Tutorial – Collision Detection for Platformers

For this tutorial we're going to be going through the code presented in the lecture slides and adding it to the code we made from last week.

You may find it useful to have the lecture slides open so that you can take note of any explanations in the slides.

Creating Collision Maps:

As explained in the lecture, we need a way to simplify our data. Checking our character's collision rectangle against every tile in our map is going to be slow. So we'll break our map into a 2D grid of cells. Each cell will either hold the value 0 (no collision) or 1(collision).

This will allow us a very simple method to determine whether or not the player is colliding with a tile. All we need to do is get convert the pixel coordinates of the player to tile coordinates and then look in our collision map to see if the tiles adjacent to the player's tile have collisions (a value of 1).

This will be the basic approach for collision detection between the player (and enemies) and our map, and will be much faster because we only need to test the player against a few tiles every frame.

1. Add the initialize() function shown in the lecture slides to your main.js file:

```
var cells = [];          // the array that holds our simplified collision data
function initialize() {
    for(var layerIdx = 0; layerIdx < LAYER_COUNT; layerIdx++) {    // initialize the collision map
        cells[layerIdx] = [];
        var idx = 0;
        for(var y = 0; y < level1.layers[layerIdx].height; y++) {
            cells[layerIdx][y] = [];
            for(var x = 0; x < level1.layers[layerIdx].width; x++) {
                if(level1.layers[layerIdx].data[idx] != 0) {
                        // for each tile we find in the layer data, we need to create 4 collisions
                        // (because our collision squares are 35x35 but the tile in the
                        // level are 70x70)
                    cells[layerIdx][y][x] = 1;
                    cells[layerIdx][y-1][x] = 1;
                    cells[layerIdx][y-1][x+1] = 1;
                    cells[layerIdx][y][x+1] = 1;
                }
                else if(cells[layerIdx][y][x] != 1) {
                        // if we haven't set this cell's value, then set it to 0 now
                    cells[layerIdx][y][x] = 0;
                }
                idx++;
            }
        }
    }
}
```

2. Add the call to the initialize() function right AFTER the run() function. Be very sure NOT to put it inside the function itself, otherwise the initialization will happen every frame – which would be disastrous.

```
function run()
{
   . . .
}

initialize();

//-------------------- Don't modify anything below here

. . .
```

   The reason we place the call to initialize() here is so that it will run before we enter the game loop.

3. In order to use the newly created collision maps, we need a few utility functions.

   These will be used by the player object when testing for collisions with the platforms.

   We need the cellAtPixelCoord(layer, x, y) which will tell us if a tile with a collision exists at the given pixel coordinates on the given layer.
   We also have a similar function cellAtTileCoord(layer, tx, ty), which uses tile coordinates instead.

   The pixelToTile and tileToPixel functions convert between pixel and tile coordinates. These functions will also be useful in a few weeks when we add side-scrolling to our game.

   And we have one more convenience function, bound(value, min, max), which takes a value and will return the same value if it is between the input minimum and maximum, otherwise it returns the minimum or maximum. This will be helpful to make sure our player doesn't go over our maximum speed.

Insert this code before the drawMap() function:

```
function cellAtPixelCoord(layer, x,y)
{
	if(x<0 || x>SCREEN_WIDTH || y<0)
		return 1;
	// let the player drop of the bottom of the screen (this means death)
	if(y>SCREEN_HEIGHT)
		return 0;
	return cellAtTileCoord(layer, p2t(x), p2t(y));
};

function cellAtTileCoord(layer, tx, ty)
{
	if(tx<0 || tx>=MAP.tw || ty<0)
		return 1;
	// let the player drop of the bottom of the screen (this means death)
	if(ty>=MAP.th)
		return 0;
	return cells[layer][ty][tx];
};

function tileToPixel(tile)
{
	return tile * TILE;
};

function pixelToTile(pixel)
{
	return Math.floor(pixel/TILE);
};

function bound(value, min, max)
{
	if(value < min)
		return min;
	if(value > max)
		return max;
	return value;
}
```

4. Lastly, we need some constant variables that map to the layers in our level.

```
var LAYER_COUNT = 3;
var LAYER_BACKGOUND = 0;
var LAYER_PLATFORMS = 1;
var LAYER_LADDERS = 2;
```

I have three layers in my map, but you could have any number. Just make sure that the LAYER_PLATFORMS variable corresponds to the layer in your map that has the platforms. Otherwise your player won't collide with the platforms.

Player Collisions:

The collision maps will now get created when the game starts. All the data comes straight from the level maps, and there's nothing more that's needed from us.

The following code relates to changes that are necessary in order for the player to use the collision maps and start colliding with the platforms in our level.

1. While you're still in the main.js file, lets add all the variables listed on the slide 'Applying Forces' (sorry to jump around like this)

```
      // abitrary choice for 1m
var METER = TILE;
      // very exaggerated gravity (6x)
var GRAVITY = METER * 9.8 * 6;
      // max horizontal speed (10 tiles per second)
var MAXDX = METER * 10;
      // max vertical speed   (15 tiles per second)
var MAXDY = METER * 15;
      // horizontal acceleration -  take 1/2 second to reach maxdx
var ACCEL = MAXDX * 2;
      // horizontal friction -  take 1/6 second to stop from maxdx
var FRICTION = MAXDX * 6;
      // (a large) instantaneous jump impulse
var JUMP = METER * 1500;
```

2. Since we're using these new variables to control the speed of the player, you can remove the PLAYER_SPEED variable from the player.js file as it's no longer needed.

3. Now go back a slide, where the player's update function is first shown in the slides. Let's make those changes to our code.

   From here on out the tutorials are going to assume you've made the Vector2.js file from the exercises for lesson 6 (Splitting a Project). If you haven't done that, either do that now or modify the following code accordingly (although I strongly recommend you make the vector2 object).

   a. Add a velocity variable to the definition of the player.

      While you're at it, let's change the x and y variables to a single vector called position. We can also add a width and a height (these will define the collision box of the player for bullet/enemy collisions and might not necessarily be the same dimensions as the image), an offset vector (because the origin of our Chuck Norris sprite is at his feet, not in the top left corner), and the jumping and falling Boolean variables.

```
var Player = function() {
	this.image = document.createElement("img");
	this.position = new Vector2();
	this.position.set( 9*TILE, 0*TILE );

	this.width = 159;
	this.height = 163;

	this.offset = new Vector2();
	this.offset.set(-55,-87);

	this.velocity = new Vector2();

	this.falling = true;
	this.jumping = false;

	this.image.src = "hero.png";
};
```

b. Replace the existing update function with the one from the slides:

```
Player.prototype.update = function(deltaTime)
{
	// we'll insert code here later


	// collision detection
	// Our collision detection logic is greatly simplified by the fact that the
	// player is a rectangle and is exactly the same size as a single tile.
	// So we know that the player can only ever occupy 1, 2 or 4 cells.

	// This means we can short-circuit and avoid building a general purpose
	// collision detection engine by simply looking at the 1 to 4 cells that
	// the player occupies:
	var tx = pixelToTile(this.position.x);
	var ty = pixelToTile(this.position.y);
	var nx = (this.position.x)%TILE;		// true if player overlaps right
	var ny = (this.position.y)%TILE;		// true if player overlaps below
	var cell = cellAtTileCoord(LAYER_PLATFORMS, tx, ty);
	var cellright = cellAtTileCoord(LAYER_PLATFORMS, tx + 1, ty);
	var celldown = cellAtTileCoord(LAYER_PLATFORMS, tx, ty + 1);
	var celldiag = cellAtTileCoord(LAYER_PLATFORMS, tx + 1, ty + 1);
}
```

4. We can now insert the code to apply forces (gravity, acceleration and friction).

Jump to the code on the slide titled ‘Applying Forces’. We need to insert this code above the first line in the update function, but there are some important pieces missing.

You’ll notice the left, right and jump variables being read in the *if* statements, but they aren’t created anywhere. These Boolean variables are set when we have keypress events. So we’ll need to add the keyboard handing code too.

Insert the following code so it is at the top of the player’s update function:

```
Player.prototype.update = function(deltaTime)
{
    var left = false;
    var right = false;
    var jump = false;

    // check keypress events
    if(keyboard.isKeyDown(keyboard.KEY_LEFT) == true) {
        left = true;
    }
    if(keyboard.isKeyDown(keyboard.KEY_RIGHT) == true) {
        right = true;
    }
    if(keyboard.isKeyDown(keyboard.KEY_SPACE) == true) {
        jump = true;
    }

    var wasleft = this.velocity.x < 0;
    var wasright = this.velocity.x > 0;
    var falling = this.falling;
    var ddx = 0;                      // acceleration
    var ddy = GRAVITY;

    if (left)
        ddx = ddx - ACCEL;           // player wants to go left
    else if (wasleft)
        ddx = ddx + FRICTION;        // player was going left, but not any more

    if (right)
        ddx = ddx + ACCEL;           // player wants to go right
    else if (wasright)
        ddx = ddx - FRICTION;        // player was going right, but not any more

    if (jump && !this.jumping && !falling)
    {
        ddy = ddy - JUMP;            // apply an instantaneous (large) vertical impulse
        this.jumping = true;
    }

    // calculate the new position and velocity:
    this.position.y = Math.floor(this.position.y  + (deltaTime * this.velocity.y));
    this.position.x = Math.floor(this.position.x  + (deltaTime * this.velocity.x));
    this.velocity.x = bound(this.velocity.x + (deltaTime * ddx), -MAXDX, MAXDX);
    this.velocity.y = bound(this.velocity.y + (deltaTime * ddy), -MAXDY, MAXDY);
```

5. We also need to insert the clamping code as mentioned in the slide titled 'Getting' Jiggy wit' it'

   This goes **after** the calculations for the new position and velocity, but **before** the code for the collision detection:

```
Player.prototype.update = function(deltaTime)
{
    . . .
    // calculate the new position and velocity:
    this.position.y = Math.floor(this.position.y  + (deltaTime * this.velocity.y));
    this.position.x = Math.floor(this.position.x  + (deltaTime * this.velocity.x));
    this.velocity.x = bound(this.velocity.x + (deltaTime * ddx), -MAXDX, MAXDX);
    this.velocity.y = bound(this.velocity.y + (deltaTime * ddy), -MAXDY, MAXDY);

    if ((wasleft  && (this.velocity.x > 0)) ||
        (wasright && (this.velocity.x < 0)))
    {
        // clamp at zero to prevent friction from making us jiggle side to side
        this.velocity.x = 0;
    }


    // collision detection
    . . .
}
```

6. We can now insert the last two code segments from the slides – the code for testing vertically and testing horizontally.

   This code will read the values in cell, celldown, cellright and celldiag, and let us know which cells have platforms (or other collisions) in them.

   If the player is falling and the cell immediately below is a platform, then stop falling. If we're moving right and the cell immediately right of us is a platform, then stop moving. And so on.

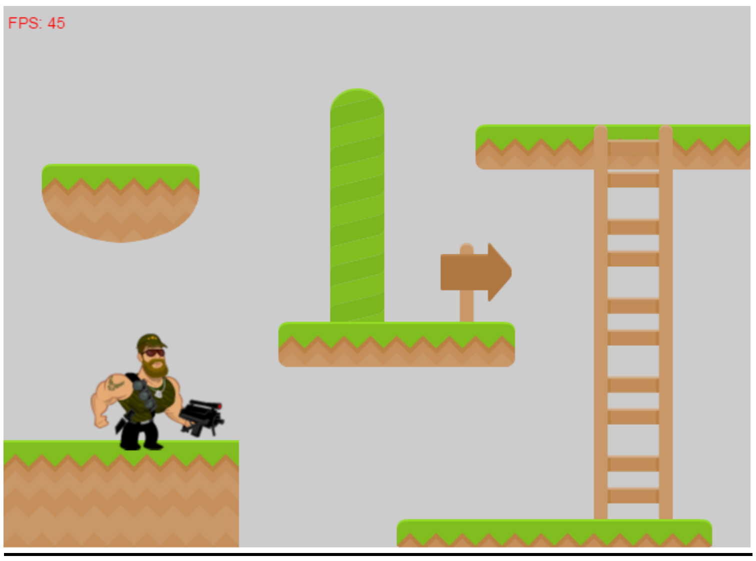
We insert the following code at the end of the update function:

```
Player.prototype.update = function(deltaTime)
{
    . . .
    // calculate the new position and velocity:
    this.position.y = Math.floor(this.position.y  + (deltaTime * this.velocity.y));
    this.position.x = Math.floor(this.position.x  + (deltaTime * this.velocity.x));
    this.velocity.x = bound(this.velocity.x + (deltaTime * ddx), -MAXDX, MAXDX);
    this.velocity.y = bound(this.velocity.y + (deltaTime * ddy), -MAXDY, MAXDY);

    if ((wasleft  && (this.velocity.x > 0)) ||
        (wasright && (this.velocity.x < 0)))
    {
        // clamp at zero to prevent friction from making us jiggle side to side
        this.velocity.x = 0;
    }

    // collision detection
    // Our collision detection logic is greatly simplified by the fact that the
    // player is a rectangle and is exactly the same size as a single tile.
    // So we know that the player can only ever occupy 1, 2 or 4 cells.

    // This means we can short-circuit and avoid building a general purpose
    // collision detection
    // engine by simply looking at the 1 to 4 cells that the player occupies:
    var tx = pixelToTile(this.position.x);
    var ty = pixelToTile(this.position.y);
    var nx = (this.position.x)%TILE;         // true if player overlaps right
    var ny = (this.position.y)%TILE;         // true if player overlaps below
    var cell = cellAtTileCoord(LAYER_PLATFORMS, tx, ty);
    var cellright = cellAtTileCoord(LAYER_PLATFORMS, tx + 1, ty);
    var celldown = cellAtTileCoord(LAYER_PLATFORMS, tx, ty + 1);
    var celldiag = cellAtTileCoord(LAYER_PLATFORMS, tx + 1, ty + 1);

    // If the player has vertical velocity, then check to see if they have hit a platform
    // below or above, in which case, stop their vertical velocity, and clamp their
    // y position:
    if (this.velocity.y > 0)  {
       if ((celldown && !cell) || (celldiag && !cellright && nx))  {
          // clamp the y position to avoid falling into platform below
          this.position.y = tileToPixel(ty);
          this.velocity.y = 0;                     // stop downward velocity
          this.falling = false;                    // no longer falling
          this.jumping = false;                    // (or jumping)
          ny = 0;                                  // no longer overlaps the cells below
       }
    }
```

```
        else if (this.velocity.y < 0)  {
          if ((cell && !celldown) || (cellright && !celldiag && nx))  {
              // clamp the y position to avoid jumping into platform above
              this.position.y = tileToPixel(ty + 1);
              this.velocity.y = 0;            // stop upward velocity
              // player is no longer really in that cell, we clamped them to the cell below
              cell = celldown;
              cellright = celldiag;           // (ditto)
              ny = 0;                         // player no longer overlaps the cells below
                 }
          }
          if (this.velocity.x > 0) {
               if ((cellright && !cell) || (celldiag  && !celldown && ny))  {
                    // clamp the x position to avoid moving into the platform we just hit
                    this.position.x = tileToPixel(tx);
                    this.velocity.x = 0;      // stop horizontal velocity
               }
          }
          else if (this.velocity.x < 0) {
              if ((cell && !cellright) || (celldown && !celldiag && ny))  {
                  // clamp the x position to avoid moving into the platform we just hit
                  this.position.x = tileToPixel(tx + 1);
                  this.velocity.x = 0;        // stop horizontal velocity
              }
          }
}
```

7. Don't forget, if you changed your player's x and y variables to be a position vector, you'll need to update the player's drawing code, and any other code that references the player's position.

Congratulations! If everything went as expected you be able to move your player around the platforms in your level.
Use the left and right arrow keys for movement, and the space bar to jump.



Exercises:

We're making good progress with our game. Before long we'll have our platformer working. But just because we've got some gameplay doesn't mean we've got a game.

This week you should add some different game states to your game. Start off with a splash screen, then have the game state, and when the player 'dies' switch to the game-over state.

For now, you could have a countdown timer or switch to the game-over state when the user presses a certain key.

If you want to add a main menu and a high-scores state, that would certainly make your game look very professional.

If you need a bit of revision with game state management, take a quick look back over lesson 5 (Managing Game States).