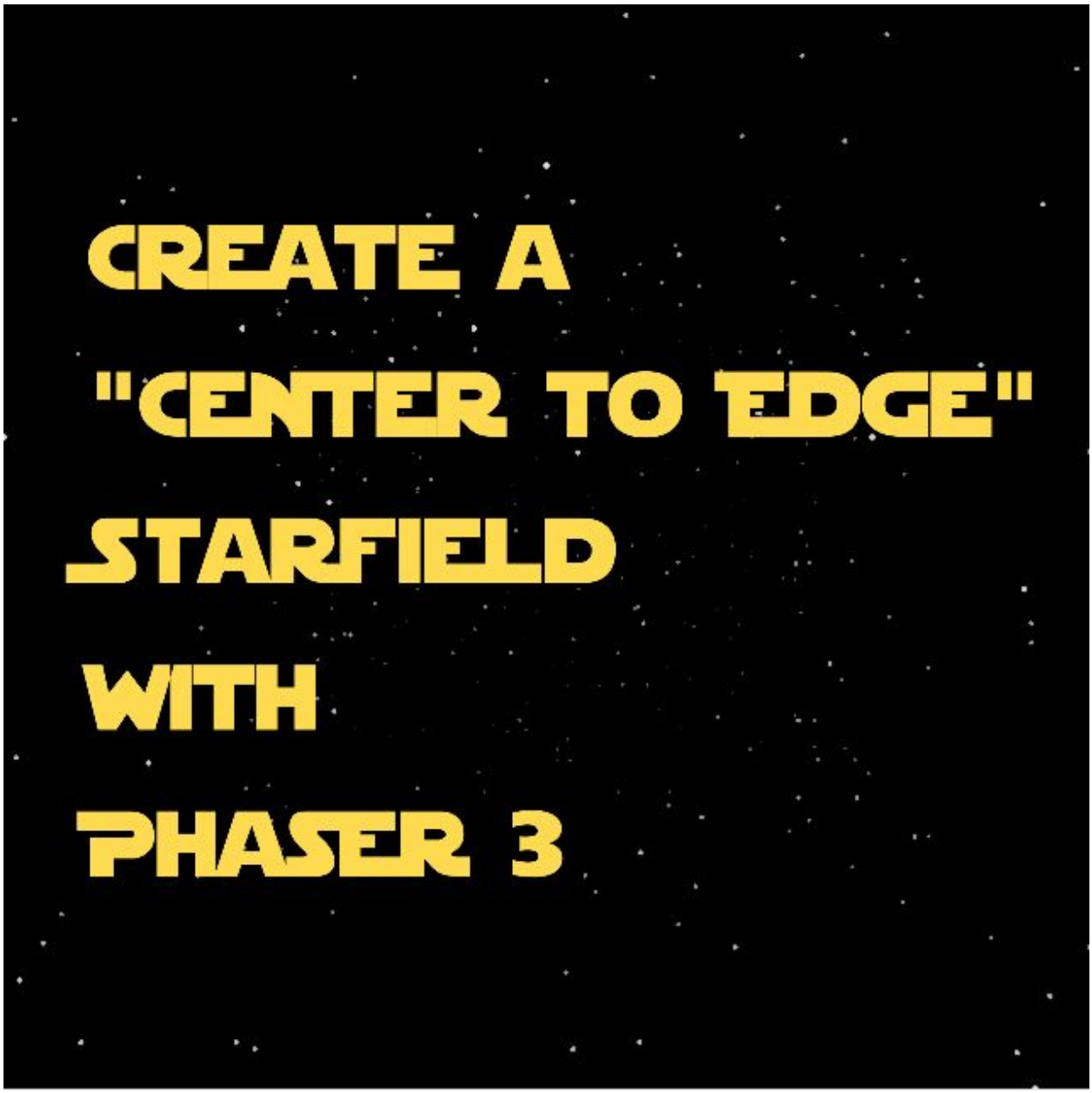
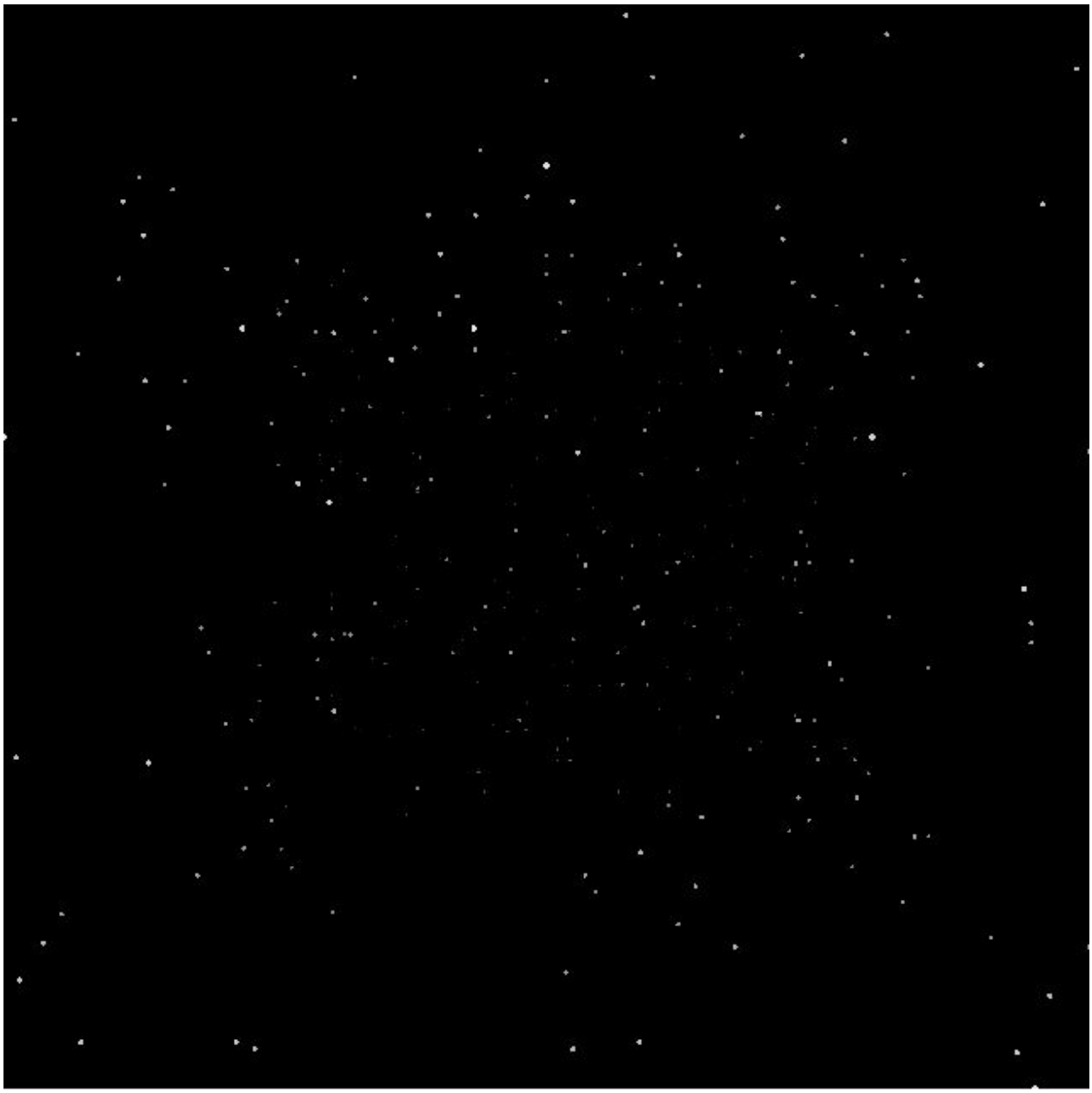


Create a “Center to Edge” Starfield with Phaser 3
Version 1.0
Written by Jared York

In this tutorial, we will be creating what I call, a “center to edge” starfield. The final result will look like the following:



Tutorial Requirements

- Basic to intermediate knowledge of JavaScript
- Web server
- Code editor (not necessarily required, but highly recommended)

Ensure a Web Server is Set Up

Even though Phaser games are ran in the browser, unfortunately you can't just run local HTML files directly from your file system. When requesting files over http, the server security only allows you to access files you're allowed to. When loading a file from the local file system (the file:// protocol), your browser highly restricts it for obvious security reasons. It's not good to allow code on a website to read anything in your raw file system. Because of this, we will need to host our game on a local web server.
We recommend taking a look at Phaser's official guide, "Getting Started with Phaser 3", to learn which web server is compatible with your system and there are links to each one. The guide also provides some detailed summaries on each of the various web servers mentioned.

Create the Folders and Files Needed

First, find the directory where your web server serves files from (WAMP Server, for example, hosts files from the www directory within it's installation folder at C:/wamp64.) Once you have found the location, create a new folder inside it and call it anything you like.
Next, enter the folder and create a new file named, "index.html". Our index file is where we will declare the location of our game scripts and Phaser script.
We will also need to create a folder inside our game folder for our JavaScript. This folder can be named anything you like, but we will be using it for storing our JavaScript. Once we have our folder for JavaScript, create two new files inside this folder named: game.js, and SceneMain.js. The file structure of our project should look like so:

```
(game folder)/
|_index.html
|_js/
  |_ game.js
  |_ SceneMain.js
```

The last step we have to do is grabbing the latest Phaser script from GitHub. For our purposes, feel free to click either, "phaser.js" or "phaser.min.js". I personally chose "phaser.js". Click the link for either script, and you should see a new page that displays a button named, "View raw", near the center of the page, about halfway down. Next, click the "View raw" link, then right click anywhere on the new page of code that appears. Once the dropdown appears, click "Save Page As" or something similar. A save dialog should then appear where you can save the Phaser script in the JavaScript directory we created.
Let's start by adding to our index.html file. Add the following code to create the HTML document and link our scripts:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <meta lang="en-us">
        <title>TutorialStarfield</title>
    </head>

    <body>
        <script src="js/phaser.js"></script>
        <script src="js/SceneMain.js"></script>
        <script src="js/game.js"></script>
    </body>
</html>
```

We are now finished with index.html. Moving on to game.js, add the following the file:

```
var config = {
    type: Phaser.WEBGL,
    width: 640,
    height: 640,
    backgroundColor: "black",
    physics: {
        default: "arcade",
        arcade: {
            Gravity: { x: 0, y: 0 }
        }
    },
    scene: [
        SceneMain
    ],
    pixelArt: true,
    roundPixels: true
};
var game = new Phaser.Game(config);
```

The above code defines the configuration properties needed for creating the Phaser game. Feel free to play around with these.
Jumping to our SceneMain.js file, we can create the class for our scene, SceneMain. It's good to point out that classes in JavaScript aren't true classes. Classes are objects, but with some syntactic sugar sprinkled on top to make organizing object-oriented code easier. We will want our class to extend Phaser.Scene since we are going to be building on-top of the default Phaser.Scene object. Add the following to SceneMain.js:

```
var config = {
    type: Phaser.WEBGL,
    width: 640,
    height: 640,
    backgroundColor: "black",
    physics: {
        default: "arcade",
        arcade: {
            Gravity: { x: 0, y: 0 }
        }
    },
    scene: [
        SceneMain
    ],
    pixelArt: true,
    roundPixels: true
};
var game = new Phaser.Game(config);
```

The create function will be triggered as soon as the scene is started. We will want to add some properties, as well as generate the points for our stars there. In the create function, add the following properties:

```
var config = {
    type: Phaser.WEBGL,
    width: 640,
    height: 640,
    backgroundColor: "black",
    physics: {
        default: "arcade",
        arcade: {
            Gravity: { x: 0, y: 0 }
        }
    },
    scene: [
        SceneMain
    ],
    pixelArt: true,
    roundPixels: true
};
var game = new Phaser.Game(config);
```

Our star field will work by having an array of points, and a group for the graphics objects (the circles drawn). Each point will have three properties: x, y, and z. Positions x and y will determine where on the screen a star graphics object will be created, and z is used calculating how close a star is to the “observer”. The property maxDepth will be used for determining the maximum distance a star can be from the “observer”. Next, we will need to create a for loop which will create the point objects and add them to the points array:

```
for (var i = 0; i < 512; i++) {
    this.points.push({
        x: Phaser.Math.Between(-25, 25),
        y: Phaser.Math.Between(-25, 25),
        z: Phaser.Math.Between(1, this.maxDepth)
    });
}
```

The last thing we need to do to make our star field work, is add some code to the update function of SceneMain. We will be clearing the stars each update, iterate through our points to calculate the new star positions, as well as create the star graphics objects at those positions. We will first start by adding some code to clear the stars group and also create a for loop to iterate through our points:

```
this.stars.clear(true, true);

for (var i = 0; i < this.points.length; i++) {
    var point = this.points[i];

}
```

In addition to calculating the new position of each point, we will also be resetting the position once it gets close enough to the “observer”. This will save us from having to destroy and create new points. Add the following code inside the for loop to subtract from the z position of each point (which results in the star moving closer to the “observer” as z nears zero):

```
point.z -= 0.2;
```

After that, we will also need to add the reset logic I mentioned above:

```
if (point.z <= 0) {
    point.x = Phaser.Math.Between(-25, 25);
    point.y = Phaser.Math.Between(-25, 25);
    point.z = this.maxDepth;
}
```

Now that we've added the reset logic, we can calculate the new position of the current point. Add the following:

```
var px = point.x * (128 / point.z) + (this.game.config.width * 0.5);
var py = point.y * (128 / point.z) + (this.game.config.height * 0.5);
```

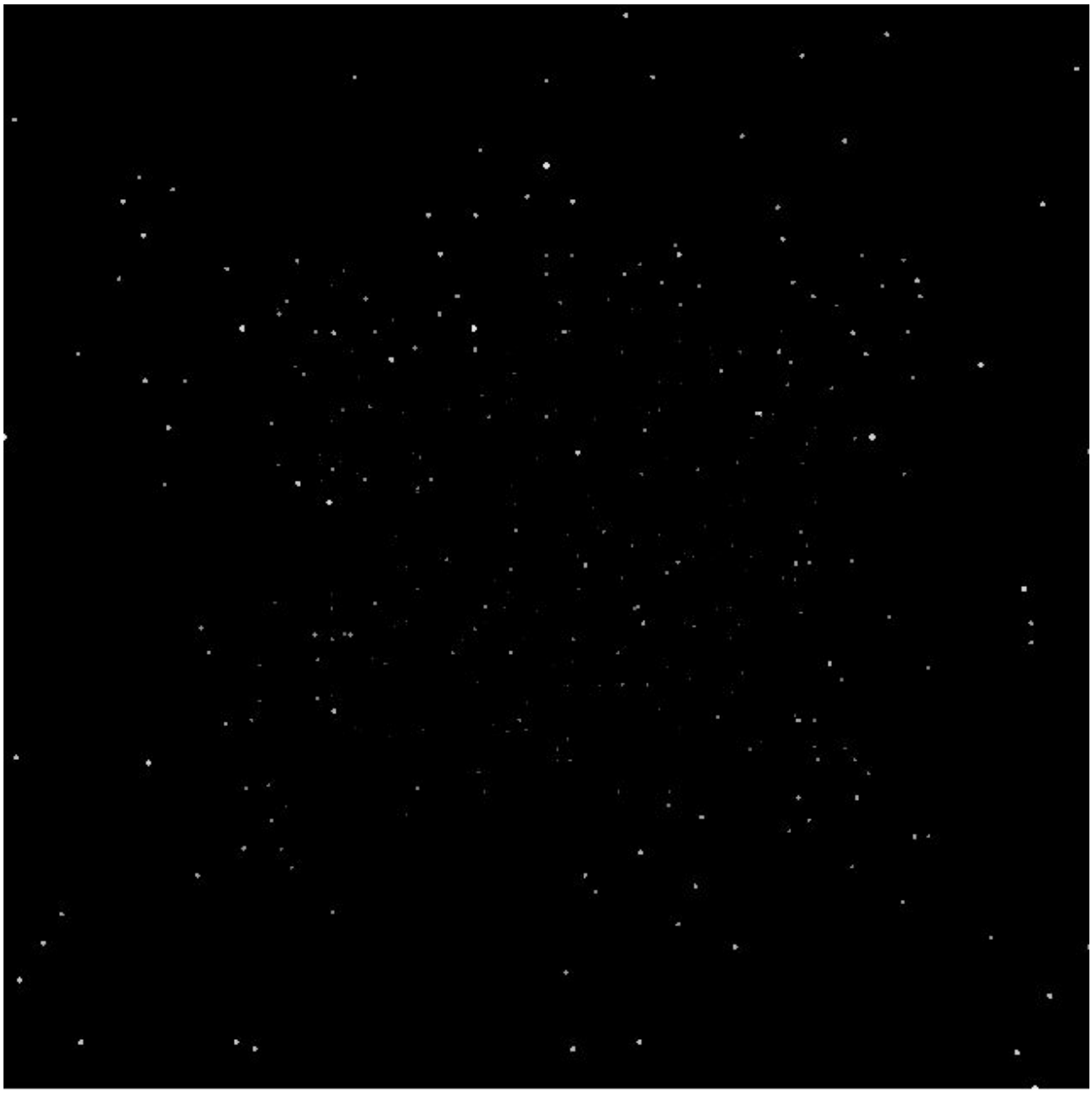
The last thing we have to do is create the physical circle that will be drawn to represent a star. We will be creating an instance of Phaser's circle object. This instance will act pretty much as a template to provide to an instance of Phaser's graphics object to draw. After the previous code, add the following to create a circle at the position of the point:

```
var circle = new Phaser.Geom.Circle(
    px,
    py,
    (1 - point.z / 32) * 2
);
```

The cool thing about the last parameter we added when creating our circle, is that it will grow in size as it nears the "observer". Now, we can create the graphics object, and add it to the stars group. Add the following:

```
var graphics = this.add.graphics({ fillStyle: { color: 0xffffff } });
graphics.setAlpha((1 - point.z / 32));
graphics.fillCircleShape(circle);
this.stars.add(graphics);
```

By setting the alpha, we can make stars in the distance much more faint, and have them become brighter as they near the "observer".
If we navigate to the project in our browser, we should see finished result:



Got it? Fantastic!

The fun part now is to tweak the various numbers. Some interesting results can be achieved! I look forward to seeing what you can create with this! If you have any questions, suggestions, or feedback, feel free to tweet at me, my handle is @jaredyork. You can also reach me via my email at jared.york@yorkcs.com. I'm more than happy to help anyone out. Hopefully this will be useful for some of you. 🙂

You can find the full source code for this tutorial on GitHub.

If you’re interested in hearing about more of my tutorials and courses, be sure to fill out the form.