SE3VR11 : Virtual Reality

Professor Paul Sharkey p.m.sharkey@reading.ac.uk

Lecture 2 – 21/01/2016

Course Outline

Split into the following topics:

- **2D/3D Graphics**
  - Object Representations
  - Mathematics involved in 3D Graphics
  - Materials and Texturing
  - Lighting and Shading
  - Rendering Methods
  - Animation
- **3D Modelling Approaches**
- **Scene Graphs and Software**
- **(Distributed Virtual Environments)**

Computer Graphics

Computer graphics deals with all aspects of producing an image using a computer

- Imaging - representing 2D images
- Modelling - representing 3D objects
- Rendering - creating an image from models
- Animation - simulating changes over time

It involves

- Hardware
- Software Libraries
- Software Applications

Computer Graphics

What hardware is needed to create this image?

- **Processing Power**
- **Storage / Memory**
- **Input / Output devices**

Computer Graphics

What software is needed to create this image?

- **Libraries to control the hardware**
- **Generating object data**
  (Modelling)
- **Mathematical models for lighting and materials**
  (Shading)
- **Conversion to something that can be displayed**
  (Rendering)

Computer Graphics

Adding animation and interaction to the graphics...

- **Collisions and intersections**
- **Object manipulation**
- **Key framing**
- **Inverse kinematics**
- **Simulations**

A Brief History of Computer Graphics
1950-1960

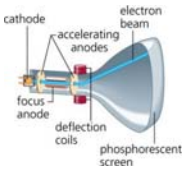
Computer graphics goes back to the earliest days of computing

- Strip charts
- Pen plotters
- Simplistic displays from arrays of lights to 'nixie-tubes'

A Brief History of Computer Graphics
1960-1970

Computer graphics starts improving …

- Wire-frame graphics
- Dedicated display processors
- Storage Tubes
- Cathode-ray-tubes



- Sketchpad – Ivan Sutherland

A Brief History of Computer Graphics
Ivan Sutherland

Ivan Sutherland's PhD thesis at MIT

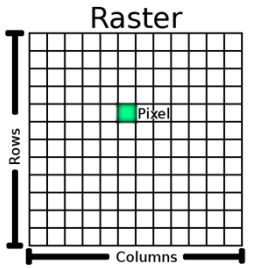
- **Recognition of man and machine interaction**
- **Sketchpad** (predecessor to the GUI)
  - Display image
  - User moves light pen
  - Computer generates new display
  - Loop…

Sutherland created many of the now common algorithms for basic computer graphics

A Brief History of Computer Graphics
1970-1980

Introduction of …

- Raster Graphics



- Graphics standards start forming
- Workstations and PCs

The acceleration of graphics technology really starts to climb

A Brief History of Computer Graphics
1980-1990

Introduction of …

- Special purpose hardware – Silicon Graphics

- Industry standards in place
- Graphics over a network – X Window System
- Human Computer Interfaces (HCI)

A Brief History of Computer Graphics
1990-2000

- **OpenGL API (**application program interface**)**
- **Computer generated feature length films**
- **Graphics techniques at hardware levels**
  - Texture mapping
  - Blending
  - Stencil Buffers – typically used to add shadows
- **More funky SGI**

A Brief History of Computer Graphics
2000 to present

- Photorealism
- Multi core and multi processors dedicated to graphics
- Programmable pipelines
- Real time ray tracing

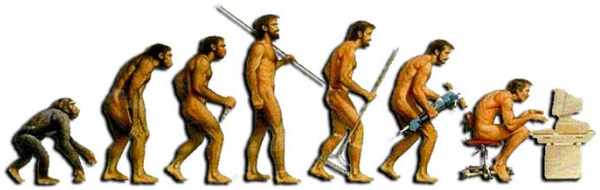
  (...nearly)
- and a LOT more

Computer Graphics … what is it good for?

- Entertainment
- Computer-aided Design (CAD)
- Scientific Visualisation
- Training
- Education
- E-Commerce
- Computer Art
- Virtual Reality
- . . .

Computer Graphics … what is it good for?

- . . .
- wherever your imagination takes you …



Components for 3D Graphics

Outline

- Displaying an Image
  - Hardware
  - Systems
  - Colour Representation
- 3D Scene Representation
  - Intro
  - Coordinate Systems
  - 3D Primatives

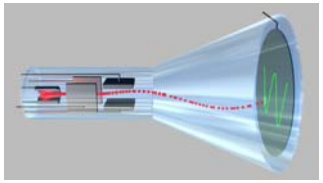
Components for 3D Graphics

Outline

- **Displaying an Image**
  - **Hardware**
  - Systems
  - Colour Representation
- **3D Scene Representation**
  - Intro
  - Coordinate Systems
  - 3D Primitives

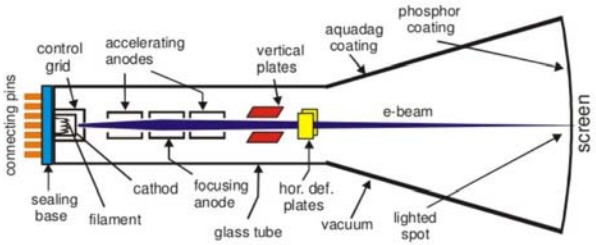
Components for 3D Graphics
Hardware

CRT – Cathode Ray Tube



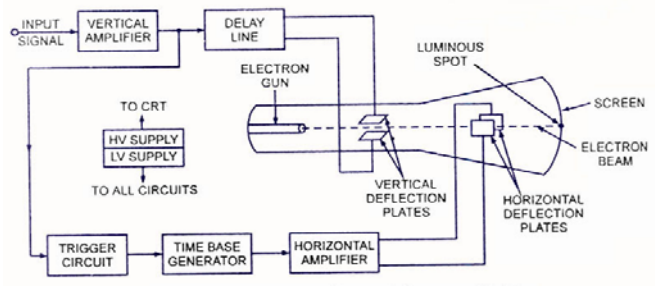
Components for 3D Graphics
Hardware

CRT – Cathode Ray Tube

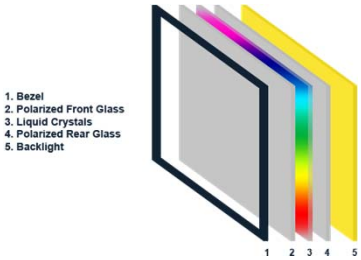


Components for 3D Graphics
Hardware

CRT – Cathode Ray Tube

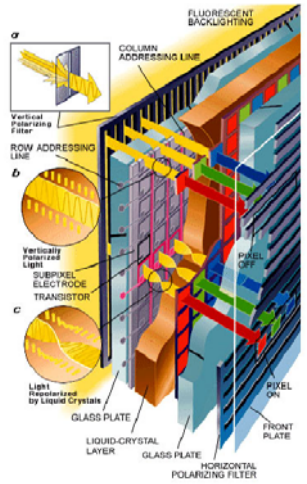


Components for 3D Graphics
Hardware
LCD – Liquid Crystal Display



Components for 3D Graphics
Hardware
LCD – Liquid Crystal Display

- Compact in depth
- Limited in size
  - size vs cost
  - getting cheaper every day
- Light is polarized
  - stereo cannot be generated using polarizing glasses



Components for 3D Graphics
Hardware
AMOLED – Active Matrix Organic Light Emitting Diode Display

- Since 2008 in smart phones
  - low power, low cost
  - Smartphones can now be used in HMDs (*see* Net Gear)
- Used increasingly in larger screens

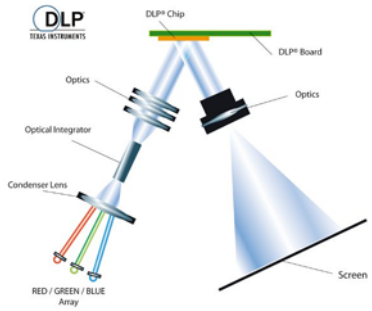
Components for 3D Graphics
Hardware
DLP – Digital Light Processing Projectors

- Used for back projected systems
  - Used in the Reading CAVE
  - Low cost (very few >£500)
  - Cost = resolution + Lumens

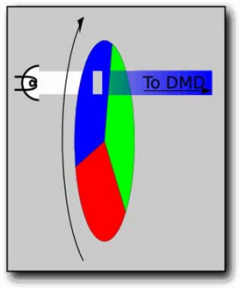
- (LCDs can also be used in projection systems)

Components for 3D Graphics
Hardware

DLP – Digital Light Processing Projectors

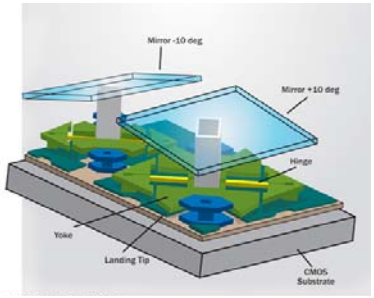


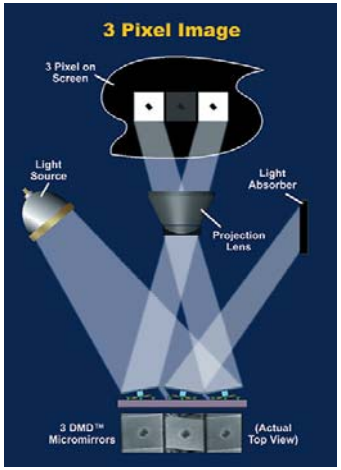
or



Components for 3D Graphics
Hardware

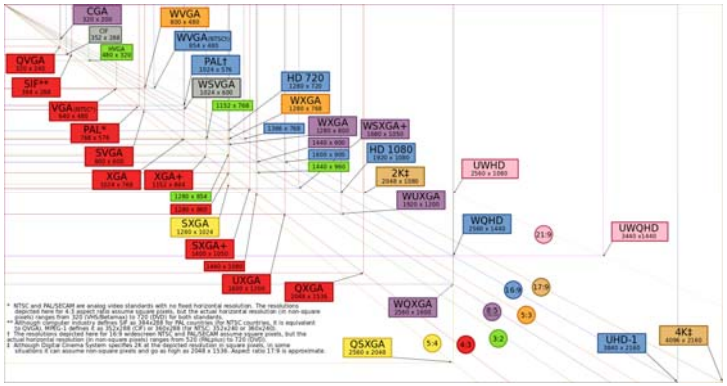
DLP – Digital Light Processing





Components for 3D Graphics
Hardware

Common Display Resolution Standards

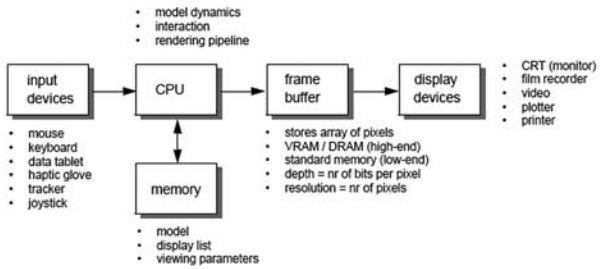


Components for 3D Graphics

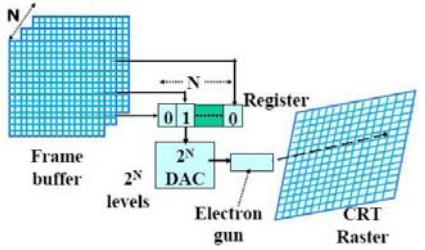
Outline

- **Displaying an Image**
  - Hardware
  - **Systems**
  - Colour Representation
- **3D Scene Representation**
  - Intro
  - Coordinate Systems
  - 3D Primitives

Components for 3D Graphics
Systems
The Frame Buffer

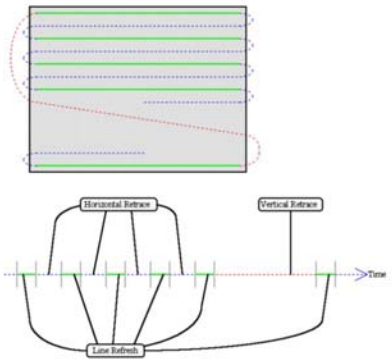


Components for 3D Graphics
Systems
The Frame Buffer

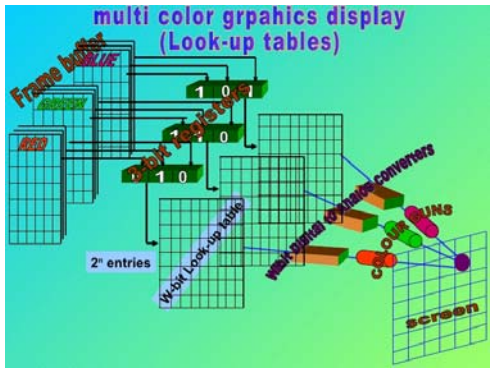


An N- bit plane gray level frame buffer

Components for 3D Graphics
Systems
Screen Refresh



Components for 3D Graphics
Systems
Colour Frame Buffer



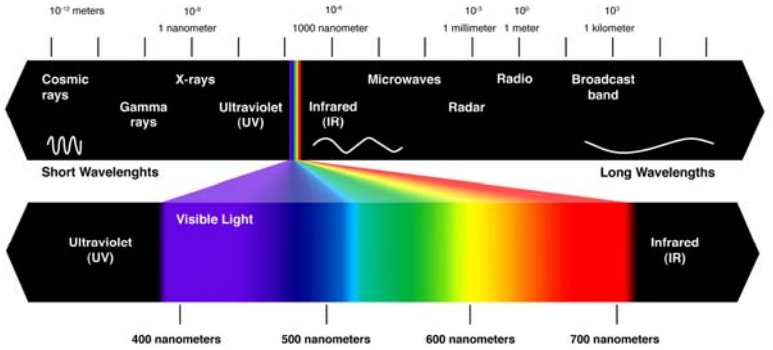
Components for 3D Graphics

Outline

- **Displaying an Image**
  - Hardware
  - Systems
  - **Colour Representation**
- **3D Scene Representation**
  - Intro
  - Coordinate Systems
  - 3D Primitives

Components for 3D Graphics
Colour Representation

The Colour Spectrum

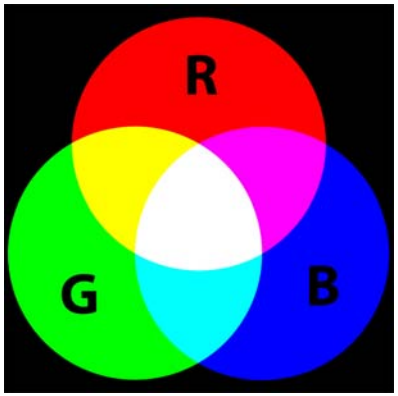


Components for 3D Graphics
Colour Representation

RGB – Red Green Blue

- **Based on Light**
- **Additive mixing**
- **Mainly for projecting**

R	G	B	Colour
0	0	0	Black
1	1	1	White
1	0	0	Red
0	1	0	Green
0	0	1	Blue
1	1	0	Yellow
0	1	1	Cyan
1	0	1	Magenta

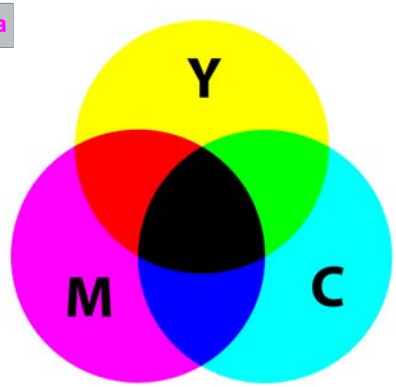


Components for 3D Graphics
Colour Representation

CYM – Cyan Yellow Magenta

- **Based on Pigment**
- **Subtractive mixing**
- **Mainly for printing**

C	M	Y	Colour
1	1	1	≈Black
0	0	0	White
0	1	1	Red
1	0	1	Green
1	1	0	Blue
0	0	1	Yellow
1	0	0	Cyan
0	1	0	Magenta



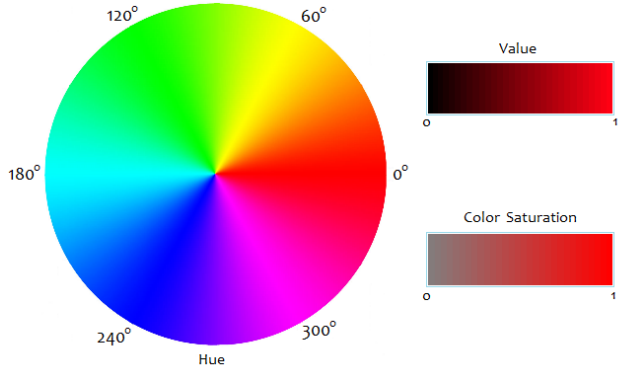
Components for 3D Graphics
Colour Representation

CYMK – Cyan Yellow Magenta Black

- K = "key"
- Based on Pigment
- Subtractive mixing
- Mainly used in printing
- Colours typically applied in the order CYMK

Components for 3D Graphics
Colour Representation

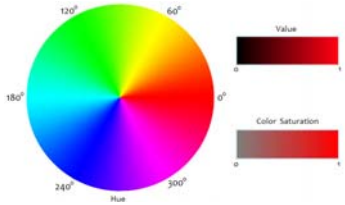
HSV – Hue Saturation Value

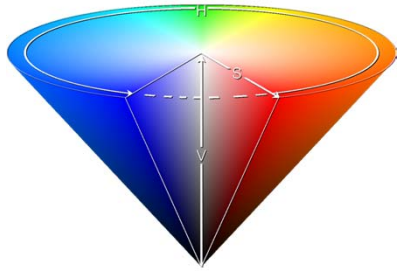


Components for 3D Graphics
Colour Representation

HSV – Hue Saturation Value

- Based on RGB
- Designed for computer graphics



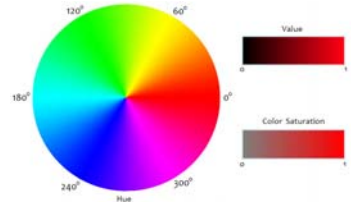


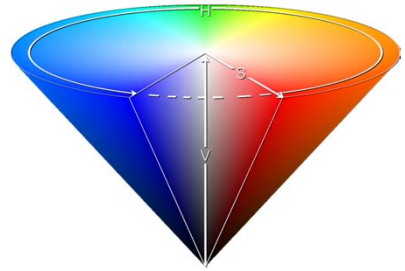
H	S	V	Colour
*	*	0	Black
*	0	1	White
0	1	1	Red
120	1	1	Green
240	1	1	Blue
*	0	0.5	Grey

Components for 3D Graphics
Colour Representation

HSV – Hue Saturation Value

- Based on RGB
- Designed for computer graphics





H	S	V	Colour
*	*	0	Black
*	0	1	White
0	1	1	Red
120	1	1	Green
240	1	1	Blue
*	0	0.5	Grey

SE3VR11 : Virtual Reality

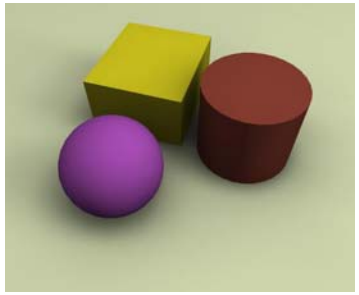
Professor Paul Sharkey p.m.sharkey@reading.ac.uk

Lecture 3 – 28/01/2016

3D Scene Representation
Intro

Representation

- Points
- Lines or line segments
- Polygons
- Surfaces
- Solids
- Voxels
- . . .



We will look at some of these in more detail shortly ...

Components for 3D Graphics

Outline

- **Displaying an Image**
  - Hardware
  - Systems
  - Colour Representation
- **3D Scene Representation**
  - Intro
  - **Coordinate Systems**
  - 3D Primitives

3D Scene Representation
Coordinate Systems

We need some sort of geometric base to form our computations, these come in the form of coordinate systems

- **Cartesian rectilinear system:**

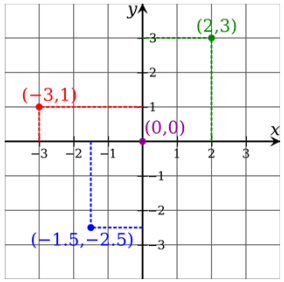
$$(x, y, z)$$

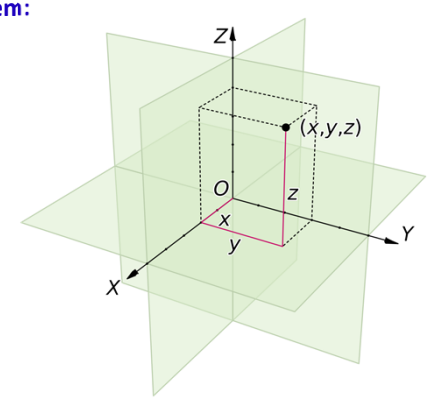
- **Spherical, Polar or Angular system:**

$$(r, \theta, \varphi)$$

3D Scene Representation
Coordinate Systems

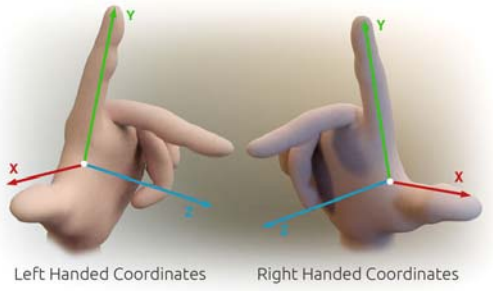
Cartesian rectilinear system:





3D Scene Representation
Coordinate Systems

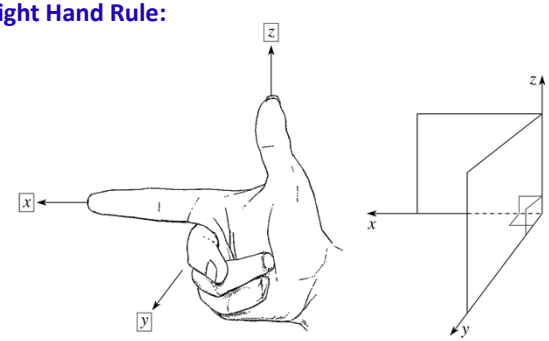
Left Hand Rule **vs** **Right Hand Rule**



clockwise rotation **vs** **counter clockwise rotation**

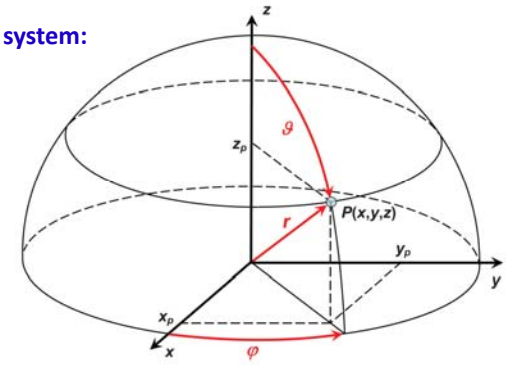
3D Scene Representation
Coordinate Systems

Right Hand Rule:



3D Scene Representation
Coordinate Systems

Spherical system:



Exercises: How to translate between the two systems?
How to translate between right/left hand systems?

Components for 3D Graphics

Outline

- **Displaying an Image**
  - Hardware
  - Systems
  - Colour Representation
- **3D Scene Representation**
  - Intro
  - Coordinate Systems
  - **3D Primitives**

3D Scene Representation
3D Primitives

Points

- **A point specifies a location**
- **Represented by 3 values**
- **The point has no volume (it is infinitely small)**
- **Can be expressed as**

$$\mathbf{p} = (x, y, z)$$

or

$$\mathbf{p} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

3D Scene Representation
3D Primitives

Vectors

- **Specifies a direction and a magnitude**
- **Represented by 3 values**
- **A point can be represented as a vector from (0, 0, 0)**
- **Commonly used with a unit length (normalised)**

$$\underline{\mathrm{V}} = \overline{\mathrm{V}} = (dx, dy, dz) = \langle dx, dy, dz \rangle$$

or

$$\underline{\mathrm{V}} = \overline{\mathrm{V}} = \begin{bmatrix} dx \\ dy \\ dz \end{bmatrix}$$

3D Scene Representation
3D Primitives

Vector Addition

- **Adding**

$$\overline{\mathrm{C}} = \overline{\mathrm{A}} + \overline{\mathrm{B}} = (a_x + b_x, a_y + b_y, a_z + b_z)$$

- **Subtracting**

$$\overline{\mathrm{C}} = \overline{\mathrm{A}} - \overline{\mathrm{B}} = \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} - \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} = \begin{bmatrix} a_x - b_x \\ a_y - b_y \\ a_z - b_z \end{bmatrix}$$

3D Scene Representation
3D Primitives

Vector Normalisation

- **Magnitude:** $\left\|\bar{V}\right\| = \sqrt{x^2 + y^2 + z^2}$
- **When** $\left\|\bar{V}\right\| = 1$ **the vector is normalised**
- **To normalise a vector:** $\hat{V} = \dfrac{\bar{V}}{\left\|\bar{V}\right\|}$
- $\hat{V}$ **is now of unit length**

3D Scene Representation
3D Primitives

Vector Dot Product (AKA **scalar** product)

- **With** $\bar{a} = \langle a_1, a_2, a_3 \rangle$ **and** $\bar{b} = \langle b_1, b_2, b_3 \rangle$
- **The dot product is given by**

$$\bar{a} \bullet \bar{b} = a_1 b_1 + a_2 b_2 + a_3 b_3$$

- **The result is a scalar value**

3D Scene Representation
3D Primitives

Angle between two Vectors

- **The dot product is also defined as**

$$\bar{a} \bullet \bar{b} = \left\|\bar{a}\right\| \cdot \left\|\bar{b}\right\| \cos\theta$$

- **Hence**

$$\cos\theta = \frac{\bar{a} \bullet \bar{b}}{\left\|\bar{a}\right\| \cdot \left\|\bar{b}\right\|}$$

- **The result** θ **can be found by taking the inverse** $\cos^{-1}(\ldots)$

3D Scene Representation
3D Primitives

Orthogonal Vectors

- **Two vectors are orthogonal (i.e. perpendicular) to each other when**

$$\bar{a} \bullet \bar{b} = 0$$

- **This is easily verified** $\cos\theta = \dfrac{\bar{a} \bullet \bar{b}}{\left\|\bar{a}\right\| \cdot \left\|\bar{b}\right\|} = 0$
- **Therefore** $\theta = \cos^{-1}\{0\} = 90°$

3D Scene Representation
3D Primitives

3D Line

- Given a point $\bar{\mathbf{p}}$ and a direction vector $\bar{\mathbf{V}}$
- Then $\bar{\mathbf{p}}_1$ is another point on the same line

$$\bar{\mathbf{p}}_1 = \bar{\mathbf{p}} + t\bar{\mathbf{V}}$$

where the parameter t is a scalar

$$(-\infty \le t \le \infty)$$

3D Scene Representation
3D Primitives

Rays

- Given a point $\bar{\mathbf{p}}$ and a direction vector $\bar{\mathbf{V}}$, and another point on the same line $\bar{\mathbf{p}}_1$

$$\bar{\mathbf{p}}_1 = \bar{\mathbf{p}} + t\bar{\mathbf{V}}$$

- Then, if the parameter t is restricted to the range

$$(0 \le t \le \infty)$$

- the point can be considered to a general point on a ray, originating at $\bar{\mathbf{p}}$ and shining in the direction $\bar{\mathbf{V}}$

3D Scene Representation
3D Primitives

3D Line Segment

- Given two points $\bar{\mathbf{p}}_1$ and $\bar{\mathbf{p}}_2$
- Then $\bar{\mathbf{p}}_3$ is another point on the same line between them

$$\bar{\mathbf{p}}_3 = \bar{\mathbf{p}}_1 + t(\bar{\mathbf{p}}_2 - \bar{\mathbf{p}}_1)$$

where $(0 \le t \le 1)$

Note that two points can be used to define a direction:

$$\bar{\mathbf{p}}_2 - \bar{\mathbf{p}}_1 = \bar{\mathbf{V}}$$

3D Scene Representation
3D Primitives

Matrices

2×2 matrix:
$$\mathbf{M} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

3×3 matrix:
$$\mathbf{M} = \begin{bmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \end{bmatrix}$$

4×4 matrix:
$$\mathbf{M} = \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix}$$

(note different styles in notation can be used)

3D Scene Representation
3D Primitives

2×2 Matrix Determinant

$$|\mathbf{M}| = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$$

This will enable us to calculate the vector cross product between two vectors

3D Scene Representation
3D Primitives

Vector Cross Product (AKA **vector** product)

- **The cross product between two vectors is defined as**

$$\bar{\mathbf{a}} \times \bar{\mathbf{b}} = \|\bar{\mathbf{a}}\| \cdot \|\bar{\mathbf{b}}\| \hat{\mathbf{n}} \sin\theta$$

- **In words, it is …**
  - The magnitude of vector $\|\bar{\mathbf{a}}\|$
  - multiplied by the magnitude of vector $\|\bar{\mathbf{b}}\|$
  - multiplied by the **sine** of the angle between them $\sin\theta$
  - multiplied by the **normal** to both vectors $\hat{\mathbf{n}}$

Importantly, the cross product can be used to find the normal

3D Scene Representation
3D Primitives

Vector Cross Product $\bar{\mathbf{a}} \times \bar{\mathbf{b}} = \|\bar{\mathbf{a}}\| \cdot \|\bar{\mathbf{b}}\| \hat{\mathbf{n}} \sin\theta$

- **The cross product results in a vector**
- **The cross product can also be found using determinants as**

$$\bar{\mathbf{a}} \times \bar{\mathbf{b}} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} \quad c_1 = \begin{vmatrix} a_2 & b_2 \\ a_3 & b_3 \end{vmatrix} \quad c_2 = -\begin{vmatrix} a_1 & b_1 \\ a_3 & b_3 \end{vmatrix} \quad c_3 = \begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}$$

- **Note the negative sign in the middle term!**
- **Note that** $\bar{\mathbf{a}} \times \bar{\mathbf{b}} = -\bar{\mathbf{b}} \times \bar{\mathbf{a}}$

3D Scene Representation
3D Primitives

Vector Cross Product – Cover up method

- **One way to remember how to calculate cross products is using the cover up method**

$$\bar{\mathbf{a}} \times \bar{\mathbf{b}} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \times \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

- **To determine any element, cover up that row and calculate the resulting determinant (remembering to apply a negative in row 2)**

$$\bar{\mathbf{a}} \times \bar{\mathbf{b}} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \times \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

3D Scene Representation
3D Primitives

Vector Cross Product – Cover up method

- One way to remember how to calculate cross products is using the cover up method

$$\bar{\mathbf{a}} \times \bar{\mathbf{b}} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \times \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

- To determine any element, cover up that row and calculate the resulting determinant (remembering to apply a negative in row 2)

$$\bar{\mathbf{a}} \times \bar{\mathbf{b}} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} a_2 \\ a_3 \end{bmatrix} \times \begin{bmatrix} b_2 \\ b_3 \end{bmatrix} \quad \therefore \quad c_1 = \begin{vmatrix} a_2 & b_2 \\ a_3 & b_3 \end{vmatrix}$$

3D Scene Representation
3D Primitives

Normal of two vectors

The cross product can be used to find the normal to both vectors

A normal is found by taking the cross product

$$\mathbf{n} = \bar{\mathbf{a}} \times \bar{\mathbf{b}}$$

A unit direction normal is found by dividing by the magnitude of the resultant cross product

$$\hat{\mathbf{n}} = \frac{\bar{\mathbf{a}} \times \bar{\mathbf{b}}}{\left\| \bar{\mathbf{a}} \times \bar{\mathbf{b}} \right\|}$$

3D Scene Representation
3D Primitives

3D Plane

A plane can be defined in 2 ways:

- Any known point and where the normal vector to the plane is known
- Any three points that are not co-linear
  - if the three points were co-linear then the best they can be used for is to define an infinite number of planes about a common axis

3D Scene Representation
3D Primitives

3D Plane – Defined using 1 point and the normal

Given a point $\mathbf{p}_0$ and a normal vector $\hat{\mathbf{n}}$ then the general point $\mathbf{p} = (x, y, z)$ will be on the plane if the following constraint is true

$$(\mathbf{p} - \mathbf{p}_0) \bullet \hat{\mathbf{n}} = 0$$

Proof: If the dot product is equal to zero, then $\cos\theta = 0$ and the angle must be $90°$

Therefore the vector $(\mathbf{p} - \mathbf{p}_0)$ must lie in the plane and so the point $\mathbf{p}$ must be in the plane.

3D Scene Representation
3D Primitives

3D Plane – Defined using 3 non co-linear points

Given a points $\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2$ the general point $\mathbf{p} = (x, y, z)$

will be on the plane if the following constraint is true

$$(\mathbf{p} - \mathbf{p}_0) \bullet \frac{(\mathbf{p}_2 - \mathbf{p}_0) \times (\mathbf{p}_1 - \mathbf{p}_0)}{\left\| (\mathbf{p}_2 - \mathbf{p}_0) \times (\mathbf{p}_1 - \mathbf{p}_0) \right\|} = 0$$

Proof: The quotient term is in fact the normal to the plane

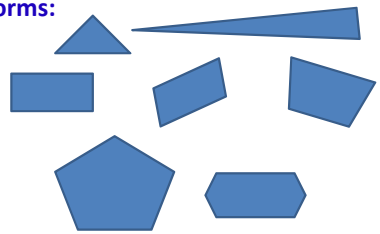
Each bracketed term is a vector in the plane and the cross product, which is then normalised, defines the normal $\hat{\mathbf{n}}$

(*The rest of the proof is the same as before*)

3D Scene Representation
3D Primitives

Bounded Plane or Polygon

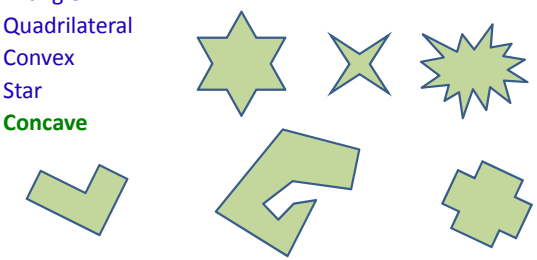
- **If a plane is bounded we get a polygon and they come in many forms:**
  - Triangle
  - Quadrilateral
  - **Convex**



3D Scene Representation
3D Primitives

Bounded Plane or Polygon

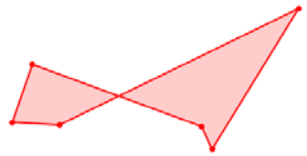
- **If a plane is bounded we get a polygon and they come in many forms:**
  - Triangle
  - Quadrilateral
  - Convex
  - Star
  - **Concave**



3D Scene Representation
3D Primitives

Bounded Plane or Polygon

- **If a plane is bounded we get a polygon and they come in many forms:**
  - Triangle
  - Quadrilateral
  - Convex
  - Star
  - Concave
  - **Self-intersecting**



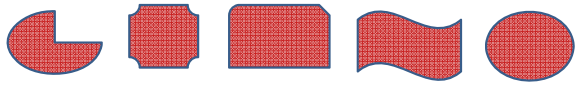
- **If a hole is required, then more than one polygon is required**

3D Scene Representation
3D Primitives

Polygons or not?

- Polygons

- Not Polygons



3D Scene Representation
3D Primitives

Matrix Multiplication

2×2 Example

$$\mathbf{A}=\begin{bmatrix}3 & 5\\ -1 & 2\end{bmatrix} \qquad \mathbf{B}=\begin{bmatrix}-4 & 7\\ 6 & 1\end{bmatrix}$$

Then

$$\mathbf{AB}=\begin{bmatrix}3 & 5\\ -1 & 2\end{bmatrix}\begin{bmatrix}-4 & 7\\ 6 & 1\end{bmatrix}$$

$$=\begin{bmatrix}(3)(-4)+(5)(6) & (3)(7)+(5)(1)\\ (-1)(-4)+(2)(6) & (-1)(7)+(2)(1)\end{bmatrix}$$

$$=\begin{bmatrix}+18 & +26\\ +16 & -5\end{bmatrix}$$

Be sure you are familiar with the process

3D Scene Representation
3D Primitives

Matrix Multiplication

3×3 Example

$$\mathbf{A}=\begin{bmatrix}2 & 3 & -4\\ 4 & -5 & 6\\ 8 & -7 & 9\end{bmatrix} \qquad \mathbf{B}=\begin{bmatrix}10 & -3 & -4\\ 2 & 0 & 1\\ 12 & 12 & 20\end{bmatrix}$$

Show that $\mathbf{C}=\mathbf{AB}=\begin{bmatrix}-22 & -54 & -85\\ 102 & 60 & 99\\ 174 & 84 & 141\end{bmatrix}$

3D Scene Representation
3D Primitives

Matrix Multiplication

4×4 Example

$$\mathbf{A}=\begin{bmatrix}1 & 2 & 3 & 4\\ 5 & 6 & 7 & 8\\ 9 & 8 & 7 & 6\\ 5 & 4 & 3 & 2\end{bmatrix} \qquad \mathbf{B}=\begin{bmatrix}1 & -2 & 3 & -4\\ -5 & 6 & 7 & -8\\ 10 & 12 & 14 & 16\\ 0 & 10 & 20 & 18\end{bmatrix}$$

Show that $\mathbf{C}=\mathbf{AB}=\begin{bmatrix}21 & 86 & 139 & 100\\ 45 & 190 & 315 & 188\\ 39 & 174 & 301 & 120\\ 15 & 70 & 125 & 32\end{bmatrix}$

SE3VR11 : Virtual Reality

Professor Paul Sharkey p.m.sharkey@reading.ac.uk

Lecture 4 – 04/02/2016

3D Scene Representation
3D Primitives

Matrix Transforms

All transformations appropriate for computer graphics can be represented with a single 4×4 matrix

$$\mathbf{M} = \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix}$$

If a transformation is represented by the matrix $\mathbf{T}$, the point $\mathbf{p}$ can be transformed to the new point $\mathbf{p}'$ by matrix multiplication:

$$\mathbf{p}' = \mathbf{T}\mathbf{p}$$

3D Scene Representation
3D Primitives

Matrix Transforms

Example: If $\mathbf{T}$ and $\mathbf{p}$ are given by $\mathbf{T} = \begin{bmatrix} 1 & 3 & -2 & 5 \\ 4 & 8 & 4 & 4 \\ -1 & 0 & 2 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ $\mathbf{p} = \begin{bmatrix} 1 \\ 5 \\ 2 \end{bmatrix}$

Find $\mathbf{p}'$

$$\mathbf{p}' = \mathbf{T} \times \mathbf{p} = \begin{bmatrix} 1 & 3 & -2 & 5 \\ 4 & 8 & 4 & 4 \\ -1 & 0 & 2 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 5 \\ 2 \end{bmatrix}$$

X

But, we have a problem …

The number of columns in the matrix DOES NOT equal the number of rows in the vector. The dimensions are not right!

3D Scene Representation
3D Primitives

Matrix Transforms

The solution is to augment the point into an homogenous form

$$\mathbf{p} = \begin{bmatrix} 1 \\ 5 \\ 2 \end{bmatrix} \Rightarrow \mathbf{p}_h = \begin{bmatrix} 1 \\ 5 \\ 2 \\ 1 \end{bmatrix}$$

Now

$$\mathbf{p}'_h = \mathbf{T}\mathbf{p}_h = \begin{bmatrix} 1 & 3 & -2 & 5 \\ 4 & 8 & 4 & 4 \\ -1 & 0 & 2 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 5 \\ 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 17 \\ 56 \\ 4 \\ 1 \end{bmatrix} \Rightarrow \mathbf{p}' = \begin{bmatrix} 17 \\ 56 \\ 4 \end{bmatrix}$$

where the transformed point is found by conversion back from the homogenous form

3D Scene Representation
3D Primitives

$$\mathbf{p}'_h = \mathbf{T}\mathbf{p}_h = \begin{bmatrix} 1 & 3 & -2 & 5 \\ 4 & 8 & 4 & 4 \\ -1 & 0 & 2 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 5 \\ 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 17 \\ 56 \\ 4 \\ 1 \end{bmatrix}$$

Matrix Transforms

How do such transformation matrices come about?

Transformation matrices can be used to define:

- Translations
- Rotations
- Scaling
- Skewing *or* Shearing
- Reflections
- Perspective transforms
- Any combination of the above

Combinations are realised by multiplication of matrices

3D Scene Representation
3D Primitives

Matrix Transforms – Translations

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Applied as (eg.)

$$\mathbf{p} = \begin{bmatrix} 1 \\ 5 \\ 2 \end{bmatrix} \Rightarrow \mathbf{p}'_{(h)} = \begin{bmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 5 \\ 2 \\ 1 \end{bmatrix} \Rightarrow \mathbf{p}' = \begin{bmatrix} 1+dx \\ 5+dy \\ 2+dz \end{bmatrix}$$

3D Scene Representation
3D Primitives

Matrix Transforms – Rotations about x

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Applied as (eg.)

$$\mathbf{p} = \begin{bmatrix} 1 \\ 5 \\ 2 \end{bmatrix} \Rightarrow \mathbf{p}'_{(h)} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos 60 & -\sin 60 & 0 \\ 0 & \sin 60 & \cos 60 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 5 \\ 2 \\ 1 \end{bmatrix} \Rightarrow \mathbf{p}' = \begin{bmatrix} 1 \\ 0.7680 \\ 5.3301 \end{bmatrix}$$

3D Scene Representation
3D Primitives

Matrix Transforms – Rotations about y

$$\mathbf{T} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Applied as (eg.)

$$\mathbf{p} = \begin{bmatrix} 1 \\ 5 \\ 2 \end{bmatrix} \Rightarrow \mathbf{p}'_{(h)} = \begin{bmatrix} \cos 60 & 0 & \sin 60 & 0 \\ 0 & 1 & 0 & 0 \\ -\sin 60 & 0 & \cos 60 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 5 \\ 2 \\ 1 \end{bmatrix} \Rightarrow \mathbf{p}' = \begin{bmatrix} 2.2321 \\ 5 \\ 0.1340 \end{bmatrix}$$

3D Scene Representation
3D Primitives

Matrix Transforms – Rotations about z

$$\mathbf{T} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Applied as (eg.)

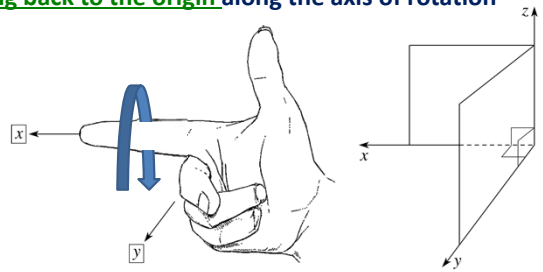
$$\mathbf{p} = \begin{bmatrix} 1 \\ 5 \\ 2 \end{bmatrix} \Rightarrow \mathbf{p}'_{(h)} = \begin{bmatrix} \cos 60 & -\sin 60 & 0 & 0 \\ \sin 60 & \cos 60 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 5 \\ 2 \\ 1 \end{bmatrix} \Rightarrow \mathbf{p}' = \begin{bmatrix} -3.801 \\ 3.3660 \\ 2 \end{bmatrix}$$

3D Scene Representation
3D Primitives

Matrix Transforms – Rotations general

Rotations are always about the main coordinate axes

Rotations are positive in an anti-clockwise direction when looking back to the origin along the axis of rotation



3D Scene Representation
3D Primitives

Matrix Transforms – Scaling

$$\mathbf{T} = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Applied as (eg.)

$$\mathbf{p} = \begin{bmatrix} 1 \\ 5 \\ 2 \end{bmatrix} \Rightarrow \mathbf{p}'_{(h)} = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 5 \\ 2 \\ 1 \end{bmatrix} \Rightarrow \mathbf{p}' = \begin{bmatrix} S_x \\ 5S_y \\ 2S_z \end{bmatrix}$$

3D Scene Representation
3D Primitives

Matrix Transforms – Shear

$$\mathbf{T} = \begin{bmatrix} 1 & sh_x^y & sh_x^z & 0 \\ sh_y^x & 1 & sh_y^z & 0 \\ sh_z^x & sh_z^y & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

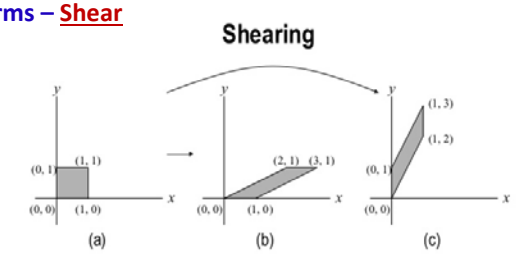
Applied as (eg.)

$$\mathbf{p} = \begin{bmatrix} 1 \\ 5 \\ 2 \end{bmatrix} \Rightarrow \mathbf{p}'_{(h)} = \begin{bmatrix} 1 & sh_x^y & sh_x^z & 0 \\ sh_y^x & 1 & sh_y^z & 0 \\ sh_z^x & sh_z^y & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 5 \\ 2 \\ 1 \end{bmatrix} \Rightarrow \mathbf{p}' = \begin{bmatrix} 1+5sh_x^y+2sh_x^z \\ 5+sh_y^x+2sh_y^z \\ 2+sh_z^x+5sh_z^y \end{bmatrix}$$

3D Scene Representation
3D Primitives

Matrix Transforms – Shear

Examples



Here $sh_x^y = 2$

and $sh_y^x = 2$

$$\begin{bmatrix} 1 & SH_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 0 \\ SH_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The matrices shown here are the equivalent of the top left 3×3 block

3D Scene Representation
3D Primitives

Matrix Transforms – Shear

Exercise: Show that applying the following shear matrix

$$\mathbf{T} = \begin{bmatrix} 1 & sh_x^y & 0 & 0 \\ sh_y^x & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

results in a very different outcome to applying one shear after another

$$\mathbf{T} = \mathbf{T}_1\mathbf{T}_2 = \begin{bmatrix} 1 & sh_x^y & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ sh_y^x & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3D Scene Representation
3D Primitives

Matrix Transforms – Shear

$$\mathbf{T} = \begin{bmatrix} 1 & sh_x^y & 0 & 0 \\ sh_y^x & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Mathematically:

$$\mathbf{T} = \mathbf{T}_1\mathbf{T}_2 = \begin{bmatrix} 1 & sh_x^y & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ sh_y^x & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1+\left(sh_x^y\right)\left(sh_y^x\right) & sh_x^y & 0 & 0 \\ sh_y^x & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

What is the graphical result for the cube?

3D Scene Representation
3D Primitives

Matrix Transforms – Shear

$$\mathbf{T} = \begin{bmatrix} 1+\left(sh_x^y\right)\left(sh_y^x\right) & sh_x^y & 0 & 0 \\ sh_y^x & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

What is the graphical result for the cube?

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

Show that the points are located at:

$$\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 5 \\ 2 \\ 0 \end{bmatrix}, \begin{bmatrix} 7 \\ 3 \\ 0 \end{bmatrix}$$

3D Scene Representation
3D Primitives

Matrix Transforms – Shear

$$\mathbf{T} = \begin{bmatrix} 1 & sh_x^y & 0 & 0 \\ sh_y^x & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

What is the graphical result for the cube?

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

Show that the points are located at:

$$\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix}, \begin{bmatrix} 3 \\ 3 \\ 0 \end{bmatrix}$$

3D Scene Representation
3D Primitives

Matrix Transforms – Reflections in x-axis

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

Applied as (eg.)

$$\mathbf{p} = \begin{bmatrix} 1 \\ 5 \\ 2 \end{bmatrix} \Rightarrow \mathbf{p}'_{(h)} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 5 \\ 2 \\ 1 \end{bmatrix} \Rightarrow \mathbf{p}' = \begin{bmatrix} -1 \\ 5 \\ 2 \end{bmatrix}$$

3D Scene Representation
3D Primitives

Matrix Transforms – Reflections in y-axis

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

Applied as (eg.)

$$\mathbf{p} = \begin{bmatrix} 1 \\ 5 \\ 2 \end{bmatrix} \Rightarrow \mathbf{p}'_{(h)} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 5 \\ 2 \\ 1 \end{bmatrix} \Rightarrow \mathbf{p}' = \begin{bmatrix} 1 \\ -5 \\ 2 \end{bmatrix}$$

3D Scene Representation
3D Primitives

Matrix Transforms – Reflections in z-axis

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

Applied as (eg.)

$$\mathbf{p} = \begin{bmatrix} 1 \\ 5 \\ 2 \end{bmatrix} \Rightarrow \mathbf{p}'_{(h)} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 5 \\ 2 \\ 1 \end{bmatrix} \Rightarrow \mathbf{p}' = \begin{bmatrix} 1 \\ 5 \\ -2 \end{bmatrix}$$

3D Scene Representation
3D Primitives

Matrix Transforms – Combinations

For more complex transformations it is possible to combine multiple transformation matrices in a single matrix

Example: Move point $\mathbf{p}$ 10 units along the y-axis, then rotate it 20 degrees around the z-axis and then move it –15 units along the x-axis.

3D Scene Representation
3D Primitives

Matrix Transforms – Combinations

Example: Move point $\mathbf{p}$ 10 units along the y-axis, rotate it 20 degrees around the z-axis and then move it –15 units along the x-axis

$$\mathbf{T}_{trans,y} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 10 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{T}_{rot,z} = \begin{bmatrix} \cos 20 & -\sin 20 & 0 & 0 \\ \sin 20 & \cos 20 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{T}_{trans,x} = \begin{bmatrix} 1 & 0 & 0 & -15 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 0.4081 & -0.9129 & 0 & 0 \\ 0.9129 & 0.4081 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3D Scene Representation
3D Primitives

Matrix Transforms – Combinations

Example: Move point $\mathbf{p}$ 10 units along the y-axis, rotate it 20 degrees around the z-axis and then move it –15 units along the x-axis …

$$\mathbf{p}' = \mathbf{T}_{trans,y}\mathbf{p} \qquad (\mathbf{p}' = \mathbf{T}_1\mathbf{p})$$
$$\mathbf{p}'' = \mathbf{T}_{rot,z}\mathbf{p}' \qquad (\mathbf{p}'' = \mathbf{T}_2\mathbf{p}')$$
$$\mathbf{p}''' = \mathbf{T}_{trans,x}\mathbf{p}'' \qquad (\mathbf{p}''' = \mathbf{T}_3\mathbf{p}'')$$

This is combined as

$$\mathbf{p}''' = \mathbf{T}_{trans,x}\mathbf{T}_{rot,z}\mathbf{T}_{trans,y}\mathbf{p} \qquad (\mathbf{p}''' = \mathbf{T}_3\mathbf{T}_2\mathbf{T}_1\mathbf{p})$$

Note the order of the transforms is from right to left
This is most important

3D Scene Representation
3D Primitives

Matrix Transforms – Combinations

The order of matrix multiplication is important!

$$\mathbf{T} \times \mathbf{S} \neq \mathbf{S} \times \mathbf{T}$$

3D Scene Representation
3D Primitives

Matrix Transforms – Combinations

$$\mathbf{p}''' = \mathbf{T}_{trans,x}\mathbf{T}_{rot,z}\mathbf{T}_{trans,y}\mathbf{p}$$

Example:

$$\mathbf{p}''' = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 10 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0.4081 & -0.9129 & 0 & 0 \\ 0.9129 & 0.4081 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -15 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{p}$$

$$\mathbf{p}''' = \begin{bmatrix} 0.4081 & -0.9129 & 0 & -6.1215 \\ 0.9129 & 0.4081 & 0 & -3.6935 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{p} \qquad \mathbf{p} = \begin{bmatrix} 1 \\ 5 \\ 2 \end{bmatrix} \Rightarrow \mathbf{p}' = \begin{bmatrix} -10.2779 \\ -0.7401 \\ 2 \end{bmatrix}$$

(the z component is unaffected in this example)

3D Scene Representation
3D Primitives

Transforming Objects

In the previous example, the transformations were applied to a single point

How is the same transform applied to a complex object?

Consider a cuboid, defined by eight coherent points

$$\mathbf{obj}_{cuboid} = \begin{bmatrix} 2 & 8 & 8 & 2 & 2 & 8 & 8 & 2 \\ 3 & 3 & 3 & 3 & 5 & 5 & 5 & 5 \\ 4 & 4 & -3 & -3 & 4 & 4 & -3 & -3 \end{bmatrix}$$

(This cuboid is aligned with the major axes – sketch it out!)

3D Scene Representation
3D Primitives

Transforming Objects

Applying the combined transform requires augmented the cuboid description to include a fourth row of '1's

$$\mathbf{obj}_{cuboid(h)} = \begin{bmatrix} 2 & 8 & 8 & 2 & 2 & 8 & 8 & 2 \\ 3 & 3 & 3 & 3 & 5 & 5 & 5 & 5 \\ 4 & 4 & -3 & -3 & 4 & 4 & -3 & -3 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

The transformed object is

$$\mathbf{obj}'_{cuboid(h)} = \begin{bmatrix} 0.4081 & -0.9129 & 0 & -6.1215 \\ 0.9129 & 0.4081 & 0 & -3.6935 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 8 & 8 & 2 & 2 & 8 & 8 & 2 \\ 3 & 3 & 3 & 3 & 5 & 5 & 5 & 5 \\ 4 & 4 & -3 & -3 & 4 & 4 & -3 & -3 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

3D Scene Representation
3D Primitives

Transforming Objects

Result:

$$\mathbf{obj}'_{cuboid} = \begin{bmatrix} -8.044 & -5.595 & -5.595 & -8.044 & -9.870 & -7.421 & -7.421 & -9.870 \\ -0.643 & 4.834 & 4.834 & -0.643 & 0.173 & 5.650 & 5.650 & 0.173 \\ 4 & 4 & -3 & -3 & 4 & 4 & -3 & -3 \end{bmatrix}$$

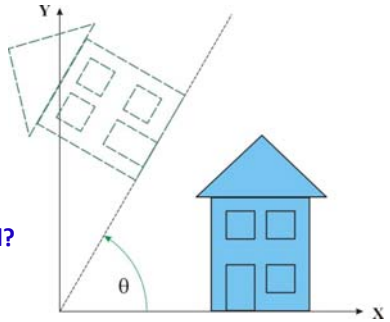
- The ordering of the points is the same as originally defined
- It is clear from the points that there is still a degree of symmetry with the main coordinate axes
- This is expected as the object was only rotated by one primary rotation
- More complex shapes may have hundreds or thousands of vertices

3D Scene Representation
3D Primitives

Rotating Objects – rotate the following object about z by 60°

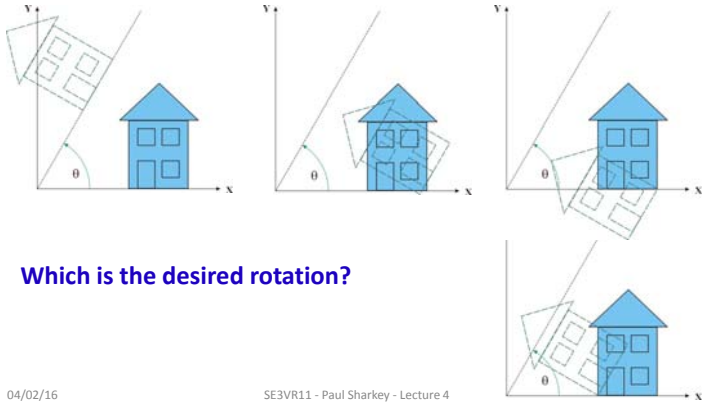
This is the result of applying the $Rot_{z,60}$ transform.

Was this what was wanted?



3D Scene Representation
3D Primitives

Rotating Objects – rotate the following object about z by 45°



Which is the desired rotation?

3D Scene Representation
3D Primitives

Rotating Objects

Rotations need to be defined not only by the axis of rotation but also by the point on the object that the rotation axis goes through.

If a point is identified in the object through which the axis of desired rotation is defined, the rotation can be achieved by

- translating this point to the origin of the coordinate system
- affecting the rotation
- translating back to its original location

$$\mathbf{obj}_{rotated} = \mathbf{T}_{Tx,back}\mathbf{T}_{Rot,\theta}\mathbf{T}_{Tx,to\ origin}\mathbf{obj}_{original}$$

3D Scene Representation
3D Primitives

Rotating Objects

- translating this point to the origin of the coordinate system
- affecting the rotation
- translating back to its original location

$$\mathbf{obj}_{rotated} = \mathbf{T}_{Tx,back}\mathbf{T}_{Rot,\theta}\mathbf{T}_{Tx,to\ origin}\mathbf{obj}_{original}$$

Note the order of transformation

3D Scene Representation
3D Primitives

What about Arbitrary Rotations?

A more general rotation can be achieved by defining

- A point $\mathbf{p}_1$ relative to the object (or within the object)
- A normalised direction vector, $\mathbf{k}$, defined by a second point $\mathbf{p}_2$ relative to point $\mathbf{p}_1$ and
- The angle θ about this axis to rotate

The final rotation is

$$\mathbf{obj}_{rotated} = \mathbf{T}_{Tx,+\mathbf{p}_1}\mathbf{T}_{Rot,\mathbf{k},\theta}\mathbf{T}_{Tx,-\mathbf{p}_1}\mathbf{obj}_{original}$$

where the rotation is defined on the following slides

3D Scene Representation
3D Primitives

$\mathbf{T}_{Rot,\mathbf{k},\theta}$

Equivalent Axis Rotations

An equivalent axes rotation is achieved by aligning the equivalent axis with one of the principal coordinate axes (e.g. the z axis), rotating about that axis, and then putting everything back in its original orientation

(Similar to translating to the origin and back again)

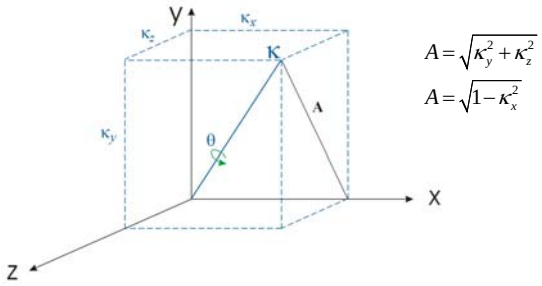
This initial alignment requires two rotations and hence two reverse rotations plus the required rotation in between

This gives FIVE rotations in total

3D Scene Representation
3D Primitives

$\mathbf{T}_{Rot,\mathbf{k},\theta}$

Equivalent Axis Rotations

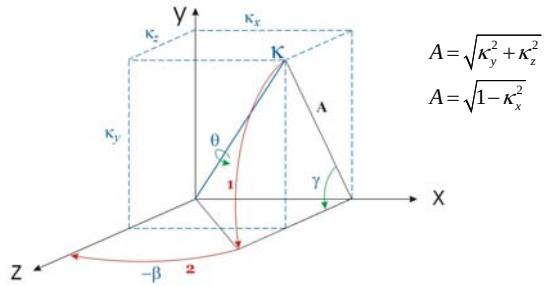


$$A = \sqrt{\kappa_y^2 + \kappa_z^2}$$
$$A = \sqrt{1 - \kappa_x^2}$$

3D Scene Representation
3D Primitives

$\mathbf{T}_{Rot,\mathbf{k},\theta}$

Equivalent Axis Rotations

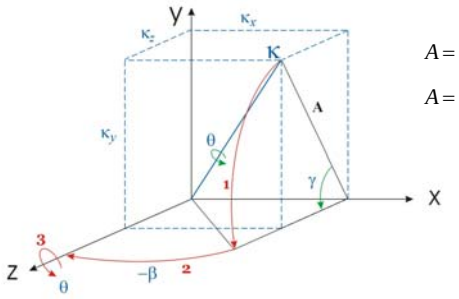


$$A = \sqrt{\kappa_y^2 + \kappa_z^2}$$
$$A = \sqrt{1 - \kappa_x^2}$$

3D Scene Representation
3D Primitives

Equivalent Axis Rotations

$$\mathbf{T}_{Rot,\mathbf{k},\theta}$$

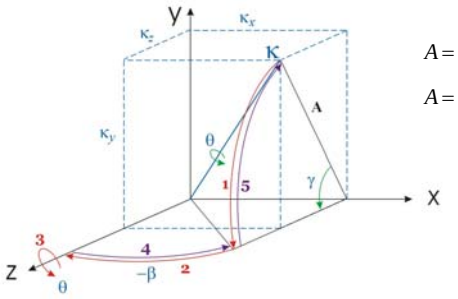


$$A = \sqrt{\kappa_y^2 + \kappa_z^2}$$
$$A = \sqrt{1 - \kappa_x^2}$$

3D Scene Representation
3D Primitives

Equivalent Axis Rotations

$$\mathbf{T}_{Rot,\mathbf{k},\theta}$$



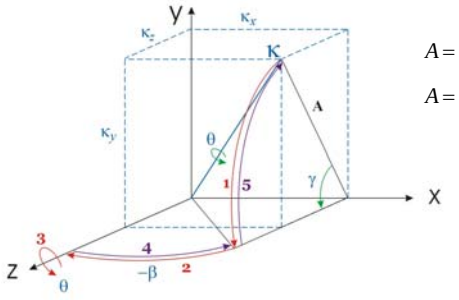
$$A = \sqrt{\kappa_y^2 + \kappa_z^2}$$
$$A = \sqrt{1 - \kappa_x^2}$$

$$\mathbf{T}_{Rot,\mathbf{k},\theta} = \mathbf{T}_5\mathbf{T}_4\mathbf{T}_3\mathbf{T}_2\mathbf{T}_1$$

3D Scene Representation
3D Primitives

Equivalent Axis Rotations

$$\mathbf{T}_{Rot,\mathbf{k},\theta}$$



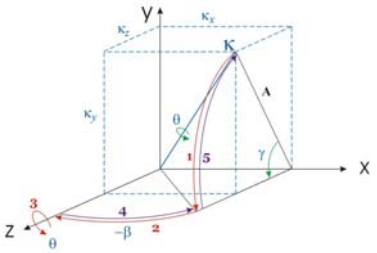
$$A = \sqrt{\kappa_y^2 + \kappa_z^2}$$
$$A = \sqrt{1 - \kappa_x^2}$$

$$\mathbf{T}_{Rot,\mathbf{k},\theta} = \mathbf{T}_{Rot,\mathbf{x},+\gamma}\mathbf{T}_{Rot,\mathbf{y},-\beta}\mathbf{T}_{Rot,\mathbf{z},\theta}\mathbf{T}_{Rot,\mathbf{y},+\beta}\mathbf{T}_{Rot,\mathbf{x},-\gamma}$$

3D Scene Representation
3D Primitives

Equivalent Axis Rotations

$$\mathbf{T}_{Rot,\mathbf{k},\theta}$$



$$A = \sqrt{\kappa_y^2 + \kappa_z^2}$$
$$A = \sqrt{1 - \kappa_x^2}$$

$$\sin\beta = \kappa_x$$
$$\cos\beta = A$$

$$A\cos\gamma = \kappa_z \qquad \cos\gamma = \frac{\kappa_z}{A}$$

$$A\sin\gamma = \kappa_y \qquad \sin\gamma = \frac{\kappa_y}{A}$$

$$\mathbf{T}_{Rot,\mathbf{k},\theta} = \mathbf{T}_{Rot,\mathbf{x},+\gamma}\mathbf{T}_{Rot,\mathbf{y},-\beta}\mathbf{T}_{Rot,\mathbf{z},\theta}\mathbf{T}_{Rot,\mathbf{y},+\beta}\mathbf{T}_{Rot,\mathbf{x},-\gamma}$$

3D Scene Representation
3D Primitives

$\mathbf{T}_{Rot,\mathbf{k},\theta}$

Equivalent Axis Rotations

An equivalent axes rotation transform is then found as

lots

and lots

and lots of algebra later …

3D Scene Representation
3D Primitives

$\mathbf{T}_{Rot,\mathbf{k},\theta}$

Equivalent Axis Rotations

An equivalent axes rotation transform is then found as

$$\mathbf{T}_{Rot,\mathbf{k},\theta} = \begin{bmatrix} xxv+c & xyv-zs & xzv+ys & 0 \\ yxv+zs & yyv+c & yzv-xs & 0 \\ zxv-ys & zyv+xs & zzv+c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where shorthand is used, with

$$s = \sin(\theta) \qquad c = \cos(\theta) \qquad v = 1 - c$$

and

$$\mathbf{k} = [x, y, z] \qquad \|\mathbf{k}\| = 1$$

SE3VR11 : Virtual Reality

Professor Paul Sharkey p.m.sharkey@reading.ac.uk

Lecture 5 – 25/02/2016

3D Scene Representation
3D Primitives

$\mathbf{T}_{Rot,\mathbf{k},\theta}$

Equivalent Axis Rotations

An equivalent axes rotation is achieved by aligning the equivalent axis with one of the principal coordinate axes (e.g. the z axis), rotating about that axis, and then putting everything back in its original orientation

(Similar to translating to the origin and back again)

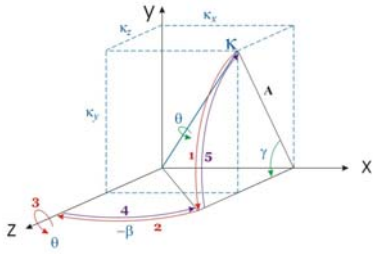
This initial alignment requires two rotations and hence two reverse rotations plus the required rotation in between

This gives FIVE rotations in total

3D Scene Representation
3D Primitives

$\mathbf{T}_{Rot,\mathbf{k},\theta}$

Equivalent Axis Rotations



$$A = \sqrt{\kappa_y^2 + \kappa_z^2} \qquad \sin\beta = \kappa_x$$

$$A = \sqrt{1 - \kappa_x^2} \qquad \cos\beta = A$$

$$A\cos\gamma = \kappa_z \qquad \cos\gamma = \frac{\kappa_z}{A}$$

$$A\sin\gamma = \kappa_y \qquad \sin\gamma = \frac{\kappa_y}{A}$$

$$\mathbf{T}_{Rot,\mathbf{k},\theta} = \mathbf{T}_{Rot,\mathbf{x},+\gamma}\mathbf{T}_{Rot,\mathbf{y},-\beta}\mathbf{T}_{Rot,\mathbf{z},\theta}\mathbf{T}_{Rot,\mathbf{y},+\beta}\mathbf{T}_{Rot,\mathbf{x},-\gamma}$$

3D Scene Representation
3D Primitives

$\mathbf{T}_{Rot,\mathbf{k},\theta}$

Equivalent Axis Rotations

An equivalent axes rotation transform is then found as

$$\mathbf{T}_{Rot,\mathbf{k},\theta} = \begin{bmatrix} xxv+c & xyv-zs & xzv+ys & 0 \\ yxv+zs & yyv+c & yzv-xs & 0 \\ zxv-ys & zyv+xs & zzv+c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where shorthand is used, with

$s = \sin(\theta)$ $c = \cos(\theta)$ $v = 1 - c$

and

$$\mathbf{k} = [x, y, z] \qquad \|\mathbf{k}\| = 1$$

3D Scene Representation
3D Primitives

$$\mathbf{T}_{Rot,\mathbf{k},\theta} = \begin{bmatrix} xxv+c & xyv-zs & xzv+ys & 0 \\ yxv+zs & yyv+c & yzv-xs & 0 \\ zxv-ys & zyv+xs & zzv+c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Final Transform Set – RECAP

The final Transform is found as:

1. Translate to origin
2. Rotate about $\mathbf{k}$ by θ
3. Translate back to original position

$$\mathbf{obj}_{rotated} = \mathbf{T}_{Tx,back}\mathbf{T}_{Rot,\mathbf{k},\theta}\mathbf{T}_{Tx,to\ origin}\mathbf{obj}_{original}$$

3D Scene Representation
3D Primitives – Example

Example

The origin of an object is defined at a position [10,20,30].

It is desired to re-orient the object by rotating it by 150° about an axis defined by the vector pointing from the object origin to another point on the object at [25, 10, 10].

Find the transform that implements the above.

$$\mathbf{T}_1 = \mathbf{T}_{Tx,back}\mathbf{T}_{Rot,\mathbf{k},\theta}\mathbf{T}_{Tx,to\ origin}$$

3D Scene Representation
3D Primitives – Example

$$\mathbf{T}_1 = \mathbf{T}_{Tx,back}\mathbf{T}_{Rot,\mathbf{k},\theta}\mathbf{T}_{Tx,to\ origin}$$

Example

$$\mathbf{T}_{Tx,to\ origin} = \begin{bmatrix} 1 & 0 & 0 & -10 \\ 0 & 1 & 0 & -20 \\ 0 & 0 & 1 & -30 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{T}_{Tx,back} = \begin{bmatrix} 1 & 0 & 0 & +10 \\ 0 & 1 & 0 & +20 \\ 0 & 0 & 1 & +30 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The rotation matrix is

$$\mathbf{T}_{Rot,\mathbf{k},150} = \begin{bmatrix} xxv+c & xyv-zs & xzv+ys & 0 \\ yxv+zs & yyv+c & yzv-xs & 0 \\ zxv-ys & zyv+xs & zzv+c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3D Scene Representation
3D Primitives – Example

$$\mathbf{T}_1 = \mathbf{T}_{Tx,back}\mathbf{T}_{Rot,\mathbf{k},\theta}\mathbf{T}_{Tx,to\ origin}$$

Example

The direction vector is found as

$$\mathbf{V}_{01} = \begin{bmatrix} 25 \\ 10 \\ 10 \end{bmatrix} - \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix} = \begin{bmatrix} 15 \\ -10 \\ -20 \end{bmatrix} \qquad \mathbf{k} = \frac{\begin{bmatrix} 15 \\ -10 \\ -20 \end{bmatrix}}{\sqrt{15^2+10^2+20^2}} = \begin{bmatrix} 0.5571 \\ -0.3714 \\ -0.7428 \end{bmatrix}$$

Check that the direction vector is of unit length

$$|\mathbf{k}| = \sqrt{(0.5571)^2 + (-0.3714)^2 + (-0.7428)^2} = 1.000025\ldots$$

OK to the 4DP retained in this illustration

Programmes should maintain much higher accuracy than 4DP

3D Scene Representation
3D Primitives – Example

$$\mathbf{k} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0.5571 \\ -0.3714 \\ -0.7428 \end{bmatrix}$$

Example

Equivalent axis rotation, θ = 150°

$s = \sin(\theta) = 0.5$ $c = \cos(\theta) = -0.866$ $v = 1 - c = 0.1340$

$$\mathbf{T}_{Rot,\mathbf{k},150} = \begin{bmatrix} -0.2869 & -0.0147 & -0.9578 & 0 \\ -0.7575 & -0.6086 & 0.2362 & 0 \\ -0.5865 & 0.7933 & 0.1635 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Check that resulting matrix is a true rotation

$$\mathbf{R} \times \mathbf{R}^T = \mathbf{I}_3$$

(The inverse of a rotation matrix is the transpose of the 3×3 rotation matrix). This is true to ~4DP here

3D Scene Representation
3D Primitives – Example

$$\mathbf{T}_1 = \mathbf{T}_{Tx,back}\,\mathbf{T}_{Rot,\mathbf{k},\theta}\,\mathbf{T}_{Tx,to\ origin}$$

Example

The final transform is

$$\mathbf{T}_1 = \begin{bmatrix} 1 & 0 & 0 & +10 \\ 0 & 1 & 0 & +20 \\ 0 & 0 & 1 & +30 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -0.2869 & -0.0147 & -0.9578 & 0 \\ -0.7575 & -0.6086 & 0.2362 & 0 \\ -0.5865 & 0.7933 & 0.1635 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -10 \\ 0 & 1 & 0 & -20 \\ 0 & 0 & 1 & -30 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} -0.2869 & -0.0147 & -0.9578 & 41.8981 \\ -0.7575 & -0.6086 & 0.2362 & 32.6608 \\ -0.5865 & 0.7933 & 0.1635 & 15.0932 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3D Scene Representation
3D Primitives – Example

$$\mathbf{T}_1 = \mathbf{T}_{Tx,back}\,\mathbf{T}_{Rot,\mathbf{k},\theta}\,\mathbf{T}_{Tx,to\ origin}$$

Example

The final transform is

$$= \begin{bmatrix} -0.2869 & -0.0147 & -0.9578 & 41.8981 \\ -0.7575 & -0.6086 & 0.2362 & 32.6608 \\ -0.5865 & 0.7933 & 0.1635 & 15.0932 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The top left **3×3 matrix** represents the compound rotation of the whole object

The top right **1×3 matrix** represents the additional translation of all points on the object

3D Scene Representation
3D Primitives – Example

$$\mathbf{T}_1 = \mathbf{T}_{Tx,back}\,\mathbf{T}_{Rot,\mathbf{k},\theta}\,\mathbf{T}_{Tx,to\ origin}$$

Example

Note that the two points that define the axis of rotation are

- The origin at a position [10,20,30]
- The second point at [25, 10, 10]

Neither points (or indeed any points along the axis) should be changed by application of the compound transformation

- This is easily verified (e.g. Matlab)
- This is also a good test to ensure your code is working correctly

(Also easily seen that off-axis points are transformed as required)

Remaining topics

- **Representation**
  - Planar Surfaces
  - Constructive Solid Geometry
  - Solids/Voxel modelling
- **Camera Models/Scene Views**
  - Perspective
- **Shading and Lighting**
  - Shading models
- **(Distributed VR)**
  - (Perception Filters)

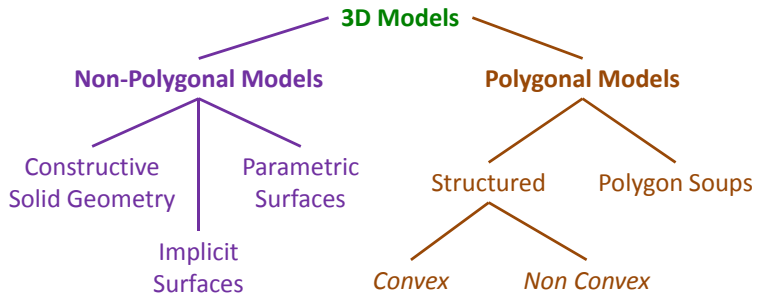
Representation

An scene can contain different type of objects (clouds, trees, stones, buildings, furniture etc.). For all of them, a wide variety of representation models are available

- **Polygonal surfaces and quadrics**
- **Spline surfaces**
- **Solid modeling**
- **Volumetric models**
- **Procedural models (fractals, particle systems,…)**
- **Physic based modeling**

Representation

There are lots of methods of representing a surface and each has advantages/disadvantages



Planar Surface Modelling

Polygonal Modelling

- **Polygon mesh: vertex, edges and polygon collection where each edge is shared by two polygons as maximum**
  - vertex: point with coordinates (x,y,z)
  - edge: line segment that joins two vertices
  - polygon: closed sequence of edges
- **Different type of representation that can be used at the same time in the same application**
  - Explicit
  - Pointers to list of vertices
  - Pointers to list of edges
- **Criteria to evaluate different representations:**
  - Time
  - Space
  - Topological Information

Planar Surface Modelling
Polygonal Modelling

Explicit Representation

- **Each polygon is represented by a list of vertex coordinates**

$$P_1 = \{(x_1, y_1, z_1), ..., (x_n, y_n, z_n)\}$$
$$P_2 = \{(x_3, y_3, z_3), ..., (x_m, y_m, z_m)\}$$

- **Vertices are stored in order (clockwise or counter-clockwise)**
- **Shared vertices are duplicated**
- **There is no explicit representation for shared vertices and edges**

- **Advantages**
  - Efficient representation for individual polygons
- **Disadvantages**
  - High storage cost
  - In order to move a vertex, it is necessary to traverse all the polygons
  - If the edges are drawn, the shared ones are drawn twice

Planar Surface Modelling
Polygonal Modelling

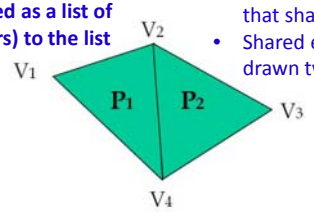
Pointers to list of vertices

- **Each vertex is stored once in a list**

$$V = \{V_1, V_2, ..., V_n\}$$
$$= \{(x_1, y_1, z_1), ..., (x_n, y_n, z_n)\}$$

- **A polygon is defined as a list of indexes (or pointers) to the list of vertices**

$$P_1 = \{V_1, V_3, V_4\}$$
$$P_2 = \{V_4, V_2, V_3\}$$



- **Advantages**
  - Each vertex is stored just once
  - Coordinates of vertices can be easily changed
- **Disadvantages**
  - Difficult to find polygons that share an edge
  - Shared edges are still drawn twice

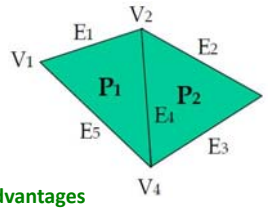
Planar Surface Modelling
Polygonal Modelling

Pointers to list of edges

- **Again, a list of vertices**
- **A polygon is defined as a list of indexes to the list of edges**
- **Each edge points to two vertices and to the polygons it belongs to**

$$E_1 = \{V_1, V_2, P_1, \lambda\}$$
$$E_2 = \{V_2, V_3, P_2, \lambda\}$$
$$E_3 = \{V_3, V_4, P_2, \lambda\}$$
$$E_4 = \{V_4, V_2, P_1, P_2\}$$
$$E_5 = \{V_4, V_1, P_1, \lambda\}$$

$$P_1 = \{E_1, E_4, E_5\}$$
$$P_2 = \{E_2, E_3, E_4\}$$

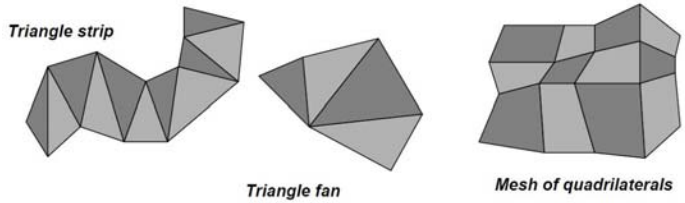


- **Advantages**
  - Each vertex is stored just once
  - The shared edges are drawn just once
  - Coordinates of vertices can be easily changed
- **Disadvantages**
  - Difficult to determine which edges share a vertex

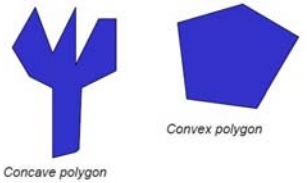
Planar Surface Modelling
Polygonal Meshes

Types of Polygonal Meshes

- **Triangle Strip**
  - For n vertices, produces (n-2) connected triangles
- **Triangle Fan**
  - For n vertices, produces (n-2) connected triangles
- **Mesh of Quadrilaterals**
  - Generates a mesh of (n-1) × (m-1) quadrilaterals for n × m vertices

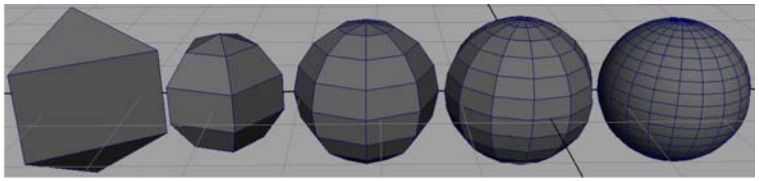


Planar Surface Modelling
Polygonal Modelling



Characteristics

- Comprise any number of polygons, usually triangles or quadrilaterals (quads)
- Concave vs Convex Shapes
- Varying levels of topological information, from none (soups) to complex connectivity graphs
- Not great for smooth curves, need huge numbers of polygons

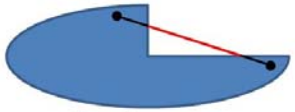


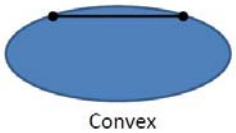
Planar Surface Modelling
Polygonal Modelling (Convex)

Convex Polygons

- Can speed up interference tests for Collision Detection if the topology has know properties
- Convex objects are much faster to test
- An object is convex *iff* (if and only if)

  "A line segment between any two points on an object is on or within its boundary"
- Two convex objects can only ever touch at one point or in one plane – very useful

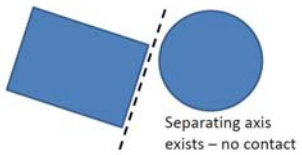


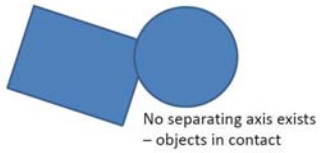


Planar Surface Modelling
Polygonal Modelling (Convex)

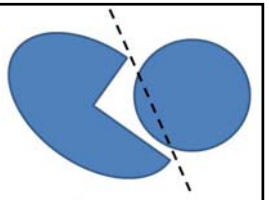
Convex Polygons

- If two objects are convex, testing for collision involves finding a plane where all points of object A lie on one side and all points of object B lie on the other
- Known as the Separating Axis Theorem (SAT)
- Many highly optimised algorithms for convex Collision Detection exist



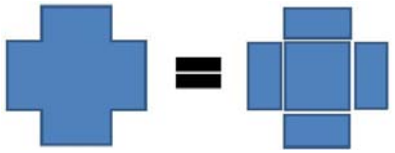


Planar Surface Modelling
Polygonal Modelling (Non-Convex)



Non-Convex Polygons

- Non-convex models much harder
- Can chop up non-convex models into convex pieces (convex decomposition)
- Heuristic search methods can result in incorrect results
- 'Brute-Force' methods test every polygon against every other polygon – which is slow
- Can make use of geometric coherence to speed up element by element approaches



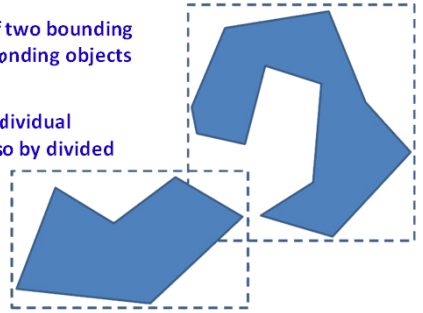
Chopping up complex objects into convex parts greatly speeds up collision detection

Geometric Coherence

- As models become populated with geometric shapes, the geometric coherence of the model needs to be maintained by avoiding overlapping of objects
  - During run-time, this coherence needs to continually tested as objects move or are moved within the environment
- To minimise the number of pair-wise tests a collision detection pass is usually divided into two phases: Broad and Narrow

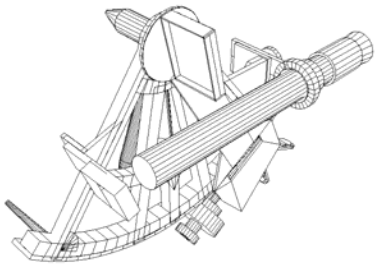
Geometric Coherence

- In a broad phase pass only the bounding boxes of objects are tested
- A bounding box is the smallest box which fits all the geometry of an object within it. Can also use other convex shapes as bounding volumes
- In a narrow-phase pass only if two bounding boxes overlap are the corresponding objects tested
- Objects comprised of many individual elements like polygons can also by divided into a tree of bounding boxes

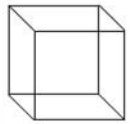


Planar Surface Modelling

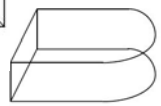
Wireframe Modelling



- **Elements:**
  - points, lines, **arcs** and **circles**, **conic** and **curves**
- **Advantages:**
  - easy to build, low memory requirements and storage
- **Disadvantages:**
  - ambiguous representation (hidden-lines removal algorithms)
  - lack of visual coherence (line-inclusion algorithms)



Ambiguity



Lack of coherence

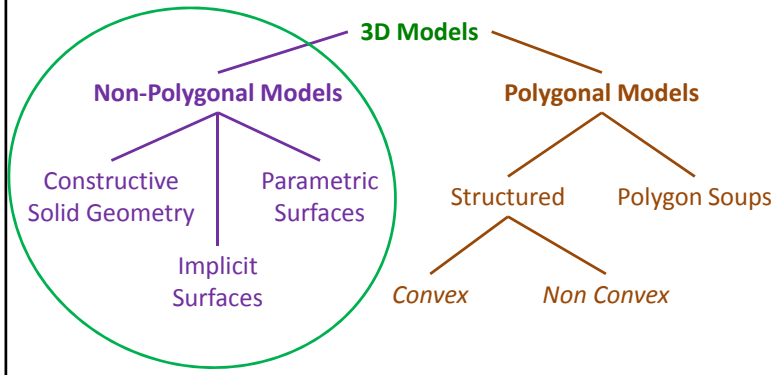
SE3VR11 : Virtual Reality

Professor Paul Sharkey p.m.sharkey@reading.ac.uk

Lecture 6 – 03/03/2016

Representation

There are lots of methods of representing a surface and each has advantages/disadvantages

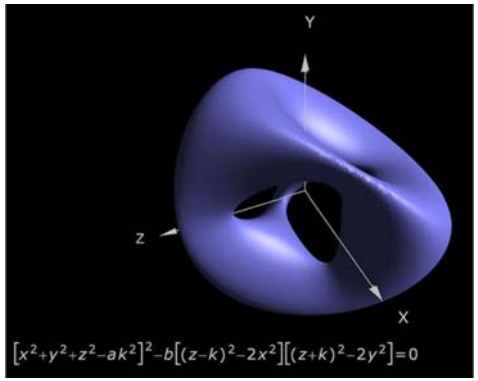


Non-Polygonal Models – Implicit Surfaces

- **Non-polygonal techniques result in smoother, less memory consuming shape representations**
- **In general distance and intersection computation is much more difficult**
- **An implicit surface is defined by the locus of points at f(x,y,z)=0 for a function which is smooth and has a non zero gradient at any point**
- I.e. a sphere is a function in x, y and z such that: $(x-a)^2+(y-b)^2+(z-c)^2-r^2=0$
- **If, when you input a coordinate in 3D space into the above equation the result is negative then it's inside the sphere, if it's positive then it lies outside**
- **If an implicit surface is defined by a low degree polynomial then collision detection is often simple. Higher degree functions are much more complicated**

Non-Polygonal Models – Implicit Surfaces

- **Implicit Surface Example**



Non-Polygonal Models – Parametric Surfaces

- Mapping a 2D planar shape into 3D coordinates forms a parametric surface
- A parametric surface need not be a topological or closed surface – unlike implicit surfaces
- Generally used to define a surface boundary rather than a complete object
- Hence, parametric objects are usually made up of many surface 'patches'
- A specific class known as Non-Uniform Rational B-Splines (NURBS) are widely used
- Finding the intersection between two parametric curves is not trivial and usually only known to a finite level of precision based on the processing time available
- Often not suitable for real-time simulation and often tessellated into polygons before performing collision detection

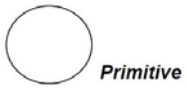
Non-Polygonal Models – Parametric Surfaces

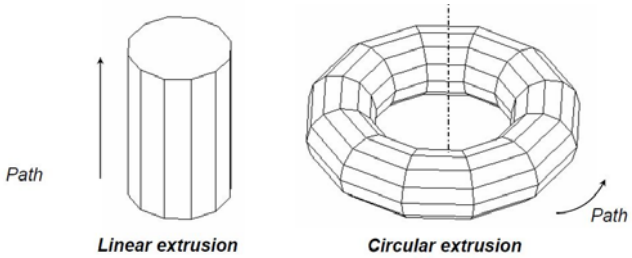
- Parametric Surface Example

Solid Modelling

Extrusion

- Extrusion
  - Useful representation in order to build object with rotational or translational symmetries and others
  - Objects are defined with a 2D primitive and a path in 3D space

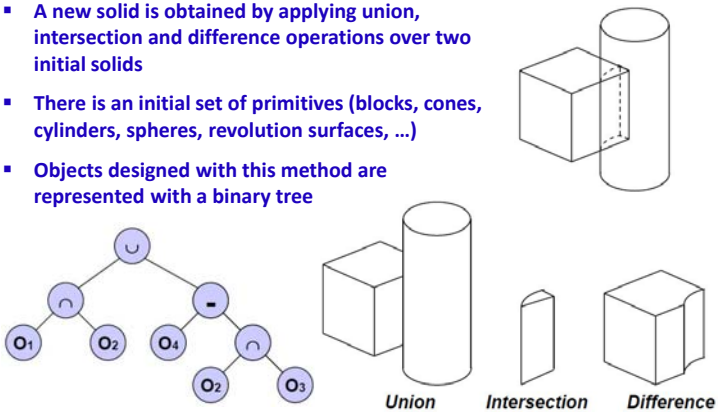




Solid Modelling

Constructive Solid Geometry (CGS)

- A new solid is obtained by applying union, intersection and difference operations over two initial solids
- There is an initial set of primitives (blocks, cones, cylinders, spheres, revolution surfaces, …)
- Objects designed with this method are represented with a binary tree



Solid Modelling

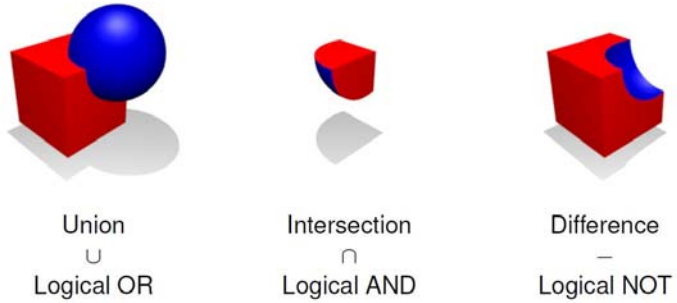
Constructive Solid Geometry (CGS)

- **Copies or "instances" of primitive shapes are created**
  - Instances can be scaled, rotated, reflected, skewed
- **A complete solid model is generated by combining "instances" using set specific, Boolean logic operations**
- **The Valid Set of Logical Operations include**
  - Union
  - Intersection
  - Difference
- **The resulting object can be further scaled, reflected, etc**
- **Or stored as a new primitive**

Solid Modelling

Constructive Solid Geometry (CGS)

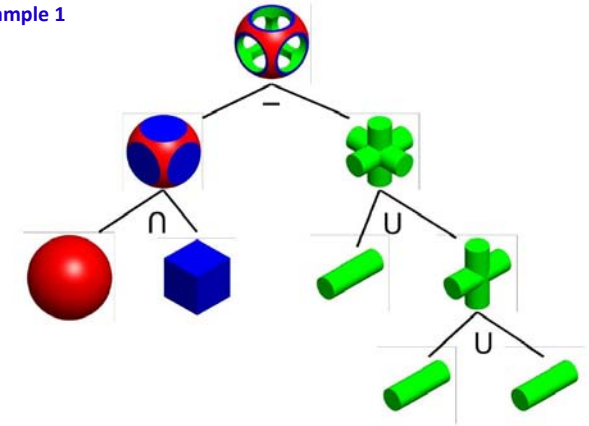
- **Union (∪) … Intersection (∩) … and Difference (–)**



Solid Modelling

Constructive Solid Geometry (CGS)

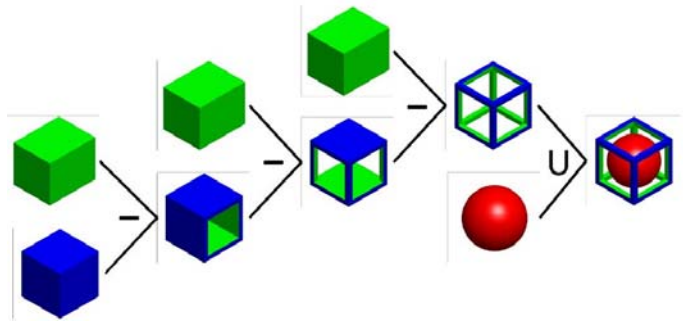
- **Example 1**



Solid Modelling

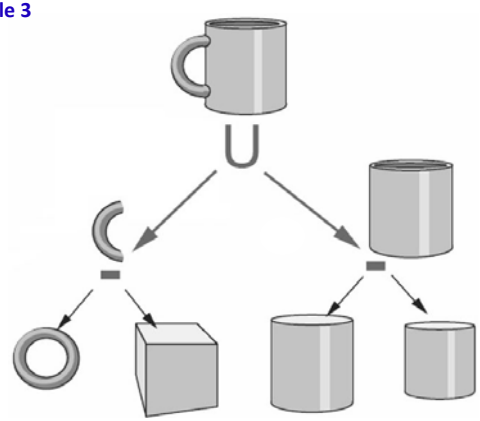
Constructive Solid Geometry (CGS)

- **Example 2**



Solid Modelling
Constructive Solid Geometry (CGS)

- **Example 3**

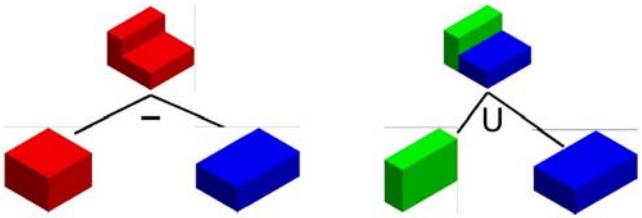


Solid Modelling
Applications of Constructive Solid Geometry (CGS)

- **Constructive solid geometry has a number of practical uses:**
  - Where simple geometric objects are desired
  - Where mathematical accuracy is important
  - Many graphics engines, such as the Unreal engine, Hammer and Torque Game Engine use CSG
  - There are also libraries that add CSG to Unity
- **CSG is popular because a modeller can use a set of relatively simple objects to create very complicated geometry**
- **One of the advantages of CSG is that you can easily assure that objects are "solid" or water-tight if all of the primitive shapes are water-tight, which can be important for some manufacturing or engineering computation applications**

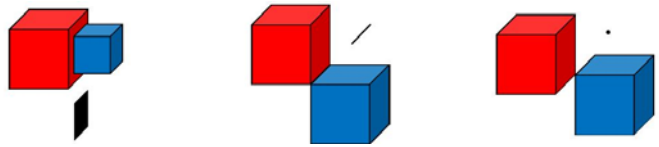
Solid Modelling
CGS Workflow

- **More than one procedure can be used to arrive at the same geometry**



Solid Modelling
CGS Hazards

- **Boolean operation:**
  - Are intuitive to the user
  - Are easy to use and understand
  - Provide for the rapid manipulation of large amounts of data
- **Because of this, many non-CSG systems also use Boolean operations!**
- **When using Boolean operations, be careful to avoid situations that do not result in a valid solid object, e.g.**



Solid Modelling
CGS Disadvantages

- **Only Boolean operations are allowed in the modeling process**
- **With Boolean operation alone, the range of shapes to be modeled is severely restricted**
- **It is not possible to construct unusual shapes**
- **Requires a great deal of computation to derive information on boundary's, faces and edges which is important for interactive display and manipulation of solid objects**

Solid Modelling
CGS Example Code

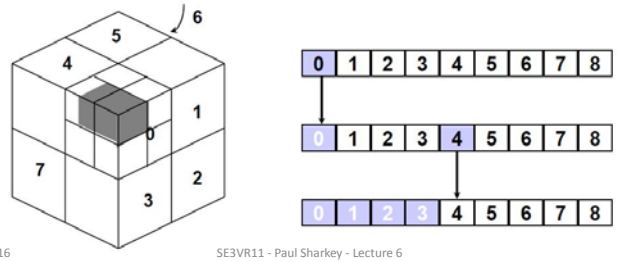
- **PovRay – natively uses CSG to create models from code representations**
- **Example:**

```
cylinder { <0,-4,0> <0,4,0> 0.2 pigment { Yellow } }
difference {
    box { <-1.5,-0.5,-0.5> <1.5,0.5,0> 0.1 }
    cylinder { <-0.8,0,-3> <-0.8,0,3> 0.1 }
    cylinder { <0.8,0,-3> <0.8,0,3> 0.1 }
    pigment { Red }
}
```

- **PovRay – Persistence of Vision Raytracer – www.povray.com**
  - Freeware multi-platform ray-tracing package

Solid Modelling
Spatial Numbering (Octtrees)/Voxel Modelling

- **Hierarchical tree structure, in which each node is a region of 3D space**
- **3D space regions are divided in octants (cubes), 8 elements are stored in each node of the tree**
- **Nodes can be of 3 types:** white (out the object); black (inside the object)
  - grays (neither inside nor outside)
- **Individual elements of the 3D space are called voxels**



Components for 3D Graphics

Next Topics

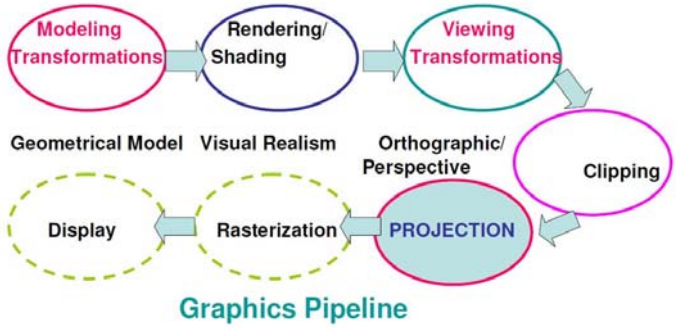
- **Representation**
  - Surfaces/Solids/Voxels
- **Scene Geometry**
- **Camera Models**
  - Projective & Perspective Geometry
  - Ray Tracing
  - Shading
- **(Distributed VR)**
  - (Perception Filters)

3D Viewing and Projective Geometry

- **Our eyes collapse 3-D world to 2-D retinal image**
  - The brain then has to reconstruct 3D
- **In the VR modelling engine, this process occurs by projection**
- **Projection has two parts:**
  - Viewing transformations:
    - camera position and direction
  - Perspective/orthographic transformation:
    - reduces 3-D to 2-D
- **Each use homogeneous transformations**

Projective Geometry

As part of the Graphics Pipeline

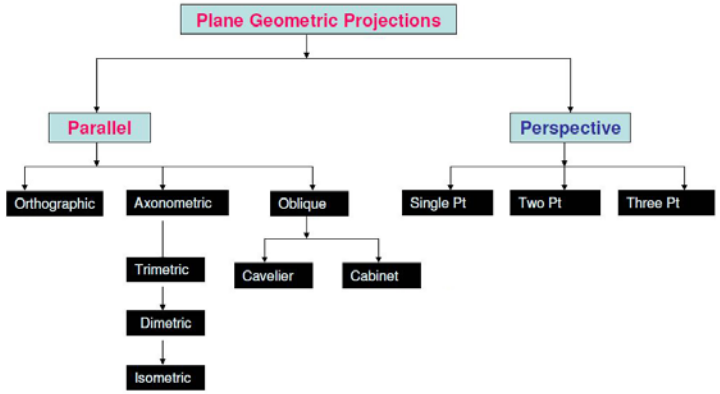


Projective Geometry

- **Projections come in many forms**
- **Can be generally grouped into two types:**
  - Affine or Perspective Projections
- **Both affine and perspective transformations are 3-dimensional**
- **They are transformations from one 3-D space to another**
- **Viewing 3D transformations (results) on a 2-Dimensional surface(screen) requires projections from 3-Space to 2-Space**
- **This is known as plane geometric projection**

Projective Geometry

Projections Classifications



Projective Geometry
Parallel Projections

- Parallel projection discards the z-coordinate
- Parallel lines from each vertex on the object are extended until they intersect the view plane
- A direction of projection is specified instead of center of projection
- The distance from the center of projection to project plane is infinite

Projective Geometry
Parallel Projections

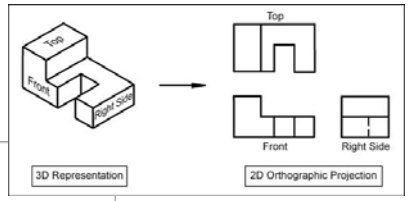
- The projected vertices are connected by line segments which correspond to connections on the original object
- Parallel projections are less realistic, but they are good for exact measurements
- In this type of projections, parallel lines remain parallel and angles are not preserved
- Such projections are also termed affine projections

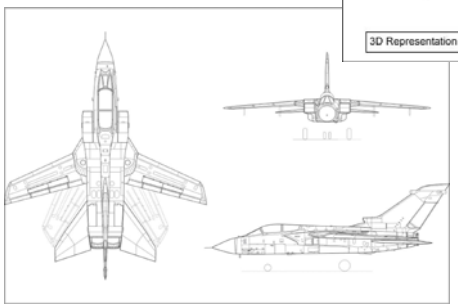
Projective Geometry
Orthographic Projections

- In orthographic projection the direction of projection is normal to the projection of the plane
- There are three types of orthographic projections
  - Front Projection
  - Top Projection
  - Side Projection

Projective Geometry
Orthographic Projections

- Examples



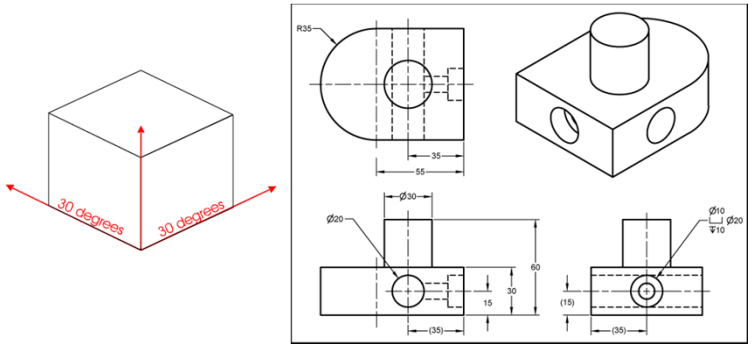


Projective Geometry
Anoxometric Projections

- Orthographic projections that show more than one side of an object are called axonometric orthographic projections
- The most common axonometric projection is an isometric projection where the projection plane intersects each coordinate axis in the model coordinate system at an equal distance
- In this projection parallelism of lines are preserved but angles are not preserved

Projective Geometry
Anoxometric Projections

- Examples – isometric projection



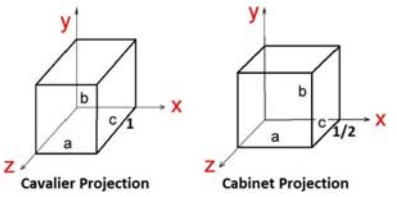
Projective Geometry
Oblique Projections

- In oblique projection the direction of projection is not normal to the projection of the plane (it is oblique)
- We can view the object better than orthographic projection
- Two types of oblique projections – Cavalier and Cabinet
- Cavalier projection makes 45° angle with the projection plane
  - Lines perpendicular to the view plane have the same length as the line itself. The foreshortening factors for all three principal directions are equal.
- Cabinet projection makes 63.4° angle with the projection plane
  - Lines perpendicular to the viewing surface are projected at ½ their actual length

Projective Geometry
Oblique Projections

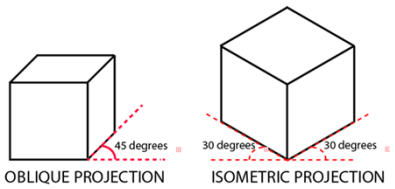
- Example –

  Cabinet vs Cavalier



- Example –

  Oblique vs Isometric

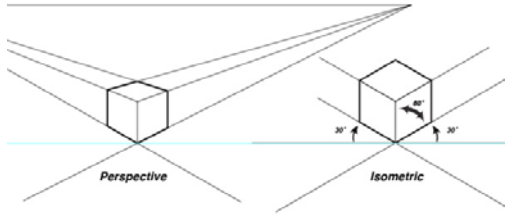


Projective Geometry
Perspective Projections

- In perspective projection the distance from the center of projection to project plane is finite and the size of the object varies inversely with distance which looks more realistic
- The distance and angles are not preserved and parallel lines do not remain parallel
- They all converge at a single point called center of projection
- There are 3 types of perspective projections
  - One point perspective projection is simple to draw
  - Two point perspective projection gives better impression of depth
  - Three point perspective projection is most difficult to draw

Projective Geometry
Perspective Projections

- Perspective vs Isometric



SE3VR11 : Virtual Reality

Professor Paul Sharkey p.m.sharkey@reading.ac.uk

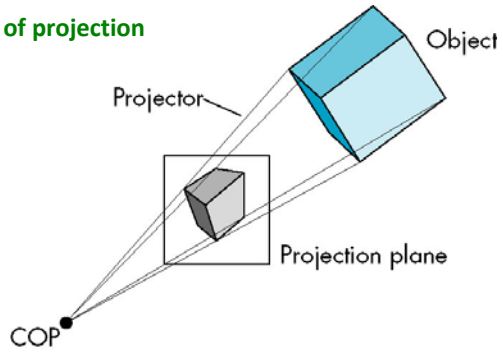
Lecture 7 – 10/03/2016

Projective Geometry
Perspective Projections

- In perspective projection the distance from the center of projection to projection plane is finite and the size of the object varies inversely with distance which looks more realistic
- The distance and angles are not preserved and parallel lines do not remain parallel
- They all converge at a single point called the center of projection

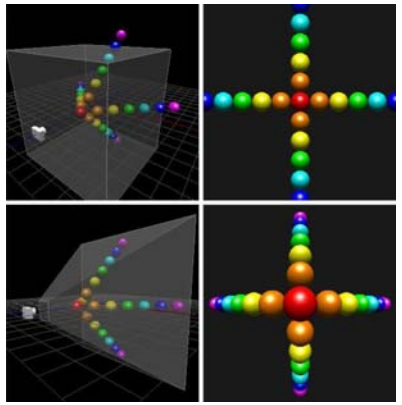
Projective Geometry
Perspective Projections

- The center of projection



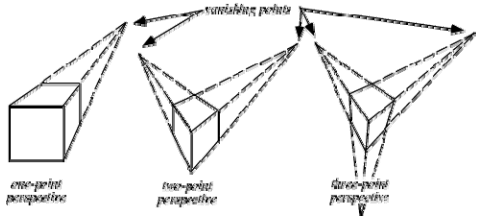
Projective Geometry
Perspective Projections

- Images on left: a 3D scene with view volumes
- Images on the right: projections onto a view plane
- Images on the top: parallel projection
- Images on the bottom: perspective projection



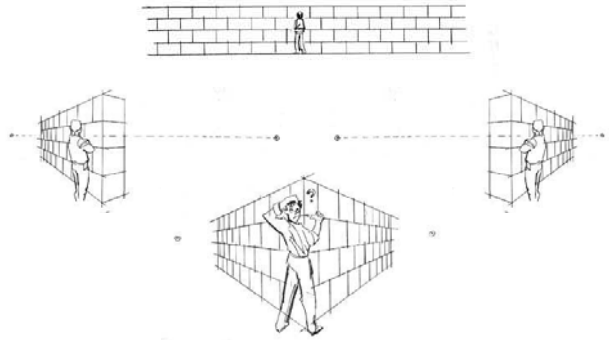
Projective Geometry
Perspective Projections

- **The 3 types of perspective projections**
  - One point perspective projection is simple to draw
  - Two point perspective projection gives better impression of depth
  - Three point perspective projection is most difficult to draw



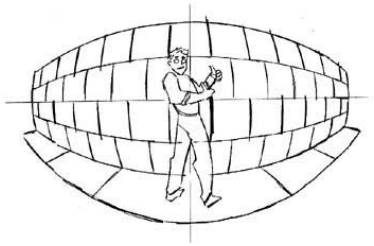
Projective Geometry
Perspective Projections

- **The comic book "5-point perspective"**



Projective Geometry
Perspective Projections

- **The comic book "5-point perspective"**



- **AKA Curvilinear Perspective**
- **Is this a realistic presentation?**

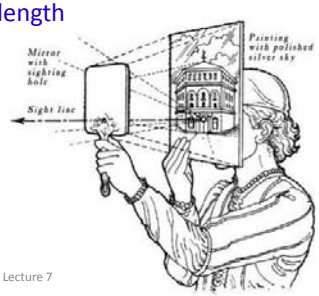
Projective Geometry
Perspective Projections

- **Is 1-point perspective the only true perspective?**

- **Vanishing points are related to eye direction and change as the eye changes the view direction**
- **So difficult to argue for even 2-point as a "natural perspective"**

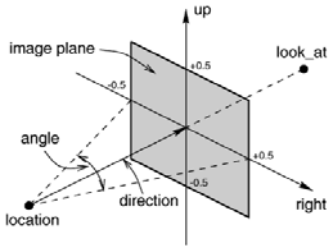
Projective Geometry
Perspective Projections

- For the rendering of objects in 3D space, a planar view has to be generated
- 3D space is projected onto a 2D plane considering external and internal camera parameters
  - position, orientation, focal length



Projective Geometry
Perspective Projections

- To generate a realistic perspective viewpoint, we need to develop the mathematical tools to do so
- In homogeneous notation, 3D projections can be represented with a 4×4 transformation matrix



Projective Geometry
2D Homogenous coordinates

- To understand 3D perspective transforms and projections let us first consider 2D homogenous coordinates
- A 2D point (x, y) can be manipulated in translation and rotation (reflection, skew, etc) using 3×3 homogenous transformation matrices as

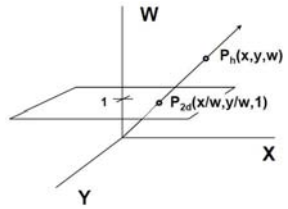
$$\mathbf{p} = \begin{bmatrix} x \\ y \end{bmatrix} \Rightarrow \mathbf{p}_1 = \begin{bmatrix} m_{11} & m_{12} & t_x \\ m_{21} & m_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix}$$

- This is a 2D version of the 3D case in lectures 4, 5

Projective Geometry
2D Homogenous coordinates

- In homogenous coordinate system the point is represented in a generalised form

$$\mathbf{p}_h = \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$



- The 2D planar point is recovered by dividing all terms by the scalar w

$$\mathbf{p}_{2D} = \begin{bmatrix} x/w \\ y/w \\ 1 \end{bmatrix}$$

Projective Geometry
2D Homogenous coordinates

- In homogenous coordinate system a generalised transformation matrix can then be used

$$\mathbf{p} = \begin{bmatrix} x \\ y \end{bmatrix} \Rightarrow \mathbf{p}_1 = \begin{bmatrix} m_{11} & m_{12} & t_x \\ m_{21} & m_{22} & t_y \\ \rho_x & \rho_y & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x_1 \\ y_1 \\ w_1 \end{bmatrix}$$

- The two parameters on the bottom row allow for perspective transformation

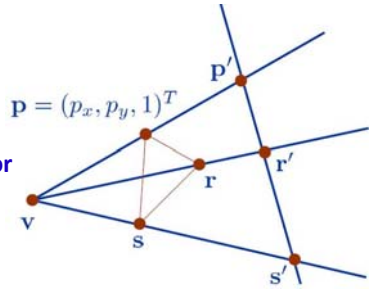
Projective Geometry
2D Homogenous coordinates

- ***Finally***, the last value (1, bottom right) can be generalised as s to allow global scaling
- The most general homogenous transform is then

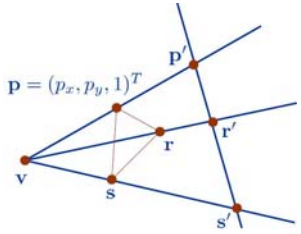
$$\mathbf{p}_1 = \begin{bmatrix} m_{11} & m_{12} & t_x \\ m_{21} & m_{22} & t_y \\ \rho_x & \rho_y & s \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix} = \begin{bmatrix} x_1 \\ y_1 \\ w_1 \end{bmatrix}$$

Projective Geometry
2D Homogenous coordinates

- Consider a point **p** being mapped onto a line **l** from a viewpoint **v** as shown in the diagram
- the point **v** is the viewpoint, the centre of perspectivity
- the line **l** is the viewline
- the line through **p** and **v** is the projector
- the point **p′** is the intersection of the projector with the line **l**
- the viewpoint is not on the viewline and $\mathbf{v} \neq \mathbf{p}$



Projective Geometry
2D Homogenous coordinates



- the point **v** is given as

$$\mathbf{v} = \left(v_x, v_y, v_w\right)^T$$

- the line **l** can be represented as

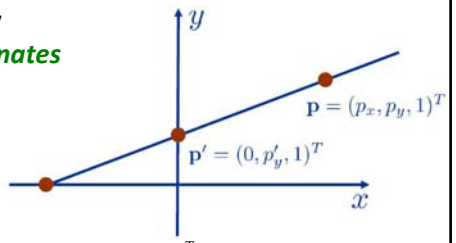
$$\mathbf{l} = \left\{ax + by + c = 0\right\} = \left(a, b, c\right)^T$$

- Then the 2D projection matrix can be defined as

$$\mathbf{T} = \mathbf{v}\mathbf{l}^T - \left(\mathbf{l} \cdot \mathbf{v}\right)\mathbf{I}_3$$

Projective Geometry
2D Homogenous coordinates

- **Example**



$\mathbf{v} = (d, 0, 1)^T$

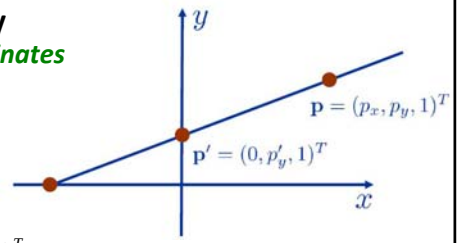
$\mathbf{l} = \{x = 0\} = \{1x + 0y + 0 = 0\} = (1, 0, 0)^T$

- **Then the 2D projection matrix can be defined as**

$$\mathbf{T} = \begin{bmatrix} d \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} - \left(\begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} d \\ 0 \\ 1 \end{bmatrix} \right) \mathbf{I}_3 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & -d & 0 \\ 1 & 0 & -d \end{bmatrix}$$

Projective Geometry
2D Homogenous coordinates

- **With** $d = -1$



the point $\mathbf{p} = (1, 2)^T$

is mapped to $\mathbf{p}' = (0, 1)^T$

as follows:

$$\mathbf{Tp} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & -d & 0 \\ 1 & 0 & -d \end{bmatrix} \mathbf{p} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & -1 & 0 \\ 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 2 \\ 2 \end{bmatrix} \equiv \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} \approx \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Projective Geometry
2D Homogenous coordinates

- **The point p can be written as**

$$\mathbf{p} = \begin{bmatrix} x \\ y \\ w \end{bmatrix} \text{ or } \mathbf{p} = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \text{ or } \mathbf{p} = k \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

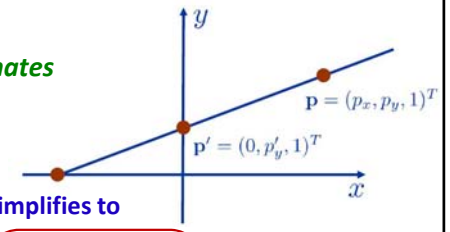
where k **can be any constant**

- **Therefore the transform** $k\mathbf{T}$ **represents the same transform as** $\mathbf{T}$ **because**

$$\mathbf{Tp} = \mathbf{T}(k\mathbf{p}) = k\mathbf{Tp} \quad \Rightarrow \quad \mathbf{T} \equiv k\mathbf{T}$$

Projective Geometry
2D Homogenous coordinates

With $k = \dfrac{-1}{d}$



the 2D projection matrix simplifies to

$$\mathbf{T} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & -d & 0 \\ 1 & 0 & -d \end{bmatrix} \Leftrightarrow \mathbf{T} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ \dfrac{-1}{d} & 0 & 1 \end{bmatrix}$$

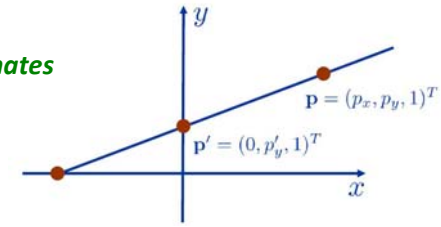
(As d tends to infinity this becomes a parallel projection onto the $x = 0$ line)

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

Projective Geometry
2D Homogenous coordinates

Compare

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



with the general form

$$\mathbf{T} = \begin{bmatrix} m_{11} & m_{12} & t_x \\ m_{21} & m_{22} & t_y \\ \rho_x & \rho_y & s \end{bmatrix}$$

Clearly the perspective transform parameters are related to the centre of projection, being the (negative) inverse of the COP point coordinates

Projective Geometry
2D Homogenous coordinates

"Perspective Transform" or "2D Projection matrix"

- Why use different terms?
  - "2D Projection matrix" (from above)

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

  - "Perspective Transform"

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

Projective Geometry
2D Homogenous coordinates

"Perspective Transform" or "2D Projection matrix"

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \frac{-1}{d} & 0 & 1 \end{bmatrix} \qquad \mathbf{T} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ \frac{-1}{d} & 0 & 1 \end{bmatrix}$$

- The perspective transform (left) will transform the 2D space into another 2D space – there is no loss of information
- The 2D projection (right) projects 2D space onto a 1D line so depth information is lost

Projective Geometry
3D Homogenous coordinates

- For the 3D case, instead of a viewline, we need to consider a viewplane
- Everything is calculated in the same way and a 4×4 transform results

$$\mathbf{p} = \left(p_x, p_y, p_z, 1\right)^T$$

$$\mathbf{v} = \left(v_x, v_y, v_z, v_w\right)^T$$

$$\mathbf{n} = \left\{ax + by + cz + d = 0\right\} = \left(a, b, c, d\right)^T$$

$$\mathbf{T} = \mathbf{v}\mathbf{n}^T - \left(\mathbf{n} \cdot \mathbf{v}\right)\mathbf{I}_4$$

Projective Geometry
3D Homogenous coordinates

- The most general homogenous transform for 3D case is

$$\mathbf{p}_1 = \begin{bmatrix} m_{11} & m_{12} & m_{13} & t_x \\ m_{21} & m_{22} & m_{23} & t_y \\ m_{31} & m_{32} & m_{33} & t_z \\ \rho_x & \rho_y & \rho_z & s \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ w_1 \end{bmatrix}$$

- Top left 3×3: rotation, shear, reflection, scaling
- Top right 3×1: translation
- Bottom left 1×3: perspective
- Top right 1×1: global scaling

Projective Geometry
3D Homogenous coordinates

- We can now use this transformation matrix to transform the 3D virtual world model into a (3D) perspective view from any viewpoint coordinates

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \rho_x & \rho_y & \rho_z & 1 \end{bmatrix}$$

where the centre of projection is $\mathbf{p}_{COP} = \begin{bmatrix} -1/\rho_x \\ -1/\rho_y \\ -1/\rho_z \end{bmatrix}$

Projective Geometry
3D Homogenous coordinates

- It is usual to treat the camera axis as the z axis
- Hence the centre of projection is $COP = (0, 0, d_z)$
- The perspective projection is onto the $z = 0$ plane
- The final 3D→2D projection matrix takes the form

$$\mathbf{T}_{projection} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \rho_z & 1 \end{bmatrix} \qquad \rho_z = -\frac{1}{d_z}$$

- d_z can also be the focal length of the camera lens

Projective Geometry
3D Homogenous coordinates

- To find the inverse of a perspective transform, simply negate the COP coordinates

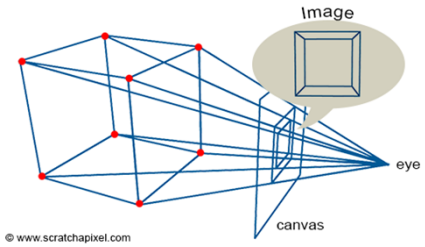
$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \rho_z & 1 \end{bmatrix} \qquad \mathbf{T}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -\rho_z & 1 \end{bmatrix}$$

- This will be useful later
- Note that this is the inverse of the transform and not the projection
- The inverse of a projection matrix is meaningless as all information on depth has been lost

Projective Geometry

Visualising 3D Perspective Transforms and Projections

- Visualising 3D Perspective Transforms and Projections



(this is a projection) – more next week

SE3VR11 : Virtual Reality

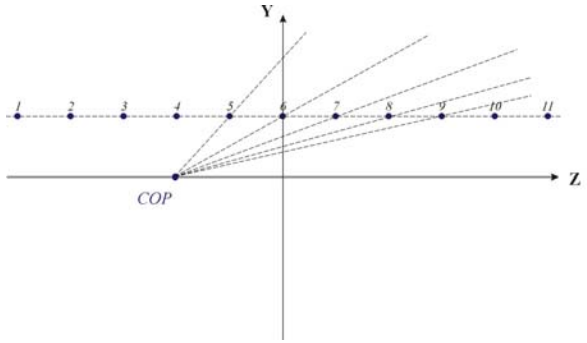
Professor Paul Sharkey p.m.sharkey@reading.ac.uk

Lecture 8 – 17/03/2016

Projective Geometry

Visualising 3D Perspective Transforms and Projections

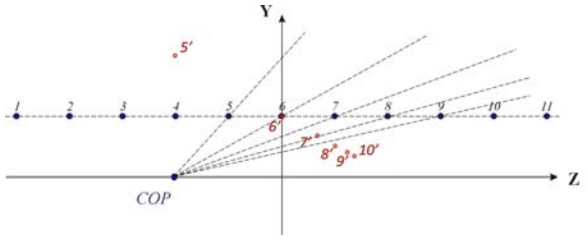
- **Straight line**



Projective Geometry

Visualising 3D Perspective Transforms and Projections

- **Straight line**

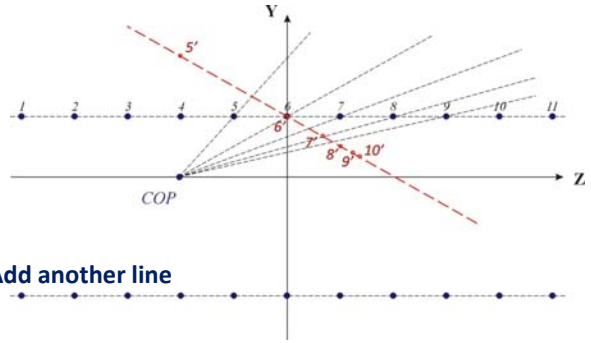


- the red points are correctly placed – but obviously do not match those projector lines
- they do, however, appear to be co-linear in themselves

Projective Geometry

Visualising 3D Perspective Transforms and Projections

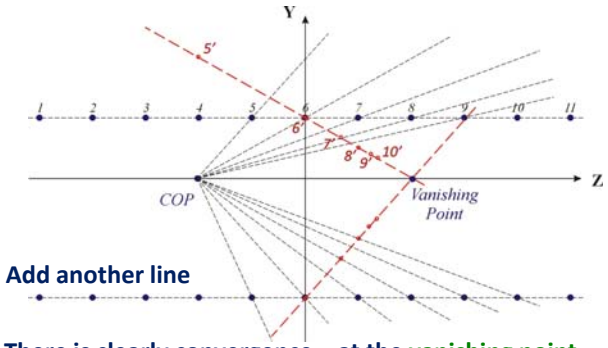
- **Straight line**



- **Add another line**

Projective Geometry
Visualising 3D Perspective Transforms and Projections

- Straight line

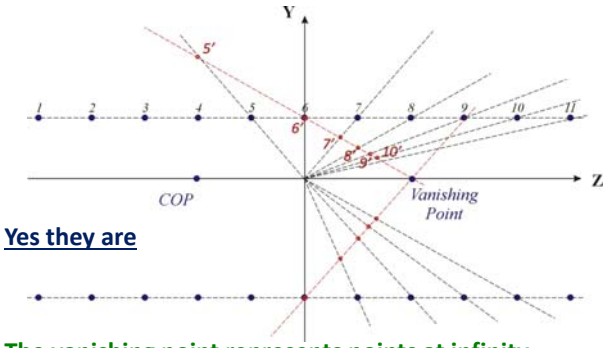


- Add another line

- There is clearly convergence – at the vanishing point
- The vanishing point and the COP are equidistant from the origin

Projective Geometry
Visualising 3D Perspective Transforms and Projections

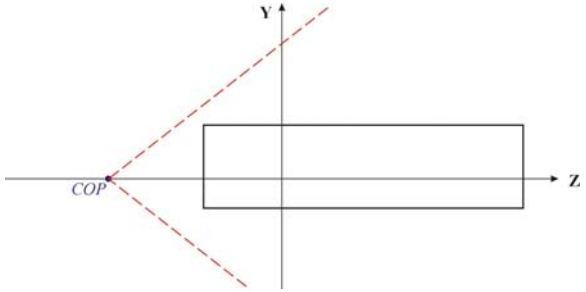
- Are points p and p' geometrically related ?



- Yes they are
- The vanishing point represents points at infinity

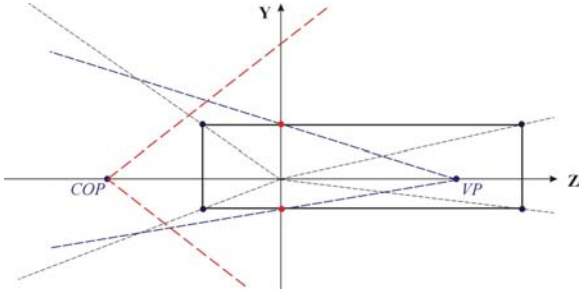
Projective Geometry
Visualising 3D Perspective Transforms and Projections

- Consider a Real World with Camera defined at COP and a single object (a cuboid) within
- The camera has a defined field of view



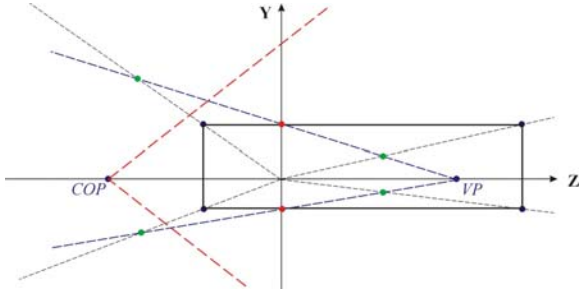
Projective Geometry
Visualising 3D Perspective Transforms and Projections

- The Vanishing Point and points of interest on the object are found
- Radial lines constructed



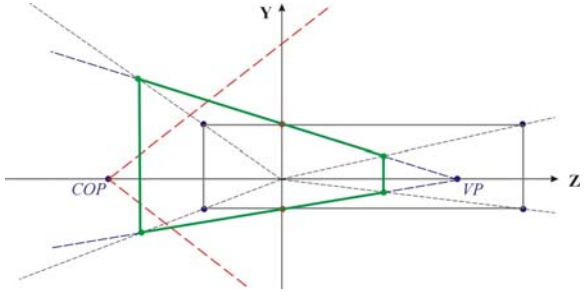
Projective Geometry
Visualising 3D Perspective Transforms and Projections

- Now corresponding corners can be found in the perspective view at the intersection of the radial lines
- And the perspective view of the shape can be determined



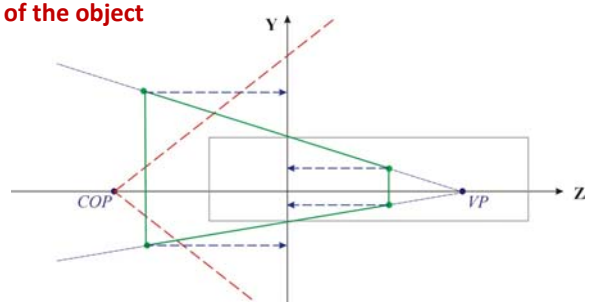
Projective Geometry
Visualising 3D Perspective Transforms and Projections

- Now corresponding corners can be found in the perspective view at the intersection of the radial lines
- And the perspective view of the shape can be determined



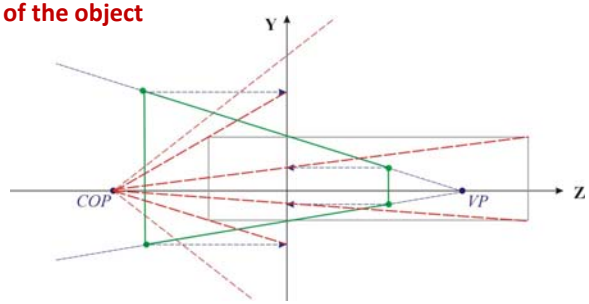
Projective Geometry
Visualising 3D Perspective Transforms and Projections

- A 2D Projected view can be found be projecting the perspective view back onto the $z = 0$ plane
- All being well, this should correspond to the camera view of the object



Projective Geometry
Visualising 3D Perspective Transforms and Projections

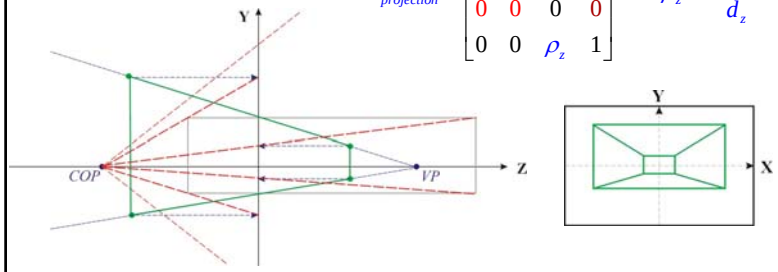
- A 2D Projected view can be found be projecting the perspective view back onto the $z = 0$ plane
- All being well, this should correspond to the camera view of the object



Projective Geometry
Visualising 3D Perspective Transforms and Projections

- The camera view is shown on the right
- All of that can be achieved by applying the projection matrix to the 8 corners of the object

$$\mathbf{T}_{projection} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \rho_z & 1 \end{bmatrix} \qquad \rho_z = -\frac{1}{d_z}$$



Projective Geometry
Visualising 3D Perspective Transforms and Projections

- The previous 3D object is defined as

$$OBJ = \begin{bmatrix} -40 & -40 & 40 & 40 & -40 & -40 & 40 & 40 \\ -10 & 25 & -10 & 25 & -10 & 25 & -10 & 25 \\ -20 & -20 & -20 & -20 & 50 & 50 & 50 & 50 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

- The Perspective (homogenous) view of the object (d = -40) is

$$OBJ_{persp} = \begin{bmatrix} -40 & -40 & 40 & 40 & -40 & -40 & 40 & 40 \\ -10 & 25 & -10 & 25 & -10 & 25 & -10 & 25 \\ -20 & -20 & -20 & -20 & 50 & 50 & 50 & 50 \\ 0.5 & 0.5 & 0.5 & 0.5 & 2.25 & 2.25 & 2.25 & 2.25 \end{bmatrix}$$

i.e. $1+\frac{-20}{40}=0.5$ $\qquad$ $1+\frac{50}{40}=2.25$

Projective Geometry
Visualising 3D Perspective Transforms and Projections

- The Perspective (homogenous) view of the object (d = -40) is

$$OBJ_{persp} = \begin{bmatrix} -40 & -40 & 40 & 40 & -40 & -40 & 40 & 40 \\ -10 & 25 & -10 & 25 & -10 & 25 & -10 & 25 \\ -20 & -20 & -20 & -20 & 50 & 50 & 50 & 50 \\ 0.5 & 0.5 & 0.5 & 0.5 & 2.25 & 2.25 & 2.25 & 2.25 \end{bmatrix}$$

- Equivalently

$$OBJ_{persp} = \begin{bmatrix} -80 & -80 & 80 & 80 & -17.78 & -17.78 & 17.78 & 17.78 \\ -20 & 50 & -20 & 50 & -4.44 & 11.11 & -4.44 & 11.11 \\ -40 & -40 & -40 & -40 & 22.22 & 22.22 & 22.22 & 22.22 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Projective Geometry
Visualising 3D Perspective Transforms and Projections

- Perspective 3D object

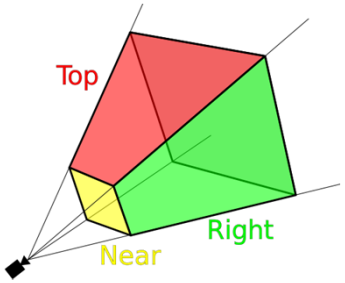
$$OBJ_{persp} = \begin{bmatrix} -80 & -80 & 80 & 80 & -17.78 & -17.78 & 17.78 & 17.78 \\ -20 & 50 & -20 & 50 & -4.44 & 11.11 & -4.44 & 11.11 \\ -40 & -40 & -40 & -40 & 22.22 & 22.22 & 22.22 & 22.22 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

- Projected Perspective View (Screen)

$$OBJ_{persp} = \begin{bmatrix} -80 & -80 & 80 & 80 & -17.78 & -17.78 & 17.78 & 17.78 \\ -20 & 50 & -20 & 50 & -4.44 & 11.11 & -4.44 & 11.11 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Projective Geometry
Clipping and the View Frustum

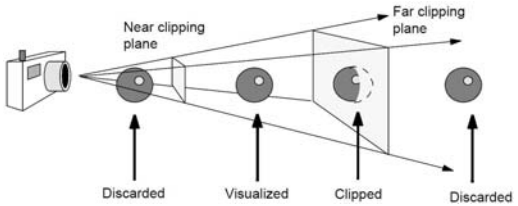
- The camera view of a 3D scene is confined to a view frustum
  - A frustum is simply a 3D object truncated by two parallel planes
- A camera has horizontal and vertical fields of view that define a pyramidal shape
- The frustum is found by defining, in addition, a near clipping frame and a far clipping frame



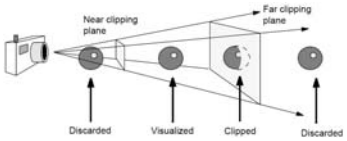
Projective Geometry
Clipping and the View Frustum

Why clip?

- Objects that are not within the camera's view do not need to be rendered
- It saves computation to pre-determine these objects and remove them from the list of objects to be processed
- Objects outside the frustum need not be not drawn
- Objects that intersect are clipped



Projective Geometry
Clipping and the View Frustum

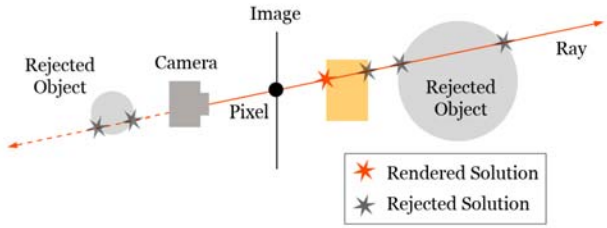


Why clip near and far planes?

- At the near plane, objects that are too close will obscure the whole model so a preferred interaction is to remove such near objects
- At the far plane, objects are diminished in size (vanishing point effect), and can be obscured by closer scene elements
- Other terms for clipping include frustum culling, occlusion culling, and depth- or "z" clipping
  - z refers to optical direction

Projective Geometry
Clipping and the View Frustum

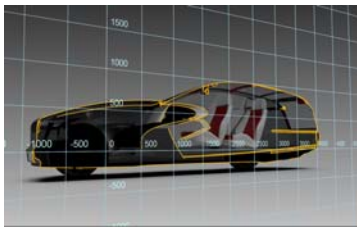
- Occlusion culling

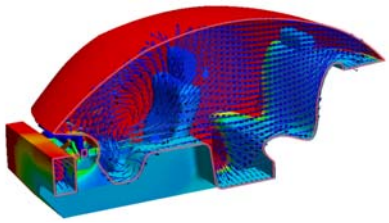


Projective Geometry
Clipping and the View Frustum

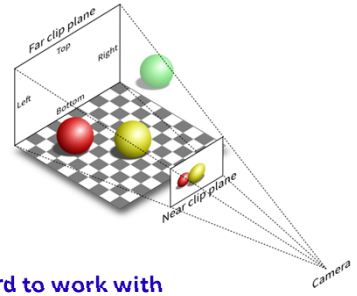
Creative clipping

- A clipping plane need not be restricted to the camera parameter
- A general clipping plane can be arbitrarily defined to reveal unusual views of objects and structures





Projective Geometry
Clipping and the View Frustum

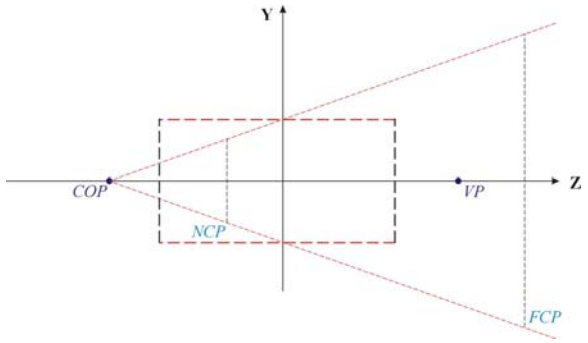


How to clip a view frustum

- A pyramidal frustum is awkward to work with
- Using the perspective transform, the frustum is transformed into a regular cuboid
- As the camera's fields of view and the user defined near and far planes are fixed … the cuboid so generated as well defined parallel planes in the 3D perspective view
- All objects outside of these delimits can be ignored

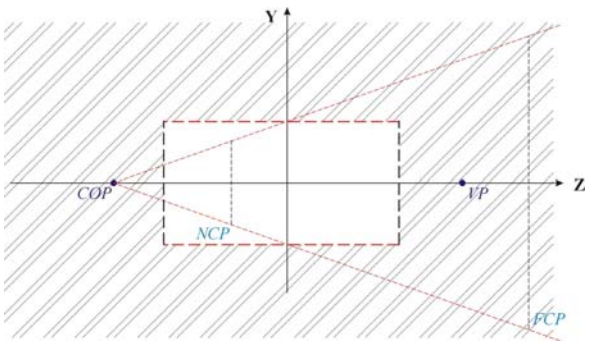
Projective Geometry
Clipping and the View Frustum

- Frustum converts to cuboid under perspective transform



Projective Geometry
Clipping and the View Frustum

- Frustum converts to cuboid under perspective transform

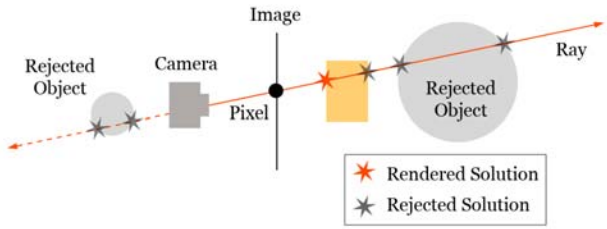


Camera/User Motion

- To achieve this a continuously varying view of the 3D model needs to be "instantaneously" updated as the user moves their viewpoint
- The viewpoint/viewplane can be translated and rotated
- The direction and orientation of the viewpoint needs to be accommodated in the perspective transform
- Rather than move the camera, the 3D model is rotated and translated so that perspective transform is always along the z-axis
- Clearly the use of clipping algorithms optimises this process

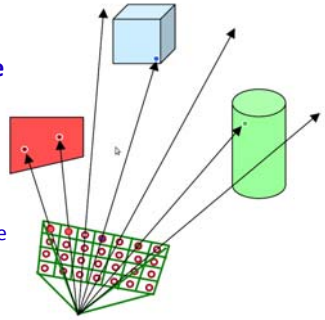
Ray Casting
AKA Ray Tracing

- Ray casting is the use of surface intersection tests to solve a variety of problems in computer graphics or computational geometry
- Here we use it to determine whether or not an object will need to be rendered



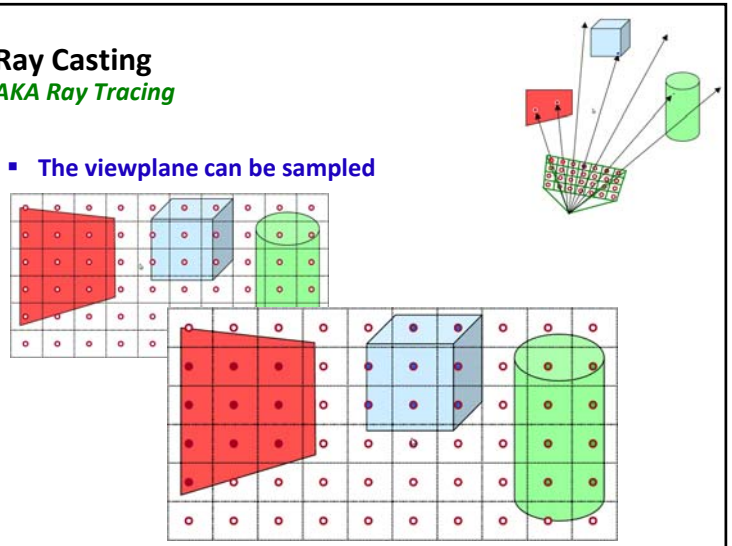
Ray Casting
AKA Ray Tracing

- The colour of each pixel on the viewing plane depends on the radiance emanating from visible surfaces
- For each sample:
  - Construct a ray from the eye position through the view plane
  - Find first surface intersected by the ray
  - Compute the colour of the sample based on the surface radiance



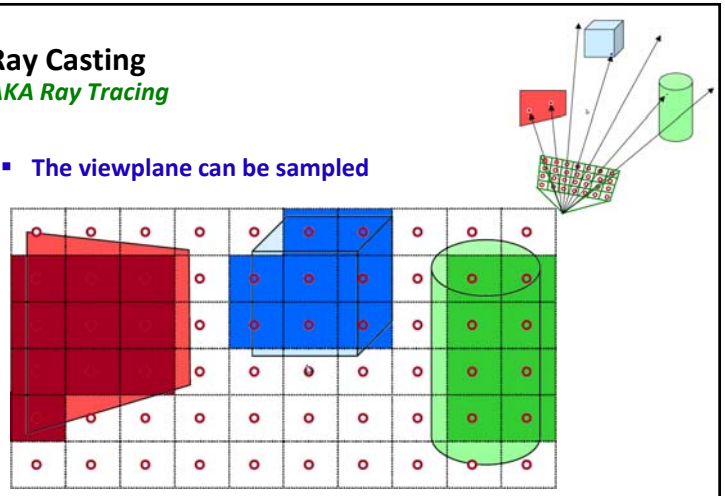
Ray Casting
AKA Ray Tracing

- The viewplane can be sampled



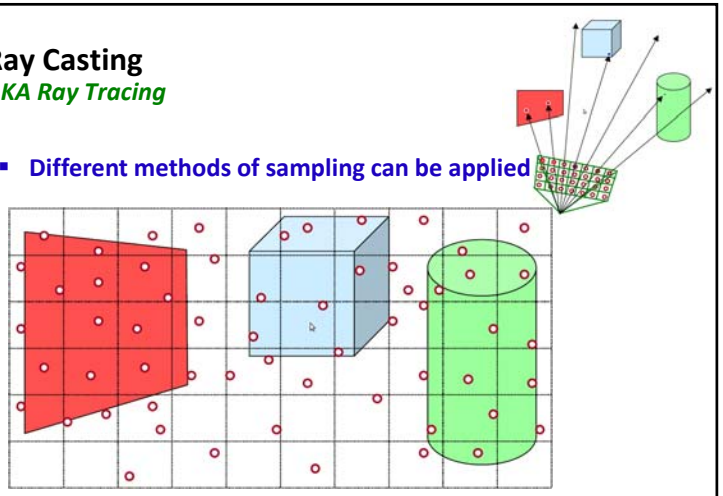
Ray Casting
AKA Ray Tracing

- The viewplane can be sampled



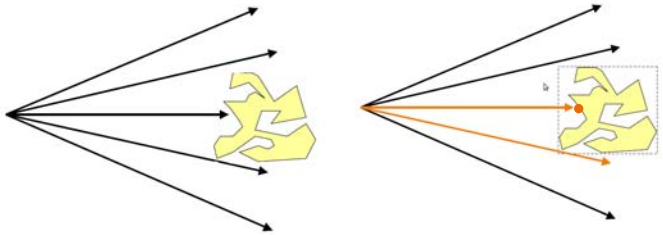
Ray Casting
AKA Ray Tracing

- Different methods of sampling can be applied



Bounding Boxes

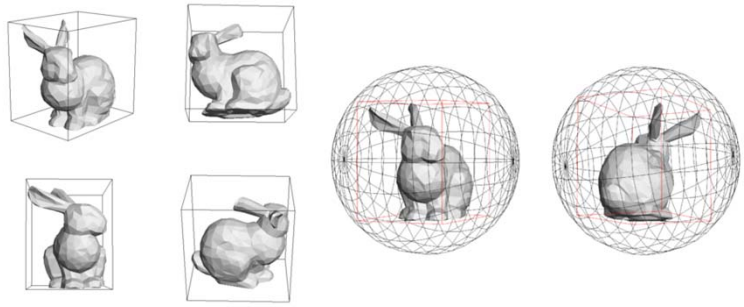
- Complex objects are expensive to compute collisions on
- Complex objects are expensive to compute collisions on
- Check for intersection with simple shapes first



- Still need to check within the bounding volume for actual intersections with the shape

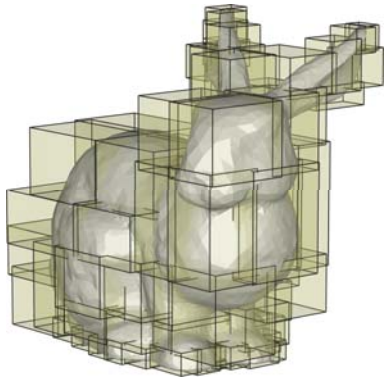
Bounding Boxes

- Bounding “boxes” come in different shapes



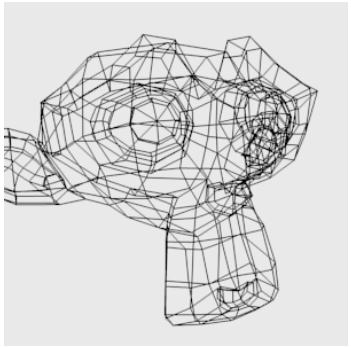
Bounding Boxes

- Bounding boxes can define sub-elements of objects



Light and Shade
Illuminating the Model

- We can now represent our 3D objects:



Light and Shade
Illuminating the Model

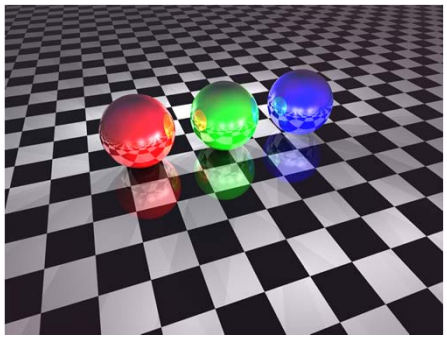
- So far our image looks like this:

Light and Shade
Illuminating the Model

- By simulating a light source we are able to change the illumination of the polygons to bring these details back
- To do this we need to derive computer models for:
  - Emission at a light source
  - Scattering at surfaces
  - Reception at the camera
- We need to describe the intensity of the light energy ...
  - leaving a light source
  - arriving at a location/surface
  - from a particular direction
  - with a given wavelength

Light and Shade
Illuminating the Model

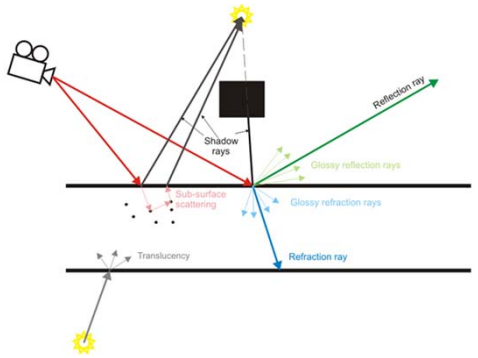
- Ideally we would need to model all light sources radiating off all surfaces



- including reflections … refractions … shadows

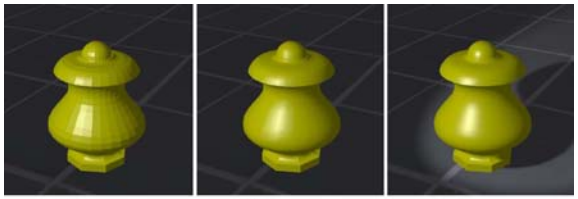
Light and Shade
Illuminating the Model

General schematic overview



Light and Shade
Illuminating the Model

- This requires a huge amount of computational resource
- This is difficult to achieve in practice ("real-time" for VR)
- Simplified models are needed
  - illumination models
  - shading techniques



Flat Gouraud Phong

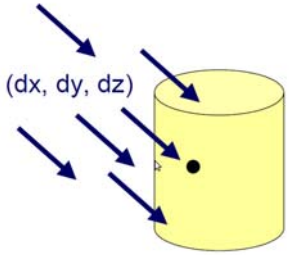
Light Sources

- Most common OpenGL Light Source Models are:
  - Directional Light
  - Point Light
  - Spot Light

Light Sources
Directional Light

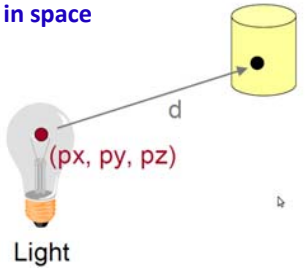
- **Directional light model is by far the simplest to implement**
- **Models a light source that is at infinity**
- **There is no attenuation with distance**
- **Properties:**
  - Intensity: $\boldsymbol{I}$
  - Direction: (dx; dy; dz)

$$I_L = I$$



Light Sources
Omni-Directional Point Light

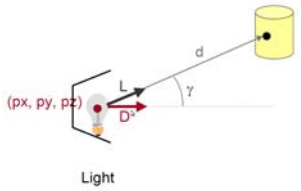
- **Omni-Directional point light model is next complex**
- **Models a light source at a point in space**
- **Attenuates with distance**
- **Properties:**
  - Intensity: $\boldsymbol{I}$
  - Position: (px; py; pz)
  - Attenuation over distance
    - Constant: k_c
    - Linear: k_l
    - Quadratic: k_q



$$I_L = \frac{I}{k_c + k_l d + k_q d^2}$$

Light Sources
Spot Light

- **Spot light directional model is most complex**
- **Computationally expensive**
- **Attenuates radially as well as with distance**
- **Properties:**
  - Intensity: $\boldsymbol{I}$
  - Position: (px; py; pz)
  - Direction: $\boldsymbol{D}$ = (dx; dy; dz)
  - Attenuation over distance
    - Constant: k_c
    - Linear: k_l
    - Quadratic: k_q



$$I_L = \frac{I(D.L)}{k_c + k_l d + k_q d^2}$$

SE3VR11 : Virtual Reality

Professor Paul Sharkey p.m.sharkey@reading.ac.uk

Lecture 9 – 24/03/2016

Light Sources

Directional Light

$$I_L = I$$

Point Light

$$I_L = \frac{I}{k_c + k_l d + k_q d^2}$$

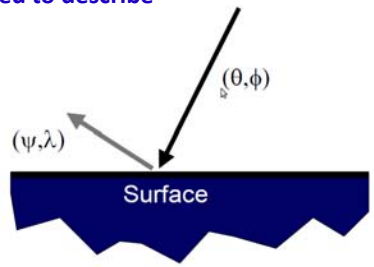
Spot Light

$$I_L = \frac{I(D.L)}{k_c + k_l d + k_q d^2}$$

Modelling Surface Reflectance

Illuminating the Model

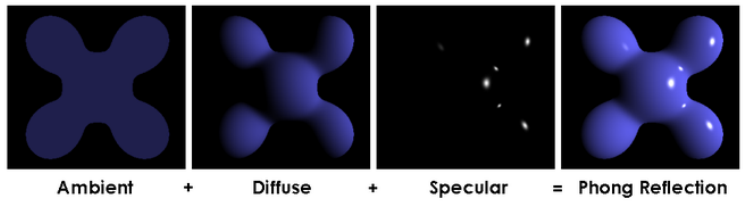
- **A mathematical model for computing illumination at a point on a surface is called a Reflectance Model**
- **The reflectance model is used to describe**
  - the amount of incident light energy
  - arriving from a given direction
  - leaving in a new direction
  - with a new wavelength



Modelling Surface Reflectance

OpenGL Reflectance Model – Phong Reflectance Model

- **The OpenGL model is based on a proposal by BT Phong:**
  - (Emmissive +)
  - Ambient +
  - Diffuse reflection +
  - Specular reflection = **Phong Reflectance**



Modelling Surface Reflectance
OpenGL Reflectance Model – Phong Reflectance Model

- **Emmissive**
  - There are no lights in the scene
  - Each polygon is self-luminous
  - Emissive lighting can have some interesting effects, such as car headlights or neon signs
  - Each polygon has its own colour - constant over its surface
    - Colour is not affected by anything else in the world
    - Colour is not affected by the position or orientation of the polygon in the world
  - Very fast to compute, but not very realistic
  - Position of viewer is not important
  - No 3-D information provided

$$I = I_E$$

Modelling Surface Reflectance
OpenGL Reflectance Model – Phong Reflectance Model

- **Emmissive – Examples**

- **The emmisive lighting parts are the objects themselves**
  (not the reflections – see later)

Modelling Surface Reflectance
OpenGL Reflectance Model – Phong Reflectance Model

- **Ambient**
  - Non-directional light source
    - Simulates light that has been reflected so many times from so many surfaces it appears to come equally from all directions – lacks shadows
  - Intensity is constant over polygon's surface
    - Intensity is not affected by anything else in the world (not affected by position or orientation of polygon)
  - Position of viewer is not important
  - No 3-D information provided
  - **I_A is Ambient Light intensity**
  - **K_A is object's reflection coefficient**

$$I = I_A K_A$$

Modelling Surface Reflectance
OpenGL Reflectance Model – Phong Reflectance Model

- **Ambient – Example**

("accent lighting" = directional/point/spot lighting)

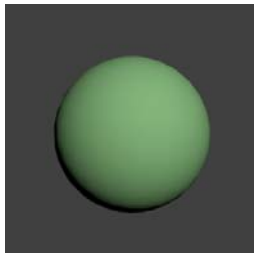
Note that the plinth almost disappears without shadows/directional lights

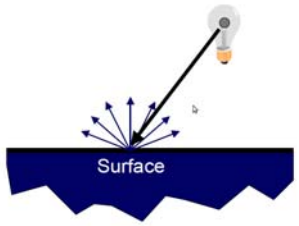
Modelling Surface Reflectance
OpenGL Reflectance Model – Phong Reflectance Model

- **Diffuse Reflection**
  - AKA Lambertian Reflection [JH Lambert, **1760**]
    - Can use either a point light or a directional light
  - Comes from a specific direction
  - Reflects off of dull surfaces
  - Light reflected with **equal intensity in all directions**
  - Brightness depends on the angle between surface normal (**N**) and the direction to the light source (**L**)
  - **Position of viewer is not important**

Modelling Surface Reflectance
OpenGL Reflectance Model – Phong Reflectance Model

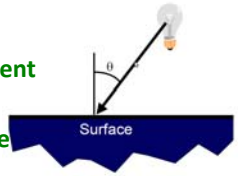
- **Diffuse Reflection**
  - **The surface scatters light equally in all directions**





Modelling Surface Reflectance
OpenGL Reflectance Model – Phong Reflectance Model

- **Diffuse Reflection Mathematical Model**
  - **I_P is the Point Light Intensity**
  - **K_D is object's diffuse reflection coefficient**
  - **N is normalized surface normal**
  - **L is normalized direction to light source**



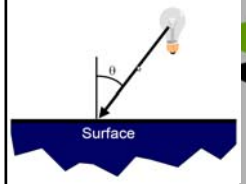
$$I_D = I_P K_D \cos(\theta) = I_P K_D N \bullet L$$

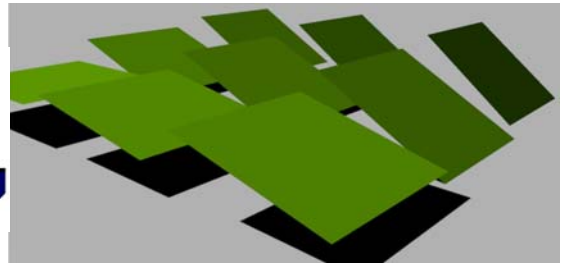
- **The dot product of two unit vectors produces the cosine of the angle between**
- **Can also add an attenuation of the light over distance**

$$I_D = F_{ATT} I_P K_D N \bullet L$$

Modelling Surface Reflectance
OpenGL Reflectance Model – Phong Reflectance Model

- **Diffuse Reflection Mathematical Model**

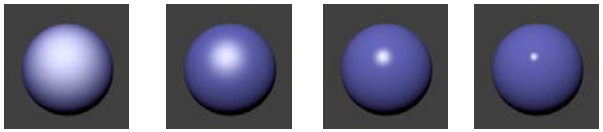




$$I_D = F_{ATT} I_P K_D N \bullet L$$

Modelling Surface Reflectance
OpenGL Reflectance Model – Phong Reflectance Model

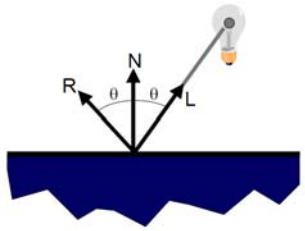
- **Specular Reflection**
  - Reflection off of shiny surfaces – seen as a highlight
  - Shiny metal or plastic has high specular component
  - Chalk or carpet has very low specular component
  - This is called "hardness" in Blender, varies with "n values"



  - Position of the viewer IS important in specular reflection

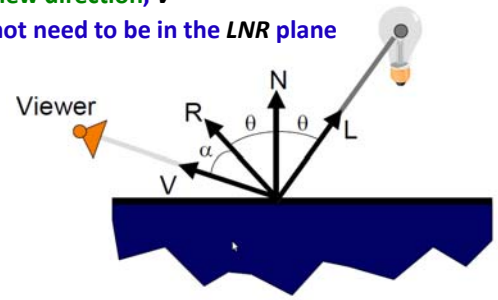
Modelling Surface Reflectance
OpenGL Reflectance Model – Phong Reflectance Model

- **Specular Reflection**
  - **The specular reflection depends only on the surface normal and the light direction**
  - ***N* and *L* will form a plane, *R* will be in that plane by default**



Modelling Surface Reflectance
OpenGL Reflectance Model – Phong Reflectance Model

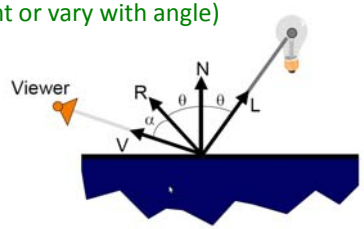
- **Specular Reflection**
  - **However, how that reflection is actually seen depends on the view direction, *V***
  - ***V* does not need to be in the *LNR* plane**



Modelling Surface Reflectance
OpenGL Reflectance Model – Phong Reflectance Model

- **Specular Reflection Mathematical Model**
  - **I_P is the Point Light Intensity**
  - **n_s is object's specular-reflection exponent**
    (higher = sharper fall-off)
  - ***W(θ)* gives specular component of non-specular materials** (can be constant or vary with angle)

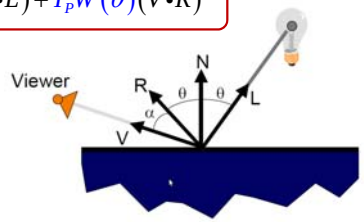
$$I_S = I_P W(\theta)(V \bullet R)^{n_s}$$



Modelling Surface Reflectance
OpenGL Reflectance Model – Phong Reflectance Model

- **Complete Mathematical Model**
- **It is usual to have a combination of Ambient + Diffuse + Specular Reflections**
- **Therefore the model becomes**

$$I_D = K_A I_A + F_{ATT} I_P K_D (N \bullet L) + I_P W(\theta)(V \bullet R)^{n_s}$$



Modelling Surface Reflectance
OpenGL Reflectance Model – Phong Reflectance Model

- **Comparison**

Diffuse + Specular

Ambient + Diffuse + Specular

Modelling Surface Reflectance
OpenGL Reflectance Model – Phong Reflectance Model

- **Multiple Light Sources**

$$I = K_A I_A + \sum_i \left\{ F_{ATT} I_{Pi} K_D (N \bullet L_i) + I_{Pi} W(\theta)(V \bullet R_i)^n \right\}$$

- **exponentially harder as each specular reflection can be treated as a separate light source in itself**

Modelling Surface Reflectance
Calculations

- **Calculating Specular and Diffuse Reflections**
- **To calculate specular and diffuse reflections we need to know:**
  - **The position of the viewpoint on the surface of the object – *P***
  - **The normal direction vector at that viewpoint – *N***
  - **The view direction vector – *V***
  - **The light source direction vector – *L***
    - Direction Vectors should all be in normalised form
    - Dot products then produce the angle cosine alone
  - **Light intensity I_P and coefficients K_D, K_P, n_s and $W(\theta)$**
- Then apply the equations

Modelling Surface Reflectance
Example – Specular Reflection

- Consider the following:
  - A point light source at : P_L = [200, 300, 500]
  - A viewing point at : P_V = [-100, 500, 300]
  - A object surface defined by a plane :

$$4x + 3y + 12z = 1200$$

  - Let n_s = 15 and $W(\theta)$ = 0.7
  - Find the viewed specular light intensity of a point on the plane defined by x = 0 and y = 300 the viewing in terms of the point light intensity I_P

(the x, y points were guestimates to hopefully give a good specular reflection, i.e. chosen near the x,y midpoints between P_L and P_V and taking plane incline into account)

Modelling Surface Reflectance
Example – Specular Reflection

- Consider the plane:
  - Surface defined by a plane : $4x + 3y + 12z = 1200$
  - The point is defined by x = -30 and y = 300
  - Therefore

$$4(-30) + 3(300) + 12z = 1200$$

  - Solving for z, the reflection point of interest is

$$P_R = [-30 \quad 300 \quad 35]$$

  - The normal to the plane by definition is $[4 \quad 3 \quad 12]$
  - Normalised : $N = [4 \quad 3 \quad 12] / \sqrt{169} = [4 \quad 3 \quad 12]/13$

Modelling Surface Reflectance
Example – Specular Reflection

- Consider the plane:
  - The view direction vector (pointing to the viewer) is

$$V = \begin{bmatrix} -100 \\ 500 \\ 300 \end{bmatrix} - \begin{bmatrix} -30 \\ 300 \\ 35 \end{bmatrix} = \begin{bmatrix} -70 \\ 200 \\ 265 \end{bmatrix} \rightarrow \begin{bmatrix} -0.2063 \\ 0.5894 \\ 0.7810 \end{bmatrix}$$

  - The light direction vector (pointing to the light) is

$$L = \begin{bmatrix} 200 \\ 300 \\ 500 \end{bmatrix} - \begin{bmatrix} -30 \\ 300 \\ 35 \end{bmatrix} = \begin{bmatrix} 230 \\ 0 \\ 465 \end{bmatrix} \rightarrow \begin{bmatrix} 0.4434 \\ 0 \\ 0.8963 \end{bmatrix}$$

  - Checking $|V| = 1$ and $|L| = 1$ confirms no errors

Modelling Surface Reflectance
Example – Specular Reflection

- We need the reflection direction vector R:
- The Reflection vector is found using the following formula

$$R = 2(L \bullet N)N - L$$

- On Blackboard Slides:

For explanation of reflection vector calculation, see:

http://tinyurl.com/SE3VR11-reflection

For explanation of vector projection, see KhanAcademy video:

http://tinyurl.com/SE3VR11-reflection2

Modelling Surface Reflectance
Example – Specular Reflection

- We need the reflection direction vector R:
- The Reflection vector is found using the following formula

$$R = 2(L \bullet N)N - L$$

- Calculated as

$$R = 2\left(\begin{bmatrix}0.4434\\0\\0.8963\end{bmatrix}^T \bullet \begin{bmatrix}4\\3\\12\end{bmatrix}/13\right)\begin{bmatrix}4\\3\\12\end{bmatrix}/13 - \begin{bmatrix}0.4434\\0\\0.8963\end{bmatrix} \qquad R = \begin{bmatrix}0.1498\\0.4448\\0.8830\end{bmatrix}$$

- $|\boldsymbol{R}| = 1$ and directions seem plausible

Modelling Surface Reflectance
Example – Specular Reflection

- The specular formula is now applied:

$$I_S = I_P W(\theta)(V \bullet R)^{n_s}$$

- Hence

$$I_S = I_P 0.7\left(\begin{bmatrix}-0.2063\\0.5894\\0.7810\end{bmatrix}^T \bullet \begin{bmatrix}0.1498\\0.4448\\0.8830\end{bmatrix}\right)^{15}$$
$$= 0.2035 I_P$$

- which is ~20% of the original light intensity

Modelling Surface Reflectance
Example – Specular Reflection – SPHERE

- Try a second object defined by the equation

$$(x-50)^2 + (y-400)^2 + (z+100)^2 = 90{,}000$$

- This is a sphere with a radius of ***r* = 300** (as $r^2 = 90{,}000$) and a centre located at

$$P_C = [50 \quad 400 \quad -100]$$

- Using the same coefficients, what is the specular reflection of a point on the sphere which has a normal

$$N = [0 \quad 0 \quad 1]$$

Modelling Surface Reflectance
Example – Specular Reflection – SPHERE

- The reflection point is easier to find and is at the point

$$P_R = [50 \quad 400 \quad +200]$$

- ***r* = 300 and N = [0 0 1]**
- Calculating the light and view vectors as before, show that

$$V = \ldots \rightarrow \begin{bmatrix}-0.7276\\0.4851\\0.4851\end{bmatrix} \qquad L = \ldots \rightarrow \begin{bmatrix}0.4286\\-0.2857\\0.8571\end{bmatrix}$$

Modelling Surface Reflectance
Example – Specular Reflection – SPHERE

- **The reflection vector is** (as before)

$$R = 2(L \bullet N)N - L$$

- **Calculated as**

$$R = 2\left(\begin{bmatrix} 0.4286 \\ -0.2857 \\ 0.8571 \end{bmatrix}^T \bullet \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}\right)\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} - \begin{bmatrix} 0.4286 \\ -0.2857 \\ 0.8571 \end{bmatrix} \qquad R = \begin{bmatrix} -0.4286 \\ 0.2857 \\ 0.8571 \end{bmatrix}$$

- **Note the symmetry between *R* and *L***
- This is as a result of the symmetrical location of the sphere and the specular point of interest (being a reflection in a plane ⊥ to the **z** axis)

Modelling Surface Reflectance
Example – Specular Reflection

- **The specular formula is now applied** (as before):

$$I_S = I_P W(\theta)(V \bullet R)^{n_s}$$

- **Hence**

$$I_S = I_P 0.7\left(\begin{bmatrix} -0.7276 \\ 0.4851 \\ 0.4851 \end{bmatrix}^T \bullet \begin{bmatrix} -0.4286 \\ 0.2857 \\ 0.8571 \end{bmatrix}\right)^{15}$$
$$= 0.0812 I_P$$

- **i.e. ~8% of the point light intensity !**
- **What happens if the radius is smaller e.g. r = 100?**

Modelling Surface Reflectance
OpenGL Reflectance Model – Phong Reflectance Model

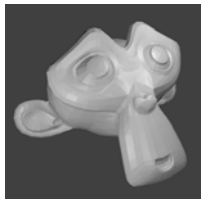
$$I = K_A I_A + \sum_i \left\{ F_{ATT} I_{Pi} K_D (N \bullet L_i) + I_{Pi} W(\theta)(V \bullet R_i)^n \right\}$$

- **Calculating for each and every directional ray can be very computationally expensive**
- **Solution is to use shading techniques**

Shading
Types of Shading

- **Three main types of Shading Models**
  - **Flat Shading**
  - **Gouraud Shading**
  - **Phong Shading**

  (Not to be confused with **Phong Reflectance Model**)

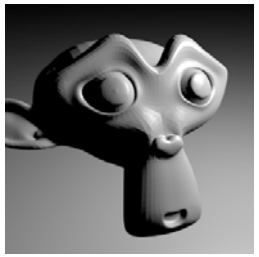

Flat


Gouraud


Phong

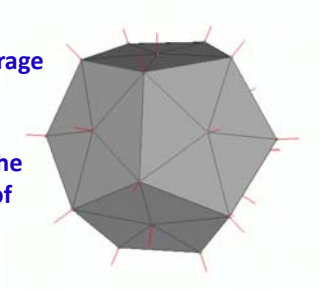
Shading
Flat Shading

- Flat shading computes lighting values for one normal per polygon
- Uses the resulting color for the entire polygon, resulting in a single flat color for every polygon
- Very fast to compute
- One lighting calculation per face
- The individual polygons can be seen
- OK for polyhedral objects
- Gouraud and Phong shading use the concept of "vertex normals" and interpolation



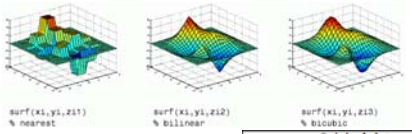
Shading
Vertex Normals

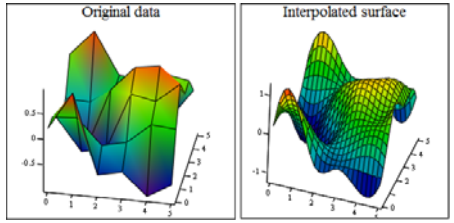
- Vertex Normals
  - The "normal" at a vertex is computed by taking the average of the normals of each connecting polygon
  - An alternative is to weight the normals based on the area of each face



Shading
Interpolation

- Interpolation
  - Shading is achieved by interpolating between values





Gouraud Shading
Interpolation over vertex normals

- Gouraud Shading computes lighting values per vertex and interpolates them over a polygon, i.e. for each fragment on the polygon, the color values from the vertices are interpolated
- Characteristics:
  - will get visible specular highlights on vertices, but you won't see much of the specular highlight if it should be on the inside of a triangle
  - Around the specular highlight, you will see noticeable polygon edges
  - The quality of your specular highlights depends on the number of vertices in your model

Gouraud Shading

Interpolation over vertex normals

- **Characteristics (cont)**
  - There is still only one lighting calculation needed per vertex
  - Specular highlights will appear to jump from vertex to vertex when moving the model, camera or light source

See animated gif at:

https://en.wikipedia.org/wiki/Gouraud_shading

Phong Shading

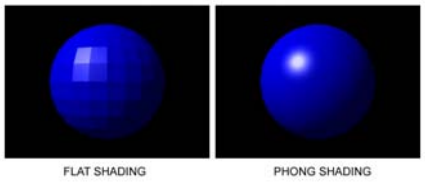
Interpolation over both vertex and face normals

- **Phong Shading interpolates the lighting parameters across the polygon and uses both vertex and face normals in the computation**
- **Characteristics:**
  - You get good fidelity, round, smooth specular highlights that move smoothly along the surface as the camera model or light moves
  - No visible artifacts from the polygon edges
  - More computationally expensive
  - Approximates surface normals for each point of the polygon

Phong Shading

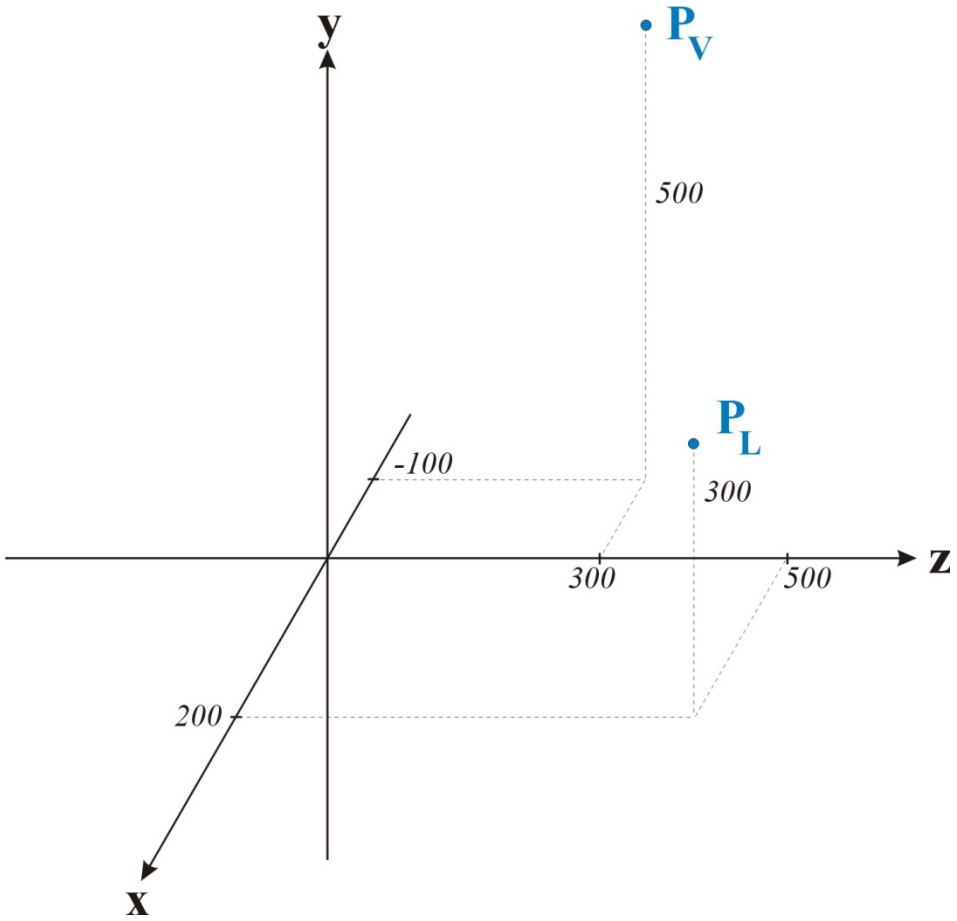
Interpolation over both vertex and face normals

- **Characteristics (cont):**
  - Approximates surface normals for each point of the polygon
  - One lighting calculation per pixel



Additional Diagrams for Examples

View Point and Light Point Locations

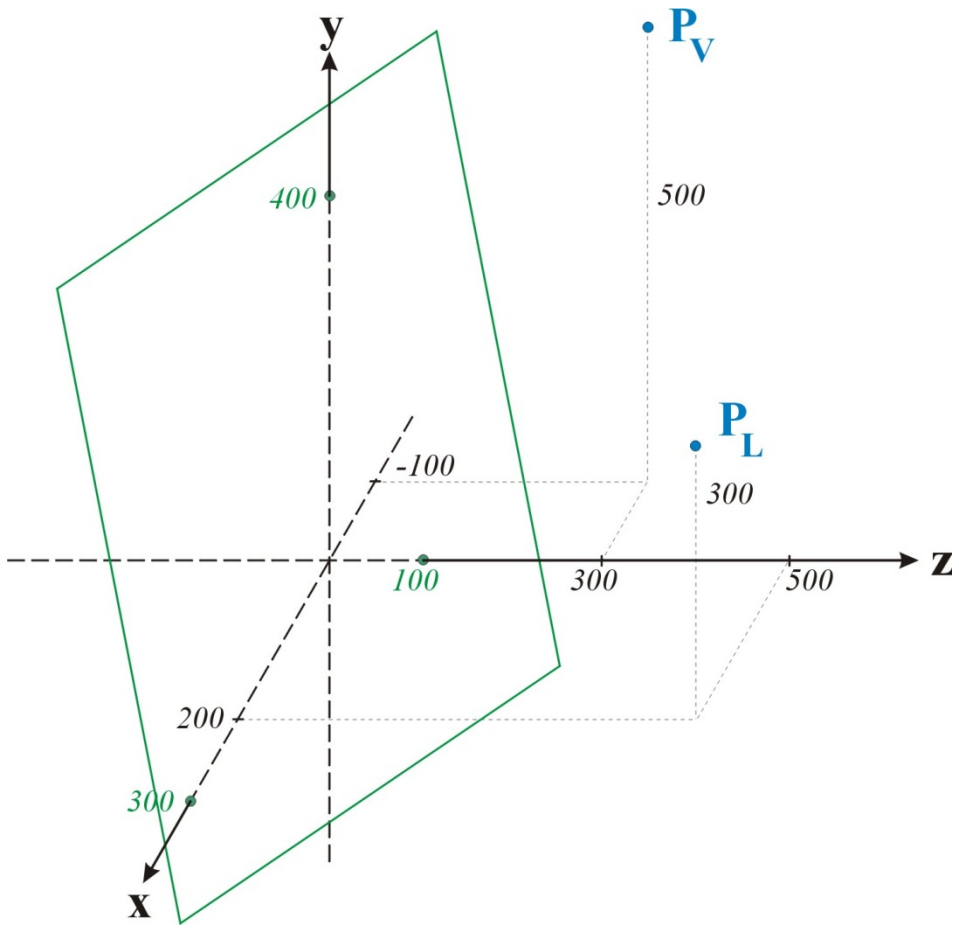


$$P_V = \begin{bmatrix} -100 \\ 500 \\ 300 \end{bmatrix}$$

$$P_L = \begin{bmatrix} 200 \\ 300 \\ 500 \end{bmatrix}$$

Additional Diagrams for Example #1

Planar object (reflection surface)



- **Planar Surface defined as**

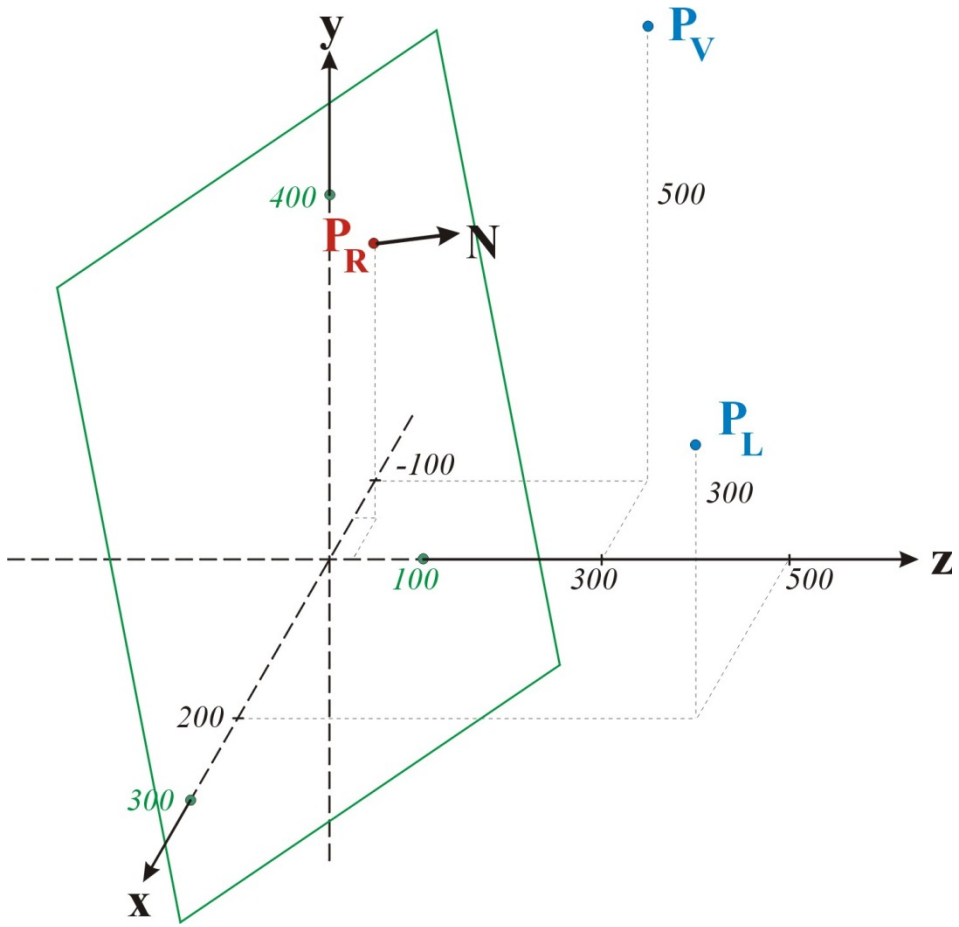
$$4x + 3y + 12z = 1200$$

- **Intercepts can be easily found as**
  - **x = 300**
  - **y = 400**
  - **z = 100**

- i.e. sub 0 in for two to find the third intercept

Additional Diagrams for Example #1

Planar object (reflection surface)



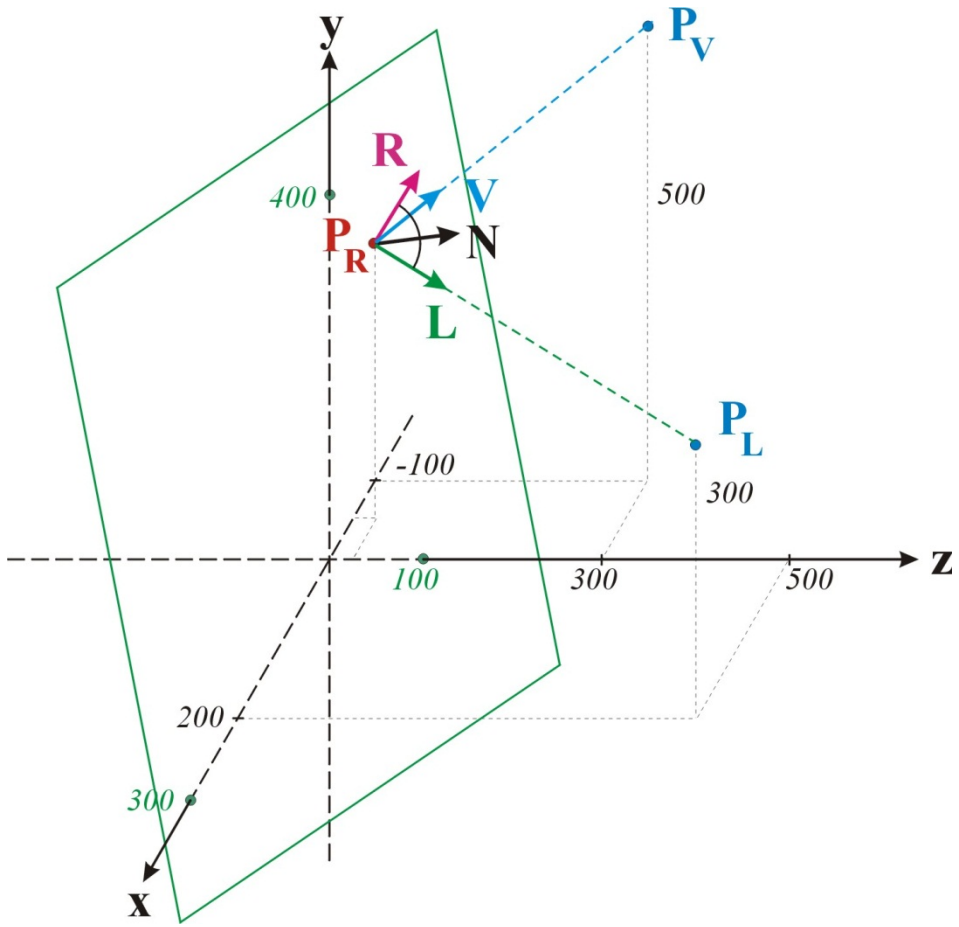
- **With a point defined as**
  - **x = 30**
  - **y = 300**
- **the z value can be found**
  - **z = 35**
- Normal is found from the plane equation

$$N = \begin{bmatrix} 4 & 3 & 12 \end{bmatrix} / 13$$

(i.e. normalised)

Additional Diagrams for Example #1

Planar object (reflection surface)



- **Hence**

$$V = \begin{bmatrix} -0.2063 \\ 0.5894 \\ 0.7810 \end{bmatrix}$$

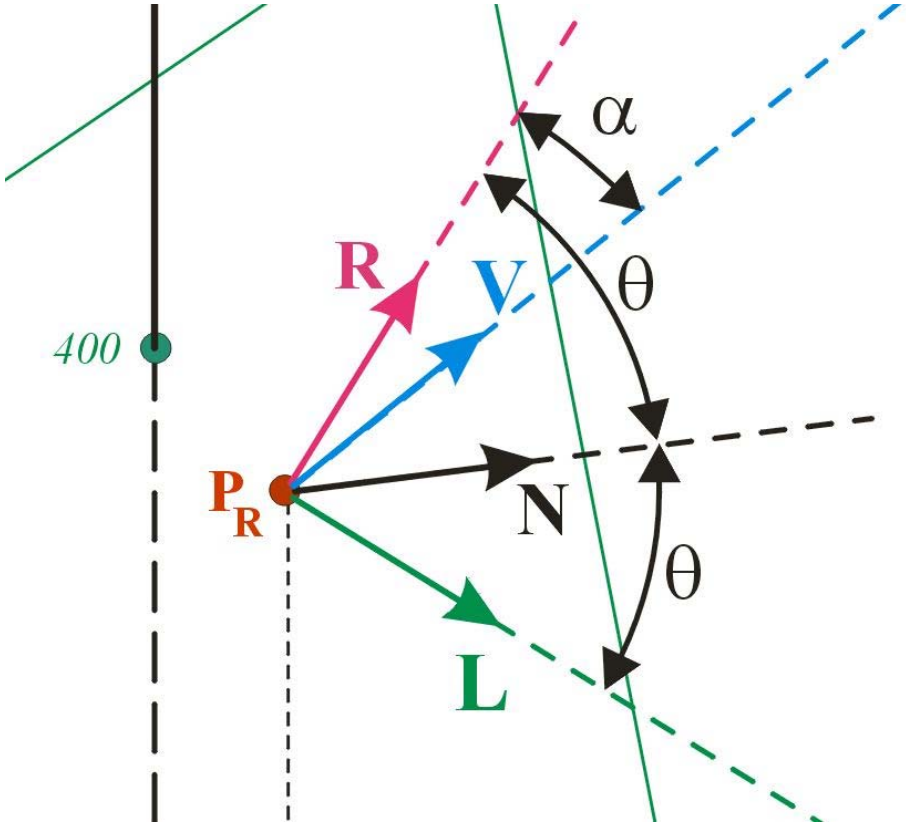
$$L = \begin{bmatrix} 0.4434 \\ 0 \\ 0.8963 \end{bmatrix}$$

$$R = \begin{bmatrix} 0.1498 \\ 0.4448 \\ 0.8830 \end{bmatrix}$$

- ***LNR* coplanar**
- ***V* can be any direction**

Additional Diagrams for Example #1

Planar object (reflection surface)



- **Hence**

$$V = \begin{bmatrix} -0.2063 \\ 0.5894 \\ 0.7810 \end{bmatrix}$$

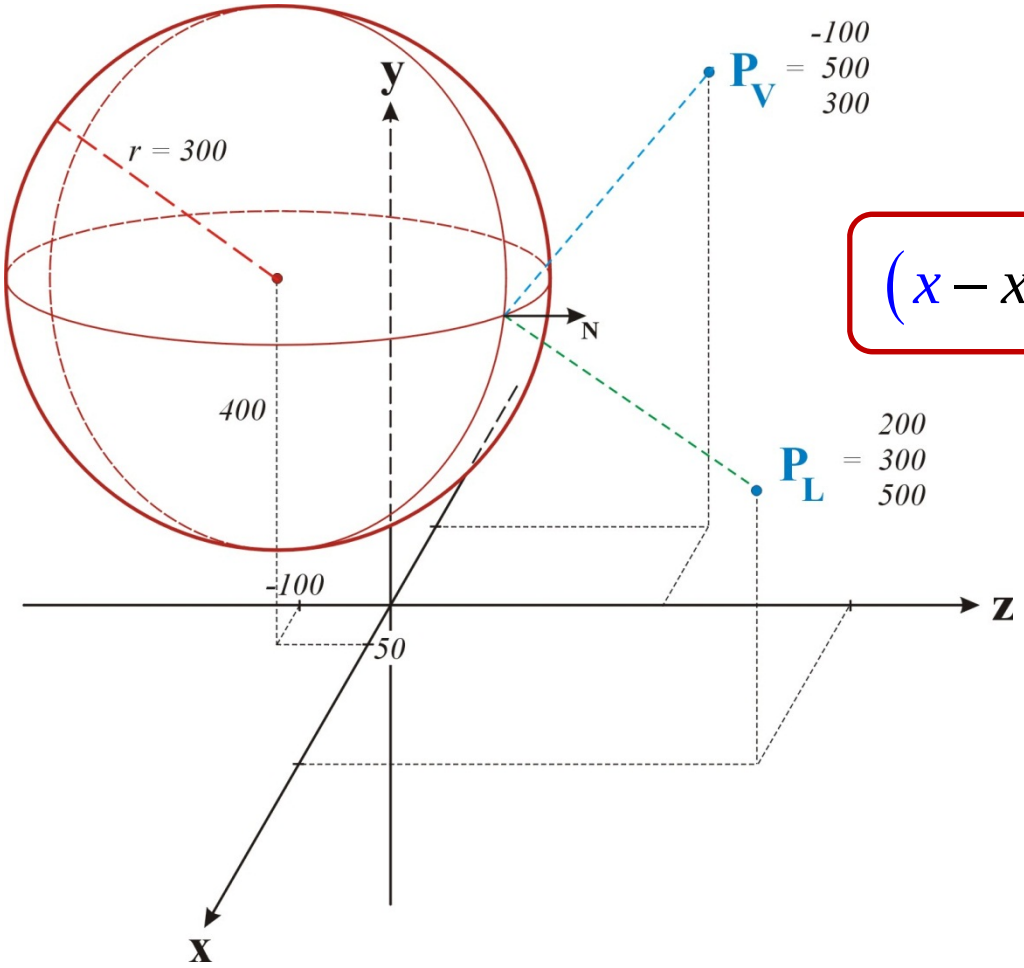
$$L = \begin{bmatrix} 0.4434 \\ 0 \\ 0.8963 \end{bmatrix}$$

$$R = \begin{bmatrix} 0.1498 \\ 0.4448 \\ 0.8830 \end{bmatrix}$$

- ***LNR* coplanar**
- ***V* can be any direction**

Additional Diagrams for Example #2

Sphere object (reflection surface)



- **The sphere is defined by the equation**

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

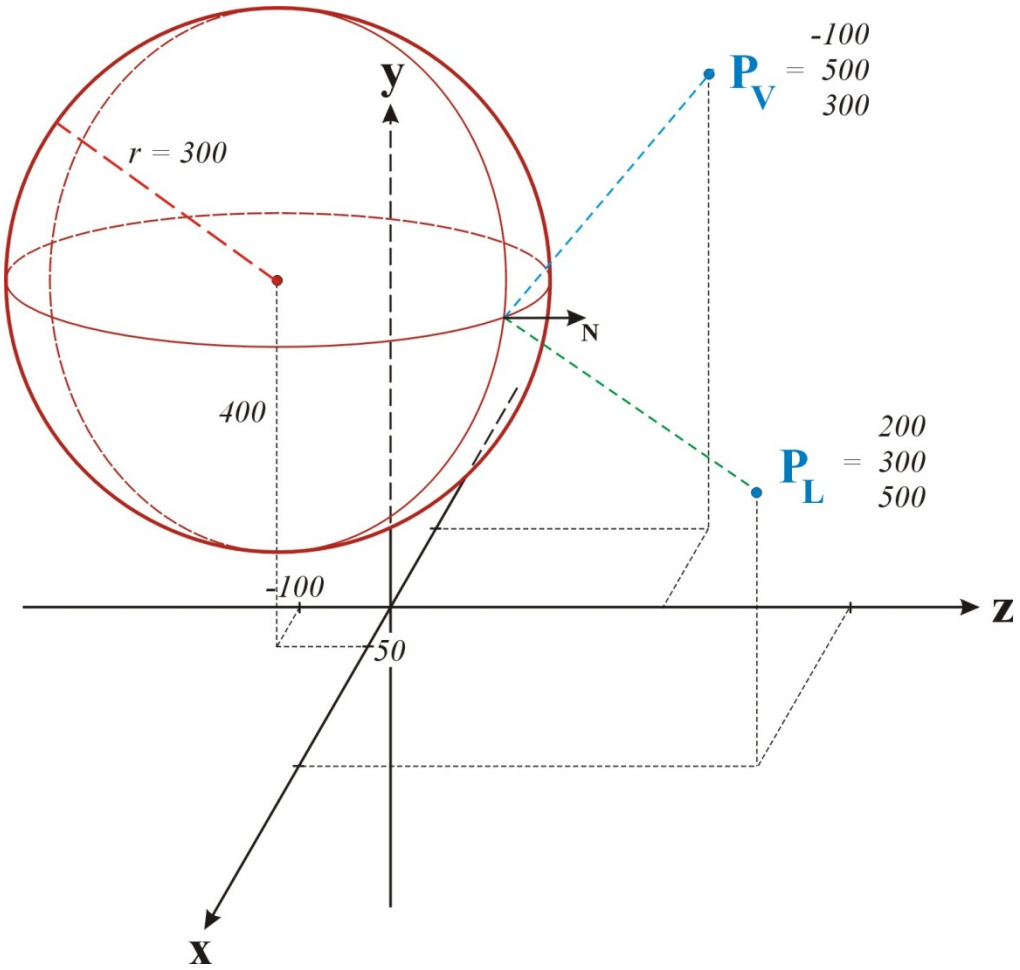
- **with**

$$P_C = \begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix} = \begin{bmatrix} 50 \\ 400 \\ -100 \end{bmatrix}$$

- **and *r* = 300**

Additional Diagrams for Example #2

Sphere object (reflection surface)



- **In this example, the normal vector *N* of interest is specified as**

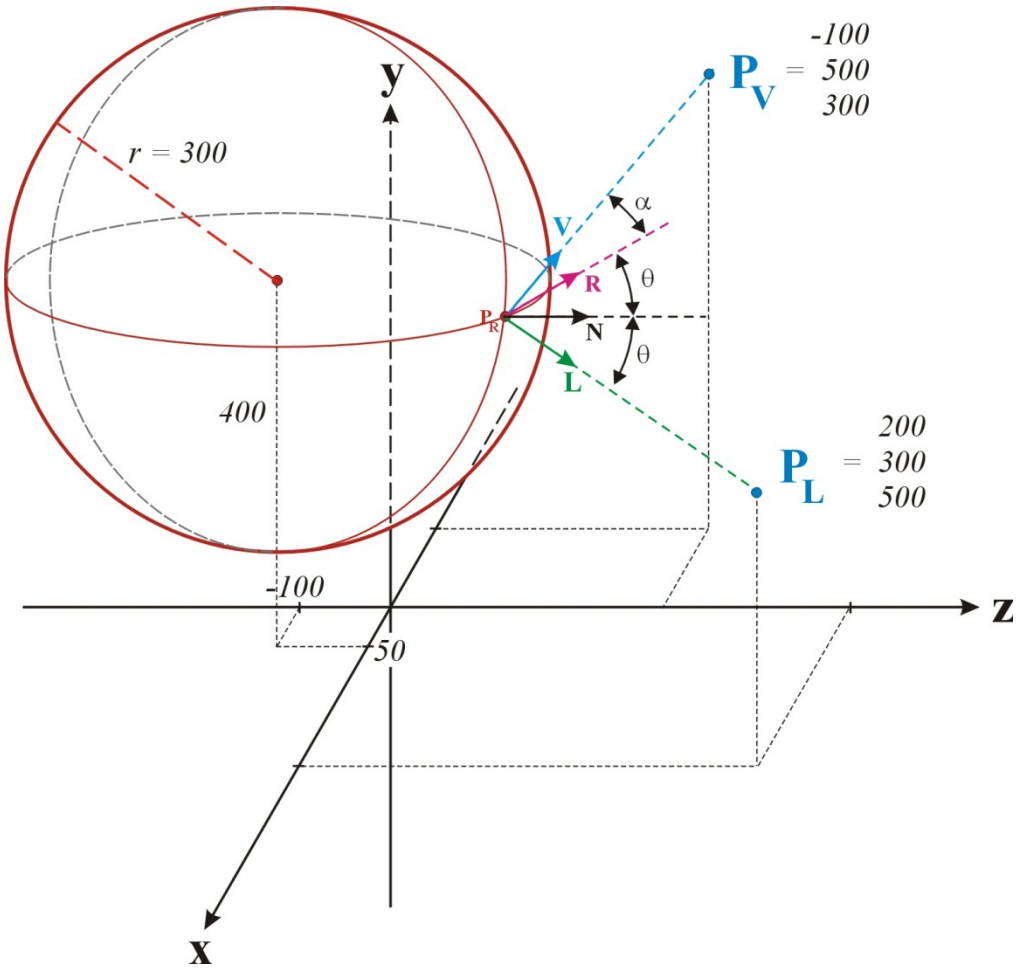
$$N = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$$

- **i.e. pointing parallel to the z direction**
- **This determines the point of interest at**

$$P_R = \begin{bmatrix} 50 & 400 & +200 \end{bmatrix}$$

Additional Diagrams for Example #2

Sphere object (reflection surface)



$$P_R = \begin{bmatrix} 50 & 400 & +200 \end{bmatrix}$$

- As before ***LNR*** are in a plane
- ***V*** can be any direction