

学习Pixi

learningPixi中文版

一步一步的告诉你如何通过[Pixi渲染引擎](#) 制作游戏和交互媒体。 **更新至 Pixi v4.0.0**. 如果你喜欢本教程 你肯定会喜欢这本书，它包含了比本教程多出80%的内容！

目录表：

1. [介绍](#)
2. [建立环境](#)
 1. [安装Pixi的简单方法](#)
 2. [通过Git安装Pixi](#)
 3. [通过Node和Gulp安装Pixi](#)
3. [创建舞台的渲染器](#)
4. [Pixi 精灵](#)
5. [把图片加载到纹理缓存里](#)
6. [展示精灵](#)
 1. [通过别名](#)
 2. [进一步了解『加载』](#)
 1. [从普通JavaScript图像对象或画布中创建一个精灵](#)
 2. [为已加载的文件指定一个名字](#)
 3. [监控加载进度](#)
 4. [进一步了解『Pixi的加载器』](#)
7. [定位精灵](#)
8. [尺寸和缩放](#)
9. [旋转](#)
10. [从背景子图像集中制作精灵](#)
11. [使用纹理图集](#)
12. [加载纹理图集](#)
13. [从一个已加载的纹理地图集中制作精灵](#)
14. [移动精灵](#)
15. [使用速度属性](#)
16. [游戏状态](#)
17. [键盘动作](#)
18. [分组的精灵](#)
 1. [局部和全局位置](#)
 2. [给分组精灵用 ParticleContainer](#)
19. [Pixi的图元](#)
 1. [矩形](#)
 2. [圆形](#)
 3. [椭圆](#)
 4. [圆角矩形](#)
 5. [线条](#)
 6. [多边形](#)
20. [展示文字](#)

21. [碰撞检测](#)
 1. [hitTestRectangle 函数](#)
22. [实战学习: 宝藏猎手](#)
 1. [利用 setup 函数初始化游戏](#)
 1. [创建游戏场景](#)
 2. [制作地牢, 门, 探险者 和 宝藏](#)
 3. [制作一堆怪物](#)
 4. [制作血条](#)
 5. [制作消息文字](#)
 2. [操作游戏](#)
 3. [移动探险者](#)
 1. [控制移动范围](#)
 4. [移动怪物](#)
 5. [碰撞检查](#)
 6. [到达出口并结束游戏](#)
23. [进一步了解『精灵』](#)
24. [更进一步学习](#)
 1. [Hexi](#)
25. [支持该项目](#)

介绍

PixiJs是一个速度极快的2D精灵图渲染引擎。这意味着什么？意味着它能帮你展示，驱动和管理富有交互性的图形以便于制作游戏和通过使用JavaScript以及其他HTML5技术而创建的一系列应用。它有一套合理的、整齐的API并且提供了许多有用的特性，像支持纹理地图集并且还提供了动画精灵图（交互式图像）的简化流程。它也给你了一个完整的场景图以便于你能够创建嵌套的精灵图（一个精灵图在另一个精灵图里面），与此同时让你可以直接在精灵图上绑定鼠标和触摸事件。最重要的是，Pixi让你能够按照自己所想和所需，适应自己的编码风格，和其他有用的框架完美的结合。

Pixi的API是由Macromedia / Adobe Flash开创一个经过磨练和测试的API的改良版。原Flash开发者使用起来会得心应手。目前其他的一些精灵渲染框架用了相似的API：CreateJS、Starling、Sparrow和苹果的SpriteKit。Pixi的API的优点在于它致力于通用性：它不是一个游戏引擎。这一点很好，因为它给你了一个完全的自由让你做任何想做的东西，把你自己的游戏引擎和它揉和在一起。

在本教程中，你将弄清楚如何结合Pixi的强大的图像渲染引擎以及场景图去制作游戏。你也会学习如何通过一个纹理背景图去准备你的游戏图像，如何通过质子粒子引擎去制作粒子效果，以及如何把Pixi整合到自己的游戏引擎中。但是Pixi并不只是为游戏而生的 - 你可以用这些相同的技术去创造任何可交互的多媒体应用。这意味着手机应用！

在开始学习本教程之前你需要了解什么？

你应该对HTML以及JavaScript有一个适当的理解。你不需要成为一个专家，只需要做一个渴望学习的富有雄心的初学者。如果你不了解HTML和JavaScript，最好的开始学习的地方是这本书：

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

我能确信的一点的是它是一本最好的书，因为是我写的！（手动斜眼笑- -）

这里也有一些非常好的互联网资源能帮助你学习：

[Khan Academy: Computer Programming](#)

[Code Academy: JavaScript](#)

选择最适合你学习方式的就行了。

好了，明白了吗？你了解了什么是JavaScript变量、函数、数组以及对象以及如何使用它们吗？你明白了什么是JSON数据格式文件了吗？你是否使用过Canvas绘图 API呢？

为了使用Pixi，你也需要在你项目的根目录下开启一个web服务器。你知道什么是web服务器以及如何在你的项目文件夹里启动吗？最好的方式是使用[node.js](#)，然后十分简单的安装一个[http-server](#)。然而，如果你想那么做，你需要在Unix的命令行中舒适的工作。你可以在[这个视频里](#)学习如何使用Unix，当你完成时候，接着学习[这个视频](#)。你应当学习如何使用Unix - 它只会耗费你几个小时的时间去学习而且它是一个和电脑交互的非常简单有趣的方式。

但是如果你不想搅入刚才提到的命令行的乱麻中，试试Mongoose服务器：

[Mongoose](#)

或者，只需要用出色的[Brackets text editor](#)去编写你的所有代码。当你在Brackets的主要工作区点击那个闪电按钮，它会自动为你启动一个web服务器和浏览器。

现在如果你觉得你准备好了，那就继续往下读吧！

（读者要求：这是一个 *在线文档*。如果你对某个指定的细节有任何问题或者需要任何相关内容的解释，请在这个GitHub仓库中创建一个 **issue**，我会更新更多的信息。）

建立环境

在你开始写代码之前，为你的项目创建一个文件夹，然后在项目的根目录下启动一个web服务器。如果你不开启一个web服务器的话，Pixi将不会工作。

接下来，你需要安装Pixi。这里有两种方式：**简单**的方法，用 **Git** 或者用 **Gulp** 和 **Node**。

安装Pixi的简单方法

本介绍用的Pixi版本是 **v4.0.0**，你可以在[Pixi's release page for v4.0.0](#)找到[pixi.min.js](#)文件。或者你可以从[Pixi's main release page](#)获取最新的版本。

这个文件就是使用Pixi的全部所需。你可以忽略此仓库中的其他所有文件：**你不需要它们**。

接下来，创建一个基本的HTML页面，然后用<script> 标签去链接刚才你下载的[pixi.min.js](#) 文件，这个<script> 标签的 **src**应当是你根目录的相对路径。你的<script> 标签应当看起来像这样

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

这是一个完整的HTML页面，你可以用它来加载Pixi然后测试一下它是可以工作的：

```
<!doctype html>
<html>
```

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

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

如果Pixi载入成功，下面的信息将会在你的web浏览器的JavaScript控制台中默认显示出来：

```
Pixi.js 4.0.0 - ☆ WebGL ☆      http://www.pixijs.com/      ♥♥♥
```

现在你可以通过Pixi来工作啦！

通过Git安装Pixi

你也可以通过Git安装使用Pixi。（啥是 **git**？如果你不知道，[你可以在这儿弄清楚](#)。）用Git安装有一些优势就是：你可以只通过在命令行运行 `git pull origin master` 去更新Pixi到最新的版本。并且，如果你觉得你在Pixi中发现了一个bug，你可以修复它并在主仓库中提交一个修复bug的pull请求。

为了通过Git克隆Pixi仓库，`cd`进入你的项目根目录并键入：

```
git clone git@github.com:pixijs/pixi.js.git
```

它会自动创建一个名为**pixi.js**的文件夹而且会加载**最新版本**的Pixi。请时刻记住，本手册是围绕 *version 4.0.0* 展开的。通过简单的切换分支的方式：

```
git checkout tags/v4.0.0
```

可以获取4.0版本。

Pixi安装之后，创建一个基本的HTML文档，然后用`<script>`标签从Pixi的**bin**文件夹里加载**pixi.js**文件：

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

(如果你喜欢, 你可以用 `pixi.min.js` 文件去替换上一节我建议的文件。这个压缩过的文件实际上运行的稍微快一点, 而且它当然也会加载的更快。用原生的未压缩过的文件的优势在于如果编译器发现Pixi的源代码里有bug, 它会用一个可读的格式, 显示出有问题代码的错误信息。这对你开发一个项目很有用, 因为如果bug不是在Pixi里面, 那么这个错误可能按时你的代码有问题。)

在这个 **Learning Pixi** 仓库里 (你正在读!) 你会发现一个文件夹名为 `examples`。打开它, 你会找到一个文件叫 `helloWorld.html`。假设服务器实在此仓库的根目录下运行的, 这就是 `helloWorld.html` 文件如何正确的加载Pixi:

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>Hello World</title>
</head>
<body>
  <script src="../pixi.js/bin/pixi.js"></script>
</body>
</html>
```

如果Pixi加载正确, 下面的东西会默认出现在你的web浏览器的JavaScript控制台里:

```
Pixi.js 4.0.0 - ☆ WebGL ☆      http://www.pixijs.com/      ♥♥♥
```

通过Nod和Gulp安装Pixi

你也可以通过 [Node](#) 和 [Gulp](#)来安装Pixi。如果你需要对Pixi进行一次常规的构建来引入或者排除某些特性, 你应该选择这条路线。[看看Pixi的GitHub仓库了解更多细节](#)。但是, 通常没有必要这么做。

创建渲染器和舞台

现在可以开始使用Pixi了!

但是咋使用呢?

第一步就是创建一个矩形显示区域用来展示图片。可以用Pixi的 `renderer` 对象可以创建。它能自动生成一个HTML `<canvas>` 节点并且能指定如何在Canvas上展示你的图像。然后你需要创建一个特殊的Pixi `Container` 对象名为 `stage`。正如你将看到, 这个舞台对象会被用作持有所有你想让Pixi展示的东西的根容器。


你需要写如下代码去创建一个 `renderer` 和 `stage`。把这段代码通过 `<script>` 便签括起来并添加到你的HTML文档中:

```
//Create the renderer
var renderer = PIXI.autoDetectRenderer(256, 256);

//Add the canvas to the HTML document
document.body.appendChild(renderer.view);
```

```
//Create a container object called the `stage`  
var stage = new PIXI.Container();  
  
//Tell the `renderer` to `render` the `stage`  
renderer.render(stage);
```

这是你开始使用Pixi之前所需要写的最基本的代码。它会生成一个黑色的256*256大小的Canvas节点并且把它添加到你的HTML文档中。当你运行这段代码的时候，会有如下效果：

基本展示

恩，一个黑色的正方形！

Pixi的`autoDetectRenderer`方法能计算出是用Canvas的绘图API还是WebGL去渲染图像，这取决于哪一种可用。它的第一个和第二个参数是canvas的宽和高。不仅如此，你可通过可配置的第三个参数去设置其他一些配置。第三个参数是一个字面量对象，这儿有一个如何通过它去设置抗锯齿、透明和分辨率的例子：

```
renderer = PIXI.autoDetectRenderer(  
    256, 256,  
    {antialias: false, transparent: false, resolution: 1}  
);
```

第三个参数是可配置的 - 如果你很满意Pixi的默认设置你可以不用管它，通常也并没有需要去改变他们。（但是，如果你需要，请查询有关[Canvas Renderer](#)和[WebGLRenderer](#) 了解更多信息）

这些配置都是干啥的？

```
{antialias: false, transparent: false, resolution: 1}
```

`antialias`（抗锯齿）能够平滑字体和图元的边缘。（WebGL的图形保真并不适用于所有的平台，所以你需要在你游戏的目标平台上测试一下。）`transparent`能让canvas背景透明。`resolution`能使在不同分辨率和像素密度的显示器上工作起来更简单。设置分辨率有点儿稍微超出本教程的范畴，查看[Mat Grove's explanation](#)中关于如果使用`resolution`可以找到所有细节。但是通常呢，在大多数项目里只需要保持`resolution`的值为1就好了。

（注意：渲染器有一个额外的、第四个选项叫 `preserveDrawingBuffer` 默认为 `false`。唯一需要设置它为 `true`的理由就是你是否需要在一个WebGL canvas上下文中调用Pixi的专用`dataToURL`方法。）

Pixi的渲染器对象会默认为WebGL，因为WebGL不可思议的快，并且能够让你使用一些将来你会学到的壮观的视觉效果。但是如果你需要强制使用Canvas绘图API渲染，你可以这样做：

```
renderer = new PIXI.CanvasRenderer(256, 256);
```

只有前两个参数是必要的：'width' 和 'height'。

你可以像这样强制WebGL渲染：

```
renderer = new PIXI.WebGLRenderer(256, 256);
```

`renderer.view` 对象只是一个再普通不过的 `<canvas>` 对象，所以你可以像控制其他任何 `canvas` 对象一样控制它。这里告诉你如何给`canvas`一个虚线边框：

```
renderer.view.style.border = "1px dashed black";
```

如果你想在创建`canvas`后改变它的背景颜色，设置 `renderer` 对象的 `backgroundColor` 属性为其他任何十六进制的颜色值：

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

如果你想获取 `renderer` 的宽和高，用 `renderer.view.width` 和 `renderer.view.height`。

(重要提示：尽管 `stage` 也有 `width` 和 `height` 属性，它们不是指渲染窗口的大小。舞台的 `width` 和 `height` 只是告诉你你放的东西所占用的区域 - 更多解释将会在下面看到！)

可以用 `renderer` 的 `resize` 方法设置任何新的 `width` 和 `height` 值去改变 `canvas` 的大小。但是，请确保 `canvas`和分辨率是大小相匹配的，设置 `autoResize` 为 `true`。

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

如果你想让`canvas`占据整个窗口，你可以设置CSS样式并且调整渲染器的大小为浏览器窗口的大小。

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

但是如果你那么做了，请确保用下面的一小段css代码设置你所有的HTML节点的默认内边距和外边距为0：

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

(在上面代码中的星号：*，是CSS的『通用选择符』，表示『在HTML文档中的所有标签』。)

如果你想让`canvas`可以自动缩放适应适应所有的浏览器窗口大小，你可以用[这个自动缩放窗口函数](#)。

Pixi 精灵

在上一个章节，你学会了如何去创建一个 `stage` 对象：

```
var stage = new PIXI.Container();
```

`stage` 是一个 `Pixi Container` 对象。你可以把这个容器想象成一个可以把所有你想放的东西放进去的一种空盒子。我们创建的 `stage` 对象是在你创建的场景中所有可视物体的根容器。Pixi需要你有一个根容器，因为 `renderer` 需要有一个东西去渲染上去：

```
renderer.render(stage);
```

不管你往这个 `stage` 里面放什么都会被渲染到canvas上面。现在这个 `stage` 是空的，但是一会儿我们就会往里面放一些东西。

(注意：你可以给你的根容器起任何你想起的名字。如果你喜欢的话可以把它叫做 `scene` 或者 `root`。 `stage` 这个名字仅仅是一个约定俗成的名字，我们也会在本教程中一直使用它。)

那么你想在舞台上放什么？特殊的图片对象被称为 **精灵图**。你可以控制它们的位置，尺寸以及其他许多有用的可以制作交互式动画图形的属性。学习如何控制一个精灵图对学习如何使用Pixi来说是十分重要的。如果你知道了如何制作精灵图并把它们添加到舞台上，你才刚刚迈出了制作游戏的一小步。

Pixi有一个 `Sprite` 类，它是创建游戏精灵的通用方式。这里还有三个主要的方式去创建它们：

- 使用单独一个图片文件
- 使用**图片集**的一个子图像。图片集是一个单独的，大的图像，它包含了所有游戏中你需要的图片。（译者注：就是我们说的雪碧图吧）。
- 使用**纹理图集**（一个定义了雪碧图中每个子图的位置和大小的JSON文件）

你将会学习所有的三种方式，但是在此之前，让我们先弄清楚利用pixi展示图片之前，你需要了解哪些关于图片的知识。

加载图像到纹理缓存中

因为Pixi通过WebGL在GPU上渲染图像，图像需要被格式化为GPU可以处理的格式。一个为WebGL准备的图像称为**纹理**。在你制作精灵展示图像之前，你需要把一个原始的图像转化为一个WebGL纹理。为了保持所有的东西都能在底层快速高效的工作，Pixi使用了一个 **纹理缓存** 去存储和引用所有精灵图所需要的纹理。纹理的名字是一个指向图片文件的位置的字符串。这意味着如果你有一个纹理是从 `"images/cat.png"` 载入的，那么你可以在纹理缓存中通过这种方式来找到它：

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

纹理都是以WebGL兼容格式存储的，这能让Pixi渲染器高效率的工作。然后你可以利用 `Sprite` 类通过纹理来制作一个新的精灵。


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

但是你怎么加载一个图片并把它转化为一个纹理？用Pixi的内置 `loader` 对象。

Pixi强大的`loader` 对象是你加载任何一种图像的全部所需。这里告诉你了如何加载一个图像并在图像加载完成之后调用 `setup` 函数。

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

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

[Pixi开发团队推荐](#) 如果你使用了加载器，你应该通过引用纹理中的`loader` 的 `resources` 对象来创建精灵，像这样：

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

这里有一个例子，可以通过它来加载一个图像，调用`setup` 函数，并且从加载过的图像中创建一个精灵：

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

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

我们将在本教程中使用这种一般的格式来图像和创建精灵。

你可以通过一连串的 `add` 方法来一次性加载许多图像：

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

更好的做法是，把所有的你想加载的文件放到一个数组里，只通过一个`add`方法：

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

`loader`也可以让你加载JSON文件，下面你将会学习到。

展示精灵

在你加载了一个图片并用它制作了一个精灵之后，想要在canvas上看到它，还有两件事情你不得不做。

- 1. 你需要通过 `stage.addChild` 方法把精灵添加到 Pixi的 `stage`中：

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

舞台是容纳所有精灵的主要容器。

- 2. 你需要告诉Pixi的 `renderer` 去渲染这个舞台。

```
renderer.render(stage);
```

在你不做以上两步之前任何精灵你都看不到

在我们继续之前，让我们看一个如何用刚才学的东西去展示一个单独的图片的实际例子。在 `examples/images` 文件夹中，你能找到一个64*64的猫的PNG图片。

基本展示

加载图像，创建一个精灵，并把它展示在Pixi舞台上的所有的JavaScript代码：

```
var stage = new PIXI.Container(),
    renderer = PIXI.autoDetectRenderer(256, 256);
document.body.appendChild(renderer.view);

//Use Pixi's built-in `loader` object to load an image
PIXI.loader
  .add("images/cat.png")
  .load(setup);

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

```
function setup() {  
  
  //Create the `cat` sprite from the texture  
  var cat = new PIXI.Sprite(  
    PIXI.loader.resources["images/cat.png"].texture  
  );  
  
  //Add the cat to the stage  
  stage.addChild(cat);  
  
  //Render the stage  
  renderer.render(stage);  
}
```

当运行以上代码时，你将会看到：



我们正在取得一些进展！

如果你想从舞台中移除一个精灵，用 `removeChild` 方法：

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

但是通常设置精灵的 `visible` 属性为 `false` 是一个让精灵消失的高效而且简单方式。

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

使用别名

你可以通过给Pixi对象和方法设置一个你经常用的简短的别名来使你的代码有更高的可读性。比如说，`PIXI.utils.TextureCache` 长不长？我认为长，特别是在一个大型项目你需要多次用到它。所以可以通过创建一个简短的别名指向它：

```
var TextureCache = PIXI.utils.TextureCache
```

然后，用别名替换之前的位置：

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

除了让你能给更加简洁的代码，用别名还有一个好处：它有助于你缓存Pixi频繁变动的API。如果Pixi的API在将来的版本中改变了 - 一定会变！ - 你仅仅需要在你的程序中更新你的别名，而不需要替换代码里所有的实例。所以当Pixi的开发团队决定重新调整的时候，你比他们还提前一步！

如何这么做，让我们重构刚才加载并展示图像的代码，给所有的Pixi对象和方法用别名。

```
//Aliases
var Container = PIXI.Container,
    autoDetectRenderer = PIXI.autoDetectRenderer,
    loader = PIXI.loader,
    resources = PIXI.loader.resources,
    Sprite = PIXI.Sprite;

//Create a Pixi stage and renderer and add the
//renderer.view to the DOM
var stage = new Container(),
    renderer = autoDetectRenderer(256, 256);
document.body.appendChild(renderer.view);

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

function setup() {

    //Create the `cat` sprite, add it to the stage, and render it
    var cat = new Sprite(resources["images/cat.png"].texture);
    stage.addChild(cat);
    renderer.render(stage);
}
```

本教程中的大部分例子都将遵从这个相同的范式来为Pixi设置别名。除非另有说明，你可以假设所有的例子代码都会像这样一样使用别名。

这就是加载图像和创建精灵你所需要知道的全部知识。

进一步了解『加载物品』

刚才上面给你展示的格式是我建议的一个标准的模板用于加载图像和展示精灵。所以你可放心的忽略接下来的一小段章节，直接跳到下一章，『定位精灵』。但是Pixi的 `loader` 对象是相当复杂的，包括一些你需要关心的特性，尽管在基础的用法中你可能用不到。让我们看一些最有用的东西。

从普通JavaScript图像对象或画布中创建一个精灵

为了优化和效率，最好的方式是从已经在Pixi的纹理缓存中拿出一个缓存来制作一个精灵。但是如果因为某些原因你需要通过普通JavaScript图像对象制作一个纹理，你可以通过用 `BaseTexture` 和 `Texture` 类：

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

如果你想从任何已经存在的canvas节点中制造一个纹理，你可以用 `BaseTexture.fromCanvas` 。

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

如果你想改变正在展示的精灵纹理，用 `texture` 属性。设置它的值为任何 `Texture` 对象：

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

如果在游戏里发生了什么重大的事情，你可以用这个技术来交互式的改变精灵的表现方式。

为加载文件分配一个名字

为每一个资源设置一个独一无二的名字是有可能的。只要在`add`方法的第一个参数传入一个字符串名字。举个例子，怎么给猫的图片命名为 `catImage`：

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

它在 `loader.resources` 中创建了一个名为 `catImage` 的对象。这意味着你可以通过引用 `catImage` 对象来创建一个精灵：

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

然而，我个人不推荐你用这种特性！这是因为你需要不得不记住你给每一个加载文件的所有名字，同时也得确保你不会不经意间多次使用相同的名字。使用文件别名，正如我们以前例子里的一样，更简单和不容易出错。

监控加载进度

Pixi的加载器有一个特殊的 `progress` 事件会在每次当一个文件加载的时候调用一个可定制的函数。 `progress` 事件会被`loader`的`on`方法调用：

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

如何在一个加载链中添加 `on` 方法，然后在每次文件加载的时候调用用户自定义的 `loadProgressHandler` 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");
}

```

每次文件加载的时候，进度事件都会调用 `loadProgressHandler` 在控制台打印 "loading"。当三个文件都加载完毕时，`setup` 函数会执行。这是上面的代码执行后在控制台的输出：

```

loading
loading
loading
setup

```

这很简洁，但还可以变得更好。你也可以明确的指明哪个文件已经加载了以及文件加载的百分比，可以通过给 `loadProgressHandler` 添加 `loader` 和 `resource` 参数：

```

function loadProgressHandler(loader, resource) { //...

```

然后你就可以用 `resource.url` 去找到当前已经加载的文件。（如果你想调用之前你通过 `add` 方法的一个参数给文件指定的名字，可以用 `resource.name`。）你也可以调用 `loader.progress` 来找出全部资源加载的百分比。下面的代码就做了上面的事情：

```

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 precentage 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");
}
```

下面的内容就是当上面的代码执行时会在控制台输出的信息：

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

这真的很酷，因为你可以以此为基础制作进度条。

(注意：你可以在 `resource` 对象里获取更多额外的属性。 `resource.error` 会告诉你在加载文件过程中任何可能发生的错误。 `resource.data` 能让你获取到文件的原始二进制数据。)

进一步了解『Pixi的加载器』

Pixi的加载器功能丰富而且可配置。在你开始使用之前，让我们快速一览一下它的用法吧。

加载器的可链式执行的 `add` 方法接收4个基本参数：

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

这里是加载器的源代码文档关于这些参数的描述：

`name` (string): 准备加载的资源的名字，如果不传递，则默认为 `url`。

`url` (string): 资源的路径, 和加载器的 `baseUrl` 相对。

`options` (object literal): 加载的配置。

`options.crossOrigin` (Boolean): 请求是跨域的吗? 默认是自动确定。

`options.loadType`: 这个资源是怎么被加载的? 默认值是 `Resource.LOAD_TYPE.XHR`。

`options.xhrType`: 当用XHR加载的资源时如何被解释? 默认值为 `Resource.XHR_RESPONSE_TYPE.DEFAULT`

。

`callbackFunction`: 当指定资源加载完成时调用此函数。

唯一必须写的参数就是 `url`（你想要加载的文件）。

下面是一些你可能用 `add` 方法加载文件的例子。第一部分的写法是文档中被称为加载器的『正常语法』：

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

这些例子是加载器的『对象语法』：

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

你也可以往 `add` 方法里传一个对象数组或者路径或者全部：

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

（注意：如果你想重置加载器去加载一批新的文件，调用加载器的 `reset` 方法：`PIXI.loader.reset();`）

Pixi的加载器有很多先进的功能，包括让你能配置加载和解析所有类型的二进制文件的选项。这不是你每天都要做的，也超出了本教程的范畴，所以[通过查看加载器的GitHub仓库获取更多信息](#)。

定位精灵

现在你知道如何创建和展示精灵图了，接下来让我们弄清楚如何定位和调节它们的尺寸吧。

在前面的例子中，猫精灵图被添加在舞台的左上角。猫的 `x` 和 `y` 位置都为0。你可以通过改变 `x` 和 `y` 的值来改变猫的位置。如何把猫居中的代码：

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

在创建完精灵图之后，把这两行加入到 `setup` 函数中：


```
function setup() {  
  
  //Create the `cat` sprite  
  var 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  
  stage.addChild(cat);  
  
  //Render the stage  
  renderer.render(stage);  
}
```

(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。我将会接下的例子的方法中都用这种格式。)

这两行新的代码会把猫向右移动96像素，向下移动96像素，这是结果：

猫在舞台居中

猫的左上角（它的左耳朵）代表了它的 `x` 和 `y` 的锚点。增加 `x` 属性值，可以让猫能够向右移动；增加 `y` 属性值，可以让猫能够向下移动。

猫在舞台居中 - 图表

你可以用一行代码把`x`和`y`的值一起设置，而不是单独设置：

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

尺寸和缩放

你可以通过设置 `width` 和 `height` 来改变精灵图的大小，下面代码展示了如何给猫一个80像素宽和120像素高的大小：

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

在 `setup` 函数中添加这两行代码：

```
function setup() {  
  
  //Create the `cat` sprite  
  var 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  
  stage.addChild(cat);  
}
```

这是结果：



猫的高和宽改变

你会看到猫的位置（它的左上角）没有改变，只是它的宽和高改变了。



猫的宽和高改变 - 图表

精灵图页有 `scale.x` 和 `scale.y` 属性可以成比例的改变精灵的宽和高。这里告诉你如何设置猫的大小为其原来的一半：

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

缩放的值在0和1之间取值，代表精灵图尺寸的百分比。1 代表 100%（全部尺寸），0.5 代表 一半尺寸。你可以通过设置缩放的值为2，从而成倍的设置精灵图的尺寸：

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

Pixi有一个可供替代的，简洁的方式通过一行代码 `scale.set` 方法设置精灵的尺寸。

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

如果它很吸引你，就用它！


旋转

你可以通过设置 `rotation` 属性的弧度值来使精灵旋转。

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

但是在哪一点发生旋转呢？

你已经知道了精灵的左上角代表它的 `x` 和 `y` 位置。这个点叫做 **锚点**。如果你设置了精灵的 `rotation` 的属性值为 `0.5`，精灵的旋转会围绕它的锚点旋转。这个图表展示了猫精灵会发生什么效果。

 通过锚点旋转精灵 - 图表

你可以看到锚点，猫的左耳朵，是猫在旋转时候的假想圆的中心点。如果你想让精灵图围绕自己的中心旋转呢？改变 `anchor` 的值，让它定位到精灵的中心：

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

`anchor.x` 和 `anchor.y` 的值代表了纹理尺寸的百分比，从0 到 1（0% 到 100%）。把它设置为0.5可以把纹理的锚点设在中间。点的位置本身不会改变，只是纹理放在它上面的方式改变了。

下面一张图展示了当你把锚点居中时会发生什么。

 居中旋转 - 图表

你可以看到精灵的纹理向上和向左移动了。这是一个重要的需要记住的副作用！

就像设置 `position` 和 `scale` 一样，你同样可以通过一行代码设置锚点的 `x` 和 `y` 的值：

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

精灵图有一个 `pivot`（中心点）属性，它和 `anchor` 的工作方式很相似。如果你改变了中心点的值并旋转精灵，它将围绕这个原点旋转。举个例子，下面的代码会设置精灵的 `pivot.x` 点为32，以及它的 `pivot.y` 为32。

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

设想一下精灵图为64*64大小，这个精灵现在将会围绕他的中心点旋转了。但是谨记：如果你改变精灵的中心点，你同时也改变了它的 `x/y` 原点。

所以，`anchor` 和 `pivot` 的区别到底是什么？它们真的很相似！`anchor` 可以移动精灵图的纹理原点，通过设置 0-1 的值。`pivot` 通过设置像素值来改变精灵的 x 和 y 的原点值。


用雪碧图的子图像制作精灵

你现在知道了如何用一个单一的图像文件制作精灵。但是，作为一个游戏设计者，你通常会把你的精灵图制作成**图集**（也叫**雪碧图**。）Pixi 有一些内置的简便的方式帮助你做到这些。一个雪碧图是包含了很多子图的一个单一图片文件。子图是你在游戏中准备用到的所有图片。这里有一个雪碧图的例子，它包含了游戏角色和游戏对象作为子图。

 一个雪碧图的例子

整张雪碧图的大小是 192x192。每个图片都在它自己的 32x32 的网格中。把你的所有游戏图像保存到雪碧图中并从中获取是一种能让处理器和内存高效率处理图像的方式。Pixi 也给它做了优化。

你可以通过定义一个和子图相同大小和位置的矩形区域来捕捉你想提取的子图。这里有一个从雪碧图中提取火箭子图的例子：

 从雪碧图中提取的火箭

让我们看看这段代码，首先通过 Pixi `loader` 的加载 `tileset.png` 图片，就像在之前的例子你做的那样：

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

接下来，当图片加载完成，用一个雪碧图的矩形项去创建一个精灵图片。这里的代码就展示如何提取子图，创建火箭图像，定位并且在 canvas 中展示它。

```
function setup() {

  //Create the `tileset` sprite from the texture
  var 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
  var rectangle = new Rectangle(192, 128, 64, 64);

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

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

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

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

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

这是怎么工作的？

Pixi有一个内置的 `Rectangle` 对象 (`PIXI.Rectangle`) 是一个通用的定义矩形形状的对象。它接收四个参数。前两个参数定义了矩形的x和y的位置。后两个定义了它的宽和高。这里格式化定义了一个新的 `Rectangle` 对象。

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

矩形对象只是一个数据对象；你想怎么使用取决于你。在我们的例子中，我们使用它定义了雪碧图的一个位置和区域去提取我们想要的子图。Pixi纹理有一个非常有用的属性叫：`frame`，它能被设置为任何 `Rectangle` 对象。`frame` 通过 `Rectangle`所标尺寸来裁剪纹理。这里告诉你如何用 `frame` 来裁剪火箭纹理的大小和位置。

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

你可以用裁剪过的纹理去创建精灵：

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

这就是它工作的原理！


因为从雪碧图创建精灵纹理是一个你将经常做的事情，Pixi有更多简便的方法去帮助你做这些 - 接下来让我们弄清楚这些方法。

使用纹理图集

如果你正在开发一个大型的、复杂的游戏，你需要一个快速高效的方式从雪碧图中制作精灵。所以**纹理图集**就变得非常有用了。纹理图集是一个包含了雪碧图PNG文件所有子图大小和位置的JSON文件。如果你用了纹理图集，只需要知道子图的名字就能展示它。你可以把你的雪碧图整理成任何次序，这个JSON文件会保持跟踪它们的位置和大小。这十分便捷，意味着雪碧图子图的大小和位置在你的游戏编程中将变得不再困难。如果你对雪碧图做了改变，比如添加图像，缩放它们，或者移动它们，只需要重新编译一个JSON文件，你的游戏会利用这些数据去展示正确的图像。你不需要把你游戏代码做任何改变。

Pixi兼容从**Texture Packer**输出的标准的JSON纹理图集格式，Texture Packer很受欢迎。Texture Packer 的“Essential”许可证是免费的。让我们弄清楚如何用它来制作一个纹理图集，然后把图集加载到Pixi。（你不需要必须用Texture Packer。相似的工具还有 **Shoobox** 或者 **spritesheet.js**，输出的PNG和标准格式的JSON文件都是和Pixi兼容的。）

首先，从你需要在你游戏中使用的一系列单一图像开始。

 图片文件

(在本章节的所有图片文件都是由 Lanea Zimmerman 创造的。你可以在[这儿](#)进一步了解它的作品，感谢 Lanea!)

接下来，打开 Texture Packer 并选择 **JSON Hash** 作为框架类型。把你的图片都拖拽到 Texture Packer 的工作区。(你也可以让 Texture Packer 指向游戏包含的文件夹。) 它会自动在雪碧图上整理些子图，然后设定它们的名字为它们原始图片的名字。

图片文件

(如果你正在使用免费版的Texture Packer，设置 **Algorithm** 为 **Basic**，设置 **Trim mode** 为 **None**，设置 **Size constraints** 为 **Any size** 然后滑动 **PNG Opt Level**到最左为 **0**。这是让你利用免费版的 Texture Packer 创建你的文件不会报错和警告的最基本的做法。)

当你完成之后，点击 **Publish** 按钮。选择文件的名字的位置，然后保存发布文件。你将会得到两个文件：一个 PNG文件和一个JSON文件。在这个例子中我的文件名字叫 **treasureHunter.json** 和 **treasureHunter.png**。方便起见，把它们都放在你项目的 **images** 文件夹中。(你可以认为JSON文件是图片文件的元数据，所以保持两个文件在同一个文件夹是有意义的。) 这个JSON文件描述了每个子图的名字，尺寸和位置。这里是一个描述了怪物子图的摘要：

```
"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}
},
```

treasureHunter.json 文件也用同样的方式包含了 "dungeon.png", "door.png", "exit.png", 和 "explorer.png"。每一个子图被称为 **frames**。拥有这些数据是非常有帮助的，因为你不需要在知道雪碧图中每个子图的大小和位置了。所有你需要知道的就是精灵的 **frame id**。这个id就是元图片的名字，比如说 "blob.png" 或者 "explorer.png"。

使用纹理图集的优点之一是你轻松的为每个子图添加2像素的内边距 (Texture Packer 默认这么做) 这是防止**纹理流血**的重要做法。纹理流血的现象会在当相邻图像的边缘在另一个精灵图旁边是出现，它出现的原因是你的电脑的GPU (图像处理单元) 如何去四舍五入像素值。是该舍弃还是该进入？每个GPU都可能不一样。所以在图片的周围保留1或者2两个像素可以使图片一直显示正常。

(注意：如果你有在图片周围保留了两个像素的空白，在Pixi里展示的时候你需要注意一个奇怪的"一像素"差错，尝试改变纹理的尺度模式算法。这里是：`texture.baseTexture.scaleMode = PIXI.SCALE_MODES.NEAREST`；这些差错会在GPU舍入浮点数误差时出现。)

现在你知道如何制作一个纹理图集了，然后让我们弄清楚如何在你的游戏代码中加载它。

加载纹理图集

为了把纹理图集加入Pixi，用Pixi的 **loader** 加载它。如果你是用 Texture Packer 制作的JSON文件，**loader** 会自动在雪碧图上解释数据和创建纹理图。这里是如何用 **loader** 去加载 **treasureHunter.json** 文件的代码，

当加载完成，`setup` 执行：

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

现在雪碧图的每个子图在Pixi的缓存里都是一个单独的纹理了。你可以通过在 Texture Packer 设定的名字 ("blob.png", "dungeon.png", "explorer.png", 等)来获取每一个纹理。

从加载过的纹理图集创建精灵

Pixi给了你三种方式从加载过的纹理图集创建精灵：

1. 用 `TextureCache`:

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

2. 如果你用了Pixi的 `loader` 去加载纹理图集，用加载器的 `resources`：

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

3. 用这种方式去创建精灵要码太多字了！所以我建议为纹理图集的 `textures` 对象取一个别名 `id` 指向它：

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

然后你可以像这样创建每一个精灵：

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

好多了！

这里告诉你了如何在`setup` 函数中用三种不同的创建方式去展示`dungeon`, `explorer`, 和 `treasure` 精灵：

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

```
function setup() {

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

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

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

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

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

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

这是这段代码执行后的显示效果:

 Explorer, dungeon and treasure

舞台的尺寸是512x512像素，你可以在上面的代码中看到，`stage.height` 和 `stage.width` 属性被用来定位精灵。探险者的y坐标如何被垂直居中：

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

学习如何用纹理图集去创建和展示精灵是一个重要的基准。在我们继续之前。让我们看一下如何把剩余的精灵：怪物和出口，添加进来，然后你会创建一个像这样的场景：

 All the texture atlas sprites

这是做到上面效果的所有代码。我把HTML代码页添加进来了，所以你可以在它的上下文中看到所有东西。

(你会在这个仓库的[examples/spriteFromTextureAtlas.html](#)文件看到这个作品) 注意怪物精灵是被循环创建和添加到舞台中的，并被赋予了随机的位置。

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

//Aliases
var Container = PIXI.Container,
    autoDetectRenderer = PIXI.autoDetectRenderer,
    loader = PIXI.loader,
    resources = PIXI.loader.resources,
    TextureCache = PIXI.utils.TextureCache,
    Texture = PIXI.Texture,
    Sprite = PIXI.Sprite;

//Create a Pixi stage and renderer and add the
//renderer.view to the DOM
var stage = new Container(),
    renderer = autoDetectRenderer(512, 512);
document.body.appendChild(renderer.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
var dungeon, explorer, treasure, door, id;

function setup() {

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

    //1. Access the `TextureCache` directly
    var dungeonTexture = TextureCache["dungeon.png"];
    dungeon = new Sprite(dungeonTexture);
    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 = stage.height / 2 - explorer.height / 2;
```

```
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"]);
stage.addChild(treasure);

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

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

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

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

    //Make a blob
    var 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.
    var x = spacing * i + xOffset;

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

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

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

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

//The `randomInt` helper function
function randomInt(min, max) {
```

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

</script>
</body>
```

你会在上面的代码中看到所有的怪物都是通过for循环创建的。每个怪物在x轴上被均匀间隔开来：

```
var x = spacing * i + xOffset;
blob.x = x;
```

`spacing` 的值为48, `xOffset` 的值为150, 这意味着第一个怪物的x位置为150。这个偏移量是距离舞台左侧的150像素。每一个后来的怪物都会比前一个怪物多48像素。这就创建了一行从左到右, 间隔均匀的怪物。

每个怪物被赋予了一个随机的y值：

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

怪物的y值会被赋予一个0-512的随机值, 不回超过`stage.height`的值。 `randomInt` 返回一个随机值：

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

如果你想获取1-10的随机值：

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

这里是 `randomInt` 函数的实现方式：

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

`randomInt` 是一个很棒的小函数, 我会一直使用它。

移动精灵

现在你知道了如何展示精灵, 但是让它们动起来呢? 很简单: 用 `requestAnimationFrame` 创建一个循环函数。这被称为 **游戏循环**。任何在游戏循环里的代码都会1s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。

```
function gameLoop() {

    //Loop this function at 60 frames per second
    requestAnimationFrame(gameLoop);

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

    //Render the stage to see the animation
    renderer.render(stage);
}

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

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。

Moving sprites

这就是移动的全部。只要在循环中改变一小点的增加精灵的属性，它们就会随着时间动画。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：

```
//Aliases
var Container = PIXI.Container,
    autoDetectRenderer = PIXI.autoDetectRenderer,
    loader = PIXI.loader,
    resources = PIXI.loader.resources,
    Sprite = PIXI.Sprite;

//Create a Pixi stage and renderer
var stage = new Container(),
    renderer = autoDetectRenderer(256, 256);
document.body.appendChild(renderer.view);

//Load an image and the run the `setup` function
loader
    .add("images/cat.png")
    .load(setup);

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

function setup() {

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

    //Start the game loop
```

```
    gameLoop();
}

function gameLoop(){

    //Loop this function 60 times per second
    requestAnimationFrame(gameLoop);

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

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

(注意 `cat` 变量需要在 `setup` 和 `gameLoop` 函数之外定义，然后你可以在里面都能获取到它们)

你可以动画精灵的尺寸，旋转或者尺寸 - 任何都可以！你会在下面看到更多动画精灵的例子。

利用速度特性

为了给你更多的灵活性，用两个 **速度属性**：`vx` 和 `vy`，去控制精灵的运动速度是一个不错的方式。`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。直接更新速度变量并且给精灵指派这些速度值。如果你需要交互性的游戏动画，这是一个额外的模块。

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 表示精灵不移动。

接下来，在游戏循环中，更新 `vx` 和 `vy` 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猫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);
    stage.addChild(cat);

    //Initialize the cat's velocity variables
    cat.vx = 0;
    cat.vy = 0;

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



```
function gameLoop(){

    //Loop this function 60 times per second
    requestAnimationFrame(gameLoop);

    //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;

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

当你运行这段代码，猫会每帧像右下方移动一个像素：



如果你想让猫在不同的方向移动怎么办？为了让猫向左移动，可以给它的 `vx` 赋值为 `-1`。为了让猫向上移动，可以给猫的 `vy` 赋值为 `-1`。为了让猫移动的更快一点，把 `vx` 和 `vy` 的值设的更大一点，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现象。

游戏状态

作为一种风格，也是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：

```
//Set the game's current state to `play`:
var state = play;

function gameLoop() {

    //Loop this function at 60 frames per second
    requestAnimationFrame(gameLoop);

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

    //Render the stage to see the animation
    renderer.render(stage);
}

function play() {
```

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

你会看到 `gameLoop` 每秒60次调用了 `state` 函数。 `state` 函数是什么？ 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果用这个新模式来重构上一个例子的代码：

```
//Define any variables that are used in more than one function
var 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;
  stage.addChild(cat);

  //Set the game state
  state = play;

  //Start the game loop
  gameLoop();
}

function gameLoop(){

  //Loop this function 60 times per second
  requestAnimationFrame(gameLoop);

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

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

function play() {

  //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，花几分钟在脑海中过一遍这些函数是如何联系在一起的。正如你将在前面看到的，结构化你的游戏循环代码，会使像切换游戏场景和关卡变得更简单。

键盘移动

只需再做一小点工作，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) {
  var key = {};
  key.code = keyCode;
  key.isDown = false;
  key.isUp = true;
  key.press = undefined;
  key.release = undefined;
  //The `downHandler`
  key.downHandler = function(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 = function(event) {
    if (event.keyCode === key.code) {
      if (key.isDown && key.release) key.release();
      key.isDown = false;
      key.isUp = true;
    }
    event.preventDefault();
  };

  //Attach event listeners
  window.addEventListener(
    "keydown", key.downHandler.bind(key), false
  );
  window.addEventListener(
    "keyup", key.upHandler.bind(key), false
  );
  return key;
}
```

`keyboard` 函数很容易使用，可以像这样创建一个新的键盘对象：

```
var 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ASCII值列表](#)。

然后给键盘对象赋值 `press` 和 `release` 方法：

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

键盘对象也有 `isDown` 和 `isUp` 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` 文件你如何用 `keyboard` 函数，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。

 Keyboard movement

这里是代码：

```
function setup() {  
  
    //Create the `cat` sprite  
    cat = new Sprite("images/cat.png");  
    cat.y = 96;  
    cat.vx = 0;  
    cat.vy = 0;  
    stage.addChild(cat);  
  
    //Capture the keyboard arrow keys  
    var left = keyboard(37),  
        up = keyboard(38),  
        right = keyboard(39),  
        down = keyboard(40);  
  
    //Left arrow key `press` method  
    left.press = function() {  
  
        //Change the cat's velocity when the key is pressed  
        cat.vx = -5;  
        cat.vy = 0;  
    };  
  
    //Left arrow key `release` method  
    left.release = function() {  
  
        //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 = function() {
    cat.vy = -5;
    cat.vx = 0;
};
up.release = function() {
    if (!down.isDown && cat.vx === 0) {
        cat.vy = 0;
    }
};

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

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

//Set the game state
state = play;

//Start the game loop
gameLoop();
}

function gameLoop() {
    requestAnimationFrame(gameLoop);
    state();
    renderer.render(stage);
}

function play() {

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

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`，可以让你做这些。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*

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

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

//The tiger
var tiger = new Sprite(id["tiger.png"]);
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：

```
var animals = new Container();
```

然后用 `addChild` 去把精灵图添加到分组中。

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

最后把分组添加到舞台上。

```
stage.addChild(animals);
renderer.render(stage);
```

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

这就是上面代码产生的效果：

 Grouping sprites

你是看不到这个包含精灵图的 `animals` 分组的。

 Grouping sprites

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。

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

这告诉你 `animals` 有三个精灵孩子。

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。任何你改变父容器的属性值，都会相对的改变它的孩子精灵。所以如果你设置分组的 `x` 和 `y` 的位置，所有的孩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？

```
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。你可以通过下面的方式来获取 `width` 和 `height` 的值：

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

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



如果你改变了分组的宽和高会发生什么呢？

```
animals.width = 200;
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，这是一个你不用担心的有用的管理方式。

局部和全局位置

当你往一个容器添加一个精灵时，它的 **x** 和 **y** 的位置是*相对于分组的左上角*。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：

```
console.log(cat.x);  
//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：

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

这意味着你能在 **animals** 分组里找到猫的全局位置：

```
console.log(animals.toGlobal(cat.position));  
//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。

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额外的得到猫的全局位置：

```
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，所以这里需要注意啦！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：

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

它会给你返回 `x` 和 `y` 的值为128. 一个关于 `getGlobalPosition` 特殊的事情是它是高精度的：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。

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：

```
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。

```
tiger.toLocal(tiger.position, hedgehog).x  
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 去分组精灵

Pixi有一个额外的，高性能的方式去分组精灵叫：`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。这是用于提升游戏的一个很棒的性能。

可以像这样创建 `ParticleContainer`：

```
var superFastSprites = new 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（所以你不得不更换雪碧图如果你想让精灵有不同的表现方式）。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`，有两个参数可以传递，这个容器可以包裹的最大数量的精灵以及其他可配置对象。

```
var superFastSprites = new ParticleContainer(size, options);
```

默认的尺寸是 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：

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


但是，如果你感觉你不会用这些属性，就保持它们为 `false` 以挤出更大的性能。

`uvs` 是什么鬼？只有当它们在动画的时候需要改变它们纹理颗粒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**。）

Pixi的图元

用图片纹理是制作精灵最常用的方式，但是Pixi也有它自己的底层绘图工具。你可以用它们来制作矩形，形状，线条，复杂的多边形和文本。不但如此，幸运的是，它用了和Canvas绘图API几乎一样的API。如果你已经对canvas非常熟悉了，这就没什么新的东西要学了。但是有一个巨大的优势是，和Canvas的绘图API不同的是，用Pixi画的形状是通过WebGL在GPU上渲染的。Pixi让你能够达到所有未开发的性能。让我们来看看如何制作一些基本形状，这是我们要在代码里创建的所有形状：

 Graphic primitives

矩形

每个形状都是 Pixi 的 `Graphics` 类 (`PIXI.Graphics`)的一个新实例。

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

用 `beginFill` 传入一个16进制的色值去设置矩形的填充颜色。这儿告诉你如何试着它为浅蓝：

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

如果你想给形状一个轮廓，用 `lineStyle` 方法。这儿告诉你如果给矩形一个4像素宽的红色轮廓，透明度为1。

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

用 `drawRect` 方法画一个矩形。它有四个参数：`x`, `y`, `width` 和 `height`。

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

当你完成时，调用 `endFill`。

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

跟Canvas的绘图API很像！这就是你画一个矩形，改变它的位置，然后添加到舞台上的所有代码：

```
var 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;
stage.addChild(rectangle);
```

这段代码制作了一个64x64的，蓝色的，带红色边框的，x和y的位置是170的矩形。

圆形

用 `drawCircle` 方法制作一个圆形。它的三个参数是：`x`, `y` 和 `radius`。

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

和矩形以及精灵图不同的是，圆形的x和y的值是它的中心点，这儿告诉你如何制作一个半径为32像素，蓝紫色的圆形。

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

椭圆

比Canvas的绘图API更胜一筹的是，Pixi能让你通过 `drawEllipse` 方法画一个椭圆。

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

x/y的位置定义了椭圆的左上角（想象一下椭圆被一个可见的矩形包围 - 这个矩形的左上角代表了椭圆的x和y的锚点）。这是一个宽为50像素，高为20像素的黄色椭圆。

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

圆角矩形

Pixi也能让你通过 `drawRoundedRect` 方法制作圆角矩形。最后一个参数，`cornerRadius`是一个以像素为单位的数值，决定了圆角的弯曲程度。

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

下面告诉你如何制作一个圆角为10像素的圆角矩形。

```
var 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;
stage.addChild(roundBox);
```

线条

你已经在前面的例子中看到 `lineStyle` 方法能够让你定义一条线。你可以用 `moveTo` 和 `lineTo` 方法去画线条的开始和结束点，跟你在Canvas绘图API是一样的。这告诉你如何画一个4像素宽的，白色的对角线。

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

`PIXI.Graphics` 对象，像线条，有 `x` 和 `y` 的值，和精灵是一样的，所以在画完他们之后，你可以把它们定位在舞台上的任何地方。

多边形

你可以把线条聚合在一起然后给它们填充不同的颜色从而用 `drawPolygon` 方法制作复杂的图形。

`drawPolygon` 的参数是一个xy点的数组，定义了图形上的每个点的位置：

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

graphicsObject.drawPolygon(path);
```

`drawPolygon` 会把这三个点结合在一起制作一个形状。这里告诉你如何用 `drawPolygon` 去连接三条线在一起，去做一个蓝色边框的红色三角形。这个三角形在0,0位置被绘制，然后用它的 `x` 和 `y` 属性移动它在舞台上的位置。

```
var 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;
```

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

展示文字

用 `Text` 对象 (`PIXI.Text`) 在舞台上显示文字。这个构造函数接收两个参数：一个是你想展示的文字，一个是样式对象，它定义了字体的属性。这里告诉你如何展示一个白色的，32像素高的，“Hello Pixi”单词。

```
var message = new Text(  
    "Hello Pixi!",  
    {fontFamily: "Arial", fontSize: 32, fill: "white"}  
);  
  
message.position.set(54, 96);  
stage.addChild(message);
```



Displaying text

Pixi的文字对象继承了 `Sprite` 类，所以它们包含了像 `x`, `y`, `width`, `height`, `alpha` 和 `rotation` 相同的属性。在舞台上定位和缩放文字跟其他精灵是一样的。

如果你想改变文字对象的内容，用 `text` 属性。

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

用 `style` 属性去重新定义字体的属性。

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

Pixi通过使用Canvas的绘图API去渲染文字为一个无形的临时画布元素。然后它这个画布转化为WebGL纹理，让它能够映射到一个精灵元素。这就是为什么文字的颜色需要是字符串格式：它是Canvas绘图API的颜色值。作为任何其他Canvas颜色值，你可以用像“red”或者“green”，或者用 `rgba`, `hsla` 或者 `hex` 值。

其他你可以添加的样式属性有 `stroke` 用来给字体设定轮廓的颜色，还有 `strokeThickness` 设置文字的轮廓的厚度。设置文字的 `dropShadow` 的属性为 `true` 可以让文字显示阴影。用 `dropShadowColor` 设置阴影一个16进制的颜色值，用 `dropShadowAngle` 设置阴影的角度弧度，用 `dropShadowDistance` 设置阴影的像素高度。这儿还有更多: [查看Pixi的文字文档](#)。

Pixi也能够包裹很长的一行文字。设置文字的 `wordWrap` 样式为 `true`，然后设置 `wordWrapWidth` 为最大长度像素 - 就是这行文字该有的长度。用 `align` 属性设置多行文字的对齐方式。

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

(注意: `align` 不影响单行文字。)

如果你想使用自定义字体, 用CSS的 `@font-face` 规则在Pixi应用运行的HTML页面中去链接字体文件。

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

添加这个 `@font-face` 规则到你的HTML页面的CSS样式表中。

Pixi也支持点阵字体。你可以用Pixi的加载器加载点阵字体XML文件, 跟你加载JSON和图片文件是一样的。

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。你可以用自定义的函数叫做: `hitTestRectangle` 用来检测两个矩形精灵是否接触。

```
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:

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

正如你所看到的, `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。

 Displaying text

你已经看到了创建这些所有元素的代码, 包括键盘控制系统可以使猫移动。唯一的新的东西就是 `hitTestRectangle` 函数被用在 `play` 函数里用来检测碰撞。

```
function play() {  
  
  //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;  
  
}  
}
```

因为 `play` 函数被每秒调用了60次，这个 `if` 条件语句持续的在猫和盒子之间进行碰撞检测，如果 `hitTestRectangle` 为 `true`，这个文字 `message` 对象用 `setText` 去显示 "Hit"：

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

这个盒子的颜色通过设置盒子的 `tint` 属性一个16进制的红色的值，把盒子从绿色变为红色。

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

如果没有碰撞，消息和盒子会保持它们的原始状态。

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

代码很简单，但是你突然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，你现在可能已经拥有了你需要用Pixi制作游戏的全部技能！

hitTestRectangle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点儿超出了本教程的范围。最重要的事情是你知道如何使用它。但是，只是作为你的参考资料，防止你好奇，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？

```
function hitTestRectangle(r1, r2) {  
  
    //Define the variables we'll need to calculate  
    //定义我们需要计算的变量  
    var hit, combinedHalfWidths, combinedHalfHeights, vx, vy;
```

```
//hit will determine whether there's a collision
//hit决定是否有碰撞
hit = false;

//Find the center points of each sprite
//找到每个精灵的中心点
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
//找到每个精灵的一半宽度和一半高度
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
//计算精灵之间的向量距离
vx = r1.centerX - r2.centerX;
vy = r1.centerY - r2.centerY;

//Figure out the combined half-widths and half-heights
//计算出两个精灵的的一半宽度和一半高度的和
combinedHalfWidths = r1.halfWidth + r2.halfWidth;
combinedHalfHeights = r1.halfHeight + r2.halfHeight;

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

    //A collision might be occurring. Check for a collision on the y axis
    //碰撞可能发生, 检查y轴碰撞
    if (Math.abs(vy) < combinedHalfHeights) {

        //确实有碰撞发生
        hit = true;
    } else {

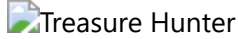
        //在y轴没有碰撞
        hit = false;
    }
} else {

    //在x轴没有碰撞
    hit = false;
}

//`hit` 返回 `true` 或者 `false`
return hit;
};
```

实战教学：宝藏猎手

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！让我们来做简单的对象收集和躲避的敌人的游戏叫：**宝藏猎手**。（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之一，它是一个很好的例子，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携带它出去。六只怪物在地牢的地板上上下下移动，如果它们碰到了探险者，探险者变为半透明，而且他右上角的血槽会收缩。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：纹理图集图像，互动性，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：

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

用这个作为你游戏代码的蓝图，让我们看看每一部分是如何工作的。

在 setup 函数中初始化游戏

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。这是一个很好的地方去创建和初始化对象、精灵、游戏场景、填充数据数组或解析加载JSON游戏数据。

这是宝藏猎手 `setup` 函数的缩略图和它要执行的任务。

```
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  
  gameLoop();  
}
```

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 `Container` 分组被称为 `gameScene` 和 `gameOverScene`。每一个都被添加到了舞台上。

```
gameScene = new Container();  
stage.addChild(gameScene);  
  
gameOverScene = new Container();  
stage.addChild(gameOverScene);
```

所有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。

 Displaying text

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`。

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

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

制作地牢，门，探险家和宝藏

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，他们都是被当做 `gameScene` 的孩子添加进来的。

```
//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);
```

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。

制造一堆怪物

六个怪物是被循环创建的。每一个怪物都被赋予了一个随机的初始位置和速度。每个怪物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。

```
var 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 (var i = 0; i < numberOfBlobs; i++) {

    //Make a blob
    var 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
    var x = spacing * i + xOffset;

    //Give the blob a random `y` position
    var 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);
}
```

制作血条

当你玩儿宝藏猎手的时候，你会发现当探险者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。

```
//Create the health bar
healthBar = new PIXI.DisplayObjectContainer();
healthBar.position.set(stage.width - 170, 6)
gameScene.addChild(healthBar);

//Create the black background rectangle
var innerBar = new PIXI.Graphics();
```

```
innerBar.beginFill(0x000000);
innerBar.drawRect(0, 0, 128, 8);
innerBar.endFill();
healthBar.addChild(innerBar);

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

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。

```
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`。

```
message = new Text(
    "The End!",
    {font: "64px Futura", fill: "white"}
);

message.x = 120;
message.y = stage.height / 2 - 32;

gameOverScene.addChild(message);
```

操作游戏

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() {  
  //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  
}
```

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。

```
explorer.x += explorer.vx;  
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。

 Displaying text

通过一个名为 `contain` 的自定义函数可以帮助实现。

```
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) {  
  
  var 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，它就会被掉头。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。

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

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

  //Check the blob's screen boundaries
  var 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;
}
});
```

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：

```
var 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：

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

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` 去指名是否有敌人碰到了探险者。

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

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` 函数让探险者变为半透明，然后把 `health` 条减少1像素的宽度。

```
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;
}
```

如果 `explorerHit` 是 `false`，探险者的 `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。看起来像是探险者携带者宝藏一样。

 Displaying text

这段代码实现了上述效果：

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

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下游戏，或者你的血用完你就失败了。

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(){

    //Loop this function 60 times per second
    requestAnimationFrame(gameLoop);

    //Update the current game state
    state();

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

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！

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

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。

这就是完成宝藏猎手所需要的一切了。然后在通过做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！

进一步了解『精灵』

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的精灵遵循以下链条构建了一个继承模型：

```
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，不久你就会学习如何用 `Container` 去给精灵分组。

更进一步

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。
- [Charm](#): 给Pixi精灵准备的简单易用的欢动动画效果。
- [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。 [Learn 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。

请帮助支持这个项目!

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!

[Learn 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。

怎么做到的:

- 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。

如果你想支持这个项目，请买一份本书，然后在买一份给你的妈妈！2333