目　录

致谢

当前文档 《Pixi教程》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-03-01。

阅前必读

Pixi教程

基于官方教程翻译；水平有限，如有错误欢迎提issue，转载请注明出处。翻译者为htkz（完成了用 Pixi 绘制几何图形 和 显示文本 章节）和zainking(完成了其他所有章节)

这个教程将要一步步介绍怎么用Pixi做游戏或者交互式媒体。这个教程已经升级到 **Pixi v4.5.5**。如果你喜欢这个教程，你一定也喜欢这本书，它比这个教程多了80%的内容。

介绍

介绍

Pixi是一个超快的2D渲染引擎。这意味着什么呢？这意味着它会帮助你用JavaScript或者其他HTML5技术来显示媒体，创建动画或管理交互式图像，从而制作一个游戏或应用。它拥有语义化的，简洁的API接口并且加入了一些非常有用的特性。比如支持纹理贴图集和为精灵（交互式图像）提供了一个简单的动画系统。它也提供了一个完备的场景图，你可以在精灵图层里面创建另一个精灵，当然也可以让精灵响应你的鼠标或触摸事件。最重要的的是，Pixi没有妨碍你的编程方式，你可以自己选择使用多少它的功能，你可以遵循你自己的编码风格，或让Pixi与其他有用的框架无缝集成。

Pixi的API事实上比起久经沙场又老旧的Macromedia/Adobe Flash API要精致。如果你是一个Flash开发者，将会对这样的API感觉更好。其他的同类渲染框架（比如CreateJS,Starling, Sparrow 和 Apple’s SpriteKit.）也在使用类似的API。Pixi API的优势在于它是通用的：它不是一个游戏引擎。这是一个优势，因为它给了你所有的自由去做任何你想做的事，甚至用它可以写成你自己的游戏引擎。（译者：作者这点说的很对，译者有一个朋友就使用它制作自己的Galgame引擎AVG.js）。

在这个教程里，你将会明白怎样用Pixi的强大的图片渲染能力和场景图技术来和做一个游戏联系起来。但是Pixi不仅仅能做游戏 —— 你能用这个技术去创建任何交互式媒体应用。这甚至意味着手机应用。

你在开始这个教程之前需要知道什么呢？

你需要一个对于HTML和JavaScript大致的了解。你没必要成为这方

面的专家才能开始，即使一个野心勃勃的初学者也可以开始学习。这本书就是一个学习的好地方：

Foundation Game Design with HTML5 and JavaScript

我知道这本书是最好的，因为这本书是我写的！

这里有一些好的代码来帮助你开始：

Khan Academy: Computer Programming

Code Academy: JavaScript

选择一个属于你的最好的学习方式吧！

所以，明白了么？

你知道JavaScript的变量，函数，数组和对象怎么使用么？你知道JSON 数据文件是什么么？ 你用过 Canvas 绘图 API么？

为了使用Pixi，你也需要在你项目的根目录运行一个web服务器，你知道什么是web服务器，怎么在你的项目文件夹里面运行它么？最好的方式是使用node.js 并且去用命令行安装http-server. 无论如何，你需要习惯和Unix命令行一起工作。你可以在这个视频中去学习怎样使用 Unix当你完成时，继续去学习 这个视频.你应该学会怎样用Unix，这是一个很有趣和简单的和电脑交互的方式，并且仅仅需要两个小时。

如果你真的不想用命令行的方式，就尝试下 Mongoose webserver：

Mongoose

或者来使用Brackets text editor这个令人惊艳的代码编辑器。他会在你点击那个“闪电按钮”的时候自动启动web服务器和浏览器。

现在，如果你觉得你准备好了了，开始吧！

（给读者的小提示：这是一个 交互式的文档.如果你有关于特殊细节的任何问题或需要任何澄清都可以创建一个GitHub工程 **issue** ，我会对这个文档更新更多信息。）

安装

安装

在你开始写任何代码之前，给你的工程创建一个目录，并且在根目录下运行一个web服务器。如果你不这么做，Pixi不会工作的。

现在，你需要去安装Pixi。

安装 Pixi

这个教程使用的版本是 **v4.5.5**
你可以选择使用 Pixi v4.5.5的发布页面 `pixi` 文件夹下的 `pixi.min.js` 文件，或者从Pixi的主要发布页面中获取最新版本。

这个文件就是你使用Pixi唯一需要的文件，你可以忽视所有这个工程的其他文件，你不需要他们。

现在，创建一个基础的HTML页面，用一个 `<script>` 标签去加载你刚刚下载的 `pixi.min.js` 文件。 `<script>` 标签的 `src` 属性应该是你根目录文件的相对路径——当然请确保你的web服务器在运行。你的 `<script>` 标签应该看起来像是这样：

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

这是你用来链接Pixi和测试它是否工作的基础页面。（这里假设 `pixi.min.js` 在一个叫做 `pixi` 的子文件夹中）：

```
1.  <!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>
```

如果Pixi连接成功，一些这样的东西会在你的浏览器控制台里显示：

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

创建舞台（stage）和画布（renderer）

创建Pixi应用和 `舞台`

现在你可以开始使用Pixi！

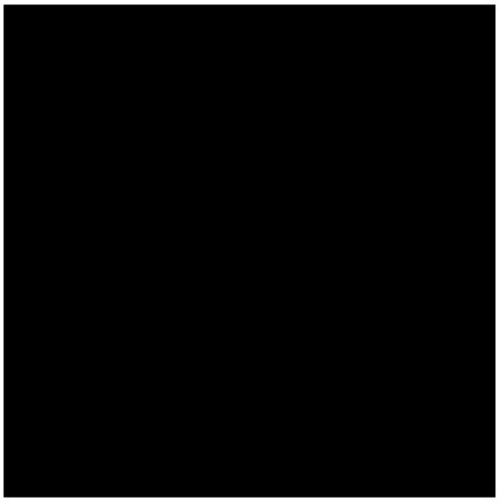
但是怎么用？

第一步就是去创建一个可以显示图片的矩形显示区。Pixi拥有一个 `Pixi应用` 对象来帮助你创建它。它会自动创建一个 `<canvas>` HTML标签并且计算出怎么去让你的图片在这个标签中显示。你现在需要创建一个特殊的Pixi `容器` 对象，他被称作 `舞台` 。正如你所见，这个 `舞台` 对象将会被当作根容器而使用，它将包裹所有你想用Pixi显示的东西。

这里是你需要创建一个名叫 `app` 的Pixi应用对象和一个 `舞台` 的必要的代码。这些代码需要在你的HTML文档中以 `<script>` 标签包裹。

```
1. //Create a Pixi Application
2. let app = new PIXI.Application({width: 256, height: 256});
3. 
4. //Add the canvas that Pixi automatically created for you to the
   HTML document
5. document.body.appendChild(app.view);
```

这是你想要开始使用Pixi的最基本的代码。它在你的文档中创建了一个256像素宽高的黑色canvas标签。当你运行这个代码的时候浏览器应该显示成这样：



啊哈，一个 black square!

`PIXI.Application` 算出了应该使用Canvas还是WebGL去渲染图象，它取决于你正在使用的浏览器支持哪一个。它的参数是一个被称作 `options` 的对象。在这儿例子中，它的 `width` 和 `height` 属性已经被设置了，它们决定了canvas的宽和高（单位是像素）。你能够在 `options` 对象中使用更多的属性设置，这里展示了你如何使用它来圆滑边界，设置透明度和分辨率：

```
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
  }
);
```

如果你觉得Pixi的默认设置也不错，你就不需要作任何的设置，但是如果你需要，就在这里看一下Pixi的文档吧：
PIXI.Application.

这些设置做了些什么呢？

`antialias` 使得字体的边界和几何图形更加圆滑（WebGL的anti-aliasing在所有平台都不可用，所以你需要在你的游戏的标签平台上测试他们）。 `transparent` 将整个Canvs标签的透明度进行了设置。 `resolution` 让Pixi在不同的分辨率和像素密度的平台上运行变得简单。设置分辨率对于这个教程而言有些超纲了，到那时你可以看Mat Grove’sexplanation之中是如何使用 `resolution` 的所有细节的。但是平常，只要保持 `resolution` 是1，就可以应付大多数工程了。

Pixi的 `画布` 对象将会默认选择WebGL引擎渲染模式，它更快并且可以让你使用一些壮观的视觉特效——如果你把他们都学了。但是如果你需要强制使用Canvas引擎绘制而抛弃WebGL，你可以设置 `forceCanvas` 选项为 `true` ，像这样：

```
1.  forceCanvas: true,
```

如果你需要在你创建canvs标签之后改变它的背景色，设置 `app.renderer` 对象的 `backgroundColor` 属性为一个任何的十六进制颜色：

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

如果你想要去找到 `画布` 的宽高，使用 `app.renderer.view.width` 和 `app.renderer.view.height` 。

使用 `画布` 的 `resize` 方法可以改变canvas的大小，提供任何新的 `width` 和 `height` 变量给他都行。但是为了确认宽高的格式正确，将 `autoResize` 设置为 `true` 。

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

如果你想让canvas占据整个窗口，你可以将这些CSS代码放在文档中，并且刷新你浏览器窗口的大小。

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

但是，如果你这么做了，要记得把padding和margin都设置成0:

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

(*这个通配符， 是CSS选择所有HTML元素的意思。)

如果你想要canvs在任何浏览器中统一尺寸，你可以使用 `scaleToWindow` 成员函数.

Pixi 精灵

Pixi 精灵

现在你就有了一个画布，可以开始往上面放图像了。所有你想在画布上显示的东西必须被加进一个被称作 `舞台` 的Pixi对象中。你能够像这样使用舞台对象：

```
1.  app.stage
```

这个 `舞台` 是一个Pixi `容器` 对象。你能把它理解成一种将放进去的东西分组并存储的空箱子。 `舞台` 对象是在你的场景中所有可见对象的根容器。所有你放进去的东西都会被渲染到canvas中。现在 `舞台` 是空的，但是很快我们就会放进去一点东西。 （你可以从这了解关于Pixi `容器` 对象的更多信息here）.

（重要信息：因为 `舞台` 是一个Pixi `容器` 对象，所以他有很多其他 `容器` 对象都有的属性和方法。但是，尽管舞台拥有 `width` 和 `height` 属性， 他们都不能查看画布窗口的大小 。舞台的 `width` 和 `height` 属性仅仅告诉了你你放进去的东西占用的大小 - 更多的信息在前面！）

所以你可以放些什么到舞台上呢？那就是被称作 精灵 的特殊图像对象。精灵是你能用代码控制图像的基础。你能够控制他们的位置，大小，和许多其他有用的属性来产生交互和动画。学习怎样创建和控制精灵是学习Pixi最重要的部分。如果你知道怎么创建精灵和把他们添加进舞台，离做出一个游戏就仅仅剩下一步之遥！

Pixi拥有一个 `精灵` 类来创建游戏精灵。有三种主要的方法来创建它：

- 用一个单图像文件创建。
- 用一个 雪碧图 来创建。雪碧图是一个放入了你游戏所需的所有图像的大图。
- 从一个纹理贴图集中创建。（纹理贴图集就是用JSON定义了图像大小和位置的雪碧图）

你将要学习这三种方式，但是在开始之前，你得弄明白图片怎么用Pixi显示。

把图像加载进纹理缓存

将图片加载到纹理缓存中

因为Pixi用WebGL和GPU去渲染图像，所以图像需要转化成GPU可以处理的版本。可以被GPU处理的图像被称作 纹理 。在你让精灵显示图片之前，需要将普通的图片转化成WebGL纹理。为了让所有工作执行的快速有效率，Pixi使用 纹理缓存 来存储和引用所有你的精灵需要的纹理。纹理的名称字符串就是图像的地址。这意味着如果你有从 `"images/cat.png"` 加载的图像，你可以在纹理缓存中这样找到他：

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

纹理被以WEBGL兼容的格式存储起来，它可以使Pixi的渲染有效率的进行。你现在可以使用Pixi的 `精灵` 类来创建一个新的精灵，让它使用纹理。

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

但是你该怎么加载图像并将它转化成纹理？答案是用Pixi已经构建好的 `loader` 对象。

Pixi强大的 `loader` 对象可以加载任何你需要种类的图像资源。这里展示了怎么加载一个图像并在加载完成时用一个叫做 `setup` 的方法来使用它。

```
1.  PIXI.loader
2.    .add("images/anyImage.png")
3.    .load(setup);
4.
```

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

Pixi的最佳实践

如果你使用了Loader，你就应该创建一个精灵来连接 `loader` 的 `resources` 对象，像下面这样：

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

这里是一个完整的加载图像的代码。调用 `setup` 方法，并未加载的图像创建一个精灵。

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

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

这是这个教程之中用来家在图像和创建精灵的通用方法。

你可以链式调用 `add` 方法来加载一系列图像，像下面这样：

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

更好的方式则是用数组给一个 `add` 方法传参，像这样：

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

这个 `loader` 也允许你使用JSON文件，关于JSON文件你应该已经在前面学过了。

显示精灵（sprite）

显示精灵

在你加载一个图像之后，可以用它来创建一个精灵，你需要用 `stage.addChild` 方法把它放到Pixi的 `舞台` 上面去，像这样：

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

记住， `舞台` 是用来包裹你所有精灵的主要容器。

重点：你不应该看见任何没被加入 `舞台` 的精灵

在我们继续之前，让我们看一个怎样使用显示一个单图像的例子。在 `examples/images` 文件夹中，你将找到一个64*64像素大小的猫的PNG图像文件。

这里是所有的显示一个图像，创建一个精灵，显示在Pixi的舞台上所需要的代码。

```
//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);
}
```

程序跑起来，你会看到：

现在我们已经取得了一些进展！

如果你想把一个精灵从舞台上挪走，就可以使用 `removeChild` 方法：

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

但是通常，我们都把精灵的 `visible` 属性设置成 `false` 来让精灵简单的隐藏。

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

使用别名

你可以对你使用频繁的Pixi对象和方法设置一些简略的可读性更强的别名。举个例子，你想给所有的Pixi对象增加 `PIXI` 前缀么？如果你这样想，那就创建一个简短的别名给他吧。下面是一个给 `TextureCache` 对象创建别名的例子：

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

现在就可以像这样使用别名了：

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

使用别名给写出简洁的代码提供了额外的好处：他帮助你缓存了Pixi的常用API。如果Pixi的API在将来的版本里改变了 - 没准他真的会变！ - 你将会需要在一个地方更新这些对象和方法，你只用在工程的开头而不是所有的实例那里！所以Pixi的开发团队想要改变它的时候，你只用一步即可完成这个操作！

来看看怎么将所有的Pixi对象和方法改成别名之后，来重写加载和显示图像的代码。

```
//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);

```

```
25. //This `setup` function will run when the image has loaded
26. function setup() {
27.
28.   //Create the cat sprite
29.   let cat = new Sprite(resources["images/cat.png"].texture);
30.
31.   //Add the cat to the stage
32.   app.stage.addChild(cat);
33. }
```

大多数教程中的例子将会使用Pixi的别名来处理。除非另有说明，否则你可以假定下面所有的代码都使用了这些别名。

这就是你需要的所有的关于加载图像和创建精灵的知识。

一些关于加载的其他知识

我们的例子中的格式是加载图像和显示精灵的最佳实践。所以你可以安全的忽视这些章节直接看”定位精灵”。但是Pixi的加载器有一些你不常用的复杂功能。

使用普通的javaScript Img对象或canvas创建一个精灵

为了优化和效率我们常常选择从预加载的纹理缓存的纹理之中创建精灵。但是如果因为某些原因你需要从JavaScript的 `Image` 对象之中创建，你可以使用Pixi的 `BaseTexture` 和 `Texture` 类：

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

你可以使用 `BaseTexture.fromCanvas` 从任何已经存在canvas标签中创建纹理：

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

如果你想改变已经显示的精灵的纹理，使用 `texture` 属性，可以设置任何 `Texture` 对象，像下面这样：

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

你可以使用这个技巧在游戏发生一些重大变化时交互式的改变精灵的纹理。

给加载的文件设置别名

你可以给任何你加载的源文件分配一个独一无二的别名。你只需要在 `add` 方法中第一个参数位置传进去这个别名就行了，举例来说，下面实现了怎么给这个猫的图片重命名为 `catImage` 。

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

这种操作在 `loader.resources` 中创建了一个叫做 `catImage` 的对象。这意味着你可以创建一个引用了 `catImage` 对象的精灵，像这样：

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

然而，我建议你永远别用这个操作！因为你将不得不记住你所有加载文件的别名，而且必须确信你只用了它们一次，使用路径命名，我们将将这些事情处理的更简单和更少错误。

监视加载进程

Pixi的加载器有一个特殊的 `progress` 事件，它将会调用一个可以定制的函数，这个函数将在每次文件加载时调用。 `progress` 事件将会

被 `loader` 的 `on` 方法调用，像是这样：

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

这里展示了怎么将 `on` 方法注入加载链中，并且每当文件加载时调用一个用户定义的名叫 `loadProgressHandler` 的函数。

```
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");
}
```

每一个文件加载，progress事件调用 `loadProgressHandler` 函数在控制台输出 “loading”。当三个文件都加载完毕， `setup` 方法将会运行，下面是控制台的输出：

```
loading
loading
loading
setup
```

这就不错了，不过还能变的更好。你可以知道哪个文件被加载了以及有百分之多少的文件被加载了。你可以在 `loadProgressHandler` 增

加 `loader` 参数和 `resource` 参数实现这个功能，像下面这样：

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

你现在可以使用 `resource.url` 变量来找到现在已经被加载的文件。（如果你想找到你定义的别名，使用resource.name参数。）你可以使用 `loader.progress` 来找到现在有百分之多少的文件被加载了，这里有一些关于上面描述的代码：

```
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");
}
```

这里是程序运行后的控制台显示：

```
loading: images/one.png
progress: 33.333333333333336%
loading: images/two.png
progress: 66.66666666666667%
loading: images/three.png
progress: 100%
All files loaded
```

这实在太酷了！因为你能用这个玩意做个进度条出来。
（注意：还有一些额外的 `resource` 对象属性， `resource.error` 会告诉你有哪些加载时候的错误， `resource.data` 将会给你文件的原始二进制数据。 ）

一些关于Pixi的加载器的其他知识

Pixi的加载器有很多可以设置的功能，让我速览一下：

`add` 方法有四个基础参数：

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

这里有文档里面对这些参数的描述：

`name` (string): 加载源文件的别名,如果没设置， `url` 就会被放在这.
`url` (string): 源文件的地址，是加载器 `baseUrl` 的相对地址.
`options` (object literal): 加载设置.
`options.crossOrigin` (Boolean): 源文件请求跨域不？默认是自动设定的。
`options.loadType` : 源文件是怎么加载进来的？默认是 `Resource.LOAD_TYPE.XHR` 。
`options.xhrType` : 用XHR的时候该怎么处理数据？ 默认

是 `Resource.XHR_RESPONSE_TYPE.DEFAULT` 。

`callbackFunction` ： 当这个特定的函数加载完，这个特定的函数将会被执行。

只有 `url` 必填（你总得加载个文件吧。）

这里有点用了 `add` 方法加载文件的例子。第一个就是文档里所谓的“正常语法”：

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

这些就是所谓“对象语法”啦：

```
.add({
  name: 'key2',
  url: 'http://...'
}, function () {})

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

.add({
  name: 'key3',
  url: 'http://...'
  onComplete: function () {}
})

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

你也可以给 `add` 方法传一个对象的数组，或者既使用对象数组，又使用链式加载：

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

（注意：如果你需要重新加载一批文件，调用加载器的 `reset` 方法： `PIXI.loader.reset();` ）

Pixi的加载器还有许多其他的高级特性，包括可以让你加载和解析所有类型二进制文件的选项。这些并非你每天都要做的，也超出了这个教程的范围，所以从GitHub项目中获取更多信息吧！

精灵位置

精灵位置

现在你知道了怎么创建和显示一个精灵，让我们学习如何定位他们的位置和改变他们的大小
在最早的示例里那个猫的精灵被放在了舞台的左上角。它的 `x` 和 `y` 坐标都是0。你可以通过改变它的 `x` 和 `y` 坐标的值来改变他们的位置。下面的例子就是你通过设置 `x` 和 `y` 为96坐标让它在舞台上居中。

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

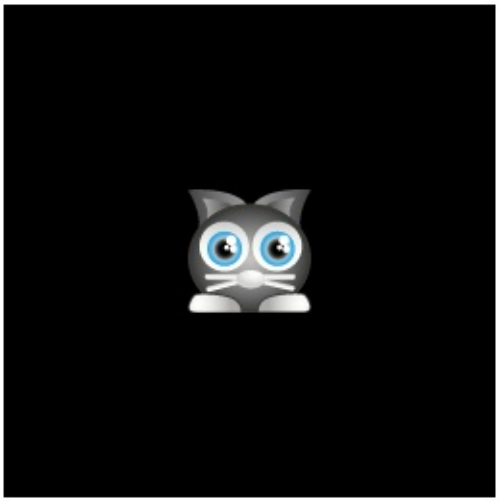
在你创建这个精灵之后，把这两行代码放进 `setup` 方法。

```
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);
}
```

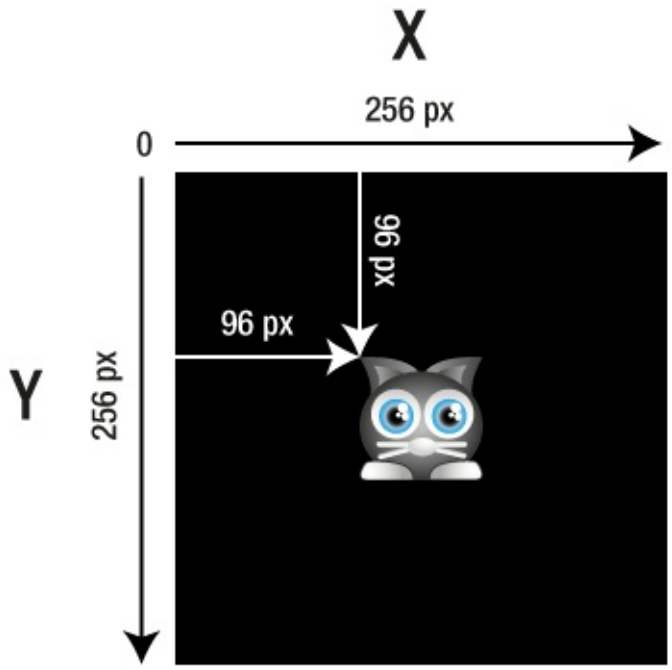
（注意：在这个例子里，`Sprite` 是 `PIXI.Sprite` 的别名，`TextureCache` 是 `PIXI.utils.TextureCache` 的别名，`resources` 是 `PIXI.loader.resources` 的别名，我从现在开始在代

码中使用这些别名。）

这两行代码将把猫往右移动96像素，往下移动96像素。



这只猫的左上角（它的左耳朵）（译者注：从猫的角度看其实是它的右耳朵。。。）表示了它的 `x` 和 `y` 坐标点。为了让他向右移动，增加 `x` 这个属性的值，为了让他向下移动，就增加 `y` 属性的值。如果这只猫的 `x` 属性为0，他就呆在舞台的最左边，如果他的 `y` 属性为0，他就呆在舞台的最上边。



你可以一句话设置精灵的 `x` 和 `y`：

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

大小和比例

大小和比例

你能够通过精灵的 `width` 和 `height` 属性来改变它的大小。这是怎么把 `width` 调整成80像素， `height` 调整成120像素的例子：

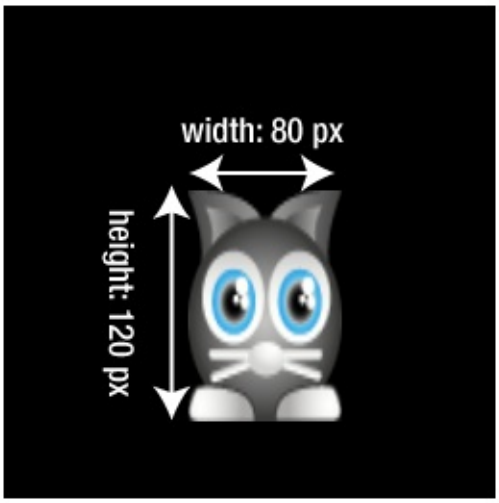
```
cat.width = 80;
cat.height = 120;
```

在 `setup` 函数里面加上这两行代码，像这样：

```
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);
}
```

结果看起来是这样：

你能看见，这只猫的位置（左上角的位置）没有改变，只有宽度和高度改变了。



精灵都有 `scale.x` 和 `scale.y` 属性，他们可以成比例的改变精灵的宽高。这里的例子把猫的大小变成了一半：

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

Scale的值是从0到1之间的数字的时候，代表了它对于原来精灵大小的百分比。1意味着100%（原来的大小），所以0.5意味着50%（一半大小）。你可以把这个值改为2，这就意味着让精灵的大小成倍增长。像

这样：

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

Pixi可以用一行代码缩放你的精灵，那要用到 `scale.set` 方法。

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

如果你喜欢这种，就用吧！

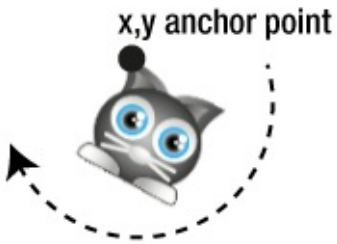
角度

旋转

你可以通过对一个精灵的 `rotation` 设定一个角度来旋转它。

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

但是旋转是针对于哪一个点发生的呢？
你已经了解了，精灵的左上角代表它的位置，这个点被称之为 锚点 。如果你用像 `0.5` 这种值设定 `rotation` ，这个旋转将会 围绕着锚点发生 。下面这张图就是结果：

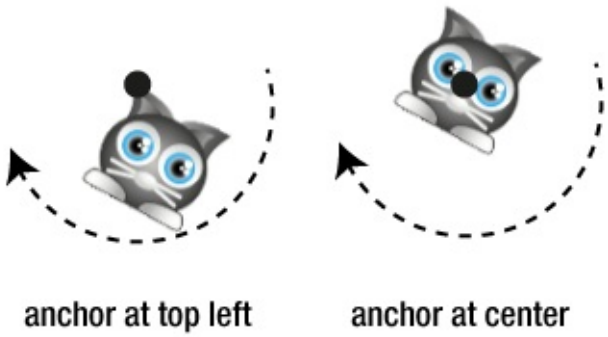


你能看见锚点是猫的左边耳朵（译者:对猫来说实际上是它的右耳朵！），那里成了猫的图片的旋转中心。
你该怎么改变锚点呢？通过改变精灵的 `anchor` 属性的xy值来实现。像下面这样：

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

`anchor.x` 和 `anchor.y` 的值如果是从0到1，就会被认为是整个纹理的长度或宽度百分比。设置他们都为0.5，锚点就处在了图像中心。精灵定位的依据点不会改变，锚点的改变是另外一回事。

下面的图显示把锚点居中以后旋转的精灵。



你可以看到精灵的纹理向左移动了，这是个必须记住的重要副作用！

像是 `position` 和 `scale` 属性一样，你也可以在一行内像这样设置锚点的位置：

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

精灵也提供和 `anchor` 差不多的 `pivot` 属性来设置精灵的原点。如果你改变了它的值之后旋转精灵，它将会围绕着你设置的原点来旋转。举个例子，下面的代码将精灵的 `pivot.x` 和 `pivot.y` 设置为了32。

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

假设精灵图是64x64像素，它将绕着它的中心点旋转。但是记住：你如果改变了精灵的 `pivot` 属性，你也就改变了它的原点位置。

所以 `anchor` 和 `pivot` 的不同之处在哪里呢？他们真的很像！ `anchor` 改变了精灵纹理的图像原点，用0到1的数据来填充。 `pivot` 则改变了精灵的原点，用像素的值来填充。你要用哪个取决于你。两个都试试就知道哪个对你而言最适合。

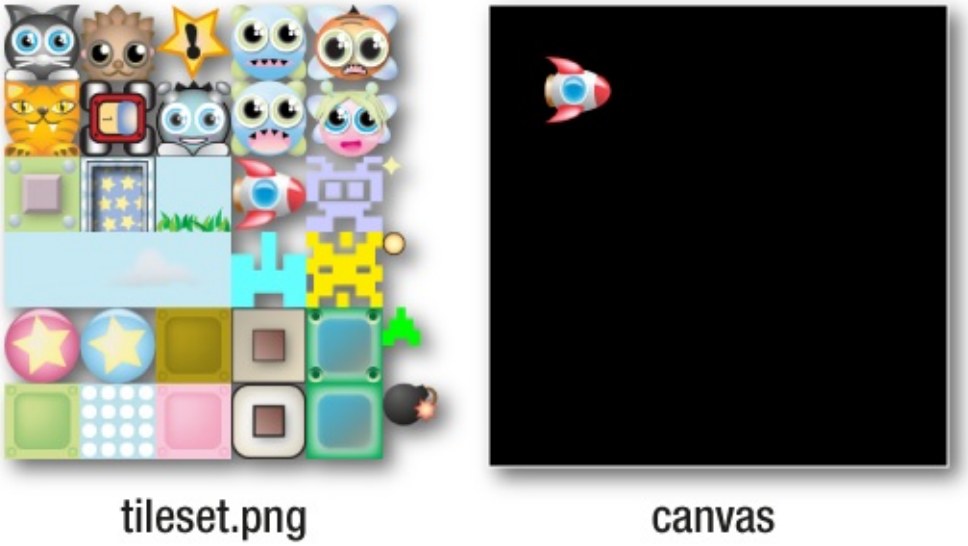
从精灵图（雪碧图）中获取精灵

从精灵图（雪碧图）中获取精灵

从精灵图（雪碧图）中创建精灵【为了防止与精灵混淆，我在之后的译文中都将采用雪碧图这一译法】

你现在已经知道了怎么从一个单文件内加载图像。但是作为一个游戏设计师，你没准更经常使用 雪碧图（也被称之为 精灵图）。Pixi封装了一些方便的方式来处理这种情况。所谓雪碧图就是用一个单文件包含你游戏中需要的所有文件，这里就是一个包含了游戏对象和游戏觉得的雪碧图。

整个雪碧图是192*192*像素宽高，但每一个单图像只占有一个*32*32的网格。把你的所有游戏图像存储在一个雪碧图上是一个非常有效率和工程化的手段，Pixi为此做出了优化。你可以从一个雪碧图中用一个矩形区域捕获一个子图像。这个矩形拥有和你想提取的子图像一样的大小和位置。这里有一个怎么从一个精灵图中获取“火箭”这个子图像的例子。



让我们看看这部分的代码，用Pixi的 `加载器` 加载 `tileset.png` ，就像你在之前的示例之中做到的那样。

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

现在，在图像被加载之后，用一个矩形块去截取雪碧图来创建精灵的纹理。下面是提取火箭，创建精灵，在canvs上显示它的代码。

```
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;

```

```
14.    //Create the sprite from the texture
15.    let rocket = new Sprite(texture);
16.
17.    //Position the rocket sprite on the canvas
18.    rocket.x = 32;
19.    rocket.y = 32;
20.
21.    //Add the rocket to the stage
22.    app.stage.addChild(rocket);
23.
24.    //Render the stage
25.    renderer.render(stage);
26.  }
```

它是如何工作的呢？

Pixi内置了一个通用的 `Rectangle` 对象 （ `PIXI.Rectangle` ），他是一个用于定义矩形形状的通用对象。他需要一些参数，前两个参数定义了 `x` 和 `y` 轴坐标位置，后两个参数定义了矩形的 `width` 和 `height` ，下面是新建一个 `Rectangle` 对象的格式。

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

这个矩形对象仅仅是一个 数据对象，如何使用它完全取决于你。在我们的例子里，我们用它来定义子图像在雪碧图中的位置和大小。Pixi的纹理中有一个叫做 `frame` 的很有用的属性，它可以被设置成任何的 `Rectangle` 对象。 `frame` 将纹理映射到 `Rectangle` 的维度。下面是怎么用 `frame` 来定义火箭的大小和位置。

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

你现在可以用它裁切纹理来创建精灵了：

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

现在成功了！

因为从一个雪碧图创建精灵的纹理是一个用的很频繁的操作，Pixi有一个更加合适的方式来帮助你处理这件事情。欲知后事如何，且听下回分解。

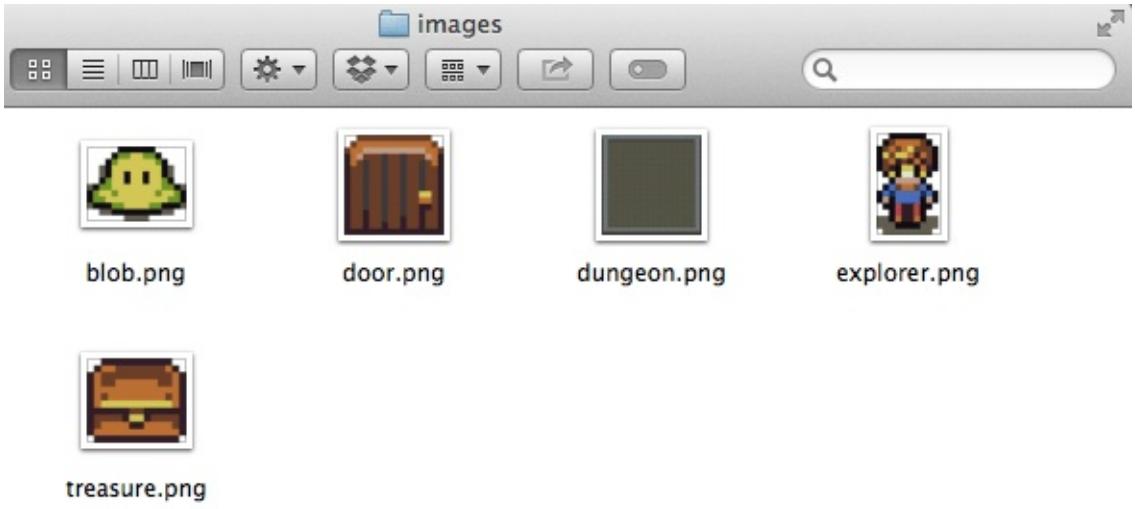
使用一个纹理贴图集

使用一个纹理贴图集

如果你正在处理一个很大的，很复杂的游戏，你想要找到一种快速有效的方式来从雪碧图创建精灵。纹理贴图集 就会显得很有用处，一个纹理贴图集就是一个JSON数据文件，它包含了匹配的PNG雪碧图的子图像的大小和位置。如果你使用了纹理贴图集，那么想要显示一个子图像只需要知道它的名字就行了。你可以任意的排序你的排版，JSON文件会保持他们的大小和位置不变。这非常方便，因为这意味着图片的位置和大小不必写在你的代码里。如果你想要改变纹理贴图集的排版，类似增加图片，修改图片大小和删除图片这些操作，只需要修改那个JSON数据文件就行了，你的游戏会自动给程序内的所有数据应用新的纹理贴图集。你没必要在所有用到它代码的地方修改它。

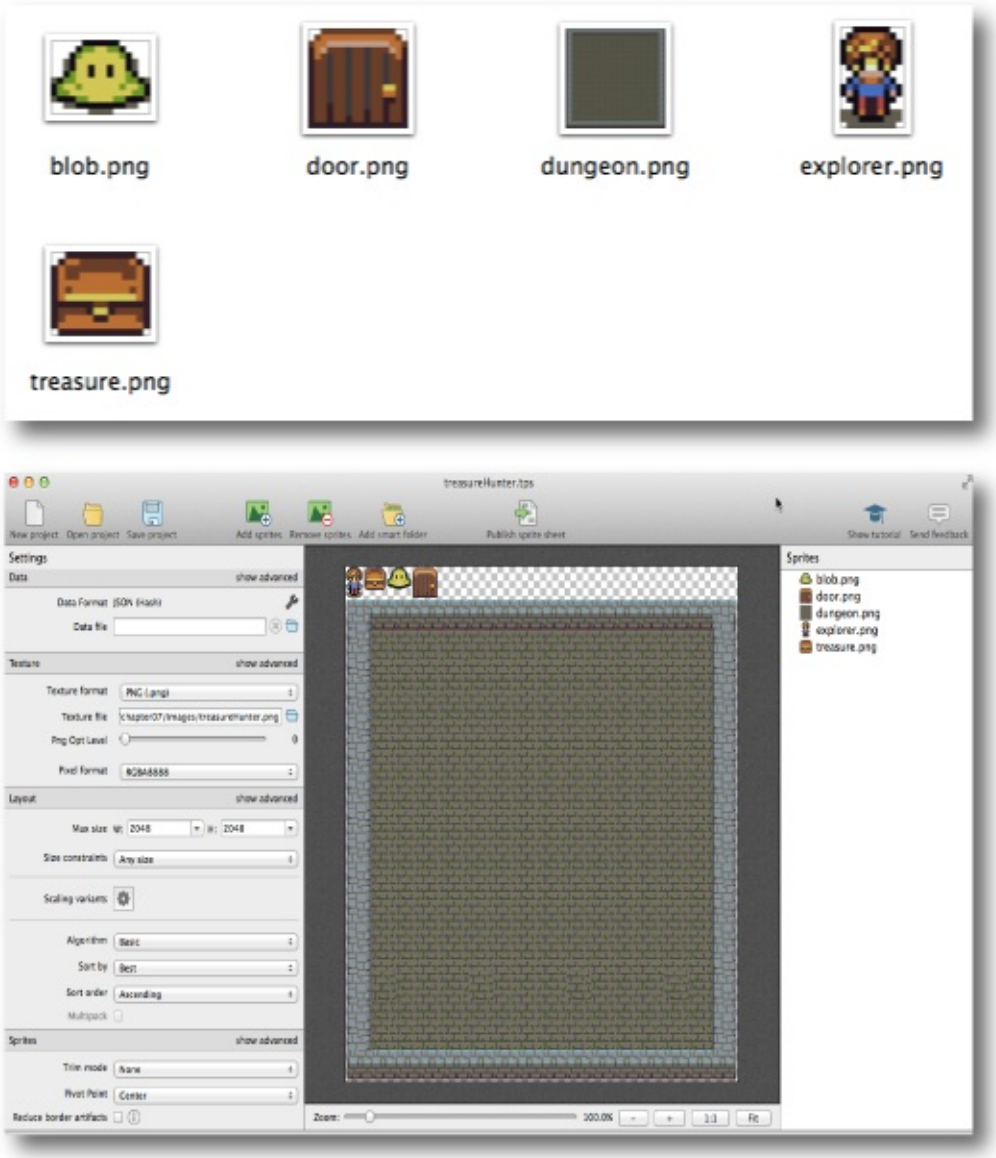
Pixi兼容著名软件Texture Packer输出的标准纹理贴图集格式。Texture Packer的基本功能是免费的。让我们来学习怎么用它来制作一个纹理贴图集，并把它加载进Pixi吧！（你也不是非得用它，还有一些类似的工具输出的纹理贴图集Pixi也是兼容的，例如：Shoebox和spritesheet.js。）

首先，从你要用在游戏的图片文件们开始。



在这个章节所有的图片都是被Lanea Zimmerman创作的。你能在他的艺术工作室里面找到更多类似的东西：这里，谢谢你，Lanea！

下面，打开Texture Packer，选择 **JSON Hash** 框架类型。把你的图片放进Texture Packer的工作区。（你也可以把Texture Packer放进包含你图片的文件夹里面去。）他将自动的把你的图片们生成单个图片文件，并且将他们的原始名称命名为纹理贴图集中的图片名称。



如果你正在用免费版的Texture Packer，把 **Algorithm** 选项设为 `Basic` ，把 **Trim mode** 选项设为 `None` ，把 **Extrude** 选项设为 `0` ，把 **Size constraints** 选项设为 `Any size` ，把 **PNG Opt Level** 中所有的东西都滑到左边的 `0` 位置。这就可以使得Texture Packer正常的输出你的纹理贴图集。

如果你做完了，点击 **Publish** 按钮。选择输出文件名和存储地址，把生成文件保存起来。你将会获得两个文件：一个叫做 `treasureHunter.json` ，另外一个就是 `treasureHunter.png` 。为了让目

录干净些，我们把他俩都放到一个叫做 `images` 的文件夹里面去。（你可以认为那个json文件是图片文件的延伸，所以把他们放进一个文件夹是很有意义的。）那个JSON文件里面写清楚了每一个子图像的名字，大小和位置。下面描述了“泡泡怪”这个怪物的子图像的信息。

```
1.  "blob.png":
2.  {
3.      "frame": {"x":55,"y":2,"w":32,"h":24},
4.      "rotated": false,
5.      "trimmed": false,
6.      "spriteSourceSize": {"x":0,"y":0,"w":32,"h":24},
7.      "sourceSize": {"w":32,"h":24},
8.      "pivot": {"x":0.5,"y":0.5}
9.  },
```

`treasureHunter.json` 里面也包含了“dungeon.png”, “door.png”, “exit.png”, 和 “explorer.png”的数据信息，并以和上面类似的信息记录。这些子图像每一个都被叫做 帧 ，有了这些数据你就不用去记每一个图片的大小和位置了，你唯一要做的就只是确定精灵的 帧**ID** 即可。帧ID就是那些图片的原始名称，类似”blob.png”或者 “explorer.png”这样。

使用纹理贴图集的巨大优势之一就是你可以很轻易的给每一个图像增加两个像素的内边距。Texture Packer默认这么做。这对于保护图像的 出血（译者：出血是排版和图片处理方面的专有名词，指在主要内容周围留空以便印刷或裁切）来说很重要。出血对于防止两个图片相邻而相互影响来说很重要。这种情况往往发生于你的GPU渲染某些图片的时候。把边上的一两个像素加上去还是不要？这对于每一个GPU来说都有不同的做法。所以对每一个图像空出一两个像素对于显示来说是最好的兼容。

（注意：如果你真的在每个图像的周围留了两个像素的出血，你必须时

时刻刻注意Pixi显示时候“丢了一个像素”的情况。尝试着去改变纹理的规模模式来重新计算它。 `texture.baseTexture.scaleMode = PIXI.SCALE_MODES.NEAREST;` ，这往往发生于你的GPU浮点运算凑整失败的时候。)

现在你明白了怎么创建一个纹理贴图集，来学习怎么把他加载进你的游戏之中吧。

加载纹理贴图集

加载纹理贴图集

可以使用Pixi的 `loader` 来加载纹理贴图集。如果是用Texture Packer生成的JSON， `loader` 会自动读取数据，并对每一个帧创建纹理。下面就是怎么用 `loader` 来加载 `treasureHunter.json` 。当它成功加载， `setup` 方法将会执行。

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

现在每一个图像的帧都被加载进Pixi的纹理缓存之中了。你可以使用Texture Packer中定义的他们的名字来取用每一个纹理。

从一个纹理贴图集创建精灵

从已经加载的纹理贴图集中创建精灵

通常Pixi给你三种方式从已经加载的纹理贴图集中创建精灵：

1. 使用 `TextureCache`：

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

2. 如果你是使用的 `loader`来加载纹理贴图集，使用loader的 `resources`：

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

3. 要创建一个精灵需要输入太多东西了！
   所以我建议你给纹理贴图集的`textures`对象创建一个叫做`id`的别名，象是这样：

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

   现在你就可以像这样实例化一个精灵了：

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

   真不错啊~！

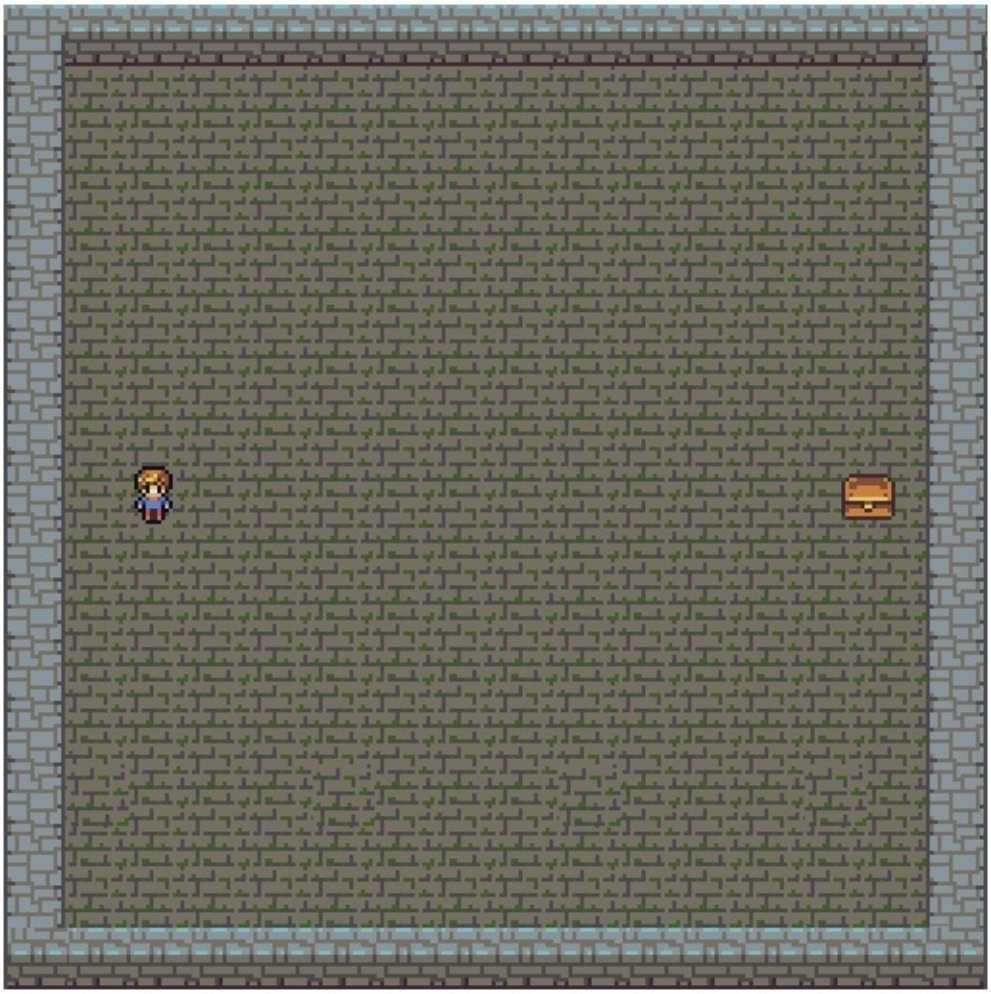
这里在 `setup` 函数中用三种不同的创建方法创建和显示了 `dungeon`，`explorer`，和 `treasure` 精灵。

```

```

```
2.  //Define variables that might be used in more
3.  //than one function
4.  let dungeon, explorer, treasure, id;
5.
6.  function setup() {
7.
8.    //There are 3 ways to make sprites from textures atlas frames
9.
10.   //1. Access the `TextureCache` directly
11.   let dungeonTexture = TextureCache["dungeon.png"];
12.   dungeon = new Sprite(dungeonTexture);
13.   app.stage.addChild(dungeon);
14.
15.   //2. Access the texture using throuhg the loader's `resources`:
16.   explorer = new Sprite(
17.
    resources["images/treasureHunter.json"].textures["explorer.png"]
18.   );
19.   explorer.x = 68;
20.
21.   //Center the explorer vertically
22.   explorer.y = app.stage.height / 2 - explorer.height / 2;
23.   app.stage.addChild(explorer);
24.
25.   //3. Create an optional alias called `id` for all the texture
    atlas
26.   //frame id textures.
27.   id =
    PIXI.loader.resources["images/treasureHunter.json"].textures;
28.
29.   //Make the treasure box using the alias
30.   treasure = new Sprite(id["treasure.png"]);
31.   app.stage.addChild(treasure);
32.
33.   //Position the treasure next to the right edge of the canvas
34.   treasure.x = app.stage.width - treasure.width - 48;
35.   treasure.y = app.stage.height / 2 - treasure.height / 2;
36.   app.stage.addChild(treasure);
```

```
37. }
```

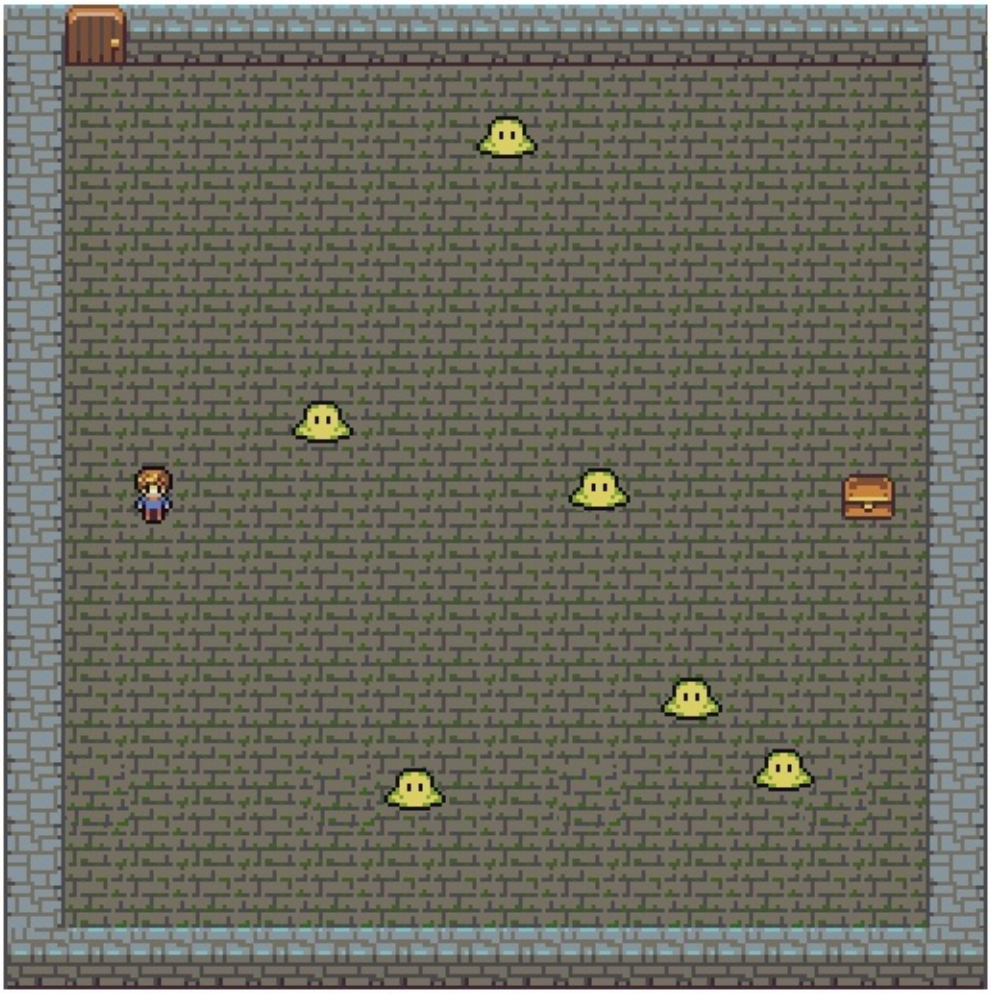
这里是代码运行的结果：



舞台定义为512像素见方的大小，你可以看到代码中 `app.stage.height` 和 `app.stage.width` 属性使得精灵们排成了一排。下面的代码使得 `explorer` 的 `y` 属性垂直居中了。

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

学会使用纹理贴图集来创建一个精灵是一个基本的操作。所以在我们继续之前，你来试着写一些这样的精灵吧： `blob` 们和 `exit` 的门，让他们看起来象是这样：



下面就是所有的代码啦。我也把HTML放了进来，现在你可以看见所有的上下文。（你可以在 `examples/spriteFromTextureAtlas.html` 找到可以用于演示的代码。）注意， `blob` 精灵是用一个循环加进舞台的，并且他有一个随机的位置。

```
<!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);

```

```
48.   //2. Access the texture using throuhg the loader's `resources`:
49.   explorer = new Sprite(
50.
    resources["images/treasureHunter.json"].textures["explorer.png"]
51.   );
52.   explorer.x = 68;
53.
54.   //Center the explorer vertically
55.   explorer.y = app.stage.height / 2 - explorer.height / 2;
56.   app.stage.addChild(explorer);
57.
58.   //3. Create an optional alias called `id` for all the texture
    atlas
59.   //frame id textures.
60.   id =
    PIXI.loader.resources["images/treasureHunter.json"].textures;
61.
62.   //Make the treasure box using the alias
63.   treasure = new Sprite(id["treasure.png"]);
64.   app.stage.addChild(treasure);
65.
66.   //Position the treasure next to the right edge of the canvas
67.   treasure.x = app.stage.width - treasure.width - 48;
68.   treasure.y = app.stage.height / 2 - treasure.height / 2;
69.   app.stage.addChild(treasure);
70.
71.   //Make the exit door
72.   door = new Sprite(id["door.png"]);
73.   door.position.set(32, 0);
74.   app.stage.addChild(door);
75.
76.   //Make the blobs
77.   let numberOfBlobs = 6,
78.       spacing = 48,
79.       xOffset = 150;
80.
81.   //Make as many blobs as there are `numberOfBlobs`
82.   for (let i = 0; i < numberOfBlobs; i++) {
```

```
83.
84.     //Make a blob
85.     let blob = new Sprite(id["blob.png"]);
86.
87.     //Space each blob horizontally according to the `spacing`
    value.
88.     //`xOffset` determines the point from the left of the screen
89.     //at which the first blob should be added.
90.     let x = spacing * i + xOffset;
91.
92.     //Give the blob a random y position
93.     //(`randomInt` is a custom function - see below)
94.     let y = randomInt(0, app.stage.height - blob.height);
95.
96.     //Set the blob's position
97.     blob.x = x;
98.     blob.y = y;
99.
100.    //Add the blob sprite to the stage
101.    app.stage.addChild(blob);
102.  }
103. }
104.
105. //The `randomInt` helper function
106. function randomInt(min, max) {
107.   return Math.floor(Math.random() * (max - min + 1)) + min;
108. }
109.
110. </script>
111. </body>
```

你可以看见所有的泡泡怪都用一个 `for` 循环被创建了，每一个泡泡怪都有一个独一无二的 `x` 坐标，像是下面这样：

```
1. let x = spacing * i + xOffset;
2. blob.x = x;
```

`spacing` 变量的值是48， `xOffset` 的值是150。这意味着第一个 `blob` 怪的位置的 `x` 坐标将会是150。这个偏移使得泡泡怪离舞台左边的距离有150个像素。每一个泡泡怪都有个48像素的空余，也就是说每一个泡泡怪都会比在循环之中前一个创建的泡泡怪的位置的 `x` 坐标多出48像素以上的增量。它使得泡泡怪们相互间隔，从地牢地板的左边排向右边。

每一个 `blob` 也被赋予了一个随机的 `y` 坐标，这里是处理这件事的代码：

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

泡泡怪的 `y` 坐标将会从0到512之间随机取值，它的变量名是 `stage.height` 。它的值是利用 `randomInt` 函数来得到的。 `randomInt` 返回一个由你定义范围的随机数。

```
1.  randomInt(lowestNumber, highestNumber)
```

这意味着如果你想要一个1到10之间的随机数，你可以这样得到它：

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

这是 `randomInt` 方法的定义：

```
1.  function randomInt(min, max) {
2.    return Math.floor(Math.random() * (max - min + 1)) + min;
3.  }
```

`randomInt` 是一个很好的用来做游戏的工具函数，我经常用他。

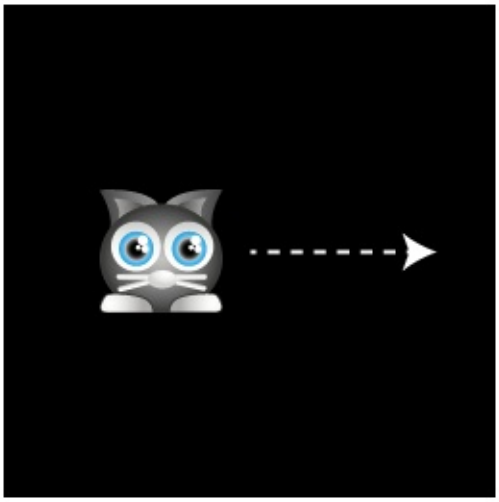
移动精灵

移动精灵

现在你知道了如何展示精灵，但是让它们移动呢？很简单：使用Pixi的 `ticker` 。这被称为 游戏循环 。任何在游戏循环里的代码都会1秒更新60次。你可以用下面的代码让 `cat` 精灵以每帧1像素的速率移动。

```
1.
2.  function setup() {
3.
4.    //Start the game loop by adding the `gameLoop` function to
5.    //Pixi's `ticker` and providing it with a `delta` argument.
6.    app.ticker.add(delta => gameLoop(delta));
7.  }
8.
9.  function gameLoop(delta){
10.
11.   //Move the cat 1 pixel
12.   cat.x += 1;
13. }
```

如果你运行了上面的代码，你会看到精灵逐步地移动到舞台的一边。



因为每当开始 `游戏循环` 的时候，都会为这只猫增加1像素的x轴位移。

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

每一个你放进Pixi的 `ticker` 的函数都会每秒被执行60次。你可以看见函数里面提供了一个 `delta` 的内容，他是什么呢？

`delta` 的值代表帧的部分的延迟。你可以把它添加到cat的位置，让cat的速度和帧率无关。下面是代码：

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

是否加进去这个 `delta` 的值其实是一种审美的选择。它往往只在你的动画没法跟上60帧的速率时候出现（比如你的游戏运行在很老旧的机器上）。教程里面不会用到 `delta` 变量，但是如果你想用就尽情的用吧。

你也没必要非得用Pixi的ticker来创建游戏循环。如果你喜欢，也可以用 `requestAnimationFrame` 像这样创建：

```
1.  function gameLoop() {
```

```
2.
3.    //Call this `gameLoop` function on the next screen refresh
4.    //(which happens 60 times per second)
5.    requestAnimationFrame(gameLoop);
6.
7.    //Move the cat
8.    cat.x += 1;
9.  }
10.
11. //Start the loop
12. gameLoop();
```

随你喜欢。

这就是移动的全部。只要在循环中改变精灵的一点点属性，它们就会开始相应的动画。如果你想让它往相反的方向移动，只要给它一个负值，像 `-1`。

你能在 `movingSprites.html` 文件中找到这段代码 - 这是全部的代码：

```
1.  //Aliases
2.  let Application = PIXI.Application,
3.      Container = PIXI.Container,
4.      loader = PIXI.loader,
5.      resources = PIXI.loader.resources,
6.      TextureCache = PIXI.utils.TextureCache,
7.      Sprite = PIXI.Sprite,
8.      Rectangle = PIXI.Rectangle;
9.
10. //Create a Pixi Application
11. let app = new Application({
12.     width: 256,
13.     height: 256,
14.     antialias: true,
15.     transparent: false,
16.     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;
}
```

（注意 `cat` 变量需要在 `setup` 和 `gameLoop` 函数之外定义，然后你可以在全局中任何地方都能获取到它们）

你可以让精灵的位置，角度或者大小动起来 - 什么都可以！你会在下

面看到更多精灵动画的例子。

使用速度属性

使用速度属性

为了给你更多的灵活性，这里有两个 速度属性 ： `vx` 和 `vy` 去控制精灵的运动速度。 `vx` 被用来设置精灵在x轴（水平）的速度和方向。 `vy` 被用来设置精灵在y轴（垂直）的速度和方向。 他们可以直接更新速度变量并且给精灵设定这些速度值。这是一个用来让你更方便的更新交互式动画的额外的模块。

第一步是给你的精灵创建 `vx` 和 `vy` 属性，然后给他们初始值。

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

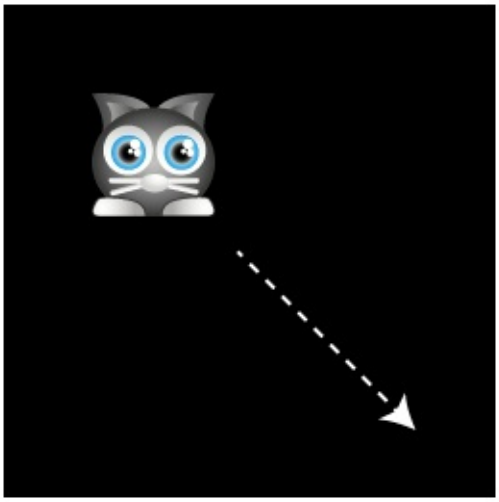
给 `vx` 和 `vy` 设置为0表示精灵静止。

接下来，在游戏循环中，更新 `vx` 和 `vy` 为你想让精灵移动的速度值。然后把这些值赋给精灵的 `x` 和 `y` 属性。下面的代码讲明了你如何利用该技术让cat能够每帧向右下方移动一个像素：

```
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));
}
```

```
13.
14. function gameLoop(delta){
15.
16.   //Update the cat's velocity
17.   cat.vx = 1;
18.   cat.vy = 1;
19.
20.   //Apply the velocity values to the cat's
21.   //position to make it move
22.   cat.x += cat.vx;
23.   cat.y += cat.vy;
24. }
```

当你运行这段代码，猫会每帧向右下方移动一个像素：：



如果你想让猫往不同的方向移动怎么办？可以把它的 `vx` 赋值为 `-1` 让猫向左移动。可以把它的 `vy` 赋值为 `-1`让猫向上移动。为了让猫移动的更快一点，把值设的更大一点，像`3`, `5`, `-2`, 或者 `-4`。

你会在前面看到如何通过利用 `vx` 和 `vy` 的速度值来模块化精灵的速度，它对游戏的键盘和鼠标控制系统很有帮助，而且更容易实现物理模拟。

游戏状态

游戏状态

作为一种代码风格，也为了帮你模块你的代码，我推荐在游戏循环里像这样组织你的代码：

```
1.  //Set the game state
2.  state = play;
3.
4.  //Start the game loop
5.  app.ticker.add(delta => gameLoop(delta));
6.
7.  function gameLoop(delta){
8.
9.    //Update the current game state:
10.   state(delta);
11. }
12.
13. function play(delta) {
14.
15.   //Move the cat 1 pixel to the right each frame
16.   cat.vx = 1
17.   cat.x += cat.vx;
18. }
```

你会看到 `gameLoop` 每秒60次调用了 `state` 函数。 `state` 函数是什么呢？它被赋值为 `play` 。意味着 `play` 函数会每秒运行60次。

下面的代码告诉你如何用这个新模式来重构上一个例子的代码：

```
1.  //Define any variables that are used in more than one function
2.  let cat, state;
3.
```

```
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;
}
```

是的，我知道这有点儿 `head-swirler`！但是，不要害怕，花几分钟在脑海中想一遍这些函数是如何联系在一起的。正如你将在下面看到的，结构化你的游戏循环代码，会使得切换游戏场景和关卡这种操作变得更简单。

键盘响应

键盘移动

只需再做一点微小的工作，你就可以建立一个通过鼠标控制精灵移动的简单系统。为了简化你的代码，我建议你用一个名为 `keyboard` 的自定义函数来监听和捕捉键盘事件。

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

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

```

```
28.   //Attach event listeners
29.   window.addEventListener(
30.     "keydown", key.downHandler.bind(key), false
31.   );
32.   window.addEventListener(
33.     "keyup", key.upHandler.bind(key), false
34.   );
35.   return key;
36. }
```

`keyboard` 函数用起来很容易，可以像这样创建一个新的键盘对象：

```
1. let keyObject = keyboard(asciiKeyCodeNumber);
```

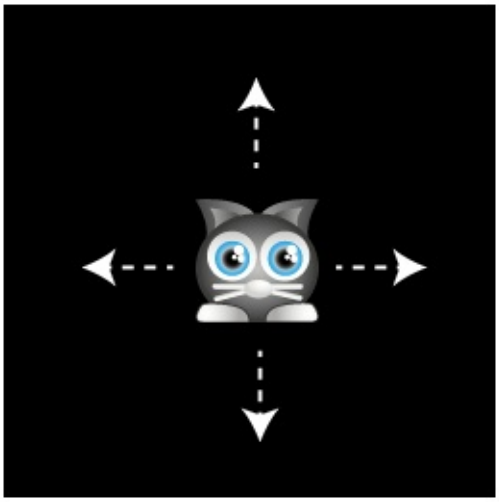
这个函数只接受一个参数就是键盘对应的ASCII键值数，也就是你想监听的键盘按键。 这是键盘键ASSII值列表).

然后给键盘对象赋值 `press` 和 `release` 方法：

```
1. keyObject.press = () => {
2.   //key object pressed
3. };
4. keyObject.release = () => {
5.   //key object released
6. };
```

键盘对象也有 `isDown` 和 `isUp` 的布尔值属性，你可以用它们来检查每个按键的状态。

在 `examples` 文件夹里看一下 `keyboardMovement.html` 文件是怎么用 `keyboard` 函数的，利用键盘的方向键去控制精灵图。运行它，然后用上下左右按键去让猫在舞台上移动。



这里是代码：

```
1.  //Define any variables that are used in more than one function
2.  let cat, state;
3.
4.  function setup() {
5.
6.    //Create the `cat` sprite
7.    cat = new Sprite(resources["images/cat.png"].texture);
8.    cat.y = 96;
9.    cat.vx = 0;
10.   cat.vy = 0;
11.   app.stage.addChild(cat);
12.
13.   //Capture the keyboard arrow keys
14.   let left = keyboard(37),
15.       up = keyboard(38),
16.       right = keyboard(39),
17.       down = keyboard(40);
18.
19.   //Left arrow key `press` method
20.   left.press = () => {
21.     //Change the cat's velocity when the key is pressed
22.     cat.vx = -5;
23.     cat.vy = 0;
24.   };
```

```
25.
26.   //Left arrow key `release` method
27.   left.release = () => {
28.     //If the left arrow has been released, and the right arrow
    isn't down,
29.     //and the cat isn't moving vertically:
30.     //Stop the cat
31.     if (!right.isDown && cat.vy === 0) {
32.       cat.vx = 0;
33.     }
34.   };
35.
36.   //Up
37.   up.press = () => {
38.     cat.vy = -5;
39.     cat.vx = 0;
40.   };
41.   up.release = () => {
42.     if (!down.isDown && cat.vx === 0) {
43.       cat.vy = 0;
44.     }
45.   };
46.
47.   //Right
48.   right.press = () => {
49.     cat.vx = 5;
50.     cat.vy = 0;
51.   };
52.   right.release = () => {
53.     if (!left.isDown && cat.vy === 0) {
54.       cat.vx = 0;
55.     }
56.   };
57.
58.   //Down
59.   down.press = () => {
60.     cat.vy = 5;
61.     cat.vx = 0;
```

```
62.   };
63.   down.release = () => {
64.     if (!up.isDown && cat.vx === 0) {
65.       cat.vy = 0;
66.     }
67.   };
68.
69.   //Set the game state
70.   state = play;
71.
72.   //Start the game loop
73.   app.ticker.add(delta => gameLoop(delta));
74. }
75.
76. function gameLoop(delta){
77.
78.   //Update the current game state:
79.   state(delta);
80. }
81.
82. function play(delta) {
83.
84.   //Use the cat's velocity to make it move
85.   cat.x += cat.vx;
86.   cat.y += cat.vy
87. }
```

将精灵分组

给精灵分组

分组让你能够让你创建游戏场景，并且像一个单一单元那样管理相似的精灵图。Pixi有一个对象叫 `Container` ，它可以帮你做这些工作。让我们弄清楚它是怎么工作的。

想象一下你想展示三个精灵：一只猫，一只刺猬和一只老虎。创建它们，然后设置它们的位置 - 但是不要把它们添加到舞台上。

```
1.  //The cat
2.  let cat = new Sprite(id["cat.png"]);
3.  cat.position.set(16, 16);
4.
5.  //The hedgehog
6.  let hedgehog = new Sprite(id["hedgehog.png"]);
7.  hedgehog.position.set(32, 32);
8.
9.  //The tiger
10. let tiger = new Sprite(id["tiger.png"]);
11. tiger.position.set(64, 64);
```

让后创建一个 `animals` 容器像这样去把他们聚合在一起：

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

然后用 `addChild` 去把精灵图 添加到分组中 。

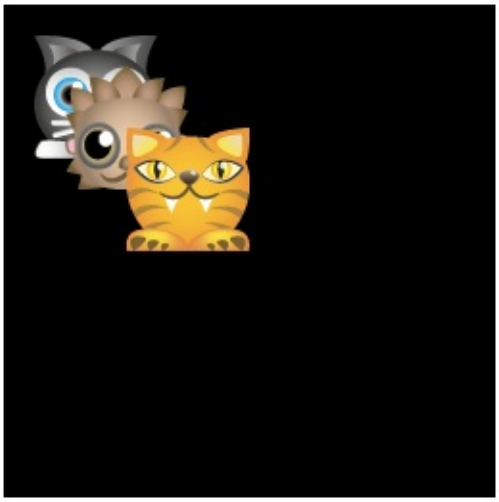
```
1. animals.addChild(cat);
2. animals.addChild(hedgehog);
3. animals.addChild(tiger);
```

最后把分组添加到舞台上。

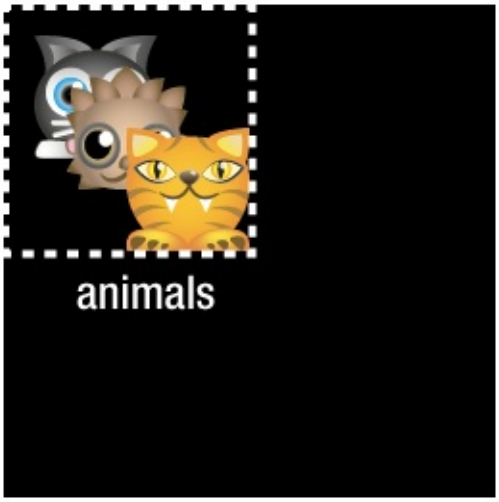
```
1. app.stage.addChild(animals);
```

（你知道的，`stage`对象也是一个`Container`。它是所有Pixi精灵的根容器。）

这就是上面代码的效果：



你是看不到这个包含精灵图的`animals`分组的。它仅仅是个容器而已。



不过你现在可以像对待一个单一单元一样对待 `animals` 分组。你可以把 `Container` 当作是一个特殊类型的不包含任何纹理的精灵。

如果你需要获取 `animals` 包含的所有子精灵，你可以用它的 `children` 数组获取。

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

这告诉你 `animals` 有三个子精灵。

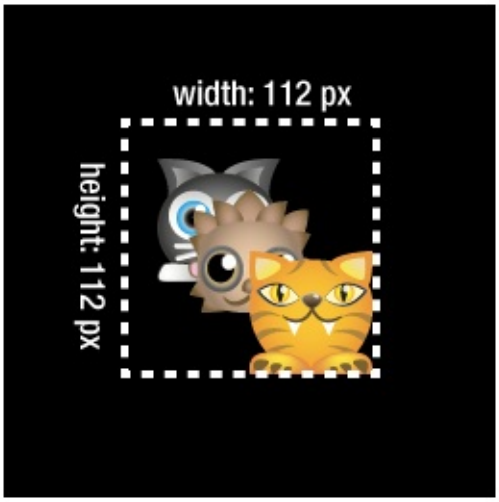
因为 `animals` 分组跟其他精灵一样，你可以改变它的 `x` 和 `y` 的值， `alpha` ， `scale` 和其他精灵的属性。所有你改变了的父容器的属性值，都会改变它的子精灵的相应属性。所以如果你设置分组的 `x` 和 `y` 的位置，所有的子精灵都会相对于分组的左上角重新定位。如果你设置了 `animals` 的 `x` 和 `y` 的位置为64会发生什么呢？

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

整个分组的精灵都会向右和向下移动64像素。

`animals` 分组也有它自己的尺寸，它是以包含的精灵所占的区域计算出来的。你可以像这样来获取 `width` 和 `height` 的值：

```
1. console.log(animals.width);
2. //Displays: 112
3.
4. console.log(animals.height);
5. //Displays: 112
```



如果你改变了分组的宽和高会发生什么呢？

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

所有的孩子精灵都会缩放到刚才你设定的那个值。

如果你喜欢，你可以在一个 `Container` 里嵌套许多其他 `Container`，如果你需要，完全可以创建一个更深的层次。然而，一个 `DisplayObject` （像 `Sprite` 或者其他 `Container`）只能一次属于一个父级。如果你用 `addChild` 让一个精灵成为其他精灵的孩子。Pixi会自动移除它当前的父级，这是一个不用你操心的有用的管理方式。

局部位置和全局位置

当你往一个 `Container` 添加一个精灵时，它的 `x` 和 `y` 的位置是 相对于分组的左上角 的。这是精灵的局部位置，举个例子，你认为这个猫在这张图的哪个位置？

让我们看看：

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

16？是的！这因为猫的只往分组的左上角偏移了16个像素。16是猫的局部位置。

精灵图还有 全局位置 。全局位置是舞台左上角到精灵锚点（通常是精灵的左上角）的距离。你可以通过 `toGlobal` 方法的帮助找到精灵图的全局位置：

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

这意味着你能在 `animals` 分组里找到猫的全局位置：

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

上面给你返回了 `x` 和 `y` 的值为80。这正是猫相对于舞台左上角的相对位置，也就是全局位置。

如果你想知道一个精灵的全局位置，但是不知道精灵的父容器怎么办？每个精灵图有一个属性叫 `parent` 能告诉你精灵的父级是什么。在上面的例子中，猫的父级是 `animals` 。这意味着你可以像如下代码一样得到猫的全局位置：

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

即使你不知道猫的当前父级是谁，上面的代码依然能够正确工作。

这还有一种方式能够计算出全局位置！而且，它实际上最好的方式，所以听好啦！如果你想知道精灵到canvas左上角的距离，但是不知道或者不关心精灵的父亲是谁，用 `getGlobalPosition` 方法。这里展示如何用它来找到老虎的全局位置：

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

它会给你返回 `x` 和 `y` 的值为128。 特别的是， `getGlobalPosition` 是高精度的：当精灵的局部位置改变的同时，它会返回给你精确的全局位置。我曾要求Pixi开发团队添加这个特殊的特性，以便于开发精确的碰撞检测游戏。（谢谢Matt和团队真的把他加上去了！）

如果你想转换全局位置为局部位置怎么办？你可以用 `toLocal` 方法。它的工作方式类似，但是通常是这种通用的格式：

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

用 `toLocal` 找到一个精灵和其他任何一个精灵之间的距离。这段代码告诉你如何获取老虎的相对于猫头鹰的局部位置。

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

上面的代码会返回给你一个32的 `x` 值和一个32的 `y` 值。你可以在例子中看到老虎的左上角和猫头鹰的左上角距离32像素。

使用 ParticleContainer 分组精灵

Pixi有一个额外的，高性能的方式去分组精灵的方法称作： `ParticleContainer` （ `PIXI.ParticleContainer` ）。任何在 `ParticleContainer` 里的精灵都会比在一个普通的 `Container` 的渲染速度快2到5倍。这是用于提升游戏性能的一个很棒的方法。

可以像这样创建 ParticleContainer ：

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

然后用 `addChild` 去往里添加精灵，就像往普通的 `Container` 添加一样。

如果你决定用 `ParticleContainer` 你必须做出一些妥协。在 `ParticleContainer` 里的精灵图只有一小部分基本属性： `x` ， `y` ， `width` ， `height` ， `scale` ， `pivot` ， `alpha` ， `visible` - 就这么多。而且，它包含的精灵不能再继续嵌套自己的孩子精灵。 `ParticleContainer` 也不能用Pixi的先进的视觉效果像过滤器和混合模式。每个 `ParticleContainer` 只能用一个纹理（所以如果你想让精灵有不同的表现方式你将不得不更换雪碧图）。但是为了得到巨大的性能提升，这些妥协通常是值得的。你可以在同一个项目中同时用 `Container` 和 `ParticleContainer` ，然后微调一下你自己的优化。

为什么在 `Particle Container` 的精灵图这么快呢？因为精灵的位置是直接在GPU上计算的。Pixi开发团队正在努力让尽可能多的雪碧图在GPU上处理，所以很有可能你用的最新版的Pixi的 `ParticleContainer` 的特性一定比我现在在这儿描述的特性多得多。查看当前 `ParticleContainer` 文档以获取更多信息。

当你创建一个 `ParticleContainer` ，有四个参数可以传递， `size` ， `properties` ， `batchSize` 和 `autoResize` 。

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

默认的 `maxSize` 是 15,000。所以，如果你需要包裹更多的精灵，把它设置为更高的数字。配置参数是一个拥有五个布尔值的对象： `scale` ， `position` ， `rotation` ， `uvs` 和 `alpha` 。默认的值是 `position` 为 `true` ，其他都为 `false` 。这意味着如果你想在 `ParticleContainer` 改变精灵的 `rotation` ， `scale` ， `alpha` ，或者 `uvs` ，你得先把这些属性设置为 `true` ，像这样：

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

但是，如果你感觉你不需要用这些属性，就保持它们为 `false` 以实现出更好的性能。

`uvs` 是什么呢？只有当它们在动画时需要改变它们纹理子图像的时候你需要设置它为 `true` 。（想让它工作，所有的精灵纹理需要在同一张雪碧图上。）

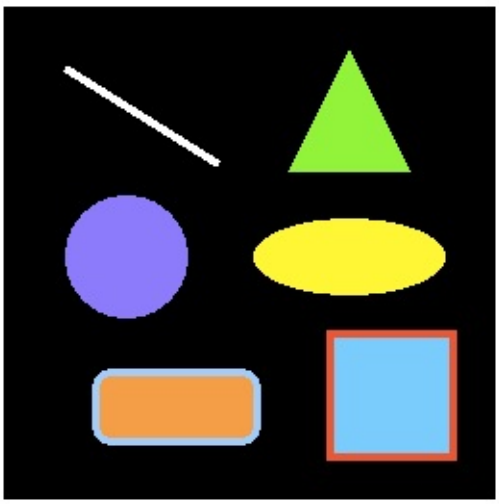
（注意：**UV mapping** 是一个3D图表展示术语，它指纹理（图片）准备映射到三维表面的 `x` 和 `y` 的坐标。 `U` 是 `x` 轴， `V` 是 `y` 轴。WebGL用 `x` , `y` 和 `z` 来进行三维空间定位，所以 `U` 和 `V` 被选为表示2D图片纹理的 `x` 和 `y` 。）

(我真不知道最后两个参数干什么用的，就是 `batchSize` 和 `autoResize` ，如果你知道，就赶紧提个Issue吧！)

用 Pixi 绘制几何图形

用Pixi绘制几何图形

使用图片纹理是制作精灵最有效的方式之一，但是Pixi也提供了自己低级的绘画工具。你可以使用它们来创造矩形、线段、复杂的多边形以及文本。并且它使用和Canvas Drawing API几乎一致的api，所以如果你熟悉canvas的话，那么几乎没有什么新东西需要学习。当然另一个巨大的优势在于，不同于Canvas的绘画api，你使用Pixi绘制的图形是通过WebGL在GPU上渲染的。Pixi能够让你获得所有未触碰到的性能。让我们简单看一下如何创造一些基本图形。下面是我们将要使用前面代码来创造的图形。



矩形

所有的形状的初始化都是先创造一个Pixi的 `Graphics` 的类 (`PIXI.Graphics`)的实例。

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

调用 `beginFill` 和一个16进制的颜色值来设置矩形的填充颜色。下面展示如何设置颜色为淡蓝色。

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

如果你想要给图形设置一个轮廓，使用 `lineStyle` 方法。下面展示如何给矩形设置一个4像素宽 `alpha` 值为1的红色轮廓

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

调用 `drawRect` 方法来画一个矩形。它的四个参数是 `x` , `y` , `width` 和 `height` 。

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

调用 `endFill` 结束绘制。

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

它看起来就像Canvas的绘画api一样！下面是绘制一个矩形涉及到的所有代码，调整它的位置并且把它添加到舞台吧。

```
let rectangle = new Graphics();
rectangle.lineStyle(4, 0xFF3300, 1);
rectangle.beginFill(0x66CCFF);
rectangle.drawRect(0, 0, 64, 64);
```

```
5.  rectangle.endFill();
6.  rectangle.x = 170;
7.  rectangle.y = 170;
8.  app.stage.addChild(rectangle);
```

这些代码可以在（170，170）这个位置创造一个宽高都为64的蓝色的红框矩形。

圆形

调用 `drawCircle` 方法来创造一个圆。它的三个参数是 `x`， `y` 和 `radius`。

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

不同于矩形和精灵，一个圆形的x和y坐标也是它自身的圆点。下面展示如何创造半径32像素的紫色圆形。

```
1.  let circle = new Graphics();
2.  circle.beginFill(0x9966FF);
3.  circle.drawCircle(0, 0, 32);
4.  circle.endFill();
5.  circle.x = 64;
6.  circle.y = 130;
7.  app.stage.addChild(circle);
```

椭圆

`drawEllipse` 是一个卓越的Canvas绘画api，Pixi也能够让你调用 `drawEllipse` 来绘制椭圆。

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

x/y坐标位置决定了椭圆的左上角（想象椭圆被一个不可见的矩形边界

盒包围着-盒的左上角代表了椭圆x/y的锚点位置）。下面是50像素宽20像素高的黄色椭圆。

```
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);
```

圆角矩形

Pixi同样允许你调用 `drawRoundedRect` 方法来创建圆角矩形。这个方法的最后一个参数 `cornerRadius` 是单位为像素的数字，它代表矩形的圆角应该有多圆。

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

下面展示如何创建一个圆角半径为10的圆角矩形。

```
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);
```

线段

想必你已经看过上面定义线段的 `lineStyle` 方法了。你可以调用 `moveTo` 和 `lineTo` 方法来画线段的起点和终点，就和Canvas绘

画api中的一样。下面展示如何绘制一条4像素宽的白色对角线。

```
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` 对象，比如线段，都有 `x` 和 `y` 值，就像精灵一样，所以你可以在绘制完它们之后将他们定位到舞台的任意位置。

多边形

你可以使用 `drawPolygon` 方法来将线段连接起来并且填充颜色来创造复杂图形。 `drawPolygon` 的参数是一个路径数组，数组中的值为决定图形上每个点位置的x/y坐标。

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

graphicsObject.drawPolygon(path);
```

`drawPolygon` 会将上面三个点连接起来创造图形。下面是如何使用 `drawPolygon` 来连接三条线从而创建一个红底蓝边的三角形。我们将三角形绘制在（0，0）的位置上，之后通过调整它的 `x` 和 `y` 属性来移动它在舞台上的位置。

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

```
3.
4.  //Use `drawPolygon` to define the triangle as
5.  //a path array of x/y positions
6.
7.  triangle.drawPolygon([
8.      -32, 64,              //First point
9.      32, 64,               //Second point
10.     0, 0                  //Third point
11. ]);
12.
13. //Fill shape's color
14. triangle.endFill();
15.
16. //Position the triangle after you've drawn it.
17. //The triangle's x/y position is anchored to its first point in the
    path
18. triangle.x = 180;
19. triangle.y = 22;
20.
21. app.stage.addChild(triangle);
```

显示文本

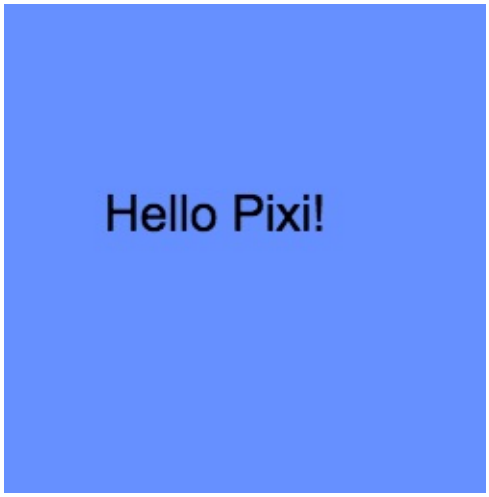
显示文本

使用一个 `Text` 对象 (`PIXI.Text`)在舞台上展示文本。简单来说，你可以这样使用它：

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

这将会在画布上展示文本“Hello, Pixi”。Pixi的文本对象继承自 `Sprite` 类，所以它包含了所有相同的属性，像 `x` , `y` , `width` , `height` , `alpha` , 和 `rotation` 。你可以像处理其他精灵一样在舞台上定位或调整文本。例如，你可以像下面这样使用 `position.set` 来设置 `message` 的 `x` 和 `y` 位置：

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



这样你会得到基础的未加修饰的文本。但是如果你想要更绚丽的文字，

使用Pixi的 `TextStyle` 函数来自定义文字效果。下面展示如何操作：

```
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,
});
```

这将创建一个新的包含所有你想用的样式的 `style` 对象。所有样式属性，see here。

添加 `style` 对象作为 `Text` 函数的第二个参数来应用样式到文本上，就像这样：

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

如果你想要在你创建文本对象之后改变它的内容，使用 `text` 属性。

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

如果你想要重新定义样式属性，使用 `style` 属性。

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

Pixi通过调用Canvas绘画api将文本渲染成不可见或临时的canvas元素来创建文本对象。它之后会将画布转化为WebGL纹理，所以可以被映射到精灵上。这就是为什么文本的颜色需要被包裹成字符串：那是Canvas绘画api的颜色值。与任何canvas颜色值一样，你可以使用“red”或“green”等常用颜色的单词，或使用rgba，hsla或十六进制值。

Pixi也能包裹文本的长段。设置文本的 `wordWrap` 样式属性到 `true`，然后设置 `wordWrapWidth` 到一行文字应该到的最大像素。调用 `align` 属性来设置多行文本的对齐方式。

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

（注意： `align` 不会影响单行文本。）

如果你想要使用自定义的字体文件，使用CSS的 `@font-face` 规则来链接字体文件到Pixi应用运行的HTML页面。

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

添加这个 `@font-face` 语句到你的HTML页面的CSS里面。

Pixi也支持位图字体。你可以使用Pixi的加载器来加载XML位图文

件，就像你加载JSON或图片文件一样。

碰撞检测

碰撞检测

现在你知道了如何制造种类繁多的图形对象，但是你能用他们做什么？一个有趣的事情是利用它制作一个简单的 碰撞检测系统 。你可以用一个叫做： `hitTestRectangle` 的自定义的函数来检测两个矩形精灵是否接触。

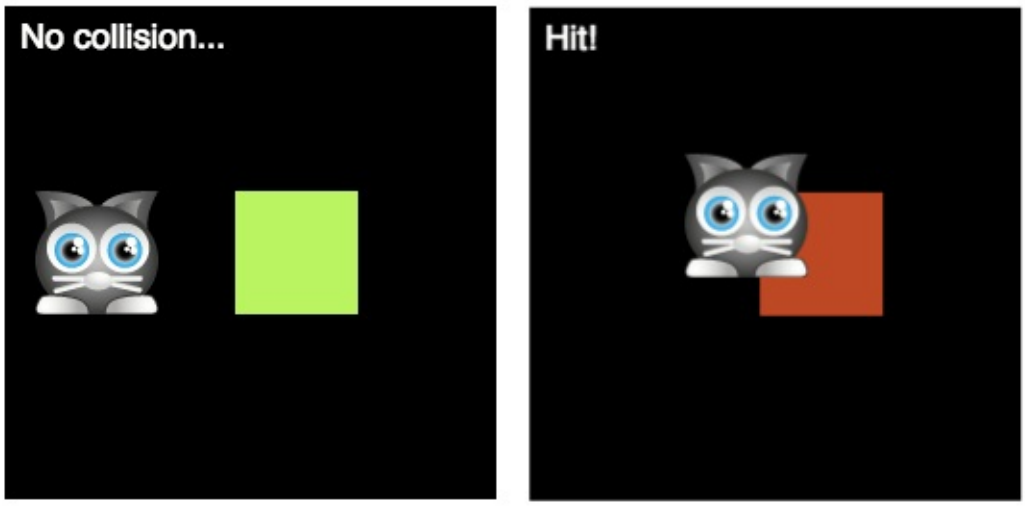
```
1. hitTestRectangle(spriteOne, spriteTwo)
```

如果它们重叠， `hitTestRectangle` 会返回 `true` 。你可以用 `hitTestRectangle` 结合 `if` 条件语句去检测两个精灵是否碰撞：

```
1. if (hitTestRectangle(cat, box)) {
2.   //There's a collision
3. } else {
4.   //There's no collision
5. }
```

正如你所见， `hitTestRectangle` 是走入游戏设计这片宇宙的大门。

运行在 `examples` 文件夹的 `collisionDetection.html` 文件，看看怎么用 `hitTestRectangle` 工作。用方向按键去移动猫，如果猫碰到了盒子，盒子会变成红色，然后 “Hit!” 文字对象会显示出来。



你已经看到了创建这些所有元素的代码，让猫移动的键盘控制。唯一的新的东西就是 `hitTestRectangle` 函数被用在 `play` 函数里检测碰撞。

```
1.  function play(delta) {
2.
3.    //use the cat's velocity to make it move
4.    cat.x += cat.vx;
5.    cat.y += cat.vy;
6.
7.    //check for a collision between the cat and the box
8.    if (hitTestRectangle(cat, box)) {
9.
10.     //if there's a collision, change the message text
11.     //and tint the box red
12.     message.text = "hit!";
13.     box.tint = 0xff3300;
14.
15.   } else {
16.
17.     //if there's no collision, reset the message
18.     //text and the box's color
19.     message.text = "No collision...";
20.     box.tint = 0xccff99;
```

```
21.   }
22. }
```

`play` 函数被每秒调用了60次，每一次这个 `if` 条件语句都会在猫和盒子之间进行碰撞检测。如果 `hitTestRectangle` 为 `true`，那么文字 `message` 对象会用 `setText` 方法去显示 “Hit”:

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

这个盒子的颜色改变的效果是把盒子的 `tint` 属性改成一个16进制的红色的值实现的。

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

如果没有碰撞，消息和盒子会保持它们的原始状态。

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

代码很简单，但是你已经创造了一个看起来完全活着的互动的世界！它简直跟魔术一样！令人惊讶的是，你大概已经拥有了你需要用Pixi制作游戏的全部技能！

碰撞检测函数

`hitTestRectangle` 函数都有些什么呢？它做了什么，还有它是如何工作的？关于碰撞检测算法的细节有些超出了本教程的范围。最重要的事情是你要知道如何使用它。但是，只是作为你的参考资料，不让你好奇，这里有全部的 `hitTestRectangle` 函数的定义。你能从注释弄明白它都做了什么吗？

```
1. function hitTestRectangle(r1, r2) {
2.
```

```
  //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 occuring. 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
```

```
40.         hit = false;
41.       }
42.     } else {
43.
44.       //There's no collision on the x axis
45.       hit = false;
46.     }
47.
48.     //`hit` will be either `true` or `false`
49.     return hit;
50.   };
```

实例学习: 宝物猎人

实例学习： 宝物猎人

我要告诉你你现在已经拥有了全部的技能去开始制作一款游戏。什么？你不相信我？让我为你证明它！让我们来做一个简单的对象收集和躲避的敌人的游戏叫：宝藏猎人。（你能在 `examples` 文件夹中找到它。）

宝藏猎手是一个简单的完整的游戏的例子，它能让你把目前所学的所有工具都用上。用键盘的方向键可以帮助探险者找到宝藏并带它出去。六只怪物在地牢的地板上上下移动，如果它们碰到了探险者，探险者变为半透明，而且他右上角的血条会减少。如果所有的血都用完了，"You Lost!"会出现在舞台上；如果探险者带着宝藏到达了出口，显示“You Won!”。尽管它是一个基础的原型，宝藏猎手包含了很多大型游戏里很大一部分元素：纹理贴图集，人机交互，碰撞以及多个游戏场景。让我们一起去看看游戏是如何被它们组合起来的，以便于你可以用它作你自己开发的游戏的起点。

代码结构

打开 `treasureHunter.html` 文件，你将会看到所有的代码都在一个大的文件里。下面是一个关于如何组织所有代码的概览：

```
1. //Setup Pixi and load the texture atlas files - call the `setup`
2. //function when they've loaded
3.
4. function setup() {
5.   //Initialize the game sprites, set the game `state` to `play`
6.   //and start the 'gameLoop'
7. }
```

```
8.
9.  function gameLoop(delta) {
10.   //Runs the current game `state` in a loop and renders the sprites
11. }
12.
13. function play(delta) {
14.   //All the game logic goes here
15. }
16.
17. function end() {
18.   //All the code that should run at the end of the game
19. }
20.
21. //The game's helper functions:
22. //`keyboard`, `hitTestRectangle`, `contain` and `randomInt`
```

把这个当作你游戏代码的蓝图，让我们看看每一部分是如何工作的。

用 setup 函数初始化游戏

一旦纹理图集图片被加载进来了， `setup` 函数就会执行。它只会执行一次，可以让你为游戏执行一次安装任务。这是一个用来创建和初始化对象、精灵、游戏场景、填充数据数组或解析加载JSON游戏数据的好地方。

这是宝藏猎手 `setup` 函数的缩略图和它要执行的任务。

```
1. function setup() {
2.   //Create the `gameScene` group
3.   //Create the `door` sprite
4.   //Create the `player` sprite
5.   //Create the `treasure` sprite
6.   //Make the enemies
7.   //Create the health bar
8.   //Add some text for the game over message
9.   //Create a `gameOverScene` group
```

```
10.   //Assign the player's keyboard controllers
11.
12.   //set the game state to `play`
13.   state = play;
14.
15.   //Start the game loop
16.   app.ticker.add(delta => gameLoop(delta));
17. }
```

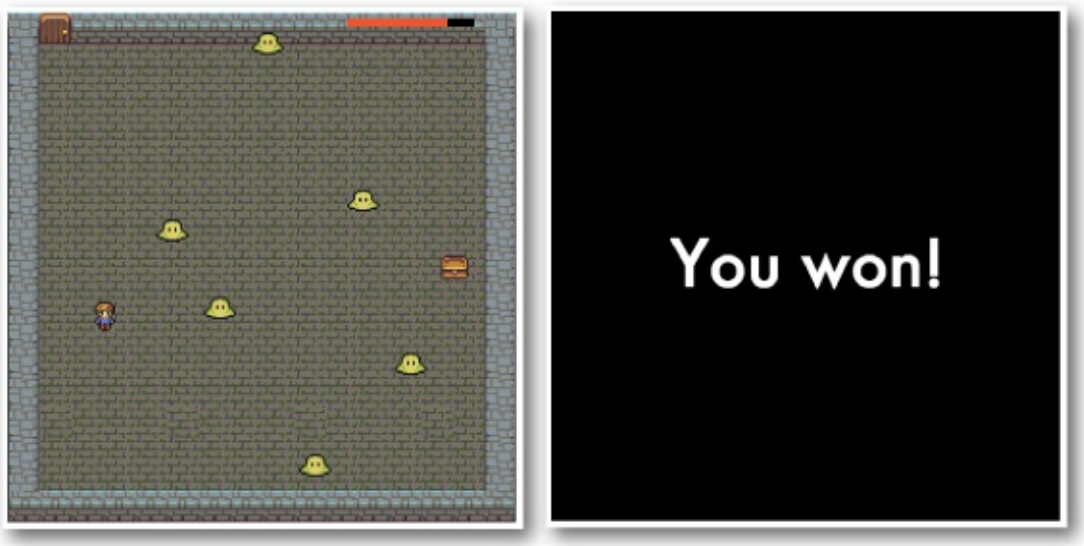
最后两行代码，`state = play;` 和 `gameLoop()` 可能是最重要的。运行 `gameLoop` 切换了游戏的引擎，而且引发了 `play` 一直被循环调用。但是在我们看它如何工作之前，让我们看看 `setup` 函数里的代码都做了什么。

创建游戏场景

`setup` 函数创建了两个被称为 `gameScene` 和 `gameOverScene` 的 `Container` 分组。他们都被添加到了舞台上。

```
1. gameScene = new Container();
2. app.stage.addChild(gameScene);
3.
4. gameOverScene = new Container();
5. app.stage.addChild(gameOverScene);
```

所有的的游戏主要部分的精灵都被添加到了 `gameScene` 分组。游戏结束的文字在游戏结束后显示，应当被添加到 `gameOverScene` 分组。



尽管它是在 `setup` 函数中添加的，但是 `gameOverScene` 不应在游戏一开始的时候显示，所以它的 `visible` 属性被初始化为 `false` 。

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

你会在后面看到，为了在游戏结束之后显示文字，当游戏结束 `gameOverScene` 的 `visible` 属性会被设置为 `true` 。

制作地牢，门，猎人和宝藏

玩家、出口、宝箱和地牢背景图都是从纹理图集制作而来的精灵。有一点很重要的是，他们都是被当做 `gameScene` 的孩子添加进来的。

```
1. //Create an alias for the texture atlas frame ids
2. id = resources["images/treasureHunter.json"].textures;
3.
4. //Dungeon
5. dungeon = new Sprite(id["dungeon.png"]);
6. gameScene.addChild(dungeon);
```

```
7.
8.  //Door
9.  door = new Sprite(id["door.png"]);
10. door.position.set(32, 0);
11. gameScene.addChild(door);
12.
13. //Explorer
14. explorer = new Sprite(id["explorer.png"]);
15. explorer.x = 68;
16. explorer.y = gameScene.height / 2 - explorer.height / 2;
17. explorer.vx = 0;
18. explorer.vy = 0;
19. gameScene.addChild(explorer);
20.
21. //Treasure
22. treasure = new Sprite(id["treasure.png"]);
23. treasure.x = gameScene.width - treasure.width - 48;
24. treasure.y = gameScene.height / 2 - treasure.height / 2;
25. gameScene.addChild(treasure);
```

把它们都放在 `gameScene` 分组会使我们在游戏结束的时候去隐藏 `gameScene` 和显示 `gameOverScene` 操作起来更简单。

制造泡泡怪们

六个泡泡怪是被循环创建的。每一个泡泡怪都被赋予了一个随机的初始位置和速度。每个泡泡怪的垂直速度都被交替的乘以 `1` 或者 `-1`，这就是每个怪物和相邻的下一个怪物运动的方向都是相反的原因，每个被创建的怪物都被放进了一个名为 `blobs` 的数组。

```
1.  let numberOfBlobs = 6,
2.      spacing = 48,
3.      xOffset = 150,
4.      speed = 2,
5.      direction = 1;
6.
```

```
7.  //An array to store all the blob monsters
8.  blobs = [];
9.
10. //Make as many blobs as there are `numberOfBlobs`
11. for (let i = 0; i < numberOfBlobs; i++) {
12.
13.   //Make a blob
14.   let blob = new Sprite(id["blob.png"]);
15.
16.   //Space each blob horizontally according to the `spacing` value.
17.   //`xOffset` determines the point from the left of the screen
18.   //at which the first blob should be added
19.   let x = spacing * i + xOffset;
20.
21.   //Give the blob a random `y` position
22.   let y = randomInt(0, stage.height - blob.height);
23.
24.   //Set the blob's position
25.   blob.x = x;
26.   blob.y = y;
27.
28.   //Set the blob's vertical velocity. `direction` will be either
    `1` or
29.   //`-1`. `1` means the enemy will move down and `-1` means the
    blob will
30.   //move up. Multiplying `direction` by `speed` determines the
    blob's
31.   //vertical direction
32.   blob.vy = speed * direction;
33.
34.   //Reverse the direction for the next blob
35.   direction *= -1;
36.
37.   //Push the blob into the `blobs` array
38.   blobs.push(blob);
39.
40.   //Add the blob to the `gameScene`
41.   gameScene.addChild(blob);
```

```
42. }
```

制作血条

当你玩儿宝藏猎人的时候，你会发现当猎人碰到其中一个敌人时，场景右上角的血条宽度会减少。这个血条是如何被制作的？他就是两个相同的位置的重叠的矩形：一个黑色的矩形在下面，红色的上面。他们被分组到了一个单独的 `healthBar` 分组。 `healthBar` 然后被添加到 `gameScene` 并在舞台上被定位。

```
1.  //Create the health bar
2.  healthBar = new PIXI.DisplayObjectContainer();
3.  healthBar.position.set(stage.width - 170, 4)
4.  gameScene.addChild(healthBar);
5.
6.  //Create the black background rectangle
7.  let innerBar = new PIXI.Graphics();
8.  innerBar.beginFill(0x000000);
9.  innerBar.drawRect(0, 0, 128, 8);
10. innerBar.endFill();
11. healthBar.addChild(innerBar);
12.
13. //Create the front red rectangle
14. let outerBar = new PIXI.Graphics();
15. outerBar.beginFill(0xFF3300);
16. outerBar.drawRect(0, 0, 128, 8);
17. outerBar.endFill();
18. healthBar.addChild(outerBar);
19.
20. healthBar.outer = outerBar;
```

你会看到 `healthBar` 添加了一个名为 `outer` 的属性。它仅仅是引用了 `outerBar` （红色的矩形）以便于过会儿能够被很方便的获取。

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

你可以不这么做，但是为什么不呢？这意味如果你想控制红色 `outerBar` 的宽度，你可以像这样顺畅的写如下代码：

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

这样的代码相当整齐而且可读性强，所以我们会一直保留它！

制作消息文字

当游戏结束的时候，“You won!”或者“You lost!”的文字会显示出来。这使用文字纹理制作的，并添加到了 `gameOverScene`。因为 `gameOverScene` 的 `visible` 属性设为了 `false`，当游戏开始的时候，你看不到这些文字。这段代码来自 `setup` 函数，它创建了消息文字，而且被添加到了 `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);
```

开始游戏

所有的让精灵移动的游戏逻辑代码都在 `play` 函数里，这是一个被循环执行的函数。这里是 `play` 函数都做了什么的总体概览：

```
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
```

```
5.    //Check for a collision between the explorer and the treasure
6.    //Check for a collision between the treasure and the door
7.    //Decide whether the game has been won or lost
8.    //Change the game `state` to `end` when the game is finsihed
9.  }
```

让我们弄清楚这些特性都是怎么工作的吧。

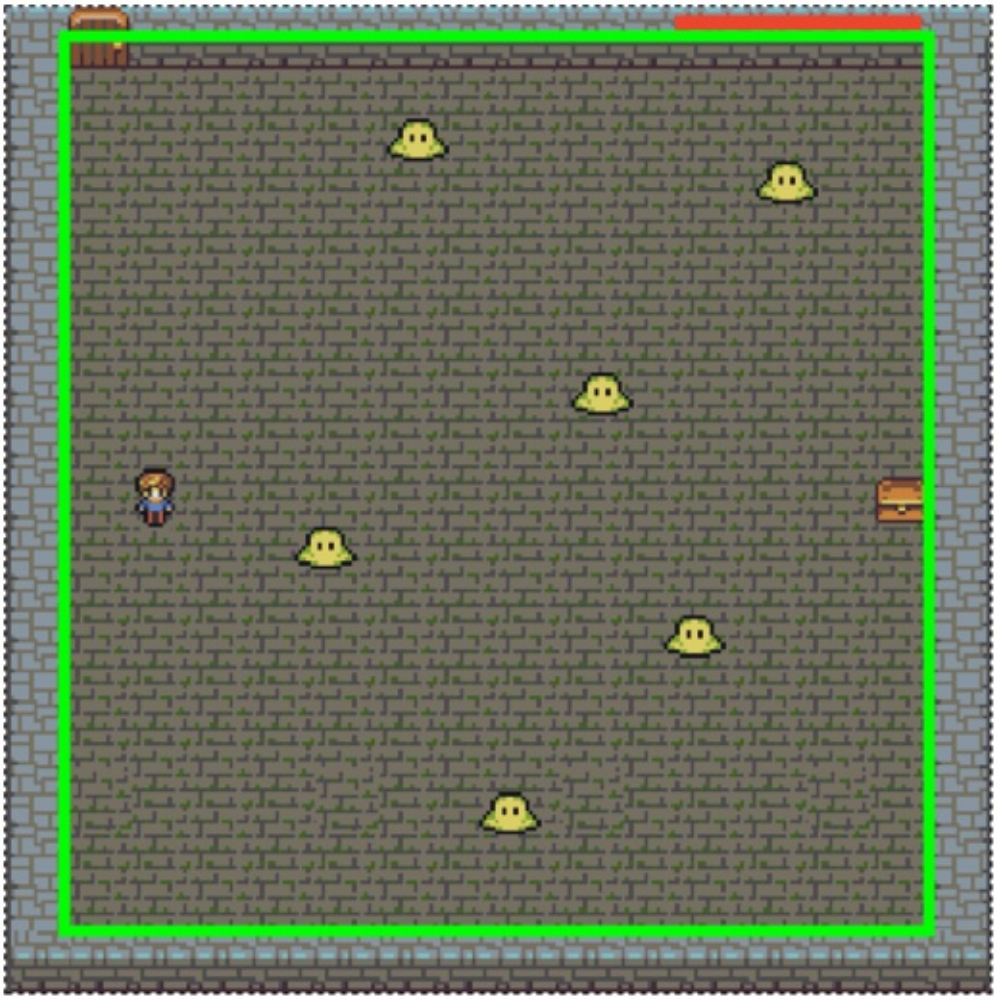
移动探险者

探险者是被键盘控制的，实现它的代码跟你在之前学习的键盘控制代码很相似。在 `play` 函数里， `keyboard` 对象修改探险者的速度，这个速度和探险者的位置相加。

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

控制运动的范围

一个新的地方的是，探险者的运动是被包裹在地牢的墙体之内的。绿色的轮廓表明了探险者运动的边界。



通过一个名为 `contain` 的自定义函数可以帮助实现。

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

`contain` 接收两个参数。第一个是你想控制的精灵。第二个是包含了 `x` , `y` , `width` 和 `height` 属性的任何一个对象。在这个例子中，控制对象定义了一个区域，它稍微比舞台小了一点，和地牢的尺寸一样。

这里是实现了上述功能的 `contain` 函数。函数检查了精灵是否跨越了控制对象的边界。如果超出，代码会把精灵继续放在那个边界上。 `contain` 函数也返回了一个值可能为"top", "right", "bottom" 或者 "left" 的 `collision` 变量，取决于精灵碰到了哪一个边界。

（如果精灵没有碰到任何边界，`collision` 将返回 `undefined` 。）

```
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;
}
```

你会在接下来看到 `collision` 的返回值在代码里是如何让怪物在地牢的顶部和底部之间来回反弹的。

移动怪物

`play` 函数也能够移动怪物，保持它们在地牢的墙体之内，并检测每个怪物是否和玩家发生了碰撞。如果一只怪物撞到了地牢的顶部或者底部的墙，它就会被设置为反向运动。完成所有这些功能都是通过一个 `forEach` 循环，它每一帧都会遍历在 `blobs` 数组里的每一个怪物。

```
1.  blobs.forEach(function(blob) {
2.
3.    //Move the blob
4.    blob.y += blob.vy;
5.
6.    //Check the blob's screen boundaries
7.    let blobHitsWall = contain(blob, {x: 28, y: 10, width: 488,
    height: 480});
8.
9.    //If the blob hits the top or bottom of the stage, reverse
10.   //its direction
11.   if (blobHitsWall === "top" || blobHitsWall === "bottom") {
12.     blob.vy *= -1;
13.   }
14.
15.   //Test for a collision. If any of the enemies are touching
16.   //the explorer, set `explorerHit` to `true`
17.   if(hitTestRectangle(explorer, blob)) {
18.     explorerHit = true;
19.   }
20. });
```

你可以在上面这段代码中看到， `contain` 函数的返回值是如何被用来让怪物在墙体之间来回反弹的。一个名为 `blobHitsWall` 的变量被用来捕获返回值：

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

`blobHitsWall` 通常应该是 `undefined`。但是如果怪物碰到了顶部的墙，`blobHitsWall` 将会变成 “top”。如果碰到了底部的墙，`blobHitsWall` 会变为 “bottom”。如果它们其中任何一种情况为 true，你就可以通过给怪物的速度取反来让它反向运动。这是实现它的代码：

```
1. if (blobHitsWall === "top" || blobHitsWall === "bottom") {
2.   blob.vy *= -1;
3. }
```

把怪物的 `vy` （垂直速度）乘以 `-1` 就会反转它的运动方向。

检测碰撞

在上面的循环代码里用了 `hitTestRectangle` 来指明是否有敌人碰到了猎人。

```
1. if(hitTestRectangle(explorer, blob)) {
2.   explorerHit = true;
3. }
```

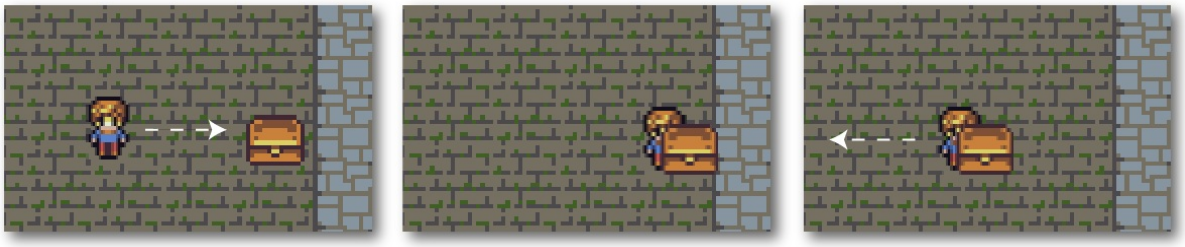
如果 `hitTestRectangle` 返回 `true`，意味着发生了一次碰撞，名为 `explorerHit` 的变量被设置为了 `true`。如果 `explorerHit` 为 `true`， play 函数让猎人变为半透明，然后把 `health` 条减少1像素的宽度。

```
1. if(explorerHit) {
2.
3.   //Make the explorer semi-transparent
4.   explorer.alpha = 0.5;
5.
6.   //Reduce the width of the health bar's inner rectangle by 1 pixel
7.   healthBar.outer.width -= 1;
```

```
8.
9.  } else {
10.
11.    //Make the explorer fully opaque (non-transparent) if it hasn't
    been hit
12.    explorer.alpha = 1;
13.  }
```

如果 `explorerHit` 是 `false`，猎人的 `alpha` 属性将保持1，完全不透明。

`play` 函数也要检测宝箱和探险者之间的碰撞。如果发生了一次撞击， `treasure` 会被设置为探险者的位置，在做一点偏移。看起来像是猎人携带着宝藏一样。



这段代码实现了上述效果：

```
1.  if (hitTestRectangle(explorer, treasure)) {
2.    treasure.x = explorer.x + 8;
3.    treasure.y = explorer.y + 8;
4.  }
```

处理到达出口和结束游戏

游戏结束有两种方式：如果你携带宝藏到达出口你将赢得游戏，或者你的血用完你就死了。

想要获胜，宝箱只需碰到出口就行了。如果碰到了出口，游戏的

`state` 会被设置为 `end`， `message` 文字会显示 “You won！”。

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

如果你的血用完，你将输掉游戏。游戏的 `state` 也会被设置为 `end`， `message` 文字会显示 “You Lost！”。

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

但是这是什么意思呢？

```
state = end;
```

你会在早些的例子看到 `gameLoop` 在持续的每秒60次的更新 `state` 函数。 `gameLoop` 的代码如下：

```
function gameLoop(delta){

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

你也会记住我们给 `state` 设定的初始值为 `play`，这也就是为什么 `play` 函数会循环执行。通过设置 `state` 为 `end` 我们告诉代码我们想循环执行另一个名为 `end` 的函数。在大一点的游戏你可能会为每一个游戏等级设置 `tileScene` 状态和状态集，像 `leveOne`，

`levelTwo` 和 `levelThree`。

`end` 函数是什么？就是它！

```
1. function end() {
2.   gameScene.visible = false;
3.   gameOverScene.visible = true;
4. }
```

它仅仅是反转了游戏场景的显示。这就是当游戏结束的时候隐藏 `gameScene` 和显示 `gameOverScene` 。

这是一个如何更换游戏状态的一个很简单的例子，但是你可以想在你的游戏里添加多少状态就添加多少状态，然后给它们添加你需要的代码。然后改变 `state` 为任何你想循环的函数。

这就是完成宝藏猎人所需要的一切了。然后再通过更多一点的工作就能把这个简单的原型变成一个完整的游戏 - 快去试试吧！

一些关于精灵的其他知识

一些关于精灵的其他知识

目前为止你已经学会了如何用相当多有用的精灵的属性，像 `x` ， `y` ， `visible` ， 和 `rotation` ，它们让你能够让你很大程度上控制精灵的位置和外观。但是Pixi精灵也有其他很多有用的属性可以使用。 这是一个完整的列表

Pixi的类继承体系是怎么工作的呢？（什么是 类 ， 什么是 继承？点击这个链接了解。）Pixi的精灵遵循以下原型链构建了一个继承模型：

```
1. DisplayObject > Container > Sprite
```

继承意味着在继承链后面的类可以用之前的类的属性和方法。最基础的类是 `DisplayObject` 。任何只要是 `DisplayObject` 都可以被渲染在舞台上。 `Container` 是继承链的下一个类。它允许 `DisplayObject` 作为其他 `DisplayObject` 的容器。继承链的第三个类是 `Sprite` 。这个类被你用来创建你游戏的大部分对象。然而，不久前你就会学习了如何用 `Container` 去给精灵分组。

展望未来

展望未来

Pixi能做很多事情，但是不能做全部的事情！如果你想用Pixi开始制作游戏或者复杂的交互型应用，你可能会需要一些有用的库：

- Bump: 一个为了游戏准备的完整的2D碰撞函数集.
- Tink: 拖放, 按钮, 一个通用的指针和其他有用的交互工具集。
- Charm: 给Pixi精灵准备的简单易用的缓动动画效果。
- Dust: 创建像爆炸，火焰和魔法等粒子效果。
- Sprite Utilities: 创建和使用Pixi精灵的一个更容易和更直观的做法，包括添加状态机和动画播放器。让Pixi的工作变得更有趣。
- Sound.js: 一个加载，控制和生成声音和音乐效果的微型库。包含了一切你需要添加到游戏的声音。
- Smoothie: 使用真正的时间增量插值实现的超平滑精灵动画。它也允许为你的运行的游戏和应用指定 fps (帧率) ，并且把你的精灵图循环渲染完全从你的应用逻辑循环中分离出去。

你可以在这儿在这本书里找到如何用结合Pixi使用这些库。
学习PixiJS.

Hexi

如果你想使用全部的这些功能库，但又不想给自己整一堆麻烦？用

Hexi: 创建游戏和交互应用的完整开发环境:

https://github.com/kittykatattack/hexi

它把最新版本的Pixi（最新的 稳定 的一个）和这些库（还有更多！）打包在了一起，为了可以通过一种简单而且有趣的方式去创建游戏。Hexi 也允许你直接获取 `PIXI` 对象，所以你可直接写底层的Pixi代码，然后任意的选择你需要的Hexi额外的方便的功能。

BabylonJS

Pixi可以很好地完成2D交互式媒体，但是对于3D却无能为力。当你准备踏进3D领域，这个最有潜力的领域的时候，不妨使用这个为WEB游戏开发者准备的用起来非常简单的3D库：BabylonJS。它是提升你技能的下一步。

支持这个工程

支持这个工程

买这本书吧！不敢相信，有人居然会付钱让我完成这个教程并把它写成一本书！

学习 PixiJS

（它可不是一本毫无价值的“电子书”，而是一本真实的，很厚的纸质书！由世界上最大的出版商，施普林格出版！这意味着你可以邀请你的朋友过来，放火，烤棉花糖！！）它比本教程多出了80%的内容，它包含了所有如何用Pixi制作所有交互应用和游戏的必要技术。）

你可以在里面学到：

- 制作动画游戏角色。
- 创建一个全功能动画状态播放器。
- 动态的动画线条和形状。
- 用平铺的精灵实现无限滚动视差。
- 使用混合模式，滤镜，调色，遮罩，视频，和渲染纹理。
- 为多种分辨率生成内容。
- 创建交互按钮。
- 为Pixi创建一个灵活的拖动和拖放界面。
- 创建粒子效果。
- 建立一个可以展到任何大小的稳定的软件架构模型。
- 制作一个完整的游戏。

不仅如此，作为一个福利，所有的代码完全使用最新版本的 JavaScript: ES6/2015 编写而成的。尽管这本书基于Pixi

V3.x，但是它仍旧能在Pixi 4.x上很好的运行！