6. Phaser: a simple Platform Game

Design and development of web games (VJ1217), *Universitat Jaume I*

Estimated duration: 5 hours (+ 8–10 hours of exercises)

This sixth lab session completes the work with **Phaser**. Thanks to the software design of our platform game, you will learn to design the levels of your game using external files written in a structured, language-independent data format such as JSON, and we will show you how to load and read these files. Frequently, platform games have worlds bigger than the stage itself. **Phaser** has a built-in camera model to be able to cope with this feature. We employ this camera model in our game and you will learn its basic usage. Besides, platform games require a heavy use of Physics, therefore we will introduce some more concepts to complete your knowledge about the Arcade Physics engine of **Phaser**. Other important aspects of a game are the HUD —Head-Up Display— and the use of awesome fonts. We will also address them in this lab session. Our platform game uses external fonts, which are downloaded from Internet, and employs a new approach to the design of the HUD using tweens. Finally, this lab session provides some clues that you can follow to design and build successfully a platform-like game and introduces the basic concepts to debug **Phaser** games. In this document, a shallow review of the game's code given to you is made, outlining the main features which you should take into account to get a good understanding. It is very important that you inspect the code as you read the script and that you complete the *Your turn* boxes so that the game can be played. At the end, there are some exercises that you can do during the lab sessions or as homework.

Contents

1 Introduction

In the last two lab sessions, focused on **Phaser**, you got a wide knowledge of the main features and capabilities of this JavaScript game engine. This lab session explores some fundamental concepts in the programming of games and relates them with unseen properties and methods of **Phaser**:

- Use of external specific fonts for the texts of a game.
- Use of tweens to play short animations and to achieve awesome effects.
- Separation of the design and creation of the game content from the actual programming and the making of assets (images and audio) and the artwork.
- Use of a camera model to guide the player inside a game world bigger than the stage itself.
- Fine tuning the collisions by adjusting the bounding box of some sprites to achieve a better management of the behaviour of some characters in the game.

We address all of these concepts in the development of a simple platform game. The game is easy to play, as most arcade games: the player controls a character by means of the cursor keys and has to achieve a concrete goal to move on to the next level. The game ends when all the levels have been completed. There are enemies which try to prevent the player from accomplishing her goal.

Our game shows an initial welcome screen which allows the user to choose between several options as shown in Figure 1.

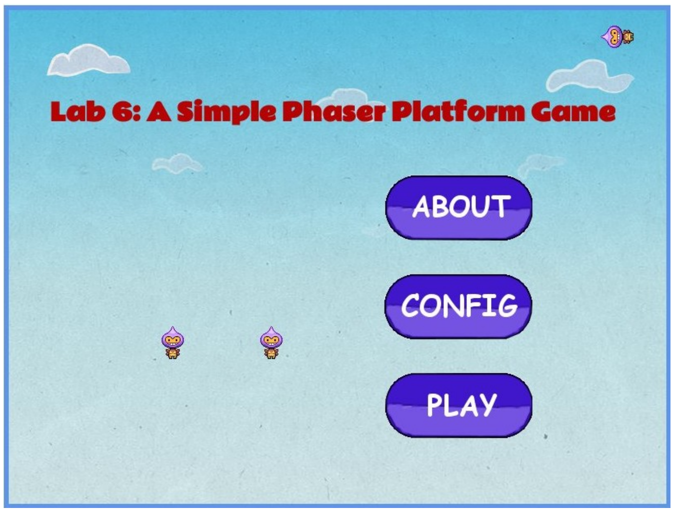
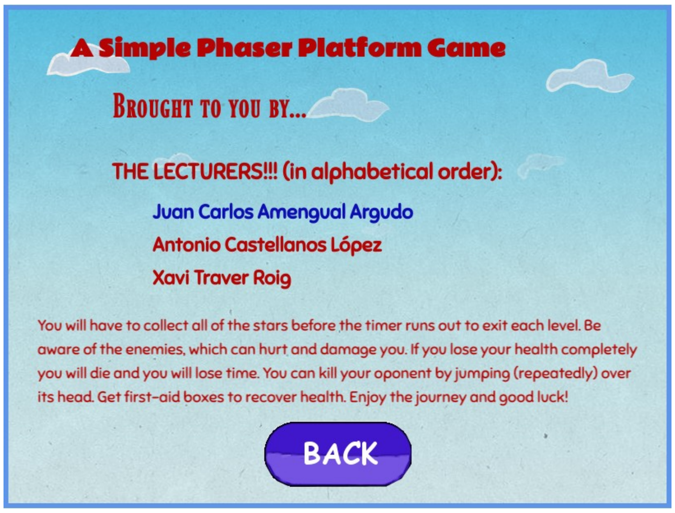


Figure 1: *A screenshot of the platform game's welcome screen.*

The first button from top to bottom (*ABOUT*) leads to a screen which shows the authors of the game and briefly explains its aim. This kind of screen is commonly known as the "about" screen. Figure 2 shows its contents.



A Simple Phaser Platform Game

Brought to you by...

THE LECTURERS!!! (in alphabetical order):

Juan Carlos Amengual Argudo

Antonio Castellanos López

Xavi Traver Roig

You will have to collect all of the stars before the timer runs out to exit each level. Be aware of the enemies, which can hurt and damage you. If you lose your health completely you will die and you will lose time. You can kill your oponent by jumping (repeatedly) over its head. Get first-aid boxes to recover health. Enjoy the journey and good luck!

Figure 2: *A screenshot of the screen that shows the credits and explains the purpose of the game.*

The second button (*CONFIG*) lets the user choose the difficulty of the game through another screen which shows three buttons to this respect —see Figure 3. Finally, the third button (*PLAY*) starts the game and lets the user play.

The player controls the main character —the "stars collector"— by means of the cursor keys: left and right to move the character in both directions, and up to jump. To reach the next level she has to collect all of the stars that

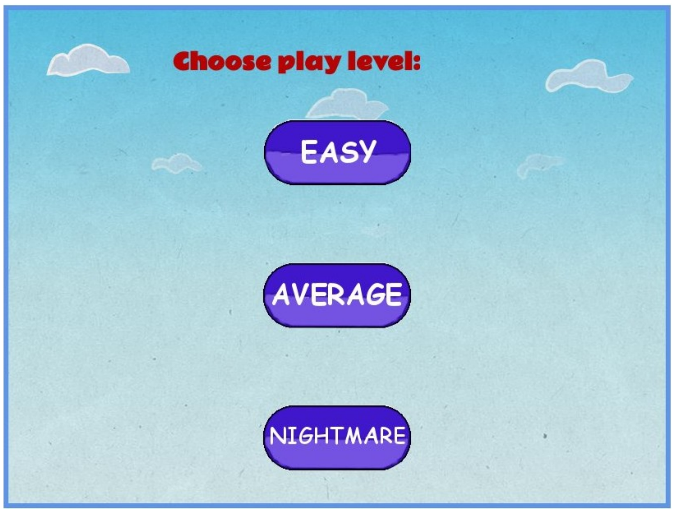


Figure 3: *A screenshot of the screen that lets the user choose the difficulty level of the game.*

are scattered throughout the ground and the platforms, and to drive the character to the exit sign before the time that is shown in the HUD runs out. There are enemies —the mice— that can damage the main character, taking health away from it. They are always walking —patrolling— and when the main character approaches them they will attack it.

If the health level in the HUD reaches its minimum, then the main character will be reset to its initial position and some seconds will be subtracted from the time remaining to achieve the goal. Health can be recovered by collecting the first-aid boxes that are placed in the ground and platforms. The player can kill a mouse by jumping repeatedly over its head.

Your turn 1

Download the archive `4students.zip` from *Aula Virtual* to your desktop. Then, unzip the archive and open the folder `A Simple Phaser Platform Game` in *Visual Studio Code*.
Next, observe the structure of the main folder (files and subfolders) and check the references to external files in `index.html`. Also, and for the moment, take a look at the code in the files `main.js`, `welcome.js`, `about.js`, and `configure.js`.
Finally, add the instruction required to start the `'welcome'` state at the end of the file `main.js`.

2 The structure of the game

The structure of this game is quite similar to the one used for the project developed in the previous lab session (shoot'em up game). The main folder contains the folders `assets`, `css`, and `js`. The first folder contains other three folders: `imgs` and `snds`, which hold, respectively, all of the images, buttons and sprites, and all of the sounds employed in the game, and `levels`, which holds the JSON files that make up the two levels of the game (see Section 6).

The second folder contains only a CSS file which sets

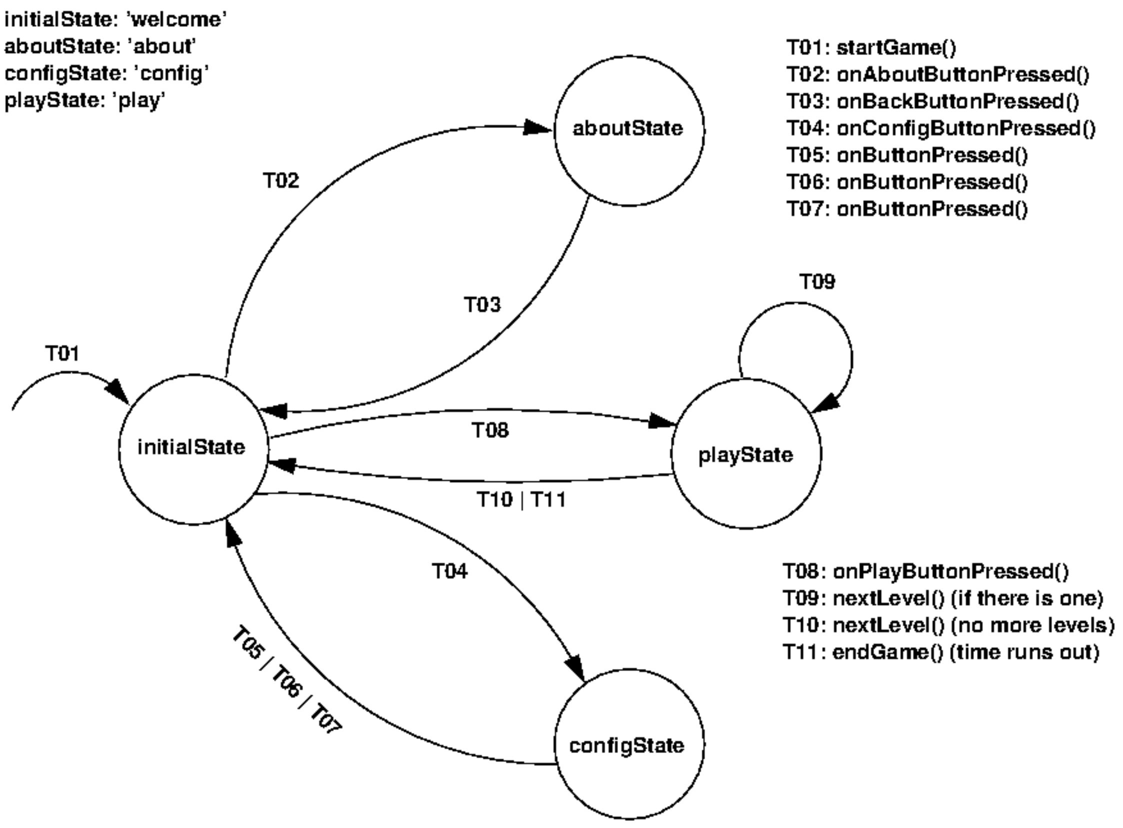


Figure 4: *A Finite State Machine which depicts the different states (screens) in the platform game that is studied in this lab.*

some properties for the canvas used by **Phaser**. Finally, the `js` folder holds the five JS files that comprise the whole code of the game:

- `welcome.js`, `about.js`, `configure.js`, and `play.js` define, respectively, the **Phaser** states `initialState`, `aboutState`, `configState`, and `playState`.
- `main.js` loads some external fonts through the **webfontloader** library —thus, making them available to **Phaser**— and, then, creates the **Phaser** game, adds the aforementioned states to it, and starts the `initialState`.

The last site file, `index.html`, sets the title of the web page, links the CSS file, defines a `<div>` element which will hold the game stage, and loads all of the JS files. Bear in mind that the load of these files should be done in the appropriate order: first, the libraries `webfontloader.js` and `phaser.js`, then `welcome.js`, `about.js`, `configure.js`, and `play.js`, and finally, `main.js`. Since the code in the last one requires the definition of the states `initialState`, `aboutState`, `configState`, and `playState`, their corresponding JS files have to be loaded beforehand. Otherwise, undefined name errors will be raised. Also, `play.js` uses some global variables defined in `welcome.js`. Besides, the values of some of these variables could be modified in `configure.js`. Therefore, both files are loaded after `welcome.js`.

As you already know from the previous lab session, placing the **Phaser** states into separate files helps to properly organise the code of the game, thus easing its development and maintenance. This way, the different phases of each state are grouped together within their corresponding state file.

It is also advisable to design a finite-state machine[1] which accounts for the different states of the game and the events that trigger the transitions between these states. Figure 4 shows the finite-state machine designed for this platform game.

You can see that each transition has been labelled with the name of the function in the code which triggers it. Also, the labels used in the game to refer to each state have been included (see the top left corner of the figure). This kind of diagram is very useful to get the complete picture of the working of a game, those developed using **Phaser** in particular, and it should be part of the documentation of any game.

With regard to `welcome.js`, `about.js`, and `configure.js` we would like to highlight that they only employ the `preload` and `create` phases of the states involved. They do not need to actually add code to the `update` phase since they do not require to perform calculations at regular intervals or move elements of the game in their respective screens (see Figures 1, 2, and 3).

Your turn 2

Add the required instructions to respectively start the states `'about'`, `'config'`, and `'play'` in each of the functions `on______ButtonPressed` of the file `welcome.js`.
Write the code for the function `onButtonPressed` in the file `configure.js`. To this end, you

[1] https://en.wikipedia.org/wiki/Finite-state_machine

should set the values of the variables damage, healthAid, secondsToGo, jumpsToKill, and playerDeathTimePenalty according to the button actually pressed. You need to use the operator === to ascertain which button has been pressed: btnEasy, btnAvg, or btnNgtm. These variables control the difficulty of the game: *easy*, *average*, or *nightmare*.
For the easy setting, their values have to be, respectively, DEFAULT_DAMAGE, DEFAULT_HEALTH, DEFAULT_TIME, DEFAULT_JUMPS_TO_KILL, and DEFAULT_PLAYER_DEATH_TIME_PENALTY. For the average setting, they will be DEFAULT_DAMAGE * 1.5, DEFAULT_HEALTH - 2, DEFAULT_TIME - 90, DEFAULT_JUMPS_TO_KILL + 1, and DEFAULT_PLAYER_DEATH_TIME_PENALTY + 10.
Lastly, their values for the nightmare setting must be DEFAULT_DAMAGE * 2, DEFAULT_HEALTH - 5, DEFAULT_TIME - 150, DEFAULT_JUMPS_TO_KILL + 3, and DEFAULT_PLAYER_DEATH_TIME_PENALTY + 25.
The last action to be carried out in this function is to start the 'welcome' state.
Run the project in *Code* and check the proper working of the welcome, about and configuration screens. Do not click on the *Play* button yet.

3 Using external fonts

Texts are an outstanding issue to consider when developing games. They are a key aspect for getting awesome looks in games. Think, for instance, in the fonts used in HUDs —an “ugly” font can ruin otherwise nice HUDs—, or in the feedback messages shown to the user, or in the dialogues in RPG-like (Role Playing Game) games.

The problem with games, and web games in particular, is that the developer has no control over, or even a clue on, the platform which the user will employ to play her game. It is not only that the users can use different browsers —Mozilla Firefox, Google Chrome, Microsoft Edge, Opera, Safari...— but also that they have distinct operating systems (OS) running on their computers —Linux, Windows, MacOS X. And the management of fonts depends on both the OS and the browser.

This means that even widely used font families, like Arial, Times New Roman, Courier, Helvetica, etc., could not be available to some users, according to the configuration of their browsers or of their OS desktop environments. Thus, in general, you can rely only on generic font families like serif, sans-serif, monospace or cursive when developing your games. It is guaranteed that these generic fonts will work since the system will always fall back to the fonts installed in the computer to provide a font matching the generic type employed.

Unfortunately, this does not guarantee the same look of the game's texts in every possible platform since the actual fonts employed could differ among users. Moreover, the generic font families are usually not awesome enough.

There is a solution for both problems, the (likely) unavailability of the fonts employed by the developer of the game and the use of awesome, stunning font families in web games. This solution consists in *downloading* the fonts from Internet, either from free available resources or from your own server, when your web game is loading.

In the first lab session you learnt to download and use Google fonts, freely available in Internet, in your web pages. Nevertheless, that scheme cannot be employed when using **Phaser**[2] since **Phaser** actually draws the texts in the canvas that creates for rendering. This means that you will have to load and to make available these fonts *before* starting **Phaser**.

We suggest to employ the **webfontloader**[3] JS library to overcome this issue. This library can be easily imported to your project in *Visual Studio Code* by means of the *cdnjs* extension just as we saw in the theory class. However, a workaround is required to start **Phaser**, since it is needed that **webfontloader** runs first and only when it has finished downloading and rendering the fonts then **Phaser** can be run.

The **webfontloader** library uses a loader to download the required fonts asynchronously. The configuration for this loader is defined by a global variable —an object— named WebFontConfig, which must be defined before you call the WebFont Loader. You have to employ this configuration variable to specify the font source provider and the font families to be loaded into your web pages.

In any case, you can call the global variable that stores this configuration object as you please, but in this case, you will have to explicitly call the load method passing this variable as an argument (we have named this object wfConfig). This is the approach that we have followed in main.js:

```
let game;

let wfConfig = {
    active: function () {
        startGame();
    },

    google: {
        families: ['Rammetto One', 'Sniglet']
    },

    custom: {
        families: ['FerrumExtracondensed'],
      urls: ["https://fontlibrary.org/face/ferrum"]
    }
};

WebFont.load(wfConfig);
```

Observe that we have defined a global variable, game, which will store the instance of the Phaser.Game class as we will see later.

Inside the wfConfig object we have used a set of predefined properties such as google or custom. The first one is used to download fonts from *Google* —such as the

[2]This would not be the case if you were developing a “pure” JavaScript game, without the help of external libraries.
[3]https://developers.google.com/fonts/docs/webfont_loader

ones employed in the first lab session. Note the use of the property `families` to specify the array of strings with the names of the desired font families.

There are also other font providers that can be used by declaring a predefined property, such as *TypeKit* (`typekit`), *Fonts.com* (`monotype`), or *Fontdeck* (`fontdeck`). If you want to indicate another font source provider, which is not in the list of known providers, then you will have to use the property `custom`, as shown in the code above. In this case, you will have to declare another array of strings, `urls`, indicating the URLs from where to download the desired fonts. Bear in mind that an URL can also be a path in your local hard drive —useful if it is a font that you have created or that you have previously downloaded.

The use of the `active` property deserves an explanation. The WebFont Loader provides an event system that developers can hook into. It gives you notifications of the font loading sequence in both CSS and JavaScript. One of such events is the `active` event, which is triggered when the fonts have rendered. We use this event to call the `startGame` function in `main.js`, which actually starts **Phaser** when the fonts are available for sure:

```
function startGame() {
  game = new Phaser.Game(800, 600, Phaser.CANVAS,
             'platformGameStage');
  // Welcome Screen
  game.state.add('welcome', initialState);
  // About Screen
  game.state.add('about', aboutState);
  // Config Screen
  game.state.add('config', configState);
  // Play Screen
  game.state.add('play', playState);
  // Add the instruction required to start the
  // 'welcome' state
}
```

There are another interesting events for which a developer can register a handler too, such as the `fontinactive` event, which is triggered if the font cannot be loaded and, therefore, it could be used to load an alternate font thus preventing the game from coming to a grinding halt.

WebFont Loader GitHub

↗ **webfontloader documentation**

See how

↗ **How to use the Google WebFont Loader**
↗ **Loading Web Fonts with the Web Font Loader**

webfontloader and the network

The code of `main.js`, as it currently reads, requires a working Internet connection to load the fonts employed in the game. If there is no active network in your computer the game will fail to run. This is not really a problem since every user who wants to play this web game needs an active Internet connection. However, this could be an inconvenience when you are developing the game and you need to test it. Bear this in mind when you are doing the exercises suggested in this document because you will need to connect your computer to the network to test your code. Alternatively, you could use the `fontinactive` event to load local alternative fonts or, even better, have a local copy of the font files.

Once the required fonts have been loaded and rendered, **Phaser** is started and they become available in our game. For instance, we have used the *Rammetto One* font in `welcome.js`:

```
function displayScreen() {
  // ...
  let textTitle = 'Lab 6: A Simple Phaser
                   Platform Game';
  let styleTitle = {
    font: 'Rammetto One',
    fontSize: '22pt',
    fontWeight: 'bold',
    fill: '#b60404'
  };
  game.add.text(50, game.world.height / 6,
       textTitle, styleTitle);
  // ...
}
```

The three external fonts loaded are used in our platform game. In particular, we suggest that you look at the code in `about.js` since this is the screen of the game where more fonts are used:

```
function showInstructions() {
  // ...
  let credits = 'Brought to you by...\n';
  game.add.text(125, game.world.height / 6,
    credits, {
      font: 'bold 26pt FerrumExtracondensed',
      fill: '#b60404'
  });

  let msgAuthors = 'THE LECTURERS!!! (in
                    alphabetical order):';
  let styleAuthors = {
    font: 'bold 20pt Sniglet',
    fill: '#b60404'
  };
  game.add.text(125, game.world.height / 6 + 60,
                msgAuthors, styleAuthors);
  // ...
}
```

Your turn 3

Employ *Code* to create a folder named `fonts` inside the folder `assets`. Download an awesome font from Internet on that folder (a `.ttf` or `.otf` file for instance).
Then, load this font in the game —you will have to modify `main.js`— and use it to write the instructions of the game in `about.js` instead of the *Sniglet* font.

3.1 Text Properties and Events

Precisely, there are a couple of relevant issues in the code of the `about.js` file that we would like to address. The first one has to do with the fact that we can also set event handlers for texts in **Phaser**. We enable input in the text object via the `inputEnabled` property (`inputEnableChildren` actually since we are using a group). Observe that in the code of `about.js` we have registered a handler for the event that is triggered when the mouse is over some of the texts shown —the name of the lecturers— which makes them change their foreground colour (see also Figure 2):

```
let authors;
// ...
function showInstructions() {
  // ...
  authors = game.add.group();
  authors.inputEnableChildren = true;
  authors.onChildInputOver.add(overText, this);
  authors.onChildInputOut.add(outText, this);
  let styleSingleAuthor = {
    font: 'bold 18pt Sniglet',
    fill: '#b60404'
  };

  let author=game.add.text(175,game.world.height
        / 6 + 110, 'Juan Carlos Amengual Argudo',
        styleSingleAuthor);
  authors.add(author);
  // ...
}
// ...
function overText(text, pointer) {
  text.fill = '#0e0eb3';
}

function outText(text, pointer) {
  text.fill = '#b60404';
}
```

There are other events a text object can listen to. Just read the **Phaser** documentation[4].

The second issue has to do with some properties that can be used to place long texts in the screen. For instance, the instructions given in the about screen to play the game are a single text which uses the `wordWrap` and `wordWrapWidth` properties to wrap the text inside the margins, and the `boundsAlignH` and `boundsAlignV` properties to properly align it:

```
function showInstructions() {
  // ...
  let instructions = 'You will have to collect
      all of the stars before the timer runs
      out to exit each level. ';
  // ...
  let instrucText = game.add.text(0, 0,
      instructions, { font: '14pt Sniglet',
                      fill: '#b60404' });
  instrucText.setTextBounds(30,
      game.world.height - 170,
      game.world.width - 60);
  instrucText.boundsAlignH = 'center';
  instrucText.boundsAlignV = 'middle';
  instrucText.wordWrap = true;
  instrucText.wordWrapWidth =
      game.world.width - 60;
  // ...
}
```

4 Tweens

Tweens are a way to animate properties of an object from one value to another over a specific lapse of time. In this sense, they can be used to play short simple animations. In general, a tween allows you to alter one or more properties of a target object over a defined period of time. This can be used for things such as alpha fading sprites, scaling them or simple motion. You will have to employ the methods `Tween.to` or `Tween.from` to set up the tween values. Among the most widely used "tweenable" properties of objects, we have:

- The position in both the X and Y axes.
- The `scale.x` and `scale.y`, but they need a separate tween for the `object.scale` property acting as the target.
- The `alpha`, which controls the transparency of the object. It is useful to fade in (0 to 1) or fade out (1 to 0) objects.

Phaser tweens were already introduced in the fourth lab session and are similar to the CSS animations that you studied in the first lab session: they need to know the element to which we want to apply the animation and the corresponding frame(s), and to set the properties we want to change over time. Tweens carry out some interpolation to play the animation smoothly. To this end, **Phaser** provides several "easing" algorithms that can be selected by means of the `Phaser.Easing`[5] class. Since they may influence the quality of the animation, we suggest that you try several methods and choose the one which best matches your needs.

We have used tweens in the initial welcome screen, `welcome.js`, and in the game itself, `play.js`. The code of `welcome.js` shows the potential of tweens:

[4] https://phaser.io/examples/v2/text/text-events

[5] https://photonstorm.github.io/phaser-ce/Phaser.Easing.html

```
let mainTween, downTween1, downTween2;
// ...
function displayScreen() {
  // ...
  mainTween = game.add.tween(hero3)
    .to({y: 32}, 2000, Phaser.Easing.Linear.None)
    .to({angle: -90}, 500,
         Phaser.Easing.Linear.None)
    .to({x: game.world.width - (32 * 2)}, 4000,
         Phaser.Easing.Linear.None);
  mainTween.delay(3000);
  mainTween.loop(true);
  mainTween.start();

  downTween1 = game.add.tween(hero1.scale)
    .to({x: 5, y: 5}, 1500,
         Phaser.Easing.Cubic.Out)
    .to({x: 1, y: 1}, 1500,
         Phaser.Easing.Cubic.Out);
  downTween1.onComplete.add(
       onDownTweenCompleted, this);
  downTween2 = game.add.tween(hero2)
    .to({alpha: 0.05}, 2500,
         Phaser.Easing.Linear.None)
    .to({alpha: 1.0}, 2500,
         Phaser.Easing.Linear.None);
  downTween2.onComplete.add(
       onDownTweenCompleted, this)
  downTween1.start();
  // ...
}

function onDownTweenCompleted(object,tween) {
  if (tween === downTween1)
      downTween2.start();
  else
      downTween1.start();
}
```

Observe that the three tweens declared —mainTween, downTween1, and downTween2— indeed consist of several tweens "chained" one after the other. This is a powerful feature of **Phaser** tweens, since it lets us carry out complex animations more easily.

These tweens have an image stored in the dude.png asset as their target. The file dude.png is actually a spritesheet which holds several frames of an animation. However, we only have added the frame number four to the stage —observe the last argument passed to game.add.sprite— in welcome.js —three different times since we want to add the same image to the stage three times:

```
function displayScreen() {
  // ...
  let hero1 = game.add.sprite(game.world.width
      / 4, game.world.height - 200, 'hero', 4);
  hero1.anchor.setTo(0.5, 0.5);
  // ...
}
```

The chained tweens stored in mainTween move first the image upwards. Then, they rotate the image 90 degrees counterclockwise and, finally, they move it to the right. This tween —actually a sequence of tweens— is run by calling its start method. But, previously, we have called the methods delay and loop to, respectively, run the tween three seconds after the call to start and set it to loop until explicitly stopped.

The tween downTween1 has the property hero1.scale as its target. Remember that we said before that the scale property is a special case that requires its own tween. This tween simply scales the image to be five times greater —both in the X and Y axes— and then reverses the scale. The remaining tween, downTween2, performs a simple fade out/fade in effect on the target image.

You can see that we simply run the downTween1 tween by calling its start method. However both tweens are run in an endless loop one after the other. How could we do this? The answer is simple: by means of an event. The tweens in **Phaser** can handle some events as you can read in the documentation[6]. One of these events is onComplete which is triggered when the tween and all of its children completes (ends). We have set the same handler for this event in both tweens: onDownTweenCompleted. You can see that the code of this function simply calls the start method of a tween when the other has completed.

A pair of tweens will be used in play.js, but they will be explained later, in Sections 5.3 and 7, respectively.

5 The game

The whole code of the actual game is stored in the file play.js, which corresponds to the playState **Phaser** state. This is the only state of the game that hooks code into three phases: preload, create, and update. When **Phaser** runs the preload phase of this state, the function loadPlayAssets will be called:

```
function loadPlayAssets() {
  loadSprites();
  loadImages();
  loadSounds();
  loadLevel(levelToPlay);
}
```

This function simply calls other functions that will load all of the sprites, images, and sounds employed in the game. There is nothing new to comment about these methods —you have used similar functions in previous lab sessions. In fact, we only want to draw your attention to the loadLevel function, which is used to read the contents of a level of the game so that we can be able to place them later in the stage. We will give the details in Section 6.

In the create phase, **Phaser** will call the createLevel function. There are several interesting issues in this function, but, besides those that will be addressed in Sections 5.1, 5.2, and 5.3, we only want to highlight the following:

1.- The variable exitingLevel declared in play.js is initialised to false. This variable is set to true when

[6] https://photonstorm.github.io/phaser-ce/Phaser.Tween.html

the player has collected all of the stars and, therefore, is allowed to leave the level. Thus, we prevent our program from playing the sound for leaving the level more than once.

2.- A body is enabled for the exit sign so that the player can collide with it, thus leaving the level:

```
exit = game.add.sprite(game.world.width-100,
             game.world.height-64, 'exit');
game.physics.arcade.enable(exit);
```

We check a possible collision between the player and the exit sign at the end of the `updateLevel` function. The function `endLevel` will indeed check whether the player can leave the level:

```
function updateLevel() {
 // ...
 if (!exitingLevel)
   game.physics.arcade.overlap(player, exit,
                endLevel, null, this);
}
// ...
function endLevel() {
  if (totalNumOfStars === 0) {
    exitingLevel = true;
    resetInput();
    soundLevelPassed.play();
    stopPlayer();
    game.time.events.remove(timerClock);
    game.time.events.add(4000, nextLevel,
       this);
  }
}
```

If `exitingLevel` did not exist, the function `endlevel` would be called repeatedly from the `updateLevel` function during the four seconds that the game is paused until `nextLevel` is called since `updateLevel` is the function called in the `update` phase, which, in turn, is run in the update/render loop of **Phaser** until another state is started. This would cause undesirable effects like playing the sound `soundLevelPassed` more than once and, thus, overlapping audio playbacks.

3.- The variable `totalNumOfStars` declared in `play.js` is initialised to 0. This initialisation is run before the beginning of each level of the game since it is done in the `create` phase.

4.- All of the platforms of a level of the game, including the ground, will be added to the `platforms` group, which is created here. Also, physics is enabled for all the platforms:

```
platforms = game.add.group();
platforms.enableBody = true;
```

5.- The variable `remainingTime` declared in `play.js` is initialised to the value of `secondsToGo`, which will be explained later. This variable will hold at all times the number of seconds left to complete the level. Again, this initialisation must be done in the `create` phase so that the timer is reset before the beginning of each level.

6.- The player sprite —the “stars collector”— is added to the game in the coordinates set in the JSON file which holds the level data (see Section 6). Then, physics is enabled for it and we set some properties so that the sprite has a slight bounce in its movement, and a gravity, and can collide with the bounds of the world. Also, we add animations to walk right and left.

```
// Create player. Initial position according
// to JSON data
player = game.add.sprite(
   levelConfig.collectorStart.x,
   game.world.height -
   levelConfig.collectorStart.y,
   'collector');
player.anchor.setTo(0.5, 0.5);
game.physics.arcade.enable(player);

//  Player physics properties. Give the
// little guy a slight bounce.
player.body.bounce.y = 0.2;
player.body.gravity.y = BODY_GRAVITY;
player.body.collideWorldBounds = true;
// ...
//  Our two animations, walking left and
// right.
player.animations.add('left', [0, 1, 2, 3],
   10, true);
player.animations.add('right', [5, 6, 7, 8],
   10, true);
```

7.- Finally, we enable the cursor keys and set a timer to update the time remaining to complete the level at each second.

```
cursors = game.input.keyboard.
          createCursorKeys();
timerClock = game.time.events.loop(
     Phaser.Timer.SECOND, updateTime, this);
```

The variables `damage`, `healthAid`, `secondsToGo`, `jumpsToKill`, and `playerDeathTimePenalty` are actually declared and initialised in `welcome.js`:

- `damage` — value to be subtracted to the health bar when the player collides with the left or right side of an enemy.
- `healthAid` — value to be added to the health bar when the player collects a first-aid box.
- `secondsToGo` — number of seconds granted to complete a level.
- `jumpsToKill` — number of bumps that has to be given on the head of an enemy to kill it.
- `playerDeathTimePenalty` — number of seconds that will be subtracted from the time remaining to complete a level when the health bar is exhausted.

Their values will be used to set the difficulty of the game. Therefore, they will be increased or decreased suitably in the state `configState` according to the level chosen by the player. In fact, you already did that in the box *Your turn 2* when you changed the code in `configure.js`.

The core of the game is run in the update phase of the state playState, that is, in the updateLevel function of play.js. First of all, we have to detect collisions of the player with the platforms. When this happens, we have to check whether the player is standing on a platform[7]. If so, we will store the platform where the player stands in the variable playerPlatform (hitPlatform is a boolean which tells us whether there was a collision with any platform):

```
let hitPlatform = game.physics.arcade.collide(
    player, platforms, playerInPlatform, null,
    this);
// ...
function playerInPlatform(player, platform) {
  if (player.body.touching.down)
    playerPlatform = platform;
}
```

Remember that the method arcade.collide of **Phaser** passes the sprites involved in the collision as parameters to the callback function (playerInPlatform in this case). It happens the same with arcade.overlap.

Next, we check for the collisions of the stars and first-aid boxes with the platforms. As we will see in Section 5.2 physics is enabled for both of them, and they have a gravity. We want them to stand gently on the platforms:

```
game.physics.arcade.collide(stars, platforms);
game.physics.arcade.collide(firstAids,
     platforms);
```

Then, we have to check whether the player collides with a star or a first-aid box. Observe that we have used arcade.overlap instead of arcade.collide. Could you tell the difference? The callback functions, collectStar and getFirstAid, play the corresponding sound, kill the item, and update, respectively, the number of stars and the health bar:

```
game.physics.arcade.overlap(player, stars,
     collectStar, null, this);
game.physics.arcade.overlap(player, firstAids,
     getFirstAid, null, this);
// ...
function collectStar(player, star) {
  soundCollectStar.play();
  star.kill();
  totalNumOfStars -= 1;
}

function getFirstAid(player, aid) {
  soundGetAid.play();
  aid.kill();
  healthValue = Math.min(MAX_HEALTH,
        healthValue + healthAid);
  updateHealthBar();
}
```

After that, we have to check whether the player has collided with any enemy (mouse), but this code will be explained in Section 5.3. The rest of the code of the updateLevel function is similar to other code used in theory classes or in previous lab sessions. Remember that we explained the purpose of the variable exitingLevel at the beginning of this section. Nevertheless, we would like to comment some issues:

- The variable toRight tells us whether the main character (the collector) is moving from left to right or the other way round. It will be used later whenever the player collides with an enemy (see Section 5.3).
- The function stopPlayer simply stops the player's animation which is currently playing —if any— and sets the frame to be displayed when neither the left nor the right cursor are being pressed.
- If the player is jumping then playerPlatform is set to undefined since the player is not standing actually on any platform.

You can take a look at this platform game in Figure 5, which shows a snapshot of the first level of the game.

5.1 The world and the camera

As we already saw in Section 3, when the instance of Phaser.Game is created, the stage used by **Phaser** is 800 pixels wide and 600 pixels high. However, the world of the game is actually bigger (1100×825) than the stage (800×600), something that happens frequently in this kind of games. When this is the case, it is advisable that the background is loaded as a tile (sprite) even if it is actually an image:

```
function createLevel() {
  // ...
  game.world.setBounds(0, 0, 1100, 825);
  let bg = game.add.tileSprite(0, 0,
      game.world.width, game.world.height,
      'bgGame');
  bg.scrollFactorX = 0.7;
  bg.scrollFactorY = 0.7;
 // ...
}
```

Observe that the game.world.setBounds method changes the world's size. The values for the scroll factor both in the X and Y axes —less than 1— provide a smooth scrolling of the tile (sprite).

How can we get that the stage scrolls automatically as the player moves near its limits? The answer to this question is simple enough. First of all, you have to bear in mind that **Phaser** implements a *camera* model inside the world. This means that you can set a camera to follow the main character as it moves on the screen. Provided that you have previously set scroll values for the background —as it is the case— the stage will move to reach the world's bounds as required. We have set the camera in the create phase —createLevel function— of play.js:

```
// Camera follows the player inside the world
game.camera.follow(player);
```

[7]Remember that the ground is also a platform.

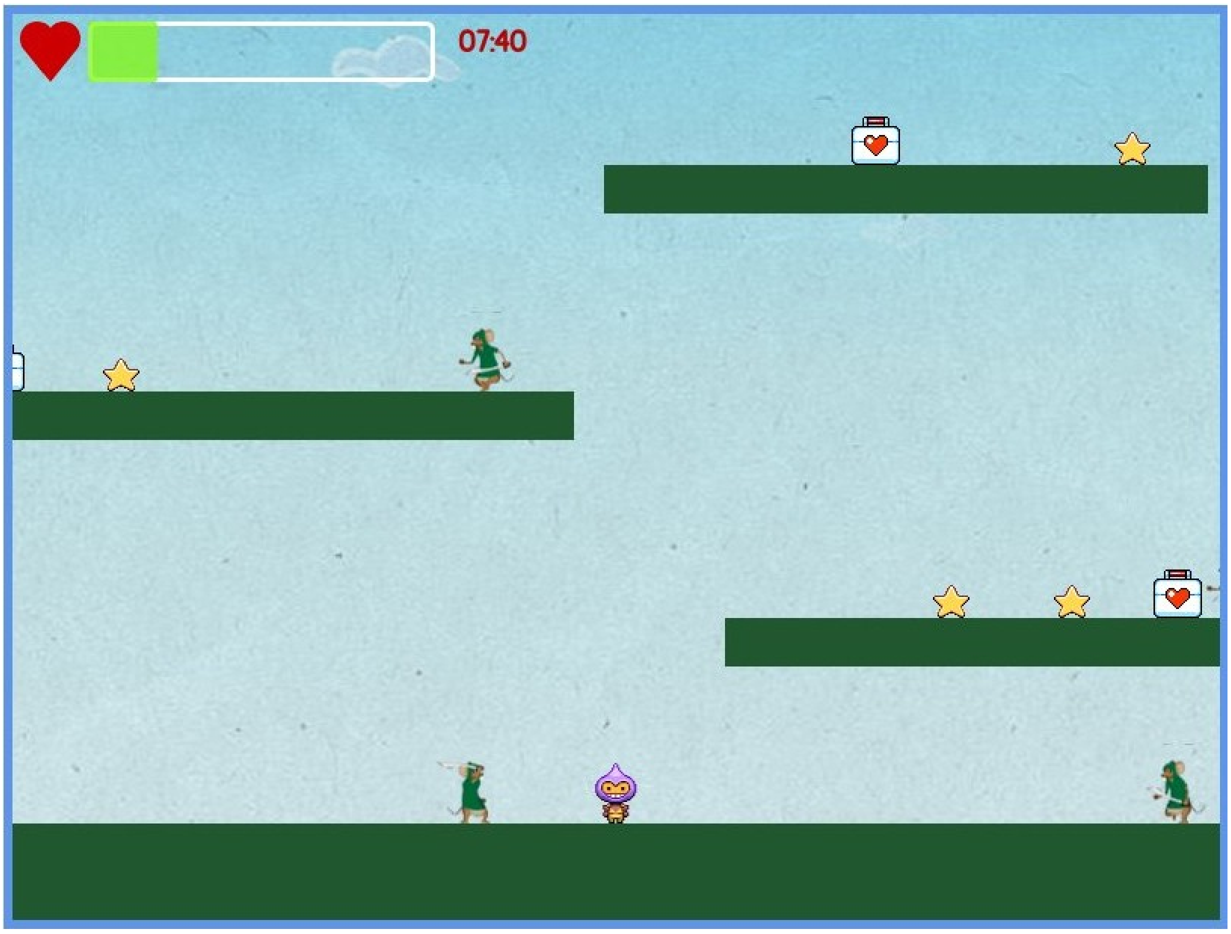


Figure 5: *A snapshot of the first level of our platform game.*

But keep in mind that the game will return to the welcome screen if either there are no more levels to play or the time granted to complete the level has run out (see Figure 4). The former case will be addressed in Section 6. The latter is detected inside the function `updateTime`, which is the callback set to run each second in the timer stored in the variable `timerClock`:

```
function updateTime() {
  remainingTime = Math.max(0,remainingTime-1);
  hudTime.setText(setRemainingTime(
     remainingTime));
  if (remainingTime === 0) {
    resetInput();
    soundOutOfTime.play();
    stopPlayer();
    game.time.events.remove(timerClock);
    game.time.events.add(2500, endGame, this);
  }
}
// ...
function endGame() {
  clearLevel();
  goToWelcome();
}
```

If the game returns to the welcome screen we will have to reset the world's bounds so that they match the limits of the stage. This is something done in the `goToWelcome` function:

```
function goToWelcome() {
  game.world.setBounds(0, 0, game.width,
                       game.height);
  game.state.start('welcome');
}
```

5.2 Stars and first-aid boxes

The stars and the first-aid boxes are created in the functions `createStars` and `createAids`, respectively. Both of them are called inside the `createLevel` function. We have followed a similar approach to that used in the previous lab session with the lasers and the UFOs: to create a pool of objects, which can be reused if requested, inside a group.

```
function createAids() {
  firstAids = game.add.group();
  firstAids.enableBody = true;
  firstAids.createMultiple(MAX_AIDS, 'aid');
  firstAids.forEach(setupItem, this);
}

function createStars() {
  // similar to the code above
}

function setupItem(item) {
  item.anchor.setTo(0.5, 0.5);
  item.body.gravity.y = BODY_GRAVITY;
}
```

Your turn 4

Write the code of the function `createStars`. Bear in mind that the number of stars to be created in the pool is defined by the constant `MAX_STARS` and that they are identified by the string `'star'`. This group will be stored in the variable `stars`.

Observe that we have enabled a body (physics) and set a gravity for each star and each first-aid box created. Then, whenever it is required to add a star or a first-aid box to the stage either the function `setupStar` or the function `setupAid` will be called:

```
function setupAid(aid, floorY) {
  let item = firstAids.getFirstExists(false);
  if (item)
    item.reset(aid.x, floorY -
               AID_STAR_Y_OFFSET);
}

function setupStar(star, floorY) {
  let item = stars.getFirstExists(false);
  if (item) {
    item.reset(star.x, floorY -
               AID_STAR_Y_OFFSET);
    totalNumOfStars += 1;
  }
}
```

Note that both items are placed slightly above the Y axis with regard to their location —`AID_STAR_Y_OFFSET`. Thus, we can see them landing gently on their respective platforms after appearing in the stage, due to the gravity.

5.3 The attack of the mice

The player has some enemies placed in each level to hinder its completion. These enemies are pictured by mice which are constantly moving, and can attack the main character if it is near enough (euclidean distance is computed by means of the method `Phaser.Math.distance`). Unlike the stars and the first-aid boxes, these game elements show a behaviour —they patrol and they attack. Besides, a very short animation —in fact, a tween— is played whenever the player jumps on their heads. Therefore, a class is used to store their attributes and implement their behaviour:

```
class Enemy {
  constructor(spritesheet, tween, plat, right,
              limit, hits,
              isPatrolling = true) {
    this.sprite = spritesheet;
    this.flash = tween;
    this.platform = plat;
    this.faceright = right;
    this.stepLimit = limit;
    this.origX = spritesheet.x;
    this.hitsToBeKilled = hits;
    this.isPatrolling = isPatrolling;
  }
  // ...
  patrol() {
    // ...
  }

  attack(thePlayer) {
    // ...
  }
}
```

For each mouse, we store, from top to bottom, (a) its **Phaser** sprite, (b) the short tween animation, (c) the platform where it stands, (d) a boolean that tells whether it is originally facing right —as the `enemySprite.png` spritesheet is— or not, (e) the maximum distance that the mouse will walk originally, (f) its original X coordinate, (g) the number of hits on its head required to be killed, and (h) a boolean that tells whether the mouse is walking —patrolling— or not. Only the first six values are required to be passed as arguments when creating an instance.

The mice are stored into the array `enemies`. The function `setupEnemy` will be called whenever an enemy is added to the stage. It receives two parameters: an object with two properties (its X coordinate and a number that tells whether it is facing right or not) and the platform —indeed a sprite— where it stands.

Bear in mind that if the mouse is originally facing right, `faceright` will be set to `true`, and its rightmost X coordinate will be stored in `stepLimit` and set as the minimum between the maximum distance that a mouse can walk —`ENEMY_STEP_LIMIT`— and its platform's edge, and its leftmost X coordinate will be its original position —`origX`. When the mouse is originally facing left, the opposite applies.

At the beginning of this function, the sprite for the mouse is added to the stage and the above explained calculation is performed. Since the anchor point is located in the centre of the sprite, two constants —`ENEMY_X_OFFSET`, which is equal to half the width of the sprite, and `ENEMY_Y_OFFSET`, which is equal to half the height of the sprite— are used to properly place the sprite.

```
let theEnemy = game.add.sprite(enemy.x, plat.y -
                  ENEMY_Y_OFFSET, 'enemy');
theEnemy.anchor.setTo(0.5, 0.5);
if (enemy.right === 0) {
  theEnemy.scale.x = -1;
  isRight = false;
  limit = Math.max(Math.max(0, plat.x) +
    ENEMY_X_OFFSET, enemy.x-ENEMY_STEP_LIMIT);
} else {
  isRight = true;
  limit = Math.min(Math.min(plat.x + plat.width,
    game.world.width) - ENEMY_X_OFFSET,
    enemy.x + ENEMY_STEP_LIMIT);
}
```

Then, a “chained” tween is created to carry out a very short “fade out, fade in” effect that will be played whenever the mouse is hit on its head:

```
let flash = game.add.tween(theEnemy).to(
            {alpha: 0.0}, 50,
            Phaser.Easing.Bounce.Out)
      .to({alpha: 0.8}, 50,
           Phaser.Easing.Bounce.Out)
      .to({alpha: 1.0}, 50,
           Phaser.Easing.Circular.Out);
```

Next, physics is enabled for the sprite, some properties of its body are set, and two animations are added: one to be used when the mouse attacks —'swing'— and the other to be used when the mouse is walking —'run'.

Finally, an instance of Enemy is created and inserted into the array enemies:

```
let newEnemy = new Enemy(theEnemy, flash, plat,
                  isRight, limit, jumpsToKill);
enemies.push(newEnemy);
```

Whenever we want a mouse to be walking (i.e., patrolling), the method patrol of the class Enemy will be called. It receives no parameters at all. Its code is simple:

- A velocity is given to the sprite's body and the corresponding animation is played. Observe that the velocity will be negative (the mouse will move to the left) or positive (the mouse will move to the right) depending on whether the sprite is flipped or not.
- If the mouse reaches its "walking limit" either to the right or to the left, its position is reset and the sprite is flipped by changing the scale in X. Remember that if the mouse is originally facing right, its rightmost position has been stored in stepLimit and its leftmost position is its original X coordinate, origX, and that if the mouse is originally facing left, the opposite applies.

If the mouse is not patrolling, then it is attacking the player. In this case, the method attack will be called. It receives only one parameter: the player's sprite. This method is slightly more complex than patrol. First of all, it has to place the mouse in the right direction to face the player:

```
if (this.sprite.body.x < thePlayer.body.x) {
      this.sprite.scale.x = 1;
      playerAtRight = true;
} else {
      this.sprite.scale.x = -1;
      playerAtRight = false;
}
```

Next, it increases the velocity of the mouse's sprite and plays the corresponding animation:

```
this.sprite.body.velocity.x = Math.trunc(
      ENEMY_VELOCITY * 1.4) *
      this.sprite.scale.x;
this.sprite.animations.play('swing');
```

Then, and since the mouse will chase the player when it is attacking, the function properly changes the values of the mouse's "walking limits" —stepLimit and origX— taking into account the position of the edges of the platform where the mouse stands[8]. Observe that this will have an effect on the mouse when it returns to patrolling. Every time a mouse attacks the player, its "patrolling range" is changed forever.

Finally, this function checks whether the mouse has reached its "walking limits". If so, its body's velocity will be set to 0.

Now it's time to come back to the updateLevel function. The part of the code which checks collisions between the player and the enemies remains to be explained. We have to iterate over the array of enemies not only to check for possible collisions but also to set each enemy's behaviour —either it patrols or it attacks.

We have to find out whether a mouse has to attack or to patrol. To this end, we need to know whether the player is close enough to the mouse. Therefore, we compute the euclidean distance between the player's sprite and the mouse's sprite by means of the Phaser.Math.distance function. A constant, ENEMY_DISTANCE_ATTACK, is used as a threshold for defining what is regarded as "close" or not. The distance will be computed only if the mouse is on the same platform where the player stands. If they stand on different platforms or the player is jumping —it is in the air— then the mouse will not attack:

```
if (enemies[i].platform === playerPlatform) {
  dist = Phaser.Math.distance(enemies[i].sprite.
           body.x, enemies[i].sprite.body.y,
           player.body.x, player.body.y);
  if (Math.round(dist) <= ENEMY_DISTANCE_ATTACK)
    enemies[i].setIsPatrolling(false);
  else
    enemies[i].setIsPatrolling(true);
} else
  enemies[i].setIsPatrolling(true);
```

Then, we set the behaviour of the mouse by calling the corresponding method of the class:

```
if (enemies[i].getIsPatrolling())
  enemies[i].patrol();
else
  enemies[i].attack(player);
```

And, finally, if there is a collision, we will call the function playerVsEnemy:

```
if (game.physics.arcade.collide(player,
         enemies[i].sprite))
  playerVsEnemy(player, enemies[i].sprite,
              enemies[i], i);
```

This function has to check whether the collision happened on the top of the mouse's sprite or not —it uses enemySprite.body.touching.up to this end. If it has happened on top, this function:

[8]The mice are not allowed to fall from their platforms.

- Plays the corresponding sound and the short tween of the mouse's sprite.
- Slightly shifts the player according to the value of the `PLAYER_COLLIDE_OFFSET_X` constant.
- Updates the number of hits to kill the mouse and if the mouse has to be killed it will destroy the sprite[9] and remove the object from the array `enemies`.

Otherwise, this function:

- Plays the damaged player's sound.
- Updates the value of the health bar.
- Slightly shifts the player according to the value of the `PLAYER_COLLIDE_OFFSET_X` constant —this shift is different: observe the use of the boolean variable `toRight`.
- Calls `resetPlayer` if the player's health is over. This function simply stops the player, resets its position in the level, updates the timer, and refills its health level:

```
function resetPlayer() {
  stopPlayer();
  player.x = levelConfig.collectorStart.x;
  player.y = game.world.height -
     levelConfig.collectorStart.y;
  remainingTime = Math.max(0, remainingTime -
     playerDeathTimePenalty);
  healthValue = MAX_HEALTH;
  updateHealthBar();
}
```

In any case, the player is slightly shifted according to the value of the `PLAYER_COLLIDE_OFFSET_Y` constant.

6 Designing levels (JSON)

It is desirable to separate the design and preparation of the game contents from the game programming and development. We suggest that you use a structured, programming language-independent data format such as JSON or XML to design the contents of your games —remember that we already introduced the use of JSON in the fourth lab session. In our case, the game contents are the elements present in the two levels that we have prepared for our platform game. We have written a JSON file to describe the contents of each level. For example, below you have part of the contents of the file `level01.json`:

```
{
  "collectorStart": {"x": 32, "y": 90},
  "ground": {"enemies": [ {"x": 240, "right": 0},
                          {"x": 880, "right": 0}
           ],
           "aids": [ {"x": 470}, {"x": 955}
           ],
           "stars": [ {"x": 100}, {"x": 190},
                                           {"x": 840}
                    ]
  },
  "platformData": [
    {"x": 0, "y": 350,
     "enemies": [ {"x": 200, "right": 1}
     ],
     "aids": [ {"x": 20}
     ],
     "stars": [ {"x": 100}
     ]
    },
// ...
    {"x": 800, "y": 650,
     "enemies": [ {"x": 900, "right": 0}
     ],
     "aids": [
     ],
     "stars": [ {"x": 1050}
     ]
    }
  ]
}
```

JSON syntax

Introducing JSON

This file contains a single object —the level—, which has a series of name/value pairs. Names, which are actually strings, are written in quotation marks. Each name plays the role of a key in a dictionary. In JSON, an object is an unordered set of name/value pairs. An object begins with "{" (left curly bracket) and ends with "}" (right curly bracket). Each name is followed by a colon and the name/value pairs are separated by commas. Each value can be, in turn, another object or an array of objects. An array begins with "[" (left bracket) and ends with "]" (right bracket). Any array can be empty: `[]`.

For instance, the key `"collectorStart"` refers to the object `{"x": 32, "y": 90}`, which has two name/value pairs itself. You can see snapshots of both levels in Figures 5 and 6. Any object which contains the data of a level will have the following three names (keys):

- `"collectorStart"`, which refers to an object with two name/value pairs: the X and Y coordinates where the main character —the "stars collector"— is initially placed on the stage.
- `"ground"`, which is really a platform that does not require its X and Y coordinates to be specified. Each platform is an object that will have five name/value pairs:
  - `"x"`, its X coordinate.
  - `"y"`, its Y coordinate.
  - `"enemies"`, which refers to an array of objects that will have two name/value pairs: the X coordinate where the enemy is placed and a number (0/1) which tells whether the mouse's sprite will

[9] We are not reusing enemy sprites. No problem is expected with garbage collection because the number of enemies placed in a level will be small.

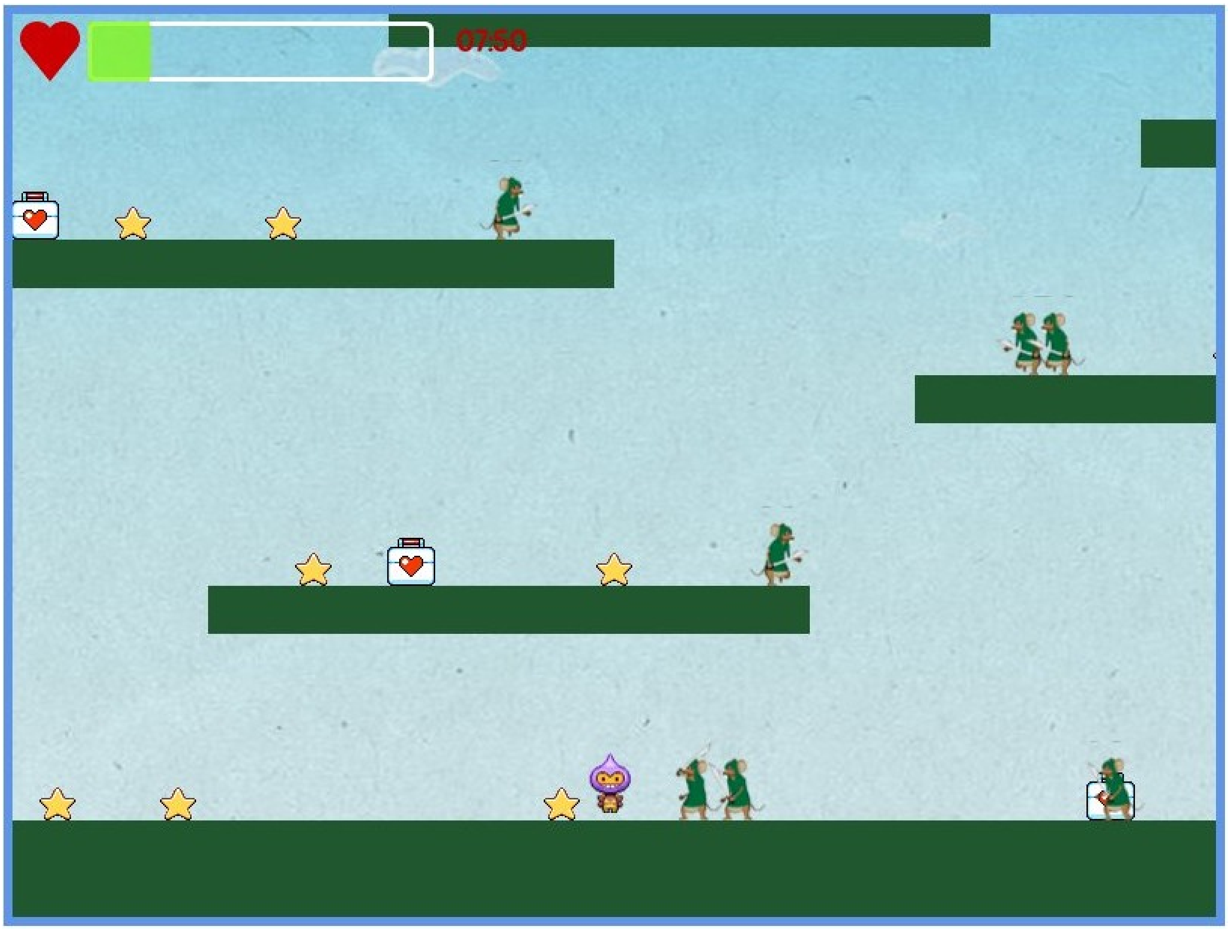


Figure 6: *A snapshot of the second level of our platform game.*

be added to the stage facing to the left or to the right.

  – "aids", which refers to an array of objects that will have only a name/value pair: the X coordinate where the first-aid box is placed.

  – "stars", which is similar to "aids" but placing stars instead of first-aid boxes.

  The enemies, stars, and first-aid boxes do not need a name/value pair for the Y coordinate since this value can be computed from the position of the platform in the Y axis.

- "platformData", which refers to an array of platforms —excluding the ground.

Now it's time to explain how to read a JSON file and how to work with it. First of all, we have declared and initialised a global variable in `play.js`: `levelsData`. This variable holds an array of strings with the relative path to each JSON file. Remember that these files are stored into the `levels` folder which is inside the `assets` folder (see Section 2). The order matters: the levels are played in the same order as they are stored into the array —from the beginning to the end of the array.

```
let levelsData = ['assets/levels/level01.json',
                  'assets/levels/level02.json'];
```

Observe that thanks to the separation between the game contents and the programming logic we can add several new levels to our platform game without the need for changing the code: we only have to add the relative paths of the new levels to the array `levelsData` in the adequate order.

Next, we have to load the JSON file in the `preload` phase just as we do with any other resources. This is done in the `loadLevel` function:

```
function loadPlayAssets() {
  // ...
  loadLevel(levelToPlay);
}
// ...
function loadLevel(level) {
  game.load.text('level', levelsData[level-1],
                 true);
}
```

The variable `levelToPlay` is actually declared and initialised in `welcome.js` —it needs to be initialised in the `create` phase:

```
let levelToPlay;
// ...
function displayScreen() {
  levelToPlay = 1;
  // ...
}
```

Note that we use the method `game.load.text` to load the JSON file and that the third argument —`true`— tells

Phaser to overwrite data in its cache. This is a safer option since different JSON files can have some data with common names.

Then, in the `create` phase of `play.js`, inside the `createLevel` function, we call the `JSON.parse` method passing the data stored in the **Phaser** cache as argument:

```
levelConfig = JSON.parse(game.cache.
                  getText('level'));
```

This will store the parsed data —all of the objects and arrays with their name/value pairs— in the variable `levelConfig`. Therefore, we could easily access to, for example, the X and Y coordinates of the object identified by the key `"collectorStart"` by means of:

```
levelConfig.collectorStart.x
```

and

```
levelConfig.collectorStart.y
```

Observe that, for instance, `levelConfig.platformData` or `levelConfig.ground.enemies` are arrays which can be easily traversed in a loop.

After loading the JSON data in `levelConfig`, we call the functions `createGround` and `createPlatforms` to place all of the elements on the stage —the function `createPlatform` is similar to `createGround`:

```
function createGround() {
  ground = platforms.create(0, game.world.height
                             - 64, 'ground');
  ground.scale.setTo(2.75, 2);
  ground.body.immovable = true;

  for (let i = 0, max = levelConfig.ground.
       enemies.length; i < max; i++)
    setupEnemy(levelConfig.ground.enemies[i],
               ground);

  for (let i = 0, max = levelConfig.ground.
       aids.length; i < max; i++)
    setupAid(levelConfig.ground.aids[i],
             ground.y);

  for (let i = 0, max = levelConfig.ground.
       stars.length; i < max; i++)
    setupStar(levelConfig.ground.stars[i],
              ground.y);
}

function createPlatforms() {
  levelConfig.platformData.forEach(
       createPlatform, this);
}
```

As you can see, these functions simply iterate over the arrays loaded from the JSON file and call the functions `setupStar` and `setupAid` (Section 5.2), and `setupEnemy` (Section 5.3).

To end this section, it only remains to be explained how to move on to the next level once the actual one has been completed. As we explained in previous sections, we check whether the player can leave the actual level at the end of the `updateLevel` function in `play.js`:

Your turn **5**

Write the code of the function `createPlatform`. Bear in mind that the `element` parameter is a platform, i.e. an element of the JSON array `"platformData"`, as it is passed from the function `createPlatforms` in the call to the `forEach` method. Observe that the argument `this` will be instantiated to each element —platform— of the `platformData` array in the loop.

```
function updateLevel() {
  // ...
  if (!exitingLevel)
    game.physics.arcade.overlap(player, exit,
                 endLevel, null, this);
}
// ...
function endLevel() {
  if (totalNumOfStars === 0) {
    exitingLevel = true;
    resetInput();
    soundLevelPassed.play();
    stopPlayer();
    game.time.events.remove(timerClock);
    game.time.events.add(4000, nextLevel,
                         this);
  }
}
```

The function `endLevel` will be called if the player collides with the exit sign. Inside this function, we check whether the player has collected all of the stars. If so, we proceed with leaving the level. The function that actually will leave the level is `nextLevel`. Note that this function will be called with a four-second delay. We set this timer because we have to do a "clean" exit: we want to play the sound before actually moving on to the next level and we want the user to become aware of leaving the level. Therefore, we need to cancel the (user) input because the actions for the key presses would be carried out in the next level otherwise —for instance, the player could start the next level jumping or moving right or left, something that is not desirable in general. To this end, we have followed a similar approach to that employed in the previous lab session:

```
function resetInput() {
  game.input.enabled = false;
  cursors.left.reset(true);
  cursors.right.reset(true);
  cursors.up.reset(true);
  cursors.down.reset(true);
}
```

The function `nextLevel` is quite simple:

```
function nextLevel() {
  clearLevel();
  levelToPlay += 1;
  if (levelToPlay > levelsData.length)
    goToWelcome();
  else {
    game.input.enabled = true;
    game.state.start('play');
  }
}
```

First of all, it destroys all the remaining sprites and elements in groups —clearLevel— so that some memory is released and left available for garbage collection. Then, it increases the value of `levelToPlay` by one: remember that this is the index used to traverse the array `levelsData`. If we have not reached the last element of this array —there are more levels to play— then input is enabled again and the same **Phaser** state —`playState`— is restarted.

If there were no more levels to play, we would come back to the welcome screen by calling the `goToWelcome` function, whose code was shown at the end of Section 5.1. However, there is a problem: when we return to the welcome screen —either because there are no more levels to play or because the time to complete the current level runs out— the input is still cancelled. This is the reason why we always resume input in the `create` phase of `welcome.js`:

```
function displayScreen() {
  levelToPlay = 1;
  game.input.enabled = true;
  // ...
}
```

Phaser examples: Loader

Load JSON file

Phaser Tutorial

How to Load Level Data from JSON Files

7 HUD

The HUD of our game is actually a **Phaser** group which consists of (see Figures 5 and 6):

- The image of a heart.
- The image of a holder for the health bar.
- The image of a filled rectangle which represents the player's health (health bar).
- A text showing the time that is left to complete the level.

This HUD is created in the function `createHUD`, which is called inside the `createLevel` function in `play.js`:

```
function createHUD() {
  hudGroup = game.add.group();
  hudGroup.create(5, 5, 'heart');
  hudGroup.create(50, 5, 'healthHolder');
  healthBar = hudGroup.create(50,5,'healthBar');
  hudTime=game.add.text(295,5,setRemainingTime(
     remainingTime), {font: 'bold 14pt Sniglet',
                      fill: '#b60404'});
  hudGroup.add(hudTime);
  hudGroup.fixedToCamera = true;
  healthValue = MAX_HEALTH;
}
```

Observe that we have added all of the components of the HUD to `hudGroup` so that we can work with them as a single entity. However, we have also stored in a separate variable — `healthBar`— the sprite which represents the player's health. This is needed to be able to update the HUD's health level as we will see shortly.

The function `setRemainingTime` simply converts the seconds remaining to complete the level to a string with the format `mm:ss`.

Since we would like that the entire HUD moved in the world as the player moves, we have set the property `fixedToCamera` of `hudGroup` to `true`. This is the tool that **Phaser** provides for attaching objects to its camera model inside the world.

To update the HUD — the health bar, actually— we use a single tween on the `scale` property of the `healthBar`:

```
function updateHealthBar() {
  if (healthTween)
    healthTween.stop();
  // write the required instructions to set
  // the tween and start it
}
```

Your turn 6

In the function `updateHealthBar`, write the instructions required to set a tween on the `healthBar.scale` property so that it displays the current health's value of the player. Bear in mind that the tween has to be stored in the variable `healthTween` and will last 300 ms. The tween will play an animation on the property x whose value will be given by a simple rule of thumb: `healthValue / MAX_HEALTH`.

8 Adjusting the bounding box of a sprite

The Arcade physics engine of **Phaser** relies on the well-known "bounding box" algorithm to detect collisions. This can be adequate in most cases, but problems arise with some sprites, particularly with spritesheets which contain several frames (animations), some of them with convex and/or concave forms: the bounding box is sometimes too big and, thus, some collisions that can happen in the game are somewhat weird.

We can solve this kind of problems by "manually" adjusting the size of the bounding box (see theory slides).

To this end, you can use the `body.setSize` method (see documentation[10] for further details):

```
function createLevel() {
  // ...
  exit.body.setSize(88, 58, 20, 33);
  // ...
}
// ...
function setupEnemy(enemy, plat) {
  // ...
  theEnemy.body.setSize(41, 43, 3, 10);
  // ...
}
```

Besides, we can find out whether the collision came from up, right, left, or down just using the right properties of a `body`:

```
function playerInPlatform(player, platform) {
  if (player.body.touching.down)
    playerPlatform = platform;
}
```

Remember that the function `playerInPlatform` is called whenever the player collides with a platform. To store the platform where the player stands we need to be sure that the player is *actually standing*. This is accomplished by checking the value of the boolean `body.touching.down` of the sprite. A similar approach is used in the function `playerVsEnemy` to check whether the collision happened on the head of the mouse:

```
function playerVsEnemy(player, enemySprite,
               enemyObject, position) {
  if (enemySprite.body.touching.up) {
// ...
}
```

Your turn 7

Now, it's time to check whether you have correctly completed the code of the game as instructed in the previous *Your turn* boxes. Therefore, run the project in *Code* and play the game. Take your time to check all of its features and functionality. See what happens whenever a level is completed and when you run out of time.

9 Basics for debugging Phaser games

Through its Debug class[11], **Phaser** offers a collection of methods for displaying debug information about game objects. We have used this feature of **Phaser** to adjust the bounding boxes of some sprites (see Section 8) and to fix a couple of nasty bugs regarding collisions.

If your game is running in `Canvas` mode, as it is our case, then you should call all of the debug methods from the function linked to the `render` phase of the state of the game you want to debug. This is because they are drawn directly onto the game canvas itself, so if you call any debug methods outside of `render` they are likely to be overwritten by the game itself.

Remember that after the `create` phase, **Phaser** will execute its internal game loop which, in turn, consists of two phases: `update` and `render`. This loop runs until another state is started or the current running state is stopped (see theory slides).

We suggest that you do the following simple exercise. We are going to perform a basic debugging of the camera model and part of the physics: concretely we will debug the bodies of the player, exit sign, mice and platforms.

Your turn 8

1.- Go to file `play.js`.

2.- Add the render phase to `playState`:

```
render: debugGame
```

3.- Add the body of the `debugGame` function to the end of the file:

```
function debugGame() {
  game.debug.camera(game.camera);
  game.debug.cameraInfo(game.camera);

  game.debug.body(player);
  game.debug.body(exit);
  for (let i = 0, max = enemies.length;
      i < max; i++) {
    game.debug.body(enemies[i].sprite);
  }
  game.debug.physicsGroup(platforms,
      'rgba(255,0,0,0.5)', false);
}
```

4.- Finally, save changes and run your project. Pay attention to the result.

More on debugging

- **Physics example**
- **All debug examples**
- **The Complete Guide to Debugging Phaser Games**

10 Exercises

1.- When the game ends, either because there are no more levels to play or because the time to complete a level runs out, the game currently returns to the welcome screen with no feedback to the user. You will have to change that. Add a final screen that will be shown when the game ends. This final screen should show:

[10] https://photonstorm.github.io/phaser-ce/Phaser.Physics.Arcade.Body.html#setSize
[11] https://photonstorm.github.io/phaser-ce/Phaser.Utils.Debug.html

- A sad image and a text message of regret if the time ran out —i.e. the player lost the game. Use a new font downloaded from Internet to write the text message.
- An image of joy and a text message congratulating the user if she successfully completed all of the levels of the game. Besides, there should be text messages showing the remaining time in each level. Again, use at least a new font downloaded from Internet to write the messages.

In any case, this final screen should display two buttons: one leading to the welcome screen and the other leading to playing again the game.

Do not forget to change the FSM (see Figure 4) to reflect the new design of the game.

2.- Design a new level of your own for the game. Write the corresponding JSON file and add it to the game. Do not forget to properly order the levels of the game from the easiest to the toughest in the array `levelsData`.

3.- Add a new type of enemy which will be harder to kill. This enemy will show up only in the last two levels of the game. Obviously, apart from the code of the game you should also modify the corresponding JSON files.