Web Version Unsubscribe

Welcome to Issue 130 of Phaser World

The newsletter had last week off, as I need a little break after the massive releases of Phaser 3.13 and 3.14 but here's another issue packed full of Phaser content for you! Below you'll find details about the first official Phaser sticker pack. They look great on laptops and you'll be supporting open source at the same time as kitting out your gear :) If you'd like some please get your orders in quick as I'm only doing a single print run this year.

Until the next issue keep on coding. Drop me a line if you've got any news you'd like featured by simply replying to this email, messaging me on Slack, Discord or Twitter.

Phaser Sticker Packs

I'm pleased to announce that the Phaser sticker packs are now finally a reality! I've had the first batch through from the printers and I'm over the moon with how they have turned out :)

In the pack you'll get 10 high-quality durable vinyl stickers, perfect for a laptop, or plastering all over your office or geek den. You'll also get a Phaser magnet and finally a nice glossy print of the beautiful pixel art Phaser 3 mech.

The stickers are available to pre-order today.

Pre-orders are running until the end of October, after which they will be printed and shipped. I have to get them printed in bulk to make it worthwhile, so please order early if you'd like to reserve some! Once the stock has run out, that's it.

If you support Phaser on Patreon then please check your personal messages as I have sent you all a custom discounted ordering link.

The Latest Games

Game of the Week

Transmorpher 3

Shape shift between three unique aliens as they work together to solve the puzzles in this addictive platformer.

Staff Pick

Battleships Armada

A beautifully presented remake of the classic board game. Sink their ships before they sink yours!

Farms and Polders Tycoon

Manage a city near the sea, building sea walls and looking after the people in order to make your home prosperous.

Snot Kid

Shoot snot bubble's at invading insects and prevent them from entering through the window crack. You can even charge up a powerful burp!

Exquisite Corpse

What happens when six people work on a game, one person at a time, with nobody knowing what they would do until they got it?

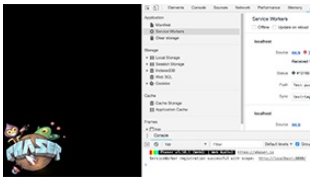
What's New?

Phaser 3.14.0 Released

The latest release includes some important fixes, Tiled 1.2 and multi tileset support, Matter physics updates and lots more!

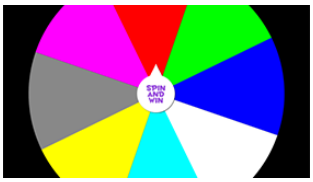
Phaser CE v2.11.1 Released

The latest version of Phaser CE is now available with updates including compressed texture support, audio pause fixes and more.



Progressive Web Apps Tutorial

Create Progressive Web Apps and Offline-First Phaser games in this tutorial.

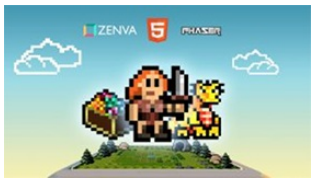


Wheel of Fortune Tutorial Part 1

Build a wheel of fortune for your Phaser games in only a few lines. Draw the wheel on the fly with large room for customization.

Wheel of Fortune Tutorial Part 2

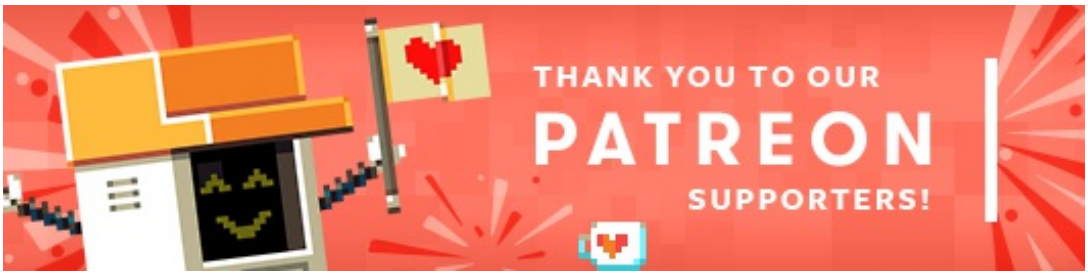
In this part of the tutorial series you learn how to add gradients, rings and an inertia effect to your wheel.



Phaser 3 Game Development Course

A complete Phaser 3 and JavaScript Game Development package. 9 courses, 119 lessons and over 15 hours of video content. Learn to code and create a huge portfolio of cross platform games.

Try this 4 course sample for free!



Thank you to these awesome Phaser Patrons who joined us recently:

Stanislav Salnikov
ScannerDarkly
Facundo Estevez

Thank you to **Paul McGarry (Rubble Games)** and **Richard Hebert** for increasing their pledges :)

Also, a massive thank you to **Quest AI** for their extremely generous donation towards the project.

Patreon is a way to contribute towards the Phaser project on a monthly basis. This money is used *entirely* to fund development costs and is the only reason we're able to invest so much time into our work. You can also donate via PayPal.

Backers get forum badges, discounts on plugins and books, and are entitled to free monthly coding support via Slack or Skype.

Dev Log #130

We've a huge Dev Log this issue. As well as an update on the Doc Jam, there is news about the 3.14 release, the new Spine plugin and to top it all off, a massive guide on what factories are within Phaser 3, how to use them and extend them.

Let's get stuck in ...

Phaser 3 Doc Jam - Write Docs, Win Prizes

I announced the Phaser 3 Doc Jam two weeks ago and since then we have collectively completed the documentation for 1,518 items - and at the time of writing this newsletter there are 469 new entries waiting for me to approve them! This is an incredible amount for such a short space of time. It means (after approval) there are just 1,465 items left before Phaser 3 has 100% API documentation coverage. Could you help get that number even closer to zero?

All you need to do to take part is point your browser to docjam.phaser.io. When the page loads it will randomly pick one of the descriptions that needs writing.

The source code is displayed and the description line highlighted. Take a look at it. Can you figure out what should be written? Maybe it's a property that needs describing? Or perhaps a return value from a method? Look at the surrounding code and see if you can infer the meaning of what's going on. There's a link to open the file on GitHub if you'd like a better look at it.

Everyone who contributes a description, which I then approve, will be automatically entered into a prize draw. You get one entry into the draw for every description that is approved.

At the end of November, I will pick 6 winners. The first two will win a **$100 Amazon gift voucher** each. The remaining 4 will each win a $50 gift voucher. The vouchers will be sent digitally, in time for you to spend on Christmas presents :) This is entirely optional. If you'd like to just help for the sake of helping,

then you don't need to give your email address at all. However, I feel it should make what is quite an arduous task at least a little more fun.

Phaser 3.14 Released

I'm pleased to announce that Phaser 3.14 is now out on GitHub and npm. Hot on the heels of the massive 3.13 release, 3.14 brings some sought-after new features to the party, including support for the new Tiled Map Editor 1.2 file formats, as well as the long-requested feature allowing the use of multiple tilesets per single tilemap layer.

There are also new features to make Matter JS debugging easier and body creation when using lists of vertices is now much cleaner too. It's never just features though. There are lots of important fixes and updates in 3.14, including a fix causing gl canvas resizing to fail, better handling of the game shutdown process and fixes for an issue with Graphics.generateTexture.

If you're building an active project on 3.13 then please upgrade to 3.14 and, as usual, report any issues you find on GitHub so we can look at them immediately.

A few of you asked where the new **Scale Manager** was in the 3.14 release. I had flagged the Scale Manager for 3.14 based on our current roadmap but it has been pushed back to 3.15 now. The reason is that 3.14 was originally just going to be a point release that only contained some important fixes. However, I had also completed work on improvements to the tilemap renderer so merged those in too, which then warranted the major version number bump. This means every version number on the roadmap is now increased by one.

Phaser 3.15 is in active development with the Scale Manager being the headliner feature planned for it. The Containers rewrite will be started in 3.16, and so on.

Phaser 3 Spine Plugin

I know that Phaser support for Esoteric's Spine animation software has been on the 'wanted' list for a long time now. It's something I've always wanted to do but other features had gotten in the way. Now that I'm down to just one core system left to finish (the Scale Manager) I was already thinking about what to work on in the future. Then, an email arrived from a company who use Phaser for a lot of their work. They said they desperately needed Spine support for a forthcoming project and would be willing to help fund it.

Even with an offer of money on the table I still knew it would be a lot of work to undertake. But I didn't know exactly how much. Truth be told, it had been years since I last looked at the Spine runtimes. So, before I agreed to anything I took some time to investigate just how complicated it would be to get the Spine runtimes working within Phaser. Suffice to say the runtimes have evolved dramatically in that time and it didn't take long before I had them rendering with the Phaser canvas renderer. Canvas is easy though, the real test would be WebGL. Thankfully, a couple of hours of hacking around and figuring out the Spine internals, and that was working too!

Knowing that integration of the runtimes with the Phaser renderer was now technically possible, I went back and reported my findings. Panoramik, the company who was offering the sponsorship, delivered on their promise to help fund Phaser Spine support and it's now being worked on. They will benefit because it will allow them to deliver their current project and you will benefit because you'll be able to use Spine animations in your games too.

Just to be clear: I am using the official Spine runtimes for the plugin, there will be no 'custom' code for any part of it other than asset loading and hooking up the renderer states. Most of my work will be providing an interface between native Phaser Game Object's and the skeleton data provided by Spine. This means that when Esoteric publish a new runtime version it should be a trivial case to update Phaser as well. It also means that everything the Spine runtime supports, such as two-color bone tints and mesh deformations, will be supported in the plugin as well. As development progresses I'll publish more news in the Dev Logs, so keep an eye out.

Equally, if reading this has made you think "actually, my studio could really do with this feature", to the point that you're willing to help fund it, then do drop me a line. The money allows me to focus on Phaser, you get a feature you need and everyone wins.

Industrial Revolution - Phaser Factories Guide

Factories have existed in Phaser since version 0, yet they are commonly misunderstood. In this guide, I'm going to take a look at the different types of Factory classes within Phaser, describe what they do and how to bypass them entirely when needed. If you've ever created your own custom class, that extended a Phaser base class and it didn't do anything when instantiated, this guide is for you.

You've almost certainly used a Factory already, even if you weren't aware you were doing it. Take the following code for example:

```
this.add.sprite(100, 300, 'ripley');

this.add.text(100, 200, "These people are here to protect you. They're soldiers.");

this.add.sprite(400, 300, 'newt');

this.add.text(400, 200, "It won't make any difference.");
```

Here we're creating two sprites and two text objects, but it's the use of `**this.add**` that means we're creating them via a Factory class, in this case the Game Object Factory. In a nutshell, factory classes provide an easy way for you to create objects within Phaser and have them ready for use right away.

Without using the Game Object Factory the above 4 lines of code would need to look something like this:

```
var ripley = new Phaser.GameObjects.Sprite(this, 100, 300, 'ripley');

this.sys.displayList.add(ripley);
this.sys.updateList.add(ripley);

var ripleyText = new Phaser.GameObjects.Text(this, 100, 200, "These people are here to protect you. They're
soldiers."));

this.sys.displayList.add(ripleyText);

var newt = new Phaser.GameObjects.Sprite(this, 400, 300, 'newt');

this.sys.displayList.add(newt);
this.sys.updateList.add(newt);

var newtText = new Phaser.GameObjects.Text(this, 400, 200, "It won't make any difference."));

this.sys.displayList.add(newtText);
```

While the extra code required isn't complex, it does, to me at least, get in the way of you understanding the flow of your code. That's a whole lot of repeated operations and this is just with 4 Game Objects in a Scene. It can easily expand to be hundreds of lines of duplicated code. Therefore, at their heart, Factory functions are just friendly wrappers that let you cut down the volume of required code by encapsulating common tasks.

The Game Object Factory

The one factory you are likely to interact with more than any other is the Game Object Factory. This factory registers itself with a Scene under the default alias of `add`. So when you call `this.add` from within a Scene, you're communicating with the Game Object Factory itself. Every Scene has its own instance of a Game Object Factory.

Feel free to take a look at the source code for the class. You'll notice that it's extremely light-weight with hardly any code in it at all. In fact, if you scroll quickly through it, you'll see there isn't a single function called `sprite`, or `text` like we used above. So, where are those coming from?

The answer is that every Game Object has its own Factory *function*. Due to the way Phaser 3 is designed, it was intended from the start that Game Objects were entirely optional. If you didn't need the Text object in your game, for example, then you didn't need to include it in your build. This posed a structural problem, because if the Game Object Factory had a built-in method called `text`, it would have needed the ability to create a Text object, which in turn would have meant it would be bundled into the build, even if never used.

To avoid this, the Factory functions that belong to Game Objects register themselves with the Game Object Factory when Phaser is being created for the first time. You can find these in the repository, for example the Sprite Game Object has a SpriteFactory function which contains this code:

```
GameObjectFactory.register('sprite', function (x, y, key, frame)
{
    var sprite = new Sprite(this.scene, x, y, key, frame);

    this.displayList.add(sprite);
    this.updateList.add(sprite);

    return sprite;
});
```

Here you can see it's calling the `register` function on the factory itself, passing in the key ('sprite') and the function to execute when that is called. So, when you invoke `this.add.sprite` from a Scene, you're in effect communicating with the above function, that has registered itself with the factory. All of the scoping issues and access to things like the display list are handled internally, so you get to make one call in your code, and a fully working Sprite comes out the other end.

It's really not magical, the paper-chain is there to follow quite easily if you look through the code, but it can feel like it if coming from a more traditional JavaScript background where things are often declared or created explicitly.

Skipping the Factory

The thing is, the factories are *entirely optional*. And it's important to understand this point, especially when creating your own Game Object classes. The actions that the factory functions perform are essential for the Game Object but it isn't essential that the factory itself performs them.

For example, let's create our own custom Sprite, the Brain:

```
class Brain extends Phaser.GameObjects.Sprite {

    constructor (scene, x, y)
    {
        super(scene, x, y, 'brain', 'squishyFrame1');
    }

    preUpdate (time, delta)
    {
        super.preUpdate(time, delta);

        this.rotation += 0.01;
    }

}
```

We extend the Sprite class in the usual way, calling the constructor and passing across the `x` and `y` coordinates. So far, so normal. However, if you were to create an instance of this class in your Scene, nothing would appear. This is because at no point does the class add itself to the Scene's Display List, nor to the Update List which manages update handling for animated sprites.

There are two ways to resolve this. Firstly, we could use the factory function `existing`, which allows you to add an existing object to the Scene:

```
function create ()
{
    this.add.existing(new Brain(this, 264, 250));
    this.add.existing(new Brain(this, 464, 350));
    this.add.existing(new Brain(this, 664, 450));
}
```

But as this section is all about how *not* to use the factory, we could also modify our game object constructor to perform the factory function tasks within it:

```
class Brain extends Phaser.GameObjects.Sprite {

    constructor (scene, x, y)
    {
        super(scene, x, y, 'brain', 'squishyFrame1');

        scene.sys.displayList.add(this);
        scene.sys.updateList.add(this);
    }

}
```

Now, you can directly create fresh Brains all over the place:

```
function create ()
{
    new Brain(this, 264, 250);
    new Brain(this, 464, 350);
    new Brain(this, 664, 450);
}
```

As long as your Game Object performs the same actions as the factory function of the class it is based on, it'll work happily within the Scene. Not sure what those actions should be? Just look at the Factory function within the Game Objects folder in the repository.

Of course, if you don't want your custom Game Objects to be added to the Scene as soon as they are instantiated, then you shouldn't put the factory function in their constructors. There are plenty of cases where you may need to create a whole batch of Game Object's and then selectively add them to your Scene, rather than at once up front. In this case, you may wish to add a special method to your class that only runs the factory functions when called.

The Physics Factory

While the Game Object Factory may be the most commonly used, it's not the only factory to be found in Phaser. In fact, anywhere you see the use of `add`, you are talking to a factory. Some others you may be familiar with are the physics factories.

Each physics system has its own factory and just as before they're essentially

wrappers to help you cut-down on repetitive code. The following is how you create an Arcade Physics Sprite via its factory:

```
this.physics.add.sprite(400, 100, 'hudson');
```

`this.physics` maps to the ArcadePhysics class, and the `add` property maps to the Arcade Physics Factory. When you call `add.sprite` it runs the following factory function:

```
sprite: function (x, y, key, frame)
{
    var sprite = new ArcadeSprite(this.scene, x, y, key, frame);

    this.sys.displayList.add(sprite);
    this.sys.updateList.add(sprite);

    this.world.enableBody(sprite, CONST.DYNAMIC_BODY);

    return sprite;
}
```

You can see that the contents of this function are very similar to the Sprite's Game Object factory function. Again, the Sprite is being added to the Display List and Update List, except this time it's also being passed to the Arcade Physics world instance to have its physics body enabled, before being returned.

The actions performed by the factory functions are essential. This is why if you merely create an instance of an Arcade Sprite in your Scene, absolutely nothing will happen:

```
new ArcadeSprite(this, 400, 100, 'brain', 'squishyFrame1');
```

While this has indeed created the Sprite object, it hasn't added it to the Display List, or the Update List, or given it a physics body. This is why if you wish to create custom classes that extends a base physics class you need to perform the operations that the factory does on its behalf. Here's our Brain class from before, extended to be an Arcade Sprite:

```
class Brain extends Phaser.Physics.Arcade.Sprite {

    constructor (scene, x, y)
    {
        super(scene, x, y, 'brain', 'squishyFrame1');
    }

}
```

In this form it won't render or have a physics body. As before, you can pass it to the factories `existing` method:

```
this.physics.add.existing(new Brain(this, 400, 100));
```

Which will make sure it is added to the correct lists and is given a physics body. But, what if we want the Brain to automatically have some horizontal velocity after it has been created? It would be tempting to alter the constructor, adding in a call to `setVelocity` like so:

```
class Brain extends Phaser.Physics.Arcade.Sprite {

    constructor (scene, x, y)
    {
        super(scene, x, y, 'brain', 'squishyFrame1');

        this.setVelocity(200, 0);
    }

}
```

Except, this will throw a runtime error because `setVelocity` expects that the Game Object has a physics body, and at this point in its life it doesn't, because it hasn't yet been passed to the `enableBody` function.

To resolve this we need the constructor to perform the factory function tasks for us again:

```
class Brain extends Phaser.Physics.Arcade.Sprite {

    constructor (scene, x, y)
    {
        super(scene, x, y, 'brain', 'squishyFrame1');

        scene.sys.displayList.add(this);
        scene.sys.updateList.add(this);
        scene.sys.arcadePhysics.world.enableBody(this, 0);

        this.setVelocity(200, 0);
    }

}
```

Now, if we create an instance of Brain in our Scene, it'll render, update, have a physics body and start moving. Essentially, we've moved the actions of the factory to within our class. You need to go through this same process for anything that has a factory function, regardless of what it is, if you create your own classes based on them. It's not a complicated process, it only takes a few lines of code, but it is important and requires you get a little bit familiar with the Phaser source code. If you'd absolutely rather not have to do that, you could modify the above class to this:

```
class Brain extends Phaser.Physics.Arcade.Sprite {

    constructor (scene, x, y)
    {
        super(scene, x, y, 'brain', 'squishyFrame1');

        scene.add.existing(this);
        scene.physics.add.existing(this);

        this.setVelocity(200, 0);
    }

}
```

Which has the same end result, it just requires one additional jump to achieve it.

Creating your own Game Object Factory functions

You can register your own Game Object's with the factory, allowing you to create

instances of them directly in your Scene. To do this you have to call the `register` method and pass in a callback for it.

First, here is our custom class. It extends Sprite, has a pre-defined texture and will automatically rotate during its update:

```
class EnemyRobot extends Phaser.GameObjects.Sprite {

    constructor (scene, x, y)
    {
        super(scene, x, y);

        this.setTexture('contra');
        this.setScale(2);
    }

    preUpdate (time, delta)
    {
        super.preUpdate(time, delta);

        this.rotation += 0.01;
    }

}
```

Here is the code we use to set-up our example. You'll notice that inside of the `init` function we are registering our callback with the Game Object Factory. All this does is create an instance of the EnemyRobot class and add it to the required lists. Notice the string we're using is `robot`. If you were to provide a string already in use by the factory it would skip the call, so make sure your strings are unique:

```
let config = {
    type: Phaser.AUTO,
    parent: 'phaser-example',
    width: 800,
    height: 600,
    pixelArt: true,
    scene: {
        init: init,
        preload: preload,
        create: create
    }
};

let game = new Phaser.Game(config);

function init ()
{
    Phaser.GameObjects.GameObjectFactory.register('robot', function (x, y)
    {
        let sprite = new EnemyRobot(this.scene, x, y);

        this.displayList.add(sprite);
        this.updateList.add(sprite);

        return sprite;
    });
}

function preload ()
{
    this.load.image('contra', 'assets/pics/contra3.png');
}
```

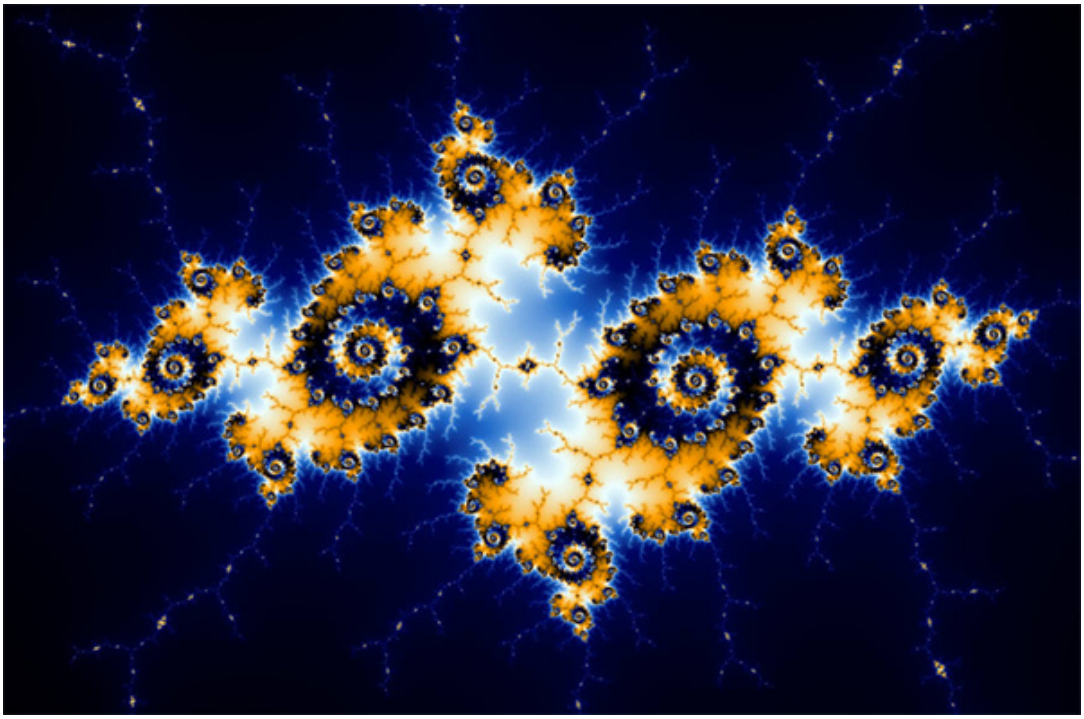
With this done, we can now call `add.robot` from within our Scene:

```
function create ()
{
    this.add.robot(200, 200);
    this.add.robot(400, 300);
    this.add.robot(600, 400);
}
```

And voila...

We have our custom class appearing, rendering and updating, created via a custom factory function. Click the screen shot to see the full code.

Hopefully after reading this guide to factories you now understand a little more about the architecture behind Phaser and how to adapt it for your own needs.



A really nice introduction to Generative Art.

What if modern internet companies existed in the 1970s?

A fun video explaining how the ghosts in the game Pacman work.

Phaser Releases

Phaser 3.14.0 released October 1st 2018.
Phaser CE 2.11.1 released October 2nd 2018.

Please help support Phaser development

Have some news you'd like published? Email support@phaser.io or tweet us.

Missed an issue? Check out the Back Issues page.

Preferences Forward