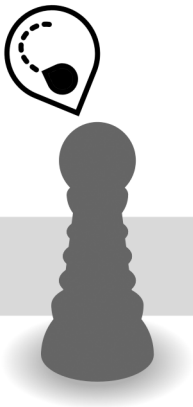

3

MOVEMENT

One of the most fundamental requirements of game AI is to sensibly move characters around. Even the earliest AI-controlled characters (the ghosts in *Pac-Man*, for example, or the opposing bat in some *Pong* variants) had movement algorithms that weren't far removed from modern games.

Movement forms the lowest level of AI techniques in our model, shown in Figure 3.1.

Many games, including some with quite decent-looking AI, rely solely on movement algorithms and don't have any more advanced decision making. At the other extreme, some games don't need moving characters at all. Resource management games and turn-based games often don't need movement algorithms; once a decision is made where to move, the character can simply be placed there.

There is also some degree of overlap between AI and animation; animation is also about movement. This chapter looks at large-scale movement: the movement of characters around the game level, rather than the movement of their limbs or faces. The dividing line isn't always clear, however. In many games, animation can take control over a character, including some large-scale movement. This may be as simple as the character moving a few steps to pull a lever. Or as complex as mini cut-scenes, completely animated, that seamlessly transition into and out of gameplay. These are not AI driven and therefore aren't covered here.

This chapter will look at a range of different AI-controlled movement algorithms, from the simple Pac-Man level up to the complex steering behaviors used for driving a racing car or piloting a spaceship in three dimensions.

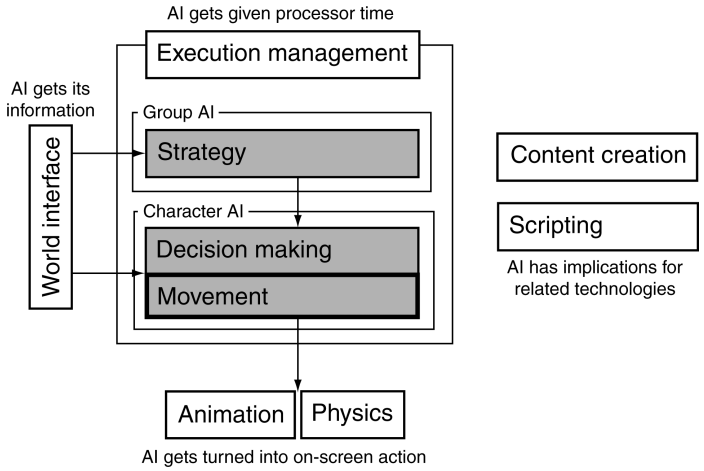

Figure 3.1: The AI model

3.1 THE BASICS OF MOVEMENT ALGORITHMS

Unless you're writing an economic simulator, chances are the characters in your game need to move around. Each character has a current position and possibly additional physical properties that control its movement. A movement algorithm is designed to use these properties to work out where the character should be next.

All movement algorithms have this same basic form. They take geometric data about their own state and the state of the world, and they come up with a geometric output representing the movement they would like to make. Figure 3.2 shows this schematically. In the figure, the velocity of a character is shown as optional because it is only needed for certain classes of movement algorithms.

Some movement algorithms require very little input: just the position of the character and the position of an enemy to chase, for example. Others require a lot of interaction with the game state and the level geometry. A movement algorithm that avoids bumping into walls, for example, needs to have access to the geometry of the wall to check for potential collisions.

The output can vary too. In most games it is normal to have movement algorithms output a desired velocity. A character might see its enemy to the west, for example, and respond that its movement should be westward at full speed. Often, characters in older games only had two speeds: stationary and running (with maybe a walk speed in there, too, for patrolling). So the output was simply a direction to move in. This is kinematic movement; it does not account for how characters accelerate and slow down.

It is common now for more physical properties to be taken into account. Producing movement algorithms I will call "steering behaviors." Steering behaviors is the name given by Craig

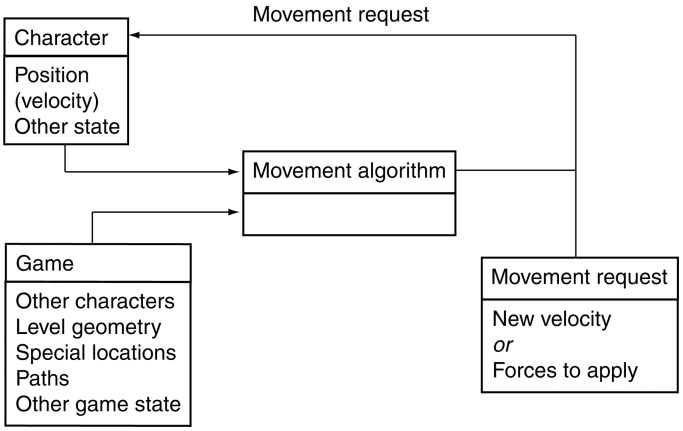

Figure 3.2: The movement algorithm structure

Reynolds to his movement algorithms [51]; they are not kinematic, but dynamic. Dynamic movement takes account of the current motion of the character. A dynamic algorithm typically needs to know the current velocities of the character as well as its position. A dynamic algorithm outputs forces or accelerations with the aim of changing the velocity of the character.

Dynamics adds an extra layer of complexity. Let's say your character needs to move from one place to another. A kinematic algorithm simply gives the direction to the target; your character moves in that direction until it arrives, whereupon the algorithm returns no direction. A dynamic movement algorithm needs to work harder. It first needs to accelerate in the right direction, and then as it gets near its target it needs to accelerate in the opposite direction, so its speed decreases at precisely the correct rate to slow it to a stop at exactly the right place. Because Craig's work is so well known, in the rest of this chapter I'll usually follow the most common terminology and refer to all dynamic movement algorithms as steering behaviors.

Craig Reynolds also invented the flocking algorithm used in countless films and games to animate flocks of birds or herds of other animals. We'll look at this algorithm later in the chapter. Because flocking is the most famous steering behavior, all steering algorithms (in fact, all movement algorithms) are sometimes wrongly called "flocking."

3.1.1 TWO-DIMENSIONAL MOVEMENT

Many games have AI that works in two dimensions (2D). Even games that are rendered in three dimensions (3D), usually have characters that are under the influence of gravity, sticking them to the floor and constraining their movement to two dimensions.

A lot of movement AI can be achieved in 2D, and most of the classic algorithms are only

defined for this case. Before looking at the algorithms themselves, we need to quickly cover the data needed to handle 2D math and movement.

Characters as Points

Although a character usually consists of a 3D model that occupies some space in the game world, many movement algorithms assume that the character can be treated as a single point. Collision detection, obstacle avoidance, and some other algorithms use the size of the character to influence their results, but movement itself assumes the character is at a single point.

This is a process similar to that used by physics programmers who treat objects in the game as a "rigid body" located at its center of mass. Collision detection and other forces can be applied anywhere on the object, but the algorithm that determines the movement of the object converts them so it can deal only with the center of mass.

3.1.2 STATICS

Characters in two dimensions have two linear coordinates representing the position of the object. These coordinates are relative to two world axes that lie perpendicular to the direction of gravity and perpendicular to each other. This set of reference axes is termed the *orthonormal basis of the 2D space*.

In most games the geometry is typically stored and rendered in three dimensions. The geometry of the model has a 3D orthonormal basis containing three axes: normally called x, y, and z. It is most common for the y-axis to be in the opposite direction to gravity (i.e., "up") and for the x and z axes to lie in the plane of the ground. Movement of characters in the game takes place along the x and z axes used for rendering, as shown in Figure 3.3. For this reason this chapter will use the x and z axes when representing movement in two dimensions, even though books dedicated to 2D geometry tend to use x and y for the axis names.

In addition to the two linear coordinates, an object facing in any direction has one orientation value. The orientation value represents an angle from a reference axis. In our case we use a counterclockwise angle, in radians, from the positive z-axis. This is fairly standard in game engines; by default (i.e., with zero orientation a character is looking down the z-axis).

With these three values the static state of a character can be given in the level, as shown in Figure 3.4.

Algorithms or equations that manipulate this data are called *static* because the data do not contain any information about the movement of a character.

We can use a data structure of the form:

```
class Static:
    position: Vector
    orientation: float
```

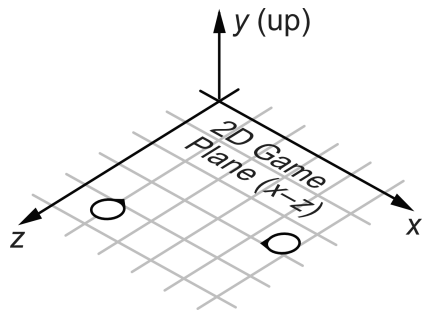

Figure 3.3: The 2D movement axes and the 3D basis

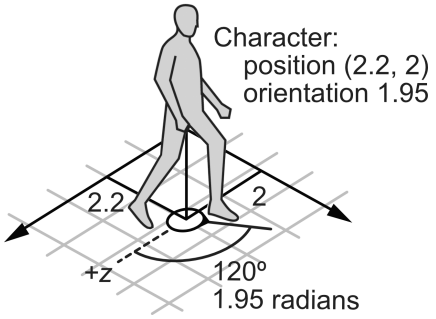

Figure 3.4: The positions of characters in the level

I will use the term *orientation* throughout this chapter to mean the direction in which a character is facing. When it comes to rendering characters, we will make them appear to face one direction by rotating them (using a rotation matrix). Because of this, some developers refer to orientation as *rotation*. I will use rotation in this chapter only to mean the process of changing orientation; it is an active process.

2½ Dimensions

Some of the math involved in 3D geometry is complicated. The linear movement in three dimensions is quite simple and a natural extension of 2D movement, but representing an orientation has tricky consequences that are better to avoid (at least until the end of the chapter).

As a compromise, developers often use a hybrid of 2D and 3D calculations which is known as 2½D, or sometimes four degrees of freedom.

In 2½ dimensions we deal with a full 3D position but represent orientation as a single value, as if we are in two dimensions. This is quite logical when you consider that most games involve characters under the influence of gravity. Most of the time a character's third dimension is constrained because it is pulled to the ground. In contact with the ground, it is effectively operating in two dimensions, although jumping, dropping off ledges, and using elevators all involve movement through the third dimension.

Even when moving up and down, characters usually remain upright. There may be a slight tilt forward while walking or running or a lean sideways out from a wall, but this tilting doesn't affect the movement of the character; it is primarily an animation effect. If a character remains upright, then the only component of its orientation we need to worry about is the rotation about the up direction.

This is precisely the situation we take advantage of when we work in 2½D, and the simplification in the math is worth the decreased flexibility in most cases.

Of course, if you are writing a flight simulator or a space shooter, then all the orientations are very important to the AI, so you'll have to support the mathematics for three-dimensional orientation. At the other end of the scale, if your game world is completely flat and characters can't jump or move vertically in any other way, then a strict 2D model is needed. In the vast majority of cases, 2½D is an optimal solution. I'll cover full 3D motion at the end of the chapter, but aside from that, all the algorithms described in this chapter are designed to work in 2½D.

Math

In the remainder of this chapter I will assume that you are comfortable using basic vector and matrix mathematics (i.e., addition and subtraction of vectors, multiplication by a scalar). Explanations of vector and matrix mathematics, and their use in computer graphics, are beyond the scope of this book. If you can find it, the older book Schneider and Eberly [56], covers mathematical topics in computer games to a much deeper level. An excellent, and more recent alternative is [36].

Positions are represented as a vector with x and z components of position. In 2½D, a y component is also given.

In two dimensions we need only an angle to represent orientation. The angle is measured from the positive z-axis, in a right-handed direction about the positive y-axis (counterclockwise as you look down on the x–z plane from above). Figure 3.4 gives an example of how the scalar orientation is measured.

It is more convenient in many circumstances to use a vector representation of orientation. In this case the vector is a unit vector (it has a length of one) in the direction that the character is facing. This can be directly calculated from the scalar orientation using simple trigonometry:

$$\vec{\omega_v} = \begin{bmatrix} \sin \omega_s \\ \cos \omega_s \end{bmatrix}$$

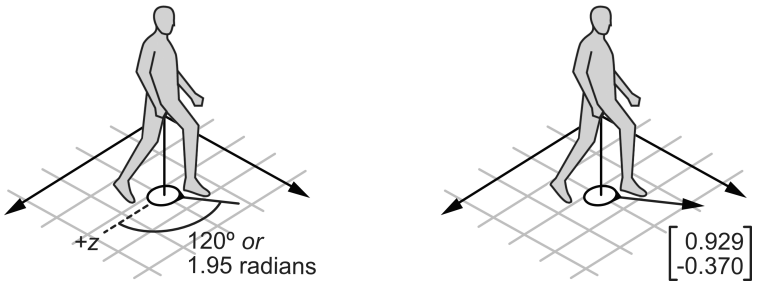

Figure 3.5: The vector form of orientation

where ω_s is the orientation as a scalar (i.e. a single number representing the angle), and $\vec{\omega_v}$ is the orientation expressed as a vector. I am assuming a right-handed coordinate system here, in common with most of the game engines I've worked on,[1] if you use a left-handed system, then simply flip the sign of the x coordinate:

$$\vec{\omega_v} = \begin{bmatrix} -\sin\omega_s \\ \cos\omega_s \end{bmatrix}$$

If you draw the vector form of the orientation, it will be a unit length vector in the direction that the character is facing, as shown in Figure 3.5.

3.1.3 KINEMATICS

So far, each character has two associated pieces of information: its position and its orientation. We can create movement algorithms to calculate a target velocity based on position and orientation alone, allowing the output velocity to change instantly.

While this is fine for many games, it can look unrealistic. A consequence of Newton's laws of motion is that velocities cannot change instantly in the real world. If a character is moving in one direction and then instantly changes direction or speed, it may look odd. To make smooth motion or to cope with characters that can't accelerate very quickly, we need either to use some kind of smoothing algorithm or to take account of the current velocity and use accelerations to change it.

To support this, the character keeps track of its current velocity as well as position. Algorithms can then operate to change the velocity slightly at each frame, giving a smooth motion.

Characters need to keep track of both their linear and their angular velocities. Linear velocity has both x and z components, the speed of the character in each of the axes in the

1. Left-handed coordinates work just as well with all the algorithms in this chapter. See [13] for more details of the difference, and how to convert between them.

orthonormal basis. If we are working in 2½D, then there will be three linear velocity components, in x, y, and z.

The angular velocity represents how fast the character's orientation is changing. This is given by a single value: the number of radians per second that the orientation is changing.

I will call angular velocity *rotation*, since rotation suggests motion. Linear velocity will normally be referred to as simply velocity. We can therefore represent all the kinematic data for a character (i.e., its movement and position) in one structure:

```
class Kinematic:
    position: Vector
    orientation: float
    velocity: Vector
    rotation: float
```

Steering behaviors operate with these kinematic data. They return accelerations that will change the velocities of a character in order to move them around the level. Their output is a set of accelerations:

```
class SteeringOutput:
    linear: Vector
    angular: float
```

Independent Facing

Notice that there is nothing to connect the direction that a character is moving and the direction it is facing. A character can be oriented along the x-axis but be traveling directly along the z-axis. Most game characters should not behave in this way; they should orient themselves so they move in the direction they are facing.

Many steering behaviors ignore facing altogether. They operate directly on the linear components of the character's data. In these cases the orientation should be updated so that it matches the direction of motion.

This can be achieved by directly setting the orientation to the direction of motion, but doing so may mean the orientation changes abruptly.

A better solution is to move it a proportion of the way toward the desired direction: to smooth the motion over many frames. In Figure 3.6, the character changes its orientation to be halfway toward its current direction of motion in each frame. The triangle indicates the orientation, and the gray shadows show where the character was in previous frames, to illustrate its motion.

Updating Position and Orientation

If your game uses a physics engine, it may be used to update the position and orientation of characters. Even with a physics engine, some developers prefer to use only its collision

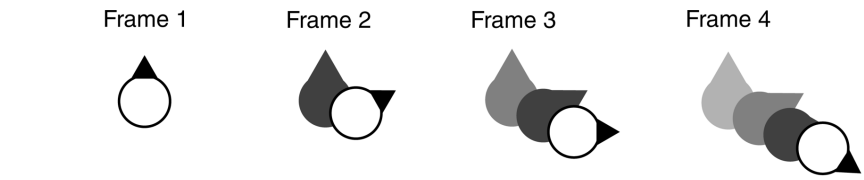

Figure 3.6: Smoothing facing direction of motion over multiple frames

detection on characters, and code custom movement controllers. So if you need to update position and orientation manually, you can use a simple algorithm of the form:

```
class Kinematic:
    # ... Member data as before ...

    function update(steering: SteeringOutput, time: float):
        # Update the position and orientation.
        half_t_sq: float = 0.5 * time * time
        position += velocity * time + steering.linear * half_t_sq
        orientation += rotation * time + steering.angular * half_t_sq

        # and the velocity and rotation.
        velocity += steering.linear * time
        rotation += steering.angular * time
```

The updates use high-school physics equations for motion. If the frame rate is high, then the update time passed to this function is likely to be very small. The square of this time is likely to be even smaller, and so the contribution of acceleration to position and orientation will be tiny. It is more common to see these terms removed from the update algorithm, to give what's known as the Newton-Euler-1 integration update:

```
class Kinematic:
    # ... Member data as before ...

    function update(steering: SteeringOutput, time: float):
        # Update the position and orientation.
        position += velocity * time
        orientation += rotation * time

        # and the velocity and rotation.
        velocity += steering.linear * time
        rotation += steering.angular * time
```

This is the most common update used for games. Note that, in both blocks of code I've assumed that we can do normal mathematical operations with vectors, such as addition and

multiplication by a scalar. Depending on the language you are using, you may have to replace these primitive operations with function calls.

The *Game Physics* [12] book in the Morgan Kaufmann Interactive 3D Technology series, and my *Game Physics Engine Development* [40], also in that series, have a complete analysis of different update methods and cover a more complete range of physics tools for games (as well as detailed implementations of vector and matrix operations).

Variable Frame Rates

So far I have assumed that velocities are given in units per second rather than per frame. Older games often used per-frame velocities, but that practice largely died out, before being resurrected occasionally when gameplay began to be processed in a separate execution thread with a fixed frame rate, separate to graphics updates (Unity's `FixedUpdate` function compared to its variable `Update`, for example). Even with these functions available, it is more flexible to support variable frame rates, in case the fixed update rate needs changing. For this reason I will continue to use an explicit update time.

If the character is known to be moving at 1 meter per second and the last frame was of 20 milliseconds' duration, then they will need to move 20 millimeters.

Forces and Actuation

In the real world we can't simply apply an acceleration to an object and have it move. We apply forces, and the forces cause a change in the kinetic energy of the object. They will accelerate, of course, but the acceleration will depend on the inertia of the object. The inertia acts to resist the acceleration; with higher inertia, there is less acceleration for the same force.

To model this in a game, we could use the object's mass for the linear inertia and the moment of inertia (or inertia tensor in three dimensions) for angular acceleration.

We could continue to extend the character data to keep track of these values and use a more complex update procedure to calculate the new velocities and positions. This is the method used by physics engines: the AI controls the motion of a character by applying forces to it. These forces represent the ways in which the character can affect its motion. Although not common for human characters, this approach is almost universal for controlling cars in driving games: the drive force of the engine and the forces associated with the steering wheels are the only ways in which the AI can control the movement of the car.

Because most well-established steering algorithms are defined with acceleration outputs, it is not common to use algorithms that work directly with forces. Usually, the movement controller considers the dynamics of the character in a post-processing step called *actuation*.

Actuation takes as input a desired change in velocity, the kind that would be directly applied in a kinematic system. The actuator then calculates the combination of forces that it can apply to get as near as possible to the desired velocity change.

At the simplest level this is just a matter of multiplying the acceleration by the inertia to

give a force. This assumes that the character is capable of applying any force, however, which isn't always the case (a stationary car can't accelerate sideways, for example). Actuation is a major topic in AI and physics integration, and I'll return to actuation at some length in Section 3.8 of this chapter.

3.2 KINEMATIC MOVEMENT ALGORITHMS

Kinematic movement algorithms use static data (position and orientation, no velocities) and output a desired velocity. The output is often simply an on or off and a target direction, moving at full speed or being stationary. Kinematic algorithms do not use acceleration, although the abrupt changes in velocity might be smoothed over several frames.

Many games simplify things even further and force the orientation of a character to be in the direction it is traveling. If the character is stationary, it faces either a pre-set direction or the last direction it was moving in. If its movement algorithm returns a target velocity, then that is used to set its orientation.

This can be done simply with the function:

```
function newOrientation(current: float, velocity: Vector) -> float:
    # Make sure we have a velocity.
    if velocity.length( ) > 0:
        # Calculate orientation from the velocity.
        return atan2(-static.x, static.z)

    # Otherwise use the current orientation.
    else:
        return current
```

We'll look at two kinematic movement algorithms: seeking (with several of its variants) and wandering. Building kinematic movement algorithms is extremely simple, so we'll only look at these two as representative samples before moving on to dynamic movement algorithms, the bulk of this chapter.

I can't stress enough, however, that this brevity is not because they are uncommon or unimportant. Kinematic movement algorithms still form the bread and butter of movement systems in a lot of games. The dynamic algorithms in the rest of the book are widespread, where character movement is controlled by a physics engine, but they are still not ubiquitous.

3.2.1 SEEK

A kinematic seek behavior takes as input the character's static data and its target's static data. It calculates the direction from the character to the target and requests a velocity along this line. The orientation values are typically ignored, although we can use the `newOrientation` function above to face in the direction we are moving.

The algorithm can be implemented in a few lines:

```
class KinematicSeek:
    character: Static
    target: Static

    maxSpeed: float

    function getSteering( ) -> KinematicSteeringOutput:
        result = new KinematicSteeringOutput( )

        # Get the direction to the target.
        result.velocity = target.position - character.position

        # The velocity is along this direction, at full speed.
        result.velocity.normalize( )
        result.velocity *= maxSpeed

        # Face in the direction we want to move.
        character.orientation = newOrientation(
            character.orientation,
            result.velocity)

        result.rotation = 0
        return result
```

where the `normalize` method applies to a vector and makes sure it has a length of one. If the vector is a zero vector, then it is left unchanged.

Data Structures and Interfaces

We use the `Static` data structure as defined at the start of the chapter and a `KinematicSteeringOutput` structure for output. The `KinematicSteeringOutput` structure has the following form:

```
class KinematicSteeringOutput:
    velocity: Vector
    rotation: float
```

In this algorithm rotation is never used; the character's orientation is simply set based on their movement. You could remove the call to `newOrientation` if you want to control orientation independently somehow (to have the character aim at a target while moving, as in *Tomb Raider* [95], or twin-stick shooters such as *The Binding of Isaac* [151], for example).

Performance

The algorithm is O(1) in both time and memory.

Flee

If we want the character to run away from the target, we can simply reverse the second line of the `getSteering` method to give:

```
# Get the direction away from the target.
steering.velocity = character.position - target.position
```

The character will then move at maximum velocity in the opposite direction.

Arriving

The algorithm above is intended for use by a chasing character; it will never reach its goal, but it continues to seek. If the character is moving to a particular point in the game world, then this algorithm may cause problems. Because it always moves at full speed, it is likely to overshoot an exact spot and wiggle backward and forward on successive frames trying to get there. This characteristic wiggle looks unacceptable. We need to end by standing stationary at the target spot.

To achieve this we have two choices. We can just give the algorithm a large radius of satisfaction and have it be satisfied if it gets closer to its target than that. Alternatively, if we support a range of movement speeds, then we could slow the character down as it reaches its target, making it less likely to overshoot.

The second approach can still cause the characteristic wiggle, so we benefit from blending both approaches. Having the character slow down allows us to use a much smaller radius of satisfaction without getting wiggle and without the character appearing to stop instantly.

We can modify the seek algorithm to check if the character is within the radius. If so, it doesn't worry about outputting anything. If it is not, then it tries to reach its target in a fixed length of time. (I've used a quarter of a second, which is a reasonable figure; you can tweak the value if you need to.) If this would mean moving faster than its maximum speed, then it moves at its maximum speed. The fixed time to target is a simple trick that makes the character slow down as it reaches its target. At 1 unit of distance away it wants to travel at 4 units per second. At a quarter of a unit of distance away it wants to travel at 1 unit per second, and so on. The fixed length of time can be adjusted to get the right effect. Higher values give a more gentle deceleration, and lower values make the braking more abrupt.

The algorithm now looks like the following:

```
class KinematicArrive:
    character: Static
    target: Static
```

```
    maxSpeed: float

    # The satisfaction radius.
    radius: float

    # The time to target constant.
    timeToTarget: float = 0.25

    function getSteering( ) -> KinematicSteeringOutput:
        result = new KinematicSteeringOutput( )

        # Get the direction to the target.
        result.velocity = target.position - character.position

        # Check if we're within radius.
        if result.velocity.length( ) < radius:
            # Request no steering.
            return null

        # We need to move to our target, we'd like to
        # get there in timeToTarget seconds.
        result.velocity /= timeToTarget

        # If this is too fast, clip it to the max speed.
        if result.velocity.length( ) > maxSpeed:
            result.velocity.normalize( )
            result.velocity *= maxSpeed

        # Face in the direction we want to move.
        character.orientation = newOrientation(
            character.orientation,
            result.velocity)

        result.rotation = 0
        return result
```

I've assumed a `length` function that gets the length of a vector.

3.2.2 WANDERING

A kinematic wander behavior always moves in the direction of the character's current orientation with maximum speed. The steering behavior modifies the character's orientation, which allows the character to meander as it moves forward. Figure 3.7 illustrates this. The character is shown at successive frames. Note that it moves only forward at each frame (i.e., in the direction it was facing at the previous frame).

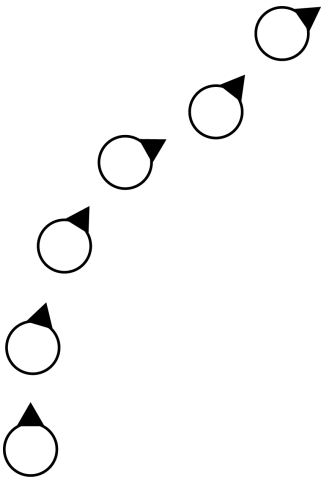

Figure 3.7: A character using kinematic wander

Pseudo-Code

It can be implemented as follows:

```
class KinematicWander:
    character: Static
    maxSpeed: float

    # The maximum rotation speed we'd like, probably should be smaller
    # than the maximum possible, for a leisurely change in direction.
    maxRotation: float

    function getSteering() -> KinematicSteeringOutput:
        result = new KinematicSteeringOutput()

        # Get velocity from the vector form of the orientation.
        result.velocity = maxSpeed * character.orientation.asVector()

        # Change our orientation randomly.
        result.rotation = randomBinomial() * maxRotation

        return result
```

Data Structures

Orientation values have been given an `asVector` function that converts the orientation into a direction vector using the formulae given at the start of the chapter.

Implementation Notes

I've used `randomBinomial` to generate the output rotation. This is a handy random number function that isn't common in the standard libraries of programming languages. It returns a random number between −1 and 1, where values around zero are more likely. It can be simply created as:

```
function randomBinomial() -> float:
    return random() - random()
```

where `random()` returns a random number from 0 to 1.

For our wander behavior, this means that the character is most likely to keep moving in its current direction. Rapid changes of direction are less likely, but still possible.

3.3 STEERING BEHAVIORS

Steering behaviors extend the movement algorithms in the previous section by adding velocity and rotation. In some genres (such as driving games) they are dominant; in other genres they are more context dependent: some characters need them, some characters don't.

There is a whole range of different steering behaviors, often with confusing and conflicting names. As the field has developed, no clear naming schemes have emerged to tell the difference between one atomic steering behavior and a compound behavior combining several of them together.

In this book I'll separate the two: fundamental behaviors and behaviors that can be built up from combinations of these.

There are a large number of named steering behaviors in various papers and code samples. Many of these are variations of one or two themes. Rather than catalog a zoo of suggested behaviors, we'll look at the basic structures common to many of them before looking at some exceptions with unusual features.

3.3.1 STEERING BASICS

By and large, most steering behaviors have a similar structure. They take as input the kinematic of the character that is moving and a limited amount of target information. The target information depends on the application. For chasing or evading behaviors, the target is often another moving character. Obstacle avoidance behaviors take a representation of the collision

geometry of the world. It is also possible to specify a path as the target for a path following behavior.

The set of inputs to a steering behavior isn't always available in an AI-friendly format. Collision avoidance behaviors, in particular, need to have access to collision information in the level. This can be an expensive process: checking the anticipated motion of the character using ray casts or trial movement through the level.

Many steering behaviors operate on a group of targets. The famous flocking behavior, for example, relies on being able to move toward the average position of the flock. In these behaviors some processing is needed to summarize the set of targets into something that the behavior can react to. This may involve averaging properties of the whole set (to find and aim for their center of mass, for example), or it may involve ordering or searching among them (such as moving away from the nearest predator or avoiding bumping into other characters that are on a collision course).

Notice that the steering behavior isn't trying to do everything. There is no behavior to avoid obstacles while chasing a character and making detours via nearby power-ups. Each algorithm does a single thing and only takes the input needed to do that. To get more complicated behaviors, we will use algorithms to combine the steering behaviors and make them work together.

3.3.2 VARIABLE MATCHING

The simplest family of steering behaviors operates by variable matching: they try to match one or more of the elements of the character's kinematic to a single target kinematic.

We might try to match the position of the target, for example, not caring about the other elements. This would involve accelerating toward the target position and decelerating once we are near. Alternatively, we could try to match the orientation of the target, rotating so that we align with it. We could even try to match the velocity of the target, following it on a parallel path and copying its movements but staying a fixed distance away.

Variable matching behaviors take two kinematics as input: the character kinematic and the target kinematic. Different named steering behaviors try to match a different combination of elements, as well as adding additional properties that control how the matching is performed.

It is possible, but not particularly helpful, to create a general steering behavior capable of matching any subset of variables and simply tell it which combination of elements to match. I've made the mistake of trying this implementation myself. The problem arises when more than one element of the kinematic is being matched at the same time. They can easily conflict. We can match a target's position and orientation independently. But what about position and velocity? If we are matching velocities, between a character and its target, then we can't simultaneously be trying to get any closer.

A better technique is to have individual matching algorithms for each element and then combine them in the right combination later. This allows us to use any of the steering behavior

combination techniques in this chapter, rather than having one hard-coded. The algorithms for combing steering behaviors are designed to resolve conflicts and so are perfect for this task.

For each matching steering behavior, there is an opposite behavior that tries to get as far away from matching as possible. A behavior that tries to catch its target has an opposite that tries to avoid its target; a behavior that tries to avoid colliding with walls has an opposite that hugs them (perhaps for use in the character of a timid mouse) and so on. As we saw in the kinematic seek behavior, the opposite form is usually a simple tweak to the basic behavior. We will look at several steering behaviors in a pair with their opposite, rather than separating them into separate sections.

3.3.3 SEEK AND FLEE

Seek tries to match the position of the character with the position of the target. Exactly as for the kinematic seek algorithm, it finds the direction to the target and heads toward it as fast as possible. Because the steering output is now an acceleration, it will accelerate as much as possible.

Obviously, if it keeps on accelerating, its speed will grow larger and larger. Most characters have a maximum speed they can travel; they can't accelerate indefinitely. The maximum can be explicit, held in a variable or constant, or it might be a function of speed-dependent drag, slowing the character down more strongly the faster it goes.

With an explicit maximum, the current speed of the character (the length of the velocity vector) is checked regularly, and is trimmed back if it exceeds the maximum speed. This is normally done as a post-processing step of the update function. It is not normally performed in a steering behavior. For example,

```
class Kinematic:
    # ... Member data as before ...

    function update(steering: SteeringOutput,
                    maxSpeed: float,
                    time: float):
        # Update the position and orientation.
        position += velocity * time
        orientation += rotation * time

        # and the velocity and rotation.
        velocity += steering.linear * time
        rotation += steering.angular * time

        # Check for speeding and clip.
        if velocity.length( ) > maxSpeed:
            velocity.normalize( )
            velocity *= maxSpeed
```

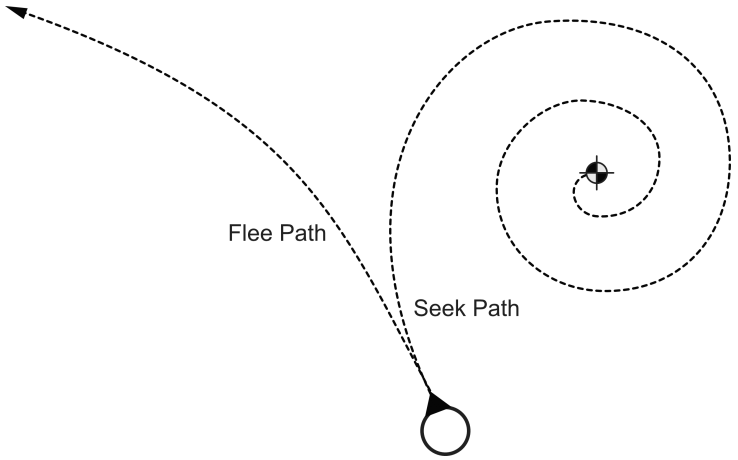

Figure 3.8: Seek and flee

Games that rely on physics engines typically include drag instead of a maximum speed (though they may use a maximum speed as well for safety). Without a maximum speed, the update does not need to check and clip the current velocity; the drag (applied in the physics update function) automatically limits the top speed.

Drag also helps another problem with this algorithm. Because the acceleration is always directed toward the target, if the target is moving, the seek behavior will end up orbiting rather than moving directly toward it. If there is drag in the system, then the orbit will become an inward spiral. If drag is sufficiently large, the player will not notice the spiral and will see the character simply move directly to its target.

Figure 3.8 illustrates the path that results from the seek behavior and its opposite, the flee path, described below.

Pseudo-Code

The dynamic seek implementation looks very similar to our kinematic version:

```
class Seek:
    character: Kinematic
    target: Kinematic

    maxAcceleration: float

    function getSteering( ) -> SteeringOutput:
        result = new SteeringOutput( )
```

```
9
10          # Get the direction to the target.
11          result.linear = target.position - character.position
12
13          # Give full acceleration along this direction.
14          result.linear.normalize()
15          result.linear *= maxAcceleration
16
17          result.angular = 0
18          return result
```

Note that I've removed the change in orientation that was included in the kinematic version. We can simply set the orientation, as we did before, but a more flexible approach is to use variable matching to make the character face in the correct direction. The align behavior, described below, gives us the tools to change orientation using angular acceleration. The "look where you're going" behavior uses this to face the direction of movement.

Data Structures and Interfaces

This class uses the `SteeringOutput` structure we defined earlier in the chapter. It holds linear and angular acceleration outputs.

Performance

The algorithm is again O(1) in both time and memory.

Flee

Flee is the opposite of seek. It tries to get as far from the target as possible. Just as for kinematic flee, we simply need to flip the order of terms in the second line of the function:

```
1 # Get the direction to the target.
2 steering.linear = character.position - target.position
```

The character will now move in the opposite direction of the target, accelerating as fast as possible.

3.3.4 ARRIVE

Seek will always move toward its goal with the greatest possible acceleration. This is fine if the target is constantly moving and the character needs to give chase at full speed. If the character

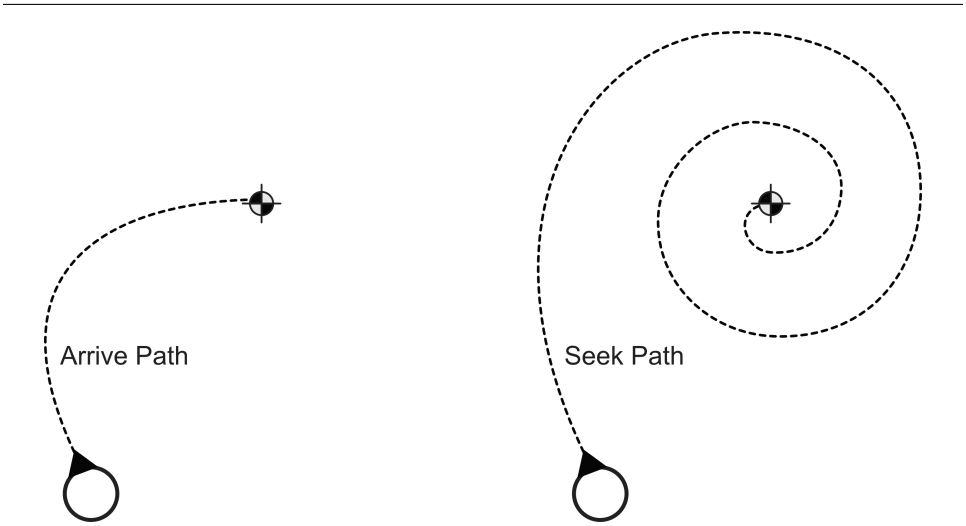

Figure 3.9: Seeking and arriving

arrives at the target, it will overshoot, reverse, and oscillate through the target, or it will more likely orbit around the target without getting closer.

If the character is supposed to arrive at the target, it needs to slow down so that it arrives exactly at the right location, just as we saw in the kinematic arrive algorithm. Figure 3.9 shows the behavior of each for a fixed target. The trails show the paths taken by seek and arrive. Arrive goes straight to its target, while seek orbits a bit and ends up oscillating. The oscillation is not as bad for dynamic seek as it was in kinematic seek: the character cannot change direction immediately, so it appears to wobble rather than shake around the target.

The dynamic arrive behavior is a little more complex than the kinematic version. It uses two radii. The arrival radius, as before, lets the character get near enough to the target without letting small errors keep it in motion. A second radius is also given, but is much larger. The incoming character will begin to slow down when it passes this radius. The algorithm calculates an ideal speed for the character. At the slowing-down radius, this is equal to its maximum speed. At the target point, it is zero (we want to have zero speed when we arrive). In between, the desired speed is an interpolated intermediate value, controlled by the distance from the target.

The direction toward the target is calculated as before. This is then combined with the desired speed to give a target velocity. The algorithm looks at the current velocity of the character and works out the acceleration needed to turn it into the target velocity. We can't immediately change velocity, however, so the acceleration is calculated based on reaching the target velocity in a fixed time scale.

This is exactly the same process as for kinematic arrive, where we tried to get the character

to arrive at its target in a quarter of a second. Because there is an extra layer of indirection (acceleration effects velocity which affects position) the fixed time period for dynamic arrive can usually be a little smaller; we'll use 0.1 as a good starting point.

When a character is moving too fast to arrive at the right time, its target velocity will be smaller than its actual velocity, so the acceleration is in the opposite direction—it is acting to slow the character down.

Pseudo-Code

The full algorithm looks like the following:

```
class Arrive:
    character: Kinematic
    target: Kinematic

    maxAcceleration: float
    maxSpeed: float

    # The radius for arriving at the target.
    targetRadius: float

    # The radius for beginning to slow down.
    slowRadius: float

    # The time over which to achieve target speed.
    timeToTarget: float = 0.1

    function getSteering( ) -> SteeringOutput:
        result = new SteeringOutput( )

        # Get the direction to the target.
        direction = target.position - character.position
        distance = direction.length( )

        # Check if we are there, return no steering.
        if distance < targetRadius:
            return null

        # If we are outside the slowRadius, then move at max speed.
        if distance > slowRadius:
            targetSpeed = maxSpeed
        # Otherwise calculate a scaled speed.
        else:
            targetSpeed = maxSpeed * distance / slowRadius

        # The target velocity combines speed and direction.
```

```
        targetVelocity = direction
        targetVelocity.normalize()
        targetVelocity *= targetSpeed

        # Acceleration tries to get to the target velocity.
        result.linear = targetVelocity - character.velocity
        result.linear /= timeToTarget

        # Check if the acceleration is too fast.
        if result.linear.length() > maxAcceleration:
            result.linear.normalize()
            result.linear *= maxAcceleration

        result.angular = 0
        return result
```

Performance

The algorithm is O(1) in both time and memory, as before.

Implementation Notes

Many implementations do not use a target radius. Because the character will slow down to reach its target, there isn't the same likelihood of oscillation that we saw in kinematic arrive. Removing the target radius usually makes no noticeable difference. It can be significant, however, with low frame rates or where characters have high maximum speeds and low accelerations. In general, it is good practice to give a margin of error around any target, to avoid annoying instabilities.

Leave

Conceptually, the opposite behavior of arrive is leave. There is no point in implementing it, however. If we need to leave a target, we are unlikely to want to accelerate with minuscule (possibly zero) acceleration first and then build up. We are more likely to accelerate as fast as possible. So for practical purposes the opposite of arrive is flee.

3.3.5 ALIGN

Align tries to match the orientation of the character with that of the target. It pays no attention to the position or velocity of the character or target. Recall that orientation is not directly

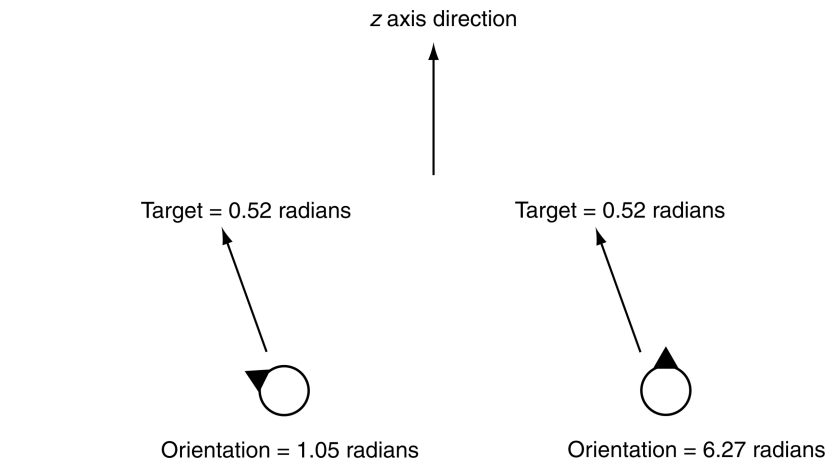

Figure 3.10: Aligning over a 2π radians boundary

related to direction of movement for a general kinematic. This steering behavior does not produce any linear acceleration; it only responds by turning.

Align behaves in a similar way to arrive. It tries to reach the target orientation and tries to have zero rotation when it gets there. Most of the code from arrive we can copy, but orientations have an added complexity that we need to consider.

Because orientations wrap around every 2π radians, we can't simply subtract the target orientation from the character orientation and determine what rotation we need from the result. Figure 3.10 shows two very similar align situations, where the character is the same angle away from its target. If we simply subtracted the two angles, the first one would correctly rotate a small amount clockwise, but the second one would travel all around to get to the same place.

To find the actual direction of rotation, we subtract the character orientation from the target and convert the result into the range $(-\pi, \pi)$ radians. We perform the conversion by adding or subtracting some multiple of 2π to bring the result into the given range. We can calculate the multiple to use by using the mod function and a little jiggling about. Most game engines or graphics libraries have one available (in Unreal Engine it is `FMath::FindDeltaAngle`, in Unity it is `Mathf.DeltaAngle`, though be aware that Unity uses degrees for its angles not radians).

We can then use the converted value to control rotation, and the algorithm looks very similar to arrive. Like arrive, we use two radii: one for slowing down and one to make orientations near the target acceptable. Because we are dealing with a single scalar value, rather than a 2D or 3D vector, the radius acts as an interval.

When we come to subtracting the rotation values, we have no problem we values repeat-

ing. Rotations, unlike orientations, don't wrap: a rotation of π is not the same as a rotational of 3π. In fact we can have huge rotation values, well out of the $(-\pi, \pi)$ range. Large values simply represent very fast rotation. Objects at very high speeds (such as the wheels in racing cars) may be rotating so fast that the updates we use here cause numeric instability. This is less of a problem on machines with 64-bit precision, but early physics on 32-bit machines often showed cars at high-speed having wheels that appeared to wobble. This has been resolved in robust physics engines, but it is worth knowing about, as the code we're discussing here is basically implementing a simple physics update. The simple approach is effective for moderate rotational speeds.

Pseudo-Code

Most of the algorithm is similar to arrive, we simply add the conversion:

```
class Align:
    character: Kinematic
    target: Kinematic

    maxAngularAcceleration: float
    maxRotation: float

    # The radius for arriving at the target.
    targetRadius: float

    # The radius for beginning to slow down.
    slowRadius: float

    # The time over which to achieve target speed.
    timeToTarget: float = 0.1

    function getSteering() -> SteeringOutput:
        result = new SteeringOutput()

        # Get the naive direction to the target.
        rotation = target.orientation - character.orientation

        # Map the result to the (-pi, pi) interval.
        rotation = mapToRange(rotation)
        rotationSize = abs(rotation)

        # Check if we are there, return no steering.
        if rotationSize < targetRadius:
            return null

        # If we are outside the slowRadius, then use maximum rotation.
        if rotationSize > slowRadius:
```

```
            targetRotation = maxRotation
        # Otherwise calculate a scaled rotation.
        else:
            targetRotation =
                maxRotation * rotationSize / slowRadius

        # The final target rotation combines speed (already in the
        # variable) and direction.
        targetRotation *= rotation / rotationSize

        # Acceleration tries to get to the target rotation.
        result.angular = targetRotation - character.rotation
        result.angular /= timeToTarget

        # Check if the acceleration is too great.
        angularAcceleration = abs(result.angular)
        if angularAcceleration > maxAngularAcceleration:
            result.angular /= angularAcceleration
            result.angular *= maxAngularAcceleration

        result.linear = 0
        return result
```

where the function `abs` returns the absolute (i.e., positive) value of a number; for example, −1 is mapped to 1.

Implementation Notes

Whereas in the arrive implementation there are two vector normalizations, in this code we need to normalize a scalar (i.e., turn it into either +1 or −1). To do this we use the result that:

```
normalizedValue = value / abs(value)
```

In a production implementation in a language where you can access the bit pattern of a floating point number (C and C++, for example), you can do the same thing by manipulating the non-sign bits of the variable. Some C libraries provide an optimized `sign` function faster than the approach above.

Performance

The algorithm, unsurprisingly, is O(1) in both memory and time.

The Opposite

There is no such thing as the opposite of align. Because orientations wrap around every 2π, fleeing from an orientation in one direction will simply lead you back to where you started. To face the opposite direction of a target, simply add π to its orientation and align to that value.

3.3.6 VELOCITY MATCHING

So far we have looked at behaviors that try to match position (linear position or orientation) with a target. We will do the same with velocity.

On its own this behavior is seldom used. It could be used to make a character mimic the motion of a target, but this isn't very useful. Where it does become critical is when combined with other behaviors. It is one of the constituents of the flocking steering behavior, for example.

We have already implemented an algorithm that tries to match a velocity. Arrive calculates a target velocity based on the distance to its target. It then tries to achieve the target velocity. We can strip the arrive behavior down to provide a velocity matching implementation.

Pseudo-Code

The stripped down code looks like the following:

```
class VelocityMatch:
    character: Kinematic
    target: Kinematic

    maxAcceleration: float

    # The time over which to achieve target speed.
    timeToTarget = 0.1

    function getSteering() -> SteeringOutput:
        result = new SteeringOutput()

        # Acceleration tries to get to the target velocity.
        result.linear = target.velocity - character.velocity
        result.linear /= timeToTarget

        # Check if the acceleration is too fast.
        if result.linear.length() > maxAcceleration:
            result.linear.normalize()
            result.linear *= maxAcceleration

        result.angular = 0
        return result
```

Performance

The algorithm is O(1) in both time and memory.

3.3.7 DELEGATED BEHAVIORS

We have covered the basic building block behaviors that help to create many others. Seek and flee, arrive, and align perform the steering calculations for many other behaviors.

All the behaviors that follow have the same basic structure: they calculate a target, either a position or orientation (they could use velocity, but none of those we're going to cover does), and then they delegate to one of the other behaviors to calculate the steering. The target calculation can be based on many inputs. Pursue, for example, calculates a target for seek based on the motion of another target. Collision avoidance creates a target for flee based on the proximity of an obstacle. And wander creates its own target that meanders around as it moves.

In fact, it turns out that seek, align, and velocity matching are the only fundamental behaviors (there is a rotation matching behavior, by analogy, but I've never seen an application for it). As we saw in the previous algorithm, arrive can be divided into the creation of a (velocity) target and the application of the velocity matching algorithm. This is common. Many of the delegated behaviors below can, in turn, be used as the basis of another delegated behavior. Arrive can be used as the basis of pursue, pursue can be used as the basis of other algorithms, and so on.

In the code that follows I will use a polymorphic style of programming to capture these dependencies. You could alternatively use delegation, having the primitive algorithms called by the new techniques. Both approaches have their problems. In our case, when one behavior extends another, it normally does so by calculating an alternative target. Using inheritance means we need to be able to change the target that the super-class works on.

If we use the delegation approach, we'd need to make sure that each delegated behavior has the correct character data, `maxAcceleration`, and other parameters. This requires duplication and data copying that using sub-classes removes.

3.3.8 PURSUE AND EVADE

So far we have moved based solely on position. If we are chasing a moving target, then constantly moving toward its current position will not be sufficient. By the time we reach where it is now, it will have moved. This isn't too much of a problem when the target is close and we are reconsidering its location every frame. We'll get there eventually. But if the character is a long distance from its target, it will set off in a visibly wrong direction, as shown in Figure 3.11.

Instead of aiming at its current position, we need to predict where it will be at some time in the future and aim toward that point. We did this naturally playing tag as children, which

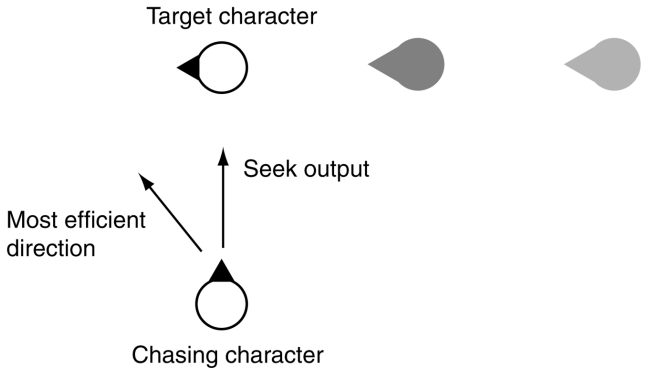

Figure 3.11: Seek moving in the wrong direction

is why the most difficult tag players to catch were those who kept switching direction, foiling our predictions.

We could use all kinds of algorithms to perform the prediction, but most would be overkill. Various research has been done into optimal prediction and optimal strategies for the character being chased (it is an active topic in military research for evading incoming missiles, for example). Craig Reynolds's original approach is much simpler: we assume the target will continue moving with the same velocity it currently has. This is a reasonable assumption over short distances, and even over longer distances it doesn't appear too stupid.

The algorithm works out the distance between character and target and works out how long it would take to get there, at maximum speed. It uses this time interval as its prediction lookahead. It calculates the position of the target if it continues to move with its current velocity. This new position is then used as the target of a standard seek behavior.

If the character is moving slowly, or the target is a long way away, the prediction time could be very large. The target is less likely to follow the same path forever, so we'd like to set a limit on how far ahead we aim. The algorithm has a maximum time parameter for this reason. If the prediction time is beyond this, then the maximum time is used.

Figure 3.12 shows a seek behavior and a pursue behavior chasing the same target. The pursue behavior is more effective in its pursuit.

Pseudo-Code

The pursue behavior derives from seek, calculates a surrogate target, and then delegates to seek to perform the steering calculation:

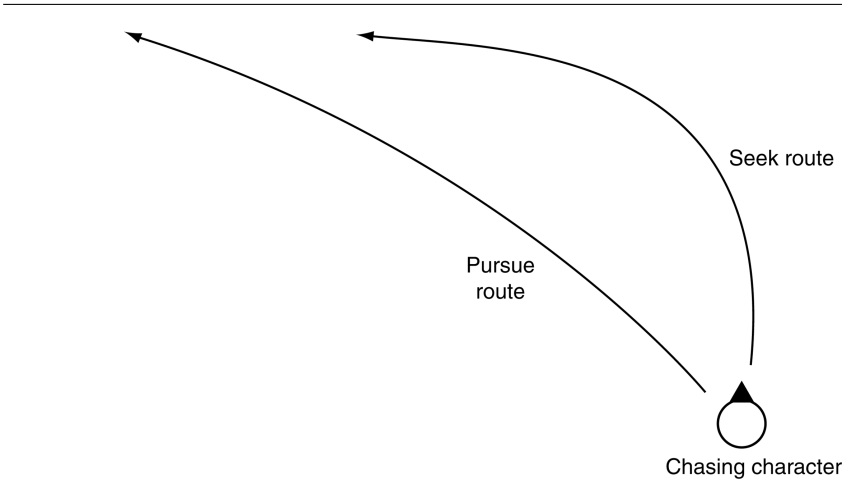

Figure 3.12: Seek and pursue

```
class Pursue extends Seek:
    # The maximum prediction time.
    maxPrediction: float

    # OVERRIDES the target data in seek (in other words this class has
    # two bits of data called target: Seek.target is the superclass
    # target which will be automatically calculated and shouldn't be
    # set, and Pursue.target is the target we're pursuing).
    target: Kinematic

    # ... Other data is derived from the superclass ...

    function getSteering() -> SteeringOutput:
        # 1. Calculate the target to delegate to seek
        # Work out the distance to target.
        direction = target.position - character.position
        distance = direction.length()

        # Work out our current speed.
        speed = character.velocity.length()

        # Check if speed gives a reasonable prediction time.
        if speed <= distance / maxPrediction:
            prediction = maxPrediction

```

```
        # Otherwise calculate the prediction time.
        else:
            prediction = distance / speed

        # Put the target together.
        Seek.target = explicitTarget
        Seek.target.position += target.velocity * prediction

        # 2. Delegate to seek.
        return Seek.getSteering()
```

Implementation Notes

In this code I've used the slightly unsavory technique of naming a member variable in a derived class with the same name as the super-class. In most languages this will have the desired effect of creating two members with the same name. In our case this is what we want: setting the pursue behavior's target will not change the target for the seek behavior it extends.

Be careful, though! In some languages (Python, for example) you can't do this. You'll have to name the target variable in each class with a different name.

As mentioned previously, it may be beneficial to cut out these polymorphic calls altogether to improve the performance of the algorithm. We can do this by having all the data we need in the pursue class, removing its inheritance of seek, and making sure that all the code the class needs is contained in the `getSteering` method. This is faster, but at the cost of duplicating the delegated code in each behavior that needs it and obscuring the natural reuse of the algorithm.

Performance

Once again, the algorithm is O(1) in both memory and time.

Evade

The opposite behavior of pursue is evade. Once again we calculate the predicted position of the target, but rather than delegating to seek, we delegate to flee.

In the code above, the class definition is modified so that it is a sub-class of `Flee` rather than `Seek` and thus `Seek.getSteering` is changed to `Flee.getSteering`.

Overshooting

If the chasing character is able to move faster than the target, it will overshoot and oscillate around its target, exactly as the normal seek behavior does.

To avoid this, we can replace the delegated call to seek with a call to arrive. This illustrates the power of building up behaviors from their logical components; when we need a slightly different effect, we can easily modify the code to get it.

3.3.9 FACE

The face behavior makes a character look at its target. It delegates to the align behavior to perform the rotation but calculates the target orientation first.

The target orientation is generated from the relative position of the target to the character. It is the same process we used in the `getOrientation` function for kinematic movement.

Pseudo-Code

The implementation is very simple:

```
class Face extends Align:
    # Overrides the Align.target member.
    target: Kinematic

    # ... Other data is derived from the superclass ...

    # Implemented as it was in Pursue.
    function getSteering( ) -> SteeringOutput:
        # 1. Calculate the target to delegate to align
        # Work out the direction to target.
        direction = target.position - character.position

        # Check for a zero direction, and make no change if so.
        if direction.length( ) == 0:
            return target

        # 2. Delegate to align.
        Align.target = explicitTarget
        Align.target.orientation = atan2(-direction.x, direction.z)
        return Align.getSteering( )
```

3.3.10 LOOKING WHERE YOU'RE GOING

So far, I have assumed that the direction a character is facing does not have to be its direction of motion. In many cases, however, we would like the character to face in the direction it is moving. In the kinematic movement algorithms we set it directly. Using the align behavior, we can give the character angular acceleration to make it face the correct direction. In this way the character changes facing gradually, which can look more natural, especially for aerial

vehicles such as helicopters or hovercraft or for human characters that can move sideways (providing sidestep animations are available).

This is a process similar to the face behavior, above. The target orientation is calculated using the current velocity of the character. If there is no velocity, then the target orientation is set to the current orientation. We have no preference in this situation for any orientation.

Pseudo-Code

The implementation is even simpler than face:

```
class LookWhereYoureGoing extends Align:
    # No need for an overridden target member, we have
    # no explicit target to set.

    # ... Other data is derived from the superclass ...

    function getSteering( ) -> SteeringOutput:
        # 1. Calculate the target to delegate to align
        # Check for a zero direction, and make no change if so.
        velocity: Vector = character.velocity
        if velocity.length( ) == 0:
            return null

        # Otherwise set the target based on the velocity.
        target.orientation = atan2(-velocity.x, velocity.z)

        # 2. Delegate to align.
        return Align.getSteering( )
```

Implementation Notes

In this case we don't need another target member variable. There is no overall target; we are creating the current target from scratch. So, we can simply use `Align.target` for the calculated target (in the same way we did with pursue and the other derived algorithms).

Performance

The algorithm is O(1) in both memory and time.

3.3.11 WANDER

The wander behavior controls a character moving aimlessly about.

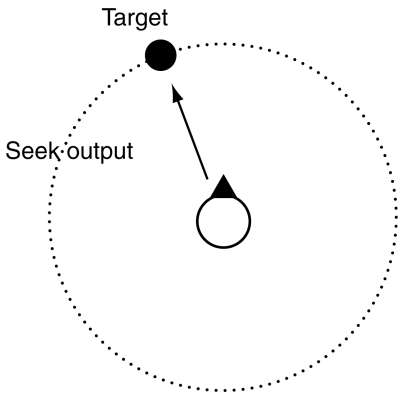

Figure 3.13: The kinematic wander as a seek

When we looked at the kinematic wander behavior, we perturbed the wander direction by a random amount each time it was run. This makes the character move forward smoothly, but the rotation of the character is erratic, appearing to twitch from side to side as it moves.

The simplest wander approach would be to move in a random direction. This is unacceptable, and therefore almost never used, because it produces linear jerkiness. The kinematic version added a layer of indirection but produced rotational jerkiness. We can smooth this twitching by adding an extra layer, making the orientation of the character indirectly reliant on the random number generator.

We can think of kinematic wander as behaving as a delegated seek behavior. There is a circle around the character on which the target is constrained. Each time the behavior is run, we move the target around the circle a little, by a random amount. The character then seeks the target. Figure 3.13 illustrates this configuration.

We can improve this by moving the circle around which the target is constrained. If we move it out in front of the character (where front is determined by its current facing direction) and shrink it down, we get the situation in Figure 3.14.

The character tries to face the target in each frame, using the face behavior to align to the target. It then adds an extra step: applying full acceleration in the direction of its current orientation.

We could also implement the behavior by having it seek the target and perform a 'look where you're going' behavior to correct its orientation.

In either case, the orientation of the character is retained between calls (so smoothing the changes in orientation). The angles that the edges of the circle subtend to the character determine how fast it will turn. If the target is on one of these extreme points, it will turn quickly. The target will twitch and jitter around the edge of the circle, but the character's orientation will change smoothly.

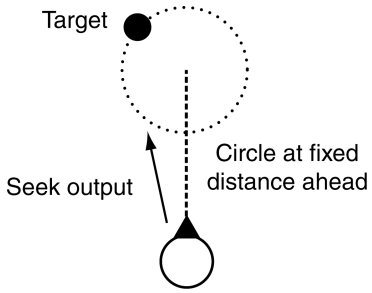

Figure 3.14: The full wander behavior

This wander behavior biases the character to turn (in either direction). The target will spend more time toward the edges of the circle, from the point of view of the character.

Pseudo-Code

```
class Wander extends Face:
    # The radius and forward offset of the wander circle.
    wanderOffset: float
    wanderRadius: float

    # The maximum rate at which the wander orientation can change.
    wanderRate: float

    # The current orientation of the wander target.
    wanderOrientation: float

    # The maximum acceleration of the character.
    maxAcceleration: float

    # Again we don't need a new target.
    # ... Other data is derived from the superclass ...

    function getSteering() -> SteeringOutput:
        # 1. Calculate the target to delegate to face
        # Update the wander orientation.
        wanderOrientation += randomBinomial() * wanderRate

        # Calculate the combined target orientation.
        targetOrientation = wanderOrientation + character.orientation

```

```
        # Calculate the center of the wander circle.
        target = character.position +
                 wanderOffset * character.orientation.asVector()

        # Calculate the target location.
        target += wanderRadius * targetOrientation.asVector()

        # 2. Delegate to face.
        result = Face.getSteering()

        # 3. Now set the linear acceleration to be at full
        # acceleration in the direction of the orientation.
        result.linear =
            maxAcceleration * character.orientation.asVector()

        # Return it.
        return result
```

Data Structures and Interfaces

I've used the same `asVector` function as earlier to get a vector form of the orientation.

Performance

The algorithm is O(1) in both memory and time.

3.3.12 PATH FOLLOWING

So far we've seen behaviors that take a single target or no target at all. Path following is a steering behavior that takes a whole path as a target. A character with path following behavior should move along the path in one direction.

Path following, as it is usually implemented, is a delegated behavior. It calculates the position of a target based on the current character location and the shape of the path. It then hands its target off to seek. There is no need to use arrive, because the target should always be moving along the path. We shouldn't need to worry about the character catching up with it.

The target position is calculated in two stages. First, the current character position is mapped to the nearest point along the path. This may be a complex process, especially if the path is curved or made up of many line segments. Second, a target is selected which is further along the path than the mapped point by some distance. To change the direction of motion along the path, we can change the sign of this distance. Figure 3.15 shows this in action. The current path location is shown, along with the target point a little way farther along. This

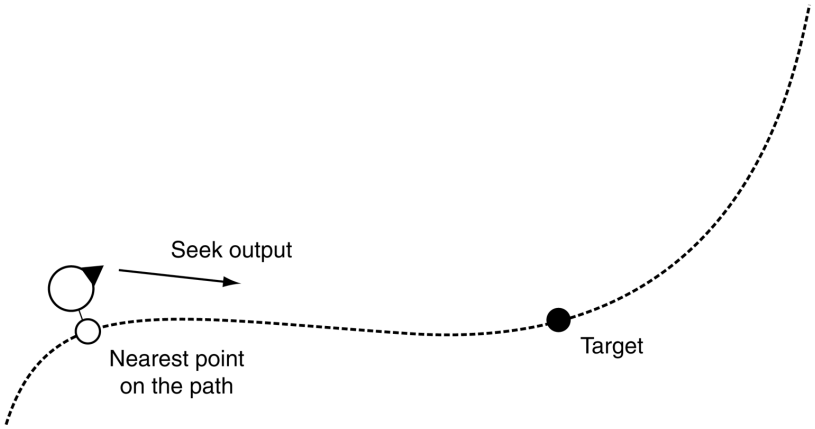

Figure 3.15: Path following behavior

approach is sometimes called "chase the rabbit," after the way greyhounds chase the cloth rabbit at the dog track.

Some implementations generate the target slightly differently. They first predict where the character will be in a short time and then map this to the nearest point on the path. This is a candidate target. If the new candidate target has not been placed farther along the path than it was at the last frame, then it is changed so that it is. I will call this *predictive path following*. It is shown in Figure 3.16. This latter implementation can appear smoother for complex paths with sudden changes of direction, but has the downside of cutting corners when two paths come close together.

Figure 3.17 shows this cutting-corner behavior. The character misses a whole section of the path. The character is shown at the instant its predictive future position crosses to a later part of the path.

This might not be what you want if, for example, the path represents a patrol route or a race track.

Pseudo-Code

```
class FollowPath extends Seek:
    path: Path

    # The distance along the path to generate the target. Can be
    # negative if the character is moving in the reverse direction.
    pathOffset: float

```

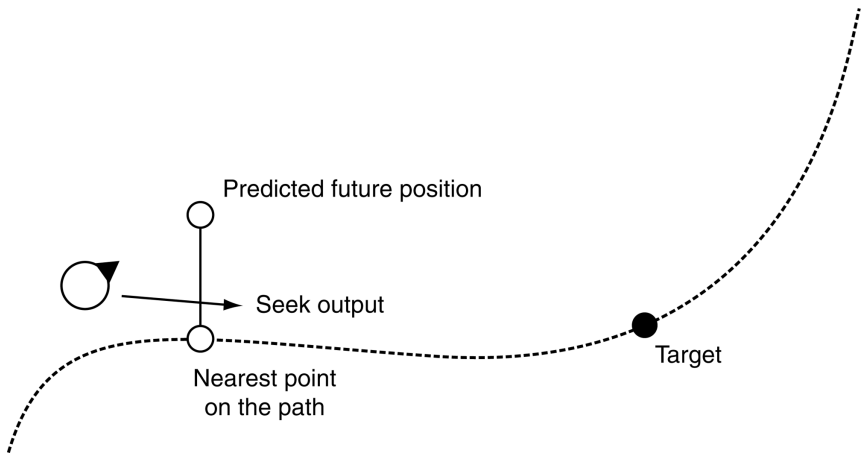

Figure 3.16: Predictive path following behavior

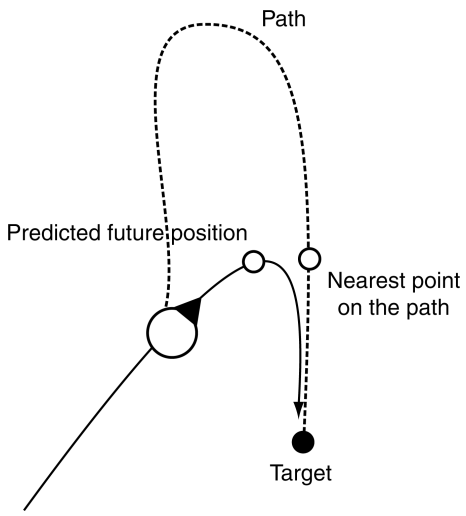

Figure 3.17: Vanilla and predictive path following

```
    # The current position on the path.
    currentParam: float

    # ... Other data is derived from the superclass ...

    function getSteering() -> SteeringOutput:
        # 1. Calculate the target to delegate to face.
        # Find the current position on the path.
        currentParam = path.getParam(character.position, currentPos)

        # Offset it.
        targetParam = currentParam + pathOffset

        # Get the target position.
        target.position = path.getPosition(targetParam)

        # 2. Delegate to seek.
        return Seek.getSteering()
```

We can convert this algorithm to a predictive version by first calculating a surrogate position for the call to `path.getParam`. The algorithm looks almost identical:

```
class FollowPath extends Seek:
    path: Path

    # The distance along the path to generate the target. Can be
    # negative if the character is moving in the reverse direction.
    pathOffset: float

    # The current position on the path.
    currentParam: float

    # The time in the future to predict the character's position.
    predictTime: float = 0.1

    # ... Other data is derived from the superclass ...

    function getSteering() -> SteeringOutput:
        # 1. Calculate the target to delegate to face.
        # Find the predicted future location.
        futurePos = character.position +
                    character.velocity * predictTime

        # Find the current position on the path.
        currentParam = path.getParam(futurePos, currentPos)

        # Offset it.
        targetParam = currentParam + pathOffset
```

```
27
28          # Get the target position.
29          target.position = path.getPosition(targetParam)
30
31          # 2. Delegate to seek.
32          return Seek.getSteering()
```

Data Structures and Interfaces

The path that the behavior follows has the following interface:

```
1 class Path:
2     function getParam(position: Vector, lastParam: float) -> float
3     function getPosition(param: float) -> Vector
```

Both these functions use the concept of a path parameter. This is a unique value that increases monotonically along the path. It can be thought of as a distance along the path. Typically, paths are made up of line or curve splines; both of these are easily assigned parameters. The parameter allows us to translate between the position on the path and positions in 2D or 3D space.

Path Types

Performing this translation (i.e., implementing a path class) can be tricky, depending on the format of the path used.

It is most common to use a path of straight line segments as shown in Figure 3.18. In this case the conversion is not too difficult. We can implement `getParam` by looking at each line segment in turn, determining which one the character is nearest to, and then finding the nearest point on that segment. For smooth curved splines common in some driving games, however, the math can be more complex. A good source for closest-point algorithms for a range of different geometries is Schneider and Eberly [56].

Keeping Track of the Parameter

The pseudo-code interface above provides for sending the last parameter value to the path in order to calculate the current parameter value. This is essential to avoid nasty problems when lines are close together.

We limit the `getParam` algorithm to only considering areas of the path close to the previous parameter value. The character is unlikely to have moved far, after all. This technique, assuming the new value is close to the old one, is called *coherence*, and it is a feature of many geometric algorithms. Figure 3.19 shows a problem that would fox a non-coherent path follower but is easily handled by assuming the new parameter is close to the old one.

Of course, you may really want corners to be cut or a character to move between very different parts of the path. If another behavior interrupts and takes the character across the

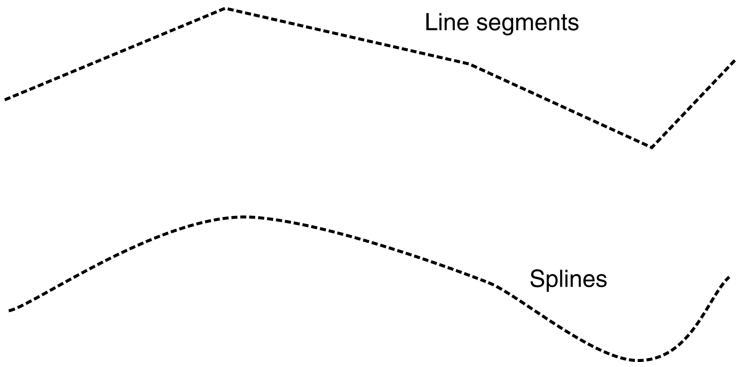

Figure 3.18: Path types

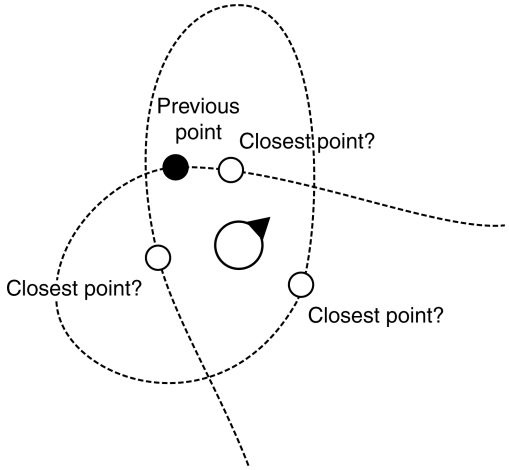

Figure 3.19: Coherence problems with path following

level, for example, you don't necessarily want it to come all the way back to pick up a circular patrol route. In this case, you'll need to remove coherence or at least widen the range of parameters that it searches for a solution.

Performance

The algorithm is O(1) in both memory and time. The getParam function of the path will usually be O(1), although it may be O(n), where n is the number of segments in the path. If this is the case, then the `getParam` function will dominate the performance scaling of the algorithm.

3.3.13 SEPARATION

The separation behavior is common in crowd simulations, where a number of characters are all heading in roughly the same direction. It acts to keep the characters from getting too close and being crowded.

It doesn't work as well when characters are moving across each other's paths. The collision avoidance behavior, below, should be used in this case.

Most of the time, the separation behavior has a zero output; it doesn't recommend any movement at all. If the behavior detects another character closer than some threshold, it acts in a way similar to an evade behavior to move away from the character. Unlike the basic evade behavior, however, the strength of the movement is related to the distance from the target. The separation strength can decrease according to any formula, but a linear or an inverse square law decay is common.

Linear separation looks like the following:

```
strength = maxAcceleration * (threshold - distance) / threshold
```

The inverse square law looks like the following:

```
strength = min(k / (distance * distance), maxAcceleration)
```

In each case, `distance` is the distance between the character and its nearby neighbor, `threshold` is the minimum distance at which any separation output occurs, and `maxAcceleration` is the maximum acceleration of the character. The `k` constant can be set to any positive value. It controls how fast the separation strength decays with distance.

Separation is sometimes called the "repulsion" steering behavior, because it acts in the same way as a physical repulsive force (an inverse square law force such as magnetic repulsion).

Where there are multiple characters within the avoidance threshold, either the closest can be selected, or the steering can be calculated for each in turn and summed. In the latter case, the final value may be greater than the `maxAcceleration`, in which case it should be clipped to that value.

Pseudo-Code

```
class Separation:
    character: Kinematic
    maxAcceleration: float

    # A list of potential targets.
    targets: Kinematic[]

    # The threshold to take action.
    threshold: float

    # The constant coefficient of decay for the inverse square law.
    decayCoefficient: float

    function getSteering() -> SteeringOutput:
        result = new SteeringOutput()

        # Loop through each target.
        for target in targets:
            # Check if the target is close.
            direction = target.position - character.position
            distance = direction.length()

            if distance < threshold:
                # Calculate the strength of repulsion
                # (here using the inverse square law).
                strength = min(
                    decayCoefficient / (distance * distance),
                    maxAcceleration)

                # Add the acceleration.
                direction.normalize()
                result.linear += strength * direction

        return result
```

Implementation Notes

In the algorithm above, we simply look at each possible character in turn and work out whether we need to separate from them. For a small number of characters, this will be the fastest approach. For a few hundred characters in a level, we need a faster method.

Typically, graphics and physics engines rely on techniques to determine what objects are close to one another. Objects are stored in spatial data structures, so it is relatively easy to

make this kind of query. Multi-resolution maps, quad- or octrees, and binary space partition (BSP) trees are all popular data structures for rapidly calculating potential collisions. Each of these can be used by the AI to get potential targets more efficiently.

Implementing a spatial data structure for collision detection is beyond the scope of this book. Other books in this series cover the topic in much more detail, particularly Ericson [14] and van den Bergen [73].

Performance

The algorithm is O(1) in memory and O(n) in time, where n is the number of potential targets to check. If there is some efficient way of pruning potential targets before they reach the algorithm above, the overall performance in time will improve. A BSP system, for example, can give O($\log n$) time, where n is the total number of potential targets in the game. The algorithm above will always remain linear in the number of potential targets it checks, however.

Attraction

Using the inverse square law, we can set a negative value for the constant of decay and get an attractive force. The character will be attracted to others within its radius. This works, but is rarely useful.

Some developers have experimented with having lots of attractors and repulsors in their level and having character movement mostly controlled by these. Characters are attracted to their goals and repelled from obstacles, for example. Despite being ostensibly simple, this approach is full of traps for the unwary.

The next section, on combining steering behaviors, shows why simply having lots of attractors or repulsors leads to characters that regularly get stuck and why starting with a more complex algorithm ends up being less work in the long run.

Independence

The separation behavior isn't much use on its own. Characters will jiggle out of separation, but then never move again. Separation, along with the remaining behaviors in this chapter, is designed to work in combination with other steering behaviors. We return to how this combination works in the next section.

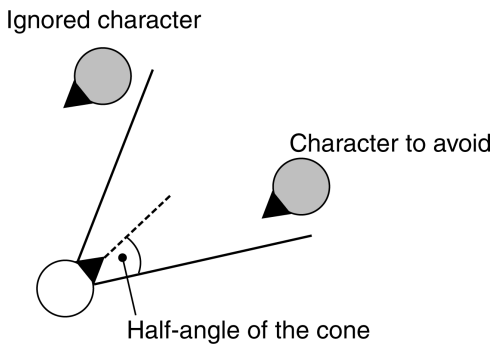

Figure 3.20: Separation cones for collision avoidance

3.3.14 COLLISION AVOIDANCE

In urban areas, it is common to have large numbers of characters moving around the same space. These characters have trajectories that cross each other, and they need to avoid constant collisions with other moving characters.

A simple approach is to use a variation of the evade or separation behavior, which only engages if the target is within a cone in front of the character. Figure 3.20 shows a cone that has another character inside it.

The cone check can be carried out using a dot product:

```
if dotProduct(orientation.asVector(), direction) > coneThreshold:
    # Do the evasion.
else:
    # Return no steering.
```

where `direction` is the direction between the behavior's character and the potential collision. The `coneThreshold` value is the cosine of the cone half-angle, as shown in Figure 3.20.

If there are several characters in the cone, then the behavior needs to avoid them all. We can approach this in a similar way to the separation behavior. It is often sufficient to find the average position and speed of all characters in the cone and evade that target. Alternatively, the closest character in the cone can be found and the rest ignored.

Unfortunately, this approach, while simple to implement, doesn't work well with more than a handful of characters. The character does not take into account whether it will actually collide but instead has a "panic" reaction to even coming close. Figure 3.21 shows a simple situation where the character will never collide, but our naive collision avoidance approach will still take action.

Figure 3.22 shows another problem situation. Here the characters will collide, but neither

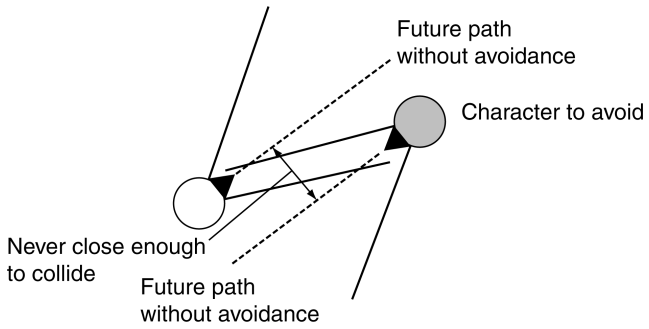

Figure 3.21: Two in-cone characters who will not collide

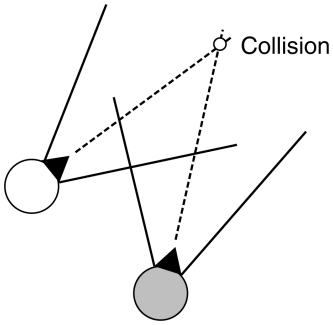

Figure 3.22: Two out-of-cone characters who will collide

will take evasive action because they will not have the other in their cone until the moment of collision.

A better solution works out whether or not the characters will collide if they keep to their current velocity. This involves working out the closest approach of the two characters and determining if the distance at this point is less than some threshold radius. This is illustrated in Figure 3.23.

Note that the closest approach will not normally be the same as the point where the future trajectories cross. The characters may be moving at very different velocities, and so are likely to reach the same point at different times. We simply can't see if their paths will cross to check if the characters will collide. Instead, we have to find the moment that they are at their closest, use this to derive their separation, and check if they collide.

The time of closest approach is given by

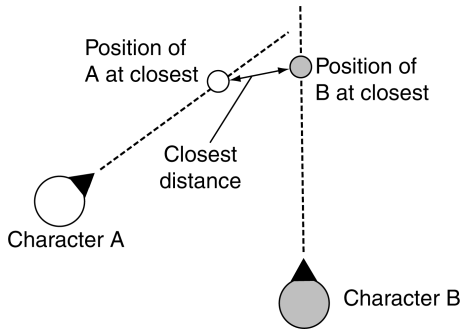

Figure 3.23: Collision avoidance using collision prediction

$$t_{\text{closest}} = \frac{d_p.d_v}{|d_v|^2}$$

where d_p is the current relative position of target to character (what we called the distance vector from previous behaviors):

$$d_p = p_t - p_c$$

and d_v is the relative velocity:

$$d_v = v_t - v_c$$

If the time of closest approach is negative, then the character is already moving away from the target, and no action needs to be taken.

From this time, the position of character and target at the time of closest approach can be calculated:

$$p'_c = p_c + v_c t_{\text{closest}}$$

$$p'_t = p_t + v_t t_{\text{closest}}$$

We then use these positions as the basis of an evade behavior; we are performing an evasion based on our predicted future positions, rather than our current positions. In other words, the behavior makes the steering correction now, as if it were already at the most compromised position it will get to.

For a real implementation it is worth checking if the character and target are already in collision. In this case, action can be taken immediately, without going through the calculations to work out if they will collide at some time in the future. In addition, this approach

will not return a sensible result if the centers of the character and target will collide at some point. A sensible implementation will have some special case code for this unlikely situation to make sure that the characters will sidestep in different directions. This can be as simple as falling back to the evade behavior on the current positions of the character.

For avoiding groups of characters, averaging positions and velocities do not work well with this approach. Instead, the algorithm needs to search for the character whose closest approach will occur first and to react to this character only. Once this imminent collision is avoided, the steering behavior can then react to more distant characters.

Pseudo-Code

```
class CollisionAvoidance:
    character: Kinematic
    maxAcceleration: float

    # A list of potential targets.
    targets: Kinematic[]

    # The collision radius of a character (assuming all characters
    # have the same radius here).
    radius: float

    function getSteering() -> SteeringOutput:
        # 1. Find the target that's closest to collision
        # Store the first collision time.
        shortestTime: float = infinity

        # Store the target that collides then, and other data that we
        # will need and can avoid recalculating.
        firstTarget: Kinematic = null
        firstMinSeparation: float
        firstDistance: float
        firstRelativePos: Vector
        firstRelativeVel: Vector

        # Loop through each target.
        for target in targets:
            # Calculate the time to collision.
            relativePos = target.position - character.position
            relativeVel = target.velocity - character.velocity
            relativeSpeed = relativeVel.length()
            timeToCollision = dotProduct(relativePos, relativeVel) /
                              (relativeSpeed * relativeSpeed)

            # Check if it is going to be a collision at all.
```

```
            distance = relativePos.length()
            minSeparation = distance - relativeSpeed * timeToCollision
            if minSeparation > 2 * radius:
                continue

            # Check if it is the shortest.
            if timeToCollision > 0 and timeToCollision < shortestTime:
                # Store the time, target and other data.
                shortestTime = timeToCollision
                firstTarget = target
                firstMinSeparation = minSeparation
                firstDistance = distance
                firstRelativePos = relativePos
                firstRelativeVel = relativeVel

        # 2. Calculate the steering
        # If we have no target, then exit.
        if not firstTarget:
            return null

        # If we're going to hit exactly, or if we're already
        # colliding, then do the steering based on current position.
        if firstMinSeparation <= 0 or firstDistance < 2 * radius:
            relativePos = firstTarget.position - character.position

        # Otherwise calculate the future relative position.
        else:
            relativePos = firstRelativePos +
                          firstRelativeVel * shortestTime

        # Avoid the target.
        relativePos.normalize()

        result = new SteeringOutput()
        result.linear = relativePos * maxAcceleration
        result.anguar = 0
        return result
```

Performance

The algorithm is O(1) in memory and O(n) in time, where n is the number of potential targets to check.

As in the previous algorithm, if there is some efficient way of pruning potential targets before they reach the algorithm above, the overall performance in time will improve. This algorithm will always remain linear in the number of potential targets it checks, however.

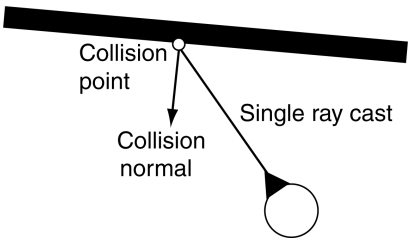

Figure 3.24: Collision ray avoiding a wall

3.3.15 OBSTACLE AND WALL AVOIDANCE

The collision avoidance behavior assumes that targets are spherical. It is interested in avoiding a trajectory too close to the center point of the target.

This can also be applied to any obstacle in the game that is easily represented by a bounding sphere. Crates, barrels, and small objects can be avoided simply this way.

More complex obstacles cannot be easily represented as circles. The bounding sphere of a large object, such as a staircase, can fill a room. We certainly don't want characters sticking to the outside of the room just to avoid a staircase in the corner. By far the most common obstacles in the game are walls, and they cannot be simply represented by bounding spheres at all.

The obstacle and wall avoidance behavior uses a different approach to avoiding collisions. The moving character casts one or more rays out in the direction of its motion. If these rays collide with an obstacle, then a target is created that will avoid the collision, and the character does a basic seek on this target. Typically, the rays are not infinite. They extend a short distance ahead of the character (usually a distance corresponding to a few seconds of movement).

Figure 3.24 shows a character casting a single ray that collides with a wall. The point and normal of the collision with the wall are used to create a target location at a fixed distance from the surface.

Pseudo-Code

```
class ObstacleAvoidance extends Seek:
    detector: CollisionDetector

    # The minimum distance to a wall (i.e., how far to avoid
    # collision) should be greater than the radius of the character.
    avoidDistance: float

    # The distance to look ahead for a collision
```

```
    # (i.e., the length of the collision ray).
    lookahead: float

    # ... Other data is derived from the superclass ...

    function getSteering():
        # 1. Calculate the target to delegate to seek
        # Calculate the collision ray vector.
        ray = character.velocity
        ray.normalize()
        ray *= lookahead

        # Find the collision.
        collision = detector.getCollision(character.position, ray)

        # If have no collision, do nothing.
        if not collision:
            return null

        # 2. Otherwise create a target and delegate to seek.
        target = collision.position + collision.normal * avoidDistance
        return Seek.getSteering()
```

Data Structures and Interfaces

The collision detector has the following interface:

```
class CollisionDetector:
    function getCollision(position: Vector,
                          moveAmount: Vector) -> Collision
```

where `getCollision` returns the first collision for the character if it begins at the given position and moves by the given movement amount. Collisions in the same direction, but farther than `moveAmount`, are ignored.

Typically, this call is implemented by casting a ray from `position` to `position + moveAmount` and checking for intersections with walls or other obstacles.

The `getCollision` method returns a collision data structure of the form:

```
class Collision:
    position: Vector
    normal: Vector
```

where `position` is the collision point and `normal` is the normal of the wall at the point of collision. These are standard data to expect from a collision detection routine, and most engines provide such data as a matter of course (see `Physics.Raycast` in Unity, in Unreal Engine it requires more setup, see the documentation for Traces).

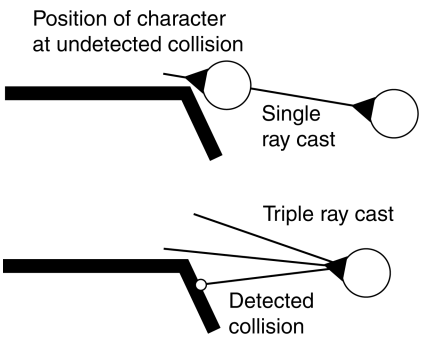

Figure 3.25: Grazing a wall with a single ray and avoiding it with three

Performance

The algorithm is O(1) in both time and memory, excluding the performance of the collision detector (or rather, assuming that the collision detector is O(1)). In reality, collision detection using ray casts is quite expensive and is almost certainly not O(1) (it normally depends on the complexity of the environment). You should expect that most of the time spent in this algorithm will be spent in the collision detection routine.

Collision Detection Problems

So far we have assumed that we are detecting collisions with a single ray cast. In practice, this isn't a good solution.

Figure 3.25 shows a one-ray character colliding with a wall that it never detects. Typically, a character will need to have two or more rays. The figure shows a three-ray character, with the rays splayed out to act like whiskers. This character will not graze the wall.

I have seen several basic ray configurations used over and over for wall avoidance. Figure 3.26 illustrates these.

There are no hard and fast rules as to which configuration is better. Each has its own particular idiosyncrasies. A single ray with short whiskers is often the best initial configuration to try but can make it impossible for the character to move down tight passages. The single ray configuration is useful in concave environments but grazes convex obstacles. The parallel configuration works well in areas where corners are highly obtuse but is very susceptible to the corner trap, as we'll see.

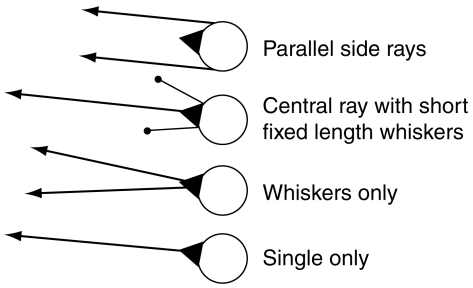

Figure 3.26: Ray configurations for obstacle avoidance

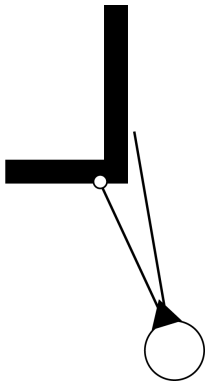

Figure 3.27: The corner trap for multiple rays

The Corner Trap

The basic algorithm for multi-ray wall avoidance can suffer from a crippling problem with acute angled corners (any convex corner, in fact, but it is more prevalent with acute angles). Figure 3.27 illustrates a trapped character. Currently, its left ray is colliding with the wall. The steering behavior will therefore turn it to the left to avoid the collision. Immediately, the right ray will then be colliding, and the steering behavior will turn the character to the right.

When the character is run in the game, it will appear to home into the corner directly, until it slams into the wall. It will be unable to free itself from the trap.

The fan structure, with a wide enough fan angle, alleviates this problem. Often, there is a trade-off, however, between avoiding the corner trap with a large fan angle and keeping the angle small to allow the character to access small passageways. At worst, with a fan angle near

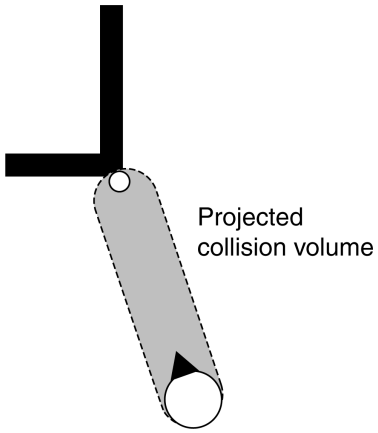

Figure 3.28: Collision detection with projected volumes

π radians, the character will not be able to respond quickly enough to collisions detected on its side rays and will still graze against walls.

Several developers have experimented with adaptive fan angles. If the character is moving successfully without a collision, then the fan angle is narrowed. If a collision is detected, then the fan angle is widened. If the character detects many collisions on successive frames, then the fan angle will continue to widen, reducing the chance that the character is trapped in a corner.

Other developers implement specific corner-trap avoidance code. If a corner trap is detected, then one of the rays is considered to have won, and the collisions detected by other rays are ignored for a while.

Both approaches work well and represent practical solutions to the problem. The only complete solution, however, is to perform the collision detection using a projected volume rather than a ray, as shown in Figure 3.28.

Many game engines are capable of doing this, for the sake of modeling realistic physics. Unlike for AI, the projection distances required by physics are typically very small, however, and the calculations can be very slow when used in a steering behavior.

In addition, there are complexities involved in interpreting the collision data returned from a volume query. Unlike for physics, it is not the first collision point that needs to be considered (this could be the edge of a polygon on one extreme of the character model), but how the overall character should react to the wall. So far there is no widely trusted mechanism for doing volume prediction in wall avoidance.

For now, it seems that the most practical solution is to use adaptive fan angles, with one long ray cast and two shorter whiskers.

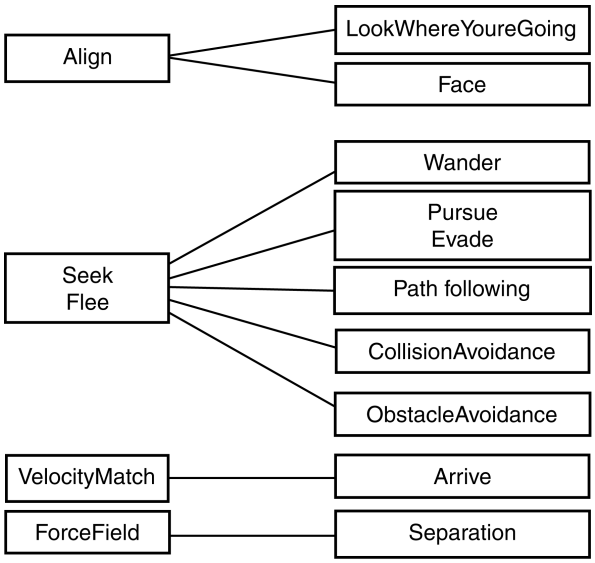

Figure 3.29: Steering family tree

3.3.16 SUMMARY

Figure 3.29 shows a family tree of the steering behaviors we have looked at in this section. We've marked a steering behavior as a child of another if it can be seen as extending the behavior of its parent.

3.4 COMBINING STEERING BEHAVIORS

Individually, steering behaviors can achieve a good degree of movement sophistication. In many games steering simply consists of moving toward a given location: the seek behavior.

Higher level decision making tools are responsible for determining where the character intends to move. This is often a pathfinding algorithm, generating intermediate targets on the path to a final goal.

This only gets us so far, however. A moving character usually needs more than one steering behavior. It needs to reach its goal, avoid collisions with other characters, tend toward safety as it moves, and avoid bumping into walls. Wall and obstacle avoidance can be particularly difficult to get when working with other behaviors. In addition, some complex steering, such as flocking and formation motion, can only be accomplished when more than one steering behavior is active at once.

This section looks at increasingly sophisticated ways of accomplishing this combination:

from simple blending of steering outputs to complicated pipeline architectures designed explicitly to support collision avoidance.

3.4.1 BLENDING AND ARBITRATION

By combining steering behaviors together, more complex movement can be achieved. There are two methods of combining steering behaviors: blending and arbitration.

Each method takes a portfolio of steering behaviors, each with its own output, and generates a single overall steering output. Blending does this by executing all the steering behaviors and combining their results using some set of weights or priorities. This is sufficient to achieve some very complex behaviors, but problems arise when there are a lot of constraints on how a character can move. Arbitration selects one or more steering behaviors to have complete control over the character. There are many arbitration schemes that control which behavior gets to have its way.

Blending and arbitration are not exclusive approaches, however. They are the ends of a continuum.

Blending may have weights or priorities that change over time. Some process needs to change these weights, and this might be in response to the game situation or the internal state of the character. The weights used for some steering behaviors may be zero; they are effectively switched off.

At the same time, there is nothing that requires an arbitration architecture to return a single steering behavior to execute. It may return a set of blending weights for combining a set of different behaviors.

A general steering system needs to combine elements of both blending and arbitration. Although we'll look at different algorithms for each, an ideal implementation will mix elements of both.

3.4.2 WEIGHTED BLENDING

The simplest way to combine steering behaviors is to blend their results together using weights.

Suppose we have a crowd of rioting characters in our game. The characters need to move as a mass, while making sure that they aren't consistently bumping into each other. Each character needs to stay by the others, while keeping a safe distance. Their overall behavior is a blend of two behaviors: arriving at the center of mass of the group and separation from nearby characters. At no point is the character doing just one thing. It is always taking both concerns into consideration.

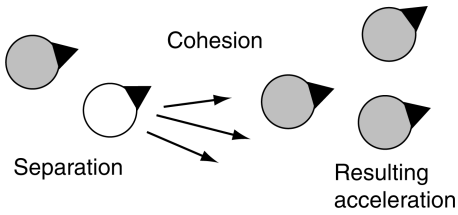

Figure 3.30: Blending steering outputs

The Algorithm

A group of steering behaviors can be blended together to act as a single behavior. Each steering behavior in the portfolio is asked for its acceleration request, as if it were the only behavior operating.

These accelerations are combined together using a weighted linear sum, with coefficients specific to each behavior. There are no constraints on the blending weights; they don't have to sum to one, for example, and rarely do (i.e., it isn't a weighted mean).

The final acceleration from the sum may be too great for the capabilities of the character, so it is trimmed according to the maximum possible acceleration (a more complex actuation step can always be used; see Section 3.8 on motor control later in the chapter).

In our crowd example, we may use weights of 1 for both separation and cohesion. In this case the requested accelerations are summed and cropped to the maximum possible acceleration. This is the output of the algorithm. Figure 3.30 illustrates this process.

As in all parameterized systems, the choice of weights needs to be the subject of inspired guesswork or good trial and error. Research projects have tried to evolve the steering weights using genetic algorithms or neural networks. Results have not been encouraging, however, and manual experimentation still seems to be the most sensible approach.

Pseudo-Code

The algorithm for blended steering is as follows:

```
class BlendedSteering:
    class BehaviorAndWeight:
        behavior: SteeringBehavior
        weight: float

    behaviors: BehaviorAndWeight[]

    # The overall maximum acceleration and rotation.
    maxAcceleration: float
```

```
    maxRotation: float

    function getSteering( ) -> SteeringOutput:
        result = new SteeringOutput( )

        # Accumulate all accelerations.
        for b in behaviors:
            result += b.weight * b.behavior.getSteering( )

        # Crop the result and return.
        result.linear = max(result.linear, maxAcceleration)
        result.angular = max(result.angular, maxRotation)
        return result
```

Data Structures

I have assumed that instances of the `SteeringOutput` structure can be added together and multiplied by a scalar. In each case these operations should be performed component-wise (i.e., linear and angular components should individually be added and multiplied).

Performance

The algorithm requires only temporary storage for the acceleration. It is O(1) in memory. It is O(n) for time, where n is the number of steering behaviors in the list. The practical execution speed of this algorithm depends on the efficiency of the component steering behaviors.

Flocking and Swarming

The original research into steering behaviors by Craig Reynolds modeled the movement patterns of flocks of simulated birds (known as "boids"). Though not the most commonly implemented in a game, flocking is the most commonly cited steering behavior. It relies on a simple weighted blend of simpler behaviors.

It is so ubiquitous that all steering behaviors are sometimes referred to, incorrectly, as "flocking." I've even seen AI programmers fall into this habit at times.

The flocking algorithm relies on blending three simple steering behaviors: move away from boids that are too close (separation), move in the same direction and at the same velocity as the flock (alignment and velocity matching), and move toward the center of mass of the flock (cohesion). The cohesion steering behavior calculates its target by working out the center of mass of the flock. It then hands off this target to a regular arrive behavior.

For simple flocking, using equal weights may be sufficient. To get a more biologically

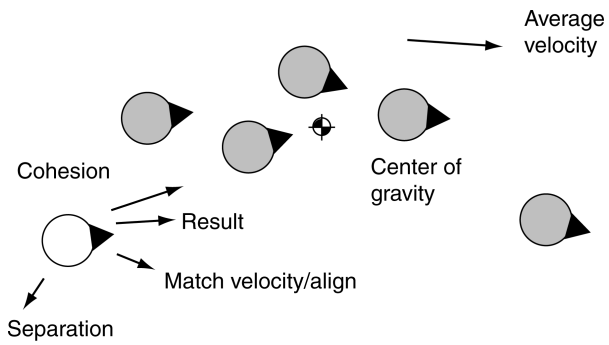

Figure 3.31: The three components of flocking behaviors

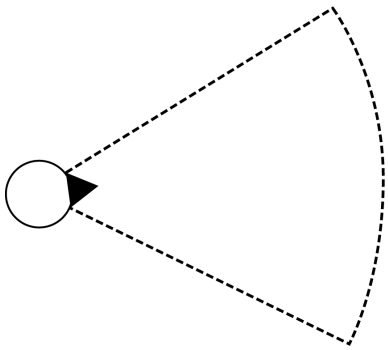

Figure 3.32: The neighborhood of a boid

believable behavior, however, separation is more important than cohesion, which is more important than alignment. The latter two are sometimes seen reversed.

These behaviors are shown schematically in Figure 3.31.

In most implementations the flocking behavior is modified to ignore distant boids. In each behavior there is a neighborhood in which other boids are considered. Separation only avoids nearby boids; cohesion and alignment calculate and seek the position, facing, and velocity of only neighboring boids. The neighborhood is most commonly a simple radius, although Reynolds suggests it should have an angular cut-off, as shown in Figure 3.32.

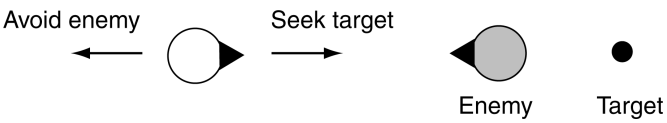

Figure 3.33: An unstable equilibrium

Problems

There are several important problems with blended steering behaviors in real games. It is no coincidence that demonstrations of blended steering often use very sparse outdoor environments, rather than indoor or urban levels.

In more realistic settings, characters can often get stuck in the environment in ways that are difficult to debug. As with all AI techniques, it is essential to be able to get good debugging information when you need it and at the very least to be able to visualize the inputs and outputs to each steering behavior in the blend.

Some of these problems, but by no means all of them, will be solved by introducing arbitration into the steering system.

Stable Equilibria

Blending steering behaviors causes problems when two steering behaviors want to do conflicting things. This can lead to the character doing nothing, being trapped at an equilibrium. In Figure 3.33, the character is trying to reach its destination while avoiding the enemy. The seek steering behavior is precisely balanced against the evade behavior.

This balance will soon sort itself out. As long as the enemy is stationary, numerical instability will give the character a minute lateral velocity. It will skirt around increasingly quickly before making a dash for the destination. This is an unstable equilibrium.

Figure 3.34 shows a more serious situation. Here, if the character does make it out of equilibrium slightly (as a result of limited floating point accuracy, for example), it will immediately head back into equilibrium. There is no escape for the character. It will stay fixed to the spot, looking stupid and indecisive. The equilibrium is stable.

Stable equilibria have a basin of attraction: the region of the level where a character will fall into the equilibrium point. If this basin is large, then the chances of a character becoming trapped are very large. Figure 3.34 shows a basin of attraction that extends in a corridor for an unlimited distance. Unstable equilibria effectively have a basin of zero size.

Basins of attraction aren't only defined by a set of locations. They might only attract characters that are traveling in a particular direction or that have a particular orientation. For this reason they can be very difficult to visualize and debug.

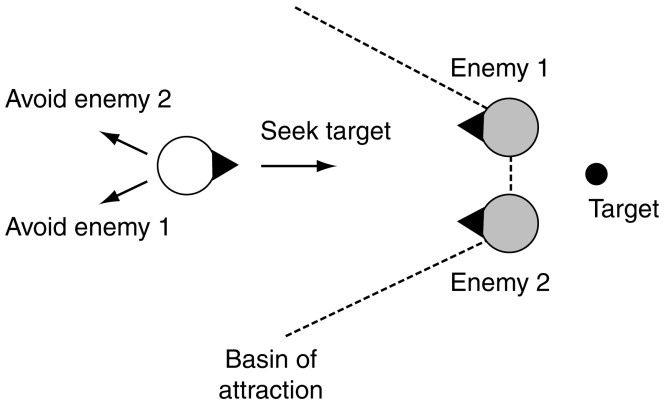

Figure 3.34: A stable equilibrium

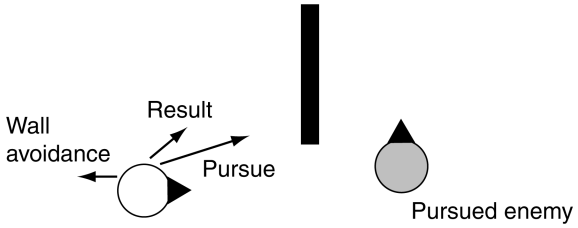

Figure 3.35: Can't avoid an obstacle and chase

Constrained Environments

Steering behaviors, either singly or blended, work well in environments with few constraints. Movement in an open 3D space has the fewest constraints. Most games, however, take place in constrained 2D worlds. Indoor environments, racetracks, and groups of characters moving in formation all greatly increase the number of constraints on a character's movement.

Figure 3.35 shows a chasing steering behavior returning a pathological suggestion for the motion of a character. The pursue behavior alone would collide with the wall, but adding the wall avoidance makes the direction even farther from the correct route for capturing the enemy.

This problem is often seen in characters trying to move at acute angles through narrow doorways, as shown in Figure 3.36. The obstacle avoidance behavior kicks in and can send the character past the door, missing the route it wanted to take.

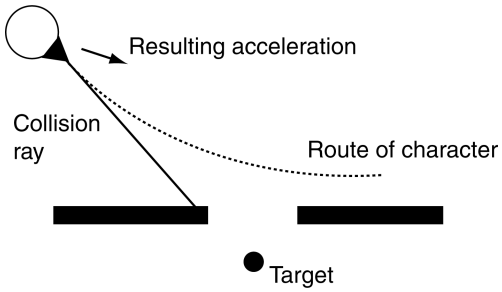

Figure 3.36: Missing a narrow doorway

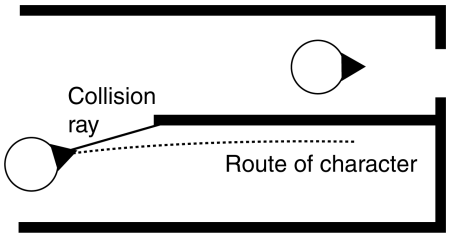

Figure 3.37: Long distance failure in a steering behavior

The problem of navigating into narrow passages is so perennial that many developers deliberately get their level designers to make wide passages where AI characters need to navigate.

Nearsightedness

Steering behaviors act locally. They make decisions based on their immediate surroundings only. As human beings, we anticipate the result of our actions and evaluate if it will be worth it. Basic steering behaviors can't do this, so they often take the wrong course of action to reach their goal.

Figure 3.37 shows a character avoiding a wall using a standard wall avoidance technique. The movement of the character catches the corner on just the wrong side. It will never catch the enemy now, but it won't realize that for a while.

There is no way to augment steering behaviors to get around this problem. Any behavior that does not lookahead can be foiled by problems that are beyond its horizon. The only way to solve this is to incorporate pathfinding into the steering system. This integration is discussed below, and the pathfinding algorithms themselves are found in the next chapter.

3.4.3 PRIORITIES

We have met a number of steering behaviors that will only request an acceleration in particular situations. Unlike seek or evade, which always produce an acceleration, collision avoidance, separation, and arrive will suggest no acceleration in many cases.

When these behaviors do suggest an acceleration, it is unwise to ignore it. A collision avoidance behavior, for example, should be honored immediately to avoid banging into another character.

When behaviors are blended together, their acceleration requests are diluted by the requests of the others. A seek behavior, for example, will always be returning maximum acceleration in some direction. If this is blended equally with a collision avoidance behavior, then the collision avoidance behavior will never have more than 50% influence over the motion of the character. This may not be enough to get the character out of trouble.

The Algorithm

A variation of behavior blending replaces weights with priorities. In a priority-based system, behaviors are arranged in groups with regular blending weights. These groups are then placed in priority order.

The steering system considers each group in turn. It blends the steering behaviors in the group together, exactly as before. If the total result is very small (less than some small, but adjustable, parameter), then it is ignored and the next group is considered. It is best not to check against zero directly, because numerical instability in calculations can mean that a zero value is never reached for some steering behaviors. Using a small constant value (conventionally called the epsilon parameter) avoids this problem.

When a group is found with a result that isn't small, its result is used to steer the character.

A pursuing character working in a team, for example, may have three groups: a collision avoidance group, a separation group, and a pursuit group. The collision avoidance group contains behaviors for obstacle avoidance, wall avoidance, and avoiding other characters. The separation group simply contains the separation behavior, which is used to avoid getting too close to other members of the chasing pack. The pursuit group contains the pursue steering behavior used to home in on the target.

If the character is far from any interference, the collision avoidance group will return with no desired acceleration. The separation group will then be considered but will also return with no action. Finally, the pursuit group will be considered, and the acceleration needed to continue the chase will be used. If the current motion of the character is perfect for the pursuit, this group may also return with no acceleration. In this case, there are no more groups to consider, so the character will have no acceleration, just as if they'd been exclusively controlled by the pursuit behavior.

In a different scenario, if the character is about to crash into a wall, the first group will return an acceleration that will help avoid the crash. The character will carry out this acceleration immediately, and the steering behaviors in the other groups will never be considered.

Pseudo-Code

The algorithm for priority-based steering is as follows:

```
# Should be a small value, effectively zero.
epsilon: float

class PrioritySteering:
    # Holds a list of BlendedSteering instances, which in turn
    # contain sets of behaviors with their blending weights.
    groups: BlendedSteering[]

    function getSteering( ) -> SteeringOutput:
        for group in groups:
            # Create the steering structure for accumulation.
            steering = group.getSteering( )

            # Check if we're above the threshold, if so return.
            if steering.linear.length( ) > epsilon or
               abs(steering.angular) > epsilon:
                return steering

        # If we get here, it means that no group had a large enough
        # acceleration, so return the small acceleration from the
        # final group.
        return steering
```

Data Structures and Interfaces

The priority steering algorithm uses a list of `BlendedSteering` instances. Each instance in this list makes up one group, and within that group the algorithm uses the code we created before to blend behaviors together.

Implementation Notes

The algorithm relies on being able to find the absolute value of a scalar (the angular acceleration) using the `abs` function. This function is found in the standard libraries of most programming languages.

The method also uses the `length` method to find the magnitude of a linear acceleration vector. Because we're only comparing the result with a fixed `epsilon` value, we could use the squared magnitude instead (making sure our epsilon value is suitable for comparing against a squared distance). This would save a square root calculation.

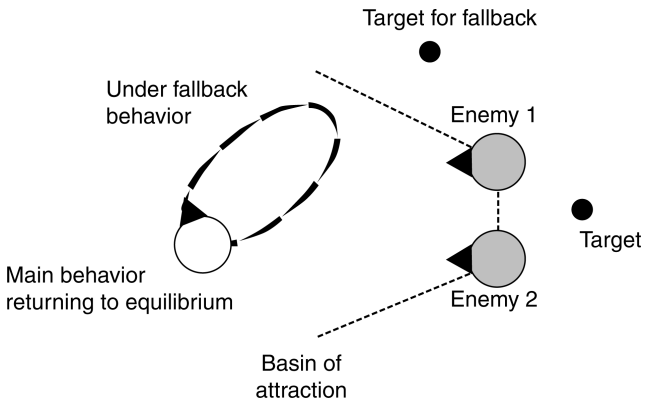

Figure 3.38: Priority steering avoiding unstable equilibrium

Performance

The algorithm requires only temporary storage for the acceleration. It is O(1) in memory. It is O(n) for time, where n is the total number of steering behaviors in all the groups. Once again, the practical execution speed of this algorithm depends on the efficiency of the `getSteering` methods for the steering behaviors it contains.

Equilibria Fallback

One notable feature of this priority-based approach is its ability to cope with stable equilibria. If a group of behaviors is in equilibrium, its total acceleration will be near zero. In this case the algorithm will drop down to the next group to get an acceleration.

By adding a single behavior at the lowest priority (wander is a good candidate), equilibria can be broken by reverting to a fallback behavior. This situation is illustrated in Figure 3.38.

Weaknesses

While this works well for unstable equilibria (it avoids the problem with slow creeping around the edge of an exclusion zone, for example), it cannot avoid large stable equilibria.

In a stable equilibrium the fallback behavior will engage at the equilibrium point and move the character out, whereupon the higher priority behaviors will start to generate acceleration requests. If the fallback behavior has not moved the character out of the basin of attraction, the higher priority behaviors will steer the character straight back to the equilibrium point. The character will oscillate in and out of equilibrium, but never escape.

Variable Priorities

The algorithm above uses a fixed order to represent priorities. Groups of behavior that appear earlier in the list will take priority over those appearing later in the list. In most cases priorities are fairly easy to fix; a collision avoidance, when activated, will always take priority over a wander behavior, for example.

In some cases, however, we'd like more control. A collision avoidance behavior may be low priority as long as the collision isn't imminent, becoming absolutely critical near the last possible opportunity for avoidance.

We can modify the basic priority algorithm by allowing each group to return a dynamic priority value. In the `PrioritySteering.getSteering` method, we initially request the priority values and then sort the groups into priority order. The remainder of the algorithm operates in exactly the same way as before.

Despite providing a solution for the occasional stuck character, there is only a minor practical advantage to using this approach. On the other hand, the process of requesting priority values and sorting the groups into order adds time. Although it is an obvious extension, my feeling is that if you are going in this direction, you may as well bite the bullet and upgrade to a full cooperative arbitration system.

3.4.4 COOPERATIVE ARBITRATION

So far we've looked at combining steering behaviors in an independent manner. Each steering behavior knows only about itself and always returns the same answer. To calculate the resulting steering acceleration, we select one or blend together several of these results. This approach has the advantage that individual steering behaviors are very simple and easily replaced. They can be tested on their own.

But as we've seen, there are a number of significant weaknesses in the approach that make it difficult to let characters loose without glitches appearing.

There is a trend toward increasingly sophisticated algorithms for combining steering behaviors. A core feature of this trend is the cooperation among different behaviors.

Suppose, for example, a character is chasing a target using a pursue behavior. At the same time it is avoiding collisions with walls. Figure 3.39 shows a possible situation. The collision is imminent and so needs to be avoided.

The collision avoidance behavior generates an avoidance acceleration away from the wall. Because the collision is imminent, it takes precedence, and the character is accelerated away.

The overall motion of the character is shown in Figure 3.39. It slows dramatically when it is about to hit the wall because the wall avoidance behavior is providing only a tangential acceleration.

The situation could be mitigated by blending the pursue and wall avoidance behaviors (although, as we've seen, simple blending would introduce other movement problems in situations with unstable equilibria). Even in this case it would still be noticeable because the forward acceleration generated by pursue is diluted by wall avoidance.

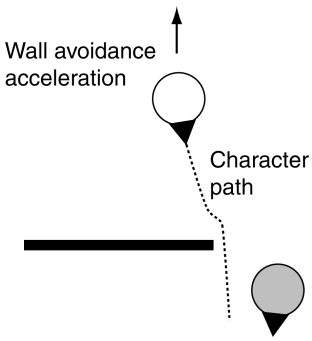

Figure 3.39: An imminent collision during pursuit

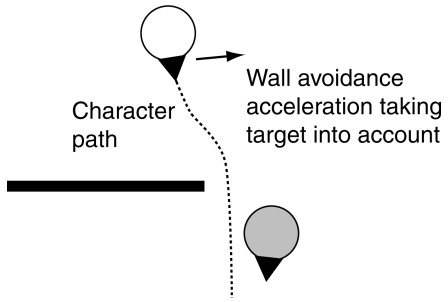

Figure 3.40: A context-sensitive wall avoidance

To get a believable behavior, we'd like the wall avoidance behavior to take into account what pursue is trying to achieve. Figure 3.40 shows a version of the same situation. Here the wall avoidance behavior is context sensitive; it understands where the pursue behavior is going, and it returns an acceleration which takes both concerns into account.

Obviously, taking context into account in this way increases the complexity of the steering algorithm. We can no longer use simple building blocks that selfishly do their own thing.

Many collaborative arbitration implementations are based on techniques we will cover in Chapter 5 on decision making. It makes sense; we're effectively making decisions about where and how to move. Decision trees, state machines, and blackboard architectures have all been used to control steering behaviors. Decision trees are the most common, as it is natural to implement each steering behavior as a decision tree node. Although less common, blackboard architectures are also particularly suited to cooperating steering behaviors; each

behavior is an expert that can read (from the blackboard) what other behaviors would like to do before having its own say.

Although decision trees are the most common, they can be finicky to get right. It isn't clear whether a breakthrough or another alternative approach will become the *de facto* standard for games. Cooperative steering behaviors is an area that many developers have independently stumbled across, and it is likely to be some time before any consensus is reached on an ideal implementation.

In the next section I will introduce the steering pipeline algorithm, an example of a dedicated approach that doesn't use the decision making technology in Chapter 5.

3.4.5 STEERING PIPELINE

The steering pipeline approach was pioneered by Marcin Chady, when we worked together at the AI middleware company Mindlathe. It was intended as an intermediate step between simply blending or prioritizing steering behaviors and implementing a complete movement planning solution (discussed in Chapter 4). It is a cooperative arbitration approach that allows constructive interaction between steering behaviors. It provides excellent performance in a range of situations that are normally problematic, including tight passages and integrating steering with pathfinding.

Bear in mind when reading this section that this is just one example of a cooperative arbitration approach that doesn't rely on other decision-making techniques. I am not suggesting this is the only way it can be done.

Algorithm

Figure 3.41 shows the general structure of the steering pipeline.

There are four stages in the pipeline: the targeters work out where the movement goal is, decomposers provide sub-goals that lead to the main goal, constraints limit the way a character can achieve a goal, and the actuator limits the physical movement capabilities of a character.

In all but the final stage, there can be one or more components. Each component in the pipeline has a different job to do. All are steering behaviors, but the way they cooperate depends on the stage.

Targeters

Targeters generate the top-level goal for a character. There can be several targets: a positional target, an orientation target, a velocity target, and a rotation target. We call each of these elements a channel of the goal (e.g., position channel, velocity channel). All goals in the algorithm can have any or all of these channels specified. An unspecified channel is simply a "don't care."

Individual channels can be provided by different behaviors (a chase-the-enemy targeter

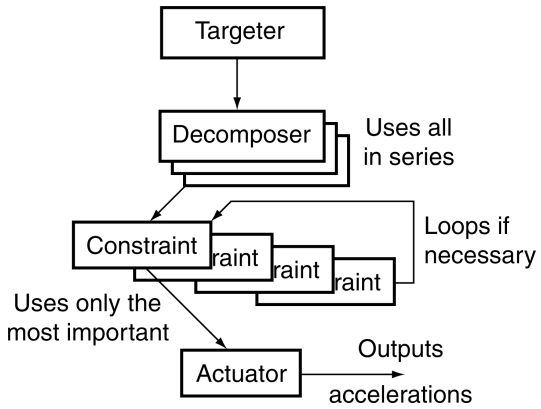

Figure 3.41: The steering pipeline

may generate the positional target, while a look-toward targeter may provide an orientation target), or multiple channels can be requested by a single targeter. When multiple targeters are used, only one may generate a goal in each channel. The algorithm we develop here trusts that the targeters cooperate in this way. No effort is made to avoid targeters overwriting previously set channels.

To the greatest extent possible, the steering system will try to fulfill all channels, although some sets of targets may be impossible to achieve all at once. We'll come back to this possibility in the actuation stage.

At first glance it can appear odd that we're choosing a single target for steering. Behaviors such as run away or avoid obstacle have goals to move away from, not to seek. The pipeline forces you to think in terms of the character's goal. If the goal is to run away, then the targeter needs to choose somewhere to run to. That goal may change from frame to frame as the pursuing enemy weaves and chases, but there will still be a single goal.

Other "away from" behaviors, like obstacle avoidance, don't become goals in the steering pipeline. They are constraints on the way a character moves and are found in the constraints stage.

Decomposers

Decomposers are used to split the overall goal into manageable sub-goals that can be more easily achieved.

The targeter may generate a goal somewhere across the game level, for example. A decomposer can check this goal, see that is not directly achievable, and plan a complete route (using a pathfinding algorithm, for example). It returns the first step in that plan as the sub-goal.

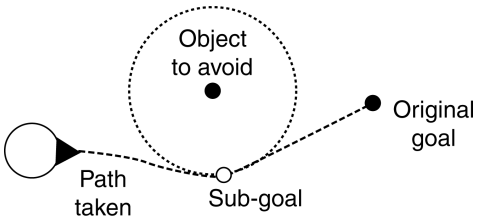

Figure 3.42: Collision avoidance constraint

This is the most common use for decomposers: to incorporate seamless path planning into the steering pipeline.

There can be any number of decomposers in the pipeline, and their order is significant. We start with the first decomposer, giving it the goal from the targeter stage. The decomposer can either do nothing (if it can't decompose the goal) or can return a new sub-goal. This sub-goal is then passed to the next decomposer, and so on, until all decomposers have been queried.

Because the order is strictly enforced, we can perform hierarchical decomposition very efficiently. Early decomposers should act broadly, providing large-scale decomposition. For example, they might be implemented as a coarse pathfinder. The sub-goal returned may still be a long way from the character. Later decomposers can then refine the sub-goal by decomposing it. Because they are decomposing only the sub-goal, they don't need to consider the big picture, allowing them to decompose in more detail. This approach will seem familiar when we look at hierarchical pathfinding in the next chapter. With a steering pipeline in place, we don't need a hierarchical pathfinding engine; we can simply use a set of decomposers pathfinding on increasingly detailed graphs.

Constraints

Constraints limit the ability of a character to achieve its goal or sub-goal. They detect if moving toward the current sub-goal is likely to violate the constraint, and if so, they suggest a way to avoid it. Constraints tend to represent obstacles: moving obstacles like characters or static obstacles like walls.

Constraints are used in association with the actuator, described below. The actuator works out the path that the character will take toward its current sub-goal. Each constraint is allowed to review that path and determine if it is sensible. If the path will violate a constraint, then it returns a new sub-goal that will avoid the problem. The actuator can then work out the new path and check if that one works and so on, until a valid path has been found.

It is worth bearing in mind that the constraint may only provide certain channels in its sub-goal. Figure 3.42 shows an upcoming collision. The collision avoidance constraint could generate a positional sub-goal, as shown, to force the character to swing around the obstacle.

Equally, it could leave the position channel alone and suggest a velocity pointing away from the obstacle, so that the character drifts out from its collision line. The best approach depends to a large extent on the movement capabilities of the character and, in practice, takes some experimentation.

Of course, solving one constraint may violate another constraint, so the algorithm may need to loop around to find a compromise where every constraint is happy. This isn't always possible, and the steering system may need to give up trying to avoid getting into an endless loop. The steering pipeline incorporates a special steering behavior, deadlock, that is given exclusive control in this situation. This could be implemented as a simple wander behavior in the hope that the character will wander out of trouble. For a complete solution, it could call a comprehensive movement planning algorithm.

The steering pipeline is intended to provide believable yet lightweight steering behavior, so that it can be used to simulate a large number of characters. We could replace the current constraint satisfaction algorithm with a full planning system, and the pipeline would be able to solve arbitrary movement problems. I've found it best to stay simple, however. In the majority of situations, the extra complexity isn't needed, and the basic algorithm works fine.

As it stands, the algorithm is not always guaranteed to direct an agent through a complex environment. The deadlock mechanism allows us to call upon a pathfinder or another higher level mechanism to get out of trickier situations. The steering system has been specially designed to allow you to do that only when necessary, so that the game runs at the maximum speed. Always use the simplest algorithms that work.

The Actuator

Unlike each of the other stages of the pipeline, there is only one actuator per character. The actuator's job is to determine how the character will go about achieving its current sub-goal. Given a sub-goal and its internal knowledge about the physical capabilities of the character, it returns a path indicating how the character will move to the goal.

The actuator also determines which channels of the sub-goal take priority and whether any should be ignored.

For simple characters, like a walking sentry or a floating ghost, the path can be extremely simple: head straight for the target. The actuator can often ignore velocity and rotation channels and simply make sure the character is facing the target.

If the actuator does honor velocities, and the goal is to arrive at the target with a particular velocity, we may choose to swing around the goal and take a run up, as shown in Figure 3.43.

More constrained characters, like an AI-controlled car, will have more complex actuation: the car can't turn while stationary, it can't move in any direction other than the one in which it is facing, and the grip of the tires limits the maximum turning speed. The resulting path may be more complicated, and it may be necessary to ignore certain channels. For example, if the sub-goal wants us to achieve a particular velocity while facing in a different direction, then we know the goal is impossible. Therefore, we will probably throw away the orientation channel.

In the context of the steering pipeline, the complexity of actuators is often raised as a prob-

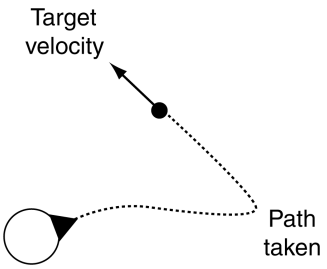

Figure 3.43: Taking a run up to achieve a target velocity

lem with the algorithm. It is worth bearing in mind that this is an implementation decision; the pipeline supports comprehensive actuators when they are needed (and you obviously have to pay the price in execution time), but they also support trivial actuators that take virtually no time at all to run.

Actuation as a general topic is covered later in this chapter, so we'll avoid getting into the grimy details at this stage. For the purpose of this algorithm, we will assume that actuators take a goal and return a description of the path the character will take to reach it.

Eventually, we'll want to actually carry out the steering. The actuator's final job is to return the forces and torques (or other motor controls—see Section 3.8 for details) needed to achieve the predicted path.

Pseudo-Code

The steering pipeline is implemented with the following algorithm:

```
class SteeringPipeline:
    character: Kinematic

    # Lists of components at each stage of the pipe.
    targeters: Targeter[]
    decomposers: Decomposer[]
    constraints: Constraint[]
    actuator: Actuator

    # The number of attempts the algorithm will make to find an
    # unconstrained route.
    constraintSteps: int

    # The deadlock steering behavior.
    deadlock: SteeringBehavior

```

```
    function getSteering() -> SteeringOutput:
        # Firstly we get the top level goal.
        goal: Goal = new Goal()
        for targeter in targeters:
            targeterGoal = targeter.getGoal(character)
            goal.updateChannels(targeterGoal)

        # Now we decompose it.
        for decomposer in decomposers:
            goal = decomposer.decompose(character, goal)

        # Now we loop through the actuation and constraint process.
        for i in 0..constraintSteps:
            # Get the path from the actuator.
            path = actuator.getPath(character, goal)

            # Check for constraint violation.
            for constraint in constraints:
                # If we find a violation, get a suggestion.
                if constraint.isViolated(path):
                    goal = constraint.suggest(character, path, goal)

                    # Go to the next iteration of the 'for i in ...'
                    # loop to try for the new goal.
                    break continue

            # If we're here it is because we found a valid path.
            return actuator.output(character, path, goal)

        # We arrive here if we ran out of constraint steps.
        # We delegate to the deadlock behavior.
        return deadlock.getSteering()
```

Data Structures and Interfaces

We are using interface classes to represent each component in the pipeline. At each stage, a different interface is needed.

Targeter

Targeters have the form:

```
class Targeter:
    function getGoal(character: Kinematic) -> Goal
```

The `getGoal` function returns the targeter's goal.

Decomposer

Decomposers have the interface:

```
class Decomposer:
    function decompose(character: Kinematic, goal: Goal) -> Goal
```

The `decompose` method takes a goal, decomposes it if possible, and returns a sub-goal. If the decomposer cannot decompose the goal, it simply returns the goal it was given.

Constraint

Constraints have two methods:

```
class Constraint:
    function willViolate(path: Path) -> bool
    function suggest(character: Kinematic,
                     path: Path,
                     goal: Goal) -> Goal
```

The `willViolate` method returns true if the given path will violate the constraint at some point. The `suggest` method should return a new goal that enables the character to avoid violating the constraint. We can make use of the fact that `suggest` always follows a positive result from `willViolate`. Often, `willViolate` needs to perform calculations to determine if the path poses a problem. If it does, the results of these calculations can be stored in the class and reused in the `suggest` method that follows. The calculation of the new goal can be entirely performed in the `willViolate` method, leaving the `suggest` method to simply return the result. Any channels not needed in the suggestion should take their values from the current goal passed into the method.

Actuator

The actuator creates paths and returns steering output:

```
class Actuator:
    function getPath(character: Kinematic, goal: Goal) -> Path
    function output(character: Kinematic,
                    path: Path,
                    goal: Goal) -> SteeringOutput
```

The `getPath` function returns the route that the character will take to the given goal. The output function returns the steering output for achieving the given path.

Deadlock

The deadlock behavior is a general steering behavior. Its `getSteering` function returns a steering output that is simply returned from the steering pipeline.

Goal

Goals need to store each channel, along with an indication as to whether the channel should be used. The `updateChannel` method sets appropriate channels from another goal object. The structure can be implemented as:

```
class Goal:
    # Flags to indicate if each channel is to be used.
    hasPosition: bool = false
    hasOrientation: bool = false
    hasVelocity: bool = false
    hasRotation: bool = false

    # Data for each channel.
    position: Vector
    orientation: float
    velocity: Vector
    rotation: float

    # Updates this goal.
    function updateChannels(other: Goal):
        if other.hasPosition:
            position = other.position
            hasPosition = true
        if other.hasOrientation:
            orientation = other.orientation
            hasOrientation = true
        if other.hasVelocity:
            velocity = other.velocity
            hasVelocity = true
        if other.hasRotation:
            rotation = other.rotation
            hasRotation = true
```

Paths

In addition to the components in the pipeline, I have used an opaque data structure for the path. The format of the path doesn't affect this algorithm. It is simply passed between steering components unaltered.

In real projects I've used two different path implementations to drive the algorithm. Pathfinding-style paths, made up of a series of line segments, give point-to-point movement information. They are suitable for characters who can turn very quickly—for example, human beings walking. Point-to-point paths are very quick to generate, they can be extremely quick to check for constraint violation, and they can be easily turned into forces by the actuator.

The original version of this algorithm used a more general path representation. Paths are made up of a list of maneuvers, such as "accelerate" or "turn with constant radius." They are

suitable for the most complex steering requirements, including race car driving, which is a difficult test for any steering algorithm. This type of path can be more difficult to check for constraint violation, however, because it involves curved path sections.

It is worth experimenting to see if your game can make do with straight line paths before implementing more complex structures.

Performance

The algorithm is O(1) in memory. It uses only temporary storage for the current goal.

It is O(cn) in time, where c is the number of constraint steps and n is the number of constraints. Although c is a constant (and we could therefore say the algorithm is O(n) in time), it helps to increase its value as more constraints are added to the pipeline. In the past I've used a number of constraint steps similar to the number of constraints, giving an algorithm O(n^2) in time.

The constraint violation test is at the lowest point in the loop, and its performance is critical. Profiling a steering pipeline with no decomposers will show that most of the time spent executing the algorithm is normally spent in this function.

Since decomposers normally provide pathfinding, they can be very long running, even though they will be inactive for much of the time. For a game where the pathfinders are extensively used (i.e., the goal is always a long way away from the character), the speed hit will slow the AI unacceptably. The steering algorithm needs to be split over multiple frames.

Example Components

Actuation will be covered in Section 3.8 later in the chapter, but it is worth taking a look at a sample steering component for use in the targeter, decomposer, and constraint stages of the pipeline.

Targeter

The chase targeter keeps track of a moving character. It generates its goal slightly ahead of its victim's current location, in the direction the victim is moving. The distance ahead is based on the victim's speed and a lookahead parameter in the targeter.

```
1  class ChaseTargeter extends Targeter:
2      chasedCharacter: Kinematic
3
4      # Controls how much to anticipate the movement.
5      lookahead: float
6
```

```
    function getGoal(kinematic):
        goal = new Goal()
        goal.position = chasedCharacter.position +
                        chasedCharacter.velocity * lookahead
        goal.hasPosition = true
        return goal
```

Decomposer

The pathfinding decomposer performs pathfinding on a graph and replaces the given goal with the first node in the returned plan. See Chapter 4 on pathfinding for more information.

```
class PlanningDecomposer extends Decomposer:
    graph: Graph
    heuristic: function(GraphNode, GraphNode) -> float

    function decompose(character: Kinematic, goal: Goal) -> Goal:
        # First we quantize our current location and our goal into
        # nodes of the graph.
        start: GraphNode = graph.getNode(kinematic.position)
        end: GraphNode = graph.getNode(goal.position)

        # If they are equal, we don't need to plan.
        if startNode == endNode:
            return goal

        # Otherwise plan the route.
        path = pathfindAStar(graph, start, end, heuristic)

        # Get the first node in the path and localize it.
        firstNode: GraphNode = path[0].asNode
        position: Vector = graph.getPosition(firstNode)

        # Update the goal and return.
        goal.position = position
        goal.hasPosition = true
        return goal
```

Constraint

The avoid obstacle constraint treats an obstacle as a sphere, represented as a single 3D point and a constant radius. For simplicity, I assume that the path provided by the actuator is a series of line segments, each with a start point and an end point.

```
class AvoidObstacleConstraint extends Constraint:
    # The obstacle bounding sphere.
    center: Vector
```

```
    radius: float

    # Holds a margin of error by which we'd ideally like
    # to clear the obstacle. Given as a proportion of the
    # radius (i.e. should be > 1.0).
    margin: float

    # If a violation occurs, stores the part of the path
    # that caused the problem.
    problemIndex: int

    function willViolate(path: Path) -> bool:
        # Check each segment of the path in turn.
        for i in 0..len(path):
            segment = path[i]

            # If we have a clash, store the current segment.
            if distancePointToSegment(center, segment) < radius:
                problemIndex = i
                return true

        # No segments caused a problem.
        return false

    function suggest(_, path: Path, goal: Goal) -> Goal:
        # Find the closest point on the segment to the sphere center.
        segment = path[problemIndex]
        closest = closestPointOnSegment(segment, center)

        # Check if we pass through the center point.
        if closest.length() == 0:
            # Any vector at right angles to the segment will do.
            direction = segment.end - segment.start
            newDirection = direction.anyVectorAtRightAngles()
            newPosition = center + newDirection * radius * margin

        # Otherwise project the point out beyond the radius.
        else:
            offset = closest - center
            newPosition = center +
                          offset * radius * margin / closest.length()

        # Set up the goal and return.
        goal.position = newPosition
        goal.hasPosition = true
        return goal
```

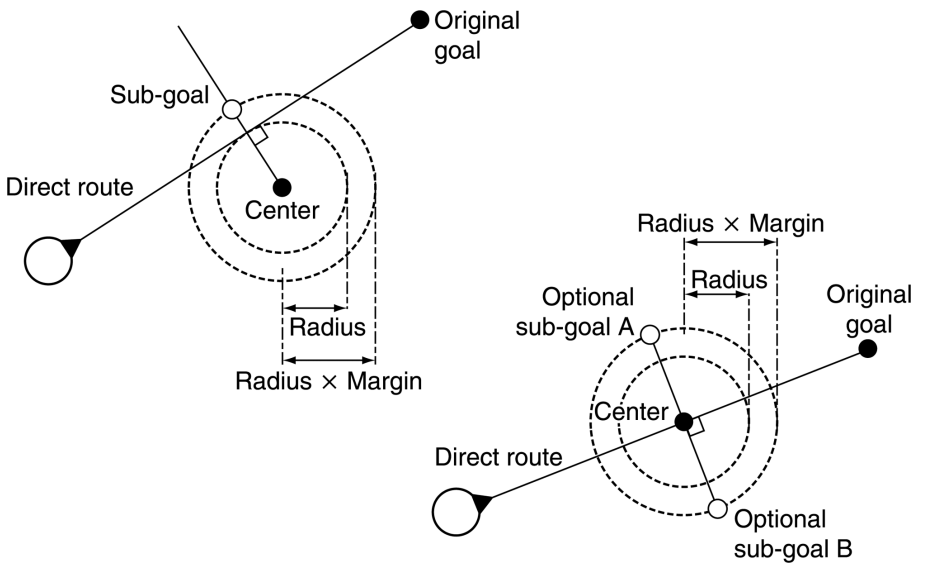

Figure 3.44: Obstacle avoidance projected and at right angles

The `suggest` method appears more complex that it actually is. We find a new goal by finding the point of closest approach and projecting it out so that we miss the obstacle by far enough. We need to check that the path doesn't pass right through the center of the obstacle, however, because in that case we can't project the center out. If it does, we use any point around the edge of the sphere, at a tangent to the segment, as our target. Figure 3.44 shows both situations in two dimensions and also illustrates how the margin of error works.

I added the `anyVectorAtRightAngles` method just to simplify the listing. It returns a new vector at right angles to its instance. This is normally achieved by using a cross product with some reference direction and then returning a cross product of the result with the original direction. This will not work if the reference direction is the same as the vector we start with. In this case a backup reference direction is needed.

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 to have your characters 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 or priorities.

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, which tracked the current position of the ball and moved the bat so that it would rebound, 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 a simple example of this already. The pursue steering behavior predicted the future position of its target by assuming it would carry on with its current velocity. A more complex prediction 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 the vast majority of action games, 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 rocket-propelled grenades (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 result in your bullets landing in the dirt at their feet. Predicting correct trajectories is now a core part of the AI in shooters.

Figure 3.45: Parabolic arc

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, as shown in Figure 3.45.

The projectile moves according to the formula:

$$\vec{p_t} = \vec{p_0} + \vec{u}s_m t + \frac{\vec{g}t^2}{2} \tag{3.1}$$

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 three dimensional 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. 9.81ms^{-2} in the down direction) this can look too slow in a game environment. Physics programmers often choose a value around double that for games, although some tweaking is needed to get the exact look. Even 9.81ms^{-2} may appear far too fast for certain objects, however, particularly the jumping of characters. To get the correct overall effect it isn't uncommon to have different objects with different gravity.

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 a target 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 character 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):

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

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:

$$\vec{p_i} = \begin{bmatrix} p_{x0} + u_x s_m t_i + \frac{1}{2} g_x t_i^2 \\ p_{yi} \\ p_{z0} + u_z s_m t_i + \frac{1}{2} g_z t_i^2 \end{bmatrix} \tag{3.3}$$

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

$$\vec{p_i} = \begin{bmatrix} p_{x0} + u_x s_m t_i \\ p_{yi} \\ p_{z0} + u_z s_m t_i \end{bmatrix} \tag{3.4}$$

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, 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 predicted impact point 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 in which to fire (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 their 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 their grenade more slowly so 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 (often trials are limited to a few fixed values: 'throw fast', 'throw slow' and 'drop', for example). For the purpose of this book we can assume that s_m is constant and known in advance.

The quadratic equation (Equation 3.1) has vector coefficients; add to this the requirement that the firing vector should be normalized,

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

and we have four equations in four unknowns:

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

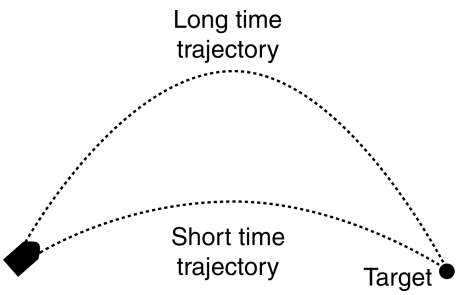

Figure 3.46: Two possible firing solutions

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

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

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 we are only interested in the positive solutions. Note that we should strictly take into account 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:

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

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).

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

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:

```
function calculateFiringSolution(start: Vector,
                                 end: Vector,
                                 muzzleV: float,
                                 gravity: Vector) -> Vector:
    # Calculate the vector from the target back to the start.
    delta: Vector = start - end

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

    # Check for no real solutions.
    b2minus4ac = b * b - 4 * a * c
    if b2minus4ac < 0:
        return null

    # Find the candidate times.
    time0 = sqrt((-b + sqrt(b2minus4ac)) / (2 * a))
    time1 = sqrt((-b - sqrt(b2minus4ac)) / (2 * a))

    # Find the time to target.
    if time0 < 0:
        if time1 < 0:
            # We have no valid times.
            return null
        else:
            ttt = time1
    else:
        if time1 < 0:
            ttt = time0
        else:
            ttt = min(time0, time1)

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

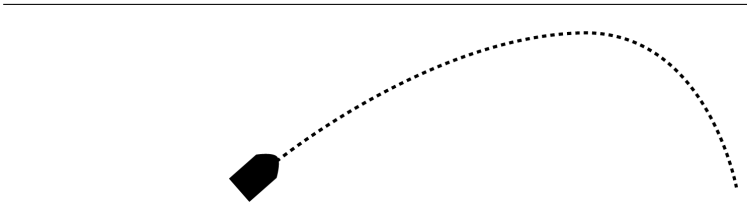

Figure 3.47: Projectile moving with drag

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.

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 projectile 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 (in one dimension) by:

$$D = -kv - cv^2$$

where v is the velocity of the projectile, and k and c are both constants. The k coefficient is sometimes called the viscous drag and c the aerodynamic drag (or ballistic coefficient). These terms are somewhat confusing, however, because they do not correspond directly to real-world viscous or aerodynamic drag.

Adding these terms changes the equation of motion from a simple expression into a second-order differential equation:

$$\ddot{\vec{p_t}} = g - k\dot{\vec{p_t}} - c\dot{\vec{p_t}}|\dot{\vec{p_t}}|$$

Unfortunately, the second term in the equation, $c\dot{\vec{p_t}}|\dot{\vec{p_t}}|$, is where the complications set in. It relates the drag in one direction to the drag in another direction. Up to this point, we've assumed that for each of the three dimensions the projectile motion is independent of what is happening in the other directions. Here the drag is relative to the total speed of the projectile: even if it is moving slowly in the x-direction; for example, it will experience a great deal of drag if it is moving quickly in the z-direction. This is the characteristic of a non-linear differential equation, and with this term included there can be no simple equation for the firing solution.

Our only option is to use an iterative method that performs a simulation of the projectile's flight. We will return to this approach below.

More progress can be made if we remove the second term to give:

$$\ddot{\vec{p_t}} = g - k\dot{\vec{p_t}} \tag{3.5}$$

While this makes the mathematics tractable, it isn't the most common setup for a physics engine. If you need very accurate firing solutions and you have control over the kind of physics you are running, this may be an option. Otherwise, you will need to use an iterative method.

We can solve this equation to get an equation for the motion of the particle. If you're not interested in the math, you can skip to the next section.

Omitting the derivations, we solve Equation 3.5 and find that the trajectory of the particle is given by:

$$\vec{p_t} = \frac{\vec{g}t - \vec{A}e^{-kt}}{k} + \vec{B} \tag{3.6}$$

where $\vec{A}$ and $\vec{B}$ are constants found from the position and velocity of the particle at time $t = 0$:

$$\vec{A} = s_m\vec{u} - \frac{\vec{g}}{k}$$

and

$$\vec{B} = \vec{p_0} - \frac{\vec{A}}{k}$$

We can use this equation for the path of the projectile on its own, if it corresponds to the drag in our physics (or if accuracy is less important). Or we can use it as the basis of an iterative algorithm in more complex physics systems.

Rotating and Lift

Another complication in the movement calculations occurs if the projectile is rotating while it is in flight.

We have treated all projectiles as if they are not rotating during their flight. Spinning projectiles (golf balls, for example) have additional lift forces applying to them as a result of their spin and are more complex still to predict. If you are developing an accurate golf game that simulates this effect (along with wind that varies over the course of the ball's flight), then it is likely to be impossible to solve the equations of motion directly. The best way to predict where the ball will land is to run it through your simulation code (possibly with a coarse simulation resolution, for speed).

3.5.5 ITERATIVE TARGETING

When we cannot create an equation for the firing solution, or when such an equation would be very complex or prone to error, we can use an iterative targeting technique. This is similar to the way that long-range weapons and artillery (euphemistically called "effects" in military speak) are really targeted.

The Problem

We would like to be able to determine a firing solution that hits a given target, even if the equations of motion for the projectile cannot be solved or if we have no simple equations of motion at all.

The generated firing solution may be approximate (i.e., it doesn't matter if we are slightly off center as long as we hit), but we need to be able to control its accuracy to make sure we can hit small or large objects correctly.

The Algorithm

The process has two stages. We initially make a guess as to the correct firing solution. The trajectory equations are then processed to check if the firing solution is accurate enough (i.e., does it hit the target?). If it is not accurate, then a new guess is made, based on the previous guess.

The process of testing involves checking how close the trajectory gets to the target location. In some cases we can find this mathematically from the equations of motion (although it is very likely that if we can find this, then we could also solve the equation of motion and find a firing solution without an iterative method). In most cases the only way to find the closest approach point is to follow a projectile through its trajectory and record the point at which it made its closest approach.

To make this process faster, we only test at intervals along the trajectory. For a relatively

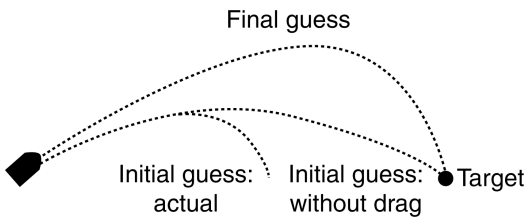

Figure 3.48: Refining the guess

slow-moving projectile with a simple trajectory, we might check every half second. For a fast-moving object with complex wind, lift, and aerodynamic forces, we may need to test every tenth of a second, or at worst the simulation frequency of the game. The position of the projectile is calculated at each time interval. These positions are linked by straight line segments, and we find the nearest point to our target on this line segment. We are approximating the trajectory by a piecewise linear curve.

We can add additional tests to avoid checking too far in the future. This is not normally a full collision detection process, because of the time that would take, but we do a simple test such as stopping when the projectile's height is a good deal lower than its target.

The initial guess for the firing solution can be generated from the firing solution function described earlier; that is, we assume there is no drag or other complex movement in our first guess.

After the initial guess, the refinement depends to some extent on the forces that exist in the game. If no wind is being simulated, then the direction of the first-guess solution in the x–z plane will be correct (called the "bearing"). We only need to tweak the angle between the x–z plane and the firing direction (called the "elevation"). This is shown in Figure 3.48.

If we have a drag coefficient, then the elevation will need to be higher than that generated by the initial guess. If the projectile experiences no lift, then the maximum elevation should be 45°. Any higher than that and the total flight distance will start decreasing again. If the projectile does experience lift, then it might be better to send it off higher, allowing it to fly longer and to generate more lift, which will increase its distance.

If we have a crosswind, then just adjusting the elevation will not be enough. We will also need to adjust the bearing. It is a good idea to iterate between the two adjustments in series: getting the elevation right first for the correct distance, then adjusting the bearing to get the projectile to land in the direction of the target, then adjusting the elevation to get the right distance, and so on.

You would be quite right if you get the impression that refining the guesses is akin to complete improvisation. In fact, real targeting systems for military weapons use complex simulations for the flights of their projectiles and a range of algorithms, heuristics, and search techniques to find the best solution. In games, the best approach is to get the AI running in a

real game environment and adjust the guess refinement rules until good results are generated quickly.

Whatever the sequence of adjustment or the degree to which the refinement algorithm takes into account physical laws, a good starting point is a binary search, the stalwart of many algorithms in computer science, described in depth in any good text on algorithmics or computer science.

Pseudo-Code

Because the refinement algorithm depends to a large extent on the kind of forces we are modeling in the game, the pseudo-code presented below will assume that we are trying to find a firing solution for a projectile moving with drag alone. This allows us to simplify the search from a search for a complete firing direction to just a search for an angle of elevation.

This is the most complex technique I've seen in a commercial action game for this situation, although, as I described, in military simulation more complex situations occur.

The code uses the equation of motion for a projectile experiencing only viscous drag, as we derived earlier.

```
function refineTargeting(start: Vector,
                         end: Vector,
                         muzzleV: float,
                         gravity: Vector,
                         margin: float) -> Vector:

    # Calculate a firing solution based on a firing angle.
    function checkAngle(angle):
        deltaPosition: Vector = target - source
        direction = convertToDirection(deltaPosition, angle)
        distance = distanceToTarget(direction, source, target, muzzleV)
        return direction, distance

    # Take an initial guess using the dragless firing solution.
    direction: Vector = calculateFiringSolution(
        source, target, muzzleVelocity, gravity)

    # Check if this is good enough.
    distance = distanceToTarget(direction, source, target, muzzleV)
    if -margin < distance < margin:
        return direction

    # Otherwise we will binary search, but we must ensure our minBound
    # undersoots and our maxBound overshoots.
    angle: float = asin(direction.y / direction.length())
    if distance > 0:
        # We've found a maximum bound. Use the shortest possible shot
```

```
        # (shooting straight down) as the minimum bound.
        maxBound = angle
        minBound = - pi / 2
        direction, distance = checkAngle(minBound)
        if -margin < distance < margin:
            return direction

    # Otherwise we need to check we can find a maximum bound: maximum
    # distance is achieved when we fire at 45 degrees = pi / 4.
    else:
        minBound = angle
        maxBound = pi / 4
        direction, distance = checkAngle(maxBound)
        if -margin < distance < margin:
            return direction

        # Check if our longest shot can't make it.
        if distance < 0:
            return null

    # Now we have a minimum and maximum bound, so binary search.
    distance = infinity
    while abs(distance) >= margin:
        angle = (maxBound - minBound) / 2
        direction, distance = checkAngle(angle)

        # Change the appropriate bound.
        if distance < 0:
            minBound = angle
        else:
            maxBound = angle

    return direction
```

Data Structures and Interfaces

In the code we rely on three functions. The `calculateFiringSolution` function is the function we defined earlier. It is used to create a good initial guess.

The `distanceToTarget` function runs the physics simulator and returns how close the projectile got to the target. The sign of this value is critical. It should be positive if the projectile overshot its target and negative if it undershot. Simply performing a 3D distance test will always give a positive distance value, so the simulation algorithm needs to determine whether the miss was too far or too near and set the sign accordingly.

The `convertToDirection` function creates a firing direction from an angle. It can be implemented in the following way:

```
function convertToDirection(deltaPosition: Vector, angle: float):
    # Find the planar direction.
    direction = deltaPosition
    direction.y = 0
    direction.normalize()

    # Add in the vertical component.
    direction *= cos(angle)
    direction.y = sin(angle)

    return direction
```

Performance

The algorithm is O(1) in memory and O($r \log n^{-1}$) in time, where r is the resolution of the sampling we use in the physics simulator for determining the closest approach to target, and n is the accuracy threshold that determines if a hit has been found.

Iterative Targeting without Motion Equations

Although the algorithm given above treats the physical simulation as a black-box, in the discussion we assumed that we could implement it by sampling the equations of motion at some resolution.

The actual trajectory of an object in the game may be affected by more than just mass and velocity. Drag, lift, wind, gravity wells, and all manner of other exotica can change the movement of a projectile. This can make it impossible to calculate a motion equation to describe where the projectile will be at any point in time.

If this is the case, then we need a different method of following the trajectory to determine how close to its target it gets. The real projectile motion, once it has actually been released, is likely to be calculated by a physics system. We can use the same physics system to perform miniature simulations of the motion for targeting purposes.

At each iteration of the algorithm, the projectile is set up and fired, and the physics is updated (normally at relatively coarse intervals compared to the normal operation of the engine; extreme accuracy is probably not needed). The physics update is repeatedly called, and the position of the projectile after each update is recorded, forming the piecewise linear curve we saw previously. This is then used to determine the closest point of the projectile to the target.

This approach has the advantage that the physical simulation can be as complex as necessary to capture the dynamics of the projectile's motion. We can even include other factors, such as a moving target.

On the other hand, this method requires a physics engine that can easily set up isolated simulations. If your physics engine is only optimized for having one simulation at a time (i.e., the current game world), then this will be a problem. Even if the physics system allows it, the technique can be time consuming. It is only worth contemplating when simpler methods (such as assuming a simpler set of forces for the projectile) give visibly poor results.

Other Uses of Prediction

Prediction of projectile motion is the most complex common type of motion prediction in games.

In games involving collisions as an integral part of gameplay, such as ice hockey games and pool or snooker simulators, the AI may need to be able to predict the results of impacts. This is commonly done using an extension of the iterative targeting algorithm: we have a go in a simulation and see how near we get to our goal.

Throughout this chapter I've used another prediction technique that is so ubiquitous that developers often fail to realize that its purpose is to predict motion.

In the pursue steering behavior, for example, the AI aims its motion at a spot some way in front of its target, in the direction the target is moving.It assumes that the target will continue to move in the same direction at the current speed, and it chooses a target position to effectively cut it off. If you remember playing tag at school, the good players did the same thing: predict the motion of the player they wanted to catch or evade.

We can add considerably more complex prediction to a pursuit behavior, making a genuine prediction as to a target's motion (if the target is coming up on a wall, for example, we know it won't carry on in the same direction and speed; it will swerve to avoid impact). Complex motion prediction for chase behaviors is the subject of active academic research (and is beyond the scope of this book). Despite the body of research done, games still use the simple version, assuming the prey will keep doing what they are doing.

Motion prediction is also used outside character-based AI. Networking technologies for multi-player games need to cope when the details of a character's motion have been delayed or disrupted by the network. In this case, the server can use a motion prediction algorithm (which is almost always the simple "keep doing what they were doing" approach) to guess where the character might be. If it later finds out it was wrong, it can gradually move the character to its correct position (common in massively multi-player games) or snap it immediately there (more common in shooters), depending on the needs of the game design.

3.6 JUMPING

Jumping is a significant problem for character movement in shooters. The regular steering algorithms are not designed to incorporate jumps, which are a core part of the shooter genre.

Jumps are inherently risky. Unlike other steering actions, they can fail, and such a failure may make it difficult or impossible to recover (at the very limit, it may kill the character).

For example, consider a character chasing an enemy around a flat level. The steering algorithm estimates that the enemy will continue to move at its current speed and so sets the character's trajectory accordingly. The next time the algorithm runs (usually the next frame, but it may be a little later if the AI is running every few frames) the character finds that its estimate was wrong and that its target has decelerated fractionally. The steering algorithm again assumes that the target will continue at its current speed and estimates again. Even though the character is decelerating, the algorithm can assume that it is not. Each decision it makes can be fractionally wrong, and the algorithm can recover the next time it runs. The cost of the error is almost zero.

By contrast, if a character decides to make a jump between two platforms, the cost of an error may be greater. The steering controller needs to make sure that the character is moving at the correct speed and in the correct direction and that the jump action is executed at the right moment (or at least not too late). Slight perturbations in the character's movement (caused by clipping an obstacle, for example, from gun recoil, or the blast wave from an explosion) can lead to the character missing the landing spot and plummeting to its doom, a dramatic failure.

Steering behaviors effectively distribute their thinking over time. Each decision they make is very simple, but because they are constantly reconsidering the decision, the overall effect is competent. Jumping is a one-time, failure-sensitive decision.

3.6.1 JUMP POINTS

The simplest support for jumps puts the onus on the level designer. Locations in the game level are labeled as being jump points. These regions need to be manually placed. If characters can move at many different speeds, then jump points also have an associated minimum velocity set. This is the velocity at which a character needs to be traveling in order to make the jump.

Depending on the implementation, characters either may seek to get as near their target velocity as possible or may simply check that the component of their velocity in the correct direction is sufficiently large.

Figure 3.49 shows two walkways with a jump point placed at their nearest point. A character that wishes to jump between the walkways needs to have enough velocity heading toward the other platform to make the jump. The jump point has been given a minimum velocity in the direction of the other platform.

In this case it doesn't make sense for a character to try to make a run up in that exact direction. The character should be allowed to have any velocity with a sufficiently large component in the correct direction, as shown in Figure 3.50.

If the structure of the landing area is a little different, however, the same strategy would result in disaster. In Figure 3.51 the same run up has disastrous results.

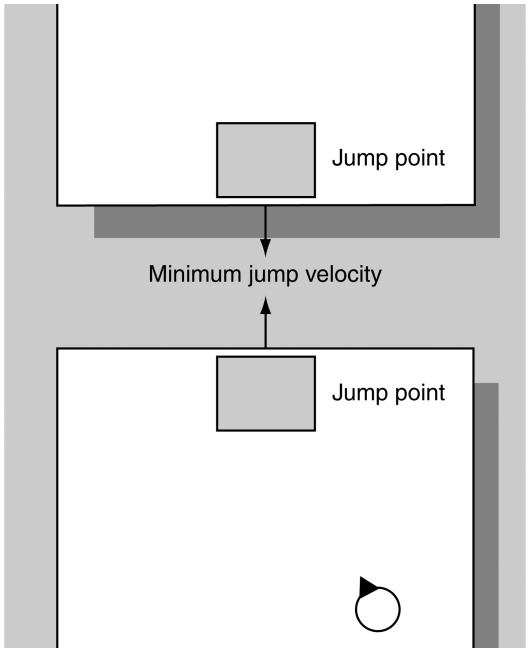

Figure 3.49: Jump points between walkways

Achieving the Jump

To achieve the jump, the character can use a velocity matching steering behavior to take a run up. For the period before its jump, the movement target is the jump point, and the velocity the character is matching is that given by the jump point. As the character crosses onto the jump point, a jump action is executed, and the character becomes airborne.

This approach requires very little processing at runtime:

1. The character needs to decide to make a jump. It may use some pathfinding system to determine that it needs to be on the other side of the gap, or else it may be using a simple steering behavior and be drawn toward the ledge.
2. The character needs to recognize which jump it will make. This will normally happen automatically when we are using a pathfinding system (see the section on jump links, below). If we are using a local steering behavior, then it can be difficult to determine that a jump is ahead in enough time to make it. A reasonable lookahead is required.
3. Once the character has found the jump point it is using, a new steering behavior takes over. That behavior performs velocity matching to bring the character into the jump point with the correct velocity and direction.

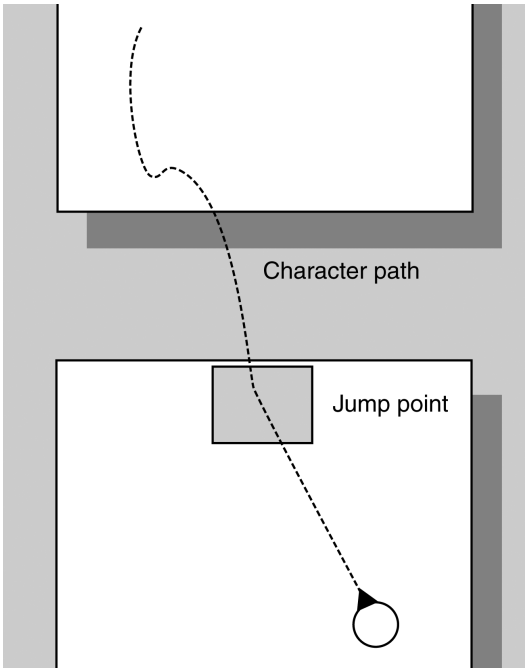

Figure 3.50: Flexibility in the jump velocity

4. When the character touches the jump point, a jump action is requested. The character doesn't need to work out when or how to jump, it simply gets thrown into the air as it hits the jump point.

Weaknesses

The examples at the start of this section hint at the problems suffered by this approach. In general, the jump point does not contain enough information about the difficulty of the jump for every possible jumping case.

Figure 3.52 illustrates a number of different jumps that are difficult to mark up using jump points. Jumping onto a thin walkway requires velocity in exactly the right direction, jumping onto a narrow ledge requires exactly the right speed, and jumping onto a pedestal involves correct speed and direction. Notice that the difficulty of the jump also depends on the direction it is taken from. Each of the jumps in Figure 3.52 would be easy in the opposite direction.

In addition, not all failed jumps are equal. A character might not mind occasionally miss-

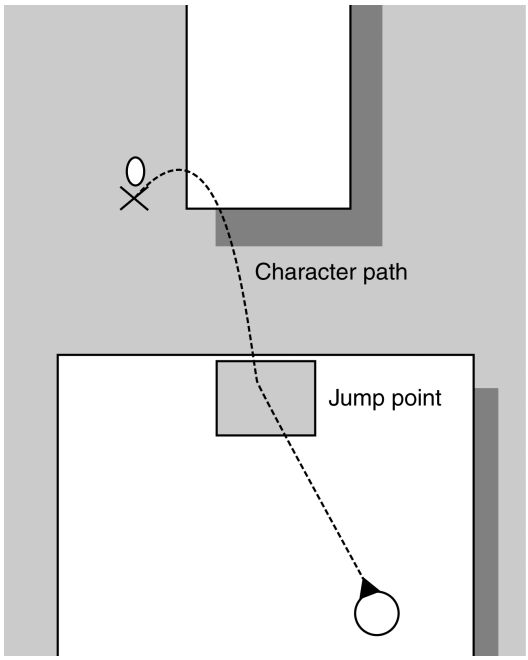

Figure 3.51: A jump to a narrower platform

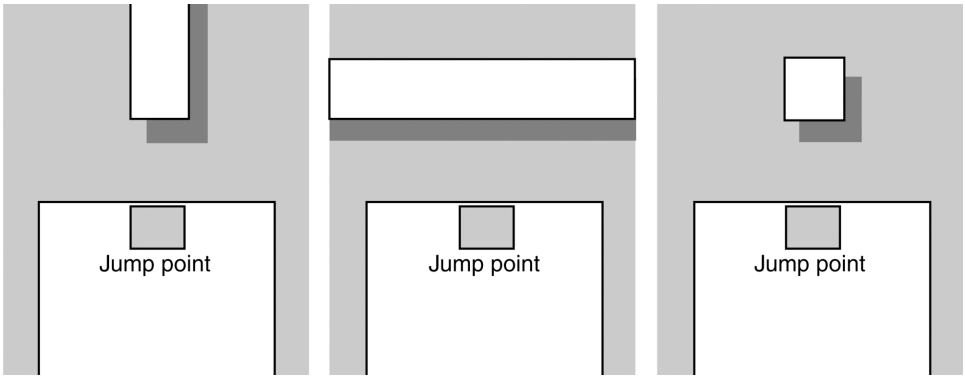

Figure 3.52: Three cases of difficult jump points

ing a jump if it only lands in two feet of water with an easy option to climb out. If the jump crosses a 50-foot drop into boiling lava, then accuracy is more important.

We can incorporate more information into the jump point—data that include the kinds of restrictions on approach velocities and how dangerous it would be to get it wrong. Because they are created by the level designer, such data are prone to error and difficult to tune. Bugs in the velocity information may not surface throughout QA if the AI characters don't happen to attempt the jump in the wrong way.

A common workaround is to limit the placement of jump points to give the AI the best chance of looking intelligent. If there are no risky jumps that the AI knows about, then it is less likely to fail. To avoid this being obvious to the player, some restrictions on the level structure are commonly imposed, reducing the number of risky jumps that the player can make, but AI characters choose not to. This is typical of many aspects of AI development: the capabilities of the AI put natural restrictions on the layout of the game's levels. Or, put another way, the level designers have to avoid exposing weaknesses in the AI.

3.6.2 LANDING PADS

A better alternative is to combine jump points with landing pads. A landing pad is another region of the level, very much like the jump point. Each jump point is paired with a landing pad. We can then simplify the data needed in the jump point. Rather than require the level designer to set up the required velocity, we can leave that up to the character.

When the character determines that it will make a jump, it adds an extra processing step. Using trajectory prediction code similar to that provided in the previous section, the character calculates the velocity required to land exactly on the landing pad when taking off from the jump point. The character can then use this calculation as the basis of its velocity matching algorithm.

This approach is significantly less prone to error. Because the character is calculating the velocity needed, it will not be prone to accuracy errors in setting up the jump point. It also benefits from allowing characters to take into account their own physics when determining how to jump. If characters are heavily laden with weapons, they may not be able to jump up so high. In this case they will need to have a higher velocity to carry themselves over the gap. Calculating the jump trajectory allows them to get the exact approach velocity they need.

The Trajectory Calculation

The trajectory calculation is slightly different to the firing solution we met previously. In the current case we know the start point S, end point E, gravity g, and the y component of velocity v_y. We don't know the time t, or the x and z components of velocity. We therefore have three equations in three unknowns:

$$
\begin{aligned}
E_x &= S_x + v_x t \\
E_y &= S_y + v_y t + \frac{1}{2} g_y t^2 \\
E_z &= S_z + v_z t
\end{aligned}
$$

I have assumed here that gravity is acting in the vertical direction only and that the known jump velocity is also only in the vertical direction. To support other gravity directions, we would need to allow the maximum jump velocity not only to be just in the y-direction but also to have an arbitrary vector. The equations above would then have to be rewritten in terms of both the jump vector to find and the known jump velocity vector. This causes significant problems in the mathematics that are best avoided, especially since the vast majority of cases require y-direction jumps only, exactly as shown here.

I have also assumed that there is no drag during the trajectory. This is the most common situation. Drag is usually non-existent or negligible for these calculations. If you need to include drag for your game, then replace these equations with those given in Section 3.5.4; solving them will be correspondingly more difficult.

We can solve the system of equations to give:

$$
t = \frac{-v_y \pm \sqrt{2g(E_y - S_y) + v_y^2}}{g} \tag{3.7}
$$

and then:

$$
v_x = \frac{E_x - S_x}{t}
$$

and:

$$
v_z = \frac{E_z - S_z}{t}
$$

Equation 3.7 has two solutions. We'd ideally like to achieve the jump in the fastest time possible, so we want to use the smaller of the two values. Unfortunately, this value might give us an impossible launch velocity, so we need to check and use the higher value if necessary.

We can now implement a jumping steering behavior to use a jump point and landing pad. This behavior is given a jump point when it is created and tries to achieve the jump. If the jump is not feasible, it will have no effect, and no acceleration will be requested.

Pseudo-Code

The jumping behavior can be implemented in the following way:

```
class Jump extends VelocityMatch:
    # The jump point to use.
    jumpPoint: JumpPoint

    # Keeps track of whether the jump is possible.
    canAchieve: bool = false

    # The movement capability of the character.
    maxSpeed: float
    maxTakeoffYSpeed: float

    # Retrieve the steering to make this jump.
    function getSteering( ) -> SteeringOutput:
        # Ensure we have a velocity we want to achieve.
        if not target: calculateTarget( )
        if not canAchieve: return null

        # Check if we've hit the jump point. (NB: 'character'
        # is inherited from the VelocityMatch base class)
        if character.position.near(target.position) and
           character.velocity.near(target.velocity):
            # Perform the jump, and return no steering
            # (we'll be airborne, no need to steer).
            scheduleJumpAction( )
            return null

        # Delegate the steering to get top the takeoff location.
        return VelocityMatch.getSteering( )

    # Work out the trajectory calculation.
    function calculateTarget( ):
        target = new Kinematic( )
        target.position = jumpPoint.takeoffLocation

        jumpVector = jumpPoint.landingLocation -
                     jumpPoint.takeoffLocation

        # Calculate the first jump time, and check if we can use it.
        sqrtTerm = sqrt(2 * gravity.y * jumpVector.y +
                        maxTakeoffYSpeed * maxTakeoffYSpeed)
        time: float = (maxTakeoffYSpeed - sqrtTerm) / gravity.y
        checkCanAchieveJumpTime(jumpVector, time)
        if not canAchieve:
            # Otherwise try the other square root.
            time = (maxTakeoffYSpeed + sqrtTerm) / gravity.y
            checkCanAchieveJumpTime(jumpVector, time)
```

```
47
48      # Check whether a jump taking the given time is achievable.
49      function checkJumpTime(jumpVector: Vector, time: float):
50          # Calculate the planar speed.
51          vx = jumpVector.x / time
52          vz = jumpVector.z / time
53          speedSq = vx * vx + vz * vz
54
55          # Check it.
56          if speedSq < maxSpeed * maxSpeed:
57              # We have a valid solution, so store it.
58              target.velocity.x = vx
59              target.velocity.z = vz
60              canAchieve = true
```

Data Structures and Interfaces

This code relies on a simple jump point data structure that has the following form:

```
1  class JumpPoint:
2      takeoffLocation: Vector
3      landingLocation: Vector
```

In addition, I have used the `near` method of a vector to determine if the vectors are roughly similar. This is used to make sure that we start the jump without requiring absolute accuracy from the character. The character is unlikely to ever hit a jump point completely accurately, so this function provides some margin of error. The particular margin for error depends on the game and the velocities involved: faster moving or larger characters require larger margins for error.

Finally, I have used a `scheduleJumpAction` function to force the character into the air. This can schedule an action to a regular action queue (a structure we will look at in depth in Chapter 5), or it can simply add the required vertical velocity directly to the character, sending it upward. The latter approach is fine for testing but makes it difficult to schedule a jump animation at the correct time. As we'll see later in the book, sending the jump through a central action resolution system allows us to simplify animation selection.

Implementation Notes

When implementing this behavior as part of an entire steering system, it is important to make sure it can take complete control of the character. If the steering behavior is combined with others using a blending algorithm, then it will almost certainly fail eventually. A character that is avoiding an enemy at a tangent to the jump will have its trajectory skewed. It either will not

arrive at the jump point (and therefore not take off) or will jump in the wrong direction and plummet.

Performance

The algorithm is O(1) in both time and memory.

Jump Links

Rather than have jump points as a new type of game entity, many developers incorporate jumping into their pathfinding framework. Pathfinding will be discussed at length in Chapter 4, so I don't want to anticipate too much here.

As part of the pathfinding system, we will have to create a network of locations in the game. The connections that link locations have information stored with them (the distance between the locations in particular). We can simply add jumping information to this connection.

A connection between two nodes on either side of a gap is labeled as requiring a jump. At runtime, the link can be treated just like a jump point and landing pad pair, and the algorithm above can be applied to carry out the jump.

3.6.3 HOLE FILLERS

Another approach used by several developers allows characters to choose their own jump points. The level designer fills holes with an invisible object, labeled as a jumpable gap.

The character steers as normal but has a special variation of the obstacle avoidance steering behavior (we'll call it a jump detector). This behavior treats collisions with the jumpable gap object differently from collisions with walls. Rather than trying to avoid the wall, it moves toward it at full speed. At the point of collision (i.e., the last possible moment that the character is on the ledge), it executes a jump action and leaps into the air.

This approach has great flexibility; characters are not limited to a particular set of locations from which they can jump. In a room that has a large chasm running through it, for example, the character can jump across at any point. If it steers toward the chasm, the jump detector will execute the jump across automatically. There is no need for separate jump points on each side of the chasm. The same jumpable gap object works for both sides.

We can easily support one-directional jumps. If one side of the chasm is lower than the other, we could set up the situation shown in Figure 3.53. In this case the character can jump from the high side to the low side, but not the other way around. In fact, we can use very small versions of this collision geometry in a similar way to jump points (label them with a target velocity and they are the 3D version of jump points).

While hole fillers are flexible and convenient, this approach suffers even more from the problem of sensitivity to landing areas. With no target velocity, or notion of where the char-

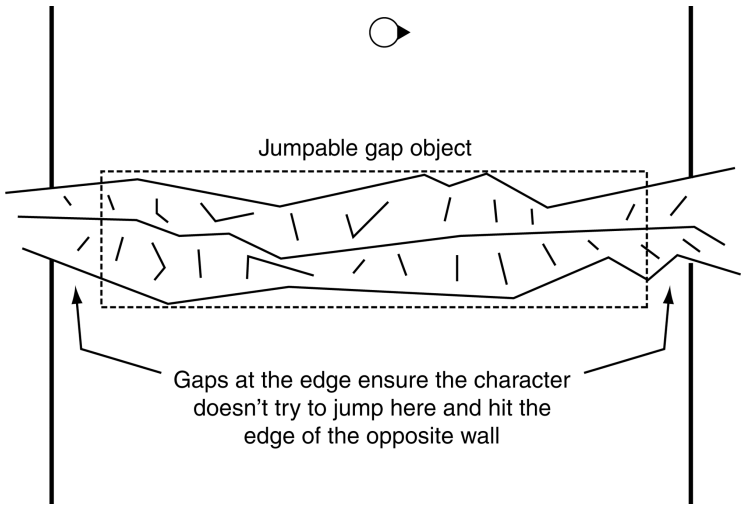

Figure 3.53: A one-direction chasm jump

acter wants to land, it will not be able to sensibly work out how to take off to avoid missing a landing spot. In the chasm example above, the technique is ideal because the landing area is so large, and there is very little possibility of failing the jump.

If you use this approach, then make sure you design levels that don't show the weaknesses in the approach. Aim only to have jumpable gaps that are surrounded by ample take off and landing space.

3.7 COORDINATED MOVEMENT

Games increasingly require groups of characters to move in a coordinated manner. Coordinated motion can occur at two levels. The individuals can make decisions that compliment each other, making their movements appear coordinated. Or they can make a decision as a whole and move in a prescribed, coordinated group.

Tactical decision making will be covered in Chapter 6. This section looks at ways to move groups of characters in a cohesive way, having already made the decision that they should move together. This is usually called *formation motion*.

Formation motion is the movement of a group of characters so that they retain some group organization. At its simplest it can consist of moving in a fixed geometric pattern such as a V or line abreast, but it is not limited to that. Formations can also make use of the environment. Squads of characters can move between cover points using formation steering with only minor modifications, for example. Formation motion is used in team sports games, squad-based

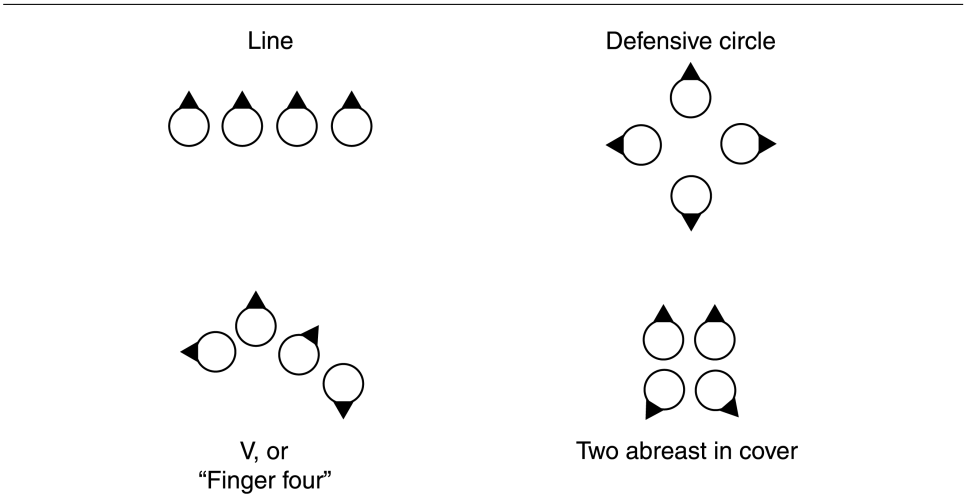

Figure 3.54: A selection of formations

games, real-time strategy games, and an increasing number of first-person shooters, driving games, and action adventures. It is a simple and flexible technique that is much quicker to write and execute and can produce much more stable behavior than collaborative tactical decision making.

3.7.1 FIXED FORMATIONS

The simplest kind of formation movement uses fixed geometric formations. A formation is defined by a set of slots: locations where a character can be positioned. Figure 3.54 shows some common formations used in military-inspired games.

One slot is marked as the leader's slot. All the other slots in the formation are defined relative to this slot. Effectively, it defines the "zero" for position and orientation in the formation.

The character at the leader's location moves through the world like any non-formation character would. It can be controlled by any steering behavior, it may follow a fixed path, or it may have a pipeline steering system blending multiple movement concerns. Whatever the mechanism, it does not take into account the fact that it is positioned in the formation.

The formation pattern is positioned and oriented in the game so that the leader is located in its slot, facing the appropriate direction. As the leader moves, the pattern also moves and turns in the game. In turn, each of the slots in the pattern move and turn in unison.

Each additional slot in the formation can then be filled by an additional character. The position of each character can be determined directly from the formation geometry, without

requiring a kinematic or steering system of its own. Often, the character in the slot has its position and orientation set directly.

If a slot is located at r_s relative to the leader's slot, then the position of the character at that slot will be

$$p_s = p_l + \Omega_l r_s$$

where p_s is the final position of slot s in the game; p_l is the position of the leader character and Ω_l is the orientation of the leader character, in matrix form. In the same way the orientation of the character in the slot will be:

$$\omega_s = \omega_l + \omega_s$$

where ω_s is the orientation of the slot s, relative to the leader's orientation and ω_l is the orientation of the leader.

The movement of the leader character should take into account the fact that it is carrying the other characters with it. The algorithms it uses to move will be no different to a non-formation character, but it should have limits on the speed it can turn (to avoid outlying characters sweeping round at implausible speeds), and any collision or obstacle avoidance behaviors should take into account the size of the whole formation.

In practice, these constraints on the leader's movement make it difficult to use this kind of formation for anything but very simple formation requirements (small squads of troops in a strategy game where you control 10,000 units, for example).

3.7.2 SCALABLE FORMATIONS

In many situations the exact structure of a formation will depend on the number of characters that are participating in it. A defensive circle, for example, will be wider with 20 defenders than with 5. With 100 defenders, it may be possible to structure the formation in several concentric rings. Figure 3.55 illustrates this.

It is common to implement scalable formations without an explicit list of slot positions and orientations. A function can dynamically return the slot locations, given the total number of characters in the formation, for example.

This kind of implicit, scalable formation can be seen very clearly in *Homeworld* [172]. When additional ships are added to a formation, the formation accommodates them, changing its distribution of slots accordingly. Unlike our example so far, Homeworld uses a more complex algorithm for moving the formation around.

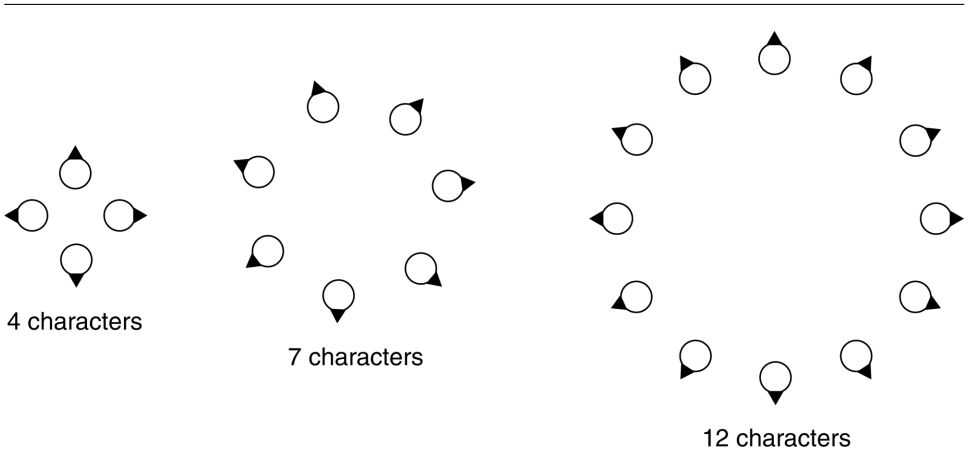

Figure 3.55: A defensive circle formation with different numbers of characters

3.7.3 EMERGENT FORMATIONS

Emergent formations provide a different solution to scalability. Each character has its own steering system using the arrive behavior. The characters select their target based on the position of other characters in the group.

Imagine that we are looking to create a large V formation. We can force each character to choose another target character in front of it and select a steering target behind and to the side, for example. If there is another character already selecting that target, then it selects another. Similarly, if there is another character already targeting a location very near, it will continue looking. Once a target is selected, it will be used for all subsequent frames, updated based on the position and orientation of the target character. If the target becomes impossible to achieve (it passes into a wall, for example), then a new target will be selected.

Overall, this emergent formation will organize itself into a V formation. If there are many members of the formation, the gap between the bars of the V will fill up with smaller V shapes. As Figure 3.56 shows, the overall arrowhead effect is pronounced regardless of the number of characters in the formation. In the figure, the lines connect a character with the character it is following.

There is no overall formation geometry in this approach, and the group does not necessarily have a leader (although it helps if one member of the group isn't trying to position itself relative to any other member). The formation emerges from the individual rules of each character, in exactly the same way as we saw flocking behaviors emerge from the steering behavior of each flock member.

This approach also has the advantage of allowing each character to react individually to obstacles and potential collisions. There is no need to factor in the size of the formation when

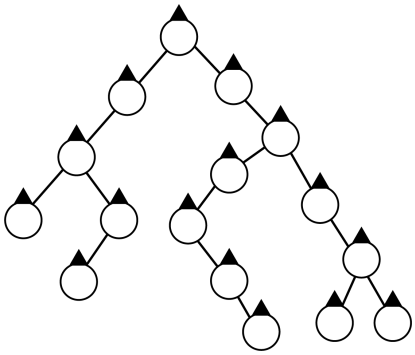

Figure 3.56: Emergent arrowhead formation

considering turning or wall avoidance, because each individual in the formation will act appropriately (as long as it has those avoidance behaviors as part of its steering system).

While this method is simple and effective, it can be difficult to set up rules to get just the right shape. In the V example above, a number of characters often end up jostling for position in the center of the V. With more unfortunate choices in each character's target selection, the same rule can give a formation consisting of a single long diagonal line with no sign of the characteristic V shape.

Debugging emergent formations, like any kind of emergent behavior, can be a challenge. The overall effect is often one of controlled disorder, rather than formation motion. For military groups, this characteristic disorder makes emergent formations of little practical use.

3.7.4 TWO-LEVEL FORMATION STEERING

We can combine strict geometric formations with the flexibility of an emergent approach using a two-level steering system. We use a geometric formation, defined as a fixed pattern of slots, just as before. Initially, we will assume we have a leader character, although we will remove this requirement later.

Rather than directly placing each character it its slot, it follows the emergent approach by using the slot at a target location for an arrive behavior. Characters can have their own collision avoidance behaviors and any other compound steering required.

This is two-level steering because there are two steering systems in sequence: first the leader steers the formation pattern, and then each character in the formation steers to stay in the pattern. As long as the leader does not move at maximum velocity, each character will have some flexibility to stay in its slot while taking account of its environment.

Figure 3.57 shows a number of agents moving in V formation through the woods. The

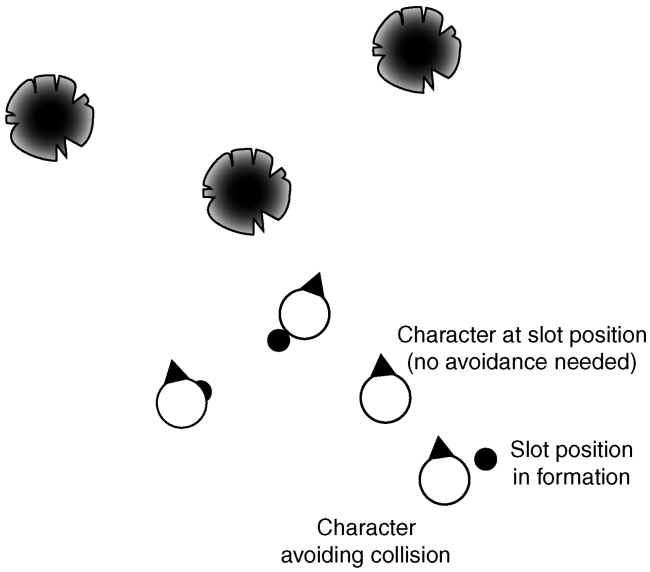

Figure 3.57: Two-level formation motion in a V

characteristic V shape is visible, but each character has moved slightly from its slot position to avoid bumping into trees.

The slot that a character is trying to reach may be briefly impossible to achieve, but its steering algorithm ensures that it still behaves sensibly.

Removing the Leader

In the example above, if the leader needs to move sideways to avoid a tree, then all the slots in the formation will also lurch sideways and every other character will lurch sideways to stay with the slot. This can look odd because the leader's actions are mimicked by the other characters, although they are largely free to cope with obstacles in their own way.

We can remove the responsibility for guiding the formation from the leader and have all the characters react in the same way to their slots. The formation is moved around by an invisible leader: a separate steering system that is controlling the whole formation, but none of the individuals. This is the second level of the two-level formation.

Because this new leader is invisible, it does not need to worry about small obstacles, bumping into other characters, or small terrain features. The invisible leader will still have a fixed location in the game, and that location will be used to lay out the formation pattern and determine the slot locations for all the proper characters. The location of the leader's slot in the

pattern will not correspond to any character, however. Because it is not acting like a slot, we call this the pattern's *anchor point*.

Having a separate steering system for the formation typically simplifies implementation. We no longer have different characters with different roles, and there is no need to worry about making one character take over as leader if another one dies.

The steering for the anchor point is often simplified. Outdoors, we might only need to use a single high-level arrive behavior, for example, or maybe a path follower. In indoor environments the steering will still need to take account of large-scale obstacles, such as walls. A formation that passes straight through into a wall will strand all its characters, making them unable to follow their slots.

Moderating the Formation Movement

So far information has flowed in only one direction: from the formation to the characters within it.

When we have a two-level steering system, this causes problems. The formation could be steering ahead, oblivious to the fact that its characters are having problems keeping up. When the formation was being led by a character, this was less of a problem, because difficulties faced by the other characters in the formation were likely to also be faced by the leader.

When we steer the anchor point directly, it is usually allowed to disregard small-scale obstacles and other characters. The characters in the formations may take considerably longer to move than expected because they are having to navigate these obstacles. This can lead to the formation and its characters getting a long way out of sync.

One solution is to slow the formation down. A good rule of thumb is to make the maximum speed of the formation around half that of the characters. In fairly complex environments, however, the slow down required is unpredictable, and it is better not to burden the whole game with slow formation motion for the sake of a few occasions when a faster speed would be problematic.

A better solution is to moderate the movement of the formation based on the current positions of the characters in its slots: in effect to keep the anchor point on a leash. If the characters in the slots are having trouble reaching their targets, then the formation as a whole should be held back to give them a chance to catch up.

This can be simply achieved by resetting the kinematic of the anchor point at each frame. Its position, orientation, velocity, and rotation are all set to the average of those properties for the characters in its slots. If the anchor point's steering system gets to run first, it will move forward a little, moving the slots forward and forcing the characters to move also. After the slot characters are moved, the anchor point is reined back so that it doesn't move too far ahead.

Because the position is reset at every frame, the target slot position will only be a little way ahead of the character when it comes to steer toward it. Using the arrive behavior will mean that each character is fairly nonchalant about moving such a small distance, and the speed for the slot characters will decrease. This, in turn, will mean that the speed of the formation

decreases (because it is being calculated as the average of the movement speeds for the slot characters). On the following frame the formation's velocity will be even less. Over a handful of frames it will slow to a halt.

An offset is generally used to move the anchor point a small distance ahead of the center of mass. The simplest solution is to move it a fixed distance forward, as given by the velocity of the formation:

$$p_{\text{anchor}} = p_c + k_{\text{offset}} v_c \tag{3.8}$$

where p_c is the position, and v_c is the velocity of the center of mass. It is also necessary to set a very high maximum acceleration and maximum velocity for the formation's steering. The formation will not actually achieve this acceleration or velocity because it is being held back by the actual movement of its characters.

Drift

Moderating the formation motion requires that the anchor point of the formation always be at the center of mass of its slots (i.e., its average position). Otherwise, if the formation is supposed to be stationary, the anchor point will be reset to the average point, which will not be where it was in the last frame. The slots will all be updated based on the new anchor point and will again move the anchor point, causing the whole formation to drift across the level.

It is relatively easy, however, to recalculate the offsets of each slot based on a calculation of the center of mass of a formation. The center of mass of the slots is given by:

$$p_c = \frac{1}{n} \sum_{i=1..n} \begin{cases} p_{s_i} & \text{if slot } i \text{ is occupied} \\ 0 & \text{otherwise} \end{cases}$$

where p_{s_i} is the position of slot i. Changing from the old anchor point to the new involves changing each slot coordinate according to:

$$p'_{s_i} = p_{s_i} - p_c \tag{3.9}$$

For efficiency, this should be done once and the new slot coordinates stored, rather than being repeated every frame. It may not be possible, however, to perform the calculation offline. Different combinations of slots may be occupied at different times. When a character in a slot gets killed, for example, the slot coordinates will need to be recalculated because the center of mass will have changed.

Drift also occurs when the anchor point is not at the average orientation of the occupied slots in the pattern. In this case, rather than drifting across the level, the formation will appear to spin on the spot. We can again use an offset for all the orientations based on the average orientation of the occupied slots:

$$\vec{\omega_c} = \frac{\vec{v_c}}{|\vec{v_c}|}$$

where

$$\vec{v_c} = \frac{1}{n} \sum_{i=1..n} \begin{cases} \vec{\omega_{s_i}} & \text{if slot } i \text{ is occupied} \\ 0 & \text{otherwise} \end{cases}$$

and $\vec{\omega_{s_i}}$ is the orientation of slot i. The average orientation is given in vector form, and can be converted back into an angle ω_c, in the range $(-\pi, \pi)$. As before, changing from the old anchor point to the new one involves changing each slot orientation according to:

$$\omega'_{s_i} = \omega_{s_i} - \omega_c$$

This should also be done as infrequently as possible, being cached internally until the set of occupied slots changes.

3.7.5 IMPLEMENTATION

We can now implement the two-level formation system. The system consists of a formation manager that processes a formation pattern and generates targets for the characters occupying its slots.

```
class FormationManager:
    # The assignment characters to slots.
    class SlotAssignment:
        character: Character
        slotNumber: int
    slotAssignments: SlotAssignment[]

    # A Static (i.e., position and orientation) representing the
    # drift offset for the currently filled slots.
    driftOffset: Static

    # The formation pattern.
    pattern: FormationPattern

    # Update the assignment of characters to slots.
    function updateSlotAssignments():
        # A trivial assignment algorithm: we simply go through
        # each character and assign sequential slot numbers.
        for i in 0..slotAssignments.length():
            slotAssignments[i].slotNumber = i

        # Update the drift offset.
        driftOffset = pattern.getDriftOffset(slotAssignments)

    # Add a new character. Return false if no slots are available.
    function addCharacter(character: Character) -> bool:
```

```
        # Check if the pattern supports more slots.
        occupiedSlots = slotAssignments.length()
        if pattern.supportsSlots(occupiedSlots + 1):
            # Add a new slot assignment.
            slotAssignment = new SlotAssignment()
            slotAssignment.character = character
            slotAssignments.append(slotAssignment)
            updateSlotAssignments()
            return true
        else:
            # Otherwise we've failed to add the character.
            return false

    # Remove a character from its slot.
    function removeCharacter(character: Character):
        slot = charactersInSlots.findIndexOfCharacter(character)
        slotAssignments.removeAt(slot)
        updateSlotAssignments()

    # Send new target locations to each character.
    function updateSlots():
        # Find the anchor point.
        anchor: Static = getAnchorPoint()
        orientationMatrix: Matrix = anchor.orientation.asMatrix()

        # Go through each character in turn.
        for i in 0..slotAssignments.length():
            slotNumber: int = slotAssignments[i].slotNumber
            slot: Static = pattern.getSlotLocation(slotNumber)

            # Transform by the anchor point position and orientation.
            location = new Static()
            location.position = anchor.position +
                                orientationMatrix * slot.position
            location.orientation = anchor.orientation +
                                   slot.orientation

            # And add the drift component.
            location.position -= driftOffset.position
            location.orientation -= driftOffset.orientation

            # Send the static to the character.
            slotAssignments[i].character.setTarget(location)

    # The characteristic point of this formation (see below).
    function getAnchorPoint() -> Static
```

For simplicity, in the code I've assumed that we can look up a slot in the `slotAssignments` list by its character using a `findIndexFromCharacter` method. Similarly, I've used a `removeAt` method of the same list to remove an element at a given index.

Data Structures and Interfaces

The formation manager relies on access to the current anchor point of the formation through the `getAnchorPoint` function. This can be the location and orientation of a leader character, a modified center of mass of the characters in the formation, or an invisible but steered anchor point for a two-level steering system.

One implementation of `getAnchorPoint` could be to return the current center of mass of the characters in the formation.

The formation pattern class generates the slot offsets for a pattern, relative to its anchor point. It does this after being asked for its drift offset, given a set of assignments. In calculating the drift offset, the pattern works out which slots are needed. If the formation is scalable and returns different slot locations depending on the number of slots occupied, it can use the slot assignments passed into the `getDriftOffset` function to work out how many slots are used and therefore what positions each slot should occupy.

Each particular pattern (such as a V, wedge, circle) needs its own instance of a class that matches the formation pattern interface:

```
class FormationPattern:
    # The drift offset when characters are in the given set of slots.
    function getDriftOffset(slotAssignments) -> Static

    # Calculate and return the location of the given slot index.
    function getSlotLocation(slotNumber: int) -> Static

    # True if the pattern can support the given number of slots.
    function supportsSlots(slotCount) -> bool
```

In the manager class, I've also assumed that the characters provided to the formation manager can have their slot target set. The interface is simple:

```
class Character:
    # Set the steering target of the character.
    function setTarget(static: Static)
```

Implementation Caveats

In reality, the implementation of this interface will depend on the rest of the character data we need to keep track of for a particular game. Depending on how the data are arranged in

your game engine, you may need to adjust the formation manager code so that it accesses your character data directly.

Performance

The target update algorithm is O(n) in time, where n is the number of occupied slots in the formation. It is O(1) in memory, excluding the resulting data structure into which the assignments are written, which is O(n) in memory, but is part of the overall class and exists before and after the class's algorithms run.

Adding or removing a character consists of two parts in the pseudo-code above: (1) the actual addition or removal of the character from the slot assignments list, and (2) the updating of the slot assignments on the resulting list of characters.

Adding a character is an O(1) process in both time and memory. Removing a character involves finding if the character is present in the slot assignments list. Using a suitable hashing representation, this can be O($\log n$) in time and O(1) in memory.

As given above, the assignment algorithm is O(n) in time and O(1) in memory (again excluding the assignment data structure). Typically, assignment algorithms will be more sophisticated and have worse performance than O(n), as we will see later in this chapter.

In the (somewhat unlikely) event that this kind of assignment algorithm is suitable, we can optimize it by having the assignment only reassign slots to characters that need to change (adding a new character, for example, may not require the other characters to change their slot numbers). I have deliberately not tried to optimize this algorithm, because we will see that it has serious behavioral problems that must be resolved with more complex assignment techniques.

Sample Formation Pattern

To make things more concrete, let's consider a usable formation pattern. The defensive circle posts characters around the circumference of a circle, so their backs are to the center. The circle can consist of any number of characters (although a huge number might look silly, we will not put any fixed limit).

The defensive circle formation class might look something like the following:

```
class DefensiveCirclePattern:
    # The radius of one character, this is needed to determine how
    # close we can pack a given number of characters around a circle.
    characterRadius: float

    # Calculate the number of slots in the pattern from the assignment
    # data. This is not part of the formation pattern interface.
    function calculateNumberOfSlots(assignments) -> int:
        # Find the number of filled slots: it will be the
        # highest slot number in the assignments.
```

```
        filledSlots: int = 0
        for assignment in assignments:
            if assignment.slotNumber >= maxSlotNumber:
                filledSlots = assignment.slotNumber

        # Add one to go from the index of the highest slot to
        # the number of slots needed.
        return filledSlots + 1

    # Calculate the drift offset (average position) of the pattern.
    function getDriftOffset(assignments) -> Static:
        # Add each assignment's contribution to the result.
        result = new Static()
        for assignment in assignments:
            location = getSlotLocation(assignment.slotNumber)
            result.position += location.position
            result.orientation += location.orientation

        # Divide through to get the drift offset.
        numberOfAssignments = assignments.length()
        result.position /= numberOfAssignments
        result.orientation /= numberOfAssignments
        return result

    # Calculate the position of a slot.
    function getSlotLocation(slotNumber: int) -> Static:
        # Place the slots around a circle based on their slot number.
        angleAroundCircle = slotNumber / numberOfSlots * pi * 2

        # The radius depends on the radius of the character, and the
        # number of characters in the circle: we want there to be no
        # gap between characters' shoulders.
        radius = characterRadius / sin(pi / numberOfSlots)

        result = new Static()
        result.position.x = radius * cos(angleAroundCircle)
        result.position.z = radius * sin(angleAroundCircle)

        # Characters face out.
        result.orientation = angleAroundCircle

        return result

    # We support any number of slots.
    function supportsSlots(slotCount) -> bool:
        return true
```

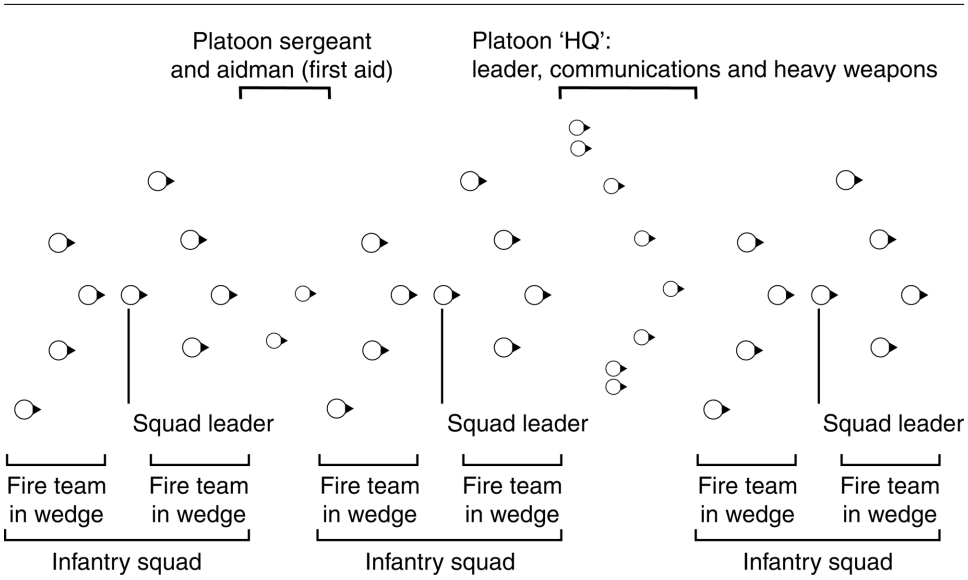

Figure 3.58: Nesting formations to greater depth

If we know we are using the assignment algorithm given in the previous pseudo-code, then we know that the number of slots will be the same as the number of assignments (since characters are assigned to sequential slots). In this case the `calculateNumberOfSlots` method can be simplified:

```
function calculateNumberOfSlots(assignments) -> int:
    return assignments.length()
```

In general, with more useful assignment algorithms, this may not be the case, so the long form above is usable in all cases, at the penalty of some decrease in performance.

3.7.6 EXTENDING TO MORE THAN TWO LEVELS

The two-level steering system can be extended to more levels, giving the ability to create formations of formations. This is becomingly increasingly important in military strategy games with lots of units; real armies are organized in this way.

The framework above can be simply extended to support any depth of formation. Each formation has its own steering anchor point, either corresponding to a leader character or representing the formation in an abstract way. The steering for this anchor point can be managed in turn by another formation. The anchor point is trying to stay in a slot position of a higher level formation.

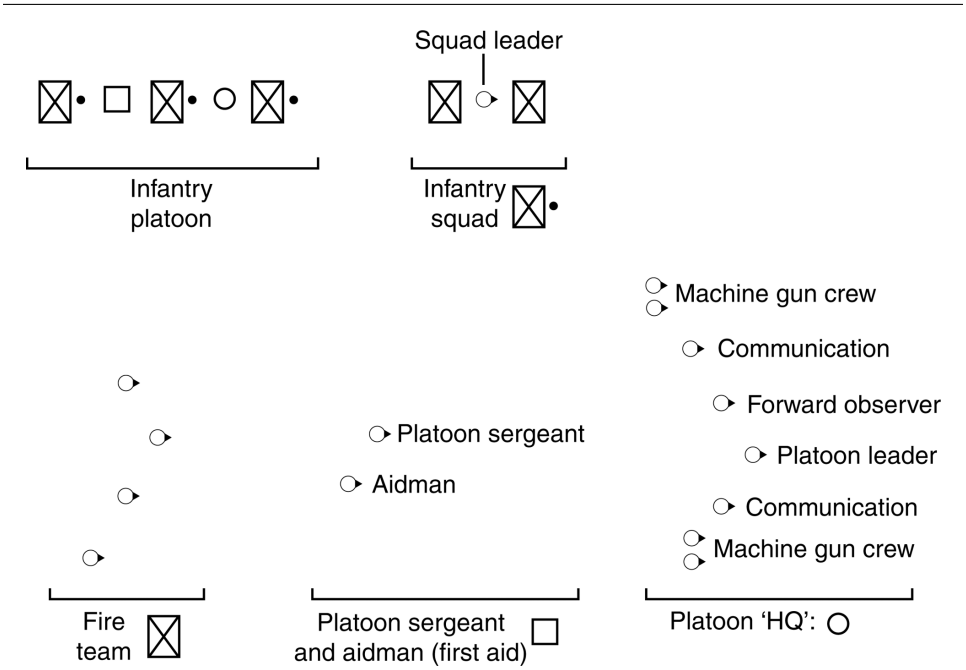

Figure 3.59: Nesting formations shown individually

Figure 3.58 shows an example adapted from the U.S. infantry soldiers training manual [71]. The infantry rifle fire team has its characteristic finger-tip formation (called the "wedge" in army-speak). These finger-tip formations are then combined into the formation of an entire infantry squad. In turn, this squad formation is used in the highest level formation: the column movement formation for a rifle platoon.

Figure 3.59 shows each formation on its own to illustrate how the overall structure of Figure 3.58 is constructed.[2] Notice that in the squad formation there are three slots, one of which is occupied by an individual character. The same thing happens at an entire platoon level: additional individuals occupy slots in the formation. As long as both characters and formations expose the same interface, the formation system can cope with putting either an individual or a whole sub-formation into a single slot.

The squad and platoon formations in the example show a weakness in our current implementation. The squad formation has three slots. There is nothing to stop the squad leader's slot from being occupied by a rifle team, and there is nothing to stop a formation from having

2. The format of the diagram uses military mapping symbols common to all NATO countries. A full guide on military symbology can be found in Kourkolis [31], but it is not necessary to understand any details for our purposes in this book.

two leaders and only one rifle team. To avoid these situations we need to add the concept of slot roles.

3.7.7 SLOT ROLES AND BETTER ASSIGNMENT

So far I have assumed that any character can occupy each slot. While this is often the case, some formations are explicitly designed to give each character a different role. A rifle fire team in a military simulation game, for example, will have a rifleman, grenadier, machine gunner, and squad leader in very specific locations. In a real-time strategy game, it is often advisable to keep the heavy artillery in the center of a defensive formation, while using agile infantry troops in the vanguard.

Slots in a formation can have roles so that only certain characters can fill certain slots. When a formation is assigned to a group of characters (often, this is done by the player), the characters need to be assigned to their most appropriate slots. Whether using slot roles or not, this should not be a haphazard process, with lots of characters scrabbling over each other to reach the formation.

Assigning characters to slots in a formation is not difficult or error prone if we don't use slot roles. With roles it can become a complex problem. In game applications, a simplification can be used that gives good enough performance.

Hard and Soft Roles

Imagine a formation of characters in a fantasy RPG game. As they explore a dungeon, the party needs to be ready for action. Magicians and missile weapon users should be in the middle of the formation, surrounded by characters who fight hand to hand.

We can support this by creating a formation with roles. We have three roles: magicians (we'll assume that they do not need a direct line of sight to their enemy), missile weapon users (including magicians with fireballs and spells that do follow a trajectory), and melee (hand to hand) weapon users. Let's call these roles "melee," "missile," and "magic" for short.

Similarly, each character has one or more roles that it can fulfill. An elf might be able to fight with a bow or sword, while a dwarf may rely solely on its axe. Characters are only allowed to fill a slot if they can fulfill the role associated with that slot. This is known as a hard role.

Figure 3.60 shows what happens when a party is assigned to the formation. We have four kinds of character: fighters (F) fill melee slots, elves (E) fill either melee or missile slots, archers (A) fill missile slots, and mages (M) fill magic slots. The first party maps nicely onto the formation, but the second party, consisting of all melee combatants, does not.

We could solve this problem by having many different formations for different compositions of the party. In fact, this would be the optimal solution, since a party of sword-wielding thugs will move differently to one consisting predominantly of highly trained archers. Unfortunately, it requires lots of different formations to be designed. If the player can switch formation, this could multiply up to several hundred different designs.

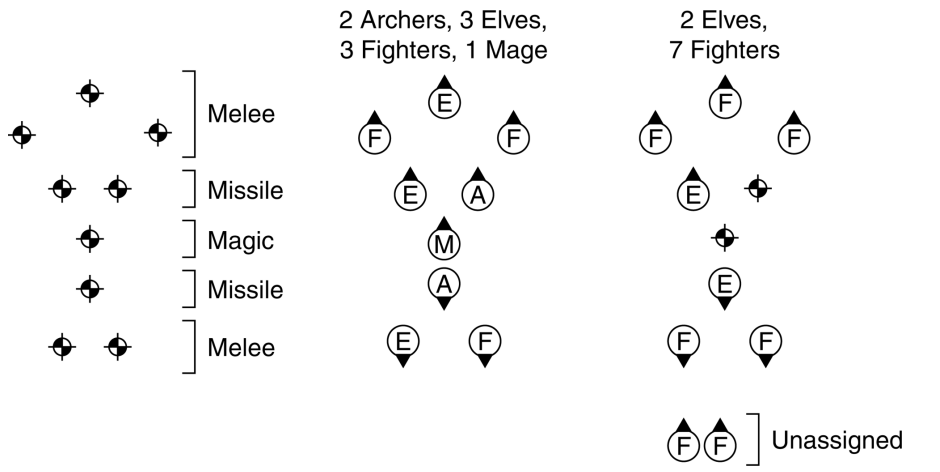

Figure 3.60: An RPG formation, and two examples of the formation filled

On the other hand, we could use the same logic that gave us scalable formations: we feed in the number of characters in each role, and we write code to generate the optimum formation for those characters. This would give us impressive results, again, but at the cost of more complex code. Most developers would ideally want to move as much content out of code as possible, ideally using separate tools to structure formation patterns and define roles.

A simpler compromise approach uses soft roles: roles that can be broken. Rather than a character having a list of roles it can fulfill, it has a set of values representing how difficult it would find it to fulfill every role. In our example, the elf would have low values for both melee and missile roles, but would have a high value for occupying the magic role. Similarly, the fighter would have high values in both missile and magic roles, but would have a very low value for the melee role.

The value is known as the *slot cost*. To make a slot impossible for a character to fill, its slot cost should be infinite. Normally, this is just a very large value. The algorithm below works better if the values aren't near to the upper limit of the data type (such as `FLT_MAX` in C) because several costs will be added. To make a slot ideal for a character, its slot cost should be zero. We can have different levels of unsuitable assignment for one character. Our mage might have a very high slot cost for occupying a melee role but a slightly lower cost for missile slots.

We would like to assign characters to slots in such a way that the total cost is minimized. If there are no ideal slots left for a character, then it can still be placed in a non-suitable slot. The total cost will be higher, but at least characters won't be left stranded with nowhere to go. In our example, the slot costs are given for each role below:

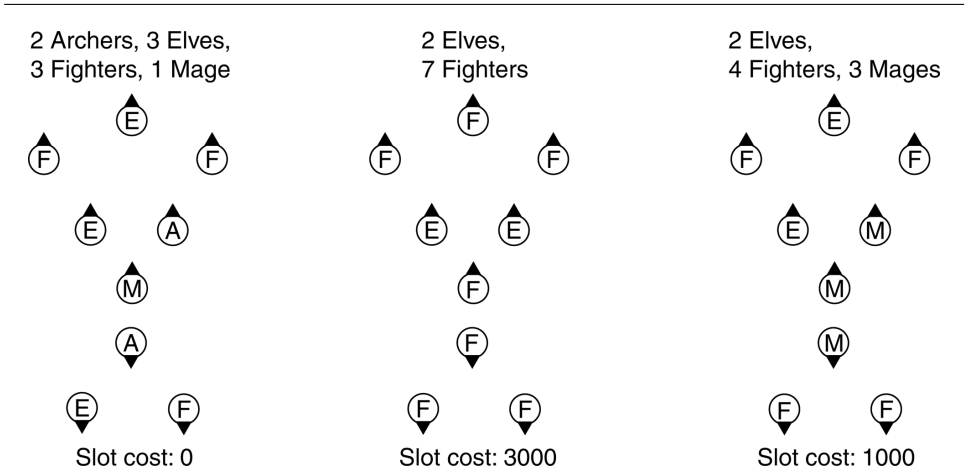

Figure 3.61: Different total slot costs for a party

	Magic	Missile	Melee
Archer	1000	0	1500
Elf	1000	0	0
Fighter	2000	1000	0
Mage	0	500	2000

Figure 3.61 shows that a range of different parties can now be assigned to our formation. These flexible slot costs are called soft roles. They act just like hard roles when the formation can be sensibly filled but don't fail when the wrong characters are available.

3.7.8 SLOT ASSIGNMENT

We have grazed along the topic of slot assignment several times in this section, but have not looked at the algorithm.

Slot assignment needs to happen relatively rarely in a game. Most of the time a group of characters will simply be following their slots around. Assignment usually occurs when a group of previously disorganized characters are assigned to a formation. We will see that it also occurs when characters spontaneously change slots in tactical motion.

For large numbers of characters and slots, the assignment can be done in many different ways. We could simply check each possible assignment and use the one with the lowest slot cost. Unfortunately, the number of assignments we need to check very quickly gets huge. The number of possible assignments of k characters to n slots is given by the permutations formula:

$$ {}_nP_k \equiv \frac{n!}{(n-k)!} $$

For a formation of 20 slots and 20 characters, this gives nearly 2500 trillion different possible assignments. Clearly, no matter how infrequently we need to do it, we can't check every possible assignment. And a highly efficient algorithm won't help us here. The assignment problem is an example of a non-polynomial time complete (NP-complete) problem; it cannot be properly solved in a reasonable amount of time by any algorithm.

Instead, we simplify the problem by using a heuristic. We won't be guaranteed to get the best assignment, but we will usually get a decent assignment very quickly. The heuristic assumes that a character will end up in a slot best suited to it. We can therefore look at each character in turn and assign it to a slot with the lowest slot cost.

We run the risk of leaving a character until last and having nowhere sensible to put it. We can improve the performance by considering highly constrained characters first and flexible characters last. The characters are given an ease of assignment value which reflects how difficult it is to find slots for them.

The ease of assignment value is given by:

$$ \sum_{i=1..n} \begin{cases} \frac{1}{1+c_i} & \text{if } c_i < k \\ 0 & \text{otherwise} \end{cases} $$

where c_i is the cost of occupying slot i, n is the number of possible slots, and k is a slot-cost limit, beyond which a slot is too expensive to consider occupying.

Characters that can only occupy a few slots will have lots of high slot costs and therefore a low ease rating. Notice that we are not adding up the costs for each role, but for each actual slot. Our dwarf may only be able to occupy melee slots, but if there are twice the number of melee slots than other types, it will still be relatively flexible. Similarly, a magician that can fulfill both magic and missile roles will be inflexible if there is only one of each to choose from in a formation of ten slots.

The list of characters is sorted according to their ease of assignment values, and the most awkward characters are assigned first. This approach works in the vast majority of cases and is the standard approach for formation assignment.

Generalized Slot Costs

Slot costs do not necessarily have to depend only on the character and the slot roles. They can be generalized to include any difficulty a character might have in taking up a slot.

If a formation is spread out, for example, a character may choose a slot that is close instead of a more distant slot. Similarly, a light infantry unit may be willing to move farther to get into position than a heavy tank. This is not a major issue when the formations will be used for motion, but it can be significant in defensive formations. This is the reason we used a slot cost, rather than a slot score (i.e., high is bad and low is good, rather than the other way around). Distance can be directly used as a slot cost.

There may be other trade-offs in taking up a formation position. There may be a number of defensive slots positioned at cover points around the room. Characters should take up positions in order of the cover they provide. Partial cover should only be occupied if no better slot is available.

Whatever the source of variation in slot costs, the assignment algorithm will still operate normally. In a real implementation, I would generalize the slot cost mechanism to be a method call; the assignment code then asks a character how costly it will be to occupy a particular slot.

Implementation

We can now implement the assignment algorithm using generalized slot costs. The `calculateAssignment` method is part of the formation manager class, as before.

```
class FormationManager

    # ... other content as before ...

    function updateSlotAssignments():
        # A slot and its corresponding cost.
        class CostAndSlot:
            cost: float
            slot: int

        # A character's ease of assignment and its list of slots.
        class CharacterAndSlots:
            character: Character
            assignmentEase: float
            costAndSlots: CostAndSlot[]

        characterData: CharacterAndSlots[]

        # Compile the character data.
        for assignment in slotAssignments:
            datum = new CharacterAndSlots()
            datum.character = assignment.character

            # Add each valid slot to it.
            for slot in 0..pattern.numberOfSlots:
                cost: float = pattern.getSlotCost(assignment.character)
                if cost >= LIMIT: continue

                slotDatum = new CostAndSlot()
                slotDatum.slot = slot
                slotDatum.cost = cost
                datum.costAndSlots += slotDatum

```

```
                # Add this slot to the character's ease of assignment.
                datum.assignmentEase += 1 / (1 + cost)

            datum.costAndSlots.sortByCost()
            characterData += datum

        # Arrange characters in order of ease of assignment,
        # with the least easy first.
        characterData.sortByAssignmentEase()

        # Keep track of which slots we have filled. Initially all
        # values in this array should be false.
        filledSlots = new bool[pattern.numberOfSlots]

        # Make assignments.
        slotAssignments = []
        for characterDatum in characterData:
            # Choose the first slot in the list that is still open.
            for slot in characterDatum.costAndSlots:
                if not filledSlots[slot]:
                    assignment = new SlotAssignment()
                    assignment.character = characterDatum.character
                    assignment.slotNumber = slot
                    slotAssignments.append(assignment)

                    # Reserve the slot.
                    filledSlots[slot] = true

                    # Go to the next character.
                    break continue

            # If we reach here, it is because a character has no
            # valid assignment. Some sensible action should be
            # taken, such as reporting to the player.
            throw new Error()
```

The `break continue` statement indicates that the innermost loop should be left and the surrounding loop should be restarted with the next element. In some languages this is not an easy control flow to achieve. In C/C++ it can be done by labeling the outermost loop and using a named `continue` statement (which will continue the named loop, automatically breaking out of any enclosing loops), but unfortunately this construct was not included in C#. See the reference information for your language, or search Stack Overflow, to see how to achieve the same effect.

Data Structures and Interfaces

In this code we have hidden a lot of complexity in data structures. There are two lists, `characterData` and `costAndSlots`, within the `CharacterAndSlots` structure that are both sorted.

In the first case, the character data are sorted by the ease of assignment rating, using the `sortByAssignmentEase` method. This can be implemented as any sort, or alternatively the method can be rewritten to sort as it goes, which may be faster if the character data list is implemented as a linked list, where data can be very quickly inserted. If the list is implemented as an array (which is normally faster), then it is better to leave the sort till last and use a fast in-place sorting algorithm such as quicksort.

In the second case, the character data is sorted by slot cost using the `sortByCost` method. Again, this can be implemented to sort as the list is compiled if the underlying data structure supports fast element inserts.

Performance

The performance of the algorithm is O(kn) in memory, where k is the number of characters and n is the number of slots. It is O($ka \log a$) in time, where a is the average number of slots that can be occupied by any given character. This is normally a lower value than the total number of slots but grows as the number of slots grows. If this is not the case, if the number of valid slots for a character is not proportional to the number of slots, then the performance of the algorithm is also O(kn) in time.

In either case, this is significantly faster than an O($_nP_k$) process.

Often, the problem with this algorithm is one of memory rather than speed. There are ways to get the same algorithmic effect with less storage, if necessary, but at a corresponding increase in execution time.

Regardless of the implementation, this algorithm is often not fast enough to be used regularly for large groups (such as the armies in a RTS game). Because assignment happens rarely (when the user selects a new pattern, for example, or adds a unit to a formation), it can be split over several frames. The player is unlikely to notice a delay of a few frames before the characters begin to assemble into a formation.

3.7.9 DYNAMIC SLOTS AND PLAYS

So far we have assumed that the slots in a formation pattern are fixed relative to the anchor point. A formation is a fixed 2D pattern that can move around the game level. The framework we've developed so far can be extended to support dynamic formations that change shape over time.

Slots in a pattern can be dynamic, moving relative to the anchor point of the formation.

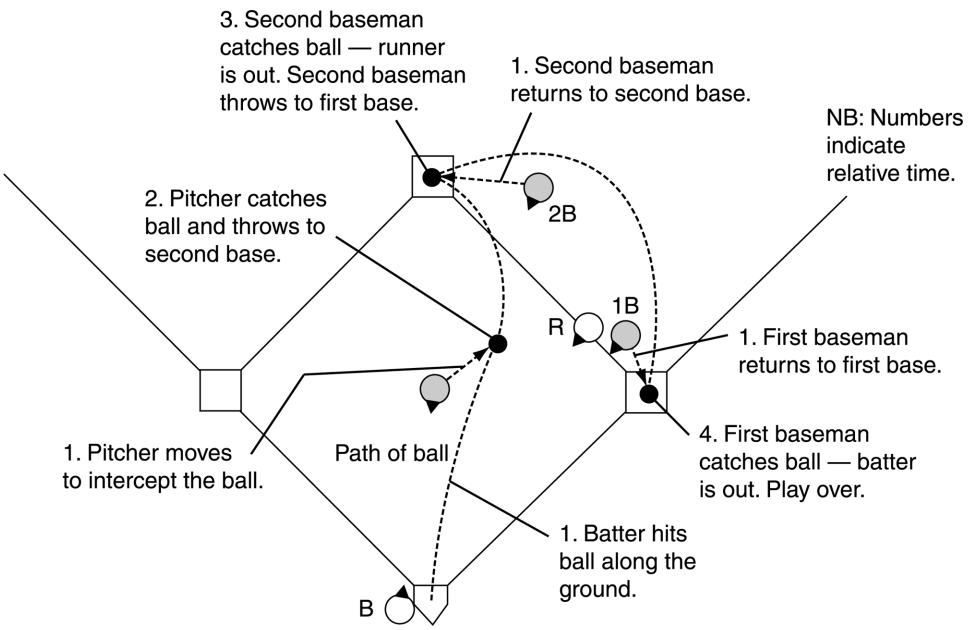

Figure 3.62: A 1-4-3 baseball double play

This is useful for introducing a degree of movement when the formation itself isn't moving, for implementing set plays in some sports games, and for using as the basis of tactical movement.

In an example from baseball, Figure 3.62 shows how fielders move to carry out one of several routine double plays.

This can be implemented as a formation. Each fielder has a fixed slot depending on the position they play. Initially, they are in a fixed pattern formation and are in their normal fielding positions (actually, there may be many of these fixed formations depending on the strategy of the defense). When the AI detects that the double play is on (in this case, there is a runner on first base and the ball can be intercepted by the pitcher but not before it hits the ground), it sets the formation pattern to a dynamic double play pattern. The slots move along the paths shown, bringing the fielders in place to throw out both batters.

In some cases, the slots don't need to move along a path; they can simply jump to their new locations and have the characters use their arrive behaviors to move there. In more complex plays, however, the route taken is not direct, and characters weave their way to their destination.

To support dynamic formations, an element of time needs to be introduced. We can simply extend our pattern interface to take a time value. This will be the time elapsed since the formation began. The pattern interface now looks like the following:

```
class FormationPattern:

    # ... other elements as before ...

    # Get the location of the given slot index at a given time.
    function getSlotLocation(slotNumber: int, time: float) -> Static
```

Unfortunately, this can cause problems with drift, since the formation will have its slots changing position over time. We could extend the system to recalculate the drift offset in each frame to make sure it is accurate. Many games that use dynamic slots and set plays do not use two-level steering, however. For example, the movement of slots in a baseball game is fixed with respect to the field, and in a football game, the plays are often fixed with respect to the line of scrimmage, possibly only scaled depending on how far from the goal-line that is. In this case, there is no need for two-level steering (the anchor point of the formation is fixed), and drift is not an issue, since it can be removed from the implementation.

Many sports titles use techniques similar to formation motion to manage the coordinated movement of players on the field. Some care does need to be taken to ensure that the players don't merrily follow their formation oblivious to what's actually happening on the field.

There is nothing to say that the moving slot positions have to be completely pre-defined. The slot movement can be determined dynamically by a coordinating AI routine. At the extreme, this gives complete flexibility to move players anywhere in response to the tactical situation in the game. But that simply shifts the responsibility for sensible movement onto a different bit of code and begs the question of how should that be implemented.

In practical use some intermediate solution is sensible. Figure 3.63 shows a set soccer play for a corner kick, where only three of the players have fixed play motions.

The movement of the remaining offensive players will be calculated in response to the movement of the defending team, while the key set-play players will be relatively fixed, so the player taking the corner knows where to place the ball. The player taking the corner may wait until just before they kick to determine which of the three potential scorers they will cross to. This again will be in response to the actions of the defense.

The decision can be made by any of the techniques in the decision making chapter (Chapter 5). We could, for example, look at the opposing players in each of A, B, and C's shot cone and pass to the character with the largest free angle to aim for.

3.7.10 TACTICAL MOVEMENT

An important application of formations is tactical squad-based movement.

When they are not confident of the security of the surrounding area, a military squad will move in turn, while other members of the squad provide a lookout and rapid return-of-fire if an enemy should be spotted. Known as bounding overwatch, this movement involves stationary squad members who remain in cover, while their colleagues run for the next cover point. Figure 3.64 illustrates this.

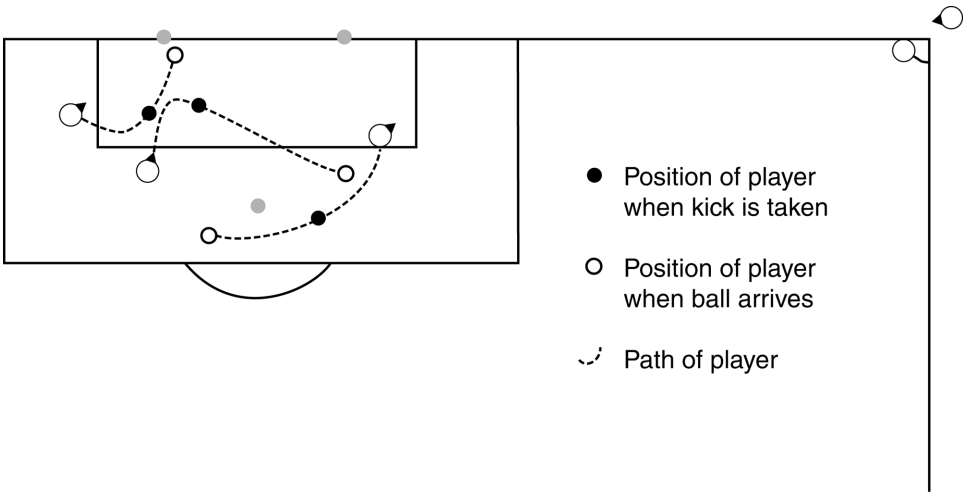

Figure 3.63: A corner kick in soccer

Dynamic formation patterns are not limited to creating set plays for sports games, they can also be used to create a very simple but effective approximation of bounding overwatch. Rather than moving between set locations on a sports field, the formation slots will move in a predictable sequence between whatever cover is near to the characters.

First we need access to the set of cover points in the game. A cover point is some location in the game where a character will be safe if it takes cover. These locations can be created manually by the level designers, or they can be calculated from the layout of the level. Chapter 6 will look at how cover points are created and used in much more detail. For our purposes here, we'll assume that there is some set of cover points available.

We need a rapid method of getting a list of cover points in the region surrounding the anchor point of the formation. The overwatch formation pattern accesses this list and chooses the closest set of cover points to the formation's anchor point. If there are four slots, it finds four cover points, and so on.

When asked to return the location of each slot, the formation pattern uses one of this set of cover points for each slot. This is shown in Figure 3.65. For each of the illustrated formation anchor points, the slot positions correspond to the nearest cover points.

Thus the pattern of the formation is linked to the environment, rather than geometrically fixed beforehand. As the formation moves, cover points that used to correspond to a slot will suddenly not be part of the set of nearest points. As one cover point leaves the list, another (by definition) will enter. The trick is to give the new arriving cover point to the slot whose cover point has just been removed and not assign all the cover points to slots afresh.

Because each character is assigned to a particular slot, using some kind of slot ID, the

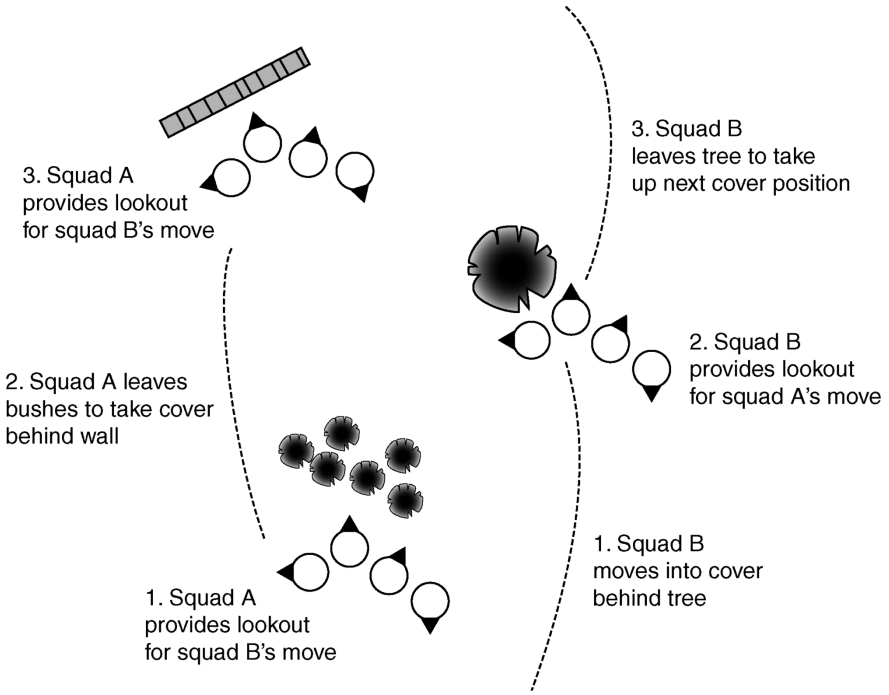

Figure 3.64: Bounding overwatch

newly valid slot should have the same ID as the recently disappeared slot. The cover points that are still valid should all still have the same IDs. This typically requires checking the new set of cover points against the old ones and reusing ID values.

Figure 3.66 shows the character at the back of the group assigned to a cover point called slot 4. A moment later, the cover point is no longer one of the four closest to the formation's anchor point. The new cover point, at the front of the group, reuses the slot 4 ID, so the character at the back (who is assigned to slot 4) now finds its target has moved and steers toward it.

Tactical Motion and Anchor Point Moderation

We can now run the formation system on our tactical formation. We need to turn off moderation of the anchor point's movement; otherwise, the characters are likely to get stuck at one set of cover points. Their center of mass will not change, since the formation is stationary at their cover points. Therefore, the anchor point will not move forward, and the formation will not get a chance to find new cover points.

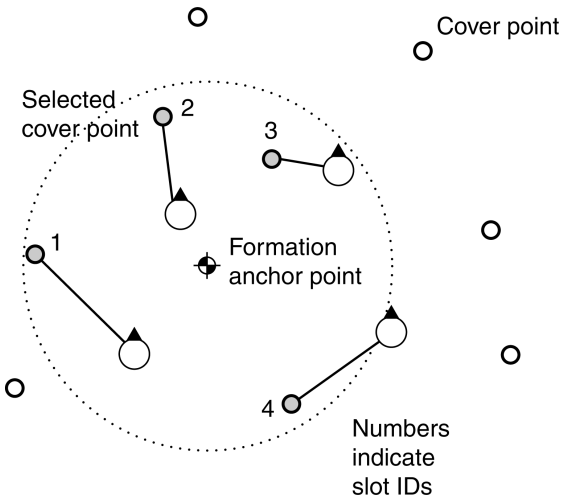

Figure 3.65: Formation patterns match cover points

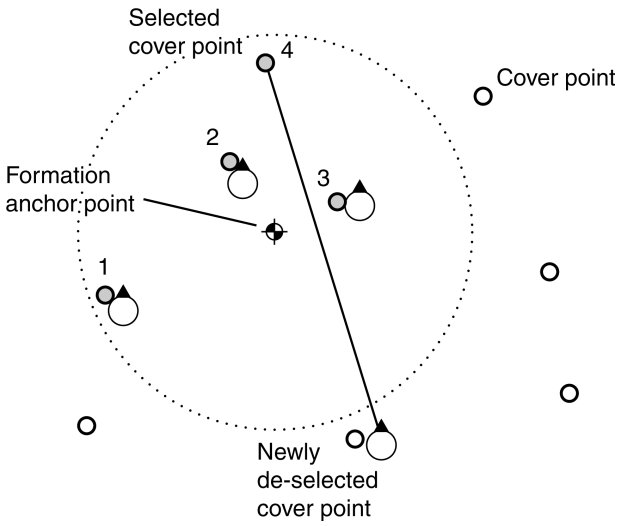

Figure 3.66: An example of slot change in bounding overwatch

Because moderation is now switched off, it is essential to make the anchor point move slowly in comparison with the individual characters. This is what you'd expect to see in any case, as bounding overwatch is not a fast maneuver.

An alternative is to go back to the idea of having a leader character that acts as the anchor point. This leader character can be under the player's control, or it can be controlled with some regular steering behavior. As the leader character moves, the rest of the squad moves in bounding overwatch around it. If the leader character moves at full speed, then its squad doesn't have time to take their defensive positions, and it appears as if they are simply following behind the leader. If the leader slows down, then they take cover around it.

To support this, make sure that any cover point near the leader is excluded from the list of cover points that can be turned into slots. Otherwise, other characters may try to join the leader in its cover.

3.8 MOTOR CONTROL

So far the chapter has looked at moving characters by being able to directly affect their physical state. This is an acceptable approximation in many cases. But, increasingly, motion is being controlled by physics simulation. This is almost universal in driving games, where it is the cars that are doing the steering. It was typically not used for human characters, but with the advent of integrated game engines such as Unity and Unreal Engine, it is often easier to use the physics system for all large-scale movement, including humans.

The outputs from steering behaviors can be seen as movement requests. An arrive behavior, for example, might request an acceleration in one direction. We can add a motor control layer to our movement solution that takes this request and works out how to best execute it; this is the process of actuation. In simple cases this is sufficient, but there are occasions where the capabilities of the actuator need to have an effect on the output of steering behaviors.

Think about a car in a driving game. It has physical constraints on its movement: it cannot turn while stationary; the faster it moves, the slower it can turn (without going into a skid); it can brake much more quickly than it can accelerate; and it only moves in the direction it is facing (we'll ignore power slides for now). On the other hand, a tank has different characteristics; it can turn while stationary, but it also needs to slow for sharp corners. And human characters will have different characteristics again. They will have sharp acceleration in all directions and different top speeds for moving forward, sideways, or backward.

When we simulate vehicles in a game, we need to take into account their physical capabilities. A steering behavior may request a combination of accelerations that is impossible for the vehicle to carry out. We need some way to end up with a maneuver that the character can perform.

A very common situation that arises in first- and third-person games is the need to match animations. Typically, characters have a palette of animations. A walk animation, for example, might be scaled so that it can support a character moving between 0.8 and 1.2 meters per second. A jog animation might support a range of 2.0 to 4.0 meters per second. The character

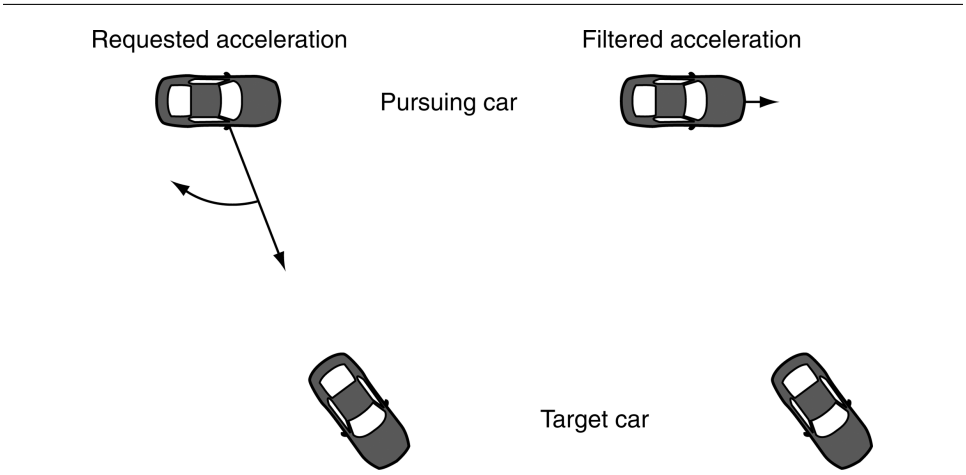

Figure 3.67: Requested and filtered accelerations

needs to move in one of these two ranges of speed; no other speed will do. The actuator, therefore, needs to make sure that the steering request can be honored using the ranges of movement that can be animated.

There are two ways of approaching actuation: output filtering and capability-sensitive steering.

3.8.1 OUTPUT FILTERING

The simplest approach to actuation is to filter the output of steering based on the capabilities of the character. In Figure 3.67, we see a stationary car that wants to begin chasing another. The indicated linear and angular accelerations show the result of a pursue steering behavior. Clearly, the car cannot perform these accelerations: it cannot accelerate sideways, and it cannot begin to turn without moving forward.

A filtering algorithm simply removes all the components of the steering output that cannot be achieved. In the example case the result has no angular acceleration and a smaller linear acceleration in its forward direction.

If the filtering algorithm is run every frame (even if the steering behavior isn't), then the car will take the indicated path. At each frame the car accelerates forward, allowing it to accelerate angularly. The rotation and linear motion serve to move the car into the correct orientation so that it can go directly after its quarry.

This approach is very fast, easy to implement, and surprisingly effective. It even naturally provides some interesting behaviors. If we rotate the car in the example below so that the target is almost behind it, then the path of the car will be a J-turn, as shown in Figure 3.68.

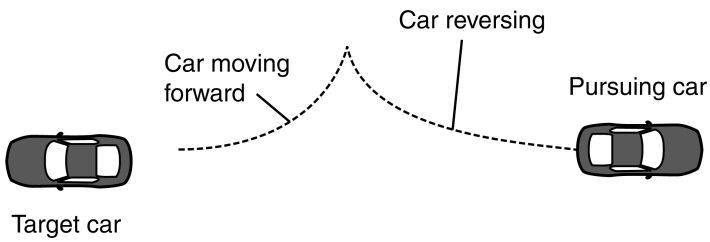

Figure 3.68: A J-turn emerges

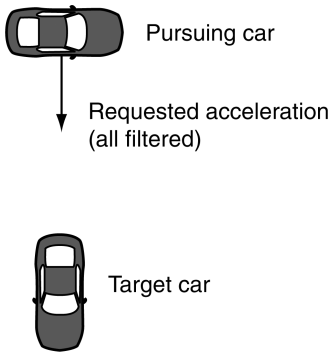

Figure 3.69: Everything is filtered: nothing to do

There are problems with this approach, however. When we remove the unavailable components of motion, we will be left with a much smaller acceleration than originally requested. In the first example above, the initial acceleration is small in comparison with the requested acceleration. In this case it doesn't look too bad. We can justify it by saying that the car is simply moving off slowly to perform its initial turn. In many cases it will be obvious and problematic.

The most obvious solution is to scale the final request so that it is the same magnitude as the initial request. This makes sure that a character doesn't move more slowly because its request is being filtered.

This works in some cases, but in Figure 3.69 the problem of filtering has become pathological. There is now no component of the request that can be performed by the car. Filtering alone will leave the car immobile until the target moves or until numerical errors in the calculation resolve the deadlock.

To resolve this last case, we can detect if the final result is zero and engage a different ac-

tuation method. This might be a complete solution such as the capability-sensitive technique below, or it could be a simple heuristic such as drive forward and turn hard.

In my experience a majority of cases can simply be solved with filtering-based actuation. Where it tends not to work is where there is a small margin of error in the steering requests. For driving at high speed, maneuvering through tight spaces, matching the motion in an animation, or jumping, the steering request needs to be honored as closely as possible. Filtering can cause problems, but, to be fair, so can the other approaches in this section (although to a lesser extent).

3.8.2 CAPABILITY-SENSITIVE STEERING

A different approach to actuation is to move the actuation into the steering behaviors themselves. Rather than generating movement requests based solely on where the character wants to go, the AI also takes into account the physical capabilities of the character.

If the character is pursuing an enemy, it will consider each of the maneuvers that it can apply and choose the one that best achieves the goal of catching the target. If the set of maneuvers that can be performed is relatively small (we can move forward or turn left or right, for example), then we can simply look at each in turn and determine the situation after the maneuver is complete. The winning action is the one that leads to the best situation (the situation with the character nearest its target, for example).

In most cases, however, there is an almost unlimited range of possible actions that a character can take. It may be able to move with a range of different speeds, for example, or to turn through a range of different angles. A set of heuristics is needed to work out what action to take depending on the current state of the character and its target. Section 3.8.3 gives examples of heuristic sets for a range of common movement AIs.

The key advantage of this approach is that we can use information discovered in the steering behavior to determine what movement to take. Figure 3.70 shows a skidding car that needs to avoid an obstacle. If we were using a regular obstacle avoiding steering behavior, then path A would be chosen. Using output filtering, this would be converted into putting the car into reverse and steering to the left.

We could create a new obstacle avoidance algorithm that considers both possible routes around the obstacle, in the light of a set of heuristics (such as those in Section 3.8.3).

Because a car will prefer to move forward to reach its target, it would correctly use route B, which involves accelerating to avoid the impact. This is the choice a rational human being would make.

There isn't a particular algorithm for capability-sensitive steering. It involves implementing heuristics that model the decisions a human being would make in the same situation: when it is sensible to use each of the vehicle's possible actions to get the desired effect.

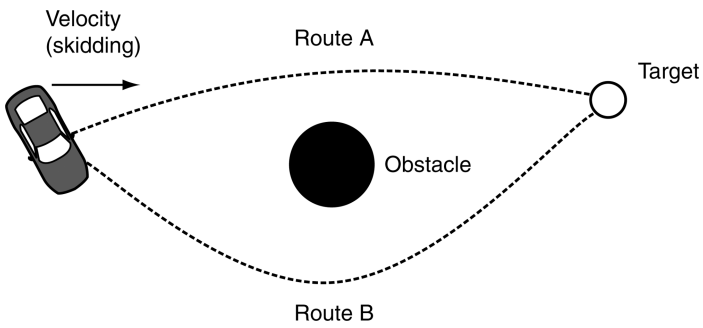

Figure 3.70: Heuristics make the right choice

Coping with Combined Steering Behaviors

Although it seems an obvious solution to bring the actuation into the steering behaviors, it causes problems when combining behaviors together. In a real game situation, where there will be several steering concerns active at one time, we need to do actuation in a more global way.

One of the powerful features of steering algorithms, as we've seen earlier in the chapter, is the ability to combine concerns to produce complex behaviors. If each behavior is trying to take into account the physical capabilities of the character, they are unlikely to give a sensible result when combined.

If you are planning to blend steering behaviors, or combine them using a decision tree, blackboard system, or steering pipeline, it is advisable to delay actuation to the last step, rather than actuating as you go.

This final actuation step will normally involve a set of heuristics. At this stage we don't have access to the inner workings of any particular steering behavior; we can't look at alternative obstacle avoidance solutions, for example. The heuristics in the actuator, therefore, need to be able to generate a roughly sensible movement guess for any kind of input; they will be limited to acting on one input request with no additional information.

3.8.3 COMMON ACTUATION PROPERTIES

This section looks at common actuation restrictions for a range of movement AI in games, along with a set of possible heuristics for performing context-sensitive actuation.

Human Characters

Human characters can move in any direction relative to their facing, although they are considerably faster in their forward direction than any other. As a result, they will rarely try to achieve their target by moving sideways or backward, unless the target is very close.

They can turn very fast at low speed, but their turning abilities decrease at higher speeds. This is usually represented by a "turn on the spot" animation that is only available to stationary or very slow-moving characters. At a walk or a run, the character may either slow and turn on the spot or turn in its motion (represented by the regular walk or run animation, but along a curve rather than a straight line).

Actuation for human characters depends, to a large extent, on the animations that are available. At the end of Chapter 4, we will look at a technique that can always find the best combination of animations to reach its goal. Most developers simply use a set of heuristics, however.

- If the character is stationary or moving very slowly, and if it is a very small distance from its target, it will step there directly, even if this involves moving backward or sidestepping.
- If the target is farther away, the character will first turn on the spot to face its target and then move forward to reach it.
- If the character is moving with some speed, and if the target is within a speed-dependent arc in front of it, then it will continue to move forward but add a rotational component (usually while still using the straight line animation, which puts a natural limit on how much rotation can be added to its movement without the animation looking odd).
- If the target is outside its arc, then it will stop moving and change direction on the spot before setting off once more.

The radius for sidestepping, how fast is "moving very slowly," and the size of the arc are all parameters that need to be determined and, to a large extent, that depend on the scale of the animations that the character will use.

Cars and Motorbikes

Typical motor vehicles are highly constrained. They cannot turn while stationary, and they cannot control or initiate sideways movement (skidding). At speed, they typically have limits to their turning capability, which is determined by the grip of their tires on the ground.

In a straight line, a motor vehicle will be able to brake more quickly than accelerate and will be able to move forward at a higher speed (though not necessarily with greater acceleration) than backward. Motorbikes almost always have the constraint of not being able to travel backward at all.

There are two decision arcs used for motor vehicles, as shown in Figure 3.71. The forward arc contains targets for which the car will simply turn without braking. The rear arc contains

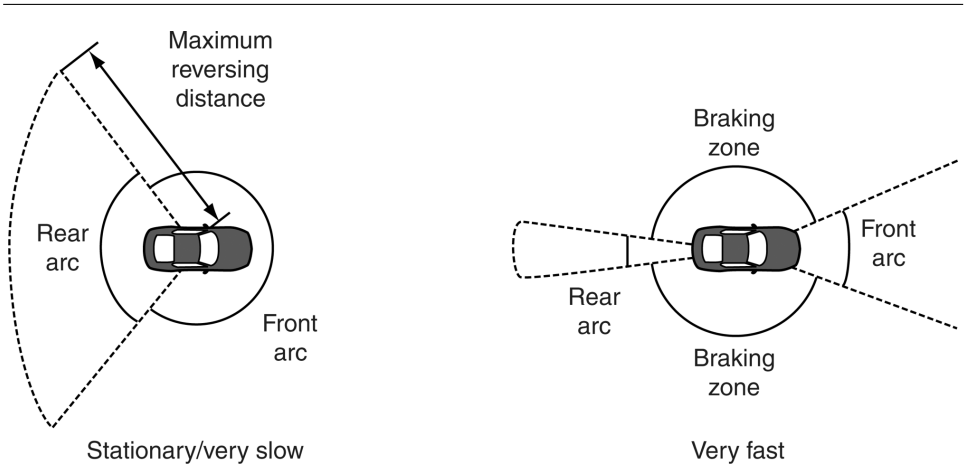

Figure 3.71: Decision arcs for motor vehicles

targets for which the car will attempt to reverse. This rear arc is zero for motorbikes and will usually have a maximum range to avoid cars reversing for miles to reach a target behind them.

At high speeds, the arcs shrink, although the rate at which they do so depends on the grip characteristics of the tires and must be found by tweaking. If the car is at low speed (but not at rest), then the two arcs should touch, as shown in the figure. The two arcs must still be touching when the car is moving slowly. Otherwise, the car will attempt to brake to stationary in order to turn toward a target in the gap. Because it cannot turn while stationary, this will mean it will be unable to reach its goal. If the arcs are still touching at too high a speed, then the car may be traveling too fast when it attempts to make a sharp turn and might skid.

- If the car is stationary, then it should accelerate.
- If the car is moving and the target lies between the two arcs, then the car should brake while turning at the maximum rate that will not cause a skid. Eventually, the target will cross back into the forward arc region, and the car can turn and accelerate toward it.
- If the target is inside the forward arc, then continue moving forward and steer toward it. Cars that should move as fast as possible should accelerate in this case. Other cars should accelerate to their optimum speed, whatever that might be (the speed limit for a car on a public road, for example).
- If the target is inside the rearward arc, then accelerate backward and steer toward it.

This heuristic can be a pain to parameterize, especially when using a physics engine to drive the dynamics of the car. Finding the forward arc angle so that it is near the grip limit of the tires but doesn't exceed it (to avoid skidding all the time) can be a pain. In most cases it is best to err on the side of caution, giving a healthy margin of error.

A common tactic is to artificially boost the grip of AI-controlled cars. The forward arc

can then be set so it would be right on the limit, if the grip was the same as for the player's car. In this case it is the AI that is limiting the capabilities of the car, not the physics, but its vehicle does not behave in an unbelievable or unfair way. The only downside with this approach is that the car will never skid out, which may be a desired feature of the game.

These heuristics are designed to make sure the car does not skid. In some games lots of wheel spinning and handbrake turns are the norm, and the parameters need to be tweaked to allow this.

Tracked Vehicles (Tanks)

Tanks behave in a very similar manner to cars and bikes. They are capable of moving forward and backward (typically with much smaller acceleration than a car or bike) and turning at any speed. At high speeds, their turning capabilities are limited by grip once more. At low speed or when stationary, they can turn very rapidly.

Tanks use decision arcs in exactly the same way as cars. There are two differences in the heuristic.

- The two arcs may be allowed to touch only at zero speed. Because the tank can turn without moving forward, it can brake right down to nothing to perform a sharp turn. In practice this is rarely needed, however. The tank can turn sharply while still moving forward. It doesn't need to stop.
- The tank does not need to accelerate when stationary.

3.9 MOVEMENT IN THE THIRD DIMENSION

So far we have looked at 2D steering behavior. We allowed the steering behavior to move vertically in the third dimension, but forced its orientation to remain about the up vector. This is 2½D, suitable for most development needs.

Full 3D movement is required if your characters aren't limited by gravity. Characters scurrying along the roof or wall, airborne vehicles that can bank and twist, and turrets that rotate in any direction are all candidates for steering in full three dimensions.

Because 2½D algorithms are so easy to implement, it is worth thinking hard before you take the plunge into full three dimensions. There is often a way to shoehorn the situation into 2½D and take advantage of the faster execution that it provides. At the end of this chapter is an algorithm, for example, that can model the banking and twisting of aerial vehicles using 2½D math. There comes a point, however, where the shoehorning takes longer to code and is more finicky than the 3D math.

This section looks at introducing the third dimension into orientation and rotation. It then considers the changes that need to be made to the primitive steering algorithms we saw earlier. Finally, we'll look at a common problem in 3D steering: controlling the rotation for air and space vehicles.

3.9.1 ROTATION IN THREE DIMENSIONS

To move to full three dimensions we need to expand our orientation and rotation to be about any angle. Both orientation and rotation in three dimensions have three degrees of freedom. We can represent rotations using a 3D vector, consisting of the rotation around the x–, y– and z–axes. But for reasons beyond the scope of this book, doing so makes it difficult or impossible to combine or adjust orientations: we can end up with situations where seemingly very different vectors represent the same orientation, or when a desired change in orientation is impossible.

The most practical data structure for 3D orientation—the representation used most often in games—is the quaternion: a value with 4 real components, the size of which (i.e., the Euclidean size of the 4 components) is always 1. The requirement that the size is always 1 reduces the degrees of freedom from 4 (for 4 values) to 3.

Mathematically, quaternions are hypercomplex numbers. Their mathematics is not the same as that of a 4-element vector, so dedicated routines are needed for multiplying quaternions and multiplying position vectors by them. A good 3D math library will have the relevant code, and the graphics engine you are working with will almost certainly use quaternions. The Unity game engine, for example, uses quaternions internally, but provides functions to convert to and from axis angles.

It is possible to also represent orientation using matrices, and this was the dominant technique up until the mid-1990s. These 9-element structures have additional constraints to reduce the degrees of freedom to 3. Because they require a good deal of checking to make sure the constraints are not broken, they are no longer widely used.

In the two dimensional case, orientation was more complex than rotation. 2D orientation wraps every 2π radians, where rotation can have any value. An analogous thing happens in 3D. Orientation is best represented as a quaternion, but rotation can be a regular vector.

The rotation vector has three components. It is related to the axis of rotation and the speed of rotation according to:

$$\vec{r} = \begin{bmatrix} a_x\omega \\ a_y\omega \\ a_z\omega \end{bmatrix} \tag{3.10}$$

where $\begin{bmatrix} a_x & a_y & a_z \end{bmatrix}^T$ is the axis of rotation, and ω is the angular velocity, in radians per second (units are critical; the math is more complex if degrees per second are used).

The orientation quaternion has four components: $\begin{bmatrix} r & i & j & k \end{bmatrix}$, (sometimes called $\begin{bmatrix} w & x & y & z \end{bmatrix}$—although personally I think that confuses them with a position vector, which in homogenous form has an additional w coordinate).

It is also related to an axis and angle. This time the axis and angle correspond to the minimal rotation required to transform from a reference orientation to the desired orientation. Every possible orientation can be represented as some rotation from a reference orientation about a single fixed axis.

The axis and angle are converted into a quaternion using the following equation:

$$\hat{q} = \begin{bmatrix} \cos\frac{\theta}{2} \\ a_x \sin\frac{\theta}{2} \\ a_y \sin\frac{\theta}{2} \\ a_z \sin\frac{\theta}{2} \end{bmatrix} \tag{3.11}$$

where $\begin{bmatrix} a_x & a_y & a_z \end{bmatrix}^T$ is the axis, as before, θ is the angle, and $\hat{p}$ indicates that p is a quaternion.

Note that different implementations use different orders for the elements in a quaternion. Often, the r component appears at the end.

We have four numbers in the quaternion, but we only need 3 degrees of freedom. The quaternion needs to be further constrained, so that it has a size of 1 (i.e., it is a unit quaternion). This occurs when:

$$r^2 + i^2 + j^2 + k^2 = 1$$

Verifying that this always follows from the axis and angle representation is left as an exercise. Even though the math of quaternions used for geometrical applications normally ensure that quaternions remain of unit length, numerical errors can make them wander. Most quaternion math libraries have extra bits of code that periodically normalize the quaternion back to unit length. We will rely on the fact that quaternions are unit length.

The mathematics of quaternions is a wide field, and I will only cover those topics that we need in the following sections. Other books in this series, particularly Eberly [12], contain in-depth mathematics for quaternion manipulation.

3.9.2 CONVERTING STEERING BEHAVIORS TO THREE DIMENSIONS

In moving to three dimensions, only the angular mathematics has changed. To convert our steering behaviors into three dimensions, we divide them into those that do not have an angular component, such as pursue or arrive, and those that do, such as align. The former translates directly to three dimensions, and the latter requires different math for calculating the angular acceleration required.

Linear Steering Behaviors in Three Dimensions

In the first two sections of the chapter we looked at 14 steering behaviors. Of these, 10 did not explicitly have an angular component: seek, flee, arrive, pursue, evade, velocity matching, path following, separation, collision avoidance, and obstacle avoidance.

Each of these behaviors works linearly. They try to match a given linear position or velocity, or they try to avoid matching a position. None of them requires any modification to move from 2½D to 3 dimensions. The equations work unaltered with 3D positions.

Angular Steering Behaviors in Three Dimensions

The remaining four steering behaviors are align, face, look where you're going, and wander. Each of these has an explicit angular component. Align, look where you're going, and face are all purely angular. Align matches another orientation, face orients toward a given position, and look where you're going orients toward the current velocity vector.

Between the three purely angular behaviors we have orientation based on three of the four elements of a kinematic (it is difficult to see what orientation based on rotation might mean). We can update each of these three behaviors in the same way.

The wander behavior is different. Its orientation changes semi-randomly, and the orientation then motivates the linear component of the steering behavior. We will deal with wander separately.

3.9.3 ALIGN

Align takes as input a target orientation and tries to apply a rotation to change the character's current orientation to match the target.

In order to do this, we'll need to find the required rotation between the target and current quaternions. The quaternion that would transform the start orientation to the target orientation is

$$\hat{q} = \hat{s}^{-1}\hat{t}$$

where $\hat{s}$ is the current orientation and $\hat{t}$ is the target quaternion. Because we are dealing with unit quaternions (the square of their elements sum to 1), the quaternion inverse is equal to the conjugate $\hat{q}^*$, and is given by:

$$\hat{q}^{-1} = \begin{bmatrix} r \\ i \\ j \\ k \end{bmatrix}^{-1} = \begin{bmatrix} r \\ -i \\ -j \\ -k \end{bmatrix}$$

In other words, the axis components are flipped. This is because the inverse of the quaternion is equivalent to rotating about the same axis, but by the opposite angle (i.e. $\theta^{-1} = -\theta$). For each of the x, y and z components, related to $\sin\theta$, we have $\sin -\theta = -\sin\theta$, where as the w component is related to $\cos\theta$, and $\cos -\theta = -\cos\theta$, leaving the w component unchanged.

We now need to convert this quaternion into a rotation vector. Firstly we split the quaternion back into an axis and angle:

$$\theta = 2\cos^{-1} q_w$$

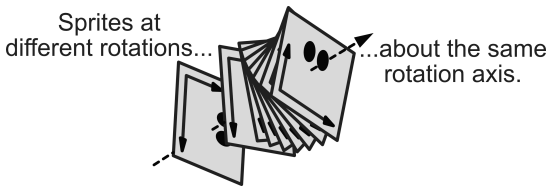

Figure 3.72: Infinite number of possible orientations per axis vector

$$\vec{a} = \frac{1}{\sin\frac{\theta}{2}} \begin{bmatrix} q_i \\ q_j \\ q_k \end{bmatrix}$$

In the same way as for the original align behavior, we would like to choose a rotation so that the character arrives at the target orientation with zero rotation speed. We know the axis through which this rotation needs to occur, and we have a total angle that needs to be achieved. We only need to find the rotation speed to choose.

Finding the correct rotation speed is equivalent to starting at zero orientation in two dimensions and having a target orientation of θ. We can apply the same algorithm used in two dimensions to generate a rotation speed, ω, and then combine this with the axis $\vec{a}$ above to produce an output rotation, using Equation 3.10.

3.9.4 ALIGN TO VECTOR

Both the face steering behavior and look where you're going started with a vector along which the character should align. In the former case it is a vector from the current character position to a target, and in the latter case it is the velocity vector. We are assuming that the character is trying to position its z-axis (the axis it is looking down) in the given direction.

In two dimensions it is simple to calculate a target orientation from a vector using the `atan2` function available in most languages. In three dimensions there is no such shortcut to generate a quaternion from a target facing vector.

In fact, there are an infinite number of orientations that look down a given vector, as illustrated in Figure 3.72. This means that there is no single way to convert a vector to an orientation. We have to make some assumptions to simplify things.

The most common assumption is to bias the target toward a "base" orientation. We'd like to choose an orientation that is as near to the base orientation as possible. In other words, we start with the base orientation and rotate it through the minimum angle possible (about an appropriate axis) so that its local z-axis points along our target vector.

This minimum rotation can be found by converting the z-direction of the base orientation

into a vector and then taking the vector product of this and the target vector. The vector product gives:

$$\vec{z_b} \times \vec{t} = \vec{r}$$

where $\vec{z_b}$ is the vector of the local z-direction in the base orientation, $\vec{t}$ is the target vector and $\vec{r}$, being a cross product is defined to be:

$$\vec{r} = \vec{z_b} \times \vec{t} = (|\vec{z_b}||\vec{t}| \sin\theta)\vec{a_r} = \sin\theta\vec{a_r}$$

where θ is the angle, and $\vec{a_r}$ is the axis of minimum rotation. Because the axis will be a unit vector (i.e., $|\vec{a_r}| = 1$), we can recover angle $\theta = \sin^{-1}|\vec{r}|$ and divide $\vec{r}$ by this to get the axis. This will not work if $\sin\theta = 0$ (i.e. $\theta = n\pi$ for all $n \in \mathbb{Z}$). This corresponds to an intuition about the physical properties of rotation: if the rotation angle is zero, then it doesn't make sense to talk about a rotation axis. If the rotation is of π radians ($90°$), then any axis will do: there is no particular axis that requires a smaller rotation than any other.

As long as $\sin\theta \neq 0$, we can generate a target orientation by first turning the axis and angle into a quaternion, $\hat{r}$ (using Equation 3.11) and applying the formula:

$$\hat{t} = \hat{b}^{-1}\hat{r}$$

where $\hat{b}$ is the quaternion representation of the base orientation, and $\hat{t}$ is the target orientation to align to.

If $\sin\theta = 0$, then we have two possible situations: either the target z-axis is the same as the base z-axis, or it is π radians away from it. In other words: $\vec{z_b} = \pm\vec{z_t}$. In each case we use the base orientation's quaternion, with the appropriate sign change.

$$\hat{t} = \begin{cases} +\hat{b} & \text{if } \vec{z_b} = \vec{z_t} \\ -\hat{b} & \text{otherwise} \end{cases}$$

The most common base orientation is the zero orientation: $\begin{bmatrix}1 & 0 & 0 & 0\end{bmatrix}$. This has the effect that the character will stay upright when its target is in the x–z plane. Tweaking the base vector can provide visually pleasing effects. We could tilt the base orientation when the character's rotation is high to force them to lean into their turns, for example.

We will implement this process in the context of the face steering behavior below.

3.9.5 FACE

Using the align to vector process above, both face and look where you're going can be easily implemented using the same algorithm as we used at the start of the chapter, by replacing the `atan2` calculation with the procedure above to calculate the new target orientation.

By way of an illustration, I will give an implementation for the face steering behavior in three dimensions. Since this is a modification of the algorithm given earlier in the chapter, I won't discuss the algorithm in any depth (see the previous version for more information).

```
class Face3D extends Align3D:
    # The base orientation used to calculate facing.
    baseOrientation: Quaternion

    # Overridden target.
    target: Kinematic3D

    # ... Other data is derived from the superclass ...

    # Calculate an orientation to face along a given vector.
    function calculateOrientation(vector):
        # Get the base vector by transforming the z-axis by base
        # orientation (this only needs to be done once for each
        # base orientation, so could be cached between calls).
        zVector = new Vector(0, 0, 1)
        baseZVector = zVector * baseOrientation

        # If we're done (or the opposite) use the base quaternion.
        if baseZVector == vector:
            return baseOrientation
        elif baseZVector == -vector:
            return -baseOrientation

        # Otherwise find the minimum rotation to the target.
        axis = crossProduct(baseZVector, vector)
        angle = asin(axis.length())
        axis.normalize()

        # Pack these into a quaternion and return it.
        sinAngle = sin(angle / 2)
        return new Quaternion(
            cos(angle / 2),
            sinAngle * axis.x,
            sinAngle * axis.y,
            sinAngle * axis.z)

    # Implemented as it was in Pursue.
    function getSteering() -> SteeringOutput3D:
        # 1. Calculate the target to delegate to align
        # Work out the direction to target.
        direction = target.position - character.position

        # Check for a zero direction, and make no change if so.
        if direction.length() == 0:
            return null

```

```
        # 2. Delegate to align.
        Align3D.target = explicitTarget
        Align3D.target.orientation = calculateOrientation(direction)
        return Align3D.getSteering()
```

This implementation assumes that we can take the vector product of two vectors using a `crossProduct` function.

We also need to look at the mechanics of transforming a vector by a quaternion. In the code above this is performed with the `*` operator, so `vector * quaternion` should return a vector that is equivalent to rotating the given vector by the quaternion. Mathematically, this is given by:

$$\hat{v}' = \hat{q}\hat{v}\hat{q}^*$$

where $\hat{v}$ is a quaternion derived from the vector, according to

$$\hat{v} = \begin{bmatrix} 0 \\ v_x \\ v_y \\ v_z \end{bmatrix}$$

and $\hat{q}^*$ is the conjugate of the quaternion, which is the same as the inverse for unit quaternions. This can be implemented as:

```
# Transform the vector by the given quaternion.
function transform(vector, orientation):
    # Convert the vector into a quaternion.
    vectorAsQ = new Quaternion(0, vector.x, vector.y, vector.z)

    # Transform it.
    vectorAsQ = orientation * vectorAsQ * (-orientation)

    # Unpick it into the resulting vector.
    return new Vector(vectorAsQ.i, vectorAsQ.j, vectorAsQ.k)
```

Quaternion multiplication, in turn, is defined by:

$$\hat{p}\hat{q} = \begin{bmatrix} p_r q_r - p_i q_i - p_j q_j - p_k q_k \\ p_r q_i + p_i q_r + p_j q_k - p_k q_j \\ p_r q_j + p_j q_r - p_i q_k + p_k q_i \\ p_r q_k + p_k q_r + p_i q_j - p_j q_i \end{bmatrix}$$

It is important to note that the order does matter. Unlike normal arithmetic, quaternion multiplication isn't commutative. In general, $\hat{p}\hat{q} \neq \hat{q}\hat{p}$.

3.9.6 LOOK WHERE YOU'RE GOING

Look where you're going would have a very similar implementation to face. We simply replace the calculation for the direction vector in the `getSteering` method with a calculation based on the character's current velocity:

```
# Work out the direction to target.
direction: Vector = character.velocity
direction.normalize()
```

3.9.7 WANDER

In the 2D version of wander, a target point was constrained to move around a circle offset in front of the character at some distance. The target moved around this circle randomly. The position of the target was held at an angle, representing how far around the circle the target lay, and the random change in that was generated by adding a random amount to the angle.

In three dimensions, the equivalent behavior uses a 3D sphere on which the target is constrained, again offset at a distance in front of the character. We cannot use a single angle to represent the location of the target on the sphere, however. We could use a quaternion, but it becomes difficult to change it by a small random amount without a good deal of math.

Instead, we represent the position of the target on the sphere as a 3D vector, constraining the vector to be of unit length. To update its position, we simply add a random amount to each component of the vector and normalize it again. To avoid the random change making the vector zero (and hence making it impossible to normalize), we make sure that the maximum change in any component is smaller than $\frac{1}{\sqrt{3}}$.

After updating the target position on the sphere, we transform it by the orientation of the character, scale it by the wander radius, and then move it out in front of the character by the wander offset, exactly as in the 2D case. This keeps the target in front of the character and makes sure that the turning angles are kept low.

Rather than using a single value for the wander offset, we now use a vector. This would allow us to locate the wander circle anywhere relative to the character. This is not a particularly useful feature. We will want it to be in front of the character (i.e., having only a positive z coordinate, with 0 for x and y values). Having it in vector form does simplify the math, however. The same thing is true of the maximum acceleration property: replacing the scalar with a 3D vector simplifies the math and provides more flexibility.

With a target location in world space, we can use the 3D face behavior to rotate toward it and accelerate forward to the greatest extent possible.

In many 3D games we want to keep the impression that there is an up and down direction. This illusion is damaged if the wanderer can change direction up and down as fast as it can in the x–z plane. To support this, we can use two radii for scaling the target position: one for scaling the x and z components and the other for scaling the y component. If the y scale is smaller, then the wanderer will turn more quickly in the x–z plane. Combined with using the

face implementation described above, with a base orientation where up is in the direction of the y-axis, this gives a natural look for flying characters, such as bees, birds, or aircraft.

The new wander behavior can be implemented as follows:

```
class Wander3D extends Face3D:
    # The radius and offset of the wander circle.
    wanderOffset: Vector
    wanderRadiusXZ: float
    wanderRadiusY: float

    # The maximum rate at which the wander orientation can change.
    # Should be strictly less than 1/sqrt(3) = 0.577 to avoid the
    # chance of ending up with a zero length wanderTarget.
    wanderRate: float

    # The current offset of the wander target.
    wanderTarget: Vector

    # The maximum acceleration of the character, though this is a 3D
    # vector, it typically has only a non-zero z component.
    maxAcceleration: Vector

    # ... Other data is derived from the superclass ...

    function getSteering( ) -> SteeringOutput3D:
        # 1. Calculate the target to delegate to face
        # Update the wander direction.
        wanderTarget.x += randomBinomial( ) * wanderRate
        wanderTarget.y += randomBinomial( ) * wanderRate
        wanderTarget.z += randomBinomial( ) * wanderRate
        wanderTarget.normalize( )

        # Calculate the transformed target direction
        # and scale it.
        target = wanderTarget * character.orientation
        target.x *= wanderRadiusXZ
        target.y *= wanderRadiusY
        target.z *= wanderRadiusXZ

        # Offset by the center of the wander circle.
        target += character.position +
                  wanderOffset * character.orientation

        # 2. Delegate it to face.
        result = Face3D.getSteering(target)

        # 3. Now set the linear acceleration to be at full
```

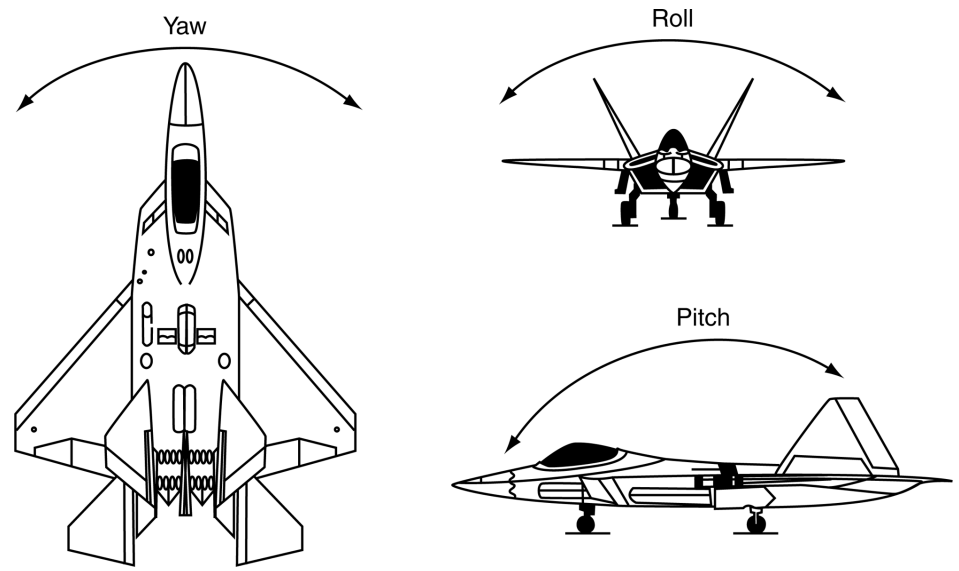

Figure 3.73: Local rotation axes of an aircraft

```
        # acceleration in the direction of the orientation.
        result.linear = maxAcceleration * character.orientation

        # Return it.
        return result
```

Again, this is heavily based on the 2D version and shares its performance characteristics. See the original definition for more information.

3.9.8 FAKING ROTATION AXES

A common issue with vehicles moving in three dimensions is their axis of rotation. Whether spacecraft or aircraft, they have different turning speeds for each of their three axes (see Figure 3.73): roll, pitch, and yaw. Based on the behavior of aircraft, we assume that roll is faster than pitch, which is faster than yaw.

If a craft is moving in a straight line and needs to yaw, it will first roll so that its up direction points toward the direction of the turn, then it can pitch up to turn in the correct direction. This is how aircraft are piloted, and it is a physical necessity imposed by the design of the wing and control surfaces. In space there is no such restriction, but we want to give the player some

kind of sense that craft obey physical laws. Having them yaw rapidly looks unbelievable, so we tend to impose the same rule: roll and pitch produce a yaw.

Most aircraft don't roll far enough so that all the turn can be achieved by pitching. In a conventional aircraft flying level, using only pitch to perform a right turn would involve rolling by π radians. This would cause the nose of the aircraft to dive sharply toward the ground, requiring significant compensation to avoid losing the turn (in a light aircraft it would be a hopeless attempt). Rather than tip the aircraft's local up vector so that it is pointing directly into the turn, we angle it slightly. A combination of pitch and yaw then provides the turn. The amount to tip is determined by speed: the faster the aircraft, the greater the roll. A Boeing 747 turning to come into land might only roll by $\frac{\pi}{6}$ radians ($15°$); an F-22 Raptor might tilt by $\frac{\pi}{2}$ radians ($45°$) or the same turn in an X-Wing by $\frac{5\pi}{6}$ ($75°$).

Most craft moving in three dimensions have an "up–down" axis. This can be seen in 3D space shooters as much as in aircraft simulators. Homeworld, for example, had an explicit up and down direction, to which craft would orient themselves when not moving. The up direction is significant because craft moving in a straight line, other than in the up direction, tend to align themselves with up.

The up direction of the craft points as near to up as the direction of travel will allow. This again is a consequence of aircraft physics: the wings of an aircraft are designed to produce lift in the up direction, so if you don't keep your local up direction pointing up, you are eventually going to fall out of the sky.

It is true that in a dog fight, for example, craft will roll while traveling in a straight line to get a better view, but this is a minor effect. In most cases the reason for rolling is to perform a turn.

It is possible to bring all this processing into an actuator to calculate the best way to trade off pitch, roll, and yaw based on the physical characteristics of the aircraft. If you are writing an AI to control a physically modeled aircraft, you may have to do this. For the vast majority of cases, however, this is overkill. We are interested in having enemies that just look right.

It is also possible to add a steering behavior that forces a bit of roll whenever there is a rotation. This works well but tends to lag. Pilots will roll before they pitch, rather than afterward. If the steering behavior is monitoring the rotational speed of the craft and rolling accordingly, there is a delay. If the steering behavior is being run every frame, this isn't too much of a problem. If the behavior is running only a couple of times a second, it can look very strange.

Both of the above approaches rely on techniques already covered in this chapter, so I won't revisit them here. There is another approach, used in some aircraft games and many space shooters, that fakes rotations based on the linear motion of the craft. It has the advantages that it reacts instantly and it doesn't put any burden on the steering system because it is a post-processing step. It can be applied to 2½D steering, giving the illusion of full 3D rotations.

The Algorithm

Movement is handled using steering behaviors as normal. We keep two orientation values. One is part of the kinematic data and is used by the steering system, and one is calculated for display. This algorithm calculates the latter value based on the kinematic data.

First, we find the speed of the vehicle: the magnitude of the velocity vector. If the speed is zero, then the kinematic orientation is used without modification. If the speed is below a fixed threshold, then the result of the rest of the algorithm will be blended with the kinematic orientation. Above the threshold the algorithm has complete control. As it drops below the threshold, there is a blend of the algorithmic orientation and the kinematic orientation, until at a speed of zero, the kinematic orientation is used.

At zero speed the motion of the vehicle can't produce any sensible orientation; it isn't moving. So we'll have to use the orientation generated by the steering system. The threshold and blending are there to make sure that the vehicle's orientation doesn't jump as it slows to a halt. If your application never has stationary vehicles (aircraft without the ability to hover, for example), then this blending can be removed.

The algorithm generates an output orientation in three stages. This output can then be blended with the kinematic orientation, as described above.

First, the vehicle's orientation about the up vector (its 2D orientation in a 2½D system) is found from the kinematic orientation. Call this value θ.

Second, the tilt of the vehicle is found by looking at the component of the vehicle's velocity in the up direction. The output orientation has an angle above the horizon given by:

$$\phi = \sin^{-1} \frac{\vec{v}.\vec{u}}{|\vec{v}|}$$

where v is its velocity (taken from the kinematic data) and u is a unit vector in the up direction.

Third, the roll of the vehicle is found by looking at the vehicle's rotation speed about the up direction (i.e., the 2D rotation in a 2½D system). The roll is given by:

$$\psi = \tan^{-1} \frac{r}{k}$$

where r is the rotation, and k is a constant that controls how much lean there should be. When the rotation is equal to k, then the vehicle will have a roll of $\frac{\pi}{2}$ radians. Using this equation, the vehicle will never achieve a roll of π radians, but very fast rotation will give very steep rolls.

The output orientation is calculated by combining the three rotations in the order θ, ϕ, ψ.

Pseudo-Code

The algorithm has the following structure when implemented:

```
function getFakeOrientation(kinematic: Kinematic3D,
                            maxSpeed: float,
                            rollScale: float):
    current: Quaternion = kinematic.orientation

    # Find the blend factors.
    speed = kinematic.velocity.length()
    if speed == 0:
        # No change if we're stationary.
        return current
    else if speed < maxSpeed:
        # Partly use the unchanged orientation.
        fakeBlend = speed / maxSpeed
    else:
        # We're completely faked.
        fakeBlend = 1.0
    kinematicBlend = 1.0 - fakeBlend

    # Find the faked axis orientations.
    yaw = current.as2DOrientation()
    pitch = asin(kinematic.velocity.y / speed)
    roll = atan2(kinematic.rotation, rollScale)

    # Combine them as quaternions.
    faked = orientationInDirection(roll, Vector(0, 0, 1))
    faked *= orientationInDirection(pitch, Vector(1, 0, 0))
    faked *= orientationInDirection(yaw, Vector(0, 1, 0))

    # Blend result.
    return current * (1.0 - fakeBlend) + faked * fakeBlend
```

Data Structures and Interfaces

The code relies on appropriate vector and quaternion mathematics routines being available, and we have assumed that we can create a vector using a three argument constructor.

Most operations are fairly standard and will be present in any vector math library. The `orientationInDirection` function of a quaternion is less common. It returns an orientation quaternion representing a rotation by a given angle about a fixed axis. It can be implemented in the following way:

```
function orientationInDirection(angle, axis):
    sinAngle = sin(angle / 2)
    return new Quaternion(
        cos(angle / 2),
        sinAngle * axis.x,
```

```
        sinAngle * axis.y,
        sinAngle * axis.z)
```

which is simply Equation 3.11 in code form.

Implementation Notes

The same algorithm also comes in handy in other situations. By reversing the direction of roll (ψ), the vehicle will roll outward with a turn. This can be applied to the chassis of cars driving (excluding the ϕ component, since there will be no controllable vertical velocity) to fake the effect of soggy suspension. In this case a high k value is needed.

Performance

The algorithm is O(1) in both memory and time. It involves an arcsine and an arctangent call and three calls to the `orientationInDirection` function. Arcsine and arctan calls are typically slow, even compared to other trigonometry functions. Various faster implementations are available. In particular, an implementation using a low-resolution lookup table (256 entries or so) would be perfectly adequate for our needs. It would provide 256 different levels of pitch or roll, which would normally be enough for the player not to notice that the tilting isn't completely smooth. The remainder of the algorithm is so efficient, however, that slow trigonometry functions are unlikely to be noticeable unless your game has thousands of moving characters.

EXERCISES

3.1 An character is at $p = (5, 6)$ and it is moving with velocity $v = (3, 3)$: If its target is at location $q = (8, 2)$, what is the desired direction to *seek* the target? (Hint: No trigonometry is required for this and other questions like it, just simple vector arithmetic.)

3.2 Using the same scenario as in Exercise 3.1, what is the desired direction to *flee* the target?

3.3 Using the same scenario as Exercise 3.1 and assuming the maximum speed of the AI character is 5, what are the final steering velocities for seek *and* flee?

3.4 Explain why the `randomBinomial` function described in Section 3.2.2 is more likely to return values around zero.

3.5 Using the same scenario as in Exercise 3.1, what are the final steering velocities for seek and flee if we use the dynamic version of seek and assume a maximum acceleration of 4.

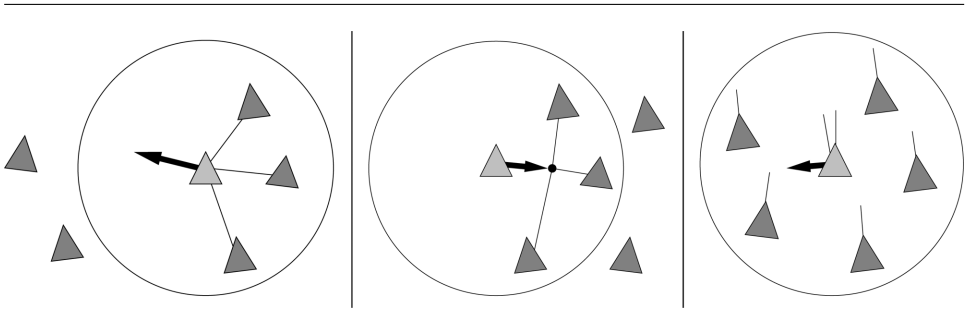

Figure 3.74: Components of the flocking steering behavior

3.6 Using the dynamic movement model and the answer from Exercise 3.5, what is the final position and orientation of the character after the update call? Assume that the time is $\frac{1}{60}$sec and that the maximum speed is still 5.

3.7 If the target in Exercise 3.1 is moving at a velocity $u = (3, 4)$ and the maximum prediction time is $\frac{1}{2}$sec, what is the predicted position of the target?

3.8 Using the predicted target position from Exercise 3.7 what are the resulting steering vectors for *pursuit* and *evasion*?

3.9 The three diagrams in Figure 3.74 represent Craig Reynolds's concepts of *separation*, *cohesion*, and *alignment* that are commonly used for flocking behavior.

Assume the following table gives the positions (in relative coordinates) and velocities of the 3 characters (including the first) in the first character's neighborhood:

Character	Position	Velocity	Distance	Distance squared
1	$(0, 0)$	$(2, 2)$	0	0
2	$(3, 4)$	$(2, 4)$		
3	$(5, 12)$	$(8, 2)$		

a) Fill in the the remainder of the table.

b) Use the values you filled in for the table to calculate the unnormalized separation direction using the inverse square law (assume $k = 1$ and that there is no maximum acceleration).

c) Now calculate the center of mass of all the characters to determine the unnormalized cohesion direction.

d) Finally, calculate the unnormalized alignment direction as the average velocity of the other characters.

3.10 Use the answers to Exercise 3.9 and weighting factors $\frac{1}{5}$, $\frac{2}{5}$, $\frac{2}{5}$ for (respectively) separation, cohesion, and alignment to show that the desired (normalized) *flocking* direction is approximately: $(0.72222, 0.69166)$.

3.11 Suppose a character A is located at (4, 2) with a velocity of (3, 4) and another character B is located at (20, 12) with velocity (−5, −1). By calculating the time of closest approach (see 3.1), determine if they will collide. If they will collide, determine a suitable evasive steering vector for character A.

3.12 Suppose an AI-controlled spaceship is pursuing a target through an asteroid field and the current velocity is (3, 4). If the high-priority collision avoidance group suggests a steering vector of $(0.01, 0.03)$, why might it be reasonable to consider a lower priority behavior instead?

3.13 Use Equation 3.2 to calculate the time before a ball in a soccer game lands on the pitch again if it is kicked from the ground at location (11, 4) with speed 10 in a direction $(\frac{3}{5}, \frac{4}{5})$

3.14 Use your answer to Exercise 3.13 and Equation 3.4 to calculate the position of impact of the ball. Why might the ball not actually end up at this location even if no other players interfere with it?

3.15 With reference to Figure 3.49, suppose a character is heading toward the jump point and will arrive in 0.1 time units and is currently traveling at velocity (0, 5), what is the required velocity matching steering vector if the minimum jump velocity is (0, 7)?

3.16 Show that in the case when the jump point and landing pad are the same height, Equation 3.7 reduces to approximately

$$t = 0.204 v_y$$

3.17 Suppose there is a jump point at (10, 3, 12) and a landing pad at (12, 3, 20), what is the required jump velocity if we assume a maximum jump velocity in the y-direction of 2?

3.18 Suppose we have three characters in a V formation with coordinates and velocities given by the following table:

Character	Assigned Slot Position	Actual Position	Actual Velocity
1	(20, 18)	(20, 16)	(0, 1)
2	(8, 12)	(6, 11)	(3, 1)
3	(32, 12)	(28, 9)	(9, 7)

a) Calculate the center of mass of the formation p_c and the average velocity v_c.

b) Use these values and Equation 3.8 with $k_{\text{offset}} = 1$ to calculate p_{anchor}.

c) Use your calculations to update the slot positions using the new calculated anchor point as in Equation 3.9.

d) What would be the effect on the anchor and slot positions if character 3 was killed?

3.19 In Figure 3.60 if the 2 empty slots in the formation on the right (with 2 elves and 7 fighters) are filled with the unassigned fighters, what is the total slot cost? Use the same table that was used to calculate the slot costs in Figure 3.61.

3.20 Calculate the ease of assignment value for each of the four character types (archer, elf, fighter, mage) used in Figure 3.61 (assume $k = 1600$).

3.21 Verify that the axis and angle representation always results in unit quaternions.

3.22 Suppose a character's current orientation in a 3D world is pointing along the x-axis, what is the required rotation (as a quaternion) to align the character with a rotation of $\frac{2\pi}{3}$ around the axis $(\frac{8}{170}, \frac{15}{17}, 0)$?

3.23 Suppose a plane in a flight simulator game has velocity (5, 4, 1), orientation $\frac{p}{4}$, rotation $\frac{p}{16}$, and roll scale $\frac{p}{4}$. What is the associated fake rotation?