CMSC 425: Lecture 6
Affine Transformations and Rotations

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.

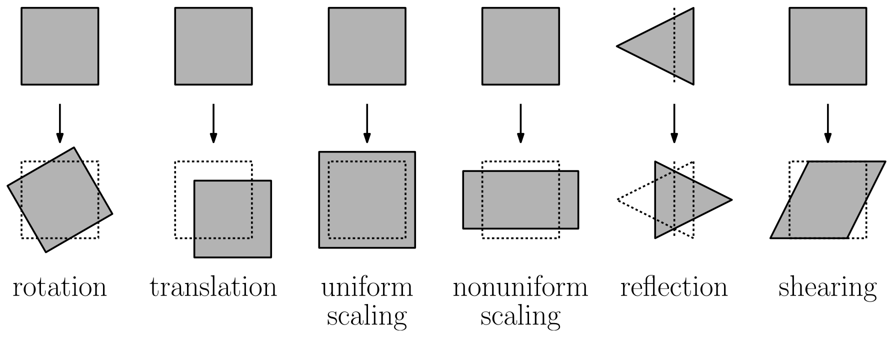


Fig. 1: Examples of affine transformations.

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.

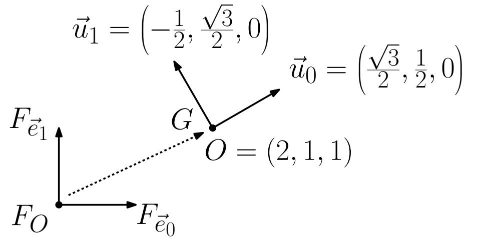
Formal Definition: Formally, an *affine transformation* is a mapping from one affine space to another (which may be, and in fact usually is, the same space) that preserves affine combinations. For example, this implies that given any affine transformation T and two points p and q, and any scalar α,

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

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 d-dimensional frame F consisting of an origin point $F.O$ and basis vectors $F.\vec{e}_0$ through $F.\vec{e}_{d-1}$. 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 $F.O$ 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

$$T(\vec{v}) = \begin{pmatrix} 1 & 0 & 0 & \alpha_x \\ 0 & 1 & 0 & \alpha_y \\ 0 & 0 & 1 & \alpha_z \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Scaling: *Uniform 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 scalar β, this transformation maps the object (point or vector) with coordinates $(\alpha_x, \alpha_y, \alpha_z, \alpha_w)$ to $(\beta\alpha_x, \beta\alpha_y, \beta\alpha_z, \alpha_w)$ (see Fig. 3(b)).

In general, it is possible to specify separate scaling factors for each of the axes. This is called *nonuniform scaling*. The unit vectors are each stretched by the corresponding

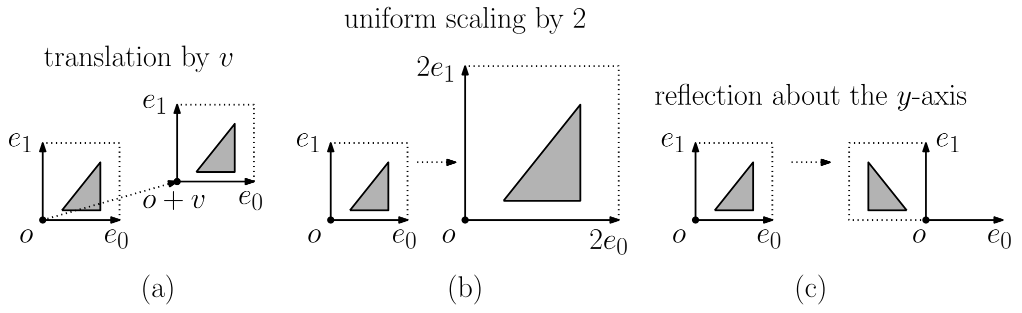


Fig. 3: Derivation of transformation matrices.

scaling factor, and the origin is unmoved. Thus, the transformation matrix has the following form:

$$S(\beta_x, \beta_y, \beta_z) = \begin{pmatrix} \beta_x & 0 & 0 & 0 \\ 0 & \beta_y & 0 & 0 \\ 0 & 0 & \beta_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

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:

$$F_x = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

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:

$$R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \qquad R_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

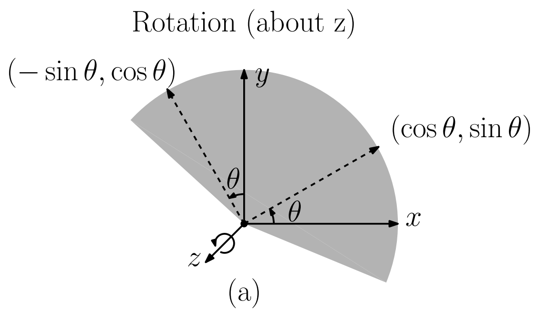


Fig. 4: Rotation and shearing.

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:

$$H_{xy}(h_x, h_y) = \begin{pmatrix} 1 & 0 & h_x & 0 \\ 0 & 1 & h_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Shears involving any other pairs of axes are defined analogously.

$$H_{yz}(h_y, h_z) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ h_y & 1 & 0 & 0 \\ h_z & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad H_{zx}(h_z, h_x) = \begin{pmatrix} 1 & h_x & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & h_z & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

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 coordinate frame.

  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 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. (I believe that, because of Unity's convention of using left-handed coordinate frames, a positive rotation corresponds to a *clockwise* rotation, but I am not entirely sure about this.)

  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, speed * Time.deltaTime);
  ```

Rotation and Orientation in 3-Space: One of the trickier problems 3-d geometry is that of parameterizing rotations and the orientation of frames. We have introduced the notion of orientation before (e.g., clockwise or counterclockwise). Here we mean the term in a somewhat different sense, as a directional position in space. Describing and managing rotations in 3-space is a somewhat more difficult task (at least conceptually), compared with the relative simplicity of rotations in the plane. We will explore two methods for dealing with rotation, *Euler angles* and *quaternions*.

Euler Angles: Leonard Euler was a famous mathematician who lived in the 18th century. He proved many important theorems in geometry, algebra, and number theory, and he is credited as the inventor of graph theory. Among his many theorems is one that states that the composition any number of rotations in three-space can be expressed as a single rotation in 3-space about an appropriately chosen vector. Euler also showed that any rotation in 3-space could be broken down into exactly three rotations, one about each of the coordinate axes.

Suppose that you are a pilot, such that the x-axis points to your left, the y-axis points ahead of you, and the z-axis points up (see Fig. 5). (This is the coordinate frame that I prefer, which is also used by the Unreal engine. Note that Unity swaps the z and y axes.) Then a rotation about the x-axis, denoted by ϕ, is called the *pitch*. A rotation about the y-axis, denoted by θ, is called *roll*. A rotation about the z-axis, denoted by ψ, is called *yaw*. Euler's theorem states that any position in space can be expressed by composing three such rotations, for an appropriate choice of (ϕ, θ, ψ).

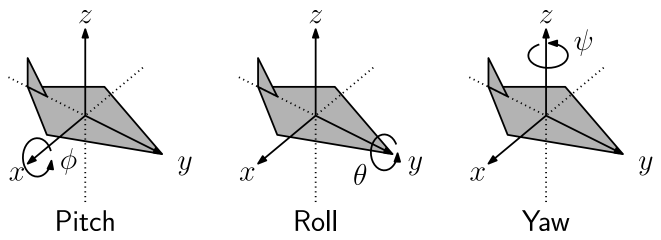


Fig. 5: Euler angles: pitch, roll, and yaw.

The order in which the rotations are performed is significant. In Unity (using the command transform.Rotate(x, y, z)), the order is the z-axis first, x-axis second, and y-axis third. Recalling that Unity switches the rolls of the z and y axes relative to the above figure, this means that it performs the operations in the order roll, then pitch, then yaw.

Shortcomings of Euler angles: There are some problems with Euler angles. One issue is the fact that this representation depends on the choice of coordinate system. In the plane, a 30-degree rotation is the same, no matter what direction the axes are pointing (as long as they are orthonormal and right-handed). However, the result of an Euler-angle rotation depends very much on the choice of the coordinate frame and on the order in which the axes are named. (Later, we will see that quaternions do provide such an intrinsic system.)

Another problem with Euler angles is called *gimbal lock*. Whenever we rotate about one axis, it is possible that we could bring the other two axes into alignment with each other. (This happens, for example if we rotate x by $90°$.) This causes problems because the other two axes no longer rotate independently of each other, and we effectively lose one degree of freedom.

Gimbal lock as induced by one ordering of the axes can be avoided by changing the order in which the rotations are performed. But, this is rather messy, and it would be nice to have a system that is free of this problem.

Quaternions: We will now delve into a subject, which at first may seem quite unrelated. But keep the above expression in mind, since it will reappear in most surprising way. This story begins in the early 19th century, when the great mathematician William Rowan Hamilton was searching for a generalization of the complex number system.

Imaginary numbers can be thought of as linear combinations of two basis elements, 1 and i, which satisfy the multiplication rules $1^2 = 1$, $i^2 = -1$ and $1 \cdot i = i \cdot 1 = i$. (The interpretation of $i = \sqrt{-1}$ arises from the second rule.) A complex number $a + bi$ can be thought of as a vector in 2-dimensional space (a, b). Two important concepts with complex numbers are the *modulus*, which is defined to be $\sqrt{a^2 + b^2}$, and the *conjugate*, which is defined to be $(a, -b)$. In vector terms, the modulus is just the length of the vector and the conjugate is just a vertical reflection about the x-axis. If a complex number is of modulus 1, then it can be expressed as $(\cos\theta, \sin\theta)$. Thus, there is a connection between complex numbers and 2-dimensional rotations. Also, observe that, given such a unit modulus complex number, its conjugate is $(\cos\theta, -\sin\theta) = (\cos(-\theta), \sin(-\theta))$. Thus, taking the conjugate is something like negating the associated angle.

Hamilton was wondering whether this idea could be extended to three dimensional space. You might reason that, to go from 2D to 3D, you need to replace the single imaginary quantity i with two imaginary quantities, say i and j. Unfortunately, this this idea does not work. After many failed attempts, Hamilton finally came up with the idea of, rather than using two imaginaries, instead using three imaginaries i, j, and k, which behave as follows:

$$i^2 = j^2 = k^2 = ijk = -1 \qquad ij = k,\ jk = i,\ ki = j.$$

Combining these, it follows that $ji = -k$, $kj = -i$ and $ik = -j$. The skew symmetry of multiplication (e.g., $ij = -ji$) was actually a major leap, since multiplication systems up to that time had been commutative.)

Hamilton defined a *quaternion* to be a generalized complex number of the form

$$\mathbf{q} = q_0 + q_1 i + q_2 j + q_3 k.$$

Thus, a quaternion can be viewed as a 4-dimensional vector $\mathbf{q} = (q_0, q_1, q_2, q_3)$. The first quantity is a scalar, and the last three define a 3-dimensional vector, and so it is a bit more intuitive to express this as $\mathbf{q} = (s, u)$, where $s = q_0$ is a scalar and $u = (q_1, q_2, q_3)$ is a vector in 3-space. We can define the same concepts as we did with complex numbers:

Conjugate: $\mathbf{q}^* = (s, -u)$
Modulus: $|\mathbf{q}| = \sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^2} = \sqrt{s^2 + (u \cdot u)}$
Unit Quaternion: $\mathbf{q}$ is said to be a unit quaternion if $|\mathbf{q}| = 1$

Quaternion Multiplication: Consider two quaternions $\mathbf{q} = (s, u)$ and $\mathbf{p} = (t, v)$:

$$\begin{aligned} \mathbf{q} &= (s, u) = s + u_x i + u_y j + u_z k \\ \mathbf{p} &= (t, v) = t + v_x i + v_y j + v_z k. \end{aligned}$$

If we multiply these two together, we'll get lots of cross-product terms, such as $(u_x i)(v_y j)$, but we can simplify these by using Hamilton's rules. That is, $(u_x i)(v_y j) = u_x v_h (ij) = u_x v_h k$. If we do this, simplify, and collect common terms, we get a very messy formula involving 16 different terms. (The derivation is left as an exercise.) The formula can be expressed somewhat succinctly in the following form:

$$\mathbf{qp} \ = \ (st - (u \cdot v), sv + tu + u \times v).$$

Note that the above expression is in the quaternion scalar-vector form. The first term $st-(u \cdot v)$ evaluates to a scalar (recalling that the dot product returns a scalar), and the second term $(sv+tu+u \times v)$ is a sum of three vectors, and so is a vector. It can be shown that quaternion multiplication is associative, but not commutative.

Quaternion Multiplication and 3-d Rotation: So far, everything we have said about quaternions seems to be purely abstract algebraic manipulations. We will show that in fact, they provide a natural way to encode 3-dimensional rotations. First, let us define a *pure quaternion* to be one with a 0 scalar component

$$\mathbf{p} = (0, v).$$

Any quaternion of nonzero magnitude has a multiplicative *inverse*, which is defined to be

$$\mathbf{q}^{-1} = \frac{1}{|\mathbf{q}|^2}\mathbf{q}^*.$$

(To see why this works, try multiplying $\mathbf{q}\mathbf{q}^{-1}$, and see what you get.) Observe that if $\mathbf{q}$ is a unit quaternion, then it follows that $\mathbf{q}^{-1} = \mathbf{q}^*$.

As you might have guessed, our objective will be to show that there is a relation between rotating vectors and multiplying quaternions. In order apply this insight, we need to first show how to represent rotations as quaternions and 3-dimensional vectors as quaternions. After a bit of experimentation, the following does the trick:

Vector: Given a vector $v = (v_x, v_y, v_z)$ to be rotated, we will represent it by the pure quaternion $(0, v)$.

Rotation: To represent a rotation by angle θ about a unit vector u, you might think, we'll use the scalar part to represent θ and the vector part to represent u. (This is called the *axis-angle representation* of a 3-dimensional rotation. Unfortunately, this doesn't quite work. After a bit of experimentation, you will discover that the right way to encode this rotation is with the quaternion $\mathbf{q} = (\cos(\theta/2), (\sin(\theta/2))u)$. (You might wonder, why we do we use $\theta/2$, rather than θ. The reason, as we shall see below, is that "this is what works.")

Rotation Operator: Given a vector v represented by the quaternion $\mathbf{p} = (0, v)$ and a rotation represented by a unit quaternion $\mathbf{q}$, we define the *rotation operator* to be:

$$R_{\mathbf{q}}(\mathbf{p}) \ = \ \mathbf{q}\mathbf{p}\mathbf{q}^{-1} \ = \ \mathbf{q}\mathbf{p}\mathbf{q}^*.$$

(The last equality results from the fact that $\mathbf{q}^{-1} \ = \ \mathbf{q}^*$, if $\mathbf{q}$ is a unit quaternion). We claim that the result of this operation will always be a pure quaternion, and so it is possible to interpret the result as a vector. In particular, this vector will be the result of applying the rotation $\mathbf{q}$ to v.

We will give a formal justification of this later, but for now, let's consider what this gives us. Let us apply the above quaternion multiplication rule and use the fact that $\mathbf{q}^{-1} = \mathbf{q}^*$ for a unit quaternion $\mathbf{q} = (s, u)$. Letting $\mathbf{p} = (0, v)$ denote the object to be rotated and expanding/simplifying we obtain:

$$R_{\mathbf{q}}(\mathbf{p}) \;=\; (0,\ (s^2 - (u \cdot u))v + 2u(u \cdot v) + 2s(u \times v)). \tag{1}$$

(We leave the derivation as an exercise, but a few nontrivial facts regarding dot products and cross products need to applied.) It is not obvious that this has anything to do with rotation, but later we will show that this corresponds exactly to rotating v about the axis u by θ degrees.

Quaternions in Unity: Unity supports an object called Quaternion that encapsulates a quaternion. You can generate a quaternion that performs a rotation by x degrees about a given 3-dimensional vector $\vec{u}$ using the command Quaternion.AngleAxis(x, u). For example, the following command sets the current object's rotation to a rotation by a 30° about the vertical axis.

```
transform.rotation = Quaternion.AngleAxis(30, Vector.up);
```

There are a number of useful operations defined on quaternions, such as converting Euler angles to quaternions, multiplying quaternions, interpolating between quaternions, and computing a frame that is oriented with a quaternion.

Example: Consider the 3-d "roll" rotation shown in Fig. 6. This rotation can be achieved by performing a rotation about the y-axis by $\theta = 90$ degrees. Thus $\theta = \pi/2$, and the axis of rotation is $\widehat{u} = (0, 1, 0)$, and so we have $s = \cos(\theta/2) = 1/\sqrt{2}$ and $u = (\sin(\theta/2))\widehat{u} = (0, 1/\sqrt{2}, 0)$, and hence

$$\mathbf{q} \;=\; (\cos(\theta/2), (\sin(\theta/2))u) \;=\; \left(\cos\left(\frac{\pi}{4}\right), \sin\left(\frac{\pi}{4}\right)(0,1,0)\right) \;=\; \left(\frac{1}{\sqrt{2}}, \left(0, \frac{1}{\sqrt{2}}, 0\right)\right).$$

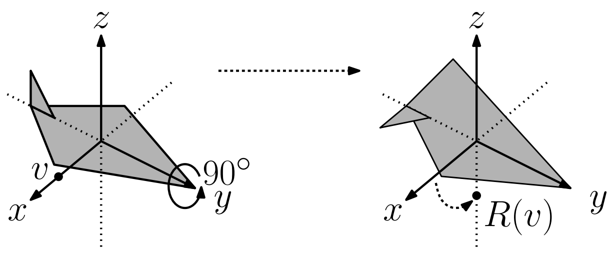


Fig. 6: Rotation example.

Let us consider how the x-unit vector $v = (1, 0, 0)$ is transformed under this rotation. To reduce this to a quaternion operation, we encode v as a pure quaternion $\mathbf{p} = (0, v) = (0, (1, 0, 0))$. Observe that

$$s^2 - (u \cdot u) = \frac{1}{2} - \frac{1}{2} = 0, \qquad (u \cdot v) = 0, \qquad \text{and} \qquad (u \times v) = \left(0, 0, \frac{-1}{\sqrt{2}}\right).$$

By applying the rotation operator, by Eq. (1), we have

$$\begin{aligned} R_{\mathbf{q}}(\mathbf{p}) &= (0,\ (s^2 - (u \cdot u))v + 2u(u \cdot v) + 2s(u \times v)) \\ &= (0,\ 0v + 2u0 + 2s(0, 0, -1/\sqrt{2})) \\ &= (0,\ \vec{0} + \vec{0} + (2/\sqrt{2})(0, 0, -1/\sqrt{2})) \\ &= (0, (0, 0, -1)). \end{aligned}$$

Interpreting $\mathbf{p}$ as a vector $(0, 0, -1)$, we see that, as expected, quaternion rotation rotates the vector $v = (1, 0, 0)$ by 90° to $(0, 0, -1)$ (see Fig. 6).

Why Quaternions Work: (Optional) In order to understand why the above quaternion operation implements rotation, we begin with the concept of *angular displacement*, which involves rotating a given vector v about a given rotation axis u (any unit vector) by a certain number of degrees θ.

Let $R(v)$ denote this rotated vector (see Fig. 7(a)). In order to derive this, we begin by decomposing v as the sum of its components that are parallel to and orthogonal to u, respectively.

$$v_{\|} = (u \cdot v)u \qquad v_{\perp} = v - v_{\|} = v - (u \cdot v)u.$$

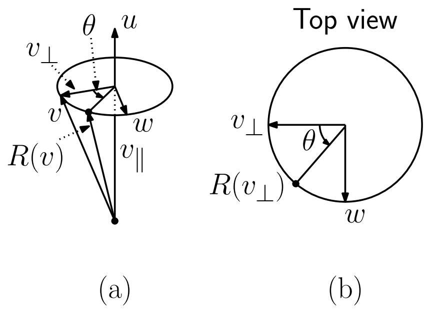


Fig. 7: Angular displacement.

Note that $v_{\|}$ is unaffected by the rotation, but $v_{\perp}$ is rotated to a new position $R(v_{\perp})$. To determine this rotated position, we will first construct a vector that is orthogonal to $v_{\perp}$ lying in the plane of rotation.

$$w = u \times v_{\perp} = u \times (v - v_{\|}) = (u \times v) - (u \times v_{\|}) = u \times v.$$

The last step follows from the fact that u and $v_{\|}$ are parallel, and so the cross product is zero. Clearly w is orthogonal to both $v_{\perp}$ and u. Furthermore, because $v_{\perp}$ is orthogonal to the unit vector u, it follows from basic properties of the cross product that w is the same length as $v_{\perp}$.

Now, consider the plane spanned by $v_{\perp}$ and w (see Fig. 7(b)). We have

$$R(v_{\perp}) = (\cos\theta)v_{\perp} + (\sin\theta)w.$$

From this and the fact that $R(v_\|) = v_\|$, we have

$$\begin{aligned} R(v) &= R(v_\|) + R(v_\perp) \;=\; v_\| + (\cos\theta)v_\perp + (\sin\theta)w \\ &= (u \cdot v)u + (\cos\theta)(v - (u \cdot v)u) + (\sin\theta)w \\ &= (\cos\theta)v + (1 - \cos\theta)u(u \cdot v) + (\sin\theta)(u \times v). \end{aligned}$$

In summary, we have the following formula expressing the effect of the rotation of vector v by angle θ about a rotation axis u:

$$R(v) \;=\; (\cos\theta)v + (1 - \cos\theta)u(u \cdot v) + (\sin\theta)(u \times v). \tag{2}$$

This expression is the image of v under the rotation. Notice that, unlike Euler angles, this is expressed entirely in terms of intrinsic geometric functions (such as dot and cross product), which do not depend on the choice of coordinate frame. This is a major advantage of this approach over Euler angles.

Now that we know how to express rotation in terms of vector operations, let's see how this relates to the quaternion rotation operation. Let us see if we can express this in a more suggestive form. Since $\mathbf{q}$ is of unit magnitude, we can express it as

$$\mathbf{q} = \left(\cos\frac{\theta}{2}, \left(\sin\frac{\theta}{2}\right) u\right), \qquad \text{where } \|u\| = 1.$$

Plugging this into Eq. (1) and applying some standard trigonometric identities, we obtain

$$\begin{aligned} R_{\mathbf{q}}(\mathbf{p}) &= \left(0, \left(\cos^2\frac{\theta}{2} - \sin^2\frac{\theta}{2}\right) v + 2\left(\sin^2\frac{\theta}{2}\right) u(u \cdot v) + 2\cos\frac{\theta}{2}\sin\frac{\theta}{2}(u \times v)\right) \\ &= (0,\; (\cos\theta)v + (1 - \cos\theta)u(u \cdot v) + \sin\theta(u \times v)). \end{aligned}$$

Observe that the vector part of this quaternion is *identical* to the angular displacement equation for $R(v)$ presented in Eq. (2), implying that the quaternion rotation operator achieves the desired rotation.

Composing Rotations: (Optional) We have shown that each unit quaternion corresponds to a rotation in 3-space. This is an elegant representation, but can we manipulate rotations through quaternion operations? The answer is yes. In particular, the action of multiplying two unit quaternions results in another unit quaternion. Furthermore, the resulting product quaternion corresponds to the composition of the two rotations. In particular, given two unit quaternions $\mathbf{q}$ and $\mathbf{q}'$, a rotation by $\mathbf{q}$ followed by a rotation by $\mathbf{q}'$ is equivalent to a single rotation by the product $\mathbf{q}'' = \mathbf{q}'\mathbf{q}$. That is,

$$R_{\mathbf{q}'} R_{\mathbf{q}} = R_{\mathbf{q}''} \qquad \text{where } \mathbf{q}'' = \mathbf{q}'\mathbf{q}.$$

This follows from the associativity of quaternion multiplication, and the fact that $(\mathbf{q}\mathbf{q}')^{-1} = \mathbf{q}^{-1}\mathbf{q}'^{-1}$, as shown below.

$$\begin{aligned} R_{\mathbf{q}'}(R_{\mathbf{q}}(\mathbf{p})) &= \mathbf{q}'(\mathbf{q}\mathbf{p}\mathbf{q}^{-1})\mathbf{q}'^{-1} \;=\; (\mathbf{q}'\mathbf{q})\mathbf{p}(\mathbf{q}^{-1}\mathbf{q}'^{-1}) \\ &= (\mathbf{q}'\mathbf{q})\mathbf{p}(\mathbf{q}\mathbf{q}')^{-1} \;=\; \mathbf{q}''\mathbf{p}\mathbf{q}''^{-1} \\ &= R_{\mathbf{q}''}(\mathbf{p}). \end{aligned}$$