**Phaser Tutorial**

What is Phaser?

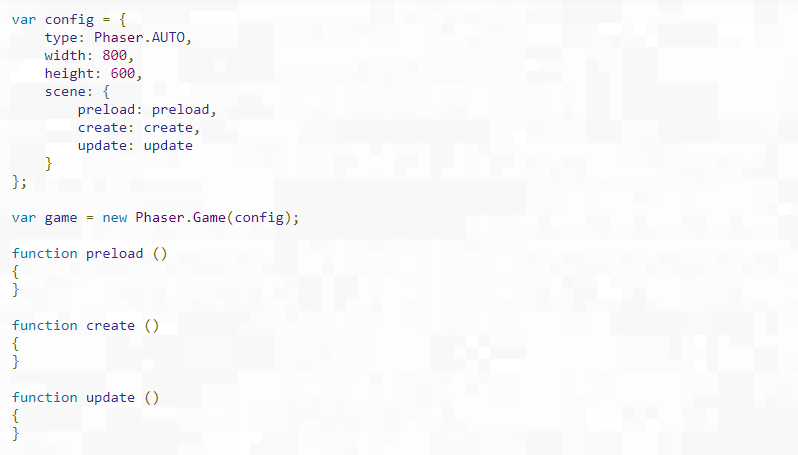
Phaser is an HTML5 game framework which aims to help developers make powerful, cross-browser HTML5 games really quickly and, unlike some others, has solely been built to work with the mobile browsers. The only browser requirement is the support of the canvas tag. It also borrows a lot from Flixel.

# PART 1: Setting Things Up

Download the assets that go with the tutorial. They are available with the lab assignment.

Create a new file and name it index.html. It needs the basic HTML tags(html, head, body). It also needs a script tag to link the phaser.min.js file. Make sure to use Brackets when testing the code, using the Live Preview.

In your script tag (for your code), type the following code:



The config object is how you configure your Phaser game. Many options can be placed in this object. As you expand your Phaser knowledge, you will encounter more of them. In this tutorial, we are just going to set the renderer, dimensions, and a default Scene.

An instance of a Phaser.Game object is assigned to a local variable called 'game' and the config object is passed to it. Calling it 'game' is a common practice, but not a requirement, and this is what you will find in the Phaser examples.

The type property can be either Phaser.CANVAS, Phaser.WEBGL or Phaser.AUTO. This is the rendering context you want to use for your game. The recommended value is Phaser.AUTO which automatically tries to use WebGL. But if the browser or device doesn’t support it, it will fall back to Canvas. The canvas element that Phaser creates will be simply appended to the document at the point the script was called. But you can also specify a parent container in the game config should you wish.

The width and the height properties set the size of the canvas element that Phase will create; In this case, 800 by 600 pixels. Your game world can be any size you like.

The scene property of the configuration object will be covered in more detail further on in this tutorial.

# PART 2: Loading Assets

Now, let’s load the assets we need for our game. You do this by putting calls to the Phaser Loader inside of a Scene function called preload. Phaser will automatically look for this function when it starts and load anything defined within it. Currently, the preload function is empty. Change it as shown below.



This will load in 5 assets: 4 images and a spritesheet. The first parameter is known as the asset key (‘sky’ or ‘bomb’). This string is a link to the loaded asset and is what we will use in the code when creating game objects.

We are loading the assets needed for the game: 3 images and 1 sprite sheet.

In order to display one of the images we have loaded, place the following code inside the create function.

Now, let’s create a sprite by adding the following code in the create function:

Code that creates an image for the background using the loaded asset named sky.

If you run the page, this is what you should see, a game screen with a blue backdrop covering it:



The values 400 and 300 are the x and y coordinates of the image. Why 400 and 300? In Phaser3, all game objects are positioned based on their center by default. The background image is 800 by 600 pixels in size. So if we were to display it centered at (0,0) we would only see the bottom right corner of it. If we display it at (400,300), then we see the whole image.

You can use setOrigin to change this. For example, the code this.add.image(0,0,’sky’).setOrigin(0,0) would reset the drawing position of the image to the top left.

The order in which game objects are rendered in the display matches the order in which you create them. So, if you wish to place a star above the background, you would need to ensure that it was added as an image second, after the sky image.

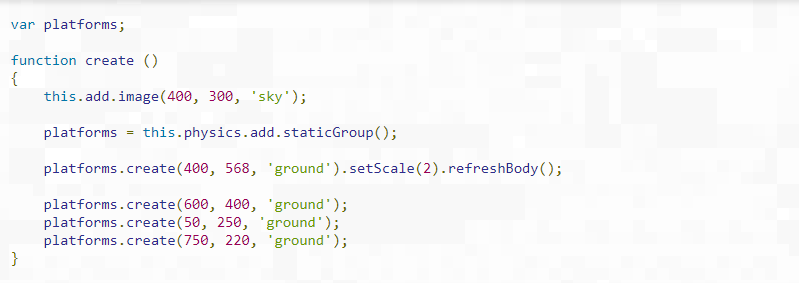


# PART 3: World Building

Under the hood this.add.image is creating a new Image Game Object and adding it to the current Scenes display list. This list is where all your Game Objects live. You could position the image anywhere and Phaser will not mind. Of course, if it’s outside of the (0,0) to (800,600) region then you are not going to be able to see it (off screen). But it will still exist within the Scene.

The Scene itself has no fixed size and extends infinitely in all directions. The Camera system controls your view into the Scene and you can move and zoom the active camera as required. You can also create new cameras for other views into the Scene. This topic is beyond the scope of this tutorial.

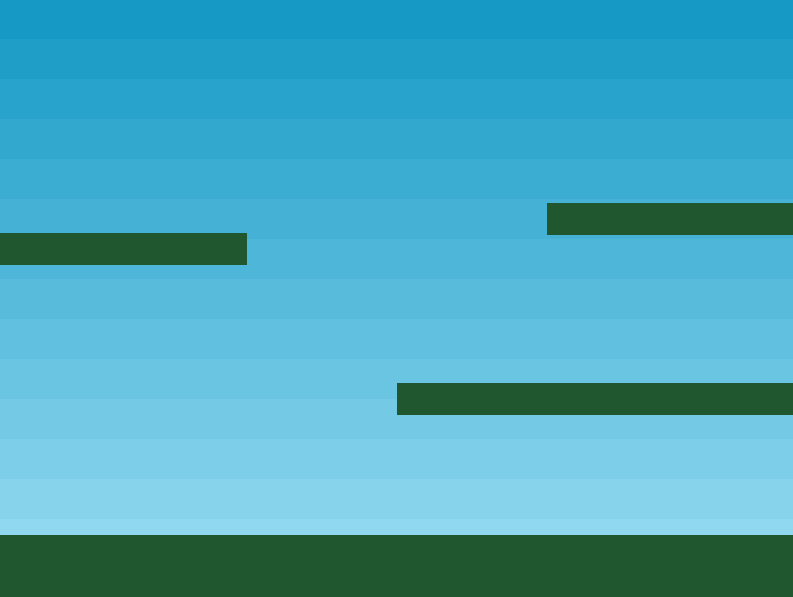
For now, let’s build up the Scene by adding a background image and some platforms. Here is the updated create function:



Glancing quickly at the code, you will see a call to this.physics. This means we are using the Arcade Physics system, but before we can do that we need to add it to our game config object to tell Phaser our game requires it. So, let’s update that to include the physics support:



The new addition is the physics property. With this code in place, if you run it, you should see a game-like scene:



We now have a background and some platforms. But how are those platforms working?

# PART 4: The Platforms

We just added a bunch of code to our create function that deserves a more detailed explanation.

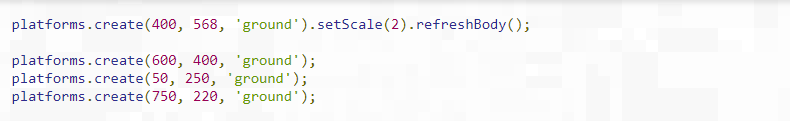
platforms = this.physics.add.staticGroup();

This creates a new static physics group and assigns it to the local variable platforms. In Arcade Physics, there are two types of physics bodies: dynamic and static. A dynamic body is one that can move around via forces such as velocity or acceleration. It can bounce and collide with other objects and that collision is influenced by the mass of the body and other elements.

In stark contrast, a static body simply has a position and a size. It is not touched by gravity, you cannot set velocity on it and when something collides with it, it never moves. This is perfect for the ground and platforms that we are going to let the player run around on.

What is a Group? As the name implies they allow you to group together similar objects and control them all as one single unit. You can also check for collision between Groups and other game objects. Groups are capable of creating their own game objects via functions like create. A Physics group will automatically create physics enabled children, saving some work in the process.

With our platforms group created, we can use it to create the platforms (don’t write this code again as it only serves to help understand the explanation):



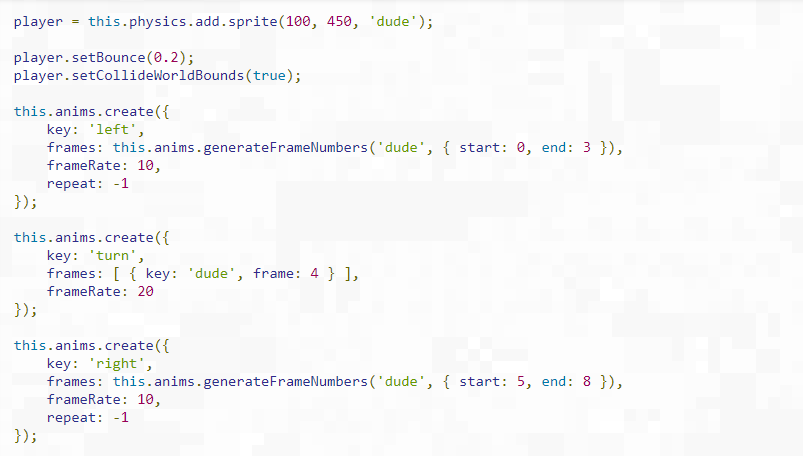
The first line of code adds a new ground image at (400,568). The actual ground image is a simple green rectangle that is 400 by 32 pixels in size. The problem is that, for the ground, we want the platform to span the full width of the game. If not, the player will just drop off the sides. To do that, we scale it by 2 with the function setScale(2). It is now 800 by 64 in size, which is perfect for our needs. The call to refreshBody() is required because we have scaled a static physics body, so we have to tell the physics world about the changes we made.

The ground is scaled and in place. It is time for the other platforms. The process is the same, only we don’t need to scale these platforms, as they are the right size already.

Now, let’s move on to the player.

# PART 5: Ready Player One

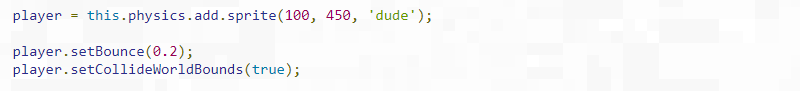
Create a new variable called player (where you declared the game and platforms variables) and add the following code to the create function.



There are two separate things going on here: the creation of a Physics Sprite and the creation of some animations that it can use.

## Physics Sprite

The first part of the code creates the sprite:



This creates a new sprite called player, positioned at (100, 450). The sprite was created via the Physics Game Object Factory (this.physics. add) which means it has a Dynamic Physics body by default.

After creating the sprite, it is given a slight bounce value of 2. This means when it lands after jumping, it will slightly bounce. The sprite is then set to collide with the world bounds. The bounds, by default, are on the outside of the game dimensions. As we set the game to be 800 by 600, then the player won’t be able to run off the edges of the screen or jump through the top.

## Animations

If you glance back at the preload function, you will see that ‘dude’ was loaded as a sprite sheet, not an image. That is because it contains animation frames. This is what the full sprite sheet looks like:



There are 9 frames in total, 4 for running left, 1 for facing the camera, and 4 for running right. Phaser supports flipping sprites to save animation frames, but for this tutorial, we’ll use different images.

We define two animations called left and right. The left animation uses frames 0, 1, 2, and 3 and runs at 10 frames per second. The ‘repeat: -1’ tells the animation to loop.

This is our standard run-cycle and we repeat it for running in the opposite direction, using the key right and a final one for turn.

# PART 6: The Body and Velocity (physics)

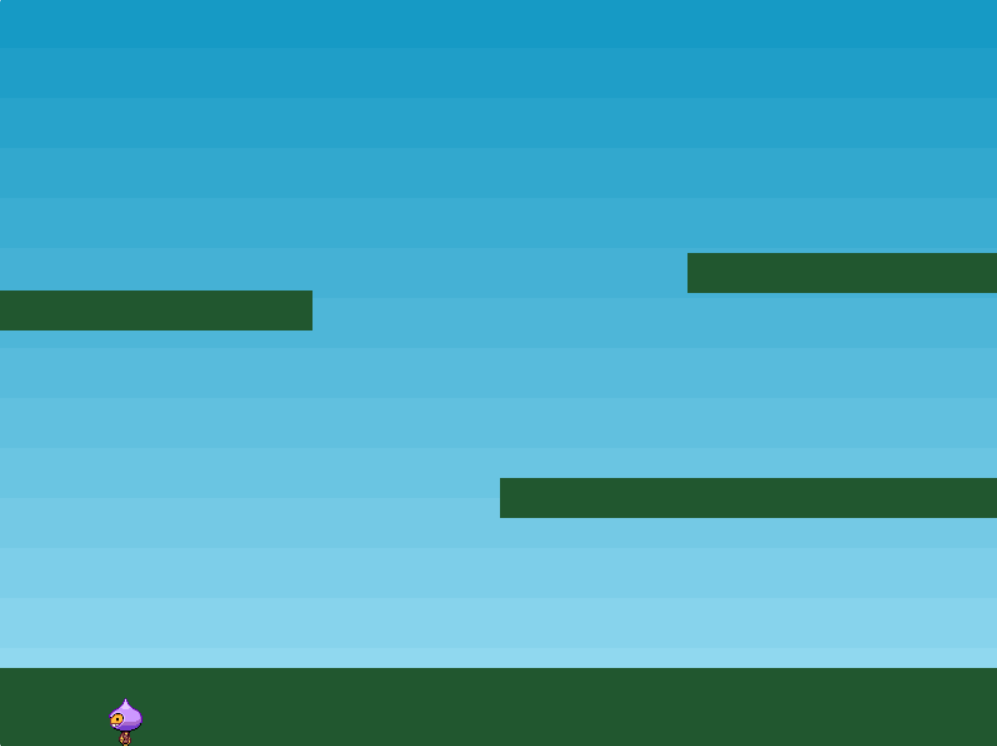
Phaser has support for a variety of different physics systems, each acting as a plugin available to any Phaser scene. It ships with Arcade Physics, Impact Physics and Matter.js Physics. For this tutorial, we will be using the Arcade Physics system. It is simple and light-weight.

When a Physics Sprite is created, it is given a body property, which is a reference to its Arcade Physics Body. This represents the sprite as a physical body in Phaser’s Arcade Physics engine. The body object has a lot of properties and methods that we can play with.

For example, to simulate the effects of gravity on a sprite, it is as simple as writing:

Code to add gravity to the player's body.

This is an arbitrary value, but logically, the higher the value, the heavier your object feels and the quicker it falls. If you add this to your code and run it, you will see that the player falls down without stopping, completely ignoring the ground we created earlier.

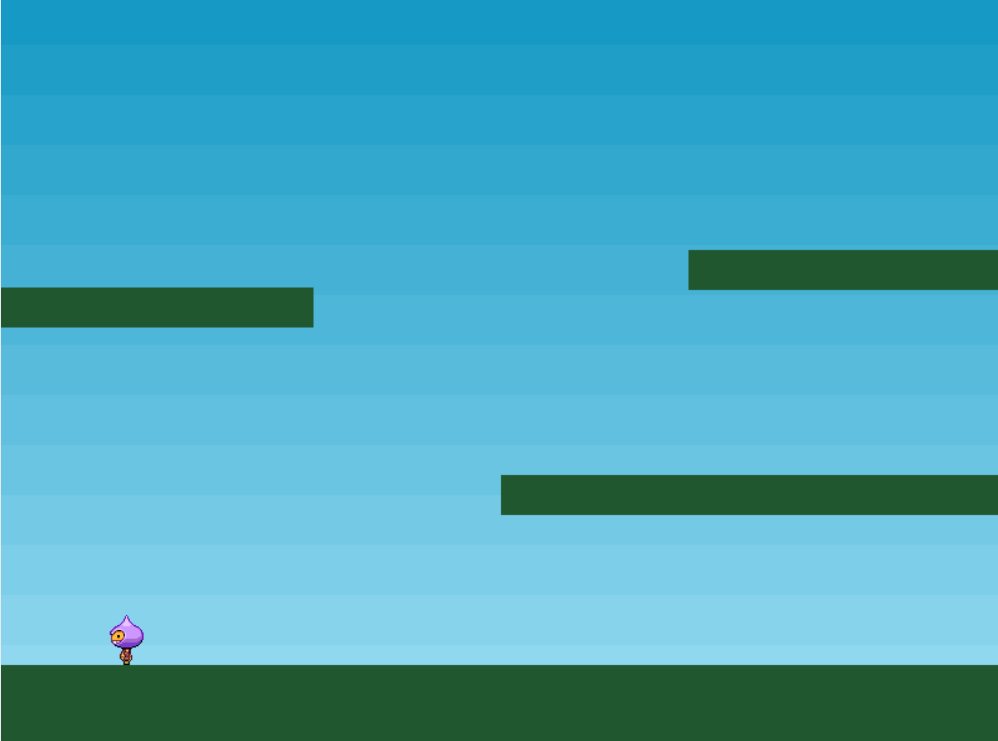


The reason for this is that we're not yet testing for collision between the ground and the player. We already told Phaser that our ground and platforms would be static bodies. Had we not done that, and created dynamic ones instead, then when the player collided with them, it would stop for a moment and then everything would have collapsed. This is because unless told otherwise, the ground sprite is a moving physical object and when the player hits it, the resulting force of the collision is applied to the ground, therefore, the two bodies exchange their velocities and ground starts falling as well.

In order to allow the player to collide with the platforms, we can create a Collider object. This object monitors two physics objects (which can include Groups) and checks for collisions or overlaps between them. If that occurs, it can then optionally invoke your own callback. In this tutorial, we don’t require that. Add this line of code in the create function.



The Collider is the one that performs the magic. It takes two objects and tests for collision and performs separation against them. In this case, we are giving it the player sprite and the platforms Group. It’s clever enough to run collision against all Group members, so this one call will collide against the ground and all platforms. The result is a firm platform that doesn’t collapse.



# PART 6: Controlling the player with the keyboard

Colliding is all good and well, but we really need the player to move. You would probably think of heading to the documentation and searching about how to add an event listener, but that is not necessary here. Phaser has a built-in Keyboard manager and one of the benefits of using that is this handy little function. Create a cursors variable (with the platforms, game, and player ones) and add this code in the create function:

Code that creates a cursor object so that we can use keyboard events in the game.

This populates the cursors object with four properties: up, down, left, right, that are all instances of Key objects. Then all we need to do is poll these in our update loop:

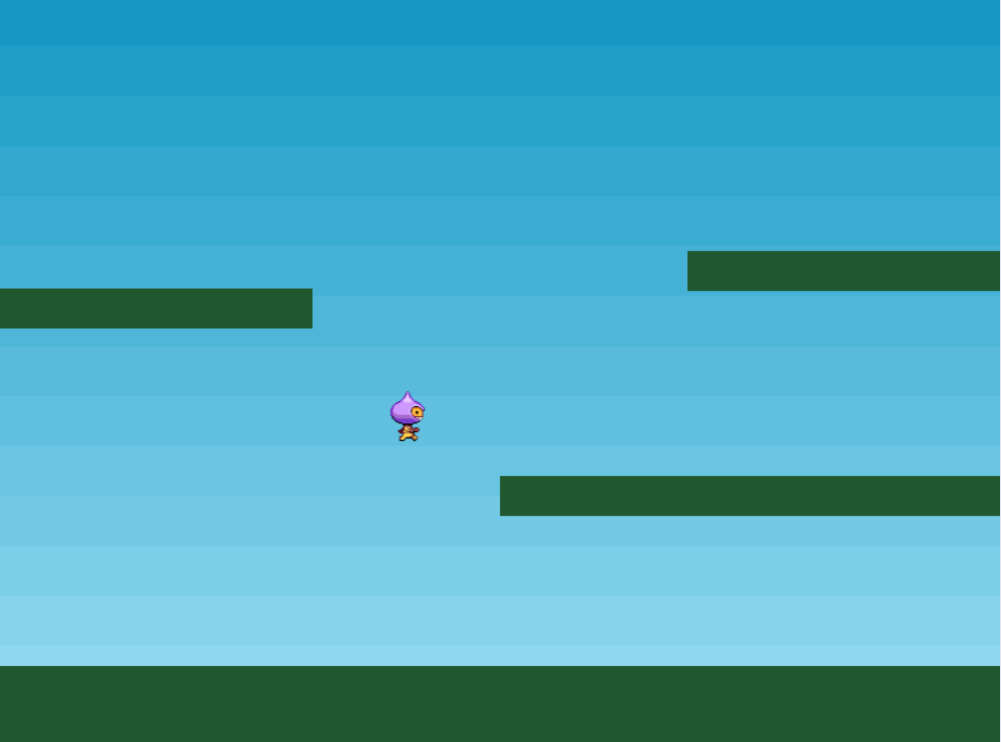


Although we've added a lot of code it should all be pretty readable. The first thing we do is check to see if the left cursor key is held down. If it is, we apply a negative horizontal velocity and start the 'left' running animation. If they are holding down 'right' instead, we literally do the opposite. By clearing the velocity and setting it in this manner, every frame, it creates a 'stop-start' style of movement.

The player sprite will move only when a key is held down and stop immediately they are not. Phaser also allows you to create more complex motions, with momentum and acceleration, but this gives us the effect we need for this game. The final part of the key check sets the frame to 4 if no key is held down.

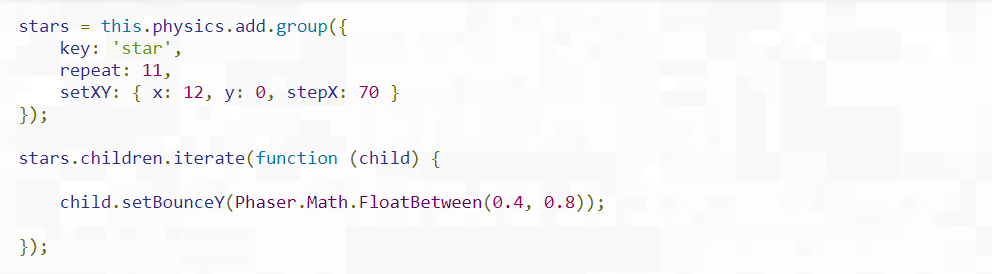
The final part of the code adds the ability to jump. The up cursor is our jump key and we test if that is down. However, we also test if the player is touching the floor, otherwise they could jump while in mid-air.

If both of these conditions are met, we apply a vertical velocity of 330 px/sec. The player will fall to the ground automatically because of the gravity value we applied to it. With the controls in place, we now have a game world we can explore. Run your code and try tweaking values like the 330 for the jump to lower and higher values to see the effect it will have.



# PART 8: Add Stars to the game

It's time to give our little game a purpose. Let's drop a sprinkling of stars into the scene and allow the player to collect them. To achieve this, we'll create a new Group called 'stars' and populate it. In our create function we add the following code:



The process is similar to when we created the platforms Group. As we need the stars to move and bounce, we create a dynamic physics group instead of a static one.

Groups are able to take configuration objects to aid in their setup. In this case, the group configuration object has 3 parts. First, it sets the texture key to be the star image. This means that any children created as a result of the config object will all be given the star texture by default. The it sets the repeat value to be 11. Because it creates 1 child automatically, repeating 11 times means we will get 12 in total, which is what we need for our game.

The final part is setXY. This is used to set the position of the 12 children the Group creates. Each child will be placed starting at (12,0) and with an x step of 70. This means that the first child will be positioned at (12,0), the second one at (82,0), the third one at (152,0) and so on. The step values are a really handy way of spacing out a Group’s children during creation.

The next piece of code iterates all children in the Group and gives them a random Y bounce value between 0.4 and 0.8. The bounce range is between 0 (no bounce at all) and 1 (full bounce). Because the stars are spawned at y=0, gravity is going to pull them down until they collide with the platforms or ground. The bounce value means they will randomly bounce back up again, until finally settling to rest.

If we were to run to code like it is ow, the stars would fall through the bottom of the game and out of sight. To stop that, we need to check for their collision against the platforms. We can use another Collider object to do this.

Line of code than enables collisions between the stars and the platforms.

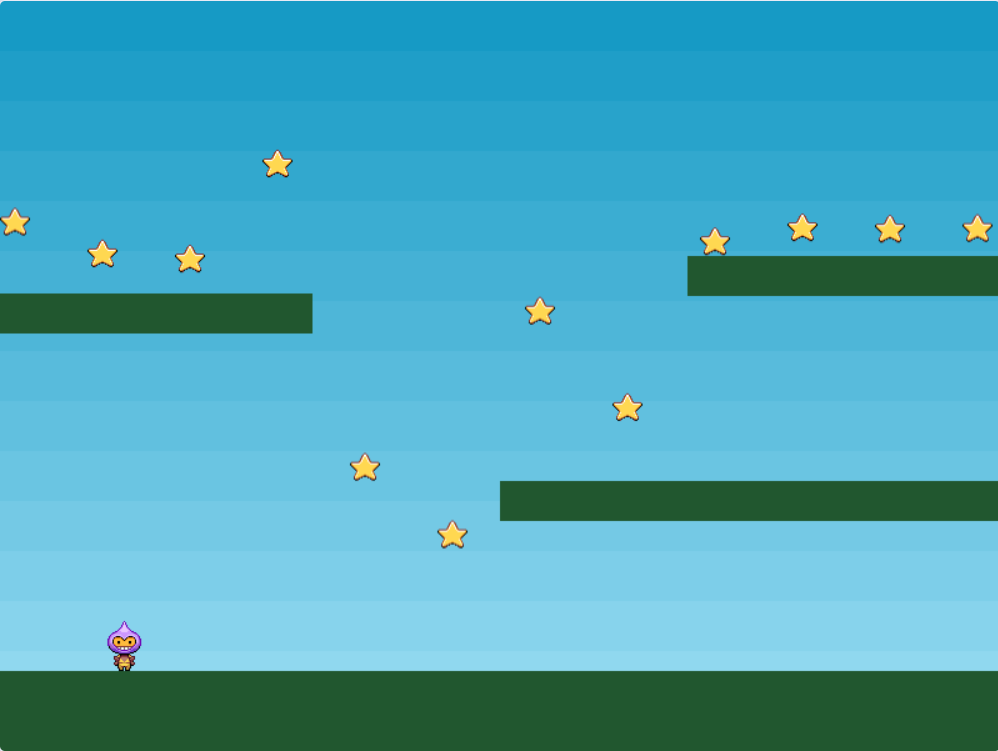
As well as doing this, we will also check if the player overlaps with a star or not.

Line of code that checks for overlaps between the player and the stars and calls collectStar when that happens.

This tells Phaser to check for an overlap between the player and any star in the stars Group. If found, then they are passed to the collectStar function:



Quite simply the star has its physics body disabled and its parent Game Object is made inactive and invisible, which removes it from display. Running the game now gives us a player that can dash about, jumping, bouncing off the platforms and collecting the stars that fall from above. Not bad for a few lines of hopefully mostly quite readable code.



# PART 9: A Score to Settle

There are two final touches we are going to add to our game: an enemy to avoid that can kill the player and a score when you collect the stars.

Let’s start with the score. To do this we'll make use of a Text Game Object. Here we create two new variables, one to hold the actual score and the text object itself:

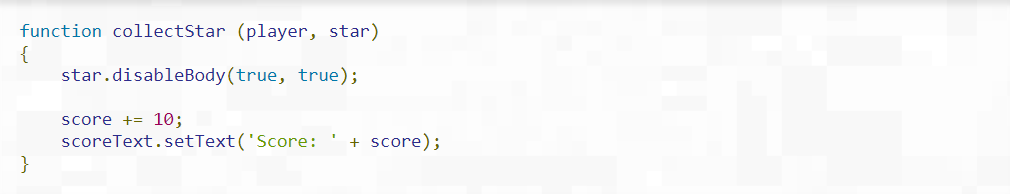
Code initializing the score and the score display.

The scoreText is set-up in the create function:

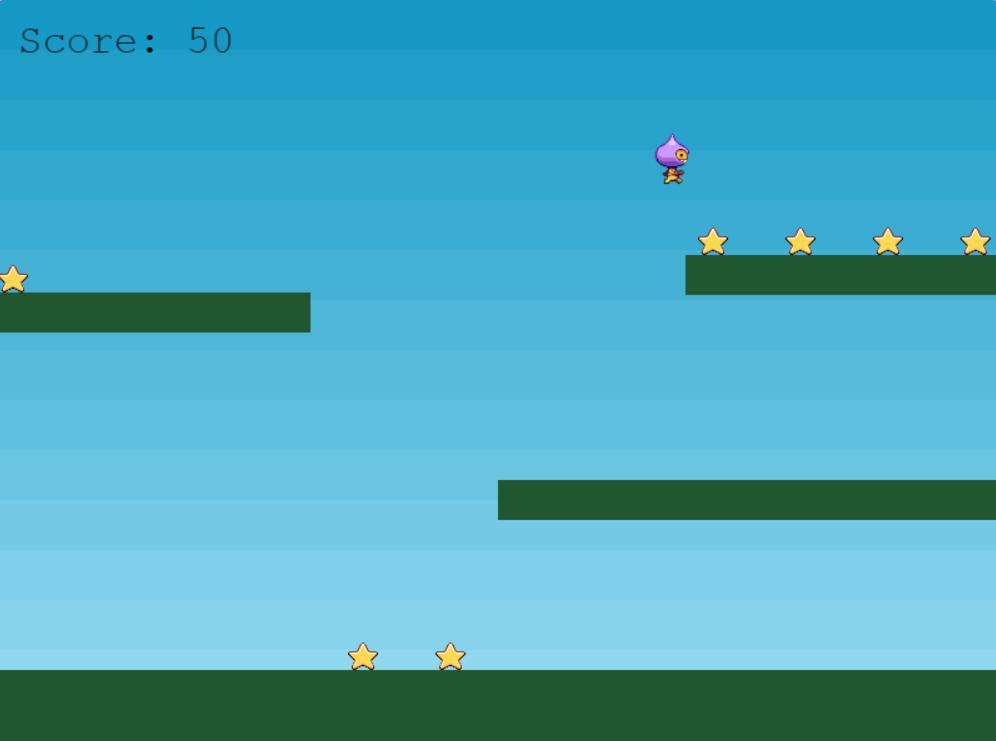
Line of code that creates the score Text.

(16,16) is the coordinate to display the text at. 'score: 0' is the default string to display and the object that follows contains a font size and fill color. By not specifying which font, we'll actually use the Phaser default (Courier).

Next, we need to modify the collectStar function so that when the player picks-up a star their score increases and the text is updated to reflect this:



So 10 points are added for every star and the scoreText is updated to show this new total. If you run your code, you will see the stars fall and the score increase as you collect them.

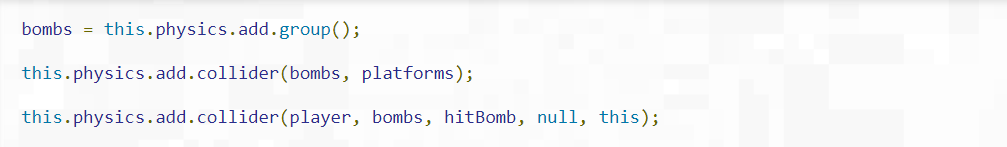


# PART 10: Bouncing Bombs

In order to round our game, it’s time to add some baddies. This will give a nice element of challenge to the game.

The idea is this. When you collect all the stars the first time, it will release a bouncing bomb. The bomb will just randomly bounce around the level and, if you collide with it, you die. All of the stars will respawn so you can collect them again, and if you do, it will release another bomb. This will give the player a challenge: get as high a score as possible without dying.

The first thing we need is a Group for the bombs and a couple of Colliders:



The bombs will of course bounce off the platforms and, if the player hits then, we will call the hitBomb function. All that will do is stop the game and turn the player red:



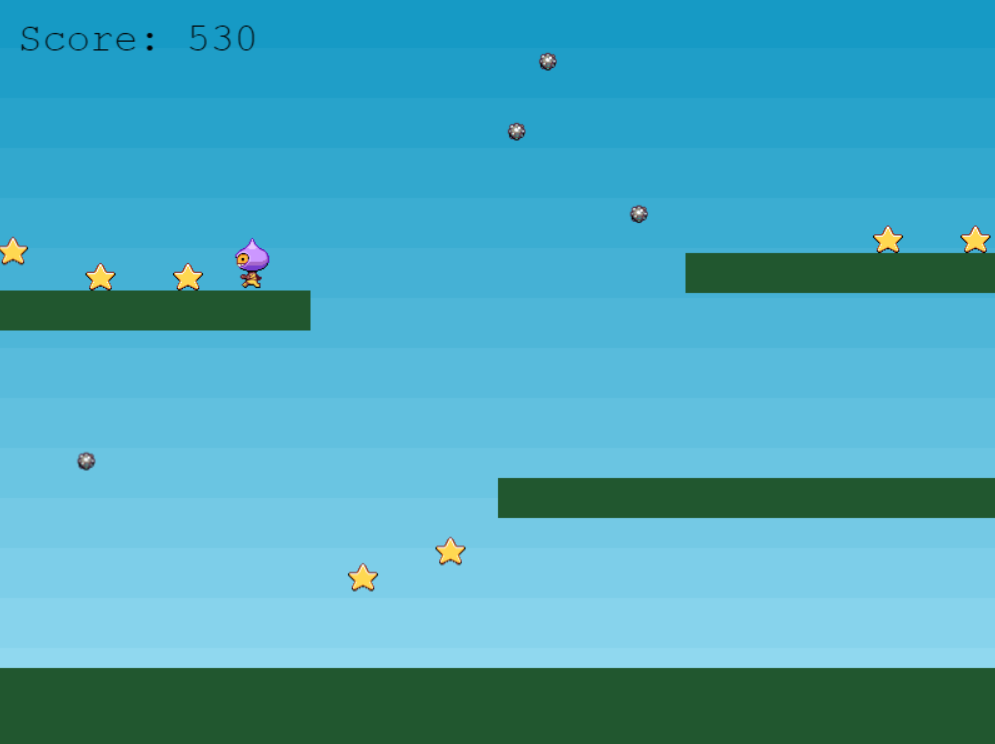
So far, so good. But we need to release a bomb. To do that, we modify the collectStar function:



We use a Group method called countActive to see how many stars are left alive. If it is none, then the player has collected them all. Se we use the iterate function to re-enable all of the stars and reset their y position to zero. This will make all of the stars drop from the top of the screen again.

The next part of the code creates a bomb. First, we pick a random x coordinate for it, always on the opposite side of the screen to the player, just to give them a chance. Then the bomb is created, it’s set to collide with the world, bounce, and have a random velocity.

The end result is a nice little bomb sprite that rebounds around the screen. It is small enough to be easy to avoid at the start. But as soon as the numbers build up, it becomes a lot harder.



Once your tutorial is completed, go to blackboard and submit the assignment with all the files needed in a zipped folder.