Game Report

Contents

Game Summary

DueP: Yet Another 2D Cooperative Platformer, or *DueP* for short, is a two-player, two-dimensional platformer that was created using the Phaser.io framework. This game takes influence from several well-known and often-remembered titles, such as *Super Mario Bros* and *Bubble Bobble*.

Objective(s)

The main objective of *DueP* is to avoid and/or defeat all the enemies that are placed in a collection of different levels and reach the end of said levels, in which case the players will encounter additional obstacles that are ever increasing in difficulty. Not everything is out to harm the players, though; different power ups that players can use to their advantage are littered across the landscapes, and there are also several coins to collect that serve to increase the characters' scores.

Rules

The two characters can run left/right and jump; Player 1 uses the "A" and "D" buttons to run left and right respectively, the "W" button to jump, and the "C" button to shoot, while Player 2 uses the left, right and up arrow keys to move left, move right and jump, while the "M" button is used to shoot. They also have separate score, health and life systems.

Gameplay.

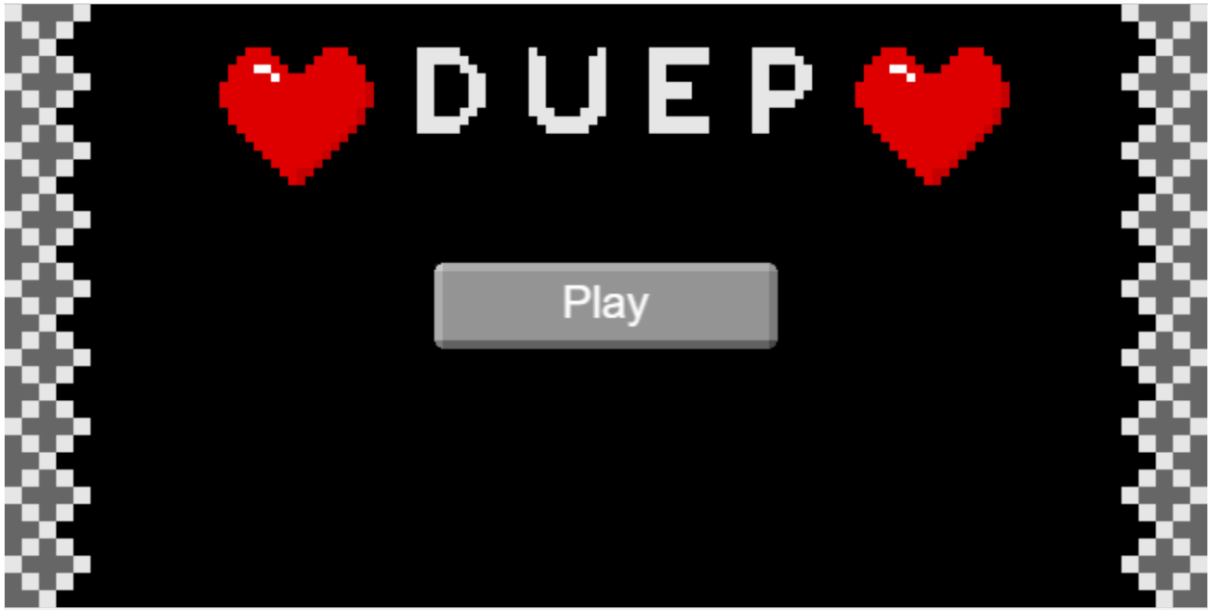
DueP's playable characters are forced to fight against several flying, bee-like enemies. Much like in *Super Mario Brothers*, direct contact with creatures will cause a player to lose some of their health. Unlike *Super Mario*, however, players cannot defeat enemies by jumping on them; instead, they are required to press a separate button to have the characters throw their hats to defeat them.

Power ups, that can be found in abundance in the game's two-level maps, include the "shoe" power up, which temporarily increases the speed of whoever touches it, and the "high jump" power up, which temporarily increases a player's jump height. The game also features a "high score" mechanic,

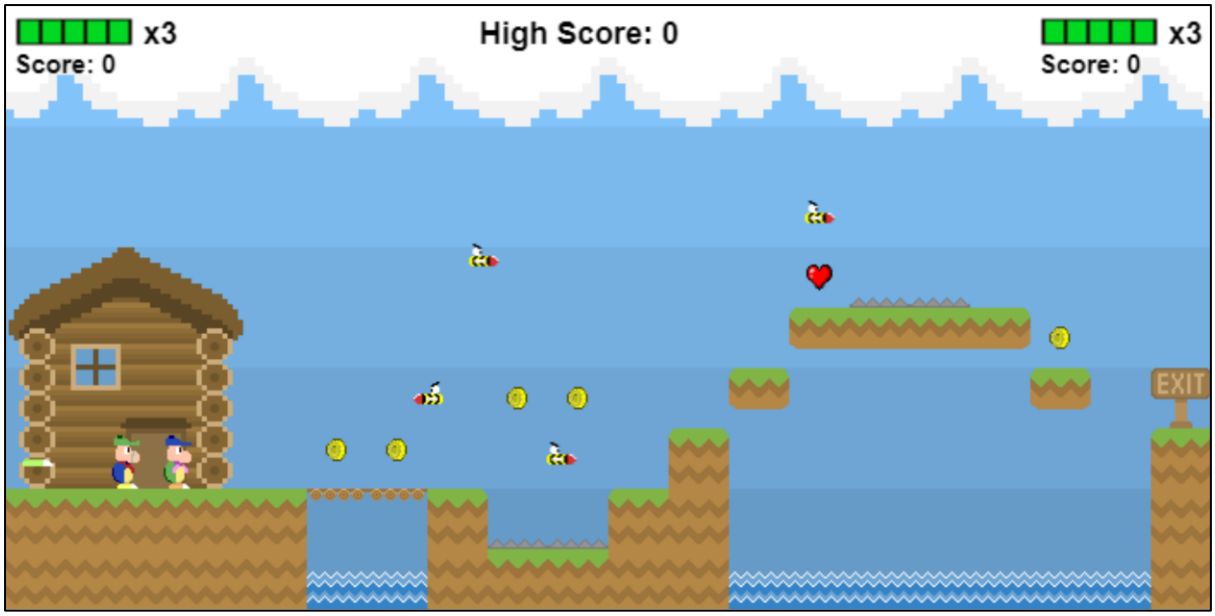
again taking cues from several popular arcade games from the 1980s and the 1990s, such as *Donkey Kong, Pacman* and *Street Fighter II*.

Game Structure

The current version of *DueP* features a total of two fully implemented levels. They both borrow certain tropes from the typical "grasslands level" theme, which has become famous for being featured in the first levels of many a different platformer. Obstacles that the players must stay clear from include spikes and bottomless pits. If a player touches a spike, then one point is taken from their health bar (Player 1 and Player 2 both start the game with five of these so called "health" points"). If a player loses all their health points, they will immediately lose a life, and they will be sent back to the beginning of the level. If one player loses all their lives, then both players will be forced to start from the beginning of the level, and the current level will restart.

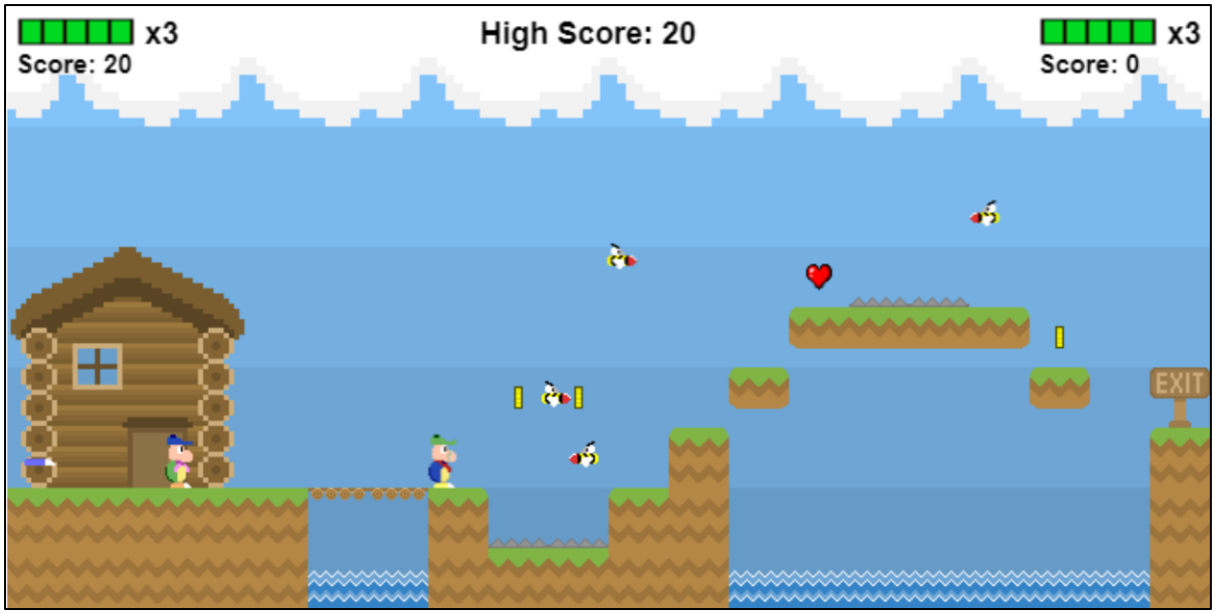


This is *DueP*'s "Menu" state. A simplified version of the game's logo (created by Daniele) is displayed, along with other miscellaneous graphical assets and a button that, when pressed, will immediately send the player(s) to the game's first level.

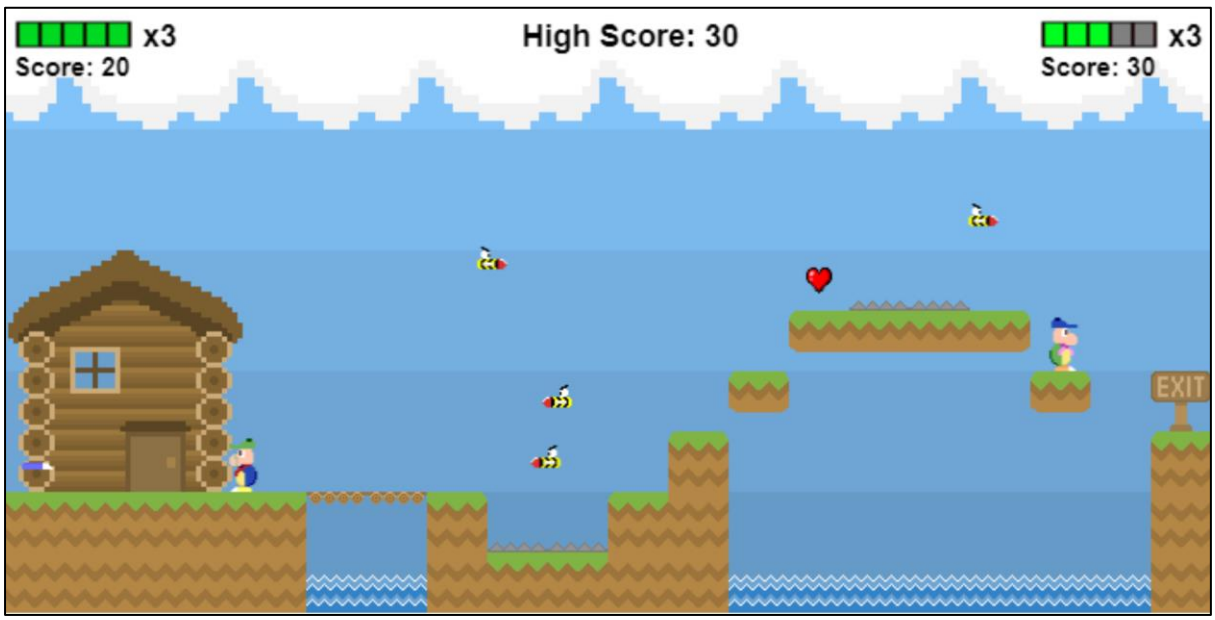


Pressing the “Play” button that is featured in *DueP*’s previously mentioned Menu state will result in the game instantly loading the game’s first “level” state. *DueP*’s levels were constructed using the Tiled application, which can be used to create maps for both side-scrolling and isometric games. Such approach was deemed favourable to creating the levels and hand placing the various collectables; as a result, the data for the game’s two levels has been laid out in the JSON data format.

The first level was intended from the start of development to be a “tutorial” level of sorts; the map is rather small – the second level’s map twice as large in comparison – and the number of obstacles/enemies that the players are required to outmanoeuvre is relatively small, and the existing ones should be low enough in difficulty that the players are able to finish the level with little to no difficulty. Besides the previously mentioned obstacles, other features include multiple coins to collect and a “shoe” power up. It is here that new players can observe the game’s multiplayer-oriented hub for the first time. In this screenshot, the two players have just started the level, having not yet made any progress; the two score counters, positioned at the far left and the far right of the screen, are recording zero points for both players, along with the high score, and there is the uncollected power up that has been placed to the left of the players’ initial starting points.

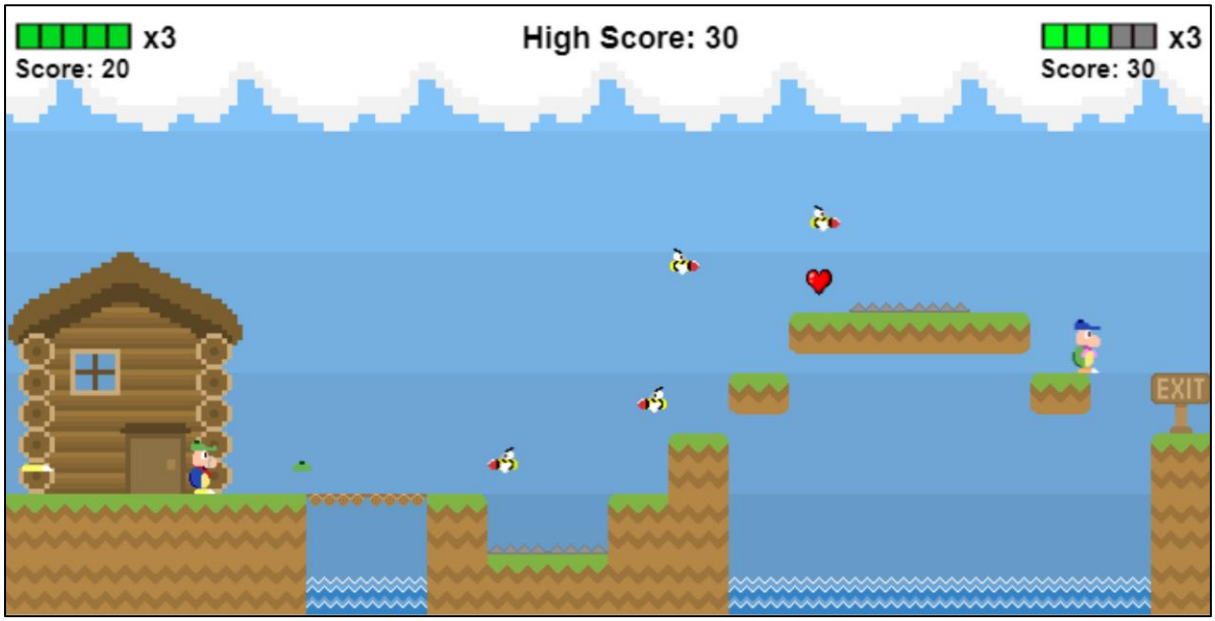


Here, the first player has started to walk towards the right of the screen, which is also the location of the “checkpoint” that will take both players to the beginning of the next level when/if it has been touched. It has also managed to collect a few “coin” collectables, which has increased the first player’s score, as well as the high score, since this is the first time that the game is being played in this particular setup. At this point in the game, players will most likely be able to note that the playable characters are well animated – animations are played when they move, jump and attack (attacking is further explained in the next paragraph). This also applies to the game’s enemies, who have animations for moving around the maps of both the first and second levels.

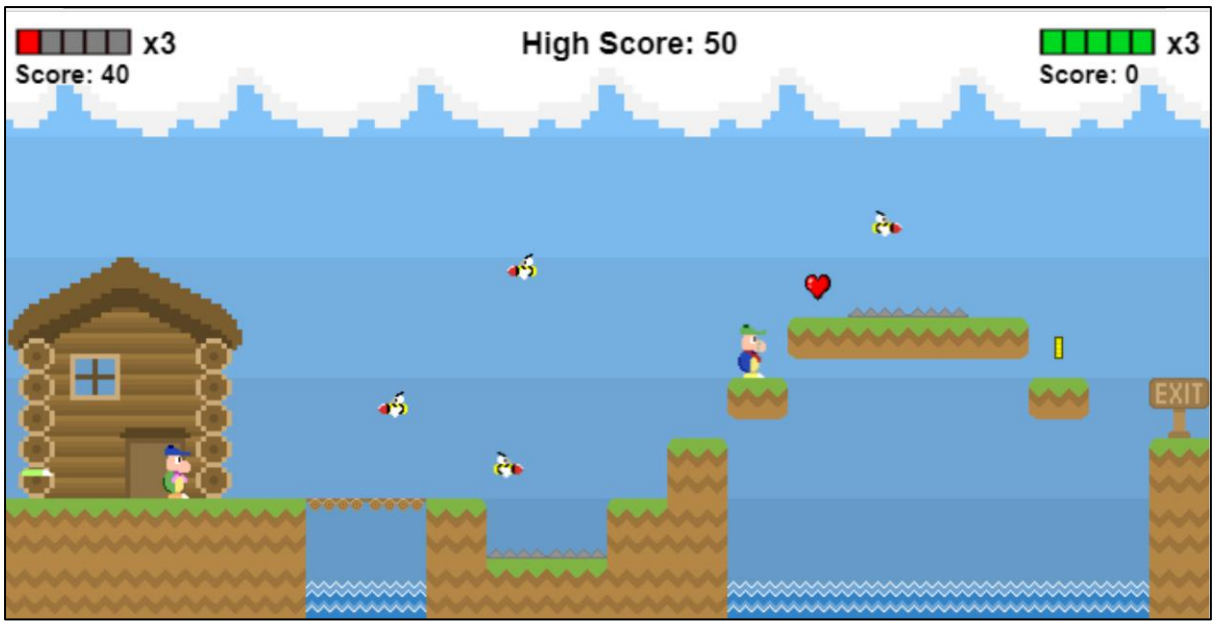


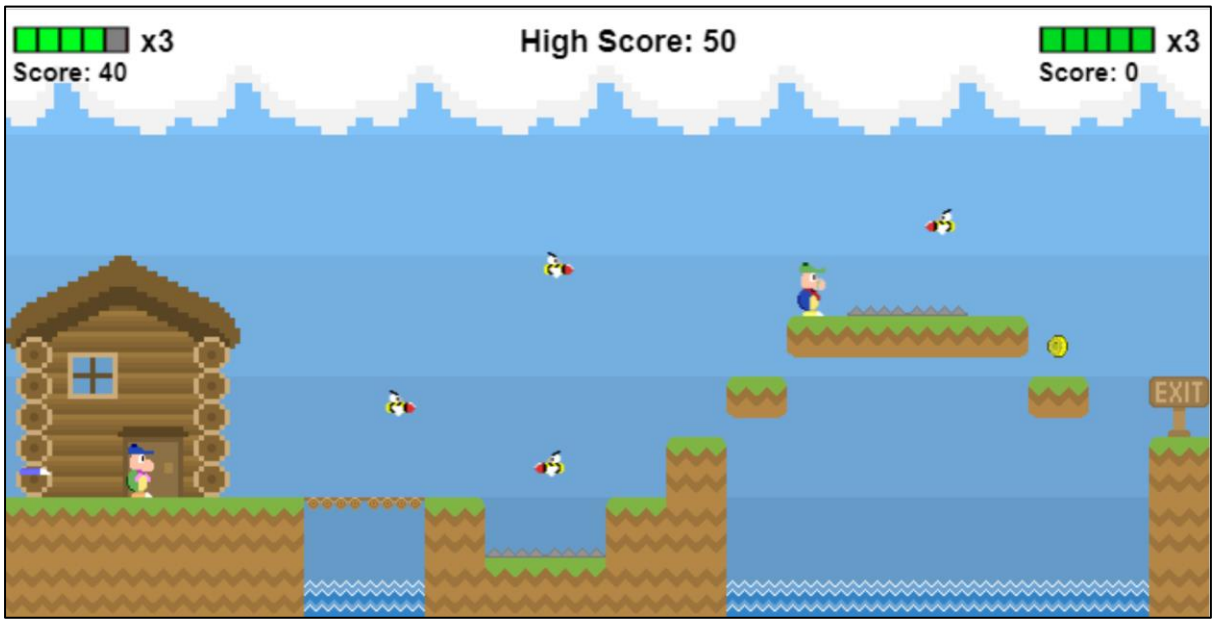
In this screenshot, the second character has also gotten their character to move; in fact, the second player has managed to accrue the biggest number of collectables of the two characters, and as a

result, the second player has the higher score, and the high score for the level has been updated with this new information in mind. In addition to this, the second player has managed to collide with a dangerous bed of spikes, which has resulted in the character taking damage and losing one of its five available health points. In-game, this event is symbolised by the player character's sprite temporarily becoming transparent, while sporting a red glow. While the character is in this state, he is unable to take any further damage for a few seconds, which was intended to give the player enough time to move their respective player away from damage-inflicting obstacles before they can lose any more health points.

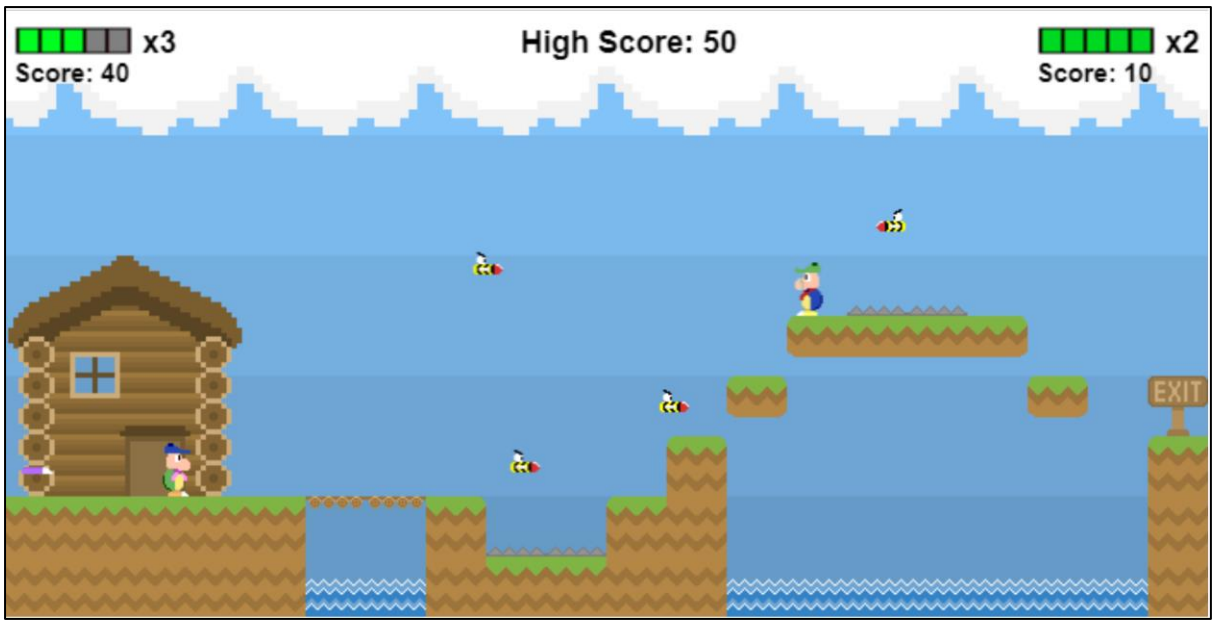


In this screenshot, the first player character can be seen shooting a projectile – its hat – at one of the level's multiple enemies. The hat projectiles provide the only available means of defence for both characters, and they will never run out of ammo – jumping on an enemy will only result in a loss of health.

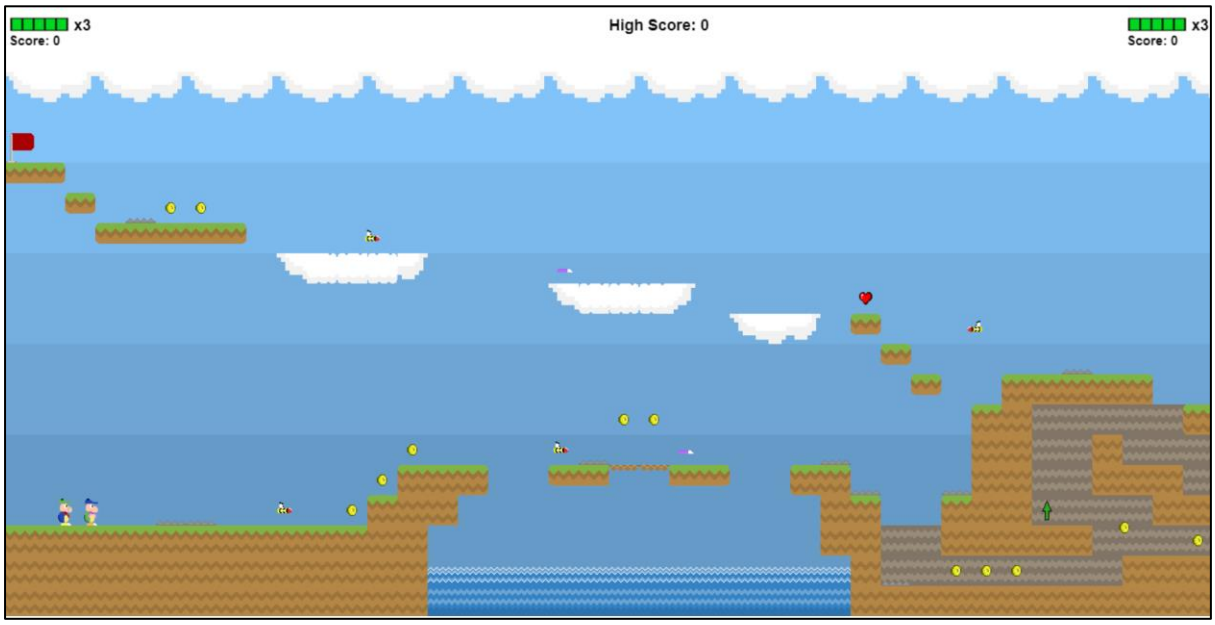




These two screenshots give a basic visualization of the “heart” power up’s in-game functionality – namely that if a player comes into contact with it, their total health will be restored by three points. The first screenshot features the two players and the heart. One of the player characters only has one health point left – if it were to collide with another obstacle, the player would have to resume their progress from the beginning of the stage. The second screenshot shows the previously explained effect of picking the heart up.

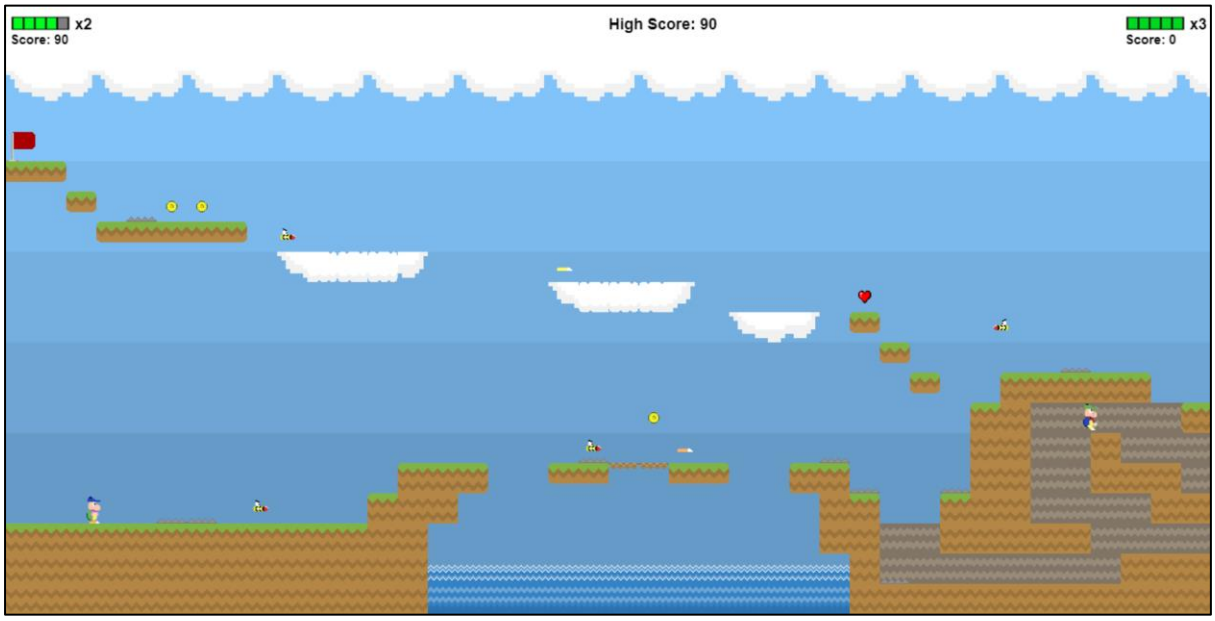


Here, one can see that the second player now only has two lives instead of three. In the current version of *DueP*, there are two ways that this can happen: either the player touches an enemy/obstacle one too many times and loses all their available health points, or they fall into a bottomless pit. If one player loses all three of their given lives, then the game's current state will immediately restart, the scores for both players will be set at 0, and any items that the players had collected beforehand will respawn. The high score, however, will always stay intact.

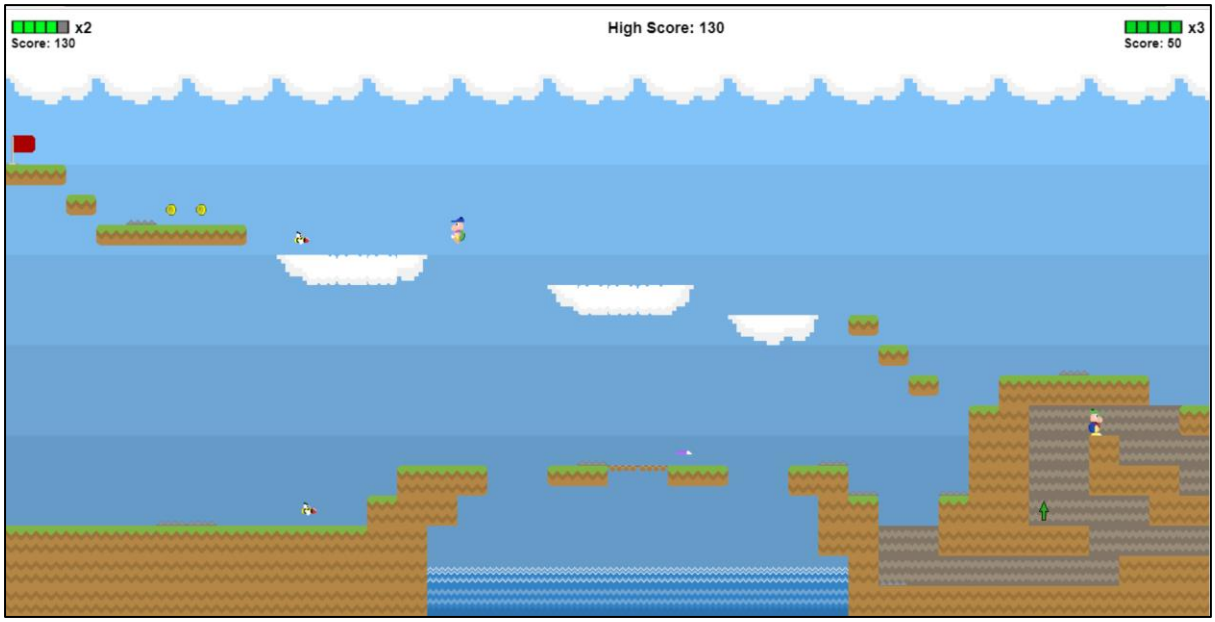


Here, the two players have managed to complete the first level and are now faced with overcoming obstacles/enemies in the second level. The second level was intended to be a departure from the relatively simple layout of the first level – as was previously mentioned, it is currently twice as large as the preceding stage, and the power ups influence the level design to a greater degree. It is here

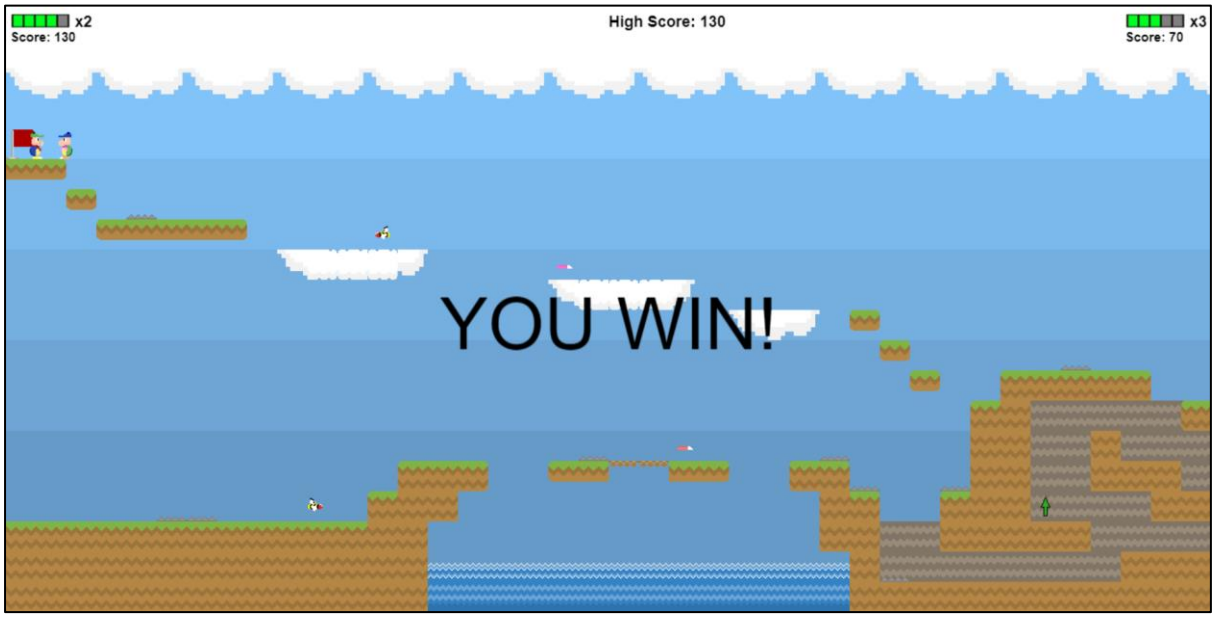
that a new power-up, the “High Jump”, needs to be taken advantage of for the players to have any hope of finishing the level. The second level is also slightly more difficult –the larger map size has resulted in the bottomless pits being larger and even more of a hazard.



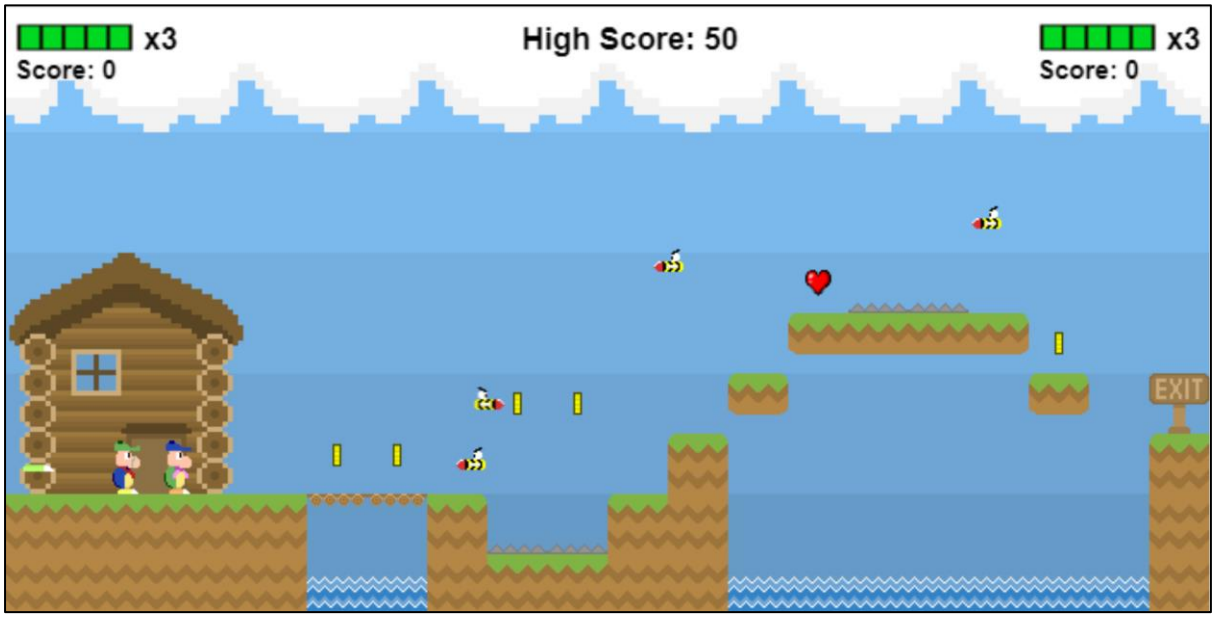
Here the first player has just collected the “High Jump” power-up and is currently in the middle of making a jump. As the name implies, collecting the power-up, which is represented in-game as a small, green arrow, will grant the player an increased jump height for a few seconds. To prevent the otherwise likely situation where one player is unable to further progress because another player had collected this power-up, these items have been designed to re-spawn soon after collection.



The second player has now collected the “Speed Shoe” power-up that was present in the previous level; here, it is required in order to jump between floating “cloud” platforms; without it, the player would be unable to make the jump, and would instead plummet to the bottom of the level.



This screenshot shows that the first player has managed to “touch” the second level’s checkpoint. Since this is the last level, both players will be greeted with a short congratulatory message for completing the game, and they will be sent back to the main menu after a few seconds, allowing for an infinite amount of continuous playthroughs.



The players have opted to play the game again from the beginning, and for the most part, the gameplay mechanics this time around are functionally identical to how they were when they had

played it for the first time. There is, however, one major difference: the high score from the previous playthrough remains untouched – during the first playthrough, the high score was 0 points, since the game had not previously been played. This feature was implemented with the help of the window.localStorage HTML5 object, which can potentially also be used to save current game states.

Implementation Specification

From the start of development, the game was intended to be a platformer; both students had designed games belonging to different genres for the Semester 1 projects, so the conclusion was that a platformer, which was seen as something that could be created within a reasonable amount of time, while at the same time allowing for potential room of enhancement, was an adequate concept to fully realise. Initially designed to be a "fixed-room" platformer that would feature two players competing against each other for points – taking cues from multiplayer 2D games such as *Mario Bros* and the previously mentioned *Bubble Bobble* – The game concept switched to a more traditional side scrolling platformer later in the development process.

The current version of *DueP* features a grand total of two fully implemented levels. They both borrow certain tropes from the typical "grasslands level" theme, which has become famous for being featured in the first levels of many a different platformer. Obstacles that the players must stay clear from include spikes and bottomless pits. If a player touches a spike, then one point is taken from their health bar (Player 1 and Player 2 both start the game with five of these so called "health" points"). In the event that a player loses all of their health points, they will immediately lose a life, and they will be sent back to the beginning of the level. If one player loses all of their lives, then both players will be forced to start from the beginning of the level, and the current level will restart.

To better accommodate multiple players, these UI elements were turned into objects that were derived from respective prefab classes.

Both Player 1 and Player 2 are represented in-code as two "Player" objects, which in term are derived from the custom-made "Player" prefab class. Based on what has been laid out and specified in this prefab file, any object instance of this class is able to move left and right (code has been written to handle situations where the player has to turn to the left), as well as jump, shoot projectiles and react to the environments around them, whether the elements surrounding said environments are safe or hazardous.

Sound effects have also been implemented in the current version of *DueP*; examples of scenarios where these are played in regular gameplay include the player jumping, falling into a bottomless pit, picking up a power up and touching an obstacle or an enemy.

Unlike *the Semester 1 projects*, *DueP* was not created with touch devices in mind; the game was tested exclusively on computers with keyboards, and as of the completion of the game, it has been tested that the game will work as intended on personal desktops and laptops.

Implementation Evaluation

DueP, as it stands, is a game that has successfully implemented the gameplay mechanics required to meet, and potentially even surpass, the university assessment's requirements. The game features good implementations of AI, collision detection and physics, and the addition of multiplayer gives the game a certain "edge" of sorts that can serve to improve on critical real-world skills such as teamwork/cooperation and immersion. There are three reasons to hold the opinion that the implementation/design of the game more than meets expectations:

Multiplayer

DueP was intended from the very start of development to be a multiplayer cooperative game, and based on playthroughs of the most recent version, it has more than fulfilled this particular requirement. Up to two players can play the game at any given moment, but this is not mandatory – only one player needs to touch the levels' checkpoints to further progress, so people who wish to play solo are not necessarily left out.

High score mechanic

One of the main criticisms that appeared in the critical review of *Wizard Clash* was that it did not utilize any sort of "high score" mechanic. While this is obviously not a mandatory feature in video games, it does continue to serve a purpose as something that can potentially increase a game's replay value – for example, a person may pass by an arcade cabinet and look at the highest scores that had been earned by past players, and in turn they may be encouraged to play the game themselves, while being fuelled primarily by competitive intentions. This feature has been fully implemented in *DueP*, and it could even be considered a primitive "save" system, since the value stored in the "high score" variable stays intact even if a player were to close the browser that the game is running on.

The game is relatively simple, both code-wise and gameplay-wise.

DueP, like *Wizard Clash* before it, has a relatively simple control scheme – in fact, even less buttons are required to fully control a player and/or manipulate its surroundings. The current version of *Wizard Clash* uses a total of nine buttons: the arrow keys and the "A" (for attacking), "D" (for summoning protective shields), "H" (for using one of a limited number of potions to restore health)

and “S” (for running) buttons, while *DueP* only uses one action button for each player, which, when pressed, forces the characters to shoot their projectiles at nearby enemies. This lack of complexity is likely to be seen as a good thing, especially considering the video game genre that the game is linked to; for reference, the original *Super Mario Bros*. games on the Nintendo Entertainment System only utilized two action buttons: one for jumping, and one for running. The original *Sonic the Hedgehog games* on the Sega Mega Drive took this one step further and only used one action button, with more focus being given towards taking advantage of action button/directional pad combinations. Too much complexity in a game control-wise could potentially result in audiences being confused by the mechanics, and most 2D platformers require a rather small number of buttons to be above playable.

The topic of simplicity also somewhat applies to the game’s code; compared to *Wizard Clash, DueP’s* code implements more prefab classes, as well as JSON data and Phaser states that allows for the addition of multiple levels with relative ease. This is in stark contrast to *Wizard Clash,* where all the code for the game logic was stored in one state, with only the coins and the enemies having accompanying prefab files.

Code Structure

The code is structured to have a total of five game states where each state oversees a specific task.

Main

```
7   // If Platformer exists, then use it. Otherwise initiate it as an empty object
8   var Platformer = Platformer || {};
9
10  var WIDTH = 20,
11      HEIGHT = 10,
12      TILEWIDTH = 35;
13
14  Platformer.game = new Phaser.Game(WIDTH * TILEWIDTH, HEIGHT * TILEWIDTH, Phaser.AUTO, '');
15
16  Platformer.game.state.add('Boot', Platformer.Boot);
17  Platformer.game.state.add('Preload', Platformer.Preload);
18  Platformer.game.state.add('MainMenu', Platformer.MainMenu);
19  Platformer.game.state.add('Level1', Platformer.Level1);
20  Platformer.game.state.add('Level2', Platformer.Level2);
21
22  Platformer.game.state.start('Boot');
```

The main class specifies the width and height of the game, adds the states that form the game and starts the first state (Boot state).

Boot state

```
Platformer.Boot = function(){};

// Setting game configuration and loading the assets for the loading screen
Platformer.Boot.prototype = {

    preload: function() {

        // Loading screen image
        this.load.image('title', 'assets/images/title.png');

    },

    create: function() {

        // Loading screen will have a white background
        this.game.stage.backgroundColor = '#fff';

        // Scaling options
        this.scale.scaleMode = Phaser.ScaleManager.SHOW_ALL;

        // Have the game centered horizontally
        this.scale.pageAlignHorizontally = true;
        this.scale.pageAlignVertically = true;

        // Physics system
        this.game.physics.startSystem(Phaser.Physics.ARCADE);

        // Gravity
        this.game.physics.arcade.gravity.y = 400;

        this.state.start('Preload');

    }

};
```

The Boot state is the first state that is loaded upon first launching the game. In here, game configurations such as the physics system and gravity are initialised along with loading the splash screen. Then, the Preload state is loaded.

Preload

```
var background;
var startTimer;

// Loading the game assets
Platformer.Preload = function(){};

Platformer.Preload.prototype = {

    preload: function() {

        // Load game assets
        this.game.load.tilemap('level1', 'assets/tilemaps/level1.json', null, Phaser.Tilemap.TILED_JSON);
        this.game.load.tilemap('level2', 'assets/tilemaps/level2.json', null, Phaser.Tilemap.TILED_JSON);
        this.game.load.image('gameTiles', 'assets/images/tiles.png');
        this.game.load.image('menu', 'assets/images/menu.png');
        this.game.load.image('spike', 'assets/images/spike.png');
        this.game.load.image('heart', 'assets/images/heart.png');
        this.game.load.image('goal', 'assets/images/exit.png');
        this.game.load.image('arrow', 'assets/images/arrow.png');
        this.game.load.image('win', 'assets/images/win.png', 35, 35);
        this.game.load.image('enemyWalls', 'assets/images/enemyWalls.png');
        this.game.load.spritesheet('button', 'assets/images/button.png', 200, 50);
        this.game.load.spritesheet('player1', 'assets/images/player1.png', 18, 32, 8);
        this.game.load.spritesheet('player2', 'assets/images/player2.png', 18, 32, 8);
        this.game.load.spritesheet('healthbar1', 'assets/images/healthbar1.png', 67, 15);
        this.game.load.spritesheet('coin', 'assets/images/coin.png', 25, 25, 4);
        this.game.load.spritesheet('speedShoe', 'assets/images/speedShoe.png', 27, 12, 8);
        this.game.load.spritesheet('missile1', 'assets/images/missile1.png', 17, 8);
        this.game.load.spritesheet('missile2', 'assets/images/missile2.png', 17, 8);
        this.game.load.spritesheet('enemy', 'assets/images/enemy.png', 19, 18);

        this.game.load.audio('collect_point', ['assets/audio/mp3/Collect_Point.mp3', 'assets/audio/ogg/Collect_Point.ogg']);
        this.game.load.audio('collect_powerup', ['assets/audio/mp3/Collect_Powerup.mp3', 'assets/audio/ogg/Collect_Powerup.ogg']);
        this.game.load.audio('hero_death', ['assets/audio/mp3/Hero_Death.mp3', 'assets/audio/ogg/Hero_Death.ogg']);
        this.game.load.audio('hurt', ['assets/audio/mp3/Hurt.mp3', 'assets/audio/ogg/Hurt.ogg']);
        this.game.load.audio('jump', ['assets/audio/mp3/Jump.mp3', 'assets/audio/ogg/Jump.ogg']);
        this.game.load.audio('restore_health', ['assets/audio/mp3/Restore_Health.mp3', 'assets/audio/ogg/Restore_Health.ogg']);

    },
```

As shown in the picture above, the Preload state is where all the game assets are loaded. Here the maps making up the in-game levels, the in-game objects such as players, power ups, enemies and the spikes are, as the name of the state itself suggests, preloaded so that when the Level state is initialised, the assets will have been already loaded.

```
    create: function() {

        // Splash screen
        background = this.game.add.sprite(this.game.world.centerX, this.game.world.centerY, 'title');
        background.anchor.set(0.5);
        background.scale.setTo(0.215, 0.215);

        this.game.time.events.add(Phaser.Timer.SECOND * 3, fadePicture, this);

        // Store the time at startup
        startTimer = this.time.now;

    },

    update: function () {

        // Switch to MainMenu state after 6 seconds
        if (this.time.now - startTimer >= 6000) {
            this.state.start('MainMenu');
        }
    }

};

// The picture will fade away
function fadePicture() {

    this.game.add.tween(background).to( { alpha: 0 }, 2000, Phaser.Easing.Linear.None, true);

}
```

Following, in the create function, the splash screen (previously loaded in the Boot State) is finally created. Said splash screen is also set to have a timer so that it will fade away after 3 seconds from when it first appears. Then, in the update function there is an additional timer set to fire off after 6 seconds. This is done by storing the time at start up in a variable and then compare the current time with the time stored. If the aforementioned statement holds true, then start the “MainMenu” state.

MainMenu

```
var button;

// Title screen
Platformer.MainMenu = function(){};

Platformer.MainMenu.prototype = {

    create: function() {

        this.game.add.image(0, 0, 'menu')

        // Play button
        button = this.game.add.button(this.game.world.centerX, this.game.world.centerY, 'button', actionOnClick, this, 2, 1, 0);
        button.anchor.set(0.5);

        // Start game text
        var text = "Play";
        var style = { font: "26px Arial", fill: "#fff", align: "center" };
        var t = this.game.add.text(this.game.world.centerX, this.game.world.centerY, text, style);
        t.anchor.set(0.5);

    }

};

function actionOnClick () {

    this.game.state.start('Level1');

}
```

Here is where the main menu, consisting of an image and a button, is created. The button’s only functionality is to start the next state when the user clicks on it. This was achieved by using the built-in function in Phaser called “actionOnClick”. This allows the user to interact with an object upon clicking on it. In this case the object is the button and then action performed is launching the next state.

The MainMenu is also the last state before the players can start playing the actual game. After this, the player will be sent to the first level of the game, hence the Level 1 state.

List of functions:

Since Level 2 features the same number of functions, with a few additionalities, as Level 1, The former will be used to describe the functions used to create the game mechanics so to avoid repetition.

create: function

```
Platformer.Level2.prototype = {

    create: function () {

        // Set new game size to match size of the tiled map
        this.game.scale.setGameSize(1400, 700);
        // Update world bounds
        this.game.world.setBounds(0, 0, 1400, 700);
        // Update camera info
        this.game.camera.setSize(1400, 700);

        map = this.add.tilemap('level2');

        // 1st parameter: tileset name as specified in Tiled,
        // 2nd parameter: key to the asset
        map.addTilesetImage('tiles', 'gameTiles');

        // Create layer
        backgroundLayer = map.createLayer('backgroundLayer');
        nonCollisionLayer = map.createLayer('nonCollisionLayer');
        collisionLayer = map.createLayer('collisionLayer');

        // Resizes the game world to match the layer dimensions
        backgroundLayer.resizeWorld();

        // Collision on collisionLayer
        // In the json file, the highest number is 70
        // Just to be sure, we check any tiles from 0 to 100
        map.setCollisionBetween(0, 162, true, 'collisionLayer');

        // Add a group to the collisionLayer to hold the group of objects
        spikeGroup = this.game.add.group(collisionLayer);
        spikeGroup.enableBody = true;

        coinGroup = this.game.add.group();
        coinGroup.enableBody = true;
        coinGroup.allowGravity = false;

        heartGroup = this.game.add.group();
        heartGroup.enableBody = true;
        heartGroup.allowGravity = false;

        speed_shoeGroup = this.game.add.group();
        speed_shoeGroup.enableBody = true;
        speed_shoeGroup.allowGravity = false;

        arrowGroup = this.game.add.group();
        arrowGroup.enableBody = true;
        arrowGroup.allowGravity = false;
```

The screenshot partially shows what the create function does. However, the sole task of said function is to create the objects that have been previously loaded in the Preload state. Worth noticing that, unlike Level 1, Level 2 needs to reset the size of the game, its world bounds and even the camera size so to match the json file making up the game's Level 2.

```
109         // Add sprite (GID) based on the infomation in the
110         // tilemap objects layer
111         map.createFromObjects('objectLayer', 48, 'spike', 0, true, false, spikeGroup);
112         map.createFromObjects('objectLayer', 55, 'coin', 0, true, false, coinGroup);
113         map.createFromObjects('objectLayer', 60, 'speedShoe', 0, true, false, speed_shoeGroup);
114         map.createFromObjects('objectLayer', 59, 'heart', 0, true, false, heartGroup);
115         map.createFromObjects('objectLayer', 68, 'arrow', 0, true, false, arrowGroup);
116         map.createFromObjects('objectLayer', 69, 'enemyWalls', 0, true, false, enemyGroupWalls);
117         map.createFromObjects('objectLayer', 70, 'win', 0, true, false, winGroup);
```

This is where Tiled shows all its potential. In here, Phaser communicates with the json file created in Tiled and creates the objects from the tilemap with given GID (global ID). Such objects are then put into their respective groups in order to better specify the features each group should have.

Finally, below is another example of how different objects are created in the Create function. This time the objects inherits their properties from external classes. For example, the line 170 shows how player1 is initialised by calling the class Player, which is a constructor that inherits from the Phaser.Sprite constructor. The player is fed with the parameters specified in the Player class in the following order: a pointer to the game (this.game), a sprite which will represent the player, the player's position along the x and y axis in the world and the value 1 which is needed to make a distinction between player1 and player2.

```
169         // Player 1
170         player1 = new Platformer.Player(this.game, 'player1', 70, 590, 1);
171         healthBar1 = new Platformer.HealthBar(this.game, 'healthbar1', 40, 15);
172 ▶       score1 = new Platformer.Score(this.game, 5, 25, "Arial", { ··· });
178 ▶       lives1 = new Platformer.Lives(this.game, 80, 5, "Arial", { ··· });
```

Update: function

```
231    update: function () {
232
233        // Players' controls
234        player1.controls.right = this.game.input.keyboard.isDown(Phaser.Keyboard.D);
235        player1.controls.left = this.game.input.keyboard.isDown(Phaser.Keyboard.A);
236        player1.controls.up = this.game.input.keyboard.isDown(Phaser.Keyboard.W);
237        player1.controls.shoot = this.game.input.keyboard.isDown(Phaser.Keyboard.C);
238
239        player2.controls.right = this.game.input.keyboard.isDown(Phaser.Keyboard.RIGHT);
240        player2.controls.left = this.game.input.keyboard.isDown(Phaser.Keyboard.LEFT);
241        player2.controls.up = this.game.input.keyboard.isDown(Phaser.Keyboard.UP);
242        player2.controls.shoot = this.game.input.keyboard.isDown(Phaser.Keyboard.M);
243
244        // Call collision detection for each layer that we care about
245        this.game.physics.arcade.collide(spikeGroup, collisionLayer);
246        this.game.physics.arcade.collide(winGroup, collisionLayer);
247
248        this.game.physics.arcade.overlap(player1, spikeGroup, this.playerGetsHit, null, this);
249        this.game.physics.arcade.overlap(player2, spikeGroup, this.playerGetsHit, null, this);
250
251        this.game.physics.arcade.overlap(player1, enemyAI, this.playerGetsHit, null, this);
252        this.game.physics.arcade.overlap(player2, enemyAI, this.playerGetsHit, null, this);
253        this.game.physics.arcade.overlap(player1, enemyAI2, this.playerGetsHit, null, this);
254        this.game.physics.arcade.overlap(player2, enemyAI2, this.playerGetsHit, null, this);
255        this.game.physics.arcade.overlap(player1, enemyAI3, this.playerGetsHit, null, this);
256        this.game.physics.arcade.overlap(player2, enemyAI3, this.playerGetsHit, null, this);
257        this.game.physics.arcade.overlap(player1, enemyAI4, this.playerGetsHit, null, this);
258        this.game.physics.arcade.overlap(player2, enemyAI4, this.playerGetsHit, null, this);
259
260        this.game.physics.arcade.overlap(player1, coinGroup, this.coinHit, null, this);
261        this.game.physics.arcade.overlap(player2, coinGroup, this.coinHit, null, this);
262
263        this.game.physics.arcade.overlap(player1, heartGroup, this.heartHit, null, this);
264        this.game.physics.arcade.overlap(player2, heartGroup, this.heartHit, null, this);
265
266        this.game.physics.arcade.overlap(player1, speed_shoeGroup, this.speed_shoeHit, null, this);
267        this.game.physics.arcade.overlap(player2, speed_shoeGroup, this.speed_shoeHit, null, this);
268
269        this.game.physics.arcade.overlap(player1, arrowGroup, this.arrowHit, null, this);
270        this.game.physics.arcade.overlap(player2, arrowGroup, this.arrowHit, null, this);
271
272        this.game.physics.arcade.overlap(player1, winGroup, this.goalHit, null, this);
273        this.game.physics.arcade.overlap(player2, winGroup, this.goalHit, null, this);
274
275        this.die(player1, healthBar1, lives1);
276        this.die(player2, healthBar2, lives2);
```

Again, the screenshot only shows a portion of what the update function contains. The update function is mainly used to check the collision between objects. This is also where the players' controls are specified. The collision are checked using the Phaser built-in function that makes use of physics.arcade. Inside these built-in functions the collision between two objects, for example player1 and enemyAI are constantly checked and the playerGetsHit is called.

render: function

```
296     render: function () {
297
298         // Uncomment if you wish to see the hit boxes of the objects below,
299         // the first player's velocity along the Y axis, and the camera info
300         //this.game.debug.body(player1);
301         //this.game.debug.text('Player N: ' + player1.data.playerNumber, this.game.world.centerX, 20);
302         //this.game.debug.text('Player Y: ' + player1.body.velocity.y, this.game.world.centerX, 40);
303         //this.game.debug.cameraInfo(this.game.camera, this.game.world.centerX, 60);
304
305         //spikeGroup.forEach(this.game.debug.body, this.game.debug);
306         //coinGroup.forEach(this.game.debug.body, this.game.debug);
307         //heartGroup.forEach(this.game.debug.body, this.game.debug);
308     },
```

The render function was used for debugging purpose only. As shown in the above picture, here you can check the hit boxes of the object listed inside the function.

die: function

```
310     /* Players' lives are decreased by 1 upon falling off the tiles
311      * or if their healthbars hit 0.
312      * Players' positions and healthbars will be reset after death
313      * If one of the players loses all of the lives, the game will be restared
314      */
315     die: function (player, healthBar, lives) {
316
317         // Player still has lives
318         if (lives.data.lives > 0) {
319             if (player.bottom >= this.game.world.height || healthBar.data.healthPoints === 0) {
320
321                 this.hero_deathSound.play();
322
323                 // Decrease lives
324                 lives.data.lives--;
325
326                 // Reset player's position
327                 if (player.data.playerNumber === 1) {
328                     player.reset(70, 500);
329                 } else {
330                     player.reset(100, 500);
331                 }
332
333                 // Restore health bar
334                 healthBar.data.healthPoints = 5;
335             }
336         }  else {
337
338             //Restart the current level
339             this.game.state.restart(this.game.state.current);
340         }
341
342     },
```

The die function takes in 3 parameters which are the reference to the player object, the healthBar and the lives respectively. The players' lives decrease by 1 when falling into the bottomless pits or if their health-bar is equal to 0. If, however, one of the players loses all the lives, the game will restart.

playerGetsHit: function

```
344    playerGetsHit: function(player) {
345
346        if(!player.invincible) {
347            if (player.data.playerNumber === 1) {
348                healthBar1.data.healthPoints -= 1;
349            } else {
350                healthBar2.data.healthPoints -= 1;
351            }
352
353            this.hurtSound.play();
354
355            player.invincible = !player.invincible;
356            player.tint = 0x8B0000;
357
358            player.alpha =  0.2;
359
360            this.game.time.events.add(Phaser.Timer.SECOND * 2, player.toggleInvincible, player);
361            this.game.time.events.add(Phaser.Timer.SECOND * 2, player.restoreTint, player);
362
363        }
364
365    },
```

This function checks whether a player is hit either by the spikes or an enemy. In order for it to work, the function is fed a parameter that is the reference to the player object. Inside, it checks whose player has been it and decreases the health-bar of the respective player by 1. Following, it sets a 2 second invincibility status so to give the player enough time to move to a safer place.

coin: function

This function checks the collision between a player and a collectable, it takes two parameters which are the player and the coin. If the player overlaps with a collectable, the player is awarded a certain amount of points and the collectable is destroyed.

```
367    coinHit: function(player, coin) {
368        if(player.data.playerNumber === 1) {
369            score1.data.score += 10;
370        }
371
372        if (player.data.playerNumber === 2) {
373            score2.data.score +=10;
374        }
375
376        this.collect_pointSound.play();
377
378        coin.destroy();
379    },
```

Power up – speed shoe

```
381    speed_shoeHit: function(player, shoe) {
382
383        player.runningSpeed = player.runningSpeed * 2;
384        this.game.time.events.add(Phaser.Timer.SECOND * 3, player.resetSpeed, player);
385
386        this.collect_powerupSound.play();
387
388        shoe.kill();
389
390        // Reset power up after 3 seconds
391        this.game.time.events.add(Phaser.Timer.SECOND * 3, this.resetShoe, shoe);
392
393    },
394
395    resetShoe: function() {
396
397        speed_shoeGroup.forEach(function(shoe) {
398            if (!shoe.live) {
399                shoe.reset(shoe.x, shoe.y);
400            }
401        });
402    },
```

The above two functions work hand in hand as speed_shoeHit needs resetShoe in order to work as intended. The first function doubles the player's speed when this overlaps with the shoe. Such speed is maintained for a few seconds before the player's speed is set to default again. Finally, a timer is set so that the power up can respawn after 3 seconds from when it was first picked up. It is worth noticing that resetShoe is a feature exclusive to level 2. As a matter of fact, Level 2 is designed in such a way that it would be impossible to complete if the power ups, in this case the shoe, was not made into a respawnable.

Power up – arrow

The two set of functions work the same way as the shoe power up. Again, the resetArrow is a function needed to complete the Level2. This time, instead of doubling the player's speed, the function allows the player to jump higher so to reach higher heights.

```
404    arrowHit: function(player, arrow) {
405
406        player.jumpingSpeed = player.jumpingSpeed * 1.5;
407        this.game.time.events.add(Phaser.Timer.SECOND * 3, player.resetJump, player);
408
409        this.collect_powerupSound.play();
410
411        arrow.kill();
412
413        // Reset power up after 3 seconds
414        this.game.time.events.add(Phaser.Timer.SECOND * 3, this.resetArrow, arrow);
415
416    },
417
418    resetArrow: function() {
419
420        arrowGroup.forEach(function(arrow) {
421            if (!arrow.live) {
422                arrow.reset(arrow.x, arrow.y);
423            }
424        });
425    },
```

heartHit: function:

```
427     heartHit: function(player, heart) {
428
429         if (player.data.playerNumber === 1) {
430             healthBar1.data.healthPoints +=3;
431
432             if (healthBar1.data.healthPoints > 5) {
433                 healthBar1.data.healthPoints = 5;
434             }
435         } else {
436             healthBar2.data.healthPoints +=3;
437
438             if (healthBar2.data.healthPoints > 5) {
439                 healthBar2.data.healthPoints = 5;
440             }
441         }
442
443         this.restore_healthSound.play();
444
445         heart.destroy();
446
447     },
```

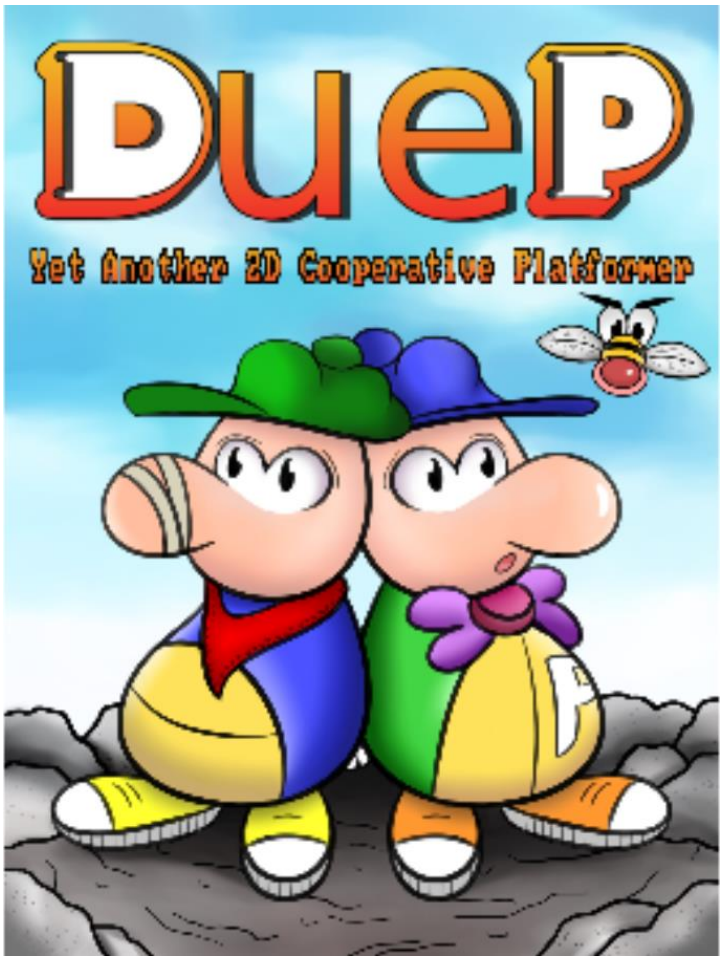
The heartHit functions checks the collision between a player and the heart object used to restore the player's health. Like in the previous functions, there is a need to make a distinction between player1 and player2.

goalHit: function

```
449     goalHit: function() {
450
451         winText.visible = true;
452         this.game.time.events.add(Phaser.Timer.SECOND * 3, this.restartGame, this);
453
454     },
455
456     restartGame: function() {
457
458         this.game.scale.setGameSize(700, 350);
459         this.game.world.setBounds(0, 0, 700, 350);
460         this.game.camera.setSize(700, 350);
461         this.state.start('MainMenu');
462     }
463
464 };
```

This function is triggered when either one of the two player overlaps with the checkpoint at the end of the second level. This consists of a congratulatory message being displayed for a few seconds the restarting of the game. In fact, the restart function restores the game size to its default value and tells to bring the game back to the main menu state.

Assets



The artwork shown above, used as a splash screen while the game is launching and the “Boot” state has been fully processed, was designed using a graphics tablet and the digital painting and drawing software CLIP STUDIO PAINT. Created by Daniele Di Modica, it is intended to indicate to players at least some visual cues regarding the game they would be playing. For example, the two playable characters are positioned near the very front, which is meant to symbolise the relatively important roles they play in regular gameplay. The game’s full title (subtitle included) can be seen at the “top” of the picture, along with a brief glimpse at what serves as the game’s main “enemy”, the previously mentioned malicious bee-like creatures.

As for the assets, those were created using a mouse and the graphics editor GIMP. The players’ assets idea come from the popular platformers Bobble Bubble, starring the twin Dragons *Bub* and *Bob* and Super Mario’s Characters *Mario* and *Luigi*.

The above images show the Player 1 and Player 2's sprite-sheets used for the characters' animations in the actual game.

Critical Review

The three areas the game could be improved the most are explained as follows:

Boss

A staple in many video games, including 2D platformers, bosses are typically included in video games whenever a rise in challenge and overall difficulty is needed, and adding such enemies in *DueP* would only have served to enrich the gameplay for the better. If one were to look at *DueP's* code, they could point out that a simple boss could be included in a level by, say, creating a new "Platformer.Enemy" object, increasing its size(represented in Phaser by increasing the horizontal and/or physical scale), giving it a "hitPoints" variable and adding code for making the enemy disappear when the value of this new variable is reduced to zero, but doing this would have only served to make the gameplay more "boring" in a way – these bosses would have had the exact same type of AI as the regular enemies, which would subsequently result in even slightly perceptive players defeating them with ease. A smarter method would be to construct different prefabs for individual bosses, which could have the additional benefit of giving these bosses unique types of AI behaviour.

Additional Levels

This critique ties into the "no bosses" critique to a degree – while adding new levels is more likely to be a positive in the grand scheme of things, there were concerns during the development of the game that developing too many levels would take away development time that could otherwise be used to implement/enhance gameplay mechanics and fix game-breaking bugs. In retrospect, at least one more level could have been implemented, perhaps as a small arena where the players have to engage one of the proposed "boss" enemies.

More enemy AI variation

The bee-like flying enemies in the game behave much like the *Goombas* and Koopa *Troopas* from *Super Mario Bros.* – they move in one direction for a few seconds, after which they promptly change direction and continue moving, unaffected, unless a player uses their character's projectile to defeat them. While in 2D platformers this is more than serviceable, in most cases they feature a large,

diverse group of enemies that all have unique behaviours: for example, there could be an enemy that walks on the ground, along with enemy that can fly, an enemy that can home in on the player, and an enemy that can summon spikes on top of its head for a few seconds, temporarily preventing players from attacking them from above.

Reference

Zenva. 2018. *HTML5 Phaser Tutorial – Top-Down Games with Tiled – Zenva | GameDev Academy*. [ONLINE] Available at: https://gamedevacademy.org/html5-phaser-tutorial-top-down-games-with-tiled/. [Accessed 04 May 2018].

Zenva. 2018. *Creating a Preloading Screen in Phaser 3 – Zenva | GameDev Academy*. [ONLINE] Available at: https://gamedevacademy.org/creating-a-preloading-screen-in-phaser-3/. [Accessed 04 May 2018].

Leshy Labs LLC. 2018. *Leshy SpriteSheet Tool - Online Sprite Sheet & Texture Atlas Packer*. [ONLINE] Available at: https://www.leshylabs.com/apps/sstool/. [Accessed 04 May 2018].

HTML5 Game Devs Forum. 2018. *make the camera follow 2 players / split screen - Phaser 2 - HTML5 Game Devs Forum*. [ONLINE] Available at: http://www.html5gamedevs.com/topic/13416-make-the-camera-follow-2-players-split-screen/. [Accessed 04 May 2018].

Photon Storm - photonstorm.com. 2018. *Phaser - Examples - Camera - Deadzone*. [ONLINE] Available at: https://phaser.io/examples/v2/camera/deadzone. [Accessed 04 May 2018].

HTML5 Game Devs Forum. 2018. *How to create multiple levels? - Phaser 2 - HTML5 Game Devs Forum*. [ONLINE] Available at: http://www.html5gamedevs.com/topic/5446-how-to-create-multiple-levels/. [Accessed 04 May 2018].

Photon Storm - photonstorm.com. 2018. *Phaser - Examples - Sprites - Out Of Bounds*. [ONLINE] Available at: http://phaser.io/examples/v2/sprites/out-of-bounds. [Accessed 04 May 2018].

HTML5 Game Devs Forum. 2018. *Restarting a Game? - Phaser 2 - HTML5 Game Devs Forum*. [ONLINE] Available at: http://www.html5gamedevs.com/topic/18744-restarting-a-game/. [Accessed 04 May 2018].

Stack Overflow. 2018. *javascript - createFromObjects() not working in phaser? - Stack Overflow*. [ONLINE] Available at: https://stackoverflow.com/questions/42039879/createfromobjects-not-working-in-phaser. [Accessed 04 May 2018].

Photon Storm - photonstorm.com. 2018. *Phaser - Shop - Box2d - World Bounds*. [ONLINE] Available at: https://phaser.io/examples/v2/box2d/world-bounds. [Accessed 04 May 2018].

Photon Storm - photonstorm.com. 2018. *Phaser - Examples - Tilemaps - Create From Objects*. [ONLINE] Available at: https://phaser.io/examples/v2/tilemaps/create-from-objects. [Accessed 04 May 2018].

HTML5 Games Workshop - Walking enemies. 2018. *HTML5 Games Workshop - Walking enemies*. [ONLINE] Available at: https://mozdevs.github.io/html5-games-workshop/en/guides/platformer/walking-enemies/. [Accessed 04 May 2018].

Lecture notes available on student central

Appendix 1: Game Concept Presentation