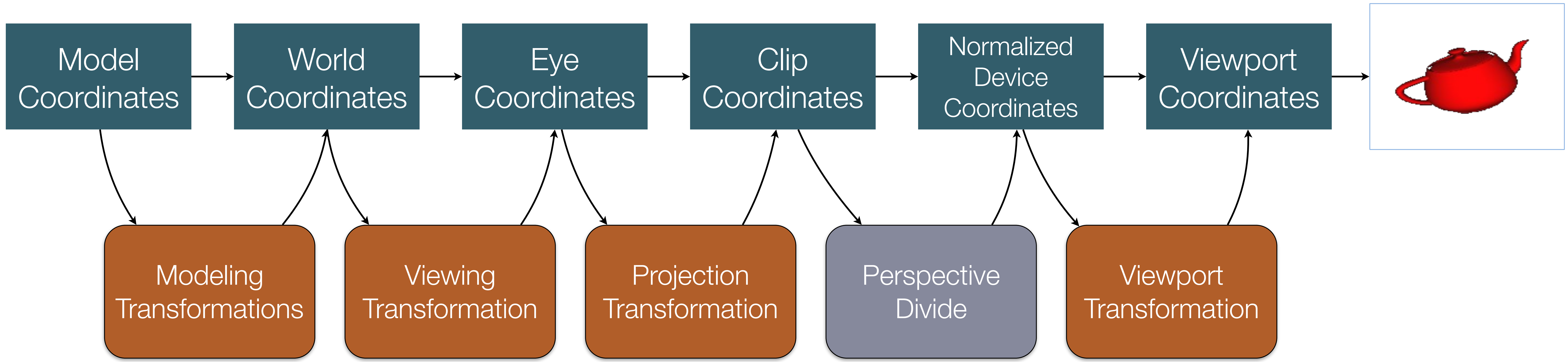
Coordinate Systems

CS 385 - Class 5
10 February 2022

Coordinate Systems in Computer Graphics



Projection Transformations

Aside: Some Mathematics

- We can encode a line equation (in *slope-intercept form*) into a matrix

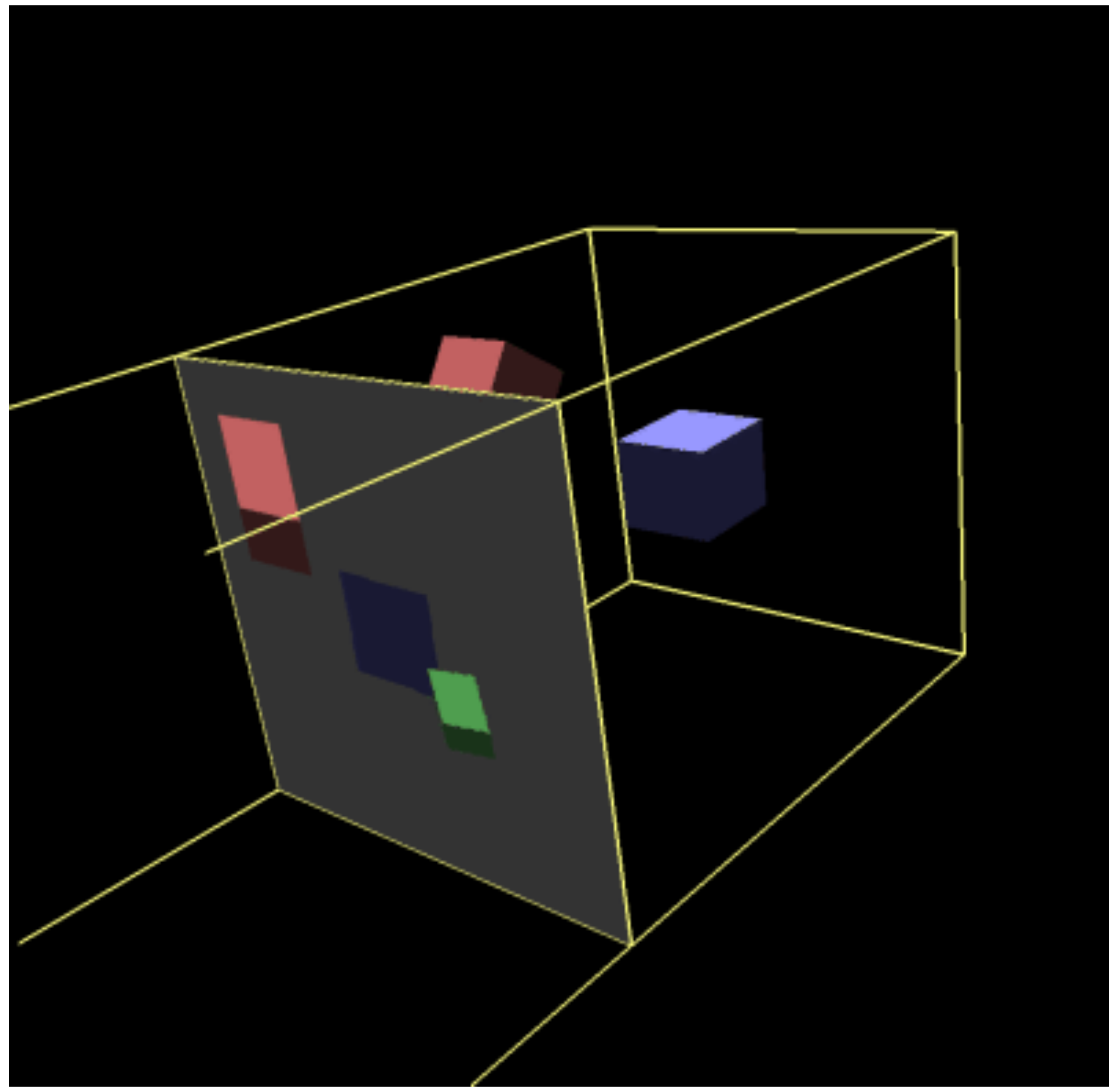
$$y = mx + b \Rightarrow \begin{pmatrix} y \\ 1 \end{pmatrix} = \begin{pmatrix} m & b \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ 1 \end{pmatrix}$$

Orthographic Projections

- Projects objects onto the imaging plane without distortion

- An object's size is constant regardless of distance from the eye

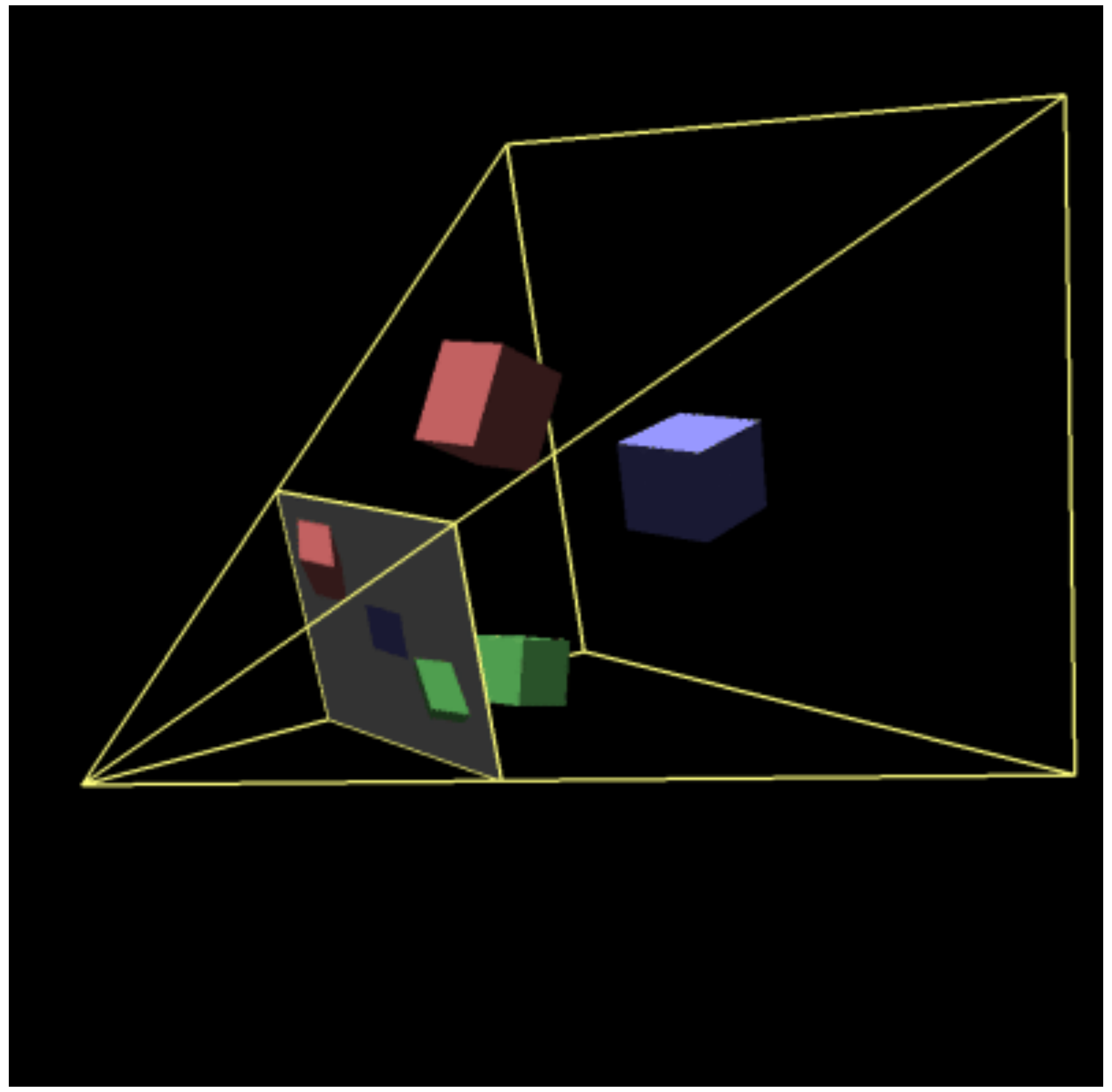
- Use `ortho(l, r, b, t, n, f);`



$$\begin{pmatrix} \frac{2}{r-l} & 0 & 0 & \frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & \frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & \frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Perspective Projections

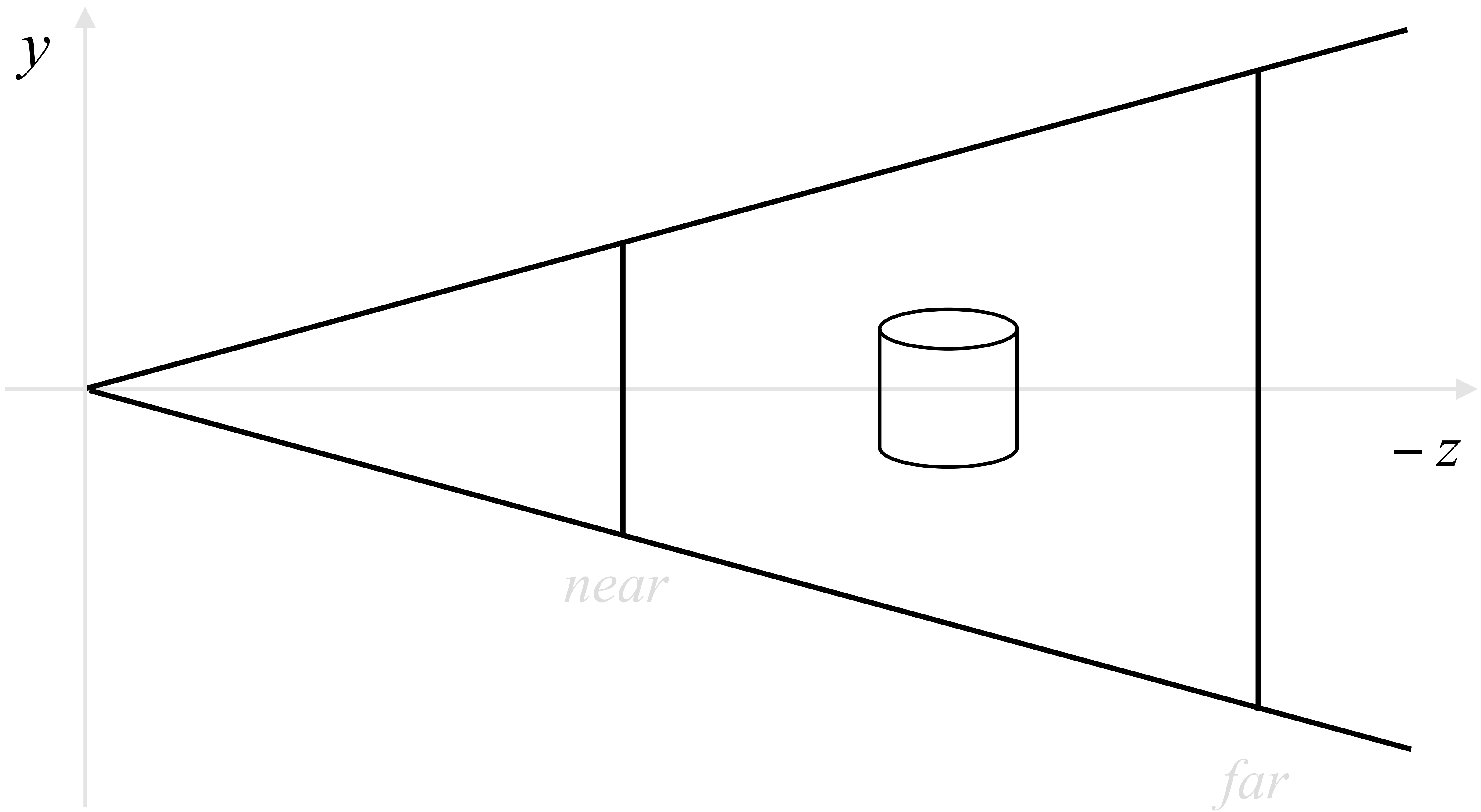
- “Works” similar to your eyes
  - objects farther from the viewer appear smaller
- Assumptions about perspective projections
  - the eye is located at the origin (apex of the pyramid)
  - line-of-sight is down the negative z-axis
- Use `perspective(fovy, aspect, n, f)` for a view-centered projection



$$\begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(n+f)}{f-n} & \frac{-2nf}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Two Important Things about Perspective Projections

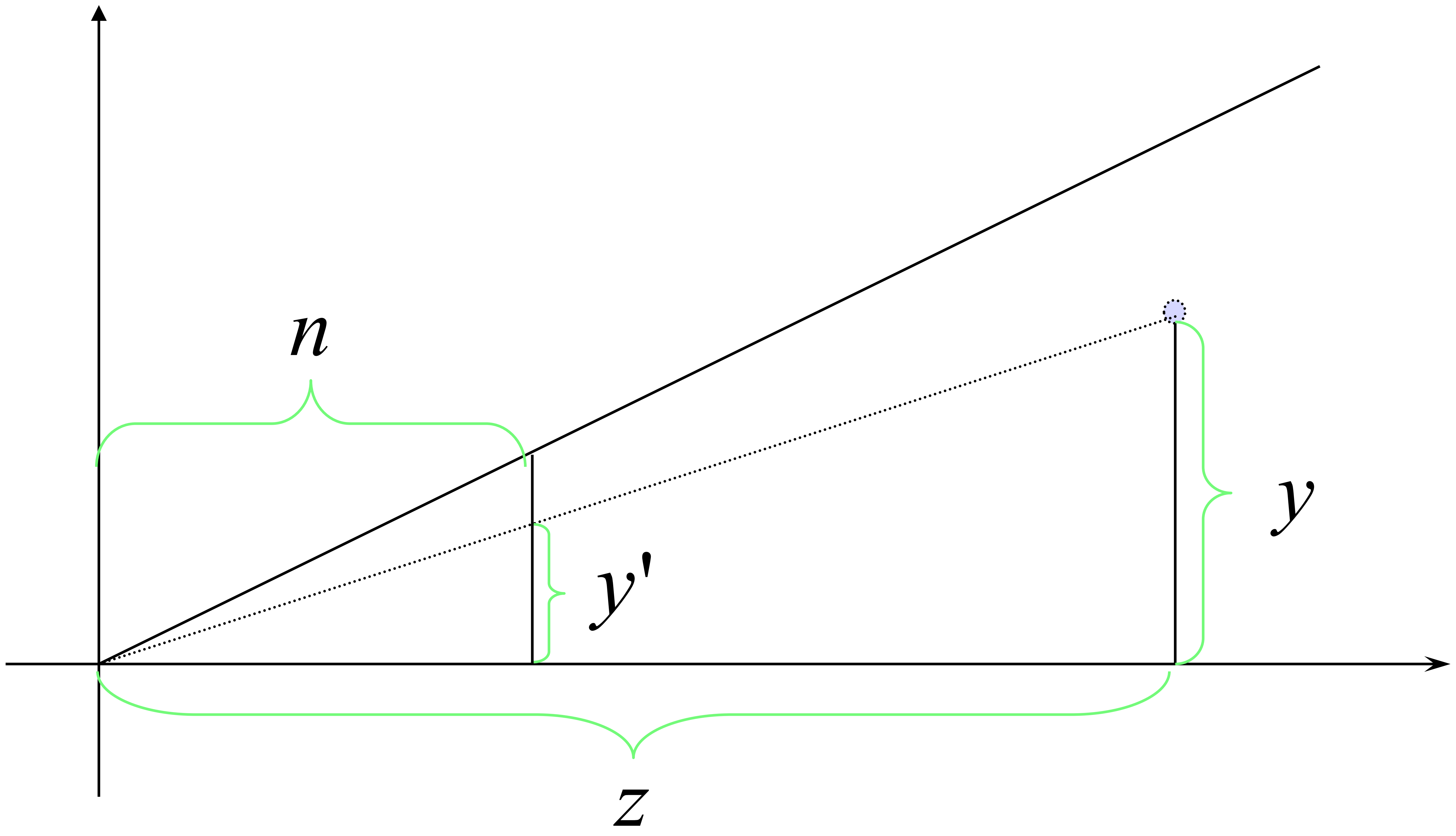
- The "eye" is located at the origin
  - as we'll see, the viewing transformation takes care of this
- The viewer is looking down the *negative* z-axis (-z)



Perspective Projections (cont.)

- Based on similar triangles

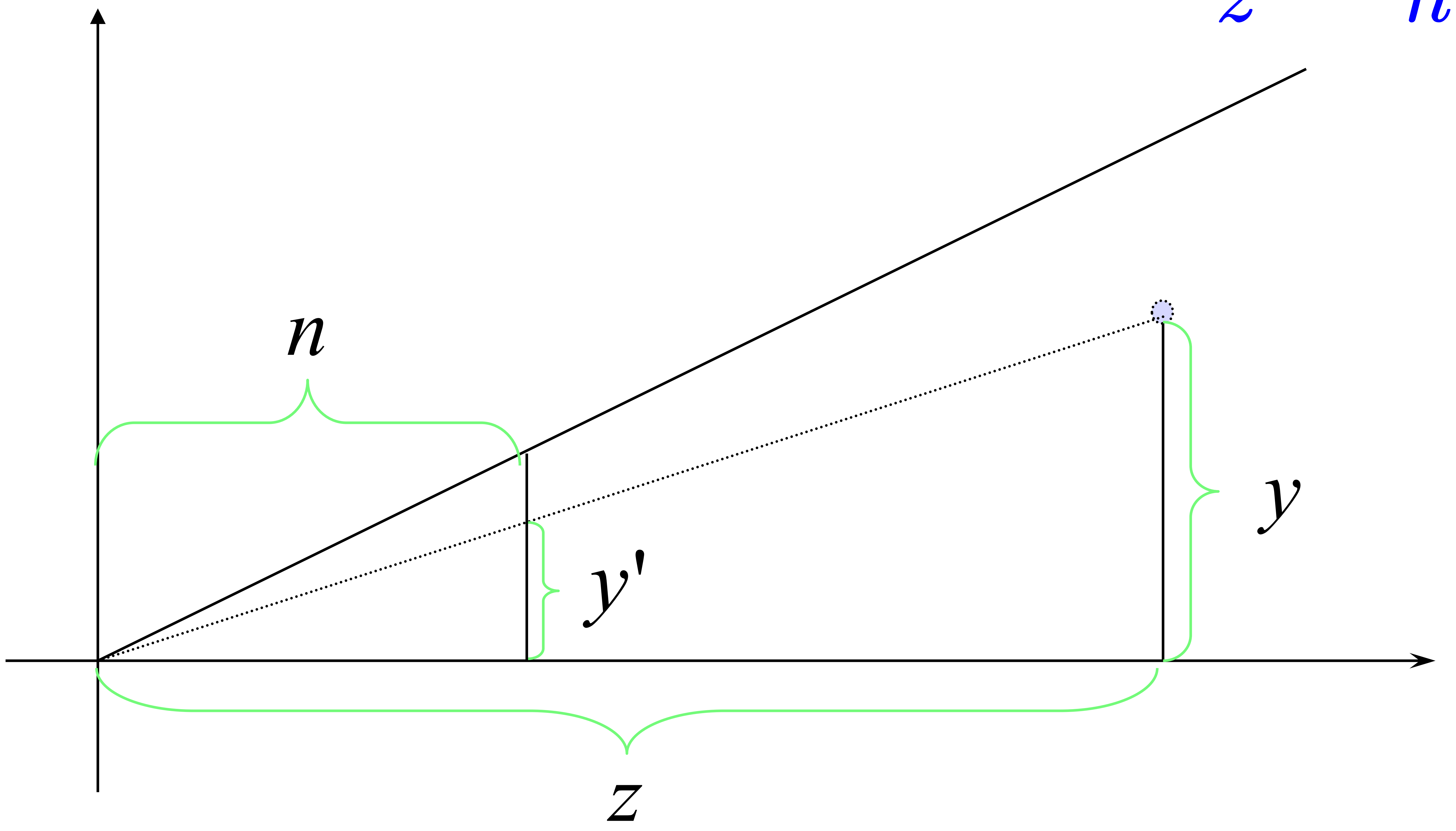
$$\frac{y}{z} = \frac{y'}{n}$$



Perspective Projections (cont.)

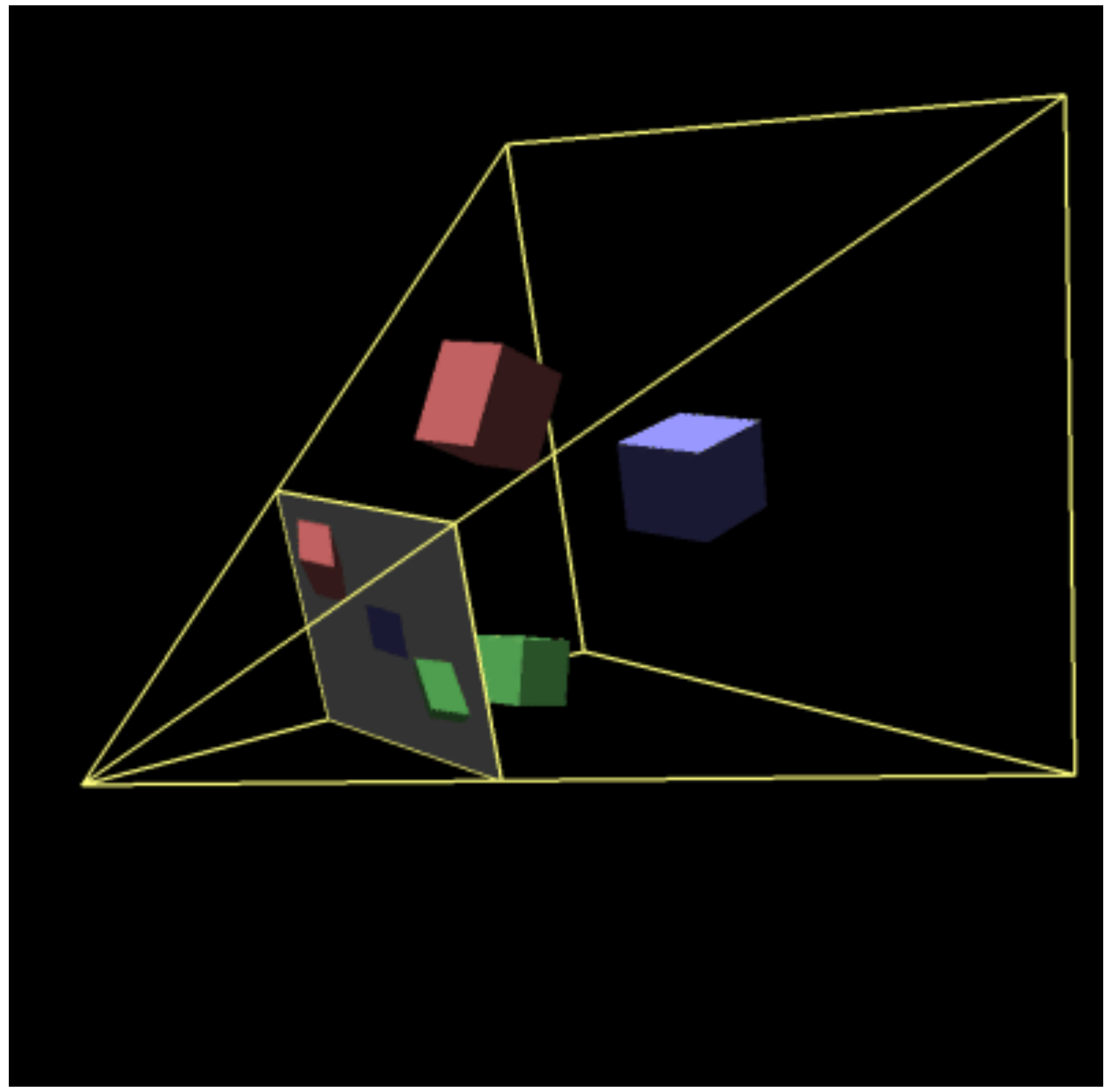
- Based on similar triangles

$$\frac{y}{z} = \frac{y'}{n} \Rightarrow y' = \frac{ny}{z}$$



Perspective Projections

- "Works" similar to your eyes
  - objects farther from the viewer appear smaller
- Assumptions about perspective projections
  - the eye is located at the origin (apex of the pyramid)
  - line-of-sight is down the negative z-axis
- Use `perspective(fovy, aspect, n, f)` for a view-centered projection



$$\begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(n+f)}{f-n} & \frac{-2nf}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Handling a Canvas Resize

- When a window changes size, adjust the viewport
  - the canvas's size is adjusted automatically
- Here's where we need to take the aspect ratio into account with our projection

```
var canvas;
var P; // our Projection transformation

function resize() {
  var width = canvas.clientWidth,
      height = canvas.clientHeight;

  gl.viewport(0, 0, width, height);

  aspect = width/height;

  P = perspective(fovy, aspect, near, far);
 }

window.onresize = resize;
```

Aside: Determining $near$ and far

- Guidelines

$$0 < near < far$$

- How you configure your viewing frustum is application dependent
- For example
  - CAD applications often want to see an entire object for any orientation
    - determine a *bounding volume* for the object
    - specify frustum parameters so that the entire object is inside the frustum

Aside: Determining *near* and *far* (cont.)

- "Viewer" games (e.g., FPS, driving, etc.)
  - $near$ will often be very close to the eye (i.e., $near \approx 1$)
  - far probably dictated by the "environment"

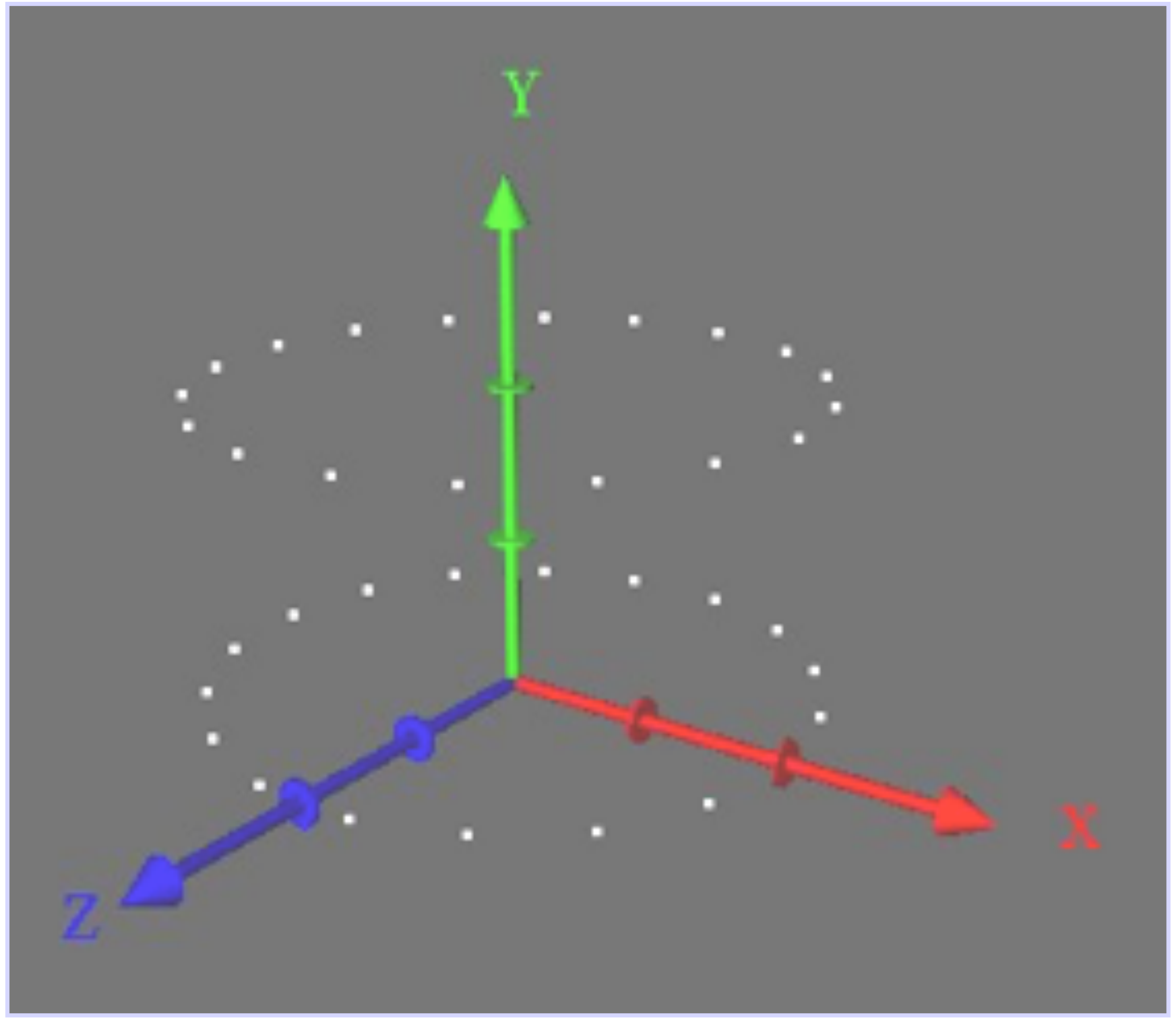
Indoors	Rooms, walls, portals, etc.
Outdoors	Mountains, Skybox, etc.

- There are some other tricks

Modeling Transformations

Modeling Objects

- Recall objects are composed of *geometric primitives*
  - each primitive is specified by its vertices
- The *modeling process* is merely determine the object's vertices
- Model objects around the *origin* to make life simple

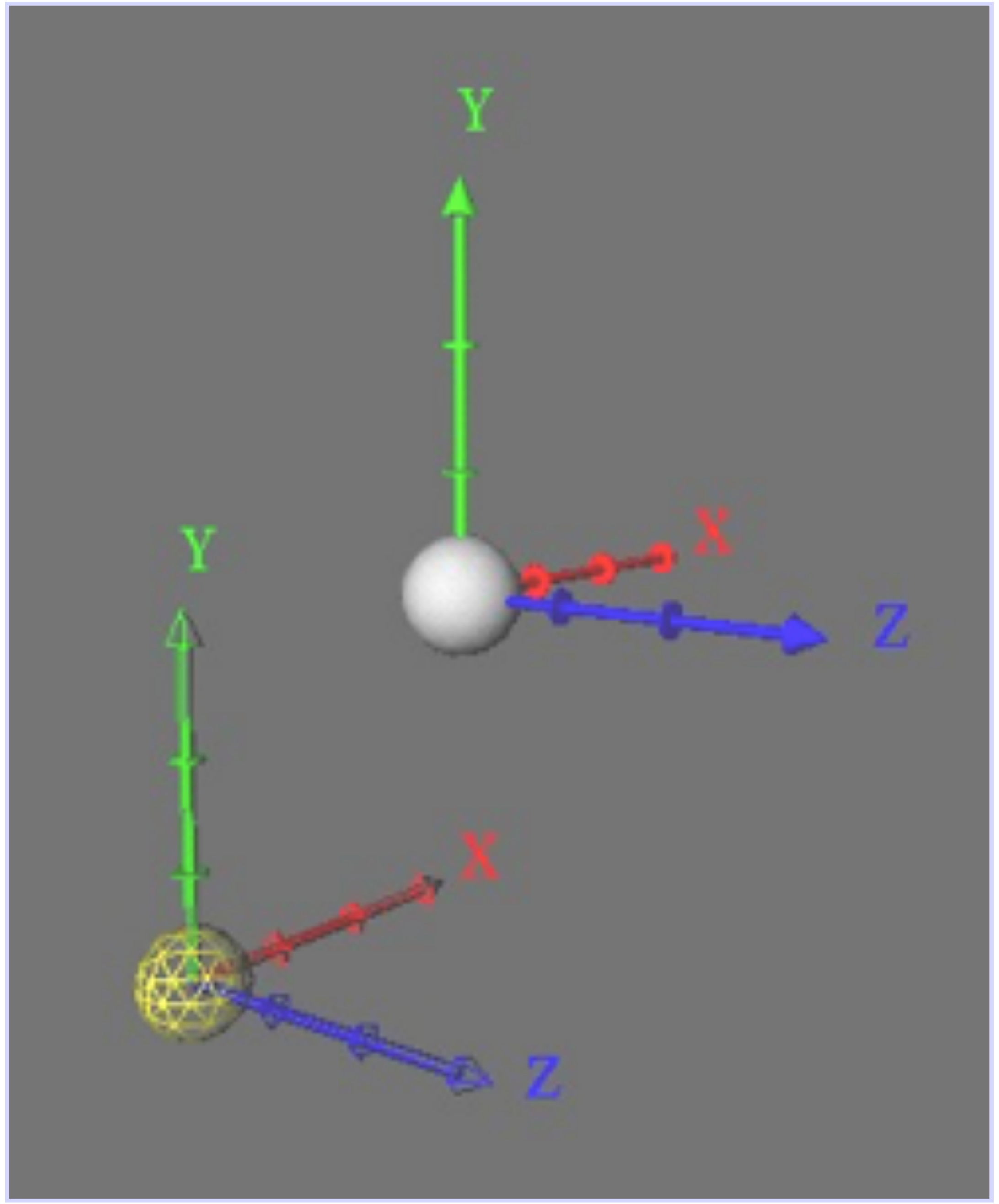


Modeling Transformations

- Transform *object coordinates* into *world coordinates*
- They don't operate on objects, but rather *coordinate systems*
- Vertices are specified relative to the *current coordinate system*

Translation

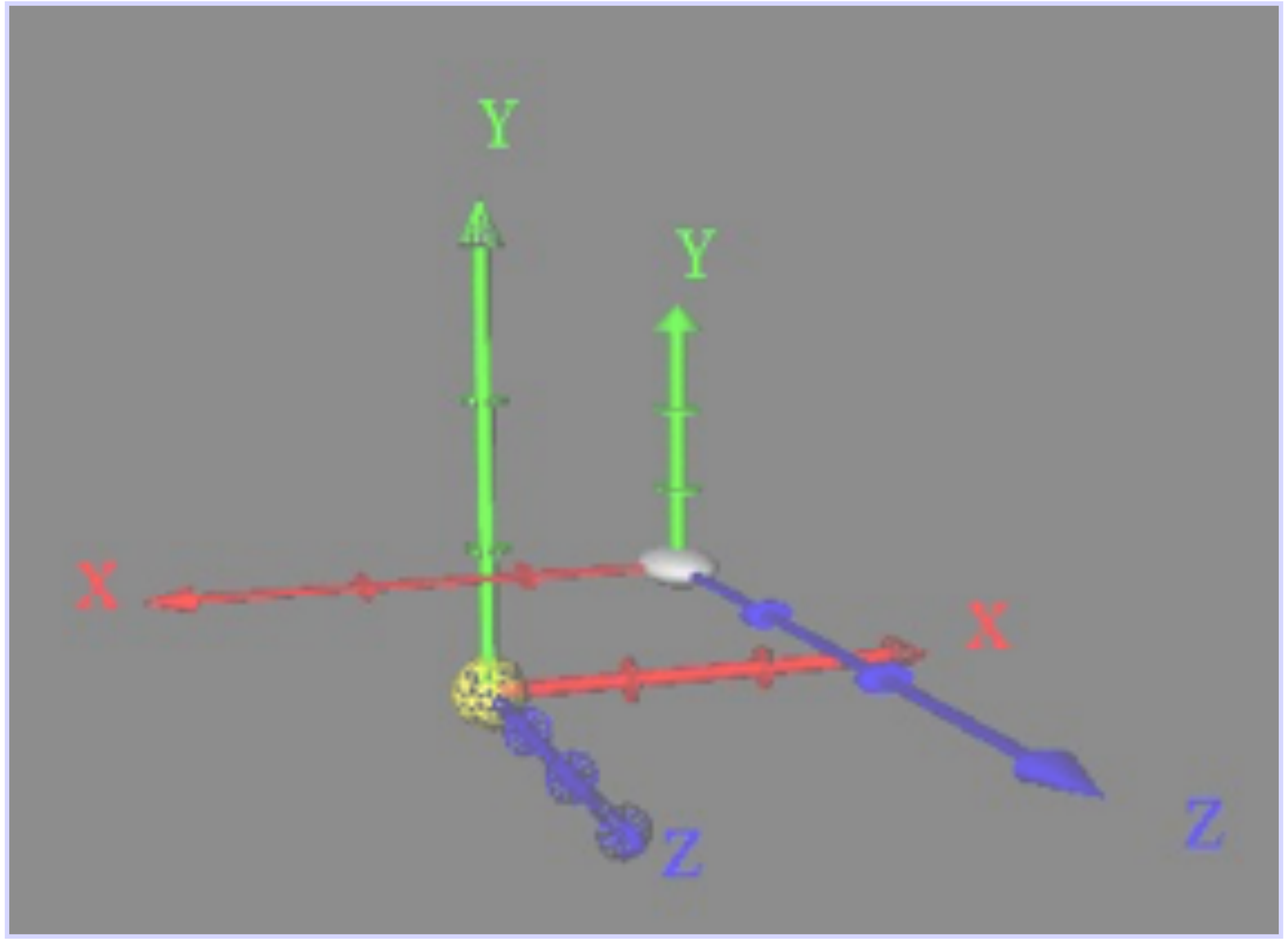
- Moves the origin to a new location
- Use `translate(x, y, z)`



$$T(t_x, t_y, t_z) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Scale

- Scales the coordinate system around the origin
- Use `scale(x, y, z)`



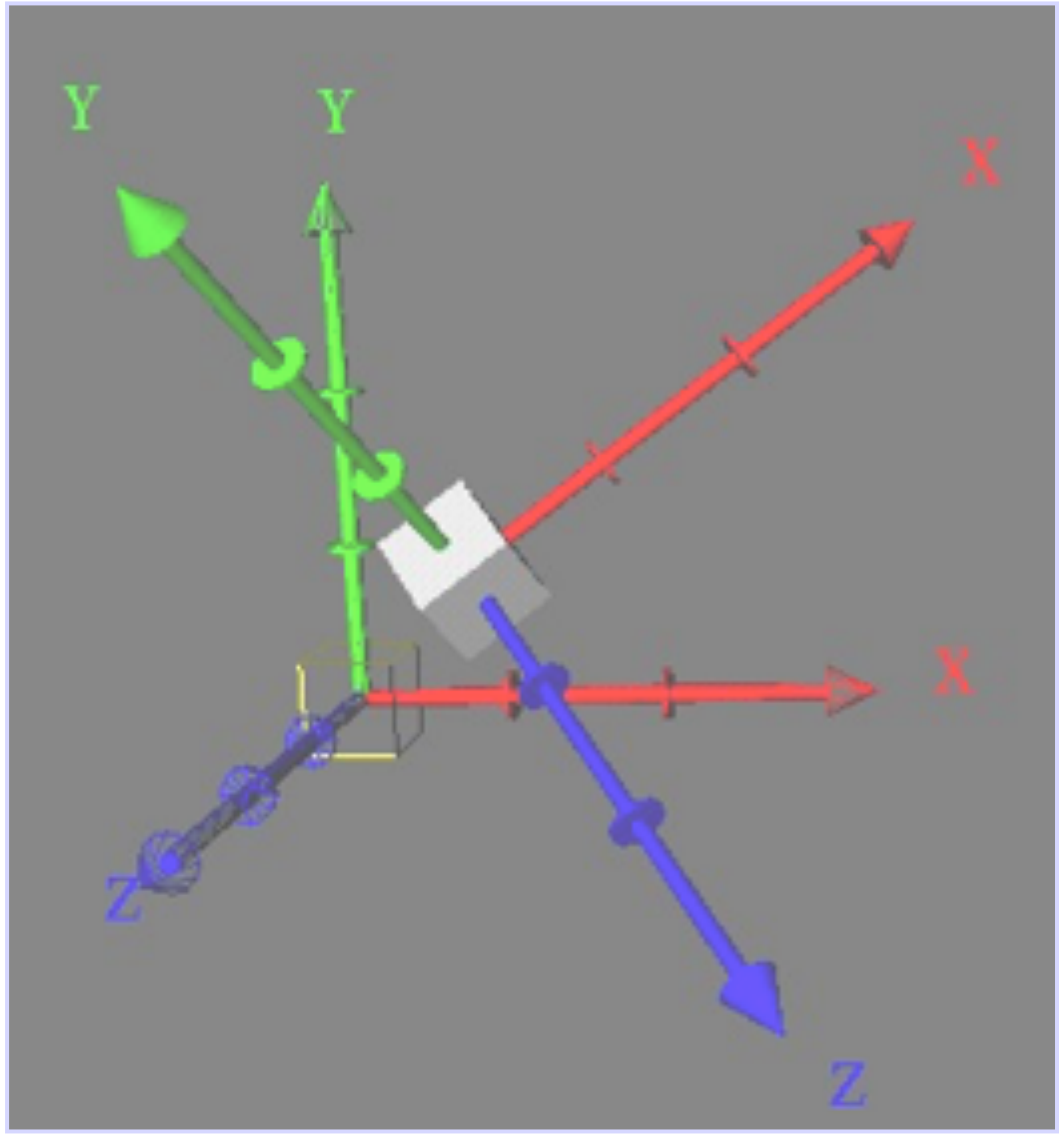
There's also a translation applied in the above illustration

$s > 1$	stretch
$0 > s \geq 1$	shrink
0	decimate
$-1 \leq s < 0$	reflect/shrink
$-1 > s$	reflect/strecth

$$S(s_x, s_y, s_z) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotation

- Rotates the coordinate system around an axis
  - use `rotate(`θ`, `$\vec{v}$`)`



There's also a translation applied in the above illustration

$$\vec{v} = \begin{pmatrix} x & y & z \end{pmatrix}$$

$$\vec{u} = \frac{\vec{v}}{\|\vec{v}\|} = \begin{pmatrix} x' & y' & z' \end{pmatrix}$$

$$M = \vec{u}^t\vec{u} + \cos(\theta)(I - \vec{u}^t\vec{u}) + \sin(\theta)S$$

$$S = \begin{pmatrix} 0 & -z' & y' \\ z' & 0 & -x' \\ -y' & x' & 0 \end{pmatrix}$$

$$R_{\vec{v}}(\theta) = \begin{pmatrix} & M & & \begin{matrix} 0 \\ 0 \\ 0 \end{matrix} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotation (cont.)

- Right!!!!
- Angel provides convenience functions for rotating around the principal axes:
  - use `rotateX(angle)`
  - use `rotateY(angle)`
  - use `rotateZ(angle)`

$$R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_y(\theta) = \begin{pmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

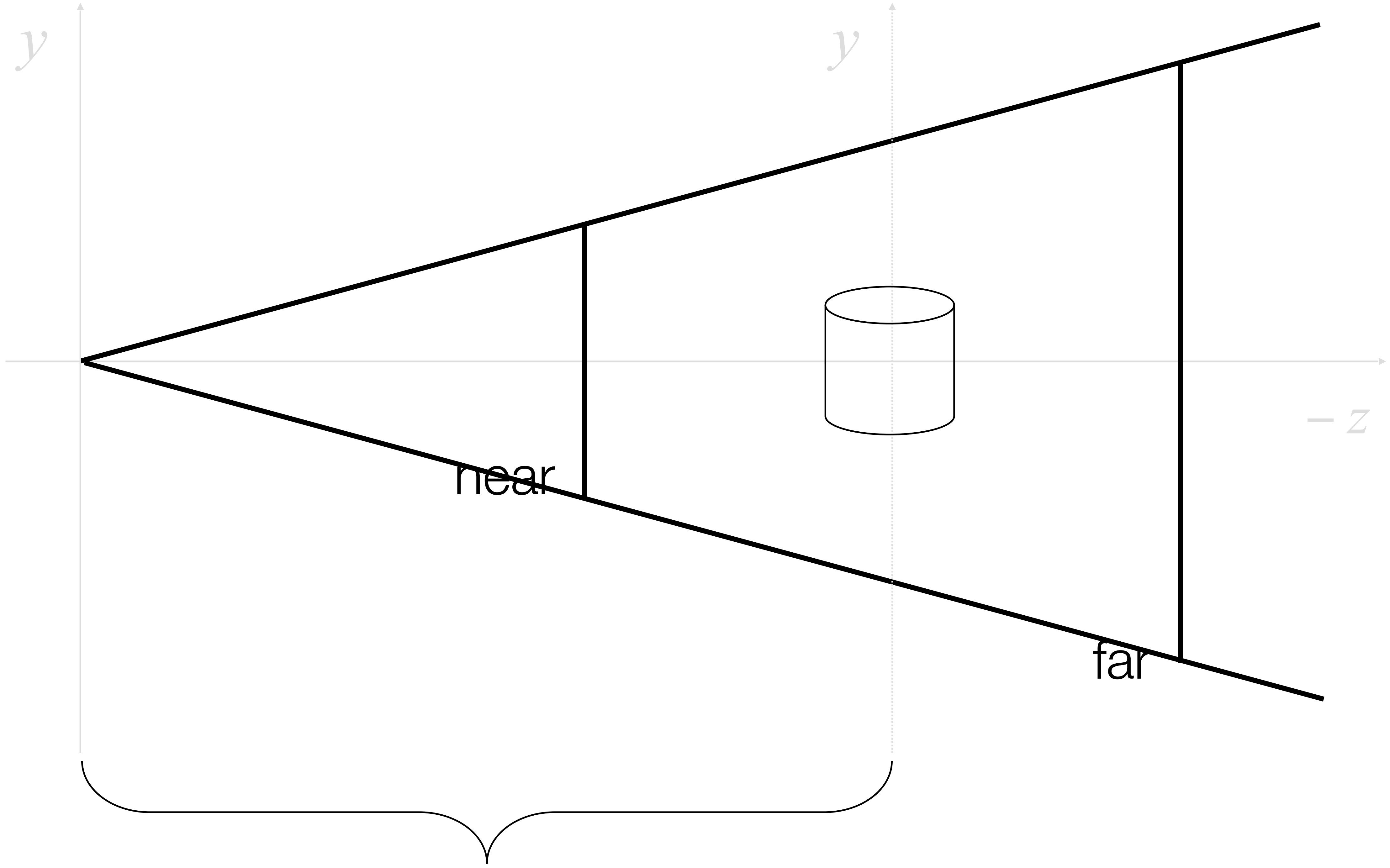
Viewing Transformations

Viewing Transformations

- Reorient world coordinates to match eye coordinates
- Basically just a modeling transform
  - affects the entire scene
  - usually a translation and a rotation
- Usually set up after the projection transform, but before any modeling transforms

The Simplest Viewing Transform

- “Push” the origin into the viewing frustum



$$z = -\frac{1}{2}(near + far)$$

Transforming World to Eye Coordinates

- Viewing transform

```
vec3 eye, look, up;
// assign values for eye, look, ...
var m = lookAt(eye, look, up);
```

- Creates an *orthonormal basis*
  - a set of linearly independent vectors of unit length

Creating an Orthonormal Basis

$$\hat{n} = \frac{\overrightarrow{look} - \overrightarrow{eye}}{\left\| \overrightarrow{look} - \overrightarrow{eye} \right\|}$$

$$\hat{u} = \frac{\hat{n} \times \overrightarrow{up}}{\left\| \hat{n} \times \overrightarrow{up} \right\|} \quad \Rightarrow \begin{pmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ -n_x & -n_y & -n_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\hat{v} = \hat{u} \times \hat{n}$$

- To complete `lookAt()`, we need a final translation to the eye position

Multiplying Matrices in JavaScript (using MV.js)

- This process creates a lot of matrices
    - you’ll use them in your shader (or perhaps even your application)
- Recall, order of matrices for multiplication is important
- Always **multiply on the right**

```
v = vec4(...);
S = scale(...);
T = translate(...);
V = lookAt(...);
MV = mul(mul(V, T), S);
P = perspective(...);

pos = mul(mul(P, MV), v);
```

Multiplying Matrices in a Shader

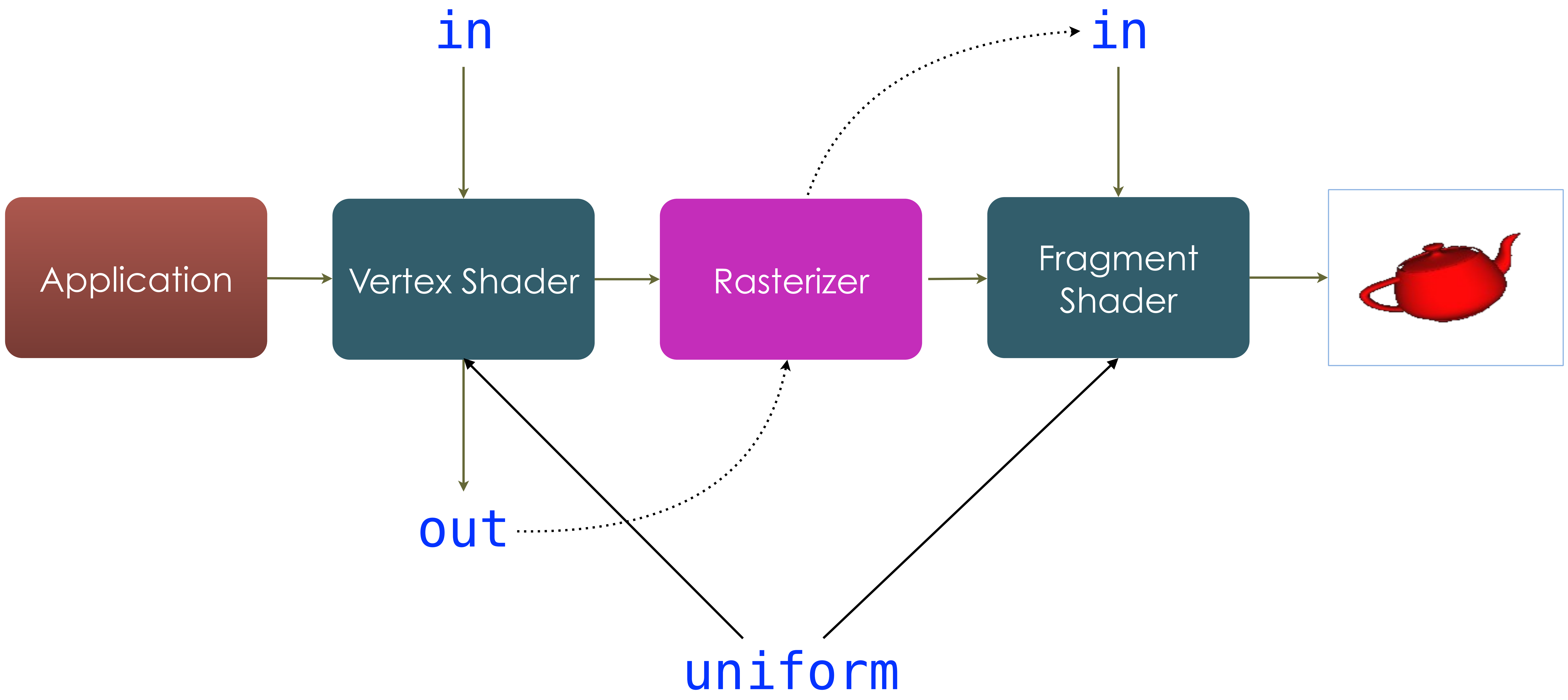
- GLSL makes that much cleaner
- Again, build up transforming the vertex from left-to-right
  1. projection transform
  2. viewing transform
  3. modeling transforms
  4. vertex position

```
in  vec4 aPosition;
in  vec4 aColor;
out vec4 vColor;

// Magic we'll discuss momentarily

void main()
{
  vColor = aColor;
  gl_Position = P * MV * aPosition;
}
```

Always **multiply on the right**

Uniform Variables

Graphics Pipeline



Uniform Variables

- Shared between vertex and fragment shaders in the same shader program
- Uniforms are like "constants" inside of a shader
  - their value doesn't change until the application updates it
- declared as a `uniform`

```
in  vec4 aPosition;
in  vec4 aColor;
out vec4 vColor;

uniform float t;

void main()
{
  vColor = aColor;
  gl_Position = t * aPosition;
}
```

Transformations are (usually) Uniforms

- Use uniforms for sending transformation matrices into shaders

```
in  vec4 aPosition;
in  vec4 aColor;
out vec4 vColor;

uniform mat4 MV;
uniform mat4 P;

void main()
{
  vColor = aColor;
  gl_Position = P * MV * aPosition;
}
```

Managing Uniforms

- Since we're encapsulating our models as JavaScript objects, we can create a useful interface for managing our uniforms
- Create a uniforms property to hold all the uniform locations used in a shader
  - those values you get back from `gl.getUniformLocation()`

```
function Cylinder( gl, ... ) {

  this.positions = { ... };
  this.colors = { ... };

  this.program = initShaders( ... );

  this.uniforms = {
    MV : gl.getUniformLocation(this.program, "MV"),
    P : gl.getUniformLocation(this.program, "P")
  };

  this.render = function () { ... };
}
```

Managing Uniforms

- It can be helpful to have an interface for the application to set the uniform's values
- Create some top-level properties to hold the uniforms values
- In your application, you can set their values:

```
Cylinder.P = perspective( ... );
Cylinder.MV = mult( ... );
```

```
function Cylinder( gl, ... ) {

  this.positions = { ... };
  this.colors = { ... };

  this.program = initShaders( ... );

  this.uniforms = {
    MV : gl.getUniformLocation(this.program, "MV"),
    P : gl.getUniformLocation(this.program, "P")
  };

  this.P = mat4();
  this.MV = mat4();

  this.render = function () { ... };
}
```

Drawing with Uniforms

- In the object's `render()` function
  - set the uniform's values using `gl.uniformMatrix4fv()`
  - Be care with which variable is which
    - `this.uniforms.MV` - uniform *location* from the shader program
    - `this.MV` - matrix's *value* set by your JavaScript application
- the `false` parameter indicates if the matrix should be transposed
  - it will always be false

```
function Cylinder( gl, ... ) {

  this.positions = { ... };
  this.colors = { ... };

  this.program = initShaders( ... );

  this.uniforms = {
    MV : gl.getUniformLocation(this.program, "MV"),
    P : gl.getUniformLocation(this.program, "P")
  };

  this.P = mat4();
  this.MV = mat4();

  this.render = function () {
    gl.uniformMatrix4fv(this.uniforms.MV, false,
      flatten(this.MV));
    gl.uniformMatrix4fv(this.uniforms.P, false,
      flatten(this.P);
  };
}
```

`flatten()`

- Helper function in **MV.js** that converts JavaScript arrays into `Float32Arrays`
- All of the **MV.js** types are JavaScript arrays or objects
- They all need to be `flatten()`ed before being passed into a WebGL function