# Ninja Stealth Game

*By Joshua Petherick*

*Ci328 Internet game design and development*

## Table of Contents

## Game Summary

### Objectives

NSG (Ninja Stealth Game) is a stealth-oriented platformer where the players goal is to reach to the door at the end of every level. The player is, essentially, climbing up the many levels of the building in the hopes of reaching the important Intel at the top floor. Once the player has collected the Intel they must then make their way back down to the bottom floor to escape the building. The goal is to do all of this in as short a time as possible.

### Rules

The rules are simple and should be simple for the player to grasp. Please see the list of them below:

1. Player can move Left, Right and Jump.
2. Ladders allow the player to climb up or down.
3. Dark spots make the player invisible to guards' vision however if player is spotted in lighted areas then guards will pursue.
4. Player must NOT be caught by guards. Being caught will reset the player to the last checkpoint.
5. High score will be earned by time-taken to complete the game (e.g. 5:12).
6. Progressing to the next level will create a new checkpoint.

### Game play

Players interact with the game using the on-screen arrows found at the bottom of the screen. Each arrow follows the rules seen above, moving the player left or right and the up and down arrows moving the player up and down the ladder.

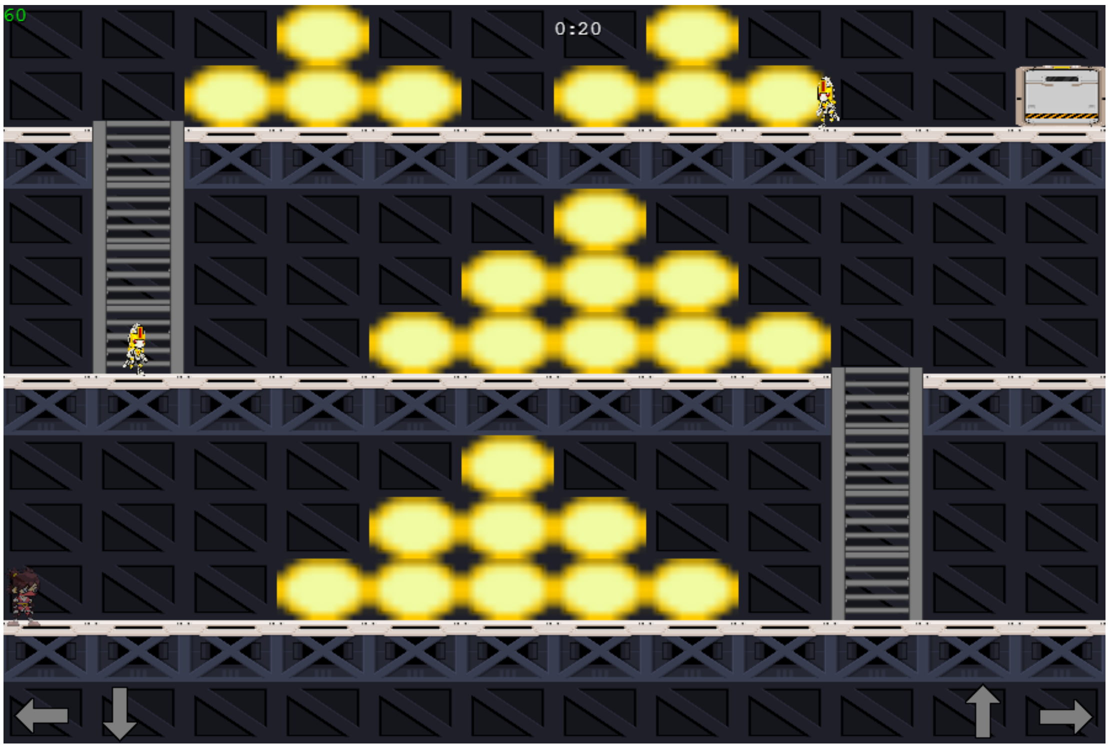


*Figure 1. Screenshot of Level 1*

## Game Structure

This section will present the different screens which the user will see.

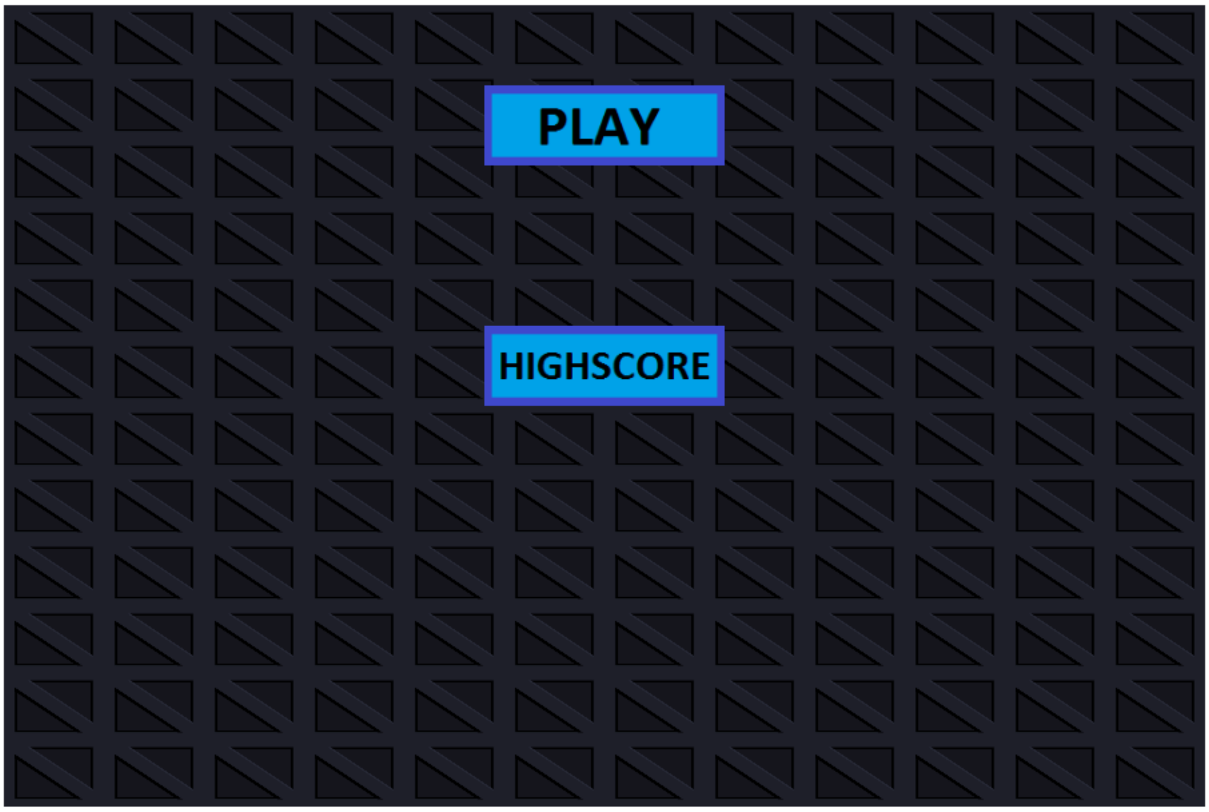


*Figure 2. The main menu*

Shows the main menu.

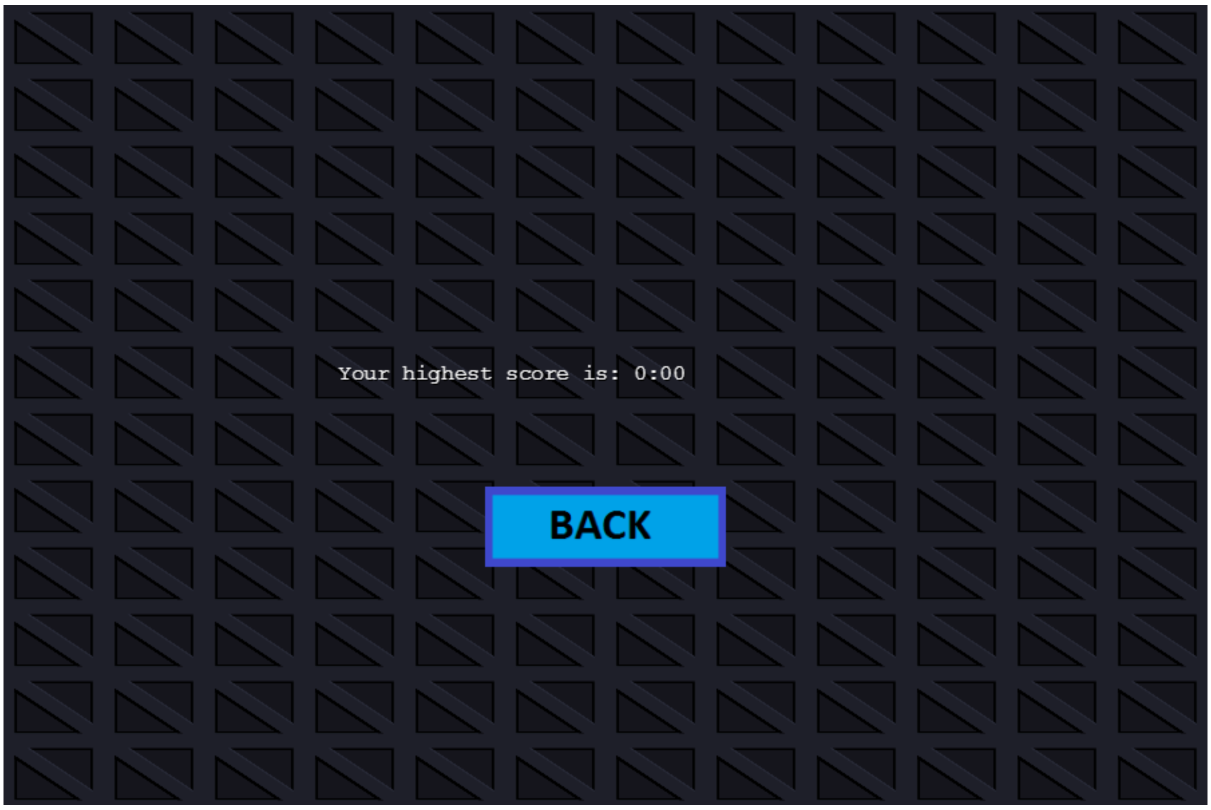


*Figure 3. The Highscore display menu*

Shows the High score screen.

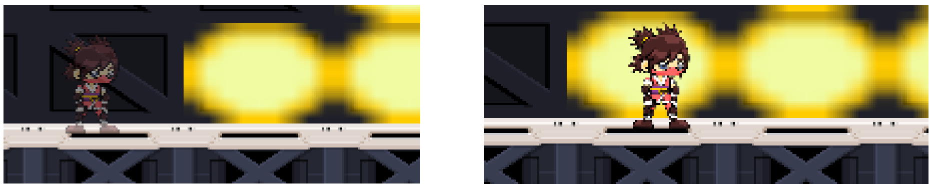


*Figure 4. & 5. Players visibility indicator*

Left image shows shadow. Right image shows in light.

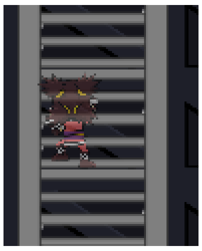


*Figure 6. Player climbing a ladder*

Casually climbing up the ladder.

## Implementation Specification

This section will go through the design, structure and implementation of the code used to create NSG. It will also briefly discuss the used of assets.

## Code Structure

I'll start by going through each of the classes that are part of this project, discussing their purpose and the parameters they hold.

### NSG

This file contains the body of the game, and is responsible for starting and managing the games flow. It holds all the global variables (i.e. GAMEHEIGHT, GAMEWIDTH, Level, etc) and even the game object itself (see below).

```
var game = new Phaser.Game(GAMEWIDTH, GAMEHEIGHT, Phaser.AUTO, 'Ninja Stealth
Game', {
    preload: preload,
    create: create,
    update: update,
    render: render
});
```

The first function to be run is the preload function which maintains loading all the assets that are to be used within the game. I saw this as the initialiser of the game and therefore put in other bits of code which I only wanted to be called once, such as changing the screen size and loading the image groups.

```
function preload() {
    if (window.screen.availHeight > (GAMEHEIGHT*2) && window.screen.availWidth >
(GAMEWIDTH*2)) {
        // Adjust for bigger screens
        GAMEHEIGHT = GAMEHEIGHT*2;
        GAMEWIDTH = GAMEWIDTH*2;
        game.scale.setGameSize(GAMEWIDTH, GAMEHEIGHT)
    }
    console.log('Phaser Version: ' + Phaser.VERSION);
    console.log('B3 Version: ' + b3.VERSION);
    // Start state
    gameState = gameStates.MENU;
    var ASSETPATH = 'assets/images/';
    var SOUNDPATH = 'assets/sounds/';
    var TREEPATH = 'assets/behaviourTrees/';
    // XML Files
    game.load.atlasXML('player', ASSETPATH + 'Player/playerSpriteSheet.png',
ASSETPATH + 'Player/playerSpriteSheet.xml');
    game.load.atlasXML('enemy', ASSETPATH + 'Enemy/enemySpriteSheet.png', ASSETPATH
+ 'Enemy/enemySpriteSheet.xml');
    // Images
    game.load.image('background', ASSETPATH + 'Background/BGTile.png');
    game.load.image('floor', ASSETPATH + 'Background/Floor.png');
    game.load.image('wall', ASSETPATH + 'Background/Wall.png');
    game.load.image('light', ASSETPATH + 'TestImages/light.png');
    game.load.image('stairs', ASSETPATH + 'Background/Ladder.png');
    game.load.image('exit', ASSETPATH + 'Background/Door.png');
    game.load.image('intel', ASSETPATH + 'Background/Intel.png');
    // Buttons
    game.load.image('leftArrow', ASSETPATH + 'Buttons/ArrowLeft.png');
    game.load.image('rightArrow', ASSETPATH + 'Buttons/ArrowRight.png');
    game.load.image('downArrow', ASSETPATH + 'Buttons/ArrowDown.png');
    game.load.image('upArrow', ASSETPATH + 'Buttons/ArrowUp.png');
    // Spritesheets
    game.load.spritesheet('buttonPlay', ASSETPATH + 'Buttons/buttonPlay.png', 194,
66);
    game.load.spritesheet('buttonHighscore', ASSETPATH +
'Buttons/buttonHighscore.png', 194, 66);
    game.load.spritesheet('buttonBack', ASSETPATH + 'Buttons/buttonBack.png', 194,
66);
    // Audio
    game.load.audio('background1', SOUNDPATH + 'background_eerie.mp3');
    game.load.audio('background2', SOUNDPATH + 'background_epic.wav');
    game.load.audio('steps', SOUNDPATH + 'steps.wav');
    game.load.audio('alert', SOUNDPATH + 'robot_intruder.wav');
    // Text
    var maxLvls = 15;
    for (var i = 1; i <= maxLvls; i++) {
        // Load all level text files!
        game.load.text('level' + i, 'assets/levels/lvl' + i + '.txt')
    }
    game.load.text('AITree', TREEPATH + 'aiTree.json');
    // Init Groups
    TILEBACKGROUND = game.add.group();
    background = game.add.group();
    wallLayer = game.add.group();
    lightLayer = game.add.group();
    stairLayer = game.add.group();
    exitLayer = game.add.group();
    foreground = game.add.group();
    // Used for FPS counter
    game.time.advancedTiming = true;
} //preload();
```

The next function to be run is the create method, which I saw as the “state loader” meaning that it loads all the objects and assets required when loading or changing the gameState. I used a switch statement so it would only load certain stuff based on what state the game was in. For example, if the game state was MENU then it would load the squared background and the two buttons (See fig. 1).

```
function create() {
    //  Will create objects for the game
    switch(gameState)
    {
        case gameStates.MENU:
            // Load game background!
            var text = prepLevel();

            TileSizeY = Math.round(GAMEHEIGHT/text.length);
            TileSizeX = Math.round(GAMEWIDTH/(text[0].length));
            loadLevel(text); // Will only load the background!

            buttons.push(new button("PLAY"));
            buttons.push(new button("HIGHSCORE"));
            break;

        case gameStates.SCORE:
            buttons.push(new button("BACK"));
            break;

        case gameStates.PLAY:
            game.physics.startSystem(Phaser.Physics.ARCADE); // ARCADE physics as
fits game best
            game.physics.arcade.gravity.y = 350;
            timer = game.time.create(false);

            playerDied = new Signal();
            newLevel = new Signal();
            getIntel = new Signal();

            newLevel.addSignal(nextLevel);
            getIntel.addSignal (function() {
                exit = new Exit(player.origX, player.origY);
                backgroundMusic.queueSong('background2');
            });

            level = 1; // Reset level for every create!
            var text = prepLevel();
            loadLevel(text);

            backgroundMusic = new sound('background1'); // Background music, open
to update
            backgroundMusic.musicVol(0.75);
            backgroundMusic.musicLoop();

            timer.start(); // Timer to begin starts last!
            break;
    }
} // create()
```

After calling create the game would now go into the self-described “running” state, meaning it would loop the update and render functions until the game is over. The update function manages logical updates to the game such as player inputs or updating the enemy positions.

```
function update() {
    //  Change game states and call update for all objects
    switch(gameState) {
        case gameStates.PLAY:
            player.playerInput(); // Check input
            player.playerUpdate(); // Update Player
            handleCollision(); // Handle all collisions
            for(e in enemies) {
                enemies[e].enemyUpdate(); // Update enemy AI, collision, etc
            }
            backgroundMusic.musicUpdate();
            break;
    }
} // update()
```

The render function handles drawing text onto the screen, such as the timer and the FPS.

```
function render() {
    //game.debug.text.clean;
    switch(gameState) {
        case gameStates.SCORE:
            var score = localStorage.getItem('timerScore');
            if (!score) { score = '0:00'; } // If null then add value!
            game.debug.text('Your highest score is: ' + score, (GAMEWIDTH/2)-200,
(GAMEHEIGHT/2)-20); // Prints Timer
            break;

        case gameStates.PLAY:
            game.debug.text(game.time.fps || '--', 2, 14, '#00ff00'); // Prints FPS
            game.debug.text(sortTimer(timer.seconds), GAMEWIDTH / 2, 25); // Prints
Timer
            break;
    }
} // render()
```

Next I’ll discuss the two methods responsible for loading in the different levels within the game.

The “loadLevel” function reads in an array of text, generated by the .txt files found within the assets/levels folder to determine the position of objects. It reads the array line-by-line and then passes the character to the “addObject” method.

```
function loadLevel(text) {
    // For each Line in Text  -  Determines Y
    for (i = 0; i < text.length; i++) {
        // For each Character in Line  -  Determines X
        for (j = 0; j < text[i].length; j++) {
            // Initialise based on character in txt file
            var x = (j*TileSizeX); // X based on position in txt file
            var y = (i*TileSizeY); // Y based on position in txt file
            // Designed so will only load background when in MENU state
            if (gameState === gameStates.MENU) {
                new BGTile(x, y);
            }
            else {
                addObject(text[i].charAt(j), x, y);
            }
        }
    }
}
```
The “addObject” method is responsible for loading an object based on the character type passed across from the “loadLevel” method. I tried to redo this function several times, so it looked less messy, but ultimately this was the best way of managing and loading each new object.

```
function addObject (char, x, y) {
    switch (char) {
        case "P":
            player = new Player(x, y);
            break;

        case "G":
            enemies.push(new Enemy(x, y)); // Add new to Array
            break;

        case "F":
            new Floor(x, y); // Add new to Array
            break;

        case "W":
            new Wall(x, y); // Add new to Array
            break;

        case "S":
            new Stair(x, y); // Add new to Array
            break;

        case "L":
            new Light(x, y); // Add new to Array
            break;

        case "X":
            enemies.push(new Enemy(x, y)); // Add new to Array
            new Light(x, y); // Add new to Array
            break;

        case "Y":
            player = new Player(x, y);
            new Stair(x, y); // Add new to Array
            break;

        case "Z":
            player = new Player(x, y);
            new Light(x, y); // Add new to Array
            break;

        case "I":
            intel = new Intel(x, y);
            new Light(x, y); // Add new to Array
            break;

        case "E":
            exit = new Exit(x, y);
            break;
    }
}
```
The next method is designed to take the value of the timer and present it into the value that is seen on the screen.

```
function sortTimer(time) {
    var mins = 0;
    if (Math.round(time) >= 60) {
        mins = Math.floor(time/60);
    }
    var secs = Math.floor(time) - (mins*60);
    if (secs < 10) {
        secs = "0" + secs;
    }
    return mins + ":" + secs;
}
```

Next is our collision method, which handles collision for most of the objects in this game (Aside from the enemy class which has their own collision). It uses the phaser physics mechanic to determine whether something is colliding, but my personal methods for the results.

```
function handleCollision () {
    game.physics.arcade.collide(player.playerSprite, wallLayer); // Checks if
player is colliding with Walls
    for (e in enemies) {
        if (game.physics.arcade.overlap(player.playerSprite,
enemies[e].enemySprite)) {
            // For each enemy, check if overlapping, if so then reset level
            playerDied.call();
        }
    }
    if (game.physics.arcade.overlap(player.playerSprite, lightLayer)) {
        // If overlapping with LIGHT then change state
        player.updateState(player.playerStates.LIGHT);
    }
    else {
        player.updateState(player.playerStates.DARK);
    }
    if (game.physics.arcade.overlap(player.playerSprite, stairLayer)) {
        // If overlapping with STAIR then no gravity, so can climb up/down
        player.setGravity(false);
    }
    else {
        player.setGravity(true);
    }
    if (intel) {
        if (game.physics.arcade.overlap(player.playerSprite, intel.intelSprite)) {
            getIntel.call();
        }
    }
    if (game.physics.arcade.overlap(player.playerSprite, exitLayer)) {
        // Check if player has collidied with exit, if so progress to next level
        newLevel.call();
    }
}
```

The “nextLevel” method does precisely what the name suggests, and that is to prep everything for loading the next level. It does this by emptying all the variables and groups, incrementing level and then called “loadLevel”. It is also responsible for calling the “gameComplete” method.

```
function nextLevel() {
    // Empty arrays
    player = null;
    enemies = [];
    exit = null;
    // Empty phaser group
    background.removeAll();
    wallLayer.removeAll();
    lightLayer.removeAll();
    stairLayer.removeAll();
    foreground.removeAll();
    exitLayer.removeAll();
    // Load next level
    level++;
    if (level < 16 ) {
        var text = prepLevel();
        loadLevel(text);
    }
    else {
        gameComplete();
    }
}
```
The “gameComplete” method gets called once the player has completed level 15. It simply stops the timer, records the users score and then changes the gameState to SCORE (Meaning it will load the high score screen, see figure 2).

```
function gameComplete() {
    timer.pause(); // Pause
    if(!localStorage.getItem('timerScore') && localStorage.getItem('timerScore') >
timer.seconds) {
        localStorage.setItem('timerScore', sortTimer(timer.seconds)); // Store
local time score for player
    }
    timer.stop(); // Kill timer off
    gameState = gameStates.SCORE;
    create();
}
```
Lastly the “prepLevel” method gets the text from the text file, removes any line breaks and then returns it.

```
function prepLevel() {
    var txt = game.cache.getText('level' + level).split('\n'); // Stores it as an
array
    for(i = 0; i < txt.length; i++) {
        txt[i] = txt[i].replace(/\n|\r/g, ""); // Cleans up Line breaks
    }
    return txt;
}
```
Finally here is a list of all the global variables which are stored in the NSG file.

```
// Core-game variables
var GAMEHEIGHT = 300;
var GAMEWIDTH = 450;
var backgroundMusic;
var level = 1;
var timer;

// Phaser draw groups
var background;
var wallLayer;
var lightLayer;
var stairLayer;
var foreground;
var exitLayer;
var TILEBACKGROUND;

// Tile Info
var TileSizeX;
var TileSizeY;

// Sprite Variables
var player;
var enemies = []; // Array of Enemies
var buttons = []; // Array of Menu buttons
var intel;
var exit;

// State variables
var gameState;
var gameStates = {
    MENU: 0,
    PLAY: 1,
    SCORE: 2
};

// Signals
```

```
var playerDied; // https://phaser.io/docs/2.6.2/Phaser.Signal.html
var newLevel;
var getIntel;
```

Button

This object class handles all the buttons you can see on the Main Menu. The type gets passed during the creation of the button, which then goes into a switch statement. The difference between the three is the position and the method which gets called when the button is clicked. The buttons position is designed to change based on the size of the screen.

```
function button(type) {
    this.height = GAMEHEIGHT/10;
    this.width = GAMEWIDTH/5;
    switch(type) {
        case "PLAY":
            this.button = game.add.button(0 + this.width*2, 0+this.height,
'buttonPlay', playClick, this, 1, 0, 1);
            break;

        case "HIGHSCORE":
            this.button = game.add.button(0 + this.width*2, (GAMEHEIGHT/2)-
this.height, 'buttonHighscore', highscoreClick, this, 1, 0, 1);
            break;

        case "BACK":
            this.button = game.add.button(0 + this.width*2,
(GAMEHEIGHT/2)+this.height, 'buttonBack', backClick, this, 1, 0, 1);
            break;
    }
    this.button.width = this.width;
    this.button.height = this.height;
    background.add(this.button);

    this.destroyButton = function() {
       this.button.destroy();
    }
}
```

The remainder of the class holds the methods which got called when the button gets clicked on. Each method changes the gameState and then calls the create method from NSG.

```
function playClick() {
    clear();
    gameState = gameStates.PLAY;
    create();
}

function highscoreClick() {
    clear();
    gameState = gameStates.SCORE;
    create();
}

function backClick() {
    clear();
    gameState = gameStates.MENU;
    create();
}

function clear() {
    for (i in buttons) {
        buttons[i].destroyButton();
    }
    background.removeAll();
    buttons = [];
}
```

## BackgroundTile

This object class manages the squares in the background, which are loaded in from the “levelLoad” method. BGTile is a basic function which only requires an x and y position.

```
function BGTile(x, y) {
    this.tileSprite = game.add.sprite(x, y, 'background');
    this.tileSprite.width = TileSizeX;
    this.tileSprite.height = TileSizeY;
    TILEBACKGROUND.add(this.tileSprite);
}
```

## Player

This class handles the Player, and everything that the player does, such as movement, physics, animations and states. The first part is similar to all the other objects, except with enabling physics after creating the game sprite.

```
function Player(x, y) {
    // Init
    this.playerSprite = game.add.sprite(x, y, 'player');
    this.playerSprite.width = (TileSizeX/2);
    this.playerSprite.height = TileSizeY;
    this.gotIntel = false;
    this.origX = x; // Set origin so can restart level!
    this.origY = y;

    // Physics
    game.physics.enable(this.playerSprite, Phaser.Physics.ARCADE);
    this.playerSprite.body.collideWorldBounds = true; // Thou shall never leave
this world!
    this.playerSprite.body.mass = 0; // Remove mass so doesn't push other objects
down...
    this.speed = 100; // Set base speed here!
…

}
```

Next we move into the animations which include walking, climbing, jumping and idling. It works by calling the objects, which were initialised in the NSG.js, and stating the frame names and numbers. The line Phaser.Animation.generateFrameNames(‘Idle_’, 0, 9, ‘’, 3) is what sorts what images should be used, and what order, from the xml files.

```
// Animations
this.idleAnimation = this.playerSprite.animations.add('Idle',
Phaser.Animation.generateFrameNames('Idle_', 0, 9, '', 3));
this.idleAnimation.speed = 20

this.walkAnimation = this.playerSprite.animations.add('Walk',
Phaser.Animation.generateFrameNames('Run_', 0, 9, '', 3));
this.walkAnimation.speed = 20;

this.jumpAnimation = this.playerSprite.animations.add('Jump',
Phaser.Animation.generateFrameNames('Jump_', 0, 9, '', 3));
this.jumpAnimation.speed = 10;

this.climbAnimation = this.playerSprite.animations.add('Climb',
Phaser.Animation.generateFrameNames('Climb_', 0, 9, '', 3));
this.climbAnimation.speed = 10;
```

After the animation we look at input, which are four on screen buttons, and when they are pressed down they set the move Booleans to true, and sets them to false when no longer being pressed down.

```
// Handle button input
this.leftKey = false;
this.rightKey = false;
this.upKey = false;
this.downKey = false;

this.leftButton = game.add.button(10, GAMEHEIGHT-40,'leftArrow', null, this, 0, 1,
0, 1);
this.leftButton.events.onInputOut.add(function(){player.leftKey = false;});
this.leftButton.events.onInputDown.add(function(){player.leftKey = true;});
this.leftButton.events.onInputUp.add(function(){player.leftKey = false;});
foreground.add(this.leftButton); // Add to foreground layer so it's drawn last!

this.rightButton = game.add.button(GAMEWIDTH-60, GAMEHEIGHT-40, 'rightArrow', null,
this, 0, 1, 0, 1);;
this.rightButton.events.onInputOut.add(function(){player.rightKey = false;});
this.rightButton.events.onInputDown.add(function(){player.rightKey = true;});
this.rightButton.events.onInputUp.add(function(){player.rightKey = false;});
foreground.add(this.rightButton); // Add to foreground layer so it's drawn last!

this.upButton = game.add.button(GAMEWIDTH-120, GAMEHEIGHT-50, 'upArrow', null,
this, 0, 1, 0, 1);
this.upButton.events.onInputOut.add(function(){player.upKey = false;});
this.upButton.events.onInputDown.add(function(){player.upKey = true;});
this.upButton.events.onInputUp.add(function(){player.upKey = false;});
foreground.add(this.upButton); // Add to foreground layer so it's drawn last!

this.downButton = game.add.button(80, GAMEHEIGHT-50, 'downArrow', null, this, 0, 1,
0, 1);
this.downButton.events.onInputOut.add(function(){player.downKey = false;});
this.downButton.events.onInputDown.add(function(){player.downKey = true;});
this.downButton.events.onInputUp.add(function(){player.downKey = false;});
foreground.add(this.downButton); // Add to foreground layer so it's drawn last!
```

Now we add the players reactions when a Signal goes off, such as player dying or when collecting the Intel. These are straightforward thanks to the Signal object.

```
playerDied.addSignal(function () {
    player.playerSprite.x = player.origX;
    player.playerSprite.y = player.origY;
});

getIntel.addSignal(function () {
    player.gotIntel = true;
});
```

There are also a couple properties which need to be added, including the player state and the movement sounds.

```
this.playerStates = {
    DARK: 0,
    LIGHT: 1
};
this.state = this.playerStates.DARK;
foreground.add(this.playerSprite);

this.steps = new sound('steps');
this.steps.musicVol(2.0);
```

Finally we have the functions which get called to update the player, such as player Input or changing the state. The first function created is the input method, which works by checking the movement Booleans, and updating the position accordingly. There is also collision detection statements in this method, as this seemed the best way to check if the player could or could not move somewhere. It also stops and starts the required player animations.

```
this.playerInput = function() {
    this.playerSprite.body.velocity.x = 0; // Empty velocity so not sliding
left/right
    //this.playerSprite.scale.x *= -1;
    if (!this.playerSprite.body.allowGravity) {
        this.playerSprite.body.velocity.y = 0; // Empty velocity so not sliding
up/down
    }

    if (game.physics.arcade.collide(player.playerSprite, background)) {
        if (this.upKey) {
            this.playerSprite.body.velocity.y = -(this.speed+50); // Jump up
            this.stopAnimations();
            this.jumpAnimation.play();
        }
    }
    if (this.leftKey) {
        this.playerSprite.body.velocity.x = -this.speed; // Move left
        if (!this.walkAnimation.isPlaying && !this.jumpAnimation.isPlaying
            && this.playerSprite.body.allowGravity === true) {
            if (this.playerSprite.width > 0) { // Flip image vertically (Face Left)
                this.playerSprite.scale.x *= -1;
                this.playerSprite.x -= this.playerSprite.width; // Prevents
position from changing after flip
            }
            this.stopAnimations();
            this.walkAnimation.play();
            this.steps.musicLoop();
        }

    }
    if (this.rightKey) {
        this.playerSprite.body.velocity.x = this.speed; // Move right
        if (!this.walkAnimation.isPlaying && !this.jumpAnimation.isPlaying
            && this.playerSprite.body.allowGravity === true) {
            if (this.playerSprite.width < 0) { // Flip image vertically (Face
Right)
                this.playerSprite.scale.x *= -1;
                this.playerSprite.x -= this.playerSprite.width; // Prevents
position from changing after flip
            }
            this.stopAnimations();
            this.walkAnimation.play();
            this.steps.musicLoop();
        }

    }
    if (this.upKey) {
        if (!this.playerSprite.body.allowGravity) {
            this.playerSprite.body.velocity.y = -this.speed; // Move up
            if (!this.climbAnimation.isPlaying) {
                this.stopAnimations();
                this.climbAnimation.play();
            }
        }
    }
    if (this.downKey) {
        if (!this.playerSprite.body.allowGravity) {
            this.playerSprite.body.velocity.y = this.speed; // Move down
            if (!this.climbAnimation.isPlaying) {
                this.stopAnimations();
                this.climbAnimation.play();
            }
        }
    }
}
```

Next we have the playerUpdate function, which is primarily used for handling the player animations and deciding if the player is “idle”.

```
this.playerUpdate = function () {
    // Update
    if (this.leftKey === false && this.rightKey === false) {
        this.walkAnimation.stop();
    }
    if (!this.walkAnimation.isPlaying && !this.jumpAnimation.isPlaying &&
        !this.idleAnimation.isPlaying && !this.climbAnimation.isPlaying &&
        this.playerSprite.body.allowGravity === true) {
        this.stopAnimations();
        this.idleAnimation.play();
    }
}
```

Last, but not least, we have three small methods, which handle setting the players gravity, stopping all current animations and updating the players state. Please note that I used topic at http://www.html5gamedevs.com/topic/3003-sprite-transparency/ to find out how to change the players transparency.

```
this.setGravity = function(gravity) {
    // Gravity set-er
    this.playerSprite.body.allowGravity = gravity;
}

this.stopAnimations = function() {
    this.idleAnimation.stop();
    this.walkAnimation.stop();
    this.jumpAnimation.stop();
    this.climbAnimation.stop();
    this.steps.musicStop();
}

this.updateState = function(state) {
    // http://www.html5gamedevs.com/topic/3003-sprite-transparency/
    if (state === this.playerStates.LIGHT) {
        this.state = this.playerStates.LIGHT;
        this.playerSprite.alpha = 1;
    }
    else if (state === this.playerStates.DARK) {
        this.state = this.playerStates.DARK;
        this.playerSprite.alpha = 0.5;
    }
}
```

## Enemy

The enemy class handles everything related to the enemy class, following a similar design and parameters to the player class. A main difference is the AITree class, which I’ll explain in the next section, and the states are different.

```
function Enemy(x, y) {
    this.enemySprite = game.add.sprite(x, y, 'enemy');
    this.enemySprite.width = (TileSizeX/2);
    this.enemySprite.height = TileSizeY;
    this.origX = x;
    this.origY = y;
    game.physics.enable(this.enemySprite, Phaser.Physics.ARCADE);
    this.enemySprite.body.allowGravity = true;
    this.enemySprite.body.mass = 0;
    this.baseSpeed = 75; // Base speed required as it gets updated by Behaviour
tree
    this.speed = this.baseSpeed;

    foreground.add(this.enemySprite);

    this.walkAnimation = this.enemySprite.animations.add('Walk',
Phaser.Animation.generateFrameNames('Run_', 0, 7, '', 3));
    this.walkAnimation.speed = 10;

    this.climbAnimation = this.enemySprite.animations.add('Climb',
Phaser.Animation.generateFrameNames('Climb_', 0, 4, '', 3));
    this.climbAnimation.speed = 10;

    this.yell = new sound('alert');
    this.yell.musicVol(0.75);

    this.enemyStates = {
        LEFT: 0,
        RIGHT: 1,
        CHASING: 2
    };
    this.state = this.enemyStates.RIGHT;
    this.ai = new AITree(this);

    // Add functions below
    ...
}
```

There are also functions, such as the update method and the resetPos method. The enemyUpdate method gets called every frame, and updates the enemies state and position accordingly. It also handles it's collision with other objects. The other methods are almost identical to the player class.

```
    // Add functions below
    this.enemyUpdate = function () {
        // Update
        this.enemySprite.body.velocity.x = 0;
        //this.enemySprite.body.velocity.y = 0;
        this.ai.treeUpdate();

        game.physics.arcade.collide(this.enemySprite, background);
        // Turn around if colliding with a wall!
        if (game.physics.arcade.collide(this.enemySprite, wallLayer)) {
            if (this.state === this.enemyStates.LEFT) {
                this.state = this.enemyStates.RIGHT;
            }
            else if (this.state === this.enemyStates.RIGHT) {
                this.state = this.enemyStates.LEFT;
            }
        }
        if (game.physics.arcade.overlap(this.enemySprite, stairLayer)) {
            this.setGravity(false);
        }
        else {
            this.setGravity(true);
        }
    }

    this.setGravity = function(gravity) {
        this.enemySprite.body.allowGravity = gravity;
    }

    this.resetPos = function() {
        this.state = this.enemyStates.RIGHT;
        this.enemySprite.body.allowGravity = false;
```

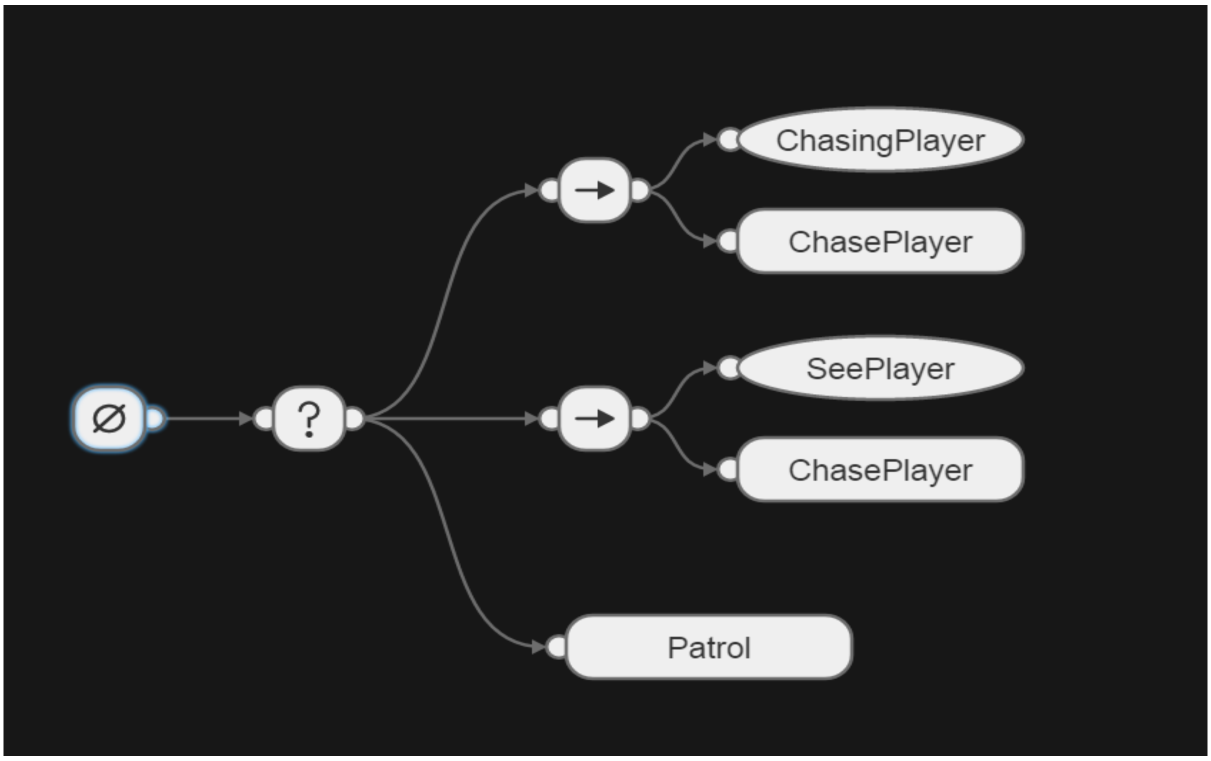
```
            this.enemySprite.x = this.origX;
            this.enemySprite.y = this.origY;
            this.speed = this.baseSpeed;
        }

        this.stopAnimations = function() {
            this.walkAnimation.stop();
            this.climbAnimation.stop();
        }

        playerDied.addSignal(function() {
            for (e in enemies) {
                enemies[e].resetPos();
            }
        });
        getIntel.addSignal(function() {
            for (e in enemies) {
                enemies[e].speed = enemies[e].baseSpeed +
    Math.round(enemies[e].baseSpeed/4); // Increase speed by a quarter!
            }
        });
    }
```

BehaviourTree

Using the Behavior3JS class by Renato Pereira I was able to produce an AI tree for the game, following the design of the image below. I created the diagram using http://behavior3js.guineashots.com/editor/, a visual edit for creating behaviour trees and then exporting them as .json.



The code to create the design above was complicated, and I had to use an example repository by efbenson call behavior3Test.  Using his example I was able to create a code structure which read my tree above and ran personalised functions. The code below shows the base of the AITree class, where it creates the load methods (loadAction, loadCondition) then calls the initaliser at the bottom.

```
var tree = {};

function AITree(enemy) {
    this.treeUpdate = function() {
        this.ai.guy.tick(this.character, this.character.memory);
    }

    this.loadAction = function(name, properties) {
        return loadTreeNode(name, properties, b3.Action);
    }

    this.loadCondition = function(name, properties) {
        return loadTreeNode(name, properties, b3.Condition);
    }

    ...

    this.treeInit = function(action, condition) {
        // Initalise custom actions & conditions
        this.actions(action);
        this.conditions(condition);
    }
    // Constructor:
    this.treeInit(this.loadAction, this.loadCondition);

    this.ai = {'guy': new b3.BehaviorTree()};
    this.ai.guy.load(JSON.parse(game.cache.getText('AITree')), tree);

    this.character = {
        memory:  new b3.Blackboard()
    };
    this.character.memory.set('pointer', enemy);
}
```

The initialiser is passing across the custom conditions and actions, which return either a SUCCESS or FAILURE result. These will all get passed into the loadTreeNode class which will create a code version of the tree diagram above.

To summarise the loadAction and loadCondition grab all the actions and conditions then passes them into the loadTreeNode. This code creates a new type, so either an action or condition, passes across the action/conditions name (i.e. ChasePlayer) and stores the function as the property. Finally it adds that node to the tree (An array of nodes), using the name as a key. It then returns the array to confirm that it was added to the tree.

```
function loadTreeNode(name, properties, type) {
    var node = b3.Class(type);
    var nodeProto = node.prototype;
    nodeProto.name = name;
    for (var prop in properties) {
        nodeProto[prop] = properties[prop];
    }
    tree[name] = node;
    return node;
}
```

And below are the methods will all the custom functions inside of them.

```
this.actions = function(action) {
    action('ChasePlayer', {
        tick: function(tick) {
            var enemy = tick.blackboard.get('pointer');
            enemy.enemySprite.body.velocity.x = 0;

            // Handles moving Left & Right
            if (player.playerSprite.x < enemy.enemySprite.x ) {
                enemy.enemySprite.body.velocity.x = -enemy.speed;
                if (enemy.enemySprite.width > 0) { // Flip image vertically (Face
Left)
                    enemy.enemySprite.scale.x *= -1;
                    enemy.enemySprite.x -= enemy.enemySprite.width; // Prevents
```

```
position from changing after flip
                }
            }
            else if (player.playerSprite.x > enemy.enemySprite.x) {
                enemy.enemySprite.body.velocity.x = enemy.speed;
                if (enemy.enemySprite.width < 0) { // Flip image vertically (Face
Right)
                    enemy.enemySprite.scale.x *= -1;
                    enemy.enemySprite.x -= enemy.enemySprite.width; // Prevents
position from changing after flip
                }
            }
            if (!enemy.walkAnimation.isPlaying && !enemy.climbAnimation.isPlaying)
{
                enemy.stopAnimations();
                enemy.walkAnimation.play();
            }
            // Handles moving Up or Down
            if (player.playerSprite.y != enemy.enemySprite.y) {
                // Check if player above or below AI
                if (game.physics.arcade.overlap(enemy.enemySprite, stairLayer)) {
                    if (player.playerSprite.y < enemy.enemySprite.y) {
                        enemy.enemySprite.body.velocity.y = -enemy.speed/2; // Move
at half speed
                    }
                    else {
                        enemy.enemySprite.body.velocity.y = enemy.speed/2; // Move
at half speed
                    }
                    if (!enemy.climbAnimation.isPlaying) {
                        enemy.stopAnimations();
                        enemy.climbAnimation.play();
                    }
                }
                if (!enemy.walkAnimation.isPlaying &&
!enemy.climbAnimation.isPlaying) {
                    enemy.stopAnimations();
                    enemy.walkAnimation.play();
                }
            }
            return b3.SUCCESS;
        }
    });
    action('Patrol', {
        tick: function(tick) {
            var enemy = tick.blackboard.get('pointer');
            if(enemy.state === enemy.enemyStates.LEFT) {
                enemy.enemySprite.body.velocity.x = -enemy.speed;
                if (enemy.enemySprite.width > 0) { // Flip image vertically (Face
Left)
                    enemy.enemySprite.scale.x *= -1;
                    enemy.enemySprite.x -= enemy.enemySprite.width; // Prevents
position from changing after flip
                }
                if (enemy.enemySprite.x <= (0 - enemy.enemySprite.width) ) {
                    // Needs to be minus as width becomes negative when flipped
                    enemy.state = enemy.enemyStates.RIGHT;
                }
            }
            else {
                enemy.enemySprite.body.velocity.x = enemy.speed;
                if (enemy.enemySprite.width < 0) { // Flip image vertically (Face
Right)
                    enemy.enemySprite.scale.x *= -1;
                    enemy.enemySprite.x -= enemy.enemySprite.width; // Prevents
position from changing after flip
                }
                if (enemy.enemySprite.x >= (GAMEWIDTH - enemy.enemySprite.width)) {
                    enemy.state = enemy.enemyStates.LEFT;
                }
            }
            if (!enemy.walkAnimation.isPlaying) {
                enemy.stopAnimations();
                enemy.walkAnimation.play();
            }
            return b3.SUCCESS;
        }
    });
}

this.conditions = function(condition) {
    condition('ChasingPlayer', {
        tick: function(tick) {
            var enemy = tick.blackboard.get('pointer');
            if (enemy.state === enemy.enemyStates.CHASING) {
                return b3.SUCCESS;
            }
            return b3.FAILURE;
        }
    });
    condition('SeePlayer', {
        tick: function(tick) {
            var enemy = tick.blackboard.get('pointer');
            if(enemy.state === enemy.enemyStates.LEFT) {
                if(player.state === player.playerStates.LIGHT
                    && player.playerSprite.x <= enemy.enemySprite.x
                    && player.playerSprite.y === enemy.enemySprite.y) {
                    enemy.state = enemy.enemyStates.CHASING;
                    enemy.speed = enemy.baseSpeed+50;
                    enemy.yell.musicPlay();
                    return b3.SUCCESS;
                }
            }
            else {
                if(player.state === player.playerStates.LIGHT
                    && player.playerSprite.x >= enemy.enemySprite.x
                    && player.playerSprite.y === enemy.enemySprite.y) {
                    enemy.state = enemy.enemyStates.CHASING;
                    enemy.speed = enemy.baseSpeed+50;
                    enemy.yell.musicPlay();
                    return b3.SUCCESS;
                }
            }
            return b3.FAILURE;
        }
    });
}
```

### Floor

Very similar to the BGTile in design except that the arcade physics apply, so that the player and enemy can collide with them. Despite this the gravity has been turned off and the floor objects are immovable, meaning objects can't move them.

```
function Floor(x, y) {
    this.floorSprite = game.add.sprite(x, y, 'floor');
    this.floorSprite.width = TileSizeX;
    this.floorSprite.height = TileSizeY;

    game.physics.enable(this.floorSprite, Phaser.Physics.ARCADE);
    this.floorSprite.body.allowGravity = false;
    this.floorSprite.body.immovable = true;
    background.add(this.floorSprite);
}
```

Wall

The Wall class is almost identical to the Floor object, the only difference being the graphic passed being used.

```
function Wall(x, y) {
    this.wallSprite = game.add.sprite(x, y, 'wall');
    this.wallSprite.width = TileSizeX;
    this.wallSprite.height = TileSizeY;

    game.physics.enable(this.wallSprite, Phaser.Physics.ARCADE);
    this.wallSprite.body.allowGravity = false;
    this.wallSprite.body.immovable = true;
    wallLayer.add(this.wallSprite);
}
```

Light

The Light class is almost identical to the Wall object, the only difference being the graphic passed being used.

```
function Light(x, y) {
    this.lightSprite = game.add.sprite(x, y, 'light');
    this.lightSprite.width = TileSizeX;
    this.lightSprite.height = TileSizeY;

    game.physics.enable(this.lightSprite, Phaser.Physics.ARCADE);
    this.lightSprite.body.allowGravity = false;
    this.lightSprite.body.immovable = true;
    lightLayer.add(this.lightSprite);
}
```

Stairs

The Stair class is almost identical to the Light object, the only difference being the graphic passed being used.

```
function Stair(x, y) {
    this.stairSprite = game.add.sprite(x, (y-5), 'stairs');
    this.stairSprite.width = TileSizeX;
    this.stairSprite.height = (TileSizeY+5);

    game.physics.enable(this.stairSprite, Phaser.Physics.ARCADE);
    this.stairSprite.body.allowGravity = false;
    this.stairSprite.body.gravity.y = 0;
    this.stairSprite.body.immovable = true;
    stairLayer.add(this.stairSprite);

    this.stairCollision = function () {
        // Turn player gravity off!
        player.setGravity(false);
    }
}
```

Exit

The Exit class is almost identical to the Stair object, the only difference being the graphic passed being used.

```
function Exit(x, y) {
    this.exitSprite = game.add.sprite(x, y, 'exit');
    this.exitSprite.width = TileSizeX;
    this.exitSprite.height = TileSizeY;

    game.physics.enable(this.exitSprite, Phaser.Physics.ARCADE);
    this.exitSprite.body.allowGravity = false;
    this.exitSprite.body.immovable = true;
    exitLayer.add(this.exitSprite);
}
```

## Intel

The Intel class is similar to the Exit object except for the graphic passed being used, having a destroy method and adding to the “getIntel” signal. Signals will be explained in the next session.

```
function Intel(x, y) {
    this.intelSprite = game.add.sprite(x, (y+(TileSizeY/2)), 'intel');
    this.intelSprite.width = (TileSizeX/2);
    this.intelSprite.height = (TileSizeY/2);

    game.physics.enable(this.intelSprite, Phaser.Physics.ARCADE);
    this.intelSprite.body.allowGravity = false;
    this.intelSprite.body.immovable = true;
    foreground.add(this.intelSprite);

    this.destroy = function() {
        this.intelSprite.destroy();
    }

    getIntel.addSignal (function() {
        intel.destroy();
    })
}
```

## Signals

A signal is like an event listener (or a middle man) where certain functions related to objects will get called when an event occurs. For example if the playerDied signal gets called then 2 functions will be run, one resets the players position and the other resets the enemies positon. Below is the object class that each signal is based off.

```
function Signal() {
    this.signal = new Phaser.Signal();

    this.addSignal = function (func) {
        this.signal.add(func);
    }
    this.call = function () {
        this.signal.dispatch();
    }
}
```

## Music

The name music may be a little misleading as this object handles all sounds, but it was originally designed to just handle the background music. I treated this as a juke box, where you could pass across a song, play it, loop it, stop it or queue a replacement song.

```
function sound(song) {
    // Init
    this.music = game.add.audio(song);

    this.musicUpdate = function() {
        if(!this.music.loop && !this.music.isPlaying) {
            this.music = game.add.audio(this.nextSong);
            this.musicLoop();
        }
    }

    this.queueSong = function(song) {
        this.nextSong = song;
        this.music.loop = false;
        this.music.fadeOut(5000);
    }

    this.musicPlay = function() {
        this.music.play();
    }

    this.musicLoop = function() {
        this.music.play();
        this.music.loopFull();
    }

    this.musicStop = function() {
        this.music.stop();
    }

    this.musicVol = function(vol) {
        this.music.volume = vol;
    }
}
```

### Assets

My image assets, apart from the Ladder.png, have come from Game Art 2D and are under the "Creative Common Zero (CCO) a.k.a Public Domain license" (GameArt, 2016). They provided me with a large amount of PNG files and I used a tool to merge them into a single spritesheet image, and created an xml file for them. The tool I used was Sprite Sheet Packer which is an open-source tool for merging PNGs.

For my sounds assets I used freesound, a website which provides sounds under the Creative Commons license. I've got details for all the sounds I've used below:

*background_eerie.wav* - Ambient electronic loop 001 by frankumjay

*background_epic.wav* – "Mike TheTunk Woloszyn" or "Mike Woloszyn" and www.Senproductions.de

*robot_intruder.wav* - Mudkip2016 robot requested by kennysvoice

*Steps.wav* - Footsteps on an interior wooden floor by ftpalad

## Investigation and Research

When I began this project, I planned to use Intelligent agents for my AI. Intelligent agents is a form of AI which reacts to the environment state rather than individual actions, for example if the level was "on fire" then the AI might walk away from the location of the fire. I thought this would suit the design of my game but after doing some research I came across behaviour trees.

The result can be read at the behaviour tree section but it did change my beginning design. I found this change allowed me to create AI who would better react to situations, like spotting the player, and leave room for extension (As represented in graph form).

Another later addition to the program was Phaser Signals, a feature that allowed me to create a middle man between all the objects. They work by adding "calls" to a signal, meaning that it will execute one or several functions, when the signal gets called. That meant that if the player died then one function will reset the player position and then several others would reset all the enemies' positions.

There wasn't much support for this feature, and the majority of my research came from the lecturer and http://phaser.io/docs/2.6.2/Phaser.Signal.html.

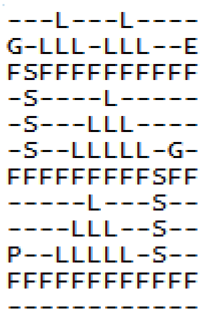
## Review of Final Product

In this section I'll give three pros to the design and implementation of the application and then list three improvements and how I might implement them.

## Pros

My first Pro is the Behaviour trees, which provided independent AI with the ability to react to situations. It also allows for me to easily expand upon or alter if I wanted to make the game more challenging/fair.

The second Pro is the level design, which is crafted from text files stored with a folder called levels. It loads levels by going through the document and crafting objects based on what character is used, and where it is in the file. The image below shows the level design for level 1.



```
---L---L----
G-LLL-LLL--E
FSFFFFFFFFFF
-S----L-----
-S---LLL----
-S--LLLLL-G-
FFFFFFFFFSFF
-----L---S--
----LLL--S--
P--LLLLL-S--
FFFFFFFFFFFF
------------
```

The dashes represent empty squares, the P represents the Player, the G represents a guard, the F represents a Floor object, etc. This design allowed for levels to be created or altered without having to change any of the code.

Finally the ability to store your time in local storage means that players can show off their success to others or remember how far they got in their last run.

## Improvements

My first improvement would be to implement different difficulty levels to the game, allowing players to partake in a more challenging experience. I would do this by increasing the movement speed of the enemies, and add one or two more guards (Based on the difficulty level).

Next, I'd add a tutorial level as my level one, as opposed to what is currently there. This would help players understand the controls and grasp the games challenge faster – rather than the current trial and error approach. I'd do this by simplifying level one then adding on-screen text explaining what to do.

Finally, I'd like to add an opening scene/video to the game, explaining the story behind the game and why the player must guide the ninja to retrieve the intel. I'd do this by creating then recording the story window, then present it to the player once they click the start button.

# References

http://www.freesound.org/

http://www.gameart2d.com/freebies.html

http://spritesheetpacker.codeplex.com/

http://behavior3js.guineashots.com/

http://behavior3js.guineashots.com/editor/

https://github.com/efbenson/behavior3Test

http://www.html5gamedevs.com/topic/3003-sprite-transparency/

## Appendix

### Appendix 1:

No installation is required, just click on the link provided and that will run the game.

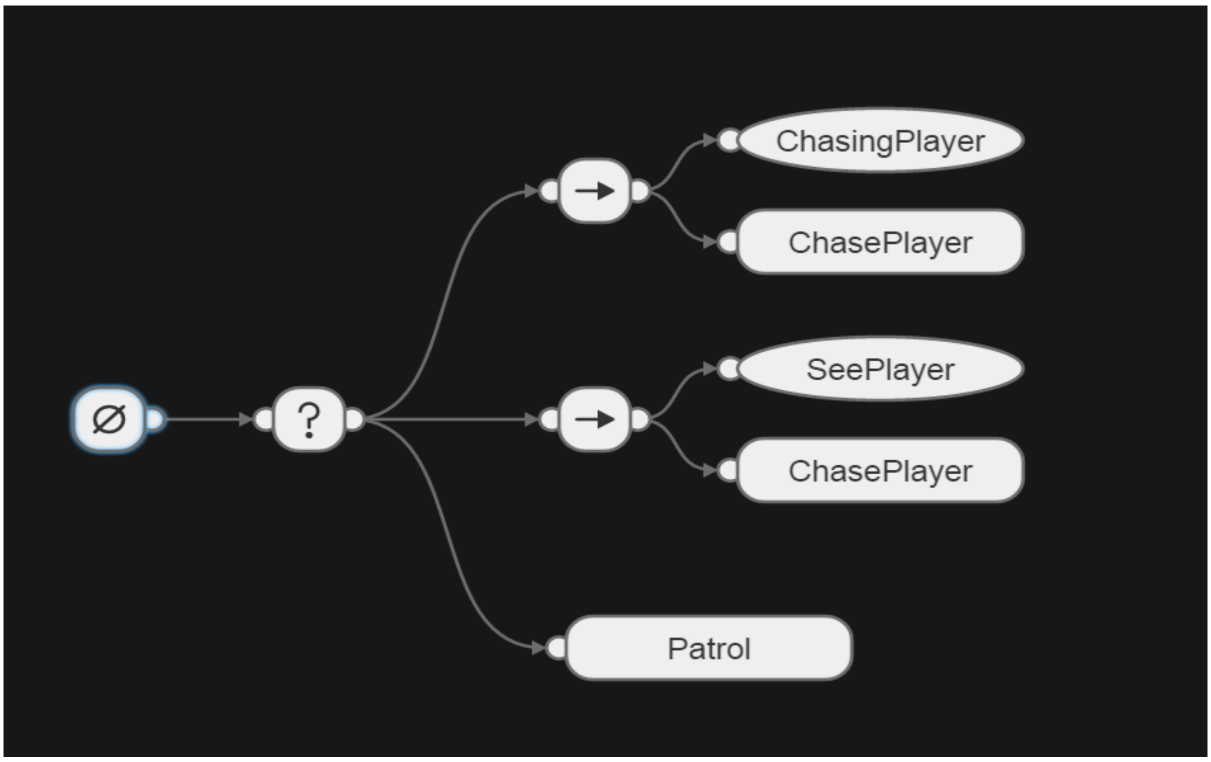
https://joshuapetherick.github.io/NSG/

### Appendix 2:

The Game Concept Presentation can be found within the Github repository.

https://github.com/JoshuaPetherick/NSG/raw/master/PresentationDraft.pptx

### Appendix 3:


∅
?
→
ChasingPlayer
ChasePlayer
→
SeePlayer
ChasePlayer
Patrol