Chapter 7

Action-Oriented AI

KEY TOPICS

- On Action Games
- Choreographed AIs
- Object Tracking
- Chasing
- Evasion
- Patrolling
- Hiding and Taking Cover
- Shooting
- Putting It All Together
- In Closing

"Action may not always bring happiness; but there is no happiness without action."

Benjamin Disraeli

Now that we have a global understanding of the building blocks of any AI system, it is time to delve into the details of how AI is built into video games. Because this is a lengthy subject, with different techniques used for different gameplay styles, I have divided the information into three separate chapters. Most game AI techniques might not be very complex, but there are many twists and interesting ideas to which we will have to devote some space.

In this chapter, I will provide an overview of AI methods used in fast-paced action games. We will review general techniques, and then do a case-by-case analysis on fighting games, racing simulators, and so on, providing insider information on many popular algorithms.

The next chapter will deal with the subject of tactical AI, which finds its main use in strategy games. We will learn to build plans, analyze enemy configurations, and trace maneuvers that would marvel most real-world generals.

We will end our journey through artificial intelligence techniques with a chapter on scripting, which is one of the most powerful paradigms for coding AIs. By separating the AI code from the main game engine, scripting provides a

robust and flexible way of creating large AI systems. In fact, most professional AIs today are built via some sort of scripting engine. So we will analyze the different scripting techniques in detail.

Let's begin with the basics. We need to create little creatures that chase us and shoot, so let's get to work.

On Action Games

For this book, I will define action as intelligent activity that involves changes of behavior at a fast speed. Examples of action are all locomotion behaviors (a character that runs in *Mario*), simple aggression or defense (enemies shooting or ducking in *Quake*), and so on. Notice how action is put in a contraposition to tactical reasoning, which is in turn described as the analysis process used to create a plan that will then guide the actions of an intelligent character. So, action deals with immediate activity, and tactics plan that activity.

Action is thus driven by relatively simple tests. We will need to compute distances to targets, angular separations, and so on. Action is also quite fast-paced. Action games have higher rhythms than tactic/strategic games.

Choreographed AIs

In its simplest form, an action AI system can be implemented as a preprogrammed action sequence that is executed repeatedly. This form of AI is used in industrial robots and can be applied to simple games as well. An elevator in *Quake*, for example, is just executing a very simple movement sequence. These systems are merely state machines with no optional transitions—just a series of states the automata loops through endlessly.

But we can use the same technique for quite interesting AI entities, well beyond elevators and robot arms. A security guard in an adventure game, for example, can be programmed to walk up and down the alleys in a prison, maybe performing some specific actions. Ships in top-down arcade scrollers sometimes exhibit quite beautiful

choreographies, often involving tens of ships. Gameplay in these games emerges from analyzing movement patterns and detecting the pitfall that allows us to avoid the guard, shoot down enemy ships, and so on.

Choreographed AI systems usually employ a very simple scripting engine to represent movement sequences. Figure 7.1 shows a diagram of such a system. Note that we are not talking about a full-blown AI system built with a scripting engine. We are starting with very simple and deterministic behaviors that are stored in files, as in the following example:

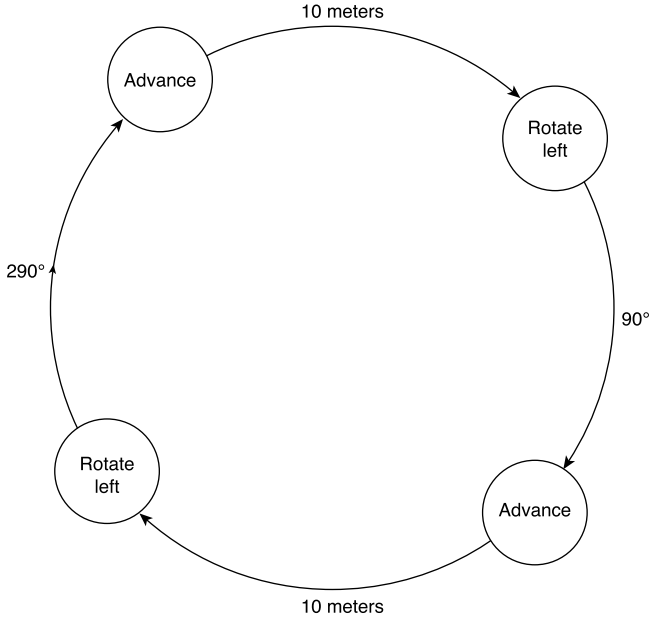

Figure 7.1 Sequencer for a choreographed AI system.

```
set animation "walk"
go to 0 10 0
set animation "rotate"
rotate 180
set animation "walk"
go to 0 0 0
set animation "rotate"
rotate 180
```

The preceding example could very well be an animation for a security guard in a prison. He just walks up and down an alley, executing different animation cycles. Most coin-operated machines from the 70s and 80s used variations of this approach, shared (among other classics) by the LOGO programming language and the *BigTrak* vehicle toy sold by Milton Bradley.

The main design decision you must make when coding any choreographed AI system is selecting the range of commands the AI will respond to, or in other words, the instruction set for your programming language. I recommend you take a second to carefully think about the level of abstraction you will use with your AI system. Should any movement be decomposed into a series of straight lines, or should more complex tasks like spirals and circles be directly built into the language? Clearly, setting the right abstraction level will allow us to create more interesting scripts with less work. Unfortunately, this is a context-sensitive decision. Designing the scripting language largely depends on the kind of game (and AI) you are trying to craft. Once the level is set, decide on the instructions and parameters you will be implementing. Again, try to think in terms of what gives you the most expressive potential for the game.

Implementation

The first choreographed AIs stored sequences that were hard-coded into the game's assembly source. But as time passed and sequencing languages became more and more powerful, small scripting languages were evolved, so content creators could work on more complex sequences easily.

Today, implementing a choreographed AI system is, fundamentally, implementing a very simple script language. You need a parser that reads the script into main memory. Then, a run-time interpreter processes the file and executes its instructions, affecting game engine internals such as enemy positions and orientations. We will discuss script languages in detail in Chapter 9, "Scripting," so I suggest you refer to it when trying to code a choreographed AI system.

Object Tracking

One of the first problems we must deal with in any action AI system is maintaining eye contact with a target, whether it's moving or static. This is used everywhere in action-oriented games, from rotating a static shooting device (like the turrets in *Star Wars*' trenches) to aiming at an enemy we are chasing or evading, or basic navigation (to track a waypoint we must reach).

Eye contact can be formulated as, given an orientation (both position and orientation angles) and a point in space, computing the best rotation to align the orientation with the point. It can be solved in a variety of ways, which I'll cover one by one in the following sections.

Eye Contact: 2D Hemiplane Test

If we are in a 2D world, we can solve the eye contact problem easily with very little math. Let's first take a look at the variables involved in our system. I will assume we are in a top-down view, overseeing the X,Z plane.

```
point mypos;              // position of the AI
float myyaw;              // yaw angle in radians. I assume
                          top-down view

point hispos;             // position of the moving target
```

The first step is to compute whether `hispos` effectively lies to the left or right of the line formed by `mypos` and `myyaw`. Using parametric equations, we know the line can be represented as:

```
X = mypos.x + cos(myyaw) * t
Z = mypos.z + sin(myyaw) * t
```

where *t* is the parameter we must vary to find points that lie on the line. Solving for *t*, we get the implicit version of the preceding equation, which is as follows:

```
(X - mypos.x)/cos(myyaw) = (Z - mypos.z)/sin(myyaw)
```

By making this into a function, we know the points on the line are

```
F(X,Z)= (X - mypos.x)/cos(myyaw) - (Z - mypos.z)/sin(myyaw) = 0
```

If we have a point in space and test it against our newly created function, we will have

`F(X,Z) > 0` (it lies to one side of the line)

`F(X,Z) = 0` (it lies exactly on the line)

`F(X,Z) < 0` (it lies to the opposite side)

This test, which is illustrated in Figure 7.2, requires:

- 3 subtractions
- 2 divisions
- 1 comparison (to extract the result)

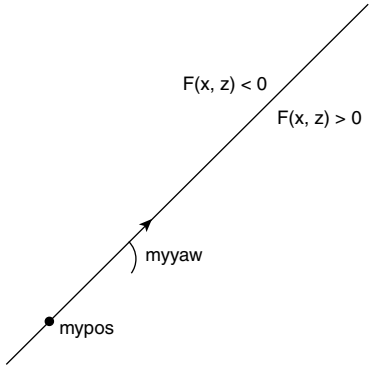

Figure 7.2 Hemispace test.

We can speed the routine up if we need to do batches of tests (against several targets) by calculating the expensive trigonometric functions only once, and storing them as the inverse so we can save the expensive divides. The formula will be

```
F(X,Z)= (X - mypos.x)*(1/cos(myyaw)) - (Z - mypos.z)*(1/sin(myyaw))
```

If we do these optimizations, the overall performance of the test will be (for *N* computations):

3*N subtractions

2*N multiplies

2 trigonometric evaluations

2 divisions (for the invert)

N comparisons (for the result)

which is fairly efficient. For sufficiently large batches, the cost of divisions and trigonometric evaluations (which can be tabulated anyway) will be negligible, yielding a cost of three subtractions and two multiplies.

For completeness, here is the C code for the preceding test:

```
int whichside(point pos, float yaw, point hispos)
// returns -1 for left, 0 for aligned, and 1 for right
{
float c=cos(yaw);
float s=sin(yaw);
if (c==0) c=0.001;
if (s==0) s=0.001;
float func=(pos.x-hispos.x)/c - (pos.z-hispos.z)/s;
if (func>0) return 1;
if (func==0) return 0;
if (func<0) return -1;
}
```

3D Version: Semispaces

To compute a 3D version of the tracking code, we need to work with the pitch and yaw angles to ensure that both our elevation and targeting are okay. Think of a turret in the *Star Wars* Death Star sequence. The turret can rotate around the vertical axis (yaw) and also aim up and down (changing its pitch). For this first version, we will work with the equations of a unit sphere, as denoted by:

```
x = cos(pitch) cos(yaw)
y = sin(pitch)
z = cos(pitch) sin(yaw)
```

Clearly, the best option in this case is to use two planes to detect both the left-right and the above-below test. One plane will divide the world into two halves vertically. This plane is built as follows:

```
point pos=playerpos;
point fwd(cos(yaw),0,sin(yaw));
fwd=fwd+playerpos;
point up(0,1,0);
up=up+playerpos;
plane vertplane(pos, fwd, up);
```

Notice how we are defining the plane by three passing points. Thus, the plane is vertical, and its normal is pointing to the left side of the world, as seen from the local player position.

The second plane is built with the pitch and divides the world into those points above and below the current aiming position. Here is the code:

```
point pos=playerpos;
point fwd(cos(pitch)*cos(yaw), sin(pitch), cos(pitch) sin(yaw));
fwd=fwd+playerpos;
point left(cos(yaw+PI/2),0,sin(yaw+PI/2));
left=left+playerpos;
plane horzplane(pos,fwd,left);
```

In this case, the normal points up. Then, all we need to do to keep eye space with a point in 3D space is compute the quadrant it's located at and react accordingly:

```
if (vertplane.eval(target)>0) yaw-=0.01;
else yaw+=0.01;

if (horzplane.eval(target)>0) pitch-=0.01;
else pitch+=0.01;
```

Notice that the signs largely depend on how you define pitch and yaw to be aligned. Take a look at Figure 7.3 for a visual explanation of the preceding algorithm.

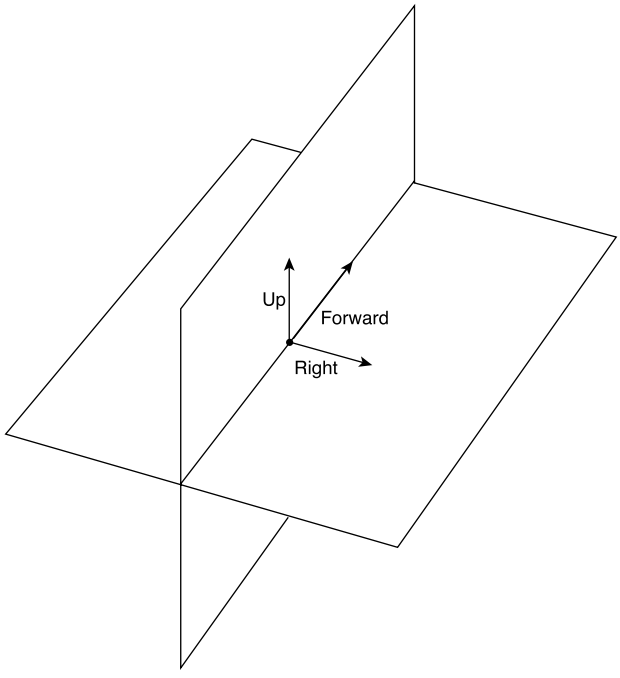

Figure 7.3 Semispace test, 3D version.

Chasing

Now that we know how to aim at a target, we will use that knowledge to implement a chase behavior. To be honest, chasing is easy once you know how to keep eye contact, so this section should be pretty straightforward.

In its simplest form, chasing involves moving forward while keeping eye contact with the target. Thus, we will keep aligned with the target and advance toward it. If by some unknown reason we lose sight of the target (because the target moved, for example), we will correct our orientation and keep moving.

Chase 2D: Constant Speed

The source code for a constant-speed, 2D chaser follows. Notice how we follow the guideline in the previous section, and just try to re-aim while moving forward. More sophisticated, variable-speed methods can be devised for specific purposes.

```
void chase(point mypos, float myyaw, point hispos)
{
reaim(mypos,myyaw,hispos);

mypos.x = mypos.x + cos(myyaw) * speed;
mypos.z = mypos.z + sin(myyaw) * speed;
}
```

The success of this approach largely depends on the relationship between our speed, the target's speed, and our turning ability. If we can turn quickly, we can assume that we will be effectively facing the target much of the time, so we will perform a pretty optimal chase. But we will need our speed to be higher than the target's speed to ensure we make contact. A faster target will thus be unreachable. On the other hand, a much slower target might also escape our pursuit, especially if our maneuverability is restricted. To understand this, think of a fighter jet trying to chase a helicopter. Quite likely, the jet will have a hard time trying to stay aligned with the helicopter.

Predictive Chasing

One alternative to ensure a better chase is to use predictive techniques. Here we will not aim at the target directly, but try to anticipate his movements and guess his intentions. In much the same way as a good chess player can use intuition to discover what his opponent is trying to do, a chaser can use some clever interpolation to anticipate his opponent's moves.

This idea is really straightforward. Keep track of the position history of the opponent and use that information to create a "predicted position" some time in the future. Then, aim at that position. In Figure 7.4, you can see how a predictive chaser can outperform an opponent, even if their respective speeds and maneuverability are the same. The predictive chaser will make more informed decisions, which will ultimately make him succeed.

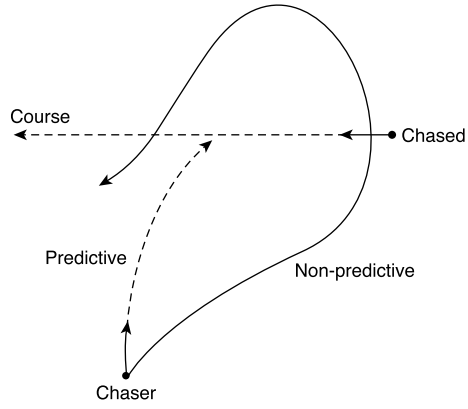

Figure 7.4 Predictive chasing, where the dotted line shows the predictive course.

Predictive chasing is performed as a preprocess to the chase routine just explained. Instead of aiming and advancing, we will use a three-step approach, which involves:

1. Calculating a projected position
2. Aiming at that position
3. Advancing

Calculating the projected position can be implemented with a number of interpolation techniques. We could, for example, take the last two position samples from the enemy and use a straight line as the interpolator. The code for this approach would look something like this:

```
void chase(point mypos, float myyaw, point hispos,point prevpos)
{
point vec=hispos-prevpos;       // vec is the 1-frame position difference
vec=vec*N;                      // we project N frames into the
                                future
point futurepos=hispos+vec;     // and build the future projection

reaim(mypos,myyaw,futurepos);
```

```
mypos.x = mypos.x + cos(myyaw) * speed;
mypos.z = mypos.z + sin(myyaw) * speed;
}
```

Just these three simple lines allow our chaser to perform better. By varying the value of *N* (which depends on the relationships of speeds and our chaser's turning ability), we can find the perfect match.

An added value of the preceding code is that it will correctly treat the degenerate case of a target that holds still. It will predict that the target will remain unchanged.

We can implement variants of the preceding code using better interpolators. Instead of using the last two points in the trajectory, we could use *N* points for better results. By using *N* points, we can derive an *N*-1 interpolation polynomial. Obviously, as we increase the degree of the polynomial, the fitness to the trajectory will improve, and we will be able to make longer-lived predictions. But we will soon see that this improvement comes at a price. Computing higher-order polynomials has a computational cost that you might not be willing to pay.

One of the best ways to compute interpolation polynomials is to use the method of finite differences. These are a set of equations that define the *N*th degree polynomial (thus, passing through *N*+1 values). For a quadratic polynomial, the equation has the form:

```
P(x) = a0 + a1*(X-x0) + a2*(X-x0)*(X-x1)
```

where:

```
a0 = y1
a1 = (y1 - y0) / (x1-x0)
a2 = (((y2 - y1) / (x2-x1)) - ((y1-y0) / (x1-x0))) / (x2-x0)
```

You can guess the pattern that generates the *N*th degree equation by examining the way each extra parameter—a0, a1, and a2—value is generated.

Evasion

Once we know how to chase targets around, it is easy to learn to evade. Essentially, evading is the opposite of chasing. Instead of trying to decrease the distance to the target, we will try to maximize it. Thus, an evasion algorithm will be very similar to a chasing algorithm except for some sign changes. Here it is in detail:

```
void evade(point mypos, float myyaw, point hispos)
{
reaim(mypos,myyaw,hispos); negated

mypos.x = mypos.x + cos(myyaw) * speed;
mypos.z = mypos.z + sin(myyaw) * speed;
}
```

Patrolling

Another interesting behavior in any action AI system is patrolling. Policemen, dogs, and many other in-game characters patrol a predefined area, sometimes engaging in combat when an enemy is discovered.

To implement a patrolling behavior, we have to store a set of waypoints that will determine the path followed by the AI. These waypoints can be followed in two configurations: cyclic and ping-pong. Given waypoints W1…W5, here are the visit sequences for both schemes:

> Cyclic: W1 W2 W3 W4 W5 W1 W2 W3 W4 W5 …
>
> Ping-pong: W1 W2 W3 W4 W5 W4 W3 W2 W1 W2 …

I recommend using the cyclic approach because ping-pong trajectories can be expressed as cycles (explicitly adding the way-back), whereas the opposite is not true. Following a waypoint is not very different from a chase behavior. It is just a chase where we follow a sequence of targets. The easiest way to implement this is through a minimal, two-state finite-state machine. The first state is used to advance toward the next waypoint (represented internally by an integer). Then, as we approach closer to a predefined threshold, we move to the second state. This is a nonpersistent state, where

the integer used to represent the next waypoint is updated. We then go back to the first state, follow the waypoint, and so on.

Patrol behaviors (depicted in Figure 7.5) can often be enhanced by adding a third state, which implements a chase behavior. This is usually triggered by using a view cone for the AI, testing whether the player actually entered the cone (and was thus discovered by the AI). Games such as *Commandos: Behind Enemy Lines* made good use of this technique.

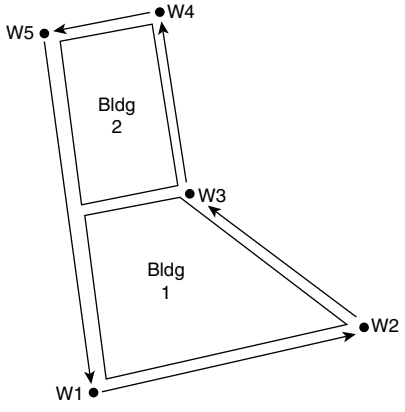

Figure 7.5 A patrolling AI, surrounding two buildings.

Hiding and Taking Cover

Sometimes we will need our AIs to run for cover or remain unnoticed by the player. For example, maybe they want to catch the player by surprise, or maybe the AI is controlling a soldier under heavy player fire. Whatever the case, knowing how (and where) to seek cover is always an interesting twist for the AI. Many game reviews praise this behavior as highly sophisticated, and players enjoy seeing it in action because it offers an extra challenge.

Taking cover is usually a matter of two actions performed sequentially. First, a good hiding spot must be located in the game level. Second, we must move there quickly. This second phase is nothing but a chase routine, very similar to those explained earlier in this chapter; so we will focus on detecting good hiding spots. To do so, we will need three data items:

- The position and orientation of the player
- Our position and orientation
- The geometry of the level

The geometry of the level must be stored in such a way that it allows proximity queries on objects that can actually become hiding spots. A good structure, for example, is having the world separated into terrain data with objects laid on top. Then, the objects must be laid out using quadtrees, spatial indices, or any other technique, so we can quickly compute which objects lie closest to certain map locations.

The actual algorithm involves finding the closest object to the AI's location and computing a hiding position for that object. The algorithm works as follows. We start by using the scene graph to select the closest object to the AI. This step largely depends on how the world is laid out. Once we have selected the object, we shoot one ray from the player's position to the barycenter of the object. We propagate the ray beyond that point, computing a point along the ray that's actually behind the object (from the player's standpoint). As shown in Figure 7.6, that's a good hiding spot. Keep in mind that hiding in places can be sped up if we can guarantee convex hiding spots.

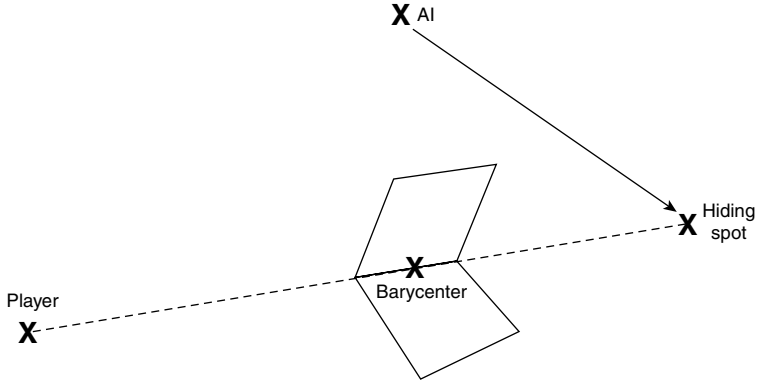

Figure 7.6 Geometry of playing hide-and-seek with the player.

AIs that implement hiding behaviors can easily be enhanced to work with the player's view cone. This is very useful for shooters. As shooters, we remain hidden while the view cone is focused on us. As soon as the player is looking somewhere else, we know we can leave our hiding spot and attack him. Games such as *Medal of Honor* have made great use of this technique, with German soldiers staying under cover until the right opportunity arises.

Shooting

We know how to chase the player, evade him, and keep an eye on his position. We have even predicted his future actions. So, now it's time to work on the mechanics of shooting. We need to learn when is it a good idea to shoot at the player in different contexts. We also need to know whether we are handling a machine gun, a sniper rifle, or a catapult. As you will soon see, each one requires slightly different approaches to the problem of targeting.

Before starting, I need to give some personal words of warning. The same way I enjoyed movies like *Star Wars*, *Alien*, and *Saving Private Ryan*, I think games with a fighting/shooting element should be recognized as enjoyable experiences. I don't have any moral problem with that, and I think any healthy person can differentiate between the fictitious violence shown by games/movies/books and real violence. On the other hand, I'd recommend that people play many different games, not all of them with a combat/violence component, just as I'd go to see different types of movies. That said, the following sections deal with the specifics of shooting, so they necessarily focus on the algorithms required to target and shoot down enemies.

Infinite-Speed Targeting

The first approach we will explore is shooting with a weapon that has infinite speed, or in practical terms, very high speed compared to the speed of the target. This can be the case of a laser gun, which would advance at light speed, for example. Then, we can assume the time it takes for the projectile to reach the target is virtually zero. Thus, the selection of the shooting moment is really easy. All you have to do is make sure you are well aligned with the target at the moment of shooting. As the velocity is very high, we will have a sure hit because the target will have very little time to move and avoid the collision with the bullet. Clearly, it is not a good idea to abuse infinite-speed weapons because they can unbalance your game. If you build these weapons into the game, make sure you balance them well in terms of gameplay. For example, the firing rate can be very low, the ammunition limited, or the weapon might be really hard to get.

Real-World Targeting

What happens with a real-world firing device? Even a real gun shoots projectiles at a limited speed (approximately 300-600 meters per second). This means shooting a fast moving target is harder than shooting one that stands still. Thus, most weapons must be modeled as finite-speed devices, where some careful planning is used. I will explain two popular approaches.

Version A: The Still Shooter

The still shooter targets the enemy and only shoots whenever the enemy is standing still for a certain period of time. The reason is simple. If the bullet takes one second to hit the target, and the target has been standing still for a certain period of time, it is a good hypothesis to assume the target will stand still for another second, thus making it a good moment to attempt shooting.

An enhancement to this algorithm is to watch the target for specific actions that indicate restrictions in his ability to move. For example, if the target is standing still, he might begin walking in any given moment, thus making it an unsafe target. But what happens if he sits down or if he is tying one of his shoes? Clearly, we have a better aim here because we know for sure he won't be going anywhere in the next few seconds. This would be the kind of reasoning that would drive a sniper-style AI. He looks for very safe shoots that hit the target most of the time. By shooting only when a safe hit is granted, the shooter ensures one kill while not giving away his position easily. The disadvantage is that maybe the shooter will have very few opportunities to actually shoot, so it is a good idea to make him less restrictive. The way to do this is to introduce errors in his processing. He might sense time incorrectly, confuse animations, and so on. So sometimes he will shoot when he's not supposed to. When done carefully, this can accurately model fatigue and morale, affecting the ability of the sniper to stay focused.

As a summary, here is the algorithm in detail:

```
Global variables:
Timestill             time since the enemy began standing still
StandingStill         1 if standing still, 0 otherwise

When it begins standing still
```

```
       StandingStill=1
       Timestill=now

If StandingStill and more than X seconds have elapsed since
Timestill
       Shoot
```

Version B: The Tracker

The tracker AI also tries to model the behavior of a sniper. In this case, he will shoot moving targets, not just those who are standing still. Shooting a moving target is really hard. We need to combine the shooting behavior with a target tracking routine, and there is a predictive component going on as well. If the gun has a finite speed, we need to target not the current position, but the position where the target will be when the bullet hits him.

The idea is simple: Compute the distance from the sniper to the target, use the projectile velocity to compute how long it will take for the projectile to reach the target, and predict where the target will be in the future, exactly when the projectile arrives. This way you can aim at that spot and get a safer shoot, especially in distant or fast-moving targets. The algorithm in full is depicted in Figure 7.7.

```
float d=distance (sniper, target)
float time=d/bulletspeed
point pos=predictposition(target,time)
if aiming at pos shoot()
else target at pos;
```

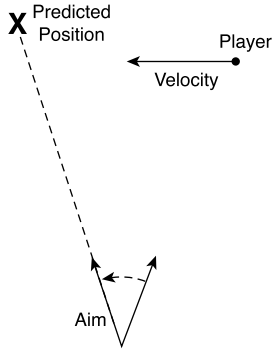

Figure 7.7 Predictive shooter.

Whether predictive or still shooters, we have focused so far on single-shot firing devices, where each shot is considered an individual AI decision. But other weapons, such as machine guns, offer the possibility of shooting bursts of bullets at high frequency but with reduced precision. The AI logic for such weapons is a completely different subject, and thus deserves its own separate discussion.

Machine Guns

Machine guns offer fast firing rates at the cost of inferior precision. Shots cause the cannon to shake due to recoil, making it hard to aim accurately. Thus, their main use is found not in targeting people, but areas. The machine gun is aimed in the right direction, and short bursts are fired to hit anyone in the area.

The first type of gun we will examine is the fixed machine gun. This kind of behavior is exhibited by gunners in bunkers, trenches, and so on. Some classic guns would be the MG-42 used by the German army in World War II, the M60 used in Vietnam, and so on. Here are some stats from the former:

MG-42 (with lightweight tripod)

- Firing rate: 25 rounds per second
- Range: 1000 meters
- Muzzle velocity: 820 meters per second
- Weight: 11.6 Kg

MG-42 (with Lafette tripod)

- Firing rate: 25 rounds per second
- Range: 1000 meters
- Muzzle velocity: 820 meters per second
- Weight: 31.1 Kg

From these statistics, several lessons can be extracted. First, these guns hardly ever moved, but instead kept on targeting and shooting down enemies from a fixed position. Second, these guns did not have a lot of autonomy, the standard feed type for the MG-42 was a 50/250 metal belt. Thus, a burst could not last longer than 10 seconds,

followed by a pause to change the metal belt. These guns were thus used for performing short firing bursts. Their algorithm is relatively straightforward. By default, the soldier stands still, waiting for new enemies to arrive. Then, as they begin to get closer, the gunner must rotate the gun to face the enemy. Rotation must somehow be penalized for slower models. When the angular difference between the gunner and the enemy is smaller than a certain threshold, the gunner will hold down the trigger while trying to refine his aiming. Keep in mind each shot introduces some distortion to the aiming due to recoil, so the gunner must re-aim every time. As a result, fixed gunners do not usually aim carefully; they aim at an area. Thus, these gunners are especially useful when we need to stop a wave composed of many soldiers. By pure chance, some bullets shot by the gunner will reach their target.

A common mistake is to forget about feed sizes. Many World War II games display machine guns that seem to have infinite ammunition.

Let's now examine the problem of a moving character carrying a light machine gun, such as an AK-47 or an M-16. As a rule of thumb, only movie characters use moving machine guns to shoot long bursts. Recoil makes it impossible to aim, especially if standing up. So, ammunition is wasted because most projectiles will be lost. Besides, these guns do not have long cartridges, so ammunition must be used with care. Here are some stats from the World War II Thompson submachine gun, aka the "Tommy gun":

Thompson

- Firing rate: 10–12 rounds per second
- Range: 50 meters
- Muzzle velocity: approximately 400 meters per second
- Weight: 5 Kg

The gun came with 30 bullet cartridges, and a soldier in World War II usually carried three such magazines. As you can see, ammunition was still more of an issue than with heavy, fixed machine guns. Thus, the most common practice is to treat these assault guns as rifles with very high firing rates. Bullets are shot one by one or in very short bursts. The only situation where a moving gunner can effectively waste ammo is in a fantasy setting, such as space ship games. Here we can forget about realism and make the tie fighter or other ship of your choice shoot long firing bursts.

Putting It All Together

We have seen the basic routines to move a character, chase other people, and shoot different weapons. But these are all individual tasks, which must somehow be combined. To do so, I will now examine two techniques that allow us to blend these simple behaviors into a rich, engaging AI system: parallel automata and AI synchronization. The techniques we have seen so far will thus behave like bricks in a *LEGO* game, each serving a specific function or task.

Parallel Automata

The first way to blend different actions together is to use a parallel automata. In this case, the first automata would control locomotion (chasing, evading, patrolling, hiding), whereas a second automata would evaluate the firing opportunities. The algorithm for this solution would be best implemented via state machines (thus, two state variables would be required). You can see this in the following example:

```
class enemy
       {
       int locomstate,gunstate;
       void recalc();
       };


void enemy::recalc()
{
switch (locomstate)
       {
       state IDLE:
       state WALKING:
       }
switch (gunstate)
       {
       state IDLE:
       state WALKING:
       }
}
```

We have thus divided the problem in two. Now, the locomotion AI is in charge of tasks such as reaching waypoints, collision detection, seeking cover as needed, and so on. Then, the gunner AI takes care of targeting and shooting down enemies.

AI Synchronization

Another way of combining simple behaviors into complex systems is to make use of AI synchronization. This is generally considered an advanced topic, but adding it to our toolbox of action AI techniques can greatly increase the expressive potential of the overall system—the same way a group of ants looks more intelligent to the observer than an individual ant. Groups of enemies that coordinate, implement tactics, and work as a team are one of the most impressive features of any AI system. This technique was made popular by *Half-Life*, where enemy soldiers would call for help, cover each other, and operate as a squad realistically.

Implementing enemy synchronization is just a matter of using a shared memory pool, which is visible to all entities and can be written and read by the AIs. Then, the rule systems or finite-state machines must be enhanced to take advantage of this shared memory.

At the simplest level, our shared memory pool can be something like this:

```
typedef struct
       {
       bool flags[64];
       } sharedmemory;
```

We need this structure to be visible from all automata, so we can add sentences like this:

```
if (sharedmemory[3]) state=(…)
```

As an example, imagine a game that takes place in a prison. The center of the prison is dominated by a large tower with a powerful light cannon, which continually scans the ground at night to prevent inmates from escaping. A simple AI controls the light. Then, there are a few patrol AIs that walk around the complex, and if the player enters their view cone, they chase him and kill him. Now, imagine that the watchtower AI

uses a shared memory location to indicate whether the player is actually inside the light cone. If this is so, another memory location stores his position. Then, patrol AIs are implemented just like we discussed earlier, but with an exception. They read the said memory location, and if the player is detected, they run to the location specified by the watchtower. This very simple synchronization, illustrated in Figure 7.8, can yield a significant improvement in the gameplay. The player can voluntarily enter the light cone to be detected, then run away, so guards are sent to an area of the camp while he escapes somewhere else.

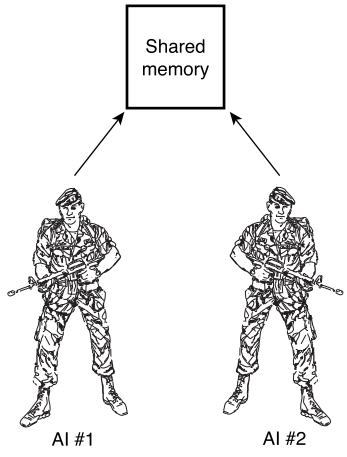

Figure 7.8 Two AIs use the shared memory to communicate.

More complex synchronization mechanisms can be devised. A soldier can confront the player, and if the player is carrying a more powerful weapon, the soldier can run for cover. Once hidden, he can raise a signal so other AIs can come to help him. Add some radio messages as sound effects, and you have a very credible AI behavior.

Synchronization becomes more complex if we try to model larger groups involved in more sophisticated interactions. If we need to create lifelike behaviors involving dozens of entities, we better use artificial life techniques. These techniques use specific tools to convey the illusion of living entities acting in groups, such as troops, schools of fish, or birds in the sky.

In Closing

As a conclusion, I will provide some AI coding hints for a variety of action-based genres.

Platform Games

The first genre we will address is that of platform/jump'n'run games such as *Mario* or *Jak and Daxter*. In these games, AI is all about variety, having lots of enemies to keep the game fresh and new. On the other hand, these AIs are not very complex because encounters with AI enemies are quite short and simple. From the moment we see the baddie in such a game to the moment either he or we are dead, only a few seconds pass.

Platform game enemies can be easily coded using finite-state machines. In the simplest incarnation, they can be sequential/choreographed AIs that perform deterministic moves. Turtles in the first *Mario* game, for example, walked up and down map areas and killed on contact. Many old-school classics were built with this idea in mind.

One step higher in the complexity scale, we can implement chasers that get activated whenever the player is within a certain range. This represents the vast majority of AIs found in platformers, often hidden in great aesthetics that largely increase the believability of the character. Sometimes, these chasers will have the ability to shoot, which is implemented with parallel automata. One controls locomotion and the other just shoots whenever we are within angular reach.

Another interesting AI type is the turret, be it an actual turret that shoots, a poisonous plant that spits, or anything in between. A turret is just a fixed entity that rotates, and when aiming at the player, shoots at him. This behavior can be implemented with the eye contact approach explained at the beginning of the chapter.

Whichever AI type you choose, platform games require AIs to perform sequences that show off their personalities. The basic behavior will be one of the behaviors we have seen so far. But usually, we will code an AI system with some extra states, so the character somehow has a personality that is engaging to the player. I will provide two examples here.

In *Jak and Daxter: The Precursor Legacy*, there are gorillas that chase the player and kill on contact. They are triggered by distance. A gorilla stands still until we approach to within 10 meters. Then, he just walks toward us, and whenever he makes contact, tries to hit us. But this is just a plain chase behavior, which would make the game too linear and boring if that is all the gorilla did. Besides, we need a way to beat the gorillas—maybe a weak spot that makes them winnable. Thus, the gorilla expands his behavior to incorporate a short routine that involves hitting his chest with both fists, the same way real gorillas do. As a result, the gorilla chases us around, and whenever he's tired or a certain amount of time has elapsed, he stops, does the chest-hitting routine, and starts over. This makes sense because the gorilla is showing off his personality with this move. But this is also useful because we know we can attack the gorilla whenever he is hitting his chest. That's his weak spot. Many enemies in *Jak and Daxter* work the same way. They have a basic behavior that is not very different from the behaviors explained in this chapter and some special moves that convey their personalities to the player.

Another, more involved type of enemy is the end-of-level bosses. I'm thinking about usually large enemies that perform complex AI routines. Despite their spectacular size, these AIs are in fact not much different than your everyday chaser or patrolling grunt. The main difference is usually their ability to carry out complex, long-spanning choreographies. Although these routines can become an issue from a design standpoint, their implementation is nearly identical to that of the cases we have analyzed so far. As an example, consider the killing plant from *Jak and Daxter: The Precursor Legacy*. This is a large creature, about 5 meters tall, that tries to kill our hero by hitting him with its head. The flower is fixed to the ground, so it's not easy to avoid it. To make things more interesting, every now and then the flower will spawn several small spiders, which will become chasers. So you have to avoid the flower and keep an eye on the spiders while killing them. Then, the carnivorous flower will fall asleep every so often, so we can climb on it and hit it. By following this strategy repeatedly, we can finally beat it. Take a look at the summary presented in Figure 7.9.

As engaging and well designed as this may sound, all this complexity does not really affect our implementation. The boss is just a sequential automata that has a special ability of generating new creatures, but overall the implementation is really straightforward.

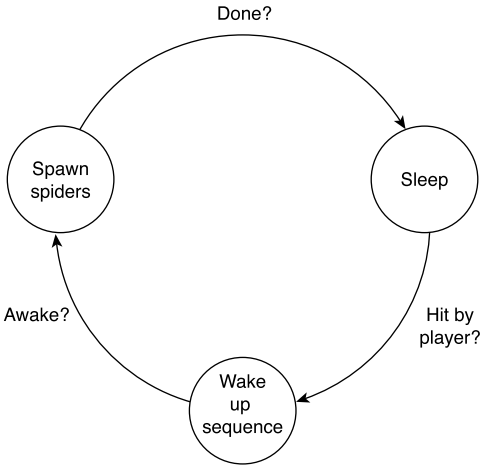

Figure 7.9 A classic boss from *Jak and Daxter: The Precursor Legacy*.

Shooters

We will now focus on shooters like *Half-Life* or *Goldeneye*. These games are a bit more complex than platformers because the illusion of realistic combat must be conveyed. The core behavior engine is usually built around finite state machines (FSMs): simple sequential algorithms that convey the attack and defend sequences. Also, the comment about aesthetics-driven AI in the previous section is also valid here. We need the character to convey his personality in the game.

Most shooters these days allow enemies to think in terms of the scenario and its map. Enemies can follow us around the game level, understand context-sensitive ideas like taking cover, and so on. Thus, we need a logical way to lay out the scenario. A popular approach is to use a graph structure with graph nodes for every room/zone and transitions for gates, doors, or openings between two rooms. This way you can take advantage of the graph exploration algorithm explained in the next chapter. We can use Dijkstra's algorithm to compute paths, we can use crash and turn (also in the next chapter) to ensure that we avoid simple objects such as columns, and so on.

Another interesting trend, which was started by *Half-Life*, is group behavior for games—being chased by the whole army and so on. Group dynamics are easily integrated into a game engine, and the result is really impressive. Clearly, it's one of those features in which what you get is much more than what you actually ordered.

Fighting Games

The algorithms used to code fighting games vary greatly from one case to another. From quasi-random action selection in older titles to today's sophisticated AIs and learning features, fighting AI has evolved spectacularly.

As an example, we could implement a state machine with seven states: attack, stand, retreat, advance, block, duck, and jump. When connected in a meaningful way, these states should create a rather realistic fighting behavior. All we need to do is compute the distance to the player and the move he's performing and decide which behavior to trigger. Adding a timer to the mix ensures the character does not stay in the same state forever.

If you want something more sophisticated, we can build a predictive AI, where the enemy learns to detect action sequences as he fights us. Here, the enemy would learn our favorite moves and adapt to them. The idea is quite straightforward: Keep a list with the chronological sequence of the last *N* player movements. This can be a circular queue, for example. Then, using standard statistics, compute the degree of independence from the events in the queue. For example, if the player is performing a kick and then a punch, we need to compute whether these two events are correlated. Thus, we compute the number of times they appear in sequence versus the number of times a kick is not followed by a punch. Tabulating these correlations, we can learn about the player's fighting patterns and adapt to them. For example, the next time he begins the kick sequence, we will know beforehand whether or not he's likely to punch afterward, so we can take appropriate countermeasures.

The finite-state machine plus correlations approach is very powerful. It is just a problem of adding states to the machine so we have more and more flexibility. If we need to create several fighters, each with its own style, all we need to do is slightly change the states or the correlation-finding routine to change the fighter's personality. Don't be overly optimistic, though, most games implement character personality at the purely aesthetic level, giving each character specific moves, a certain attitude in the animation, and so on.

So far, our fighter is highly reactive. He knows how to respond to attacks efficiently by learning our behavior patterns. It would be great, especially for higher-difficulty enemies, to make him proactive as well, making him capable of performing sophisticated tactics. To do so, the ideal technique would be to use space-state search. The idea would be to build a graph (not a very big one, indeed) with all the movement possibilities, chained in sequences. Then, by doing a limited depth search (say, three or four levels), we can get

the movement sequence that better suits our needs according to a heuristic. Then, the heuristic would be used to implement the personality. Although this looks like a good idea, executing the state search in real time at 60 frames per second (fps) can be an issue.

Thus, a simpler approach is to use a tabulated representation. We start with a table with simple attack moves and their associated damage. Then, every time we do a new attack combination, we store the damage performed and the distance at which we triggered it. For example, we can store:

Distance = 3 meters, high kick, jump, punch, Damage = 5

Then, we can use this list afterward, accessing it by distance and selecting the attack that maximizes damage. It's all just very simple pattern matching both for the attack and defense, but it produces the right look and feel.

Racing Titles

Racing games are easily implemented by means of rule-based systems. Generally speaking, most racing games are just track followers with additional rules to handle actions like advancing on a slower vehicle, blocking the way of another vehicle that tries to advance, and so on. A general framework of the rule set would be (starting with the higher priority rules):

If we are behind another vehicle and moving faster → advance

If we are in front of another vehicle and moving slower → block his way

Else → follow the track

The advance behavior can be implemented as a state machine or, even simpler, using field theory to model the repulsion that makes a car advance on another by passing it from the side. Here the cars would just be particles attracted to the center of the track, and each car we happen to find standing in our way would be repulsive particles.

The track follow is often coded using a prerecorded trajectory that traverses the track optimally. A plug-in is used to analyze the track and generate the ideal trajectory. Then, at runtime the drivers just try to follow that race pattern, using the higher-priority rules as needed.