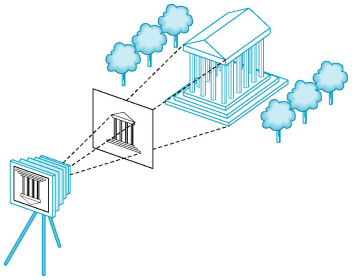
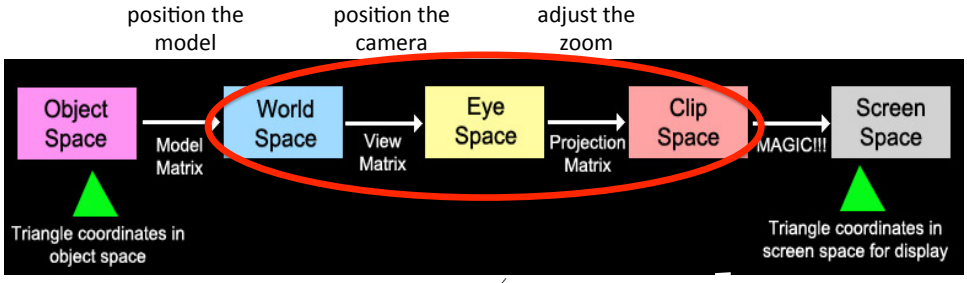
Computer Graphics
3D Viewing and Projection



Review Vertex Transformations

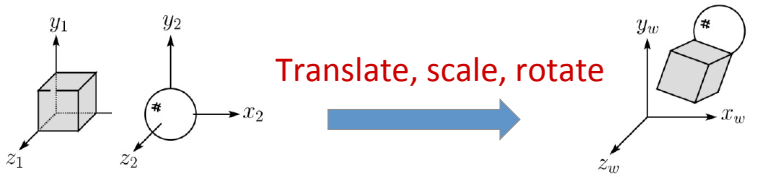


Vertex shader:

```
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;
in vec3 vert;
```

```
void main()
{
    gl_Position = projection*view*model*vec4(vert, 1.0);
}
```

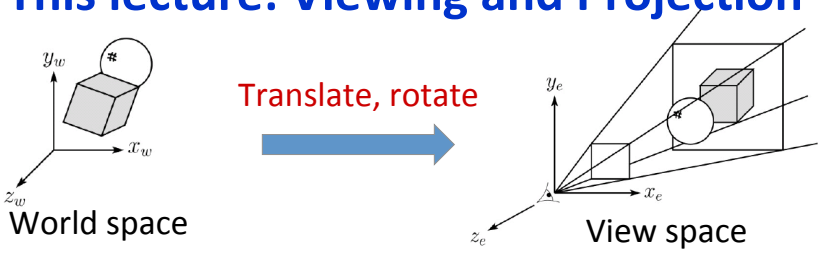
Review 3D Transformations



$$\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_x \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$T(t_x,t_y,t_z)$ $S(s_x,s_y,s_z)$ $R(\theta,0,0,1)$

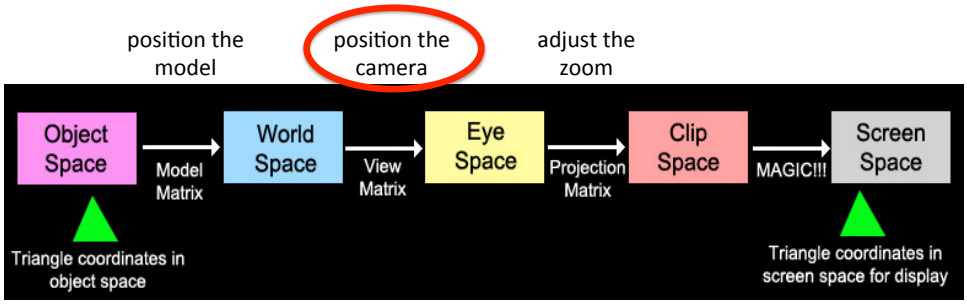
This lecture: Viewing and Projection



- Taking pictures with a real camera:

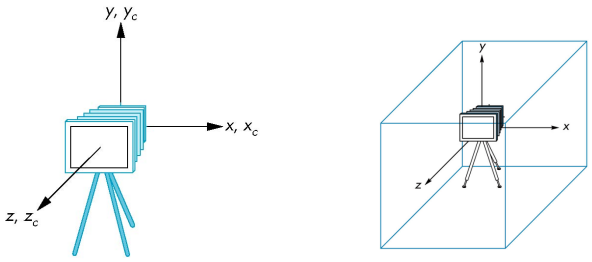
  - Identify interesting objects
  - Rotate and translate the camera to desired viewpoint
  - Adjust camera settings such as focal length
  - Choose desired resolution and aspect ratio, etc.
  - Take a snapshot
- Graphics does the same thing for image rendering

Positioning the Camera



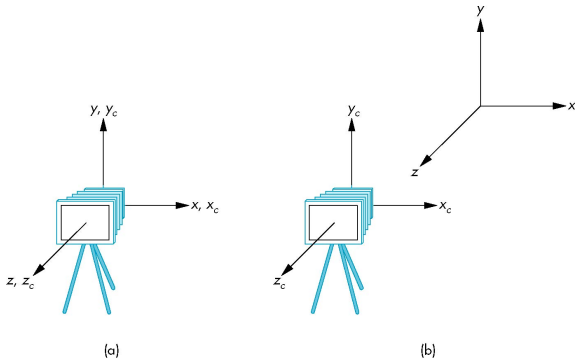
The Default OpenGL Camera

- In OpenGL, initially the world and camera origin are the same
- The camera is located at origin and points in the -z direction



- OpenGL also specifies a default view volume that is a cube with sides of length 2 centered at the origin

Positioning the Camera

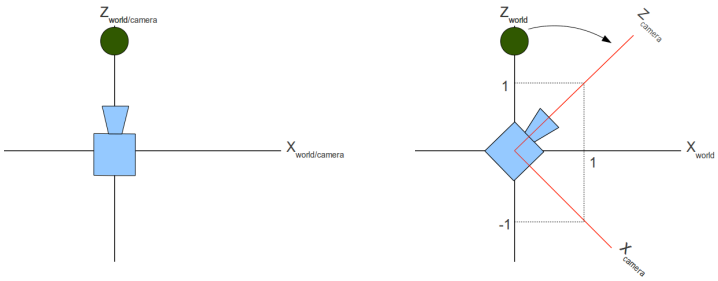


- Suppose that we wish to position the camera at (0, 0, 2) with respect to the world
- In real life we move the camera. In OpenGL we move the world:

  glm::translate (mat4(), vec3(0, 0, -2));

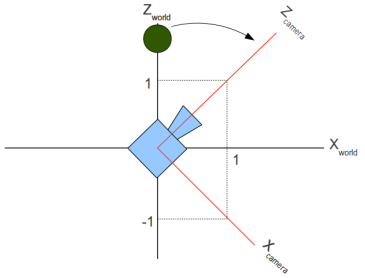
Positioning the Camera

- Suppose that we wish to rotate the camera 45 degrees clockwise
  - This is the same as the green ball rotating 45 degrees counterclockwise

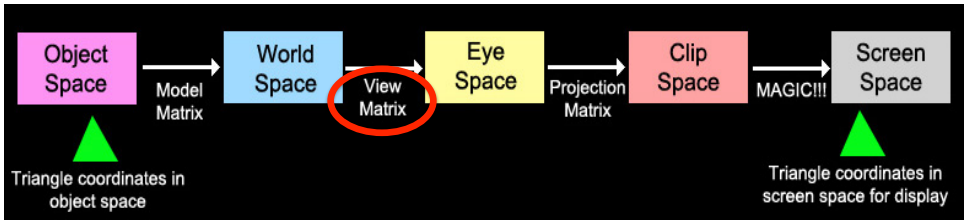


- The green ball is located at (0,y,z) in world space, but it has a negative X and a positive Z in camera space
- We need to find the location of the ball in camera space. So we add new transformations to the pipeline. Then we can simply forget all about the world space and use only the camera space.

View Matrix M_{view}



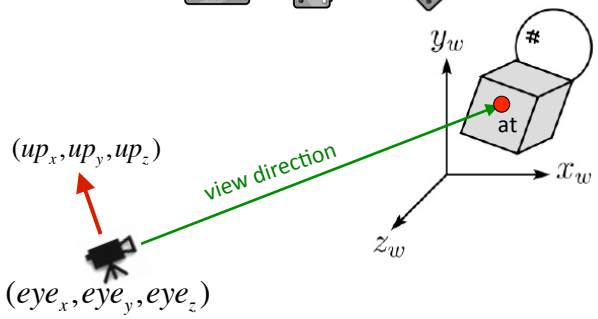
$$M_{view}\begin{pmatrix} x_{world} \\ y_{world} \\ z_{world} \\ 1 \end{pmatrix} = \begin{pmatrix} x_{camera} \\ y_{camera} \\ z_{camera} \\ 1 \end{pmatrix}$$



- The view matrix would take the camera from its position in the world back to origin, facing in the negative z-direction.

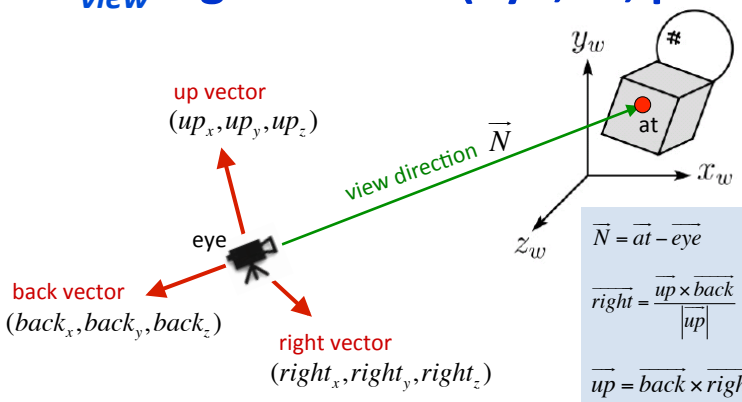
M_{view} = glm::lookAt(eye, at, position)

- Camera *location* (eye): position in the world coordinates
- Viewing *direction* (at): the direction in which we aim the camera
- Camera *orientation* (*up*):



- glm::lookat creates the view matrix that transforms world coordinates to camera coordinates

M_{view} = glm::lookAt(eye, at, position)



$$\vec{N} = \vec{at} - \vec{eye} \qquad \overrightarrow{back} = -\frac{\vec{N}}{|\vec{N}|}$$

$$\overrightarrow{right} = \frac{\vec{up} \times \overrightarrow{back}}{|\vec{up}|}$$

$$\vec{up} = \overrightarrow{back} \times \overrightarrow{right}$$

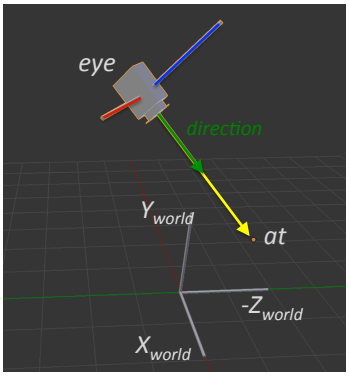
- Given eye, at and up:
  - Construct the camera system
  - Create the view matrix

$$M_{view} = \begin{pmatrix} right_x & right_y & right_z & -dot(right, eye) \\ up_x & up_y & up_z & -dot(up, eye) \\ back_x & back_y & back_z & -dot(back, eye) \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Hands-on Session

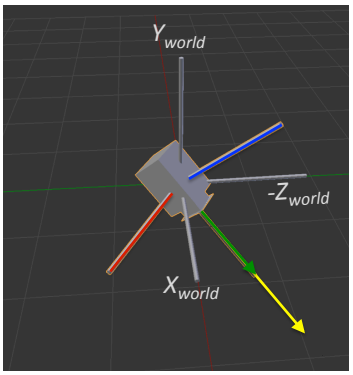
- The OpenGL tutor programs
  - Download the OpenGL tutors from
    https://user.xmission.com/~nate/tutors.html
  - Run the *projection* tutor
  - Play with the parameters of `lookAt`
    1. Modify the eye position
    2. Modify the `lookAt` point
    3. Modify the up vector (turn the image upside-down)

Alternate Way to Compute the View Matrix M_{view}



- Camera position in world space

```
vec3 direction = normalize(at-eye);
```

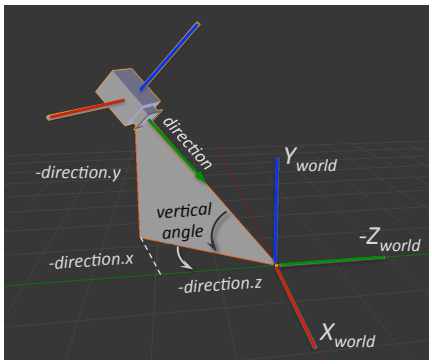


- First we translate the camera to origin.

```
mat4 trans = translate(mat4(), -eye);
```

Alternate Way to Compute the View Matrix M_{view}

- Imagine the camera at the origin, pointing in the indicated *direction*. It is shown shifted here (along the *-direction* vector) for clarity only.



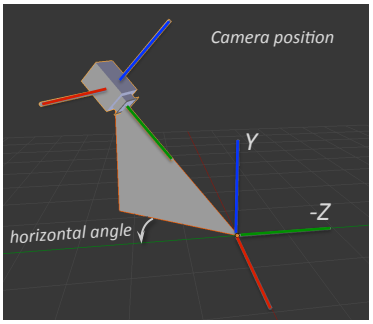

```
vec3 direction = normalize(at-eye);
_verticalAngle = radians(asinf(-direction.y));
_horizontalAngle = -glm::radians(atan2f(-direction.x, -direction.z));
```

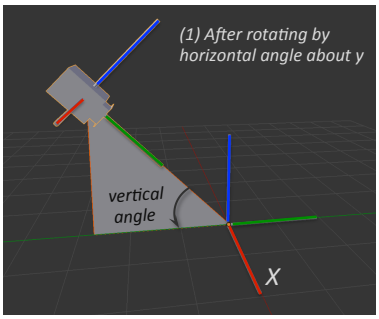
```
mat4 orientation;
orientation = glm::rotate(orientation, glm::radians(_verticalAngle), glm::vec3(1,0,0));
orientation = glm::rotate(orientation, glm::radians(_horizontalAngle), glm::vec3(0,1,0));
```

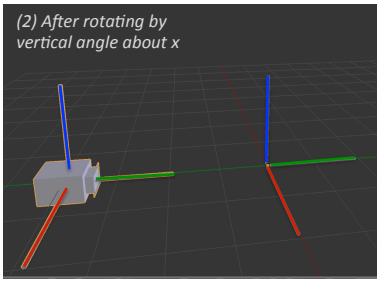
```
mat4 view = orientation * translate(mat4(), -eye);
```

Visualizing the Rotations

- Imagine the camera at the origin, pointing in the indicated direction. It is shown shifted here (along the direction vector) for clarity only.





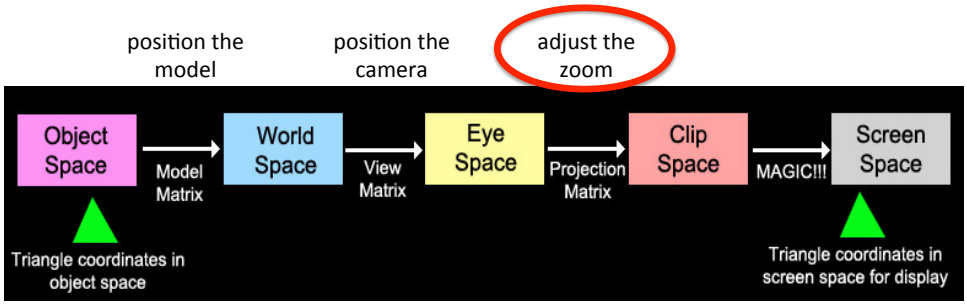


- Because the camera starts at the origin prior to these rotations, the two coordinate systems end up overlapping after these rotations.

Hands-On Activity 1

- Open the opengl_series `04_camera` project
- Run the code and interact with the scenes using the mouse scroll, left mouse button, WASD keys, and XZ keys for up and down
  - Interaction is handled by the `Update` and `OnScroll` functions
  - Bind the scene movement to the up/down/left/right keys instead of XZAD
- Study the `Camera` implementation in the togl namespace
- Note the way the view matrix is computed
  - One translation followed by two rotations
  - What happens if you reverse the order: rotate first, then translate? Try it out and see how the mouse interaction changes (then revert to original code)
- Add code to the `Update` function to make the camera rotate around the world. You can accomplish this by setting its position to some time-dependent value (*radius*cos(time), 0, radius*sin(time)*), so that it moves in a circle. This will create a spinning animation for the cube.
- Do *not delete* existing code (just comment it out).

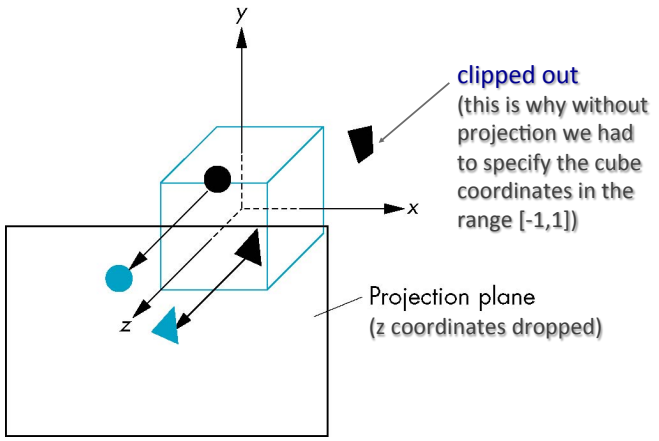
Projection



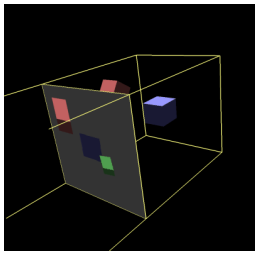
- We have determined how objects are placed relative to camera
- But how are the objects projected to the image?
  (Converting from 3D to 2D)

The Default OpenGL Projection

- Default projection is orthogonal (orthographic)
- Default viewing volume is a 2x2x2 cube centered at the origin
- Default projection matrix is identity

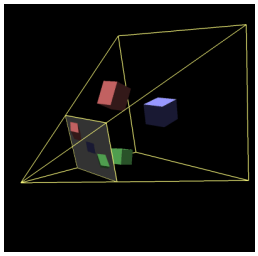


Orthographic Projection



$$O = \begin{pmatrix} \frac{2}{right-left} & 0 & 0 & \frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & \frac{top+bottom}{top-bottom} \\ 0 & 0 & \frac{-2}{far-near} & \frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

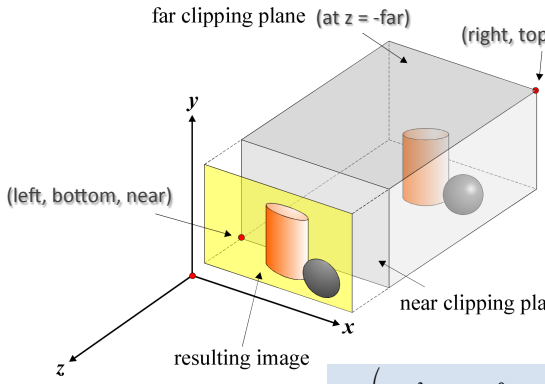
Perspective Projection



$$P = \begin{pmatrix} \frac{1}{\tan\left(\frac{fov}{2}\right)*aspect} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan\left(\frac{fov}{2}\right)} & 0 & 0 \\ 0 & 0 & \frac{-(far+near)}{far-near} & \frac{-2*far*near}{far-near} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

O = glm::ortho(left, right, bottom, top, near, far);
P = glm::perspective(radians(_fieldOfView), _viewportAspectRatio, _nearPlane, _farPlane);

glm::ortho(left, right, bottom, top, near, far)



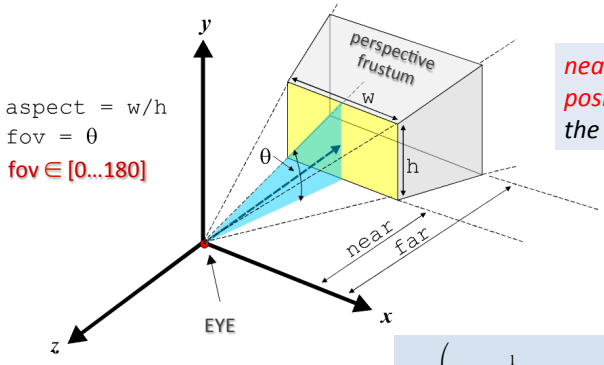
- We always look in -*z* direction
- *near* and *far* are specified as *positive* distances *relative to the camera.* Example: glm::ortho(-1.0,1.0,-1.0,1.0,1.0,3.0)

$$O = \begin{pmatrix} \frac{2}{right-left} & 0 & 0 & \frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & \frac{top+bottom}{top-bottom} \\ 0 & 0 & \frac{-2}{far-near} & \frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Hands-on Session

- Run the *projection* tutor again
  - Select orthographic projection (right click on lower window)
  - Play with the parameters of ortho
    1. Move the near plane closer to the object. What happens if it intersects the object?
    2. Move the left plane closer to the right plane. What effect does this have on the image?
    3. What do you expect to happen if you double the viewing volume? Try it out
  - Play with the parameters of lookAt

glm::perspective(fov, aspect, near, far)



aspect = w/h
fov = θ
fov ∈ [0...180]

near and *far* are specified as *positive* distances *relative to the camera*

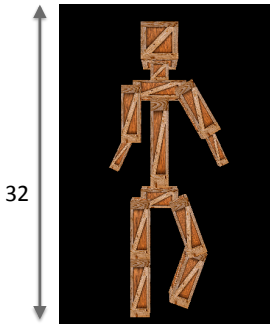
$$\frac{h/2}{near} = \tan\frac{\theta}{2} \Rightarrow h = 2\,near\tan\frac{\theta}{2}$$

$$P = \begin{pmatrix} \frac{1}{\tan\left(\frac{fov}{2}\right)*aspect} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan\left(\frac{fov}{2}\right)} & 0 & 0 \\ 0 & 0 & \frac{-(far+near)}{far-near} & \frac{-2*far*near}{far-near} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

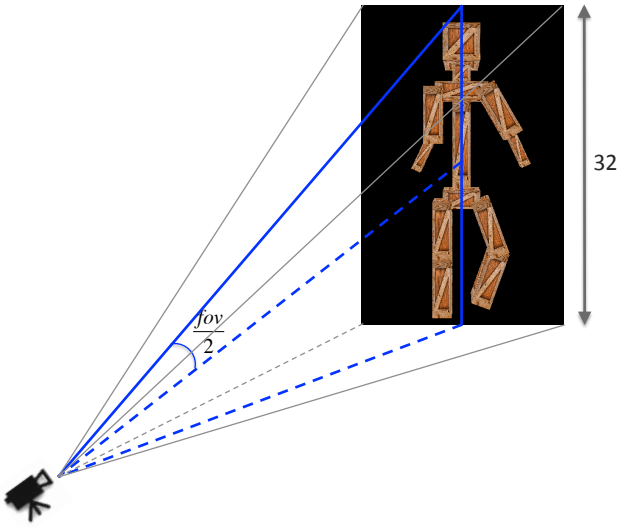
Understanding Previous Activity

- Previously we modified 03_matrices to accommodate the robot:
  - In the `LoadShaders` function, we changed the arguments of the perspective call by `perspective(radians(~~50.0f~~ 70.0f), SCREEN_SIZE.x/SCREEN_SIZE.y, 0.1f, ~~10.0f~~ 40.0f)`
  - We also changed the first argument of `glLookAt` to `glm::vec3(0,0,~~3~~ 30)`, so that the camera was positioned along the z-axis, 30 units away from the world, and looks into the center of the world

- Why did we need these changes?

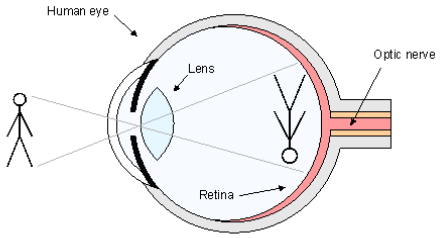


Understanding Previous Activity

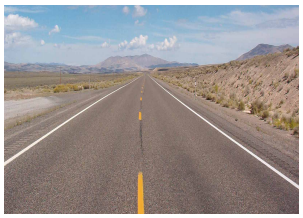


Why Perspective Projection?

- Models the way human eyes see

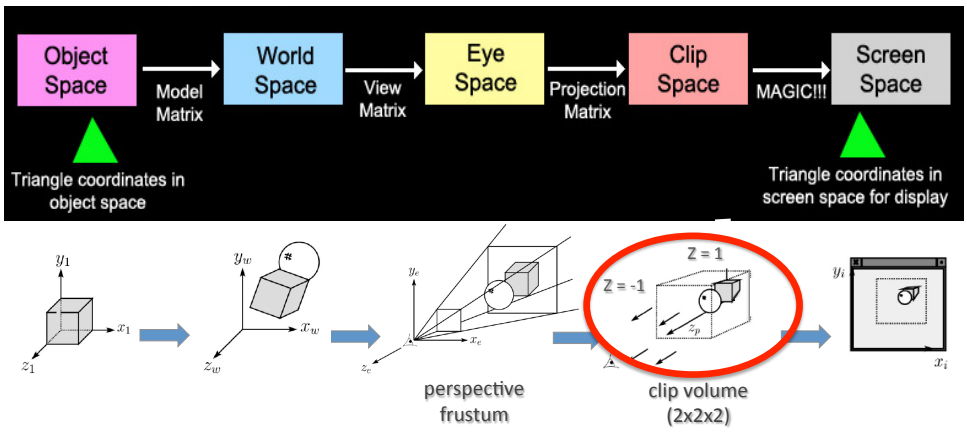


- Objects further away from the camera appear smaller

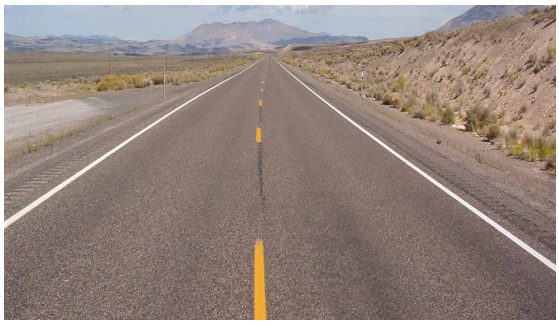


Perspective Division

- Going from the perspective frustum to the 2x2x2 clip volume



Perspective Projection

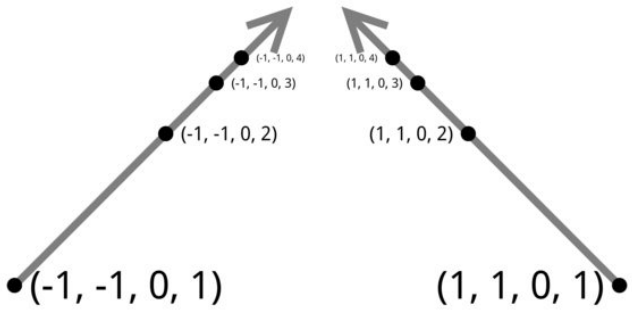


- In Euclidean space (geometry) two parallel lines cannot intersect
- In projective space parallel lines meet at infinity
  - Introduce homogeneous coordinates to resolve this issue

Homogeneous Coordinates

- Add a 4th dimension w
- Cartesian point (X,Y,Z) becomes (x,y,z,w) in homogeneous space

$$X = \frac{x}{w} \qquad Y = \frac{y}{w} \qquad Z = \frac{z}{w}$$



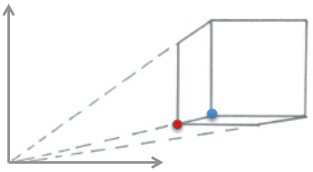
Perspective Division Example

- Projection transformation:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} 4 \\ 1 \\ -4 \\ 1 \end{pmatrix} = \begin{pmatrix} 4 \\ 1 \\ 4 \\ 4 \end{pmatrix} \qquad \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} 4 \\ 1 \\ -5 \\ 1 \end{pmatrix} = \begin{pmatrix} 4 \\ 1 \\ 5 \\ 5 \end{pmatrix}$$

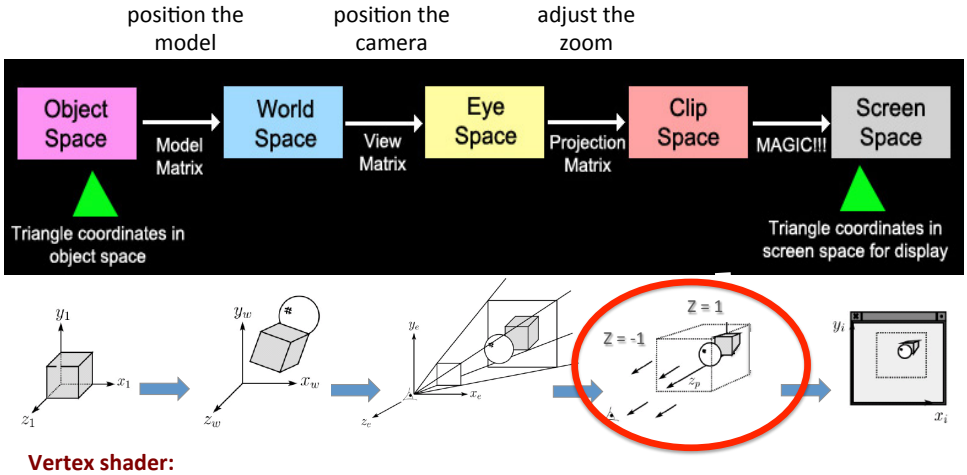
- Notice how the W value has changed
- Next step is perspective division:

$$\frac{1}{4}\begin{pmatrix} 4 \\ 1 \\ 4 \\ 4 \end{pmatrix} = \begin{pmatrix} 1 \\ 0.25 \\ 1 \\ 1 \end{pmatrix} \qquad \frac{1}{5}\begin{pmatrix} 4 \\ 1 \\ 5 \\ 5 \end{pmatrix} = \begin{pmatrix} 0.8 \\ 0.2 \\ 1 \\ 1 \end{pmatrix}$$



- After perspective division, the W value is discarded, and we are left with coordinates that have been correctly scaled according to a 3D perspective projection

After Perspective Division



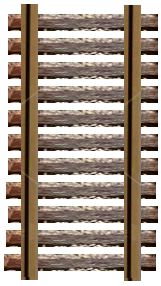
Vertex shader:

```
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;
in vec3 vert;
```

```
void main()
{
    gl_Position = projection*view*model*vec4(vert, 1.0);
}
```

Orthographic vs. Perspective

• Object appears same size, no matter how far from the camera	• Farther objects appear smaller
• Parallel lines in the world scene are parallel lines in the image	• Parallel lines in the world scene are not generally parallel lines in the image



Hands-on Session

- Run again the *projection* tutor
  - Play with perspective parameters
    - What happens if you increase the field of view?
    - What happens if you change the aspect ratio?
    - How does the image change with changes in near and far?
  - Play with camera orientation (lookAt)
  - Switch between perspective and ortho

Hands-On Activity 2

- Experiment with the field-of-view and aspect-ratio parameters of the `Camera` in `AppMain`. See if you can figure out how these affect the perspective frustum
- Check how the zoom feature bound to the mouse scroll in `Update` uses the field of view. What would be an alternative way to implement zooming?
- Add the same interactive features to your robot program from `05_asset_instance` as in `04_camera`
  - Have the mouse control robot rotation about the world origin
  - Have the camera move around the robot