CptS 442/542 (Computer Graphics)
Unit 11: Transforms (and Viewing)

Bob Lewis

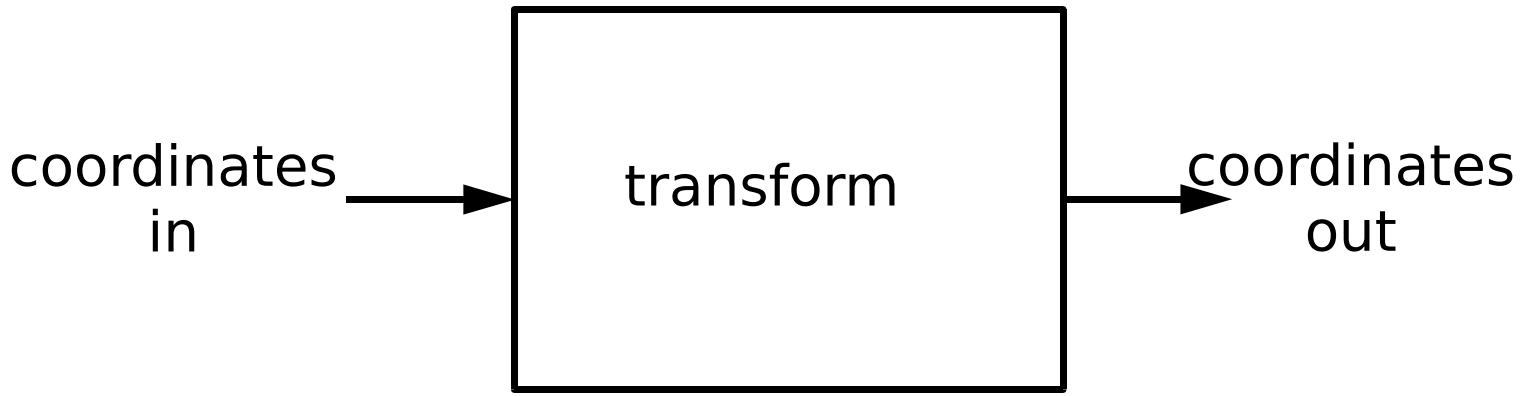
School of Engineering and Applied Sciences
Washington State University

Fall, 2017

Motivation

- In this unit, we will emphasize the mathematical nature of transforms and their use in solving geometric problems that arise in graphics.
- You'll see how *coaster* uses them when we review `pa08_first_person`.
- As you'll see, transforms are more generally useful in graphics than what OpenGL uses them for (modelling and viewing).

What's a Transform?



A transform (call it $\mathbf{T}$) is a special case of an operator, which is represented mathematically as:

$$\tilde{\mathbf{p}}' = \mathbf{T}\tilde{\mathbf{p}}$$

Note: This does not *necessarily* imply multiplication.

Aside: Linear and Affine Transforms

- linear transform:

$$\tilde{\mathbf{p}}' = \mathbf{T}\tilde{\mathbf{p}}$$

  where $\mathbf{T}$ is a matrix that multiplies $\tilde{\mathbf{p}}$. (Recall how these work from your linear algebra course.)

- affine transform:

$$\tilde{\mathbf{p}}' = \mathbf{T}\tilde{\mathbf{p}} + \vec{\mathbf{D}}$$

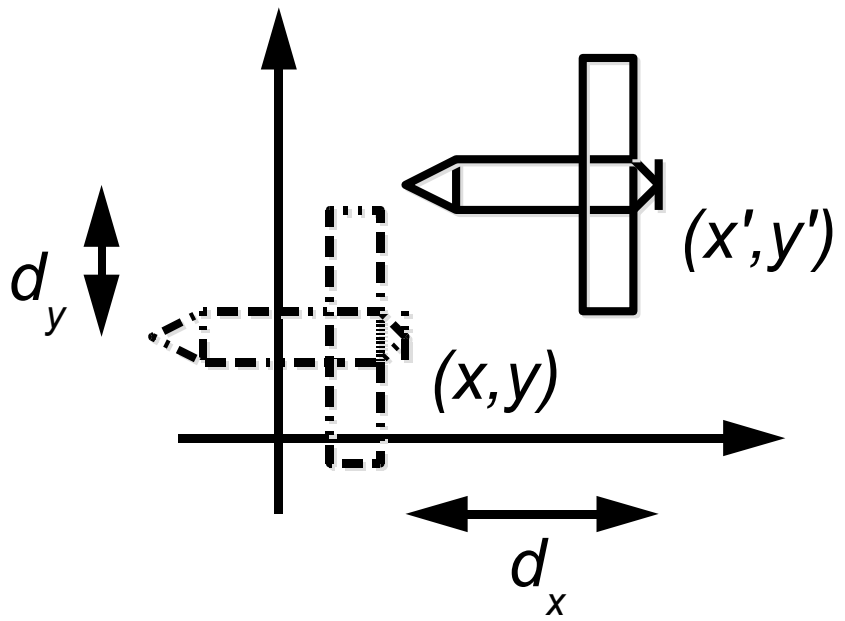
  where $\mathbf{T}$ is (still) a matrix and $\vec{\mathbf{D}}$ is a vector.
  These are more useful in graphics than linear transforms.

Note: At no time do the coordinates of $\tilde{\mathbf{P}}$ appear in $\mathbf{T}$ or $\vec{\mathbf{D}}$.

Aside: Homogeneous Coordinates

- Given N dimensions, add one more, traditionally called w:
  - points have $w = 1$
  - vectors have $w = 0$
- mathematical convenience (affine transforms look linear)
- computational overhead (in principal) ...
  - ... but none in practice with hardware (i.e. GPU) support
- HC's let us treat affine transforms as higher-dimensional linear transforms, so our linear algebra rules still work.
- borrowed from "Computational Geometry"
  - branch of math used in (e. g.) geographical application and robot motion planning
  - uses HC's extensively.
  - slight mind boggle: In CG, vectors are "points at infinity".

Translation



Displace every point by a constant vector $\vec{\mathbf{d}}$:

$$\begin{aligned} x' &= x + d_x \\ y' &= y + d_y \end{aligned}$$

How would we invert (i.e. undo) this transform?

Affine Translation Matrices

- given

$$\tilde{\mathbf{P}} = \begin{bmatrix} x \\ y \end{bmatrix} \text{(2D) or } \begin{bmatrix} x \\ y \\ z \end{bmatrix} \text{(3D)}$$

- 2D

$$\tilde{\mathbf{P}}' = \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tilde{\mathbf{P}} + \begin{bmatrix} d_x \\ d_y \end{bmatrix}$$

- 3D

$$\tilde{\mathbf{P}}' = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tilde{\mathbf{P}} + \begin{bmatrix} d_x \\ d_y \\ d_z \end{bmatrix}$$

Homogeneous Translation Matrices

- given

$$\tilde{\mathbf{P}} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \text{(2D) or } \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \text{(3D)}$$

- 2D

$$\tilde{\mathbf{P}}' = \begin{bmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{bmatrix} \tilde{\mathbf{P}}$$

- 3D

$$\tilde{\mathbf{P}}' = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tilde{\mathbf{P}}$$

- Note: Again, the coordinates of $\tilde{\mathbf{P}}$ don't appear in the matrix.
- What's the inverse?

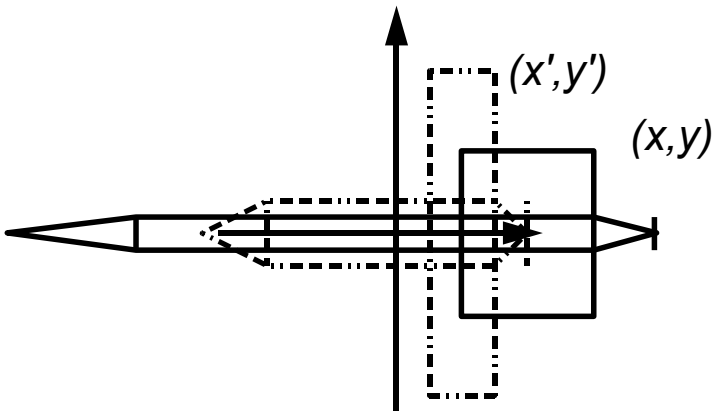
Transforming Vectors

We've only talked about transforming points so far. So what happens when we translate a vector $\vec{\mathbf{V}}$?

$$\vec{\mathbf{V}}' = \begin{bmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{bmatrix} \vec{\mathbf{V}} = \begin{bmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} V_x \\ V_y \\ 0 \end{bmatrix} = \begin{bmatrix} \\ \\ \end{bmatrix}$$

- What's the result? (It's the same for 3D.)
- Does it describe the effect of translation on a vector?
- See why homogeneous coordinates are handy? The same matrix transforms both points and vectors.
- Note: The rules are different for normals. (See below.)

Scaling



Scale every point by a factors of s_x and s_y around the origin.

$$\begin{aligned} x' &= s_x x \\ y' &= s_y y \end{aligned}$$

We can also do reflections with this. (How?)

Scaling Matrices

2D affine

$$\tilde{\mathbf{P}}' = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \tilde{\mathbf{P}}$$

2D homogeneous

$$\tilde{\mathbf{P}}' = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \tilde{\mathbf{P}}$$

3D affine

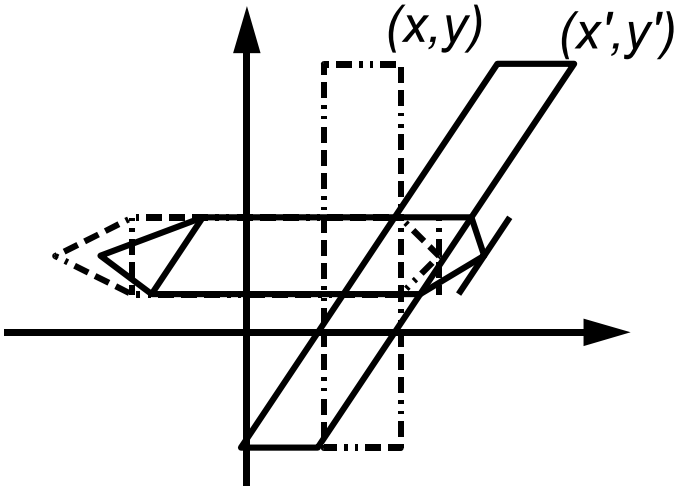
$$\tilde{\mathbf{P}}' = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} \tilde{\mathbf{P}}$$

3D homogeneous

$$\tilde{\mathbf{P}}' = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tilde{\mathbf{P}}$$

- Warning: It is almost always an error when a scale factor is zero.
- What's the inverse?
- If we replace point $\tilde{\mathbf{P}}$ with a vector $\vec{\mathbf{V}}$, does it describe the effect of scaling on a vector?

Shearing



(Shearing in the x direction:)

$$\begin{aligned} x' &= x + ay \\ y' &= y \end{aligned}$$

Shearing Matrices

2D affine

$$\tilde{\mathbf{P}}' = \begin{bmatrix} 1 & a \\ 0 & 1 \end{bmatrix} \tilde{\mathbf{P}}$$

2D homogeneous

$$\tilde{\mathbf{P}}' = \begin{bmatrix} 1 & a & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tilde{\mathbf{P}}$$

3D affine

$$\tilde{\mathbf{P}}' = \begin{bmatrix} 1 & a & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tilde{\mathbf{P}}$$

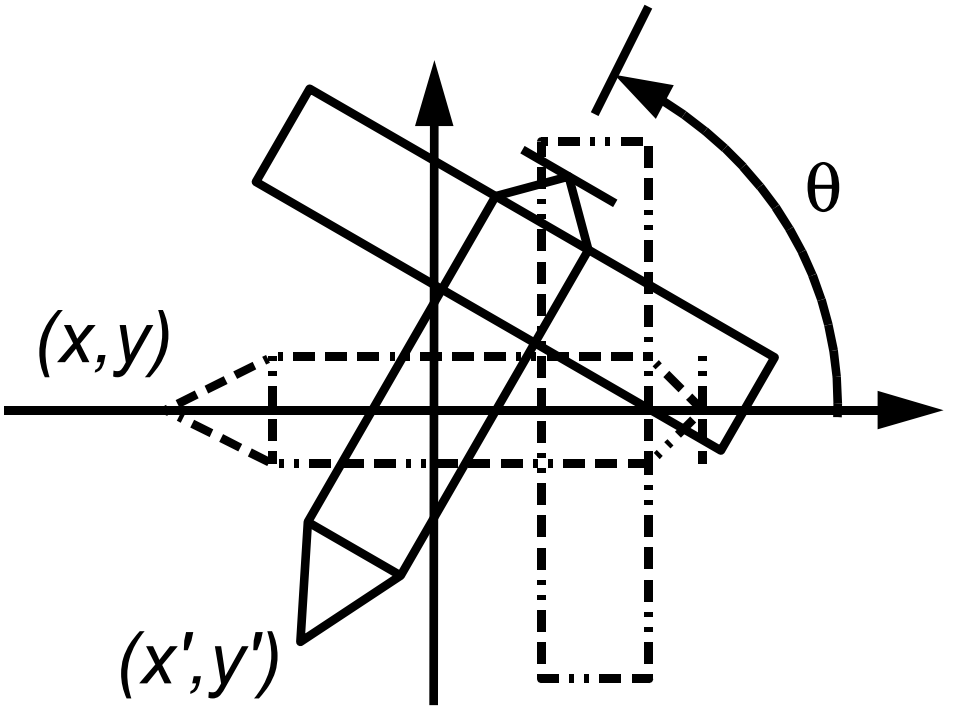
3D homogeneous

$$\tilde{\mathbf{P}}' = \begin{bmatrix} 1 & a & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tilde{\mathbf{P}}$$

- What's the inverse?
- If we replace point $\tilde{\mathbf{P}}$ with a vector $\vec{\mathbf{V}}$, does it describe the effect of shearing on a vector? (Actually, no, because shearing vectors is kind of meaningless. We usually only shear points.)

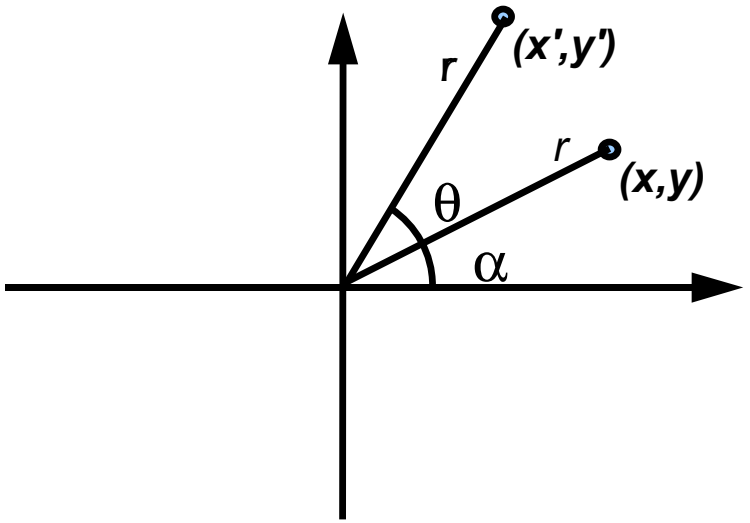
Rotation

Here's what we want to do:



Rotate every point by an angle θ around the origin.

Derivation of Rotational Transform



We start off with x and y in polar form:

$$\begin{aligned} x &= r\cos\alpha \\ y &= r\sin\alpha \end{aligned}$$

and then derive:

$$\begin{aligned} x' &= r\cos(\alpha+\theta) \\ &= r\cos\alpha\cos\theta + r\sin\alpha\sin\theta \\ &= x\cos\theta - y\sin\theta \\ y' &= r\sin(\alpha+\theta) \\ &= r\cos\alpha\sin\theta + r\sin\alpha\cos\theta \\ &= x\sin\theta + y\cos\theta \end{aligned}$$

2D Rotational Matrices

2D affine

$$\tilde{\mathbf{P}}' = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \tilde{\mathbf{P}}$$

2D homogeneous

$$\tilde{\mathbf{P}}' = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tilde{\mathbf{P}}$$

- What's the inverse?
- If we replace point $\tilde{\mathbf{P}}$ with a vector $\vec{\mathbf{V}}$, does it describe the effect of rotation on a vector? (Remember that points are rotated about their origin. Vectors are rotated "in-place".)

3D Rotational Matrices I

3D affine (by θ around $\hat{\mathbf{x}}$)

$$\tilde{\mathbf{P}}' = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \tilde{\mathbf{P}}$$

3D affine (by θ around $\hat{\mathbf{y}}$)

$$\tilde{\mathbf{P}}' = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \tilde{\mathbf{P}}$$

3D affine (by θ around $\hat{\mathbf{z}}$)

$$\tilde{\mathbf{P}}' = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tilde{\mathbf{P}}$$

3D homogeneous (by θ around $\hat{\mathbf{x}}$)

$$\tilde{\mathbf{P}}' = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tilde{\mathbf{P}}$$

3D homogeneous (by θ around $\hat{\mathbf{y}}$)

$$\tilde{\mathbf{P}}' = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tilde{\mathbf{P}}$$

3D homogeneous (by θ around $\hat{\mathbf{z}}$)

$$\tilde{\mathbf{P}}' = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tilde{\mathbf{P}}$$

3D Rotational Matrices II

- What are the inverses?
- If we replace point $\tilde{\mathbf{P}}$ with a vector $\vec{\mathbf{V}}$, does it describe the effect of rotation on a vector?
- You can combine these, but remember that order makes a difference, in general.
- There's a general matrix that will do 3D rotation around any axis $\widehat{\mathbf{A}}$, not just $\widehat{\mathbf{x}}$, $\widehat{\mathbf{y}}$, or $\widehat{\mathbf{z}}$. Look it up if you need it.

Transforming Normals

- another slight mind boggle: Normal vectors aren't really vectors.
- (sketch example of why not)
- They're "pseudovectors".
- given tangent plane:

$$\hat{\mathbf{N}} \cdot \tilde{\mathbf{P}} + b = \hat{\mathbf{N}}^t \tilde{\mathbf{P}} + b = 0$$

- and $\tilde{\mathbf{P}}$ gets transformed by $\mathbf{T}$:

$$\tilde{\mathbf{P}}' = \mathbf{T}\tilde{\mathbf{P}}$$

- How can we transform $\hat{\mathbf{N}}$ so that the tangent equation

$$\left(\hat{\mathbf{N}}'\right)^t \tilde{\mathbf{P}}' + b = 0$$

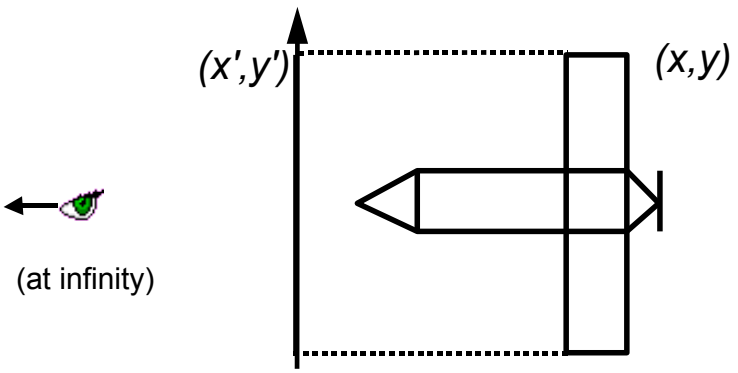
still holds?

This is why we pass `normalMatrix` to compute lighting.

Projections

- a subclass of transformations
- reduce the dimensionality of the projected object
- cannot be inverted

Flat (Orthographic) Projection



Transform (2D) is trivial:

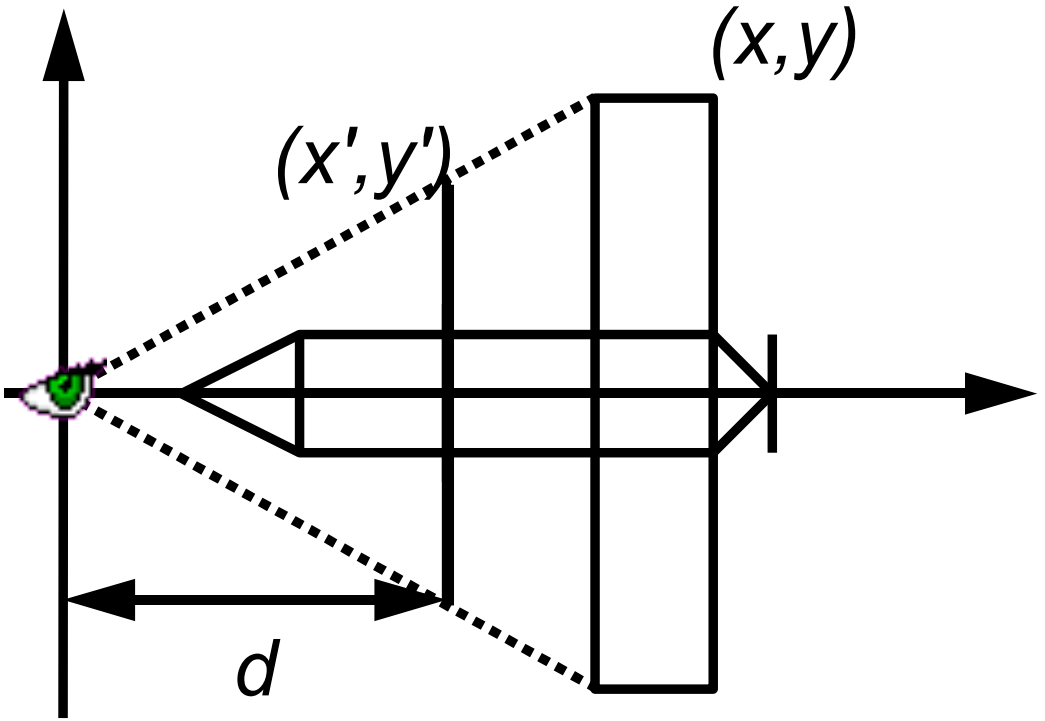
$$\begin{aligned} x' &= d \\ y' &= y \end{aligned}$$

and so is the 3D transform.

Thought exercise: What are the transform matrices?

Perspective Projection (2D)

We choose x to be the *axis of projection*.



Transform is:

$$\begin{aligned} x' &= d \\ y' &= \frac{dy}{x} \end{aligned}$$

but how can we write this in matrix form? It isn't linear and it isn't affine.

Perspective Projection Matrices

affine

(not possible, in 2D or 3D)

2D homogeneous

$$\tilde{\mathbf{P}}' = \begin{bmatrix} w'x' \\ w'y' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \frac{1}{d} & 0 & 0 \end{bmatrix} \tilde{\mathbf{P}}$$

3D homogeneous (x is still the axis of projection)

$$\tilde{\mathbf{P}}' = \begin{bmatrix} w'x' \\ w'y' \\ w'z' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \frac{1}{d} & 0 & 0 & 0 \end{bmatrix} \tilde{\mathbf{P}}$$

See the trick?

Notes on Perspective Projection

- We can use this to project 3D objects onto a 2D surface (projector screen, paper, T-shirt, etc.).
- "Perspective division" goes beyond what you learned in (most) linear algebra courses.
- It was first discovered by artists (see below), not mathematicians. (Better answer: by artists who were also mathematicians.)
- related to computational geometry
- What's the inverse? (Trick question.)
- Do *not* confuse perspective projection with the (similar) perspective 3D viewing *transform* we develop later:
  - not a projection
  - (and therefore) has an inverse
- Perspective projection does have an application in 3D graphics: shadows.

Compositing Transforms

Suppose we want to perform *several* operations on an object.
Matrix multiplication is associative:

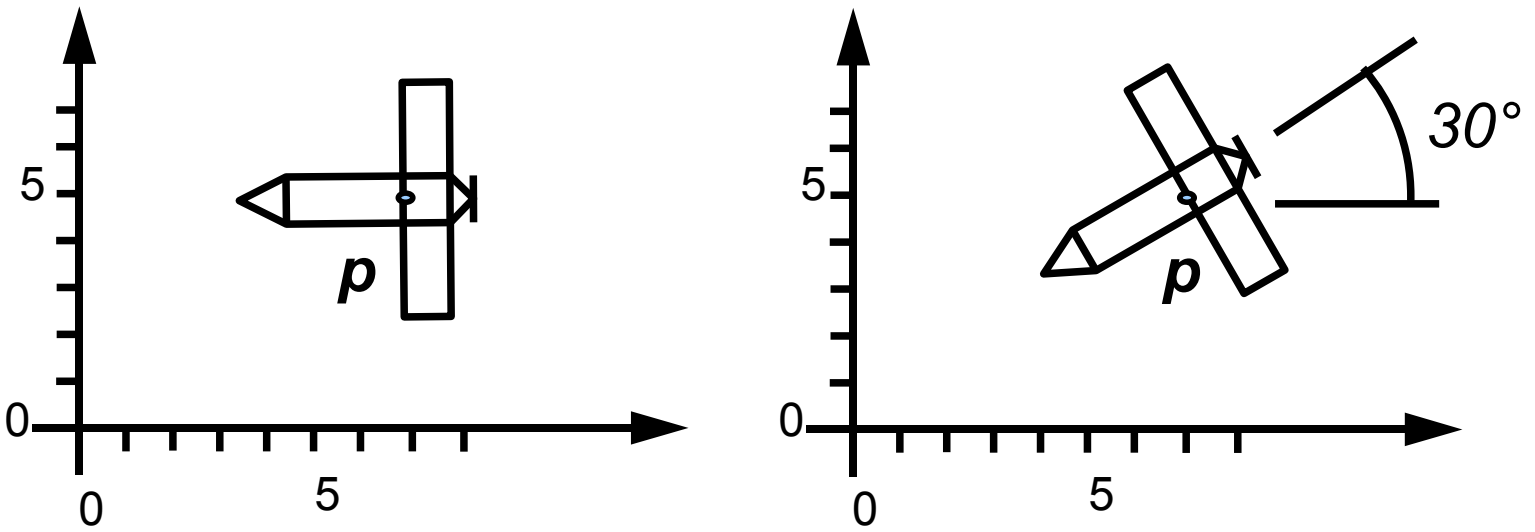
$$\mathbf{T_a}(\mathbf{T_b}\tilde{\mathbf{P}}) \equiv (\mathbf{T_a}\mathbf{T_b})\tilde{\mathbf{P}}$$

So (if we use homogeneous matrices), we can combine all transforms into a *single* matrix multiply.
To find $\mathbf{M}$ such that $\tilde{\mathbf{P}}' = \mathbf{M}\tilde{\mathbf{P}}$ for all $\tilde{\mathbf{P}}$ in the model:

1. Find a series of N steps (translations, scales, shears, rotations, projections, etc.) that do what you want.
2. For each step i, construct $\mathbf{T}_i$.
3. $\mathbf{M} = \mathbf{T}_{N-1}\mathbf{T}_{N-2}...\mathbf{T}_1\mathbf{T}_0$

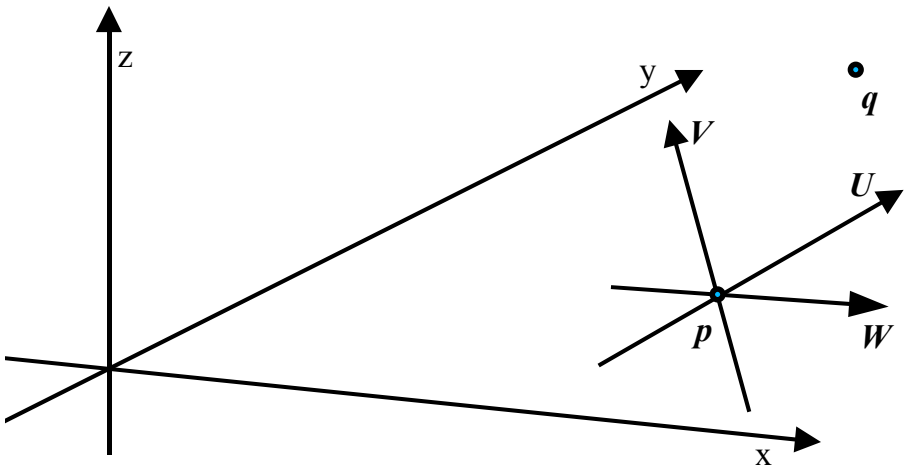
Compositing Transform Example: Rotation about a Point

To rotate around an arbitrary point $\tilde{\mathbf{P}}$:



"From Scratch" Transform: Change of Coordinate System

Suppose we have a coordinate frame whose origin is $\tilde{\mathbf{p}}$ and whose unit vectors are given as $\vec{\mathbf{U}}$, $\vec{\mathbf{V}}$, and $\vec{\mathbf{W}}$. Let $\mathbf{T}$ transform **UVW** to **xyz** coordinates.



- $\mathbf{T}\begin{bmatrix}1\\0\\0\\0\end{bmatrix} = \vec{\mathbf{U}}$
- $\mathbf{T}\begin{bmatrix}0\\1\\0\\0\end{bmatrix} = \vec{\mathbf{V}}$
- $\mathbf{T}\begin{bmatrix}0\\0\\1\\0\end{bmatrix} = \vec{\mathbf{W}}$
- $\mathbf{T}\begin{bmatrix}0\\0\\0\\1\end{bmatrix} = \tilde{\mathbf{p}}$

Can we combine these equations into one that will allow us to solve for $\mathbf{T}$?

Insight: Derive $\mathbf{T}$ "from scratch" using what we know about it.

"From Scratch" Transform: Change of Coordinate System

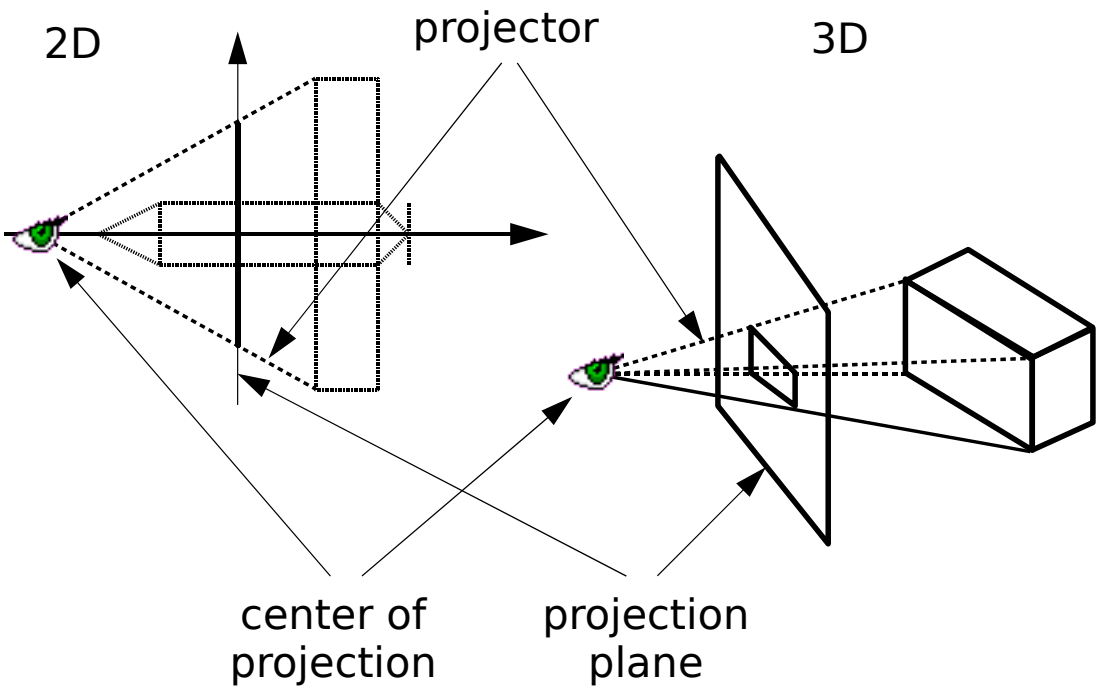
- $\mathbf{T}\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \vec{\mathbf{U}}$
- $\mathbf{T}\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \vec{\mathbf{V}}$
- $\mathbf{T}\begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \vec{\mathbf{W}}$
- $\mathbf{T}\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \tilde{\mathbf{p}}$

So what if we wanted to go the other way, from **xyz** to **UVW** coordinates?

A Drafting "Perspective" on Projections

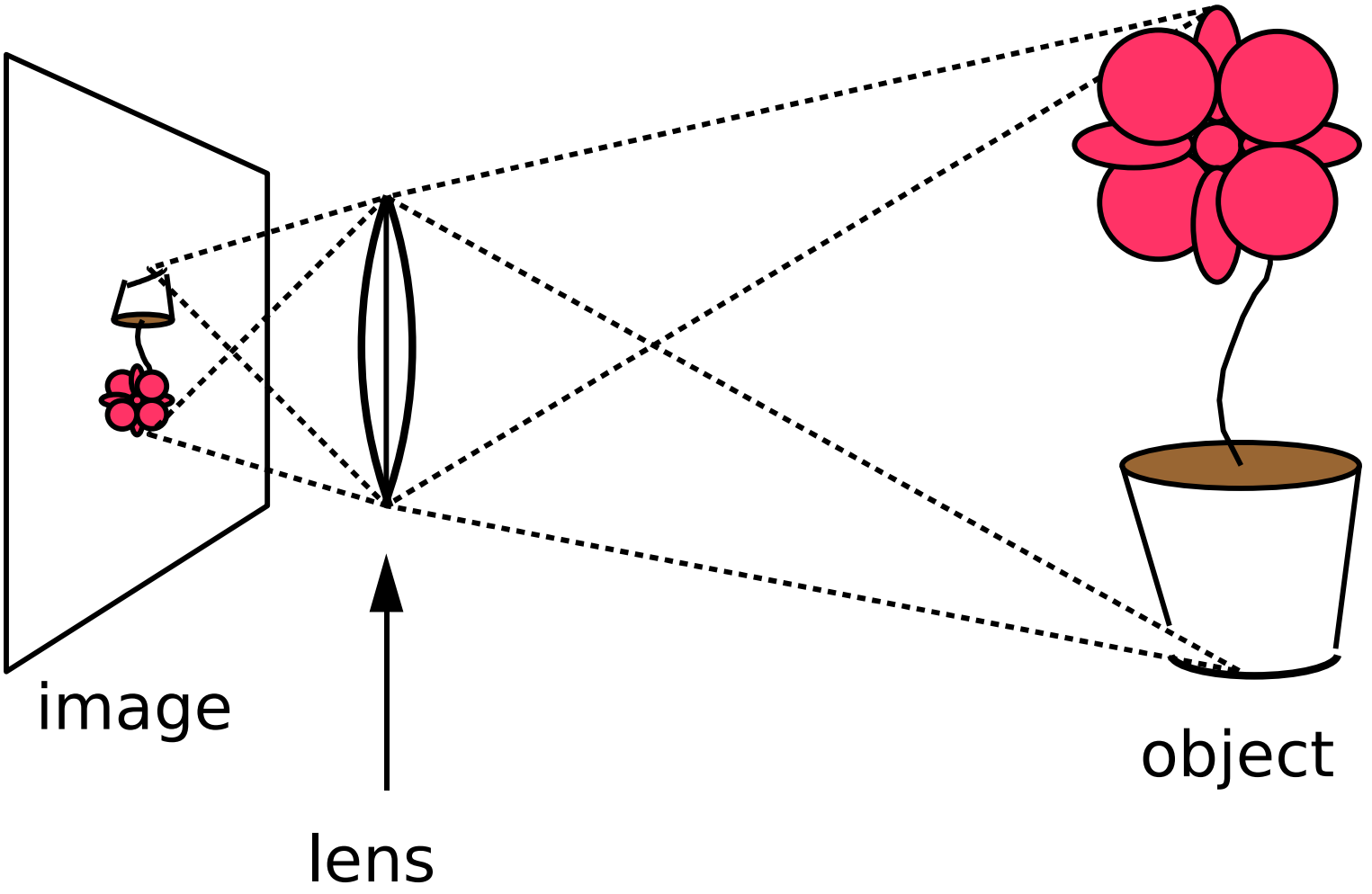
- Extensive knowledge base, long predating computer graphics: How do users *want* to look at drawings of objects?
- How to present three (or more) dimensions on a two-dimensional display?
- Two basic classes:
  - Perspective
  - Parallel
- These are planar geometric projections. There are lots of others. (Think dome projectors.)

Perspective Projection (Review)

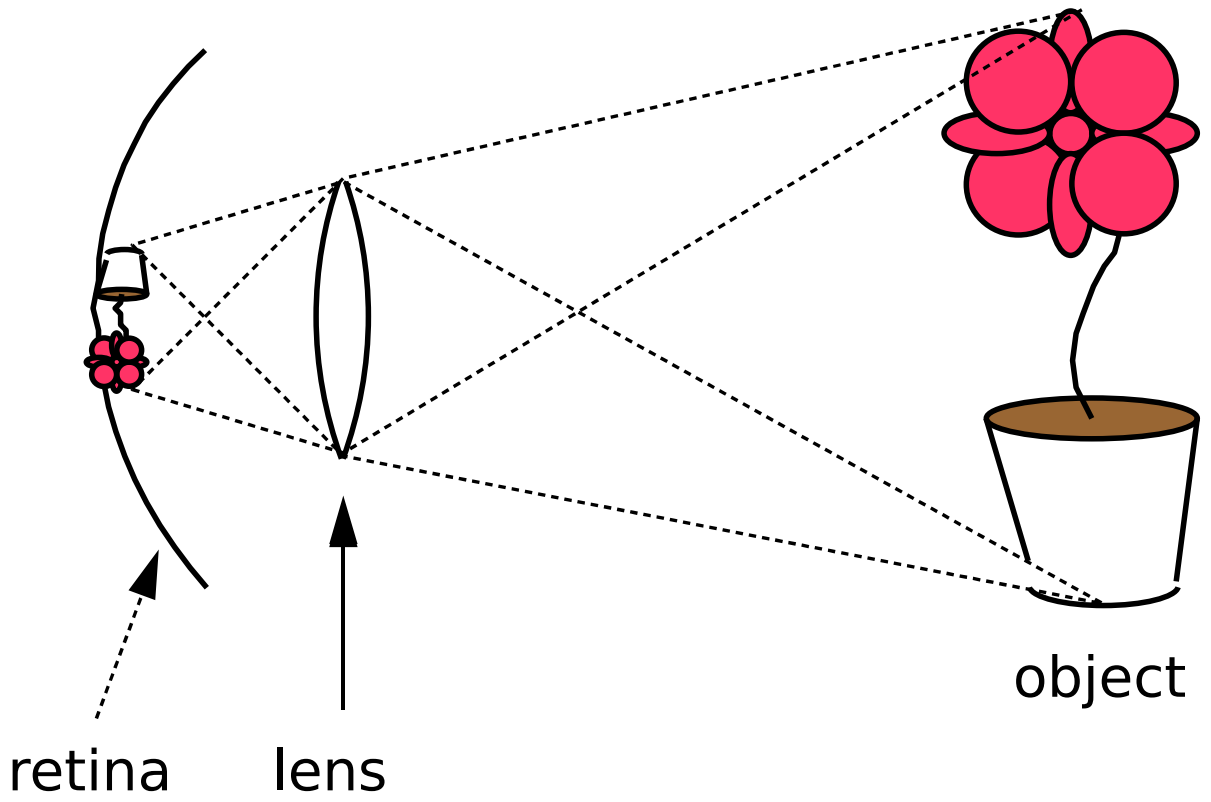


But what has it got to do with the way we see things?
How do the center-of-projection and projection plane relate to how human vision works?

Image Formation (Optics)

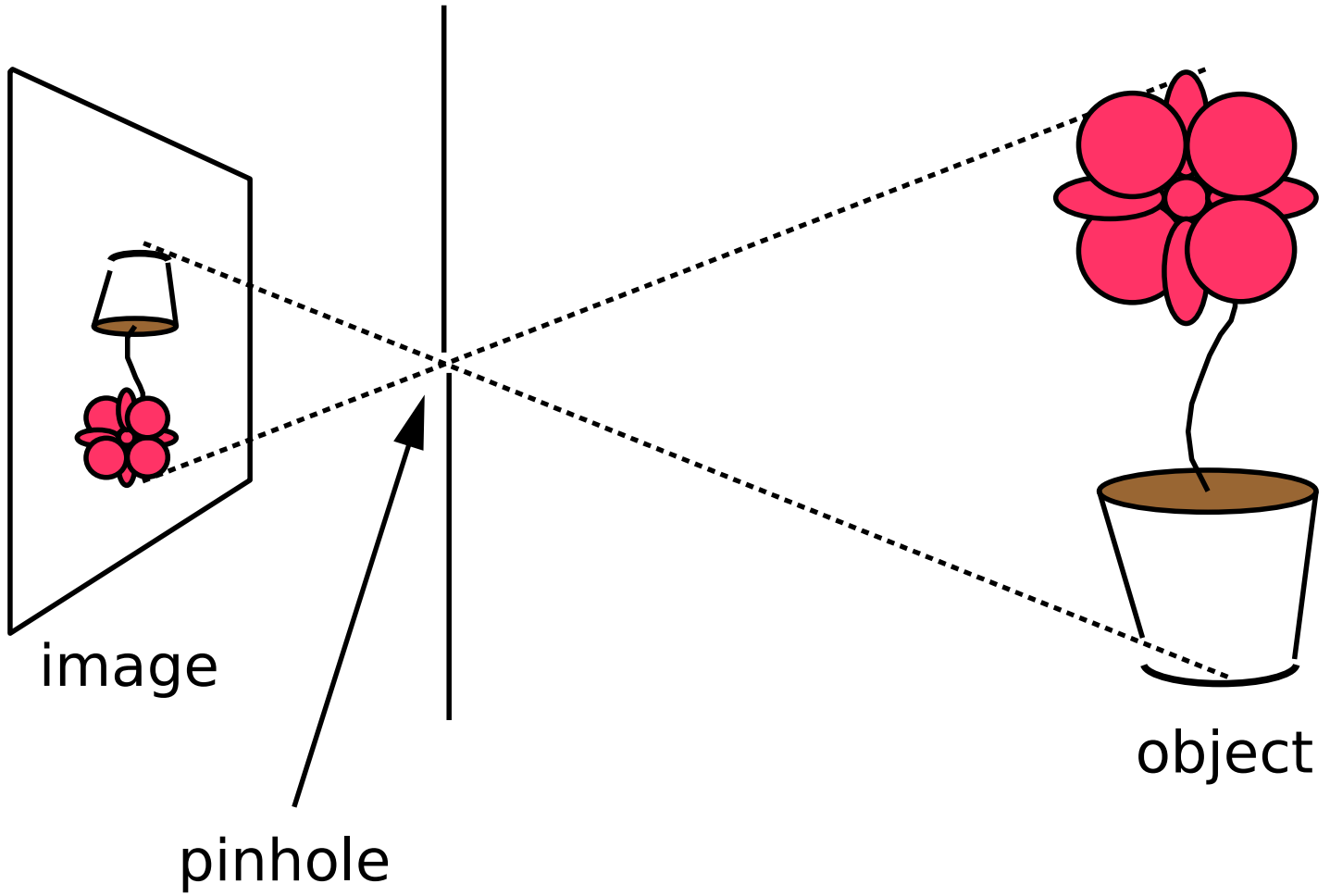


The Human Viewing System (greatly simplified)



So how does this relate to perspective projection?

The Pinhole Camera



A Perspective on Perspective

Albrecht Dürer is credited with the first use of correct perspective in art, circa 1500:

Earlier artists generally got perspective wrong.

Dürer's *St. Jerome in His Study*

One of the things he discovered was the "vanishing point," a characteristic of all perspective drawing.

The Discovery of the Vanishing Point (David Macaulay)

From his *Great Moments in Architecture* (recommended):

(Macaulay is just kidding!)

Parallel Projections

parallel lines stay parallel

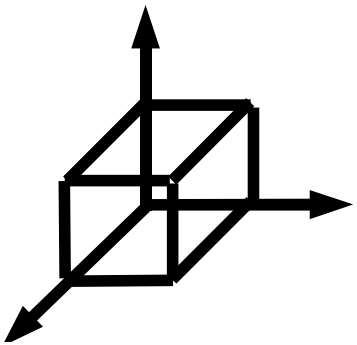
- Orthographic
  - Top ("plan")
  - Front
  - Side
- Axonometric: planes not normal to a principal axis (Do any of these look familiar?)
  - Isometric: all axes project to equal length
  - Dimetric: only two axes project to equal length
  - Trimetric: no two axes project to equal length

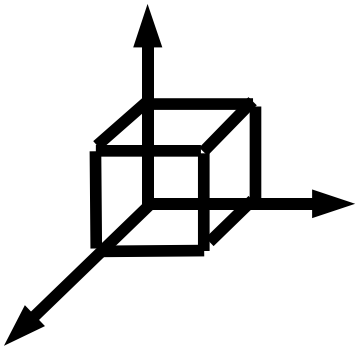
Parallel Projections: Oblique

Oblique $\equiv$ projection plane normal differs from projection direction

Cabinet: all edges project to unit length

Cavalier: edges parallel to x and y project to unit length

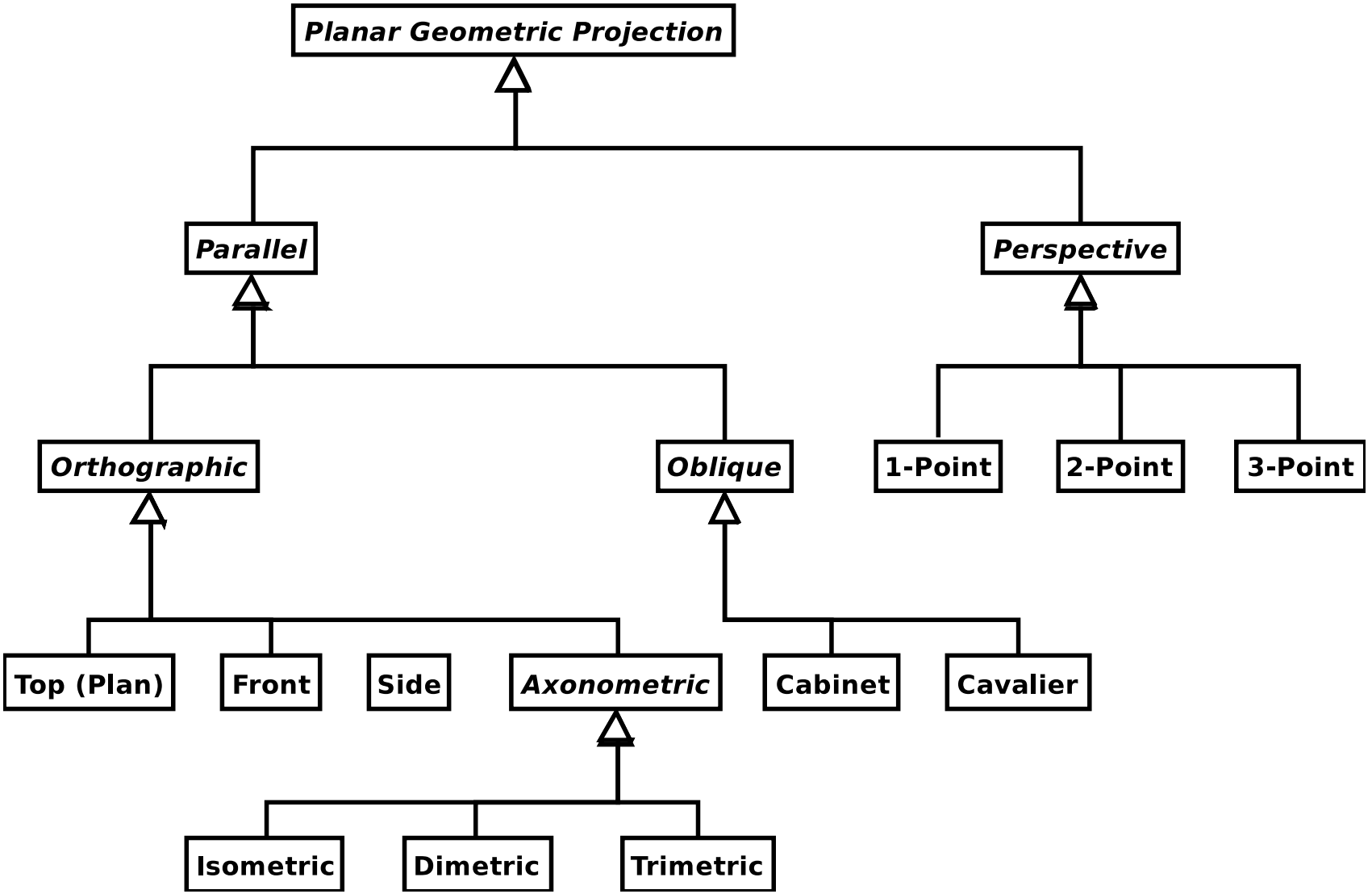




There are others.

How do these relate to perspective projections?

Summary: Planar Projections as a UML Class Diagram



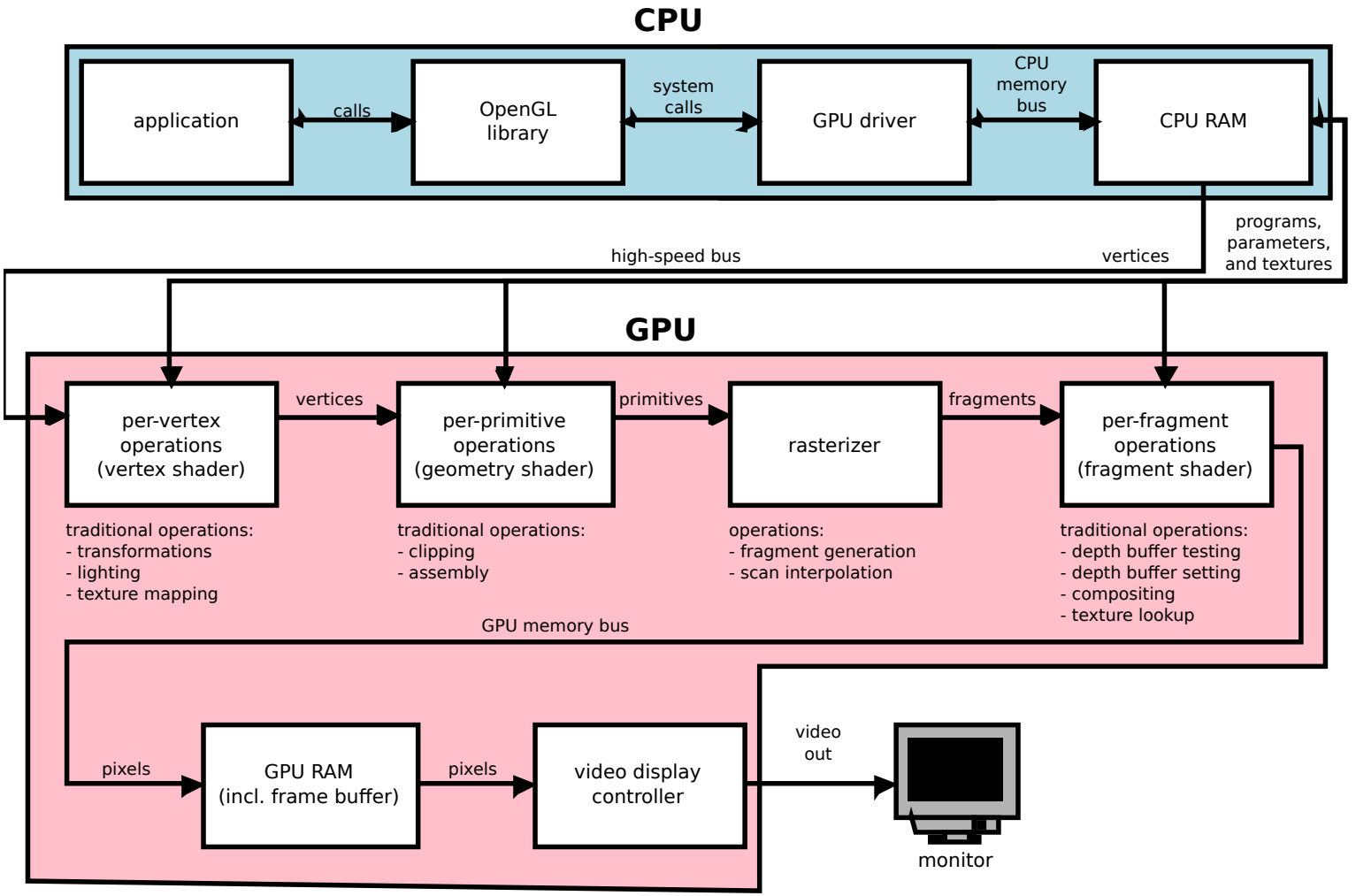
One Way to Specify a View

in world coordinates:

- $\vec{\mathbf{E}}$ view reference ("VRP") or "eye" point, center of viewing plane, origin of view-reference coordinate (VRC) system
- $\widehat{\mathbf{N}}$ view plane normal ("VPN"), normal to the view ($u - v$ projection) plane
- $\widehat{\mathbf{V}}$ upward vector, projection on view plane is v axis
- $\widehat{\mathbf{U}}$ perpendicular to $\widehat{\mathbf{N}}$ and $\widehat{\mathbf{V}}$, can be constructed automatically (how?)

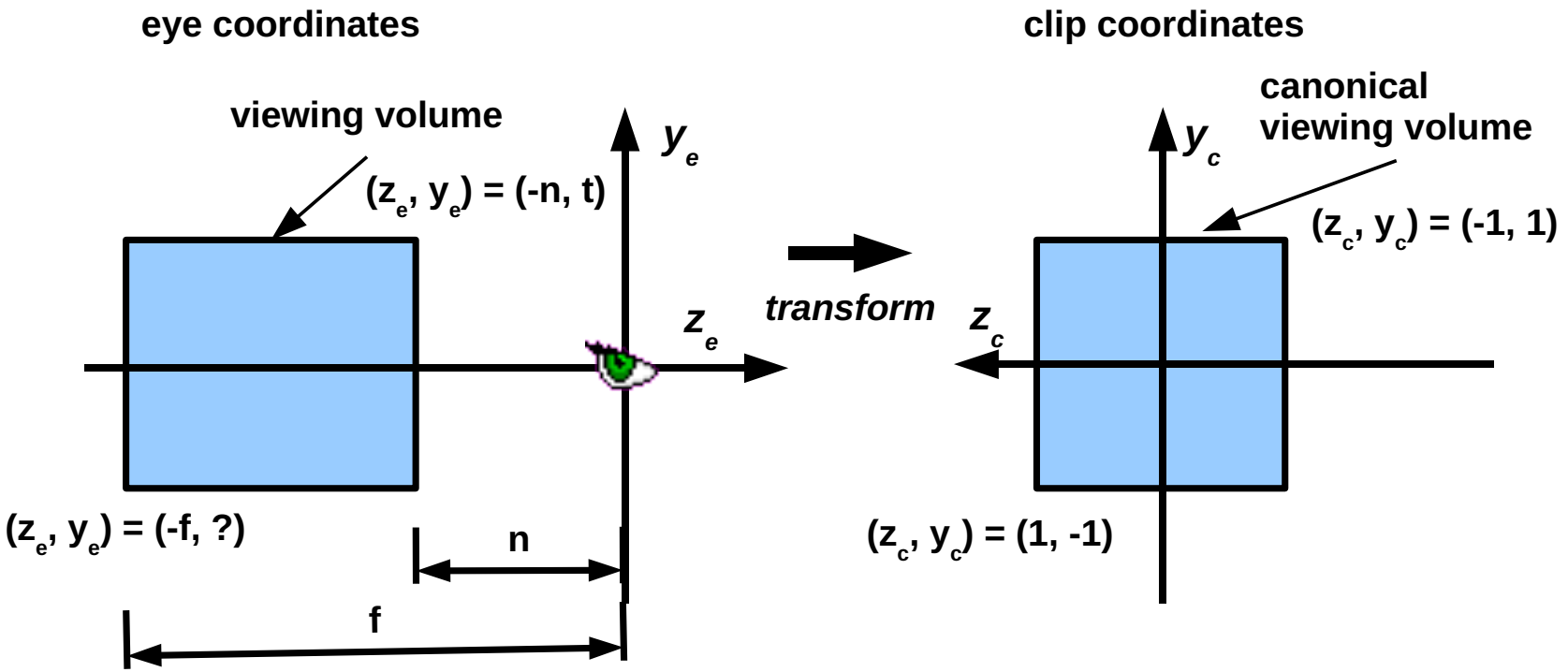
These form the camera's "viewing" transform we use for *coaster*.

Where Does OpenGL Transform Coordinates?



OpenGL's Orthographic ("Projection") Transform I

"Projection" is really a misnomer. Don't confuse it with the viewing transform.



Key: l: left, r: right, t: top, b: bottom, n: near, f: far

OpenGL's Orthographic ("Projection") Transform II

- n (*near*) and f (*far*) are distances measured from the eye. Both are almost always positive.
- maps "rectangular parallelepiped" (i.e. box) in eye coordinates to $2 \times 2 \times 2$ "canonical viewing volume" (CVV) centered on the origin. We have referred to coordinates in this volume as "normalized device coordinates" (NDC).
- The $x_e \rightarrow x_c$ transform is just like that of $y_e \rightarrow y_c$, except that it uses (*left*, *right*) instead of (*bottom*, *top*).
- What's going to happen to the z (hint: depth) value during fragment processing? (note change in its direction)
- What *has* happened to the z value?
- How should we pick *near* and *far*?

OpenGL's Orthographic ("Projection") Transform III

$$\begin{bmatrix} w_c x_c \\ w_c y_c \\ w_c z_c \\ w_c \end{bmatrix} = \mathbf{M}_{\text{ortho}} \begin{bmatrix} w_e x_e \\ w_e y_e \\ w_e z_e \\ w_e \end{bmatrix}$$

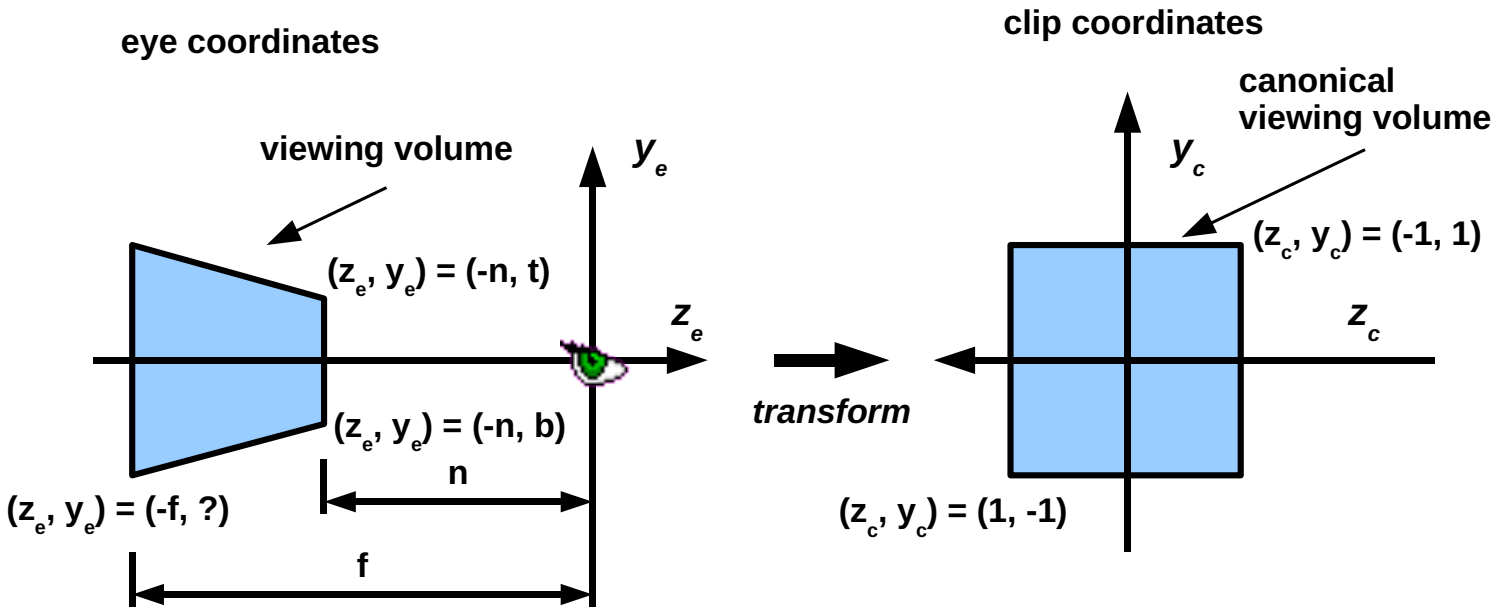
where

$$\mathbf{M}_{\text{ortho}} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & \frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & \frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & \frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

What can go wrong? (Hint: denominators)

So much for the orthographic view. Suppose we want a "first person" viewing experience?

OpenGL's Perspective ("Projection") Transform I



OpenGL's Perspective ("Projection") Transform II

Same as for the orthographic case:

- *near* and *far*
- the CVV
- the relation of $x_e \rightarrow x_c$ to $y_e \rightarrow y_c$
- What's going to happen to the z value?

Note:

- Why do we still need the z value? (This is why we have a transform and not a projection.)
- How should we pick *near* and *far*?

OpenGL's Perspective ("Projection") Transform III

$$\begin{bmatrix} w_c x_c \\ w_c y_c \\ w_c z_c \\ w_c \end{bmatrix} = \mathbf{M}_{\text{persp}} \begin{bmatrix} w_e x_e \\ w_e y_e \\ w_e z_e \\ w_e \end{bmatrix}$$

where

$$\mathbf{M}_{\text{persp}} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

What's (new) that can go wrong? (This is not so obvious unless we consider a large scene.)

Transforms in *coaster*: Basics

- We've been using transforms in *coaster* almost from the beginning.
- `Transform` in C++ becomes `mat4` in GLSL.
- Shaders always use a single 4x4 transform to transform vertices.
- In tradtional OpenGL, this is the "model-view-projection" transform. *coaster* is different.
- Transforms default to the identity transform.

Transforms in *coaster*: `pa01_circles`

- No transform set.
- Implicitly uses a (4×4) identity transform in GLSL.
- In OpenGL/GLSL you're always drawing in 3D (technically, projected 4D).
- z coordinate defaults to 0.
- w coordinate defaults to 1 (orthographic projection).

Transforms in *coaster*: `pa02_wire_track` and `pa03_wire_car`

- `viewTransform` set in `controller` module using UI (e. g., arrow keys)
- `viewTransform` passed to `render()` methods
- `render()` methods apply `viewTransform` to vertices (in CPU) and invoke `updateBuffers()` every time they're redrawn (ineffcient!)
- w coordinate defaults to 1 and w row of transform to $[0\,0\,0\,1]$ (orthographic projection).

Transforms in *coaster*: `pa04_hedgehog_car`

same as `pa03_wire_car`, except

- `viewTransform` passed to vertex shaders (as a GLSL `mat4`)
- vertex shader applies transform to input vertices
- `updateBuffers()` method only needs to be called once, ever (objects are static)

Transforms in *coaster*: `pa05_shaded_car` and `pa06_shaded_track`

same as `pa04_hedgehog_car`, except

- `normalTransform` is a 3×3 (not 4×4) matrix used to – wait for it – transform normals for lighting computation (see above)
- `normalTransform` is passed to vertex shader (as a uniform variable) as well as `viewTransform`

Transforms in *coaster*: `pa07_surfaces`

same as `pa06_shaded_track`, except

- every `SceneObject` now gets a `modelTransform`
- `SceneObjects` are defined in their own (model) coordinate system
- `modelTransform` can place object anywhere in "world" (i.e. "world coordinates")
- same `GeometricObject` can be placed multiple places with different `modelTransforms` (e. g. cars on track)
- `SceneObjects` can nest inside other `SceneObjects` by concatenating `modelTransforms` in `display()` method calls (e. g. rails, supports, and ties inside track)
- ultimately, `display()`s call `GeometricObject` `draw()` methods, which set (concatenated) `modelTransforms` to transform model coordinates into world coordinates (the world transform)

Transforms in *coaster*: `pa08_first_person`

same as `pa07_surfaces`, except

- car model transforms are derived from their coordinate frames, which change with "time" (frame number)
- view transform derived from first car transform
- `viewTransform` only depends on the camera position, not the projection.
- projection transform is orthographic or perspective, depending on UI
- `Scene::display()` passes
  - the view-projection transform: the product of the projection transform and the view transform, and
  - the initial world transform (an identity transform)

  to every top-level `SceneObject`'s `display()` method.
- The normal transform is computed (transpose-of-inverse) from the world transform.
- Call tree now embodies "scene graph".