Gamedev Phaser Content Kit

Learn how to build HTML5 Games with Phaser

Welcome to the **Gamedev Phaser Content Kit**, also known as **Mobile Web Games Starter Kit** or **HTML5 Game Development with Phaser**. This **Content Kit** contains resources on how to start building games using the **Phaser** framework, it is prepared especially for beginners. You can use it to learn it yourself or teach others either by giving a conference talk or during the hands-on workshop.

Content Kit materials

- Step-by-step tutorial split into 16 lessons
- Demos with the source code available for every step
- Slide decks with the presenter notes included

This Content Kit can be used as a base for the conference talk material, workshop, online training or whatever else comes to your mind. You can use the slides containing the commented, explained snippets of code to present the topic to the audience, or you can run a live-coding session showing people how to build the final version of the game from scratch. You can also add your own content or trim the existing one if you have limited time for the talk.

Current version and licensing

Prerequisites for the presenter

- Computer running Windows, Linux or Mac OS X.
- Modern browser: Firefox, Chrome, Opera, Safari, or Internet Explorer 10+.
- Text editor: Sublime Text, Notepad++.
- Intermediate JavaScript/Canvas knowledge.

Prerequisites for the audience

- Optional setup described above to test demos or take part in the workshop.
- Basic JavaScript knowledge.

Key points

Here's what the audience will be able to do or get from your talk/workshop:

- Know how to start making games using Phaser.
- Be able to write JavaScript code properly.

- Know how the Phaser-specific functions work and how to use them effectively.
- Understand the basic 2D game logic.

Presentation setup

There are two sets of slides that can be used independently, or together one after another if you have enough time. The first, optional "theory" slide deck walks you through the HTML5 game development in general, its tools and resources. The second, main "talk" slide deck is using the Breakout HTML5 game as an example and explains the key elements needed to have such game implemented.

- Optional "theory" slide deck
- Main "talk" slide deck

Presenting about HTML5 game development is fairly simple — you just need the slides and demo materials, downloaded locally if possible so network connectivity is not a problem. Just running the "theory" presentation without a code walkthrough or workshop should take 30 minutes, and the "talk" presentation should also take about 30 minutes. You can present those two in this particular order if you have an hour for your talk. If your time is limited to a half an hour, you can go with the main slide deck only.

You can visit the slides subpage where all the decks are available - those two for the presentation, a short live-coding instructions, and the workshop transcript.

Demo setup

In its simplest form, the demos can be run simply by double clicking the html files in the demos directory (from lesson01.html to lesson16.html), or navigating to the live demos directly.

You can find the full list of demos and their short description on the demos subpage. You can launch the last, final demo (lesson16.html) during the talk if you want to show the actual gameplay and show the full source code.

Reference materials

Follow these external resources to learn more about HTML5 game development.

- MDN Games area
- HTML5 Game Devs forums
- HTML5 Gamedev Starter
- Gamedev.js Weekly newsletter
- #BBG IRC channel at Freenode

Frequently Asked Questions (FAQs)

Remember that the audience may have some questions about the topic. Be sure to feel comfortable with what you're presenting. Don't worry if you don't know the answer to a question - you can always tell them you'll check it and answer later, or you can point them to any of the external resources so they can check it themselves.

Should I start from scratch or use a framework?

It depends on whether you want to learn about the basics with a simple 2D game or already know JavaScript and want to build the game quickly. In the first case start from scratch, and in the second use the framework.

Feedback

Send pull requests if you find anything that should be fixed or updated.

Translations

The English version is the only one available right now, but the Polish version is being worked on and should be available in the near future. Feel free to translate the resources to your native language and send the pull request, so other developers can benefit from that.

About the author

This Content Kit was created by Andrzej Mazur - you can ping him on Twitter @end3r or send him an email. Feel free to ask questions, request training on using the Content Kit or preparing for the talk. Send feedback if you have any.

More about Content Kits

If you're interested in Content Kits you can find the full list on the MDN page. If you want to create one yourself there's Mozilla Wiki for that with all the details you need. The Content Kits project started as the collaboration between the Mozilla MDN Technical Writers and Technical Evangelism teams. They are here to help you, so don't hesitate if you need anything or have any questions: ask them directly via the #mdn IRC channel or the dev-mdc mailing list.

Gamedev Phaser Content Kit

Step-by-step tutorial split into 10 lessons

In this step-by-step tutorial we create a simple mobile MDN Breakout game written in JavaScript and using the Phaser framework. Every step has editable, live samples available to play with so you can see what the intermediate stages should look like. You will learn the basics of using the Phaser framework to implement fundamental game mechanics like rendering and moving images, collision detection, control machanisms, framework-specific helper functions, animations and tweens, and winning and losing states.

Here's the list of the articles:

To get the most out of this series of articles you should already have basic to intermediate JavaScript knowledge. After working through this tutorial you should be able to build your own simple Web games using Phaser.

You can also go back to the Content Kit's index page.

Gamedev Phaser Content Kit

Article 01: Initialize the framework

Before we can start writing the game's functionality, we need to create a basic structure to render the game inside. This can be done using HTML - Phaser framework will generate the Canvas element on its own.

The game's HTML

The HTML document structure is quite simple, as the game will be rendered entirely on the Canvas element generated by the framework. Using your favourite text editor, create a new HTML document, save it as `index.html`, in a sensible location, and add the following code to it:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Gamedev Phaser Workshop - lesson 01: Initialize the framework</title>
    <style>* { padding: 0; margin: 0; }</style>
    <script src="js/phaser.2.4.2.min.js"></script>
</head>
<body>
<script>
var game = new Phaser.Game(480, 320, Phaser.AUTO, null, {
    preload: preload, create: create, update: update
});
function preload() {}
function create() {}
function update() {}
</script>
</body>
</html>
```

We have a `charset` defined, `<title>` and some basic CSS in the header resetting the default `margin` and `padding`. There is also the `<script>` element including the Phaser's source code. The body also contains the `<script>` element, but we will render the game there and write the JavaScript code that controls it.

The Canvas element is generated automatically by the framework. We are initializing it by creating a new `Phaser.Game` object and assigning it to the game variable. The following parameters are: width and height of the Canvas, rendering method, `id` of the Canvas if it exist already and the names of the key functions. All the JavaScript code we will write in this tutorial will go between the opening <script> and closing </script> tags.

Our Canvas will be 480 pixels high and 320 pixels wide. Using `Phaser.AUTO` we are setting up an automatic way to render the game - the other two options are `CANVAS` and `WEBGL`. We can set one of them explicitly or use `AUTO` to let Phaser decide which one to use. It usually works in a way that if the WebGL context is available, the framework will use it, and fall back to Canvas 2D context if it's not available.

We don't have the Canvas element created on our own, so we don't have to enter any reference `id` and leave it blank as `null`. The last object contains the three key function names reserved for Phaser - we will use the same names to keep it clean. The `preload` function takes care of preloading the assets, `create` is executed once when everything is loaded and ready, and the `update` function is executed on every frame. These three are enough to make the proper mechanism of loading, starting and updating our game loop.

Compare your code

Here's the full source code of the first lesson, running live in a JSFiddle:

Next steps

Now we've set up the basic HTML and learned a bit about Phaser initialization, so let's continue to the second lesson and see how scaling work.

Gamedev Phaser Content Kit

Article 02: Scaling

We can make the game scale on any mobile device right at the beginning, so we don't have to worry about it later. That way it doesn't matter what the screen size there is - the game will adjust accordingly. There's a special `scale` object available in Phaser with a few handy methods and variables:

```
function preload() {
    game.scale.scaleMode = Phaser.ScaleManager.SHOW_ALL;
    game.scale.pageAlignHorizontally = true;
    game.scale.pageAlignVertically = true;
}
```

The `scaleMode` have a few different options on how the Canvas can be scaled:

- NO_SCALE
- EXACT_FIT
- SHOW_ALL
- RESIZE
- USER_SCALE

The first one, `NO_SCALE` does exactly that - nothing is scaled. The `EXACT_FIT` mode scales the Canvas to fill all the available space both vertically and horizontally, but it's not keeping the aspect ratio. The `SHOW_ALL` mode is something we will use as it scales the Canvas, but keeps the aspect ratio untouched, so the images won't be skewed like in the previous mode. There might be black stripes visible on the edges of the screen, but we can live with that. The `RESIZE` mode is a little bit more complicated as it creates the Canvas with the size of the available width and height, so you have to place the objects inside your game dynamically, which might be confusing at the beginning. The `USER_SCALE` is an extra mode that allows you to have custom dynamic scaling, calculating the size, scale and ratio on your own.

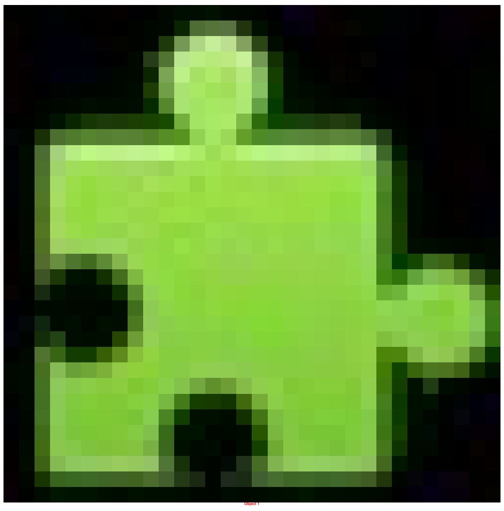
Two other lines of code in the preload function are responsible for aligning the Canvas element horizontally and vertically, so it is always centered on screen no matter how much space it take.

We can also add the background color to our Canvas, so it won't stay black. The `stage` object have the `backgroundColor` property that we can set using the CSS color definition syntax:

```
game.stage.backgroundColor = '#eee';
```

Compare your code

You can check the finished code for this lesson for yourself in the live demo below, and play with it to understand better how it works:



Next steps

Now we've set up the scaling, so let's continue to the third lesson and work out how to [load the assets and print them on screen]().

Gamedev Phaser Content Kit

Article 03: Load the assets and print them on screen

Our game's idea is to have the ball rolling on the screen bouncing off the paddle, hitting the bricks and collecting the points. Let's start with creating the JavaScript variable for the ball - add it between the `game` initialization and the `preload` function:

```
var game = new Phaser.Game(480, 320, Phaser.AUTO, null, {
    preload: preload, create: create, update: update
});

var ball;

function preload() {
    game.scale.scaleMode = Phaser.ScaleManager.SHOW_ALL;
    game.scale.pageAlignHorizontally = true;
    game.scale.pageAlignVertically = true;
    game.stage.backgroundColor = '#eee';
}
function create() {
}
function update() {
}
```

Note: for the sake of this tutorial we will use global variables. The purpose of the tutorial is to teach Phaser-specific approach to game development and don't enforce any of the available code style guides which are subjective to preferences of every single developer.

Loading and printing the images on our Canvas is a lot easier using Phaser than doing that in pure JavaScript. To load the asset, we will use the `game` object created by Phaser and execute the `load.image` method inside the `preload` function:

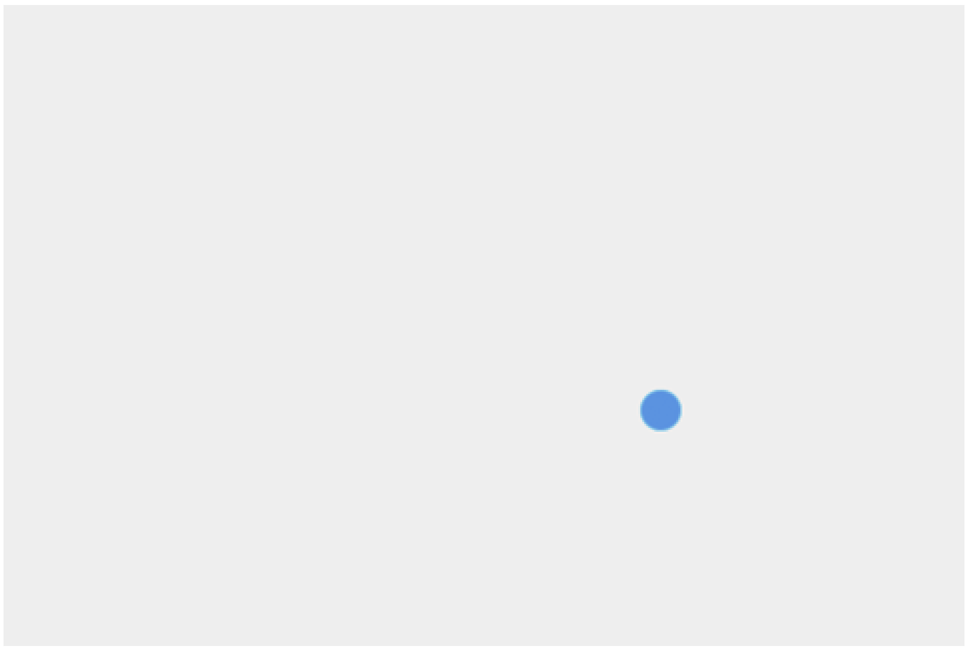
```
function preload() {
    // ...
    game.load.image('ball', 'img/ball.png');
}
```

The first parameter is the name of the asset we want to have - it will be used across our game, so it would be great to give it a proper meaning. The second parameter is the relative path to the graphic asset. In our case we will load the image of the blue ball that will be used later in the game to destroy blocks and earn points.

Now, to show it on the screen we will use another Phaser method called `add.sprite` in the `create` function:

```
function create() {
    ball = game.add.sprite(50, 50, 'ball');
}
```

It will add the ball and render it on the screen. The first two parameters are the top and left positions on the screen and the third one is the name of the asset we defined earlier. That's it - if you launch the `index.html` file you will see the image already loaded and rendered on Canvas!



Compare your code

You can check the finished code for this lesson for yourself in the live demo below, and play with it to understand better how it works:

Next steps

Printing out the ball was easy, so let's try to move the ball on screen.

Gamedev Phaser Content Kit

Article 04: Move the ball

We have our blue ball printed on screen, but it's doing nothing. It would be cool to make it move somehow. Remember the `update` function and its definition? The code inside it is executed on every frame, so it's a perfect place to put the code that will update the ball's position on screen:

```
function update() {
    ball.x += 1;
    ball.y += 1;
}
```

The code above will add 1 to `x` and `y` representing the position of the ball on screen, on each frame. That's it, the ball is rolling!

Compare your code

You can check the finished code for this lesson for yourself in the live demo below, and play with it to understand better how it works:

Next steps

Now the next step would be to add some basic collision detection, so our ball can bounce off the walls. We could write down the JavaScript code that check all the walls against our ball and change its direction accordingly. It would take some lines of code and complicate it, even more if we want to add paddle and bricks collisions in the next steps, but we can take the advantage of using the Phaser framework and do it nice and easy. To do that though we need to introduce physics first.

Gamedev Phaser Content Kit

Article 05: Physics

For the proper use of collision detection between objects in our game we will need to have physics. Phaser is bundled with three different physics engines: Arcade Physics, P2 and Ninja Physics with the fourth Box2D available as a commercial plugin. For simple games, which in our case is the Breakout game, we can use the Arcade Physics engine. We don't need any heavy geometry calculations, after all it's just a ball bouncing off the walls and bricks.

First, let's initialize the Arcade Physics in our game. Write down the `physics.startSystem` method at the beginning of the `create` function:

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

Next, we need to enable our ball for the physics system - Phaser objects are not enabled for that by default:

```
game.physics.enable(ball, Phaser.Physics.ARCADE);
```

Then, if we want to move our ball on the screen, we can set `velocity` to its `body`:

```
ball.body.velocity.set(150, 150);
```

Remember to remove our old method of adding values to `x` and `y` in the `update` function:

function update() {
~~   ball.x += 1;~~
~~   ball.y += 1;~~
}

All the code should look like this:

```
var ball;

function preload() {
    game.scale.scaleMode = Phaser.ScaleManager.SHOW_ALL;
    game.scale.pageAlignHorizontally = true;
    game.scale.pageAlignVertically = true;
    game.stage.backgroundColor = '#eee';
    game.load.image('ball', 'img/ball.png');
}

function create() {
    game.physics.startSystem(Phaser.Physics.ARCADE);
    ball = game.add.sprite(50, 50, 'ball');
    game.physics.enable(ball, Phaser.Physics.ARCADE);
    ball.body.velocity.set(150, 150);
}

function update() {
}
```

The ball is now moving constantly in given direction, because the physics engine have the gravity and friction defaults set to zero. Adding gravity would result in the ball falling down while friction would eventually stop the ball after some time.

Compare your code

You can check the finished code for this lesson for yourself in the live demo below, and play with it to understand better how it works:

Next steps

Now we can move to the next lesson and see how to make the ball bounce off the walls.

Gamedev Phaser Content Kit

Article 06: Bounce off the walls

Now when the physics are introduced, we can start implementing collision detection into the game - first comes the walls. The easiest way to do it is to tell the framework that we want to treat boundaries of the Canvas element as walls and bounce the ball off of them. Add this line right after the place where we enabled the physics for the ball:

```
ball.body.collideWorldBounds = true;
```

Now the ball stops at the edge of the screen instead of disappearing, but it doesn't look like bouncing. To make it work we have to set its bounciness:

```
ball.body.bounce.set(1);
```

Now it works as expected - the ball bounces off all the walls and moves inside the world, our Canvas area.

Compare your code

You can check the finished code for this lesson for yourself in the live demo below, and play with it to understand better how it works:

Next steps

It looks nice already, but we can't control the game in any way - it's the high time to introduce player paddle and controls.

Gamedev Phaser Content Kit

Article 07: Player paddle and controls

We have the ball moving and bouncing off the walls, but it gets boring after some time. There's no interactivity, we can't do anything with the ball. It feels more like a movie than a game. We need a way to introduce the gameplay, so let's create the paddle that we will be able to move and use to hit and bounce the ball. From the framework point of view the paddle is very similar to the ball - we need a variable, load the image, and then do the magic.

First, add the `paddle` variable we will be using in our game, right after the `ball` variable:

```
var ball;
var paddle;
```

Then, in the `preload` function, load the `paddle` image:

```
function preload() {
    // ...
    game.load.image('ball', 'img/ball.png');
    game.load.image('paddle', 'img/paddle.png');
}
```

After that, we can init our paddle in the `create` function:

```
paddle = game.add.sprite(game.world.width*0.5, game.world.height-5, 'paddle');
```

We can use the `world.width` and `world.height` values to position the paddle exactly where we want it: `game.world.width*0.5` will be right in the middle of the screen. In our case the world is the same as the Canvas, but for other types of games, like side-scrollers for example, you can tinker with it and create interesting effects.

As you probably noticed the paddle is not exactly in the middle. Why? Because the anchor from which the position is calculated always starts from the top left edge of the object. We can change that to have the anchor in the middle of the paddle's width and at the bottom of it's height, so it's easier to position against the bottom edge:

```
paddle.anchor.set(0.5,1);
```

The paddle is positioned right where we wanted it to be. Now, to make it collide with the ball we have to enable physics. Continue adding the code in the `create` function:

```
game.physics.enable(paddle, Phaser.Physics.ARCADE);
```

And then the magic happens - the framework will take care of checking the collision detection on every frame. To do that, just add the `collide` method to the `update` function:

```
function update() {
    game.physics.arcade.collide(ball, paddle);
}
```

The first parameter is one of the objects, in our case the ball, and the second is the paddle. It works, but not quite as we expected it to work - when the ball hits the paddle, the paddle fall off the screen!

All we want is the ball bouncing off the paddle and the paddle staying in the same place. We can set the `body` of the paddle to be `immovable`, so the ball won't move it when it hits it. Add that line at the bottom of the `create` function:

```
paddle.body.immovable = true;
```

Now it works! The next problem is - we can't move it. To do that we can use the input (whether mouse or touch) and set the paddle position to where the `input` position is. Add the next line to the `update` function:

```
function update() {
    game.physics.arcade.collide(ball, paddle);
    paddle.x = game.input.x;
}
```

Now on every new frame the paddle's `x` position will adjust accordingly to the input's `x` position. But when we start the game, the position of the paddle is not in the middle. It's because the input position is not yet defined. To fix that we can set the default position in the middle of the screen if it's not available:

```
paddle.x = game.input.x || game.world.width*0.5;
```

It works as expected.

Compare your code

You can check the finished code for this lesson for yourself in the live demo below, and play with it to understand better how it works:

Next steps

We can move the paddle and bounce off the ball, but what's the point if the ball is bouncing off the bottom edge of the screen anyway? Let's introduce the possibility of losing - a game over logic.

Gamedev Phaser Content Kit

Article 08: Game over

To make the game more interesting we can introduce the ability to lose - if you don't hit the ball before it reaches the bottom edge of the screen it will be game over. To do that, we need to disable the ball's collision with the bottom edge of the screen. Add the code below in the `create` function where you define the ball's attributes:

```
game.physics.arcade.checkCollision.down = false;
```

This will make the three walls (top, left and right) collidable against the ball, but the fourth (bottom) will dissapear. The ball will go out of screen at the bottom, so we need a way to detect that and act accordingly:

```
ball.checkWorldBounds = true;
ball.events.onOutOfBounds.add(function(){
    alert('Game over!');
    location.reload();
}, this);
```

Adding those lines will make the ball check the world (in our case Canvas) bounds and execute the function bound to the `onOutOfBounds` event. Inside it, when you click on the alert, the page will be reloaded so you can play again.

Compare your code

You can check the finished code for this lesson for yourself in the live demo below, and play with it to understand better how it works:

Next steps

Now when the basic gameplay is in place let's make it more interesting by introducing the bricks we can smash - it's time to build the brick field.

Gamedev Phaser Content Kit

Article 09: Build the brick field

Building the brick field is a little bit more complicated than adding a single objects to the screen. It's still easier with Phaser than doing that in pure JavaScript though. We will create a group of bricks and print them on screen in a loop. First, let's define the needed variables:

```
var bricks;
var newBrick;
var brickInfo;
```

The `bricks` variable will be used to create a group, `newBrick` a new object added to the group on every iteration of the loop and `brickInfo` will store all the data we need. Next, let's load the image of the brick:

```
function preload() {
    // ...
    game.load.image('brick', 'img/brick.png');
}
```

We will place all the code inside the `initBricks` function to keep it separated from the rest of the code. Add `initBricks` at the end of the `create` function:

```
function create(){
    // ...
    initBricks();
}
```

Write down the `initBricks` function's body at the end of our games's code, just before the closing </script> tag. Let's start with `brickInfo` - it will come in handy really quick:

```
function initBricks() {
    brickInfo = {
        width: 50,
        height: 20,
        count: {
            row: 3,
            col: 7
        },
        offset: {
            top: 50,
            left: 60
        },
        padding: 10
    }
}
```

This `brickInfo` object will hold all the information we need: width and height of the single brick, number of rows and columns of bricks we will see on screen, top and left starting point where we begin to print the bricks on screen and the padding between single bricks. Now, let's start creating the bricks themselves - add the empty group first:

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

We can loop through the rows and columns to create new brick on every iteration:

```
for(c=0; c<brickInfo.count.col; c++) {
    for(r=0; r<brickInfo.count.row; r++) {
        // create new brick and add it to the group
    }
}
```

That way we will create the exact number of bricks we need and have them all in a group.

```
for(c=0; c<brickInfo.count.col; c++) {
    for(r=0; r<brickInfo.count.row; r++) {
        var brickX = 0;
        var brickY = 0;
        newBrick = game.add.sprite(brickX, brickY, 'brick');
        game.physics.enable(newBrick, Phaser.Physics.ARCADE);
        newBrick.body.immovable = true;
        newBrick.anchor.set(0.5);
        bricks.add(newBrick);
    }
}
```

We're looping through the rows and columns to create the new brick and place it on the screen - at coordinates (0,0) for now, but we will calculate the proper positions in a minute. The newly created brick is enabled for Arcade physics engine, it's body is set to be immovable (so it won't move when hit by the ball), we're also setting the anchor to be in the middle and adding the brick to the group.

The problem is that we're painting all bricks in one place, at coordinates (0,0). What we need to do is figure out a way to have x and y position of new brick in each loop iteration:

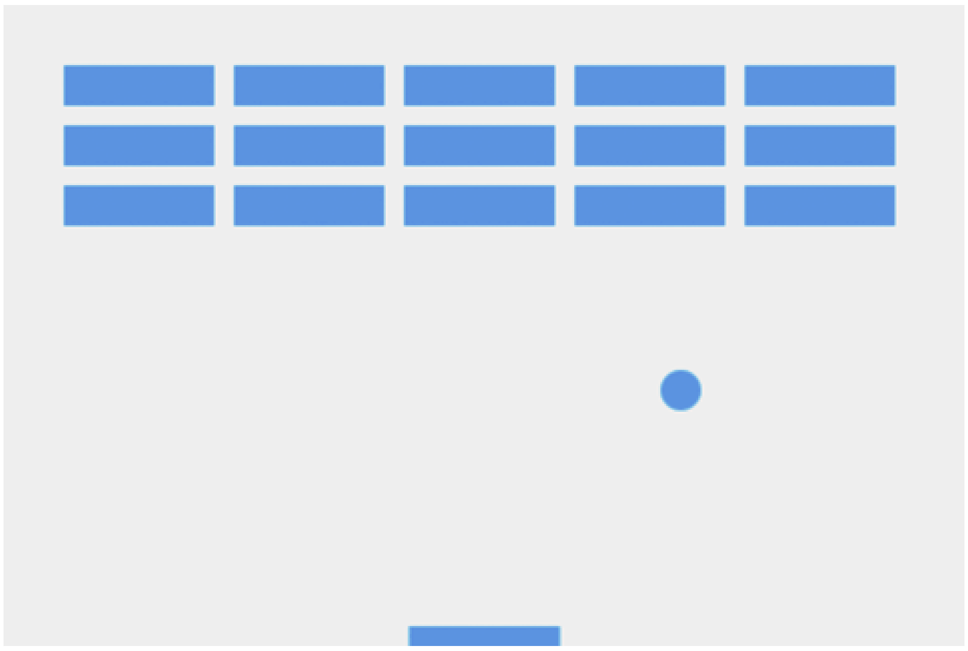
```
var brickX = (r*(brickInfo.width+brickInfo.padding))+brickInfo.offset.left;
var brickY = (c*(brickInfo.height+brickInfo.padding))+brickInfo.offset.top;
```

Each `brickX` position is worked out as `brickInfo.width` plus `brickInfo.padding`, multiplied by the row number, `r`, plus the `brickInfo.offset.left`; the logic for the `brickY` is identical except that it uses the values for column number, `c`, `brickInfo.height`, and `brickInfo.offset.top`. Now every single brick can be placed in its correct place row and column, with padding between each brick, drawn at an offset from the left and top Canvas edges.

Here is the complete code for the `initBricks` function:

```
function initBricks() {
    brickInfo = {
        width: 50,
        height: 20,
        count: {
            row: 3,
            col: 7
        },
        offset: {
            top: 50,
            left: 60
        },
        padding: 10
    }
    bricks = game.add.group();
    for(c=0; c<brickInfo.count.col; c++) {
        for(r=0; r<brickInfo.count.row; r++) {
            var brickX = (c*(brickInfo.width+brickInfo.padding))
+brickInfo.offset.left;
            var brickY = (r*(brickInfo.height+brickInfo.padding))
+brickInfo.offset.top;
            newBrick = game.add.sprite(brickX, brickY, 'brick');
            game.physics.enable(newBrick, Phaser.Physics.ARCADE);
            newBrick.body.immovable = true;
            newBrick.anchor.set(0.5);
            bricks.add(newBrick);
        }
    }
}
```

Everything works fine as expected - the bricks are printed on screen in an evenly distances from each other.



Compare your code

You can check the finished code for this lesson for yourself in the live demo below, and play with it to understand better how it works:

Next steps

Something is missing though. The ball goes through the bricks without stopping - we need proper collision detection.

Gamedev Phaser Content Kit

Article 10: Collision detection

I hope you survived the previous lesson, it was quite intense. Now onto the next challenge - the collision detection between the ball and the bricks. Luckily enough we can use the physics engine to check collisions not only between single objects (like the ball and the paddle), but also between an object and the group. This makes everything a lot easier:

```
function update() {
    game.physics.arcade.collide(ball, paddle);
    game.physics.arcade.collide(ball, bricks, ballHitBrick);
    paddle.x = game.input.x || game.world.width*0.5;
}
```

The ball's position is calculated against the positions of all the bricks in the group. The third, optional parameter is the function executed when the collision occurs. Create that function just before the closing </script> tag:

```
function ballHitBrick(ball, brick) {
    brick.kill();
}
```

As you can see thanks to Phaser there are two parameters passed to the function - first one is the ball which we explicitly defined in the collide method, and the second one is the single brick from the bricks group that the ball is colliding with. When we have the reference to it, we can then kill the brick which will remove it from the screen.

Sorry to disappoint you if you expected to have more calculations for collision detection on your own if you coded it in pure JavaScript. That's the beauty of using the framework - you can leave a lot of boring code to it, just like in our case, and focus on the most fun and interesting parts of making a game.

Compare your code

You can check the finished code for this lesson for yourself in the live demo below, and play with it to understand better how it works:

Next steps

We can hit the bricks and remove them, which is a nice addition to the gameplay already. It would be even better to count the destroyed bricks and keep it as the score.

Gamedev Phaser Content Kit

Article 11: The score

Having a score can boost player's interest in the game - you can try to beat your own or your friend's highscore and play the game again and again. To make it work in our game we will use separate variable for storing the score and Phaser's text element to print it out on the screen. Add those two variables right after the previously defined ones:

```
// ...
var scoreText;
var score = 0;
```

Now add this line at the end of the `create` function:

```
scoreText = game.add.text(5, 5, 'Points: 0', { font: '18px Arial', fill:
'#0095DD' });
```

As you can see the text element can take four parameters: x and y position, the actual text that will be rendered and the font style - the last parameter looks very similar to CSS styling. In our case the score text will be blue and will have 18 pixels using the Arial font. We will increase the number of points every time the ball hits the brick and update the text with the current value using the `setText` method:

```
function ballHitBrick(ball, brick) {
    brick.kill();
    score += 10;
    scoreText.setText('Points: '+score);
}
```

That's it - the score is now added on every hit and updated on the screen.

Compare your code

You can check the finished code for this lesson for yourself in the live demo below, and play with it to understand better how it works:

Next steps

We have the score already, but what's the point of playing and keeping the score if you can't win? Let's see how we can make it work to win the game.

Gamedev Phaser Content Kit

Article 12: Win the game

Implementing winning in our game is quite easy: if you happen to destroy all the bricks, then you win:

```
function ballHitBrick(ball, brick) {
    brick.kill();
    score += 10;
    scoreText.setText('Points: '+score);
    if(score === brickInfo.count.row*brickInfo.count.col*10) {
        alert('You won the game, congratulations!');
        location.reload();
    }
}
```

If your score is the same as the number of bricks initially available on the screen - times 10 as you score 10 points instead of 1 for every brick - then show the proper message.

Compare your code

You can check the finished code for this lesson for yourself in the live demo below, and play with it to understand better how it works:

Next steps

Both losing and winning is already implemented, so the core gameplay of our game is finished. Now we can add something extra, like three lives instead of one.

Gamedev Phaser Content Kit

Article 13: Extra lives

We can make the game enjoyable for longer by adding lives. If the player lose one life he can continue playing until he runs out of them. Add the variables we will need for lives and the text message that will be shown on screen when the player lose one of his lives:

```
var lives = 3;
var livesText;
var lifeLostText;
```

Defining the texts look like something we already did in the score lesson:

```
scoreText = game.add.text(5, 5, 'Points: 0', { font: '18px Arial', fill:
'#0095DD' });
livesText = game.add.text(game.world.width-5, 5, 'Lives: '+lives, { font: '18px
Arial', fill: '#0095DD' });
livesText.anchor.set(1,0);
lifeLostText = game.add.text(game.world.width*0.5, game.world.height*0.5, 'Life
lost, click to continue', { font: '18px Arial', fill: '#0095DD' });
lifeLostText.anchor.set(0.5);
lifeLostText.visible = false;
```

The `lifeText` object looks very similar to the `scoreText` one - have a position on the screen, the actual text and the font styling. It is also anchored on its top right edge to align properly with the screen.

The `lifeLostText` will be shown only when the life will be lost, so its visibility is set to `false` at the beginning. Its anchor is set in the middle of the text to show it in the center of the screen.

As you probably noticed we're using the same styling for all three texts: `scoreText`, `livesText` and `lifeLostText`. If we ever want to change the font size or color we will have to do it in multiple places. To make it easier for us to maintain in the future we can create a separate variable that will hold our styling, let's call it `textStyle`:

```
textStyle = { font: '18px Arial', fill: '#0095DD' };
```

We can use this variable when creating the texts:

```
scoreText = game.add.text(5, 5, 'Points: 0', textStyle);
livesText = game.add.text(game.world.width-5, 5, 'Lives: '+lives, textStyle);
livesText.anchor.set(1,0);
lifeLostText = game.add.text(game.world.width*0.5, game.world.height*0.5, 'Life
lost, click to continue', textStyle);
lifeLostText.anchor.set(0.5);
lifeLostText.visible = false;
```

This way changing the font in one variable will apply the changes in every place it is used.

To implement lives in our game, let's first change the ball's function bound to the `onOutOfBounds` event. Instead of executing the function right away and showing the alert:

```
~~ball.events.onOutOfBounds.add(function(){~~
~~    alert('Game over!');~~
~~    location.reload();~~
~~}, this);~~
```

We will assign new function called `ballLeaveScreen`:

```
ball.events.onOutOfBounds.add(ballLeaveScreen, this);
```

We want to decrease the number of lives every time the ball leaves the Canvas. Add that function at the end of our code:

```
function ballLeaveScreen() {
    lives--;
    if(lives) {
        livesText.setText('Lives: '+lives);
        lifeLostText.visible = true;
        ball.reset(game.world.width*0.5, game.world.height-25);
        paddle.reset(game.world.width*0.5, game.world.height-5);
        game.input.onDown.addOnce(function(){
            lifeLostText.visible = false;
            ball.body.velocity.set(150, -150);
        }, this);
    }
    else {
        alert('You lost, game over!');
        location.reload();
    }
}
```

Instead of instantly printing out the alert when you lose the life, we will subtract one life from the current number and check if it's a non-zero value. If yes, then the player still have some lives left and can continue to play - he will see the life lost message, the ball's and paddle's positions will be reset on screen and on the next input (click or touch) the message will be hidden and the ball will start to move.

If the number of available lives will reach zero, then the game is over and the proper alert message will be shown.

Compare your code

You can check the finished code for this lesson for yourself in the live demo below, and play with it to understand better how it works:

Next steps

Lives made the game more forgiving - if you lose one life, you still have two more left and can continue to play. Now let's expand the look and feel of the game by adding animations and tweens.

Gamedev Phaser Content Kit

Article 14: Animations and tweens

To make the game look more juicy and alive we can use the advantages of animations and tweens. It will result in having the game experience feel better and look more entertaining. For example, we can make the ball wobble when it hits something - we would use the spritesheet for that:

```
game.load.spritesheet('ball', 'img/wobble.png', 20, 20);
```

Instead of loading a single image of the ball we can load the whole spritesheet, a collection of different images.

We will use them to show the frames in a particular way and create an illusion of animation. The spritesheet method's two extra paremeters are determining the width and height of the single frame in the given spritesheet file.

```
ball = game.add.sprite(50, 250, 'ball');
ball.animations.add('wobble', [0,1,0,2,0,1,0,2,0], 24);
```

To add an animation to the object use the `animations.add` method. The first parameter is the name we chose for the animation, the second is the array of frames from the file mentioned earlier and third is the framerate. In the method handling the collision between the ball and the paddle we can add an extra function that will be executed every time the collision happen, just like we're handling the `ballHitBrick` function:

```
function update() {
    game.physics.arcade.collide(ball, paddle, ballHitPaddle);
    game.physics.arcade.collide(ball, bricks, ballHitBrick);
    paddle.x = game.input.x || game.world.width*0.5;
}
```

Then we can create that function (having `ball` and `paddle` as default parameters) and play the wobble animation inside it:

```
function ballHitPaddle(ball, paddle) {
    ball.animations.play('wobble');
}
```

The animation is played every time the ball hits the paddle. We can add that line to `ballHitBrick` too if we feel it would make the game look better.

Tweens

Animations are now covered, so let's see how tweens work:

```
function ballHitBrick(ball, brick) {
    var killTween = this.add.tween(brick.scale);
    killTween.to({x:0,y:0}, 200, Phaser.Easing.Linear.None);
    killTween.onComplete.addOnce(function(){
        brick.kill();
    }, this);
    killTween.start();
    score += 10;
    scoreText.setText('Points: '+score);
    if(score === brickInfo.count.row*brickInfo.count.col*10) {
        alert('You won the game, congratulations!');
        location.reload();
    }
}
```

When defining a new tween you have to specify which property will be tweened - in our case, instead of hiding the bricks instantly when hit by the ball, we will make their width and height scale to zero, so they will nicely dissapear. The `to` method takes the object of desired parameters and values, time of the tween in miliseconds and the type of tween from the defined collection. We can also add the optional `onComplete` function which will be executed when the tween will be finished. The last thing do to is to start the tween right away.

That's the expanded version of the tween definition, but we can also use the shorthand syntax:

```
game.add.tween(brick.scale).to({x:2,y:2}, 500, Phaser.Easing.Elastic.Out, true, 
100);
```

This tween will double the brick's scale in half the second using the Elastic easing, will start automatically and have a delay of 100 miliseconds.

Compare your code

You can check the finished code for this lesson for yourself in the live demo below, and play with it to understand better how it works:

Next steps

Animations and tweens look very nice, but we can add even more to the game - we'll handle the button input.

Gamedev Phaser Content Kit

Article 15: Buttons

Instead of starting the game right away we can leave that decision to the player by adding the Start button he can press. Before that, the game would wait for the input. We will need a variable storing the boolean value of whether or not the game is currently being played, and the second one for the button:

```
var playing = false;
var startButton;
```

We can load the button spritesheet the same way we loaded the ball's wobble animation:

```
game.load.spritesheet('button', 'img/button.png', 120, 40);
```

A single button frame is 120 pixels wide and 40 pixels high:

START
START
START

Adding the new button to the game is done by using the `add.button` method:

```
startButton = game.add.button(game.world.width*0.5, game.world.height*0.5,
'button', startGame, this, 1, 0, 2);
startButton.anchor.set(0.5);
```

We define the button's x and y position, the graphic asset name, callback function that will be executed when the button will be pressed, reference to `this` to have the execution context available, and the frames that will be used for the *over*, *out* and *down* events.

The over event is the same as hover, out is when the pointer moves out of the button and down is when the button is pressed. Now, to make it work we need to define the `startGame` function:

```
function startGame() {
    startButton.destroy();
    ball.body.velocity.set(150, -150);
    playing = true;
}
```

It removes the button, sets the ball's initial velocity and the `playing` variable to `true`. It works as expected, but we can still move the paddle when the game hasn't started yet. We can leave it as it is, but we can also take an advantage of the `playing` variable and make the paddle movable only when the game is on. To do that, adjust the `update` function to check that:

```
function update() {
    game.physics.arcade.collide(ball, paddle, ballHitPaddle);
    game.physics.arcade.collide(ball, bricks, ballHitBrick);
    if(playing) {
        paddle.x = game.input.x || game.world.width*0.5;
    }
}
```

That way the paddle is immovable after everything is loaded and prepared, but before the start of the actual game.

Compare your code

You can check the finished code for this lesson for yourself in the live demo below, and play with it to understand better how it works:

Next steps

The last thing to add and make the gameplay even more interesting is some randomization in the way the ball bounces off the paddle.

Gamedev Phaser Content Kit

Article 16: Randomizing gameplay

Our game appears to be completed, but if you look close enough you'll notice that the ball is bouncing off the paddle on the same angle throughout the whole gameplay. You can predict and remember where it will go the more you play, which isn't very helpful in terms of high replayability. To fix that we can change the ball's velocity depending on the exact spot it hits the paddle:

```
function ballHitPaddle(ball, paddle) {
    ball.animations.play('wobble');
    ball.body.velocity.x = -1*5*(paddle.x-ball.x);
}
```

It's a little bit of magic - the new velocity is higher the bigger distance there is between the center of the paddle and the place where the ball hits it. Also, the direction (left or right) is determined by that value - if the ball hits the left side of the paddle, it will bounce left and hitting the right side will bounce it to the right. It ended up that way because of a little bit of experiments with the given values, you can change them on your own and see what happens.

Compare your code

You can check the finished code for this lesson for yourself in the live demo below, and play with it to understand better how it works:

Summary

You've finished all the lessons - congratulations! By this point you would have learnt the basics of Phaser and the logic behind simple 2D games.

You can do a lot more in the game, add whatever you feel would be the best. It's a basic intro scratching the surface of the countless helpful methods that Phaser provide. Be sure the check the ever growing list of examples and the official documentation, and visit the HTML5 Gamedevs forums if you ever need any help.

You could also go back to this tutorial series' index page.

http://end3r.github.io/Gamedev-Phaser-Content-Kit/

https://end3r.github.io/Gamedev-Phaser-Content-Kit/slides/