Torch Run: Design and Implementation

Kyle Tuckey

Game Summary

Torch run is a top down, rogue-like game with elements of time-based and puzzle games with the singular goal of escaping the Ghost ridden dungeon before your torch runs out.

Objective

The objective of the game is to escape the dungeon before your torch runs out. In order to escape the dungeon the player needs to navigate the dark corridors within each level using the torch in their possession to illuminate the way through the dungeon. The player will encounter roaming Ghosts who will attack the player when revealed by the torch's light, and will also need to open gate's that block their path by finding the various levers spread throughout the game.

Rules

There are a few rules that the player needs to follow in order to successfully escape the dungeon:

- The player must use his torch to illuminate his path through the level.
- The player has to collect torch's throughout the level to extend the duration and power of his torch.
- Avoid the insidious Ghost's that roam the dungeon, they will extinguish your torch when they catch you.
- Find levers to open gates that will block the player’s path to the exit of the level.

Reach the end of the level.

Gameplay

The player will chase the mouse cursor in order to move, on a mobile device the player will simply move in the direction that was last pointed to on the screen. While moving the players light will reveal paths throughout the dungeon that the player can take, some paths will lead the player to a dead end, others will take the player on the right path to his escape.

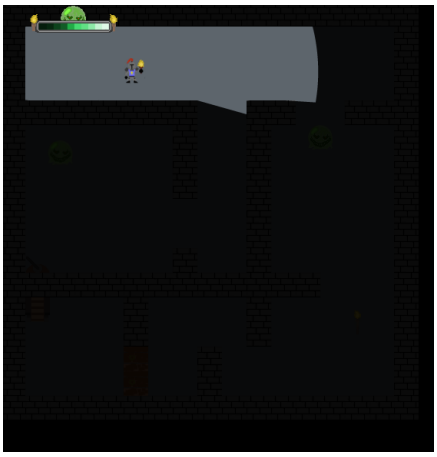
Game Structure

The structure of torch run consists of a total of 4 screens’, a menu screen, a death screen, a play screen, and an end screen.

Menu Screen

The menu screen is the first screen the user should see and consists of a title, some text instructing the player on how to move, and a play button that takes the player to the game screen. There was no need for anything else to be on the screen, so that is all there is.

Play screen



The play screen is the second screen the player should see, the play screen consists of multiple object's, it contains the player, the enemies, the walls, the torches, the torch indicator, levers, gates and an exit. The play screen contains 2 level's all of which are read in using JSON files.

Death screen and End screen

The death and end screens can be seen in either order, if the player dies (collides with a ghost or depletes his torch) then he will see the death screen, if he makes it past both level's he will get the end screen. Both are very similar in appearance, one having a red title and a button and the other having a green title. The buttons on both screens do different thing's the death screen button will take the player to the start of his current level. The end screen button will take the player to the menu.

Implementation Explanation

One of the design goal's I had when I started creating this game, was to have the majority of the code be self-contained and be readable and understandable.

Game.js

```
/**
 * Created by Kyle Tuckey on 02/11/2016.
 */
var GAMEWIDTH = 544;
var GAMEHEIGHT = 544;

var game = new Phaser.Game(GAMEWIDTH, GAMEHEIGHT, Phaser.AUTO, 'Torch Run');
game.state.add('boot', bootState);
game.state.add('load', loadState);
game.state.add('menu', menuState);
game.state.add('play', playState);
game.state.add('dead', deadState);
game.state.add('end', endState);


game.state.start('boot');
```

The very first JS file that gets called is the game.js file, what this does is it creates our Phaser game object and uses that object to add game state's, game states are useful for breaking up code that do different things so we have a state that load's all the sprite imagery to the game, and we have another state that creates a menu. We also define the game height as 544 and the width as 544 this was so that the level structure would fit neatly on the screen. Once we have defined our state's we tell the file to start the boot state. I got the idea for this structure from a blog that discusses how to effectively use game states(Emanuele Feronato, 2017).

Boot.js

```
/**
 * Created by Kyle Tuckey on 20/11/2016.
 */
var bootState = {
    create: function(){
        game.physics.startSystem(Phaser.Physics.ARCADE);

        game.state.start('load');
    }
}
```

The boot file is very simple, all that happens here is I enable the arcade physics engine for the game, this would be used for other things that need to be booted in order for the game to work before I start loading in assets, and however the physics engine was the only thing that needed pre booting.

Load.js

```
/**
 * Created by Kyle Tuckey on 20/11/2016.
 */
var loadState = {
    preload: function() {
        game.load.image('wall_spr', 'assets/wall.png');
        game.load.image('gate_spr', 'assets/gate.png');
        game.load.image('door_spr', 'assets/door.png');
        game.load.image('lever_spr', 'assets/lever.png');
        game.load.image('lever_spr2', 'assets/lever2.png');
        game.load.spritesheet('player_spr', 'assets/playerSheet.png', 26, 32, 7);
        game.load.spritesheet('enemy_spr', 'assets/enemySheet.png', 32, 32, 5);
        game.load.image('fuel_spr', 'assets/fuel.png');
        game.load.spritesheet('continue_spr', 'assets/continue.png',128,64,2);
        game.load.image('play_spr', 'assets/play.png');
        game.load.image('placeHolder_spr', 'assets/placeHolder.png');
        game.load.text('level1', 'assets/levels/level1.json');
        game.load.text('level2', 'assets/levels/level2.json');
        game.load.spritesheet('torchBar','assets/torch_bar_sheet.png',113, 35, 11);
    },

    create: function() {
        game.state.start('menu');
    }
};
```

The load file loads in all the games asset's so all the sprites, and sprite sheets as well as the levels. After it has loaded in all the games asset's the file call's the menu state to initialise the game menu.

Menu.js

```
/**
 * Created by Kyle Tuckey on 20/11/2016.
 */
var menuState = {
    create: function() {
        var nameLabel = game.add.text(80,80,'Torch Run', {font:'50px Arial', fill:'#ffffff'});
        var startLabel = game.add.text(80, game.world.height-80,'click the screen to move', {font: '25px Arial', fill: '#ffffff'});
        var button = game.add.button(game.world.width/2, game.world.height/1.5, 'play_spr', this.start)
        if(game.input.mousePointer.isDown)
            game.state.start('play');
    },

    start: function() {
        game.state.start('play');
    }
}
```

The menu is the first screen the user sees when the game is started; it is made up of 2 text objects from the phaser library, and a button. It also has an, if statement to detect when the play button has been pressed in order to start the play state, which is where the majority of the game's functionality is.

Player.js

Before we delve into the functionality of the play.js file, It is important to understand what object's exist within the game in order to better understand what is happening within the play.js file. To start with we will look at the player.js file which handles the majority of the logic in regards to the player.

```
var Player = function(p_x, p_y, file){
    this.sprite = game.add.sprite(p_x, p_y, file);
    this.sprite.anchor.setTo(0.5, 0.5);
    this.sprite.animations.add('idle');
    this.sprite.animations.play('idle', 10, true);
    game.physics.enable(this.sprite, Phaser.Physics.ARCADE);
```

The player object takes in an X and Y position as well as an image to represent it's sprite, these are used to create a game sprite for the player, we then select the idle animation from the sprite sheet we have provided and tell it to play at a rate of 10 frames per second, we also enable physics on the sprite.

```
this.update = function()
{
    if(game.input.onDown.add(this.itemTouched, this))
    {
        game.physics.arcade.moveToPointer(this.sprite, 200);

        if(Phaser.Rectangle.contains(this.sprite.body, game.input.x, game.input.y))
        {
            this.sprite.body.velocity.setTo(0,0);
        }
    }
    else
    {
        this.sprite.body.velocity.setTo(0, 0);
    }
}
```

The player object contains its own update function, all that is handled within this function is the player's movement, the way this work's is the player detects if there has been any user input and then make's the player sprite follow the cursor position, on a mobile device what happens is the player will move to wherever was last pressed on the screen if it can make it there.

```
this.fuelCollision = function(player, fuel)
{
    if(torch.torchPower + 30 > 100){
        torch.torchPower = 100;
        currentPower = 100;
        sheetCount = 0;
    }
    else {
        torch.torchPower += 30;
        currentPower += 30;
        sheetCount -=3;
    }


    fuel.kill();
}
```

This function handle's the collision behaviour between the player and any torch fuel he encounters. the fuel behaviour is relatively simple, all that happens is we add 30 to the torch power if the player encounter's any fuel, if the torch power exceeds 100 we set it back to 100 there are 2 other variable's at play here called currentPower and sheetCount these 2 are used for the torch bar within the game and handle what frame we are currently at in regards to the torch bar. After we have completed the logic we remove the fuel from the game.

```
this.leverCollision = function(player, lever)
{
    lever.loadTexture('lever_spr2');
    for(var set in walls.children){
        if(walls.children[set].wallState > 0)
        {
            if(lever.state == walls.children[set].wallState)
                walls.children[set].kill();
        }
    }
}

this.enemyCollision = function(){
    game.state.start('dead');
}

this.doorCollision = function(player, door)
{
    nextLevel();
}

this.itemTouched = function(pointer)
{
    return true;
}
```

These are the last few functions within the player class the leverCollision function handle's what happens when the player overlap's with a lever. We change the lever's sprite to show that the lever has been pulled, then we iterate over each wall and check to see if the wallState is anything other than 0, if it is, then that wall is a gate. We then check to see if the lever state and the gate's state match if they do then we destroy those gates so that the player can progress. The enemyCollision function kill's the player and loads the dead state. DoorCollision loads the next level, itemTouched is actually not an accurate name for the function, what this does is it detects if the point has pressed anywhere on the screen and return's true if it has.

Wall.js

```
/**
 * Created by Kyle Tuckey on 05/11/2016.
 */
var Wall = function(w_x, w_y, state){
    this.state = state;

    if(this.state == 0)
        this.sprite = game.add.sprite(w_x, w_y, 'wall_spr');
    else
        this.sprite = game.add.sprite(w_x, w_y, 'gate_spr');

    this.sprite.anchor.setTo(0, 0);
    game.physics.enable(this.sprite, Phaser.Physics.ARCADE);
    this.sprite.body.immovable = true;
    this.sprite.wallState = state;
}
```

The wall object has a rather simple implementation. The wall has 3 properties, a sprite, a state and a wallstate, state is parsed in to determine the wall's sprite, if it is a standard wall the state will be 0, otherwise it is a gate. The class also enable arcade physics on the objects sprite.

Waypoint.js

```
/**
 * Created by Kyle Tuckey on 23/11/2016.
 */
var WayPoint = function(w_x, w_y, file, id){
    this.sprite = game.add.sprite(w_x, w_y, file);
    this.sprite.id = id;
};
```

The waypoint object has the simplest implementation of the objects. The waypoint is used in the AI pathing for the enemy's and all it contains is a sprite (which is invisible) and a unique ID. The waypoint's have ID's so that the enemies can distinguish each waypoint from each other.

Lever.js

```
/**
 * Created by Kyle Tuckey on 16/12/2016.
 */
var Lever = function(l_x, l_y, file, state){
    this.sprite = game.add.sprite(l_x, l_y, file);
    this.sprite.state = state;
    game.physics.enable(this.sprite, Phaser.Physics.ARCADE);
    this.sprite.body.immovable = true;
}
```

The lever implementation is also very simple, It does not do anything special on its own within its own file, we define a sprite for the lever and enable physics on the object.

Fuel.js

```
/**
 * Created by Kyle Tuckey on 05/11/2016.
 */
var Fuel = function(f_x, f_y, file){
    this.sprite = game.add.sprite(f_x, f_y, file);
    this.sprite.anchor.setTo(0, 0);
    game.physics.enable(this.sprite, Phaser.Physics.ARCADE);
};
```

Like the lever implementation the fuel object is just a sprite with arcade physics enabled on it.

Torch.js

The torch file is one of if not the most complicated object within the game; the reason for this is because it is the game's core mechanic.

```
var Torch = function(player)
{
    this.torchPower = 100;
    this.circleRadius = game.add.sprite(player.sprite.x - player.sprite.width / 2, player.sprite.y, 'placeHolder_spr');
    game.physics.enable(this.circleRadius, Phaser.Physics.ARCADE);
    this.circleRadius.anchor.setTo(0.5, 0.5);
    this.circleRadius.body.setCircle(30);
    this.circleRadius.body.x = this.circleRadius.x - this.circleRadius.width / 2;
    this.circleRadius.body.y = this.circleRadius.y - this.circleRadius.height / 2;

    this.bitmap = game.add.bitmapData(game.world.width, game.world.height);
    this.bitmap.context.fillStyle = 'rgb(255, 255, 255)';
    this.bitmap.context.strokeStyle = 'rgb(255, 255, 255)';


    this.barBitmap = game.add.bitmapData(game.world.width, game.world.height);
    this.barBitmapImage = game.add.image(0, 0, this.barBitmap);
    this.barBitmapImage.visible = true;

    this.lightBitmap = game.add.image(0, 0, this.bitmap);
    this.lightBitmap.blendMode = Phaser.blendModes.MULTIPLY;
```

The torch has a few properties that it requires to function correctly. The torchPower property holds the torch's power level this value decreases overtime as the game goes on. The next property is the circleRadius property, this is a circle that is used to represent the area of the light and it also scales with the light radius as it decreases and increases. The bitmap property is a bitmap image that covers the entire game world to make it look dark. The lightBitmap is another image we layer on top of the world bitmap to render our light in.

```
this.update = function(player, walls)
{
    this.torchPower -= (2.0/24.0);

    var decreaseTorch = Math.round(2.5 * this.torchPower);
    var outerShadow = Math.round(decreaseTorch/10);

    // Next, fill the entire light bitmap with a dark shadow color.
    this.bitmap.context.fillStyle = 'rgb('+outerShadow+','+outerShadow+','+outerShadow+')';
    //bitmap.context.fillStyle = 'rgb(255,255,255)';
    this.bitmap.context.fillRect(0, 0, game.world.width, game.world.height);

    this.circleRadius.body.setCircle(decreaseTorch);

    this.circleRadius.body.x = player.sprite.x  - decreaseTorch;
    this.circleRadius.body.y = player.sprite.y  - decreaseTorch;

    var points = [];
    for(var a = 0; a < Math.PI * 2; a += Math.PI/360) {
        // Create a ray from the light to a point on the circle
        var ray = new Phaser.Line(player.sprite.x + (player.sprite.width/2 - 0.5), player.sprite.y + (player.sprite.height/2
            player.sprite.x + Math.cos(a) * decreaseTorch, player.sprite.y + Math.sin(a) * decreaseTorch);

        // Check if the ray intersected any walls
        var intersect = this.getWallIntersection(ray, walls);

        // Save the intersection or the end of the ray
        if (intersect) {
            points.push(intersect);
        } else {
            points.push(ray.end);
        }
    }
```

This is the first half of the torch's update function. The calculation's that take place within this file involve rendering our light rays and gradually decreasing the light radius and its intensity. it also handle's the intensity of the shadow's around the level.

```
        this.bitmap.context.beginPath();

        this.bitmap.context.fillStyle = 'rgb('+decreaseTorch+','+decreaseTorch+','+decreaseTorch+')';

        if(this.torchPower <= 0){
            game.state.start('dead');
        }
        this.bitmap.context.moveTo(points[0].x, points[0].y);
        for(var i = 0; i < points.length; i++) {
            this.bitmap.context.lineTo(points[i].x, points[i].y);
        }
        this.bitmap.context.closePath();
        this.bitmap.context.fill();
        // This just tells the engine it should update the texture cache
        this.bitmap.dirty = true;
```

The second half of the update function is about opening up a path for our bitmap context and actually using the calculations in the first half to fill the context with the light radius. Other calculations occur here as well such as detecting if the torch power has hit 0.

```
this.getWallIntersection = function(ray, walls) {
    var distanceToWall = Number.POSITIVE_INFINITY;
    var closestIntersection = null;

    walls.forEach(function (wall) {
        // Create an array of lines that represent the four edges of each wall
        if (game.physics.arcade.overlap(this.circleRadius, wall, null, null, null)) {
            if (wall.wallState == 0) {
                var lines = [
                    new Phaser.Line(wall.x, wall.y, wall.x + wall.width, wall.y),
                    new Phaser.Line(wall.x, wall.y, wall.x, wall.y + wall.height),
                    new Phaser.Line(wall.x + wall.width, wall.y,
                        wall.x + wall.width, wall.y + wall.height),
                    new Phaser.Line(wall.x, wall.y + wall.height,
                        wall.x + wall.width, wall.y + wall.height)
                ];

                for (var i = 0; i < lines.length; i++) {
                    var intersect = Phaser.Line.intersects(ray, lines[i]);
                    if (intersect) {
                        // Find the closest intersection
                        distance =
                            game.math.distance(ray.start.x, ray.start.y, intersect.x, intersect.y);
                        if (distance < distanceToWall) {
                            distanceToWall = distance;
                            closestIntersection = intersect;
                        }
                    }
                }
            }

        }

    }, this);

    return closestIntersection;
}
```

This function is how we check to see if there are any intersections between a ray and the walls within the environment. The function iterates over a collection of wall's and check's to see if any of the walls are within the radius of our light, it then check's to make sure the wall is in fact a wall and not a gate. Once it has done this it creates an array of line's that represent the 4 edges of each wall, after the edges have been stored we iterate over the lines collection and check to see if the ray is intersecting with any of the edge's if it is we then need to calculate the distance between the ray's starting point and where it is intersecting, we then check to see if this is less than the previous distanceToWall value, if it is we store the distance variable as its new value and then store the intersection, we do this until we have found the closest intersection to the player and return it.

Enemy.js

The enemy class has a lot of AI logic in it so it's a pretty big file.

```
var Enemy = function(e_x, e_y, file, path) {
    this.sprite = game.add.sprite(e_x, e_y, file);
    this.sprite.anchor.setTo(0, 0);
    this.sprite.animations.add('walk');
    this.sprite.animations.play('walk', 3, true);
    game.physics.enable(this.sprite, Phaser.Physics.ARCADE);
    this.sprite.state = 0;
    this.sprite.path = path;
    this.sprite.nextPoint = this.sprite.path[0];
    this.sprite.pos = 1;
```

The enemy class uses a sprite sheet, so its sprite is animated at a rate of 3 frames a second. The state property is to determine what behaviour state the NPC is currently in, the path property is specified by one of the parameter's that are parsed in that determine it's movement path. The nextPoint property holds the next waypoint the enemy need's to travel to. The pos property tells us what waypoint he enemy is currently at.

```
this.sprite.enUpdate = function(waypoints, player, intercept)
{
    if(this.state == 1) {
        waypoints.forEach(function (wayPoint) {

            var movementX = wayPoint.x - this.x;
            var movementY = wayPoint.y - this.y;

            //Normalise the movement

            var toWaypointLength = Math.sqrt(movementX * movementX + movementY * movementY);
            if(toWaypointLength > 0){
                movementX = movementX / toWaypointLength;
                movementY = movementY / toWaypointLength;
            }

            if (this.nextPoint > this.path[0]) {
                if (wayPoint.id == this.nextPoint) {

                    this.body.x += movementX * 2;
                    this.body.y += movementY * 2;

                    if (wayPoint.x - this.x < 1 && wayPoint.y - this.y < 1 && wayPoint.x - this.x > -1 && wayPoint.y - this.y > -1) {
                        this.x = wayPoint.x;
                        this.y = wayPoint.y;
                    }
                }
                if (this.x == wayPoint.x && this.y == wayPoint.y) {

                    this.nextPoint++;
                    this.pos++;
                }
                if (this.pos > this.path.length){
                    this.pos = 1;
                    this.nextPoint = this.path[0];
                }

            }
```

In this function we check to see if the current state of the enemy is 1, this is the roaming state. In this behaviour state we work out the vector between the enemy's position and the waypoint's position. Once we have worked out the movement value we need to check to see if the nextPoint we need to travel to is greater than the first value in our path property, if it is then we know we are moving to a new point, so we then need to check if the waypoint ID match's our nextPoint value, if it does then we have successfully identified the next waypoint the enemy needs to travel to. After we have confirmed the next point the enemy has to move to we increase its position by the movement value's we worked out earlier. The following if statement is to make sure we do not have a floating point value that prevents the enemy from registering that he has reached the correct waypoint, if the value is something like 1.01 or 0.09 then we want to force it to match the waypoint's position. Once it has reached the waypoint we increment the nextPoint we have to travel to and the enemy's current position, we then check to see if our position has exceeded the number of point's in the path that has been assigned to the enemy, if it has then the enemy has successfully made it to every point in its path, so we reset it's position to 1 and set the nextPoint to be the start of the path.

```
                else {
                    if (wayPoint.id == this.path[0]) {
                        this.body.x += movementX * 2;
                        this.body.y += movementY * 2;


                        if (this.x - wayPoint.x < 1 && this.y - wayPoint.y < 1) {
                            this.x = wayPoint.x;
                            this.y = wayPoint.y;
                        }

                         if (wayPoint.x - this.x < 1 && wayPoint.y - this.y < 1 && wayPoint.x - this.x > -1 && wayPoint.y - this.y > -1)
                            this.x = wayPoint.x;
                            this.y = wayPoint.y;
                         }

                        if (this.x == wayPoint.x && this.y == wayPoint.y) {
                            this.nextPoint++;
                            this.pos++;
                        }
                    }
                }
            }, this);
        }
        else
        {
            if(!intercept) {
                var movementX = player.x - this.x;
                var movementY = player.y - this.y;

                var toWaypointLength = Math.sqrt(movementX * movementX + movementY * movementY);
                movementX = movementX / toWaypointLength;
                movementY = movementY / toWaypointLength;

                this.body.x += movementX * 2;
                this.body.y += movementY * 2;
            }
        }
```

If the nextPoint value is not greater than the id of the first waypoint in our path then we have yet to reach the first point in our path. So we check to see if the waypoint matches the start of the path, if it does then the waypoint is our enemy's first waypoint to travel to. The logic in this section is pretty much identical to the earlier implementation. The other else statement in this image detail's what happens when the behaviour state is not 1, in which case the enemy is chasing the player. All we do is check to see if the intercept bool is false, if it is then there are no wall's between the player and the enemy. Then we work out a movement vector between the enemy and the player, and make the enemy move towards the player.

Level.js

```
function levelLoader() {
//read in the level's JSON file and grab each elements data
    if(JSON.parse(game.cache.getText('level' + level))){
        var text = JSON.parse(game.cache.getText('level' + level));
        var wallData = text.wallData;
        var fuelData = text.fuelData;
        var wayPointData = text.wayPointData;
        var enemyData = text.enemyData;
        var leverData = text.leverData;

        currentPower = 100;
        sheetCount = 0;

        walls = game.add.group();
        fuel = game.add.group();
        waypoints = game.add.group();
        enemys = game.add.group();
        levers = game.add.group();
        //create a new instance of our Player class
        player = new Player(text.playerStart.x, text.playerStart.y, 'player_spr');
        //enemy = new Enemy(text.enemyStart.x, text.enemyStart.y, 'enemy_spr', [1, 2, 3]);

        door = game.add.sprite(text.doorPos.x, text.doorPos.y, 'door_spr');
        game.physics.enable(door, Phaser.Physics.ARCADE);

        //generate each enemy using the JSON data and add it to the enemys group
        if(typeof enemyData !== 'undefined' && enemyData.length > 0) {
            for (var e in enemyData) {
                enemys.add(new Enemy(enemyData[e].x, enemyData[e].y, 'enemy_spr', enemyData[e].path).sprite)
            }
        }

        //genereate waypoints using the JSON data and add it to the waypoints group
        if(typeof wayPointData !== 'undefined' && wayPointData.length > 0) {
            for (var w in wayPointData) {
                waypoints.add(new WayPoint(wayPointData[w].x, wayPointData[w].y, 'placeHolder_spr', wayPointData[w].id).sprite);
            }
        }
```

The first function in the level.js file is the levelLoader the above image shows the first half of this function where it read's in a JSON file, which contains our current level. We then create the phaser game groups for the game and begin populating these object group's with the data in the levelLoader file.

```
        //generate walls using the JSON data and add it to the walls group
        if(typeof wallData !== 'undefined' && wallData.length > 0) {
            for (var i in wallData) {
                walls.add(new Wall(wallData[i].x, wallData[i].y, wallData[i].state).sprite);
            }
        }

        //generate fuel
        if(typeof fuelData !== 'undefined' && fuelData.length > 0) {
            for (var f in fuelData) {
                fuel.add(new Fuel(fuelData[f].x, fuelData[f].y, 'fuel_spr').sprite)
            }
        }

        //initialise the torch
        torch = new Torch(player);
    }
    else
    {
        level = 1;
        game.state.start('end');
    }
}

function nextLevel() {
    walls = [];
    fuel = [];
    waypoints = [];
    level++
    game.world.removeAll();
    levelLoader();
}
```

The second half of the level.js file shows the rest of the game object's being read in by the levelloader and the next level function, which when called empty's the object array's and clear's the game world before incrementing the level counter and reading in a new level.

Play.js

```
var player;
var walls;
var levers;
var fuel;
var waypoints;
var enemys;
var torch;
var barSprite;
var door;
var level = 1;
var sheetCount = 0;
var currentPower = 100;
var count = 0;

var playState = {
    create: function () {
        game.time.advancedTiming = true;
        game.stage.backgroundColor = '#626A72';
        levelLoader();
    },
```

The first part of the play file shows the global variables in the game as well as the create function, the create function does not do much except set the background colour, and load the level.

```
update: function () {
    player.update();
    torch.update(player, walls);

    if(Math.trunc(torch.torchPower) === currentPower-10)
    {
        currentPower = Math.trunc(torch.torchPower);
        sheetCount++
    }

    enemys.forEach(function (enemy) {
        if (game.physics.arcade.overlap(torch.circleRadius, enemy, null, null)) {
            var ray = new Phaser.Line(enemy.x + enemy.width / 2, enemy.y + enemy.height / 2, player.sprite.x, player.sprite.y);
            var intercept = torch.getWallIntersection(ray, walls);
            if (!intercept)
                enemy.state = 2;
            else
                enemy.state = 1;
        }
        else {
            enemy.state = 1;

        }
        enemy.enUpdate(waypoints, player.sprite, false);
    });

    game.physics.arcade.collide(player.sprite, walls, null, null, this);
    game.physics.arcade.overlap(player.sprite, fuel, player.fuelCollision, null, null);
    game.physics.arcade.overlap(player.sprite, levers, player.leverCollision, null, null);
    game.physics.arcade.collide(player.sprite, door, player.doorCollision, null, null);
    game.physics.arcade.collide(player.sprite, enemys, player.enemyCollision, null, this);
},

render: function () {
    //game.debug.body(torch.circleRadius);
    //game.debug.text(game.time.fps || '--', 2, 14, "#00ff00");    }
    barSprite = game.add.sprite(game.world.width / 14, 5, 'torchBar');
    barSprite.frame = sheetCount;
}
```

The second half of the play file shows the update function which handles the various update function's for our objects as well as the enemy's line of sight ray, and the various collision's between game objects. It also shows the render function which adds the torchbar sprite and set's the frame to sheetcount.

Dead.js

```
var deadState = {
    create: function(){
        game.stage.backgroundColor = '#000000';
        dead_label = game.add.text(120, GAMEHEIGHT/3, 'The dungeon

        var button = game.add.button(GAMEWIDTH/2, GAMEHEIGHT/2, 'c

    },

    actionOnClick: function(){
        game.state.start('play');
    }
}
```

This file is called when the player has died; this file loads the death screen for the player.

End.js

```
var endState = {
    create: function(){
        game.stage.backgroundColor = '#000000';
        dead_label = game.add.text(120, GAMEHEIGHT/3, 'You

        var button = game.add.button(GAMEWIDTH/2, GAMEHEIGH

    },

    actionOnClick: function(){
        game.state.start('menu');
    }
}
```

This file is called when the player successfully make's it past the 2 level's in the game.

Assets

The assets that are present in the game were all created by me.

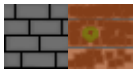
Player Sprite

The player sheet is made up of 7 frames; these frames don't differ greatly except for the flame of the torch.

Enemy Sprite

The enemy sheet is more complicated than the player sheet, it has fewer frame's but there is more going on with it such as its eyes and mouth moving as well as it's tentacle's there is even some transparency between frame's to give it a spectral feel.

Wall Sprite



The wall sprite is very simple; it is designed in a way so that it naturally fits with other wall sprites in the scene to give it a tile effect.

Door Sprite

The door sprite is a simple set of stairs.

Fuel Sprite

The fuel sprite is a torch.

Continue Sprite

This sprite appears on both the death and end screen's it's made up of 2 frames.

Play Sprite

The play button you see in the menu.

Research

A lot of research time went into looking for an appropriate lighting method for the game. The torch is the most essential piece of the game itself, so the lighting needed to do more than just be a semi-transparent image masked onto the scene. Eventually the lighting method that showed the most potential was the use of ray casting the light rays. I looked at various implementations of how this could be achieved but the effective method I could find was a tutorial on casting shadow's with ray tracing in phaser (Gamemechanicexplorer.com, 2017).

Although his implementation achieved exactly what I wanted, it was not efficient, so I optimised the code by implementing a circular radius around the light. What the radius does is it limits the amount of wall's that are being checked for intersections to only the light radius. I also added the dimming of the torch and its intensity. After these small optimisations to the implementation I ended up with what I have now for the torch. Any other time I needed help with some aspect of the game I typically went to the Phaser website for help, the documentation surrounding the library itself and how to implement certain functionality is vast and more often than not had the answer I was looking for.

Critical Review

Three good things about the design and implementation

Simplest Design

I think the fact that the design for the game was as simple as it was, ended up being a benefit ot the game's development. There was not too much for the game to be overly ambitious, but enough for a solid game idea to be implemented. It also meant that there was more time to focus on the important parts of the game such as the AI and the torch.

Well Organised Code

The code structure of the game is well organised, overall functionality of the game is where you would expect it to be, with a few exceptions. This makes it more readable and understandable and also sped up the development time of the project.

It feels like a game

I think the best thing about the game is the fact it feel's and play's like an actual game. There is a sense of challenge to it, the kind that make's the player want to keep trying, the frustration of being caught by the ghost's, trying to find a way to the end.

Three Improvement that could be made

Mobile Performance is poor

Although the game is playable on a mobile the performance is very poor. This is because the CPU demand of the torch's ray casting is too large for the mobile device to handle. The reason the demand is so high is because the amount of ray's being rendered to screen is technically infinite, there is not a specific number of rays being rendered. So by having a set number of rays be rendered we can reduce the CPU cost therefore improving performance. There is also the fact that even with the optimisation's I made to reduce the intersection checking with walls' and the cast ray's there are still wall's that are being checked that don't need to be, so this is another potential area for improvement.

No scoring system or reason to replay the game

Having a simple design is a good thing; however there is not a lot in this game that enhances the need for competition. There is no timer, so you can't try to complete the game as fast as possible and there is no scoring system, so you can't compete for a high score with other people. This means that once the game has been completed there is no longer any real need to play the game. A scoring system could enhance the gaming experience; a difficulty system would also be a decent approach to this kind of game.

The game is too short

The game is 2 short level's long which is not a particularly long or engaging experience. The reason why the game is so short is due to the level system in the game. All the JSON file's for the levels were created by hand, I manually input every sprite and coordinate in the file. This was a long and tedious process that was taking up too much development time. A way this could be changed would be to implement something that makes use of Phaser's tile map system. This would mean that I could create a map in tilemap software and export the JSON file it generates and instead read that file in.

This would in turn make the level creation process a lot faster, it would also enable other people ot create a level for the game as well.

References

Gamemechanicexplorer.com. (2017). Game Mechanic Explorer. [online] Available at: https://gamemechanicexplorer.com/#raycasting-2 [Accessed 10 Jan. 2017].

Emanuele Feronato. (2017). Phaser Tutorial: understanding Phaser states. [online] Available at: http://www.emanueleferonato.com/2014/08/28/phaser-tutorial-understanding-phaser-states/ [Accessed 10 Jan. 2017].

Appendix A: Installation Instruction's

You can play the game at https://malithium.github.io/Torch-Run/

Appendix B: Game Concept Presentation's