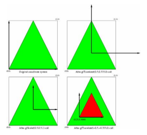
# Display Lists
# 2D Transformations



CS148: Intro to CG
Instructor: Dan Morris
TA: Sean Walker
June 30, 2005

## Outline for today

- Moving Data Around
- 2D Transformations
- SIGGRAPH video break
- Matrix Transformations
- Composite Transformations

## Sending data to the video card

- OpenGL needs to know where you want to put your vertices
- There are several ways to send your vertices to the video hardware

- The first part of today's lecture will explore three different ways

## Approach 1: Immediate Mode

- In "immediate mode" (everything so far in CS148), commands are sent to the video card immediately (hence the name)

```
// Somewhere in my drawing code:
glBegin(GL_POINTS);
   glVertex2i(10,20);
   glVertex2i(40,50);
glEnd();
```

**What's wrong with this approach?**

## Approach 2: Display lists [list.cpp linelist.cpp stroke.cpp]

```
// Just once, in some initialization function:

// Ask OpenGL for one new display list
g_mySlickFerrariDL = glGenLists(1);

// Record the display list
glNewList(g_mySlickFerrariDL, GL_COMPILE);
glBegin(GL_TRIANGLES);
   // …maybe millions of glVertex calls…
   // …maybe change colors, other GL commands…
glEnd();
glEndList();

// Every frame I just need to do:
glCallList(g_mySlickFerrariDL);
```

## Display lists: Pros and Cons

- Save the overhead of making 1,000,000 function calls per frame
- Can draw the same object multiple times with different OpenGL state

**What's one other advantage of using display lists?**

**What's one disadvantage of using display lists?**

## Immediate Mode with Arrays

- Usually I don't know all my vertex locations when I code my program
- So let's say I read my vertices from a file into a big array…

```
glBegin(GL_TRIANGLES);
for(int i=0; i<numVertices*6; i+=6) {
    glVertex2i(vertices[i+0],vertices[i+1]);
    glVertex2i(vertices[i+2],vertices[i+3]);
    glVertex2i(vertices[i+4],vertices[i+5]);
}
glEnd();
```

**What's wrong with this approach?**

## Approach 3: Vertex Arrays [varrays.cpp]

- Tell OpenGL to grab all of your vertices from a block of memory

```
// somewhere in my drawing code

// tell GL I'm going to use vertex arrays
glEnableClientState(GL_VERTEX_ARRAY);

// tell GL where my vertices live
glVertexPointer(2,GL_FLOAT,0,myvertices);

// tell GL to draw my triangles
glDrawArrays(GL_TRIANGLES,0,numVertices);

// leave GL the way I found it
glDisableClientState(GL_VERTEX_ARRAY);
```

## Vertex Arrays: Pros and Cons

- Pro: Avoid the overhead of 1,000,000 function calls
- Pro: You can change the contents of the array whenever you want (unlike display lists)
- Con: Vertex data still gets copied every frame

## What do super-hard-core game developers do?

- Method 4: Vertex buffer objects
- Allocate memory *on the video card*, only transfer once
- Similar to display lists but often faster

## Outline for today

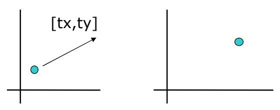
- Moving Data Around
- 2D Transformations
- SIGGRAPH video break
- Matrix Transformations
- Composite Transformations

## 2D Transformations

- *Transformations* are functions that change the *position* of a *point*
  - Take one point in $R^n$, return another point in $R^n$
- If we apply a transformation to every point in an object, we can change the *shape* or *position* of the whole object

## 2D Translation

x′ = x + tx      y′ = y + ty

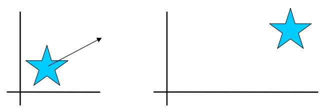


tx = 2.0
ty = 1.0

- Moves input point by a vector [tx,ty]

## 2D Translation for Objects

x′ = x + tx      y′ = y + ty

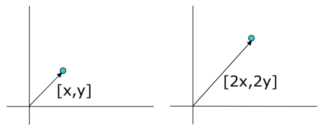


tx = 2.0
ty = 1.0

- Moves entire object by [tx,ty]

## 2D Scale

x′ = x * sx      y′ = y * sy



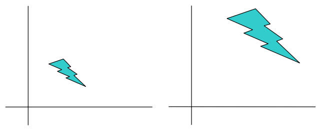
sx = sy = 2.0
(this is a *uniform* scale)

- Multiplies input point by (sx,sy)
- Moves point relative to the origin

**What would the scale (1,-1) do?**

## 2D Scale for Objects

x′ = x * sx      y′ = y * sy



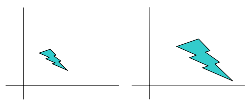
sx = sy = 2.0

- Resizes entire object

**What else did this do to my object?**

## 2D scaling with a fixed point

Maybe we want (xf,yf) not to move when we scale…



sx = sy = 2.0
(xf,yf) = object center

**What point doesn’t move when we scale?**

**How can that help us scale with a fixed point?**

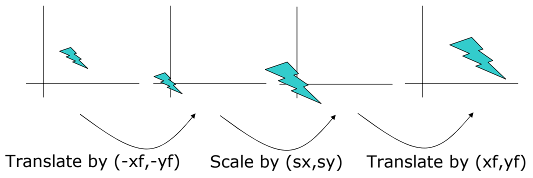
x′ = xf + (x - xf)sx      y′ = yf + (y - yf)sy

x′ = x * sx + (1 - sx) xf      y′ = y * sy + (1 - sy) yf

**Why use the second form of these equations?**

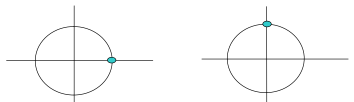
## 2D scaling with a fixed point

- What we really just did was combine two translations and a scale



## 2D Rotation

- Moves a point θ degrees along a circle centered at the origin



$\theta = 90°$

- But the other transformations had a formula on their slides…
  - Doesn't rotation get a formula too? Isn't rotation good enough for Mr. Bigshot graphics lecturer? This is outright transformationism! As someone who believes in the equity of all transformations, I'm appalled by this behavior and I'm going to write a report to the something or other board of something or other unless I get a formula for rotation also. Right now.

## 2D Rotation

- This is straightforward in polar coordinates:

$x = r \cos \phi \qquad y = r \sin \phi$

$x' = r \cos(\phi+\theta) \qquad y' = r \sin(\phi+\theta)$

- But converting to polar coordinates is a mess
- So we'll use trig identities…

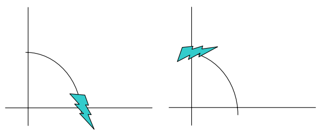
$x' = r \cos(\phi+\theta) = r \cos \phi \cos \theta - r \sin \phi \sin \theta$

$y' = r \sin(\phi+\theta) = r \cos \phi \sin \theta + r \sin \phi \cos \theta$

$\mathbf{x' = x \cos \theta - y \sin \theta}$

$\mathbf{y' = x \sin \theta + y \cos \theta}$

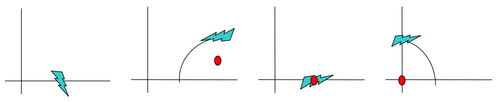
## 2D Rotation for objects



**What else did this do to my object?**

**How do we fix this?**

## 2D Rotation about a fixed point



- Combine two translations and a rotation
- To rotate about a point (xr,yr):

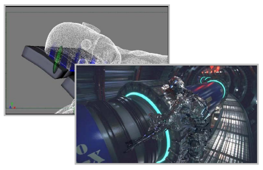
$x' = xr + (x - xr) \cos \theta - (y - yr) \sin \theta$

$y' = yr + (x - xr) \sin \theta + (y - yr) \cos \theta$

## SIGGRAPH video break

Ron Fedkiw et al (Stanford)… various physical simulation techniques

- Physical simulation: using physics to get realistic behavior from virtual objects
  - Examples: f = ma, Navier-Stokes
- Simulation vs. rendering



## Outline for today

- Moving Data Around
- 2D Transformations
- SIGGRAPH video break
- Matrix Transformations
- Composite Transformations

## Representing Transformations

- We do *lots* of transformations in computer graphics
- We do so many transformations in computer graphics that I want to say that again
- We do *lots* of transformations in computer graphics
- Need an efficient way of representing transformations

## Homogeneous Coordinates

- Write a point (x,y) as a triple:

  [xh,yh,w]

  …where xh = x*w, yh = y*w
- w is called the 'homogeneous coordinate' and is usually equal to one
- When w = 1, x = xh and y = yh

## Matrix transformations

- In homogeneous coords, our basic transformations can be written as matrix multiplications
- I submit to you that this matrix represents a translation by [tx,ty]:

$$\begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix}$$

## 2D Matrix Translation

$$\begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 1*x+0*y+tx*1 \\ 0*x+1*y+ty*1 \\ 0*x+0*y+1*1 \end{bmatrix} = \begin{bmatrix} x+tx \\ y+ty \\ 1 \end{bmatrix}$$

- From a few slides ago:

x' = x + tx     y' = y + ty

- Hooray! It works!

## 2D Matrix Transformations

Translation by [tx,ty]

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Scale by [sx,sy]

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} ? & ? & ? \\ ? & ? & ? \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Rotation by θ

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

## Outline for today

- Moving Data Around
- 2D Transformations
- SIGGRAPH video break
- Matrix Transformations
- Composite Transformations

## Composite transformations

- Often we want to apply multiple transformations sequentially
- Easy to do with matrices: just build a new transformation matrix as the product of multiple transformations
- **Any sequence of transformations can be collapsed into one matrix by multiplying the individual transformation matrices**

## Example: Composite Translation

- Let's say we want to translate a point by [tx1,ty1], then by [tx2,ty2]
- The two transformation matrices are:

$$\begin{bmatrix} 1 & 0 & tx1 \\ 0 & 1 & ty1 \\ 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & tx2 \\ 0 & 1 & ty2 \\ 0 & 0 & 1 \end{bmatrix}$$

- Their product is:

$$\begin{bmatrix} 1 & 0 & tx1 \\ 0 & 1 & ty1 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & tx2 \\ 0 & 1 & ty2 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & tx1+tx2 \\ 0 & 1 & ty1+ty2 \\ 0 & 0 & 1 \end{bmatrix}$$

## Example: Rotation with a fixed point

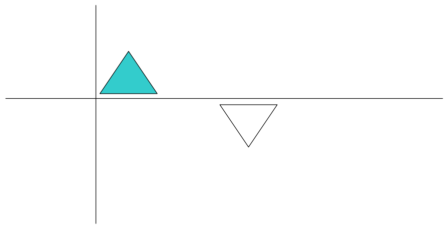
- Let's say we want to rotate a point by θ degrees, with a fixed point [fx,fy]
- Translate, then rotate, then translate

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

$$\begin{bmatrix} \cos\theta & -\sin\theta & -\cos\theta * \mathrm{fx} + \sin\theta * \mathrm{fy} + \mathrm{fx} \\ \sin\theta & \cos\theta & -\sin\theta * \mathrm{fx} - \cos\theta * \mathrm{fy} + \mathrm{fy} \\ 0 & 0 & 1 \end{bmatrix}$$

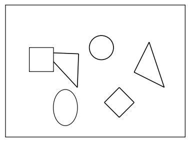
## Fun with composite transformations

- How would you transform the dark triangle into the light triangle?



## Instances

- Consider a scene composed of many simple 2D objects



- we could store vertices for each object
- …or we could define simple primitives at convenient locations, and apply transformations before I draw each object

## Instances as Display Lists

- A typical program structure in graphics
  - Define vertices for a few primitives that will appear in many places
  - Render those vertices into a display list for each primitive
  - At every frame of your program, ask OpenGL to apply a separate transformation each time you call your display list
- [movingsquares.cpp]
  - A bigger program that we now have the tools to understand

## Next time

- 3D Transformations
- Transformations in OpenGL

$$\longrightarrow \begin{bmatrix} 1.3 & 5.6 & 6.5 & 55.2 \\ 3.2 & 2.1 & 12.33 & 12.1 \\ 56 & 22.3 & 12.1 & 22.1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \longrightarrow$$