Samples: In the last few lectures, we have been discussing affine and Euclidean geometry, coordinate frames and affine transformations, and rotations. In this lecture, we work through a few examples of how to apply these concepts to solve a few concrete problems that might arise in the context of game programming.

Beam Bouncing:

Problem: You are working on a platformer game that takes place in a 2-dimensional space. The hero of your game has a gun that shoots a laser beam. This beam can bounce off of special objects (mirrors) in your environment. For inexperienced users, you provide a mode that will automatically aim the gun to hit a given target by bouncing it off one of these mirrors.

More formally, let \( a \) and \( b \) be two points in the plane. Let \( \ell \) denote the line passing through these points. Let \( p \) denote the location of the laser gun and let \( q \) denote the desired target point. Both of these points lie on the same side of \( \ell \) (see Fig. 1(a)). The objective is to compute the point \( r \) on \( \ell \) such that a shot fired from \( p \) towards \( r \) will bounce off and hit \( q \). As is normal with reflections, the angle of incidence equals the angle of reflection (the angle \( \theta \) shown in Fig. 1(a)).

Solution: Let \( \vec{v} \gets b - a \) be the vector directed from \( a \) to \( b \). We can express any point on the line \( \ell \) as an affine combination \( (1 - t)a + tb \). (When \( t = 0 \) the point is at \( a \), and when \( t = 1 \) the point is at \( b \)).

In order to compute the point of reflection, we will employ a well-known trick. Let us reflect the point \( q \) to the other side of \( \ell \), and then we will draw a straight line from this reflected point \( q' \) to \( p \). The point where the line segment \( pq' \) intersects \( \ell \) will be the desired point \( r \) (see Fig. 1(b)).

We can compute the point \( q' \) through an application of orthogonal projection. Let \( \vec{u} \gets q - a \) be the vector directed from \( a \) to \( q \) (see Fig. 1(c)). Recall from the lecture on the dot-product that the orthogonal projection of \( \vec{u} \) onto \( \vec{v} \), which we denote by \( \vec{u}' \), is given by the formula

\[
\vec{u}' \gets \frac{\vec{u} \cdot \vec{v}}{\vec{v} \cdot \vec{v}} \vec{v}
\]
Letting \( \vec{u}'' \) denote the orthogonal complement, we have \( \vec{u}'' \leftarrow \vec{u} - \vec{u}' \). We can now obtain \( q' \) by starting at the point \( q \) and add two copies of \(-\vec{u}''\). That is, \( q' \leftarrow q - 2\vec{u}'' \).

Finally, let’s compute point \( r \) as the intersection of the line segment \( \overline{pq} \) with \( \ell \). We will do this by solving a linear system of equations. As mentioned above, \( r = (1-t)a + tb \), for some scalar \( t \). Since \( r \) also lies on the line between \( p \) and \( q' \) we have \( r = (1-s)p + sq' \) for some scalar \( s \). Equating these, we have \( (1-t)a + tb = (1-s)p + sq' \), that is

\[
a + t(b - a) = p + s(q' - p).
\]

Expressing the \( x \)- and \( y \)-coordinates separately we have two equations:

\[
\begin{align*}
a_x + t(b_x - a_x) &= p_x + s(q'_x - p_x) \\
a_y + t(b_y - a_y) &= p_y + s(q'_y - p_y).
\end{align*}
\]

Solving for \( t \), we obtain:

\[
t \leftarrow \frac{(a_x - p_x)(q'_y - p_y) - (a_y - p_y)(q'_x - p_x)}{(b_y - a_y)(q'_x - p_x) - (b_x - a_x)(q'_y - p_y)}.
\]

Given this value of \( t \), we finally obtain the desired point \( r \leftarrow (1-t)a + tb \). If \( 0 \leq t \leq 1 \), then the point of reflection lies along the mirror \( \overline{ab} \). Otherwise, the reflection point lies outside of this segment, and it is impossible to hit \( q \) from \( p \). The code block below (which I have not tested) demonstrates how this can be implemented in Unity.

```csharp
Vector3 Reflect(Vector3 a, Vector3 b, Vector3 p, Vector3 q) {
    Vector3 v = b - a; // vector from a to b
    Vector3 u = q - a; // vector from a to q
    Vector3 u1 = (Vector3.Dot(u,v)/Vector3.Dot(v,v)) * v; // proj u on v
    Vector3 u2 = u - u1; // orthogonal complement
    Vector3 qq = q - 2 * u2; // reflection of q across line ab
    // compute intersection scalar
    float t = ((a.x - p.x) * (qq.y - p.y) - (a.y - p.y) * (qq.x - p.x)) / ((b.y - a.y) * (qq.x - p.x) - (b.x - a.x) * (qq.y - p.y));
    if (t < 0 || t > 1) return NoHit;
    else return (1 - t) * a + t * b;
}
```

Evasive Action:

**Problem:** In your latest game, you are simulating the AI for some alien space ships. Each space ship is associated with its current position, a point \( p \), and two unit-length vectors. The first vector \( \vec{v} \) indicates the direction in which the space ship is flying. The second vector \( \vec{u} \) is orthogonal to \( \vec{v} \) and indicates the direction that is up relative to the pilot flying the space ship (see Fig. 2(a)). There are various obstacles to be avoided (asteroids, and such) and the AI system needs to issue turning commands to avoid these obstacles. Given an obstacle at some point \( q \), the question that we want to determine is whether
we should turn (yaw) to the left or right and whether we should turn (pitch) up or down
to avoid the collision. (We won’t worry about the actual number of degrees of rotation
for now, just the direction.)

Solution: First observe that \( \vec{w} \leftarrow q - p \) defines a vector that is
directed from the space ship to the obstacle (see Fig. 2(b)). To convert this into a unit vector
(since we just care about the direction), let us normalize it to unit length. (Recall that
normalizing a vector involves dividing a vector by its length.) We can compute the
length of a vector as the square root of it dot product with itself. Thus, we have

\[
\vec{w} \leftarrow q - p \\
\hat{w} \leftarrow \text{normalize}(\vec{w}) = \frac{\vec{w}}{\|\vec{w}\|} = \frac{\vec{w}}{\sqrt{\vec{w} \cdot \vec{w}}} = \frac{\vec{w}}{\sqrt{w_x^2 + w_y^2 + w_z^2}}.
\]

What we want to know is whether this vector is pointing to the space-ship pilot’s left or
right (in which case we will turn the opposite direction), or is above or below (in which
case we will pitch in the opposite direction).

Let’s first tackle the problem of whether to turn pitch up or down. We can determine this
by checking whether the angle between the up-vector \( \vec{u} \) and \( \hat{w} \). If this angle is smaller
than 90°, then the obstacle is above us and we should pitch downward. Otherwise, we
should pitch upward. Given that both vectors have unit length, we can compute the
cosine of the angle between them by the dot product. If the dot product is positive, the
angle is smaller than 90°, thus the obstacle is above, and we turn down. Otherwise, we
turn down. We have

\[
\hat{w} \cdot \vec{u} \geq 0 \implies \text{(obstacle above) pitch downwards}
\]
\[
\hat{w} \cdot \vec{u} < 0 \implies \text{(obstacle below) pitch upwards}.
\]

(By the way, since we are only checking the sign of this dot product, not its magnitude,
it was not really necessary to normalize \( \vec{w} \) to unit length. We could have substituted \( \vec{w} \)
for \( \hat{w} \) above without affecting the correctness of the result.)

Next, let’s consider whether to turn left or right. We would like to perform a similar
type of computation, but to do so, we should generate a vector that indicates left and
right relative to the pilot of the ship. Such a vector will be orthogonal both to the
direction that we are flying and to the up direction. We can obtain such a vector using
the cross-product. In particular, define a vector \( \vec{r} \leftarrow \vec{v} \times \vec{u} \). By the right-hand rule, this vector will point to the pilot’s right. By our assumption that \( \vec{v} \) and \( \vec{u} \) are orthogonal to each other and of unit length, it follows from the definition of the cross product that \( \vec{r} \) will also be of unit length.

We reason in an analogous manner to the up-down case. If the angle between \( \hat{w} \) and \( \vec{r} \) is smaller than 90\(^\circ\), then the obstacle is to our right, and we turn left to avoid it. Otherwise, we turn right. This is equivalent to testing whether the cosine of the angle is positive or negative. Thus, we have

\[
\vec{r} \leftarrow \hat{u} \times \vec{u}
\]

\[
\hat{w} \cdot \vec{r} \geq 0 \implies \text{(obstacle to the right) yaw to the left}
\]

\[
\hat{w} \cdot \vec{r} < 0 \implies \text{(obstacle to the left) yaw to the right}
\]

A Unity implementation of this procedure (which I haven’t tested) can be found in the following code block.

```csharp
void Evade (Vector3 p, Vector3 v, Vector3 u, Vector3 q) {

Vector3 w = q - p; // vector from pilot to obstacle
float l = w.magnitude; // distance to obstacle
Vector3 ww = w.normalized; // directional vector to obstacle
if (Vector3.Dot (ww, u) >= 0) // obstacle is above?
    PitchDown ();
else
    PitchUp ();
Vector3 r = Vector3.Cross(v, u); // vector to pilot's right
if (Vector3.Dot (ww, r) >= 0) // obstacle is to the right
    YawToLeft ()
else
    YawToRight ();
}
```

We have not discussed how to perform the pitch or yaw operations. In Unity, these could be expressed as rotations about the vectors \( \vec{r} \) and \( \vec{u} \), respectively.

**Shot-Gun Simulator:**

**Problem:** You have been asked to implement a new weapon that behaves something like a shot gun. It has a wide range of effectiveness, but it is only effective at relatively small distances. Suppose that the end of the muzzle of the gun is located at a point \( p \) (in 3-dimensional space), the direction that it is pointed is given by a vector \( \vec{v} \) (which you may assume has been normalized to unit length). The gun shoots a spray of pellets within angle \( \theta \) about \( \vec{v} \), but bullets are only effective up to a distance of \( r \) from the end of the muzzle. (Let’s assume that \( \theta \) is given in degrees and is strictly smaller than 90\(^\circ\).) Given a point \( q \), write a short procedure to determine whether the point \( q \) will be hit when the gun is fired.
Solution: We need to determine (1) whether $q$ lies within the infinite cone and (2) whether $q$ is close enough to be hit. Define the vector $\vec{u}$ to be a vector that is directed from $p$ to $q$. We have $\vec{u} \leftarrow q - p$. By properties of the dot-product, the distance $\ell$ from $p$ to $q$ is given by $\ell \leftarrow ||\vec{u}|| = \sqrt{\vec{u} \cdot \vec{u}}$. If $\ell > r$ then $q$ is too far away to be hit, and we are done. Otherwise, in order for $q$ to lie within the cone, let’s scale $\vec{u}$ to unit length. Define $\hat{\vec{u}} \leftarrow$ normalize($\vec{u}$) = $\vec{u}/\ell$. The angle between $\vec{v}$ and $\hat{\vec{u}}$ should be at most $\theta$. Recall, that we can compute the cosine of two unit vectors by taking their dot product. Since the cosine is a monotonically decreasing function (for the angles we are interested in), this is equivalent to the condition $\hat{\vec{u}} \cdot \vec{v} \geq \cos \theta$. Oops! Remember that $\theta$ is given in degrees and the cosine function assumes that the argument is given in radians. To convert from degrees to radians we multiply by $\pi/180$. So the correct expression is

$$\hat{\vec{u}} \cdot \vec{v} \geq \cos \left( \theta \cdot \frac{\pi}{180} \right).$$

To summarize, we have the following test.

1. $\vec{u} \leftarrow q - p$
2. $\ell \leftarrow ||\vec{u}|| = \sqrt{\vec{u} \cdot \vec{u}} = \sqrt{u_x^2 + u_y^2 + u_z^2}$
3. $\hat{\vec{u}} \leftarrow$ normalize($\vec{u}$) = $\vec{u}/\ell$
4. $c_1 \leftarrow \hat{\vec{u}} \cdot \vec{v} = \hat{u}_x v_x + \hat{u}_y v_y + \hat{u}_z v_z$
5. $c_2 \leftarrow \cos \left( \theta \cdot \frac{\pi}{180} \right)$.

A hit occurs if and only if $c_1 \geq c_2$ and $\ell \leq r$. A Unity implementation of this procedure (which I haven’t tested) can be found in the following code block.
Does a shot gun fired at $p$ hit point $q$?

```c
bool HitMe(Vector3 p, Vector3 v, float theta, float r, Vector3 q) {
    Vector3 u = q - p; // vector from muzzle end (p) to target (q)
    float l = u.magnitude; // distance to target
    Vector3 uu = u.normalized; // directional vector to target
    float c1 = Vector3.Dot(uu, v); // cosine of angle between u and v
    float c2 = Mathf.Cos(theta * Mathf.PI / 180); // cosine of hit cone
    return (c1 >= c2) && (l <= r); // target within cone and distance
}
```