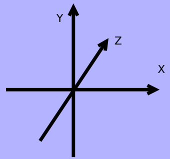
Perspective

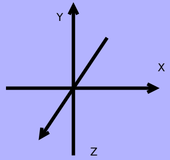
Motivation

- Isn't this class supposed to be about *3D* graphics?
- Enough of the 2D! Let's get to the 3D!

3D Space

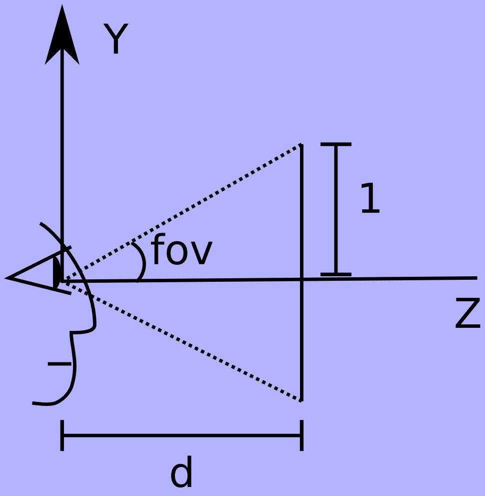
- Two choices: LHS vs. RHS
  - DirectX tends to favor LHS
  - OpenGL has traditionally used RHS
- We'll use LHS here
- Need to *project* from 3D to 2D for display
- Mathematically: Map from $\mathbb{R}^3$ to $\mathbb{R}^2$
- For ease of math, assume camera at origin looking down +Z axis





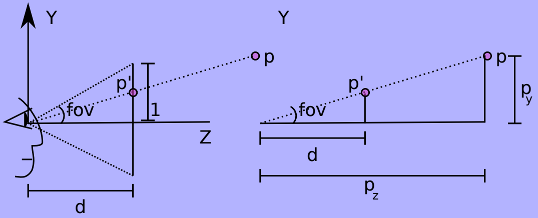
Frustum

- We have certain field of view (frustum)
- Anything outside = invisible
- Assume we put projection screen some distance from eye. Call it d.
- Assume we want height of view to be 2 (so each half has size 1)
- Let fov = half angle of view
- Given fov, how can we get d?



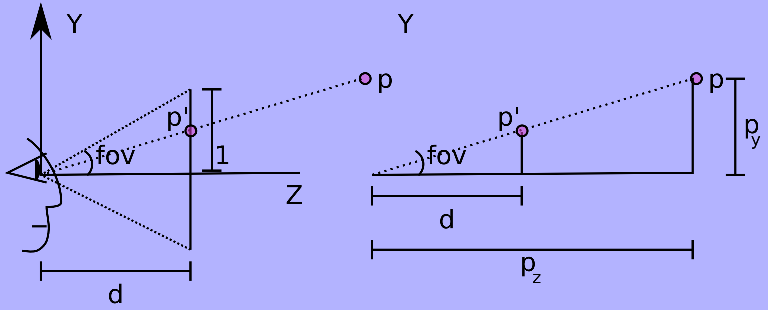
d

- Let field of view be denoted θ
- Then: $tan\,\theta = \frac{1}{d} \Rightarrow d = \frac{1}{tan\,\theta}$
- Project point p to projection screen



Projection

- Similar triangles: $d/p_z = (p'_y)/p_y$
- $p'_y = (dp_y)/p_z$
- Same idea for x: $p'_x = (dp_x)/p_z$



Non-Square

- What if screen is not square?
  - Ex: 1300x650
- Def: Aspect ratio: width÷height
  - For 1300x650, it's 2
- We have two fov's: θ_h and θ_v
- We know $\theta_h = (w/h)\theta_v$
- Compute: $d_h = 1/(tan\,\theta_h)$
- Also: $d_v = 1/(tan\,\theta_v)$
- Do projections:
  - $p'_x = \frac{d_h p_x}{p_z} \qquad p'_y = \frac{d_v p_y}{p_z}$

Matrices

- We have a point $p = (p_x, p_y, p_z)$
- We compute a projected point $p' = \left(\frac{d_h p_x}{p_z}, \frac{d_v p_y}{p_z}, ?\right)$
  - Don't know what projected z should be...ignore for now
- We'd like to express as vector-matrix multiply

Matrices

- Why matrix?
  - Can composite multiple transformations easily
  - Compact, portable notation
  - We already use matrices for world and camera transforms
  - Can do math efficiently in shader

Recall

- When we had 2D points: Needed to add an extra coordinate (w) so we could do translations with matrices
- With 3D points: Need to add a fourth coordinate (w) so we can do translations with matrices

Matrices

- 4x4 matrices $\rightarrow$ We will use 4D points
- Recall: VS outputs gl_Position $\rightarrow$ vec4
- GL automatically divides gl_Position.xyz by gl_Position.w
- This is very useful

Matrices

- We want to compute $p' = pM$ where M is some matrix we need to compute
- And we want $p' = \left(\frac{d_h p_x}{p_z}, \frac{d_v p_y}{p_z}, ?\right) = \left(\frac{d_h p_x}{p_z}, \frac{d_v p_y}{p_z}, ?, 1\right)$
- So we can also say: $p' = (d_h p_x \ , \ d_v p_y \ , \ ? \ , \ p_z)$
- Thus:

$$[d_h p_x, d_v p_y, ?, p_z] = [p_x, p_y, p_z, 1] \begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \end{bmatrix}$$

- What to put in the matrix?

Matrices

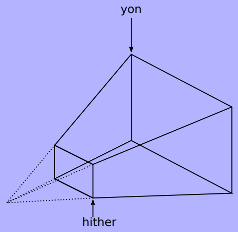
- $[d_h p_x, d_v p_y, ?, p_z] = [p_x, p_y, p_z, 1] \begin{bmatrix} d_h & 0 & ? & 0 \\ 0 & d_v & ? & 0 \\ 0 & 0 & ? & 1 \\ 0 & 0 & ? & 0 \end{bmatrix}$

Matrices

- Since $d_h = 1/(tan\,\theta_h)$ and $d_v = 1/(tan\,\theta_v)$ we can rewrite the matrix:
- $[d_h p_x, d_v p_y, ?, p_z] = [p_x, p_y, p_z, 1] \begin{bmatrix} 1/(tan\,\theta_h) & 0 & ? & 0 \\ 0 & 1/(tan\,\theta_v) & ? & 0 \\ 0 & 0 & ? & 1 \\ 0 & 0 & ? & 0 \end{bmatrix}$

Z

- We still need to deal with the Z's
- We need to maintain information about relative Z ordering so we can do depth-sorting later
- Define two distances: Hither and Yon:



Z

- Want to map z=hither $\rightarrow$ -1 and z=yon $\rightarrow$ +1
- Only part of the matrix we can change is third column
  - The rest has to be the way it is already set
- What to do?

Z

- Z mapping doesn't depend on x and y, so we fill in zeros for first two rows of third column.
- Put P and Q for other two unknowns.
- $[d_h p_x, d_v p_y, ?, p_z] = [p_x, p_y, p_z, 1] \begin{bmatrix} 1/(tan\,\theta_h) & 0 & 0 & 0 \\ 0 & 1/(tan\,\theta_v) & 0 & 0 \\ 0 & 0 & P & 1 \\ 0 & 0 & Q & 0 \end{bmatrix}$

Z

- When we multiply, we'll get: $[\ldots, \ldots, P \cdot p_z + Q, p_z]$
- After homogeneous divide: $[\ldots, \ldots, P + Q/p_z, 1]$
- So: If z == hither:
  P + Q/z = -1
- If z == yon:
  P + Q/z = 1

Solve

- What if z==hither (H)?

$$P + \frac{Q}{H} = -1$$

- Put the P on one side by itself:

$$P = -1 - \frac{Q}{H} = -\left(1 + \frac{Q}{H}\right)$$

Solve

- What if z == yon (Y)?

$$P + \frac{Q}{Y} = 1 \quad \text{or:} \quad 1 = P + \frac{Q}{Y}$$

- Substitute $-\left(1 + \frac{Q}{H}\right)$ for P:

$$1 = -\left(1 + \frac{Q}{H}\right) + \frac{Q}{Y}$$

$$1 = -1 + \frac{-Q}{H} + \frac{Q}{Y}$$

$$2 = \frac{-Q}{H} + \frac{Q}{Y} = \frac{-QY}{HY} + \frac{QH}{HY}$$

$$2 = \frac{QH-QY}{HY} = \frac{Q(H-Y)}{HY}$$

$$\frac{2HY}{H-Y} = Q$$

Solve

- Substitute back into first equation to get P:

$$P = -\left(1 + \frac{Q}{H}\right) = -\left(1 + Q \cdot \frac{1}{H}\right)$$

$$P = -\left(1 + \frac{2HY}{H-Y} \cdot \frac{1}{H}\right)$$

$$P = -\left(1 + \frac{2Y}{H-Y}\right)$$

Solve

- Now we have our final projection matrix. H and Y are hither and yon:

$$\begin{bmatrix} \frac{1}{tan\,\theta_h} & 0 & 0 & 0 \\ 0 & \frac{1}{tan\,\theta_v} & 0 & 0 \\ 0 & 0 & -\left(1+\frac{2Y}{H-Y}\right) & 1 \\ 0 & 0 & \frac{2HY}{H-Y} & 0 \end{bmatrix}$$

Shader

- Recall: Vertex shader is responsible for determining 2d positions for each input point
- So vertex shader will perform projection computation
  - We need to send projection matrix in as uniform

Spaces

- Recall: We had object space
  - Unique per-mesh, 3D, mesh usually centered around origin
- And world space
  - 3D, each mesh positioned correctly in world relative to other meshes
- And view space
  - After we translated the world so coi was at origin
  - This is going to change in a moment...
- Now we have a new space: Projection space (or clip space)
  - 2½D (altered z coordinate)

Transformations

- VS transformation order:
  - p * worldMatrix * viewMatrix * projMatrix
- Sometimes, we elect to premultiply view & projection matrices on CPU and send viewProjMatrix to GPU

Organization

- Since projection is tied to our camera's field of view, it seems reasonable to include projection matrix in camera:

```
class Camera:
    def __init__(self):
        ...set up view and proj matrices...
    def setUniforms(self):
        Program.setUniform("viewMatrix",self.viewMatrix)
        Program.setUniform("projMatrix",self.projMatrix)
```

Camera

- The camera work becomes a bit different, though!
- Recall: With 2D camera, we only had coi + head tilt
- But now we have additional degrees of freedom
  - Roll, pitch, yaw
  - Eye position
  - Center of interest

Operation

- Our basic camera idea will be the same
  - If camera eye is at
$$(e_x, e_y, e_z)$$
  - We translate entire scene by
$$-(e_x, e_y, e_z)$$
  - Then we must do a rotation operation

Camera

- Recall: In 2D, our camera was defined by two axes
  - Right and Up
- In 3D, our camera is defined by *three* axes
  - Right, Up, Look

Rotation

- We need to rotate such that:
  - The camera's 'right' vector aligns with the global X axis
  - The camera's 'up' vector aligns with the global Y axis
  - The camera's 'look' vector aligns with the global Z axis

Matrix

- We can use same computation scheme we used previously
- Write a matrix with X axis, Y axis, and Z axis in successive rows:

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- Write matrix with right, up, and look in successive rows:

$$Q = \begin{bmatrix} r_x & r_y & r_z \\ u_x & u_y & u_z \\ l_x & l_y & l_z \end{bmatrix}$$

Goal

- Now, we want to find matrix R such that:

$$I = RQ$$

- What is it?

Now...

- As before, we have an identity matrix on the left
- So R is inverse of Q
- Since Q is orthogonal, $R=Q^T$

$$R = \begin{bmatrix} r_x & u_x & l_x \\ r_y & u_y & l_y \\ r_z & u_z & l_z \end{bmatrix}$$

Result

- Final view matrix is T*R
- If we precompute T*R:

$$\begin{bmatrix} r_x & u_x & l_x & 0 \\ r_y & u_y & l_y & 0 \\ r_z & u_z & l_z & 0 \\ -e\cdot r & -e\cdot u & -e\cdot l & 1 \end{bmatrix}$$

  - r= right, u = up, l = look, e=eye point
  - Notice dot products in last row (pay careful attention to the signs!)

lookAt

- lookAt function: Often useful to be able to say "put the eye here, center of interest such and such, up = so and so
- We can code a function for this:

```
class Camera:
    ...
    def lookAt(self,eye,coi,up):
        #up = approximate up direction
        self.eye = eye.xyz
        self.coi = coi.xyz
        self.look = normalize(coi.xyz-self.eye)
        self.right = normalize(cross(up,self.look))
        #no normalize because look, right unit length
        #and mutually perpendicular
        self.up = cross(self.look,self.right)
        self.updateViewMatrix()
```

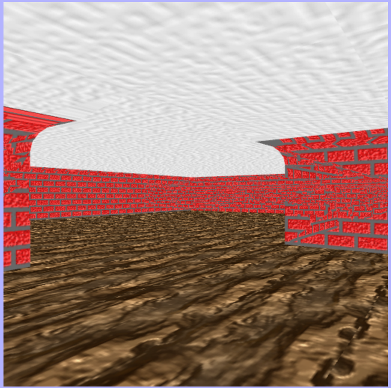
Camera

- We should also allow user to walk and turn and strafe
- Code: In-class

```
class Camera:
    ...
    def strafe(self,dr, du, dl):
        ...
    def turn(self, amt):        #yaw
        ...
    def roll(self, amt):
        ...
    def pitch(self,amt):
        ...
```

Assignment

- Allow the user to roam the 3D dungeon
  - WASD = forward/backward; ER = turn
  - We might use mouse-look later on
- Note that we don't have lighting yet, so everything is flat shaded
- We also expect lots of depth artifacts...We'll fix those next time



Changes

- Quick notes on what I changed from the previous lab:
  - Make camera 3D, not 2D
    - Add projection matrix to camera, camera.setUniforms
  - Teach shaders how to handle 3D
    - Uniforms.txt and VS need projMatrix
    - Make VS input position vec3 instead of vec2
    - gl_Position $\leftarrow$ all four components of vector-matrix multiply
  - Add event handling in main update() to strafe/turn camera
  - Don't need hero or enemies or particle systems or bullets or the boss or the starfield or the map object for the moment
    - Some of these will be coming back later though
  - The map is now just a Mesh
  - In Mesh: Change mesh position (but not texture) buffers to be 3D and glVertexAttribPointer so positions are 3D, not 2D

Sources

- E. Lengyel. Mathematics for 3D Game Programming and Computer Graphics. Charles River Media.

Created using LaTeX.

Main font: Gentium Book Basic, by Victor Gaultney. See http://software.sil.org/gentium/
Monospace font: Source Code Pro, by Paul D. Hunt. See https://fonts.google.com/specimen/Source+Code+Pro and http://sourceforge.net/adobe
Icons by Ulisse Perusin, Steven Garrity, Lapo Calamandrei, Ryan Collier, Rodney Dawes, Andreas Nilsson, Tuomas Kuosmanen, Garrett LeSage, and Jakub Steiner. See http://tango-project.org