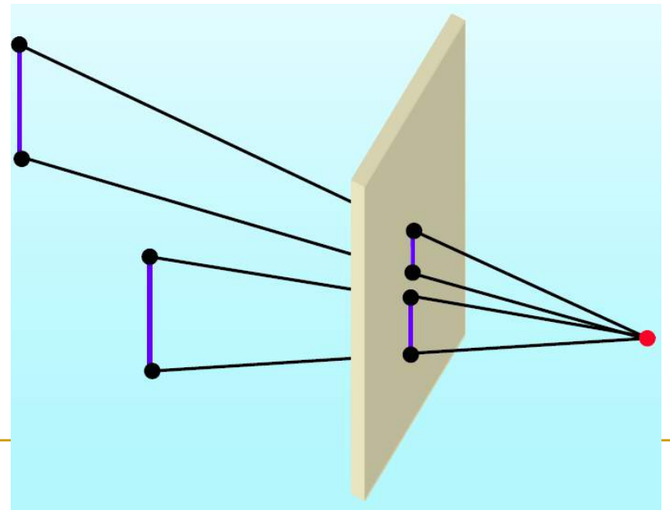
- Object positions are transformed to the view plane along lines that converge to the **projection reference (center) point.**
- Produces realistic views but does not preserve relative proportion of objects.





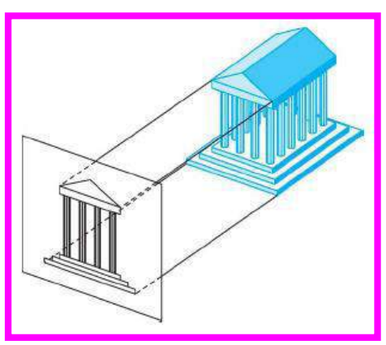
Perspective Projection

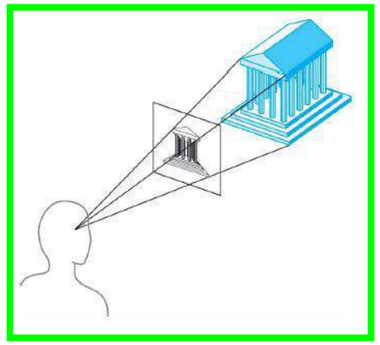
- Projections of distant objects are smaller than the projections of objects of the same size are closer to the projection plane.

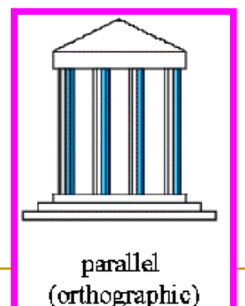


Parallel Projection

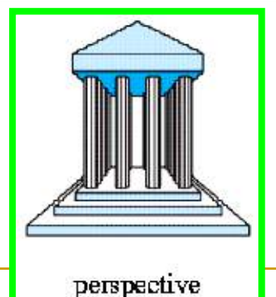
Parallel and Perspective Projection







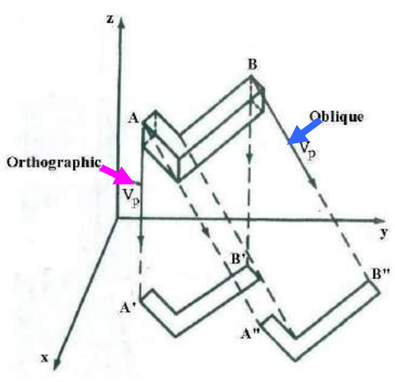
parallel
(orthographic)

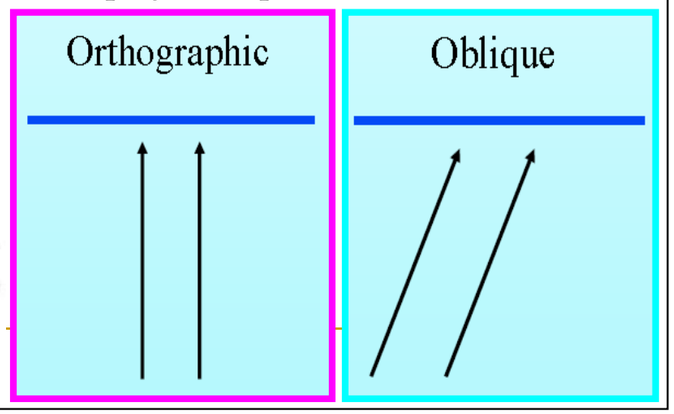


perspective

Parallel Projection

- **Projection vector**: Defines the direction for the projection lines (projectors).
- ***Orthographic Projection***: Projectors (projection vectors) are **perpendicular** to the projection plane.
- ***Oblique Projection***: Projectors (projection vectors) are ***not*** perpendicular to the projection plane.

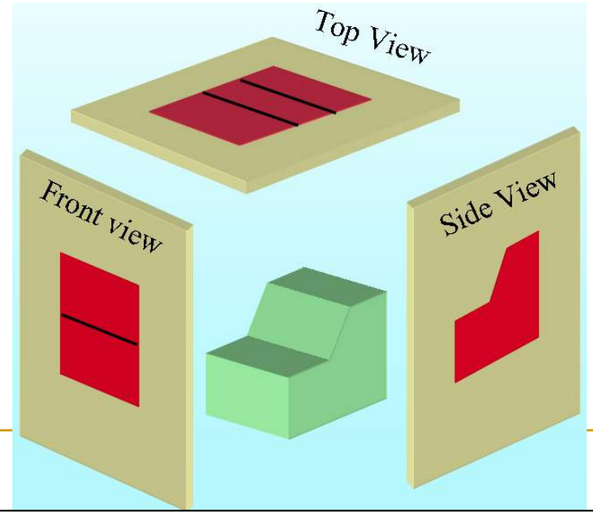




Orthographic Parallel Projection

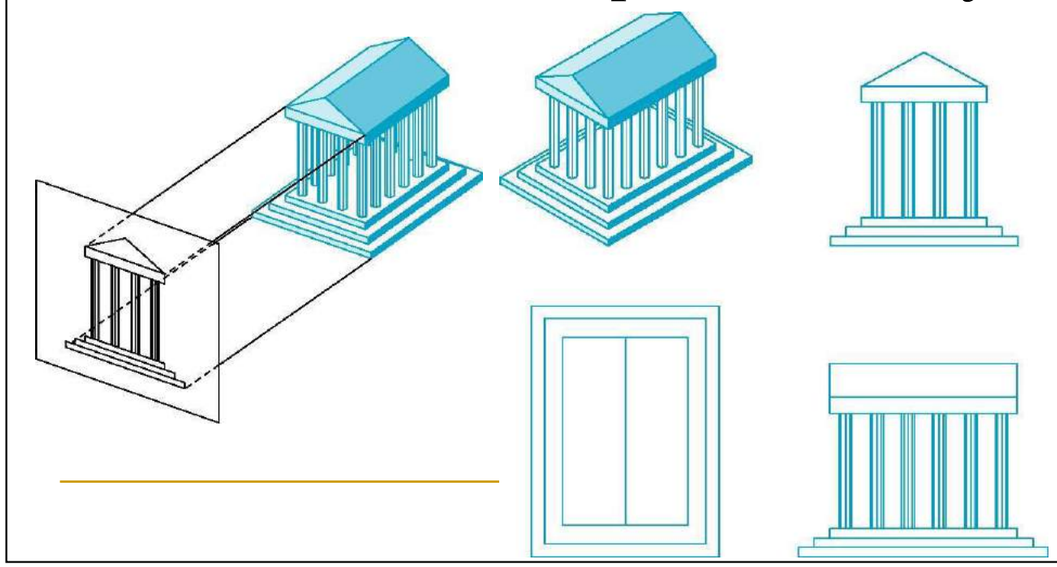
Orthographic Parallel Projection

- ***Front***, ***side***, and ***rear*** orthographic projections of an object are called ***elevations***.
- ***Top*** orthographic projection is called a ***plan*** view.

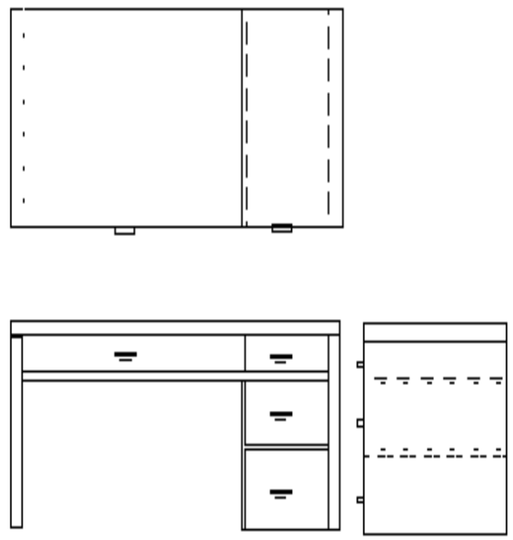


Orthographic Parallel Projection

- Orthographic projection used to produce the **front**, **side**, and **top** views of an object.



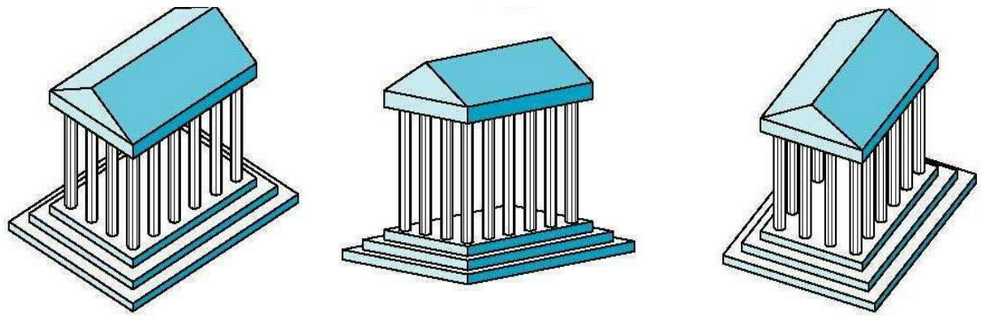
Orthographic Parallel Projection



Multi View Orthographic

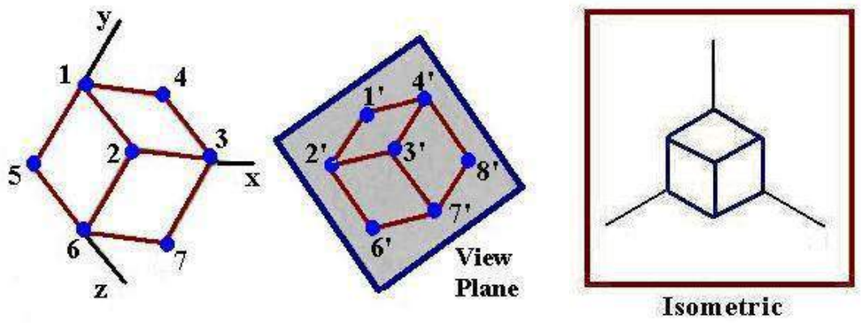
Orthographic Parallel Projection

- ***Axonometric orthographic*** projections display more than one face of an object.



Orthographic Parallel Projection

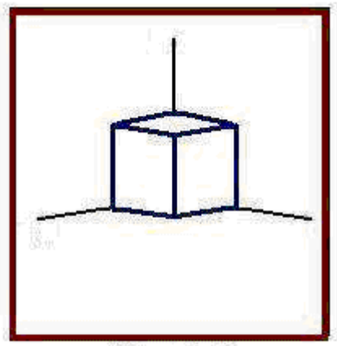
- ***Isometric Projection***: Projection plane intersects each coordinate axis in which the object is defined (principal axes) at the same distant from the origin.
- Projection vector makes equal angles with all of the **three principal axes**.



Isometric projection is obtained by **aligning** the **projection vector** with the **cube diagonal**.

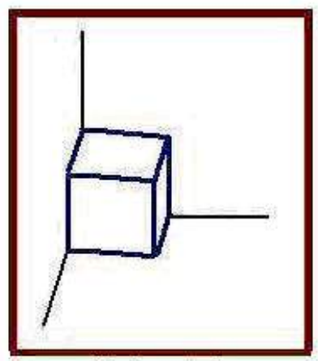
Orthographic Parallel Projection

- ***Dimetric Projection***: Projection vector makes **equal angles** with exactly **two** of the principal axes.

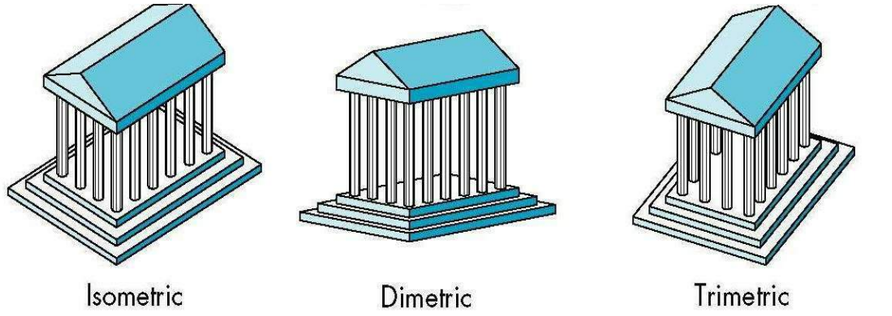


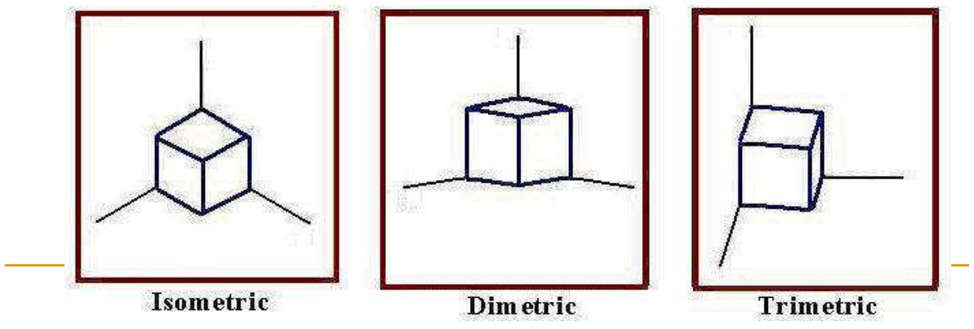
Orthographic Parallel Projection

- ***Trimetric Projection***: Projection vector makes un**equal angles** with the three principal axes.



Orthographic Parallel Projection

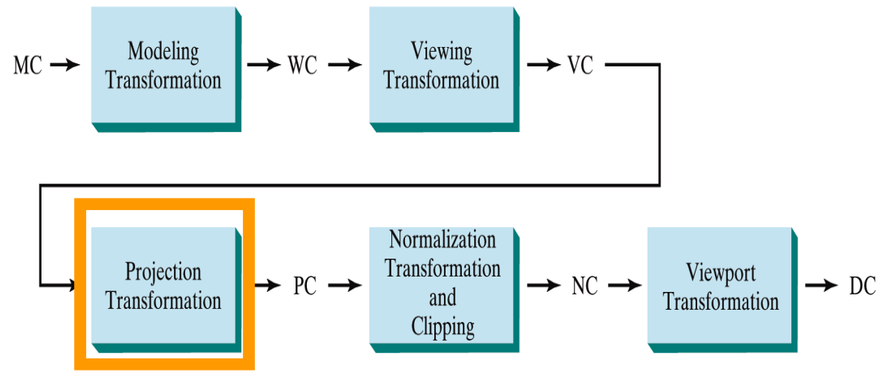




Orthographic Parallel Projection Transformation

Orthographic Parallel Projection Transformation

- Convert the **viewing coordinate** description of the scene to coordinate positions on the **Orthographic parallel projection plane**.

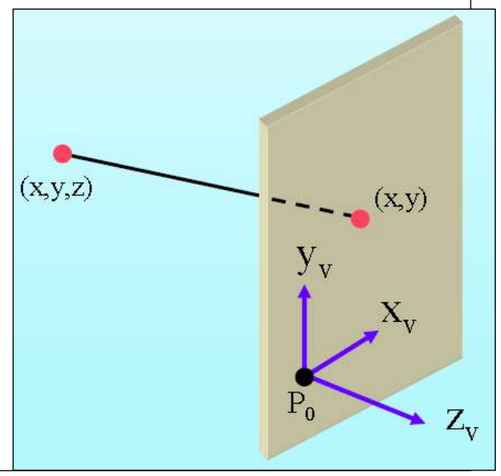


Orthographic Parallel Projection Transformation

- Since the view plane is placed at position z_{vp} along the z_v axis. Then any point (x,y,z) in viewing coordinates is transformed to projection coordinates as:

$$x_p = x, \quad y_p = y$$

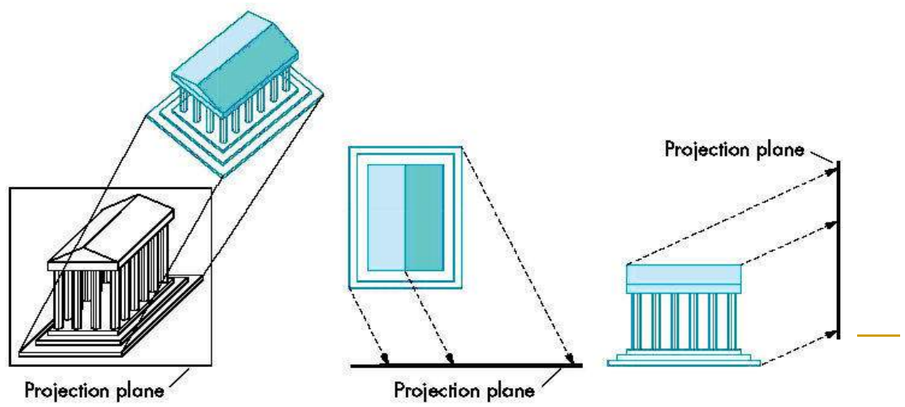
$$\mathbf{M}_{Orthographic\ \ Parallel} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Oblique
Parallel Projection

Oblique Parallel Projection

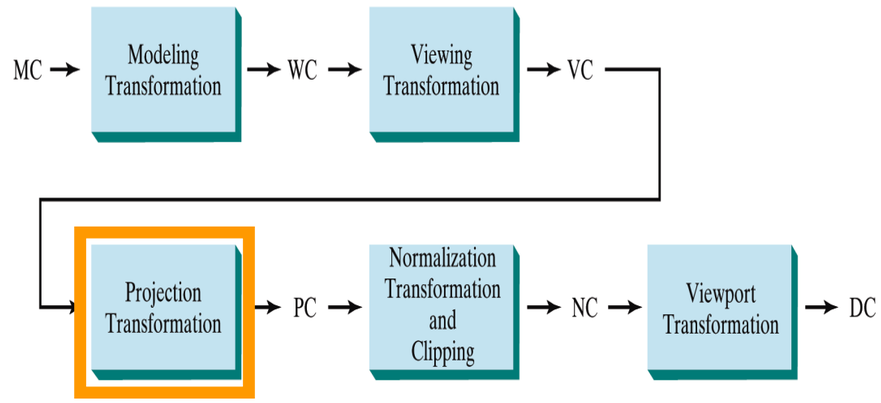
- Projection are **not** perpendicular to the viewing plane.
- Angles and lengths are preserved for faces parallel the plane of projection.
- Preserves 3D nature of an object.



Oblique
Parallel Projection
Transformation

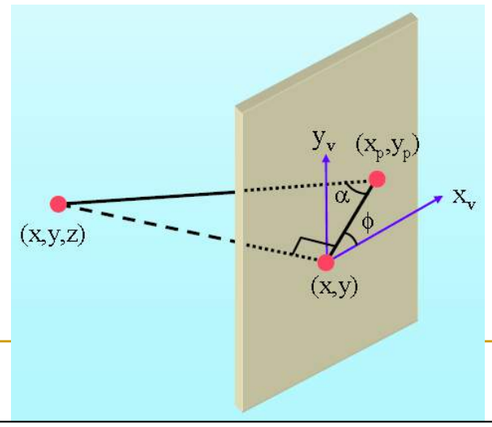
Oblique Parallel Projection Transformation

- Convert the **viewing coordinate** description of the scene to coordinate positions on the **Oblique parallel projection plane**.



Oblique Parallel Projection

- Point (x,y,z) is projected to position (x_p,y_p) on the view plane.
- Projector (oblique) from **(x,y,z)** to (x_p,y_p) makes an angle α with the line (**L**) on the projection plane that joins (x_p,y_p) and **(x,y).**
- Line **L** is at an angle φ with the horizontal direction in the projection plane.



Oblique Parallel Projection

$$x_p = x + L\cos\varphi$$

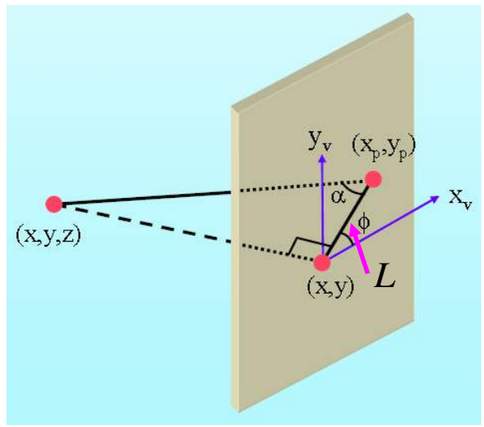
$$y_p = y + L\sin\varphi$$

$$L = \frac{z}{\tan\alpha} = zL_1$$

$$x_p = x + z(L_1\cos\varphi)$$

$$y_p = y + z(L_1\sin\varphi)$$

$$\mathbf{M}_{Parallel} = \begin{bmatrix} 1 & 0 & L_1\cos\varphi & 0 \\ 0 & 1 & L_1\sin\varphi & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Oblique Parallel Projection

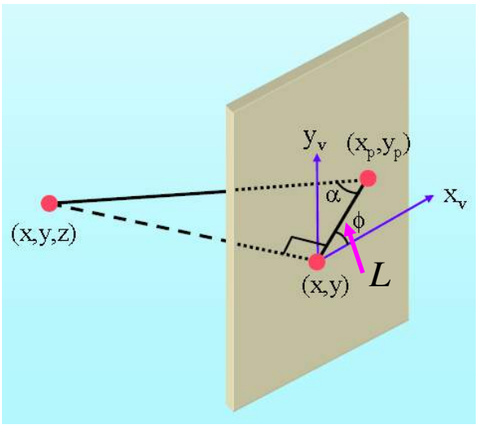
Orthographic Projection:

$$L_1 = 0$$

$$\alpha = 90^\circ$$

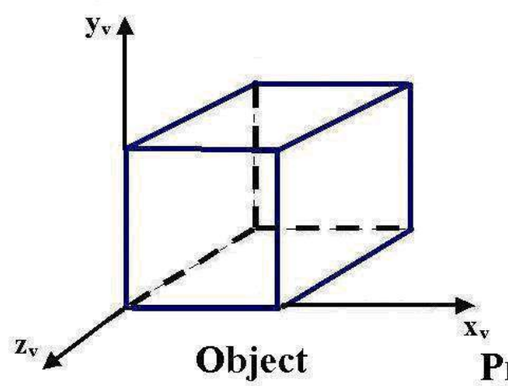
$$x_p = x, \quad y_p = y$$

$$\mathbf{M}_{Orthographic\ \ Parallel} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

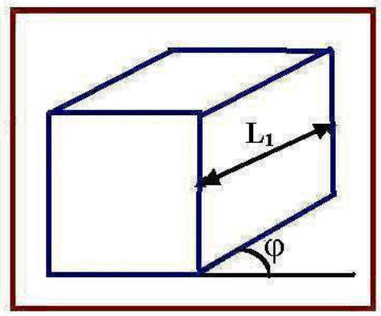


Oblique Parallel Projection

- **Angles**, **distances**, and **parallel lines** in the plane are projected accurately.



Object



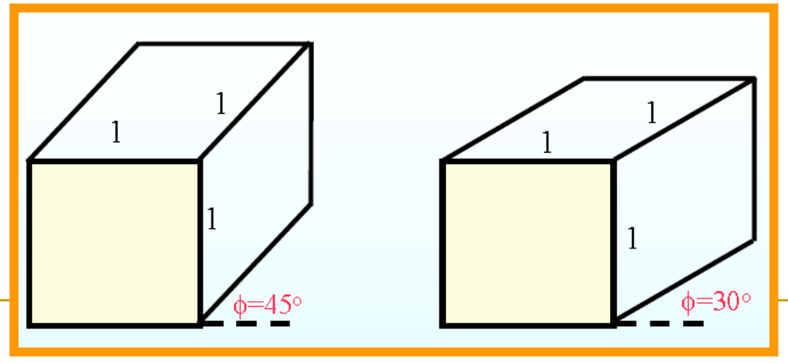
Projection on the Viwing Plane

Cavalier Projection

Cavalier Projection:

$$\tan\alpha = 1$$
$$\alpha = 45^\circ$$

- Preserves lengths of lines perpendicular to the viewing plane.
- 3D nature can be captured but shape seems distorted.
- Can display a combination of front, and side, and top views.

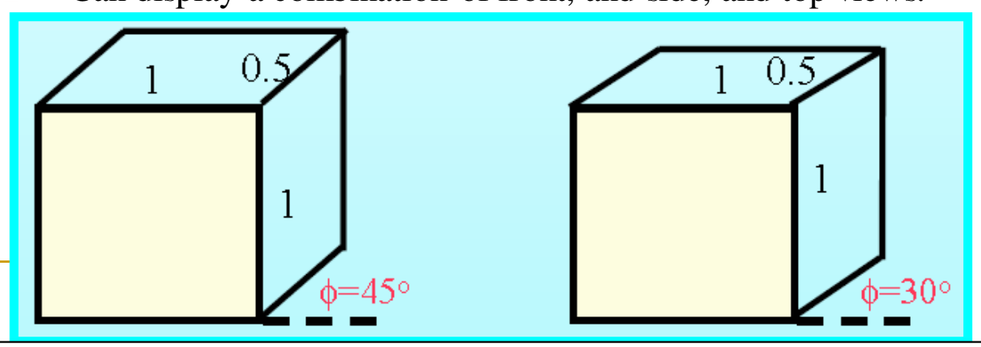


Cabinet Projection

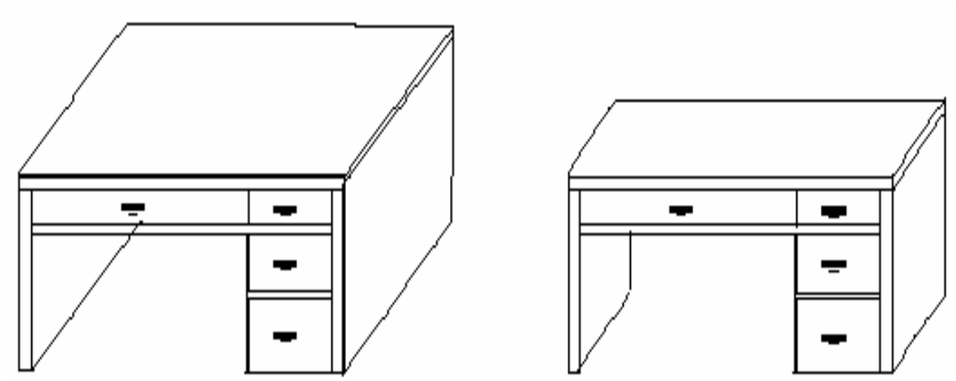
Cabinet Projection:

$$\tan\alpha = 2$$
$$\alpha \approx 63.4^\circ$$

- Lines perpendicular to the viewing plane project at ½ of their **length**.
- A more realistic view than the cavalier projection.
- Can display a combination of front, and side, and top views.



Cavalier & Cabinet Projection

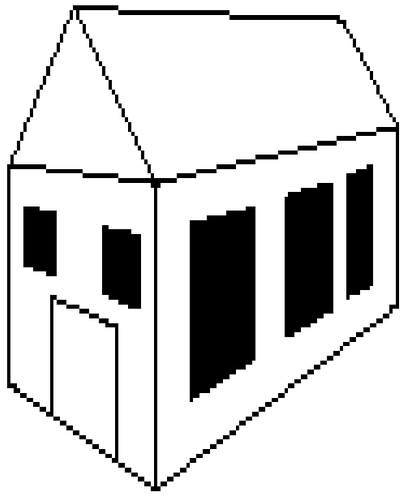


Cavalier

Cabinet

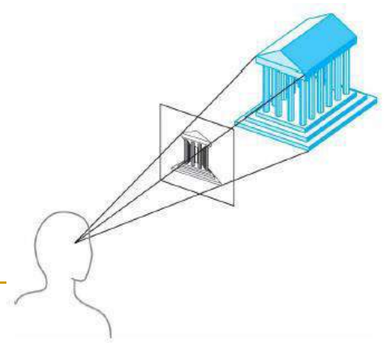
Perspective Projection

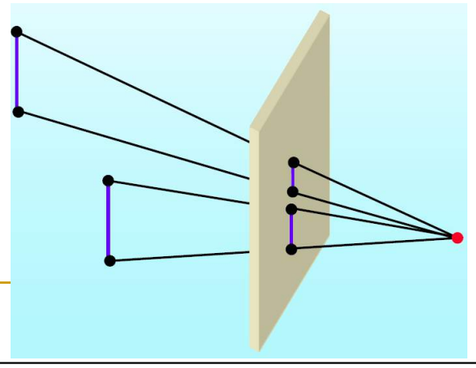
Perspective Projection



Perspective Projection

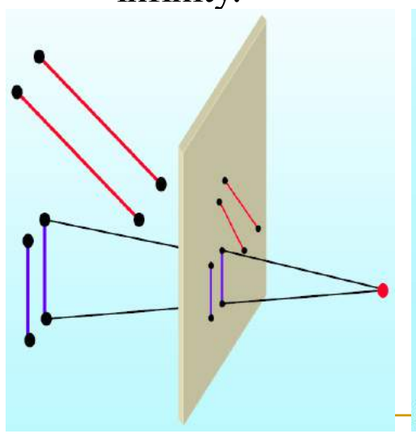
- In a perspective projection, the center of projection is at a finite distance from the viewing plane.
- Produces realistic views but does not preserve relative proportion of objects
- The size of a projection object is inversely proportional to its distance from the viewing plane.

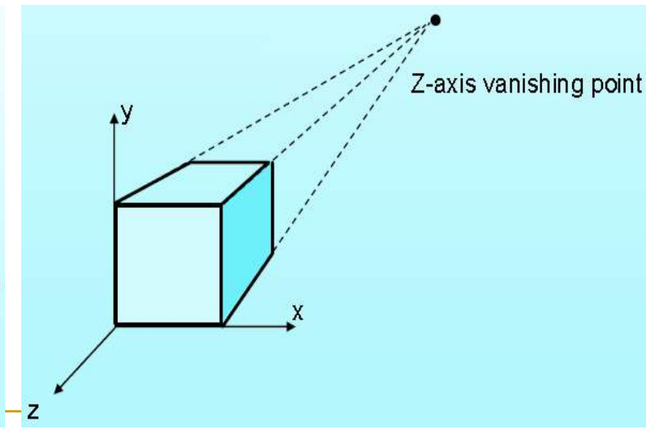




Perspective Projection

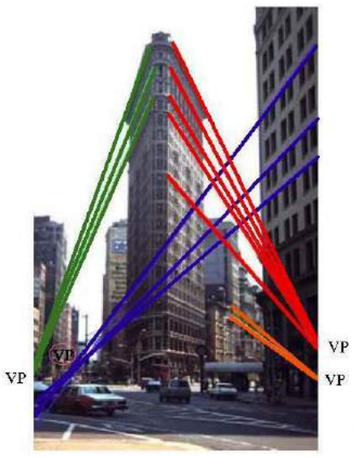
- Parallel lines that are not parallel to the viewing plane, **converge** to a ***vanishing point***.
- A vanishing point is the projection of a point at infinity.

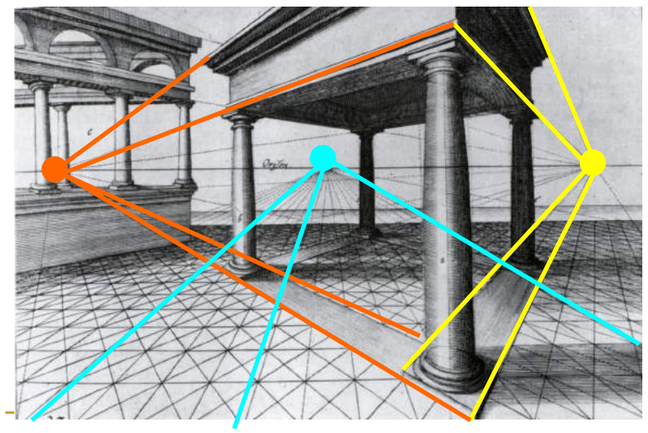




Vanishing Points

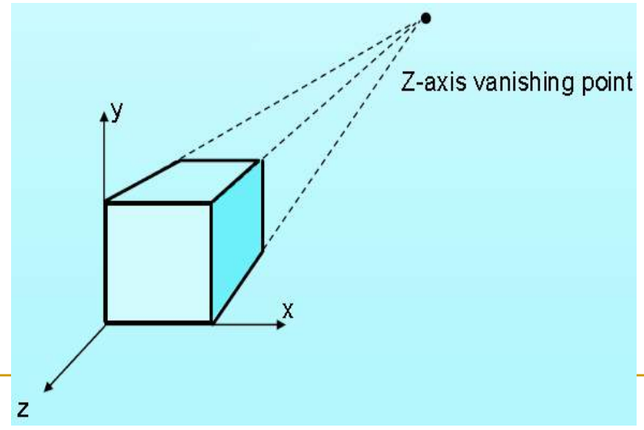
- Each set of projected parallel lines will have a separate vanishing points.
- There are infinity many **general** vanishing points.



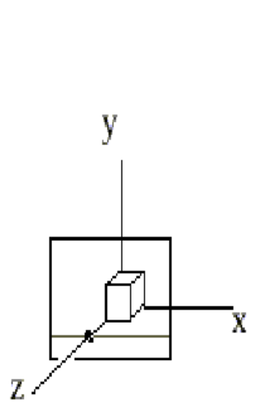


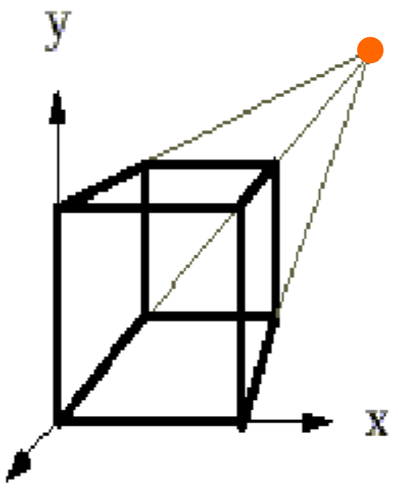
Perspective Projection

- The vanishing point for any set of lines that are parallel to one of the principal axes of an object is referred to as a **principal vanishing point**.
- We control the number of principal vanishing points (one, two, or three) with the orientation of the projection plane.



Perspective Projection





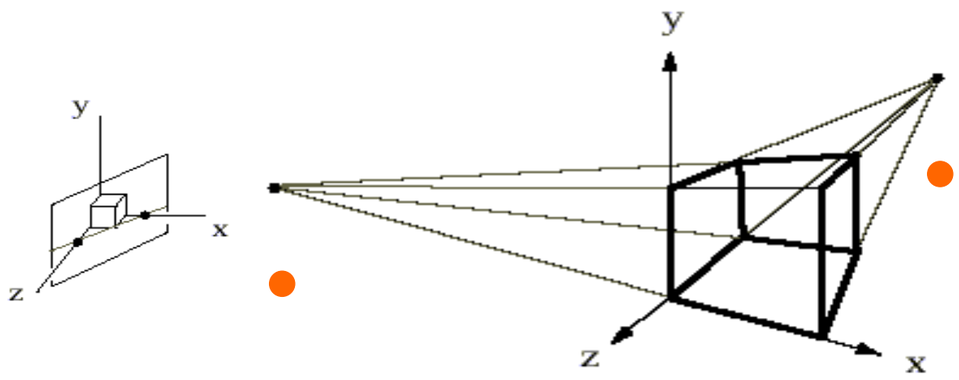
One Point Perspective

(*z*-axis vanishing point)

Perspective Projection

- The number of principal vanishing points in a projection is determined by the number of principal axes **intersecting** the view plane.

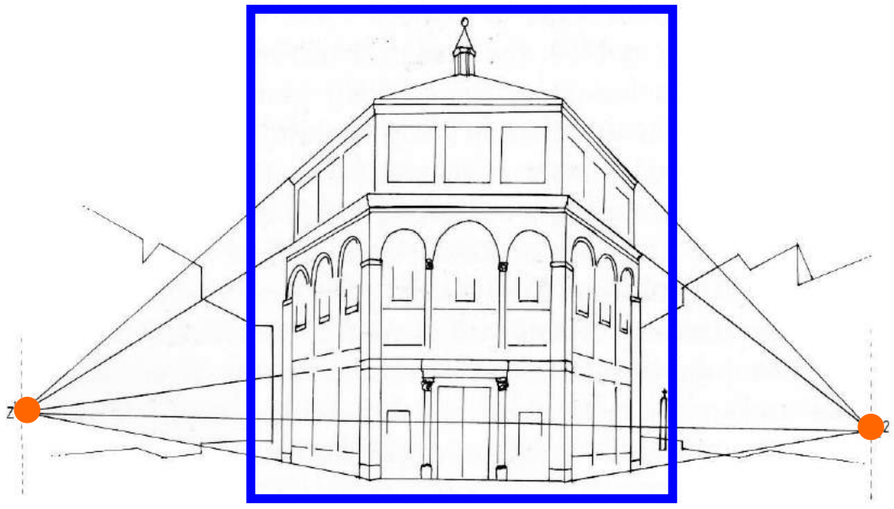
Perspective Projection



Two Point Perspective

(z, and x-axis vanishing points)

Perspective Projection



Two Point Perspective

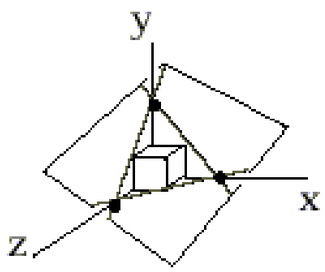
Perspective Projection

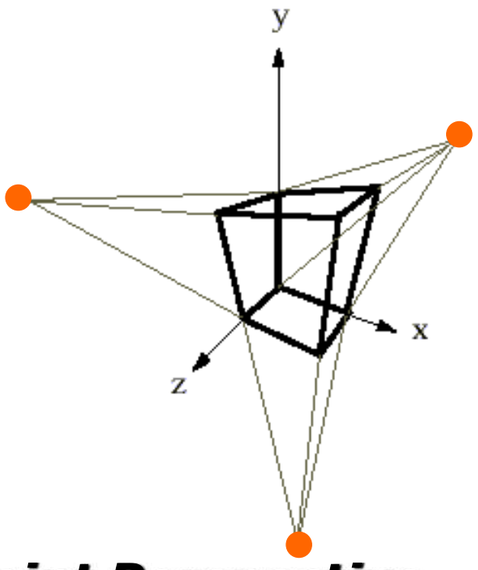
One-Point Perspective Projection

Two-Point Perspective Projection

Tree-Point Perspective Projection

Perspective Projection





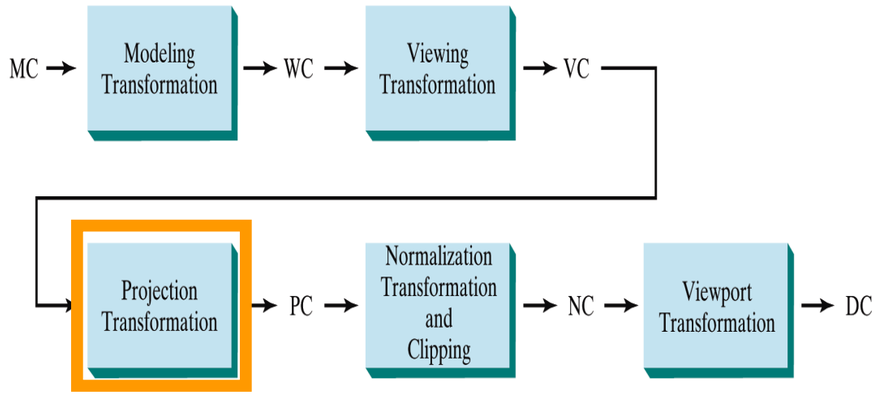
Three Point Perspective

(z, x, and y-axis vanishing points)

Perspective Projection Transformation

Perspective Projection Transformation

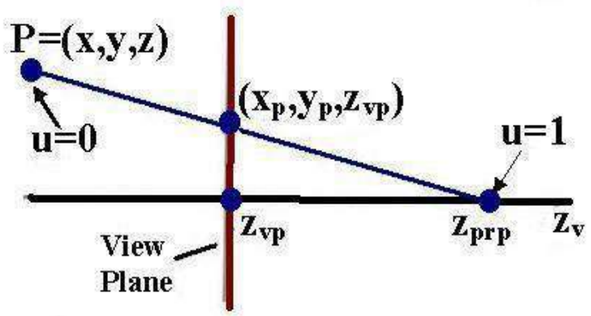
- Convert the **viewing coordinate** description of the scene to coordinate positions on the **perspective projection plane**.



Perspective Projection Transformation

- Suppose the projection reference point at position z_{prp} along the z_v axis, and the view plane at z_{vp}.

$$x' = x - xu$$
$$y' = y - yu$$
$$z' = z - (z - z_{prp})u$$



Perspective Projection Transformation

On the view plane:

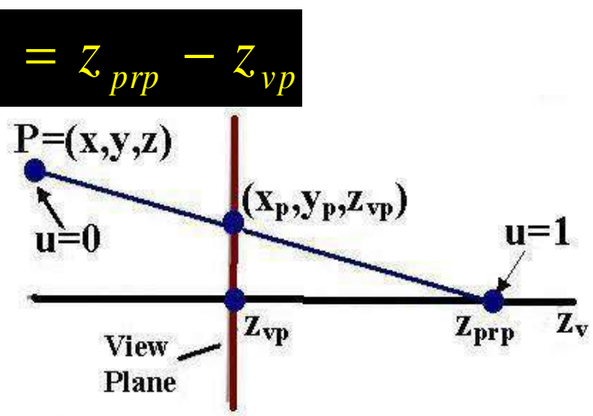
$$u = \frac{z_{vp} - z}{z_{prp} - z}$$

$$z' = z_{vp}$$

$$x' = x - xu$$
$$y' = y - yu$$
$$z' = z - (z - z_{prp})u$$

$$d_p = z_{prp} - z_{vp}$$

$$x_p = x\left(\frac{z_{prp} - z_{vp}}{z - z_{prp}}\right) = x\left(\frac{d_p}{z - z_{prp}}\right)$$
$$y_p = y\left(\frac{z_{prp} - z_{vp}}{z - z_{prp}}\right) = y\left(\frac{d_p}{z - z_{prp}}\right)$$



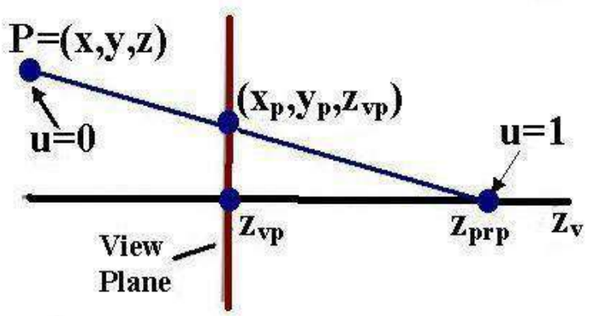
Perspective Projection Transformation

On the view plane: $z' = z_{vp}$

$$\begin{bmatrix} x_h \\ y_h \\ z_h \\ h \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & z_{vp}/d_p & -z_{vp}(z_{prp}/d_p) \\ 0 & 0 & 1/d_p & -z_{prp}/d_p \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$x_p = x_h/h, \quad y_p = y_h/h$$

$$x_p = x\left(\frac{z_{prp} - z_{vp}}{z - z_{prp}}\right) = x\left(\frac{d_p}{z - z_{prp}}\right)$$
$$y_p = y\left(\frac{z_{prp} - z_{vp}}{z - z_{prp}}\right) = y\left(\frac{d_p}{z - z_{prp}}\right)$$



Perspective Projection Transformation

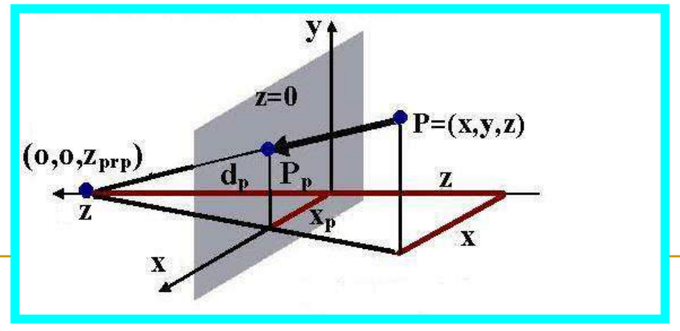
Special Cases: $z_{vp} = 0$

$$x_p = x\left(\frac{z_{prp} - z_{vp}}{z - z_{prp}}\right) = x\left(\frac{d_p}{z - z_{prp}}\right)$$

$$y_p = y\left(\frac{z_{prp} - z_{vp}}{z - z_{prp}}\right) = y\left(\frac{d_p}{z - z_{prp}}\right)$$

$$x_p = x\left(\frac{z_{prp}}{z - z_{prp}}\right) = x\left(\frac{1}{z/z_{prp} - 1}\right)$$

$$y_p = y\left(\frac{z_{prp}}{z - z_{prp}}\right) = y\left(\frac{1}{z/z_{prp} - 1}\right)$$



Perspective Projection Transformation

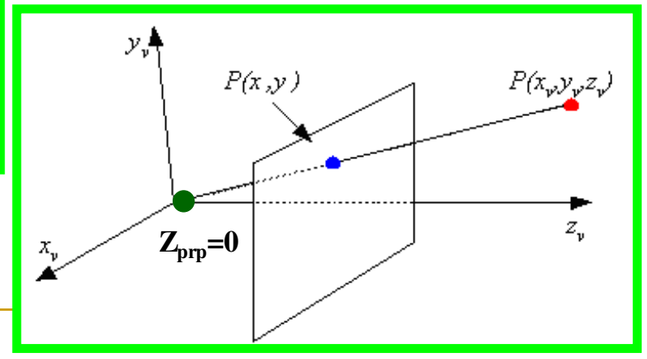
Special Cases: The projection reference point is at the viewing coordinate origin: $z_{prp} = 0$

$$x_p = x\left(\frac{z_{prp} - z_{vp}}{z - z_{prp}}\right) = x\left(\frac{d_p}{z - z_{prp}}\right)$$

$$y_p = y\left(\frac{z_{prp} - z_{vp}}{z - z_{prp}}\right) = y\left(\frac{d_p}{z - z_{prp}}\right)$$

$$x_p = x\left(\frac{-z_{vp}}{z}\right) = x\left(\frac{-1}{z/z_{vp}}\right)$$

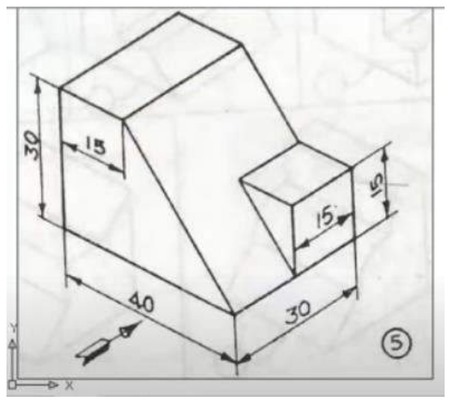
$$y_p = y\left(\frac{-z_{vp}}{z}\right) = y\left(\frac{-1}{z/z_{vp}}\right)$$



Summary

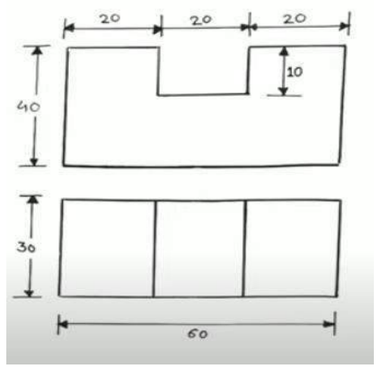
Orthographic projections

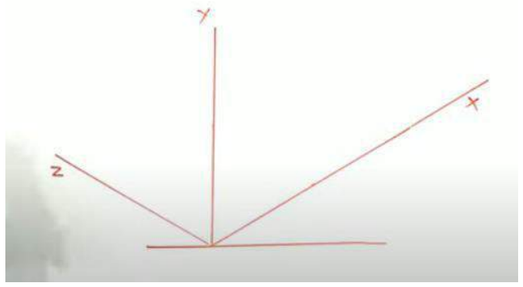
- Obtain the orthographic projection. Obtain the front and top views.



ISOmetric projections

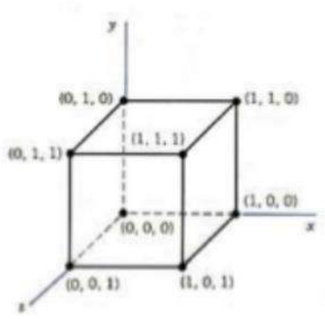
- Obtain the Isometric projections of the given diagram





Oblique Projections

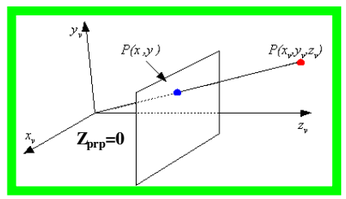
- Given the cube, find the oblique projections with alpha=45 and theta=30 and alpha=90 and theta=45.



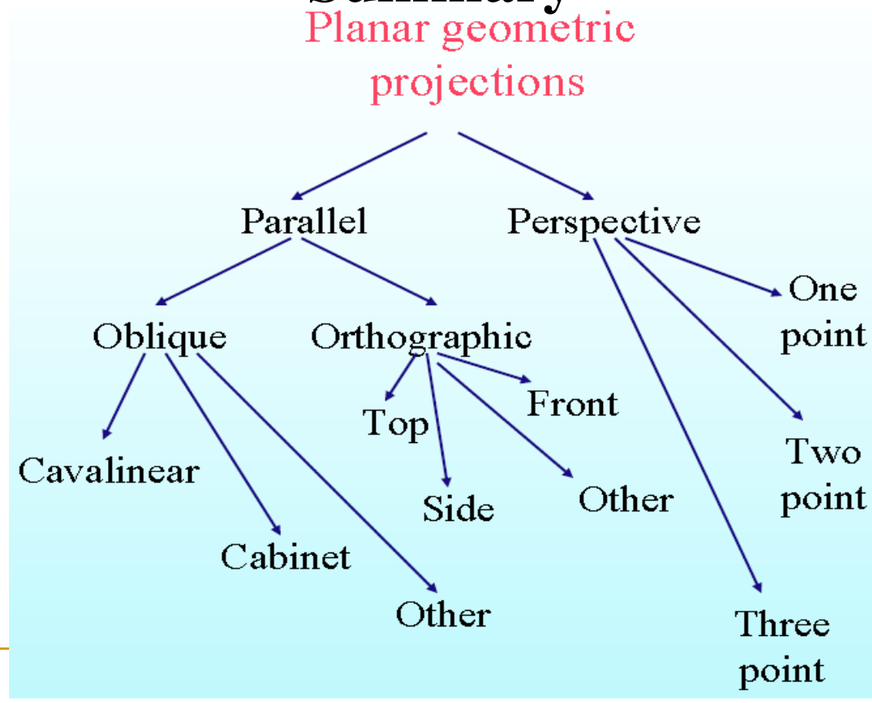
Perspective Projection.

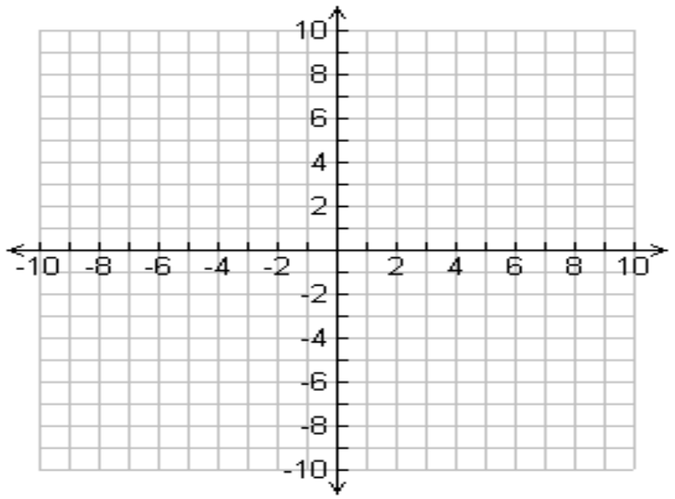
- A single point perspective transformation has to be performed on a triangle (0,1,0),(1,1,0) and (0,1,1) from a center zv=10 on the z-axis, followed by its projection on z=0 plane.

$$\begin{bmatrix} x_h \\ y_h \\ z_h \\ h \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & z_{vp}/d_p & -z_{vp}\left(z_{prp}/d_p\right) \\ 0 & 0 & 1/d_p & -z_{prp}/d_p \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$



Summary





Module 4 - Syllabus

Module No. 4	Visibility	7 hours
Image and object precision; z-buffer algorithms; area based algorithms Binary space partition trees, Open-GL culling, hidden-surface algorithms		

Visibility

Visible surface detection

- Key part in 3D viewing pipeline
- To identify those parts of a scene that are visible from a chosen viewing position.
- Surfaces which are obscured by other opaque surfaces along the line of sight (projection) are invisible to the viewer.
- To define visible surface detections the following are constrains:
- Characteristics of approaches:
  - memory size?
  - processing time?
  - Applicable to which types of objects?
- Considerations:

  - Complexity of the scene

  – Type of objects in the scene

  - Available equipment - Static or animated?

Classification of Visible surface detection

- **Object-Space method**
  - Implemented in physical/local coordinate system
- **Image-space method**
  - Implemented in screen coordinate system

Object Space Methods

- Compare each object with all other objects to determine the visibility of the object parts.
- If there are n objects in the scene, complexity = O(n2)
- Calculations are performed at the resolution in which the objects are defined.(object space/vector space)
- Process is unrelated to display resolution or the individual pixel in the image.
- Display is more accurate but computationally more expensive. Due to the possibility of intersection between surfaces.
- Suitable for scene with small number of objects and objects with simple relationship with each other.

Object Space Methods

- Compare objects and parts of objects to each other within the scene definition.
- Determine which surfaces, as a whole, we should label as visible:

For each object in the scene do

Begin

1. Determine those parts of the object whose view is unobstructed by other parts of it or any other object with respect to the viewing specification.
2. Draw those parts in the object color.

End

Image-space Methods

- Mostly used methods
- Visibility is determined point by point at each pixel position on the projection plane

For each pixel in the image do

Begin

1. Determine the object closest to the viewer that is pierced by the projector through the pixel
2. Draw the pixel in the object color.

End

Image-space Methods

- For each pixel, examine all n objects to determine the one closest to the viewer.
- If there are p pixels in the image, complexity depends on n and p(O(np)).
- Accuracy of the calculation is bounded by the display resolution.
- A change of display resolution requires re-calculation.

Coherence in Visible Surface Detection Methods:

- Making use of the results calculated for one part of the scene or image for other nearby parts.
- Coherence is the result of local similarity.
- As objects have continuous spatial extent, object properties vary smoothly within a small local region in the scene. Calculations can then be made incremental.

Object space method vs Image space methods

S.No	Object space method	Image space methods
1	Vector graphics system	Raster graphics system
2	Accuracy is high	Easy computation
3	Back face detection	Depth buffer method
4	Continuous operation	Discrete

Types of coherence

- Object Coherence:
  - Visibility of an object can often be decided by examining a circumscribing solid (which may be of simple form, eg. A sphere or a polyhedron.)
- Face Coherence:
  - Surface properties computed for one part of a face can be applied to adjacent parts after small incremental modification. (eg. If the face is small, we sometimes can assume if one part of the face is invisible to the viewer, the entire face is also invisible).
- Edge Coherence:
  - The Visibility of an edge changes only when it crosses another edge, so if one segment of an nonintersecting edge is visible, the entire edge is also visible.

Types of coherence

- Scan line Coherence:
  - Line or surface segments visible in one scan line are also likely to be visible in adjacent scan lines. Consequently, the image of a scan line is similar to the image of adjacent scan lines.
- Area and Span Coherence:
  - A group of adjacent pixels in an image is often covered by the same visible object. This coherence is based on the assumption that a small enough region of pixels will most likely lie within a single polygon. This reduces computation effort in searching for those polygons which contain a given screen area (region of pixels) as in some subdivision algorithms.

Visible surface detection methods

- Back-face detection method → Preprocess
- Z-buffer or Depth buffer method → Very fast, Accurate results
- Scan line method
- Depth sorting method
- Area Sub division method
- BSP tree method
- Octree method
- Ray scan method

Types of coherence

- Depth Coherence:
  - The depths of adjacent parts of the same surface are similar.
- Frame Coherence:
  - Pictures of the same scene at successive points in time are likely to be similar, despite small changes in objects and viewpoint, except near the edges of moving objects.

Most visible surface detection methods make use of one or more of these coherence properties of a scene.

To take advantage of regularities in a scene, eg., constant relationships often can be established between objects and surfaces in a scene.

Back-face detection

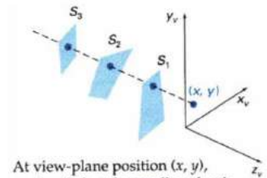
- In a solid object, there are surfaces which are facing the viewer (front faces) and there are surfaces which are opposite to the viewer (back faces).
- These back faces contribute to approximately half of the total number of surfaces.
- Since we cannot see these surfaces anyway, to save processing time, we can remove them before the clipping process with a simple test.
- Each surface has a normal vector.
- If this vector is pointing in the direction of the center of projection, it is a front face and can be seen by the viewer.
- If it is pointing away from the center of projection, it is a back face and cannot be seen by the viewer.

To test whether the points lie on the front of back surface

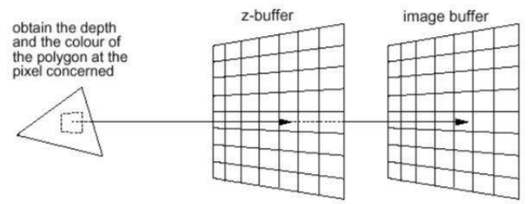
- suppose the z axis is pointing towards the viewer,
- if the z component of the normal vector is negative, then, it is a back face.
- If the z component of the vector is positive, it is a front face.
- Works well for nonoverlapping or convex objects.
- Concave objects or overlapping objects, we still need to apply other methods(**Depth or Z buffer method)** to further determine where the obscured faces are partially or completely hidden by other objects

Depth or Z buffer method

- **Image-space method**
- The basic idea is to test the Z-depth of each surface to determine the closest (visible) surface.
- This approach compares surface depths at each pixel position on the projection plane.

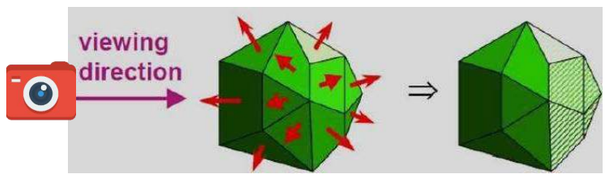


At view-plane position (x, y), surface S_1 has the smallest depth from the view plane and so is visible at that position.



Back-face detection

- Culling :Remove entire object or place entire object.
- Inside and outside test
  - To test whether the points lie on the surface or not
    - Use Front clipping plane to clip the front face
    - Use Back clipping plane to Cull the back face
- Fails in partial visible surfaces

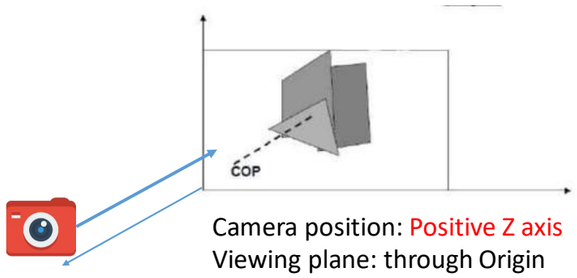


Z buffer method :Buffers

- It is applied very efficiently on surfaces (of polygon/spline/Any shape).
- Surfaces can be processed in any order.
- To override the closer objects from the far ones, two buffers named **image/frame buffer** and **z/depth buffer,** are used.
- **Depth buffer** is used to store depth values for (x, y) position, as surfaces are processed ($0 \leq$ depth ≤ 1).
  - *z* values is stored in depth buffers.
- The **frame buffer** is used to store the intensity value of color value at each position (x, y).

Z buffer method : Back clipping volume vs Front clipping volume

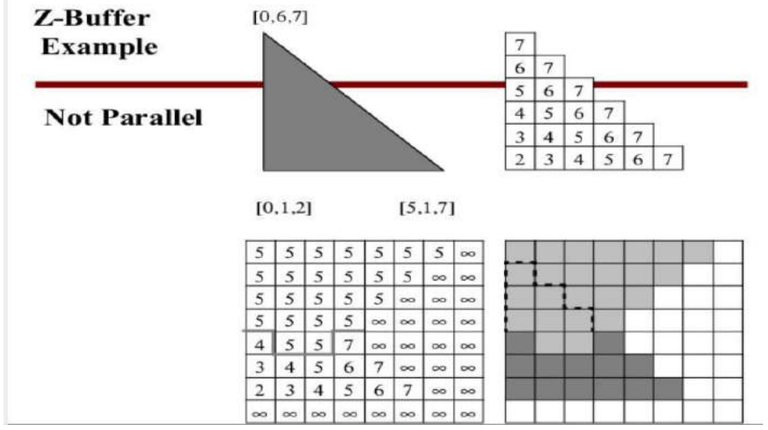
- The z-coordinates are usually normalized to the range [0, 1].
  - The 0 value for z-coordinate indicates back clipping volume .
  - The 1 value for z-coordinates indicates front clipping volume.



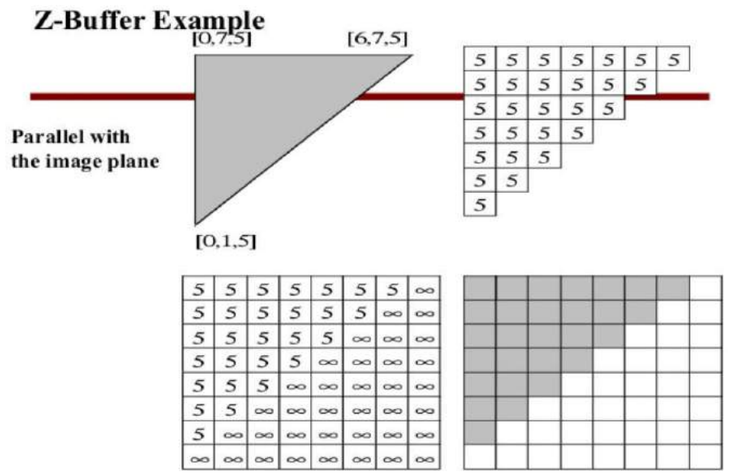
What happens if the camera position is moved to Negative Z axis?

Camera position: Positive Z axis
Viewing plane: through Origin
Z value : Zmin or 0

Second object processed



First object processed



Depth buffer or Z- Buffer algorithm

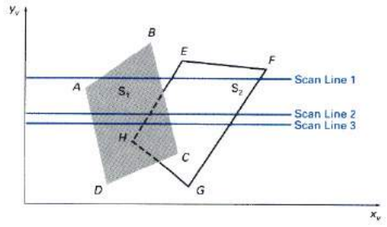
- Step 1: Initialize depth buffer and frame buffer
  - Z/Depth buffer (x, y) = infinity
  - Image/Framebuffer (x, y) = background color
- Step 2: Surfaces are rendered one at a time.
- Step 3:For each surface, the depth value Z of each pixel is calculated.
    - If Z < depth buffer (x, y)
      - Compute Surface color,
      - Set depth buffer (x, y) = z,
      - Frame buffer (x, y) = Surface color (x, y)

Advantages of Depth Buffer

- This method requires an additional buffer the overheads involved in updating the buffer.
- Not suitable for less objects.
- Simple and does not require additional data structures.
- The z-value of a polygon can be calculated incrementally.
- No pre-sorting of polygons is needed.
- No object-object comparison is required.
- Can be applied to non-polygonal objects

Scan Line method

- As each scan line is processed, all polygon surfaces intersecting that line are examined to determine which are visible.
- Across each scan line, depth calculations are made for each overlapping surface to determine which is nearest to the view plane.
- When the visible surface has been determined, the intensity value for that position is entered into the image buffer.



Some common questions?

- How games with a long map get rendered in the same display? How the coordinates are handled?
- In recent days, games are having auto saving options. But in olden days we have to manually save the game based on check points or direct save option. How the game gets saved? What will be the data stored?
- If the game has two or more object in a scene, if the game renders all the objects including occluding objects what will be the cost?

```
For each scan line do
    Begin
        For each pixel (x,y) along the scan line do          ---------- Step 1
        Begin
            z_buffer(x) = infinity
            Image_buffer(x,y) = background_color
        End

        For each polygon in the scene do                     ---------- Step 2
        Begin
            For each pixel (x,y) along the scan line that is covered by the polygon do
            Begin
                2a. Compute the depth or z of the polygon at pixel location (x,y).
                2b. If z < z_buffer(x) then
                        Set z_buffer(x) = z
                        Set Image_buffer(x,y) = polygon's colour
            End
        End
    End
```

Binary space partition trees

- Existing algorithms cannot deal with overlapping polygons (occlusion problem).
- BSP trees is a method that produces a data structure (BSP tree) of the polygon in the world space.
- BSP trees are often used in 3D video games, particularly first-person shooters and those with indoor environments.
- BSP trees containing the static geometry of a scene are often used together with a Z-buffer, to correctly merge movable objects such as doors and characters onto the background scene.

Use of BSP trees in map generation

- You can randomly generate map without any collision in polygons.
  - You can fix the static coordinates of the object and generate map without any collision
- Before the rendering of the map we must perform a number of calculations on it. However, once these calculations are performed their results can be used many times.
- This is one of the advantages of BSP — once the calculations are performed they do not need to be done again, unless the map is changed.
- For FPP games maps are static component and can be rendered from front to back
  - Frontal polygon renders first then followed by back polygons.

Need for BSP trees

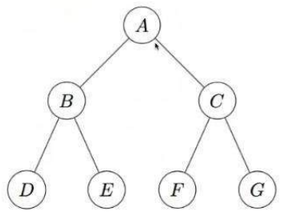
- One of the biggest problems faced by the designers in 3-D rendering engines is that of visibility calculation: only visible walls and objects must be drawn, and they must be drawn in the right order (close walls should be drawn in front of far away walls).
- More importantly, for applications such as games it is important to develop algorithms which allow a scene to be rendered quickly.
- Binary Space Partitioning (BSP) is a technique which can be used to greatly speed up visibility calculations in 3D Rendering.

Defining static object in the map

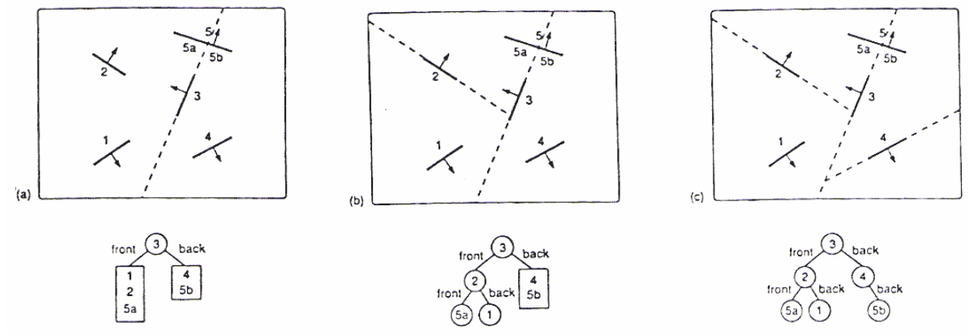
- Placing static objects in the map
- Demo IGI

Binary tree – A quick recap

- A binary tree consists of more than two nodes with nodes and branches
- Each node has at most two branches
- Nodes that are attached to branches of a node are called child nodes and the node above them are called parent node
- A node without a parent node is root node
- A node without child is called a leaf node

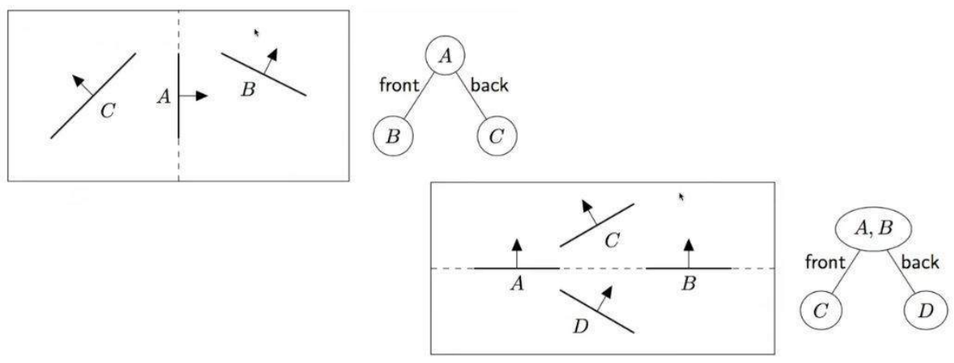


Example



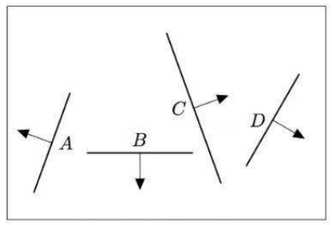

```
BSP Algorithm
Procedure DisplayBSP(tree: BSP_tree)
Begin
    If tree is not empty then
        If viewer is in front of the root then
            Begin
                DisplayBSP(tree.back_child)
                displayPolygon(tree.root)
                DisplayBSP(tree.front_child)
            End
        Else
            Begin
                DisplayBSP(tree.front_child)
                displayPolygon(tree.root)
                DisplayBSP(tree.back_child)
            End
End
```

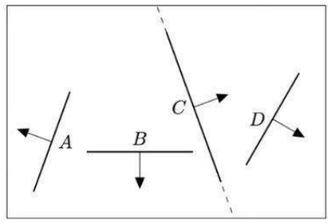
How BSP works



Coincident polygons and can be treated as one polygon

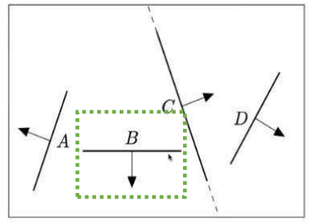
How BSP works – Four polygons

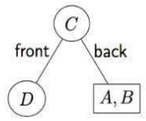




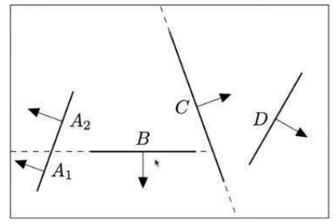
- I have to take C as my primary polygon. Why?
- Why not other polygons such as A, B and D?
- What will be the BSP?

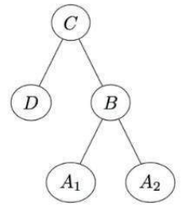
How BSP works – Four polygons





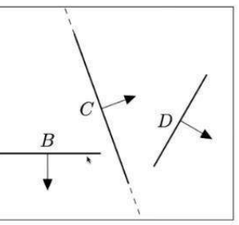
Step 5

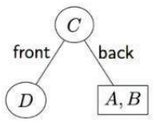


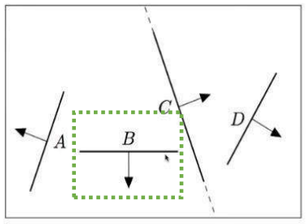


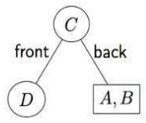
Step 6

How BSP works – Four polygons





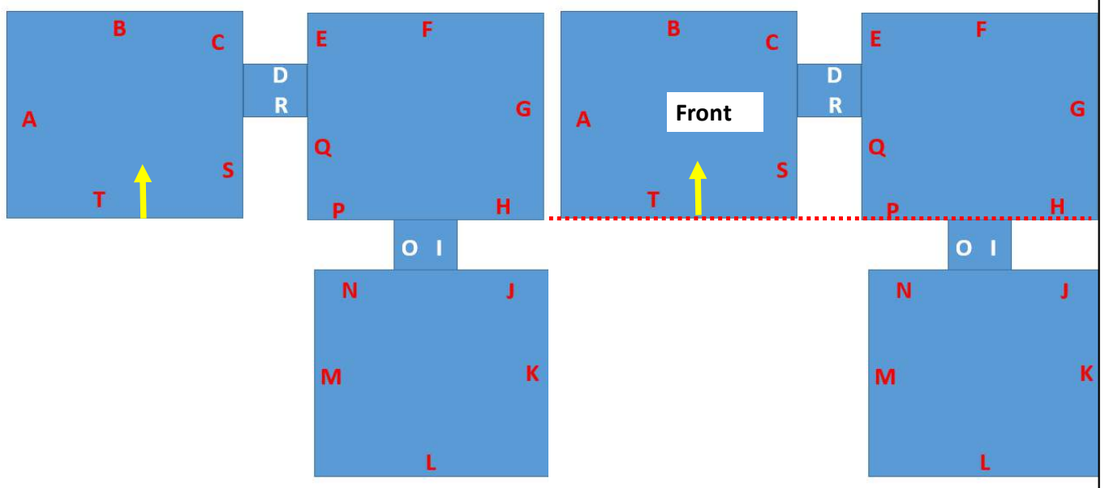




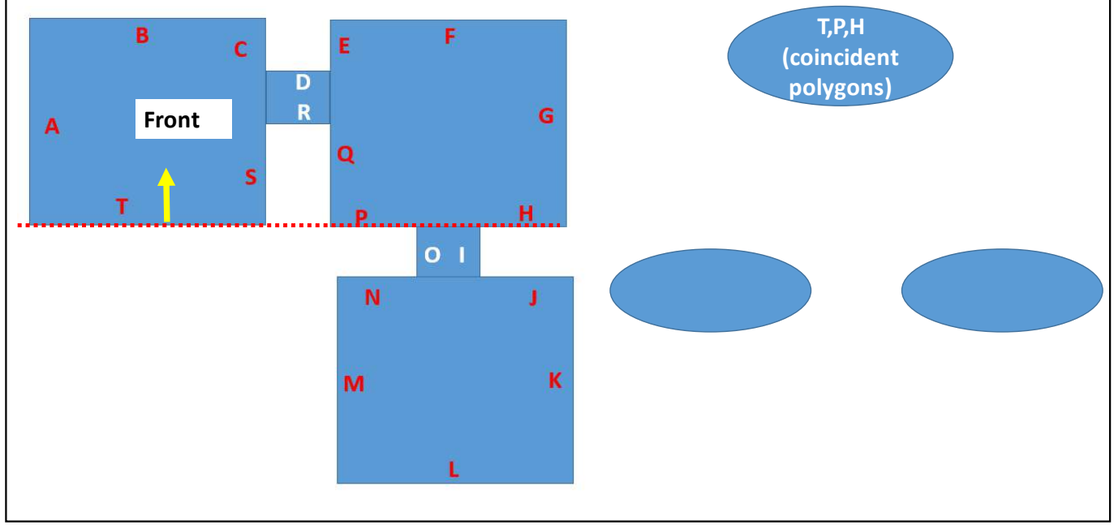
Step 3

Step 4

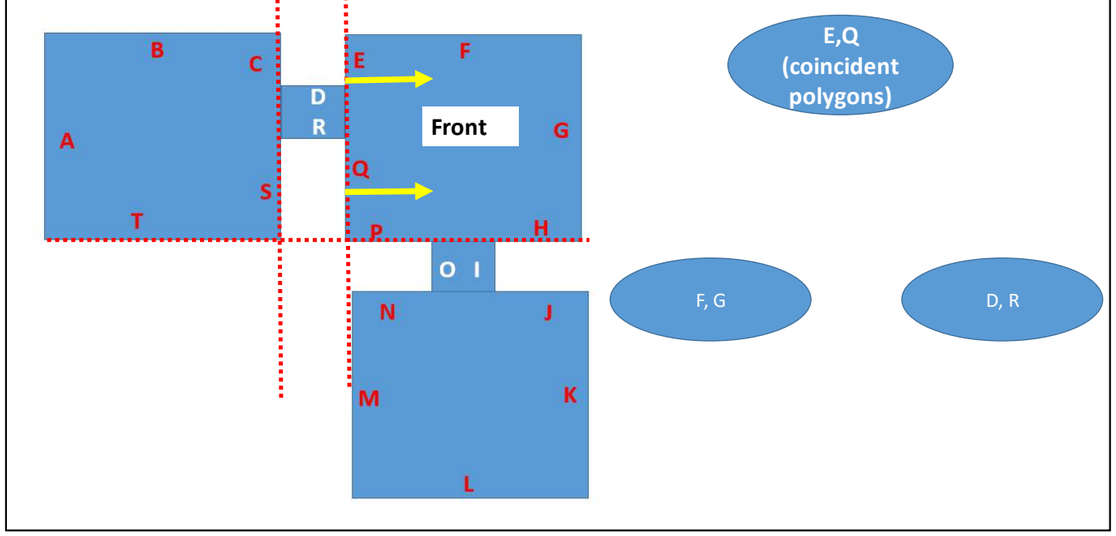
Let's we explore a map



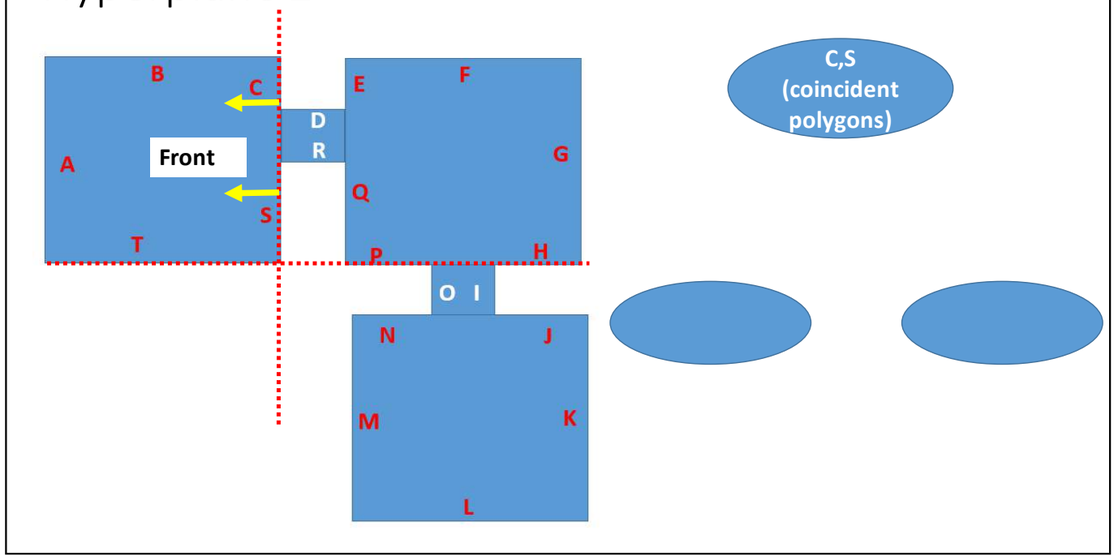
Hyper plane 1



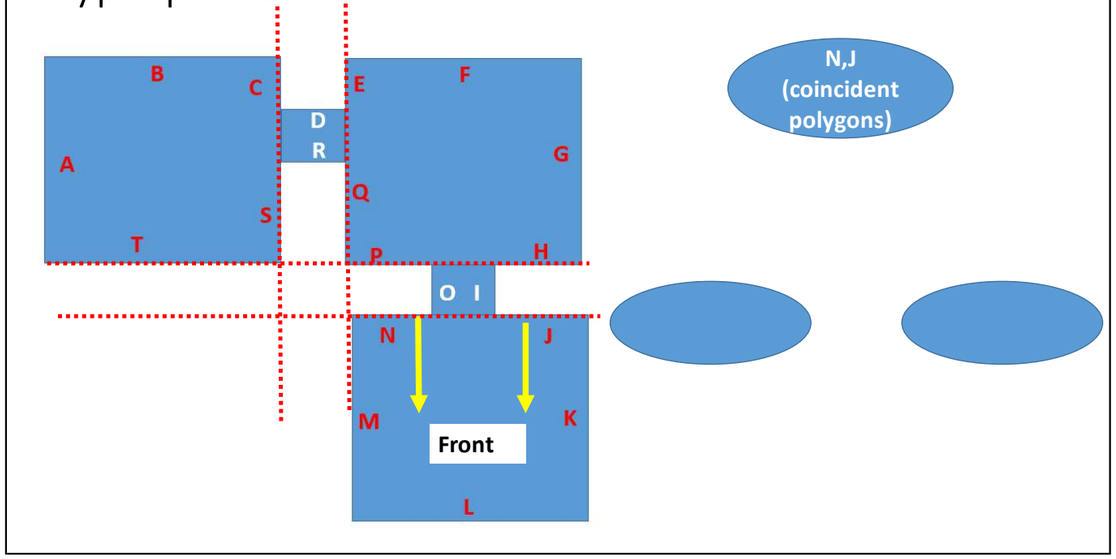
Hyperplane 3



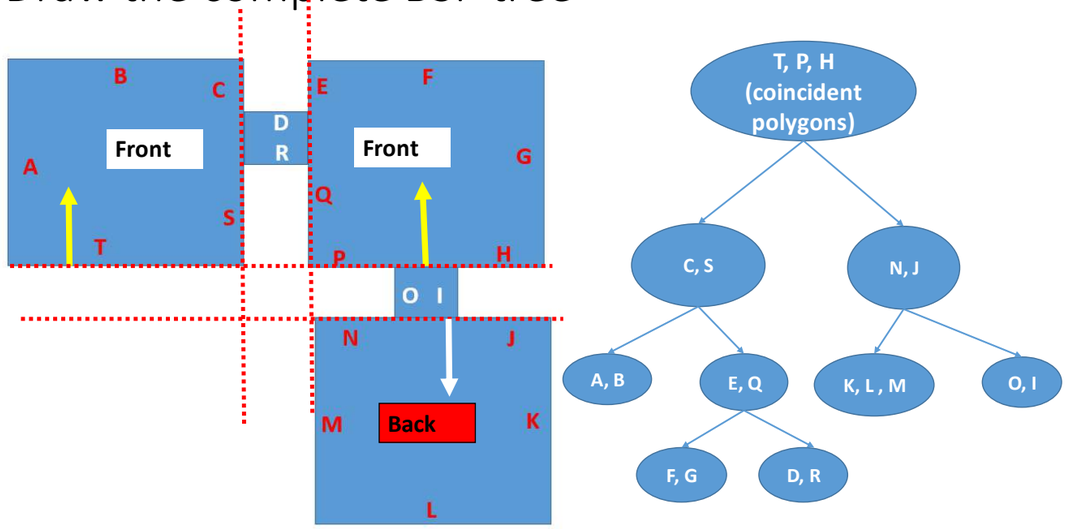
Hyperplane 2



Hyperplane 4



Draw the complete BSP tree



Rendering BSP trees

- **Back-to-front**
  - In a back to front renderer, far away walls are rendered first and are obscured by closer walls.
  - The disadvantage of back-to-front rendering is one of overdraw - parts of walls are drawn which are obscured by closer ones and are not seen. This is unnecessary overhead.
- **Front-to-back**
  - A front to back renderer works in the opposite way: closer walls are rendered first, and walls further away are clipped against the walls already drawn.
  - Almost all practical BSP renderers use a front-to-back method.

Construct BSP for these polygons

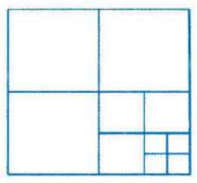
Area Sub division method

- The area-subdivision method takes advantage of area coherence in a scene.
- Locates those view areas based on coherence that represent part of a single surface.
- The total viewing area is successively divided into smaller and smaller rectangles until each small area is simple.
- This small area may be a single pixel, or is covered wholly by a part of a single visible surface or no surface at all.

Procedure for division of area-step 1

We first classify each of the surfaces, according to their relations with the area:

- Surrounding surface - a single surface completely encloses the area
- Overlapping surface - a single surface that is partly inside and partly outside the area
- Inside surface - a single surface that is completely inside the area



Dividing a square area into equal-sized quadrants at each step.

Possible relationships between polygon surfaces and a rectangular area.

Image and Object precision

Object precision: The computation results to machine precision (the precision used to represent object coordinates, most of the time it is floating point precision – depends on the hardware).

- The resulting image may be enlarged many times without significant loss of accuracy.
- The output is a set of visible object faces, and the portions of faces that are only partially visible.

Image precision: The computation results to the precision of a pixel of the image.

- Thus, once the image is generated, any attempt to enlarge some portion of the image will result in reduced resolution.

Procedure for division of area-step 2

- After step1, if any of the following condition is true, then, no subdivision of this area is needed.
  a. All surfaces are outside the area.
  b. Only one surface is inside, overlapping or surrounding surface is in the area.
  c. A surrounding surface obscures all other surfaces within the area boundaries.

Rendering in Image and Object

- **Object Space**
  - Geometric calculations involving polygons
  - Floating point precision: Exact
  - Process scene in object order
- **Image Space**
  - Visibility at pixel levels
  - Integer precision : Pixel
  - Process scene in image order

Summary

- The most appropriate algorithm to use depends on the scene
- Z buffer is particularly suited to the scenes with objects which are spreading out along the z-axis.
- Scan-line and area subdivision algorithms are suitable to the scenes where objects are spreading out horizontally and/or the scenes with small number of objects (about several thousand surfaces).
- Z-buffer and subdivision algorithms perform best for scene with fewer than a few thousand surfaces.

Syllabus

Module No. 5	Basic Raster Graphics	8 hours
Scan conversion; filling and clipping Light sources, Illumination model, Gouraud and Phong shading for polygons. Rasterization - Line segment and polygon clipping, 3D clipping, scan conversion, polygonal fill, Bresenham's algorithm.		

Basic Raster Graphics

Scan conversions

- The process of representing continuous objects with a collection of discrete pixels.
- The circuitry of the video display device of the computer is capable of converting binary values (0, 1) into a pixel on and pixel off information. 0 is represented by pixel off. 1 is represented using pixel on.
- Most human beings think graphics objects as points, lines, circles, ellipses.
- For generating graphical object, many algorithms have been developed.

- Advantage of developing algorithms for scan conversion

1. Algorithms can generate graphics objects at a faster rate.
2. Using algorithms memory can be used efficiently.
3. Algorithms can develop a higher level of graphical objects.

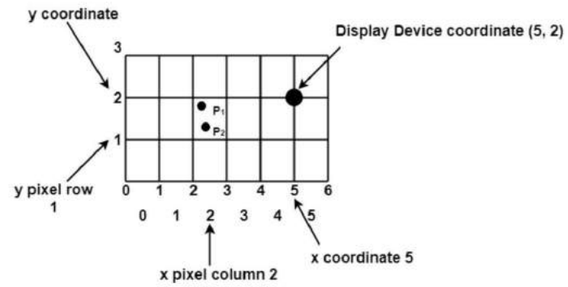
Examples of objects which can be scan converted

1. Point
2. Line
3. Sector
4. Arc
5. Ellipse
6. Rectangle
7. Polygon
8. Characters
9. Filled Regions

Scan Converting a Point

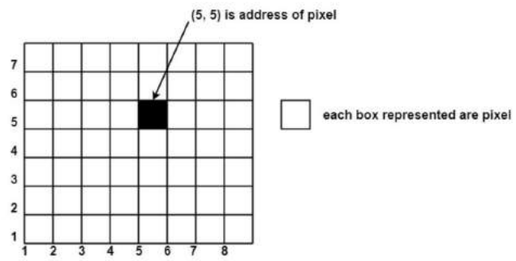
- Scan-Converting a point involves illuminating the pixel that contains the point.

Example: Display coordinates points $P_1(2\frac{1}{4}, 1\frac{3}{4})$ & $P_2(2\frac{2}{3}, 1\frac{1}{4})$ as shown in fig would both be represented by pixel (2, 1). In general, a point p (x, y) is represented by the integer part of x & the integer part of y that is pixels [(INT (x), INT (y)].



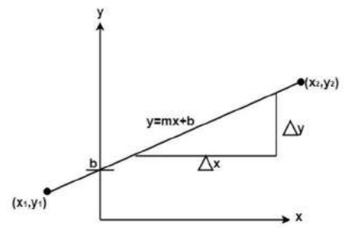
Pixel or Pel representation

- The term pixel is a short form of the picture element. It is also called a point or dot.
- P (5, 5) used to represent a pixel in the 5th row and the 5th column. Each pixel has some intensity value which is represented in memory of computer called a **frame buffer**



Scan Converting a Straight Line

- A straight line may be defined by two endpoints & an equation.
- In fig the two endpoints are described by (x1,y1) and (x2,y2).
- The equation of the line is used to determine the x, y coordinates of all the points that lie between these two endpoints.



Algorithm for line Drawing

1. Direct use of line equation
2. DDA (Digital Differential Analyzer)
3. Bresenham's Algorithm

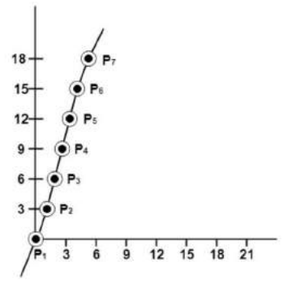
Direct use of line equation

- It is the simplest form of conversion.
- First of all scan P_1 and P_2 points. P_1 has co-ordinates (x_1,y_1) and (x_2, y_2).
- Then $m = (y_2-y_1)/(x_2-x_1)$ and $y_1 = mx_1 +b$
- If value of $|m| \leq 1$ for each integer value of x. But do not consider x_2 and x_1
- If value of $|m|>1$ for each integer value of y. But do not consider y_2 and y_1

Example

A line with starting point as (0, 0) and ending point (6, 18) is given. Calculate value of intermediate points and slope of line.

- P_1 (0,0) P_7 (6,18)=> $x_1=0$ $y_1=0$ $x_2=6$ $y_2=18$
- M=18-0/6-0=3
- Y=3x+b
- Put initial point(0,0) in Y=3x+b ➔ 0=3*0+b➔b=0
- Put b=0 in Y=3x+b this will give y=3x
- Now calculate intermediate points
  - Let x = 1 ⇒ y = 3 x 1 ⇒ y = 3
  - Let x = 2 ⇒ y = 3 x 2 ⇒ y = 6
  - Let x = 3 ⇒ y = 3 x 3 ⇒ y = 9
  - Let x = 4 ⇒ y = 3 x 4 ⇒ y = 12
  - Let x = 5 ⇒ y = 3 x 5 ⇒ y = 15
  - Let x = 6 ⇒ y = 3 x 6 ⇒ y = 18
- So points are P_1 (0,0)
  - P_2 (1,3)
  - P_3 (2,6)
  - P_4 (3,9)
  - P_5 (4,12)
  - P_6 (5,15)
  - P_7 (6,18)



Algorithm for drawing line using equation

Step1: Start Algorithm
Step2: Declare variables x_1,x_2,y_1,y_2,dx,dy,m,b,
Step3: Enter values of x_1,x_2,y_1,y_2.
The (x_1,y_1) are co-ordinates of a starting point of the line.
The (x_2,y_2) are co-ordinates of a ending point of the line.
Step4: Calculate $dx = x_2 - x_1$
Step5: Calculate $dy = y_2-y_1$
Step6: Calculate $m = \frac{dy}{dx}$
Step7: Calculate $b = y_1-m* x_1$
Step8: Set (x, y) equal to starting point, i.e., lowest point and x_{end} equal to largest value of x.
If dx < 0
then $x = x_2$
$y = y_2$
$x_{end} = x_1$
If dx > 0
then $x = x_1$
$y = y_1$
$x_{end} = x_2$
Step9: Check whether the complete line has been drawn if $x=x_{end}$, stop
Step10: Plot a point at current (x, y) coordinates
Step11: Increment value of x, i.e., x = x+1
Step12: Compute next value of y from equation y = mx + b
Step13: Go to Step9.

P5js program for Line drawing

```
function setup() {
createCanvas(400, 400);
}


function draw()
 {
     background(220);
    myLine(10,20,150,300);
     updatePixels();
}
```

```
function myLine(x1,y1,x2,y2){
 m=(y2-y1)/(x2-x1);
 b=y1-(m*x1);
 if (x2-x1 <0)
 {
    x=x2;
    y=y2;
    xend=x1;
 }
  else
 {
   x=x1;
  y=y1;
  xend=x2;
 }
 while (x<=xend)     {
     set (x, y, color('RED'));
     x++;
      y=(m*x) +b;
  }
  }
```

DDA (Digital Differential Analyzer)

- DDA stands for Digital Differential Analyzer.
- It is an incremental method of scan conversion of line.
- In this method calculation is performed at each step but by using results of previous steps.
- Suppose at step i, the pixels is (x_i, y_i)

The line of equation for step i

- $y_i = m_{xi} + b$.....equation 1
- Next value will be

  $y_{i+1} = m_{xi+1} + b$.....equation 2

  $m = \frac{\Delta y}{\Delta x}$ .....equation 3

  $y_{i+1} - y_i = dy$.....equation 4

  $x_{i+1} - x_i = dx$.....equation 5

If (dx) > (dy)
step = dx
else
step=dy

$x_{i+1}=x_i+dx$ from equation 5
$y_{i+1}=y_i+dy$ from equation 4.....eqn 6

dy=m*dx from equation 4----eqn 7

Substitute eqn 7 in eqn 6
$y_{i+1}=y_i+ m*dx$ and
$x_{i+1}=x_i+dx$

the constant is divided with step which is dx as we replaced dy with xdx assuming dx>dy

Hence m<1 we will get

$x_{i+1}=x_i+dx$ from equation 5
$y_{i+1}=y_i+dy$ from equation 4.....eqn 6

dx=dy/m from equation 4----eqn 7

Substitute eqn 7 in eqn 5
$x_{i+1}=x_i+dy/m$
$y_{i+1}=y_i+ dy$

the constant is divided with step which is dy as we replaced dx with dy/m assuming dy>dx

Hence m>1 we will get

DDA Algorithm Two case:

Case 1: $m < 1$	$x_{i+1} = x_i + 1$
	$y_{i+1} = y_i + m$
Case 2: $m > 1$	$x_{i+1} = x_i + (1/m)$
	$y_{i+1} = y_i + 1$

- DDA Algorithm:
- **Step1:** Start Algorithm
- **Step2:** Declare x_1, y_1, x_2, y_2, dx, dy, x, y as integer variables.
- **Step3:** Enter value of x_1, y_1, x_2, y_2.
- **Step4:** Calculate $dx = x_2 - x_1$
- **Step5:** Calculate $dy = y_2 - y_1$
- **Step6:** If ABS (dx) > ABS (dy)
  Then step = abs (dx)
  Else
- step=abs(dy)
- **Step7:** $x_{inc} = dx/step$
  $y_{inc} = dy/step$
  assign $x = x_1$
  assign $y = y_1$
- **Step8:** Set pixel (x, y)
- **Step9:** $x = x + x_{inc}$
  $y = y + y_{inc}$
  Set pixels (Round (x), Round (y))
- **Step10:** Repeat step 9 until $x = x_2$
- **Step11:** End Algorithm

DDA p5.js

```
function setup()
{  createCanvas(400, 400);
}
function draw()
{   background(220);
myLine(10,10,50,300);
 updatePixels();
}
```

```
function myLine(x1,y1,x2,y2)
{  background(220);
  dx=x2-x1;
  dy=y2-y1;
  if(dx>=dy)
   {         steps = dx;     }
    else
    {         steps = dy;     }
   dx = dx/steps;
    dy = dy/steps;
    x = x1;      y = y1;      i = 1;
    while(i<= steps)
    {
          set(x, y, color('RED'));
          x += dx;
          y += dy;
          i=i+1;
    }
 }
```

Bresenham Line algorithm

- The Bresenham algorithm is another incremental scan conversion algorithm.
- It is useful alternative for the DDA, The big advantage of DDA algorithm is that it uses only integer calculations.
- It is an efficient method because it involves only integer addition, subtractions, and multiplication operations.
- These operations can be performed very rapidly so lines can be generated quickly.
- In this method, next pixel selected is that one who has the least distance from true pixel.

- Advantage:

1. It is a faster method than method of using direct use of line equation.
2. This method does not use multiplication theorem.
3. It allows us to detect the change in the value of x and y ,so plotting of same point twice is not possible.
4. This method gives overflow indication when a point is repositioned.
5. It is an easy method because each step involves just two additions.

- Disadvantage:

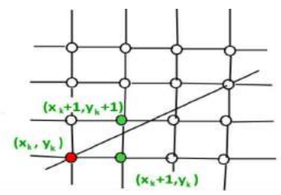
1. It involves floating point additions rounding off is done. Accumulations of round off error cause accumulation of error.
2. Rounding off operations and floating point operations consumes a lot of time.
3. It is more suitable for generating line using the software. But it is less suited for hardware implementation.

- Bresenham Line Drawing Algorithm contains two phases :

1. slope(m) < 1.

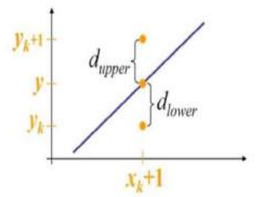
2 slope(m) > 1.

2 slope(m) = 1.



To chooses the next one between the bottom pixel L and top pixel T.

If L is chosen
We have $x_{i+1}=x_i+1$ and $y_{i+1}=y_i$
If T is chosen
We have $x_{i+1}=x_i+1$ and $y_{i+1}=y_i+1$



The actual y coordinates of the line at x = xi+1is

y=m(xi+1)+b …eqn 1

The dl distance from L to the actual line in y direction

dlower = y-yi

The distance from T to the actual line in y direction

dupper = (yi+1)-y

Now consider the difference between these 2 distance values

if(dlower-dupper)<0 indicates L is close

else if (dlower-dupper)>=0 indicates T is close

So we will find the difference;

This difference is

dlower-dupper = (y-yi)-[(yi+1)-y]

= 2y - 2yi -1 substitute eqn1 in place of y

=2(m(xi+1)+b)-2yi-1

=2m(xi+1)+2b-2yi-1

(dlower-dupper)dx =(2m(xi+1)+2b-2yi-1) dx --------eqn 2 multiply dx on both sides

Let (dlower-dupper)dx be di.

di=2dy(xi+1)+2dxb-2yidx-dx

di=2xidy+2dy+2dxb-2yidx-dx

di=2xidy-2yidx+2dy+2dxb-dx let c=2dy+2dxb-dx as it is constant

di= 2xidy-2yidx+c

di= 2dyxi-2dxyi+c

we can write the decision variable di+1 for the next slip on

di+1=2dy(xi+1)-2dx(yi+1)+c

Lets differentiate b/w di+1 and di

di+1-di=2dy(xi+1-xi)- 2dx(yi+1-yi)

di+1=di+2dy(xi+1-xi)- 2dx(yi+1-yi)

- Special Cases
- If chosen pixel is at the top pixel T (i.e., $d_i \geq 0 \Rightarrow y_{i+1}=y_i+1$
  $d_{i+1}=d_i+2dy-2dx$
- If chosen pixel is at the bottom pixel L (i.e., $d_i<0 \Rightarrow y_{i+1}=y_i$
  $d_{i+1}=d_i+2dy$
- Finally, we calculate d1 by substituting (x1,y1) in eqn 2
  d1=dx[2m(x1+1)+2b-2y1-1]
  d1=dx[2(mx1+b-y1)+2m-1]
- Since mx1+b-y1=0 and m = dy/dx , we have
  d1=2dy-dx
- first d1 (if di≥0)$\Rightarrow y_{i+1}=y_i+1$, $d_{i+1}=d_i+2dy-2dx$ ➔I1
- else (i.e., $d_i<0 \Rightarrow y_{i+1}=y_i$ $d_{i+1}=d_i+2dy$ ➔I2

Example: Starting and Ending position of the line are (1, 1) and (8, 5). Find intermediate points.

Solution: x1=1, y1=1 , x2=8 , y2=5

dx= x2-x1=8-1=7

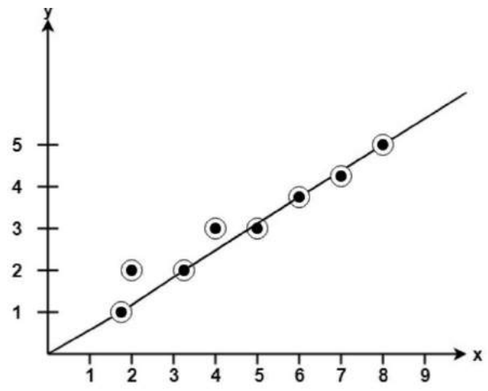
dy=y2-y1=5-1=4

I1=2* dy=2*4=8

I2=2*(dy-dx)=2*(4-7)=-6

d1 = 2dy-dx=8-7=1 >0

x	y	d=d+I1 or I2
1	1	d+I2=1+(-6)=-5
2	2	d+I1=-5+8=3
3	2	d+I2=3+(-6)=-3
4	3	d+I1=-3+8=5
5	3	d+I2=5+(-6)=-1
6	4	d+I1=-1+8=7
7	4	d+I2=7+(-6)=1
8	5	



Step1: Start Algorithm

Step2: Declare variable $x_1,x_2,y_1,y_2,d,i_1,i_2,dx,dy$

Step3: Enter value of x_1,y_1,x_2,y_2
Where $x_1,y_1$are coordinates of starting point
And x_2,y_2 are coordinates of Ending point

Step4: Calculate $dx = x_2-x_1$, Calculate $dy = y_2-y_1$
Calculate $i_1=2*dy$
Calculate $i_2=2*(dy-dx)$
Calculate $d=2*dy-dx$

Step5: Consider (x, y) as starting point and x_{end} as maximum possible value of x.
If $dx < 0$
Then $x = x_2$
$y = y_2$
$x_{end}=x_1$
If $dx > 0$
Then $x = x_1$
$y = y_1$
$x_{end}=x_2$

Step6: Generate point at (x,y)coordinates.

Step7: Check if whole line is generated.
If $x >= x_{end}$
Stop.

Step8: Calculate co-ordinates of the next pixel
If $d < 0$
Then $d = d + i_1$
If $d \geq 0$
Then $d = d + i_2$
Increment $y = y + 1$

Step9: Increment $x = x + 1$

Step10: Draw a point of latest (x, y) coordinates

Step11: Go to step 7

Step12: End of Algorithm

```
function myLine(x1,y1,x2,y2){
        dy=(y2-y1);
        dx=(x2-x1);
         i1=2*dy;
        i2=2*(dy-dx);
        d=2*dy-dx;
         if (x2-x1 <0)
         {   x=x2;
               y=y2;
                xend=x1;
            }
         else {
               x=x1;
                y=y1;
                 xend=x2;
              }
        while (x<=xend)     {
           if(d<0)    {     d=d+i1;    }
              else    {     d=d+i2;     y=y+1;   }
              set (x, y, color('RED'));
               x++;
        }
    }
```

```
function setup()
{
    createCanvas(400, 400);
}
function draw()
 {
          background(220);
         myLine(50,10,10,300);
         updatePixels();
}
```


Drawing Circles and Arcs

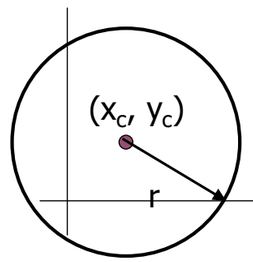
- Similar to line drawing
  - In that you have to determine what pixels to activate
- But non-linear
  - Simple equation implementation
  - Optimized

Cartesian form of Circle Equations

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

or

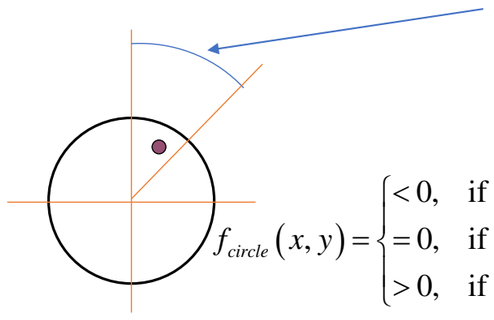
$$y = y_c \pm \sqrt{r^2-(x-x_c)^2}$$



Not a very good method. Why?

Bresenham's Midpoint Circle Algorithm

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

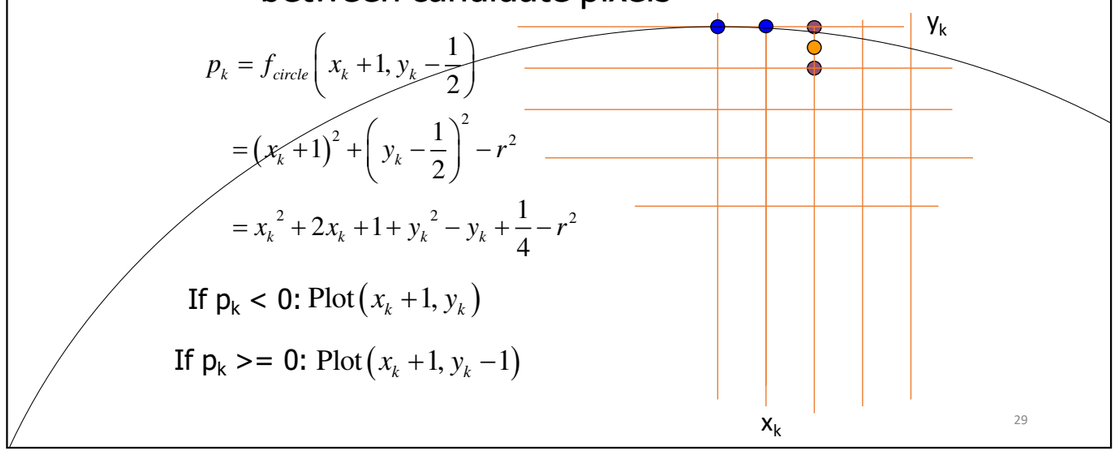


0<|m|<1 in this region

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

The Circle Decision Parameter

Calculate f_{circle} for point midway between candidate pixels



$$p_k = f_{circle}\left(x_k + 1, y_k - \frac{1}{2}\right)$$

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

$$= x_k^{\,2} + 2x_k + 1 + y_k^{\,2} - y_k + \frac{1}{4} - r^2$$

If $p_k < 0$: Plot $\left(x_k + 1, y_k\right)$

If $p_k >= 0$: Plot $\left(x_k + 1, y_k - 1\right)$

Calculating p_{k+1}

$$p_{k+1} = f_{circle}\left(x_{k+1} + 1, y_{k+1} - \frac{1}{2}\right)$$

$$= \left(x_k + 1 + 1\right)^2 + \left(y_{k+1} - \frac{1}{2}\right)^2 - r^2$$

$$= x_k^{\,2} + 4x_k + 4 + y_{k+1}^{\,2} - y_{k+1} + \frac{1}{4} - r^2$$

Recurrence Relation for p_k

$$p_k = x_k^{\,2} + 2x_k + 1 + y_k^{\,2} - y_k + \frac{1}{4} - r^2$$

$$p_{k+1} = x_k^{\,2} + 4x_k + 4 + y_{k+1}^{\,2} - y_{k+1} + \frac{1}{4} - r^2$$

$$p_{k+1} = p_k + (2x_k + 2) + 1 + \left(y_{k+1}^{\,2} - y_k^{\,2}\right) - \left(y_{k+1} - y_k\right)$$

$$= p_k + 2x_{k+1} + 1 + \left(y_{k+1}^{\,2} - y_k^{\,2}\right) - \left(y_{k+1} - y_k\right)$$

One Last Term …

$$\left(y_{k+1}{}^2 - y_k{}^2\right) - \left(y_{k+1} - y_k\right) = ?$$

If $y_{k+1} = y_k$, then:

$$\left(y_k{}^2 - y_k{}^2\right) - \left(y_k - y_k\right) = 0$$

If $y_{k+1} = y_k - 1$, then:

$$\left(\left(y_k - 1\right)^2 - y_k{}^2\right) - \left(\left(y_k - 1\right) - y_k\right) =$$

$$\left(y_k{}^2 - 2y_k + 1 - y_k{}^2\right) - \left(-1\right) =$$

$$-2y_k + 2 = -2\left(y_k - 1\right)$$

Initial Values

$$\left(x_0, y_0\right) = \left(0, r\right)$$

$$p_0 = f_{circle}\left(0+1, r - \frac{1}{2}\right)$$

$$= 1 + \left(r - \frac{1}{2}\right)^2 - r^2$$

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

$$= \frac{5}{4} - r \approx 1 - r$$

Bresenham Midpoint Circle Algorithm Summary

At each point:

If $p_k < 0$:

$$\text{Plot}\left(x_k + 1, y_k\right)$$

$$p_{k+1} = p_k + 2\left(x_k + 1\right) + 1$$

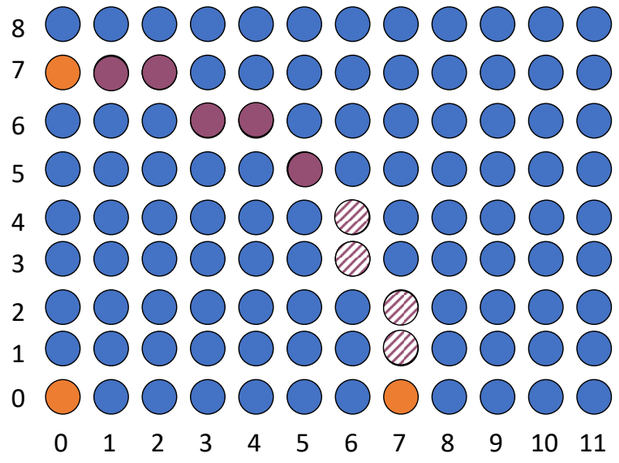
If $p_k >= 0$:

$$\text{Plot}\left(x_k + 1, y_k - 1\right)$$

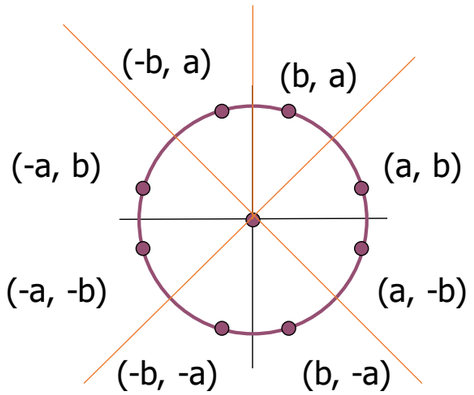
$$p_{k+1} = p_k + 2\left(x_k + 1\right) + 1 - 2\left(y_k - 1\right)$$

Circle Example 1

r = 7



Symmetry Optimization



Calculate points for one octant; replicate in other seven


```
function setup() {
  createCanvas(400, 400);
}
function draw() {
  background(220);
  BresenhamCircle(100,50,50);
  updatePixels();
}
function EightWaySymmetricPlot(xc,yc,x,y)
  {
   set(x+xc,y+yc,color('RED'));
   set(x+xc,-y+yc,color('YELLOW'));
   set(-x+xc,-y+yc,color('GREEN'));
   set(-x+xc,y+yc,color('YELLOW'));
   set(y+xc,x+yc, color('RED'));
   set(y+xc,-x+yc, color('YELLOW'));
   set(-y+xc,-x+yc, color('GREEN'));
   set(-y+xc,x+yc, color('RED'));
  }
```

```
function BresenhamCircle( xc,yc,r)
{
 x=0;
 y=r;
 d=1-r;
 EightWaySymmetricPlot(xc,yc,x,y);

 while(x<=y)
 {
  if(d<=0)
   {
    d=d+(2*x)+3;
   }
  else
   {
    d=d+(2*x)-(2*y)+5;
    y=y-1;
   }
    x=x+1;
    EightWaySymmetricPlot(xc,yc,x,y);
  }
 }
```

Example: Plot 6 points of circle using Bresenham Algorithm.
When radius of circle is 10 units. The circle has centre (50, 50).
Solution: Let r = 10 (Given)
Step1: Take initial point (0, 10)

d = 1-r
d = 1-10 = -9
d < 0 ∴ d = d + 2x + 3
= -9 + 2 (0) + 3
= -6

Step2: Plot (1, 10)

d = d + 2x + 3 (∵ d < 0)
= -6 + 2 (1) + 3
= -1

Step3: Plot (2, 10)

d = d + 2x + 3 (∵ d < 0)
= -1 + 2 x 2 + 3
= 6

Step4: Plot (3, 9) d is > 0 so x = x + 1, y = y - 1
d = d + 2 (x-y) + 5 (∵ d > 0)
= 6+ 2 (3-9) + 5
= 6 + 2 (-6) + 5
= 6-12+5=-1

Step5: Plot (4, 9)

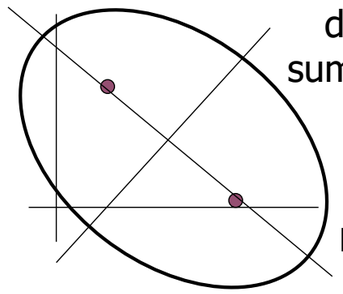
d = -6 + 2x + 3
= -6 + 2(4) + 3
= 11

Step6: Plot (5, 8)

d = d + 2 (x-y) + 5 (∵ d > 0)
= 11 + 2 (5-8) + 5
= 11-6 + 5 = 10

So P_1 (0,0)⟹(50,50)
P_2 (1,10)⟹(51,60)
P_3 (2,10)⟹(52,60)
P_4 (3,9)⟹(53,59)
P_5 (4,9)⟹(54,59)
P_6 (5,8)⟹(55,58)

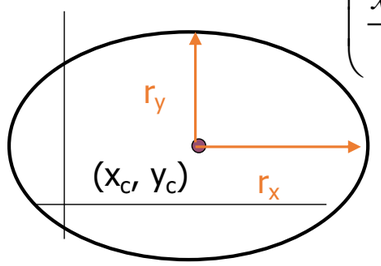
General Ellipse Equation



In general, ellipse described by constant sum of distances from foci

Difficult to solve equations and plot points (use parametric polar form?)

Special Case Ellipses



$$\left(\frac{x-x_c}{r_x}\right)^2+\left(\frac{y-y_c}{r_y}\right)^2=1$$

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

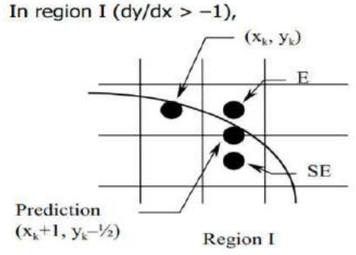
Modified midpoint algorithm possible

Consider the general equation of an ellipse,
$b^2x^2 + a^2y^2 - a^2b^2 = 0$

where a is the horizontal radius and b is the vertical radius.

Set $f(x,y) = b^2x^2 + a^2y^2 - a^2b^2$

In region I (dy/dx > −1),



x is always incremented in each step, i.e. $x_{k+1} = x_k + 1$.
$y_{k+1} = y_k$ if E is selected, or $y_{k+1} = y_k - 1$ if SE is selected.

In order to make decision between S and SE, a prediction $(x_k+1, y_k-½)$ is set at the middle between the two candidate pixels. A prediction function Pk can be defined.

Drawing Ellipses

- General ellipses
  - Polygon approximation
  - Rotation (we'll defer this until later, when we cover coordinate transformations)
- Aligned axes
  - Constrained (no rotation)
  - Midpoint algorithm

Equations used

Mid-Point Ellipse Algorithm :

1. Take input radius along x axis and y axis and obtain center of ellipse.
2. Initially, we assume ellipse to be centered at origin and the first point as : $(x, y_0)= (0, b)$.
- Obtain the initial decision parameter for region 1 as: $p1_0=a^2+1/4a^2-a^2b$
1. For every x_k position in region 1 :If $p1_k<0$ then the next point along the is (x_{k+1}, y_k) and $p1_{k+1}=p1_k+2b^2x_{k+1}+b^2$
2. Else, the next point is (x_{k+1}, y_{k-1})
   And $p1_{k+1}=p1_k+2b^2x_{k+1} - 2a^2y_{k+1}+b^2$
3. Obtain the initial value in region 2 using the last point (x_0, y_0) of region 1 as: $p2_0=b^2(x_0+1/2)^2+a^2 (y_0-1)^2-a^2b^2$
4. At each y_k in region 2 starting at k =0 perform the following task.If $p2_k>0$ the next point is (x_k, y_{k-1}) and $p2_{k+1}=p2_k-2a^2y_{k+1}+b^2$
5. Else, the next point is (x_{k+1}, y_{k-1}) and $p2_{k+1}=p2_k+2a^2x_{k+1} -2a^2y_{k+1}+a^2$
6. Now obtain the symmetric points in the three quadrants and plot the coordinate value as: x=x+xc, y=y+yc
7. Repeat the steps for region 1 until $2b^2x >= 2a^2y$ and $2a^2y >= 2b^2x$

Example- Straight line

- A line with starting point as (5, 12) and ending point (2, 3) is given. Calculate value of intermediate points and slope of line.

Example 3: Breshenhams Line drawing

- If a line is drawn from P_1 (5,6) P_2 (8,12) with use of Bresenham. How many points will needed to generate such line?

Example 2 DDA

- If a line is drawn from P_1 (5,6) P_2 (8,12) with use of DDA. How many points will needed to generate such line?

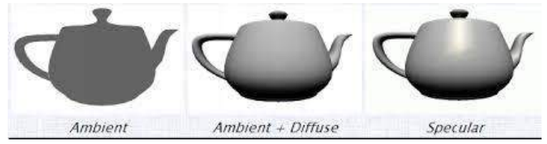
Example 4:

- Design a circle by obtaining the point with radius of 5 from origin 20,20.

Example 5

- In the example 4 problem obtain the points on the circle and draw a polygon.

Guess ?



Lighting

What is the need for light/illumination? Why lighting is so important?

lighting is a very important part of image synthesis. The proper use of lights in a scene is one of the things that differentiates the talented CG people from the untalented.

Lighting is a fundamental to film because it creates a visual mood, atmosphere, and sense of meaning for the audience. Whether it's dressing a film set or blocking actors, every step of the cinematic process affects the lighting setup, and vice-versa. Lighting tells the audience where to look.

Just take a moment and think

- Think about star wars without light
- Think about avatar without light

Illumination model

- **Illumination model**, also known as Shading **model** or Lightning **model** is used to **calculate the intensity of light** that is **reflected at a given point of surface**.
- Rendering methods use the intensity calculations from the illumination model to determine the light intensity at all the pixels in the image, by possibly, considering light propagation between the surfaces in the screen.

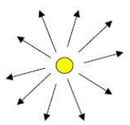
Illumination

- Factors of illumination
  - Light
  - Reflection
  - Surface
  - Shadows (dynamic)

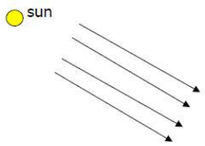
There are three factors on which lightening effect depends on:

1. **Light Source**
2. **Surface**
3. **Observer**

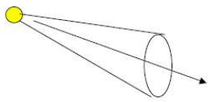
Basic Light Sources



Point light



Directional light



Spot light

Light intensity can be independent or dependent of the distance between object and the light source

light sources

1. Light source

- It is the light emitting source. It has three types.

1. Point source
   - The source that emit rays in all directions
   - Light ray coming from particular point. (A bulb in a room/torch light).
2. Parallel source
   - Can be considered as a point source which is far from the surface
   - Light rays are parallel to each other (infinite rays).

   Example: Laser, Sun
3. Distributed source
   - Rays originate from a finite area
   - Light rays are distributed and finite (room tubelight).

- Their position, electromagnetic spectrum and shape determine the lightning effect.

More about Lights and reflection

- **Ambient light means** the **light** that is already present in a scene, before any additional **lighting** is added. It usually refers to natural **light,** either outdoors or coming through windows etc. It can also **mean** artificial **lights** such as normal room **lights**.
- Ambient lights are almost always used in combination with other types of lights.
- **Lights()**
- Sets the default ambient light, directional light.

Lights and reflection – cont'd

- **Directional light**
- Directional light comes from one direction
- It is stronger when hitting a surface squarely, and weaker if it hits at a gentle angle.
- After hitting a surface, directional light scatters in all directions.

- **directionalLight (100, 100, 100, -0.3, 0.5, -1) //x,y,z,nx,ny,nz**

Lights and reflection – cont'd

- spotlight
- Adds a spot light.

spotlight (H,S,V,x,y,z,nx,ny,nz,angle,concentration)

(x, y, and z parameters set the position of the light)

Angle - defines the angle of the spotlight cone

Concentration - sets the bias of light focusing toward the center of that cone

v1	float: red or hue value (depending on current color mode)
v2	float: green or saturation value (depending on current color mode)
v3	float: blue or brightness value (depending on current color mode)
x	float: x-coordinate of the light
y	float: y-coordinate of the light
z	float: z-coordinate of the light
nx	float: direction along the x axis
ny	float: direction along the y axis
Nz	float: direction along the z axis
angle	float: angle of the spotlight cone
concentration	float: exponent determining the center bias of the cone

Lights and reflection – cont'd

- Pointlight
- Adds a point light.

Pointlight (H,S,V,x,y,z)

(x, y, and z parameters set the position of the light.)

Filling lights

- Lightspecular()
  - Sets the specular color for lights.
  - Like fill()
- Lightspecular(H,S,V)

Point Light

```
function setup() {
  createCanvas(400, 400, WEBGL);
}
function draw() {
background(0);
let locX = mouseX - width / 2;
 let locY = mouseY - height / 2;
pointLight(250, 250, 250, locX, locY, 50);
rotateX(millis() / 1000);
rotateY(millis() / 1000);
 rotateZ(millis() / 1000);
fill(color('red'));
noStroke();
 sphere(100);}
```

Ambient Light

```
function setup() {
  createCanvas(400, 400, WEBGL);
}
function draw() {
  background(0);
 let locX = mouseX - width / 2;
 let locY = mouseY - height / 2;
 ambientLight(50, 100, 200, locX, locY, 50);
  rotateX(millis() / 1000);
rotateY(millis() / 1000);
rotateZ(millis() / 1000);
// fill(color('red'));
noStroke();
sphere(100);}
```

Light

```
function setup() {
  createCanvas(100, 100, WEBGL);
}
function draw() {
  background(0);
  lights();
  rotateX(millis() / 1000);
  rotateY(millis() / 1000);
  rotateZ(millis() / 1000);
  sphere(100);
}
```

Spot Light

```
function setup() {
 createCanvas(400, 400, WEBGL);
}
function draw() {
 background(255);
 let locX = mouseX - width / 2;
let locY = mouseY - height / 2;
 spotLight(150, 100, 100, locX, locY, 100, 50, 50, -1, Math.PI / 16);
 rotateX(millis() / 1000);
rotateY(millis() / 1000);
 rotateZ(millis() / 1000);
 fill(color('red'));
 noStroke();
  sphere(100);}
```

```
function setup() {
createCanvas(100, 100, WEBGL);
}
function draw() {
background(0);
let dirX = (mouseX / width - 0.5) * 2;
 let dirY = (mouseY / height - 0.5) * 2;
  directionalLight(250, 250, 250, -dirX, -dirY, -1);
 noStroke();
 sphere(40);}
```

2. Surface

- What is a Surface?
  - Rays and Objects interaction
  - Object surface or surface plays a major role
- What are different types of surface you know.
  - Specular surface
  - Diffuse surface
  - Translucent surface (allow light to shine)

When light falls on a surface part of it is reflected and part of it is absorbed. Now the surface structure decides the amount of reflection and absorption of light. The position of the surface and positions of all the nearby surfaces also determine the lightning effect.

3.Observer

- The observer's position and sensor spectrum sensitivities also affect the lightning effect.

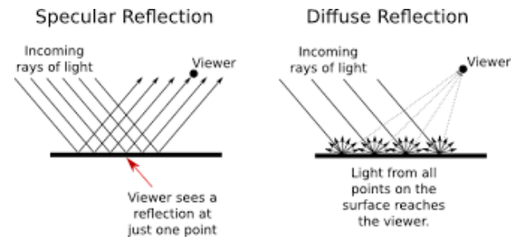
Surface

- Specular()
  - Sets the specular reflectance (color) of the materials
- Ambient()
  - Sets the ambient reflectance (color) of the materials
- Emissive()
  - Sets the diffusive reflectance (color) of the materials

Three Surface reflection Components to **Illumination** of Objects(refer from slide 24 for more details)

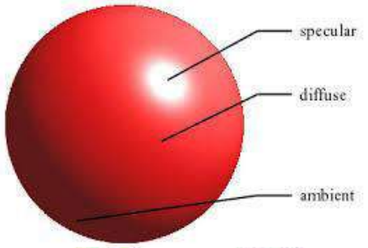
- Ambient reflection
- Diffuse reflection
- Specular reflection

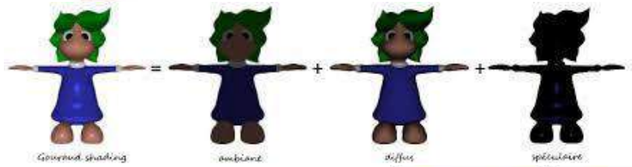
- Ambient - Total of all the light bouncing.
- Diffuse reflection - Matted surfaces.
- Specular reflection - Mirror effect

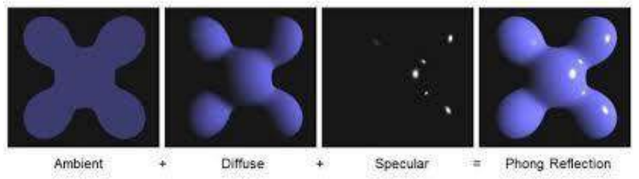


How did you see an object?

- Completely based on reflection.
- Ambient **Illumination(natural light)**
  - Lights are coming from different sources
  - Reflected rays or lights
  - In simple words, Ambient Illumination is the one where source of light is indirect.
- Diffuse reflection or irregular reflection
  - Diffuse reflection occurs on the surfaces which are rough or grainy /Happens with rough surfaces
  - In this reflection the brightness of a point depends upon the angle made by the light source and the surface.
  - Example: Paper or wood, snow, Movie screen
- Specular reflection or regular reflection
  - When light falls on any shiny or glossy surface most of it is reflected back, such reflection is known as Specular Reflection.
  - Example: Mirror/Water







Module 5 - Shading

Lighting

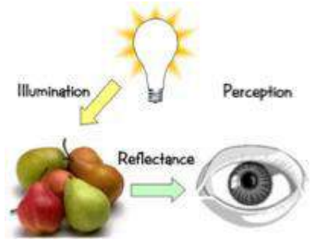
- Phong illumination model

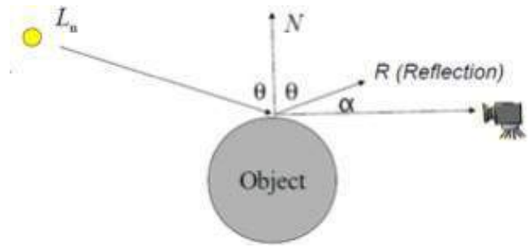
Shading

- Flat shading
- Gouraud shading
- Phong shading

Light coming into the eye

- Position of the point the eye is looking at
- Position of the light source
- Colour and intensity of light
- Vector from eye to point
- Normal vector of surface at point
- Physical properties of the object

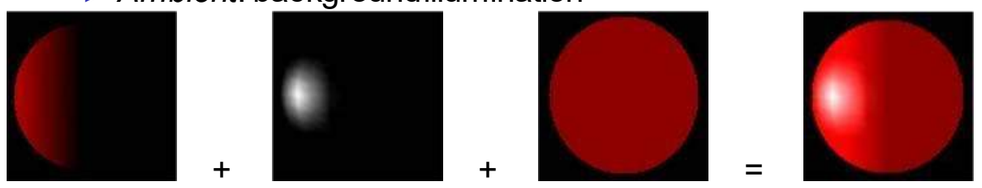




Phong illumination model

A simple 3 parameter model comprised of 3 illumination terms

- *Diffuse*: non-shiny illumination and shadows
- *Specular*: shiny reflections
- *Ambient*: background illumination



Diffuse

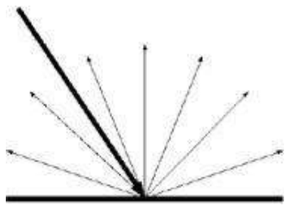
Specular

Ambient

Diffuse (Lambertian) reflection

When light hits an object with a rough surface, it is reflected in all directions

- Amount of light hitting the surface depends on the angle between the normal vector and the incident vector of the incoming light.
- The larger the angle (up to 90 degrees), the larger the area the incident light is spread over



Specular reflection

- Direct reflections of the light source off of a shiny surface
- Smooth surfaces

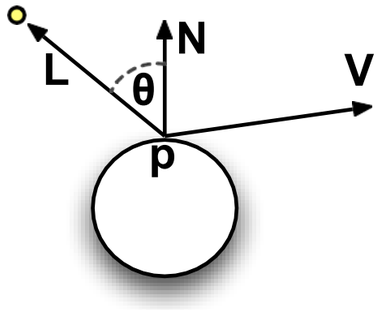
Diffuse reflection

$I = I_p k_d \cos\theta$

I_p Light intensity

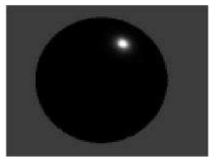
θ Angle between normal vector and direction to light source

k_d Diffuse reflectivity



Note that there is no dependence on the angle between the direction to the camera and the surface normal.

Specular reflection



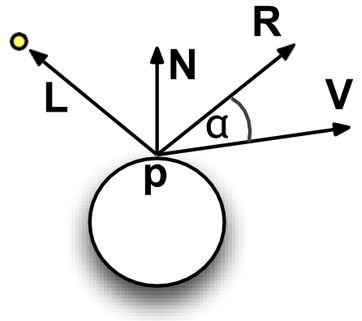
$I = I_p k_s \cos^n \alpha$

I_p Light intensity

α Angle between reflection vector and direction to camera

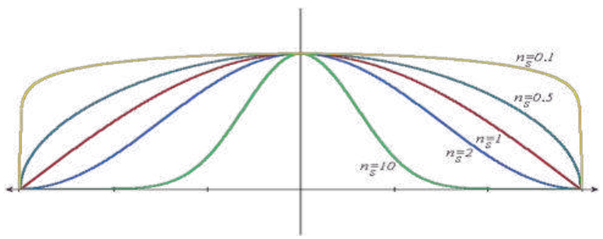
k_s Specular reflectivity

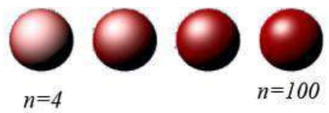
n Specular intensity

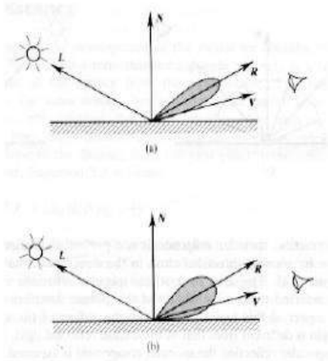


Specular reflection

- Specular light with different *n* values

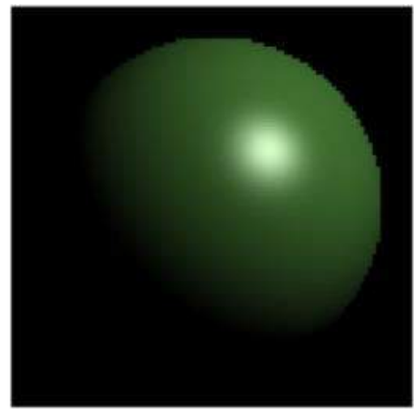




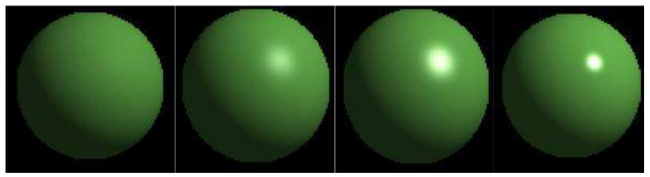


What is missing?

- Only points on the surface that are directly lit by the light are illuminated

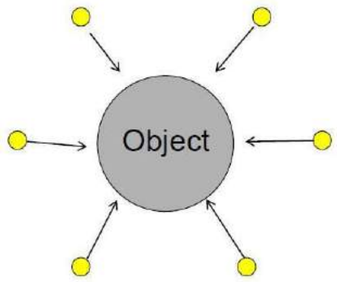


Combining diffuse and specular reflection

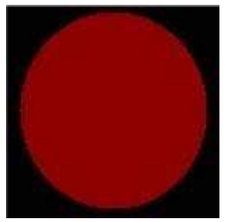


Ambient lighting

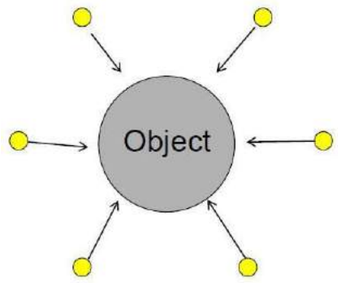
- Light reflected or scattered from other objects in the scene
- Enivornmental light
- Precise simulation of this is very hard!



Ambient lighting



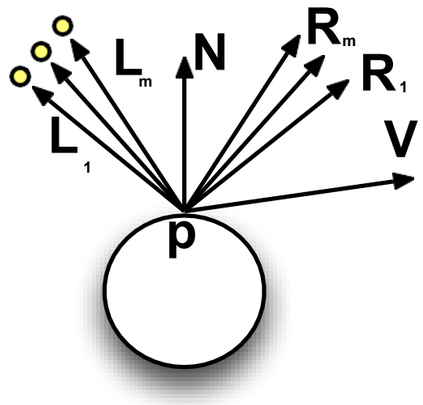
Very simple approximation:
$I = k_a I_a$



Multiple light sources

- For multiple light sources we simply compute the illumination from each source and sum them.

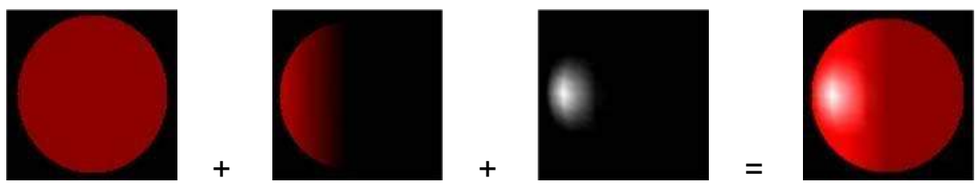
$$I = I_a k_a + \sum_{l=1}^{m} I_l (k_d \cos\theta + k_s \cos^n \alpha)$$



Combined lighting models

Combining ambient, diffuse and specular highlights gives the Phong Illumination model

$$I = I_a k_a + I_p (k_d \cos\theta + k_s \cos^n \alpha)$$



Ambient + Diffuse + Specular = I

Local illumination model

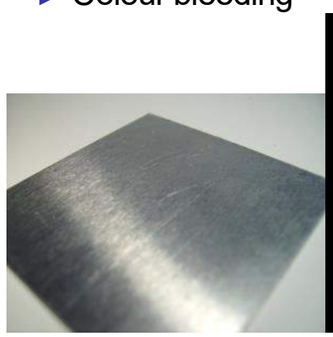
This model considers only light sources and the properties of surfaces. We don't consider light reflected from other surfaces.

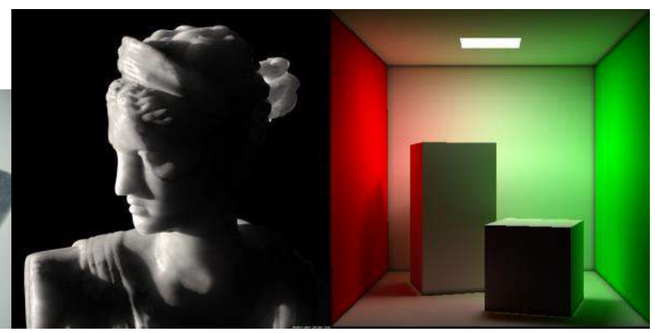
- Real time rendering
- Cost depends on number of light sources

Problems

Certain things cannot easily be rendered with this model:

- Brushed metal
- Marble (subsurface scattering)
- Colour bleeding

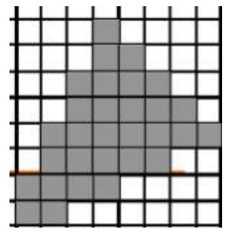




How do we colour the surface?

We know how to colour single points on the surface, but how do we colour the whole object

- Shading
- Performed during rasterisation



Overview

Lighting

- Phong illumination model

Shading

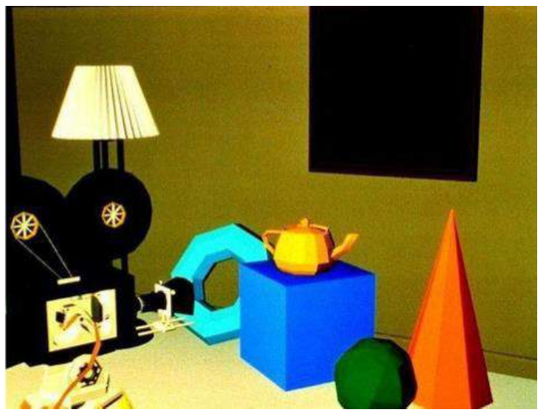
- Flat shading
- Gouraud shading
- Phong shading

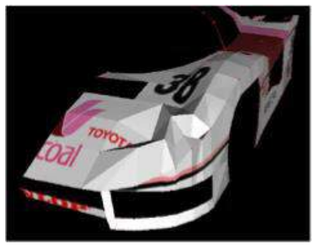
Shading models

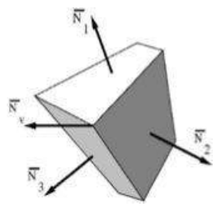
- Flat shading (one lighting calculation per polygon)
- Gouraud shading (one lighting calculation per vertex)
- Phong shading (one calculation per pixel)

Flat shading

- Colour is computed once for each face of the polygon
- All pixels in a polygon are set to the same colour
- Works for objects made of flat faces



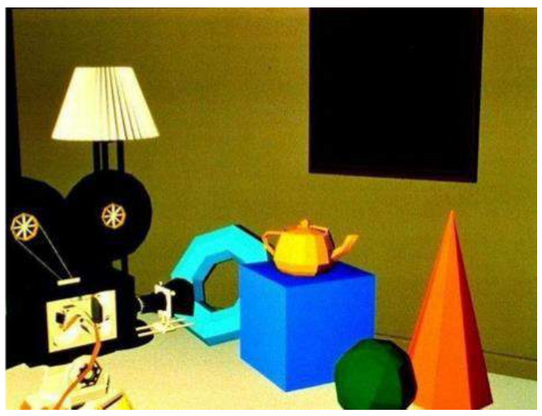


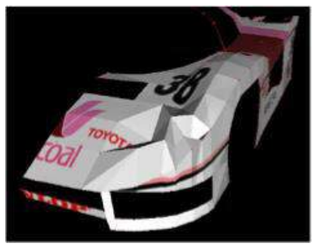


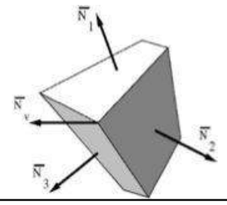
Flat shading

Suffers from an effect called Mach banding

- Eyes are sensitive to sudden changes in brighness
- Artificial changes in brightness are introduced on either side of the boundary



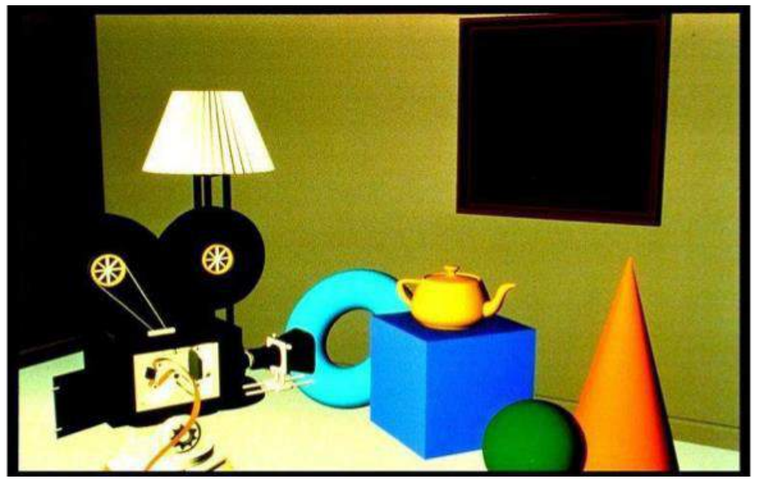




Mach band

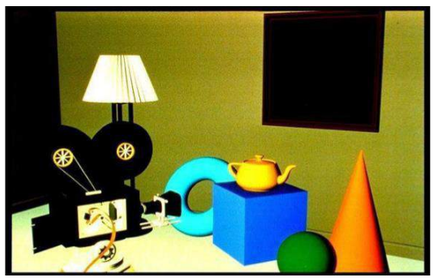
An optical illusion, discovered by Ernst Mach

Gouraud shading



Gouraud shading

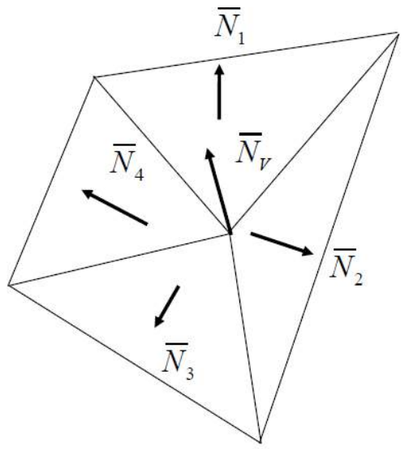
- Average Normal vector computation at every vertex.
- Colour is computed once per vertex using the local illumination model
- Polygons interpolate colours over their surface



Computing vertex normals

Vertex normals are found by averaging the face normals:

$$N_V = \frac{\sum_{i=1}^{n} \overline{N}_i}{\left\| \sum_{i=1}^{n} \overline{N}_i \right\|}$$



Interpolation of intensity

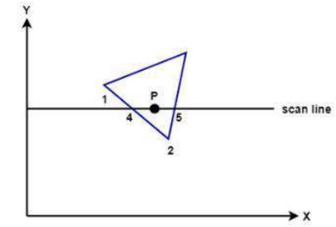


Fig: For Gouraud Shading, the intensity at point 4 is linearly interpolated from the intensities at vertices 1 and 2. The intensity at point 5 is linearly interpolated from intensities at vertices 2 and 3. An interior point P is then assigned an intensity value that is linearly interpolated from intensities at position 4 and 5.

$$I_4 = \frac{y_4 - y_2}{y_1 - y_2} I_1 + \frac{y_1 - y_4}{y_1 - y_2} I_2$$

$$I_P = \frac{x_5 - x_p}{x_5 - x_4} I_4 + \frac{x_p - x_4}{x_5 - x_4} I_5$$

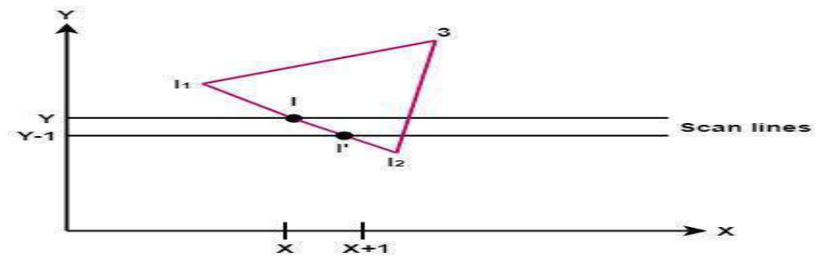


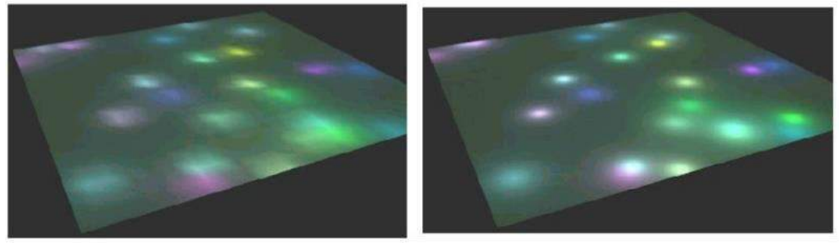
Fig: Incremental Interpolation of intensity value along a polygon edge for successive scan lines.

$$I = \frac{y - y_2}{y_1 - y_2} I_1 + \frac{y_1 - y}{y_1 - y_2} I_2$$

$$I' = I + \frac{I_2 - I_1}{y_1 - y_2}$$

Advantages:
i. It removes the intensity discontinuity which exists in constant shading model.
ii. It can be combined with hidden surface algorithm to fill in the visible polygons along each scan line.

Problems with Gouraud shading

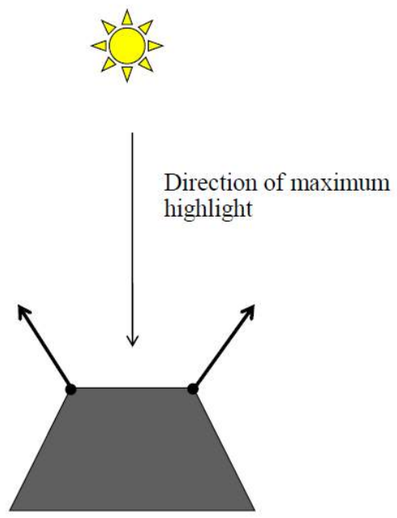


Disadvantages:
i. Gouraud shading has a problem with specular reflections.
ii. Gouraud shading can introduce anomalies known as Mach bands.

Problems with Gouraud shading

In specular reflection the highlight can be sharp, depending on the shape of $\cos^n \alpha$

- Gouraud shading interpolates linearly and so can make the highlight much bigger
- Gouraud shading can miss highlights that occur in the middle of a polygon

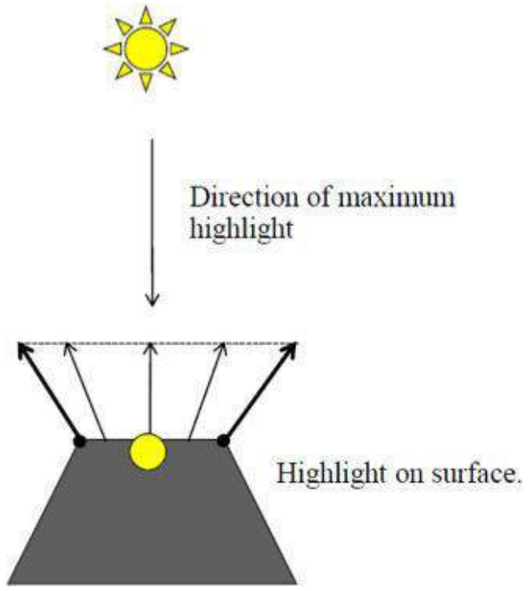


Advantages:
i. It removes the intensity discontinuity which exists in constant shading model.
ii. It can be combined with hidden surface algorithm to fill in the visible polygons along each scan line.
Disadvantages:
i. Gouraud shading has a problem with specular reflections.
ii. Gouraud shading can introduce anomalies known as Mach bands.

Phong shading

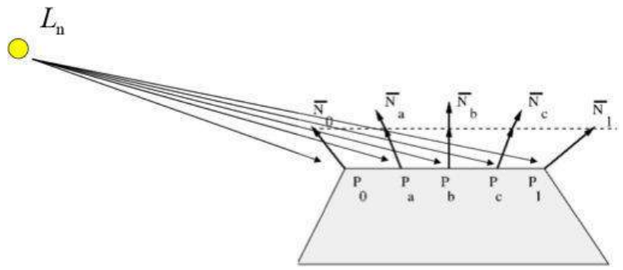
Phong shading

- Able to produce highlights that occur in the middle of a polygon



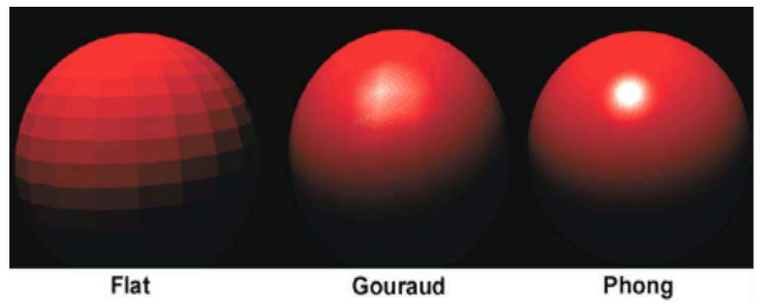
Phong shading

- Lighting computation is performed at each pixel
- Normal vectors are interpolated over the polygon



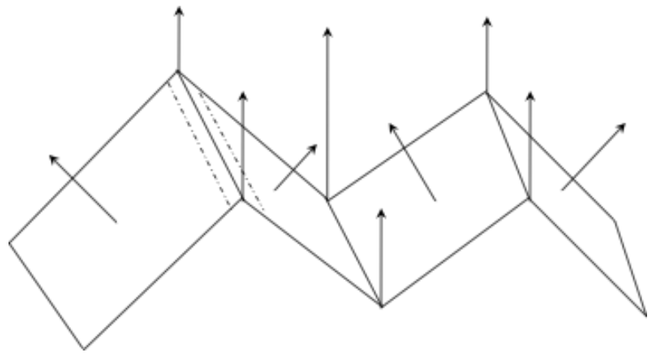
Phong shading model

Phong example



Problems with interpolation shading

Vertex normals can be incorrect when calculated as average of face normals

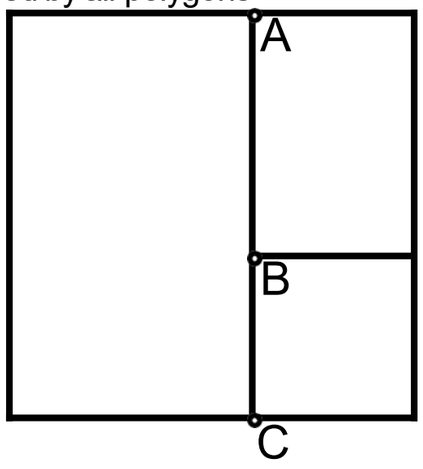


Solutions:

- Add more polygons
- Test for angles and use different vertex normals for adjacent polygons

Problems with interpolation shading

Vertices not shared by all polygons



B is not a vertex of the large polygon. Shading calculated at vertex B won't necessarily be the same as the interpolated calculations made when shading the large polygon.

Summary

Illumination

- Phong Illumination model combining ambient, diffuse and specular lighting

Shading

- Gouraud shading
- Phong shading

References

- Foley, Chapter 16 (Illumination and shading), up to 16.3
- Shirley, Chapter 10 (Surface shading)

Clipping

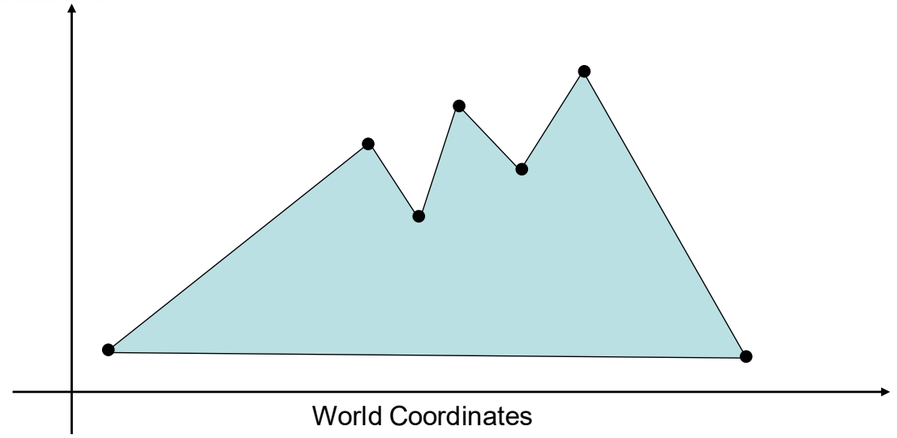
2D Clipping

1. **Introduction**
2. Point Clipping
3. Line Clipping
4. Polygon/Area Clipping
5. Text Clipping
6. Curve Clipping

2D Clipping

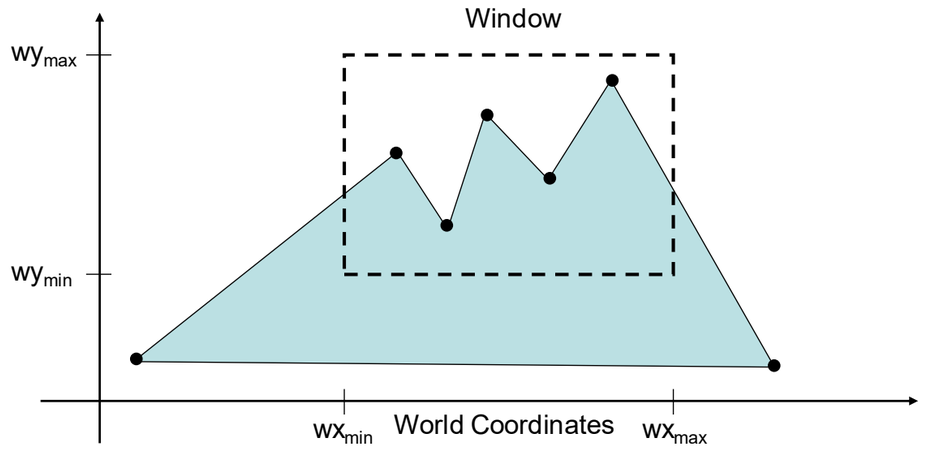
1. Introduction:

A scene is made up of a collection of objects specified in world coordinates



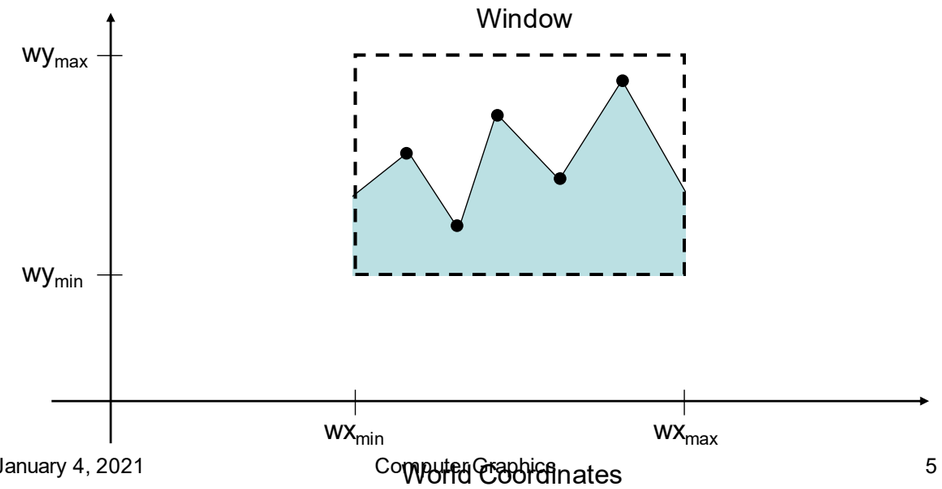
2D Clipping

When we display a scene only those objects within a particular window are displayed



2D Clipping

Because drawing things to a display takes time we *clip* everything outside the window



2D Clipping

1.1 Definition:

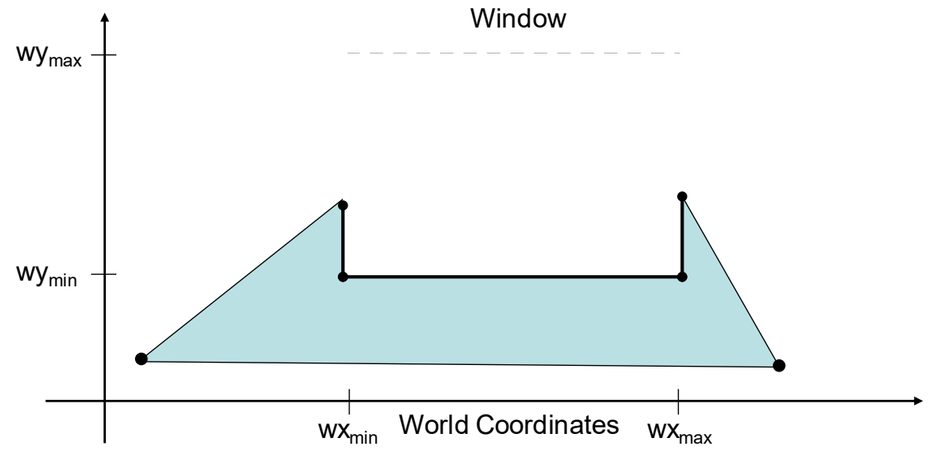
- ***Clipping*** is the process of determining which elements of the picture lie inside the window and are visible.
- By default, the "***clip window***" is the entire canvas
  - not necessary to draw outside the canvas
  - for some devices, it is damaging (plotters)
  - \
- Sometimes it is convenient to restrict the "clip window" to a smaller portion of the canvas
  - partial canvas redraw for menus, dialog boxes, other obscuration

2D Clipping

1.2 Shielding:

- ***Shielding or exterior clipping*** is the reverse operation of clipping where window act as the block used to abstract the view.
- Examples
  - A multi view window system
  - The design of page layouts in advertising or publishing applications or for adding labels or design patterns to picture.
  - Combining graphs, maps o schematics

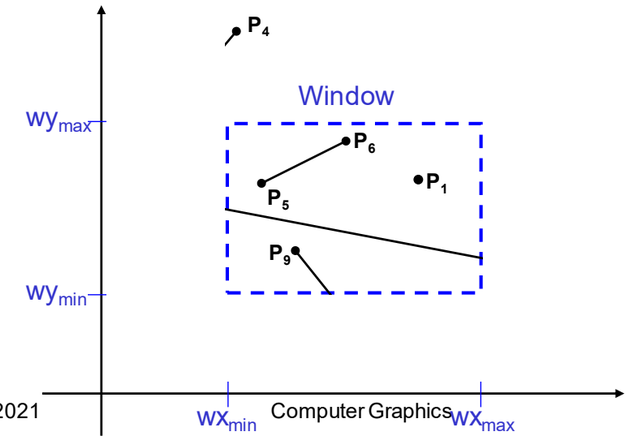
2D Clipping



2D Clipping

1.3 Example:

For the image below consider which lines and points should be kept and which ones should be clipped against the clipping window



2D Clipping

1.4 Applications:

- Extract part of a defined scene for viewing.
- Drawing operations such as erase, copy, move etc.
- Displaying multi view windows.
- Creating objects using solid modeling techniques.
- Anti-aliasing line segments or object boundaries.
- Identify visible surfaces in 3D views.

2D Clipping

b. Scissoring

- Clip it during scan conversion
- a brute force technique
  - scan convert the primitive, only write pixels if inside the clipping region
  - easy for thick and filled primitives as part of scan line fill
  - if primitive is not much larger than clip region, most pixels will fall inside
  - can be more efficient than analytical clipping.

2D Clipping

c. Raster Clipping

- Clip it after scan conversion
- render everything onto a temporary canvas and copy the clipping region
  - wasteful, but simple and easy,
  - often used for text

2D Clipping

Foley and van Dam suggest the following:

- for floating point graphics libraries, try to use analytical clipping
- for integer graphics libraries
  - analytical clipping for lines and polygons
  - others, do during scan conversion
- sometimes both analytical and raster clipping performed

2D Clipping

1.6 Levels of clipping:

- Point Clipping
- Line Clipping
- Polygon Clipping
- Area Clipping
- Text Clipping
- Curve Clipping

2D Clipping

1. Introduction
2. **Point Clipping**
3. Line Clipping
4. Polygon/Area Clipping
5. Text Clipping
6. Curve Clipping

Point Clipping

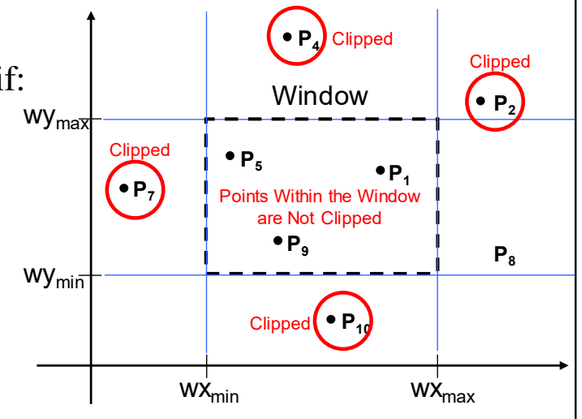
- Simple and Easy
- a point (x,y) is not clipped if:

$$wx_{min} \le x \le wx_{max}$$

$$\&$$

$$wy_{min} \le y \le wy_{max}$$

- otherwise it is clipped

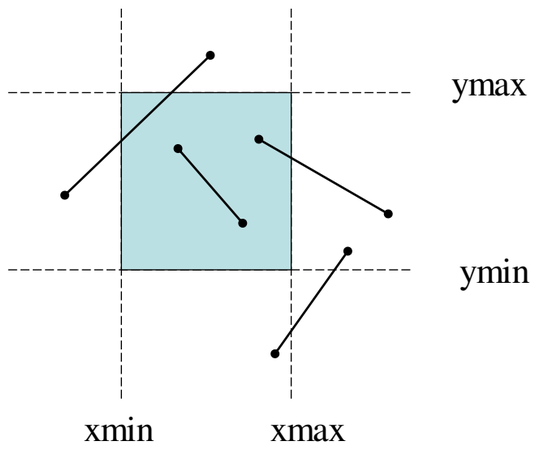


2D Clipping

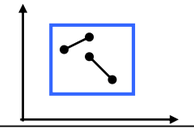
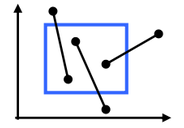
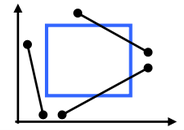
1. Introduction
2. Point Clipping
3. **Line Clipping**
4. Polygon/Area Clipping
5. Text Clipping
6. Curve Clipping

Line Clipping

- It is Harder than point clipping
- We first examine the end-points of each line to see if they are in the window or not
  - Both endpoints inside, line trivially accepted
  - One in and one out, line is partially inside
  - Both outside, might be partially inside
  - What about trivial cases?



Line Clipping

Situation	Solution	Example
Both end-points inside the window	Don't clip	
One end-point inside the window, one outside	Must clip	
Both end-points outside the window	Don't know!	

2D Line Clipping Algorithms

1. Analytical Line Clipping
2. **Cohen Sutherland Line Clipping**
3. Liang Barsky Line Clipping

Cohen-Sutherland Line Clipping

- An efficient line clipping algorithm
- The key advantage of the algorithm is that it vastly reduces the number of line intersections that must be calculated.

Cohen is something of a mystery – can anybody find out who he was?

Dr. Ivan E. Sutherland co-developed the Cohen-Sutherland clipping algorithm. Sutherland is a graphics giant and includes amongst his achievements the invention of the head mounted display.

Cohen-Sutherland Line Clipping

- Two phases Algorithm

Phase I: Identification Phase

All line segments fall into one of the following categories

1. Visible: Both endpoints lies inside
2. Invisible: Line completely lies outside
3. Clipping Candidate: A line neither in category 1 or 2

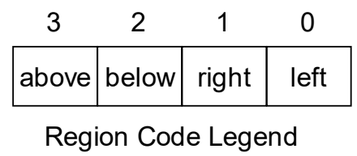
Phase II: Perform Clipping

Compute intersection for all lines that are candidate for clipping.

Cohen-Sutherland Line Clipping

Phase I: Identification Phase: World space is divided into regions based on the window boundaries

- Each region has a unique four bit region code
- Region codes indicate the position of the regions with respect to the window

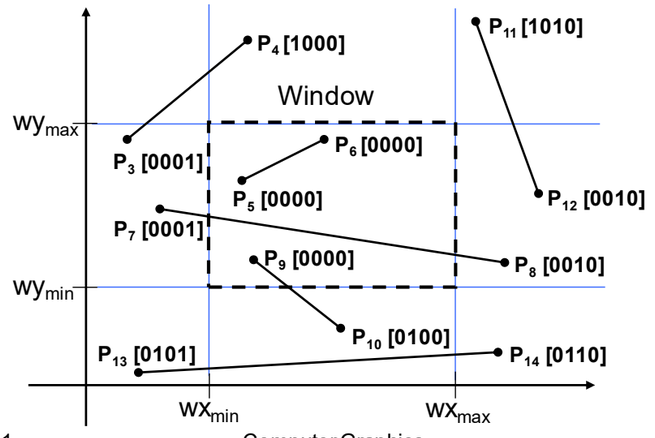


Region Code Legend

1001	1000	1010
0001	0000 Window	0010
0101	0100	0110

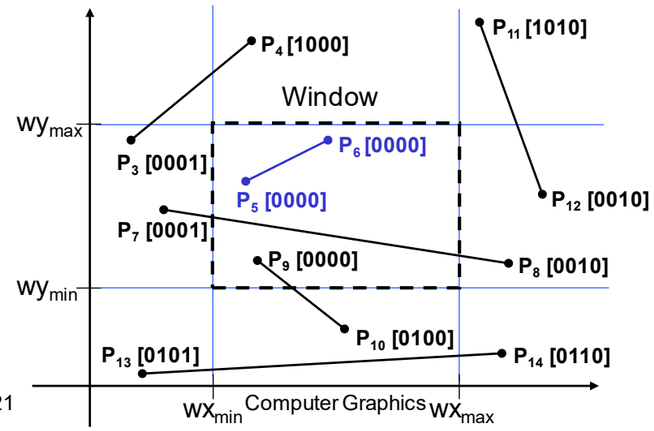
Cohen-Sutherland Line Clipping

Every end-point is labelled with the appropriate region code



Cohen-Sutherland Line Clipping

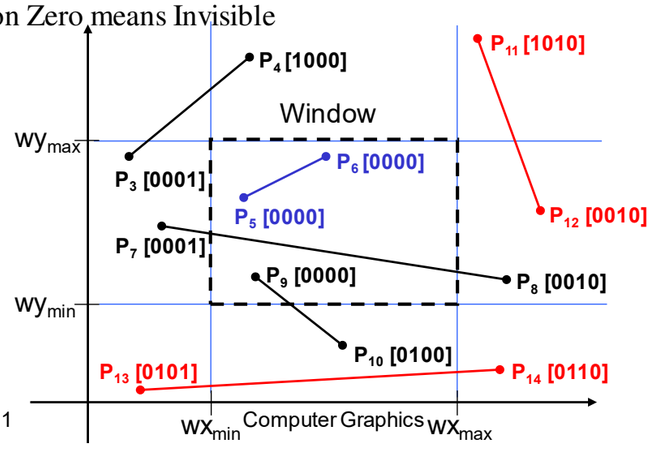
Visible Lines: Lines completely contained within the window boundaries have region code [0000] for both end-points so are not clipped



Cohen-Sutherland Line Clipping

Invisible Lines: Any line with a common set bit in the region codes of both end-points can be clipped completely

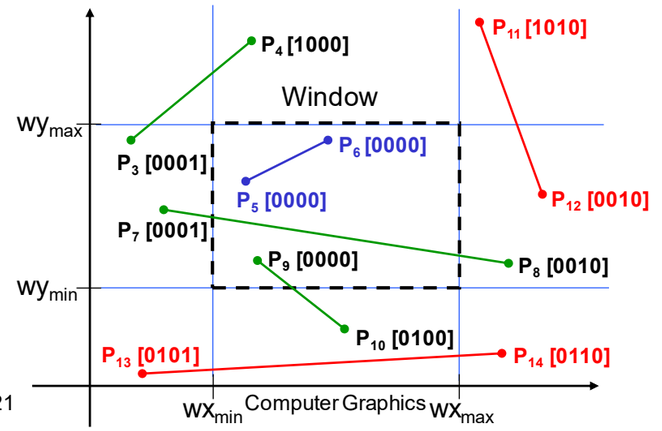
- The AND operation can efficiently check this
- Non Zero means Invisible



Cohen-Sutherland Line Clipping

Clipping Candidates: Lines that cannot be identified as completely inside or outside the window may or may not cross the window interior. These lines are processed in Phase II.

- If AND operation result in 0 the line is candidate for clipping



Cohen-Sutherland Line Clipping

Assigning Codes

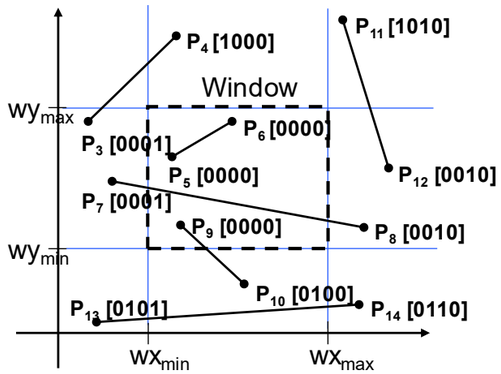
- Let point (x,y) is be given code $b_3b_2b_1b_0$:

$bit\ 3 = 1\ if\ wy_{max} - y \leq 0$

$bit\ 2 = 1\ if\ y - wy_{min} \leq 0$

$bit\ 1 = 1\ if\ wx_{max} - x \leq 0$

$bit\ 0 = 1\ if\ x - wx_{min} \leq 0$



Cohen-Sutherland Clipping Algorithm

Phase II: Clipping Phase: Lines that are in category 3 are now processed as follows:

- Compare an end-point outside the window to a boundary (choose any order in which to consider boundaries e.g. left, right, bottom, top) and determine how much can be discarded
- If the remainder of the line is entirely inside or outside the window, retain it or clip it respectively
- Otherwise, compare the remainder of the line against the other window boundaries
- Continue until the line is either discarded or a segment inside the window is found

Cohen-Sutherland Line Clipping

- Intersection points with the window boundaries are calculated using the line-equation parameters
  - Consider a line with the end-points (x_1, y_1) and (x_2, y_2)
  - The y-coordinate of an intersection with a vertical window boundary can be calculated using:

$$y = y_1 + m\,(x_{boundary} - x_1)$$

where $x_{boundary}$ can be set to either wx_{min} or wx_{max}

  - The x-coordinate of an intersection with a horizontal window boundary can be calculated using:

$$x = x_1 + (y_{boundary} - y_1) / m$$

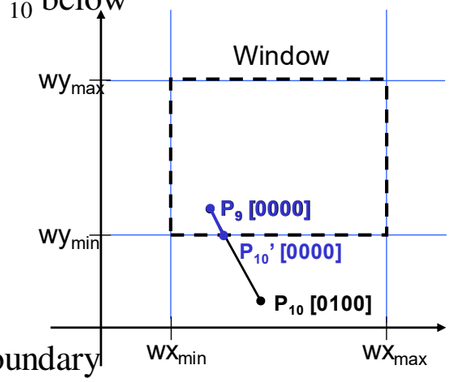
where $y_{boundary}$ can be set to either wy_{min} or wy_{max}

Cohen-Sutherland Line Clipping

- We can use the region codes to determine which window boundaries should be considered for intersection
  - To check if a line crosses a particular boundary we compare the appropriate bits in the region codes of its end-points
  - If one of these is a 1 and the other is a 0 then the line crosses the boundary.

Cohen-Sutherland Line Clipping

Example1: Consider the line P_9 to P_{10} below

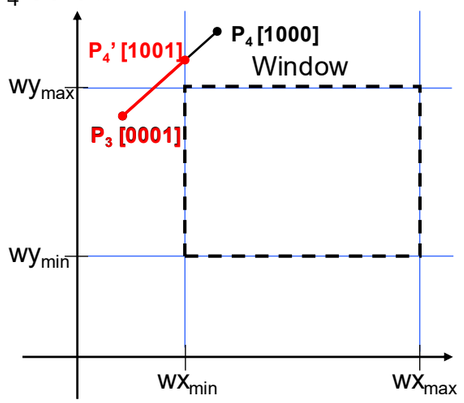


- Start at P_{10}
- From the region codes of the two end-points we know the line doesn't cross the left or right boundary
- Calculate the intersection of the line with the bottom boundary to generate point P_{10}'
- The line P_9 to P_{10}' is completely inside the window so is retained

Cohen-Sutherland Line Clipping

Example 2: Consider the line P_3 to P_4 below

- Start at P_4
- From the region codes of the two end-points we know the line crosses the left boundary so calculate the intersection point to generate P_4'

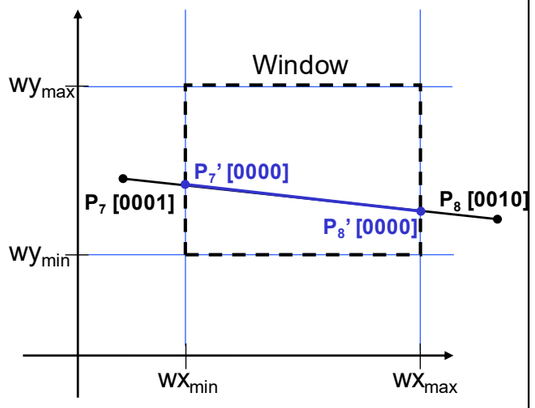


- The line P_3 to P_4' is completely outside the window so is clipped

Cohen-Sutherland Line Clipping

Example 3: Consider the line P_7 to P_8 below

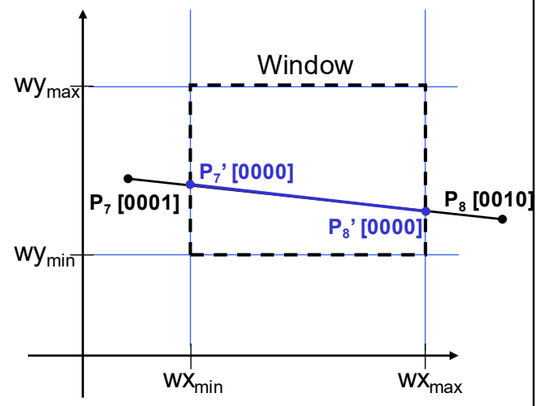
- Start at P_7
- From the two region codes of the two end-points we know the line crosses the left boundary so calculate the intersection point to generate P_7'



Cohen-Sutherland Line Clipping

Example 4: Consider the line P_7' to P_8

- Start at P_8
- Calculate the intersection with the right boundary to generate P_8'
- P_7' to P_8' is inside the window so is retained

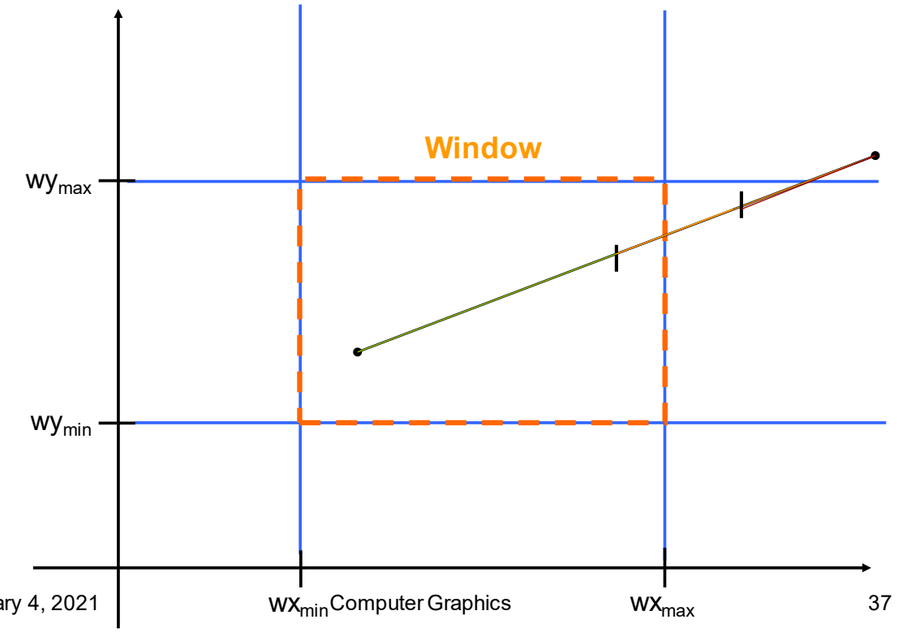


Cohen-Sutherland Line Clipping

Mid-Point Subdivision Method

- Algorithm
  1. Initialise the list of lines to all lines
  2. Classify lines as in ***Phase I***
     i. Assign 4 point bit codes to both end points $a_3a_2a_1a_0$ and $b_3b_2b_1b_0$
     ii. If ($a_3a_2a_1a_0 = b_3b_2b_1b_0 = 0$)Line in category 1
     iii. If ($a_3a_2a_1a_0$)AND ($b_3b_2b_1b_0$) # 0) Line in category 2
     iv. If ($a_3a_2a_1a_0$)AND ($b_3b_2b_1b_0$) = 0) Line in category 3
  3. Display all lines from the list in category 1 and remove;
  4. Delete all lines from the list in category 2 as they are invisible;
  5. Divide all lines of category 3 are into two smaller segments at mid-point (x_m,y_m) where $x_m = (x_1 + x_2)/2$ *and* $y_m = (y_1 + y_2)/2$
  6. Remove the original line from list and enter its two newly created segments.
  7. Repeat step 2-5 until list is null.

Cohen-Sutherland Line Clipping



Cohen-Sutherland Line Clipping

Mid-Point Subdivision Method

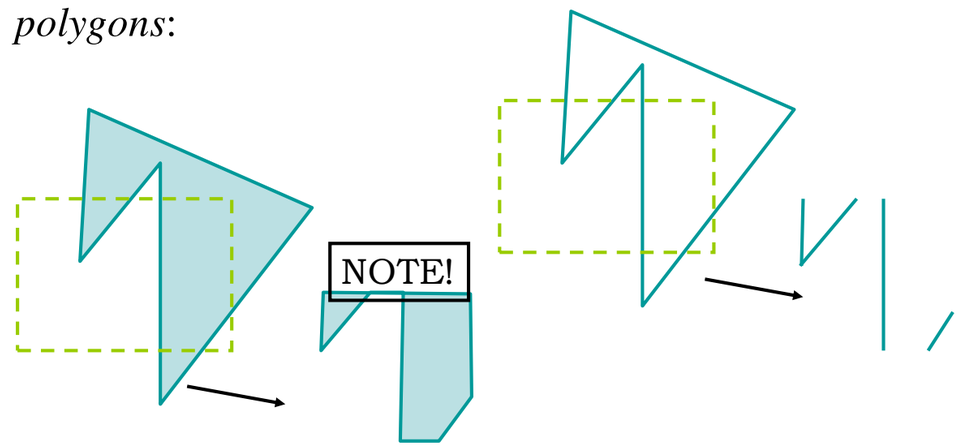
- Integer Version
- Fast as Division by 2 can be performed by simple shift right operation
- For NxN max dimension of line number of subdivisions required $\log_2 N$.
- Thus a 1024x1024 raster display require just 10 subdivisions……

2D Clipping

1. Introduction
2. Point Clipping
3. Line Clipping
4. **Polygon / Area Clipping**
5. Text Clipping
6. Curve Clipping

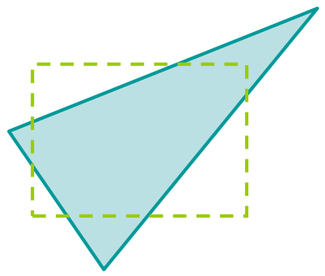
Polygon Clipping

- Note the difference between clipping *lines* and *polygons*:



Polygon Clipping

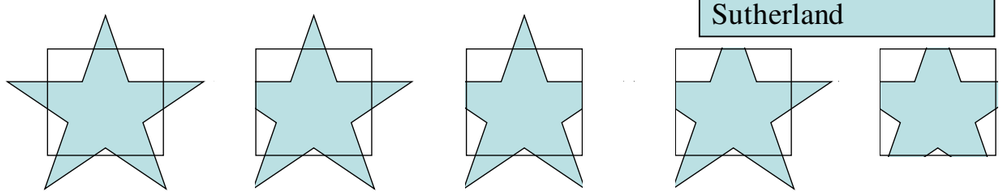
- Some difficulties:
  - Maintaining correct inside/outside
  - Variable number of vertices
  - Handle screen corners correctly



Sutherland-Hodgman Area Clipping

- A technique for clipping areas developed by Sutherland & Hodgman
- Put simply the polygon is clipped by comparing it against each boundary in turn

Sutherland turns up again. This time with Gary Hodgman with whom he worked at the first ever graphics company Evans & Sutherland



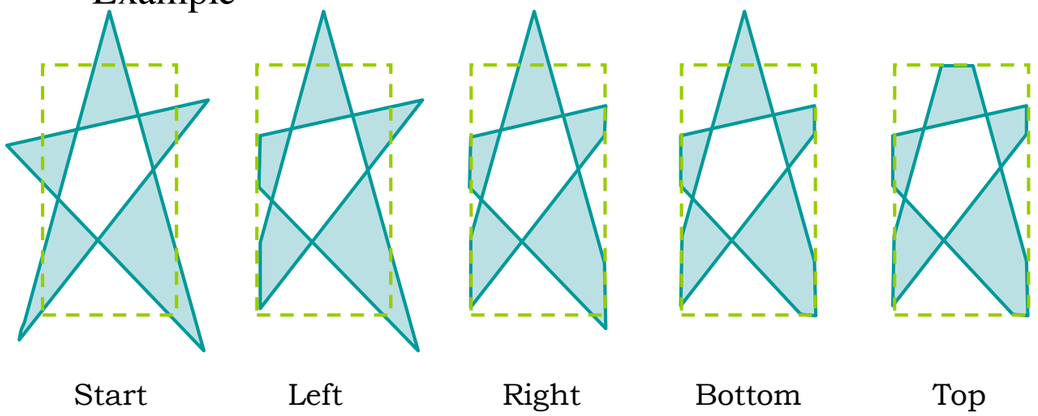
Sutherland-Hodgeman Polygon Clipping

1. Basic Concept:

- Simplify via separation
- Clip whole polygon against one edge
  - Repeat with output for other 3 edges
  - Similar for 3D
- You can create intermediate vertices that get thrown out

Sutherland-Hodgeman Polygon Clipping

- Example

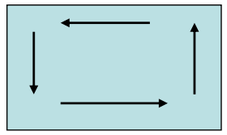


Note that the point one of the points added when clipping on the right gets removed when we clip with bottom

Sutherland-Hodgeman Polygon Clipping

2. Algorithm:

Let (P_1, P_2,…. P_N) be the vertex list of the Polygon to be clipped and E be the edge of *+vely oriented, convex clipping window*.



We clip each edge of the polygon in turn against each window edge E, forming a new polygon whose vertices are determined as follows:

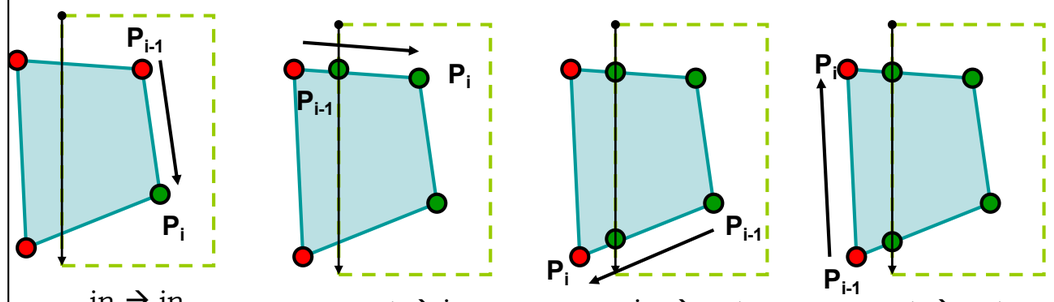
Sutherland-Hodgeman Polygon Clipping

Four cases

1. ***Inside:*** If both P_{i-1} and P_i are to the left of window edge vertex then P_i is placed on the output vertex list.
2. ***Entering:*** If P_{i-1} is to the right of window edge and P_i is to the left of window edge vertex then intersection (I) of P_{i-1} P_i with edge E and P_i are placed on the output vertex list.
3. ***Leaving:*** If P_{i-1} is to the left of window edge and P_i is to the right of window edge vertex then only intersection (I) of P_{i-1} P_i with edge E is placed on the output vertex list.
4. ***Outside:*** If both P_{i-1} and P_i are to the right of window edge nothing is placed on the output vertex list.

Sutherland-Hodgeman Polygon Clipping

Creating New Vertex List



in → in
save ending vert
Inside
(1 output)

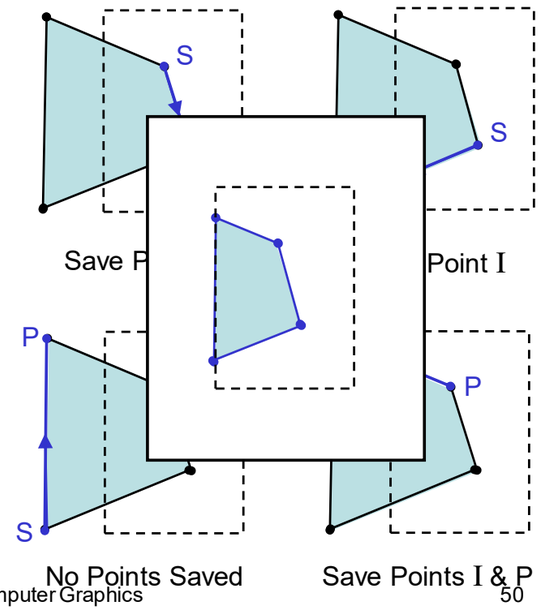
out → in
save new clip vert
and ending vert
Entering
(2 outputs)

in → out
save new clip vert
Leaving
(1 output)

out → out
save nothing
Outside
(0 output)

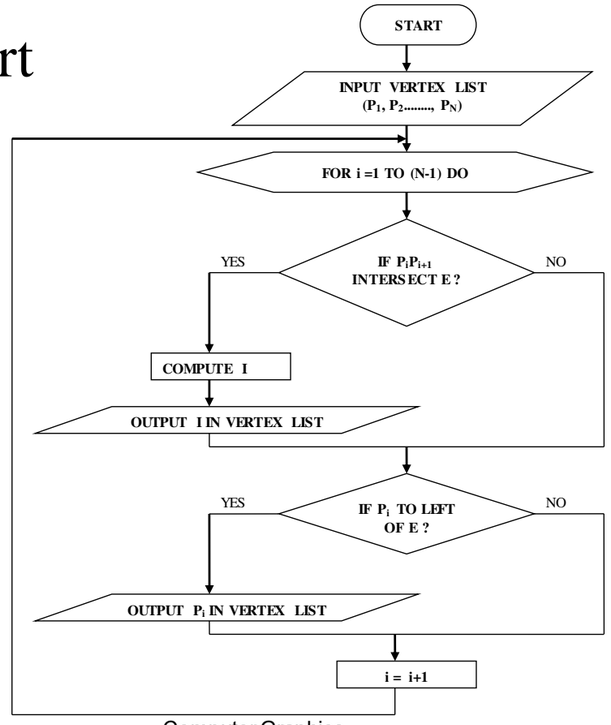
Sutherland-Hodgman Polygon Clipping

- Each example shows the point being processed (P) and the previous point (S)
- Saved points define area clipped to the boundary in question



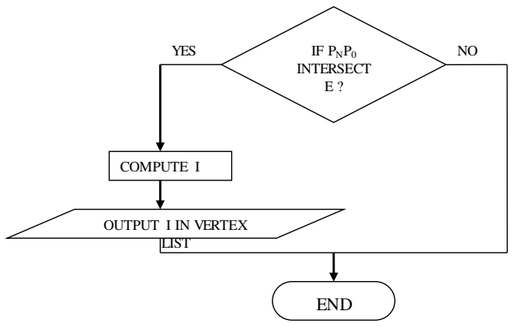
Flow Chart

Special case for first Vertex



Flow Chart

Special case for first Vertex



YOU CAN ALSO APPEND AN ADDITIONAL VERTEX $P_{N+1} = P_1$ AND AVOID SPECIAL CASE FOR FIRST VERTEX

Sutherland-Hodgeman Polygon Clipping

Inside/Outside Test:

Let $P(x,y)$ be the polygon vertex which is to be tested against edge E defined form A(x_1, y_1) to B(x_2, y_2). Point P is to be said to the left (inside) of E or AB iff

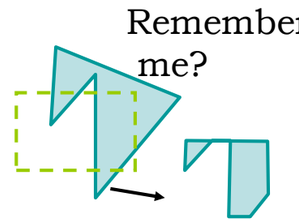
$$\frac{y - y_1}{y_2 - y_1} - \frac{x - x_1}{x_2 - x_1} > 0$$

or $C = (x_2 - x_1)(y - y_1) - (y_2 - y_1)(x - x_1) > 0$

otherwise it is said to be the right/Outside of edge E

Weiler-Atherton Polygon Clipping

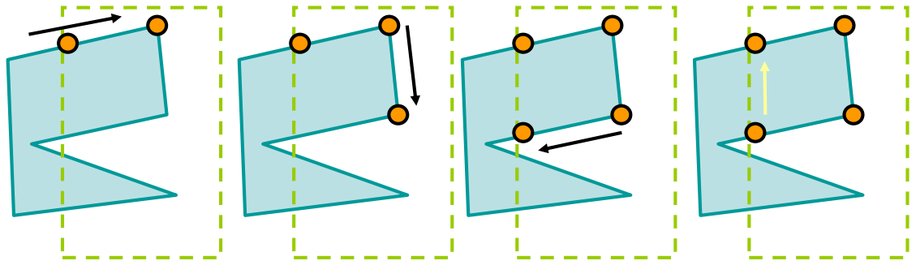
- Problem with Sutherland-Hodgeman:
  - Concavities can end up linked



- Weiler-Atherton creates separate polygons in such cases

Weiler-Atherton Polygon Clipping

- Example



add clip pt.
and end pt.

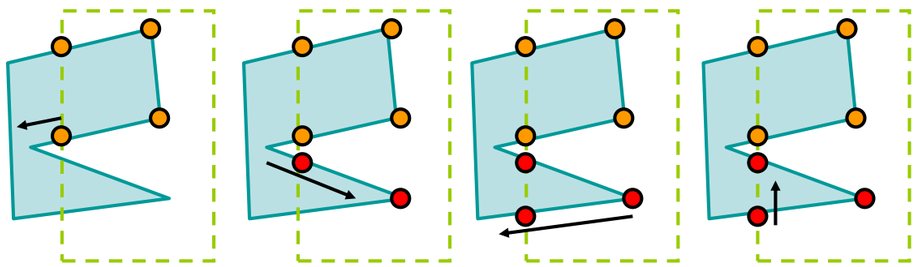
add end pt.

add clip pt.
cache old dir.

follow clip edge until
a) new crossing found
b) reach pt. already added

Weiler-Atherton Polygon Clipping

- Example (cont)



continue from
cached location

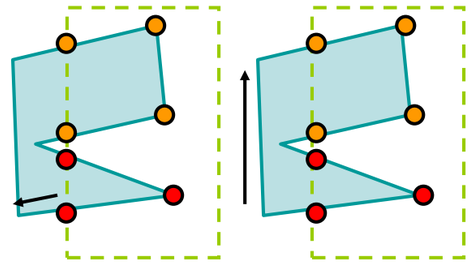
add clip pt.
and end pt.

add clip pt.
cache dir.

follow clip edge until
a) new crossing found
b) reach pt. already added

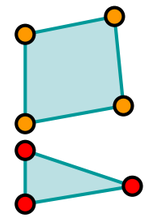
Weiler-Atherton Polygon Clipping

- Example (concluded)



continue from
cached location

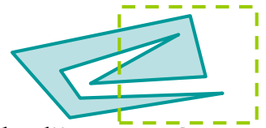
nothing added
finished



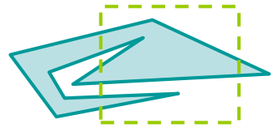
Final result:
Two *unconnected*
polygons

Weiler-Atherton Polygon Clipping

- Difficulties:
  - What if the polygon re-crosses edge?



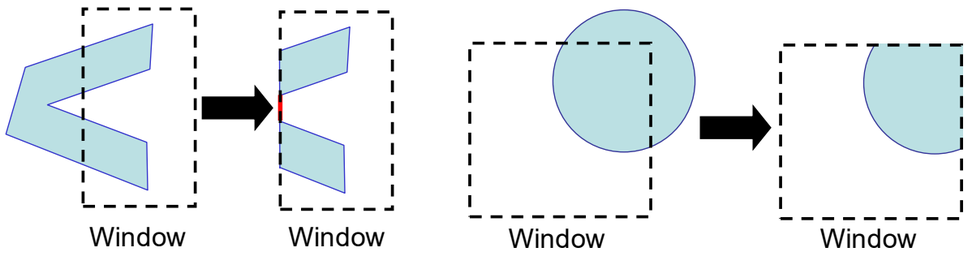
  - How many "cached" crosses?



  - Your geometry step must be able to *create* new polygons
    - Instead of 1-in-1-out

Other Area Clipping Concerns

- Clipping concave areas can be a little more tricky as often superfluous lines must be removed



- Clipping curves requires more work
  - For circles we must find the two intersection points on the window boundary

2D Clipping

1. Introduction
2. Point Clipping
3. Line Clipping
4. Polygon/Area Clipping
5. **Text Clipping**
6. Curve Clipping

Text Clipping

Text clipping relies on the concept of bounding rectangle

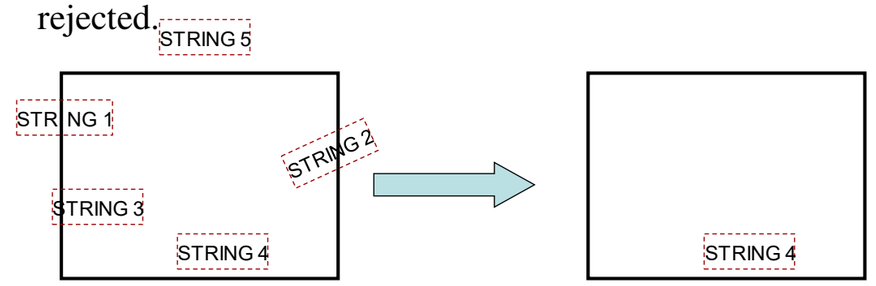
TYPES

1. All or None String Clipping
2. All or None Character Clipping
3. Component Character Clipping

Text Clipping

1. All or None String Clipping

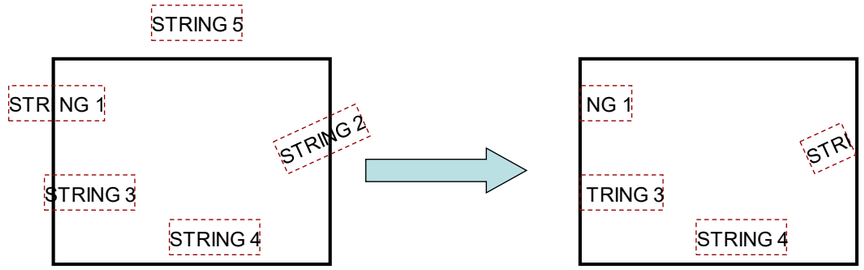
- In this scheme, if all of the string is inside window, we clip it, otherwise the string is discarded. This is the fastest method.
- The procedure is implemented by consider a ***bounding rectangle*** around the text pattern. The boundary positions are compared to the window boundaries. In case of overlapping the string is rejected.



Text Clipping

2. All or None Character Clipping

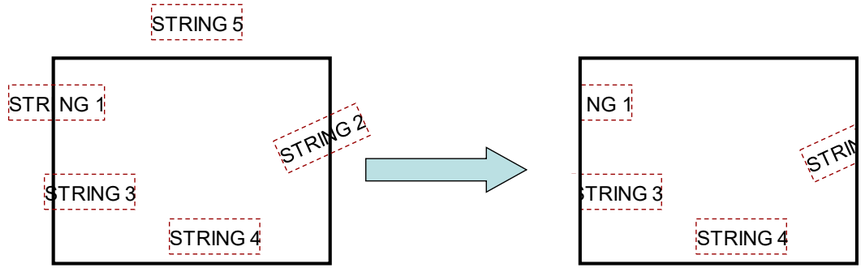
- In this scheme, we discard only those characters that are not completely inside window.
- Boundary limits of individual characters are compared against window. In case of overlapping the character is rejected.



Text Clipping

3. Component Character Clipping

- Characters are treated like graphic objects.
  - Bit Mapped Fonts : Point Clipping
  - Outlined Fonts : Line/Curve Clipping
- In case of overlapping the part of the character inside is displayed and the outside portion of the character is rejected.

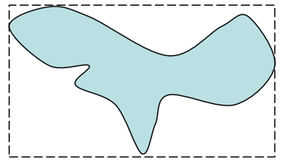


2D Clipping

1. Introduction
2. Point Clipping
3. Line Clipping
4. Polygon/Area Clipping
5. Text Clipping
6. **Curve Clipping**

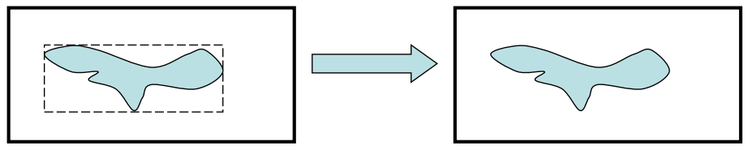
Curve Clipping

- Areas with curved boundaries can be clipped with methods similar to line and polygon clipping.
- Curve clipping requires more processing as it involve non linear equations.
- ***Bounding Rectangles*** are used to test for overlap with rectangular clip window.

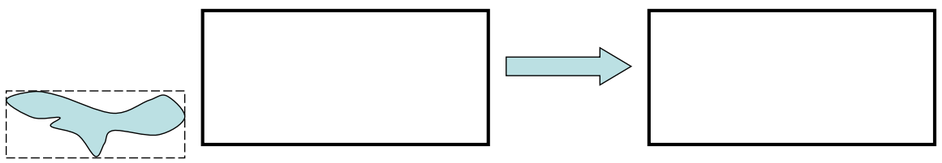


Curve Clipping

- If bounding rectangle is completely inside the object/curve is saved.

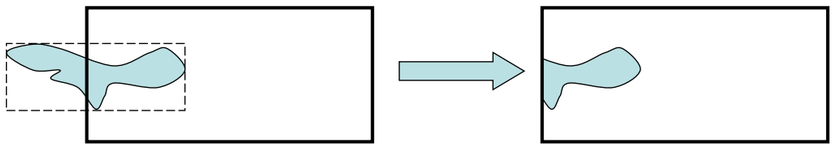


- If bounding rectangle is completely outside the object/curve is discarded.



Curve Clipping

- If both the above tests fails we use other computation saving approaches depending upon type of object
  - **Circle:** Use coordinate extent of individual quadrant, then octant if required.
  - **Ellipse:** Use coordinate extent of individual quadrant.
  - **Point:** Use point clipping



Any Question !

Module 6- Rendering

Module No. 6	Rendering	8 hours
Lighting; Radiosity; Raytracing texture mapping, compositing, textures in OpenGL; Ray Tracing- Recursive ray tracer, ray-sphere intersection ; Bezier curves and surfaces, B-splines, visualization, interpolation, marching squares algorithm.		

Rendering

Rendering or image synthesis is the process of generating a photorealistic or non-photorealistic image from a 2D or 3D model by means of a computer program. The resulting image is referred to as the render.

Outline

1. Light & Surface
2. Ray Tracing
3. Radiosity
4. Texture
5. Compositing

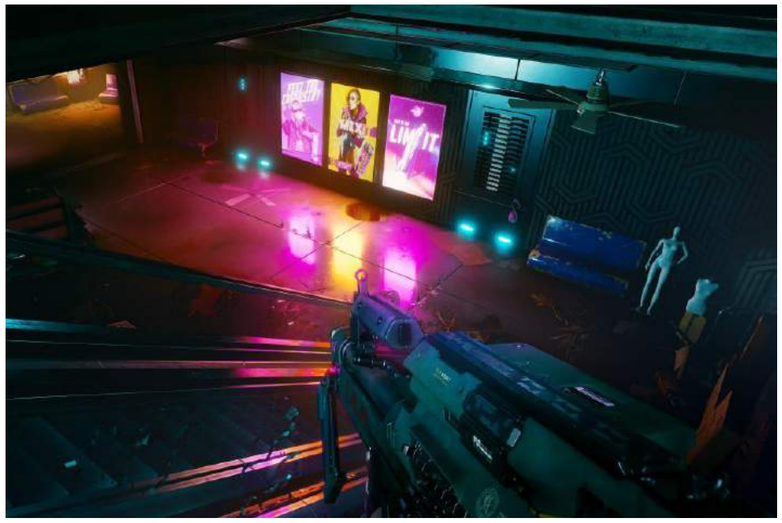
Lighting and Surface

Different types of light
- Point light(bulb/torch light)
- Parallel/directional light(sun/laser)
- Distributed/Spot light(tube light)

Different types of surface
- Ambient
- Diffuse
- Specular

Idea behind rendering

- The importance of generating realistic images from electronically stored scenes has significantly increased during the last few years.
- For this reason a number of methods have been introduced to simulate various effects which increase the realism of computer generated images.
- Among these effects are specular reflection and refraction, diffuse interreflection, spectral effects, and various others.
- They are mostly due to the interaction of light with the surfaces of various objects, and are in general very costly to simulate.



The two most popular methods for calculating realistic images are:

1. RAY TRACING.

2. RADIOSITY

Right now, only Nvidia RTX graphics cards support ray tracing. Not every PC game will support the technology either, as the developer must do some serious updating to get it working. The PC games that support ray tracing with RTX GPUs are as follows:

- Atomic Heart
- Battlefield 5
- Boundary
- Call of Duty: Modern Warfare
- Control
- Convallaria
- Cyberpunk 2077
- Deliver Us The Moon: Fortuna
- Dying Light 2
- Enlisted
- FIST,
- Justice
- JX3
- MechWarrior 5: Mercenaries
- Metro Exodus
- Minecraft
- Product DH
- Project X
- Quake II
- Shadow of the Tomb Raider
- SYNCED: Off-Planet
- Vampire: The Masquerade – Bloodlines 2
- Watch Dogs Legion
- Wolfenstein: Youngblood
- Xuan-Yuan Sword VII

Ray Tracing

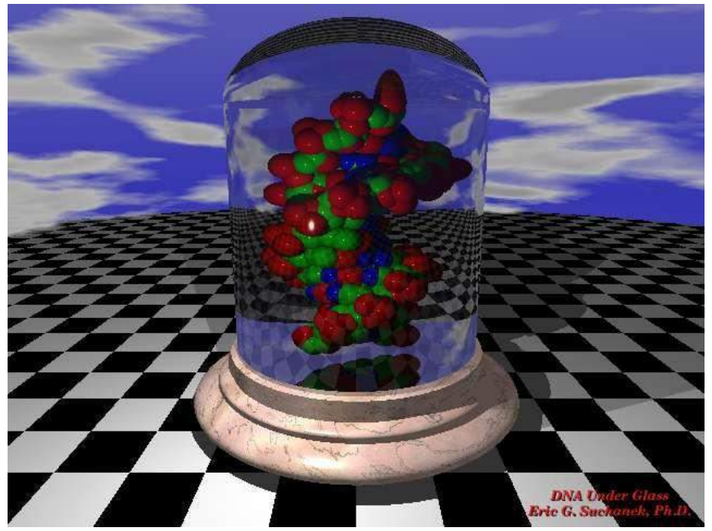
What is ray tracing

- Ray tracing is a technique for rendering three-dimensional graphics with very complex light interactions. This means you can create pictures full of mirrors, transparent surfaces, and shadows, with stunning results.
- A very simple method to both understand and implement.
- It is based on the idea that you can model reflection and refraction by recursively following the path that light takes as it bounces through an environment

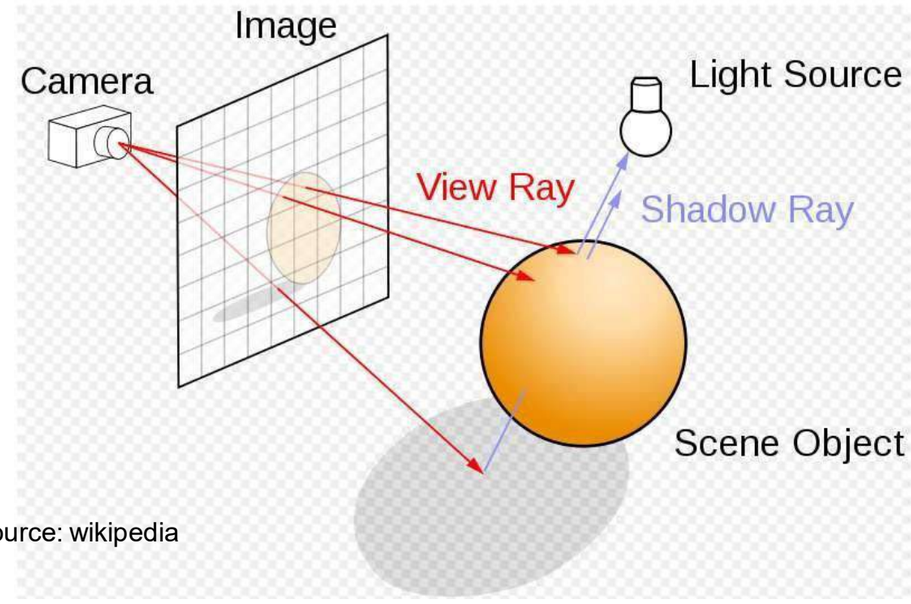
Raytraced Images

PCKTWTCH by Kevin Odhner, POV-Ray

Kettle, Mike Miller, POV-Ray

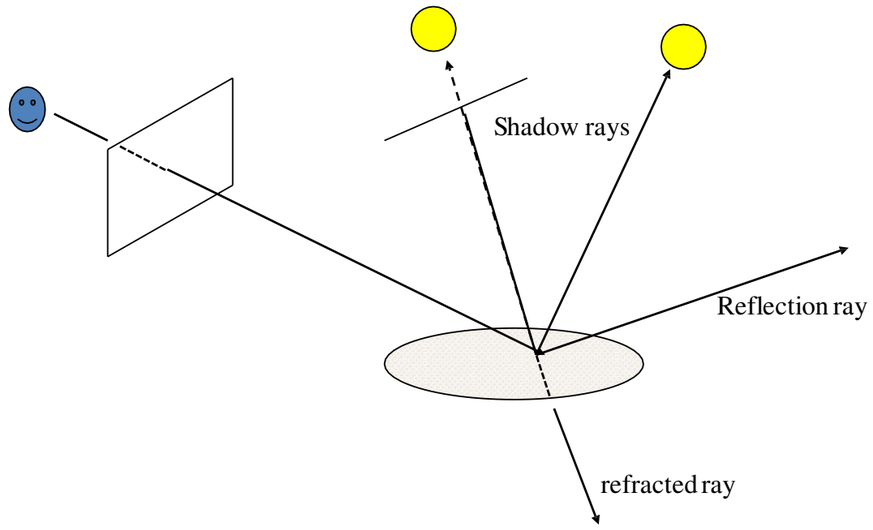


Ray Tracing Model



Source: wikipedia

Ray tracing



Ray tracing algorithm

- Builds the image pixel by pixel
- Cast additional rays from the hit point to determine the pixel color
  - Shoot rays toward each light. If they hit something, the object is shadowed from that light, otherwise use “standard model” for the light
  - Reflection rays for mirror surfaces, to see what should be reflected in the mirror
  - Refraction rays to see what can be seen through transparent objects
  - Sum all the contributions to get the pixel color

Recursive ray tracing

- When a reflected or refraction ray hits a surface, repeat the whole process from that point
  - Send more out shadow rays
  - Send out new reflected rays (if required)
  - Send out a new refracted ray (if required)
  - Generally, reduce the weight of each additional ray when computing the contributions to the surface color
  - Stop when the contribution from a ray is too small to notice or maximum recursion level has been reached

Ray tracing implementation

- Ray tracing breakdown into two tasks
  - Constructing the ray to cast
  - Intersection rays with geometry
- The former problem is simple vector arithmetic
- Intersection calculation can be done in world coordinates or model coordinates

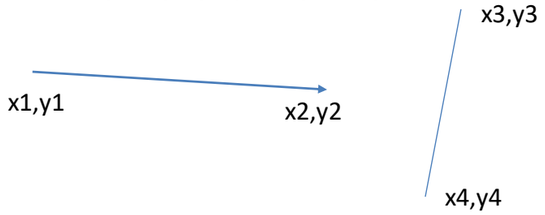
Constructing Rays

- Define rays by an initial point and a direction: $\mathbf{x}(t)=\mathbf{x}_0+t\mathbf{d}$
- Eye rays: Rays from the eye through a pixel
  - Construct using the eye location and the pixel's location on the image plane. $\mathbf{X}_0$ = **eye**
- Shadow rays: Rays from a point on a surface to the light.
  - $\mathbf{X}_0$ = point on surface
- Reflection rays: Rays from a point on a surface in the reflection direction
  - Construct using laws of reflection. $\mathbf{X}_0$ = surface point
- Transmitted rays: Rays from a point on a transparent surface through the surface
  - Construct using laws of refraction. $\mathbf{X}_0$ = surface point


```
var posx=100;
var posy=100;
var dirx=1;
var diry=0;
var x1=300;
var y1=50;
var x2=300;
var y2=200;
var px=0;
var py=0;
function setup() {
 createCanvas(400, 400);
}

function lookup(x,y){
 dirx=x-posx;
 diry=y-posx;
}
```

```
function draw() {
 background(0);
 stroke(255);
 push();
 translate(posx,posy);
 line(0,0,dirx*5,diry*5)
 pop();

 line(x1,y1,x2,y2);
 lookup(mouseX,mouseY);


 x3=posx;
 y3=posy;
 x4=posx+dirx;
 y4=posy+diry;

 t=((x1-x3)*(y3-y4)-(y1-y3)*(x3-x4))/((x1-x2)*(y3-y4)-(y1-y2)*(x3-
x4));
 u=-((x1-x2)*(y1-y3)-(y1-y2)*(x1-x3))/((x1-x2)*(y3-y4)-(y1-y2)*(x3-
x4));

 if(t>0 && t<=1 && u>0)
  {
   px=x1+t*(x2-x1)
   py=y1+t*(y2-y1)
   fill(255);
   ellipse(px,py,10,10);
  }

}
```

Ray-Line Intersections



t=((x1-x3)*(y3-y4)-(y1-y3)*(x3-x4))/((x1-x2)*(y3-y4)-(y1-y2)*(x3-x4));
u=-((x1-x2)*(y1-y3)-(y1-y2)*(x1-x3))/((x1-x2)*(y3-y4)-(y1-y2)*(x3-x4));

The intersection point can be obtained with the following

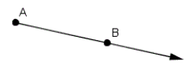
$$(P_x, P_y) = (x_1 + t(x_2 - x_1),\ y_1 + t(y_2 - y_1)) \quad \text{or} \quad (P_x, P_y) = (x_3 + u(x_4 - x_3),\ y_3 + u(y_4 - y_3))$$

Ray-Sphere Intersection

Ray

P(t)=A+tB

P(t) is the point on the ray. A is the origin of the ray. B is the direction of the ray which is **a unit vector**. t is a parameter used to move P away from A on the direction of B. Thus, P can be located just using t so we used the notation P(t) to make it look like a function.



Sphere

In analytic geometry, a sphere with center (x0,y0,z0) and radius r is the *locus* of all points (x,y,z) such that

(x−x0)2+(y−y0)2+(z−z0)2=r2.

write in the form of vector we get

$$\|P-C\|2=r2$$

which is an equivalent of
dot((P−C),(P−C))=**r2**
where P is the point on the sphere, C is sphere center point x0,y0,z0) and r is sphere radius.

Ray–Sphere Intersection
When the ray and sphere intersect, the intersection points are shared by both equations. Searching for points that are on the ray and on the sphere means combining the equations and solving for tt.

- Sphere: dot((P−C),(P−C))=r2
- Ray: p(t)=A+tB
- Combined: dot((A+tB−C),(A+tB−C))=r2

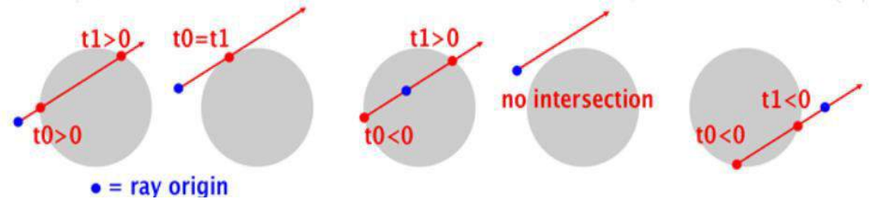
$$t2\cdot dot(B,B)+2t\cdot dot(B,A-C)+dot(A-C,A-C)-r2=0$$

Intersection Distance

If we are talking about line-sphere intersection mathematically, there are only 3 different ways:

1. No intersection.
2. One point intersection, aka tangent.
3. Two point intersection.

But ray has origin and direction, so there are more spacific scenarios:



The form of a quadratic equation is now observable:
at2+bt+c=0
where:
- a=dot(B,B)
- b=2·dot(B,A−C)
- c=dot(A−C,A−C)−r2

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

where $b^2 - 4ac$ is the **discriminant** of the equation, and

- If $discriminant < 0$, the line of the ray does not intersect the sphere (missed);
- If $discriminant = 0$, the line of the ray just touches the sphere in one point (tangent);
- If $discriminant > 0$, the line of the ray touches the sphere in two points (intersected).

Sketch.js

```
let ray1;
let sphere_center,oc;
let radius,x,y;
function setup() {
 createCanvas(400, 400);
 ray1=new Ray(100,100);
 sphere_center=createVector();
 oc=createVector();
 sphere_center.x=200;
 sphere_center.y=200;
 radius=10;
}

function draw() {
 background(255);
 ray1.show();
 ray1.lookup(mouseX,  mouseY);
 let t =hit_sphere(sphere_center,radius,ray1);
 pt=createVector();
 pt.x=ray1.origin().x+ t*ray1.direction().x;
 pt.y=ray1.origin().y+ t*ray1.direction().y;
 ellipse(pt.x,pt.y,2);

}
```

```
function hit_sphere(sphere_center, radius, ray1){

oc.x = ray1.origin().x - sphere_center.x;
oc.y = ray1.origin().y - sphere_center.y;
let a = p5.Vector.dot(ray1.direction(),  ray1.direction());
let b = 2.0 * p5.Vector.dot(oc  , ray1.direction());
let c = p5.Vector.dot(oc  , oc) - radius*radius;

let  discriminant = b*b - 4*a*c;
if(discriminant < 0){
return ;
}
else{
let numerator = -b - sqrt(discriminant);
if(numerator > 0){
return numerator / (2 * a);
}

numerator = -b + sqrt(discriminant);
if(numerator > 0) {
return numerator / (2 * a);
}
else {
return ;
}
}
}
```

ray.js

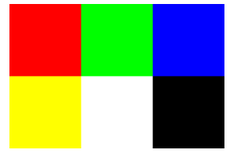
```
class Ray{
 constructor(x,y)
 {
  this.pos=createVector(x,y);
  this.dir=createVector(1,0);
  this.pos.x=x;
  this.pos.y=y;
 }
 origin()
 {
  return this.pos;
 }
 direction()
 {
  return this.dir;
 }
 show()
 {
  stroke(color(255,0,0));

line(this.pos.x,this.pos.y,this.pos.x+this.dir.x,this.pos.y+this.d
ir.y)
 }
 lookup(x,y){
 this.dir.x=x-this.pos.x;
 this.dir.y=y-this.pos.x;
}
}
```

Intersections

- It is necessary to determine the intersection between a ray with objects to decide which pixel is
- Intersection with a sphere
- Intersection with a plane
- …

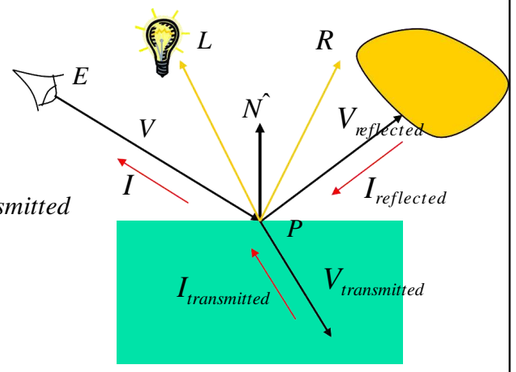
Image ray casting

```
function setup() {
 createCanvas(400, 400);
 img = loadImage('Tiny6pixel.png');

}

function draw() {
 background(220);
 image(img, 0, 0);
 for (let j = img.height-1; j >= 0; j--)
  {
    for (let i = 0; i < img.width; i++)
    {
      let r = i / img.height;
      let g = j / img.width;
      let b = 0.2;
      // Get rgb values within range (0.0, 1.0)
      // Cast to integers and map to (0, 255)
      let ir =  (255*r);
      let ig = (255*g);
      let ib =  (255*b);
      set(i,j,color(ir,ig,ib));
    }
  }
 updatePixels();
}
```



Ray Tracing Illumination

Recursive



$$I(E,V) = I_{direct} + I_{reflected} + I_{transmitted}$$

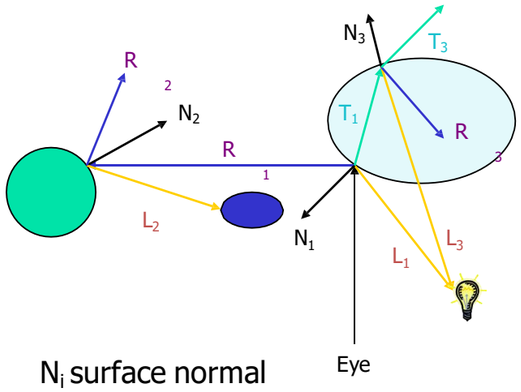
$$I_{reflected} = k_r I(P, V_{reflected})$$

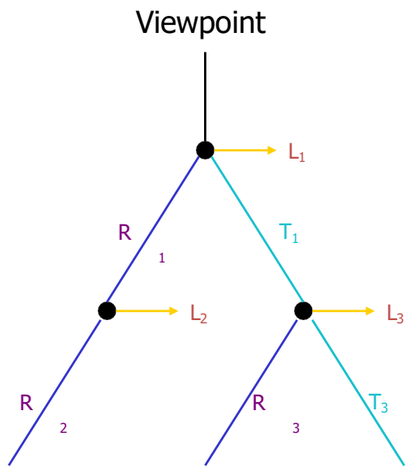
$$I_{transmitted} = k_t I(P, V_{transmitted})$$

$$I_{direct} = k_a I_{ambient} + I_{light}\left[k_d \left(\hat{N} \cdot \hat{L}\right) + k_s \left(\hat{V} \cdot \hat{R}\right)^{n_{shiny}} \right]$$

Check for shadowing (intersection with object along ray (P,L))

The Ray Tree





N_i surface normal

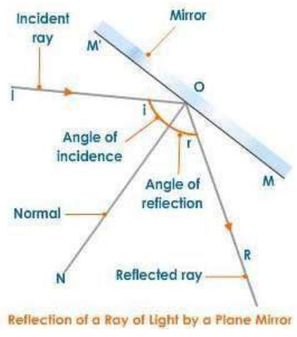
R_i reflected ray

L_i shadow ray

T_i transmitted (refracted) ray

Reflection Rays

- The laws of reflection

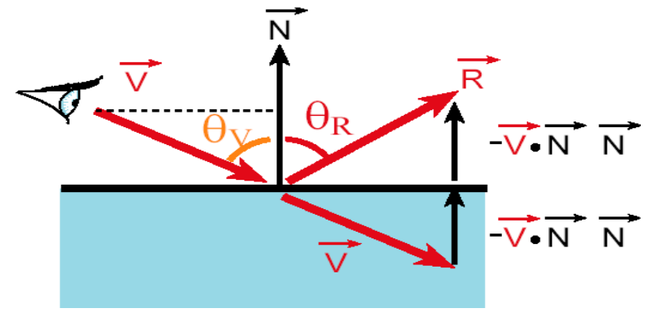


Reflection of a Ray of Light by a Plane Mirror

Reflection

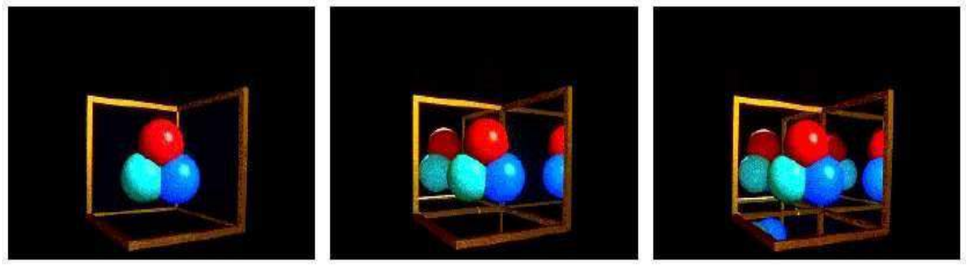
- Reflection angle = view angle

$$\vec{R} = \vec{V} - 2\left(\vec{V} \bullet \vec{N}\right)\vec{N}$$



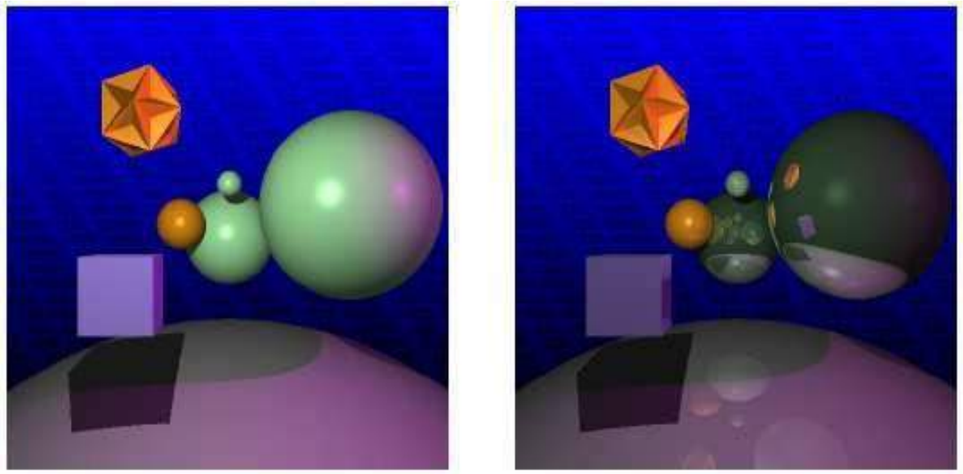
Reflection

- The maximum depth of the tree affects the handling of refraction
- If we send another reflected ray from here, when do we stop? 2 solutions (complementary)
  - Answer 1: Stop at a fixed depth.
  - Answer 2: Accumulate product of reflection coefficients and stop when this product is too small.

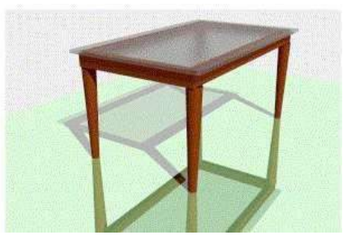


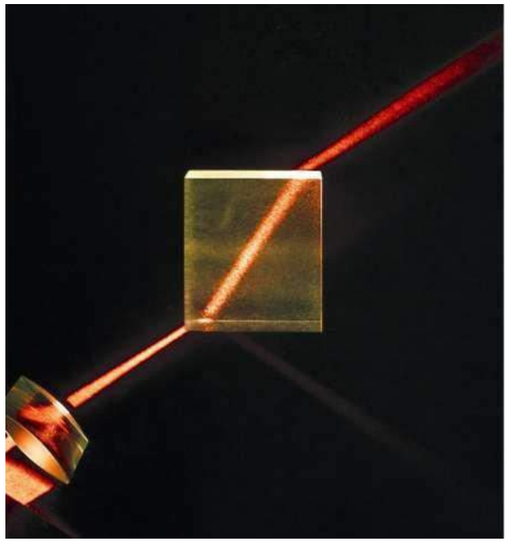
0 recursion 1 recursion 2 recursions

Reflection



Refraction





Shadow Rays

- Shadows are important lighting effect that can be easily with ray tracing
- If we wish to compute the illinumination with shadows for a point, we shoot an additional ray from the point to every light source
- A light is only allowed to contribute to the final color if the ray doesn't hit anything between the point and a light source

Pseudo Code for Ray Tracing

```
rgb lsou;        // intensity of light source
rgb back;        // background intensity
rgb ambi;        // ambient light intensity

Vector L         // vector pointing to light source
Vector N         // surface normal
Object objects [n] //list of n objects in scene
float Ks [n]     // specular reflectivity factor for each object
float Kr [n]     //  refractivity index for each object
float Kd [n]     // diffuse reflectivity factor for each object
Ray r;

void raytrace() {
   for (each pixel P of projection viewport in raster order) {
        r = ray emanating from viewer through P
        int depth = 1; // depth of ray tree consisting of multiple paths
        the pixel color at P = intensity(r, depth)
   }
}
```

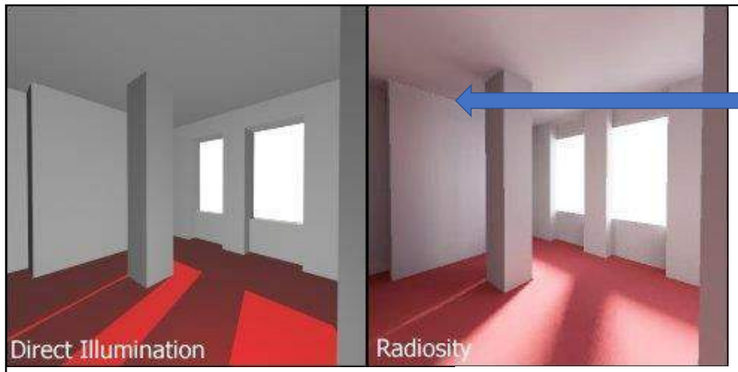
```
rgb intensity (Ray r, int depth) {
   Ray  flec, frac;
   rgb   spec, refr, dull, intensity;

    if (depth >= 5) intensity = back;
    else {
        find the closest intersection of r with all objects in scene
        if (no intersection) {
            intensity =back;
        } else {
            Take closest intersection which is object[j]
            compute normal N at the intersection point
            if (Ks[j] >0)    {  // non-zero specular reflectivity
                  compute reflection ray flec;
                  refl = Ks[j]*intensity(flec, depth+1);
            } else refl =0;
            if (Kr[j]>0)  {     // non-zero refractivity
                 compute refraction ray frac;
                 refr = Kr[j]*intensity(frac, depth+1);
            } else refr =0;
            check for shadow;
            if (shadow) direct = Kd[j]*ambi
            else direct = Phong illumination computation;
            intensity = direct + refl +refr;
      } }
     return intensity; }
```


2.Radiosity

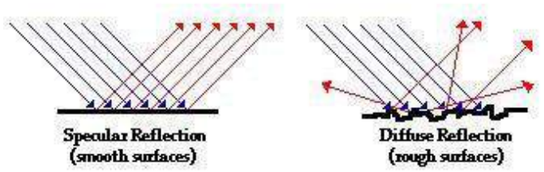
- The method of radiosity has the goal of calculating the illumination and shading in which predominate reflectors surfaces. Radiosity is an addition to render methods that increase the realism of images.
- Radiosity is a global illumination algorithm in the sense that the illumination arriving on a surface comes not just directly from the light sources, but also from other surfaces reflecting light.
- **Radiosity** is a global illumination technique that simulates the distribution of indirect light throughout an environment using a physically accurate lighting model.
- To visualize the radiosity, imagine a room with 1 white wall, one red wall and one blue wall. If we turn on a white light in the roof of this room, we can see that the white floor receives some contributions of red light from red wall and blue light from blue wall.

Thank for attention!

- Notable commercial radiosity engines are Enlighten by Geomerics (used for games including Battlefield 3 and Need for Speed: The Run); 3ds Max; form•Z; LightWave 3D and the Electric Image Animation System.



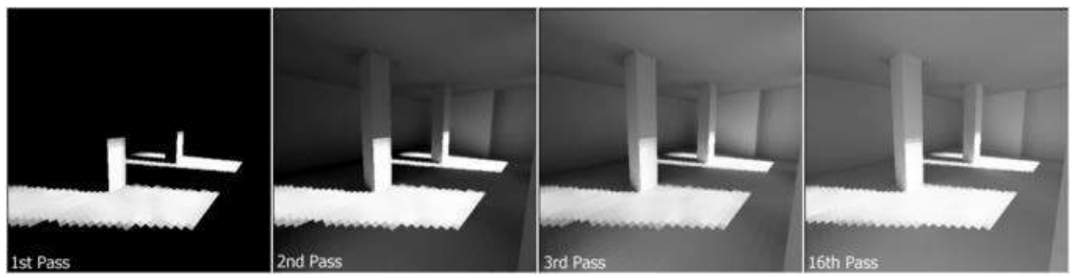
sun light is illuminated due to that white floor receives some contributions of red light from red wall .this effect is radiosity.



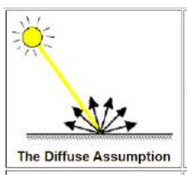
- The most interesting about radiosity, is that the human eye is very sensitive to it. Even if the radiosity isn't present, your mind will realize it immediately and will consider your image not real.

- It is important to note that there is a high cost processing and not always it will be the best solution. We can soothe this cost applying some simulations methods through the use of different kinds of lights.

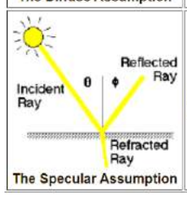
As the algorithm iterates, light can be seen to flow into the scene, as multiple bounces are computed. Individual patches are visible as squares on the walls and floor.



Recall that there are two very different ways to characterize how light interacts with surfaces in the built environment. They are *diffuse* and *specular*, as illustrated below.



Everything that we've looked at up to this point has been built around the assumption of a perfectly diffuse reflection. In this scheme incident light is reflected uniformly in all directions, regardless of the angle of incidence.



In contrast, in most science courses light is described in terms of a perfectly **specular reflection**, as illustrated in the diagram at left, where an **incident ray** arrives at the surface from somewhere, is partially absorbed (not shown), partially reflected as a **reflected ray**, and (possibly) partially transmitted, as a **refracted ray**.

Each of the algorithms that we have looked at (flat, Gouraud, Phong, etc.) is built on the assumption that light reflects diffusely off of the surfaces of the model, and that only light that moves directly from a surface to the eye (or camera) is of interest.

Radiosity for Real-Time Rendering ?

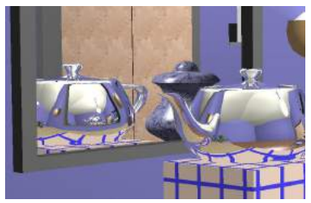
- Yes!. The current methods of radiosity are so efficient that extrapolate the needs of the human brain perception. The method used by most of 3D engines has a solution where, after the development of the scenes and positioning of fixed lights, a program is executed to calculate the total radiosity and stores the contributions on light maps (or radiosity maps).

- The result will be a variation of the illumination deposited in the object map. This method is very efficient, but it has some limitations when working with dynamic lights or distant objects.

Ray tracing vs Monte Carlo ray tracing vs Radiosity

- Ray tracing:
  - **Ray tracing** is a rendering technique for generating an image by **tracing** the **path of light** as **pixels** in an **image plane.**
  - Models lighting in scenes only for specular or partially specular surfaces.
- Ray tracing (Monte Carlo):
  - Models lighting in scenes, partially for specular and diffuse surfaces.
  - **Path tracing** is a computer graphics **Monte Carlo** method of rendering images of three-dimensional scenes such that the global illumination is faithful to reality. Fundamentally, the algorithm is integrating over all the illuminance arriving to a single point on the surface of an object.
  - An enhanced ray tracing technique
  - When hitting a diffuse surface, pick one ray at random and find the color of the incoming light.
  - Traces many paths per pixel (100 -10000 per pixel)
- Radiosity:
  - Models lighting in scenes for diffusely reflecting surfaces only
  - gives a realistic rendering of shadows and diffuse light.

Global illumination

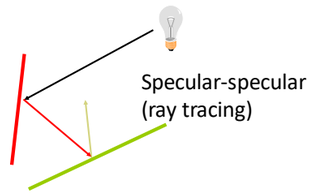


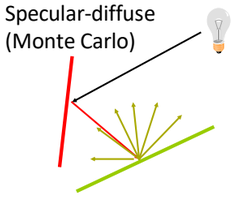
(a) Point light source (ray tracing), global illumination for specular surfaces. (+ textures)

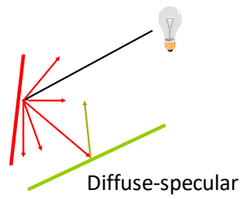
(b) global illumination FOR diffuse surfaces (radiosity)...

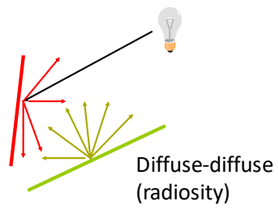
(c) … showing colour bleeding (+ textures)

Types of Surface Reflectance







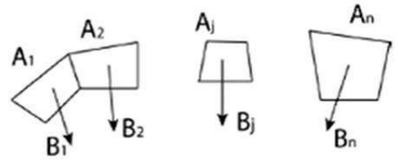


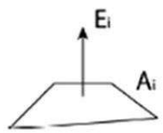
If we denote

- the radiosity of patch A_i with B_i,
- the energy it emits by E_i,
- and its reflectivity by ρ_i

we can write:

$$B_i = E_i + \rho_i \sum_j B_j F_{ij}$$





F_{ij} is dimensionless and called the form factor from A_i to A_j.

Notice that there's symmetry in form factors:

$$F_{ij} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{\cos\theta_i \, \cos\theta_j}{\pi r^2} dA_j \, dA_i$$

and

$$F_{ji} = \frac{1}{A_j} \int_{A_j} \int_{A_i} \frac{\cos\theta_j \, \cos\theta_i}{\pi r^2} dA_i \, dA_j$$

Hence, we have

$$A_i F_{ij} = A_j F_{ji}$$

and our Radiosity equation can also be written as

$$B_i = E_i + \rho_i \sum_j B_j F_{ji} \frac{A_j}{A_i}$$

Form factors specify the fraction of the energy leaving one patch that arrives at the other one.

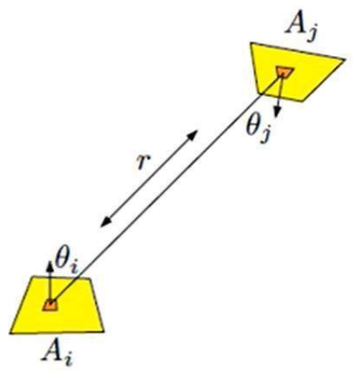
They depend on

- the shapes of patches A_i and A_j
- their distance
- their orientation

(not all energy leaving A_j reaches A_i)

They are given by

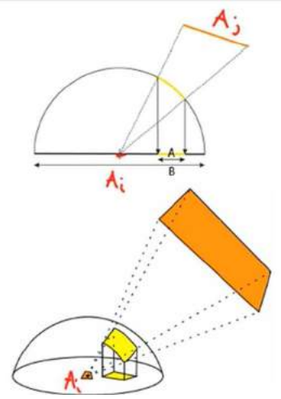
$$F_{ij} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{\cos\theta_i \, \cos\theta_j}{\pi r^2} dA_j \, dA_i$$



Hemisphere Method

Nusselt showed that computing F_{ij} for a differential patch dA_i is equivalent to

- projecting A_j onto a unit hemisphere centered about dA_i
- projecting the projected area orthographically onto the hemisphere's unit base circle
- and dividing by the area of the circle

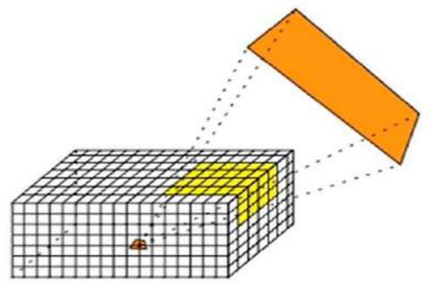


Hemi cube method

Instead of projecting analytically onto a hemisphere, we can also project onto a hemicube that is subdivided into cells, each cell having a weight that represents its contribution to the form factor.

By summing the weights of the cells onto which A_j is projected, we approximate the form factor.

This can be implemented on standard graphics hardware.



Ray tracing (RT) vs Radiosity

- In RT, rays are send through each image pixel and followed as they reflect off objects in scene
  - Mostly specular surfaces
  - View dependent
- Radiosity is based on exchange of light between object surfaces
  - Diffuse surfaces
  - View independent
  - Used for rendering soft gradual shadows.

Radiosity computation.

For each patch, we have to compute $B_i = E_i + \rho_i \sum_i B_j F_{ji}$.

We approximate this iteratively:

- Initially, set $B_i = E_i$ for every patch.
- For every patch A_i compute the energy reaching it from all other patches and add this (multiplied by ρ_i) to the radiosity of patch A_i that has been computed so far.
- Repeat the previous step, but only account for the unshot radiosity that was added to each patch A_j in the previous iteration.
- Repeat until the added radiosity per iteration is less than a threshold for each patch.

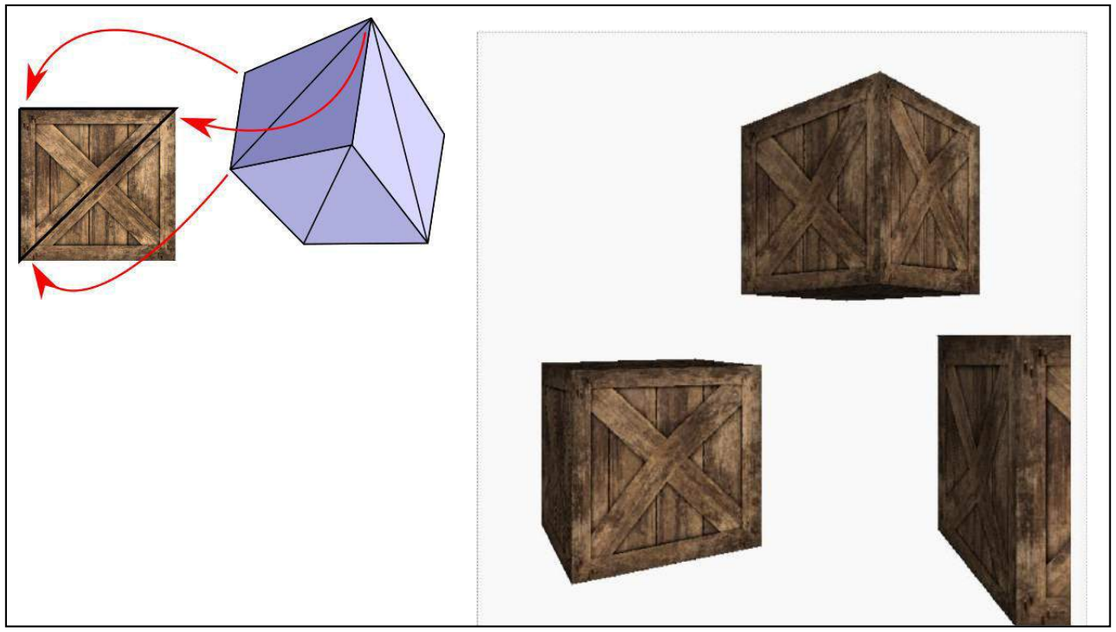
3. Texture

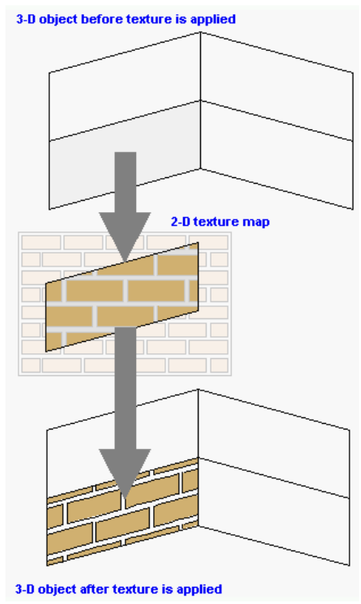
- Texture: A detailed pattern that is repeated many times to tile the plane.
- Add more visual detail to surfaces in a 3d/2d model/object/scenes
- A texture can be uniform, such as a brick wall, or irregular, such as wood grain or marble.
- An alternate method is to compute the texture entirely via mathematics instead of bitmaps.

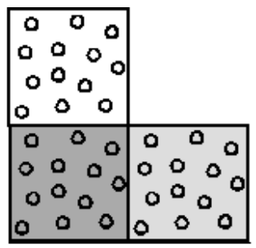
- So far we have talked about using polygons and lights to generate the look of all the objects in the scene. When fine detail is needed this may not be the most efficient way.

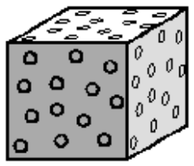
Texture Mapping

- **Texture mapping** is a method for defining surface texture, or color information on a computer-generated graphic or 3D model.
- Texture mapping originally referred to **diffuse mapping**, a method that simply mapped pixels from a texture to a 3D surface ("wrapping" the image around the object).
- There are various other technique (controlled by a materials system) have made it possible to simulate near-photorealism in real time by vastly reducing the number of polygons and lighting calculations needed to construct a realistic and functional 3D scene.









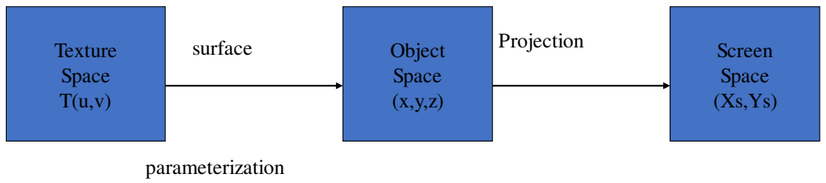
Mapping a 2-D surface onto a 3-D rendition alters the sizes and shapes of the image elements

- Instead of filling a polygon with a colour in the scan conversion process we fill the pixels of the polygon with the pixels of the texture (texels.)
- Used to:
  - add ‘detail’
  - add 'roughness'
  - add ‘patterns’

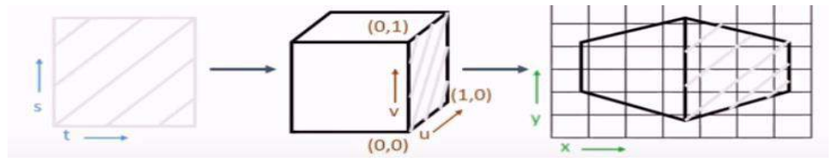
texture space -> object space -> image space

Texture Mapping

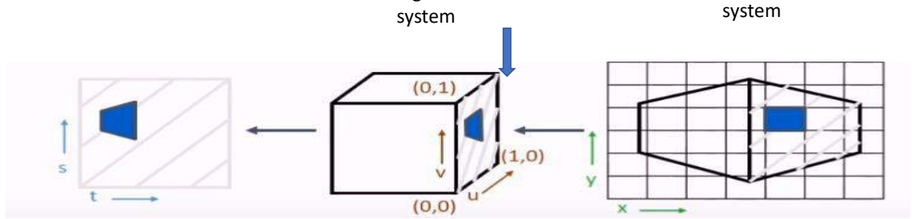
- Texture mapping is the process of taking a 2D image and mapping onto a polygon in the scene. This texture acts like a painting, adding 2D detail to the 2D polygon.
  - Define texture(add ‘detail’)
  - Specifying mapping from texture to surface(add 'roughness’)
  - Lookup texture values during scan conversion(add ‘patterns’)



When it comes to 2d projection

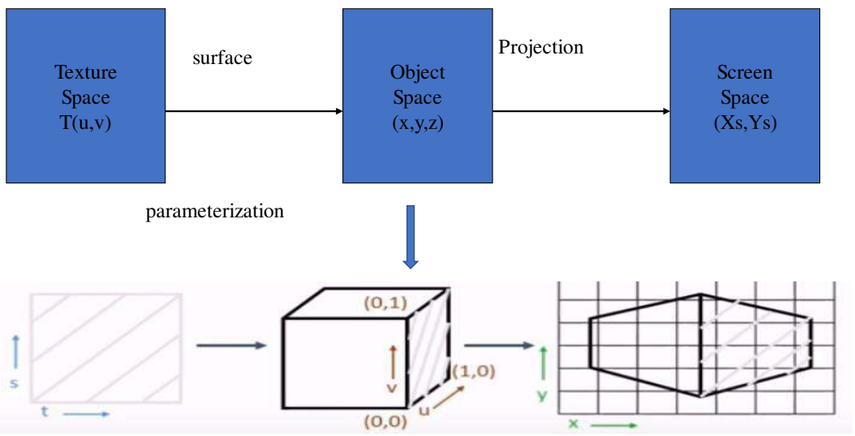


Texture | Modelling coordinate system | Screen coordinate system



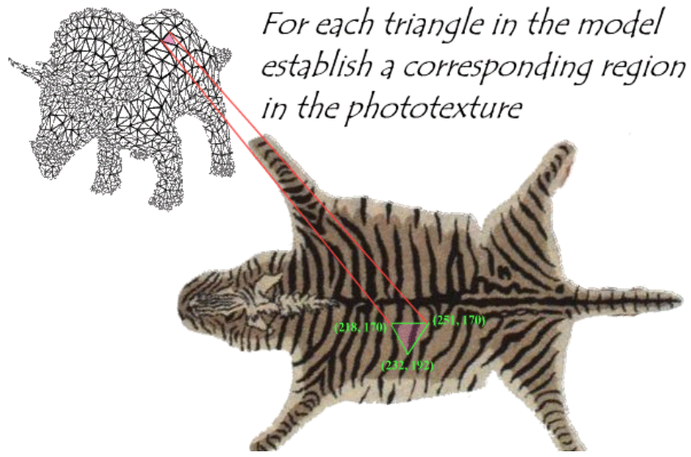
Texture image | Modelling coordinate system | Screen coordinate system

Texture mapping

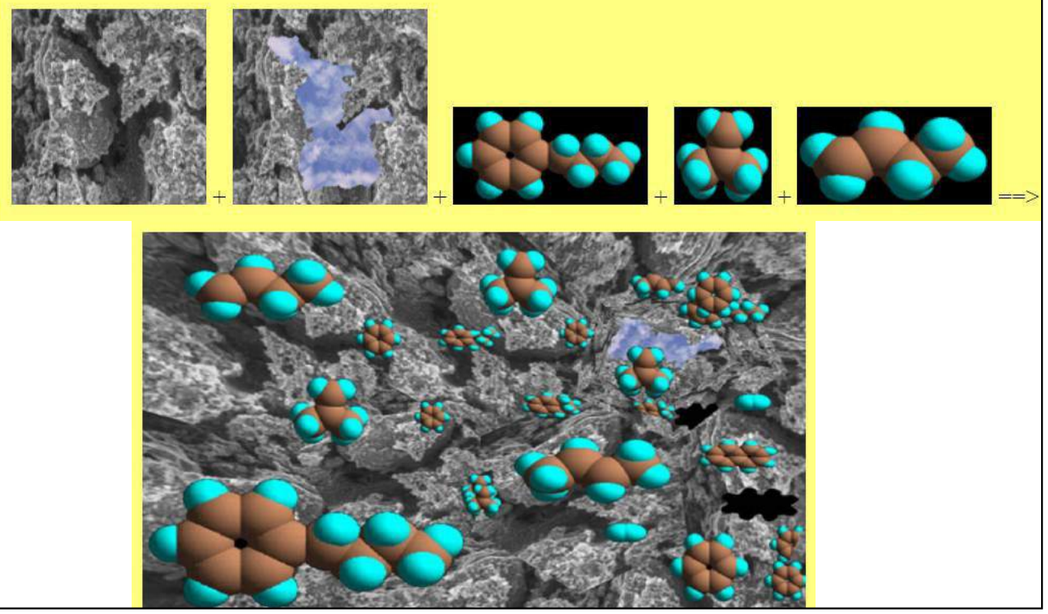


Texture | Modelling coordinate system | Screen coordinate system

Texture mapping - example



Ex

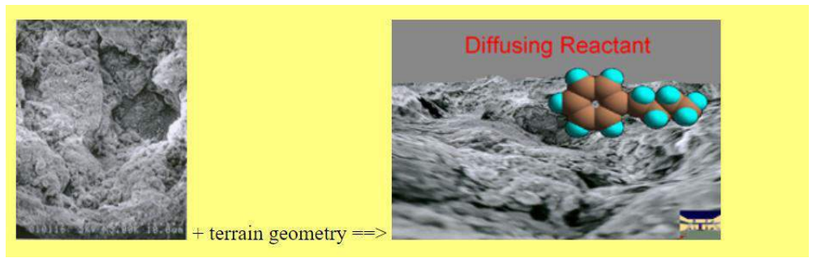


Important texture mapping techniques

1. Bump mapping
2. Displacement mapping
3. Environmental mapping
4. Photorealistic mapping (Image based rendering)
5. Non- photorealistic mapping

terrain mapping

- Yet another technique commonly used with texture maps is to "drape" them, or have a single texture map stretch over multiple polygons.
- Because this technique is often used with terrains, it is also commonly known as "terrain mapping":



1.Bump mapping or Normal mapping

- **Bump mapping** is a technique in computer graphics for simulating **bumps** and wrinkles on the surface of an object. This is achieved by disturbing the surface normal of the object

2.Displacement mapping

- Displacement mapping is a technique for adding geometric detail to surfaces at render time.
- Displacement mapping modifies the surface itself which leads to correct outline, shadow and graphics interface.

- There are other image map types for adding details and relief to your 3D art: bump and displacement maps.
- These both use black and white imagery to create relief data for a model, making bump and displacement imagery far easier to create and manipulate in any 2D painting application when compared to the complex three-colour arrangement of normal maps.

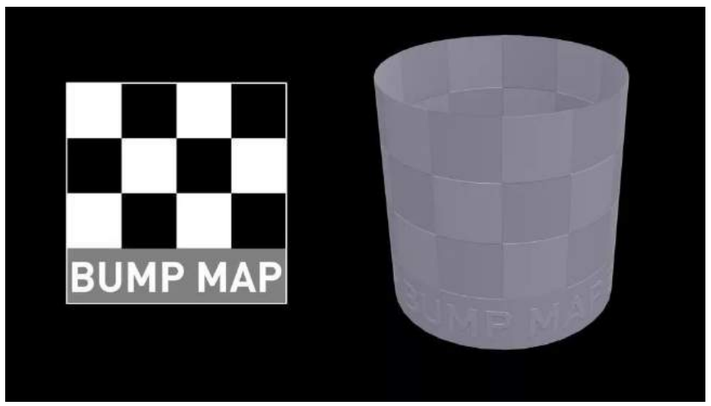
How does bump mapping work?

- Bump maps are one of the oldest form of image map types (normal maps are derived from bump maps), and have been used for decades to add surface relief to models.
- Bump maps are not very resource-intensive, making them a popular choice for a wide range of relief work.
- The catch with bump maps is that they cannot render corner or edge detail, which makes them problematic in certain situations, for example adding brick detail to a corner edge. Bump maps are by far the easiest type of relief image to manage as they work with practically any surface, no matter the geometry.

What does a displacement map do?

- Displacement maps, although they can be derived from the same type of image as a bump map, are much more powerful. They can truly deform geometry up to and including edge detail, making them ideal for a much wider range of uses such as terrain creation (sometimes a displacement is called a height map for large-scale deformation) and detail modelling.
- The reason that displacement maps are not as commonly used is that they can be computationally intensive and they tend to like high-resolution geometry to work with, which can make them less than ideal for some tasks.
- Either way, understanding bump and displacement maps will enable any artist to add detail to their models more quickly and intuitively than through other image-based methods.

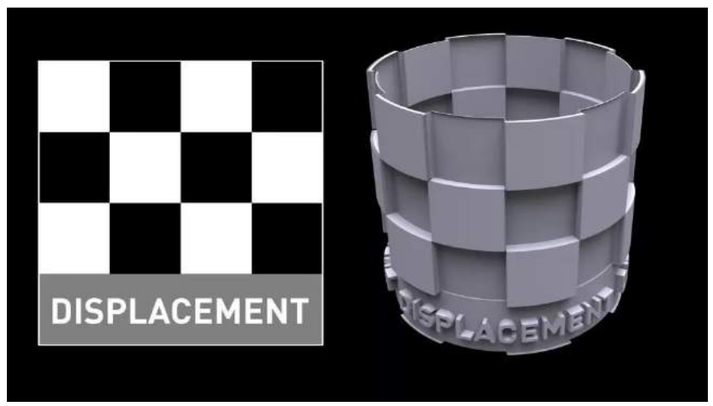
When to use a bump map



At its simplest, bump mapping only modifies the surface of a piece of geometry, whereas displacement mapping is actually altering the geometry.

Bump maps are great at adding a lot of low-relief detail on low-polygon objects, so a one-polygon wall could show hundreds of bricks thanks to bump mapping. It can be an issue when edge detail needs to be shown, as bump mapping does not work with side detail – it only shows the true underlying geometry.

When to use a displacement map

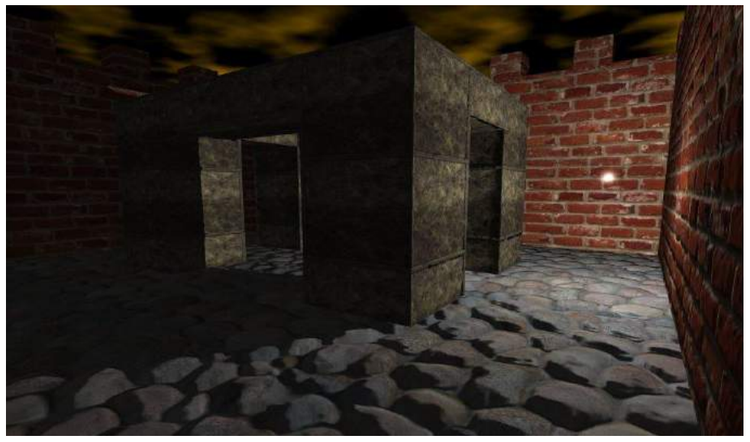


Displacement maps are a hugely powerful technique as they can intuitively allow model detail to be added with a simple greyscale image. A perfect example is when they are used as a simple method of creating the height data for a landscape.

But they can produce stunning results.

Create your own bump and displacement maps

Bump and Displacement mapping in a scene



- One of the best things about bump and displacement maps is that visually they make sense, with white areas usually denoting the highest areas, black the lowest and 50% grey equalling no change.

- This means that while there are applications like Bitmap2Material that can make a good guess at creating relief, it is sometimes better to use a 2D image application.
- Using a high-pass filter can be an excellent way to get started in creating a relief map which can then be painted into using traditional 2D painting techniques.

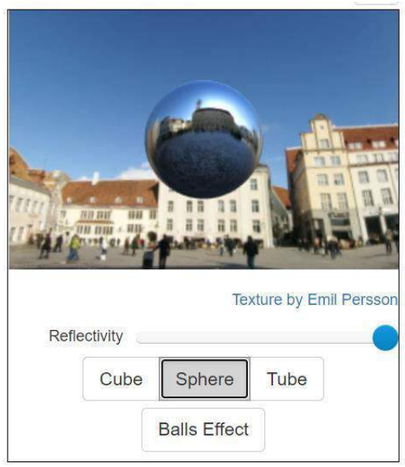
Environmental mapping or reflection mapping

Environment mapping, or reflection mapping, is an efficient image-based lighting technique for approximating the appearance of a reflective surface by means of a precomputed texture image.

3.Environment mapping

- **Environment mapping**, also called **refection mapping**, is a method for applying environment reflections on a reflective surface.
- Environmental mapping may have nothing to do with the actual surroundings of the object. This technique generates a realistic appearance of a reflective surface and requires much less time than ray tracing.
- However, for **Reflection Maps**, the lighter areas are more reflective and the darker areas are less reflective

- dynamic cube map
- https://cglearn.codelight.eu/pub/computer-graphics/environment-mapping



4.Photorealistic mapping vs non-photorealistic mapping

- Non-photorealistic rendering is a commonly used method in computational photography to produce a cartoon-like version of an image by using color quantification, edge detection, and edge sharpening.

- Example Video: Tao Tei
- Taotie are one of the "four evil creatures of the world" in ancient Chinese mythology. In Chinese classical texts such as the "Classic of Mountains and Seas", the fiend is named alongside the Hundun, Qiongqi and Taowu
- https://www.youtube.com/watch?v=V2XYnBb4BrI

- Render Texture Basics (Unity 5) - YouTube
https://www.youtube.com/watch?v=Goqe5IorfN0

How do you render textures in Photoshop?
https://www.youtube.com/watch?time_continue=8&v=ZHtmuGE_LDM&feature=emb_logo

4.Compositing

- **Compositing** is the combining visual elements from separate sources into single images, often to create the illusion that all those elements are parts of the same scene.
- Live-action shooting for compositing is variously called "chroma key“
- Chroma key is a visual effect technique for compositing (layering) two images or video streams together based on color hues.
- Color hues may be blue screen, green screen.

- Ex: Memes, Ads

What is compositing in VFX?

- Compositors create the final image of a frame, shot or **VFX** sequence.
- They take all the different digital materials used (assets), such as computer-generated (CG) images, live action footage and matte paintings, and combine them to appear as one cohesive image and shot

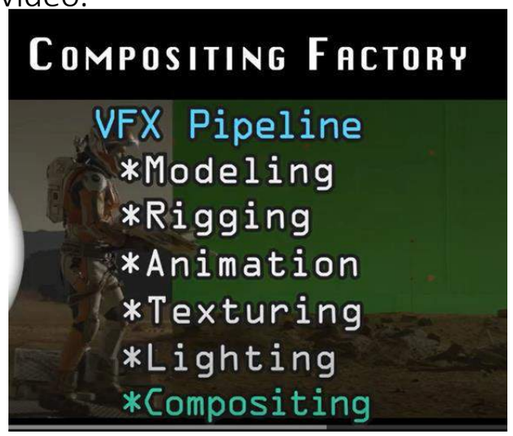
- Compositing is the combination of multiple layers of images or video elements to render a final still or moving image. The combination of layers can be a physical or software-based operation. *Rotoscoping is one compositing method.
- Compositing is often used for 2D content, layering multiple images and print for still images in advertisements, memes and other content for print publications, websites and apps.

*Rotoscoping is an animation technique that animators use to trace over motion picture footage, frame by frame, to produce realistic action.

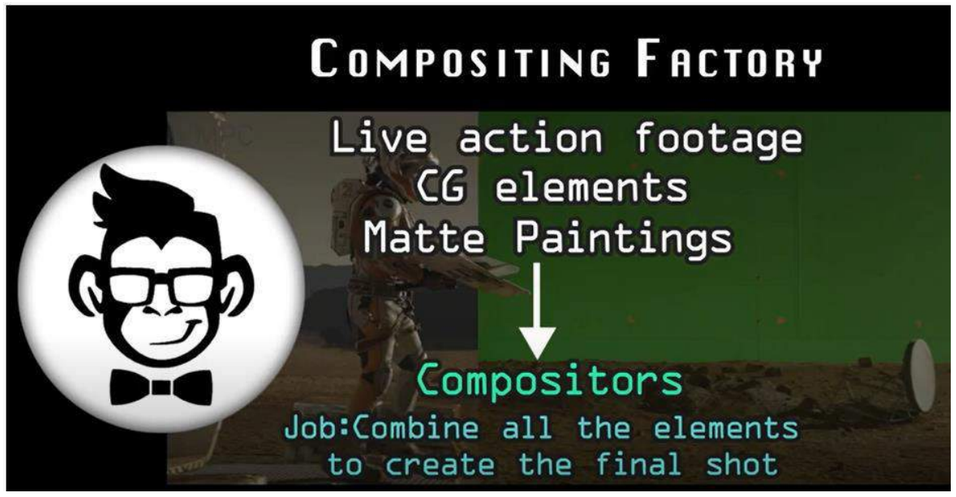
Four images of the same subject, removed from their original backgrounds and composited onto a new background

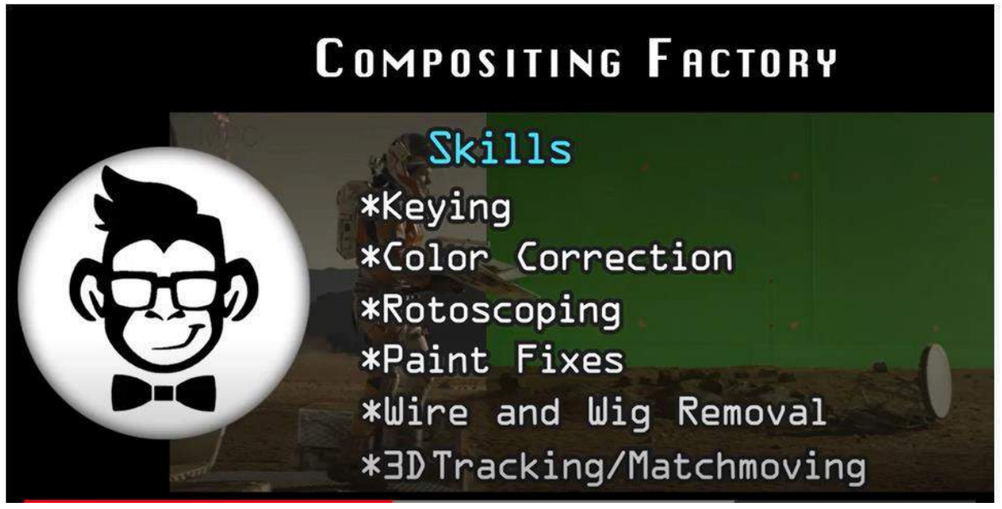
- For television, compositing on green screens is often used to provide virtual backgrounds and sets. Later they will move to necessary backgrounds.
- Augmented reality (AR) uses compositing techniques for various purposes, such as displaying layers of images at different levels of depth in a display. The combined layering technique is also used in motion pictures. In pictures depicting faraway or imaginary places, compositing may be used along with green screens to create convincing background visuals. CGI (computer-generated imagery) effects are added to films with compositing as well.

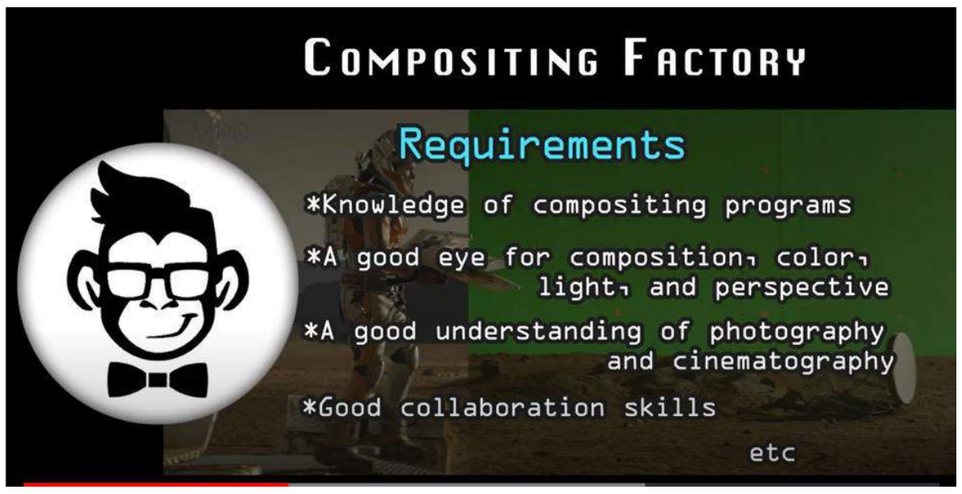
Compositing is one of the stages in the VFX pipeline. But, you might be wondering, “What is compositing? What do compositors do?” I’ll show you in this video.



https://www.youtube.com/watch?v=gAtUGfSURxk

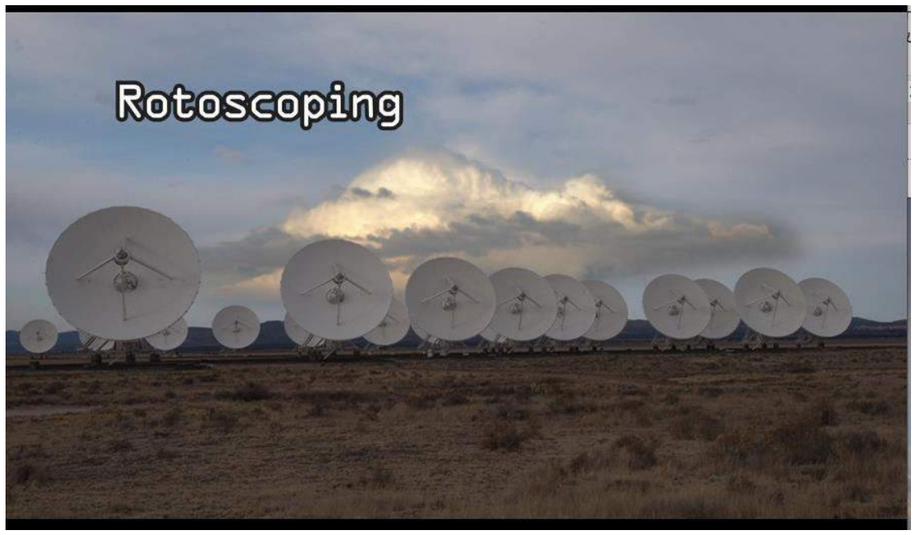

Compositing Factory
Live action footage
CG elements
Matte Paintings
Compositors
Job:Combine all the elements
to create the final shot

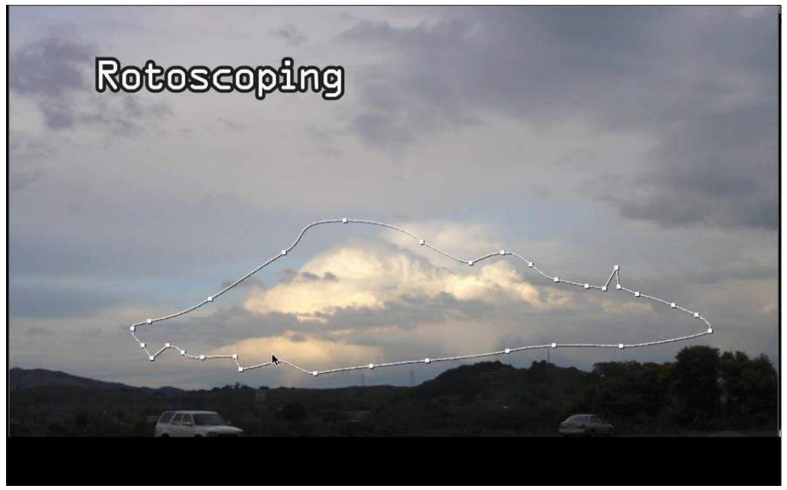

Compositing Factory
Skills
*Keying
*Color Correction
*Rotoscoping
*Paint Fixes
*Wire and Wig Removal
*3D Tracking/Matchmoving


Compositing Factory
Requirements
*Knowledge of compositing programs
*A good eye for composition, color,
light, and perspective
*A good understanding of photography
and cinematography
*Good collaboration skills
etc

Keying


Green Screen Technique



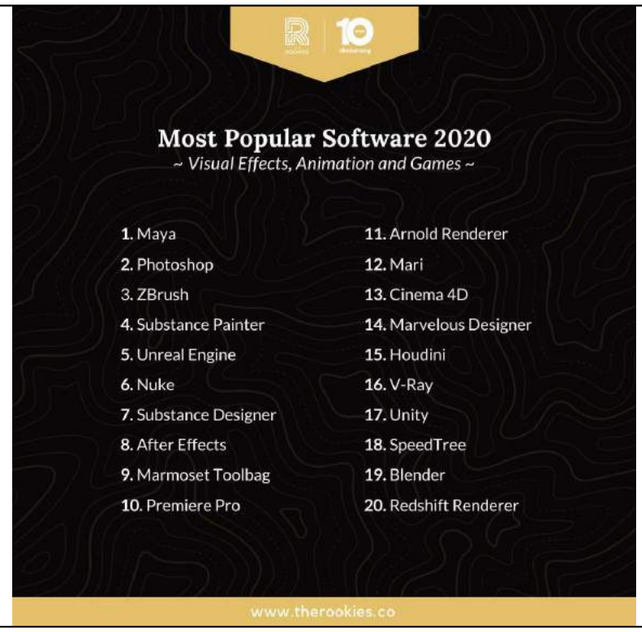


Removal thru painting/fixing/rotoscoping

Summary

- That's all about CG Rendering Concepts:
  - Ray tracing
  - Ray Casting
  - Radiosity
  - Texturing
  - Compositing

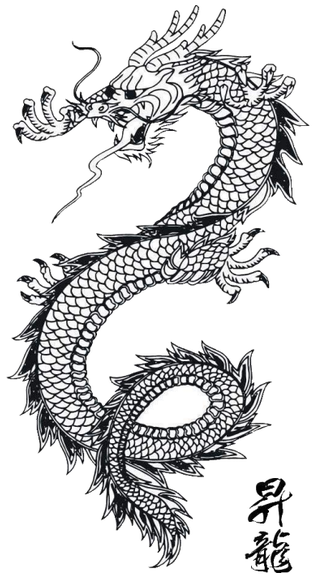
- These are the basic skills required for composting like
  - Keying
  - Color correction
  - Rotoscope
  - Removal
  - Transform to final
- Click for skills explanations (watch the video from 3rd sec)



Bézier Curves and Splines

Today

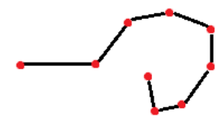
- Smooth curves in 2D
  - Useful in their own right
  - Provides basis for surface editing



This image is in the public domain
Source: Wikimedia Commons

Modeling 1D Curves in 2D

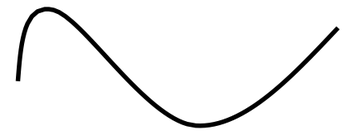
- Polylines
  - Sequence of vertices connected by straight line segments
  - Useful, but not for smooth curves
  - This is the representation that usually gets drawn in the end (a curve is converted into a polyline)

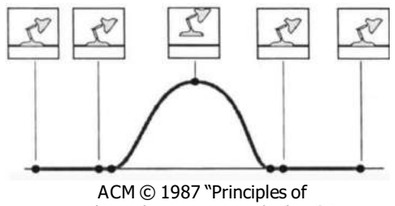


- **Smooth curves**
  - How do we specify them?
  - A little harder (but not too much)

Splines

- A type of smooth curve in 2D/3D
- Many different uses
  - 2D illustration (e.g., Adobe Illustrator)
  - Fonts (e.g., PostScript, TrueType)
  - 3D modeling
  - Animation: trajectories
- In general: interpolation and approximation

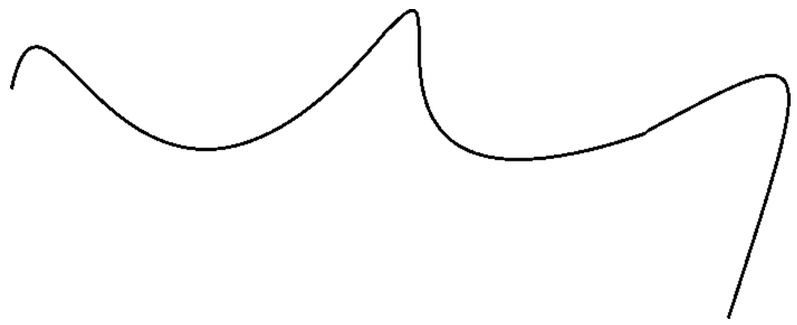




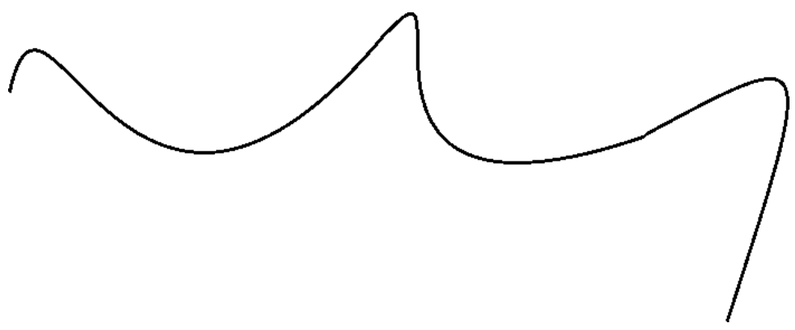
ACM © 1987 "Principles of traditional animation applied to 3D computer animation"

© ACM. All rights reserved. This content is excluded from our Creative Commons license. For more information, see http://ocw.mit.edu/help/faq-fair-use/.

How Many Dimensions?

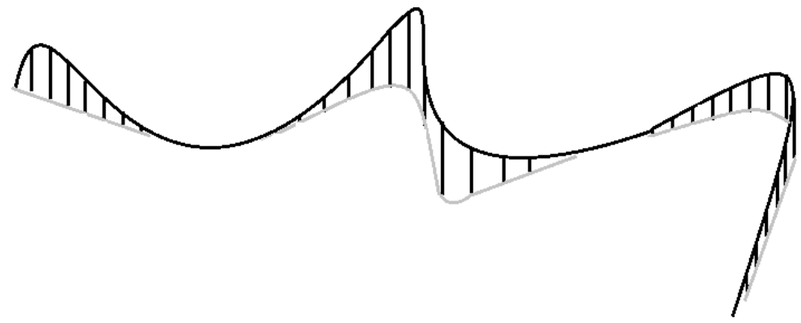


How Many Dimensions?



This curve lies on the 2D plane, but is itself 1D.

How Many Dimensions?



This curve lies on the 2D plane, but is itself 1D.

You can just as well define 1D curves in 3D space.

Two Definitions of a Curve

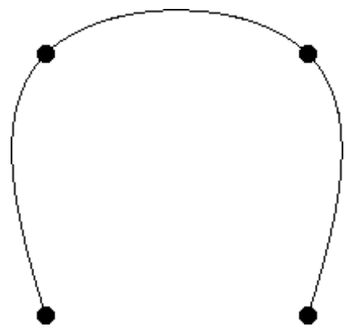
- A continuous 1D set of points in 2D (or 3D)
- A mapping from an interval S onto the plane
  - That is, *P(t)* is the point of the curve at parameter *t*

$$P : \mathbb{R} \ni S \mapsto \mathbb{R}^2, \quad P(t) = \begin{pmatrix} x(t) \\ y(t) \end{pmatrix}$$

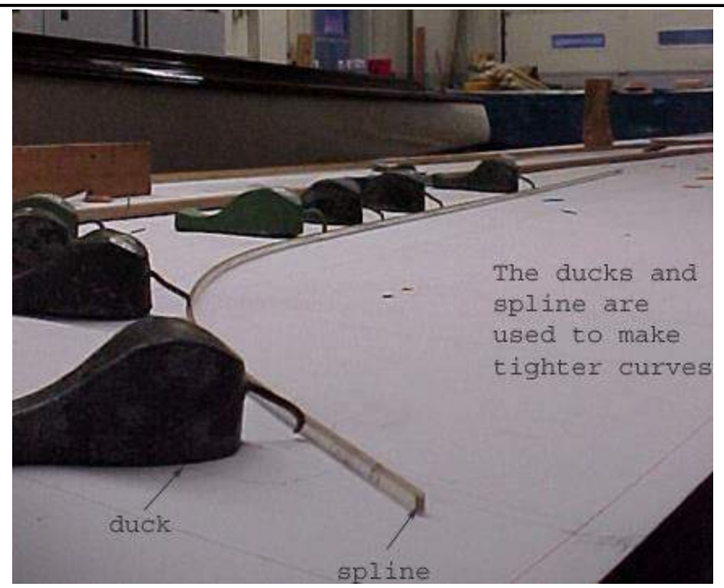
- Big differences
  - It is easy to generate points on the curve from the 2nd
  - The second definition can describe trajectories, the speed at which we move on the curve

General Principle of Splines

- User specifies **control points**
- We will interpolate the control points by a smooth curve
  - The curve is completely determined by the control points.



Physical Splines



Courtesy of The Antique Boat Museum.

See http://en.wikipedia.org/wiki/Flat_spline

Two Application Scenarios

- Approximation/interpolation
  - We have "data points", how can we interpolate?
  - Important in many applications
- User interface/modeling
  - What is an easy way to specify a smooth curve?
  - Our main perspective today.

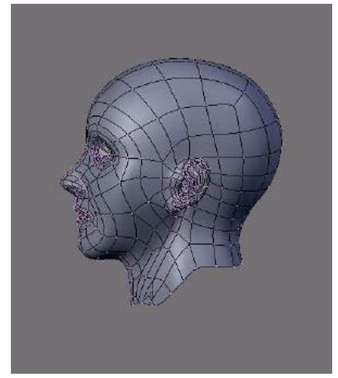


Image courtesy of SaphireS on Wikimedia Commons. License: CC-BY-SA. This content is excluded from our Creative Commons license. For more information, see http://ocw.mit.edu/help/faq-fair-use/.

Splines

- Specified by a few control points
  - Good for UI
  - Good for storage
- Results in a smooth parametric curve P(t)
  - Just means that we specify $x(t)$ and $y(t)$
  - In practice: low-order polynomials, chained together
  - Convenient for animation, where t is time
  - Convenient for *tessellation* because we can discretize t and approximate the curve with a polyline

Tessellation

- It is easy to rasterize mathematical line segments into pixels
  - Softwares and the graphics hardware can do it for you
- But polynomials and other parametric functions are harder

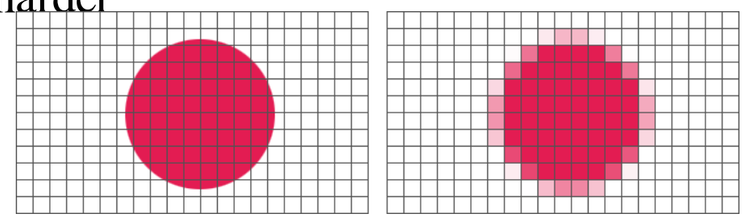
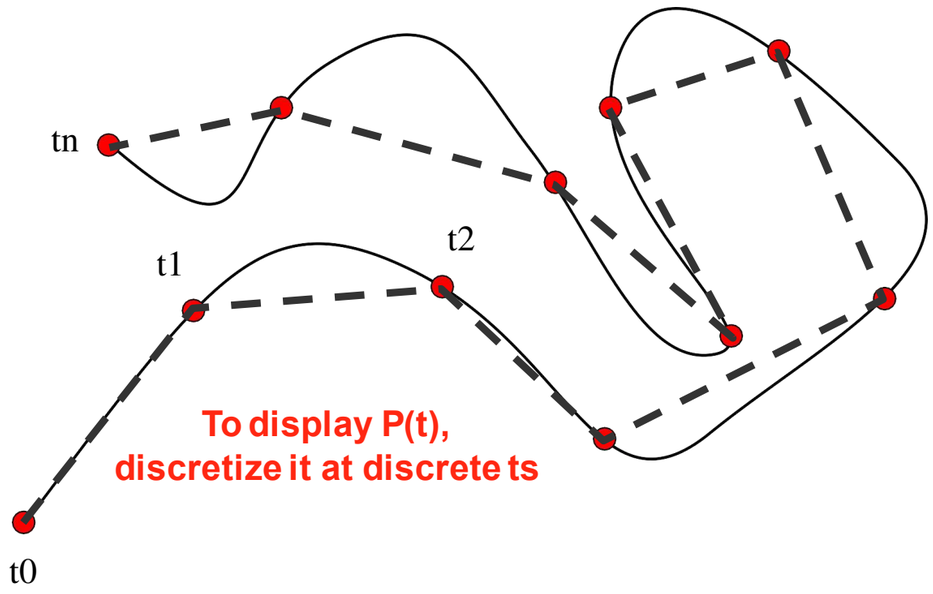
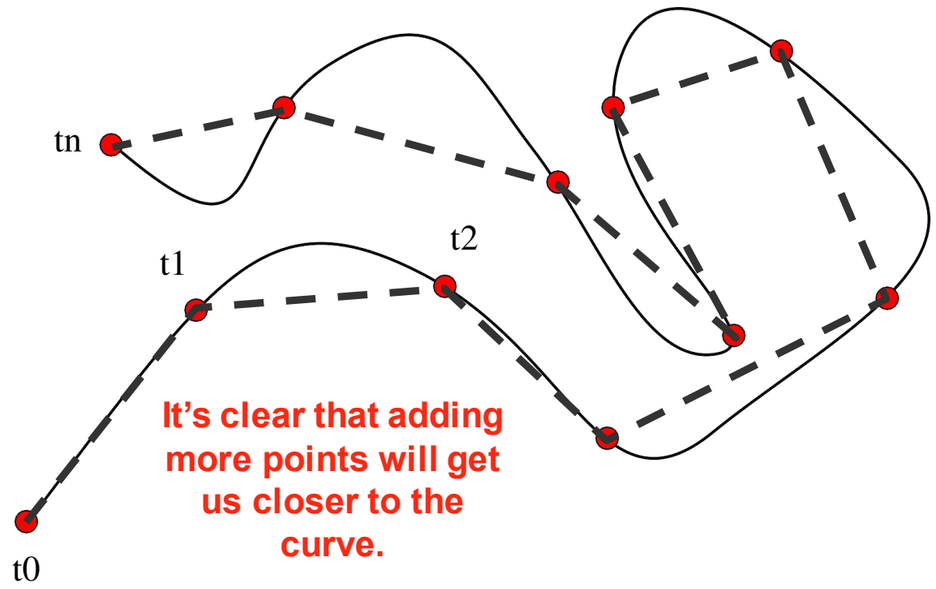


Image courtesy of Phrood on Wikimedia Commons. License: CC-BY-SA.This content is excluded from our Creative Commons license. For moreinformation, see http://ocw.mit.edu/help/faq-fair-use/.

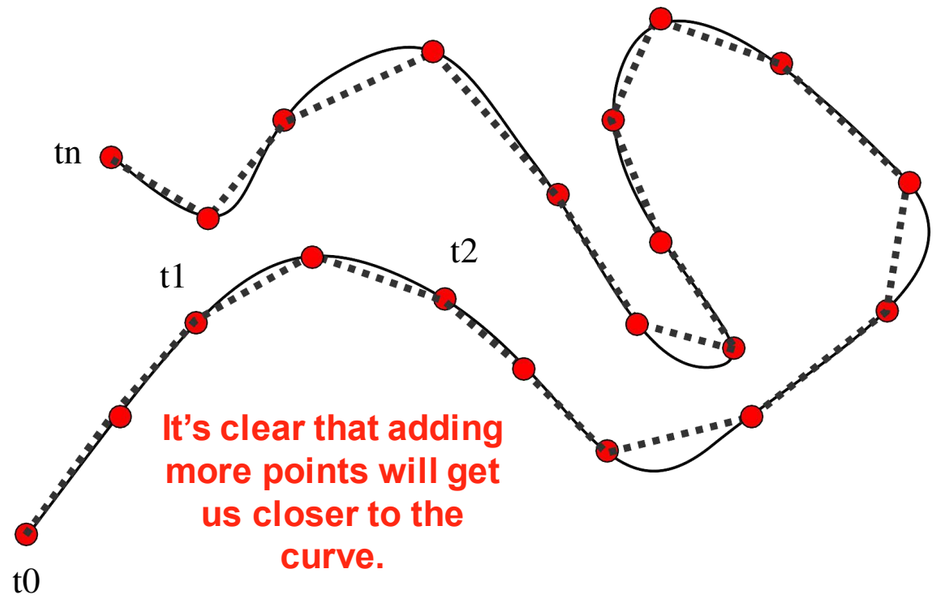
Tessellation



Tessellation

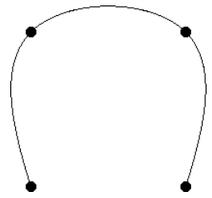


Tessellation



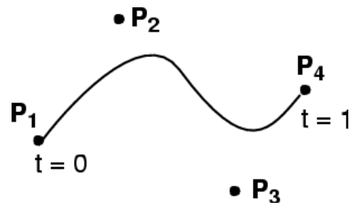
Interpolation vs. Approximation

- Interpolation
  - Goes through all specified points
  - Sounds more logical



Interpolation

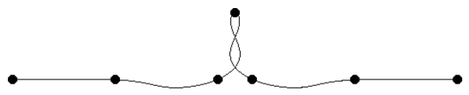
- Approximation
  - Does not go through all points



Approximation

Interpolation vs. Approximation

- Interpolation
  - Goes through all specified points
  - Sounds more logical
  - But can be more unstable



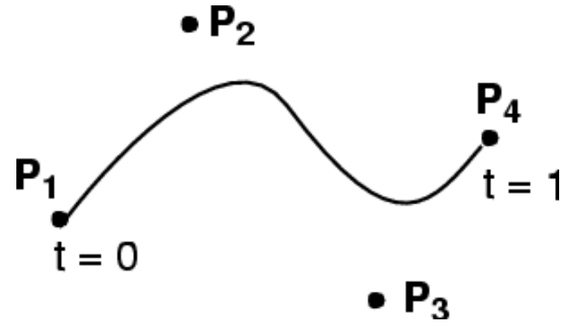
Interpolation

- Approximation
  - Does not go through all points
  - Turns out to be convenient

Approximation

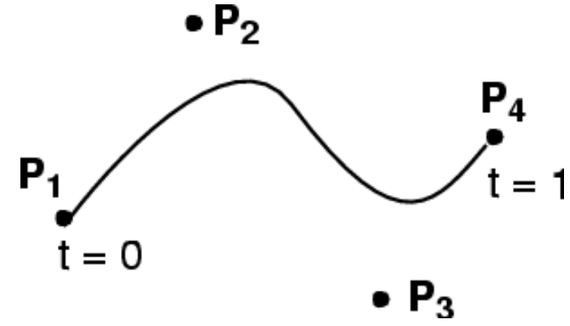
Cubic Bézier Curve

- User specifies 4 control points **P**1 ... **P**4
- Curve goes through (interpolates) the ends **P**1, **P**4
- Approximates the two other ones
- Cubic polynomial



Cubic Bézier Curve

- P(t) = $(1-t)^3$ **P**1
  + $3t(1-t)^2$ **P**2
  + $3t^2(1-t)$ **P**3
  + t^3 **P**4



That is,

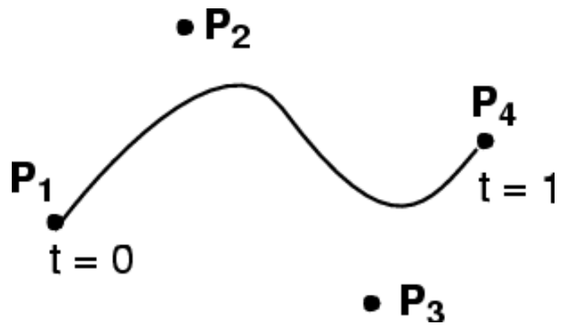
$$x(t) = (1-t)^3\, x_1 + 3t(1-t)^2\, x_2 + 3t^2(1-t)\, x_3 + t^3\, x_4$$

$$y(t) = (1-t)^3\, y_1 + 3t(1-t)^2\, y_2 + 3t^2(1-t)\, y_3 + t^3\, y_4$$

Cubic Bézier Curve

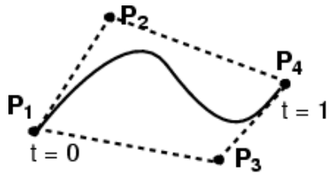
- $P(t) = (1-t)^3 \quad \mathbf{P1}$
  $+ \quad 3t(1-t)^2 \quad \mathbf{P2}$
  $+ \quad 3t^2(1-t) \quad \mathbf{P3}$
  $+ \quad t^3 \quad \mathbf{P4}$

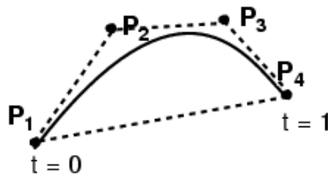
Verify what happens for t=0 and t=1

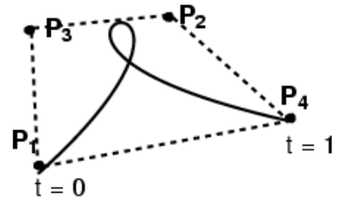


Cubic Bézier Curve

- 4 control points
- Curve passes through first & last control point



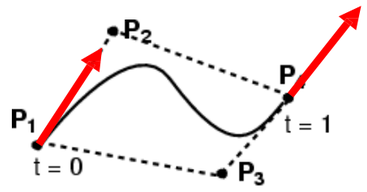


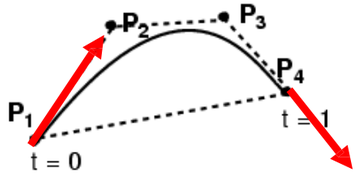


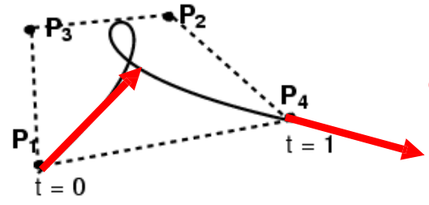
Courtesy of Seth Teller.
Used with permission.

Cubic Bézier Curve

- 4 control points
- Curve passes through first & last control point
- Curve is tangent at **P1** to (**P1-P2**) and at **P4** to (**P4-P3**)







A Bézier curve is bounded by the **convex hull** of its control points.

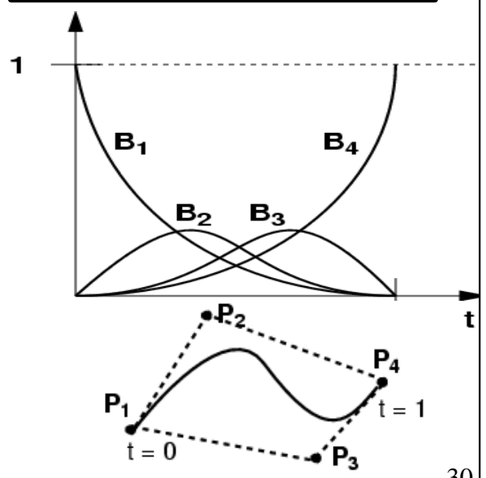
Why Does the Formula Work?

- Explanation 1:
  - Magic!
- Explanation 2:
  - These are smart weights that describe the influence of each control point
- Explanation 3:
  - It is a linear combination of ***basis polynomials.***

Weights

- P(t) is a weighted combination of the 4 control points with weights:
  - $B1(t)=(1-t)^3$
  - $B2(t)=3t(1-t)^2$
  - $B3(t)=3t^2(1-t)$
  - $B4(t)=t^3$
- First, P1 is the most influential point, then P2, P3, and P4

$$P(t) = (1-t)^3\ \mathbf{P}1 + 3t(1-t)^2\ \mathbf{P}2 + 3t^2(1-t)\ \mathbf{P}3 + t^3\ \mathbf{P}4$$



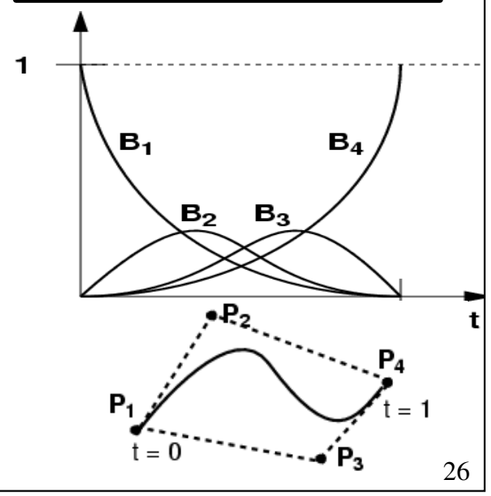
Why Does the Formula Work?

- Explanation 1:
  - Magic!
- Explanation 2:
  - These are smart weights that describe the influence of each control point
- Explanation 3:
  - It is a linear combination of ***basis polynomials.***
  - ***The opposite perspective: control points are the weights of polynomials!!!***

Weights

- P2 and P3 never have full influence
  - Not interpolated!

$$P(t) = (1-t)^3\ \mathbf{P}1 + 3t(1-t)^2\ \mathbf{P}2 + 3t^2(1-t)\ \mathbf{P}3 + t^3\ \mathbf{P}4$$



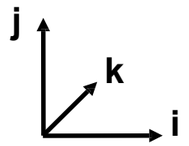
Why Study Splines as Vector Space?

- Understand relationships between types of splines
  - Conversion
- Express what happens when a spline curve is transformed by an affine transform (rotation, translation, etc.)
- Cool simple example of non-trivial vector space
- Important to understand for advanced methods such as finite elements

Usual Vector Spaces

- In 3D, each vector has three components x, y, z
- But geometrically, each vector is actually the sum

$$v = x\,\vec{\boldsymbol{i}} + y\,\vec{\boldsymbol{j}} + z\,\vec{\boldsymbol{k}}$$



- **i**, **j**, **k** are basis vectors

- Vector addition: just add components
- Scalar multiplication: just multiply components

Polynomials as a Vector Space

- Polynomials $y(t) = a_0 + a_1 t + a_2 t^2 + \ldots + a_n t^n$
- Can be added: just add the coefficients

$$(y+z)(t) = (a_0 + b_0) + (a_1 + b_1)t + (a_2 + b_2)t^2 + \ldots + (a_n + b_n)t^n$$

- Can be multiplied by a scalar: multiply the coefficients

$$s \cdot y(t) = (s \cdot a_0) + (s \cdot a_1)t + (s \cdot a_2)t^2 + \ldots + (s \cdot a_n)t^n$$

Polynomials as a Vector Space

- Polynomials $y(t) = a_0 + a_1 t + a_2 t^2 + \ldots + a_n t^n$

- In the polynomial vector space, {1, *t*, ..., *t*n} are the basis vectors, *a*0, *a*1, ..., *a*n are the components

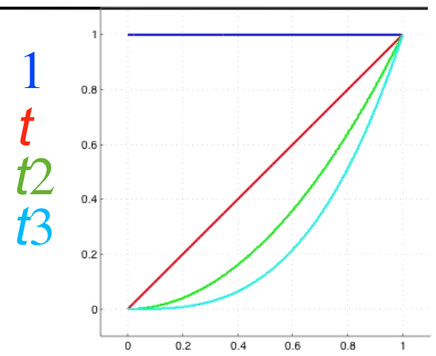
Subset of Polynomials: Cubic

$$y(t) = a_0 + a_1\,t + a_2\,t^2 + a_3\,t^3$$

- Closed under addition & scalar multiplication
  - Means the result is still a cubic polynomial
- Cubic polynomials also compose a vector space
  - A 4D **subspace** of the full space of polynomials
- The *x* and *y* coordinates of cubic Bézier curves belong to this subspace as functions of *t*.

Canonical Basis for Cubics

$$\{1, t, t^2, t^3\}$$



- Any cubic polynomial is a linear combination of these:

$a0+a1t+a2t2+a3t3 = a0*1+a1*t+a2*t2+a3*t3$

- They are linearly independent
  - Means you cannot write any of the four monomials as a linear combination of the others. (You can try.)

Matrix-Vector Notation

- For example:

1, 1+t, 1+t+t², 1+t-t²+t³

t³, t³+t², t³+t, t³+1

Change-of-basis matrix

"Canonical" monomial basis

These relationships hold for each value of *t*

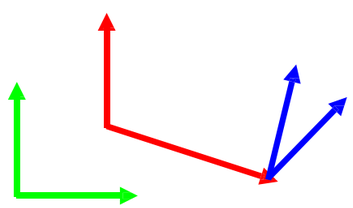
$$\begin{pmatrix} 1 \\ 1+t \\ 1+t+t^2 \\ 1+t-t^2+t^3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & -1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ t \\ t^2 \\ t^3 \end{pmatrix}$$

$$\begin{pmatrix} t^3 \\ t^3+t^2 \\ t^3+t \\ t^3+1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ t \\ t^2 \\ t^3 \end{pmatrix}$$

Different Basis

- For example:
  - {*1*, *1+t*, *1+t+t2*, *1+t-t2+t3*}
  - {*t3*, *t3+t2*, *t3+t*, *t3+1*}

2D examples



- These can all be obtained from $1, t, t^2, t^3$ by linear combination
- Infinite number of possibilities, just like you have an infinite number of bases to span R2

Matrix-Vector Notation

- For example:

1, 1+t, 1+t+t2, 1+t-t2+t3

t3, t3+t2, t3+t, t3+1

Change-of-basis matrix

"Canonical" monomial basis

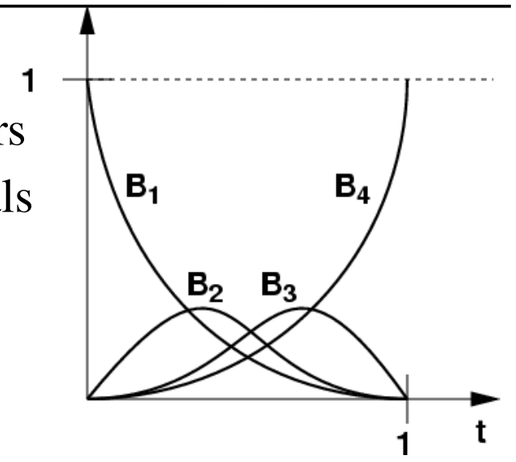
$$\begin{pmatrix} 1 \\ 1+t \\ 1+t+t^2 \\ 1+t-t^2+t^3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & -1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ t \\ t^2 \\ t^3 \end{pmatrix}$$

Not any matrix will do! If it's singular, the basis set will be linearly dependent, i.e., redundant and incomplete.

$$\begin{pmatrix} t^3 \\ t^3+t^2 \\ t^3+t \\ t^3+1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ t \\ t^2 \\ t^3 \end{pmatrix}$$

Bernstein Polynomials

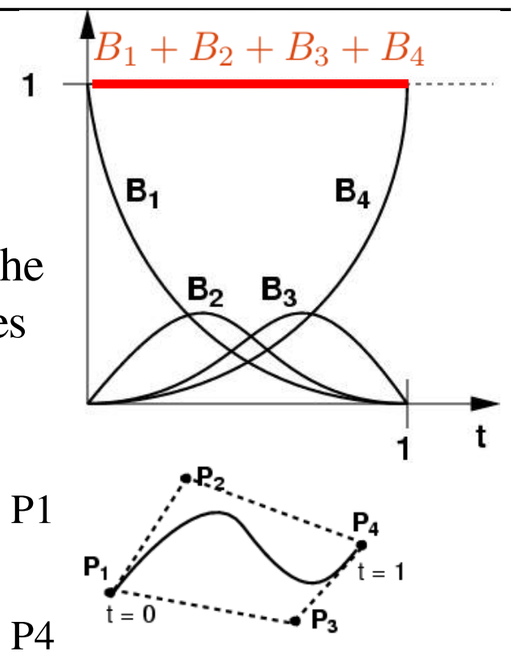
- For Bézier curves, the basis polynomials/vectors are Bernstein polynomials



- For cubic Bezier curve:
  $B1(t)=(1-t)^3$ $B2(t)=3t(1-t)^2$
  $B3(t)=3t^2(1-t)$ $B4(t)=t^3$
  (careful with indices, many authors start at 0)
- Defined for any degree

Properties of Bernstein Polynomials

- ≥ 0 for all $0 \leq t \leq 1$
- Sum to 1 for every t
  - called *partition of unity*
- These two together are the reason why Bézier curves lie within convex hull
- $B1(0) = 1$
  - Bezier curve interpolates P1
- $B4(1) = 1$
  - Bezier curve interpolates P4



Bézier Curves in Bernstein Basis

- $P(t) = P1B1(t) + P2B2(t) + P3B3(t) + P4B4(t)$
  - Pi are 2D points (xi, yi)
- $P(t)$ is a linear combination of the control points with weights equal to Bernstein polynomials at t
- But at the same time, the control points (P1, P2, P3, P4) are the "coordinates" of the curve in the Bernstein basis
  - In this sense, specifying a Bézier curve with control points is exactly like specifying a 2D point with its x and y coordinates.

Two Different Vector Spaces!!!

- The plane where the curve lies, a 2D vector space
- The space of cubic polynomials, a 4D space
- Don't be confused!
- The 2D control points can be replaced by 3D points – this yields space curves.
  - The math stays the same, just add $z(t)$.
- The cubic basis can be extended to higher-order polynomials
  - Higher-dimensional vector space
  - More control points

Change of Basis

- How do we go from Bernstein basis to the canonical monomial basis 1, t, t^2, t^3 and back?
  - With a matrix!

- B1(t)=(1-t)³
- B2(t)=3t(1-t)²
- B3(t)=3t²(1-t)
- B4(t)=t³

$$\begin{pmatrix} B_1(t) \\ B_2(t) \\ B_3(t) \\ B_4(t) \end{pmatrix} = \begin{pmatrix} 1 & -3 & 3 & -1 \\ 0 & 3 & -6 & 3 \\ 0 & 0 & 3 & -3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ t \\ t^2 \\ t^3 \end{pmatrix}$$

New basis vectors

How You Get the Matrix

Cubic Bernstein:

- B1(t)=(1-t)³
- B2(t)=3t(1-t)²
- B3(t)=3t²(1-t)
- B4(t)=t³

Expand these out and collect powers of t. The coefficients are the entries in the matrix B!

$$\begin{pmatrix} B_1(t) \\ B_2(t) \\ B_3(t) \\ B_4(t) \end{pmatrix} = \overbrace{\begin{pmatrix} 1 & -3 & 3 & -1 \\ 0 & 3 & -6 & 3 \\ 0 & 0 & 3 & -3 \\ 0 & 0 & 0 & 1 \end{pmatrix}}^{\boldsymbol{B}} \begin{pmatrix} 1 \\ t \\ t^2 \\ t^3 \end{pmatrix}$$

Change of Basis, Other Direction

- Given B1...B4, how to get back to canonical 1, t, t^2, t^3 ?

$$\begin{pmatrix} B_1(t) \\ B_2(t) \\ B_3(t) \\ B_4(t) \end{pmatrix} = \overbrace{\begin{pmatrix} 1 & -3 & 3 & -1 \\ 0 & 3 & -6 & 3 \\ 0 & 0 & 3 & -3 \\ 0 & 0 & 0 & 1 \end{pmatrix}}^{\boldsymbol{B}} \begin{pmatrix} 1 \\ t \\ t^2 \\ t^3 \end{pmatrix}$$

Change of Basis, Other Direction

That's right, with the inverse matrix!

- Given B1...B4, how to get back to canonical 1, t, t^2, t^3 ?

$$\begin{pmatrix} 1 \\ t \\ t^2 \\ t^3 \end{pmatrix} = \overbrace{\begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1/3 & 2/3 & 1 \\ 0 & 0 & 1/3 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}}^{\boldsymbol{B^{-1}}} \begin{pmatrix} B_1(t) \\ B_2(t) \\ B_3(t) \\ B_4(t) \end{pmatrix}$$

Recap

- Cubic polynomials form a 4D vector space.
- Bernstein basis is canonical for Bézier.
  - Can be seen as influence function of data points
  - Or data points are coordinates of the curve in the Bernstein basis
- We can change between basis with matrices.

More Matrix-Vector Notation

$$P(t) = \sum_{i=1}^{4} P_i \; B_i(t) = \sum_{i=1}^{4} \left[\begin{pmatrix} x_i \\ y_i \end{pmatrix} B_i(t) \right]$$

Bernstein polynomials
(4x1 vector)

$$P(t) = \begin{pmatrix} x(t) \\ y(t) \end{pmatrix} = \begin{pmatrix} x_1 & x_2 & x_3 & x_4 \\ y_1 & y_2 & y_3 & y_4 \end{pmatrix} \begin{pmatrix} B_1(t) \\ B_2(t) \\ B_3(t) \\ B_4(t) \end{pmatrix}$$

point on curve
(2x1 vector)

matrix of
control points (2 x 4)

Flashback

$$\begin{pmatrix} B_1(t) \\ B_2(t) \\ B_3(t) \\ B_4(t) \end{pmatrix} = \overbrace{\begin{pmatrix} 1 & -3 & 3 & -1 \\ 0 & 3 & -6 & 3 \\ 0 & 0 & 3 & -3 \\ 0 & 0 & 0 & 1 \end{pmatrix}}^{\boldsymbol{B}} \begin{pmatrix} 1 \\ t \\ t^2 \\ t^3 \end{pmatrix}$$

Cubic Bézier in Matrix Notation

point on curve
(2x1 vector)

$$\mathrm{P}(t) = \begin{pmatrix} x(t) \\ y(t) \end{pmatrix} =$$

Canonical
monomial basis

$$\begin{pmatrix} x_1 & x_2 & x_3 & x_4 \\ y_1 & y_2 & y_3 & y_4 \end{pmatrix} \begin{pmatrix} 1 & -3 & 3 & -1 \\ 0 & 3 & -6 & 3 \\ 0 & 0 & 3 & -3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ t \\ t^2 \\ t^3 \end{pmatrix}$$

"Geometry matrix"
of control points P1..P4
(2 x 4)

"Spline matrix"
(Bernstein)

General Spline Formulation

$$Q(t) = \mathbf{GBT(t)} = \text{Geometry } \mathbf{G} \cdot \text{Spline Basis } \mathbf{B} \cdot \text{Power Basis } \mathbf{T(t)}$$

- Geometry: control points coordinates assembled into a matrix (P1, P2, …, Pn+1)
- Spline matrix: defines the type of spline
  - Bernstein for Bézier
- Power basis: the monomials (1, t, ..., *t*n)
- Advantage of general formulation
  - Compact expression
  - Easy to convert between types of splines
  - Dimensionality (plane or space) does not really matter

A Cubic Only Gets You So Far

- What if you want more control?

Higher-Order Bézier Curves

- > 4 control points
- Bernstein Polynomials as the basis functions
  - For polynomial of order *n*, the *ith* basis function is

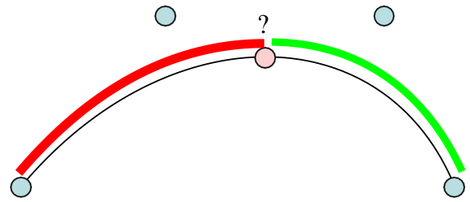
$$B_i^n(t) = \frac{n!}{i!(n-i)!} t^i (1-t)^{n-i}$$

Courtesy of Seth Teller. Used with permission.

- Every control point affects the entire curve
  - Not simply a local effect
  - More difficult to control for modeling

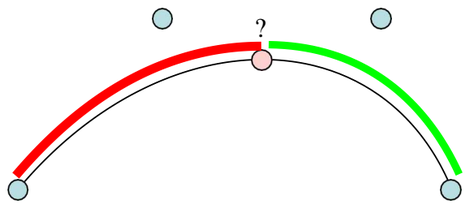
Subdivision of a Bezier Curve

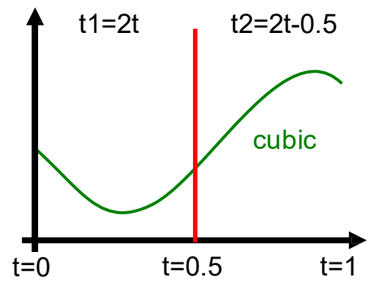
- Can we split a Bezier curve in the middle into two Bézier curves?
  - This is useful for adding detail
  - It avoids using nasty higher-order curves



Subdivision of a Bezier Curve

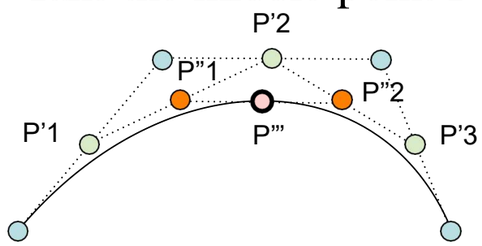
- Can we split a Bezier curve in the middle into two Bézier curves?
  - The resulting curves are again a cubic (Why? A cubic in t is also a cubic in $2t$)
  - Hence it must be representable using the Bernstein basis. So yes, we can!





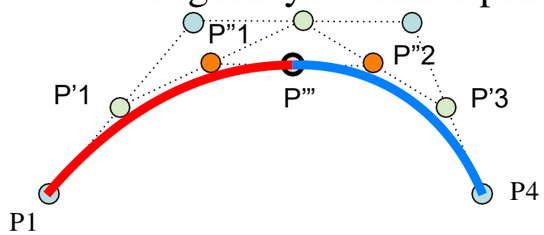
De Casteljau Construction

- Take the middle point of each of the 3 segments
- Construct the two segments joining them
- Take the middle of those two new segments
- Join them
- Take the middle point P''



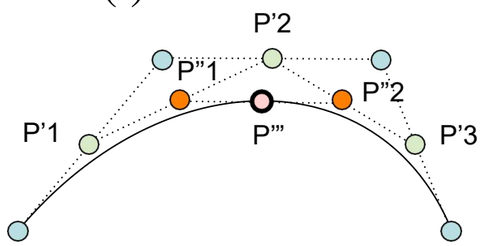
Result of Split in Middle

- The two new curves are defined by
  - P1, P'1, P"1, and P''
  - P''', P"2, P'3, and P4
- Together they exactly replicate the original curve!
  - Originally 4 control points, now 7 (more control)



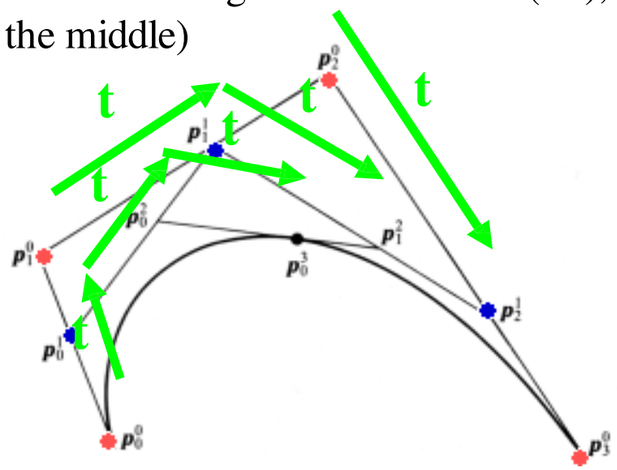
Sanity Check

- Do we actually get the middle point?
- $B1(t)=(1-t)^3$
- $B2(t)=3t(1-t)^2$
- $B3(t)=3t^2(1-t)$
- $B4(t)=t^3$



De Casteljau Construction

- Actually works to construct a point at any *t*, not just 0.5
- Just subdivide the segments with ratio (1-*t*), *t* (not in the middle)



Recap

- Bezier curves: piecewise polynomials
- Bernstein polynomials
- Linear combination of basis functions
  - Basis: control points weights: polynomials
  - Basis: polynomials weights: control points
- Subdivision by de Casteljau algorithm
- All linear, matrix algebra

Recap

- Bezier curves: piecewise polynomials
- Bernstein polynomials
- Linear combination of basis functions
  - Basis: control points weights: polynomials
  - Basis: polynomials weights: control points
- Subdivision by de Casteljau algorithm
- All linear, matrix algebra

Marching Squares

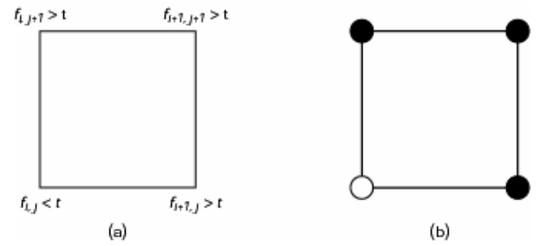
Marching Squares

- Displays isocurves or contours for functions $f(x,y) = t$
- Sample $f(x,y)$ on a regular grid yielding samples $\{f_{ij}(x,y)\}$
- These samples can be greater than, less than, or equal to t
- Consider four samples $f_{ij}(x,y)$, $f_{i+1,j}(x,y)$, $f_{i+1,j+1}(x,y)$, $f_{i,j+1}(x,y)$
- These samples correspond to the corners of a cell
- Color the corners by whether than exceed or are less than the contour value t

Objectives

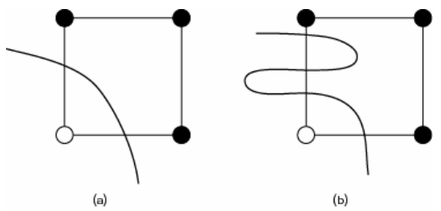
- Nontrivial two-dimensional application
- Important method for
  - Contour plots
  - Implicit function visualization
- Extends to important method for volume visualization
- This lecture is optional
- Material not needed to continue to Chapter 3

Cells and Coloring



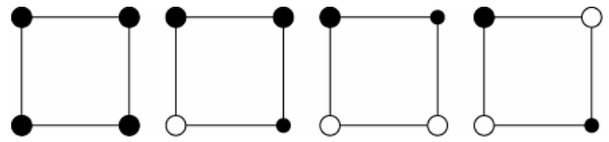
Occum's Razor

- Contour must intersect edge between a black and white vertex an odd number of times
- Pick simplest interpretation: one crossing

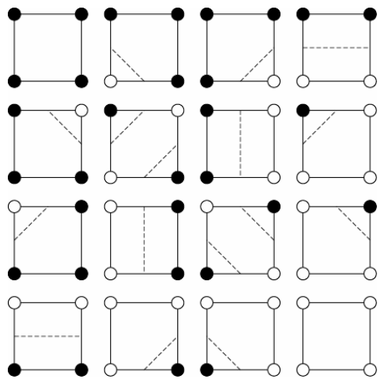


Unique Cases

- Taking out rotational and color swapping symmetries leaves four unique cases
- First three have a simple interpretation



16 Cases



Program on marching squares

- https://editor.p5js.org/namburianupama/sketches/s_B1mZghQ