5. Phaser: a basic Shoot 'em up Game

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

Estimated duration: 3 hours (+ 4 hours of exercises)

> "Code is like humor. When you have to explain it, it's bad." — Cory House (@CodeWisdom)

Shoot 'em up is a well-known genre of games whose basic features (simple player movements, a great body of fires and enemies, no need for realistic physics. . .) are suitable for exploring essential capabilities of **Phaser**. Some of them have already been briefly introduced and need to expand on, like a project organisation based on *states*, group methods for setting up a value or a function on each member, and different uses of timers. Others need to be presented here to properly account for all the game purposes, like the *Arcade system* of the **Phaser** physics. In addition to widen **Phaser** knowledge, in this lab session, complementary mathematical and algorithmic ideas to deal with some behaviours of the game elements are also explored. Moreover, the ability to persistently store data in a browser, by means of the JavaScript *web storage* objects, allows to record the score achievements of the game. This fact has led us to conveniently implement the management of score recordings based on a JavaScript `HallOfFame` class.

Contents

1 Introduction

A wide view of **Phaser** features and capabilities was given in the previous lab session. Now, to deal with the development of a specific, typical game, we elaborate on some of them, among which the convenience of organising the project based on *states* stands out. Also, new necessary features are introduced to properly cover all the game purposes, from which the role of the *physics system* becomes essential.

1.1 Phaser states

The concept of *state* is at the core of **Phaser**. States provide practical mechanisms for handling conceptual sections in a game, like a title screen or a game over screen, or different level screens. They allow to splite up the code of the various sections into separate chunks that can be more easily and logically managed.

Only one state can be active at any given time, though the actions in a game can move from one state to another, and even go back to a state that has already been used or restart the current state.

Two main benefits are obtained from using states:

- The game code is more properly organised, making it easier to build and maintain.

- Game resources, like the assets, can be managed per state, meaning that their use can be fitted in memory as strictly required.

In a previous lab session, we already learnt how to define, add, and start running states in a **Phaser** game. Also, we learnt the main predefined state methods, `preload()`, `create()` and `update()`, also known as *phases* since they are always executed in order. These are the state tasks we will mainly require in the short term, but more properties and methods are available for advanced uses of the **Phaser** states.

See how

- Class StateManager in Phaser
- Class State in Phaser

1.2 Phaser physics

In general, *physics engines* for games try to simulate real world physical behaviours for the objects in a game stage, acting on them phenomena such as velocity, acceleration, friction, bounce, gravity or collisions, and also handling their effects. Normally, processor speed and game playability take priority over accuracy of simulation, leading to physics engines that produce real-time results by simplifying or approximating real world physics. In other words, physics engines for games are usually geared towards providing "perceptually correct" approximations rather than real simulations.

For the time being, **Phaser** provides three physics systems in itself: *Arcade*, *Ninja* and *P2* (Box 1), though a fourth system (*Box2D*) is available via a commercial plugin and other two systems (*Chipmunk* and *Matter*) are under construction. Each game object (such as a sprite) can only belong to one physics system, but multiple systems can be active in a single game.

After starting a physics system, each **Phaser** object in the stage that needs physics simulations requires to associate a *body* in the physics world. Bodies are not visual elements but object's projections onto the physics world, mainly used for calculations.

The example game we are going to develop during this lab session will introduce *Arcade Physics* and some of its features and capabilities. In future theory and lab sessions you will learn more about physics in games, but for now you can explore **Phaser** examples and tutorials to pick up some useful ideas.

Box 1. Phaser physics systems

Arcade Physics is probably the most popular and the one for which most tutorials and examples have been developed. It is the fastest system, but also the simplest. Most notably, all collisions are only checked through the bounding boxes (rectangles) of the objects.

Ninja Physics is slightly more complex than *Arcade Physics*. Originally created for Flash games, it was ported to JavaScript by Phaser's creator Richard Davey. An outstanding feature of this system is that it allows to check collisions with slopes.

P2 Physics is the most complex and realistic system of these three. Its downside is its greater need for computing resources. With this system, elements like springs or pendulums can be properly simulated.

See how

- Phaser Arcade Physics examples

2 BasicShooter game

In this lab session, the exposition of **Phaser** capabilities and some complementary JavaScript objects and functions is guided by the progressive development of a basic shooter, a kind of *Shoot 'em up* game, since the common features of this genre of games (simple player movements, a great body of fires and enemies, no need for realistic physics...) fit properly with our learning purposes.

Your turn 1

Open in *Visual Studio Code* the folder `BasicShooter`, that you will find within `4students` after unzipping the corresponding ZIP archive. Run the project and check its current simple functionality. You will see the *Hall of Fame* screen (`hofState`) after clicking on the spacecraft, as requested in the `initState` texts. Note the structure of the site and check the references to external files in `index.html`. Also, take a look at the code in the files `main.js`, `init.js`, `play.js` and `hof.js`.

2.1 Structure: `index.html` and `main.js`

The site root of the project `BasicShooter` contains some folders, among which `assets/imgs` (for image files), `assets/snds` (for sound files), and `js` include the specific content for the game. In this last folder, four JS files share the code:

- `init.js`, `play.js` and `hof.js` define, respectively, the **Phaser** states `initState`, `playState` and `hofState`.
- `main.js` simply creates the **Phaser** game, adds the three states to the game and starts the `initState`.

The last site file, `index.html`, defines a `<div>` for containing the game stage and loads all the JS files. Very importantly, the loading of files should be done in the appropriate order: first `phaser.js`, then `init.js`, `play.js` and `hof.js`, and finally `main.js`. Since the code in this last file needs the states `initState`, `playState` and `hofState` to be defined, their corresponding files have to be loaded before. Otherwise, undefined name errors will raise.

Placing **Phaser** states in separate files helps to the purposes of properly organising the game code and easing its construction and maintenance. In this way, it's also convenient to place the phases of the states, i.e., function definitions for `preload()`, `create()`, `update()`..., within their corresponding state file.

2.2 `init.js` and `initState`

The file `init.js` contains the definition of the `initState`, the first screen of the game. It is a presentation screen which displays some instructions for the forthcoming `playState`, credits for the assets, and two images, one of which is also a start button (Fig. 1). In general, the first screen of a game could be used for these and other functions: displaying a title or a game story, giving access to a configuration section, introducing characters...

The code of the `initState` should be easy to understand for you. Read it and learn its details to be able to develop some changes in it.

At this moment, just clicking on the spacecraft immediately starts the `playState`. This sudden state transition can be smoothed out by making both graphic elements to move along, up to disappear behind the bounds of the scene. In this way, such an event would now become accountable for the state change. Let's see how to do it!

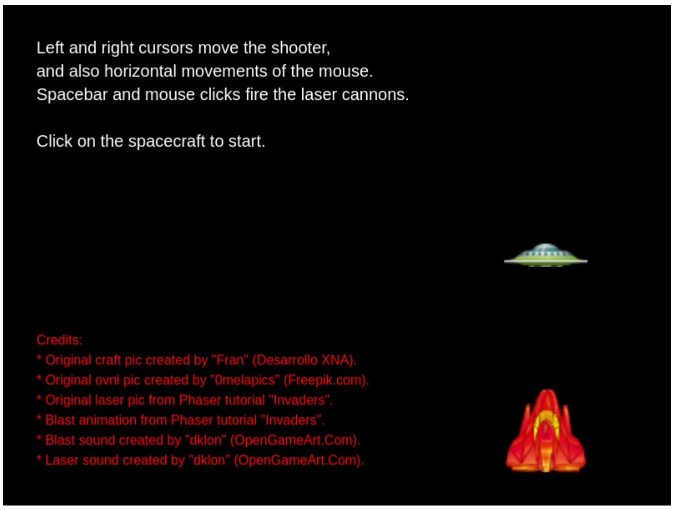

Figure 1: *Presentation screen of the game `BasicShooter`, managed by `initState` as described in Sect. 2.2.*

Your turn 2

Begin by changing the handler of the button in the creation phase within the function `createInit()`:

```
btnStart = game.add.button(
        posX, posY, 'craft', clickStart);
```

Secondly, add the next two lines to the configuration of the button, also within `createInit()`, to activate bounds checking and to register the effective `playState` starter as a handler for the `onOutOfBounds` event (Box 2).

```
btnStart.checkWorldBounds = true;
btnStart.events.onOutOfBounds.add(
                        startPlay, this);
```

Then, at the end of the file, add the definition of the handler that the button click have to run now. This new handler has to be in charge of activating the motion handler on the master timer at a rate of 30 times a second. (Box 3)

```
const FREQUENCY = 1000/30;
function clickStart() {
  btnStart.inputEnabled = false;
  game.time.events.loop(
     FREQUENCY, moveButtonAndImage, this);
}
```

Finally, the definition of the graphics motion handler must be also added at the end of the file. Motion is achieved simply by constantly decreasing the desired coordinate in each graphic item.

```
const DECREASE_Y = 8;
const DECREASE_X = 10;
function moveButtonAndImage() {
  btnStart.y -= DECREASE_Y;
  imgUfo.x -= DECREASE_X;
}
```

Box 2. `checkWorldBounds` & `events.onOutOfBounds`

If its attribute `checkWorldBounds` is set to `true`, the object checks if it is within the world bounds each frame. When it is no longer intersecting the world bounds, it dispatches an `onOutOfBounds` event. If it was previously out of bounds but is now intersecting the world bounds again, it dispatches an `onEnterBounds` event.
When `checkWorldBounds` is enabled, the object calculates its full bounds every frame. This is a relatively expensive operation, especially if enabled on hundreds of objects. So enable it only if you know it's required, or you have tested performance and find it acceptable.
On the other hand, all the **Phaser** objects have an `events` attribute which contains all of the events that are dispatched when certain things happen to it, or any of its components. So, an event like `onOutOfBounds` can register a handler to manage its deliveries.

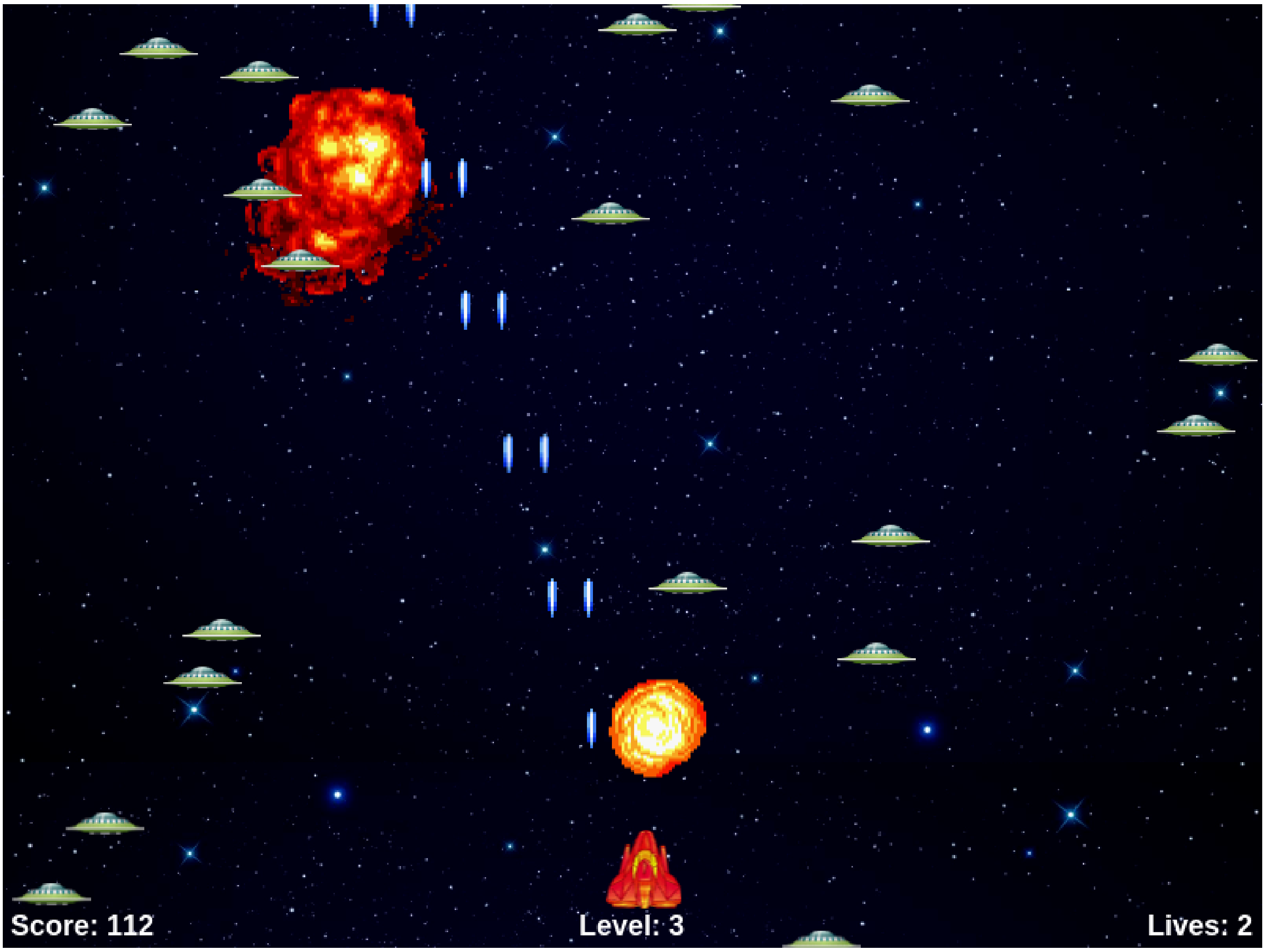

Figure 2: *Snapshot of the play screen of the game `BasicShooter` showing most of the elements that the implementation of the `playState` is going to deal with along the next sections.*

Run the project with the modified `initState` and test the new behaviour of the transition to `playState`.

Box 3. `game.time` and its `events` timer

`game.time` is the *core internal game clock* of **Phaser** and its attribute `events` is the main timer available, bound to this core clock and for which timed events can be added to, through the methods `loop()`, `add()` and `repeat()`. Independent timers can be created through `game.time`, as you already know.

3 Spacecraft

We now begin the development of the state `playState`. In the file `play.js`, the object `playState` and the function `startHOF()` are defined. Our task is to fill the state functions `preload()`, `create()` and `update()` to get a game like that illustrated in Fig. 2.

The first element we incorporate to our scene is a spacecraft, our game player. Just placing an image in the scene is an easy operation.

Your turn 3

First define a global variable for the spacecraft and a global constant to give some space at the bottom, where we will place a Head-Up Display (HUD). The spacecraft will operate on top of the HUD.

```
let craft;
const HUD_HEIGHT = 50;
```

In the function `preloadPlay()`, load the proper file:

```
game.load.image('craft',
                'assets/imgs/craft.png');
```

Then, in the function `createPlay()`, remove now the invocation of `startHOF()` and write a new invocation:

```
createCraft();
```

The actual creation of the spacecraft in the scene is carried out by the following function:

```
function createCraft() {
  let x = game.world.centerX;
  let y = game.world.height - HUD_HEIGHT;
  craft = game.add.sprite(x, y, 'craft');
```

```
  craft.anchor.setTo(0.5, 0.5);
}
```

Run BasicShooter and check the placement of the spacecraft.

A Sprite is a **Phaser** object with a texture, capable of managing animations, input events and physics. It is a more complex object than an Image, since this last object does not allow physics or animation handlers.

The game.world is an abstract place in which all game objects live. By default it is created the same size as the stage, but it is not bound by stage limits and can be any size. The game.world object has attributes, among many others, to get the coordinates of its center point or to get and set its width and height, though remember that the world can never be smaller than the game (canvas) dimensions.

3.1 Keyboard and mouse movements

The next task consists in providing the spacecraft with the ability to move to the left and to the right. Only these movements suffice, since moving backwards is not allowed and moving forward is implicitly done, by scrolling the background, as we will show below. On the other hand, we want to optionally use keyboard or mouse inputs to support these movements.

Your turn 4

Begin by declaring a global variable to account for keyboard inputs and a constant for setting the velocity of the movements.

```
let cursors;
const CRAFT_VELOCITY = 150;
```

In createPlay(), add an invocation to a new function after the invocation to createCraft():

```
  createKeyControls();
```

And, below createPlay(), write the definition for this new function, which creates a controller for all cursor keys:

```
function createKeyControls() {
  cursors =
    game.input.keyboard.createCursorKeys();
}
```

Afterwards, at the end of the function createCraft(), enable the craft in the *Arcade* physics system (Box 4). This, among other features, creates a body attribute in craft, needed for the subsequent code.

```
  game.physics.arcade.enable(craft);
```

Finally, in updatePlay(), introduce a call to another function:

```
  manageCraftMovements();
```

and define it below updatePlay(). It will be in charge of checking the left and right cursors and the mouse movement.

```
function manageCraftMovements() {
  craft.body.velocity.x = 0;
  if (cursors.left.isDown
      || game.input.speed.x < 0)
    craft.body.velocity.x = -CRAFT_VELOCITY;
  else if (cursors.right.isDown
           || game.input.speed.x > 0)
    craft.body.velocity.x = CRAFT_VELOCITY;
}
```

Run the project and check the left and right movements of the spacecraft.

Box 4. game.physics.startSystem()

The method game.physics.startSystem() creates in the game an instance of the requested physics simulation, i.e., the attribute (object) arcade, box2d, ninja or p2 in game.physics. The object game.physics.arcade is running by default, but all the others need to be activated explicitly. For instance,

```
game.physics.startSystem(Phaser.Physics.P2JS);
```

creates the object game.physics.p2. The other two systems (*Ninja* and *Box2D*) require their respective plugins to be loaded before they can be started, since they are not bundled into the core **Phaser** library.

The enable() method of the game.physics.arcade object (Box 4) creates an *Arcade* physics body on the object or the array of objects given as an argument. An object can only have one physics body active at any time, and it can't be changed until the object is destroyed. Afterwards, all *Arcade* physics attributes and methods of the body are available to the object, like the attribute velocity, used here to cause the left and right movements of the craft.

On the other hand, the object game.input.keyboard provides the method createCursorKeys(), which creates and returns an object containing properties: up, down, left and right of Phaser.Key objects. Among many others, each one of these four objects has the property isDown, that will remain true as long as the corresponding key is held down. This property of left and right keys is checked in our code for determining the movement of the craft, as far as the keyboard is concerned.

To check for the left and right movements of the mouse our code tests the horizontal speed of the input manager game.input, negative for left movements and positive for right ones. This speed property accounts for the speed of any pointer active in the game, so that it is only useful in single pointer games.

3.2 Stopping at the world bounds

As you may have realised, the spacecraft doesn't stop at the left and right world bounds. This is a task that we can code ourselves or that we can let the physics system do it for us.

Your turn 5

Provided that we have enabled before a physics `body` in the `craft`, within the function `createCraft()`, now we have only to set up the proper `body` attribute after the enabling sentence.

```
craft.body.collideWorldBounds = true;
```

Run the project and check the new behaviour.

An *Arcade* physics `body` enabled in a **Phaser** object can be set to collide against the world bounds automatically and remain within these world bounds, if the `body` attribute `collideWorldBounds` is set to `true`. Otherwise, it will be able to leave the world.

3.3 Scrolling the background

Phaser facilitates the creation of scrolling backgrounds by means of its `TileSprite` objects. In this way, we will be able to incorporate a vertical scrolling background to our game, which, besides, will support the implicit infinite forward movement of the player.

Your turn 6

First, define a global variable for the stars background:

```
let stars;
```

Secondly, load the appropriate file in `preloadPlay()`:

```
game.load.image('stars',
                'assets/imgs/stars.png');
```

Then, in `createPlay()`, a `TileSprite` is added, just like a standard sprite. It is set to fit the whole dimensions of the world. Should the `TileSprite` adding sentence be placed before or after the call to the creation of the spacecraft?

```
let w = game.world.width;
let h = game.world.height;
stars = game.add.tileSprite(
                    0, 0, w, h, 'stars');
```

Finally, it only remains to increase the `tilePosition` attribute at the beginning of the `updatePlay()` function:

```
stars.tilePosition.y += 1;
```

Run the project and test the new code added.

Our use of a tile sprite is a very simple one. More complex alternatives are available for moving independently the tile sprite and the texture it contains.

Learn different uses of tile sprites

Phaser tile sprites examples

4 Lasers

We now deal with the task of providing the spacecraft with the capability of firing lasers, to eventually destroy the invaders. To that end, a **Phaser** group of sprites will be created whose members will be initially hidden and will be placed in the scene on individual demand.

Your turn 7

First, we declare a global variable for the group of sprites and a constant for its size:

```
const LASERS_GROUP_SIZE = 40;
let lasers;
```

And, in `preloadPlay()`, we load the laser image:

```
game.load.image('laser',
                'assets/imgs/laser.png');
```

At the end of `createPlay()`, we make a call to the effective creation function:

```
createLasers(LASERS_GROUP_SIZE);
```

And, below `createPlay()`, we define this function:

```
function createLasers(number) {
  lasers = game.add.group();
  lasers.enableBody = true;
  lasers.createMultiple(number, 'laser');
  lasers.callAll('events.onOutOfBounds.add',
      'events.onOutOfBounds', resetMember);
  lasers.callAll(
       'anchor.setTo', 'anchor', 0.5, 1.0);
  lasers.setAll('checkWorldBounds', true);
}
```

Finally, we also add the definition of the handler above registered for the `onOutOfBounds` event:

```
function resetMember(item) {
  item.kill();
}
```

Run the project and check that no error raises, since no visible effect should be produced on the scene.

Some properties and methods of the **Phaser** groups are essential in these last lines of code. If it is set to `true`, the property `enableBody` of a group will enable a physics `body` on each member created, or added after to the group. If there are members already in the group at the time this property is set to `true`, they are not changed.

The group method `createMultiple()` automatically creates multiple sprite objects and adds them to the top of the group. It is useful for generating a pool of sprites, such as our laser shots. Initially, all the sprites are positioned at $(0, 0)$, relative to the group (x, y) values, and have their `exists` property set to `false` (to avoid being processed by the core **Phaser** game loops).

The method `callAll()` invocates a function, specified by name (`'events.onOutOfBounds.add'` and `'anchor.setTo'`), on each group member, with a *con-*

text ('events.onOutOfBounds' and 'anchor') and the required *arguments* (resetMember and 0.5, 1.0) specified in the invocation. Similarly, the method setAll() quickly set the same property ('checkWorldBounds') across all members of a group to a new value (true). These methods allow to configure each sprite member of our lasers group for future behaviour: checking when a laser shot is onOutOfBounds, after being fired, and then reseting it (kill()). The function resetMember() will be shared later to do the same with UFOs.

4.1 Fire lasers

To fire a laser, a hidden image member of the lasers group will be retrieved and activated in front of a spacecraft cannon. Similarly to the spacecraft movements, the fire action will be managed by both pushing the spacebar and clicking the left button of the mouse.

Your turn 8

Declare a global variable for the fire button (key) and some constants for managing lasers shots:

```
const LEFT_LASER_OFFSET_X = 11;
const RIGHT_LASER_OFFSET_X = 12;
const LASERS_OFFSET_Y = 10;
const LASERS_VELOCITY = 500;
let fireButton;
```

Also, at the end of createKeyControls(), create the key object for the fire button:

```
  fireButton = game.input.keyboard.addKey(
                    Phaser.Keyboard.SPACEBAR);
```

Then, at the end of updatePlay(), write an invocation for the function managing the shots:

```
  manageCraftShots();
```

Finally, below updatePlay(), write the code for checking the spacebar and the mouse left button to fire lasers:

```
function manageCraftShots() {
  if (fireButton.justDown ||
      game.input.mousePointer.
                leftButton.justPressed(30))
    fireLasers();
}
```

and the code for preparing two simultaneous shots, for the left and right cannons of the spacecraft:

```
function fireLasers() {
  let lx = craft.x - LEFT_LASER_OFFSET_X;
  let rx = craft.x + RIGHT_LASER_OFFSET_X;
  let y = craft.y - LASERS_OFFSET_Y;
  let vy = -LASERS_VELOCITY;
  let laserLeft = shootLaser(lx, y, vy);
  let laserRight = shootLaser(rx, y, vy);
}
```

besides the actual code for retrieving a laser image from the group, placing it in the scene and giving it a velocity:

```
function shootLaser(x, y, vy) {
  let shot = lasers.getFirstExists(false);
  if (shot) {
    shot.reset(x, y);
    shot.body.velocity.y = vy;
  }
  return shot;
}
```

Run BasicShooter and test if it silently fires by means of both the spacebar and mouse clicks.

The method addKey() of the game.input.keyboard manager allows to create a Key object for a given key (Phaser.Keyboard.SPACEBAR). This Key object provides more fine-grained control over the key: tests its status, checks if there are events attached to it, etc. For instance, the property justDown can be checked to know if the key has just been pressed down or not. It will only return true once, until the key is released and pressed down again.

The property mousePointer of the game.input manager is the specific Pointer object to handle the mouse which runs by default. The leftButton object of the mousePointer provides a lot of properties and methods for a great variety of tests. Among them, the method justPressed() returns true if the mouse left button is pressed down within the duration given (in milliseconds). The duration value of 30 ms for our use of this method is adjusted for capturing isolated high speed user clicks.

The function fireLasers() does not contain new **Phaser** features, but note the calculations made, from the (x, y) coordinates of the craft, to place a laser shot in front of each of the two cannons of the craft.

The group method getFirstExists() finds the first member (sprite) of the group that exists (true) or not (false), i.e., that is enabled to be processed by the core **Phaser** game loops or not. Keep in mind that any member of a group sets its exists property to false by default at its creation time (createMultiple()) and every time it runs its kill() method (for instance, when leaving the stage). Thus, after retrieving a non existing laser sprite, it applies the method reset() to set up its new (x, y) position and the value true to its exists property. Also, by means of its physics body, a velocity is assigned to the sprite.

4.2 Laser sound

We will end our work on lasers by adding a sound to each visual shot. This is an easy task that we know well.

Your turn 9

Declare first a global variable for the laser sound:

```
let soundLaser;
```

And, in preloadPlay(), load the proper audio file:

```
  game.load.audio('sndlaser',
                  'assets/snds/laser.wav');
```

After that, at the end of createPlay(), write a call to a function which will create all sounds:

```
createSounds();
```

Then, after `createPlay()`, write the function for creating all sounds. However, at this moment, it will only create the laser sound:

```
function createSounds() {
  soundLaser = game.add.audio('sndlaser');
}
```

Finally, at the end of `fireLasers()`, code the activation of the sound if at least one laser has been fired:

```
if (laserLeft || laserRight)
  soundLaser.play();
```

Run the project and test if the sound plays on the activation of each laser image.

We already knew how to load and add an audio to a game, and how to play it in the appropriate moment. We only want to remark that the existence of one laser sprite, at least, in the scene is required to play the sound. This is an unlikely case, but not impossible for some browser and computer conditions.

5 UFOs

Most of the work to be done for introducing invader UFOs in the game is the same as for the lasers. The differences appear only in the size of the supporting group (40 versus 200), in the way they are initially placed on the screen (keystrokes or mouse clicks versus randomly), in their position (above cannons of the spacecraft versus above the top bound of the stage), and in the direction of their movements (bottom up versus top down).

Your turn **10**

Declare a global variable for a new group of sprites and a constant for its size:

```
const UFOS_GROUP_SIZE = 200;
let ufos;
```

Next, in `preloadPlay()`, load the UFO image:

```
game.load.image('ufo',
                'assets/imgs/ufo.png');
```

Then, at the end of `createPlay()`, invoke the creation function:

```
createUfos(UFOS_GROUP_SIZE);
```

And, below `createPlay()`, define this function:

```
function createUfos(number) {
  ufos = game.add.group();
  ufos.enableBody = true;
  ufos.createMultiple(number, 'ufo');
  ufos.callAll('events.onOutOfBounds.add',
     'events.onOutOfBounds', resetMember);
  ufos.callAll(
     'anchor.setTo', 'anchor', 0.5, 1.0);
  ufos.setAll('checkWorldBounds', true);
}
```

Again, at this moment you can run `BasicShooter` and check only that no error is produced. No perceptible changes in the working of the game can be noticed.

Note that the handler `resetMember` was already defined in Sect. 4. Each UFO which has raised an `'onOutOfBounds'` event needs to run the same action that is executed for each laser.

5.1 Random appearance

We want the UFOs to appear randomly along the width of the screen (`world`), from the outer top bound and limited by the left and right bounds. Moreover, we want their appearance timing be also random.

Your turn **11**

Begin by defining the following global constant and variables:

```
const TIMER_RHYTHM=0.1*Phaser.Timer.SECOND;
let currentUfoProbability;
let currentUfoVelocity;
```

Then, at the end of the function `createUfos()`, write the following sentences to set up probability and velocity values and to register the UFO activation handler on the master timer, at a rate of `TIMER_RHYTHM`.

```
currentUfoProbability = 0.2;
currentUfoVelocity = 50;
game.time.events.loop(
       TIMER_RHYTHM, activateUfo, this);
```

Finally, write the next UFO activation handler:

```
function activateUfo() {
  if (Math.random() <
                      currentUfoProbability) {
    let ufo = ufos.getFirstExists(false);
    if (ufo) {
      let gw = game.world.width;
      let uw = ufo.body.width;
      let w = gw - uw;
      let x = Math.floor(Math.random()*w);
      let z = uw / 2 + x;
      ufo.reset(z, 0);
      ufo.body.velocity.x = 0;
      ufo.body.velocity.y =
                          currentUfoVelocity;
    }
  }
}
```

Run the project and check how the UFOs enter in the scene.

You shouldn't have any problem to understand each individual statement of the code above. But note the way some of them relate to each other. Mainly, the handler `activateUfo()` is called every one-tenth of a second, but not all of its calls do effective work. Due to its initial condition, only a limited percentage of the calls will success to actually activate an UFO. Thus, `Math.random()` and `currentUfoProbability`, along with a thin enough timer rhythm, control the random appearance timing of the UFOs in the scene, at an average rate of approximately two per second (given the current values).

To discover how each UFO selected for activation is randomly and horizontally positioned along the width of the screen (`world`), study the inner sentences of `activateUfo()`. It's an easy calculation in three steps!

6 Collisions

At this point in the development of the game, it's very strange to see lasers overtaking UFOs or UFOs overtaking the spacecraft without destroying themselves. The reason is simple: no collision detection and handling has been implemented in the game. We are going to deal with it right away! Well, strictly speaking, we are going to let *Arcade* physics do it for us.

Your turn 12

Write the next two sentences at the beginning of the function `updatePlay()`:

```
game.physics.arcade.overlap(
    lasers,ufos,laserHitsUfo,null,this);
game.physics.arcade.overlap(
    craft,ufos,ufoHitsCraft,null,this);
```

And, below `updatePlay()`, give an initial simple code for the two callback functions:

```
function laserHitsUfo(laser, ufo) {
  ufo.kill();
  laser.kill();
}

function ufoHitsCraft(craft, ufo) {
  ufo.kill();
  craft.kill();
}
```

The method `overlap()` of the default object `arcade` of the `game.physics` checks whether two different objects overlap. The first two parameters identify these objects, which can be sprites, groups or arrays of objects, even of different types (they can be another **Phaser** objects). A group or array parameter lead to check individually each member for overlapping.

The third parameter is an optional callback function that is called if the objects overlap. If so, two objects will then be passed to it in the same order in which you specified them in the method `overlap()`. This explains the parameter order in our callback functions `laserHitsUfo()` and `ufoHitsCraft()`.

The fourth parameter is a callback function for performing additional checks against the two overlapping objects. If this is set, then the third parameter callback function will only be called if this function returns `true`. We don't need to perform additional checks to achieve our goals, thus it is set to `null` here. The fifth parameter is the context in which to run the callbacks, that in our case is the state object `playState`.

For the moment, the two callback functions only kick the overlapping objects off the screen. Next, we will complement collisions with more details.

6.1 Animated blasts

The destruction of spaceships in a battle is inconceivable without a proper explosion. Therefore, we are going to prepare plausible spacecraft and UFOs destructions by means of an animated blast.

Your turn 13

Declare a global variable for a new group of sprites and a constant for its size:

```
const BLASTS_GROUP_SIZE = 30;
let blasts;
```

Secondly, in `preloadPlay()`, load the spritesheet with the animation frames:

```
game.load.spritesheet(
'blast','assets/imgs/blast.png',128,128);
```

Then, within the function `createPlay()`, before the call to `createUfos(UFOS_GROUP_SIZE)`, invoke the blasts group creation:

```
createBlasts(BLASTS_GROUP_SIZE);
```

Complete this creation by means of two set up functions, one for the group and another for each member:

```
function createBlasts(number) {
  blasts = game.add.group();
  blasts.createMultiple(number, 'blast');
  blasts.forEach(setupBlast, this);
}

function setupBlast(blast) {
  blast.anchor.x = 0.5;
  blast.anchor.y = 0.5;
  blast.animations.add('blast');
}
```

After that, at the end of `laserHitsUfo()`, write a call to the function that activates the explosion:

```
displayBlast(ufo);
```

And, at the end of `ufoHitsCraft()`, write two calls to this function, one for each ship destroyed:

```
displayBlast(ufo);
displayBlast(craft);
```

Finally, write the code for the function that places and plays each single blast:

```
function displayBlast(ship) {
  let blast = blasts.getFirstExists(false);
  let x = ship.body.center.x;
  let y = ship.body.center.y;
  blast.reset(x, y);
  blast.play('blast', 30, false, true);
}
```

Run the project and check the animated explosions.

The method `spritesheet()` of the **Phaser** loader, `game.load`, adds a sprite sheet to the current load queue, ready to be loaded when the loader starts. A sprite sheet is an image file containing frames, usually of an animation, that are all equally dimensioned and often in sequence. For instance, if the frame size is `128x128` (the last two parameters) then every frame in the sprite sheet will be that size.

The group method `createMultiple()` is used again to create a pool of sprites, for also recycling the explosions. Then, another group method, `forEach()`, let us run a function on each created member of the group, to individually set up their desired features. Apart from the callback function name, the context for its execution is given as a second parameter. Other optional parameters of the method `forEach()` allow to pass additional arguments to the callback function, if required. The last step of the function `setupBlast()` consists in incorporating the loaded sprite sheet to each group member property `animations`, an animation manager instance available to each object enabled for animation, such as each sprite object of the group.

In the functions that handle the hits, `laserHitsUfo()` and `ufoHitsCraft()`, a blast is displayed for each ship destroyed in a hit. To do so, the function `displayBlast()` recovers an inactive member of the group, places each explosion in the center of the corresponding ship and plays its animation. The method `play()` runs the animation identified by `'blast'` and previoulsy added via `animations.add()`, at 30 frames per second, without looping (`false`) and killing (`true`) the corresponding sprite when the animation completes.

6.2 Blast sound

To finish the processing of the collisions, an appropriate sound has to be played along with each blast animation. We will simply repeat the actions that were carried out for playing a sound along with each laser shot.

Your turn 14

Declare a global variable for the explosion sound:

```
let soundBlast;
```

Then, in `preloadPlay()`, load the corresponding audio file:

```
  game.load.audio('sndblast',
                  'assets/snds/blast.wav');
```

After that, at the end of `createSounds()`, write the sentence to create the blast sound:

```
  soundBlast = game.add.audio('sndblast');
```

Finally, at the end of both `laserHitsUfo()` and `ufoHitsCraft()`, play the explosion sound:

```
  soundBlast.play();
```

Run again the project and check the animated loud explosions.

7 HUD

The play screen of our game will be completed with a Head Up Display (HUD), to show the score achieved, the level of the game reached and the remaining player lives at any time. We will deal with levels and lives below. For the moment, we will begin by displaying the HUD at the bottom of the screen.

Your turn 15

Declare global variables to store the values of the score, level and lives, and to refer to the associated **Phaser** text objects:

```
let score; // Repeated declaration in hof.js
let scoreText;
let level;
let levelText;
let lives;
let livesText;
```

Next, at the beginning of `createPlay()`, assign the initial values of the main variables:

```
  score = 0;
  level = 1;
  lives = 3;
```

And, at the end of `createPlay()`, write the call to the creation of the HUD:

```
  createHUD();
```

Then, write the code of the HUD creation function:

```
function createHUD() {
  let scoreX = 5;
  let levelX = game.world.width / 2;
  let livesX = game.world.width - 5;
  let allY = game.world.height - 25;
  let styleHUD =
      {fontSize: '18px', fill: '#FFFFFF'};

  scoreText = game.add.text(
     scoreX,allY,'Score: '+score,styleHUD);

  levelText = game.add.text(
     levelX,allY,'Level: '+level,styleHUD);
  levelText.anchor.setTo(0.5, 0);

  livesText = game.add.text(
     livesX,allY,'Lives: '+lives,styleHUD);
```

```
  livesText.anchor.setTo(1, 0);
}
```

Finally, at the end of `laserHitsUfo()`, add the sentences to internally and externally update the score:

```
score++;
scoreText.text = 'Score: '+score;
```

Similarly, at the end of `ufoHitsCraft()`, add the sentences to update the lives:

```
lives--;
livesText.text = 'Lives: '+lives;
```

Run `BasicShooter` and check the appearance of the HUD and its basic updating.

The variables `score`, `level` and `lives` are initialised before any call to a creation function because they are going to be used in some of these functions, concretely in `createUfos()` (see next subsection) and `createHUD()`.

In the function `createHUD()`, observe the horizontal coordinates set up for placing the `score`, `level` and `lives`, and relate them with the corresponding positions of their `anchor` points. Check the placement of these three texts on the bottom of the screen, derived from the values of their horizontal coordinates and `anchor` points. The vertical coordinate along with the style established are common to all three texts.

7.1 Increasing difficulty through the levels

The variable `level` has been introduced to control the level of difficulty of the game. We want to set five levels in our game, each one more difficult than the previous, and to change the level as the score increases: every fifty points a new level is reached, up to the last one, which will stay until the end of the game. The difficulty of each level is set to depend on the probability of appearance of the UFOs and on their velocity, increasing both as the level ups.

Your turn **16**

Declare the next global constants:

```
const NUM_LEVELS = 5;
const LEVEL_UFO_PROBABILITY =
                  [0.2, 0.4, 0.6, 0.8, 1.0];
const LEVEL_UFO_VELOCITY =
                   [50, 100, 150, 200, 250];
const HITS_FOR_LEVEL_CHANGE = 50;
```

Within the function `createUfos()`, change the assignments of the initial probability and velocity of the UFOs:

```
currentUfoProbability =
         LEVEL_UFO_PROBABILITY[level-1];
currentUfoVelocity =
            LEVEL_UFO_VELOCITY[level-1];
```

Then, at the end of `laserHitsUfo()`, write the code to manage the update of the level, and the probability and velocity of the UFOs:

```
if (level < NUM_LEVELS &&
  score===level*HITS_FOR_LEVEL_CHANGE) {
  level++;
  levelText.text = 'Level: ' + level;
  currentUfoProbability =
         LEVEL_UFO_PROBABILITY[level-1];
  currentUfoVelocity =
            LEVEL_UFO_VELOCITY[level-1];
}
```

Run the project and check the working of this code.

Note the use of the two arrays to define the probability and velocity values, which are indexed by means of the variable `level`. Note also the condition for advancing to the next level, which relates the variables `score` and `level`, and the constant `HITS_FOR_LEVEL_CHANGE`, if the current level is not the last one.

7.2 Spacecraft replacement

To end the playing section of our game, we are going to manage spacecraft replacement, since if there are lives left, the screen has to be reinitialised. Otherwise, the **Phaser** state has to be changed. Besides, every time a spacecraft is destroyed, during the following 2 seconds, we want to:

- clear the stage for remaining UFOs and lasers,
- disable all input signals (keyboard and mouse), and
- deactivate new appearances of UFOs in the stage.

At the end of this lapse, the game has to continue by checking the variable `lives` to decide if the game should be reinitialised or the state `hofState` should be started. In both cases, the input signals have to be enabled again. And, in the first case only:

- a new spacecraft has to be placed in its standard initial position on the stage, and
- new appearances of UFOs in the stage need to be reactivated, according to the value of the current level.

Additionally, also in the first case, reseting the left and right cursors is likely to be needed, since their eventual previous down status could remain active.

Your turn **17**

At the end of the function `ufoHitsCraft()`, write the following sentences:

```
ufos.forEach(clearStage, this);
lasers.forEach(clearStage, this);
game.input.enabled = false;
currentUfoProbability = -1;
game.time.events.add(
               2000, continueGame, this);
```

Then, write the code for the clearing function:

```
function clearStage(item) {
  item.kill();
}
```

Finally, write the callback function for the timer event:

```
function continueGame() {
  game.input.enabled = true;
  if (lives > 0) {
    let x = game.world.centerX;
    let y = game.world.height - HUD_HEIGHT;
    craft.reset(x, y);
    cursors.left.reset();
    cursors.right.reset();
    currentUfoProbability =
           LEVEL_UFO_PROBABILITY[level-1];
  }
  else
    startHOF();
}
```

Run `BasicShooter` and check its complete functionality, but mainly the last features introduced.

Relate the above mentioned desired features with their implementation in the previous lines of code. Most of them should be understandable for you.

To deactivate new appearances of UFOs in the stage, a value smaller than zero is assigned to the variable `currentUfoProbability` to let the initial condition of the function `activateUfo()` evaluate to false. Afterwards, the UFO probability of the current level is recovered. Remember that, during the 2 seconds lapse, the master timer is working and we only want to suspend some features.

On the other hand, note that the method `add()` of the master timer `game.time.events` (in general, any timer) lets you register a timed handler to be run only once. As arguments, you have to provide a delay (in milliseconds), a callback function to be run after the delay, and a context (`playState` in this case). Optional arguments can also be passed, in case the callback function needs them.

8 Hall of Fame

To complete our implementation of a game based on **Phaser** states, we are going to dedicate the third state to handle a *Hall of Fame* for the game. Strictly speaking, it won't be a proper Hall of Fame since it won't record identified users (persons) and it will be local (per origin —domain and protocol). Fig. 3 illustrates the kind of screen we are going to display and manage, for showing the top ten shooting sessions (with positions, scores and dates), a message related to the last session run and another one to allow the user to restart the game.

8.1 Web storage

Before paying attention to the implementation of a Hall of Fame manager, we need to talk about the current kinds of storage available in a browser. Since the arrival of HTML5, *web storage* lets web applications store data locally, in a more secure way and with larger capacity limits than traditional *cookies*, along with avoiding the inclusion of the stored data in every server request, that cookies do require. Web storage is done per origin (per domain and protocol).

Figure 3: *Score recording screen of the game* `BasicShooter` *managed by* `hofState` *as described in Sect. 8.*

HTML5 web storage provides two objects for storing data at the client:

`localStorage`: Stores data persistently, with no expiration date.

`sessionStorage`: Stores data only for one session. The data is lost when the browser tab is closed.

HTML5 web storage objects store simple key-value pairs, similar to JavaScript objects. Keys and values are always strings, using automatic conversions when they are needed. The basic available methods for handling these objects are: `getItem()`, `setItem()`, and `removeItem()`, but alternative forms are allowed for the first two ones:

```
localStorage.name = 'Shooter';
localStorage['name'] = 'Shooter';
localStorage.setItem('name', 'Shooter');
let myname = localStorage.name;
myname = localStorage['name'];
myname = localStorage.getItem('name');
localStorage.removeItem('name');
```

For an effective storage of all JavaScript objects, to combine these methods with conversions into JSON is recommended.

Learn more features and details

- Web storage

8.2 JavaScript class `HallOfFame`

A JavaScript class named `HallOfFame` has been developed for modelling the permanent score recording of the best results in our game. The class `HallOfFame` is located in the file `hof.js` and provides basic methods for loading from and saving to `localStorage`, adding a new score to the best scores and maintaining the order of the list, and displaying this list on a **Phaser** stage.

Your turn 18

Open the file `hof.js` and take a look at the code of the class `HallOfFame` while, at the same time, you read the next descriptions.

A class declaration `HallOfFame` allows to instantiate an object created to manage a list of a specified size. Then, all of its methods are defined inside this class declaration, which implies that they are actually implemented by means of the property `prototype`: to be used by any object through inheritance and, thus, to save space in memory.

The `loadFromStorage()` and `saveToStorage()` methods are in charge of interfacing with `localStorage`, which is the proper storage object for our purposes, in contrast to `sessionStorage` or cookies. In both methods, note the use of the JSON methods `parse()` and `stringify()` to easily and reliably transform data between the strings required by the storage object and the integer and the array internally used in our class. In `loadFromStorage()`, note also the previous tests for ensuring key-value pairs exist in `localStorage` before trying to make use of them. This situation will happen, at least, every time `localStorage` is set initially empty at the beginning of an origin score recording.

The three methods `resetStorage()`, `getSize()` and `setSize()` are not used in our remaining code, but are provided here for a foreseeable complete implementation of the class.

The method `addNewScore()` is in charge of inserting in order a new score in the internal ordered list of the class, which represents the best scores recorded for the `HallOfFame` list. Each new score aggregates the complete timing data of the instant that it is recorded. Obviously, if a new score is to be inserted in a previously complete list, given the specified size, its current last item has to be removed, preserving this size. At the end, this method returns an integer value with the index of the inserted score in the list, if it has been actually inserted, or `-1`, if it has not been inserted (it is a score worse than the last one recorded).

The method `displayOnStage()` writes the `HallOfFame` list on a **Phaser** stage. It receives four parameters: the coordinates `x` and `y` for the top left position of the list on the **Phaser** world, and the width (`w`) and height (`h`) that the list must take up. Then, the title line and all the lines and columns of the list are scaled to fit in the requested width and height, and they all are placed on the screen from the received coordinates `x` and `y`. To get this writing, some calculations are first carried out in order to then use the method `setTextBounds()`, of the **Phaser** text objects, for sizing and organising frames for all text items. The style properties of **Phaser** texts, `boundsAlignH` and `boundsAlignV`, take also part of the process, aligning each text within its frame. Note that the method `setTextBounds()` also requires four arguments, similar to those of our method: the coordinates x and y for the position of the text frame (dependent of the anchor set up for the text object), and the width and height of the text frame. All these four arguments are interpreted in relation to the world dimensions of the game.

8.3 `hof.js` and `hofState`

The **Phaser** state `hofState` is the main object in the file `hof.js`. It leads the actions on the stage and behind the scenes of this last game state. `hofState` only uses the phases (functions) `preload()` and `create()` since they suffice for displaying the score record and simple messages, and waiting for the user response.

The function `preloadHOF()` is in charge of creating the `HallOfFame` object, using the global variable `shooterHOF`, and immediately loading in it the stored score recording. Then, the insertion of the new score (coming from `playState`) in the object `shooterHOF` list is attempted, and its integer result is checked for creating an appropriate string message for the user. Finally, the current status of the object `shooterHOF` is saved back to the permanent storage.

Note the use of the function `ordinalNumAbbrev()` for adding the proper suffix to the ordinal number abbreviation. Read the code for this function and realise that it is general, for any positive number.

The function `createHOF()` generates on the screen the Hall of Fame processed in `preloadHOF()`, fitted to the given position and dimensions by means of the arguments for the method `displayOnStage()`. Two additional messages are also fitted to specific positions and dimensions on the screen, now by using again the text method `setTextBound()`. The first message is that prepared in `preloadHOF()` to notify the user of the positive or negative result of trying to insert the last score in the Hall of Fame. The second message simply informs that the user can restart the game.

To this end, a specific `Key` object for the `R` key is created. On the signal property `onDown` of this object, dispatched every time the key is pressed down, the callback function `restartPlay()` is registered by means of the method `addOnce()`, which adds a one-time listener for the signal. The function `restartPlay()` simply starts again the state `playState`.

Your turn 19

Run several times the project and test all the elements appearing in `hof.js`. Particularly, check different positions at the score recording list.

References

To prepare this document we have found inspiration in, basically, these books:

- Travis Faas. *An Introduction to HTML5 Game Development with Phaser.js*, CRC Press, 2017.
- Stephen Gose. *Phaser.js Game Design Workbook*, Leanpub, 2018.

A good place to start with the **Phaser** framework is https://phaser.io/:

- **Phaser** Learn
- **Phaser** Examples.

9 Exercises

1. The functions `createLasers()` and `createUfos()`, included in the file `play.js`, share part of their code: their first six sentences are the same, except for the names of the groups and the labels linking to the appropriate images. Factorize the common code of these two functions, by writing an auxiliary new function running the common actions and conveniently calling it from the two functions, substituting the current shared code.

2. Modify `BasicShooter` to allow the spacecraft (player) to also move explicitly up and down over the screen. Let these movements be controlled either with the up/down arrow keys or with the mouse movement, at the player's wish.

   Be aware of stopping at the world upper bound and, mainly, at the HUD upper limit, i.e., at the initial vertical coordinate of the spacecraft itself.

3. Modify `BasicShooter` to add *asteroids* to the play state. Find the image of a rock and create a **Phaser** group with many different instances of this image, varying randomly at the same time their scale and rotation, but limiting their maximum size to 10 % of the world width, for instance.

   Make them go downwards and appear at random horizontal positions, like UFOs, and increase their probability and velocity as level changes but always being less frequent and slower than UFOs, for instance, a quarter of the UFOs probability and velocity in each level.

   Make UFOs pass over them and lasers collide with them. When a laser collides with an asteroid, both will only dissapear (no explosion at all). On the contrary, when an asteroid collides with the spacecraft, the craft has to be destroyed as before, with an explosion.

   For other asteroid behaviours, imitate those of the UFOs: disappearing at the bottom bound to be recycled, cleared from the screen and deactivated for 2 seconds on each spacecraft replacement, and so on.

4. Modify `BasicShooter` to add a new **Phaser** state `otherPlayState`, alternative to the current `playState`. For instance, make it start by clicking on the UFO in the `initState` and end jointly with `playState` at `hofState`, merging their scores in the Hall of Fame. Then, the restarting option should return to `initState` in place of `playState`, to select which playing state the user desires.

   Afterwards, implement `otherPlayState` as you want but developing an alternative shooter. Some ideas for you to consider are: different spacecraft movements, different arrangement or movements for the invaders, invaders also firing lasers or bombs, invaders with shields that require increasing numbers of shots to be destroyed, etc.

   You should also explore the next **Phaser** tutorial to get more ideas:

   Phaser invaders.