

4. Phaser: assets

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

Estimated duration: 4 hours (+ 5 hours of exercises)

In this lab session we will learn the basic structure of a game in **Phaser**, one of the most popular game frameworks for the web. First, we will see how **Phaser** allows the programmer to easily define *states*, and switch between them. We will also look at how to manage keyboard, mouse, and time events. Special attention is paid to learning how to include, load and manage various media types (text, images, buttons, videos, and audios), which are collectively called *assets*. Three common and very useful concepts are also presented: sprite sheets (and texture atlases), tweening, and particles. As you can see, this lab session covers most of the essential ingredients that are required in a wide variety of games. Therefore, after working on this material, you will find yourself not only well equipped to develop a wide range of games, but also endowed with the fundamental skills to create more sophisticated games with **Phaser** in the next lab sessions.

sounds, physics, and collisions. It also has an active and wide community of developers.

Just have a quick look at...

[Phaser Examples](#) [Phaser Community](#)

We can see at html5gameengine.com that **Phaser** is, as of this writing, one of the most popular and rated HTML game frameworks. If we exclude commercial frameworks like **Construct 2** or **ImpactJS** from consideration, then **Phaser** seems to be definitively a good option for free game development, such as **EaselJS**. However, **Phaser** is somehow more complete than **CreateJS** (which **EaselJS** is part of), including support for collision and physics.

Contents

- 1 What is Phaser?
- 2 A basic program with Phaser
- 3 Keyboard and mouse events
- 4 Monitoring the game progress
- 5 Assets using images
- 6 Assets using sounds and videos
- 7 Examples
- 8 Exercises

1 What is Phaser?

As you may guess, **Phaser** is a JavaScript game framework widely used for the creation of graphically rich 2D games on the web. It is a free, open-source, and full-fledged framework for advanced features such as

Setting the environment

Before we look at a first program in **Phaser**, you can follow the instructions in the “Phaser.js: part One” class to set up a **Phaser** project in *VS Code*. In this lecture, there are two videos (also available in the *Aula Virtual*) that show the basic structure (folders, file types, etc.) of a **Phaser** project and how to run it in *VS Code*, taking advantage of its autocompletion and IntelliSense features for **Phaser**. It is strongly recommended to follow this project skeleton for your own game projects because, in this way, *VS Code* will automatically recognise and understand **Phaser** objects and methods. If needed, refer to Appendix A in the first lab session to find tips to use *VS Code*.

The **Phaser** API documentation is also installed in the lab computers: search for **Phaser** in Programes Instal·lats, go to the folder docs, and open index.html.

2 A basic program with Phaser

Now we are ready to look at a very simple program with **Phaser**. Download the archive **4students.zip** corresponding to this lab session on your computer and unzip it. Once unzipped, you will find a folder named **4students** which holds the folder of the basic **Phaser** game *Letters*.

Box 1. What are phaser.min.js and phaser.map?

In the build folder of the **Phaser** installation on the lab computers, you will find the library `phaser.js`. But, you can rightfully wonder: **what are the other files for?** `phaser.min.js` has exactly the same functionality as `phaser.js`, but it is a *minimised* version with shorter variable names and unnecessary characters removed. This results in a smaller file so that the web page loads faster. **How small is phaser.min.js?** While `phaser.js` takes up over 3 MB, `phaser.min.js` is below 1 MB (takes about 800 KB), so its size is about one fourth of that of `phaser.js`. It is a good experience to have a look at both files and compare them. One is human friendly, the other is network friendly. The next logical question is: **how can one minimise JS files?** Although there are other alternatives, this is extremely easy in *VS Code*, with the extension “JS & CSS Minifier”. Once installed, the text “Minify” is displayed at the bottom left corner of *VS Code*. Click on it to *minify* the current selected file.

□

As for `phaser.map`, it is called the *source map*, and its usage makes sense in tandem with the minimised version of the library. The source map allows to reconstruct the original version of the file from the minimised file. This makes sense only for debugging purposes. **What files are you expected to use for debugging?** During development, you can use the full version of the library, `phaser.js`. During production, you can more easily debug your code if the source map accompanies the minimised file. There are also tools to generate the source maps of given JS files. **Should we have phaser.map for ever?** Even though the size of `phaser.min.js` and `phaser.map` together is below the size of `phaser.js`, `phaser.map` should not be included once debugging is not required anymore.

Your turn

1

Open the folder *Letters* in *VS Code* (File → Open Folder...). Explore the existing files and the folder structure of the project. To run the game, however, you need to slightly edit `index.html`. First, you will need to add the following line:

```
<div id="game"> </div>
```

Next, you will need to import two JavaScript files in the `index.html` file: `phaser.js` and `letters.js`. Keep in mind that the order matters.

Once you run the game, let's have a careful look at the code in `letters.js` to understand the bits and pieces of a minimal **Phaser** game.

① We first set up a **Phaser** game by creating an instance of a `Phaser.Game()`:

```
let game = new Phaser.Game(
  GAME_AREA_WIDTH, GAME_AREA_HEIGHT,
  Phaser.CANVAS, "game"); // game instance
```

The first two arguments are the width and height of the game. The next argument specifies the rendering context for the canvas (here, `phaser.CANVAS`). The last argument is an id (here, `"game"`) that instructs **Phaser** to place the canvas (that **Phaser** creates for you) in the `div` element with that id.

② A key characteristic of **Phaser** is the concept of *states*. You can think of states as the main “screens” of the game. In a typical simple game, we might have the presentation state (with the credits and a play button), the main state (where the actual game takes place) and the game over state. In the example, we only have one state, which is named `'main'`. We will later add the `'game over'` state.

```
window.onload = startGame;
```

```
function startGame() {
  game.state.add('main', mainState);
  game.state.start('main');
}
```

Phaser simplifies the management of states a lot. The function `startGame()` above creates a unique state and starts it. We simply call the method `game.state.add()` to add a state. The first argument is an id (`'main'`), and the second one is a JavaScript object (`mainState`) that defines the configuration of the state. Next, we explicitly start the state by invoking the `game.state.start()` method, passing the id of the state as the argument.

③ We now focus on the `mainState` variable. It defines the *phases* of the only state defined in the game (`'main'`). This state is composed of three phases: *preload*, *create* and *update*.

```
let mainState = { // state's phases
  preload: preloadAssets,
  create: initGame,
  update: updateGame
};
```

Box 2 clarifies what a **Phaser** state is.

Box 2. Phaser states

In short, a **Phaser** state is an object whose property names are the *phases* of a state, and the property values are the functions or methods to execute in each phase.

In our case, we associate three key functions with arbitrary names (`preloadAssets`, `initGame` and `updateGame`) to each of three phases. These functions are part of the game when the state is added. What are these functions? They are executed in each phase of the game:

- `preloadAssets()` preloads the assets we will need to add in the game. Here, we load a single image only:

```
function preloadAssets() {
    game.load.image("logo", "assets/imgs/uji.png");
}
```

In general, we would load several assets: images, sprite sheets, audio files, etc.

- `initGame()` initialises some elements of the game, and includes some of the assets we loaded. Here, we set the colour of the background, and add the image:

```
function initGame() {
    game.stage.backgroundColor = BG_COLOR;
    logoImg = game.add.image(0, 0, "logo");
    logoImg.scale.setTo(0.2);
}
```

Without `game.add.image()`, the image would not have been displayed, even though it was already loaded. Note also that we scale the image to 20% of its size.

- `updateGame()` changes the game appearance according to the passing of time.

```
function updateGame() {}
```

This is the **Phaser** way to implement the *game loop*. In our case, this function is currently empty, and therefore nothing happens. But we are going to remedy this right now.

Your turn

2

Let's move our logo from left to right. Add this sentence to the `updateGame()` function and try the result.

```
logoImg.x += 1;
```

Obviously, the logo disappears when it reaches the right side of the scene. Let's make the logo reappear again at the left side, and repeat this cycle over and over. **Hint:** Use the logo width (`logoImg.width`) and the game width (`game.width`).

□

Since we are going to put many things inside the `updateGame()` function, it is good to make auxiliary functions that group related functionality. Thus, create the function `moveLogo()` with the code we have now in `updateGame()` and call it from `updateGame()`. Cleaner and better!

You imported `phase.js` before, but it is good to know that there are two other related files that we will care about, as explained in Box 1.

3 Keyboard and mouse events

It is time to add interaction with the user. A screenshot of the game is shown in Fig. 1, which implements the following requirements:

- the user can press any key, but *only* those corresponding to letters will be displayed at random positions;



Figure 1: Screenshot of the main state of *Letters*

- the letters will move randomly;
- the user will click on the displayed letters to remove them;
- a simple [Head-Up Display](#) (HUD) will display the number of “hits” (i.e. clicked letters).

Although these are arguably simple and ordinary things (not much fun yet, really), they cover essential aspects when programming games in **Phaser**. Let's start with the keyboard events.

Keyboard events

By adding this sentence to the `initGame()` function,

```
game.input.keyboard.onDownCallback = getKeyboardInput;
```

we are saying to **Phaser** that `getKeyboardInput()` will be the *callback* function for the event corresponding to pressing down a key on the keyboard. In other words, every time a key is pressed down, **Phaser** (i.e. `game.input.keyboard` handler) will execute the corresponding callback function. Therefore, we need to define this callback function:

```
function getKeyboardInput(e) {
    if (e.keyCode >= Phaser.Keyboard.A
        && e.keyCode <= Phaser.Keyboard.Z) {
        let a = game.add.text(Math.random() * game.width,
            Math.random() * game.height, e.key,
            {fontSize: '40px', fill: '#FA2'},
            letters); // group to add to
    }
}
```

You can probably understand most of this code without further explanation. Here, we highlight some important comments:

- The function has the parameter `e` which is the event object. This object contains information related to the *actual* event. In our case, we are interested in the numerical code of the pressed key, `e.keyCode`. It is

important to use existing defined constants, as we do here for the A and Z codes, or define our own constants if they are not available yet.

- With `game.add.text()` we add the text (`e.key`) at a random position, with a given colour (`'#FA2'`) and font size (`'40px'`).
- We choose random positions for the text. For instance, for the *x* coordinate, we use

```
Math.random() * game.width
```

But we could have done the same with the predefined `game.world.randomX()` instead.

An important detail remains unresolved yet: what is `letters`? In the call to `game.add.text()`, we say that the text (keyboard letters) is added to `letters`. Instead of adding letters individually, we group them. So, we have to define a *group* for the variable `letters`. A group is a list of some game objects. It's like a bag where you can put many items inside. The benefit of a group is that **Phaser** can operate on all the child objects together. To create the group `letters`, we add the following code in the `initGame()` function:

```
letters = game.add.group();
```

The *children* of this group are all the texts we will be adding. Since we want to be able to click on each of them, we must enable this explicitly.

```
letters.inputEnableChildren = true;
```

If you noticed that you are using `letters` without having been declared first, congratulations: you are an attentive student! Certainly, we must declare the variable. We do it in a global scope (outside any function), since we will need to refer to it from different functions. Actually, we're already using it in `getKeyboardInput()`, aren't we?

Your turn

3

Write the code that we have been describing before, and verify the result. You should see the letters corresponding to the keys you pressed. If this is not the case, check your code again.

Before studying the mouse events, let's shake the letters a bit to add some dynamism:

```
function shakeLetters() {
  for (const child of letters.children) {
    child.x += Math.random() * 2 - 1;
    child.y += Math.random() * 2 - 1;
    child.angle += Math.random() * 10 - 5;
  }
}
```

Of course, we need to call `shakeLetters()` somewhere for it to take effect. Where should we do it? You are right: in our `updateGame()` function. Pay attention to how to iterate over the children of a group object, since you may need it many times.

Your turn

4

Write this code and see how the letters shake. Can you appreciate around which point they rotate? It seems that each letter hangs from its upper left corner. Often, we prefer rotations around the center of the objects. This is easy to do: all we need to do is setting properly the *anchor* point:

```
// relative positions in [0,1]
a.anchor.setTo(0.5, 0.5); // 0.5 for the center
```

You may (easily) guess where to place this sentence. Note that `a` is the name of a variable we already used in the code to refer to each letter we have added to `letters`.

Mouse events

We have already enabled the mouse click on the children of our `letters` group, with the `inputEnableChildren` property. But as you may well know, we also need a callback function to associate the click event with the action we want to perform in response. This is how we do this:

```
letters.onChildInputDown.add(processLetter);
```

And yes, we also need to define the callback function:

```
function processLetter(item, pointer) {
  item.destroy(); // frees up memory
  // kill() removes it from display list,
  // but not from the group
}
```

See Box 3 for a distinction between two related functions, `kill()` and `destroy()`.

Note the two important parameters that the callback function receives: the *item* that received the click, and the *pointer*, that is contextual information about the event. For instance, we can get the coordinates of the click event with `pointer.x` and `pointer.y`.

Box 3. What's behind the names?

Among the names used in **Phaser** for managing sprites we find `kill()` and `destroy()`. Aren't these names unnecessarily violent for a general-purpose game framework? Would not other names such as `hide()` or `remove()` be not only more general and peaceful but perhaps also even more descriptive of the intended functionality? What's actually the difference between "killing" and "destroying" something?



In our **Phaser** code, `kill()` would remove a letter from the game (display list) but not from the group. It can be *revived* again. However, with `destroy()` the letter is definitely deleted from the group, and you can never use it again. If your game plans to re-use a letter (or sprite, object, etc), it is a good idea to get comfortable with the kill/revive strategy.

Imagine now that we would like the player to be able to remove letters not by clicking on specific individual ones, but by clicking anywhere within the game world and removing all letters close enough to the mouse pointer. Let's see how we can do this.

Your turn

5

When a mouse click is detected, we will remove all letters whose (☞ [Euclidean](#)) distance to the mouse cursor position is below a given distance threshold. Let's do it step by step.

1. To detect the click, there are at least two options:

(a) We can register an event listener with

```
game.input.onDown.add(checkDistance);
```

(b) Alternatively, we can explicitly call the function `checkDistance()` in the `updateGame()` function:

```
function updateGame() {  
  [...]   
  checkDistance();  
}
```

You can try both alternatives, one at a time, commenting on the other.

2. Code the distance computation. We give you most of the code; complete the few missing lines.

```
function checkDistance() {  
  const DIST_THRESH = 50;  
  if (game.input.activePointer.isDown) {  
    let xPointer = game.input.x;  
    let yPointer = game.input.y;  
    for (const letter of letters.children) {  
      let x = // ...  
      let y = // ...  
      let d = // ...  
      if (d <= DIST_THRESH)  
        processLetter(letter);  
    }  
  }  
}
```

Note that we only pass an argument to `processLetter()`, even though this function has two parameters. You already know how flexible JS is.

Note: Testing for the condition

```
game.input.activePointer.isDown
```

to be true is only required if `checkDistance()` is called explicitly from the `updateGame()` function. If `checkDistance()` is used as a callback (handler) for the **"onDown"** event, this test is obviously unnecessary.

3. Verify that when you click somewhere in the game world, all letters close enough to the clicked position disappear.

4 Monitoring the game progress

We now want to display the number of letters the user has successfully clicked on. We can create a score by initialising the text, in `initGame()`, as follows.

```
scoreText = game.add.text(  
  15, // x  
  logoImg.y+logoImg.height, // y  
  'Hits: '+numHits, // text  
  {fontSize: '32px', fill: '#000'});
```

It is interesting to note how this text is positioned *relatively* with respect to the vertical position of the logo, to prevent the logo and the text from overlapping.

Your turn

6

Complete what you need to properly count and display the number of hits. In case you need some help, here you have the step-by-step process:

1. Declare the global variable `scoreText`.
2. Declare the global variable `numHits`, and initialize it inside `initGame()`.

Please, read carefully Box 4 at this point.

Box 4. Initialise variables inside `initGame()`

This piece of advice is more than an anecdotal detail. If the initialisation is done outside `initGame()`, then the risk is that when switching between states, the behaviour of the game is unexpectedly incorrect. You will better understand it when we will be dealing with the *over game state*, from which the game can be resumed. We will later remind you this issue so that you experience what we mean.

3. Update `numHits`.
4. Update `scoreText.text` (using the updated `numHits`). A good way to do it is creating a new function `updateScore()` which is called within... guess which function! Yes, `updateGame()`.
5. Verify it works as intended.

Let's define as the simple condition for the game end when the number of hits is above a given minimum, say:

```
const HITS_MIN = 5; // small for quick debugging
```

Now we are going to check if the game is over and, when it happens, we will switch to another *state*. Let's see how.

Your turn

7

1. Create a new function, `checkGameOver()`, to check whether the game is over by comparing `numHits` against the threshold `HITS_MIN`.

Box 5. On displaying debugging information

Although we have seen the most common phases (**preload**, **create** and **update**) in a state within a **Phaser** game, there might be a few others. One of them is **render**, and since in general all objects are rendered automatically (without us having to explicitly say so), the documentation says:

“the render method is called AFTER the game renderer and plugins have rendered, so you’re able to do any final post-processing style effects here”.

In many **Phaser** examples, though, we find this method is conveniently used to display debugging information. A general approach to quickly display the text information on the canvas for debugging purposes is as follows:

```
game.debug.text('Phaser ' + Phaser.VERSION + '. Num. hits: ' + numHits, 0, game.height-30);
```

where `debug` is a predefined instance of the `Debug` class that every game object comes with. This can be seen as a convenient alternative or as a complementary mechanism to printing on the browser console.

Beyond this simple example, there are other examples of [displaying debugging information](#). You can have a look at [How to debug Phaser games](#). No worries, we will revisit this topic in a coming lab session.

At this point, our `updateGame()` function will look like as nice and tidy as this:

```
function updateGame() {  
  moveLogo();  
  shakeLetters();  
  updateScore();  
  checkGameOver(); // <-- new  
}
```

2. If the game is over, we will switch to the new state **'over'**. In **Phaser**, this is as easy as adding:

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

Of course, we need to define and add this new state. Let's go for it!

3. Just below the line where we added the **'main'** state, we add the **'over'** state:

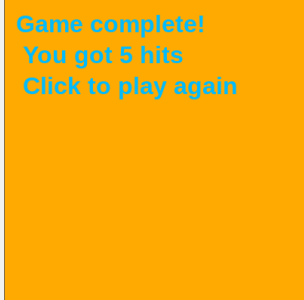
```
game.state.add('over', overState);
```

4. Create a new JS file, say `over.js`, where we will define `overState`. Why so? It is a good programming practice to have the code of different game states in separate files. This way, it is easier to manage the code and to avoid messing you up. For this state you will only need the `create` phase. Therefore, we will have:

```
let overState = {  
  create: createOver  
};
```

5. Don't forget to add the proper `<script>` element, in the right order, to include `over.js` in `index.html`.

6. Define the `createOver()` function so that the end game screen looks similar to this screenshot.



Game complete!
You get 5 hits
Click to play again

7. Let the game be resumed when the user clicks on this text. To that end, set `scoreText.inputEnabled` to **true** to tell **Phaser** to capture the click events on this text, and don't forget to set the callback via `scoreText.events.onInputDown.add()`. What must this callback function do? **Hint:** A single sentence is required.

Since we have performed many little steps, you might have lost the big picture. Please, recap to make sure you understand the overall idea and then realise how easy all the steps actually are.



By the way, do you remember (Box 4) that we recommended you to initialise `numHits` inside `initGame()`? Well, you can now test what happens if `numHits` is initialised outside of it. Can you appreciate it? You should understand why this happens, and remember it when you observe some malfunctioning in your own games that might be related to incorrect state switching.

As a side note, see Box 5 for a debugging tip.

5 Assets using images

Although images can be used individually *per se*, they are also useful to build other game elements. We now look at two examples of these usages: sprite sheets and buttons.

Animating with sprite sheets

A *sprite sheet* is nothing but a collection of images. Sprite sheets commonly contain images of the same object each with slightly different appearance. Thus, when these images are displayed in the proper order at a proper frame rate, the illusion of animation is created.

A related concept to sprite sheet is *texture atlas*. Like a sprite sheet, a texture atlas has the sprite images in an image file. Unlike a sprite sheet, a texture atlas has, additionally, an XML or JSON file which describes where each sprite image is located within the sprite sheet. You can refer to Box 6 for a gentle introduction to JSON.

A common dilemma is choosing between sprite sheets or texture atlases. Sprite sheets may be easier to use, at least from a beginner's point of view, and may be sufficient for a basic usage. However, texture atlases are generally preferable over sprite sheets, because they offer a solution which is more general, more flexible, and computationally advantageous.

How to animate...

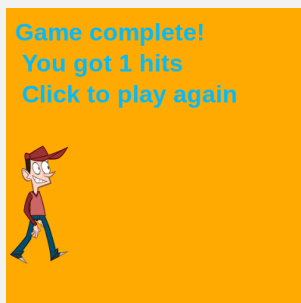
- 🔗 [Sprite sheets with Phaser](#)
- 🔗 [Texture atlases with Phaser](#)

There are software packages that can help you create sprite sheets or texture atlases from a set of existing images. A free and simple, but useful program is 🔗 [TexturePacker](#).

Your turn

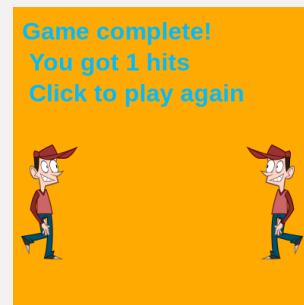
8

1. Read this tutorial and make sure you understand most of it:
🔗 [Sprite sheets for Phaser with TexturePacker](#)
2. Download and test the 🔗 [code available](#).
3. Include the animation of the guy, excluding its translation, in the end-game screen of our `Letters`. A snapshot of this screen would look like this:



Hint: From the downloaded files, take only those files and code snippets you need, and adapt them as required. Don't forget to create an appropriate folder (say, `textureAtlas`) to include the texture files, i.e. `cityscene.[png|json]`.

4. Change the animation speed: try both faster and slower animations.
5. Examine the contents of `cityscene.png` and `cityscene.json` and try to understand how the latter is organised, and the role of the different attributes. Change the value of some attributes for a particular frame, and check its effect in the animation. Then, don't forget to restore the original value for the modified attributes
6. Include another guy facing in the opposite direction, i.e. like a vertical flip. Mathematically, a flip can be seen as equivalent to a change of scale; we use a *negative* scaling argument. Now, run the animation in reverse order.



Note that the “flip” trick is useful for a character to move in different directions without changing the sprite contents. What attribute would you change to give the impression that your character is walking on a gentle slope?

Buttons

A button in **Phaser** can be created by associating an image to it.

A simple button in Phaser

```
game.add.button(x, y, 'image', handler);  
// 'image': the image key of a loaded image  
// 'handler': function run when button clicked
```

Let's create a button “Play” and another one “Stop”, and switch between them upon user clicking (Use the `Videos` project within the `4students.zip` file).

Your turn

9

1. Create two images for the intended buttons. See Box 7 for a useful tip.



2. Load both images in **Phaser**, and then create the corresponding buttons. Place them in the same position, but make only one of them visible.
Hint: Use the buttons' attribute `visible`.

Box 6. What is JSON?

JSON stands for *JavaScript Object Notation*, and is a file format commonly used for browser-server communication. It is an alternative to the more verbose XML (Extensible Markup Language). Despite its name, any programming language can process this format, not only JavaScript. In JS, the two basic methods are:

- `JSON.stringify(object)` to generate the string in JSON format of a given object.
- `JSON.parse(string_json)` to create an object from a string in JSON format.

For instance, we may have this JS object

```
let puzzle = {};  
puzzle.pieces = [{x: 10, y: 10}, {x: 20, y: 50}];  
puzzle.solved = 0.8;
```

Then, the following code:

```
let puzzle_json = JSON.stringify(puzzle);  
console.log(puzzle_json);
```

will print this string in JSON format in the browser console:

```
{"pieces":[{"x":10,"y":10},{"x":20,"y":50}],"solved":0.8}
```

We can get also the object back from the string, and access whatever contents we want. So, with this code snippet:

```
let puzzle_obj = JSON.parse(puzzle_json);  
console.log("Number of remaining pieces:", puzzle_obj.pieces.length, " ~ ",  
           "Percent complete:", puzzle_obj.solved*100+"%");
```

we will get this when the object is printed out in the console:

```
Number of remaining pieces: 2 ~ Percent complete: 80%
```

Box 7. Quickly design buttons online!

🔗 [Dabuttonfactory.com](https://dabuttonfactory.com) is a very handy site which allows you to quickly create images for buttons. Give it a try!

3. Switch the visibility of buttons with a click handler. **Hint:** Associate the same handler function to both buttons. In this example, we do not need to know which of them was actually clicked.

Since a button can be in a number of states (normal, mouse over, mouse out, and clicked), we can associate a different image to each state. Interestingly, one convenient way to do that is with a sprite sheet which contains these different images.

How to ...

- 🔗 [Loading the button spritesheet](#)
- 🔗 [Adding the button to the game](#)

Phaser offers some more functionality related to buttons. For instance, we can indicate which sound to play for each state of the button. But do not explore this possibility by now; we can do it when we need it. Let's now move on to temporal assets (sounds and videos), dynamical effects (particles, animations) and timers.

6 Assets using sounds and videos

Although we can move images around or display one after another rapidly to produce the effect of animation, single images are essentially static. In this section, we will look at assets that are inherently dynamic such as sounds and videos, and will also study two techniques (tweens and particles) that will help us produce dynamic visual effects.

Playing sounds

To play sounds (or background music), we need to load and add audio files, in the same way we load and add other types of assets. When assets can take some time to load or decode, we must make sure they are available before we attempt to use them.

How to wait for audio files, and then play them

🔗 [Sounds with Phaser](#)

Box 8. Playing audios in a loop

Since the game may take longer than the time length of the audio file, we might want to repeatedly play it. For this purpose, we can set the property `loop` before playing the sound:

```
music.loop = true;  
music.play();
```


Your turn

10

Play a background music during our simple *Letters* game. Stop the music when the game ends, i.e. it should not play in the 'game over' state. You can get some free audio file from sites such as freesound.org or opengameart.org. **Tip:** Don't be picky now; be practical and choose *any* audio file that serves the learning purpose now. Do you want play an audio repeatedly? Then, please refer to Box 8.

Playing videos

Including and playing video files is simple and essentially follows the same steps we have seen for other assets.

The steps to play a video

```
// 1. Loading the video file
game.load.video(...);

// 2. Creating the video object
video = game.add.video(...);

// 3. Adding the video to the game world
video.addToWorld(...);

// 4. Playing the video
video.play();
```

Certainly, more actions on videos are possible, such as stopping them and even changing dynamically the source.

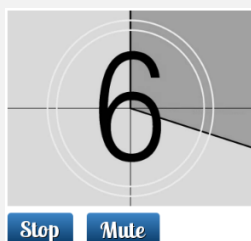
See how to...

[Playing video files and changing the source](#)

Your turn

11

To practice buttons and videos, let's create a simple program to play/stop a video and to turn on/off its sound track. The result should look similar to this:



playing and sound on



stopped and muted

1. Reuse the play and stop buttons you created before and place them below the video as in the example above.
2. Extend the handler for the play and stop buttons so that besides setting their visibility, the video can also be re-played or stopped.

```
if (buttonPlay.visible)
    video.stop();
else
    video.play();
```

3. Create an additional button for muting the video, and place it next to the above buttons. See Box 9 for a placement tip.
4. Define the handler for the mute button so that:
 - the button's transparency is changed, and
 - the video's sound is toggled.

The expected behaviour is illustrated in the example above. For changing the transparency, use the alpha property (in the range [0, 1], as usual).

Box 9. Relative placements

As a good design practice, place the elements of an interface relative one to each other. For instance, the mute button must be at the same vertical position of the other two buttons, and horizontally next to them:

```
buttonMute = game.add.button(
    buttonPlay.x+buttonPlay.width+gap, // x
    buttonPlay.y, // y
    'mute', // image key
    muteVideo); // handler
```

Animation with tweens

Generating intermediate frames between two images is called *inbetweening* or *tweening*. **Phaser** facilitates animating properties of objects by tweening. Although as a first thought we may consider the *position* as the property that is animated, other properties (rotation, size, opacity, colour...) can be animated (smoothly changed) as well.

The following example will animate the angle and the vertical position of the mute button in two steps:

```
tween_mute = game.add.tween(buttonMute);
tween_mute.to({ y: 0, angle: 90}, 500);
tween_mute.to({ y: buttonPlay.y, angle: 0 }, 500);
tween_mute.start();
```

As you can see, we indicate to which object we want to apply the tween (here, `buttonMute`), the properties we want to change (here, `y` and `angle`), the intermediate values for them (e.g. the first intermediate angle is 90°), and the duration of each steps (here, $500\text{ ms} = \frac{1}{2}\text{ s}$).

Your turn

12

Try this tween and then modify it and define some other. Play a bit with tweens and have fun.

We can do more things with tweens: stating what to do when a tween is complete, chaining tweens, using different speed profiles, etc.

Explore examples of tweens

[Tweens in Phaser](#)

Particles

In games, and computer graphics in general, particle systems use a (large) number of (small) objects that animated together can produce some nice visual effects. To use particles with **Phaser**, we may essentially follow these steps:

Using particles

```
// 1. Define an emitter at position (x,y)
//    and given width w and height h
emitter = game.add.emitter(x,y,w,h);

// 2. Set the images of the particles
emitter.makeParticles(['star','ball']);

// 3. Define properties of the particles:
//    gravity, speed, opacity, scale, etc.
emitter.setAlpha(0.6, 0.8); // transparency range

// 4. Start emitting
emitter.start(true, // all particles at once
             1000, // lifespan of each particle
             null, // ignored in "explode" mode
             15); // # particles in this burst
```

Of course, there are many possible variations, such as non-stop emitting particles (e.g. to simulate a continuous process such as snow). Properties can be dynamically changed after a first initial assignment.

Explore examples

[Particles in Phaser](#)

Your turn

13

Let's create a simple and short particle effect associated to clicking on the mute button.



particles emitted upon user click

1. Download a free image of a simple element of what could be a single particle. In the example above, we use a star.



[A star icon \(can be reused with modification\)](#)

2. Load the particle image (in the `preloadAssets()` method).
3. Define the emitter (in the `initGame()` method).
4. Define a function `startParticles()` which starts the emitter at the position of the mouse cursor (`game.input.mousePointer`). Emit a single burst of 20 particles, each with a lifespan of half a second.

5. Modify the handler of the click on the mute button to call `startParticles()`.

Not surprisingly, the power of particles can be boosted when combined with other ingredients. For instance:

- the emission point can be changed smoothly using *tweens*;
- the set of particle images can be given by a *sprite sheet*;
- the visual effect can be reinforced with playing a proper *sound*;
- the particles can interact with each other or with the environment by checking for *collisions*.

Note that we have already studied all of these game components except collisions and physics, which will be covered in subsequent lab sessions.

Timers

As you well know, timers are very important in games, and they are introduced in Box 10. You will be asked to use timers in Section 8.

7 Examples

To review and reinforce many of the concepts studied in this lab session, we are going to explore a couple of simple but illustrative examples of games. A snapshot of each game is shown in Fig. 2.

Sliding puzzle

The sliding puzzle in **Phaser** is simple enough for you to understand (almost) everything, but it is interesting for reviewing some concepts we have studied in this session.

Understand the code

[Sliding puzzle](#)

Your turn

14

Try to answer these questions regarding the “sliding puzzle” game.

1. What is the purpose of the sprite sheet?
2. We have one **Phaser** group, `piecesGroup`, which contains the pieces of the puzzle. How are the contents of each element in the group (i.e. each piece) defined?
3. There is one special piece, which is the “empty” one. How is this piece identified in the array `shuffledIndexArray`?
4. There is no `update()` function. So, how does the game loop work in this case?

Box 10. Timing in Phaser

In almost any game, we need to control the passing of time in some way. A *timer* is useful to verify something at a regular rate, or spawn objects, and so on.

Basic usage of timers

```
// 1. Create a timer object
timer = game.time.create(false);

// 2. Set the frequency and the handler
// e.g. here: call spawn() every 1.5 s
timer.loop(1500, spawn);

// 3. Start the timer running
timer.start();

// 4. Define what to do regularly
function spawn() {
    spawnNewObject(); // whatever we want
}
```

The boolean value in the first step is set to false so that the timer is not “autodestroyed” ([Phaser.Timer](#)). The first three steps can be compressed into:

```
game.time.events.loop(1.5*Phaser.Timer.SECOND,
    spawn);
```

Alternatively, if we want to spawn a predefined number of objects, say 30, we can use repeat instead of loop:

```
game.time.events.repeat(1.5*Phaser.Timer.SECOND,
    30, spawn);
```

See how

[Timer examples in Phaser](#)

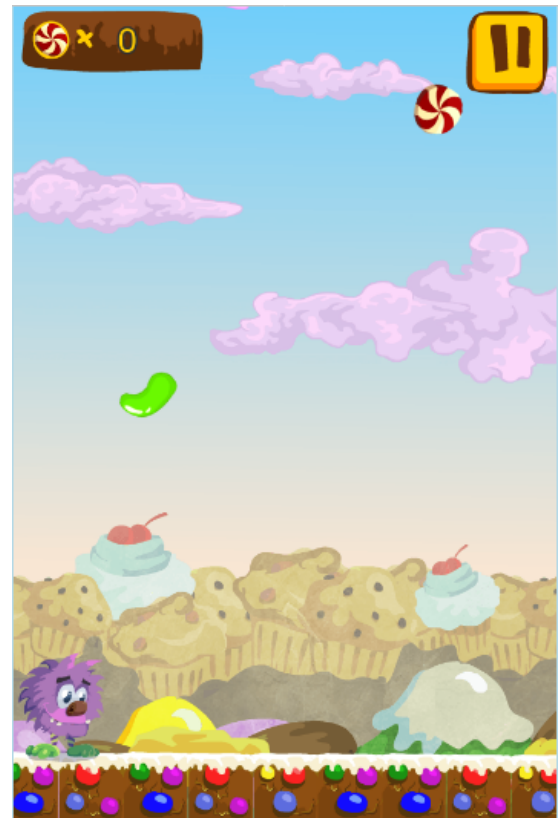
5. Do we have different handler functions for different pieces or a single common handler?
6. Loops over the contents of the group, `piecesGroup`, are not performed with a conventional `for` loop. How is it then?
7. Each piece has an attribute `name` which includes the indices of the piece within the 2D board. What is this name useful for?
8. What does the tween for? Which property is animated with the tween?
9. Would you know how to smoothly rotate 360° the piece while moving to the empty space of the “black” piece?

Monster wants candy

The monster wants candy game is just a little longer and possibly a little more complex, but it is encouraging for



(a) Sliding puzzle



(b) Monster wants candy

Figure 2: Two games to review the key topics in this session

you to appreciate that you already know all that is needed to build it. Well, *almost* everything: this example uses physics which we have deferred until next lab sessions. But do not worry about that: You can still follow the suggested tutorial without any difficulty. An important difference with the other examples in this session is that an object-oriented design is followed. Thus, for a number of reasons, this example is very good to wrap this session up.

Play first and then study how it was built

- [Monster wants candy: the demo game](#)
- [Monster wants candy: the how-to tutorial](#)

Try to answer these questions regarding the “Monster wants candy” game.

1. How many states are added to the game and which one is initially started?
2. What is the size of the canvas where the game elements are rendered on?
3. The monster is displayed using a sprite sheet. How many frames are used to animate the monster? At which rate are the frames changed?
4. This game offers the player the possibility of pausing the game.
 - a) How can the player pause the game?
 - b) How can the player resume the game?
 - c) How is the game actually paused or resumed?
5. Regarding candy spawning
 - a) How often is a new candy object spawned?
 - b) Which variable is used to check for the total elapsed time?
 - c) How are candy objects rotated while they are falling?
 - d) How many different candy objects can be spawned and how are they chosen?
 - e) What do these values `[-27, -36, -36, -38, -48]` represent and how are they used?
 - f) Why is the anchor point for a candy set like this: `candy.anchor.setTo(0.5, 0.5)`?
 - g) Is the rotation velocity of every candy set at the same value? Which is/are this/these value(s)?
6. Regarding collecting candies for the monster
 - a) What's the name of the handler for the user click and where it is set to be so?
 - b) After being clicked, are the candy items reused?
 - c) How are the candy items that are clicked differently treated than those disappearing at the bottom of the screen?
 - d) When is the monsters' health decreased and to what extent?

The tutorial shows us some bits and pieces of the source code, but not the whole of it. You can download the full code and do your own tests for a better understanding.

Get the code and make your own changes!

🔗 [Monster wants candy: Source code](#)

The physics in this game are so simple that we can readily do the same without **Phaser's** physics system. Identify and remove the sentences related to the physics. **Hint:** There are only two things we have to take care of: the gravity for the candy objects to fall down, and checking when they should disappear as they reach the bottom of the game world. Replace the missing functionality with your own code.

This game is a simpler version of the full-fledged game that you can also play.

Play it!

🔗 [Monster wants candy: the full-fledged game](#)

How easy do you think it would be to program this extended version compared to the tutorial version? You're right: it is certainly not much harder.

8 Exercises

1. Modify our `Letters` to count the number of times each vowel has been clicked, and display these five counts in the “over” state.
2. Take the **Phaser** skeleton project, unzip it, change the name of the folder to **TypeIt!**, and open that folder in *VS Code*. The game **TypeIt!** must work as follows:
 - Every 2 seconds, a new random letter is shown at a random position within the canvas.
 - The player should type a letter to remove it.

This is different from the `Letters` game in that it is not the user who decides which letter should be displayed.

TypeIt!



Figure 3: A screenshot of our version of **TypeIt!**.

Apart from that, many parts of the `Letters` code can be reused.

3. Let's follow up. Download two short audio files, load them, and play each for each good and wrong pressed key, respectively. By "good" key press, we mean that the letter corresponding to the pressed key has appeared (and not removed yet); otherwise, the press is "wrong". To save you time, here are two examples of sounds:

[🔊 Punch Or Whack](#)

[🔊 Pew Pew](#)

4. Let's follow up. Define a score which is a function of:
 - The time elapsed between the moment a letter is displayed and the moment the player presses it on the keyboard.
 - The wrong presses (i.e. the user presses keys of letters that are not displayed).

The shorter the time, and the fewer the wrong types, the higher the score should be. Write the code to initialise and update the score upon each key press. Keep in mind that we do not need to update the score regularly, since it only changes when the user types any key. This is a major computational issue you must take into account when designing and implementing games.

5. Let's follow up. Use the score for these purposes:
 - To inform the user, by displaying it prominently.
 - To progressively increase the difficulty of the game, by increasing the rate (speed) at which the new letters are displayed.

At this stage, the game might look similar to Fig. 3.

6. Finally, to complete this game, think about the conditions of winning and losing, and implement them by properly switching to a new **Phaser** state, the "end-game" state. The player should be able to start a new game from the end-game state. Make sure the key presses have no effect while in the end-game state. A single state can be used for both conditions (win and lose) if their contents are not very different.