Getting Started with Phaser!

Tuesday, May 26, 2015 4:58 PM

['steven-10', 'alexandre', 'steven-9']
['tanay', 'taeuk', 'eason']
['jung-woo', 'genya', 'luke', 'maria']
['dominick', 'vadim', 'tiffany', 'andy']

Download the set of files from s-share.

The text below is taken from the tutorials at http://phaser.io/tutorials/making-your-first-phaser-game . I've added some little bits to help out when needed.

Open the `part1.html` page in your editor of choice and let's have a closer look at the code. After a little boilerplate HTML that includes Phaser the code structure looks like this:

```
var game = new Phaser.Game(800, 600, Phaser.AUTO, '',
{ preload: preload, create: create, update: update });
function preload() {
}


function create() {
}


function update() {
```

```
}
```

Line 1 is where you bring Phaser to life by creating an instance of a Phaser.Game object and assigning it to a local variable called 'game'. Calling it 'game' is a common practice, but not a requirement, and this is what you will find in the Phaser examples.

The first two parameters are the width and the height of the canvas element that Phaser will create. In this case 800 x 600 pixels. Your game world can be any size you like, but this is the resolution the game will display in.

The fourth parameter is an empty string, this is the id of the DOM element in which you would like to insert the canvas element that Phaser creates. As we've left it blank it will simply be appended to the body. The final parameter is an object containing four references to Phasers essential functions. Their use is thoroughly explained here. Note that this object isn't required - Phaser supports a full State system allowing you to break your code into much cleaner single objects. But for a simple Getting Started guide such as this we'll use this approach as it allows for faster prototyping.

Let's load the assets we need for our game. You do this

Let's load the assets we need for our game. You do this by putting calls to game.load inside of a 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 to:

```
function preload() {
game.load.image('sky', 'assets/sky.png');
    game.load.image('ground', 'assets/platform.png');
    game.load.image('star', 'assets/star.png');
    game.load.spritesheet('dude', 'assets/dude.png', 32, 48);

}
```

This will load in 4 assets: 3 images and a sprite sheet. It may appear obvious to some of you, but I would like to point out the first parameter, also known as the asset key. This string is a link to the loaded asset and is what you'll use in your code when creating sprites. You're free to use any valid JavaScript string as the key.

Create a Sprite

In order to add a sprite to our game place the following code in the create function:

```
game.add.sprite(0, 0, 'star');
```

If you bring up the page in a browser you should now see a black game screen with a single star sprite in the top left corner:

The order in which items are rendered in the display matches the order in which you create them. So if you wish to place a background behind the star sprite you would need to ensure that it was added as a sprite first, before the star.

Part 2 - World Building

Part 3 - World Building

Under the hood, game.add.sprite is creating a new Phaser.Sprite object and adding the sprite to the "game world". This world is where all your objects live, it can be compared to the Stage in Actionscript3.

Note: The game world has no fixed size and extends infinitely in all directions, with `0, 0` being the center of it. For convenience Phaser places `0, 0` at the top left of your game for you, but by using the built-in Camera you can move around as needed.

The world class can be accessed via game.world and comes with a lot of handy methods and properties to help you distribute your objects inside the world. It includes some simple properties like game.world.height, but also some more advanced ones that we will use in another tutorial.

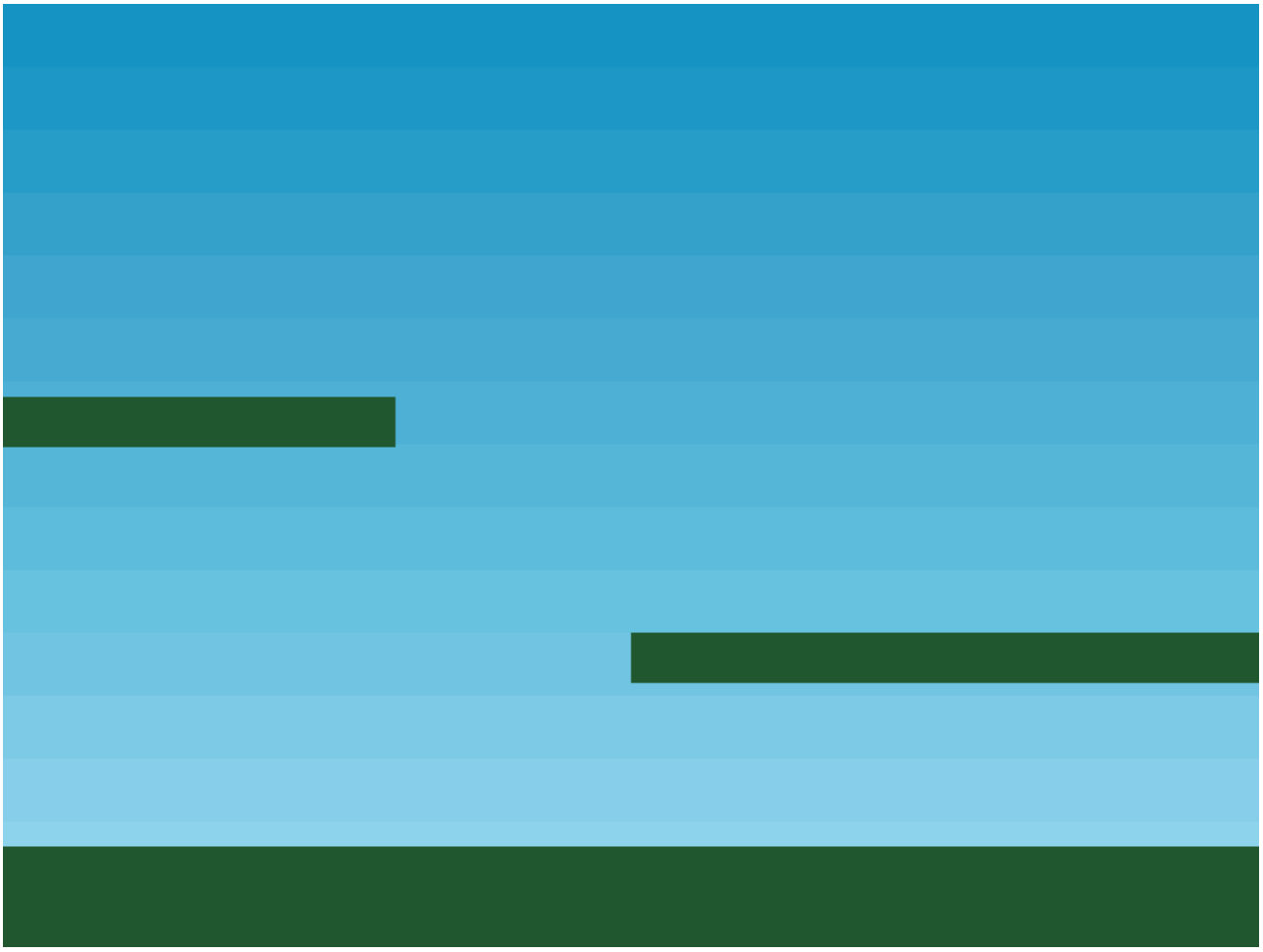
For now let's build up the scene by adding a background and platforms. Here is the updated create function:

```
var platforms;
function create() {
```

```
//  We're going to be using physics, so enable the Arcade Physics system
    game.physics.startSystem(Phaser.Physics.ARCADE);

//  A simple background for our game
    game.add.sprite(0, 0, 'sky');

//  The platforms group contains the ground and the 2 ledges we can jump on
    platforms = game.add.group();

//  We will enable physics for any object that is created in this group
    platforms.enableBody = true;

// Here we create the ground.
    var ground = platforms.create(0, game.world.height - 64, 'ground');

//  Scale it to fit the width of the game (the original sprite is 400x32 in size)
    ground.scale.setTo(2, 2);

//  This stops it from falling away when you jump on it
    ground.body.immovable = true;

//  Now let's create two ledges
    var ledge = platforms.create(400, 400, 'ground');

ledge.body.immovable = true;

ledge = platforms.create(-150, 250, 'ground');
```

```
ledge.body.immovable = true;

}
```

If you run this, which you'll find as `part4.html` in the tutorial zip file, you should see a much more game-like scene:



The first part is the same as the star sprite we had before, only instead we changed the key to 'sky' and it has displayed our sky background instead. This is an 800x600 PNG that fills the game screen

800x600 PNG that fills the game screen.

Part 4 - Groups

Groups are really powerful. As their 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 for this game we'll be using two different Groups, one of which is created in the code above for the platforms.

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

As with sprites `game.add` creates our Group object. We assign it to a new local variable called `platforms`. Now created we can add objects to it. First up is the ground. This is positioned at the bottom of the game and uses the 'ground' image loaded earlier. The ground is scaled to fill the width of the game. Finally we set its `immovable` property to true. Had we not done this the ground would move when the player collides with it (more

on this in the Physics section).

With the ground in place we create two smaller ledges to jump on to using the exact same technique as for the ground.

Ready Player One

Create a new local variable called `player` and add the following code to the create function. You can see this in `part5.html`:

```
    // The player and its settings
    player = game.add.sprite(32, game.world.height - 150, 'dude');
//  We need to enable physics on the player
    game.physics.arcade.enable(player);


//  Player physics properties. Give the little guy a slight bounce.
    player.body.bounce.y = 0.2;
    player.body.gravity.y = 300;
    player.body.collideWorldBounds = true;


//  Our two animations, walking left and right.
    player.animations.add('left', [0, 1, 2, 3], 10, true);
    player.animations.add('right', [5, 6, 7, 8], 10, true);
```

This creates a new sprite called 'player', positioned at 32 pixels by 150 pixels from the bottom of the game. We're

telling it to use the 'dude' asset previously loaded. If you glance back to the preload function you'll 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:

You can see 9 frames in total, 4 for running left, 1 for facing the camera and 4 for running right. Note: Phaser supports flipping sprites to save on animation frames, but for the sake of this tutorial we'll keep it old school.

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 'true' parameter tells the animation to loop. This is our standard run-cycle and we repeat it for running in the opposite direction. With the animations set we create a few physics properties.

Part 5 - The Body and Velocity: A world of physics

By Alvin Ourrad and Richard Davey on 7th December 2013 @photonstorm

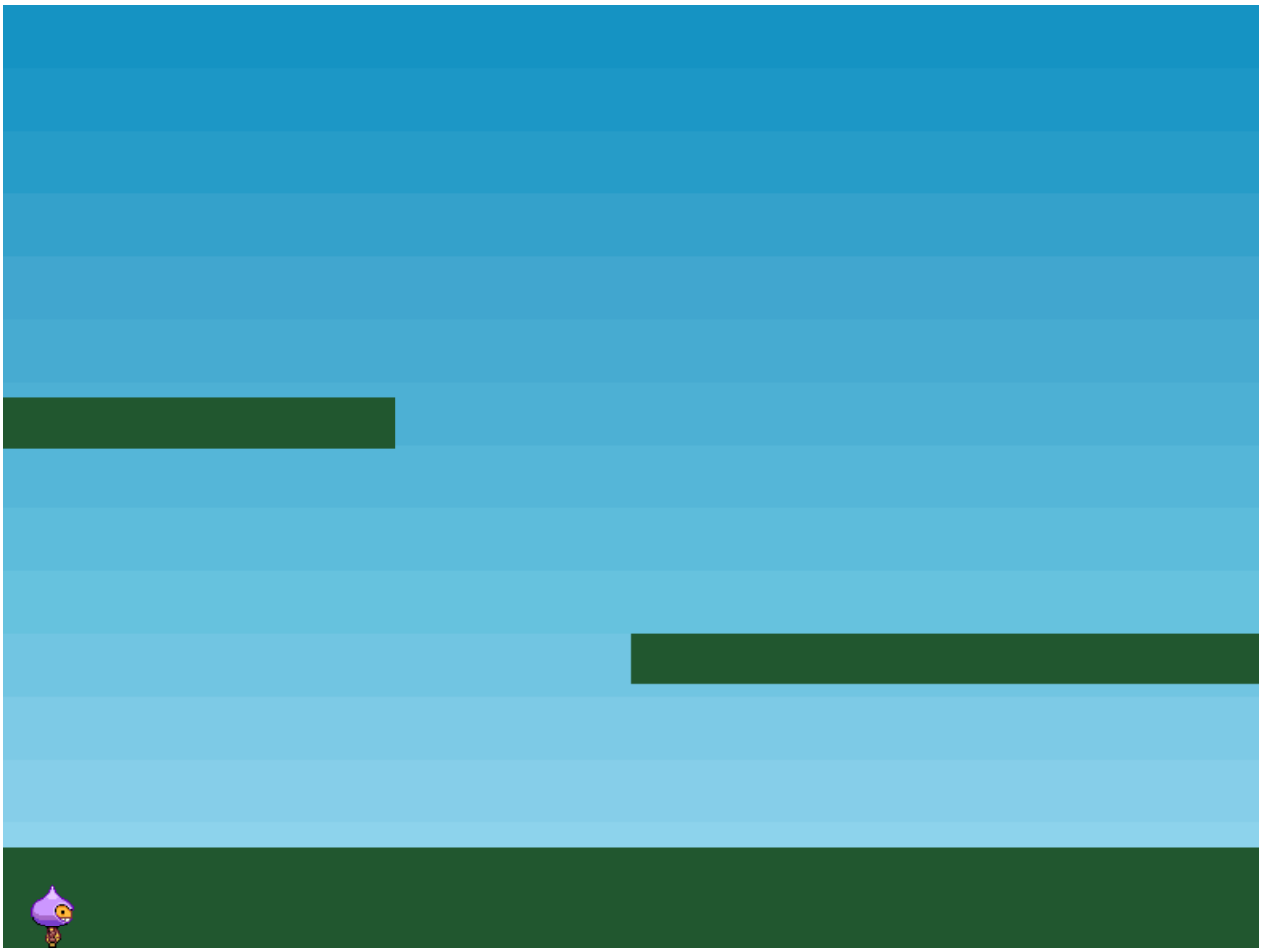
Phaser has support for a variety of different physics systems. It ships with Arcade Physics, Ninja Physics and P2.JS Full-Body Physics. For the sake of this tutorial we will be using the Arcade Physics system, which is simple and light-weight, perfect for mobile browsers. You'll notice in the code that we have to start the physics system running, and then for every sprite or Group that we wish to use physics on we enable them.

Once done the sprites gain a new body property, which is an instance of `ArcadePhysics.Body`. This represents the sprite as a physical body in Phaser's Arcade Physics engine. The body object has itself a lot of properties that we can play with. To simulate the effects of gravity on a sprite, it's as simple as writing this:

```
player.body.gravity.y = 300;
```

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 or run `part5.html` 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 ledges would be immovable. Had we not done that 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 (also known as a dynamic body) 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.

So to allow the player to collide and take advantage of the physics properties we need to introduce a collision check in the update function:

```
function update() {
//  Collide the player and the stars with the platforms
    game.physics.arcade.collide(player,  platforms);


}
```

The update function is called by the core game loop every frame. The Physics.collide function is the one that performs the magic. It takes two objects and tests for collision and performs separation against them. In this case we're 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 both ledges. The result is a firm platform:

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:

```
cursors = game.input.keyboard.createCursorKeys();
```

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

update loop:

```
    //  Reset the players velocity (movement)
    player.body.velocity.x  = 0;
if (cursors.left.isDown)
    {
        //  Move to the left
        player.body.velocity.x  = -150;


player.animations.play('left');
    }
    else if (cursors.right.isDown)
    {
        //  Move to the right
        player.body.velocity.x  = 150;


player.animations.play('right');
    }
    else
    {
        //  Stand still
        player.animations.stop();


player.frame  = 4;
    }


//  Allow the player to jump if they are touching the ground.
    if (cursors.up.isDown  && player.body.touching.down)
    {
        player.body.velocity.y  = -350;
    }
```

Although we've added a lot of code it should all be pretty readable. The first thing we do is reset the horizontal

velocity on the sprite. Then we 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 keyis held down. Frame 4 in the sprite sheet is the one of the player looking at you, idle.

Jump to it

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 350 px/sec sq. 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. Load up part7.html and have a play. Try tweaking values like the 350 for the jump to lower and higher values to see the effect it will have.

Part 7 - Starshine

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 (this can be seen in `part8.html`):

```
    stars = game.add.group();
stars.enableBody = true;

//  Here we'll create 12 of them evenly spaced apart
    for (var i = 0; i < 12; i++)
    {
        //  Create a star inside of the 'stars' group
        var star = stars.create(i * 70, 0, 'star');

//  Let gravity do its thing
        star.body.gravity.y = 6;

//  This just gives each star a slightly random bounce value
```

```
        star.body.bounce.y = 0.7 + Math.random() * 0.2;
    }
```

The process is similar to when we created the platforms Group. Using a JavaScript 'for' loop we tell it to create 12 stars in our game. They have an x coordinate of `i * 70`, which means they will be evenly spaced out in the scene 70 pixels apart. As with the player we give them a gravity value so they'll fall down, and a bounce value so they'll bounce a little when they hit the platforms.

Bounce is a value between 0 (no bounce at all) and 1 (a full bounce). Ours will bounce somewhere between 0.7 and 0.9. If we were to run the code like this the stars would fall through the bottom of the game. To stop that we need to check for their collision against the platforms in our update loop:

```
game.physics.arcade.collide(stars, platforms);
```

As well as doing this we will also check to see if the player overlaps with a star or not:

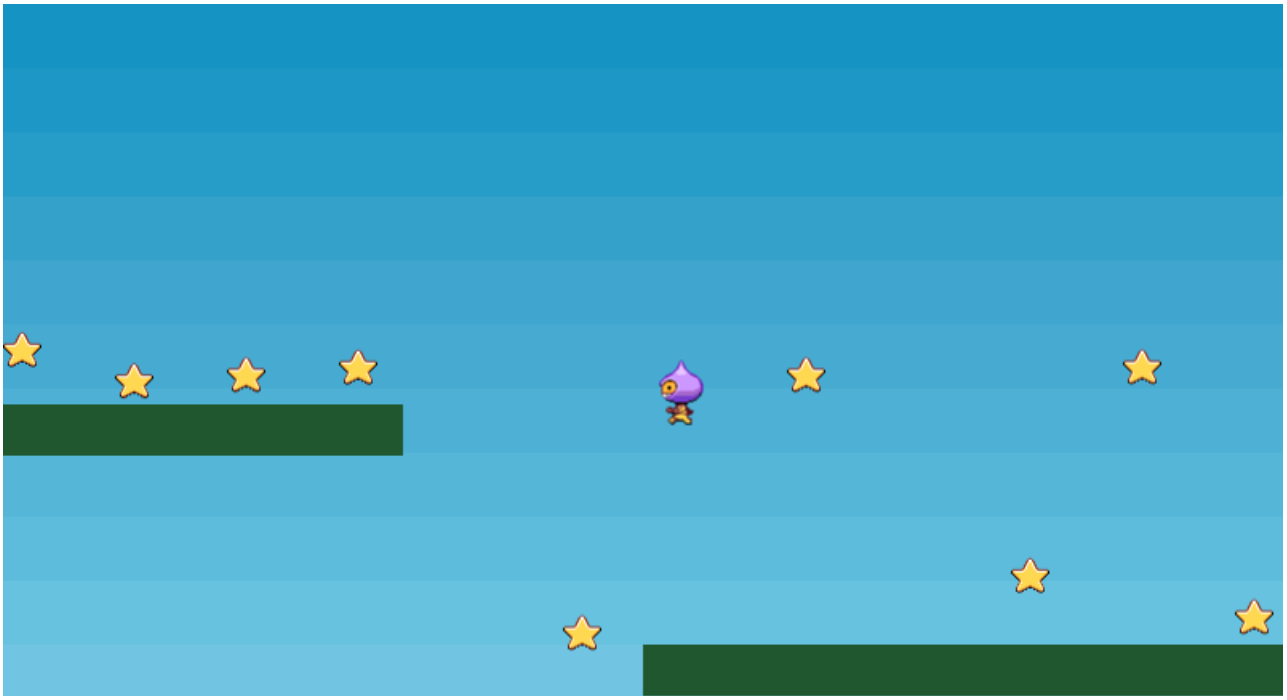
```
game.physics.arcade.overlap(player, stars, collectStar,
```

```
null, this);
```

This tells Phaser to check for an overlap between the player and any star in the stars Group. If found then pass them to the 'collectStar' function:

```
function collectStar (player, star) {
// Removes the star from the screen
    star.kill();

}
```

Quite simply the star is killed, 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 8 - Finishing touches

The final tweak we'll make is to add a score. To do this we'll make use of a Phaser.Text object. Here we create two new variables, one to hold the actual score and the text object itself:

```
var score = 0;
var scoreText;
```

The scoreText is set-up in the create function:

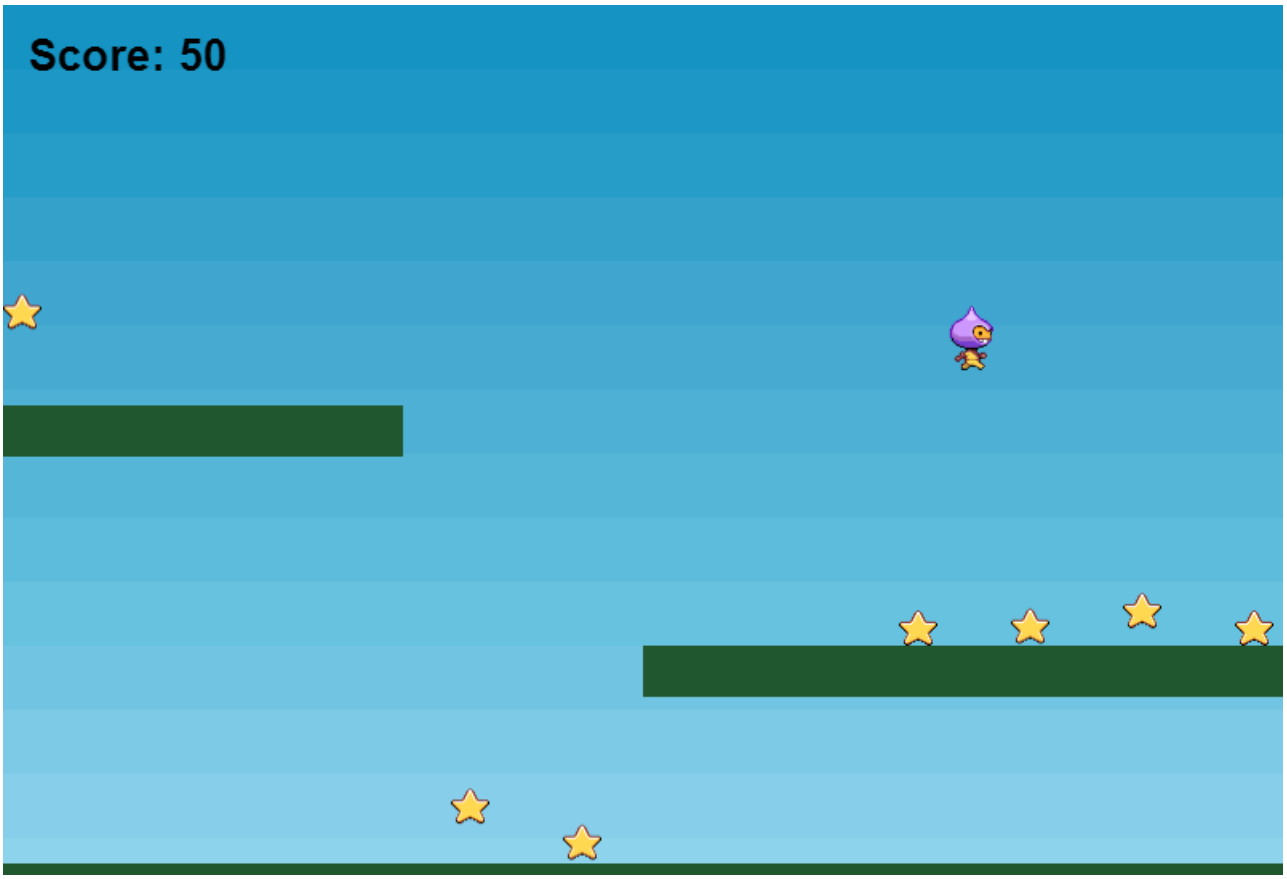
```
scoreText = game.add.text(16, 16, 'score: 0', { fontSize:
'32px', fill: '#000' });
```

16x16 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 colour. By not specifying which font we'll actually use the browser will default, so on Windows it will be Arial. 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:

```
function collectStar (player, star) {
// Removes the star from the screen
    star.kill();

//  Add and update the score
    score += 10;
    scoreText.text = 'Score: ' + score;

}
```

So 10 points are added for every star and the scoreText is updated to show this new total. If you run part9.html you will see the final game.



Conclusion

You have now learned how to create a sprite with physics properties, to control its motion and to make it interact with other objects in a small game world. There are lots more things you can do to enhance this, for example there is no sense of completion or jeopardy yet. Why not add some spikes you must avoid? You could create a new 'spikes' group and check for collision vs. the player, only instead of killing the spike sprite you kill the player instead. Or for a non-violent style game you could make it a speed-run and simply challenge them to collect the stars as quickly as possible. We've included a few extra graphics in the zip file to help inspire you.

With the help of what you have learned in this tutorial and the 450+ examples available to you, you should now have a solid foundation for a future project. But as always if you have questions, need advice or want to share what you've been working on then feel free to ask for help in the Phaser forum.