Chapter 7

Intelligent Unit Movement

In the previous chapter, we built a framework for animating and drawing entities within our game and then added different types of buildings, vehicles, aircraft, and terrain to it. Finally, we added the ability to select these entities.

In this chapter, we will add a framework to give selected units commands and to get the entities to follow orders. We will then implement the most basic of these orders: unit movement by using a combination of pathfinding and steering algorithms to move our units intelligently.

Now let's get started. We will use the code from Chapter 6 as a starting point.

Commanding Units

We will command units using a convention that has now become standard within most modern RTS games. We will select units using left-clicks and command them by using right-clicks.

Right-clicking a navigable spot on the map will command selected units to move to the spot. Right-clicking an enemy unit or building will command all selected units that can attack to attack the enemy. Right-clicking a friendly unit will tell all selected units to follow it around and protect it. And finally, right-clicking an oil field with a harvester vehicle selected will tell the harvester to move to the oil field and deploy on it.

The first thing we need to do is modify the `mouse` object's `click()` method inside `mouse.js` to handle right-click events, as shown in Listing 7-1.

Listing 7-1. Modifying click() to Handle Commands on Right-Click (mouse.js)

```
click:function(ev,rightClick){
    // Player clicked inside the canvas

    var clickedItem = this.itemUnderMouse();
    var shiftPressed = ev.shiftKey;

    if (!rightClick){ // Player left clicked
        if (clickedItem){
            // Pressing shift adds to existing selection. If shift is not pressed, clear
existing selection
            if(!shiftPressed){
                game.clearSelection();
            }
            game.selectItem(clickedItem,shiftPressed);
        }
    } else { // Player right-clicked
        // Handle actions like attacking and movement of selected units
        var uids = [];
```

```
        if (clickedItem){ // Player right-clicked on something
            if (clickedItem.type != "terrain"){
                if (clickedItem.team != game.team){ // Player right-clicked on an enemy item
                    // Identify selected items from players team that can attack
                    for (var i = game.selectedItems.length - 1; i >= 0; i--){
                        var item = game.selectedItems[i];
                        if(item.team == game.team && item.canAttack){
                            uids.push(item.uid);
                        }
                    };
                    // then command them to attack the clicked item
                    if (uids.length>0){
                        game.sendCommand(uids,{type:"attack",toUid:clickedItem.uid});
                    }
                } else  { // Player right-clicked on a friendly item
                    //identify selected items from players team that can move
                    for (var i = game.selectedItems.length - 1; i >= 0; i--){
                        var item = game.selectedItems[i];
                        if(item.team == game.team && (item.type == "vehicles" ||
item.type == "aircraft")){
                            uids.push(item.uid);
                        }
                    };
                    // then command them to guard the clicked item
                    if (uids.length>0){
                        game.sendCommand(uids,{type:"guard", toUid:clickedItem.uid});
                    }
                }
            } else if (clickedItem.name == "oilfield"){ // Player right licked on an oilfield
                // identify the first selected harvester from players team (since only one can
deploy at a time)
                for (var i = game.selectedItems.length - 1; i >= 0; i--){
                    var item = game.selectedItems[i];
                    if(item.team == game.team && (item.type == "vehicles" && item.name ==
"harvester")){
                        uids.push(item.uid);
                        break;
                    }
                };
                // then command it to deploy on the oilfield
                if (uids.length>0){
                    game.sendCommand(uids,{type:"deploy",toUid:clickedItem.uid});
                }
            }
        } else { // Player clicked on the ground
            //identify selected items from players team that can move
            for (var i = game.selectedItems.length - 1; i >= 0; i--){
                var item = game.selectedItems[i];
                if(item.team == game.team && (item.type == "vehicles" || item.type == "aircraft")){
                    uids.push(item.uid);
                }
            };
```

```
            // then command them to move to the clicked location
            if (uids.length>0){
                game.sendCommand(uids, {type:"move", to:{x:mouse.gameX/game.gridSize,
y:mouse.gameY/game.gridSize}});
            }
        }
    }
},
```

When the player right-clicks inside the game map, we first check to see whether the mouse is above an object.

If the player has not clicked an object, we call the `game.sendCommand()` method to send a `move` order to all friendly vehicles and aircraft that are selected.

If the player has clicked an object, we similarly send either an `attack`, `guard`, or `deploy` command to the appropriate units. We also pass the UID of the clicked item as a parameter called `toUid` within the order.

With the right-click logic in place, we now have to implement methods for sending and receiving game commands.

Sending and Receiving Commands

We could have implemented sending commands by modifying the `orders` property of selected items inside the `click()` method that we modified earlier. However, we are going to use a slightly more complex implementation.

Any clicking action that generates a command will call the `game.sendCommand()` method. The `sendCommand()` method will pass the call to either the `singleplayer` or `multiplayer` object. These objects will then send the command details back to the `game.processCommand()` method. Within the `game.processCommand()` method, we will update the orders for all the appropriate objects. We will start by adding these methods to the `game` object inside `game.js`, as shown in Listing 7-2.

Listing 7-2. Implementing sendCommand() and processCommand() (game.js)

```
// Send command to either singleplayer or multiplayer object
sendCommand:function(uids,details){
    if (game.type=="singleplayer"){
         singleplayer.sendCommand(uids,details);
    } else {
        multiplayer.sendCommand(uids,details);
    }
},
getItemByUid:function(uid){
    for (var i = game.items.length - 1; i >= 0; i--){
        if(game.items[i].uid == uid){
            return game.items[i];
        }
    };
},
// Receive command from singleplayer or multiplayer object and send it to units
processCommand:function(uids,details){
    // In case the target "to" object is in terms of uid, fetch the target object
    var toObject;
    if (details.toUid){
        toObject = game.getItemByUid(details.toUid);
        if(!toObject || toObject.lifeCode=="dead"){
```

```
            // To object no longer exists. Invalid command
            return;
        }
    }

    for (var i in uids){
        var uid = uids[i];
        var item = game.getItemByUid(uid);
        //if uid is a valid item, set the order for the item
        if(item){
            item.orders = $.extend([],details);
            if(toObject) {
                item.orders.to = toObject;
            }
        }
    };
},
```

The `sendCommand()` method passes the call to either the `singleplayer` or `multiplayer` object's `sendCommand()` method based on the game type. Using this layer of abstraction allows us to use the same code for both single-player and multiplayer while handling the commands differently.

While the single-player version of `sendCommand()` will just call `processCommand()` back immediately, the multiplayer version will send the command to the server, which will then forward the command to all the players at the same time.

We also implement the `getItemByUid()` method that looks up item UIDs and returns entity objects.

We pass UIDs instead of actual game objects to the `sendCommand()` method because of the multiplayer version of the game. A typical item object contains a lot of details for animating and drawing the objects such as methods, sprite sheet images, and all the item properties. While needed for drawing the item, transmitting this extra data to the server and getting it back is a waste of bandwidth and quite unnecessary, especially since the entire object can be replaced by a single integer (the UID).

The `processCommand()` method first looks up any `toUid` property and gets the resulting item. If no item with the UID exists, it assumes the command is invalid and ignores the command. The method then looks up the items passed in the `uids` array and sets their `orders` object to a copy of the order details provided in the parameters.

The next thing we will do is implement the `singlePlayer` object's `sendCommand()` method inside `singleplayer.js`, as shown in Listing 7-3.

Listing 7-3. Implementing the Single-Player sendCommand() Method (singleplayer.js)

```
sendCommand:function(uids,details){
    game.processCommand(uids,details);
}
```

As you can see, the implementation of `sendCommand()` is fairly simple. We merely forward the call right to `game.processCommand()`. However, if we wanted, we could also use this method to add functionality for saving game commands, along with details about the currently running animation cycle, to implement the ability to replay saved games.

Now that we have set up a mechanism for commanding units and setting their orders, we need to set up a way for the units to process these orders and execute them.

Processing Orders

Our implementation for processing orders will be fairly simple. We will implement a method called `processOrders()` for every entity that needs it and call the `processOrders()` method for all game items from inside the game animation loop.

We will start by modifying the `game` object's `animationLoop()` method inside `game.js`, as shown in Listing 7-4.

Listing 7-4. Calling processOrders() from Inside the Animation Loop (game.js)

```
animationLoop:function(){
    // 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);
    });
},
```

The code iterates through every game item and calls the item's `processOrders()` method if it exists. Now, we can implement the `processOrders()` method for the game entities one by one and watch as these entities start obeying our commands.

Let's start by implementing movement for aircraft.

Implementing Aircraft Movement

Unlike land vehicles, moving aircraft is fairly simple since aircraft are not affected by terrain, buildings, or other vehicles. When an aircraft is given a `move` order, it will just turn toward the destination and then move forward in a straight line. Once the aircraft nears its destination, it will go back to its `float` state.

We will implement this as a default `processOrders()` method for aircraft inside `aircraft.js`, as shown in Listing 7-5.

Listing 7-5. Movement in the Aircraft Object's Default processOrders() Method (aircraft.js)

```
processOrders:function(){
    this.lastMovementX = 0;
    this.lastMovementY = 0;
    switch (this.orders.type){
        case "move":
```

```
            // Move towards 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;
    }
},
```

We first reset two variables related to movement that we will use later. We then check the order type inside a `case` statement.

In case the order type is `move`, we call the `moveTo()` method until the aircraft's distance from the destination (stored in the `to` parameter) is less than the aircraft's radius. Once the aircraft has reached its destination, we change the order back to `float`.

Right now, we have implemented only one order. Any time the aircraft gets an order it doesn't know how to handle, it will continue floating at its current location. We will be implementing more orders as we go along.

The next thing we will do is implement a default `moveTo()` method that will be used by both the aircraft (see Listing 7-6).

Listing 7-6. The Aircraft Object's Default moveTo() Method (aircraft.js)

```
moveTo:function(destination){
    // Find out where we need to turn to get to destination
    var newDirection = findAngle(destination,this,this.directions);
    // Calculate difference between new direction and current direction
    var difference = angleDiff(this.direction,newDirection,this.directions);
    // Calculate amount that aircraft 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);
    } else {
        // Calculate distance that aircraft can move per animation cycle
        var movement = this.speed*game.speedAdjustmentFactor;
        // Calculate x and y components of the movement
        var angleRadians = -(Math.round(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);
    }
},
```

We first calculate the angle from the aircraft to its destination using the `findAngle()` method and the difference between the current and new direction using the `angleDiff()` method. The `newDirection` variable will have a value between 0 and 7 (to reflect the directions that the aircraft can take), while the difference variable will have a value between -4 and 4, with a negative sign indicating an anticlockwise turn is shorter than a clockwise turn.

We then calculate the amount the aircraft can turn based on its `turnSpeed` property and check to see whether the item needs to turn more by comparing the angle difference with the turn amount.

In case the aircraft still needs to turn, we add the `turnAmount` value to its direction while keeping the sign of the difference variable. We use the `wrapDirection()` method to ensure that the final aircraft direction is still between 0 and 7.

In case the aircraft has turned toward its destination, we calculate the movement distance based on its speed. We then calculate the x and y components of the movement and add it to the aircraft's x and y coordinates.

Of course, now that the aircraft direction can take noninteger values, we need to modify the `aircraft` object's default `animate()` method to ensure it rounds the direction before selecting the sprite (see Listing 7-7).

Listing 7-7. Modifying animate() to Handle Noninteger Direction Values (aircraft.js)

```
animate:function(){
    // Consider an item healthy if it has more than 40% life
    if (this.life>this.hitPoints*0.4){
        this.lifeCode = "healthy";
    } else if (this.life <= 0){
        this.lifeCode = "dead";
        game.remove(this);
        return;
    } else {
        this.lifeCode = "damaged";
    }
    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 + this.animationIndex;
            this.animationIndex++;
            if (this.animationIndex>=this.imageList.count){
                this.animationIndex = 0;
            }
            break;
    }
},
```

We first round the aircraft direction and then call `wrapDirection()` to ensure that the direction lies between 0 and 7. The rest of the method remains the same.

Next we will add the `findAngle()`, `angleDiff()`, and `wrapDirection()` methods to `common.js`, as shown in Listing 7-8.

Listing 7-8. Implementing findAngle(), angleDiff(), and wrapDirection() (common.js)

```
/* Common functions for turning and movement */

// Finds the angle between two objects in terms of a direction (where 0 <= angle < directions)
function findAngle(object,unit,directions){
     var dy = (object.y) - (unit.y);
     var dx = (object.x) - (unit.x);
    //Convert Arctan to value between (0 - directions)
    var angle = wrapDirection(directions/2-(Math.atan2(dx,dy)*directions/(2*Math.PI)),directions);
    return angle;
 }
```

```
// returns the smallest difference (value ranging between -directions/2 to +directions/2)
between two angles (where 0 <= angle < directions)
function angleDiff(angle1,angle2,directions){
    if (angle1>=directions/2){
        angle1 = angle1-directions;
    }
    if (angle2>=directions/2){
        angle2 = angle2-directions;
    }

    diff = angle2-angle1;

    if (diff<-directions/2){
        diff += directions;
    }
    if (diff>directions/2){
        diff -= directions;
    }

    return diff;
}

// Wrap value of direction so that it lies between 0 and directions-1
function wrapDirection(direction,directions){
    if (direction<0){
        direction += directions;
    }
    if (direction >= directions){
        direction -= directions;
    }
    return direction;
}
```

The last change we need to make is defining two movement-related properties within the `game` object inside `game.js` (see Listing 7-9).

Listing 7-9. Adding Movement-Related Properties to the game Object (game.js)

```
//Movement related properties
speedAdjustmentFactor:1/64,
turnSpeedAdjustmentFactor:1/8,
```

These two factors are used to convert an entity's `speed` and `turnSpeed` values into in-game units for movement and turning.

We are now ready to start moving our aircraft within the game, but before we do that, let's simplify our level by removing all the unnecessary items from the map. The new `maps.js` will look like Listing 7-10.

Listing 7-10. Removing Unnecessary Items from the Map (maps.js)

```
var maps = {
    "singleplayer":[
        {
            "name":"Entities",
            "briefing": "In this level you will start commanding units and moving them around the map.",

            /* Map Details */
            "mapImage":"images/maps/level-one-debug-grid.png",
            "startX":2,
            "startY":3,

            /* Entities to be loaded */
            "requirements":{
                "buildings":["base","starport","harvester","ground-turret"],
                "vehicles":["transport","harvester","scout-tank","heavy-tank"],
                "aircraft":["chopper","wraith"],
                "terrain":["oilfield","bigrocks","smallrocks"]
            },

            /* 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":"buildings","name":"harvester","x":20,"y":10, "team":"blue"},
                {"type":"buildings","name":"ground-turret","x":24,"y":7,
"team":"blue","direction":3},

                {"type":"vehicles","name":"transport","x":24,"y":10, "team":"blue","direction":2},
                {"type":"vehicles","name":"harvester","x":16,"y":12, "team":"blue","direction":3},
                {"type":"vehicles","name":"scout-tank","x":24,"y":14, "team":"blue","direction":4},
                {"type":"vehicles","name":"heavy-tank","x":24,"y":16, "team":"blue","direction":5},

                {"type":"aircraft","name":"chopper","x":7,"y":9, "team":"blue","direction":2},
                {"type":"aircraft","name":"wraith","x":11,"y":9, "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}
            ]
        }
    ]
}
```

When you run the game in the browser, you should be able to select the two aircraft and move them around on the new map shown in Figure 7-1.

Figure 7-1. *Moving aircraft around the new map*

When you select an aircraft and right-click the map somewhere, the aircraft should turn and move toward the destination. You will notice that the wraith aircraft moves faster than the chopper because we specified a higher value for `speed` in the `wraith` entity's properties.

You may also notice that right-clicking a building or a friendly unit doesn't do anything. This is because right-clicking a friendly item generates the `guard` order, which we have not yet implemented.

Implementing movement for our aircraft was fairly simple because we took the creative liberty of assuming that aircraft could avoid buildings, vehicles, and other aircraft by virtue of adjusting their height.

However, when it comes to vehicles, we can no longer do that. We need to worry about finding the shortest path between a vehicle and its destination while driving around obstacles such as buildings and terrain. This is where pathfinding comes in.

Pathfinding

Pathfinding, or pathing, is the process of finding the shortest path between two points. Typically it involves the use of various algorithms to traverse a graph of nodes starting at one vertex and exploring adjacent nodes until the destination node is reached.

Two of the most commonly used algorithms for graph-based pathfinding are Dijkstra's algorithm and its variant called the A* (pronounced "A star") algorithm.

A* uses an additional distance heuristic that helps it find paths faster than Dijkstra. Because of its performance and accuracy, it is widely used in games. You can read more about the algorithm at `http://en.wikipedia.org/wiki/A*`. We will also be using A* for the vehicle pathing in our game.

We will use an excellent MIT-licensed JavaScript implementation of A* by Andrea Giammarchi. The code has been optimized for JavaScript, and its performance even on large graphs is fairly good. You can see the latest code

as well as play with a live demo at http://devpro.it/javascript_id_137.html. We will add a reference to the A* implementation (stored in `astar.js`) to the head section of `index.html`, as shown in Listing 7-11.

Listing 7-11. Adding Reference to the A* Implementation (index.html)

```
<!-- A* Implementation by Andrea Giammarchi -->
<script src="js/astar.js" type="text/javascript" charset="utf-8"></script>
```

The implementation, while fairly complex, is relatively easy to use. The code gives us access to an `Astar()` method that accepts four parameters: the map graph for us to use, the starting coordinates, the ending coordinates, and optionally a name for the heuristic to use.

The method returns either an array with all the intermediate steps of the shortest path or an empty array in case there is no possible path.

Now that we have our A* algorithm in place, we need to provide it with a graph or grid for pathfinding.

Defining Our Pathfinding Grid

We have already broken our map into a grid of squares with the dimensions 20 pixels by 20 pixels. We will store the pathfinding grid as a two-dimensional array with values of 0 and 1 for passable and impassable squares, respectively.

Before we can create this array, we will need to modify our map to define all the impassable areas of terrain on the map. We will do this by adding a few new properties to the first level in `maps.js`, as shown in Listing 7-12.

Listing 7-12. Adding Properties for Pathfinding to the Level (maps.js)

```
/* Map coordinates that are obstructed by terrain*/
"mapGridWidth":60,
"mapGridHeight":40,
"mapObstructedTerrain":[
    [49,8], [50,8], [51,8], [51,9], [52,9], [53,9], [53,10], [53,11], [53,12], [53,13], [53,14],
[53,15], [53,16], [52,16], [52,17], [52,18], [52,19], [51,19], [50,19], [50,18], [50,17], [49,17],
[49,18], [48,18], [47,18], [47,17], [47,16], [48,16], [49,16], [49,15], [49,14], [48,14], [48,13],
[48,12], [49,12], [49,11], [50,11], [50,10], [49,10], [49,9], [44,0], [45,0], [45,1], [45,2],
[46,2], [47,2], [47,3], [48,3], [48,4], [48,5], [49,5], [49,6], [49,7], [50,7], [51,7], [51,6],
[51,5], [51,4], [52,4], [53,4], [53,3], [54,3], [55,3], [55,2], [56,2], [56,1], [56,0], [55,0],
[43,19], [44,19], [45,19], [46,19], [47,19], [48,19], [48,20], [48,21], [47,21], [46,21], [45,21],
[44,21], [43,21], [43,20], [41,22], [42,22], [43,22], [44,22], [45,22], [46,22], [47,22], [48,22],
[49,22], [50,22], [50,23], [50,24], [49,24], [48,24], [47,24], [47,25], [47,26], [47,27], [47,28],
[47,29], [47,30], [46,30], [45,30], [44,30], [43,30], [43,29], [43,28], [43,27], [43,26], [43,25],
[43,24], [42,24], [41,24], [41,23], [48,39], [49,39], [50,39], [51,39], [52,39], [53,39], [54,39],
[55,39], [56,39], [57,39], [58,39], [59,39], [59,38], [59,37], [59,36], [59,35], [59,34], [59,33],
[59,32], [59,31], [59,30], [59,29], [0,0], [1,0], [2,0], [1,1], [2,1], [10,3], [11,3], [12,3],
[12,2], [13,2], [14,2], [14,3], [14,4], [15,4], [15,5], [15,6], [14,6], [13,6], [13,5], [12,5],
[11,5], [10,5], [10,4], [3,9], [4,9], [5,9], [5,10], [6,10], [7,10], [8,10], [9,10], [9,11],
[10,11], [11,11], [11,10], [12,10], [13,10], [13,11], [13,12], [12,12], [11,12], [10,12], [9,12],
[8,12], [7,12], [7,13], [7,14], [6,14], [5,14], [5,13], [5,12], [5,11], [4,11], [3,11], [3,10],
[33,33], [34,33], [35,33], [35,34], [35,35], [34,35], [33,35], [33,34], [27,39], [27,38], [27,37],
[28,37], [28,36], [28,35], [28,34], [28,33], [28,32], [28,31], [28,30], [28,29], [29,29], [29,28],
[29,27], [29,26], [29,25], [29,24], [29,23], [30,23], [31,23], [32,23], [32,22], [32,21], [31,21],
[30,21], [30,22], [29,22], [28,22], [27,22], [26,22], [26,21], [25,21], [24,21], [24,22], [24,23],
[25,23], [26,23], [26,24], [25,24], [25,25], [24,25], [24,26], [24,27], [25,27], [25,28], [25,29],
[24,29], [23,29], [23,30], [23,31], [24,31], [25,31], [25,32], [25,33], [24,33], [23,33], [23,34],
```

```
[23,35], [24,35], [24,36], [24,37], [23,37], [22,37], [22,38], [22,39], [23,39], [24,39], [25,39],
[26,0], [26,1], [25,1], [25,2], [25,3], [26,3], [27,3], [27,2], [28,2], [29,2], [29,3], [30,3],
[31,3], [31,2], [31,1], [32,1], [32,0], [33,0], [32,8], [33,8], [34,8], [34,9], [34,10], [33,10],
[32,10], [32,9], [8,29], [9,29], [9,30], [17,32], [18,32], [19,32], [19,33], [18,33], [17,33],
[18,34], [19,34], [3,27], [4,27], [4,26], [3,26], [2,26], [3,25], [4,25], [9,20], [10,20], [11,20],
[11,21], [10,21], [10,19], [19,7], [15,7], [29,12], [30,13], [20,14], [21,14], [34,13], [35,13],
[36,13], [36,14], [35,14], [34,14], [35,15], [36,15], [16,18], [17,18], [18,18], [16,19], [17,19],
[18,19], [17,20], [18,20], [11,19], [58,0], [59,0], [58,1], [59,1], [59,2], [58,3], [59,3], [58,4],
[59,4], [59,5], [58,6], [59,6], [58,7], [59,7], [59,8], [58,9], [59,9], [58,10], [59,10], [59,11],
[52,6], [53,6], [54,6], [52,7], [53,7], [54,7], [53,8], [54,8], [44,17], [46,32], [55,32], [54,28],
[26,34], [34,34], [4,10], [6,11], [6,12], [6,13], [7,11], [8,11], [12,11], [27,0], [27,1], [26,2],
[28,1], [28,0], [29,0], [29,1], [30,2], [30,1], [30,0], [31,0], [33,9], [46,0], [47,0], [48,0],
[49,0], [50,0], [51,0], [52,0], [53,0], [54,0], [55,1], [54,1], [53,1], [52,1], [51,1], [50,1],
[49,1], [48,1], [47,1], [46,1], [48,2], [49,2], [50,2], [51,2], [52,2], [53,2], [54,2], [52,3],
[51,3], [50,3], [49,3], [49,4], [50,4], [50,5], [50,6], [50,9], [51,10], [52,10], [51,11], [52,11],
[50,12], [51,12], [52,12], [49,13], [50,13], [51,13], [52,13], [50,14], [51,14], [52,14], [50,15],
[51,15], [52,15], [50,16], [51,16], [51,17], [48,17], [51,18], [44,20], [45,20], [46,20], [47,20],
[42,23], [43,23], [44,23], [45,23], [46,23], [47,23], [48,23], [49,23], [44,24], [45,24], [46,24],
[44,25], [45,25], [46,25], [44,26], [45,26], [46,26], [44,27], [45,27], [46,27], [44,28], [45,28],
[46,28], [44,29], [45,29], [46,29], [11,4], [12,4], [13,4], [13,3], [14,5], [25,22], [31,22],
[27,23], [28,23], [27,24], [28,24], [26,25], [27,25], [28,25], [25,26], [26,26], [27,26], [28,26],
[26,27], [27,27], [28,27], [26,28], [27,28], [28,28], [26,29], [27,29], [24,30], [25,30], [26,30],
[27,30], [26,31], [27,31], [26,32], [27,32], [26,33], [27,33], [24,34], [25,34], [27,34], [25,35],
[26,35], [27,35], [25,36], [26,36], [27,36], [25,37], [26,37], [23,38], [24,38], [25,38], [26,38],
[26,39], [2,25], [9,19], [36,31]
],
```

We first define two properties called `mapGridWidth` and `mapGridHeight` and finally an extremely large and scary-looking array called `mapObstructedTerrain`. This array merely contains the x and y coordinates for every grid inside our map that is impassable. This includes areas with trees, mountains, water, craters, and lava.

■ **Note** If you plan to add a lot of levels to your game, you should take the time to design a level editor that generates this array for you automatically instead of trying to create it by hand.

Now that we have these properties in place, we need to generate a terrain grid from this data when we load the level. We will do this within the `singleplayer` object's `startCurrentLevel()` method inside `singleplayer.js` (see Listing 7-13).

Listing 7-13. Creating the Terrain Grid When Starting Level (singleplayer.js)

```
startCurrentLevel:function(){
    // Load all the items for the level
    var level = maps.singleplayer[singleplayer.currentLevel];

    // Don't allow player to enter mission until all assets for the level are loaded
    $("#entermission").attr("disabled", true);

    // Load all the assets for the level
    game.currentMapImage = loader.loadImage(level.mapImage);
    game.currentLevel = level;
```

```
    game.offsetX = level.startX * game.gridSize;
    game.offsetY = level.startY * game.gridSize;

    // Load level Requirements
    game.resetArrays();
    for (var type in level.requirements){
        var requirementArray = level.requirements[type];
        for (var i=0; i < requirementArray.length; i++) {
            var name = requirementArray[i];
            if (window[type]){
                window[type].load(name);
            } else {
                console.log('Could not load type :',type);
            }
        };
    }

    for (var i = level.items.length - 1; i >= 0; i--){
        var itemDetails = level.items[i];
        game.add(itemDetails);
    };

    // Create a grid that stores all obstructed tiles as 1 and unobstructed as 0
    game.currentMapTerrainGrid = [];
    for (var y=0; y < level.mapGridHeight; y++) {
        game.currentMapTerrainGrid[y] = [];
        for (var x=0; x< level.mapGridWidth; x++) {
           game.currentMapTerrainGrid[y][x] = 0;
        }
    };
    for (var i = level.mapObstructedTerrain.length - 1; i >= 0; i--){
        var obstruction = level.mapObstructedTerrain[i];
        game.currentMapTerrainGrid[obstruction[1]][obstruction[0]] = 1;
    };
    game.currentMapPassableGrid = undefined;

    // Enable the enter mission button once all assets are loaded
    if (loader.loaded){
        $("#entermission").removeAttr("disabled");
    } else {
        loader.onload = function(){
            $("#entermission").removeAttr("disabled");
        }
    }

    // Load the mission screen with the current briefing
    $('#missonbriefing').html(level.briefing.replace(/\n/g,'<br><br>'));
    $("#missionscreen").show();
},
```

We initialize an array called `currentMapTerrainGrid` inside the `game` object and set it to the dimensions of our map using `mapGridWidth` and `mapGridHeight`. We then set all the obstructed squares to 1 and leave the unobstructed tiles as 0.

If we were to highlight the obstructed squares in `currentMapTerrainGrid` on our map, it would look like Figure 7-2.

Figure 7-2. *Obstructed grid squares defined in currentMapTerrainGrid*

While `currentMapTerrainGrid` marks out all the obstacles in the map terrain, it still does not include the buildings and terrain entities on the map.

We will keep another array inside the `game` object called `currentMapPassableGrid` that will combine the building and terrain entities and the `currentMapTerrainGrid` array we defined earlier. This array will need to be re-created every time buildings or terrain get added to or removed from the game. We will do this in a `rebuildPassableGrid()` method within the `game` object (see Listing 7-14).

Listing 7-14. rebuildPassableGrid() Method in game Object (game.js)

```
rebuildPassableGrid:function(){
    game.currentMapPassableGrid = $.extend(true,[],game.currentMapTerrainGrid);
    for (var i = game.items.length - 1; i >= 0; i--){
        var item = game.items[i];
        if(item.type == "buildings" || item.type == "terrain"){
            for (var y = item.passableGrid.length - 1; y >= 0; y--){
                for (var x = item.passableGrid[y].length - 1; x >= 0; x--){
```

```
                    if(item.passableGrid[y][x]){
                        game.currentMapPassableGrid[item.y+y][item.x+x] = 1;
                    }
                };
            };
        }
    };
},
```

We first copy the `currentMapTerrainGrid` array into `currentMapPassableGrid`. We then iterate through all the game items and use the `passableGrid` property that we defined for all buildings and terrain to mark out grid squares that are not passable. If we were to highlight the obstructed squares in our map based on the `currentMapPassableGrid`, it would look like Figure 7-3.

Figure 7-3. *Obstructed grid squares defined in currentMapPassableGrid*

Because of the way we define `passableGrid` for each building, it is possible to allow portions of buildings to be passable (for example, the lower portion of the starport).

We will need to ensure that `game.currentMapPassableGrid` is reset anytime a building is added or removed from the game. We do this by adding an extra condition inside the `add()` and `remove()` methods of the `game` object, as shown in Listing 7-15.

Listing 7-15. Clearing currentMapPassableGrid Inside add() and remove() (game.js)

```
add:function(itemDetails) {
    // Set a unique id for the item
    if (!itemDetails.uid){
        itemDetails.uid = game.counter++;
    }

    var item = window[itemDetails.type].add(itemDetails);

    // Add the item to the items array
    game.items.push(item);
    // Add the item to the type specific array
    game[item.type].push(item);

    if(item.type == "buildings" || item.type == "terrain"){
        game.currentMapPassableGrid = undefined;
    }
    return item;
},
remove:function(item){
    // Unselect item if it is selected
    item.selected = false;
    for (var i = game.selectedItems.length - 1; i >= 0; i--){
           if(game.selectedItems[i].uid == item.uid){
               game.selectedItems.splice(i,1);
               break;
           }
       };

    // Remove item from the items array
    for (var i = game.items.length - 1; i >= 0; i--){
        if(game.items[i].uid == item.uid){
            game.items.splice(i,1);
            break;
        }
    };

    // Remove items from the type specific array
    for (var i = game[item.type].length - 1; i >= 0; i--){
        if(game[item.type][i].uid == item.uid){
           game[item.type].splice(i,1);
           break;
        }
    };

    if(item.type == "buildings" || item.type == "terrain"){
        game.currentMapPassableGrid = undefined;
    }
},
```

Within both methods, we check whether the item being added or removed is a building or terrain type and, if so, reset the `currentMapPassableGrid` variable.

Now that we have defined the movement grid for our A* algorithm, we are ready to implement vehicle movement.

Implementing Vehicle Movement

We will start by adding a default `processOrders()` method for the `vehicles` object inside `vehicles.js`, as shown in Listing 7-16.

Listing 7-16. The Default processOrders() Method for Vehicles (vehicles.js)

```
processOrders:function(){
    this.lastMovementX = 0;
    this.lastMovementY = 0;
    switch (this.orders.type){
        case "move":
            // Move towards 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)) {
                this.orders = {type:"stand"};
                return;
            } else {
                // Try to move to the destination
                var moving = this.moveTo(this.orders.to);
                if(!moving){
                    // Pathfinding couldn't find a path so stop
                    this.orders = {type:"stand"};
                    return;
                }
            }
            break;
    }
},
```

The method is fairly similar to the `processOrders()` method that we defined for aircraft. The one subtle difference is that we check whether the `moveTo()` method returns a value of true indicating it is able to move toward the destination and reset the order to `stand` in case it does not. We do this because it is possible for the pathfinding algorithm to not find a valid path, and `moveTo()` will return a value indicating this.

Next, we will implement the default `moveTo()` method for vehicles, as shown in Listing 7-17.

Listing 7-17. The Default moveTo() Method for Vehicles (vehicles.js)

```
moveTo:function(destination){
    if(!game.currentMapPassableGrid){
        game.rebuildPassableGrid();
    }
```

```
    // First find path to destination
    var start = [Math.floor(this.x),Math.floor(this.y)];
    var end = [Math.floor(destination.x),Math.floor(destination.y)];

    var grid = $.extend(true,[],game.currentMapPassableGrid);
    // Allow destination to be "movable" so that algorithm can find a path
    if(destination.type == "buildings"||destination.type == "terrain"){
        grid[Math.floor(destination.y)][Math.floor(destination.x)] = 0;
    }

    var newDirection;
    // if vehicle is outside map bounds, just go straight towards goal
    if (start[1]<0 || start[1]>=game.currentLevel.mapGridHeight || start[0]<0 ||
start[0]>= game.currentLevel.mapGridWidth){
        this.orders.path = [this,destination];
        newDirection = findAngle(destination,this,this.directions);
    } else {
        //Use A* algorithm to try and find a path to the destination
        this.orders.path = AStar(grid,start,end,'Euclidean');
        if (this.orders.path.length>1){
            var nextStep = {x:this.orders.path[1].x+0.5,y:this.orders.path[1].y+0.5};
            newDirection = findAngle(nextStep,this,this.directions);
        } else if(start[0]==end[0] && start[1] == end[1]){
            // Reached destination grid;
            this.orders.path = [this,destination];
            newDirection = findAngle(destination,this,this.directions);
        } else {
            // There is no path
            return false;
        }
    }

    // Calculate turn amount for new direction
    var difference = angleDiff(this.direction,newDirection,this.directions);
    var turnAmount = this.turnSpeed*game.turnSpeedAdjustmentFactor;

    // Move forward, but keep turning as needed
    var movement = this.speed*game.speedAdjustmentFactor;
    var angleRadians = -(Math.round(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);

    if (Math.abs(difference)>turnAmount){
        this.direction = wrapDirection(this.direction + turnAmount*Math.abs(difference)/difference,
this.directions);
    }

    return true;
},
```

We start by checking whether `game.currentMapPassableGrid` is defined and calling `game.rebuildPassableGrid()` if it is not defined. We then define the start and end values for our path by truncating the vehicle and destination locations.

Next, we copy the `game.currentMapPassableGrid` into a grid variable and define the destination grid square as passable in case the destination is a building or terrain. This hack lets the A* algorithm find a path to a building even though the destination is impassable.

The next step is calculating the path and new direction. We first check whether the vehicle is outside the map bounds and, if so, head straight toward the destination by defining the start and end locations of our path using the vehicle and destination and calculating `newDirection` using the `findAngle()` method. We do this because the `AStar()` method would fail if we passed it starting coordinates that were outside the grid.

If the vehicle is within the map bounds, we call the `AStar()` method while passing it values of `start`, `end`, and the grid. We specify a heuristic method of `Euclidean`, which allows diagonal movement and seems to work well for our game.

If the `AStar()` method returns a path with at least two elements, we calculate `newDirection` by finding the angle from the vehicle to the middle of the next grid.

If the path does not contain at least two elements, we check whether this is because we have reached the destination grid square and, if so, aim toward the final destination. If not, we assume this is because `AStar()` could not find a path and return false.

Finally, we use the `newDirection` and `turnSpeed` and `speed` values to both move the vehicle forward and turn it toward `newDirection`. Unlike our aircraft, vehicles should not be able to turn in place, and we implement this by making both movement and turning happen simultaneously.

With the heart of our pathfinding method implemented, we will need to make one last change to the `vehicles` object. We will modify the default `animate()` method to account for noninteger values of direction, as shown in Listing 7-18.

Listing 7-18. Modifying animate() to Handle Noninteger Direction Values (vehicles.js)

```
animate:function(){
    // Consider an item healthy if it has more than 40% life
    if (this.life>this.hitPoints*0.4){
        this.lifeCode = "healthy";
    } else if (this.life <= 0){
        this.lifeCode = "dead";
        game.remove(this);
        return;
    } else {
        this.lifeCode = "damaged";
    }

    switch (this.action){
        case "stand":
            var direction = wrapDirection(Math.round(this.direction),this.directions);
            this.imageList = this.spriteArray["stand-"+direction];
            this.imageOffset = this.imageList.offset + this.animationIndex;
            this.animationIndex++;

            if (this.animationIndex>=this.imageList.count){
                this.animationIndex = 0;
            }
            break;
    }
},
```

If you run the game now, you should be able to select vehicles and move them around the map by right-clicking a spot on the map. The vehicles will move along a path that avoids all the terrain and building obstacles. Figure 7-4 illustrates a typical path returned by the pathfinding algorithm.

Figure 7-4. *Typical movement path using pathfinding algorithm*

One thing that you will notice is that while the vehicles avoid unpassable terrain, they still drive over other vehicles.

One simple way to fix this is to just mark all squares occupied by any vehicle as unpassable. However, this simplistic approach might end up blocking very large portions of the map since vehicles often move across multiple grid squares. The other disadvantage of this method is that if we try to move a bunch of vehicles through a narrow passage, the first vehicle will block the passage, causing the pathfinding for the vehicles behind to try to find a longer alternate route or, worse, assume there is no possible path and give up.

A better alternative is to implement a steering step that checks for collisions with other objects and modifies the vehicle's direction while still trying to maintain the original path as far as possible.

Collision Detection and Steering

Steering, like pathfinding, is a fairly vast AI subject. The idea of applying steering behavior in games has been around for a long time, but it was popularized by the work of Craig Reynolds in the mid to late 1980s. His paper "Steering Behaviors for Autonomous Characters" and his Java demos are still considered the basic starting point for developing steering mechanisms in games. You can read more about his research and look at demos of various steering mechanisms at `http://www.red3d.com/cwr/steer/`.

We will use a fairly simple implementation for our game. We will first check to see whether moving a vehicle along its present direction will result in collisions with any object. If there are colliding objects, we will create

repulsive forces from any colliding objects to our vehicle and a mild attractive force toward the next grid square in the pathfinding path.

We will then combine all of these forces as vectors to see which direction the vehicle will need to move to get away from the collisions. We will steer the vehicle toward this direction until the vehicle is no longer colliding with any object, at which point the vehicle will return to the basic pathfinding mode.

We will distinguish between hard and soft collisions based on the distance from colliding objects. A vehicle that is about to have a soft collision may still move while turning; however, a vehicle about to have hard collision will not move forward at all and will only turn away.

We will start by implementing a default `checkCollisionsObject()` method for the `vehicle` object inside `vehicles.js`, as shown in Listing 7-19.

Listing 7-19. The Default checkCollisionObjects() Method (vehicles.js)

```
// Make a list of collisions that the vehicle will have if it goes along present path
checkCollisionObjects:function(grid){
    // Calculate new position on present path
    var movement = this.speed*game.speedAdjustmentFactor;
    var angleRadians = -(Math.round(this.direction)/this.directions)*2*Math.PI;
    var newX = this.x - (movement*Math.sin(angleRadians));
    var newY = this.y - (movement*Math.cos(angleRadians));

    // List of objects that will collide after next movement step
    var collisionObjects = [];

    var x1 = Math.max(0,Math.floor(newX)-3);
    var x2 = Math.min(game.currentLevel.mapGridWidth-1,Math.floor(newX)+3);
    var y1 = Math.max(0,Math.floor(newY)-3);
    var y2 = Math.min(game.currentLevel.mapGridHeight-1,Math.floor(newY)+3);
    // Test grid upto 3 squares away
    for (var j=x1; j <= x2;j++){
        for(var i=y1; i<= y2 ;i++){
            if(grid[i][j]==1){ // grid square is obsutructed
                if (Math.pow(j+0.5-newX,2)+Math.pow(i+0.5-newY,2) <
Math.pow(this.radius/game.gridSize+0.1,2)){
                    // Distance of obstructed grid from vehicle is less than hard collision threshold
                    collisionObjects.push({collisionType:"hard", with:{type:"wall",x:j+0.5,y:i+0.5}});
                } else if (Math.pow(j+0.5-newX,2)+Math.pow(i+0.5-newY,2) <
Math.pow(this.radius/game.gridSize+0.7,2)){
                    // Distance of obstructed grid from vehicle is less than soft collision
threshold
                     collisionObjects.push({collisionType:"soft", with:{type:"wall",x:j+0.5,y:i+0.5}});
                }
            }
        };
    };

    for (var i = game.vehicles.length - 1; i >= 0; i--){
        var vehicle = game.vehicles[i];
        // Test vehicles that are less than 3 squares away for collisions
        if (vehicle != this && Math.abs(vehicle.x-this.x)<3 && Math.abs(vehicle.y-this.y)<3){
            if (Math.pow(vehicle.x-newX,2) + Math.pow(vehicle.y-newY,2) <
Math.pow((this.radius+vehicle.radius)/game.gridSize,2)){
```

```
                // Distance between vehicles is less than hard collision threshold (sum of vehicle
radii)
                    collisionObjects.push({collisionType:"hard",with:vehicle});
            } else if (Math.pow(vehicle.x-newX,2) + Math.pow(vehicle.y-newY,2) <
Math.pow((this.radius*1.5+vehicle.radius)/game.gridSize,2)){
                // Distance between vehicles is less than soft collision threshold (1.5 times
vehicle radius + colliding vehicle radius)
                collisionObjects.push({collisionType:"soft",with:vehicle});
            }
        }
    };

    return collisionObjects;
},
```

We first calculate the new position of the vehicle if it moves along its current direction. We then check to see whether there are any impassable grid squares nearby that might collide with the vehicle in its new position by comparing the distances between their centers with certain threshold values based on the vehicle radius. We mark collisions as "hard" if they are colliding or "soft" if they are almost ready to collide. All collisions are then added to the `collisionObjects` array.

We then repeat this process with the `vehicles` array by testing all vehicles that are close by for possible collisions using the sum of their radii as a threshold distance.

Now that we have a list of colliding objects, we will modify the default `moveTo()` method we defined earlier to handle collisions (see Listing 7-20).

Listing 7-20. Handling Collisions Inside the default moveTo() Method (vehicles.js)

```
moveTo:function(destination){
    if(!game.currentMapPassableGrid){
        game.rebuildPassableGrid();
    }

    // First find path to destination
    var start = [Math.floor(this.x),Math.floor(this.y)];
    var end = [Math.floor(destination.x),Math.floor(destination.y)];

    var grid = $.extend(true,[],game.currentMapPassableGrid);
    // Allow destination to be "movable" so that algorithm can find a path
    if(destination.type == "buildings"||destination.type == "terrain"){
        grid[Math.floor(destination.y)][Math.floor(destination.x)] = 0;
    }

    var newDirection;
    // if vehicle is outside map bounds, just go straight towards goal
    if (start[1]<0 || start[1]>=game.currentLevel.mapGridHeight || start[0]<0 ||
start[0]>= game.currentLevel.mapGridWidth){
        this.orders.path = [this,destination];
        newDirection = findAngle(destination,this,this.directions);
    } else {
        //Use A* algorithm to try and find a path to the destination
        this.orders.path = AStar(grid,start,end,'Euclidean');
```

```
        if (this.orders.path.length>1){
            var nextStep = {x:this.orders.path[1].x+0.5,y:this.orders.path[1].y+0.5};
            newDirection = findAngle(nextStep,this,this.directions);
        } else if(start[0]==end[0] && start[1] == end[1]){
            // Reached destination grid square
            this.orders.path = [this,destination];
            newDirection = findAngle(destination,this,this.directions);
        } else {
            // There is no path
            return false;
        }
    }

    // check if moving along current direction might cause collision..
    // If so, change newDirection
    var collisionObjects = this.checkCollisionObjects(grid);
    this.hardCollision = false;
    if (collisionObjects.length>0){
        this.colliding = true;

        // Create a force vector object that adds up repulsion from all colliding objects
        var forceVector = {x:0,y:0}
        // By default, the next step has a mild attraction force
        collisionObjects.push({collisionType:"attraction", with:{x:this.orders.path[1].x+0.5,
y:this.orders.path[1].y+0.5}});
        for (var i = collisionObjects.length - 1; i >= 0; i--){
            var collObject = collisionObjects[i];
            var objectAngle = findAngle(collObject.with,this,this.directions);
            var objectAngleRadians = -(objectAngle/this.directions)* 2*Math.PI;
            var forceMagnitude;
            switch(collObject.collisionType){
                case "hard":
                    forceMagnitude = 2;
                    this.hardCollision = true;
                    break;
                case "soft":
                    forceMagnitude = 1;
                    break;
                case "attraction":
                    forceMagnitude = -0.25;
                    break;
            }

            forceVector.x += (forceMagnitude*Math.sin(objectAngleRadians));
            forceVector.y += (forceMagnitude*Math.cos(objectAngleRadians));
        };
        // Find a new direction based on the force vector
        newDirection = findAngle(forceVector,{x:0,y:0},this.directions);
    } else {
        this.colliding = false;
    }
```

```
    // Calculate turn amount for new direction
    var difference = angleDiff(this.direction,newDirection,this.directions);
    var turnAmount = this.turnSpeed*game.turnSpeedAdjustmentFactor;

    // Either turn or move forward based on collision type
    if (this.hardCollision){
        // In case of hard collision, do not move forward, just turn towards new direction
        if (Math.abs(difference)>turnAmount){
            this.direction = wrapDirection(this.direction+
turnAmount*Math.abs(difference)/difference, this.directions);
        }
    } else {
        // Otherwise, move forward, but keep turning as needed
        var movement = this.speed*game.speedAdjustmentFactor;
        var angleRadians = -(Math.round(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);
        if (Math.abs(difference)>turnAmount){
            this.direction = wrapDirection(this.direction+
turnAmount*Math.abs(difference)/difference, this.directions);
        }
    }
    return true;
},
```

After the initial pathfinding step, we call the `checkCollisionObjects()` method and get a list of objects that the vehicle will collide with.

We then iterate through this list of objects and define a repulsive force for each with a magnitude of either 1 or 2 based on whether the collision is "soft" or "hard." We also define an attractive force toward the next pathfinding grid square. Finally, we add up all these forces into a `forceVector` object and use it to calculate the direction that would take the vehicle the farthest away from all the forces and assign it to the `newDirection` variable.

What this means is as long as there are no colliding objects, the vehicle will head toward the next grid square defined in its path. The moment the vehicle senses a collision, its primary motivation will be to avoid the collision by taking evasive action. Once the collision threat has been averted, the vehicle will return to its original path-following behavior.

We add an extra check to prevent the vehicle from moving forward if movement will result in a hard collision. As a result, the vehicle will stop completely rather than actually collide with another object.

If you run the game now and try to move a vehicle, you will find that it steers around other vehicles to avoid colliding with them.

One problem that you might notice is that if you try to move multiple vehicles to the same spot, the first one stops at the correct location, while the others keep circling around trying in vain to reach the occupied stop. We will need to fix this by adding some intelligence to the way a vehicle handles trying to move to a blocked spot.

The ideal behavior would be to stop at a little distance from the destination if the destination is blocked and to stop even farther away if the vehicle has been colliding for a long time without reaching its destination.

We will implement this by modifying the default `processOrders()` method, as shown in Listing 7-21.

Listing 7-21. Modifying processOrders() to Handle Stopping (vehicles.js)

```
processOrders:function(){
    this.lastMovementX = 0;
    this.lastMovementY = 0;
    switch (this.orders.type){
        case "move":
            // Move towards 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;
    }
},
```

We first try to stop at the destination if the vehicle is within 1 radius of the destination. We also stop if the vehicle is colliding and within a 3-radius distance of the destination. Finally, we stop if the vehicle has been colliding more than 30 times while being within a 5-radius distance of the destination. This last condition handles situations where the vehicle has been bumping around a crowded area for a while without finding a way to reach its destination.

If you run the game now and try to move multiple vehicles together, you will see that they intelligently stop near their destination even in crowded areas.

At this point, we have a reasonably good pathfinding and steering solution for intelligent unit movement. This system can be developed further to improve performance and add other intelligent behavior such as queuing, flocking, and leader following, depending on your game requirements. You should definitely research this topic further as you implement unit movement within your own games, starting with the work by Craig Reynolds (www.red3d.com/cwr/steer/).

Now that we have vehicle movement working, let's take the time to implement one more movement-related order: deploying the harvester.

Deploying the Harvester

We designed the harvester as a deployable vehicle that opened up into a harvester building when deployed on an oil field. We already set up the code to pass the `deploy` order to a harvester when the player right-clicks an oil field. Now we will implement the `deploy` case within the `vehicle` object's `processOrders()` method inside `vehicles.js`, as shown in Listing 7-22.

Listing 7-22. Implementation of the deploy Case Inside processOrders() (vehicles.js)

```
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 towards 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;
```

We start by using the `moveTo()` method to move the harvester to the middle of the oil field. Once the harvester reaches the oil field, we use the `angleDiff()` method and turn the harvester toward the left (direction 6). Finally, we remove the harvester vehicle and oil field items from the game and add a harvester building at the oil field's location with the action set to `deploy`.

If we run our game, select the harvester vehicle, and then right-click an oil field, we should see the harvester move to the oil field and deploy into a building, as shown in Figure 7-5.

Figure 7-5. *Harvester vehicle deploying into a harvester building*

The harvester moves to the oil field, turns into position, and then seems to expand into a harvester building. As you can see, with the movement framework in place, handling different orders is very easy.

Before we wrap up unit movement, we will address one last thing. You may have noticed that the unit movement especially for fast units such as the wraith seems a little choppy. We will try to smoothen this unit movement.

Smoother Unit Movement

Our game animation loop currently runs at a steady 10 frames per second. Even though our drawing loop runs faster (typically 30 to 60 frames per second), it has no new information to draw during these extra loops, so effectively it too draws at 10 frames per second. This results in the choppy-looking movement that we see.

One simple way to make the animation look much smoother is interpolating the vehicle movement between animation frames. We can calculate the time since the last animation loop and use it to create an interpolation factor that is used to position the units during intermediate drawing loops. This little adjustment will make the units seem to move at a much higher frame rate, even though they are actually being moved only at 10 frames per second.

We will start by modifying the game object's `animationLoop()` method to save the last animation time and the `drawingLoop()` method to calculate an interpolation factor based on the current drawing time and the last animation time. The final version of `animationLoop()` and `drawingLoop()` will look like Listing 7-23.

Listing 7-23. Calculating a Movement Interpolation Factor (game.js)

```
animationLoop:function(){
    // 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);
    });

    //Save the time that the last animation loop completed
    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--){
        game.sortedItems[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 save the current time into `game.lastAnimationTime` at the end of the `animationLoop()` method. We then use this variable and the current time to calculate the `game.drawingInterpolationFactor` variable that is a number between -1 and 0. A value of -1 indicates that we draw the unit at the previous location, while a value of 0 means that we draw the unit at its present location. Any value between -1 and 0 means that we draw the unit at an intermediate location between the two points. We cap the value at 0 to prevent any extrapolation from happening (i.e., drawing the unit beyond the point that it has been animated).

■ **Note** Using techniques such as extrapolation and client-side prediction to position units is much more common in multiplayer first-person shooter games to compensate for lag because of high latency.

Now that we have calculated the interpolation factor, we will use it along with the unit `lastMovementX` and `lastMovementY` values to position the element while drawing. First, we will modify the default `draw()` method for the `aircrafts` object inside `aircraft.js`, as shown in Listing 7-24.

Listing 7-24. Interpolating Movement While Drawing Aircraft (aircraft.js)

```
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.pixelShadowHeight +
this.lastMovementY*game.drawingInterpolationFactor*game.gridSize;
    this.drawingX = x;
    this.drawingY = y;
    if (this.selected){
```

```
        this.drawSelection();
        this.drawLifeBar();
    }
    var colorIndex = (this.team == "blue")?0:1;
    var colorOffset = colorIndex*this.pixelHeight;
    var shadowOffset = this.pixelHeight*2; // The aircraft shadow is on the second row of the sprite sheet

    game.foregroundContext.drawImage(this.spriteSheet, this.imageOffset*this.
pixelWidth,colorOffset,this.pixelWidth,this.pixelHeight,x,y,this.pixelWidth,this.pixelHeight);
    game.foregroundContext.drawImage(this.spriteSheet, this.imageOffset*this.
pixelWidth,shadowOffset,this.pixelWidth,this.pixelHeight,x,y+this.pixelShadowHeight,this.
pixelWidth,this.pixelHeight);
}
```

The only change that we made is adding the extrapolation-related term to the x and y coordinate calculations. Next we will make the same change to the default `draw()` method for vehicles inside `vehicles.js` (see Listing 7-25).

Listing 7-25. Interpolating Movement While Drawing Vehicles (vehicles.js)

```
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;
    this.drawingX = x;
    this.drawingY = y;

    if (this.selected){
        this.drawSelection();
        this.drawLifeBar();
    }

    var colorIndex = (this.team == "blue")?0:1;
    var colorOffset = colorIndex*this.pixelHeight;

    game.foregroundContext.drawImage(this.spriteSheet, this.imageOffset*this.pixelWidth,colorOffset,
           this.pixelWidth,this.pixelHeight,x,y,this.pixelWidth,this.pixelHeight);
}
```

If we run the game and move the units around, the movement should now be visibly smoother than before. With this last change, we can now consider unit movement wrapped up.

Summary

In this chapter, we implemented intelligent unit movement for our game.

We started by developing a framework to give selected units commands and for the entities to then follow orders.

We implemented the move order for aircraft by moving them straight toward their destination and for vehicles by using a combination of A* for pathfinding and repulsive forces for steering. We then implemented the deploy order for harvesters using the movement code we developed.

Finally, we smoothened the unit movement by integrating an interpolation step within our drawing code.

In the next chapter, we will implement more of our game rules: creating and placing buildings, teleporting vehicles and aircraft from the starport, and harvesting for money. So, let's keep going.