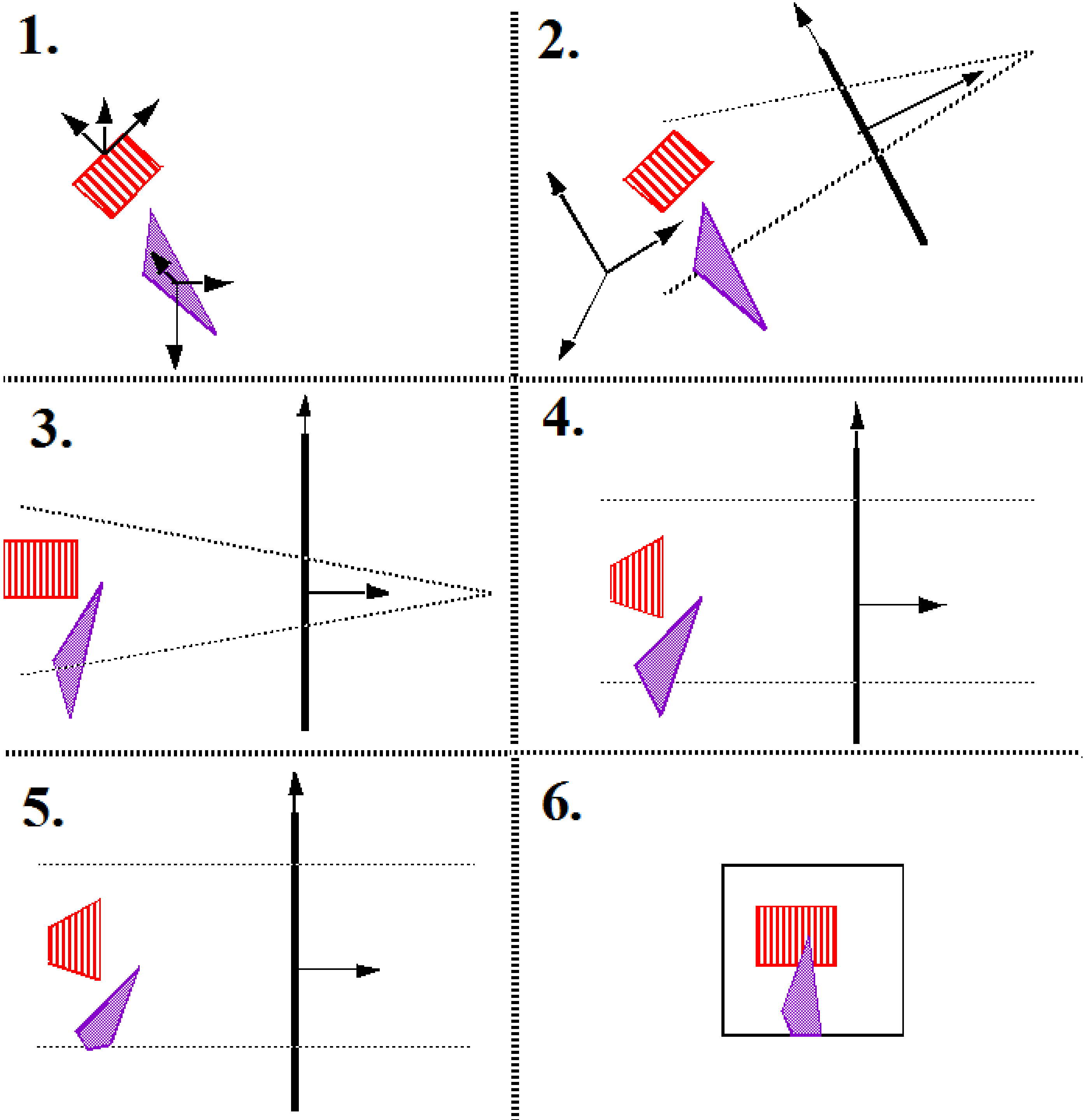
Viewing
and
Projection


1.
2.
3.
4.
5.
6.

OpenGL Graphics Geometry Pipeline

Object Space	World Space	Eye Space	Clipping Space	Canonical view volume	Screen Space

- **Object space**: coordinate space where each object is defined
- **World space**: all objects are put together into this same 3D space via affine transformations. Camera, lighting are also defined in this space.
- **Eye space**: camera at the origin, view direction coincides with the z axis. Near and far planes perpendicular to the z axis
- **Clipping space**: apply perspective transformation, but before perspective division. All points are in homogeneous coordinates, i.e., each point is represented by (x,y,z,w)
- **Canonical view volume**: A parallelpiped shape. Obtained after perspective division. Objects in this space are distorted (farther are smaller)
- **Screen space**: x and y coordinates are pixel coordinates, z coordinate used for screen-space hidden surface removal

OpenGL: Virtual Camera

- Camera's configuration is represented by **two** 4x4 matrices for vertex transformations: **Modelview** and **Projection** matrices
- Two things to maintain:

  1) Location of camera (eye) in the scene (**MODELVIEW matrix**)
     - Where is the camera?
     - Which direction is it pointing?
     - What is the orientation of the camera?

  2) Projection properties of the camera (**PROJECTION matrix**)
     - Depth of field?
     - Field of view in the x and y directions?

Modelview matrix & Projection matrix

- Object vertices are transformed from **object space** to **clipping space** by modeling matrices, viewing matrix and projection matrix

$$v_{clip} = \underbrace{Proj \times \underbrace{View \times \underbrace{Model \times v_{obj}}_{v_{obj} \text{ in world space}}}_{v_{obj} \text{ in eye space}}}_{v_{obj} \text{ in clipping space}}$$

- v_{obj} is in homogeneous coordinates. $Proj$, $View$ and $Model$ are all 4x4 matrices.

Modeling & Viewing

Modelview Matrix

- **Modelview** matrix comprises
  - **Modeling** transformations:
    - local coordinates → world coordinates
    - affine transformations, e.g., translation, rotation, scaling…
    - Be aware of the matrix multiplication order!
  - **Viewing** transformation:
    - world coordinates → eye coordinates
    - rigid-body transformations

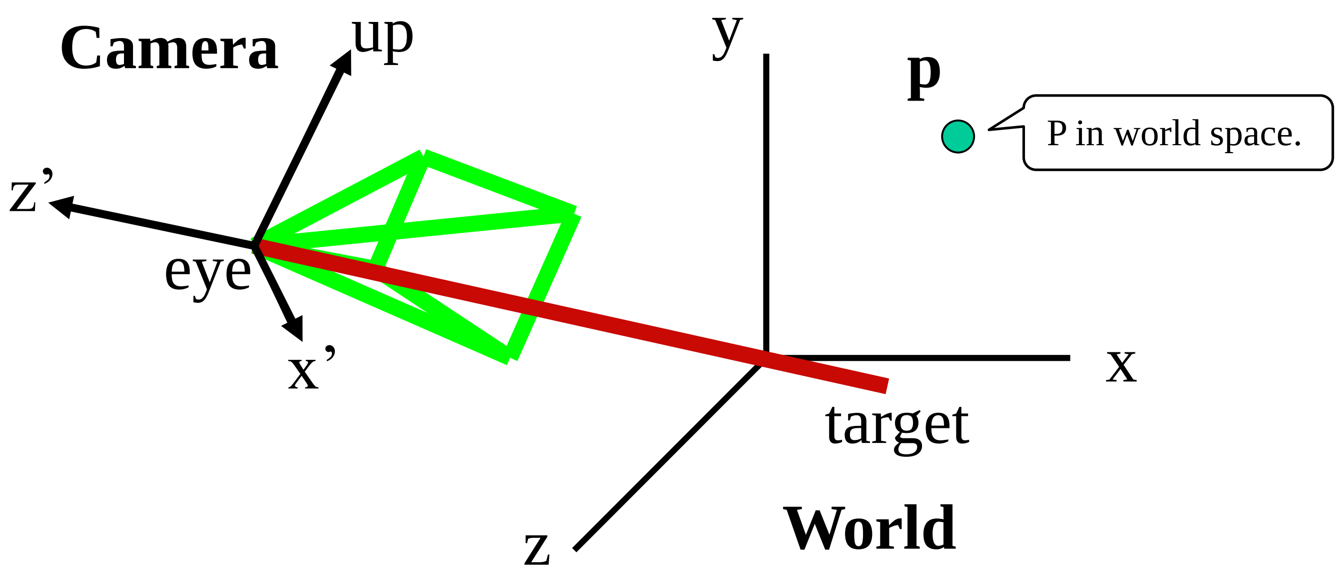
T*R*S

$$v_{eye} = \underbrace{View \cdot \underbrace{Model \cdot v_{obj}}_{v_{obj}\text{ in world space}}}_{v_{obj}\text{ in eye space}}$$

Local coord.

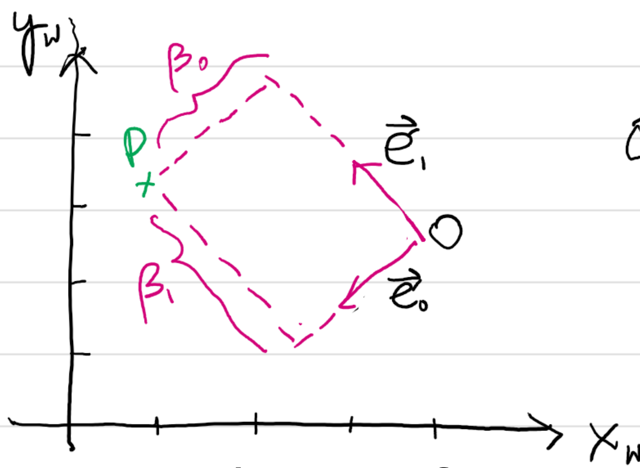
Viewing Transformation

- Given **eye position** & **target position** and an **up vector**:
  - Viewing direction: target – eye
  - Up vector specifies vertical orientation of camera
- Define the **eye coordinate system** (i.e., $View$ matrix)
  - Origin is at eye location
  - Z axis is opposite direction of viewing vector (**e2 = normalize(eye – target)**)
  - X axis is normal to the plane spanned by view vector and up vector, pointing to the right of viewer (**e0 = normalize(up** x **e2)**)
  - Y axis is orthonormal to x axis and z axis (**e1 = normalize(e2** x **e0)**)



Viewing Transformation

- Construct $View$ matrix using axis vectors $(\boldsymbol{e_0},\ \boldsymbol{e_1},\ \boldsymbol{e_2})$ and the eye position $(\boldsymbol{o})$



$$O + \beta_0 \vec{e}_0 + \beta_1 \vec{e}_1 = P$$

$$(\boldsymbol{e_0} \mid \boldsymbol{e_1} \mid \boldsymbol{o}) \begin{pmatrix} \beta_0 \\ \beta_1 \\ 1 \end{pmatrix} = P$$

P_{world} : coordinates of a point in the world space

P_{eye}: $(\beta_0,\ \beta_1,\ \beta_2)$ coordinates of P_{world} in eye space

– $P_{world} = \beta_0\ \boldsymbol{e_0} + \beta_1 \boldsymbol{e_1} + \beta_2\ \boldsymbol{e_2} + \boldsymbol{O}\ =\ \mathrm{M}\ P_{eye}$

where

$$M = (\boldsymbol{e_0} \mid \boldsymbol{e_1} \mid \boldsymbol{e_2} \mid \boldsymbol{o}) = \begin{pmatrix} e_{0x} & e_{1x} & e_{2x} & o_x \\ e_{0y} & e_{1y} & e_{2y} & o_y \\ e_{0z} & e_{1z} & e_{2z} & o_z \\ 0 & 0 & 0 & 1 \end{pmatrix} = T(o_x\ , o_y\ , o_z\ )R$$

Viewing Transformation

- $P_{world} = \beta_0\ \boldsymbol{e_0} + \beta_1 \boldsymbol{e_1} + \beta_2\ \boldsymbol{e_2} + \boldsymbol{O}\ =\ \mathrm{M}\ P_{eye}$

  where

$$M = (\boldsymbol{e_0} \mid \boldsymbol{e_1} \mid \boldsymbol{e_2} \mid \boldsymbol{o}) = \begin{pmatrix} e_{0x} & e_{1x} & e_{2x} & o_x \\ e_{0y} & e_{1y} & e_{2y} & o_y \\ e_{0z} & e_{1z} & e_{2z} & o_z \\ 0 & 0 & 0 & 1 \end{pmatrix} = T(o_x\ , o_y\ , o_z\ )R$$

- $P_{eye} = M^{-1} P_{world}$. Invert M to get $View$ matrix.

$$View = M^{-1}$$

- M^{-1} can be easily computed:

$$View = M^{-1} = R^{-1} T^{-1} = R^{\mathrm{T}} \cdot T(-o_x\ , -o_y\ , -o_z)$$

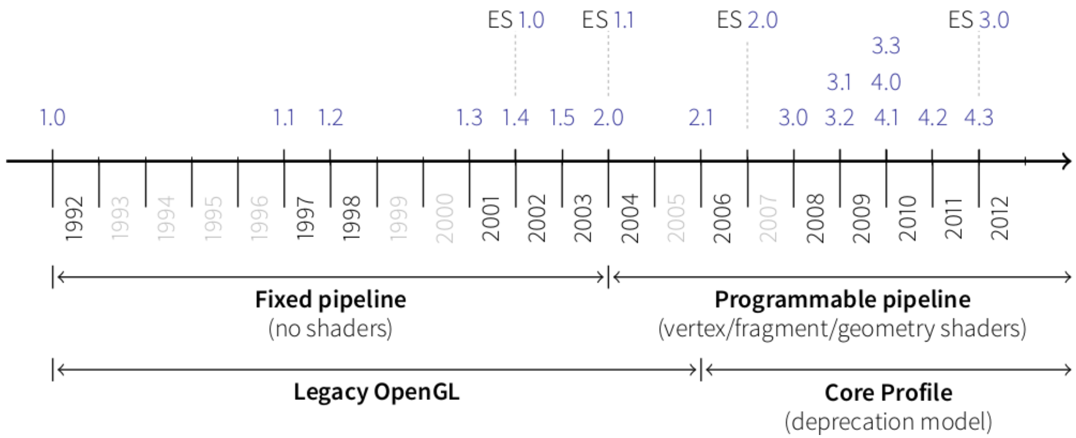
$$R^{\mathrm{T}} = \begin{pmatrix} e_{0x} & e_{0y} & e_{0z} & 0 \\ e_{1x} & e_{1y} & e_{1z} & 0 \\ e_{2x} & e_{2y} & e_{2z} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad M^{-1} = \begin{pmatrix} e_{0x} & e_{0y} & e_{0z} & -\boldsymbol{e}_0 \cdot \boldsymbol{O} \\ e_{1x} & e_{1y} & e_{1z} & -\boldsymbol{e}_1 \cdot \boldsymbol{O} \\ e_{2x} & e_{2y} & e_{2z} & -\boldsymbol{e}_2 \cdot \boldsymbol{O} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Viewing Transformation

- What if $View$ matrix is the **identity matrix?**
- Then, the eye coordinate system coincides with the world coordinate system
  - Camera is at origin of world space
  - Looking down the negative z axis
  - Right of viewer is positive x axis
  - Up direction is positive y axis

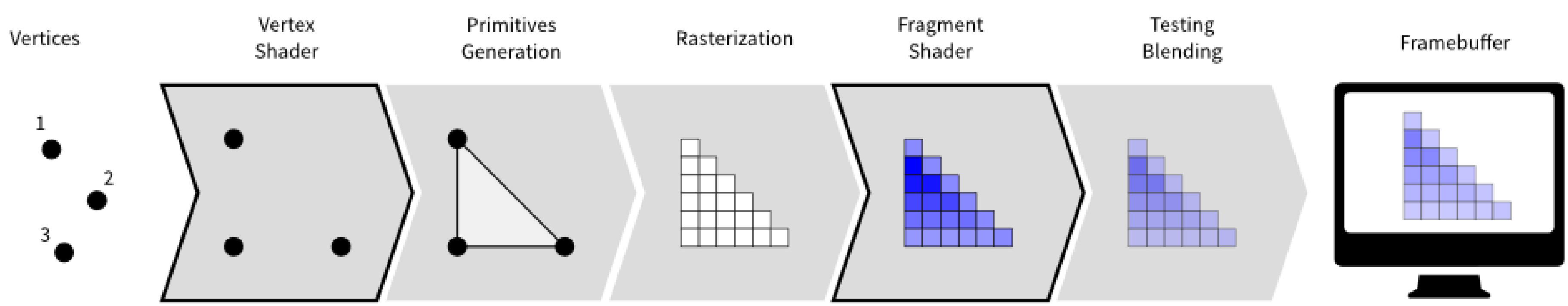
OpenGL Programming

- In traditional fixed OpenGL pipeline:
  - OpenGL maintains a **MODELVIEW** matrix for you.
  - To modify *Model* matrix, use *glTranslatef(), glRotatef(), glScalef()*…
  - To modify *View* matrix, call *gluLookAt(eyeX, eyeY, eyeZ, centerX, centerY, centerZ, upX, upY, upZ)*
- But these are obsolete in modern programmable OpenGL pipeline.



OpenGL Programming

- In modern programmable OpenGL pipeline:
  - You need to self-maintain the 4x4 $View$ and $Model$ matrices as variables and do the construction on your own in your program.
  - During rendering, feed these matrices as uniform variables to shaders. (Rendering part of this course will cover this.)



OpenGL Programming

- In modern programmable OpenGL pipeline:
  - Alternatively, you may use existing **3rd party matrix libraries** to ease the pain during programming. For example, in C++,
    - **GLM** (https://glm.g-truc.net), specifically designed for OpenGL
    - **Eigen**(http://eigen.tuxfamily.org), more general linear algebra library, widely used, contains many linear algebra routines.

  - Examples using GLM:
    - A scaling matrix:

      glm::mat4 modelMat = glm::scale(2.0f, 2.0f ,2.0f);
    - View matrix:

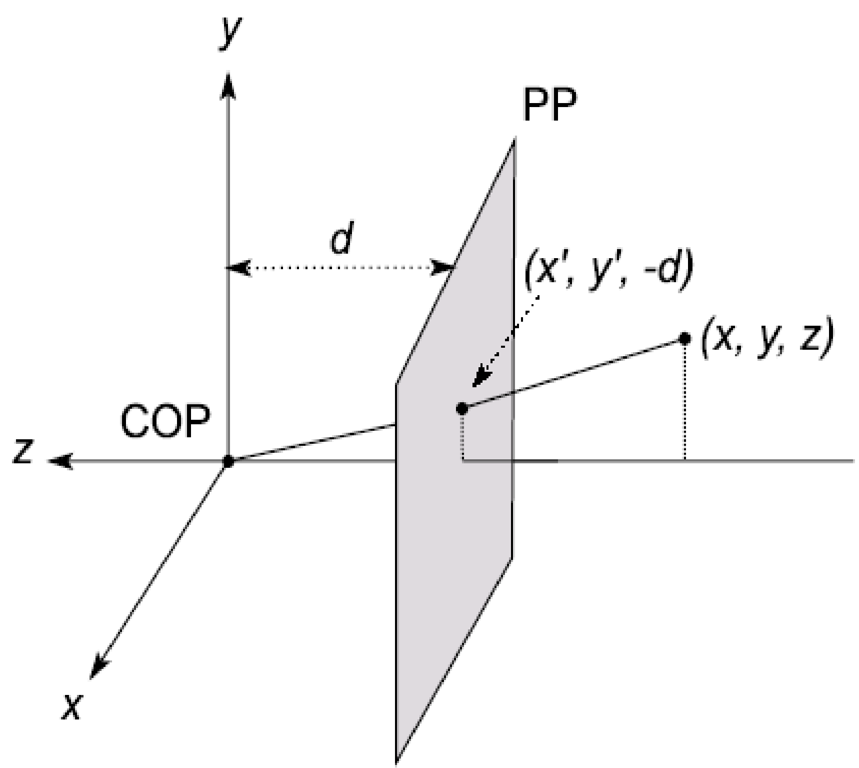
      glm::mat4 viewMat = glm::lookAt(cameraPosition, cameraTarget, upVector);
    - Modelview matrix:

      *glm::mat4 modelviewMat = viewMat * modelMat;*

Projection

Derivation of perspective projection

- Consider the projection of a point onto the projection plane:



By similar triangles, we can compute how much the x and y coordinates are scaled:

$$\frac{y}{-z} = \frac{y'}{d}$$

$$y' = -\frac{dy}{z}$$

Similarly,

$$x' = -\frac{dx}{z}$$

Division by z cannot be represented by 3x3 matrix.

Perspective projection

- How to represent the perspective projection as a matrix equation? Introduce **homogeneous coordinates**, go to projective space. (First, assume no depth)

$$\begin{bmatrix} x^* \\ y^* \\ w^* \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{-1}{d} & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ \frac{-z}{d} \end{bmatrix}$$

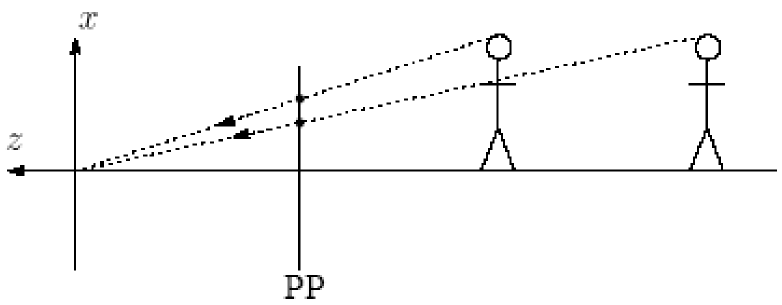
- By performing the **perspective division,** we get the correct projected coordinates in Euclidean space:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} x^* / w^* \\ y^* / w^* \\ w^* / w^* \end{bmatrix} = \begin{bmatrix} -\frac{x}{z} d \\ -\frac{y}{z} d \\ 1 \end{bmatrix}$$

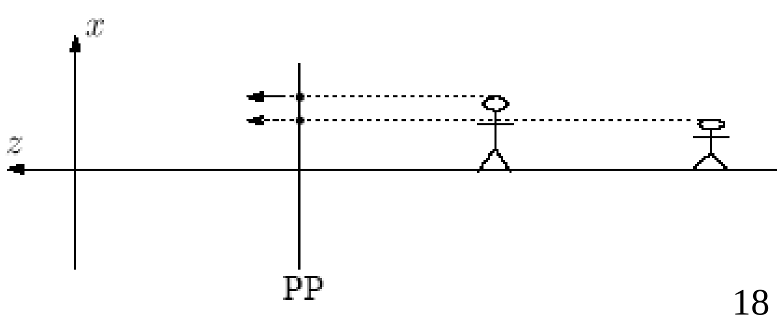
Projection normalization

- The **division step** converts a **perspective projection** to an **orthogonal projection** (why? see figure below)
  - Lines through the eye are mapped into **lines parallel to the z-axis**
  - View frustum is transformed into a **canonical view volume** (x±1, y ±1, z ±1), hence called **normalization**

What does this imply about the shape of things after projection normalization?



Then we are free to do a simple parallel projection to get the 2D image.



OpenGL perspective projection

- So far, we discuss projection with **no depth** (z is always –d)
- In graphics, we need **depth** for **hidden surface removal**
  - When two points project to the same point on the image plane, we need to know which point is closer to the eye
- But, actual distance is cumbersome to compute
- Sufficient to use **pseudodepth**
  - what is a good choice?

OpenGL perspective projection

- Choose a function that has the ***same denominator*** as x and y so that the projection can be represented as a matrix
- Projection plane at $z = -n$. Thus $d = n$

projection with no depth

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{-1}{d} & 0 \end{bmatrix} \equiv \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

The two matrices have the same effect when transforming points in homogeneous coordinates.

projection with depth

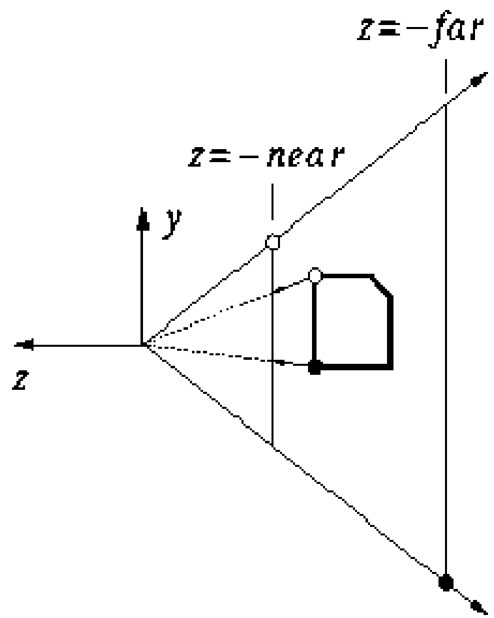
$$\begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ \boxed{0 \quad 0 \quad a \quad b} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ \boxed{z} \\ 1 \end{bmatrix} = \begin{bmatrix} nx \\ ny \\ \boxed{az+b} \\ -z \end{bmatrix} \longrightarrow \begin{bmatrix} -\frac{nx}{z} \\ -\frac{ny}{z} \\ \boxed{-\frac{az+b}{z}} \\ 1 \end{bmatrix}$$

- This matrix leaves x and y coordinates of points with $z = -n$ unchanged.
- What choice of a & b?

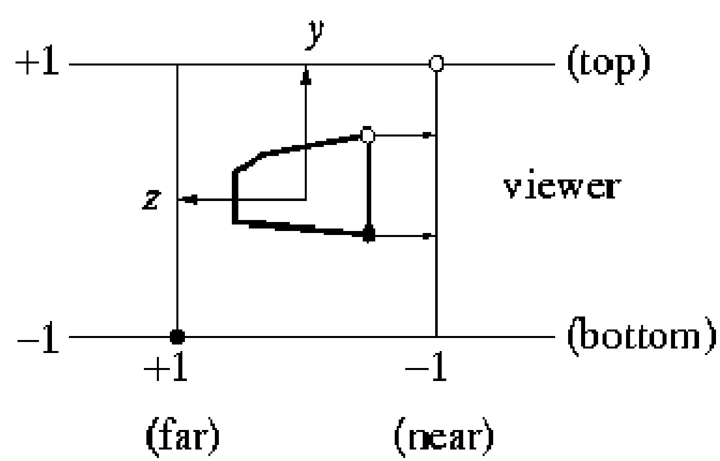
OpenGL Perspective transformation

$$z = -n \quad \Rightarrow \quad z' = -1$$
$$z = -f \quad \Rightarrow \quad z' = +1$$

Note: this transformation involves a “reflection”



Viewing frustum



Canonical view volume

OpenGL perspective projection

$$z' = -\frac{az+b}{z}$$

OpenGL: Choose a and b such that the pseudodepth z′ is in [-1, 1].

Setting 2 constraints:

$$z = -n \quad \Rightarrow \quad z' = -1$$
$$z = -f \quad \Rightarrow \quad z' = +1$$

Solving 2 linear equations with 2 unknowns a and b:

$$a = \frac{f+n}{n-f}$$

$$b = \frac{2fn}{n-f}$$

OpenGL perspective projection

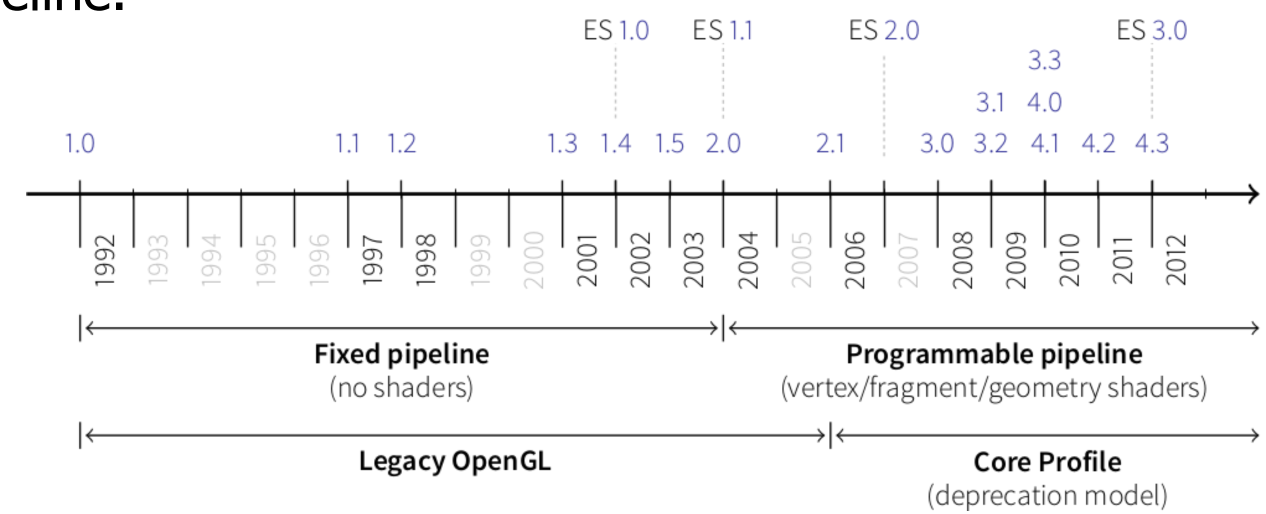
- The view frustum may not center along the view vector. If so, need to first shear the window to center it, then scale in both x and y to also map to [-1,1] range.
- Projection matrix

$$M_{persp}S = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f+n}{n-f} & \frac{2fn}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

- Refer http://www.songho.ca/opengl/gl_projectionmatrix.html for more details

OpenGL Programming

- In traditional fixed OpenGL pipeline:
  - OpenGL helps you maintain a **PROJECTION** matrix.
  - To construct $Proj$ matrix, call *glFrustum(left, right, bottom, top, nearVal, farVal)*
  - The other alternative function is *gluPerspective(fovY, aspect, zNear, zFar)*
- But again, these are obsolete in modern programmable OpenGL pipeline.

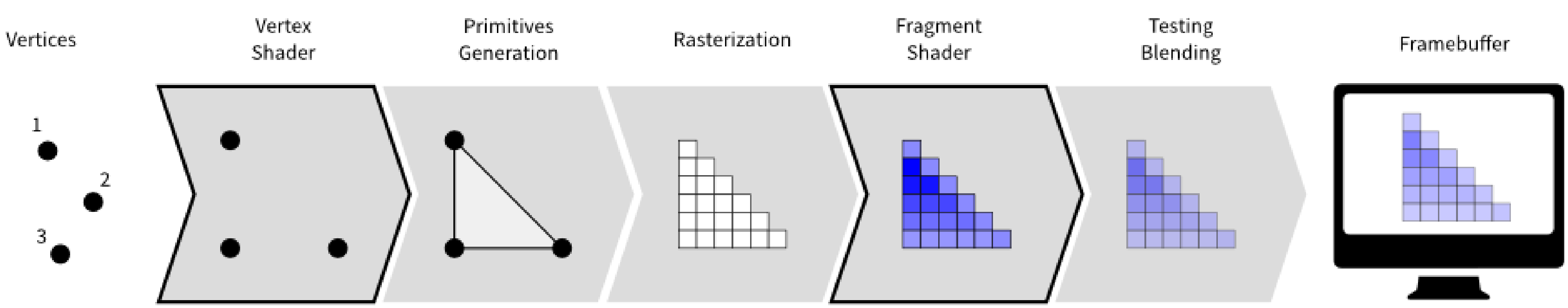


OpenGL Programming

- In modern programmable OpenGL pipeline:
  - You need to self-maintain the 4x4 $Proj$ matrix and do the construction on your own in your program.
  - During rendering, feed this matrix as a uniform variable to shaders. (Rendering part of this course will cover this.)
  - Examples in GLM:

*glm::mat4 projMat = glm::**frustum** (left, right, bottom, top, near, far);*

*glm::mat4 projMat = glm::**perspective**(glm::radians(45.0f), width /height, 0.1f, 100.0f);*

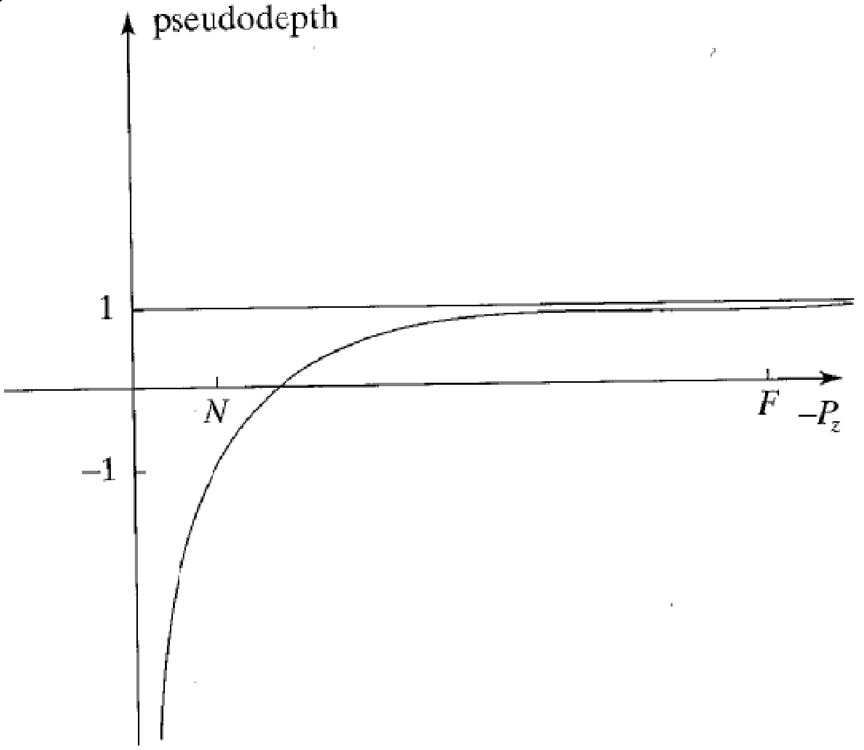


Near/far clipping

Why need near/far clipping planes?

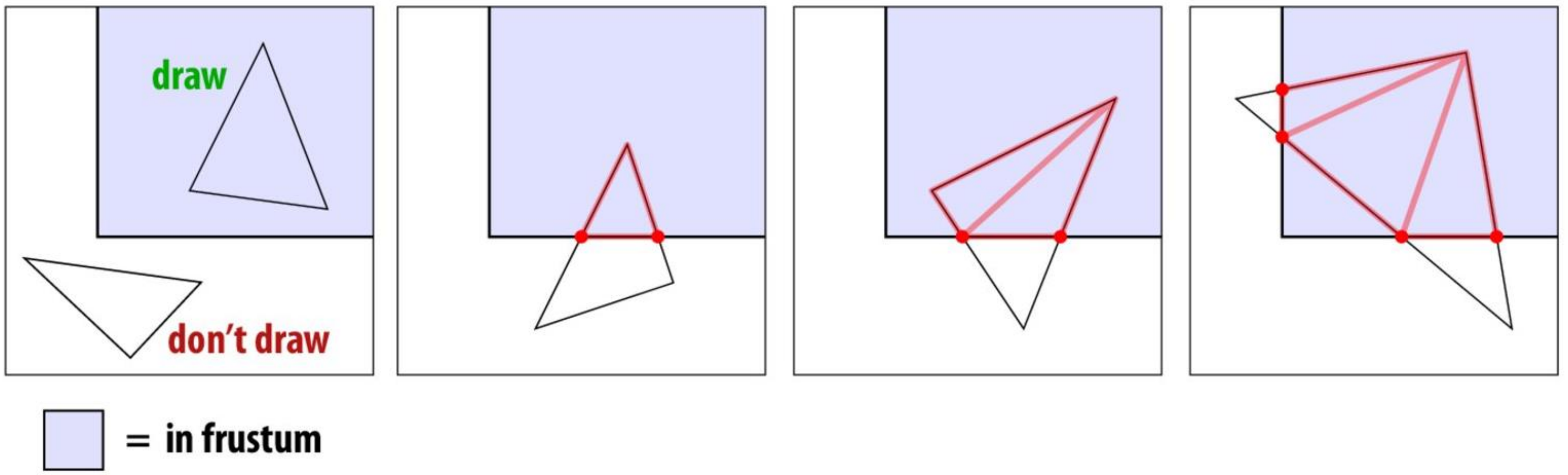
- Deals with finite precision of depth buffer/limitations on storing depths as floating point values

- Some triangles may have vertices both in front & behind eye! (causes headaches for rasterization)



Clipping

- Triangulate resulting clipped polygons



OpenGL Clipping

- Transforming to canonical view volume simplifies the clipping process
  - sides are aligned with the coordinate axes
- But OpenGL performs clipping in homogeneous space (why?)
  - After applying M_{persp} , before perspective division
- A point (x, y, z, w), where w is positive, is in the canonical view volume if

$$-1 < \frac{x}{w} < 1, \quad -1 < \frac{y}{w} < 1, \quad -1 < \frac{z}{w} < 1$$

- Thus, in homogeneous space, the clipping limits are

$$-w < x < w, \quad -w < y < w, \quad -w < z < w$$