So far you only draw 2D objects
Also control the range of your objects to \([-1,1]\) in x and y and use 0.0 for Z, i.e., keep them flat
This \([-1,1] \times [-1,1] \times [-1,1]\) is called the canonical view volume, and only objects within this range at the end of the transformation pipeline will be visible
OpenGL 3D viewing

- But now let’s make your program more flexible
  - Help your objects escape the 2D flat land
  - Place a camera in the 3D world pointing to arbitrary locations within arbitrary ranges
  - Allow perspective or orthographic projection
- To allow the second and third bullets above, we need to add two more transformation stages to ensure that at the end of the transformation pipeline, visible objects are still in the range of $[-1,1]^3$
Escape the flatland

- If the objects in your program already have the z coordinates (although set to 0), escape the flatland is easy. Just use non-zero z coordinates.
- Because we have not added camera and projection transformations, for now the z range needs to be kept in [-1,1] in order to stay visible.
- To ensure correct visibility order, you need to enable depth test.
  - `gl.enable(gl.DEPTH_TEST);`
- Next slide tells you how to do it.
Set Up Camera Viewing and Projection

- To more flexibly place your objects in the 3D world, you need to place a 3D camera (conceptually) and set up its lens
  - Camera position and direction
  - Perspective or orthographic projection
- This corresponds to two more transformation stages:
  - Viewing transformation
  - Projection transformation
Transformation Pipeline

Local (Object) Space → Modeling transformation → World Space

Clip Space ← Projection transformation ← Eye Space

Perspective divide → NDC space → Viewport mapping → Screen space

Normalized Device Coordinates
Set up OpenGL Camera

- Set up a 3D scene is like taking a photograph
Viewing

- Position and orient your camera
Viewing

- Important camera parameters to specify
  - Camera (eye) position \((Ex, Ey, Ez)\) in world coordinate system
  - Center of interest \((coi)\) \((cx, cy, cz)\)
  - Orientation (which way is up?) View-up vector \((Up_x, Up_y, Up_z)\)
Viewing

- Camera position and orientation forms a camera (eye) coordinate frame

- Transform objects from world to eye space
Eye Space

- Right hand coordinate system

Transform to eye space can simplify many downstream operations (such as projection) in the pipeline
World to Eye Transformation

- Head tilt: Rotate your head by $\delta$
- Just rotate the object about the eye space z axis - $\delta$
- $M_{w2e} =$
  \[
  \begin{bmatrix}
  \cos(-\delta) & -\sin(-\delta) & 0 & 0 & ux & uy & ux & 0 & 1 & 0 & 0 & -ex \\
  \sin(-\delta) & \cos(-\delta) & 0 & 0 & vx & vy & vz & 0 & 0 & 1 & 0 & -ey \\
  0 & 0 & 1 & 0 & nx & ny & nz & 0 & 0 & 0 & 1 & -ez \\
  0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
  \end{bmatrix}
  \]

I will explain the math behind for this matrix later.
Use glMatrix to Construct the viewing Matrix

-Create a view matrix

var vMatrix = mat4.create();
vMatrix = mat4.lookAt([0.0, 0.0, 5.0], [0.0, 0.0, 0.0], [0.0, 1.0, 0.0], vMatrix);

-(1) vec3: position; (2) vec3: center of interest, (3) vec3: viewUp vector

-The returned matrix, vMatrix is the same as the matrix shown in the previous slide
Combine viewing and modeling matrices

- The viewing matrix is multiplied with the modeling matrix, and the result is called modelview matrix

\[
vMatrix = \text{mat4.lookAt}([0.0, 0.0, 5.0], [0.0, 0.0, 0.0], [0.0, 1.0, 0.0]);
\]

\[
\text{mat4.identify}(\text{mMatrix});
\]
\[
\text{mMatrix} = \text{mat4.rotate}(\text{mMatrix}, \text{angle}, [0.0f, 0.0f, 1.0f]);
\]
\[
\text{mMatrix} = \text{mat4.scale}(\text{mMatrix}, [1,2,1]);
\]

\[
\text{mat4.multiply}(\text{vMatrix}, \text{mMatrix}, \text{mvMatrix});
\]

\[
\text{modelView Matrix} = \text{view Matrix} \times \text{model Matrix}
\]
Use the modelview matrix

When multiplying the modelview matrix with the vertices, you transform the geometry from local space to eye space.
Transformation Pipeline

- Local (Object) Space → Modeling transformation → World Space
- Projection transformation → Eye Space → Viewing transformation
- Clip Space
- Perspective divide
- NDC space → Viewport mapping → Screen space

- Normalized Device Coordinates
Projection

- Control the “lens” of the camera
- Project the object from 3D world to 2D screen
Projection Transformation

- Control how to project the object from 3D to 2D
  - Perspective or Orthographic
  - Field of view and image aspect ratio
  - Near and far clipping planes
Orthographic Projection

- No foreshortening effect – distance from camera does not matter
- The projection center is at infinite

- Projection calculation – just drop z coordinates
Perspective Projection

- Characterized by object foreshortening
  - Objects appear to be larger if they are closer to the camera
  - This is what happens in the real world

- Need:
  - Projection center
  - Field of View
  - Image, near, far planes

- Projection: Connecting the object to the projection center
Determine how much of the world is taken into the picture.

The larger is the field view, the smaller is the object projection size.
Near and Far Clipping Planes

- Only objects between near and far planes are drawn

- Near plane + far plane + field of view = Viewing Frustum
Viewing Frustum

- 3D counterpart of 2D world clip window

- Objects outside the frustum are clipped
Projection Matrix

- Orthographic Projection

\[
\begin{pmatrix}
\frac{2}{(xmax-xmin)} & 0 & 0 & -\frac{1}{2}\frac{xmax+xmin}{xmax-xmin} \\
0 & \frac{2}{(ymax-ymin)} & 0 & -\frac{1}{2}\frac{ymax+ymin}{ymax-ymin} \\
0 & 0 & -\frac{2}{(zmax-zmin)} & -\frac{1}{2}\frac{zmax+zmin}{zmax-zmin} \\
0 & 0 & 0 & 1
\end{pmatrix}
\]
### Projection Matrix

**Perspective Projection**

\[
\begin{align*}
x' &= \frac{2N}{(x_{\text{max}}-x_{\text{min}})} \quad 0 \quad \frac{(x_{\text{max}}+x_{\text{min}})}{(x_{\text{max}}-x_{\text{min}})} \quad 0 \quad x \\
y' &= 0 \quad \frac{2N}{(y_{\text{max}}-y_{\text{min}})} \quad \frac{(y_{\text{max}}+y_{\text{min}})}{(y_{\text{max}}-y_{\text{min}})} \quad 0 \quad y \\
z' &= 0 \quad 0 \quad \frac{-F + N}{F-N} \quad \frac{-2F*N}{(F-N)} \quad z \\
w' &= 0 \quad 0 \quad -1 \quad 0 \quad 1
\end{align*}
\]
- Set up the perspective projection matrix

var pMatrix = mat4.create();
mat4.perspective(60.0, 1.0, .1, 100.0, pMatrix);

- (1): field of view
- (2): aspect ratio
- (3): near plane distance
- (4): far plane distance
Aspect ratio is used to calculate the window width

\[ \text{Aspect} = \frac{w}{h} \]
Use projection matrix

- The projection matrix is then combined with the model view matrix and becomes modelviewprojection (MVP) matrix
- \( \text{MVP} = \text{projection} \times \text{viewing} \times \text{modeling} \)
- Multiply MVP to your vertices will transform them from local space to the canonical view volume, i.e., in the range of \([-1,1]^3\) if they are visible
- Remember to use 4D vectors \([x,y,z,1]\) to multiply
Transformation Pipeline

- Local (Object) Space
- Modeling transformation
- World Space
- Viewing transformation
- Projection transformation
- Eye Space
- NDC space
- Viewport mapping
- Screen space
- Perspective divide

Normalized Device Coordinates
3D viewing under the hood

Modeling Transformation  |  Viewing Transformation  |  Projection Transformation

Viewport Transformation

Display
3D viewing under the hood

Topics of Interest:

- Viewing transformation
- Projection transformation
Viewing Transformation

- Transform the object from world to eye space
  - Construct an eye space coordinate frame
  - Construct a matrix to perform the coordinate transformation
Eye Coordinate Frame

- Known: eye position, center of interest, view-up vector
- To find out: new origin and three basis vectors

Assumption: the direction of view is orthogonal to the view plane (the plane that objects will be projected onto)
Eye Coordinate Frame (2)

- **Origin**: eye position (that was easy)
- Three basis vectors: one is the normal vector \( n \) of the viewing plane, the other two are the ones \( u \) and \( v \) that span the viewing plane

\[ n = \frac{N}{|N|} \]

Center of interest (COI)

\( (u,v,n) \) should be orthogonal to each other

\( n \) is pointing away from the world because we use right hand coordinate system

Remember \( u,v,n \) should be all unit vectors
Eye Coordinate Frame (3)

How about \( u \) and \( v \)?

We can get \( u \) first -

\( u \) is a vector that is perpendicular to the plane spanned by \( N \) and view up vector (\( V_{up} \))

\[
U = V_{up} \times n
\]

\[
u = U / |U|
\]
Eye Coordinate Frame (4)

- How about $v$? Knowing $n$ and $u$, getting $v$ is easy.

$$v = n \times u$$

$v$ is already normalized.
Put it all together

Eye space origin: \((\text{Eye.x}, \text{Eye.y}, \text{Eye.z})\)

Basis vectors:

\[
\begin{align*}
\mathbf{n} & = \frac{\mathbf{eye} - \mathbf{COI}}{|\mathbf{eye} - \mathbf{COI}|} \\
\mathbf{u} & = \frac{\mathbf{V}_{\text{up}} \times \mathbf{n}}{|\mathbf{V}_{\text{up}} \times \mathbf{n}|} \\
\mathbf{v} & = \mathbf{n} \times \mathbf{u}
\end{align*}
\]
World to Eye Transformation

- Transformation matrix \( (M_{w2e}) \)?
  \[ P' = M_{w2e} \times P \]

1. Come up with the transformation sequence to move eye coordinate frame to the world
2. And then apply this sequence to the point \( P \) in a reverse order
World to Eye Transformation

- Rotate the eye frame so that it will be “aligned” with the world frame
- Translate (-ex, -ey, -ez)

Rotation:
\[
\begin{bmatrix}
ux & uy & uz & 0 \\
vx & vy & vz & 0 \\
nx & ny & nz & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

How to verify the rotation matrix?

Translation:
\[
\begin{bmatrix}
1 & 0 & 0 & -ex \\
0 & 1 & 0 & -ey \\
0 & 0 & 1 & -ez \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
World to Eye Transformation (2)

- Transformation order: apply the transformation to the object in a reverse order - translation first, and then rotate

\[
M_{w2e} = \begin{bmatrix}
ux & uy & ux & 0 \\
vx & vy & vz & 0 \\
xn & ny & nz & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} \begin{bmatrix}
1 & 0 & 0 & -ex \\
0 & 1 & 0 & -ey \\
0 & 0 & 1 & -ez \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
World to Eye Transformation (3)

- Head tilt: Rotate your head by $\delta$
- Just rotate the object about the eye space z axis - $\delta$
- $M_{w2e} = \begin{bmatrix}
\cos(-\delta) & -\sin(-\delta) & 0 & 0 & ux & uy & ux & 0 & 1 & 0 & 0 & -ex \\
\sin(-\delta) & \cos(-\delta) & 0 & 0 & vx & vy & vz & 0 & 0 & 1 & 0 & -ey \\
0 & 0 & 1 & 0 & nx & ny & nz & 0 & 0 & 1 & 0 & -ez \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1
\end{bmatrix}$

Why $-\delta$?

When you rotate your head by $\delta$, it is like rotate the object by $-\delta$
Projection Transformation

- Projection – map the object from 3D space to 2D screen

Perspective: `gluPerspective()`

Parallel: `glOrtho()`
Parallel Projection

- After transforming the object to the eye space, parallel projection is relatively easy – we could just drop the Z
  
  \[ X_p = x \]
  \[ Y_p = y \]
  \[ Z_p = -d \]

- We actually want to keep \( Z \) – why?
OpenGL maps (projects) everything in the visible volume into a **canonical view volume**

$$\text{glOrtho}(\text{xmin, xmax, ymin, ymax, near, far})$$

Canonical View Volume

$$(-1, -1, -1) \rightarrow (1, 1, 1)$$
Parallel Projection (3)

Transformation sequence:

1. Translation (M1): \((-\text{near} = \text{zmax}, -\text{far} = \text{zmin})\)
   
   \(-\frac{\text{xmax}+\text{xmin}}{2}, -\frac{\text{ymax}+\text{ymin}}{2}, -\frac{\text{zmax}+\text{zmin}}{2}\)

2. Scaling (M2):

   \(\frac{2}{\text{xmax}-\text{xmin}}, \frac{2}{\text{ymax}-\text{ymin}}, -\frac{2}{\text{zmax}-\text{zmin}}\)

\[
\begin{bmatrix}
\frac{2}{\text{xmax}-\text{xmin}} & 0 & 0 & -\frac{\text{xmax}+\text{xmin}}{\text{xmax}-\text{xmin}} \\
0 & \frac{2}{\text{ymax}-\text{ymin}} & 0 & -\frac{\text{ymax}+\text{ymin}}{\text{ymax}-\text{ymin}} \\
0 & 0 & -\frac{2}{\text{zmax}-\text{zmin}} & -\frac{\text{zmax}+\text{zmin}}{\text{zmax}-\text{zmin}} \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
Perspective Projection

Side view:

- Projection plane
- Eye (projection center)
- Based on similar triangle:
  \[
  \frac{y}{y'} = \frac{-z}{d}
  \]
  \[
  Y' = y \times \frac{d}{-z}
  \]
Perspective Projection (2)

- Same for $x$. So we have:
  
  \[ x' = \frac{x \times d}{-z} \]
  \[ y' = \frac{y \times d}{-z} \]
  \[ z' = -d \]

- Put in a matrix form:

\[
\begin{pmatrix}
  x' \\
  y' \\
  z' \\
  w
\end{pmatrix} =
\begin{pmatrix}
  1 & 0 & 0 & 0 & x \\
  0 & 1 & 0 & 0 & y \\
  0 & 0 & 1 & 0 & z \\
  0 & 0 & (1/-d) & 0 & 1
\end{pmatrix}
\]

- OpenGL assume $d = \text{near}$, i.e. the image plane is at $z = -\text{near}$
We are not done yet. We want to somewhat keep the z information so that we can perform depth comparison.

Use pseudo depth – OpenGL maps the near plane to -1, and far plane to 1.

Need to modify the projection matrix: solve a and b

\[
\begin{pmatrix}
  x' \\
y' \\
z' \\
w
\end{pmatrix} =
\begin{pmatrix}
  1 & 0 & 0 & 0 & x \\
  0 & 1 & 0 & 0 & y \\
  0 & 0 & a & b & z \\
  0 & 0 & (1/-d) & 0 & 1
\end{pmatrix}
\]

How to solve a and b?
Perspective Projection (4)

- Solve a and b

\[
\begin{align*}
    \begin{bmatrix} x' \\ y' \\ z' \\ w \end{bmatrix} &= 
    \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & (1/-d) & 0 \end{bmatrix} 
    \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \\
    (0,0,-1)^T &= M \times (0,0,-\text{near})^T \\
    (0,0,1)^T &= M \times (0,0,-\text{far})^T
\end{align*}
\]

- Verify this!

- \( a = (\text{far} + \text{near})/(d*(\text{far}-\text{near})) \)
- \( b = (2 \times \text{far} \times \text{near}) / (d*(\text{far}-\text{near})) \)
Not done yet. OpenGL also normalizes the x and y ranges of the viewing frustum to [-1, 1] (translate and scale).

And takes care the case that eye is not at the center of the view volume (shear).
Perspective Projection (6)

Final Projection Matrix:

\[
\begin{bmatrix}
x' & 2N/(xmax-xmin) & 0 & (xmax+xmin)/(xmax-xmin) & 0 & x \\
y' & 0 & 2N/(ymax-ymin) & (ymax+ymin)/(ymax-ymin) & 0 & y \\
z' & 0 & 0 & -(F + N)/(F-N) & -2F*N/(F-N) & z \\
w' & 0 & 0 & -1 & 0 & 1 \\
\end{bmatrix}
\]

\text{glFrustum}(xmin, xmax, ymin, ymax, N, F) \quad N = \text{near plane}, \ F = \text{far plane}
After perspective projection, the viewing frustum is also projected into a canonical view volume (like in parallel projection).