



Phaser - Introduction

Lesson Plan

Overview & Purpose

Introduction to game development using the HTML 5 Phaser Framework.

Objectives

1. Learn how to set up the basic HTML code structure for a single screen Phaser game.
2. Learn how to instantiate the Phaser Game Engine using the Phaser.Game class and a configuration object.
3. Learn about Phaser.Scene class and its main functions: preload, create and update.

Prerequisites

1. Students should be familiar with HTML, CSS and JavaScript basics.

2. Students should be familiar with Computer Science core concepts that include: variables, conditionals, loops and functions.
3. Students should have some knowledge of advanced concepts like Classes and Objects in JavaScript.

Materials Needed

1. Machine [MacOS, Windows, Linux]
2. Editor [Sublime Text, MS Visual Studio]
3. Server [`http-server`]

Verification

Steps to check for student understanding

1. Students should be able to answer what Phaser is and list the core functionalities it offers.
2. Students should have successfully built the basic HTML code structure for Phaser.
3. Students should understand the purpose of configuration object and the preload, create and update functions belonging to Phaser.Scene class.



Introduction

Phaser5 is an HTML- 5 game development **platform framework** for building 2D games. It is a very fun and easy to learn framework. It has many features that make game development easy such as image rendering, sprites and creation of animations , handling of input control that includes mouse and keyboard events and three physics engines that can be used for things like collision detection, speed and acceleration settings and many more exciting things!

There are a number of ways of incorporating the Phaser framework in your project. These lesson series will focus on being simple and straightforward by using the boilerplate code pattern available on Phaser's official website. The boilerplate code is a single HTML document. It includes the Phaser framework available on jsDelivr CDN. Both CSS and JavaScript are also incorporated in this single document using HTML style and script elements.

/*To understand the framework, we will start building a simple game called Shoot the Stars. As we build, we will learn along all the way. */Let's look at the HTML boilerplate code structure in detail.

Glossary

A **platform framework** is a collection of code, tools and runtime environment that provides a building block or predefined structure for creating another software. This makes it easier for other developers to reuse and extend the framework functionalities by building their applications on top of it and customizing the functionalities according to their software requirements.

Boilerplate HTML that includes Phaser, CSS and JavaScript

```
1  <!Doctype html>
2  <html lang="en">
3  <head>
4      <meta charset = "UTF-8">
5      <title>My First Game</title>
6      <script src="https://cdn.jsdelivr.net/npm/phaser@3.15.1/dist/phaser-arcade-physics.min.js">
7          </script>
8
9      <style type = "text/css">
10
11          body{
12
13              margin-top: 0px;
14
15          }
16
17      </style>
18  </head>
19
20  <body>
21      <script type ="text/javascript">
22
23          var config = {
24              type: Phaser.AUTO,
25              width: 800,
26              height: 600,
27              scene: {
28                  preload: preload,
29                  create: create,
30                  update: update
31              }
32          };
33
34          var game = new Phaser.Game(config);
35
36          function preload ()
37          {
38
39          }
40
41          function create ()
42          {
43
44          }
45
46          function update ()
47          {
48
49          }
50
51      </script>
52  </body>
53
54  </html>
```

Brief overview of the HTML document

The HTML

1. The HTML document starts with a document type declaration followed by the `<html>` element. The `<html>` element is the root container in an HTML document and holds different elements of the document. It consists mainly of the `<head>` and `<body>` elements.
2. The `<head>` element defines the meta-data of the document such as the character encoding, title of the document, links to additional scripts and style elements for writing CSS.
3. The `<body>` element is the document body which is the visible part of the HTML document. The JavaScript goes inside of it under the `<script>` element.
4. Within the HTML's `<head>` element, a `<script>` element is used to acquire the Phaser script from the jsDelivr CDN . It is done by setting the `src` attribute of the `<script>` element to the specified url. The line of code is circled in green in the screenshot below.

```
4 <!Doctype html>
5 <html lang = "en">
6 <head>
7   <meta charset = "UTF-8">
8   <title>Title of Game</title>
9   <script src = "https://cdn.jsdelivr.net/npm/phaser@3.15.1/dist/
  phaser-arcade-physics.min.js"></script>
10  <style type = "text/css">
11    body{
12      margin:0;
13      text-align:center;
14    }
15  </style>
16
17
18 </head>
```

The CSS

5. The `<style>` element uses the body selector and sets the *margin-top* property to a value of *0 pixels* and *text-center* property to *center* so that the scene/canvas is drawn in the center of the browser.

```
4 <!Doctype html>
5 <html lang="en">
6 <head>
7   <meta charset="UTF-8">
8   <title>Title of Game</title>
9   <script src="https://cdn.jsdelivr.net/npm/phaser@3.15.1/dist/
  phaser-arcade-physics.min.js"></script>
10  <style type="text/css">
11    body{
12      margin:0;
13      text-align:center;
14    }
15  </style>
16
17
18 </head>
```

The JavaScript

6. The `<body>` element includes the `<script>` element. This is where all the JavaScript will go.

The Configuration Object

7. The `<script>` element has a variable defined called `config` of type object `{ }`. It is the configuration object. It has a couple of `{key:value}` pairs. These `{key:value}` pairs are some basic properties that can be passed as an argument to the

Phaser.Game constructor in order to instantiate a Phaser.Game object.

8. The configuration object contains all the values for what type of renderer to be used, the height and width of the game screen and the scenes that are part of the game. There are more properties. If we don't set them, they are initialized with default values by Phaser.

```
19 <body>
20 <script type = "text/javascript">
21
22   var config = {
23
24     type: Phaser.AUTO,
25     width: 400,
26     height: 450,
27     backgroundColor: '#3598db',
28     scene: {
29       preload: preload,
30       create: create,
31       update: update
32     },
33     physics: {
34       default: 'arcade'
35     }
36   }
37
```

The Phaser.Game Object

9. Phaser.Game object is the main controller of the Phaser framework. It does all the initial setup for the game which includes booting, parsing the configuration object that has been passed to it, creating a renderer and making the sound and input controls available. After that it calls the Scene Manager which is responsible for starting the first scene in the game. A Phaser

game is made up of scenes and each scene has the actual game logic. Following code instantiates the Phaser.Game object:

```
var game = new Phaser.Game(config);
```

For more details you can browse through the official Phaser documentation website:

<https://photonstorm.github.io/phaser3-docs/Phaser.Game.html> .

The Phaser.Scene Object

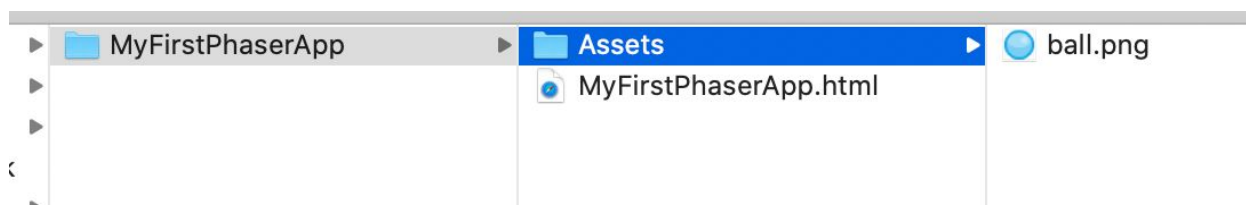
10. The above configuration object sets its scene key's values to some predefined method names which are to be overridden by our code. These are later declared in the code after instantiating the Phaser.Game object. These predefined functions are: preload, create and update. Each of these methods has a specific purpose.

preload():

In this predefined function, we can use the Loader object to load external assets into our game. Our assets can be sprites, images, sounds, texture atlases and data files. A Scene object has a unique Loader object associated with it. However, whatever assets the Loader of any Scene loads, it gets stored inside of the global game-level cache and hence are accessible to other Scene objects in the game.

Try -out:

Let's find an image and download it to our machines and save it at location where your html file is stored inside of a folder called *Assets*. Your directory structure should look like this:



Inside of your preload function type:

```
this.load.image('key', 'path');
```

The above line of code is how we use the Loader object. The *this* keyword points to the Scene object you are working on. The Scene object has a `load.image()` function that is used to access the Loader. The *key* would be a string that you would use later in your code to access the loaded asset. The *path* is a string that is the location of your asset relative to your HTML file.

```
42
43 ▾  function preload(){
44
45      |this.load.image('ball','Assets/ball.png')
46
47      |
48      }
```

create():

This function is where you can actually create your loaded assets, display them, give them properties and use them according to

your game requirements. It is automatically called after the `preload()` function.

Try-out:

Now let's display our image. Declare an empty global variable after the `Phaser.Game` instantiation and call it *ball*.

Then type in the following code inside of your `preload()` function:

```
ball = this.add.image(x , y , 'key');
```

The above line of code will store the created game object in the *image* variable. The function `this.add.image()` takes in 3 arguments. The first two arguments: *x* and *y* set the location of your game object relative to canvas; (0,0) being the top-left coordinate of the scene/canvas. The *key* is the string that you had provided to the Loader in your `preload` function for your asset.

```
37
38     var game = new Phaser.Game(config);
39
40     var ball;
41
42
43     function preload(){
44         this.load.image('ball','Assets/ball.png')
45     }
46
47
48
49     function create(){
50
51         ball = this.add.image(10,30,'ball');
52
53     }
```

update():

This function is called 60 frames every second. It is a function that loops as long as your game Scene is running. It updates and

redraws the game objects. This property of the function allows us to add event handling logics such as collision detections, keyboard and mouse events and many more.

Try-out:

Now let's try moving our image from left to right. Type in the following line of code inside of your `update()` function:

```
ball.x +=10;
```

You will see your image move to the right and then disappear as we have not added any code to stop it if it goes out of the scene boundaries.

```
55     function update(){  
56  
57         ball.x += 10;  
58  
59     }
```

Save your work.



To test your code:

1. Open up your terminal/command prompt,
2. Navigate into your folder where your html file,
3. Type *http-server* to start the server,

```
salah@salahs-MacBook-Pro MyFirstPhaserApp % http-server
Starting up http-server, serving ./
Available on:
  http://127.0.0.1:8080
  http://192.168.0.45:8080
Hit CTRL-C to stop the server
```

4. Open up your browser and type in any one of the urls shown,
5. Since our main file is not called index.html, our files will be displayed on the browser,

Index of /

 (drwxr-xr-x)		Assets/
 (-rw-r--r--)	8.0k	.DS_Store
 (-rw-r--r--)	886B	MyFirstPhaserApp.html

Node.js v12.14.0/ [ecstatic](#) server running @ 192.168.0.45:8080

6. Click on the MyFirstPhaserApp.html to run it,
7. You should see your ball move from left-to-right.