Conclusion

The steering pipeline is one of many possible cooperative arbitration mechanisms. Unlike other approaches, such as decision trees or blackboard architectures, it is specifically designed for the needs of steering.

On the other hand, it is not the most efficient technique. While it will run very quickly for simple scenarios, it can slow down when the situation gets more complex. If you are determined for your characters to move intelligently, then you will have to pay the price in execution speed sooner or later (in fact, to guarantee it, you'll need full motion planning, which is even slower than pipeline steering). In many games, however, the prospect of some foolish steering is not a major issue, and it may be easier to use a simpler approach to combining steering behaviors, such as blending.

3.5 Predicting Physics

A common requirement of AI in 3D games is to interact well with some kind of physics simulation. This may be as simple as the AI in variations of Pong, that tracked the current position of the ball and moved the bat so that it intercepted the ball, or it might involve the character correctly calculating the best way to throw a ball so that it reaches a teammate who is running. We've seen examples of this already. The pursue steering behavior predicted the future position of its target by assuming it would carry on with its current velocity. At its most complex, it may involve deciding where to stand to minimize the chance of being hit by an incoming grenade.

In each case, we are doing AI not based on the character's own movement (although that may be a factor), but on the basis of other characters' or objects' movement.

By far, the most common requirement for predicting movement is for aiming and shooting firearms. This involves the solution of ballistic equations: the so-called "Firing Solution." In this section we will first look at firing solutions and the mathematics behind them. We will then look at the broader requirements of predicting trajectories and a method of iteratively predicting objects with complex movement patterns.

3.5.1 Aiming and Shooting

Firearms, and their fantasy counterparts, are a key feature of game design. In almost any game you choose to think of, the characters can wield some variety of projectile weapon. In a fantasy game it might be a crossbow or fireball spell, and in a science fiction (sci-fi) game it could be a disrupter or phaser.

This puts two common requirements on the AI. Characters should be able to shoot accurately, and they should be able to respond to incoming fire. The second requirement is often omitted, since the projectiles from many firearms and sci-fi

weapons move too fast for anyone to be able to react to. When faced with weapons such as RPGs or mortars, however, the lack of reaction can appear unintelligent.

Regardless of whether a character is giving or receiving fire, it needs to understand the likely trajectory of a weapon. For fast-moving projectiles over small distances, this can be approximated by a straight line, so older games tended to use simple straight line tests for shooting. With the introduction of increasingly complex physics simulation, however, shooting along a straight line to your targets is likely to see your bullets in the dirt at their feet. Predicting correct trajectories is now a core part of the AI in shooters.

3.5.2 Projectile Trajectory

A moving projectile under gravity will follow a curved trajectory. In the absence of any air resistance or other interference, the curve will be part of a parabola, shown in Figure 3.45.

The projectile moves according to the formula

$$\vec{p}_t = \vec{p}_0 + \vec{u}s_m t + \frac{\vec{g}t^2}{2} \qquad [3.1]$$

where $\vec{p}_t$ is its position (in three dimensions) at time t, $\vec{p}_0$ is the firing position (again in three dimensions), s_m is the muzzle velocity (the speed the projectile left the weapon—it is not strictly a velocity because it is not a vector), $\vec{u}$ is the direction the weapon was fired in (a normalized 3D vector), t is the length of time since the shot was fired, and $\vec{g}$ is the acceleration due to gravity. The notation $\vec{x}$ denotes that x is a vector. Others values are scalar.

It is worth noting that although the acceleration due to gravity on earth is

$$\vec{g} = \begin{bmatrix} 0 \\ -9.81 \\ 0 \end{bmatrix} \text{ms}^{-2}$$

(i.e., 9.81 ms^{-2} in the down direction), this can look too slow in a game environment. Physics middleware vendors such as Havok recommend using a value

Figure 3.45 Parabolic arc

around double that for games, although some tweaking is needed to get the exact look.

The simplest thing we can do with the trajectory equations is to determine if a character will be hit by an incoming projectile. This is a fairly fundamental requirement of any character in a shooter with slow-moving projectiles (such as grenades).

We will split this into two elements: determining where a projectile will land and determining if its trajectory will touch the character.

Predicting a Landing Spot

The AI should determine where an incoming grenade will land and then move quickly away from that point (using a flee steering behavior, for example, or a more complex compound steering system that takes into account escape routes). If there's enough time, an AI might move toward the grenade point as fast as possible (using arrive, perhaps) and then intercept and throw back the ticking grenade, forcing the player to pull the grenade pin and hold it for just the right length of time.

We can determine where a grenade will land by solving the projectile equation for a fixed value of p_y (i.e., the height). If we know the current velocity of the grenade and its current position, we can solve for just the y component of the position and get the time at which the grenade will reach a known height (i.e., the height of the floor on which the character is standing):

$$t_i = \frac{-u_y s_m \pm \sqrt{u_y^2 s_m^2 - 2g_y(p_{y0} - p_{yt})}}{g_y}, \qquad [3.2]$$

where p_{yi} is the position of impact, and t_i is the time at which this occurs. There may be zero, one, or two solutions to this equation. If there are zero solutions, then the projectile never reaches the target height; it is always below it. If there is one solution, then the projectile reaches the target height at the peak of its trajectory. Otherwise, the projectile reaches the height once on the way up and once on the way down. We are interested in the solution when the projectile is descending, which will be the greater time value (since whatever goes up will later come down). If this time value is less than zero, then the projectile has already passed the target height and won't reach it again.

The time t_i from Equation 3.2 can be substituted into Equation 3.1 to get the complete position of impact:

$$\vec{p}_i = \begin{bmatrix} p_{x0} + u_x s_m t_i + \frac{1}{2} g_x t_i^2 \\ p_{yi} \\ p_{z0} + u_z s_m t_i + \frac{1}{2} g_z t_i^2 \end{bmatrix} \qquad [3.3]$$

which further simplifies, if (as it normally does) gravity only acts in the down direction, to

$$\vec{p}_i = \begin{bmatrix} p_{x0} + u_x s_m t_i \\ p_{yi} \\ p_{z0} + u_z s_m t_i \end{bmatrix}.$$

For grenades, we could compare the time to impact with the known length of the grenade fuse to determine whether it is safer to run from or catch and return the grenade.

Note that this analysis does not deal with the situation where the ground level is rapidly changing. If the character is on a ledge or walkway, for example, the grenade may miss impacting at its height entirely and sail down the gap behind it. We can use the result of Equation 3.3 to check if the impact point is valid.

For outdoor levels with rapidly fluctuating terrain, we can also use the equation iteratively, generating (x, z) coordinates with Equation 3.3 and then feeding the p_y coordinate of the impact point back into the equation, until the resulting (x, z) values stabilize. There is no guarantee that they will ever stabilize, but in most cases they do. In practice, however, high explosive projectiles typically damage a large area, so inaccuracies in the impact point prediction are difficult to spot when the character is running away.

The final point to note about incoming hit prediction is that the floor height of the character is not normally the height at which the character catches. If the character is intending to catch the incoming object (as it will in most sports games, for example), it should use a target height value at around chest height. Otherwise, it will appear to maneuver in such a way that the incoming object drops at its feet.

3.5.3 The Firing Solution

To hit a target at a given point $\vec{E}$, we need to solve Equation 3.1. In most cases we know the firing point $\vec{S}$ (i.e., $\vec{S} \equiv \vec{p}_0$), the muzzle velocity s_m, and the acceleration due to gravity $\vec{g}$; we'd like to find just $\vec{u}$, the direction to fire in (although finding the time to collision can also be useful for deciding if a slow-moving shot is worth it).

Archers and grenade throwers can change the velocity of the projectile as they fire (i.e., they select an s_m value), but most weapons have a fixed value for s_m. We will assume, however, that characters who can select a velocity will always try to get the projectile to its target in the shortest time possible. In this case they will always choose the highest possible velocity.

In an indoor environment with many obstacles (such as barricades, joists, and columns), it might be advantageous for a character to throw its grenade more slowly so that it arches over obstacles. Dealing with obstacles in this way gets to be very complex and is best solved by a trial and error process, trying different s_m values (normally trials are limited to a few fixed values: "throw fast," "throw slow," and "drop," for example). For the purpose of this book, we'll assume that s_m is constant and known in advance.

The quadratic Equation 3.1 has vector coefficients. Add the requirement that the firing vector should be normalized,

$$|\vec{u}| = 1,$$

and we have four equations in four unknowns:

$$\begin{aligned} E_x &= S_x + u_x s_m t_i + \frac{1}{2} g_x t_i^2, \\ E_y &= S_y + u_y s_m t_i + \frac{1}{2} g_y t_i^2, \\ E_z &= S_z + u_z s_m t_i + \frac{1}{2} g_z t_i^2, \\ 1 &= u_x^2 + u_y^2 + u_z^2. \end{aligned}$$

These can be solved to find the firing direction and the projectile's time to target. First, we get an expression for t_i:

$$|\vec{g}|^2 t_i^4 - 4\left(\vec{g}.\vec{\Delta} + s_m^2\right) t_i^2 + 4|\vec{\Delta}|^2 = 0,$$

where $\vec{\Delta}$ is the vector from the start point to the end point, given by $\vec{\Delta} = \vec{E} - \vec{S}$. This is a quartic in t_i, with no odd powers. We can therefore use the quadratic equation formula to solve for t_i^2 and take the square root of the result. Doing this, we get

$$t_i = +2\sqrt{\frac{\vec{g}.\vec{\Delta} + s_m^2 \pm \sqrt{(\vec{g}.\vec{\Delta} + s_m^2)^2 - |\vec{g}|^2 |\vec{\Delta}|^2}}{2|\vec{g}|^2}}$$

which gives us two real-valued solutions for time, of which a maximum of two may be positive. Note that we should strictly take into account the two negative solutions also (replacing the positive sign with a negative sign before the first square root). We omit these because solutions with a negative time are entirely equivalent to aiming in exactly the opposite direction to get a solution in positive time.

There are no solutions if

$$\left(\vec{g}.\vec{\Delta} + s_m^2\right)^2 < |\vec{g}|^2 |\vec{\Delta}|^2.$$

In this case the target point cannot be hit with the given muzzle velocity from the start point. If there is one solution, then we know the end point is at the absolute limit of the given firing capabilities. Usually, however, there will be two solutions, with different arcs to the target. This is illustrated in Figure 3.46. We will almost always choose the lower arc, which has the smaller time value, since it gives the target less time to react to the incoming projectile and produces a shorter arc that is less likely to hit obstacles (especially the ceiling).

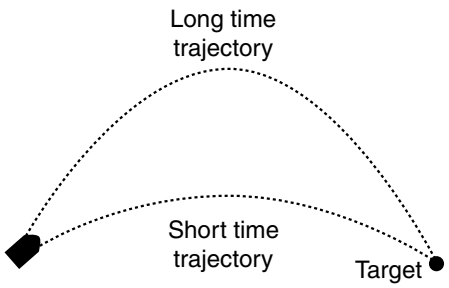

Figure 3.46 Two possible firing solutions

We might want to choose the longer arc if we are firing over a wall, in a castle-strategy game, for example.

With the appropriate t_i value selected, we can determine the firing vector using the equation

$$\vec{u} = \frac{2\vec{\Delta} - \vec{g}t_i^2}{2s_m t_i}.$$

The intermediate derivations of these equations are left as an exercise.

This is admittedly a mess to look at, but can be easily implemented as follows:

```
def calculateFiringSolution(start, end, muzzle_v, gravity):

  # Calculate the vector from the target back to the start
  delta = start - end

  # Calculate the real-valued a,b,c coefficients of a conventional
  # quadratic equation
  a = gravity * gravity
  b = -4 * (gravity * delta + muzzle_v*muzzle_v)
  c = 4 * delta * delta

  # Check for no real solutions
  if 4*a*c > b*b: return None

  # Find the candidate times
  time0 = sqrt((-b + sqrt(b*b-4*a*c)) / (2*a))
  time1 = sqrt((-b - sqrt(b*b-4*a*c)) / (2*a))

  # Find the time to target
  if times0 < 0:
```

```
    if times1 < 0:
      # We have no valid times
      return None
    else:
      ttt = times1
  else:
    if times1 < 0:
      ttt = times0
    else:
      ttt = min(times0, times1)

  # Return the firing vector
  return (2 * delta - gravity * ttt*ttt) / (2 * muzzle_v * ttt)
```

This code assumes that we can take the scalar product of two vectors using the `a * b` notation. The algorithm is O(1) in both memory and time. There are optimizations to be had, and the C++ source code on the CD contains an implementation of this function where the math has been automatically optimized by a commercial equation to code converter for added speed.

LIBRARY

3.5.4 Projectiles with Drag

The situation becomes more complex if we introduce air resistance. Because it adds complexity, it is very common to see developers ignoring drag altogether for calculating firing solutions. Often, a drag-free implementation of ballistics is a perfectly acceptable approximation. Once again, the gradual move toward including drag in trajectory calculations is motivated by the use of physics engines. If the physics engine includes drag (and most of them do to avoid numerical instability problems), then a drag-free ballistic assumption can lead to inaccurate firing over long distances. It is worth trying an implementation without drag, however, even if you are using a physics engine. Often, the results will be perfectly usable and much simpler to implement.

The trajectory of a projective moving under the influence of drag is no longer a parabolic arc. As the projectile moves, it slows down, and its overall path looks like Figure 3.47.

Adding drag to the firing calculations considerably complicates the mathematics, and for this reason most games either ignore drag in their firing calculations or use a kind of trial and error process that we'll look at in more detail later.

Although drag in the real world is a complex process caused by many interacting factors, drag in computer simulation is often dramatically simplified. Most physics engines relate the drag force to the speed of a body's motion with components related to either velocity or velocity squared or both. The drag force on a body, D, is given