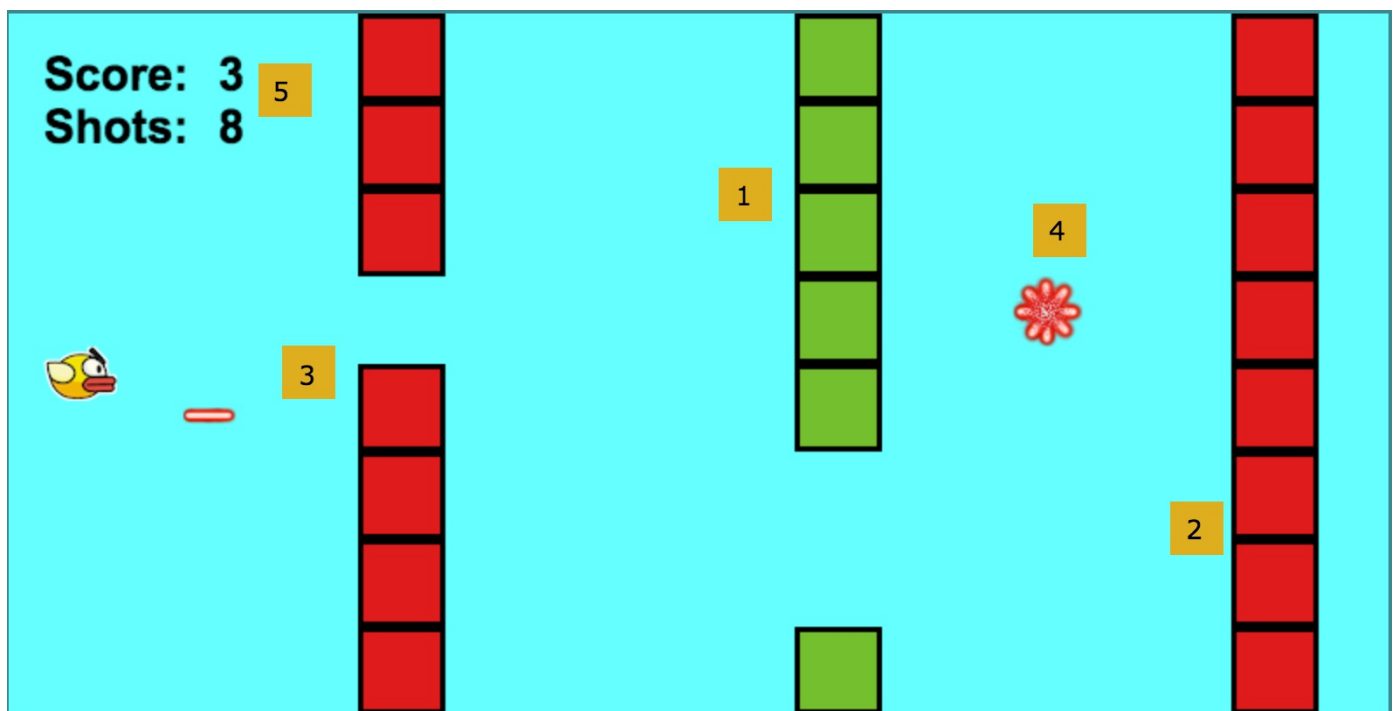


# An Extension to Your Flappy Bird

By this point in the course you have a working clone of the game Flappy Bird. A possible extension to the game would be to add a new kind of obstacle that can only be avoided by destroying it. This would add variety to the gameplay, as the player has to shoot at times as well as jump.

The screenshot below shows what we are aiming for.



We have the typical green pipe to jump through (1), but we also have the new red pipe that generates with no gaps (2). Our player can now shoot lasers to clear a path through the pipe (3). The player can also pick up more ammunition (4), so that they don't run out of shots (5).

To add these features to your game, we need to make several changes. We will go through them one at a time. The changes we must make are:

1. Make both red and green pipes be generated.
2. Allow the bird to shoot lasers.
3. Make the lasers destroy the red pipes, but be destroyed by the green pipes.
4. Make the player have a limited number of shots.
5. Have extra ammunition spawn.
6. Allow the player to pick up new ammo.

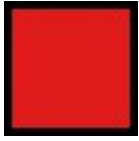
After some of these steps we will have a complete and playable game. For example, if you wish to stop after completing step 3, the player will have unlimited ammunition and so can destroy as many red pipes as they like, and this is a valid game. However, if you were to stop after step 2 then the player would not be able to get past red pipes, as they couldn't be destroyed, and if you were to stop after step 4, then the player would eventually run out of shots, as there would be no extra ammunition spawning. Stopping after step 5 would be especially cruel, as ammunition would spawn, but the player would be unable to pick it up!

Now that we understand what we are aiming to do, and have a little understanding of how we'll do it, let's get started!

## **Step 1: Pipe Generation**

This step should be relatively easy, as we have an example of how to

generate pipes already! First, before we can add the pipe, we need to get the image of the pipe block itself, the *asset*. I used the following image:



which can be found here (<https://github.com/rjmcfl/webapp-flappy/raw/master/assets/pipeRed.png>). Once you have this file in your assets folder, then loading the image and displaying it is easy, and something you have done many times before!

**Challenge:** Load the image of the pipe block, call it `pipeRed` and display it in the top left hand corner of the screen.

Did you remember how to do it? In case you didn't, here is the code we need:

```
function preload() {  
    ... // Previous code here  
    game.load.image("pipeRed", "../assets/pipeRed.png");  
}  
  
function create() {  
    ... // Previous code here  
    game.add.sprite(0,0,"pipeRed");  
}
```

Make sure you delete the `game.add.sprite(0,0,"pipeRed");` line

when you have made sure it works.

Now that we have loaded the relevant asset, we need to turn it into a pipe. If we look at our old pipe generation code, we see we have two functions, `addPipeBlock(x,y)` and `generatePipe()`. These are both used to make our green pipes. Because we will need to make our red pipes in much the same way, let's add two new functions, with empty function blocks:

```
function addRedPipeBlock(x,y) {  
  
}  
  
function generateRedPipe() {  
  
}
```

The other thing that we needed when we generate the green pipes is a place to store them, an *array*. Because we want to keep our red pipes separate from our green pipes, let's add another array to keep track of them:

```
var pipes = [] // You already have this line.  
var redPipes = [] // This line should be added.
```

**Challenge:** Can you fill in the function block for the `addRedPipeBlock(x,y)` function? It is very similar to our original

`addPipeBlock(x,y)` function, just remember which asset you are loading!

Here is the code to go in the first function:

```
function addRedPipeBlock(x,y) {  
    var redPipeBlock = game.add.sprite(x,y,"pipeRed");  
    redPipes.push(redPipeBlock);  
    game.physics.arcade.enable(redPipeBlock);  
    redPipeBlock.body.velocity.x = -200;  
}
```

The code for the second function is also very similar to that of `generatePipe()`, except that we wanted a gap in the green pipes, and we want **no** gap in the red pipes.

**Challenge:** Can you fill in the function block for the `generateRedPipe()` function?

Here is the code for the second function. Notice how much simpler it is than the code to generate a green pipe, as there is no randomness and no gap.

```
function generateRedPipe() {  
    for (var count = 0; count < 8; count++) {  
        addRedPipeBlock(800, count*50);  
    }  
}
```

```
changeScore();  
}
```

Great! We have the functions that do what we want, now we just need to call them at the right time! We want a pipe to generate at set intervals of time (controlled by `pipeInterval`), and we want each pipe to be either red or green. Let's write a new function that chooses, at random, to generate either a red pipe or a green pipe. Then, every time we want a pipe to be generated, we can just call this function. Let's call it `chooseGen()`, because we are choosing which pipe to generate:

```
function chooseGen() {  
  
}
```

At this point, you can personalise your game a little! I decided to make the red pipes generate a lot less frequently than the green pipes, so that only 1 in 6 pipes generated would be red.

```
function chooseGen() {  
    var type = game.rnd.integerInRange(1, 6);  
    if (type == 1) {  
        generateRedPipe();  
    } else  
        generatePipe();  
    }  
}
```

Let's have a little look at what this code is doing. Each time the function is called, `type` will be assigned 1, 2, 3, 4, 5 or 6 at random. If `type == 1`, then a red pipe will be generated. Otherwise (i.e. if `type` is 2, 3, 4, 5 or 6), a green pipe will be generated. This means that the chance of a red pipe being generated is 1 in 6. You can make this more likely by reducing the number 6 to some other number. For example, writing `type = game.rnd.integerInRange(1, 2);` would cause a red pipe to be generated about half the time, a chance of 1 in 2.

**Challenge:** *How would you make red pipes be generated more than green pipes? Try to make the chance of a green pipe being generated 1 in 4.*

Now we need to make sure that `chooseGen()` is called instead of `generatePipe()` in the timer:

```
function create() {  
    ... // Other code here  
  
    // We remove this code...  
    /*  
    game.time.events  
        .loop(pipeInterval * Phaser.Timer.SECOND,  
            generatePipe);  
    */  
}
```

```
// And replace it with this.  
game.time.events  
    .loop(pipeInterval * Phaser.Timer.SECOND,  
    chooseGen);  
}
```

If you play the game now, the bird will simply sail through the red pipes as if they weren't there! To fix this, we need to add just one more thing...

***Challenge:*** What needs to be added? What else did we do to make the bird die when it hit the pipes? Add the necessary code.

Of course, we need to make the bird collide with the red pipes too! This means we simply add this code to `update()` :

```
function update() {  
    ... // Previous code here  
    game.physics.arcade.  
        overlap(player, redPipes,  
        gameOver);  
}
```

There, now we have both red and green pipes being generated, and both are dangerous! Before we move on, let's make a general point about good practice when coding...



# Code Reuse

Notice that in each function `generatePipe()` and `generateRedPipe()` we call the function `changeScore()`. This is not only a waste of space, but also a bit dangerous. What happens if you want to add another option for things to generate? We would have to remember to add the `changeScore()` call to this new function as well, and this is quite easy to forget.


A better way to do this is to notice that whatever gets generated, somewhere during the `chooseGen()` function we change the score. This means we can remove the line from each of the `generateBlah()` functions and simply add it to the end of the `chooseGen()` function:

```
function chooseGen() {  
    ... // Other code here  
    changeScore();  
}
```

This fixes our potential problem before it even has the chance to be a problem! Remember, as a programmer, you need to protect yourself from your future mistakes. Make it as easy for yourself in the future, whether you're reading your code or adding to it, as possible!

There, we've done the easy bit, now let's move on to something a little trickier!

## Step 2: Shooting Lasers

First off, let's load the asset. I used the following image:  which can be found here (<https://github.com/rjmcfl/webapp-flappy/raw/master/assets/laser.png>).

**Challenge:** You know what to do! Load that asset and name it `laser`!

The code we need is given below.

```
function preload() {  
    ... // Previous code here  
    game.load.image("laser", "../assets/laser.png");  
}
```

We also need a way to keep track of these lasers, a little like we keep track of the pipe blocks, so let's add another array to the start of the code:

```
var lasers = [];
```

Now we need a way to shoot our laser. Here is an opportunity for more customisation! I decided to make the X key shoot a laser, but you can make it any key you wish. A list of all the keys that can be used is given here (<http://phaser.io/docs/2.4.4/Phaser.KeyCode.html>). We then need to

add an empty function block for the function we will call

`shootLaser()` :

```
function shootLaser() {  
  
  
}
```

**Challenge:** Add something to the `create()` function block that will mean that the `shootLaser()` function is called whenever you press the desired key (I chose X). If you need help, look at how we made the bird jump using the spacebar...

To bind a key to a function, we add the following code to the `create()` function block:

```
game.input.keyboard  
    .addKey(Phaser.Keyboard.X)  
    .onDown.add(shootLaser);
```

This ensures that whenever we press the X key, the function `shootLaser` will be called. **Remember to write `shootLaser` and NOT `shootLaser()` !**

Now we just need to fill in our `shootLaser()` function block. Adding a laser to the screen is very similar to adding a pipe block to the screen. We want four things to happen:

1. Add the image in the correct place.
2. Keep track of the laser by adding it to the array.
3. Give the laser physics so it can move.
4. Make the laser move. (But in what direction? How fast?)

**Challenge:** Try to use the function `addPipeBlock(x,y)` as a guide for how to write the `shootLaser()` function block. There is one difference however: instead of using the parameters `x` and `y` to determine the laser's position, we want to use the player's position instead. When you make the laser move, try different speeds and directions and find one that works for your game.

Let's look at my solution to this challenge:

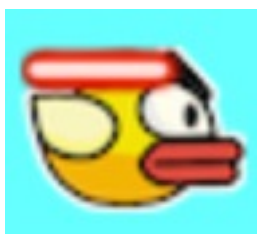
```
function shootLaser() {  
    var laserShot = game.add.sprite(player.x + 45,  
                                     player.y + 15, "laser")  
; // 1  
    lasers.push(laserShot);  
    // 2  
    game.physics.arcade.enable(laserShot);  
    // 3  
    laserShot.body.velocity.x = 300;  
    // 4  
}
```

Each line does one of the four things we want to make happen. Lines 2

and 3 are pretty self-explanatory and can be almost directly copied from the `addPipeBlock(x,y)` function block. Lines 1 and 4 however might require more thought. If yours are different to mine, think about why I might have chosen to write mine as I did.

In the first line, I set the x position to be `player.x + 45` and the y position to be `player.y + 15`. Why did I do this? Why didn't I simply use `player.x` and `player.y`?

The reason for this decision is shown below.



What you see is what happens if you use `player.x` and `player.y` for the x and y positions of the laser. The laser is fired from directly on top of the bird, which doesn't look very good. If instead you use `player.x + 45` and `player.y + 15` for the x and y position of the laser, we get what is shown below,



which I think looks much better.

In the fourth line, I set the laser's `velocity.x` to be 300, and leave its `velocity.y` as 0. This is much more down to personal choice, and as long as you have a positive `velocity.x`, there shouldn't be any

problem with what you have written.

Excellent! We now have a bird that shoots lasers and something for it to shoot at. Now we just need to make the blocks be destroyed when they are shot!

## Step 3: Destroying Pipes and Lasers

This part will require a little more work. The essence of what we are trying to do is rather simple, but the actual execution of that is more complicated. Basically, as often as possible, we want to check some things:

1. Is any laser touching any green block? If it is, destroy the laser, but not the block.
2. Is any laser touching any red block? If it is, destroy both the laser and the block.

Now it makes sense why we kept the red pipe blocks in a different array, away from the green pipe blocks: we want to treat them differently!

Let's consider the first point. We want to check if any laser is touching any green block. Let's make this a bit simpler, let's say that we have only one laser and some green pipes, and we want to know if the laser overlaps with any of the pipes, and do something if it does. We've already seen something very similar! Let's have a look at how we handled the **player** colliding with a pipe:

```
/** This is a demonstration, DO NOT write this! */  
function update() {  
    game.physics.arcade  
        .overlap(player, pipes,  
            gameOver);  
    ... // other code here  
}
```

This code checks if the `player` overlaps with anything in the `pipes` array, and if it does, the function `gameOver` is called. This means that we want a function to be called whenever a laser overlaps with a pipe. Let's pretend we write an empty version of this function now:

```
/** This is a demonstration, DO NOT write this! */  
function destroyLaser() {  
    // Will destroy the laser  
}
```

Great! now assuming we only ever have one laser we could write something like this:

```
/** This is a demonstration, DO NOT write this! */  
function update() {  
    ... // other code here  
    game.physics.arcade  
        .overlap(laser, pipes,
```

```
        destroyLaser);  
    ... // other code here  
}
```

This is a good start, but it doesn't solve our problem. We don't want to only ever allow one laser to be in the air at once, we want lots! That's why we have an array of lasers! We could change it to something like this:

```
/** This is a demonstration, DO NOT write this! */  
game.physics.arcade  
    .overlap(lasers, pipes, // both are arrays  
        destroyLaser);
```

But this code has another problem. How would we know which laser to destroy? Think about it, all this code tells us is that ONE of the lasers overlaps with ONE of the pipe blocks. We don't know which laser to destroy! We need a better solution. We need a way to specifically refer to one of the members of the array of lasers, and to specifically check if that laser collides with any block, and to specifically destroy that one if that is the case. The way to do this is known as *array indexing*.

## Arrays.

We have seen before that an array is a collection of objects. You may remember that we said that the first object has **index 0**, and that the second has **index 1** etc.

---



```
var arr = ["1st", "2nd", "3rd"];  
alert(arr[0]); // alert will say "1st"  
alert(arr[2]); // alert will say "3rd"  
alert(arr.length); // alert will say 3  
alert(arr[3]); // alert will say "undefined"
```

Here, `arr[3]` is `undefined`, because there is no element of the array after `arr[2]`.

What if we don't know how many elements are in the array? How do we access the last element? Well, we saw that the `length` attribute can be used to tell us how many elements are in the array, and that the index of the last element was one less than the length (`arr[2]` worked but `arr[3]` did not). This will work for every array!

```
var arr2 = ["1st", "2nd", ... , "last"]; // How many items?  
alert(arr2.length); // alert will tell us how many.  
alert(arr2[arr2.length]); // alert will say "undefined".  
alert(arr2[arr2.length - 1]); // alert will say "last".
```

Great, now for any array we can access the first element, the last element, and every element in between!

Can't we?

**Challenge:** This is a tricky one. Say you have the array `arr =`

`["1st", "2nd", "3rd"]` . How would you cause an alert to be shown for every element of the array, each one containing that element? i.e. I want 3 alerts, the first one would say "1st", the second would say "2nd", etc. Hint: a `for` loop might make it easier.

**Ultra Hard Challenge:** Can you do it backwards? i.e. 3 alerts, the first of which says "3rd", the second says "2nd" etc.

These challenges are a little tricky, as they require you to remember a lot of things. Let's start with the first one. We'll start with an empty `for` loop, and slowly fill in the gaps:

```
for (1; 2; 3) {  
    4  
}
```

What should happen at number 1: how do we initialise our looping variable? Well, we want to start by printing the element with index 0, so lets initialise our variable `i` to 0. What should happen at number 2: when do we stop looping? We want to stop when we reach index `arr.length` , so we'll check that `i < arr.length` . What should happen at number 3: what do we do in between loops? We want to increase our index to the next value, so we'll increment `i` . What should happen at number 4: what do we want to do each loop? We want an alert with the element with index `i` in it, so we'll call `alert(arr[i])` . The `for` loop then becomes:

```
for (var i = 0; i < arr.length; i++) {  
    alert(arr[i]);  
}
```

Take as long as you need over this, as it is important and we'll need it later.

How do we go backwards then? Well we need to change the first 3 parts of the loop, but the 4th part can remain the same. We want to start with the largest possible index ( `arr.length - 1` ), stop when the index is too small (less than 0) and decrease the index each time.

```
for (var i = arr.length - 1; i >= 0; i--) {  
    alert(arr[i]);  
}
```

Great, we can now move forwards and backwards through an array. But how can we delete an element? When we destroy a laser, we don't want to keep checking if it is colliding with things, we want it gone from the array! Removing something from an array is known as **splicing** the array, and we have a function `splice()` to do this very thing:

```
var arr3 = [0,1,2,3,4];  
alert(arr3); // alert says "[0,1,2,3,4]"  
arr3.splice(3,1); // removes 1 item, starting with index 3  
alert(arr3); // alert says "[0,1,2,4]"
```

```
arr3.splice(0,2); // removes 2 items, starting with index 0
alert(arr3); // alert says "[2,4]"
```

In general, the function `arr.splice(i,n)`, means "remove `n` items from `arr`, starting at index `i`".

Now we have every thing we need to destroy that laser. We want to walk through the array of lasers, check if each one collides with any of the pipe blocks using `overlap`, and if it does, destroy it and delete it from the array. Let's put everything we know together!

```
/** This is a demonstration, DO NOT write this! */
for (var i = 0; i < lasers.length; i++) {
    game.physics.arcade
        .overlap(lasers[i], pipes,
            destroyLaser);
}
```

Excellent! Now we just need to write the `destroyLaser` function... and here we have a problem. `destroyLaser` needs to know which laser to destroy. The way to do that is to pass the laser (or the index of the laser) as an argument. But passing it as an argument means calling the function, and that means writing `destroyLaser(argument)` which we know we can't do in the `overlap` function! We need to be able to write a function that can destroy `lasers[i]`, but also not need to be called to know what `i` even is. If only we could write a function in the same scope as `i`...

As it happens, we can do exactly that.

## Anonymous Functions

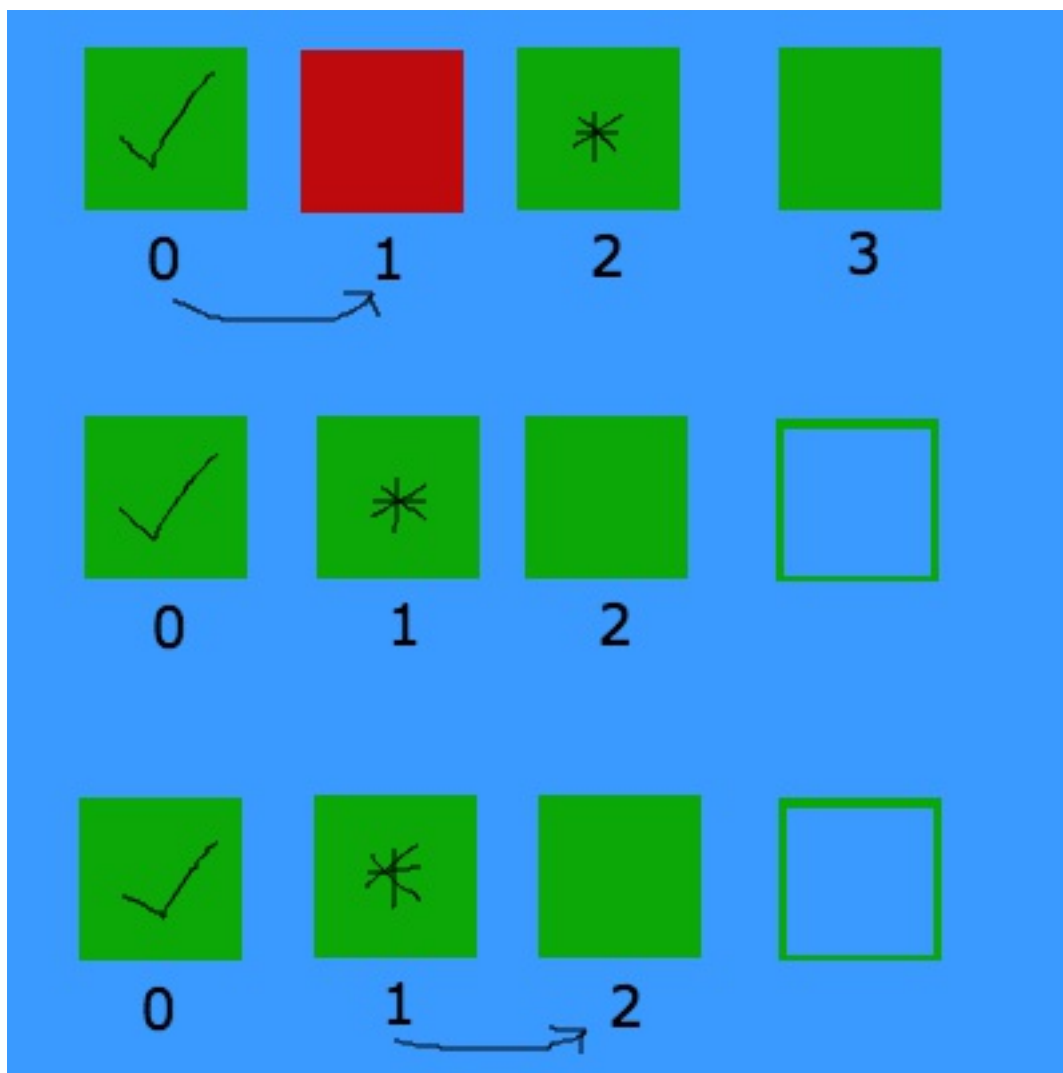
So far, we have seen that every function has a name. We have seen how we can define a function and separately call it, or simply pass that function around without calling it. One way to understand this is to think of a phone number in your contacts. In your contacts, there is a list of names, each of which has their appropriate number stored alongside it. Say you have “Paul” in your contacts. If someone asked you to call Paul, you would find the number, and dial the number and have a nice chat with Paul. But if someone asked for Paul’s number to use later, there wouldn’t be much use dialing Paul’s number and handing the phone over! Whenever you define a function, that is like storing a name with a number to dial if you want to talk to them. When you call a function, that is like dialing the number to actually have the conversation. When you pass the function to another function (like you pass `gameOver` to the `overlap()` function), that is like giving someone Paul’s number to dial later.

But what if you saw a building burning down? You would probably get your phone and dial 999. This number isn’t saved in your contacts, there is no name attached to it, because you wouldn’t need this number except in very specific circumstances. This is like an anonymous function. An anonymous function has no name, and is defined in the place it is needed, because it isn’t going to be used anywhere else. We will explore their syntax in the solution to our problem:

```
/** This is a demonstration, DO NOT write this! */  
for (var i = 0; i < lasers.length; i++) {  
    game.physics.arcade  
        .overlap(lasers[i], pipes,  
        function () {  
            lasers[i].destroy();  
            lasers.splice(i,1);  
        });  
}
```

Notice how this function has no name, there is nothing between **function** and the parentheses. This means that it can't be called from anywhere else, the only time this function can be called is when a laser overlaps with a pipe. What does the function do? It destroys the right laser and removes it from the array. Perfect! There's just one problem.

Imagine you are the **for** loop, and that the item with index 1 is overlapping. You check index 0 and it doesn't overlap. You check index 1 and it does overlap, so you remove the item, meaning the item with index 2 now has index 1, index 3 is now index 2 etc. Then you check index 2... but wait! You missed an item!



Look what happens to the starred element above. Initially it is index 2. When the element at index 1 is removed, it becomes index 1. But it is never checked, because we move on to index 2! Even worse, what happens when we try to check index 3? There's nothing there!

The solution to this is to iterate backwards through the array. This is because when we splice an array and remove index  $n$ , only the items with indices bigger than  $n$  are affected, and we have already checked those.

Thus our final solution, to be put in the `update()` function block, is:

```
for (var i = lasers.length - 1; i >= 0; i--) {
```

```
game.physics.arcade
    .overlap(lasers[i], pipes,
function () {
    lasers[i].destroy();
    lasers.splice(i,1);
});
}
```

This is worth remembering: **If you're going to remove something from an array, iterate through it backwards!**

Now let's look at the second of our two aims. Do you remember that? We've only done half this step! (Don't worry though, after this we've done the hardest part!)

We have succeeded in making the green blocks destroy the lasers. Now we want the red blocks to both destroy the lasers and be destroyed by them. When we were considering the green pipes, it was fine to check for collisions with just the `pipes` array, because it didn't matter which pipe block it had collided with. When we consider the red pipes, however, this will not do, as we need to know the specific red pipe block that the specific laser has collided with!

**Challenge:** *Can you see what we need to do? We need a way to iterate through both arrays, checking for overlaps with each pair of objects. Then if we have an overlap, we need to remove both the specific laser and the specific pipe block. Remember which*



*direction to iterate in! Hint: use an anonymous function to splice the arrays and destroy the sprites.*

A correct solution is shown below. Remember this must be added to our `update()` function block.

```
for (var i = redPipes.length - 1; i >= 0; i--) {  
  for (var j = lasers.length - 1; j >= 0; j--) {  
    game.physics.arcade.overlap(redPipes[i],  
                                lasers[j],  
                                function () {  
      lasers[j].destroy();  
      lasers.splice(j, 1);  
      redPipes[i].destroy();  
      redPipes.splice(i, 1);  
    })  
  }  
}
```

Because we need to be able to access the specific items in each array, we need to iterate through both arrays, each with a `for` loop! We then need another anonymous function to handle what happens when they overlap.

Excellent! We now have a working game again. If you wish you can leave it here. However, if you want to add even more excitement to the game by limiting the player's ammo, then follow me to Step 4! Most of

the hard work has been done now, it's just reusing what we've already learnt. That said, it won't be too easy!

## Step 4: Limit the Shots

Before, there was no limit on how many times the player could fire a laser. We're about to change that. The way to do this is very similar to the way we keep track of the score. Let's add a global variable to keep track of the score, and one for the label it will be displayed in.

```
/******  
// These two we've already got  
var score = 0;  
var labelScore;  
/******  
  
// These two we need to add  
var shots = 10;  
var labelShots;
```

I decided to give the player 10 shots initially, but you can choose a different number if you wish. Try things out and see!

At the moment, the score is being displayed at the top left of the screen. If we were to put the number of shots left up there too with no indication of which was which, it may get confusing. Thus we need to add a little more text to the screen. In the `create()` function block, add

the following lines:

```
game.add.text(20,20, "Score:");
game.add.text(20,50, "Shots:");
// Delete the old initialisation of labelScore
labelScore = game.add.text(120, 20, "0");
labelShots = game.add.text(120, 50, "10");
```

We now need a way to change the number of shots we have left. This is very similar to the way we change the score using the `changeScore()` function, except we want to both increase and decrease the number of shots we have left.

**Challenge:** Write a function, `changeShots(amount)`, that works exactly like `changeScore()` except that the number of shots the player has is increased by `amount`.

The function should be written something like:

```
function changeShots(amount) {
    shots += amount;
    labelShots.setText(shots.toString());
}
```

Now we just need to think about when we want the number of shots to change, and what should happen if we run out of shots.

**Challenge:** Can you work it out? There are two things to change, we need to change the number of shots left when something happens, and we need something to happen (or something to stop happening) when we run out of shots.

Obviously, when we fire a shot, the number of shots we have left should decrease. Let's make that change now:

```
function shootLaser() {  
    ... // previous code here  
    changeShots(-1);  
}
```

Here, whenever we fire a shot, we increase the number of shots by -1, which is the same as decreasing it by 1.

Now, what should happen when we run out of shots? We shouldn't be able to fire anymore. The way I have phrased the question makes it seem like you need to wait for `shots` to be equal to 0 and then stop the X button from firing, or something along those lines. A better solution is more obvious if we slightly rephrase the question: "What should happen when we have enough shots to fire and we try to? What should happen when we don't have enough shots to fire and we try to?" This makes it clearer that the correct response is to, before firing, check whether we have enough shots left. If we do, then fire and decrease the number. If we do not, simply do nothing.

```
function shootLaser() {  
  if (shots > 0) {  
    ... // previous code here  
    changeShots(-1);  
  }  
}
```

This shows that sometimes, the way we phrase a question in our heads can hide easy solutions from ourselves. Whenever you are not sure how best to do something, think about what you are trying to do in as many different ways as possible. It may be that one of those rephrasings leads to a much simpler and neater solution than the one you came up with first.

Now that we have murdered our poor bird by not allowing it to pass the sixth red pipe it comes across, let's go some way to saving it by spawning some more ammo!

## Step 5: Generate Ammo

First off, let's get the asset for the ammo loaded into the game. I made

and used the following image:  which can be found here

(<https://github.com/rjmcfl/webapp-flappy/raw/master/assets/ammo.png>)

**Challenge:** Load the asset as usual, this time calling it `ammo`.

The code to add to the `preload()` function block is

```
game.load.image("ammo", "../assets/ammo.png");
```

Because we need a way to keep track of these ammo drops, let's add another array to the list at the top:

```
var ammoDrops = [];
```

A simple way (probably the simplest) to add ammunition drops to the game is to replace some of the spawning pipes with them. This means that when the `chooseGen()` function is called, it will generate a green pipe, a red pipe, or an ammunition drop. Let's begin by making a new function, one that will eventually generate ammo on demand:

```
function generateAmmo() {  
  
  
  
  
  
}
```

Generating ammo is very similar to generating a pipe block, except with one difference: we want the height of the ammo to be random! Thus we want this function to do 5 things:

1. Select a random height for the ammo.
2. Draw the image at the correct position.
3. Keep track of the ammo by adding it to the appropriate array.

4. Enable physics for the ammo.
5. Make the ammo move.

**Challenge:** Fill in the function block so that it does these 5 things. I decided to make the ammo line up with the pipe blocks, but you may choose not to do that. Just make sure it ends up on the screen! It's probably a good idea to make it travel at the same speed as the pipe blocks too.

My version of this function looks like this:

```
function generateAmmo() {  
    var height = game.rnd.integerInRange(0, 7);  
    var ammo = game.add.sprite(800,height*50, "ammo");  
    ammoDrops.push(ammo);  
    game.physics.arcade.enable(ammo);  
    ammo.body.velocity.x = -200;  
}
```

Again, each line does one of the 5 things we required. If yours looks different, think about why I've chosen to write it how I have, and if these differences matter at all!

We now have a way to make ammo spawn, we just need to actually call that function now! Let's update our `chooseGen()` function to do what we want it to now. Replace the old one with this new one!

```
function chooseGen() {
    var type = game.rnd.integerInRange(1, 6);
    if (type == 1) {
        generateRedPipe();
    } else if (type == 2) {
        generateAmmo();
    } else {
        generatePipe();
    }
    chooseScore();
}
```

Now, I have chosen to make the red pipe spawn with probability 1 in 6, the ammo to spawn with probability 1 in 6, and the the green pipe to spawn with probability 4 in 6 (also 2 in 3). If you want different probabilities, you can change the numbers above, in a way similar to how we chose the probabilities when we only had the two pipes to consider.

**Challenge:** How would you make the ammo spawn with probability 1 in 6, the red pipe spawn with probability 2 in 6 (or 1 in 3), and the green pipe spawn the rest of the time (i.e. 3 in 6 or 1 in 2)?

The code for this is shown below:

```
function chooseGen() {
    var type = game.rnd.integerInRange(1, 6);
```



```
if (type == 1) {  
    generateAmmo();  
} else if (type == 2 || type == 3) {  
    generateRedPipe();  
} else {  
    generatePipe();  
}  
chooseScore();  
}
```

Any other code that means the same thing is also correct. Note the use of `||` to mean “or”. We saw this when making the gap for green pipes.

## Code Reuse: Reprise

DO you remember that a long time ago I told you to move the call to `changeScore()` out of the two functions that generate pipes, and into the `chooseGen()` function? Be honest with yourself, if you hadn’t done that, would you have remembered to add that call to the `generateAmmo()` function? Would you have even thought about it if I hadn’t mentioned it now? Remember, make it as easy as possible for your future self to do things right!

Great job! We are almost there now! We just have one more thing to do: make the bird pick up that ammo!

## Step 6: Pick Up Ammo

This is very similar to how we made the pipes destroy the lasers. We simply need to check if the player is overlapping with any of the ammo drops, and delete it if it is. We also need to increase the number of shots left if this is the case. I reckon you can do this yourself! This will be your final challenge, putting together a lot of the things we have learned.

***Final Challenge:*** Add some code in the right place that detects when the player overlaps with an ammo drop. We need to be able to refer to a specific ammo drop, and we need to delete it from the array if there is overlap. Also remember to increase the number of shots left by some number (I would suggest 2). Avoid all the traps!

This final piece of code is given below. Add this to `update()` if you haven't already:

```
for (var i = ammoDrops.length - 1; i >= 0; i--) {  
    game.physics.arcade.overlap(player, ammoDrops[i], function () {  
        ammoDrops[i].destroy();  
        ammoDrops.splice(i, 1);  
        changeShots(2);  
    })  
}
```

Did you remember everything important? Here's a checklist of things to make sure you have done:

1. Iterate through the array.
2. Iterate **backwards** through the array!
3. Check for overlap with `ammoDrops[i]` , not `ammoDrops` .
4. Use an anonymous function correctly.

Well done! You have made quite a substantial addition to the game, and hopefully gameplay has been made a little more exciting. You have seen how to use a lot of different things and avoided a lot of traps on the way, and that deserves congratulations. I hope you have enjoyed this tutorial, and that you keep improving both your game and your coding skill!

## Appendix: The Code

```
// the Game object used by the phaser.io library
var stateActions = { preload: preload, create: create, update: update };

// Phaser parameters:
// - game width
// - game height
// - renderer (go for Phaser.AUTO)
// - element where the game will be drawn ('game')
// - actions on the game state (or null for nothing)
var game = new Phaser.Game(790, 400, Phaser.AUTO, 'game', stateActions);
var score = 0;
var labelScore;
```

```
var player;
var pipes = [];
var redPipes = [];
var lasers = [];
var shots = 10;
var labelShots;
var ammoDrops = [];

/*
 * Loads all resources for the game and gives them names.
 */
function preload() {
    game.load.image("playerImg", "../assets/flappy.png");
    game.load.audio("score", "../assets/point.ogg");
    game.load.image("pipe", "../assets/pipe.png");
    game.load.image("pipeRed", "../assets/pipeRed.png");
    game.load.image("laser", "../assets/laser.png");
    game.load.image("ammo", "../assets/ammo.png");
}

/*
 * Initialises the game. This function is only called once.
 */
function create() {
    game.stage.setBackgroundColor("#66ffff");
    game.physics.startSystem(Phaser.Physics.ARCADE);

    player = game.add.sprite(20, 80, "playerImg");
```

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

```
player.body.gravity.y = 400;
```

```
game.input.keyboard
```

```
    .addKey(Phaser.Keyboard.SPACEBAR)
```

```
    .onDown.add(playerJump);
```

```
game.input.keyboard
```

```
    .addKey(Phaser.Keyboard.X)
```

```
    .onDown.add(shootLaser);
```

```
game.add.text(20,20, "Score:");
```

```
game.add.text(20,50, "Shots:");
```

```
labelScore = game.add.text(120, 20, "0");
```

```
labelShots = game.add.text(120, 50, "10");
```

```
var pipeInterval = 1.75;
```

```
game.time.events
```

```
    .loop(pipeInterval * Phaser.Timer.SECOND,
```

```
    chooseGen);
```

```
}
```

```
function update() {
```

```
    game.physics.arcade
```

```
        .overlap(player, pipes,
```

```
        gameOver);
```

```
game.physics.arcade.
```

```
    overlap(player, redPipes,  
    gameOver);
```

```
    for (var j = lasers.length - 1; j >= 0; j--) {  
        game.physics.arcade.overlap(lasers[j], pipes, funct  
ion () {  
            lasers[j].destroy();  
            lasers.splice(j,1);  
        })  
    }
```

```
    for (var i = redPipes.length - 1; i >=0; i--) {  
        for (var j = lasers.length - 1; j >= 0; j--) {  
            game.physics.arcade.overlap(redPipes[i],  
                                         lasers[j], function  
( ) {  
                lasers[j].destroy();  
                lasers.splice(j,1);  
                redPipes[i].destroy();  
                redPipes.splice(i,1);  
            })  
        }  
    }
```

```
    for (var i = ammoDrops.length - 1; i >= 0; i--) {  
        game.physics.arcade.overlap(player, ammoDrops[i], f  
unction () {
```

```
        ammoDrops[i].destroy();
        ammoDrops.splice(i,1);
        changeShots(2);
    })
}
}

function gameOver(){
    game.destroy();
}

function changeScore() {
    score ++;
    labelScore.setText(score.toString());
}

function changeShots(amount) {
    shots += amount;
    labelShots.setText(shots.toString());
}

function playerJump() {
    player.body.velocity.y = -200;
}

function addPipeBlock(x, y) {
    var pipeBlock = game.add.sprite(x,y,"pipe");
    pipes.push(pipeBlock);
}
```

```
game.physics.arcade.enable(pipeBlock);
pipeBlock.body.velocity.x = -200;
}

function generatePipe() {
    var gap = game.rnd.integerInRange(1, 5);
    for (var count=0; count < 8; count++) {
        if (count != gap && count != gap+1) {
            addPipeBlock(800, count*50);
        }
    }
}
```

```
function addRedPipeBlock(x,y) {
    var redPipeBlock = game.add.sprite(x,y,"pipeRed");
    redPipes.push(redPipeBlock);
    game.physics.arcade.enable(redPipeBlock);
    redPipeBlock.body.velocity.x = -200;
}
```

```
function generateRedPipe() {
    for (var count=0; count < 8; count++) {
        addRedPipeBlock(800, count*50);
    }
}
```

```
function generateAmmo() {
    var height = game.rnd.integerInRange(0, 7);
```



```
var ammo = game.add.sprite(800,height*50, "ammo");
game.physics.arcade.enable(ammo);
ammo.body.velocity.x = -200;
ammoDrops.push(ammo);
}

function chooseGen() {
    var type = game.rnd.integerInRange(1, 6);
    if (type == 1) {
        generateRedPipe();
    } else if (type == 2) {
        generateAmmo();
    } else {
        generatePipe();
    }
    changeScore();
}

function shootLaser() {
    if (shots > 0) {
        var laserShot = game.add.sprite(player.x + 45,
                                         player.y + 15, "laser");
        lasers.push(laserShot);
        game.physics.arcade.enable(laserShot);
        laserShot.body.velocity.x = 300;
        changeShots(-1);
    }
}
```

}