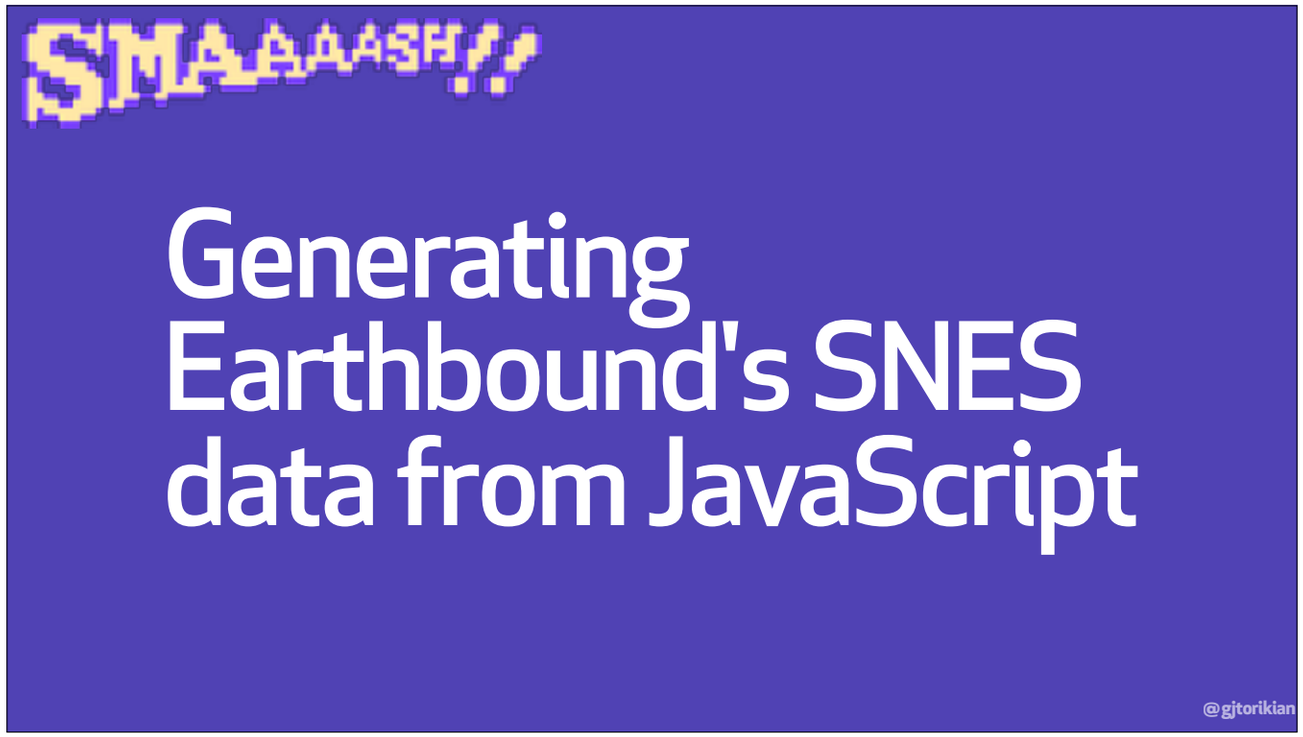
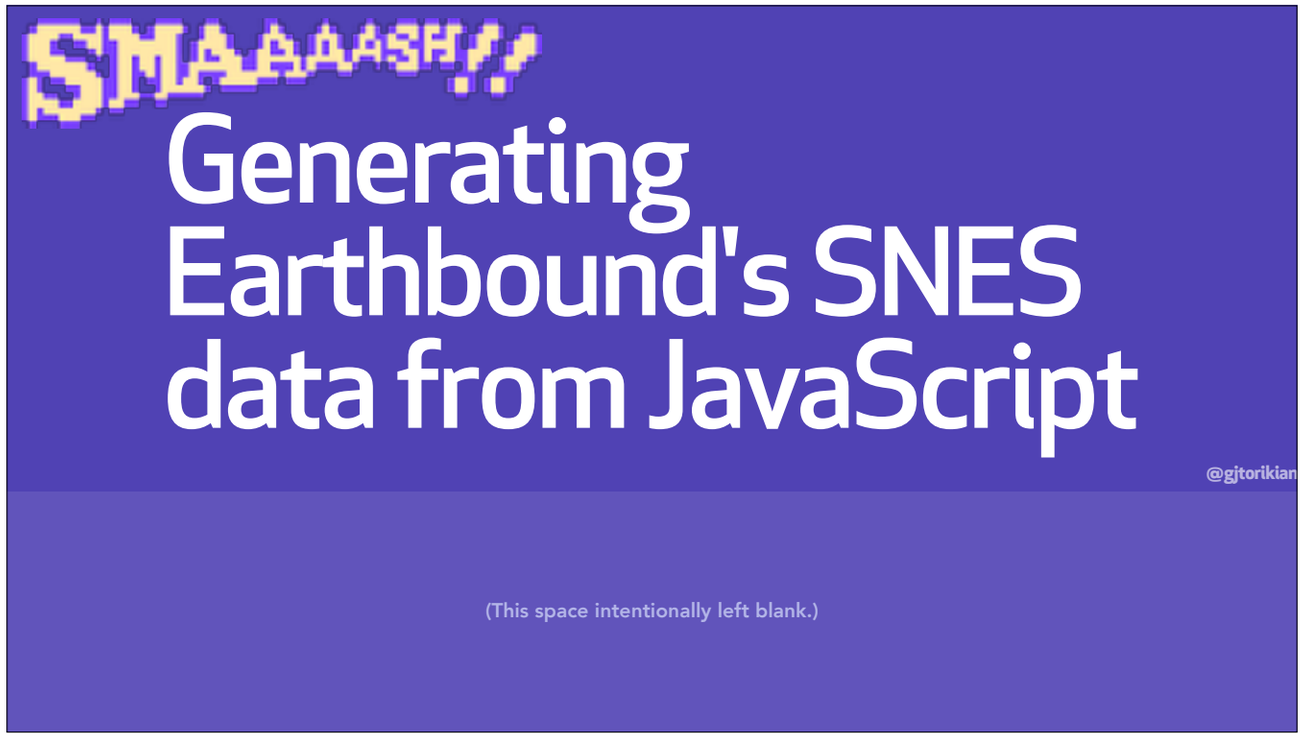
Generating Earthbound's SNES data from JavaScript

@gjtorikian



All right, hi hello. My name is Garen Torikian, I'm @gjtorikian everywhere on the Internet. Uh, before I begin, when my talk got accepted, one of the organizers, Mariko, told me that the bottom third of every slide can't be seen by people in the back

NEXT



Which I thought was a nice thing to inform me about. So the reminder of this talk will be presented on the first two-third of every slide.

The name of this talk is Generating Earthbound's SNES data from JavaScript, everyone's favorite programming language in the browser.

Before I get too far into it, just, with a show of hands, could people let me know who is familiar with this game, Earthbound, at all.

Japan: 1994

USA: 1995

Europe: n/a

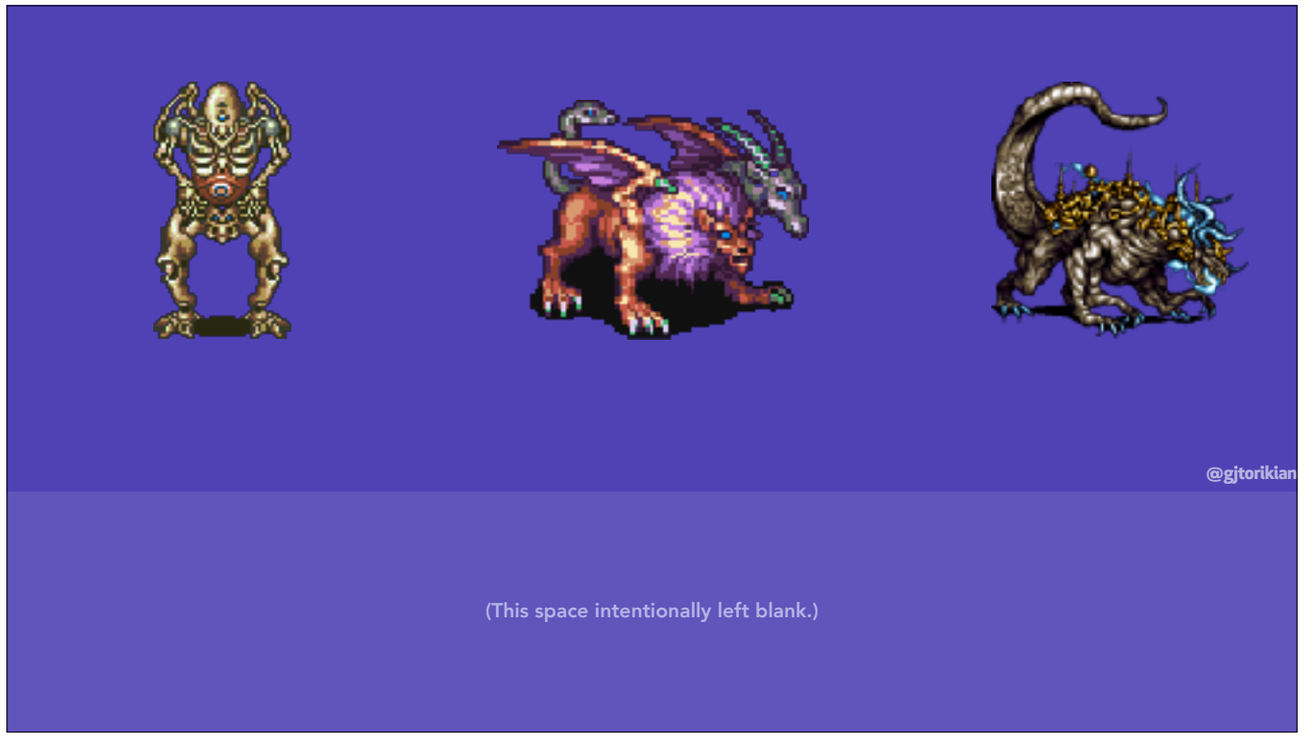
@gjtorikian

(This space intentionally left blank.)

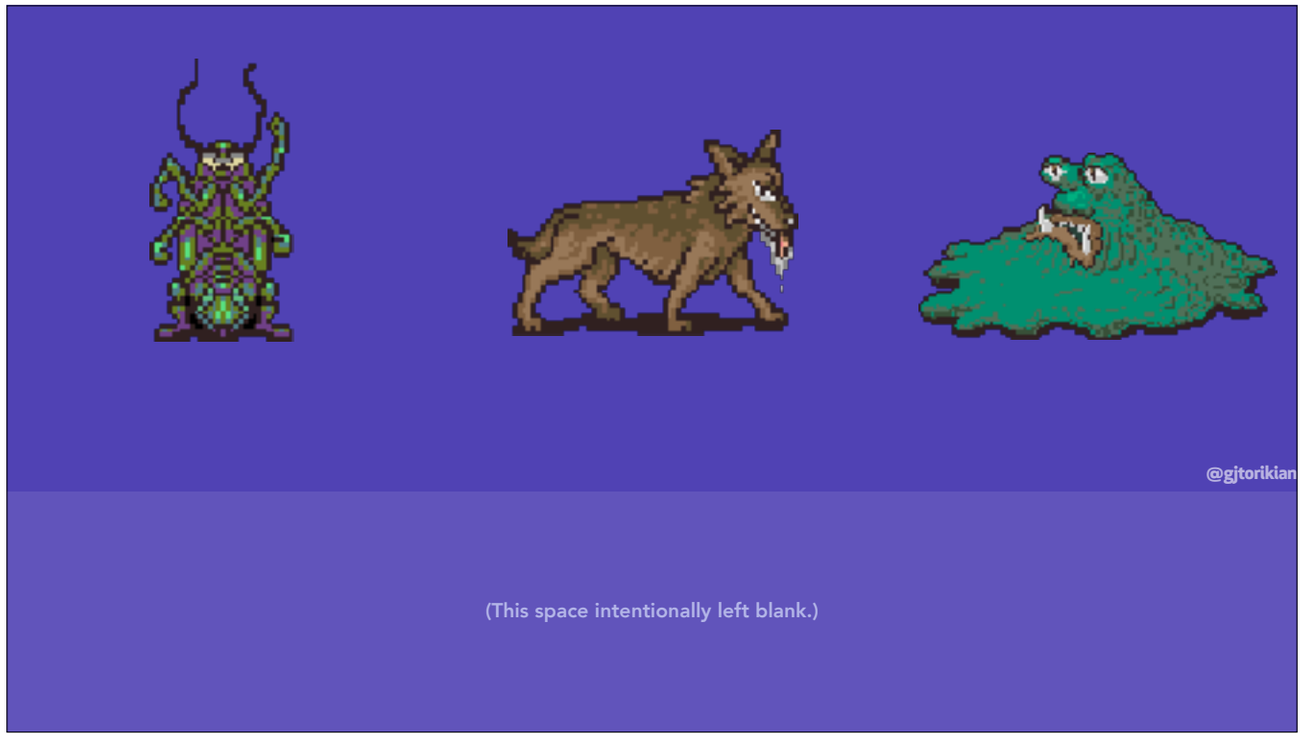
Fantastic! Earthbound was this amazing role playing game that originally came out in 94/95 for the Super Nintendo. In Japan, it was actually the middle game in a trilogy, and in America, we never officially got the first or third part of that trilogy. Actually, in the States, the game was a huge flop. It was produced by Nintendo, they spent something like two million dollars on marketing, and the game just never took off.
I wouldn't say it was a hard game, but it was definitely an extremely weird game. If you haven't played, I highly recommend it, I think that as an adult it's much, much funnier.

(This space intentionally left blank.)

One of the things that was a bit interesting about the game is that it took place in "the real world." Maybe this is a quaint idea right now, but back then most RPGs at the time would put you in a fantasy setting: you would be a knight, or some warrior from outer space, but in Earthbound you pretty much just walked around cities.



And to follow that, while in most games at the time you might be battling skeletons, chimeras, behemoths…

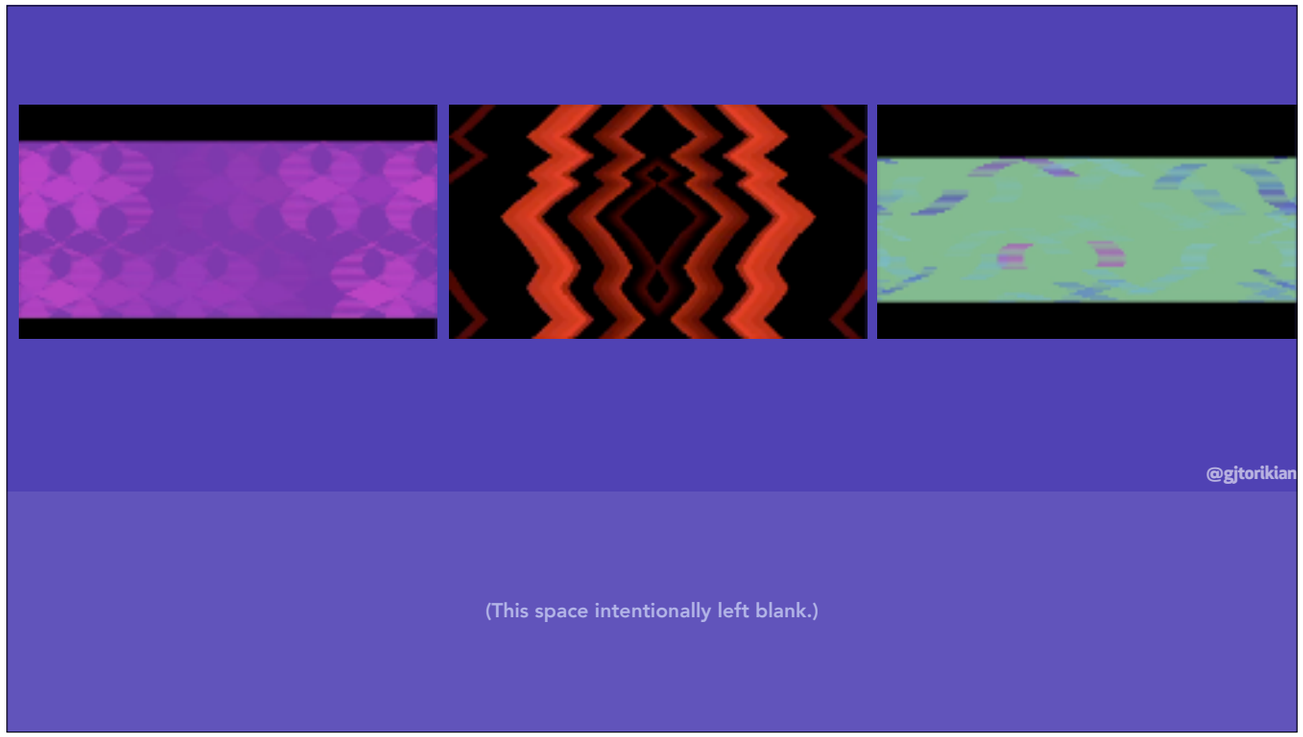


In Earthbound your main enemies were roaches, wolves, and a large, sentient pile of barf.

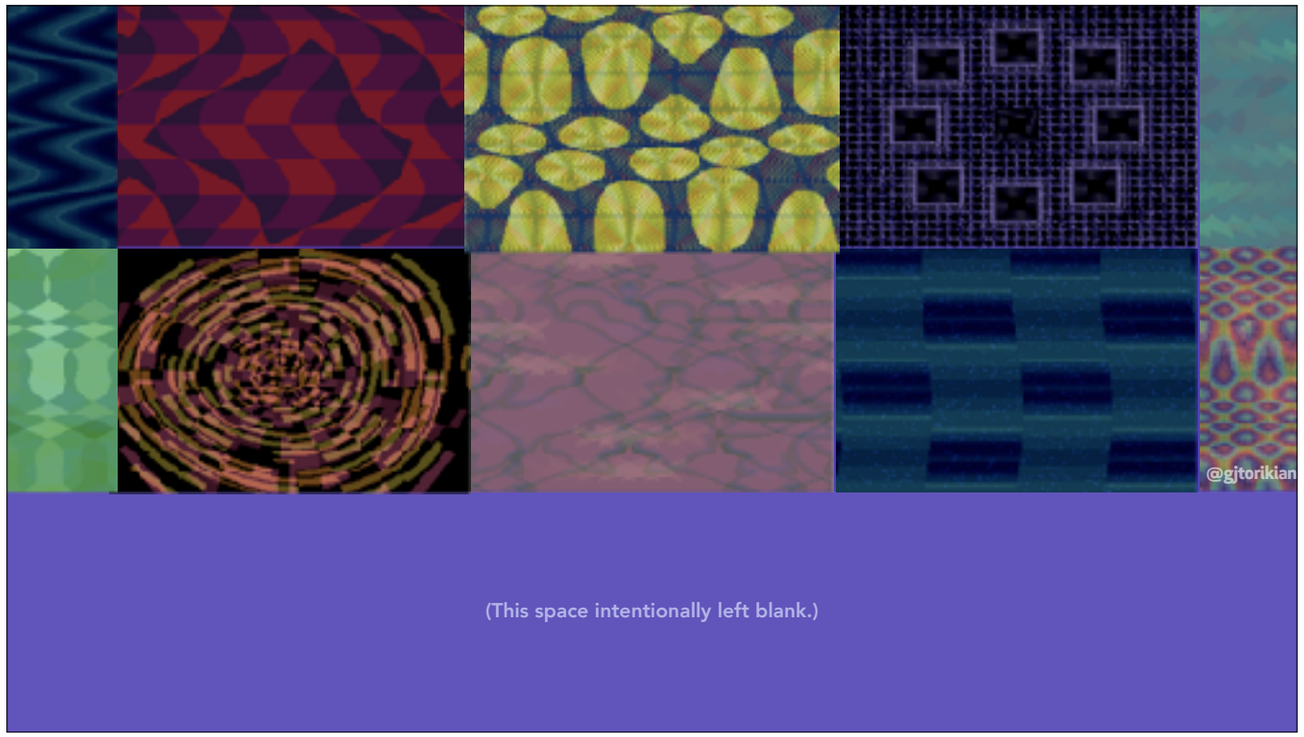
@gjtorikian

(This space intentionally left blank.)

But it wasn't just the enemies that set the game apart. Another thing that made this game so fascinating was some of the artwork. Specifically

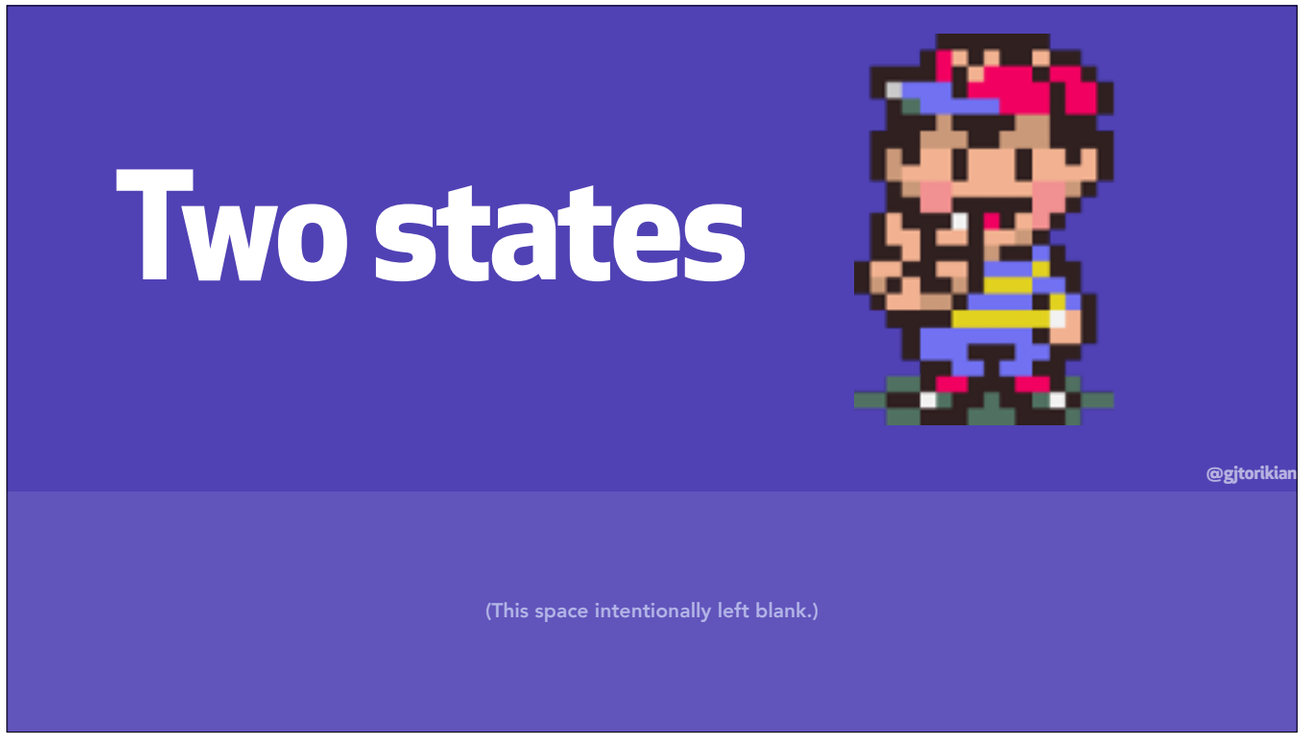


The background sequences whenever you encountered an enemy. They moved, they changed shape and color, they had different aspect ratios.

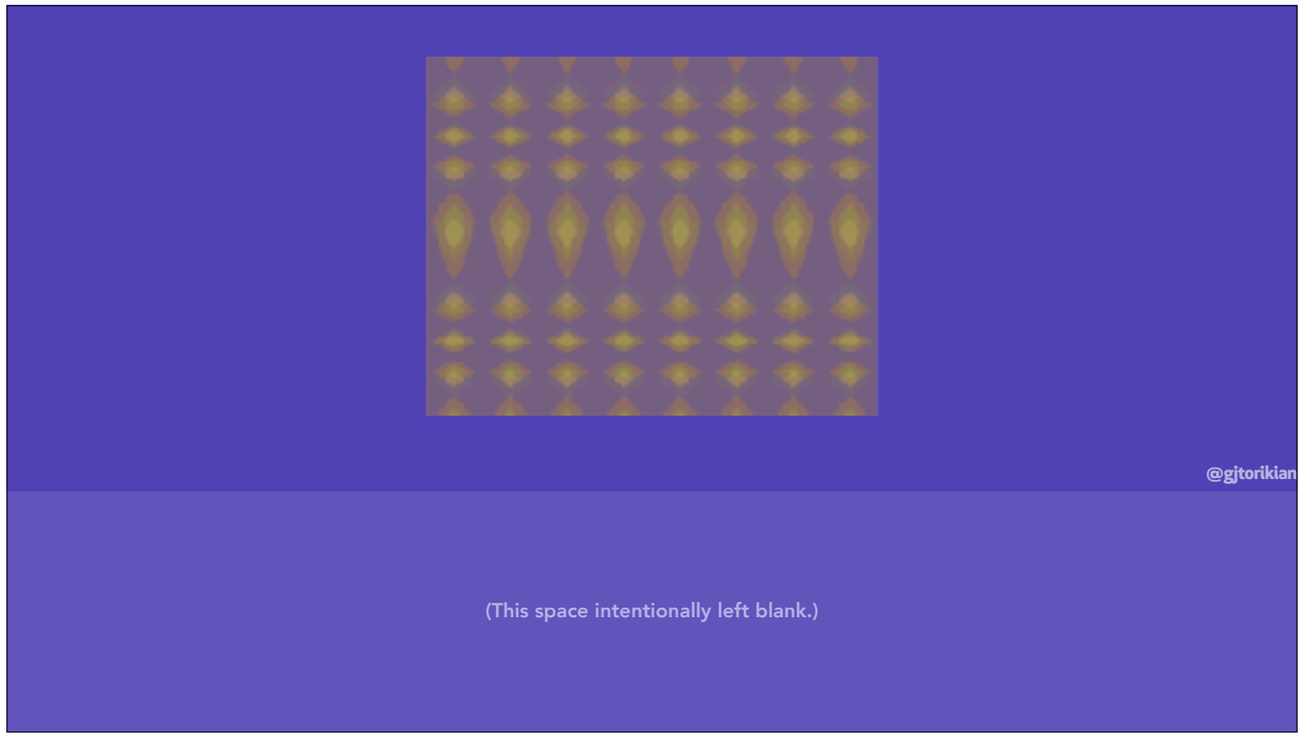


In point of fact, there are well over 300 different backgrounds that the little SNES cartridge could cycle through. They're all hypnotic, right? Like lava lamps. They go left to right, up to down, they spin, they change colors.

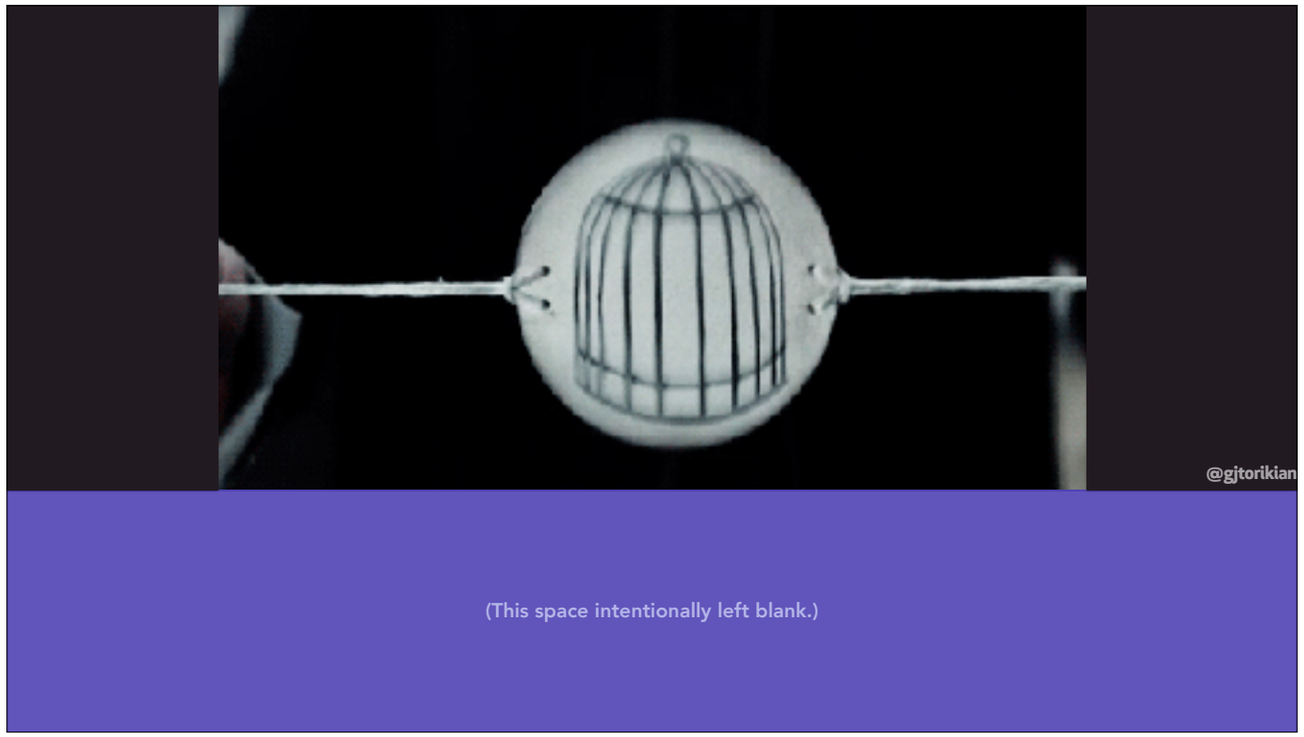
So a few years ago I got really interested in how these were being generated. They couldn't be dynamic, on the fly, because an SNES wouldn't have been able to produce that content. So, were they static images, or what?



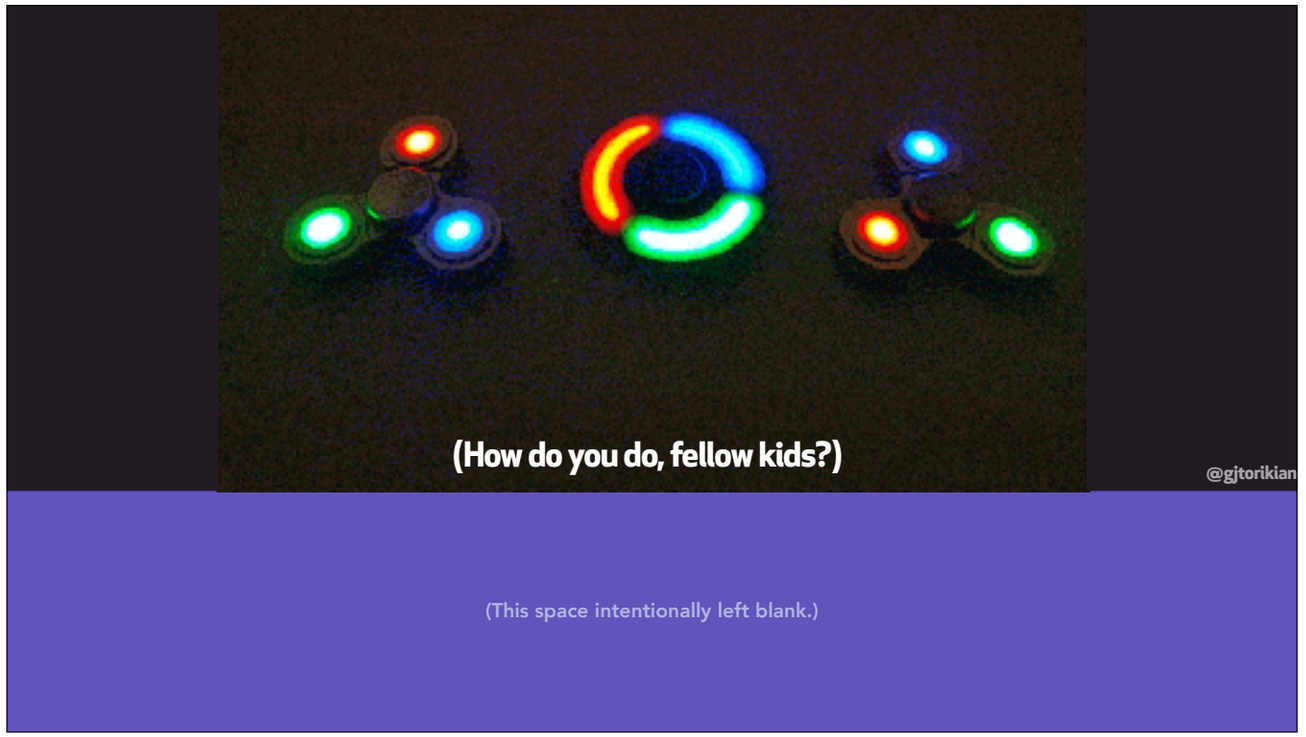
Let's detour into a quick analogy. If you look at this moving GIF, you can tell that it's comprised of two states, right? The standing relaxed one, and the cheerful peace sign one.



Here's an example of one of the battle backgrounds. If this were a looping GIF, how many different sequences or frames of images would be needed to generate this? Dozens or maybe hundreds?

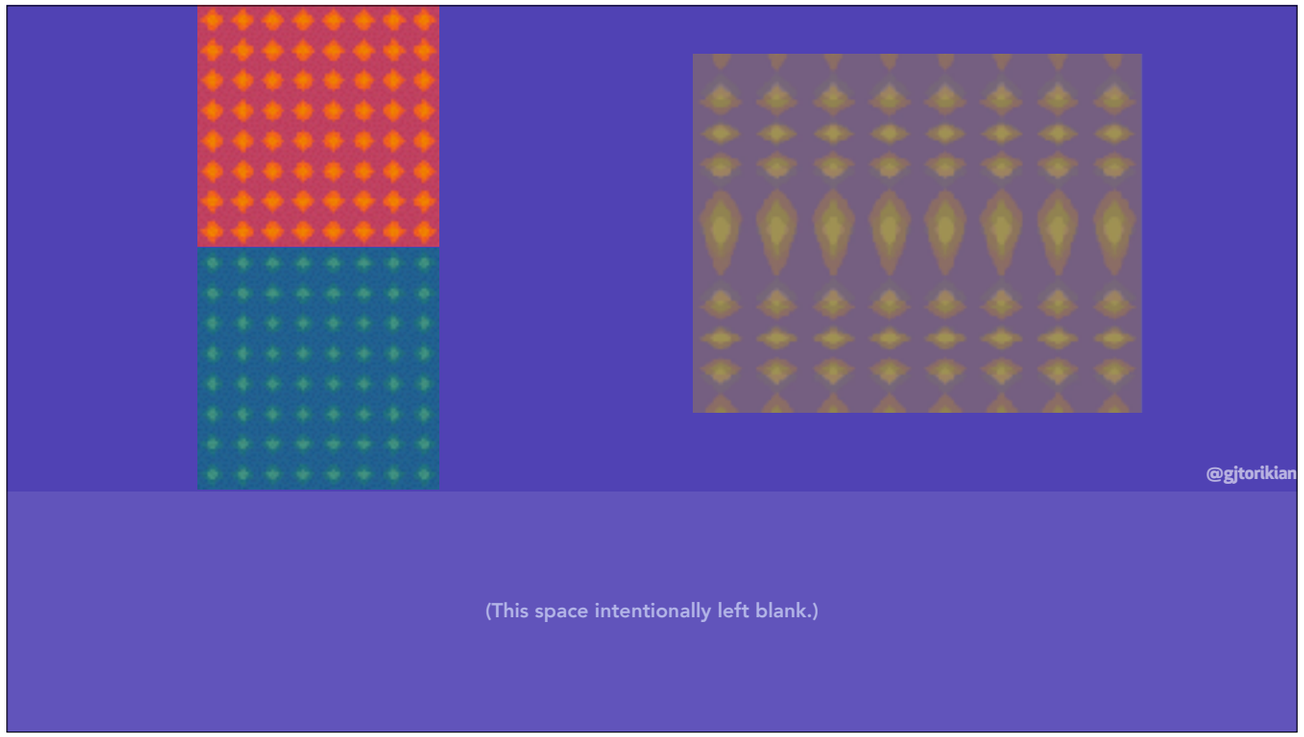


It turns out that the Earthbound uses a sort of trick based on a Zoetrope. If you're not familiar with it, a Zoetrope is a device where images are repeatedly swapped so quickly that your mind thinks that it's all happening in one fluid motion. It's the basis for most early animations in the 1920s.

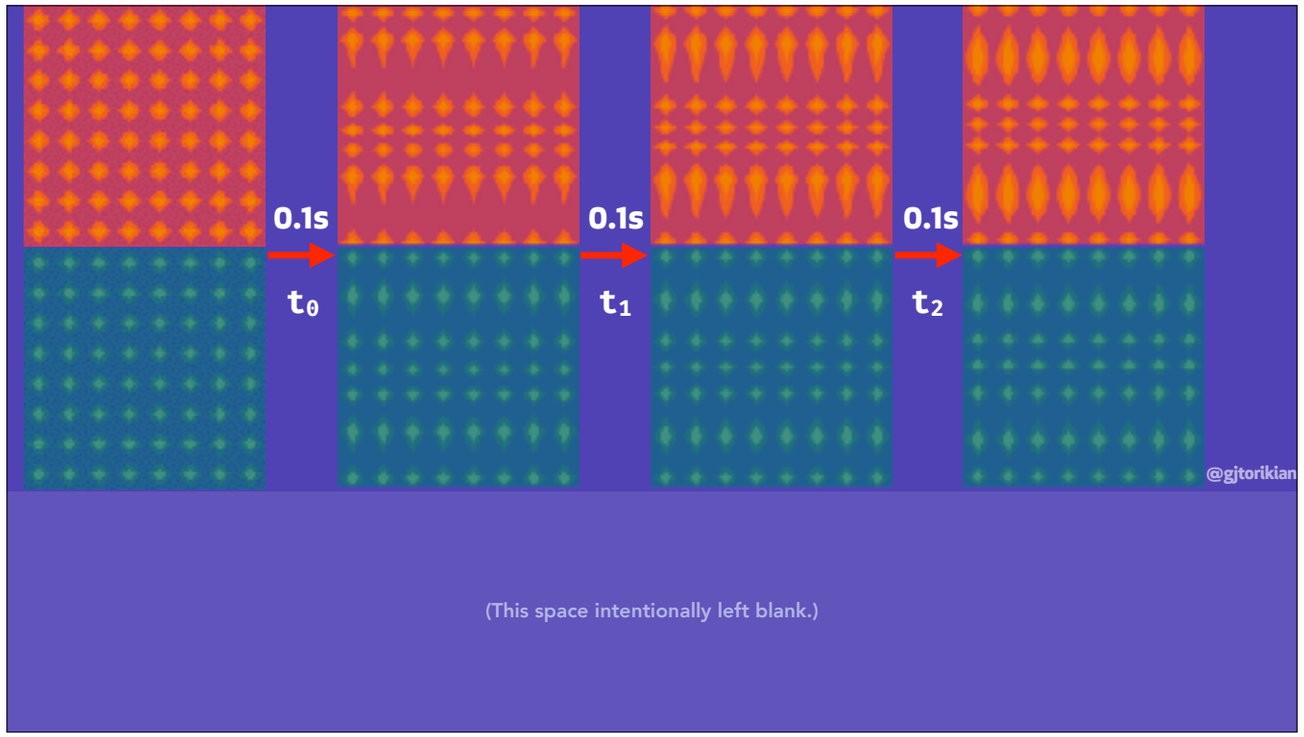


(This space intentionally left blank.)

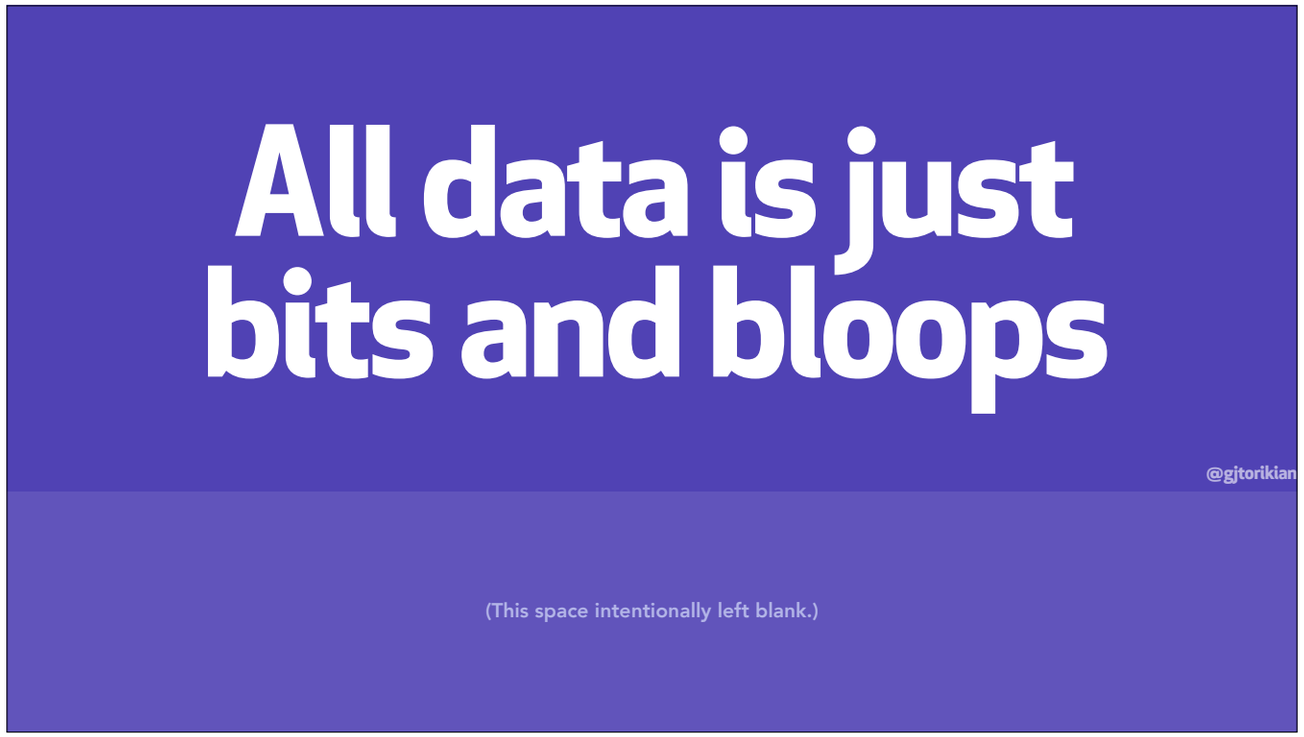
And as much as I hate to use it, a fidget spinner is a great example of this. A spinner has three distinct circles, but if you spin them fast enough, your mind stretches the lights into bands of arced and bent colors.



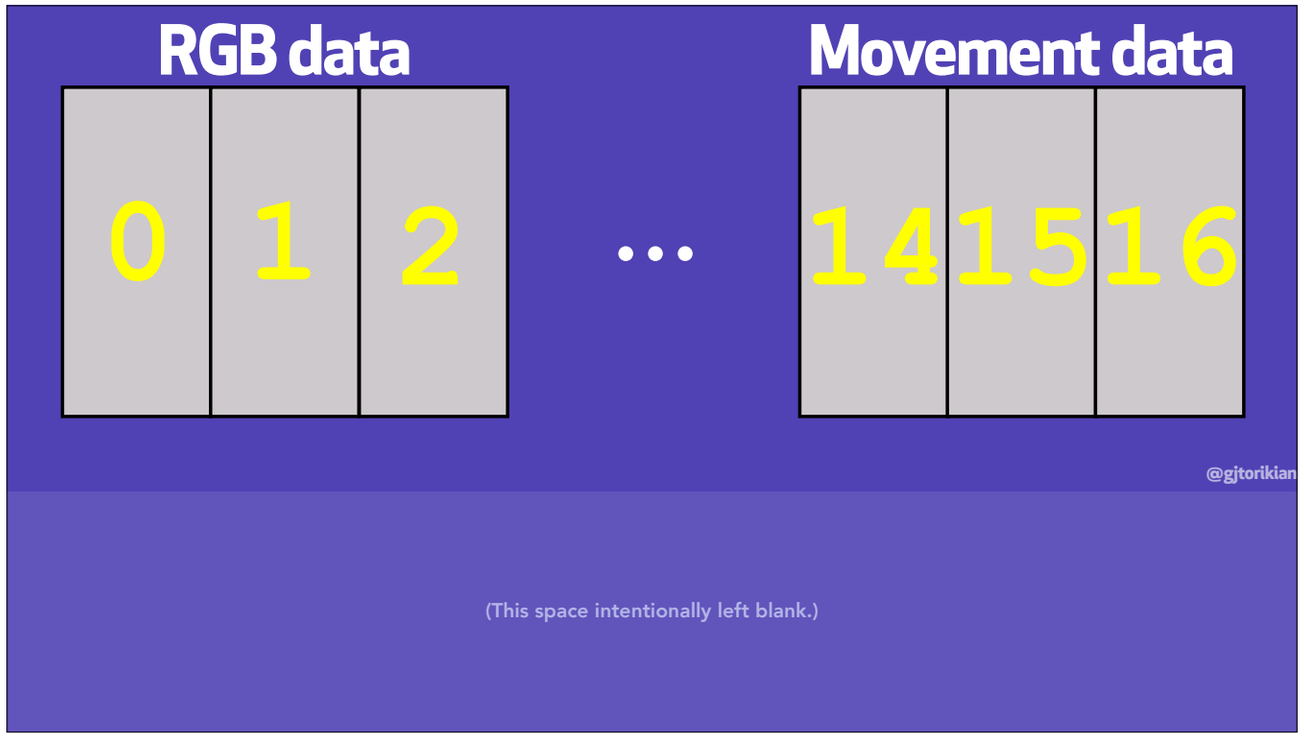
It turns out, the battle backgrounds are always **CLICK** just two images placed on top of each other.
The SNES cartridge stores these backgrounds as layers — there are 327 different versions of those simple squares on the left. Each of those squares has embedded within it not only a different design, but also, different rules on how to shimmer and move.



After these two layers are placed on top of each other, they are constantly being distorted. So one function is changing the image at the top and another is changing the image at the bottom. Even though they both look like diamond patterns, the bottom row has a function that stretches the pixels less, and when they applied on top of each other **CLICK** it gives the effect of a moving image.
So it's not a video and it's not different frames in a gif. It's the same picture being disrupted using the same function so quickly that it appears to be moving.



How is this information stored on the SNES cartridge? All data in a computer is just bits and bloops. All of the characters, maps, power-ups, game mechanics, sound effects, the music, they are all programmed somewhere in a cartridge. They're all stored somewhere.



HOPE IT IS FIVE MINUTES NOW.

Information about each battle background on an SNES 17 "segments" long of memory. Suffice to say, every single background encodes its color and movement. So what's stored on the cartridge is just a bunch of static square patterns with some functions to apply to the picture describing the movement.

Ok, by now you're probably like, Dude, this guy has been blabbing on about SNES cartridges and image distortions — right, like, Why is he here? Where is the JavaScript?

<canvas>

requestAnimationFrame()

@gjtorikian

(This space intentionally left blank.)

It turns out that in JavaScript we can basically emulate the same behavior that the Super Nintendo is doing in the browser with just a canvas and the requestAnimationFrame function

```
import data from "rom.dat";

const backgroundData =
      new Uint8Array(Array.from(data));
```

e42b 0002 0300 ff3a	0083 002c 01c7 3f37
0007 1f00 f807 8f7f	cfc0 2d00 8500 4f0b

@gjtorikian

(This space intentionally left blank.)

An SNES cartridge is just bits and bloops, right? That means it can be read. So here, we can abuse JavaScript's import function to just load the SNES cartridge data directly.
"rom.dat" is just a straight dump of the SNES cartridge data. And once we get that entire file, we can convert each element into a simple array.
Those hexadecimal codes in the rectangle at the bottom represent what is actually being loaded—each sequence encodes an entires image and its movement information, and each character is a single element in the backgroundData array.

```
for (let bg_idx = 0; bg_idx <= 326; ++i) {
 // Process each image, in sequences of 17
 // 0 * 17 = 0, 1 * 17 = 17, 2 * 17 = 34…
 this.bbgData[bg_idx] =
   new BattleBackground(backgroundData[i * 17]);
}
```

@gjtorikian

(This space intentionally left blank.)

Now that we have loaded the entire ROM into a single array called backgroundData, we can chunk each background into lengths of 17. Remember that all the information about an image is kept in a span of 17 elements.
So as we're looping through the chunks of ROM, we can assemble another array, called bbgData, to store all the information we are finding.

```
this.bbgData = [
  {
   bg0_image_data: { ... },
   bg0_movement_effect: function(...)
  },
  ...
  {
   bg326_image_data: { ... },
   bg326_movement_effect: function(...)
  }
]
```

@gjtorikian

(This space intentionally left blank.)

What ends up happening is that the bbgData becomes an array of JSON objects. Each object has a key which stores the pattern for each and every pixel on a background. And it also stores the distortion effects I sort of skipped over earlier. Each background layer knows how it needs to be manipulated, whether that's horizontally or vertically.

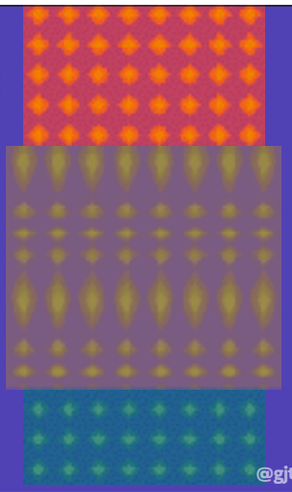
```
const layer1 = this.bbgData[x];
const layer2 = this.bbgData[y];

const engine = new Engine([layer1, layer2], {
 canvas: document.querySelector("canvas")
})
engine.animate()
```

@gjtorikian

(This space intentionally left blank.)

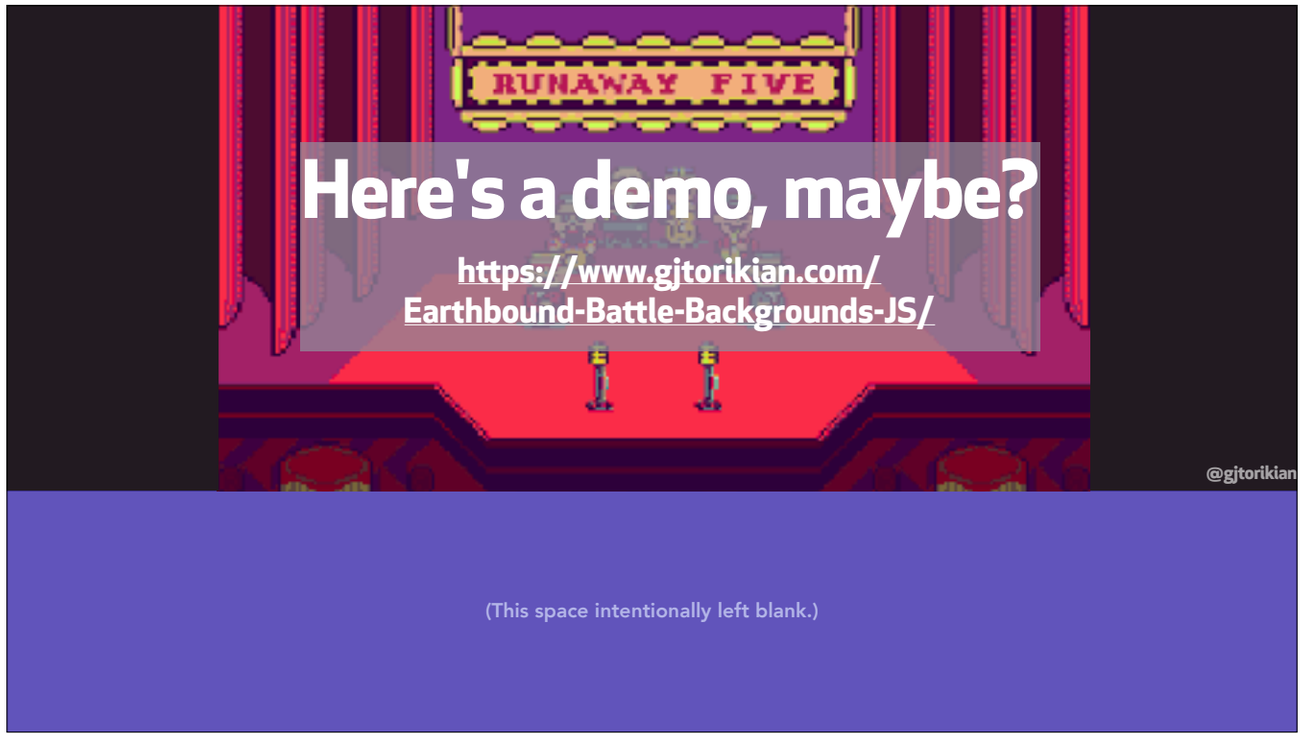
From there, we can take any two backgrounds in the bbgData array and begin to animate them. Here, we've got layers one and two chosen at random from the bbgData array, and we pass it to the engine to be animated.

```
// Grab current canvas data
const image = canvas.getImageData(0, 0, canvas.width, canvas.height)
const drawFrame = () => {
 // Tell the browser: "I'm going to animate something!"
 frameID = requestAnimationFrame(drawFrame)
 // Draw first layer's image data, manipulating it based on the time
 bitmap = layer_one.generateImage(image.data, this.tick)
 // Draw second layer's image data over the first, also manipulated
 bitmap = layer_two.generateImage(image.data, this.tick)
 // Set the image data to the current drawing & write it to the canvas
 image.data.set(bitmap); context.putImageData(image, 0, 0)
}; drawFrame(); // Animate (regenerate and draw) endlessly.
```



(This space intentionally left blank.)

Encoding in each of these layers, layer one and layer two, we know how each image needs to be distorted, given a specific moment in time, or tick. And what ends up happening in JavaScript is that we can make use of the `requestAnimationFrame` function to keep drawing on an HTML canvas.
Esentially, given the current image state and a moment in time, we are calling the same function to distort it repeatedly.



(This space intentionally left blank.)

*** IF YOU ARE IN THE AUDIENCE, DO NOT READ THIS ***

(This space intentionally left blank.)