CS 432 – Interactive Computer Graphics

Lecture 3 – Part 1

Transformations

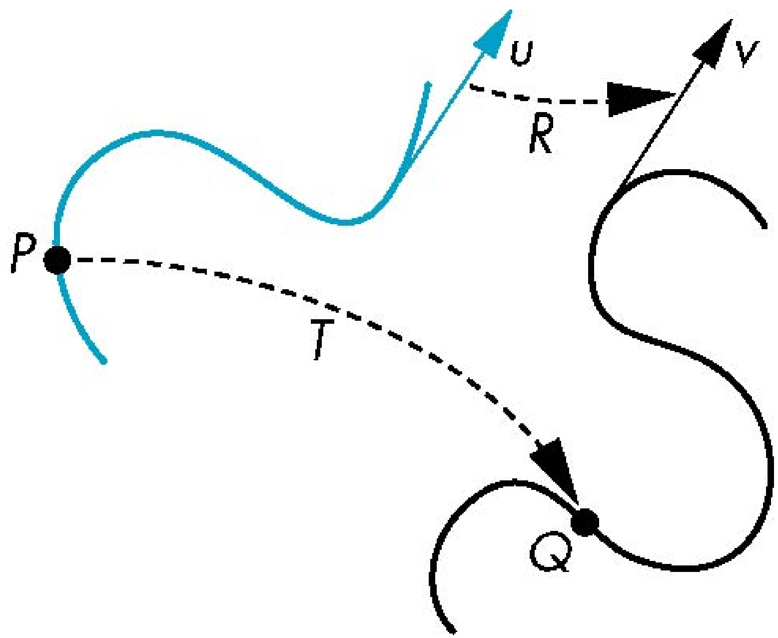
2D Transformations

Introduction

- We have already discussed two coordinate systems
  - Camera coordinate system
  - Window coordinate system
- Let's add in a 3rd, the *model coordinate system*.
- Typically we define object in their own space
- Then to build our "world" we need to *transform* them to be in the world.
- This transformation process usually involves:
  - Scaling
  - Moving (translating)
  - Rotating

Transformations

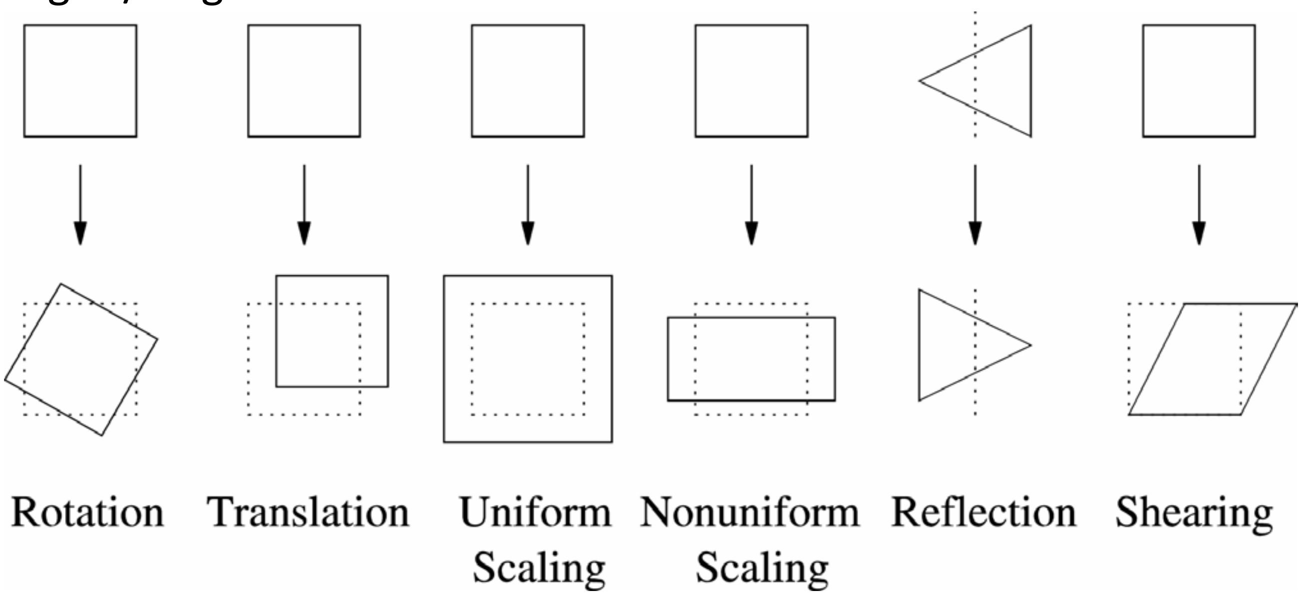
- A transformation maps points to other points



$v=T(u)$

Introduction

- We can do all the previously mentioned transformations efficiently by matrix operations
- Note: This set of transformations are call *affine transformations* and
  - Parallel lines are preserved
  - Angles/lengths are not



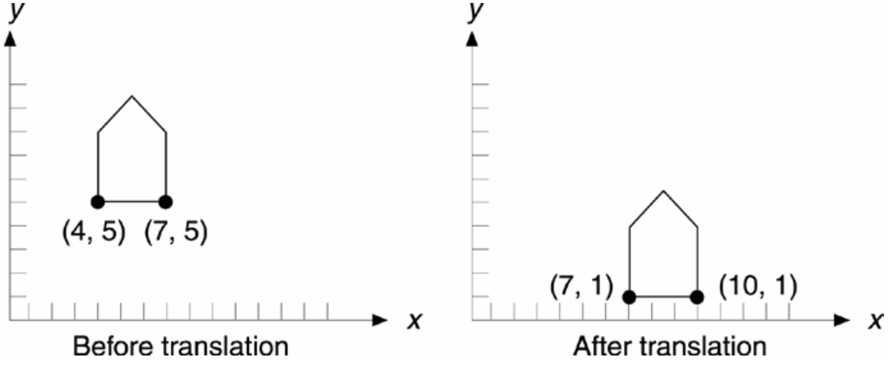
2D Transforms: Translation

- Our first transformation will be *translation*
- Translation is the rigid motion of points to new locations

$$x' = x + d_x$$
$$y' = y + d_y$$

- Let's assume that our vertex is a column vector: $P = \begin{bmatrix} x \\ y \end{bmatrix}$, then we can do translation easily with matrix addition:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} d_x \\ d_y \end{bmatrix}$$
$$P' = P + T$$



2D Transforms: Scale

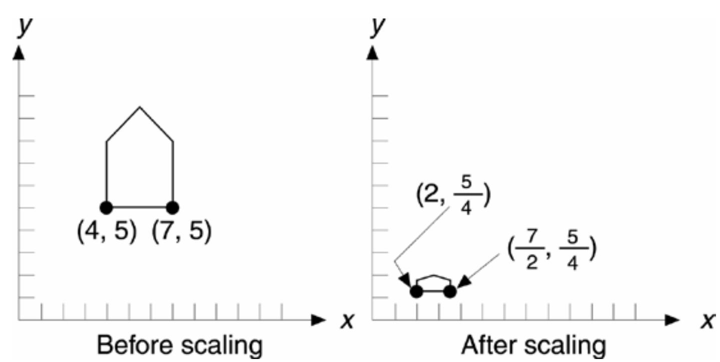
- Scaling is the stretching of points along axes:

$$x' = s_x x$$
$$y' = s_y y$$

- We can do this via matrix multiplication:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

- Or $P' = SP$



2D Transforms: Rotation

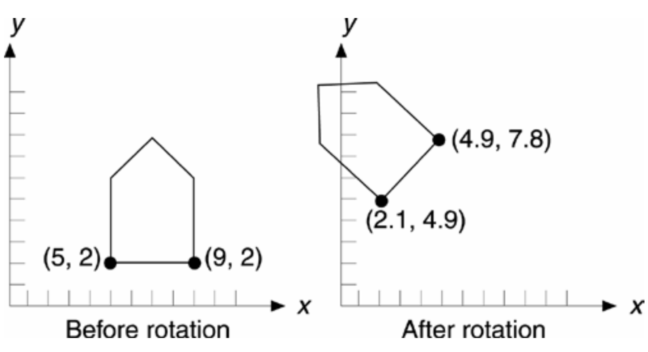
- Most rotations are done about the origin
- If we want to rotate around another point we can
  1. Translate from that point
  2. Rotate about the origin
  3. Translate back
- Rotation about the origin in 2D can be done as:

$$x' = x \cdot \cos\theta - y \cdot \sin\theta$$
$$y' = x \cdot \sin\theta + y \cdot \cos\theta$$

  - Where a positive angle = CCW
- We can also do this via matrix multiplication

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

- Or $P' = RP$



Homogeneous Coordinates

- Ok cool so we can use matrix operations to do all our affine transformations
  - Translations done via addition
  - Scaling and Rotation done via multiplication.
- Is there a way that we could make them all be multiplications?
  - Add another dimension with the value of one!
    - Represent a 2D point using a 3D vector
    - (Later) represent a 3D point using a 4D vector.
- This representation is called the *homogenous representation*
  - Because all the operations are done the same (*homo*)

Matrix Representation of 2D Affine Transformations

- Translation

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

- Scale

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

- Rotation

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

- Shear

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

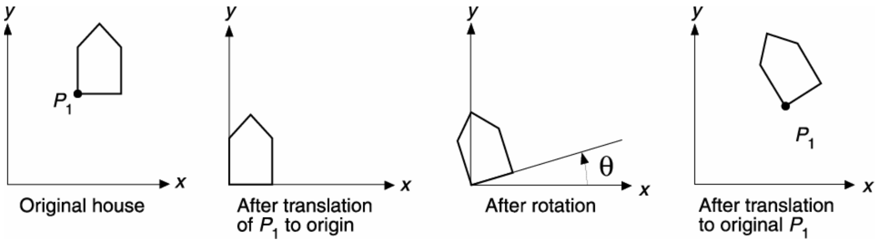
Composition of 2D Transforms

- Ok so now we can do all our transformations as multiplications
  - Big deal!?
- Well often we want to chain together transformations
  - First scale, then rotate, then translate…
- Sure we could
  - Take our (homogenous) vertex and multiply it by a matrix to do the first transformation
  - Then take that result, multiply it by a matrix to do the second transformation
  - Etc..
- But we can do better than this!
  - We can compute a *composite* transformation matrix that is the product of the individual transformation matrices
  - Just do this once, and use it often!

Composition of 2D Transforms

- For example, this let's us rotate around some arbitrary point $P1 = (x_1, y_1)$

1. Translate all point by –P1
2. Rotate
3. Translate back by $P1$



$$T(x_1 y_1) \cdot R(\theta) \cdot T(-x_1, -y_1)$$

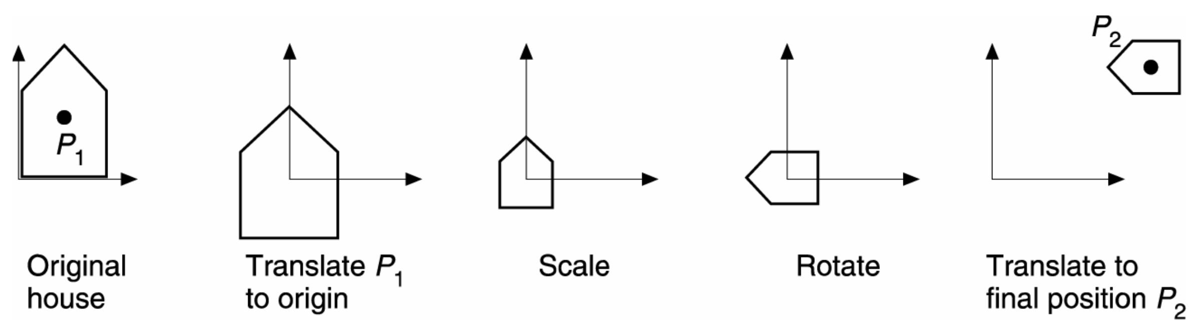
$$= \begin{bmatrix} 1 & 0 & x_1 \\ 0 & 1 & y_1 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_1 \\ 0 & 1 & -y_1 \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \cos\theta & -\sin\theta & x_1(1-\cos\theta) + y_1 \sin\theta \\ \sin\theta & \cos\theta & y_1(1-\cos\theta) - x_1 \sin\theta \\ 0 & 0 & 1 \end{bmatrix}$$

Composition of 2D Transforms

- Example: scale + rotate object around point $P1$ and move to $P2$
  1. Move $P1$ to the origin
  2. Scale
  3. Rotate
  4. Translate to $P2$

$$T(x_2, y_2) \cdot R(\theta) \cdot S(s_x, s_y) \cdot T(-x_1, -y_1)$$



Composition of 2D Transforms

- Be sure you do them in the proper order!
  - That is the order that you want to achieve the effect you want
  - The right-most matrix will be applied first.

OpenGL Transformations

- Ok. Now that we have the math of 2D homogenous transformations let's start thinking about how we can do it in OpenGL…
- For each object we should have a *model* transformation matrix.
- Then we could either
  - Apply it to each vertex on the client side and resend them to the server, or…
  - Just send the model to the server and let it do the computations.
- Which is better?

Example

- Rotate 30 degrees with a fixed point of (1.0,2.0);
- Approach:
    1. Translate (-1, -2)
    2. Rotate (30)
    3. Translate (1,2)
- For our OpenGL code we have `mat2`, `mat3` and `mat4` objects
- These structures store the data in column-major format so:

```
mat2 m(1,3,2,4); //odd?
mat2 m(vec2(1,2),vec2(3,4));
```

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

Example

- Rotate 30 degrees with a fixed point of (1.0,2.0);

```
mat3 t1(vec3(1,0,-1.0),vec3(0,1,-2.0),vec3(0,0,1));
float rad = theta*2*3.14/360;
mat3 r(   vec3(cos(rad),-sin(rad),0),
          vec3(sin(rad),cos(rad),1,0),
          vec3(0,0,1));
mat3 t2(vec3(1,0,1.0),vec3(0,1,2.0),vec3(0,0,1));
mat3 m = t2*r*t1;
```

- Remember, the last matrix specified is the first applied

Model Matrix

- Now we can either:
  - Apply this model matrix to each of the vertices in the OpenGL client application and then copy it to the GPU
  - Just pass this model matrix to the GPU and have it apply it to each vertex.

Model Matrix

Transpose

- To pass the model matrix to the GPU:
  - We must have a `uniform mat3` variable in our vertex shader.
  - Then we just pass it the matrix via:
    `glUniformMatrix3fv(model_matrix,1, GL_TRUE,m);`
    - 3fv stands for “3x3 floating point vector”
    - Second parameter says only pass one matrix
    - **Third parameter says to transpose the matrix when you send it. GLSL expects the data to be in row-major format**
- In the shader application we apply this model matrix to the `vec3` input vertex. However the `gl_Position` variable expects a `vec4` object.
- So we can do

```
gl_Position = vec4((model_view*vPosition).xy,0,1);
```

Passing the Model Matrix

Client

- Initialize:
  - `mat3 m = mat3(1.0);`
- In Display
  - `glUniformMatrix3fv(model_matrix,1, GL_TRUE,m);`

Vertex Shader

```
in  vec3 vPosition;
in  vec4 vColor;
out vec4 color;
uniform mat3 model_view;
void main()
{
    gl_Position = vec4((model_view*vPosition).xy,0,1);
    color = vColor;
}
```