```
var dy = 0;

function eventShipLoaded() {
  startUp();
}

function drawScreen() {
   x = x+dx;
   y = y+dy;

  //draw a background so we can see the Canvas edges
  context.fillStyle = "#aaaaaa";
  context.fillRect(0,0,500,500);

  context.save();
  context.setTransform(1,0,0,1,0,0)
  var angleInRadians = rotation * Math.PI / 180;
  context.translate(x+16, y+16)
  context.rotate(angleInRadians);
  var sourceX=Math.floor(animationFrames[frameIndex] % 8) *32;
  var sourceY=Math.floor(animationFrames[frameIndex] / 8) *32;

  context.drawImage(tileSheet, sourceX, sourceY,32,32,-16,-16,32,32);
  context.restore();

  frameIndex++;
  if (frameIndex ==animationFrames.length) {
     frameIndex=0;
  }

}

function startUp(){

  setInterval(drawScreen, 100 );
}
```

When Example 4-9 is running, you will see the tank move slowly across the screen to the right. Its tracks animate through the series of tiles from the tile sheet on a plain gray background.

So far, we have only used tiles to simulate sprite-based animated movement. In the next section, we will examine how to use an image tile sheet to create a much more elaborate background using a series of tiles.

Creating a Grid of Tiles

Many games use what is called a *tile-based environment* for backgrounds and level graphics. We are now going to apply the knowledge we have learned from animating an image on the canvas to create the background maze for our hypothetical game: *No Tanks!* We will use the same tile sheet from the previous tank examples, but instead of showing the tank sprite tiles, we will create a maze for the tank to move through. We will not actually cover the game-play portion of the code in this chapter because we want to focus on using images to render the screen. In Chapter 9 we will create a simple game using the type of examples shown here.

Defining a Tile Map

We will use the term *tile map* to refer to a game level or background built from a tile sheet. Take a look back at Figure 4-7—the four row by eight column tile sheet from earlier in this chapter. If we were to create a maze-chase game similar to *Pac-Man*, we could define the maze using tiles from a tile sheet. The sequence of tiles for our game maze would be considered a tile map.

The first tile is a gray square, which we can use for the "road" tiles between the wall tiles. Any tile that a game sprite can move on is referred to as *walkable*. Even though our tanks are not literally walking but driving, the concept is the same. In Chapter 9 we will create a small game using these concepts, but for now, let's concentrate on defining a tile map and displaying it on the canvas.

Our tile map will be a two-dimensional array of tile id numbers. If you recall, the tile id numbers for our tile sheet are in a single dimension, numbering from 0 to 31. Let's say we are going to create a very small game screen consisting of 10 tiles in length and 10 tiles in height. This means we need to define a tile map of 100 individual tiles (10×10). If our tiles are 32 pixels by 32 pixels, we will define a 320×320 game screen.

There are many ways to define a tile map. One simple way is to use a tile map editor program to lay out a grid of tiles, and then export the data to re-create the tile map in JavaScript. This is precisely how we are going to create our tile map.

Creating a Tile Map with Tiled

The program we are going to use, Tiled, is a great tile map editor that is available for Mac OS, Windows, and Linux. Of course, tile maps can be designed by hand, but map creation is much easier if we utilize a program such as Tiled to do some of the legwork for us. Tiled is available for free under the GNU free software license from *http://www.mapeditor.org/*.

As stated before, you do not need to use this software. Tile maps can be created with other good (and free) software such as Mappy (*http://tilemap.co.uk/mappy.php*) and Tile Studio (*http://tilestudio.sourceforge.net/*), and even by hand using MS Paint.

The goal of creating a tile map is to visually lay out a grid of tiles that represents the game screen, and then export the tile ids that represent those tiles. We will use the exported data as a two-dimensional array in our code to build the tile map on the canvas.

Here are the basic steps for creating a simple tile map in Tiled for use in the following section:

1. Create a new tile map from the File menu. When it asks for Orientation, select Orthogonal with a Map Size of 10×10 and a Tile Size of 32×32.
2. From the Map menu, import the *tanks_sheet.png* to be used as the tile set. Select "New tileset" from this menu, and give it any name you want. Browse to find the *tanks_sheet.png* that you downloaded from this book's website. Make sure that Tile Width and Tile Height are both 32; keep the Margin and Spacing both at 0.
3. Select a tile from the tile set on the bottom-right side of the screen. Once selected, you can click and "paint" the tile by selecting a location on the tile map on the top-left side of the screen. Figure 4-9 shows the tile map created for this example.
4. Save the tile map. Tiled uses a plain text file format called *.tmx*. Normally, tile data in Tiled is saved out in a base-64-binary file format; however, we can change this by editing the preferences for Tiled. On a Mac, under the Tiled menu, there should be a Preferences section. (If you are using the software on Windows or Linux, you will find this in the File menu.) When setting the preferences, select CSV in the "Store tile layer data as" drop-down menu. Once you have done this, you can save the file from the File menu.

Here is a look at what the saved *.tmx* file will look like in a text editor:

```
<?xml version="1.0" encoding="UTF-8"?>
<map version="1.0" orientation="orthogonal" width="10" height="10"
        tilewidth="32" tileheight="32">
  <tileset firstgid="1" name="tanks" tilewidth="32" tileheight="32">
  <image source="tanks_sheet.png"/>
  </tileset>
  <layer name="Tile Layer 1" width="10" height="10">
  <data encoding="csv">
32,31,31,31,1,31,31,31,31,32,
1,1,1,1,1,1,1,1,1,1,
32,1,26,1,26,1,26,1,1,32,
32,26,1,1,26,1,1,26,1,32,
32,1,1,1,26,26,1,26,1,32,
32,1,1,26,1,1,1,26,1,32,
32,1,1,1,1,1,1,26,1,32,
1,1,26,1,26,1,26,1,1,1,
32,1,1,1,1,1,1,1,1,32,
32,31,31,31,1,31,31,31,31,32
</data>
</layer>
</map>
```

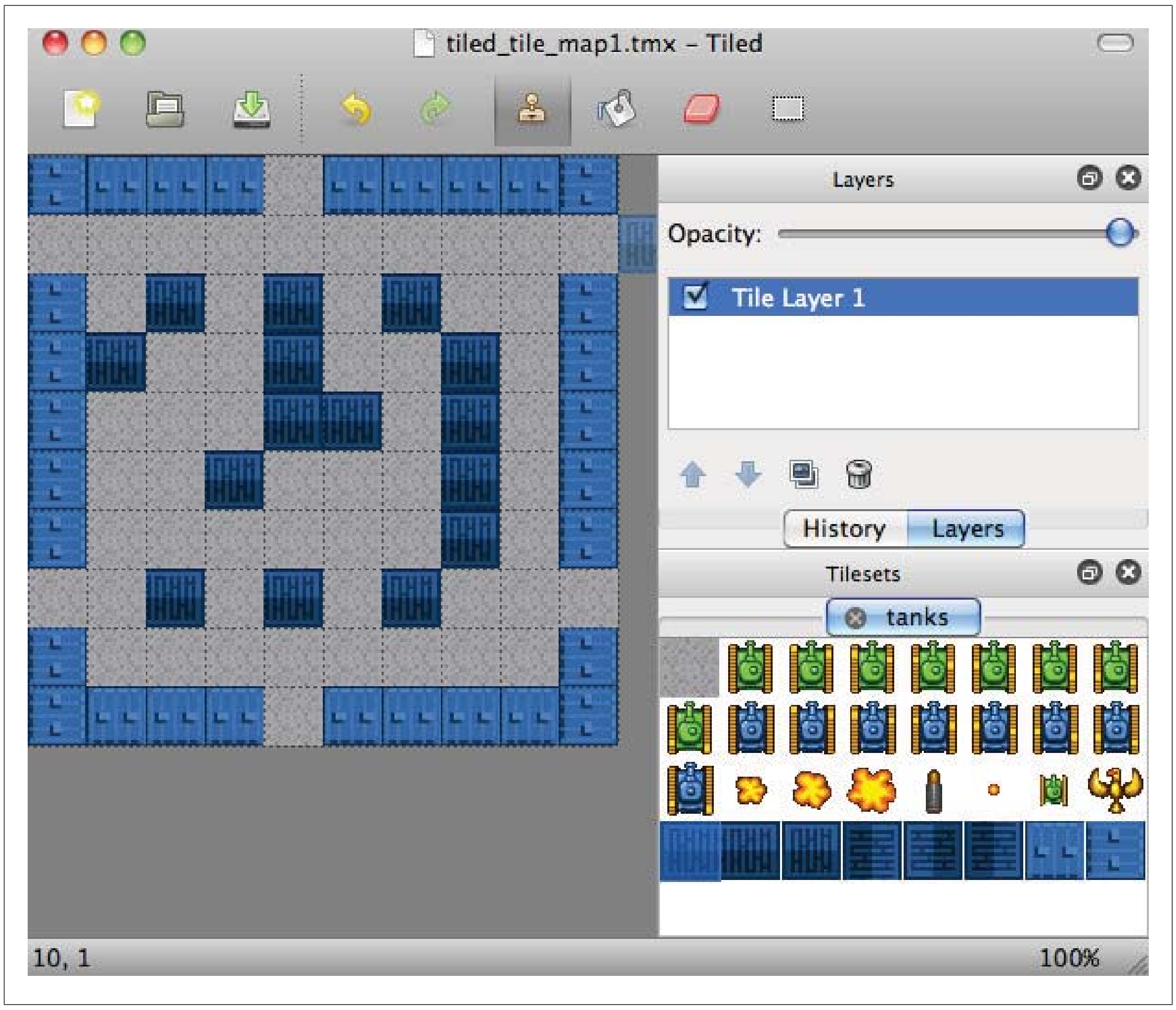

Figure 4-9. The tile map example in Tiled

The data is an XML data set used to load and save tile maps. Because of the open nature of this format and the simple sets of row data for the tile map, we can use this data easily in JavaScript. For now, we are only concerned with the 10 rows of comma-delimited numbers inside the `<data>` node of the XML—we can take those rows of data and create a very simple two-dimensional array to use in our code.

Displaying the Map on the Canvas

The first thing to note about the data from Tiled is that it is *1 relative*, not 0 relative. This means that the tiles are numbered from 1–32 instead of 0–31. We can compensate for this by subtracting one from each value as we transcribe it to our array, or programmatically during our tile sheet drawing operation. We will do it programmatically by creating an offset variable to be used during the draw operation:

```
var mapIndexOffset = -1;
```

Rather than using the `mapIndexOffset` variable, we could loop through the array of data and subtract 1 from each value. This would be done before the game begins, saving the extra processor overload from performing this math operation on each tile when it is displayed.

Map height and width

We also are going to create two variables to give flexibility to our tile map display code. These might seem simple and unnecessary now, but if you get in the habit of using variables for the height and width of the tile map, it will be much easier to change its size in the future.

We will keep track of the width and height based on the number of rows in the map and the number of columns in each row:

```
var mapRows = 10;
var mapCols = 10;
```

Storing the map data

The data that was output from Tiled was a series of rows of numbers starting in the top left and moving left to right, then down when the rightmost column in a row was completed. We can use this data almost exactly as output by placing it in a two-dimensional array:

```
var tileMap = [
      [32,31,31,31,1,31,31,31,31,32]
   ,  [1,1,1,1,1,1,1,1,1,1]
   ,  [32,1,26,1,26,1,26,1,1,32]
   ,  [32,26,1,1,26,1,1,26,1,32]
   ,  [32,1,1,1,26,26,1,26,1,32]
   ,  [32,1,1,26,1,1,1,26,1,32]
   ,  [32,1,1,1,1,1,1,26,1,32]
   ,  [1,1,26,1,26,1,26,1,1,1]
   ,  [32,1,1,1,1,1,1,1,1,32]
   ,  [32,31,31,31,1,31,31,31,31,32]

   ];
```

Displaying the map on the canvas

When we display the tile map, we simply loop through the rows in the `tileMap` array, and then loop through the columns in each row. The `tileID` number at `[row][column]` will be the tile to copy from the tile sheet to the canvas. `row *32` will be the `y` location to place the tile on the canvas; `col*32` will be the `x` location to place the tile:

The row, column referencing might seem slightly confusing because row is the *y* direction and column is the *x* direction. We do this because our tiles are organized into a two-dimensional array. The row is always the first subscript when accessing a 2D array.

```
for (var rowCtr=0;rowCtr<mapRows;rowCtr++) {
   for (var colCtr=0;colCtr<mapCols;colCtr++){

      var tileId = tileMap[rowCtr][colCtr]+mapIndexOffset;
      var sourceX = Math.floor(tileId % 8) *32;
      var sourceY = Math.floor(tileId / 8) *32;

      context.drawImage(tileSheet, sourceX,
       sourceY,32,32,colCtr*32,rowCtr*32,32,32);
   }

}
```

We use the `mapRows` and the `mapCols` variables to loop through the data and to paint it to the canvas. This makes it relatively simple to modify the height and width of the tile map without having to find the hardcoded values in the code. We could have also done this with other values such as the tile width and height, as well as the number of tiles per row in the tile sheet (8).

The `sourceX` and `sourceY` values for the tile to copy are found in the same way as in the previous examples. This time, though, we find the `tileId` using the `[rowCtr][colCtr]` two-dimensional lookup, and then adding the `mapIndexOffset`. The offset is a negative number (`-1`), so this effectively subtracts 1 from each tile map value, resulting in 0-relative map values that are easier to work with. Example 4-10 shows this concept in action, and Figure 4-10 illustrates the results.

Example 4-10. Rotation, animation, and movement

```
var tileSheet = new Image();
tileSheet.addEventListener('load', eventSheetLoaded , false);

tileSheet.src = "tanks_sheet.png";

var mapIndexOffset = -1;
var mapRows = 10;
var mapCols = 10;

var tileMap = [
       [32,31,31,31,1,31,31,31,31,32]
   ,   [1,1,1,1,1,1,1,1,1,1]
   ,   [32,1,26,1,26,1,26,1,1,32]
   ,   [32,26,1,1,26,1,1,26,1,32]
   ,   [32,1,1,1,26,26,1,26,1,32]
   ,   [32,1,1,26,1,1,1,26,1,32]
   ,   [32,1,1,1,1,1,1,26,1,32]
   ,   [1,1,26,1,26,1,26,1,1,1]
   ,   [32,1,1,1,1,1,1,1,1,32]
   ,   [32,31,31,31,1,31,31,31,31,32]

   ];

function eventSheetLoaded() {
   drawScreen()
}

function drawScreen() {
   for (var rowCtr=0;rowCtr<mapRows;rowCtr++) {
      for (var colCtr=0;colCtr<mapCols;colCtr++){

         var tileId = tileMap[rowCtr][colCtr]+mapIndexOffset;
         var sourceX = Math.floor(tileId % 8) *32;
         var sourceY = Math.floor(tileId / 8) *32;

         context.drawImage(tileSheet, sourceX,
         sourceY,32,32,colCtr*32,rowCtr*32,32,32);
      }

   }
 }
```

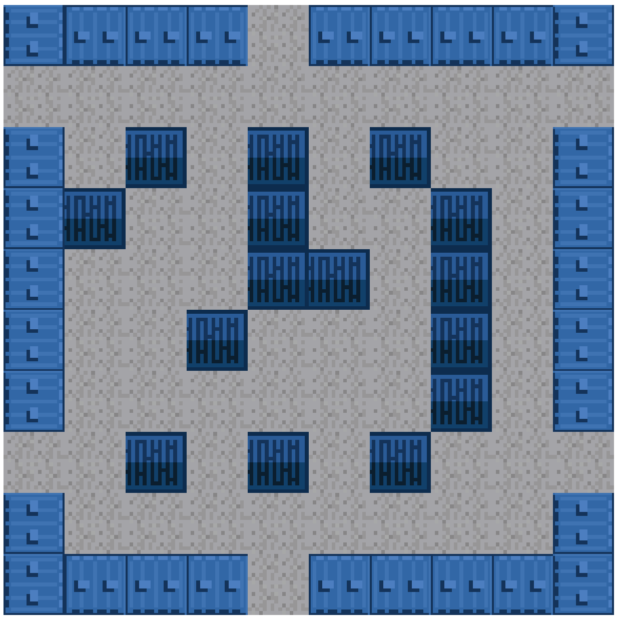

Figure 4-10. The tile map painted on the canvas

Next, we are going to leave the world of tile-based Canvas development (see Chapter 9 for an example of a small game developed with these principles). The final section of this chapter discusses building our own simple tile map editor. But before we get there, let's look at panning around and zooming in and out of an image.