Chapter 9

Adding Weapons and Combat

Over the past few chapters, we built the basic framework for our game; added entities such as vehicles, aircraft, and buildings; implemented unit movement; and created a simple economy using the sidebar. We now have a game where we can start the level, earn money, purchase buildings and units, and move these units around to achieve simple goals.

In this chapter, we will implement weapons for vehicles, aircraft, and turrets. We will add the ability to process combat-based orders such as attacking, guarding, patrolling, and hunting to allow the units to fight in an intelligent way. Finally, we will implement a fog of war that limits visibility on the map, allowing for interesting strategies such as sneak attacks and ambushes.

Let's get started. We will use the code from Chapter 8 as a starting point.

Implementing the Combat System

Our game will have a fairly simple combat system. All units and turrets will have their own weapon and bullet type defined. When attacking an enemy, units will first get within range, turn toward the target, and then fire a bullet at them. Once the unit fires a bullet, it will wait until its weapon has reloaded before it fires again.

The bullet itself will be a separate game entity with its own animation logic. When fired, the bullet will fly toward its target and explode once it reaches its destination.

The first thing we will do is add bullets to our game.

Adding Bullets

We will start by defining a new `bullets` object inside `bullets.js`, as shown in Listing 9-1.

Listing 9-1. Defining the bullets Object (bullets.js)

```
var bullets = {
    list:{
        "fireball":{
            name:"fireball",
            speed:60,
            reloadTime:30,
            range:8,
            damage:10,
            spriteImages:[
                {name:"fly",count:1,directions:8},
                {name:"explode",count:7}
            ],
        },
        "heatseeker":{
            name:"heatseeker",
            reloadTime:40,
            speed:25,
            range:9,
            damage:20,
            turnSpeed:2,
            spriteImages:[
                {name:"fly",count:1,directions:8},
                {name:"explode",count:7}
            ],
        },
        "cannon-ball":{
            name:"cannon-ball",
            reloadTime:40,
            speed:25,
            damage:10,
            range:6,
            spriteImages:[
                {name:"fly",count:1,directions:8},
                {name:"explode",count:7}
            ],
        },
        "bullet":{
            name:"bullet",
            damage:5,
            speed:50,
            range:5,
            reloadTime:20,
            spriteImages:[
                {name:"fly",count:1,directions:8},
                {name:"explode",count:3}
            ],
        },
    },
    defaults:{
        type:"bullets",
        distanceTravelled:0,
        animationIndex:0,
        direction:0,
        directions:8,
        pixelWidth:10,
        pixelHeight:11,
        pixelOffsetX:5,
        pixelOffsetY:5,
        radius:6,
        action:"fly",
        selected:false,
        selectable:false,
        orders:{type:"fire"},
        moveTo:function(destination){
            // Weapons like the heatseeker can turn slowly toward target while moving
            if (this.turnSpeed){
                // Find out where we need to turn to get to destination
                var newDirection = findFiringAngle(destination,this,this.directions);
                // Calculate difference between new direction and current direction
                var difference = angleDiff(this.direction,newDirection,this.directions);
                // Calculate amount that bullet can turn per animation cycle
                var turnAmount = this.turnSpeed*game.turnSpeedAdjustmentFactor;
                if (Math.abs(difference)>turnAmount){
                    this.direction = wrapDirection(this.direction+turnAmount*Math.abs(difference)/
difference,this.directions);
                }
            }

            var movement = this.speed*game.speedAdjustmentFactor;
            this.distanceTravelled += movement;

            var angleRadians = -((this.direction)/this.directions)*2*Math.PI ;

            this.lastMovementX = - (movement*Math.sin(angleRadians));
            this.lastMovementY = - (movement*Math.cos(angleRadians));
            this.x = (this.x +this.lastMovementX);
            this.y = (this.y +this.lastMovementY);
        },
        reachedTarget:function(){
            var item = this.target;
            if (item.type=="buildings"){
                return (item.x<= this.x && item.x >= this.x - item.baseWidth/game.gridSize &&
item.y<= this.y && item.y >= this.y - item.baseHeight/game.gridSize);
            } else if (item.type=="aircraft"){
                return (Math.pow(item.x-this.x,2)+Math.pow(item.y-(this.y+item.pixelShadowHeight/
game.gridSize),2) < Math.pow((item.radius)/game.gridSize,2));
           } else {
                  return (Math.pow(item.x-this.x,2)+Math.pow(item.y-this.y,2) <
 Math.pow((item.radius)/game.gridSize,2));
           }
        },
        processOrders:function(){
            this.lastMovementX = 0;
            this.lastMovementY = 0;
            switch (this.orders.type){
                case "fire":
                    // Move toward destination and stop when close by or if travelled past range
                    var reachedTarget = false;
                    if (this.distanceTravelled>this.range
                        || (reachedTarget = this.reachedTarget())) {
                        if(reachedTarget){
                            this.target.life -= this.damage;
                            this.orders = {type:"explode"};
                            this.action = "explode";
                            this.animationIndex = 0;
                        } else {
                            // Bullet fizzles out without hitting target
                            game.remove(this);
                        }
                    } else {
                        this.moveTo(this.target);
                    }
                    break;
            }
        },
        animate:function(){
            switch (this.action){
                case "fly":
                    var direction = wrapDirection(Math.round(this.direction),this.directions);
                     this.imageList = this.spriteArray["fly-"+ direction];
                    this.imageOffset = this.imageList.offset;
                    break;
                case "explode":
                    this.imageList = this.spriteArray["explode"];
                    this.imageOffset = this.imageList.offset + this.animationIndex;
                    this.animationIndex++;
                    if (this.animationIndex>=this.imageList.count){
                        // Bullet explodes completely and then disappears
                        game.remove(this);
                    }
                    break;
            }
        },
        draw:function(){
            var x = (this.x*game.gridSize)-game.offsetX-this.pixelOffsetX +
this.lastMovementX*game.drawingInterpolationFactor*game.gridSize;
            var y = (this.y*game.gridSize)-game.offsetY-this.pixelOffsetY +
this.lastMovementY*game.drawingInterpolationFactor*game.gridSize;
            var colorOffset = 0;
            game.foregroundContext.drawImage(this.spriteSheet, this.imageOffset*this.
pixelWidth,colorOffset, this.pixelWidth,this.pixelHeight, x,y,this.pixelWidth,this.pixelHeight);
        }
    },
    load:loadItem,
    add:addItem,
}
```

The `bullets` object follows the same pattern as all the other game entities. We start by defining a list of four bullet types: `fireball`, `heatseeker`, `cannon-ball`, and `bullet`. Each of the bullets has a common set of properties.

- `speed`: The speed at which the bullet travels
- `reloadTime`: The number of animation cycles after firing before the bullet can be fired again
- `damage`: The amount of damage to the target when the bullet explodes
- `range`: The maximum range that a bullet will fly before it loses momentum

The bullets also have two animation sequences defined: `fly` and `explode`. The `fly` state has eight directions similar to vehicles and aircraft. The `explode` state has only direction but has multiple frames.

We then define a default moveTo() method, which is similar to the aircraft moveTo() method. Within this method we first check whether the bullet can turn and, if so, gently turn the bullet toward its destination using the findFiringAngle() method to calculate the angle toward the center of the target. Next, we move the bullet forward along its current direction and update the bullet's distanceTravelled property.

Next we define a reachedTarget() method that checks whether the bullet has reached its target. We check whether the bullet's coordinates are inside the base area for buildings and within the item radius for vehicles and aircraft. If so, we return a value of true.

Within the processOrders() method, we implement the fire order. We check whether the bullet has either reached its target or traveled for more than its range. If not, we continue to move the bullet toward the target.

If the bullet travels beyond its range without hitting the target, we remove it from the game. If the bullet reaches its target, we first set the bullet's order and animation state to explode and reduce the life of its target by the damage amount.

Within the animate() method, we remove the bullet once the explode animation sequence completes.

Now that we have defined the bullets object, we will add a reference to bullets.js inside the <head> section of index.html, as shown in Listing 9-2.

Listing 9-2. Adding a Reference to the bullets Object (index.html)

```
<script src="js/bullets.js" type="text/javascript" charset="utf-8"></script>
```

We will also define the findFiringAngle() method inside common.js, as shown in Listing 9-3.

Listing 9-3. Defining the findFiringAngle() Method (common.js)

```
function findFiringAngle(target,source,directions){
    var dy = (target.y) - (source.y);
    var dx = (target.x) - (source.x);

    if(target.type=="buildings"){
        dy += target.baseWidth/2/game.gridSize;
        dx += target.baseHeight/2/game.gridSize;
    } else if(target.type == "aircraft"){
        dy -= target.pixelShadowHeight/game.gridSize;
    }

     if(source.type=="buildings"){
        dy -= source.baseWidth/2/game.gridSize;
        dx -= source.baseHeight/2/game.gridSize;
    } else if(source.type == "aircraft"){
        dy += source.pixelShadowHeight/game.gridSize;
    }

    //Convert Arctan to value between (0 - 7)
    var angle = wrapDirection(directions/2-(Math.atan2(dx,dy)*directions/(2*Math.PI)),directions);
    return angle;
}
```

The findFiringAngle() method is similar to the findAngle() method except we adjust the values of the dy and dx variables to point to the center of the source and target. For buildings, we adjust dx and dy using the baseWidth and baseHeight properties, and for aircraft we adjust dy by the pixelShadowHeight property. This way, bullets can be aimed at the center of the target.

We will also modify the loadItem() method inside common.js to load the bullet for an item when the item loads, as shown in Listing 9-4.

Listing 9-4. Loading the Bullets When Loading the Item (common.js)

```
/* The default load() method used by all our game entities*/
function loadItem(name){
    var item = this.list[name];
    // if the item sprite array has already been loaded then no need to do it again
    if(item.spriteArray){
        return;
    }
    item.spriteSheet = loader.loadImage('images/'+this.defaults.type+'/'+name+'.png');
    item.spriteArray = [];
    item.spriteCount = 0;

    for (var i=0; i < item.spriteImages.length; i++){
        var constructImageCount = item.spriteImages[i].count;
        var constructDirectionCount = item.spriteImages[i].directions;
        if (constructDirectionCount){
            for (var j=0; j < constructDirectionCount; j++) {
                var constructImageName = item.spriteImages[i].name +"-"+j;
                item.spriteArray[constructImageName] = {
                    name:constructImageName,
                    count:constructImageCount,
                    offset:item.spriteCount
                };
                item.spriteCount += constructImageCount;
            };
        } else {
            var constructImageName = item.spriteImages[i].name;
            item.spriteArray[constructImageName] = {
                name:constructImageName,
                count:constructImageCount,
                offset:item.spriteCount
            };
            item.spriteCount += constructImageCount;
        }
    };
    // Load the weapon if item has one
    if(item.weaponType){
        bullets.load(item.weaponType);
    }
}
```

When loading an item, we check whether it has a weaponType property defined and, if so, load the bullet for the weapon using the bullets.load() method. All entities that are capable of attacking will have a weaponType property.

The next change we will make is to modify the game object's drawingLoop() method to draw the bullets and explosions on top of all other items in the game. The updated drawingLoop() method will look like Listing 9-5.

Listing 9-5. Modifying drawingLoop() to Draw Bullets Above Other Items (game.js)

```
drawingLoop:function(){
    // Handle Panning the Map
    game.handlePanning();

    // Check the time since the game was animated and calculate a linear interpolation factor
(-1 to 0)
    // since drawing will happen more often than animation
    game.lastDrawTime = (new Date()).getTime();
    if (game.lastAnimationTime){
        game.drawingInterpolationFactor = (game.lastDrawTime -game.lastAnimationTime)/game.
animationTimeout - 1;
        if (game.drawingInterpolationFactor>0){ // No point interpolating beyond the next
animation loop...
            game.drawingInterpolationFactor = 0;
        }
    } else {
        game.drawingInterpolationFactor = -1;
    }

    // Since drawing the background map is a fairly large operation,
    // we only redraw the background if it changes (due to panning)
    if (game.refreshBackground){
        game.backgroundContext.drawImage(game.currentMapImage,game.offsetX,game.offsetY,
game.canvasWidth, game.canvasHeight, 0,0,game.canvasWidth,game.canvasHeight);
        game.refreshBackground = false;
    }

    // Clear the foreground canvas
    game.foregroundContext.clearRect(0,0,game.canvasWidth,game.canvasHeight);

    // Start drawing the foreground elements
    for (var i = game.sortedItems.length - 1; i >= 0; i--){
        if (game.sortedItems[i].type != "bullets"){
            game.sortedItems[i].draw();
        }
    };

    // Draw the bullets on top of all the other elements
    for (var i = game.bullets.length - 1; i >= 0; i--){
        game.bullets[i].draw();
    };

    // Draw the mouse
    mouse.draw()

    // Call the drawing loop for the next frame using request animation frame
    if (game.running){
        requestAnimationFrame(game.drawingLoop);
    }
},
```

We first draw all the items that are not bullets and finally draw the bullets. This way, bullets and explosions will always be clearly visible in the game.

Finally, we will modify the game object's `resetArrays()` method to also reset the `game.bullets[]` array, as shown in Listing 9-6.

Listing 9-6. Resetting the Bullets Array Inside resetArrays()(game.js)

```
resetArrays:function(){
    game.counter = 1;
    game.items = [];
    game.sortedItems = [];
    game.buildings = [];
    game.vehicles = [];
    game.aircraft = [];
    game.terrain = [];
    game.triggeredEvents = [];
    game.selectedItems = [];
    game.sortedItems = [];
    game.bullets = [];
},
```

Now that we have implemented the `bullets` object, it's time to implement combat-based orders for the turrets, vehicles, and aircraft.

Combat-Based Orders for Turrets

Ground turrets can fire cannonballs at any ground-based threat. When in `guard` or `attack` mode, they will search for a valid target that is in sight, aim the turret toward the target, and fire bullets until the target is either destroyed or out of range.

We will start by implementing the `processOrders()` method by modifying the `ground-turret` object inside `buildings.js`, as shown in Listing 9-7.

Listing 9-7. Modifying ground-turret Object to Implement Attack (buildings.js)

```
isValidTarget:isValidTarget,
findTargetsInSight:findTargetsInSight,
processOrders:function(){
    if(this.reloadTimeLeft){
        this.reloadTimeLeft--;
    }
    // damaged turret cannot attack
    if(this.lifeCode != "healthy"){
        return;
    }
    switch (this.orders.type){
        case "guard":
            var targets = this.findTargetsInSight();
            if(targets.length>0){
                this.orders = {type:"attack",to:targets[0]};
            }
            break;
        case "attack":
            if(!this.orders.to ||
                this.orders.to.lifeCode == "dead" ||
                !this.isValidTarget(this.orders.to) ||
                (Math.pow(this.orders.to.x-this.x,2) +
Math.pow(this.orders.to.y-this.y,2))>Math.pow(this.sight,2)
                ){

                var targets = this.findTargetsInSight();
                if(targets.length>0){
                    this.orders.to = targets[0];
                } else {
                    this.orders = {type:"guard"};
                }
            }

            if (this.orders.to){
                var newDirection = findFiringAngle(this.orders.to,this,this.directions);
                var difference = angleDiff(this.direction,newDirection,this.directions);
                var turnAmount = this.turnSpeed*game.turnSpeedAdjustmentFactor;
                if (Math.abs(difference)>turnAmount){
                    this.direction = wrapDirection(this.direction+turnAmount*Math.abs(difference)/
difference,this.directions);
                    return;
                } else {
                    this.direction = newDirection;
                    if(!this.reloadTimeLeft){
                        this.reloadTimeLeft = bullets.list[this.weaponType].reloadTime;
                        var angleRadians = -(Math.round(this.direction)/this.directions)*2*Math.PI ;
                        var bulletX = this.x+0.5- (1*Math.sin(angleRadians));
                        var bulletY = this.y+0.5- (1*Math.cos(angleRadians));
                        var bullet = game.add({name:this.weaponType,type:"bullets", x:bulletX,
y:bulletY, direction:this.direction, target:this.orders.to});
                    }
                }
            }
            break;
    }
}
```

We start by assigning two methods called `isValidTarget()` and `findTargetInSight()` inside the `ground-turret` object. We will need to define these methods. We then define the `processOrders()` method.

Within the `processOrders()` method, we decrease the value of the `reloadTimeLeft` property if the property is defined and is greater than 0. If the turret `lifeCode` is not `healthy` (it is damaged or dead), we do nothing and exit.

Next we define the behavior for both the `guard` and `attack` orders. In `guard` mode, we use the `findTargetsInSight()` method to find a target and, if we find one, attack it. In `attack` mode, if the current target of the turret is undefined, dead, or out of sight, we use `findTargetsInSight()` to find a new valid target and set an order to attack it. If we cannot find a valid target, we go back to `guard` mode.

If the turret does have a valid target, we turn it toward the target. Once the turret is facing the target and `reloadTimeLeft` is 0, we fire a bullet by adding it to the game using the `game.add()` method and reset the turret's `reloadTimeLeft` property to the bullet's reload time.

Next we will modify the guard animation case inside the default animate() method to handle directions, as shown in Listing 9-8.

Listing 9-8. Modifying the Guard Case Inside animate() (buildings.js)

```
case "guard":
    if (this.lifeCode == "damaged"){
        // The damaged turret has no directions
        this.imageList = this.spriteArray[this.lifeCode];
    } else {
        // The healthy turret has 8 directions
        var direction = wrapDirection(Math.round(this.direction),this.directions);
        this.imageList = this.spriteArray[this.lifeCode+"-"+ direction];
    }
    this.imageOffset = this.imageList.offset;
    break;
```

Next, we will add two methods called isValidTarget() and findTargetInSight() inside common.js, as shown in Listing 9-9.

Listing 9-9. Adding isValidTarget() and findTargetInSight() Methods (common.js)

```
// Common Functions related to combat
function isValidTarget(item){
    return item.team != this.team &&
(this.canAttackLand && (item.type == "buildings" || item.type == "vehicles")||
(this.canAttackAir && (item.type == "aircraft")));
}

function findTargetsInSight(increment){
    if(!increment){
        increment=0;
    }
    var targets = [];
    for (var i = game.items.length - 1; i >= 0; i--){
        var item = game.items[i];
        if (this.isValidTarget(item)){
            if(Math.pow(item.x-this.x,2) + Math.pow(item.y-this.y,2)<Math.pow(this.
sight+increment,2)){
                targets.push(item);
            }
        }
    };

    // Sort targets based on distance from attacker
    var attacker = this;
    targets.sort(function(a,b){
        return (Math.pow(a.x-attacker.x,2) + Math.pow(a.y-attacker.y,2))-(Math.pow(b.x-attacker.x,2)
+ Math.pow(b.y-attacker.y,2));
       });

    return targets;
}
```

The isValidTarget() method returns true if the target item is from the opposite team, and it can be attacked.

The findTargetsInSight() method checks all the items in the game.items() array to see whether they are valid targets and within range, and if so, it adds them to the targets array. It then sorts the targets array by distance of each target from the attacker. The method also accepts an optional increment parameter that allows us to find targets beyond the range of the item. These two common methods will be used by turrets, vehicles, and aircraft.

Before we see the results of our code, we will update our map from the last level by modifying the triggers and items arrays, as shown in Listing 9-10.

Listing 9-10. Updating the Map Items and Triggers (maps.js)

```
/* Entities to be added */
"items":[
    {"type":"buildings","name":"base","x":11,"y":14,"team":"blue"},
    {"type":"buildings","name":"starport","x":18,"y":14,"team":"blue"},

    {"type":"vehicles","name":"harvester","x":16,"y":12,"team":"blue","direction":3},
    {"type":"terrain","name":"oilfield","x":3,"y":5,"action":"hint"},

    {"type":"terrain","name":"bigrocks","x":19,"y":6},
    {"type":"terrain","name":"smallrocks","x":8,"y":3},

    {"type":"vehicles","name":"scout-tank","x":26,"y":14,"team":"blue","direction":4},
    {"type":"vehicles","name":"heavy-tank","x":26,"y":16,"team":"blue","direction":5},
    {"type":"aircraft","name":"chopper","x":20,"y":12,"team":"blue","direction":2},
    {"type":"aircraft","name":"wraith","x":23,"y":12,"team":"blue","direction":3},

    {"type":"buildings","name":"ground-turret","x":15,"y":23,"team":"green"},
    {"type":"buildings","name":"ground-turret","x":20,"y":23,"team":"green"},

    {"type":"vehicles","name":"scout-tank","x":16,"y":26,"team":"green","direction":4},
    {"type":"vehicles","name":"heavy-tank","x":18,"y":26,"team":"green","direction":6},
    {"type":"aircraft","name":"chopper","x":20,"y":27,"team":"green","direction":2},
    {"type":"aircraft","name":"wraith","x":22,"y":28,"team":"green","direction":3},

    {"type":"buildings","name":"base","x":19,"y":28,"team":"green"},
    {"type":"buildings","name":"starport","x":15,"y":28,"team":"green"},
],

/* Conditional and Timed Trigger Events */
"triggers":[
],
```

We removed the triggers that we defined in the previous chapter so the level doesn't end after 30 seconds. Now, if we run the game in the browser and move a vehicle close to the enemy turrets, the turrets should start attacking the vehicle, as shown in Figure 9-1.

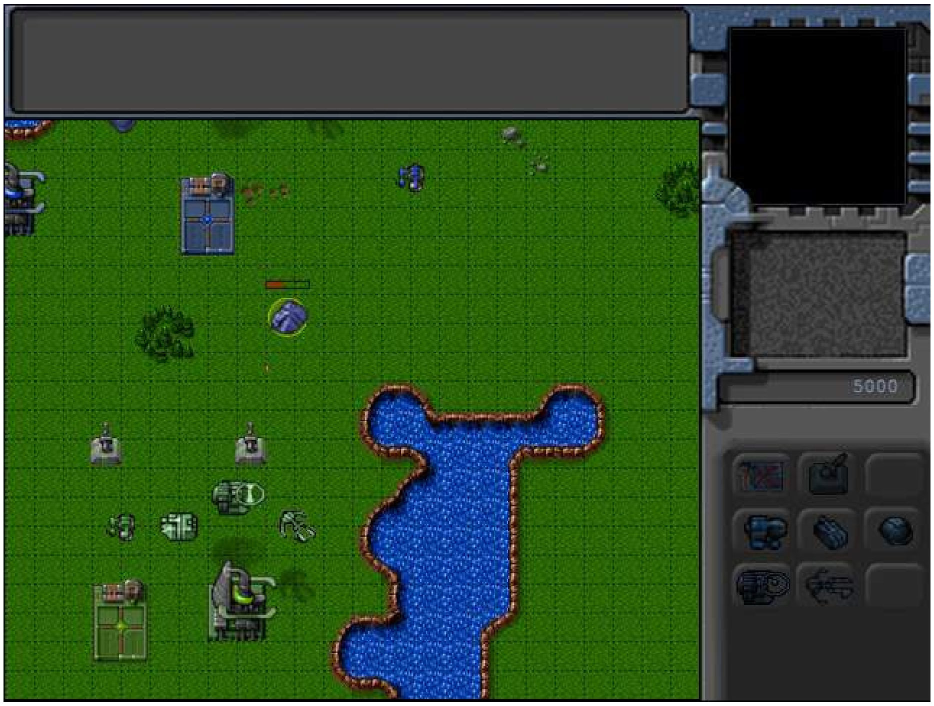


Figure 9-1. *Turret firing at a vehicle within range*

The bullets explode when they hit the vehicle and decrease the vehicle's life. Once the vehicle loses all its life, it disappears from the game. The turret stops shooting at a target if the target goes out of range and moves onto the next target.

Next we will implement combat-based orders for aircraft.

Combat-Based Orders for Aircraft

We will define several basic combat-based order states for aircraft.

- `attack`: Move within range of a target and shoot at it.
- `float`: Stay in one place and attack any enemy that comes close.
- `guard`: Follow a friendly unit and shoot at any enemy that comes close.
- `hunt`: Actively seek out enemies anywhere on the map and attack them.
- `patrol`: Move between two points and shoot at any enemy that comes within range.
- `sentry`: Stay in one place and attack enemies slightly more aggressively than in `float` mode.

We will implement these states by modifying the default `processOrders()` method inside the `aircraft` object, as shown in Listing 9-11.

Listing 9-11. Implementing Combat Orders for Aircraft (aircraft.js)

```
isValidTarget:isValidTarget,
findTargetsInSight:findTargetsInSight,
processOrders:function(){
    this.lastMovementX = 0;
    this.lastMovementY = 0;
    if(this.reloadTimeLeft){
        this.reloadTimeLeft--;
    }
    switch (this.orders.type){
        case "float":
            var targets = this.findTargetsInSight();
            if(targets.length>0){
                this.orders = {type:"attack",to:targets[0]};
            }
            break;
        case "sentry":
            var targets = this.findTargetsInSight(2);
            if(targets.length>0){
                this.orders = {type:"attack",to:targets[0],nextOrder:this.orders};
            }
            break;
        case "hunt":
            var targets = this.findTargetsInSight(100);
            if(targets.length>0){
                this.orders = {type:"attack",to:targets[0],nextOrder:this.orders};
            }
            break;
        case "move":
            // Move toward destination until distance from destination is less than aircraft radius
            var distanceFromDestinationSquared = (Math.pow(this.orders.to.x-this.x,2) +
Math.pow(this.orders.to.y-this.y,2));
            if (distanceFromDestinationSquared < Math.pow(this.radius/game.gridSize,2)) {
                this.orders = {type:"float"};
            } else {
                this.moveTo(this.orders.to);
            }
            break;
        case "attack":
            if(this.orders.to.lifeCode == "dead" || !this.isValidTarget(this.orders.to)){
                if (this.orders.nextOrder){
                    this.orders = this.orders.nextOrder;
                } else {
                    this.orders = {type:"float"};
                }
                return;
            }
            if ((Math.pow(this.orders.to.x-this.x,2) +
Math.pow(this.orders.to.y-this.y,2))<Math.pow(this.sight,2)) {
```

```
                //Turn toward target and then start attacking when within range of the target
                var newDirection = findFiringAngle(this.orders.to,this,this.directions);
                var difference = angleDiff(this.direction,newDirection,this.directions);
                var turnAmount = this.turnSpeed*game.turnSpeedAdjustmentFactor;
                if (Math.abs(difference)>turnAmount){
                    this.direction = wrapDirection(this.direction+ turnAmount*Math.abs(difference)/
difference, this.directions);
                    return;
                } else {
                    this.direction = newDirection;
                    if(!this.reloadTimeLeft){
                        this.reloadTimeLeft = bullets.list[this.weaponType].reloadTime;
                        var angleRadians = -(Math.round(this.direction)/this.directions)*2*Math.PI ;
                        var bulletX = this.x- (this.radius*Math.sin(angleRadians)/game.gridSize);
                        var bulletY = this.y- (this.radius*Math.cos(angleRadians)/game.gridSize)-
this.pixelShadowHeight/game.gridSize;
                        var bullet = game.add({name:this.weaponType, type:"bullets",x:bulletX,
y:bulletY, direction:newDirection, target:this.orders.to});
                    }
                }

            } else {
                var moving = this.moveTo(this.orders.to);
            }
            break;
        case "patrol":
            var targets = this.findTargetsInSight(1);
            if(targets.length>0){
                this.orders = {type:"attack",to:targets[0],nextOrder:this.orders};
                return;
            }
            if ((Math.pow(this.orders.to.x-this.x,2) + Math.pow(this.orders.to.y-
this.y,2))<Math.pow(this.radius/game.gridSize,2)) {
                var to = this.orders.to;
                this.orders.to = this.orders.from;
                this.orders.from = to;
            } else {
                this.moveTo(this.orders.to);
            }
            break;
        case "guard":
            if(this.orders.to.lifeCode == "dead"){
                if (this.orders.nextOrder){
                    this.orders = this.orders.nextOrder;
                } else {
                    this.orders = {type:"float"};
                }
                return;
            }
            if ((Math.pow(this.orders.to.x-this.x,2) + Math.pow(this.orders.to.y-
this.y,2))<Math.pow(this.sight-2,2)) {
                var targets = this.findTargetsInSight(1);
                if(targets.length>0){
                    this.orders = {type:"attack",to:targets[0],nextOrder:this.orders};
                    return;
                }
            } else {
                this.moveTo(this.orders.to);
            }
            break;

    }
},
```

We start by assigning the `isValidTarget()` and `findTargetInSight()` methods. We then define all the states inside the `processOrders()` method.

Within the `processOrders()` method, we decrease the value of the `reloadTimeLeft` property just like we did for turrets. We then define cases for each of the order states.

If the order type is `float`, we use `findTargetsInSight()` to check whether a target is nearby and, if so, attack it. We do the same thing when the order type is `sentry`, except we pass a range increment parameter of 2 so that the aircraft attacks units slightly beyond its typical range.

The `hunt` case is very similar except the range increment parameter is 100, which should ideally cover the entire map. This means the aircraft will attack any enemy unit or vehicle on the map starting with the nearest one.

For the `attack` case, we first check whether the target is still alive. If not, we either set orders to `orders.nextOrder` if it is defined or go back to `float` mode.

Next we check whether the target is within range, and if not, we move closer to the target. Next, we make sure that the aircraft is pointing toward the target. Finally, we wait until the `reloadTimeLeft` variable is 0 and then shoot a bullet at the target.

The `patrol` case is a combination of the `move` and `sentry` cases. We move the aircraft to the location defined in the `to` property and, once it reaches the location, turn around and move toward the `from` location. In case a target comes within range, we set the order to `attack` with the `nextOrder` set to the current order. This way, if the aircraft sees an enemy while patrolling, it will first attack the enemy and then go back to patrolling once the enemy has been destroyed.

Finally, in the case of `guard` mode, we move the aircraft within sight of the unit the aircraft is guarding and attack any enemy that comes close.

If you run the code we have so far, you should be able to see the different aircraft attacking each other, as shown in Figure 9-2.

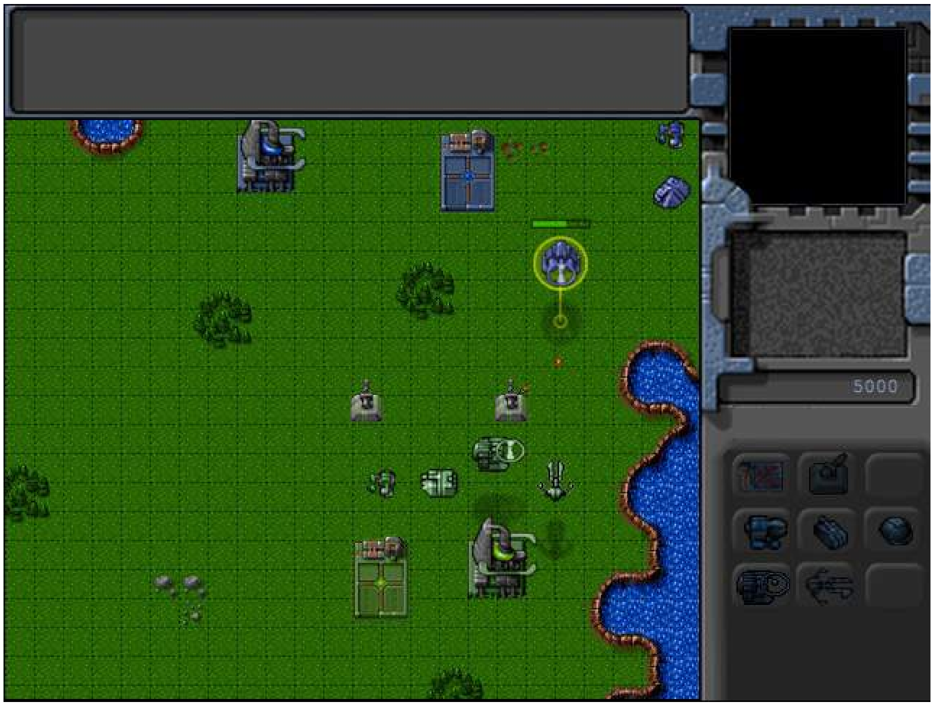


***Figure 9-2.** Aircraft attacking each other*

You can command an aircraft to attack an enemy or guard a friend by right-clicking after selecting the aircraft. The chopper can attack both land and air units, while the wraith can attack only air units.

We will typically use the `sentry`, `hunt`, and `patrol` orders to give the computer AI a slight advantage and make the game more challenging for the player. The player will not have access to these orders.

■ **Tip** We can easily implement patrol for the player by modifying the `click` method to send a patrol command if a modifier key (such as Ctrl or Shift) is pressed when the player right-clicks the ground.

Next we will implement combat-based orders for vehicles.

Combat-Based Orders for Vehicles

The combat-based order states for vehicles will be very similar to the order states for aircraft.

- `attack`: Move within range of a target and shoot at it.
- `stand`: Stay in one place and attack any enemy that comes close.
- `guard`: Follow a friendly unit and shoot at any enemy that comes close.
- `hunt`: Actively seek out enemies anywhere on the map and attack them.

- `patrol`: Move between two points and shoot at any enemy that comes within range.
- `sentry`: Stay in one place an attack enemies slightly more aggressively than in `stand` mode.

We will implement these states by modifying the default `processOrders()` method inside the `vehicles` object, as shown in Listing 9-12.

Listing 9-12. Implementing Combat Orders for vehicles (vehicles.js)

```
isValidTarget:isValidTarget,
findTargetsInSight:findTargetsInSight,
processOrders:function(){
    this.lastMovementX = 0;
    this.lastMovementY = 0;
    if(this.reloadTimeLeft){
        this.reloadTimeLeft--;
    }
    var target;
    switch (this.orders.type){
        case "move":
            // Move toward destination until distance from destination is less than vehicle radius
            var distanceFromDestinationSquared = (Math.pow(this.orders.to.x-this.x,2) +
Math.pow(this.orders.to.y-this.y,2));
            if (distanceFromDestinationSquared < Math.pow(this.radius/game.gridSize,2)) {
                //Stop when within one radius of the destination
                this.orders = {type:"stand"};
                return;
            } else if (distanceFromDestinationSquared <Math.pow(this.radius*3/game.gridSize,2)) {
                //Stop when within 3 radius of the destination if colliding with something
                this.orders = {type:"stand"};
                return;
            } else {
                if (this.colliding && (Math.pow(this.orders.to.x-this.x,2) +
Math.pow(this.orders.to.y-this.y,2))<Math.pow(this.radius*5/game.gridSize,2)) {
                    // Count collsions within 5 radius distance of goal
                    if (!this.orders.collisionCount){
                        this.orders.collisionCount = 1
                    } else {
                        this.orders.collisionCount ++;
                    }
                    // Stop if more than 30 collisions occur
                    if (this.orders.collisionCount > 30) {
                        this.orders = {type:"stand"};
                        return;
                    }
                }
                var moving = this.moveTo(this.orders.to);
                // Pathfinding couldn't find a path so stop
                if(!moving){
                    this.orders = {type:"stand"};
                    return;
                }
            }
            break;
        case "deploy":
            // If oilfield has been used already, then cancel order
            if(this.orders.to.lifeCode == "dead"){
                this.orders = {type:"stand"};
                return;
            }
            // Move to middle of oil field
            var target = {x:this.orders.to.x+1,y:this.orders.to.y+0.5,type:"terrain"};
            var distanceFromTargetSquared = (Math.pow(target.x-this.x,2) +
Math.pow(target.y-this.y,2));
            if (distanceFromTargetSquared<Math.pow(this.radius*2/game.gridSize,2)) {
                // After reaching oil field, turn harvester to point toward left (direction 6)
                var difference = angleDiff(this.direction,6,this.directions);
                var turnAmount = this.turnSpeed*game.turnSpeedAdjustmentFactor;
                if (Math.abs(difference)>turnAmount){
                    this.direction = wrapDirection(this.direction+turnAmount*Math.abs(difference)/
difference,this.directions);
                } else {
                    // Once it is pointing to the left, remove the harvester and oil field and
deploy a harvester building
                    game.remove(this.orders.to);
                    this.orders.to.lifeCode="dead";
                    game.remove(this);
                    this.lifeCode="dead";
                    game.add({type:"buildings", name:"harvester", x:this.orders.to.x,
y:this.orders.to.y, action:"deploy", team:this.team});
                }
            } else {
                var moving = this.moveTo(target);
                // Pathfinding couldn't find a path so stop
                if(!moving){
                    this.orders = {type:"stand"};
                }
            }
            break;
        case "stand":
            var targets = this.findTargetsInSight();
            if(targets.length>0){
                this.orders = {type:"attack",to:targets[0]};
            }
            break;
        case "sentry":
            var targets = this.findTargetsInSight(2);
            if(targets.length>0){
                this.orders = {type:"attack",to:targets[0],nextOrder:this.orders};
            }
            break;
        case "hunt":
            var targets = this.findTargetsInSight(100);
            if(targets.length>0){
                this.orders = {type:"attack",to:targets[0],nextOrder:this.orders};
            }
            break;
        case "attack":
            if(this.orders.to.lifeCode == "dead" || !this.isValidTarget(this.orders.to)){
                if (this.orders.nextOrder){
                    this.orders = this.orders.nextOrder;
                } else {
                    this.orders = {type:"stand"};
                }
                return;
            }
            if ((Math.pow(this.orders.to.x-this.x,2) + Math.pow(this.orders.to.y-this.y,2))
<Math.pow(this.sight,2)) {
                //Turn toward target and then start attacking when within range of the target
                var newDirection = findFiringAngle(this.orders.to,this,this.directions);
                var difference = angleDiff(this.direction,newDirection,this.directions);
                var turnAmount = this.turnSpeed*game.turnSpeedAdjustmentFactor;
                if (Math.abs(difference)>turnAmount){
                    this.direction = wrapDirection(this.direction + turnAmount*Math.abs(difference)/
difference, this.directions);
                    return;
                } else {
                    this.direction = newDirection;
                    if(!this.reloadTimeLeft){
                        this.reloadTimeLeft = bullets.list[this.weaponType].reloadTime;
                        var angleRadians = -(Math.round(this.direction)/this.directions)*2*Math.PI ;
                        var bulletX = this.x- (this.radius*Math.sin(angleRadians)/game.gridSize);
                        var bulletY = this.y- (this.radius*Math.cos(angleRadians)/game.gridSize);
                        var bullet = game.add({name:this.weaponType,type:"bullets",x:bulletX,y:
bulletY, direction:newDirection, target:this.orders.to});
                    }
                }
            } else {
                var moving = this.moveTo(this.orders.to);
                // Pathfinding couldn't find a path so stop
                if(!moving){
                    this.orders = {type:"stand"};
                    return;
                }
            }
            break;
        case "patrol":
            var targets = this.findTargetsInSight(1);
            if(targets.length>0){
                this.orders = {type:"attack",to:targets[0],nextOrder:this.orders};
                return;
            }
            if ((Math.pow(this.orders.to.x-this.x,2) +
Math.pow(this.orders.to.y-this.y,2))<Math.pow(this.radius*4/game.gridSize,2)) {
                var to = this.orders.to;
                this.orders.to = this.orders.from;
                this.orders.from = to;
            } else {
                this.moveTo(this.orders.to);
            }
            break;
        case "guard":
            if(this.orders.to.lifeCode == "dead"){
                if (this.orders.nextOrder){
                    this.orders = this.orders.nextOrder;
                } else {
                    this.orders = {type:"stand"};
                }
                return;
            }
            if ((Math.pow(this.orders.to.x-this.x,2) + Math.pow(this.orders.to.y-this.y,2))
<Math.pow(this.sight-2,2)) {
                var targets = this.findTargetsInSight(1);
                if(targets.length>0){
                    this.orders = {type:"attack",to:targets[0],nextOrder:this.orders};
                    return;
                }
            } else {
                this.moveTo(this.orders.to);
            }
            break;
    }
},
```

The implementation of the states is almost the same as for aircraft. If we run the code now, we should be able to attack with the vehicles, as shown in Figure 9-3.

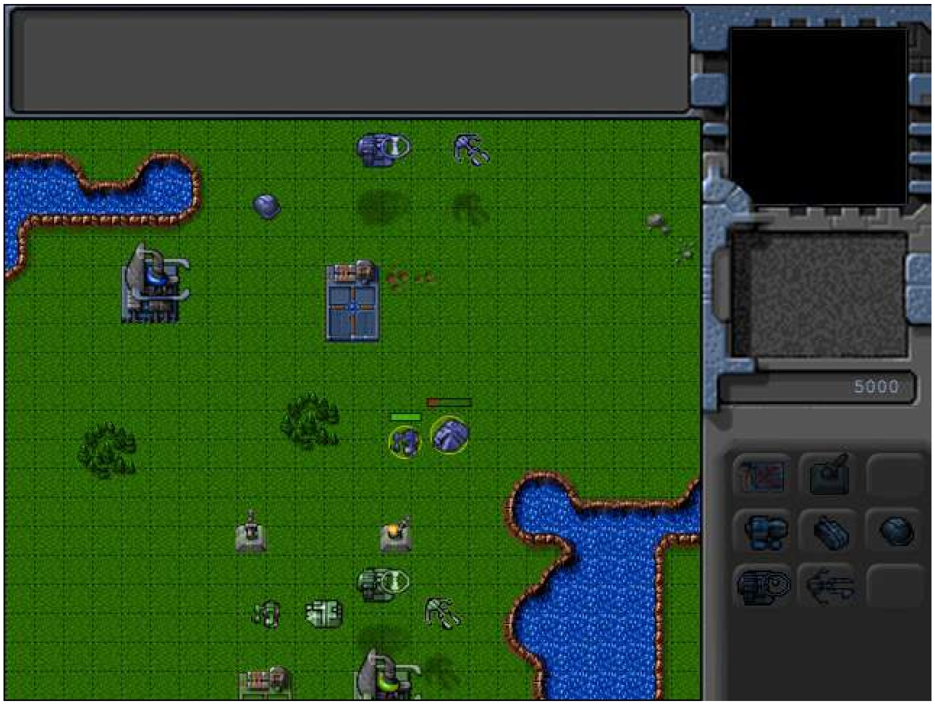


***Figure 9-3.** Attacking with the vehicles*

We can now attack with vehicles, aircraft, or turrets.

You may have noticed that while the opposing team's units attack when you come close, they are still very easily defeated. Now that the combat system is in place, we will explore ways to make the enemy more intelligent and the game more challenging.

Building Intelligent Enemy

The primary goal in building an intelligent enemy AI is to make sure that the person playing the game finds it reasonably challenging and has a fun experience completing the level. An important thing to realize about RTS games, especially the single-player campaign, is that the enemy AI doesn't need to be a grandmaster-level chess player. The fact is, we can provide the player with a very compelling experience using only a combination of combat order states and conditional scripted events.

Typically, the "intelligent" way to behave for the AI will vary with each level.

In a simple level where there are no production facilities and only ground units, the only possible behavior is to drive up to the enemy units and attack them. A combination of `patrol` and `sentry` orders is usually more than enough to achieve this. We could also make the level interesting by attacking the player at a specific time or when a certain event occurs (for example, when the player arrives at a certain location or constructs a particular building).

In a more complex level, we might make the enemy challenging by constructing and sending in waves of enemies at specific intervals using timed triggers and the `hunt` order.

We can see some of these ideas at work by adding a few more items and triggers to the map, as shown in Listing 9-13.

Listing 9-13. Adding Triggers and Items to Make the Level Challenging (maps.js)

```
/* Entities to be added */
"items":[
    {"type":"buildings","name":"base","x":11,"y":14,"team":"blue"},
    {"type":"buildings","name":"starport","x":18,"y":14,"team":"blue"},

    {"type":"vehicles","name":"harvester","x":16,"y":12,"team":"blue","direction":3},
    {"type":"terrain","name":"oilfield","x":3,"y":5,"action":"hint"},

    {"type":"terrain","name":"bigrocks","x":19,"y":6},
    {"type":"terrain","name":"smallrocks","x":8,"y":3},

    {"type":"vehicles","name":"scout-tank","x":26,"y":14,"team":"blue","direction":4},
    {"type":"vehicles","name":"heavy-tank","x":26,"y":16,"team":"blue","direction":5},
    {"type":"aircraft","name":"chopper","x":20,"y":12,"team":"blue","direction":2},
    {"type":"aircraft","name":"wraith","x":23,"y":12,"team":"blue","direction":3},


    {"type":"buildings","name":"ground-turret","x":15,"y":23,"team":"green"},
    {"type":"buildings","name":"ground-turret","x":20,"y":23,"team":"green"},

    {"type":"vehicles","name":"scout-tank","x":16,"y":26,"team":"green","direction":4,"orders":
{"type":"sentry"}},
    {"type":"vehicles","name":"heavy-tank","x":18,"y":26,"team":"green","direction":6,"orders":
{"type":"sentry"}},

    {"type":"aircraft","name":"chopper","x":20,"y":27,"team":"green","direction":2,"orders":
{"type":"hunt"}},

    {"type":"aircraft","name":"wraith","x":22,"y":28,"team":"green","direction":3,"orders":
{"type":"hunt"}},

    {"type":"buildings","name":"base","x":19,"y":28,"team":"green"},
    {"type":"buildings","name":"starport","x":15,"y":28,"team":"green","uid":-1},
],

/* Economy Related*/
"cash":{
    "blue":5000,
    "green":5000
},

/* Conditional and Timed Trigger Events */
"triggers":[
/* Timed Events*/
    {"type":"timed","time":1000,
        "action":function(){
            game.sendCommand([-1],{type:"construct-unit",details:{type:"aircraft",name:"wraith",orde
rs:{"type":"patrol","from":{"x":22,"y":30},"to":{"x":15,"y":21}}}});
        }
    },
    {"type":"timed","time":5000,
        "action":function(){
            game.sendCommand([-1],{type:"construct-unit", details:{type:"aircraft",name:"chopper",
orders:{"type":"patrol","from":{"x":15,"y":30},"to":{"x":22,"y":21}}}});
        }
    },
    {"type":"timed","time":10000,
        "action":function(){
            game.sendCommand([-1],{type:"construct-unit",details:{type:"vehicles",name:"heavy-tank",
orders:{"type":"patrol","from":{"x":15,"y":30},"to":{"x":22,"y":21}}}});
        }
    },
    {"type":"timed","time":15000,
        "action":function(){
            game.sendCommand([-1],{type:"construct-unit",details:{type:"vehicles",name:"scout-tank",
orders:{"type":"patrol","from":{"x":22,"y":30},"to":{"x":15,"y":21}}}});
        }
    },
    {"type":"timed","time":60000,
        "action":function(){
            game.showMessage("AI","Now every enemy unit is going to attack you in a wave.");
            var units = [];
            for (var i=0; i < game.items.length; i++) {
                var item = game.items[i];
                if (item.team == "green" && (item.type == "vehicles"|| item.type == "aircraft")){
                    units.push(item.uid);
                }
            };
            game.sendCommand(units,{type:"hunt"});
        }
    },
],
```

The first thing we do is order an enemy chopper and a wraith to hunt as soon as the game starts. Next, we assign a UID of –1 to the enemy starport and set a few timed triggers to build different types of patrolling units every few seconds.

Finally, after 60 seconds, we command all enemy units to hunt and notify the player using the `showMessage()` method.

If we run the code now, we can expect the AI to defend itself fairly well and attack very aggressively at the end of 60 seconds, as shown in Figure 9-4.

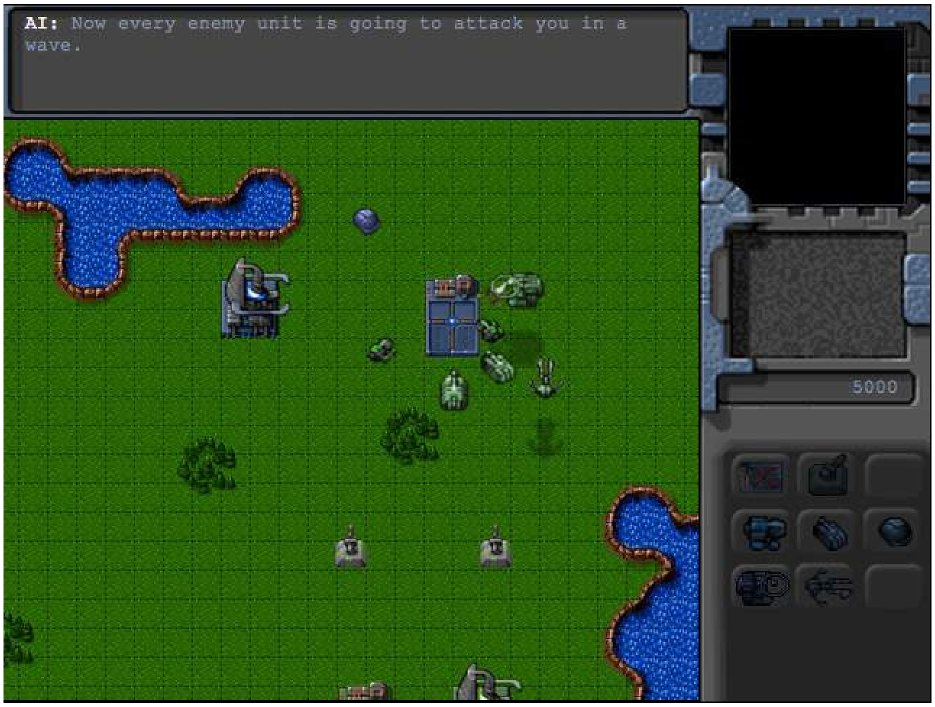

AI: Now every enemy unit is going to attack you in a wave.

Figure 9-4. *Computer AI aggressively attacking player*

Obviously, this is a fairly contrived example. No one will want to play a game where they get attacked this brutally within the first minute of playing. However, as this example illustrates, we can make the game as easy or as challenging as we need just by adjusting these triggers and orders.

■ **Tip** You can implement separate sets of triggers and starting items depending on a difficulty setting so that the player can play easy or challenging versions of the same campaign based on the setting selected.

Now that we have implemented the combat system and explored ways to make the game AI challenging, the last thing we will look at in this chapter is adding a fog of war.

Adding a Fog of War

The fog of war is typically a dark, colored shroud that covers all unexplored terrain within the map. As player units move around the map, the fog is cleared anywhere that the unit can see.

This introduces elements of exploration and intrigue to the game. The ability to hide under the fog allows the use of strategies such as hidden bases, ambushes, and sneak attacks.

Some RTS games permanently remove the fog once an area is explored, while others clear the fog only in areas within sight of a player unit and bring back the fog once the unit leaves the area. For our game, we will be using the second implementation.

Defining the Fog Object

We will start by defining a new `fog` object inside `fog.js`, as shown in Listing 9-14.

Listing 9-14. Implementing the fog Object (fog.js)

```
var fog = {
    grid:[],
    canvas:document.createElement('canvas'),
    initLevel:function(){
        // Set fog canvas to the size of the map
        this.canvas.width = game.currentLevel.mapGridWidth*game.gridSize;
        this.canvas.height = game.currentLevel.mapGridHeight*game.gridSize;

        this.context = this.canvas.getContext('2d');

        // Set the fog grid for the player to array with all values set to 1
        this.defaultFogGrid = [];
        for (var i=0; i < game.currentLevel.mapGridHeight; i++) {
           this.defaultFogGrid[i] = [];
           for (var j=0; j < game.currentLevel.mapGridWidth; j++) {
               this.defaultFogGrid[i][j] = 1;
           };
        };

    },
    isPointOverFog:function(x,y){
        // If the point is outside the map bounds consider it fogged
        if(y<0 || y/game.gridSize >= game.currentLevel.mapGridHeight || x<0 || x/game.gridSize >=
game.currentLevel.mapGridWidth ){
            return true;
           }
        // If not, return value based on the player's fog grid
        return this.grid[game.team][Math.floor(y/game.gridSize)][Math.floor(x/game.gridSize)] == 1;
    },
    animate:function(){
        // Fill fog with semi solid black color over the map
        this.context.drawImage(game.currentMapImage,0,0)
        this.context.fillStyle = 'rgba(0,0,0,0.8)';
        this.context.fillRect(0,0,this.canvas.width,this.canvas.height);

        // Initialize the players fog grid
        this.grid[game.team] = $.extend(true,[],this.defaultFogGrid);

        // Clear all areas of the fog where a player item has vision
        fog.context.globalCompositeOperation = "destination-out";
        for (var i = game.items.length - 1; i >= 0; i--){
            var item = game.items[i];
            var team = game.team;
                if (item.team == team && !item.keepFogged){
                    var x = Math.floor(item.x );
                    var y = Math.floor(item.y );
```

```
                      var x0 = Math.max(0,x-item.sight+1);
                      var y0 = Math.max(0,y-item.sight+1);
                      var x1 = Math.min(game.currentLevel.mapGridWidth-1, x+item.sight-1+
(item.type=="buildings"?item.baseWidth/game.gridSize:0));
                      var y1 = Math.min(game.currentLevel.mapGridHeight-1, y+item.sight-1+
(item.type=="buildings"?item.baseHeight/game.gridSize:0));
                    for (var j=x0; j <= x1; j++) {
                        for (var k=y0; k <= y1; k++) {
                            if ((j>x0 && j<x1) || (k>y0 && k<y1)){
                                if(this.grid[team][k][j]){
                                    this.context.fillStyle = 'rgba(100,0,0,0.9)';
                                    this.context.beginPath();
                                    this.context.arc(j*game.gridSize+12, k*game.gridSize+12,
16, 0, 2*Math.PI, false);
                                    this.context.fill();
                                    this.context.fillStyle = 'rgba(100,0,0,0.7)';
                                    this.context.beginPath();
                                    this.context.arc(j*game.gridSize+12,
k*game.gridSize+12,18, 0, 2*Math.PI, false);
                                    this.context.fill();

                                    this.context.fillStyle = 'rgba(100,0,0,0.5)';
                                    this.context.beginPath();
                                    this.context.arc(j*game.gridSize+12, k*game.gridSize+12,
24, 0, 2*Math.PI, false);
                                    this.context.fill();

                                }
                                this.grid[team][k][j] = 0;
                            }
                        };
                    };
                }
            };
            fog.context.globalCompositeOperation = "source-over";
        },
        draw:function(){
            game.foregroundContext.drawImage(this.canvas,game.offsetX, game.offsetY, game.canvasWidth,
game.canvasHeight, 0,0,game.canvasWidth,game.canvasHeight);
        }
}
```

We start by defining a canvas inside the `fog` object. The `initLevel()` method resizes the `canvas` object to the size of the current map and defines a `fogGrid` array that has the same dimensions as the map with all its elements set to 1.

Within the `animate()` method, we first initialize the fog to the map background with a semi-transparent black layer over it. This way, fogged areas of the map show up as darkened background terrain.

We then iterate through each of the items in the game and clear the `fog` array and the fog canvas around the player's items based on their `sight` property. We do not clear the fog for items that are the opposing player's or that have a `keepFogged` attribute set to true.

Finally, the `draw()` method draws the fog canvas onto the `game.foregroundContext` context using the same offsets that we used when drawing the map onto the `game.backgroundContext` context.

Drawing the Fog

Now that we have defined the fog object, we will start by adding a reference to fog.js inside the head section of index.html, as shown in Listing 9-15.

Listing 9-15. Adding a Reference to the fog Object (index.html)

```
<script src="js/fog.js" type="text/javascript" charset="utf-8"></script>
```

Next, we need to initialize the fog once the level is loaded. We will do this by calling the fog.initLevel() method inside the singleplayer object's play() method, as shown in Listing 9-16.

Listing 9-16. Initializing the fog Object for the Level (singleplayer.js)

```
play:function(){
    fog.initLevel();
    game.animationLoop();
    game.animationInterval = setInterval(game.animationLoop,game.animationTimeout);
    game.start();
},
```

Next we need to modify the game object's animationLoop() and drawingLoop() methods to call fog.animate() and fog.draw() respectively, as shown in Listing 9-17.

Listing 9-17. Calling fog.animate() and fog.draw() (game.js)

```
animationLoop:function(){
    // Animate the Sidebar
    sidebar.animate();

    // Process orders for any item that handles it
    for (var i = game.items.length - 1; i >= 0; i--){
        if(game.items[i].processOrders){
            game.items[i].processOrders();
        }
    };

    // Animate each of the elements within the game
    for (var i = game.items.length - 1; i >= 0; i--){
        game.items[i].animate();
    };

    // Sort game items into a sortedItems array based on their x,y coordinates
    game.sortedItems = $.extend([],game.items);
    game.sortedItems.sort(function(a,b){
        return b.y-a.y + ((b.y==a.y)?(a.x-b.x):0);
    });

    fog.animate();
    game.lastAnimationTime = (new Date()).getTime();
},
drawingLoop:function(){
    // Handle Panning the Map
    game.handlePanning();

    // Check the time since the game was animated and calculate a linear interpolation factor
(-1 to 0)
    // since drawing will happen more often than animation
    game.lastDrawTime = (new Date()).getTime();
       if (game.lastAnimationTime){
           game.drawingInterpolationFactor = (game.lastDrawTime-game.lastAnimationTime)/
game.animationTimeout - 1;
           if (game.drawingInterpolationFactor>0){ // No point interpolating beyond the next
animation loop...
               game.drawingInterpolationFactor = 0;
           }
       } else {
        game.drawingInterpolationFactor = -1;

    }

    // Since drawing the background map is a fairly large operation,
    // we only redraw the background if it changes (due to panning)
    if (game.refreshBackground){
        game.backgroundContext.drawImage(game.currentMapImage,game.offsetX, game.offsetY,
game.canvasWidth, game.canvasHeight, 0,0,game.canvasWidth,game.canvasHeight);
        game.refreshBackground = false;
    }

    // Clear the foreground canvas
    game.foregroundContext.clearRect(0,0,game.canvasWidth,game.canvasHeight);

    // Start drawing the foreground elements
    for (var i = game.sortedItems.length - 1; i >= 0; i--){
        if (game.sortedItems[i].type != "bullets"){
            game.sortedItems[i].draw();
        }
    };

    // Draw the bullets on top of all the other elements
    for (var i = game.bullets.length - 1; i >= 0; i--){
        game.bullets[i].draw();
    };

    fog.draw();

    // Draw the mouse
    mouse.draw()
    // Call the drawing loop for the next frame using request animation frame
    if (game.running){
        requestAnimationFrame(game.drawingLoop);
    }
},
```

If we run the code now, we should see the entire map shrouded in a fog of war, as shown in Figure 9-5.

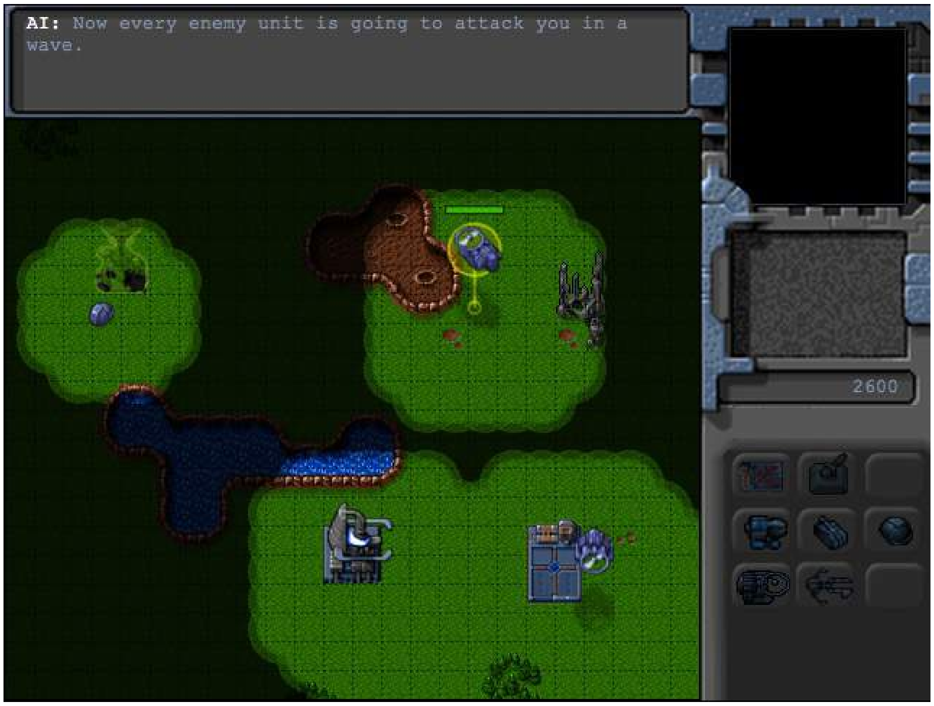

AI: Now every enemy unit is going to attack you in a wave.

Figure 9-5. *Map shrouded in fog of war*

You will see that the fog is uncovered around friendly units and buildings. Also, the fogged area shows the original terrain but does not show any units under it.

The same enemy attack feels much scarier when we have no idea about the size or the location of the opposing army. Before we wrap up the chapter, we will make a few additions, starting with making the fogged areas unbuildable.

Making Fogged Areas Unbuildable

The first change we will make is to prevent the deploying of buildings on fogged areas by making fogged areas unbuildable. We will modify the `sidebar` object's `animate()` method, as shown in Listing 9-18.

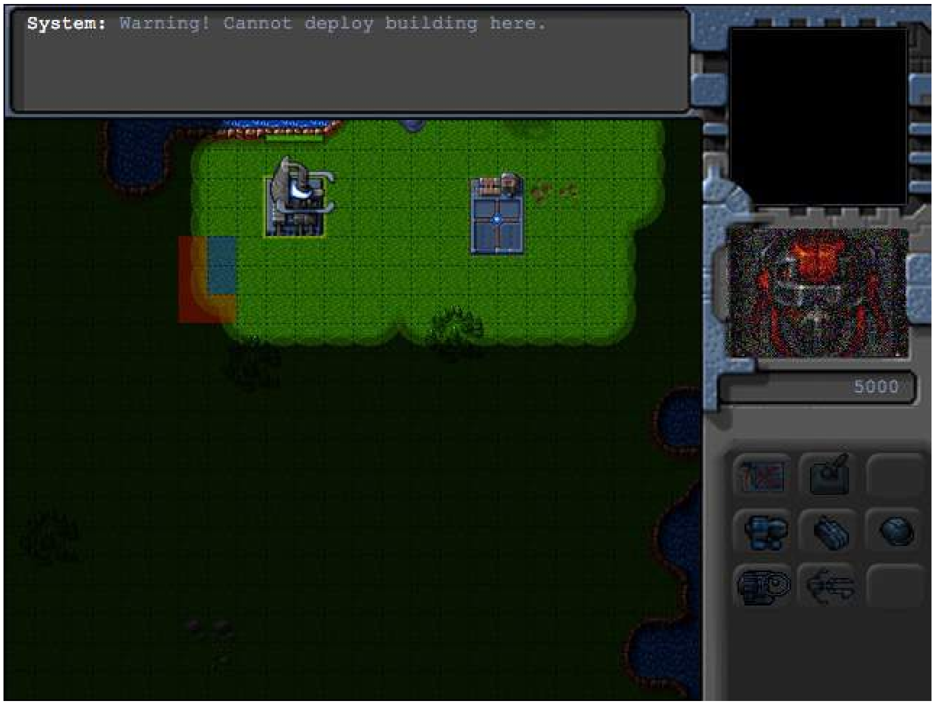
Listing 9-18. Making Fogged Areas Unbuildable (sidebar.js)

```
animate:function(){
    // Display the current cash balance value
    $('#cash').html(game.cash[game.team]);

    //  Enable or disable buttons as appropriate
    this.enableSidebarButtons();

    if (game.deployBuilding){
        // Create the buildable grid to see where building can be placed
        game.rebuildBuildableGrid();
        // Compare with buildable grid to see where we need to place the building
        var placementGrid = buildings.list[game.deployBuilding].buildableGrid;
        game.placementGrid = $.extend(true,[],placementGrid);
        game.canDeployBuilding = true;
        for (var i = game.placementGrid.length - 1; i >= 0; i--){
            for (var j = game.placementGrid[i].length - 1; j >= 0; j--){
                if(game.placementGrid[i][j] &&
                    (mouse.gridY+i>= game.currentLevel.mapGridHeight || mouse.gridX+j>=
game.currentLevel.mapGridWidth
                        || game.currentMapBuildableGrid[mouse.gridY+i][mouse.gridX+j]==1 ||
fog.grid[game.team][mouse.gridY+i][mouse.gridX+j]==1)){
                    game.canDeployBuilding = false;
                    game.placementGrid[i][j] = 0;
                }
            };
        };
    }
},
```

We add an extra condition for testing the fog grid when creating the `placementGrid` array so that a fogged grid square is no longer buildable. If we run the game and try to build on a fogged area, we should see a warning, as shown in Figure 9-6.



System: Warning! Cannot deploy building here.

Figure 9-6. *Cannot deploy buildings on fogged areas*

As you can see, the building deploy grid turns red on fogged areas to indicate that the player cannot build there. If you still try to click a fogged area, you will get a system warning.

Next we will make sure that the player cannot select or detect a building or unit that is under the fog. We do this by modifying the `mouse` object's `pointUnderFog()` method, as shown in Listing 9-19.

Listing 9-19. Hiding Objects Under the Fog (mouse.js)

```
itemUnderMouse:function(){
    if(fog.isPointOverFog(mouse.gameX,mouse.gameY)){
        return;
    }
    for (var i = game.items.length - 1; i >= 0; i--){
        var item = game.items[i];
        if (item.type=="buildings" || item.type=="terrain"){
            if(item.lifeCode != "dead"
                && item.x<= (mouse.gameX)/game.gridSize
                && item.x >= (mouse.gameX - item.baseWidth)/game.gridSize
                && item.y<= mouse.gameY/game.gridSize
                && item.y >= (mouse.gameY - item.baseHeight)/game.gridSize
                ){
                    return item;
            }
        } else if (item.type=="aircraft"){
            if (item.lifeCode != "dead" &&
                Math.pow(item.x-mouse.gameX/game.gridSize,2)+Math.pow(item.y-
(mouse.gameY+item.pixelShadowHeight)/game.gridSize,2) < Math.pow((item.radius)/game.gridSize,2)){
                return item;
            }
       }else {
            if (item.lifeCode != "dead" && Math.pow(item.x-mouse.gameX/game.gridSize,2) +
Math.pow(item.y-mouse.gameY/game.gridSize,2) < Math.pow((item.radius)/game.gridSize,2)){
                return item;
            }
        }
    }
},
```

We check whether the point under the mouse is fogged and, if so, return nothing. With this last change, we now have a working fog of war in our game.

Summary

In this chapter, we implemented a combat system for our game. We started by defining a `bullets` object with different types of bullets. We then added several combat-based order states to our turrets, aircraft, and vehicles. We used these orders along with the triggers system we defined in the previous chapter to create a fairly challenging enemy. Finally, we implemented a fog of war.

Our game now has most of the essential elements of an RTS. In the next chapter, we will polish our game framework by adding sound. We will then use this framework to build a few interesting levels and wrap up our single-player campaign.