Chapter 6

Adding Entities to Our World

In the previous chapter, we put together the basic framework for our RTS game. We loaded a level and panned around using the mouse.

In this chapter, we will build upon that by adding entities to our game world. We will build a general framework that will allow us to easily add entities such as buildings and units to a level. Finally, we will add the ability for the player to select these entities using the mouse.

Let's get started. We will use the code from Chapter 5 as our starting point.

Defining Entities

These are the game entities we will be adding to our game:

- *Buildings*: Our game will have four types of buildings.
  - *Base*: Primary structure used to construct other buildings
  - *Starport*: Used to teleport in both ground vehicles and aircraft
  - *Harvester*: Used to extract resources from oil fields
  - *Ground turret*: Defensive structure used to guard against ground vehicles
- *Vehicles*: Our game will have four types of vehicles.
  - *Transport*: An unarmed vehicle used to transport supplies and people
  - *Harvester*: A mobile unit that deploys into the harvester building at an oil field
  - *Scout tank*: A light, fast-moving tank used for scouting
  - *Heavy tank*: A slower tank with heavier armor and weaponry
- *Aircraft*: Our game will have two types of aircraft.
  - *Chopper*: A slow-moving craft that can attack both land and air
  - *Wraith*: A fast-moving jet aircraft that can attack only in the air
- *Terrain*: Apart from the terrain already integrated in our map, we will define two additional types of terrain.
  - *Oil field*: Source of mineral resources that can be extracted for cash by deploying a harvester
  - *Rocks*: Interesting rock formations

We will store our entity types in separate JavaScript files to make the code easier to maintain. The first thing we will do is add references to the new JavaScript files inside the head section of our HTML file. The modified head section will now look like Listing 6-1.

Listing 6-1. Adding References to Entities (index.html)

```
<head>
    <meta http-equiv="Content-type" content="text/html; charset=utf-8">
    <title>Last Colony</title>
    <script src="js/common.js" type="text/javascript" charset="utf-8"></script>
    <script src="js/jquery.min.js" type="text/javascript" charset="utf-8"></script>
    <script src="js/game.js" type="text/javascript" charset="utf-8"></script>
    <script src="js/mouse.js" type="text/javascript" charset="utf-8"></script>
    <script src="js/singleplayer.js" type="text/javascript" charset="utf-8"></script>
    <script src="js/maps.js" type="text/javascript" charset="utf-8"></script>

    <!-- Definitions for game entities -->
    <script src="js/buildings.js" type="text/javascript" charset="utf-8"></script>
    <script src="js/vehicles.js" type="text/javascript" charset="utf-8"></script>
    <script src="js/aircraft.js" type="text/javascript" charset="utf-8"></script>
    <script src="js/terrain.js" type="text/javascript" charset="utf-8"></script>

    <link rel="stylesheet" href="styles.css" type="text/css" media="screen" charset="utf-8">
</head>
```

With this code in place, we are now ready to start defining our first set of entities, the buildings, to the game.

Defining Our First Entity: The Main Base

The first building we will define is the main base. Unlike other buildings in the game that can be constructed by the player, the main base will always be preconstructed before the level starts. The base allows the player to teleport in other buildings as long as the player has sufficient resources.

The base will consist of a single sprite sheet image that contains different animation states for the base (see Figure 6-1).

Figure 6-1. *Sprite sheet for the base*

As you can see, the sheet consists of two different rows of frames for the blue and green teams. The sprites in this case consist of a default animation (four frames), a damaged base (one frame), and finally an animation for when the base is constructing a building (three frames). We will be using similar sprite sheets and a common loading and drawing mechanism for all the entitites within our game.

The first thing we will do is define a `buildings` object inside `buildings.js`, as shown in Listing 6-2.

Listing 6-2. Defining the buildings Object with the First Building Type (buildings.js)

```
var buildings = {
list:{
    "base":{
        name:"base",
        // Properties for drawing the object
        pixelWidth:60,
        pixelHeight:60,
        baseWidth:40,
        baseHeight:40,
        pixelOffsetX:0,
        pixelOffsetY:20,
        // Properties for describing structure for pathfinding
        buildableGrid:[
            [1,1],
            [1,1]
        ],
        passableGrid:[
            [1,1],
            [1,1]
        ],
        sight:3,
        hitPoints:500,
        cost:5000,
        spriteImages:[
            {name:"healthy",count:4},
            {name:"damaged",count:1},
            {name:"contructing",count:3},
        ],
    },
},
defaults:{
    type:"buildings",
    animationIndex:0,
    direction:0,
    orders:{ type:"stand" },
    action:"stand",
    selected:false,
    selectable:true,
    // Default function for animating a building
    animate:function(){
    },
    // Default function for drawing a building
    draw:function(){
    }
},
load:loadItem,
add:addItem,
}
```

The buildings object has four important items.

- The list property will contain the definitions for all our buildings. For now, we define the base building along with properties that we will need later. These include properties for drawing the object (such as pixelWidth), properties for pathfinding (buildableGrid), general properties such as hitPoints and cost, and finally the list of sprite images.
- The defaults property contains properties and definitions common for all buildings. This includes a placeholder for the animate() and draw() methods that are commonly used by all buildings. We will implement these methods later.
- The load() method points to a common method for all the entities called loadItem() that we still need to define. This method will load the sprite sheet and definitions for a given entity.
- The add() method points to another common method for all the entities called addItem() that we need to define. This method will create a new instance of a given entity to be added to the game.

Now that we have a basic building definition in place, we will define the loadItem() and addItem() methods inside common.js so that they can be used by all the entities (see Listing 6-3).

Listing 6-3. Defining the loadItem() and addItem() Methods (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;
        }

    }
}

/* The default add() method used by all our game entities*/
function addItem(details){
    var item = {};
    var name = details.name;
    $.extend(item,this.defaults);
    $.extend(item,this.list[name]);
    item.life = item.hitPoints;
    $.extend(item,details);
    return item;
}
```

The `loadItem()` method uses the image loader to load the sprite sheet image into the `spriteSheet` property. It then goes through the `spriteImages` definition and creates a `spriteArray` object that stores the starting offsets for each of the sprite animations.

You will notice that the code checks for the existence of count and directions properties when creating the array. This allows us to define multidirectional sprites, which will be needed for drawing entities like turrets and vehicles.

The `addItem()` method first applies the defaults for the entity type (for example, buildings) and then extends it with properties for the specific entity (for example, base), sets the life for the item, and finally applies any additional properties passed into the details parameter.

This interesting way of creating objects gives us our own implementation of multiple inheritance, allowing us to define and override properties at three different levels: building properties, base properties, and item-specific details (such as position and team color).

Now that we have defined our first entity, we need a simple way of adding entities into a level.

Adding Entities to the Level

The first thing we will do is modify our map definition to include a list of entity types required to be loaded and a list of items to add to the level before it starts. We will modify the first map that we created in `maps.js`, as shown in Listing 6-4.

Listing 6-4. Loading and Adding Entities Inside the Map (maps.js)

```
var maps = {
    "singleplayer":[
        {
            "name":"Entities",
            "briefing": "In this level you will add new entities to the map.\nYou will also select
them using the mouse",

            /* Map Details */
            "mapImage":"images/maps/level-one-debug-grid.png",
            "startX":4,
            "startY":4,
```

```
            /* Entities to be loaded */
            "requirements":{
                "buildings":["base"],
                "vehicles":[],
                "aircraft":[],
                "terrain":[]
            },

            /* Entities to be added */
            "items":[
                {"type":"buildings","name":"base","x":11,"y":14,"team":"blue"},
                {"type":"buildings","name":"base","x":12,"y":16,"team":"green"},
                {"type":"buildings","name":"base","x":15,"y":15,"team":"green", "life":50}
            ]

        }
    ]
}
```

The map is very similar to the map from Chapter 5. We have added two new sections: `requirements` and `items`.

The `requirements` property contains the buildings, vehicles, aircraft, and terrain to preload for this level. For now, we load only buildings of type `base`.

The `items` array contains details of the entities we want to add to the level. The details we provide include the item type and name, the x and y grid coordinates, and the color of the team. These are the bare-minimum properties that we need in order to uniquely define an entity.

We have added three base buildings with random positions and teams. The last building in the `items` array also contains an additional property: `life`. Because of the way we defined the `addItem()` method earlier, this `life` property will override the default value of `life` for the base. This way, we will also have an example of a damaged building.

Next we will modify the `singleplayer.startCurrentLevel()` method in `singleplayer.js` to load and add the entities when the game starts (see Listing 6-5).

Listing 6-5. Loading and Adding Entities inside the startCurrentLevel() Method (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);
    };

    // 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 do three things in the newly added code. We first initialize the game arrays by calling the `game.resetArrays()` method. We then iterate through the `requirements` object and call the appropriate `load()` method for each entity. The `load()` methods in turn will call the loader to asynchronously load all the images for the entity in the background and enable the `entermission` button once all the images have been loaded.

Finally, we iterate through the `items` array and pass the details to the `game.add()` method.

Next we will add the `resetArrays()`, `add()`, and `remove()` methods to the game object inside `game.js` (see Listing 6-6).

Listing 6-6. Adding resetArrays(), add(), and remove() to the Game Object (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 = [];
},
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);
    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;
        }
    };
},
```

The `resetArrays()` method merely initializes all the game-specific arrays and the `counter` variable.

The `add()` method generates a unique identifier (UID) for an item using the counter, invokes the appropriate entity's `add()` method, and finally saves the item in the appropriate game arrays. For the base building, this method would first call `buildings.add()` and then add the new building to the `game.items` and `game.buildings` arrays.

The `remove()` method removes a specified item from the `selectedItems`, `items`, and `entity`-specific arrays. This way, any time an item is removed from the game (for example, when it is destroyed), it is automatically removed from the selection and the `items` array.

Now that we have set up the code for both defining the entity and adding entities to the level, we are ready to start drawing them on the screen.

Drawing the Entities

To draw the entities, we need to implement the `animate()` and `draw()` methods inside the entity object and then call these methods from the game `animationLoop()` and `drawingLoop()` methods.

We start by implementing the `draw()` and `animate()` methods inside the `buildings` object in `buildings.js`. The `buildings` object's default `draw()` and `animate()` methods will now look like Listing 6-7.

Listing 6-7. Implementing the Default draw() and animate() Methods (buildings.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":
            this.imageList = this.spriteArray[this.lifeCode];
            this.imageOffset = this.imageList.offset + this.animationIndex;
            this.animationIndex++;
            if (this.animationIndex>=this.imageList.count){
                this.animationIndex = 0;
            }
            break;
        case "construct":
            this.imageList = this.spriteArray["contructing"];
            this.imageOffset = this.imageList.offset + this.animationIndex;
            this.animationIndex++;
            // Once constructing is complete go back to standing
            if (this.animationIndex>=this.imageList.count){
                this.animationIndex = 0;
                this.action = "stand";
            }
            break;
    }
},
// Default function for drawing a building
draw:function(){
    var x = (this.x*game.gridSize)-game.offsetX-this.pixelOffsetX;
    var y = (this.y*game.gridSize)-game.offsetY-this.pixelOffsetY;

    // All sprite sheets will have blue in the first row and green in the second row
    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);
}
```

In the `animate()` method, we first set the `lifeCode` property of the item based on its `health` and `hitPoints`. Any time an item's health drops below 0, we set `lifeCode` to `dead` and remove it from the game.

Next we implement behavior based on the item's `action` property. For now, we implement only the `stand` and `construct` actions.

For the stand action, we choose either the "healthy" or "damaged" sprite animation and increment the animationIndex property. In case the animationIndex exceeds the number of frames in the sprite, we roll the value back to 0. This way, the animation rotates through every frame in the sprite again and again.

For the construct action, we display the constructing sprites and roll over into the stand action once it has completed.

The draw() method is relatively simpler. We calculate the absolute x and y pixel coordinates of the building by converting the grid x and y coordinates. We then calculate the correct image offset (based on animationIndex) and the image color row (based on team). Finally, we draw the appropriate image on the foreground canvas by using the foregroundContext.drawImage() method.

Now that the draw() and animate() methods are in place, we need to call them from the game object. We will modify the game.animationLoop() and game.drawingLoop() methods inside game.js, as shown in Listing 6-8.

Listing 6-8. Calling draw() and animate() from the Game Loops (game.js)

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

    // 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);
    }
},
```

Within the `animationLoop()` method, we first iterate through all the game items and call their `animate()` methods. We then sort all the items by y and then x values and store them in the `game.sortedItems` array.

The new code inside the `drawingLoop()` method merely iterates through the `sortedItems` array and calls the `draw()` method of each item. We use the `sortedItems` array so that items are drawn in order from back to front based on their y coordinates. This is a simple implementation of depth sorting that ensures that items closer to the player obscure items behind them, giving the illusion of depth.

With this last change, we are now ready to see our first game entity drawn on the screen. If we open the game in the browser and load the first level, we should see the three base buildings we defined in the map drawn next to each other (see Figure 6-2).

Figure 6-2. *The three base buildings*

As you can see, the first "blue" team base is shown with a a flashing blue light using the "healthy" animation.

The second "green" team base is drawn on top of the first one and partially obscures it. This is a result of our depth-sorting step and lets the player clearly see that the second base is in front of the first one.

Finally, the third base with a lower value of life looks damaged. This is because we automatically use the "damaged" animation whenever the life of the building is less than 40 percent of its maximum hit points.

Now that we have the framework for showing buildings within the game, let's add the remaining buildings, starting with the starport.

Adding the Starport

The starport is used to purchase both land and air units. The starport sprite sheet has a few interesting animations that the base did not have: a teleporting animation sequence that we will use when the building is first created and an opening and closing animation sequence that we will use when we transport in new units.

The first thing we will do is add the starport definition to the buildings list just below the base definition inside `buildings.js` (see Listing 6-9).

Listing 6-9. Definition for Starport Building (buildings.js)

```
"starport":{
    name:"starport",
    pixelWidth:40,
    pixelHeight:60,
    baseWidth:40,
    baseHeight:55,
    pixelOffsetX:1,
    pixelOffsetY:5,
    buildableGrid:[
        [1,1],
        [1,1],
        [1,1]
    ],
    passableGrid:[
        [1,1],
        [0,0],
        [0,0]
    ],
    sight:3,
    cost:2000,
    hitPoints:300,
    spriteImages:[
        {name:"teleport",count:9},
        {name:"closing",count:18},
        {name:"healthy",count:4},
        {name:"damaged",count:1},
    ],
},
```

Apart from the two new sprite sets, the starport definition is very similar to the base definition. Next, we will need to account for animating the opening, closing, and teleporting animation states. We will do this by modifying the default `animate()` method for the buildings inside `buildings.js`, as shown in Listing 6-10.

Listing 6-10. Modifying animate() to Handle Teleporting, Opening, and Closing

```
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":
        this.imageList = this.spriteArray[this.lifeCode];
        this.imageOffset = this.imageList.offset + this.animationIndex;
        this.animationIndex++;
        if (this.animationIndex>=this.imageList.count){
            this.animationIndex = 0;
        }
        break;
    case "construct":
        this.imageList = this.spriteArray["contructing"];
        this.imageOffset = this.imageList.offset + this.animationIndex;
        this.animationIndex++;
        // Once contructing is complete go back to standing
        if (this.animationIndex>=this.imageList.count){
            this.animationIndex = 0;
                this.action = "Stand";
        }
        break;
    case "teleport":
        this.imageList = this.spriteArray["teleport"];
        this.imageOffset = this.imageList.offset + this.animationIndex;
        this.animationIndex++;
        // Once teleporting is complete, move to either guard or stand mode
        if (this.animationIndex>=this.imageList.count){
            this.animationIndex = 0;
            if (this.canAttack){
                this.action = "guard";
            } else {
                this.action = "stand";
            }
        }
        break;
    case "close":
        this.imageList = this.spriteArray["closing"];
        this.imageOffset = this.imageList.offset + this.animationIndex;
        this.animationIndex++;
        // Once closing is complete go back to standing
        if (this.animationIndex>=this.imageList.count){
            this.animationIndex = 0;
            this.action = "stand";
        }
        break;
    case "open":
        this.imageList = this.spriteArray["closing"];
        // Opening is just the closing sprites running backwards
```

```
            this.imageOffset = this.imageList.offset + this.imageList.count - this.animationIndex;
            this.animationIndex++;
            // Once opening is complete, go back to close
            if (this.animationIndex>=this.imageList.count){
                this.animationIndex = 0;
                this.action = "close";
            }
            break;
    }
},
```

Like the `construct` animation state, the `teleport`, `close`, and `open` animation states do not keep repeating once they end. The `teleport` animation rolls over into the `stand` animation state (or the `guard` animation state for buildings that can attack such as the gun turret). The `open` animation (which is merely the close animation state running backward) rolls over into the `close` animation state, which then rolls over into the `stand` animation state.

This way, we can initialize the starport with a `teleport` or `open` animation state, knowing that it will eventually move back to the `stand` animation state once the current animation completes.

Now, we can add the starport to the map by modifying the requirements and items inside `maps.js`, as shown in Listing 6-11.

Listing 6-11. Adding the Starport to the Map

```
/* Entities to be loaded */
"requirements":{
    "buildings":["base","starport"],
    "vehicles":[],
    "aircraft":[],
    "terrain":[]
},

/* Entities to be added */
"items":[
    {"type":"buildings","name":"base","x":11,"y":14,"team":"blue"},
    {"type":"buildings","name":"base","x":12,"y":16,"team":"green"},
    {"type":"buildings","name":"base","x":15,"y":15,"team":"green", "life":50},

    {"type":"buildings","name":"starport","x":18,"y":14,"team":"blue"},
    {"type":"buildings","name":"starport","x":18,"y":10,"team":"blue", "action":"teleport"},
    {"type":"buildings","name":"starport","x":18,"y":6,"team":"green", "action":"open"},
]
```

When we open the game in the browser and start the level, we should see three new starport buildings, as shown in Figure 6-3.

Figure 6-3. *The three starport buildings*

The first green team starport opens and then closes. The second blue team starport first glows and comes into existence and then switches to stand mode, while the last blue team starport merely waits in stand mode.

Now that the starport has been added, the next building we will look at is the harvester.

Adding the Harvester

The harvester is a unique entity in the sense that it is both a building and a vehicle. Unlike the other buildings in the game, the harvester is created by deploying a harvester vehicle at an oil field where it turns into the building (see Figure 6-4).

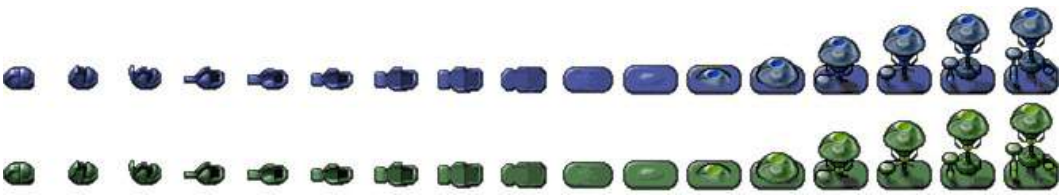

Figure 6-4. *Harvester deploying into building form*

The first thing we will do is add the harvester definition to the buildings list just below the starport definition inside `buildings.js` (see Listing 6-12).

Listing 6-12. Definition for Harvester Building (buildings.js)

```
"harvester":{
    name:"harvester",
    pixelWidth:40,
```

```
    pixelHeight:60,
    baseWidth:40,
    baseHeight:20,
    pixelOffsetX:-2,
    pixelOffsetY:40,
    buildableGrid:[
        [1,1]
    ],
    passableGrid:[
        [1,1]
    ],
    sight:3,
    cost:5000,
    hitPoints:300,
    spriteImages:[
        {name:"deploy",count:17},
        {name:"healthy",count:3},
        {name:"damaged",count:1},
    ],
},
```

Next, we will need to account for the deploying animation state. We will do this by adding the `deploy` case to the default `animate()` method inside `buildings.js`, as shown in Listing 6-13.

Listing 6-13. Handling the deploy Animation State (buildings.js)

```
case "deploy":
    this.imageList = this.spriteArray["deploy"];
    this.imageOffset = this.imageList.offset + this.animationIndex;
    this.animationIndex++;
    // Once deploying is complete, go back to stand
    if (this.animationIndex>=this.imageList.count){
        this.animationIndex = 0;
        this.action = "stand";
    }
    break;
```

The `deploy` state, like the teleport state we defined earlier, automatically rolls into the `stand` animation state once it completes.

Now, we can add the harvester to the map by modifying the requirements and items inside `maps.js`, as shown in Listing 6-14.

Listing 6-14. Adding the Harvester to the Map

```
/* Entities to be loaded */
"requirements":{
    "buildings":["base","starport","harvester"],
    "vehicles":[],
    "aircraft":[],
    "terrain":[]
},
```

```
/* Entities to be added */
"items":[
    {"type":"buildings","name":"base","x":11,"y":14,"team":"blue"},
    {"type":"buildings","name":"base","x":12,"y":16,"team":"green"},
    {"type":"buildings","name":"base","x":15,"y":15,"team":"green", "life":50},


    {"type":"buildings","name":"starport","x":18,"y":14,"team":"blue"},
    {"type":"buildings","name":"starport","x":18,"y":10,"team":"blue", "action":"teleport"},
    {"type":"buildings","name":"starport","x":18,"y":6,"team":"green", "action":"open"},

    {"type":"buildings","name":"harvester","x":20,"y":10,"team":"blue"},
    {"type":"buildings","name":"harvester","x":22,"y":12,"team":"green", "action":"deploy"},

]
```

When we open the game in the browser and start the level, we should see two new harvester buildings, as shown in Figure 6-5.

Figure 6-5. *The two harvester buildings*

The blue harvester is in the default stand mode, while the green harvester in deploy mode transforms into a building and then switches to stand mode.

Now that the harvester has been added, the last building we will look at the ground turret.

Adding the Ground Turret

The ground turret is a defensive structure that attacks only ground-based threats.

It is the only building that uses direction-based sprites. Also, unlike the other buildings, it has a default animation state of `guard`, which takes the turret's direction into account during animation and drawing.

The direction property can take values ranging from 0 to 7 increasing in the clockwise direction, with 0 pointing toward the north and 7 pointing in the northwest direction, as shown in Figure 6-6.

Figure 6-6. *Direction sprites for the gun turret ranging from 0 to 7*

The first thing we will do is add the gun turret definition to the buildings list just below the harvester definition inside buildings.js (see Listing 6-15).

Listing 6-15. Definition for Harvester Building (buildings.js)

```
"ground-turret":{
    name:"ground-turret",
    canAttack:true,
    canAttackLand:true,
    canAttackAir:false,
    weaponType:"cannon-ball",
    action:"guard", // Default action is guard unlike other buildings
    direction:0, // Face upward (0) by default
    directions:8, // Total of 8 turret directions allowed (0-7)
    orders:{type:"guard"},
    pixelWidth:38,
    pixelHeight:32,
    baseWidth:20,
    baseHeight:18,
    cost:1500,
    pixelOffsetX:9,
    pixelOffsetY:12,
    buildableGrid:[
        [1]
    ],
    passableGrid:[
        [1]
    ],
    sight:5,
    hitPoints:200,
    spriteImages:[
        {name:"teleport",count:9},
        {name:"healthy",count:1,directions:8},
        {name:"damaged",count:1},
    ],
}
```

The gun turret has a few additional properties that indicate whether it can be used to attack the enemy, the direction the turret is pointing, and the type of weapon it uses. We will use these properties later when we implement combat in our game.

The `healthy` sprites have an additional `directions` property that is used by the `itemLoad()` method to generate sprites for each direction.

Next, we will add the `guard` case to the `animate()` method inside `buildings.js`, as shown in Listing 6-16.

Listing 6-16. Handling the guard Animation State (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
        this.imageList = this.spriteArray[this.lifeCode+"-"+this.direction];
    }
     this.imageOffset = this.imageList.offset;
    break;
```

Unlike the previous animation states, the `guard` state does not use `animationIndex` and instead uses the turret direction to pick the appropriate image offset.

Now, we can add the turret to the map by modifying the requirements and items inside `maps.js`, as shown in Listing 6-17.

Listing 6-17. Adding the Ground Turret to the Map

```
/* Entities to be loaded */
"requirements":{
    "buildings":["base","starport","harvester","ground-turret"],
    "vehicles":[],
    "aircraft":[],
    "terrain":[]
},

/* Entities to be added */
"items":[
    {"type":"buildings","name":"base","x":11,"y":14,"team":"blue"},
    {"type":"buildings","name":"base","x":12,"y":16,"team":"green"},
    {"type":"buildings","name":"base","x":15,"y":15,"team":"green", "life":50},

    {"type":"buildings","name":"starport","x":18,"y":14,"team":"blue"},
    {"type":"buildings","name":"starport","x":18,"y":10,"team":"blue", "action":"teleport"},
    {"type":"buildings","name":"starport","x":18,"y":6,"team":"green", "action":"open"},

    {"type":"buildings","name":"harvester","x":20,"y":10,"team":"blue"},
    {"type":"buildings","name":"harvester","x":22,"y":12,"team":"green", "action":"deploy"},

    {"type":"buildings","name":"ground-turret","x":14,"y":9,"team":"blue", "direction":3},
    {"type":"buildings","name":"ground-turret","x":14,"y":12,"team":"green", "direction":1},
    {"type":"buildings","name":"ground-turret","x":16,"y":10,"team":"blue", "action":"teleport"},
]
```

We specify a starting direction property for the first two turrets and set the action property to `teleport` for the third. When we open the game in the browser and start the level, we should see three new turrets, as shown in Figure 6-7.

Figure 6-7. *The three ground turret buildings*

The first two turrets are in guard mode and face two different directions, while the third one teleports in facing the default direction and switches to guard mode after teleporting in.

At this point, we have implemented all the buildings that we need. Now it's time to start adding a few vehicles to our game.

Adding the Vehicles

All the vehicles in our game including the transport will have a simple sprite sheet with the vehicle pointing in eight directions similar to the ground turret, as shown in Figure 6-8.

Figure 6-8. *The transport sprite sheet*

We will set up the code for our vehicles by defining a new `vehicles` object inside `vehicles.js`, as shown in Listing 6-18.

Listing 6-18. Defining the vehicles Object (vehicles.js)

```
var vehicles = {
    list:{
        "transport":{
            name:"transport",
            pixelWidth:31,
            pixelHeight:30,
            pixelOffsetX:15,
            pixelOffsetY:15,
            radius:15,
            speed:15,
            sight:3,
            cost:400,
            hitPoints:100,
            turnSpeed:2,
            spriteImages:[
                {name:"stand",count:1,directions:8}
            ],
        },
        "harvester":{
            name:"harvester",
            pixelWidth:21,
            pixelHeight:20,
            pixelOffsetX:10,
            pixelOffsetY:10,
            radius:10,
            speed:10,
            sight:3,
            cost:1600,
            hitPoints:50,
            turnSpeed:2,
            spriteImages:[
                {name:"stand",count:1,directions:8}
            ],
        },
        "scout-tank":{
            name:"scout-tank",
            canAttack:true,
            canAttackLand:true,
            canAttackAir:false,
            weaponType:"bullet",
            pixelWidth:21,
            pixelHeight:21,
            pixelOffsetX:10,
            pixelOffsetY:10,
            radius:11,
            speed:20,
            sight:4,
```

```
            cost:500,
            hitPoints:50,
            turnSpeed:4,
            spriteImages:[
                {name:"stand",count:1,directions:8}
            ],
        },
        "heavy-tank":{
            name:"heavy-tank",
            canAttack:true,
            canAttackLand:true,
            canAttackAir:false,
            weaponType:"cannon-ball",
            pixelWidth:30,
            pixelHeight:30,
            pixelOffsetX:15,
            pixelOffsetY:15,
            radius:13,
            speed:15,
            sight:5,
            cost:1200,
            hitPoints:50,
            turnSpeed:4,
            spriteImages:[
                {name:"stand",count:1,directions:8}
            ],
        }
    },
    defaults:{
        type:"vehicles",
        animationIndex:0,
        direction:0,
        action:"stand",
        orders:{type:"stand"},
        selected:false,
        selectable:true,
        directions:8,
        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 = this.direction;
                    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;
            }
        },
        draw:function(){
            var x = (this.x*game.gridSize)-game.offsetX-this.pixelOffsetX;
            var y = (this.y*game.gridSize)-game.offsetY-this.pixelOffsetY;
            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);
        }
    },
    load:loadItem,
    add:addItem,
}
```

The structure of our `vehicles` object is very similar to the `buildings` object. We have a `list` property where we define the four vehicle types: the transport, the harvester, the scout tank, and the heavy tank.

All of the vehicle sprites have the `directions` property and the default `stand` animation implementation inside `animate()`, which uses the vehicle's direction to select the sprite to draw. We use `animationIndex` to handle multiple images within a sprite so that we can add vehicles with animation if needed.

The vehicles also have properties such as `speed`, `sight`, and `cost`. The transport and harvester do not have any weapons, while the two tanks have weapon-based properties similar to the ground turret building we defined earlier. We will use all of these properties in later chapters to implement movement and combat.

Now, we can add these vehicles to the map by modifying the `requirements` and `items` properties inside `maps.js`, as shown in Listing 6-19.

Listing 6-19. Adding the Vehicles to the Map

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

/* Entities to be added */
"items":[
    {"type":"buildings","name":"base","x":11,"y":14,"team":"blue"},
    {"type":"buildings","name":"base","x":12,"y":16,"team":"green"},
    {"type":"buildings","name":"base","x":15,"y":15,"team":"green", "life":50},
```

```
    {"type":"buildings","name":"starport","x":18,"y":14,"team":"blue"},
    {"type":"buildings","name":"starport","x":18,"y":10,"team":"blue", "action":"teleport"},
    {"type":"buildings","name":"starport","x":18,"y":6,"team":"green", "action":"open"},

    {"type":"buildings","name":"harvester","x":20,"y":10,"team":"blue"},
    {"type":"buildings","name":"harvester","x":22,"y":12,"team":"green", "action":"deploy"},

    {"type":"buildings","name":"ground-turret","x":14,"y":9,"team":"blue", "direction":3},
    {"type":"buildings","name":"ground-turret","x":14,"y":12,"team":"green", "direction":1},
    {"type":"buildings","name":"ground-turret" ,"x":16,"y":10, "team":"blue", "action":"teleport"},

    {"type":"vehicles","name":"transport","x":26,"y":10,"team":"blue","direction":2},
    {"type":"vehicles","name":"harvester","x":26,"y":12,"team":"blue","direction":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":"vehicles","name":"transport","x":28,"y":10,"team":"green", "direction":7},
    {"type":"vehicles","name":"harvester","x":28,"y":12,"team":"green", "direction":6},
    {"type":"vehicles","name":"scout-tank","x":28,"y":14,"team":"green", "direction":1},
    {"type":"vehicles","name":"heavy-tank","x":28,"y":16,"team":"green", "direction":0},
]
```

When we open the game in the browser and start the level, we should see the vehicles, as shown in Figure 6-9.

Figure 6-9. *Adding vehicles to the level*

The vehicles point in different directions based on the properties we set when adding them to the `items` list. With the vehicles implemented, it's time to add the aircraft to our game.

Adding the Aircraft

The aircraft in our game have a sprite sheet similar to vehicles except for one difference: shadows. The aircraft sprite sheet has a third row with shadows in it. Also, the chopper sprite sheet has multiple images for each direction, as shown in Figure 6-10.

Figure 6-10. *The chopper sprite sheet with shadows*

We will set up the code for our aircraft by defining a new `aircraft` object inside `aircraft.js`, as shown in Listing 6-20.

Listing 6-20. Defining the aircraft Object (aircraft.js)

```
var aircraft = {
    list:{
        "chopper":{
            name:"chopper",
            cost:900,
            pixelWidth:40,
            pixelHeight:40,
            pixelOffsetX:20,
            pixelOffsetY:20,
            weaponType:"heatseeker",
            radius:18,
            sight:6,
            canAttack:true,
            canAttackLand:true,
            canAttackAir:true,
            hitPoints:50,
            speed:25,
            turnSpeed:4,
            pixelShadowHeight:40,
            spriteImages:[
                {name:"fly",count:4,directions:8}
            ],
        },
        "wraith":{
            name:"wraith",
            cost:600,
            pixelWidth:30,
```

```
            pixelHeight:30,
            canAttack:true,
            canAttackLand:false,
            canAttackAir:true,
            weaponType:"fireball",
            pixelOffsetX:15,
            pixelOffsetY:15,
            radius:15,
            sight:8,
            speed:40,
            turnSpeed:4,
            hitPoints:50,
            pixelShadowHeight:40,
            spriteImages:[
                {name:"fly",count:1,directions:8}
            ],
        }
    },
    defaults:{
        type:"aircraft",
        animationIndex:0,
        direction:0,
        directions:8,
        action:"fly",
        selected:false,
        selectable:true,
        orders:{type:"float"},
        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 = this.direction;
                     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;
            }
        },
        draw:function(){
            var x = (this.x*game.gridSize)-game.offsetX-this.pixelOffsetX;
```

```
            var y = (this.y*game.gridSize)-game.offsetY-this.pixelOffsetY-this.pixelShadowHeight;
            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);
        }
    },
    load:loadItem,
    add:addItem,
}
```

The structure of our `aircraft` object is similar to the `vehicles` object. We have a `list` property where we define the two aircraft types: the chopper and the wraith.

All of the aircraft sprites have the `directions` property. The default `fly` animation implementation inside `animate()` uses the aircraft's direction to select the sprite to draw. In the case of the chopper, we also use `animationIndex` to handle multiple images for each direction.

The one big difference is in the way the `draw()` method is implemented. We draw a shadow at the location of the aircraft and draw the actual aircraft `pixelShadowHeight` pixels above the location of the aircraft. This way, the aircraft looks like it is floating above the ground and the shadow is on the ground below it.

Now, we can add these aircraft to the map by modifying the `requirements` and `items` properties inside `maps.js`, as shown in Listing 6-21.

Listing 6-21. Adding the Aircraft to the Map

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

/* Entities to be added */
"items":[
    {"type":"buildings","name":"base","x":11,"y":14,"team":"blue"},
    {"type":"buildings","name":"base","x":12,"y":16,"team":"green"},
    {"type":"buildings","name":"base","x":15,"y":15,"team":"green", "life":50},

    {"type":"buildings","name":"starport","x":18,"y":14,"team":"blue"},
    {"type":"buildings","name":"starport","x":18,"y":10,"team":"blue", "action":"teleport"},
    {"type":"buildings","name":"starport","x":18,"y":6,"team":"green", "action":"open"},

    {"type":"buildings","name":"harvester","x":20,"y":10,"team":"blue"},
    {"type":"buildings","name":"harvester","x":22,"y":12,"team":"green", "action":"deploy"},
```

```
    {"type":"buildings","name":"ground-turret","x":14,"y":9,"team":"blue", "direction":3},
    {"type":"buildings","name":"ground-turret","x":14,"y":12,"team":"green", "direction":1},
    {"type":"buildings","name":"ground-turret","x":16,"y":10,"team":"blue", "action":"teleport"},

    {"type":"vehicles","name":"transport","x":26,"y":10,"team":"blue","direction":2},
    {"type":"vehicles","name":"harvester","x":26,"y":12,"team":"blue","direction":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":"vehicles","name":"transport","x":28,"y":10,"team":"green", "direction":7},
    {"type":"vehicles","name":"harvester","x":28,"y":12,"team":"green", "direction":6},
    {"type":"vehicles","name":"scout-tank","x":28,"y":14,"team":"green", "direction":1},
    {"type":"vehicles","name":"heavy-tank","x":28,"y":16,"team":"green", "direction":0},
    {"type":"aircraft","name":"chopper","x":20,"y":22,"team":"blue", "direction":2},
    {"type":"aircraft","name":"wraith","x":23,"y":22,"team":"green", "direction":3},
]
```

When we open the game in the browser and start the level, we should see the aircraft hovering above the ground, as shown in Figure 6-11.

Figure 6-11. *The aircraft floating above the ground*

The shadows help create the illusion that the aircraft are floating above the ground and also mark their exact position on the ground. The chopper blades and their shadow on the ground seem to rotate because of the animation.

With the aircraft implemented, we will now add the terrain to our game.

Adding the Terrain

With the exception of the oil field, the terrain entities in our game are static bodies intended only for cosmetic use. The oil field is a special entity above which the harvester vehicle can deploy into the harvester building. The oil field sprite sheet includes two versions: a default version and a "hint" version that shows a blurry harvester above it as a hint for the player.

We will set up the code for our terrain by defining a new terrain object inside `terrain.js`, as shown in Listing 6-22.

Listing 6-22. Defining the Terrain Object (terrain.js)

```
var terrain = {
    list:{
        "oilfield":{
            name:"oilfield",
            pixelWidth:40,
            pixelHeight:60,
            baseWidth:40,
            baseHeight:20,
            pixelOffsetX:0,
            pixelOffsetY:40,
            buildableGrid:[
                [1,1]
            ],
            passableGrid:[
                [1,1]
            ],
            spriteImages:[
                {name:"hint",count:1},
                {name:"default",count:1},
            ],
        },
        "bigrocks":{
            name:"bigrocks",
            pixelWidth:40,
            pixelHeight:70,
            baseWidth:40,
            baseHeight:40,
            pixelOffsetX:0,
            pixelOffsetY:30,
            buildableGrid:[
                [1,1],
                [0,1]
            ],
            passableGrid:[
                [1,1],
                [0,1]
            ],
            spriteImages:[
                {name:"default",count:1},
            ],
        },
        "smallrocks":{
            name:"smallrocks",
            pixelWidth:20,
            pixelHeight:35,
            baseWidth:20,
            baseHeight:20,
```

```
            pixelOffsetX:0,
            pixelOffsetY:15,
            buildableGrid:[
                [1]
            ],
            passableGrid:[
                [1]
            ],
            spriteImages:[
                {name:"default",count:1},
            ],
        },
    },
    defaults:{
        type:"terrain",
        animationIndex:0,
        action:"default",
        selected:false,
        selectable:false,
        animate:function(){
            switch (this.action){
                case "default":
                     this.imageList = this.spriteArray["default"];
                     this.imageOffset = this.imageList.offset + this.animationIndex;
                     this.animationIndex++;
                     if (this.animationIndex>=this.imageList.count){
                         this.animationIndex = 0;
                     }
                break;
                case "hint":
                    this.imageList = this.spriteArray["hint"];
                    this.imageOffset = this.imageList.offset + this.animationIndex;
                    this.animationIndex++;
                    if (this.animationIndex>=this.imageList.count){
                        this.animationIndex = 0;
                    }
                break;
            }
        },
        draw:function(){
            var x = (this.x*game.gridSize)-game.offsetX-this.pixelOffsetX;
            var y = (this.y*game.gridSize)-game.offsetY-this.pixelOffsetY;

            var colorOffset = 0; // No team based colors
            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 structure of our terrain object is similar to the `buildings` object. We have a `list` property where we define the terrain types: the oil field, the big rocks, and the small rocks. We implement a `default` and a `hint` animation state inside the `animate()` method. We also implement a simpler `draw()` method that does not use team-based colors.

Now, we can add these terrain to the map by modifying the requirements and items inside `maps.js`, as shown in Listing 6-23.

Listing 6-23. Adding the Terrain to the Map

```
/* 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":"base","x":12,"y":16,"team":"green"},
    {"type":"buildings","name":"base","x":15,"y":15,"team":"green", "life":50},

    {"type":"buildings","name":"starport","x":18,"y":14,"team":"blue"},
    {"type":"buildings","name":"starport","x":18,"y":10,"team":"blue", "action":"teleport"},
    {"type":"buildings","name":"starport","x":18,"y":6,"team":"green", "action":"open"},

    {"type":"buildings","name":"harvester","x":20,"y":10,"team":"blue"},
    {"type":"buildings","name":"harvester","x":22,"y":12,"team":"green", "action":"deploy"},

    {"type":"buildings","name":"ground-turret","x":14,"y":9,"team":"blue","direction":3},
    {"type":"buildings","name":"ground-turret","x":14,"y":12,"team":"green","direction":1},
    {"type":"buildings","name":"ground-turret","x":16,"y":10,"team":"blue","action":"teleport"},

    {"type":"vehicles","name":"transport","x":26,"y":10,"team":"blue", "direction":2},
    {"type":"vehicles","name":"harvester","x":26,"y":12,"team":"blue", "direction":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":"vehicles","name":"transport","x":28,"y":10,"team":"green", "direction":7},
    {"type":"vehicles","name":"harvester","x":28,"y":12,"team":"green", "direction":6},
    {"type":"vehicles","name":"scout-tank","x":28,"y":14,"team":"green","direction":1},
    {"type":"vehicles","name":"heavy-tank","x":28,"y":16,"team":"green","direction":0},

    {"type":"aircraft","name":"chopper","x":20,"y":22,"team":"blue", "direction":2},
    {"type":"aircraft","name":"wraith","x":23,"y":22,"team":"green", "direction":3},

    {"type":"terrain","name":"oilfield","x":5,"y":7},
    {"type":"terrain","name":"oilfield","x":8,"y":7,"action":"hint"},

    {"type":"terrain","name":"bigrocks","x":5,"y":3},
    {"type":"terrain","name":"smallrocks","x":8,"y":3},
]
```

We add two oil fields, one of which has the `action` property set to `hint`. When we open the game in the browser and start the level, we should see the rocks and the oil fields, as shown in Figure 6-12.

Figure 6-12. *Adding the rocks and the oil fields*

The oil field on the right with the hint has a subtle glowing image of a harvester to let the player know that a harvester can be deployed there. This hint version of the oil field can be used in the earlier levels of our campaign when the player has just been introduced to the idea of harvesting.

With this, we have implemented all the important entities in the game. Of course, at this point all we can do is look at them. The next thing we would like to do is interact with them by selecting them.

Selecting Game Entities

We will allow players to select entities either by clicking them or by dragging a selection box across them.

We will enable click selection by modifying the mouse object inside `mouse.js`, as shown in Listing 6-24.

Listing 6-24. Enabling Selection by Clicking (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
    }
},
itemUnderMouse:function(){
    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;
            }
        }
    }
},
```

The `mouse.click()` method first checks whether there is an item under the mouse during the click using the `itemUnderMouse()` method. In case an item was under the mouse and the left button was clicked, we call the `game.selectItem()` method. The `game.clearSelection()` method is called before selecting the new item unless the Shift key is pressed during the click. This way, users can select multiple items by holding down the Shift key while selecting.

The `itemUnderMouse()` method iterates through all the items in the list and returns the first item that is under the mouse `gameX` and `gameY` coordinates using different criteria for different item types.

- In the case of buildings and terrain, we check whether the base of the item is under the mouse. This way, the player can click the base of a building to select it but won't have problems selecting vehicles behind the building.
- In the case of vehicles, we check whether the mouse is within a radius from the vehicle center.
- In the case of aircraft, we check whether the mouse is within a radius from the aircraft center and not the shadow by using the `pixelShadowHeight` property.

Next we will handle drag selection by modifying the `mouseup` event handler inside the `init()` method of the mouse object (see Listing 6-25).

Listing 6-25. Implementing Drag Selection in the mouseup Event Handler (mouse.js)

```
$mouseCanvas.mouseup(function(ev) {
    var shiftPressed = ev.shiftKey;
    if(ev.which==1){
    //Left key was released
        if (mouse.dragSelect){
            if (!shiftPressed){
                // Shift key was not pressed
                game.clearSelection();
            }

            var x1 = Math.min(mouse.gameX,mouse.dragX)/game.gridSize;
            var y1 = Math.min(mouse.gameY,mouse.dragY)/game.gridSize;
            var x2 = Math.max(mouse.gameX,mouse.dragX)/game.gridSize;
            var y2 = Math.max(mouse.gameY,mouse.dragY)/game.gridSize;
            for (var i = game.items.length - 1; i >= 0; i--){
                var item = game.items[i];
                if (item.type != "buildings" && item.selectable && item.team==game.team && x1<=
item.x && x2 >= item.x){
                    if ((item.type == "vehicles" && y1<= item.y && y2 >= item.y)
                    || (item.type == "aircraft" && (y1 <= item.y-item.pixelShadowHeight/game.
gridSize) && (y2 >= item.y-item.pixelShadowHeight/game.gridSize))){
                        game.selectItem(item,shiftPressed);
                    }

                }
            };
        }
        mouse.buttonPressed = false;
        mouse.dragSelect = false;
    }
    return false;
});
```

Inside the `mouseup` event, we check whether the mouse had been dragged and, if so, iterate through every game item and check whether it lies within the bounds of the dragged rectangle. We then select the appropriate items.

Most importantly, we only allow drag selection for our own vehicles and aircraft and not for enemy entities or our own buildings. This is because drag selection is typically used to select groups of units to move them or attack with them quickly, and selecting enemy units or our own buildings does not really help the player.

Next, we will add some selection-related code to the game object inside `game.js`, as shown in Listing 6-26.

Listing 6-26. Adding Selection-Related Code to Game Object (game.js)

```
/* Selection Related Code */
selectionBorderColor:"rgba(255,255,0,0.5)",
selectionFillColor:"rgba(255,215,0,0.2)",
healthBarBorderColor:"rgba(0,0,0,0.8)",
healthBarHealthyFillColor:"rgba(0,255,0,0.5)",
healthBarDamagedFillColor:"rgba(255,0,0,0.5)",
lifeBarHeight:5,
clearSelection:function(){
```

```
    while(game.selectedItems.length>0){
        game.selectedItems.pop().selected = false;
    }
},
selectItem:function(item,shiftPressed){
    // Pressing shift and clicking on a selected item will deselect it
    if (shiftPressed && item.selected){
        // deselect item
        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;
            }
        };
        return;
    }

    if (item.selectable && !item.selected){
        item.selected = true;
        game.selectedItems.push(item);
    }
},
```

We start by defining a few common selection-based properties related to colors and life bars. We then define the two methods used for selection.

- The `clearSelection()` method iterates through the `game.selectedItems` array, clears the selected flag from each item, and removes the item from the array.
- The `selectItem()` method either adds a selectable item to the `selectedItems()` array or removes it from the array depending on whether the Shift key is pressed. This way, players can unselect a selected item by clicking it with the Shift key pressed.

At this point, we have all the code we need to select items inside the game. However, we still need a way to highlight selected items so we can identify them visually. This is what we will implement next.

Highlighting Selected Entities

When the player selects an item, we will detect it using the item's selected property and draw an enclosing selection boundary around the item. We will also add an indicator to show us how much life the item has.

We will do this by defining two default methods, `drawSelection()` and `drawLifeBar()`, for each of the entities and modify the `draw()` method to call them.

First, we will implement these methods in the `buildings` object (see Listing 6-27).

Listing 6-27. Implementing drawSelection() and drawLifeBar() for Buildings (buildings.js)

```
drawLifeBar:function(){
    var x = this.drawingX+ this.pixelOffsetX;
    var y = this.drawingY - 2*game.lifeBarHeight;
```

```
    game.foregroundContext.fillStyle = (this.lifeCode == "healthy") ?
game.healthBarHealthyFillColor: game.healthBarDamagedFillColor;

game.foregroundContext.fillRect(x,y,this.baseWidth*this.life/this.hitPoints,game.lifeBarHeight)

    game.foregroundContext.strokeStyle = game.healthBarBorderColor;
    game.foregroundContext.lineWidth = 1;
    game.foregroundContext.strokeRect(x,y,this.baseWidth,game.lifeBarHeight)
},
drawSelection:function(){
    var x = this.drawingX + this.pixelOffsetX;
    var y = this.drawingY + this.pixelOffsetY;
    game.foregroundContext.strokeStyle = game.selectionBorderColor;
    game.foregroundContext.lineWidth = 1;
    game.foregroundContext.fillStyle = game.selectionFillColor;
    game.foregroundContext.fillRect(x-1,y-1,this.baseWidth+2,this.baseHeight+2);
    game.foregroundContext.strokeRect(x-1,y-1,this.baseWidth+2,this.baseHeight+2);
},
// Default function for drawing a building
draw:function(){
    var x = (this.x*game.gridSize)-game.offsetX-this.pixelOffsetX;
    var y = (this.y*game.gridSize)-game.offsetY-this.pixelOffsetY;
    this.drawingX = x;
    this.drawingY = y;
    if (this.selected){
        this.drawSelection();
        this.drawLifeBar();
    }
    // All sprite sheets will have blue in the first row and green in the second row
    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);
}
```

The `drawLifeBar()` method merely draws a bar slightly above the building with a green or red color depending on the life of the building. The length of the bar is proportional to the life of the building. The `drawSelection()` method draws a yellow rectangle around the base of the building. Finally, we call both these methods if the item is selected from inside the `draw()` method.

Next we will implement these methods for the `vehicles` object (see Listing 6-28).

Listing 6-28. Implementing drawSelection() and drawLifeBar() for Vehicles (vehicles.js)

```
drawLifeBar:function(){
    var x = this.drawingX;
    var y = this.drawingY - 2*game.lifeBarHeight;
    game.foregroundContext.fillStyle = (this.lifeCode == "healthy")?game.
healthBarHealthyFillColor:game.healthBarDamagedFillColor;

game.foregroundContext.fillRect(x,y,this.pixelWidth*this.life/this.hitPoints,game.lifeBarHeight)
    game.foregroundContext.strokeStyle = game.healthBarBorderColor;
    game.foregroundContext.lineWidth = 1;
    game.foregroundContext.strokeRect(x,y,this.pixelWidth,game.lifeBarHeight)
},
```

```
drawSelection:function(){
    var x = this.drawingX + this.pixelOffsetX;
    var y = this.drawingY + this.pixelOffsetY;
    game.foregroundContext.strokeStyle = game.selectionBorderColor;
    game.foregroundContext.lineWidth = 1;
    game.foregroundContext.beginPath();
    game.foregroundContext.arc(x,y,this.radius,0,Math.PI*2,false);
    game.foregroundContext.fillStyle = game.selectionFillColor;
    game.foregroundContext.fill();
    game.foregroundContext.stroke();
},
draw:function(){
    var x = (this.x*game.gridSize)-game.offsetX-this.pixelOffsetX;
    var y = (this.y*game.gridSize)-game.offsetY-this.pixelOffsetY;
    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);
}
```

This time, the `drawSelection()` method draws a yellow, lightly filled circle under the selected vehicle. Like before, the `drawLifeBar()` method draws a life bar above the vehicle.

Lastly, we will implement these methods for the `aircraft` object (see Listing 6-29).

Listing 6-29. Implementing drawSelection() and drawLifeBar() for Aircraft (aircraft.js)

```
drawLifeBar:function(){
    var x = this.drawingX;
    var y = this.drawingY - 2*game.lifeBarHeight;
    game.foregroundContext.fillStyle = (this.lifeCode ==
    "healthy")?game.healthBarHealthyFillColor:game.healthBarDamagedFillColor;

game.foregroundContext.fillRect(x,y,this.pixelWidth*this.life/this.hitPoints,game.lifeBarHeight)
    game.foregroundContext.strokeStyle = game.healthBarBorderColor;
    game.foregroundContext.lineWidth = 1;
    game.foregroundContext.strokeRect(x,y,this.pixelWidth,game.lifeBarHeight)
},
drawSelection:function(){
    var x = this.drawingX + this.pixelOffsetX;
    var y = this.drawingY + this.pixelOffsetY;
    game.foregroundContext.strokeStyle = game.selectionBorderColor;
    game.foregroundContext.lineWidth = 2;
    game.foregroundContext.beginPath();
    game.foregroundContext.arc(x,y,this.radius,0,Math.PI*2,false);
    game.foregroundContext.stroke();
    game.foregroundContext.fillStyle = game.selectionFillColor;
    game.foregroundContext.fill();
```

```
    game.foregroundContext.beginPath();
    game.foregroundContext.arc(x,y+this.pixelShadowHeight,4,0,Math.PI*2,false);
    game.foregroundContext.stroke();

    game.foregroundContext.beginPath();
    game.foregroundContext.moveTo(x,y);
    game.foregroundContext.lineTo(x,y+this.pixelShadowHeight);
    game.foregroundContext.stroke();
},
draw:function(){
    var x = (this.x*game.gridSize)-game.offsetX-this.pixelOffsetX;
    var y = (this.y*game.gridSize)-game.offsetY-this.pixelOffsetY-this.pixelShadowHeight;
    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);
}
```

This time, the `drawLifeBar()` method adjusts for the shadow height when drawing the life bar. The `drawSelection()` method draws a yellow circle around the aircraft, a straight line from the aircraft to the shadow, and finally a small circle at the center of the shadow.

With this last change, we have implemented drawing selections for all the entities. We don't need to implement selections for the terrain since it cannot be selected within the game.

If we run the game in our browser, we should now be able to select items by either clicking them or dragging the mouse over multiple units. These selected items should then show up highlighted, as shown in Figure 6-13.

Figure 6-13. *Selected items show up highlighted*

Notice that the life bar above the damaged building clearly shows us how badly damaged it is. You can add or subtract items from the selection by clicking them with the Shift key pressed. We have now completely implemented entity selection in our game.

Summary

We covered a lot of ground in this chapter. Starting with an empty level from the previous chapter, we developed a general framework for animating and drawing items within the game by implementing `draw()` and `animate()` methods for these entities.

We handled depth sorting before drawing the items so that items closer to the screen obscured items that were farther away. Using this framework, we then added buildings, vehicles, aircraft, and terrain to our game.

Finally, we implemented the ability to select these entities using the mouse and highlight these selected entities.

In the next chapter, we will implement sending commands to these entities starting with the most important one: movement. We will also look at using pathfinding and steering algorithms so that units navigate intelligently around buildings and other obstacles.

So, let's keep going.