Artificial Intelligence

Content

1. AI Basics

- The appearance of the behavior is the important part.

- When designing the AI for a game, you should design from the top down: Think about how you want your bots to appear, and then try to implement it.

- Many AI bots need to start with a good path-finding algorithm so that they know how to travel from any two points in a map. Without path finding, you could end up with annoying or distracting AI that makes bots get stuck next to walls or not be able to turn around a corner.

- One of your overall goals of your AI bots is to not be distracting. Little things such as bots moving around in unnatural patterns can be annoying to the user. Sometimes AI is noticeable only when it's bad.

- You also should try to give different characters different behaviors that match their appearance. For example, a human-shape creature would need to appear to make intelligent choices because users would expect a human-shape creature to do so; on the other hand, a spiderlike creature could follow simple patterns and still be believable.

AI Basics (2)

- Another idea is to make progressively smarter and more difficult enemies as the game moves along. Stay away from AI that easily becomes predictable so that the game play doesn't become too tedious as the game progresses.

- Also, always keep in mind the goal of AI in the first place: finding that perfect zone where enemies are not too smart and not too dumb, making the game fun and challenging at the same time. Finding the right balance is up to you.

2. Take Away Those Godlike Powers!

- You rarely want to give bots godlike powers like this. Instead, you can force certain situations in which the bot doesn't know where the player is.

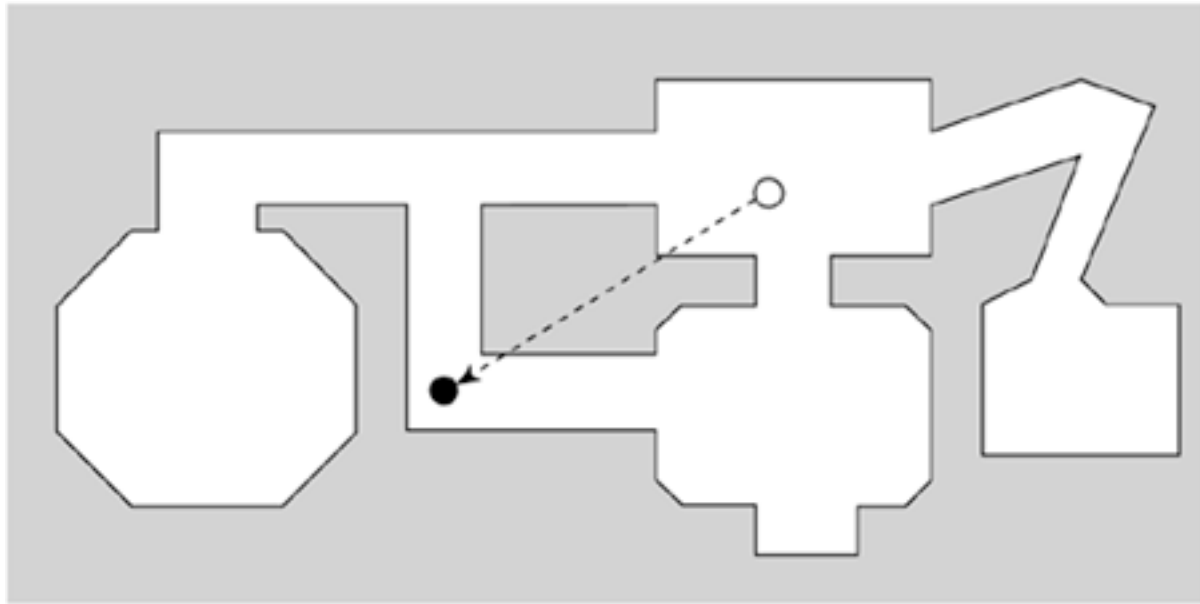
- One realistic idea that works wonders in a 3D game is to allow a bot to know where a player is only if the player is somewhere in the bot's sight. That is, the bot has artificial vision.

- **2.1. Seeing**

- Generally, the goal with artificial vision is to perform a check to find out whether a bot can see the player. Obviously, the bot shouldn't see anything that's behind an obstacle, such as something on the other side of a wall or above a ceiling.

- What you can do is fire a "vision ray" from the bot to the player to see if that ray hits any obstacles.

Take Away Those Godlike Powers! (2)



A bot (white circle) tries to locate the player (black circle). The bot uses a "vision ray" to determine whether it can "see" the player. In this case, the bot can't see the player.

- If the vision ray doesn't hit an obstacle, you can assume the bot "sees" the player and, therefore, knows the player's location. When the location is known, the bot can attack or whatnot. But how do you implement a vision ray?

Take Away Those Godlike Powers! (3)

- In a tile-based world, this is a simple problem to solve. Just draw a line using a standard line equation, and find all tiles that intersect that line. Then check each tile to determine whether it is "see-through."

- For a BSP tree, you've already implemented something like a vision ray. In Chapter 11, "Collision Detection" your collision-detection code had a method called **getFirstWallIntersection()** that found the first wall intersection between two points.

```
/**
    Checks if this object can see the specified object,
    (assuming this object has eyes in the back of its head).
*/
public boolean canSee(GameObject object) {
    // check if a line from this bot to the object
    // hits any walls
    boolean visible =
        (collisionDetection.getFirstWallIntersection(
        getX(), getZ(), object.getX(), object.getZ(),
        getY(), getY() + 1) == null);

    if (visible) {
        timeSincePlayerLastSeen = 0;
    }
    return visible;
}
```

AIBot.canSee()

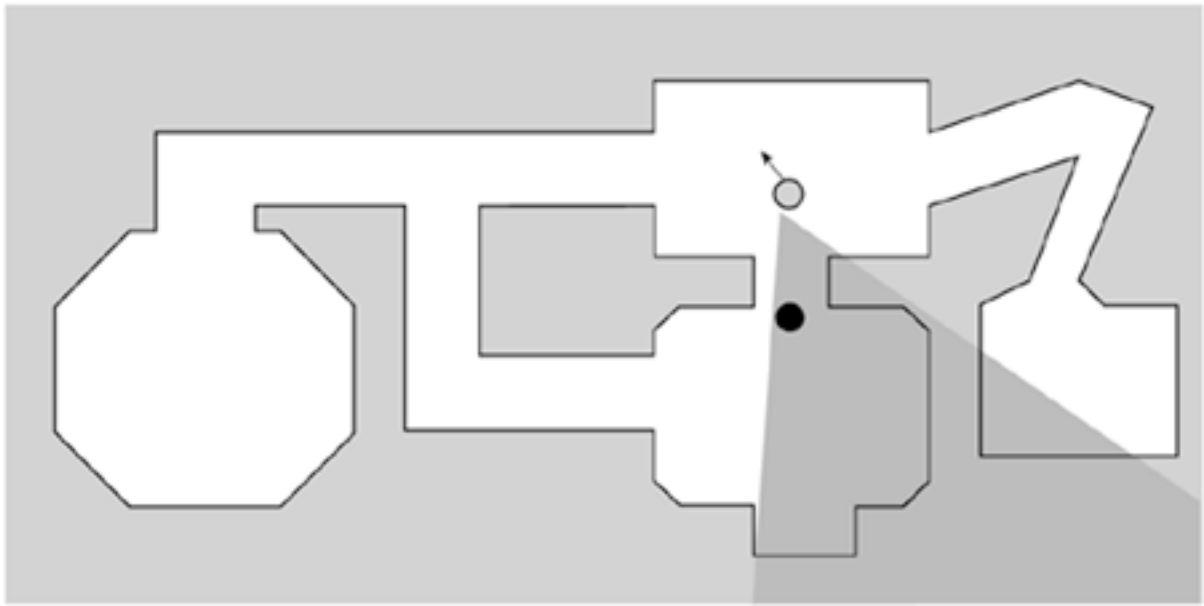
Take Away Those Godlike Powers! (4)

- Notice that you set the field **timeSincePlayerLastSeen** to 0. This variable is incremented in the **update()** method, so you always know the time since the player was last seen. This way, you can perform certain functions if the player is not visible but was visible recently, such as using the A* search if the player was visible just a few seconds ago.

- One problem with using **getFirstWallIntersection()** is basically a problem with using a 2D BSP tree: The ray is horizontal, which means the vision ray can't go upstairs. Ergo, bots can't see upstairs. Hopefully, with the combination of other AI capabilities, this won't be an issue.

- Another issue is that the bot has eyes in the back of its head. The vision ray is cast out in any direction, no matter which way the bot is facing, so the player can't sneak up behind the bot. To fix this, you could implement a "blind spot" behind the AI creature.

Take Away Those Godlike Powers! (5)



You can implement a vision "blind spot" so the player can still sneak up on the bot.

- This is fairly easy to do. Just check to make sure the angle between the vision ray and the direction the bot is facing isn't too large.

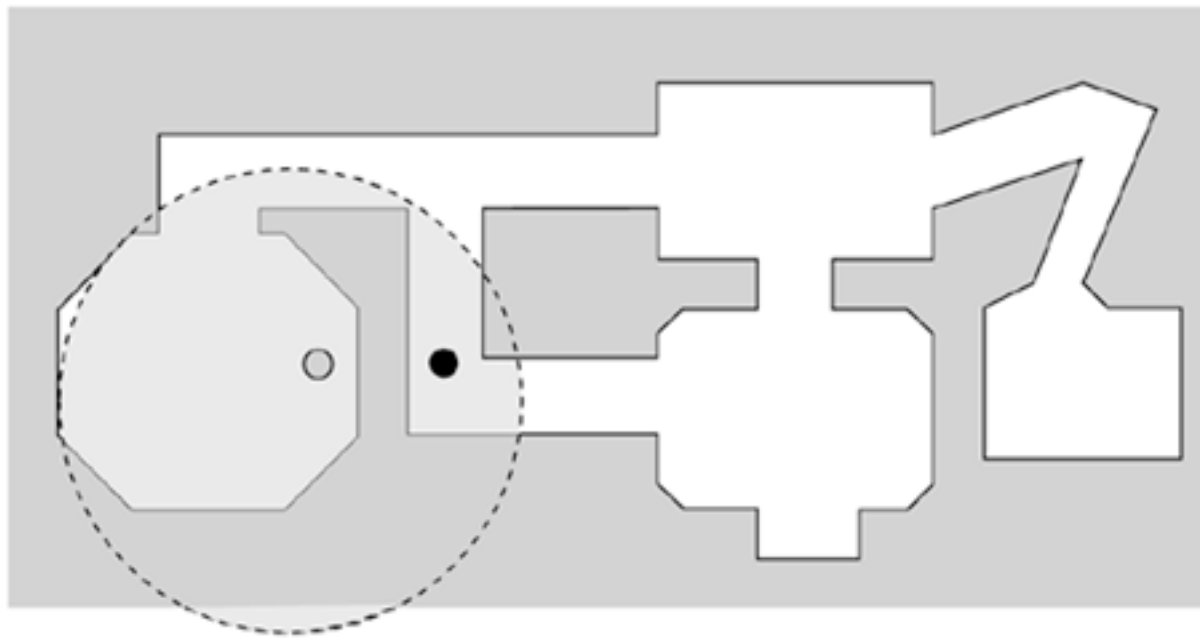
- But what if you snuck up behind a bot and just started screaming? Well, the bot couldn't hear, so it wouldn't matter. But what if the bot could hear you, such as when the player jumps, falls, or fires a weapon?

Take Away Those Godlike Powers! (6)

- **2.2. Hearing**

- In the real world, when you make a sound, the sound waves bounce off walls, run into each other, and travel down the hallway and around corners, and a sound gets quieter with distance until you're so far away from the sound source that you can't hear it at all.

- So in a game, you can give bots a "hearing radar" that can hear any noises within a certain distance, whether or not there are walls in the way, as in Figure 13.3.



Take Away Those Godlike Powers! (7)

- **2.2. Hearing**

```
float hearDistance;

...

/**
    Checks if this object can hear the specified object. The
    specified object must be making a noise and be within
    hearing range of this object.
*/
public boolean canHear(GameObject object) {

    // check if object is making noise and this bot is not deaf
    if (!object.isMakingNoise() || hearDistance == 0) {
        return false;
    }

    // check if this bot is close enough to hear the noise
    float distSq = getLocation().
         getDistanceSq(object.getLocation());
    float hearDistSq = hearDistance * hearDistance;
    return (distSq <= hearDistSq);
}
```

`AIBot.canHear()`

Take Away Those Godlike Powers! (8)

- Usually, a noise isn't just an "instant" event, but it actually lasts for a few seconds or more. So, you can keep track of the remaining time of the last noise made by each object. It's all summed up in a couple extra methods in the base **GameObject** class.

```
private long noiseDuration;

...

 /**
     Returns true if this object is making a "noise."
 */
 public boolean isMakingNoise() {
     return (noiseDuration > 0);
 }


 /**
     Signifies that this object is making a "noise" of the
     specified duration. This is useful to determine if
     one object can "hear" another.
 */
 public void makeNoise(long duration) {
     if (noiseDuration < duration) {
         noiseDuration = duration;
     }
 }


public void update(GameObject player, long elapsedTime) {
    if (isMakingNoise()) {
        noiseDuration-=elapsedTime;
    }
    ...
}
```

Take Away Those Godlike Powers! (9)

- In the code, we assume all noises have the same volume. In real life, however, loud noises travel farther than faint noises. You could extend this code to give each noise a volume or distance level that would specify either the volume of the sound or the maximum distance the sound can carry.

3. State Machines and Reacting

- In previous chapters, you've already taken advantage of **state machines**, in which an object can have different states that cause it to perform different actions. For example, the base game object class had the idle, active, and destroyed states. State machines are also called **finite** state machines because there are a limited number of possible states.

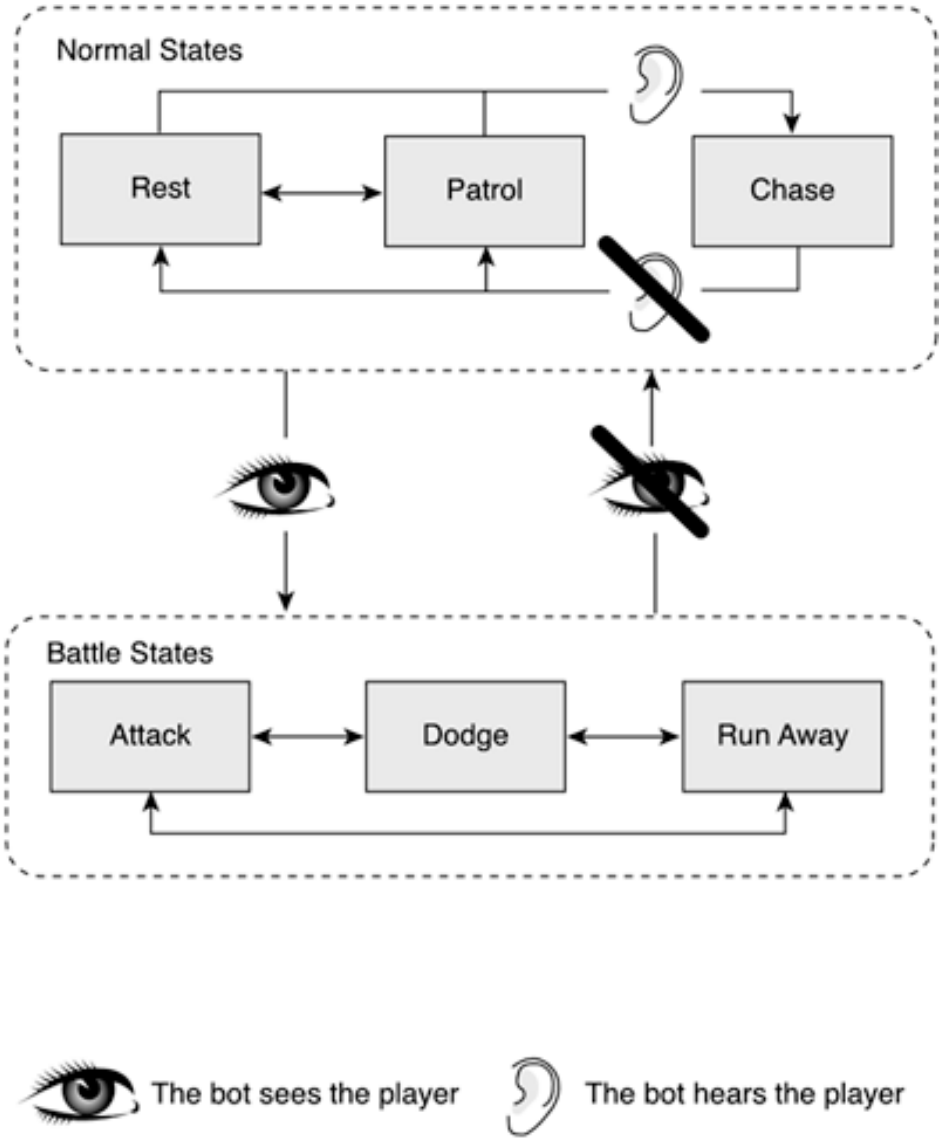
- A finite state machine also defines how and when the object's state can change. For example, a bot can go from stopped to walking to running, but it can't go directly from stopped to running. Also, a bot might start walking only if the player is visible.

- Additionally, you can create a hierarchy of states. For example, the bots in this chapter will have three states that are considered "battle" states: dodging, attacking, and running away. Furthermore, the attack state could have different kinds of attack patterns. So, the hierarchy is battle as a parent, with three children and possibly a few grandchildren.

State Machines and Reacting (2)

- Check out this figure for a visual look at the state machine you'll use for the AI bots in this chapter. This state machine has two basic parent states: *battle* and *normal*.

- A bot goes into a battle state only if the player is visible, and it returns to the normal state when the player is no longer seen.



State Machines and Reacting (3)

- Setting the correct AI state is straightforward, shown here in Listing 13.5.

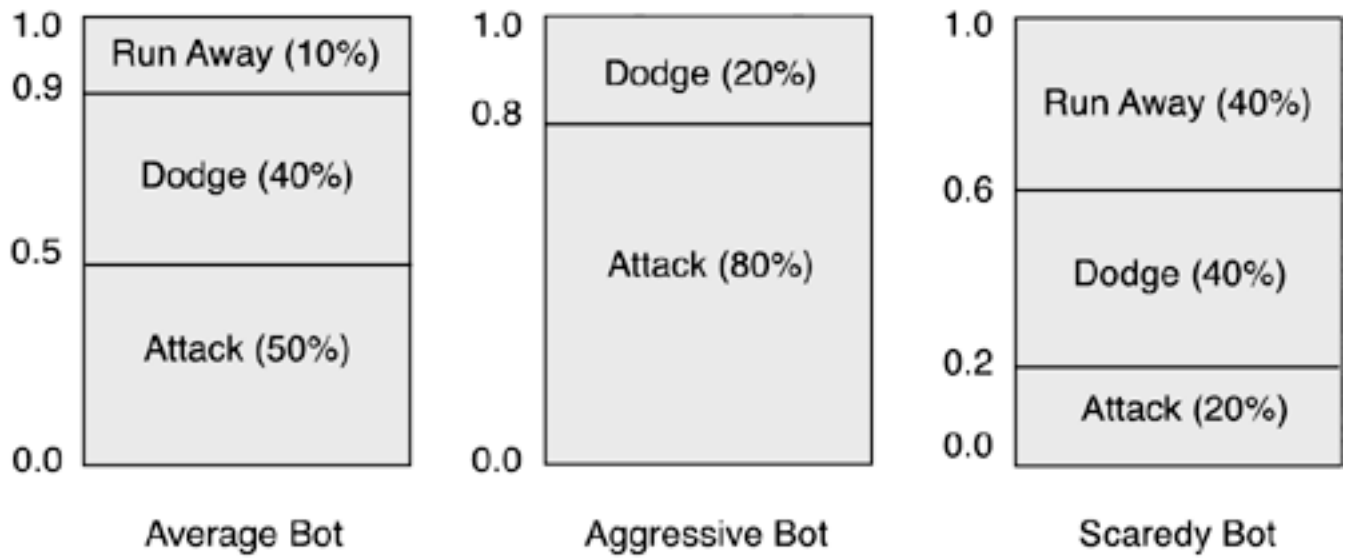
```
private int aiState;
private long elapsedTimeInState;

...

/**
    Sets the AI state for this bot (different from the
    GameObject state).
*/
protected void setAiState(int aiState, GameObject player) {
    if (this.aiState != aiState) {
        this.aiState = aiState;
        elapsedTimeInState = 0;
    }

    // later, set the appropriate path for this state here.
    ...
}


public void update(GameObject player, long elapsedTime) {
    elapsedTimeInState+=elapsedTime;
    ...
}
```

AIBot

4. Probability Machines

- Sometimes you want the decision between different states to occur purely on deterministic reasoning. For example, you might want to dodge only if you detect projectiles heading your way.

- Sometimes, though, you just want to randomly select one of these states, giving preference to one state or the other. Check out Figure 13.5 as an example.



Each bot could have a different probability of performing each battle state.

Probability Machines (2)

- This figure shows three different types of bots, with different probabilities of each state occurring. Note that the sum of the probabilities equals 100%.

- You might be used to observing probability as a percentage—for example, when you flip a coin, there's a 50% chance of the coin turning up heads. You also might have thought of probability as a fraction, such as a 1/6 chance of rolling a certain number with a die or a one-in-a-million chance of winning a lottery. In this chapter, we use the decimal representation of probability that's common in statistics textbooks. In this representation, the probability of an event occurring is between 0 and 1. A probability of 1 means the event always occurs, 0 means the event never occurs, and 0.5 means the event has a 50% chance of occurring.

- Figure 13.5 could easily have been a described as a pie chart, but laying it out in this way shows how you can determine which state is chosen. For example, a random number generator returns a number between 0 and 1. With an average bot, if the result is between 0 and 0.5, the state could be attack; between 0.5 and 0.9 would be dodge; and between 0.9 and 1.0 would be run away.

Probability Machines (3)

- You can keep track of these probabilities like this:

```
// probability of each battle state
// (the sum should be 1)
float attackProbability;
float dodgeProbability;
float runAwayProbability;
```

- Notice that the sum of these is assumed to be 1.0, just like in Figure 13.5—that way, at least one of these events will always occur. You can randomly choose a state as in Listing 13.6.

AIBot.chooseBattleState()

```
/**
    Randomly choose one of the three battle states (attack, dodge,
    or run away).
*/
public int chooseBattleState() {
    float p = (float)Math.random();
    if (p <= attackProbability) {
        return BATTLE_STATE_ATTACK;
    }
    else if (p <= attackProbability + dodgeProbability) {
        return BATTLE_STATE_DODGE;
    }
    else {
        return BATTLE_STATE_RUN_AWAY;
    }
}
```

Probability Machines (4)

- **Some Useful Random Functions**

- Using **Math.random()** in the **chooseBattleState()** method was fine because the random function returns a number from 0 to 1, but you're going to be using random numbers a lot on this chapter, and you'll often want more flexibility than just a range from 0 to 1.

- This is a good time to introduce some handy random functions. You'll add them to the **MoreMath** class, shown here in Listing 13.7.

```
/**
    Returns a random integer from 0 to max (inclusive).
*/
public static int random(int max) {
    return (int)Math.round(Math.random() * max);
}


/**
    Returns a random integer from min to max (inclusive).
*/
public static int random(int min, int max) {
    return min + random(max-min);
}


/**
    Returns a random float from 0 to max (inclusive).
*/
public static float random(float max) {
    return (float)Math.random()*max;
}


/**
    Returns a random float from min to max (inclusive).
*/
public static float random(float min, float max) {
    return min + random(max-min);
}


/**
    Returns a random object from a List.
*/
public static Object random(List list) {
    return list.get(random(list.size() - 1));
}


/**
    Returns true if a random "event" occurs. The specified
    value, p, is the probability (0 to 1) that the random
    "event" occurs.
*/
public static boolean chance(float p) {
    return (Math.random() <= p);
}
```

5. Making Decisions

- Making decisions is an easy part, but as we've talked about, it's also easy to make creatures too smart. Obviously, you don't want to make a new decision every frame, so your code should provide limits so that decisions are made on a periodic basis—say, every second or so. You can make "dumb" creatures have a longer decision period.

- How often the bots make a decision is just one of those things you'll have to tweak until you get the right feel.

- Listing 13.8 has some sample code you'll use for decision making in the AIBot. Here, a special state called **DECISION_READY** is used to signify that a decision needs to be made. The bot makes decisions every few seconds (based on decisionTime) but makes decisions more often if it's idle. In this manner, it can quickly move to other states if it notices the player. Decisions are also made when you're done with the path you're traveling on (from the code in the PathBot class).

Making Decisions (2)

- When making a decision, the bot chooses from a battle state, a chase state, or an idle state. If the player is visible, it picks a battle state. If the player was visible recently (in the last 3 seconds) or if the player is heard, the bot chooses the chase state. If none of those conditions are met, it chooses the idle state.

```
// time (milliseconds) between making decisions
long decisionTime;

...

public void update(GameObject player, long elapsedTime) {

    elapsedTimeSinceDecision+=elapsedTime;
    elapsedTimeInState+=elapsedTime;
    timeSincePlayerLastSeen+=elapsedTime;

    // if idle and player visible, make decision every 500 ms
    if ((aiState == NORMAL_STATE_IDLE ||
        aiState == NORMAL_STATE_PATROL) &&
        elapsedTimeInState >= 500)
    {
        aiState = DECESION_READY;
    }

    // if time's up, make decision
    else if (elapsedTimeSinceDecision >= decisionTime) {
        aiState = DECESION_READY;
    }

    // if done with current path, make decision
    else if (currentPath != null && !currentPath.hasNext() &&
        !getTransform().isMovingIgnoreY())
    {
        aiState = DECESION_READY;
    }

    // make a new decision
    if (aiState == DECESION_READY) {
        elapsedTimeSinceDecision = 0;

        if (canSee(player)) {
            setAiState(chooseBattleState(), player);
        }
        else if (timeSincePlayerLastSeen < 3000 ||
            canHear(player))
        {
            setAiState(NORMAL_STATE_CHASE, player);
        }
        else {
            setAiState(NORMAL_STATE_IDLE, player);
        }
    }
}
```

6. Patterns

- You'll use the **PathFinder** interface from the previous chapter to write some patterns. This interface has methods that return an Iterator of locations in a path to follow. You'll actually create an abstract **AIPattern** class, in Listing 13.9, that implements this interface.

- The AIPattern class has a couple of convenience methods to help when making patterns.

AIPattern

```
/**
    Calculates the floor for the location specified. If
    the floor cannot be determined, the specified default
    value is used.
*/
protected void calcFloorHeight(Vector3D v, float defaultY) {
   BSPTree.Leaf leaf = bspTree.getLeaf(v.x, v.z);
   if (leaf == null || leaf.floorHeight == Float.MIN_VALUE) {
       v.y = defaultY;
   }
   else {
       v.y = leaf.floorHeight;
   }
}


/**
    Gets the location between the player and the bot
    that is the specified distance away from the player.
*/
protected Vector3D getLocationFromPlayer(GameObject bot,
    GameObject player, float desiredDistSq)
{
    // get actual distance (squared)
    float distSq = bot.getLocation().
        getDistanceSq(player.getLocation());

    // if within 5 units, we're close enough
    if (Math.abs(desiredDistSq - distSq) < 25) {
        return new Vector3D(bot.getLocation());
    }

    // calculate vector to player from the bot
    Vector3D goal = new Vector3D(bot.getLocation());
    goal.subtract(player.getLocation());

    // find the goal distance from the player
    goal.multiply((float)Math.sqrt(desiredDistSq / distSq));

    goal.add(player.getLocation());
    calcFloorHeight(goal, bot.getFloorHeight());

    return goal;
}
```

Patterns (2)

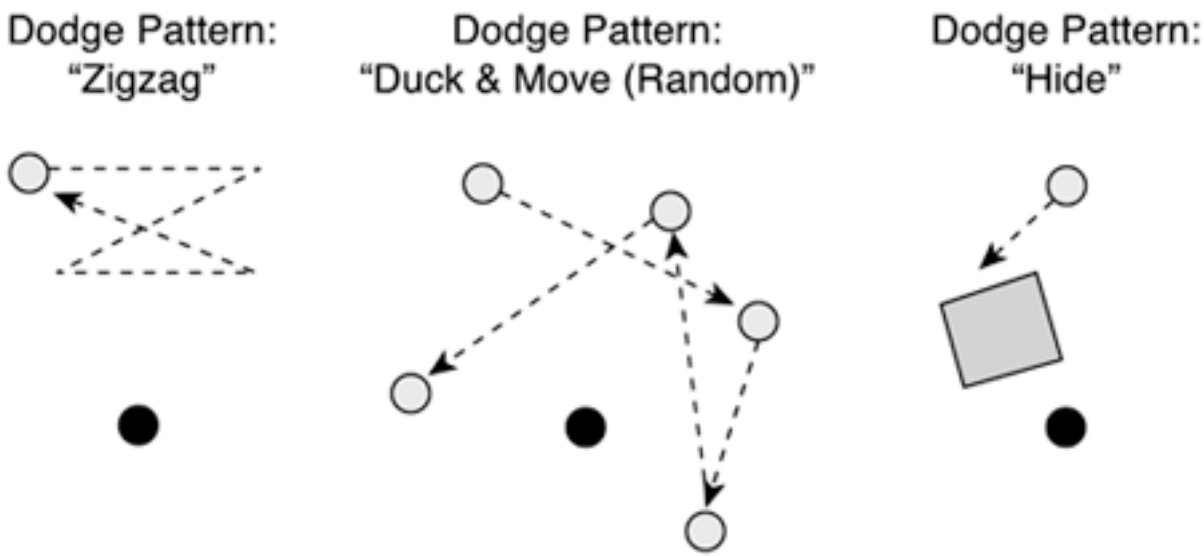
- The first convenience method, **calcFloorHeight()**, gets the height of the floor for any location. If the location is out of bounds, the default height is used.

- The second convenience method helps make several patterns easier to write. It returns the location between the bot and the player that is a specific distance away from the player. A lot of times, a bot wants to attack or dodge from a certain distance.

Patterns (3)

- **6.1. Dodging**

- The first pattern we discuss is dodging. Dodging can be very complicated or very simple, depending on what you want to accomplish. Check out Figure 13.6 for some sample dodge patterns.



Patterns (4)

- Let's create the first two dodge patterns as an example. The zigzag pattern is shown here in Listing 13.10.

- This zigzag pattern makes a pattern that moves the bot a certain distance (dodgeDist) at a tangent to the player and then back to the bot's starting point.

- There's a random chance the bot moves either left or right.

DodgePatternZigZag.java

```
public class DodgePatternZigZag extends AIPattern {

    private float dodgeDist;

    public DodgePatternZigZag(BSPTree tree, float dodgeDist) {
        super(tree);
        this.dodgeDist = dodgeDist;
    }

    public Iterator find(GameObject bot, GameObject player) {

        // create the vector to the dodge location
        Vector3D zig = new Vector3D(bot.getLocation());
        zig.subtract(player.getLocation());
        zig.normalize();
        zig.multiply(dodgeDist);
        zig.rotateY((float)Math.PI/2);

        // 50% chance - dodge one way or the other
        if (MoreMath.chance(.5f)) {
            zig.multiply(-1);
        }

        // convert vector to absolute location
        zig.add(bot.getLocation());
        calcFloorHeight(zig, bot.getFloorHeight());

        Vector3D zag = new Vector3D(bot.getLocation());
        calcFloorHeight(zag, bot.getFloorHeight());

        List path = new ArrayList();
        path.add(zig);
        path.add(zag);
        return path.iterator();
    }
}
```

Patterns (5)

- The next dodge pattern, random, moves the bot to a random location within a half-circle of the player.
- By limiting to a half-circle, you can ensure the bot doesn't have to cross the player to get to the dodge location. The code is in Listing 13.11.

```
public class DodgePatternRandom extends AIPattern {

    private float radiusSq;

    public DodgePatternRandom(BSPTree tree, float radius) {
        super(tree);
        this.radiusSq = radius * radius;
    }

    public Iterator find(GameObject bot, GameObject player)
{

        Vector3D goal = getLocationFromPlayer(bot, player,
            radiusSq);

        // pick a random location on this half-circle
        // (-90 to 90 degrees from current location)
        float maxAngle = (float)Math.PI/2;
        float angle = MoreMath.random(-maxAngle, maxAngle);
        goal.subtract(player.getLocation());
        goal.rotateY(angle);
        goal.add(player.getLocation());
        calcFloorHeight(goal, bot.getFloorHeight());

        return Collections.singleton(goal).iterator();
    }

}
```

DodgePatternRandom.java

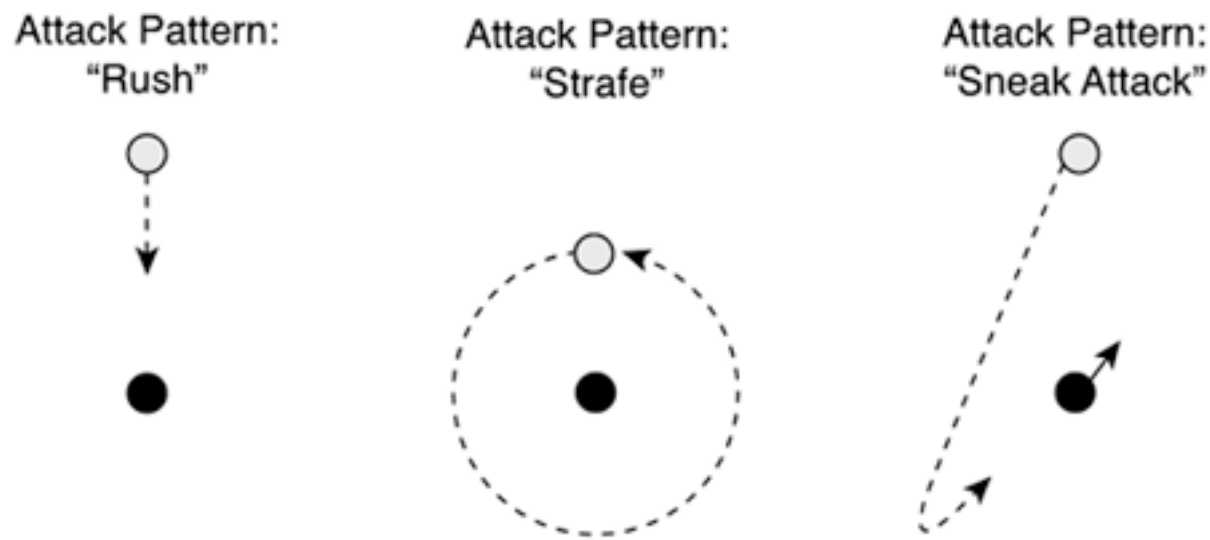
Patterns (6)

- Of course, these dodge patterns implemented here are just a beginning—you can come up with plenty more dodge patterns. Another idea for dodging is to keep track of incoming projectiles and move perpendicular to the projectile's path, or to do other movements such as ducking or jumping.

Patterns (7)

- **6.2. Attacking**

- Now you'll implement some attack patterns. Your attack patterns will vary depending on how your bots attack. For example, some might fire projectiles, some might try to ram the player, and others might simply try to make themselves look bigger to scare you away. Figure 13.7 shows some attack patterns you'll implement for the bots in this chapter.



Patterns (8)

- Here the bot just moves so that it's within a certain distance of the player. You can just use the **getLocationFromPlayer()** method you created in AIPattern, as shown in Listing 13.12.

```
public class AttackPatternRush extends AIPattern {

    private float desiredDistSq;

    public AttackPatternRush(BSPTree tree, float desiredDist)
{
        super(tree);
        this.desiredDistSq = desiredDist * desiredDist;
    }

    public Iterator find(GameObject bot, GameObject player) {
        Vector3D goal = getLocationFromPlayer(bot, player,
            desiredDistSq);
        if (goal.equals(bot.getLocation())) {
            return null;
        }
        else {
            return Collections.singleton(goal).iterator();
        }
    }

}
```

AttackPatternRush.java

Patterns (9)

- Note that this pattern returns null if the bot is already at the rush location.

- Strafing around a player might seem difficult at first, but if you can pull it off, it will really help boost the apparent intelligence of your bots. Strafing makes the bot circle around the player. This makes it easy for the bot to fire projectiles but harder for the player to fire at the bot, so it's a great offensive tactic.

Patterns (10)

- Instead of actually moving in a circle, just have the bot move in an octagon around the player. This reduces the path to eight locations. The code is in Listing 13.13.

- In this pattern, with a vector from the player to the desired radius, you can rotate this vector around the y-axis eight times to get the eight different points you want.

```
public class AttackPatternStrafe extends AIPattern {

    private float radiusSq;

    public AttackPatternStrafe(BSPTree tree, float radius) {
        super(tree);
        this.radiusSq = radius * radius;
    }

    public Iterator find(GameObject bot, GameObject player) {

        List path = new ArrayList();

        // find first location within desired radius
        Vector3D firstGoal = getLocationFromPlayer(bot, player,
            radiusSq);
        if (!firstGoal.equals(bot.getLocation())) {
            path.add(firstGoal);
        }

        // make a counter-clockwise circle around the player
        // (it's actually an octagon).
        int numPoints = 8;
        float angle = (float)(2 * Math.PI / numPoints);
        if (MoreMath.chance(0.5f)) {
            angle = -angle;
        }
        float lastY = bot.getFloorHeight();
        for (int i=1; i<numPoints; i++) {
            Vector3D goal = new Vector3D(firstGoal);
            goal.subtract(player.getLocation());
            goal.rotateY(angle * i);
            goal.add(player.getLocation());
            calcFloorHeight(goal, lastY);
            lastY = goal.y;
            path.add(goal);
        }

        // add last location (back to start)
        path.add(firstGoal);
        return path.iterator();
    }
}
```

AttackPatternStrafe.java

Patterns (11)

- Again, you could come up with a numerous other patterns besides these two. There's the sneak attack as in the previous figure. There are also other attacks, such as ramming, doing kamikaze attacks, waiting and ambushing, or jumping on the player.

- Of course, in a game, a bot doesn't always have to attack. Plenty of baddies, such as spiders or little animals, can be completely ambivalent to the player, just wandering around minding their own business. Games such as Nintendo's Mario series have plenty of creatures like these. Be sure to add some creativity and variety to the different types of patterns.

Patterns (12)

- **6.3. Running Away**

- A bot might run away from the player for lots of reasons. Perhaps the player has a really big, scary weapon, or the bot is so low on health that it just wants to get away. A lot of the time, you might want to just make a nervous bot that runs away often, or runs away for short periods as a defensive tactic.

- All you need for a run–away pattern is a spot to run to. Listing 13.14 shows a panic pattern that just makes the bot run directly away from the player, even if there is a wall in the way.

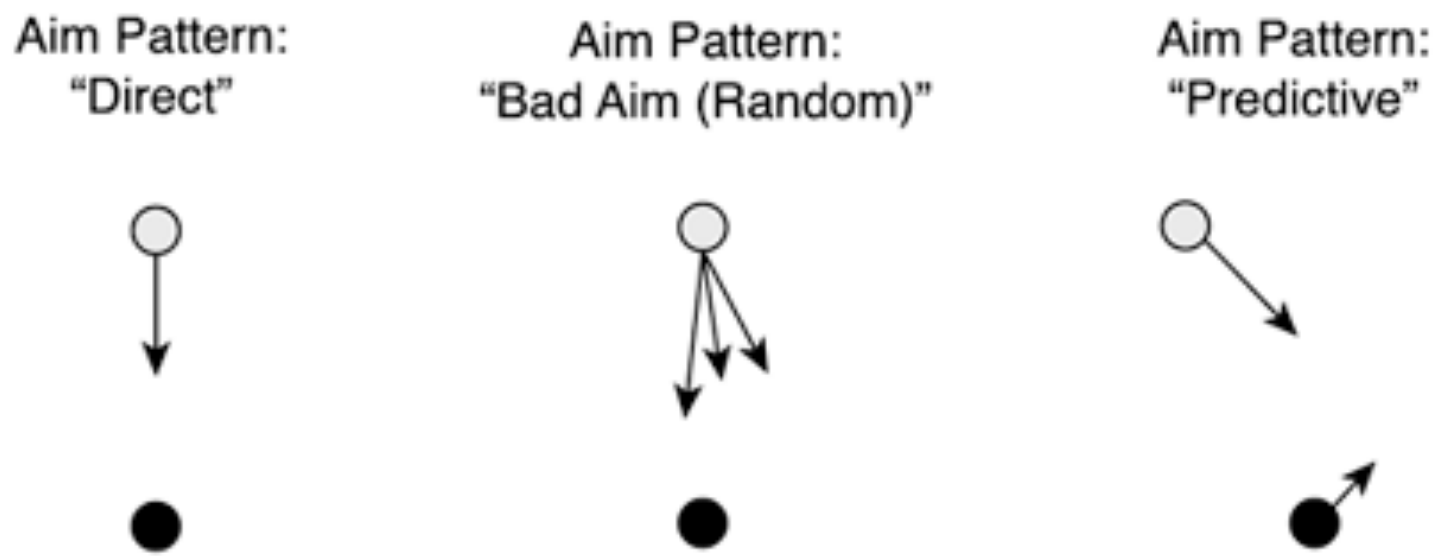
```
public class RunAwayPattern extends AIPattern {

    public RunAwayPattern(BSPTree tree) {
        super(tree);
    }

    public Iterator find(GameObject bot, GameObject player) {
        // dumb move: run in the *opposite* direction of the
        // player (will cause bots to run into walls!)

        Vector3D goal = new Vector3D(player.getLocation());
        goal.subtract(bot.getLocation());

        // opposite direction
        goal.multiply(-1);

        // far, far away
        goal.multiply(100000);
        calcFloorHeight(goal, bot.getFloorHeight());

        // return an iterator
        return Collections.singleton(goal).iterator();
    }
}
```

RunAwayPattern.java

Patterns (13)

- **6.4. Aiming**

- Although it's not really a movement pattern, often a bot needs to "aim" a weapon at the player, with various degrees of success: You'll probably want to give some bots better aiming than others. Figure 13.8 shows a few aim pattern ideas.



Patterns (14)

- You'll just implement one aim pattern that can aim with a specified accuracy from 0 to 1 (see Listing 13.15). If the accuracy is 0, the aim can be off up to 10°. If the accuracy is 1, the aim is dead on with the player, not taking the player's velocity into account.

- Note that this aim pattern returns a normalized vector direction rather than an absolute location because you're aiming in a direction.

```
public class AimPattern extends AIPattern {

    protected float accuracy;

    public AimPattern(BSPTree tree) {
        super(tree);
    }


    /**
        Sets the accuracy of the aim from 0 (worst) to 1 (best).
    */
    public void setAccuracy(float p) {
        this.accuracy = p;
    }


    public Iterator find(GameObject bot, GameObject player) {
        Vector3D goal = new Vector3D(player.getLocation());
        goal.y += player.getBounds().getTopHeight() / 2;
        goal.subtract(bot.getLocation());

        // Rotate up to 10 degrees off y-axis
        // (This could use an up/down random offset as well.)
        if (accuracy < 1) {
            float maxAngle = 10 * (1-accuracy);
            float angle = MoreMath.random(-maxAngle, maxAngle);
            goal.rotateY((float)Math.toRadians(angle));
        }
        goal.normalize();

        // return an iterator
        return Collections.singleton(goal).iterator();
    }
}
```

AimPattern.java

Patterns (15)

- **6.4. Firing**

- When aiming, another thing to consider is the amount of time it requires a bot to aim. You don't want a bot to simply fire its weapon for every frame; instead, you want it to wait for a bit, as if it's actually aiming, so that shots are fired only after the bot has spent time aiming the shot.

- In the code, we make the amount of time spent aiming proportional to the accuracy of the shot. Spending two seconds or more gives perfect aim, and anything less results in a slightly inaccurate aim. You add the code in the AIBot class in Listing 13.16.

Aiming and Firing in `AIBot`

```
AimPattern aimPathFinder;
long aimTime;

public void update(GameObject player, long elapsedTime)
{

    ...

    if (aiState == BATTLE_STATE_ATTACK &&
        elapsedTimeInState >= aimTime &&
        aimPathFinder != null)
    {
        elapsedTimeInState-=aimTime;

        // longer aim time == more accuracy
        float p = Math.min(1, aimTime / 2000f);
        aimPathFinder.setAccuracy(p);
        Vector3D direction = (Vector3D)
            aimPathFinder.find(this, player).next();
        fireProjectile(direction);
    }
}


/**
    Fires a projectile in the specified direction. The
    direction vector should be normalized.
*/
public void fireProjectile(Vector3D direction) {

    Projectile blast = new Projectile(
        (PolygonGroup)blastModel.clone(),
        direction, this, 3, 6);
    float dist = 2 * (getBounds().getRadius() +
        blast.getBounds().getRadius());
    blast.getLocation().setTo(
        getX() + direction.x*dist,
        getY() + getBounds().getTopHeight()/2,
        getZ() + direction.z*dist);

    // "spawns" the new game object
    addSpawn(blast);

}
```

7. Object Spawning

- As in the projectile example, sometimes an object needs to spawn another object. For example, an object could explode into several pieces, resulting in several spawns traveling in different directions; a bot could drop its weapon; or "sparks" could fly off a bot when it gets hit.

- It's probably a good idea to let any game object spawn another, so you'll add code in the GameObject class to add and retrieve spawns, shown here in Listing 13.17.

```
private List spawns;

...

/**
    Spawns an object, so that the GameManager can
    retrieve later using getSpawns().
*/
protected void addSpawn(GameObject object) {
    if (spawns == null) {
        spawns = new ArrayList();
    }
    spawns.add(object);
}


/**
    Returns a list of "spawned" objects
(projectiles,
    exploding parts, etc) or null if no objects
    were spawned.
*/
public List getSpawns() {
    List returnList = spawns;
    spawns = null;
    return returnList;
}
```

Spawning in GameObject

8. Putting It All Together

- Besides making a bot "explode" when it gets hit, you need to do a few other things to create a working demo using the AI bots:
  - You might want to show different object animations for each state. For example, the bot could have different animations for walking, running, strafing, aiming, or firing.
  - The bot should give some visual feedback when it gets wounded so the player knows it is damaging the bot.
  - Now that the bots can attack the player, you also need some visual feedback on the player's state. For instance, you could use a health bar and a damage indicator.
- You are using the same simple pyramid-shape bots from the previous chapter, keeping the object animations to a minimum. You'll also create a simple overlay framework to draw a health bar.

Putting It All Together (2)

- **8.1. Brains!**

- You'll store all the movement patterns and other attributes of the bots in a simple Brain class, shown in Listing 13.18. Every AIBot will have an instance of the Brain class.

- This is a bare class that just includes fields for each brain attribute. The only function here is one to fix the probabilities so their sum is 1.

```
public class Brain {

    public PathFinder attackPathFinder;
    public PathFinder dodgePathFinder;
    public PathFinder aimPathFinder;
    public PathFinder idlePathFinder;
    public PathFinder chasePathFinder;
    public PathFinder runAwayPathFinder;

    // probability of each battle state
    // (the sum should be 1)
    public float attackProbability;
    public float dodgeProbability;
    public float runAwayProbability;

    public long decisionTime;
    public long aimTime;
    public float hearDistance;

    public void fixProbabilites() {
        // make the sums of the odds == 1.
        float sum = attackProbability + dodgeProbability +
            runAwayProbability;
        if (sum > 0) {
            attackProbability /= sum;
            dodgeProbability /= sum;
            runAwayProbability /= sum;
        }
    }
}
```

Brain.java

Putting It All Together (3)

- **8.2. Health and Dying**

- Of course, now that you've got bots that can move around an attack, you need to make it so bots can be destroyed.

- In previous chapter demos, we had the "one strike and you're out" rule, where a projectile hitting a bot destroyed it, making it immediately disappear. In a game, you'll want to make it a bit more realistic than that. Here are some examples:

  - Make some bots take more hits to destroy them than others.

  - Make some bots more vulnerable to certain weapons. For example, a bot might be able to shield laser blasts but might be completely defenseless against missiles.

  - When hit, make a bot move to a wounded state that causes the bot to stop for a moment. Also, make the bot invulnerable to any more damage for the short period of time it's in the wounded state.

Putting It All Together (4)

- Make different decisions depending on the health of a bot. A bot with full health might be brave, while a bot with critically low health might just run away from the player.

- After a bot is destroyed, don't make it immediately disappear. Have it lie there for a short amount of time, and remove it a few seconds later. Some games just let dead bots lie there throughout the entire game. Another idea is to make the robot explode into oblivion (making blast marks on the floors) or disappear in a puff of smoke.

- A destroyed bot could drop certain items, such as ammo or energy, that the player can pick up.

- Be sure to find the right balance between the player and bad guys in your game, and make enemies progressively more difficult as the game moves forward.

- As always, be creative!

Putting It All Together (5)

- In this chapter, you'll create fairly simple health and dying routines. Each bot will start with 100 health points, and the player's weapon will cause a random amount of damage between 40 and 60 health points. Remember, the player fires projectiles that are game objects, so you can use our collision code to check whether a projectile hits anything.

- First, you'll add a couple more states to the bot, to signify when the bot has been hit and when it is dead:

```
public static final int WOUNDED_STATE_HURT = 6;
public static final int WOUNDED_STATE_DEAD = 7;
```

- Keep a bot in the hurt state for a few seconds after it gets hit. Likewise, if the bot's AI state is dead, keep it in that state for a few seconds and then remove it from the game object manager.

Putting It All Together (6)

- First, add some health functions to the AIBot class, shown here in Listing 13.19.
- The addHealth() method is where all the action happens. Note that this method decreases the health and puts the bot in the hurt state only if it's not currently hurt.
- The isCriticalHealth() method returns true if the bot's health is critically low; if so, you can make it run away.

Health Methods in `AIBot`

```
private static final float DEFAULT_MAX_HEALTH = 100;
private static final float CRITICAL_HEALTH_PERCENT = 5;

private float maxHealth;
private float health;

...

protected void setHealth(float health) {
    this.health = health;
}

/**
    Adds the specified amount to this bot's health. If the
    amount is less than zero, the bot's state is set to
    WOUNDED_STATE_HURT.
*/
public void addHealth(float amount) {
    if (amount < 0) {
        if (health <= 0 || aiState == WOUNDED_STATE_HURT) {
            return;
        }
        setAiState(WOUNDED_STATE_HURT, null);
    }
    setHealth(health + amount);
}


/**
    Returns true if the health is critically low (less than
    CRITICAL_HEALTH_PERCENT).
*/
public boolean isCriticalHealth() {
    return (health / maxHealth < CRITICAL_HEALTH_PERCENT /
100);
}
```

Putting It All Together (7)

- Next you must make some decisions based on the bot's health and wounded state. You need to make sure a bot stays in the hurt state for a short period of time, and you need to destroy a bot if it has been dead for a while. The code to do all this is here in Listing 13.20.

- The code shows some decision-making routines in the update() method of AIBot. After a bot is hurt, 500ms later it either moves to the idle state or, if the bot's health is 0, moves to the dead state. When in the dead state, it destroys itself after five seconds. Finally, if the health of the bot is critically low, the bot runs away from the player.

```
public void update(GameObject player, long elapsedTime) {

    elapsedTimeInState+=elapsedTime;

    ...

    if (aiState == WOUNDED_STATE_DEAD) {
        // destroy bot after five seconds
        if (elapsedTimeInState >= 5000) {
            setState(STATE_DESTROYED);
        }
        return;
    }
    else if (aiState == WOUNDED_STATE_HURT) {
        // after 500ms switch to either the idle or dead state.
        if (elapsedTimeInState >= 500) {
            if (health <= 0) {
                setAiState(WOUNDED_STATE_DEAD, player);
                return;
            }
            else {
                aiState = NORMAL_STATE_IDLE;
            }
        }
        else {
            return;
        }
    }

    // run away if health critical
    if (isCriticalHealth() && brain.runAwayPathFinder != null) {
        setAiState(BATTLE_STATE_RUN_AWAY, player);
        return;
    }

    ...

}
```

Wounded and Dying Methods in `AIBot`

Putting It All Together (8)

- Another idea is to make the bot disappear only when it's offscreen so the player doesn't notice it vanish.

- Alternatively, many games show some sort of explosion effect when a bot disappears. Earlier, we talked about how you need to set the appropriate PathFinder whenever you set the AI state.

- You also need to change the object's animation at this time. Listing 13.21 shows the extended setAIState() method in AIBot. In this code, when a bot is hurt, it stops moving and spins around a little bit, as if getting hit made it dizzy.

```
protected void setAiState(int aiState, GameObject player) {

    if (this.aiState == aiState) {
        return;
    }

    this.aiState = aiState;

    elapsedTimeInState = 0;
    Vector3D playerLocation = null;
    if (player != null) {
        playerLocation = player.getLocation();
    }

    // update path
    switch (aiState) {
        case BATTLE_STATE_ATTACK:
            setPathFinder(brain.attackPathFinder);
            setFacing(playerLocation);
            break;
        case BATTLE_STATE_DODGE:
            setPathFinder(brain.dodgePathFinder);
            setFacing(null);
            break;

        ...

        case WOUNDED_STATE_HURT:
            setPathFinder(null);
            setFacing(null);
            getTransform().stop();
            getTransform().setAngleVelocityY(
                MoreMath.random(0.001f, 0.05f),
                MoreMath.random(100, 500));
            break;
    }

}
```

Extended `setAIState()` Method of `AIBot`

Putting It All Together (9)

- Now you need to show the health of the player because the AI bots can attack, thus depleting the player's health. You'll accomplish this by adding a heads-up display.

- **8.3. Adding a Heads-Up Display**

- As far as I can tell, the term heads-up display, or HUD, originated from certain car instrument panels. On common cars, the driver has to look down to see the speed, fuel meter, odometer, flux capacitor, and whatever other gauges are on the instrument panel. But with cars that have a heads-up display, the instrument panel is projected onto the windshield or is displayed in a racing driver's helmet so drivers can keep their heads up while driving.

Putting It All Together (10)

- The primary purpose of heads-up displays is to provide information on the state of the game that can't be displayed in the game itself. Some types of information that are common in heads-up displays are listed here:
  - Amount of health, ammo, remaining lives, and so on
  - In-game messages (which could include displaying icons when you pick up an item)
  - Any currently active power-ups (such as super health or quad damage)
  - Time left to play
- Heads-up displays in games are normally drawn as an overlay on top of the view window. Also, they are often resolution independent, meaning they stay the same size onscreen no matter what the resolution of the screen is. Conversely, a resolution-dependent HUD, such as one designed for a 640x480 screen, would look smaller on a higher-resolution screen such as 1,024x768.

Putting It All Together (11)

- This implementation will be resolution independent. Start your implementation by creating a simple Overlay interface in Listing 13.22.

- Overlays are updated just like game objects and are drawn after each frame is drawn so they appear on top.

```
public interface Overlay {

    /**
        Updates this overlay with the specified amount of
        elapsed time since the last update.
    */
    public void update(long elapsedTime);

    /**
        Draws an overlay onto a frame. The ViewWindow specifies
        the bounds of the view window (usually, the entire
        screen). The screen bounds can be retrieved by calling
        g.getDeviceConfiguration().getBounds();
    */
    public void draw(Graphics2D g, ViewWindow viewWindow);


    /**
        Returns true if this overlay is enabled (should be
drawn).
    */
    public boolean isEnabled();
}
```

Overlay.java

Putting It All Together (12)

- As an example, you'll create a simple heads-up display that shows the health of the player, both as a number and as a bar. The HeadsUpDisplay class, in Listing 13.23, is a resolution-independent display of the player's health.

- Also, it's animated: The displayed health slightly lags behind the actual health of the player.

Putting It All Together (13)

```
public class HeadsUpDisplay implements Overlay {

    // increase health display by 20 points per second
    private static final float DISPLAY_INC_RATE = 0.04f;

    private Player player;
    private float displayedHealth;
    private Font font;

    public HeadsUpDisplay(Player player) {
        this.player = player;
        displayedHealth = 0;
    }

    public void update(long elapsedTime) {
        // increase or decrease displayedHealth a small amount
        // at a time, instead of just setting it to the player's
        // health.
        float actualHealth = player.getHealth();
        if (actualHealth > displayedHealth) {
            displayedHealth = Math.min(actualHealth,
                displayedHealth + elapsedTime * DISPLAY_INC_RATE);
        }
        else if (actualHealth < displayedHealth) {
            displayedHealth = Math.max(actualHealth,
                displayedHealth - elapsedTime * DISPLAY_INC_RATE);
        }
    }
    ...
```

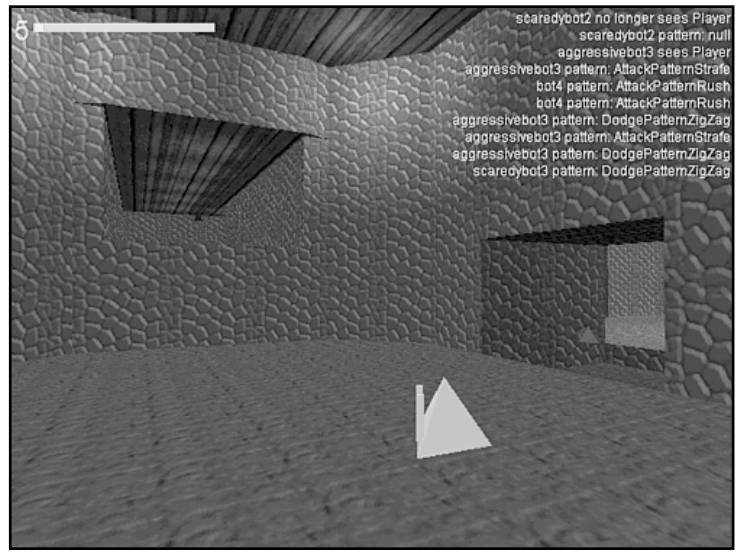
```
    ...
    public void draw(Graphics2D g, ViewWindow window) {

        // set the font (scaled for this view window)
        int fontHeight = Math.max(9, window.getHeight() / 20);
        int spacing = fontHeight / 5;
        if (font == null || fontHeight != font.getSize()) {
            font = new Font("Dialog", Font.PLAIN, fontHeight);
        }
        g.setFont(font);
        g.translate(window.getLeftOffset(), window.getTopOffset());

        // draw health value (number)
        String str = Integer.toString(Math.round(displayedHealth));
        Rectangle2D strBounds = font.getStringBounds(str,
            g.getFontRenderContext());
        g.setColor(Color.WHITE);
        g.drawString(str, spacing, (int)strBounds.getHeight());

        // draw health bar
        Rectangle bar = new Rectangle(
            (int)strBounds.getWidth() + spacing * 2,
            (int)strBounds.getHeight() / 2,
            window.getWidth() / 4,
            window.getHeight() / 60);
        g.setColor(Color.GRAY);
        g.fill(bar);

        // draw highlighted part of health bar
        bar.width = Math.round(bar.width *
            displayedHealth / player.getMaxHealth());
        g.setColor(Color.WHITE);
        g.fill(bar);
    }

    public boolean isEnabled() {
        return (player != null &&
            (player.isAlive() || displayedHealth > 0));
    }
}
```

HeadsUpDisplay.java

Putting It All Together (14)

- The font size and the health bar size are determined by the size of the view window. Other than that, there's really nothing special about this heads-up display. A text string and a couple of rectangles are drawn, and that's it. The health bar makes a great animated effect, though.

- This chapter's demo (called AIBotTest) includes one more overlay that acts as a message queue. In this case, the message queue displays the various changes in the AI bots' state in the upper-right corner of the screen. This helps debug the AI bots' behavior and gives developers a clue to what the bots are thinking, including whether a bot really can see or hear the player. Check out Figure 13.9 for a screenshot.



Putting It All Together (15)

- Note that you should include a heads-up display only when it's necessary. If you can show the health of the player in the game itself, do so. For example, in the original Mario games, Mario's health was directly related to his size: A big Mario could get hit twice before dying, but a small Mario could get hit only once. The game didn't need the words big or small in the heads-up display because it was already visually apparent.

- And you don't have to limit the health display to a bar, either. Some games use pie charts or heart icons. Feel free to be creative with how you display the health because a bar can be considered passé. For example, the game Doom shows a graphic of your face—the more damage you take, the bloodier your face becomes.

- Finally, keep in mind that the heads-up display doesn't have to be onscreen at all times. Parts of it could appear only when a significant change occurs, such as when the player gets more points or acquires a new power-up. In this case, the heads-up display could scroll on and off the screen as needed.

9. Evolution

- The last topic on our list of game AI concepts is evolution. Evolving AI bots gradually change their behavior over time, ideally becoming more adept at offensive and defensive tactics against the player.

- The advantage of adding evolution in a game is that the bots can become more challenging to an individual user's playing style. The idea is that each user will play a game using different tactics, and evolving bots will be able to adjust accordingly and gradually become more difficult for the player.

Evolution (2)

- You can implement evolution in a game in several ways. It can be done on the fly or with a bot occasionally modifying its own attributes to become better at attacking the player. Here are a few examples:
  - The bot could change attack and aim patterns until it finds one that hits the player more often.
  - The bot could fire several simultaneous "virtual" projectiles using different aim patterns to see which one is more likely to hit the player. Using virtual projectiles that the bot keeps track of instead of real projectiles means the bot can test several different aim patterns at once, instead of waiting to aim before firing a real projectile one at a time.
  - The bot could tweak other attributes on the fly as well, such as speed, decision time, hearing distance, and amount of time between making decisions.

Evolution (3)

- Another way to implement evolution is the old-fashioned way: through reproduction and genetic mutation. No, you won't actually make bots mate and have children—sorry to disappoint. But you can implement the idea of reproduction and mutation. Here it goes:

  - Only the "best" bots reproduce.

  - An offspring is a slightly mutated version of its parent—or, in other words, is the same as its parent, but with a few altered attributes.

Evolution (4)

- The idea behind mutation is that if a "bad" brain is created, it will be at the low end of the gene pool and won't be able to reproduce. But "good" brains will be at the high end of the gene pool and can reproduce. So, over time, the high end of the gene pool will get better and result in better bots. Here's how you'll implement evolution in this chapter:

  - When a bot is destroyed, it records how well it performed, which is the same as the amount of damage it caused to the player. The bot's brain is kept in a "gene pool."

  - Next, the bot is regenerated with a new brain, which is either one of the best-performing brains from the gene pool or a mutated offspring of one of the best-performing brains.

- Before we get to implementing evolution, though, we need to fill in one concept: **bot regeneration**.

Evolution (5)

- **9.1. Regeneration**

- Regeneration is a technique common in many games. This technique allows bots to "regenerate" after they die, restoring themselves to their starting location and state. This can be useful if you want to create a never-ending supply of baddies.

- You implement regeneration in the AIBot class in Listing 13.24. Instead of destroying the bot when it dies, it has the option to regenerate itself by completely restoring its state.

Regeneration Code of `AIBot`

```
/**
    Returns true if this bot regenerates after it dies.
*/
public boolean isRegenerating() {
    return isRegenerating;
}


/**
    Sets whether this bot regenerates after it dies.
*/
public void setRegenerating(boolean isRegenerating) {
    this.isRegenerating = isRegenerating;
}


/**
    Causes this bot to regenerate, restoring its location
    to its start location.
*/
protected void regenerate() {
    setHealth(maxHealth);
    setState(STATE_ACTIVE);
    setAiState(DECESION_READY, null);
    getLocation().setTo(startLocation);
    getTransform().stop();
    setJumping(false);
    setPathFinder(null);
    setFacing(null);
    // let the game object manager know this object regenerated
    // (so collision detection from old location to new
    // location won't be performed)
    addSpawn(this);
}
public void update(GameObject player, long elapsedTime) {

    ...

    elapsedTimeInState+=elapsedTime;

    // record first location
    if (startLocation == null) {
        startLocation = new Vector3D(getLocation());
    }

    // regenerate if dead for 5 seconds
    if (aiState == WOUNDED_STATE_DEAD) {
        if (elapsedTimeInState >= 5000) {
            if (isRegenerating()) {
                regenerate();
            }
            else {
                setState(STATE_DESTROYED);
            }
        }
        return;
    }
}
```

Evolution (6)

- The **update()** method is modified so that the **regenerate()** method is called if the bot has been dead for a few seconds and the bot has regeneration capabilities (isRegenerating()). The regenerate() method resets the bot and, in this case, sets its location to the place where it originated (startLocation).

- Also in the regenerate() method, you call the **addSpawn()** method to mark itself as a spawn. This is like sending a note to the game object manager that says, "Hey, I'm regenerating. Please don't perform collision detection on me this time." If collision detection were performed, the engine could think the bot virtually moved from the place where it died to the place where it regenerated instead of just reappearing there, and the bot could get stuck on a wall or against another object between those two locations. Not performing collision detection means the bot can correctly reappear at the location where it originated.

Evolution (7)

- **9.2. Evolving Bots**

- To allow only the best brains to reproduce, you need a way to track which brains are, in fact, the best. You can include many different factors in determining what makes one brain better than another, but in this case, you'll just look at the average amount of damage a bot caused with that brain.

- It's summed up in the BrainStat class in Listing 13.25, which is a subclass of the Brain class. It keeps track of the damage caused and the generation of the brain.

```
private class BrainStat extends Brain implements Comparable {
    long totalDamageCaused;
    int numBots;
    int generation;

    /**
        Gets the average damage this brain causes.
    */
    public float getAverageDamageCaused() {
        return (float)totalDamageCaused / numBots;
    }


    /**
        Reports damaged caused by a bot with this brain
        after the bot was destroyed.
    */
    public void report(long damageCaused) {
        totalDamageCaused+=damageCaused;
        numBots++;
    }


    /**
        Mutates this brain. The specified mutationProbability
        is the probability that each brain attribute
        becomes a different value, or "mutates."
    */
    public void mutate(float probability) {
        ...
    }


    /**
        Returns a smaller number if this brain caused more
        damage that the specified object, which should
        also be a brain.
    */
    public int compareTo(Object obj) {
        BrainStat other = (BrainStat)obj;
        if (this.numBots == 0 || other.numBots == 0) {
            return (other.numBots - this.numBots);
        }
        else {
            return (int)MoreMath.sign(
                other.getAverageDamageCaused() -
                this.getAverageDamageCaused());
        }
    }
}
```

BrainStat **Inner Class of** EvolutionGenePool

Evolution (8)

- The BrainStat class implements the Comparable interface so that a list of brains can easily be sorted from best to worst. You could use lots of other criteria to decide which brains are better, but using the amount of damage works well in this case.

- When a bot's projectile hits the player, the projectile needs to report back to the bot to tell it how much damaged was caused. You'll accomplish this when you implement the EvolutionBot in a little bit.

- The mutate() method isn't shown here, but it simply mutates each brain attribute if a certain random chance occurs. For example, if mutationProbability is 0.10, each attribute has a 10% chance of mutating. The code for mutating the aim time would look like this:

```
if (MoreMath.chance(mutationProbability)) {
    aimTime = MoreMath.random(300, 2000);
}
```

Evolution (9)

- When a brain is mutated, its generation count is incremented.
- Also, we're not showing the clone() method because it's a trivial method.
- Next, you must come up with a storage mechanism for all these brains in the EvolutionGenePool class in Listing 13.26.

Evolution (10)

```
public class EvolutionGenePool {

    private static final int NUM_TOP_BRAINS = 5;
    private static final int NUM_TOTAL_BRAINS = 10;

    private List brains;

    ...

    /**
        Gets a new brain from the gene pool. The brain will either
        be a "top" brain or a new, mutated "top" brain.
    */
    public Brain getNewBrain() {

        // 50% chance of creating a new, mutated brain
        if (MoreMath.chance(.5f)) {
            BrainStat brain =
                (BrainStat)getRandomTopBrain().clone();

            // 10% to 25% chance of changing each attribute
            float p = MoreMath.random(0.10f, 0.25f);
            brain.mutate(p);
            return brain;
        }
        else {
            return getRandomTopBrain();
        }
    }
```

```
    /**
        Gets a random top-performing brain.
    */
    public Brain getRandomTopBrain() {
        int index = MoreMath.random(NUM_TOP_BRAINS-1);
        return (Brain)brains.get(index);
    }


    /**
        Notify that a creature with the specified brain has
        been destroyed. The brain's stats are recorded. If the
        brain's stats are within the top total brains
        then we keep the brain around.
    */
    public void notifyDead(Brain brain, long damageCaused) {
        // update statistics for this brain
        if (brain instanceof BrainStat) {
            BrainStat stat = (BrainStat)brain;

            // report the damage
            stat.report(damageCaused);

            // sort and trim the list
            if (!brains.contains(stat)) {
                brains.add(stat);
            }
            Collections.sort(brains);
            while (brains.size() > NUM_TOTAL_BRAINS) {
                brains.remove(NUM_TOTAL_BRAINS);
            }
        }
    }
}
```

EvolutionGenePool.java

Evolution (11)

- This class keeps track of 10 brains total, and only the top 5 brains are allowed to have offspring.

- When a bot dies, it calls the **notifyDead()** method to let the gene pool know how much damage it caused using the specified brain. When a new bot is created or regenerated, it calls the **getNewBrain()** method to get a brain. This method has a 50% chance of creating a mutated offspring and a 50% chance of just returning one of the top brains.

Evolution (12)

- Finally, create the EvolutionBot class in Listing 13.27. This bot is a subclass of AIBot and performs all the necessary functions to regenerate and report how much damage it caused to the gene pool.

- The regenerate() method just overrides AIBot's method to perform the extra functionality you need.

```
public class EvolutionBot extends AIBot {

    private EvolutionGenePool genePool;
    private long damagedCaused;

    public EvolutionBot(PolygonGroup polygonGroup,
        CollisionDetection collisionDetection,
        EvolutionGenePool genePool, PolygonGroup blastModel)
    {
        super(polygonGroup, collisionDetection,
            genePool.getNewBrain(), blastModel);
        this.genePool = genePool;
        setRegenerating(true);
    }

    public void regenerate() {
        genePool.notifyDead(brain, damagedCaused);
        brain = genePool.getNewBrain();
        damagedCaused = 0;
        super.regenerate();
    }

    public void notifyHitPlayer(long damage) {
        damagedCaused+=damage;
    }
}
```

EvolutionBot.java

Evolution (13)

- Furthermore, when the game exits, it prints the attributes of the top five brains to the console so you can get an idea of what the best brains were. The longer you play, the more likely it is that this list will contain really valuable brains.

- Note that the player has one of the biggest effects on evolution. The player might have a preference to kill certain types of bots first. For example, bots that perform the strafe attack pattern might be so dangerous that the player tries to kill them first, allowing others to get in more damage, thus affecting evolution. Also, some bots might have a worse tactical advantage at their starting location than others, and a player could just hover around the regeneration location to pick off bots quickly, even if they would have been really smart.

- As we mentioned earlier, the amount of damage a bot causes isn't the only way to choose which brains are the best. Some other ideas include long life, the percentage of shots that hit the target, and the number of bullets dodged. Ideally, the best bots would have a combination of these positive characteristics.

Evolution (14)

- **9.2. Demo Enhancements**

- As usual, lots of features can be added to make the demo better. The game demo could use refinements such as health and ammo power-ups, different weapons, and "lock-on" targeting so the player can more easily attack the bots.

- The bots could use more patterns in general, including a better run away pattern, and the patterns could be smart enough to check the environment to decide on the best possible pattern (for example, a pattern could attempt to avoid a wall collision).

- Finally, the player could lose all of its health, but it never dies. You could use some sort of death sequence here. Also, a damage indicator on the heads-up display would help show when the player gets hit. This could be as easy as flashing the screen red for a few milliseconds.

10. Other Game AI Ideas

- If you're making a 2D platform game, the AI will probably be minimal, with creatures following simple patterns most of the time. Be sure to give the creatures varying level of intelligence, though. For example, some creatures could detect the edge of a platform and avoid them, while others would just fall right off.

- For a first-person shooter, the bots could spend some time trying to predict the movement of the player and adjust patterns accordingly. A bot could fire its weapon in the direction the player is predicted to be rather than where the player is when the shot was fired. Also, bots could try to "learn" a player's common patterns.

- Besides hearing and seeing, some bots could have other senses. For example, a bot could "smell" to pick up on the player's trail. Or, some bots could have special heat-sensing or x-ray vision to help them track down the player.

- You could extend evolution in a game by running the reproduction and mutation simulation over a long period, and then including only the smartest bots in the final game.

Other Game AI Ideas (2)

- **10.1. Team AI**

- In a strategy game, it can often be advantageous for AI bots to be arranged in a hierarchy of leadership. The team leaders would send commands to troops, trying to create a tactical advantage for the group as a whole rather than each troop trying to make decisions individually. This can make a game more challenging for the user. If the team leader is killed, either the troops become disorganized or one of the troops is promoted to team leader.

- There could also be team-based patterns, such as flocking to one point or completely surrounding a player on all sides.

- Flocking can show some cool AI behavior and is also common with AI for groups of fish or birds. Flocking algorithms involve keeping each bot near the group, keeping each bot a certain distance away from each other (having a "personal space"), and steering each bot in the average direction of the group. The group can be defined explicitly or can be just the bot's nearest neighbors.

Other Game AI Ideas (3)

- Also, troops could call for backup. Note that if you do something like this, make it visually apparent that a troop is calling for backup, or it could just look like more troops showing up randomly.

- Or, instead of a hierarchy, nearby troops could simply "communicate" with each other and negotiate the best possible strategy among them.

11. Summary

- In this chapter, we covered a lot of useful game AI, such as seeing and hearing, state machines, probability, and evolution. In the process, you made a demo with bots that can attack the player, and you added some basic game elements to accommodate this demo, such as bot health and dying, regeneration, and a simple overlay framework showing a heads-up display.

- As mentioned before, how you implement AI is really up to the needs of your game. A lot of the work will just be in finding a good balance that makes the game both challenging and fun. And sometimes you just have to find that balance by trial and error. So experiment, make the bad guys smart, and make some games with some cool AI.