# CMSC 425: Lecture 7
# Geometric Programming: Sample Solutions

**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. **Caveat**: I have not tested the Unity code given here, so don't trust it!

**Shot-Gun Simulator:**

**Problem:** You have been asked to implement a new weapon that behaves something like a shot gun. It has a wide ange 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), and it has been aimed at a target point $t$ (see Fig. 1(a)). The gun shoots a spray of pellets within angle $\theta$ the central axis between $p$ and $t$, 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°.) Given a point $q$, write a procedure to determine whether the point $q$ will be hit when the gun is fired.

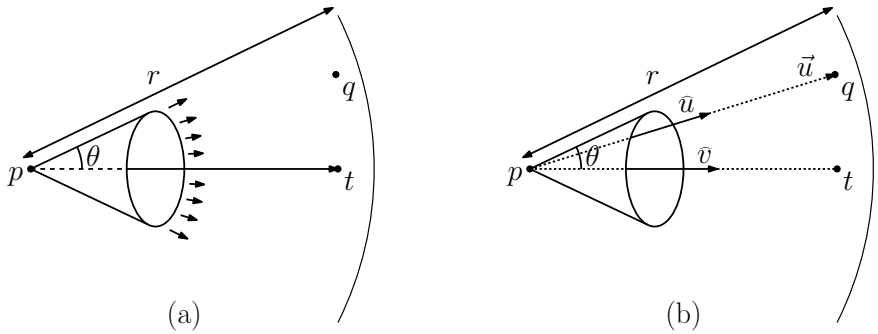


Fig. 1: Shot gun.

**Solution:** We need to determine (1) whether $q$ lies within the infinite cone about the central axis $\overline{pt}$ and (2) whether $q$ is close enough to be hit. How do we determine the first condition? We can construct two vectors, one from $p$ to $t$ and one from $p$ to $q$, and determine whether the angle between them is at most $\theta$ degrees (see Fig. 1(b)).

First, define a vector $\vec{v}$ to be the vector from $p$ to $t$, that is, $\vec{v} \leftarrow t - p$. Next, define the vector $\vec{u}$ to be a vector that is directed from $p$ to $q$, thus, $\vec{u} \leftarrow q - p$. Let us normalize these vectors to unit length, by defining $\widehat{u} \leftarrow \text{normalize}(\vec{u}) = \vec{u}/\ell(u)$, where $\ell(u) = \|u\| = \sqrt{u \cdot u}$. (Here we have used the property of dot product, that the dot product of a vector with itself is the vector's squared length.) We can do the same for $\vec{v}$.

In order for $q$ to lie within the cone, we compute the angle between these vectors. 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 in the range from 0 to 180°), this is equivalent to the condition $\widehat{u} \cdot \vec{v} \geq \cos\theta$.

Be careful! 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

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

To solve (2), it suffices to test whether If $\ell(u) > r$ then $q$ is too far away to be hit.

To summarize, we have the following test.

$$\begin{array}{rcl}
\vec{v} & \leftarrow & t - p; \qquad \vec{u} \leftarrow q - p \\
\ell(v) & \leftarrow & \|\vec{v}\| = \sqrt{\vec{v} \cdot \vec{v}}; \qquad \ell(u) \leftarrow \|\vec{u}\| = \sqrt{\vec{u} \cdot \vec{u}} \\
\widehat{v} & \leftarrow & \text{normalize}(\vec{v}) = \vec{v}/\ell(v); \qquad \widehat{u} \leftarrow \text{normalize}(\vec{u}) = \vec{u}/\ell(u) \\
c_1 & \leftarrow & \widehat{u} \cdot \widehat{v} \\
c_2 & \leftarrow & \cos\left(\theta \cdot \frac{\pi}{180}\right) \\
\text{return} & & \textbf{true} \text{ iff } (c_1 \geq c_2 \text{ and } \ell(u) \leq r).
\end{array}$$

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 $p$ toward $t$ hit point $q$?

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

**Projectile Shooting:** Your game involves a shooting an object (and arrow, rock, grenade, or other projectile) in a certain direction. Your boss wants you to write a program to determine where the projectile will land, as part of an aiming tool for inexperienced players.

Suppose that the projectile is launched from a location that is $h$ meters above the ground. Following Unity's convention the projectile starts above on the vertical ($y$) axis, at coordinates $(0, h, 0)$. Suppose that the projectile is launched with a velocity given by the vector $\vec{v}_0 = (v_{0,x}, v_{0,y}, v_{0,z})$. Let's assume that the arrow is shot upwards, that is, $v_{0,y} > 0$. To simplify matters, let's assume that the projectile is shot in the forward ($z$) direction. Thus $v_{0,x} = 0$ and $v_{0,z} > 0$. We want to determine the distance $\ell$ from the shooter where the projectile hits the ground.

Let $t = 0$ denote the time at which the object is shot. After consulting a standard textbook on Physics, we are reminded that (on Earth at least) the force of gravity results in an acceleration of $g \approx 9.8 m/s^2$.

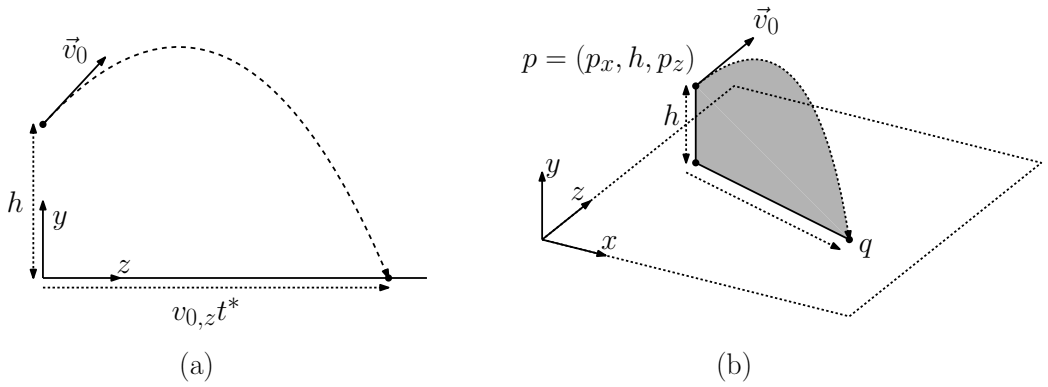


Fig. 2: Projectile shooting

After consulting your physics text, you find out that after $t$ time units have elapsed, the position of the projectile is $p(t) = (z(t), y(t))$, where

$$z(t) \;=\; v_{0,z}t \quad \text{and} \quad y(t) \;=\; h + v_{0,y}t - \frac{1}{2}gt^2.$$

(Assuming no wind resistance, the projectile's motion with the $z$-axis is constant over time as $v_{0,z}$. It's motion with respect to the $y$-axis follows a downward parabolic arc as a function of time $t$.) We are interested in the time $t$ when the projectile hits the ground, that is, $y(t) = 0$.

**Time of Impact:** Letting $a = g/2$, $b = -v_{0,y}$, and $c = -h$, we seek the value of $t$ such that $at^2 + bt + c = 0$. (We have intentionally negated the coefficients so that $a > 0$.) By the quadratic formula we have

$$t \;=\; \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \;=\; \frac{v_{0,y} \pm \sqrt{v_{0,y}^2 + 2gh}}{g}.$$

Note that the quantity under the square-root sign is positive and is larger than $v_{0,y}$, which implies that both roots exist, one is positive and one is negative. Clearly, we want the positive root. (As an exercise, what does the negative root correspond to?) Thus, we take the "+" root from the "±" option, which yields $t^* = \left(v_{0,y} + \sqrt{v_{0,y}^2 + 2gh}\right)/g$.

**Location of Impact:** We know that the projectile moves at a rate of $v_{0,z}$ units per second horizontally. Therefore, at time $t = t^*$ it has traveled a distance of $v_{0,z}t^*$ units. Since we started at the origin, the location we hit the ground is $(x, y, z) = (0, 0, v_{0,z}t^*)$.

**Generalizing this:** We assumed that the projectile was shot along the $z$-axis. Note that the generalization is very easy. We compute $t^*$ in the same manner as above, since it depends only on the height and vertical velocity. Then we apply the same reasoning for $x$ as for $z$. The projectile travels a distance of $v_{0,x}t^*$ units along the $x$-axis, so its final position is $(x, y, z) = (v_{0,x}t^*, 0, v_{0,z}t^*)$.

We also assumed that the projective was shot from $h$ units above the origin. If, instead it had been shot from some arbitrary point $(p_x, h, p_z)$, then we would displace the final location by this amount, as $q = (p_x + v_{0,x}t^*, 0, p_z + v_{0,z}t^*)$. This is where we would draw our spot for aiming tool (see Fig. 2(b)).

Aiming Tool for Projectile

```
Vector3 HitSpot (Vector3 p, Vector3 v) {
  float h = p.y;                        // height above ground
  float a = 9.8f/2.0f;                  // quadratic coefficients
  float b = -v.y;
  float c = -h;
  float t = (-b + Math.Sqrt(b*b - 4*a*c)) / (2*a); // time in the air
  return new Vector3 (p.x + v.x*t, 0, p.z + v.z*t); // where we hit
}
```

**Shooting and Arrow:** Before leaving the topic of shooting the projectile, it is worth observing that the Unity Physics engine can simulate the motion of the projectile. This will look fine if the projectile is a ball. However, if the projectile is an arrow, the arrow will not "turn" properly in the air in the manner that arrows do. Unity will simply translate the arrow through space, without rotating it (see Fig. 3(a)). This raises the question, "How can we rotate the arrow to point in the direction it is traveling?" (see Fig. 3(b))

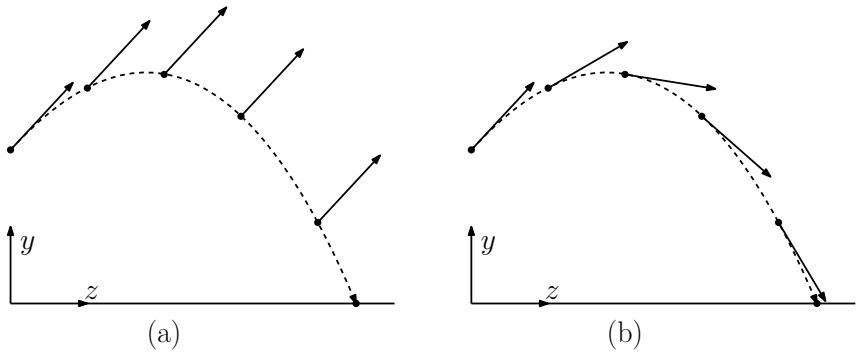


Fig. 3: Arrow shooting

This may seem to be a complicated question. Clearly, this is not a rotation about the axes, and so Euler angles would not be the right approach. We need to use quaternions. We want to specify a quaterion that will cause the arrow to rotate in the direction it is moving. Luckily for us, Unity provides a function that does almost what we need. The Unity function Quaternion.LookRotation(Vector3 forward) generates a rotation (represented as a quaternion) that aligns the forward ($z$) axis with the argument. Thus, to align the arrow with the direction it is heading, we can use the following Unity commands:

```
RigidBody rb = getComponent<RigidBody> ();
transform.rotation = Quaternion.LookRotation (rb.velocity);
```

This will rotate the object so its orientation matches its velocity, as desired.

**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. 4(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.)

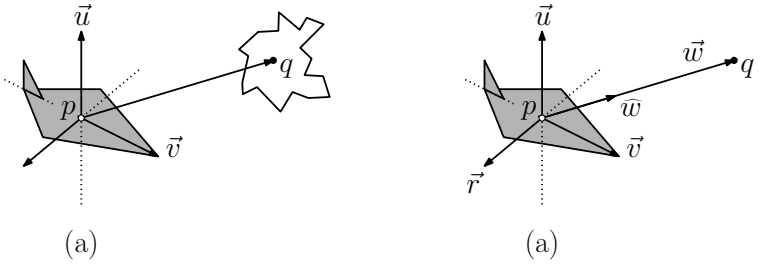


Fig. 4: Evasive action.

**Solution:** First observe that $\vec{w} \leftarrow q - p$ defines a vector that is directed from the space ship to the obstacle (see Fig. 4(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

$$\begin{aligned} \vec{w} &\leftarrow q - p \\ \widehat{w} &\leftarrow \text{normalize}(\vec{w}) = \frac{\vec{w}}{\|\vec{w}\|} = \frac{\vec{w}}{\sqrt{\vec{w} \cdot \vec{w}}} = \frac{\vec{w}}{\sqrt{w_x^2 + w_y^2 + w_z^2}}. \end{aligned}$$

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 $\widehat{w}$. 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

$$\begin{aligned} \widehat{w} \cdot \vec{u} &\geq 0 \quad \Rightarrow \quad \text{(obstacle above) pitch downwards} \\ \widehat{w} \cdot \vec{u} &< 0 \quad \Rightarrow \quad \text{(obstacle below) pitch upwards.} \end{aligned}$$

(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 $\widehat{w}$ 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 $\widehat{w}$ and $\vec{r}$ is smaller than 90°, 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

$$\begin{aligned} \vec{r} &\leftarrow \vec{v} \times \vec{u} \\ \widehat{w} \cdot \vec{r} \geq 0 &\Rightarrow \text{(obstacle to the right) yaw to the left} \\ \widehat{w} \cdot \vec{r} < 0 &\Rightarrow \text{(obstacle to the left) yaw to the right.} \end{aligned}$$

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

Turning a ship at position $p$ to avoid obstacle at $q$

```
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.