Computer Graphics

Discussion 3

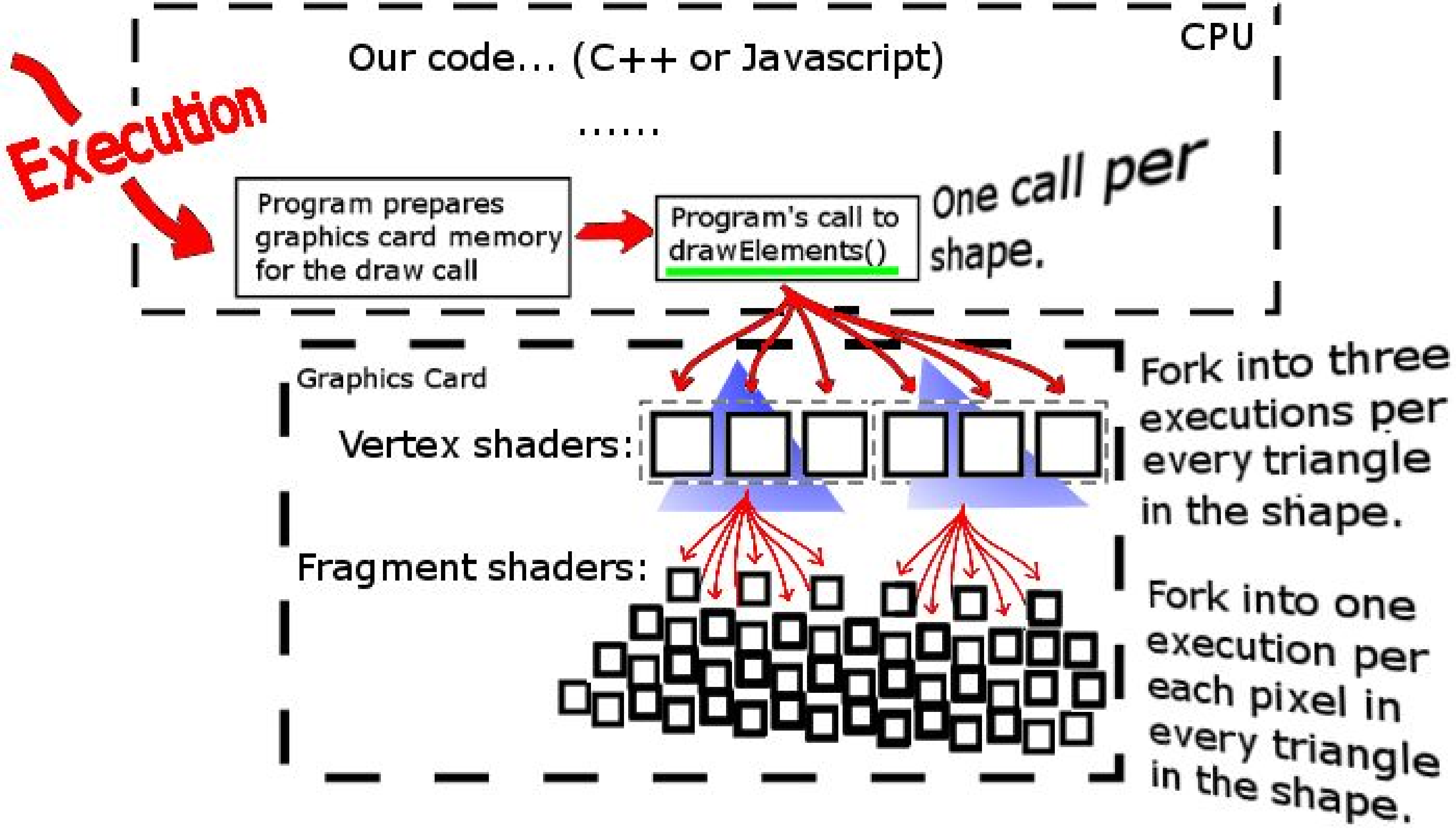
Garett Ridge and Sam Amin
garett@cs.ucla.edu samamin@ucla.edu

Part I: Another Code Tour

Draw calls and shaders

More of the code, including

- Shape class and its subclasses
  - Built_in_shapes.js contains the familiar Cube, Sphere, Cone, floating text, etc.
- What does each Shape::draw() call actually do?
  - Updates the graphics card with the current state in Shape::update_uniforms()
    - All the special matrices, other things relevant to coloring it in
  - Call to gl.drawElements() triggers the graphics card to work
    - Next thing to execute: Vertex shader program in index.html
    - Next thing to execute: Fragment shader program in index.html
- The two shaders
  - Summaries of these are provided in the code comments


CPU
Our code... (C++ or Javascript)
......
Execution
Program prepares graphics card memory for the draw call
Program's call to drawElements()
One call per shape.
Graphics Card
Vertex shaders:
Fork into three executions per every triangle in the shape.
Fragment shaders:
Fork into one execution per each pixel in every triangle in the shape.

Part II: The Matrix Chain

Sending triangles to their final places

Matrix Review

- Some of the best test questions require reasoning about long transform sequences on 2D drawings or graphs.
- The *order* of transformation is by far the hardest concept to get consistently right throughout the projects.

Matrix Review

- All the objects you draw on screen are drawn one vertex at a time, by starting with the vertex's xyz coordinate and then multiplying by a matrix to get the final xy coordinate on the screen.

$$\underset{M}{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & ? & ? \end{bmatrix}} * \underset{p}{\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}}$$

Transforms

$$\underset{M}{\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ 0 & 0 & ? & ? \end{bmatrix}} * \underset{p}{\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}}$$

- Before that matrix, the xyz coordinate is always some trivial value like (.5, .5, .5)
  - In the reference system of the shape itself
  - For example, a cube's own coordinates for its corners
- After that matrix, it's some different xy pixel coordinate denoting where that vertex will show up on the screen.
  - And z for depth, and a fourth number for translations / perspective effects
- That mapping is all that the transform does.

Matrix Order

- It's when our projects start using lines of code like this one that order suddenly becomes a thing:

```
model_transform = mult( model_transform, translate( 0, 0, .25 ) );
```

- That’s the general way to accumulate new little pieces into model and camera matrices.

Matrix Order

```
model_transform = mult( model_transform, translate( 0, 0, .25 ) );
```

- It takes your variable's current matrix and multiplies another mat4 to the right of it, and that becomes the "new" matrix.
- Since each of them sticks another matrix onto the right side of the chain, changing the order of these lines in your code changes the left-to-right ordering of your multiplication formula!!!!

Matrix Order

- That’s the only reason order matters in your code.
- On an exam, you can just write the whole chain of matrices out left-to-right into a complete formula so there is no longer any concept of "first" or "last“, then you <u>can’t mess up order</u>

Matrix Order

- Order still happens in your head as soon as you try to interpret what that formula is doing.
- One of two different mindsets will happen:
  - Thinking in "points“ (right to left), or
  - Thinking in "axes“ (left to right).

Matrix Order

- To think in points picture *starting* by drawing your shape
  - Then stretch it into place with global transforms
- To think in axes picture *finishing* by drawing your shape
  - After applying transformations to move your axis / origin

Matrix Order Example

- Suppose we wanted to swing at a distance of 10 around some point (x, y, z).
- We’ll show two pieces of code that do that.
- The difference between them:
  - One pre-multiplies new terms to the chain,
  - and the other post-multiplies.

Matrix Order Example

Pre-multiplying is “Thinking in points” :

Building a shape first (in this case an orbit shape) and then moving the whole shape’s points to the arbitrary xyz point

```
model_transform = mult( translation( 0, 0, 10 ), model_transform );                           // Send the object to the orbit’s edge
model_transform = mult( rotation( this.animation_time, 0, 1, 0 ), model_transform );          // Over time, rotate everything within the orbit
model_transform = mult( translation( x, y, z ), model_transform );                             // Send the orbit to an arbitrary xyz point

this.m_cube.draw( ... );          // Draw the shape there
```

Matrix Order Example

Post-multiplying is "Thinking in axes": Opposite ordering of code lines

Bringing your <u>origin</u> over to the arbitrary xyz point, rotating there, then move this <u>new origin</u> out 10 units away from the pivot point:

```
model_transform = mult( model_transform, translation( x, y, z ) );                       // Move coordinate system to the arbitrary pivot point
model_transform = mult( model_transform, rotation( this.animation_time, 0, 1, 0 ) );     // Rotate our system over time
model_transform = mult( model_transform, translation( 0, 0, 10 ) );                      // Travel out along our newly rotated system

this.m_cube.draw( ... );          // Draw the shape there
```

Matrix Order Example

- The key thing to notice about these two pieces of code:
- If you try to write down the matrix products they create by hand, <u>they both create the same one</u>, even in terms of left to right ordering.
- On an exam both methods are available.

Part III: Code Stuff

Transformations and Syntax

Moving Objects

Your shader program applies the final matrix product to each point in your shape.

- The matrices used are 4x4; 3 dimensions for x, y, and z and the 4th to allow translations

Translation

$$\begin{bmatrix} 1 & 0 & 0 & X \\ 0 & 1 & 0 & Y \\ 0 & 0 & 1 & Z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scale

$$\begin{bmatrix} SX & 0 & 0 & 0 \\ 0 & SY & 0 & 0 \\ 0 & 0 & SZ & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotate about Z

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Moving Objects

- Applying transformations in different orders often produces different results
    - ex: non-uniform scale followed by rotate produces shear instead of rotated object

Example transformation: scale(2), then translate(1, 2, 3)

$$M_{\text{translate}} \cdot M_{\text{scale}} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 0 & 2 & 0 & 2 \\ 0 & 0 & 2 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} 2x+1 \\ 2y+2 \\ 2z+3 \\ 1 \end{pmatrix}$$

Order of Transformations

An example of transformation order:

```
[...]rotation(90, 0, 1, 0));
this.cube.draw(...);
[...]translation(10, 0, 0));
this.cube.draw(...);
[...]scale(1, 1, 3));
this.cube.draw(...);
```

Your three core transformations are:

```
rotation(angle, x, y, z)
translation(x, y, z)
scale(x, y, z);
```

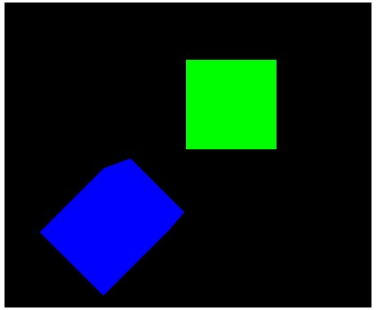
- The first cube will be rotated
- The second cube will be translated, then rotated
- The last cube will be scaled, then translated, then rotated

Order of Transformations

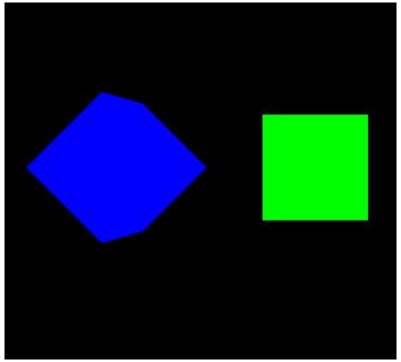
Another example-- a green cube followed by a blue cube:

```
this.cube.draw(..., greenMat);
```

```
[...]rotation(45, 0, 0, 1));
[...]translation(-2, 0, 0));
this.cube.draw(..., blueMat);
```



```
[...]translation(-2, 0, 0));
[...]rotation(45, 0, 0, 1));
this.cube.draw(..., blueMat);
```

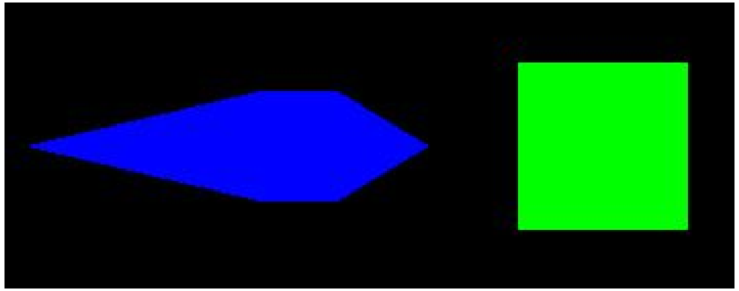


Order of Transformations

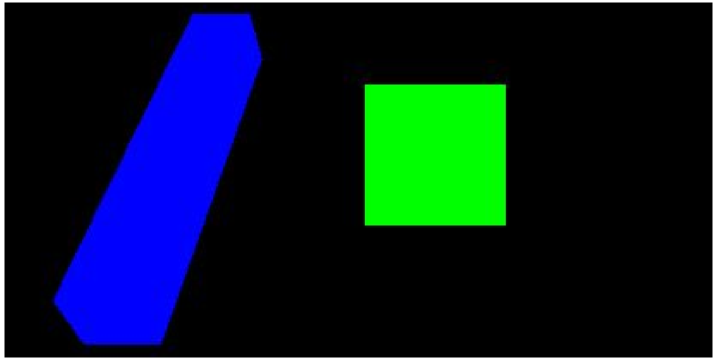
Building on the previous slide:

Shear vs. Not Shear

```
[...]scale(0.5, 0.5, 3));
[...]rotation(45, 1, 0, 0));
this.cube.draw(..., blueMat);
```



```
[...]rotation(45, 1, 0, 0));
[...]scale(0.5, 0.5, 3));
this.cube.draw(..., blueMat);
```



Transformations

Transformations are matrix operations

- Reading from top to bottom in your code: coordinate systems thinking
- Reading from bottom to top in your code: points thinking

Instead of undoing and redoing transformations, best to use a shortcut such as a stack to save states for you.

Transformations

model_transform = mult(model_transform, translation(0,0,-2));

- You use mult to apply a transformation to your current model transformation matrix
- Your options for transformations are translation, scale, and rotation. Rotation takes an extra parameter:
    - rotation(degree, x, y, z)
- After setting up your model_transform matrix how you like, make a call to draw

Draw

this.shape.draw(this.graphicsState, model_transform, material);

- Calls the draw function for whichever shape you're using, passing in the model transform matrix to determine how its unit shape will be affected
- Materials are defined in your code as needed:
    - var purplePlastic = new Material(vec4(.9, .5, .9, 1), .2, .5, .8, 40);
    - 1st parameter: color (4 floats in RGBA format), 2nd: ambient light, 3rd: diffuse reflectivity, 4th: specular reflectivity, 5th: smoothness exponent, 6th: texture image
    - 6th parameter, a picture, is not required
- One graphicsState per program; keeps track of all your stuff

Stack

When you do a complicated series of transformations, you want to be able to get back to where your model_transform was before. You could handle it by undoing all your transformations, or you could just make a stack:

- var stack = [];
    - Now just push your current model_transform matrix before a transformation and pop it back afterwards

```
stack.push(model_transform);

model_transform = stack.pop();
```

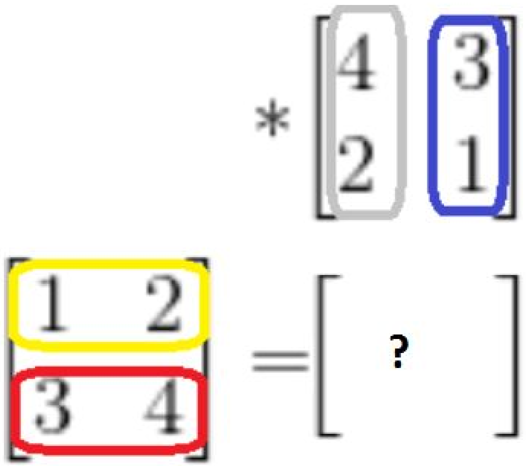
Part IV: Mat Multiplication is NOT COMMUTATIVE

Repeat after me

Matrix Multiplication Review

Use dot products for all 4 cells:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} * \begin{bmatrix} 4 & 3 \\ 2 & 1 \end{bmatrix} = ?$$



Matrix Multiplication Review

Use dot products for all 4 cells:

$$\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix} * \begin{bmatrix}4 & 3\\ 2 & 1\end{bmatrix} = ?$$

$$* \begin{bmatrix}\boxed{\begin{matrix}4\\2\end{matrix}} & \boxed{\begin{matrix}3\\1\end{matrix}}\end{bmatrix}$$

$$\begin{bmatrix}\boxed{1 \quad 2}\\ \boxed{3 \quad 4}\end{bmatrix} = \begin{bmatrix}8 & 5\\ 20 & 13\end{bmatrix}$$

Matrix Multiplication is NOT commutative.

Given Matrix A and Matrix B that are non-trivial / diagonal,

AB != BA

Matrix Multiplication is NOT commutative.

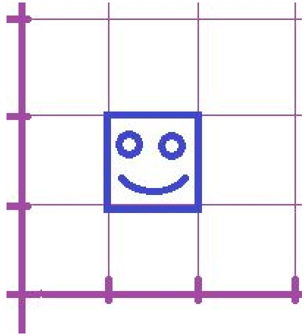
Remember our old rotation matrix:

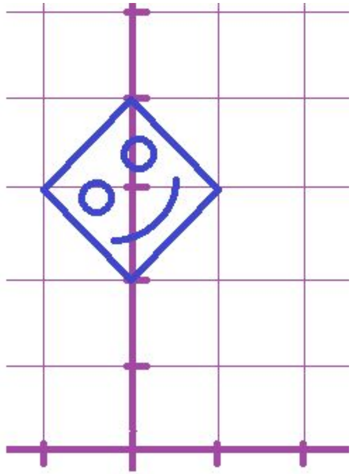
$$scale(\sqrt{2}) * rotate_z(45^\circ) = ?$$

$$\begin{bmatrix} \sqrt{2} & \\ & \sqrt{2} \end{bmatrix} * \begin{bmatrix} \cos(45^\circ) & -\sin(45^\circ) \\ \sin(45^\circ) & \cos(45^\circ) \end{bmatrix} = ?$$

$$\Rightarrow \begin{bmatrix} \sqrt{2} & \\ & \sqrt{2} \end{bmatrix} * \begin{bmatrix} \sqrt{2}/2 & -\sqrt{2}/2 \\ \sqrt{2}/2 & \sqrt{2}/2 \end{bmatrix} = ?$$

$$\Rightarrow \begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix}$$



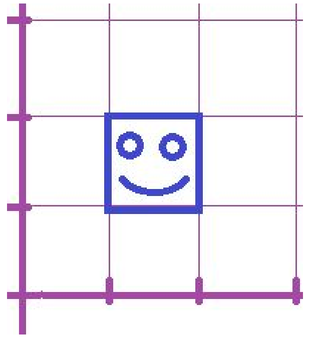


Matrix Multiplication is NOT commutative.

Suppose we modify it with a non-uniform scale matrix from the left:

$$\begin{bmatrix} 1 & \\ & 2 \end{bmatrix} * \begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} & \\ & ? & \end{bmatrix}$$

Where do the corners of the face go if we use this one?

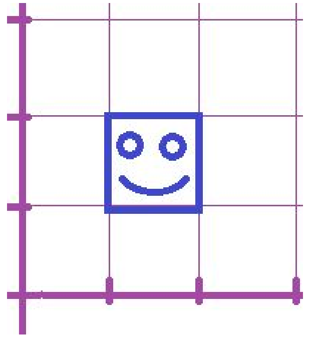


Matrix Multiplication is NOT commutative.

Suppose we modify it with a non-uniform scale matrix from the left:

$$\begin{bmatrix} 1 & \\ & 2 \end{bmatrix} * \begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 2 & 2 \end{bmatrix}$$

Where do the corners of the face go if we use this one?



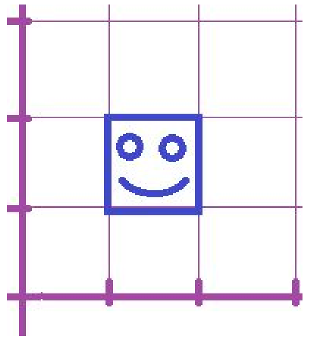
Matrix Multiplication is NOT commutative.

Suppose we modify it with a non-uniform scale matrix from the left:

$$\begin{bmatrix} 1 & -1 \\ 2 & 2 \end{bmatrix} * \begin{bmatrix} 1 \\ 1 \end{bmatrix} = [?]$$

$$\begin{bmatrix} 1 & -1 \\ 2 & 2 \end{bmatrix} * \begin{bmatrix} 2 \\ 2 \end{bmatrix} = [?]$$

Where do the corners of the face go if we use this one?



Matrix Multiplication is NOT commutative.

Suppose we modify it with a non-uniform scale matrix from the left:

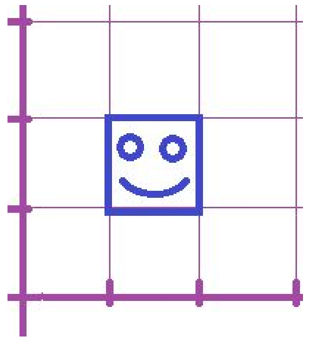
We sheared it!

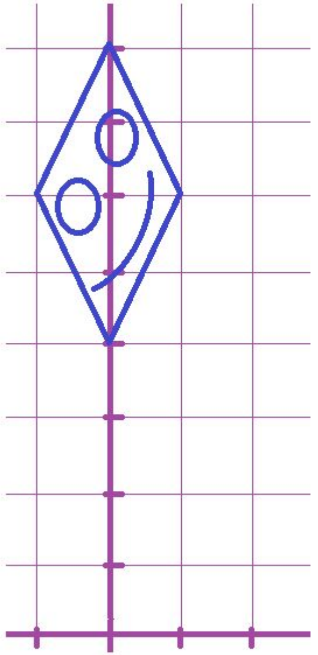
$$\begin{bmatrix} 1 & -1 \\ 2 & 2 \end{bmatrix} * \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 4 \end{bmatrix}$$

$$\begin{bmatrix} 1 & -1 \\ 2 & 2 \end{bmatrix} * \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} -1 \\ 6 \end{bmatrix}$$

$$\begin{bmatrix} 1 & -1 \\ 2 & 2 \end{bmatrix} * \begin{bmatrix} 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 6 \end{bmatrix}$$

$$\begin{bmatrix} 1 & -1 \\ 2 & 2 \end{bmatrix} * \begin{bmatrix} 2 \\ 2 \end{bmatrix} = \begin{bmatrix} 0 \\ 8 \end{bmatrix}$$





Matrix Multiplication is NOT commutative.

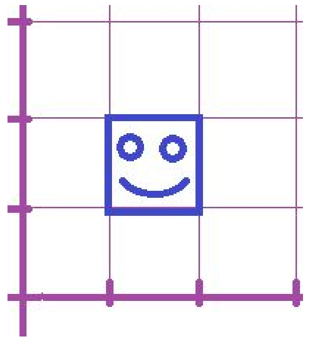
Let’s try the product the other way around now...

Matrix Multiplication is NOT commutative.

Suppose we modify it with a non-uniform scale matrix from the right:

Where do the corners of the face go if we use this one?

$$\begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix} * \begin{bmatrix} 1 & \\ & 2 \end{bmatrix} = \begin{bmatrix} & \\ & ? & \end{bmatrix}$$

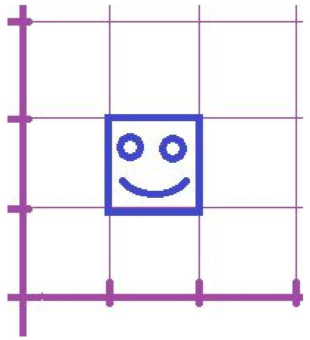


Matrix Multiplication is NOT commutative.

Suppose we modify it with a non-uniform scale matrix from the <u>right</u>:

$$\begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix} * \begin{bmatrix} 1 & \\ & 2 \end{bmatrix} = \begin{bmatrix} 1 & -2 \\ 1 & 2 \end{bmatrix}$$

Where do the corners of the face go if we use this one?



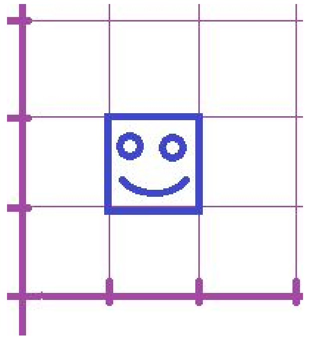
Matrix Multiplication is NOT commutative.

Suppose we modify it with a non-uniform scale matrix from the right:

$$\begin{bmatrix} 1 & -2 \\ 1 & 2 \end{bmatrix} * \begin{bmatrix} 1 \\ 1 \end{bmatrix} = [?]$$

$$\begin{bmatrix} 1 & -2 \\ 1 & 2 \end{bmatrix} * \begin{bmatrix} 2 \\ 2 \end{bmatrix} = [?]$$

Where do the corners of the face go if we use this one?



Matrix Multiplication is NOT commutative.

Suppose we modify it with a non-uniform scale matrix from the right:

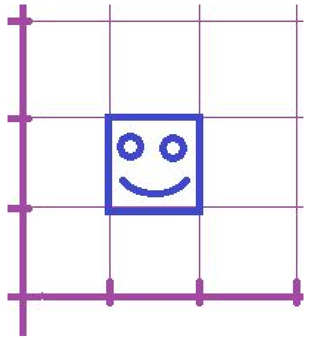
We didn't shear it!

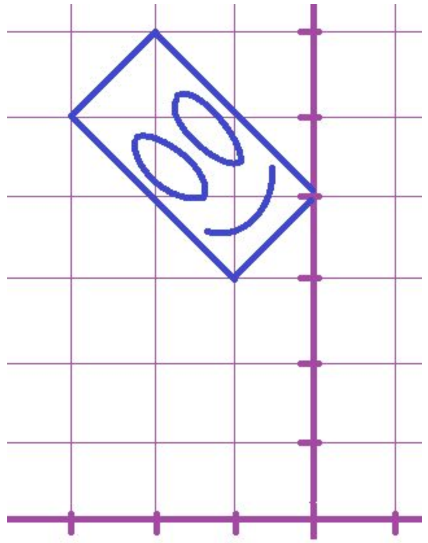
$$\begin{bmatrix} 1 & -2 \\ 1 & 2 \end{bmatrix} * \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} -1 \\ 3 \end{bmatrix}$$

$$\begin{bmatrix} 1 & -2 \\ 1 & 2 \end{bmatrix} * \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} -3 \\ 5 \end{bmatrix}$$

$$\begin{bmatrix} 1 & -2 \\ 1 & 2 \end{bmatrix} * \begin{bmatrix} 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 4 \end{bmatrix}$$

$$\begin{bmatrix} 1 & -2 \\ 1 & 2 \end{bmatrix} * \begin{bmatrix} 2 \\ 2 \end{bmatrix} = \begin{bmatrix} -2 \\ 6 \end{bmatrix}$$





Matrix Multiplication is NOT commutative.

Non-uniform scales anywhere in the matrix chain could produce a shear later.

Since shears are rarely desired, non-uniform scales are typically put at the very end of a matrix chain (to the immediate left of the point) and then we undo that most recent part of the transform before drawing the next shapes.

Matrix Multiplication is NOT commutative.

Non-commutativity is the reason that we have to unwrap our matrices in reverse order. Any other order would have a different effect and wouldn’t go back to the prior state.

Matrix Multiplication is NOT commutative.

Let’s look at another instance of scale matrices misbehaving…

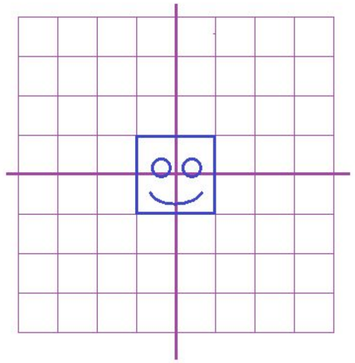
Translate * Scale

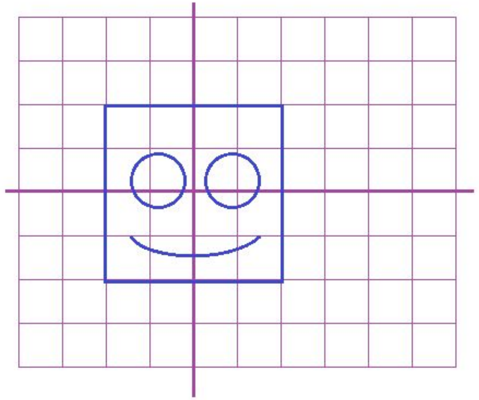
vs

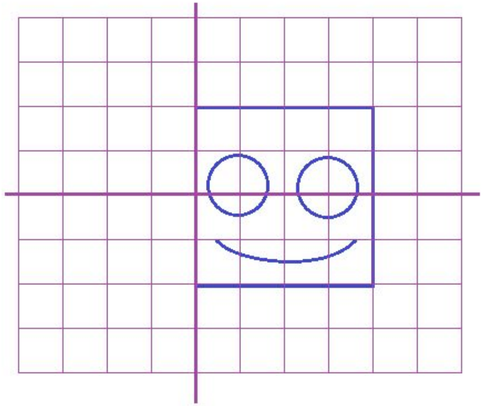
Scale * Translate

Matrix Multiplication is NOT commutative.

$\text{Translate}_x(2) * \text{Scale}(2) * p$



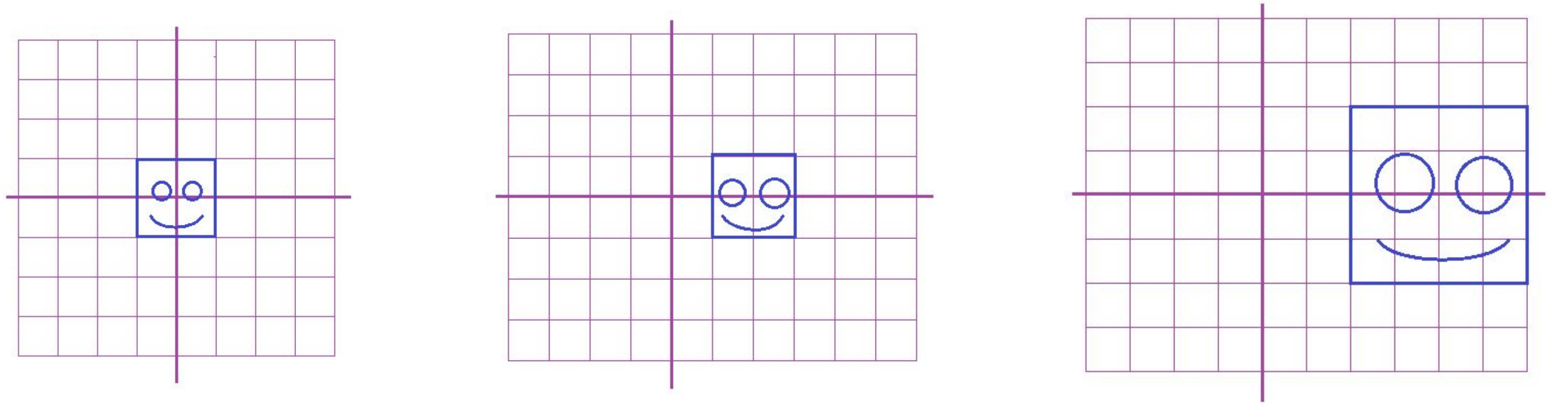




(Read it backwards -- the scale happens to points first because it’s closer to the point in the equation)

Matrix Multiplication is NOT commutative.

$\text{Scale}(2) * \text{Translate}_x(2) * p$



(Read it backwards -- the translate happens to points first because it’s closer to the point in the equation) Notice how the gap scales away from the origin same as the face!

Matrix Multiplication is NOT commutative.

Another way to think of the previous two slides:

Read the multiplication forwards instead, and this time let the coordinate system axes follow you instead of remaining static on each transform.

You should come up with the same final picture even though you did the operations backwards, because unlike last time, now a scale affects how far a future translate moves - it scales your basis.

This would be thinking in **bases** instead of **points**.