Transformations

Motivation

- We want to move objects around the screen
- We want:
  - Efficiency
  - Flexibility
  - Ease of use
- We started discussing matrices last time...

Recall

- We defined a 2D vector as having x,y and w
  - w = 0 for direction, 1 for position

Recall

- Translation matrix:

$$\begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{bmatrix} = \begin{bmatrix} x+t_x & y+t_y & 1 \end{bmatrix}$$

- Or:

$$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x+t_x \\ y+t_y \\ 1 \end{bmatrix}$$

- If w==0: No effect.

Recall

- Rotation

$$\begin{bmatrix} x & y & w \end{bmatrix} \begin{bmatrix} cos\theta & sin\,\theta & 0 \\ -sin\,\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} xcos\,\theta - ysin\,\theta & xsin\,\theta + ycos\theta & w \end{bmatrix}$$

- Or:

$$\begin{bmatrix} cos\,\theta & -sin\,\theta & 0 \\ sin\,\theta & cos\,\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix} = \begin{bmatrix} xcos\,\theta - ysin\,\theta \\ xsin\,\theta + ycos\,\theta \\ w \end{bmatrix}$$

- Notice: w is preserved, whether it's zero or one!

So...

- Suppose T_1 is a translation matrix and T_2 is another translation matrix
  - Suppose they are designed for row-vectors: v*M*, not *M*v
- Let $v' = vT_1$
- What does v' represent?

So...

- v' is the translation of v by whatever amount is represented by T_1
- Now, what if we compute $v'' = v'T_2$
- What do we get?

Result

- v” is the translation of v’ by whatever is represented by T_2
  - Or, translation of v by whatever is represented by T_1 translated by whatever is represented by T_2
- In other words, we applied T_1 first, then T_2

Or

- We have: $v'' = v'T_2$
- Which is: $v'' = (vT_1)T_2$
- But: Matrix multiplication is *associative*!
  - What does that mean?

Result

- $(v * T_1)T_2 = v(T_1T_2)$
- We could precompute T_1T_2 and then use *that* as our transformation matrix

Order

- Translations are *commutative*, so order doesn't matter
- But what about rotations?
  - In general, they aren't commutative
- Suppose we have rotation matrices R_1, R_2
- If we compute $v' = vR_1R_2$... What do we have?
  - $vR_1R_2 = (vR_1)R_2 = v(R_1R_2)$
- We apply R_1 first, then R_2

Order

- No matter how many rotations/translations we have, same idea applies
- If we have $v' = vR_1T_1R_2T_2$
  - Apply R_1 to v, then apply T_1 to that, then apply R_2 to that, and finally apply T_2 to that value
- Again, we can precompute $R_1T_1R_2T_2$ and use that:
  $M = R_1T_1R_2T_2$
  $v' = vM$

Order

- This allows us to express our transformation order
- If we use RT we are rotating, then translating
- If we use TR we are translating, then rotating
- This ordering applies no matter how many matrices we chain together

Note

- What if we interpret vectors as column-matrices?
- We must postmultiply them
- What if we compute $v' = TRv$?

Associativity

- Apply associative rule:
- $v' = TRv = (TR)v = T(Rv)$
- We apply rotation first, then translation

Column Vector

- What if we have $v' = R_1 T_1 R_2 T_2 v$?
  - Apply T_2 then R_2 then T_1 then finally R_1
  - Notice: "Temporal" order is read *right to left*
  - But when we interpreted vector as row-vector, temporal order was *left to right*

Scaling

- We mentioned that scaling was the last of the “big three” matrices
- Scale matrix:

$$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- Here, transpose doesn’t change matrix, so same thing works for either way of multiplying vectors (pre or post)

Application

- We'd like to use matrices for our shaders
- We need to extend Program.py (there's a copy in the zipfile later in the notes)

Example

- We'll declare a uniform: mat3 worldMatrix;
- Why the name "worldMatrix"?

Spaces

- Coordinate spaces:
  - Object space: Space in which our object is defined
    - Usually centered around (0,0) or else touches (0,0)
  - World space: “Universal” coordinate system
    - Objects are positioned relative to each other “in the world”
- worldMatrix transforms points from object space to world space

Shader

- VS is the one that must use worldMatrix
- We can’t multiply vec2 by mat3
  - Sizes don’t match
- So our shader will need to do a bit of data shuffling...

Code

```
layout(location=0) in vec2 position;
...tex coord, if we have it...
void main(){
    vec3 p = vec3(position,1.0);
    p = p * worldMatrix;
    gl_Position = vec4(p.xy, -1, 1 );
    ...
}
```

- Notice: We assign homogeneous coordinate (w)=1 when initializing p
  - Because ‘position’ is a location, not a direction

Example

- Let's see some examples of matrices in action
- First, we have a small testbed... testbed1.zip
  - In-class: Make the alien move around with WASD keys
  - In-class: Make the earth orbit around the sun
  - In-class: Make the moon orbit around the earth

Assignment

- Finish the framework. You probably only need to change Car.py.
  - While space is held down, the wheels should go around
  - A/D makes the car tilt up and down (like it's driving up or down a hill)
  - W/S makes the car drive left or right across the screen (this won't necessarily be the direction it's facing if A or D have been pressed)
- For 33% bonus, have W/S make the car drive in forward or reverse in the direction it's facing (this might be diagonal if A or D have been pressed)

Sources

- Jim Van Verth. Understanding Rotations. http://www.essentialmath.com/GDC2012/GDC2012_JMV_Rotations.pdf

Created using LaTeX.

Main font: Gentium Book Basic, by Victor Gaultney. See http://software.sil.org/gentium/
Monospace font: Source Code Pro, by Paul D. Hunt. See https://fonts.google.com/specimen/Source+Code+Pro and http://sourceforge.net/adobe
Icons by Ulisse Perusin, Steven Garrity, Lapo Calamandrei, Ryan Collier, Rodney Dawes, Andreas Nilsson, Tuomas Kuosmanen, Garrett LeSage, and Jakub Steiner. See http://tango-project.org