Computer Graphics

Discussion 4

Garett Ridge garett@cs.ucla.edu,
Sam Amin samamin@ucla.edu,
Theresa Tong theresa.r.tong@gmail.com,
Quanjie Geng szmun.gengqj@gmail.com

Part I: Projection Matrices and View Planes

(Hit this HARD for midterm studying)

Transform Process (A quick review)

- The transform is always just one 4x4 matrix.
- But calculating what it should be involves multiplying out a big chain of intermediate special matrices. That chain is always:

$$\underbrace{\begin{bmatrix} ? & 0 & 0 & ? \\ 0 & ? & 0 & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{Viewport} * \underbrace{\begin{bmatrix} ? & 0 & ? & ? \\ 0 & ? & ? & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & ? & ? \end{bmatrix}}_{Projection} * \underbrace{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{Camera} * \underbrace{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{Model} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Transform Process

$$\underset{Viewport}{\begin{bmatrix} ? & 0 & 0 & ? \\ 0 & ? & 0 & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \underset{Projection}{\begin{bmatrix} ? & 0 & ? & ? \\ 0 & ? & ? & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & ? & ? \end{bmatrix}} * \underset{Camera}{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \underset{Model}{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- Note: We never actually see the viewport matrix.
  - The viewport matrix is automatically applied for you at the end of the vertex shader.
  - Early during initialization, javascript set it up, calling gl.viewport(x,y,width,height).
- All the other special matrices you do manage.

Transform Process

$$\underbrace{\begin{bmatrix} ? & 0 & 0 & ? \\ 0 & ? & 0 & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{Viewport} * \underbrace{\begin{bmatrix} ? & 0 & ? & ? \\ 0 & ? & ? & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & ? & ? \end{bmatrix}}_{Projection} * \underbrace{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{Camera} * \underbrace{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{Model} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- The camera matrix is very much like the model transform matrix for placing shapes. But:
  - The shape being placed is the scene’s observer
  - You actually use the **inverse matrix** of what you would have done to a 3D model of an actual camera

Transform Process

$$\underset{Viewport}{\begin{bmatrix} ? & 0 & 0 & ? \\ 0 & ? & 0 & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \underset{Projection}{\begin{bmatrix} ? & 0 & ? & ? \\ 0 & ? & ? & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & ? & ? \end{bmatrix}} * \underset{Camera}{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \underset{Model}{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- The projection matrix is something <u>you</u> make, using special calls.
- Two built in functions make two kinds of them:
  - perspective() causes converging lines / vanishing points.
  - ortho() causes parallel lines to remain parallel -- like how scenes look when viewed from far enough away.

Transform Process

$$\underset{Viewport}{\begin{bmatrix} ? & 0 & 0 & ? \\ 0 & ? & 0 & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \underset{Projection}{\begin{bmatrix} ? & 0 & ? & ? \\ 0 & ? & ? & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & ? & ? \end{bmatrix}} * \underset{Camera}{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \underset{Model}{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- The projection matrix is something you make, using special calls.
  - perspective(): The camera is like a point, and will see everything that falls within a truncated pyramid (frustum) expanding out from it

Transform Process

$$\underbrace{\begin{bmatrix} ? & 0 & 0 & ? \\ 0 & ? & 0 & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{Viewport} * \underbrace{\begin{bmatrix} ? & 0 & ? & ? \\ 0 & ? & ? & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & ? & ? \end{bmatrix}}_{Projection} * \underbrace{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{Camera} * \underbrace{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{Model} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- The projection matrix is something <u>you</u> make, using special calls.
  - ortho(): The camera is like a flat rectangle, and will see everything that falls within a rectangular box in front of it.
  - Rectangular boxes are a special case of frustums

Transform Process

$$\underbrace{\begin{bmatrix} ? & 0 & 0 & ? \\ 0 & ? & 0 & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{Viewport} * \underbrace{\begin{bmatrix} ? & 0 & ? & ? \\ 0 & ? & ? & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & ? & ? \end{bmatrix}}_{Projection} * \underbrace{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{Camera} * \underbrace{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{Model} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- Both types, perspective() and ortho(), are projections.
- Projections are different from the camera matrix.
  - Camera matrix: places, sizes, and points the virtual camera.
  - Projection matrix: shapes the virtual camera’s lens (really, the world “frustum” containing the view volume)

Projections

- So, there are two choices for how the view frustum is shaped: Perspective or Orthographic (parallel)
- The frustum has six planes, and the closest to the camera is called the “near plane”
- The projection matrix maps all 3D points that fall inside a frustum onto the near plane of that frustum, thereby reducing all shapes to 2D, for screen display.

Projections: Online Demos

- http://threejs.org/examples/#webgl_camera
  - Perspective vs orthographic - the difference between the two projection frustums (and what they see) -- press O and P to switch between the two.
  - Clipping planes
  - Tons of other informative examples are linked there, like the demo of flat vs. smooth shading: http://threejs.org/examples/#webgl_morphnormals

Projections

- We use a right handed system (x cross y = z)
- x and y in traditional plot directions make z go out of board, so
- We look down -z
  – Projection matrix is what actually accomplishes this, with a sign flip. Without it, we’re in a left-handed system!

Transform Process

$$\underbrace{\begin{bmatrix} ? & 0 & 0 & ? \\ 0 & ? & 0 & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{Viewport} * \underbrace{\begin{bmatrix} ? & 0 & ? & ? \\ 0 & ? & ? & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & ? & ? \end{bmatrix}}_{Projection} * \underbrace{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{Camera} * \underbrace{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{Model} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

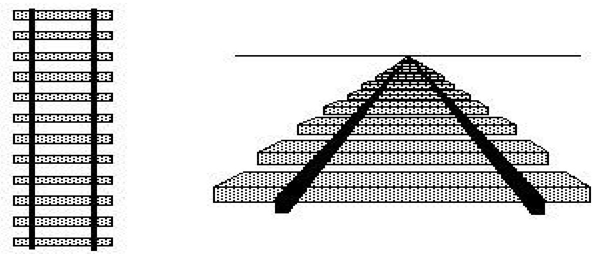
- One more invisible thing happens after our code in addition to the viewport matrix:
  <u>The Perspective Division</u>
  - (Different from Perspective Matrix)

Transform Process

$$\underset{Viewport}{\begin{bmatrix} ? & 0 & 0 & ? \\ 0 & ? & 0 & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \underset{Projection}{\begin{bmatrix} ? & 0 & ? & ? \\ 0 & ? & ? & ? \\ 0 & 0 & ? & ? \\ 0 & 0 & ? & ? \end{bmatrix}} * \underset{Camera}{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \underset{Model}{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & 0 & 1 \end{bmatrix}} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- To do it, divide final vector [x,y,z,w] by its own w
  - (Not necessarily 1 anymore after projection matrix)
  - Can pull x and y closer to zero as depth increases
- No matrix can do that “row division” effect
  - It’s not a linear operation

Perspective transforms

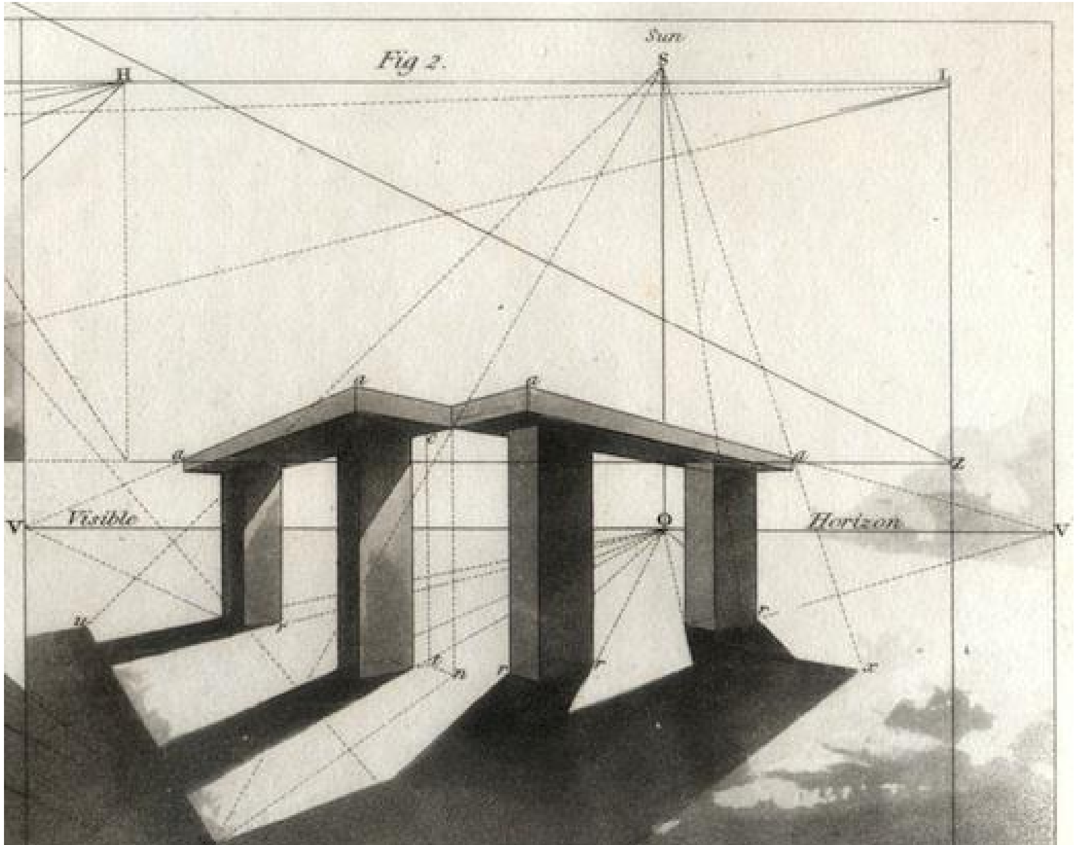


What happens to groups of parallel lines during this non-linear division effect? Why?

What happens to ratios along straight lines? (Count the tracks and see)

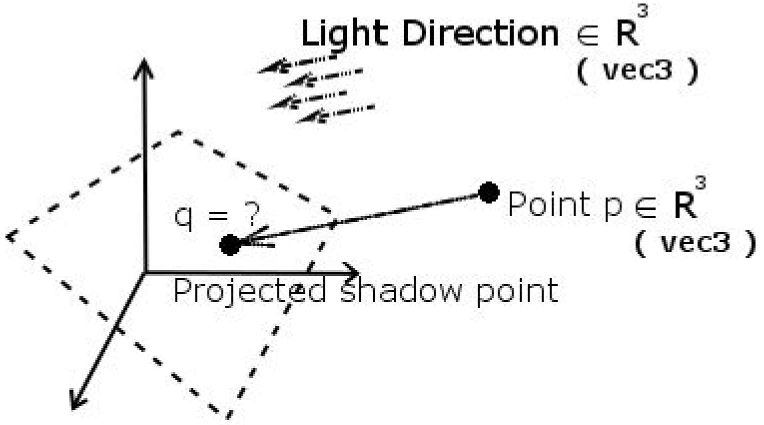
Another use for Perspective-style frustums: Shadow Mapping.

1. Render the scene an extra time to an unseen buffer, placing the camera at the light source.
2. Test any point by checking if it got obstructed from this vantage point.   If so, it's in shadow.



Projection of shadow point onto plane

- Problem from book
  - Our exam question on projection plane requires much simpler geometry



Projection of shadow point onto plane demo

- Think of a simpler 2D version first - project of a point onto line instead of plane
  - becomes 2D line-2D line intersection
- In 3D:
  - Becomes 3D line - 3D plane intersection
  - One linear equation in 3D only gives a plane
  - How to represent a 3D line then? Add more constraints, so more equations.

How do we intersect two lines?

- We could:
  - Express the linear system as a matrix, and solve
  - (Equivalently) Plug one explicit line equation into the other in place of y
  - Both of those only apply in 2D due to few equations involved
  - (Works in 3D too) Convert one line to parametric form, find parameter value that satisfies other line too

Graphics fields' names for the grade school forms of a line:

Explicit:

<u>The Slope-Intercept Form of the Equation of a Line</u>

The equation of a line with slope m and y-intercept (0, b) is given by,

$y = mx + b$

where m is the slope and (0, b) is the y-intercept

Note: Great form for performing transformations; our matrices are shorthand for it.

Implicit:

<u>The Standard Form of the Equation of a Line</u>

$Ax + By = C$

where A, B, and C are real numbers

Note: Great for testing which side of a line or surface you're on (simply change = to < or >).

Parametric: (We will derive from:)

<u>The Point-Slope Form of the Equation of a Line</u>

The equation of a line with slope m and passing through the point (x_1, y_1) is given by,

$y - y_1 = m (x - x_1)$

where m is the slope and (x_1, y_1) is the point given

Note: Great for representing locations along a line or surface, such as the solution of an intersection.

Derivation of parametric form

If the two point form is

$$(y - y_1) = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1)$$

We can write it as

$$\frac{y - y_1}{y_2 - y_1} = \frac{x - x_1}{x_2 - x_1}$$

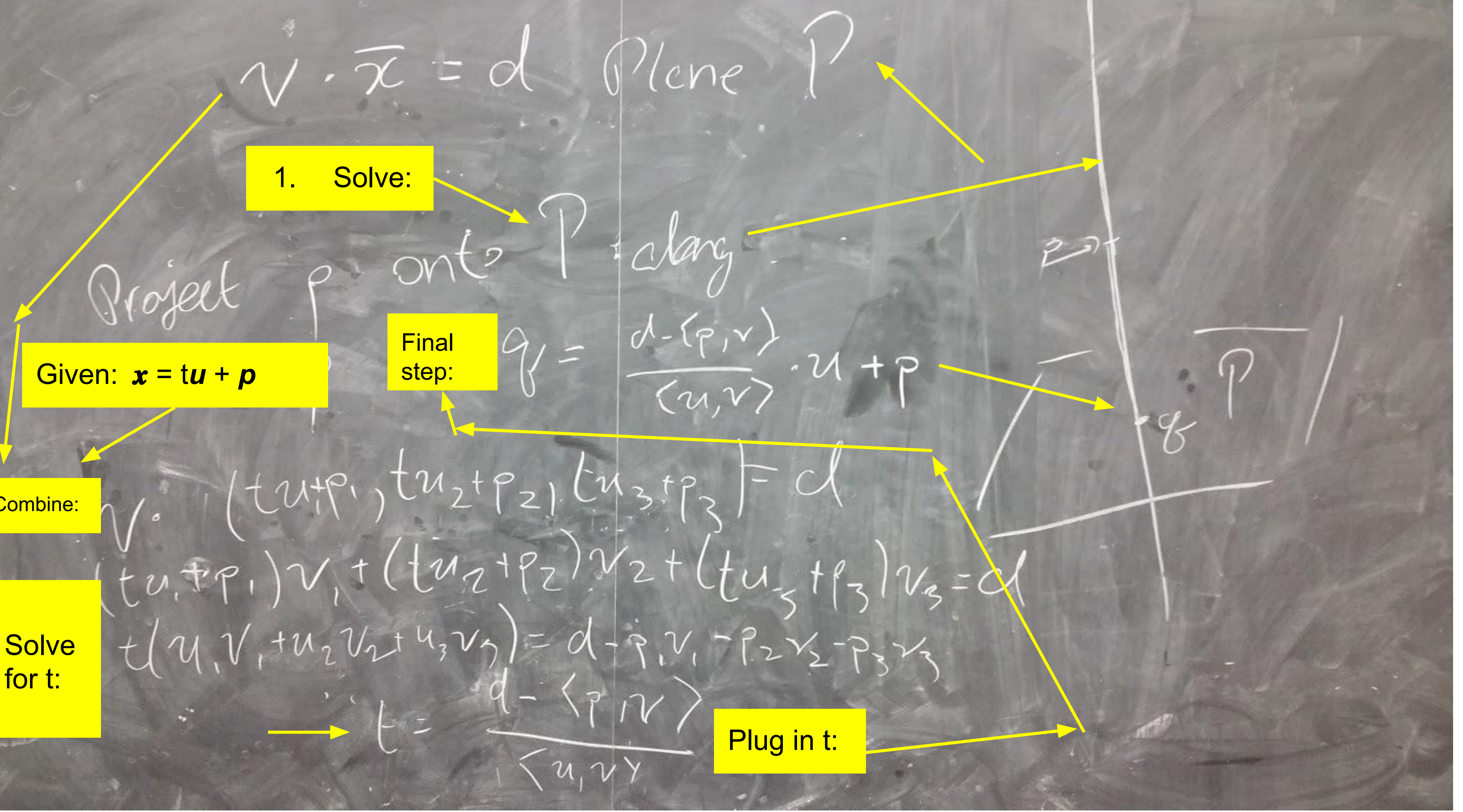
Since they are equal, we set them both equal to t:

$$y - y_1 = (y_2 - y_1)t \Rightarrow y = y_1 + (y_2 - y_1)t$$
$$x - x_1 = (x_2 - x_1)t \Rightarrow \underline{x = x_1 + (x_2 - x_1)t}$$

This is then the parametric form.

Solve for the projected point q.

Using vector algebra shorthand for the equations:



v · x̄ = d Plane P
1. Solve:
Project p onto P along u
Given: x = tu + p
Combine:
v · (tu1+p1, tu2+p2, tu3+p3) = d
(tu1+p1)v1 + (tu2+p2)v2 + (tu3+p3)v3 = d
Solve for t:
t(u1v1 + u2v2 + u3v3) = d - p1v1 - p2v2 - p3v3
t = (d - ⟨p, v⟩) / ⟨u, v⟩
Plug in t:
Final step:
q = (d - ⟨p, v⟩) / ⟨u, v⟩ · u + p
P
q

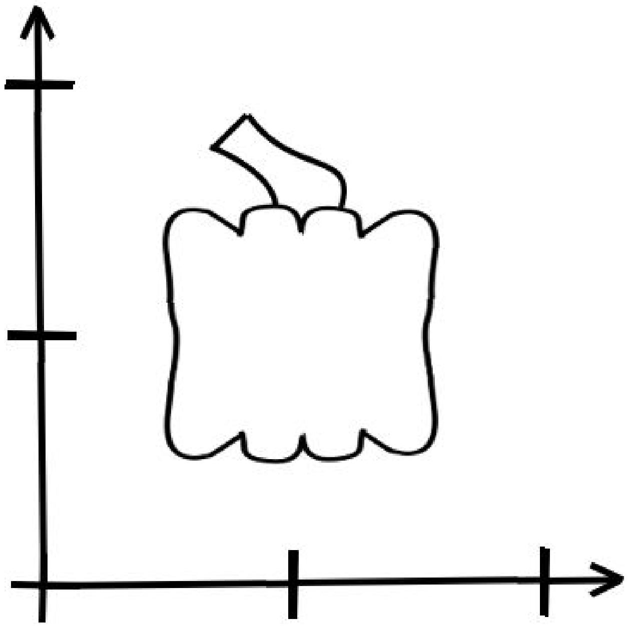
Part II: Transformations practice problem

Three Ways To Do Every Transformation Problem:

- 1. Intuition using moving bases (or axes)
  - Reading typical code forwards
  - Reading written product left-to right ending at p
  - Products formed via post-multiplication
- 2. Intuition using a moving point cloud (or shape)
  - Reading typical code backwards
  - Reading written product right-to-left starting at p
  - Products formed via pre-multiplication
- 3. Writing the product out, doing matrix multiplication by hand, and not relying on intuition at all

Drawing example: Pumpkin (Pretend it's fall quarter)

Given this pumpkin at (1,1),



Drawing example: Pumpkin

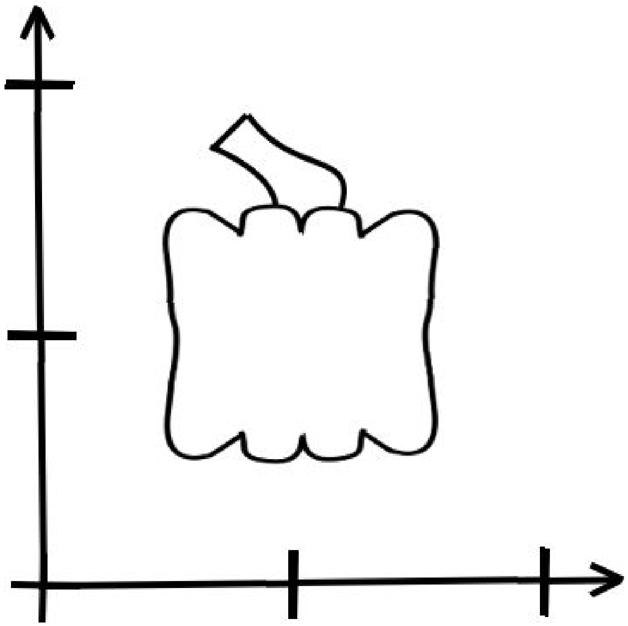
Given this pumpkin at (1,1), do the following:

model *= trans(x+2,y+2);

model *= rot_z(90);

model *= $scale_x$(-1);

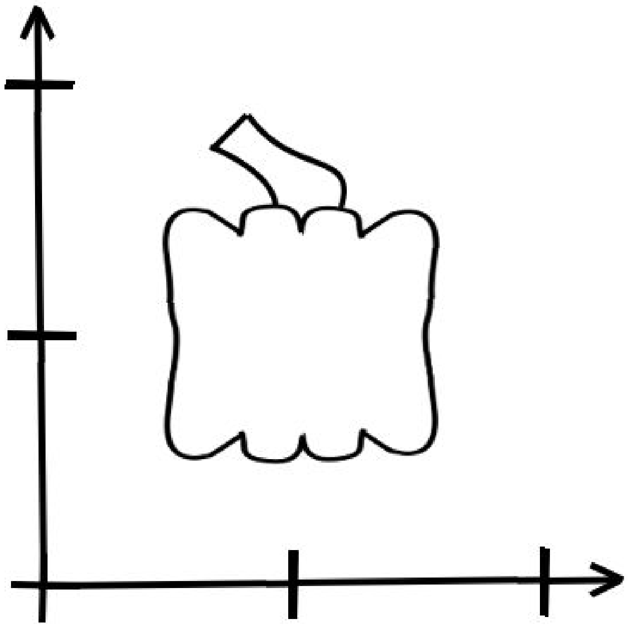
model *= trans(x-1,y-1);



Drawing example: Pumpkin

Given this pumpkin at (1,1), do the following:

trans(2,2) * rot_z(90) * $scale_x$(-1) * trans(-1,-1)

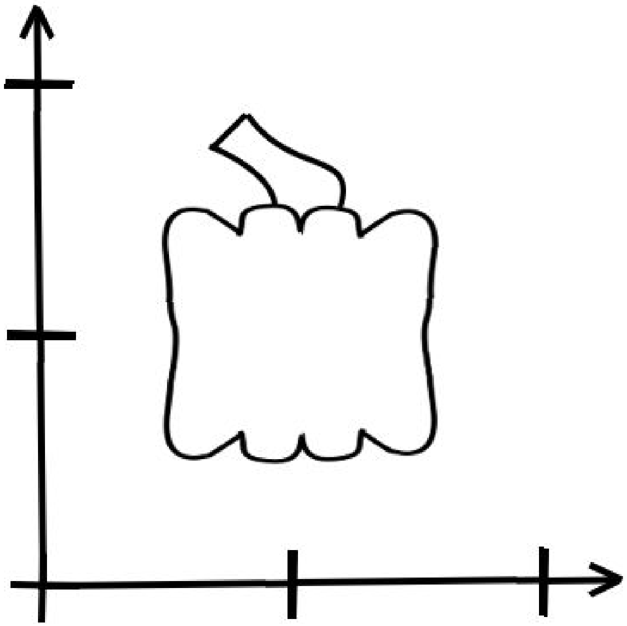


Drawing example: Pumpkin

Manually writing the product of matrices

$\text{trans}(2,2) * \text{rot}_z(90) * \text{scale}_x(-1) * \text{trans}(-1,-1)$

= what actual matrices?

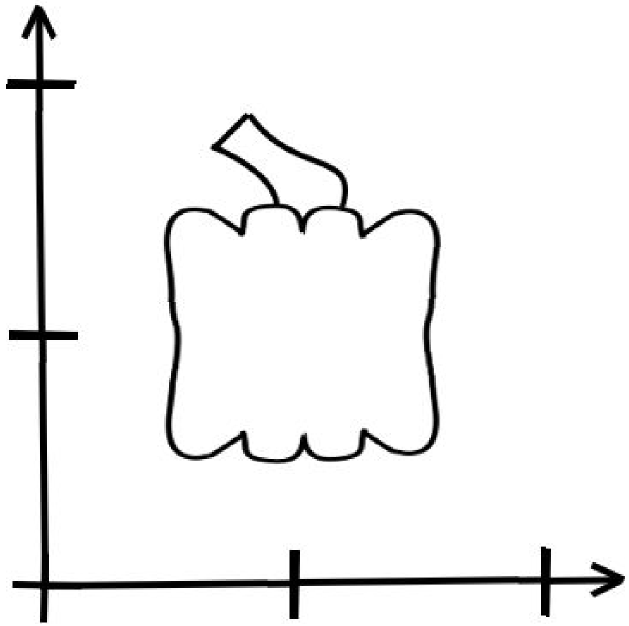


Drawing example: Pumpkin

- Manually writing the product of matrices

$\text{trans}(2,2) * \text{rot}_z(90) * \text{scale}_x(-1) * \text{trans}(-1,-1) = ?$

- Multiply out the product with the “drawing below” trick
- Apply the final product to some points (0,0), (0,2), (2,0)

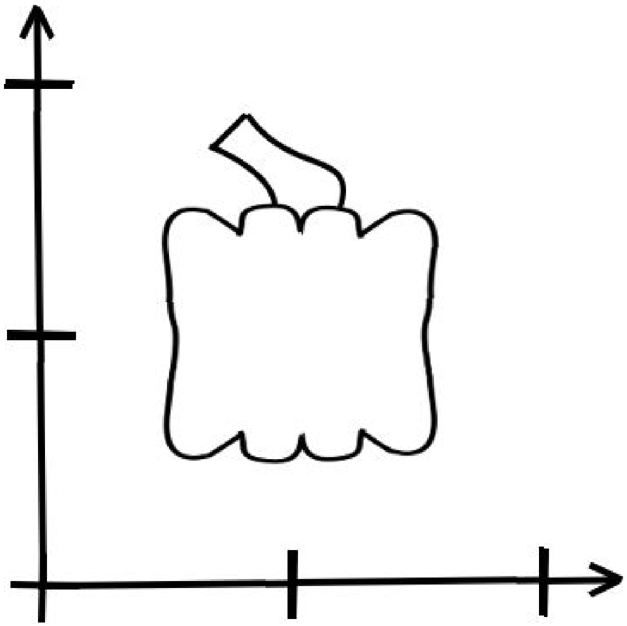


Drawing example: Pumpkin

- Actually draw out where the pumpkin moves at each step of

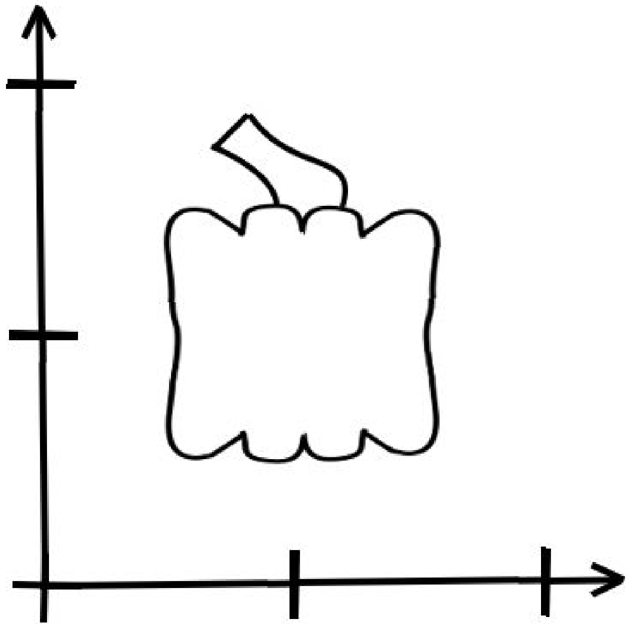
trans(2,2) * rot_z(90) * $scale_x$(-1) * trans(-1,-1)

- We’re treating it like an image -> Start at point and move Right-to-Left
- Show that where it landed is consistent with where the product displaced the 3 points to



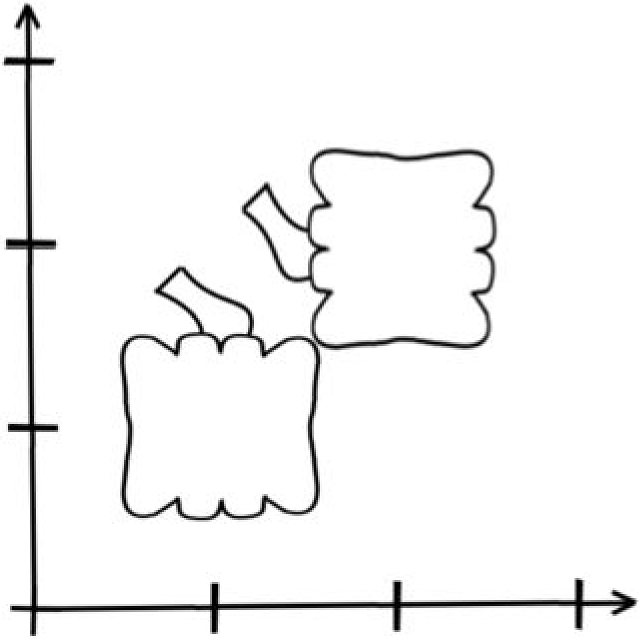
Drawing example: Pumpkin

- Actually draw out where a basis would move at each step (go left-right, maintain a basis as your temporary instead of a point)
- Wherever the origin winds up, draw the original image there using those axes

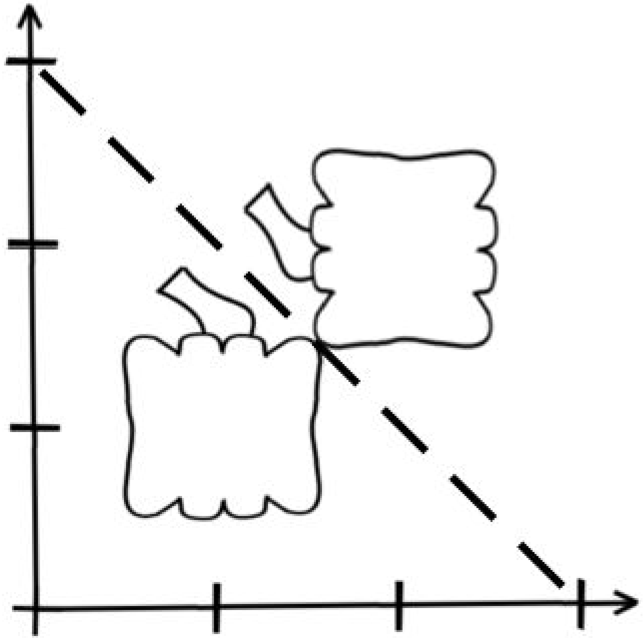


Drawing example: Pumpkin

- Why do we prefer left to right when building programs?
- Because of our temporary "partial matrices" when making the various products
  - Each sets us up for the next piece of a hierarchical model



Checking our Answer



Checking our Answer

- Easily summarized as a reflection around a line from (3,0) to (0,3)
- The sequence of transforms to do that reflection is different:
  - trans(0,3) * rot_z(-45) * $scale_y$(-1) * rot_z(45)* trans(0,-3)
  - What's the code for this?
- Numerically multiplying it out, it was the same matrix, surprise!!!