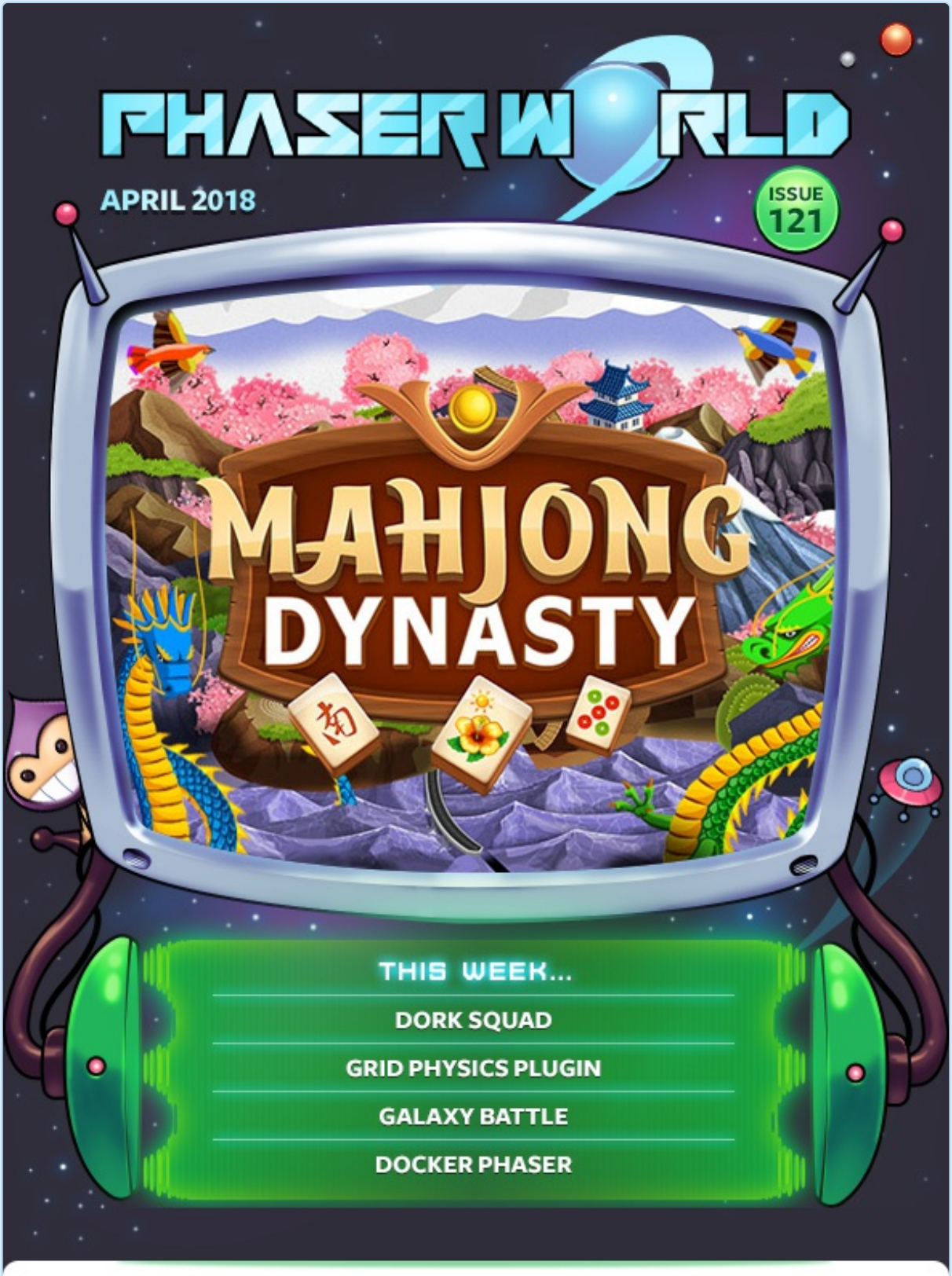


Welcome to Issue 121 of Phaser World

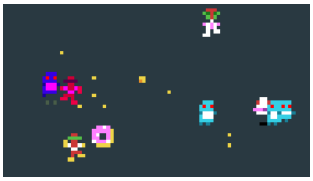
We're getting back on schedule :) and this is a monster issue too! Some great games and tutorials as usual, including a beautiful mahjong game and frantic

retro blaster. The main bulk of this issue though is the Dev Log which contains Part 2 of our comprehensive Phaser 3 Scenes Guide, with some pretty neat demos showcasing what can be done. I hope you enjoy it and find it useful, as an awful lot of work went into writing it.

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.

The Latest Games

Game of the Week
Mahjong Dynasty
Journey through Asia, matching tiles across multiple levels in this beautiful mahjong game.



Staff Pick
Dork Squad
Robots have taken over. Only a small squad of dorks remain standing. Can you save the day?

Broken Space Fish Shooter

Can you hack this game tweak the control scheme in order to survive?

Galaxy Battle

A horizontally scrolling shoot-em-up with friendly mobile controls.

Sokoban

A nice little Sokoban ban with a built-in level editor.

What's New?

Developer Q&A

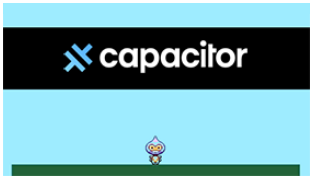
Got a question about Phaser? Need us to create an example or explain a function in more detail? Take part in our Q&A!

Grid Physics Plugin

This plugin adds support for grid / tile based movement, collision, bodies and more.

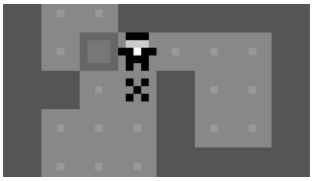
Docker Phaser

A quick and easy Phaser environment for anywhere Docker runs.



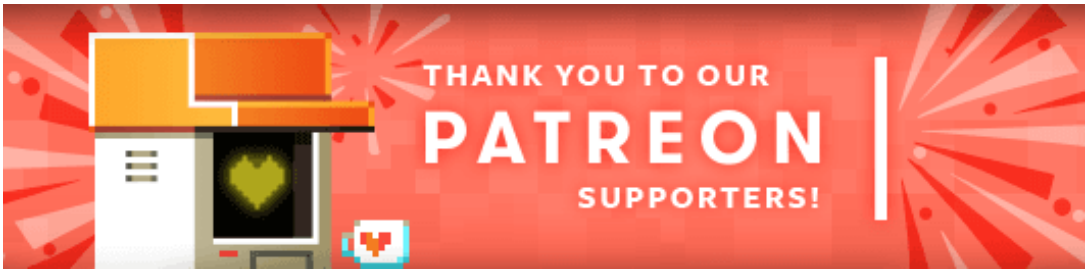
Capacitor and Phaser Tutorial

A tutorial on creating native HTML5 games with Phaser 3 and Capacitor.



Sokoban Tutorial Update

The Sokoban tutorial gets another update to add unlimited undos, camera effects, double tap and more.



Thank you to our awesome new Phaser Patrons who joined us recently:

Björn Hjorth
Jacob Grahn
Isaac Pante

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 #121

Welcome to another new Dev Log. I'm really pleased to say that I have finished the second part of the Phaser 3 Scenes Tutorial. The first part was published in Phaser World issue 119 and is available on our site. I'd strongly recommend reading it before digging into Part 2, as I'll assume you are up to speed and will just dive right in. Part 2 covers the Scene List and Scene rendering.

Before we get into the world of Scenes I just wanted to say that Phaser 3.7.0 is coming on really nicely and I anticipate a release this week. I've been busy refactoring the Loader Plugin and File Types. The File Types are now a lot more flexible and entirely self-contained, allowing us to strip them out of custom builds, or create entirely new ones as needed. The Loader has been restructured slightly and will be gaining the ability to modify the load queue during the in-flight process, bringing back the old v2 favorite Loader Packs in the process. These are json files that contain lists of files to load, and the loader will soon be able to ingest them and respond to their contents. This also means we can fully support multi-atlas loading, so you just specify the path to the json and it will inspect the file and load the images dynamically. It's a great update, and as with all updates, I'll have the documentation for the Loader 100% completed with it as well.

Once the Loader work is finished I'll release 3.7.0 and then move onto Input enhancements for 3.8.0. This will focus on multi-touch, pixel-perfect input, custom cursors and multi-shape hit areas - and naturally, I'll complete the docs as I go. As always, keep an eye on the Change Log if you want to get a sneak-peak of what's coming down the pipe. For now, though, it's time to dive into the second part of our Scenes tutorial. And things are about to get *really* interesting, and just a little bit retro in places.

Scene Updating and Rendering

We mentioned that the Game owns the truly global classes - two of these are the renderer and the Scene Manager. When the game steps, usually as a result of a Request Animation Frame update in the browser, it will tell the Scene Manager to update. The Scene Manager is responsible for looking after all Scenes in your game. It can add new ones, start and stop them and also takes direction from the game as to when to update them.

Scenes are stored in the Scene Manager in a list. The order of the Scenes in this list control two important factors. The first is that it controls the order in which Scenes are rendered. The second is that it controls the order in which Scenes are updated.

Each game step the Scene Manager will iterate through all of its Scenes in *reverse* order, updating each one of them in turn. It first checks to see if the Scene is active (i.e. hasn't been sent to sleep or shut down) and then tells the Scene Systems to update. This in turn tells all Scene plugins to update, which tells all Game Objects to update, and so on. Essentially, everything your Scene does that isn't related to rendering happens at this point.

Once the update loop has completed, the Scene Manager then iterates through all Scenes, from back to front, rendering each one. It does this by passing a reference to the Renderer to each Scene, which in turn hands it to the Scene Camera Manager to deal with. The Camera Manager takes the renderer and constructs the visual display of the Scene upon it, factoring in multiple cameras, everything they can see and any special effects running on them.

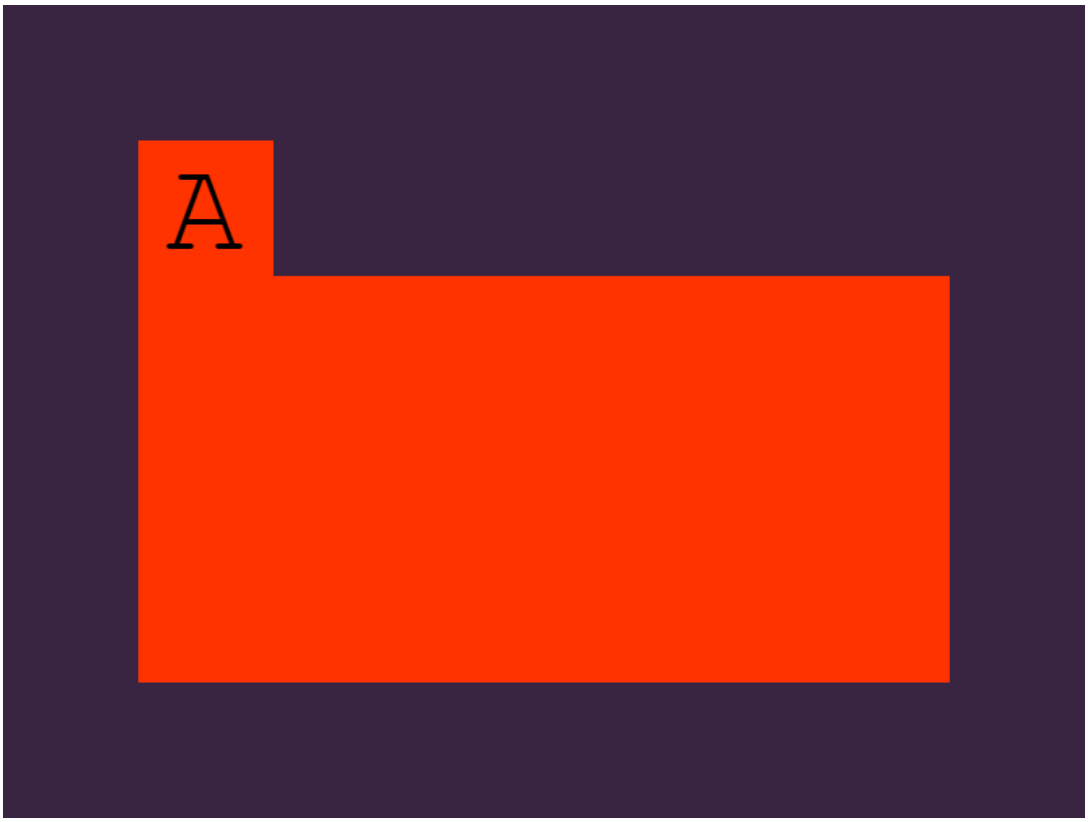
Scene List

The drawing order is important. If you add three Scenes to the Scene Manager, it will draw them in the order in which they were *added*, from back to front, meaning the first Scene added would be at the very back of the display, and the most recent Scene added would draw over the top of it, at the front. It works like a standard display list and can be controlled as such, which we'll cover in the next section. Here's a simplified example showing the effect of adding multiple Scenes.

Here's our first Scene. All it does is draw a rectangle with a letter on it, but it's enough to demonstrate what is going on:

```
class SceneA extends Phaser.Scene {

    constructor ()
    {
        super({ key: 'SceneA', active: true });
    }

    create ()
    {
        let graphics = this.add.graphics();

        graphics.fillStyle(0xff3300, 1);

        graphics.fillRect(100, 200, 600, 300);
        graphics.fillRect(100, 100, 100, 100);

        this.add.text(120, 110, 'A', { font: '96px Courier', fill: '#000000' });
    }
}

let config = {
    type: Phaser.AUTO,
    width: 800,
    height: 600,
    backgroundColor: '#392542',
    parent: 'phaser-example',
    scene: [ SceneA ]
};

let game = new Phaser.Game(config);
```

If you run the code as-is you'll see this:



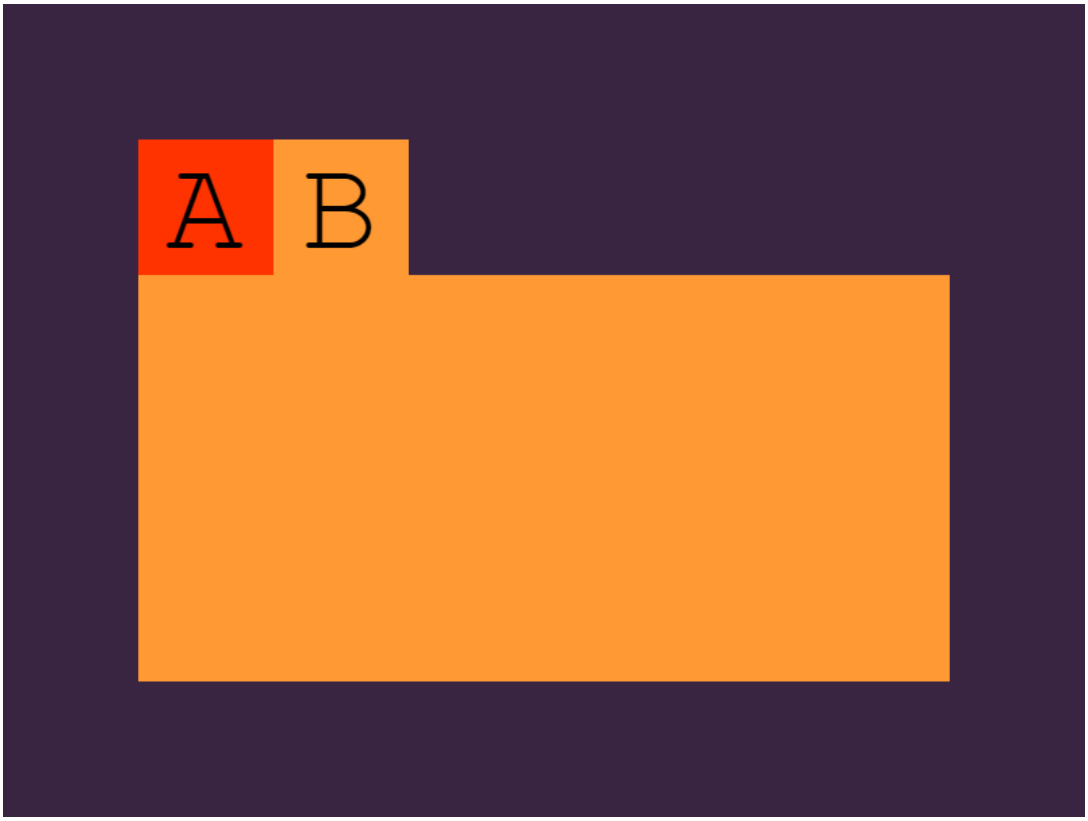
Now, let's add another Scene. It's going to do the same again but with a different color:

```
class SceneB extends Phaser.Scene {

    constructor ()
    {
        super({ key: 'SceneB', active: true });
    }

    create ()
    {
        let graphics = this.add.graphics();

        graphics.fillStyle(0xff9933, 1);

        graphics.fillRect(100, 200, 600, 300);
        graphics.fillRect(200, 100, 100, 100);

        this.add.text(220, 110, 'B', { font: '96px Courier', fill: '#000000' });
    }
}
```

We need to update the scene array in the Game Config to add in this new Scene:

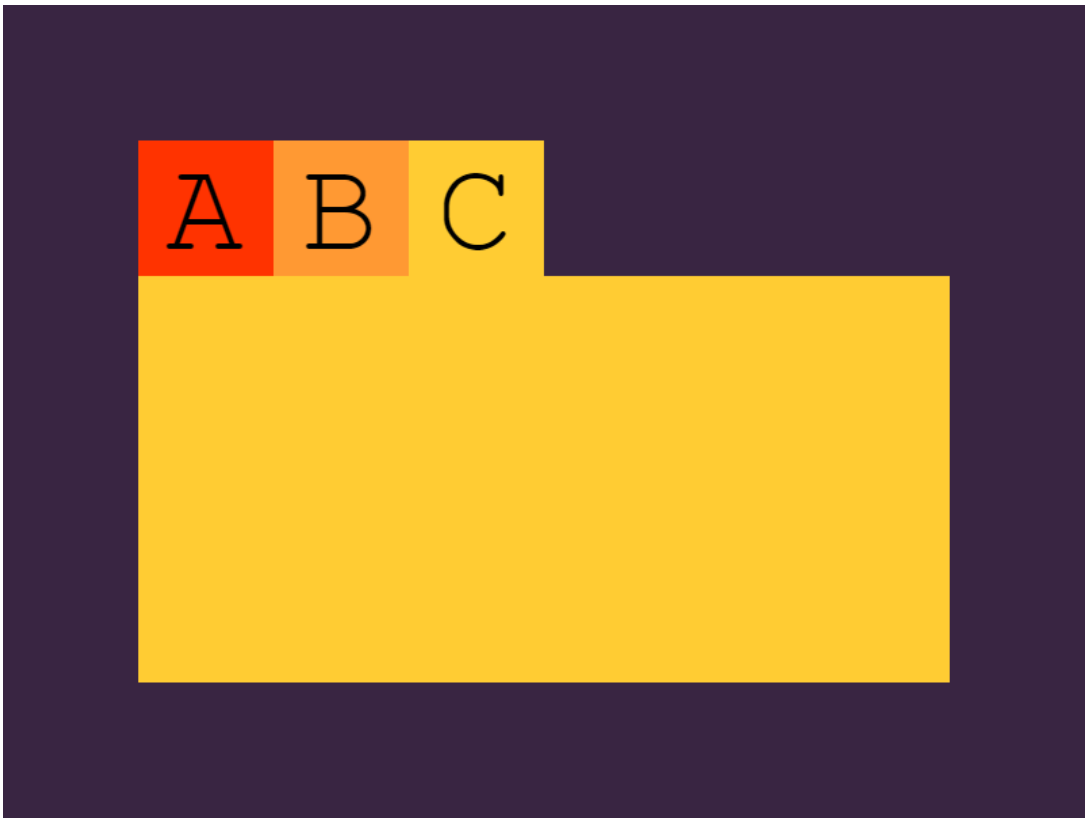
```
scene: [ SceneA, SceneB ]
```

and the result is, as you'd expect, this:



Scene B appears over the top of Scene A because it has been added second in the Scene array passed in the Game Config. If you were to swap the order in this array, Scene A would appear on the top.

We'll add one final Scene just to complete the effect. Scene C is exactly the same as before. Our final order looks like this:



It looks like this because we used this array in the Game Config: **[SceneA, SceneB, SceneC]**

Swap any of those elements around and you'll impact the order in which they are rendered and updated. Of course, you're not limited to just the order in which you specify the Scenes in your config. You can manipulate them using the ScenePlugin, which we'll demonstrate fully in a moment.

Scene List Input

You may wonder why the Scene Manager iterates through the Scenes in reverse order when updating them. The reason is to do with Input. If you look at the image above you can see that our UI Scene is the top-most Scene in the display, because naturally we'd want the UI to sit over the top of the game itself. But imagine you have an interactive object in your game, such as a clickable area, and the UI overlaps with it. You'd nearly always want the UI to receive the input event, not the interactive object it is overlapping. This is why we iterate in reverse, so the Input Plugin has a chance to process the top-most items on the display list *and* the top-most Scenes on the scene list first.

By default, if a Scene at the top of the scene list receives *and handles* a valid input request, then all of the Scenes below that one will just skip their input processing in order to save time. You can change this behavior by calling **this.input.setGlobalTopOnly(false)** from any Scene. Every Scene will then

process input, regardless of its position in the scene list.

By iterating in this order we're also able to have a UI Scene control the game scene immediately. For example, you could have a Pause icon, and clicking it would have a chance to tell the game to pause straight away, without having to wait for the next frame first. It just makes your game feel more responsive.

Controlling Scene Order

The Scene Plugin has a selection of methods that allow you to precisely control the order of the Scenes in the scene list. This is crucial because their position in the list dictates the order in which they are drawn. It's not always possible to know in advance the order of your scenes when creating them, or which scenes are going to be running at the same time, so it's vital you can easily control their order.

It's worth stating that all interactions with Scenes should be done via the Scene Plugin. Every Scene owns an instance of this core plugin and it acts as an interface to the Scene Manager. You should rarely, if ever, access the Scene Manager directly. If you find yourself calling methods on the Scene Manager then change your code, because it likely has structural problems that will grow over time.

In short, use `this.scene` and **not** `this.game.scene`. I cannot iterate this enough.

The Scene Plugin has a lot more methods available on it. For example, you can do `this.scene.restart`, or `this.scene.launch` from the Scene Plugin, but *not* from the Scene Manager. If you try calling a method and it complains it doesn't exist, double-check you're not going in via `game` by mistake, and correct your approach.

Moving Scenes

There are 6 main commands you can use to arrange the Scene List:

bringToTop()
sendToBack()
moveUp()
moveDown()
moveAbove()
moveBelow()

You access these methods via the Scene Plugin:

```
this.scene.bringToTop();
```

If you do not provide an argument then the action takes place on the current Scene, i.e. the one from which you called the method. So, the code above will try and move the current Scene to the top of the Scene List (assuming it is not already there.)

You can optionally provide a Scene to move:

```
this.scene.bringToTop('SceneB');
```

This will try and move the Scene referenced by the key `SceneB` to the top of the Scene List. As well as a string you could also pass in a reference to the target Scene itself.

The `moveAbove` and `moveBelow` methods require you to provide a target Scene:

```
this.scene.moveAbove('Game', 'PauseScreen');
```

The code above will move the `PauseScreen` Scene to be above the `Game` Scene in the Scene List. If you omit the 2nd argument then it would try to move the current (calling) Scene to be above the `Game` Scene.

Scene Ordering Demo

Using just these 6 methods you can construct complete Scene layouts. Let's make an example to demonstrate this in action, as it's the kind of thing that is easier to see running than to read about.

Dull colored boxes, like in our previous example, are all good and well but it's time to spice things up. We're going to create an example with 7 Scenes. One of them will be our Controller Scene. This Scene will house the UI that you can use to manipulate the other Scenes. The other scenes will be unique graphical

effects. Using a set of sci-fi game graphics we'll create Scenes for a rotating nebula, a glowing sun, an asteroid field, a planet, a space ship with an exhaust trail and some spinning space mines.

All of the code and assets for this example can be found in the Phaser 3 Examples repo. Because there is quite a lot of code I'm only going to cover the parts that are directly related to moving Scenes, but it's well worth digging through the files to get some ideas for other aspects of how Phaser 3 works.

The UI will allow you to select a Scene, then move it around the Scene List, toggle its active state and toggle visibility. Here is what it looks like:

The buttons on the left allow you to select a Scene. The one with the light on is the current Scene, the name of which is also displayed in the LCD screen. Below it, you can see which index it currently holds in the Scene List. In the image above the Nebula Scene is index 1, which means it is rendering right at the back. Using the d-pad you can adjust this position. Finally, the two toggles on the right control the visible and active state of the selected Scene.

All of the code for this UI can be found in the Controller.js file. It's nothing more than a few images and input events but you'll see in the events how we link from the Controller Scene to the target scene.

As you've seen in the UI there are 6 Scenes to select from. In the following screen shot you can see them all running together. Click the shot for a link to the example so you can play with it:

By using the controls you can re-arrange the Scenes. We modified it so we've hidden the Nebula entirely, bought the sun right to the front and made the ship fly behind the planet. See what combinations you can create.

Play with the controls and view the source until you've got a good feel for how the Scene List works.

Spawning Multiple Scenes

Things are going to get a bit crazier now. So far we have worked on the basis of one class = one Scene. However, that doesn't have to be the case. You can actually spawn multiple versions of the same Scene over and over again, all running at the same time if you so wish.

Here's a basic Scene class called `Spawn`:

```
class Spawn extends Phaser.Scene {

    constructor (handle)
    {
        super(handle);
    }

    create ()
    {
        this.add.image(0, 0, 'phaserRulez');
    }

}
```

Rather than adding this to the scene array in the Game Config, let's create an instance of it at run-time:

```
let key = 'Spawn1';
let spawned = new Spawn(key);

this.scene.add(key, spawned, true);
```

We define a key, create a new Spawn class and then add it to the Scene Manager (the final argument tells the Scene Manager to run it immediately)

Now, imagine that in a loop:

```
for (let i = 0; i < 10; i++)
{
    let key = 'Spawn' + i;
    let spawned = new Spawn(key);

    this.scene.add(key, spawned, true);
}
```

Suddenly you've got 10 copies of the Spawn Scene running. You can get to any of them using their key:

```
let ref = this.scene.get('Spawn5');
```

Once you have a reference to the Scene you can control it just like any other. Move it around the Scene List, pause it, hide it and so on.

There's one more element we need to cover before we can build our demo showing this all off, and that's how Scenes render.

Scene Rendering

We've already covered in detail that the order of the Scene on the Scene List controls when it renders, and it's visible state controls if it renders at all. However, there's one last element to consider: the visual size of the Scene.

By default a new Scene will be the same size as your game. It will create its own Camera Manager and the default camera will be set to your game dimensions. This is fine in lots of cases, but not all. You can change that behavior by setting the size of the default camera:

```
class MiniScene extends Phaser.Scene {

    constructor (handle)
    {
        super(handle);
    }

    create ()
    {
        this.cameras.main.setViewport(0, 0, 300, 200);
    }

}
```

The above code tells our Scene that it has one camera that is 300 x 200 pixels in size and top-left positioned at 0 x 0. We can now add whatever we like to the Scene: graphics, images, animations, physics, but they will only render within the 300 x 200 window. Everything else gets clipped away.

In this code, the Scene is a fixed size and in a fixed position. It can be moved by simply moving the main camera:

```
this.cameras.main.setPosition(128, 256);
```

Now the Scene will render at 128 x 256. It is still 300 x 200 in size, we only changed where within the game canvas it is rendering at. There are no restrictions on where the Scene can be placed (within reason) and they can be stacked on-top of each other as much as you like.

If you factor in these two things: That Scenes can have a size other than the game canvas size, and you can position them anywhere, you should start to get an inkling of what's possible. Which leads us nicely onto our final demo.

Multi Scene Demo

In order to demonstrate the ability to spawn multiple Scenes and move them around the canvas I'm going to create a mock desktop environment, leaning on my love of retro and borrowing the Amiga workbench) as my source of inspiration. As with the previous demo everything has been split into single files and is available in the examples repo, so you can pick it apart at will.

The concept for this demo is as follows:

You will see an emulated Amiga desktop with a single disk icon. Clicking the icon will open a window containing six different demos. Clicking any of the icons will launch that demo. You can click the demo icons as many times as you like, it will just keep on spawning copies of the demos all over the screen. You can drag any of the windows around and doing so brings the window to the top of the Scene List. Finally, the demo will run at the size of the browser and will respond to resize events.

Although visually there are controls for resizing and closing the windows we're not going to implement that here for the sake of brevity. Feel free to do so if you wish though, and make sure to send me a link to your demo :)

The six different demos you can launch are:

1. **Eyes**
   -
   a
   pair
   of
   googly
   eyes

that follow the pointer around the screen. If you spawn a bunch of these it can get quite surreal!

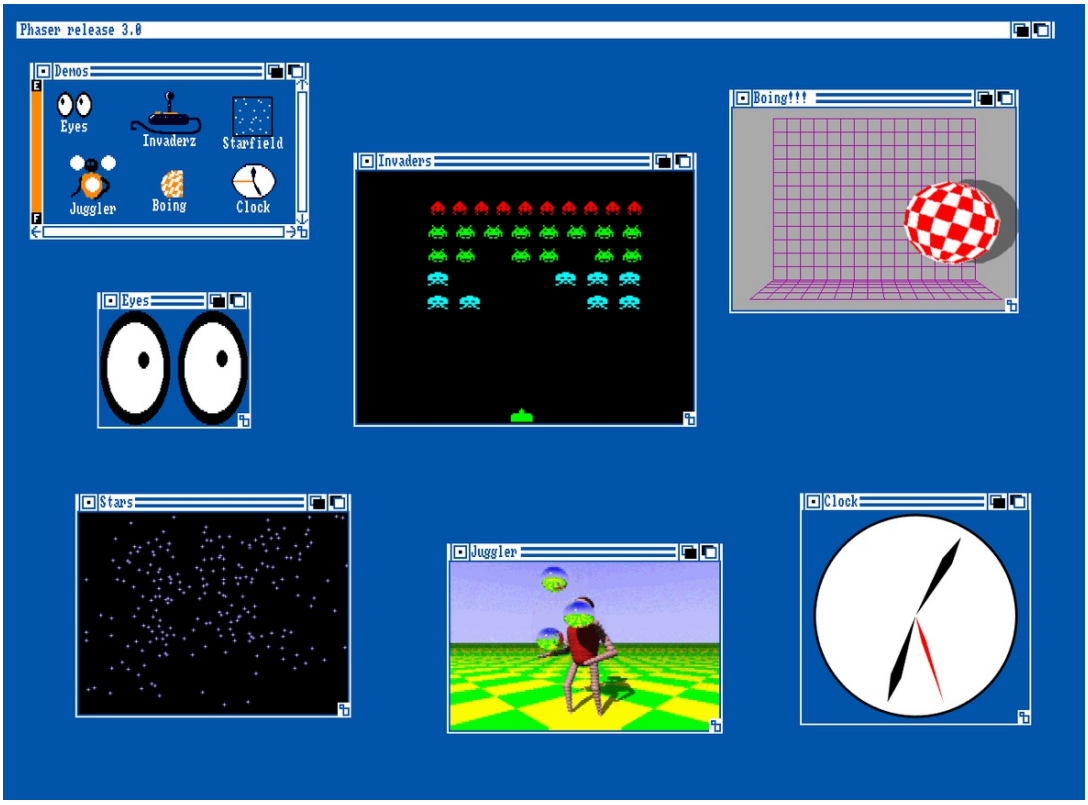
2. **Starfield** - a classic demo effect using a Blitter to draw some stars.
3. **Juggler** - an example of a looped animation playing.
4. **Boing** - a small physics

demo. The ball is bouncing within an Arcade Physics world set to the size of the window.

5. **Clock** - a real clock based on your system time.
6. **Invaders** - a playable Space Invaders game. Use the left/right cursors to control the ship. It will automatically fire.

Spawn a whole bunch of these and you can play them all at once!

Finally, here is a screen shot of each of the demos running. Click it to open the demo so you can experience it for yourself:



Honestly, I'm quite pleased with this :) It's important to remember that every little window that opens is like its own Phaser Game instance in its own right, isolated from everything else.

The above demo even runs on tablets, and likely phones too although the screen would be tiny. You can't play the invaders game on mobile but even so, you can

spawn demos and drag them around. It's quite surprising how many you can get running before it starts to suffer.

If you look at the code for each Scene you'll see they are fundamentally quite simple. There's very little in them that you will not have encountered before. The Controller is the Scene in charge of handling the 'desktop' and preloading all of the demo assets. It does all of the grunt work and then manages the windows as they open up. As with the Scene Ordering demo there's very little in the Controller beyond images and input events, although there is one thing worth covering in more detail: How to drag the Scenes around.

How the Scene dragging is done

There are lots of different ways you could achieve this effect, but for the sake of this demo I went with the following. When you click a demo icon, such as the Juggler, it calls this function:

```
createWindow (func)
{
    var x = Phaser.Math.Between(400, 600);
    var y = Phaser.Math.Between(64, 128);

    var handle = 'window' + this.count++;

    var win = this.add.zone(x, y, func.WIDTH, func.HEIGHT).setInteractive().setOrigin(0);

    var demo = new func(handle, win);

    this.input.setDraggable(win);

    win.on('drag', function (pointer, dragX, dragY) {

        this.x = dragX;
        this.y = dragY;

        demo.refresh()

    });

    this.scene.add(handle, demo, true);
}

this.createWindow(Juggler);
```

We pass in a reference to the Scene class we wish to spawn. It then assigns it a new key (i.e. `window7`) and creates a Zone Game Object within the Controller Scene. This Zone is in the same position and the same size as the Scene that was just created. It is set to be interactive and draggable. Finally, we listen for its drag event and then call the `refresh` function on the Scene it is linked to. All `refresh` does is call Camera setPosition and brings the Scene to the top.

Zones are non-rendering Game Objects. They have no visual attributes and don't

take up any rendering time - but you can still make them interactive and click or drag them and we utilize this fact to link the invisible Zone to our visible Scene.

It's not authentic because it means you can drag the window from any point, not just from the title bar like in a real OS, but that would be easy to change by just giving the Zone a smaller height. Alternatively, you could allow the Scene to deal with the input events itself and drag it from there, but I liked the simplicity of this method so kept it in this demo. Remember, there's rarely one way to skin a cat.

I hope you've had fun learning about what Scenes can do, how they work and how to visually manipulate them. As mentioned before, all of the code can be found in the repo and will work with Phaser 3.7.0 and above.

Basic8 is a 'fantasy computer', much like Pico8, with some neat built-in tools. Sprite editor, map editor, music, etc. Worth having a look if you enjoy coding in a virtual environment.

Choo Choo is a stunning example of what can be crammed into just 140 characters and this is a fascinating blog post about how he managed it.

I recently bought this awesome little Windows app called Groupy. It allows you to

join windows together into a tabbed interface, just like a browser, but it works with any app. Absolutely love it!

Phaser Releases

Phaser 3.6.0 released April 19th 2018.
Phaser CE 2.10.3 released March 21st 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.