Pharser 3 – Getting Started

1st Edition

Compiled by: geekman-web

Index

About

Purpose

This guide was created with the purpose to be used as a reference/quick start guide for developers who are already familiar with the web basics, but haven't used the Phaser Library before.

Document Structure

This document follows the following structure:

1. Title Page
2. Index
3. About
4. Tutorials

Document Format

The title page include 'title' 32px, 'edition' 20px, 'compiled by' 14px, 'contributors' 14px if any under 14px.The title headings are “Heading 1”, bold, 24px. Subheadings are 'Default', 14px underline.

Contributors

You can freely contribute to this guide. If your changes end up in the next edition, you will be listed as a contributor. Any contributions should follow the specified format to be included. The structure should also be kept.

Licence

This document can be updated to improve the guide freely. The only pages that should not be modified should be the front page and the about page. If you wish to modify these pages, or include your changes to the official document in the next release, a request should be made to https://github.com/geekman-web.

Sources

At the end of each tutorial, the source link is included for that tutorial.

Tutorial 1: The Basics

Phaser basic structure

```
var config = {
    type: Phaser.AUTO,
    width: 800,
    height: 600,
    scene: {
        preload: preload,
        create: create,
        update: update
    }
};

var game = new Phaser.Game(config);

function preload ()
{
}

function create ()
{
}

function update ()
{
}
```

The `type` property can be either `Phaser.CANVAS`, `Phaser.WEBGL`, or `Phaser.AUTO`
The `width` and `height` properties set the size of the canvas element that Phaser will create.

Loading Assets

```
function preload ()
{
    this.load.image('sky', 'assets/sky.png');
    this.load.image('ground', 'assets/platform.png');
    this.load.image('star', 'assets/star.png');
    this.load.image('bomb', 'assets/bomb.png');
    this.load.spritesheet('dude',
        'assets/dude.png',
        { frameWidth: 32, frameHeight: 48 }
    );
}
```

Phaser looks for the preload function to load assets.

Display an Image

```
function create ()
{
	this.add.image(400, 300, 'sky');
}
```

The values `400` and `300` are the x and y coordinates of the image. Why 400 and 300? It's because in Phaser 3 all Game Objects are positioned based on their center by default. The background image is 800 x 600 pixels in size, so if we were to display it centered at 0 x 0 you'd only see the bottom-right corner of it. If we display it at 400 x 300 you see the whole thing.

Hint: You can use `setOrigin` to change this. For example the code: `this.add.image(0, 0, 'sky').setOrigin(0, 0)` would reset the drawing position of the image to the top-left.

The order in which game objects are displayed matches the order in which you create them. So if you wish to place a star sprite above the background, you would need to ensure that it was added as an image second, after the sky image.

This tutorial is a trimmed down version of the official Making your first Phaser 3 game.

Tutorial 2: Sprite Size

Look for Reference

In order to determine a good sprite size, you should look for references at other games.

Screen Size

In order to determine a good screen size, the following question needs to be asked:

What device will the game be played on?

Full HD is 1920px by 1080px

Working from full HD by dividing numbers that evenly divide into HD.

384 x 216 (1/5 HD)
480 x 270 (1/4 HD)
960 x 540 (1/2 HD)
1,440 x 810 (3/4 HD)

If you do ¼ HD, then let the game view multiply everything by 4, it will display in HD.

At 480 x 270, 48x96 seems appropriate for a character. This is because 48 is 10% of 480 and 96 is 48x2. An attack would make sense at 96x96.

https://stuartspixelgames.com/2022/05/25/how-to-work-out-size-of-sprites-game-world/

Tutorial 3: Spritesheets

What is a Spritesheet?

The idea is to create one big image that contains all animations of a character instead of dealing with many single files.

Types of Spritesheets

Animation strips

An animation strip is the simplest form of a sprite sheet: It's just placing each animation frame next to each other. All frames have the same size, and the animation is aligned in each frame.

Tile sets

A tile set is not different from a sprite sheet: It contains building blocks for game levels. It's easy for a game to retrieve the sprites since they all have the same width and height. The disadvantage is that the sprites waste a lot of memory because of all the additional transparency.

Optimized Spritesheets

The easiest way is to remove the transparency surrounding the sprite and shrink it to the bounding box.

Sprite Software

Any image editor can work. Texture Packer, is designed specifically for spritesheets. Using Texture Packer will pack the sheets and create information files. Texture Packer has a 7 day free trial.

https://www.codeandweb.com/texturepacker/tutorials/how-to-create-a-sprite-sheet

Tutorial 4: Working with Animations

Adding the Spritesheet

```
function preload() {
    this.load.spritesheet("plane", "plane.png", { frameWidth: 512, frameHeight: 512
});
}
```

All assets are loaded in the preload. It is important we define the `frameWidth` and `frameHeight` as it tells our animator how large each frame is in our spritesheet.

Animations

Now we can create the animations:

```
function createScene() {
    this.anims.create({
        key: "fly",
        frameRate: 7,
        frames: this.anims.generateFrameNumbers("plane", { start: 3, end: 5 }),
        repeat: -1
    });

    this.anims.create({
        key: "explode",
        frameRate: 7,
        frames: this.anims.generateFrameNumbers("plane", { start: 0, end: 2 }),
        repeat: 2
    });
}
```

Let’s look at the animation that I’m calling `fly.` For this particular animation, we are cycling through our frames at a rate of seven frames per second. The frames in our spritesheet for this particular animation are defined by the start and end fields. If we look back at our image, we know that the actual plane is the 3rd through 5th image if looking at the spritesheet from left to right, top to bottom. The index of these frames starts at zero and not one. For this particular animation we are repeating until we specifically tell it to stop.

If we look at the other animation, we can see that it repeats 2 times and not until it's told to stop.

It is important to note that stopping an animation does not remove the sprite. We’ll get to that later. So this means when an animation stops, the sprite will stay there appearing frozen on the screen.

The Object

With two different animations defined, we need to associate them to an actual sprite game object. To do this, let’s create the following variable to represent our plane:

```
var plane;
```

The above variable should exist outside of any scene functions so that way it can be used throughout the game with the correct scope.

```
function create() {

    // Animation definitions here ...

    plane = this.add.sprite(640, 360, "plane");
    plane.play("fly");

}
```

The sprite is using the `plane` spritesheet and the `fly` animation associated to that spritesheet. Given the dimensions of our game canvas, the sprite is centered.

Changing Animations

Being able to change an animation is important because eventually you’ll want to respond to interactions within the game.

Rather than complicating this example with collisions and things like that, let’s change the animation based on a timer.

```
function create() {

    // Animation definitions here ...

    plane = this.add.sprite(640, 360, "plane");
    plane.play("fly");

      //Method 1
    setTimeout(() => {
        plane.play("explode");
    }, 3000);

      //Method 2 (Phaser way)
    this.time.addEvent({
        delay: 3000,
        callback: () => {
            plane.play("explode");
        }
    });

}
```

In the above code, in Method 1, we’re using a `setTimeout` in JavaScript to wait three seconds before playing a different animation on our sprite.

Both should accomplish the same thing. However, because of frame rate considerations, the `this.time.addEvent` may be the proper way to go as it might do calculations that the `setTimeout` does not.

Destroying Objects

The next step is to figure out when an animation ends and potentially even remove the sprite.

```
plane.once(Phaser.Animations.Events.SPRITE_ANIMATION_COMPLETE, () => {
    plane.destroy();
});
```

The above code says that once the animation stops, the plane should be destroyed, which means it is removed from the game.

https://www.thepolyglotdeveloper.com/2020/07/animate-spritesheets-phaser-game/

Tutorial 5: Tiles

What is a Tilemap?

A tilemap is a technique for creating a game world out of modular building blocks. This saves memory and performance.

The tilemap approach defines a set of modular, regularly-sized *tiles* that we can use to build our levels. That way, we only need one image, a *tileset.*

With tilemap editing software, we can easily configure properties of the tiles too. For example, we can mark some tiles — like the ground tiles — as solid tiles that Mario can stand on.

So with tilemaps, we've got a smaller image (performance & memory win) that we can use to easily create and iterate on level designs (creative win).

The Code

Inside of preload, we can load up the tileset image:

```
this.load.image("mario-tiles", "../assets/tilesets/super-mario-
tiles.png");
```

Inside the create function:

```
function create() {

  // Load a map from a 2D array of tile indices

  const level = [

    [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0 ],

    [  0,   1,   2,   3,   0,   0,   0,   1,   2,   3,   0 ],

    [  0,   5,   6,   7,   0,   0,   0,   5,   6,   7,   0 ],

    [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0 ],

    [  0,   0,   0,  14,  13,  14,   0,   0,   0,   0,   0 ],

    [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0 ],
    [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0 ],

    [  0,   0,  14,  14,  14,  14,  14,   0,   0,   0,  15 ],

    [  0,   0,   0,   0,   0,   0,   0,   0,   0,  15,  15 ],

    [ 35,  36,  37,   0,   0,   0,   0,   0,  15,  15,  15 ],

    [ 39,  39,  39,  39,  39,  39,  39,  39,  39,  39,  39 ]

  ];



  // When loading from an array, make sure to specify the tileWidth
and tileHeight

  const map = this.make.tilemap({ data: level, tileWidth: 16,
tileHeight: 16 });

  const tiles = map.addTilesetImage("mario-tiles");

  const layer = map.createLayer(0, tiles, 0, 0);

}
```

`level` is just a 2D array of numbers, or *indices*, that point to a specific tile from our tileset. 0 is the top left tile, 1 is the one next to it, etc.

Note: an index that is less than zero is considered an empty tile.

Breaking down that code, we've got three main parts: a `Tilemap`, a `Tileset` and a `TilemapLayer`. You create a `Tilemap` through `this.make.tilemap` (or `this.add.tilemap`) - you can think of this as a container for tilesets & tilemap layers. A tileset stores information about each tile in the map. A map can then be composed of one or more layers (`TilemapLayer` instances), which are the display objects that actually render tiles from a `Tileset`.

Loading from a File: CSV

```
function preload() {

  this.load.image("tiles",
"../assets/catastrophi_tiles_16_blue.png");

  this.load.tilemapCSV("map", "../assets/catastrophi_level2.csv");
}

function create() {

  // When loading a CSV map, make sure to specify the tileWidth and
tileHeight!

  const map = this.make.tilemap({ key: "map", tileWidth: 16,
tileHeight: 16 });

  const tileset = map.addTilesetImage("tiles");

  const layer = map.createLayer(0, tileset, 0, 0); // layer index,
tileset, x, y

}
```

Building a Map in Tiled

Loading from a 2D array or CSV is great when you want to test out something simple or you are generating a procedural world, but odds are, you'll want a level design tool. That's where Tiled comes in. It's a free, open source tilemap editor that can export to CSV, JSON and a bunch of other formats.

When working with Tiled to generate maps for Phaser, there are a few things you'll want to make sure to do:

1. When you load a tileset into your map, make sure to check the "Embed in map" option. (If you forget to do this, then you can click the embed tileset button the bottom of the screen.)
2. Make sure you aren't using a compressed "Tile Layer Format." You can adjust that in map properties sidebar… which you can open by hitting "Map → Map Properties" in the top toolbar.
3. When you export your map, save it as a JSON file.

Loading a Tiled Map

Using the tilemapTiledJSON loader method, we can load up and display a tilemap that we've exported from Tiled:

```
function preload() {
  this.load.image("tiles", "../assets/tilesets/tuxmon-sample-32px-
extruded.png");
  this.load.tilemapTiledJSON("map", "../assets/tilemaps/tuxemon-
town.json");
}
```

```
function create() {
  const map = this.make.tilemap({ key: "map" });

  // Parameters are the name you gave the tileset in Tiled and then
the key of the tileset image in
  // Phaser's cache (i.e. the name you used in preload)
  const tileset = map.addTilesetImage("tuxmon-sample-32px-extruded",
"tiles");

  // Parameters: layer name (or index) from Tiled, tileset, x, y
  const belowLayer = map.createLayer("Below Player", tileset, 0, 0);
  const worldLayer = map.createLayer("World", tileset, 0, 0);
  const aboveLayer = map.createLayer("Above Player", tileset, 0, 0);
}
```

This is a common design pattern when working with Tiled. It allows us to separate out elements to be placed at different "depths" in the game. With these layers, we can ensure the "Below Player" layer (the ground & path) are displayed under the player sprite and the "Above Player" layer (roof/statue/sign tops) are displayed on top of the player sprite. The "World" layer has all the rest of the stuff, including the colliding/solid stuff in the world.

https://medium.com/@michaelwesthadley/modular-game-worlds-in-phaser-3-tilemaps-1-958fc7e6bbd6

Tutorial 6: Physics

Moving with Physics

Now that we've got a world, we can add a proper character to the world and have them walk around with physics. There are currently three physics engines that are integrated into Phaser: arcade physics (AP), matter.js and impact. AP is fast and simple, so that's where we'll start — we'll get to matter.js later.

In AP, you can create physics bodies that are either rectangles or circles. Rectangle bodies are axis-aligned bounding boxes, which roughly means they can't be rotated. Colliding tiles in our map loaded up with AP will be given a rectangular body that matches the size of the tile.

There are four things we'll need to do:

1. Mark certain tiles in the `worldLayer` as colliding so that AP knows to use them for collisions.
2. Enable the AP physics engine.
3. Create a physics-based sprite for the player.
4. Set the player to collide with the `worldLayer`.

The first step to use a tilemap with physics is that you need to mark which tiles should be solid ("colliding"). One way to do that would be to mark certain tile indices as colliding within a layer:

```
// The 13th tile through and including the 45th tile will be marked
as colliding

worldLayer.setCollisionBetween(12, 44);
```

If you are working with tile indices, then there's setCollision, setCollisionBetween and setCollisionByExclusion. But thinking in terms of indices is hard, so there's a better way: setCollisionByProperty. Tiled allows you to add properties to a tileset via the Tileset Editor, so we can just mark which tiles collide directly in Tiled.

Open up the Tileset Editor by clicking on the "Edit Tileset" button (at the bottom right of the screen).

1. Click and drag (or CTRL + A) to select all the tiles.
2. Under the properties window (left side of the screen), click the plus icon and add a boolean property named "collides."
3. Select only the tiles that you want to collide and set "collides" to true by checking the box
4. Re-export your map.

Back inside of Phaser, we can simply do the following to mark our colliding tiles within worldLayer:

```
worldLayer.setCollisionByProperty({ collides: true });
```

Once we’ve got tiles marked as colliding, we can add physics. In our game’s config, we can turn on the arcade physics engine by doing the following:

```
const config = {

  // ... (rest of the config from earlier)

  physics: {

    default: "arcade",

    arcade: {

      gravity: { y: 0 } // Top down game, so no gravity

    }

  }

};
```

Tutorial 7: Basic Movement

```
let player;

function create() {
  player = this.physics.add.sprite(400, 350, "atlas", "misa-front");
}

function update(time, delta) {
  // Stop any previous movement from the last frame
  player.body.setVelocity(0);

  // Horizontal movement
  if (cursors.left.isDown) {
    player.body.setVelocityX(-100);
  } else if (cursors.right.isDown) {
    player.body.setVelocityX(100);
  }

  // Vertical movement
  if (cursors.up.isDown) {
    player.body.setVelocityY(-100);
  } else if (cursors.down.isDown) {
    player.body.setVelocityY(100);
  }

  // Normalize and scale the velocity so that player can't move
faster along a diagonal
  player.body.velocity.normalize().scale(speed);
}
```

Tutorial 8: KeyBoard Input

```
//Define a global variable
let keyJ;

create () {
        //Define the key
        keyJ = this.input.keyboard.addKey(Phaser.Input.Keyboard.KeyCodes.J)
}

controls () {
        if (keyJ.isDown) {
                //Call function or code that you want to run
        }
}
```

https://stackoverflow.com/questions/58383619/how-to-use-a-s-d-w-keys-in-phaser

Tutorial 9: Web Audio

Basics

Web Audio is usually started in a `suspended` state. The browser generally won't allow audio to play without some kind of user interaction indicating that the user wanted to play the game.

This is a good thing. It prevents *overeager* ads hidden away in a tab you've forgotten about from suddenly starting to play audio while you are on a video call.

Phaser automatically creates an `AudioContext` when your game starts even though no sounds will be played.

Phaser will automatically resume this context when the game is given focus from user interaction.

Then later, when the player clicks on another window or switches tabs, Phaser will automatically suspend the `AudioContext`. Upon regaining focus, Phaser will also automatically resume the context.

Playing Audio Immediately

Almost all games on other platforms will start playing background music as soon as it can regardless of whether the player is ready or not.

But you cannot do this with a web game. There's no way around it and if you did find one then rest assured that the browser makers will release a *fix* to correct you.

The best solution is to design your game so that it receives user input before it is important for audio to be playing.

Multiple Context Warrnings

It is normal and expected to see 1 `AudioContext was not allowed to start.` warning in the Browser Console.

Seeing more than one is also generally okay but it is possible to get rid of them. Some among us are made uncomfortable by too many warnings.

The reason for multiple `AudioContext` warnings is likely because the game is trying to play audio before the user has interacted with it

```
 create()
{
     this.music =  this.sound.add('music', {
          volume: 0.2,
          loop: true
     })

     this.music.play()
}
```

Seems innocent enough and everything will likely work fine. The music will play as soon as the user interacts with your game. But that is the cause for the extra warnings that keep you up at night.

The fix is to listen for the `Phaser.Sound.Events.UNLOCKED` event before calling `this.music.play()`.

```
create()
{
     this.music =  this.sound.add('music', {
          volume: 0.2,
          loop: true
     })

     if (!this.sound.locked)
     {
          // already unlocked so play
          this.music.play()
     }
     else
     {
          // wait for 'unlocked' to fire and then play
          this.sound.once(Phaser.Sound.Events.UNLOCKED, () => {
               this.music.play()
          })
     }
}
```

The Difference with IOS

On a desktop browser, clicking on a different window will stop your game's audio. Then clicking back will resume it.

You may find that this is not the case on iOS.

You can tap on your game and music will start. Go to a different tab and music will stop. So far so good. Working like a desktop browser.

But when you go back to the tab with your game and tap it… nothing happens. The music does not resume.

You can log the `state` of the `AudioContext` and it will report: `suspended`.

If your game doesn't fill the entire browser window then you may notice that tapping on *empty space outside of the game* will resume audio. You can also tap on the address bar and then cancel or some other menu item and cancel to get the music to resume.

This can be solved with a hack where you get a reference to the `AudioContext` and then manually resume it on a `POINTER_DOWN` event.

For a quick fix, this hack might be fine. But there is another approach that's cleaner and more user-friendly.

There is a better way to fix things on IOS, but this will not be covered. For more info, you can click on the reference link.

https://blog.ourcade.co/posts/2020/phaser-3-web-audio-best-practices-games/