Transforms: More Than Meets the Eye

1 Overview

- A transformation τ is any function that maps points to points:

$$\tau(x, y) = (x^2 - 5, y^3 + x - \pi)$$
$$\tau(x, y) = (3x + y + 2, 15)$$
$$\tau(x, y, z) = (\ln|z|, y, \frac{x^3 \sin z}{12})$$

- The challenge in computer graphics is: how do we implement transformations in a generalized way, but without becoming insanely inefficient.

- The goal is for you to know the answer to that question by the time you reach the end of this handout. You *will* read this to the very end, right?

2 A Very Fine Transformation

- The *affine transformation* is a special type of transform, and in computer graphics it easily wins the "Transform Most Likely to be Used Over and Over Again" award year after year.

- Intuitively, an affine transformation is any transformation for which straight lines remain straight, and parallel lines remain parallel.

- Mathematically, this a transformation τ is affine if and only if it can be written as:

$$\tau(x, y) = (ax + by + c, dx + ey + f) \tag{1}$$

  where a, b, c, d, e, f are scalar constants and $ae - bd \neq 0$.

  Some of you may recognize that $ae - bd$ is the *determinant* of the 2-dimensional matrix formed by the coefficients to x and y, i.e.

$$\begin{vmatrix} a & d \\ b & e \end{vmatrix}$$

- Thus, in 3D, (1) extrapolates to:

$$\tau(x, y, z) = (ax + by + cz + d, ex + fy + gz + h, ix + jy + kz + l) \tag{2}$$

where $a, b, c, d, e, f, g, h, i, j, k, l$ are scalar constants and $\begin{vmatrix} a & e & i \\ b & f & j \\ c & g & k \end{vmatrix} \neq 0$.

3 Affine Transformation Properties

- Since affine transforms preserve the straightness and parallelism of lines, then to perform an affine transform on a polygon, it is sufficient to transform its vertices.
- It can also be shown that affine transforms preserve relative or proportional distances — i.e. the affine transformation of a line's midpoint is the midpoint of the affine transformation of its endpoints; the affine transformation of a polygon's centroid is the centroid of the affine transformation of the overall polygon.
- Affine transformations have closure: the composition of two affine transformations is also affine.
- *But* transform composition is generally *not* commutative — not too hard to see intuitively.
- The inverse of an affine transformation (e.g. the transformation τ^{-1} such that $\tau^{-1}(\tau(x, y)) = (x, y)$) is also affine. Given τ as expressed in (1), τ^{-1} can be derived analytically as:

$$\tau^{-1}(x, y) = (\frac{e}{ae - bd}x + \frac{-b}{ae - bd}y + \frac{bf - ce}{ae - bd}, \frac{-d}{ae - bd}x + \frac{a}{ae - bd}y + \frac{cd - af}{ae - bd}) \tag{3}$$

Note how the inverse shows a quantitative reason for requiring $ae - bd \neq 0$.

4 T^4: The Top Three Transforms

- The three most frequently used transformations are all affine: translation, scaling, and rotation. For simplicity, we talk about them in 2D. Extrapolation to 3D is straightforward.
- Translation moves an object across the space. It can be written as $T_{\langle dx, dy \rangle}$ where $\langle dx, dy \rangle$ is the vector by which the object is to be moved:

$$T_{\langle dx, dy \rangle}(x, y) = (x + dx, y + dy) \tag{4}$$

- Scaling changes the relative size of an object. It can be written as S_{s_x,s_y} where s_x and s_y are the scalars by which to enlarge or reduce x and y, respectively. (*Quick aside* — what values of s_x and s_y determine whether the scaling will enlarge or reduce?)

$$S_{s_x,s_y}(x, y) = (s_x x, s_y y) \tag{5}$$

- Rotation in 2D rotates points about the origin $(0, 0)$. It can be written as R_θ where θ is the angle of rotation:

$$R_\theta(x, y) = (x \cos\theta - y \sin\theta, x \sin\theta + y \cos\theta) \tag{6}$$

  This is basic trigonometry — I *will* ask you, sometime, how this is derived.

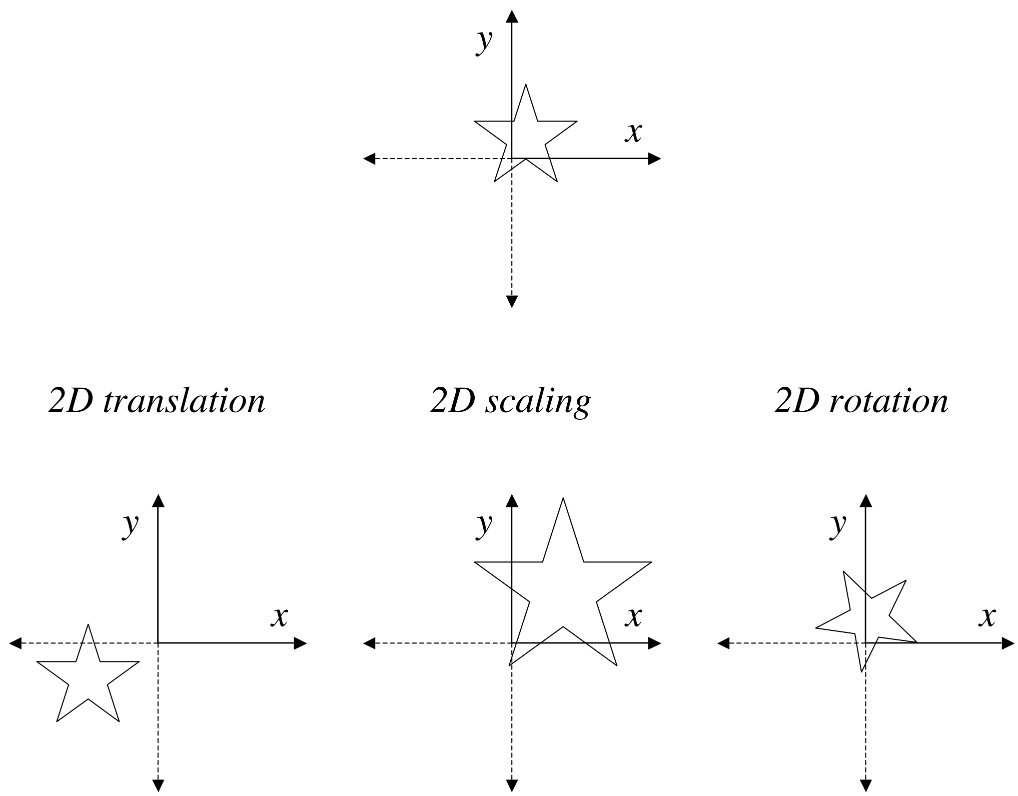


Figure 1: The top three transforms: translation, scaling, and rotation.

5 Other Affine Transforms

- There's tons more. Here's a quick rundown...
  - Shear along the x axis: $H_h^x(x, y) = (x + hy, y)$

- Shear along the y axis: $H_g^y(x, y) = (x, gx + y)$
- Shear along both axes: $H_{g,h}^{x,y}(x, y) = (x + hy, gx + y)$
- Reflect across the x axis: $F^x(x, y) = (x, -y)$
- Reflect across the y axis: $F^y(x, y) = (-x, y)$
- Reflect across the origin: $F^0(x, y) = (-x, -y)$
- Reflect across the line $L(\alpha) = (\alpha, \alpha)$: $F^{L(\alpha)=(\alpha,\alpha)}(x, y) = (y, x)$

- Sanity check: do all of these follow the formal definition of an affine transform, as given in (1)?

6 Revisiting Affine Transform Properties

- Verify these for yourselves:
  - These transforms are supposed to be affine — so they must be writable as sums of coefficients. If so, then what are these coefficients?
  - Do they preserve straight and parallel lines?
  - If you compose the transformations, does the composition preserve straight and parallel lines?
  - Transformation composition is not supposed to be generally commutative — can you think of specific combinations that illustrate this?

7 Computerizing Affine Transforms

- Okay, great, we now have all of these neat ways to move, scale, rotate, and otherwise manipulate any shape and/or vertex. How do we make it practical for a computer to do this? Remember, we want to keep it:
  - General, so that we can do any affine transform that we can think of, and
  - Efficient, so we can do millions or billions of these transformations per second
- Think think think. . .

— cue Jeopardy music —

7.1 The Insight

- Once upon a time, somewhere, somehow, it was observed that the formal definition of the affine version — 2D version in (1), 3D version in (2) — looks an awful lot like *matrix multiplication*:

$$\tau(x, y) = (ax + by + c, dx + ey + f) = \begin{bmatrix} ax + by + c \\ dx + ey + f \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{7}$$

- No. . . wait. . . that's not entirely right. Proper matrix multiplication requires that the multiplier's number of rows equals the multiplicands number of columns. Besides, as written above, the constants c and f get lost. So we need to add a third row. . .

$$\tau(x, y) = (ax + by + c, dx + ey + f) = \begin{bmatrix} ax + by + c \\ dx + ey + f \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{8}$$

- Aha. . . if we add a third row of "1" for the column matrix and "0 0 1" in the transformation matrix, we have fully expressed our 2D affine transform as a product of a square matrix consisting only of the constant coefficients and a column matrix consisting only of the "input" point. More generally, we can represent an n-dimensional transform as an $(n + 1) \times (n + 1)$ matrix! For instance, extrapolating to 3D:

$$\tau(x, y, z) = \begin{bmatrix} ax + by + cz + d \\ ex + fy + gz + h \\ ix + jy + kz + l \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{9}$$

- Furthermore, we observe that if the bottom row of the column matrix is not 1, e.g. $\begin{bmatrix} x \\ y \\ h \end{bmatrix}$, "dividing through" by h yields $\begin{bmatrix} x/h \\ y/h \\ 1 \end{bmatrix}$. Thus, $\begin{bmatrix} x \\ y \\ h \end{bmatrix}$ is another way to represent the point $(x/h, y/h)$.

- This notation for expressing a point is called *homogeneous coordinates*. "Homogeneous" roughly means "same kind" — it reflects the way this notation can represent vertices in a consistent manner, such that a single algorithm on them can calculate all possible affine transformations on that vertex.

- We can now take the affine transforms that we have defined and express them uniformly as 3×3 matrices:

$$T_{\langle dx,dy \rangle}(x, y) = \begin{bmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{10}$$

$$S_{s_x,s_y}(x,y) = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{11}$$

$$R_\theta(x,y) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{12}$$

- The other transforms that we have already given follow suit.

$$H_h^x(x,y) = \begin{bmatrix} 1 & h & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$H_g^y(x,y) = \begin{bmatrix} 1 & 0 & 0 \\ g & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$H_{g,h}^{x,y}(x,y) = \begin{bmatrix} 1 & h & 0 \\ g & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$F^x(x,y) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$F^y(x,y) = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$F^0(x,y) = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$F^{L(\alpha)=(\alpha,\alpha)}(x,y) = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

7.2 Why is This So. Freaking. Cool?!?

- Mapping geometric transformations to matrix multiplication results in a chain reaction of conclusions that are all but responsible for facilitating their practical implementation in computer graphics:
  - Matrix multiplication represents the generalized formal definition of an affine transform. Thus, *all* affine transforms can be expressed through a square matrix.
  - Matrix multiplication is associative. Thus, the composition of transforms τ_1 and τ_2 — in other words, $\tau_1(\tau_2(x,y))$ — is the same as multiplying the matrices represented by τ_1 and τ_2:

$$\tau_1(\tau_2(x,y)) = \begin{bmatrix} a_1 & b_1 & c_1 \\ d_1 & e_1 & f_1 \\ 0 & 0 & 1 \end{bmatrix} \left(\begin{bmatrix} a_2 & b_2 & c_2 \\ d_2 & e_2 & f_2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \right)$$

$$= \left(\begin{bmatrix} a_1 & b_1 & c_1 \\ d_1 & e_1 & f_1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_2 & b_2 & c_2 \\ d_2 & e_2 & f_2 \\ 0 & 0 & 1 \end{bmatrix} \right) \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Has it hit you yet? Let's put it bluntly: *the composition of any arbitrary number of transforms τ can be expressed as a single matrix* — the cost of transforming a vertex is *linear* with respect to the number of vertices to transform, regardless of the complexity of a transformation!

– Let's quantify that. Let's say that you wish to transform n 2-dimensional points P_1 to P_n using k composed transforms τ_1 to τ_k. If you sequentially transformed P_i by composition, i.e. $\tau_k(\tau_{k-1}(\dots \tau_1(P_i)))$ that results in $9nk$ multiplications and $6nk$ additions, since an individual transformation involves 9 multiplications and 6 additions.

– However, if you multiplied the transformation matrices first — which we know we can do because matrix multiplication is associative — the initial multiplication results in $27(k-1)$ multiplications and $18(k-1)$ additions since we are multiplying k 3×3 matrices. Then, to transform n points, we perform $9n$ more multiplications and $6n$ more additions:

$$9nk \text{ vs. } 9n + 27(k-1) \text{ multiplications}$$

$$6nk \text{ vs. } 6n + 18(k-1) \text{ additions}$$

– Think about real world use — when rendering 3D models, $n > k$ by a very large margin. So, in the end, the cost of transforming 10 vertices vs. 1,000,000 vertices is pretty much linear relative to n — and this bodes very well for a computerized implementation.

7.3 How Does 3D Change Things?

- Everything we've said about 2D transformations applies to 3D. Just add an axis...

$$T_{\langle dx,dy,dz \rangle} = \begin{bmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{13}$$

$$S_{s_x,s_y,s_z} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{14}$$

- Rotation, though, is a bit more complicated — in three dimensions, rotation can occur about any of the three axes. They are all fairly easy to derive though:

$$R_\theta^x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{15}$$

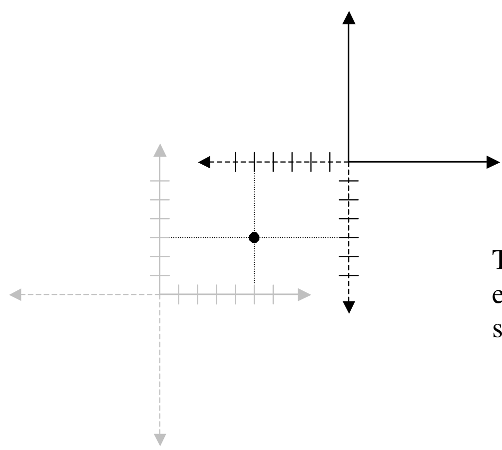
$$R_\theta^y = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{16}$$

$$R_\theta^z = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{17}$$

- How about rotation along an arbitrary axis? This is spelled out in the red book. Implementing it is a fine test of your understanding of these things.

8 A Parting Shot

- We have been discussing transforms in relation to transforming points or vertices. Note how the converse view applies — instead of transforming vertices, we are transforming the *coordinate system*.
- We saw a hint of this already when deriving rotation about an arbitrary axis.
- To convert a vertex transform to its corresponding coordinate system transform, we simply take its inverse — check out Figure 2.
- Philosophically, this type of relationship between vertex and axis transforms is frequently referred to as *duality*.
- Alright, that's all about transforms. Now go play.



Translating a point from (5, 3) to (–5, –4) is equivalent to translating its coordinate system from (0, 0) to (10, 7).

Rotating the point (3, 4) by around –53.13 degrees is equivalent to rotating the coordinate system by 53.13 degrees.

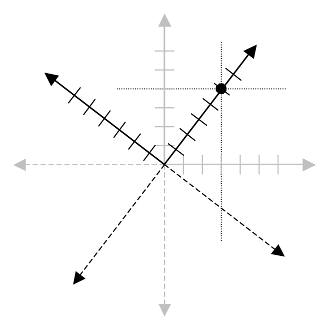


Figure 2: Transforming points vs. transforming axes.