Pixi.js Guide

Welcome

PixiJS is an open source, web-based rendering system that provides blazing fast performance for games, data visualization, and other graphics intensive projects. These guides are designed to be a companion to the API documentation, providing a structured introduction to using the API to solve problems and build projects.

About The Guides

If you're new to PixiJS, we suggest you start with the Basics and read through them in order (a good place to start is Getting Started). While PixiJS has a mature API and solid documentation, the guides go over many common issues and questions that developers new to the system encounter.

Other Resources

As you explore the guides, you may find these resources valuable:

What PixiJS Is

So what exactly *is* PixiJS? At its heart, PixiJS is a rendering system that uses WebGL (or optionally Canvas) to display images and other 2D visual content. It provides a full scene graph (a hierarchy of objects to render), and provides interaction support to enable handling click and touch events. It is a natural replacement for Flash in the modern HTML5 world, but provides better performance and pixel-level effects that go beyond what Flash could achieve. It is perfect for online games, educational content, interactive ads, data visualization... any web-based application where complex graphics are important. And coupled with technology such as Cordova and Electron, PixiJS apps can be distributed beyond the browser as mobile and desktop applications.
Here's what else you get with PixiJS:

PixiJS Is ... Fast

One of the major features that distinguishes PixiJS from other web-based rendering solutions is *speed*. From the ground up, the render pipeline has been built to get the most performance possible out of your users' browsers. Automatic sprite and geometry

batching, careful use of GPU resources, a tight scene graph - no matter your application, speed is valuable, and PixiJS has it to spare.

... More Than Just Sprites

Drawing images on a page can be handled with HTML5 and the DOM, so why use PixiJS? Beyond performance, the answer is that PixiJS goes well beyond simple images. Draw trails and tracks with MeshRope. Draw polygons, lines, circles and other primitives with Graphics. Text provides full text rendering support that's just as performant as sprites. And even when drawing simple images, PixiJS natively supports spritesheets for efficient loading and ease of development.

... Hardware accelerated

JavaScript has two APIs for handling hardware acceleration for graphical rendering: WebGL and the more modern WebGPU. Both essentially offer a JavaScript API for accessing users' GPUs for fast rendering and advanced effects. PixiJS leverages them to efficiently display thousands of moving sprites, even on mobile devices. However, using WebGL and WebGPU offers more than just speed. By using the Filter class, you can write shader programs (or use pre-built ones!) to achieve displacement maps, blurring, and other advanced visual effects that cannot be accomplished with just the DOM or Canvas APIs.

... Open Source

Want to understand how the engine works? Trying to track down a bug? Been burned by closed-source projects going dark? With PixiJS, you get a mature project with full source code access. We're MIT licensed for compatibility, and hosted on GitHub for issue tracking and ease of access.

... Extensible

Open source helps. So does being based on JavaScript. But the real reason PixiJS is easy to extend is the clean internal API that underlies every part of the system. After years of development and 5 major releases, PixiJS is ready to make your project a success, no matter what your needs.

... Easy to Deploy

Flash required the player. Unity requires an installer or app store. PixiJS requires... a browser. Deploying PixiJS on the web is exactly like deploying a web site. That's all it is - JavaScript + images + audio, like you've done a hundred times. Your users simply visit a URL, and your game or other content is ready to run. But it doesn't stop at the web. If you

want to deploy a mobile app, wrap your PixiJS code in Cordova. Want to deploy a standalone desktop program? Build an Electron wrapper, and you're ready to rock.

What PixiJS Is Not

While PixiJS can do many things, there are things it can't do, or that require additional tools to accomplish. Newcomers to PixiJS often struggle to identify which tasks PixiJS can solve, and which require outside solutions. If you're about to start a project, it can be helpful to know if PixiJS is a good fit for your needs. The following list is obviously incomplete - PixiJS is also not, for example, a duck - but it includes many common tasks or features that you might expect us to support.

PixiJS Is Not ... A Framework

PixiJS is a rendering engine, and it supports additional features such as interaction management that are commonly needed when using a render engine. But it is not a framework like Unity or Phaser. Frameworks are designed to do all the things you'd need to do when building a game - user settings management, music playback, object scripting, art pipeline management... the list goes on. PixiJS is designed to do one thing really well - render graphical content. This lets us focus on keeping up with new technology, and makes downloading PixiJS blazingly fast.

... A 3D Renderer

PixiJS is built for 2D. Platformers, adventure games, interactive ads, custom data visualization... all good. But if you want to render 3D models, you might want to check out babylon.js or three.js.

... A Mobile App

If you're looking to build mobile games, you can do it with PixiJS, but you'll need to use a deployment system like Apache Cordova if you want access to native bindings. We don't provide access to the camera, location services, notifications, etc.

... A UI Library

Building a truly generic UI system is a huge challenge, as anyone who has worked with Unity's UI tools can attest. We've chosen to avoid the complexity to stay true to our core focus on speed. While you can certainly build your own UI using PixiJS's scene graph and interaction manager, we don't ship with a UI library out of the box.

... A Data Store

There are many techniques and technologies that you can use to store settings, scores, and other data. Cookies, Web Storage, server-based storage... there are many solutions, each with advantages and disadvantages. You can use any of them with PixiJS, but we don't provide tools to do so.

... An Audio Library

At least, not out of the box. Again, web audio technology is a constantly evolving challenge, with constantly changing rules and requirements across many browsers. There are a number of dedicated web audio libraries (such as Howler.js that can be used with PixiJS to play sound effects and music. Alternatively, the PixiJS Sound plugin is designed to work well with PixiJS.

... A Development Environment

There are a number of tools that are useful for building 2D art and games that you might expect to be a part of PixiJS, but we're a rendering engine, not a development environment. Packing sprite sheets, processing images, building mipmaps or Retina-ready sprites - there are great standalone tools for this type of tooling. Where appropriate throughout the guides, we'll point you to tools that may be useful.

So Is PixiJS Right For Me?

Only you know! If you're looking for a tightly focused, fast and efficient rendering engine for your next web-based project, PixiJS is likely a great fit.
If you need a full game development framework, with native bindings and a rich UI library, you may want to explore other options.
Or you may not. It can be faster and easier to build just the subset of a full framework that your project needs than it can be to digest a monolithic API with bells and whistles you don't need. There are hundreds of complex, rich games and visual projects that use PixiJS for rendering, with plugins or custom code to add the UI and sound effects. There are benefits to both approaches. Regardless, we hope you have a better feel for what PixiJS can (and cannot!) offer your project.

Getting Started

In this section we're going to build the simplest possible PixiJS application. In doing so, we'll walk through the basics of how to build and serve the code.

Advanced Users

A quick note before we start: this guide is aimed at beginning PixiJS developers who have minimal experience developing JavaScript-based applications. If you are a coding veteran, you may find that the level of detail here is not helpful. If that's the case, you may want to skim this guide, then jump into how to work with PixiJS and packers like webpack and npm.

A Note About JavaScript

One final note. The JavaScript universe is currently in transition from old-school JavaScript (ES5) to the newer ES6 flavor:

```
// ES5
var x = 5;
setTimeout(function() { alert(x); }, 1000);
// ES6
const x = 5;
setTimeout(() => alert(x), 1000);
```

ES6 brings a number of major advantages in terms of clearer syntax, better variable scoping, native class support, etc. By now, all major browsers support it. Given this, our examples in these guides will use ES6. This doesn't mean you can't use PixiJS with ES5 programs! Just mentally substitute "var" for "let/const", expand the shorter function-passing syntax, and everything will run just fine.

Components of a PixiJS Application

OK! With those notes out of the way, let's get started. There are only a few steps required to write a PixiJS application:

- Create an HTML file
- Serve the file with a web server
- Load the PixiJS library
- Create an Application
- Add the generated view to the DOM
- Add an image to the stage
- Write an update loop

Let's walk through them together.

The HTML File

PixiJS is a JavaScript library that runs in a web page. So the first thing we're going to need is some HTML in a file. In a real PixiJS application, you might want to embed your display within a complex existing page, or you might want your display area to fill the whole page. For this demo, we'll build an empty page to start:

```
<!doctype html>
<html>
  <head>
  </head>
  <body>
    <h1>Hello PixiJS</h1>
  </body>
</html>
```

Create a new folder named `pixi-test`, then copy and paste this HTML into a new file in the `pixi-test` folder named `index.html`.

Serving the File

You will need to run a web server to develop locally with PixiJS. Web browsers prevent loading local files (such as images and audio files) on locally loaded web pages. If you just double-click your new HTML file, you'll get an error when you try to add a sprite to the PixiJS stage.

Running a web server sounds complex and difficult, but it turns out there are a number of simple web servers that will serve this purpose. For this guide, we're going to be working with Mongoose, but you could just as easily use XAMPP or the http-server Node.js package to serve your files.

To start serving your page with Mongoose, go to the Mongoose download page and download the free server for your operating system. Mongoose defaults to serving the files in the folder it's run in, so copy the downloaded executable into the folder you created in the prior step (`pixi-test`). Double-click the executable, tell your operating system that you trust the file to run, and you'll have a running web server, serving your new folder. Test that everything is working by opening your browser of choice and entering `http://127.0.0.1:8080` in the location bar. (Mongoose by default serves files on port 8080.) You should see "Hello PixiJS" and nothing else. If you get an error at this step, it means you didn't name your file `index.html` or you mis-configured your web server.

Loading PixiJS

OK, so we have a web page, and we're serving it. But it's empty. The next step is to actually load the PixiJS library. If we were building a real application, we'd want to download a target version of PixiJS from the Pixi Github repo so that our version wouldn't change on us. But for this sample application, we'll just use the CDN version of PixiJS. Add this line to the `<head>` section of your `index.html` file:

```
<script src="https://pixijs.download/release/pixi.js"></script>
```

This will include a *non-minified* version of the latest version of PixiJS when your page loads, ready to be used. We use the non-minified version because we're in development. In production, you'd want to use `pixi.min.js` instead, which is compressed for faster download and excludes assertions and deprecation warnings that can help when building your project, but take longer to download and run.

Creating an Application

Loading the library doesn't do much good if we don't *use* it, so the next step is to start up PixiJS. Start by replacing the line `<h1>Hello PixiJS</h1>` with a script tag like so:

```
<script type="module">
  const app = new PIXI.Application();
  await app.init({ width: 640, height: 360 });
</script>
```

What we're doing here is adding a JavaScript code block, and in that block creating a new PIXI.Application instance. Application is a helper class that simplifies working with PixiJS. It creates the renderer, creates the stage, and starts a ticker for updating. In production, you'll almost certainly want to do these steps yourself for added customization and control - we'll cover doing so in a later guide. For now, the Application class is a perfect way to start playing with PixiJS without worrying about the details. The `Application` class also has a method `init` that will initialize the application with the given options. This method is asynchronous, so we use the `await` keyword to start our logic after the promise has completed. This is because PixiJS uses WebGPU or WebGL under the hood, and the former API asynchronous.

Adding the Canvas to the DOM

When the PIXI.Application class creates the renderer, it builds a Canvas element that it will render *to*. In order to see what we draw with PixiJS, we need to add this Canvas element to the web page's DOM. Append the following line to your page's script block:

```
  document.body.appendChild(app.canvas);
```

This takes the canvas created by the application (the Canvas element) and adds it to the body of your page.

Creating a Sprite

So far all we've been doing is prep work. We haven't actually told PixiJS to draw anything. Let's fix that by adding an image to be displayed.

There are a number of ways to draw images in PixiJS, but the simplest is by using a Sprite. We'll get into the details of how the scene graph works in a later guide, but for now all you need to know is that PixiJS renders a hierarchy of Containers. A Sprite is a type of Container that wraps a loaded image resource to allow drawing it, scaling it, rotating it, and so forth.

Before PixiJS can render an image, it needs to be loaded. Just like in any web page, image loading happens asynchronously. We'll talk a lot more about resource loading in later guides. For now, we can use a helper method on the PIXI.Sprite class to handle the image loading for us:

```
  // load the PNG asynchronously
  await PIXI.Assets.load('sample.png');
  let sprite = PIXI.Sprite.from('sample.png');
```

Download the sample PNG here, and save it into your `pixi-test` directory next to your `index.html`.

Adding the Sprite to the Stage

Finally, we need to add our new sprite to the stage. The stage is simply a Container that is the root of the scene graph. Every child of the stage container will be rendered every frame. By adding our sprite to the stage, we tell PixiJS's renderer we want to draw it.

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

Writing an Update Loop

While you *can* use PixiJS for static content, for most projects you'll want to add animation. Our sample app is actually cranking away, rendering the same sprite in the same place multiple times a second. All we have to do to make the image move is to update its attributes once per frame. To do this, we want to hook into the application's *ticker*. A ticker is a PixiJS object that runs one or more callbacks each frame. Doing so is surprisingly easy. Add the following to the end of your script block:

```
  // Add a variable to count up the seconds our demo has been
running
  let elapsed = 0.0;
  // Tell our application's ticker to run a new callback every
frame, passing
```

```
  // in the amount of time that has passed since the last tick
  app.ticker.add((ticker) => {
    // Add the time to our total elapsed time
    elapsed += ticker.deltaTime;
    // Update the sprite's X position based on the cosine of our
elapsed time.  We divide
    // by 50 to slow the animation down a bit...
    sprite.x = 100.0 + Math.cos(elapsed/50.0) * 100.0;
  });
```

All you need to do is to call `app.ticker.add(...)`, pass it a callback function, and then update your scene in that function. It will get called every frame, and you can move, rotate etc. whatever you'd like to drive your project's animations.

Putting It All Together

That's it! The simplest PixiJS project!

Here's the whole thing in one place. Check your file and make sure it matches if you're getting errors.

```
<!doctype html>
<html>
  <head>
    <script src="https://pixijs.download/release/pixi.min.js">
</script>
  </head>
  <body>
    <script type="module">
      // Create the application helper and add its render target to
the page
      const app = new PIXI.Application();
      await app.init({ width: 640, height: 360 })
      document.body.appendChild(app.canvas);

      // Create the sprite and add it to the stage
      await PIXI.Assets.load('sample.png');
      let sprite = PIXI.Sprite.from('sample.png');
      app.stage.addChild(sprite);

      // Add a ticker callback to move the sprite back and forth
```

```
      let elapsed = 0.0;
      app.ticker.add((ticker) => {
        elapsed += ticker.deltaTime;
        sprite.x = 100.0 + Math.cos(elapsed/50.0) * 100.0;
      });
    </script>
  </body>
</html>
```

Once you have things working, the next thing to do is to read through the rest of the Basics guides to dig into how all this works in much greater depth.

Architecture Overview

OK, now that you've gotten a feel for how easy it is to build a PixiJS application, let's get into the specifics. For the rest of the Basics section, we're going to work from the high level down to the details. We'll start with an overview of how PixiJS is put together.

The Code

Before we get into how the code is laid out, let's talk about where it lives. PixiJS is an open source product hosted on GitHub. Like any GitHub repo, you can browse and download the raw source files for each PixiJS class, as well as search existing issues & bugs, and even submit your own. PixiJS is written in a JavaScript variant called TypeScript, which enables type-checking in JavaScript via a pre-compile step.

The Components

Here's a list of the major components that make up PixiJS. Note that this list isn't exhaustive. Additionally, don't worry too much about how each component works. The goal here is to give you a feel for what's under the hood as we start exploring the engine.

Major Components

Component	Description
Renderer	The core of the PixiJS system is the renderer, which displays the scene graph and draws it

	to the screen. PixiJS will automatically determine whether to provide you the WebGPU or WebGL renderer under the hood.
Container	Main scene object which creates a scene graph: the tree of renderable objects to be displayed, such as sprites, graphics and text. See Scene Graph for more details.
Assets	The Asset system provides tools for asynchronously loading resources such as images and audio files.
Ticker	Tickers provide periodic callbacks based on a clock. Your game update logic will generally be run in response to a tick once per frame. You can have multiple tickers in use at one time.

Application	The Application is a simple helper that wraps a Loader, Ticker and Renderer into a single, convenient easy-to-use object. Great for getting started quickly, prototyping and building simple projects.
Events	PixiJS supports pointer-based interaction - making objects clickable, firing hover events, etc.
Accessibility	Woven through our display system is a rich set of tools for enabling keyboard and screen-reader accessibility.

Render Loop

Now that you understand the major parts of the system, let's look at how these parts work together to get your project onto the screen. Unlike a web page, PixiJS is constantly updating and re-drawing itself, over and over. You update your objects, then PixiJS renders them to the screen, then the process repeats. We call this cycle the render loop. The majority of any PixiJS project is contained in this update + render cycle. You code the updates, PixiJS handles the rendering.
Let's walk through what happens each frame of the render loop. There are three main steps.

Running Ticker Callbacks

The first step is to calculate how much time has elapsed since the last frame, and then call the Application object's ticker callbacks with that time delta. This allows your project's code to animate and update the sprites, etc. on the stage in preparation for rendering.

Updating the Scene Graph

We'll talk a *lot* more about what a scene graph is and what it's made of in the next guide, but for now, all you need to know is that it contains the things you're drawing - sprites, text, etc. - and that these objects are in a tree-like hierarchy. After you've updated your game objects by moving, rotating and so forth, PixiJS needs to calculate the new positions and state of every object in the scene, before it can start drawing.

Rendering the Scene Graph

Now that our game's state has been updated, it's time to draw it to the screen. The rendering system starts with the root of the scene graph (`app.stage`), and starts rendering each object and its children, until all objects have been drawn. No culling or other cleverness is built into this process. If you have lots of objects outside of the visible portion of the stage, you'll want to investigate disabling them as an optimization.

Frame Rates

A note about frame rates. The render loop can't be run infinitely fast - drawing things to the screen takes time. In addition, it's not generally useful to have a frame updated more than once per screen update (commonly 60fps, but newer monitors can support 144fps and up). Finally, PixiJS runs in the context of a web browser like Chrome or Firefox. The browser itself has to balance the needs of various internal operations with servicing any open tabs. All this to say, determining when to draw a frame is a complex issue.
In cases where you want to adjust that behavior, you can set the `minFPS` and `maxFPS` attributes on a Ticker to give PixiJS hints as to the range of tick speeds you want to support. Just be aware that due to the complex environment, your project cannot *guarantee* a given FPS. Use the passed `ticker.deltaTime` value in your ticker callbacks to scale any animations to ensure smooth playback.

Custom Render Loops

What we've just covered is the default render loop provided out of the box by the Application helper class. There are many other ways of creating a render loop that may be helpful for advanced users looking to solve a given problem. While you're prototyping and learning PixiJS, sticking with the Application's provided system is the recommended approach.

Scene Graph

Every frame, PixiJS is updating and then rendering the scene graph. Let's talk about what's in the scene graph, and how it impacts how you develop your project. If you've built games before, this should all sound very familiar, but if you're coming from HTML and the DOM, it's worth understanding before we get into specific types of objects you can render.

The Scene Graph Is a Tree

The scene graph's root node is a container maintained by the application, and referenced with `app.stage`. When you add a sprite or other renderable object as a child to the stage, it's added to the scene graph and will be rendered and interactable.
PixiJS `Containers` can also have children, and so as you build more complex scenes, you will end up with a tree of parent-child relationships, rooted at the app's stage.
(A helpful tool for exploring your project is the Pixi.js devtools plugin for Chrome, which allows you to view and manipulate the scene graph in real time as it's running!)

Parents and Children

When a parent moves, its children move as well. When a parent is rotated, its children are rotated too. Hide a parent, and the children will also be hidden. If you have a game object that's made up of multiple sprites, you can collect them under a container to treat them as a single object in the world, moving and rotating as one.
Each frame, PixiJS runs through the scene graph from the root down through all the children to the leaves to calculate each object's final position, rotation, visibility, transparency, etc. If a parent's alpha is set to 0.5 (making it 50% transparent), all its children will start at 50% transparent as well. If a child is then set to 0.5 alpha, it won't be 50% transparent, it will be 0.5 × 0.5 = 0.25 alpha, or 75% transparent. Similarly, an object's position is relative to its parent, so if a parent is set to an x position of 50 pixels, and the child is set to an x position of 100 pixels, it will be drawn at a screen offset of 150 pixels, or 50 + 100.
Here's an example. We'll create three sprites, each a child of the last, and animate their position, rotation, scale and alpha. Even though each sprite's properties are set to the same values, the parent-child chain amplifies each change:

```
// Create the application helper and add its render target to the
page
const app = new Application();
await app.init({ width: 640, height: 360 })
document.body.appendChild(app.canvas);
```

```
// Add a container to center our sprite stack on the page
const container = new Container({
  x:app.screen.width / 2,
  y:app.screen.height / 2
});

app.stage.addChild(container);

// load the texture
await Assets.load('assets/images/sample.png');

// Create the 3 sprites, each a child of the last
const sprites = [];
let parent = container;
for (let i = 0; i < 3; i++) {
  let wrapper = new Container();
  let sprite = Sprite.from('assets/images/sample.png');
  sprite.anchor.set(0.5);
  wrapper.addChild(sprite);
  parent.addChild(wrapper);
  sprites.push(wrapper);
  parent = wrapper;
}

// Set all sprite's properties to the same value, animated over
time
let elapsed = 0.0;
app.ticker.add((delta) => {
  elapsed += delta.deltaTime / 60;
  const amount = Math.sin(elapsed);
  const scale = 1.0 + 0.25 * amount;
  const alpha = 0.75 + 0.25 * amount;
  const angle = 40 * amount;
  const x = 75 * amount;
  for (let i = 0; i < sprites.length; i++) {
    const sprite = sprites[i];
    sprite.scale.set(scale);
    sprite.alpha = alpha;
```

```
    sprite.angle = angle;
    sprite.x = x;
  }
});
```

The cumulative translation, rotation, scale and skew of any given node in the scene graph is stored in the object's `worldTransform` property. Similarly, the cumulative alpha value is stored in the `worldAlpha` property.

Render Order

So we have a tree of things to draw. Who gets drawn first?
PixiJS renders the tree from the root down. At each level, the current object is rendered, then each child is rendered in order of insertion. So the second child is rendered on top of the first child, and the third over the second.
Check out this example, with two parent objects A & D, and two children B & C under A:

```
// Create the application helper and add its render target to the
page
const app = new Application();
await app.init({ width: 640, height: 360 })
document.body.appendChild(app.canvas);

// Label showing scene graph hierarchy
const label = new Text({
  text:'Scene Graph:\n\napp.stage\n  ┗ A\n     ┗ B\n     ┗ C\n  ┗
D',
  style:{fill: '#ffffff'},
  position: {x: 300, y: 100}
});

app.stage.addChild(label);

// Helper function to create a block of color with a letter
const letters = [];
function addLetter(letter, parent, color, pos) {
  const bg = new Sprite(Texture.WHITE);
  bg.width = 100;
  bg.height = 100;
```

```
  bg.tint = color;

  const text = new Text({
    text:letter,
    style:{fill: "#ffffff"}
  });

  text.anchor.set(0.5);
  text.position = {x: 50, y: 50};

  const container = new Container();
  container.position = pos;
  container.visible = false;
  container.addChild(bg, text);
  parent.addChild(container);

  letters.push(container);
  return container;
}

// Define 4 letters
let a = addLetter('A', app.stage, 0xff0000, {x: 100, y: 100});
let b = addLetter('B', a,         0x00ff00, {x: 20,  y: 20});
let c = addLetter('C', a,         0x0000ff, {x: 20,  y: 40});
let d = addLetter('D', app.stage, 0xff8800, {x: 140, y: 100});

// Display them over time, in order
let elapsed = 0.0;
app.ticker.add((ticker) => {
  elapsed += ticker.deltaTime / 60.0;
  if (elapsed >= letters.length) { elapsed = 0.0; }
  for (let i = 0; i < letters.length; i ++) {
    letters[i].visible = elapsed >= i;
  }
});
```

If you'd like to re-order a child object, you can use `setChildIndex()`. To add a child at a given point in a parent's list, use `addChildAt()`. Finally, you can enable automatic sorting of an object's children using the `sortableChildren` option combined with setting the `zIndex` property on each child.

RenderGroups

As you delve deeper into PixiJS, you'll encounter a powerful feature known as Render Groups. Think of Render Groups as specialized containers within your scene graph that act like mini scene graphs themselves. Here's what you need to know to effectively use Render Groups in your projects. For more info check out the RenderGroups overview

Culling

If you're building a project where a large proportion of your scene objects are off-screen (say, a side-scrolling game), you will want to *cull* those objects. Culling is the process of evaluating if an object (or its children!) is on the screen, and if not, turning off rendering for it. If you don't cull off-screen objects, the renderer will still draw them, even though none of their pixels end up on the screen.

PixiJS doesn't provide built-in support for viewport culling, but you can find 3rd party plugins that might fit your needs. Alternately, if you'd like to build your own culling system, simply run your objects during each tick and set `renderable` to false on any object that doesn't need to be drawn.

Local vs Global Coordinates

If you add a sprite to the stage, by default it will show up in the top left corner of the screen. That's the origin of the global coordinate space used by PixiJS. If all your objects were children of the stage, that's the only coordinates you'd need to worry about. But once you introduce containers and children, things get more complicated. A child object at [50, 100] is 50 pixels right and 100 pixels down *from its parent*.

We call these two coordinate systems "global" and "local" coordinates. When you use `position.set(x, y)` on an object, you're always working in local coordinates, relative to the object's parent.

The problem is, there are many times when you want to know the global position of an object. For example, if you want to cull offscreen objects to save render time, you need to know if a given child is outside the view rectangle.

To convert from local to global coordinates, you use the `toGlobal()` function. Here's a sample usage:

```
// Get the global position of an object, relative to the top-left
of the screen
```

```
let globalPos = obj.toGlobal(new Point(0,0));
```

This snippet will set `globalPos` to be the global coordinates for the child object, relative to [0, 0] in the global coordinate system.

Global vs Screen Coordinates

When your project is working with the host operating system or browser, there is a third coordinate system that comes into play - "screen" coordinates (aka "viewport" coordinates). Screen coordinates represent position relative to the top-left of the canvas element that PixiJS is rendering into. Things like the DOM and native mouse click events work in screen space.
Now, in many cases, screen space is equivalent to world space. This is the case if the size of the canvas is the same as the size of the render view specified when you create you `Application`. By default, this will be the case - you'll create for example an 800×600 application window and add it to your HTML page, and it will stay that size. 100 pixels in world coordinates will equal 100 pixels in screen space. BUT! It is common to stretch the rendered view to have it fill the screen, or to render at a lower resolution and up-scale for speed. In that case, the screen size of the canvas element will change (e.g. via CSS), but the underlying render view will *not*, resulting in a mis-match between world coordinates and screen coordinates.

SVG's

Overview

PixiJS provides powerful support for rendering SVGs, allowing developers to integrate scalable vector graphics seamlessly into their projects. This guide explores different ways to use SVGs in PixiJS, covering real-time rendering, performance optimizations, and potential pitfalls.

Why Use SVGs?

SVGs have several advantages over raster images like PNGs:
✅ **Smaller File Sizes** – SVGs can be significantly smaller than PNGs, especially for large but simple shapes. A high-resolution PNG may be several megabytes, while an equivalent SVG could be just a few kilobytes.
✅ **Scalability** – SVGs scale without losing quality, making them perfect for responsive applications and UI elements.

✅ **Editable After Rendering** – Unlike textures, SVGs rendered via Graphics can be modified dynamically (e.g., changing stroke colors, modifying shapes).
✅ **Efficient for Simple Graphics** – If the graphic consists of basic shapes and paths, SVGs can be rendered efficiently as vector graphics.
However, SVGs can also be computationally expensive to parse, particularly for intricate illustrations with many paths or effects.

Ways to Render SVGs in PixiJS

PixiJS offers two primary ways to render SVGs:

1. **As a Texture** – Converts the SVG into a texture for rendering as a sprite.
2. **As a Graphics Object** – Parses the SVG and renders it as vector geometry.

Each method has its advantages and use cases, which we will explore below.

---

1. Rendering SVGs as Textures

Overview

SVGs can be loaded as textures and used within Sprites. This method is efficient but does not retain the scalability of vector graphics.

Example

```
const svgTexture = await Assets.load('tiger.svg');
const mySprite = new Sprite(svgTexture);
```

EditorPreviewBoth

Scaling Textures

You can specify a resolution when loading an SVG as a texture to control its size: This does increase memory usage, but it be of a higher fidelity.

```
const svgTexture = await Assets.load('path/to.svg', {
    resolution: 4 // will be 4 times as big!
});
const mySprite = new Sprite(svgTexture);
```

EditorPreviewBoth

This ensures the texture appears at the correct size and resolution.

Pros & Cons

✅ **Fast to render** (rendered as a quad, not geometry)
✅ **Good for static images**
✅ **Supports resolution scaling for precise sizing**
✅ **Ideal for complex SVGs that do not need crisp vector scaling** (e.g., UI components with fixed dimensions)
❌ **Does not scale cleanly** (scaling may result in pixelation)
❌ **Less flexibility** (cannot modify the shape dynamically) ❌ **Texture Size Limit** A texture can only be up to 4096×4096 pixels, so if you need to render a larger SVG, you will need to use the Graphics method.

Best Use Cases

- Background images
- Decorative elements
- Performance-critical applications where scaling isn't needed
- Complex SVGs that do not require crisp vector scaling (e.g., fixed-size UI components)

---

2. Rendering SVGs as Graphics

Overview

PixiJS can render SVGs as real scalable vector graphics using the `Graphics` class.

Example

```
const graphics = new Graphics()
   .svg('<svg width="100" height="100"><rect width="100"
height="100" fill="red"/></svg>');
```

If you want to use the same SVG multiple times, you can use `GraphicsContext` to share the parsed SVG data across multiple graphics objects, improving performance by parsing it once and reusing it.

```
const context = new GraphicsContext().svg('<svg width="100"
height="100"><rect width="100" height="100" fill="red"/></svg>');
```

```
const graphics1 = new Graphics(context);
const graphics2 = new Graphics(context);
```

Loading SVGs as Graphics

Instead of passing an SVG string directly, you can load an SVG file using PixiJS's `Assets.load` method. This will return a `GraphicsContext` object, which can be used to create multiple `Graphics` objects efficiently.

```
const svgContext = await Assets.load('path/to.svg', {
    parseAsGraphicsContext: true // If false, it returns a texture instead.
});
const myGraphics = new Graphics(svgContext);
```

Since it's loaded via `Assets.load`, it will be cached and reused, much like a texture.

Pros & Cons

✅ **Retains vector scalability** (no pixelation when zooming)
✅ **Modifiable after rendering** (change colors, strokes, etc.)
✅ **Efficient for simple graphics** ✅ **fast rendering if SVG structure does not change** (no need to reparse) ❌ **More expensive to parse** (complex SVGs can be slow to render)
❌ **Not ideal for static images**

Best Use Cases

- Icons and UI elements that need resizing
- A game world that needs to remain crisp as a player zooms in
- Interactive graphics where modifying the SVG dynamically is necessary

---

SVG Rendering Considerations

Supported Features

PixiJS supports most SVG features that can be rendered in a Canvas 2D context. Below is a list of common SVG features and their compatibility:

Feature	Supported
Basic Shapes (rect, circle, path, etc.)	✅

Gradients	✅
Stroke & Fill Styles	✅
Text Elements	❌
Filters (Blur, Drop Shadow, etc.)	❌
Clipping Paths	✅
Patterns	❌
Complex Paths & Curves	✅

Performance Considerations

- **Complex SVGs:** Large or intricate SVGs can slow down rendering start up due to high parsing costs. Use `GraphicsContext` to cache and reuse parsed data.
- **Vector vs. Texture:** If performance is a concern, consider using SVGs as textures instead of rendering them as geometry. However, keep in mind that textures take up more memory.
- **Real-Time Rendering:** Avoid rendering complex SVGs dynamically. Preload and reuse them wherever possible.

Best Practices & Gotchas

Best Practices

✅ **Use Graphics for scalable and dynamic SVGs**
✅ **Use Textures for performance-sensitive applications**
✅ **Use `GraphicsContext` to avoid redundant parsing**
✅ **Consider `resolution` when using textures to balance quality and memory**

Gotchas

⚠ **Large SVGs can be slow to parse** – Optimize SVGs before using them in PixiJS.
⚠ **Texture-based SVGs do not scale cleanly** – Use higher resolution if necessary.
⚠ **Not all SVG features are supported** – Complex filters and text elements may not work as expected.

By understanding how PixiJS processes SVGs, developers can make informed decisions on when to use `Graphics.svg()`, `GraphicsContext`, or SVG textures, balancing quality and performance for their specific use case.

Render Groups

Understanding RenderGroups in PixiJS

As you delve deeper into PixiJS, especially with version 8, you'll encounter a powerful feature known as RenderGroups. Think of RenderGroups as specialized containers within your scene graph that act like mini scene graphs themselves. Here's what you need to know to effectively use Render Groups in your projects:

What Are Render Groups?

Render Groups are essentially containers that PixiJS treats as self-contained scene graphs. When you assign parts of your scene to a Render Group, you're telling PixiJS to manage these objects together as a unit. This management includes monitoring for changes and preparing a set of render instructions specifically for the group. This is a powerful tool for optimizing your rendering process.

Why Use Render Groups?

The main advantage of using Render Groups lies in their optimization capabilities. They allow for certain calculations, like transformations (position, scale, rotation), tint, and alpha adjustments, to be offloaded to the GPU. This means that operations like moving or adjusting the Render Group can be done with minimal CPU impact, making your application more performance-efficient.

In practice, you're utilizing Render Groups even without explicit awareness. The root element you pass to the render function in PixiJS is automatically converted into a RenderGroup as this is where its render instructions will be stored. Though you also have the option to explicitly create additional RenderGroups as needed to further optimize your project.

This feature is particularly beneficial for:

- **Static Content:** For content that doesn't change often, a Render Group can significantly reduce the computational load on the CPU. In this case static refers to the scene graph structure, not that actual values of the PixiJS elements inside it (eg position, scale of things).
- **Distinct Scene Parts:** You can separate your scene into logical parts, such as the game world and the HUD (Heads-Up Display). Each part can be optimized individually, leading to overall better performance.

Examples

```
const myGameWorld = new Container({
  isRenderGroup:true
```

```
})

const myHud = new Container({
  isRenderGroup:true
})

scene.addChild(myGameWorld, myHud)

renderer.render(scene) // this action will actually convert the
scene to a render group under the hood
```

Check out the container example.

Best Practices

- **Don't Overuse:** While Render Groups are powerful, using too many can actually degrade performance. The goal is to find a balance that optimizes rendering without overwhelming the system with too many separate groups. Make sure to profile when using them. The majority of the time you won't need to use them at all!
- **Strategic Grouping:** Consider what parts of your scene change together and which parts remain static. Grouping dynamic elements separately from static elements can lead to performance gains.

By understanding and utilizing Render Groups, you can take full advantage of PixiJS's rendering capabilities, making your applications smoother and more efficient. This feature represents a powerful tool in the optimization toolkit offered by PixiJS, enabling developers to create rich, interactive scenes that run smoothly across different devices.

Cache As Texture

Using `cacheAsTexture` in PixiJS

The `cacheAsTexture` function in PixiJS is a powerful tool for optimizing rendering in your applications. By rendering a container and its children to a texture, `cacheAsTexture` can significantly improve performance for static or infrequently updated containers. Let's explore how to use it effectively, along with its benefits and considerations.

Note

`cacheAsTexture` is PixiJS v8's equivalent of the previous `cacheAsBitmap` functionality. If you're migrating from v7 or earlier, simply

replace `cacheAsBitmap` with `cacheAsTexture` in your code.

What Is `cacheAsTexture`?

When you set `container.cacheAsTexture()`, the container is rendered to a texture. Subsequent renders reuse this texture instead of rendering all the individual children of the container. This approach is particularly useful for containers with many static elements, as it reduces the rendering workload.

To update the texture after making changes to the container, call:

```
container.updateCacheTexture();
```

and to turn it off, call:

```
container.cacheAsTexture(false);
```

Basic Usage

Here's an example that demonstrates how to use `cacheAsTexture`:

```
import * as PIXI from 'pixi.js';

(async () =>
{
    // Create a new application
    const app = new Application();

    // Initialize the application
    await app.init({ background: '#1099bb', resizeTo: window });

    // Append the application canvas to the document body
    document.body.appendChild(app.canvas);

    // load sprite sheet..
    await
Assets.load('https://pixijs.com/assets/spritesheet/monsters.json');

    // holder to store aliens
    const aliens = [];
    const alienFrames = ['eggHead.png', 'flowerTop.png',
'helmlok.png', 'skully.png'];
```

```
    let count = 0;

    // create an empty container
    const alienContainer = new Container();

    alienContainer.x = 400;
    alienContainer.y = 300;

    app.stage.addChild(alienContainer);

    // add a bunch of aliens with textures from image paths
    for (let i = 0; i < 100; i++)
    {
        const frameName = alienFrames[i % 4];

        // create an alien using the frame name..
        const alien = Sprite.from(frameName);

        alien.tint = Math.random() * 0xffffff;

        alien.x = Math.random() * 800 - 400;
        alien.y = Math.random() * 600 - 300;
        alien.anchor.x = 0.5;
        alien.anchor.y = 0.5;
        aliens.push(alien);
        alienContainer.addChild(alien);
    }

    // this will cache the container and its children as a single
texture
    // so instead of drawing 100 sprites, it will draw a single
texture!
    alienContainer.cacheAsTexture()
})();
```

In this example, the `container` and its children are rendered to a single texture, reducing the rendering overhead when the scene is drawn.
Play around with the example here.

Advanced Usage

Instead of enabling cacheAsTexture with true, you can pass a configuration object which is very similar to texture source options.

```
container.cacheAsTexture({
    resolution: 2,
    antialias: true,
});
```

- `resolution` is the resolution of the texture. By default this is the same as you renderer or application.
- `antialias` is the antialias mode to use for the texture. Much like the resolution this defaults to the renderer or application antialias mode.

Benefits of `cacheAsTexture`

- **Performance Boost**: Rendering a complex container as a single texture avoids the need to process each child element individually during each frame.
- **Optimized for Static Content**: Ideal for containers with static or rarely updated children.

Advanced Details

- **Memory Tradeoff**: Each cached texture requires GPU memory.
  Using `cacheAsTexture` trades rendering speed for increased memory usage.
- **GPU Limitations**: If your container is too large (e.g., over 4096×4096 pixels), the texture may fail to cache, depending on GPU limitations.

How It Works Internally

Under the hood, `cacheAsTexture` converts the container into a render group and renders it to a texture. It uses the same texture cache mechanism as filters:

```
container.enableRenderGroup();
container.renderGroup.cacheAsTexture = true;
```

Once the texture is cached, updating it via `updateCacheTexture()` is efficient and incurs minimal overhead. Its as fast as rendering the container normally.

Best Practices

DO:

- **Use for Static Content**: Apply `cacheAsTexture` to containers with elements that don't change frequently, such as a UI panel with static decorations.
- **Leverage for Performance**: Use `cacheAsTexture` to render complex containers as a single texture, reducing the overhead of processing each child element individually every frame. This is especially useful for containers that contain expensive effects eg filters.
- **Switch of Antialiasing**: setting antialiasing to false can give a small performance boost, but the texture may look a bit more pixelated around its children's edges.
- **Resolution**: Do adjust the resolution based on your situation, if something is scaled down, you can use a lower resolution.If something is scaled up, you may want to use a higher resolution. But be aware that the higher the resolution the larger the texture and memory footprint.

DON'T:

- **Apply to Very Large Containers**: Avoid using `cacheAsTexture` on containers that are too large (e.g., over 4096×4096 pixels), as they may fail to cache due to GPU limitations. Instead, split them into smaller containers.
- **Overuse for Dynamic Content**: Flick `cacheAsTexture` on / off frequently on containers, as this results in constant re-caching, negating its benefits. Its better to Cache as texture when you once, and then use `updateCacheTexture` to update it.
- **Apply to Sparse Content**: Do not use `cacheAsTexture` for containers with very few elements or sparse content, as the performance improvement will be negligible.
- **Ignore Memory Impact**: Be cautious of GPU memory usage. Each cached texture consumes memory, so overusing `cacheAsTexture` can lead to resource constraints.

Gotchas

- **Rendering Depends on Scene Visibility**: The cache updates only when the containing scene is rendered. Modifying the layout after setting `cacheAsTexture` but before rendering your scene will be reflected in the cache.
- **Containers are rendered with no transform**: Cached items are rendered at their actual size, ignoring transforms like scaling. For instance, an item scaled down by 50%, its texture will be cached at 100% size and then scaled down by the scene.
- **Caching and Filters**: Filters may not behave as expected with `cacheAsTexture`. To cache the filter effect, wrap the item in a parent container and apply `cacheAsTexture` to the parent.

- **Reusing the texture**: If you want to create a new texture based on the container, its better to use `const texture = renderer.generateTexture(container)` and share that amongst you objects!

By understanding and applying `cacheAsTexture` strategically, you can significantly enhance the rendering performance of your PixiJS projects. Happy coding!

Mixing PixiJS and Three.js

In many projects, developers aim to harness the strengths of both 3D and 2D graphics. Combining the advanced 3D rendering capabilities of Three.js with the speed and versatility of PixiJS for 2D can result in a powerful, seamless experience. Together, these technologies create opportunities for dynamic and visually compelling applications. Lets see how to do this.

NOTE

This guide assumes PixiJS will be used as the top layer to deliver UI over a 3D scene rendered by Three.js. However, developers can render either in any order, as many times as needed. This flexibility allows for creative and dynamic applications.

What You'll Learn

- Setting up PixiJS and Three.js to share a single WebGL context.
- Using `resetState` to manage renderer states.
- Avoiding common pitfalls when working with multiple renderers.

Setting Up

Step 1: Initialize Three.js Renderer and Scene

Three.js will handle the 3D rendering the creation of the dom element and context.

```
const WIDTH = window.innerWidth;
const HEIGHT = window.innerHeight;

const threeRenderer = new THREE.WebGLRenderer({
    antialias: true,
    stencil: true // so masks work in pixijs
});

threeRenderer.setSize(WIDTH, HEIGHT);
threeRenderer.setClearColor(0xdddddd, 1);
document.body.appendChild(threeRenderer.domElement);
```

```
const scene = new THREE.Scene();
const camera = new THREE.PerspectiveCamera(70, WIDTH / HEIGHT);
camera.position.z = 50;
scene.add(camera);

const boxGeometry = new THREE.BoxGeometry(10, 10, 10);
const basicMaterial = new THREE.MeshBasicMaterial({ color: 0x0095dd
});
const cube = new THREE.Mesh(boxGeometry, basicMaterial);
cube.rotation.set(0.4, 0.2, 0);
scene.add(cube);
```

NOTE

We used the dom element and context created by the three.js renderer to pass to the pixijs renderer. This was the simplest way to ensure that the two renderers were using the same WebGL context. You could have done it the other way round if you wanted to.

Step 2: Initialize PixiJS Renderer and Stage

PixiJS will handle the 2D overlay.

```
const pixiRenderer = new PIXI.WebGLRenderer();

await pixiRenderer.init({
    context: threeRenderer.getContext(),
    width: WIDTH,
    height: HEIGHT,
    clearBeforeRender: false, // Prevent PixiJS from clearing the
Three.js render
});

const stage = new PIXI.Container();
const amazingUI = new PIXI.Graphics()
    .roundRect(20, 80, 100, 100, 5)
    .roundRect(220, 80, 100, 100, 5)
    .fill(0xffff00);

stage.addChild(amazingUI);
```

Rendering Loop

To ensure smooth transitions between the renderers, reset their states before each render:

```
function render() {
    // Render the Three.js scene
    threeRenderer.resetState();
    threeRenderer.render(scene, camera);

    // Render the PixiJS stage
    pixiRenderer.resetState();
    pixiRenderer.render({ container: stage });

    requestAnimationFrame(render);
}

requestAnimationFrame(render);
```

Example: Combining 3D and 2D Elements

Here's the complete example integrating PixiJS and Three.js:

EditorPreviewBoth

Gotchas

- **Enable Stencil Buffers:**
  - When creating the Three.js renderer, ensure `stencil` is set to `true`. This allows PixiJS masks to work correctly.
- **Keep Dimensions in Sync:**
  - Ensure both renderers use the same `width` and `height` to avoid visual mismatches—so be careful when resizing one, you need to resize the other!
- **Pass the WebGL Context:**
  - Pass the WebGL context from Three.js to PixiJS during initialization using `pixiRenderer.init({ context: threeRenderer.getContext() });`.
- **Disable Clear Before Render:**

    - Set `clearBeforeRender: false` when initializing the PixiJS renderer. This prevents PixiJS from clearing the Three.js content that was rendered before it.
    - Alternatively you can set `clear: false` in the `pixiRenderer.render()` call. eg `pixiRenderer.render({ container: stage, clear: false });`.
- **Manage Render Order:**
    - In this example, Three.js is rendered first, followed by PixiJS for UI layers. However, this order is flexible. You can render pixi → three → pixi is you want, just make sure you reset the state when switching renderer.
- **Separate Resources:**
    - Remember that resources like textures are not shared between PixiJS and Three.js. A PixiJS texture cannot be directly used as a Three.js texture and vice versa.

Conclusion

Mixing PixiJS and Three.js can be a powerful way to create dynamic and visually appealing applications. By carefully managing the rendering loop and states, you can achieve seamless transitions between 3D and 2D layers. This approach allows you to leverage the strengths of both technologies, creating applications that are both visually stunning and performant.
This technique can be used with other renderers too - as long as they have their own way of resetting their state (which the main ones do) you can mix them. Popular 3D engines like Babylon.js and PlayCanvas both support state management through their respective APIs, making them compatible with this mixing approach. This gives you the flexibility to choose the 3D engine that best suits your needs while still leveraging PixiJS's powerful 2D capabilities.

Render Layers

PixiJS Layer API Guide

The PixiJS Layer API provides a powerful way to control the **rendering order** of objects independently of their **logical parent-child relationships** in the scene graph. With RenderLayers, you can decouple how objects are transformed (via their logical parent) from how they are visually drawn on the screen.
Using RenderLayers ensures these elements are visually prioritized while maintaining logical parent-child relationships. Examples include:

- A character with a health bar: Ensure the health bar always appears on top of the world, even if the character moves behind an object.
- UI elements like score counters or notifications: Keep them visible regardless of the game world's complexity.
- Highlighting Elements in Tutorials: Imagine a tutorial where you need to push back most game elements while highlighting a specific object. RenderLayers can split these visually. The highlighted object can be placed in a foreground layer to be rendered above a push back layer.

This guide explains the key concepts, provides practical examples, and highlights common gotchas to help you use the Layer API effectively.

Key Concepts

1. **Independent Rendering Order**:
   - RenderLayers allow control of the draw order independently of the logical hierarchy, ensuring objects are rendered in the desired order.
2. **Logical Parenting Stays Intact**:
   - Objects maintain transformations (e.g., position, scale, rotation) from their logical parent, even when attached to RenderLayers.
3. **Explicit Object Management**:
   - Objects must be manually reassigned to a layer after being removed from the scene graph or layer, ensuring deliberate control over rendering.
4. **Dynamic Sorting**:
   - Within layers, objects can be dynamically reordered using `zIndex` and `sortChildren` for fine-grained control of rendering order.

Basic API Usage

First lets create two items that we want to render, red guy and blue guy.

```
const redGuy = new PIXI.Sprite('red guy');
redGuy.tint = 0xff0000;

const blueGuy = new PIXI.Sprite('blue guy');
blueGuy.tint = 0x0000ff;

stage.addChild(redGuy, blueGuy);
```

Now we know that red guy will be rendered first, then blue guy. Now in this simple example you could get away with just sorting the `zIndex` of the red guy and blue guy to help reorder.

But this is a guide about render layers, so lets create one of those.

Use `renderLayer.attach` to assign an object to a layer. This overrides the object's default render order defined by its logical parent.

```
// a layer..
const layer = new RenderLayer();
stage.addChild(layer);
layer.attach(redGuy);
```

So now our scene graph order is:

```
|- stage
    |-- redGuy
    |-- blueGuy
    |-- layer
```

And our render order is:

```
|- stage
    |-- blueGuy
    |-- layer
        |-- redGuy
```

This happens because the layer is now the last child in the stage. Since the red guy is attached to the layer, it will be rendered at the layer's position in the scene graph. However, it still logically remains in the same place in the scene hierarchy.

3. Removing Objects from a Layer

Now let's remove the red guy from the layer. To stop an object from being rendered in a layer, use `removeFromLayer`. Once removed from the layer, its still going to be in the scene graph, and will be rendered in its scene graph order.

```
layer.detach(redGuy); //  Stop rendering the rect via the layer
```

Removing an object from its logical parent (`removeChild`) automatically removes it from the layer.

```
stage.removeChild(redGuy); // if the red guy was removed from the stage, it will also be removed from the layer
```

However, if you remove the red guy from the stage and then add it back to the stage, it will not be added to the layer again.

```
// add red guy to his original position
stage.addChildAt(redGuy, 0);
```

You will need to reattach it to the layer yourself.

```
layer.attach(redGuy); // re attach it to the layer again!
```

This may seem like a pain, but it's actually a good thing. It means that you have full control over the render order of the object, and you can change it at any time. It also means you can't accidentally add an object to a container and have it automatically re-attach to a layer that may or may not still be around - it would be quite confusing and lead to some very hard to debug bugs!

5. Layer Position in Scene Graph

The layer's position in the scene graph determines its render priority relative to other layers and objects.

```
// reparent the layer to render first in the stage
stage.addChildAt(layer, 0);
```

Complete Example

Here's a real-world example that shows how to use RenderLayers to set ap player ui on top of the world.

Gotchas and Things to Watch Out For

1. **Manual Reassignment**:
   - When an object is re-added to a logical parent, it does not automatically reassociate with its previous layer. Always reassign the object to the layer explicitly.
2. **Nested Children**:
   - If you remove a parent container, all its children are automatically removed from layers. Be cautious with complex hierarchies.
3. **Sorting Within Layers**:
   - Objects in a layer can be sorted dynamically using their `zIndex` property. This is useful for fine-grained control of render order.
4.
```
rect.zIndex = 10; // Higher values render later
layer.sortableChildren = true; // Enable sorting
layer.sortRenderLayerChildren(); // Apply the sorting
```

5. **Layer Overlap**:
   - If multiple layers overlap, their order in the scene graph determines the render priority. Ensure the layering logic aligns with your desired visual output.

Best Practices

1. **Group Strategically**: Minimize the number of layers to optimize performance.
2. **Use for Visual Clarity**: Reserve layers for objects that need explicit control over render order.
3. **Test Dynamic Changes**: Verify that adding, removing, or reassigning objects to layers behaves as expected in your specific scene setup.

By understanding and leveraging RenderLayers effectively, you can achieve precise control over your scene's visual presentation while maintaining a clean and logical hierarchy.

Assets

The Assets package

The Assets package is a modern replacement for the old `Loader` class. It is a promise-based resource management solution that will download, cache and parse your assets into something you can use. The downloads can be simultaneous and in the background, meaning faster startup times for your app, the cache ensures that you never download the

same asset twice and the extensible parser system allows you to easily extend and customize the process to your needs.

Getting started

`Assets` relies heavily on JavaScript Promises that all modern browsers support, however, if your target browser doesn't support promises you should look into polyfilling them.

Making our first Assets Promise

To quickly use the `Assets` instance, you just need to call `Assets.load` and pass in an asset. This will return a promise that when resolved will yield the value you seek. In this example, we will load a texture and then turn it into a sprite.

```
import { Application, Assets, Sprite } from 'pixi.js';

// Create a new application
const app = new Application();

// Initialize the application
await app.init({ background: '#1099bb', resizeTo: window });

// Append the application canvas to the document body
document.body.appendChild(app.canvas);

// Start loading right away and create a promise
const texturePromise =
Assets.load('https://pixijs.com/assets/bunny.png');

// When the promise resolves, we have the texture!
texturePromise.then((resolvedTexture) =>
{
    // create a new Sprite from the resolved loaded Texture
    const bunny = Sprite.from(resolvedTexture);

    // center the sprite's anchor point
    bunny.anchor.set(0.5);

    // move the sprite to the center of the screen
    bunny.x = app.screen.width / 2;
    bunny.y = app.screen.height / 2;
```

```
    app.stage.addChild(bunny);
});
```

One very important thing to keep in mind while using `Assets` is that all requests are cached and if the URL is the same, the promise returned will also be the same. To show it in code:

```
promise1 = Assets.load('bunny.png')
promise2 = Assets.load('bunny.png')
// promise1 === promise2
```

Out of the box, the following assets types can be loaded without the need for external plugins:

- Textures (`avif` , `webp` , `png` , `jpg` , `gif`)
- Sprite sheets (`json`)
- Bitmap fonts (`xml` , `fnt` , `txt`)
- Web fonts (`ttf` , `woff` , `woff2`)
- Json files (`json`)
- Text files (`txt`)

More types can be added fairly easily by creating additional loader parsers.

Working with unrecognizable URLs

With the basic syntax, asset types are recognized by their file extension - for instance `https://pixijs.com/assets/bunny.png` ends with `.png` so `Assets.load` can figure it should use the texture loader.
In some cases you may not have control over the URLs and you have to work with ambiguous URLs without recognizable extensions. In this situation, you can specify an explicit loader:

```
promise = Assets.load({
  src: 'https://example.com/ambiguous-file-name',
  loadParser: 'loadTextures'
})
```

Here are some of the `loader` values you can use:

- Textures: `loadTextures`
- Web fonts: `loadWebFont`

- Json files: `loadJson`
- Text files: `loadTxt`

Warning about solved promises

When an asset is downloaded, it is cached as a promise inside the `Assets` instance and if you try to download it again you will get a reference to the already resolved promise. However promise handlers `.then(...)` / `.catch(...)` / `.finally(...)` are always asynchronous, this means that even if a promise was already resolved the code below the `.then(...)` / `.catch(...)` / `.finally(...)` will execute before the code inside them. See this example:

```
console.log(1);
alreadyResolvedPromise.then(() => console.log(2));
console.log(3);

// Console output:
// 1
// 3
// 2
```

To learn more about why this happens you will need to learn about Microtasks, however, using async functions should mitigate this problem.

Using Async/Await

There is a way to work with promises that is more intuitive and easier to read: `async` / `await`.

To use it we first need to create a function/method and mark it as `async`.

```
async function test() {
    // ...
}
```

This function now wraps the return value in a promise and allows us to use the `await` keyword before a promise to halt the execution of the code until it is resolved and gives us the value.

See this example:

```
// Create a new application
const app = new Application();
```

```
// Initialize the application
await app.init({ background: '#1099bb', resizeTo: window });
// Append the application canvas to the document body
document.body.appendChild(app.canvas);
const texture = await
Assets.load('https://pixijs.com/assets/bunny.png');
// Create a new Sprite from the awaited loaded Texture
const bunny = Sprite.from(texture);
// Center the sprite's anchor point
bunny.anchor.set(0.5);
// Move the sprite to the center of the screen
bunny.x = app.screen.width / 2;
bunny.y = app.screen.height / 2;
app.stage.addChild(bunny);
```

The `texture` variable now is not a promise but the resolved texture that resulted after waiting for this promise to resolve.

```
const texture = await Assets.load('examples/assets/bunny.png');
```

This allows us to write more readable code without falling into callback hell and to better think when our program halts and yields.

Loading multiple assets

We can add assets to the cache and then load them all simultaneously by using `Assets.add(...)` and then calling `Assets.load(...)` with all the keys you want to have loaded. See the following example:

```
// Append the application canvas to the document body
document.body.appendChild(app.canvas);
// Add the assets to load
Assets.add({ alias: 'flowerTop', src:
'https://pixijs.com/assets/flowerTop.png' });
Assets.add({ alias: 'eggHead', src:
'https://pixijs.com/assets/eggHead.png' });
// Load the assets and get a resolved promise once both are loaded
const texturesPromise = Assets.load(['flowerTop', 'eggHead']); //
=> Promise<{flowerTop: Texture, eggHead: Texture}>
// When the promise resolves, we have the texture!
```

```
texturesPromise.then((textures) =>
{
    // Create a new Sprite from the resolved loaded Textures
    const flower = Sprite.from(textures.flowerTop);
    flower.anchor.set(0.5);
    flower.x = app.screen.width * 0.25;
    flower.y = app.screen.height / 2;
    app.stage.addChild(flower);
    const egg = Sprite.from(textures.eggHead);
    egg.anchor.set(0.5);
    egg.x = app.screen.width * 0.75;
    egg.y = app.screen.height / 2;
    app.stage.addChild(egg);
});
```

However, if you want to take full advantage of `@pixi/Assets` you should use bundles. Bundles are just a way to group assets together and can be added manually by calling `Assets.addBundle(...)` / `Assets.loadBundle(...)`.

```
  Assets.addBundle('animals', {
    bunny: 'bunny.png',
    chicken: 'chicken.png',
    thumper: 'thumper.png',
  });

 const assets = await Assets.loadBundle('animals');
```

However, the best way to handle bundles is to use a manifest and call `Assets.init({manifest})` with said manifest (or even better, an URL pointing to it). Splitting our assets into bundles that correspond to screens or stages of our app will come in handy for loading in the background while the user is using the app instead of locking them in a single monolithic loading screen.

```
{
   "bundles":[
      {
         "name":"load-screen",
         "assets":[
```

```
            {
               "alias":"background",
               "src":"sunset.png"
            },
            {
               "alias":"bar",
               "src":"load-bar.{png,webp}"
            }
         ]
      },
      {
         "name":"game-screen",
         "assets":[
            {
               "alias":"character",
               "src":"robot.png"
            },
            {
               "alias":"enemy",
               "src":"bad-guy.png"
            }
         ]
      }
   ]
}
```

```
Assets.init({manifest: "path/manifest.json"});
```

Beware that **you can only call `init` once**.
Remember there is no downside in repeating URLs since they will all be cached, so if you need the same asset in two bundles you can duplicate the request without any extra cost!

Background loading

The old approach to loading was to use `Loader` to load all your assets at the beginning of your app, but users are less patient now and want content to be instantly available so the practices are moving towards loading the bare minimum needed to show the user some

content and, while they are interacting with that, we keep loading the following content in the background.

Luckily, `Assets` has us covered with a system that allows us to load everything in the background and in case we need some assets right now, bump them to the top of the queue so we can minimize loading times.

To achieve this, we have the methods `Assets.backgroundLoad(...)` and `Assets.backgroundLoadBundle(...)` that will passively begin to load these assets in the background. So when you finally come to loading them you will get a promise that resolves to the loaded assets immediately.

When you finally need the assets to show, you call the usual `Assets.load(...)` or `Assets.loadBundle(...)` and you will get the corresponding promise.

The best way to do this is using bundles, see the following example:

```
import { Application, Assets, Sprite } from 'pixi.js';

// Create a new application
const app = new Application();

async function init()
{
    // Initialize the application
    await app.init({ background: '#1099bb', resizeTo: window });

    // Append the application canvas to the document body
    document.body.appendChild(app.canvas);

    // Manifest example
    const manifestExample = {
        bundles: [
            {
                name: 'load-screen',
                assets: [
                    {
                        alias: 'flowerTop',
                        src:
'https://pixijs.com/assets/flowerTop.png',
                    },
```

```
                ],
            },
            {
                name: 'game-screen',
                assets: [
                    {
                        alias: 'eggHead',
                        src:
'https://pixijs.com/assets/eggHead.png',
                    },
                ],
            },
        ],
    };

    await Assets.init({ manifest: manifestExample });

    // Bundles can be loaded in the background too!
    Assets.backgroundLoadBundle(['load-screen', 'game-screen']);
}

init();
```

We create one bundle for each screen our game will have and set them all to start downloading at the beginning of our app. If the user progresses slowly enough in our app then they should never get to see a loading screen after the first one!

Containers

The Container class provides a simple display object that does what its name implies - collect a set of child objects together. But beyond grouping objects, containers have a few uses that you should be aware of.

Commonly Used Attributes

The most common attributes you'll use when laying out and animating content in PixiJS are provided by the Container class:

Property	Description

position	X- and Y-position are given in pixels and change the position of the object relative to its parent, also available directly as `object.x` / `object.y`
rotation	Rotation is specified in radians, and turns an object clockwise (0.0 - 2 * Math.PI)
angle	Angle is an alias for rotation that is specified in degrees instead of radians (0.0 - 360.0)
pivot	Point the object rotates around, in pixels - also sets origin for child objects
alpha	Opacity from 0.0 (fully transparent) to 1.0 (fully opaque), inherited by children
scale	Scale is specified as a percent with 1.0 being 100% or actual-size, and can be set independently for the x and y axis
skew	Skew transforms the object in x and y similar to the CSS skew() function, and is specified in radians
visible	Whether the object is visible or not, as a boolean value - prevents updating and rendering object and children

renderable	Whether the object should be rendered - when `false`, object will still be updated, but won't be rendered, doesn't affect children

Containers as Groups

Almost every type of display object is also derived from Container! This means that in many cases you can create a parent-child hierarchy with the objects you want to render. However, it's a good idea *not* to do this. Standalone Container objects are **very** cheap to render, and having a proper hierarchy of Container objects, each containing one or more renderable objects, provides flexibility in rendering order. It also future-proofs your code, as when you need to add an additional object to a branch of the tree, your animation logic doesn't need to change - just drop the new object into the proper Container, and your logic moves the Container with no changes to your code.
So that's the primary use for Containers - as groups of renderable objects in a hierarchy. Check out the container example code.

Masking

Another common use for Container objects is as hosts for masked content. "Masking" is a technique where parts of your scene graph are only visible within a given area.
Think of a pop-up window. It has a frame made of one or more Sprites, then has a scrollable content area that hides content outside the frame. A Container plus a mask makes that scrollable area easy to implement. Add the Container, set its `mask` property to a Graphics object with a rect, and add the text, image, etc. content you want to display as children of that masked Container. Any content that extends beyond the rectangular mask will simply not be drawn. Move the contents of the Container to scroll as desired.

```
// Create the application helper and add its render target to the
page
let app = new Application({ width: 640, height: 360 });
document.body.appendChild(app.view);

// Create window frame
let frame = new Graphics({
  x:320 - 104,
  y:180 - 104
```

```
})
.rect(0, 0, 208, 208)
.fill(0x666666)
.stroke({ color: 0xffffff, width: 4, alignment: 0 })

app.stage.addChild(frame);

// Create a graphics object to define our mask
let mask = new Graphics()
// Add the rectangular area to show
 .rect(0,0,200,200)
 .fill(0xffffff);

// Add container that will hold our masked content
let maskContainer = new Container();
// Set the mask to use our graphics object from above
maskContainer.mask = mask;
// Add the mask as a child, so that the mask is positioned relative
to its parent
maskContainer.addChild(mask);
// Offset by the window's frame width
maskContainer.position.set(4,4);
// And add the container to the window!
frame.addChild(maskContainer);

// Create contents for the masked container
let text = new Text({
  text:'This text will scroll up and be masked, so you can see how
masking works.  Lorem ipsum and all that.\n\n' +
  'You can put anything in the container and it will be masked!',
  style:{
    fontSize: 24,
    fill: 0x1010ff,
    wordWrap: true,
    wordWrapWidth: 180
  },
  x:10
});
```

```
maskContainer.addChild(text);

// Add a ticker callback to scroll the text up and down
let elapsed = 0.0;
app.ticker.add((ticker) => {
  // Update the text's y coordinate to scroll it
  elapsed += ticker.deltaTime;
  text.y = 10 + -100.0 + Math.cos(elapsed/50.0) * 100.0;
});
```

There are two types of masks supported by PixiJS:
Use a Graphics object to create a mask with an arbitrary shape - powerful, but doesn't support anti-aliasing
Sprite: Use the alpha channel from a Sprite as your mask, providing anti-aliased edging - *not* supported on the Canvas renderer

Filtering

Another common use for Container objects is as hosts for filtered content. Filters are an advanced, WebGL/WebGPU-only feature that allows PixiJS to perform per-pixel effects like blurring and displacements. By setting a filter on a Container, the area of the screen the Container encompasses will be processed by the filter after the Container's contents have been rendered.
Below are list of filters available by default in PixiJS. There is, however, a community repository with many more filters.

Filter	Description
AlphaFilter	Similar to setting `alpha` property, but flattens the Container instead of applying to children individually.
BlurFilter	Apply a blur effect
ColorMatrixFilter	A color matrix is a flexible way to apply more complex tints or

	color transforms (e.g., sepia tone).
DisplacementFilter	Displacement maps create visual offset pixels, for instance creating a wavy water effect.
NoiseFilter	Create random noise (e.g., grain effect).

Under the hood, each Filter we offer out of the box is written in both glsl (for WebGL) and wgsl (for WebGPU). This means all filters should work on both renderers.
Important: *Filters should be used somewhat sparingly. They can slow performance and increase memory usage if used too often in a scene.*

Graphics

Graphics is a complex and much misunderstood tool in the PixiJS toolbox. At first glance, it looks like a tool for drawing shapes. And it is! But it can also be used to generate masks. How does that work?
In this guide, we're going to de-mystify the `Graphics` object, starting with how to think about what it does.
Check out the graphics example code.

Graphics Is About Building - Not Drawing

First-time users of the `Graphics` class often struggle with how it works. Let's look at an example snippet that creates a `Graphics` object and draws a rectangle:

```
// Create a Graphics object, draw a rectangle and fill it
let obj = new Graphics()
  .rect(0, 0, 200, 100)
  .fill(0xff0000);

// Add it to the stage to render
app.stage.addChild(obj);
```

That code will work - you'll end up with a red rectangle on the screen. But it's pretty confusing when you start to think about it. Why am I drawing a rectangle

when *constructing* the object? Isn't drawing something a one-time action? How does the rectangle get drawn the *second* frame? And it gets even weirder when you create a `Graphics` object with a bunch of drawThis and drawThat calls, and then you use it as a *mask*. What???

The problem is that the function names are centered around *drawing*, which is an action that puts pixels on the screen. But in spite of that, the `Graphics` object is really about *building*.

Let's look a bit deeper at that `rect()` call. When you call `rect()`, PixiJS doesn't actually draw anything. Instead, it stores the rectangle you "drew" into a list of geometry for later use. If you then add the `Graphics` object to the scene, the renderer will come along, and ask the `Graphics` object to render itself. At that point, your rectangle actually gets drawn - along with any other shapes, lines, etc. that you've added to the geometry list.

Once you understand what's going on, things start to make a lot more sense. When you use a `Graphics` object as a mask, for example, the masking system uses that list of graphics primitives in the geometry list to constrain which pixels make it to the screen. There's no drawing involved.

That's why it helps to think of the `Graphics` class not as a drawing tool, but as a geometry building tool.

Types of Primitives

There are a lot of functions in the `Graphics` class, but as a quick orientation, here's the list of basic primitives you can add:

- Line
- Rect
- RoundRect
- Circle
- Ellipse
- Arc
- Bezier and Quadratic Curve

In addition, you have access to the following complex primitives:

- Torus
- Chamfer Rect
- Fillet Rect
- Regular Polygon
- Star
- Rounded Polygon

There is also support for svg. But due to the nature of how PixiJS renders holes (it favours performance) Some complex hole shapes may render incorrectly. But for the majority of shapes, this will do the trick!

```
let mySvg = new Graphics().svg(`
  <svg>
    <path d="M 100 350 q 150 -300 300 0" stroke="blue" />
  </svg>
`);
```

The GraphicsContext

Understanding the relationship between Sprites and their shared Texture can help grasp the concept of a `GraphicsContext`. Just as multiple Sprites can utilize a single Texture, saving memory by not duplicating pixel data, a GraphicsContext can be shared across multiple Graphics objects.

This sharing of a `GraphicsContext` means that the intensive task of converting graphics instructions into GPU-ready geometry is done once, and the results are reused, much like textures. Consider the difference in efficiency between these approaches:

Creating individual circles without sharing a context:

```
// Create 5 circles
for (let i = 0; i < 5; i++) {
  let circle = new Graphics()
    .circle(100, 100, 50)
    .fill('red');
}
```

Versus sharing a GraphicsContext:

```
// Create a master Graphicscontext
let circleContext = new GraphicsContext()
  .circle(100, 100, 50)
  .fill('red')

// Create 5 duplicate objects
for (let i = 0; i < 5; i++) {
  // Initialize the duplicate using our circleContext
```

```
  let duplicate = new Graphics(circleContext);
}
```

Now, this might not be a huge deal for circles and squares, but when you are using SVGs, it becomes quite important to not have to rebuild each time and instead share a `GraphicsContext`. It's recommended for maximum performance to create your contexts upfront and reuse them, just like textures!

```
let circleContext = new GraphicsContext()
  .circle(100, 100, 50)
  .fill('red')

let rectangleContext = new GraphicsContext()
  .rect(0, 0, 50, 50)
  .fill('red')

let frames = [circleContext, rectangleContext];
let frameIndex = 0;

const graphics = new Graphics(frames[frameIndex]);

// animate from square to circle:

function update()
{
  // swap the context - this is a very cheap operation!
  // much cheaper than clearing it each frame.
  graphics.context = frames[frameIndex++%frames.length];
}
```

If you don't explicitly pass a `GraphicsContext` when creating a `Graphics` object, then internally, it will have its own context, accessible via `myGraphics.context`.
The GraphicsContext class manages the list of geometry primitives created by the Graphics parent object. Graphics functions are literally passed through to the internal contexts:

```
let circleGraphics = new Graphics()
  .circle(100, 100, 50)
```

```
  .fill('red')
```

same as:

```
let circleGraphics = new Graphics()

circleGraphics.context
  .circle(100, 100, 50)
  .fill('red')
```

Calling `Graphics.destroy()` will destroy the graphics. If a context was passed to it via the constructor then it will leave the destruction of that context to you. However if the context is internally created (the default), when destroyed the Graphics object will destroy its internal `GraphicsContext`.

Graphics For Display

OK, so now that we've covered how the `Graphics` class works, let's look at how you use it. The most obvious use of a `Graphics` object is to draw dynamically generated shapes to the screen.
Doing so is simple. Create the object, call the various builder functions to add your custom primitives, then add the object to the scene graph. Each frame, the renderer will come along, ask the `Graphics` object to render itself, and each primitive, with associated line and fill styles, will be drawn to the screen.

Graphics as a Mask

You can also use a Graphics object as a complex mask. To do so, build your object and primitives as usual. Next create a `Container` object that will contain the masked content, and set its `mask` property to your Graphics object. The children of the container will now be clipped to only show through inside the geometry you've created. This technique works for both WebGL and Canvas-based rendering.
Check out the masking example code.

Caveats and Gotchas

The `Graphics` class is a complex beast, and so there are a number of things to be aware of when using it.
Memory Leaks: Call `destroy()` on any `Graphics` object you no longer need to avoid memory leaks.

Holes: Holes you create have to be completely contained in the shape or else it may not be able to triangulate correctly.
Changing Geometry: If you want to change the shape of a `Graphics` object, you don't need to delete and recreate it. Instead you can use the `clear()` function to reset the contents of the geometry list, then add new primitives as desired. Be careful of performance when doing this every frame.
Performance: `Graphics` objects are generally quite performant. However, if you build highly complex geometry, you may pass the threshold that permits batching during rendering, which can negatively impact performance. It's better for batching to use many `Graphics` objects instead of a single `Graphics` with many shapes.
Transparency: Because the `Graphics` object renders its primitives sequentially, be careful when using blend modes or partial transparency with overlapping geometry. Blend modes like `ADD` and `MULTIPLY` will work *on each primitive*, not on the final composite image. Similarly, partially transparent `Graphics` objects will show primitives overlapping. To apply transparency or blend modes to a single flattened surface, consider using AlphaFilter or RenderTexture.

Graphics Pixel Line

The `pixelLine` property is a neat feature of the PixiJS Graphics API that allows you to create lines that remain 1 pixel thick, regardless of scaling or zoom level. As part of the Graphics API, it gives developers all the power PixiJS provides for building and stroking shapes. This feature is especially useful for achieving crisp, pixel-perfect visuals, particularly in retro-style or grid-based games, technical drawing, or UI rendering.
In this guide, we'll dive into how this property works, its use cases, and the caveats you should be aware of when using it.

How to use `pixelLine`?

Here's a simple example:

```
// Create a Graphics object and draw a pixel-perfect line
let graphics = new Graphics()
  .moveTo(0, 0)
  .lineTo(100, 100)
  .stroke({ color: 0xff0000, pixelLine: true });
```

```
// Add it to the stage
app.stage.addChild(graphics);

// Even if we scale the Graphics object, the line remains 1 pixel
wide
graphics.scale.set(2);
```

In this example, no matter how you transform or zoom the `Graphics` object, the red line will always appear 1 pixel thick on the screen.

Why Use `pixelLine`?

Pixel-perfect lines can be incredibly useful in a variety of scenarios. Here are some common use cases:

1. Retro or Pixel Art Games

- Pixel art games rely heavily on maintaining sharp, precise visuals.
  The `pixelLine` property ensures that lines do not blur or scale inconsistently with other pixel elements.
- Example: Drawing pixel-perfect grids for tile-based maps.

```
// Create a grid of vertical and horizontal lines
const grid = new Graphics();

// Draw 10 vertical lines spaced 10 pixels apart
// Draw vertical lines
for (let i = 0; i < 10; i++) {
    // Move to top of each line (x = i*10, y = 0)
    grid.moveTo(i * 10, 0)
        // Draw down to bottom (x = i*10, y = 100)
        .lineTo(i * 10, 100);
}

// Draw horizontal lines
for (let i = 0; i < 10; i++) {
    // Move to start of each line (x = 0, y = i*10)
    grid.moveTo(0, i * 10)
        // Draw across to end (x = 100, y = i*10)
```

```
        .lineTo(100, i * 10);
}

// Stroke all lines in white with pixel-perfect width
grid.stroke({ color: 0xffffff, pixelLine: true });
```

---

2. UI and HUD Elements

- For UI elements such as borders, separators, or underlines, a consistent 1-pixel thickness provides a professional, clean look.
- Example: Drawing a separator line in a menu or a progress bar border.

```
// Create a separator line that will always be 1 pixel thick
const separator = new Graphics()
    // Start at x=0, y=50
    .moveTo(0, 50)
    // Draw a horizontal line 200 pixels to the right
    .lineTo(200, 50)
    // Stroke in green with pixel-perfect 1px width
    .stroke({ color: 0x00ff00, pixelLine: true });
```

---

3. Debugging and Prototyping

- Use pixel-perfect lines to debug layouts, collision boxes, or grids. Since the lines don't scale, they offer a consistent reference point during development.
- Example: Displaying collision boundaries in a physics-based game.

```
// Create a debug box with pixel-perfect stroke
const graphicsBox = new Graphics()
    .rect(0, 0, 100, 100)
  .stroke({ color: 0xff00ff, pixelLine: true });

/**
 * Updates the debug box to match the bounds of a given object
 * @param {Container} obj - The object to draw bounds for
 */
function drawDebugBounds(obj) {
    // Get the bounds of the object
    let bounds = obj.getBounds().rectangle;
```

```
    // Position and scale the debug box to match the bounds
    // this is faster than using `moveTo` and `lineTo` each frame!
    graphicsBox.position.set(bounds.x, bounds.y);
    graphicsBox.scale.set(bounds.width / 100, bounds.height / 100);
}
```

How it works

This is achieved under the hood using WebGL or WebGPU's native line rendering methods when `pixelLine` is set to `true`.

Fun fact its actually faster to draw a pixel line than a regular line. This is because of two main factors:

1. **Simpler Drawing Process**: Regular lines in PixiJS (when `pixelLine` is `false`) need extra steps to be drawn. PixiJS has to figure out the thickness of the line and create a shape that looks like a line but is actually made up of triangles.
2. **Direct Line Drawing**: When using `pixelLine`, we can tell the graphics card "just draw a line from point A to point B" and it knows exactly what to do. This is much simpler and faster than creating and filling shapes.

Think of it like drawing a line on paper - `pixelLine` is like using a pen to draw a straight line, while regular lines are like having to carefully color in a thin rectangle. The pen method is naturally faster and simpler!

Caveats and Gotchas

While the `pixelLine` property is incredibly useful, there are some limitations and things to keep in mind:

1. Its 1px thick, thats it!

- The line is always 1px thick, there is no way to change this as its using the GPU to draw the line.

2. Hardware may render differently

- Different GPUs and graphics hardware may render the line slightly differently due to variations in how they handle line rasterization. For example, some GPUs may position the line slightly differently or apply different anti-aliasing techniques. This is an inherent limitation of GPU line rendering and is beyond PixiJS's control.

4. Scaling Behavior

- While the line thickness remains constant, other properties (e.g., position or start/end points) are still affected by scaling. This can sometimes create unexpected results if combined with other scaled objects. This is a feature not a bug :)

Example: Box with Pixel-Perfect Stroke

Here's an example of a filled box with a pixel-perfect stroke. The box itself scales and grows, but the stroke remains 1 pixel wide:

```
// Create a Graphics object and draw a filled box with a pixel-perfect stroke
let box = new Graphics()
    .rect(0, 0, 100, 100)
    .fill('white')
    .stroke({ color: 0xff0000, pixelLine: true });

// Add it to the stage
app.stage.addChild(box);

// Scale the box
box.scale.set(2);
```

In this example, the blue box grows as it scales, but the red stroke remains at 1 pixel thickness, providing a crisp outline regardless of the scaling.

When to Avoid Using `pixelLine`

- **You want a line that is not 1px thick:** Don't use `pixelLine`.
- **You want the line to scale:** Don't use `pixelLine`

Conclusion

The `pixelLine` property is a super useful to have in the PixiJS toolbox for developers looking to create sharp, pixel-perfect lines that remain consistent under transformation. By understanding its strengths and limitations, you can incorporate it into your projects for clean, professional results in both visual and functional elements.

Graphics Fill

If you are new to graphics, please check out the graphics guide here. This guide dives a bit deeper into a specific aspect of graphics: how to fill them! The `fill()` method in

PixiJS is particularly powerful, enabling you to fill shapes with colors, textures, or gradients. Whether you're designing games, UI components, or creative tools, mastering the `fill()` method is essential for creating visually appealing and dynamic graphics. This guide explores the different ways to use the `fill()` method to achieve stunning visual effects.

Note

The `fillStyles` discussed here can also be applied to Text objects!

Basic Color Fills

When creating a `Graphics` object, you can easily fill it with a color using the `fill()` method. Here's a simple example:

```
const obj = new Graphics()
  .rect(0, 0, 200, 100) // Create a rectangle with dimensions 200x100
  .fill('red'); // Fill the rectangle with a red color
```

This creates a red rectangle. PixiJS supports multiple color formats for the `fill()` method. Developers can choose a format based on their needs. For example, CSS color strings are user-friendly and readable, hexadecimal strings are compact and widely used in design tools, and numbers are efficient for programmatic use. Arrays and Color objects offer precise control, making them ideal for advanced graphics.

- CSS color strings (e.g., 'red', 'blue')
- Hexadecimal strings (e.g., '#ff0000')
- Numbers (e.g., `0xff0000`)
- Arrays (e.g., `[255, 0, 0]`)
- Color objects for precise color control

Examples:

```
// Using a number
const obj1 = new Graphics().rect(0, 0, 100, 100).fill(0xff0000);

// Using a hex string
const obj2 = new Graphics().rect(0, 0, 100, 100).fill('#ff0000');

// Using an array
const obj3 = new Graphics().rect(0, 0, 100, 100).fill([255, 0, 0]);
```

```
// Using a Color object
const color = new Color();
const obj4 = new Graphics().rect(0, 0, 100, 100).fill(color);
```

Fill with a Style Object

For more advanced fills, you can use a `FillStyle` object. This allows for additional customization, such as setting opacity:

```
const obj = new Graphics().rect(0, 0, 100, 100)
  .fill({
    color: 'red',
    alpha: 0.5, // 50% opacity
  });
```

Fill with Textures

Filling shapes with textures is just as simple:

```
const texture = await Assets.load('assets/image.png');
const obj = new Graphics().rect(0, 0, 100, 100)
  .fill(texture);
```

Local vs. Global Texture Space

Textures can be applied in two coordinate spaces:

- **Local Space** (Default): The texture coordinates are mapped relative to the shape's dimensions and position. The texture coordinates use a normalized coordinate system where (0,0) is the top-left and (1,1) is the bottom-right of the shape, regardless of its actual pixel dimensions. For example, if you have a 300×200 pixel texture filling a 100×100 shape, the texture will be scaled to fit exactly within those 100×100 pixels. The texture's top-left corner (0,0) will align with the shape's top-left corner, and the texture's bottom-right corner (1,1) will align with the shape's bottom-right corner, stretching or compressing the texture as needed.

```
const shapes = new PIXI.Graphics()
    .rect(50,50,100, 100)
```

```
    .circle(250,100,50)
    .star(400,100,6,60,40)
    .roundRect(500,50,100,100,10)
    .fill({
        texture,
        textureSpace:'local' // default!
    });
```

- **Global Space**: Set `textureSpace: 'global'` to make the texture position and scale relative to the Graphics object's coordinate system. Despite the name, this isn't truly "global" - the texture remains fixed relative to the Graphics object itself, maintaining its position even when the object moves or scales. See how the image goes across all the shapes (in the same graphics) below:

```
const shapes = new PIXI.Graphics()
    .rect(50,50,100, 100)
    .circle(250,100,50)
    .star(400,100,6,60,40)
    .roundRect(500,50,100,100,10)
    .fill({
        texture,
        textureSpace:'global'
    });
```

Using Matrices with Textures

To modify texture coordinates, you can apply a transformation matrix, which is a mathematical tool used to scale, rotate, or translate the texture. If you're unfamiliar with transformation matrices, they allow for precise control over how textures are rendered, and you can explore more about them here.

```
const matrix = new Matrix().scale(0.5, 0.5);

const obj = new Graphics().rect(0, 0, 100, 100)
  .fill({
    texture: texture,
```

```
    matrix: matrix, // scale the texture down by 2
  });
```

Texture Gotcha's

1. **Sprite Sheets**: If using a texture from a sprite sheet, the entire source texture will be used. To use a specific frame, create a new texture:

```
const spriteSheetTexture = Texture.from('assets/my-sprite-sheet.png');
const newTexture = renderer.generateTexture(Sprite.from(spriteSheetTexture));

const obj = new Graphics().rect(0, 0, 100, 100)
  .fill(newTexture);
```

2. **Power of Two Textures**: Textures should be power-of-two dimensions for proper tiling in WebGL1 (WebGL2 and WebGPU are fine).

Fill with Gradients

PixiJS supports both linear and radial gradients, which can be created using the `FillGradient` class. Gradients are particularly useful for adding visual depth and dynamic styling to shapes and text.

Linear Gradients

Linear gradients create a smooth color transition along a straight line. Here is an example of a simple linear gradient:

```
const gradient = new FillGradient({
  type: 'linear',
  colorStops: [
    { offset: 0, color: 'yellow' },
    { offset: 1, color: 'green' },
  ],
});

const obj = new Graphics().rect(0, 0, 100, 100)
  .fill(gradient);
```

You can control the gradient direction with the following properties:

- `start {x, y}`: These define the starting point of the gradient. For example, in a linear gradient, this is where the first color stop is positioned. These values are typically expressed in relative coordinates (0 to 1), where `0` represents the left/top edge and `1` represents the right/bottom edge of the shape.
- `end {x, y}`: These define the ending point of the gradient. Similar to `start {x, y}`, these values specify where the last color stop is positioned in the shape's local coordinate system.

Using these properties, you can create various gradient effects, such as horizontal, vertical, or diagonal transitions. For example, setting `start` to `{x: 0, y: 0}` and `end` to `{x: 1, y: 1}` would result in a diagonal gradient from the top-left to the bottom-right of the shape.

```
const diagonalGradient = new FillGradient({
  type: 'linear',
  start: { x: 0, y: 0 },
  end: { x: 1, y: 1 },
  colorStops: [
    { offset: 0, color: 'yellow' },
    { offset: 1, color: 'green' },
  ],
});
```

Radial Gradients

Radial gradients create a smooth color transition in a circular pattern. Unlike linear gradients, they blend colors from one circle to another. Here is an example of a simple radial gradient:

```
const gradient = new FillGradient({
  type: 'radial',
  colorStops: [
    { offset: 0, color: 'yellow' },
    { offset: 1, color: 'green' },
  ],
});
```

```
const obj = new Graphics().rect(0, 0, 100, 100)
  .fill(gradient);
```

You can control the gradient's shape and size using the following properties:

- `center {x, y}`: These define the center of the inner circle where the gradient starts. Typically, these values are expressed in relative coordinates (0 to 1), where `0.5` represents the center of the shape.
- `innerRadius`: The radius of the inner circle. This determines the size of the gradient's starting point.
- `outerCenter {x, y}`: These define the center of the outer circle where the gradient ends. Like `center {x, y}`, these values are also relative coordinates.
- `outerRadius`: The radius of the outer circle. This determines the size of the gradient's ending point.

By adjusting these properties, you can create a variety of effects, such as small, concentrated gradients or large, expansive ones. For example, setting a small `r0` and a larger `r1` will create a gradient that starts does not start to transition until the inner circle radius is reached.

```
const radialGradient = new FillGradient({
  type: 'radial',
  center: { x: 0.5, y: 0.5 },
  innerRadius: 0.25,
  outerCenter: { x: 0.5, y: 0.5 },
  outerRadius: 0.5,
  colorStops: [
    { offset: 0, color: 'blue' },
    { offset: 1, color: 'red' },
  ],
});

const obj = new Graphics().rect(0, 0, 100, 100)
  .fill(gradient);
```

Gradient Gotcha's

1. **Memory Management**: Use `fillGradient.destroy()` to free up resources when gradients are no longer needed.
2. **Animation**: Update existing gradients instead of creating new ones for better performance.
3. **Custom Shaders**: For complex animations, custom shaders may be more efficient.
4. **Texture and Matrix Limitations**: Under the hood, gradient fills set both the texture and matrix properties internally. This means you cannot use a texture fill or matrix transformation at the same time as a gradient fill.

Combining Textures and Colors

You can combine a texture or gradients with a color tint and alpha to achieve more complex and visually appealing effects. This allows you to overlay a color on top of the texture or gradient, adjusting its transparency with the alpha value.

```
const gradient = new FillGradient({
    colorStops: [
        { offset: 0, color: 'blue' },
        { offset: 1, color: 'red' },
    ]
});

const obj = new Graphics().rect(0, 0, 100, 100)
  .fill({
    fill: gradient,
    color: 'yellow',
    alpha: 0.5,
  });
```

```
const obj = new Graphics().rect(0, 0, 100, 100)
  .fill({
    texture: texture,
    color: 'yellow',
    alpha: 0.5,
  });
```

Hopefully, this guide has shown you how easy and powerful fills can be when working with graphics (and text!). By mastering the `fill()` method, you can unlock endless possibilities for creating visually dynamic and engaging graphics in PixiJS. Have fun!

Interaction

PixiJS is primarily a rendering system, but it also includes support for interactivity. Adding support for mouse and touch events to your project is simple and consistent.

Event Modes

Prior to v7, interaction was defined and managed by the `Interaction` package and its `InteractionManager`. Beginning with v7, however, a new event-based system has replaced the previous `Interaction` package, and expanded the definition of what it means for a `Container` to be interactive.
With this, we have introduced `eventMode` which allows you to control how an object responds to interaction events. If you're familiar with the former `Interaction` system, the `eventMode` is similar to the `interactive` property, but with more options.

eventMode	Description
`none`	Ignores all interaction events, similar to CSS's `pointer-events: none`. Good optimization for non-interactive children.
`passive`	The **default** `eventMode` for all containers. Does not emit events and ignores hit-testing on itself, but *does* allow for events and hit-testing on its interactive children.
`auto`	Does not emit events, but is hit tested if parent is interactive. Same as `interactive = false` in v7.

`static`	Emits events and is hit tested. Same as `interaction = true` in v7. Useful for objects like buttons that do not move.
`dynamic`	Emits events and is hit tested, but will also receive mock interaction events fired from a ticker to allow for interaction when the mouse isn't moving. Useful for elements that are independently moving or animating.

Event Types

PixiJS supports the following event types:

Event Type	Description
`pointercancel`	Fired when a pointer device button is released outside the display object that initially registered a pointerdown.
`pointerdown`	Fired when a pointer device button is pressed on the display object.
`pointerenter`	Fired when a pointer device enters the display object.

`pointerleave`	Fired when a pointer device leaves the display object.
`pointermove`	Fired when a pointer device is moved while over the display object.
`globalpointermove`	Fired when a pointer device is moved, regardless of hit-testing the current object.
`pointerout`	Fired when a pointer device is moved off the display object.
`pointerover`	Fired when a pointer device is moved onto the display object.
`pointertap`	Fired when a pointer device is tapped on the display object.
`pointerup`	Fired when a pointer device button is released over the display object.
`pointerupoutside`	Fired when a pointer device button is released outside the display object that initially

	registered a pointerdown.
`mousedown`	Fired when a mouse button is pressed on the display object.
`mouseenter`	Fired when the mouse cursor enters the display object.
`mouseleave`	Fired when the mouse cursor leaves the display object.
`mousemove`	Fired when the mouse cursor is moved while over the display object.
`globalmousemove`	Fired when a mouse is moved, regardless of hit-testing the current object.
`mouseout`	Fired when the mouse cursor is moved off the display object.
`mouseover`	Fired when the mouse cursor is moved onto the display object.
`mouseup`	Fired when a mouse button is released over the display object.

`mouseupoutside`	Fired when a mouse button is released outside the display object that initially registered a mousedown.
`click`	Fired when a mouse button is clicked (pressed and released) over the display object.
`touchcancel`	Fired when a touch point is removed outside of the display object that initially registered a touchstart.
`touchend`	Fired when a touch point is removed from the display object.
`touchendoutside`	Fired when a touch point is removed outside of the display object that initially registered a touchstart.
`touchmove`	Fired when a touch point is moved along the display object.
`globaltouchmove`	Fired when a touch point is moved, regardless of hit-

	testing the current object.
`touchstart`	Fired when a touch point is placed on the display object.
`tap`	Fired when a touch point is tapped on the display object.
`wheel`	Fired when a mouse wheel is spun over the display object.
`rightclick`	Fired when a right mouse button is clicked (pressed and released) over the display object.
`rightdown`	Fired when a right mouse button is pressed on the display object.
`rightup`	Fired when a right mouse button is released over the display object.
`rightupoutside`	Fired when a right mouse button is released outside the display object that initially registered a rightdown.

Enabling Interaction

Any `Container`-derived object (`Sprite`, `Container`, etc.) can become interactive simply by setting its `eventMode` property to any of the eventModes listed above. Doing so will cause the object to emit interaction events that can be responded to in order to drive your project's behavior.

Check out the click interactivity example code.

To respond to clicks and taps, bind to the events fired on the object, like so:

```
let sprite = Sprite.from('/some/texture.png');
sprite.on('pointerdown', (event) => { alert('clicked!'); });
sprite.eventMode = 'static';
```

Check out the Container for the list of interaction events supported.

Checking if an Object is Interactive

You can check if an object is interactive by calling the `isInteractive` property. This will return true if `eventMode` is set to `static` or `dynamic`.

```
if (sprite.isInteractive()) {
    // sprite is interactive
}
```

Use Pointer Events

PixiJS supports three types of interaction events: mouse, touch, and pointer.

- Mouse events are fired by mouse movement, clicks etc.
- Touch events are fired for touch-capable devices. And,
- Pointer events are fired for *both*.

What this means is that, in many cases, you can write your project to use pointer events and it will just work when used with *either* mouse or touch input.

Given that, the only reason to use non-pointer events is to support different modes of operation based on input type or to support multi-touch interaction. In all other cases, prefer pointer events.

Optimization

Hit testing requires walking the full object tree, which in complex projects can become an optimization bottleneck.

To mitigate this issue, PixiJS `Container`-derived objects have a property named `interactiveChildren`. If you have `Container`s or other objects with complex child trees that you know will *never* be interactive, you can set this property to `false`,

and the hit-testing algorithm will skip those children when checking for hover and click events.

As an example, if you were building a side-scrolling game, you would probably want to set `background.interactiveChildren = false` for your background layer with rocks, clouds, flowers, etc. Doing so would substantially speed up hit-testing due to the number of unclickable child objects the background layer would contain.

The `EventSystem` can also be customised to be more performant:

```
const app = new Application({
    eventMode: 'passive',
    eventFeatures: {
        move: true,
        /** disables the global move events which can be very
expensive in large scenes */
        globalMove: false,
        click: true,
        wheel: true,
    }
});
```

Sprites

Sprites are the simplest and most common renderable object in PixiJS. They represent a single image to be displayed on the screen. Each Sprite contains a Texture to be drawn, along with all the transformation and display state required to function in the scene graph.

Creating Sprites

To create a Sprite, all you need is a Texture (check out the Texture guide). Load a PNG's URL using the `Assets` class, then call `Sprite.from(url)` and you're all set. Unlike v7 you now must load your texture before using it, this is to ensure best practices.

Check out the sprite example code.

Using Sprites

In our Container guide, we learned about the Container class and the various properties it defines. Since Sprite objects are also containers, you can move a sprite, rotate it, and update any other display property.

Alpha, Tint and Blend Modes

Alpha is a standard display object property. You can use it to fade sprites into the scene by animating each sprite's alpha from 0.0 to 1.0 over a period of time.
Tinting allows you to multiply the color value of every pixel by a single color. For example, if you had a dungeon game, you might show a character's poison status by setting `obj.tint = 0x00FF00`, which would give a green tint to the character.
Blend modes change how pixel colors are added to the screen when rendering. The three main modes are **add**, which adds each pixel's RGB channels to whatever is under your sprite (useful for glows and lighting), **multiply** which works like `tint`, but on a per-pixel basis, and **screen**, which overlays the pixels, brightening whatever is underneath them.

Scale vs Width & Height

One common area of confusion when working with sprites lies in scaling and dimensions. The Container class allows you to set the x and y scale for any object. Sprites, being Containers, also support scaling. In addition, however, Sprites support explicit `width` and `height` attributes that can be used to achieve the same effect, but are in pixels instead of a percentage. This works because a Sprite object owns a Texture, which has an explicit width and height. When you set a Sprite's width, internally PixiJS converts that width into a percentage of the underlying texture's width and updates the object's x-scale. So width and height are really just convenience methods for changing scale, based on pixel dimensions rather than percentages.

Pivot vs Anchor

If you add a sprite to your stage and rotate it, it will by default rotate around the top-left corner of the image. In some cases, this is what you want. In many cases, however, what you want is for the sprite to rotate around the center of the image it contains, or around an arbitrary point.
There are two ways to achieve this: *pivots* and *anchors*
An object's **pivot** is an offset, expressed in pixels, from the top-left corner of the Sprite. It defaults to (0, 0). If you have a Sprite whose texture is 100px x 50px, and want to set the pivot point to the center of the image, you'd set your pivot to (50, 25) - half the width, and half the height. Note that pivots can be set *outside* of the image, meaning the pivot may be less than zero or greater than the width/height. This can be useful in setting up complex animation hierarchies, for example. Every Container has a pivot.
An **anchor**, in contrast, is only available for Sprites. Anchors are specified in percentages, from 0.0 to 1.0, in each dimension. To rotate around the center point of a texture using anchors, you'd set your Sprite's anchor to (0.5, 0.5) - 50% in width and height. While less common, anchors can also be outside the standard 0.0 - 1.0 range.

The nice thing about anchors is that they are resolution and dimension agnostic. If you set your Sprite to be anchored in the middle then later change the size of the texture, your object will still rotate correctly. If you had instead set a pivot using pixel-based calculations, changing the texture size would require changing your pivot point.
So, generally speaking, you'll want to use anchors when working with Sprites.
One final note: unlike CSS, where setting the transform-origin of the image doesn't move it, in PixiJS setting an anchor or pivot *will* move your object on the screen. In other words, setting an anchor or pivot affects not just the rotation origin, but also the position of the sprite relative to its parent.

Spritesheets

Now that you understand basic sprites, it's time to talk about a better way to create them - the Spritesheet class.
A Spritesheet is a media format for more efficiently downloading and rendering Sprites. While somewhat more complex to create and use, they are a key tool in optimizing your project.

Anatomy of a Spritesheet

The basic idea of a spritesheet is to pack a series of images together into a single image, track where each source image ends up, and use that combined image as a shared BaseTexture for the resulting Sprites.
The first step is to collect the images you want to combine. The sprite packer then collects the images, and creates a new combined image.
As this image is being created, the tool building it keeps track of the location of the rectangle where each source image is stored. It then writes out a JSON file with that information.
These two files, in combination, can be passed into a SpriteSheet constructor. The SpriteSheet object then parses the JSON, and creates a series of Texture objects, one for each source image, setting the source rectangle for each based on the JSON data. Each texture uses the same shared BaseTexture as its source.

Doubly Efficient

SpriteSheets help your project in two ways.
First, by **speeding up the loading process**. While downloading a SpriteSheet's texture requires moving the same (or even slightly more!) number of bytes, they're grouped into a single file. This means that the user's browser can request and download far fewer files for the same number of Sprites. The number of files *itself* is a key driver of download speed,

because each request requires a round-trip to the webserver, and browsers are limited to how many files they can download simultaneously. Converting a project from individual source images to shared sprite sheets can cut your download time in half, at no cost in quality.

Second, by **improving batch rendering**. WebGL rendering speed scales roughly with the number of draw calls made. Batching multiple Sprites, etc. into a single draw call is the main secret to how PixiJS can run so blazingly fast. Maximizing batching is a complex topic, but when multiple Sprites all share a common BaseTexture, it makes it more likely that they can be batched together and rendered in a single call.

Creating SpriteSheets

You can use a 3rd party tool to assemble your sprite sheet files. Here are two that may fit your needs:

ShoeBox: ShoeBox is a free, Adobe AIR-based sprite packing utility that is great for small projects or learning how SpriteSheets work.

TexturePacker: TexturePacker is a more polished tool that supports advanced features and workflows. A free version is available which has all the necessary features for packing spritesheets for PixiJS. It's a good fit for larger projects and professional game development, or projects that need more complex tile mapping features.

Spritesheet data can also be created manually or programmatically, and supplied to a new AnimatedSprite. This may be an easier option if your sprites are already contained in a single image.

```
// Create object to store sprite sheet data
const atlasData = {
        frames: {
                enemy1: {
                        frame: { x: 0, y:0, w:32, h:32 },
                        sourceSize: { w: 32, h: 32 },
                        spriteSourceSize: { x: 0, y: 0, w: 32, h:
32 }
                },
                enemy2: {
                        frame: { x: 32, y:0, w:32, h:32 },
                        sourceSize: { w: 32, h: 32 },
                        spriteSourceSize: { x: 0, y: 0, w: 32, h:
32 }
                },
```

```
        },
        meta: {
                image: 'images/spritesheet.png',
                format: 'RGBA8888',
                size: { w: 128, h: 32 },
                scale: 1
        },
        animations: {
                enemy: ['enemy1','enemy2'] //array of frames by
name
        }
}


// Create the SpriteSheet from data and image
const spritesheet = new Spritesheet(
        Texture.from(atlasData.meta.image),
        atlasData
);

// Generate all the Textures asynchronously
await spritesheet.parse();

// spritesheet is ready to use!
const anim = new AnimatedSprite(spritesheet.animations.enemy);

// set the animation speed
anim.animationSpeed = 0.1666;
// play the animation on a loop
anim.play();
// add it to the stage to render
app.stage.addChild(anim);
```

Text

Whether it's a high score or a diagram label, text is often the best way to convey information in your projects. Surprisingly, drawing text to the screen with WebGL is a very

complex process - there's no built in support for it at all. One of the values PixiJS provides is in hiding this complexity to allow you to draw text in diverse styles, fonts and colors with a few lines of code. In addition, these bits of text are just as much scene objects as sprites - you can tint text, rotate it, alpha-blend it, and otherwise treat it like any other graphical object.

Let's dig into how this works.

There Are Three Kinds of Text

Because of the challenges of working with text in WebGL, PixiJS provides three very different solutions. In this guide, we're going to go over both methods in some detail to help you make the right choice for your project's needs. Selecting the wrong text type can have a large negative impact on your project's performance and appearance.

The Text Object

In order to draw text to the screen, you use a Text object. Under the hood, this class draws text to an off-screen buffer using the browser's normal text rendering, then uses that offscreen buffer as the source for drawing the text object. Effectively what this means is that whenever you create or change text, PixiJS creates a new rasterized image of that text, and then treats it like a sprite. This approach allows truly rich text display while keeping rendering speed high.

So when working with Text objects, there are two sets of options - standard display object options like position, rotation, etc that work *after* the text is rasterized internally, and text style options that are used *while* rasterizing. Because text once rendered is basically just a sprite, there's no need to review the standard options. Instead, let's focus on how text is styled.

Check out the text example code.

Text Styles

There are a lot of text style options available (see TextStyle), but they break down into 5 main groups:

Font: `fontFamily` to select the webfont to use, `fontSize` to specify the size of the text to draw, along with options for font weight, style and variant.

Appearance: Set the color with `fill` or add a `stroke` outline, including options for gradient fills. For more information on the `fill` property, see the Graphics Fill guide.

Drop-Shadows: Set a drop-shadow with `dropShadow`, with a host of related options to specify offset, blur, opacity, etc.

Layout: Enable with `wordWrap` and `wordWrapWidth`, and then customize the `lineHeight` and `align` or `letterSpacing`

Utilities: Add `padding` or `trim` extra space to deal with funky font families if needed.
To interactively test out feature of Text Style, check out this tool.

Loading and Using Fonts

In order for PixiJS to build a Text object, you'll need to make sure that the font you want to use is loaded by the browser. This can be easily accomplished with our good friends `Assets`

```
// load the fonts
await Assets.load('short-stack.woff2');

// now they can be used!

const text = new Text({
  text:'hello',
  style:{
    fontFamily:'short-stack'
  }
})
```

Caveats and Gotchas

While PixiJS does make working with text easy, there are a few things you need to watch out for.
First, changing an existing text string requires re-generating the internal render of that text, which is a slow operation that can impact performance if you change many text objects each frame. If your project requires lots of frequently changing text on the screen at once, consider using a BitmapText object (explained below) which uses a fixed bitmap font that doesn't require re-generation when text changes.
Second, be careful when scaling text. Setting a text object's scale to > 1.0 will result in blurry/pixely display, because the text is not re-rendered at the higher resolution needed to look sharp - it's still the same resolution it was when generated. To deal with this, you can render at a higher initial size and down-scale, instead. This will use more memory, but will allow your text to always look clear and crisp.

BitmapText

In addition to the standard Text approach to adding text to your project, PixiJS also supports *bitmap fonts*. Bitmap fonts are very different from TrueType or other general purpose fonts, in that they consist of a single image containing pre-rendered versions of

every letter you want to use. When drawing text with a bitmap font, PixiJS doesn't need to render the font glyphs into a temporary buffer - it can simply copy and stamp out each character of a string from the master font image.
The primary advantage of this approach is speed - changing text frequently is much cheaper and rendering each additional piece of text is much faster due to the shared source texture.
Check out the bitmap text example code.

BitmapFont

- 3rd party solutions
- BitmapFont.from auto-generation

Selecting the Right Approach

Text

- Static text
- Small number of text objects
- High fidelity text rendering (kerning e.g.)
- Text layout (line & letter spacing)

BitmapText

- Dynamic text
- Large number of text objects
- Lower memory

HTMLText

- Static text
- Need that HTML formatting

Textures

We're slowly working our way down from the high level to the low. We've talked about the scene graph, and in general about display objects that live in it. We're about to get to sprites and other simple display objects. But before we do, we need to talk about textures. In PixiJS, textures are one of the core resources used by display objects. A texture, broadly speaking, represents a source of pixels to be used to fill in an area on the screen. The simplest example is a sprite - a rectangle that is completely filled with a single texture. But things can get much more complex.

Life-cycle of a Texture

Let's examine how textures really work, by following the path your image data travels on its way to the screen.
Here's the flow we're going to follow: Source Image > Loader > BaseTexture > Texture

Serving the Image

To start with, you have the image you want to display. The first step is to make it available on your server. This may seem obvious, but if you're coming to PixiJS from other game development systems, it's worth remembering that everything has to be loaded over the network. If you're developing locally, please be aware that you *must* use a webserver to test, or your images won't load due to how browsers treat local file security.

Loading the Image

To work with the image, the first step is to pull the image file from your webserver into the user's web browser. To do this, we can use `Assets.load('myTexture.png')`. `Assets` wraps and deals with telling the browser to fetch the image, convert it and then let you know when that has been completed. This process is *asynchronous* - you request the load, then time passes, then a promise completes to let you know the load is completed. We'll go into the loader in a lot more depth in a later guide.

```
const texture = await Assets.load('myTexture.png');

// pass a texture explicitly
const sprite = new Sprite(texture);
// as options
const sprite2 = new Sprite({texture});
// from the cache as the texture is loaded
const sprite3 = Sprite.from('myTexture.png')
```

TextureSources Own the Data

Once the texture has loaded, the loaded `<IMG>` element contains the pixel data we need. But to use it to render something, PixiJS has to take that raw image file and upload it to the GPU. This brings us to the real workhorse of the texture system - the TextureSource class. Each TextureSource manages a single pixel source - usually an image, but can also be a Canvas or Video element. TextureSources allow PixiJS to convert the image to pixels and use those pixels in rendering. In addition, it also contains settings that control how the texture data is rendered, such as the wrap mode (for UV coordinates outside the 0.0-1.0 range) and scale mode (used when scaling a texture).

TextureSource are automatically cached, so that calling `Texture.from()` repeatedly for the same URL returns the same TextureSource each time. Destroying a TextureSource frees the image data associated with it.

Textures are a View on BaseTextures

So finally, we get to the `Texture` class itself! At this point, you may be wondering what the `Texture` object *does*. After all, the BaseTexture manages the pixels and render settings. And the answer is, it doesn't do very much. Textures are light-weight views on an underlying BaseTexture. Their main attribute is the source rectangle within the TextureSource from which to pull.

If all PixiJS drew were sprites, that would be pretty redundant. But consider SpriteSheets. A SpriteSheet is a single image that contains multiple sprite images arranged within. In a Spritesheet object, a single TextureSource is referenced by a set of Textures, one for each source image in the original sprite sheet. By sharing a single TextureSource, the browser only downloads one file, and our batching renderer can blaze through drawing sprites since they all share the same underlying pixel data. The SpriteSheet's Textures pull out just the rectangle of pixels needed by each sprite.

That is why we have both Textures and TextureSource - to allow sprite sheets, animations, button states, etc to be loaded as a single image, while only displaying the part of the master image that is needed.

Loading Textures

We will discuss resource loading in a later guide, but one of the most common issues new users face when building a PixiJS project is how best to load their textures.

here's a quick cheat sheet of one good solution:

1. Show a loading image
2. Use Assets to ensure that all textures are loaded
3. optionally update your loading image based on progress callbacks
4. On loader completion, run all objects and use `Texture.from()` to pull the loaded textures out of the texture cache
5. Prepare your textures (optional - see below)
6. Hide your loading image, start rendering your scene graph

Using this workflow ensures that your textures are pre-loaded, to prevent pop-in, and is relatively easy to code.

Regarding preparing textures: Even after you've loaded your textures, the images still need to be pushed to the GPU and decoded. Doing this for a large number of source images can be slow and cause lag spikes when your project first loads. To solve this, you can use

the Prepare plugin, which allows you to pre-load textures in a final step before displaying your project.

Unloading Textures

Once you're done with a Texture, you may wish to free up the memory (both WebGL-managed buffers and browser-based) that it uses. To do so, you should call `destroy()` on the BaseTexture that owns the data. Remember that Textures don't manage pixel data!

This is a particularly good idea for short-lived imagery like cut-scenes that are large and will only be used once. If a texture is destroyed that was loaded via `Assets` then the assets class will automatically remove it from the cache for you.

Beyond Images

As we alluded to above, you can make a Texture out of more than just images:

Video: Pass an HTML5 `<VIDEO>` element to `TextureSource.from()` to allow you to display video in your project. Since it's a texture, you can tint it, add filters, or even apply it to custom geometry.

Canvas: Similarly, you can wrap an HTML5 `<CANVAS>` element in a BaseTexture to let you use canvas's drawing methods to dynamically create a texture.

SVG: Pass in an `<SVG>` element or load a .svg URL, and PixiJS will attempt to rasterize it. For highly network-constrained projects, this can allow for beautiful graphics with minimal network load times.

RenderTexture: A more advanced (but very powerful!) feature is to build a Texture from a RenderTexture. This can allow for building complex geometry using a Geometry object, then baking that geometry down to a simple texture.

Each of these texture sources has caveats and nuances that we can't cover in this guide, but they should give you a feeling for the power of PixiJS's texture system.

Check out the render texture example code.