Viewing I: Model Transformations

Matrix Representation of Transformations

- Let $\mathcal{A}_0$ and $\mathcal{A}_1$ be affine spaces.
  Let $\mathbf{T} : \mathcal{A}_0 \mapsto \mathcal{A}_1$ be an affine transformation.
  Let $F_0 = (\vec{i}_0, \vec{j}_0, \mathcal{O}_0)$ be a frame for $\mathcal{A}_0$.
  Let $F_1 = (\vec{i}_1, \vec{j}_1, \mathcal{O}_1)$ be a frame for $\mathcal{A}_1$.
- Let $P = x\vec{i}_0 + y\vec{j}_0 + \mathcal{O}_0$ be a point in $\mathcal{A}_0$.
  The *coordinates* of P relative to $\mathcal{A}_0$ are $(x, y, 1)$.

  This can also be represented in vector form as $P = \begin{bmatrix} \vec{i}_0 \ \vec{j}_0 \ \mathcal{O}_0 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$

- What are the coordinates $(x', y', 1)$ of $\mathbf{T}(P)$ relative to F_1?
  - An affine transformation is characterized by the image of a frame in the domain.

$$\begin{aligned} \mathbf{T}(P) &= \mathbf{T}(x\vec{i}_0 + y\vec{j}_0 + \mathcal{O}_0) \\ &= x\mathbf{T}(\vec{i}_0) + y\mathbf{T}(\vec{j}_0) + \mathbf{T}(\mathcal{O}_0) \end{aligned}$$

  - $\mathbf{T}(\vec{i}_0)$ must be a linear combination of $\vec{i}_1$ and $\vec{j}_1$, say $\mathbf{T}(\vec{i}_0) = t_{1,1}\vec{i}_1 + t_{2,1}\vec{j}_1$.
  - Likewise $\mathbf{T}(\vec{j}_0)$ must be a linear combination of $\vec{i}_1$ and $\vec{j}_1$, say $\mathbf{T}(\vec{j}_0) = t_{1,2}\vec{i}_1 + t_{2,2}\vec{j}_1$.
  - Finally $\mathbf{T}(\mathcal{O}_0)$ must be an affine combination of $\vec{i}_1$, $\vec{j}_1$, and $\mathcal{O}_1$, say $\mathbf{T}(\mathcal{O}_0) = t_{1,3}\vec{i}_1 + t_{2,3}\vec{j}_1 + \mathcal{O}_1$.

– Then by substitution we get

$$
\begin{aligned}
\mathbf{T}(P) &= x(t_{1,1}\vec{i}_1 + t_{2,1}\vec{j}_1) + y(t_{1,2}\vec{i}_1 + t_{2,2}\vec{j}_1) + t_{1,3}\vec{i}_1 + t_{2,3}\vec{j}_1 + \mathcal{O}_1 \\
&= \left[t_{1,1}\vec{i}_1 + t_{2,1}\vec{j}_1 \;\; t_{1,2}\vec{i}_1 + t_{2,2}\vec{j}_1 \;\; t_{1,3}\vec{i}_1 + t_{2,3}\vec{j}_1 + \mathcal{O}_1 \right] \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \\
&= \left[\vec{i}_1 \;\; \vec{j}_1 \;\; \mathcal{O}_1 \right] \begin{bmatrix} t_{1,1} & t_{1,2} & t_{1,3} \\ t_{2,1} & t_{2,2} & t_{2,3} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}
\end{aligned}
$$

Using $\mathbf{M}_T$ to denote the matrix, we see that $F_0 = F_1\mathbf{M}_T$

- Let $\mathbf{T}(P) = P' = x'\vec{i}_1 + y'\vec{j}_1 + \mathcal{O}_1$

In vector form this is

$$
\begin{aligned}
P' &= \left[\vec{i}_1\ \vec{j}_1\ \mathcal{O}_1\right] \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \\
&= \left[\vec{i}_1\ \vec{j}_1\ \mathcal{O}_1\right] \begin{bmatrix} t_{1,1} & t_{1,2} & t_{1,3} \\ t_{2,1} & t_{2,2} & t_{2,3} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}
\end{aligned}
$$

So we see that

$$
\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} t_{1,1} & t_{1,2} & t_{1,3} \\ t_{2,1} & t_{2,2} & t_{2,3} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}
$$

We can write this in shorthand – $\mathbf{p}' = \mathbf{M}_T \mathbf{p}$

- $\mathbf{M}_T$ is the *matrix representation* of **T**
  - The first column of $\mathbf{M}_T$ represents $\mathbf{T}(\vec{i}_0)$
  - The second column of $\mathbf{M}_T$ represents $\mathbf{T}(\vec{j}_0)$
  - The third column of $\mathbf{M}_T$ represents $\mathbf{T}(\mathcal{O}_0)$

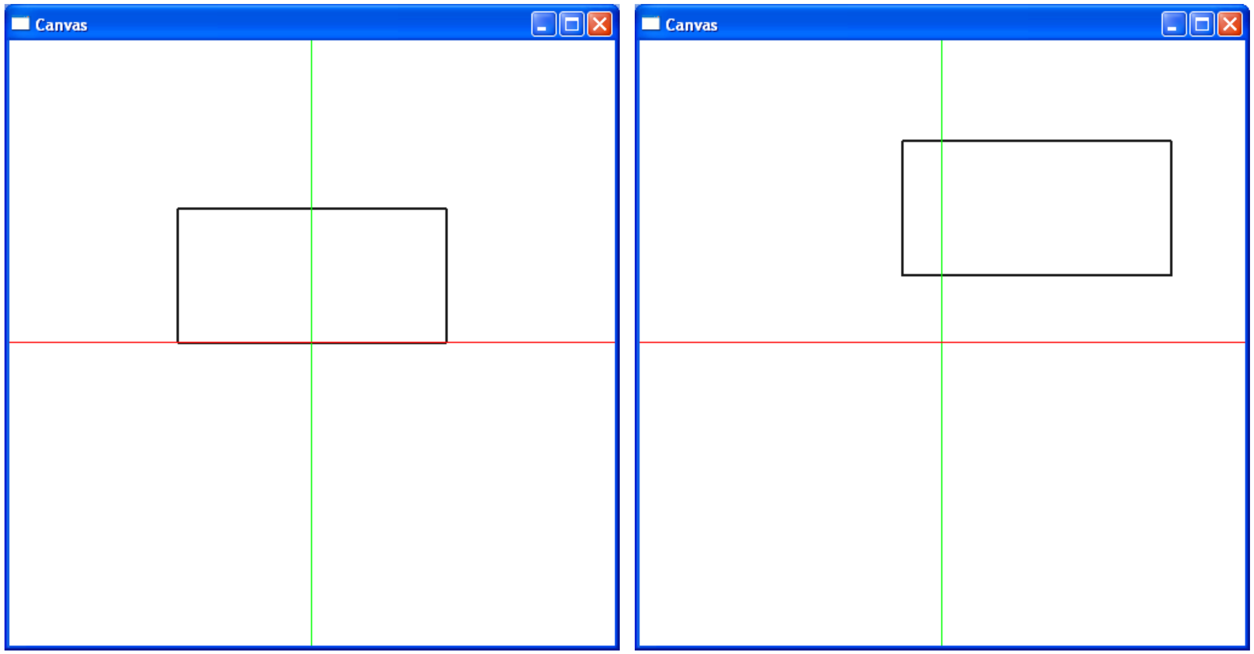
- *Translation*
  - Points are transformed as $\left[x'\ y'\ 1\right]^T = [x\ y\ 1]^T + [\Delta x\ \Delta y\ 0]^T$.
  - Vectors don't change.
  - Thus translation is affine but not linear.
    If it were linear, we would have $\mathbf{T}(P+Q) = \mathbf{T}(P) + \mathbf{T}(Q)$, but point addition is undefined.
  - Translation can be applied to sums of vectors and vector-point sums.
  - Matrix formulation:

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

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

  - Shorthand for the above matrix: $T(\Delta x, \Delta y)$

- *Example*




```
glTranslatef(.7, .5, 0);
glBegin(GL_LINE_LOOP);
    glVertex2f(-1, 0);
    glVertex2f(1, 0);
    glVertex2f(1,1);
    glVertex2f(-1,1);
glEnd();
```

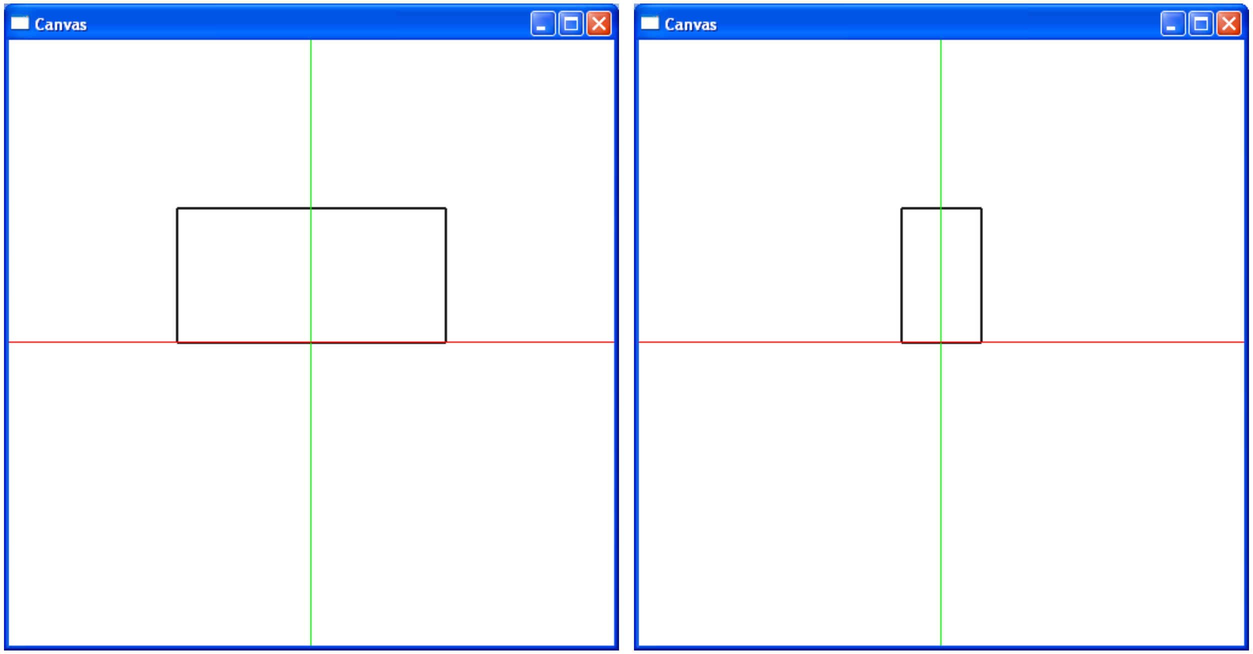
- *Scale*
  - Linear transform — applies equally to points and vectors
  - Points transform as $\left[x' \ y' \ 1\right]^T = [xS_x \ yS_y \ 1]^T$.
  - Vectors transform as $\left[x' \ y' \ 0\right]^T = [xS_x \ yS_y \ 0]^T$.
  - Matrix formulation:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} xS_x \\ yS_y \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ 0 \end{bmatrix} = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 0 \end{bmatrix} = \begin{bmatrix} xS_x \\ yS_y \\ 0 \end{bmatrix}$$

  - Shorthand for the above matrix: $S(S_x, S_y)$
  - Note that this is *origin sensitive*.
  - How do you do reflections?

- *Example*




```
glScalef(0.3, 1, 1);
glBegin(GL_LINE_LOOP);
    glVertex2f(-1, 0);
    glVertex2f(1, 0);
    glVertex2f(1,1);
    glVertex2f(-1,1);
glEnd();
```

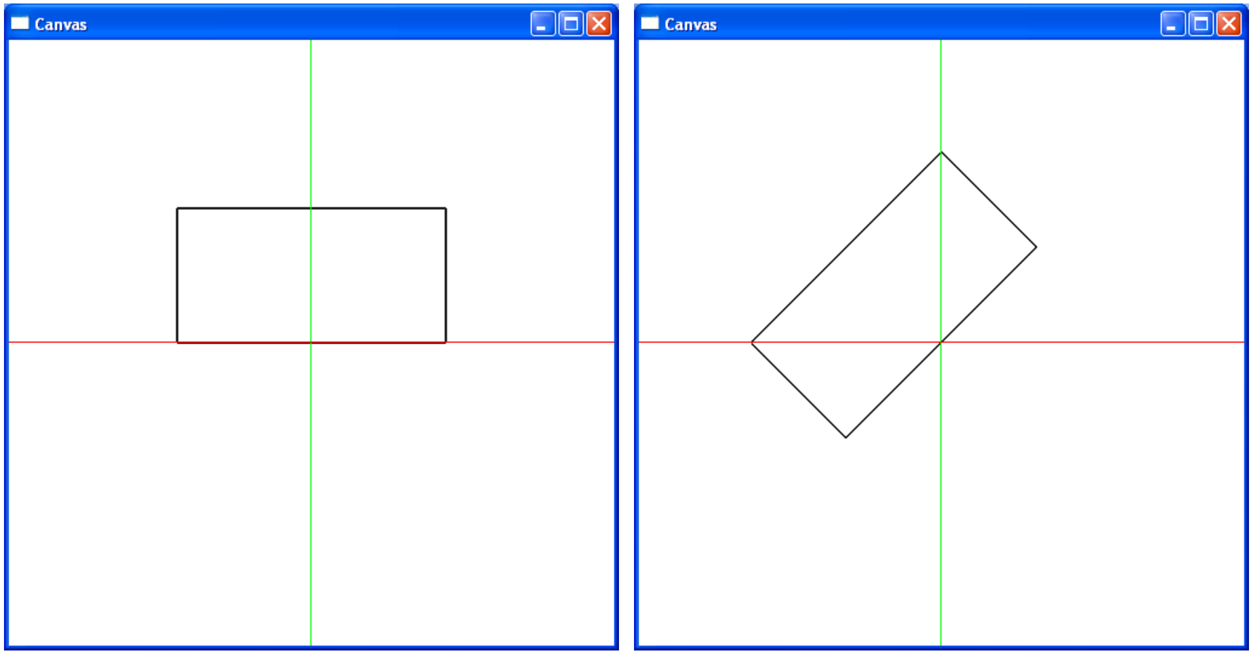
- *Rotate*
  - Linear transform — applies equally to points and vectors
  - Points transform as $\left[x'\ y'\ 1\right]^T = [x\cos(\theta) - y\sin(\theta)\ x\sin(\theta) + y\cos(\theta)\ 1]^T$.
  - Vectors transform as $\left[x'\ y'\ 0\right]^T = [x\cos(\theta) - y\sin(\theta)\ x\sin(\theta) + y\cos(\theta)\ 0]^T$.
  - Matrix formulation:

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

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

  - Shorthand for the above matrix: $R(\theta)$
  - Note that this is *origin sensitive*.

- *Example*




```
glRotatef(45, 0, 0, 0, 1);
glBegin(GL_LINE_LOOP);
    glVertex2f(-1, 0);
    glVertex2f(1, 0);
    glVertex2f(1,1);
    glVertex2f(-1,1);
glEnd();
```

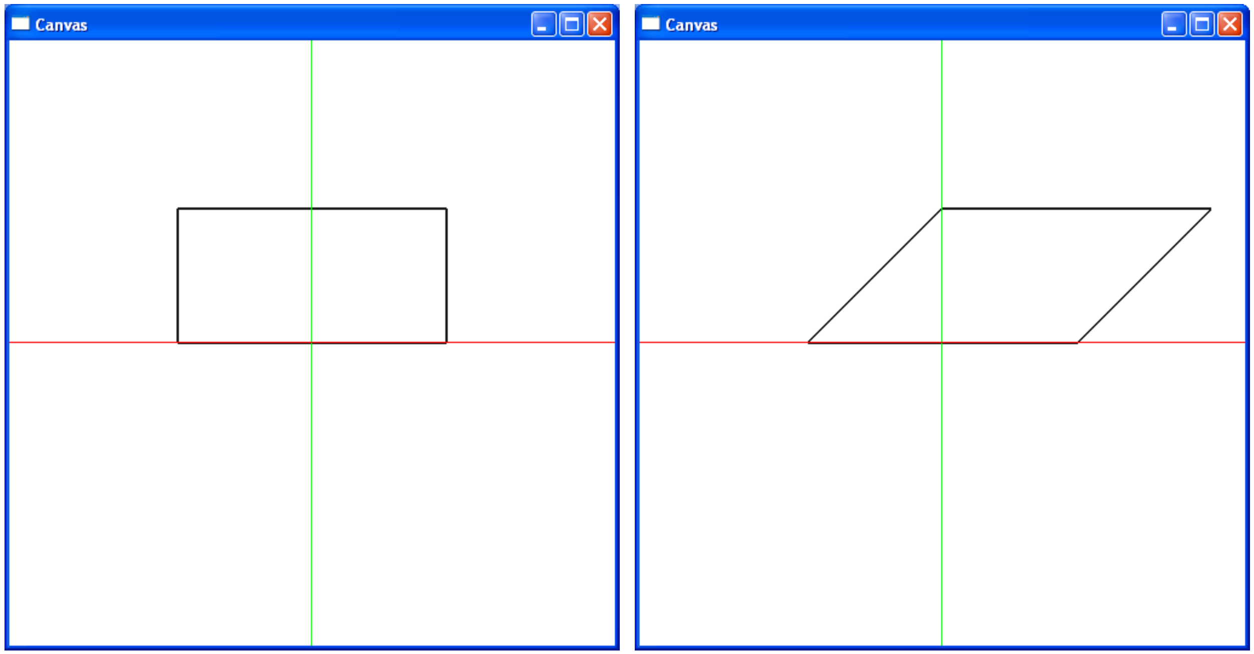
- *Shear*
  - Linear transform — applies equally to points and vectors
  - Points transform as $\left[x'\ y'\ 1\right]^T = [x + \alpha y,\ y + \beta x,\ 1]^T$.
  - Vectors transform as $\left[x'\ y'\ 0\right]^T = [x + \alpha y,\ y + \beta x,\ 0]^T$.
  - Matrix formulation:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & \alpha & 0 \\ \beta & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + \alpha y \\ y + \beta x \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & \alpha & 0 \\ \beta & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 0 \end{bmatrix} = \begin{bmatrix} x + \alpha y \\ y + \beta x \\ 0 \end{bmatrix}$$

  - Shorthand for the above matrix: $Sh(\alpha, \beta)$

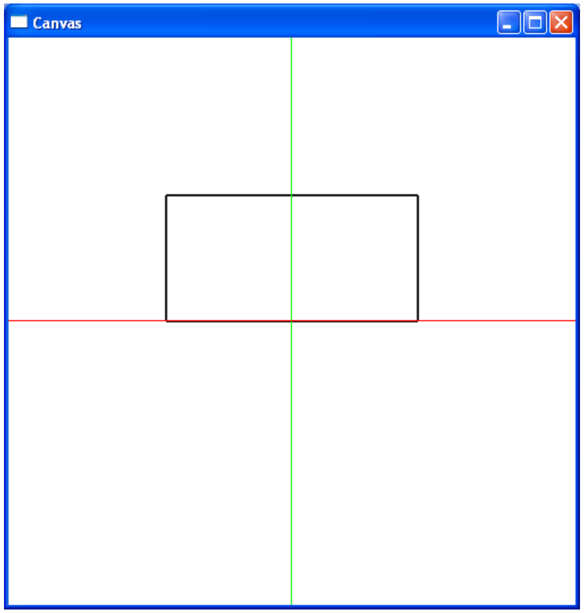
- *Example*

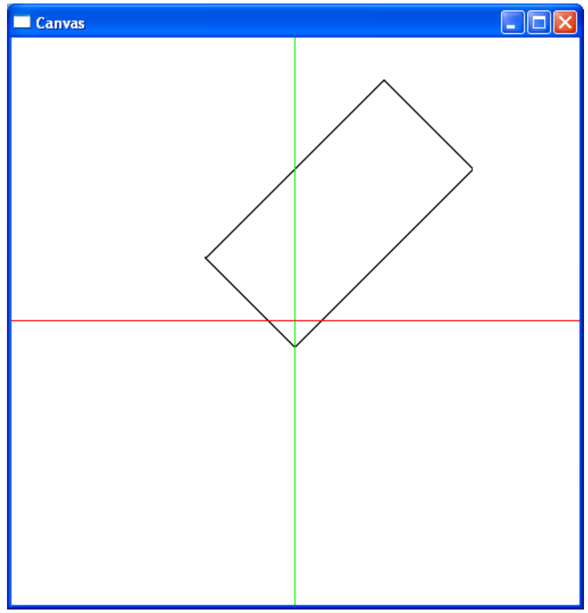


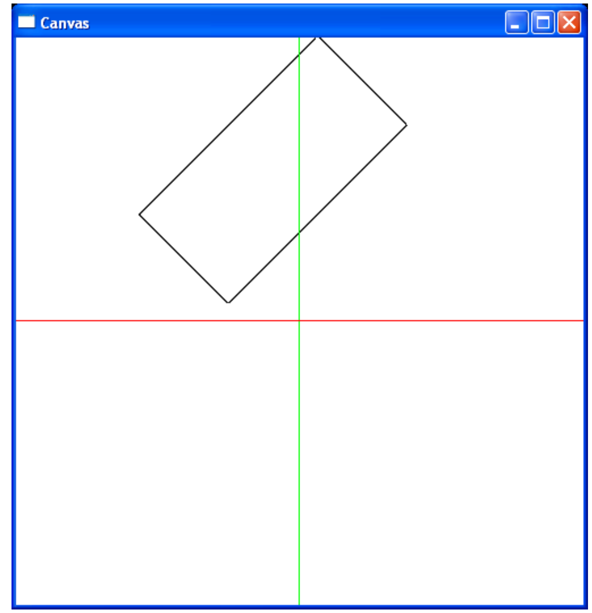

```
float ShearMatrix[] = {
          1, 1, 0, 0,
          0, 1, 0, 0,
          0, 0, 1, 0,
          0, 0, 0, 1  };
Traspose(ShearMatrix);
glMultMatrixf(ShearMatrix);
```

- Composition of Transformations
  - Now we have some basic transformations, how do we create and represent arbitrary affine transformations?
  - We can derive an arbitrary affine transform as a sequence of basic transformations, then compose the transformations
  - Example — scaling about an arbitrary point $[x_c\ y_c\ 1]^T$
    1. Translate $[x_c\ y_c\ 1]^T$ to $[0\ 0\ 1]$ $(T(-x_c, -y_c))$
    2. Scale $\left[x'\ y'\ 1\right]^T = S(S_x, S_y)\,[x\ y\ 1]^T$
    3. Translate $[0\ 0\ 1]^T$ back to $[x_c\ y_c\ 1]$ $(T(x_c, y_c))$
  - The sequence of transformation steps is $T(-x_c, -y_c) \circ S(S_x, S_y) \circ T(x_c, y_c)$

– *Example*








```
glTranslate(.7, .5, 0);
glRotatef(45, 0, 0, 0, 1);
glBegin(GL_LINE_LOOP);
    glVertex2f(-1, 0);
    glVertex2f(1, 0);
    glVertex2f(1,1);
    glVertex2f(-1,1);
glEnd();
```

```
glRotatef(45, 0, 0, 0, 1);
glTranslate(.7, .5, 0);
glBegin(GL_LINE_LOOP);
    glVertex2f(-1, 0);
    glVertex2f(1, 0);
    glVertex2f(1,1);
    glVertex2f(-1,1);
glEnd();
```

– In matrix form this is

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_c \\ 0 & 1 & y_c \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_c \\ 0 & 1 & -y_c \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} S_x & 0 & x_c(1 - S_x) \\ 0 & S_y & y_c(1 - S_y) \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

– Note that the matrices are arranged from *right to left* in the order of the steps.

– The order is important (why)?

- Three Dimensional Transformations
  - A point is $\mathbf{p} = [x\ y\ z\ 1]$, a vector $\vec{v} = [x\ y\ z\ 0]$
  - Translation:

$$T(\Delta x, \Delta y, \Delta z) = \begin{bmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

  - Scale:

$$S(S_x, S_y, S_z) = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

  - Rotation:

$$R_z(\Theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

  More on 3D Rotations later, especially using Quaternions!

OpenGL Transformation Matrices

There are three matrices that are part of the OpenGl pipeline, and all are manipulated by a common set of functions. To select the matrix type on which operations apply use glMatrixMode function. For example,

```
glMatrixMode(GL_MODELVIEW); or  glMatrixMode(GL_PROJECTION)
```

- The matrix applied to all primitives is the product of the ModelView matrix and the Projection matrix.
- Matrix is loaded with function

  ```
  glLoadMatrixf(pointer_to_matrix)
  ```

- Matrix is altered with function

  ```
  glMultMatrixf(pointer_to_matrix)
  ```

- Translation is provided with function

  ```
  glTranslatef(dx,dy,dz)
  ```

- Rotation is provided with function

  ```
  glRotatef(angle,vx,vy,vz)
  ```

- Scaling is provided with function

```
glScalef(sx,sy,sz)
```

- All three transformations alter the selected matrix by postmultiplication.

Order of Applying Transformations The rule in OpenGL: The transformation specified last is the one applied first.

Consider the example sequence to form the required matrix for a 45-degree rotation about a vector (1,2,3). The object frame's origin is (4,5,6) and that is its center of rotation. The sequence is to move the object's frame to the origin (0,0,0), rotating about the origin, and finally moving the rotated object back to its original location.

```
glMatrixModel(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(4.0,5.0,6.0);
glRotatef(45.0, 1.0,2.0,3.0);
glTranslatef(-4.0,-5.0,-6.0);
```

Projections as an Example of Projective Transformations

Perspective Projection

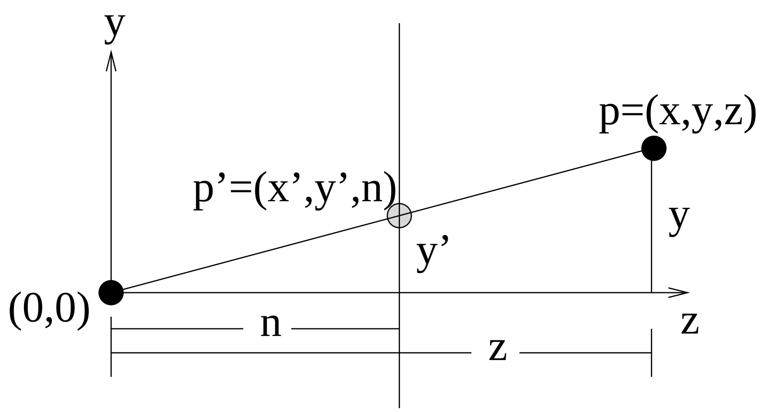
- Identify all points with a line through the eyepoint.
- Slide lines with viewing plane, take intersection point as projection.
- This is *not* an affine transformation, but a *projective transformation.*

Projective Transformations:

- Angles are not preserved.
- Distances are not preserved.
- Ratios of distances are not preserved.
- Affine combinations are not preserved.
- Straight lines are mapped to straight lines.
- Incidence relationships are preserved in a general way.
- *Cross ratios* are preserved.

Perspective Map

- Given a point P, we want to find its projection P'.



Projection plane, z = n

- Similar triangles: $P' = (xn/z, n)$
- In 3D, $(x', y', z') \mapsto (xn/z, yn/z, n)$
- Have identified all points on a line through the origin with a point in the projection plane.
- Thus, $(x, y, z) \equiv (kx, ky, kz)$, $k \neq 0$.
- These are known as homogeneous coordinates.
- If we have solids, or colored lines, then we need to know "which one is in front."
- This map loses all z information, so it is inadequate.

Why Map Z

- 3D $\mapsto$ 2D projections map all z to same value.
- Need z to determine occlusion, so a 3D to 2D projective transformation doesn't work.
- Further, we want 3D lines to map to 3D lines (this is useful in hidden surface removal).
- The mapping $(x, y, z, 1) \mapsto (xn/z, yn/z, n, 1)$ maps lines to lines, but loses all depth information.
- We could use
$$(x, y, z, 1) \mapsto (xn/z, yn/z, z, 1)$$
Thus, if we map the endpoints of a line segment, these end points will have the same relative depths after this mapping.
BUT: It fails to map lines to lines
- The map
$$(x, y, z, 1) \mapsto \left(\frac{xn}{z}, \frac{yn}{z}, \frac{zf + zn - 2fn}{z(f-n)}, 1\right)$$
does map lines to lines, *and* it preserves depth information.

Mapping Z

- It's clear how x and y map. How about z?

$$z \mapsto \frac{zf + zn - 2fn}{z(f-n)} = P(z)$$

- We know $P(f) = 1$ and $P(n) = -1$. What maps to 0?

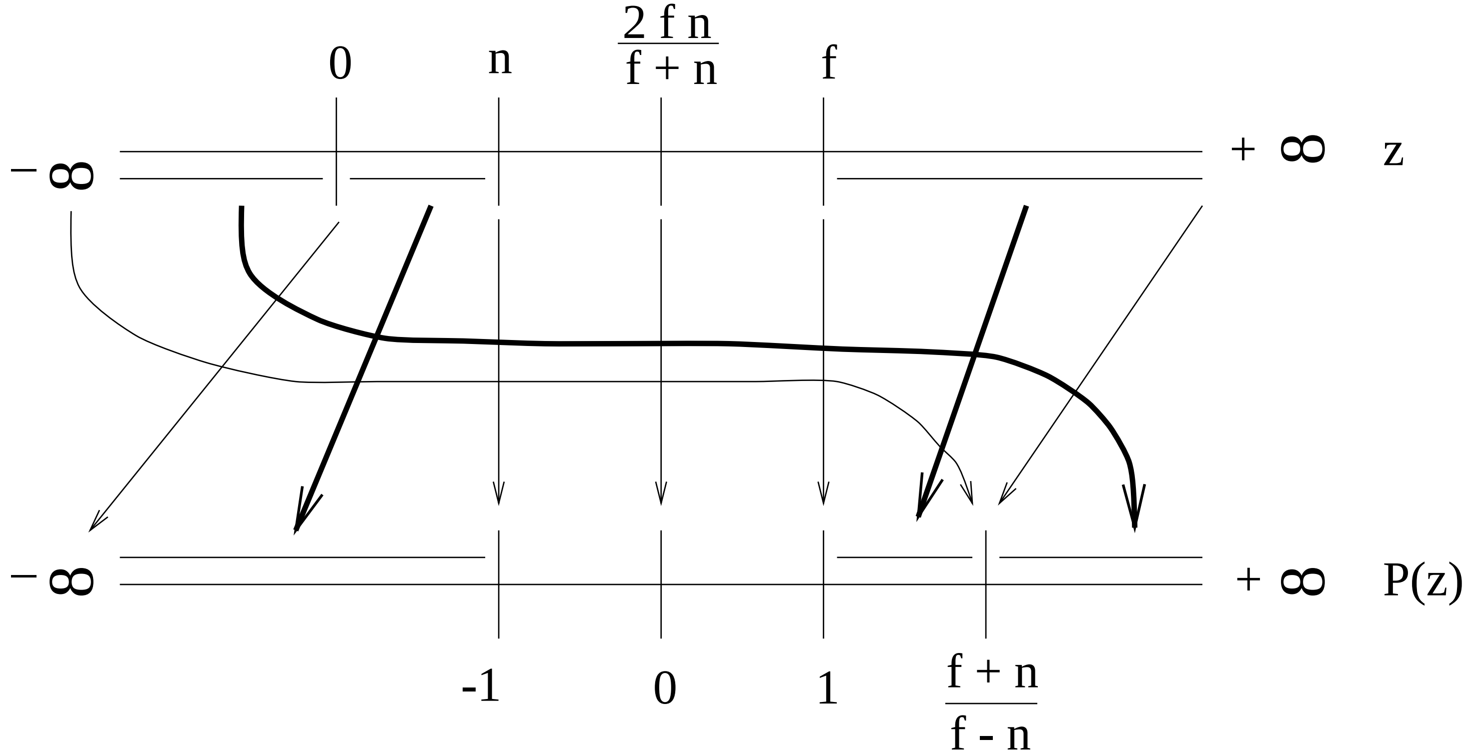
$$\begin{aligned} P(z) &= 0 \\ \Rightarrow \qquad \frac{zf + zn - 2fn}{z(f-n)} &= 0 \\ \Rightarrow \qquad z &= \frac{2fn}{f+n} \end{aligned}$$

Note that $f^2 + 2f > 2fn/(f+n) > fn + n^2$ so

$$f > \frac{2fn}{f+n} > n$$

- What happens as map z to 0 or to infinity?

$$
\begin{aligned}
\lim_{z\to 0^+} P(z) &= \frac{-2fn}{z(f-n)} \\
&= -\infty \\
\lim_{z\to 0^-} P(z) &= \frac{-2fn}{z(f-n)} \\
&= +\infty \\
\lim_{z\to +\infty} P(z) &= \frac{z(f+n)}{z(f-n)} \\
&= \frac{f+n}{f-n} \\
\lim_{z\to -\infty} P(z) &= \frac{z(f+n)}{z(f-n)} \\
&= \frac{f+n}{f-n}
\end{aligned}
$$


0
n
2 f n / f + n
f
− ∞
+ ∞
z
− ∞
+ ∞
P(z)
-1
0
1
f + n / f - n

- What happens if we vary f and n?

$$\begin{aligned}\lim_{f \to n} P(z) &= \frac{z(f+n) - 2fn}{z(f-n)} \\ &= \frac{(2zn - 2n^2)}{z \cdot 0}\end{aligned}$$

which is not surprising, since we're trying to map a single point to a line segment.

$$\begin{aligned}\lim_{f \to \infty} P(z) &= \frac{zf - 2fn}{zf} \\ &= \frac{z - 2n}{z}\end{aligned}$$

- But note that this means we are mapping an infinite region to [0,1] and we will effectively get a far plane due to floating point precision,

$$\begin{aligned}\lim_{n \to 0} P(z) &= \frac{zf}{zf} \\ &= 1\end{aligned}$$

i.e., the entire map becomes constant (again, we are mapping a point to an interval).

- Consider what happens as f and n move away from each other.
  - We are interested in the size of the regions $[n, 2fn/(f+n)]$ and $[2fn/(f+n), f]$.
  - When f is large compared to n, we have

$$\frac{2fn}{f+n} \doteq 2n$$

So

$$\frac{2fn}{f+n} - n \doteq n$$

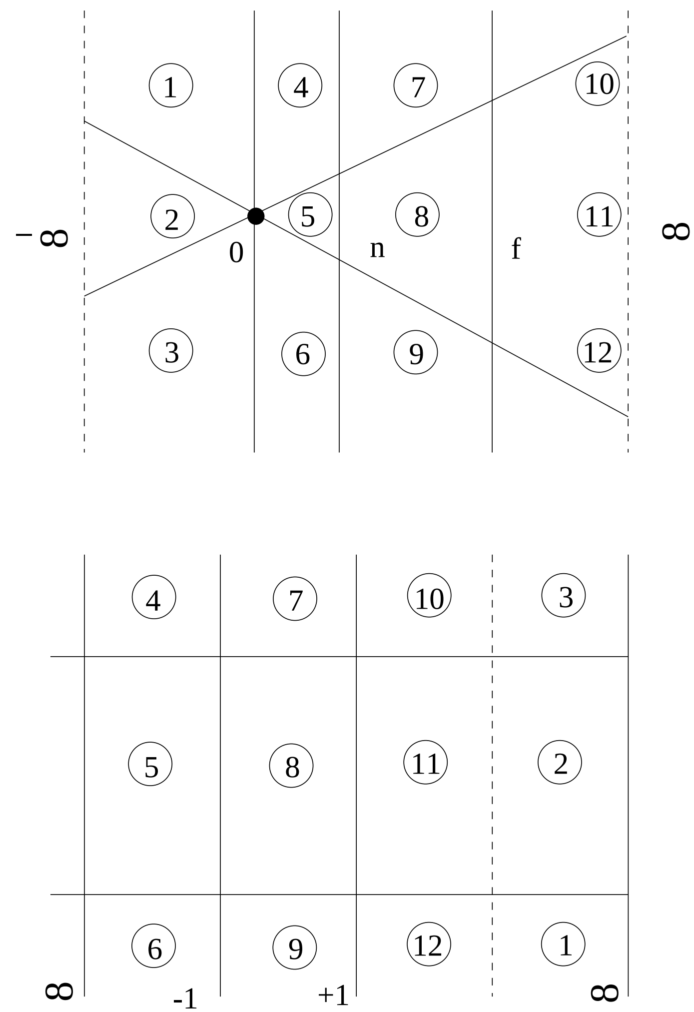
and

$$f - \frac{2fn}{f+n} \doteq f - 2n$$

But both intervals are mapped to a regions of size 1.
  - Thus, as we move the clipping planes away from one another, the far interval is compressed more than the near one. With floating point arithmetic, this means we'll lose precision.

– In the extreme case, think about what happens as we move f to infinity: we compress an infinite region to an finite one.
– Therefore, we try to place our clipping planes as close to one another as we can.

Region Mapping



Reading Assignment and News

Chapter 4 pages 200 - 212, of Recommended Text.

Please also track the News section of the Course Web Pages for the most recent Announcements related to this course.

(http://www.cs.utexas.edu/users/bajaj/graphics25/cs354/)