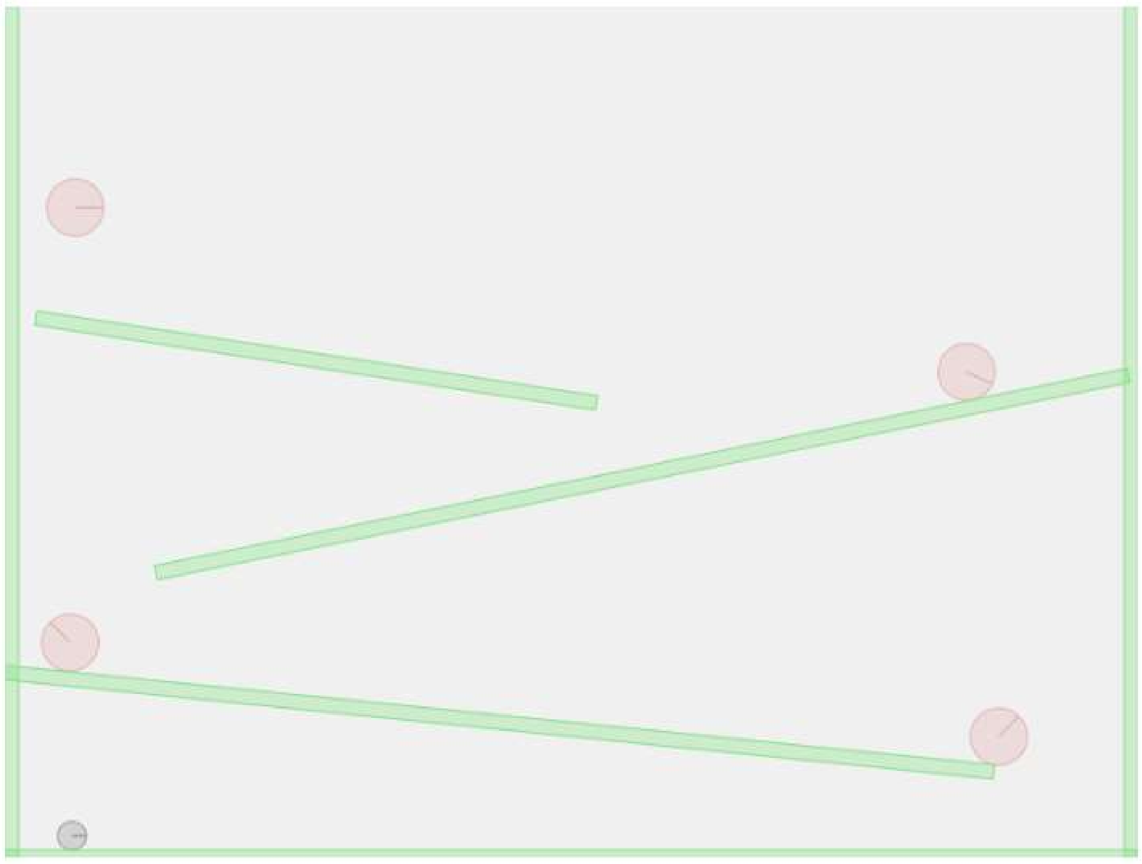
Web Programming Blog

by CALUM LINDSAY

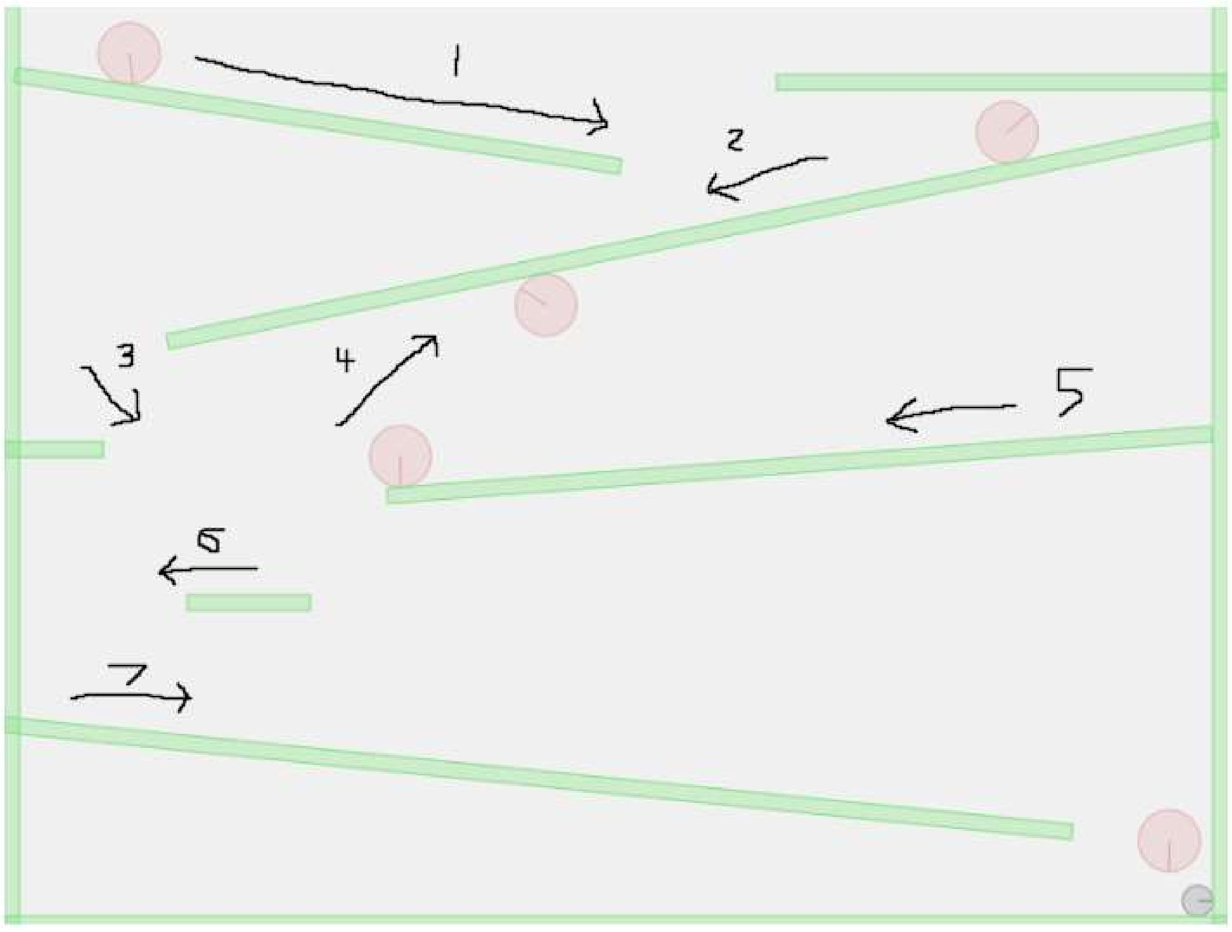
Donkey Kong

Posted by CALUM LINDSAY on 15 September 2024, 18:15
Last updated 06 December 2024, 15:37

For our first tutorial we were given a basic 2D "game" similar to classic Donkey Kong but only containing the basic physics of barrels rolling down slopes and a controllable character (the smallest circle) that can move and jump as pictured:



While adding more platforms to the game I had the inspiration to add a platform that "bounces" the barrels with the final layout and drawn in route looking as follows:



My first attempt was to add an impulse vector pointing right and up to the barrel within the beginContact function when a barrel first makes contact with the bounce platform. This did not work, in fact nothing happened to my surprise, after contemplating the behaviour of Box2D further and looking at the function names I realized Box2D was probably overwriting any changes I was making to the barrel in it's solve function and so I moved the functionality to the PostSolve function instead which resolved the issue.

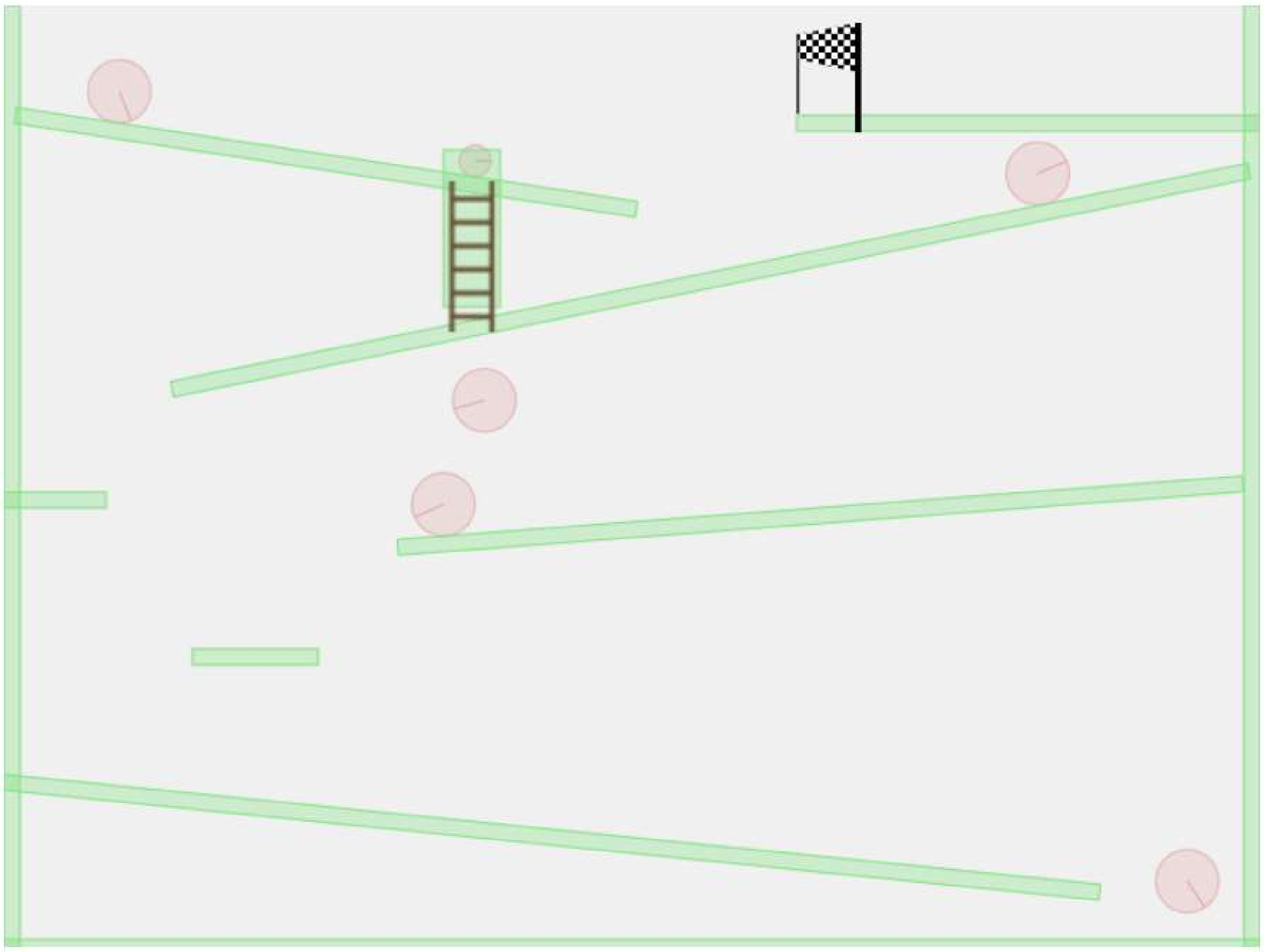
This is interesting as beginContact is presumably called by Box2D's solve function and this suggests that some sort of double buffering is taking place where we are modifying Box2D's next frame buffer and our changes do not take effect because Box2D is reading from the previous frame buffer and writing to the next frame buffer.

Additionally I decided to make the barrels only bounce on the bounce platform once rather than looking for a geometric solution to the resulting chaotic behaviour, which, would make more sense but be less fun. I simply added a flag to each barrels userData and checked this flag before applying the force, setting it when the force is applied.

Here is the final solution:

```
listener.PostSolve = function(contact, impulse) {
    var fixa=contact.GetFixtureA().GetBody().GetUserData().id;
    var fixb=contact.GetFixtureB().GetBody().GetUserData().id;

    //Handle bounce platform
    if(fixa == "barrel" && fixb == "bounceplat")
            bounceBarrel(contact.GetFixtureA().GetBody());
    else if(fixa == "bounceplat" && fixb == "barrel")
            bounceBarrel(contact.GetFixtureB().GetBody());
}

function bounceBarrel(barrelBody)
{
    var data = barrelBody.GetUserData();
    if(data.bounced != true)
    {
        data.bounced = true;
        barrelBody.ApplyImpulse(new b2Vec2(7,-14), barrelBody.GetWorldCenter());
        barrelBody.SetUserData(data);
    }
}
```

The final task I had time to complete was to add ladders:



This was predictably tricky compared with the previous tasks! My solution was:

- Set the categoryBits of the platform above the ladder to 2:

```
var filter = plat1.GetBody().GetFixtureList().GetFilterData();
filter.categoryBits = 2;
plat1.GetBody().GetFixtureList().SetFilterData(filter);
```

- Make the ladder a sensor with the SetSensor function:

```
var ladder = defineNewStatic(1.0,1.0,1.0,298,142,18,50,0,"ladder");
ladder.GetBody().GetFixtureList().SetSensor(true);
```

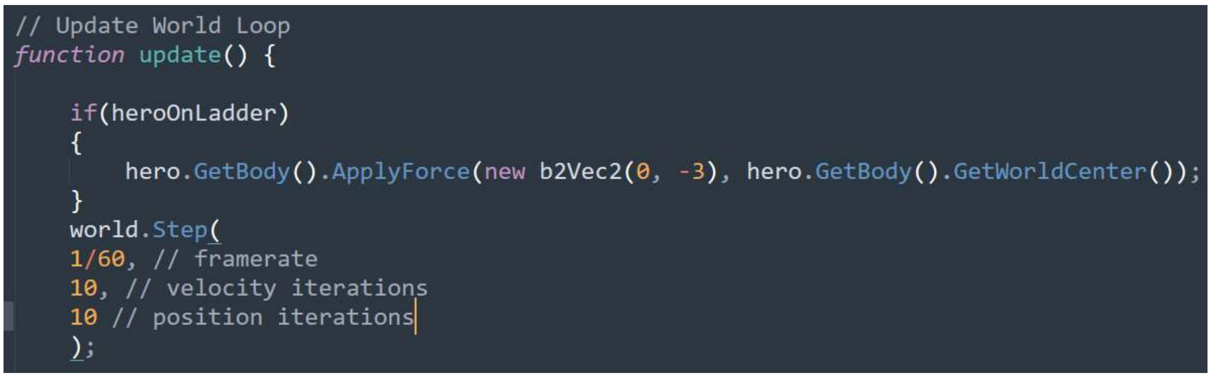
- Swapping the maskBits of the hero to 65533 (collide with every category but 2) when they begin contact with the ladder and back to 65535 (collide with every category) when they end contact:

```
//Ladder functionality
if((fixa == "hero" && fixb == "ladder") || (fixb == "hero" && fixa == "ladder"))
{
    heroOnLadder = true;
    var filter = hero.GetBody().GetFixtureList().GetFilterData();
    filter.maskBits = 65533;
    hero.GetBody().GetFixtureList().SetFilterData(filter);
    var vel = hero.GetBody().GetLinearVelocity();
    vel.y = 0;
    hero.GetBody().SetLinearVelocity(vel);
}
//Ladder functionality
if((fixa == "hero" && fixb == "ladder") || (fixb == "hero" && fixa == "ladder"))
    {
        heroOnLadder = false;
        var filter = hero.GetBody().GetFixtureList().GetFilterData();
        filter.maskBits = 65535;
        hero.GetBody().GetFixtureList().SetFilterData(filter);
    }
```

- Use a flag to tell when the hero is on the ladder and switch on this flag to a different control scheme which applies a linear velocity in whatever direction is pressed:

```
// RIGHT_CONTROL_KEY_DOWN
case 68:
case 39:
    if(heroOnLadder)
    {
        var vel = hero.GetBody().GetLinearVelocity();
        vel.x = 5;
        hero.GetBody().SetLinearVelocity(vel);
    }
    else
    {
        hero.GetBody().ApplyImpulse(new b2Vec2(5,0), hero.GetBody().GetWorldCenter());
        var vel = hero.GetBody().GetLinearVelocity();
        if(vel.x > 10)
        {
            vel.x = 10;
            hero.GetBody().SetLinearVelocity(vel);
        }
    }
    break;
```

- Using the heroOnLadder flag apply an upward force roughly equal to gravity at each physics step to counteract it's effect:



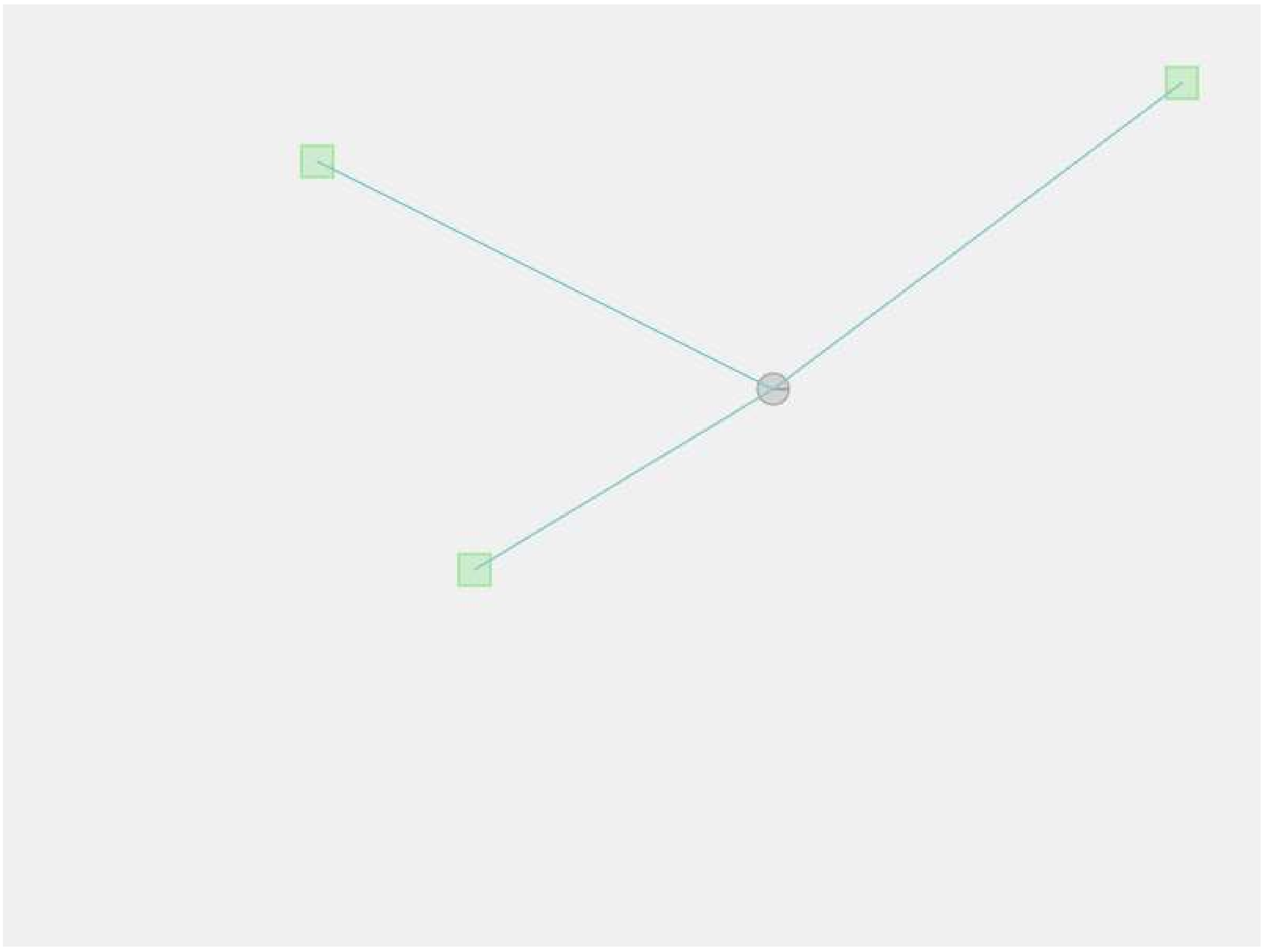
```
// Update World Loop
function update() {

    if(heroOnLadder)
    {
        hero.GetBody().ApplyForce(new b2Vec2(0, -3), hero.GetBody().GetWorldCenter());
    }
    world.Step(
    1/60, // framerate
    10, // velocity iterations
    10 // position iterations
    );
```

This works as the hero collides with the ladder but is able to pass through due to it being set as a sensor, the hero does not collide with the upper platform while colliding with the ladder, a separate control scheme is used to reduce the acceleration applied to the hero making them more controllable, and the upward force makes traversing the ladder more stable as gravity is effectively significantly reduced. There wasn't enough time to perfect this but it is a working solution.

Cut the Rope

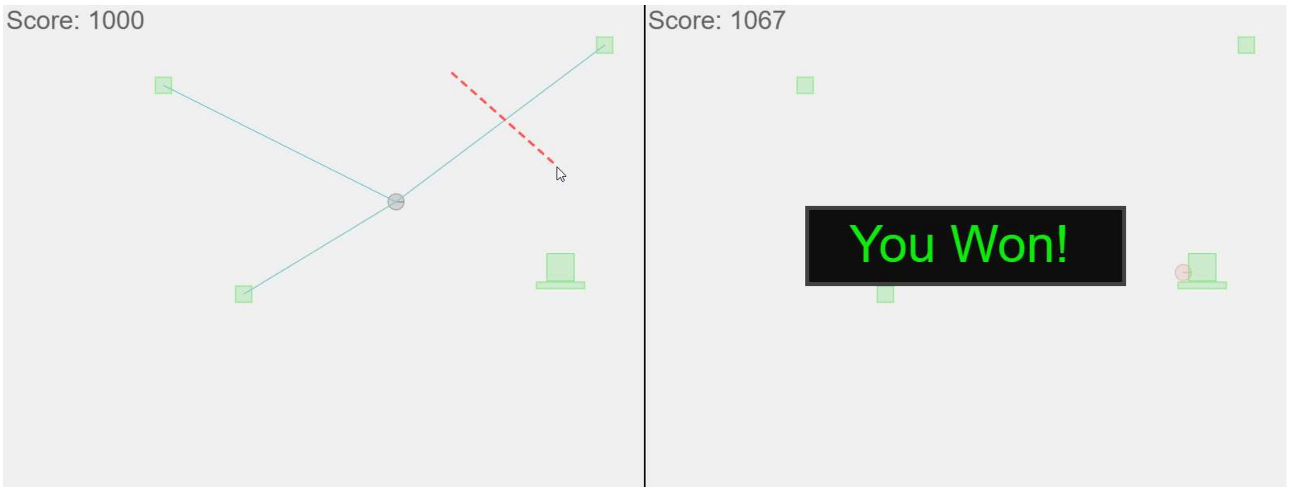
Posted by CALUM LINDSAY on 03 October 2024, 14:22
Last updated 03 October 2024, 20:55

For our second tutorial we were given a basic 2D "game" similar to classic Cut the Rope but only containing ropes consisting of a single distance joint (which gives more of a infinitely rigid metal beam feel) and the algorithm to detect line intersections in order to allow using the mouse to destroy these joints as pictured:

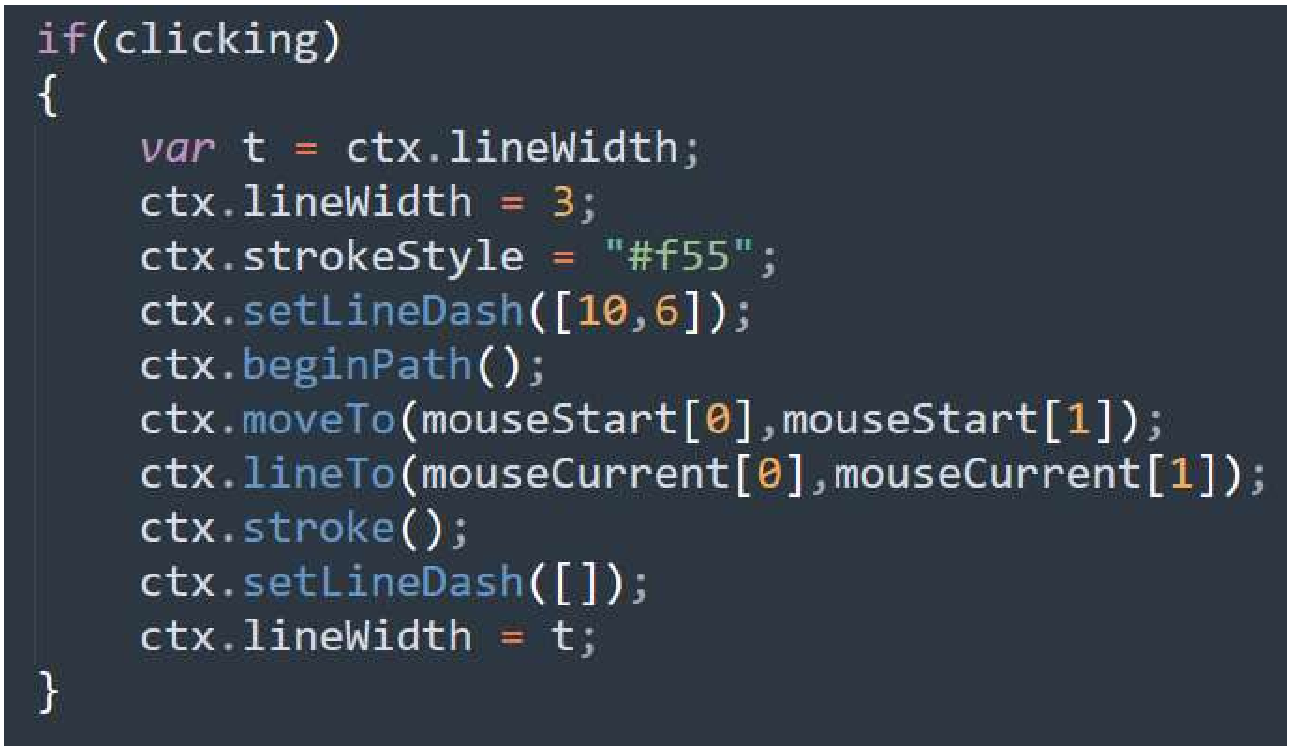


The Basics

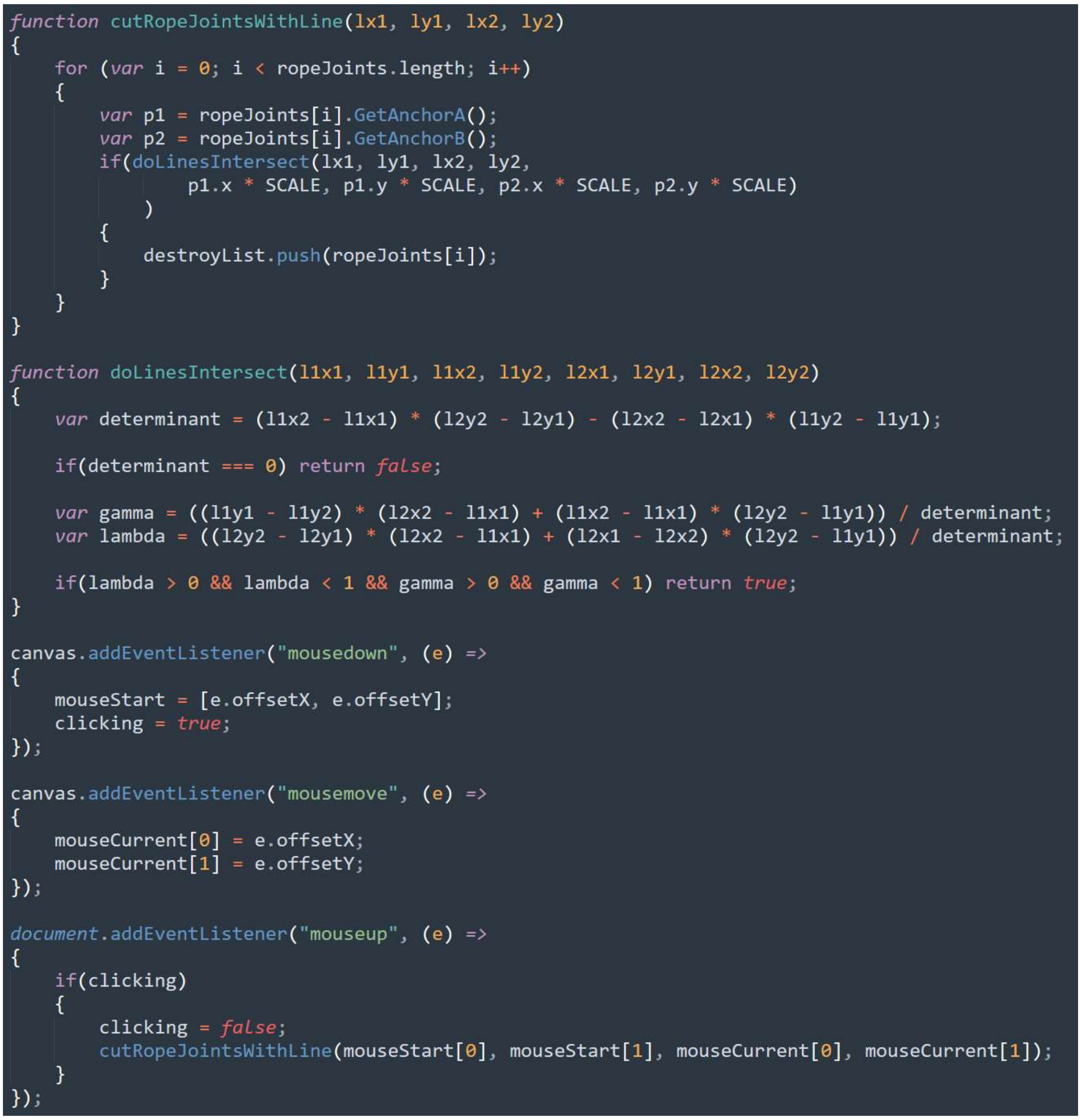
As it was covered in tutorial 1 adding a box to act as the frog, setting them to be a sensor, and having a collision with the food was an easy task, and I reused my code from Tutorial 1 to end the game and show either You Won or You Lost as appropriate. There are 2 game ending conditions, one is if the frog gets the food and the other is if the food leaves the screen by more than 200px (the buffer space is to reduce the "jerky feel" of instantly losing if the food slightly leaves the screen, plus it allows some wiggle room for level design).



I changed the cutting mechanism for personal preference so that you drag and let go at which point the cut takes effect and it is indicated where will be cut by drawing a red dashed line between where the mouse down event occurred and the current mouse position. The mouse position is tracked in the mouse move listener so that we can do the cut in a mouse up event for the whole document thus somewhat avoiding the annoyance of accidentally going out of the canvas area during a mouse drag.

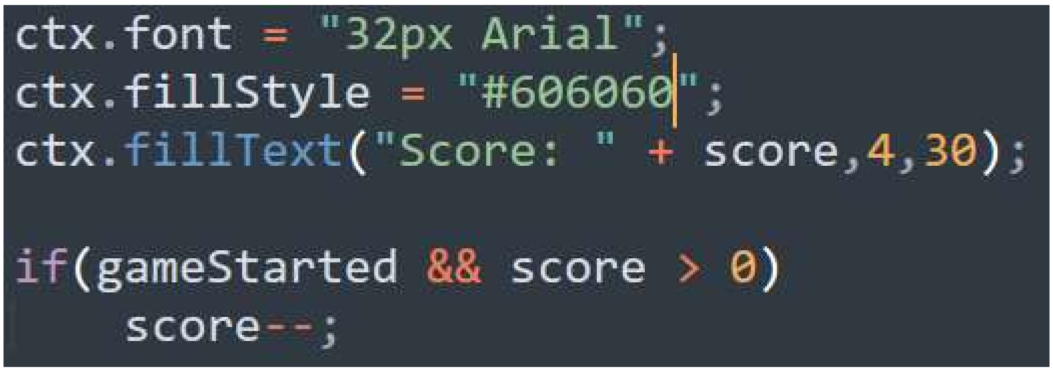


```
if(clicking)
{
    var t = ctx.lineWidth;
    ctx.lineWidth = 3;
    ctx.strokeStyle = "#f55";
    ctx.setLineDash([10,6]);
    ctx.beginPath();
    ctx.moveTo(mouseStart[0],mouseStart[1]);
    ctx.lineTo(mouseCurrent[0],mouseCurrent[1]);
    ctx.stroke();
    ctx.setLineDash([]);
    ctx.lineWidth = t;
}
```



```
function cutRopeJointsWithLine(lx1, ly1, lx2, ly2)
{
    for (var i = 0; i < ropeJoints.length; i++)
    {
        var p1 = ropeJoints[i].GetAnchorA();
        var p2 = ropeJoints[i].GetAnchorB();
        if(doLinesIntersect(lx1, ly1, lx2, ly2,
                p1.x * SCALE, p1.y * SCALE, p2.x * SCALE, p2.y * SCALE)
            )
        {
            destroyList.push(ropeJoints[i]);
        }
    }
}

function doLinesIntersect(l1x1, l1y1, l1x2, l1y2, l2x1, l2y1, l2x2, l2y2)
{
    var determinant = (l1x2 - l1x1) * (l2y2 - l2y1) - (l2x2 - l2x1) * (l1y2 - l1y1);

    if(determinant === 0) return false;

    var gamma = ((l1y1 - l1y2) * (l2x2 - l1x1) + (l1x2 - l1x1) * (l2y2 - l1y1)) / determinant;
    var lambda = ((l2y2 - l2y1) * (l2x2 - l1x1) + (l2x1 - l2x2) * (l2y2 - l1y1)) / determinant;

    if(lambda > 0 && lambda < 1 && gamma > 0 && gamma < 1) return true;
}

canvas.addEventListener("mousedown", (e) =>
{
    mouseStart = [e.offsetX, e.offsetY];
    clicking = true;
});

canvas.addEventListener("mousemove", (e) =>
{
    mouseCurrent[0] = e.offsetX;
    mouseCurrent[1] = e.offsetY;
});

document.addEventListener("mouseup", (e) =>
{
    if(clicking)
    {
        clicking = false;
        cutRopeJointsWithLine(mouseStart[0], mouseStart[1], mouseCurrent[0], mouseCurrent[1]);
    }
});
```

A score mechanic was added with the score beginning at 1000, counting down 1 every frame to a minimum of 0, and finally adding 500 for getting the food to the frog.



```
ctx.font = "32px Arial";
ctx.fillStyle = "#606060";
ctx.fillText("Score: " + score,4,30);

if(gameStarted && score > 0)
    score--;
```

```
listener.BeginContact = function(contact)
{
    var fixa = contact.GetFixtureA().GetBody().GetUserData().id;
    var fixb = contact.GetFixtureB().GetBody().GetUserData().id;
    if((fixa == "frog" && fixb == "food") || (fixb=="frog" && fixa=="food"))
    {
        won = true;
        score += 500;
        running = false;
    }
}
```

Better rope

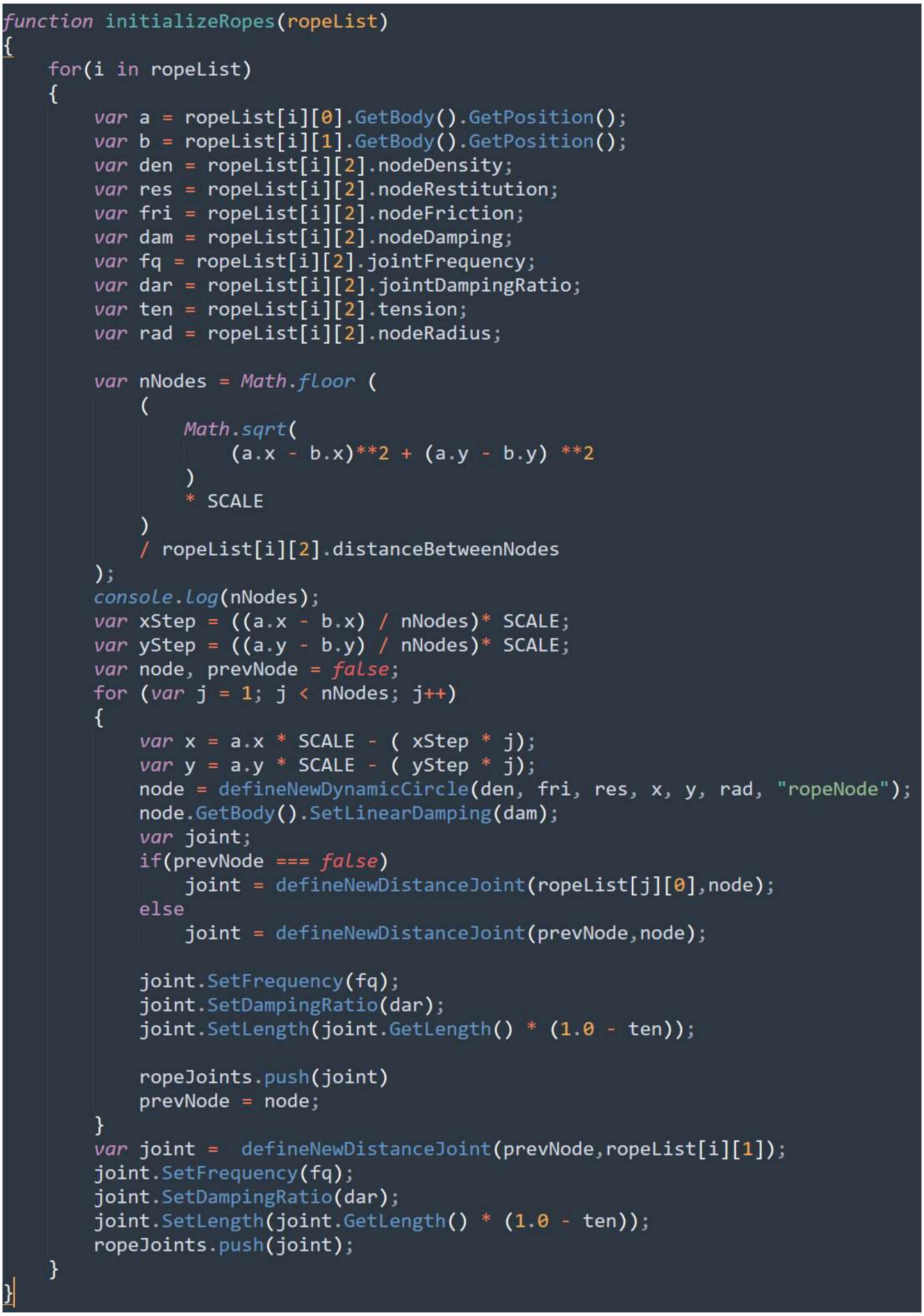
My first thoughts are that to make a better rope we just need a long chain of distance joints to allow more points of articulation and this can be achieved by adding some equally spaced bodies called hereafter "nodes" interconnected with distance joints. Additionally there are a good number of editable parameters which have been encapsulated in the following class which is being used more like a traditional struct here:

```
class ropeMaterial
{
    distanceBetweenNodes = 20;// The distance between the nodes of the rope
    tension = 0.1;            // Shortens the rope to put it under tension, max 1

    nodeRadius = 2;           // Radius of the nodes
    nodeDensity = 0.5;        // Relative mass of nodes for their volume
    nodeFriction = 0.5;       // Friction of nodes in contact with surfaces
    nodeRestitution = 0.01;   // Node bounciness
    nodeDamping = 0.2;        // Damping of node movement

    jointFrequency = 30;      // Lower values are less stretchy, max 30
    jointDampingRatio = 0.9;  // Damping of stretch behaviour
}
```

The function that creates all the the ropes takes a list of lists with each sub list representing a single rope and containing 2 anchoring bodies for the rope to connect together as well as an instance of RopeMaterial to define the rope's characteristics. The process is outlined as follows:

- We find the distance between the 2 anchors and divide by the distance between nodes for the rope being created to get the number of nodes required.
- We divide the distance along the x and y axis each by the number of nodes to get the step distance on each axis for each node.
- For each node with index j:
    - the x position is set to anchor a's x position - (the x step * j)
    - the y position is set to anchor a's y position - (the y step * j)
    - the node is created using the relevant properties from the rope's material
    - a joint is created between the previous node and the new node

    - the new joint is modified using the relevant properties from the rope's material
- a joint is created between the final node and anchor b



```
function initializeRopes(ropeList)
{
    for(i in ropeList)
    {
        var a = ropeList[i][0].GetBody().GetPosition();
        var b = ropeList[i][1].GetBody().GetPosition();
        var den = ropeList[i][2].nodeDensity;
        var res = ropeList[i][2].nodeRestitution;
        var fri = ropeList[i][2].nodeFriction;
        var dam = ropeList[i][2].nodeDamping;
        var fq = ropeList[i][2].jointFrequency;
        var dar = ropeList[i][2].jointDampingRatio;
        var ten = ropeList[i][2].tension;
        var rad = ropeList[i][2].nodeRadius;

        var nNodes = Math.floor (
            (
                Math.sqrt(
                    (a.x - b.x)**2 + (a.y - b.y) **2
                )
                * SCALE
            )
            / ropeList[i][2].distanceBetweenNodes
        );
        console.log(nNodes);
        var xStep = ((a.x - b.x) / nNodes)* SCALE;
        var yStep = ((a.y - b.y) / nNodes)* SCALE;
        var node, prevNode = false;
        for (var j = 1; j < nNodes; j++)
        {
            var x = a.x * SCALE - ( xStep * j);
            var y = a.y * SCALE - ( yStep * j);
            node = defineNewDynamicCircle(den, fri, res, x, y, rad, "ropeNode");
            node.GetBody().SetLinearDamping(dam);
            var joint;
            if(prevNode === false)
                joint = defineNewDistanceJoint(ropeList[j][0],node);
            else
                joint = defineNewDistanceJoint(prevNode,node);

            joint.SetFrequency(fq);
            joint.SetDampingRatio(dar);
            joint.SetLength(joint.GetLength() * (1.0 - ten));

            ropeJoints.push(joint)
            prevNode = node;
        }
        var joint =  defineNewDistanceJoint(prevNode,ropeList[i][1]);
        joint.SetFrequency(fq);
        joint.SetDampingRatio(dar);
        joint.SetLength(joint.GetLength() * (1.0 - ten));
        ropeJoints.push(joint);
    }
}
```

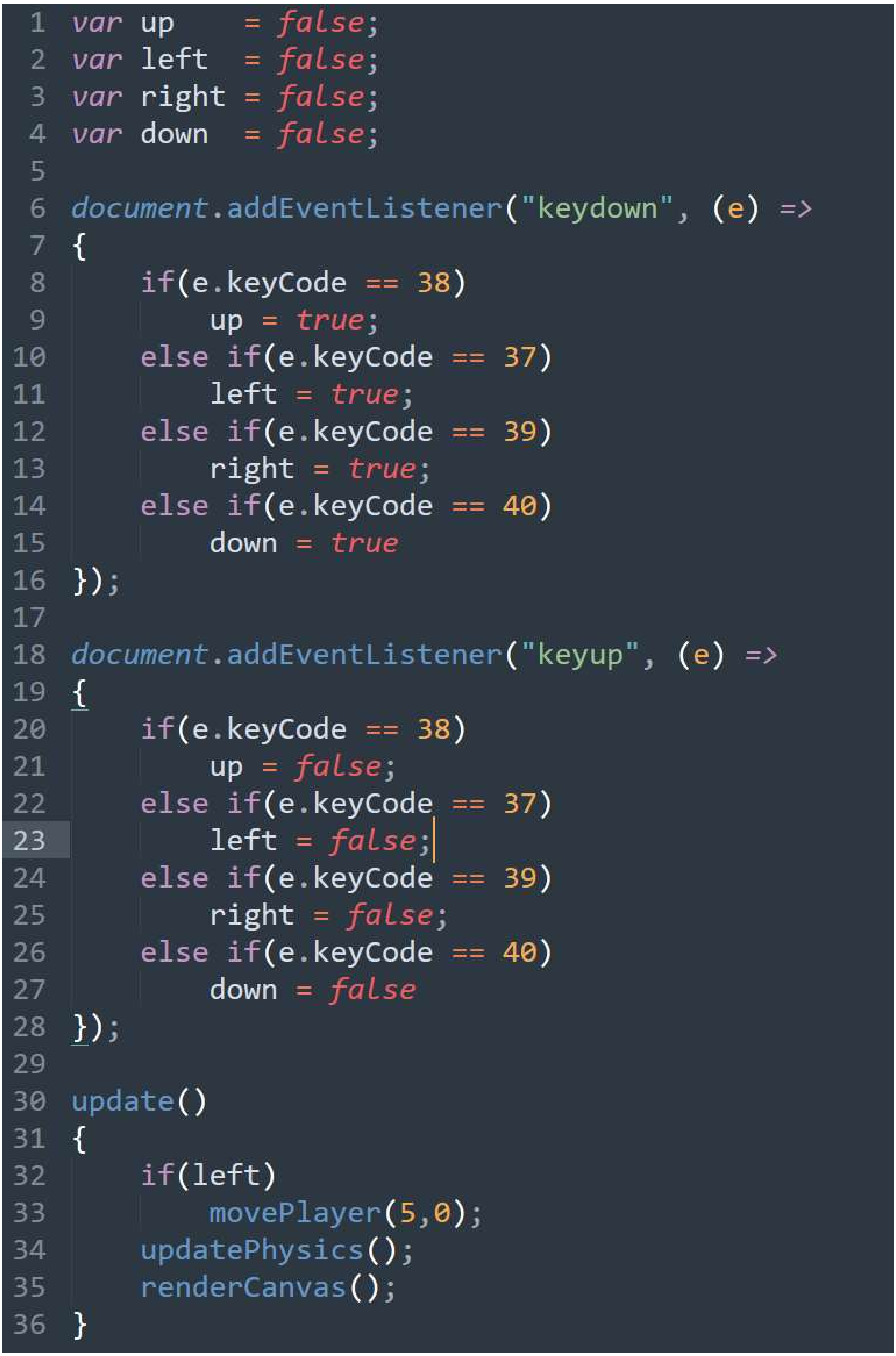
Problems and Solutions

- In the original implementation the values were hardcoded but it was quickly noticed that experimentation was awkward and a struct to hold these values would be very advantageous.
- The rope will go through objects that fit between the nodes. This is fine for the purposes here but may not be in other cases where the node size could be increased, more nodes could be used, or perhaps rectangular nodes could be used with smaller gaps, 2 distance joints per node, and the joints placed close to the 2 furthest apart edges of the rectangle.
- The original implementation did not take into account the nodes when they are completely detached from any joints, this was resolved by checking both associated nodes when a joint is deleted and deleting it if it has no associated joints (This can be seen in the 2nd image of this post).

Adapting Keyboard Events for Game Development

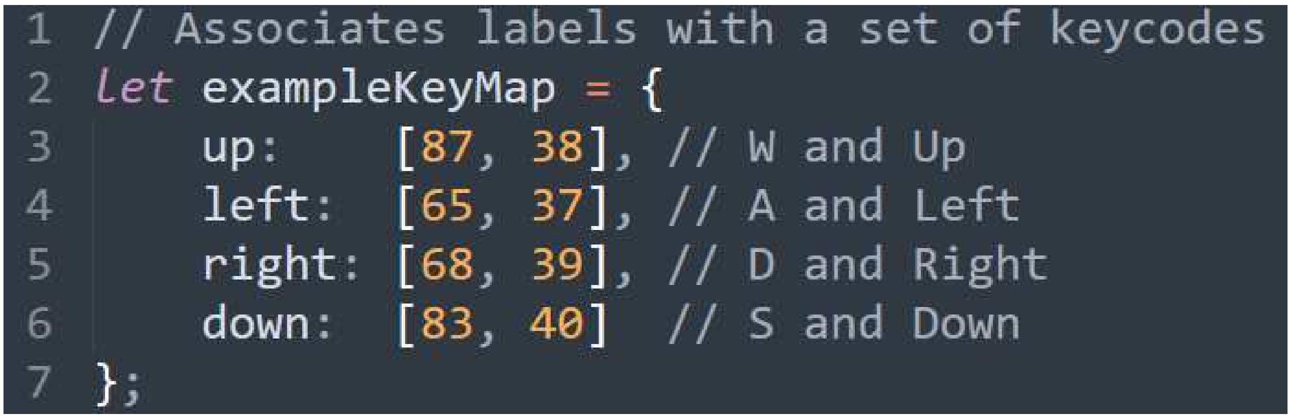
Posted by CALUM LINDSAY on 06 December 2024, 12:03
Last updated 06 December 2024, 15:23

In game development it is common to need to know the state of multiple keyboard keys every frame to perform actions like moving the character, however, the way the browser provides information to the application about this is through the registration of event callbacks. The simplest solution to this problem is to create callbacks for the up and down events and maintain a collection of booleans as such:



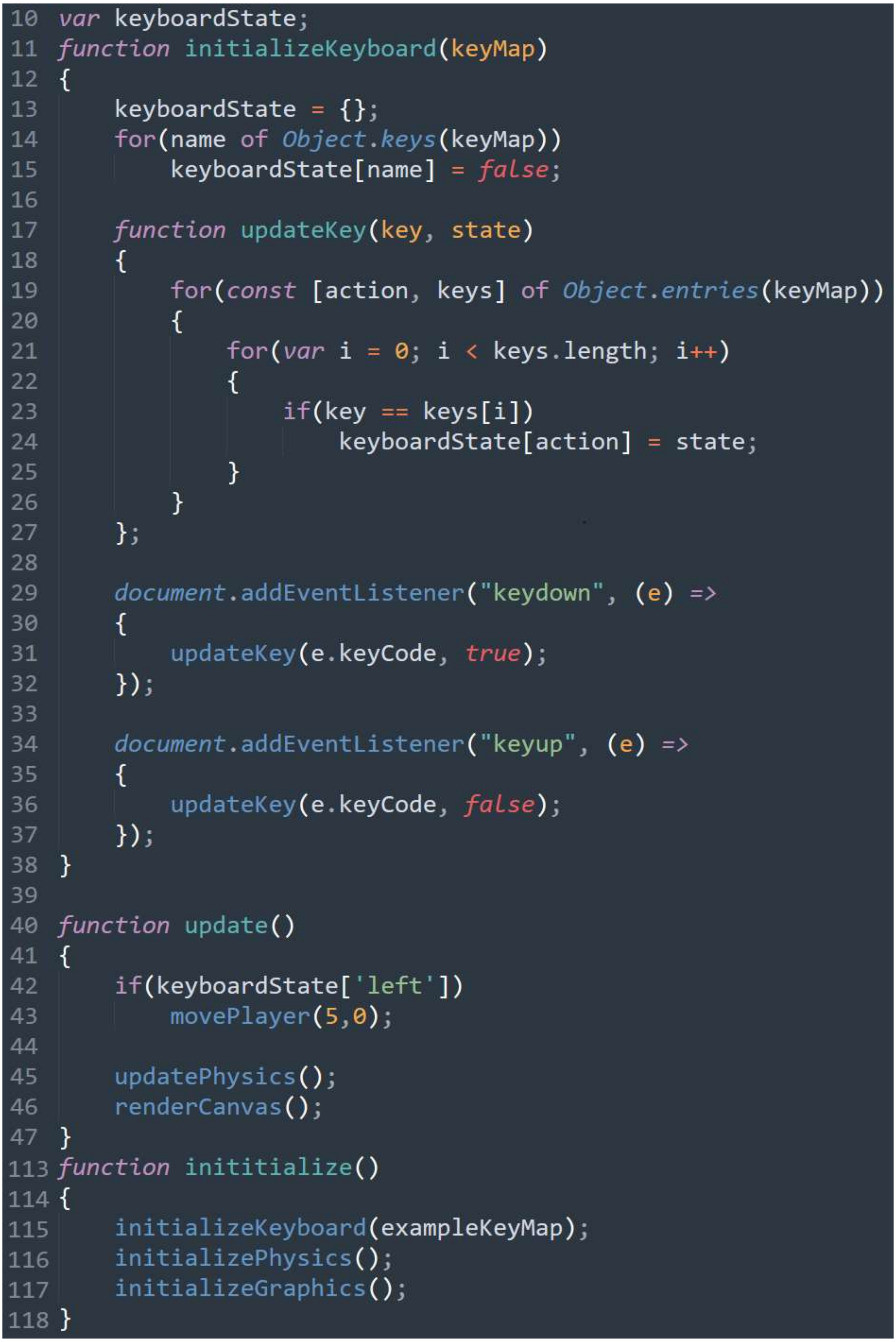
```
var up    = false;
var left  = false;
var right = false;
var down  = false;

document.addEventListener("keydown", (e) =>
{
    if(e.keyCode == 38)
        up = true;
    else if(e.keyCode == 37)
        left = true;
    else if(e.keyCode == 39)
        right = true;
    else if(e.keyCode == 40)
        down = true
});

document.addEventListener("keyup", (e) =>
{
    if(e.keyCode == 38)
        up = false;
    else if(e.keyCode == 37)
        left = false;
    else if(e.keyCode == 39)
        right = false;
    else if(e.keyCode == 40)
        down = false
});

update()
{
    if(left)
        movePlayer(5,0);
    updatePhysics();
    renderCanvas();
}
```

This is simple and works fine but what if we have multiple keys that do the same thing? Our options are to check for multiple booleans in our game logic for all the keys that do a certain action or maintain a collection of booleans that represent different actions that the player wishes to perform and handle the mapping between keys in our callback. Neither of these options is particularly elegant nor reusable, so how about we introduce the concept of an input map:



```
// Associates labels with a set of keycodes
let exampleKeyMap = {
    up:    [87, 38], // W and Up
    left:  [65, 37], // A and Left
    right: [68, 39], // D and Right
    down:  [83, 40]  // S and Down
};
```

This structure maps collections of key inputs to actions which our callback can then iterate through and update the state of our booleans very succinctly, finally, we will package this into an initializeKeyboard function that takes an input map as it's argument and a global keyboardState object which maintains the state of our named inputs. We can now initialize the keyboard with a single call and perform actions each frame based on the state of the keyboard:



```javascript
var keyboardState;
function initializeKeyboard(keyMap)
{
    keyboardState = {};
    for(name of Object.keys(keyMap))
        keyboardState[name] = false;

    function updateKey(key, state)
    {
        for(const [action, keys] of Object.entries(keyMap))
        {
            for(var i = 0; i < keys.length; i++)
            {
                if(key == keys[i])
                    keyboardState[action] = state;
            }
        }
    };

    document.addEventListener("keydown", (e) =>
    {
        updateKey(e.keyCode, true);
    });

    document.addEventListener("keyup", (e) =>
    {
        updateKey(e.keyCode, false);
    });
}

function update()
{
    if(keyboardState['left'])
        movePlayer(5,0);

    updatePhysics();
    renderCanvas();
}
function inititialize()
{
    initializeKeyboard(exampleKeyMap);
    initializePhysics();
    initializeGraphics();
}
```

Brilliant! This solves our primary case, however, what happens if we want a user to be able to press a key and say a door opens or closes depending on it's state, well, the door would toggle between open and closed each frame meaning the user has to press the key for exactly one frame in order to get the expected behaviour! We could solve this by creating a second callback for the down event or tracking when the state changes in our game logic, but our callback already has this information so why not extend it to allow registering of callbacks for individual actions? We add three objects to store our callbacks and a registration function that takes the type of callback, the action to apply it to and the callback itself:

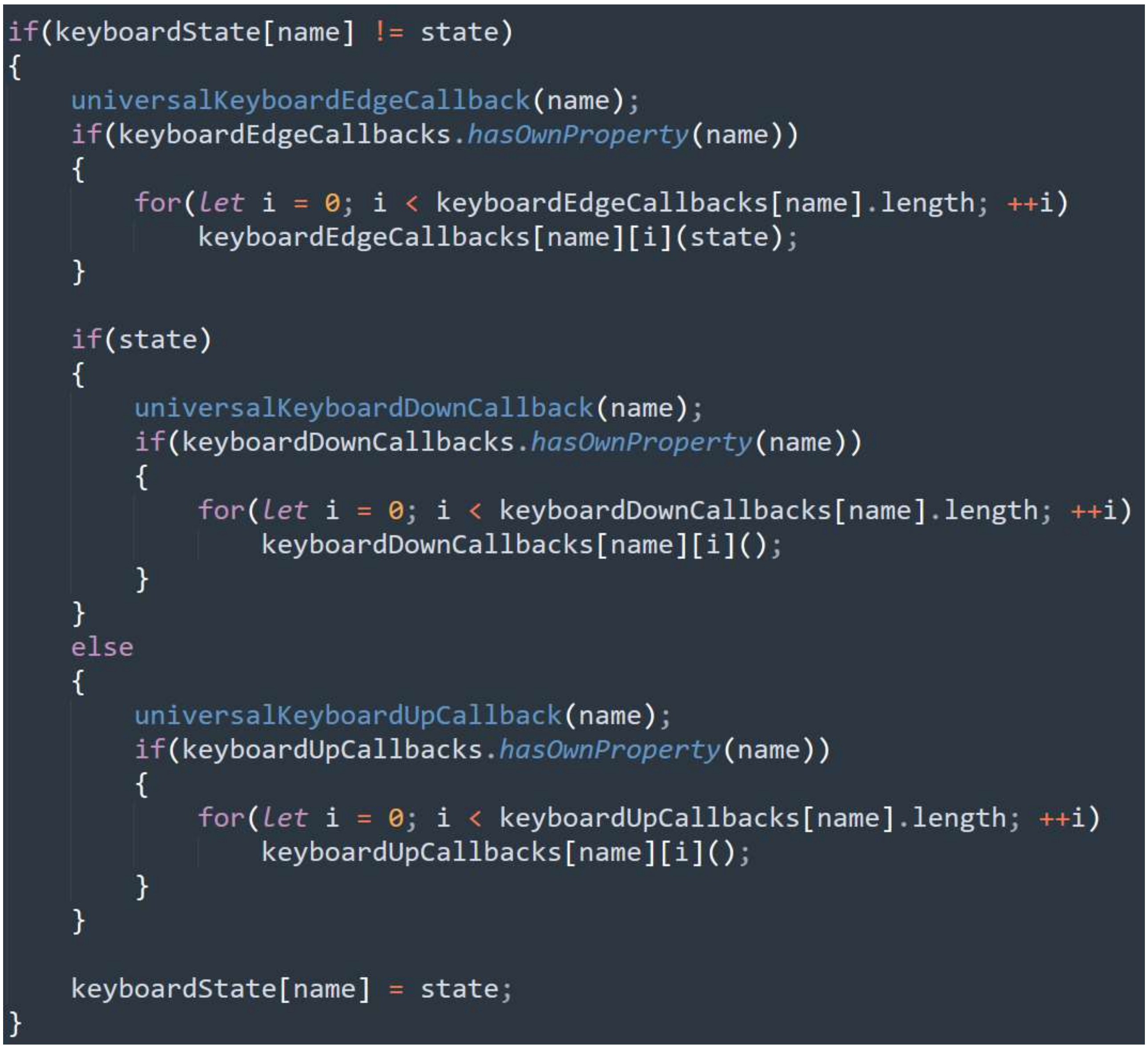
```
var keyboardUpCallbacks = {};
var keyboardDownCallbacks = {};
// Both up and down, these callbacks receives the state of the
//   key as an argument
var keyboardEdgeCallbacks = {};
```

```
if(key == keys[i])
{
    if(keyboardState[name] != state)
    {
        if(keyboardEdgeCallbacks.hasOwnProperty(name))
        {
            for(let i = 0; i < keyboardEdgeCallbacks[name].length; ++i)
                keyboardEdgeCallbacks[name][i](state);
        }

        if(state)
        {
            if(keyboardDownCallbacks.hasOwnProperty(name))
            {
                for(let i = 0; i < keyboardDownCallbacks[name].length; ++i)
                    keyboardDownCallbacks[name][i]();
            }
        }
        else
        {
            if(keyboardUpCallbacks.hasOwnProperty(name))
            {
                for(let i = 0; i < keyboardUpCallbacks[name].length; ++i)
                    keyboardUpCallbacks[name][i]();
            }
        }

        keyboardState[name] = state;
    }
}
```

```javascript
function registerInputCallback(label, type, fun)
{
    let t;
    switch(type)
    {
        case "edge": t = keyboardEdgeCallbacks; break;
        case "up": t = keyboardUpCallbacks; break;
        case "down": t = keyboardDownCallbacks; break;
        default: return;
    }

    if(t.hasOwnProperty(label))
        t[label].push(fun);
    else
        t[label] = [ fun ];
}
```

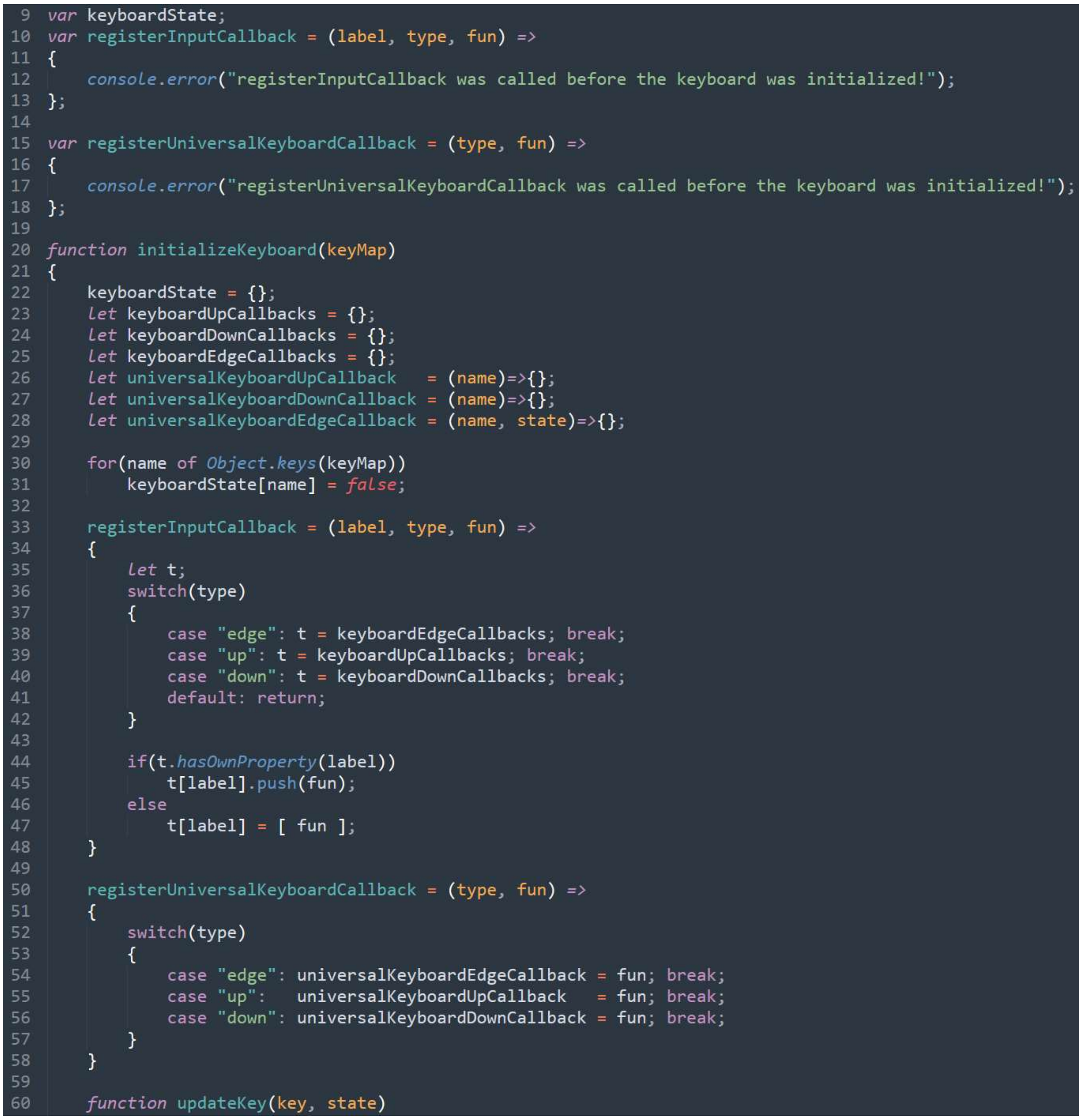
Occasionally you may want a callback that fires for any action, such as for debugging purposes, and this can easily be supported by adding another registration function, three overrideable functions, and modifying the main callback to call these where appropriate:

```javascript
// Universal callbacks - applied to every key and overrideable
//    using registerUniversalKeyboardCallback
var universalKeyboardUpCallback   = (name)=>{};
var universalKeyboardDownCallback = (name)=>{};
var universalKeyboardEdgeCallback = (name, state)=>{};
function registerUniversalKeyboardCallback(type, fun)
{
    switch(type)
    {
        case "edge": universalKeyboardEdgeCallback = fun; break;
        case "up":   universalKeyboardUpCallback   = fun; break;
        case "down": universalKeyboardDownCallback = fun; break;
    }
}
```



```
if(keyboardState[name] != state)
{
    universalKeyboardEdgeCallback(name);
    if(keyboardEdgeCallbacks.hasOwnProperty(name))
    {
        for(let i = 0; i < keyboardEdgeCallbacks[name].length; ++i)
            keyboardEdgeCallbacks[name][i](state);
    }

    if(state)
    {
        universalKeyboardDownCallback(name);
        if(keyboardDownCallbacks.hasOwnProperty(name))
        {
            for(let i = 0; i < keyboardDownCallbacks[name].length; ++i)
                keyboardDownCallbacks[name][i]();
        }
    }
    else
    {
        universalKeyboardUpCallback(name);
        if(keyboardUpCallbacks.hasOwnProperty(name))
        {
            for(let i = 0; i < keyboardUpCallbacks[name].length; ++i)
                keyboardUpCallbacks[name][i]();
        }
    }

    keyboardState[name] = state;
}
```

At this point we have a pretty solid data-driven solution that works well, however, we have some global state that doesn't need to be global so let's move everything in to the initialization function except the keyboardState and registration functions, but, wait, our registration functions need to access the objects storing our callbacks which are now in the initializeKeyboard function! Here we can use a little magic and assign the registration functions to anonymous functions within the initializeKeyboard function when it is called so they have access to the variables they need:



```
var keyboardState;
var registerInputCallback = (label, type, fun) =>
{
    console.error("registerInputCallback was called before the keyboard was initialized!");
};

var registerUniversalKeyboardCallback = (type, fun) =>
{
    console.error("registerUniversalKeyboardCallback was called before the keyboard was initialized!");
};

function initializeKeyboard(keyMap)
{
    keyboardState = {};
    let keyboardUpCallbacks = {};
    let keyboardDownCallbacks = {};
    let keyboardEdgeCallbacks = {};
    let universalKeyboardUpCallback   = (name)=>{};
    let universalKeyboardDownCallback = (name)=>{};
    let universalKeyboardEdgeCallback = (name, state)=>{};

    for(name of Object.keys(keyMap))
        keyboardState[name] = false;

    registerInputCallback = (label, type, fun) =>
    {
        let t;
        switch(type)
        {
            case "edge": t = keyboardEdgeCallbacks; break;
            case "up": t = keyboardUpCallbacks; break;
            case "down": t = keyboardDownCallbacks; break;
            default: return;
        }

        if(t.hasOwnProperty(label))
            t[label].push(fun);
        else
            t[label] = [ fun ];
    }

    registerUniversalKeyboardCallback = (type, fun) =>
    {
        switch(type)
        {
            case "edge": universalKeyboardEdgeCallback = fun; break;
            case "up":   universalKeyboardUpCallback   = fun; break;
            case "down": universalKeyboardDownCallback = fun; break;
        }
    }

    function updateKey(key, state)
```

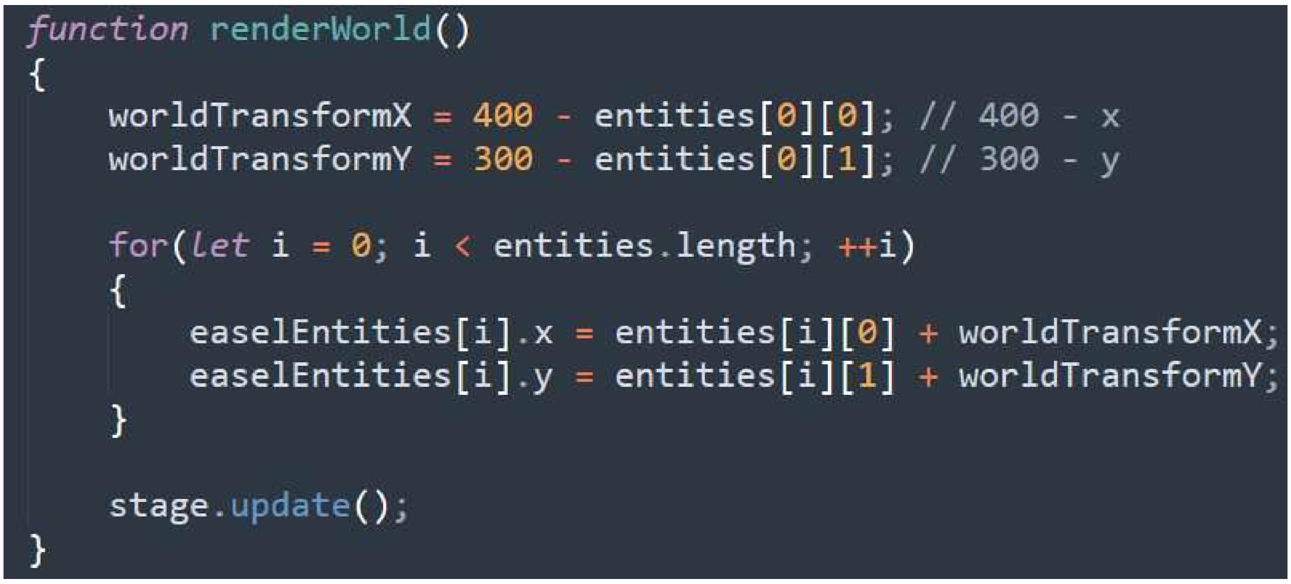
The final solution is self-contained, extensible and easy to use, however, this level of abstraction is not free, it will certainly cost us some performance but, in simple games it serves a useful role in reducing development time.

Viewport Management

Posted by CALUM LINDSAY on 09 December 2024, 18:49
Last updated 10 December 2024, 0:30

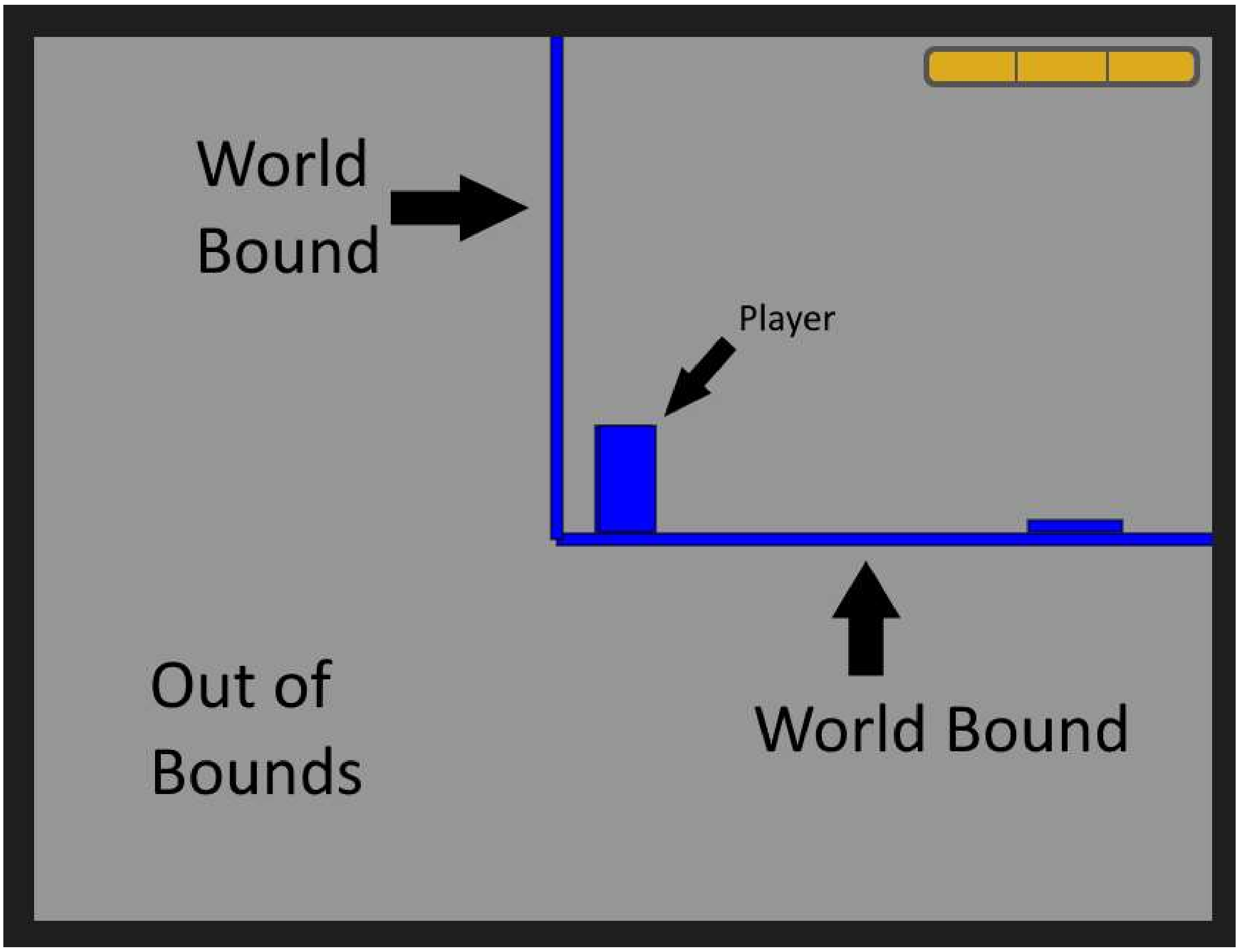
An unavoidable problem when developing a game where a continuous open world is desired is if the world is larger than can reasonably be rendered in the viewport. This problem only really requires basic maths, unless of course we are discussing 3D where we would have to introduce the concepts of quaternions and 4x4 Matrices, but I will spare you for today and deal only in 2D! We will discuss here a simple solution, there is also an accompanying video (link at the bottom) that shows the concepts discussed in action.

Ok so where do we start? The simplest thing we might want to do is follow the player's entity (object in the game world) around. The simplest way to do this is to calculate an x and y value that if added to the entities position would place them in the centre of the viewport, and then apply this to all objects in the world. In our case our viewport is 800 pixels wide by 600 pixels tall so we want the entity to be placed at (400, 300). Here is the function that calculates the translation required and applies it to every entity in the world:



```
function renderWorld()
{
    worldTransformX = 400 - entities[0][0]; // 400 - x
    worldTransformY = 300 - entities[0][1]; // 300 - y

    for(let i = 0; i < entities.length; ++i)
    {
        easelEntities[i].x = entities[i][0] + worldTransformX;
        easelEntities[i].y = entities[i][1] + worldTransformY;
    }

    stage.update();
}
```

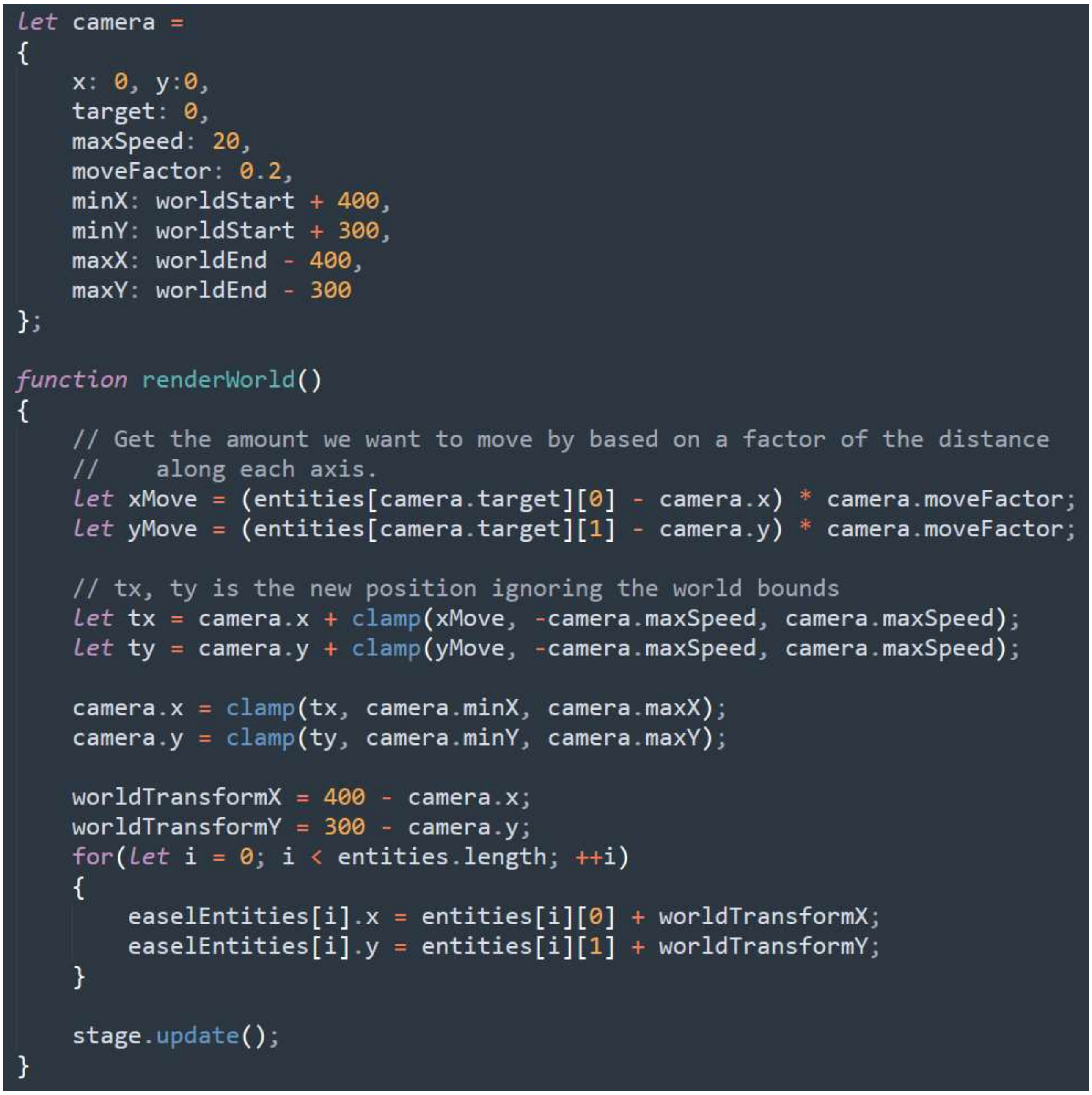
The maths is really quite simple, the harder part is understanding that we need to move the world around the player rather than moving the player within the world as seems more logical, although once this clicks the rest is easy! When the player goes to the edge of the world, the camera looks outside of the bounds of the world, which is probably not what we want in most cases.



Additionally we may want to change the entity that we are looking at or even set a fixed position in the world at times. We will add a camera object that has a target entity, position, and, minimum and maximum positions. Currently it will always follow the player entity but this could be changed to another entity during gameplay or ignored rendering whatever is at the position the camera is set to:

```
function clamp(value, min, max)
{
    return value < min ? min : (value > max ? max : value);
}

let worldStart = 256;
let worldEnd = worldStart + 2048;

let camera =
{
    x: 0, y:0,
    target: 0,
    minX: worldStart + 400,
    minY: worldStart + 300,
    maxX: worldEnd - 400,
    maxY: worldEnd - 300
};

function renderWorld()
{
    let tx = entities[camera.target][0];
    let ty = entities[camera.target][1];

    camera.x = clamp(tx, camera.minX, camera.maxX);
    camera.y = clamp(ty, camera.minY, camera.maxY);

    worldTransformX = 400 - camera.x;
    worldTransformY = 300 - camera.y;
    for(let i = 0; i < entities.length; ++i)
    {
        easelEntities[i].x = entities[i][0] + worldTransformX;
        easelEntities[i].y = entities[i][1] + worldTransformY;
    }

    stage.update();
}
```

This code is basically the same as the previous version but it can no longer go outside the bounds of the game (as shown in the video) this works by calculating the min x, min y, max x and max y, then clamping the camera's position between these values. What we have now is completely usable, however, this is where you can unleash your creativity! I will present a simple example of how you could adjust this camera to move more smoothly, especially when the target is changed:



```
let camera =
{
    x: 0, y:0,
    target: 0,
    maxSpeed: 20,
    moveFactor: 0.2,
    minX: worldStart + 400,
    minY: worldStart + 300,
    maxX: worldEnd - 400,
    maxY: worldEnd - 300
};

function renderWorld()
{
    // Get the amount we want to move by based on a factor of the distance
    //    along each axis.
    let xMove = (entities[camera.target][0] - camera.x) * camera.moveFactor;
    let yMove = (entities[camera.target][1] - camera.y) * camera.moveFactor;

    // tx, ty is the new position ignoring the world bounds
    let tx = camera.x + clamp(xMove, -camera.maxSpeed, camera.maxSpeed);
    let ty = camera.y + clamp(yMove, -camera.maxSpeed, camera.maxSpeed);

    camera.x = clamp(tx, camera.minX, camera.maxX);
    camera.y = clamp(ty, camera.minY, camera.maxY);

    worldTransformX = 400 - camera.x;
    worldTransformY = 300 - camera.y;
    for(let i = 0; i < entities.length; ++i)
    {
        easelEntities[i].x = entities[i][0] + worldTransformX;
        easelEntities[i].y = entities[i][1] + worldTransformY;
    }

    stage.update();
}
```

So in this example the camera moves towards its target on each frame by some factor of the distance between them rather than sticking absolutely to the target, the factor and maxSpeed can be adjusted to create different characteristics, visual effects could be applied, mouse input could be incorporated, really the sky is the limit here! That's all I have for today and as promised here is the video link: https://youtu.be/4ar6M8oTLO8.