Web Version Unsubscribe

Welcome to Issue 124 of Phaser World

Greetings everyone! It's been another awesome week of updates and releases. There's a huge Dev Log this issue talking all about the new renderer updates in Phaser 3. Plus some superb games to play too. I literally lost several hours to the insane Road of Fury while preparing this issue. I really ought to give 'Staff Picks' for the tutorials, and not just the games, because if I did then the absolutely incredible Modular Game Worlds tutorial featured this issue would win one! It's a complete masterclass in how a great tutorial should be written and illustrated, so please give it a read :)

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.

HTML5 Cross Platform Game Development Using Phaser 3
Learn to build HTML5 games using Phaser 3 and other free software
Emanuele Feronato

Emanuele Feronato is easily the most prolific author of Phaser tutorials on the planet. He has published over 250 of them in the past few years and started writing Phaser 3 tutorials before most others. As the framework has evolved, his tutorials have kept pace. So it's great to see that he's been busy writing a book and it's now available.

It's 155 pages long, with 28 source code examples, taking you through the process of creating a full cross platform game. What's more, buying a copy actively contributes towards funding my work on Phaser too!

The Latest Games

Game of the Week

Road of Fury - Desert Strike

Take the wheel and unleash hell in this Mad Max inspired shooter. Screens full of missiles, rockets, drones, bikers and mayhem is a sight to behold.

Staff Pick

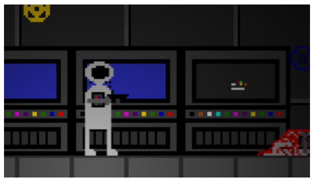
Super Starkiller 3000

Welcome to the Glorious Fleet. You are currently entering enemy space. We recommend you engage combat systems immediately.

Staff Pick

Balloons Path Swipe

Draw lines through the colored balloons to pop them and see if you can complete the levels in the number of moves allowed.



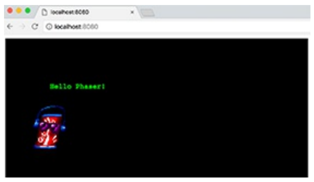
Disturbance in Sector C

Your ship has been taken over by a virus. It is killing everyone on board. You must survive.

Tap Tap Golf

Tap Tap Golf is an intuitive, challenging, Arcade Physics Golf game.

What's New?

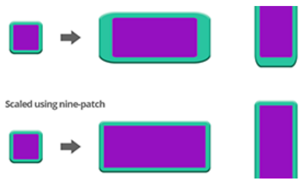


Modern Phaser 3 Web Dev Tutorial

A comprehensive guide on setting up a Modern Web Development environment for Phaser 3.

Modular Game Worlds in Phaser 3

An extremely comprehensive, well written and illustrated tutorial on tilemaps in Phaser 3, written by the developer who built the tilemap API.



Koreez Games Nine Patch Plugin

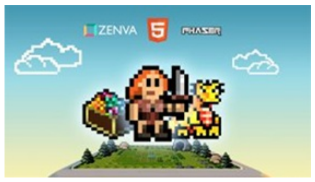
This Nine Patch plugin easily adds 9-slice scaling support to Phaser 3.

Free Phaser 2 Course

Free Udemy Course: Making Games With Phaser 2. Learn the fundamentals of JavaScript and Phaser!

Brand new Phaser 3 Book

The first book on the creation of HTML5 cross platform games using Phaser 3 and other free software.



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.

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

KC Chan
Melissa Sorrells
Ricardo Trindade

Richard Lovejoy

also, thank you **EDease** for the PayPal donation.

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 #124

Welcome to Dev Log 124. This covers all of last weeks Phaser development. Let's dive right in.

Phaser 3 uses different internal batches for WebGL rendering based on the type of Game Object you're using. The main one of those is the Texture Tint Pipeline (TTP) which, as the name implies, is responsible for all Game Objects that use textures: Sprites, TileSprites, Text objects and so on. It was this pipeline that I overhauled for the 3.11 release, which you can read about in Dev Log X.

The Flat Tint Pipeline (FTP) is another pipeline, responsible primarily for the Graphics game object. The name comes from its focus on drawing flat tinted objects, i.e. single color objects with no texture. The pipeline is full of methods such as `batchFilledRectangle` and `batchStrokePath`. It's also used by the Camera system. If a Camera has a background color set, or has a Flash or Fade effect running on it, then it uses the Flat Tint Pipeline to draw those rectangles to the viewport.

There are two other pipelines. The Forward Diffuse Light Pipeline (FDLP) is used whenever a Game Object has light effects enabled on it and renders using a normal map bound with the objects texture. The final pipeline is the Bitmap Mask Pipeline (BMP). This pipeline is used specifically for dealing with Bitmap Masks.

When the Game Objects are being iterated by the renderer if it finds one with a mask set it will call the `beginMask` method, which in turn sets everything to be rendered to the mask framebuffer.

There are other steps these two pipelines take that I won't cover in this Dev Log. Suffice to say that invoking *any* pipeline causes the WebGL renderer to do two important things. First, if a pipeline changes, then the old pipeline is flushed out. Then, once flushed, the new pipeline activates its shaders and gets ready to use them. Both of these actions are expensive in terms of adding to the total number of GPU operations being performed and the number of draw calls.

The term 'flush' means to dump all of the data that has been batched to the GPU and cause a draw call. If you've got say 100 Sprites all using the same texture, and then the display list has a Graphics object on it, it will 'flush' those 100 Sprites out. This means all of their data that has built-up in the batch is sent to the GPU and a draw call is requested.

Imagine a typical RPG game scene with a group of monsters. Each monster is a textured animated Sprite with a health bar above its head and a name below. Your first thought would rightly be "Let's use the Graphics Game Object to draw the health bars!" - after all, it's full of methods like `fillRect` so seems perfect for the task. You happily proceed to drop in Graphics based health bars but as the monster count increases, the frame rate takes a dive and the amount of GPU calls increases exponentially.

The reason is likely obvious to some of you already. As the renderer works its way through the display list it encounters a monster. First, it finds the Sprite, so it uses the Texture Tint Pipeline. Then it finds the health bar, so it has to swap to the Flat Tint Pipeline, causing the Texture Tint Pipeline to flush and the Flat Tint Pipeline shaders to bind. The moment a pipeline flushes it resets its batch back to zero. All the benefits of building up the data in the batch are gone. Then it finds the monsters name as a Bitmap Text object, so it swaps back to the Texture Tint Pipeline, flushing the Flat Tint one and rebinding its shader. This ping-pong process happens for every single monster in the Scene.

With very little effort on your part, all the benefits of batching are destroyed and WebGL will start struggling (sooner on lower-powered devices than others.) The thing is, you've done nothing wrong. There was no reason you should suspect that using Graphics objects would cause this, yet cause it, it does.

Or at least, it used to.

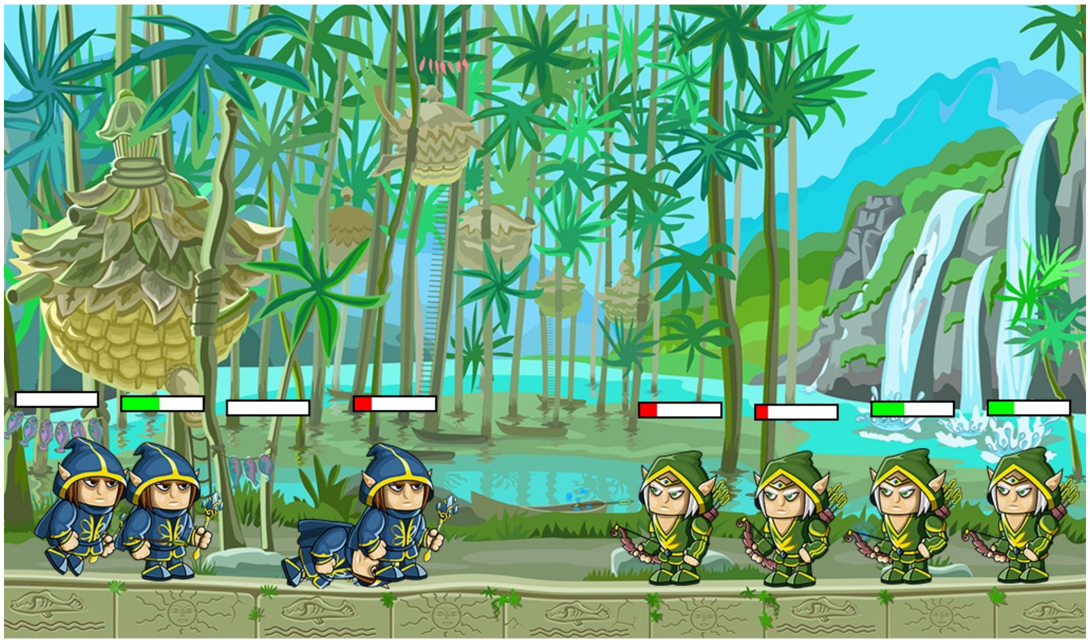
After I refactored the Texture Tint Pipeline in 3.11 I knew that I would work on the

Flat Tint Pipeline next. It exhibited lots of the same problems as the TTP did; namely hundreds of lines of duplicate and sometimes inconsistent code. Originally, I thought I was going to do the same thing as I did with the TTP, which was to just tidy-up the internals, consolidate the most common functions and take advantage of the work I'd done on the Transform Matrix class in 3.11. I started out by doing exactly this. The pipeline became smaller and more unified, relying on a core set of internal methods.

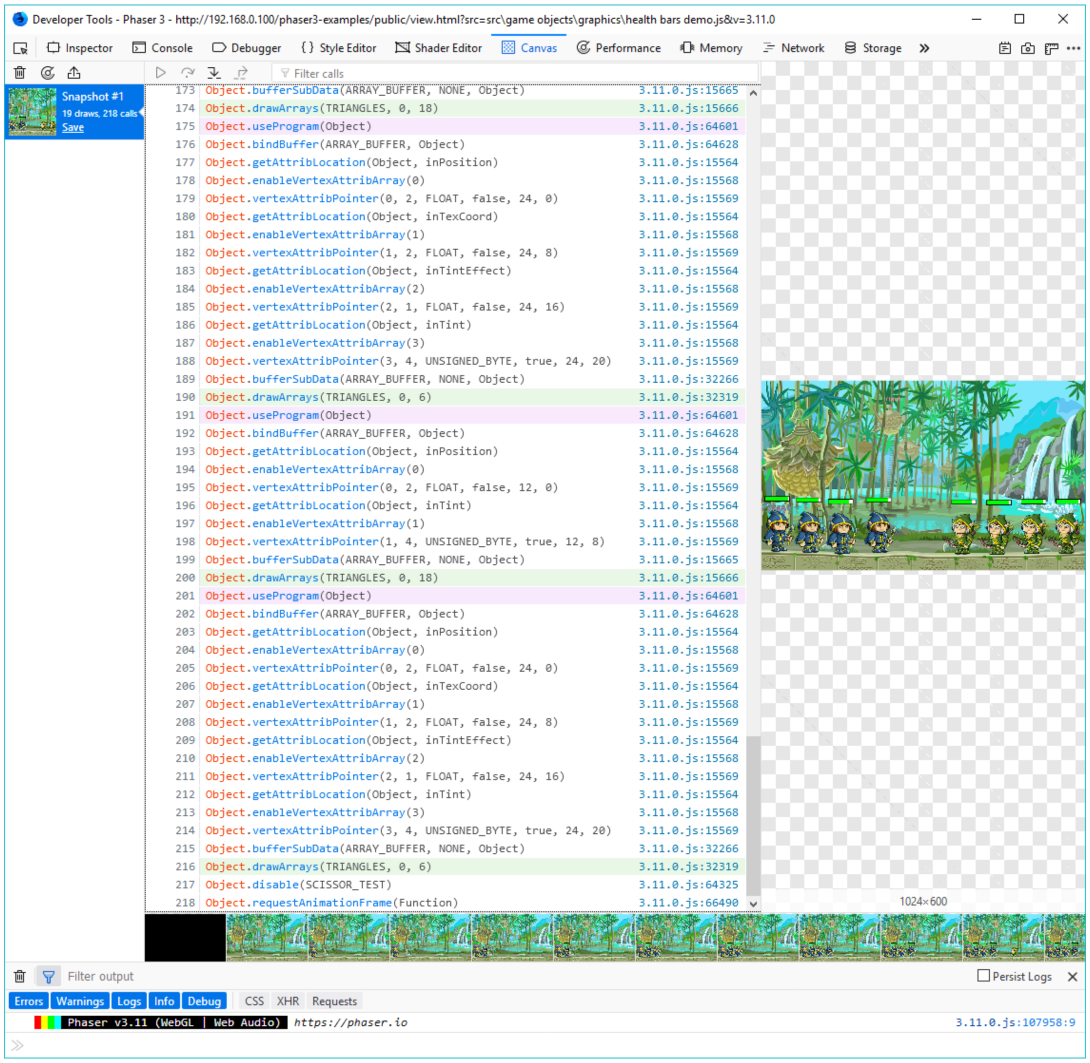
Then I took a look at the shader it was using. It was virtually identical to the shader the Texture Tint Pipeline used. Naturally, I started to think: What if the same shader could be used for them both? If that was possible then it would avoid the whole "pipeline swapping, batch flushing, shader binding" issues that were happening. A few hours of coding later and I had my answer: It worked and it worked *great*!

Buoyed by this progress I carried on consolidating the methods in the Flat Tint Pipeline. It soon became obvious that there were only really a handful of common methods needed, and one of those was identical to a method already in the Texture Tint Pipeline. The more I tidied things up, the more I realized we didn't need the Flat Tint Pipeline at all. So I merged the few remaining methods into the Texture Tint Pipeline, made one small tweak to the shader and the work was complete.

The net result was well worth the few days development it took. I removed the Flat Tint Pipeline entirely. This is over 1000 lines of code gone and a saving of 42 KB. Obviously, some of that code moved to the Texture Tint Pipeline, but only 300 lines of it, yet all of the previous functionality remains intact. File savings aside, the most important benefit is that mixing textured Game Objects and Graphics objects on the display list no longer causes any problems. They're all adding data to the same batch, they're all using the same shader, so it means the number of WebGL operations and draw calls significantly dropped. Let's have a look at a demo to see the impact of this change.



If you watch the above demo for a while you'll see the two bands of elves try to decimate each other. It's simple stuff, only 8 animated sprites, a backdrop, and a health bar each. Let's profile this running under the current version of Phaser (3.11):

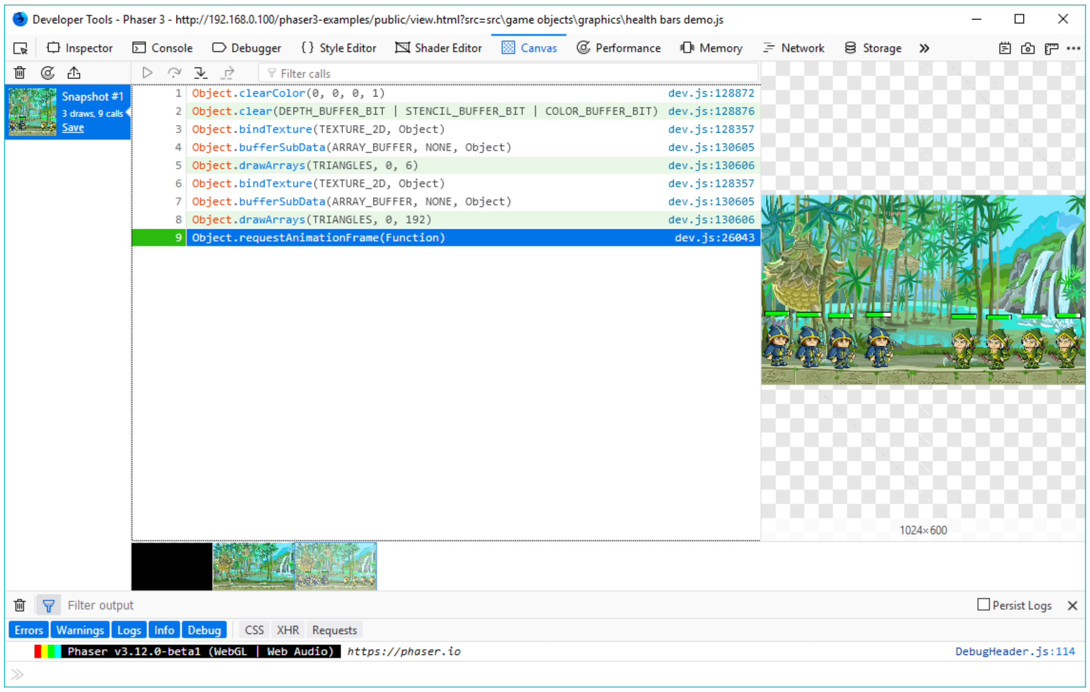


```
173 Object.bufferSubData(ARRAY_BUFFER, NONE, Object)          3.11.0.js:15665
174 Object.drawArrays(TRIANGLES, 0, 18)                       3.11.0.js:15666
175 Object.useProgram(Object)                                 3.11.0.js:64601
176 Object.bindBuffer(ARRAY_BUFFER, Object)                   3.11.0.js:64628
177 Object.getAttribLocation(Object, inPosition)              3.11.0.js:15564
178 Object.enableVertexAttribArray(0)                         3.11.0.js:15568
179 Object.vertexAttribPointer(0, 2, FLOAT, false, 24, 0)     3.11.0.js:15569
180 Object.getAttribLocation(Object, inTexCoord)              3.11.0.js:15564
181 Object.enableVertexAttribArray(1)                         3.11.0.js:15568
182 Object.vertexAttribPointer(1, 2, FLOAT, false, 24, 8)     3.11.0.js:15569
183 Object.getAttribLocation(Object, inTintEffect)            3.11.0.js:15564
184 Object.enableVertexAttribArray(2)                         3.11.0.js:15568
185 Object.vertexAttribPointer(2, 1, FLOAT, false, 24, 16)    3.11.0.js:15569
186 Object.getAttribLocation(Object, inTint)                  3.11.0.js:15564
187 Object.enableVertexAttribArray(3)                         3.11.0.js:15568
188 Object.vertexAttribPointer(3, 4, UNSIGNED_BYTE, true, 24, 20) 3.11.0.js:15569
189 Object.bufferSubData(ARRAY_BUFFER, NONE, Object)          3.11.0.js:32266
190 Object.drawArrays(TRIANGLES, 0, 6)                        3.11.0.js:32319
191 Object.useProgram(Object)                                 3.11.0.js:64601
192 Object.bindBuffer(ARRAY_BUFFER, Object)                   3.11.0.js:64628
193 Object.getAttribLocation(Object, inPosition)              3.11.0.js:15564
194 Object.enableVertexAttribArray(0)                         3.11.0.js:15568
195 Object.vertexAttribPointer(0, 2, FLOAT, false, 12, 0)     3.11.0.js:15569
196 Object.getAttribLocation(Object, inTint)                  3.11.0.js:15564
197 Object.enableVertexAttribArray(1)                         3.11.0.js:15568
198 Object.vertexAttribPointer(1, 4, UNSIGNED_BYTE, true, 12, 8) 3.11.0.js:15569
199 Object.bufferSubData(ARRAY_BUFFER, NONE, Object)          3.11.0.js:15665
200 Object.drawArrays(TRIANGLES, 0, 18)                       3.11.0.js:15666
201 Object.useProgram(Object)                                 3.11.0.js:64601
202 Object.bindBuffer(ARRAY_BUFFER, Object)                   3.11.0.js:64628
203 Object.getAttribLocation(Object, inPosition)              3.11.0.js:15564
204 Object.enableVertexAttribArray(0)                         3.11.0.js:15568
205 Object.vertexAttribPointer(0, 2, FLOAT, false, 24, 0)     3.11.0.js:15569
206 Object.getAttribLocation(Object, inTexCoord)              3.11.0.js:15564
207 Object.enableVertexAttribArray(1)                         3.11.0.js:15568
208 Object.vertexAttribPointer(1, 2, FLOAT, false, 24, 8)     3.11.0.js:15569
209 Object.getAttribLocation(Object, inTintEffect)            3.11.0.js:15564
210 Object.enableVertexAttribArray(2)                         3.11.0.js:15568
211 Object.vertexAttribPointer(2, 1, FLOAT, false, 24, 16)    3.11.0.js:15569
212 Object.getAttribLocation(Object, inTint)                  3.11.0.js:15564
213 Object.enableVertexAttribArray(3)                         3.11.0.js:15568
214 Object.vertexAttribPointer(3, 4, UNSIGNED_BYTE, true, 24, 20) 3.11.0.js:15569
215 Object.bufferSubData(ARRAY_BUFFER, NONE, Object)          3.11.0.js:32266
216 Object.drawArrays(TRIANGLES, 0, 6)                        3.11.0.js:32319
217 Object.disable(SCISSOR_TEST)                              3.11.0.js:64325
218 Object.requestAnimationFrame(Function)                    3.11.0.js:66490
```

```
Phaser v3.11 (WebGL | Web Audio) https://phaser.io            3.11.0.js:107958:9
```

It's a little tricky to make out if you're looking at the image embeddd in the newsletter - but the important part is that this simple scene is using no less than 19 draw calls and a staggering 218 WebGL operations.

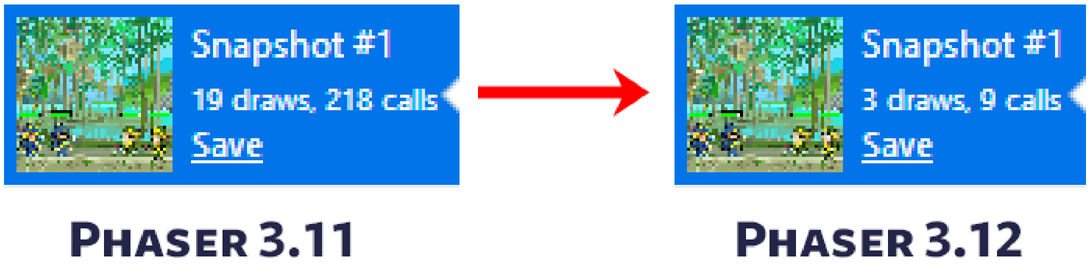
After all the work I did on the Flat Tint Pipeline, let's reprofile the exact same scene under Phaser 3.12 Beta 1:



```
Object.clearColor(0, 0, 0, 1)                                            dev.js:128872
Object.clear(DEPTH_BUFFER_BIT | STENCIL_BUFFER_BIT | COLOR_BUFFER_BIT)   dev.js:128876
Object.bindTexture(TEXTURE_2D, Object)                                   dev.js:128357
Object.bufferSubData(ARRAY_BUFFER, NONE, Object)                         dev.js:130605
Object.drawArrays(TRIANGLES, 0, 6)                                       dev.js:130606
Object.bindTexture(TEXTURE_2D, Object)                                   dev.js:128357
Object.bufferSubData(ARRAY_BUFFER, NONE, Object)                         dev.js:130605
Object.drawArrays(TRIANGLES, 0, 192)                                     dev.js:130606
Object.requestAnimationFrame(Function)                                    dev.js:26043
```

```
Phaser v3.12.0-beta1 (WebGL | Web Audio) https://phaser.io                DebugHeader.js:114
```

Again, it might be hard to tell from the size of the screen shot, but you should immediately see there are far less draws going on (the thumbnails along the bottom) and the list of gl ops is massively smaller:



218 calls down to just 9. 19 draws down to just 3. That's quite some improvement! You can profile the demo yourself, just use Firefox and it's Canvas Profiling tool and swap between testing the 3.11 build against the dev build. Try it on your own games too. It'd be great to see what impact it has there.

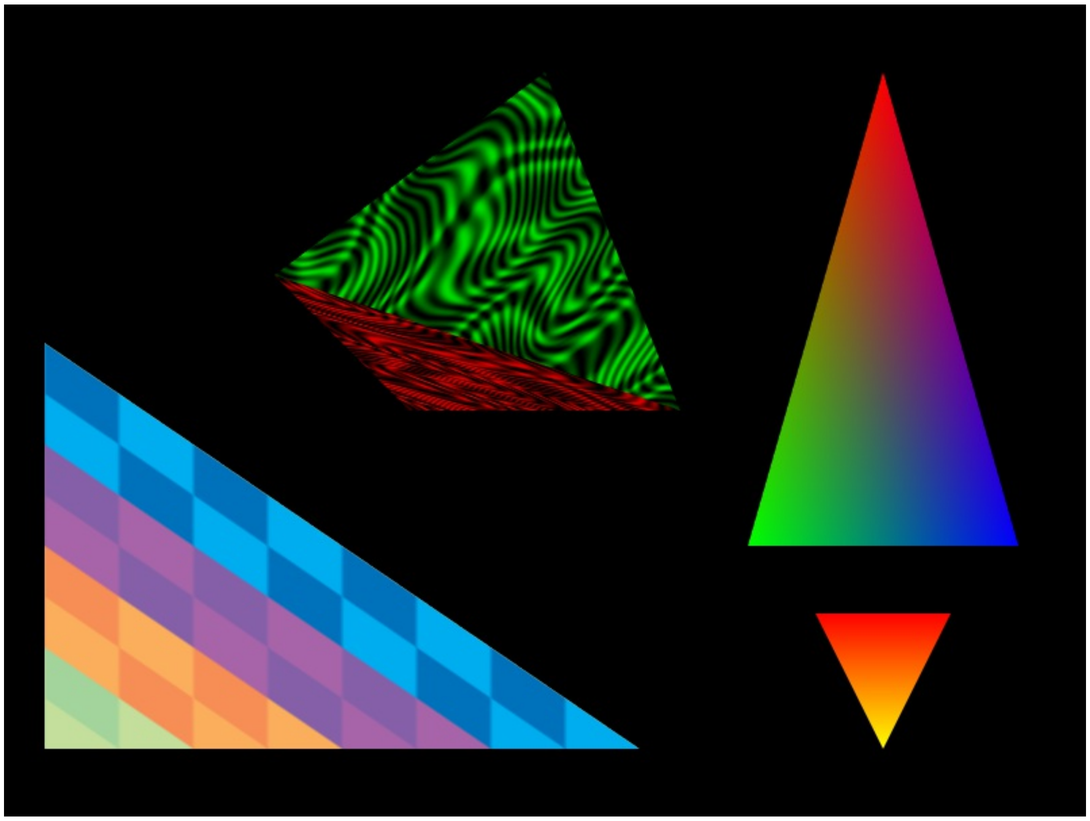
Textured Graphics

Nothing in life is truly 'free' though - so what the cost of making this change to the pipelines? The main one is that the Texture Tint Pipeline had two extra attributes and a uniform bound to its shader that the Flat Tint one didn't have. So, for example, when drawing a filled rectangle it's now sending a little more data to the GPU than before. The benefits of not flushing the batch or swapping shaders far outweigh this data cost. When I realised this it did make me think "What if we used that texture though?", and the `Graphics.setTexture` method was born.

You can now set a fill texture that will be used when filling shapes in the Graphics object. For example, look at the following code:

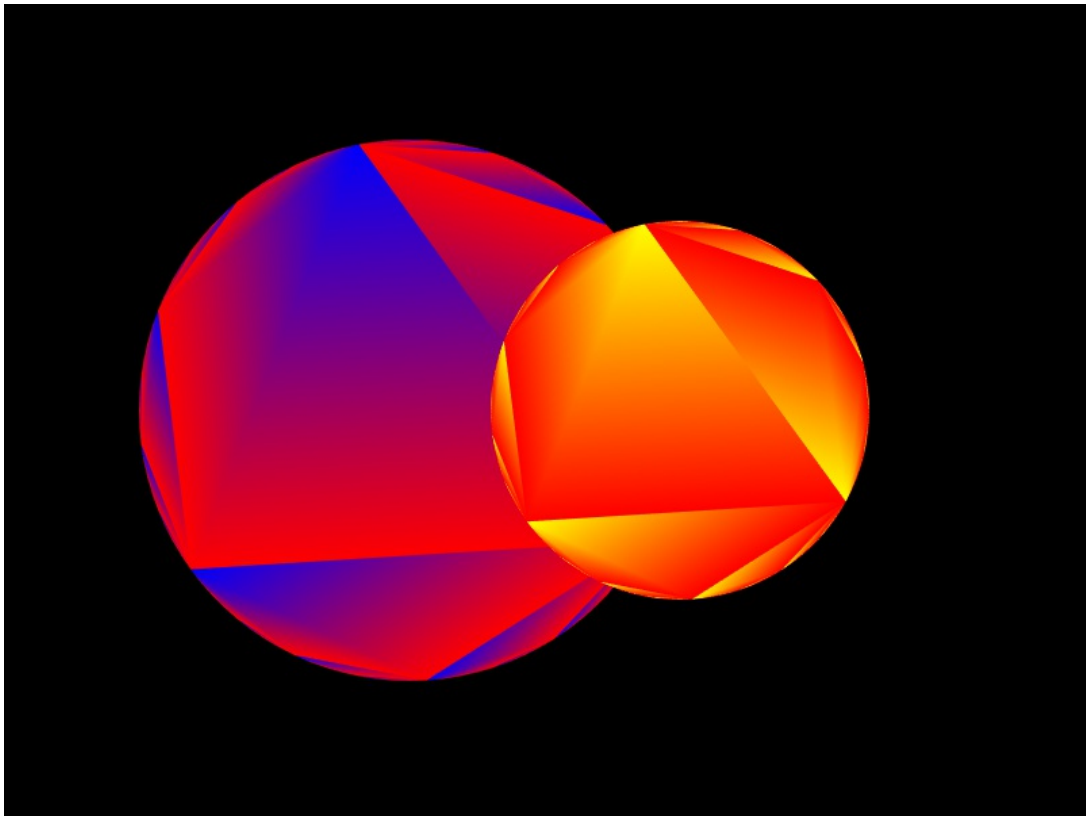
```
function preload ()
{
    this.load.image('metal', 'assets/textures/alien-metal.jpg');
    this.load.image('test', 'assets/sprites/128x128.png');
}

function create ()
{
    var graphics = this.add.graphics();

    graphics.setTexture('metal', null, 0);

    graphics.fillStyle(0x00ff00);
    graphics.fillTriangle(200, 200, 400, 50, 500, 300);

    graphics.fillStyle(0xff0000);
    graphics.fillTriangle(200, 200, 500, 300, 300, 300);

    graphics.setTexture('test', null, 2);

    graphics.fillStyle(0xff0000);
    graphics.fillTriangle(30, 550, 30, 250, 470, 550);

    graphics.setTexture();

    graphics.fillGradientStyle(0xff0000, 0x00ff00, 0x0000ff, 0xffff00);
    graphics.fillTriangle(650, 50, 550, 400, 750, 400);

    graphics.fillGradientStyle(0xff0000, 0xff0000, 0xffff00, 0xffff00);
    graphics.fillTriangle(600, 450, 700, 450, 650, 550);
}
```

And this is what it looks like when rendered:



Pretty neat, huh? :) You have control over how the texture and the fill style of the Graphics object are used. The default is to mix the fill color with the texture, as seen in the swirly looking triangles in the picture above. You can also choose to have the color solidly fill in all non-transparent areas of the texture, or have just the texture applied and no color at all (the triangle in the bottom left). As well as using textures you can now also set a gradient fill. It works in the same way as setting a per-vertex tint on a Game Object, with the exact same method arguments. You can see it in the two triangles on the right of the image.

There are limitations to how the texturing and gradients work. Firstly, this a WebGL only feature. Secondly, the texture is *not* a 'fill pattern'. There is no control over how the texture repeats and it fills the shapes in per triangle or quad. The downside of both the texture and gradient fills becomes apparent if you've got a complex path, or try using it on an arc. WebGL works using triangles, so shapes like circles are actually triangulated during runtime and turned into a series of triangles. This is handled internally using the library Earcut. Each one of these triangles is then textured, or filled, on its own. They're not textured as a collective group, which means you'll get some weird looking results like this:



^ probably not what you were expecting, right? I'm happy with this as a limitation for now. Textures and gradients work really well for triangles and rectangles and other paths depending on the result you want. It was a relatively free feature to add as it is part of the shader code anyway, so it made sense to expose it. I'd rather you had the choice to use it or not, rather than lock it down. After all, I'm sure you'll come up with some creative ways to use it and it's entirely optional: You can happily just carry on using normal solid color fills like before.

Shape Game Objects

I've lost count of the number of times people have asked me, or posted to the forum, about how it doesn't work when they add a geometry object such as a Rectangle to the display list. I've always been a bit bemused by this. The Geometry objects clearly live in their own name space and have none of the properties any of the other Game Objects do, so there's no way they would work on the display list.

Yet, if you think about it, it makes sense to be able to do this. I had considered adding a `Shape` Game Object for a while. A combination of a Geometry object with all the aspects a normal Game Object needs in order to live on the display list. I had ruled this out because of the way the Flat Tint Pipeline previously worked. As you've read above though, this pipeline is now fully integrated into the main one, meaning I can proceed and create a Shape Game Object.

I've not yet decided if this will be in 3.12 or the 3.13 release, but I have sketched out the details of it, and essentially it'll be any of the Geometry objects already supported by Phaser combined with a transform, so you can blast it around the display list like you would any other object. It's a nice side-effect of all the work I've been doing in the renderer this past month that I can now offer features like this and know they won't adversely impact the rendering.

Facebook Instant Games Plugin

If you've been keeping an eye on the repo then you'll notice I have started pushing up all of the Facebook Instant Games plugin code too. At the moment it's hanging directly off the main Game object, exposed in every Scene, but that is purely during development. When I'm finished it'll move to its own global plugin and there will be a new 'Phaser + Instant Games' build available and published, which will be updated with every release of Phaser.

I'm very pleased with development of the plugin so far. The Instant Games API is relatively easy to use as is, but there are definitely areas where it could be made more Phaser friendly, which is exactly what I've been doing.

For example, you can call the method **loadPlayerPhoto**, provide it with a key, and it'll automatically go and grab the player photo for you and store it within the Phaser Texture Manager under the given key, so you can treat it just like any other asset in your game.

I've also built the Data Manager plugin in, so you can create data values directly and modify them, and they'll be instantly synced with your Facebook app. You just have to use the values like normal and the plugin will take care of ensuring they're saved via the API. The same goes for game stats and session data too.

I also added in really easy-to-use commands like **openShare**. This will take the key of an image in the Texture Manager along with some text, and will automatically open up a Facebook sharing window, pre-populated with your text and image.

There's lots more coming as well, including really easy ways to handle in-game purchases, video ads and the display of leaderboards. Facebook have been excellent to work with on this project and the finished plugin will be available in the 3.12 release. Of course, if you don't want to use it, it won't add anything to your build size. It's entirely optional. But if you've been thinking about trying out making a Facebook Instant Game, 3.12 will make it easier than any other framework out there.

GameHut present this Sega Genesis Megademo style video showcasing lots of groundbreaking fx used in their 16-bit games.

The Modern Web Podcast is a really neat podcast to listen to. In this slightly older episode, they talk to the V8 team.

Beyond Skyrim is Skyrim's most ambitious mod ever.

Phaser Releases

Phaser 3.11.0 released July 13th 2018.
Phaser CE 2.11.0 released June 26th 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.