

A step-by-step introduction to making games and interactive media with the Pixi.js rendering engine.

174 commits

1 branch

0 packages

0 releases

42 contributors

Branch: master


New pull request

Create new file

Upload files

Find file

Clone or download

 kittykatattack Merge pull request #141 from nikolas/fix-typos ...

Latest commit 345ecd0 on Mar 1

examples	Fix typos	2 months ago
pixi	Added pixi min file	3 years ago
.gitmodules	setup directory structure	6 years ago
README.md	Fix typos	2 months ago

README.md

# Learning Pixi

A step-by-step introduction to making games and interactive media with the [Pixi rendering engine](#). **Updated for Pixi v4.5.5.**  
[Chinese version here: Pixi官方教程中文版](#). If you like this tutorial, [you'll love the book](#), which contains 80% more content!.

## Table of contents

- [Introduction](#)
- [Setting up](#)
- [Installing Pixi](#)
- [Creating the stage and renderer](#)
- [Pixi sprites](#)
- [Loading images into the texture cache](#)
- [Displaying sprites](#)
- [Using Aliases](#)
- [A little more about loading things](#) 1. [Make a sprite from an ordinary JavaScript Image object or Canvas](#) 2. [Assigning a name to a loaded file](#) 3. [Monitoring load progress](#) 4. [More about Pixi's loader](#)
- [Positioning sprites](#)
- [Size and scale](#)
- [Rotation](#)
- [Make a sprite from a tileset sub-image](#)
- [Using a texture atlas](#)
- [Loading the texture atlas](#)
- [Creating sprites from a loaded texture atlas](#)
- [Moving Sprites](#)
- [Using velocity properties](#)
- [Game states](#)
- [Keyboard Movement](#)
- [Grouping Sprites](#)
- [Local and global positions](#)
- [Using a ParticleContainer to group sprites](#)
- [Pixi's Graphic Primitives](#)
- [Rectangle](#)
- [Circles](#)
- [Ellipses](#)

<https://github.com/kittykatattack/learningPixi>

1/49

- 28. [Rounded rectangles](#)
- 29. [Lines](#)
- 30. [Polygons](#)
- 31. [Displaying text](#)
- 32. [Collision detection](#)
- 33. [The hitTestRectangle function](#)
- 34. [Case study: Treasure Hunter](#)
- 35. [Initialize the game in the setup function](#) 1. [Creating the game scenes](#) 2. [Making the dungeon, door, explorer and treasure](#) 3. [Making the blob monsters](#) 4. [Making health bar](#) 5. [Making message text](#)
- 36. [Playing the game](#)
- 37. [Moving the explorer](#) 1. [Containing movement](#)
- 38. [Moving the monsters](#)
- 39. [Checking for collisions](#)
- 40. [Reaching the exit door and ending game](#)
- 41. [More about sprites](#)
- 42. [Taking it further](#)
  - i. [Hexi](#)
  - ii. [BabylonJS](#)
- 43. [Supporting this project](#)

## Introduction

Pixi is an extremely fast 2D sprite rendering engine. What does that mean? It means that it helps you to display, animate and manage interactive graphics so that it's easy for you to make games and applications using JavaScript and other HTML5 technologies. It has a sensible, uncluttered API and includes many useful features, like supporting texture atlases and providing a streamlined system for animating sprites (interactive images). It also gives you a complete scene graph so that you can create hierarchies of nested sprites (sprites inside sprites), as well as letting you attach mouse and touch events directly to sprites. And, most importantly, Pixi gets out of your way so that you can use as much or as little of it as you want to, adapt it to your personal coding style, and integrate it seamlessly with other useful frameworks.

Pixi's API is actually a refinement of a well-worn and battle-tested API pioneered by Macromedia/Adobe Flash. Old-skool Flash developers will feel right at home. Other current sprite rendering frameworks use a similar API: CreateJS, Starling, Sparrow and Apple's SpriteKit. The strength of Pixi's API is that it's general-purpose: it's not a game engine. That's good because it gives you total expressive freedom to make anything you like, and wrap your own custom game engine around it.

In this tutorial you're going to find out how to combine Pixi's powerful image rendering features and scene graph to start making games. But Pixi isn't just for games - you can use these same techniques to create any interactive media applications. That means apps for phones!

What do you need to know before you get started with this tutorial?

You should have a reasonable understanding of HTML and JavaScript. You don't have to be an expert, just an ambitious beginner with an eagerness to learn. If you don't know HTML and JavaScript, the best place to start learning it is this book:

[Foundation Game Design with HTML5 and JavaScript](#)

I know for a fact that it's the best book, because I wrote it!

There are also some good internet resources to help get you started:

[Khan Academy: Computer Programming](#)

[Code Academy: JavaScript](#)

Choose whichever best suits your learning style.

Ok, got it? Do you know what JavaScript variables, functions, arrays and objects are and how to use them? Do you know what [JSON data files](#) are? Have you used the [Canvas Drawing API](#)?

To use Pixi, you'll also need to run a webserver in your root project directory. Do you know what a webserver is and how to launch one in your project folder? The best way is to use [node.js](#) and then to install the extremely easy to use [http-server](#). However, you need to be comfortable working with the Unix command line if you want to do that. You can learn how to use Unix [in this video](#) and, when you're finished, follow it with [this video](#). You should learn how to use Unix - it only takes a couple of hours to learn and is a really fun and easy way to interact with your computer.

But if you don't want to mess around with the command line just yet, try the Mongoose webserver:

### Mongoose

Or, just write all your code using the excellent [Brackets text editor](#). Brackets automatically launches a webserver and browser for you when you click the lightning bolt button in its main workspace.

Now if you think you're ready, read on!

(Request to readers: this is a *living document*. If you have any questions about specific details or need any of the content clarified, please create an [issue](#) in this GitHub repository and I'll update the text with more information.)

## Setting up

Before you start writing any code, create a folder for your project, and launch a webserver in the project's root directory. If you aren't running a webserver, Pixi won't work.

Next, you need to install Pixi.

### Installing Pixi

The version used for this introduction is **v4.5.5** and you can find the `pixi.min.js` file either in this repository's `pixi` folder or on [Pixi's release page for v4.5.5](#). Or, you can get the latest version from [Pixi's main release page](#).

This one file is all you need to use Pixi. You can ignore all the other files in the repository: **you don't need them**.

Next, create a basic HTML page, and use a `<script>` tag to link the `pixi.min.js` file that you've just downloaded. The `<script>` tag's `src` should be relative to your root directory where your webserver is running. Your `<script>` tag might look something like this:

```
<script src="pixi.min.js"></script>
```

Here's a basic HTML page that you could use to link Pixi and test that it's working. (This assumes that the `pixi.min.js` is in a subfolder called `pixi`):

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>Hello World</title>
</head>
<script src="pixi/pixi.min.js"></script>
<body>
  <script type="text/javascript">
    let type = "WebGL"
    if(!PIXI.utils.isWebGLSupported()){
      type = "canvas"
    }

    PIXI.utils.sayHello(type)
  </script>
</body>
</html>
```

If Pixi is linking correctly, something like this will be displayed in your web browser's JavaScript console by default:

```
PixiJS 4.4.5 - * canvas * http://www.pixijs.com/ ♥♥♥
```

## Creating the Pixi Application and stage

Now you can start using Pixi!

But how?

The first step is to create a rectangular display area that you can start displaying images on. Pixi has a `Application` object that creates this for you. It automatically `generates an HTML <canvas> element` and figures out how to display your images on the canvas. You then need to create a special Pixi `Container` object called the `stage`. As you'll see ahead, this `stage` object is going to be used as the root container that holds all the things you want Pixi to display.

Here's the code you need to write to create an `app` Pixi Application and `stage`. Add this code to your HTML document between the `<script>` tags:

```
//Create a Pixi Application
let app = new PIXI.Application({width: 256, height: 256});

//Add the canvas that Pixi automatically created for you to the HTML document
document.body.appendChild(app.view);
```

This is the most basic code you need write to get started using Pixi. It produces a black 256 pixel by 256 pixel canvas element and adds it to your HTML document. Here's what this looks like in a browser when you run this code.



Yay, a `black square`!

`PIXI.Application` figures out whether to use the Canvas Drawing API or WebGL to render graphics, depending on which is available on the web browser you're using. Its argument is a single object called the `options` object. In this example its `width` and `height` properties are set to determine the width and height of the canvas, in pixels. You can set many more optional properties inside this `options` object; here's how you could use it to set anti-aliasing, transparency and resolution:

```
let app = new PIXI.Application({
  width: 256,      // default: 800
  height: 256,    // default: 600
  antialias: true, // default: false
  transparent: false, // default: false
  resolution: 1   // default: 1
});
```

If you're happy with Pixi's default settings, you don't need to set any of these options. But, if you need to, see Pixi's documentation on [PIXI.Application](#).

What do those options do? `antialias` smoothes the edges of fonts and graphic primitives. (WebGL anti-aliasing isn't available on all platforms, so you'll need to test this on your game's target platform.) `transparent` makes the canvas background transparent. `resolution` makes it easier to work with displays of varying resolutions and pixel densities. Setting the resolutions is a little outside the scope of this tutorial, but check out [Mat Grove's explanation](#) about how to use `resolution` for all the details. But usually, just keep `resolution` at 1 for most projects and you'll be fine.

Pixi's `renderer` object will default to WebGL, which is good, because WebGL is incredibly fast, and lets you use some spectacular visual effects that you'll learn all about ahead. But if you need to force the Canvas Drawing API rendering over WebGL, you can set the `forceCanvas` option to `true`, like this:

```
forceCanvas: true,
```

If you need to change the background color of the canvas after you've created it, set the `app.renderer` object's `backgroundColor` property to any hexadecimal color value:

```
app.renderer.backgroundColor = 0x061639;
```

If you want to find the width or the height of the `renderer`, use `app.renderer.view.width` and `app.renderer.view.height`.

To change the size of the canvas, use the `renderer`'s `resize` method, and supply any new `width` and `height` values. But, to make sure the canvas is resized to match the resolution, set `autoResize` to `true`.

```
app.renderer.autoResize = true;
app.renderer.resize(512, 512);
```

If you want to make the canvas fill the entire window, you can apply this CSS styling and resize the `renderer` to the size of the browser window.

```
app.renderer.view.style.position = "absolute";
app.renderer.view.style.display = "block";
app.renderer.autoResize = true;
app.renderer.resize(window.innerWidth, window.innerHeight);
```

But, if you do that, make sure you also set the default padding and margins to 0 on all your HTML elements with this bit of CSS code:

```
<style>* {padding: 0; margin: 0}</style>
```

(The asterisk, `*`, in the code above, is the CSS "universal selector", which just means "all the tags in the HTML document".)

If you want the canvas to scale proportionally to any browser window size, you can use [this custom `scaleToWindow` function](#).

## Pixi sprites

Now that you have a `renderer`, you can start adding images to it. Anything you want to be made visible in the `renderer` has to be added to a special Pixi object called the `stage`. You can access this special `stage` object like this:

```
app.stage
```

The `stage` is a Pixi `Container` object. You can think of a container as a kind of empty box that will group together and store whatever you put inside it. The `stage` object is the root container for all the visible things in your scene. Whatever you put inside the `stage` will be rendered on the canvas. Right now the `stage` is empty, but soon we're going to start putting things inside it. (You can read more about Pixi's `Container` objects [here](#)).

(Important: because the `stage` is a Pixi `Container` it has the same properties and methods as any other `Container` object. But, although the `stage` has `width` and `height` properties, *they don't refer to the size of the rendering window*. The `stage`'s `width` and `height` properties just tell you the area occupied by the things you put inside it - more on that ahead!)

So what do you put on the stage? Special image objects called **sprites**. Sprites are basically just images that you can control with code. You can control their position, size, and a host of other properties that are useful for making interactive and animated graphics. Learning to make and control sprites is really the most important thing about learning to use Pixi. If you know how to make sprites and add them to the stage, you're just a small step away from starting to make games.

Pixi has a `Sprite` class that is a versatile way to make game sprites. There are three main ways to create them:

- From a single image file.
- From a sub-image on a **tileset**. A tileset is a single, big image that includes all the images you'll need in your game.
- From a **texture atlas** (A JSON file that defines the size and position of an image on a tileset.)

You're going to learn all three ways, but, before you do, let's find out what you need to know about images before you can display them with Pixi.

## Loading images into the texture cache

Because Pixi renders the image on the GPU with WebGL, the image needs to be in a format that the GPU can process. A WebGL-ready image is called a **texture**. Before you can make a sprite display an image, you need to convert an ordinary image file into a WebGL texture. To keep everything working fast and efficiently under the hood, Pixi uses a **texture cache** to store and reference all the textures your sprites will need. The names of the textures are strings that match the file locations of the images they refer to. That means if you have a texture that was loaded from `"images/cat.png"`, you could find it in the texture cache like this:

```
PIXI.utils.TextureCache["images/cat.png"];
```

The textures are stored in a WebGL compatible format that's efficient for Pixi's renderer to work with. You can then use Pixi's `Sprite` class to make a new sprite using the texture.

```
let texture = PIXI.utils.TextureCache["images/anySpriteImage.png"];
let sprite = new PIXI.Sprite(texture);
```

But how do you load the image file and convert it into a texture? Use Pixi's built-in `loader` object.

Pixi's powerful `loader` object is all you need to load any kind of image. Here's how to use it to load an image and call a function called `setup` when the image has finished loading:

```
PIXI.loader
  .add("images/anyImage.png")
  .load(setup);

function setup() {
  //This code will run when the loader has finished loading the image
}
```

Pixi's development team recommends that if you use the loader, you should create the sprite by referencing the texture in the loader's `resources` object, like this:

```
let sprite = new PIXI.Sprite(
  PIXI.loader.resources["images/anyImage.png"].texture
);
```

Here's an example of some complete code you could write to load an image, call the `setup` function, and create a sprite from the loaded image:

```
PIXI.loader
  .add("images/anyImage.png")
  .load(setup);

function setup() {
  let sprite = new PIXI.Sprite(
    PIXI.loader.resources["images/anyImage.png"].texture
  );
}
```

This is the general format we'll be using to load images and create sprites in this tutorial.

You can load multiple images at the same time by listing them with chainable `add` methods, like this:

```
PIXI.loader
  .add("images/imageOne.png")
  .add("images/imageTwo.png")
  .add("images/imageThree.png")
  .load(setup);
```

Better yet, just list all the files you want to load in an array inside a single `add` method, like this:

```
PIXI.loader
  .add([
    "images/imageOne.png",
    "images/imageTwo.png",
    "images/imageThree.png"
  ])
  .load(setup);
```

```
  })  
  .load(setup);
```

The `loader` also lets you load JSON files, which you'll learn about ahead.

## Displaying sprites

After you've loaded an image, and used it to make a sprite, you need to add the sprite to Pixi's `stage` with the `stage.addChild` method, like this:

```
app.stage.addChild(cat);
```

Remember that the `stage` is the main container that holds all of your sprites.

**Important:** you won't be able to see any of your sprites unless you add them to the `stage` !

Before we continue, let's look at a practical example of how to use what you've just learnt to display a single image. In the `examples/images` folder you'll find a 64 by 64 pixel PNG image of a cat.



Here's all the JavaScript code you need to load the image, create a sprite, and display it on Pixi's stage:

```
//Create a Pixi Application  
let app = new PIXI.Application({  
  width: 256,  
  height: 256,  
  antialias: true,  
  transparent: false,  
  resolution: 1  
})  
);  
  
//Add the canvas that Pixi automatically created for you to the HTML document  
document.body.appendChild(app.view);  
  
//load an image and run the `setup` function when it's done  
PIXI.loader  
  .add("images/cat.png")  
  .load(setup);  
  
//This `setup` function will run when the image has loaded  
function setup() {  
  
  //Create the cat sprite  
  let cat = new PIXI.Sprite(PIXI.loader.resources["images/cat.png"].texture);  
  
  //Add the cat to the stage  
  app.stage.addChild(cat);  
}
```

When this code runs, here's what you'll see:



Now we're getting somewhere!

If you ever need to remove a sprite from the stage, use the `removeChild` method:

```
app.stage.removeChild(anySprite)
```

But usually setting a sprite's `visible` property to `false` will be a simpler and more efficient way of making sprites disappear.

```
anySprite.visible = false;
```

## Using aliases

You can save yourself a little typing and make your code more readable by creating short-form aliases for the Pixi objects and methods that you use frequently. For example, is prefixing `PIXI` to all of Pixi's objects starting to bog you down? If you think so, create a shorter alias that points to it. For example, here's how you can create an alias to the `TextureCache` object:

```
let TextureCache = PIXI.utils.TextureCache
```

Then, use that alias in place of the original, like this:

```
let texture = TextureCache["images/cat.png"];
```

In addition to letting you write more slightly succinct code, using aliases has an extra benefit: it helps to buffer you from Pixi's frequently changing API. If Pixi's API changes in future versions - which it will! - you just need to update these aliases to Pixi objects and methods in one place, at the beginning of your program, instead of every instance where they're used throughout your code. So when Pixi's development team decides they want to rearrange the furniture a bit, you'll be one step ahead of them!

To see how to do this, let's re-write the code we wrote to load an image and display it, using aliases for all the Pixi objects and methods.

```
//Aliases
let Application = PIXI.Application,
    loader = PIXI.loader,
    resources = PIXI.loader.resources,
    Sprite = PIXI.Sprite;

//Create a Pixi Application
let app = new Application({
    width: 256,
    height: 256,
    antialias: true,
    transparent: false,
    resolution: 1
});

//Add the canvas that Pixi automatically created for you to the HTML document
document.body.appendChild(app.view);

//load an image and run the `setup` function when it's done
loader
    .add("images/cat.png")
    .load(setup);

//This `setup` function will run when the image has loaded
function setup() {

    //Create the cat sprite
    let cat = new Sprite(resources["images/cat.png"].texture);

    //Add the cat to the stage
    app.stage.addChild(cat);
}
```

Most of the examples in this tutorial will use aliases for Pixi objects that follow this same model. **Unless otherwise stated, you can assume that all the code examples that follow use aliases like these.**



This is all you need to know to start loading images and creating sprites.

## A little more about loading things

The format I've shown you above is what I suggest you use as your standard template for loading images and displaying sprites. So, you can safely ignore the next few paragraphs and jump straight to the next section, "Positioning sprites." But Pixi's `loader` object is quite sophisticated and includes a few features that you should be aware of, even if you don't use them on a regular basis. Let's look at some of the most useful.

### Make a sprite from an ordinary JavaScript Image object or Canvas

For optimization and efficiency it's always best to make a sprite from a texture that's been pre-loaded into Pixi's texture cache. But if for some reason you need to make a texture from a regular JavaScript `Image` object, you can do that using Pixi's `BaseTexture` and `Texture` classes:

```
let base = new PIXI.BaseTexture(anyImageObject),
    texture = new PIXI.Texture(base),
    sprite = new PIXI.Sprite(texture);
```

You can use `BaseTexture.fromCanvas` if you want to make a texture from any existing canvas element:

```
let base = new PIXI.BaseTexture.fromCanvas(anyCanvasElement),
```

If you want to change the texture the sprite is displaying, use the `texture` property. Set it to any `Texture` object, like this:

```
anySprite.texture = PIXI.utils.TextureCache["anyTexture.png"];
```

You can use this technique to interactively change the sprite's appearance if something significant happens to it in the game.

### Assigning a name to a loading file

It's possible to assign a unique name to each resource you want to load. Just supply the name (a string) as the first argument in the `add` method. For example, here's how to name an image of a cat as `catImage`.

```
PIXI.loader
  .add("catImage", "images/cat.png")
  .load(setup);
```

This creates an object called `catImage` in `loader.resources`. That means you can create a sprite by referencing the `catImage` object, like this:

```
let cat = new PIXI.Sprite(PIXI.loader.resources.catImage.texture);
```

However, I recommend you don't use this feature! That's because you'll have to remember all names you gave each loaded files, as well as make sure you don't accidentally use the same name more than once. Using the file path name, as we've done in previous examples is simpler and less error prone.

### Monitoring load progress

Pixi's loader has a special `progress` event that will call a customizable function that will run each time a file loads. `progress` events are called by the `loader`'s `on` method, like this:

```
PIXI.loader.on("progress", loadProgressHandler);
```

Here's how to include the `on` method in the loading chain, and call a user-definable function called `loadProgressHandler` each time a file loads.

```
PIXI.loader
  .add([
    "images/one.png",
    "images/two.png",
    "images/three.png"
```

```

    ])
    .on("progress", loadProgressHandler)
    .load(setup);

function loadProgressHandler() {
    console.log("loading");
}

function setup() {
    console.log("setup");
}

```

Each time one of the files loads, the progress event calls `loadProgressHandler` to display "loading" in the console. When all three files have loaded, the `setup` function will run. Here's the output of the above code in the console:

```

loading
loading
loading
setup

```

That's neat, but it gets better. You can also find out exactly which file has loaded and what percentage of overall files are have currently loaded. You can do this by adding optional `loader` and `resource` parameters to the `loadProgressHandler`, like this:

```
function loadProgressHandler(loader, resource) { /*...*/ }
```

You can then use `resource.url` to find the file that's currently loaded. (Use `resource.name` if you want to find the optional name that you might have assigned to the file, as the first argument in the `add` method.) And you can use `loader.progress` to find what percentage of total resources have currently loaded. Here's some code that does just that.

```

PIXI.loader
    .add([
        "images/one.png",
        "images/two.png",
        "images/three.png"
    ])
    .on("progress", loadProgressHandler)
    .load(setup);

function loadProgressHandler(loader, resource) {

    //Display the file `url` currently being loaded
    console.log("loading: " + resource.url);

    //Display the percentage of files currently loaded
    console.log("progress: " + loader.progress + "%");

    //If you gave your files names as the first argument
    //of the `add` method, you can access them like this
    //console.log("loading: " + resource.name);
}

function setup() {
    console.log("All files loaded");
}

```

Here's what this code will display in the console when it runs:

```

loading: images/one.png
progress: 33.33333333333336%
loading: images/two.png
progress: 66.6666666666667%
loading: images/three.png
progress: 100%
All files loaded

```

That's really cool, because you could use this as the basis for creating a loading progress bar.

(Note: There are additional properties you can access on the `resource` object. `resource.error` will tell you of any possible error that happened while trying to load a file. `resource.data` lets you access the file's raw binary data.)

### More about Pixi's loader

Pixi's loader is ridiculously feature-rich and configurable. Let's take a quick bird's-eye view of its usage to get you started.

The loader's chainable `add` method takes 4 basic arguments:

```
add(name, url, optionObject, callbackFunction)
```

Here's what the loader's source code documentation has to say about these parameters:

`name` (string): The name of the resource to load. If it's not passed, the `url` is used.

`url` (string): The url for this resource, relative to the `baseURL` of the loader.

`options` (object literal): The options for the load.

**`options.crossOrigin` (Boolean): Is the request cross-origin? The default is to determine automatically.**

`options.loadType` : How should the resource be loaded? The default value is `Resource.LOAD_TYPE.XHR`.

`options.xhrType` : How should the data being loaded be interpreted when using XHR? The default value is `Resource.XHR_RESPONSE_TYPE.DEFAULT`

`callbackFunction` : The function to call when this specific resource completes loading.

The only one of these arguments that's required is the `url` (the file that you want to load.)

Here are some examples of some ways you could use the `add` method to load files. These first ones are what the docs call the loader's "normal syntax":

```
.add('key', 'http://...', function () {})  
.add('http://...', function () {})  
.add('http://...')
```

And these are examples of the loader's "object syntax":

```
.add({  
  name: 'key2',  
  url: 'http://...'  
}, function () {})  
  
.add({  
  url: 'http://...'  
}, function () {})  
  
.add({  
  name: 'key3',  
  url: 'http://...'  
  onComplete: function () {}  
})  
  
.add({  
  url: 'https://...',  
  onComplete: function () {},  
  crossOrigin: true  
})
```

You can also pass the `add` method an array of objects, or urls, or both:

```
.add([  
  {name: 'key4', url: 'http://...', onComplete: function () {} },  
  {url: 'http://...', onComplete: function () {} },  
  'http://...'  
]);
```

(Note: If you ever need to reset the loader to load a new batch of files, call the loader's `reset` method: `PIXI.loader.reset();`)

Pixi's loader has many more advanced features, including options to let you load and parse binary files of all types. This is not something you'll need to do on a day-to-day basis, and way outside the scope of this tutorial, so [make sure to check out the loader's GitHub repository for more information](#).

## Positioning sprites

Now that you know how to create and display sprites, let's find out how to position and resize them.

In the earlier example the cat sprite was added to the stage at the top left corner. The cat has an `x` position of 0 and a `y` position of 0. You can change the position of the cat by changing the values of its `x` and `y` properties. Here's how you can center the cat in the stage by setting its `x` and `y` property values to 96.

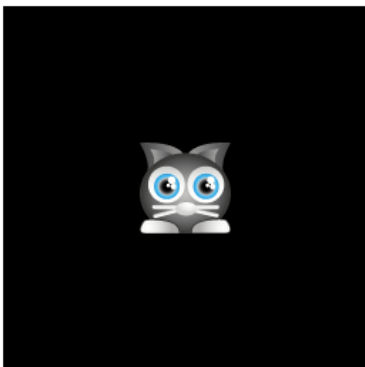
```
cat.x = 96;  
cat.y = 96;
```

Add these two lines of code anywhere inside the `setup` function, after you've created the sprite.

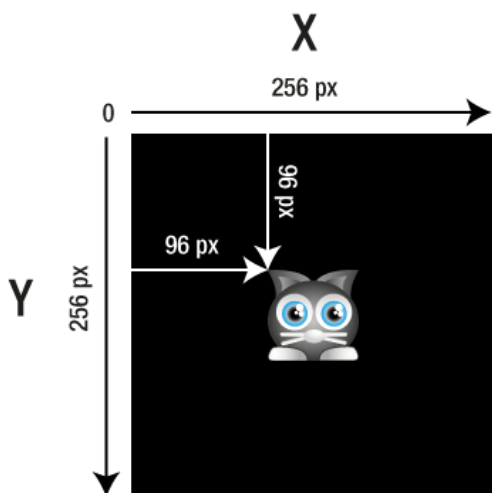
```
function setup() {  
  
  //Create the `cat` sprite  
  let cat = new Sprite(resources["images/cat.png"].texture);  
  
  //Change the sprite's position  
  cat.x = 96;  
  cat.y = 96;  
  
  //Add the cat to the stage so you can see it  
  app.stage.addChild(cat);  
}
```

(Note: In this example, `Sprite` is an alias for `PIXI.Sprite`, `TextureCache` is an alias for `PIXI.utils.TextureCache`, and `resources` is an alias for `PIXI.loader.resources` as described earlier. I'll be using aliases that follow this same format for all Pixi objects and methods in the example code from now on.)

These two new lines of code will move the cat 96 pixels to the right, and 96 pixels down. Here's the result:



The cat's top left corner (its left ear) represents its `x` and `y` anchor point. To make the cat move to the right, increase the value of its `x` property. To make the cat move down, increase the value of its `y` property. If the cat has an `x` value of 0, it will be at the very left side of the stage. If it has a `y` value of 0, it will be at the very top of the stage.



Instead of setting the sprite's `x` and `y` properties independently, you can set them together in a single line of code, like this:

```
sprite.position.set(x, y)
```

## Size and scale

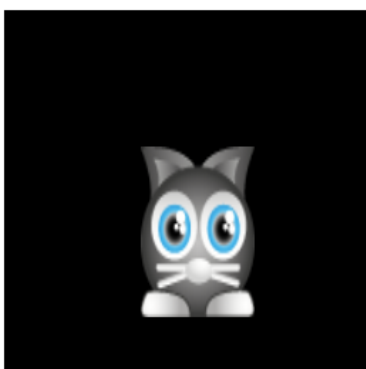
You can change a sprite's size by setting its `width` and `height` properties. Here's how to give the cat a `width` of 80 pixels and a `height` of 120 pixels.

```
cat.width = 80;  
cat.height = 120;
```

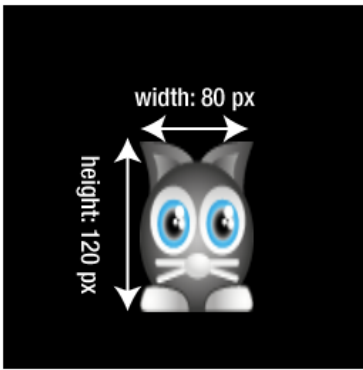
Add those two lines of code to the `setup` function, like this:

```
function setup() {  
  
  //Create the `cat` sprite  
  let cat = new Sprite(resources["images/cat.png"].texture);  
  
  //Change the sprite's position  
  cat.x = 96;  
  cat.y = 96;  
  
  //Change the sprite's size  
  cat.width = 80;  
  cat.height = 120;  
  
  //Add the cat to the stage so you can see it  
  app.stage.addChild(cat);  
}
```

Here's the result:



You can see that the cat's position (its top left corner) didn't change, only its width and height.



Sprites also have `scale.x` and `scale.y` properties that change the sprite's width and height proportionately. Here's how to set the cat's scale to half size:

```
cat.scale.x = 0.5;  
cat.scale.y = 0.5;
```

Scale values are numbers between 0 and 1 that represent a percentage of the sprite's size. 1 means 100% (full size), while 0.5 means 50% (half size). You can double the sprite's size by setting its scale values to 2, like this:

```
cat.scale.x = 2;  
cat.scale.y = 2;
```

Pixi has an alternative, concise way for you set sprite's scale in one line of code using the `scale.set` method.

```
cat.scale.set(0.5, 0.5);
```

If that appeals to you, use it!

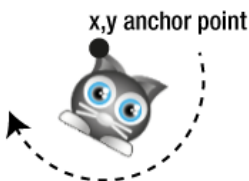
## Rotation

You can make a sprite rotate by setting its `rotation` property to a value in [radians](#).

```
cat.rotation = 0.5;
```

But around which point does that rotation happen?

You've seen that a sprite's top left corner represents its `x` and `y` position. That point is called the **anchor point**. If you set the sprite's `rotation` property to something like `0.5`, the rotation will happen *around the sprite's anchor point*. This diagram shows what effect this will have on our cat sprite.

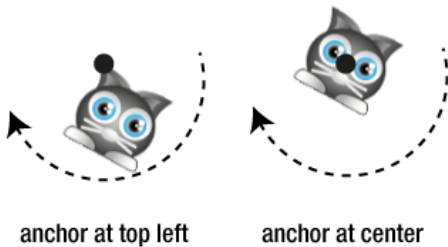


You can see that the anchor point, the cat's left ear, is the center of the imaginary circle around which the cat is rotating. What if you want the sprite to rotate around its center? Change the sprite's `anchor` point so that it's centered inside the sprite, like this:

```
cat.anchor.x = 0.5;  
cat.anchor.y = 0.5;
```

The `anchor.x` and `anchor.y` values represent a percentage of the texture's dimensions, from 0 to 1 (0% to 100%). Setting it to 0.5 centers the texture over the point. The location of the point itself won't change, just the way the texture is positioned over it.

This next diagram shows what happens to the rotated sprite if you center its anchor point.



You can see that the sprite's texture shifts up and to the left. This is an important side-effect to remember!

Just like with `position` and `scale`, you can set the anchor's x and y values with one line of code like this:

```
cat.anchor.set(x, y)
```

Sprites also have a `pivot` property which works in a similar way to `anchor`. `pivot` sets the position of the sprite's x/y origin point. If you change the pivot point and then rotate the sprite, it will rotate around that origin point. For example, the following code will set the sprite's `pivot.x` point to 32, and its `pivot.y` point to 32

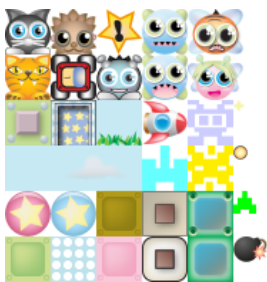
```
cat.pivot.set(32, 32)
```

Assuming that the sprite is 64x64 pixels, the sprite will now rotate around its center point. But remember: if you change a sprite's pivot point, you've also changed its x/y origin point.

So, what's the difference between `anchor` and `pivot`? They're really similar! `anchor` shifts the origin point of the sprite's image texture, using a 0 to 1 normalized value. `pivot` shifts the origin of the sprite's x and y point, using pixel values. Which should you use? It's up to you. Just play around with both of them and see which you prefer.

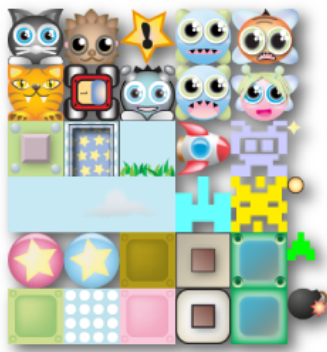
## Make a sprite from a tileset sub-image

You now know how to make a sprite from a single image file. But, as a game designer, you'll usually be making your sprites using **tilesets** (also known as **spritesheets**.) Pixi has some convenient built-in ways to help you do this. A tileset is a single image file that contains sub-images. The sub-images represent all the graphics you want to use in your game. Here's an example of a tileset image that contains game characters and game objects as sub-images.

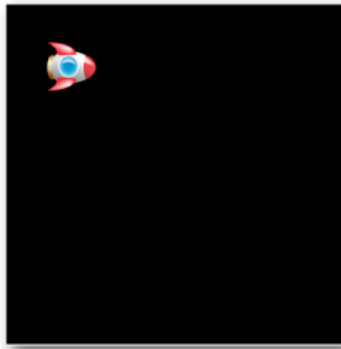


The entire tileset is 192 by 192 pixels. Each image is in its own 32 by 32 pixel grid cell. Storing and accessing all your game graphics on a tileset is a very processor and memory efficient way to work with graphics, and Pixi is optimized for this.

You can capture a sub-image from a tileset by defining a rectangular area that's the same size and position as the sub-image you want to extract. Here's an example of the rocket sub-image that's been extracted from the tileset.



tileset.png



canvas

Let's look at the code that does this. First, load the `tileset.png` image with Pixi's `loader`, just as you've done in earlier examples.

```
loader
  .add("images/tileset.png")
  .load(setup);
```

Next, when the image has loaded, use a rectangular sub-section of the tileset to create the sprite's image. Here's the code that extracts the sub image, creates the rocket sprite, and positions and displays it on the canvas.

```
function setup() {

  //Create the `tileset` sprite from the texture
  let texture = TextureCache["images/tileset.png"];

  //Create a rectangle object that defines the position and
  //size of the sub-image you want to extract from the texture
  //(`Rectangle` is an alias for `PIXI.Rectangle`)
  let rectangle = new Rectangle(192, 128, 64, 64);

  //Tell the texture to use that rectangular section
  texture.frame = rectangle;

  //Create the sprite from the texture
  let rocket = new Sprite(texture);

  //Position the rocket sprite on the canvas
  rocket.x = 32;
  rocket.y = 32;

  //Add the rocket to the stage
  app.stage.addChild(rocket);

  //Render the stage
  app.renderer.render(app.stage);
}
```

How does this work?

Pixi has a built-in `Rectangle` object (`PIXI.Rectangle`) that is a general-purpose object for defining rectangular shapes. It takes four arguments. The first two arguments define the rectangle's `x` and `y` position. The last two define its `width` and `height`. Here's the format for defining a new `Rectangle` object.

```
let rectangle = new PIXI.Rectangle(x, y, width, height);
```

The rectangle object is just a *data object*; it's up to you to decide how you want to use it. In our example we're using it to define the position and area of the sub-image on the tileset that we want to extract. **Pixi textures have a useful property called `frame` that can be set to any `Rectangle` objects. The `frame` crops the texture to the dimensions of the `Rectangle`.** Here's how to use `frame` to crop the texture to the size and position of the rocket.

```
let rectangle = new Rectangle(192, 128, 64, 64);
texture.frame = rectangle;
```



You can then use that cropped texture to create the sprite:

```
let rocket = new Sprite(texture);
```

And that's how it works!

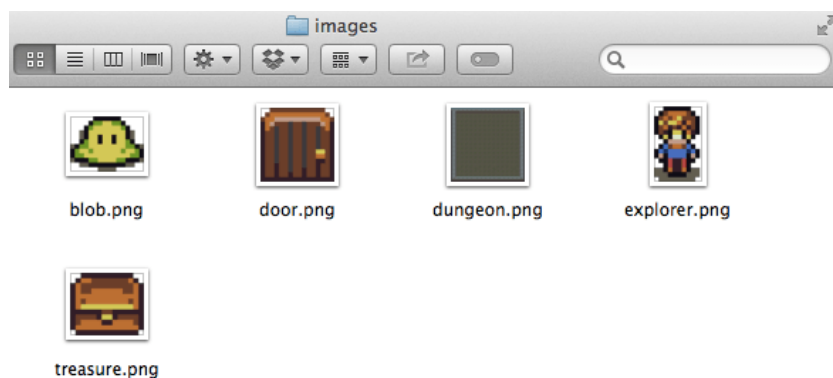
Because making sprite textures from a tileset is something you'll do with great frequency, Pixi has a more convenient way to help you do this - let's find out what that is next.

## Using a texture atlas

If you're working on a big, complex game, you'll want a fast and efficient way to create sprites from tilesets. This is where a **texture atlas** becomes really useful. A texture atlas is a JSON data file that contains the positions and sizes of sub-images on a matching tileset PNG image. If you use a texture atlas, all you need to know about the sub-image you want to display is its name. You can arrange your tileset images in any order and the JSON file will keep track of their sizes and positions for you. This is really convenient because it means the sizes and positions of tileset images aren't hard-coded into your game program. If you make changes to the tileset, like adding images, resizing them, or removing them, just re-publish the JSON file and your game will use that data to display the correct images. You won't have to make any changes to your game code.

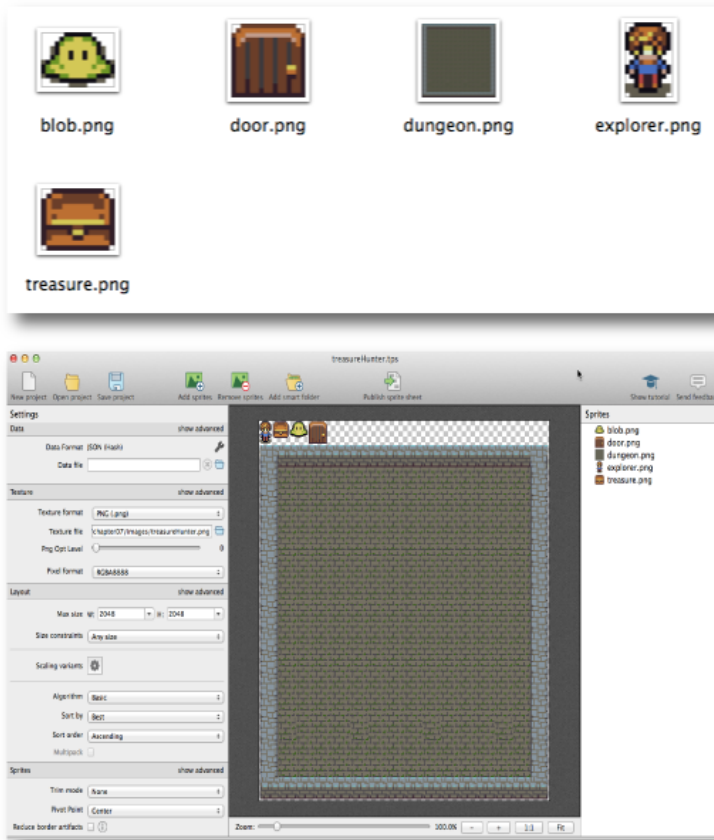
Pixi is compatible with a standard JSON texture atlas format that is output by a popular software tool called [Texture Packer](#). Texture Packer's "Essential" license is free. Let's find out how to use it to make a texture atlas, and load the atlas into Pixi. (You don't have to use Texture Packer. Similar tools, like [Shoebox](#) or [spritesheet.js](#), output PNG and JSON files in a standard format that is compatible with Pixi.)

First, start with a collection of individual image files that you'd like to use in your game.



(All the images in this section were created by Lanea Zimmerman. You can find more of her artwork [here](#). Thanks, Lanea!)

Next, open Texture Packer and choose **JSON Hash** as the framework type. Drag your images into Texture Packer's workspace. (You can also point Texture Packer to any folder that contains your images.) It will automatically arrange the images on a single tileset image, and give them names that match their original image names.



(If you're using the free version of Texture Packer, set **Algorithm** to **Basic**, set **Trim mode** to **None**, set **Extrude** to **0**, set **Size constraints** to **Any size** and slide the **PNG Opt Level** all the way to the left to **0**. These are the basic settings that will allow the free version of Texture Packer to create your files without any warnings or errors.)

When you're done, click the **Publish** button. Choose the file name and location, and save the published files. You'll end up with 2 files: a PNG file and a JSON file. In this example my file names are `treasureHunter.json` and `treasureHunter.png`. To make your life easier, just keep both files in your project's `images` folder. (You can think of the JSON file as extra metadata for the image file, so it makes sense to keep both files in the same folder.) The JSON file describes the name, size and position of each of the sub-images in the tileset. Here's an excerpt that describes the blob monster sub-image.

```
"blob.png":
{
  "frame": { "x":55,"y":2,"w":32,"h":24},
  "rotated": false,
  "trimmed": false,
  "spriteSourceSize": { "x":0,"y":0,"w":32,"h":24},
  "sourceSize": { "w":32,"h":24},
  "pivot": { "x":0.5,"y":0.5}
},
```

The `treasureHunter.json` file also contains `"dungeon.png"`, `"door.png"`, `"exit.png"`, and `"explorer.png"` properties each with similar data. Each of these sub-images are called **frames**. Having this data is really helpful because now you don't need to know the size and position of each sub-image in the tileset. All you need to know is the sprite's **frame id**. The frame id is just the name of the original image file, like `"blob.png"` or `"explorer.png"`.

Among the many advantages to using a texture atlas is that you can easily add 2 pixels of padding around each image (Texture Packer does this by default.) This is important to prevent the possibility of **texture bleed**. Texture bleed is an effect that happens when the edge of an adjacent image on the tileset appears next to a sprite. This happens because of the way your computer's GPU (Graphics Processing Unit) decides how to round fractional pixels values. Should it round them up or down? This will be different for each GPU. Leaving 1 or 2 pixels spacing around images on a tileset makes all images display consistently.

(Note: If you have two pixels of padding around a graphic, and you still notice a strange "off by one pixel" glitch in the way Pixi is displaying it, try changing the texture's scale mode algorithm. Here's how: `texture.baseTexture.scaleMode = PIXI.SCALE_MODES.NEAREST`; . These glitches can sometimes happen because of GPU floating point rounding errors.)

Now that you know how to create a texture atlas, let's find out how to load it into your game code.

## Loading the texture atlas

To get the texture atlas into Pixi, load it using Pixi's `loader`. If the JSON file was made with Texture Packer, the `loader` will interpret the data and create a texture from each frame on the tileset automatically. Here's how to use the `loader` to load the `treasureHunter.json` file. When it has loaded, the `setup` function will run.

```
loader
  .add("images/treasureHunter.json")
  .load(setup);
```

Each image on the tileset is now an individual texture in Pixi's cache. You can access each texture in the cache with the same name it had in Texture Packer ("blob.png", "dungeon.png", "explorer.png", etc.).

## Creating sprites from a loaded texture atlas

Pixi gives you three general ways to create a sprite from a texture atlas:

1. Using `TextureCache` :

```
let texture = TextureCache["frameId.png"],
    sprite = new Sprite(texture);
```

2. If you've used Pixi's `loader` to load the texture atlas, use the loader's `resources` :

```
let sprite = new Sprite(
  resources["images/treasureHunter.json"].textures["frameId.png"]
);
```

3. That's way too much typing to do just to create a sprite! So I suggest you create an alias called `id` that points to texture's atlas's `textures` object, like this:

```
let id = PIXI.loader.resources["images/treasureHunter.json"].textures;
```

Then you can just create each new sprite like this:

```
let sprite = new Sprite(id["frameId.png"]);
```

Much better!

Here's how you could use these three different sprite creation techniques in the `setup` function to create and display the `dungeon`, `explorer`, and `treasure` sprites.

```
//Define variables that might be used in more
//than one function
let dungeon, explorer, treasure, id;

function setup() {

  //There are 3 ways to make sprites from textures atlas frames

  //1. Access the `TextureCache` directly
  let dungeonTexture = TextureCache["dungeon.png"];
  dungeon = new Sprite(dungeonTexture);
  app.stage.addChild(dungeon);

  //2. Access the texture using through the loader's `resources`:
  explorer = new Sprite(
    resources["images/treasureHunter.json"].textures["explorer.png"]
  );
  explorer.x = 68;

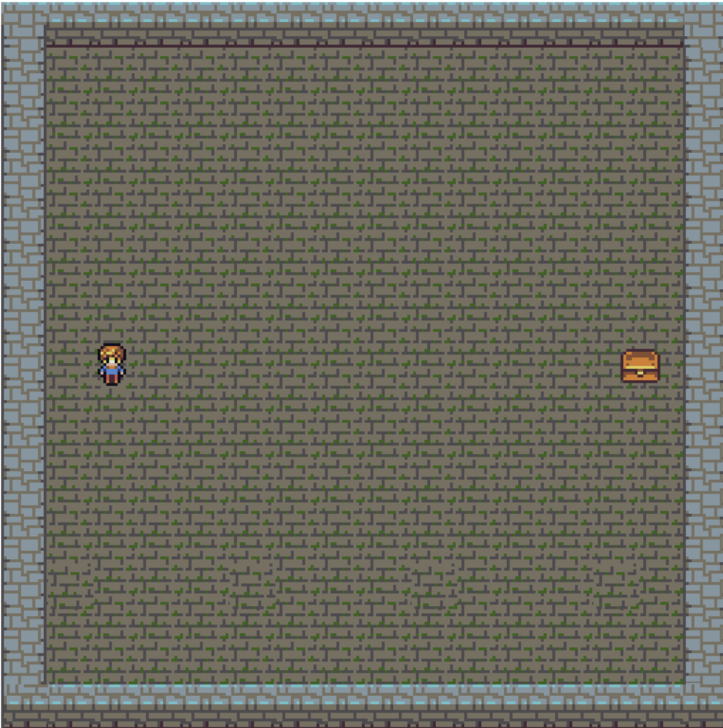
  //Center the explorer vertically
  explorer.y = app.stage.height / 2 - explorer.height / 2;
  app.stage.addChild(explorer);
```

```
//3. Create an optional alias called `id` for all the texture atlas
//frame id textures.
id = PIXI.loader.resources["images/treasureHunter.json"].textures;

//Make the treasure box using the alias
treasure = new Sprite(id["treasure.png"]);
app.stage.addChild(treasure);

//Position the treasure next to the right edge of the canvas
treasure.x = app.stage.width - treasure.width - 48;
treasure.y = app.stage.height / 2 - treasure.height / 2;
app.stage.addChild(treasure);
}
```

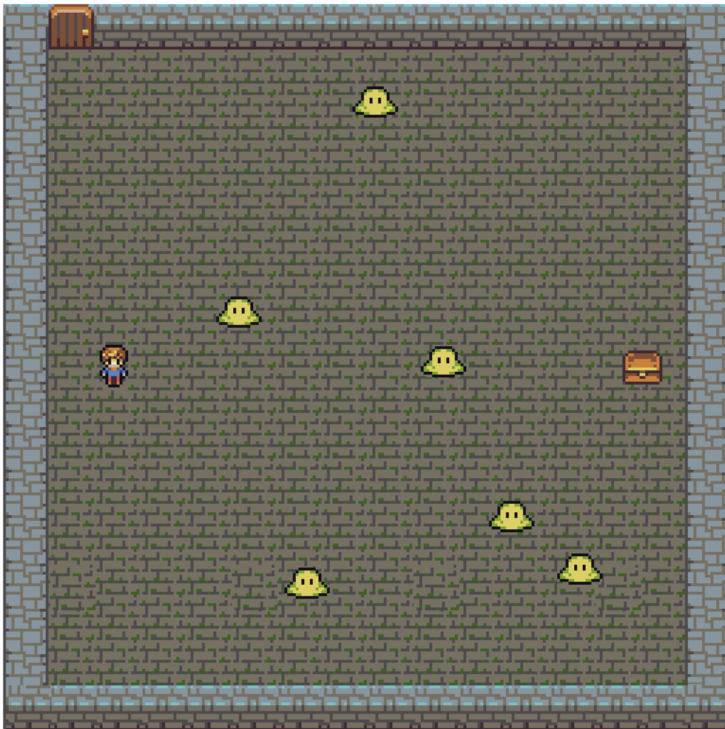
Here's what this code displays:



The stage dimensions are 512 by 512 pixels, and you can see in the code above that the `app.stage.height` and `app.stage.width` properties are used to align the sprites. Here's how the explorer's `y` position is vertically centered:

```
explorer.y = app.stage.height / 2 - explorer.height / 2;
```

Learning to create and display sprites using a texture atlas is an important benchmark. So before we continue, let's take a look at the code you could write to add the remaining sprites: the blob's and exit door, so that you can produce a scene that looks like this:



Here's the entire code that does all this. I've also included the HTML code so you can see everything in its proper context. (You'll find this working code in the `examples/spriteFromTextureAtlas.html` file in this repository.) Notice that the `blob` sprites are created and added to the stage in a loop, and assigned random positions.

```
<!doctype html>
<meta charset="utf-8">
<title>Make a sprite from a texture atlas</title>
<body>
<script src="../../pixi/pixi.min.js"></script>
<script>

//Aliases
let Application = PIXI.Application,
    Container = PIXI.Container,
    loader = PIXI.Loader,
    resources = PIXI.Loader.resources,
    TextureCache = PIXI.utils.TextureCache,
    Sprite = PIXI.Sprite,
    Rectangle = PIXI.Rectangle;

//Create a Pixi Application
let app = new Application({
    width: 512,
    height: 512,
    antialias: true,
    transparent: false,
    resolution: 1
});

//Add the canvas that Pixi automatically created for you to the HTML document
document.body.appendChild(app.view);

//load a JSON file and run the `setup` function when it's done
loader
    .add("images/treasureHunter.json")
    .load(setup);

//Define variables that might be used in more
//than one function
let dungeon, explorer, treasure, door, id;

function setup() {

    //There are 3 ways to make sprites from textures atlas frames

    //1. Access the `TextureCache` directly
```

```

let dungeonTexture = TextureCache["dungeon.png"];
dungeon = new Sprite(dungeonTexture);
app.stage.addChild(dungeon);

//2. Access the texture using through the loader's `resources`:
explorer = new Sprite(
  resources["images/treasureHunter.json"].textures["explorer.png"]
);
explorer.x = 68;

//Center the explorer vertically
explorer.y = app.stage.height / 2 - explorer.height / 2;
app.stage.addChild(explorer);

//3. Create an optional alias called `id` for all the texture atlas
//frame id textures.
id = PIXI.loader.resources["images/treasureHunter.json"].textures;

//Make the treasure box using the alias
treasure = new Sprite(id["treasure.png"]);
app.stage.addChild(treasure);

//Position the treasure next to the right edge of the canvas
treasure.x = app.stage.width - treasure.width - 48;
treasure.y = app.stage.height / 2 - treasure.height / 2;
app.stage.addChild(treasure);

//Make the exit door
door = new Sprite(id["door.png"]);
door.position.set(32, 0);
app.stage.addChild(door);

//Make the blobs
let numberOfBlobs = 6,
    spacing = 48,
    xOffset = 150;

//Make as many blobs as there are `numberOfBlobs`
for (let i = 0; i < numberOfBlobs; i++) {

  //Make a blob
  let blob = new Sprite(id["blob.png"]);

  //Space each blob horizontally according to the `spacing` value.
  //`xOffset` determines the point from the left of the screen
  //at which the first blob should be added.
  let x = spacing * i + xOffset;

  //Give the blob a random y position
  //(`randomInt` is a custom function - see below)
  let y = randomInt(0, app.stage.height - blob.height);

  //Set the blob's position
  blob.x = x;
  blob.y = y;

  //Add the blob sprite to the stage
  app.stage.addChild(blob);
}
}

//The `randomInt` helper function
function randomInt(min, max) {
  return Math.floor(Math.random() * (max - min + 1)) + min;
}
</script>
</body>

```

You can see in the code above that all the blobs are created using a `for` loop. Each `blob` is spaced evenly along the `x` axis like this:

```

let x = spacing * i + xOffset;
blob.x = x;

```

`spacing` has a value 48, and `xoffset` has a value of 150. What this means is the first `blob` will have an `x` position of 150. This offsets it from the left side of the stage by 150 pixels. Each subsequent `blob` will have an `x` value that's 48 pixels greater than the `blob` created in the previous iteration of the loop. This creates an evenly spaced line of blob monsters, from left to right, along the dungeon floor.

Each `blob` is also given a random `y` position. Here's the code that does this:

```
let y = randomInt(0, stage.height - blob.height);
blob.y = y;
```

The `blob`'s `y` position could be assigned any random number between 0 and 512, which is the value of `stage.height`. This works with the help of a custom function called `randomInt`. `randomInt` returns a random number that's within a range between any two numbers you supply.

```
randomInt(lowestNumber, highestNumber)
```

That means if you want a random number between 1 and 10, you can get one like this:

```
let randomNumber = randomInt(1, 10);
```

Here's the `randomInt` function definition that does all this work:

```
function randomInt(min, max) {
  return Math.floor(Math.random() * (max - min + 1)) + min;
}
```

`randomInt` is a great little function to keep in your back pocket for making games - I use it all the time.

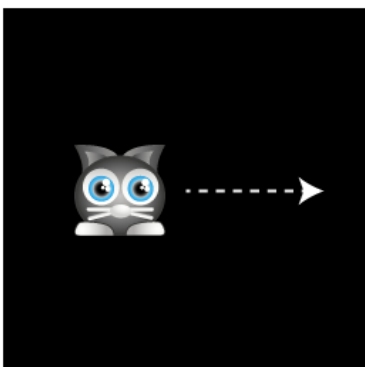
## Moving Sprites

You now know how to display sprites, but how do you make them move? That's easy: create a looping function using Pixi's `ticker`. This is called a **game loop**. Any code you put inside the game loop will update 60 times per second. Here's some code you could write to make the `cat` sprite move to the right at a rate of 1 pixel per frame.

```
function setup() {
  //Start the game loop by adding the `gameLoop` function to
  //Pixi's `ticker` and providing it with a `delta` argument.
  app.ticker.add(delta => gameLoop(delta));
}

function gameLoop(delta){
  //Move the cat 1 pixel
  cat.x += 1;
}
```

If you run this bit of code, you'll see the sprite gradually move to the right side of the stage.



That's because each time the `gameLoop` runs, it adds 1 to the cat's `x` position.

```
cat.x += 1;
```

Any function you add to Pixi's `ticker` will be called 60 times per second. You can see that the function is provided a `delta` argument - what's that?

The `delta` value represents the amount of fractional lag between frames. You can optionally add it to the cat's position, to make the cat's animation independent of the frame rate. Here's how:

```
cat.x += 1 + delta;
```

Whether or not you choose to add this `delta` value is largely an aesthetic choice. And the effect will only really be noticeable if your animation is struggling to keep up with a consistent 60 frames per second display rate (which might happen, for example, if it's running on a slow device). The rest of the examples in this tutorial won't use this `delta` value, but feel free to use it in your own work if you wish.

You don't have to use Pixi's ticker to create a game loop. If you prefer, just use `requestAnimationFrame`, like this:

```
function gameLoop() {
    //Call this `gameLoop` function on the next screen refresh
    //(which happens 60 times per second)
    requestAnimationFrame(gameLoop);

    //Move the cat
    cat.x += 1;
}

//Start the loop
gameLoop();
```

It's entirely up to you which style you prefer.

And that's really all there is to it! Just change any sprite property by small increments inside the loop, and they'll animate over time. If you want the sprite to animate in the opposite direction (to the left), just give it a negative value, like `-1`.

You'll find this code in the `movingSprites.html` file - here's the complete code:

```
//Aliases
let Application = PIXI.Application,
    Container = PIXI.Container,
    loader = PIXI.loader,
    resources = PIXI.loader.resources,
    TextureCache = PIXI.utils.TextureCache,
    Sprite = PIXI.Sprite,
    Rectangle = PIXI.Rectangle;

//Create a Pixi Application
let app = new Application({
    width: 256,
    height: 256,
    antialias: true,
    transparent: false,
    resolution: 1
});

//Add the canvas that Pixi automatically created for you to the HTML document
document.body.appendChild(app.view);

loader
    .add("images/cat.png")
    .load(setup);

//Define any variables that are used in more than one function
let cat;

function setup() {
    //Create the `cat` sprite
```



```

cat = new Sprite(resources["images/cat.png"].texture);
cat.y = 96;
app.stage.addChild(cat);

//Start the game loop
app.ticker.add(delta => gameLoop(delta));
}

function gameLoop(delta){

  //Move the cat 1 pixel
  cat.x += 1;

  //Optionally use the `delta` value
  //cat.x += 1 + delta;
}

```

(Notice that the `cat` variable needs to be defined outside the `setup` and `gameLoop` functions so that you can access it inside both of them.)

You can animate a sprite's scale, rotation, or size - whatever! You'll see many more examples of how to animate sprites ahead.

## Using velocity properties

To give you more flexibility, it's a good idea to control a sprite's movement speed using two **velocity properties**: `vx` and `vy`. `vx` is used to set the sprite's speed and direction on the x axis (horizontally). `vy` is used to set the sprite's speed and direction on the y axis (vertically). Instead of changing a sprite's `x` and `y` values directly, first update the velocity variables, and then assign those velocity values to the sprite. This is an extra bit of modularity that you'll need for interactive game animation.

The first step is to create `vx` and `vy` properties on your sprite, and give them an initial value.

```

cat.vx = 0;
cat.vy = 0;

```

Setting `vx` and `vy` to 0 means that the sprite isn't moving.

Next, in the game loop, update `vx` and `vy` with the velocity at which you want the sprite to move. Then assign those values to the sprite's `x` and `y` properties. Here's how you could use this technique to make the cat sprite move down and to right at one pixel each frame:

```

function setup() {

  //Create the `cat` sprite
  cat = new Sprite(resources["images/cat.png"].texture);
  cat.y = 96;
  cat.vx = 0;
  cat.vy = 0;
  app.stage.addChild(cat);

  //Start the game loop
  app.ticker.add(delta => gameLoop(delta));
}

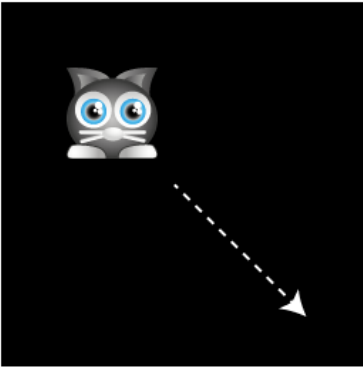
function gameLoop(delta){

  //Update the cat's velocity
  cat.vx = 1;
  cat.vy = 1;

  //Apply the velocity values to the cat's
  //position to make it move
  cat.x += cat.vx;
  cat.y += cat.vy;
}

```

When you run this code, the cat will move down and to the right at one pixel per frame:



What if you want to make the cat move in a different direction? To make the cat move to the left, give it a `vx` value of `-1`. To make it move up, give the cat a `vy` value of `-1`. To make the cat move faster, give it larger `vx` and `vy` values, like `3`, `5`, `-2`, or `-4`.

You'll see ahead how modularizing a sprite's velocity with `vx` and `vy` velocity properties helps with keyboard and mouse pointer control systems for games, as well as making it easier to implement physics.

## Game states

As a matter of style, and to help modularize your code, I recommend structuring your game loop like this:

```
//Set the game state
state = play;

//Start the game loop
app.ticker.add(delta => gameLoop(delta));

function gameLoop(delta){

    //Update the current game state:
    state(delta);
}

function play(delta) {

    //Move the cat 1 pixel to the right each frame
    cat.vx = 1
    cat.x += cat.vx;
}
```

You can see that the `gameLoop` is calling a function called `state` 60 times per second. What is the `state` function? It's been assigned to `play`. That means all the code in the `play` function will also run at 60 times per second.

Here's how the code from the previous example can be re-factored to this new model:

```
//Define any variables that are used in more than one function
let cat, state;

function setup() {

    //Create the `cat` sprite
    cat = new Sprite(resources["images/cat.png"].texture);
    cat.y = 96;
    cat.vx = 0;
    cat.vy = 0;
    app.stage.addChild(cat);

    //Set the game state
    state = play;

    //Start the game loop
    app.ticker.add(delta => gameLoop(delta));
}

function gameLoop(delta){

    //Update the current game state:
```

```

    state(delta);
  }

  function play(delta) {

    //Move the cat 1 pixel to the right each frame
    cat.vx = 1
    cat.x += cat.vx;
  }

```

Yes, I know, this is a bit of [head-swirler!](#) But, don't let it scare you and spend a minute or two walking through in your mind how those functions are connected. As you'll see ahead, structuring your game loop like this will make it much, much easier to do things like switching game scenes and levels.

## Keyboard Movement

With just a little more work you can build a **simple system to control a sprite using the keyboard**. To simplify your code, I suggest you use this custom function called `keyboard` that listens for and captures keyboard events.

```

function keyboard(value) {
  let key = {};
  key.value = value;
  key.isDown = false;
  key.isUp = true;
  key.press = undefined;
  key.release = undefined;
  //The `downHandler`
  key.downHandler = event => {
    if (event.key === key.value) {
      if (key.isUp && key.press) key.press();
      key.isDown = true;
      key.isUp = false;
      event.preventDefault();
    }
  };

  //The `upHandler`
  key.upHandler = event => {
    if (event.key === key.value) {
      if (key.isDown && key.release) key.release();
      key.isDown = false;
      key.isUp = true;
      event.preventDefault();
    }
  };

  //Attach event listeners
  const downListener = key.downHandler.bind(key);
  const upListener = key.upHandler.bind(key);

  window.addEventListener(
    "keydown", downListener, false
  );
  window.addEventListener(
    "keyup", upListener, false
  );

  // Detach event listeners
  key.unsubscribe = () => {
    window.removeEventListener("keydown", downListener);
    window.removeEventListener("keyup", upListener);
  };

  return key;
}

```

The `keyboard` function is easy to use. Create a new keyboard object like this:

```
let keyObject = keyboard(keyValue);
```

Its one argument is the key value that you want to listen for. [Here's a list of keys](#).

Then assign `press` and `release` methods to the keyboard object like this:

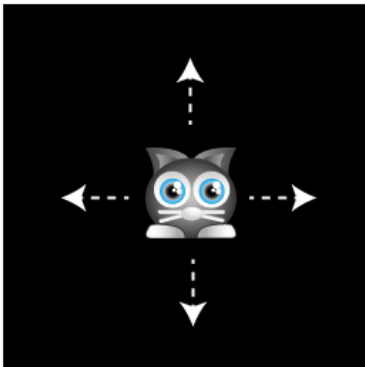
```
keyObject.press = () => {
  //key object pressed
};
keyObject.release = () => {
  //key object released
};
```

Keyboard objects also have `isDown` and `isUp` Boolean properties that you can use to check the state of each key.

Don't forget to remove event listeners by using the `unsubscribe` method:

```
keyObject.unsubscribe();
```

Take a look at the `keyboardMovement.html` file in the `examples` folder to see how you can use this `keyboard` function to control a sprite using your keyboard's arrow keys. Run it and use the left, up, down, and right arrow keys to move the cat around the stage.



Here's the code that does all this:

```
//Define any variables that are used in more than one function
let cat, state;

function setup() {

  //Create the `cat` sprite
  cat = new Sprite(resources["images/cat.png"].texture);
  cat.y = 96;
  cat.vx = 0;
  cat.vy = 0;
  app.stage.addChild(cat);

  //Capture the keyboard arrow keys
  let left = keyboard("ArrowLeft"),
      up = keyboard("ArrowUp"),
      right = keyboard("ArrowRight"),
      down = keyboard("ArrowDown");

  //Left arrow key `press` method
  left.press = () => {
    //Change the cat's velocity when the key is pressed
    cat.vx = -5;
    cat.vy = 0;
  };

  //Left arrow key `release` method
  left.release = () => {
    //If the left arrow has been released, and the right arrow isn't down,
    //and the cat isn't moving vertically:
    //Stop the cat
    if (!right.isDown && cat.vy === 0) {
      cat.vx = 0;
    }
  };

  //Up
  up.press = () => {
```

```

    cat.vy = -5;
    cat.vx = 0;
  };
  up.release = () => {
    if (!down.isDown && cat.vx === 0) {
      cat.vy = 0;
    }
  };

  //Right
  right.press = () => {
    cat.vx = 5;
    cat.vy = 0;
  };
  right.release = () => {
    if (!left.isDown && cat.vy === 0) {
      cat.vx = 0;
    }
  };

  //Down
  down.press = () => {
    cat.vy = 5;
    cat.vx = 0;
  };
  down.release = () => {
    if (!up.isDown && cat.vx === 0) {
      cat.vy = 0;
    }
  };

  //Set the game state
  state = play;

  //Start the game loop
  app.ticker.add(delta => gameLoop(delta));
}

function gameLoop(delta){

  //Update the current game state:
  state(delta);
}

function play(delta) {

  //Use the cat's velocity to make it move
  cat.x += cat.vx;
  cat.y += cat.vy
}

```

## Grouping Sprites

Groups let you create game scenes, and manage similar **sprites together as single units**. Pixi has an object called a **Container** that lets you do this. Let's find out how it works.

Imagine that you want to display three sprites: a cat, hedgehog and tiger. Create them, and set their positions - *but don't add them to the stage*.

```

//The cat
let cat = new Sprite(id["cat.png"]);
cat.position.set(16, 16);

//The hedgehog
let hedgehog = new Sprite(id["hedgehog.png"]);
hedgehog.position.set(32, 32);

//The tiger
let tiger = new Sprite(id["tiger.png"]);
tiger.position.set(64, 64);

```

Next, create an `animals` container to group them all together like this:

```
let animals = new PIXI.Container();
```

Then use `addChild` to *add the sprites to the group*.

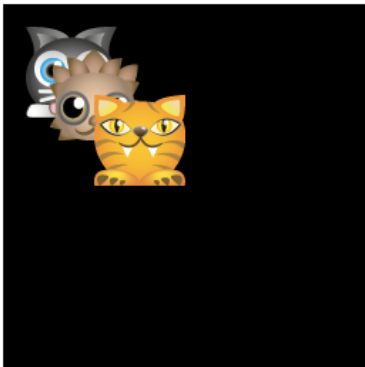
```
animals.addChild(cat);  
animals.addChild(hedgehog);  
animals.addChild(tiger);
```

Finally add the group to the stage.

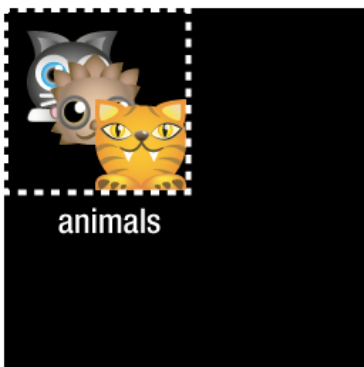
```
app.stage.addChild(animals);
```

(As you know, the `stage` object is also a `Container`. It's the root container for all Pixi sprites.)

Here's what this code produces:



What you can't see in that image is the invisible `animals` group that's containing the sprites.



You can now treat the `animals` group as a single unit. You can think of a `Container` as a special kind of sprite that doesn't have a texture.

If you need a list of all the child sprites that `animals` contains, use its `children` array to find out.

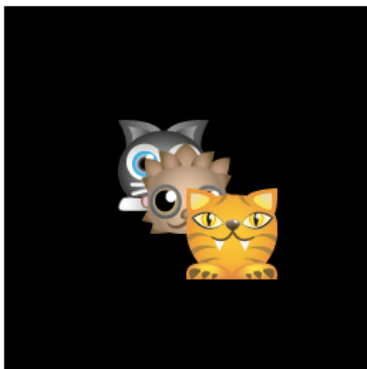
```
console.log(animals.children)  
//Displays: Array [Object, Object, Object]
```

This tells you that `animals` has three sprites as children.

Because the `animals` group is just like any other sprite, you can change its `x` and `y` values, `alpha`, `scale` and all the other sprite properties. Any property value you change on the parent container will affect the child sprites in a relative way. So if you set the group's `x` and `y` position, all the child sprites will be repositioned relative to the group's top left corner. What would happen if you set the `animals`'s `x` and `y` position to 64?

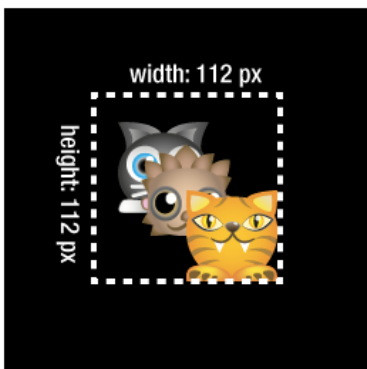
```
animals.position.set(64, 64);
```

The whole group of sprites will move 64 pixels right and 64 pixels to the down.



The `animals` group also has its own dimensions, which is based on the area occupied by the containing sprites. You can find its `width` and `height` values like this:

```
console.log(animals.width);  
//Displays: 112  
  
console.log(animals.height);  
//Displays: 112
```



What happens if you change a group's width or height?

```
animals.width = 200;  
animals.height = 200;
```

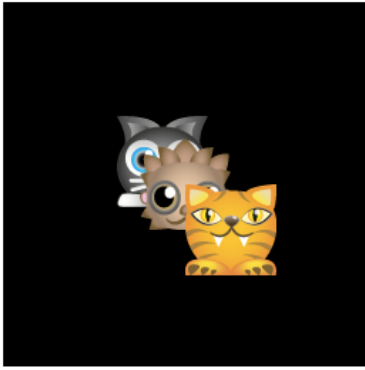
All the child sprites will scale to match that change.



You can nest as many `Container`s inside other `Container`s as you like, to create deep hierarchies if you need to. However, a `DisplayObject` (like a `Sprite` or another `Container`) can only belong to one parent at a time. If you use `addChild` to make a `sprite` the child of another object, Pixi will automatically remove it from its current parent. That's a useful bit of management that you don't have to worry about.

## Local and global positions

When you add a `sprite` to a `Container`, its `x` and `y` position is *relative to the group's top left corner*. That's the `sprite`'s **local position**. For example, what do you think the cat's position is in this image?



Let's find out:

```
console.log(cat.x);
//Displays: 16
```

16? Yes! That's because the **cat is offset by only 16 pixel's from the group's top left corner**. 16 is the cat's local position.

Sprites also have a **global position**. The global position is the distance from the top left corner of the stage, to the sprite's anchor point (usually the sprite's top left corner.) You can find a sprite's global position with the help of the `toGlobal` method. Here's how:

```
parentSprite.toGlobal(childSprite.position)
```

That means you can find the cat's global position inside the `animals` group like this:

```
console.log(animals.toGlobal(cat.position));
//Displays: Object {x: 80, y: 80...};
```

That gives you an `x` and `y` position of 80. That's exactly the cat's global position relative to the top left corner of the stage.

What if you want to find the global position of a sprite, but don't know what the sprite's parent container is? Every sprite has a property called `parent` that will tell you what the sprite's parent is. If you add a sprite directly to the `stage`, then `stage` will be the sprite's parent. In the example above, the `cat`'s parent is `animals`. That means you can alternatively get the cat's global position by writing code like this:

```
cat.parent.toGlobal(cat.position);
```

And it will work even if you don't know what the cat's parent container currently is.

There's one more way to calculate the global position! And, it's actually the best way, so listen up! If you want to know the distance from the top left corner of the canvas to the sprite, and don't know or care what the sprite's parent containers are, use the `getGlobalPosition` method. Here's how to use it to find the tiger's global position:

```
tiger.getGlobalPosition().x
tiger.getGlobalPosition().y
```

This will give you `x` and `y` values of 128 in the example that we've been using. The special thing about `getGlobalPosition` is that it's highly precise: it will give you the sprite's accurate global position as soon as its local position changes. I asked the Pixi development team to add this feature specifically for accurate collision detection for games. (Thanks, Matt and the rest of the team for adding it!)

What if you want to convert a global position to a local position? you can use the `toLocal` method. It works in a similar way, but uses this general format:

```
sprite.toLocal(sprite.position, anyOtherSprite)
```

Use `toLocal` to find the distance between a sprite and any other sprite. Here's how you could find out the tiger's local position, relative to the hedgehog.



```
tiger.toLocal(tiger.position, hedgehog).x
tiger.toLocal(tiger.position, hedgehog).y
```

This gives you an `x` value of 32 and a `y` value of 32. You can see in the example images that the tiger's top left corner is 32 pixels down and to the left of the hedgehog's top left corner.

## Using a ParticleContainer to group sprites

Pixi has an alternative, **high-performance way to group sprites called a `ParticleContainer`** (`PIXI.particles.ParticleContainer`). Any sprites inside a `ParticleContainer` will render **2 to 5 times faster** than they would if they were in a regular `Container`. It's a great performance boost for games.

Create a `ParticleContainer` like this:

```
let superFastSprites = new PIXI.particles.ParticleContainer();
```

Then use `addChild` to add sprites to it, just like you would with any ordinary `Container`.

You have to make some compromises if you decide to use a `ParticleContainer`. Sprites inside a `ParticleContainer` **only have a few basic properties: `x`, `y`, `width`, `height`, `scale`, `pivot`, `alpha`, `visible`** – and that's about it. Also, the sprites that it contains can't have nested children of their own. A `ParticleContainer` also can't use Pixi's advanced visual effects like filters and blend modes. **Each `ParticleContainer` can use only one texture** (so you'll have to use a spritesheet if you want Sprites with different appearances). But for the huge performance boost that you get, those compromises are usually worth it. And you can use `Container`s and `ParticleContainer`s simultaneously in the same project, so you can fine-tune your optimization.

Why are sprites in a `ParticleContainer` so fast? Because the positions of the **sprites are being calculated directly on the GPU**. The Pixi development team is working to offload as much sprite processing as possible on the GPU, so it's likely that the latest version of Pixi that you're using will have much more feature-rich `ParticleContainer` than what I've described here. Check the current [ParticleContainer documentation](#) for details.

Where you create a `ParticleContainer`, there are four optional arguments you can provide: `size`, `properties`, `batchSize` and `autoResize`.

```
let superFastSprites = new ParticleContainer(maxSize, properties, batchSize, autoResize);
```

The default value for `maxSize` is 1500. So, if you need to contain more sprites, set it to a higher number. The `properties` argument is an object with 5 Boolean values you can set: `scale`, `position`, `rotation`, `uvs` and `alphaAndTint`. The default value of `position` is `true`, but all the others are set to `false`. That means that if you want change the `rotation`, `scale`, `tint`, or `uvs` of sprite in the `ParticleContainer`, you have to set those properties to `true`, like this:

```
let superFastSprites = new ParticleContainer(
  size,
  {
    rotation: true,
    alphaAndTint: true,
    scale: true,
    uvs: true
  }
);
```

But, if you don't think you'll need to use these properties, keep them set to `false` to squeeze out the maximum amount of performance.

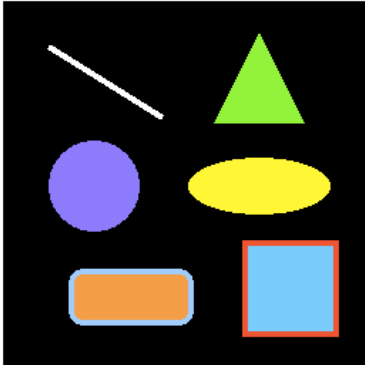
What's the `uvs` property? Only set it to `true` if you have particles which **change their textures while they're being animated**. (All the sprite's textures will also need to be on the same tileset image for this to work.)

(Note: **UV mapping** is a 3D graphics display term that refers to the `x` and `y` coordinates of the texture (the image) that is being mapped onto a 3D surface. `u` is the `x` axis and `v` is the `y` axis. WebGL already uses `x`, `y` and `z` for 3D spatial positioning, so `u` and `v` were chosen to represent `x` and `y` for 2D image textures.)

(I'm not sure what exactly what those last two optional arguments, `batchSize` and `autoResize`, so if anyone knows, please us know in the Issues!)

## Pixi's Graphic Primitives

Using image textures is one of the most useful ways of making sprites, but Pixi also has its own low-level drawing tools. You can use them **to make rectangles, shapes, lines, complex polygons and text**. And, fortunately, it uses almost the same API as the **Canvas Drawing API** so, if you're already familiar with canvas, there's nothing really new to learn. But the big advantage is that, unlike the Canvas Drawing API, the shapes you draw with Pixi are rendered by WebGL on the GPU. Pixi lets you access all that untapped performance power. Let's take a quick tour of how to make some basic shapes. Here are all the shapes we'll make in the code ahead.



### Rectangles

All shapes are made by first creating a new instance of Pixi's `Graphics` class (`PIXI.Graphics`).

```
let rectangle = new Graphics();
```

Use `beginFill` with a hexadecimal color code value to set the rectangle's fill color. Here's how to set it to light blue.

```
rectangle.beginFill(0x66CCFF);
```

If you want to give the shape an outline, use the `lineStyle` method. Here's how to give the rectangle a 4 pixel wide red outline, with an `alpha` value of 1.

```
rectangle.lineStyle(4, 0xFF3300, 1);
```

Use the `drawRect` method to draw the rectangle. Its four arguments are `x`, `y`, `width` and `height`.

```
rectangle.drawRect(x, y, width, height);
```

Use `endFill` when you're done.

```
rectangle.endFill();
```

It's just like the Canvas Drawing API! Here's all the code you need to draw a rectangle, change its position, and add it to the stage.

```
let rectangle = new Graphics();
rectangle.lineStyle(4, 0xFF3300, 1);
rectangle.beginFill(0x66CCFF);
rectangle.drawRect(0, 0, 64, 64);
rectangle.endFill();
rectangle.x = 170;
rectangle.y = 170;
app.stage.addChild(rectangle);
```

This code makes a 64 by 64 blue rectangle with a red border at an `x` and `y` position of 170.

### Circles

Make a circle with the `drawCircle` method. Its three arguments are `x`, `y` and `radius`

```
drawCircle(x, y, radius)
```

Unlike rectangles and sprites, a circle's `x` and `y` position is also its center point. Here's how to make a violet colored circle with a radius of 32 pixels.

```
let circle = new Graphics();
circle.beginFill(0x9966FF);
circle.drawCircle(0, 0, 32);
circle.endFill();
circle.x = 64;
circle.y = 130;
app.stage.addChild(circle);
```

## Ellipses

As a one-up on the Canvas Drawing API, Pixi lets you draw an ellipse with the `drawEllipse` method.

```
drawEllipse(x, y, width, height);
```

The `x/y` position defines the ellipse's top left corner (imagine that the ellipse is surrounded by an invisible rectangular bounding box - the top left corner of that box will represent the ellipse's `x/y` anchor position). Here's a yellow ellipse that's 50 pixels wide and 20 pixels high.

```
let ellipse = new Graphics();
ellipse.beginFill(0xFFFF00);
ellipse.drawEllipse(0, 0, 50, 20);
ellipse.endFill();
ellipse.x = 180;
ellipse.y = 130;
app.stage.addChild(ellipse);
```

## Rounded rectangles

Pixi also lets you make rounded rectangles with the `drawRoundedRect` method. The last argument, `cornerRadius` is a number in pixels that determines by how much the corners should be rounded.

```
drawRoundedRect(x, y, width, height, cornerRadius)
```

Here's how to make a rounded rectangle with a corner radius of 10 pixels.

```
let roundBox = new Graphics();
roundBox.lineStyle(4, 0x99CCFF, 1);
roundBox.beginFill(0xFF9933);
roundBox.drawRoundedRect(0, 0, 84, 36, 10);
roundBox.endFill();
roundBox.x = 48;
roundBox.y = 190;
app.stage.addChild(roundBox);
```

## Lines

You've seen in the examples above that the `lineStyle` method lets you define a line. You can use the `moveTo` and `lineTo` methods to draw the start and end points of the line, in just the same way you can with the Canvas Drawing API. Here's how to draw a 4 pixel wide, white diagonal line.

```
let line = new Graphics();
line.lineStyle(4, 0xFFFFFF, 1);
line.moveTo(0, 0);
line.lineTo(80, 50);
line.x = 32;
```

```
line.y = 32;
app.stage.addChild(line);
```

`PIXI.Graphics` objects, like lines, have `x` and `y` values, just like sprites, so you can position them anywhere on the stage after you've drawn them.

## Polygons

You can join lines together and fill them with colors to make complex shapes using the `drawPolygon` method. `drawPolygon`'s argument is a path array of x/y points that define the positions of each point on the shape.

```
let path = [
  point1X, point1Y,
  point2X, point2Y,
  point3X, point3Y
];

graphicsObject.drawPolygon(path);
```

`drawPolygon` will join those three points together to make the shape. Here's how to use `drawPolygon` to connect three lines together to make a red triangle with a blue border. The triangle is drawn at position 0,0 and then moved to its position on the stage using its `x` and `y` properties.

```
let triangle = new Graphics();
triangle.beginFill(0x66FF33);

//Use `drawPolygon` to define the triangle as
//a path array of x/y positions

triangle.drawPolygon([
  -32, 64,           //First point
  32, 64,            //Second point
  0, 0               //Third point
]);

//Fill shape's color
triangle.endFill();

//Position the triangle after you've drawn it.
//The triangle's x/y position is anchored to its first point in the path
triangle.x = 180;
triangle.y = 22;

app.stage.addChild(triangle);
```

## Displaying text

Use a `Text` object ( `PIXI.Text` ) to display text on the stage. In its simplest form, you can do it like this:

```
let message = new Text("Hello Pixi!");
app.stage.addChild(message);
```

This will display the words, "Hello, Pixi" on the canvas. Pixi's `Text` objects inherit from the `Sprite` class, so they contain all the same properties like `x`, `y`, `width`, `height`, `alpha`, and `rotation`. Position and resize text on the stage just like you would any other sprite. For example, you could use `position.set` to set the `message`'s `x` and `y` position like this:

```
message.position.set(54, 96);
```



Hello Pixi!

That will give you basic, unstyled text. But if you want to get fancier, use Pixi's `TextStyle` function to define custom text styling. Here's how:

```
let style = new TextStyle({
  fontFamily: "Arial",
  fontSize: 36,
  fill: "white",
  stroke: '#ff3300',
  strokeThickness: 4,
  dropShadow: true,
  dropShadowColor: "#000000",
  dropShadowBlur: 4,
  dropShadowAngle: Math.PI / 6,
  dropShadowDistance: 6,
});
```

That creates a new `style` object containing all the text styling that you'd like to use. For a complete list of all the style properties you can use, [see here](#).

To apply the style to the text, add the `style` object as the `Text` function's second argument, like this:

```
let message = new Text("Hello Pixi!", style);
```



Hello Pixi!

If you want to change the content of a text object after you've created it, use the `text` property.

```
message.text = "Text changed!";
```

Use the `style` property if you want to redefine the style properties.

```
message.style = {fill: "black", font: "16px PetMe64"};
```

Pixi makes text objects by using the Canvas Drawing API to render the text to an invisible and temporary canvas element. It then turns the canvas into a WebGL texture so that it can be mapped onto a sprite. That's why the text's color needs to be wrapped in a string: it's a Canvas Drawing API color value. As with any canvas color values, you can use words for common colors like "red" or "green", or use rgba, hsla or hex values.

Pixi can also wrap long lines of text. Set the text's `wordWrap` style property to `true`, and then set `wordWrapWidth` to the maximum length in pixels, that the line of text should be. Use the `align` property to set the alignment for multi-line text.

```
message.style = {wordWrap: true, wordWrapWidth: 100, align: center};
```

(Note: `align` doesn't affect single line text.)

If you want to use a custom font file, use the CSS `@font-face` rule to link the font file to the HTML page where your Pixi application is running.

```
@font-face {
  font-family: "fontFamilyName";
  src: url("fonts/FontFile.ttf");
}
```

Add this `@font-face` rule to your HTML page's CSS style sheet.

Pixi also has support for **bitmap fonts**. You can use Pixi's loader to load Bitmap font XML files, the same way you load JSON or image files.

## Collision detection

You now know how to make a huge variety of graphics objects, but what can you do with them? A fun thing to do is to build a simple **collision detection** system. You can use a custom function called `hitTestRectangle` that **checks whether any two rectangular Pixi sprites are touching**.

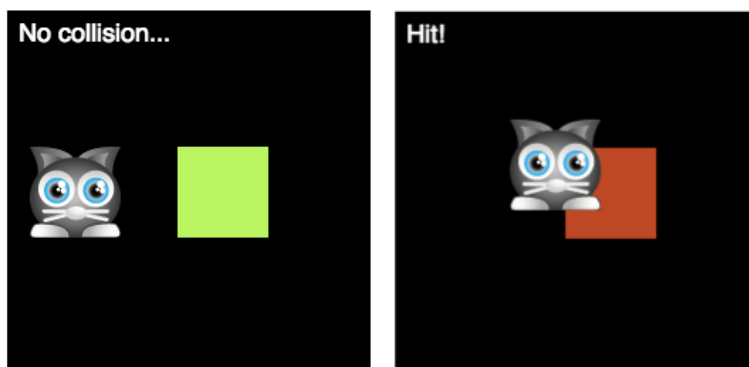
```
hitTestRectangle(spriteOne, spriteTwo)
```

if they overlap, `hitTestRectangle` will return `true`. You can use `hitTestRectangle` with an `if` statement to check for a collision between two sprites like this:

```
if (hitTestRectangle(cat, box)) {
  //There's a collision
} else {
  //There's no collision
}
```

As you'll see, `hitTestRectangle` is the front door into the vast universe of game design.

Run the `collisionDetection.html` file in the `examples` folder for a working example of how to use `hitTestRectangle`. Use the arrow keys to move the cat. If the cat hits the box, the box becomes red and "Hit!" is displayed by the text object.



You've already seen all the code that creates all these elements, as well as the keyboard control system that makes the cat move. The only new thing is the way `hitTestRectangle` is used inside the `play` function to check for a collision.

```
function play(delta) {

  //use the cat's velocity to make it move
  cat.x += cat.vx;
  cat.y += cat.vy;

  //check for a collision between the cat and the box
  if (hitTestRectangle(cat, box)) {
```

```

    //if there's a collision, change the message text
    //and tint the box red
    message.text = "hit!";
    box.tint = 0xff3300;

} else {

    //if there's no collision, reset the message
    //text and the box's color
    message.text = "No collision...";
    box.tint = 0xccff99;
}
}

```

Because the `play` function is being called by the game loop 60 times per second, this `if` statement is constantly checking for a collision between the cat and the box. If `hitTestRectangle` is `true`, the text `message` object uses `text` to display "Hit":

```
message.text = "Hit!";
```

The color of the box is then changed from green to red by setting the box's `tint` property to the hexadecimal red value.

```
box.tint = 0xff3300;
```

If there's no collision, the message and box are maintained in their original states:

```
message.text = "No collision...";
box.tint = 0xccff99;
```

This code is pretty simple, but suddenly you've created an interactive world that seems to be completely alive. It's almost like magic! And, perhaps surprisingly, you now have all the skills you need to start making games with Pixi!

## The hitTestRectangle function

But what about the `hitTestRectangle` function? What does it do, and how does it work? The details of how collision detection algorithms like this work is a little bit outside the scope of this tutorial. (If you really want to know, you can find out how [this book](#).) The most important thing is that you know how to use it. But, just for your reference, and in case you're curious, here's the complete `hitTestRectangle` function definition. Can you figure out from the comments what it's doing?

```

function hitTestRectangle(r1, r2) {

    //Define the variables we'll need to calculate
    let hit, combinedHalfWidths, combinedHalfHeights, vx, vy;

    //hit will determine whether there's a collision
    hit = false;

    //Find the center points of each sprite
    r1.centerX = r1.x + r1.width / 2;
    r1.centerY = r1.y + r1.height / 2;
    r2.centerX = r2.x + r2.width / 2;
    r2.centerY = r2.y + r2.height / 2;

    //Find the half-widths and half-heights of each sprite
    r1.halfWidth = r1.width / 2;
    r1.halfHeight = r1.height / 2;
    r2.halfWidth = r2.width / 2;
    r2.halfHeight = r2.height / 2;

    //Calculate the distance vector between the sprites
    vx = r1.centerX - r2.centerX;
    vy = r1.centerY - r2.centerY;

    //Figure out the combined half-widths and half-heights
    combinedHalfWidths = r1.halfWidth + r2.halfWidth;
    combinedHalfHeights = r1.halfHeight + r2.halfHeight;

    //Check for a collision on the x axis
    if (Math.abs(vx) < combinedHalfWidths) {

```

```
//A collision might be occurring. Check for a collision on the y axis
if (Math.abs(vy) < combinedHalfHeights) {

    //There's definitely a collision happening
    hit = true;
} else {

    //There's no collision on the y axis
    hit = false;
}

//There's no collision on the x axis
hit = false;
}

//`hit` will be either `true` or `false`
return hit;
};
```

## Case study: Treasure Hunter

I've told you that you now have all the skills you need to start making games. What? You don't believe me? Let me prove it to you! Let's take a close at how to make a simple object collection and enemy avoidance game called **Treasure Hunter**. (You'll find it in the `examples` folder.)



Treasure Hunter is a good example of one of the simplest complete games you can make using the tools you've learnt so far. Use the keyboard arrow keys to help the explorer find the treasure and carry it to the exit. Six blob monsters move up and down between the dungeon walls, and if they hit the explorer he becomes semi-transparent and the health meter at the top right corner shrinks. If all the health is used up, "You Lost!" is displayed on the stage; if the explorer reaches the exit with the treasure, "You Won!" is displayed. Although it's a basic prototype, Treasure Hunter contains most of the elements you'll find in much bigger games: texture atlas graphics, interactivity, collision, and multiple game scenes. Let's go on a tour of how the game was put together so that you can use it as a starting point for one of your own games.

## The code structure

Open the `treasureHunter.html` file and you'll see that all the game code is in one big file. Here's a birds-eye view of how all the code is organized.

```
//Setup Pixi and load the texture atlas files - call the `setup`
//function when they've loaded

function setup() {
    //Initialize the game sprites, set the game `state` to `play`
    //and start the `gameLoop`
}

function gameLoop(delta) {
    //Runs the current game `state` in a loop and renders the sprites
}

function play(delta) {
    //All the game logic goes here
```



```

}

function end() {
  //All the code that should run at the end of the game
}

//The game's helper functions:
//`keyboard`, `hitTestRectangle`, `contain` and `randomInt`

```

Use this as your world map to the game as we look at how each section works.

## Initialize the game in the setup function

As soon as the texture atlas images have loaded, the `setup` function runs. It only runs once, and lets you perform one-time setup tasks for your game. It's a great place to create and initialize objects, sprites, game scenes, populate data arrays or parse loaded JSON game data.

Here's an abridged view of the `setup` function in Treasure Hunter, and the tasks that it performs.

```

function setup() {
  //Create the `gameScene` group
  //Create the `door` sprite
  //Create the `player` sprite
  //Create the `treasure` sprite
  //Make the enemies
  //Create the health bar
  //Add some text for the game over message
  //Create a `gameOverScene` group
  //Assign the player's keyboard controllers

  //set the game state to `play`
  state = play;

  //Start the game loop
  app.ticker.add(delta => gameLoop(delta));
}

```

The last two lines of code, `state = play;` and `gameLoop()` are perhaps the most important. Adding the `gameLoop` to Pixi's ticker switches on the game's engine, and causes the `play` function to be called in a continuous loop. But before we look at how that works, let's see what the specific code inside the `setup` function does.

## Creating the game scenes

The `setup` function creates two `Container` groups called `gameScene` and `gameOverScene`. Each of these are added to the stage.

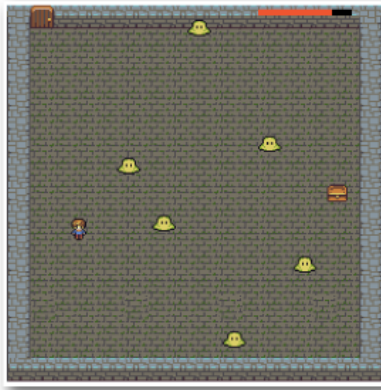
```

gameScene = new Container();
app.stage.addChild(gameScene);

gameOverScene = new Container();
app.stage.addChild(gameOverScene);

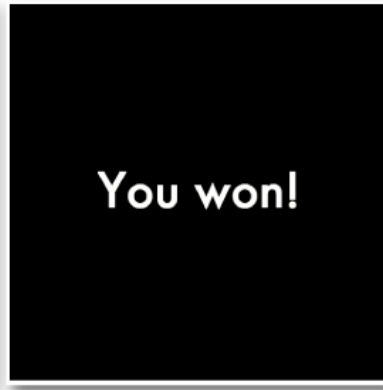
```

All of the sprites that are part of the main game are added to the `gameScene` group. The game over text that should be displayed at the end of the game is added to the `gameOverScene` group.



### gameScene

A container for all the game sprites



### gameOverScene

A container for the message text

Although it's created in the `setup` function, the `gameOverScene` shouldn't be visible when the game first starts, so its `visible` property is initialized to `false`.

```
gameOverScene.visible = false;
```

You'll see ahead that, when the game ends, the `gameOverScene`'s `visible` property will be set to `true` to display the text that appears at the end of the game.

### Making the dungeon, door, explorer and treasure

The player, exit door, treasure chest and the dungeon background image are all sprites made from texture atlas frames. Very importantly, they're all added as children of the `gameScene`.

```
//Create an alias for the texture atlas frame ids
id = resources["images/treasureHunter.json"].textures;

//Dungeon
dungeon = new Sprite(id["dungeon.png"]);
gameScene.addChild(dungeon);

//Door
door = new Sprite(id["door.png"]);
door.position.set(32, 0);
gameScene.addChild(door);

//Explorer
explorer = new Sprite(id["explorer.png"]);
explorer.x = 68;
explorer.y = gameScene.height / 2 - explorer.height / 2;
explorer.vx = 0;
explorer.vy = 0;
gameScene.addChild(explorer);

//Treasure
treasure = new Sprite(id["treasure.png"]);
treasure.x = gameScene.width - treasure.width - 48;
treasure.y = gameScene.height / 2 - treasure.height / 2;
gameScene.addChild(treasure);
```

Keeping them together in the `gameScene` group will make it easy for us to hide the `gameScene` and display the `gameOverScene` when the game is finished.

### Making the blob monsters

The six blob monsters are created in a loop. Each blob is given a random initial position and velocity. The vertical velocity is alternately multiplied by `1` or `-1` for each blob, and that's what causes each blob to move in the opposite direction to the one next to it. Each blob monster that's created is pushed into an array called `blobs`.

```

let numberOfBlobs = 6,
    spacing = 48,
    xOffset = 150,
    speed = 2,
    direction = 1;

//An array to store all the blob monsters
blobs = [];

//Make as many blobs as there are `numberOfBlobs`
for (let i = 0; i < numberOfBlobs; i++) {

    //Make a blob
    let blob = new Sprite(id["blob.png"]);

    //Space each blob horizontally according to the `spacing` value.
    //`xOffset` determines the point from the left of the screen
    //at which the first blob should be added
    let x = spacing * i + xOffset;

    //Give the blob a random `y` position
    let y = randomInt(0, stage.height - blob.height);

    //Set the blob's position
    blob.x = x;
    blob.y = y;

    //Set the blob's vertical velocity. `direction` will be either `1` or
    //`-1`. `1` means the enemy will move down and `-1` means the blob will
    //move up. Multiplying `direction` by `speed` determines the blob's
    //vertical direction
    blob.vy = speed * direction;

    //Reverse the direction for the next blob
    direction *= -1;

    //Push the blob into the `blobs` array
    blobs.push(blob);

    //Add the blob to the `gameScene`
    gameScene.addChild(blob);
}

```

## Making the health bar

When you play Treasure Hunter you'll notice that when the explorer touches one of the enemies, the width of the health bar at the top right corner of the screen decreases. How was this health bar made? It's just two overlapping rectangles at exactly the same position: a black rectangle behind, and a red rectangle in front. They're grouped together into a single `healthBar` group. The `healthBar` is then added to the `gameScene` and positioned on the stage.

```

//Create the health bar
healthBar = new PIXI.Container();
healthBar.position.set(stage.width - 170, 4)
gameScene.addChild(healthBar);

//Create the black background rectangle
let innerBar = new PIXI.Graphics();
innerBar.beginFill(0x000000);
innerBar.drawRect(0, 0, 128, 8);
innerBar.endFill();
healthBar.addChild(innerBar);

//Create the front red rectangle
let outerBar = new PIXI.Graphics();
outerBar.beginFill(0xFF3300);
outerBar.drawRect(0, 0, 128, 8);
outerBar.endFill();
healthBar.addChild(outerBar);

healthBar.outer = outerBar;

```

You can see that a property called `outer` has been added to the `healthBar`. It just references the `outerBar` (the red rectangle) so that it will be convenient to access later.

```
healthBar.outer = outerBar;
```

You don't have to do this; but, hey why not! It means that if you want to control the width of the red `outerBar`, you can write some smooth code that looks like this:

```
healthBar.outer.width = 30;
```

That's pretty neat and readable, so we'll keep it!

### Making the message text

When the game is finished, some text displays "You won!" or "You lost!", depending on the outcome of the game. This is made using a text sprite and adding it to the `gameOverScene`. Because the `gameOverScene`'s `visible` property is set to `false` when the game starts, you can't see this text. Here's the code from the `setup` function that creates the message text and adds it to the `gameOverScene`.

```
let style = new TextStyle({
  fontFamily: "Futura",
  fontSize: 64,
  fill: "white"
});
message = new Text("The End!", style);
message.x = 120;
message.y = app.stage.height / 2 - 32;
gameOverScene.addChild(message);
```

### Playing the game

All the game logic and the code that makes the sprites move happens inside the `play` function, which runs in a continuous loop. Here's an overview of what the `play` function does

```
function play(delta) {
  //Move the explorer and contain it inside the dungeon
  //Move the blob monsters
  //Check for a collision between the blobs and the explorer
  //Check for a collision between the explorer and the treasure
  //Check for a collision between the treasure and the door
  //Decide whether the game has been won or lost
  //Change the game `state` to `end` when the game is finished
}
```

Let's find out how all these features work.

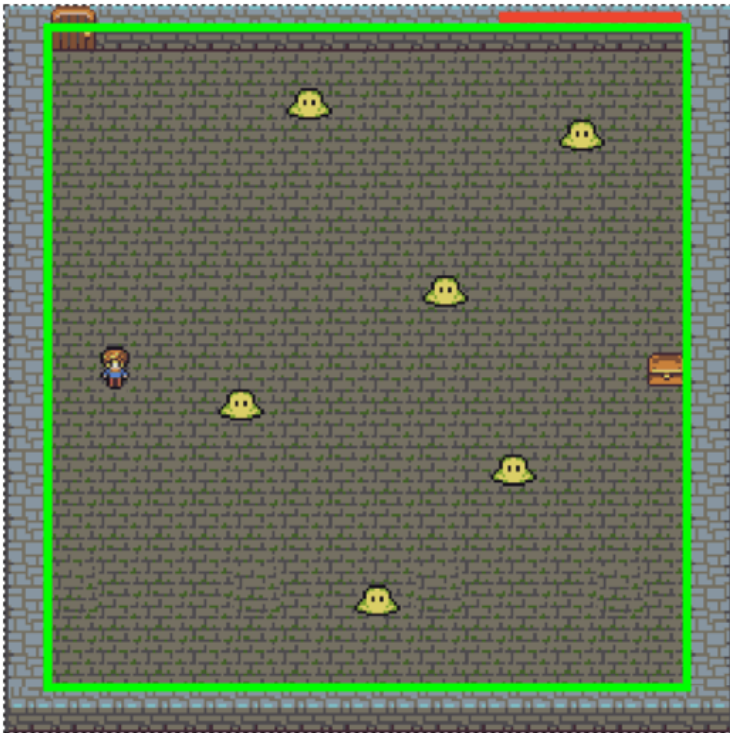
### Moving the explorer

The explorer is controlled using the keyboard, and the code that does that is very similar to the keyboard control code you learnt earlier. The `keyboard` objects modify the explorer's velocity, and that velocity is added to the explorer's position inside the `play` function.

```
explorer.x += explorer.vx;
explorer.y += explorer.vy;
```

### Containing movement

But what's new is that the explorer's movement is contained inside the walls of the dungeon. The green outline shows the limits of the explorer's movement.



That's done with the help of a custom function called `contain`.

```
contain(explorer, {x: 28, y: 10, width: 488, height: 480});
```

`contain` takes two arguments. The first is the sprite you want to keep contained. The second is any object with `x`, `y`, `width` and `height` properties that define a rectangular area. In this example, the containing object defines an area that's just slightly offset from, and smaller than, the stage. It matches the dimensions of the dungeon walls.

Here's the `contain` function that does all this work. The function checks to see if the sprite has crossed the boundaries of the containing object. If it has, the code moves the sprite back into that boundary. The `contain` function also returns a `collision` variable with the value "top", "right", "bottom" or "left", depending on which side of the boundary the sprite hit. (`collision` will be `undefined` if the sprite didn't hit any of the boundaries.)

```
function contain(sprite, container) {
    let collision = undefined;

    //Left
    if (sprite.x < container.x) {
        sprite.x = container.x;
        collision = "left";
    }

    //Top
    if (sprite.y < container.y) {
        sprite.y = container.y;
        collision = "top";
    }

    //Right
    if (sprite.x + sprite.width > container.width) {
        sprite.x = container.width - sprite.width;
        collision = "right";
    }

    //Bottom
    if (sprite.y + sprite.height > container.height) {
        sprite.y = container.height - sprite.height;
        collision = "bottom";
    }

    //Return the `collision` value
    return collision;
}
```

You'll see how the `collision` return value will be used in the code ahead to make the blob monsters bounce back and forth between the top and bottom dungeon walls.

## Moving the monsters

The `play` function also moves the blob monsters, keeps them contained inside the dungeon walls, and checks each one for a collision with the player. If a blob bumps into the dungeon's top or bottom walls, its direction is reversed. All this is done with the help of a `forEach` loop which iterates through each of `blob` sprites in the `blobs` array on every frame.

```
blobs.forEach(function(blob) {

    //Move the blob
    blob.y += blob.vy;

    //Check the blob's screen boundaries
    let blobHitsWall = contain(blob, {x: 28, y: 10, width: 488, height: 480});

    //If the blob hits the top or bottom of the stage, reverse
    //its direction
    if (blobHitsWall === "top" || blobHitsWall === "bottom") {
        blob.vy *= -1;
    }

    //Test for a collision. If any of the enemies are touching
    //the explorer, set `explorerHit` to `true`
    if(hitTestRectangle(explorer, blob)) {
        explorerHit = true;
    }
});
```

You can see in this code above how the return value of the `contain` function is used to make the blobs bounce off the walls. A variable called `blobHitsWall` is used to capture the return value:

```
let blobHitsWall = contain(blob, {x: 28, y: 10, width: 488, height: 480});
```

`blobHitsWall` will usually be `undefined`. But if the blob hits the top wall, `blobHitsWall` will have the value "top". If the blob hits the bottom wall, `blobHitsWall` will have the value "bottom". If either of these cases are `true`, you can reverse the blob's direction by reversing its velocity. Here's the code that does this:

```
if (blobHitsWall === "top" || blobHitsWall === "bottom") {
    blob.vy *= -1;
}
```

Multiplying the blob's `vy` (vertical velocity) value by `-1` will flip the direction of its movement.

## Checking for collisions

The code in the loop above uses `hitTestRectangle` to figure out if any of the enemies have touched the explorer.

```
if(hitTestRectangle(explorer, blob)) {
    explorerHit = true;
}
```

If `hitTestRectangle` returns `true`, it means there's been a collision and a variable called `explorerHit` is set to `true`. If `explorerHit` is `true`, the `play` function makes the explorer semi-transparent and reduces the width of the `health` bar by 1 pixel.

```
if(explorerHit) {

    //Make the explorer semi-transparent
    explorer.alpha = 0.5;

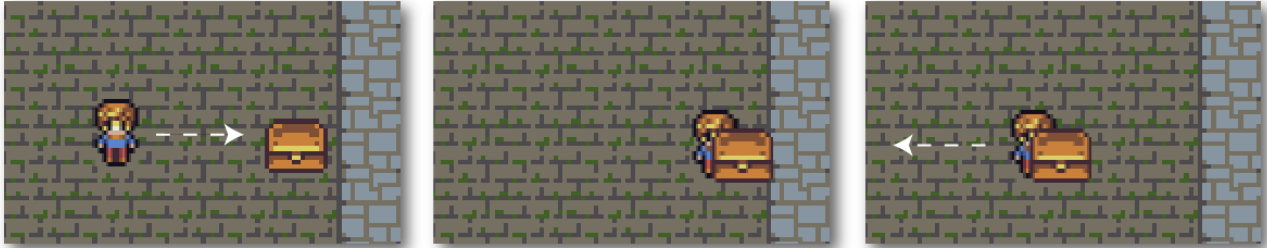
    //Reduce the width of the health bar's inner rectangle by 1 pixel
    healthBar.outer.width -= 1;

} else {
```

```
//Make the explorer fully opaque (non-transparent) if it hasn't been hit
explorer.alpha = 1;
}
```

If `explorerHit` is `false`, the explorer's `alpha` property is maintained at 1, which makes it fully opaque.

The `play` function also checks for a collision between the treasure chest and the explorer. If there's a hit, the `treasure` is set to the explorer's position, with a slight offset. This makes it look like the explorer is carrying the treasure.



Here's the code that does this:

```
if (hitTestRectangle(explorer, treasure)) {
    treasure.x = explorer.x + 8;
    treasure.y = explorer.y + 8;
}
```

## Reaching the exit door and ending the game

There are two ways the game can end: You can win if you carry the treasure to the exit, or you can lose if you run out of health.

To win the game, the treasure chest just needs to touch the exit door. If that happens, the `game state` is set to `end`, and the `message text` displays "You won".

```
if (hitTestRectangle(treasure, door)) {
    state = end;
    message.text = "You won!";
}
```

If you run out of health, you lose the game. The `game state` is also set to `end` and the `message text` displays "You Lost!"

```
if (healthBar.outer.width < 0) {
    state = end;
    message.text = "You lost!";
}
```

But what does this mean?

```
state = end;
```

You'll remember from earlier examples that the `gameLoop` is constantly updating a function called `state` at 60 times per second. Here's the `gameLoop` that does this:

```
function gameLoop(delta){
    //Update the current game state:
    state(delta);
}
```

You'll also remember that we initially set the value of `state` to `play`, which is why the `play` function runs in a loop. By setting `state` to `end` we're telling the code that we want another function, called `end` to run in a loop. In a bigger game you could have a `tileScene` state, and states for each game level, like `leveOne`, `levelTwo` and `levelThree`.

So what is that `end` function? Here it is!

```
function end() {  
  gameScene.visible = false;  
  gameOverScene.visible = true;  
}
```

It just flips the visibility of the game scenes. This is what hides the `gameScene` and displays the `gameOverScene` when the game ends.

This is a really simple example of how to switch a game's state, but you can have as many game states as you like in your games, and fill them with as much code as you need. Just change the value of `state` to whatever function you want to run in a loop.

And that's really all there is to Treasure Hunter! With a little more work you could turn this simple prototype into a full game – try it!

## More about sprites

You've learnt how to use quite a few useful sprite properties so far, like `x`, `y`, `visible`, and `rotation` that give you a lot of control over a sprite's position and appearance. But Pixi Sprites also have many more useful properties that are fun to play with. [Here's the full list.](#)

How does Pixi's class inheritance system work? ([What is a class and what is inheritance? Click this link to find out.](#)) Pixi's sprites are built on an inheritance model that follows this chain:

```
DisplayObject > Container > Sprite
```

Inheritance just means that the classes later in the chain use properties and methods from classes earlier in the chain. That means that even though `Sprite` is the last class in the chain, has all the same properties as `DisplayObject` and `Container`, in addition to its own unique properties. The most basic class is `DisplayObject`. Anything that's a `DisplayObject` can be rendered on the stage. `Container` is the next class in the inheritance chain. It allows `DisplayObject`s to act as containers for other `DisplayObject`s. Third up the chain is the `Sprite` class. Sprites can both be displayed on the stage and be containers for other sprites.

## Taking it further

Pixi can do a lot, but it can't do everything! If you want to start making games or complex interactive applications with Pixi, you'll need to use some helper libraries:

- [Bump](#): A complete suite of 2D collision functions for games.
- [Tink](#): Drag-and-drop, buttons, a universal pointer and other helpful interactivity tools.
- [Charm](#): Easy-to-use tweening animation effects for Pixi sprites.
- [Dust](#): Particle effects for creating things like explosions, fire and magic.
- [Sprite Utilities](#): Easier and more intuitive ways to create and use Pixi sprites, as well adding a state machine and animation player. Makes working with Pixi a lot more fun.
- [Sound.js](#): A micro-library for loading, controlling and generating sound and music effects. Everything you need to add sound to games.
- [Smoothie](#): Ultra-smooth sprite animation using true delta-time interpolation. It also lets you specify the fps (frames-per-second) at which your game or application runs, and completely separates your sprite rendering loop from your application logic loop.

You can find out how to use all these libraries with Pixi in the book [Learn PixiJS](#).

## Hexi

Do you want to use all the functionality of those libraries, but don't want the hassle of integrating them yourself? Use **Hexi**: a complete development environment for building games and interactive applications:

<https://github.com/kittykatattack/hexi>

It bundles the best version of Pixi (the latest **stable** one) with all these libraries (and more!) for a simple and fun way to make games. Hexi also lets you access the global `PIXI` object directly, so you can write low-level Pixi code directly in a Hexi application, and optionally choose to use as many or as few of Hexi's extra conveniences as you need.



## BabylonJS

Pixi is great for 2D, but it can't do 3D. When you're ready to step into the third dimension, the most feature rich, easy-to-use 3D game development platform for the web is [BabylonJS](#). It's a great next step for taking your skills further.

## Please help to support this project!

---

Buy the book! Incredibly, someone actually paid me to finish writing this tutorial and turn it into a book!

[Learn PixiJS](#)

(And it's not just some junky "e-book", but a real, heavy, paper book, published by Springer, the world's largest publisher! That means you can invite your friends over, set it on fire, and roast marshmallows!!) There's 80% more content than what's in this tutorial, and it's packed full of all the essential techniques you need to know to use Pixi to make all kinds of interactive applications and games.

Find out how to:

- Make animated game characters.
- Create a full-featured animation state player.
- Dynamically animate lines and shapes.
- Use tiling sprites for infinite parallax scrolling.
- Use blend modes, filters, tinting, masks, video, and render textures.
- Produce content for multiple resolutions.
- Create interactive buttons.
- Create a flexible drag and drop interface for Pixi.
- Create particle effects.
- Build a stable software architectural model that will scale to any size.
- Make complete games.

And, as a bonus, all the code is written entirely in the latest version of JavaScript: ES6/2015. And, although the book's code is based on Pixi v3.x, it all works just fine with the latest version of Pixi 4.x!

If you want to support this project, please buy a copy of this book, and buy another copy for your mom!

Or, make a generous donation to: <http://www.msf.org>