Viewing

Lecturer: Rachel McDonnell
Assistant Professor of Creative Technologies
Rachel.McDonnell@cs.tcd.ie

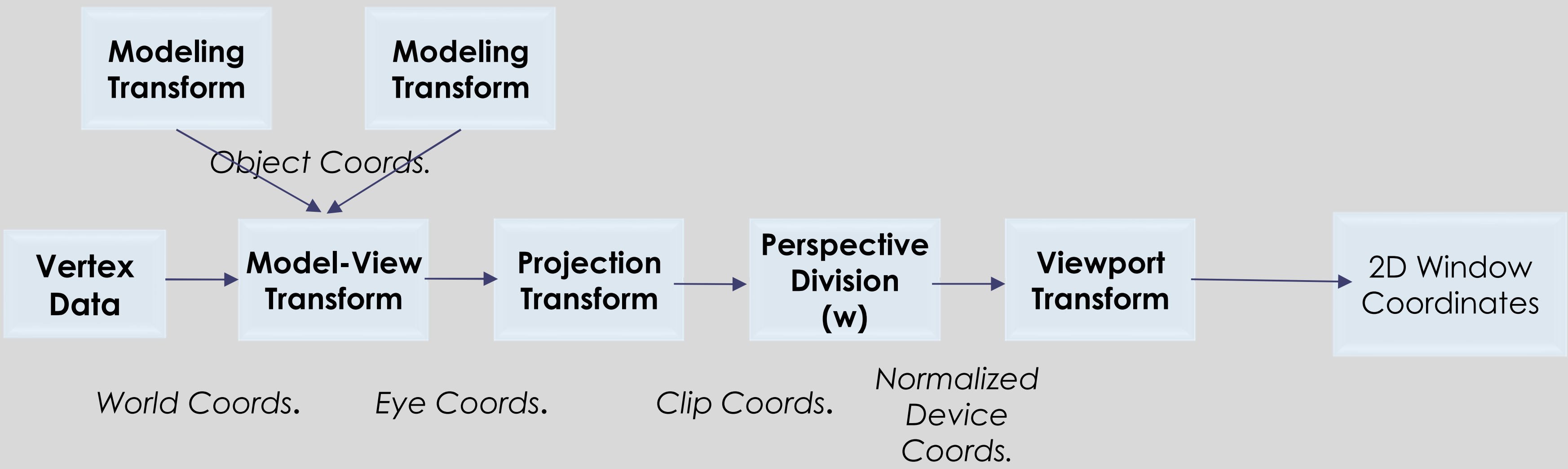
Course www: https://www.scss.tcd.ie/Rachel.McDonnell/

Overview

- Viewing
  - Transformation Pipeline
  - Parallel Projections
  - Perspective Projections
  - Viewport

Transformation Pipeline

- Transformations take us from one "space" to another
  - All of our transforms are 4 x 4 matrices

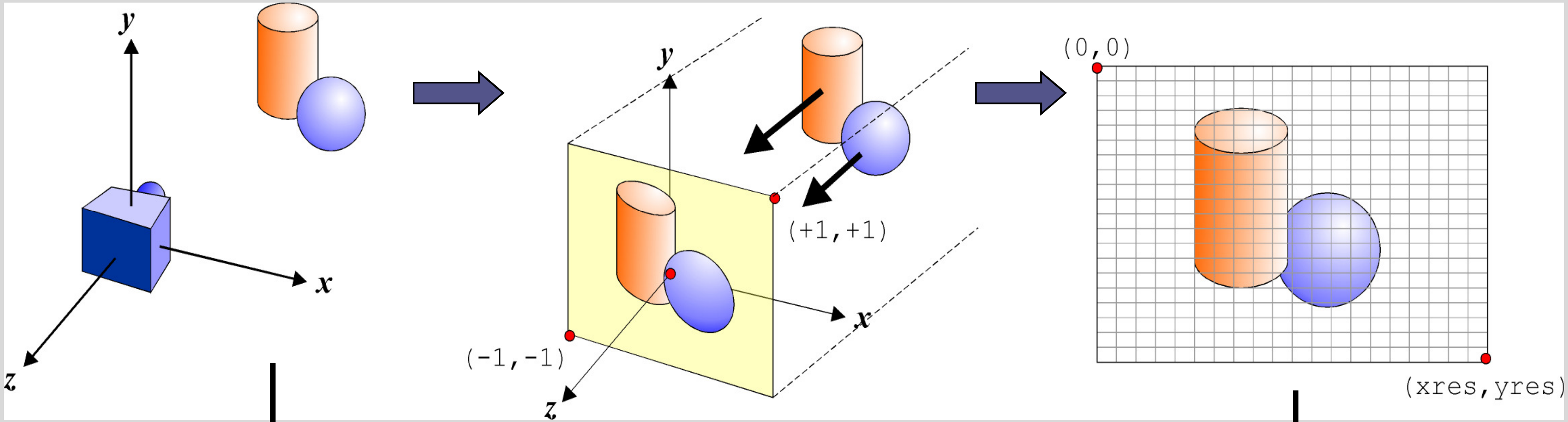


Camera Analogy

- Projection transformations
  - Adjust the lens of the camera
- Viewing transformations
  - Tripod- define position and orientation of the viewing volume in the world
- Modelling transformations
  - Moving the model
- Viewport transformations
  - Enlarge or reduce the physical photograph

Camera Modeling in OpenGL ®

camera coordinate system → viewport coordinate system → device/screen coordinate system



MODEVIEW

GL_PROJECTION

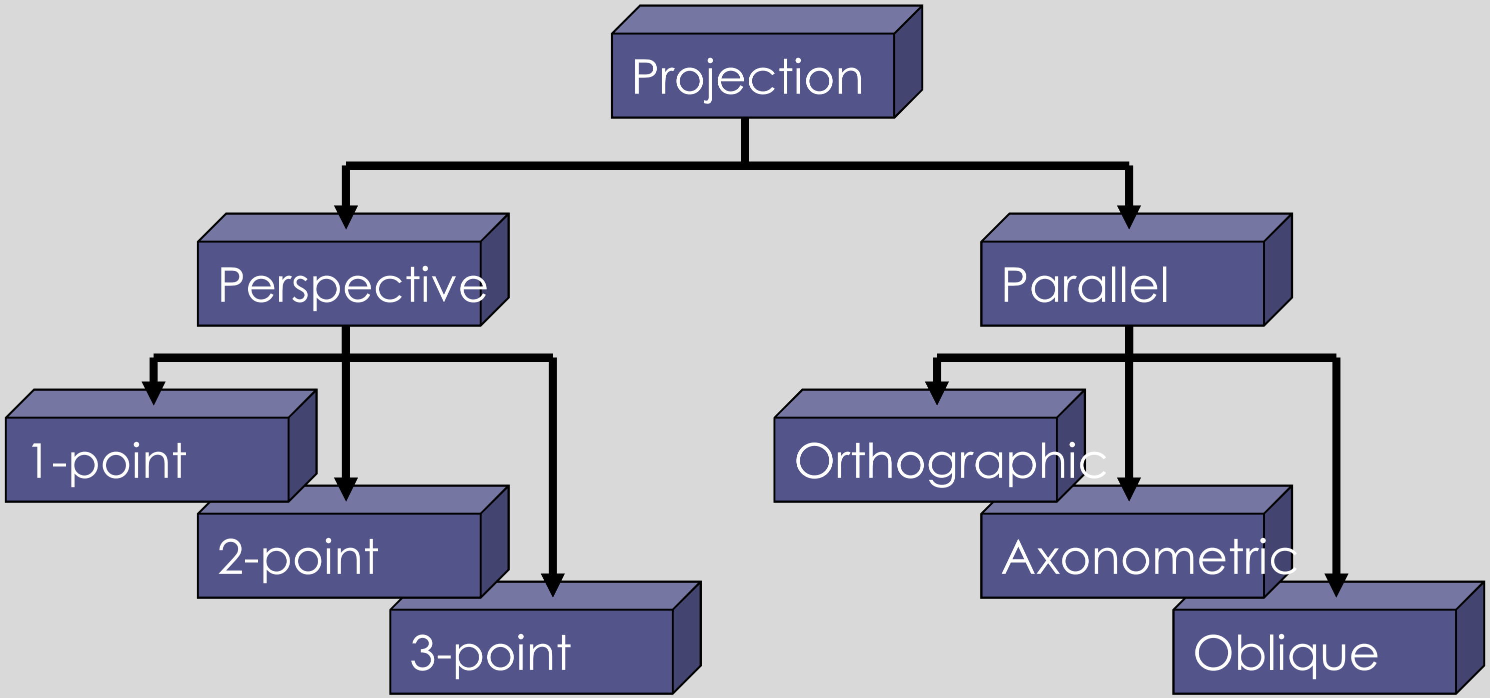
glViewport

Model Matrix

- When you create a triangle or
- Load a mesh from a file
- Has some (0,0,0) origin, local to that particular mesh
- Translate, rotate, scale to position in a virtual world
  - Multiply points with a model matrix (“world matrix”)
  - `mat4 M = T * R * S;`
- `vec4 pos_wor = M * vec4 (pos_loc, 1.0);`

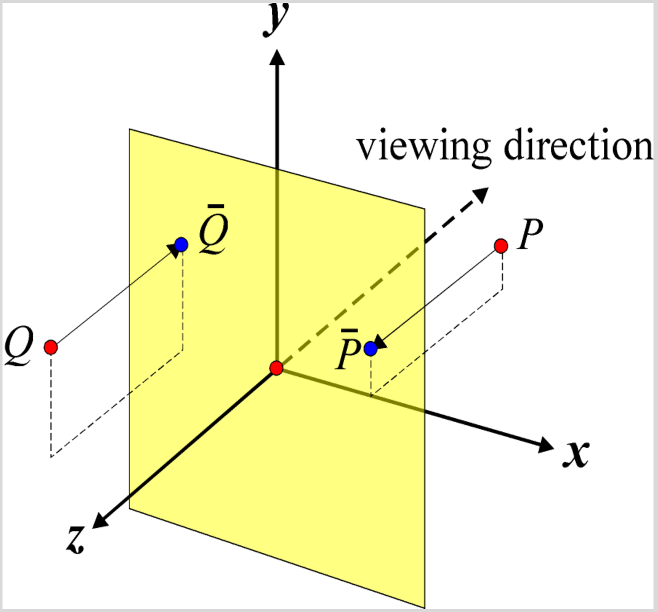
3D → 2D Projection

- Type of projection depends on a number of factors:
  - *location* and *orientation* of the viewing plane (*viewport*)
  - direction of projection (described by a vector)
  - projection type:

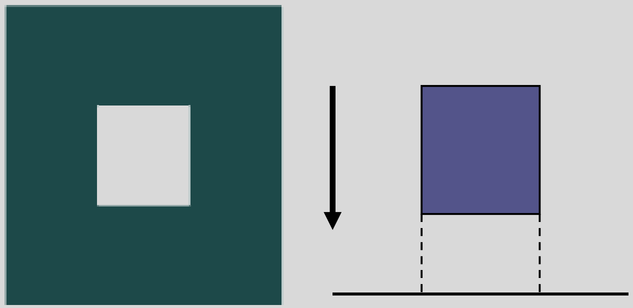


Orthogonal Projections

- The simplest of all projections, *parallel project* onto view-plane.
- Usually view-plane is *axis aligned* (often at *z=0*)



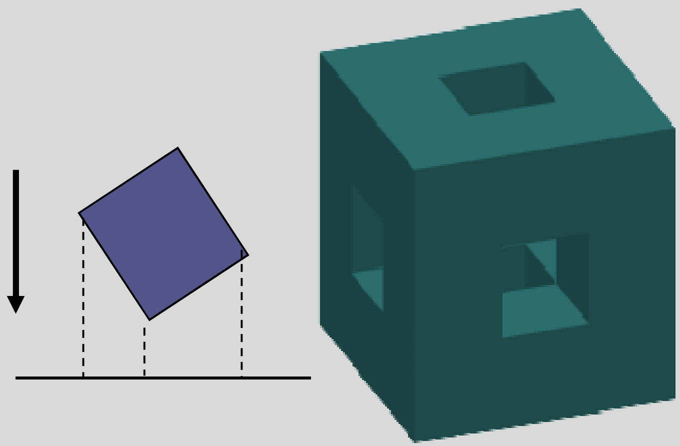
$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} x \\ y \\ 0 \\ 1 \end{bmatrix} \Rightarrow \overline{P} = \mathbf{M}P \text{ where } \mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



orthographic

Orthogonal Projections

- The result is an *orthographic* projection if the object is axis aligned, otherwise it is an *axonometric* projection.

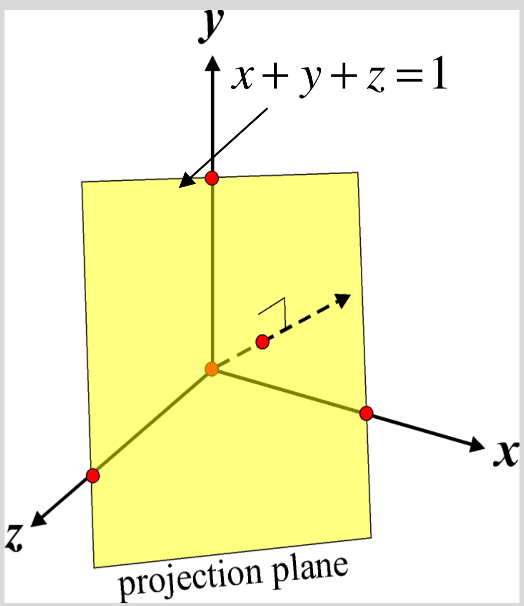


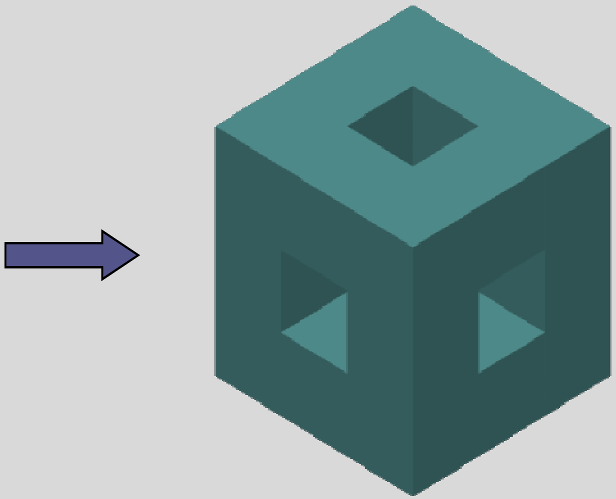
axonometric

Axonometric projection is a type of orthographic projection, used to create a pictorial drawing of an object, where the object is rotated along one or more of its axes relative to the plane of projection.

Orthogonal Projections

- The result is an *orthographic* projection if the object is axis aligned, otherwise it is an *axonometric* projection.
- If the projection plane intersects the principle axes at the same distance from the origin the projection is *isometric*.





Isometric projection

Orthogonal-Projection Matrices

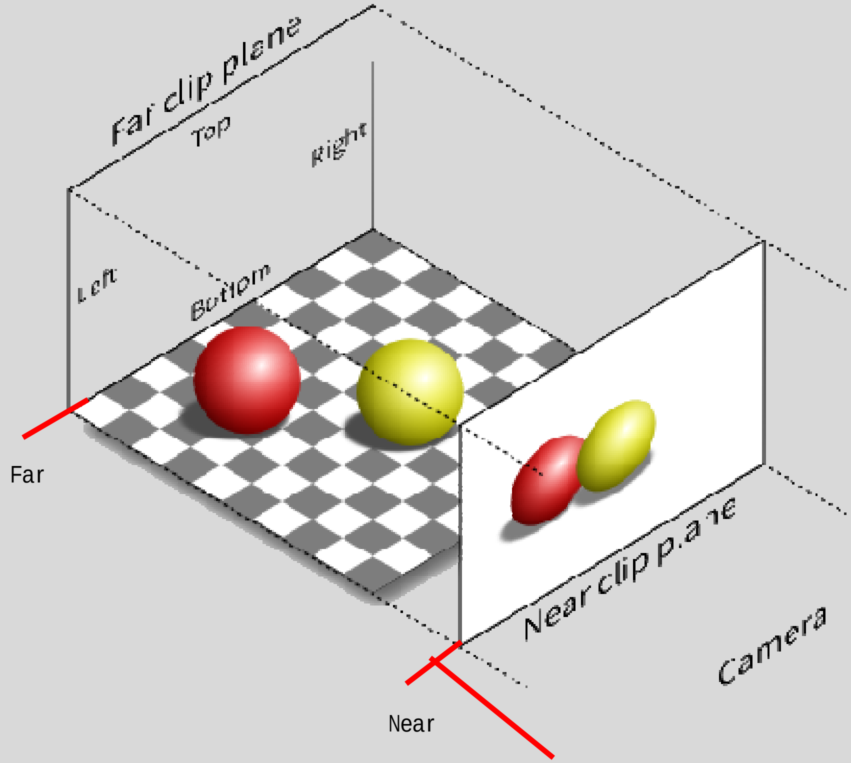
- In OpenGL the default projection matrix is an identity matrix or equivalently:

```
mat4 N = Ortho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
```

- Canonical view volume
- Points within the cube are mapped to the same cube
- Points outside remain outside and are clipped

Parallel Projections in OpenGL

```
mat4 Ortho(left, right, bottom, top, near, far);
```



Note: we always view in -z direction need to transform world in order to view in other arbitrary directions.

What does the matrix do?

$$N = \begin{bmatrix} 2/right-left & 0 & 0 & -(left+right/right-left) \\ 0 & 2/top-bottom & 0 & -(top+bottom/top-bottom) \\ 0 & 0 & -2/far-near & -(far+near/far-near) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scale and Translate

Orthogonal-Projection Matrices

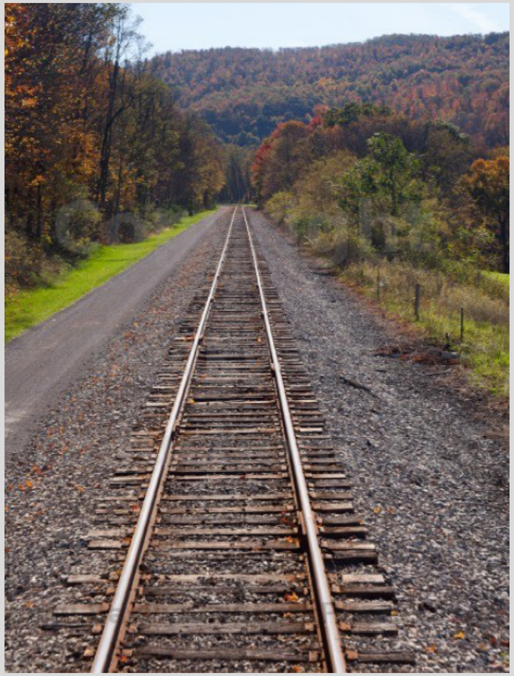
```
mat4 Ortho(left, right, bottom, top, near, far);
```

- Linearly maps view-space coordinates into clip-space coordinates
- Transform this volume to the cube centered at the origin with sides of length 2 (canonical view volume)
- Translate to origin, scale the sides to have a size of 2

$$N = ST = \begin{bmatrix} 2/right-left & 0 & 0 & -(left+right/right-left) \\ 0 & 2/top-bottom & 0 & -(top+bottom/top-bottom) \\ 0 & 0 & -2/far-near & -(far+near/far-near) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

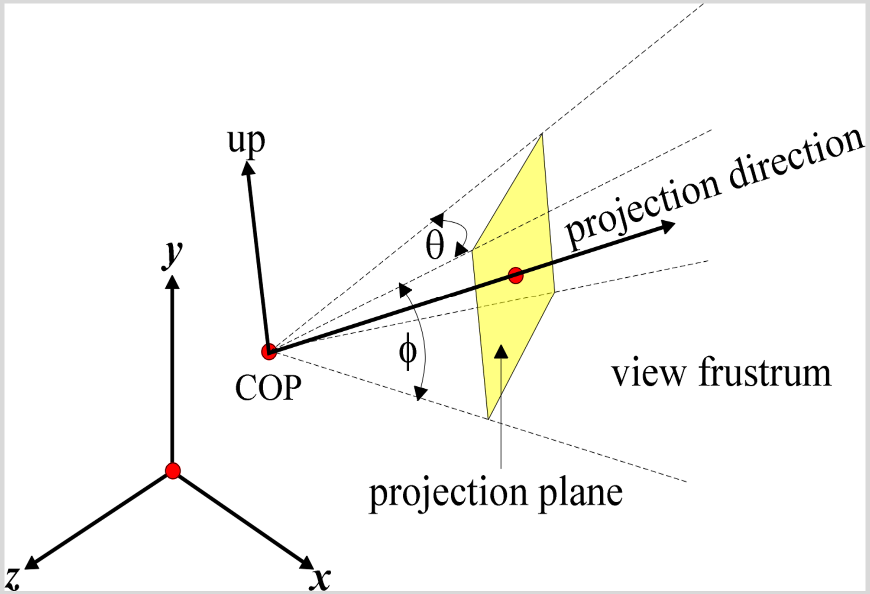
Characteristics

- Parallel Projection
  - Keep parallel lines parallel
  - Preserve size and shape of planar objects
  - Not realistic
    - Cube example
  - Use in architecture
  - Represent less natural image,
  - Simple to do
- Perspective Projection
  - Objects further away appear smaller
  - More realistic



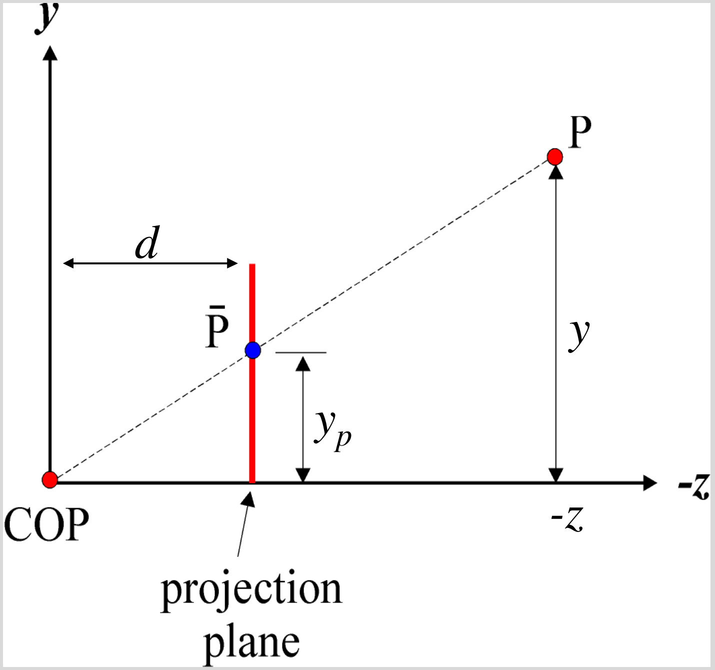
Perspective Projections

- Perspective projections are more complex and exhibit *fore-shortening* (parallel appear to converge at points).
- Parameters:
  - centre of projection (COP)
  - field of view (θ, ϕ)
  - projection direction
  - up direction



Perspective Projections

Consider a perspective projection with the viewpoint at the origin and a viewing direction oriented along the positive -*z* axis and the view-plane located at $z = -d$



$$\frac{y}{z} = \frac{y_P}{d} \Rightarrow y_P = \frac{y}{z/d}$$

Non-uniform foreshortening

a similar construction for x_p

$\Rightarrow$

$$\begin{bmatrix} x_P \\ y_P \\ z_P \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{x}{z/d} \\ \frac{y}{z/d} \\ -d \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ -z \\ z/d \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

can modify use of homogeneous coordinates to handle projections

Transformation Matrix

Homogenous Coordinates

- Consider the matrix

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}$$

- Transforms the point $\begin{bmatrix} x \\ y \\ -z \\ 1 \end{bmatrix} \Rightarrow \begin{bmatrix} x \\ y \\ -z \\ z/d \end{bmatrix}$
- Divide by w to return to original 3D: $\begin{bmatrix} \frac{x}{z/d} \\ \frac{y}{z/d} \\ -d \\ 1 \end{bmatrix}$

Perspective Projections

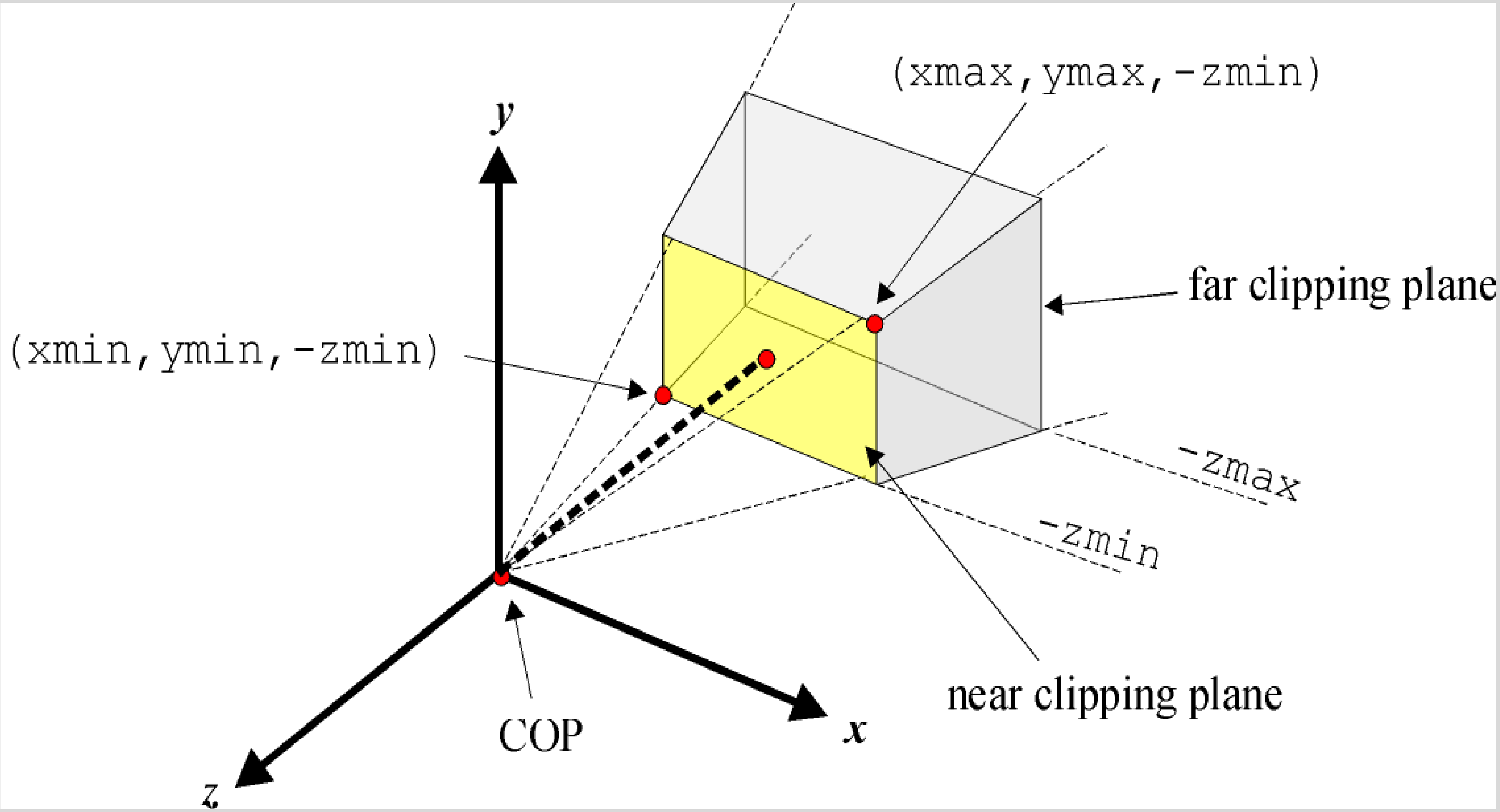
- $(x, y, z) \rightarrow (x_p, y_p, z_p)$
- Although perspective transformations preserve lines, it is not affine!
- Also, it is irreversible
  - All points along a projector project onto the same point, we cannot recover a point from its projection

Perspective Projection

- Depending on the application we can use different mechanisms to specify a perspective view.
- Example: the *field of view* angles may be derived if the distance to the viewing plane is known.
- Example: the viewing direction may be obtained if a point in the scene is identified that we wish to look at.
- You should provide different methods of specifying the perspective view:
  - **`LookAt`**, **`Frustrum`** and **`Perspective`**

Perspective Projections

```
mat4 Frustum(xmin, xmax, ymin, ymax, zmin, zmax);
```



Frustum method

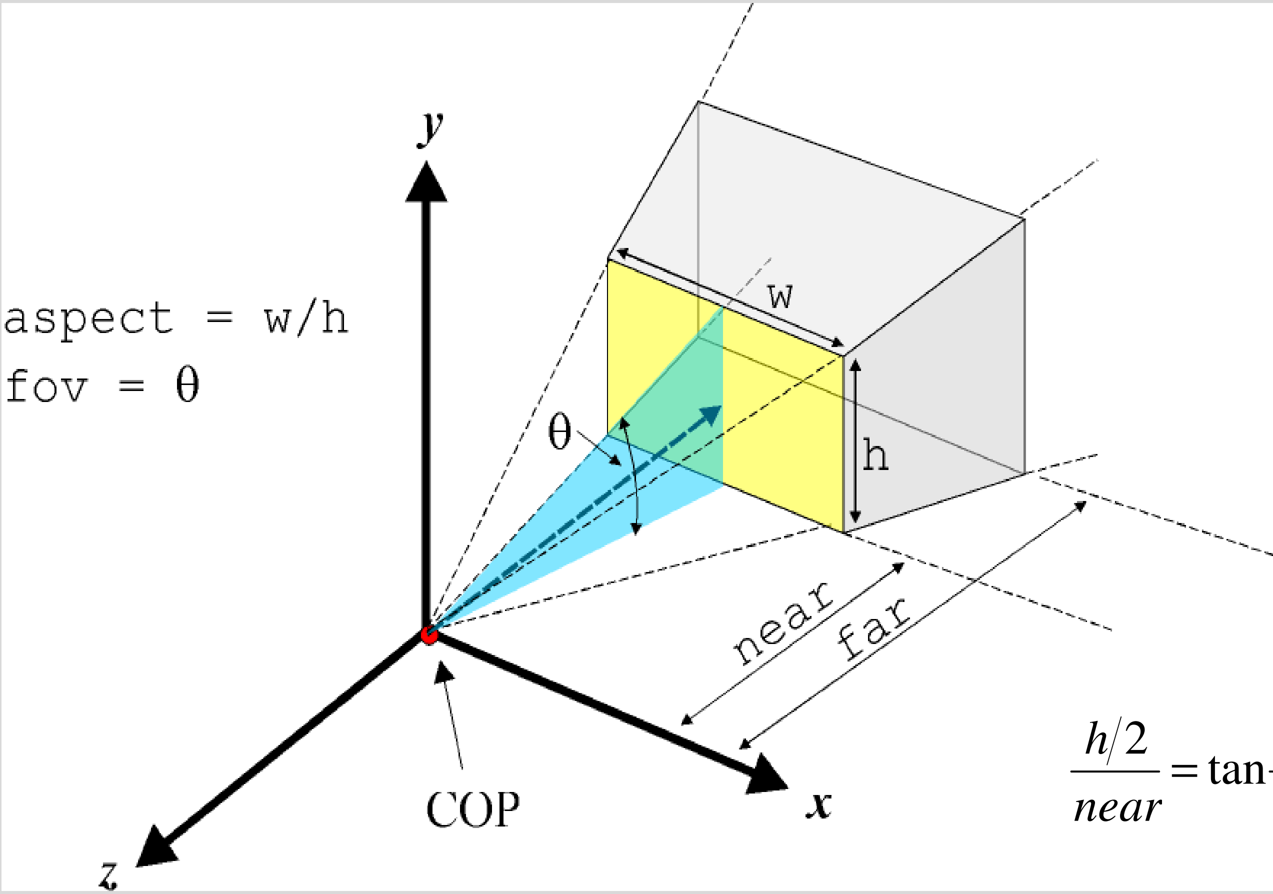
- It is not necessary to have a *symmetric frustrum* like:

```
Frustum(-1.0, 1.0, -1.0, 1.0, 5.0, 50.0);
```

- Non symmetric frustrums introduce *obliqueness* into the projection.
- **`zmin`** and **`zmax`** are specified as <u>positive</u> distances along **`-z`**

Perspective Projections

```
mat4 Perspective(fov, aspect, near, far);
```

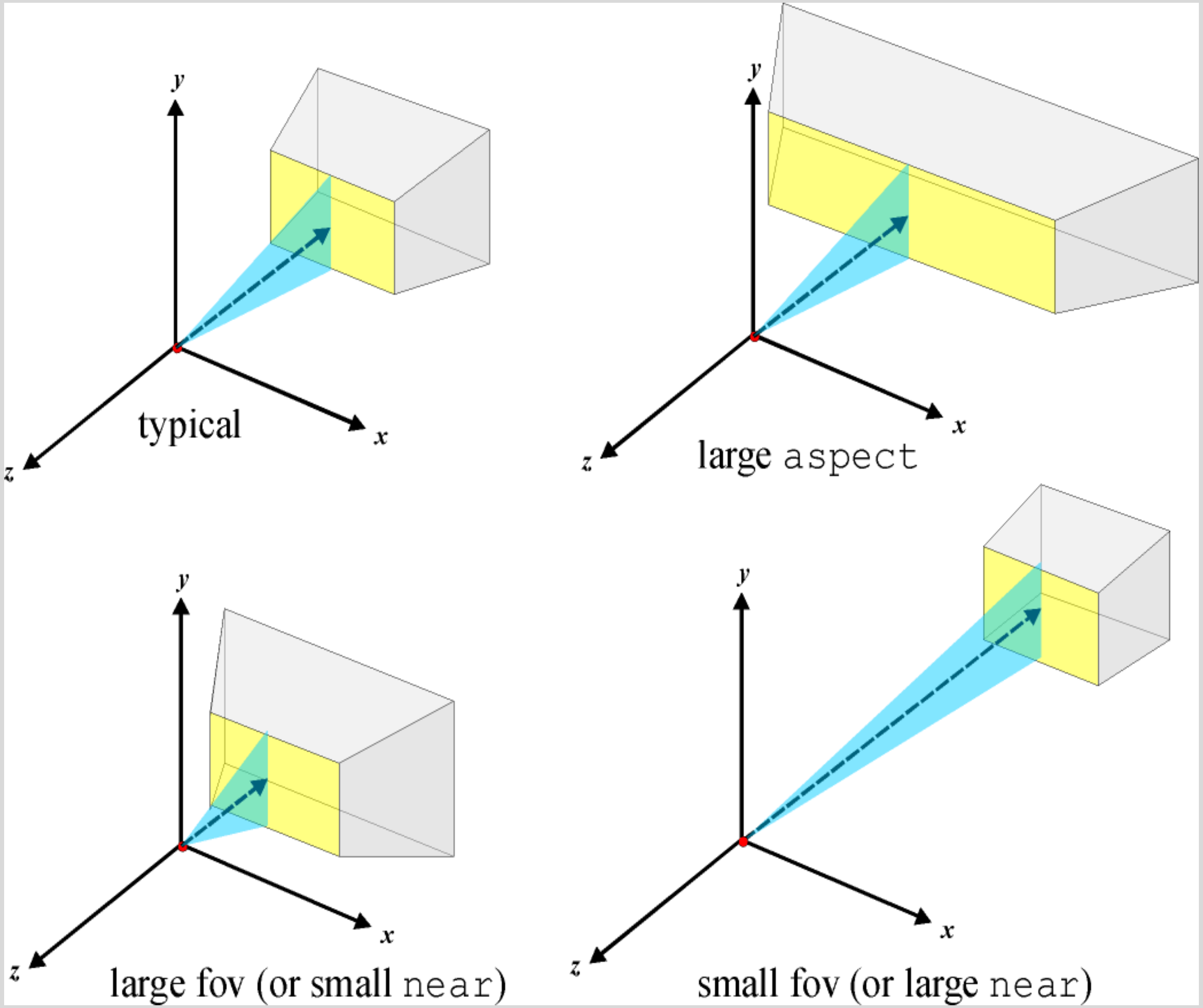


$$\frac{h/2}{near} = \tan\frac{\theta}{2} \Rightarrow h = 2near\tan\frac{\theta}{2}$$

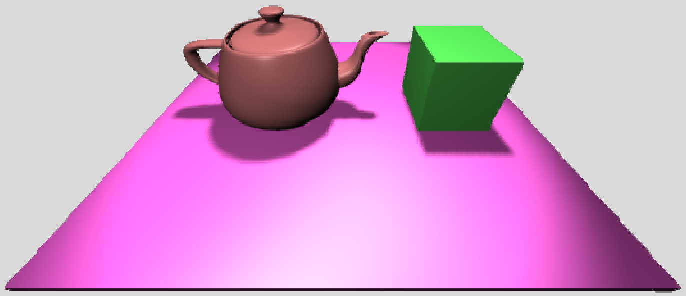
`Perspective Matrix`

- simplify the specification of perspective views.
- Only allows creation of *symmetric frustrums*.
- Viewpoint is at the origin and the viewing direction is the **-z** axis.
- The *field of view* angle, `fov`, must be in the range `[0..180]`
- `aspect` allows the creation of a view frustrum that matches the *aspect ratio* of the viewport to eliminate distortion.

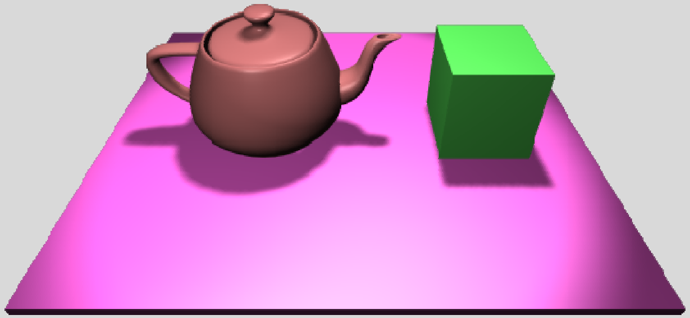
Perspective Projections



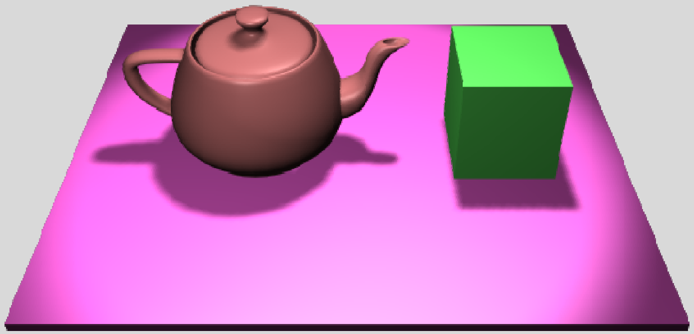
Lens Configurations



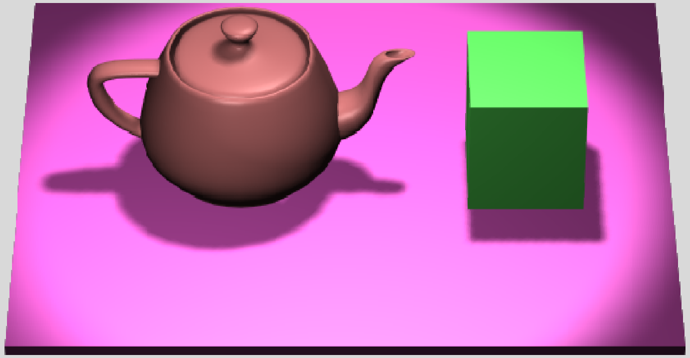
10mm Lens (fov = 122°)



20mm Lens (fov = 84°)



35mm Lens (fov = 54°)



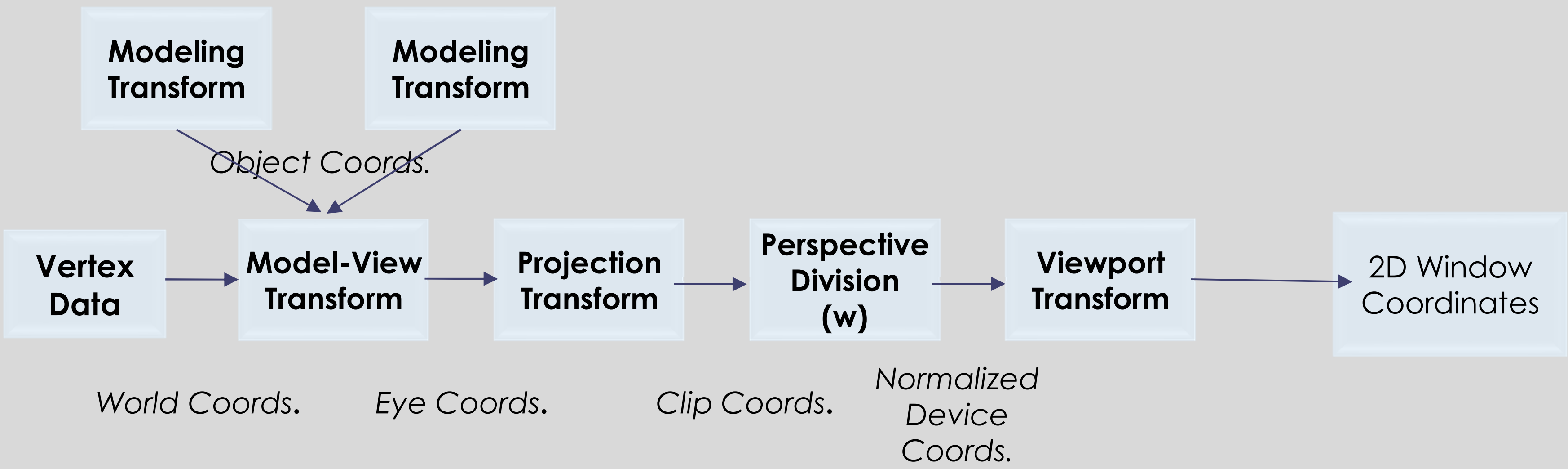
200mm Lens (fov = 10°)

Positioning the Camera

- The previous projections had limitations:
  - usually fixed origin and fixed projection direction
- To obtain arbitrary camera orientations and positions we manipulate the `VIEW` matrix. This positions the camera w.r.t. the model.
- We wish to position the camera at (10, 2, 10) w.r.t. the world
- Two possibilities:
  - transform the world prior to creation of objects using `translate` and `rotate matrices:`
  - **`Translate(-10, -2, -10);`**
  - use `LookAt` to position the camera with respect to the world co-ordinate system:
  - **`LookAt(10, 2, 10, … );`**
- Both are *equivalent*.

Transformation Pipeline

- Transformations take us from one "space" to another
  - All of our transforms are 4 x 4 matrices

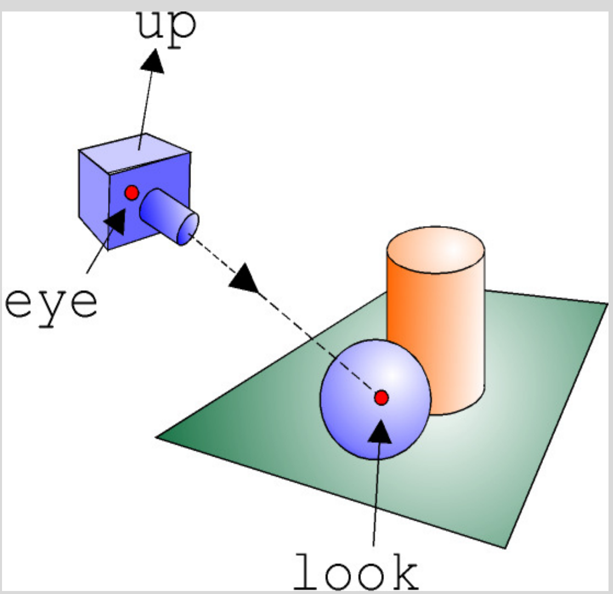


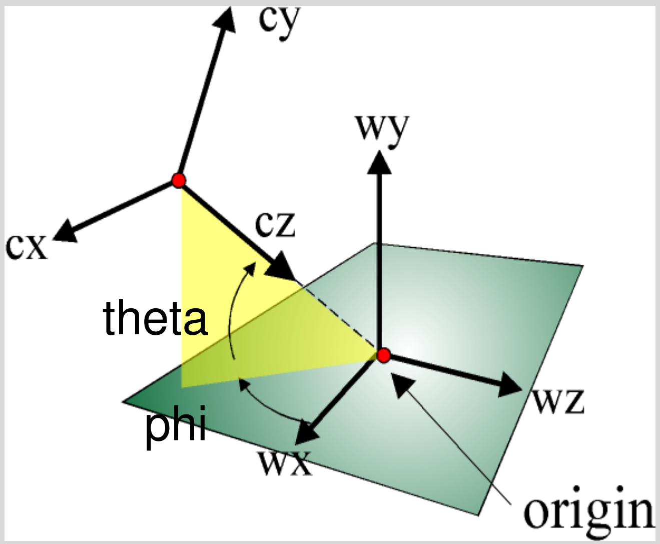
View Matrix

- Objects positioned in scene or “virtual world”
- Has a world (0,0,0) origin
- Can get distances between objects
- Now we want to show the view from a camera, moving through the virtual world
- Multiply world space points by a view matrix to get to eye space
- `mat4 V = R * T; // inverse of cam pos & angle`
- `mat4 V = lookAt (vec3 pos, vec3 target, vec3 up);`
- `vec4 pos_eye = V * pos_wor;`

Positioning the Camera

```
LookAt(eyex, eyey, eyez, lookx, looky, lookz, upx, upy, upz);
```



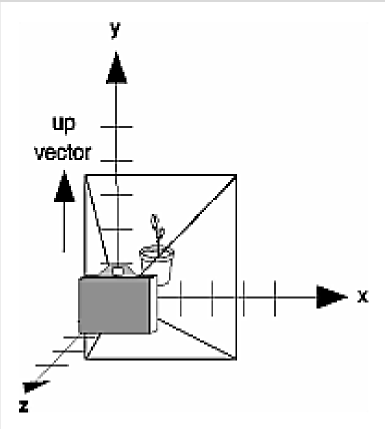


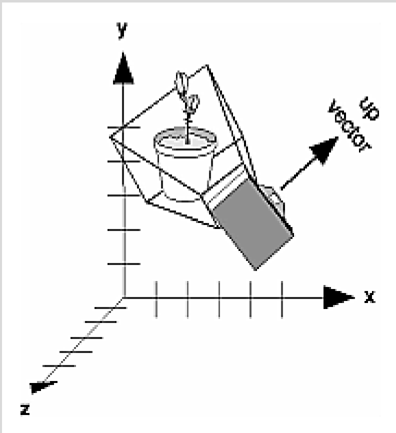
equivalent to:

```
Translate(-eyex, -eyey, -eyez);
Rotate(theta, 1.0, 0.0, 0.0);
Rotate(phi, 0.0, 1.0, 0.0);
```

Up Vector

- Up vector
  - Perpendicular to the line of sight
  - Must not be parallel
  - Tells which direction is up (i.e. the direction from the bottom to the top of the viewing volume)





Lookat

- Lookat is particularly useful when you want to pan across a scene (e.g., a landscape)

The Viewport

- The projection matrix defines the mapping from a 3D world co-ordinate to a 2D viewport co-ordinate.
- The viewport extents are defined as a parameter of the projection:
  - **Frustum(l,r,b,t,n,f)** $\Rightarrow$

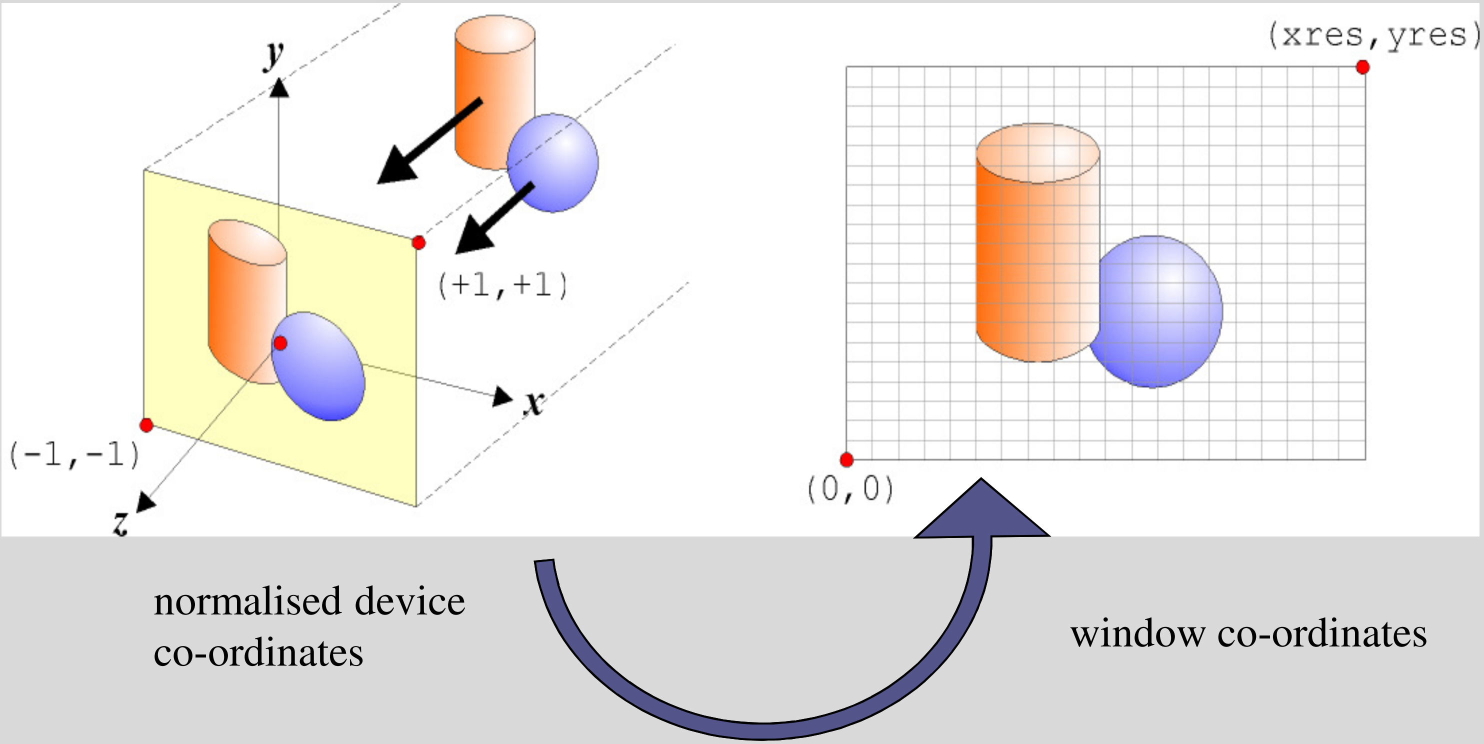
  - **Perspective(fv,a,n,f)** $\Rightarrow$

$$\mathbf{h} = \mathbf{n} \cdot \tan\frac{\mathbf{fv}}{2}$$

$$\mathbf{w} = \mathbf{h} \cdot \mathbf{a}$$

The Viewport

- We need to associate the 2D *viewport co-ordinate system* with the *window co-ordinate system* in order to determine the correct pixel associated with each vertex.



Viewport to Window Transformation

- An *affine* planar transformation is used.
- After projection to the viewplane, all points are transformed to normalised device co-ordinates: `[−1…+1, −1…+1]`

$$x_n = 2\left(\frac{x_p - x_{\min}}{x_{\max} - x_{\min}}\right) - 1$$

$$y_n = 2\left(\frac{y_p - y_{\min}}{y_{\max} - y_{\min}}\right) - 1$$

Viewport to Window Transformation

- `glViewport` used to relate the co-ordinate systems:

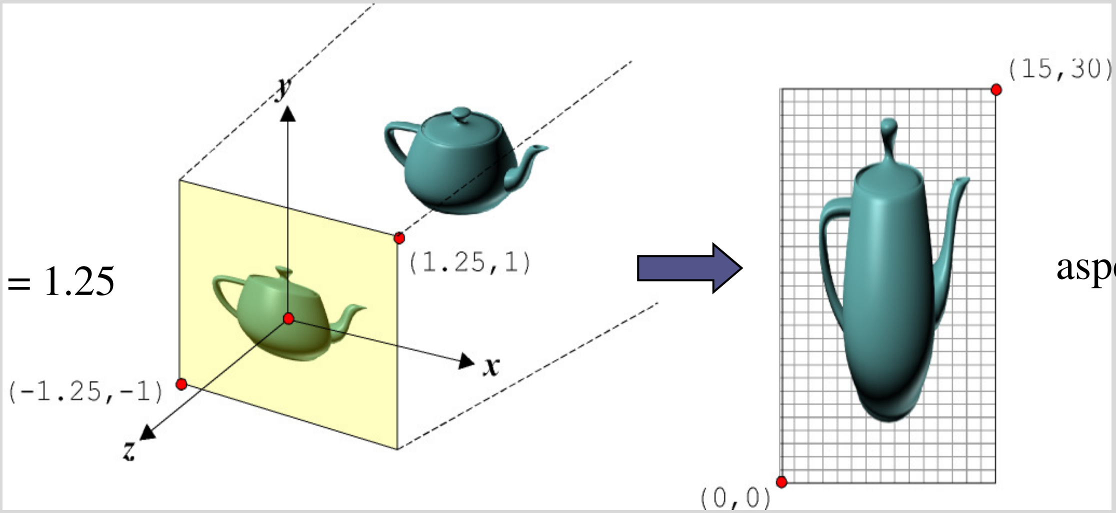
  `glViewport(int x, int y, int width, int height);`
- `(x,y)` = location of bottom left of viewport within the window
- `width,height` = dimension in pixels of the viewport $\Rightarrow$

$$x_w = (x_n + 1)\left(\frac{\mathbf{width}}{2}\right) + \mathbf{x} \quad y_w = (y_n + 1)\left(\frac{\mathbf{height}}{2}\right) + \mathbf{y}$$

Aspect Ratio

- The *aspect ratio* defines the relationship between the width and height of an image.
- Using `Perspective` matrix, a viewport aspect ratio may be explicitly provided, otherwise the aspect ratio is a function of the supplied viewport width and height.
- The aspect ratio of the window (defined by the user) must match the viewport aspect ratio to prevent unwanted *affine* distortion:

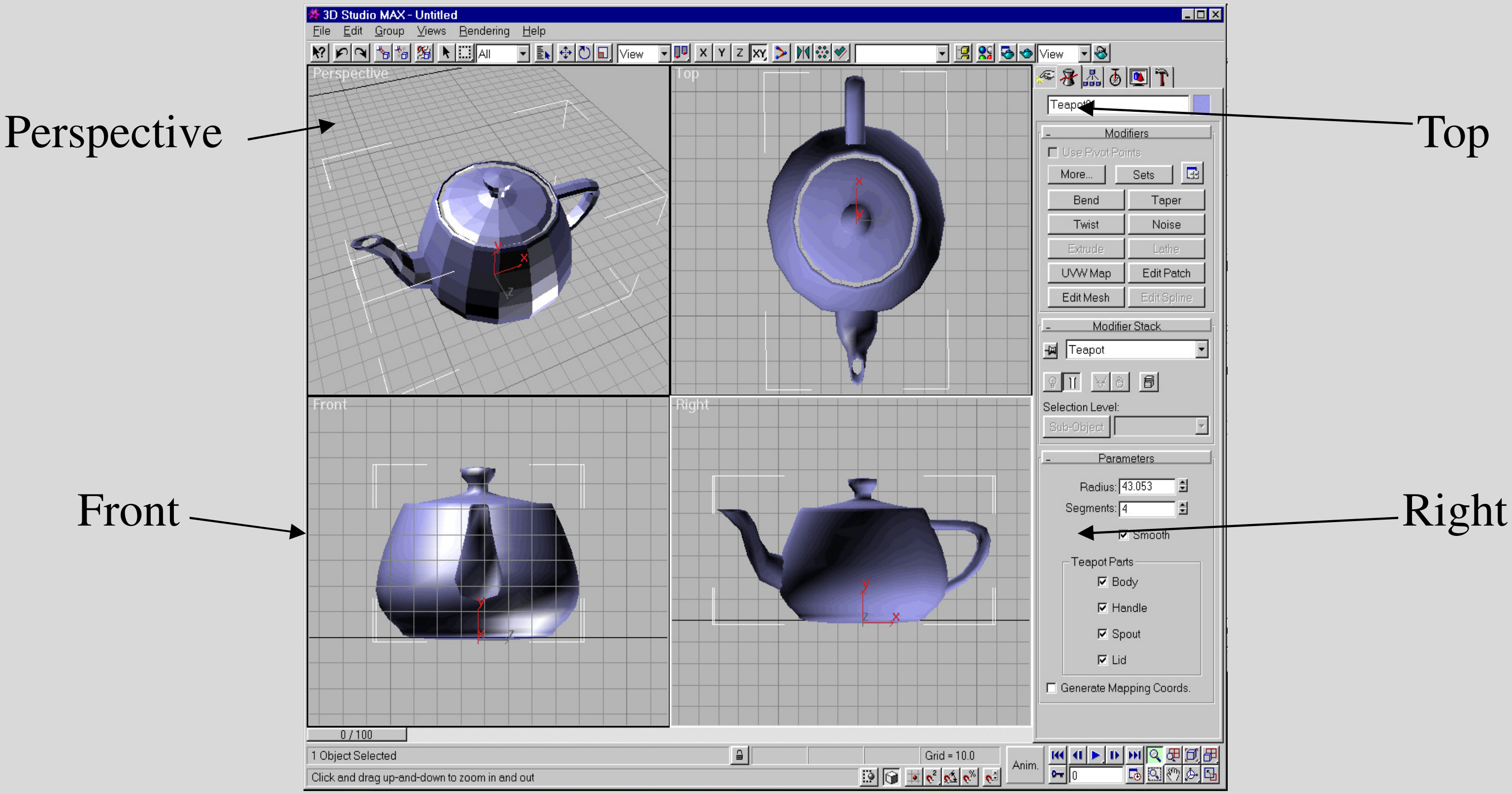
aspect ratio = 1.25



aspect ratio = 0.5

Multiple Projections

- To help 3D understanding, it can be useful to have *multiple projections* available at any given time
  - usually: plan (top) view, front & left or right elevation (side) view



```
void display(){

    // tell GL to only draw onto a pixel if the shape is closer to the viewer
    glEnable (GL_DEPTH_TEST); // enable depth-testing
    glDepthFunc (GL_LESS); // depth-testing interprets a smaller value as "closer"
    glClearColor (0.5f, 0.5f, 0.5f, 1.0f);
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glUseProgram (shaderProgramID);

    //Declare your uniform variables that will be used in your shader
    int matrix_location = glGetUniformLocation (shaderProgramID, "model");
    int view_mat_location = glGetUniformLocation (shaderProgramID, "view");
    int proj_mat_location = glGetUniformLocation (shaderProgramID, "proj");


    //Here is where the code for the viewport lab will go, to get you started I have drawn a t-pot in the bottom left
    //The model transform rotates the object by 45 degrees, the view transform sets the camera at -40 on the z-axis, and

    // bottom-left
    mat4 view = translate (identity_mat4 (), vec3 (0.0, 0.0, -40.0));
    mat4 persp_proj = perspective(45.0, (float)width/(float)height, 0.1, 100.0);
    mat4 model = rotate_z_deg (identity_mat4 (), 45);

    glViewport (0, 0, width / 2, height / 2);
    glUniformMatrix4fv (proj_mat_location, 1, GL_FALSE, persp_proj.m);
    glUniformMatrix4fv (view_mat_location, 1, GL_FALSE, view.m);
    glUniformMatrix4fv (matrix_location, 1, GL_FALSE, model.m);
    glDrawArrays (GL_TRIANGLES, 0, teapot_vertex_count);

    // bottom-right

    // top-left

    // top-right

    glutSwapBuffers();
}
```

External Shaders

simpleVertexShader - Notepad

File Edit Format View Help

```
#version 330

in vec3 vertex_position;
in vec3 vertex_normals;

out vec3 n_eye;

uniform mat4 view;
uniform mat4 proj;
uniform mat4 model;

void main(){
        n_eye = (view * vec4 (vertex_normals, 0.0)).xyz;
        gl_Position =  proj * view * model * vec4 (vertex_position, 1.0);
}
```

simpleFragmentShader - Notepad

File Edit Format View Help

```
#version 330

in vec3 n_eye;
void main(){
gl_FragColor = vec4 (n_eye, 1.0);
}
```

- Order of multiplication is fundamentally important
- Never compare variables from different coordinate spaces
- Use a postfix or prefix naming convention for variables

Reading List & Practical Tasks

- Interactive Computer Graphics, A Top-down Approach with OpenGL, 6th edition, Chapter 4 on Viewing
  - Edward Angel
- Fundamentals of Computer Graphics, 3rd Edition, Shirley and Marschner, Chapter 7
  - Equation 6.7 shows derivation of scale and translate for Orthographic matrix
  - Section 7.1 Discusses Viewing Transformations
- Akenine Moeller et. al "Real-Time Rendering" Ch. 2 and 4.6 "Projections"

- Know how to work out the pipeline by hand on paper for 1 vertex & M, V, and P
- Hint: add a "print_matrix(m)" function to check contents