COMP 3069
Computer Graphics

Lecture 6:
Projection, Clipping & Hidden Surface Removal

The University of Nottingham

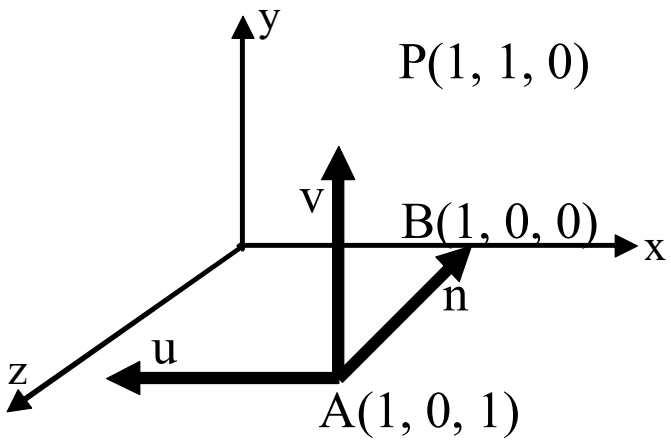
UNITED KINGDOM · CHINA · MALAYSIA

Autumn 2018

Today's Content

- Recap viewing coordinate system
- Projection (orthographic and perspective)
- Removing unwanted objects
  - Clipping (removing objects that are supposed to be outside the 'camera range')
  - Face culling (not showing polygons whose vertices are in clockwise order)
  - Hidden surface removal (not showing polygons that are hidden behind other polygons)

Recap: Viewing/Camera Coordinate System



A viewing coordinate system:

- ***eye*** at A(1,0,1),
- ***looka***t at B(1,0,0),
- ***up*** is y.
- P is in world space.

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

$$P' = R * T * P$$

$$T^{-1} * R^{-1} * P' = P$$

Projection (in View/Camera Space)

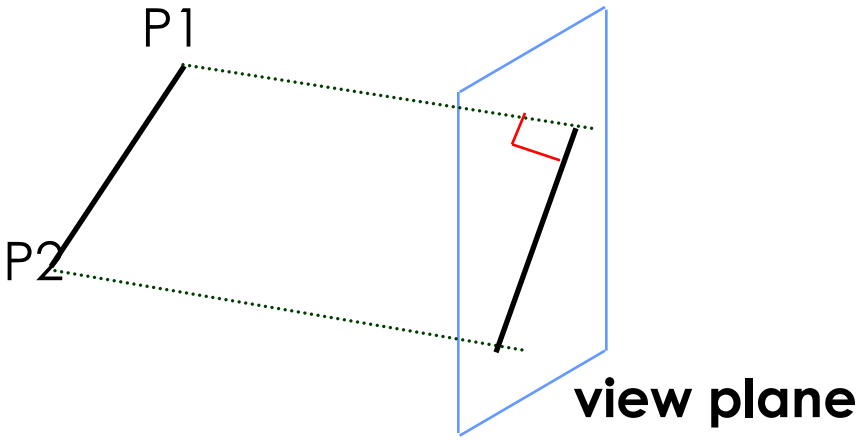
- Orthographic projection
- Perspective projection
- Projection in OpenGL

Projection

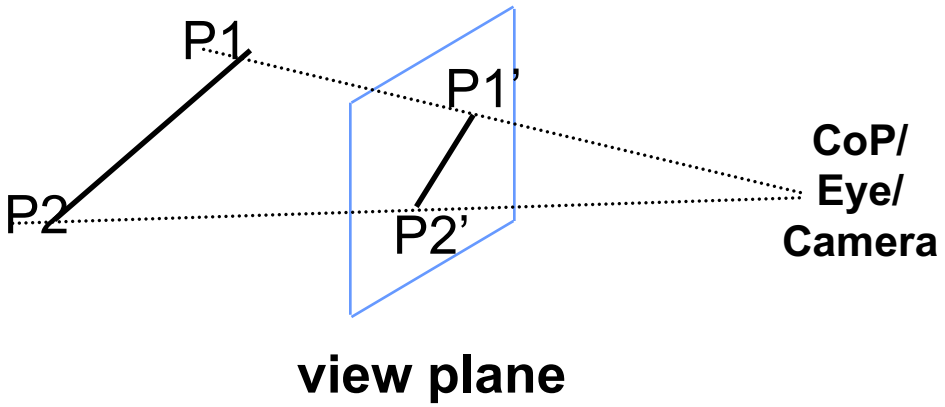
- We know how to transform objects to View space
- We now want to ‘take a photograph’of the objects in the View space – this mapping from 3D to 2D is called projection:
  - **Parallel projection** maps objects directly onto the screen without affecting their relative size, often used in architectural and CAD applications
  - **Perspective projection** gives realistic views, but does not preserve proportions - projections of distant objects are smaller than those near objects of the same size. This is called *foreshortening*.

Two Common Types of Projections

- In **Orthographic projection**, projection rays are parallel and perpendicular to the view plane:

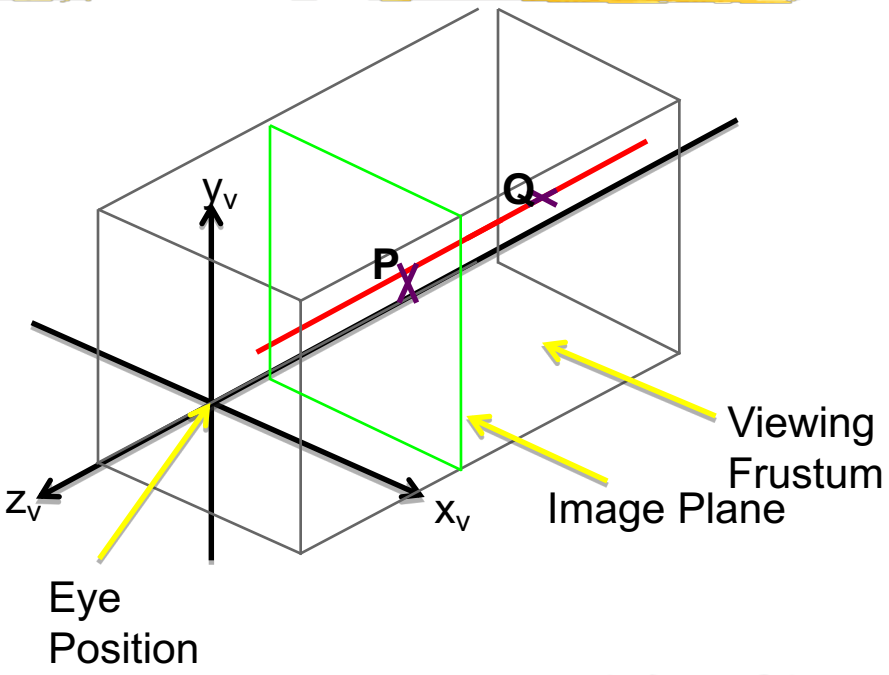


- In **Perspective projection**, projection rays converge at the observer, or Centre of Projection (CoP)



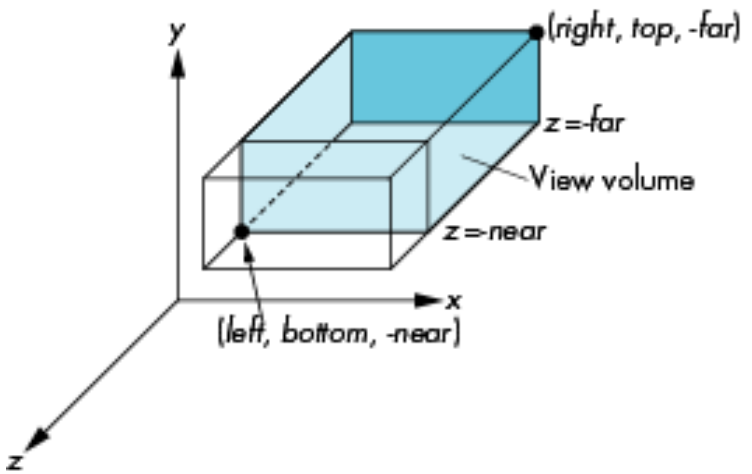
View Volume/Frustum – Defining Regions in Which Objects are Visible

- **Orthographic** view volume is a rectangular block defined by near, far, left, right, top and bottom planes



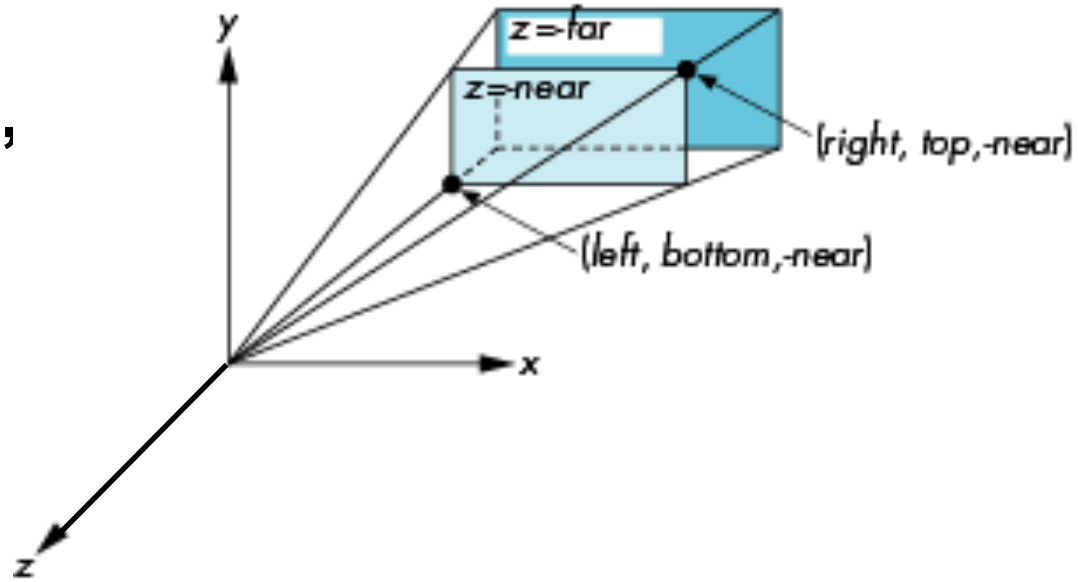
OpenGL:

glOrtho(left, right, bottom, top, near, far)



View Volume/Frustum – Defining Regions in Which Objects are Visible

- **Perspective** view volume is defined by **field of view** or **near, far, left, right**, **top**, and **bottom** planes



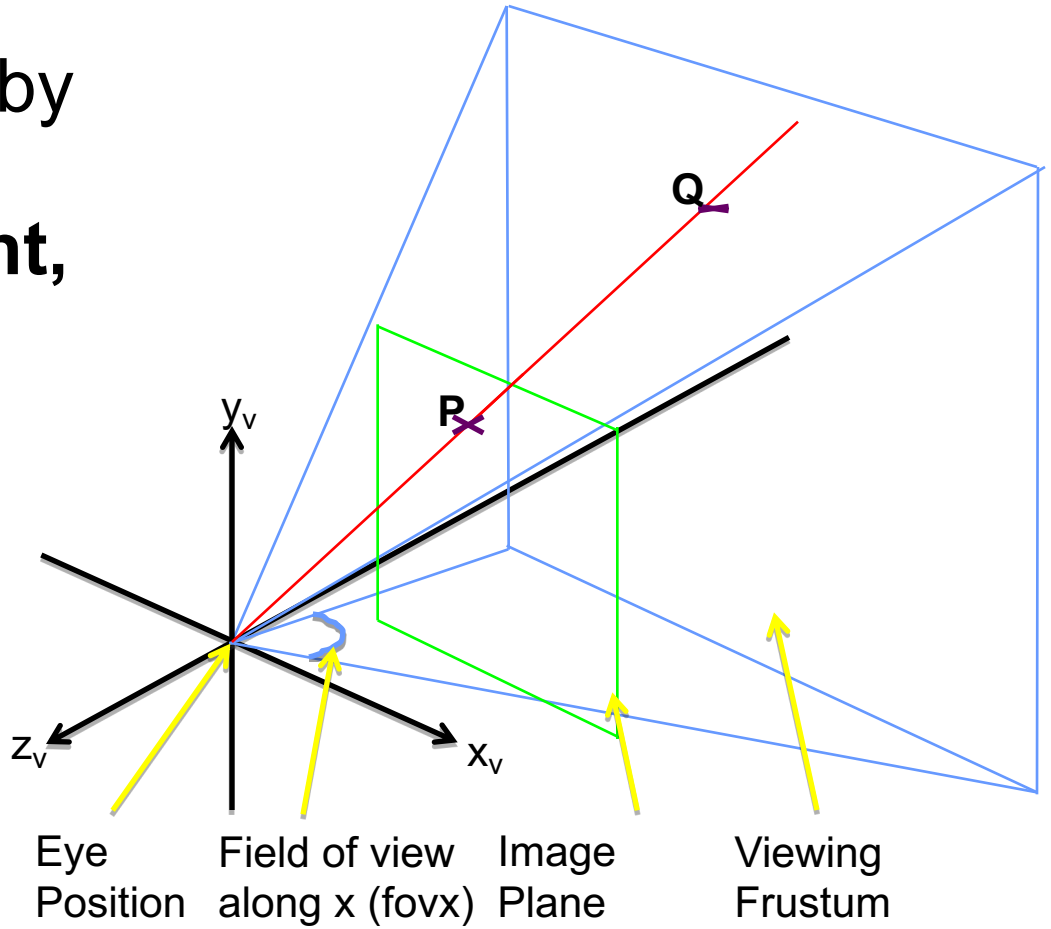
OpenGL:

glFrustum(left, right, bottom, top, near, far)

View Volume/Frustum – Defining Regions in Which Objects are Visible

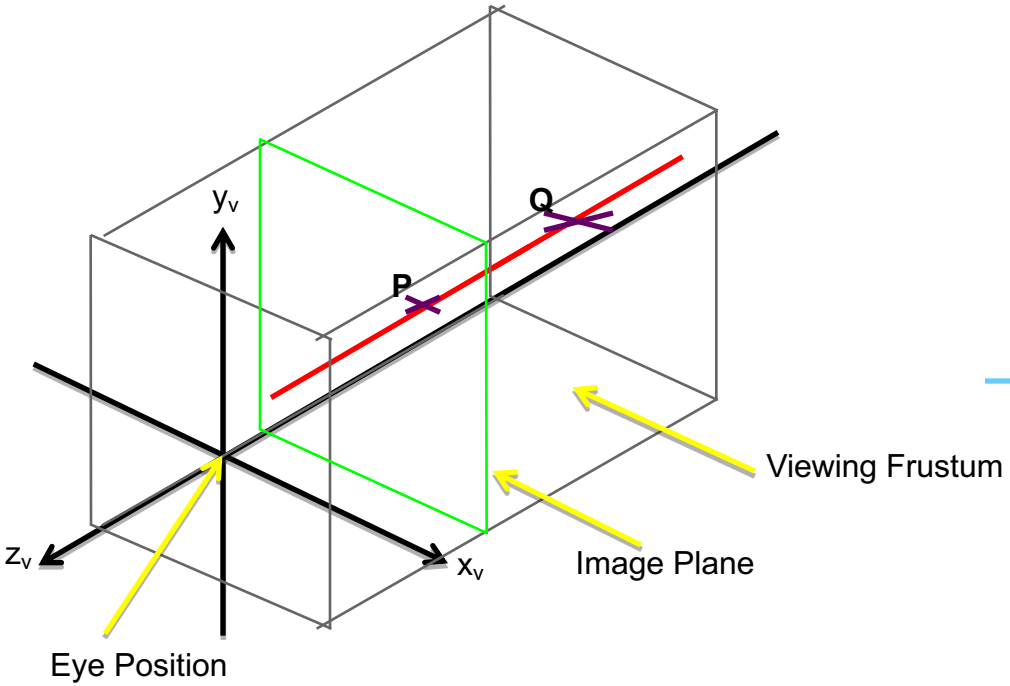
- **Perspective** view volume is defined by **field of view** or **near, far, left, right, top**, and **bottom** planes

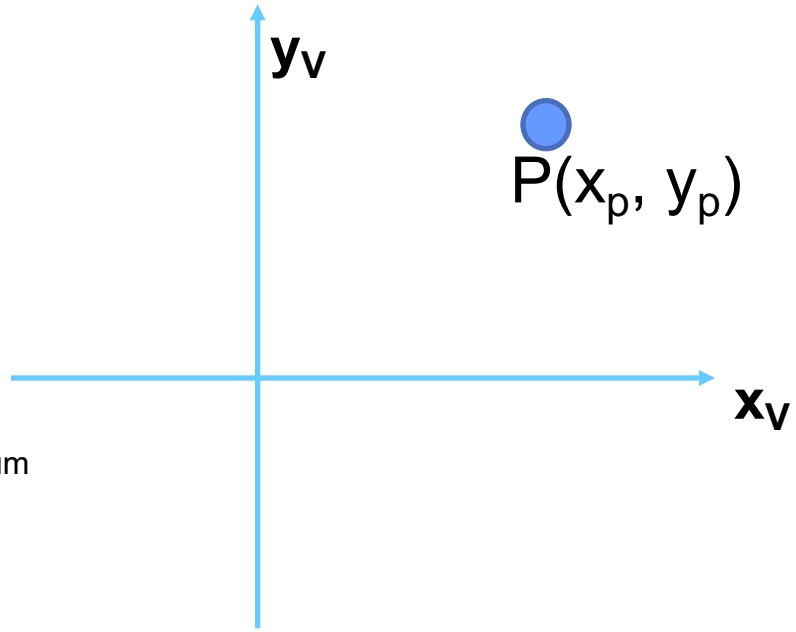
OpenGL:
gluPerspective(fovy, aspect, near, far);



Orthographic Projection

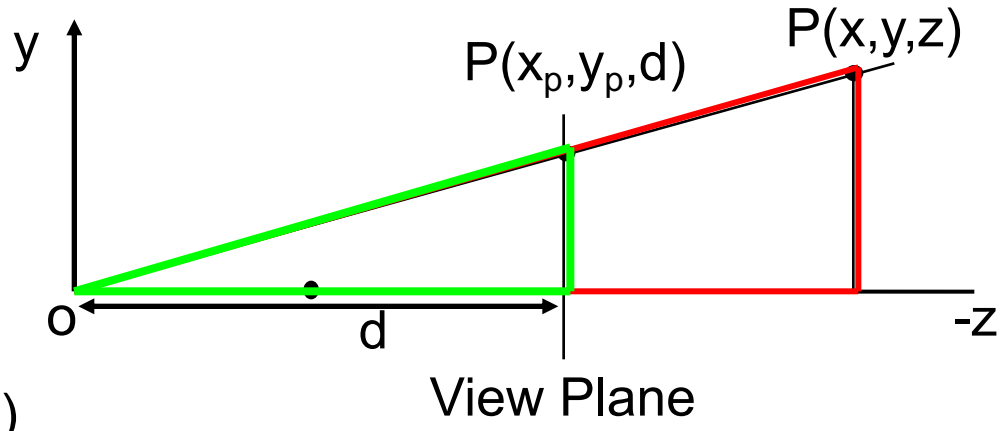
Looking at **z** axis we can see point P on the image plane, which is the projection of a 3D point Q in view space, and $x_P = x_Q$, $y_P = y_Q$ (which means P and Q have the same x, y values)

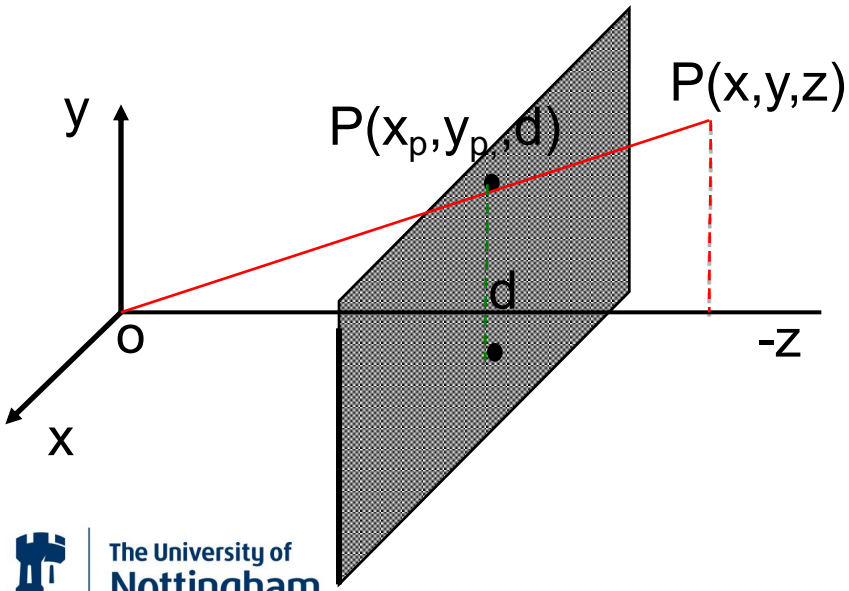




Perspective Transformation Matrix

Looking at **X** axis, we see similar triangles:

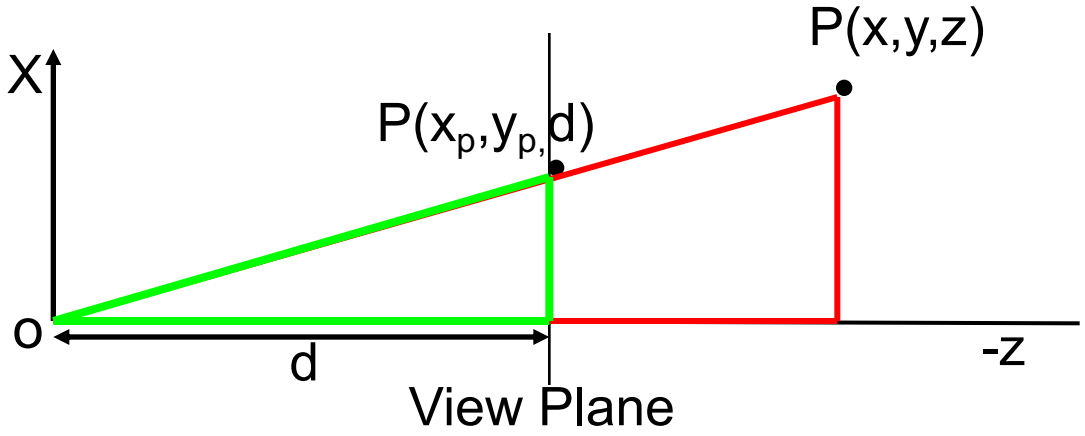


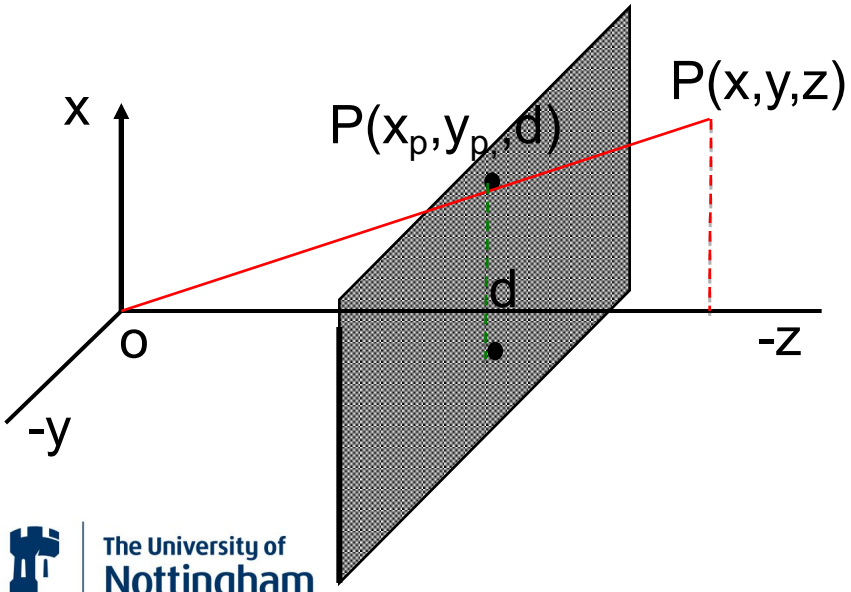


$$\frac{y_p}{d} = \frac{y}{z} \Rightarrow y_p = \frac{d \times y}{z}$$

Perspective Transformation Matrix

- Looking at **-Y** axis, we see similar triangles:





$$\frac{x_p}{d} = \frac{x}{z} \Rightarrow x_p = \frac{d \times x}{z}$$

Perspective Transformation Matrix

$$\frac{x_p}{d} = \frac{x}{z} => x_p z = dx => x_p = \frac{d \times x}{z}$$

$$\frac{y_p}{d} = \frac{y}{z} => y_p z = dy => y_p = \frac{d \times y}{z}$$

See next slide …

So in matrix form:

$$\begin{bmatrix} x_p \\ y_p \\ d \end{bmatrix} \equiv \begin{bmatrix} d \times x/z \\ d \times y/z \\ d \\ 1 \end{bmatrix} \equiv \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

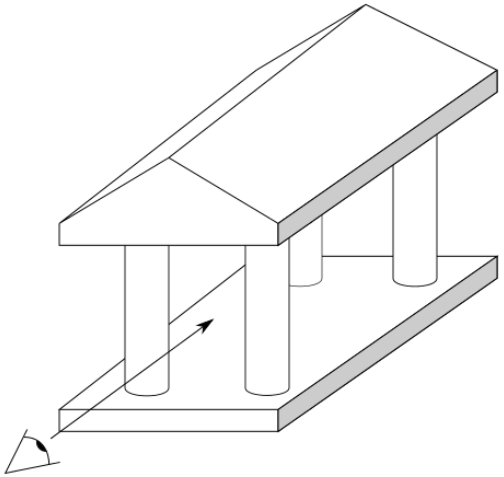
Extending Homogeneous Coordinates to Handle Perspective Projection

- When we instroduced homogeneous coordinates, we represent a 3-dimensional point (x, y, z) by the point (x, y, z, 1) in 4D.
- Suppose now we represent (x, y, z) by (wx, wy, wz, w) where w≠0.

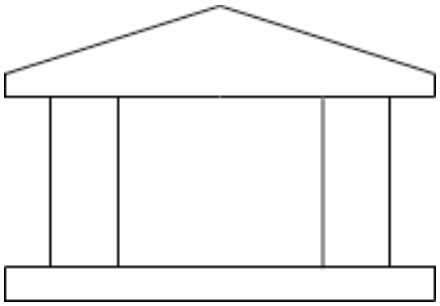
$$\begin{bmatrix} wx \\ wy \\ wz \\ w \end{bmatrix} \equiv \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- As long as w≠0, we can recover the 3-dimensional point from its 4D representation by dividing the first three components by w.
- In such a generalised represenation, points in 3D becomes lines through the origin in 4D.
- With such a representation, we can then use the 4x4 perspective transformation matrix (*see previous slide…*) to represent perspective projections.

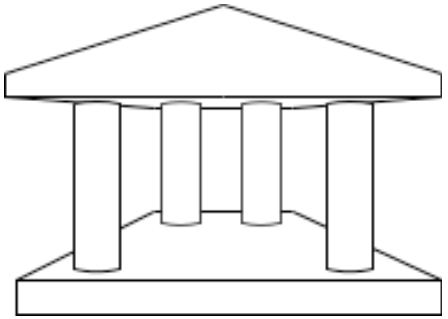
Different Types of Projections



Viewing direction



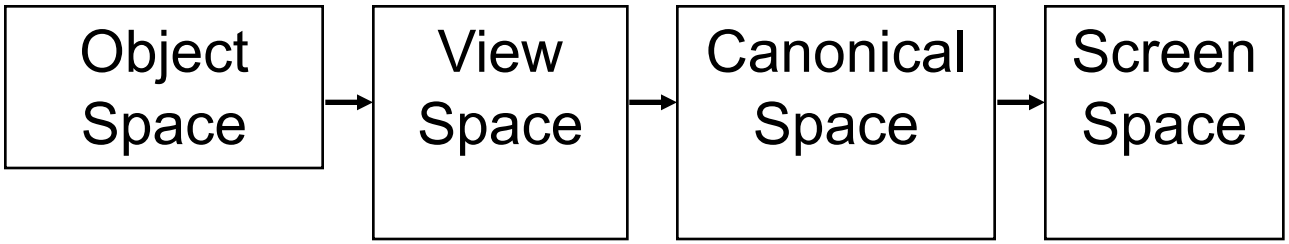
(A) Orthographic Projection



(B) Perspective Projection

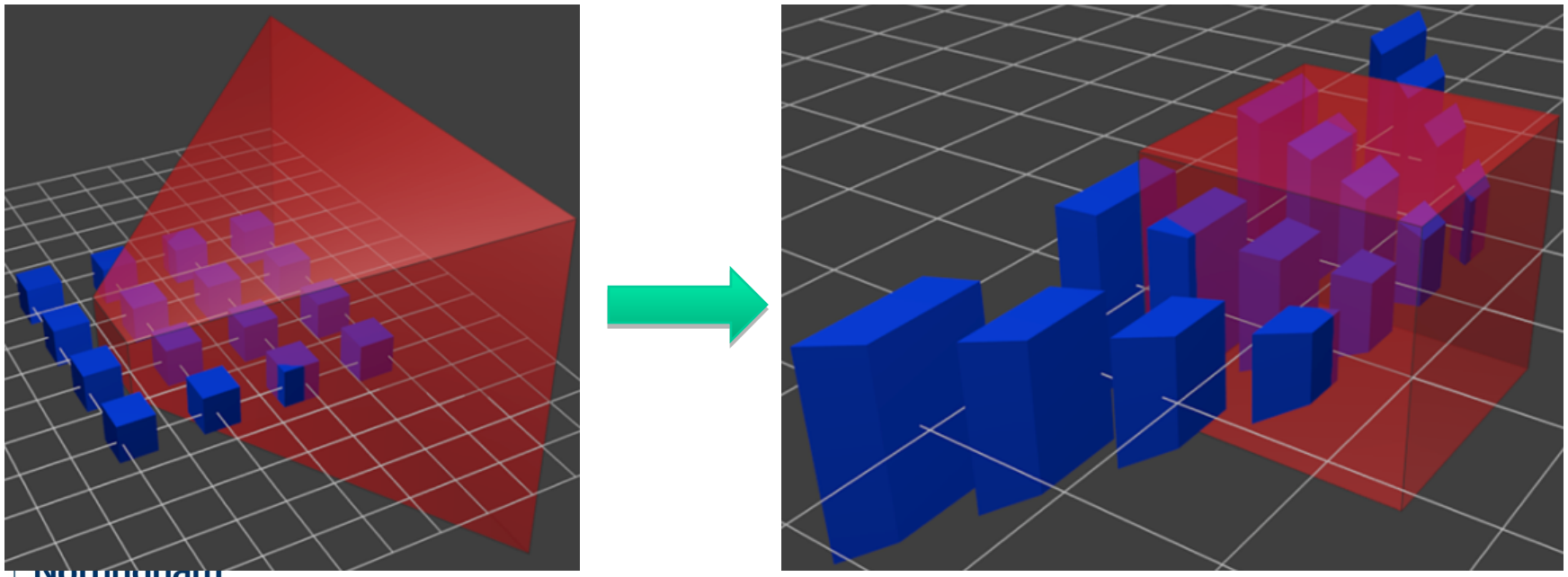
Projection in OpenGL

- An OpenGL projection matrix takes points in view space and converts them to points in ***normalised device space*** or ***canonical space.***
- The canonical space is a 2x2x2 cube, centered at the coordinate origin of the view space, with X, Y, Z each ranging from -1 to 1.
- Thus only parallel orthographic projection is needed to obtain 2D images of a 3D scene.



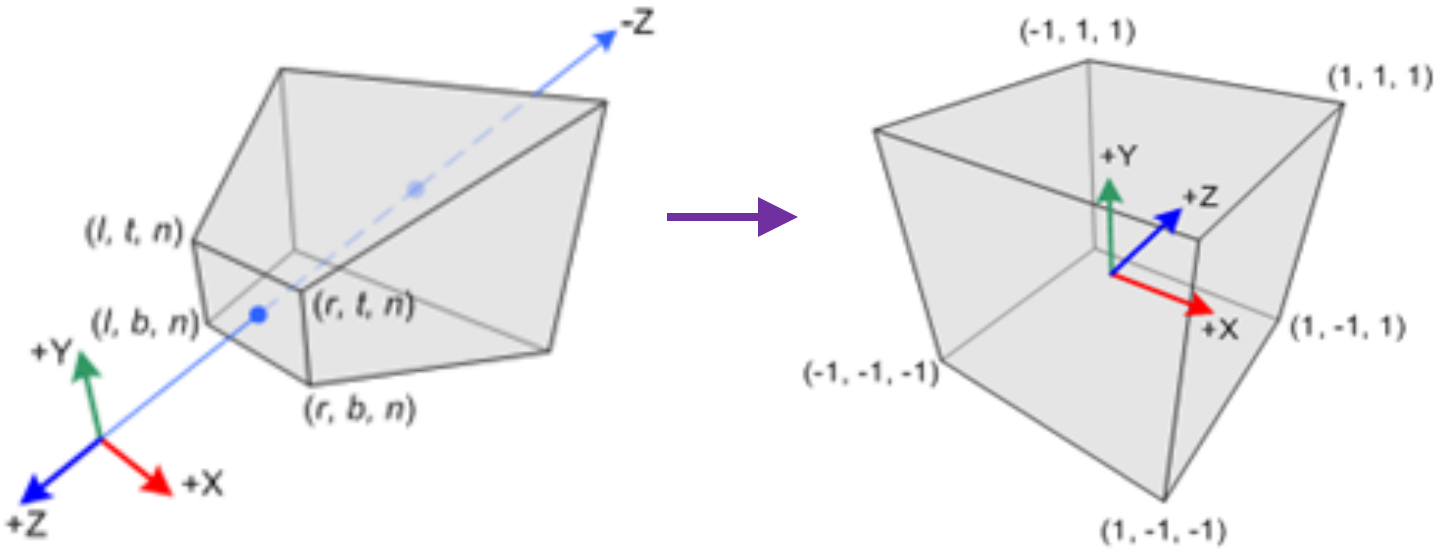
Projection in OpenGL

- OpenGL transforms all objects to a canonical space,
  - The perspective viewing frustum is transformed to the canonical space, and
  - All the objects will go through the same transformations



OpenGL Perspective Matrix

- OpenGL perspective matrix transforms view frustum in view/camera/eye space to canonical (normalised) space (a 2x2x2 cube):

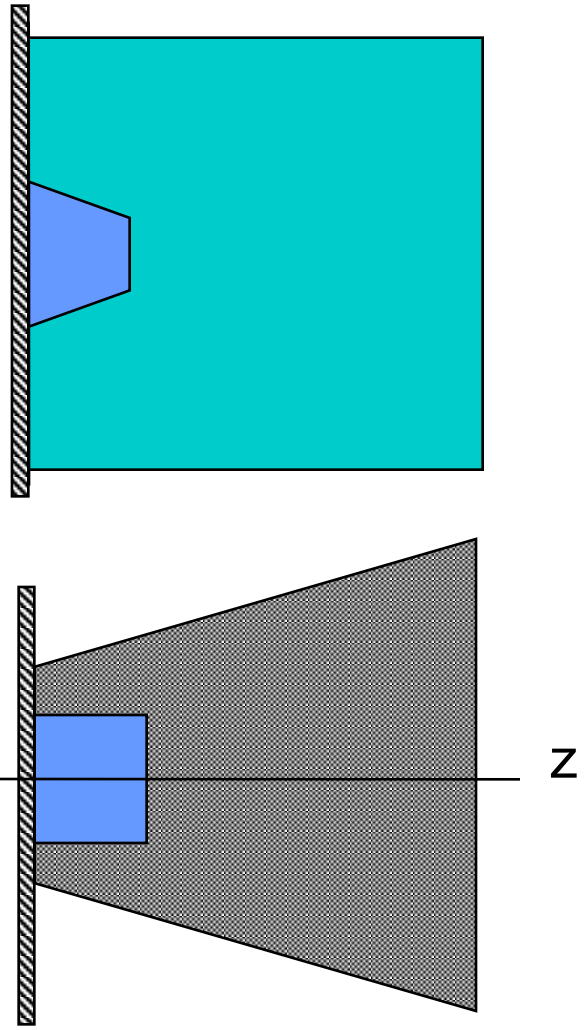


OpenGL Perspective Matrix

- Perspective matrix warps objects so that when viewed under orthographic projection, the warped objects appear the same as the original objects viewed under perspective projection

$$\begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

where l=left,r=right,t=top,b=bottom,n=near,f=far.



- glMatrixMode(GL_PROJECTION)

Projection in OpenGL

- glMatrixMode(GL_PROJECTION);
- glOrtho (left, right, bottom, top, near, far);
- gluPerspective(fovy, aspect, near, far);
- glFrustum(left, right, bottom, top, near, far);

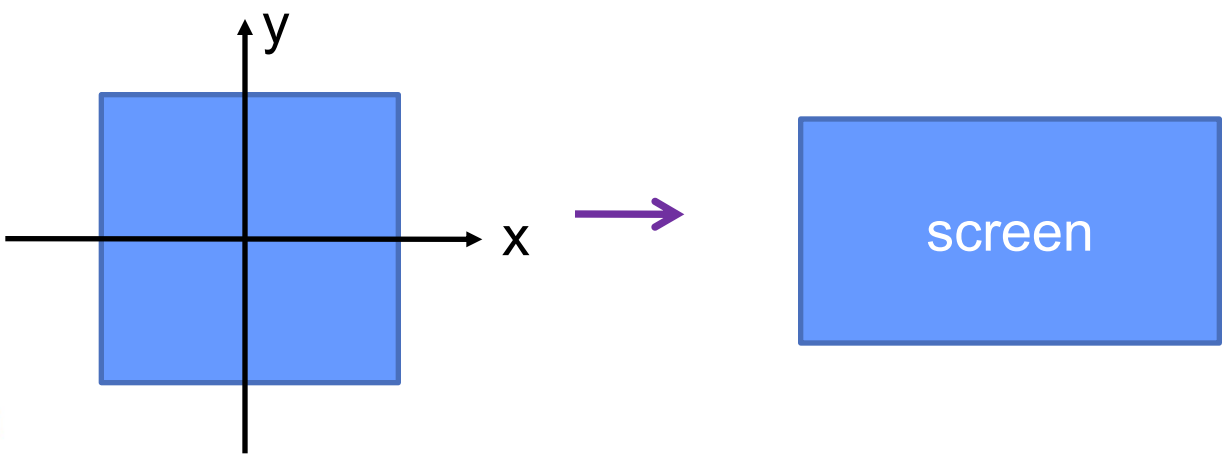
  - glOrtho(-windowWidth/2, windowWidth/2, -windowHeight/2, windowHeight/2, 1, 1000);

  - gluPerspective(60.0, windowWidth/windowHeight, 1, 1000);

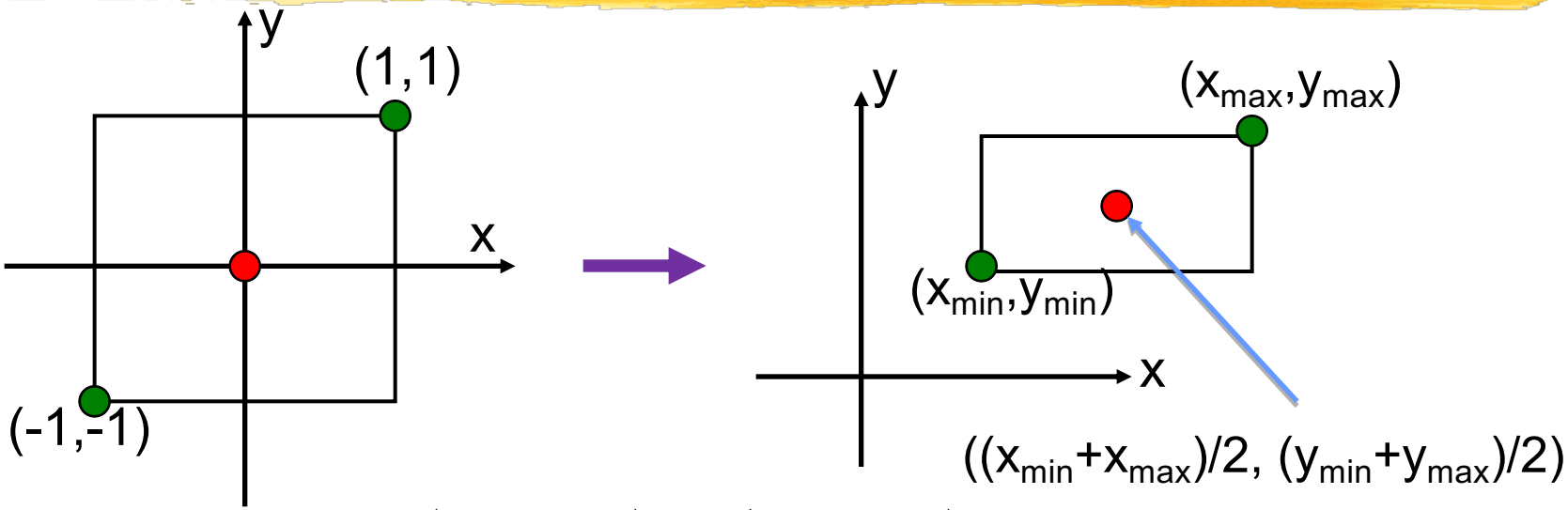
Canonical to Screen Transform

- The last step is to position the 2D image on the display screen by calling **glViewport()** in OpenGL, to transform canonical coordinates to display coordinates

- Parameters of **glViewport()** describe position of screen space within the window and the width and height of the screen, measured in pixels on the screen



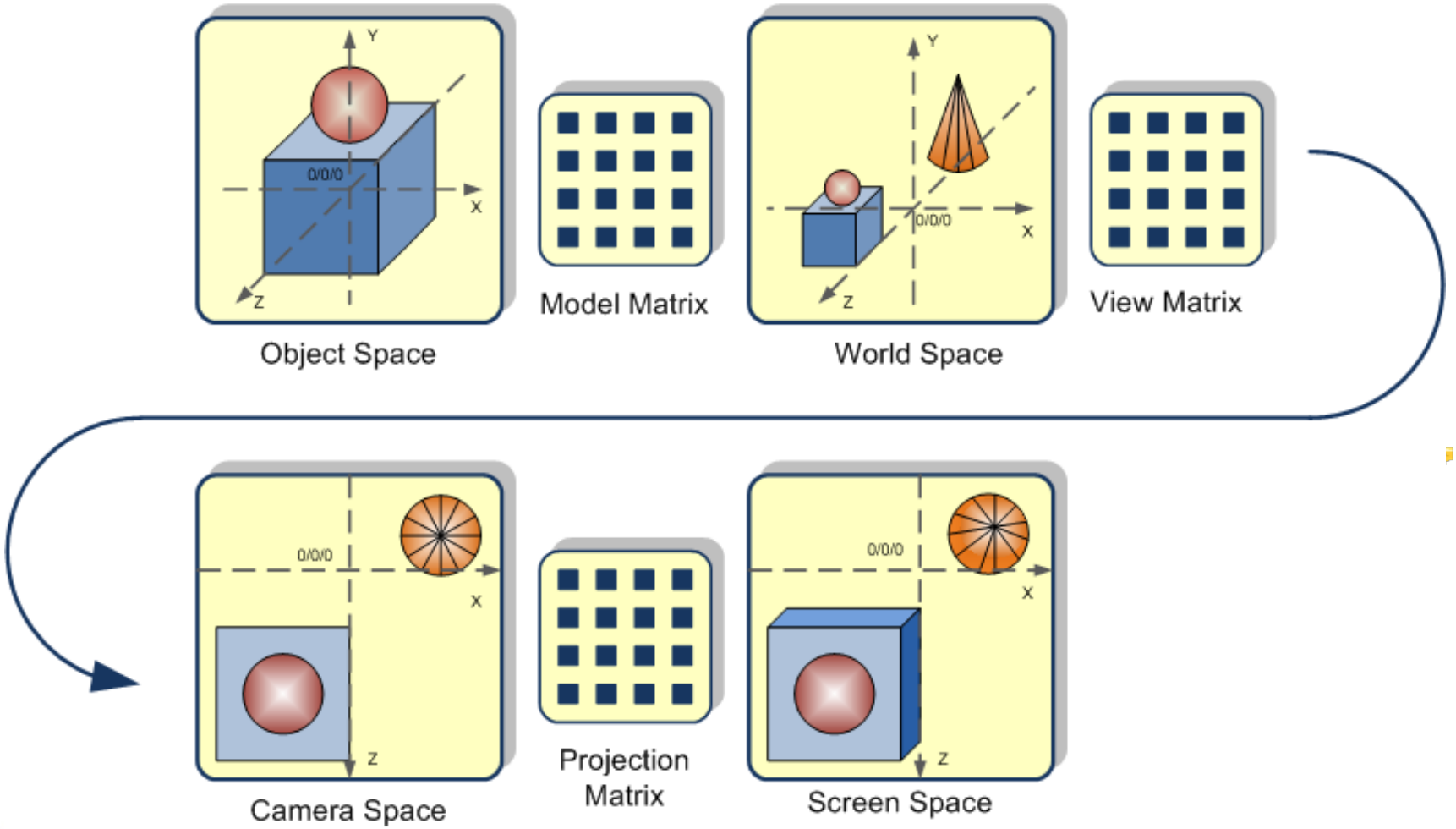
Canonical to Screen Transform



$$\begin{bmatrix} x_{pixel} \\ y_{pixel} \\ z_{pixel} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & (x_{max}+x_{min})/2 \\ 0 & 1 & 0 & (y_{max}+y_{min})/2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} (x_{max}-x_{min})/2 & 0 & 0 & 0 \\ 0 & (y_{max}-y_{min})/2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{canonical} \\ y_{canonical} \\ z_{canonical} \\ 1 \end{bmatrix}$$

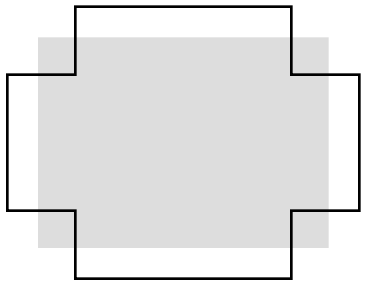
$$= \begin{bmatrix} (x_{max}-x_{min})/2 & 0 & 0 & (x_{max}+x_{min})/2 \\ 0 & (y_{max}-y_{min})/2 & 0 & (y_{max}+y_{min})/2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{canonical} \\ y_{canonical} \\ z_{canonical} \\ 1 \end{bmatrix}$$

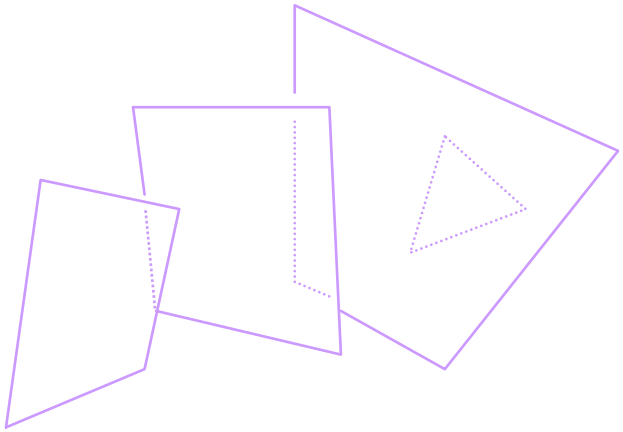
Summary:



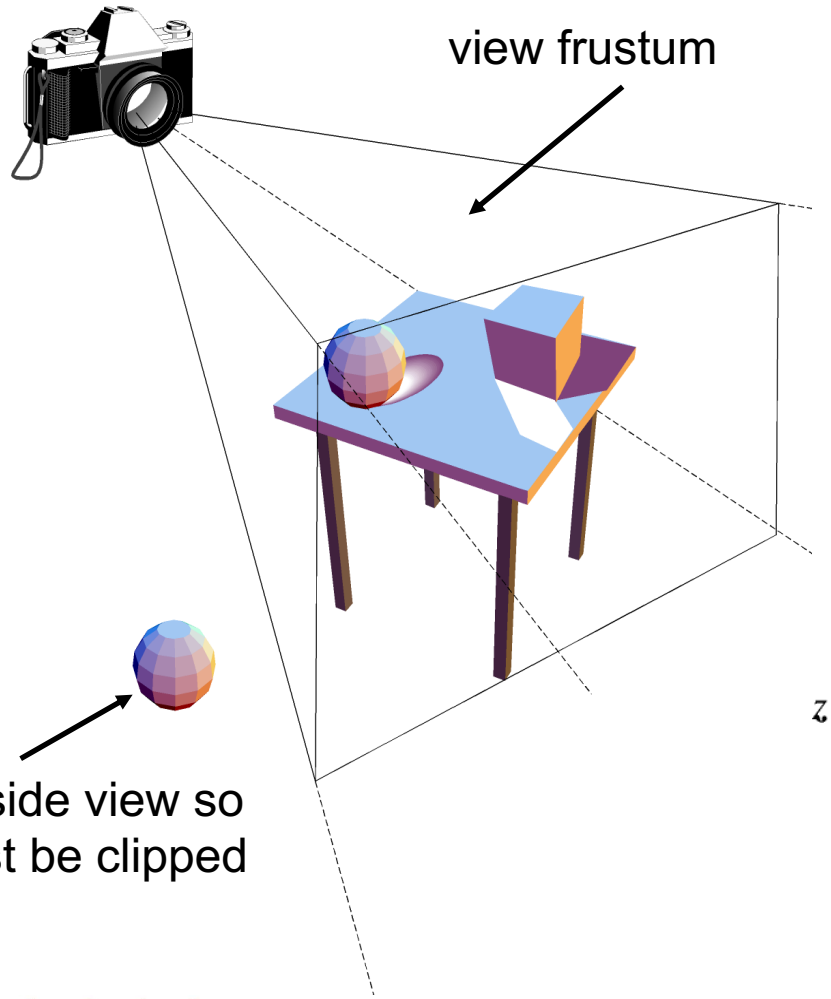
Removing Invisible Surfaces

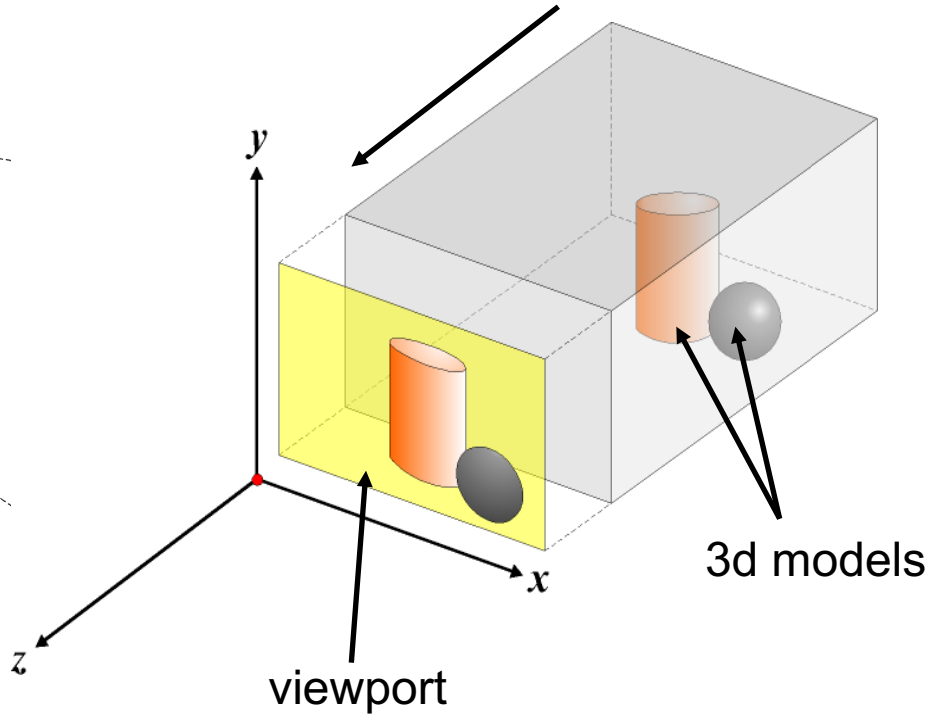
- We expect the rendered image to show only the objects within the camera range (inside viewing frustum), and are not blocked by other objects
- **Clipping** allows you to remove objects or polygons outside the camera range
- **Hidden surface removal** allows you to remove objects or polygons that are blocked by other objects





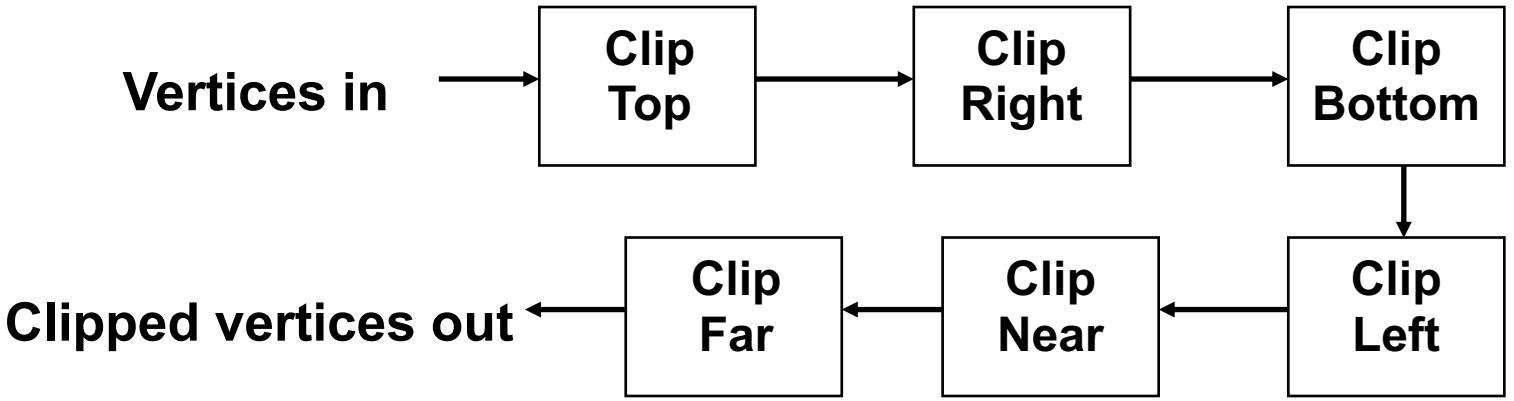
Removing Outside View Surfaces



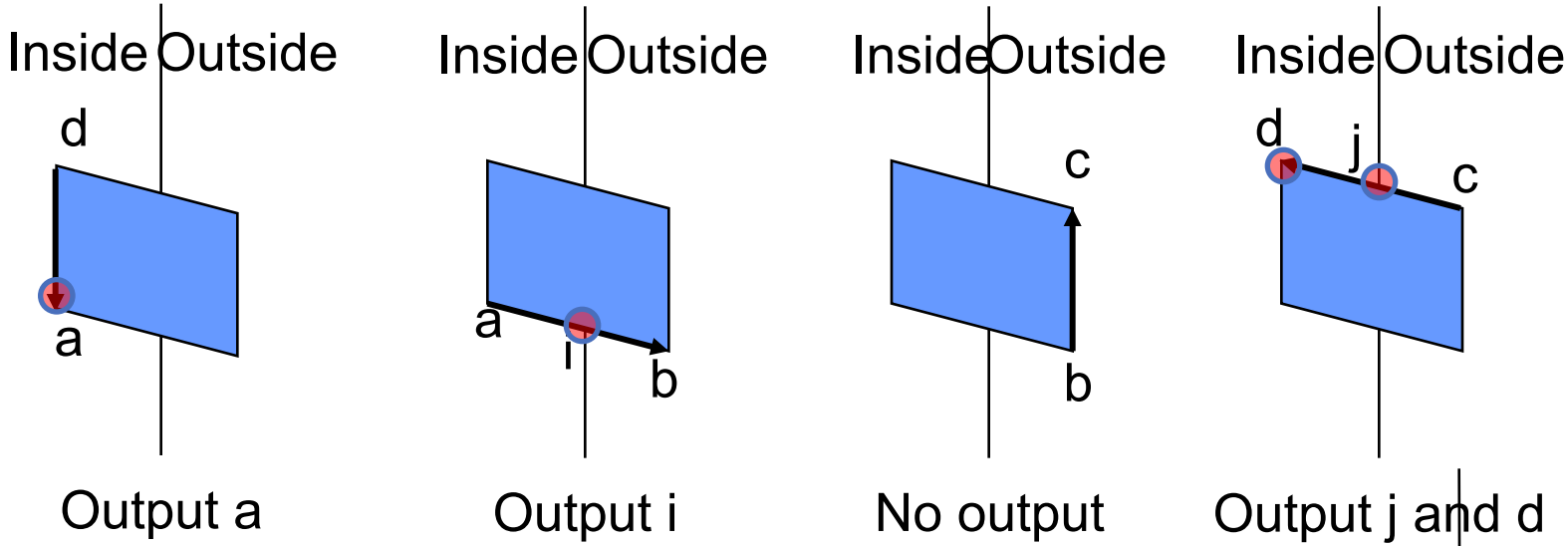


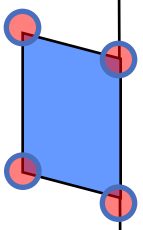
Removing Outside View Surfaces : Sutherland-Hodgman Clipping

- This concerns with polygons - is a polygon (as a list of vertices) inside clip region?
- Clip the polygon against each clip plane
- Rewrite the polygon one vertex at a time – the rewritten polygon will be the clipped polygon



Removing Outside View Surfaces: Sutherland-Hodgman Clipping





- Clip a polygon against a clip plane
  - wholly outside visible region - save nothing
  - wholly inside visible region - save endpoints
  - leave (inside to outside) visible region - save intersection point
  - enter (outside to inside) visible region - save intersection and endpoint

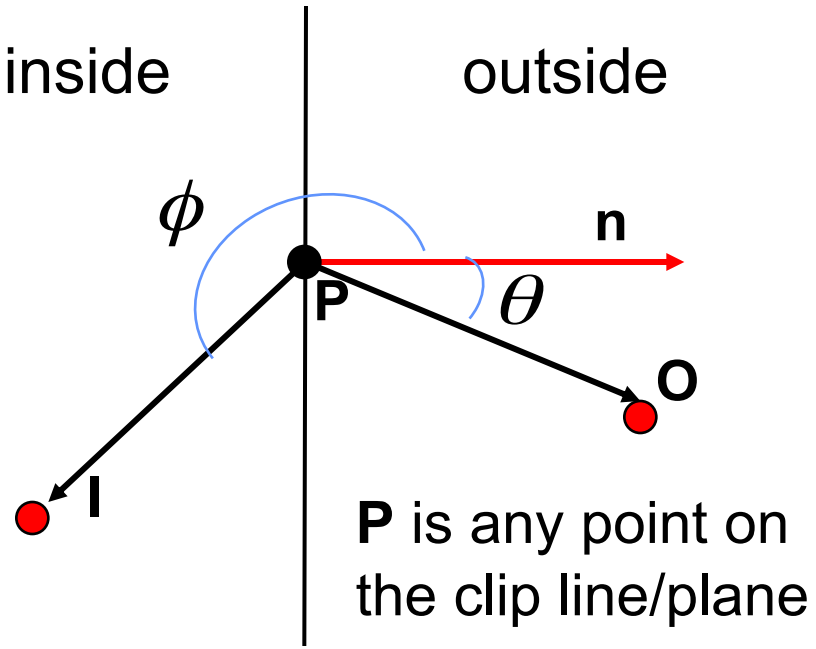
Inside-Outside Testing

- Clipping line/plane has a normal vector pointing toward the outside of the clip region.

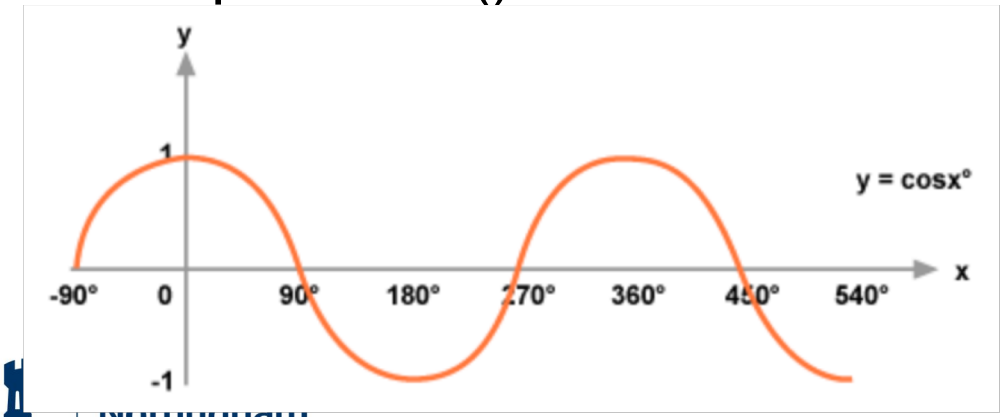
$$\mathbf{n} \bullet (I - P) = |n| \|I - P| \cos\varphi < 0$$

$$\mathbf{n} \bullet (O - P) = |n| \|O - P| \cos\theta > 0$$

- The sign of dot product gives inside/outside information of the point. *(–ve: inside; +ve: outside)*



- Graph of cos() function:



Hidden Surface Removal

To speed up the rendering process, we need to identify polygons that are not visible, e.g., hidden behind other polygons so we do not render them.

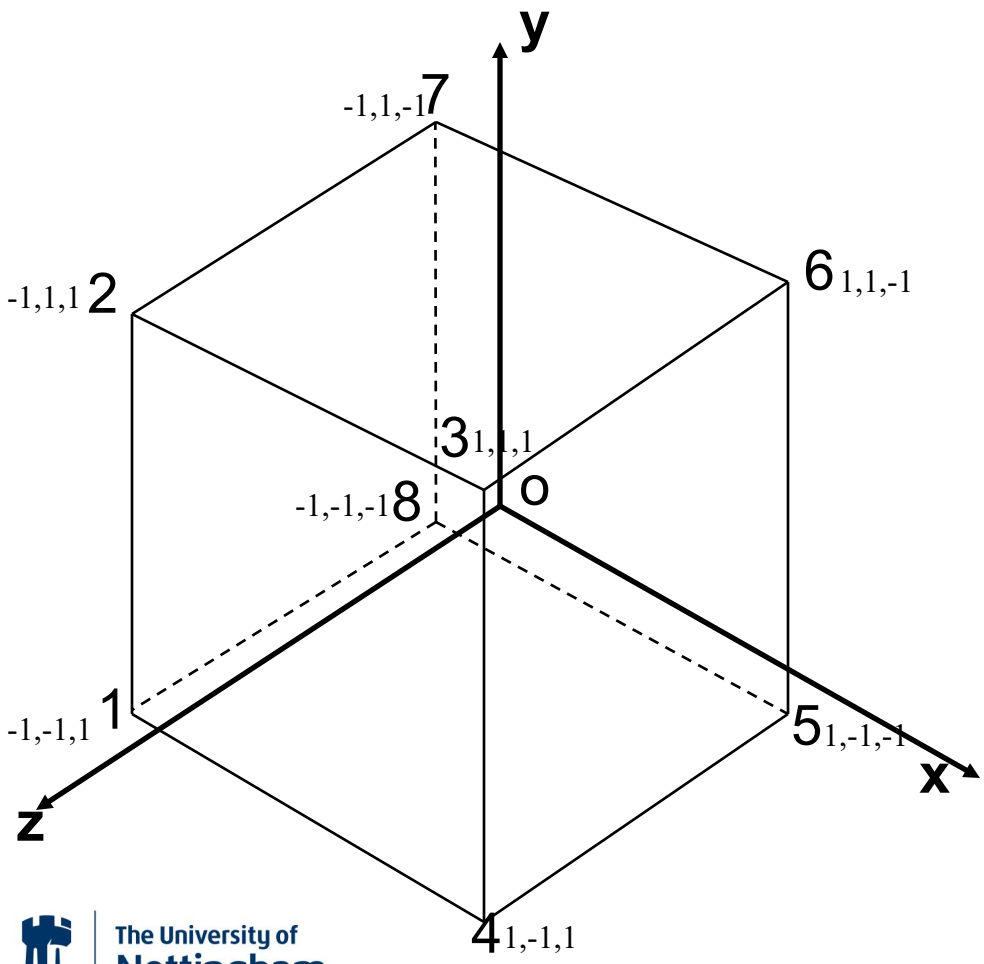
- **Object based methods** – compare each polygon with the rest of polygons. Complexity is O(k^2), where k is the number of polygons.

- **Image based methods** - consider a ray that leaves the center of projection and passes through a pixel and decide which polygon should appear at the pixel and what colour/light/texture the pixel should be.

Polygon Winding Order

- Winding defines the relative order in which the vertex points of a polygon are listed.
- When the user issues a drawing command, the vertices processed are processed in the order provided by vertex listing, which can be either clockwise or anticlockwise.
- Polygons with vertices in anticlockwise order are considered to be on the outside of object.
- Polygons with vertices listed in clockwise order are considered to be on the inside of object.

Polygon Winding Order



Anticlockwise: 4,3,2,1

- glVertex3f(1, -1, 1);
- glVertex3f(1, 1, 1);
- glVertex3f(-1, 1, 1);
- glVertex3f(-1, -1, 1);

Clockwise: 1,2,3,4

- glVertex3f(-1, -1, 1);
- glVertex3f(-1, 1, 1);
- glVertex3f(1, 1, 1);
- glVertex3f(1, -1, 1);

Hidden Surface Removal - Face Culling

- Face culling allows non-visible polygons to be removed before expensive rasterization and/or other operations.

- To activate face culling, GL_CULL_FACE should be enabled (by default, face culling is disabled).

  glEnable(GL_CULL_FACE)

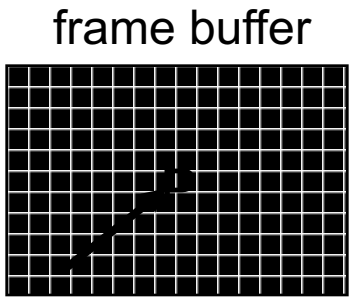
- To select which side of the face will be culled, use glCullFace (**mode**) - where **mode** can be set to GL_FRONT, GL_BACK, or GL_FRONT_AND_BACK. By default, GL_BACK is the face to be culled.

Removing Hidden Surfaces – Z Buffer Algorithm

- Z-buffer is a block of memory the same size as the image.
- Each position in the z-buffer corresponds to an image pixel and stores the z value of the polygon found to be closest to the pixel
- This is because we want the 2D image formed on the image plane to have the same colour as the polygon closest to the image plane
- To enable hidden surface removal in OpenGL: call **glEnable(GL_DEPTH_TEST);**

Removing Hidden Surfaces – Z Buffer Algorithm

- **Consider** a ray that passes through an image pixel P.
- **Calculate** the depths (z values) of polygons that the ray hits.
- **Compare** these depths with the value stored at P in the z-buffer: if a depth value is smaller (polygon nearer to image plane) than the value at P in the z-buffer, **replace** the value in the z-buffer with this value, and **colour** the pixel with the colour of the polygon at this depth.



Replace depth value at the same place in frame buffer with the depth value of the blue square