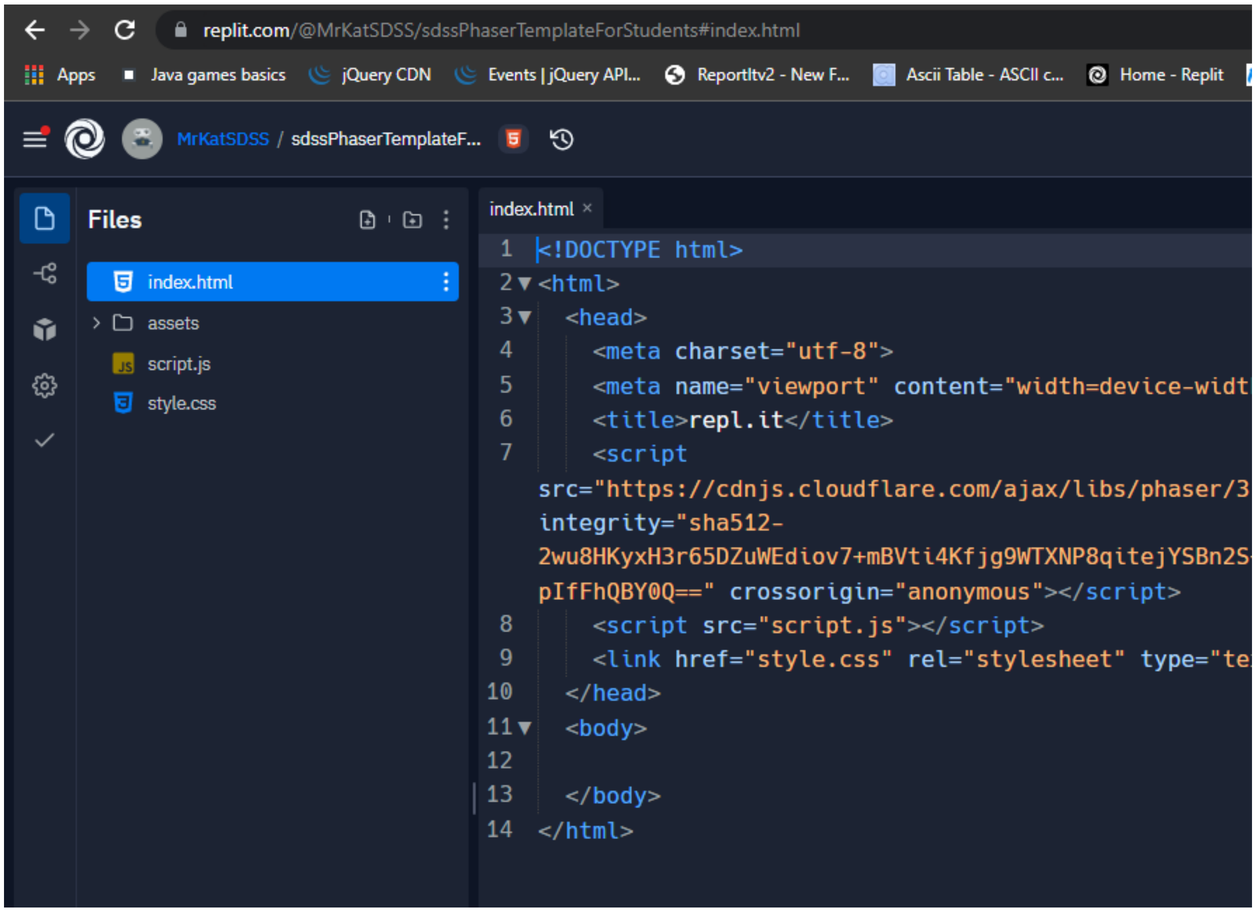
Making a Video Game Using Phaser

Phaser is a game engine or library. This is a collection of commands/functions made by people based on javascript to allow you to more easily make video games. You can make a video game without it but it would be more difficult. It would be like trying to create an alert box without using the alert function: doable, but more difficult than necessary when alert already exist
To begin learning how to make a video game we will first create a copy of a replit project I already made. The project is located at :
https://replit.com/@MrKatSDSS/sdssPhaserTemplateForStudents

Click on the link above and open up the project. There should be a large blue button labeled "Fork". Click on this and if prompted provide a suitable name. This creates a copy of the project for your own personal use.

Your project directory should look as follows:



```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-widt
    <title>repl.it</title>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/phaser/3
integrity="sha512-
2wu8HKyxH3r65DZuWEdiov7+mBVti4Kfjg9WTXNP8qitejYSBn2S
pIfFhQBY0Q==" crossorigin="anonymous"></script>
    <script src="script.js"></script>
    <link href="style.css" rel="stylesheet" type="te
  </head>
  <body>

  </body>
</html>
```

Questions:

1. **How many files are in the root directory?**
    a. **3**
2. **How many folders?**
    a. **1**
3. **Enter the root folder and list all its subfolders.**

        a. 3

4. **Explain what the purpose of these folders are.**
    a. **To store music, sprites and many other graphics for the game.**

Note that the index.html file contains your standard html tags along with a very special script tag:
<script src="https://cdnjs.cloudflare.com/ajax/libs/phaser/3.52.0/phaser.min.js" integrity="sha512-2wu8HKyxH3r65DZuWEdiov7+mBVti4Kfjg9WTXNP8qitejYSBn2S+Kgi4M3ctlnA9qTDU6hyTsiqpIfFhQBY0Q==" crossorigin="anonymous"></script>

This tag instructs the browser that loads this file where to find the phaser game engine (highlight evidence of this in the tag).

Without this script tag we would not have access to the game engine and all of its commands/code and in turn we could not make a game using phaser.

Sprites

"Sprites are images that represent game assets. Player characters, enemies, projectiles, and other items are all called sprites (more on sprite types to come). Thus, sprites appear everywhere in games, including the title screen, within game levels, and even the game over screen."
Sprites are obviously the most important part of a video game as they represent all the things you see and interact with in a game. Most often they are image files that end in the .png extension. Although you can use other types of graphics, they are not recommended.

Questions:

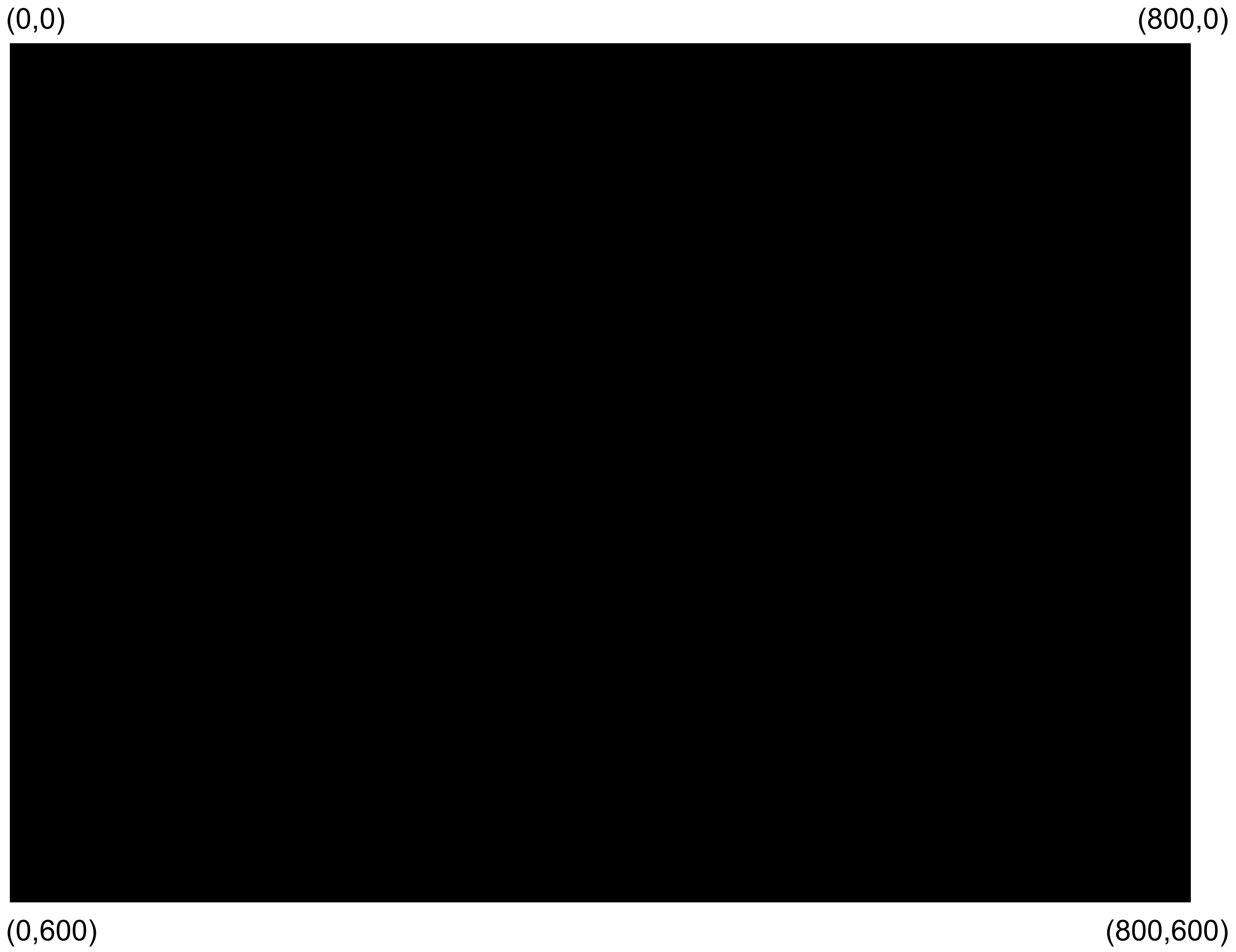
1. **Search the assets folder of your project and list the names of at least three sprites.**
    a. **Background.png, meteor.png, ship_1.png**
2. **Describe the type of game these sprites look like they might be part of.**
    a. **Maybe like a arcade shoot out game like space invaders**
3. **Find one sprite that looks different from the rest and describe what you think that image would be used for.**
    a. **The alien image looks quite different compared to the game we are trying to create.**

The Game World

The game world is the 2D representation of the world in which our sprites will reside. It has 2 dimensions: a horizontal (defined by an x coordinate) and a vertical (define, by you guessed it, a y coordinate).

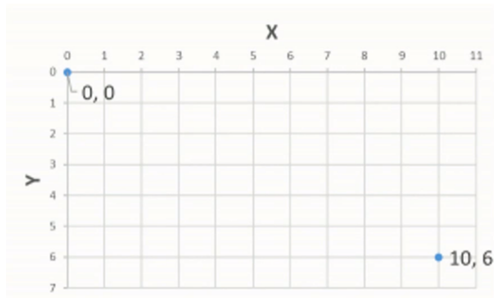
Run the project. You will see this in the replit output window:

Your world is black. That's because it's empty. There are no sprites in it. We will begin adding some very soon. Before we do please note that the origin of your game world is at the top left corner of the screen. That means (0,0) is the top left corner.



If you studied the game world above you would have noticed something very different from math class. Your origins as indicated is the top left corner and the y coordinates are positive going downwards, not upwards. That will become very important as we learn to make sprites move and to position them in the game world.

Your coordinate system looks something like this:



Note that x is positive going right while y is positive going down and both go negative in reverse. Also note that the screen width is 800 pixels wide and it's height is 600 pixels. How do I know that? Well, it was set in code and you can change this.

Exercises

Select and open the script file. Note the code that is already there. Be very careful not to edit the code without first knowing what you are doing, otherwise you may break the project and it could become difficult to debug. Search through the code and find the numbers 800 and 600. Change these numbers and reload the project. Describe what happens? Change them a few more times and experiment with different sizes. Always reload your game screen/project after each change.

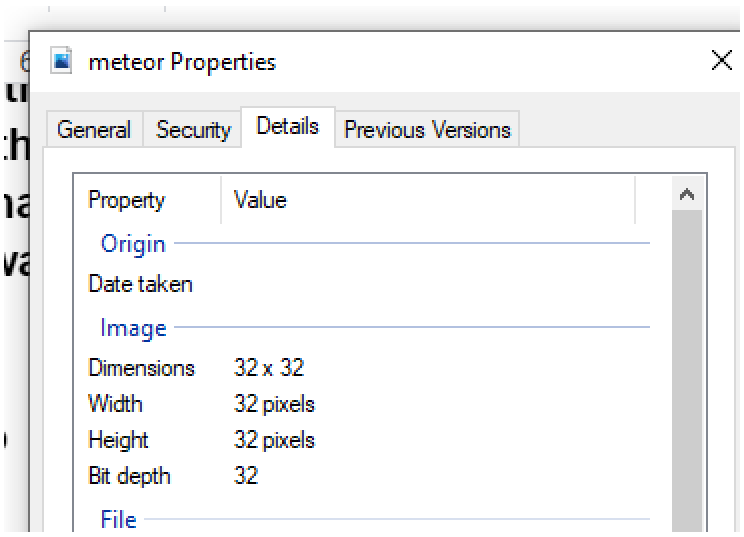
The black box in the middle changes in size

Whatever numbers you decide upon for the dimensions of your game world you need to remember it when dealing with the rest of your code.

Questions:

1. **If the game word was 1200 pixels wide and 700 pixels tall what are the coordinates for the following locations: a. Top left corner b. Bottom right corner c. top right corner d. Bottom left corner e. Centre of the screen**
    a. **0x0**
    b. **1200x700**
    c. **1200x1**
    d. **1x700**
    e. **600x350**

To position sprites we need to know their dimensions as well. You can do this in a number of different ways. We'll do it as follows: a. Right click and download the meteor.png file. B. Find it in windows using your file explorer (should be in the downloads folder) c. right click the file and select properties d. Select the Details tab and there you should find the dimensions 32x32. This means the image is 32 pixels wide by 32 pixels tall.


meteor Properties

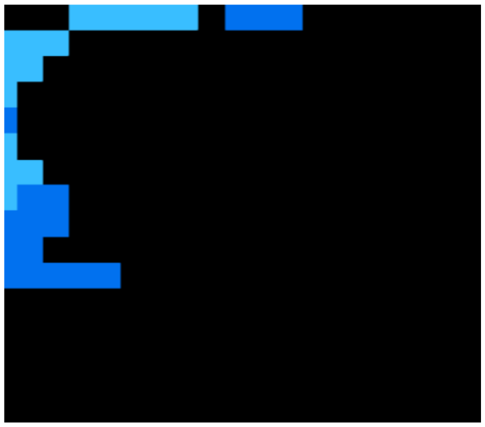
Property	Value
Origin	
Date taken	
Image	
Dimensions	32 x 32
Width	32 pixels
Height	32 pixels
Bit depth	32
File	

Exercise:
Select 3 other sprites and list their file names along with their dimensions.
I cant, the school computers wont let me.

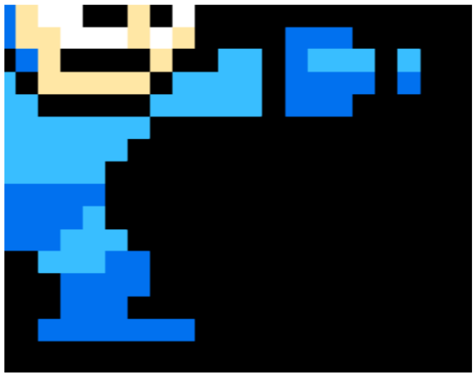
You need to know the dimensions to properly position your sprites.

Positioning Sprites
A sprite's position is set by its centre. So when you put a sprite at (0,0) its centre is put at (0,0). For example, if you look at the following sprite you should notice that it's centre represents it tummy (or approximately in that area). If I placed this sprite at (0,0) its tummy gets placed there and so parts of the rest will get hidden.

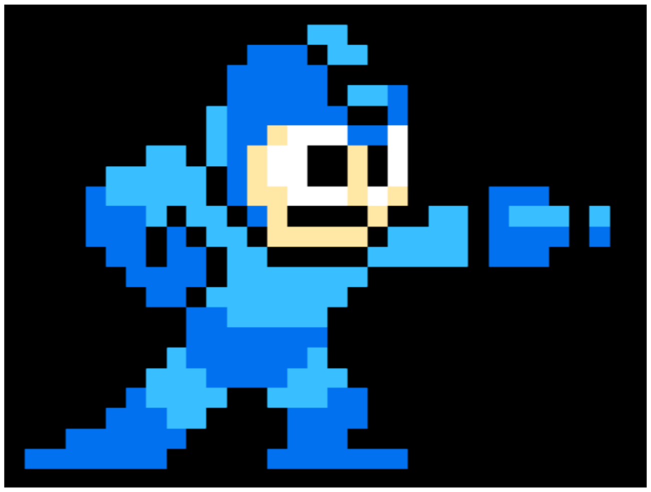
This is what you would see:



Notice you can see its right foot and art of its arm. Let's move him to the right and down by setting his position to (100,100).



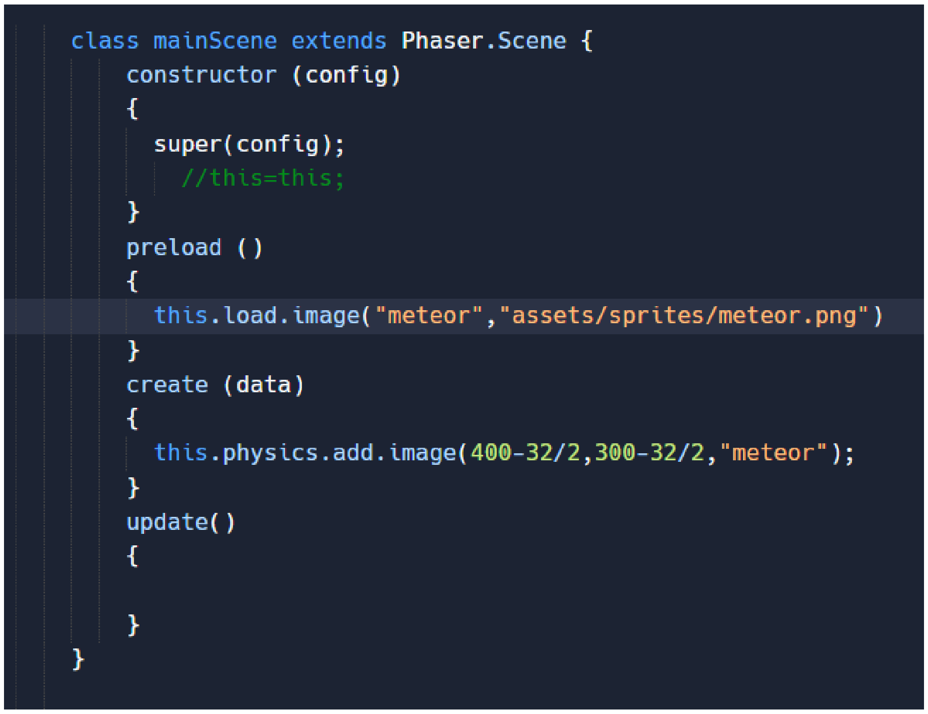
Now you see more of him but not all of him. To fix that I need to know his size. It's 779 x 602. I just need to move him half his width into the screen horizontally and half his height down vertically. 779/2=390 602/2=301 so his new coordinates should be (390,301). Let's see what happens:



This is a big sprite and so it will take up most of the screen. Nevertheless we can now see the whole sprite. You can use many tools online to resize a sprite to fit or to simply make it a size you want. We can also do it in code (but there are often implications for this).

Adding a Sprite

Let's now add a sprite by adding the necessary code. Add the following code inside the **preload** and **create** functions of the **mainScene**:



```
class mainScene extends Phaser.Scene {
    constructor (config)
    {
      super(config);
        //this=this;
    }
    preload ()
    {
      this.load.image("meteor","assets/sprites/meteor.png")
    }
    create (data)
    {
      this.physics.add.image(400-32/2,300-32/2,"meteor");
    }
    update()
    {

    }
}
```

Let's examine these two lines of code:

this.load.image("meteor","assets/sprites/meteor.png")
this.physics.add.image(400-32/2,300-32/2,"meteor");

this.load.image("meteor","assets/sprites/meteor.png")

This line of code loads the meteor.png image file into memory so that it can be used in our game. We assign it the name "meteor". That name could be anything. It's what we will use to identify that image. Of course, **"assets/sprites/meteor.png",** is the path to the file**.** Every image must be loaded into memory first by using this line of code in the preload function of the game screen.

Exercises:

Since we will be making a space game using most of the sprites in the assets folder, add the necessary code to the preload function to load all the images into memory. Make sure to give each image a unique, camelCase, memorable name. Copy and paste the code you wrote below:

```
this.load.image("ship1","assets/sprites/ship_1.png");
this.load.image("ship2","assets/sprites/ship_2.png");
this.load.image("bg","assets/sprites/background_stars.png");
this.load.image("meteor","assets/sprites/meteor.png");
this.load.image("fMeteor","assets/sprites/flaming_meteor.png");
```

```
this.physics.add.image(400-32/2,300-32/2,"meteor");
```

This code you may have noticed is what actually takes the image you loaded into memory and places it onto the game screen. The first two numbers are the math I used to find the exact centre of the meteor in the game world. You could have instead written:

```
this.physics.add.image(384,284,"meteor");
```

This command requires three pieces of information 1. The x coordinate of the sprite 2. the y coordinate 3. The name of the sprite (not the filename but the name you assigned to it when loading it into memory in the preload function)**.** All images are added to our game in the **create()** function.

Exercises:

1. **Add an image of 4 meteors to the four corners of the screen. Calculate the correct coordinates using its dimensions so that you see the full meteors. Note, you can use the same image as many times as you want. You only have to load it once and then add it as many times as you want. Copy and paste the code you used below:**

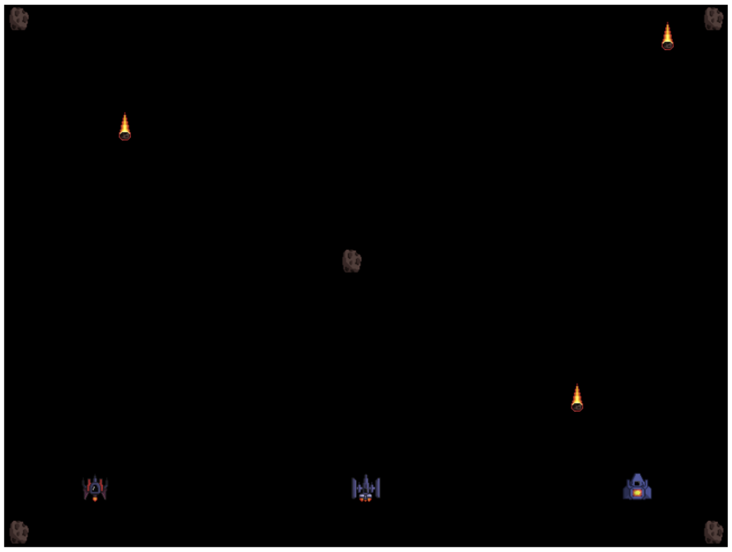
2. **Add 3 “flaming_meteors.png” in 3 random places in your game world. Copy and paste the code you used below:**

```
x=Math.round(Math.random()*800);
 y=Math.round(Math.random()*600);
 flamingMeteor1=this.physics.add.image(x,y,"fMeteor");
 x=Math.round(Math.random()*800);
 y=Math.round(Math.random()*600);
 flamingMeteor2=this.physics.add.image(x,y,"fMeteor");
 x=Math.round(Math.random()*800);
 y=Math.round(Math.random()*600);
 flamingMeteor3=this.physics.add.image(x,y,"fMeteor");
```

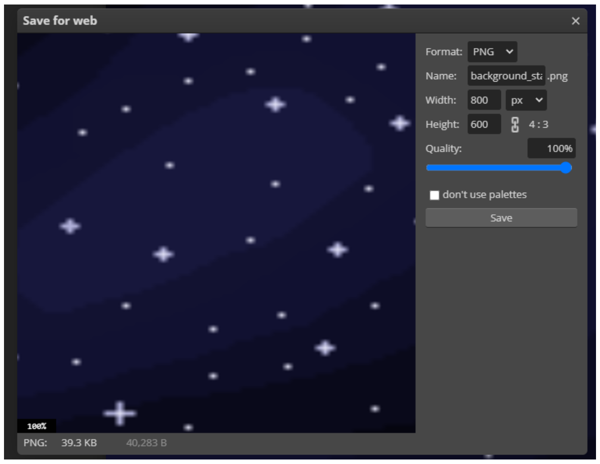
3. **Add the three ships so that they are evenly spaced apart near the bottom of the screen. Copy and paste the code you used below:**

```
ship1=this.physics.add.image(100,300,"ship1");
 ship2=this.physics.add.image(700,500,"ship2");
```

You should now have what looks as follows:

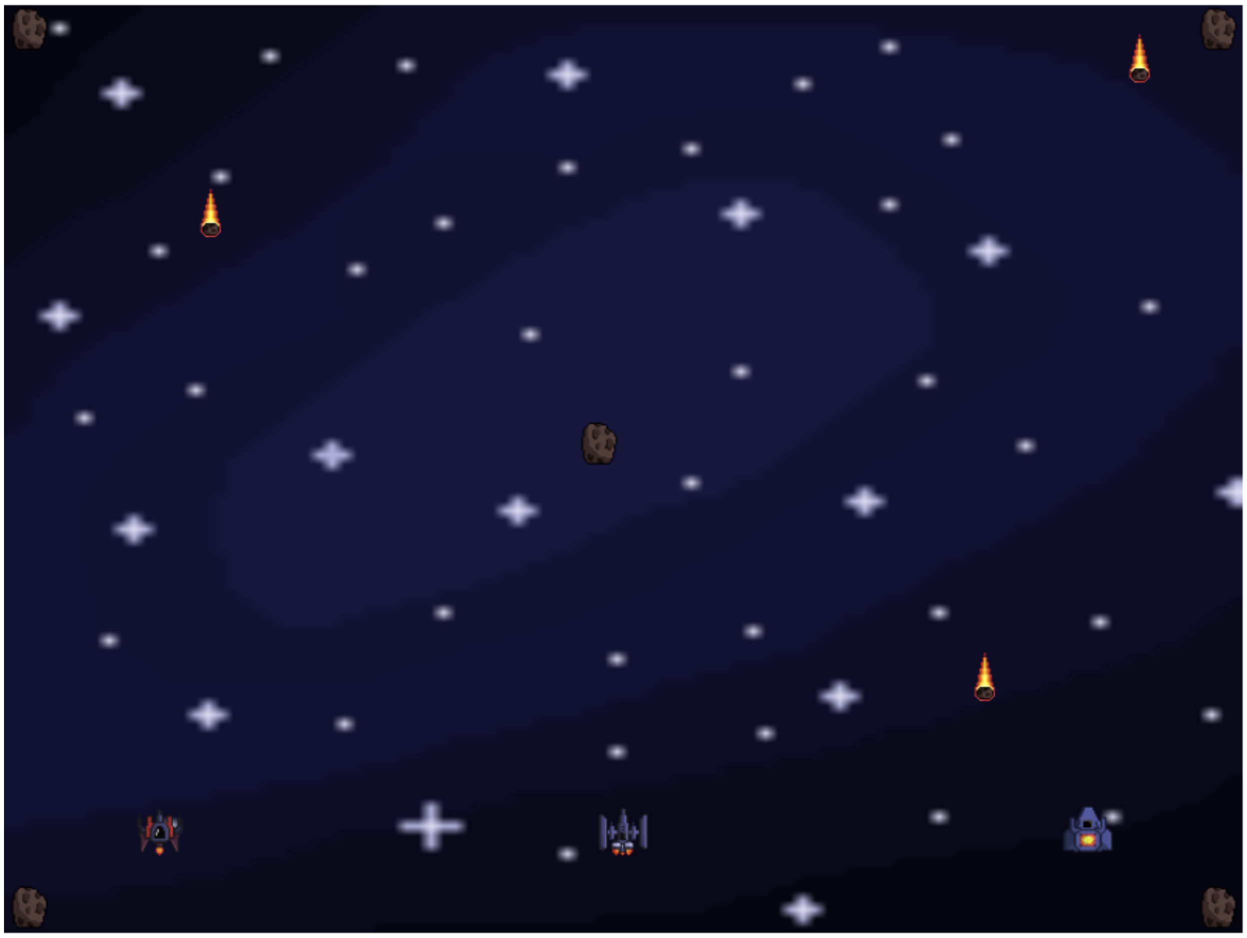


Exercises:
Now try adding the background_stars.png file. Explain below what happens?
An image is added to the game.
Let's resize the background image so that it fits our game world. Download the image and upload it to https://www.photopea.com/.
Go to File>Export As>PNG
Change the width and height to 800 and 60 (make sure to uncheck the 'keep aspect ratio' button…looks like a chain icon). Click the save button. Rename the file to something easy to type and upload it to your sprites folder. Add the image back to your game and then explain what happens?
It fit to size



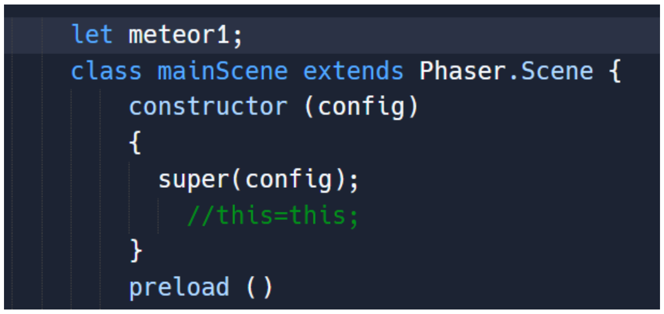
Format: PNG
Name: background_st .png
Width: 800 px
Height: 600 4 : 3
Quality: 100%
don't use palettes
PNG: 39.3 KB 40,283 B

You should have noticed that the other sprites are gone. That's because you added the largest image on top of the others. Add it before the others. You should now have something as follows:



Moving Sprites

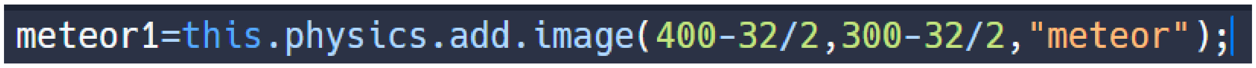
A game is useless unless we can get our sprites to move. To do that we will need to have a reference to those sprites. We use variables for this. We need to assign a variable to each sprite so that we can then control them. For reasons you don't need to understand now you will create all of your variables at the very top of your code file, outside of any braces{}. Create your first variable as shown below:



```
let meteor1;
class mainScene extends Phaser.Scene {
    constructor (config)
    {
      super(config);
        //this=this;
    }
    preload ()
```

Note that the name chosen should be reflective of what the variable will refer to. In this case we will use this variable to refer to one of our meteors. We will need separate variables for each meteor.

Assign the first meteor in your screen to the first variable you created as follows:

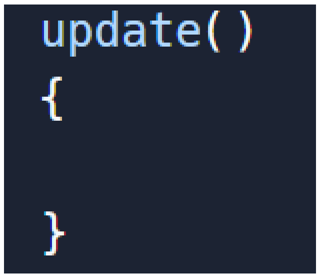


```
meteor1=this.physics.add.image(400-32/2,300-32/2,"meteor");
```

Note that this is done in the create function where we created and added the sprite to our game world.

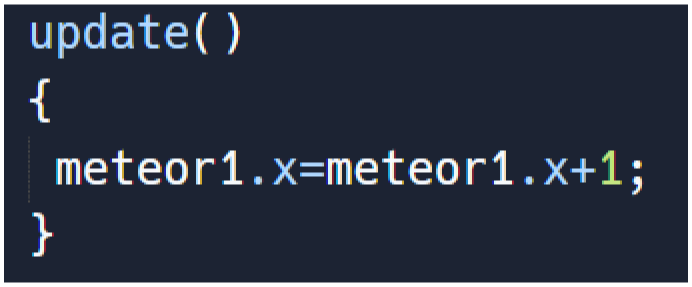
Now that we have a variable that has a reference to the sprite we can control the sprite through the variable. Variables store data. They can be numbers, strings, booleans and sprites and sounds and fonts and so on!

To move sprites we have to write our code in the final function that belong to each game scene/screen: the update() function!



```
update()
{


}
```

This block of code will get executed 30,40,50 times a second. It's the game loop. Although it doesn't look like a loop. It is. It gets executed over and over again, many times each second. It is where all game logic is written. We will move sprites here. Check for collisions. Update scores and so much more! For now let's just move the meteor across the screen. Update the code as follows:



```
update( )
{
 meteor1.x=meteor1.x+1;
}
```

Questions

1. **Explain how this code works.**
    a. **It adds 1 pixel to the right of the meteor1 image. Since its under "update" it will constantly iterate itself**
2. **What would happen if I added 2 instead of 1?**
    a. **It would constantly iterate 2 pixels**
3. **What would happen if I subtracted -1 instead of adding 1?**
    a. **It will go backwards**

Exercises:

1. **Add a new line where you add 1 to the meteor's y coordinate. Copy and paste the code you used below.**

```
update()
{
 meteor1.x=meteor.y+1
}
```

2. **Update this by adding 3 instead of 1. What happens to the speed and direction of the meteor.Copy and paste the code you used below.**

```
update()
{
 meteor1.x=meteor.y+3
}
```

Its moving up very fast

3. **Write the code below that would move the meteor towards the upper left corner of the screen.Copy and paste the code you used below.**

```
update()
{
 meteor1.x=meteor.y+1
 meteor1.x=meteor.x-1
}
```

4. **Write the code below that would move the meteor towards the bottom left corner of the screen.Copy and paste the code you used below.**

```
update()
{
  meteor1.x=meteor.y-1
  meteor1.x=meteor.x-1
}
```

5. Write the code below that would move the meteor towards the bottom right corner of the screen.Copy and paste the code you used below.

```
update()
{
  meteor1.x=meteor.y-1
  meteor1.x=meteor.x+1
}
```

6. Write the code below that would move the meteor towards the bottom right corner of the screen but moves 3 times as fast as the last meteor.Copy and paste the code you used below.

```
update()
{
  meteor1.x=meteor.y-3
  meteor1.x=meteor.x+3
}
```

7. Create a new variable for every sprite on your screen. Make sure the names are unique. Add code to your update function to move all meteors in random different directions. Copy and paste the code you used below.

```
if(flamingMeteor1.y>900)
    {
     let y=-Math.round(Math.random()*600);
     let x=(50+Math.round(Math.random()*400));
     flamingMeteor1.x=x;
     flamingMeteor1.y=y;
     flamingMeteor1Speed=3+Math.round(Math.random()*7);
    }
  flamingMeteor2.y+=flamingMeteor2Speed;//move it to the right
    if(flamingMeteor2.y>900)
    {
     let y=-Math.round(Math.random()*600);
     let x=(50+Math.round(Math.random()*400));
     flamingMeteor2.x=x;
    flamingMeteor2.y=y;
    flamingMeteor2Speed=3+Math.round(Math.random()*7);
  }
flamingMeteor3.y+=flamingMeteor3Speed;//move it to the right
  if(flamingMeteor1.y>900)
  {
    let y=-Math.round(Math.random()*600);
    let x=(50+Math.round(Math.random()*400));
    flamingMeteor3.x=x;
    flamingMeteor3.y=y;
    flamingMeteor3Speed=3+Math.round(Math.random()*7);
  }
  //control meteor1 movement
   meteor1.x+=meteor1Speed;//move it to the right
  if(meteor1.x>900)
  {
    let y=Math.round(Math.random()*600);
    let x=-(50+Math.round(Math.random()*400));
    meteor1.x=x;
    meteor1.y=y;
    meteor1Speed=3+Math.round(Math.random()*7);
  }
//control meteor2 movement
meteor2.x+=meteor2Speed;//move it to the right
  if(meteor2.x>900)
  {
    let y=Math.round(Math.random()*600);
    let x=-(50+Math.round(Math.random()*400));
    meteor2.x=x;
    meteor2.y=y;
    meteor2Speed=3+Math.round(Math.random()*7);
  }
//control meteor3 movement
meteor3.x+=meteor3Speed;//move it to the right
  if(meteor3.x>900)
  {
    let y=Math.round(Math.random()*600);
    let x=-(50+Math.round(Math.random()*400));
    meteor3.x=x;
    meteor3.y=y;
    meteor3Speed=3+Math.round(Math.random()*7);
  }
```

8. **Modify your code so that the flaming meteors are moving downwards only. Make sure they move at different speeds.**

Making Decisions

Making decisions are an important part of game programming. Some of things that we need to decide might include the following:

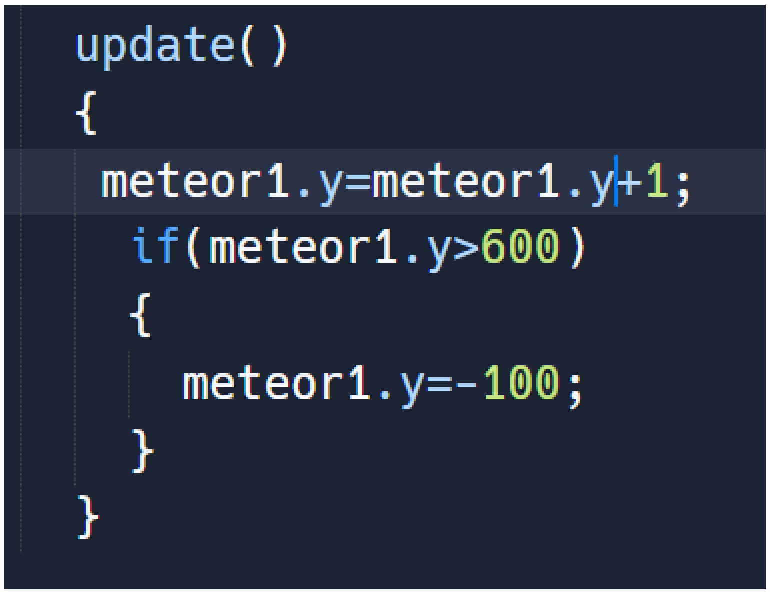
- Is the game over?
- Have two sprites collided?
- Is the level complete?
- Do you have any lives left to continue?
- Did someone collect a power up?
- Etc.

We will now modify our code to demonstrate some simple decision making. Get rid of all the code in the update function. Add the code necessary to move one meteor straight down. Eventually the meteor will move off the screen. Test this by running your code.

The meteor continues to mov off the screen. As a result its y coordinate gets larger and larger until eventually it gets too big and the program will crash. We need to decide when the meteor moves off the screen and then we can move it back up top so that it continues to move down. This makes it look like we have a new meteor!

Questions and Exercises

1. **Add the following code to your update function:**

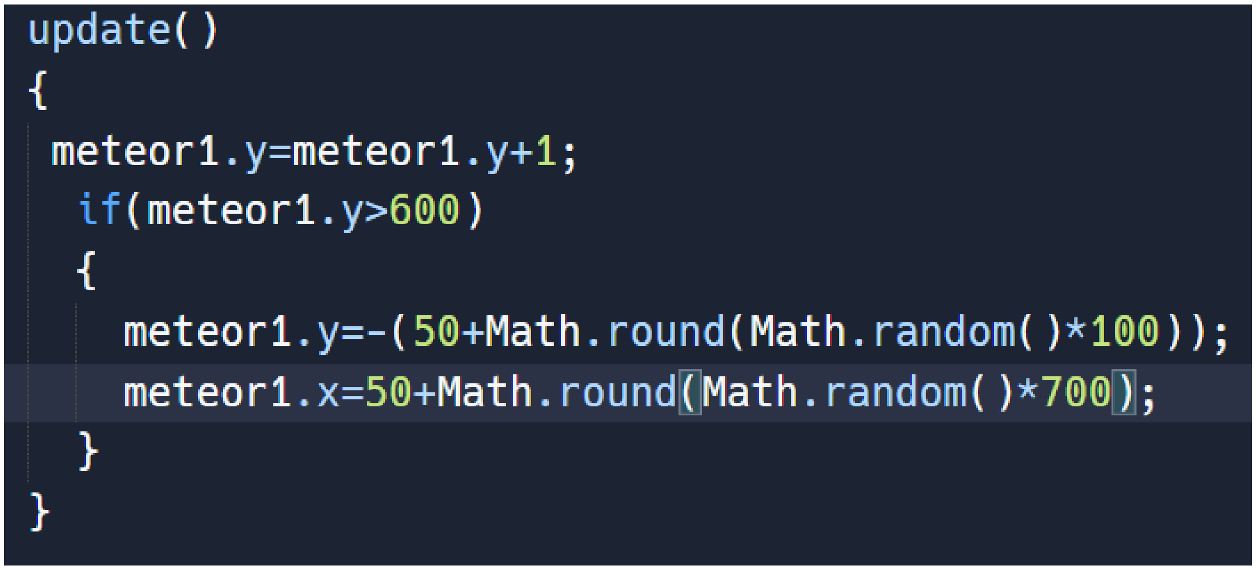


```
update( )
{
 meteor1.y=meteor1.y+1;
  if(meteor1.y>600)
  {
    meteor1.y=-100;
  }
}
```

2. **Explain what is happening here.**
   a. **Meteor1 is constantly being moved 1 pixel up. This happens right until it is greater than 600. If it is greater than 600, the meteor is sent back 100 pixels**
3. **Why do you think I set the meteor's y coordinate to -100? Why not set it to 0?**
   a. **If it was set to 0 the meteor would go back to where it came from**

4. **Try setting the y coordinate to 0 and explain what happens.**

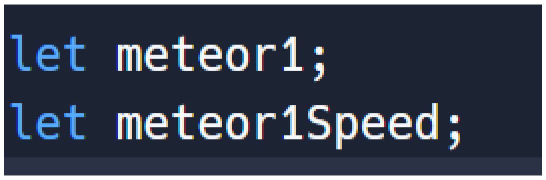
We can update the code so that the meteor also appears at different positions along the horizontal axis as well as start from different vertical positions. Update your code to the following:



```
update()
{
 meteor1.y=meteor1.y+1;
  if(meteor1.y>600)
  {
    meteor1.y=-(50+Math.round(Math.random()*100));
    meteor1.x=50+Math.round(Math.random()*700);
  }
}
```

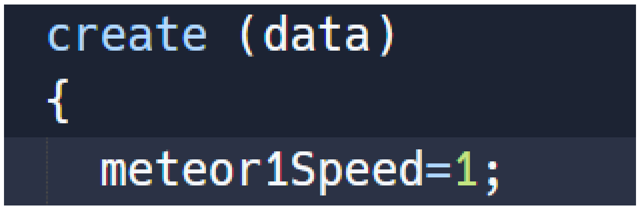
Explain what is being done in the two new lines of code inside the if statement?

Notice that the meteor is always moving at the same speed. Each time a new meteor appears from the top it moves 1 pixel at a time. Let's now update this so that the speed will change for each new incoming meteor. To do this we can't set the speed to 1. We need some way of varying this value. Note the use of the term varying … i.e. variable! We need to store it's speed in a variable and that way we can vary the speed! You should never hard code any values unless you know for certain it will never change in your game. Store everything in a variable. Create a new variable called **meteor1Speed**.



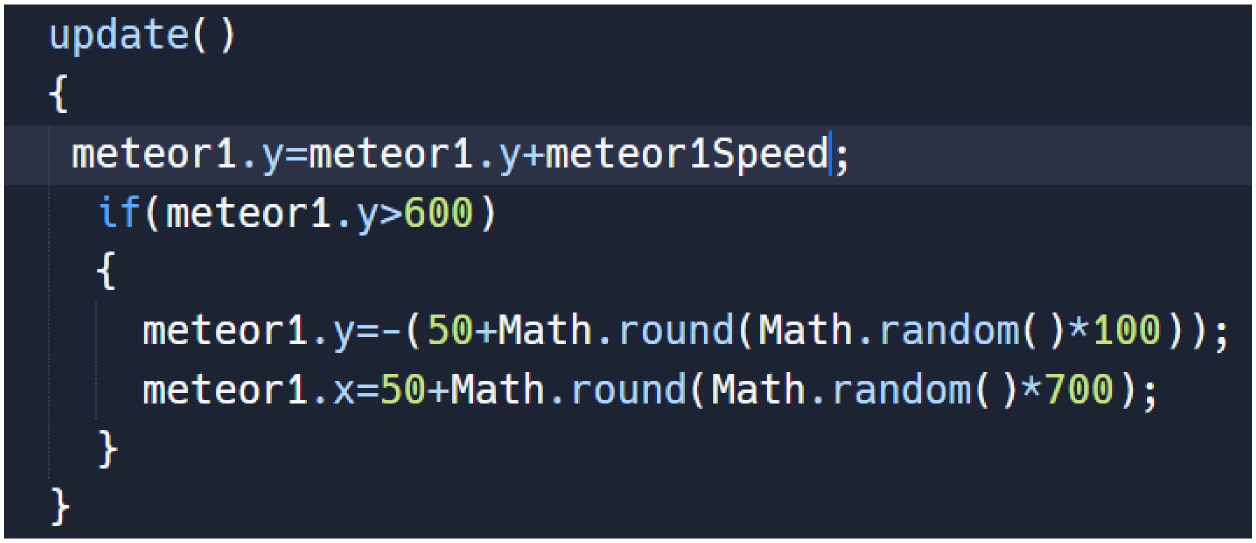
```
let meteor1;
let meteor1Speed;
```

We should initialize all variable values in the **create** function. Let's set the meteor's speed to 1 (lie it already was).



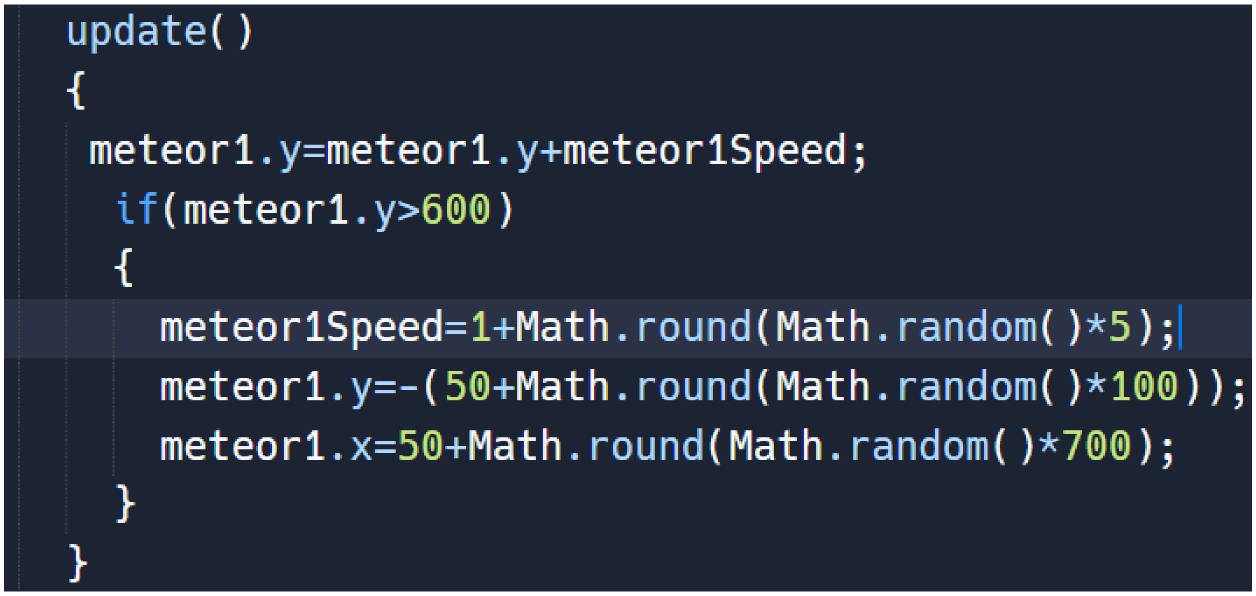
```
 create (data)
 {
   meteor1Speed=1;
```

Modify the update code to use this variable to move the meteor.



```
update()
{
 meteor1.y=meteor1.y+meteor1Speed;
  if(meteor1.y>600)
  {
    meteor1.y=-(50+Math.round(Math.random()*100));
    meteor1.x=50+Math.round(Math.random()*700);
  }
}
```

Run your game. Nothing changes! Of course not. That's because the meteor still moves 1 pixel for every frame or loop of the game. Now, however, because we have the speed stored in a variable we can change it's value at any time. Let's change it when the meteor get's reset to the top of the screen after it moves off the bottom.



```
update()
{
 meteor1.y=meteor1.y+meteor1Speed;
  if(meteor1.y>600)
  {
    meteor1Speed=1+Math.round(Math.random()*5);
    meteor1.y=-(50+Math.round(Math.random()*100));
    meteor1.x=50+Math.round(Math.random()*700);
  }
}
```

Explain below what are the possible speeds of the meteor?

Exercises:

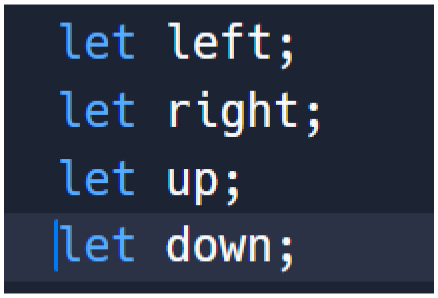
1. **Move all the meteors and reposition them to the opposite side of the screen. You may want to move some horizontally, left, right, up, down and if you really want a challenge move them diagonally and reposition them properly. The flaming meteor's should probably move down only at this point because their flames trail behind them vertically.**

Inputs

A game without input from a player is just an animation. We need to be able to get the player to interact with the game world. Let's do that by learning how to get input from the keyboard and mouse.

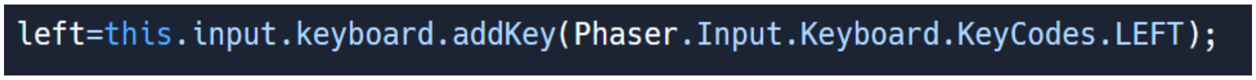
Keyboard Input

Just like everything else in the game that needs to be accessed and controlled by us we will need variables to store each key. We will move one of our ships based on the user's input of the arrow keys. To start make four variables to store these keys:



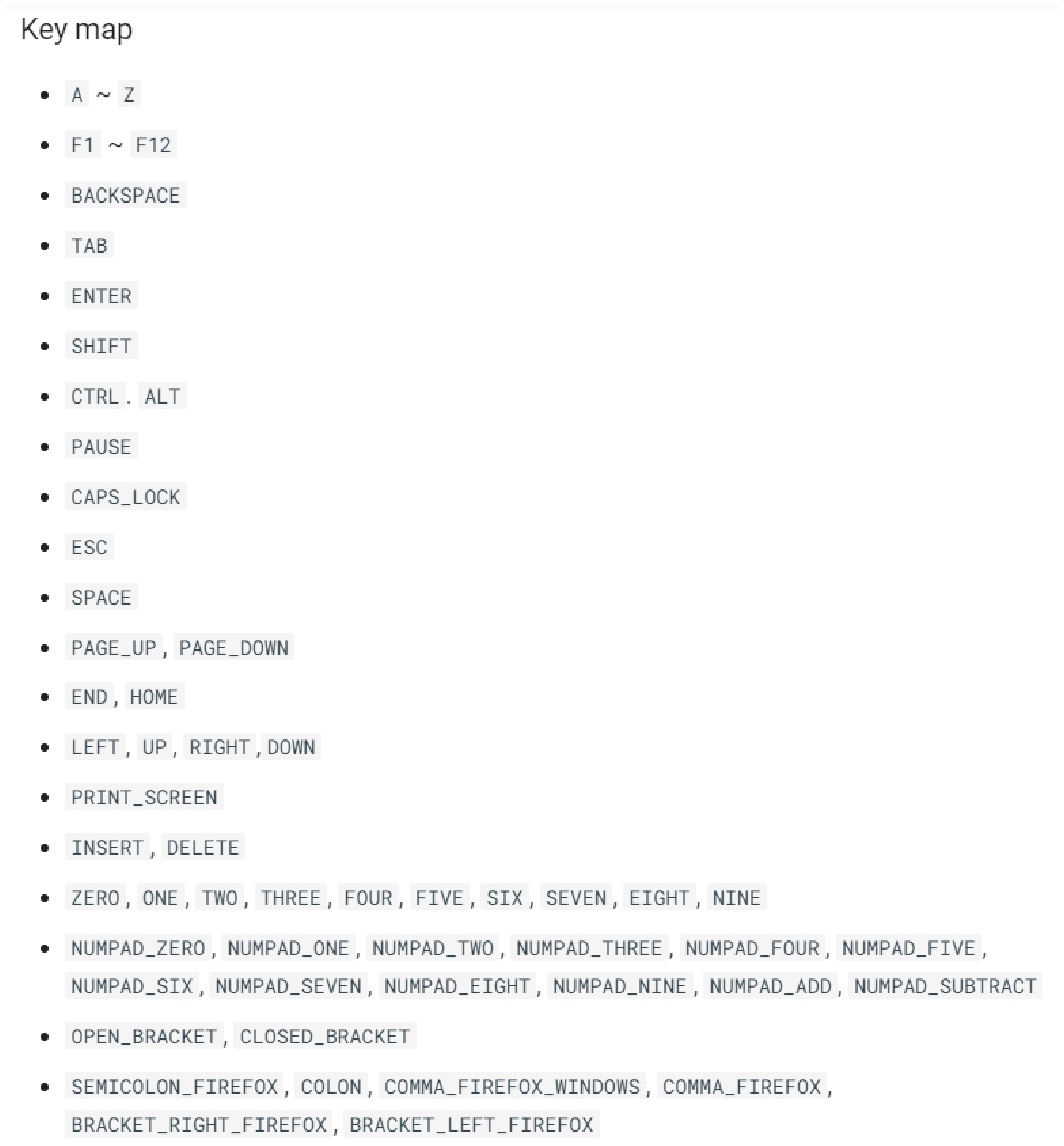
```
let left;
let right;
let up;
let down;
```

Next we create and assign the keys to the variables in the **create** function:



```
left=this.input.keyboard.addKey(Phaser.Input.Keyboard.KeyCodes.LEFT);
```

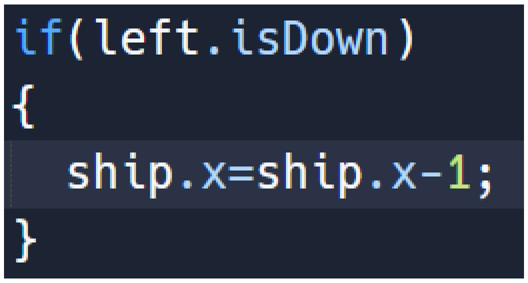
Note the end of this line of code. The LEFT property determines what key on the keyboard to assign to the variable. Phaser uses the following keymap to determine what word associates with what key:



Key map

- `A` ~ `Z`
- `F1` ~ `F12`
- `BACKSPACE`
- `TAB`
- `ENTER`
- `SHIFT`
- `CTRL` . `ALT`
- `PAUSE`
- `CAPS_LOCK`
- `ESC`
- `SPACE`
- `PAGE_UP` , `PAGE_DOWN`
- `END` , `HOME`
- `LEFT` , `UP` , `RIGHT` , `DOWN`
- `PRINT_SCREEN`
- `INSERT` , `DELETE`
- `ZERO` , `ONE` , `TWO` , `THREE` , `FOUR` , `FIVE` , `SIX` , `SEVEN` , `EIGHT` , `NINE`
- `NUMPAD_ZERO` , `NUMPAD_ONE` , `NUMPAD_TWO` , `NUMPAD_THREE` , `NUMPAD_FOUR` , `NUMPAD_FIVE` , `NUMPAD_SIX` , `NUMPAD_SEVEN` , `NUMPAD_EIGHT` , `NUMPAD_NINE` , `NUMPAD_ADD` , `NUMPAD_SUBTRACT`
- `OPEN_BRACKET` , `CLOSED_BRACKET`
- `SEMICOLON_FIREFOX` , `COLON` , `COMMA_FIREFOX_WINDOWS` , `COMMA_FIREFOX` , `BRACKET_RIGHT_FIREFOX` , `BRACKET_LEFT_FIREFOX`

Now that we have a key assigned to a variable we can now determine if that key has been pressed.
In the **update** function add the following:



```
if(left.isDown)
{
  ship.x=ship.x-1;
}
```

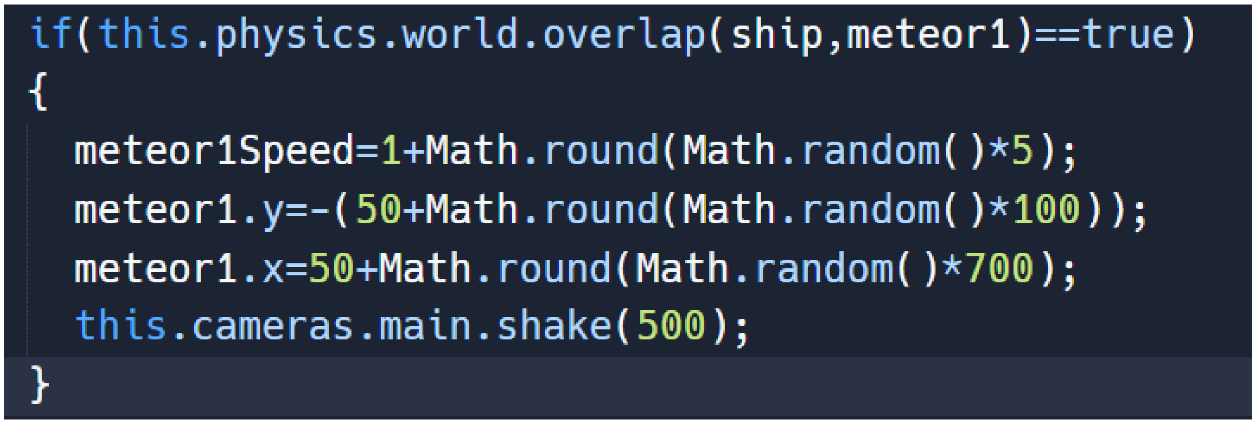
Explain below how this code works.

Exercises

1. **Add the variables for the right, up and down keys. Assign each variable their appropriate key. In the update function add the necessary code to move the ship in all directions.**
2. **Modify your code so that the ship's speed is variable i.e. create a variable to store its speed and use that variable in place of 1.**
3. **Add code so that the ship is placed on the opposite side of the screen when it moves off another side.**
4. **You have two other ships. Add the necessary code so that those ships are moveable using another set of keys i.e. W,A,S,D and I,J,K,L.**

Collisions

You should now have a lot of moving parts in this game. There has to be a purpose to any game. Just moving around without a challenge gets boring real quick. We'll add that challenge by using collisions. The purpose of the game is to avoid getting hit by any of the asteroids or other ships! Collision code is fairly simple. Add the following code to your game:



```
if(this.physics.world.overlap(ship,meteor1)==true)
{
  meteor1Speed=1+Math.round(Math.random()*5);
  meteor1.y=-(50+Math.round(Math.random()*100));
  meteor1.x=50+Math.round(Math.random()*700);
  this.cameras.main.shake(500);
}
```

Explain what happens when the ship collides with meteor1 below:

this.physics.world.overlap(ship,meteor1)

This function requires 2 sprites. In this case it's the ship and meteor1. It determines if they overlap and if so it will generate a true value otherwise it will be false.

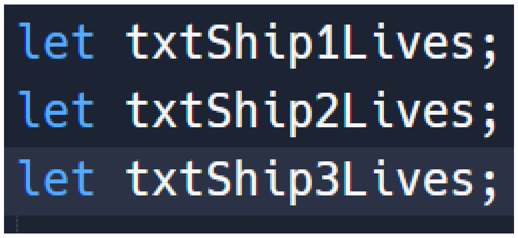
You can use this function, along with an if statement to see whether or not any two sprites collide and when collision occurs you decide what to do.

Exercises:

1. **Check for collisions between all ships and all asteroids. Upon collisions reposition the asteroids to be off the screen but make sure they move their way back on. Rather than shake the camera on each collision try the following effect on some of the collisions:**
   this.cameras.main.flash();

Text

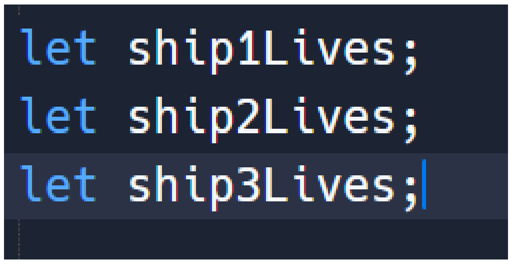
We have to have a way of displaying the game status to the player or players. For example we might want to show the current scores, level, lives, inventory etc. Usually we do this through text. In this game we're working on we'll need to show each player how many lives they have. The first person to be left with at least 1 life wins the game! As always, we'll need variables to hold our text objects.



```
let txtShip1Lives;
let txtShip2Lives;
let txtShip3Lives;
```

Note that I preceded the names with txt to help me remember that these variables will hold the text objects while the next three will hold the actual number of lives each ship has.

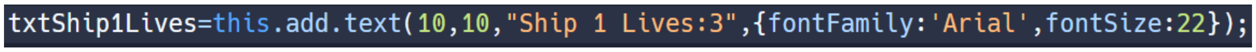
Add 3 other variables to hold the lives of the three ships.



```
let ship1Lives;
let ship2Lives;
let ship3Lives;
```

Initialize each to hold the value (for 3 lives). Do this in the **create** function.

Let's now create and assign each text object to the variables meant to hold them. The following is an example of how to assign the first text object.



```
txtShip1Lives=this.add.text(10,10,"Ship 1 Lives:3",{fontFamily:'Arial',fontSize:22});
```

There's a lot of info here. Let's go over it in some detail. It should be obvious that we are adding a text object to the screen and assigning it to the variable. The first two numbers indicate the coordinates of where to place the text. The third is the actual string it should hold and display. Then we set the font family (there are some common ones you can use…see https://jordanm.co.uk/tinytype/ for a list that should work). The final value is the size of the font.
 To update the text when the game is running is very easy. Find the if statement that check when the first ship collides with any asteroid and add the following code:

```
ship1Lives--;
txtShip1Lives.text="Ship 1 Lives:"+ship1Lives;
```

Test this by intentionally colliding the ship with the asteroid.

Questions and Exercises:

1. **What does ship1Lives– do?**
    a. **ship1Lives-- deducts 1 life from ship 1**
2. **Explain the difference between the variables *ship1Lives* and *txtShip1Lives*.**
    a. **txShip1Lives is a spring rather than a number like ship1Lives**
3. **Update your code so that you check for collisions between all ships and all asteroids and deduct lives accordingly as well as make sure to display the status on the screen in the appropriate locations.**
    a.

Sound Effects
Games are just not the same without sounds. Let's add some sounds. As always, we need variables to store the sounds. Add a variable called **explosion**. In the preload function we will need to load the sound file into memory (very similar to what we did for sprite images).

```
this.load.audio("explosion",["assets/effects/explosion-01.mp3"]);
```

In the **create** function assign the sound by using the id or name you used in the previous code. In our case we named the sound file '**explosion**'.

```
explosion=this.sound.add("explosion");
```

Now the easiest part, playing the sound requires you to call the play function on the variable holding the sound.

```
explosion.play();
```

The best place to place this code is in the if statement detecting collisions.

Exercises

1. **Update your code to add the explosion sound effect for all meteor to ship collisions.**

2. **Using the internet (https://freesound.org/) find, download and upload 3 new sound effects for explosions. Make sure they're no longer than a few seconds. Vary the sound effects played. The best type of sound files are mp3 but you should be able to use wav and ogg files as well.**
3. **Modify your code so that when each ship loses their lives you make them disappear. When 2 of 3 ships are gone you use a new text object to indicate that the game is over and let the players know who won the game!**

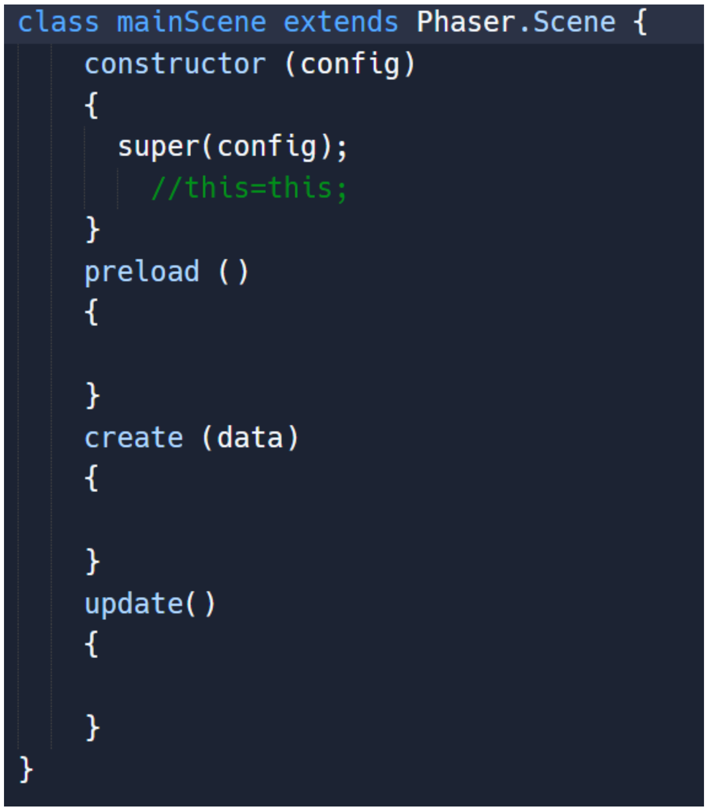
Miscellaneous

There are so many, many, many more things we can learn to do. We could probably spend an entire course just on game programming. Of course we don't have that time so the next best thing is to use what you have learned in this course and try to apply it to learning new techniques/concepts. You can find lots of examples of Phaser techniques at https://phaser.io/examples.

I will try and cover some additional things here. They're not necessary to make a game (as you could see from this tutorial). They're here to help you do more.

Game Scenes

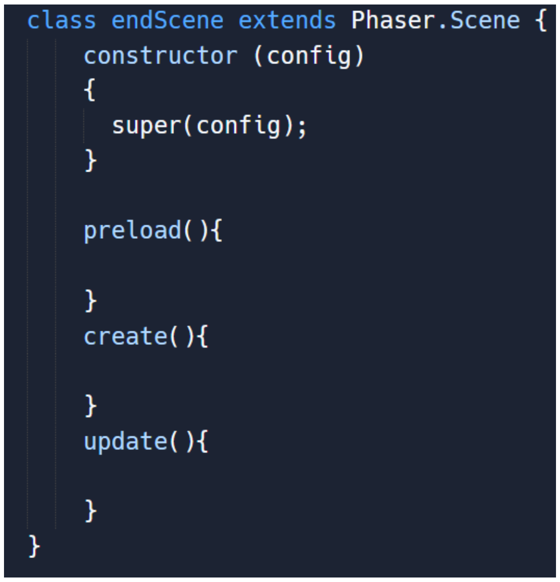
Each screen in a game in phaser is called a **scene** and is enclosed in something called a class:



```
class mainScene extends Phaser.Scene {
    constructor (config)
    {
      super(config);
        //this=this;
    }
    preload ()
    {

    }
    create (data)
    {

    }
    update()
    {

    }
}
```

We made our whole game in the mainScene class.

You should have noticed that we had another scene called endScene:



```
  class endScene extends Phaser.Scene {
      constructor (config)
      {
        super(config);
      }

      preload(){

      }
      create(){

      }
      update(){

      }
  }
```

We never used this. It could have been used to add other scenes with new levels, help screens with options to be selected, high score scenes and so on. Whenever you add a scene, empty or not, you need to add it to the game:

```
game.scene.add("game",mainScene);
game.scene.add("endScene",endScene);
game.scene.start("game");
```

Notice this code (found at the bottom of your code file). Here we add each scene and give it a name. We then start the game by starting the mainScene. We could have started the endScene as well. If you ever needed to switch screens then do the following:

```
game.scene.start("endScene");
game.scene.remove("game");
```

Remove the current scene after you start the new scene.

Exercises:

1. Fork the project at https://replit.com/@MrKatSDSS/sdssPhaserTemplateForStudents. Name it **phaserScenes**. In the first scene draw text with the message “**Game Scene**”. Perform a **keypress** check for the **space bar** and when this happens start the second screen i.e. **endScene** and remove the current or **gameScene**. Add a text message in the endScene “**End Scene**”.
2. Add three more scenes. Check for keypress A,S and D. When A is pressed show one scene, S the other and the D the last new scene. Add text messages to each scene with the name of the scene that was displayed.

Animations

Animations can be played using a number of different techniques in Phaser. We’ll look at one of them. Look at our spritesheet: **exp2_0.png** (you can download it from here if you don’t have it: https://opengameart.org/sites/default/files/exp2_0.png) . Find the total number of frames across and down. Divide the total width by the number of frames to get the frame width and do the same for the frame height. Using this information, load the spritesheet into memory:

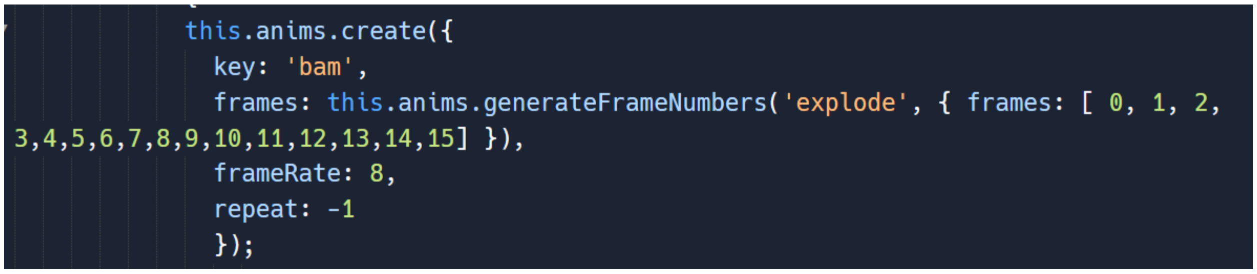
```
                this.load.spritesheet('explode', 'assets/sprites/exp2_0.png', { frameWidth:
64, frameHeight: 64 });
```

Don’t forget to make a variable and assign the spritesheet to it as follows:

```
    explode1 = this.add.sprite(600, 370);
```

This adds an empty sprite object at position 600,370.

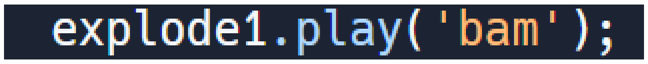
You could add the sprite when you need it. It doesn't have to be in the create function. If you do this however you may see the first frame. In the create function we need to create an animation out of the spritesheet and assign it some properties:



```
            this.anims.create({
              key: 'bam',
              frames: this.anims.generateFrameNumbers('explode', { frames: [ 0, 1, 2,
3,4,5,6,7,8,9,10,11,12,13,14,15] }),
              frameRate: 8,
              repeat: -1
              });
```

The name assigned is 'bam'. The spritesheet assigned is 'explode'. The frame to play (left to right from the top left) are 0 to 16 (there are only 16 frames and the first always numbered 0). The speed to play the frames is set at 8 and -1 tells us to repeat forever (use another number to limit this). Note that you can create as many of these named animations as you want. Just make sure to give each a unique key.

Finally, playing the animation is simple. You attach it to the sprite object you made and tell it which key to play:



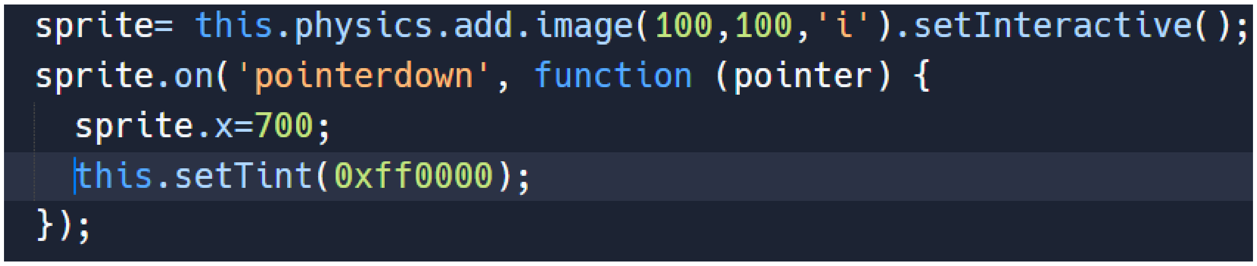
```
explode1.play('bam');
```

Exercises:

1. Fork the project at https://replit.com/@MrKatSDSS/sdssPhaserTemplateForStudents. Name it **phaserAnimations**. Create 4 animations baked on the **alien.png** file. One for each direction the alien is walking. Check for keypress W (walking away) ,A(left),S(towards) and D(right). Add to your code so that the alien actually moves in that direction and stops when the key is released.

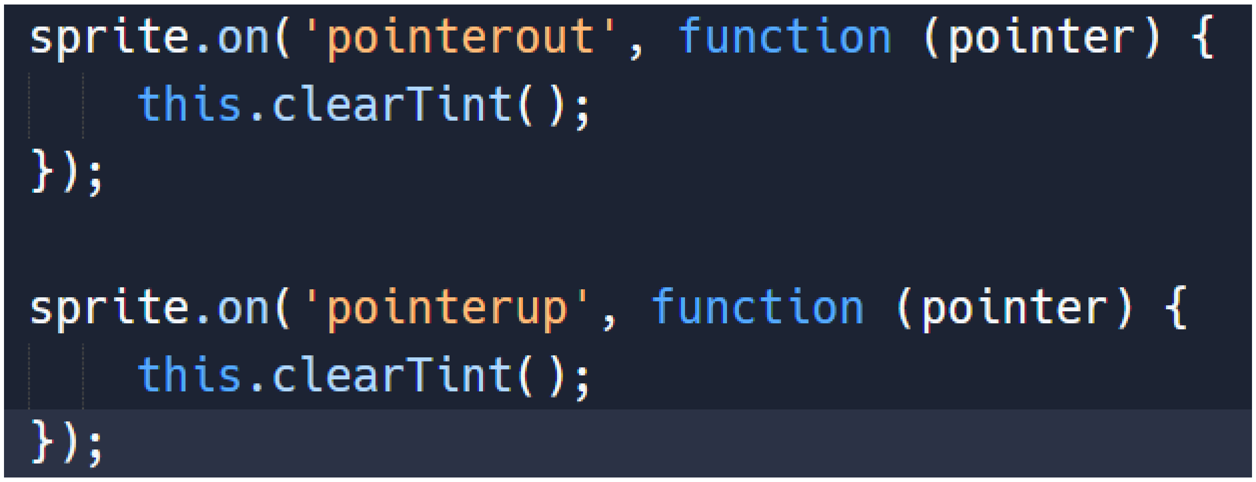
Click a Sprite

Setting up mouse events for sprites is easy. After you add a sprite in the create function you can add and set up a response to different events as follows (note how we add the **setInteractive** function at the end…this allows the sprite to interact with the mouse):



```
sprite= this.physics.add.image(100,100,'i').setInteractive();
sprite.on('pointerdown', function (pointer) {
  sprite.x=700;
  this.setTint(0xff0000);
});
```

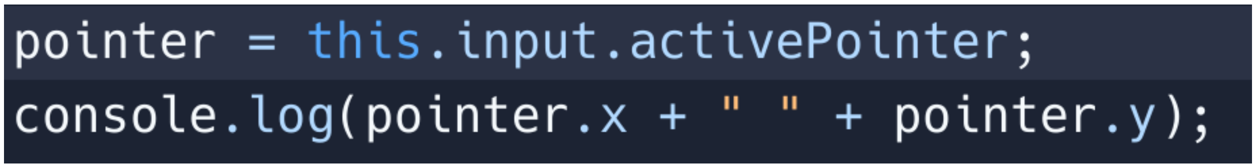
In this example once the mouse is clicked the sprite is moved to the x coordinate 700 and its colour is tinted. You could clear the tint when you release the mouse button or move it out of the sprite:



```
sprite.on('pointerout', function (pointer) {
    this.clearTint();
});

sprite.on('pointerup', function (pointer) {
    this.clearTint();
});
```

Mouse Location

To get the mouse pointer location you'll need a variable to store the object that contains that information. Call it **pointer**. In the update add the following code:



```
pointer = this.input.activePointer;
console.log(pointer.x + " " + pointer.y);
```

You can now check the console window where you will find the current mouse pointer's x and y coordinates in the game screen.

Exercises:

1. Fork the project at https://replit.com/@MrKatSDSS/sdssPhaserTemplateForStudents. Name it **phaserMouseEvents**. Add three sprites, all of **flare.png**, across the screen. When each sprite is clicked change their tint and reposition them in a random location.
2. In the same project add the **ship1_.png**. As the mouse moves, have the sprite follow it.

3. Go to https://photonstorm.github.io/phaser3-docs/Phaser.Input.Pointer.html. This page has all the information you can obtain from the activePointer. Write down and explain three other pieces of information that might be useful.

Camera

You can have complete control over the cameras in your game. To fade in and out a camera use the following lines of code:

```
this.cameras.main.fadeIn(6000);
this.cameras.main.fadeOut(6000);
```

6000 is the time it takes to fade in or out.
You can also get the camera to follow a sprite. For example you could write:

```
this.cameras.main.startFollow(ship);
```

Doing this would make the camera follow the ship sprite.

More info on the abilities of the cameras can be found here:
https://photonstorm.github.io/phaser3-docs/Phaser.Cameras.Scene2D.Camera.html

Exercises:

1. Modify the last project so that when the F key is pressed you fadeOut the screen. When the E key is pressed you fade back in.
2. Add a meteor to the last project. Move it across the screen. Have the camera follow the moving meteor.

More Sprites

There's a whole more you can do with sprites. To find many of these visit:
https://photonstorm.github.io/phaser3-docs/Phaser.GameObjects.Sprite.html

Let's try and use some of the information on this page.

```
m = this.physics.add.image(200, 200, 'm');
m.setDisplaySize(100,100);
```

This obviously sets the width and height of the sprite. This is easier than creating a whole new image using an image editor. There may be side effects, like accuracy of collisions, so be careful here.

```
m.angle=m.angle+1;
```

This allows you to set the current angle in degrees of a sprite. It can be useful if you want to rotate them to any angle.

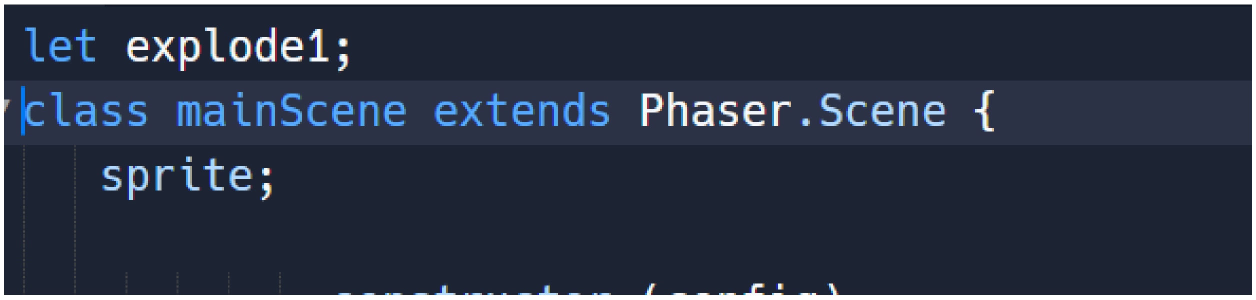
m.depth=2;
This sets the order in which the sprite is drawn relative to the others. Default is set to 0. A 1 one is drawn below a 2 and so on.
m.height
m.width
This is useful if you need to know the true unscaled width and height of the sprite.
Exercise

1. Research the web page with the docs on the sprite and provide a description of two other functions and/or properties of a sprite.

Advanced Features

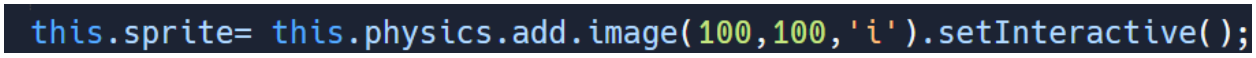
In this following I'll be demonstrating how to use some advanced programming techniques i.e. arrays and functions. This is primarily aimed at my grade 11 students but if you are interested in learning more then continue onwards.

Scope
Remember the rule: narrow your scope and then widen it if necessary. As a result we'll avoid global variables as much as possible. Notice the two scoped variables in the following:



```
let explode1;
class mainScene extends Phaser.Scene {
    sprite;
```

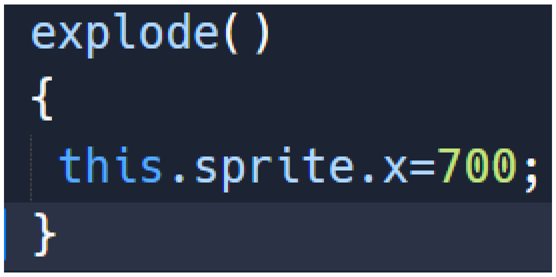
explode1 is global and sprite is **local** to the scene. We could and probably should have **explode1** more local. Notice as well that if you declare a variable in the class you do not use let. You simply name the variable (this is part of OOP which you'll learn more about in grade 12). Another difference is that you need to access a class variable using the **this** keyword.



```
this.sprite= this.physics.add.image(100,100,'i').setInteractive();
```

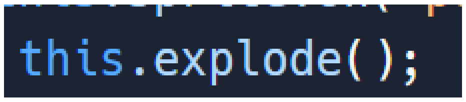
If you make a local variable i.e. one declared in a function than you do not need to precede the variable with **this**.

Functions
Any functions that are needed in a scene can be made in the scene. The only difference is that the function keyword is not needed.



```
explode()
{
  this.sprite.x=700;
}
```

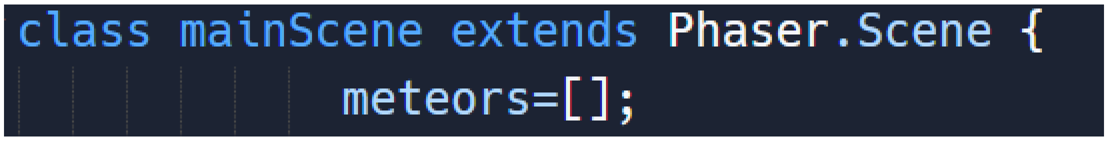
However because the function belongs to the scene i.e. in a class then to call it you must use the **this** keyword.



```
this.explode();
```

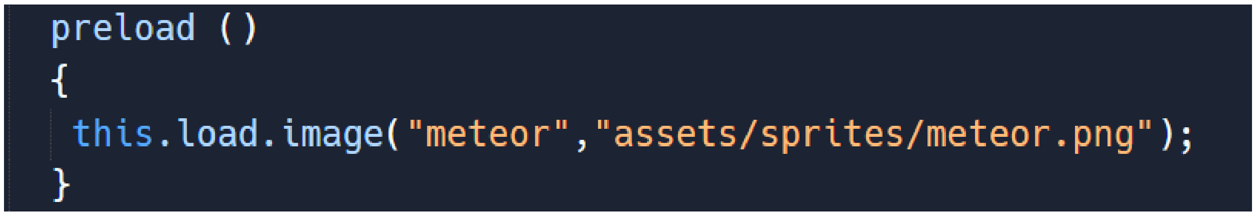
Functions and Arrays and Scope

Make a meteors array to hold all our meteors:



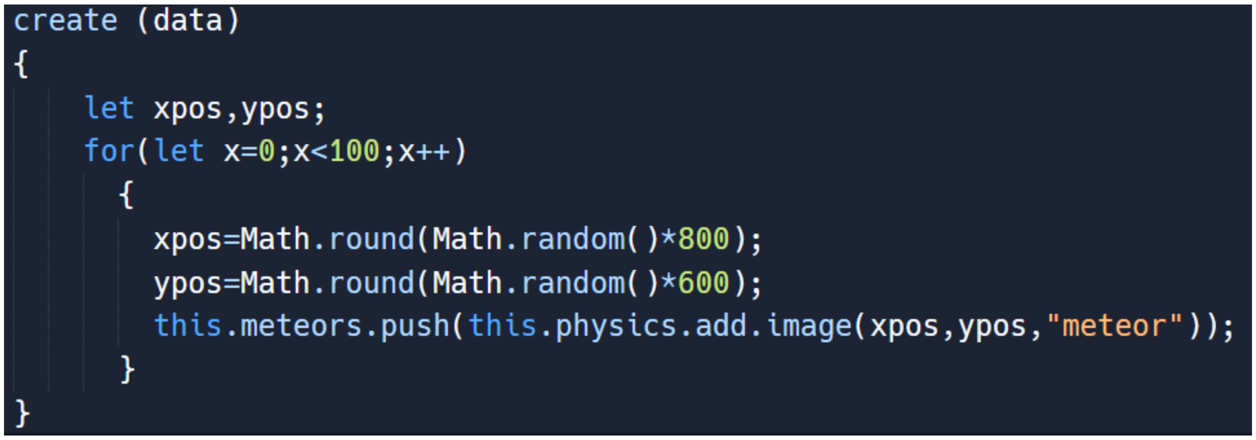
```
class mainScene extends Phaser.Scene {
                meteors=[];
```

Preload the image:



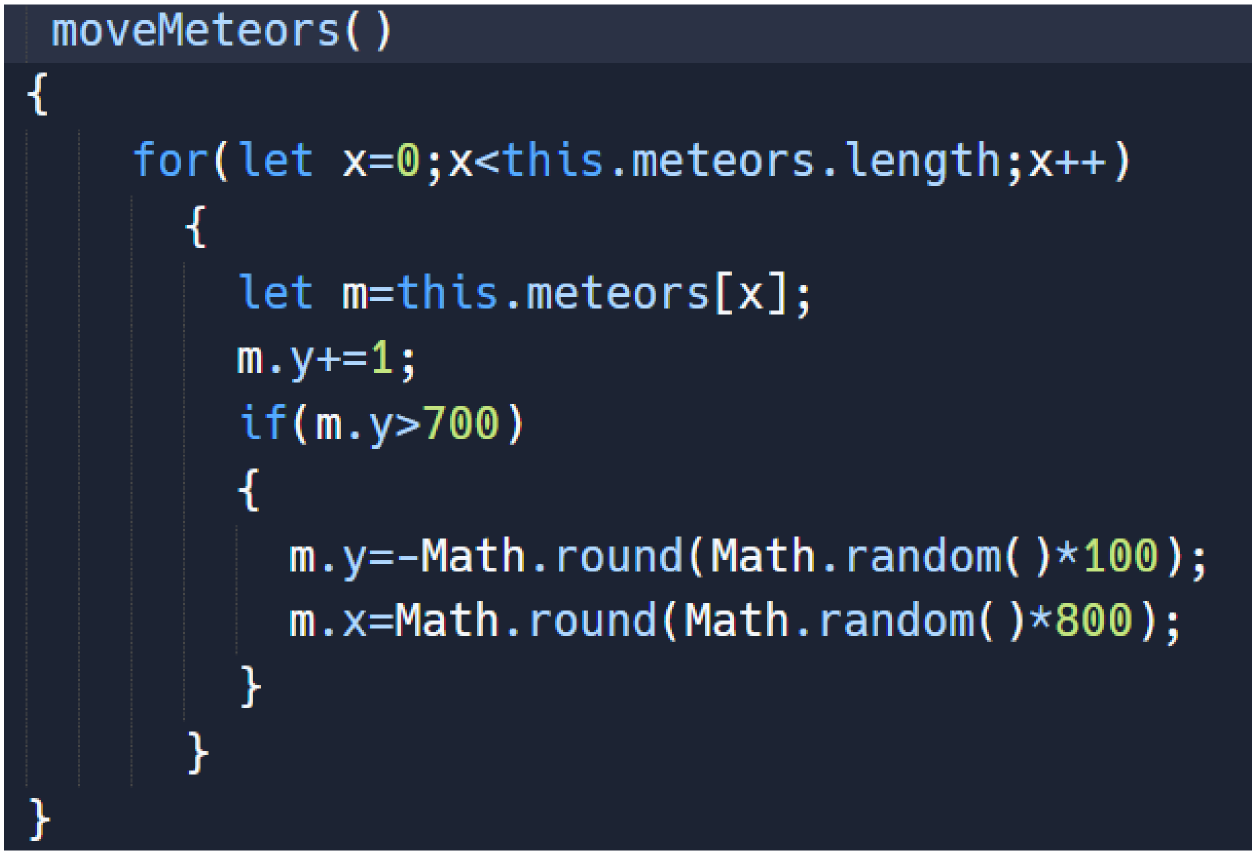
```
preload ()
{
 this.load.image("meteor","assets/sprites/meteor.png");
}
```

Fill the array:



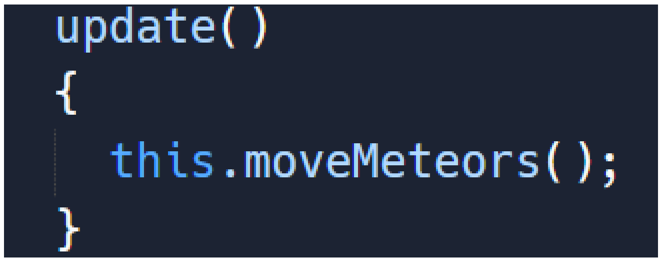
```
create (data)
{
    let xpos,ypos;
    for(let x=0;x<100;x++)
      {
        xpos=Math.round(Math.random()*800);
        ypos=Math.round(Math.random()*600);
        this.meteors.push(this.physics.add.image(xpos,ypos,"meteor"));
      }
}
```

Create a function to move the meteors:



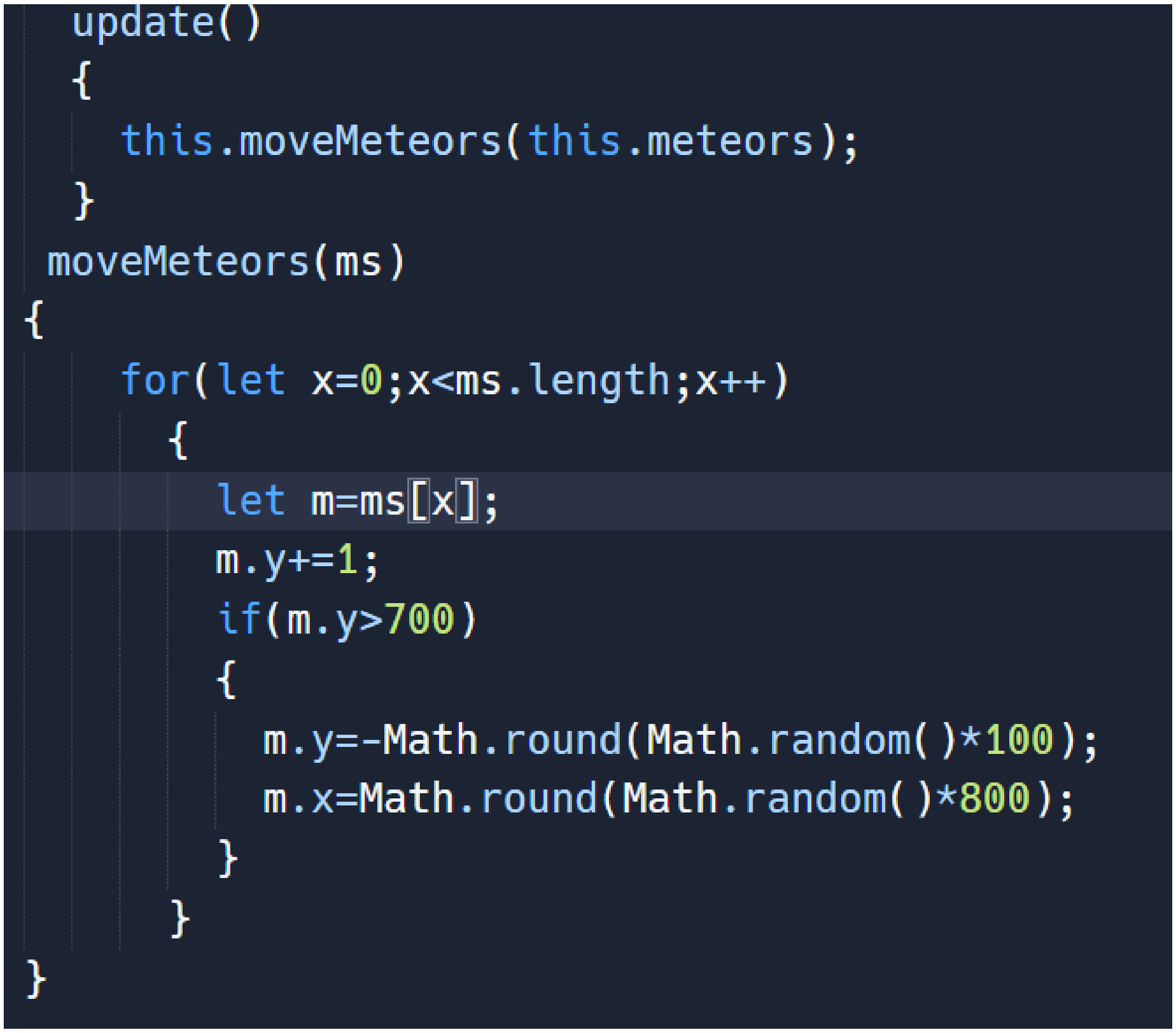
```
moveMeteors()
{
    for(let x=0;x<this.meteors.length;x++)
      {
        let m=this.meteors[x];
        m.y+=1;
        if(m.y>700)
        {
          m.y=-Math.round(Math.random()*100);
          m.x=Math.round(Math.random()*800);
        }
      }
}
```

Actually move all the meteors:



```
update()
{
  this.moveMeteors();
}
```

Do the same thing with a parameter:



```
  update()
  {
    this.moveMeteors(this.meteors);
  }
 moveMeteors(ms)
{
    for(let x=0;x<ms.length;x++)
      {
        let m=ms[x];
        m.y+=1;
        if(m.y>700)
        {
          m.y=-Math.round(Math.random()*100);
          m.x=Math.round(Math.random()*800);
        }
      }
}
```