Joao Ortiz

Ojoao\_funkyman@hotmail.com

Descripción breve

[Dibujar su lector con un resumen de la participación. Normalmente es un breve resumen del documento.   
Cuando esté listo para agregar contenido, haga clic aquí y empiece a escribir.]



Manual de phaser 3

[Subtítulo del documento]

Contenido

[1. Descarga de Phaser 2](#_Toc139396544)

[1.1. Descarga de Phaser con Node NPM 2](#_Toc139396545)

[1.2. Incluir Phaser en HTML5 2](#_Toc139396546)

[1.3. Extensión Live Server para Visual Studio 3](#_Toc139396547)

[2. Configuración de Phaser (init.js) 6](#_Toc139396548)

[2.1. Carga de Imágenes 7](#_Toc139396549)

[2.2. Carga de Audios 11](#_Toc139396550)

[2.3. Añadir físicas 11](#_Toc139396551)

[2.4. Control con el teclado 13](#_Toc139396552)

[2.4.1. Método en escucha 13](#_Toc139396553)

[2.4.2. Asignar cursor predeterminado 13](#_Toc139396554)

[2.4.3. Crear cursores personalizados 14](#_Toc139396555)

[2.5. Escenas 14](#_Toc139396556)

[2.6. Módulos 17](#_Toc139396557)

[2.7. Animación 19](#_Toc139396558)

[2.7.1. Sprite 19](#_Toc139396559)

[2.7.2. Atlas 20](#_Toc139396560)

# Descarga de Phaser

## Descarga de Phaser con Node NPM

El primer paso es crear la carpeta contenedora del juego, por ejemplo “PHASER\_CONFIGURACION”.

Desde la consola de comandos de Visual Studio realizar un npm install para instalar Phaser desde la carpeta que hemos creado anteriormente.

Para abrir la consola desde Visual Studio Code podemos hacerlo con CTRL + Ñ

|  |
| --- |
| $ npm install --save phaser |

Este comando nos creará la carpeta node\_modules donde se encuentran todos los módulos necesarios para el arranque de Phaser.

Si queremos subir nuestro código a un repositorio como GitHub no es necesario incluir esta carpeta ya que se puede volver a generar a través de los ficheros de configuración package.json de nuestro proyecto. Para ello ejecutamos npm install y se volverán a generar todos los archivos

|  |
| --- |
| $ npm i |

## Incluir Phaser en HTML5

Dentro de la carpeta “PHASER\_CONFIGURACION” donde hemos instalado Phaser creamos el fichero index.html y añadimos la estructura por defecto como se ve en la siguiente imagen:

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta http-equiv="X-UA-Compatible" content="IE=edge">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

</body>

</html>

Dentro de body creamos un div “container” donde estará alojado nuestro juego.

Para enlazar phaser tenemos que añadir la etiqueta script indicando donde esta nuestro fichero pahser.js . Este fichero siempre va estar alojado dentro de la carpeta node\_modules que se ha creado después de la instalación de Phaser.

Mediante la etiqueta script también añadimos un fichero init.js . Este fichero es donde vamos a realizar la configuración inicial para arrancar Phaser.

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta http-equiv="X-UA-Compatible" content="IE=edge">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

    <div id="container"></div>

    <script src="./node\_modules/phaser/dist/phaser.js"></script>

    <script src="./src/init.js"></script>

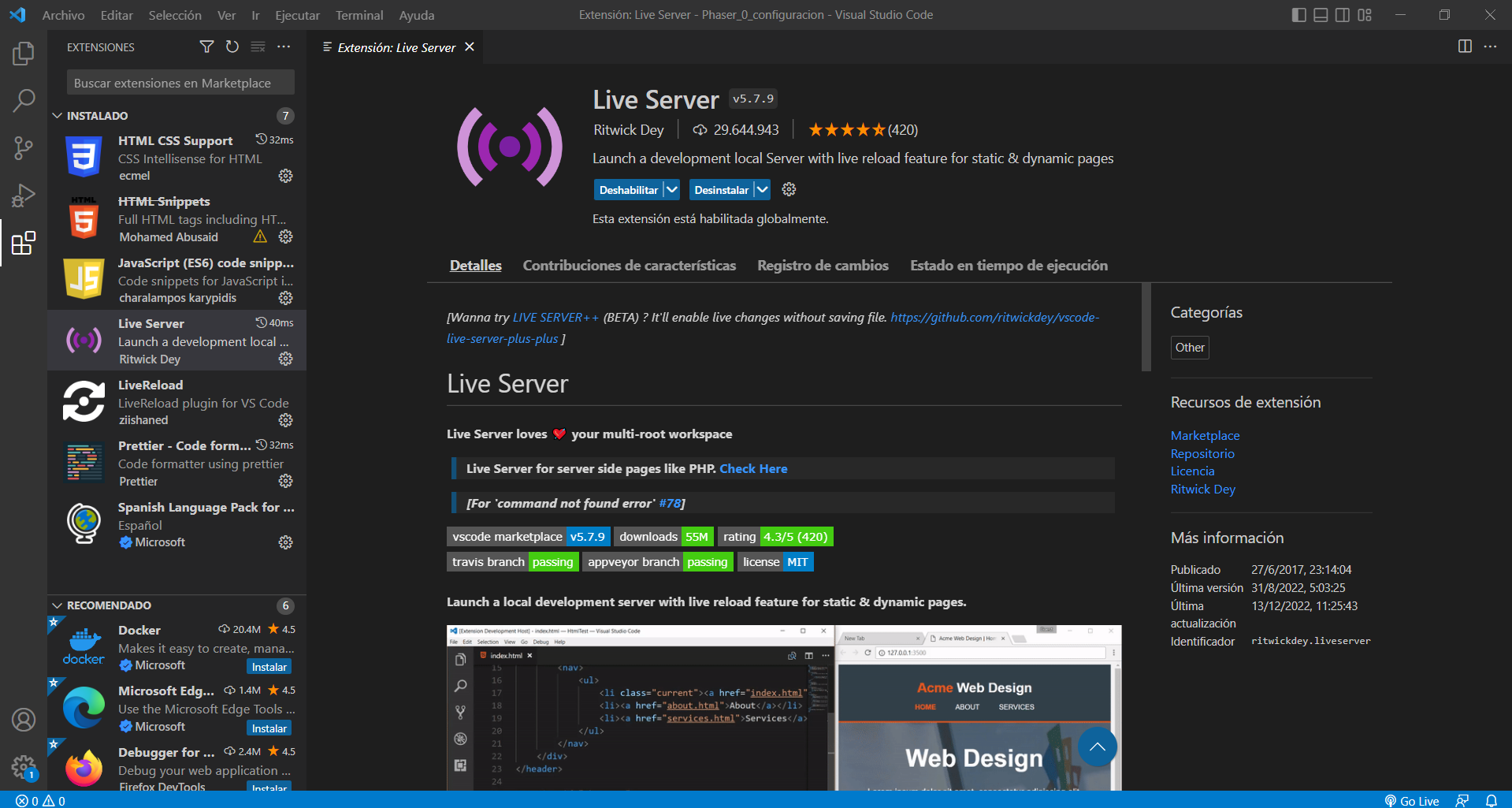
</body>

</html>

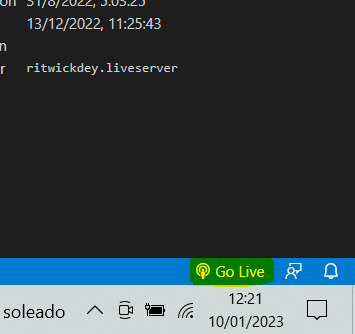
## Extensión Live Server para Visual Studio

Esta extensión de Visual Studio Code nos permite crear un servidor en vivo e ir actualizando la página sin necesidad de presionar CTRL + R

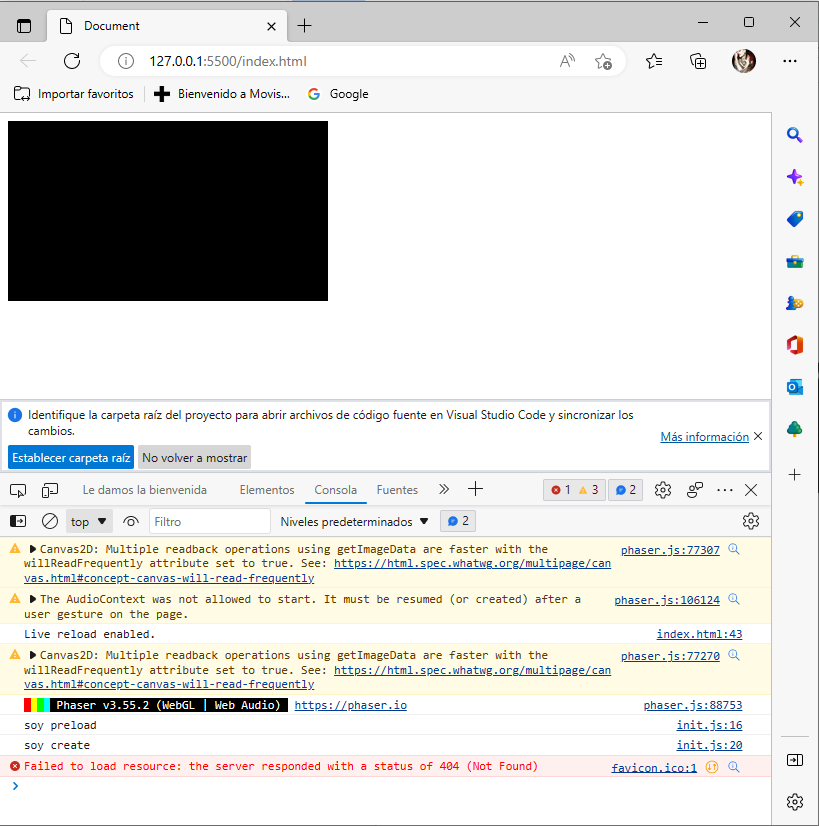
Para instalarlo nos vamos a las extensiones y buscamos Live Server



Una vez instalado en la esquina inferior derecha del visual Studio aparecerá un icono Go Live como se ve en la siguiente imagen.



Si presionamos sobre el icono este cargara el servidor en una página del navegador Web donde se ejecuta Phaser.



# Configuración de Phaser (init.js)

En este fichero indicaremos las variables de configuración para poder inicializar Phaser. Estas variables vienen indicadas en un Json config y son inicializadas creando una nueva instancia new Phaser.Game(config);

const config = {

    width:320,

    height: 180,

    parent: "container",

    type: Phaser.AUTO,

    scene: {

        preload: preload,

        create: create,

        update: update

    }

}

var game = new Phaser.Game(config);

function preload(){

    console.log("soy preload");

}

function create() {

console.log("soy create");

}

function update(time, delta) {

    //console.log(delta)

}

**Width**: Ancho del contenedor.

**Height**: Alto del contenedor.

**Parent**: el contenedor padre "container".

**Type**: Puede recibir los siguientes parámetros: phaser.CANVAS, phaser.WEBGL y phaser.AUTO

**Scene**: escenas del juego

**Function** **preload**: renderizar elementos. En esta función se pueden cargar imágenes, sonido, movimientos, físicas.

**Function** **create**: create:

**Function** **update**: en esta función se ejecuta un loop infinito, recibe 2 parámetros que son el tiempo y delta

**Tiempo**: tiempo.

**Delta**: se usa para que en todos los navegadores que se ejecuta el juego el movimiento sea el mismo.

## Carga de Imágenes

Para cargar las imágenes en Phaser se usa la función **load.image**(nombre, ruta) en la función preload() de nuestra escena.

function preload(){

    this.load.image("pajaro","./assets/bird.png");

}

Esta función load.image() soporta 2 parámetros , por un lado el nombre interno de Phaser y por otro lado la ruta donde se encuentra la imagen.

Una vez precargada la imagen debemos cargarla en la función create para que se muestre en pantalla mediante la función add.image(x, y, nombre).

function create() {

    this.pajaro = this.add.image(50,100,"pajaro");

}

Esta función **add.image**() recibe 3 parámetros, primer parámetro es el eje X donde se va posicionar la imagen en el lienzo, el segundo parámetro es el eje Y, y un tercer parámetro es el nombre interno de la imagen.

Para trabajar con la imagen podemos usar las funciones específicas de Phaser que se encuentran en el objeto. Si entramos en This.Pajaro en la sección de Prototype podemos ver todo lo que es posible realizar tanto a nivel de imagen como de objeto de juego, como cambiar de escala, rotar, establecer una velocidad, etc...

1. [[Prototype]]: Image
   * 1. **constructor**: *ƒ ArcadeImage(scene, x, y, texture, frame)*
     2. **debugBodyColor**: (...)
     3. **debugShowBody**: (...)
     4. **debugShowVelocity**: (...)
     5. **disableBody**: *ƒ (disableGameObject, hideGameObject)*
     6. **enableBody**: *ƒ (reset, x, y, enableGameObject, showGameObject)*
     7. **refreshBody**: *ƒ ()*
     8. **setAcceleration**: *ƒ (x, y):establece una aceleración en el eje x, y + suma valor incrementando el anterior*
     9. **setAccelerationX**: *ƒ (value)*
     10. **setAccelerationY**: *ƒ (value)*
     11. **setAngularAcceleration**: *ƒ (value)*
     12. **setAngularDrag**: *ƒ (value)*
     13. **setAngularVelocity**: *ƒ (value)*
     14. **setBodySize**: *ƒ (width, height, center)*
     15. **setBounce**: *ƒ (x, y): definimos el rebote que tiene que tener el objeto al caer con las físicas del juego*
     16. **setBounceX**: *ƒ (value)*
     17. **setBounceY**: *ƒ (value)*
     18. **setCircle**: *ƒ (radius, offsetX, offsetY)*
     19. **setCollideWorldBounds**: *ƒ (value, bounceX, bounceY)*
     20. **setDamping**: *ƒ (value)*
     21. **setDebug**: *ƒ (showBody, showVelocity, bodyColor)*
     22. **setDebugBodyColor**: *ƒ (value)*
     23. **setDrag**: *ƒ (x, y)*
     24. **setDragX**: *ƒ (value)*
     25. **setDragY**: *ƒ (value)*
     26. **setFriction**: *ƒ (x, y)*
     27. **setFrictionX**: *ƒ (x)*
     28. **setFrictionY**: *ƒ (y)*
     29. **setGravity**: *ƒ (x, y)*
     30. **setGravityX**: *ƒ (x)*
     31. **setGravityY**: *ƒ (y)*
     32. **setImmovable**: *ƒ (value)*
     33. **setMass**: *ƒ (value)*
     34. **setMaxVelocity**: *ƒ (x, y)*
     35. **setOffset**: *ƒ (x, y)*
     36. **setPushable**: *ƒ (value)*
     37. **setSize**: *ƒ (width, height, center)*
     38. **setVelocity**: *ƒ (x, y): establece una velocidad constante*
     39. **setVelocityX**: *ƒ (x)*
     40. **setVelocityY**: *ƒ (y)*
     41. alpha: (...)
     42. alphaBottomLeft: (...)
     43. alphaBottomRight: (...)
     44. alphaTopLeft: (...)
     45. alphaTopRight: (...)
     46. angle: (...)
     47. blendMode: (...)
     48. depth: (...)
     49. displayHeight: (...)
     50. displayOriginX: (...)
     51. displayOriginY: (...)
     52. displayWidth: (...)
     53. isTinted: (...)
     54. rotation: (...)
     55. scale: (...)
     56. scaleX: (...)
     57. scaleY: (...)
     58. visible: (...)
     59. get debugBodyColor: *ƒ ()*
     60. set debugBodyColor: *ƒ (value)*
     61. get debugShowBody: *ƒ ()*
     62. set debugShowBody: *ƒ (value)*
     63. get debugShowVelocity: *ƒ ()*
     64. set debugShowVelocity: *ƒ (value)*
2. [[Prototype]]: GameObject
   1. **alpha**: (...)
   2. **alphaBottomLeft**: (...)
   3. **alphaBottomRight**: (...)
   4. **alphaTopLeft**: (...)
   5. **alphaTopRight**: (...)
   6. **angle**: (...)
   7. **blendMode**: (...)
   8. **clearAlpha**: *ƒ ()*
   9. **clearMask**: *ƒ (destroyMask)*
   10. **clearTint**: *ƒ ()*
   11. **constructor**: *ƒ Image(scene, x, y, texture, frame)*
   12. **copyPosition**: *ƒ (source)*
   13. **createBitmapMask**: *ƒ (renderable)*
   14. **createGeometryMask**: *ƒ (graphics)*
   15. **defaultPipeline**: null
   16. **depth**: (...)
   17. **displayHeight**: (...)
   18. **displayOriginX**: (...)
   19. **displayOriginY**: (...)
   20. **displayWidth**: (...)
   21. **flipX**: false : *realiza un flip de la imagen sobre el eje X ejemplo:*this.pajaro.flipX = true;
   22. **flipY**: false : *realiza un flip de la imagen sobre el eje Y ejemplo:*this.pajaro.flipY = true;
   23. **frame**: null
   24. **getBottomCenter**: *ƒ (output, includeParent)*
   25. **getBottomLeft**: *ƒ (output, includeParent)*
   26. **getBottomRight**: *ƒ (output, includeParent)*
   27. **getBounds**: *ƒ (output)*
   28. **getCenter**: *ƒ (output)*
   29. **getLeftCenter**: *ƒ (output, includeParent)*
   30. **getLocalPoint**: *ƒ (x, y, point, camera)*
   31. **getLocalTransformMatrix**: *ƒ (tempMatrix)*
   32. **getParentRotation**: *ƒ ()*
   33. **getPipelineName**: *ƒ ()*
   34. **getPostPipeline**: *ƒ (pipeline)*
   35. **getRightCenter**: *ƒ (output, includeParent)*
   36. **getTopCenter**: *ƒ (output, includeParent)*
   37. **getTopLeft**: *ƒ (output, includeParent)*
   38. **getTopRight**: *ƒ (output, includeParent)*
   39. **getWorldTransformMatrix**: *ƒ (tempMatrix, parentMatrix)*
   40. **hasPostPipeline**: false
   41. **height**: 0
   42. **initPipeline**: *ƒ (pipeline)*
   43. **isCropped**: false
   44. **isTinted**: (...)
   45. **mask**: null
   46. **originX**: 0.5
   47. **originY**: 0.5
   48. **pipeline**: null
   49. **pipelineData**: null
   50. **postPipelines**: null
   51. **prepareBoundsOutput**: *ƒ (output, includeParent)*
   52. **removePostPipeline**: *ƒ (pipeline)*
   53. **renderCanvas**: *ƒ (renderer, src, camera, parentMatrix)*
   54. **renderWebGL**: *ƒ (renderer, src, camera, parentMatrix)*
   55. **resetCropObject**: *ƒ ()*
   56. **resetFlip**: *ƒ ()*
   57. **resetPipeline**: *ƒ (resetPostPipelines, resetData)*
   58. **resetPostPipeline**: *ƒ (resetData)*
   59. **rotation**: (...)
   60. **scale**: (...)
   61. **scaleX**: (...)
   62. **scaleY**: (...)
   63. **scrollFactorX**: 1
   64. **scrollFactorY**: 1
   65. **setAlpha**: *ƒ (topLeft, topRight, bottomLeft, bottomRight)*
   66. **setAngle**: *ƒ (degrees) : establece la rotación de la imagen en grados ejemplo:*this.pajaro.setAngle(30);
   67. **setBlendMode**: *ƒ (value)*
   68. **setCrop**: *ƒ (x, y, width, height)*
   69. **setDepth**: *ƒ (value)*
   70. **setDisplayOrigin**: *ƒ (x, y)*
   71. **setDisplaySize**: *ƒ (width, height)*
   72. **setFlip**: *ƒ (x, y)*
   73. **setFlipX**: *ƒ (value):*
   74. **setFlipY**: *ƒ (value)*
   75. **setFrame**: *ƒ (frame, updateSize, updateOrigin)*
   76. **setMask**: *ƒ (mask)*
   77. **setOrigin**: *ƒ (x, y): establece el origen del eje X / Y ejemplo:*this.pajaro.setOrigin(0.5, 0.5); // origen en el centro *ejemplo:*this.pajaro.setOrigin(0, 0.5); // origen en la izquierda centrado *ejemplo:*this.pajaro.setOrigin(0, 0); // origen en la izquierda arriba *ejemplo:*this.pajaro.setOrigin(0, 1); // origen en la izquierda abajo
   78. **setOriginFromFrame**: *ƒ ()*
   79. **setPipeline**: *ƒ (pipeline, pipelineData, copyData)*
   80. **setPipelineData**: *ƒ (key, value)*
   81. **setPosition**: *ƒ (x, y, z, w)*
   82. **setPostPipeline**: *ƒ (pipelines, pipelineData, copyData)*
   83. **setRandomPosition**: *ƒ (x, y, width, height)*
   84. **setRotation**: *ƒ (radians): establece la rotación de la imagen en radianes ejemplo:*this.pajaro.Rotation(30);
   85. **setScale**: *ƒ (x, y) :* cambiar la escala (tamaño) del pajaro *ejemplo:*this.pajaro.setScale(2);
   86. **setScrollFactor**: *ƒ (x, y)*
   87. **setSize**: *ƒ (width, height)*
   88. **setSizeToFrame**: *ƒ (frame)*
   89. **setTexture**: *ƒ (key, frame)*
   90. **setTint**: *ƒ (topLeft, topRight, bottomLeft, bottomRight)*
   91. **setTintFill**: *ƒ (topLeft, topRight, bottomLeft, bottomRight)*
   92. **setVisible**: *ƒ (value)*
   93. **setW**: *ƒ (value)*
   94. **setX**: *ƒ (value)*
   95. **setY**: *ƒ (value)*
   96. **setZ**: *ƒ (value)*
   97. **texture**: null
   98. **tintBottomLeft**: 16777215
   99. **tintBottomRight**: 16777215
   100. **tintFill**: false
   101. **tintTopLeft**: 16777215
   102. **tintTopRight**: 16777215
   103. **toggleFlipX**: *ƒ ()*
   104. **toggleFlipY**: *ƒ ()*
   105. **updateDisplayOrigin**: *ƒ ()*
   106. **visible**: (...)
   107. **w**: 0
   108. **width**: 0
   109. **x**: 0
   110. **y**: 0
   111. **z**: 0
   112. **\_alpha**: 1
   113. **\_alphaBL**: 1
   114. **\_alphaBR**: 1
   115. **\_alphaTL**: 1
   116. **\_alphaTR**: 1
   117. **\_blendMode**: 0
   118. **\_depth**: 0
   119. **\_displayOriginX**: 0
   120. **\_displayOriginY**: 0
   121. **\_originComponent**: true
   122. **\_rotation**: 0
   123. **\_scaleX**: 1
   124. **\_scaleY**: 1
   125. **\_sizeComponent**: true
   126. **\_visible**: true
   127. get alpha: *ƒ ()*
   128. set alpha: *ƒ (value)*
   129. get alphaBottomLeft: *ƒ ()*
   130. set alphaBottomLeft: *ƒ (value)*
   131. get alphaBottomRight: *ƒ ()*
   132. set alphaBottomRight: *ƒ (value)*
   133. get alphaTopLeft: *ƒ ()*
   134. set alphaTopLeft: *ƒ (value)*
   135. get alphaTopRight: *ƒ ()*
   136. set alphaTopRight: *ƒ (value)*
   137. get angle: *ƒ ()*
   138. set angle: *ƒ (value)*
   139. get blendMode: *ƒ ()*
   140. set blendMode: *ƒ (value)*
   141. get depth: *ƒ ()*
   142. set depth: *ƒ (value)*
   143. get displayHeight: *ƒ ()*
   144. set displayHeight: *ƒ (value)*
   145. get displayOriginX: *ƒ ()*
   146. set displayOriginX: *ƒ (value)*
   147. get displayOriginY: *ƒ ()*
   148. set displayOriginY: *ƒ (value)*
   149. get displayWidth: *ƒ ()*
   150. set displayWidth: *ƒ (value)*
   151. get isTinted: *ƒ ()*
   152. get rotation: *ƒ ()*
   153. set rotation: *ƒ (value)*
   154. get scale: *ƒ ()*
   155. set scale: *ƒ (value)*
   156. get scaleX: *ƒ ()*
   157. set scaleX: *ƒ (value)*
   158. get scaleY: *ƒ ()*
   159. set scaleY: *ƒ (value)*
   160. set tint: *ƒ (value)*
   161. get visible: *ƒ ()*
   162. set visible: *ƒ (value)*

## Carga de Audios

## Añadir físicas

Para añadir las físicas tendremos que modificar la variable config definida en el fichero init.js

const config = {

    width:320\*2,

    height: 180\*2,

    parent: "container",

    type: Phaser.AUTO,

    scene: {

        preload: preload,

        create: create,

        update: update

    },

    physics: {

        default: "arcade",

        arcade: {

            gravity:{

                y: 500

            }

        }

    }

}

**physics**: indicamos que el juego va tener físicas y la configuración de las mismas

**default**: por defecto es arcade

**arcade**: Json para indicar la configuración de las físicas arcade.

**gravity**: añadimos la gravedad sobre el mundo.

Una vez añadidas las variables de configuración para indicar que el juego va tener físicas podemos hacer uso de ellas en las funciones create() y update().

Dentro de la función create() podemos indicar los siguientes valores:

* Añadir una imagen con físicas:

this.pajaro = this.physics.add.image(50,100,"pajaro");

* Establecer la colisión del objeto pajaro con el mundo (canvas)

this.pajaro.setCollideWorldBounds(true);

* Rebote del objeto física al caer

this.pajaro.setBounce(0.5);

* Establecer aceleración en eje X, Y + suma valor incrementando al anterior

this.pajaro.setAcceleration(10,0);

* Establece una velocidad constante

this.pajaro.setVelocity(50);

* Indicar que la colisión del mundo sea solo arriba, abajo y no los laterales izquierda, derecha

this.physics.world.setBoundsCollision(false,false,true,true);

* Añadir una colisión entre 2 objetos this.ball y la pala this.izquierda. Cuando ambos objetos chocan se ejecuta una funcion this.chocapala

this.physics.add.collider(this.ball, this.izquieda, this.chocaPala, null, this);

chocaPala() {

   this.ball.setVelocityY(Phaser.Math.Between(-120,120));

}

## Control con el teclado

Podemos hacer uso del teclado para ejecutar acciones en nuestro juego. A continuación se indica cómo acceder a los controles del teclado siguiendo con el ejemplo de la imagen cargada del pajaro en Phaser.

### Método en escucha

Una de las opciones es establecer en la función create() un método de escucha , para cuando pulsamos una tecla se genere un evento. En el siguiente ejemplo se indica que el pajaro tenga una aceleración cuando pulsamos sobre la tecla flecha derecha:

function create() {

    this.pajaro = this.physics.add.image(50,100,"pajaro");

    //1.asignar el pulso de teclado para ejecutar funcion

    this.input.keyboard.on("keydown-RIGHT", () => {

        //this.pajaro.x ++;

        this.pajaro.setAcceleration(100,0);

    });

    this.input.keyboard.on("keyup-RIGHT", () => {

        //al levantar el teclado se establece la velocidad a 0 para que deje de moverse

        this.pajaro.setAcceleration(0,0);

        this.pajaro.setVelocity(0);

    });

}

Para poder ver el nombre reservado para las teclas usamos Phaser.Input.Keyboard.KeyCodes

console.log(Phaser.Input.Keyboard.KeyCodes

);

### Asignar cursor predeterminado

Cuando añadimos este método del cursor se añaden automáticamente las flechas arriba, abajo, izquierda, derecha y espacio.

En la funcion create() añadimos lo siguiente y definimos la variable cursor:

function create() {

    this.pajaro = this.physics.add.image(50,100,"pajaro");

    //2. asignar pulso de teclas por cursor flechas

    this.cursor = this.input.keyboard.createCursorKeys();

}

En la funcion update podemos hacer uso de esta variable para mover el pajaro en el sentido que queramos.

function update(time, delta) {

    if (this.cursor.right.isDown)

    {

        this.pajaro.x++;

    }else if (this.cursor.left.isDown)

    {

        this.pajaro.x--;

    }

}

### Crear cursores personalizados

También podemos crear cursores predeterminados. En el siguiente ejemplo podemos indicar en una variable derecha que sea la tecla D

this.right = this.input.keyboard.addKey(Phaser.input.keyboard.KeyCodes.D);

También podemos indicar en número reservado para la tecla D en este caso a la hora de usar addKe().

Para poder ver el nombre reservado para las teclas usamos Phaser.Input.Keyboard.KeyCodes

console.log(Phaser.Input.Keyboard.KeyCodes

);

## Escenas

Una escena en Phaser es un contenedor de un mundo independiente, podemos crear distintas pantallas sin necesidad de chocar unas con otras.

La escenas de guardan en el manejador de escenas (escenas []), este gestiona el orden en el que las escenas son renderizadas y el orden en que las escenas son actualizadas.

El primer paso es crear los ficheros js que hagan referencia a las escenas. Estos ficheros serán las clases que heredaran las propiedades de Phaser.Scene. En este caso se van a llamar sceneA.js, sceneB.js y sceneC.js. Los linkamos en el fichero índex a través de la etiqueta script como se ve en la imagen siguiente:

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta http-equiv="X-UA-Compatible" content="IE=edge">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

    <div id="container"></div>

    <script src="./node\_modules/phaser/dist/phaser.js"></script>

    <script src="./src/sceneC.js"></script>

    <script src="./src/sceneB.js"></script>

    <script src="./src/sceneA.js"></script>

    <script src="./src/init.js"></script>

</body>

</html>

Dentro de la clase sceneA tendremos un constructor de la clase y las funciones preload(), create() y update()

class SceneA extends Phaser.Scene{

    constructor(){

        super({

            key:"SceneA"

        });

    }

    preload(){

    }

    create(){

        let graphics = this.add.graphics();

        graphics.fillStyle(0xff3300, 1);

        graphics.fillRect(100,200,600,300);

        graphics.fillRect(100,100,100,100);

        this.add.text(120,110,"A",{font: "96px Courier", fill:"#000000"});

        //instanciar una escena dentro de otra

        this.scene.add("SceneC", new SceneC);

    }

    update(){

    }

}

Cuando entre en constructor de sceneA se cargara super para que se cargue el constructor de Phaser.Scene. Dentro de super, en la clave key indicaremos el nombre de la SceneA interno. La primera escena que se carga siempre es la que está activa, si queremos visualizar otras escenas tendremos que indicar en super la clave active: true

    constructor(){

        super({

            key:"SceneB",

            active: true

        });

    }

Para instancia la escena lo tenemos que indicar en el fichero init.js a través de la clave scene

const config = {

    width:800,

    height: 600,

    parent: "container",

    type: Phaser.AUTO,

    backgroundColor: '#392542',

    scene: [SceneA, SceneB]

}

var game = new Phaser.Game(config);

También podemos instanciar una escena dentro de otra como se ve en el siguiente código añadido en el create() de la sceneA

        //instanciar una escena dentro de otra

        this.scene.add("SceneC", new SceneC);

para inicializar una escena y que la otra desaparezca, por ejemplo queremos pulsar un botón en la escena A que nos lleve a la escena C, en este caso tenemos que indicar que la escena C no es activa (en el constructor de la escena) y le indicamos que arranque la escena con scene.start() . Esto hará que la escena A desaparezca y se muestre en su lugar la escena C

this.scene.start("SceneC");

Podemos indicarle a una escena que se mueva al frente, al fondo o que se posicione en un lugar concreto entre escenas con el siguiente código

        this.scene.sendToBack(); // mandar atrás

        this.scene.bringToTop(); // traer al frente

        this.scene.moveUp(); //un paso arriba

        this.scene.moveDown(); // un paso atrás

        this.scene.moveAbove(); //arriba de una escena

        this.scene.moveBelow(); //abajo de una escena

## Módulos

A medida que nuestra aplicación crece, queremos dividirla en múltiples archivos, llamados “módulos”. Un módulo puede contener una clase o una biblioteca de funciones para un propósito específico.

Durante mucho tiempo, JavaScript existió sin una sintaxis de módulo a nivel de lenguaje. Eso no fue un problema, porque inicialmente los scripts eran pequeños y simples, por lo que no era necesario.

Pero con el tiempo los scripts se volvieron cada vez más complejos, por lo que la comunidad inventó una variedad de formas de organizar el código en módulos, bibliotecas especiales para cargar módulos a pedido.

Un módulo es solo un archivo. Un script es un módulo. Tan sencillo como eso.

Los módulos pueden cargarse entre sí y usar directivas especiales export e import para intercambiar funcionalidad, llamar a funciones de un módulo de otro:

* La palabra clave export etiqueta las variables y funciones que deberían ser accesibles desde fuera del módulo actual.
* import permite importar funcionalidades desde otros módulos.

Tomando el ejemplo del anterior punto escenas vemos que estas se cargaban haciendo referencia en index.html a los distintos ficheros JS. Este método lo vamos a eliminar e indicaremos que el fichero init.js va ser del tipo modulo como se ve en el siguiente código:

<body>

    <div id="container"></div>

    <script src="./node\_modules/phaser/dist/phaser.js"></script>

    <!-- <script src="./src/sceneC.js"></script>

    <script src="./src/sceneB.js"></script>

    <script src="./src/sceneA.js"></script> -->

    <script src="./src/init.js" type="module"></script>

</body>

Para convertir una clase en un módulo tenemos que darle la propiedad de que sea exportable, para ello bastan con indicar en la clase que sea de tipo export default

export default class SceneA extends Phaser.Scene {}

Otra forma de exportar la clase es indicando al final de la clase el tipo export default con el nombre de la escena a la que hacemos referencia

export default SceneB;

Una vez que las clases ya son exportable podemos importarlas desde las clases o ficheros que vamos a hacer referencia. En el fichero init.js hacemos uso de la ScenaA y SceneB por lo que es necesario realizar un import de ambas escenas indicando su ubicación en el proyecto:

import SceneA from "./sceneA.js";

import SceneB from "./sceneB.js";

const config = {

    width:800,

    height: 600,

    parent: "container",

    type: Phaser.AUTO,

    backgroundColor: '#392542',

    scene: [SceneA, SceneB]

}

var game = new Phaser.Game(config);

## Animación

### Sprite

 El sprite es una imagen que se representa por distintos frames para crear la ilusión de la animación.

Phaser separa las imágenes de un sprite indicándoles los puntos de coordenada que tiene cada frame para realizar la animación.

Para cargar el sprite en la funcion preload() llamamos a load.spriteseet() indicando como parámetros un nombre interno, la ruta donde se encuentra nuestro sprite y un parámetro Json indicando el alto y ancho del frame

function preload(){

this.load.spritesheet('evil\_tomato', './assets/evil\_tomato.png', { frameWidth:16, frameHeight:25});

}

Una vez cargado el sprite tenemos que añadirlo en la funcion create()

this.tomato = this.add.sprite (100, 100, 'evil\_tomato', 0); // añadimos el sprite x, y , nombre del sprite y posicion del frame

Para crear la animación también lo definimos dentro de la funcion create()

 this.anims.create({

        key: 'tomato\_walk', // se definie el nombre interno de la animacion

        frames: this.anims.generateFrameNumbers('evil\_tomato', {

            //frames: [1,2,3,4,5,6,7,8] // en el json puedo cargar los frames de 2 formas

            start: 1,

            end: 8

        }),

        repeat: -1, // indica la repeticion de la animacion. con -1 crea un bucle infinito

        frameRate: 10 // la velocidad en la que se ejecuta cada frame , por defecto es 24

    });

    // le decimos a la animacion que arranque con play

    this.tomato.anims.play('tomato\_walk')

### Atlas

