40| 实战（四）：如何实现3D地球可视化（上）？

2020-10-09 月影

跟月影学可视化 进入课程 >

讲述：月影
时长 09:34 大小 8.77M

你好，我是月影。

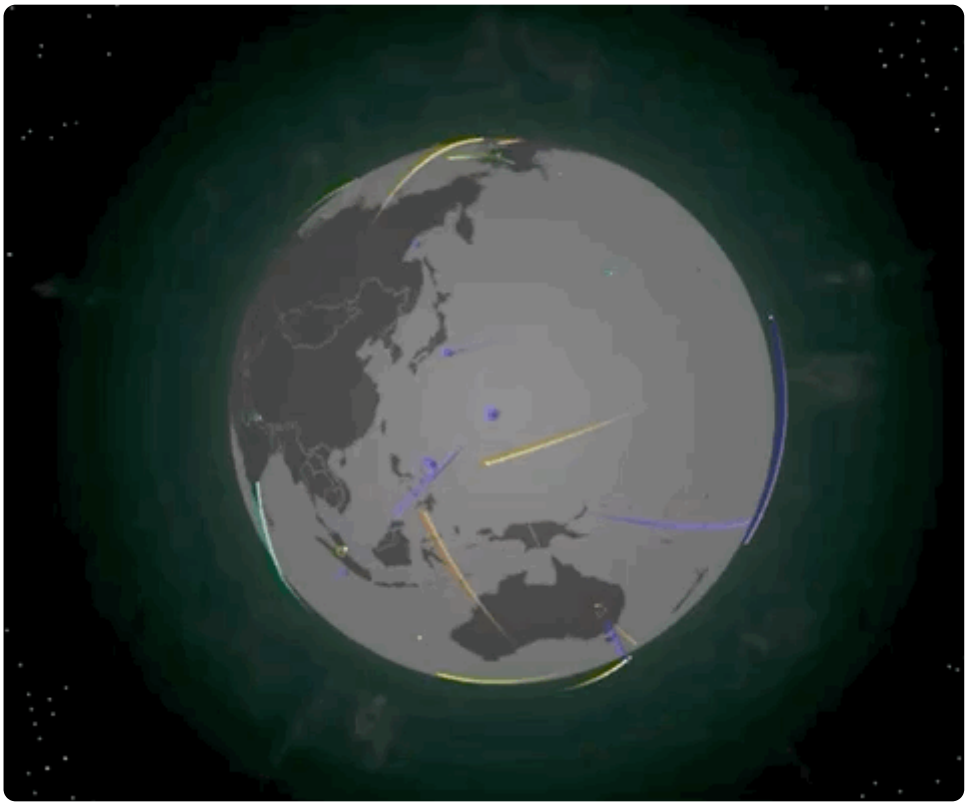
前几节课我们一起进行了简单图表和二维地图的实战，这节课，我们来实现更炫酷的 3D 地球可视化。

3D 地球可视化主要是以 3D 的方式呈现整个地球的模型，视觉上看起来更炫酷。它是可视化应用里常见的一种形式，通常用来实现全球地理信息相关的可视化应用，例如全球黑客攻防示意图、全球航班信息示意图以及全球贸易活动示意图等等。

因为内容比较多，所以我会用两节课来讲解 3D 地球的实现效果。而且，由于我们的关注点在效果，因此为了简化实现过程和把重点聚焦在效果上，我就不刻意准备数据了，我们

用一些随机数据来实现。不过，即使我们要实现的是包含真实数据的 3D 可视化应用项目，前面学过的数据处理方法仍然是适用的。这里，我就不多说了。

在学习之前，你可以先看一下我们最终要实现的 3D 地球可视化效果，先有一个直观的印象。



如上面动画图像所示，我们要做的 3D 可视化效果是一个悬浮于宇宙空间中的地球，它的背后是一些星空背景和浅色的光晕，并且地球在不停旋转的同时，会有一些不同的地点出现飞线动画。

接下来，我们就来一步步实现这样的效果。

如何实现一个 3D 地球

第一步，我们自然是要实现一个旋转的地球。通过前面课程的学习，我们知道直接用 SpriteJS 的 3D 扩展就可以方便地绘制 3D 图形。这里，我们再系统地说一下实现的方法。

1. 绘制一个 3D 球体

首先，我们加载 SpriteJS 和 3D 扩展，最简单的方式还是直接使用 CDN 上打包好的文件，代码如下：

复制代码

```
<script src="http://unpkg.com/spritejs/dist/spritejs.js"></script>
<script src="http://unpkg.com/sprite-extend-3d/dist/sprite-extend-3d.js"></scr
```

加载完成之后，我们创建场景对象，添加 Layer，代码如下：

复制代码

```
const {Scene} = spritejs;
const container = document.getElementById('container');
const scene = new Scene({
  container,
});
const layer = scene.layer3d('fglayer', {
  alpha: false,
  camera: {
    fov: 35,
    pos: [0, 0, 5],
  },
});
```

与 2D 的 Layer 不同，SpriteJS 的 3D 扩展创建的 Layer 需要设置相机。这里，我们设置了一个透视相机，视角为 35 度，位置为 0, 0, 5

接着是创建 WebGL 的 Program，我们通过 Layer 对象的 createProgram 来创建，代码如下：

复制代码

```
const {Sphere, shaders} = spritejs.ext3d;


const program = layer.createProgram({
  ...shaders.GEOMETRY,
  cullFace: null,
});
```

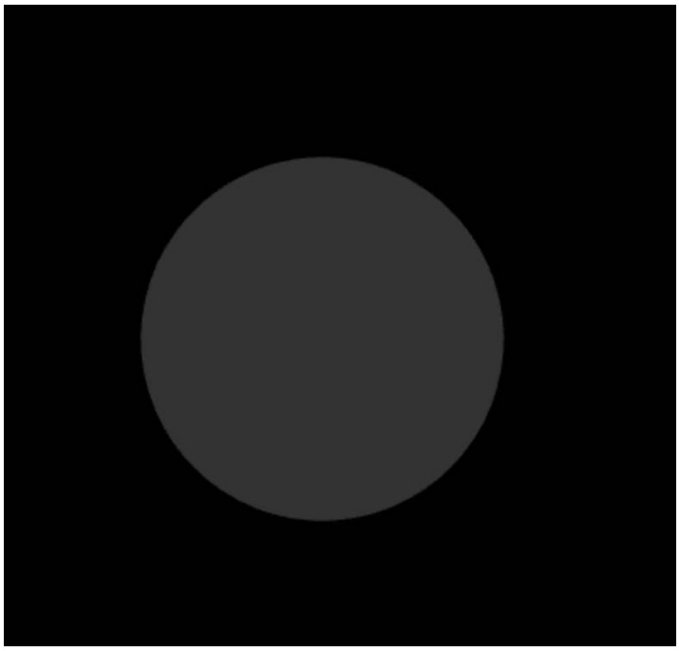
SpriteJS 的 3D 扩展内置了一些常用的 Shader，比如 shaders.GEOMETRY 就是一个符合 Phong 反射模型的几何体 Shader，所以这次，我们直接使用它。

接着，我们创建一个球体，它在 SpriteJS 的 3D 扩展中对应 Sphere 对象。

复制代码

```
const globe = new Sphere(program, {
  colors: '#333',
  widthSegments: 64,
  heightSegments: 32,
  radius: 1,
});


layer.append(globe);

```

我们给球体设置颜色、宽度、高度和半径这些默认的属性，然后将它添加到 layer 上，这样我们就能在画布上将这个球体显示出来了，效果如下所示。



现在，我们只在画布上显示了一个灰色的球体，它和我们要实现的地球还相差甚远。别着急，我们一步一步来。

2. 绘制地图

上节课，我们已经讲了绘制平面地图的方法，就是把表示地图的 JSON 数据利用墨卡托投影到平面上。接下来，我们也要先绘制一张平面地图，然后把它以纹理的方式添加到我们创建的 3D 球体上。

不过，与平面地图采用墨卡托投影不同，作为纹理的球面地图需要采用**等角方位投影** (Equirectangular Projection)。d3-geo 模块中同样支持这种投影方式，我们可以直接加载 d3-geo 模块，然后使用对应的代码来创建投影。

从 CDN 加载 d3-geo 模块需要加载以下两个 JS 文件：

复制代码

```
<script src="https://d3js.org/d3-array.v2.min.js"></script>
<script src="https://d3js.org/d3-geo.v2.min.js"></script>

```

然后，我们创建对应的投影：

复制代码

```
const mapWidth = 960;
const mapHeight = 480;
const mapScale = 4;

const projection = d3.geoEquirectangular();
projection.scale(projection.scale() * mapScale).translate([mapWidth * mapScale
```

这里，我们首先通过 d3.geoEquirectangular 方法来创建等角方位投影，再将它进行缩放。d3 的地图投影默认宽高为 960 * 480，我们将投影缩放为 4 倍，也就是将地图绘制为 3480 * 1920 大小。这样一来，它就能在大屏上显示得更清晰。

然后，我们通过 tanslate 将中心点调整到画布中心，因为 JSON 的地图数据的 0,0 点在画布正中心。仔细看我上面的代码，你会注意到我们在 Y 方向上多调整一个像素，这是因为原始数据坐标有一点偏差。

通过我刚才说的这些步骤，我们就创建好了投影，接下来就可以开始绘制地图了。我们从 topoJSON 数据加载地图。

复制代码

```
async function loadMap(src = topojsonData, {strokeColor, fillColor} = {}) {
  const data = await (await fetch(src)).json();
  const countries = topojson.feature(data, data.objects.countries);
  const canvas = new OffscreenCanvas(mapScale * mapWidth, mapScale * mapHeight
  const context = canvas.getContext('2d');
  context.imageSmoothingEnabled = false;
  return drawMap({context, countries, strokeColor, fillColor});
}

```

这里我们创建一个离屏 Canvas，用加载的数据来绘制地图到离屏 Canvas 上，对应的绘制地图的逻辑如下：

复制代码

```
function drawMap({
  context,
  countries,
  strokeColor = '#666',
  fillColor = '#000',
  strokeWidth = 1.5,
} = {}) {
  const path = d3.geoPath(projection).context(context);


  context.save();
  context.strokeStyle = strokeColor;
  context.lineWidth = strokeWidth;
  context.fillStyle = fillColor;
  context.beginPath();
  path(countries);
  context.fill();
  context.stroke();
  context.restore();


  return context.canvas;
```

这样，我们就完成了地图加载和绘制的逻辑。当然，我们现在还看不到地图，因为我们只是将它绘制到了一个离屏的 Canvas 对象上，并没有将这个对象显示出来。

3. 将地图作为纹理

要显示地图为 3D 地球，我们需要将刚刚绘制的地图作为纹理添加到之前绘制的球体上。之前我们绘制球体时，使用的是 SpriteJS 中默认的 shader，它是符合 Phong 光照模型的几何材质的。因为考虑到地球有特殊光照，我们现在自己实现一组自定义的 shader。

复制代码

```
const vertex = `
  precision highp float;
  precision highp int;


  attribute vec3 position;
  attribute vec3 normal;
  attribute vec4 color;
  attribute vec2 uv;


  uniform mat4 modelViewMatrix;
  uniform mat4 projectionMatrix;
  uniform mat3 normalMatrix;


  varying vec3 vNormal;
  varying vec2 vUv;
  varying vec4 vColor;


  uniform vec3 pointLightPosition; //点光源位置


  void main() {
    vNormal = normalize(normalMatrix * normal);


    vUv = uv;
    vColor = color;


    gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1.0);
  }
`;



const fragment = `
  precision highp float;
  precision highp int;
```

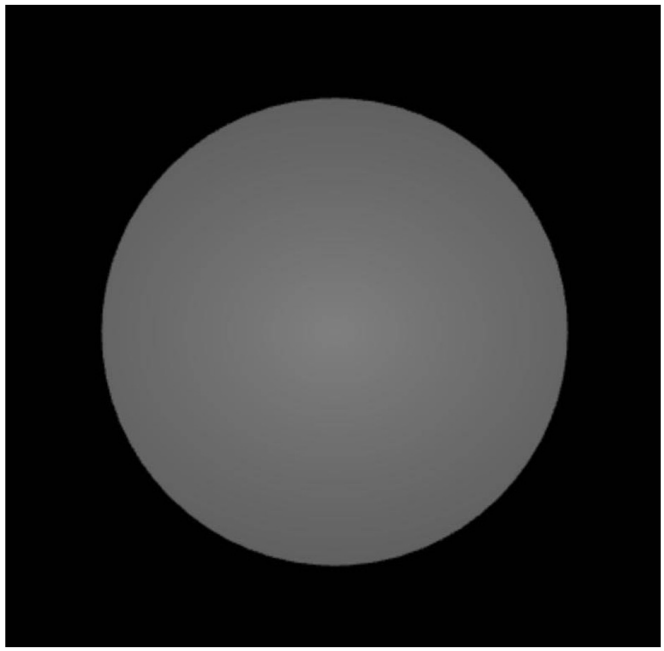
```
43
44
45    varying vec3 vNormal;
46    varying vec4 vColor;
47
48
49    uniform sampler2D tMap;
50    varying vec2 vUv;
51
52
53    uniform vec2 uResolution;
54
55
56    void main() {
57      vec4 color = vColor;
58      vec4 texColor = texture2D(tMap, vUv);
59      vec2 st = gl_FragCoord.xy / uResolution;
60
61
62      float alpha = texColor.a;
63      color.rgb = mix(color.rgb, texColor.rgb, alpha);
64      color.rgb = mix(texColor.rgb, color.rgb, clamp(color.a / max(0.0001, texCo
65      color.a = texColor.a + (1.0 - texColor.a) * color.a;
66
67
68      float d = distance(st, vec2(0.5));
69
70
71      gl_FragColor.rgb = color.rgb + 0.3 * pow((1.0 - d), 3.0);
72      gl_FragColor.a = c
```

我们用上面的 Shader 来创建 Program。这组 Shader 并不复杂，原理我们在视觉篇都已经解释过了。如果你觉得理解起来依然有困难，可以复习一下视觉篇的内容。接着，我们创建一个 Texture 对象，将它赋给 Program 对象，代码如下。

复制代码

```
1  const texture = layer.createTexture({});
2
3
4  const program = layer.createProgram({
5    vertex,
6    fragment,
7    texture,
8    cullFace: null,
9  });
10
```

现在，画布上就显示出了一个中心有些亮光的球体。



从中，我们还是看不出地球的样子。这是因为我们给的 texture 对象是一个空的纹理对象。接下来，我们只要执行 loadMap 方法，将地图加载出来，再添加给这个空的纹理对象，然后刷新画布就可以了。对应代码如下：

复制代码

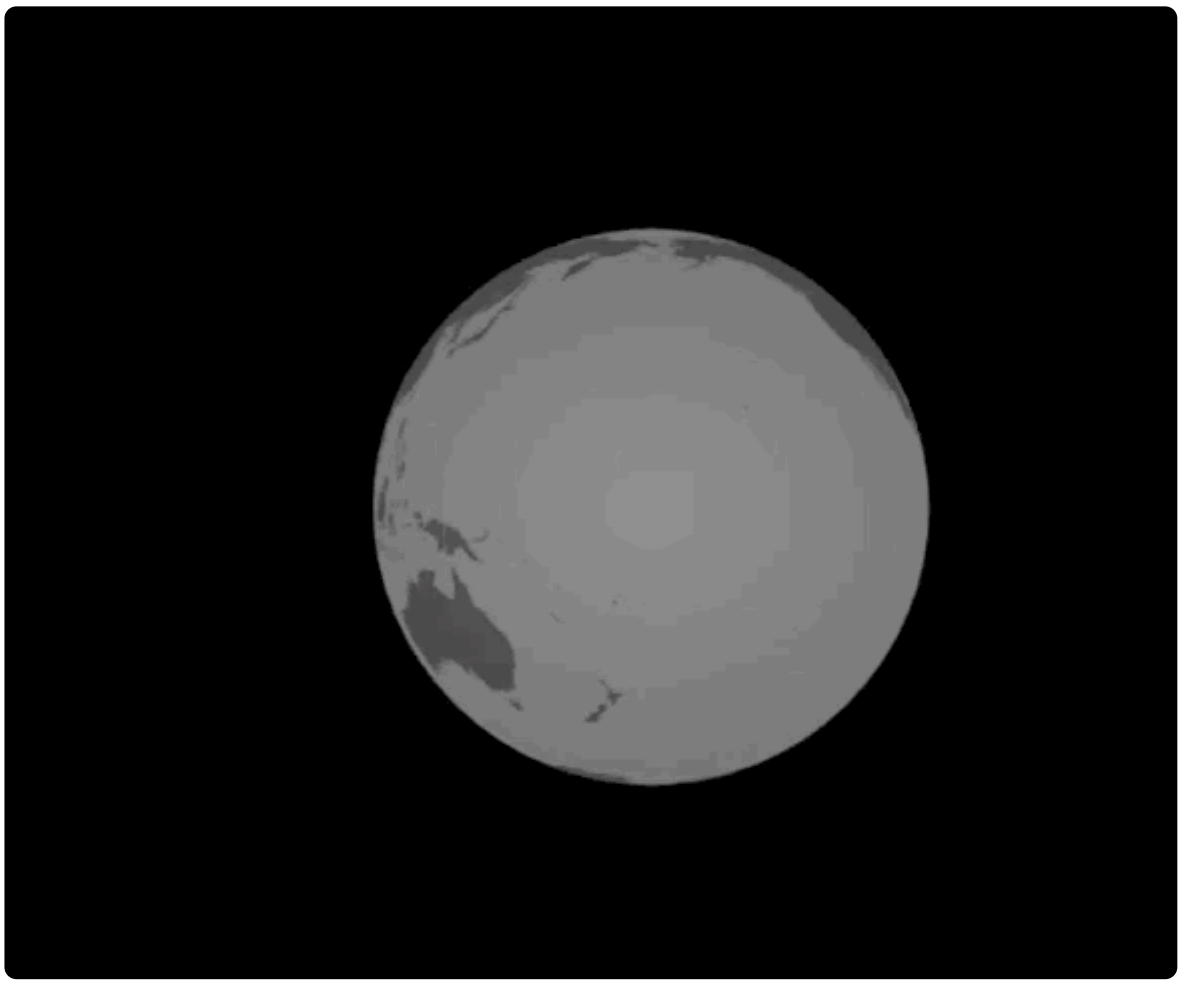
```
loadMap().then((map) => {
  texture.image = map;
  texture.needsUpdate = true;
  layer.forceUpdate();
});
```

最终，我们就显示出了地球的样子。

我们还可以给地球添加轨迹球控制，并让它自动旋转。在 SpriteJS 中非常简单，只需要一行代码即可完成。

复制代码

```
layer.setOrbit({autoRotate: true}); // 开启旋转控制
```



这样我们就得到一个自动旋转的地球效果了。

如何实现星空背景

不过，这个孤零零的地球悬浮在黑色背景的空间里，看起来不是很吸引人，所以我们可以给地球添加一些背景，比如星空，让它真正悬浮在群星闪耀的太空中。

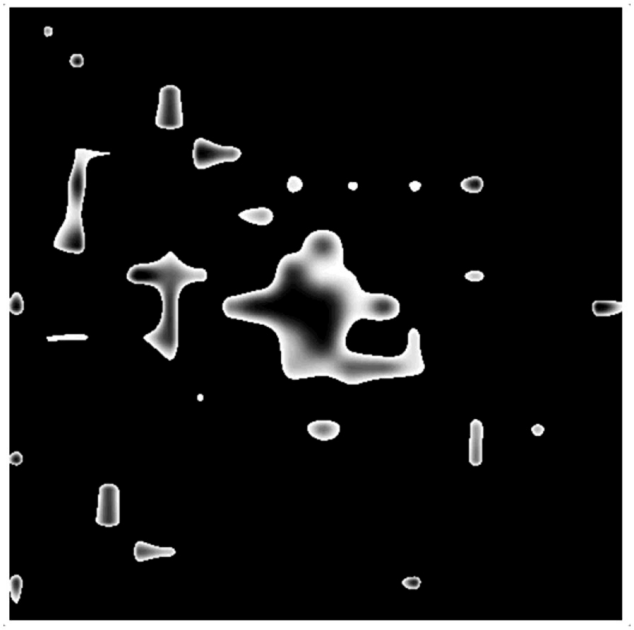
要实现星空的效果，第一步是要创建一个天空包围盒。天空包围盒也是一个球体（Sphere）对象，只不过它要比地球大很多，以此让摄像机处于整个球体内部。为了显示群星，天空包围盒有自己特殊的 Shader。我们来看一下：

复制代码

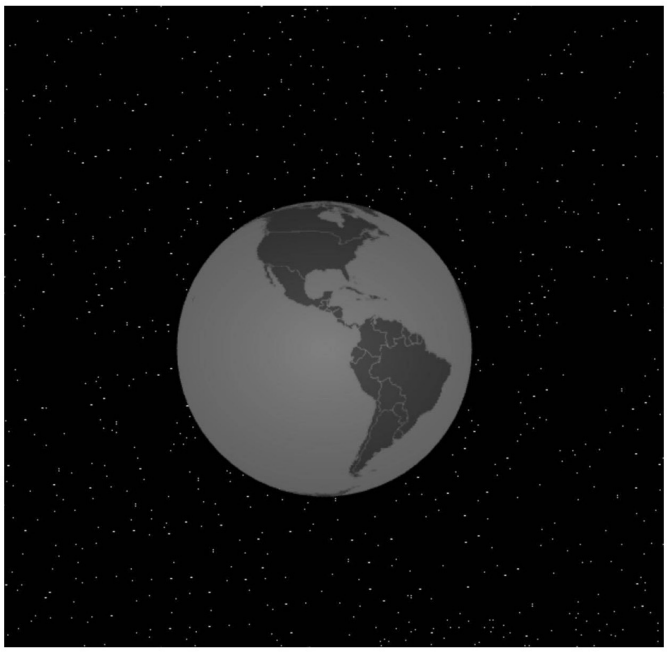
```
const skyVertex = `
  precision highp float;
  precision highp int;


  attribute vec3 position;
  attribute vec3 normal;
  attribute vec2 uv;

```

```
10
11    uniform mat3 normalMatrix;
12    uniform mat4 modelViewMatrix;
13    uniform mat4 projectionMatrix;
14
15
16    varying vec2 vUv;
17
18
19    void main() {
20      vUv = uv;
21      gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1.0);
22    }
23  `;
24
25
26  const skyFragment = `
27    precision highp float;
28    precision highp int;
29    varying vec2 vUv;
30
31
32    highp float random(vec2 co)
33    {
34      highp float a = 12.9898;
35      highp float b = 78.233;
36      highp float c = 43758.5453;
37      highp float dt= dot(co.xy ,vec2(a,b));
38      highp float sn= mod(dt,3.14);
39      return fract(sin(sn) * c);
40    }
41
42
43    // Value Noise by Inigo Quilez - iq/2013
44    // https://www.shadertoy.com/view/lsf3WH
45    highp float noise(vec2 st) {
46      vec2 i = floor(st);
47      vec2 f = fract(st);
48      vec2 u = f * f * (3.0 - 2.0 * f);
49      return mix( mix( random( i + vec2(0.0,0.0) ),
50                       random( i + vec2(1.0,0.0) ), u.x),
51                  mix( random( i + vec2(0.0,1.0) ),
52                       random( i + vec2(1.0,1.0) ), u.x), u.y);
53    }
54
55
56    void main() {
57      gl_FragColor.rgb = vec3(1.0);
58      gl_FragColor.a = step(0.93, noise(vUv * 6000.0));
```

上面的代码是天空包围盒的 Shader，实际上它是我们使用二维噪声的技巧来实现的。在第 16 节课中也有过类似的做法，当时我们是用它来模拟水滴滚过的效果。



但在这里，我们通过 step 函数和 vUv 的缩放，将它缩小之后，最终呈现出来星空效果。



对应的创建天空盒子的 JavaScript 代码如下：

复制代码

```
function createSky(layer, skyProgram) {
  skyProgram = skyProgram || layer.createProgram({
    vertex: skyVertex,
    fragment: skyFragment,
    transparent: true,
    cullFace: null,
  });
  const skyBox = new Sphere(skyProgram);
  skyBox.attributes.scale = 100;
  layer.append(skyBox);
  return skyBox;
}


createSky(layer);
```

不过，光看这些代码，你可能还不能完全明白，为什么二维噪声技巧就能实现星空效果。那也不要紧，完整的示例代码在GitHub 仓库中，最好的理解方式还是你自己试着手动修改一下 skyFragment 中的绘制参数，看看实现出来效果，你就能明白了。

要点总结

这节课，我们讲了实现 3D 地球可视化效果的方法，以及给 3D 地球添加天空背景的方法。

要实现 3D 地球效果，我们可以使用 SpriteJS 和它的 3D 扩展库。首先，我们绘制一个 3D 球体。然后，我们用 topoJSON 数据绘制地图，注意地图的投影方式必须选择等角方位投影。最后，我们把地图作为纹理添加到 3D 球体上，这样就绘制出了 3D 地球。

而要实现星空背景，我们需要创建一个天空盒子，它可以看成是一个放大很多倍的球体，包裹在地球的外面。具体的思路就是，我们创建一组特殊的 Shader，通过二维噪声来实现星空的效果。

说的这里，你可能会有一些疑问，我们为什么要用 topoJSON 数据来绘制地图，而不采用现成的等角方位投影的平面地图图片，直接用它来作为纹理，那样不是能够更快绘制出 3D 地球吗？的确，这样确实也能够更简单地绘制出 3D 地球，但这么做也有代价，就是我们没有地图数据就不能进一步实现交互效果了，比如说，点击某个地理区域实现当前国家地区的高亮效果了。

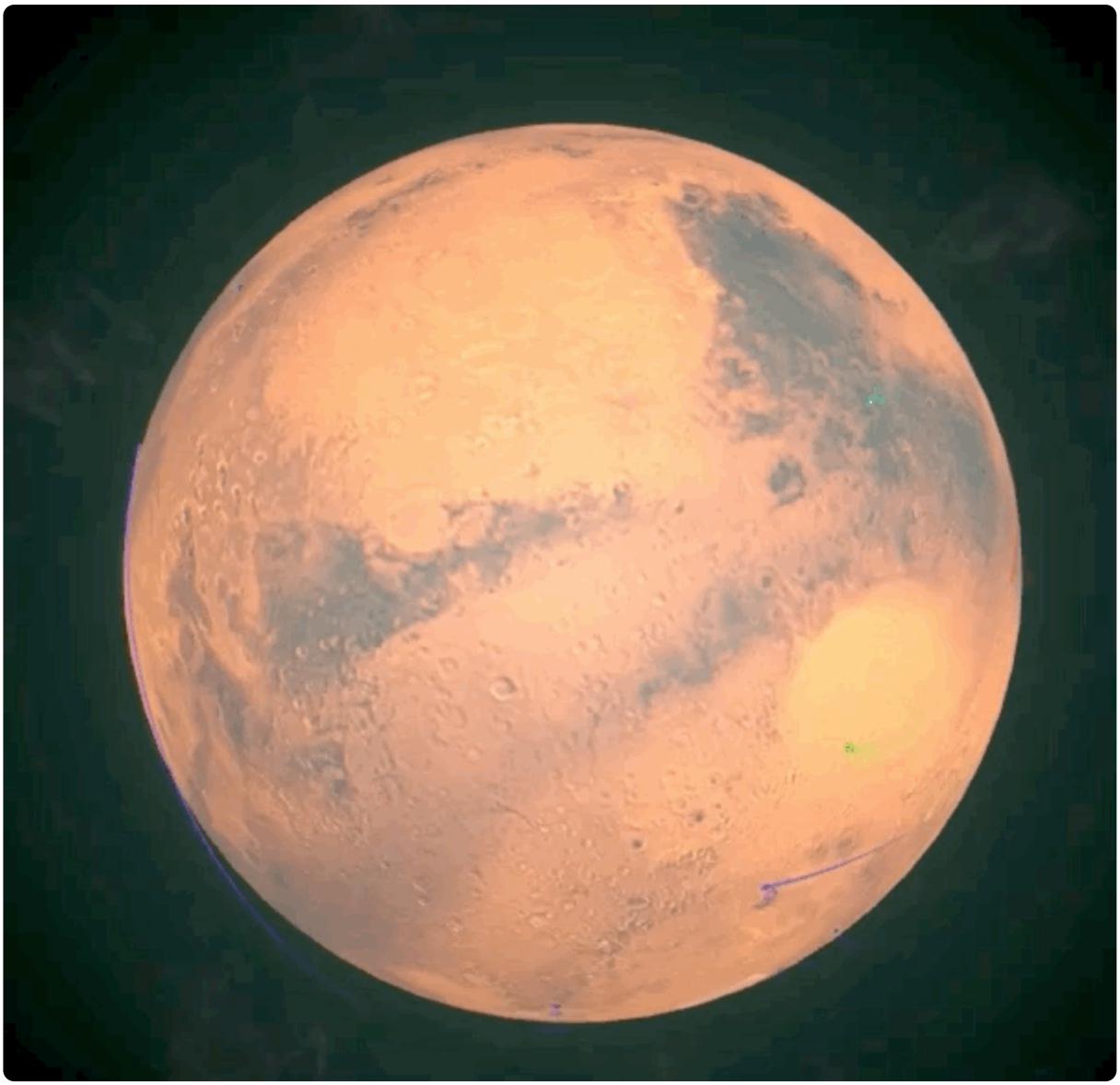
那在下节课，我们就会进一步讲解怎么在 3D 地球上添加交互效果，以及根据地理位置来放置各种记号。你的疑问也都会一一解开。

小试牛刀

我们说，如果不考虑交互，可以直接使用更简单的等角方位投影地图作为纹理来直接绘制 3D 地球。你能试着在网上搜索类似的纹理图片来实现 3D 地球效果吗？

另外，你可以找类似的其他行星的图片，比如火星、木星图片来实现 3D 火星、木星的效果吗？

最后，你也可以想想，除了星空背景，如果我们还想在地球外部实现一层淡绿色的光晕，又该怎么做呢（提示：你可以使用距离场和颜色插值来实现）？



今天的 3D 地球可视化实战就到这里了。欢迎把你实现的效果分享到留言区，我们一起交流。也欢迎把这节课转发出去，我们下节课见！

源码

课程完整示例代码详

提建议

上一篇 39 | 实战（三）：如何实现地理信息的可视化?

下一篇 41 | 实战（五）：如何实现3D地球可视化（下）？

精选留言

写留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。