Implementing the Game Interface

The first thing we will do is add the HTML markup for the game interface screen into the `gamecontainer div` in our HTML file. The `gamecontainer div` will now look like Listing 6-11.

Listing 6-11. Adding the Game Interface Layer (index.html)

```
<div id="gamecontainer">
    <div id="gamestartscreen" class="gamelayer">
        <span class="game-title">LAST<br>COLONY</span>
        <span class="game-option" onclick = "singleplayer.start();">Campaign</span>
        <span class="game-option" onclick = "multiplayer.start();">Multiplayer</span>
    </div>

    <div id="missionbriefingscreen" class="gamelayer">
        <img src="images/screens/interface-left.png" class="left-panel" draggable="false">
        <img src="images/screens/interface-right-briefing.png" class="right-panel"
        draggable="false">
        <input type="button" id="entermission" onclick = "singleplayer.play();">
        <input type="button" id="exitmission" onclick = "singleplayer.exit();">
        <div id="missionbriefing"></div>
    </div>
```

```
    <div id="gameinterfacescreen" class="gamelayer">
        <img src="images/screens/interface-left.png" class="left-panel" draggable="false">
        <img src="images/screens/interface-right-game.png" class="right-panel"
        draggable="false">
        <div id="gamemessages"></div>
        <div id="callerpicture"></div>
        <div id="cash"></div>
        <div id="sidebarbuttons">
        </div>
        <canvas id="gamebackgroundcanvas"></canvas>
        <canvas id="gameforegroundcanvas"></canvas>
    </div>

    <div id="loadingscreen" class="gamelayer">
        <div id="loadingmessage"></div>
    </div>
</div>
```

Our game interface layer consists of several different areas within it:

- *Game area*: This is where the player can see the map and interact with the buildings, units, and other entities within the game. This is implemented using two canvas elements: `gamebackgroundcanvas` for the map and `gameforegroundcanvas` for the entities inside the level (such as buildings and units).
- *Game messages*: This is where the player can see system notifications or story-driven messages.
- *Caller picture*: This is where the player will see profile pictures of the person sending story-driven messages.
- *Cash*: This is where players will see their cash reserves.
- *Sidebar buttons*: This is where players will see buttons they can use for creating units and buildings within the game.

We also use left and right background images just as we did in the mission briefing screen.

Now that the HTML is in place, we will add the CSS for the game interface screen to `styles.css`, as shown in Listing 6-12.

Listing 6-12. CSS for the Game Interface Screen

```
/* Game Interface Screen */

#gameinterfacescreen {
    background: url("images/screens/interface-middle.png");
}

#gameinterfacescreen #gamemessages {
    position: absolute;
    padding: 5px;

    top: 4px;
    left: 5px;
```

```
    right: 168px;
    height: 60px;

    color: rgb(130, 150, 162);

    overflow: hidden;
    font-size: 13px;
    font-family: "Courier New", Courier, monospace;
}

#gamemessages span {
    color: white;
}

#callerpicture {
    position: absolute;

    right: 20px;
    top: 155px;
    width: 114px;
    height: 72px;

    overflow: hidden;
}

#cash {
    width: 120px;
    height: 22px;
    position: absolute;
    right: 20px;
    top: 241px;

    color: rgb(130, 150, 162);
    overflow: hidden;
    font-size: 14px;
    font-family: "Courier New", Courier, monospace;
    text-align: right;
}

#gameinterfacescreen canvas {
    position: absolute;
    top: 79px;
    left: 0;
}
```

We start by defining a background for the center of the `gameinterfacescreen div`, just as we did for the `gamebriefingscreen div`, and then position the various other elements at the appropriate locations within the interface area. Both game canvas elements are positioned at the same location, with `foregroundcanvas` on top of `backgroundcanvas`.

Next we will modify the `init()` method of the game object to initialize the canvas elements when the game is initialized, as shown in Listing 6-13.

Listing 6-13. Initializing the Canvas Elements (game.js)

```
// Start initializing objects, preloading assets, and display start screen
init: function() {
    // Initialize objects
    loader.init();

    // Initialize and store contexts for both the canvases
    game.initCanvases();

    // Display the main game menu
    game.hideScreens();
    game.showScreen("gamestartscreen");
},

canvasWidth: 480,
canvasHeight: 400,

initCanvases: function() {
    game.backgroundCanvas = document.getElementById("gamebackgroundcanvas");
    game.backgroundContext = game.backgroundCanvas.getContext("2d");

    game.foregroundCanvas = document.getElementById("gameforegroundcanvas");
    game.foregroundContext = game.foregroundCanvas.getContext("2d");

    game.foregroundCanvas.width = game.canvasWidth;
    game.backgroundCanvas.width = game.canvasWidth;

    game.foregroundCanvas.height = game.canvasHeight;
    game.backgroundCanvas.height = game.canvasHeight;
},
```

We add a call to the `initCanvases()` method, which stores the `canvas` and `context` objects and sets their initial width and height.

We also need to handle resizing the canvas elements whenever the window size changes. We will do this by modifying the `game.resize()` method as shown in Listing 6-14.

Listing 6-14. Resizing the Canvas Elements (game.js)

```
resize: function() {

    var maxWidth = window.innerWidth;
    var maxHeight = window.innerHeight;

    var scale = Math.min(maxWidth / 640, maxHeight / 480);

    var gameContainer = document.getElementById("gamecontainer");

    gameContainer.style.transform = "translate(-50%, -50%) " + "scale(" + scale + ")";

    game.scale = scale;
```

```
    // What is the maximum width we can set based on the current scale
    // Clamp the value between 640 and 1024
    var width = Math.max(640, Math.min(1024, maxWidth / scale ));

    // Apply this new width to game container and game canvas
    gameContainer.style.width = width + "px";

    // Subtract 160px for the sidebar
    var canvasWidth = width - 160;

    // Set a flag in case the canvas was resized
    if (game.canvasWidth !== canvasWidth) {
        game.canvasWidth = canvasWidth;
        game.canvasResized = true;
    }

},
```

We calculate the new width for the canvas based on the container width that we calculated earlier. If the value has changed, we also set a `canvasResized` flag to `true`. We will use this flag inside our drawing loop to decide whether we need to redraw parts of the game.

Now we will implement animation and drawing loops, as well as a `game.start()` method in our game, as shown in Listing 6-15.

Listing 6-15. Adding Animation and Drawing Loops and Starting the Game(game.js)

```
start: function() {
    // Display the game interface
    game.hideScreens();
    game.showScreen("gameinterfacescreen");

    game.running = true;
    game.refreshBackground = true;
    game.canvasResized = true;

    game.drawingLoop();
},

// A control loop that runs at a fixed period of time
animationTimeout: 100, // 100 milliseconds or 10 times a second

animationLoop: function() {
},

// The map is broken into square tiles of this size (20 pixels x 20 pixels)
gridSize: 20,
// X & Y panning offsets for the map
offsetX: 0,
offsetY: 0,
```

```
drawingLoop: function() {
    // Draw the background whenever necessary
    game.drawBackground();

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

drawBackground: function() {
    // Since drawing the background map is a fairly large operation,
    // we only redraw the background if it changes (due to panning or resizing)
    if (game.refreshBackground || game.canvasResized) {
        if (game.canvasResized) {
            game.backgroundCanvas.width = game.canvasWidth;
            game.foregroundCanvas.width = game.canvasWidth;

            // Ensure the resizing doesn't cause the map to pan out of bounds
            if (game.offsetX + game.canvasWidth > game.currentMapImage.width) {
                game.offsetX = game.currentMapImage.width - game.canvasWidth;
            }

            if (game.offsetY + game.canvasHeight > game.currentMapImage.height) {
                game.offsetY = game.currentMapImage.height - game.canvasHeight;
            }

            game.canvasResized = false;
        }

        game.backgroundContext.drawImage(game.currentMapImage, game.offsetX, game.offsetY,
        game.canvasWidth, game.canvasHeight, 0, 0, game.canvasWidth, game.canvasHeight);
        game.refreshBackground = false;
    }
},
```

We define a `start()` method that hides other layers and displays the game interface screen. It then sets the `game.running`, `game.backgroundChanged`, and `game.canvasResized` variables to `true` for later use. Finally, we call the `drawingLoop()` method for the first time.

We also define two different methods called `animationLoop()` and `drawingLoop()`. The `animationLoop()` method will handle all control-related and animation-related logic and needs to be run at a fixed interval (defined in `animationTimeout`). An animation timeout of 100 milliseconds is usually sufficient for a fairly smooth game. For now the `animationLoop()` method is empty. The `drawingLoop()` method handles the actual drawing of all the game elements onto the two game canvas objects. The method is called using `requestAnimationFrame()` and will run as many times a second as the browser allows.

We start by calling the `game.drawBackground()` method, which will draw the map on the background canvas whenever necessary.

We then call the `drawingLoop()` method again using `requestAnimationFrame()` if the game is still running. This way, once the `drawingLoop()` method has been called once, it will keep running and drawing the game until `game.running` becomes false.

In the `drawBackground()` method, the first thing that we do is check the `canvasResized` and `refreshBackground` flags to determine whether the background needs to be redrawn.

If the canvas was resized, we also check and adjust the panning offsets to ensure that the screen doesn't pan outside the map bounds. We then draw the map image (stored in `currentMapImage` when the map was loaded) using the panning offsets (`offsetX`, `offsetY`) and the canvas dimensions.

Finally, we reset both the flags to `false`. We use this optimization so that we don't need to redraw the entire background after each refresh, and only do so when something has actually changed.

The reason we break out the code into two different timer loops is because the animation code will contain logic such as pathfinding, processing commands, and changing the animation states of sprites, which will not need to be executed as often as the drawing code.

The animation code will also control the actual movement of units. By keeping this code independent of the drawing code, we ensure that units will move the same amount after each animation cycle. This will become very important when we handle multiplayer mode and need the game state to be synchronized across different machines. If we aren't careful, slight calculation differences between browsers and machines can cause unexpected results such as a bullet hitting an enemy unit in one browser but missing the enemy in the other browser.

Now that we have these loops in place, we will finally implement the `singleplayer.play()` method inside `singleplayer.js`, as shown in Listing 6-16.

Listing 6-16. The singleplayer.play() Method (singleplayer.js)

```
play: function() {
    // Run the animation loop once
    game.animationLoop();

    // Start the animation loop interval
    game.animationInterval = setInterval(game.animationLoop, game.animationTimeout);

    game.start();
},
```

This method is fairly simple. It calls the `game.animationLoop()` method for the first time and then uses the `setInterval()` method to call the method every 100 milliseconds (as set in `game.animationTimeout`). Finally, it calls the `game.start()` method that we defined earlier. The `game.animationLoop()` method is currently empty, but we will start using it when we add entities to our game in the next chapter.

If we run the game code we have so far, we should be able to click the Enter Mission button at the mission briefing screen and then see the game interface screen with the map loaded, as shown in Figure 6-5.

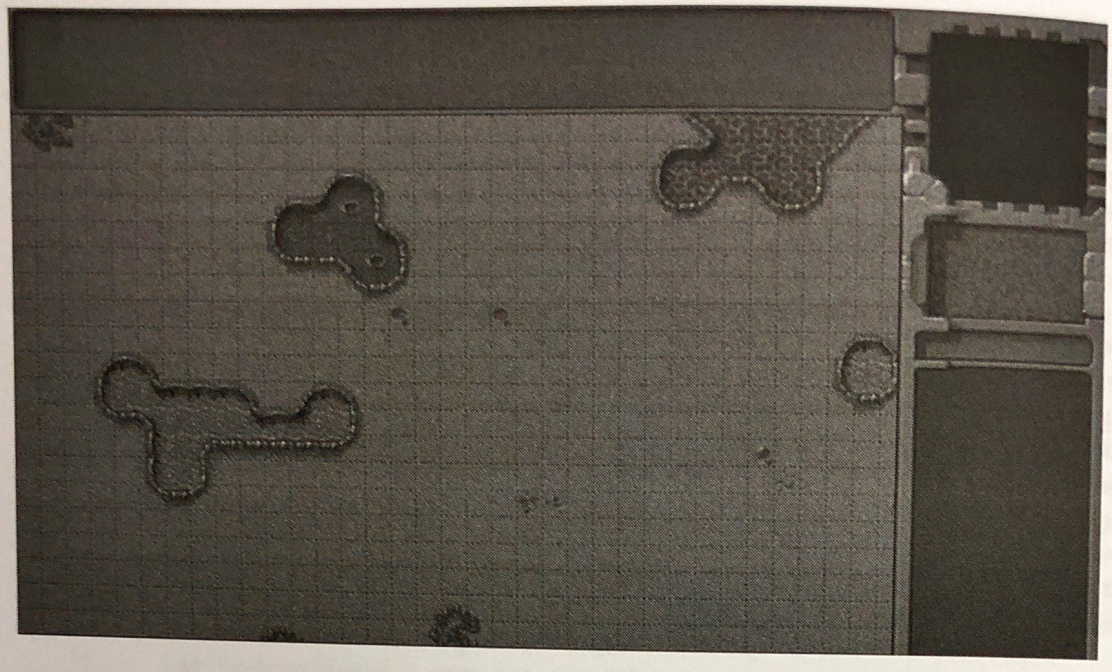

Figure 6-5. *The game interface screen with the first map loaded*

You can even resize the browser window and see that the game automatically shows more of the map as the window gets wider.

One thing you might notice is that the game starts off at the top-left corner of the map. To use the initial map offset settings that we provided in `levels.js`, we will need to load the offset values when we start the level. We will do this by modifying the `initLevel()` method in `singleplayer.js`, as shown in Listing 6-17.

Listing 6-17. Setting the Map Offset Inside initLevel() (singleplayer.js)

```
initLevel: function() {
    game.type = "singleplayer";
    game.team = "blue";

    // Don't allow player to enter mission until all assets for the level are loaded
    var enterMissionButton = document.getElementById("entermission");

    enterMissionButton.disabled = true;

    // Load all the items for the level
    var level = levels.singleplayer[singleplayer.currentLevel];

    game.loadLevelData(level);

    // Set player starting location
    game.offsetX = level.startX * game.gridSize;
    game.offsetY = level.startY * game.gridSize;

    // Enable the Enter Mission button once all assets are loaded
    loader.onload = function() {
        enterMissionButton.disabled = false;
    };
```

```
    // Update the mission briefing text and show briefing screen
    this.showMissionBriefing(level.briefing);
},
```

We added just two new lines to set `game.offsetX` and `game.offsetY` based on `level.startX` and `level.startY`. This time when we load the map, it loads at the offset we defined in the map.

Now that we have finished loading the map, we will implement panning around the map using the mouse.