CMSC 425: Lecture 6
Affine Transformations

Affine Transformations: So far we have been stepping through the basic elements of geometric programming. We have discussed points, vectors, and their operations, and coordinate frames and how to change the representation of points and vectors from one frame to another. Our next topic involves how to map points from one place to another. Suppose you want to draw an animation of a spinning ball. How would you define the function that maps each point on the ball to its position rotated through some given angle?

We will consider a limited, but interesting class of transformations, called *affine transformations*. These include (among others) the following transformations of space: translations, rotations, uniform and nonuniform scalings (stretching the axes by some constant scale factor), reflections (flipping objects about a line) and shearings (which deform squares into parallelograms). They are illustrated in Fig. 1.

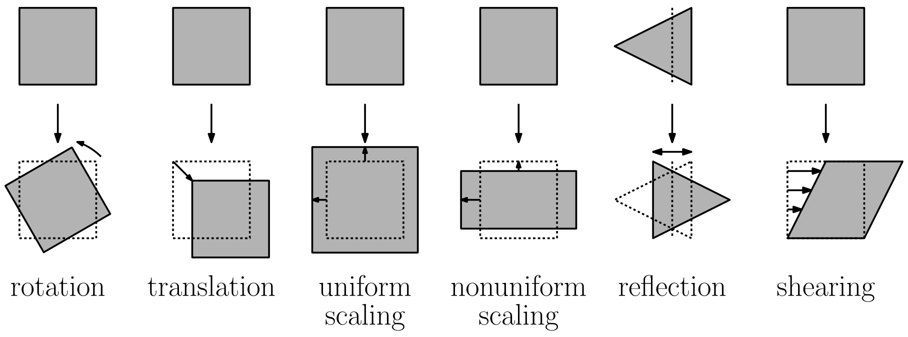


Fig. 1: Examples of affine transformations.

These transformations all have a number of things in common. For example, they all map lines to lines. Note that some (translation, rotation, reflection) preserve the lengths of line segments and the angles between segments. These are called *rigid transformations*. Others (like uniform scaling) preserve angles but not lengths. Still others (like nonuniform scaling and shearing) do not preserve angles or lengths.

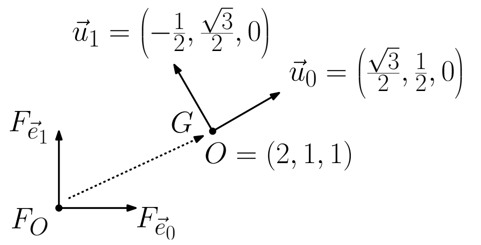
Formal Definition: Formally, an *affine transformation* is a mapping from one affine space to another (which may be, and in fact usually is, the same space) that preserves affine combinations. For example, this implies that given any affine transformation T and two points p and q, and any scalar α,

$$r = (1-\alpha)p + \alpha q \qquad \Rightarrow \qquad T(r) = (1-\alpha)T(p) + \alpha T(q).$$

For example, if r is the midpoint of segment $\overline{pq}$, then $T(r)$ is the midpoint of the transformed line segment $\overline{T(p)T(q)}$.

Matrix Representations of Affine Transformations: The above definition is rather abstract. It is possible to present any affine transformation T in d-dimensional space as a $(d+1)\times(d+1)$ matrix. For example, suppose that we have a 2-dimensional frame F consisting of an origin point O and basis vectors $\vec{e}_0$ through $\vec{e}_{d-1}$. To express T in the form of a matrix, we determine where each of these frame components is mapped, and then generate a matrix whose first d columns are the images of the basis vectors and whose last component is the image of the origin point. (It follows, therefore, that the last row of such a matrix must be $(0, \ldots, 0, 1)$, because the basis vectors must map to vectors, which must end in 0 according to the rules of affine homogeneous coordinates, and the origin point maps to a point, which must end in 1 by these same rules.)

For example, consider the affine transformation (in 2-dimensional space) shown in Fig. 2, which rotates by 30° degrees about the origin and translates 2 units to the right and 1 unit up. This transformation maps the origin O to the point O with homogeneous coordinates $(2, 1, 1)$, the x-axis is mapped to the vector $\vec{u}_0 = (\cos 30°, \sin 30°, 0) = (\sqrt{3}/2, 1/2, 0)$, and the y-axis is mapped to $(-\sin 30°, \cos 30°, 0) = (-1/2, \sqrt{3}/2, 0)$. By forming a matrix whose columns are consist of $\vec{u}_0$, $\vec{u}_1$, and O, as shown in the figure.


$\vec{u}_1 = \left(-\frac{1}{2}, \frac{\sqrt{3}}{2}, 0\right)$
$\vec{u}_0 = \left(\frac{\sqrt{3}}{2}, \frac{1}{2}, 0\right)$
$F_{\vec{e}_1}$
G
$O = (2, 1, 1)$
F_O
$F_{\vec{e}_0}$

$$\begin{bmatrix} \frac{\sqrt{3}}{2} & \frac{1}{2} & 2 \\ \frac{1}{2} & \frac{\sqrt{3}}{2} & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

Fig. 2: Generating a homogeneous matrix for an affine transformation.

Let's consider a number of examples representing affine transformations in 3-dimensional space by 4×4 matrices. (The two dimensional cases can be extracted by just ignoring the rows and columns for the z-coordinates.)

Translation: Translation by a fixed vector $\vec{v}$ maps any point p to $p + \vec{v}$. Note that, since free vectors have no position in space, they are not altered by translation (see Fig. 3(a)). Suppose that relative to the standard frame, $v[F] = (\alpha_x, \alpha_y, \alpha_z, 0)$ are the homogeneous coordinates of $\vec{v}$. The three unit vectors are unaffected by translation, and the origin is mapped to $O + \vec{v}$, whose homogeneous coordinates are $(\alpha_x, \alpha_y, \alpha_z, 1)$. Thus, by the rule given earlier, the homogeneous matrix representation for this translation transformation is

$$T_{\vec{v}} = \begin{pmatrix} 1 & 0 & 0 & \alpha_x \\ 0 & 1 & 0 & \alpha_y \\ 0 & 0 & 1 & \alpha_z \\ 0 & 0 & 0 & 1 \end{pmatrix}. \quad T_{\vec{v}}\, p = \begin{pmatrix} 1 & 0 & 0 & \alpha_x \\ 0 & 1 & 0 & \alpha_y \\ 0 & 0 & 1 & \alpha_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} p_x + \alpha_x \\ p_y + \alpha_y \\ p_z + \alpha_z \\ 1 \end{pmatrix}.$$

Scaling: *Scaling* is a transformation which is performed relative to some central fixed point. We will assume that this point is the origin of the standard coordinate frame. (We will leave the general case of scaling about an arbitrary point in space as an exercise.) Given a vector $\beta = (\beta_x, \beta_y, \beta_z)$, this transformation maps the object (point or vector) with coordinates $(\alpha_x, \alpha_y, \alpha_z, \alpha_w)$ to $(\beta_x\alpha_x, \beta_y\alpha_y, \beta_z\alpha_z, \alpha_w)$ (see Fig. 3(b)).

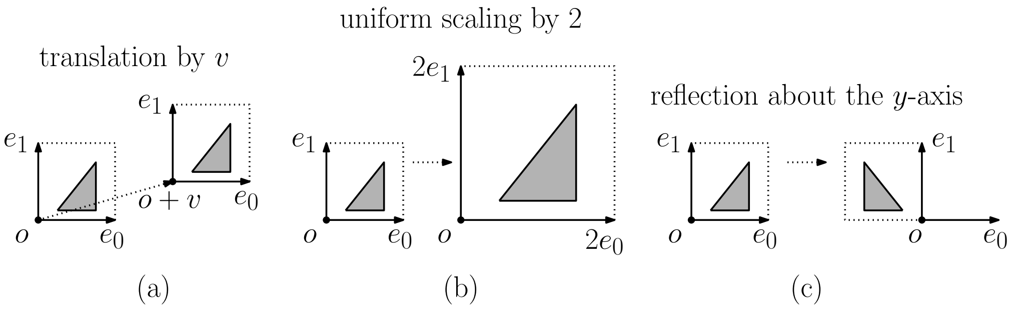


Fig. 3: Derivation of transformation matrices.

When $\beta_x = \beta_y = \beta_z$ this is called *uniform scaling*, since all three coordinates are stretched by the same amount. Otherwise, it is called *nonuniform scaling*. The associated transformation matrix is:

$$S_\beta = \begin{pmatrix} \beta_x & 0 & 0 & 0 \\ 0 & \beta_y & 0 & 0 \\ 0 & 0 & \beta_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \quad S_\beta\, p = \begin{pmatrix} \beta_x & 0 & 0 & 0 \\ 0 & \beta_y & 0 & 0 \\ 0 & 0 & \beta_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} \beta_x p_x \\ \beta_y p_z \\ \beta_z p_z \\ 1 \end{pmatrix}.$$

Observe that both points and vectors are altered by scaling.

Reflection: In its most general form, a reflection in the plane is given a line and maps points by flipping the plane about this line. A reflection in 3-space is given a plane, and flips points in space about this plane. In this case, reflection is just a special case of scaling, but where the scale factor is negative. A common simple version of this is when the plane about which the reflection is performed is one of the coordinate planes (corresponding to $x = 0$, $y = 0$, or $z = 0$).

For example, to reflect points about the yz-coordinate plane (that is, the plane $x = 0$), we can scale the x-coordinate by -1 (see Fig. 3(c)). Using the scaling matrix above, we have the following transformation matrix:

$$F_x = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \quad F_x\, p = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} -p_x \\ p_y \\ p_z \\ 1 \end{pmatrix}.$$

The cases for the other two coordinate frames are similar. Reflection about an arbitrary line (in 2-space) or a plane (in 3-space) is left as an exercise.

Rotation: In its most general form, rotation is defined to take place about some fixed point, and around some fixed vector in space. We will consider the simplest case where the fixed point is the origin of the coordinate frame, and the vector is one of the coordinate axes. There are three basic rotations: about the x, y and z-axes. In each case the rotation is counterclockwise through an angle θ (given in radians). The rotation is assumed to be in accordance with a right-hand rule: if your right thumb is aligned with the axes of rotation, then positive rotation is indicated by the direction in which the fingers of this hand are pointing. To produce a clockwise rotation, simply negate the angle involved.

Consider a rotation about the z-axis. The z-unit vector and origin are unchanged. The x-unit vector is mapped to $(\cos\theta, \sin\theta, 0, 0)$, and the y-unit vector is mapped to $(-\sin\theta, \cos\theta, 0, 0)$ (see Fig. 4(a)). Thus the rotation matrix is:

$$R_{z,\theta} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Observe that both points and vectors are altered by rotation. For the other two axes we have:

$$R_{x,\theta} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \qquad R_{y,\theta} = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

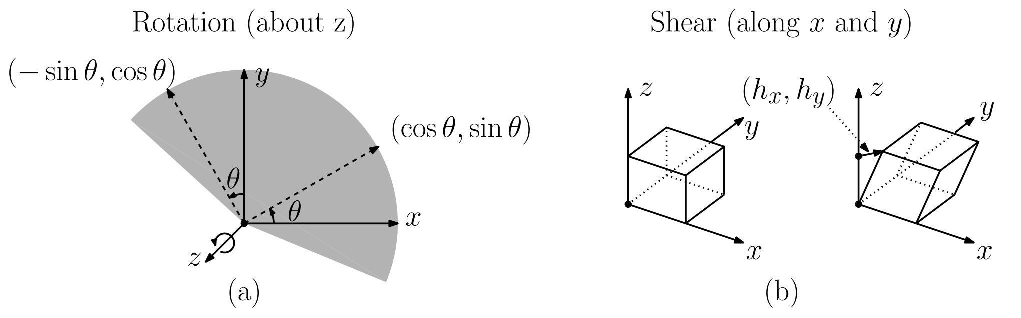


Fig. 4: Rotation and shearing.

If (as with Unity) the coordinate frame is left-handed, then the directions of all the rotations are reversed as well (clockwise, rather than counter-clockwise). Rotations about the coordinate axes are often called *Euler angles*. Rotations can generally be performed around any vector, called the *axis of rotation*, but the resulting transformation matrix is significantly more complex than the above examples.

Shearing: (Optional) A shearing transformation is perhaps the hardest of the group to visualize. Think of a shear as a transformation that maps a square into a parallelogram by sliding one side parallel to itself while keeping the opposite side fixed. In 3-dimensional space, it maps a cube into a parallelepiped by sliding one face parallel while keeping the opposite face fixed (see Fig. 4(b)). We will consider the simplest form, in which we start with a unit cube whose lower left corner coincides with the origin. Consider one of the axes, say the z-axis. The face of the cube that lies on the xy-coordinate plane does not move. The face that lies on the plane $z = 1$, is translated by a vector (h_x, h_y). In general, a point $p = (p_x, p_y, p_z, 1)$ is translated by the vector $p_z(h_x, h_y, 0, 0)$. This vector is orthogonal to the z-axis, and its length is proportional to the z-coordinate of p. This is called an *xy-shear*. (The yz- and xz-shears are defined analogously.)

Under the xy-shear, the origin and x- and y-unit vectors are unchanged. The z-unit vector is mapped to $(h_x, h_y, 1, 0)$. Thus the matrix for this transformation is:

$$H_{h_x,h_y} = \begin{pmatrix} 1 & 0 & h_x & 0 \\ 0 & 1 & h_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad H_{h_x,h_y}\, p = \begin{pmatrix} 1 & 0 & h_x & 0 \\ 0 & 1 & h_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} p_x + h_x p_z \\ p_y + h_y p_z \\ p_z \\ 1 \end{pmatrix}.$$

Shears involving any other pairs of axes are defined analogously.

Transformations in Unity: Recall that all game objects in Unity (in particular, Monobehaviour objects) are associated with a member called transform, which is of type Transform. This object controls the position and orientation of the object. If the object is *not* being controlled by the physics engine (that is, if it is kinematic), then you can control its movement through your scripts. (Otherwise, you should just let the physics engine do its job.)

Any Transform object supports the following operations for the two most common rigid transformations, translation and rotation:

- void Translate(Vector3 translation, Space relativeTo = Space.Self):
  This performs translation of the current object by the vector translation. The second argument specifies the coordinate frame relative to which the rotation is performed. By default, it is with respect to the object's own (local) coordinate frame. By setting the second argument to Space.World, the operation is performed with respect to the world coordinates (see Fig. 5).
  For example, if in your update method you wanted to translate the current object forward by some given linear speed (in units per second), you could use

  ```
  transform.Translate(Vector3.forward * speed * Time.deltaTime);
  ```

- void Rotate(Vector3 eulerAngles, Space relativeTo = Space.Self):
  This performs an Euler-angle based rotation clockwise in degrees (see below). Specifically, it rotates by eulerAngles.z degrees around the z- axis, eulerAngles.x degrees around the x-axis, and eulerAngles.y degrees around the y-axis (in that order). Given Unity's coordinate system, this means roll, then pitch, then yaw. The second argument specifies the coordinate frame relative to which the rotation is performed. By default, it is with respect to the object's own coordinate frame.
  For example, if in your update method you wanted to rotate the current object by some given angular speed (in degrees per second) about its own vertical axis, you could use

  ```
  transform.Rotate(Vector3.up * Time.deltaTime, Space.Self);
  ```

  This example works because Vector3.up $= (0, 1, 0)$, and thus this is an Euler-angle rotation about the y-axis. Here is another example:

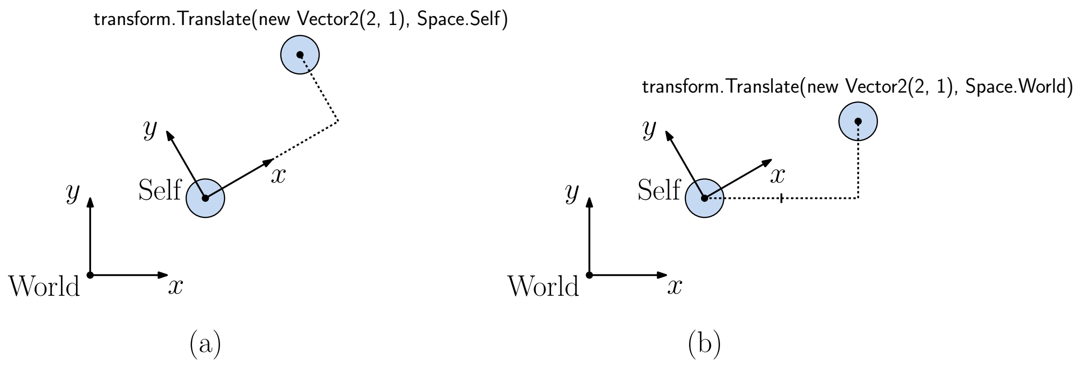

transform.Translate(new Vector2(2, 1), Space.Self)
y
x
Self
y
x
World
(a)
transform.Translate(new Vector2(2, 1), Space.World)
y
x
Self
y
x
World
(b)

Fig. 5: Translation relative to Space.Self and Space.World.