Tutorial – Sound Effects and Side-Scroling

The lecture slides for this week cover to challenging topics – sound and side-scrolling.

The first topic, sound, is made significantly easier through the use of the third-party library Howler. Side-scrolling, being a little more time-consuming to get working, will be covered in this tutorial first.

Scrolling the Level:

We want the player to be in the centre of the screen as much as possible, so when we move the player the map will scroll accordingly.

This is a little tricky, so the math has been done for you – although it's all just simple multiplication and addition. However the map for this tutorial is only scrolling along the x-axis. If you want your game to scroll along the y-axis you'll need to modify the code accordingly.

This tutorial assumes that your Player character can move left or right when the arrow keys on the keyboard are pressed. If you have not yet implemented the player character movement you will need to review the earlier tutorials to complete that. Make sure your player can move left and right before continuing.

1. We now need to edit the drawMap() function in main.js. I'll step through what needs to happen, and then we'll take a look at the code. (The following changes will be placed at the top of the drawMap() function in main.js).

    a. First, we need to calculate how many tiles can fit on the screen. Then add 2 to this number (for the overhang on the left and right). Because as we scroll left or right we need to draw part of the next tile in that direction.

```
var maxTiles = Math.floor(SCREEN_WIDTH / TILE) + 2;
```

    b. Calculate the tile the player is currently on. Here we use a utility function (pixelToTile) that takes a pixel coordinate and converts it to a tile coordinate.

```
var tileX = pixelToTile(player.position.x);
```

    c. Calculate the offset of the player from the origin of the tile it's on. I've add the width of the tile to this, just to make the numbers work.

```
var offsetX = TILE + Math.floor(player.position.x%TILE);
```

d. Now we can calculate the starting tile on the x-axis to draw from. This will be the x-coordinate (in tiles) for the left-most column of tiles. I'm dividing by 2 here because the player is in the centre of the screen.
If the player is too close to the start of the level, then the map will stop scrolling left and the player will move instead. Likewise for when the player gets to the right-most edge of the level.
At all other times the map will scroll and the player will stay in the centre of the screen.

```
startX = tileX - Math.floor(maxTiles / 2);

if(startX < -1)
{
	startX = 0;
	offsetX = 0;
}
if(startX > MAP.tw - maxTiles)
{
	startX = MAP.tw - maxTiles + 1;
	offsetX = TILE;
}
```

e. From here we calculate a world x-axis offset. This is the amount (in pixels) that the world has been scrolled. We'll use this value later when drawing the player.

```
worldOffsetX = startX * TILE + offsetX;
```

f. Finally we modify our loops. We're still drawing all the layers, so the first *for* statement stays the same.
We're not scrolling on the y, so the second *for* statement also stays the same.
But for the third *for* statement – which loops through the x-axis – we need to start from our startX variable (which tells us which column of tiles is now on the left-most side of the screen) and keep going for maxTiles iterations (so the right-most column is at index startX+maxTiles).
Because we're modifying the x index, we can no longer just increment the value of idx (which gives the index of the current tile to draw). So this needs to be calculated before we enter the x-axis loop.
The rest stays the same, except for the call to drawImage, where we must now pass updated screen coordinates based on what tile we're currently drawing.
The three for loops look like this:

```
for( var layerIdx=0; layerIdx < LAYER_COUNT; layerIdx++ )
{
   for( var y = 0; y < level1.layers[layerIdx].height;  y++ )
   {
        var idx = y * level1.layers[layerIdx].width + startX;
        for( var x = startX; x < startX + maxTiles;  x++ )
        {
           if( level1.layers[layerIdx].data[idx] != 0 )
           {
              // the tiles in the Tiled map are base 1 (meaning a value of 0 means no tile),
              // so subtract one from the tileset id to get the
              // correct tile
              var tileIndex = level1.layers[layerIdx].data[idx] - 1;
              var sx = TILESET_PADDING + (tileIndex % TILESET_COUNT_X) *
                       (TILESET_TILE + TILESET_SPACING);
              var sy = TILESET_PADDING + (Math.floor(tileIndex / TILESET_COUNT_Y)) *
                        (TILESET_TILE + TILESET_SPACING);
              context.drawImage(tileset, sx, sy, TILESET_TILE, TILESET_TILE,
                       (x-startX)*TILE - offsetX, (y-1)*TILE, TILESET_TILE, TILESET_TILE);
           }
           idx++;
        }
    }
 }
```

That's quite a bit to take in, but hopefully it seems fairly straight forward. The complete code for the updated drawMap() function in main.js is as follows:

```
var worldOffsetX =0;
function drawMap()
{
  var startX = -1;
  var maxTiles = Math.floor(SCREEN_WIDTH / TILE) + 2;
  var tileX = pixelToTile(player.position.x);
  var offsetX = TILE + Math.floor(player.position.x%TILE);

  startX = tileX - Math.floor(maxTiles / 2);

  if(startX < -1)
  {
    startX = 0;
    offsetX = 0;
  }
  if(startX > MAP.tw - maxTiles)
  {
    startX = MAP.tw - maxTiles + 1;
    offsetX = TILE;
  }

  worldOffsetX = startX * TILE + offsetX;

  for( var layerIdx=0; layerIdx < LAYER_COUNT; layerIdx++ )
  {
    for( var y = 0; y < level1.layers[layerIdx].height;  y++ )
    {
      var idx = y * level1.layers[layerIdx].width + startX;

      for( var x = startX; x < startX + maxTiles;  x++ )
      {
        if( level1.layers[layerIdx].data[idx] != 0 )
        {
          // the tiles in the Tiled map are base 1 (meaning a value of 0 means no tile),
          // so subtract one from the tileset id to get the correct tile
          var tileIndex = level1.layers[layerIdx].data[idx] - 1;
          var sx = TILESET_PADDING + (tileIndex % TILESET_COUNT_X) *
                      (TILESET_TILE + TILESET_SPACING);
          var sy = TILESET_PADDING + (Math.floor(tileIndex / TILESET_COUNT_Y)) *
                      (TILESET_TILE + TILESET_SPACING);
          context.drawImage(tileset, sx, sy, TILESET_TILE, TILESET_TILE,
                      (x-startX)*TILE - offsetX, (y-1)*TILE, TILESET_TILE, TILESET_TILE);
        }
        idx++;
      }
    }
  }
}
```

2. You should already have the two utility functions tileToPixel and pixelToTile. In case you don't, add these to main.js

```
function tileToPixel(tile)
{
	return tile * TILE;
};
function pixelToTile(pixel)
{
	return Math.floor(pixel/TILE);
};
```

Do you already have this code in your game?

Only add it if you don't.

3. Now we can return to the player script and update the player's drawing based on the worldOffsetX variable.

```
Player.prototype.draw = function()
{
	context.drawImage(this.image,
				this.position.x - worldOffsetX, this.position.y);
}
```

```
Player.prototype.draw = function()
{
	this.sprite.draw(context,
		this.position.x - worldOffsetX,
		this.position.y);
}
```

Is your Player using a sprite? (It should be)

If it is, you'll need this code instead

If you run the game now, the map should scroll as you move the player left and right. However you may notice that when you move the player it sort of jitters a little.

This is because we haven't separated our updating from our drawing.

If you look at the run() function you'll notice that first we draw the map, then we update the player, then we draw the player.

The problem with doing this is that the map is calculating the world offset based on the players position, then when the player's update function runs the player's position changes (so now the world offset is out of date.) Its' not until the next frame that the world draws using the correct position of the player, but after the map draws the player's position will go out of date again.

This results in the player jittering and old, out-of-date values are used for drawing.

The solution is to do all the updating first, then do all the drawing.

4. Update the run() function in main.js as follows:

```
function run()
{
	context.fillStyle = "#ccc";
	context.fillRect(0, 0, canvas.width, canvas.height);

	var deltaTime = getDeltaTime();

	player.update(deltaTime);  // update the player before drawing the map

	drawMap();
	player.draw();

	. . .
}
```

Congratulations! If everything went as expected you should now be able to move your player around your level, and the level will scroll on the x-axis according to the position of your player.

Adding Sound:

1. Download the howler.js script from AIE Portal, along with the background music and sound effect files if you don't have your own sounds to use.
2. Add a reference to the howler.js script to the html file.
3. In main.js, add variables for your music and sfx, and in the initialize() function create the howler objects:

```
var musicBackground;
var sfxFire;

function initialize() {
	. . .

	musicBackground = new Howl(
	{
		urls: ["background.ogg"],
		loop: true,
		buffer: true,
		volume: 0.5
	} );
	musicBackground.play();

	sfxFire = new Howl(
		{
			urls: ["fireEffect.ogg"],
			buffer: true,
			volume: 1,
			onend: function() {
				isSfxPlaying = false;
			}
		} );
}
```

The call to musicBackground.play() will set the background music to start playing as soon as it is loaded.

4. Inside player.js, add the following code for shooting in the player's update() function:

```
	if(keyboard.isKeyDown(keyboard.KEY_UP) == true) {
		jump = true;
	}

	if(this.cooldownTimer > 0)
	{
		this.cooldownTimer -= deltaTime;
	}
	if(keyboard.isKeyDown(keyboard.KEY_SPACE) == true && this.cooldownTimer <= 0) {
		sfxFire.play();
		this.cooldownTimer = 0.3;

		// Shoot a bullet
	}
```

Notice that I've changed the key for jumping from *space* to *up*. This is so that I can use the *space* key for shooting.

5. The last thing to do is to add definition for the cooldown timer variable to the player's constructor. The cooldown variable ensures that we only fire 1 bullet every 0.3 seconds (without it we'd fire a bullet each frame when the space bar is held).

```
var Player = function() {
        . . .

        this.cooldownTimer = 0;
};
```

Exercises:

1. Find some background music and some sound effects and add these to your game. Don't forget that somewhere in your assignment's documentation you must credit the original source for any sounds or images that aren't your own work.

2. See if you can complete the shooting code by creating a bullet when the space bar is pressed.

   You should have a bullet array (a good place to put it is in your main.js file), and the bullet should have its own definition (bullet.js). When you create a bullet you'll set its velocity (left or right) and the bullet will keep updating until it hits an enemy or flies off the screen.

   Attempt this yourself, but if you can't get it don't worry, its briefly covered in next week's lesson.