EE5808
Topics in Computer Graphics

Dr. Shiu Yin YUEN, Kelvin

Email: kelviny.ee Tel: x 7717 Rm: G6359

Web: http://www.ee.cityu.edu.hk/~syyuen

Goals of Computer Graphics

- To use computer and mathematical techniques to build a virtual, real-like 3D world, animated by time changes, inside the computer
- To study techniques that can render the virtual 3D world to real-like 2D images and movies

Movie Industry Applications

Different kinds of “CG” movies

- Type I: Created entirely Using CG

  e.g. ”Fozen 2”
- Type 2: Real people + CG characters

  e.g. “District 9”
- Type 3: CG Movie + Real People

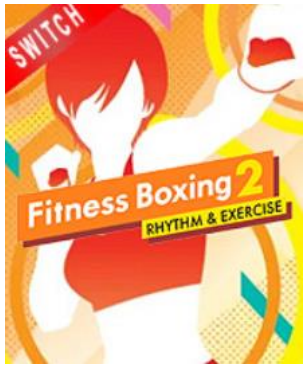
  e.g. “Space Battleship Yamato”
- Type 4: Conventional movie with CG special effects

  e.g. “Initial D” (production documentary)
- Type 5: “3D Movies”

  e.g. “Avatar”

Game Industry Applications

- Mobile phone games (i-phone apps, android …)
- Playstation (PS, Nintendo, Sega …)
- PC Single Person games (Single player and multiple player)
- Hand held games
- Web games
- Motion control games
- …



Advertising Industry Applications

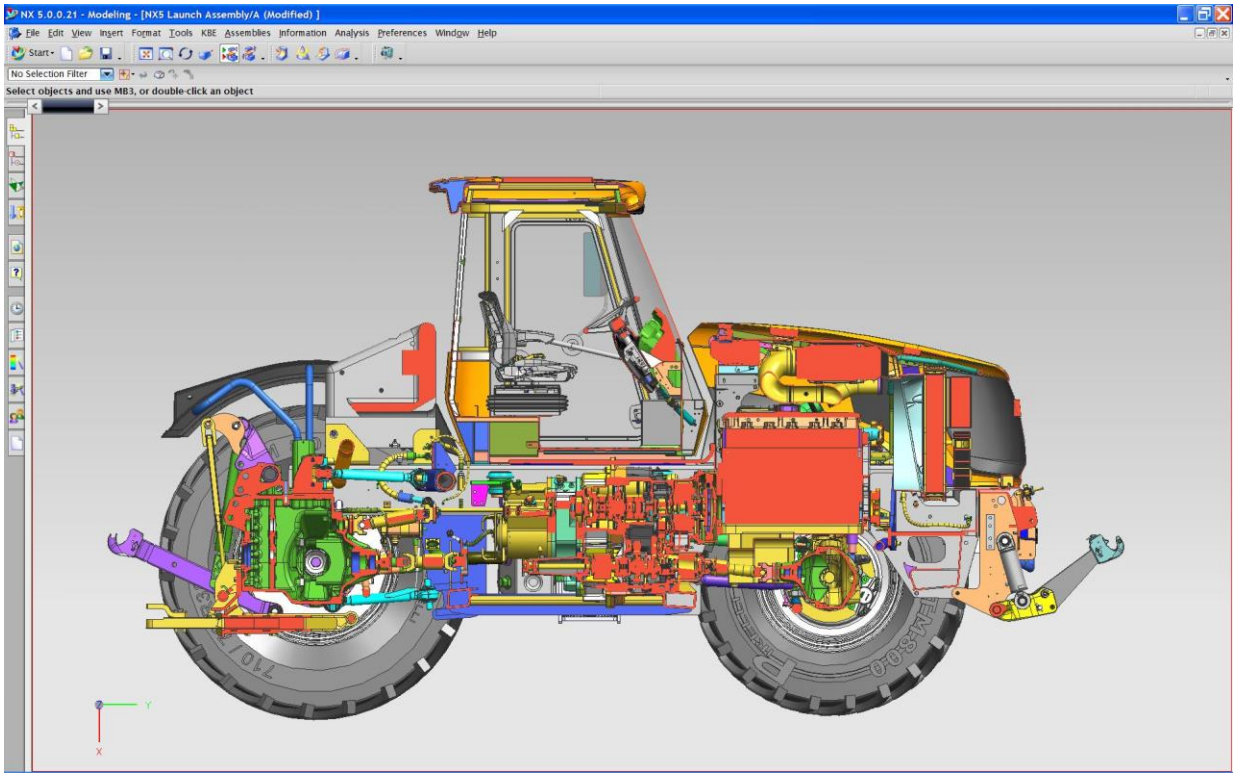
- Commercials in TV

  e.g. TVB Jade
- animations in web page

  e.g. South China Morning Post

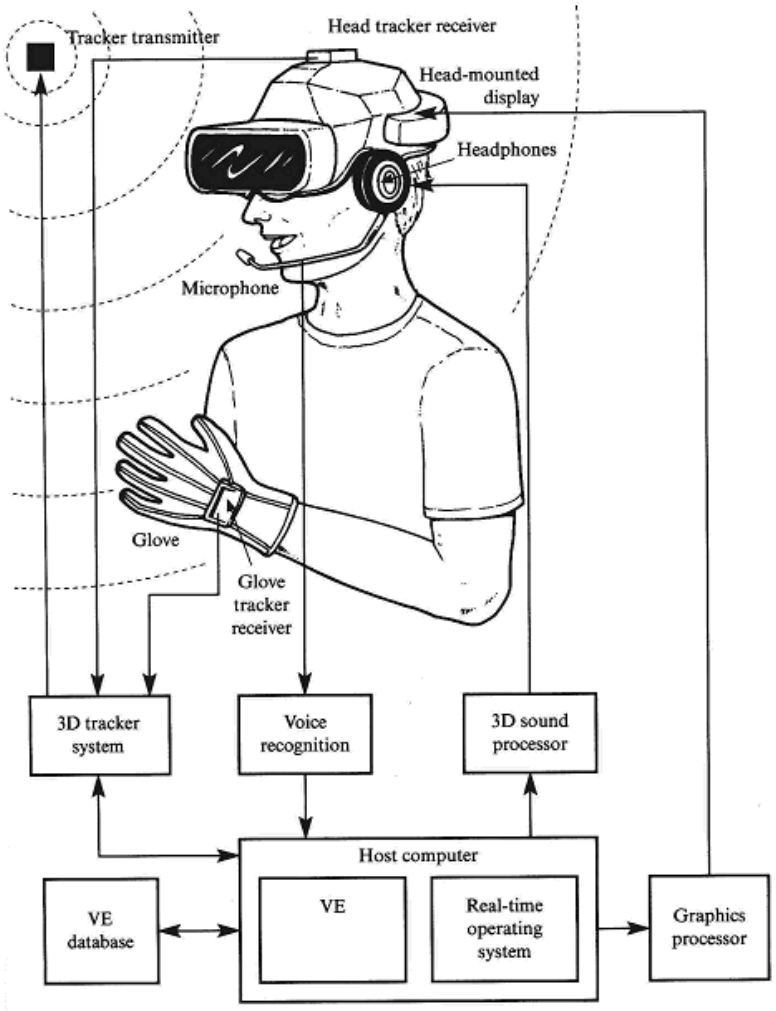
Design Industry Applications

- Computer Aided Design (CAD)



Virtual Reality Industry Application

- VR creates an immersive environment such that the user has the false but real sensation of being in an artificially created world
- Applications in games, medical therapy, visualization, design, surgery practice, teaching, …



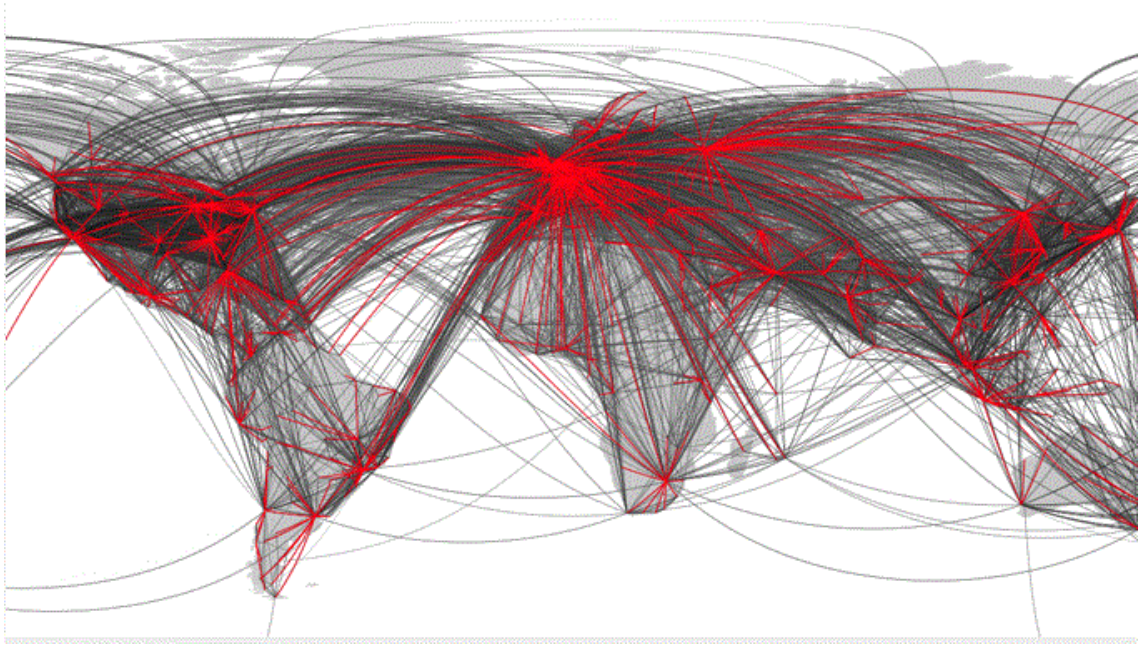
Other less well known Applications

- Visualization

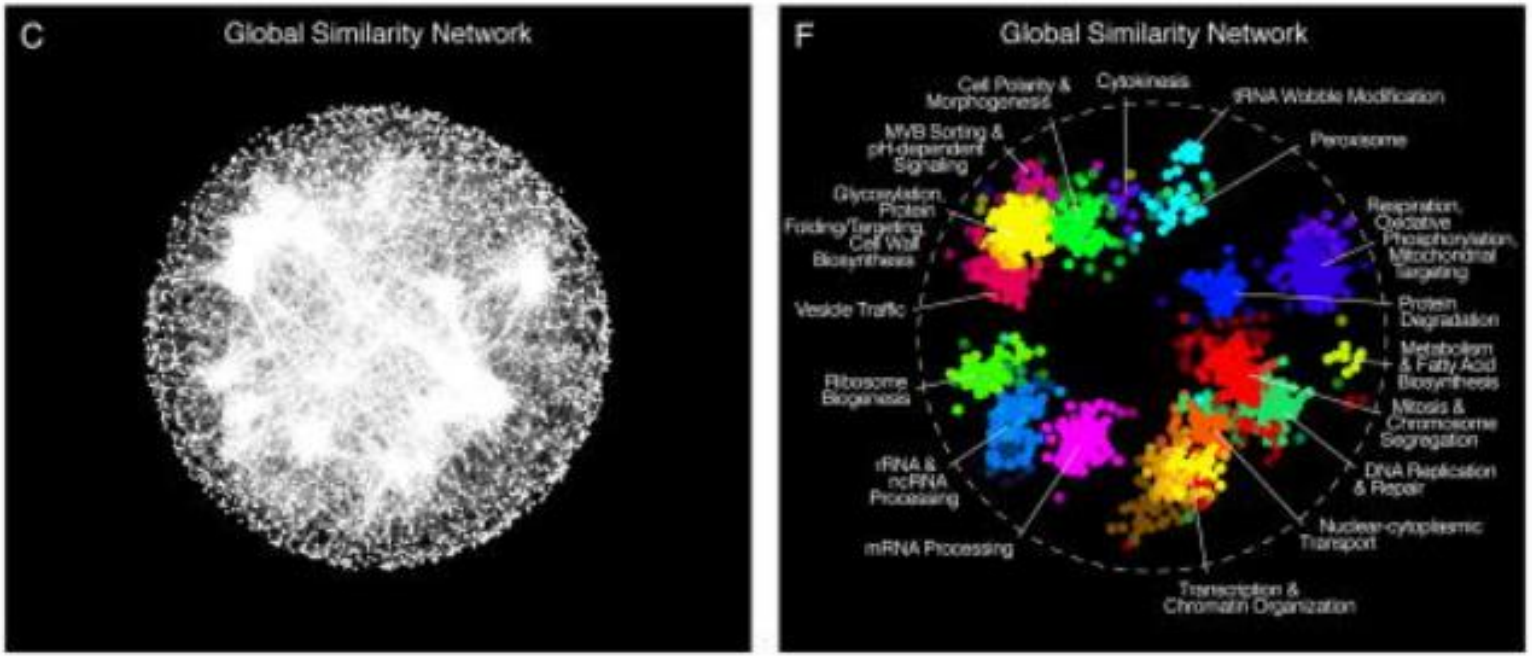
  Visualize mathematical problems

  The shape that no one thought was possible

Complex networks



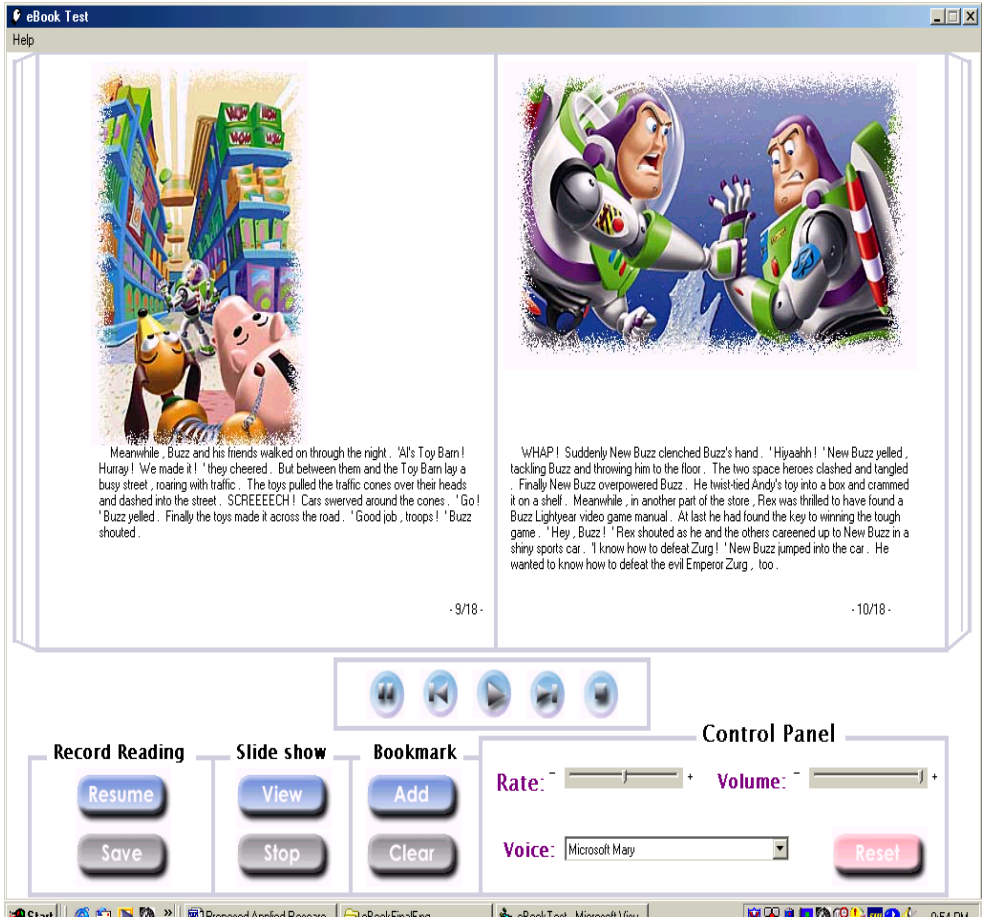
Worldwide air transportation network



A global genetic interaction network maps a wiring diagram of cellular function

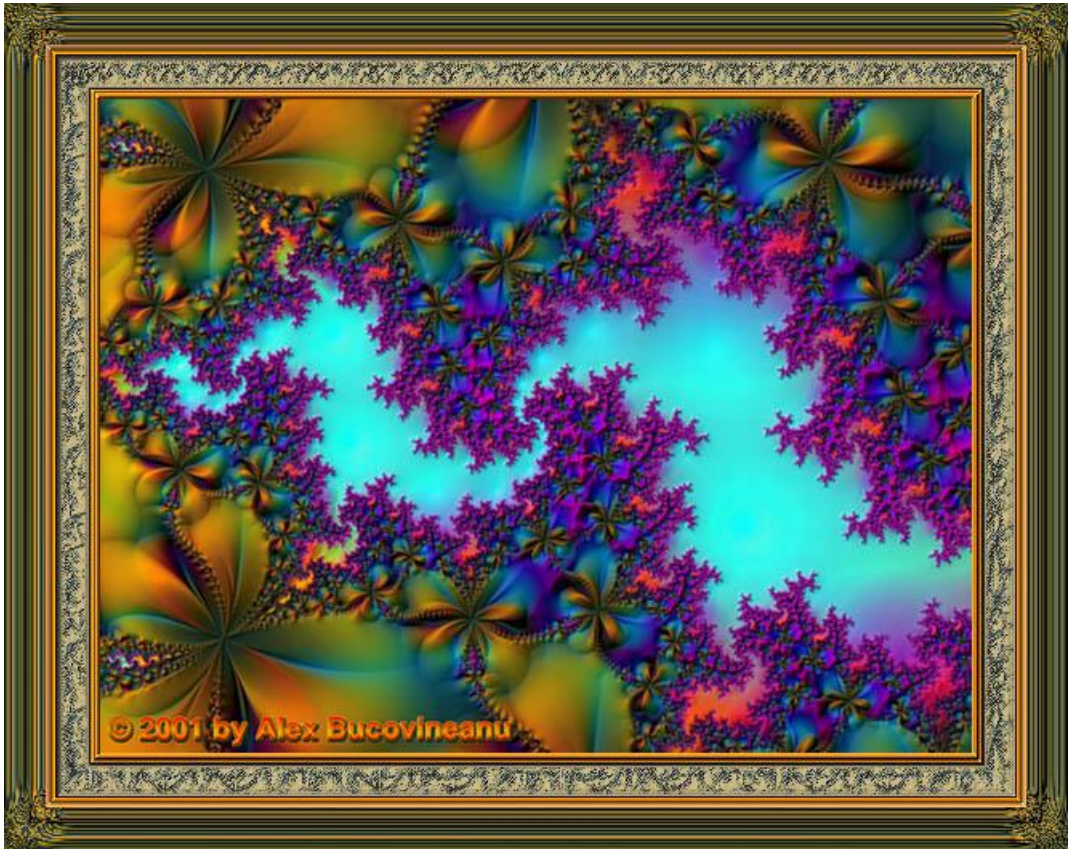
- Training
  - flight simulator
  - car simulator
  - spaceship cabin simulator
  - …

- Education
  - animated story book
  - animated presentation



Electronic books for children

- Computer Art
  - new type of painting
  - New type of art form
  - …



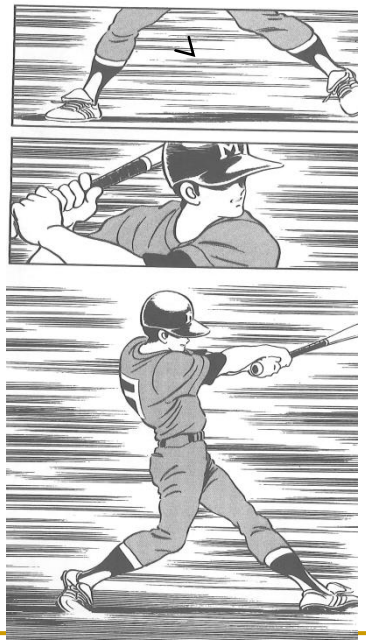
“Butterfly 6228” Author: Human and Computer

Full-body anime generation with Generative Adversarial Nets (GAN)

Converting movie to cartoon and vice versa

- Movie to cartoon
- Cartoon to movie

key frames



Add several intermediate frames in between each key frame by interpolation

H2 CG movie

Comic H2

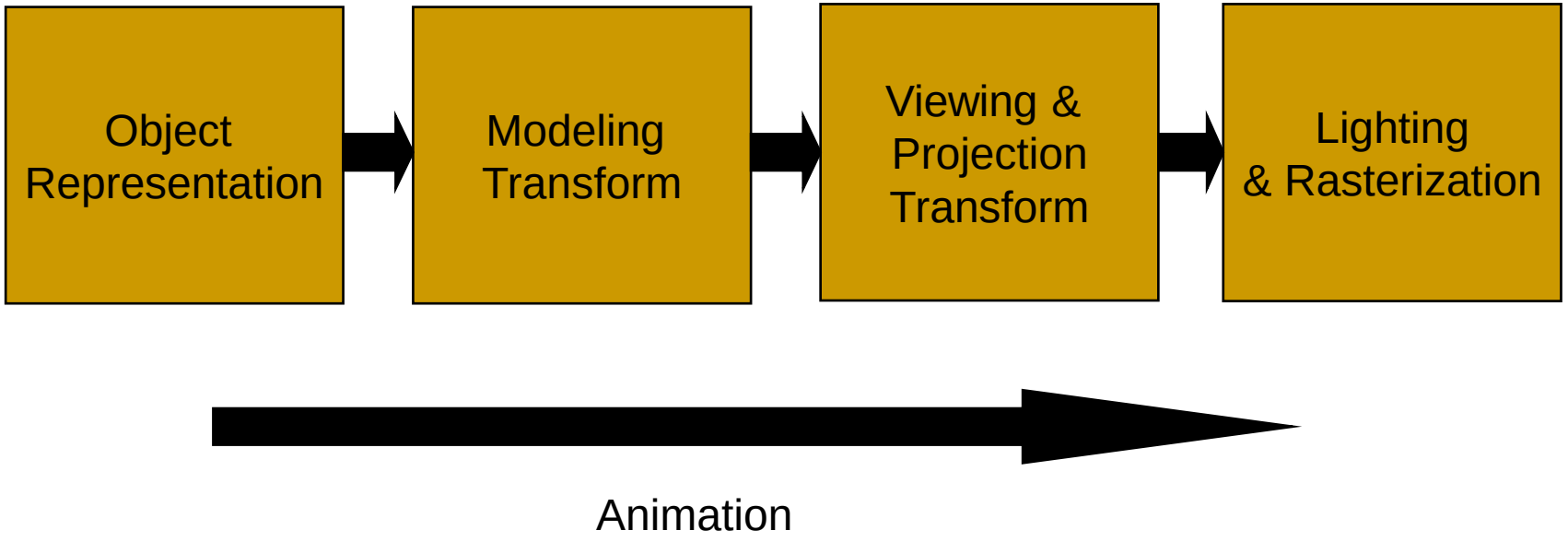
Course Aim

- The aim of this course is to provide students with an understanding of the basic principles, concepts, and techniques of computer graphics from an engineering viewpoint.

CILOS

- (CILO1) Apply 3D object representation techniques to build up a graphics scene
- (CILO2) Model and view articulated objects by hierarchical structuring techniques and coordinate transform
- (CILO3) Apply lighting, shading and rasterization techniques to create a 2D image
- (CILO4) Apply texture mapping and animation techniques to create a movie
- (CILO5) Apply and evaluate advanced graphics techniques
- (CILO6) Create an animation or a game using computer graphics

Course Content



Object Representation (CILO1)

Lecture 2

- How to construct simple objects such as spheres, cones, boxes …

Modeling Transform (CILO2)

Lecture 3

- how to move the simple objects around, rotate them, scale them, reflect them, …

Lecture 4

- introduce the idea of local coordinate system, and how to use the concepts to build a complex coherent moving object by using the hierarchy concept

Viewing and Projection Transform (CILO2)

Lecture 5

- how to put the camera in a desired configuration within the graphics scene and
- how to use different projections to project a 2D image on the camera, and as a result, the different projection effect that can be achieved

Lighting and Rasterization (CILO3)

Lecture 7

- how to create light sources, shading and colour

Lecture 8

- how to eliminate hidden parts

Lecture 9

- how to create realistic shadows

Animation (CILO4)

Lecture 6

- how to animate the images to create a smooth flowing movie

Advanced Graphics Techniques (CILO5)

Lecture 10

- How to create more realistic graphics by texture mapping

Lecture 11

- A selection of advanced techniques

OpenGL Mini Project (CILO6)

- This course uses the open source de facto industry standard: OpenGL, It is a C/C++ library that allows C programmers to write programs that directly access graphics hardware
- The gl and glut libraries

- How to learn OpenGL
  a) Learn during lecture, tutorial and mini project
  b) Search the web for the command
  c) OpenGL Function Index at the end of the text

Other forms of OpenGL

- Fixed function OpenGL is taught in this course first as it is the best for beginners. There are other forms of OpenGL

- WebGL - OpenGL JAVA version is popular
- OpenGL ES is used in iphone
- OpenGL shading language (GLSL) is used nowadays

Relationship of this course with commercial software

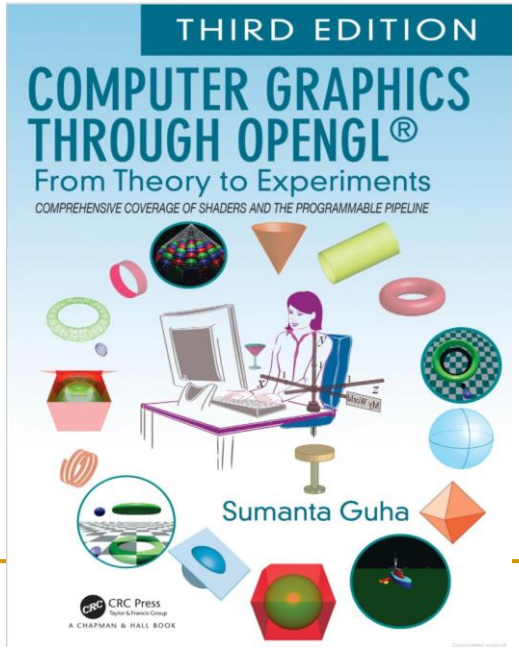
- Commercial software e.g. 3D studio

  used by game developers, many TV commercial studios and architectural visualization studios, movie effects etc.
- This course gives you the technical knowhow behind the techniques in these software. Thus
  - You can use them more sensibly
  - You know the limitations of these software and why
  - You acquire the background for more advanced state of the art knowledge (e.g. SIGGRAPH is the premier conference in CG)
  - You can create a new special effect not supported by the software or research your own novel effect
  - OpenGL is also a popular tool

References

EE5808 Course Reserve (Library -> Course Reserve -> EE5808)

- Computer Graphics with OpenGL. D. Hearn, M.P. Baker, W.R. Carithers. Pearson 4th Ed. 2011.
- Computer Graphics through OpenGL. S. Guha. CRC Press 3rd Ed. 2019. (E-book available in Library. Please read online. Do not download. Otherwise you would borrow it and others cannot access it)



Assessment

- **"To pass the course, students are required to achieve at least 30% in coursework and 30% in the examination"**
- "30% coursework marks" pass requirement. It means that if you treat the assignments (20%), test 1 (15%), test 2 (15%) (total 50% of the whole course), you should get at least 15% out of the 50% of the whole course, or at least 30% in the coursework component with total 100 marks
- "30% exam marks" pass requirement. It means that you should get at least 15% out of the 50% of the exam, or more simply, at least 30 marks in an exam paper with total 100 marks
- **Late hand in of assignments or non-attendance in tests will get no mark**

(Departmental guideline: "For course assessment work with weighting less than 20% such as short quiz, test, etc., make-up assessment will not be provided to students. The students will score "0" for the assessment work concerned.")

Coursework Components (50%)

Pls mark your diary. No reminder will be given

↓

Time	Item	Scope	Percentage
Tests			
Wk 6 (23 Feb.)	**Test 1**	everything taught in Wk 1- 5	15%
Wk 11 (30 Mar.)	**Test 2**	everything taught in Wk 6 -10	15%
Assignments			
Wk 13 (15 Apr.)	**Mini-Project**		12.5%
	4 assignments		5%
Wk 9 (16 Mar.)	**Mini-Project Progress**		2.5%

Mini-project progress

- Hand in mini-project progress with
  - Realistic hierarchical structures
  - Realistic animation
- The hierarchical structures should be much more sophisticated than the robotic hand in OpenGL Ex 2
- You do not need to use the same hierarchical structures for the final mini project
- Animation must use glutIdle function and must be realistic with effects such as acceleration and deceleration, and at least one accelerate-then-decelerate motion. It should start automatically. There should be no user/keyboard input required
- See mini project for the format but no need to hand in report

Mathematical Background

- You should have the mathematical background below:
  - 3D coordinate systems in Euclidean coordinates and polar coordinates
  - Basic matrix and vector arithmetic
  - Calculation of determinant
  - Scalar (dot) product: how to calculate and its physical meanings
  - Vector (cross) product: how to calculate and its physical meanings
  - Concepts of partial derivatives
- Please consult any standard text in Linear Algebra

Non-standard mathematical notation used

- **|N|** is normally used to denote the magnitude of vector **N** and is a scalar. In this course, **|N|** is sometimes also used to denote "normalize the vector **N** to a unit vector"

  e.g. The light source is at (3, 3, 3) and the surface point is at (0, 0, 0). The unit lighting vector

$$L = |(3,3,3) - (0,0,0)| = \left(\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}\right)$$

3D Object Representation

Intended Learning Outcomes

- Understand the concept of standard graphics object
- Able to mathematically manipulate and program in OpenGL two types of planar representation: tables and mesh
- Distinguish the concepts of parametric and non-parametric equations and understand the advantage of using the former in computer graphics
- Able to mathematically manipulate and program in OpenGL quadrics and super-quadrics

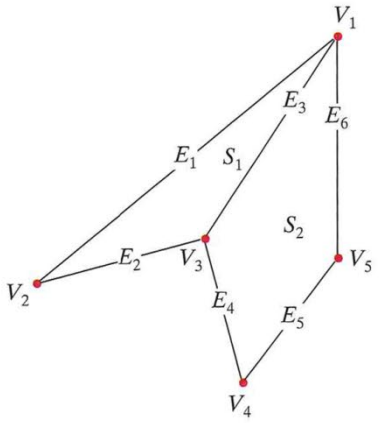
Standard Graphics Object

- **standard graphics object = a set of (planar) polygons**

- Complicated objects can be described by using many polygons
- Dedicated hardware are designed to speed up rendering of standard graphics objects.

Two methods for storing standard graphics objects

- Method 1: use table (vertex, edge, polygon, attribute)



VERTEX TABLE	
V_1:	x_1, y_1, z_1
V_2:	x_2, y_2, z_2
V_3:	x_3, y_3, z_3
V_4:	x_4, y_4, z_4
V_5:	x_5, y_5, z_5

EDGE TABLE	
E_1:	V_1, V_2
E_2:	V_2, V_3
E_3:	V_3, V_1
E_4:	V_3, V_4
E_5:	V_4, V_5
E_6:	V_5, V_1

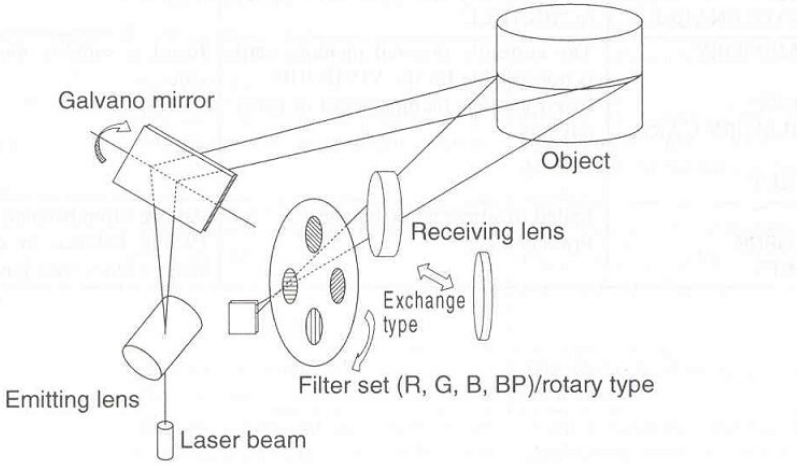
SURFACE-FACET TABLE	
S_1:	E_1, E_2, E_3
S_2:	E_3, E_4, E_5, E_6

Geometric data-table representation for two adjacent polygon surface facets, formed with six edges and five vertices.

Method 2: Quadrilateral Mesh

- A n x m array of vertex positions (X, Y, Z)
- Represent a surface of (n-1) x (m-1) quadrilaterals
- Each quadrilateral may be further subdivided into two triangles
- Two ways to obtain data in the mesh
  - Way 1: By specifying an equation
  - Way 2: By 3D digitizer

3-D scanner



3D data obtained by triangulation

3D scanner is available in CityU Library:

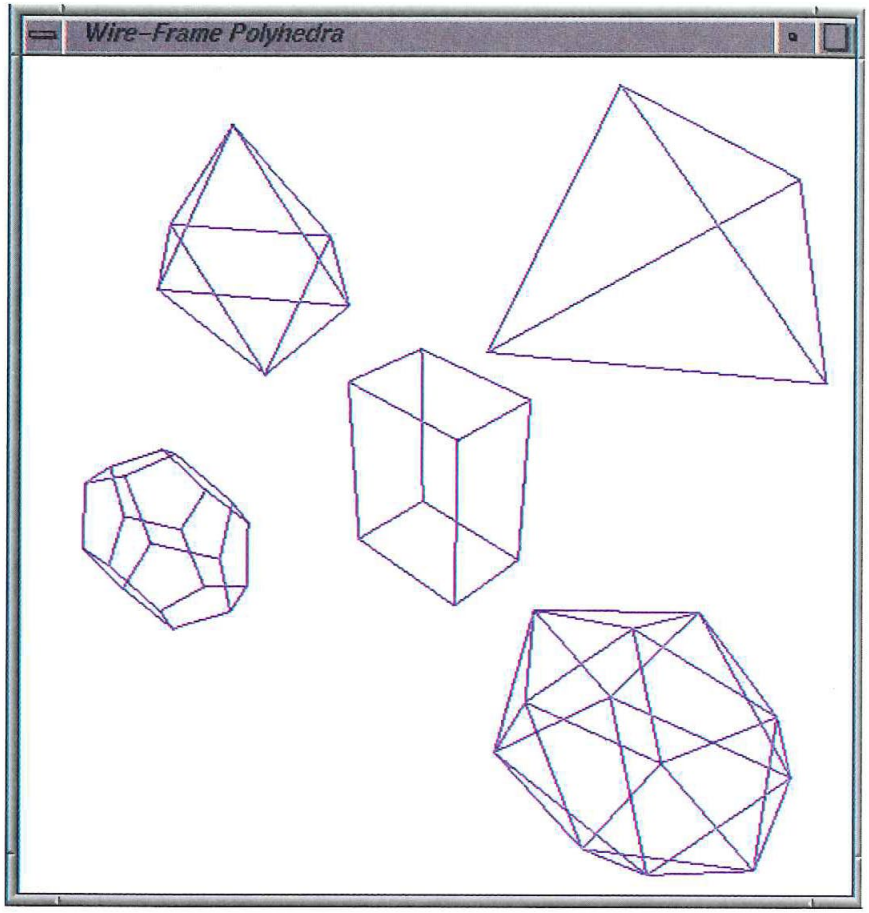
http://www.cityu.edu.hk/lib/about/facility/3d/index.htm

Glut functions

- *glutWire* as wireframe
- *glutSolid* as fill area polygon patches

glutSolidCube (edgelength);

- Tetrahedron, Cube, Octahedron, Dodecahedron, Icosahedron



A perspective view of the five GLUT polyhedra, scaled and positioned within a display window by procedure `displayWirePolyhedra`.

Mathematical Concepts for Plane

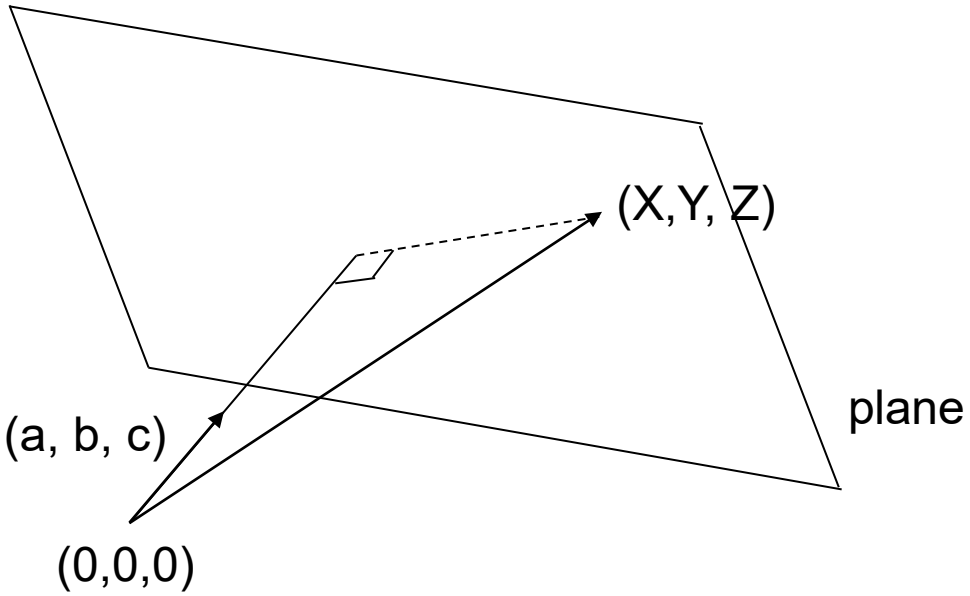
- Plane

$$aX + bY + cZ + d = 0$$

- Only 3 parameters define the plane, the fourth can be set to 1 or 0
- d = 1 does not pass through (0, 0, 0)
- d = 0 pass through (0, 0, 0)

Rewritten as

$$(a, b, c) \cdot (X, Y, Z) = -d$$



If (a, b, c) is a unit vector, then $-d$ is the distance of the plane from the origin

Normal

- Important concept in lighting and shading
- Normal vector
  - vector ⊥ to the plane
  - “Unit vector” - L2 norm is 1.
- Solving for Normal
  - Normal **n** = (a, b, c)
  - Select 3 vertices on the plane **V1**, **V2**, **V3**

$$\mathbf{n} = (V2 - V1) \times (V3 - V1)$$

Distinguishing "Inside" from "Outside"

- Useful for "collision detection"

- Use (a, b, c)

aX+bY+cZ+d	> 0	Outside
	= 0	On the plane
	< 0	Inside

- Use **V1**, **V2**, **V3**

V1, **V2**, **V3**	selected CCW	=>	Outside
	CW	=>	Inside

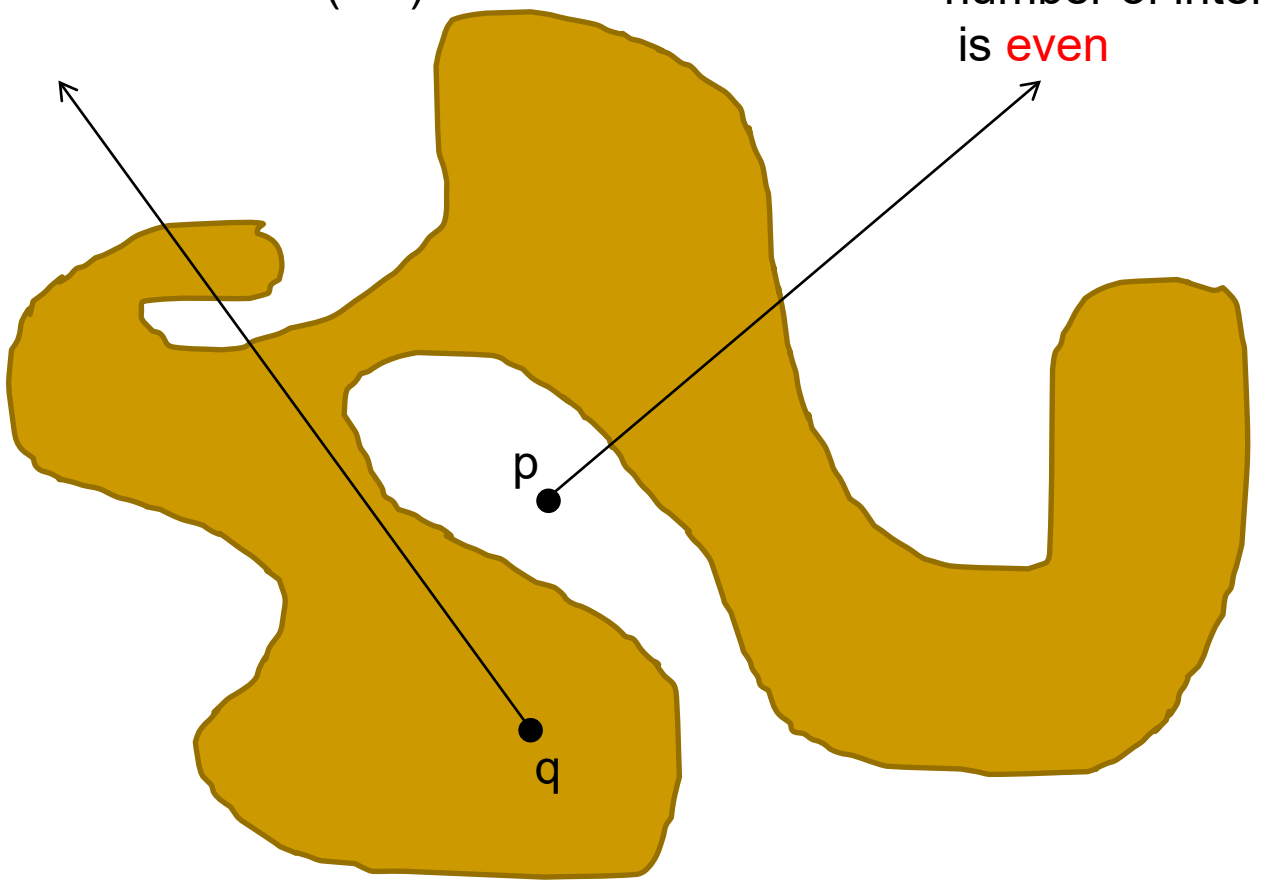
Inside-Outside Test

- To determine whether a pixel ***p*** is inside or outside an object ***S***
- Send a ray $p + t\,v$ which starts at the pixel, t is a scalar, ***v*** is an arbitrary direction vector
- Find all non-degenerate† intersections between the ray and ***S***
- *If the number of intersections is odd (even),* ***p*** *is inside (outside)* ***S***
- It is not easy to check non-degenerate intersections. One can solve this problem by sending out n rays in random directions and then use majority voting

† a degenerate intersection is one which the ray grazes the surface

Point q is inside as the number of intersections (= 3) is odd

Point p is outside as the number of intersections (= 2) is even



The yellow object is depicted as a 2D object but the technique can be applied to any n-dimensional object (n > 2)

Superquadrics

- 2D QUADRICS (conic section)

$$aX^2 + bY^2 + cXY + dX + eY + f = 0$$

- 3D QUADRICS

$$aX^2 + bY^2 + cZ^2 + dXY + eXZ + fYZ + gX + hY + iZ + k = 0$$

In 2D,

- Circle $X^2 + Y^2 = r^2$
- Ellipse $\left(\frac{X}{a}\right)^2 + \left(\frac{Y}{b}\right)^2 = 1$
- Parabola $Y^2 = 4aX$
- Hyperbola $X^2 - Y^2 = r^2$

In 3D

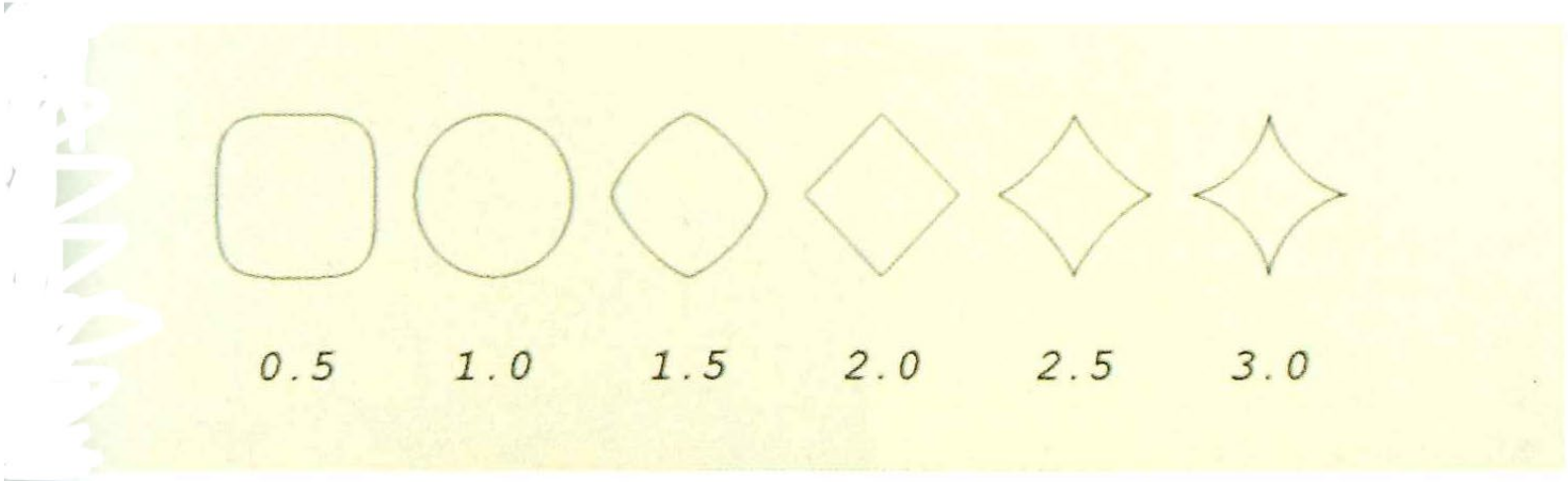
- Sphere $X^2 + Y^2 + Z^2 = r^2$
- Ellipsoid $\left(\frac{X}{a}\right)^2 + \left(\frac{Y}{b}\right)^2 + \left(\frac{Z}{c}\right)^2 = 1$
- Paraboloid ?
- Hyperboloid ?

(ans. to be discussed in tut.)

“Super”-quadrics

- Introduce to additional parameters s1 and s2
- Allow continuous transformation from “circle” to “square” (Idiom)
- Example (2D) “Super-ellipse”

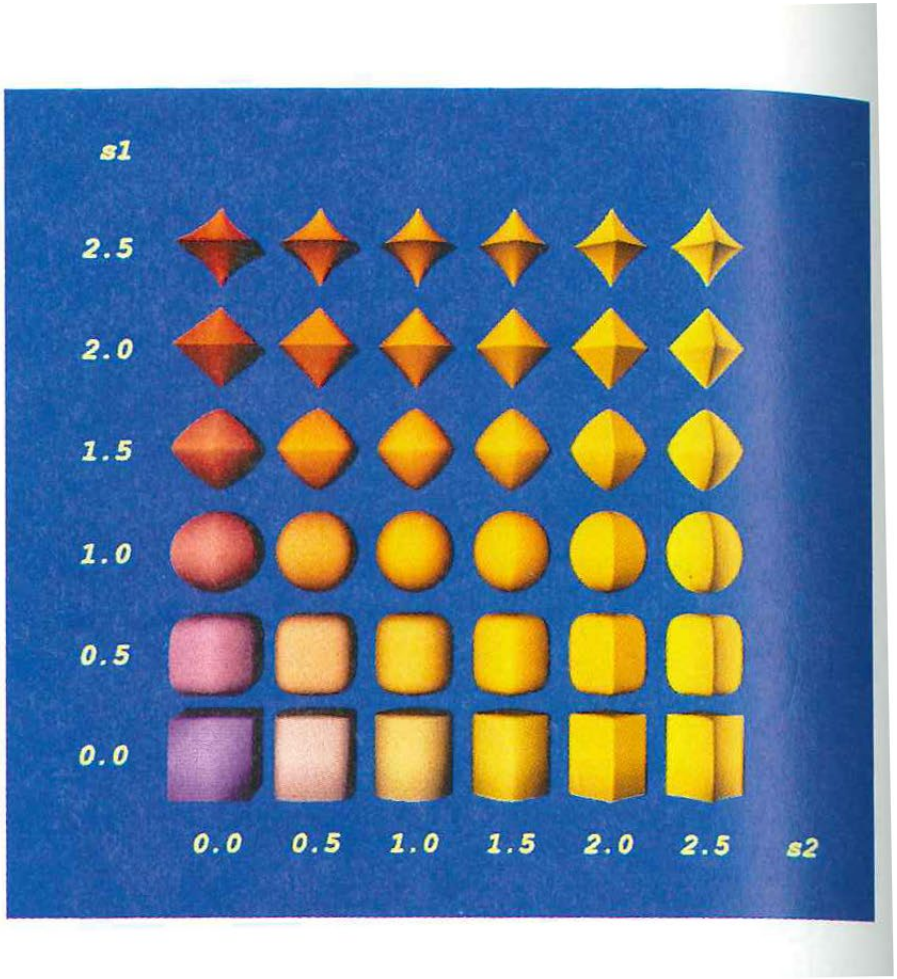
$$\left(\frac{X}{a}\right)^{\frac{2}{s}} + \left(\frac{Y}{b}\right)^{\frac{2}{s}} = 1$$



Superellipses plotted with values for parameter s ranging from 0.5 to 3.0 and with $r_x = r_y$.

Super-ellipsoid

$$\left[\left(\frac{X}{r_x}\right)^{2/s_2} + \left(\frac{Y}{r_y}\right)^{2/s_2}\right]^{s_2/s_1} + \left(\frac{Z}{r_z}\right)^{2/s_1} = 1$$



Superellipsoids plotted with values for parameters s_1 and s_2 ranging from 0.0 to 2.5 and with $r_x = r_y = r_z$.

Non-parametric and Parametric forms

- Non-parametric form
  - Z = f(X, Y) or f(X, Y, Z) = 0
  - Used in mathematics

- Parametric form
  - Introduced two additional parameters u, v
  - X = f1 (u, v) Y = f2 (u, v) Z = f3 (u, v)
  - Used in CG

Parametric form of the super-ellipsoid

$$\left[\left(\frac{X}{r_x}\right)^{2/s_2} + \left(\frac{Y}{r_y}\right)^{2/s_2}\right]^{s_2/s_1} + \left(\frac{Z}{r_z}\right)^{2/s_1} = 1$$

Non-parametric

$$X = r_x \cos^{s_1}\phi \cos^{s_2}\theta$$

$$Y = r_y \cos^{s_1}\phi \sin^{s_2}\theta$$

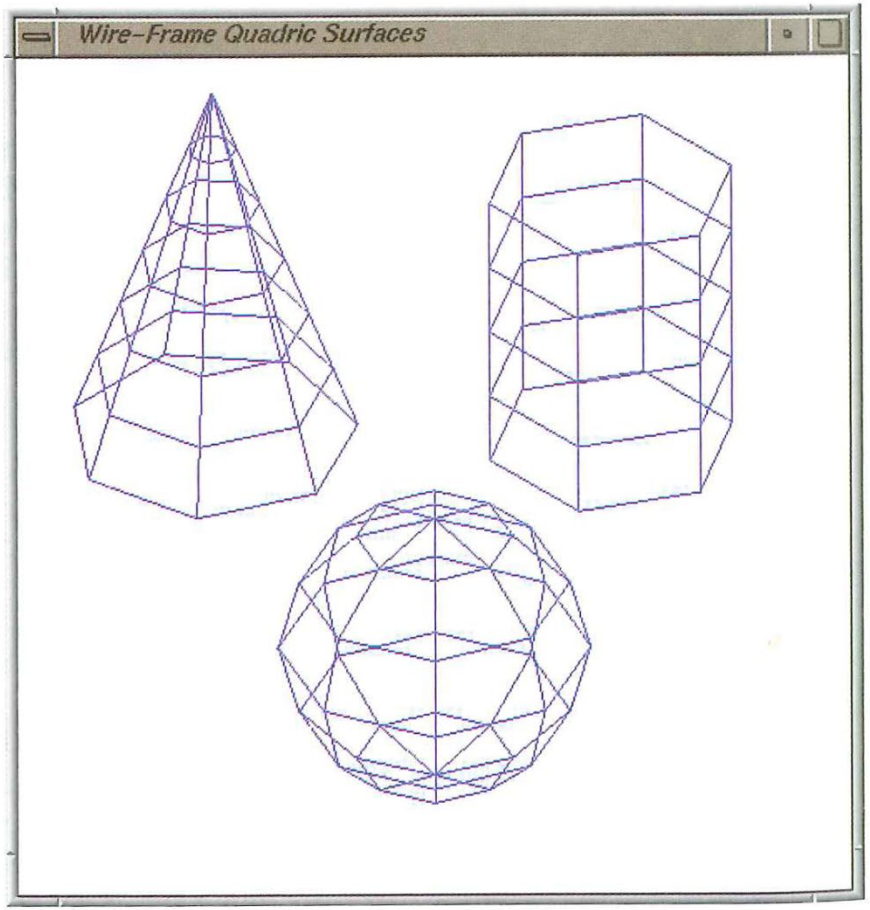
$$Z = r_z \sin^{s_1}\phi$$

Parametric

OpenGL functions

- Does not have superquadrics function
- Can display sphere, cone, cylinder
- Quadrilateral mesh

glutWireSphere (r, nLongitudes, nLatitudes)



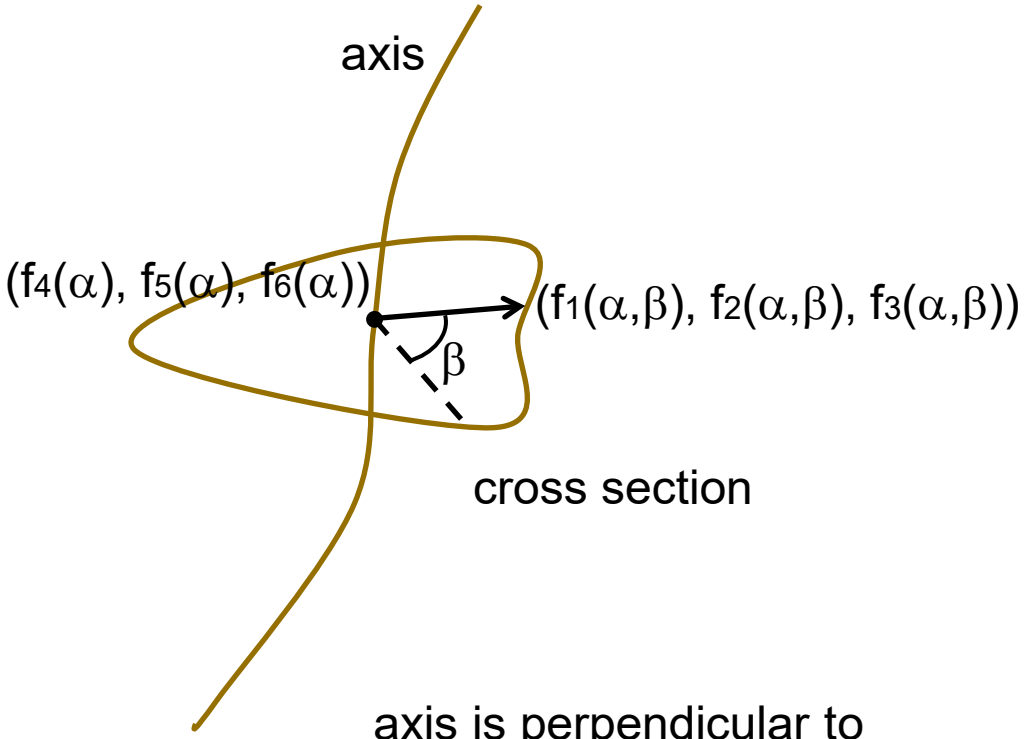
Display of a GLUT sphere, GLUT cone, and GLU cylinder, positioned within a display window by procedure `wireQuadSurfs`.

Generation of complicated shapes

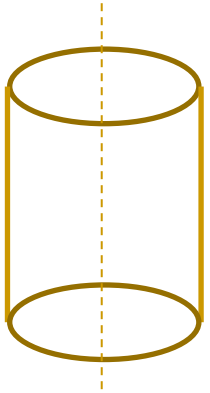
- Complicated shapes can be generated using quadrilateral mesh and parametric form
- Two examples are
  - Generalized Cylinder
  - Generalized Symmetry

Generalized Cylinder

real life example



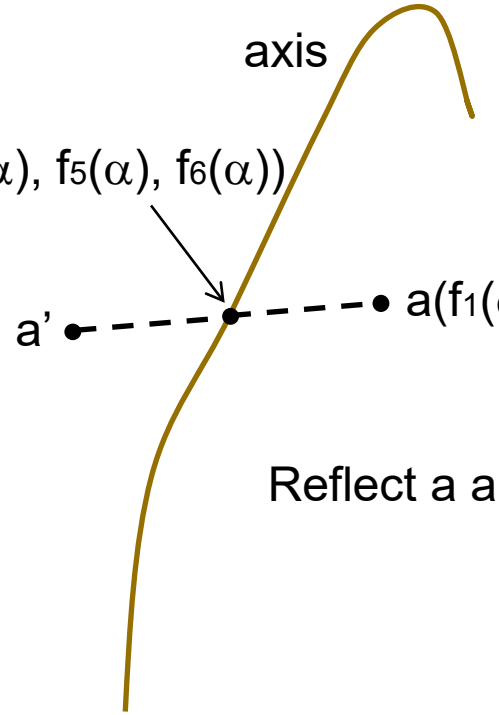
primordial shape



quadrilateral mesh parameterized by α and β

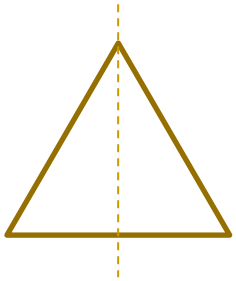
Generalized Reflectional Symmetry

real life example



Reflect a about b to get a'

primordial shape



quadrilateral mesh parameterized by α and β, with β varying linearly from a to a'

3D Modelling Transformations

Intended Learning Outcomes

- Understand the use of homogeneous coordinates
- Learn different types of 3D transforms and the concept of composite transform
- Able to use coordinate transform to switch between one coordinate frame to another
- Able to use OpenGL to implement coordinate transform

Homogeneous coordinates

- Represent a n-dimensional entity as a (n+1)-dimensional entity
- Allow all linear transforms to be expressed as matrix multiplications; eliminate matrix addition/subtraction

Linear Transform

- $\mathbf{P}_2 = \mathbf{M}_1\mathbf{P}_1 + \mathbf{M}_2$

  $\mathbf{P}_1$ n-dimensional points (n x 1 column vector)
  $\mathbf{P}_2$ Transformed n-dimensional points (n x 1 column vector)
  $\mathbf{M}_1$ n x n square transform matrix
  $\mathbf{M}_2$ n x 1 column transform vector

- Homogeneous coordinates allow us to express the multiplicative term $\mathbf{M}_1$ and the addition term $\mathbf{M}_2$ in a common 4 x 4 matrix. This is achieved by adding one dimension w.

3D Point

A 3D point (n = 3) can be expressed as

- (X, Y, Z) Euclidean coordinates
- (X_W, Y_W, Z_W, W) Homogeneous coordinates

$$X = \frac{X_W}{W} \quad Y = \frac{Y_W}{W} \quad Z = \frac{Z_W}{W}$$

- W can be any non-zero value.

3D Translation

- Euclidean

$\mathbf{P}_2 = \mathbf{P}_1 + \mathbf{T}(t_X, t_Y, t_Z)$

$$\begin{pmatrix} X_2 \\ Y_2 \\ Z_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X_1 \\ Y_1 \\ Z_1 \end{pmatrix} + \begin{pmatrix} t_X \\ t_Y \\ t_Z \end{pmatrix}$$

- Homogeneous

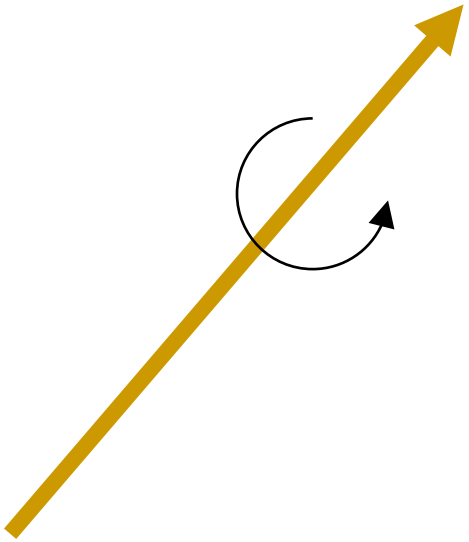
$\mathbf{P}_2 = \mathbf{T}(t_X, t_Y, t_Z)\mathbf{P}_1$

$$\begin{pmatrix} X_2 \\ Y_2 \\ Z_2 \\ W_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & t_X \\ 0 & 1 & 0 & t_Y \\ 0 & 0 & 1 & t_Z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X_1 \\ Y_1 \\ Z_1 \\ W_1 \end{pmatrix}$$

Note : $W_2 = W_1 = 1$

3D Rotations

- Rotation about an axis

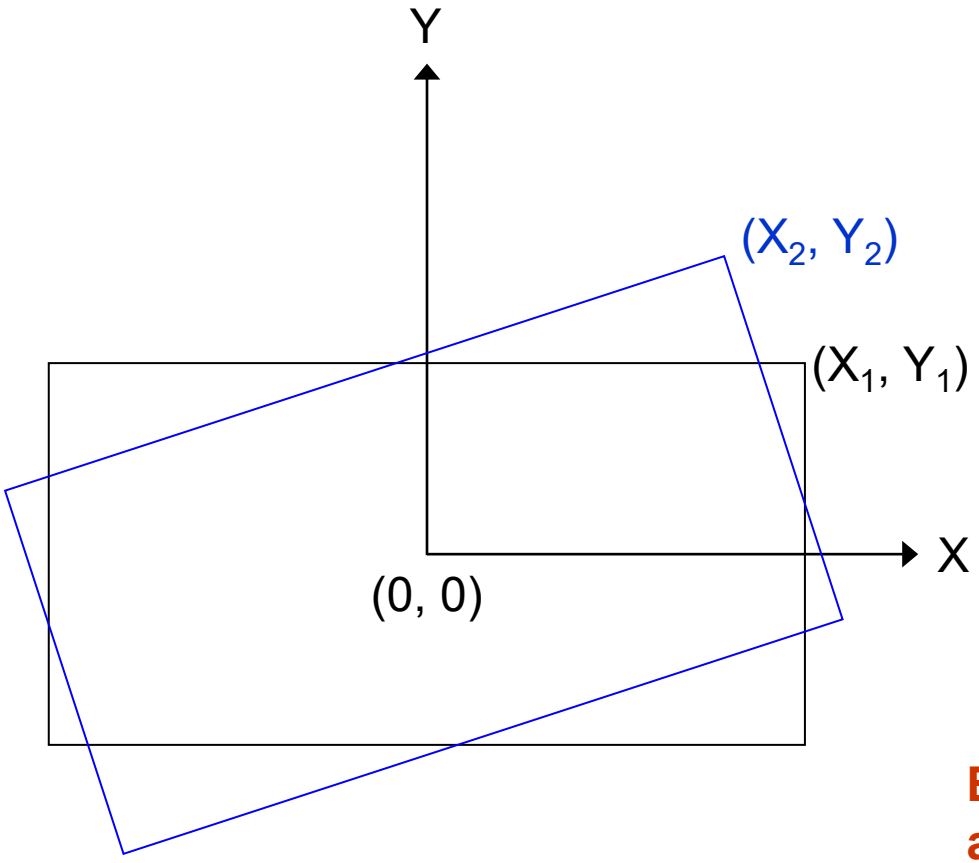


CCW $\Rightarrow$ POSITIVE rotation

Right Hand Rule

2D Rotations about the origin

- About a common coordinate system X-Y



$$\begin{pmatrix} X_2 \\ Y_2 \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} X_1 \\ Y_1 \end{pmatrix}$$

Equivalent to rotation about Z axis, which is pointing out of paper

Rotation about Z

- Euclidean

  $\mathbf{P}_2=\mathbf{R}_Z(\theta)\mathbf{P}_1$

$$\begin{pmatrix} X_2 \\ Y_2 \\ Z_2 \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X_1 \\ Y_1 \\ Z_1 \end{pmatrix}$$

- Homogeneous

  $\mathbf{P}_2=\mathbf{R}_Z(\theta)\mathbf{P}_1$

$$\begin{pmatrix} X_2 \\ Y_2 \\ Z_2 \\ W_2 \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X_1 \\ Y_1 \\ Z_1 \\ W_1 \end{pmatrix}$$

Rotation about X

- Euclidean

  $\mathbf{P}_2=\mathbf{R}_X(\theta)\mathbf{P}_1$

$$\begin{pmatrix} X_2 \\ Y_2 \\ Z_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} X_1 \\ Y_1 \\ Z_1 \end{pmatrix}$$

- Homogeneous

  $\mathbf{P}_2=\mathbf{R}_X(\theta)\mathbf{P}_1$

$$\begin{pmatrix} X_2 \\ Y_2 \\ Z_2 \\ W_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X_1 \\ Y_1 \\ Z_1 \\ W_1 \end{pmatrix}$$

Rotation about Y

- Euclidean

  $\mathbf{P}_2=\mathbf{R}_Y(\theta)\mathbf{P}_1$

$$\begin{pmatrix} X_2 \\ Y_2 \\ Z_2 \end{pmatrix} = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix} \begin{pmatrix} X_1 \\ Y_1 \\ Z_1 \end{pmatrix}$$

- Homogeneous

  $\mathbf{P}_2=\mathbf{R}_Y(\theta)\mathbf{P}_1$

$$\begin{pmatrix} X_2 \\ Y_2 \\ Z_2 \\ W_2 \end{pmatrix} = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X_1 \\ Y_1 \\ Z_1 \\ W_1 \end{pmatrix}$$

Scaling about the origin

- Euclidean

  $\mathbf{P}_2=\mathbf{S}(s_X, s_Y, s_Z)\mathbf{P}_1$

$$\begin{pmatrix} X_2 \\ Y_2 \\ Z_2 \end{pmatrix} = \begin{pmatrix} s_X & 0 & 0 \\ 0 & s_Y & 0 \\ 0 & 0 & s_Z \end{pmatrix} \begin{pmatrix} X_1 \\ Y_1 \\ Z_1 \end{pmatrix}$$

- Homogeneous

  $\mathbf{P}_2=\mathbf{S}(s_X, s_Y, s_Z)\mathbf{P}_1$

$$\begin{pmatrix} X_2 \\ Y_2 \\ Z_2 \\ W_2 \end{pmatrix} = \begin{pmatrix} s_X & 0 & 0 & 0 \\ 0 & s_Y & 0 & 0 \\ 0 & 0 & s_Z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X_1 \\ Y_1 \\ Z_1 \\ W_1 \end{pmatrix}$$

Reflection about the X-Y plane

- Euclidean

  $\mathbf{P}_2 = \mathbf{R}\mathbf{F}_Z\mathbf{P}_1$

$$\begin{pmatrix} X_2 \\ Y_2 \\ Z_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix} \begin{pmatrix} X_1 \\ Y_1 \\ Z_1 \end{pmatrix}$$

- Homogeneous

  $\mathbf{P}_2 = \mathbf{R}\mathbf{F}_Z\mathbf{P}_1$

$$\begin{pmatrix} X_2 \\ Y_2 \\ Z_2 \\ W_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X_1 \\ Y_1 \\ Z_1 \\ W_1 \end{pmatrix}$$

Shearing about the Z axis

- Euclidean

  $\mathbf{P}_2 = \mathbf{Sh}_z(a,b)\mathbf{P}_1$

$$\begin{pmatrix} X_2 \\ Y_2 \\ Z_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X_1 \\ Y_1 \\ Z_1 \end{pmatrix}$$

- Homogeneous

  $\mathbf{P}_2 = \mathbf{Sh}_Z(a,b)\mathbf{P}_1$

$$\begin{pmatrix} X_2 \\ Y_2 \\ Z_2 \\ W_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & a & 0 \\ 0 & 1 & b & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X_1 \\ Y_1 \\ Z_1 \\ W_1 \end{pmatrix}$$

Affine Transform

$$\begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

- a_{ij} and b_i are constants.
- a linear transformation
- // lines are transformed to // lines
- Translation, rotation, scaling, reflection, shearing are special cases
- Any affine transform can be expressed as composition of the above 5 transforms

Composite Transformation

- A number of (relative) transformations applied in sequence
- Models the complex movement of an object in the world coordinate system
- The transformation is pre-computed where possible.
- In practice, ONLY the final 4 x 4 composite transformation needs to be stored.

E.g. 1 Rotation about an axis // to X axis.

- Let (X_f, Y_f, Z_f) be a point on the axis. The composite rotation is

$$\mathbf{P}_2 = \mathbf{T}^{-1}\mathbf{R}_x(\theta)\mathbf{T}\,\mathbf{P}_1$$

$$\mathbf{T} = \mathbf{T}(-X_f, -Y_f, -Z_f)$$

- For the composite transformation

$$\begin{pmatrix} 1 & 0 & 0 & X_f \\ 0 & 1 & 0 & Y_f \\ 0 & 0 & 1 & Z_f \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -X_f \\ 0 & 1 & 0 & -Y_f \\ 0 & 0 & 1 & -Z_f \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Only the product

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & -Y_f\cos\theta + Z_f\sin\theta + Y_f \\ 0 & \sin\theta & \cos\theta & -Y_f\sin\theta - Z_f\cos\theta + Z_f \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

is stored

E.g. 2 Scaling about (X_f, Y_f, Z_f)

- $\mathbf{P}_2 = \mathbf{T}^{-1}\mathbf{S}(s_X, s_Y, s_Z)\mathbf{T}\,\mathbf{P}_1$

$\mathbf{T} = \mathbf{T}(-X_f, -Y_f, -Z_f)$

Similarly, only the final 4 x 4 composite transformation is stored

Concept

- A composite transformation may have two physical meaning:
- Either

  It represents a physical action
- Or

  It represents a change of coordinate system

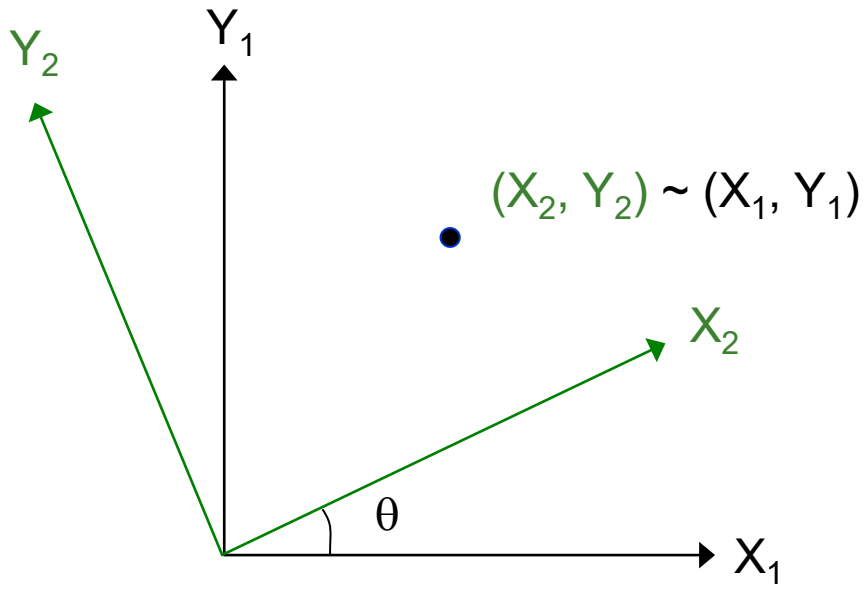
3 Kinds of Coordinate System in CG

- Each object defined in their own natural coordinate system – Modelling coordinate system (MC)
- All objects being placed in a common world coordinate system (WC)
- For correct viewing by a camera, objects need to be expressed in a common viewer or camera coordinate system (VC, CC)

MC → WC → VC/CC

A point in two different coordinate sy.

- The SAME point has DIFFERENT coordinates in DIFFERENT coordinate systems



$$\mathbf{P}^{(2)} = \mathbf{R}(-\theta)\mathbf{P}^{(1)}$$

$$\begin{pmatrix} X_2 \\ Y_2 \end{pmatrix} = \begin{pmatrix} \cos(-\theta) & -\sin(-\theta) \\ \sin(-\theta) & \cos(-\theta) \end{pmatrix} \begin{pmatrix} X_1 \\ Y_1 \end{pmatrix}$$

- $\mathbf{P}^{(i)}$ A point in coordinate system i
- $\mathbf{M}_{j \leftarrow i}$ 4 x 4 transformation that transforms a point in coordinate system i to coordinate system j

- $\mathbf{P}^{(j)} = \mathbf{M}_{j \leftarrow i} \mathbf{P}^{(i)}$

- Rule 1 for computing $\mathbf{M}_{j\leftarrow i}$:

  $\mathbf{M}_{j\leftarrow i}$ is the inverse of the transformation that takes the **i**$^{\mathbf{th}}$ coordinate system frame as if it is an object to the **j**$^{\mathbf{th}}$ coordinate system frame position, all the time using the **i**$^{\mathbf{th}}$ coordinate system as the reference coordinate system

  As $\mathbf{M}_{j\leftarrow i} = \mathbf{M}_{i\leftarrow j}^{-1}$, we have the alternative rule:

- Alternative rule (rule 2) for computing $\mathbf{M}_{j\leftarrow i}$:

  $\mathbf{M}_{j\leftarrow i}$ is the transformation that takes the **j**$^{\mathbf{th}}$ coordinate system frame as if it is an object to the **i**$^{\mathbf{th}}$ coordinate system frame position, all the time using the **j**$^{\mathbf{th}}$ coordinate system as the reference coordinate system

- which rule to use depends on which coordinate system is easier to get on hand

$M_{j\leftarrow i}$ is the INVERSE of the transformation that takes the ith coordinate system frame as if it is an object to the jth coordinate system frame

Proof

Suppose we have two coordinate systems x_i-y_i-z_i and x_j-y_j-z_j. Treat x_i-y_i-z_i and x_j-y_j-z_j as two objects that consist of two sets of points, both defined in the x_i-y_i-z_i coordinate system. Let

$x_i = (1, 0, 0)^T \rightarrow x_j = (a_{11}, a_{21}, a_{31})^T + (t_x, t_y, t_z)^T$
$y_i = (0, 1, 0)^T \rightarrow y_j = (a_{12}, a_{22}, a_{32})^T + (t_x, t_y, t_z)^T$
$z_i = (0, 0, 1)^T \rightarrow z_j = (a_{13}, a_{23}, a_{33})^T + (t_x, t_y, t_z)^T$

where all the coordinates are defined in the x_i-y_i-z_i coordinate system. $\rightarrow$ means "corresponds to".

The transformation T that transforms the three points x_i, y_i, z_i to x_j, y_j, z_j in the x_i-y_i-z_i coordinate system is thus

$$\mathbf{T} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & t_x \\ a_{21} & a_{22} & a_{23} & t_y \\ a_{31} & a_{32} & a_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

However, it can also be interpreted as changing from coordinate system j to coordinate system i. Thus

$\mathbf{P}^{(j)} = (1, 0, 0)^T \rightarrow \mathbf{P}^{(i)} = (a_{11}, a_{21}, a_{31})^T + (t_x, t_y, t_z)^T$
$\mathbf{P}^{(j)} = (0, 1, 0)^T \rightarrow \mathbf{P}^{(i)} = (a_{12}, a_{22}, a_{32})^T + (t_x, t_y, t_z)^T$
$\mathbf{P}^{(j)} = (0, 0, 1)^T \rightarrow \mathbf{P}^{(i)} = (a_{13}, a_{23}, a_{33})^T + (t_x, t_y, t_z)^T$

Since any arbitrary $\mathbf{P}^{(j)}$ can be written as $\lambda_1(1,0,0)^T + \lambda_2(0,1,0)^T + \lambda_3(0,0,1)^T$, where $\lambda_1, \lambda_2, \lambda_3$ are constants, it follows that

$$\mathbf{M}_{i\leftarrow j} = \mathbf{T}$$

Since $\mathbf{M}_{j\leftarrow i} = \mathbf{M}_{i\leftarrow j}^{-1}$,

$$\mathbf{M}_{j\leftarrow i} = \mathbf{T}^{-1}$$

This gives the rule
$\mathbf{M}_{j\leftarrow i}$ is the INVERSE of the transformation that takes the ith coordinate system frame as if it is an object to the jth coordinate system frame

OpenGL Geometric Transformations

- 4 x 4 translation matrix

  glTranslatef (tx, ty, tz);
- 4 x 4 rotation matrix

  glRotatef (theta, vx, vy, vz);
- 4 x 4 scaling matrix

  glScalef (sx, sy, sz);
- 4 x 4 reflection matrix

  glScalef (1, 1, -1); // reflection about Z axis
- 4 x 4 shearing matrix

  glMultMatrixf (matrix); // matrix is a 16 element
  // matrix in column-major order

OpenGL Matrix Operations

- Calls the current matrix, responsible for geometrical transformation

  glMatrixMode (GL_MODELVIEW);

  (do not confuse with *glMatrixMode (GL_PROJECTION)*, which is responsible for projection transformation)

- Assign identity matrix to current matrix

  glLoadIdentity ( );

- Current matrix is modified by (relative) transformations
  - E.g. glTranslatef, glScalef, glRotatef …
  - The meaning of the relative transformations may either be physical action or coordinate transformations
- Current matrix are *postmultiplied. Last operation specified is first operation performed, like a LIFO stack*

Let **C** be the composite matrix

- Example 1

  glMatrixMode (GL_MODELVIEW)

  glLoadIdentity ( ); // **C** = identity matrix

  glTranslatef (-25, 50, 25); // **C** = **T**(-25,50,25)

  glRotatef (45, 0, 0, 1); // **C** = **T** (-25,50,25)$\mathbf{R}_Z(45^o)$

  glScalef (1, 2, 1); // **C** = **T** (-25,50,25)$\mathbf{R}_Z(45^o)$ **S**(1,2,1)

- Example 2

glMatrixmode (GL_MODELVIEW)

glLoadIdentity ( );

glScalef (1, 2, 1);

glRotatef (45, 0, 0, 1);

*glTranslatef (-25, 50, 25); // **C** = **S**(1,2,1)$\mathbf{R}_Z$(45^o)**T**(-25,50,25)*

Note: the order of the transformation is important

OpenGL Matrix Stacks

- OpenGL has a stack for storing the relative transformations
- Stack is a LIFO data structure
- Stores intermediate results

- Push the current matrix into the stack
  glPushMatrix ( );
- Pop the current matrix from the stack
  glPopMatrix ( );

Note: Very useful for modelling hierarchical structures

- Example

```
glMatrixMode (GL_MODELVIEW)
glLoadIdentity ( );              //  MV = identity matrix

glTranslatef (-25, 50, 25);   // MV = T(-25,50,25)
glRotatef     (45,   0,  0, 1); //  MV = T (-25,50,25)Rz(45o)
glPushMatrix ( );                 // MV is pushed to the stack
glScalef      (1, 2, 1);      // MV = T (-25,50,25)Rz(45o) S(1,2,1)
glTranslatef  (0, 0, 10);    // MV = T Rz(45o)S(1,2,1) T(0, 0, 10)
glPopMatrix   ( );
                                  //  MV = T (-25,50,25)Rz(45o)
```

3D Hierarchical Modelling

Intended Learning Outcomes

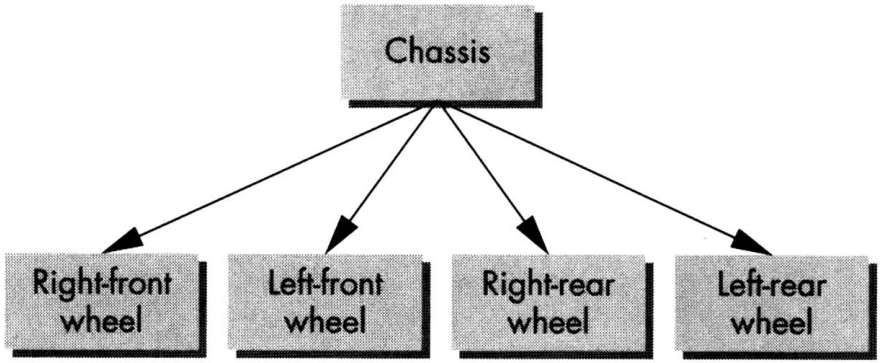
- Understand the need of hierarchical structuring for building articulated 3D objects
- Able to compute the relative coordinate transform between component parts
- Able to represent an articulated 3D object as a hierarchical structure using OpenGL

Problem:

- Given a large number of graphics models which form parts of a whole object, it is cumbersome to animate each part by individual commands

Example : Animate a car moving at a speed of 20 miles and in direction (2, 3, 4)

```
main ()
{
      float s     = 20.0; /* speed */
      float d[3] = {2.0, 3.0, 4.0}; /* direction */

      draw_chassis (s, d);

      draw_right_front_wheel (s, d);
      draw_left_front_wheel   (s, d);
      draw_right_rear_wheel  (s, d);
      draw_left_rear_wheel    (s, d);

}
```



Tree with directed edge

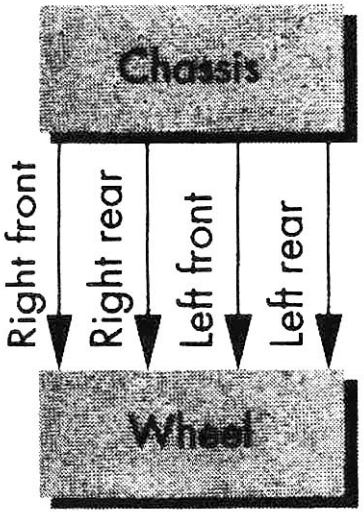
Bad Programming - Redundancy: the 4 draw wheel functions can be replaced by one function

Introduction of hierarchical structures

- Use *relative transformation* to link the movements of different parts
- Use a single function for a unique (single) part

Directed Acyclic Graph (DAG)

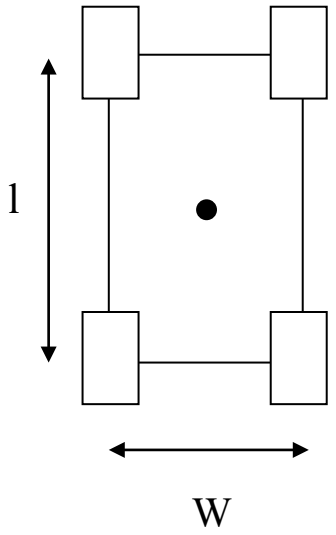
- DAG is a graph with directed arc but no cycle
- It is a tree but additional allows more than one arc from one node to another node



Revised program

```
main ()
{
      float s = 20.0;
      float d[3] ={2.0, 3.0, 4.0};
      float w = 2.0, l = 4.0;        // width and length of the car

      draw_chass (s, d);

      glTranslatef ( w/2 , l/2, 0 );    // position the right front wheel
      draw_wheel (s, d);
     glTranslatef ( -w,     0,  0 );    // position the left front wheel
     draw_wheel (s, d);
     glTranslatef (  0,     -l ,  0 );    // position the left rear wheel
     draw_wheel (s, d);
     glTranslatef ( w,     0,   0  );     // position the right rear wheel
     draw_wheel (s, d);
}
```

Let the initial coordinate system be the centroid of the car



We can make it more systematic by formally introducing coordinate system change, which we do below

Moving a Robot Arm – a 3 level hierarchy

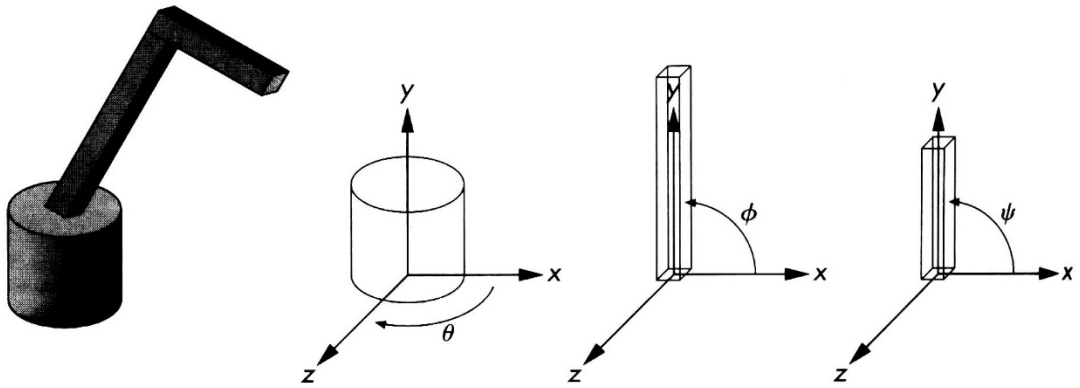
Parts : **base B** (cylinder),
lower arm La (rectangular box)
upper arm Ua (rectangular box)

Arm has 3 degree of freedom:

B rotate about Y by θ

La rotate about Z by ϕ

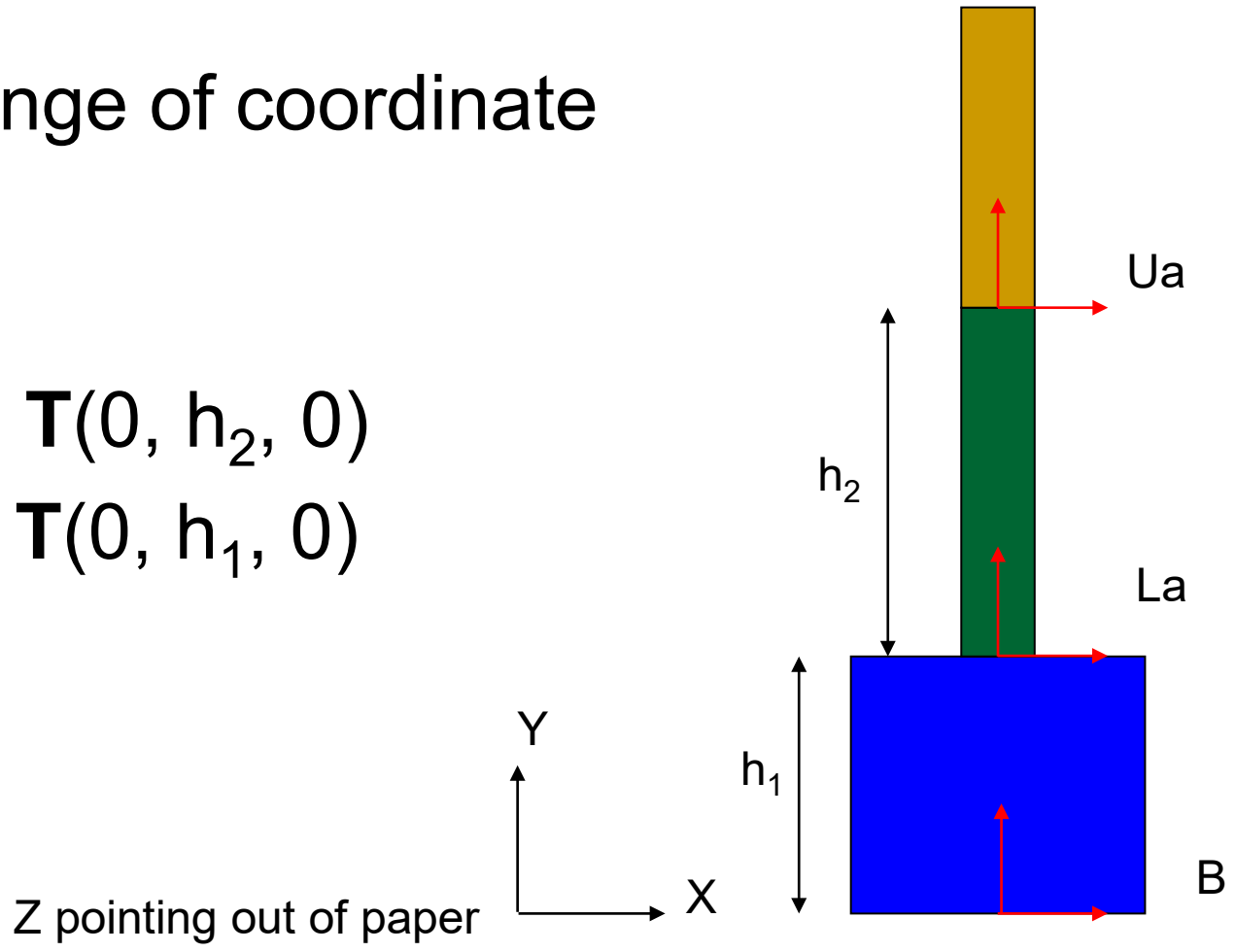
Ua rotate about Z by ψ



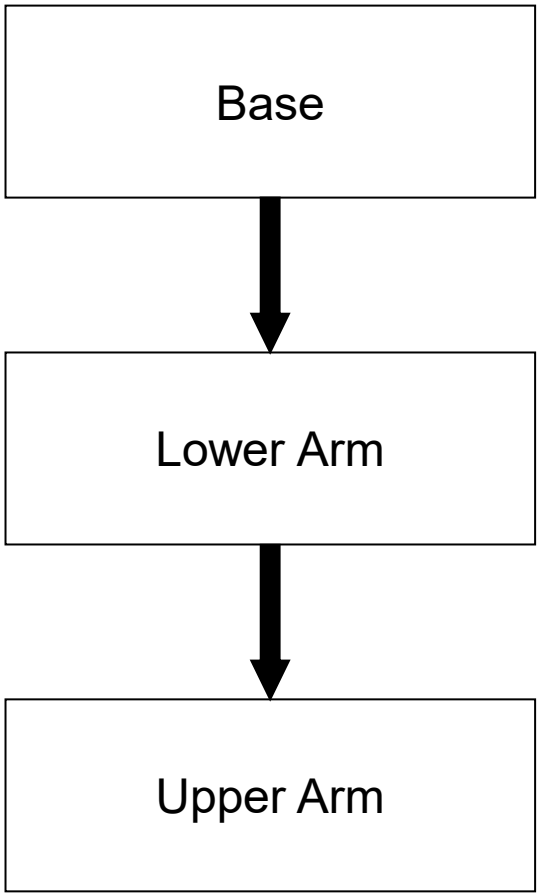
Relative Coordinate Transformations

- Use Change of coordinate system:

- $\mathbf{M}_{La \leftarrow Ua} = \mathbf{T}(0, h_2, 0)$
- $\mathbf{M}_{B \leftarrow La} = \mathbf{T}(0, h_1, 0)$



DAG



Write a program to …

- Rotate the robot arm about its base by θ, then about its lower arm by ϕ, then about its upper arm by ψ
- when rotating the whole arm, everything should move; but when rotating the lower arm, only it and the upper arm should move; when rotating the upper arm, only the upper arm should move.
- Solve this using a hierarchy concept

Program

robot_arm ()
{

glRotatef (theta, 0.0, 1.0, 0.0); // $\mathbf{R}_y(\theta)$ *rotate the whole robot arm*

// *each point of whole robot arm will be pre-multiplied by* $\mathbf{R}_y(\theta)$
base ();

glTranslatef (0.0, h_1*, 0.0);* // $\mathbf{M}_{B \leftarrow La}$ *changes lower arm coord. sy. to base coord. sy.*
glRotatef (phi, 0.0, 0.0, 1.0); // $\mathbf{R}_z(\phi)$ *rotate the lower arm*

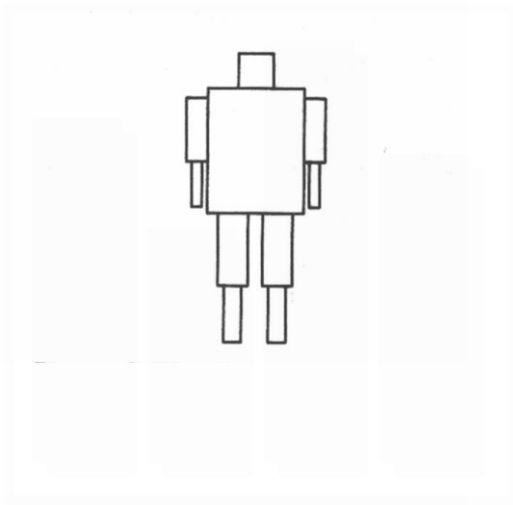
// *each point of lower arm will be pre-multiplied by* $\mathbf{R}_y(\theta)\mathbf{T}(0,h_1,0)\mathbf{R}_z(\phi)$
lower_arm ();

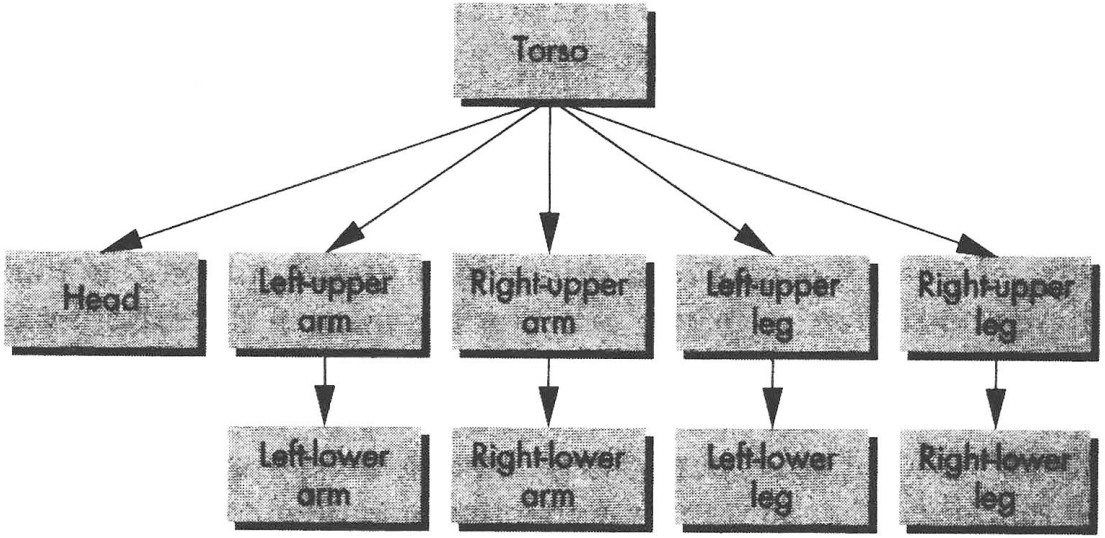
glTranslatef (0.0, h_2*, 0.0);* // $\mathbf{M}_{La \leftarrow Ua}$ *changes upper arm coord. sy. to lower arm coord. sy.*
glRotatef (psi, 0.0, 0.0, 1.0);

// each point of upper arm will be pre-multiplied by $\mathbf{R}_y(\theta)\mathbf{T}(0,h_1,0)\mathbf{R}_z(\phi)\mathbf{T}(0,h_2,0)\mathbf{R}_Z(\psi)$
upper_arm ();
}

Moving a Robot

- Need to organize the hierarchy better
- Solution: use *glPushMatrix* and *glPopMatrix* to store and retrieve intermediate composite relative transformations





Program

```
Robot ()
{
        glPushMatrix ();
        torso;

        glTranslate ...
        glRotate ...
        head ();

        glPopMatrix ();          // go back to the node of the torso
         glPushMatrix ();

        glTranslate ...          // similar technique used here as that
        glRotate ...             // used in example 2
        left_upper_arm ();
        glTranslate ...
        glRotate ...
        left_lower_arm ();

        glPopMatrix ();          // go back to the node of the torso
         glPushMatrix ();

        glTranslate ...
        glRotate ...
        right_upper_arm ();
              ⋮
}
```

Viewing Transform

Intended Learning Outcomes

- Able to set up a camera coordinate system
- Understand the properties of different projection methods
- Able to set up the required projection matrices and use appropriate OpenGL commands to realize the projection
- Describe the operation and function of clipping

Image generation process

- A camera has its own coordinate system $\mathbf{X}^{(VC)}$-$\mathbf{Y}^{(VC)}$-$\mathbf{Z}^{(VC)}$, called the viewer coordinate system
- viewer coordinate system is alternatively called camera coordinate system
- To generate an image, we first need to define a camera, then transforming the 3D scene from world coordinate system (WC) to viewing coordinate system (VC)

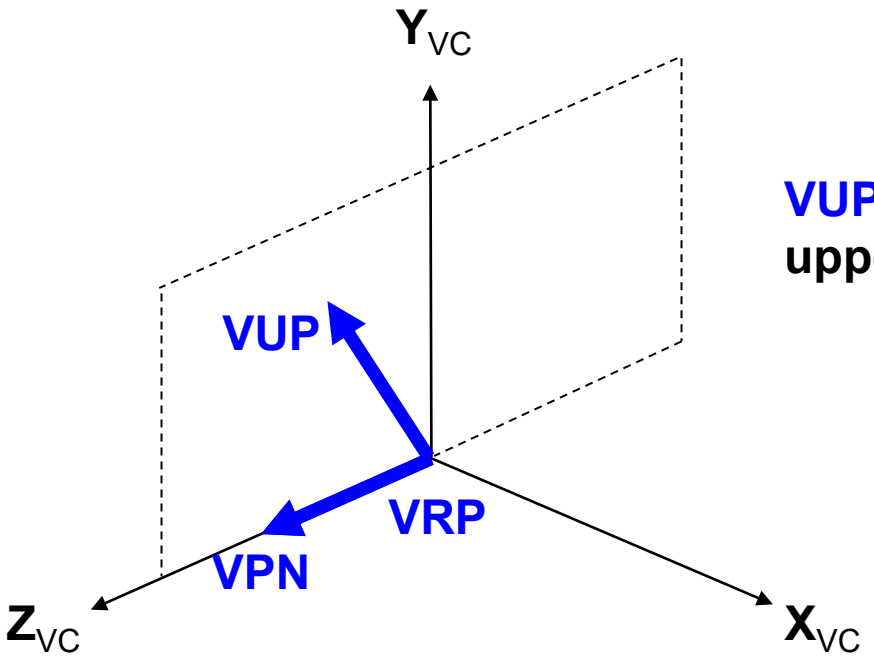
$$\mathbf{X}^{(WC)}\text{-}\mathbf{Y}^{(WC)}\text{-}\mathbf{Z}^{(WC)} \rightarrow \mathbf{X}^{(VC)}\text{-}\mathbf{Y}^{(VC)}\text{-}\mathbf{Z}^{(VC)}$$

- Then projecting each point to a view plane

To specify a viewer coordinate system, we need to specify three vectors

- View Reference Point (**VRP**): origin of the viewing coordinate system (i.e. physical location of the camera)
- View Plane Normal (**VPN**): a vector giving the pointing direction of the camera (i.e. +ve $Z^{(VC)}$ axis of the camera $X^{(VC)}$-$Y^{(VC)}$-$Z^{(VC)}$)
- View UP Vector (**VUP**) : a vector defining what is the upward direction for the film (image)

- Note 1: These vectors do not need to be unit vector
- Note 2: These vectors are in WC



VUP lies on the upper Z_{VC} – Y_{VC} plane

$\mathbf{Z}_{VC} = |\mathbf{VPN}|$ (unit vector in WC)

$\mathbf{X}_{VC} = |\ \mathbf{VUP} \times \mathbf{VPN}\ |$ (unit vector in WC)

$\mathbf{Y}_{VC} = \mathbf{Z}_{VC} \times \mathbf{X}_{VC}$ (unit vector in WC)

Note: | | is used in the notes to denote normalization to unit vector

Transformation from WC to VC

- $\mathbf{P}^{(VC)} = \mathbf{M}_{VC \leftarrow WC}\, \mathbf{P}^{(WC)}$
- Applying coordinate system transformation method 1:

$$\mathbf{M}_{VC \leftarrow WC} = \begin{pmatrix} \mathbf{X}_{VC} & \mathbf{Y}_{VC} & \mathbf{Z}_{VC} & \mathbf{VRP} \\ 0 & 0 & 0 & 1 \end{pmatrix}^{-1}$$

- Note: $\mathbf{X}_{VC}$, $\mathbf{Y}_{VC}$, $\mathbf{Z}_{VC}$ are unit column vector in WC

OpenGL commands

- *glMatrixMode (GL_MODELVIEW);*
- *gluLookAt (x0, y0, z0, xref, yref, zref, Vx, Vy, Vz);*

- **VRP** = (x0, y0, z0)
- **VPN** = (x0, y0, z0) – (xref, yref, zref)
- **VUP** = (Vx, Vy, Vz)

- To remember this, it is convenient to remember (x0, y0, z0) as where the camera is placed, (xref, yref, zref) as where the center of the scene is, and (Vx, Vy, Vz) as a vector that tells where it's up for the camera

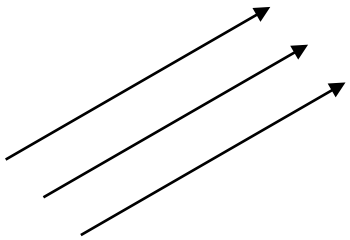
View Plane

- Also called projection plane or image plane
- It is usually a plane defined by Z_{VC} = constant, i.e., parallel to the $\mathbf{X}_{VC} - \mathbf{Y}_{VC}$ plane
- As the name implies, a 3D point (X, Y, Z) in viewer coordinates is projected to a 2D point lying on the view plane
- i.e. 3D becomes 2D

Projections : project $(X, Y, Z)^{(VC)}$ to $(x, y)^{(VC)}$

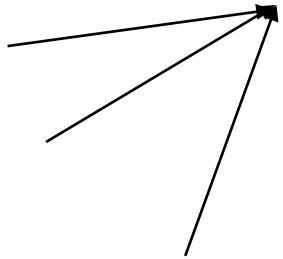
- Two general types: Parallel and Perspective Projections

Parallel projection



all light rays are parallel

Perspective projection



all light rays converge on a common point called projection reference point (PRP)

Parallel projection can be considered as the special case of perspective projection when PRP = ∞

Different properties

Parallel projection	Perspective Projection
coordinate positions are transformed to the projection plane along // lines i.e. center of projection at infinity	coordinate positions are transformed to the projection plane along lines that converge to a point called the center of projection (Projection Reference Point)
preserves relative proportions	does not preserve
use in engineering drafting	use in realistic views

Parallel Projections

- Specify a *projection vector* – a direction vector
- Two types:

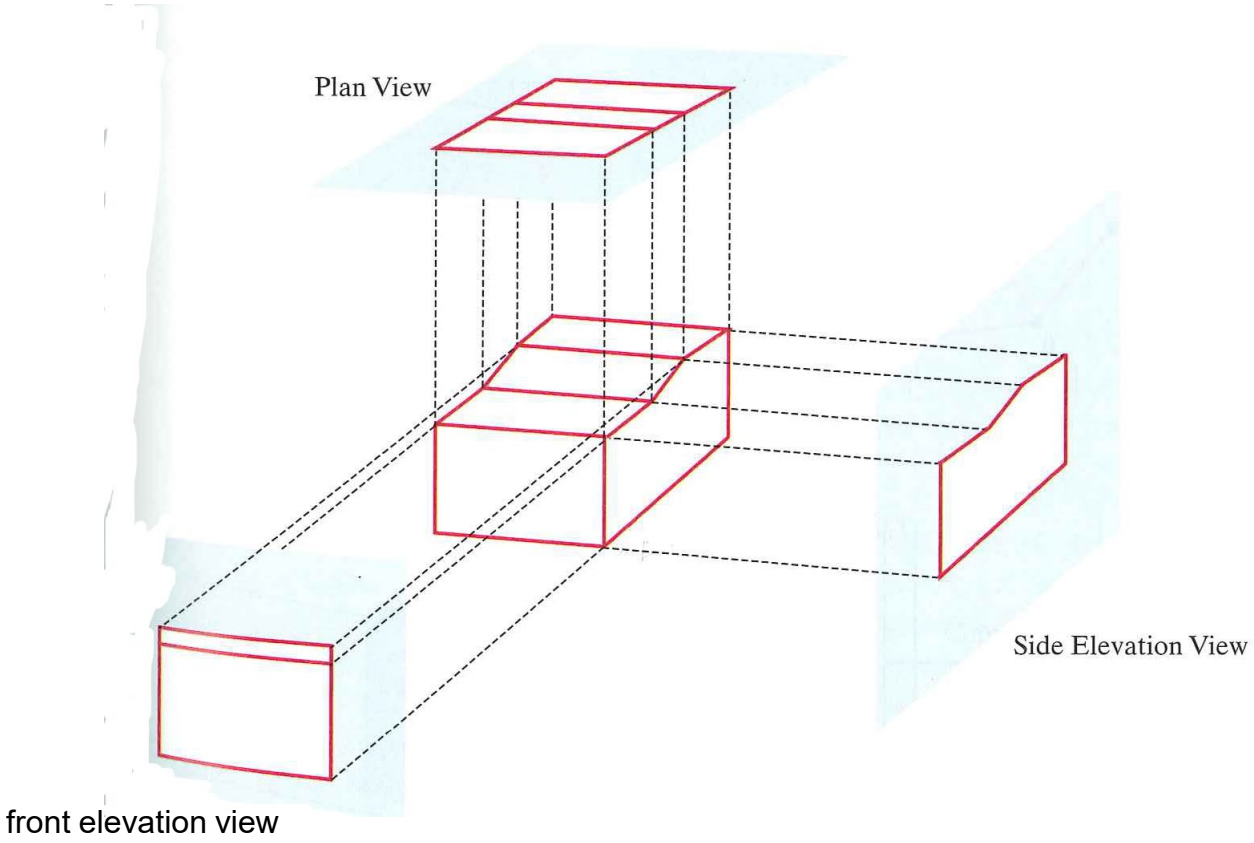
- *Orthographic projection* - projection vector ⊥ projection plane

- *Oblique projection* - projection vector not ⊥ to projection plane

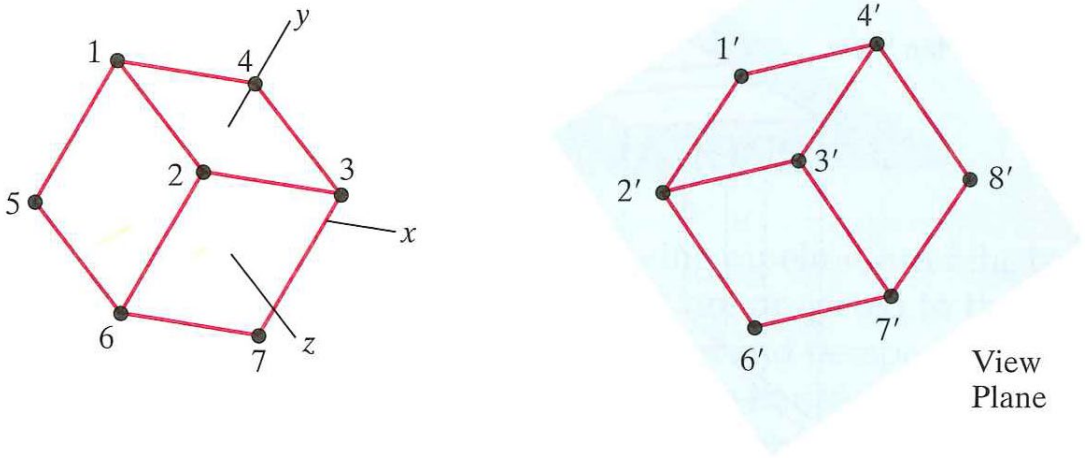
Orthographic Projection ($\alpha = 90^o$)

- Two types:
- *Front elevation*, *side elevation*, *rear elevation*, *plan view* - only X-Y, X-Z or Y-Z is shown
- *Isometric projection* - projection vector = (± 1, ± 1, ± 1)
  - For a cube, each side will be displayed equally
  - The 8 possibilities corresponds to viewing in the 8 octants

Front elevation, side elevation, rear elevation, plan view

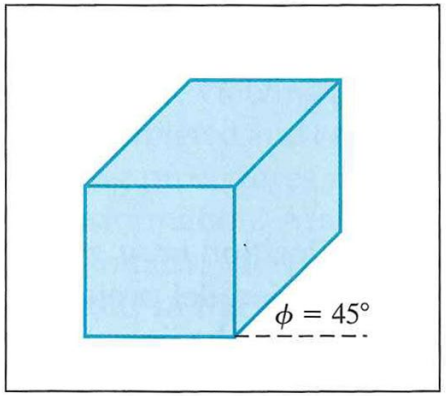


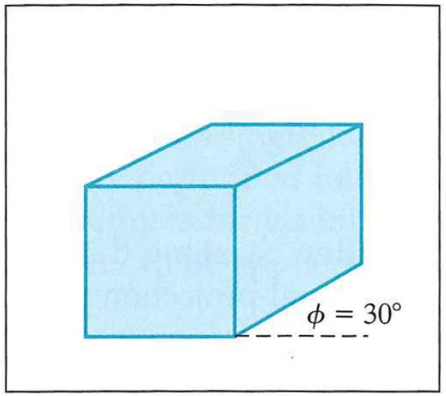
Isometric projection



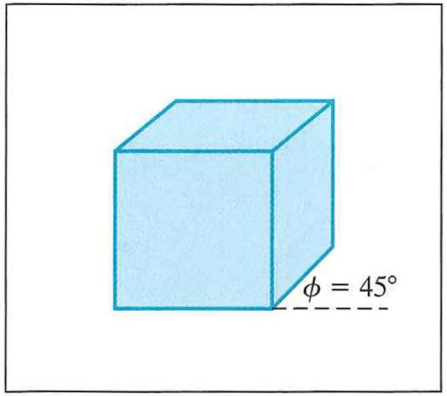
Oblique Projection ($\alpha \neq 90^o$)

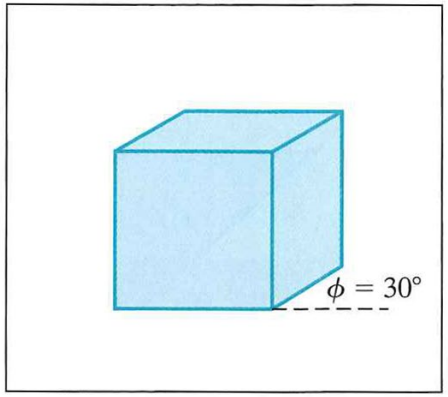
- Two types:
- *Cavalier projection* - projection vector makes an angle α of $\tan^{-1}1$ with projection plane
  - for a cube, length of X axis, Y axis and Z axis will remain the same
- *Cabinet projection* - projection vector makes an angle α of $\tan^{-1}2$ with projection plane
  - for a cube, length of X axis, Y axis will remain the same; length of Z axis will be halved.



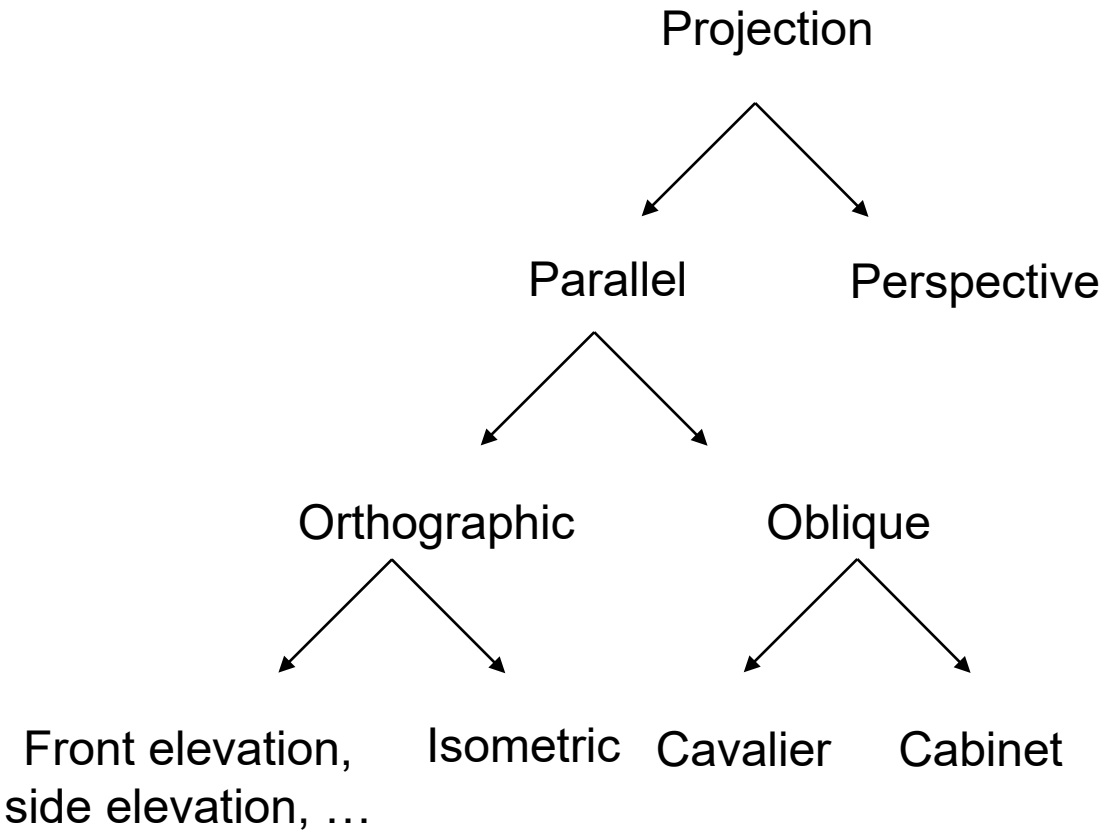


Cavalier Projection





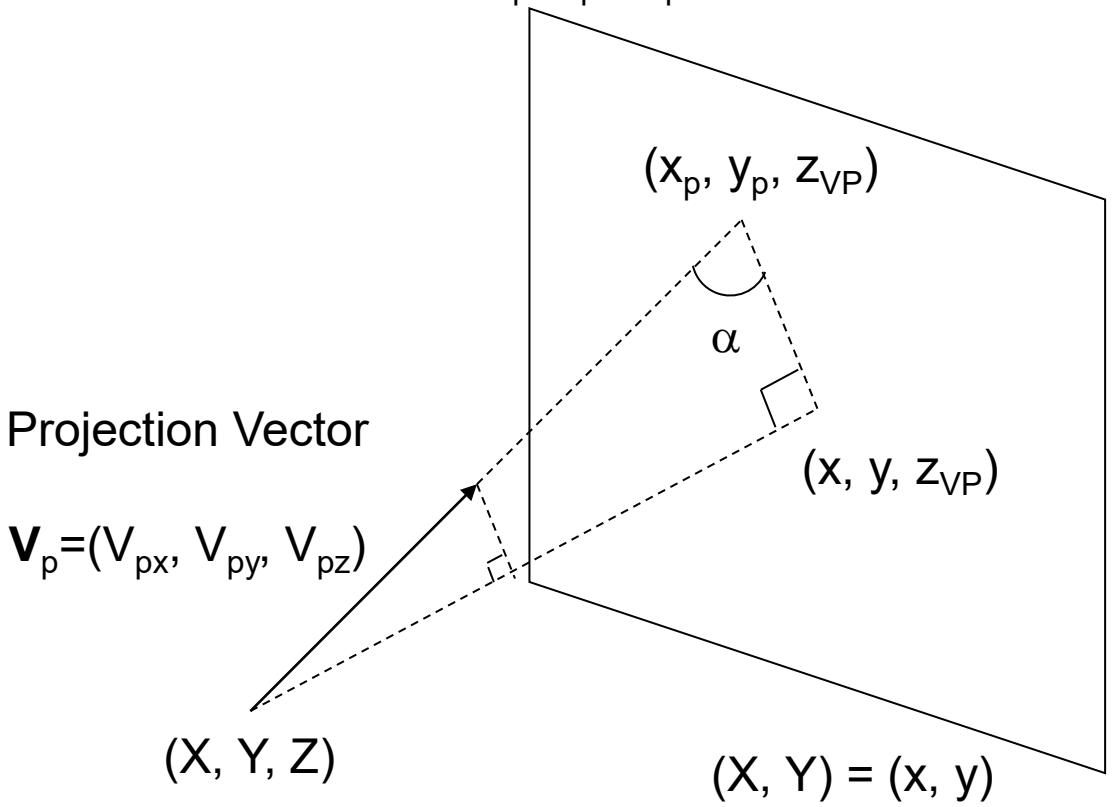
Cabinet Projection


Projection
Parallel
Perspective
Orthographic
Oblique
Front elevation, side elevation, …
Isometric
Cavalier
Cabinet

4 x 4 Transform for Parallel Projection

All quantities in this slide are already in VC

$\mathbf{P}^{(VC)} = (X, Y, Z)$ $\mathbf{P}^{(im)} = (x_p, y_p, z_{vp})$



View plane

$Z = z_{vp}$

z_{vp} is a constant

$\alpha = 90^o$ Orthographic projection

$\alpha \neq 90^o$ Oblique projection

By similar triangles,

$$\frac{x_p - X}{z_{vp} - Z} = \frac{V_{px}}{V_{pz}}$$

$$\frac{y_p - Y}{z_{vp} - Z} = \frac{V_{py}}{V_{pz}}$$

Rearranging,

$$x_p = X + (z_{vp} - Z)\frac{V_{px}}{V_{pz}} \qquad (1)$$

$$y_p = Y + (z_{vp} - Z)\frac{V_{py}}{V_{pz}}$$

- $\mathbf{P}^{(im)} = (x_p, y_p, z_{vp}, 1)$ $\quad \mathbf{P}^{(VC)} = (X, Y, Z, 1)$
- $\mathbf{P}^{(im)} = \mathbf{M}_{parallel}\, \mathbf{P}^{(VC)}$

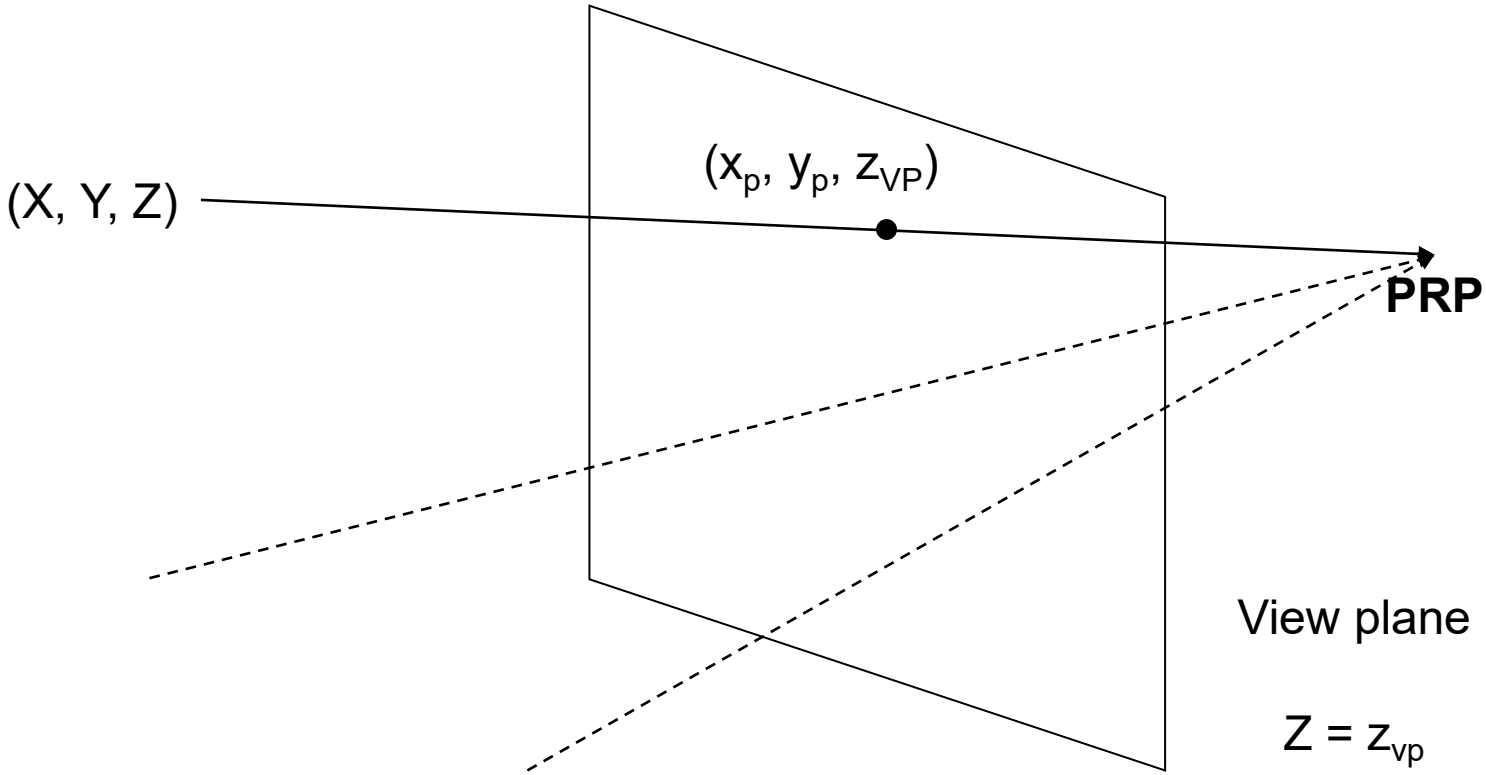
$$\mathbf{M}_{parallel} = \begin{pmatrix} 1 & 0 & -\frac{V_{px}}{V_{pz}} & z_{vp}\frac{V_{px}}{V_{pz}} \\ 0 & 1 & -\frac{V_{py}}{V_{pz}} & z_{vp}\frac{V_{py}}{V_{pz}} \\ 0 & 0 & 0 & z_{vp} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Verify that the first two rows implement eqn (1) above
- The third row is such that the projected point is at Z = z_{vp}. However, this is not maintained; in OpenGL, $\mathbf{p}^{(im)}$ = (x_p, y_p, Z), i.e. the original Z is kept for depth tests

- Setting the third row to 0 0 1 0 achieves the same effect as maintaining the original Z.

Perspective Projection

- ALL light rays goes through the Projection Reference Point (**PRP**), also called center of projection.


(X, Y, Z)
(x_p, y_p, z_{VP})
PRP
View plane
$Z = z_{vp}$

Example:

i) **PRP** = **VRP**
ii) Z = z_{vp} is the view plane

By similar triangles,

$$\frac{x_p}{z_{vp}} = \frac{X}{Z} \qquad \frac{y_p}{z_{vp}} = \frac{Y}{Z}$$

Multiplying each side by z_{vp} yields

$$x_p = \frac{z_{vp} \cdot X}{Z} = \frac{X}{Z / z_{vp}} \qquad (2)$$

$$y_p = \frac{z_{vp} \cdot Y}{Z} = \frac{Y}{Z / z_{vp}}$$

- $\mathbf{P}^{(VC)} = (X, Y, Z, 1)$
- $\mathbf{P}^{(im)} = \mathbf{M}_{perspective}\ \mathbf{P}^{(VC)}$

$$\mathbf{M}_{perspective} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/z_{vp} & 0 \end{bmatrix}$$

- Verify that the first two rows and the fourth row implements eqn (2) above using homogeneous coordinates operation
- Note that the fourth row is not 0 0 0 1 anymore
- The third row is such that the projected point is at Z = z_{vp}. However, this is not maintained; in OpenGL, $\mathbf{p}^{(im)}$ = (x_p, y_p, Z), i.e. the original Z is kept for depth tests

Clipping

- Any object not within the clipping volume does not need to be processed – this eliminates most of the objects at one go
- For a convex clipping volume bounded by planes, one can check whether a point is inside by checking the signs of the plane equations (see Lecture 2).

OpenGL – first set matrix mode

- *glMatrixMode (GL_PROJECTION);*

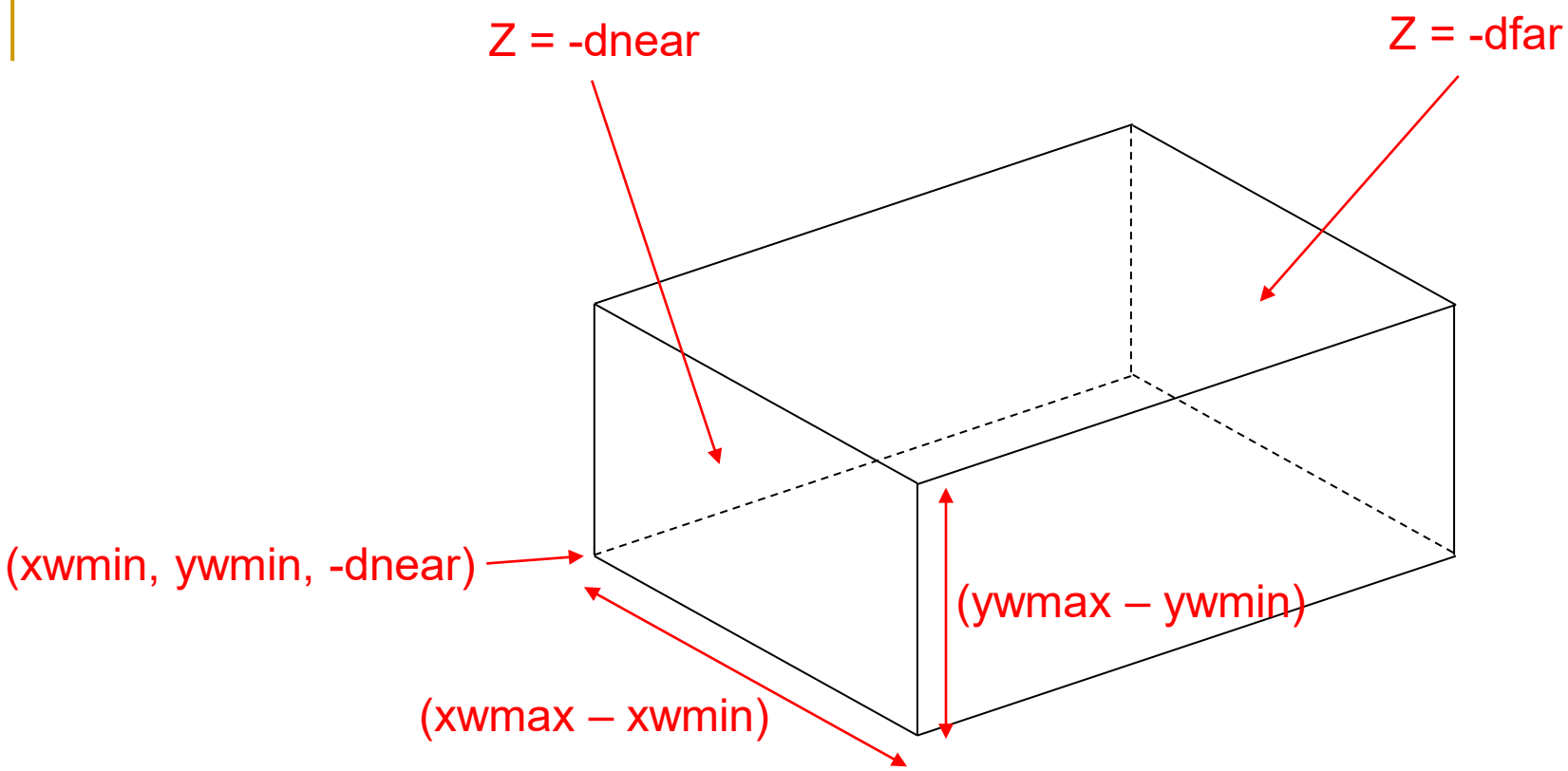
- Note: *GL_PROJECTION* is used as it deals with projection
- There are two 4 x 4 composite transformation matrices: *GL_MODELVIEW* and *GL_PROJECTION*
- A point is pre-multiplied by

  [*GL_PROJECTION*] [*GL_MODELVIEW*]

- *glOrtho* and *gluPerspective* commands may be used

OpenGL – Orthographic projection

- *glOrtho (xwmin, xwmax, ywmin, ywmax, dnear, dfar)*
  - Projection vector V_p = (0, 0, 1)
  - Clipping planes: Z = -dnear   Z = -dfar
  - Near clipping plane Z = -dnear also serve as the view plane
  - Only points whose X and Y are in |xwmin, xwmax| and |ywmin, ywmax| respestively are displayed
  - Clipping volume is a rectangular box



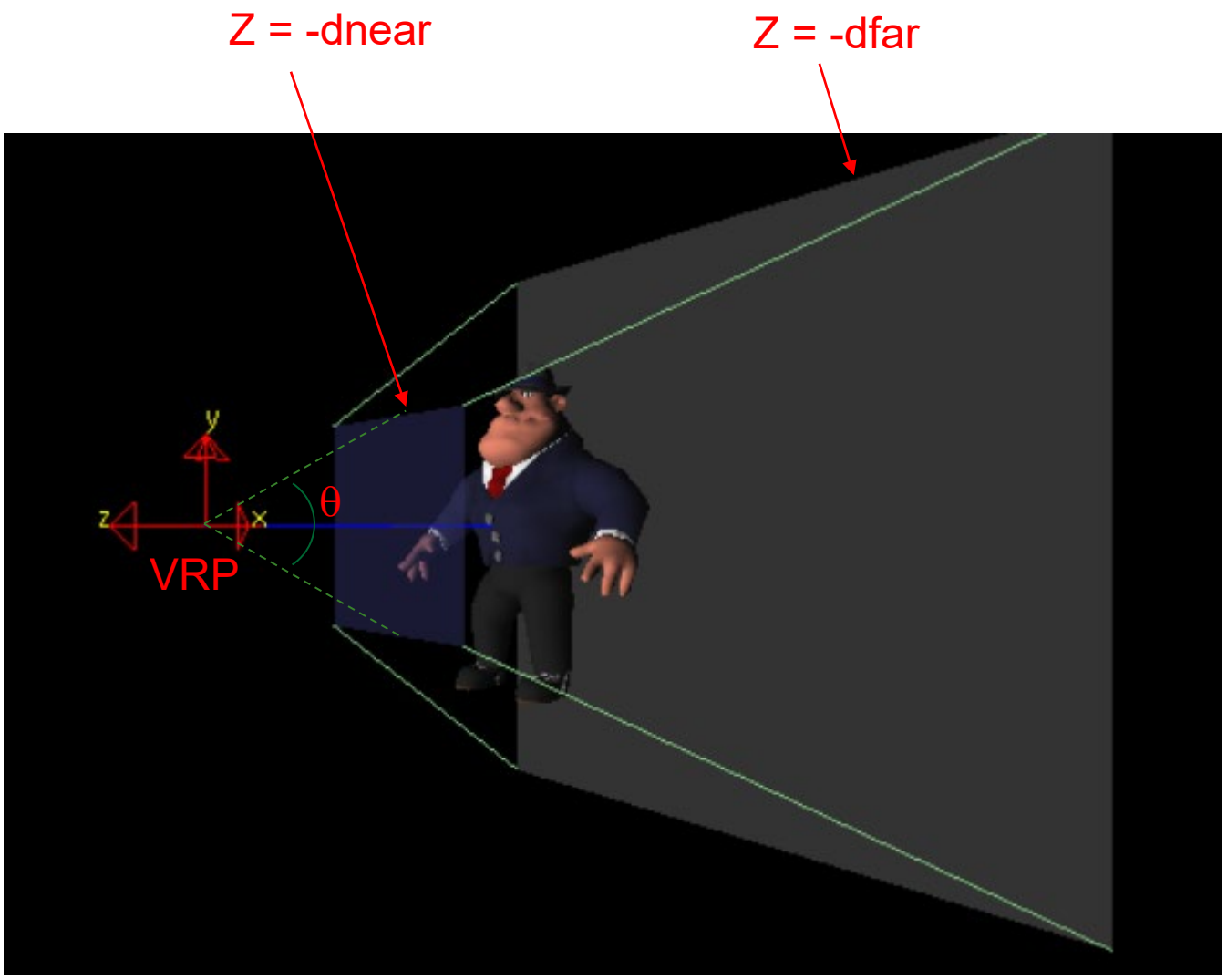
Only objects inside the rectangular shaped clipping volume is further processed

OpenGL – Perspective projection

- *gluPerspective (theta, aspect, dnear, dfar)*
  - **PRP** = **VRP**
  - Z = -*dnear* is the view plane (note the –ve sign)
  - dnear and dfar define the near and far clipping planes Z = -*dnear* and Z = -*dfar* respectively
  - *theta* is the angle of view
  - *aspect* = (width /height)
  - *theta* and *aspect* together determines size of image window

---

- clipping volume is a frustum



aspect = width /height of the blue plane

Only objects inside the frustum shaped clipping volume is further processed

Animation and Movie Making

Intended Learning Outcomes

- Distinguish two types of animation
- Describe the four steps of animation
- Describe key frame and intermediate frame generation techniques
- Able to model and program common animation effects such as acceleration, deceleration, and periodic motion

Two Types of Animation

- Real time animation
  - Update parts of image in real time as soon as available
- Frame by frame animation
  - Use two frame buffers
  - Display first buffer content
  - Update on the second buffer
  - Switch the two buffers when the new image has finished drawing on the second buffer
  - Use in system that does not require real time e.g. movie production

Comparisons

- Real time animation
  - Adv. Critical Information displayed as soon as available
  - Disadv. Refresh rate of each pixel must be at least 16 frames/sec to avoid flickering
  - Used in real time systems

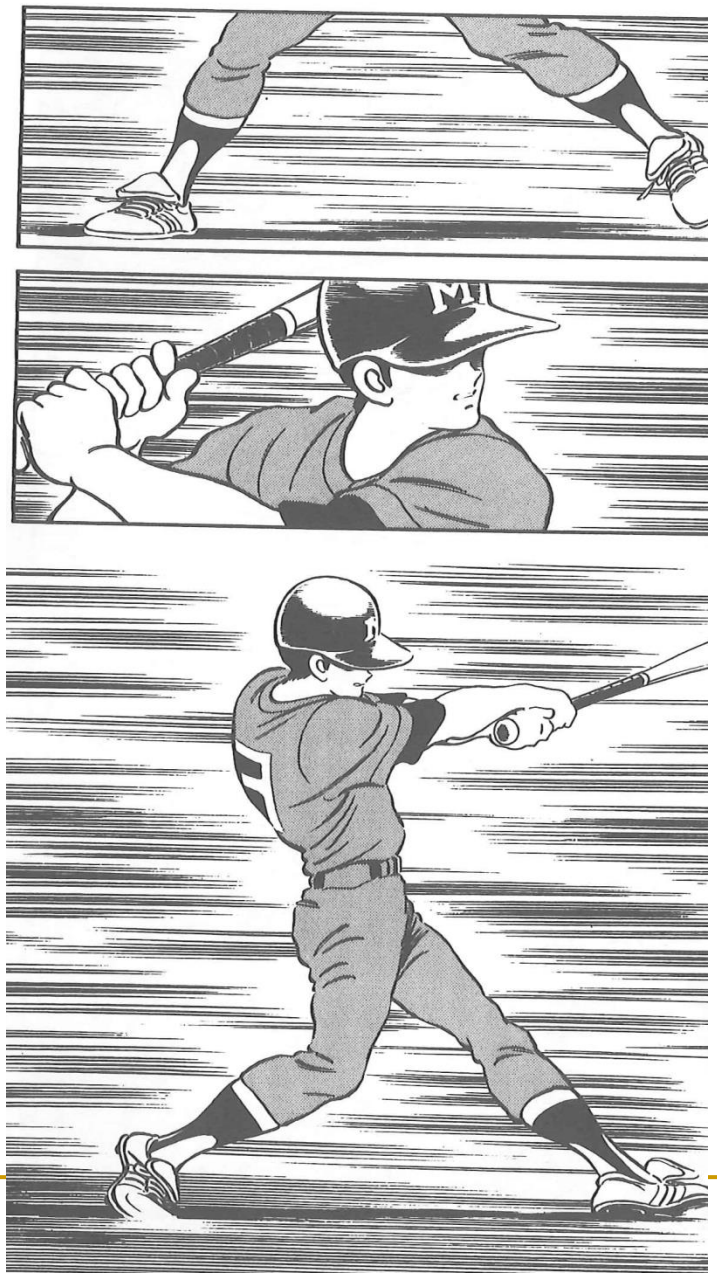
- Frame by frame animation
  - Adv. No flickering even if the refresh rate is low
  - Disadv. Display of information may be delayed up to one frame
  - Used in non-real time systems e.g. movie, and applications that do not have unreasonably slow frame rate

Designing an Animation

- Story Board
  - outline of the action. Defines the motion sequence as a set of basic events that are to take place
- Object Definitions
  - choose the object representation and movement of each object in the story
- Generation of Key Frames
  - generate a detailed image of the scene at a certain time in the animation sequence
  - More key frames are specified when the motion is intricate
- Generation of In-between Frames
  - Intermediate frames between the key frames.
  - The number of in-betweens needed is determined by the media to be used to display the animation.

Key frames

From comic “H2”



Generation of in-between frames from key frames

- Key frames can be generated by the CG pipeline
- Morphing can be used to generate in-between frames
- Morphing – short form for metamorphosis
- It is a transformation of object shape from one form to another

Morphing

- Step 1 : Equalize the number of vertices of the two shapes
- Step 2 : Find correspondence between each pair of vertices
- Step 3 : Find intermediate positions of the vertices by interpolation

Algorithm

Input : Key frames k and k+1

Algorithm

1. Let V_k be the number of vertices in key frame k. Compute

$$V_{\max} = \max(V_k, V_{k+1}) \qquad V_{\min} = \min(V_k, V_{k+1})$$

and

$$N_{ls} = (V_{\max} - 1) \bmod (V_{\min} - 1)$$

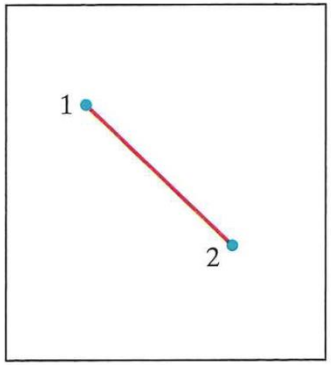
$$N_p = \text{int}\left(\frac{V_{\max} - 1}{V_{\min} - 1}\right)$$ // int (x) takes the largest integer smaller than x

2. Add N_p points to N_{ls} line sections of $\text{keyframe}_{\min}$ (the key frame with less number of vertices)
   Add N_p - 1 points to the remaining edges of $\text{keyframe}_{\min}$

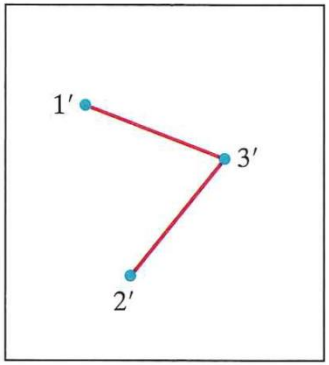
   // now both key frames have equal number of vertices

3. Linearly interpolate for each pair of corresponding vertices in the two key frames to generate the in-between frames

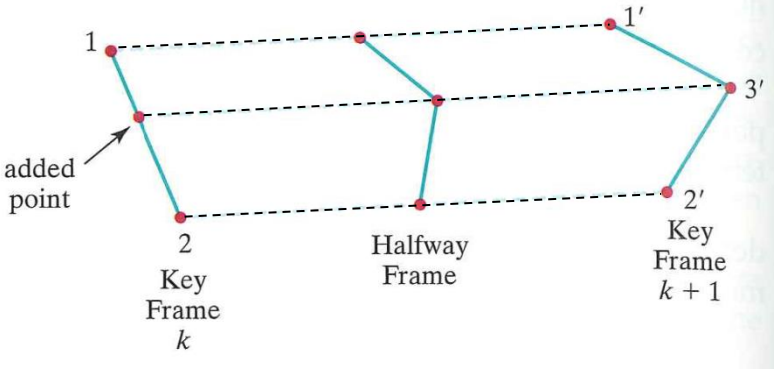
Output: a set of in-between frames

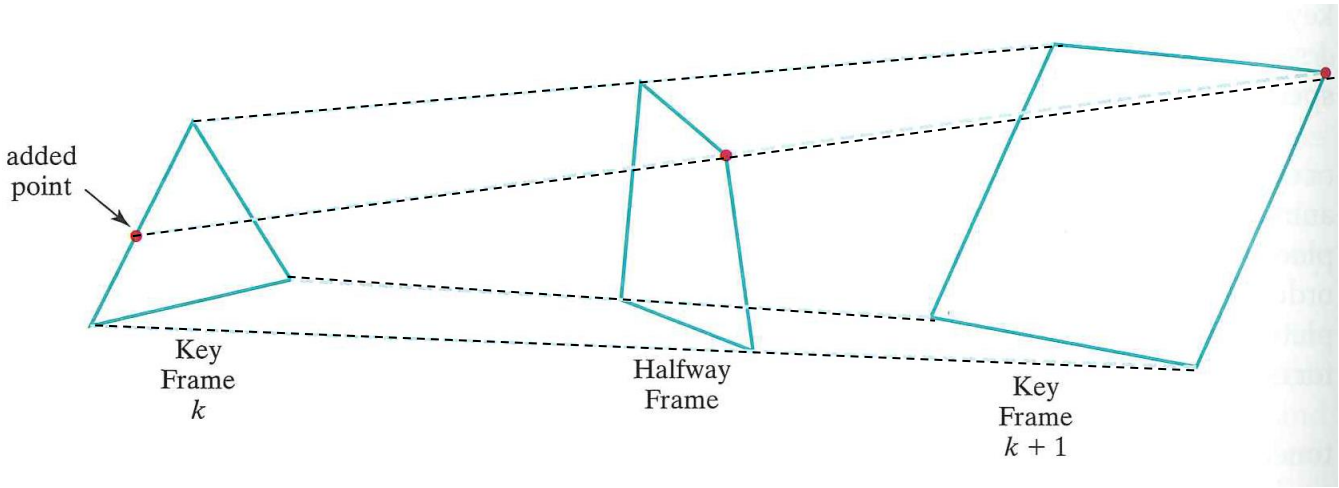


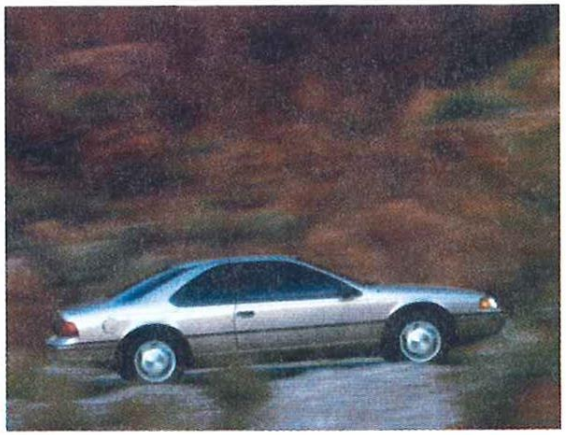
Key
Frame k



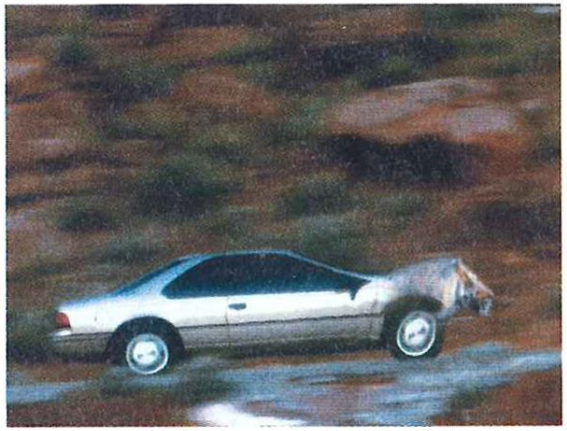
Key
Frame $k + 1$







(a)



(b)

(c)

(d)

Simulating Acceleration and Deceleration

- Idea : Adjust the time spacing of successive frames
- n in-between frames for two key frames at t = t_1 and t_2
- Constant velocity

$$tB_j = t_1 + \frac{j\Delta t}{n+1} \qquad \text{j} = 1, 2, ..., \text{n} \qquad \Delta t = t_2 - t_1$$

Empirical functions

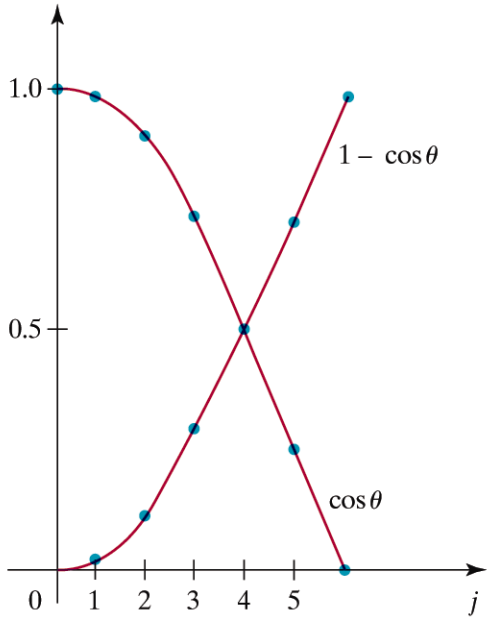
- Acceleration: Use empirical function 1- cos θ 0< θ < π/2

$$tB_j = t_1 + \Delta t[1 - \cos\frac{j\pi}{2(n+1)}]$$

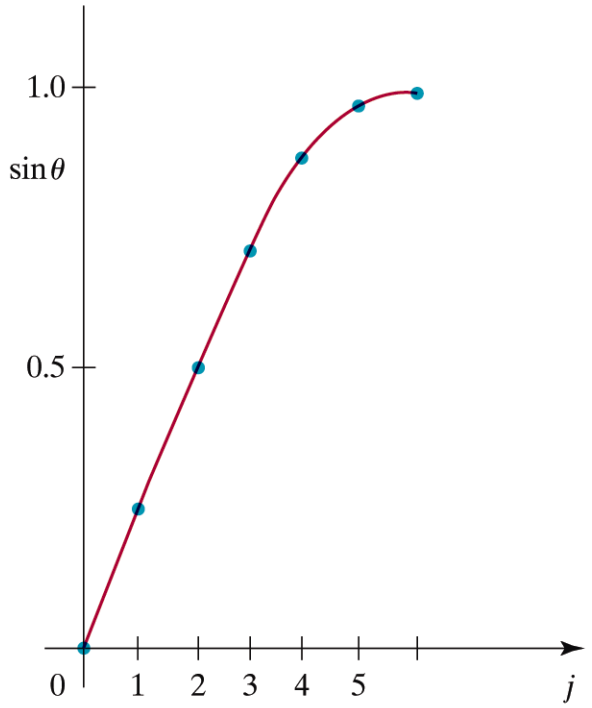
- Deceleration: Use sin θ

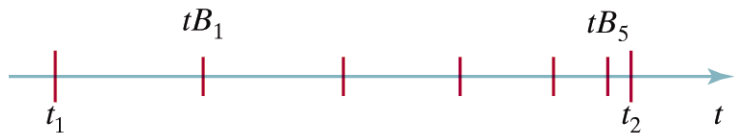
$$tB_j = t_1 + \Delta t[\sin\frac{j\pi}{2(n+1)}]$$

- Accelerate then decelerate: Use $\frac{1}{2}(1-\cos\theta)$ $0 < \theta < \pi$

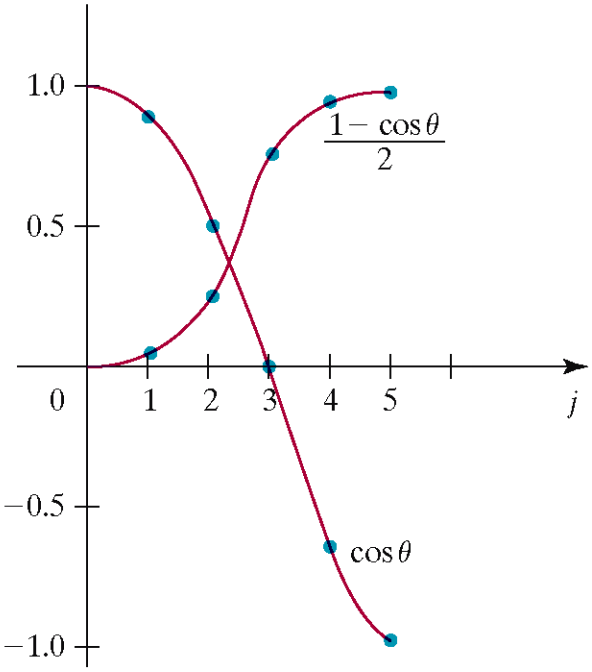


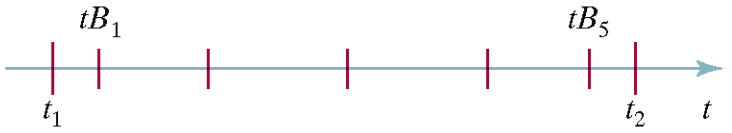
Acceleration





Deceleration



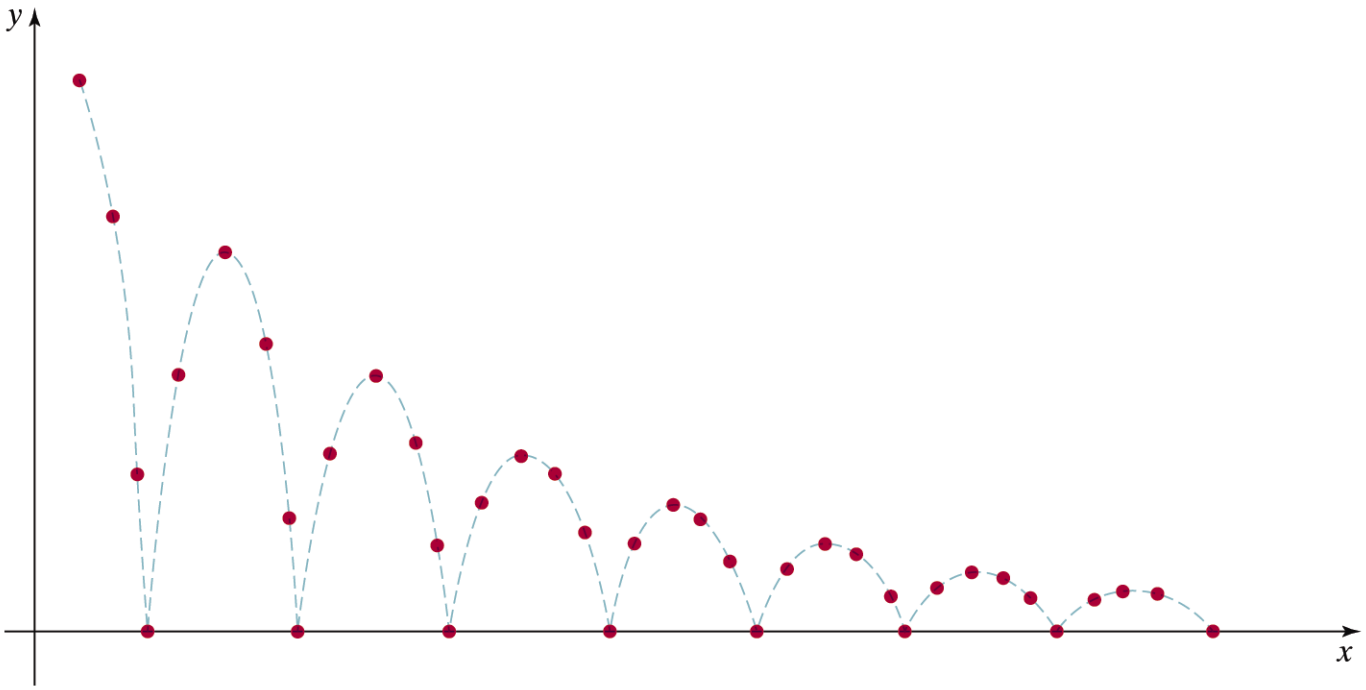


Acceleration then Deceleration

Specifying Motion (1)

- For general motions, empirical functions are not accurate enough
- Three ways to calculate motion
  - Direct Motion specification
    - Solve the motion equations, then just plot the trajectory
    - Example: simple harmonic motion
  - Kinematics and dynamics
    - Kinematics : calculate position, velocity and acceleration

$$v = u + at \qquad s = s_0 + ut + \frac{1}{2}at^2$$



Simple harmonic motion

Specifying Motion (2)

- Inverse Kinematics

  Specify the initial and final conditions, then the system solves for the motion

- Dynamics

  Specify the forces : Physically based modelling

$$F - kv - h(x - x_0) = ma$$

- Inverse Dynamics

- Goal Directed System
  - Specify desired behaviour : “Walk”, “Run”
  - Converted into mathematical motion by the system

Periodic Motion

- Motion must be synchronized with the frame rate, otherwise may result in incorrect motion
- A typical example is shown in the figures below.
- Solutions
  - Generate a frame after each fixed angle increment, but this may cause other problems if the periodic motion is too fast
  - Use timer and ask user to have a certain minimum graphics capability in their computer (common practice in games)
  - Periodically reset parameters to prevent numerical error build up

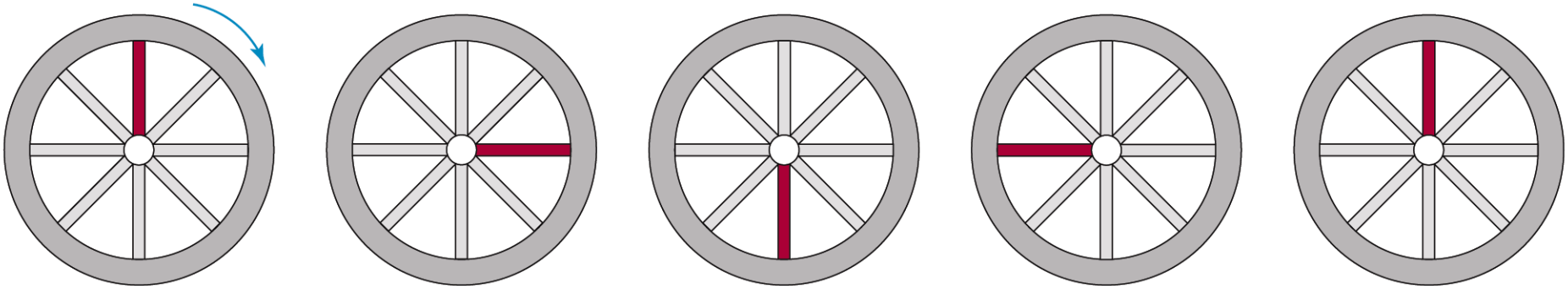
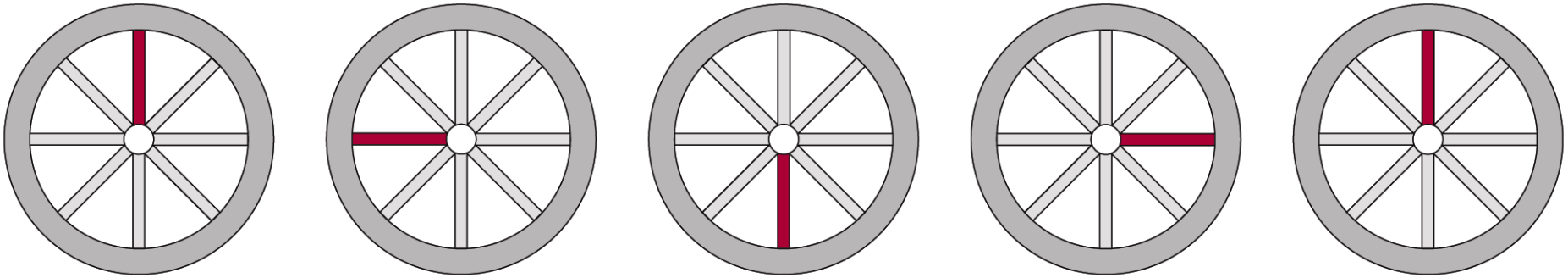


Figure 13-21

Five positions for a red spoke during one cycle of a wheel motion that is turning at the rate of 18 revolutions per second.



The first five film frames of the rotating wheel in Fig. 13-21 produced at the rate of 24 frames per second.

OpenGL Commands

- Double Buffering
  - *glutInitDisplayMode (GLUT_DOUBLE)*
  - *glutSwapBuffers ( );*

- To produce an animation
  - *glutIdleFunc (animationFcn)*
  - *animationFcn* is a procedure written by the user to update the animation parameters
  - *glutPostRedisplay ( );*

- Using the timer
  - *glutGet(GLUT_ELAPSED_TIME)*

Lighting and Rasterization - Shading

Intended Learning Outcomes

- Classify different types of light sources
- Understand the image formation process
- Mathematically model three types of reflection and understand their properties
- Understand three rendering methods and compare their pros and cons
- Able to program lighting and shading using OpenGL

Lighting and Shading Models

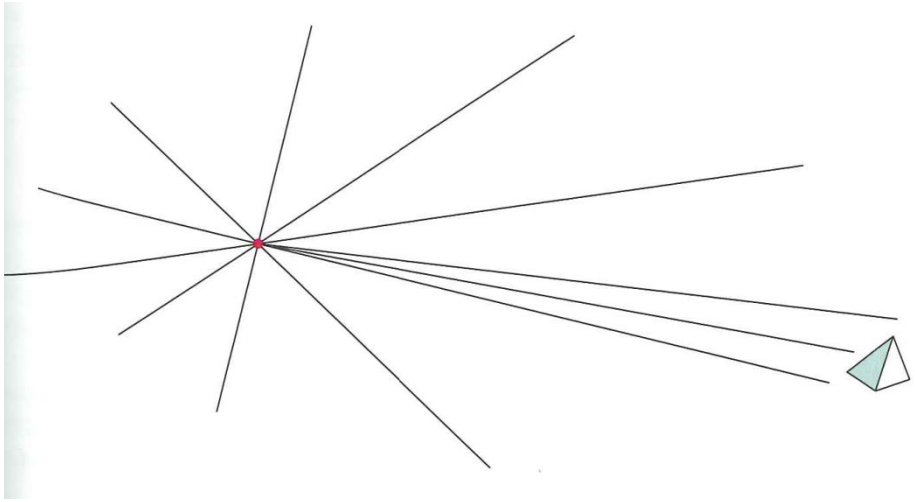
- Calculate intensity and colour of light that we should see at a given point of a scene
- Ultimate aim : *Photorealism*

- Lighting /Illumination models
  - models lighting from light sources and the environment

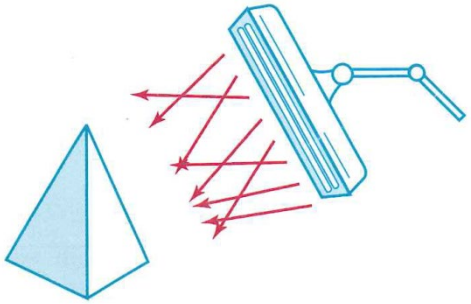
- Shading models
  - models how lights are processed (reflected, absorbed, refracted etc) by the objects and the atmosphere

Light sources

- Ambient source
  - models background light
- Point source
  - for small nearby light sources
- Distributed source
  - for large nearby light sources
  - models by a collection of point sources
- Lighting direction
  - (e.g. sun) - for distant light sources

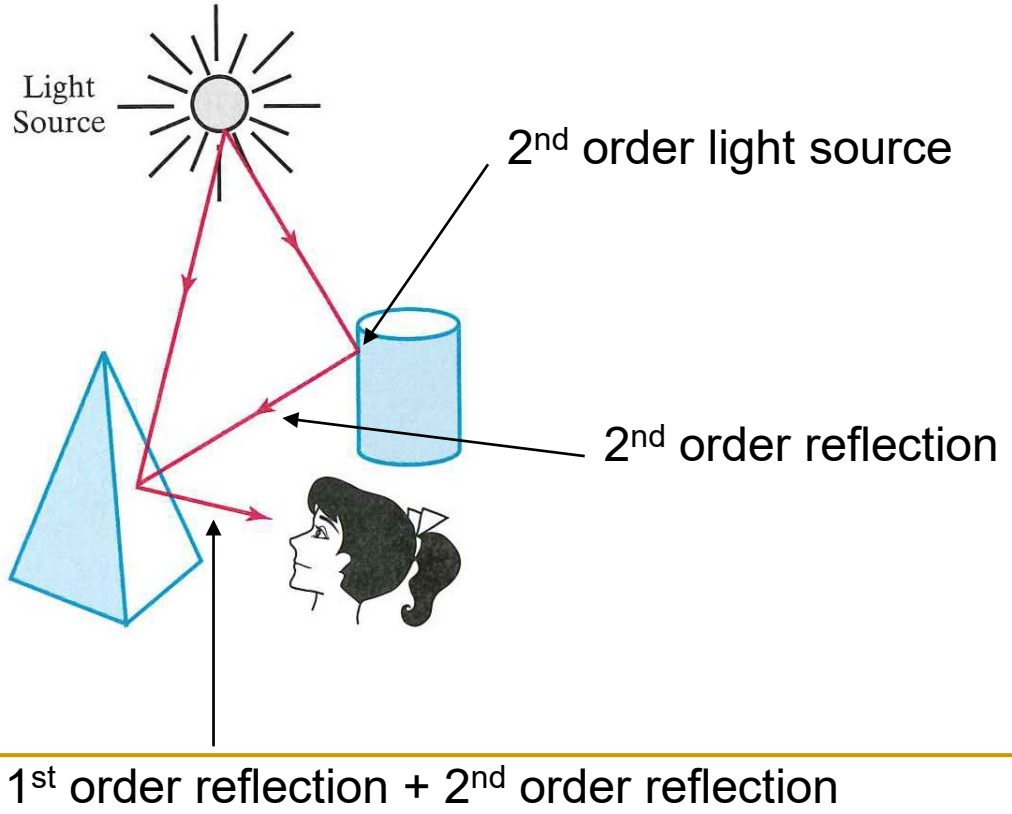


Point Source



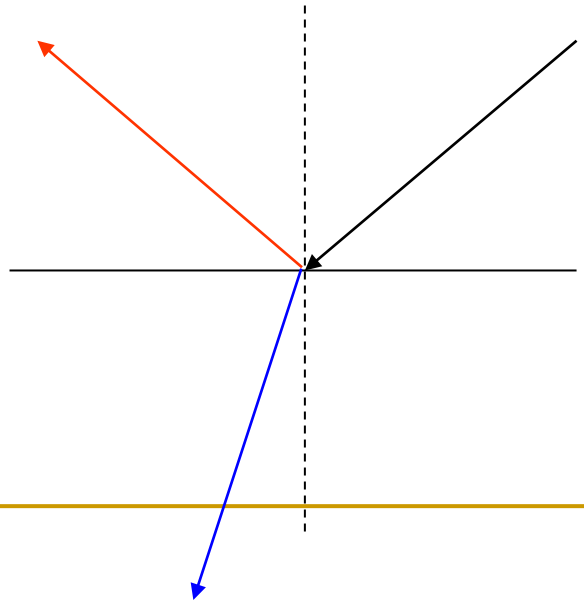
Distributed Source

- Realistic lighting is higher order and complicated



Shading

- When light is incident on an object
  - part is reflected
  - part is absorbed
  - part is refracted



Object properties

- *Opaque* object only reflect and absorb light
- *Transparent* object only refract and absorb light
- *Semi-transparent* object reflect, refract and absorb light

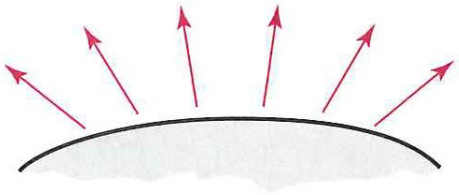
- The amount of light reflected depends on material.

  - Shiny material : reflect most of the light
  - Dull material : absorb most of the light

- Let restrict discussion to opaque object at present

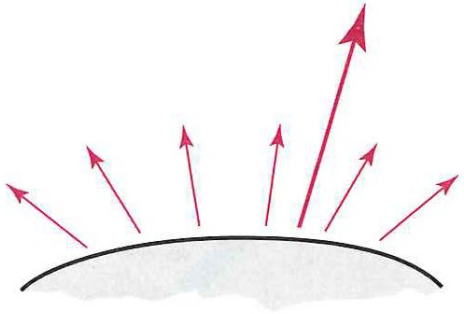
Types of Reflection

- Ambient reflection
  - Average signal from the background
  - Non-directional



Diffuse reflections from a surface.

- Diffuse reflection
  - Rough, dull, matte surfaces
  - scatter light equally in all directions



Specular reflection superimposed on diffuse reflection vectors.

- Specular reflection
  - Smooth, shiny, mirror like surfaces
  - reflect light more in one direction

Ambient reflection

$$I_{ambdiff} = k_a I_a$$

ka ambient reflection coefficent, $0 \leq k_a \leq 1$

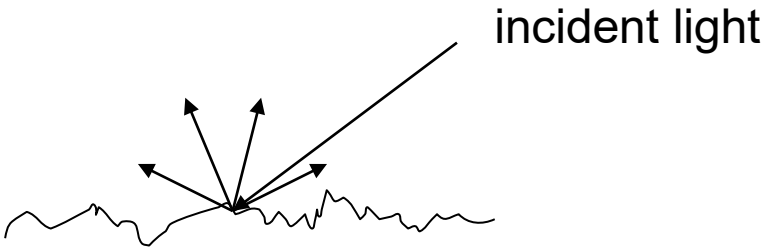
I_a incident ambient light

- Can be interpreted as the average value of diffuse reflection from numerous light sources in the background

Diffuse Reflection

- Consider a point light source or lighting direction

- *Lambertian surfaces* : Reflections from the surface are scattered with equal intensity in all directions, independent of the viewing direction



Diffuse (Lambertian)
Surface (Rough, dull
e.g. wood)

- Amount of incident light received by the surface is proportional to the projected area of the surface in the lighting direction

$$I_{l,diff} = k_d I_l (\mathbf{N} \cdot \mathbf{L})$$

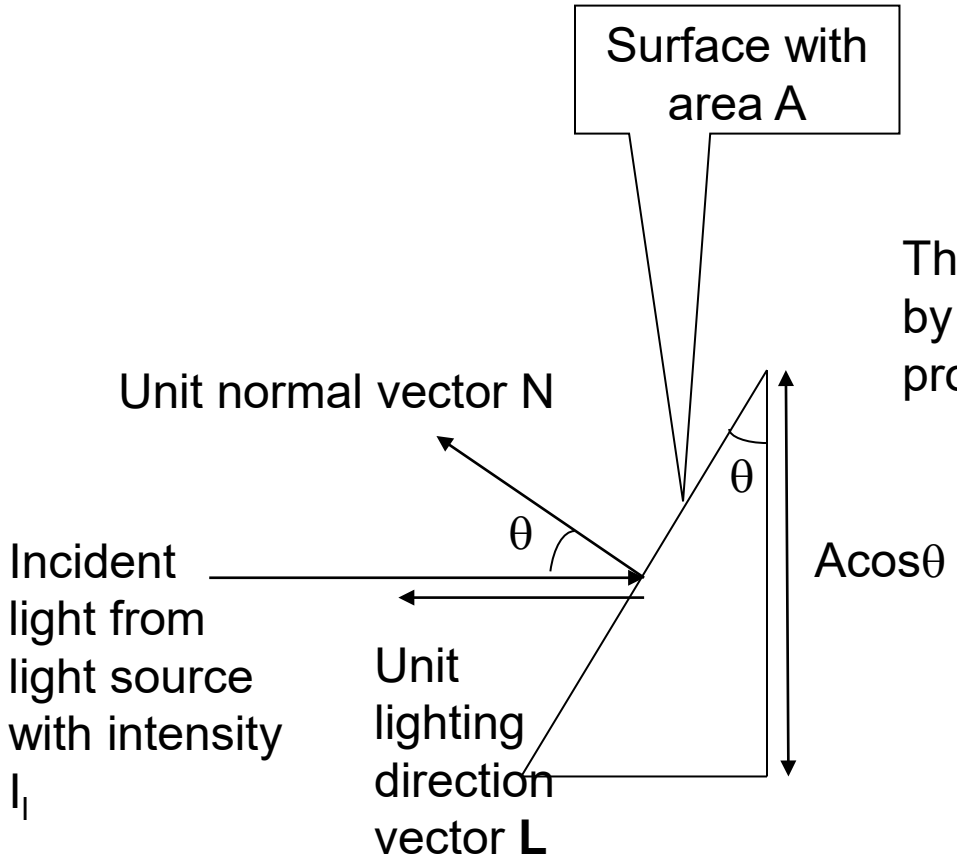
k_d diffuse reflection coefficient, $0 \leq k_d \leq 1$

I_l Incident light intensity

N unit normal of the surface

L unit light direction vector

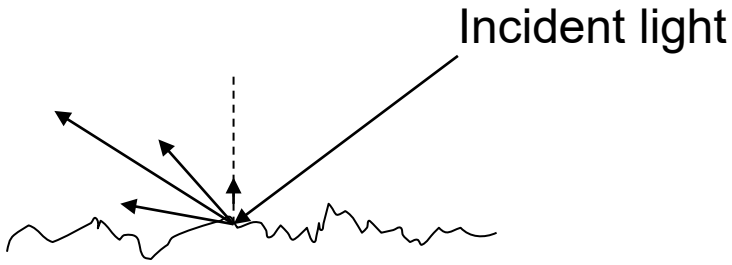
- **N**·**L** models the projected area



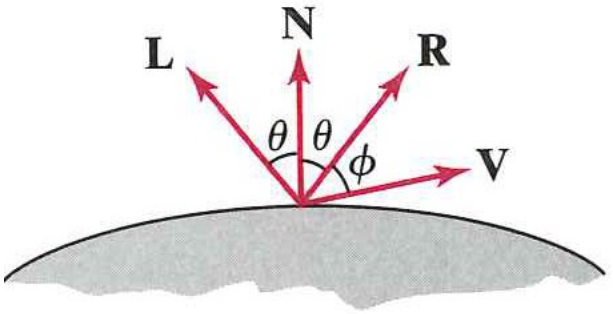
The total amount of light received by the surface with area A is proportional to Acosθ = A (**N•L**)

Specular reflection

- Consider a point light source or lighting direction.
- Ideal specular surface = perfect mirror: light is only reflected in the direction of R
- Non-ideal reflector: some light are scattered around R



Specular
Surface (Shiny e.g. mirror, gold silver, glass)



L Incident light direction

$$I_{l,spec} = W(\theta) I_l \cos^{n_s} \phi$$

$W(\theta)$ specular reflection coefficent, $0 \leq W(\theta) \leq 1$
sometimes $W(\theta)$ is assumed to be a constant k_s

N bisects **L** and **R** (incident angle = reflection angle in a perfect mirror)

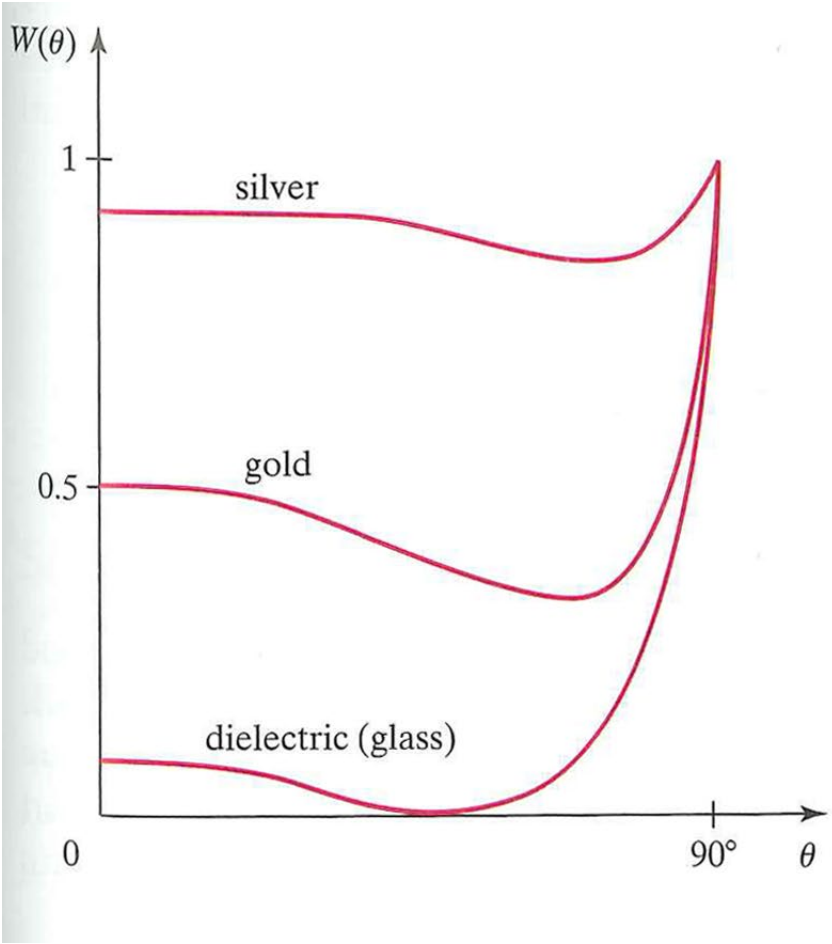
R unit specular reflection direction vector

$\mathbf{R} = (2\mathbf{N} \cdot \mathbf{L})\mathbf{N} - \mathbf{L}$

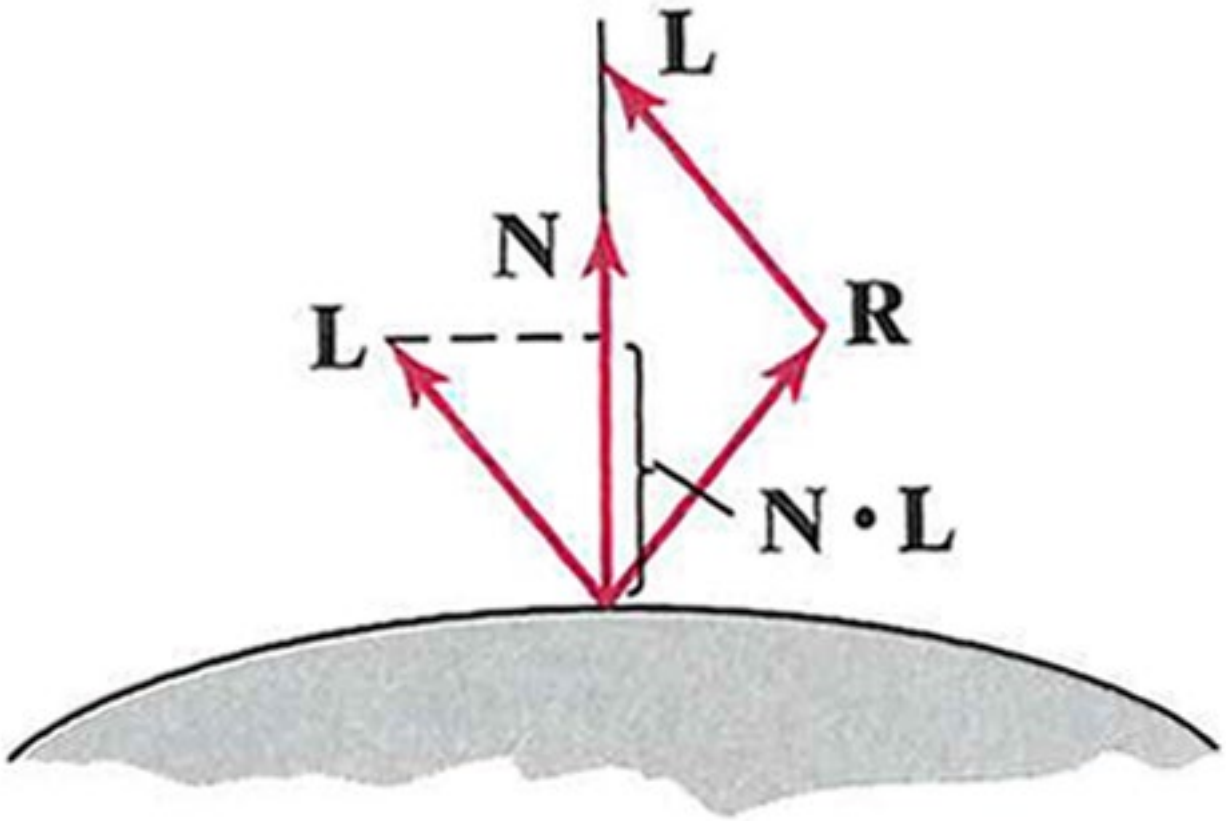
V unit viewing direction vector

$\cos(\phi) = \mathbf{R} \cdot \mathbf{V}$ $\quad 0 \leq \phi \leq \pi/2$

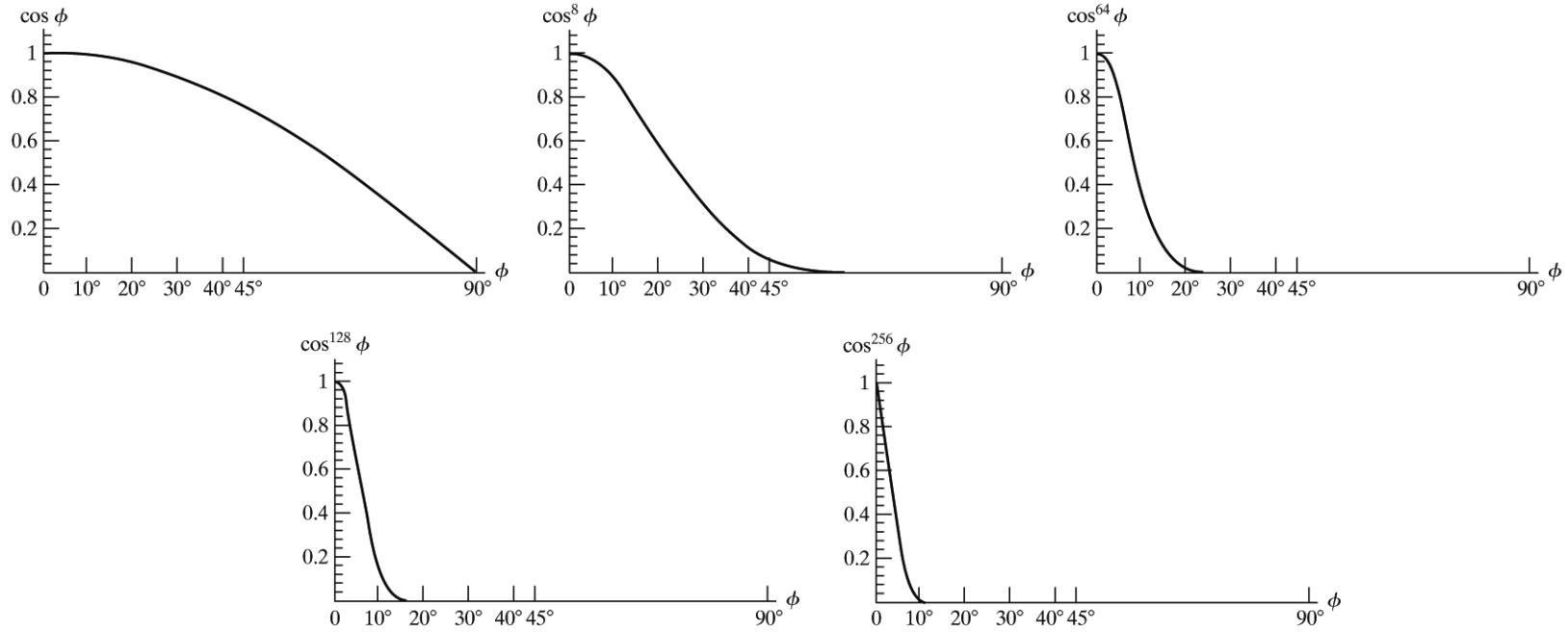
n_s specular reflection exponent, $n_s = \infty$ for perfect mirror



Approximate variation of the specular-reflection coefficient for different materials, as a function of the angle of incidence.



$$\mathbf{R} = (2\mathbf{N}\cdot\mathbf{L})\mathbf{N} - \mathbf{L}$$



Plots of $\cos^{n_s} \phi$ using five different values for the specular exponent n_s.

General Model with n light sources with ambient, diffuse and specular terms

$$I = k_a I_a + \sum_{i=1}^{n} I_{li}[k_d(\mathbf{N}\cdot\mathbf{L}_i) + W(\theta_i)(\mathbf{V}\cdot\mathbf{R}_i)^{n_s}]$$

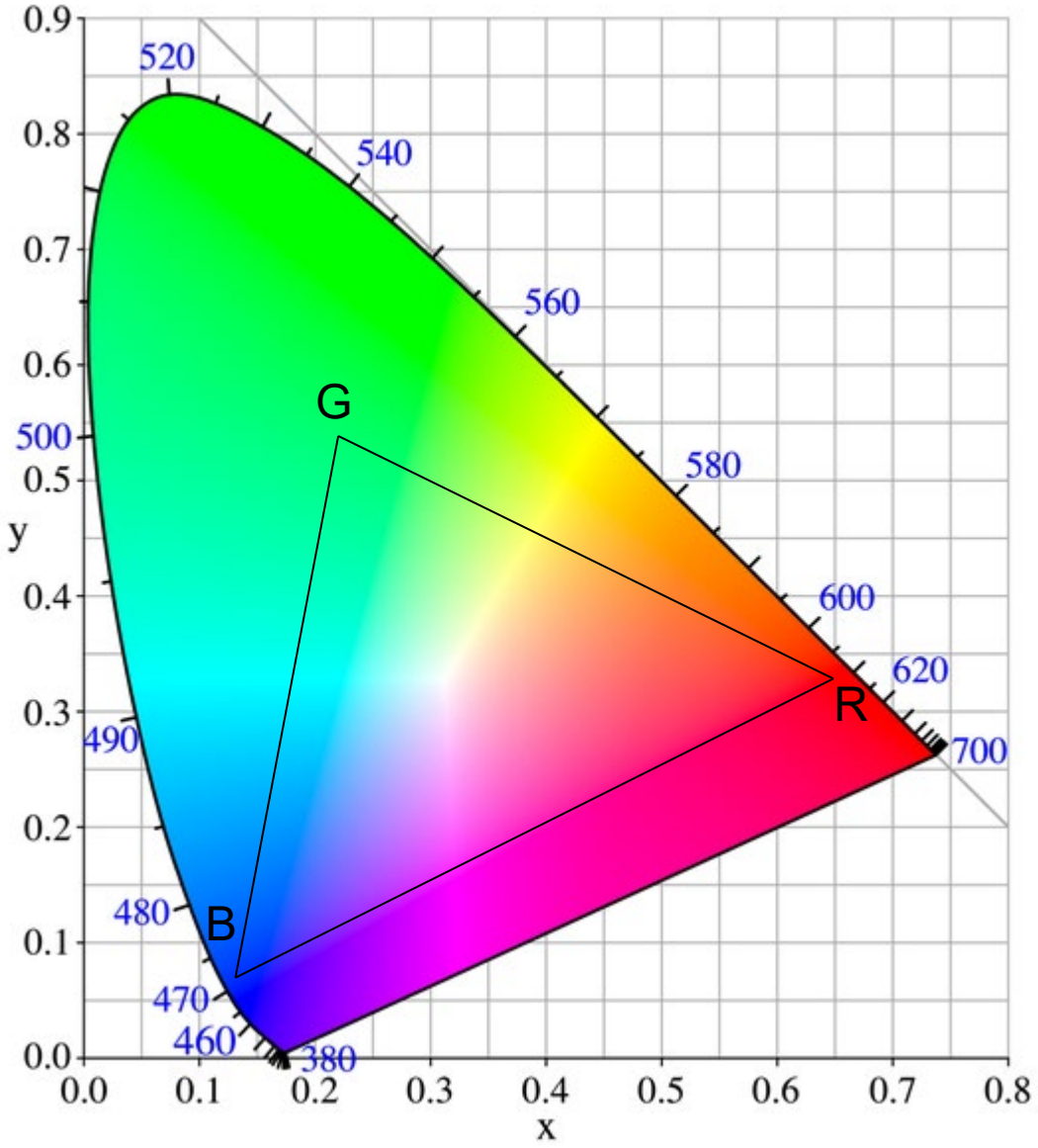
Colour model

- Each light source is a vector with Red, Green, Blue component (I_{lR}, I_{lG}, I_{lB})
- Calculates each component separately:

$$I_R = k_{aR} I_{aR} + \sum_{i=1}^{n} I_{lRi} [k_{dR} (\mathbf{N} \cdot \mathbf{L}_i) + W_R(\theta_i)(\mathbf{V} \cdot \mathbf{R}_i)^{n_{sR}}]$$

$$I_G = k_{aG} I_{aG} + \sum_{i=1}^{n} I_{lGi} [k_{dG} (\mathbf{N} \cdot \mathbf{L}_i) + W_G(\theta_i)(\mathbf{V} \cdot \mathbf{R}_i)^{n_{sG}}]$$

$$I_B = k_{aB} I_{aB} + \sum_{i=1}^{n} I_{lBi} [k_{dB} (\mathbf{N} \cdot \mathbf{L}_i) + W_B(\theta_i)(\mathbf{V} \cdot \mathbf{R}_i)^{n_{sB}}]$$



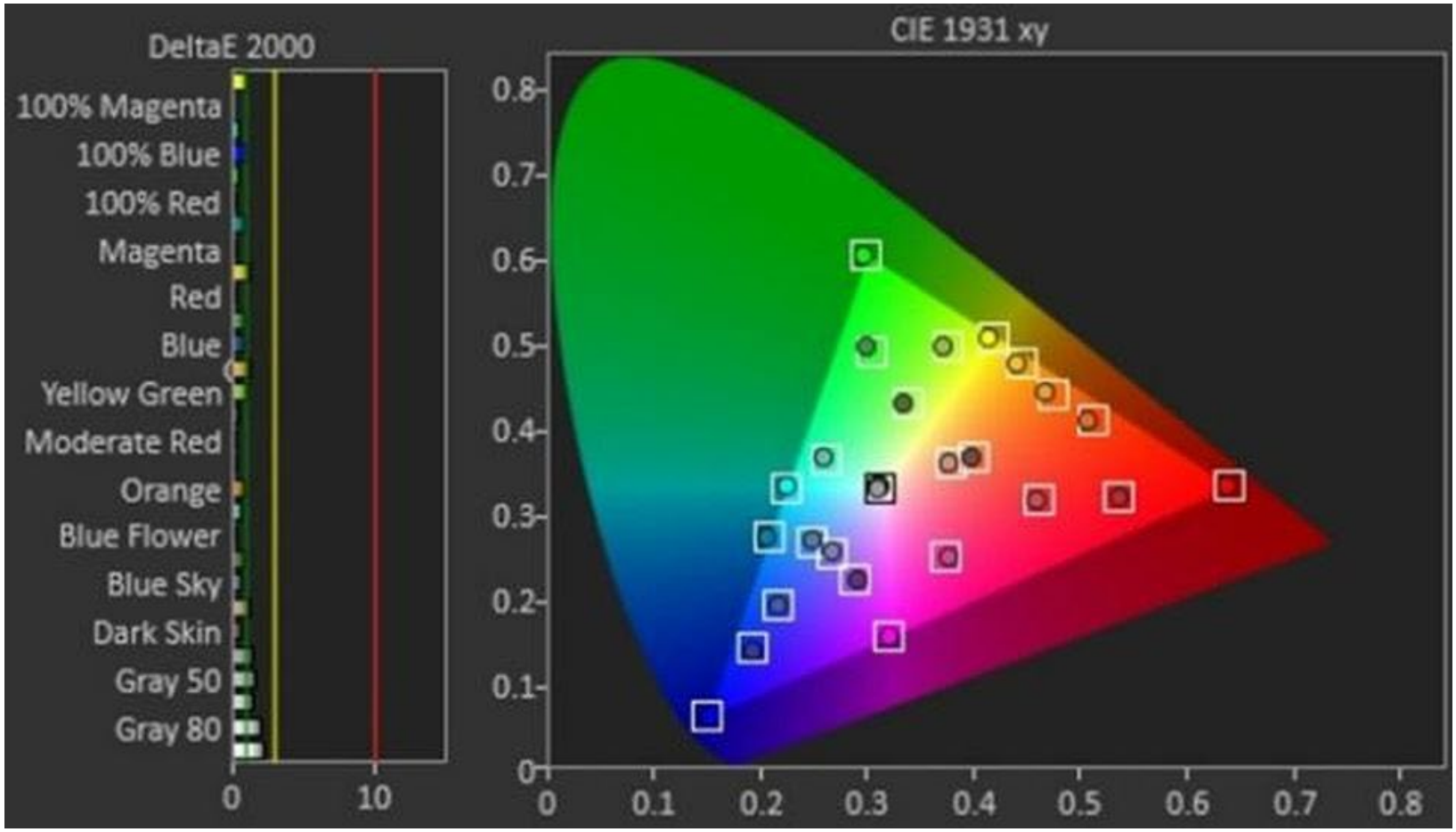
Note:

Only colours in the triangle is displayable.

Some naturally occurring colours outside the triangle cannot be displayed!

Quattron technology uses 4 primary
Colours RYGB that extends the displayble colours

CIE chromaticity diagram
-Represent all possible colours seeable by humans


DeltaE 2000
100% Magenta
100% Blue
100% Red
Magenta
Red
Blue
Yellow Green
Moderate Red
Orange
Blue Flower
Blue Sky
Dark Skin
Gray 50
Gray 80
0
10
CIE 1931 xy
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8

LG-32UD59-B

Shading Models /Rendering Models

- Input : Object tessellated into polygons (standard graphics object)

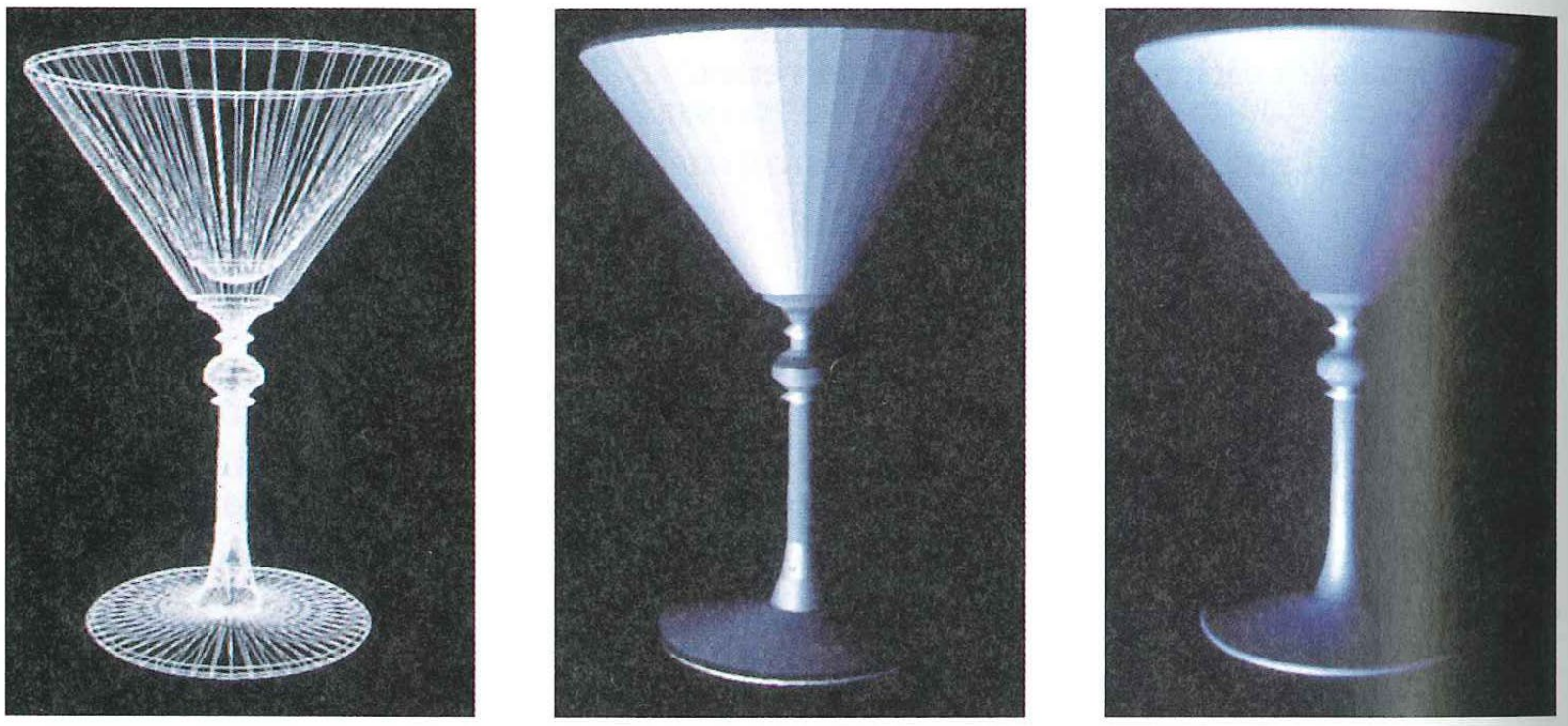
- Three common ways to shade the polygons:
  - Flat Shading
  - Gouraud Shading
  - Phong Shading

Increasing realism

Increasing computational cost

Flat shading

- A single intensity is calculated for the polygon. All points of the polygon are then displayed with the same intensity value
- Fast (Adv.)
- Faceted look - ugly!
- Human vision is subject to “Mach band effect” – intensity discontinuities are accentuated. This amplifies the edges of the polygons, which is undesirable



(a) (b) (c)

A polygon mesh approximation of an object (a) is displayed using flat surface rendering in (b) and using Gouraud surface rendering in (c).

Gouraud shading

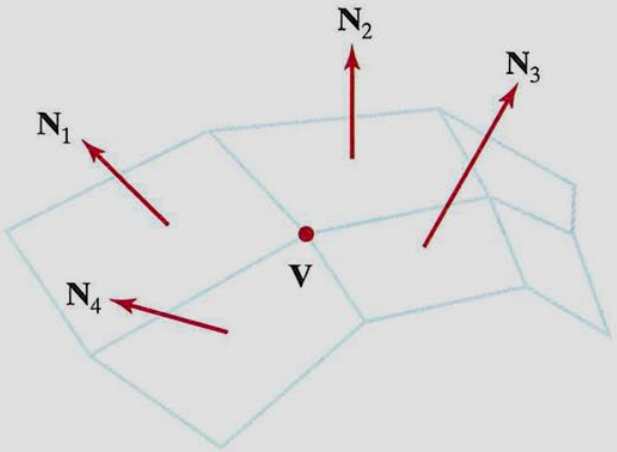
- Linearly interpolate **intensity values** across each polygon
- Intensities for each polygon are matched with the values of adjacent polygons along the common edges
- Interpolation eliminates the intensity discontinuities that occur in flat shading
- Slower (disadv.)
- Smooth out specular highlights (disadv.)

- Step 1 : Determine the average unit normal vector at each polygon vertex

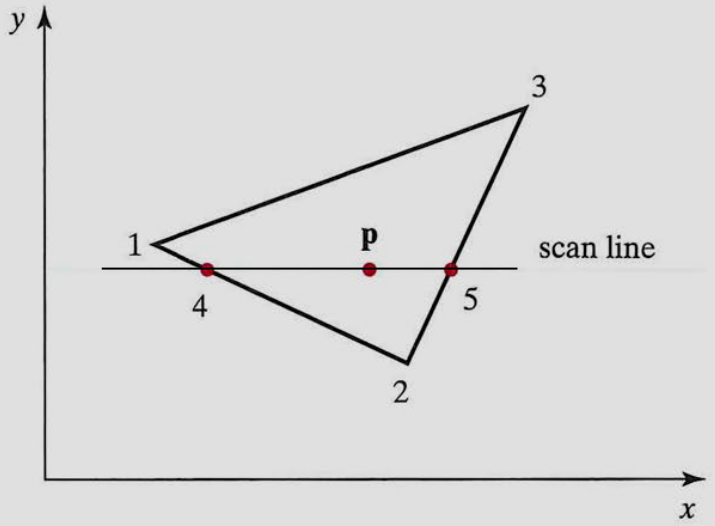
$$\mathbf{N}_v = \frac{\sum_{k=1}^{n} \mathbf{N}_k}{\left| \sum_{k=1}^{n} \mathbf{N}_k \right|}$$

(each $\mathbf{N}_k$ is a unit vector, $\mathbf{N}_v$ is a unit vector by def.)

- Step 2 : Apply an illumination model to each vertex to calculate the vertex intensity
- Step 3 : linearly interpolate the vertex intensities over the surface of the polygon



The normal vector at vertex **V** is calculated as the average of the surface normals for each polygon sharing that vertex.



For Gouraud surface rendering, 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 positions 4 and 5.

Linear Interpolation

- Points lying on an edge of the polygon : linearly interpolate between two endpoints

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

- interior points of the polygon : linearly interpolate across the scan line

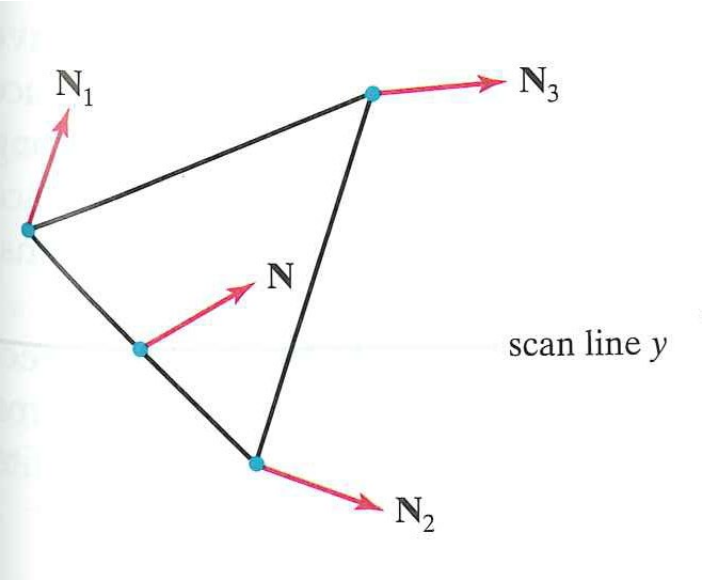
$$I_p = \frac{x_5 - x_p}{x_5 - x_4} I_4 + \frac{x_p - x_4}{x_5 - x_4} I_5$$

Phong shading

- Similar to Gouraud shading, but interpolates **normal vectors** instead.
- Captures specular highlights
- Highest realism
- Slowest (disadv.)

- Step 1 : determine the average unit normal vector at each polygon vertex

$$\mathbf{N} = \frac{y - y_2}{y_1 - y_2}\mathbf{N}_1 + \frac{y_1 - y}{y_1 - y_2}\mathbf{N}_2$$



- Step 2 : linearly interpolate the vertex normals over the surface of the polygon
- Step 3 : apply an illumination model to calculate pixel intensities of each surface point

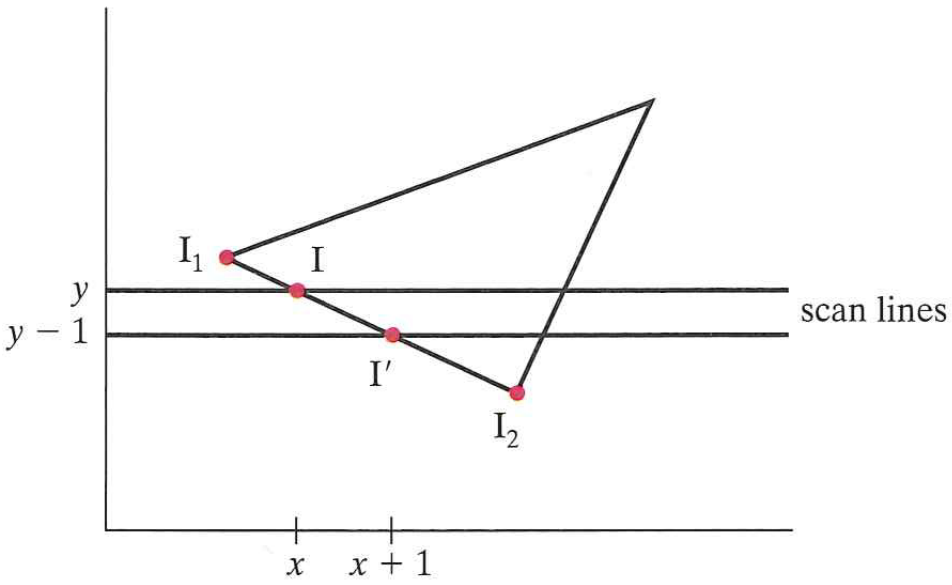
Incremental form

- Linear interpolation equation is expressed in incremental form to save computation:

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

one scan line down

$$I(y-1) = I(y) + \frac{I_2 - I_1}{y_1 - y_2}$$



OpenGL Functions : Lighting

```
glEnable (GL_LIGHTING);  // activate lighting routines

glLight* (lightName, lightProperty, propertyValue);

GLfloat  light1PosType [ ] = {2.0, 0.0, 3.0, 1.0};    // point
                                //  source; the last entry is 1.0
GLfloat  light2PosType [ ] = {0.0, 1.0, 0.0, 0.0};    // light
                               // direction; the last entry is 0.0

glLightfv (GL_LIGHT1, GL_POSITION, light1PosType);  // v
    for vector
glEnable (GL_LIGHT1);

glLightfv (GL_LIGHT2, GL_POSITION, light2PosType);
glEnable (GL_LIGHT2);
```

Light source colour

- (R, G, B, A) A stands for alpha value

```
GLfloat blackColor [ ] = {0.0, 0.0, 0.0, 1.0};
GLfloat whiteColor [ ] = {1.0, 1.0, 1.0, 1.0};

glLightfv (GL_LIGHT3, GL_AMBIENT, blackColor);
glLightfv (GL_LIGHT3, GL_DIFFUSE, whiteColor);
glLightfv (GL_LIGHT3, GL_SPECULAR, whiteColor);
```

Surface Property

glMaterial (surfFace, surfProperty, propertyValue);*

diffuseCoeff [ ] = {0.2, 0.4, 0.9, 1.0}; // kdR = 0.2, kdG = 0.4, kdB = 0.9
specularCoeff [ ] = {1.0, 1.0, 1.0, 1.0}; // $W_R(\theta)$ = 1.0, …

glMaterialfv (GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, diffuseCoeff);
glMaterialfv (GL_FRONT_AND_BACK, GL_SPECULAR, specularCoeff);
glMaterialf (GL_FRONT_AND_BACK, GL_SHININESS, 25.0); // n_s = 25

Surface Rendering

- FLAT and Gouraud Shading

  glShadeModel (surfRenderingMethod);

  surfRenderingMethod = *GL_FLAT* Flat shading

  = *GL_SMOOTH* Gouraud

- Calculating normals

  glNormal3 (Nx, Ny, Nz);*

Gouraud shade a triangle

```
glEnable (GL_NORMALIZE); // convert all normal vectors to unit vector
glLightModeli (GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);
                    // set correct V for specular calculations

glBegin (GL_TRIANGLES);
      glNormal3fv (normalVector1); // normal vector at vertex1 calculated
                                   // by average unit normal vector
      glVertex3fv  (vertex1);
      glNormal3fv (normalVector2);
      glVertex3fv  (vertex2);
      glNormal3fv (normalVector3);
      glVertex3fv  (vertex3);
glEnd ( );
```

Lighting and Rasterization - Visible Surface Determination

Intended Learning Outcomes

- Understand the goal of visible surface determination
- Describe the method of back-face detection
- Describe the method of Z buffer method
- Describe the method of ray casting
- Able to program visible surface determination techniques

Visible Surface Detection

- Also called Hidden Surface Elimination
- Only visible surfaces should be rasterized
- The problem is not easy as has to handle partially visible scenarios –
  - Concave objects
  - one object partially in front of each other

Three Methods

- Back-face detection (also called Culling)
- Z buffer (also called depth buffer)
- Ray Casting

- Back-face detection is always run as a preliminary test. It is fast and reduces about half of the workload before further processing.

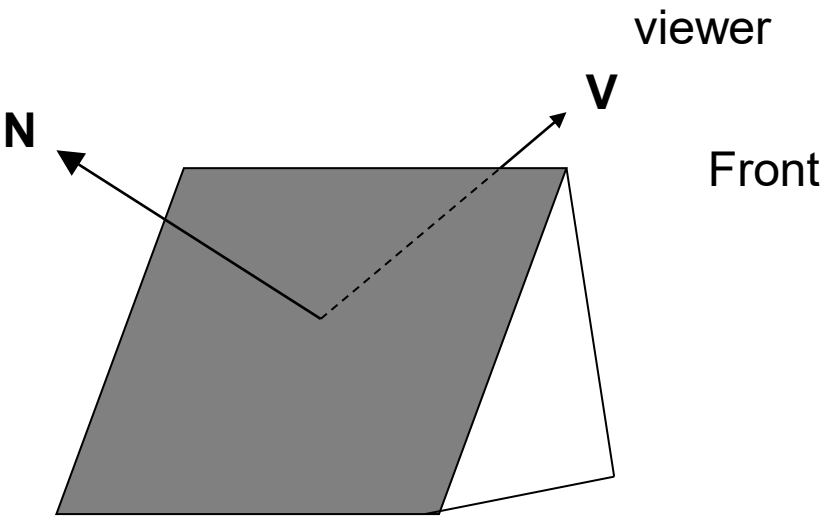
- Other methods also exist: e.g. painter's algorithm, A buffer method, …

Back-Face Detection /Culling

- Fast and simple
- Use as a preliminary step before more sophisticated visibility tests
- Eliminates ≈ 50% of faces from further consideration

$\mathbf{N}\cdot\mathbf{V} < 0 \Rightarrow$ back face

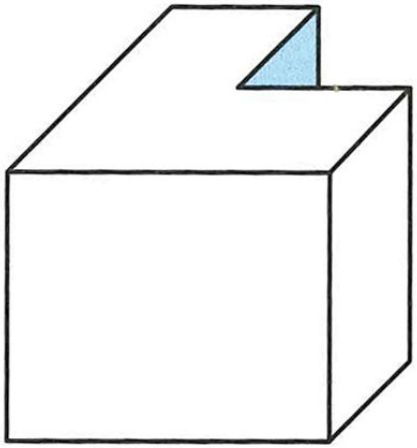
$\Rightarrow$ eliminate



Looking from the back

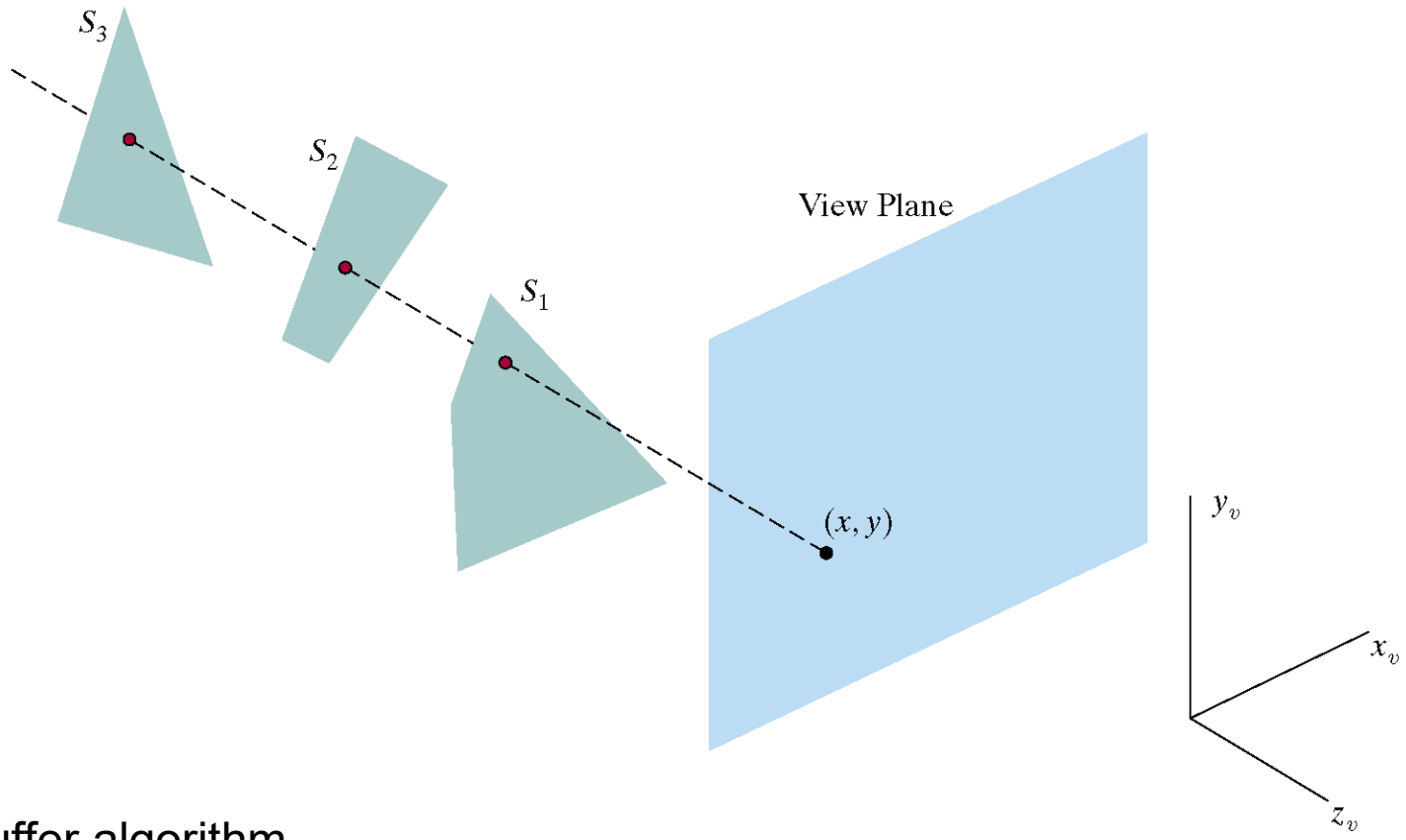
- Sometimes, **v** is replaced by the **VPN** for faster approximate processing

- Disadvantage: cannot handle concave object or partially overlapping object



Z Buffer

- Also called depth buffer method
- Two buffers
  - *Z* /Depth buffer : store depth values for each (x, y) position
  - *Frame* /*Refresh* buffer : store colour values for each (x,y) position
- Buffer stores the current visible surface information, values are updated as soon as new visible information found



Z buffer algorithm

Three surfaces overlapping pixel position (x, y)on the view plane. The visible surface, S_1, has the smallest depth value.

Algorithm

1. Initialize the depth buffer and frame buffer so that for all buffer positions (x, y)

   depthBuff (x, y) = 1.0, frameBuff (x, y) = backgndColor

2. Process each polygon in a scene, one at a time.
   a. for each projected (x, y) pixel position of a polygon, calculate the depth z (if not already known).
   b. If z < depthBuff (x, y), compute the surface colour at that position and set

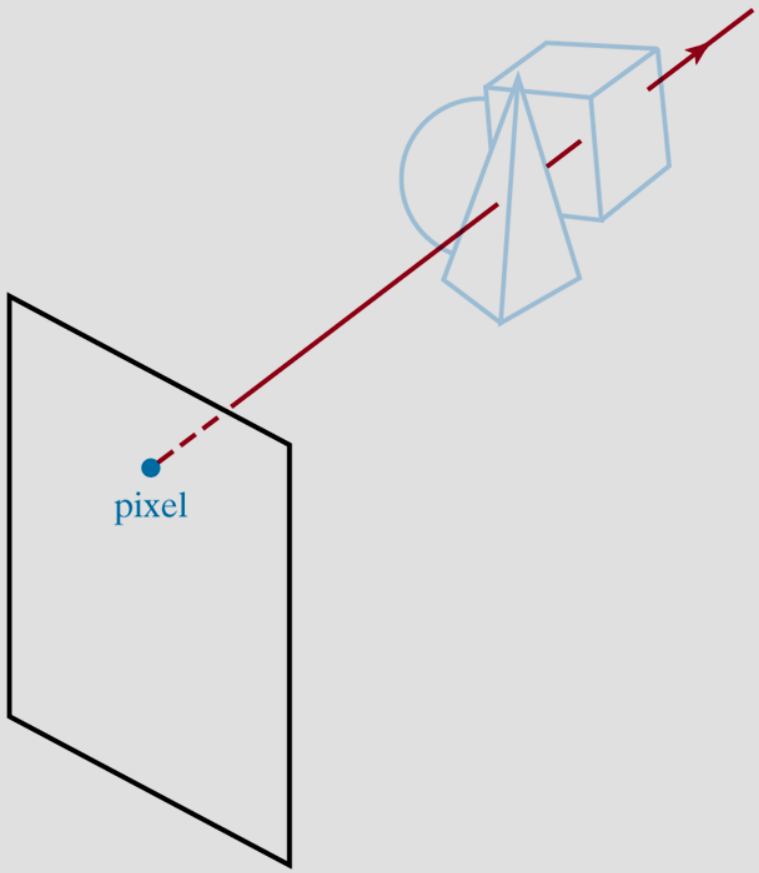
   depthBuff (x, y) = z, frameBuff (x, y) = surfColor (x, y)

   After all surfaces have been processed, the depth buffer contains depth values for the visible surfaces and the frame buffer contains the corresponding colour values for those surfaces.

Ray Casting

- retrace the light paths of the rays that arrive at the pixel

- for each pixel, send a ray from PRP that goes through the pixel
- find all intersections of the ray with the surfaces
- the nearest intersections is the visible part of the surface for that pixel



Ray casting

A ray along the line of sight from a pixel position through a scene.

Comparison of Z buffer and Ray Casting

Method	Good for situations
Z buffer	Objects that cannot be adequately described by simple equations
Ray casting	Objects that can easily be described by simple equations

OpenGL Functions

- Back face removal

  glEnable (GL_CULL_FACE);

  glCullFace (GL_BACK);

- Z Buffer

  glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);

  glClear (GL_DEPTH_BUFFER_BIT);

  glEnable (GL_DEPTH_TEST);

Lighting and Rasterization – Creating Shadows

Intended Learning Outcomes

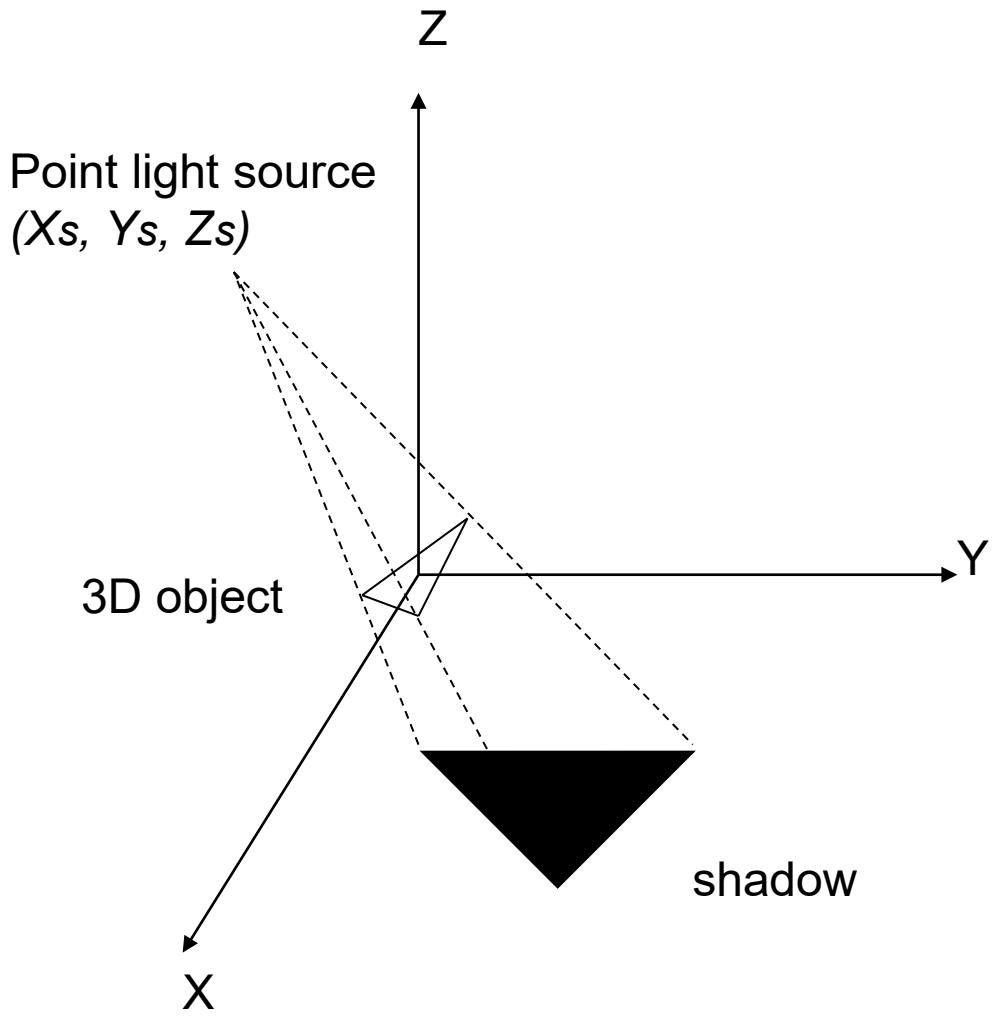
- Apply fast techniques to generate realistic shadow on the ground plane and its programming implementation
- Extend ray casting technique for general shadow creation
- Apply shadow mapping for general shadow creation

Creating Shadow on Plane

- Works only when projecting objects onto a plane and point light source

- Idea : Given a point light source s and a plane P,

1. Render the objects normally.
2. Use a coordinate system transformation that transforms s to the PRP and P to the image plane.
3. Set the object colour to the shadow colour
4. Perspective project the objects onto the image plane, creating shadows.
5. Use the inverse coordinate system transformation to transform the shadows to the normal coordinate system


Z
Point light source
(Xs, Ys, Zs)
We are the viewer
Y
3D object
X-Y is the plane
shadow
X

Coordinate transformation such that the light source becomes the origin

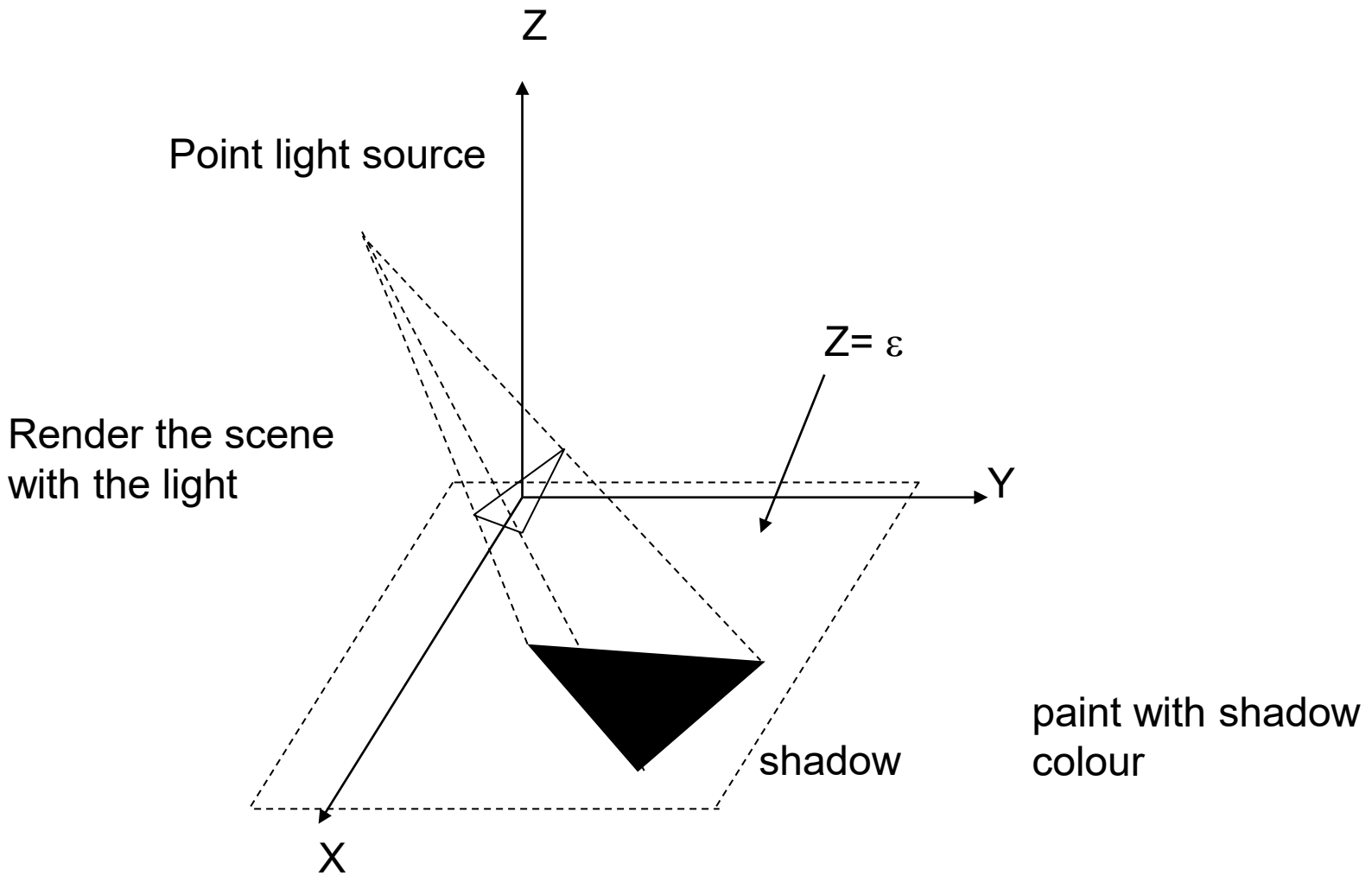
$\mathbf{M}_{s \leftarrow WC} = \mathbf{T}(-X_s, -Y_s, -Z_s)$

Perspective projection

$$\mathbf{M} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{-Z_s} & 0 \end{pmatrix}$$

OpenGL code

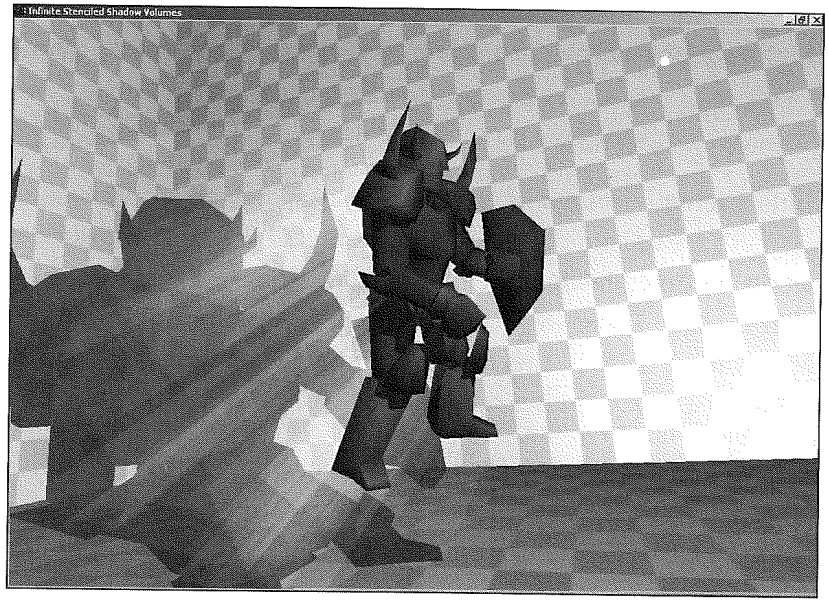
```
GLfloat  light1PosType [ ] = {Xs, Ys, Zs, 1.0};
                    :
GLfloat  M[16];                 // OpenGL is in column major format
                                //  though C is in row major format
for (i=0; i<16; i++)
     M[i]=0;
M[0]=M[5]=M[10]=1;
M[11]=-1.0/Zs;

object ( );                     // draw the objects

glPushMatrix ( );               //  save state

glMatrixMode (GL_MODELVIEW);
glTranslatef (Xs, Ys, Zs);      //  Mwc←s
glMultMatrixf (M);              //   perspective project
glTranslatef (-Xs, -Ys, -Zs);   //  Ms←wc

glColor3fv (shadowcolour);      //  set ka = kd = ks = 0 if you are using lighting model
object ( );

glPopMatrix ( );                //  restore state
```

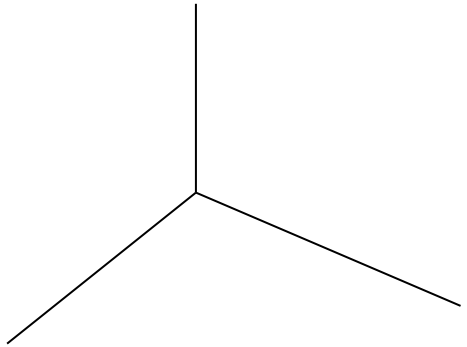


Implementation notes: in actual programming, cast the shadow on a plane $Z = \varepsilon$ after changing to the light source coordinate system, where ε is a very small number (why?)

Extension to corners

- It can be used to cast shadows on corners of the room (treat it as three planes)





- However, it cannot be used to cast shadows on general non-plane objects

General Shadow creation

Limitations:

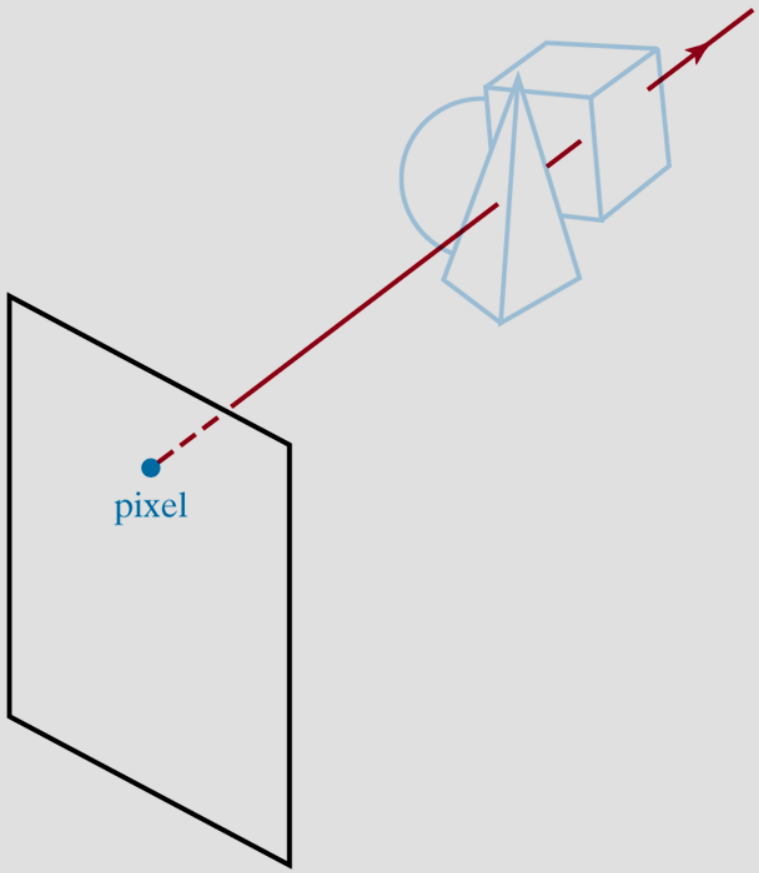
- Up to now, shadows can only be casted on planes or corners
- No shade differences for overlapping shadows
- The shadow boundary is too sharp

- Below we introduce two techniques: ray casting using shadow ray and shadow mapping, that overcome the first two limitations.
- One way to create soft shadows is radiosity, which is a sophisticated model of ambient reflection

Ray Casting

- retrace the light paths of the rays that arrive at the pixel

- for each pixel, send a ray from PRP that goes through the pixel
- find all intersections of the ray with the surfaces
- the nearest intersections is the visible part of the surface for that pixel



Ray casting

A ray along the line of sight from a pixel position through a scene.

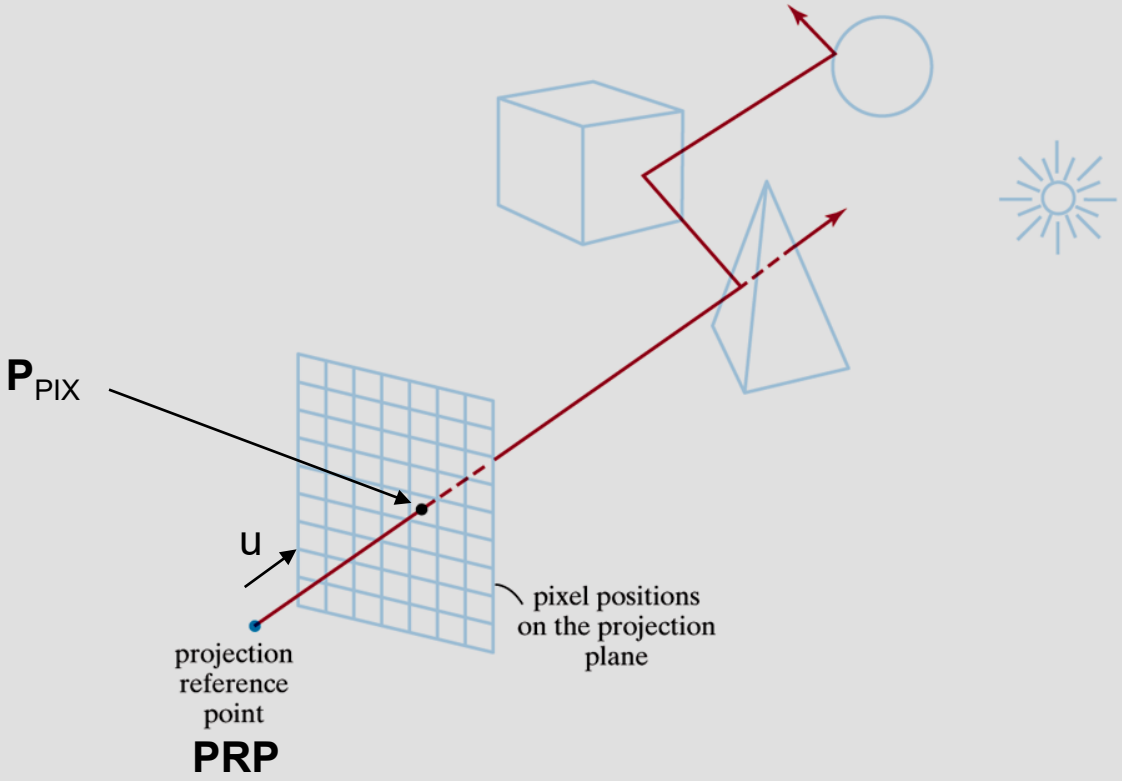
- Consider the math.

$$\boldsymbol{P} = \boldsymbol{P}_0 + s\,\boldsymbol{u} \qquad \text{(Pixel Ray Equation)}$$

$\boldsymbol{P}_0$ may either be $\boldsymbol{P}_{PRP}$ or $\boldsymbol{P}_{PIX}$

$\boldsymbol{P}_{pix}$ is the (X, Y, Z) coordinates of the pixel

$$\mathbf{u} = \frac{\mathbf{P}_{PIX} - \mathbf{P}_{PRP}}{\left|\mathbf{P}_{PIX} - \mathbf{P}_{PRP}\right|}$$ is a unit vector pointing out from **PRP**



Multiple reflection and transmission paths for a ray from the projection reference point through a pixel position and on into a scene containing several objects.

Ray – Surface Intersections

- Suppose the CG scene consists of n surfaces or polygons
- Compute the intersection point(s) of the pixel ray with each of the n surfaces/polygons
- The surface/polygons whose intersection point has the smallest s is the visible surface
- since it is the nearest

Ray – Sphere Intersection

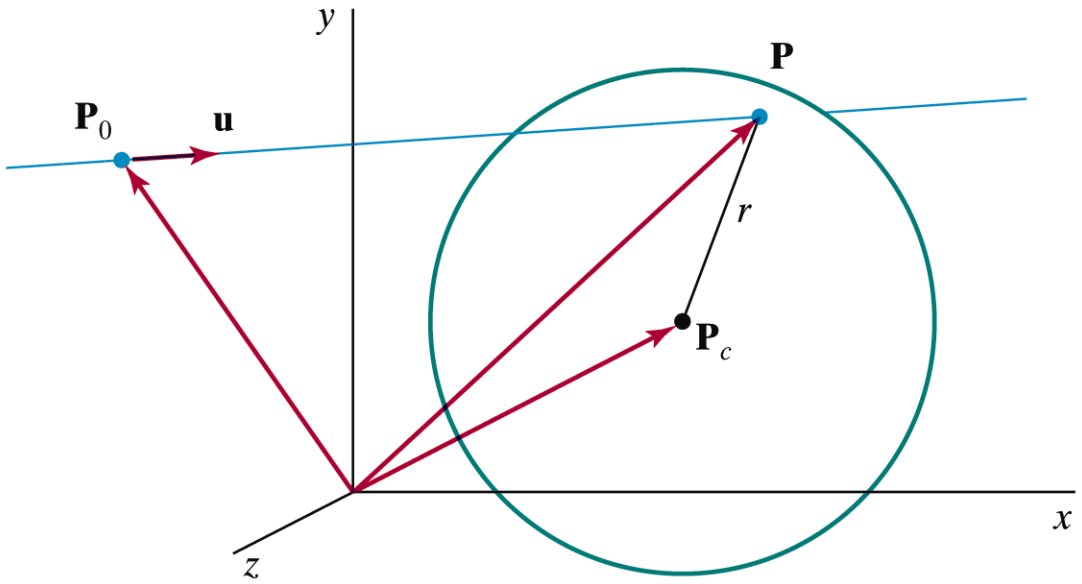
- Sphere is the simplest surface with analytical equation

$$(X - X_C)^2 + (Y - Y_C)^2 + (Z - Z_C)^2 = r^2$$

$\mathbf{P}_C(X_C, Y_C, Z_C)$ Center

r Radius

$$\left|\mathbf{P} - \mathbf{P}_c\right|^2 - r^2 = 0$$ Vector Equation



A ray intersecting a sphere with radius r and center position $\mathbf{P}_c$.

Ray-Sphere Intersections (2)

- Sub. $\mathbf{P} = \mathbf{P}_0 + s\mathbf{u}$ gives a quadratic equation
- Solution:

$$s = \mathbf{u} \cdot \Delta\mathbf{P} \pm \sqrt{r^2 - \left|\Delta\mathbf{P} - (\mathbf{u} \cdot \Delta\mathbf{P})\mathbf{u}\right|^2} \qquad \Delta\mathbf{P} = \mathbf{P}_C - \mathbf{P}_0$$

- If discriminant < 0, does not intersect
- Otherwise choose the intersection with the smaller s

- Solution is more difficult and time consuming for more complicated surfaces

Shadowing

- The ray casting method above can be used to determine the visible surface
- For each visible surface point **P**, the question is how to determine whether **P** is in shadow
- Given a set of light sources, **P** can be in shadow to a subset of light sources and illuminated by the rest
- If in shadow with respect to source $\mathbf{S_o}$, then the light intensity due to $\mathbf{S_o}$ is set to zero

Shadow ray

- To test whether **P** is in shadow w.r.t. a point light source $\mathbf{S_o}$:
- Send a pixel ray from **P** to $\mathbf{S_o}$
- If the pixel ray intersects ANY surface /polygon on its way, **P** is in shadow w.r.t. $\mathbf{S_o}$
- The pixel ray is called “shadow ray”

Shadow Mapping

- Idea: A point is in shadow iff it is not visible to the light source (a visibility determination problem)
- Change the coordinate system such that the light position is the PRP. We call this the lighting coordinate system
- Perform a perspective projection. Keep the depth buffer. The depth buffer holds, for each pixel, the nearest distance to the light source
- For each 3D point to be rendered, change to the lighting coordinate system.
- Project the point. Compare its depth to the value in the depth buffer.
- The point is in shadow if it is not the same value as that in the depth buffer.

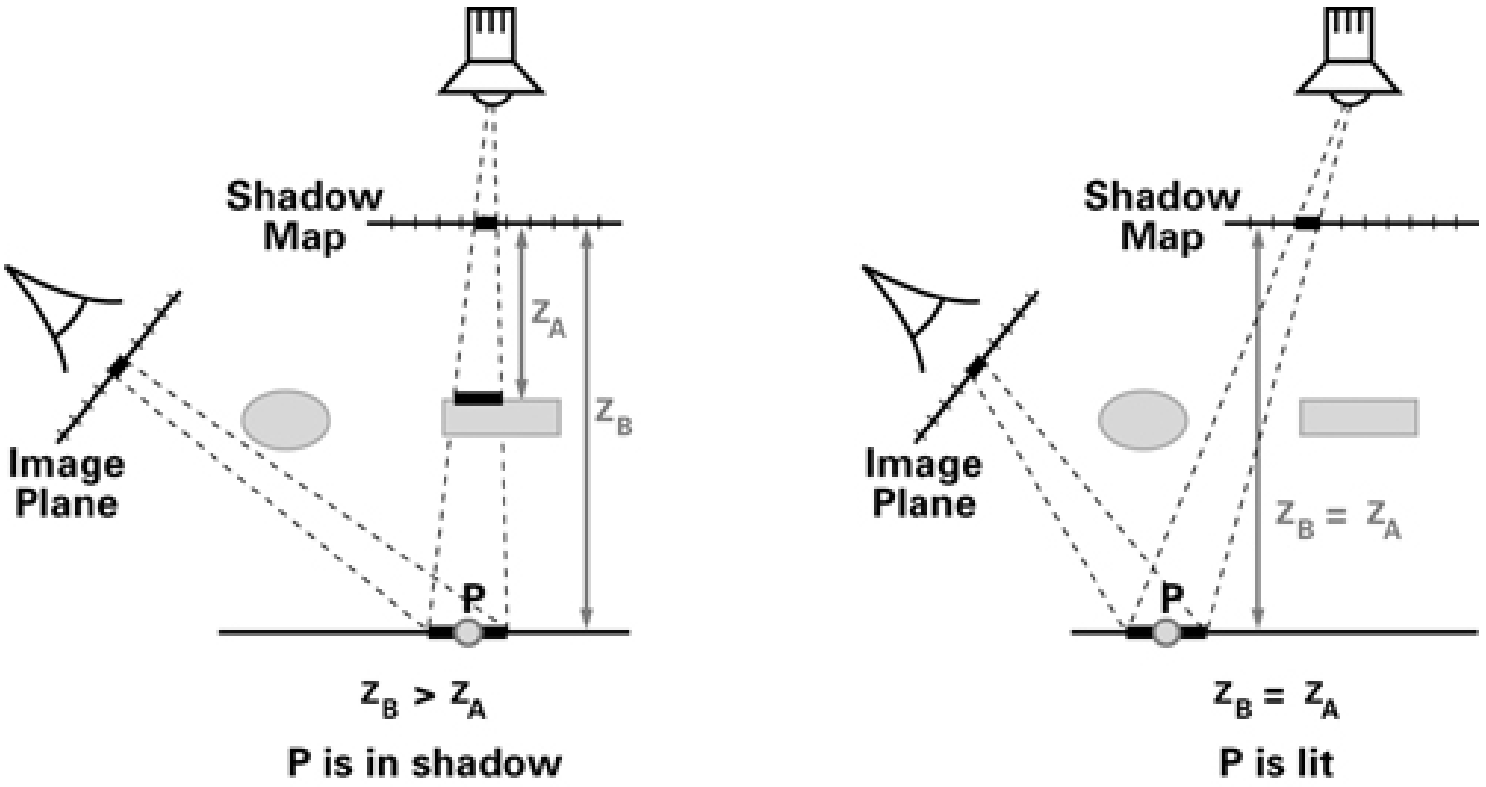


Figure from
http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter09.html

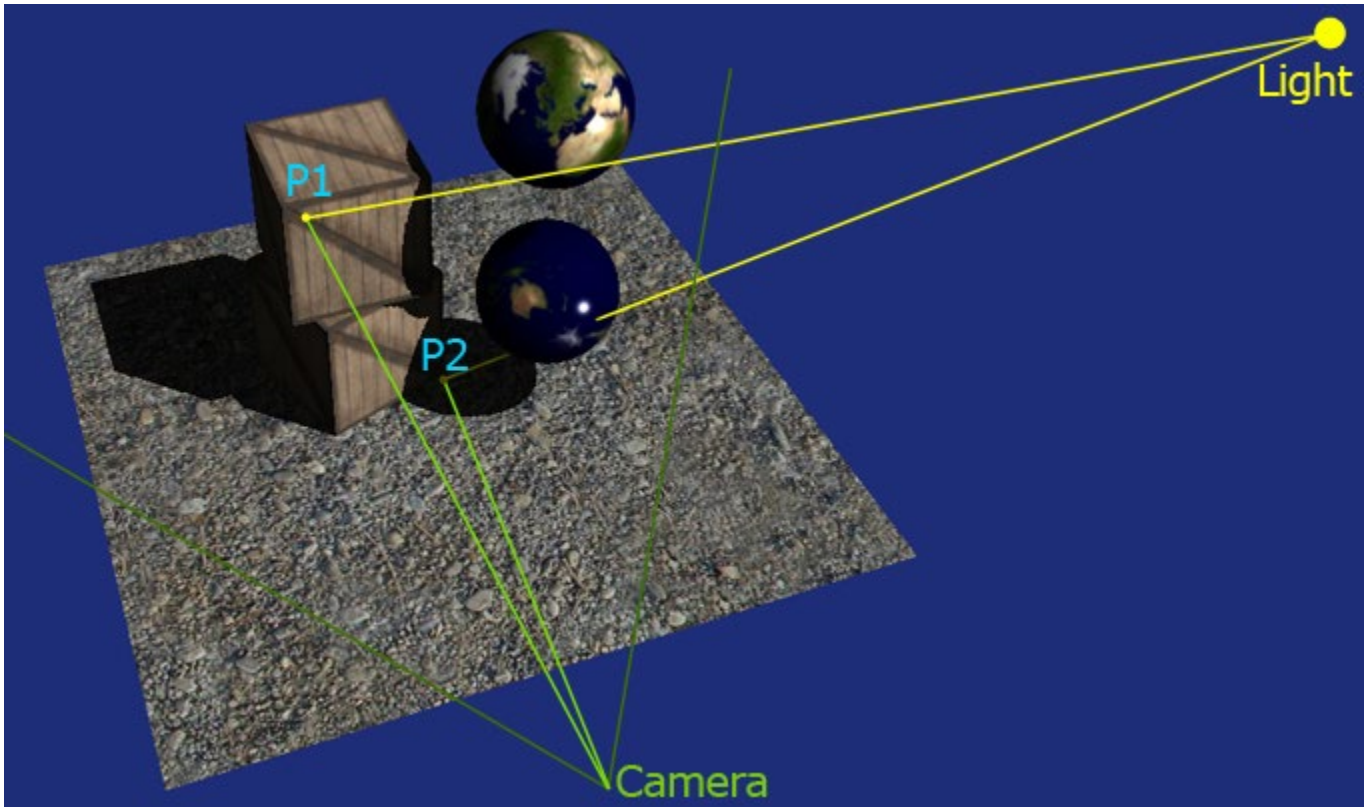


Figure from
http://www.codinglabs.net/tutorial_opengl_deferred_rendering_shadow_mapping.aspx

Texture and Other Mapping Techniques

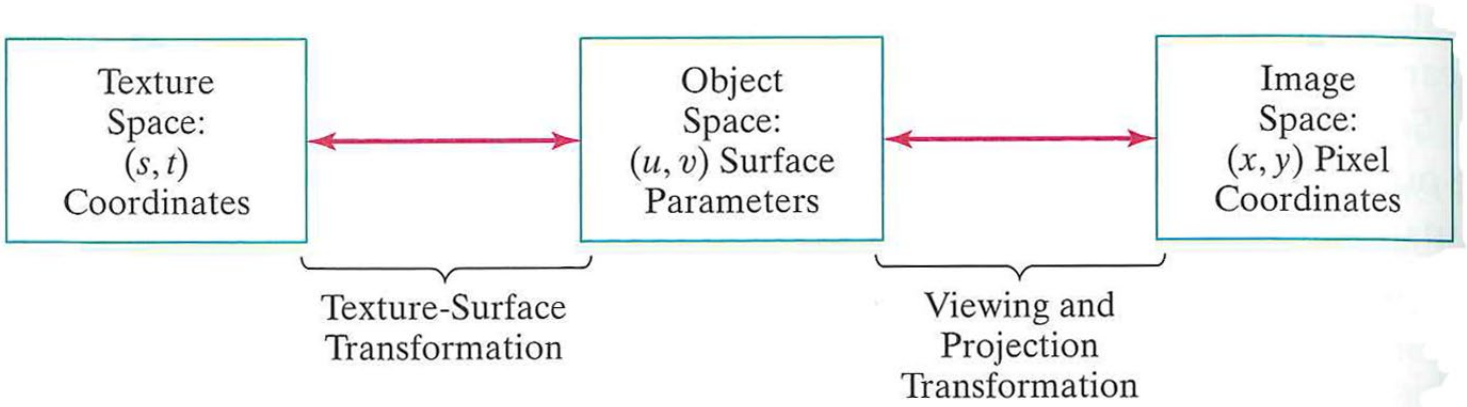
Intended Learning Outcomes

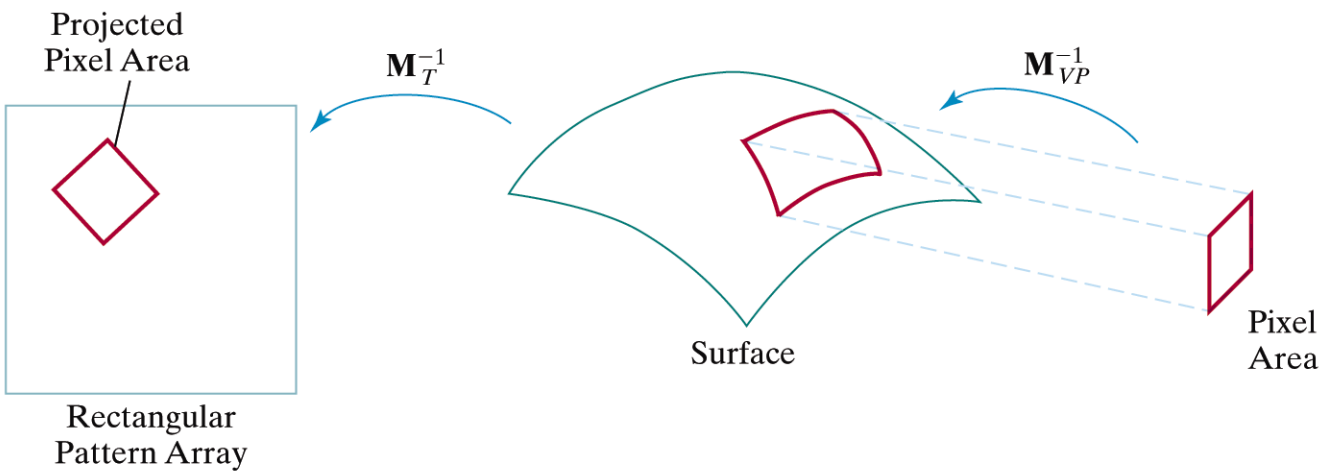
- Able to apply pixel order scanning for generating texture
- Describe and apply other advanced mapping methods

Two methods of texture mapping

- Texture scanning : map texture pattern in (s, t) to pixel (x, y). Left to right in Fig. below

- pixel order scanning : map pixel (x,y) to texture pattern in (s, t). Right to left in Fig. below





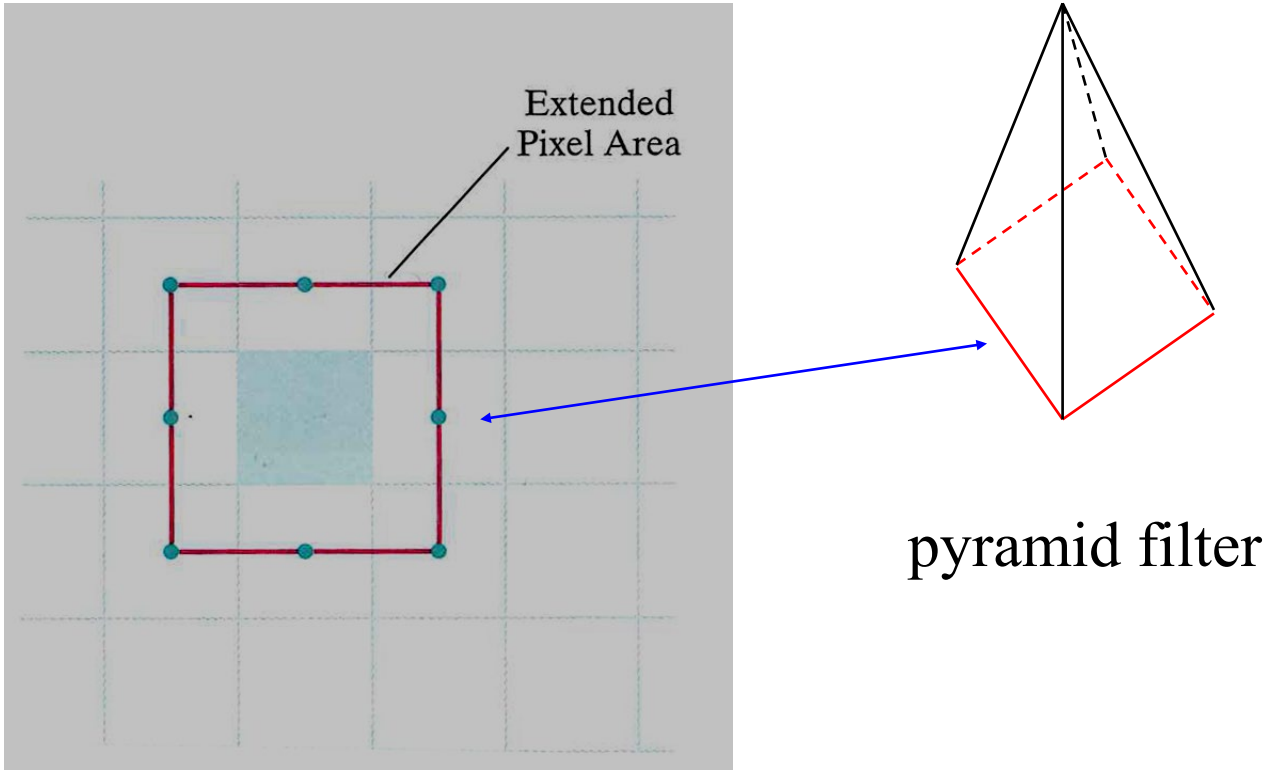
Pixel order scanning

- To simplify calculations, the mapping from texture space to object space is often specified with linear functions:

$$u = f_u(s,t) = a_u s + b_u t + c_u$$
$$v = f_v(s,t) = a_v s + b_v t + c_v$$

- The mapping from object space to image space consists of a concatenation of 1) viewing transformation followed by 2) projective transformation.

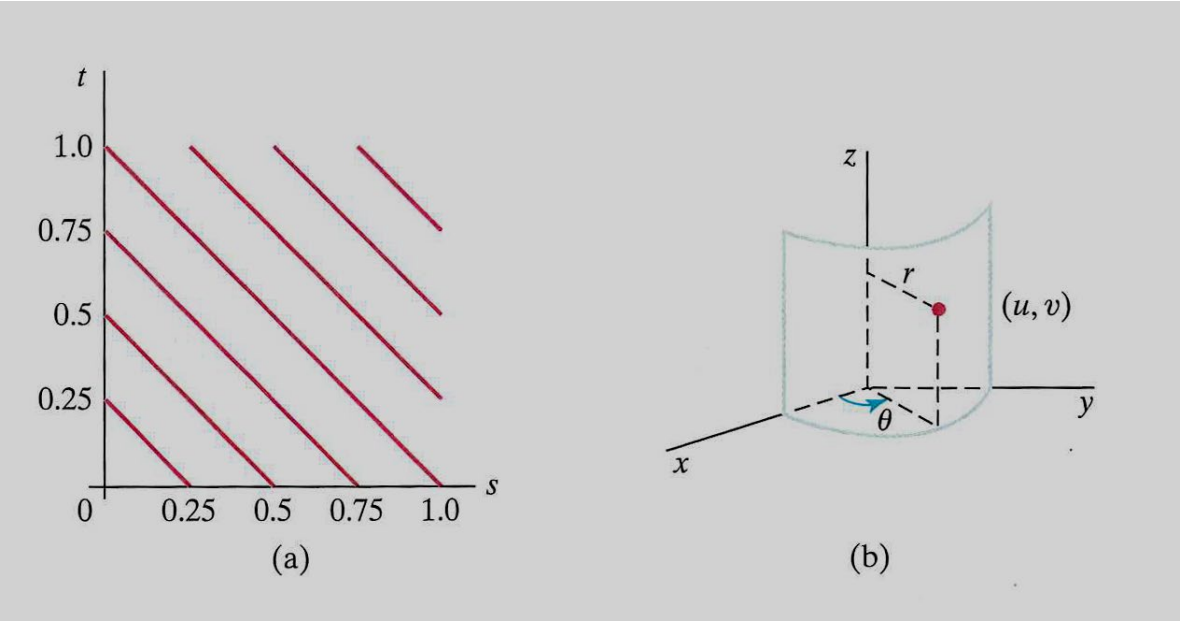
- Texture mapping is not used in practice. Pixel order scanning is used, together with antialiasing, as shown below:



Example: Pixel Order Scanning

- Map texture pattern in Fig. (a) to the cylindrical surface in Fig. (b).
- Parametric representation of the cylindrical surface:

$$X = r\cos u$$
$$Y = r\sin u$$
$$Z = v$$



- Map the texture pattern to the surface by defining the following linear function

$$u = \frac{\pi}{2} s \qquad (1)$$

$$v = t$$

- The above is the texture-surface transformation M_T
- Suppose no geometrical transformation and projection is orthographic with projection direction in the X direction. Then Y-Z is the projection plane
- Viewing and projection transformation M_{VP} is

$$Y = r \sin u \qquad (2)$$

$$Z = v$$

- For pixel order scanning, we need to compute the transformation (Y,Z)→(s, t)
- First compute $\mathbf{M}_{VP}^{-1}$, or (Y, Z)→(u, v). From (2)

$$u = \sin^{-1}\left(\frac{Y}{r}\right)$$
$$v = Z \qquad (3)$$

- Next compute $\mathbf{M}_{T}^{-1}$, or (u, v)→(s, t). From (1)

$$s = \frac{2}{\pi}u$$
$$t = v \qquad (4)$$

- Combining (3) and (4)

$$s = \frac{2}{\pi}\sin^{-1}\left(\frac{Y}{r}\right)$$

$$t = Z$$

- Using this transformation, the pixel area of a pixel (Y, Z) will be back-transformed into an area in the texture space (s, t). Intensity values in this area are averaged to obtain the pixel intensity.

Bump Mapping

- Texture mapping can be used to add fine surface detail to smooth surface. However, it is not a good method for modelling rough surface e.g., oranges, strawberries, since the illumination detail in the texture pattern usually does not correspond to the illumination direction in the scene.
- Bump mapping is a method for creating surface bumpiness. A perturbation function is applied to the surface normal. The perturbed normal is used in the illumination model calculations.

P(u, v) position on a parametric surface

N surface normal at (u, v)

$\mathbf{N} = \mathbf{P}_u \times \mathbf{P}_v$

where $P_u = \frac{\partial P}{\partial u}$ $P_v = \frac{\partial P}{\partial v}$

Add a small bump function b(u, v) to **P**(u, v). It becomes

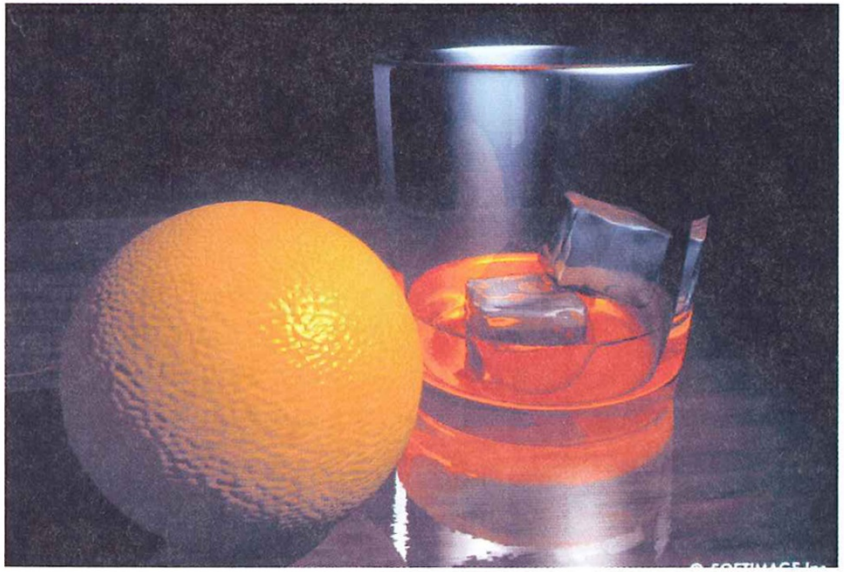
P(u,v) + b(u,v)**n**

where **n** = **N** / |**N**| is the unit (outward) surface normal

The normal $\mathbf{N} = \mathbf{P}_u \times \mathbf{P}_v$ is perturbed.

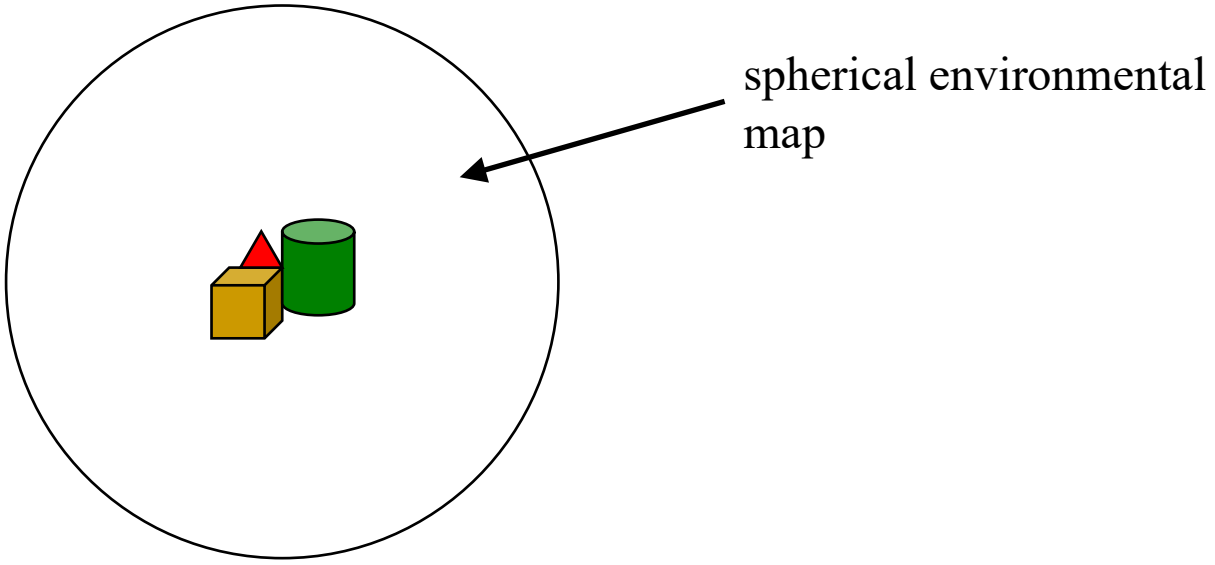
- The bump function b(u, v) are usually obtained by table lookup. It can be setup using

1) Random pattern to model irregular surfaces (e.g. raisin)
2) Repeating pattern to model regular surfaces (e.g. orange Fig. 10-110)



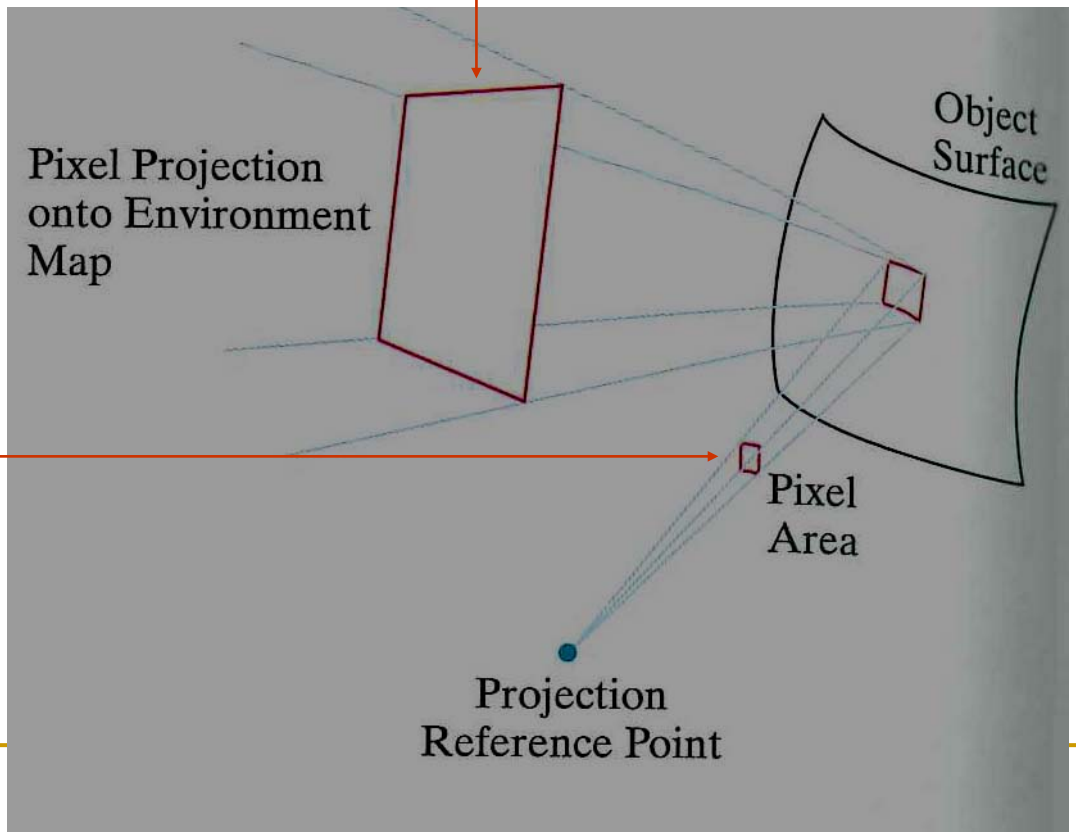
Environment Mapping

- A simplified ray tracing method that uses texture mapping concept.
- Environment map is defined over the surface of an enclosing universe. Information includes intensity values of light sources, the sky or other background objects.



- Run “Example environment map”

- A surface is rendered by projecting the pixel area to the surface, then reflect onto the environment map. If the surface is transparent, also refract onto the map.
- Pixel intensity determined by averaging the intensity values within the intersected region of the environment map.



armour (specular object) reflects the cathedral surrounding
Modelled using environmental map

OpenGL functions

glTexImage2D (GL_TEXTURE_2D, 0, GL_RGBA, texWidth, texHeight, 0, dataFormat, dataType, surfTexArray);

GL_RGBA Each colour of the texture pattern is specified with *(R, G, B, A)* *A* is the alpha parameter:
A = 1.0 $\Rightarrow$ completely transparent
A = 0.0 $\Rightarrow$ opaque

texWidth and *texHeight* is the width and height of the pattern

dataFormat and *dataType* specify the format and type of the texture pattern e.g. *GL_RGBA* and *GL_UNSIGNED_BYTE*

glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST)

glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST)

Specify what to do if the texture is to be magnified (i.e., mag) or reduced (i.e., min) in size:

GL_NEAREST	assigns the nearest texture colour
GL_LINEAR	linear interpolate

glTexCoord2 (sCoord, tCoord);*

A texture pattern is normalized such that s and t are in |0, 1|

A coordinate position in 2-D texture space is selected with

$0.0 \leq sCoord, tCoord \leq 1.0$

glEnable (GL_TEXTURE_2D)

glDisable (GL_TEXTURE_2D)

Enables / disables texture

Example: texture map a quadrilateral

```
GLubyte texArray [808][627][4];

glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_NEAREST);
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);

glTexImage2D (GL_TEXTURE_2D, 0, GL_RGBA, 808, 627, 0, GL_RGBA,
GL_UNSIGNED_BYTE, texArray);

glEnable (GL_TEXTURE_2D);

// assign the full range of texture colors to a quadrilateral
glBegin (GL_QUADS);
        glTexCoord2f (0.0, 0.0);   glVertex3fv (vertex1);
        glTexCoord2f (1.0, 0.0);   glVertex3fv (vertex2);
        glTexCoord2f (1.0, 1.0);   glVertex3fv (vertex3);
        glTexCoord2f (0.0, 1.0);   glVertex3fv (vertex4);
glEnd ( );

glDisable (GL_TEXTURE_2D);
```

Simple example

Use a large QUAD for the ground and texture map it

- To re-use the texture, we can assign a name to it

```
static GLuint    texName;
glGenTextures (1, &texName);  // generate 1 texture with name "texName"

glBindTexture (GL_TEXTURE_2D, texName);
glTexImage2D (GL_TEXTURE_2D, 0, GL_RGBA, 32, 32, 0, GL_RGBA,
GL_UNSIGNED_BYTE, texArray);    // define the texture "texName"
                                    ⋮
 glBindTexture (GL_TEXTURE_2D, texName);  // use it as current texture
```

- We can generate more than 1 name at a time. To generate 6 names:

```
static Gluint texNamesArray [6];
glGenTexures (6, texNamesArray);  // generate 6 texture names
```

- To use *texNamesArray [3]*

```
glBindTexture  (GL_TEXTURE_2D, texNamesArray [3]);
```

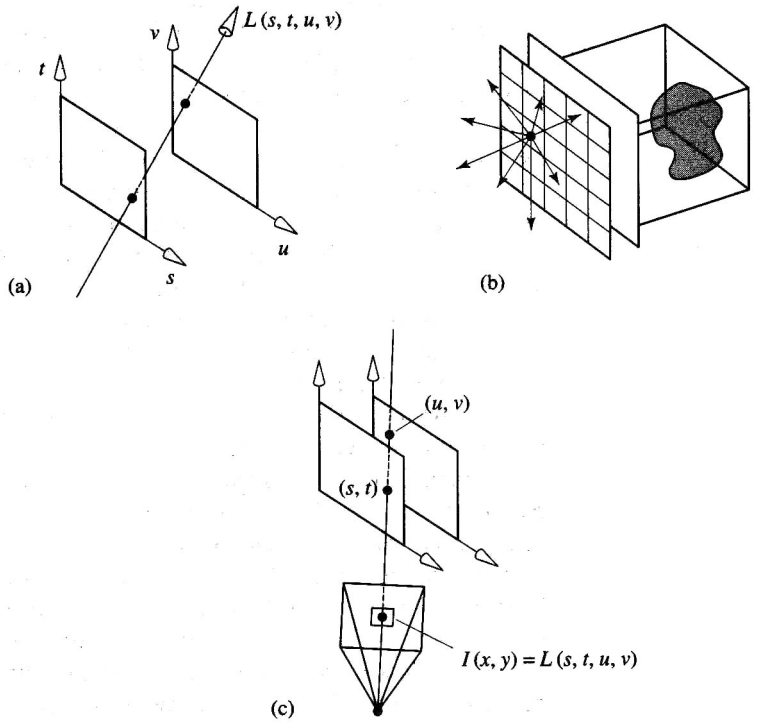
Texture mapping in Movie

- Use texture map to blend graphics object into real movie production
- Double buffering is used
- Frame rate is unimportant as movie is produced off-line
- Human artist can optionally help with later stage production to make image more realistic

Light field (Lumigraph)

- An image based rendering (IBR) approach
- A “pre-computation” idea
- Stores intensity of all rays in all directions
- Uses data compression

- Adv.: Extremely fast
- Disadv.: High Pre-computational cost



Application

- Light field camera

  https://en.wikipedia.org/wiki/Light-field_camera

- Capture instantly. Do not need to focus

Implementation notes

- One may use OpenGL SOIL library or stb_image.h for reading in texture images
- Search the web with keyword “texture images”
- A .raw file is a file with no formatting and only consist of a sequence of numbers. You can read the file into an array in C. *read_rawimage* is an example of how to read a raw image into C. However, it is difficult to find a suitable file converter that converts other file formats to raw file
- It is found that older graphics cards cannot display texture property if the source file is not in 2^n x 2^m