The Pragmatic Bookshelf

PragPub

The Second Iteration

IN THIS ISSUE

* Making a 2D Game in Phaser

Issue #84
June 2016

Contents

Published monthly in PDF, mobi, and epub formats by The Pragmatic Programmers, LLC, Dallas, TX, and Raleigh, NC. E-Mail webmaster@swaine.com. The editor is Michael Swaine (michael@pragprog.com). Visit us at http://pragprog.com for the lowdown on our books, screencasts, training, forums, and more.

ISSN: 1948-3562

Making a 2D Game in Phaser

Rapid Game Development for Web and Mobile

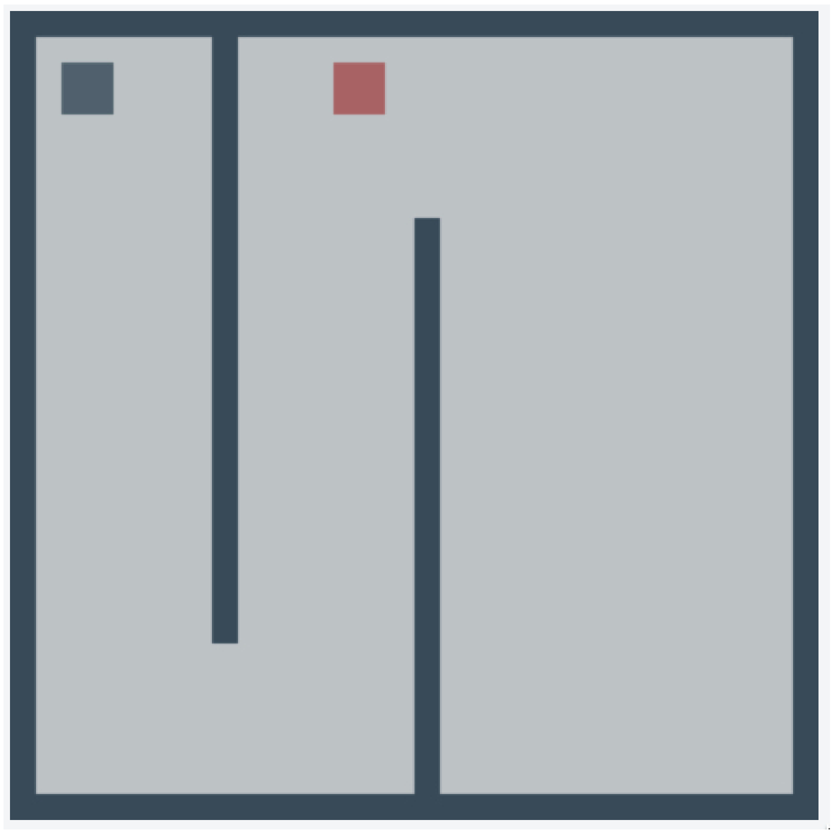
by Brian Hogan

Let's have some fun — and in the process learn about a rich JavaScript framework for creating games that run in your browser or on mobile devices.

Creating your own game is a great way to blend creativity with programming, while having fun at the same time. But dealing with states, collisions, sprites, and more can be a daunting task if you've never had any experience with those elements.

Phaser[U1] is a JavaScript framework for creating games that run in your browser or on mobile devices. Using Phaser, you can quickly create 2D games like space shooters, platformers, puzzle games, and more. All you need is the Phaser library, some basic JavaScript knowledge, and your imagination. Let's explore Phaser by creating a simple game that explores the basics.

We'll create a simple top-down game with a player, a few walls, and an enemy. When you're done, you'll have a simple game that looks like this:



The Completed Game

This will give you a good starting point to build upon, and you'll find some suggestions at the end for ways to expand on this idea.

The Skeleton

Phaser is a game library that leverages browser technologies, so we'll need to load it into a web page. To begin, create two files; one file called index.html and another file called index.js. In the index.html file, define a basic HTML skeleton like this:

```
html
<!DOCTYPE html>
<html lang="en-US">
  <head>
    <meta charset="utf-8">
    <title>Game</title>
  </head>
  <body>
    <script
      src=
      "https://cdnjs.cloudflare.com/ajax/libs/phaser/2.4.8/phaser.min.js">
    </script>
    <script src="index.js"></script>
  </body>
</html>
```

We're loading Phaser from a CDN, and then loading our index.js file, which will contain the code for our game. We load Phaser first so that our code can use the many helpers Phaser includes.

In the index.js file, declare a game variable that defines the game world:

```
var game = new Phaser.Game(500, 500);
```

This creates a new blank game stage with a black background that's 500 pixels wide and 500 pixels high. All of the elements of our game will be inside of this box.

Save the HTML file and the JavaScript file, and then open the index.html file in your browser to verify that you see the game stage:

The Blank Game Stage

It’s not much to look at yet, but we’ve got a lot left to do.

Game States

A Phaser game needs a create() function and an update() function. The create() function defines the entities of the game, and the update() function runs 60 times per second. It’s where we check for keyboard input, collisions between entities in the game, and other gameplay logic.

We can define these functions and then associate them with our game object, or we can create a “state” object that holds these functions and add that state to the game. We can then have a title screen state, a main state, and a game over state. Or even define a state for each level of the game.

Let’s define a main state. First, create a mainState object literal that contains the create() function and the update() function. We’ll fill in the bodies of these functions as we go forward. Add this code to index.js:

```
var mainState = {
  create: function() {
  },
  update: function() {
  }
};
```

Once you define the state, you have to add the state to the game. Add these lines to the index.js file at the very bottom:

```
game.state.add('main', mainState);
game.state.start('main');
```

The first line adds the state and gives it a name called main. The second tells Phaser to start that state. This is the line that tells Phaser to start your game.

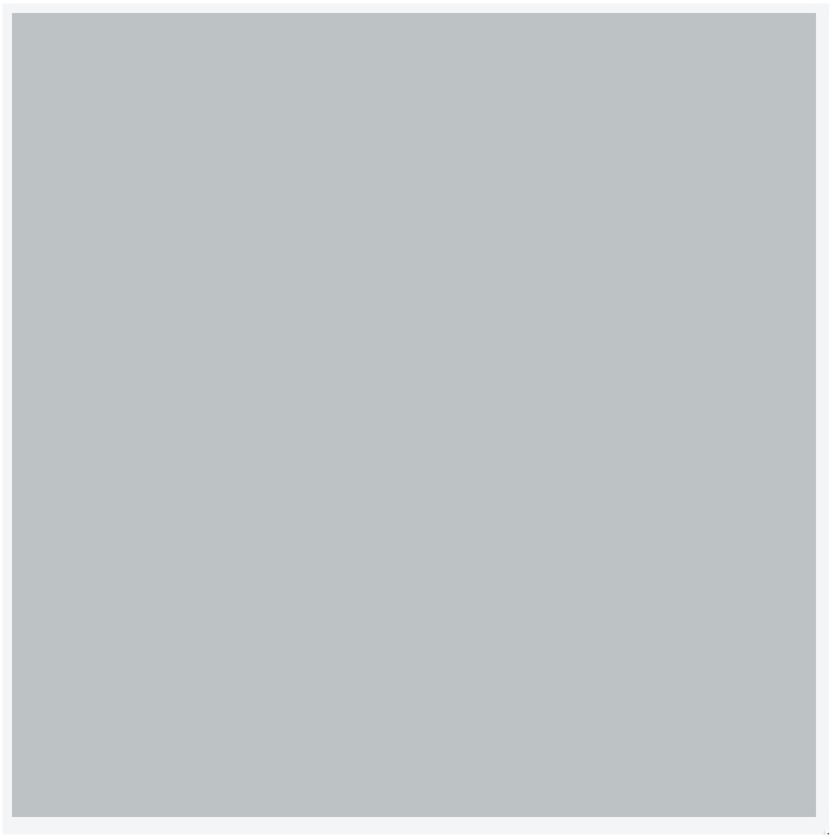
Defining Basic Properties

We define the basic properties of our game inside the create() function of our mainState object. For example, we can set a background color and we can set up a physics system so we can create game objects that can move and interact with each other. Add this to the create() function:

```
game.stage.backgroundColor = '#BDC2C5';
game.physics.startSystem(Phaser.Physics.ARCADE);
game.world.enableBody = true;
```

This sets up Phaser's Arcade physics engine, which is simple but effective for arcade-style games. The game.world.enableBody line applies physics to every item we create in our game.

Refresh your page in the browser and you'll see the background color has changed:



The New Background Color

Now that we've defined the game world, we can start adding objects.

Creating A Player

To create the player, we need to define a sprite, and to do that, we need image data. We can load a file from our hard drive or a URL, but that means we have to either make our own sprite with an image editing program, or find some graphics we can use.

We can also use Phaser to create sprites by defining bitmap data and loading it into a sprite object. This is a great technique for prototyping a game.

Let’s define a function that creates a box. We can pass the dimensions of the box and the color as arguments when we call the function. That way we can use this to create the player, the walls, and the enemy. At the top of the index.js file, add this code:

```
var box = function(options) {
  var bmd = game.add.bitmapData(options.length,options.width);
  bmd.ctx.beginPath();
  bmd.ctx.rect(0,0,options.length,options.width);
  bmd.ctx.fillStyle = options.color;
  bmd.ctx.fill();
  return bmd;
};
```

If you’ve ever worked with the Canvas API, this should look familiar. We create a rectangle by defining the starting coordinates and the dimensions, and then we fill it with a color.

We can use this function to define a colored rectangle like this:

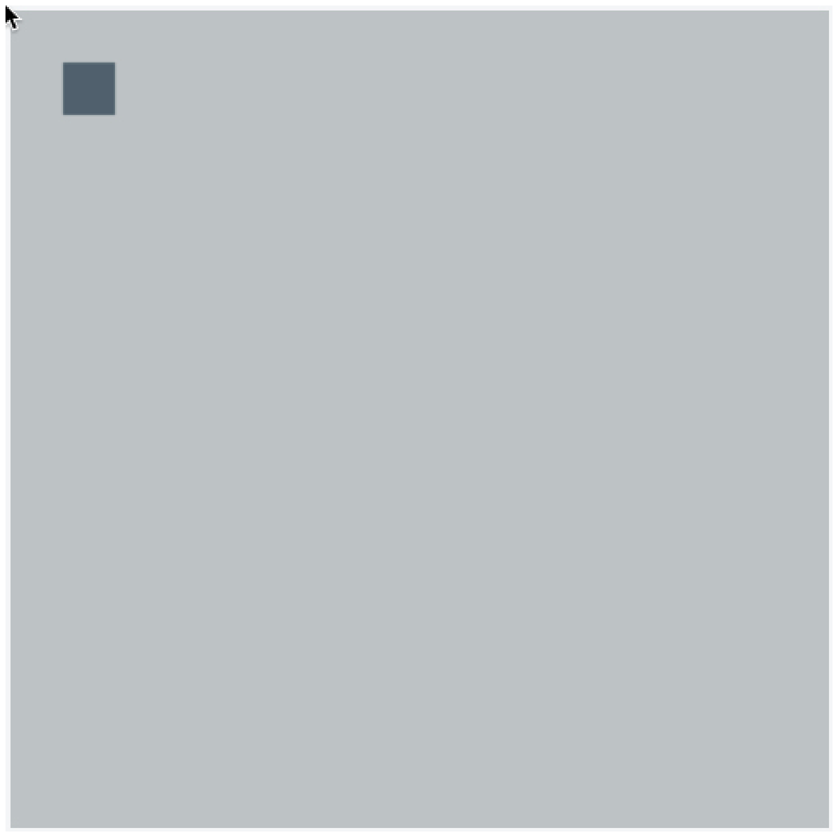
```
shape = box({
  length: 32,
  width: 32,
  color: '#FF0000'
});
```

This will return us the image data we need when we want to create a sprite. Let’s use this to create our player, which will be a small square. In the create() function of the mainState object, define a player sprite:

```
this.player = game.add.sprite(32, 32,
  box({length: 32, width: 32, color: '#4F616E'})
);
```

This places the sprite 32 pixels to the right and 32 pixels down from the top. The Phaser game uses a coordinate system just like the one used for CSS positioning or the HTML5 canvas; the upper left corner of the game world is (0,0).

Refresh the screen and you’ll see your player object:



The Player

But a player that doesn't do anything isn't very interesting.

Make the Player Move

We want to move the player when we press the arrow keys. Phaser has some nice built-in support for that. In the create() function of the mainState object, add this line to define a cursor object:

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

We can use this to detect which arrow key was pressed and react to it.

We check for movement in the update() function of the mainState object. Remember, update() runs 60 times per second. This is our game loop, which runs until we stop the game. So any animation, state changes, or game events are all defined in here.

To make things consistent, declare a variable for the speed of movement and then set the velocity of the player to zero. This way the player won't move unless a key is pressed.

```
var speed = 250;
this.player.body.velocity.y = 0;
this.player.body.velocity.x = 0;
```

Then use if statements to check to see which cursor key was pressed and assign the velocity to the player:

```
if (this.cursor.up.isDown) {
  this.player.body.velocity.y -= speed;
} else if (this.cursor.down.isDown) {
  this.player.body.velocity.y += speed;
}
if (this.cursor.left.isDown) {
  this.player.body.velocity.x -= speed;
} else if (this.cursor.right.isDown) {
  this.player.body.velocity.x += speed;
}
```

Now refresh the index.html page in the browser and press the arrow keys to move the box around. Play with the speed variable and see how it affects the movement of your character. Later on you might consider creating a powerup that increases the speed of the player, or even something that slows it down.

One thing you'll notice is that your player can leave the game world. But add this line to the create() function:

```
this.player.body.collideWorldBounds = true;
```

and now the player is restricted to the boundaries of the game world. Phaser handles the collision automatically. Refresh your browser to test it out.

Creating Walls

Let's create some walls in our game. We'll define walls on all four sides of the game world, and then create a couple of inner walls. We can use our box() function to draw the walls.

In create(), add this code to define a wall across the top of the game world:

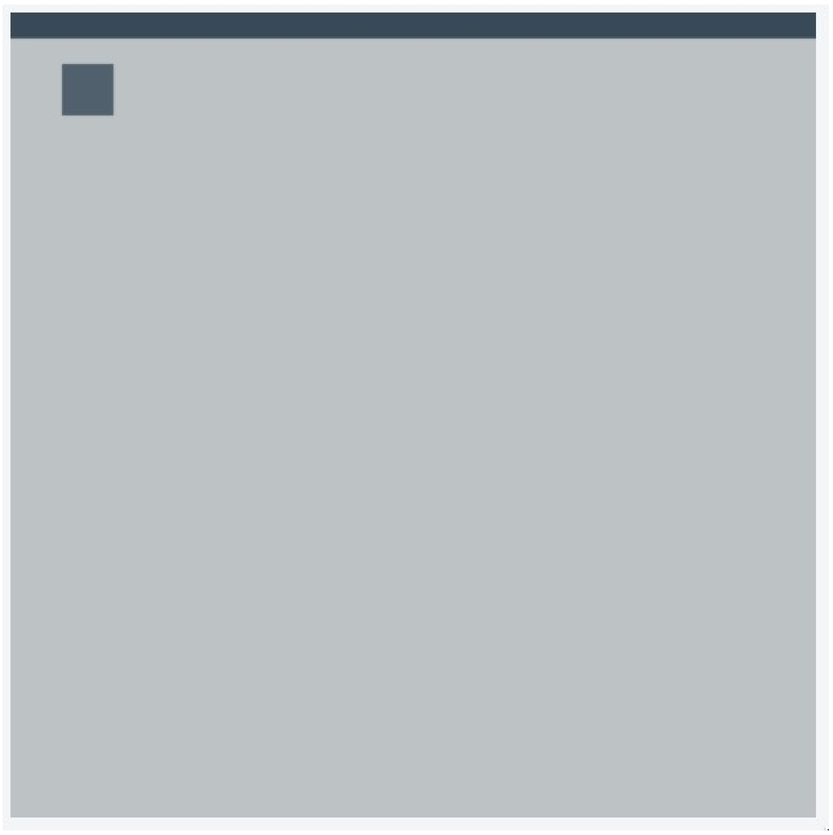
```
this.walls = game.add.group();
this.walls.enableBody = true;
var top = this.walls.create(0, 0,
  box({
    length: game.world.width,
    width: 16,
    color: '#374A59'
  })
);
top.body.immovable = true;
```

First, we create a group of sprites. This lets us apply properties to a collection of sprites. For example, we can enable physics on the entire group instead of on each individual sprite.

Then we create a top sprite, but we do so by adding it to the walls group. We position it at 0,0 and then use the box function to define a wall that's as long as the game world, and 16 pixels wide.

Finally, we set its body to immovable so that other entities bounce off of it. If we left this off, then anything that collides with the wall would move the wall. Of course, this can be great for block-breaking games or even secret passageways.

The result looks like this:


Top Wall

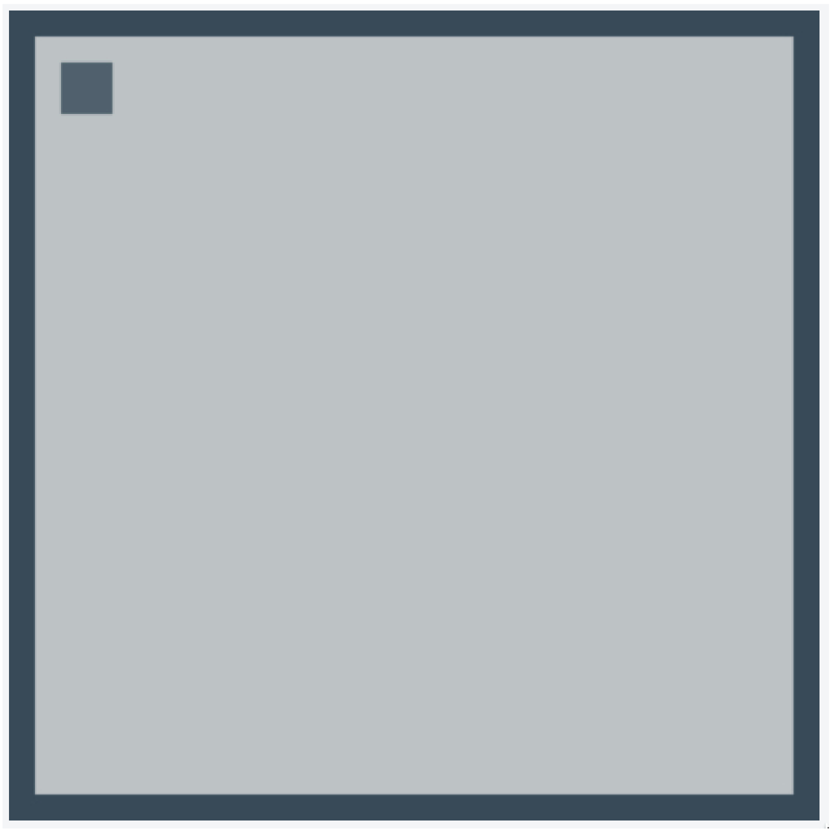
Creating the bottom wall works the same way, but we just change the starting position. Add this code to create(), right below the top wall code:

```
var bottom = this.walls.create(0, game.world.height - 16,
  box({
    length: game.world.width,
    width: 16,
    color: '#374A59'
  })
);
bottom.body.immovable = true;
```

You define the left and right walls the same way. Both walls need to start 16 pixels from the top of the world so they don’t overlap the top wall. And they need to be 32 pixels shorter than the two top walls so they don’t overlap the bottom wall. Remember, the bottom wall starts 16 pixels from the bottom of the game world, so we have to account for the 16 pixels at the top and the 16 pixels at the bottom. Add this code to define the left and right walls:

```
var leftWall = this.walls.create(0, 16,
  box({
    length: 16,
    width: game.world.height -32,
    color: '#374A59'
  })
);
leftWall.body.immovable = true;
var rightWall = this.walls.create(game.world.width - 16, 16,
  box({
    length: 16,
    width: game.world.height -32,
    color: '#374A59'
  })
);
rightWall.body.immovable = true;
```

Now we have walls all around the game world:



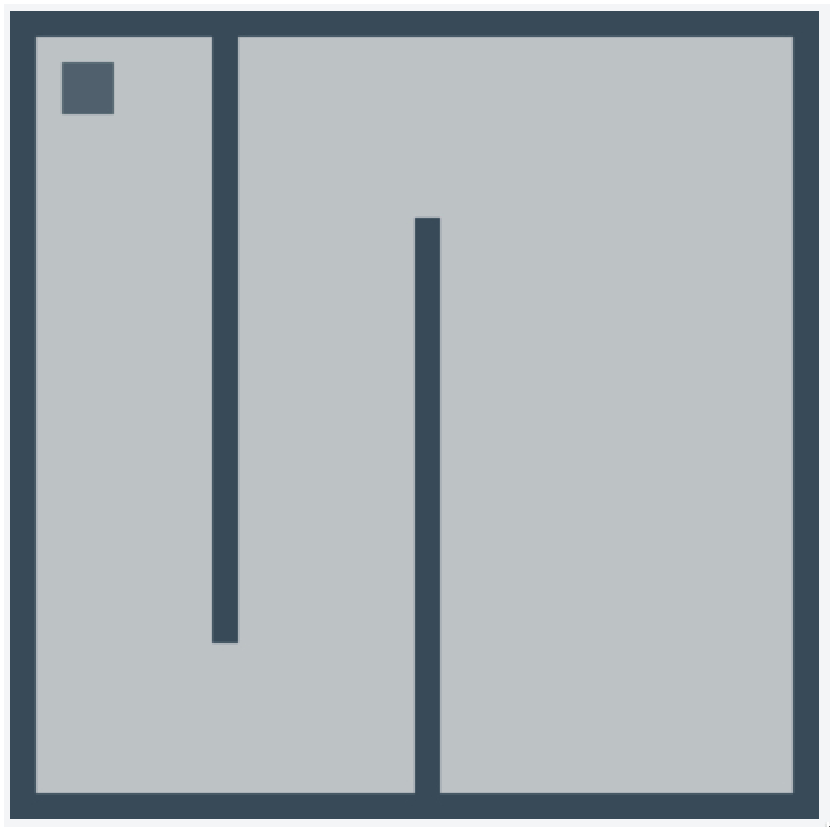
All Outer Walls

Finally, let's add two interior walls to the game. I'll let you decide how wide they are and where you place them. Remember that you can use game.world.height and game.world.width as reference points. To find the middle of your game world, take the world's width and divide by two to get the X coordinate, and then take the world's height divided by two to get the Y coordinate.

It might help to get out some graph paper and sketch out where you want the walls to go.

I'll add two walls in like this:

```
var innerWall1 = this.walls.create(game.world.width / 4, 16,
  box({
    length: 16,
    width: game.world.height - game.world.height / 4,
    color: '#374A59'
  })
);
innerWall1.body.immovable = true;
var innerWall2 = this.walls.create(game.world.width / 2, 128,
  box({
    length: 16,
    width: game.world.height - game.world.height / 4,
    color: '#374A59'
  })
);
innerWall2.body.immovable = true;
```



Inner Walls

Unfortunately, our player can walk through walls. This isn't ideal.

Handling Collisions

A "collision" is when a sprite touches, or runs into, another sprite. A lot of 2D game programming involves dealing with collisions between objects. When PacMan touches a dot, the dot disappears and the score increases. When PacMan touches a ghost, PacMan dies and loses a life. When PacMan hits a wall, he stops.

You might be thinking that you'll have to write a bunch of if statements to determine if things are touching. But Phaser's Arcade physics system handles that for you.

Add this to the top of the update() function to ensure your player can't phase-walk through walls:

```
game.physics.arcade.collide(this.player, this.walls);
```

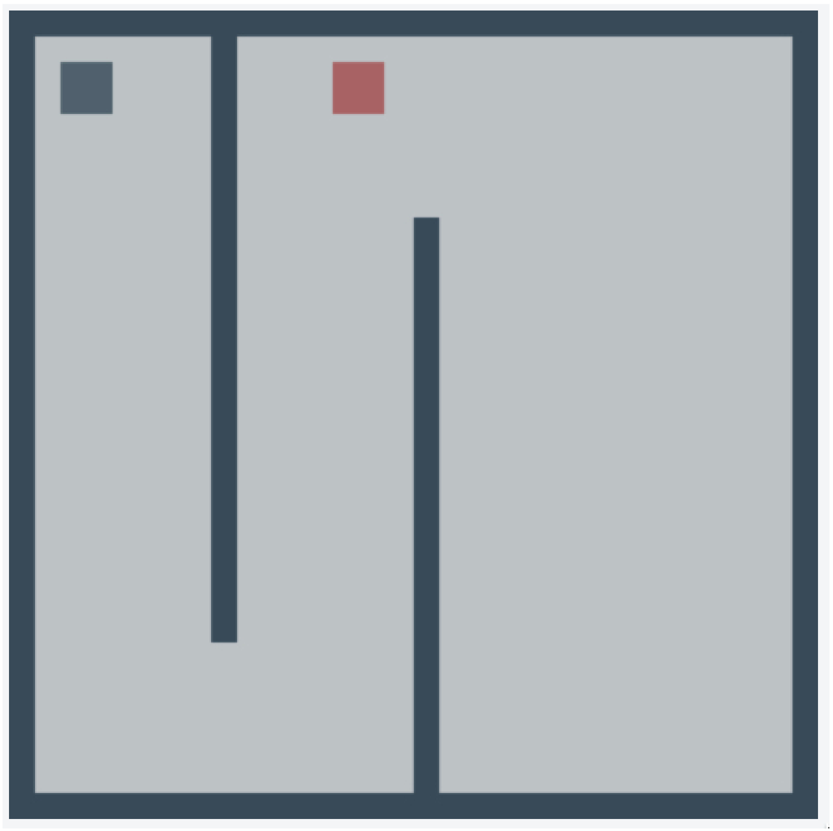
That's all you have to do. The player now can't move through walls. Phaser handles the collision automatically, just like it did with the collision with the world boundaries.

collide is great for things like walls, but if we want to handle interactions with objects or enemies, we need to take a slightly different approach.

Let's add an enemy to the game. In create(), add this code:

```
this.enemy = game.add.sprite(200, 32,
  box({
    length: 32,
    width: 32,
    color: '#A96262'
  })
);
```

This places a red box on the screen, representing our enemy. Beware the fierce red box.



Our Enemy

When the player touches the enemy, we want the player to be killed. Phaser handles this quite nicely. In update(), add this code:

```
game.physics.arcade.overlap(this.player, this.enemy,
        this.handlePlayerDeath, null, this);
```

This tells Phaser that when the player overlaps with the enemy, the function handlePlayerDeath should be called. The handlePlayerDeath function will receive both the player object and the enemy object.

So, in the mainState object, define a new function. Place it below the update() function, and make sure to add a comma after the update() function's closing curly brace.

```
// end of the update function - don't forget to add the comma!
},
handlePlayerDeath: function(player, enemy) {
  player.kill();
}
```

The handlePlayerDeath() function receives the player and the enemy, and Phaser provides a kill() function for a sprite that removes it from the game. So making a sprite disappear is a piece of cake. Just define an overlap and then define a function to handle the overlap.

This is how you handle a bullet hitting an enemy, or a player picking up an item.

Game Over

When we touch the enemy, the game just stops. Let's tidy things up by making a Game Over screen. We do this by defining a new state with its own create() and update() functions. When the game is over, we'll display this new state.

```
gameOverState = {
  create: function() {
  },
  update: function() {
  }
};
```

In this new create() function, define a text label that displays "Game Over" in the center of the screen. To define a label, you add text to the game, position it, and define how it looks:

```
label = game.add.text(game.world.width / 2 , game.world.height / 2,
  'GAME OVER\nPress SPACE to restart',
  {
    font: '22px Arial',
    fill: '#fff',
    align: 'center'
  }
);
label.anchor.setTo(0.5, 0.5);
```

This places the text right in the middle of the screen. Just like sprites, we can add text to the game and position it wherever we like. And that text can be updated in an update() function, so you can use labels to display the current score or other important information.

We tell people to press the space bar to play again, so define a keymapping for the spacebar in the create() function of the gameOverState object:

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

Then in the update() function of gameOverState, handle the spacebar event:

```
if (this.spacebar.isDown) {
  game.state.start('main');
}
```

When the spacebar is pressed, we start the main state again.

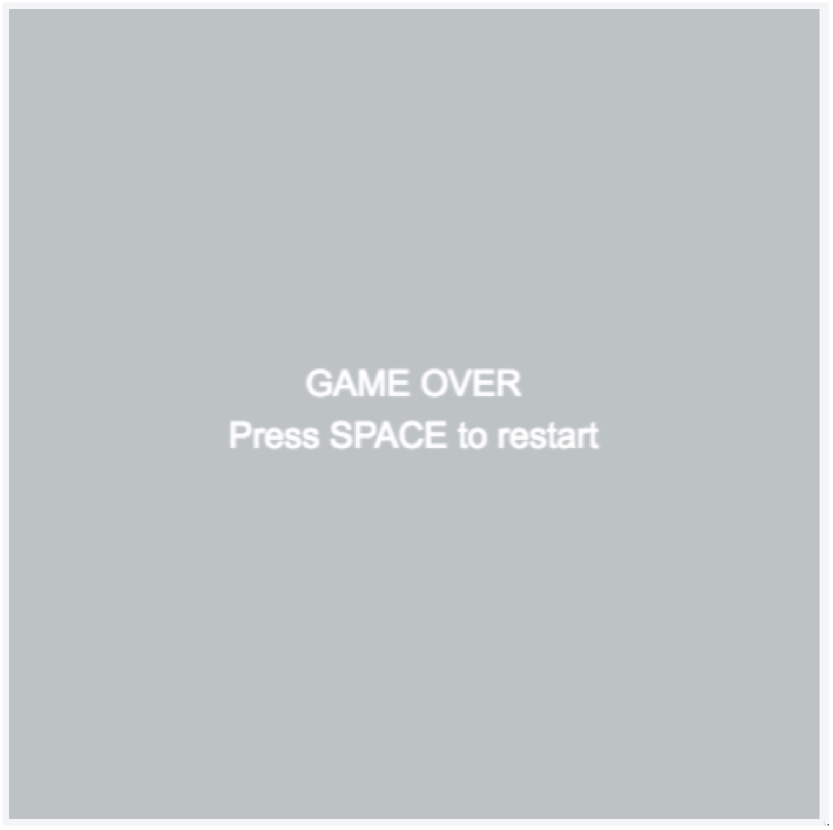
For this new state to be recognized, you must add this new state to the list of states. At the bottom of the file, find the line where you added the main state, and add this right below it:

```
game.state.add('gameOver', gameOverState);
```

Finally, change the handlePlayerDeath() function in the mainState object so that it starts the gameOverState state when the player dies:

```
handlePlayerDeath: function(player, enemy) {
  player.kill();
  game.state.start("gameOver");
}
```

Refresh your browser and play the game again. Now when you collide with the enemy, the gameOverState is started and you see the Game Over message.



```
GAME OVER
Press SPACE to restart
```

Game Over Screen

If you press the space bar, the main state is started again, and the game restarts!

You can use this strategy to create a title screen for your game, or use a state for each “stage” of the game.

Where To Go Next

The Phaser documentation is amazing, and the Phaser website contains over 200 examples that demonstrate touch input, mouse input, particle effects, and much more. Use the Phaser resources and what you learned here to make the following changes to this game:

1. Make the enemy move slowly up and down so your player has to avoid it.
2. Instead of a single enemy, make a group of enemies.
3. See if you can get the enemies to move at random speeds.
4. Spawn enemies in different spots. But make sure they don't spawn on a wall!
5. Create an exit. When the player touches the exit, display a message stating that they won.
6. Put the exit behind a locked door. Create a key in the level. When the player touches the key, remove the key and track that the player has the key. Then when the player touches the door, remove the door, and allow the player to move through it.
7. Replace our primitive boxes with real textures. We created sprites using bitmap data, but most game levels are created with Tilesets, and the Tiled [U2] editor is a great tool for designing tilesets. You can find tutorials on using Phaser and Tiled on the Phaser [U3] website.

Once you've mastered the basics, explore the other types of games you can make with Phaser, and be sure to share them. And share out links to your creations so the world can see them!

Appendix: Full Game Code

```
var game = new Phaser.Game(500, 500);
var box = function(options) {
  var bmd = game.add.bitmapData(options.length,options.width);
  // draw to the canvas context like normal
  bmd.ctx.beginPath();
  bmd.ctx.rect(0,0,options.length,options.width);
  bmd.ctx.fillStyle = options.color;
  bmd.ctx.fill();
  return bmd;
};
```

```
// Create the state that will contain the whole game
var mainState = {
  create: function() {
    // Set the background color to blue
    game.stage.backgroundColor = '#BDC2C5';
    // Start the Arcade physics system (for movements and collisions)
    game.physics.startSystem(Phaser.Physics.ARCADE);
    // Add the physics engine to all game objects
    game.world.enableBody = true;
    // use the bitmap data as the texture for the sprite
    this.player = game.add.sprite(32, 32,
      box({
        length: 32,
        width: 32,
        color: '#4F616E'
      })
    );
    this.cursor = game.input.keyboard.createCursorKeys();
    this.player.body.collideWorldBounds = true;
    this.walls = game.add.group();
    this.walls.enableBody = true;
    var top = this.walls.create(0, 0,
      box({
        length: game.world.width,
        width: 16,
        color: '#374A59'
      })
    );
    top.body.immovable = true;
    var bottom = this.walls.create(0, game.world.height - 16,
      box({
        length: game.world.width,
        width: 16,
        color: '#374A59'
      })
    );
    bottom.body.immovable = true;
    var leftWall = this.walls.create(0, 16,
      box({
        length: 16,
        width: game.world.height -32,
        color: '#374A59'
      })
    );
    leftWall.body.immovable = true;
    var rightWall = this.walls.create(game.world.width - 16, 16,
      box({
        length: 16,
        width: game.world.height -32,
        color: '#374A59'
      })
    );
    rightWall.body.immovable = true;
    var innerWall1 = this.walls.create(game.world.width / 4, 16,
      box({
        length: 16,
        width: game.world.height - game.world.height / 4,
        color: '#374A59'
      })
    );
    innerWall1.body.immovable = true;
    var innerWall2 = this.walls.create(game.world.width / 2, 128,
      box({
        length: 16,
        width: game.world.height - game.world.height / 4,
        color: '#374A59'
      })
    );
    innerWall2.body.immovable = true;
    this.enemy = game.add.sprite(200, 32,
      box({
        length: 32,
        width: 32,
        color: '#A96262'
      })
    );
  },
  update: function() {
    game.physics.arcade.overlap(this.player, this.enemy,
        this.handlePlayerDeath, null, this);
    game.physics.arcade.collide(this.player, this.walls);
    var speed = 250;
    //player movement
    this.player.body.velocity.y = 0;
    this.player.body.velocity.x = 0;
    if (this.cursor.up.isDown) {
      this.player.body.velocity.y -= speed;
    } else if (this.cursor.down.isDown) {
      this.player.body.velocity.y += speed;
    }
    if (this.cursor.left.isDown) {
      this.player.body.velocity.x -= speed;
    } else if (this.cursor.right.isDown) {
      this.player.body.velocity.x += speed;
    }
  },
  handlePlayerDeath: function(player, enemy) {
    player.kill();
    game.state.start('gameOver');
  }
};

var gameOverState = {
  create: function() {
    this.spacebar = this.game.input.keyboard.addKey(Phaser.Keyboard.SPACEBAR);
    label = game.add.text(game.world.width / 2 , game.world.height / 2,
      'GAME OVER\nPress SPACE to restart',
      {
        font: '22px Arial',
        fill: '#fff',
        align: 'center'
      }
    );
    label.anchor.setTo(0.5, 0.5);
  },
  update: function() {
    if (this.spacebar.isDown) {
      game.state.start('main');
    }
  }
};
game.state.add('main', mainState);
game.state.add('gameOver', gameOverState);
game.state.start('main');
```

About the Author

Brian P. Hogan is a developer, author, and teacher who loves building things for the web. He teaches introductory programming classes at the college level and has an interest in performance-based learning. He is the author of Automate with Grunt [U4], tmux [U5], and HTML5 and CSS3 [U6], and is the co-author of Web Development Recipes [U7].

External resources referenced in this article:

[U1]	http://phaser.io/
[U2]	http://www.mapeditor.org/
[U3]	http://phaser.io/
[U4]	https://pragprog.com/book/bhgrunt/
[U5]	https://pragprog.com/book/bhtmux/
[U6]	https://pragprog.com/book/bhh52e/
[U7]	https://pragprog.com/book/wbdev2/