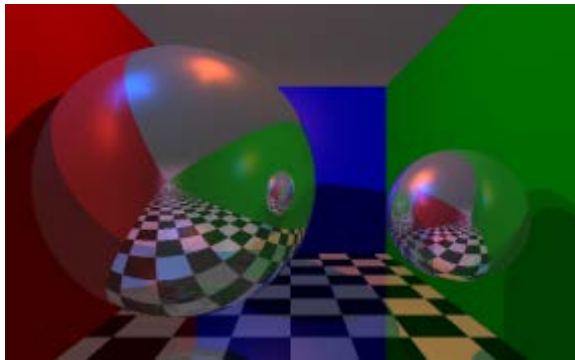
CSI 4105 Computer Graphics

Spring 2017

# Lecture 5: Viewing

Seon Joo Kim
Yonsei University



# Objectives

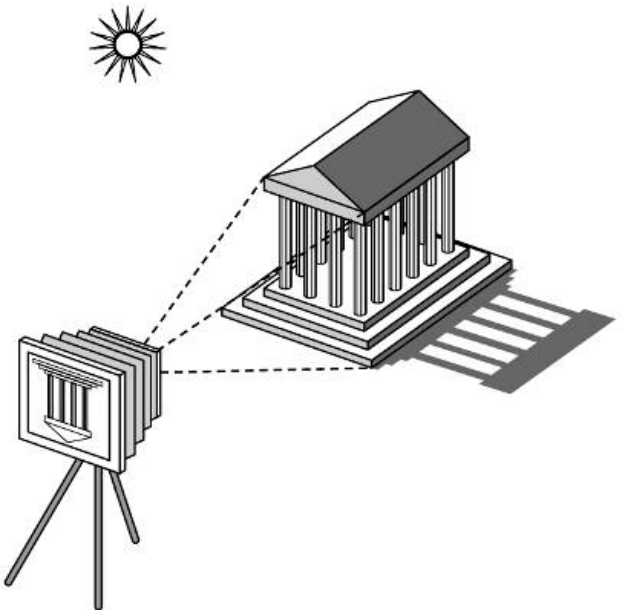
- First, discuss “elements of image” formation
- Next, learn how to specify the camera’s position and orientation
  - We call this the “View transformation”
- Discuss projection methods
  - Orthographic
  - Perspective
- Discuss relevant OpenGL commands

# Elements of Image Formation

- Objects ✓
  - We have learned how to place geometry into the world coordinate frame (Model transformations)

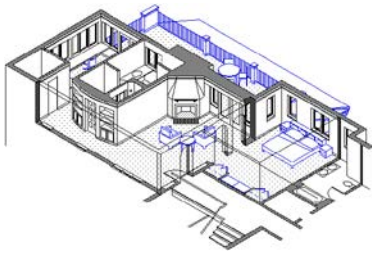
- How do we make the final 2D image on our screen?
  - **This lecture**

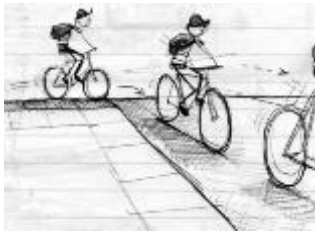
- What about lighting?
  - Light source(s)
  - Object Materials
  - **Next lecture**

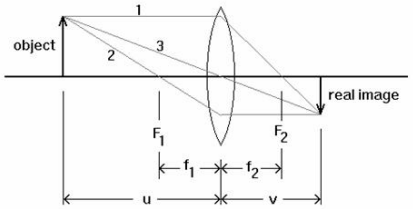


# Image formation in other disciplines

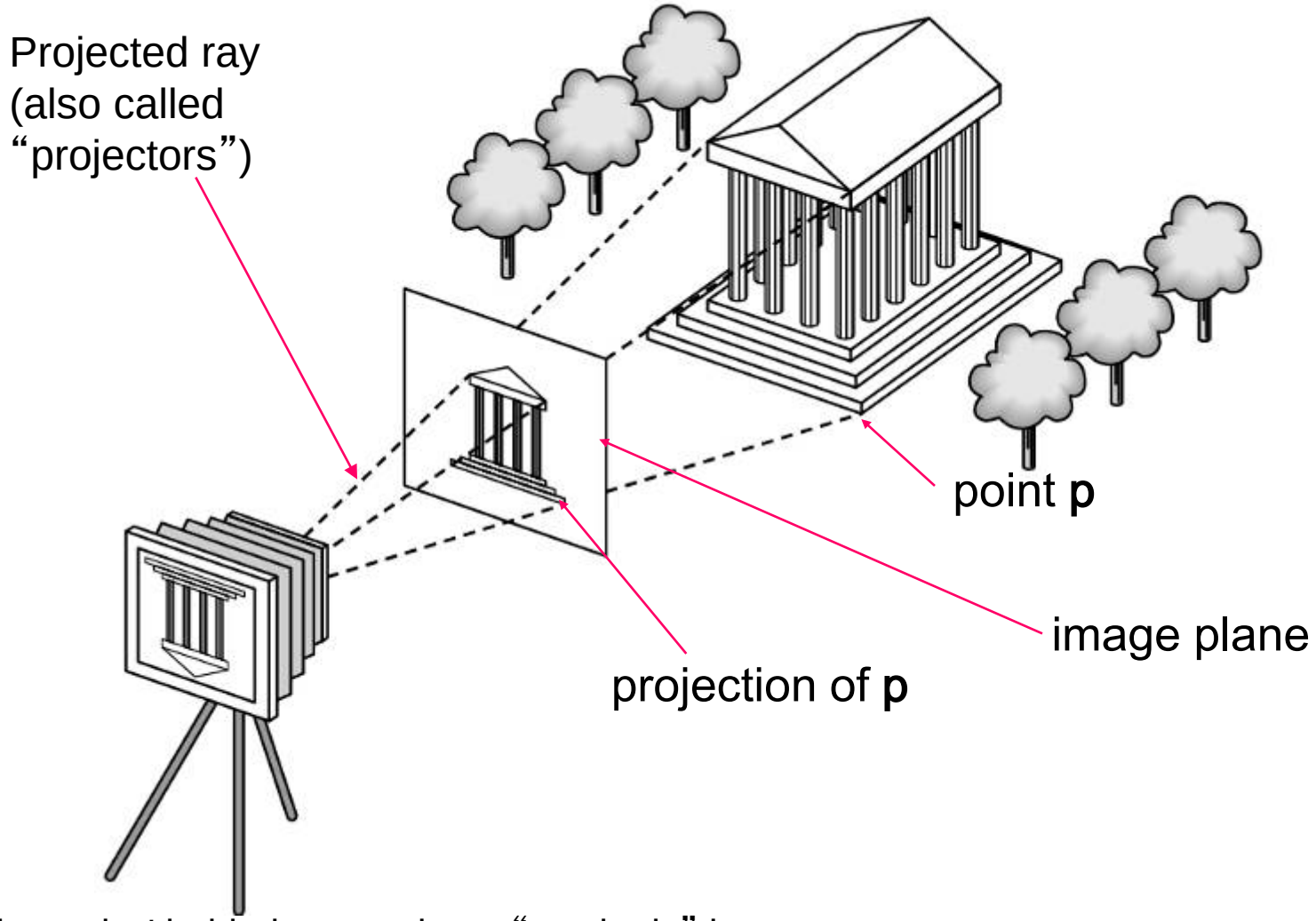
- **Architects**
  - Produce 2D drawings of 3D models
  - Very similar to graphics
- **Artists**
  - Understand the laws of perspective projections
- **Physics/Optics**
  - These are the serious guys, we are just little kids compared to them
- **Map makers (Cartography)**
  - Produce 2D images of 3D objects, but in a different manner





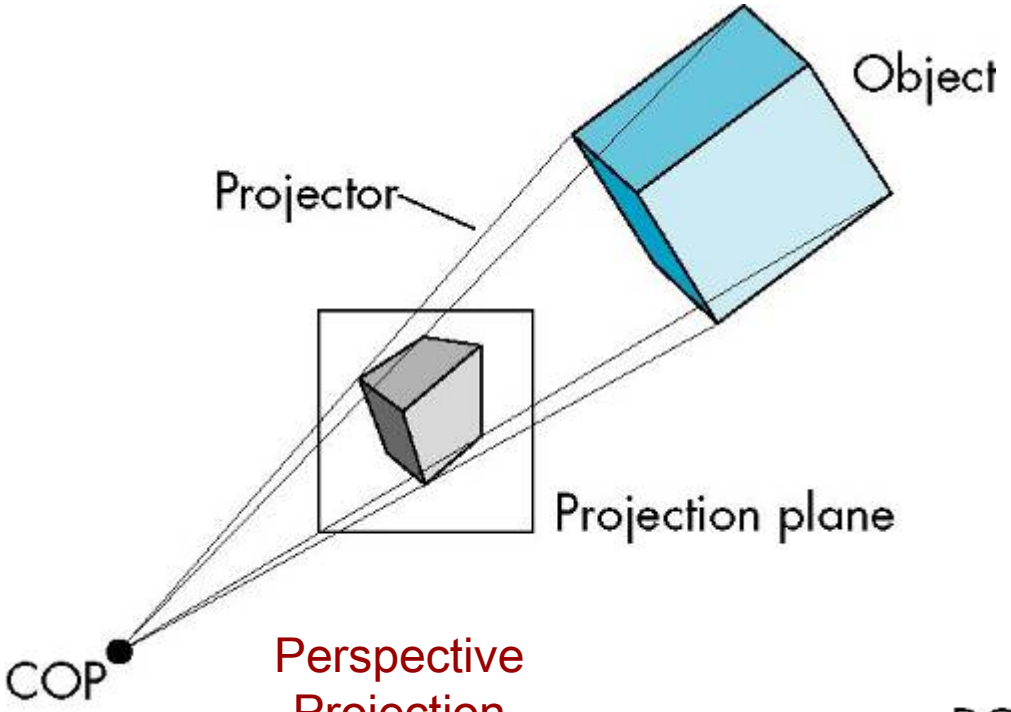



# Synthetic Camera Model



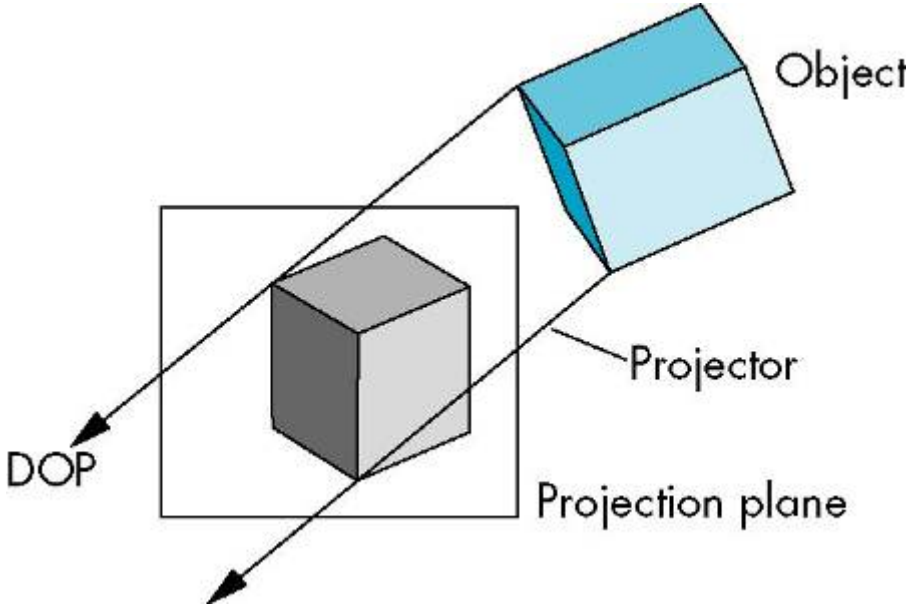

The mathematics' behind generating a “synthetic” image come from the basics of optics and how a camera works. So, we call often call this a “synthetic camera”.

# Perspective versus Parallel




Perspective Projection

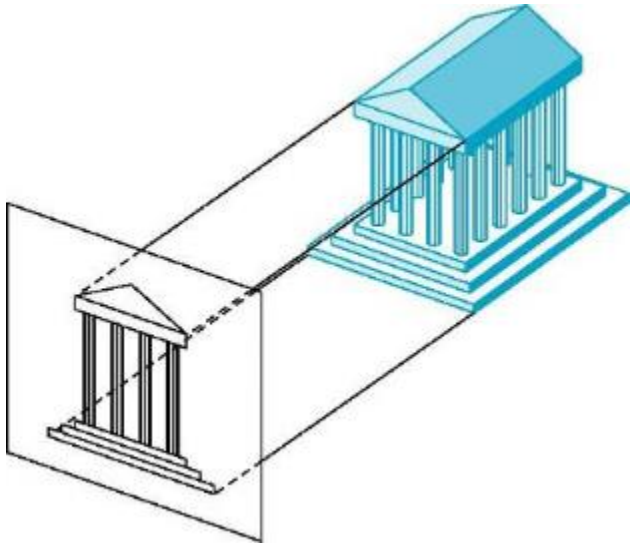
Parallel Projection




We distinguish two types of projections: 1) “perspective” where all rays pass through a “center of projection”; and 2) “parallel” where all projected rays are parallel to each other.

# Orthographic Projection

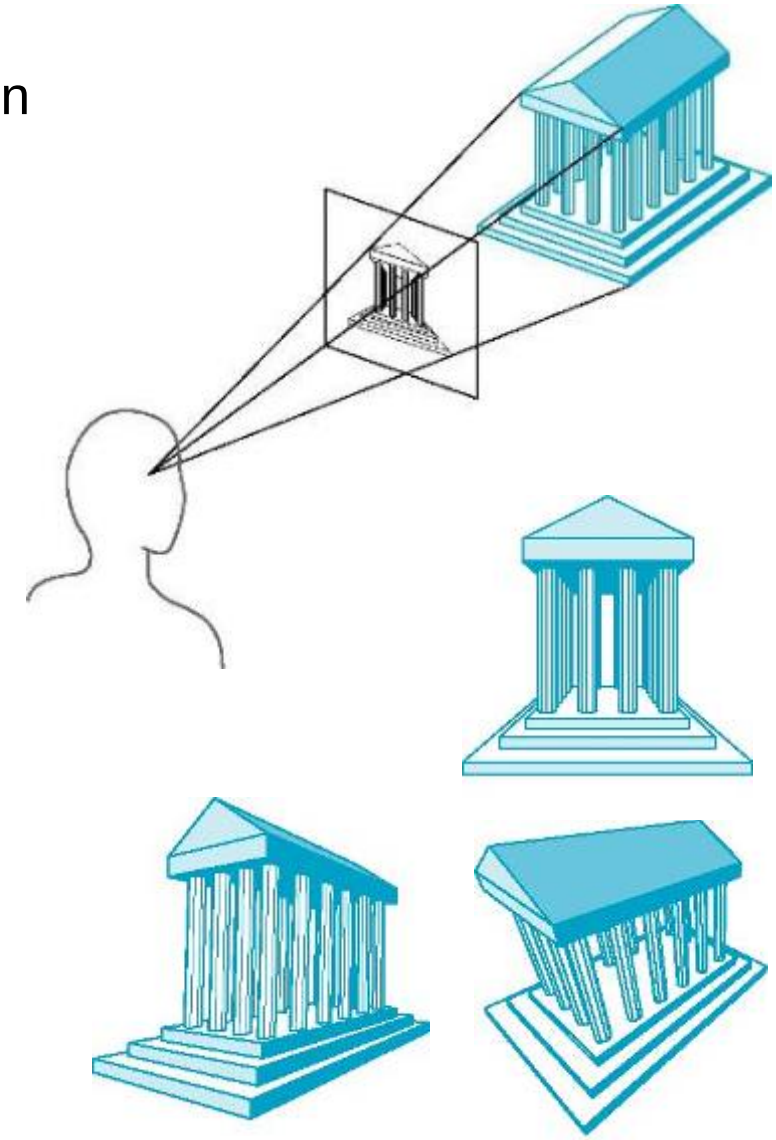
- A special case of parallel projection
  - Projectors are orthogonal to projection surface
- Preserves both distances and angles
  - Shapes preserved
  - Can be used for measurements
    - Building plans
    - Manuals



- So, far this is what we have been using in our class.

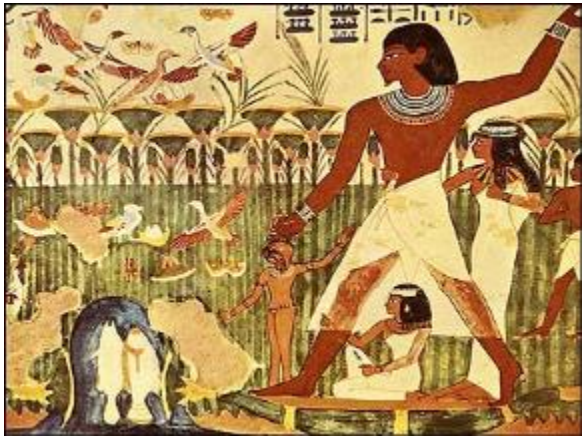
# Perspective Projection

- Projectors converge at center of projection
- Objects further from viewer are projected smaller than the same sized objects closer to the viewer
  - This looks more realistic, our eyes (and most cameras) work in a similar fashion.
- Equal distances along a line are not projected into equal distances (*foreshortening*)
- Angles preserved only in planes parallel to the projection plane



# Some interesting facts to impress your friends

- Early artist didn't understand perspective



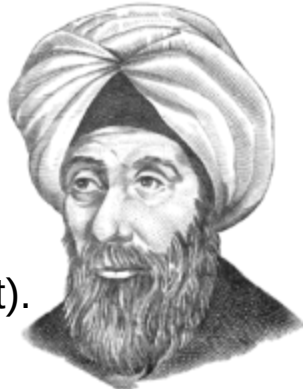
Scenes are drawn flat.

Attempts to simulate perspective – closer things are smaller, but perspective is incorrect.

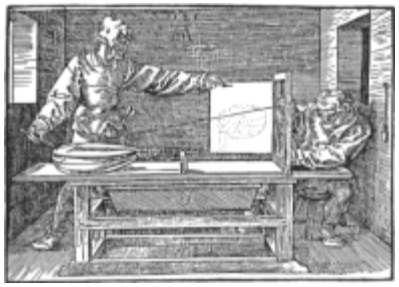
# Some interesting facts to impress your friends

- Artists had to learn perspective

~1000AD -- Alhazen Iraqi mathematician writes a book on the optics of light projection and the eye (but doesn't translate to art).



1400-1500AD westerners begin to learn perspective for 2D drawing.

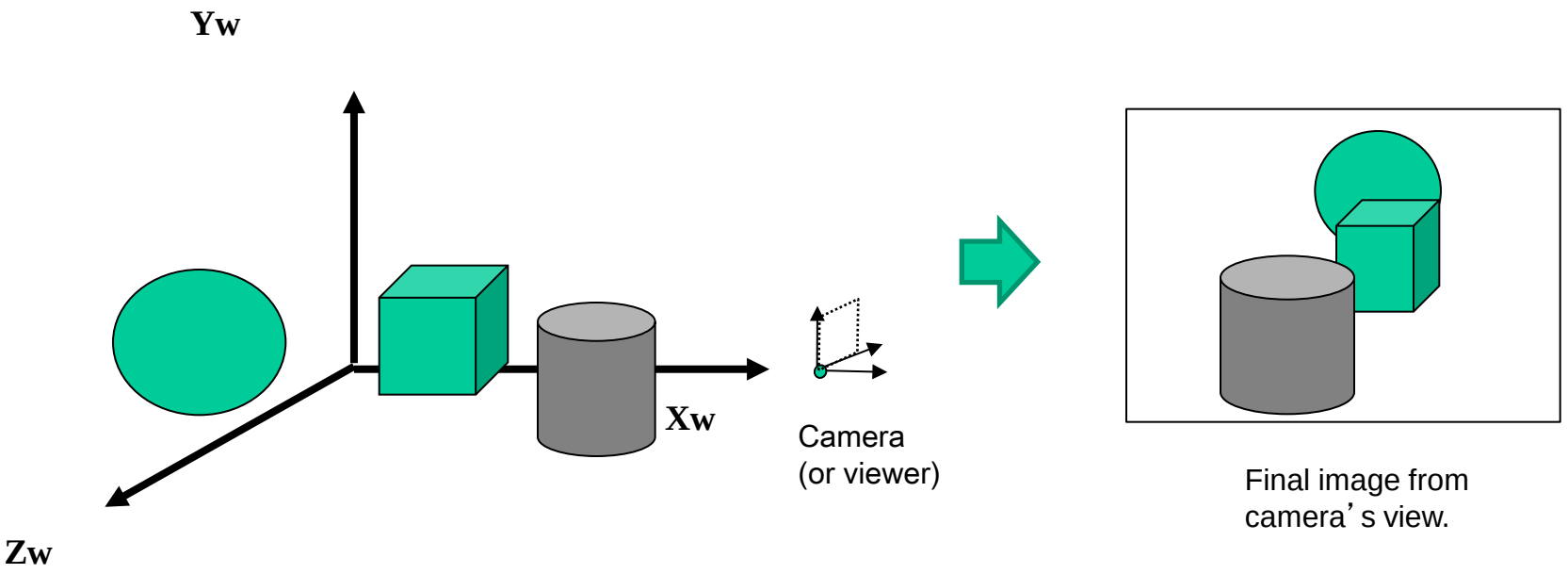


Writing & decorations on Taj Mahal get bigger the higher they are. This was purposely done to compensate for perspective distortion perceived by the viewer on the ground.

Michelangelo "David" head is bigger to compensate for the perspective distortion due to the size of the statue and the viewer being so low on the ground.

# Computer Viewing

# “Viewing in your world”




You are going to create a 3D world full of “awesome” 3D models and now you want to produce an “image” of your world at some location within the world. This is analogous to placing a “camera” (or viewer) in your world and taking a picture.

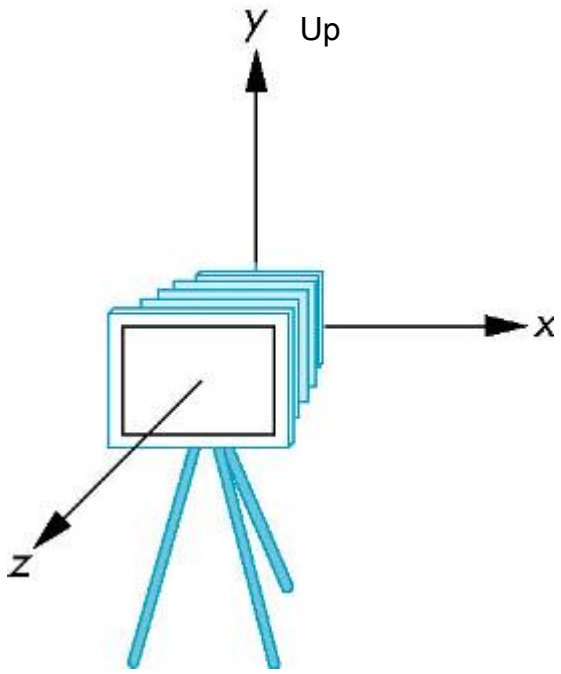
# Computer Viewing

- There **are three aspects** of the viewing process, all of which are implemented in the rendering pipeline
  - Positioning the camera*
    - Setting the “viewing transform”
  - Selecting the type of projection we want
    - Setting the projection matrix
      - Perspective or orthographic
  - Clipping
    - Setting the view volume
      - Only some part of the world appears in viewport

* We call this position the “camera” or the “viewer”.

# The OpenGL Camera

- The camera has a local coordinate frame, called the *camera coordinate frame*
  - Located at the origin
  - The camera looks in negative $z$ direction
  - $+y$-axis is the "*up-vector*"
- All projections are w.r.t. the camera frame
- Initially the world and camera frames are the same
  - Default model-view matrix is an identity

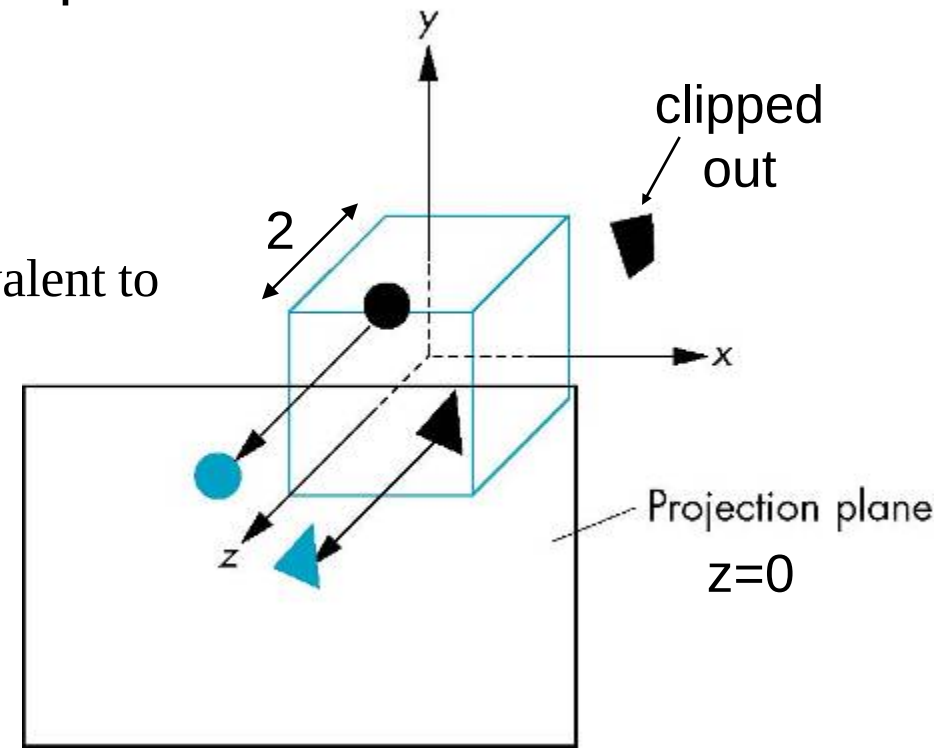


# Default Projection

- OpenGL also specifies a default view volume that is a cube with sides of length 2 centered at the origin
  - Default projection matrix is an identity
- Default projection is orthographic

Note that the default projection is equivalent to

```
glOrtho( -1.0, 1.0,
         -1.0, 1.0,
          1.0, -1.0 );
```

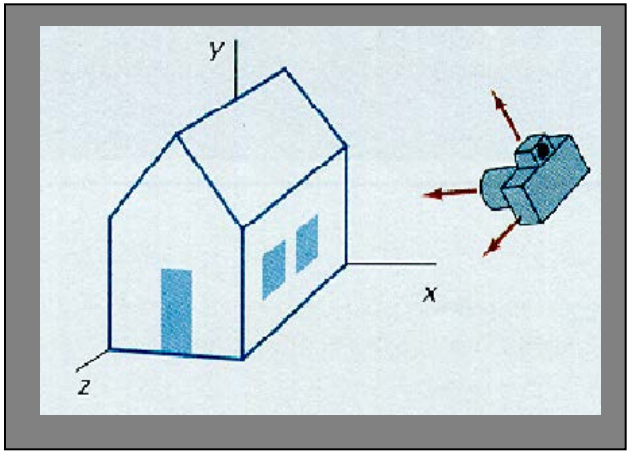


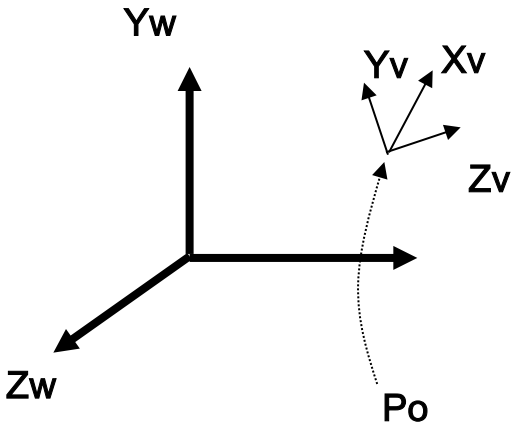
# View Transformation

## Positioning the Camera

# Specifying the Camera in the "World"

- What do we need to know about the "camera"?

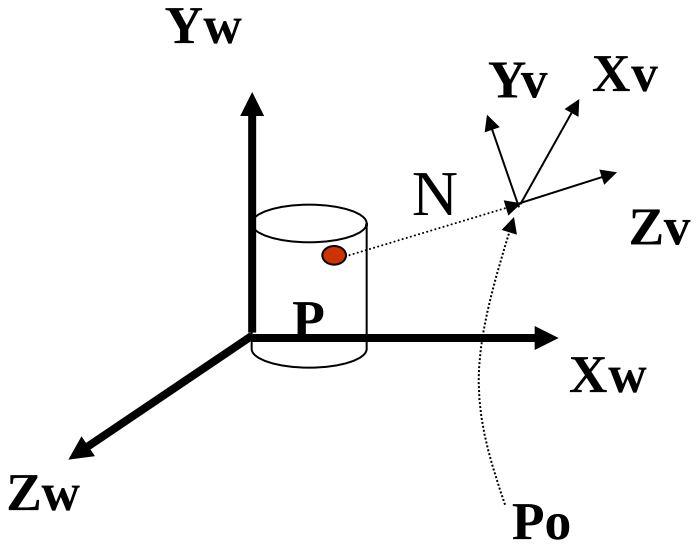



- Position in 3D space ($P_0$)
  - **View Reference Point (or "eye"location)**
  - **This one is easy – just give a point in the 3D world**
- Orientation in 3D Space (Some rotation of the axes)
  - "**Viewing-coordinate system**" or "**View reference coord. frame**."
  - Xv, Yv, Zv (see next slides show how to specify this axes)

# Specifying the View Plane Normal (Zv)

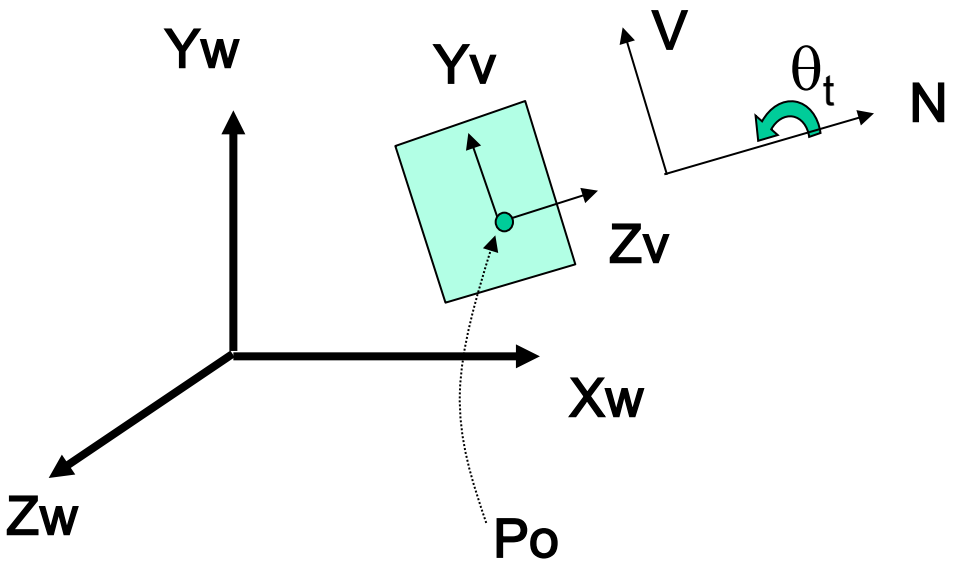
- **View-Plane Normal** Vector, **N**
- N, should be the direction for the viewing Zv axis
- One way is to specify this vector is by a look-at-point P (OpenGL approach)




$$N = (P_0 - P)$$

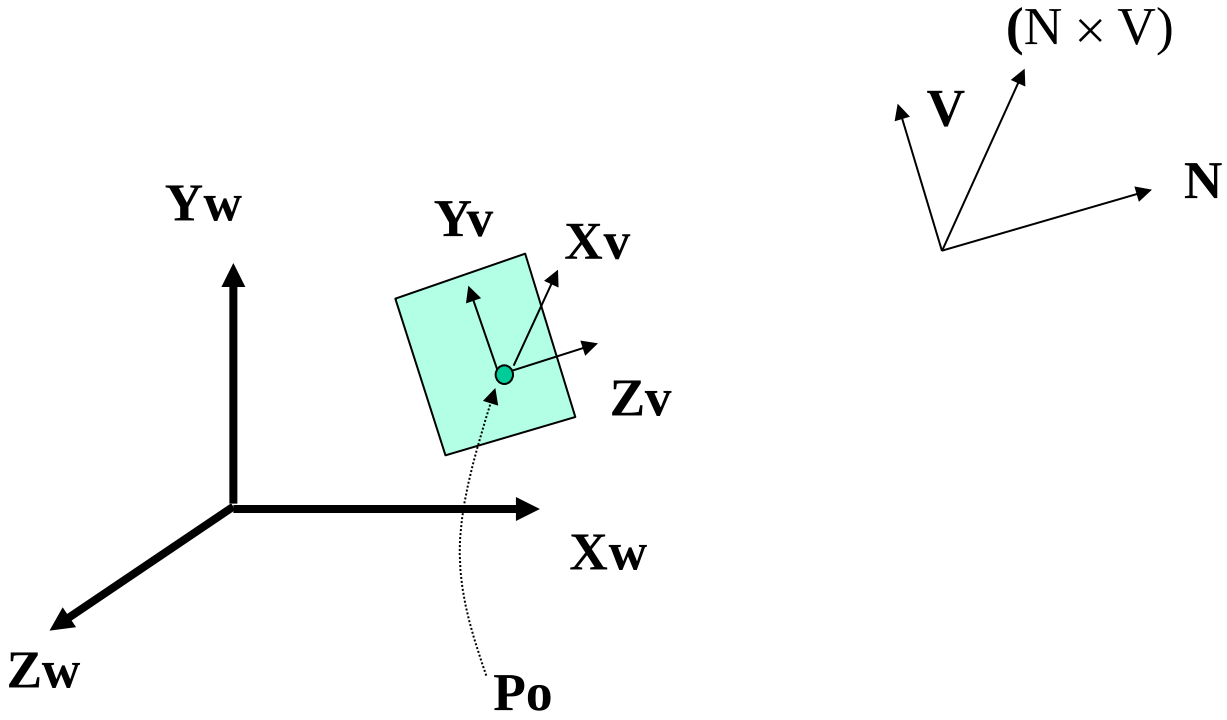
# Specifying the View-Up (Orientation of Yv)

- **View-up Vector**, **V**
- **V** is the positive direction of the Yv
- In OpenGL we specify a vector called “look up” which will be used to compute the real “V” or “Yv” (see slide 25)
- Another way is to specify an angle rotation about Zv (or twist angle $\theta_t$)



# (Orientation of Xv)

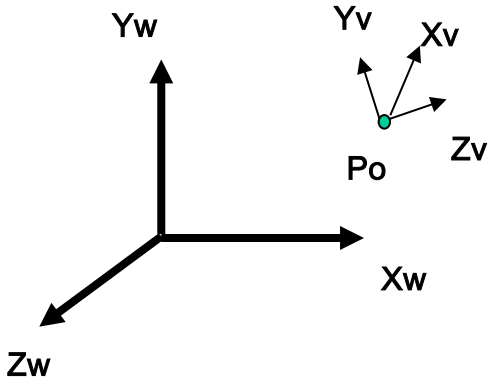
- **Xv can be computed from N and V**
- Cross product: Xv = N × V

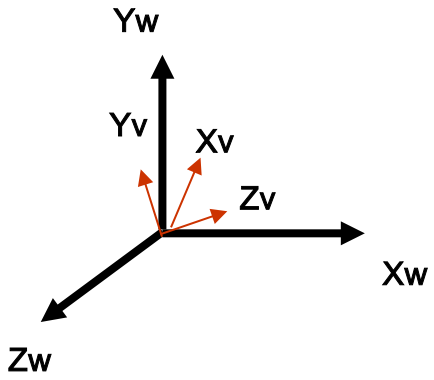



Before we can project objects in the world, we have to transform them into the view-reference coordinate frame.

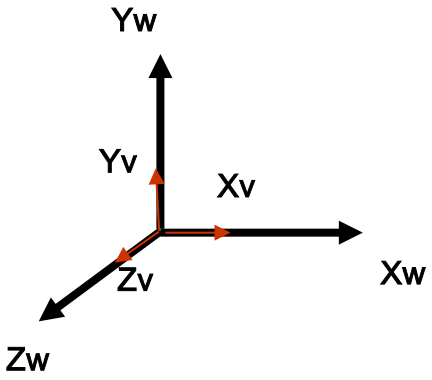
# The "Viewing Transform"

- We need to describe the world in terms of the viewing coordinates.
- This is equivalent to finding a transformation that superimposed the viewing reference frame onto the world reference frame.
- Steps
  - 1. Translate the view reference point to the original of the world-coordinate system.
  - 2. Apply rotations to align the Xv, Yv, and Zv axes with the world Xw, Yw, Zw axes




Step 1 (translate)

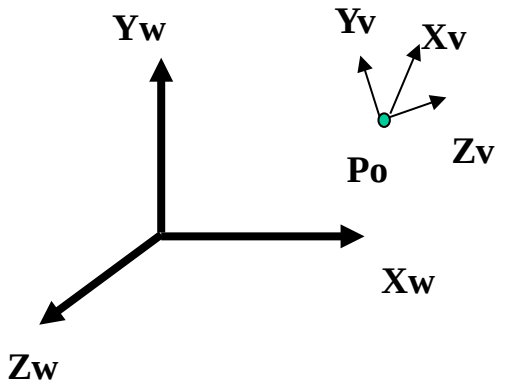


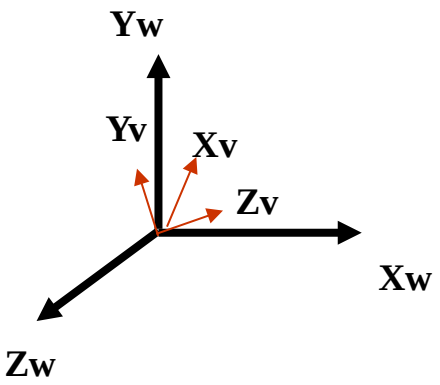

Step 2 (rotate/align axes)

# Step 1 (Translate)

- If the view reference point is specified at world position Po ($x_o$, $y_o$, $z_o$)
- Use a translation matrix

$$T = \begin{bmatrix} 1 & 0 & 0 & -x_0 \\ 0 & 1 & 0 & -y_0 \\ 0 & 0 & 1 & -z_0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$




Step 1 (translate)

# Step 2 (Align Axis)

- Could compute rotations $R_x(\alpha)\ R_y(\beta)\ R_z(\gamma)$ to align axes
- We can use the "change of basis" trick to compute this alignment
  - Just compute the unit vectors of (Xv, Yv, Zv) -- lets call them UVN
  - Remember, Zv = N and Yv = V (previous slides)

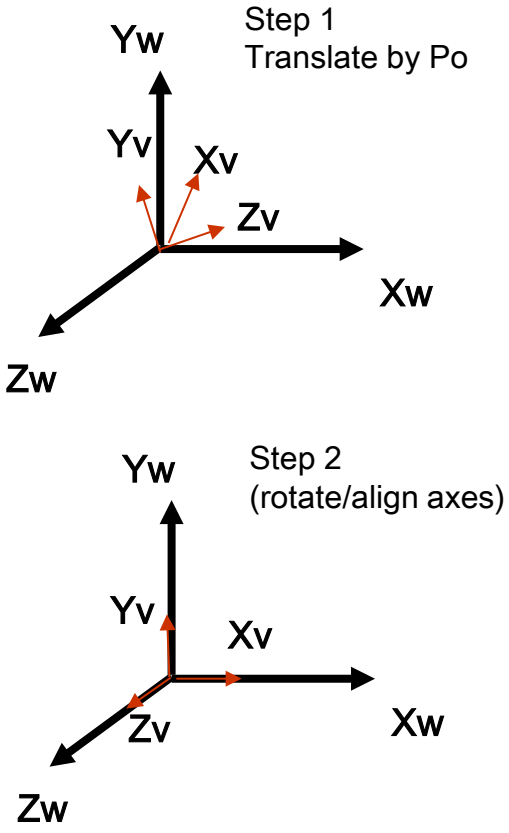
$$n = \frac{n}{|N|} = (n_1, n_2, n_3)$$

$$u = \frac{V \times N}{|V \times N|} = (u_1, u_2, u_3)$$

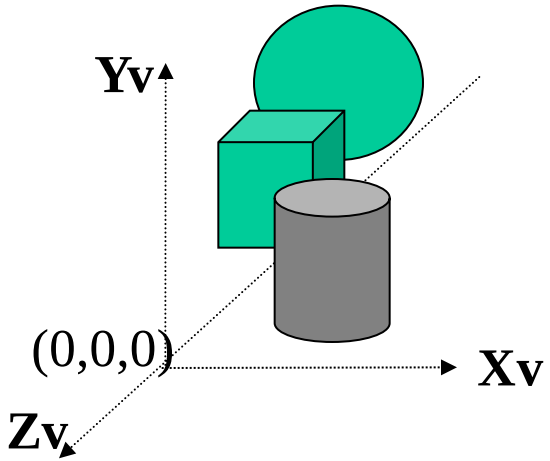
$$v = n \times u = (v_1, v_2, v_3)$$

$$R = \begin{bmatrix} u_1 & u_2 & u_3 & 0 \\ v_1 & v_2 & v_3 & 0 \\ n_1 & n_2 & n_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This matrix aligns points in the world frame with the camera's frame.



# NOTE: After Viewing-Transform




All 3D objects coordinates are now in the viewing coordinates!

Their (x,y,z) positions have been transformed.

(0,0,0) is now the origin of the viewing-coordinate-frame.

From this point on, you need to think in *this* coordinate frame and not in “world” coordinates!

# View Transformation in OpenGL

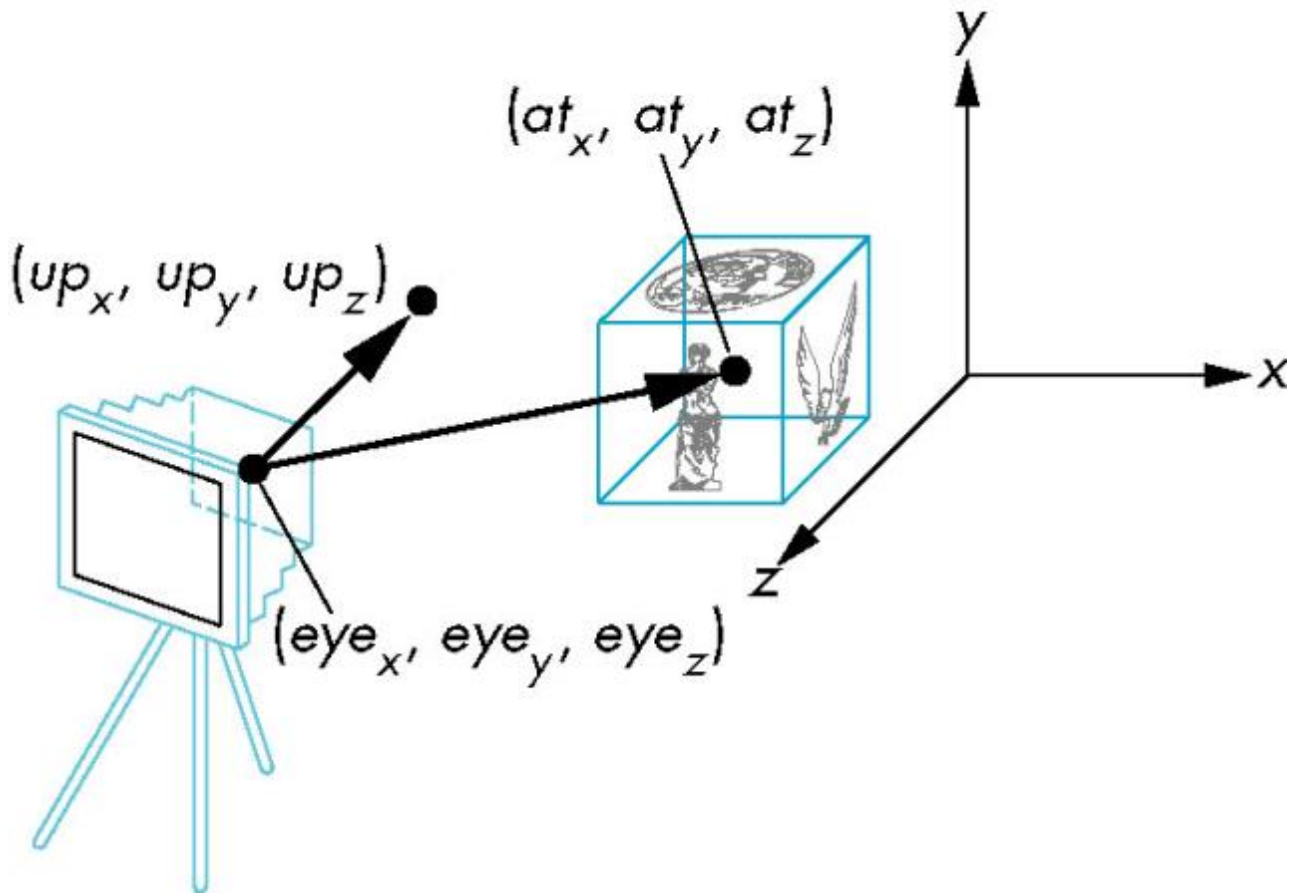
- In OpenGL, the view transformation matrix is normally the last transformation in the model-view matrix

```
glMatrixMode( GL_MODELVIEW ):
glLoadIdentity();
// specify view transformation matrix here;
...
```

- The GLU library contains the function **`gluLookAt()`** to form the required view transformation matrix through a simple interface
  - Conceptually, it positions the camera at the required location and orientation
  - Internally, it generates a view transformation matrix and post-multiplies it to the current model-view matrix

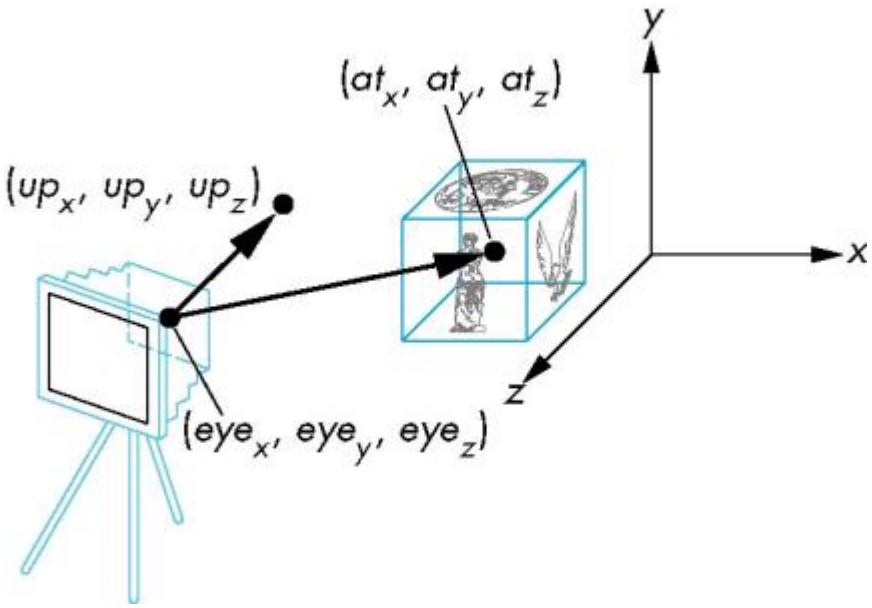
# Using the `gluLookAt()` Function

```
gluLookAt( eyex, eyey, eyez,
           atx, aty, atz,
           upx, upy, upz );
```

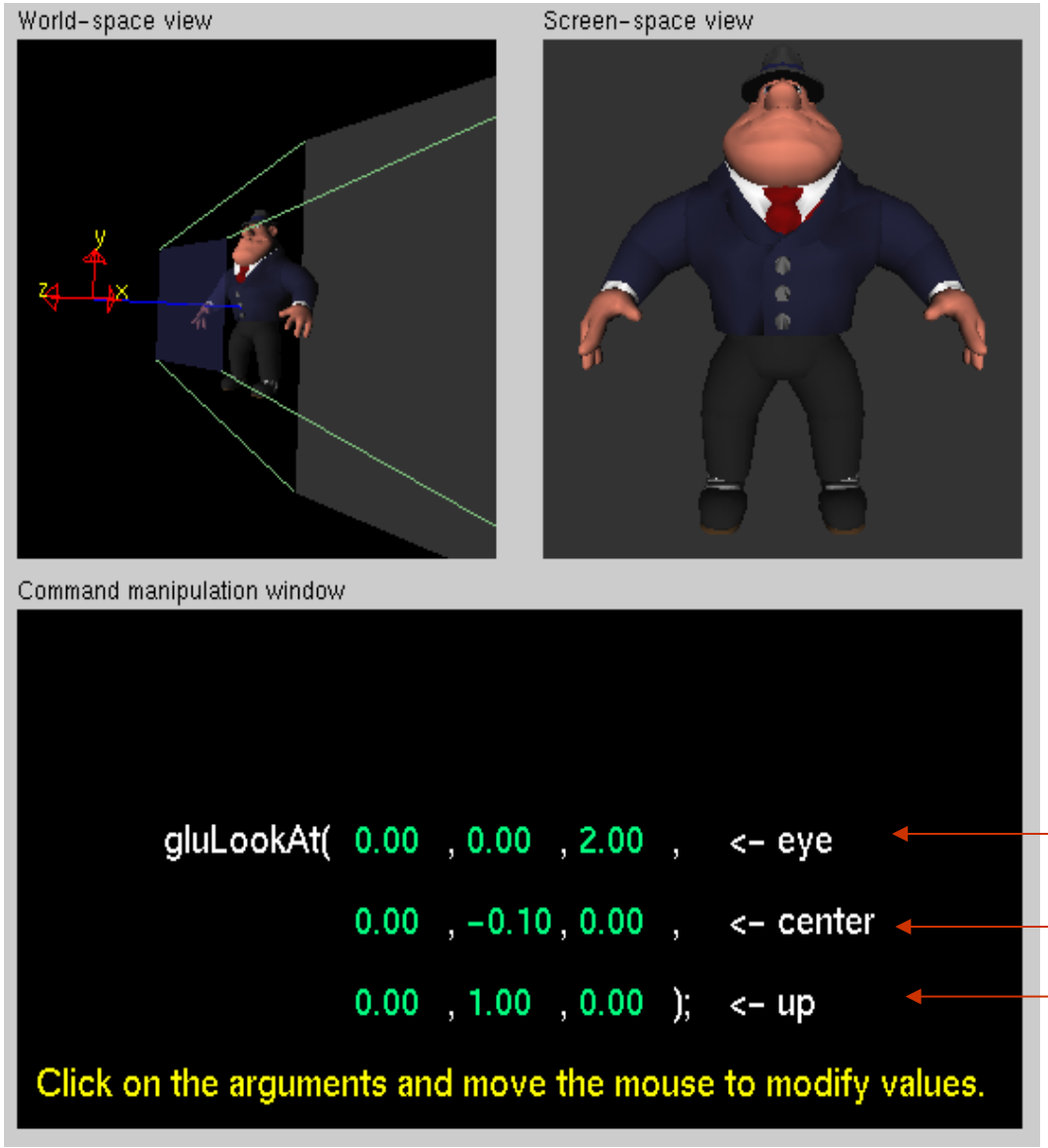


# The `gluLookAt()` Function

- Note that it does not directly specify the camera frame axes vectors **u**, **v**, **n**
- The "up-vector" may not be perpendicular to the view direction
- The vectors **u**, **v**, **n** can be derived as follows
  - **n** = normalize( **eye** – **at** )
  - **u** = normalize( **up** ) × **n**
  - **v** = **n** × **u**



# OpenGL viewing (see glTutors - Projection)




`gluLookat()`

sets the camera's position and orientation in the world.

Point **Po** (position of "camera)

Look-at-point **P**

Vector **V** (view-up)

Compute **N** using **Po** and **P**

# Projection

## Defining the View Volume

# OpenGL Projections

- In OpenGL, after a vertex is multiplied by the model-view matrix, it is then multiplied by the projection matrix
- The projection matrix is a 4x4 matrix that defines the type of projection
- The projection matrix can be specified by first defining a *view volume* (or *clipping volume*) in the camera frame
  - For orthographic projection, use **`glOrtho()`**
  - For perspective projection, use **`glFrustum()`**

Projections in OpenGL assume the world has already been transformed into the “camera’s” frame.

# OpenGL Projections and Normalization

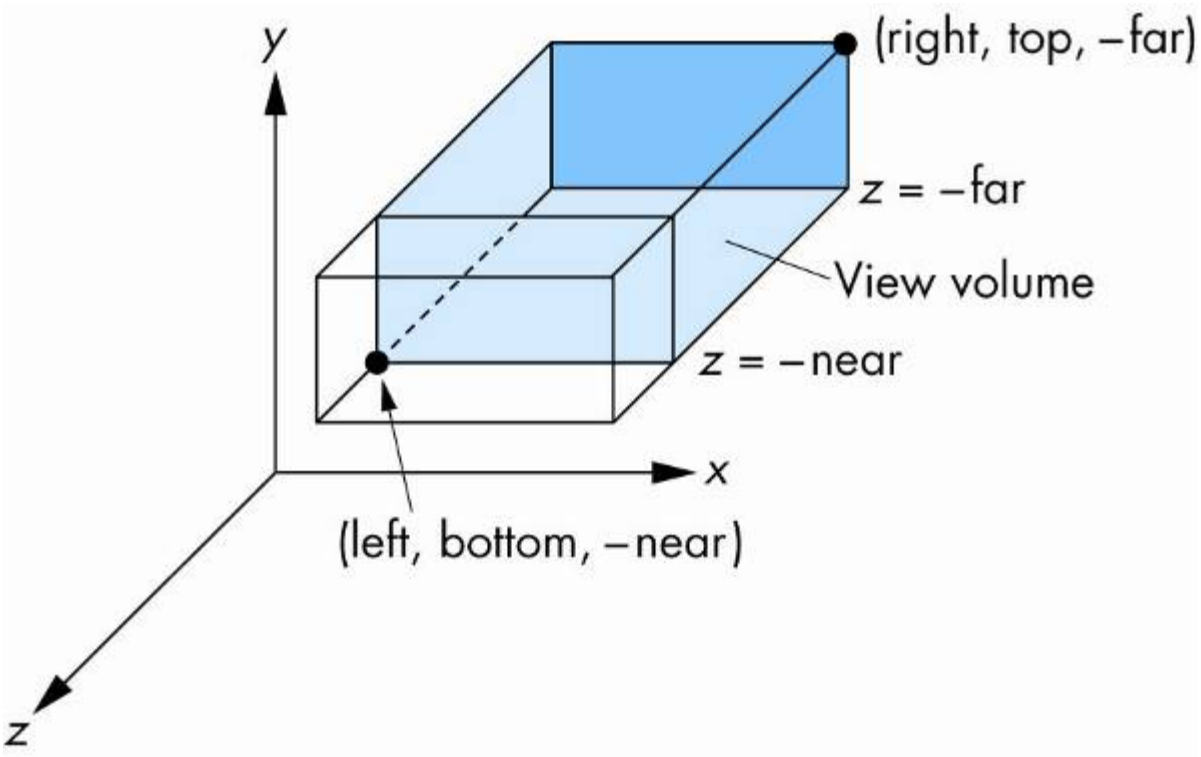
- A projection matrix is then computed such that it maps points in the view volume to a *canonical view volume*
  - The canonical view volume is the 2 x 2 x 2 cube defined by the planes $x = \pm 1$, $y = \pm 1$, $z = \pm 1$
  - Also called the *Normalized Device Coordinates* (NDC)
- The canonical view volume is then mapped to the viewport (*viewport transformation*)

# Orthographic Projection

# OpenGL Orthographic Projection

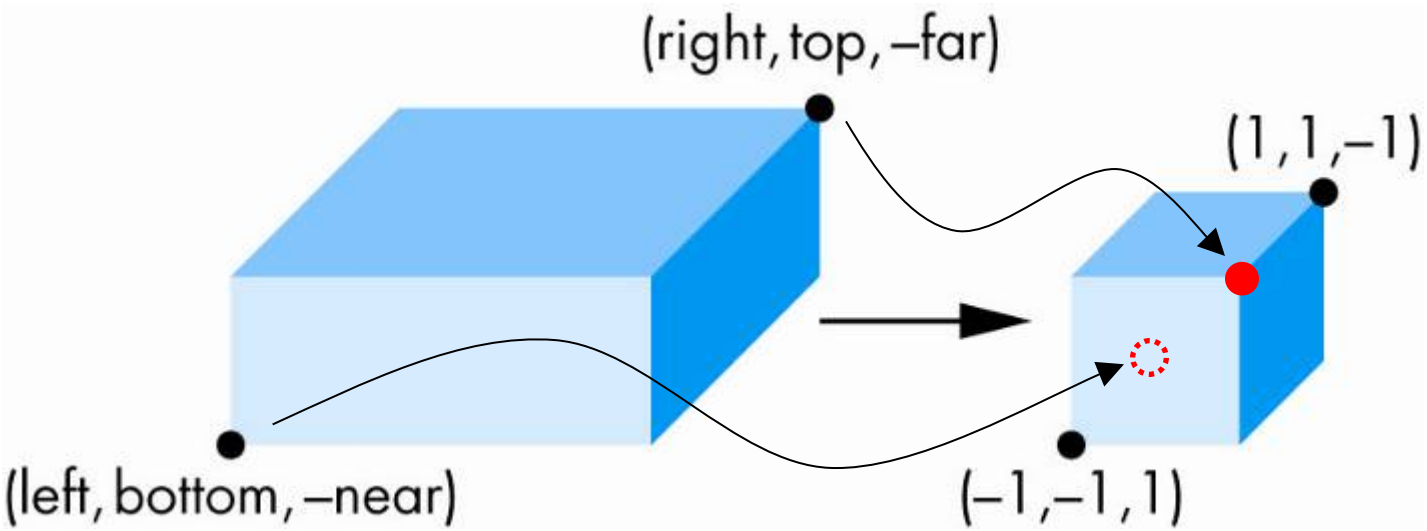
- Can be specified by defining a view volume (in the camera frame) using

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



# OpenGL Orthographic Projection

- The **`glOrtho()`** function then generates a matrix that linearly maps the view volume to the canonical view volume, where
  - (left, bottom, –near) is mapped to (–1, –1, –1)
  - (right, top, – far) is mapped to (1, 1, 1)



# Orthographic Projection Matrix

- The mapping can be found by
  - First, translating the view volume to the origin
  - Then, scaling the view volume to the size of the canonical view volume

$$\mathbf{M}_{\text{ortho}} = \mathbf{S}\left(\frac{2}{right-left}, \frac{2}{top-bottom}, \frac{2}{near-far}\right) \cdot \mathbf{T}\left(\frac{-(right+left)}{2}, \frac{-(top+bottom)}{2}, \frac{(far+near)}{2}\right)$$

  - Note that $z = -near$ is mapped to $z = -1$, and $z = -far$ to $z = +1$

# Orthographic Projection Matrix

$$\mathbf{M}_{\text{ortho}} = \begin{bmatrix} \frac{2}{right-left} & 0 & 0 & \frac{-(right+left)}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & \frac{-(top+bottom)}{top-bottom} \\ 0 & 0 & \frac{-2}{far-near} & \frac{-(far+near)}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

# Viewport Transformation

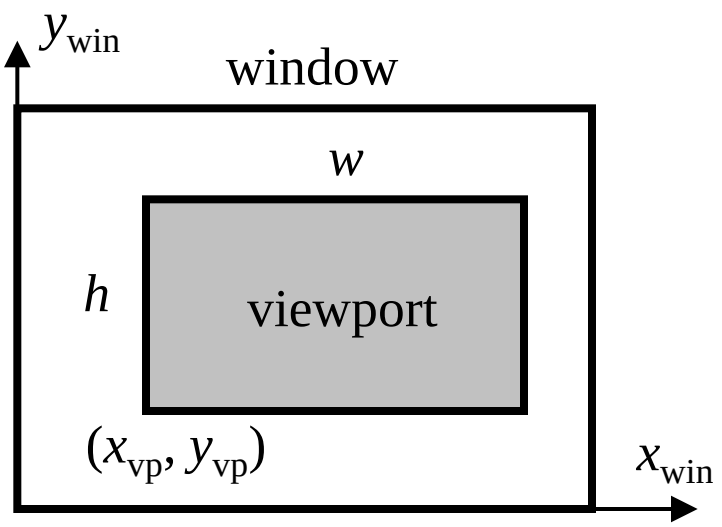
- The canonical view volume is then mapped to the viewport (from NDC to window coordinates)

$$\frac{x_{\text{NDC}} - (-1)}{2} = \frac{x_{\text{win}} - x_{\text{vp}}}{w} \quad \Rightarrow \quad x_{\text{win}} = x_{\text{vp}} + \frac{w(x_{\text{NDC}} + 1)}{2}$$

$$\frac{y_{\text{NDC}} - (-1)}{2} = \frac{y_{\text{win}} - y_{\text{vp}}}{h} \quad \Rightarrow \quad y_{\text{win}} = y_{\text{vp}} + \frac{h(y_{\text{NDC}} + 1)}{2}$$

$$z_{\text{win}} = \frac{z_{\text{NDC}} + 1}{2}$$

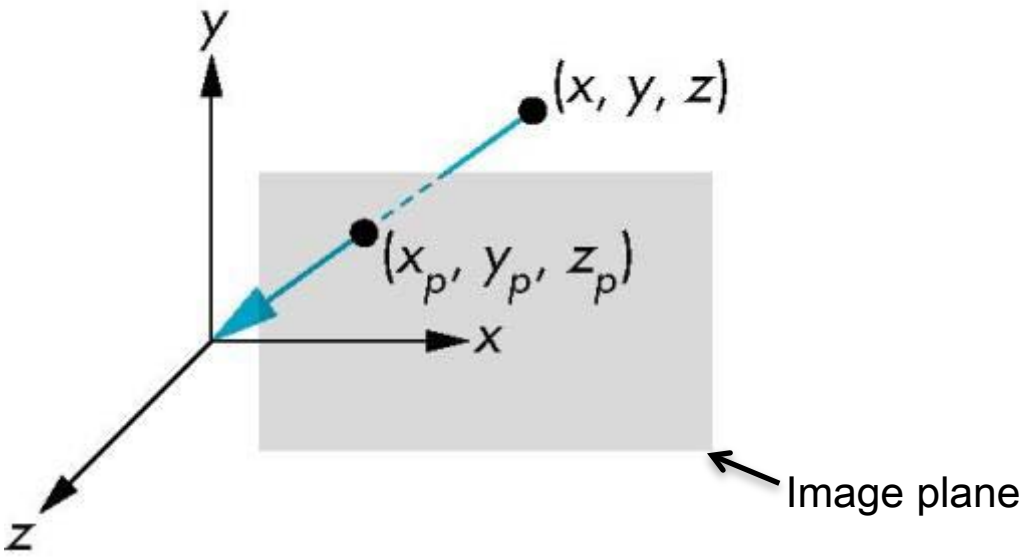
- By default, $z_{\text{win}}$ is between 0 and 1
- The $z_{\text{win}}$ is needed for z-buffer hidden surface removal



# Perspective Projection

# Simple Perspective Projection

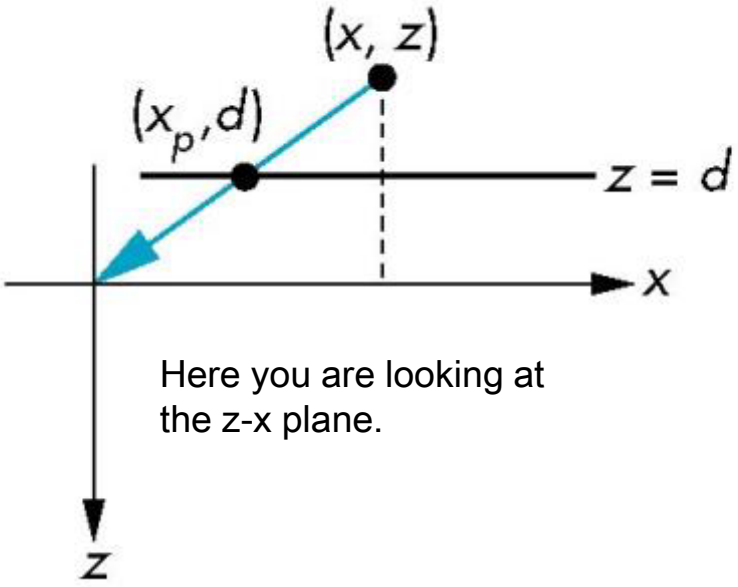
- Center of projection at the origin
- Projection plane is $z = d$, $d < 0$




All points will be projected by rays drawn from the 3D point to the origin (0,0,0). Now, we just need to place the “image” plane somewhere along the Z axis.

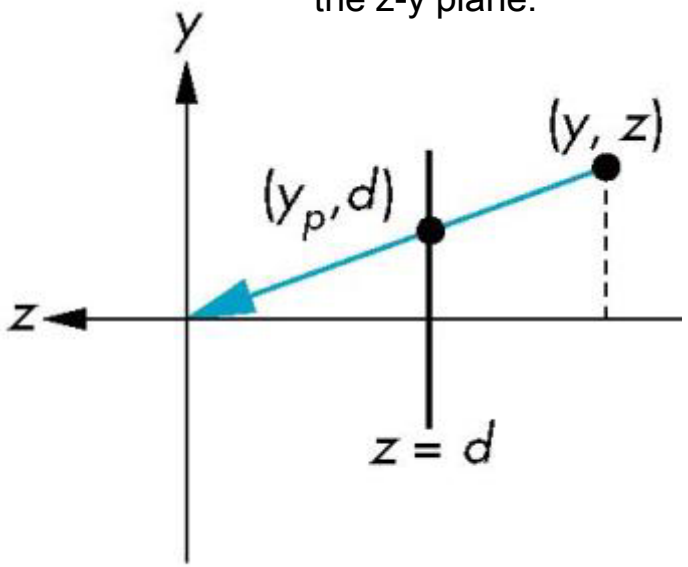
# Perspective Equations

- Consider top and side views




Here you are looking at the z-x plane.

Here you are looking at the z-y plane.




$$x_p = \frac{x}{z/d}$$

$$y_p = \frac{y}{z/d}$$

$$z_p = d$$

Assume, we put the image plane at "z=d".

All 3D points final $z_p$ value will be $d$. What changes is the $x_p$, $y_p$ for the different points.

# Perspective Equations

- On careful observation

This is just a matter of similar triangles from basic geometry.

Triangle $0 \rightarrow x_p \rightarrow d$
is similar to $0 \rightarrow X \rightarrow Z$

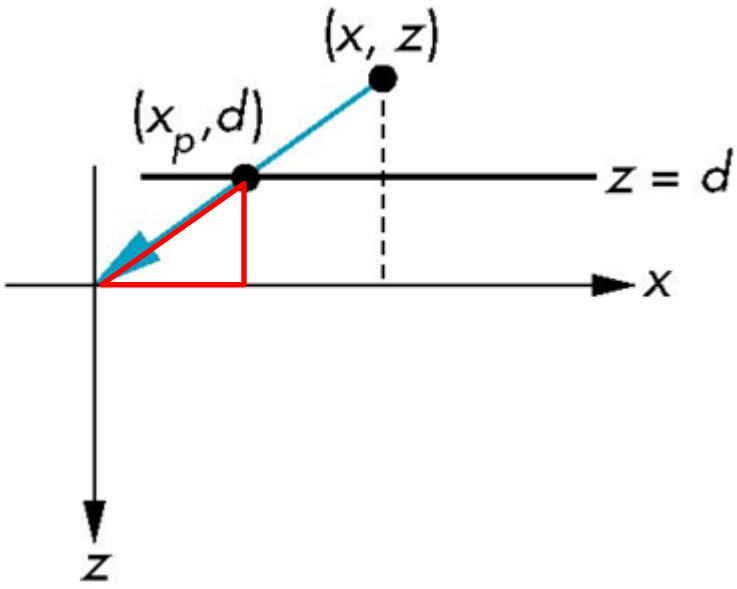
Triangle $0 \rightarrow y_p \rightarrow d$
is similar to $0 \rightarrow Y \rightarrow Z$

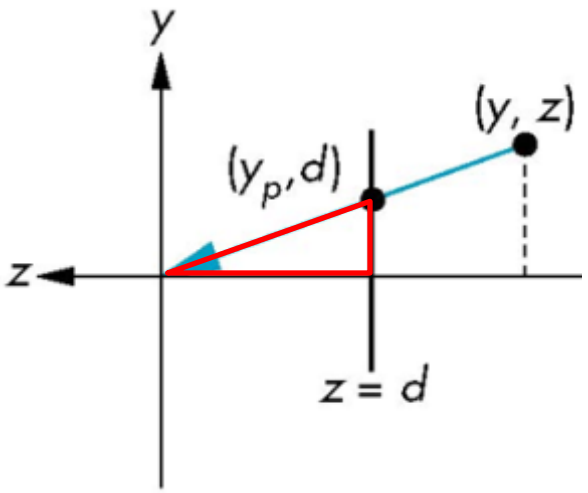
Therefore the ratio of the lengths must be the same:

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

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

From these ratios we can compute the $x_p$ and $y_p$ from the equations on the previous slides.



# Using Matrix Multiplication

- Consider $\mathbf{p} = \mathbf{Mq}$ where

$$\mathbf{p} = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix} \qquad \mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \qquad \mathbf{q} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

# Perspective Division

- However $w \neq 1$, so we must divide by $w$ to return from homogeneous coordinates
- This *perspective division* yields

$$\mathbf{p} = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix} \xrightarrow{\text{perspective division}} \mathbf{p}' = \begin{bmatrix} \frac{x}{z/d} \\ \frac{y}{z/d} \\ d \\ 1 \end{bmatrix}$$

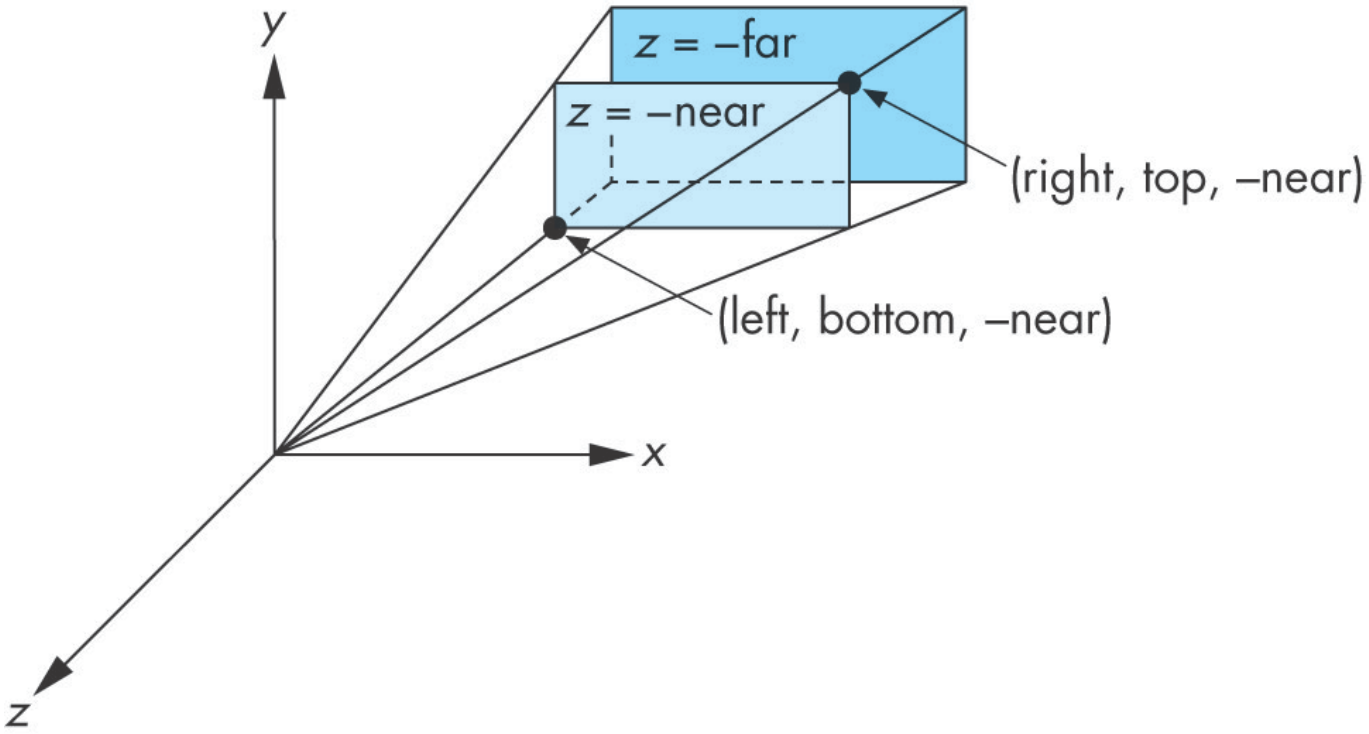
  the desired perspective equations

- This is one reason why 3D graphics API uses homogeneous coordinates

# OpenGL Perspective Projection

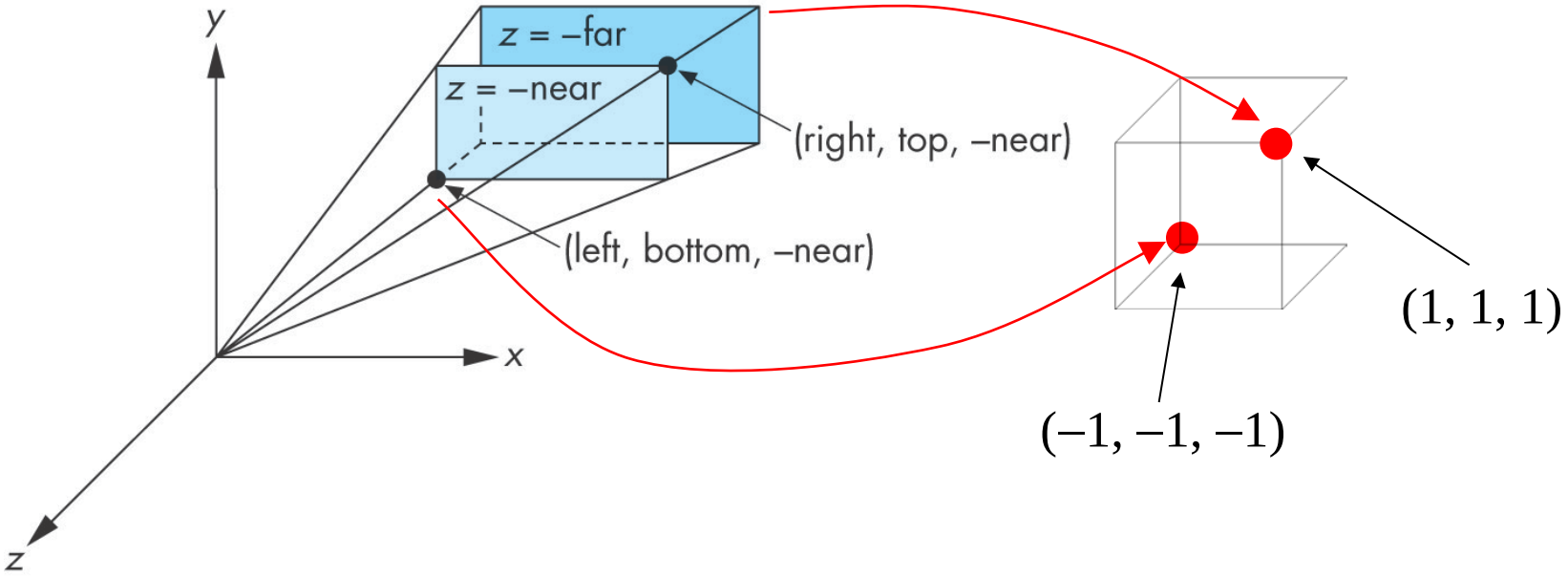
- Can be specified by defining a view volume (*view frustum*) in the camera frame using

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



# OpenGL Perspective Projection

- The **`glFrustum()`** function then generates a matrix that maps the view frustum to the canonical view volume, where



# Perspective Projection Matrix

$$\mathbf{M}_{\text{persp}} = \begin{bmatrix} \frac{2 \cdot near}{right - left} & 0 & \frac{right + left}{right - left} & 0 \\ 0 & \frac{2}{top - bottom} & \frac{top + bottom}{top - bottom} & 0 \\ 0 & 0 & \frac{-(far + near)}{far - near} & \frac{-2 \cdot far \cdot near}{far - near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

# Orthographics vs Perspective Projection

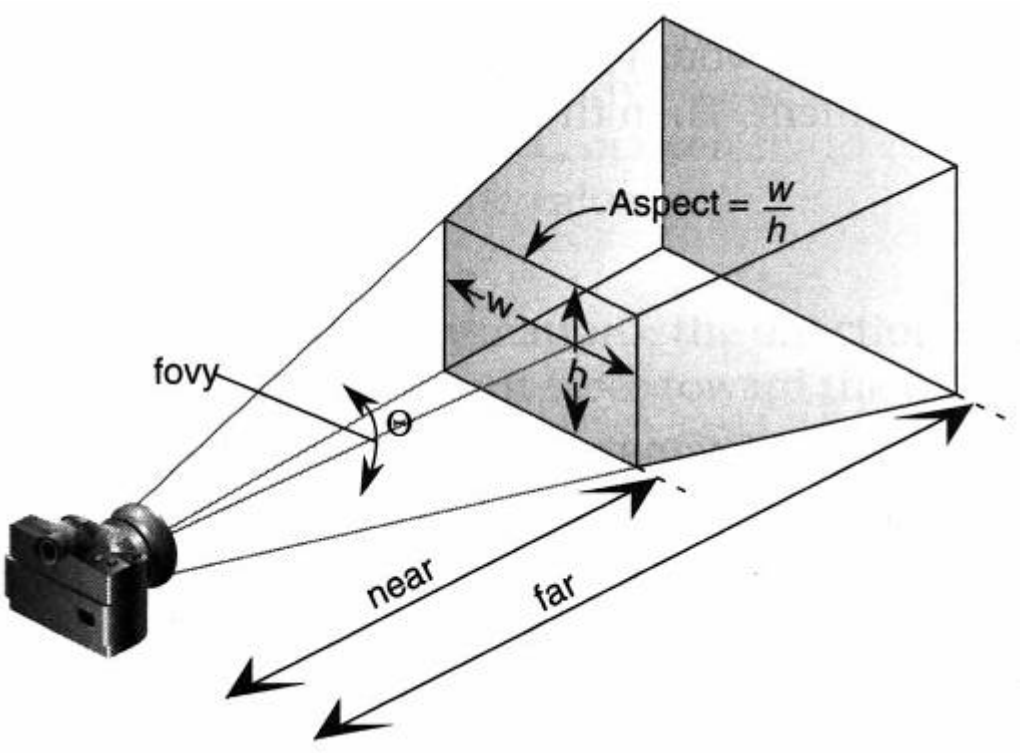
$$\mathbf{M}_{\text{ortho}} = \begin{bmatrix} \frac{2}{right-left} & 0 & 0 & \frac{-(right+left)}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & \frac{-(top+bottom)}{top-bottom} \\ 0 & 0 & \frac{-2}{far-near} & \frac{-(far+near)}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{M}_{\text{persp}} = \begin{bmatrix} \frac{2\cdot near}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2}{top-bottom} & \frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & \frac{-(far+near)}{far-near} & \frac{-2\cdot far\cdot near}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

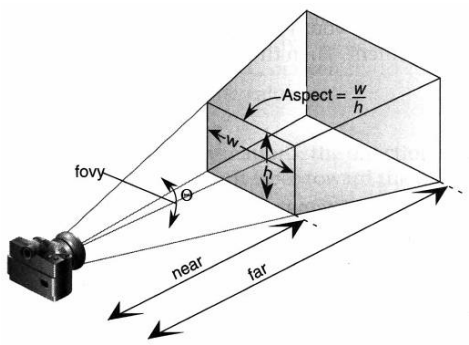
# Perspective Projection Using Field of View

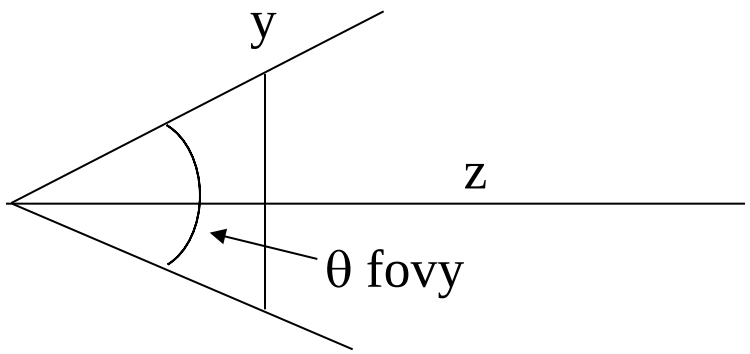
- The **`glFrustum()`** function allows non-symmetric view
- More often, we want to specify a symmetric view volume
- We can use

```
gluPerspective( fovy, aspect, near, far );
```



# glPerspective()




```
glPerspective(
fovy,    // angle [0-180]
aspect,  // w/h
near,    // same as before
far );
```

In this case, you don't explicitly set the left, bottom, top, right, but you have the same effect. One good thing with `glPerspective` is the left/right/top/bottom are always symmetric.

# CAUTION for glFrustum/glPerspective!

- Make sure you never put “0” for the near clipping plane
- It can create very strange and unpredictable results
- `Near` should always be a number greater than 0, preferably 1.

`Near=1`


`gluPerspective( 40, 1.0, 1, 300);`

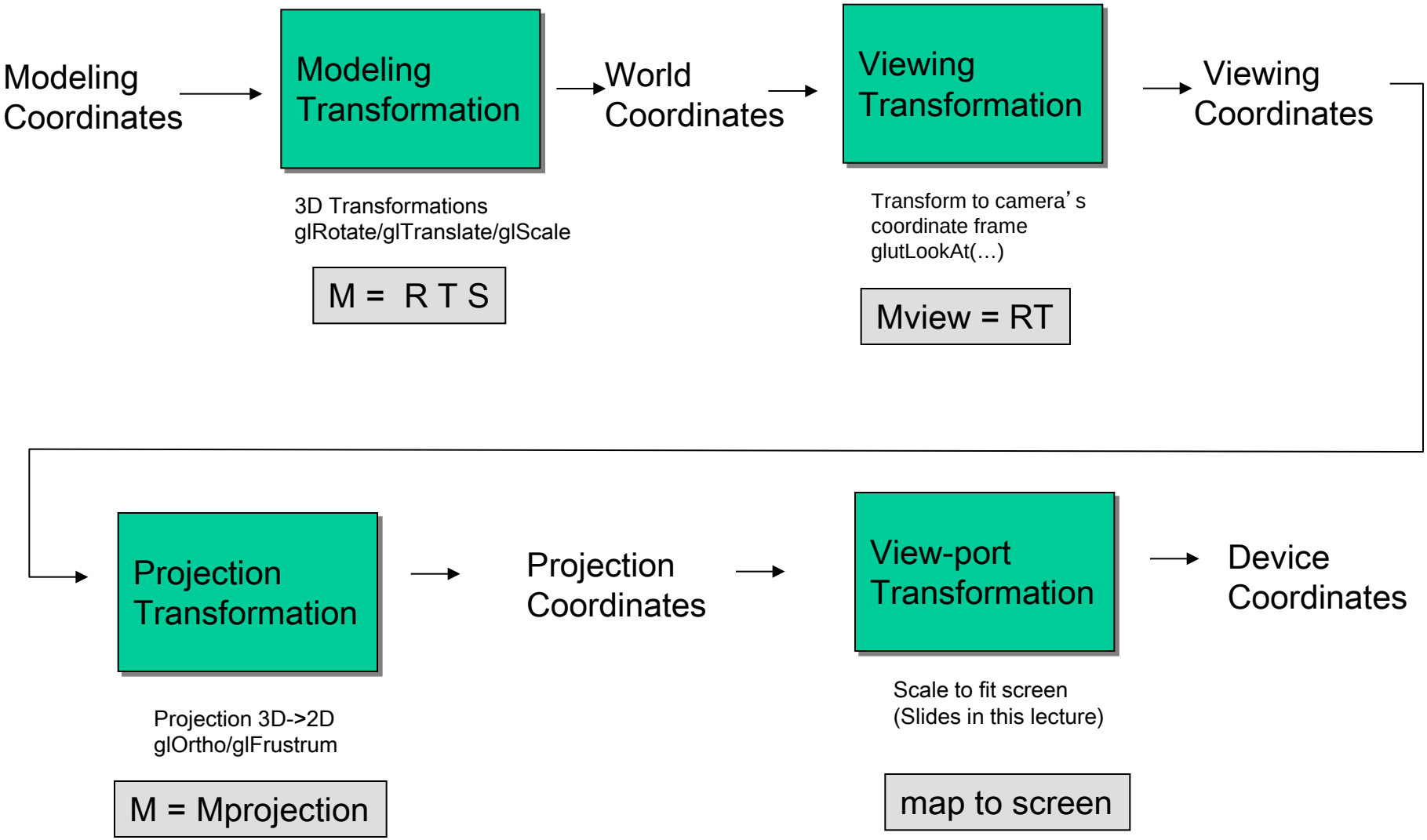
A+ student, getting a high-paid job and good looking spouse.

`Near=0`


`gluPerspective( 40, 1.0, 0, 300);`

Unleashes an evil spirit in your OpenGL app, you are going to get a D-.

# The OpenGL Transformation Stages



# The OpenGL Transformation Stages

- Using `glMatrixMode(..)` and `glPush/PopMatrix`

- `GL_PROJECTION, GL_MODELVIEW`

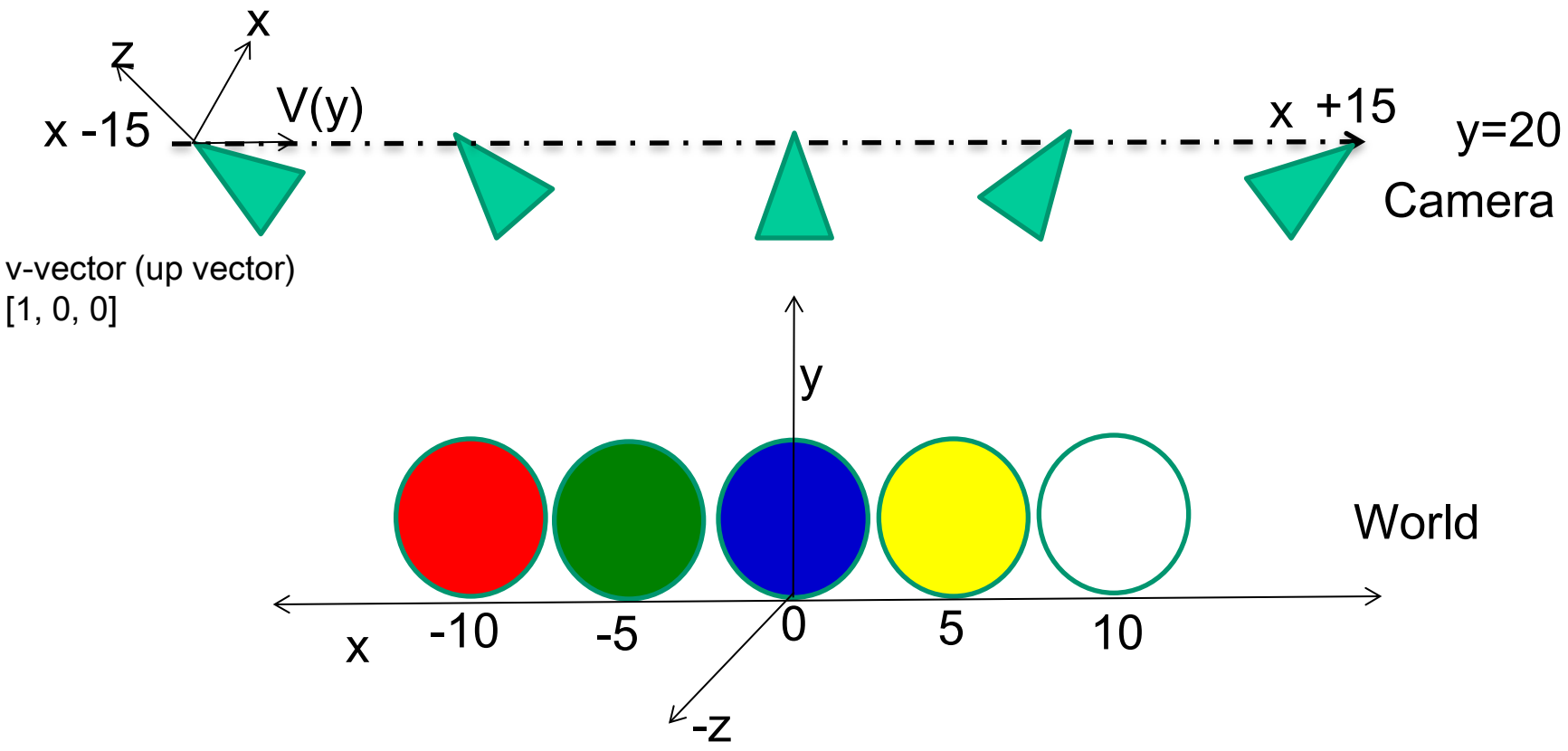

Recall from last lecture,
openGL maintains
two matrices that make
up the "Current Transform Matrix"

`gtMatrixMode(GL_PROJECTION);`
Only modifies "P" – so, calling `glLoadIdentity()` won't affect matrix M.

`glMatrixMode(GL_MODELVIEW);`
Only modifies "M" – calls here won't affect matrix P.

Important: We typically associate the "View Transform" with the GL_MODELVIEW, and not with the projection matrix.

# Putting it together in OpenGL



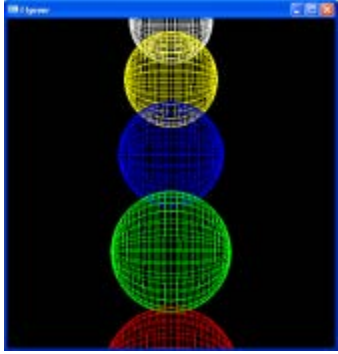

Let's create a scene where we have a camera "fly over" 5 wireframe spheres. The camera will always be looking at the origin. Our "up" vector will be in the [1, 0, 0] direction. Think about what that means in terms of the image.

# See: flyover1.c

```
#include <stdlib.h>
#include <gl/glut.h>

/*                      R,    G,   B        */
float colors[5][3] = { {1.0,  0.0, 0.0 },    // R
                       {0.0,  1.0, 0.0 },    // G
                       {0.0,  0.0, 1.0 },    // B
                       {1.0,  1.0, 0.0 },    // Y
                       {1.0,  1.0, 1.0 } };  // White

void init()
{
  // turn on normalize and depth test
  glEnable(GL_DEPTH_TEST);
}

void idle()
{
  glutPostRedisplay();
}
```

Nothing special, just setting up variables for color and standard init()/idle() funcs.

# See: flyover1.c



```
void display()
{
    static float x_offset=-15;
    int i;

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    /* Setup the view of the world */
    glMatrixMode(GL_PROJECTION);           // Switch to projection mode (P)
    glLoadIdentity();                      // P <- I
    gluPerspective( 40.0, 1.0, 1.0, 50.0); // P <- P

    glMatrixMode(GL_MODELVIEW);           // Switch to model view mode  (M)
    glLoadIdentity();                     // M = I

    gluLookAt(x_offset, 20, 0,  // EYE  // M = RT(view)
     0, 0, 0,                    // LOOK AT POINT
     1, 0, 0 );                 // UP VECTOR (direction Y)

    for(i=0; i < 5; i++) {
      glColor3f(colors[i][0], colors[i][1], colors[i][2 ] );
     glPushMatrix();                       // Save M = RT(view)
     glTranslatef((i-2)*(5), 2.5, 0);      // M = RT(view)T
     glScalef(2.5, 2.5, 2.5);              // M = RT(view)TS
     glutWireSphere(1.0, 30, 30);          // draw geometry
     glPopMatrix();                        // M = RT(view)
    }


    x_offset+=0.01;
    if (x_offset > 15)
        x_offset = -15;

    glutSwapBuffers();
}
```

Setup projection matrix. First switch to “projection” mode.

Now switch to “model” mode. Note eye position, look at, and “up” vector.

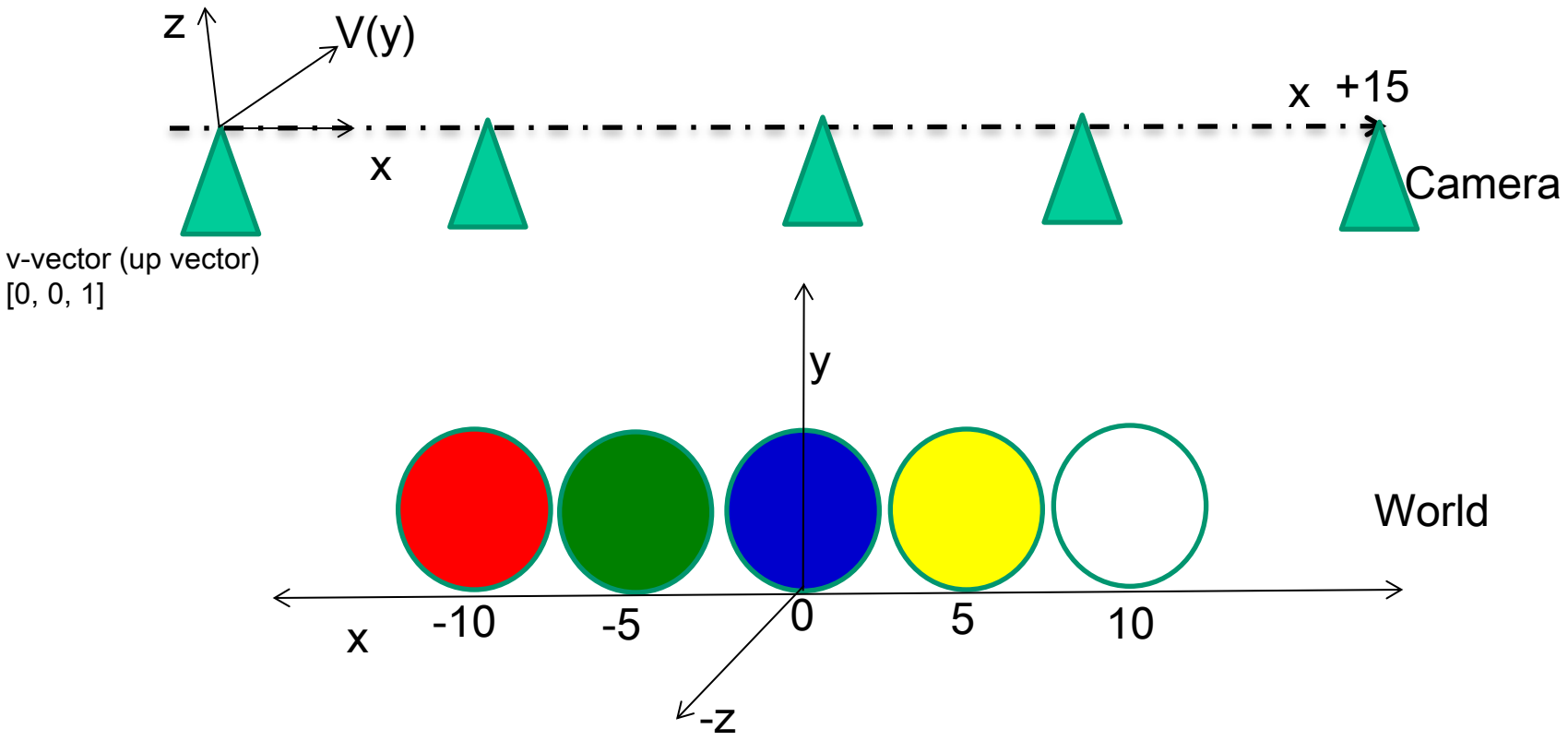
(eyeX, eyeY, eyeZ)

Lookat = (0,0,0)

Push and Pop to preserve “look at”. Modify M for each sphere.

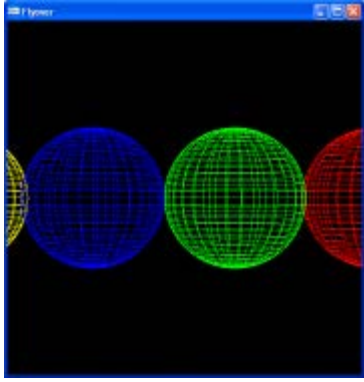
Update variable.

# Changing Look At Point




Lets modify the original scene to have the camera look “down” and have the up vector be [0,0,1]. How does this change the image?

# flyover2.c



```
void display()
{
    static float x_offset=-15;
    int i;

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    /* Setup the view of the world */
    glMatrixMode(GL_PROJECTION);           // Switch to projection mode (P)
    glLoadIdentity();                      // P <- I
    gluPerspective( 40.0, 1.0, 1.0, 50.0); // P <- P

    glMatrixMode(GL_MODELVIEW);           // Switch to model view mode  (M)
    glLoadIdentity();                     // M = I

    gluLookAt(x_offset, 20, 0,    // M = RT(view)
     x_offset, 19, 0,
     0, 0, 1 );

    for(i=0; i < 5; i++) {
      glColor3f(colors[i][0], colors[i][1], colors[i][2 ] );
     glPushMatrix();                       // Save M = RT(view)
     glTranslatef((i-2)*(5), 2.5, 0);      // M = RT(view)T
     glScalef(2.5, 2.5, 2.5);              // M = RT(view)TS
     glutWireSphere(1.0, 30, 30);          // draw geometry
     glPopMatrix();                        // M = RT(view)
    }
    glFlush();

    x_offset+=0.01;
    if (x_offset > 15)
        x_offset = -15;

    glutSwapBuffers();
}
```

Change look at point and "up" vector.

(eyeX, eyeY, eyeZ)

Lookat = (eyeX, eyeY-1, eyeZ)

[Consider this example if you want to set up a camera and have it look at a direction "in front" of the viewer. What is the direction here?]

# flyover3.c

```
void reshape()
{
/* Setup the view of the world */
    glMatrixMode(GL_PROJECTION);           // Switch to projection mode (P)
    glLoadIdentity();                      // P <- I
    gluPerspective( 40.0, 1.0, 1.0, 50.0); // P <- P
}
void display()
{   static float x_offset=-15;
    int i;

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glMatrixMode(GL_MODELVIEW);           // Switch to model view mode  (M)
    glLoadIdentity();                     // M = I

    gluLookAt(x_offset, 20, 0,    // M = RT(view)
     x_offset, 19, 0,
     0, 0, 1 );

    for(i=0; i < 5; i++) {
      glColor3f(colors[i][0], colors[i][1], colors[i][2 ] );
     glPushMatrix();                      // Save M = RT(view)
     glTranslatef((i-2)*(5), 2.5, 0);     // M = RT(view)T
     glScalef(2.5, 2.5, 2.5);             // M = RT(view)TS
     glutWireSphere(1.0, 30, 30);         // draw geometry
     glPopMatrix();                       // M = RT(view)
    }
    glFlush();

    x_offset+=0.01;
    if (x_offset > 15)
        x_offset = -15;

    glutSwapBuffers();
}
```

Place the projection matrix setup in “reshape”.

No need to call gluPerspective in “display”, just make sure to switch into Modelview mode.

# Summary

- Two important part of 3D viewing

1) Setting up camera location

  - Use `gluLookAt()`
  - Need to specify “eye point”, “look at point”, and “up vector”

2) Choosing a projection

  - Orthographic or Parallel

`glOrtho()` `gluPerspective()/glFrustum()`

  - Make sure to set clipping planes correctly!

**End of Lecture 5**
**next – models and illumination**