Deliverables and Deadlines:

- 15/10/2024: **Deliverable 1** - Project Proposal and Design Document (5 Marks)
  - Submit a written proposal (1-2 pages) outlining your intended underwater ecosystem
  - Describe key features you plan to implement (types of marine life, lighting effects, unique interactions)
  - Mention any advanced or custom features that you are considering
  - Include sketches, diagrams or screenshot illustrating the scene layout
- 05/11/2024: **Deliverable 2**: Core Scene Development and Functionality (35 Marks)
  - Basic Scene Setup and Object Loading (15 Marks): Based on your selected theme, create a cohesive 3D underwater environment with water including 3D models, and terrain.
  - User Camera Control (10 Marks): Implement user camera movement for scene exploration. Basic controls (e.g., forward, backward, left, right) are required, with additional marks for more complex camera interactions.
  - Hierarchical Animation (10 Marks): Animate at least one creature or object with a hierarchical structure (e.g., a fish with fins, plants swaying). Marks are awarded based on the hierarchy's meaningful use and complexity.
- **Deliverable 3**: Final Submission – Advanced Features, Interactivity, and Polish (60 Marks)
  - Lighting and Shading (10 Marks): Implement basic Phong Illumination with one or more light sources and proper normal transformations. The lighting type (point, directional, or spotlight) and shading model (flat, Gouraud, or Phong) should complement the scene's underwater ambiance.
  - Texture Mapping (10 Marks): Apply textures to models using image files (e.g., jpgs), with extra marks for texture quality and variety.
  - Advanced Features (30 Marks): Students add one or more advanced features such as intelligent character behavior, procedural terrain, alpha blending, or other effects. Marks will be awarded based on feature complexity, creativity, and how effectively the feature fits the underwater theme.
  - Overall Aesthetic and Technical Quality (10 Marks): Marks are given for the immersive quality and polish of the final scene. This includes fluid animations, smooth interactions, cohesive lighting, and the artistic impact of the completed ecosystem.

Viewing

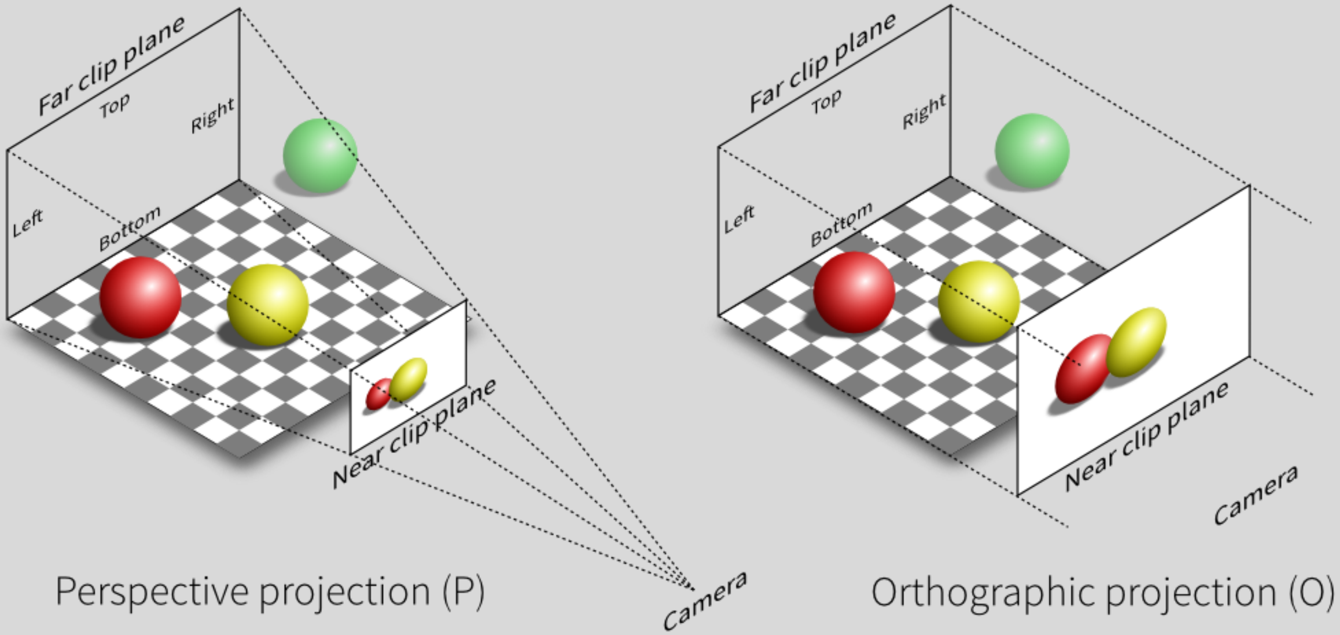
Rachel McDonnell
Associate Professor in Creative Technologies
ramcdonn@tcd.ie

Course www: **Blackboard**

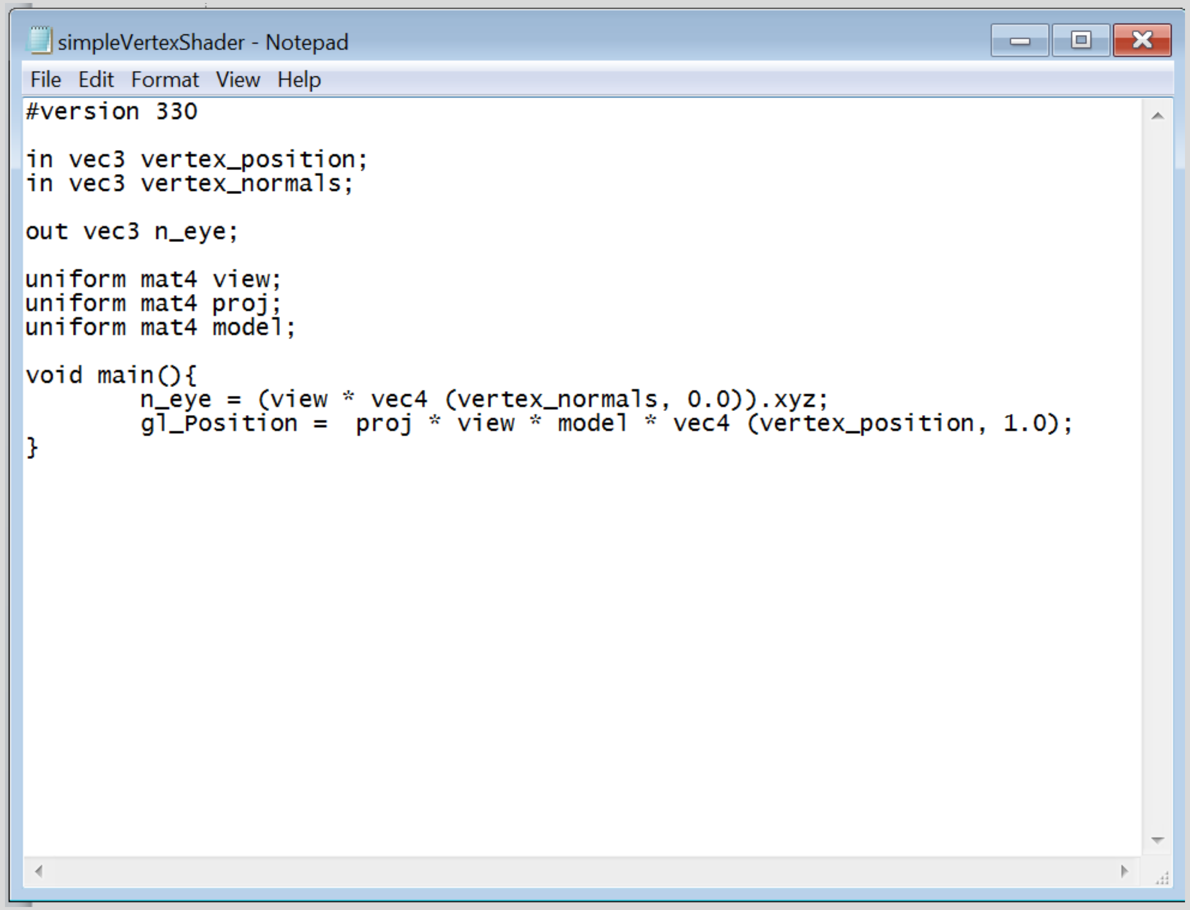
Credits: Some slides taken from Robb T. Koether, Hampden-Sydney College

Overview

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



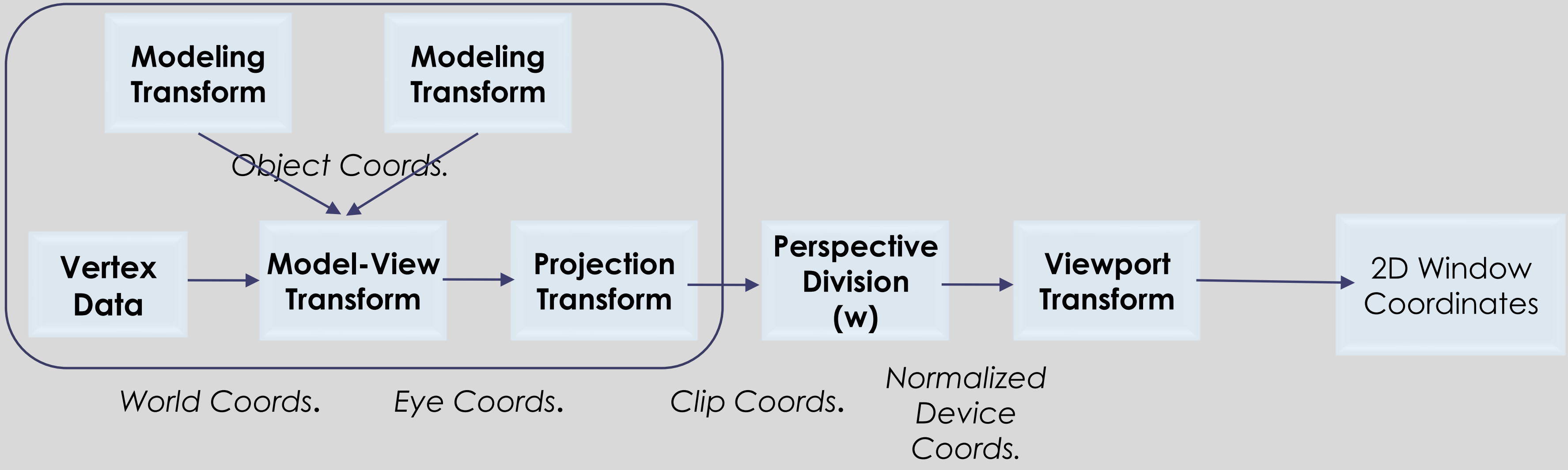
New Vertex Shader




```
#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);
}
```

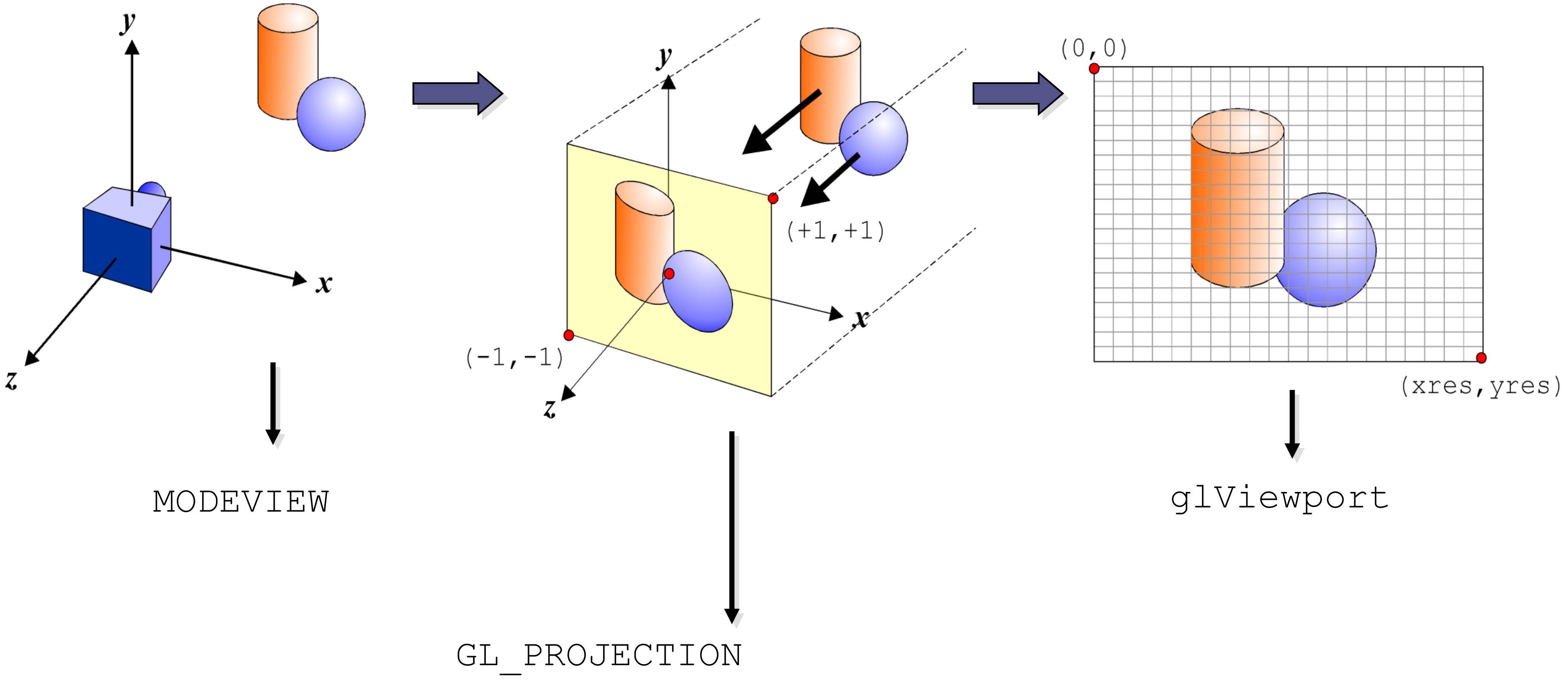
Transformation Pipeline

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



Camera Modeling in OpenGL ®

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

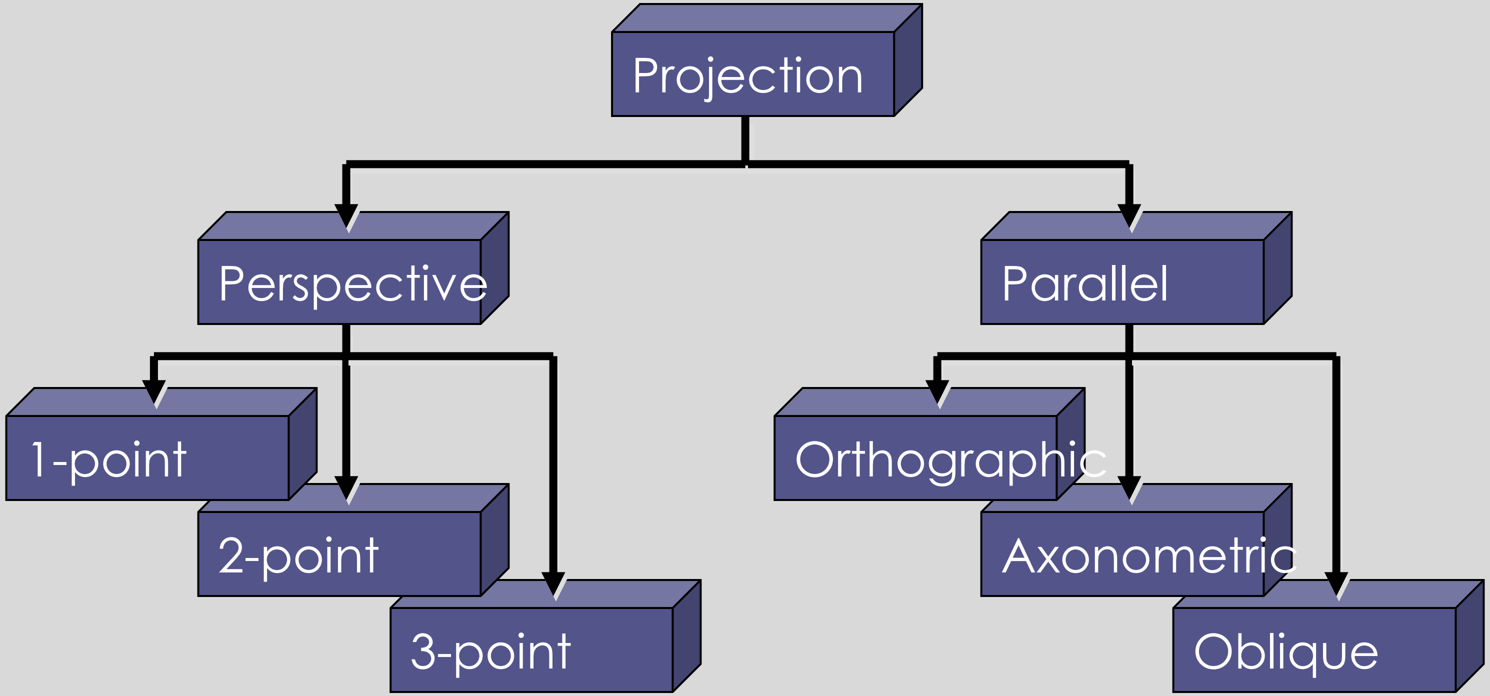


Model Matrix

- 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_world = M * vec4 (pos_loc, 1.0);`

3D → 2D Projection

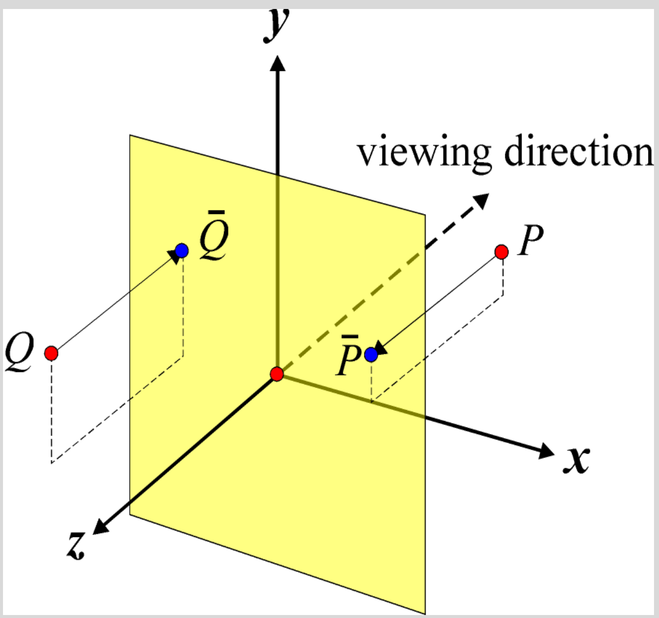
- Type of projection depends on a number of factors:
  - *location* and *orientation* of the viewing plane (*viewport*)
  - direction of projection (described by a vector)
  - projection type:

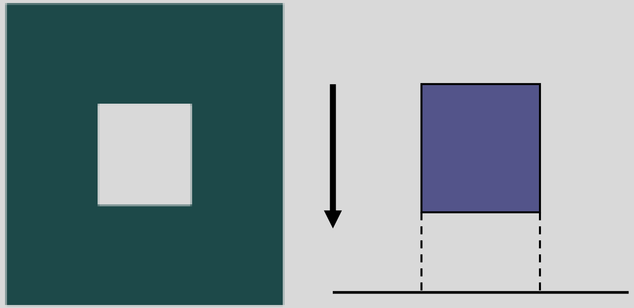


Orthogonal Projections

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

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

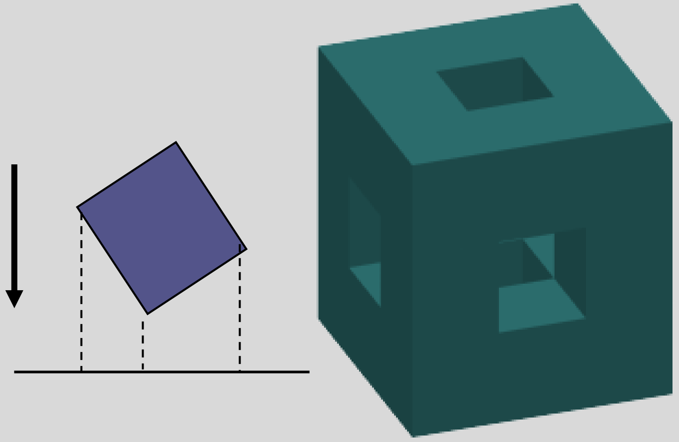




orthographic

Orthogonal Projections

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

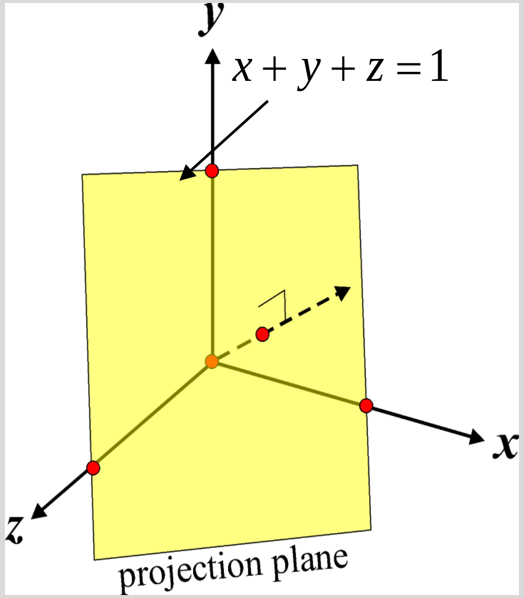


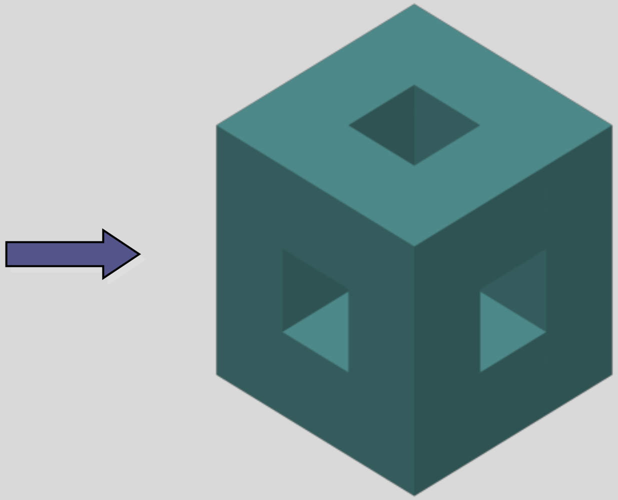
axonometric

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

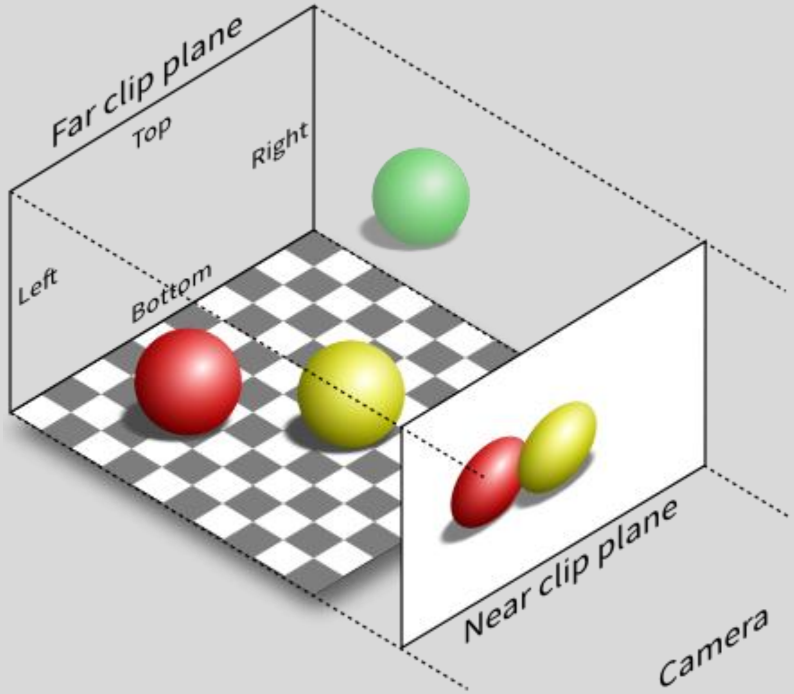
- In OpenGL the default projection matrix is an identity matrix or equivalently:

```
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);
```



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

What does the matrix do?

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

Orthogonal-Projection Matrices

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

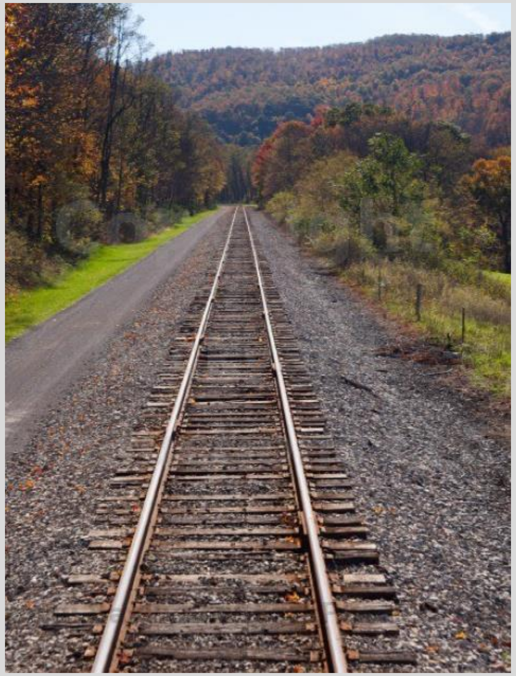
- Linearly maps view-space coordinates into clip-space coordinates
- Transform this volume to the cube centered at the origin with sides of length 2 (canonical view volume)
- Translate to origin, scale the sides to have a size of 2

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

http://www.songho.ca/opengl/gl_projectionmatrix.html

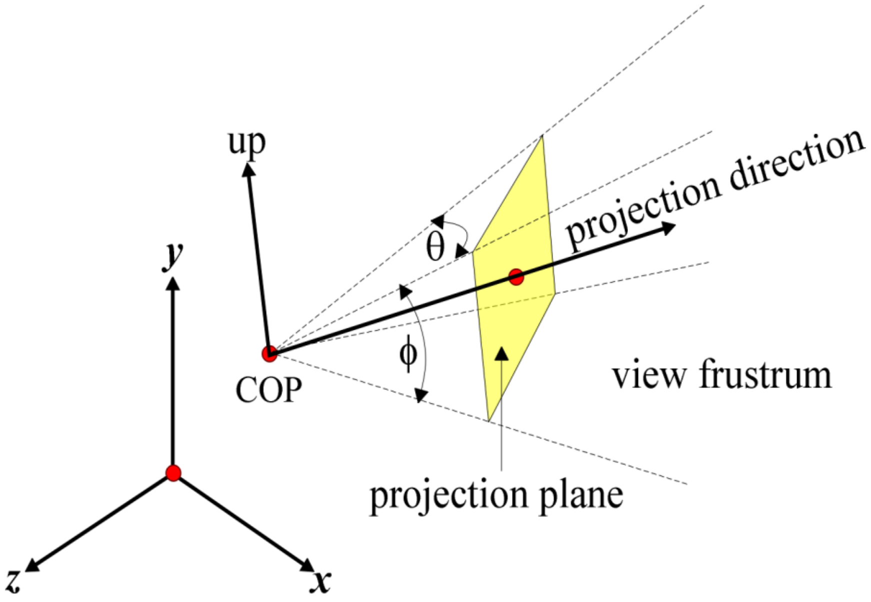
Characteristics

- 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



Homogenous Coordinates

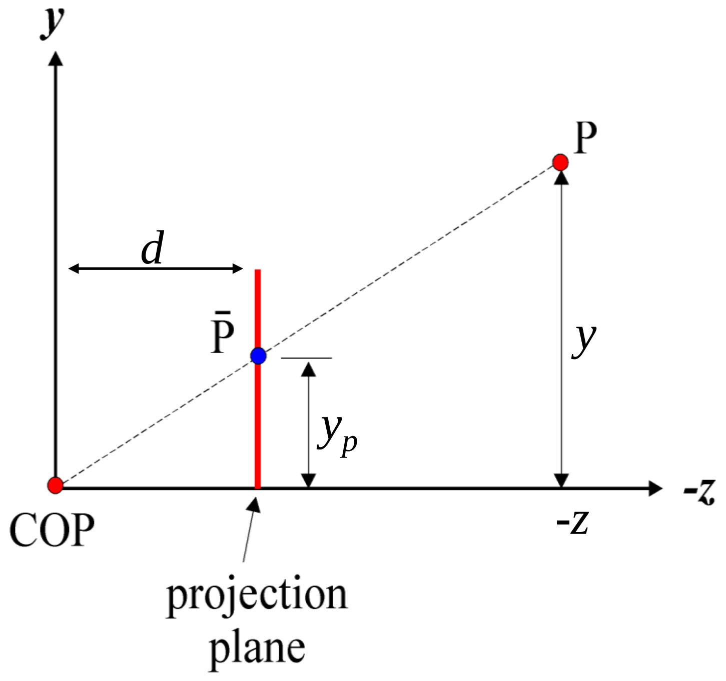
$$\begin{bmatrix} x/W \\ y/W \\ z/W \\ \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ W \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ z \\ \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- Up to now, w = 1
- For Perspective projection, w is no longer = 1

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

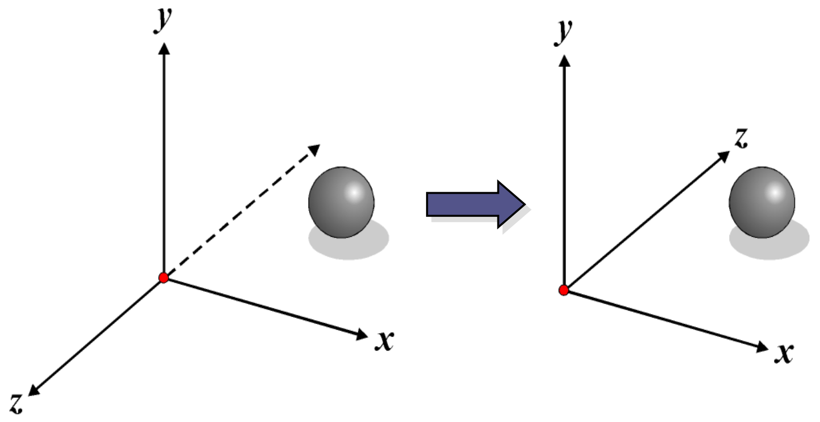
Homogenous Coordinates

- 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: $\begin{bmatrix} \frac{x}{z/d} \\ \frac{y}{z/d} \\ -d \\ 1 \end{bmatrix}$

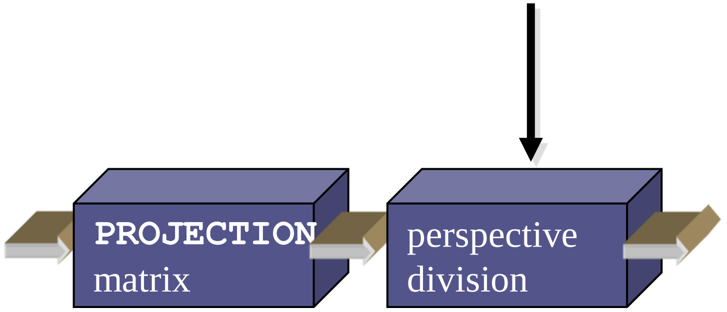
Perspective Projections Details



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

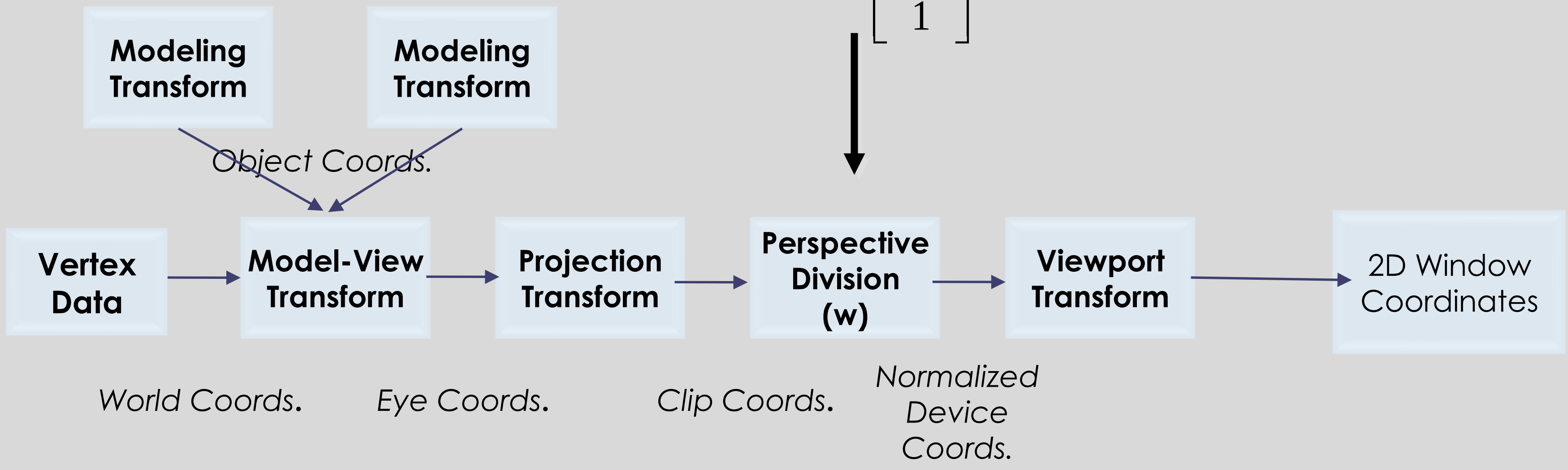
Flip ***z*** to transform to a left handed co-ordinate system $\Rightarrow$ increasing *z* values mean increasing distance from the viewer.

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



Perspective Projections Details

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



Perspective Projections

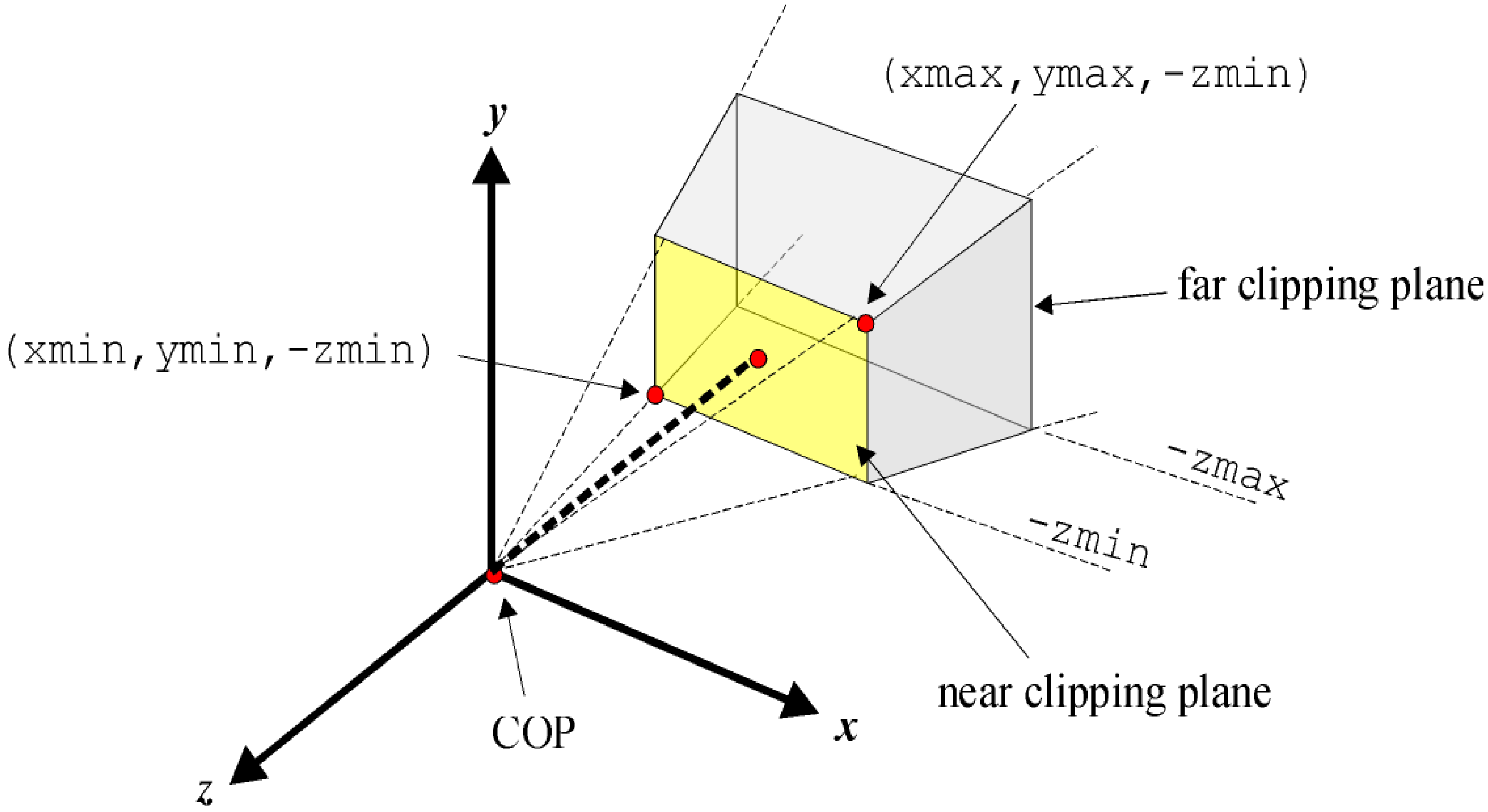
- $(x, y, z) \rightarrow (x_p, y_p, z_p)$
- Although perspective transformations preserve lines, it is not affine!
- Also, it is irreversible
  - All points along a projector project onto the same point, we cannot recover a point from its projection

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

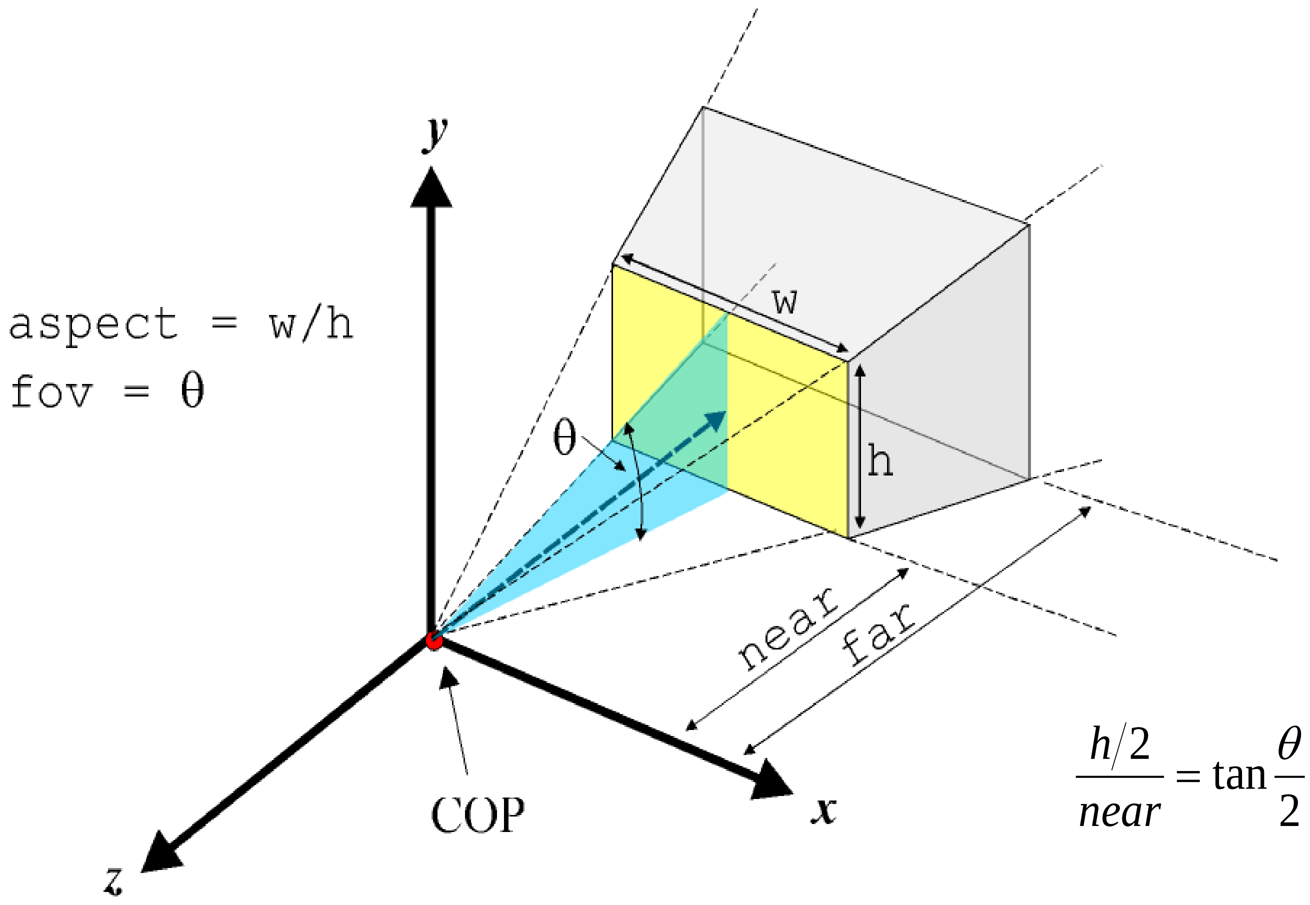
- It is not necessary to have a *symmetric frustrum* like:

```
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);
```

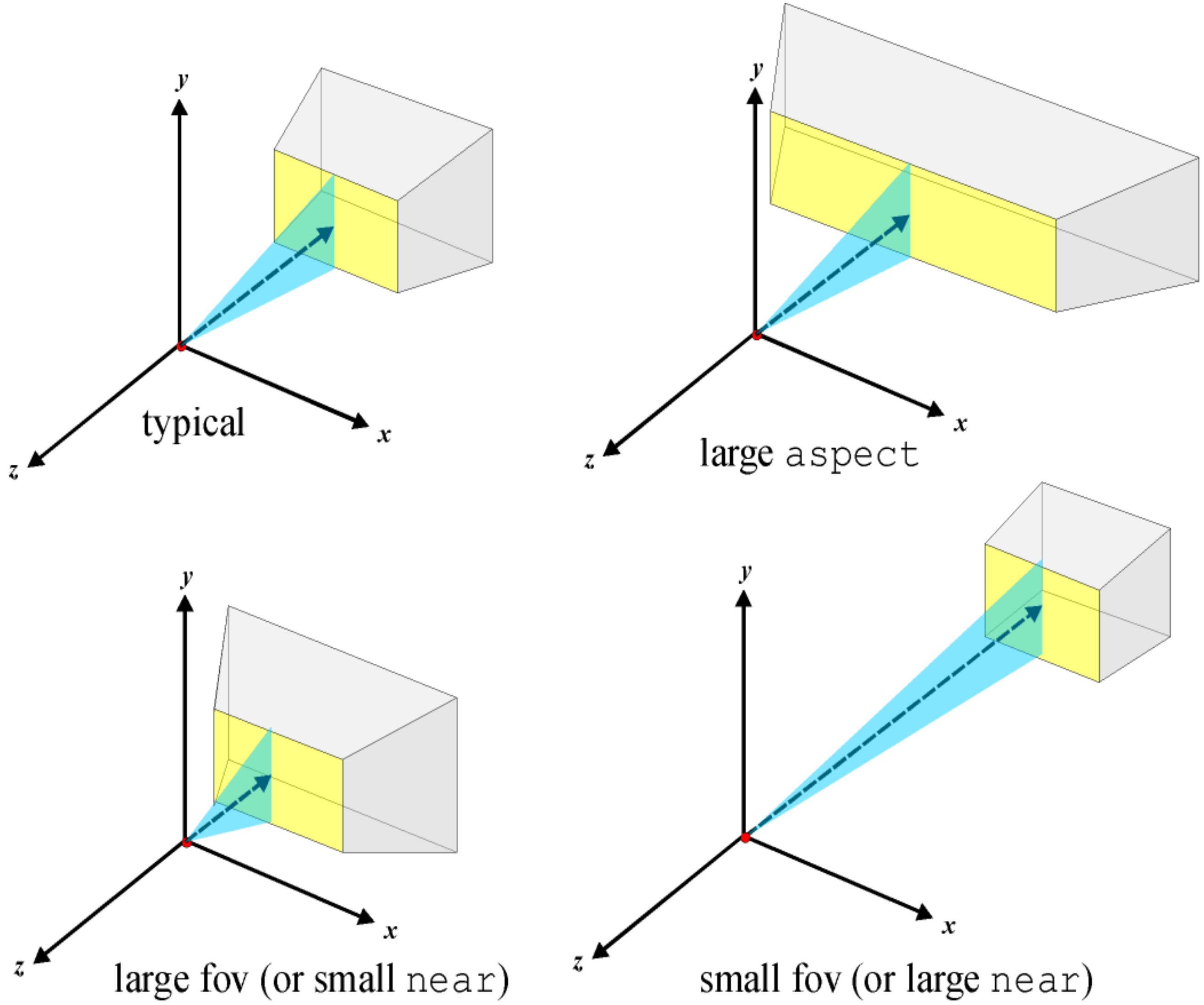


$$\frac{h/2}{near} = \tan\frac{\theta}{2} \Rightarrow h = 2\,near\tan\frac{\theta}{2}$$

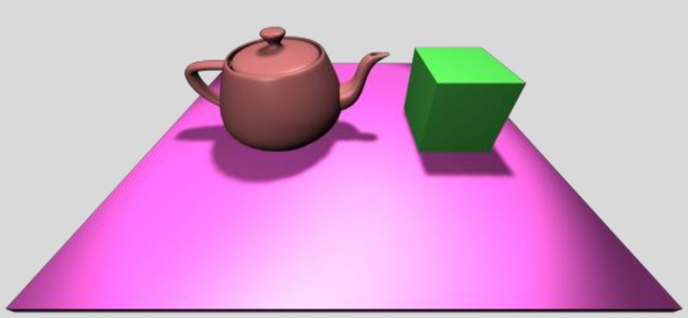
Perspective Matrix

- 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.

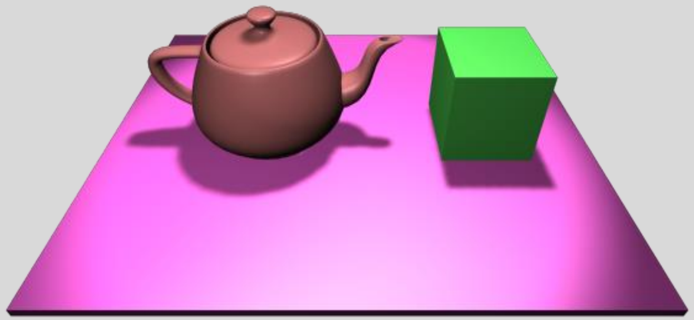
Perspective Projections



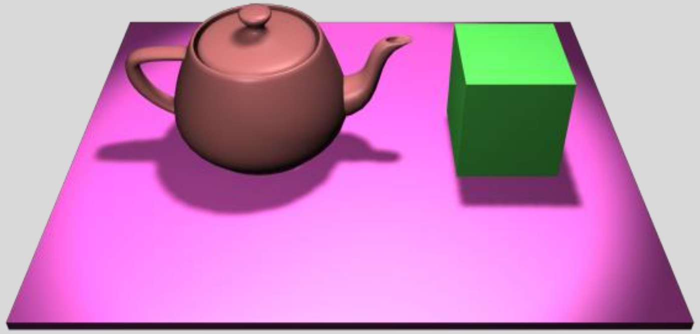
Lens Configurations



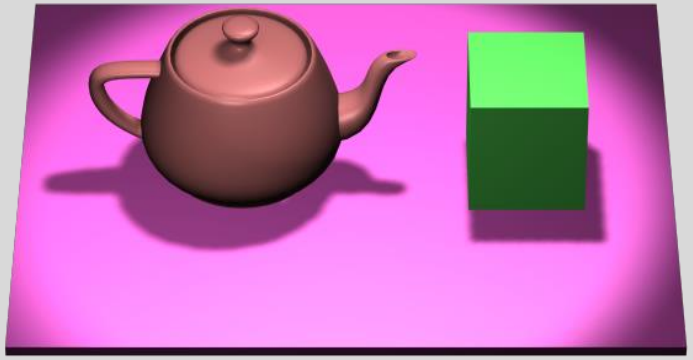
10mm Lens (fov = 122°)



20mm Lens (fov = 84°)



35mm Lens (fov = 54°)



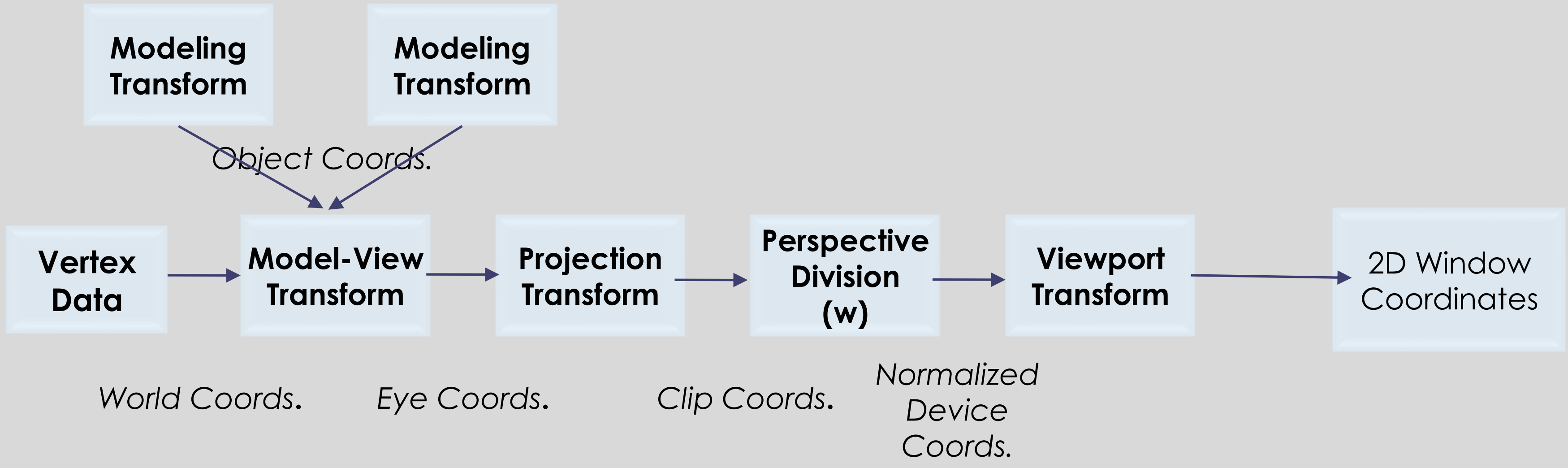
200mm Lens (fov = 10°)

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

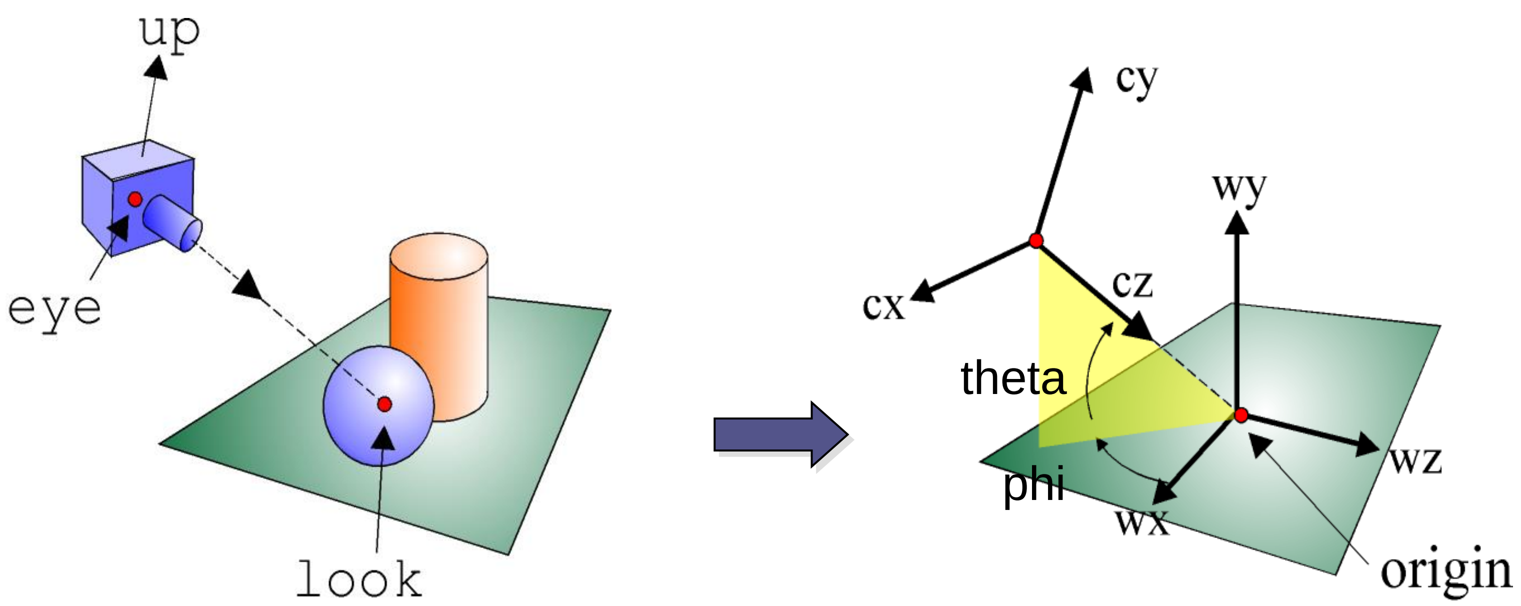


View Matrix

- 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;`

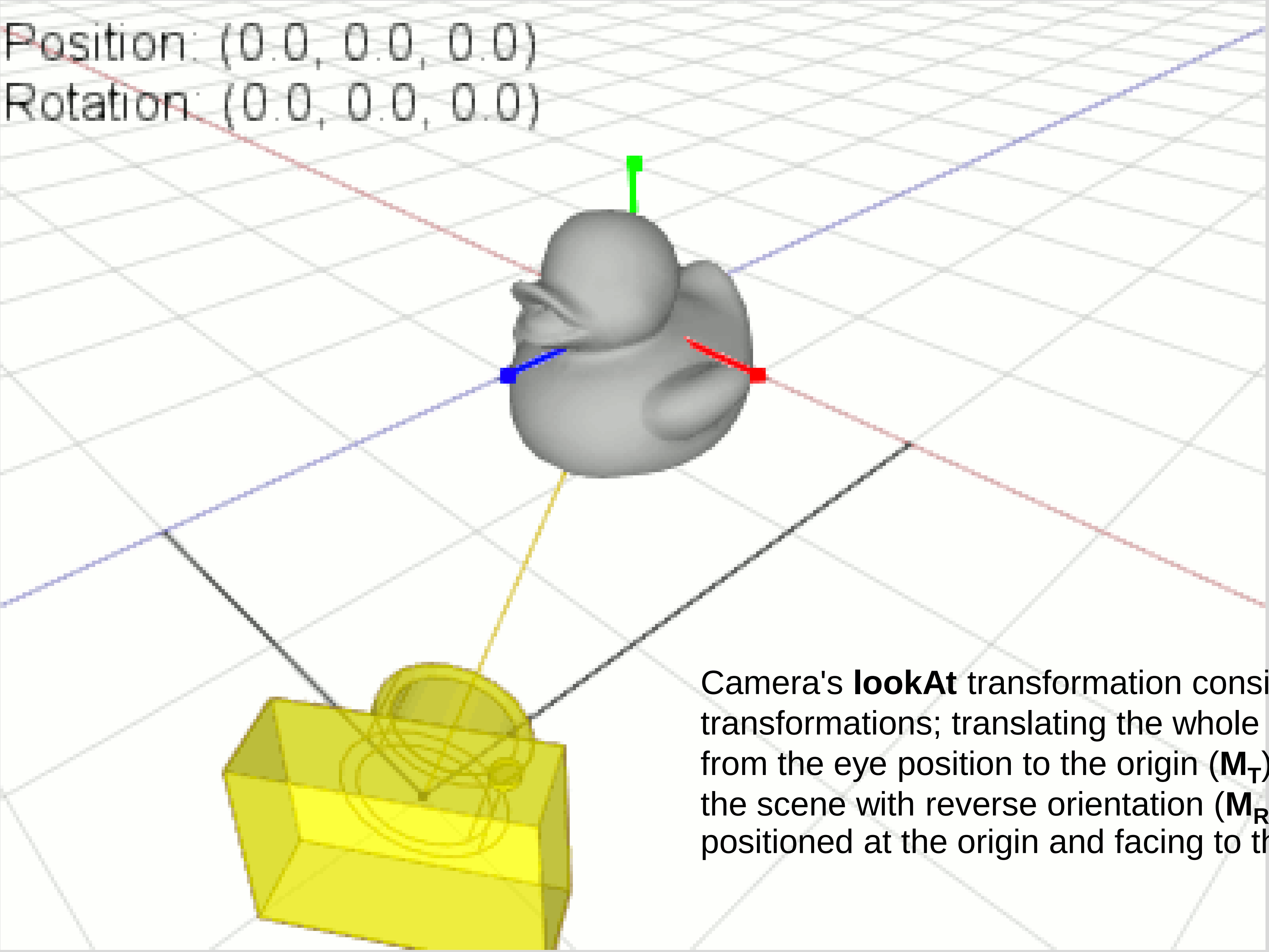
Positioning the Camera

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



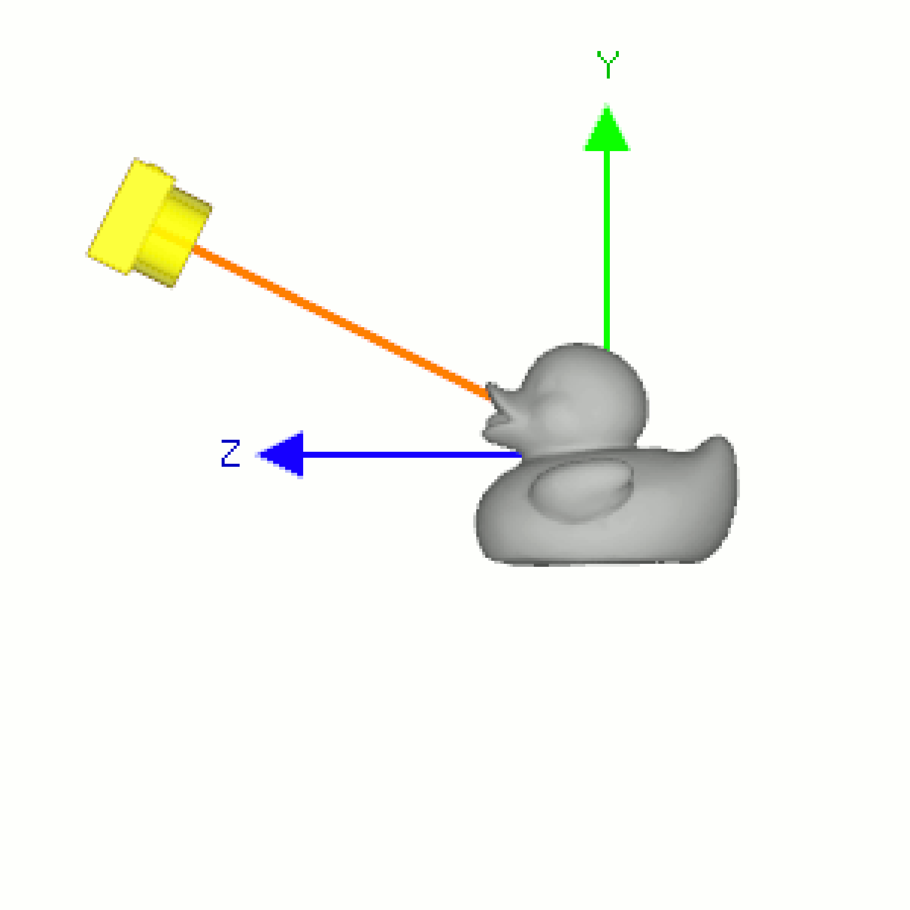
equivalent to:

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

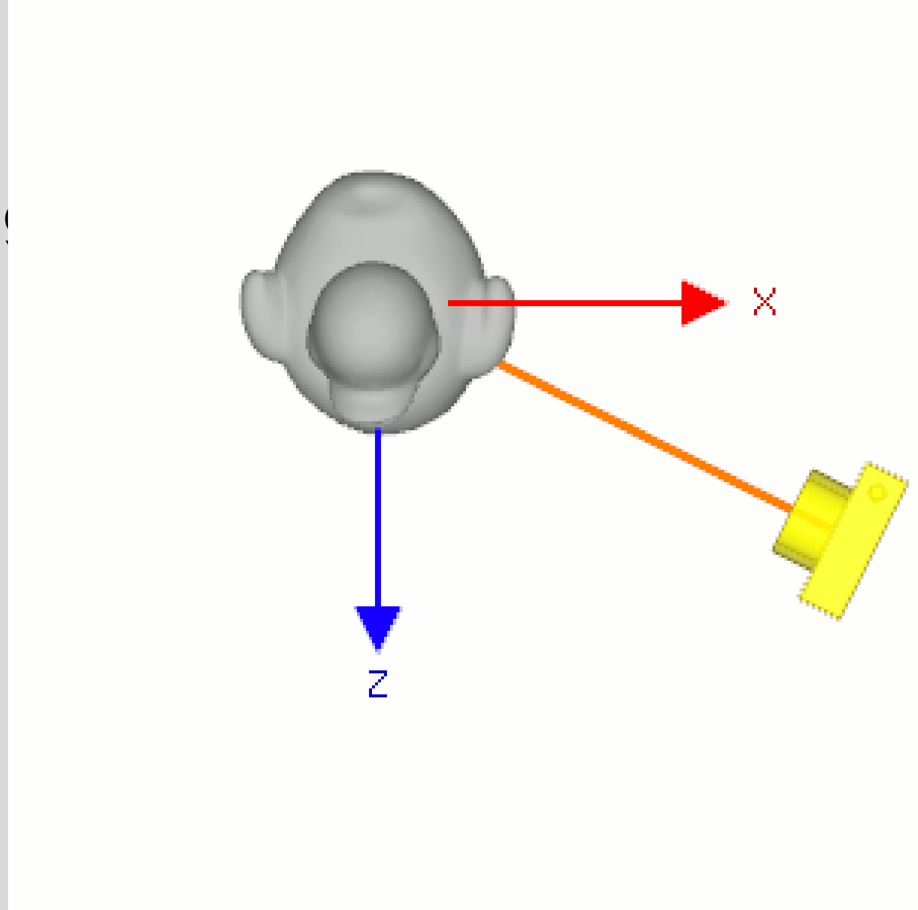


Camera's **lookAt** transformation consists of 2 transformations; translating the whole scene inversely from the eye position to the origin (**M_T**), and then rotating the scene with reverse orientation (**M_R**), so the camera is positioned at the origin and facing to the -Z axis.

http://www.songho.ca/opengl/gl_camera.html#lookat

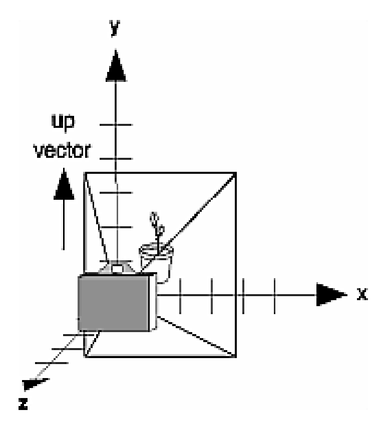


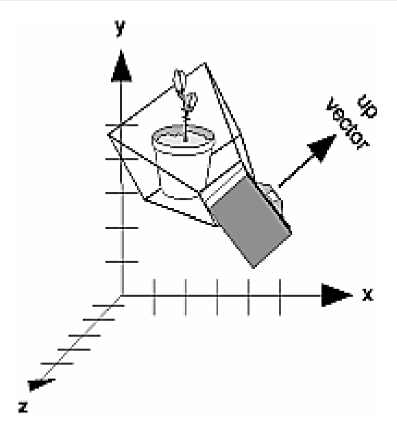
openg



Up Vector

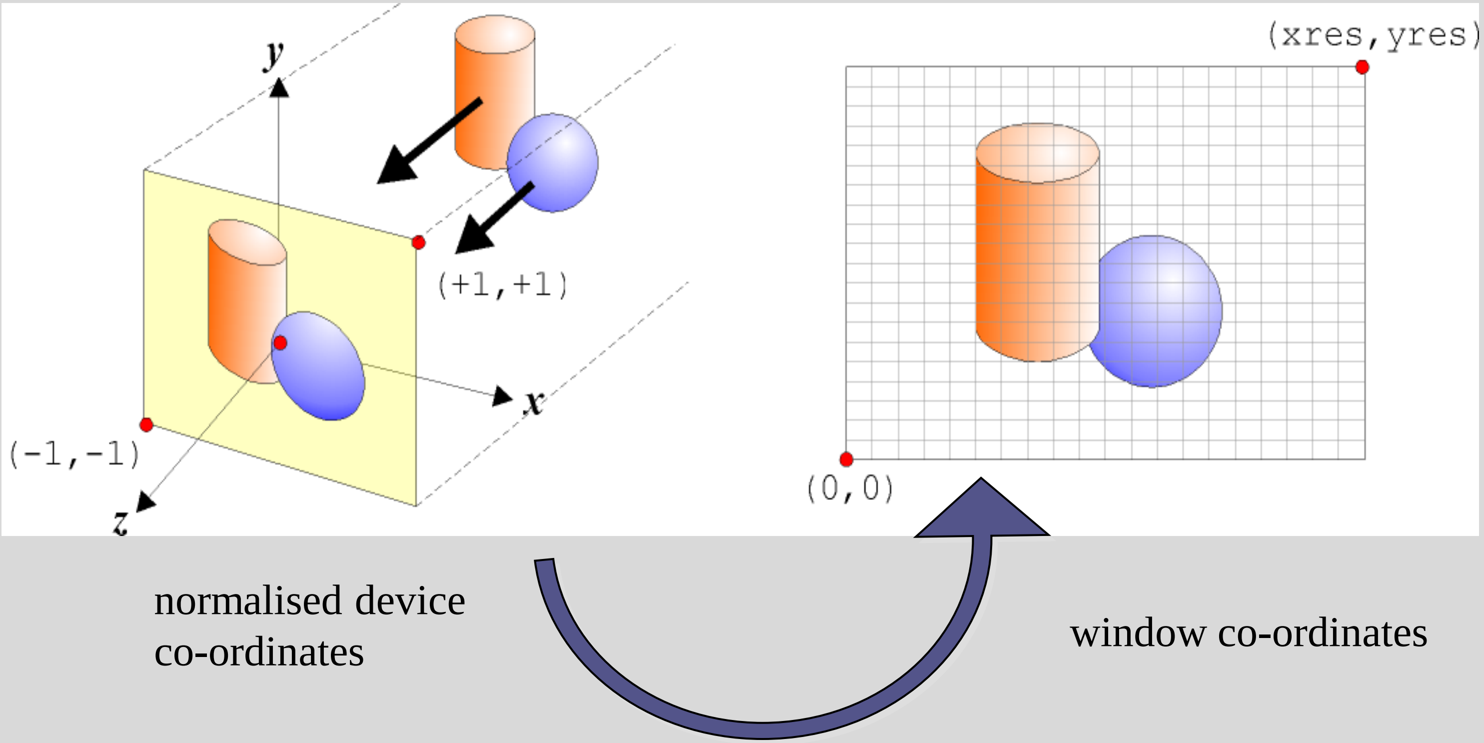
- Up vector
  - Perpendicular to the line of sight
  - Must not be parallel
  - Tells which direction is up (i.e. the direction from the bottom to the top of the viewing volume)





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);
```

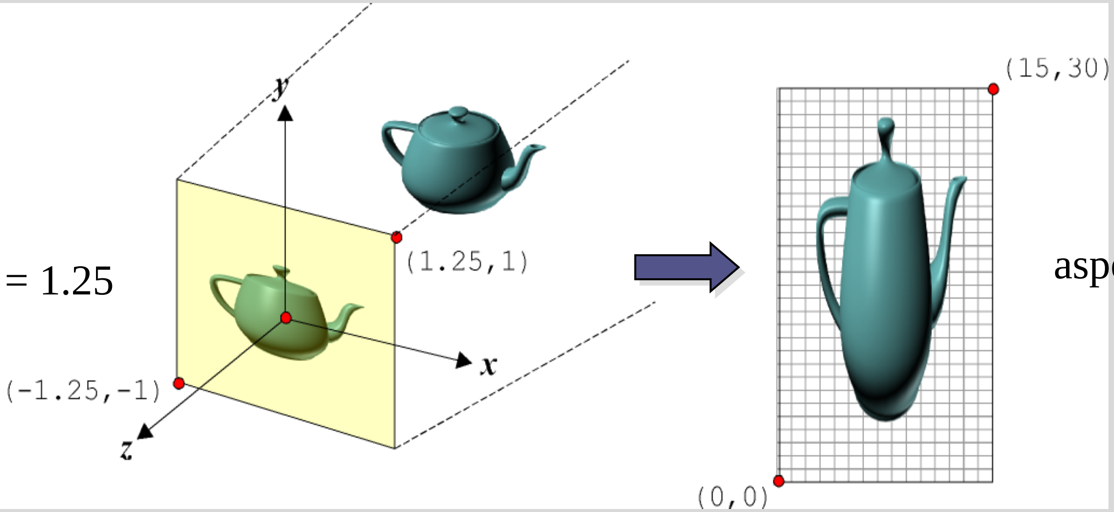
- `(x,y)` = location of bottom left of viewport within the window
- `width,height` = dimension in pixels of the viewport ⇒

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

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



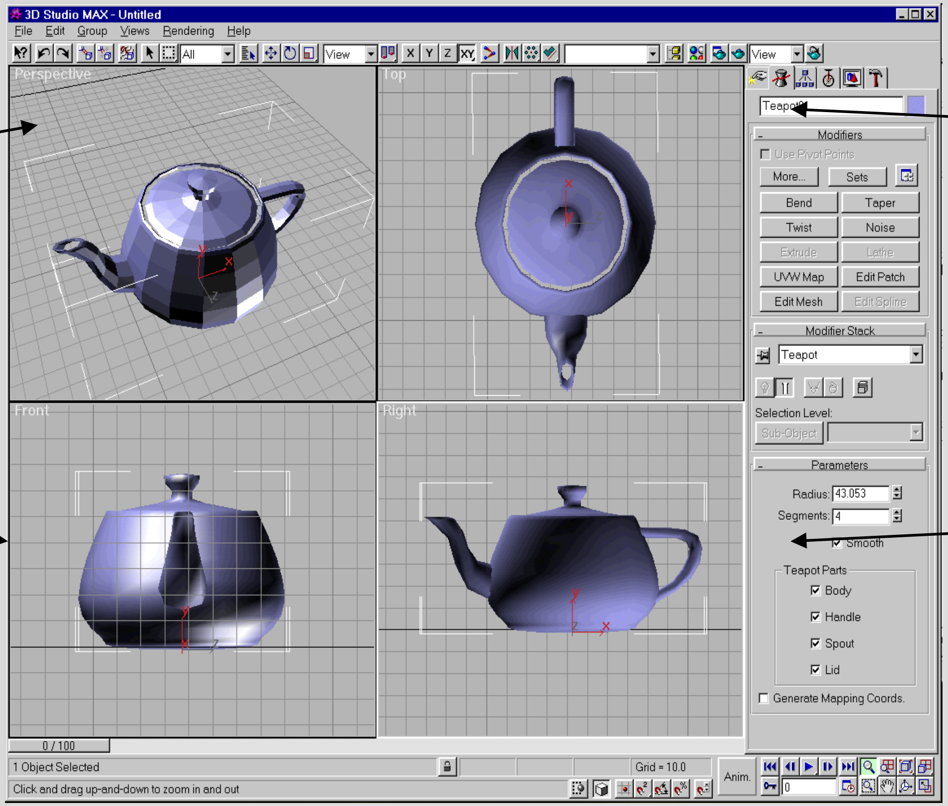
aspect ratio = 0.5

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

Perspective

Top



Front

Right

```
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();
}
```

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"
- Nice video tutorial on creating a camera in OpenGL, by Jamie King
  - https://www.youtube.com/watch?v=zHlxQoJYUhw
- 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