加餐二 | SpriteJS：我是如何设计一个可视化图形渲染引擎的？

2020-09-02 月影

跟月影学可视化 进入课程 >

讲述：月影
时长 14:06 大小 12.93M

你好，我是月影。

今天，我们来聊一个相对轻松的话题，它不会有太多的代码，也不会有什么必须要掌握的理论知识。不过这个话题对你理解可视化，了解渲染引擎也是有帮助的。因为我今天要聊的话题是 SpriteJS，这个我亲自设计和实现的图形渲染引擎的版本迭代和演进。

SpriteJS 是从 2017 年下半年开始设计的，到今天已经快三年了，它的大版本也从 1.0 升级到了 3.0。那么它为什么会被设计出来？它有什么特点？1.0、2.0、3.0 版本之间有什么区别，未来会不会有 4.0 甚至 5.0？别着急，听我一一道来。

SpriteJS v1.x （2017 年~2018 年）

我们把时间调回到 2017 年下半年，当时我还在 360 奇舞团。奇舞团是 360 技术中台的前端团队，主要负责 Web 开发，包括 PC 端和移动端的产品的前端开发，比较少涉及可视化的内容。不过，虽然团队以支持传统 Web 开发为主，但是也支持过一部分可视化项目，比如一些 toB 系统的后台图表展现。那个时候，我们团队正要开始尝试探索可视化的方向。

如果你读过专栏的预习篇，你应该知道，要实现可视化图表，我们用图表库或者数据驱动框架都能够实现，前者使用起来简单，而后者更加灵活。当时，奇舞团的小伙伴更多是使用数据驱动框架D3.js来实现可视化图表的。

对 D3.js 来说，D3-selection是其核心子模块之一，它可以用来操作 DOM 树，返回选中的 DOM 元素集合。这个操作非常有用，因为它让我们可以像使用 jQuery 那样，快速遍历 DOM 元素，并且它通过 data 映射将数据与 DOM 元素对应起来。这样，我们用很简单的代码就能实现想要的可视化效果了。

比如，我们通过 `d3.select('body').selectAll('div').dataset(data).enter().append('div')`，把对应的 div 元素根据数据的数量添加到页面上的 body 元素下，然后，我们直接通过.style 来操作对应添加的 div 元素，修改它的样式，就能轻松绘制出一个简单的柱状图效果了。

复制代码

```
const dataset = [125, 121, 127, 193, 309];
const colors = ['#fe645b', '#feb050', '#c2af87', '#81b848', '#55abf8'];

const chart = d3.select('body')
  .selectAll('div')
  .data(dataset)
  .enter()
  .append('div')
  .style('left', '450px')
  .style('top', (d, i) => {
    return `${200 + i * 45}px`;
  })
  .style('width', d => `${d}px`)
  .style('height', '40px')
  .style('background', (d, i) => colors[i]);
```

这是一个非常快速且方便的绘图方式，但它也有局限性。D3-selection 只能操作具有 DOM 结构的图形系统，也就是 HTML 和 SVG。而对于 Canvas 和 WebGL，我们就没有办法像上面一样，直接遍历元素并且将数据和元素结构对应起来。

正因为 D3-selection 操作 DOM 使用起来特别方便，所以常见的 D3 例子都是用 HTML 或者 SVG 来写的，很少使用 Canvas 和 WebGL，即便后两者的性能要大大优于 HTML 和 SVG。因此，当时实现 SpriteJS 1.0 的初衷非常简单，那就是我希望让团队的同学既能使用熟悉的 D3.js 来支持可视化图表的展现，又可以使用 Canvas 来代替默认的 SVG 进行渲染，从而达到更好的性能。

所以，**SpriteJS 1.0 实现了整个 DOM 底层的 API，我们可以像操作浏览器原生的 DOM 一样来操作 SpriteJS 元素，而我们最终渲染出的图形是调用底层 Canvas 的 API 绘制到画布上的。**这样一来，SpriteJS 和 HTML 或者 SVG，就都可以用 D3-selection 来操作了，在使用上它们没有特别大的差别，但 SpriteJS 的最终渲染还是通过 Canvas 绘制的，性能相比其他两种有了较大的提升。

比如说，我用 D3.js 配合 SpriteJS 实现的柱状图代码，与使用 HTML 绘制的代码区别不大，但是由于是绘制在 Canvas 上，性能会提升很多。

复制代码

```
1    const {Scene, Sprite} = spritejs;
```

```
const container = document.getElementById('container');
const scene = new Scene({
  container,
  width: 800,
  height: 800,
});

const dataset = [125, 121, 127, 193, 309];
const colors = ['#fe645b', '#feb050', '#c2af87', '#81b848', '#55abf8'];

const fglayer = scene.layer('fglayer');
const chart = d3.select(fglayer)
  .selectAll('sprite')
  .data(dataset)
  .enter()
  .append('sprite')
  .attr('x', 450)
  .attr('y', (d, i) => {
    return 200 + i * 45;
  })
  .attr('width', d => d)
  .attr('height', 40)
  .attr('bgcolor', (d, i) => colors[i]);
```

除了解决 API 的问题，以及让 D3-selection 可以使用之外，为了让使用方式尽可能接近于原生的 DOM，我还让 SpriteJS 1.0 实现了这 4 个特性，分别是标准的 DOM 元素盒模型、标准的 DOM 事件、Web Animation API （动画）以及缓存策略。

盒模型、DOM 事件和 Web Animation API ，我想你作为前端工程师肯定都知道，所以我多说一下缓存策略。还记得在性能篇里我们说过，要提升 Canvas 的渲染性能，就要尽量减少绘图指令的数量和执行时间，比较有效的方式是，我们可以将绘制的图形用离屏 Canvas 缓存下来。这样，在下次绘制的时候，我们就可以将缓存未失效的元素从缓存中用 drawImage 的方式直接绘制出来，而不用重新执行绘制元素的绘图指令，也就大大提升了性能。

因此，**在 SpriteJS 1.0 中，我实现了一套自动的缓存策略，它会根据代码运行判断是否对一个元素启用缓存，如果是，就尽可能地启用缓存，让渲染性能达到比较好的水平。**

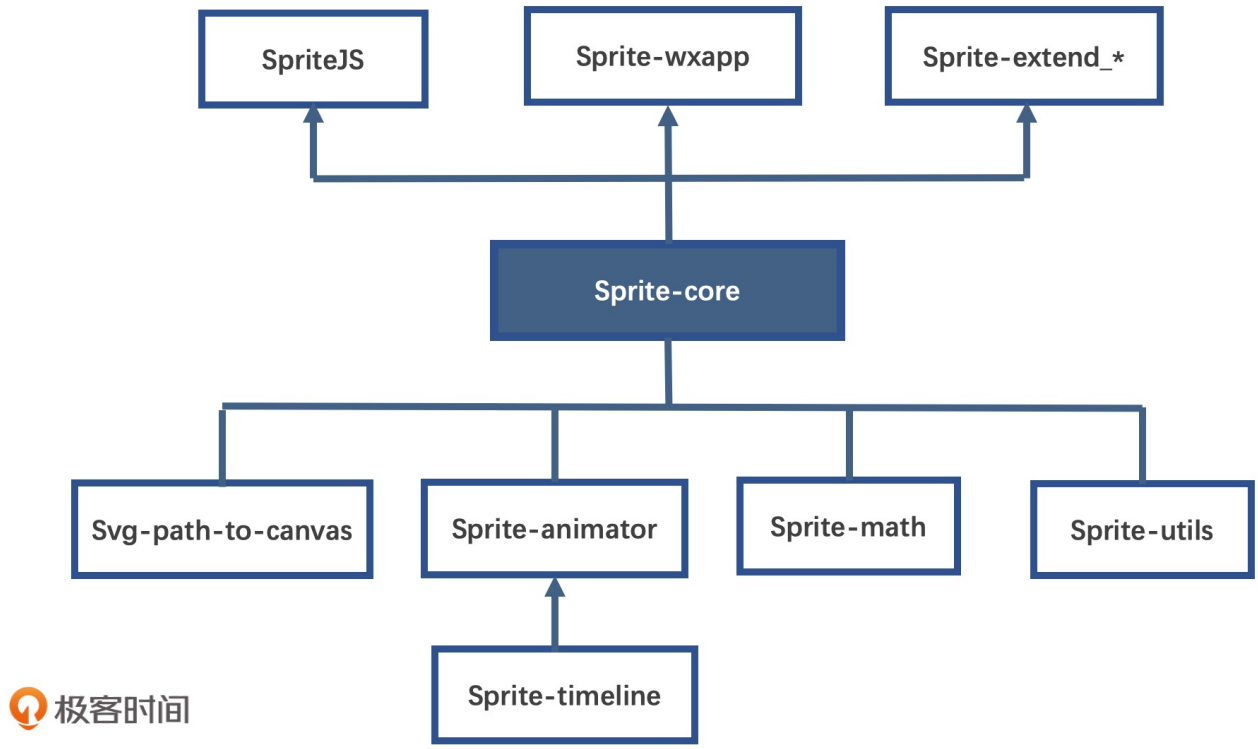
SpriteJS 1.0 实现的这些特性，基本上满足了我们当时的需要，让我们团队可以用 D3.js 配合 SpriteJS 来实现各种可视化图表项目需求，而且使用上非常接近于操作原生的 DOM，非常容易上手。

SpriteJS v2.x （2018 年~2019 年）

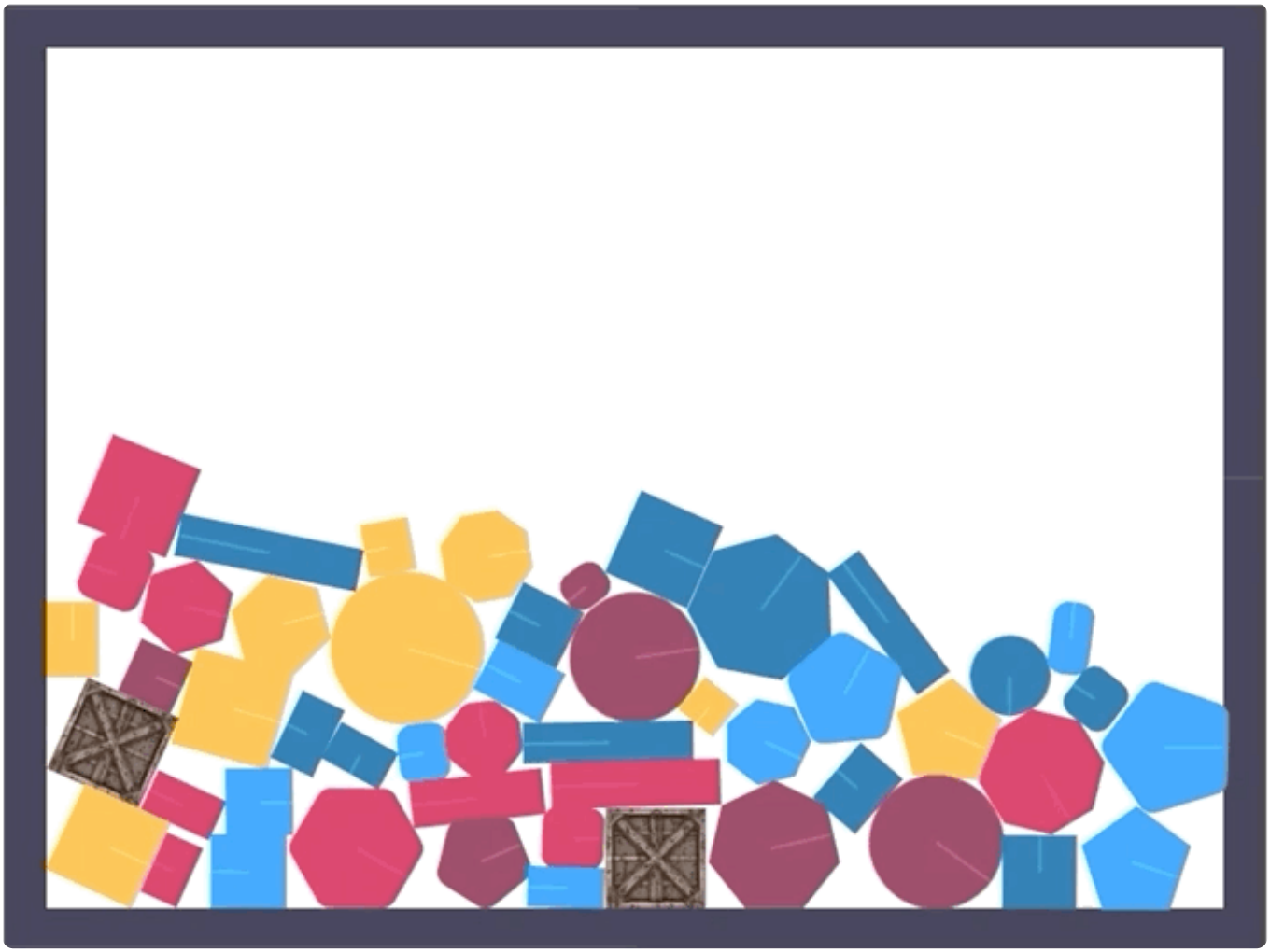
到了 2018 年底，我开始思考 SpriteJS 的下一个版本。当时我们解决了在 PC 和移动 Web 上绘制可视化图表的诉求，不过外部的使用者和我们自己，在一些使用场景中，逐渐开始有一些跨平台的需求，比如在服务端渲染，或者在小程序中渲染。

因此，我开始重构代码，将绘图系统分层设计，实现了渲染的适配层。在适配层中，所有的绘图能力都由 Canvas 底层 API 提供，与浏览器 DOM 和其他的 API 无关。这样，SpriteJS 就能够运行在任何提供了 Canvas 运行时环境的系统中，而不一定是浏览器。

重构后的代码能够通过node-canvas运行在 Node.js 环境中，所以我们就能够使用服务端渲染来实现一些特殊的可视化项目。比如，我们曾经有一个项目要处理大量的历史数据，大概有几十万到上百万条记录，如果在前端分别绘制它们，性能一定会有问题。所以，我们将它们通过服务端绘制并缓存好之后，以图像的方式发送给前端，这样就大大提升了性能。此外，我们还通过在适配层上提供不同的封装，让 SpriteJS 2.0 支持了小程序环境，也能够运行在微信小程序中。



上图是 SpriteJS 2.0 的主体架构，它的底层由一些通用模块组成，Sprite-core 是适配层，SpriteJS 是支持浏览器和 Node.js 的运行时，Sprite-wxapp 是小程序运行时，Sprite-extend-* 是一些外部扩展。我们通过外部扩展实现了粒子系统和物理引擎，以及对主流响应式框架的支持，让 SpriteJS 2.0 可以直接支持vue和react。



SpriteJS 2.0通过扩展实现物理引擎

除此以外，SpriteJS 2.0 还支持了文字排版和布局系统。其中，文字排版支持了多行文本自动换行，实现了几乎所有 CSS3 支持的文字排版属性，布局系统则支持了完整的弹性布局（Flex layout)。这两个特性被很多用户喜爱。

可以说，我们对 SpriteJS 2.0 做了加法，让它在 1.0 的基础上增加了许多强大且有用的特性。到了 2019 年底，我又开始思考实现 SpriteJS 3.0。这次我打算对特性做一些取舍，将许多特性从 SpriteJS 3.0 中去掉，甚至包括深受使用者喜爱的文字排版和布局系统。这又是为什么呢?

这是因为 SpriteJS 2.0 虽好，但是它也有一些明显的缺点：

1. 只支持 Canvas2D，尽管有缓存策略，性能仍然不足；
2. 多平台适配采用不同的分支，维护起来比较麻烦；
3. 支持了许多非核心功能，如文字排版、布局，使得 JavaScript 文件太大；
4. 不支持 3D 绘图。

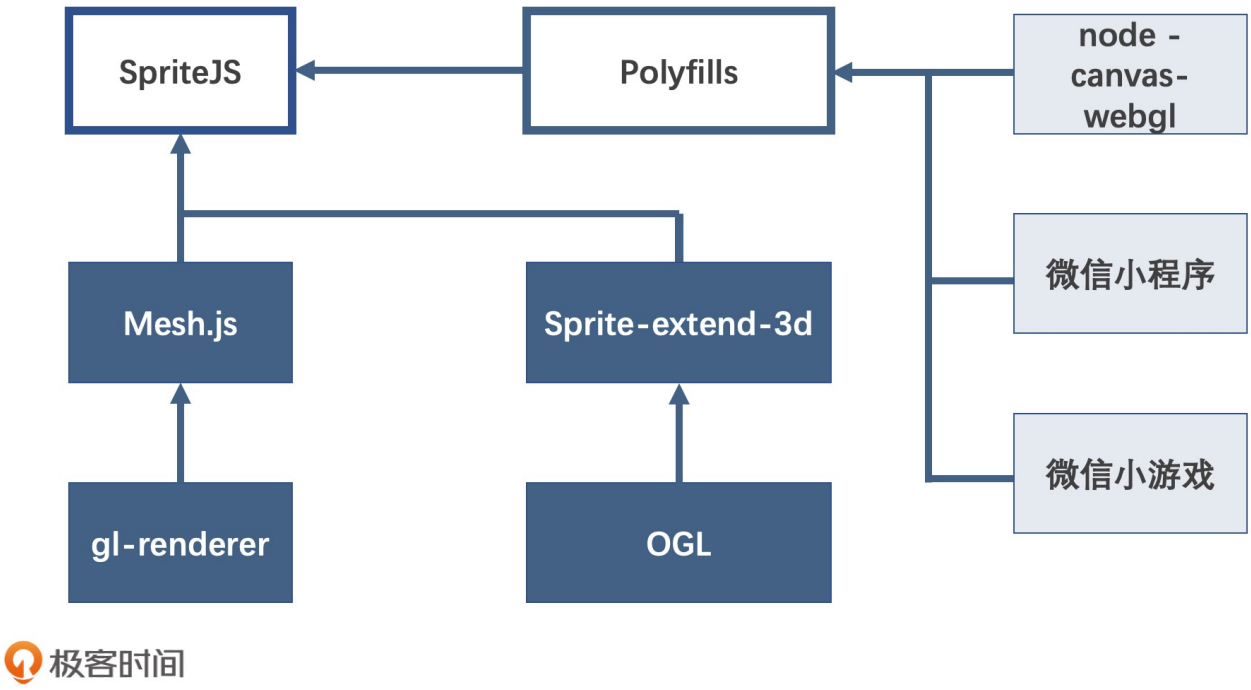
SpriteJS v3.x （2019 年~2020 年）

在 SpriteJS 3.0 中，我舍弃了非核心功能，将 SpriteJS 定位为纯粹的图形渲染引擎， 核心目标是追求极致的性能。

在适配层上，SpriteJS 3.0 完全舍弃了 2.0 设计里面较重的 sprite-core，采用了更轻量级的图形库mesh.js作为 2D 适配层，mesh.js 以 gl-renderer 作为 webgl 渲染底层库，结合 Canvas2D 的 polyfill 做到了优雅降级。当运行环境支持 WebGL2.0 时，SpriteJS 3.0 默认采用 WebGL2.0 渲染，否则降级为 WebGL1.0，如果也不支持 WebGL1.0，再最终降级为 Canvas2D。

在 3D 适配层方面，SpriteJS 3.0 采用了OGL库。这样一来，SpriteJS 3.0 就完全支持 WebGL 渲染，能够绘制 2D 和 3D 图形了。

SpriteJS 3.0 继承了 SpriteJS 2.0 的跨平台性，但是不再需要使用分支来适配多平台，而是采用了更轻量级的 polyfill 设计，同时支持服务端渲染、Web 浏览器渲染和微信小程序渲染，理论上讲还可以移植到其他支持 WebGL 或 Canvas2D 的运行环境中去。

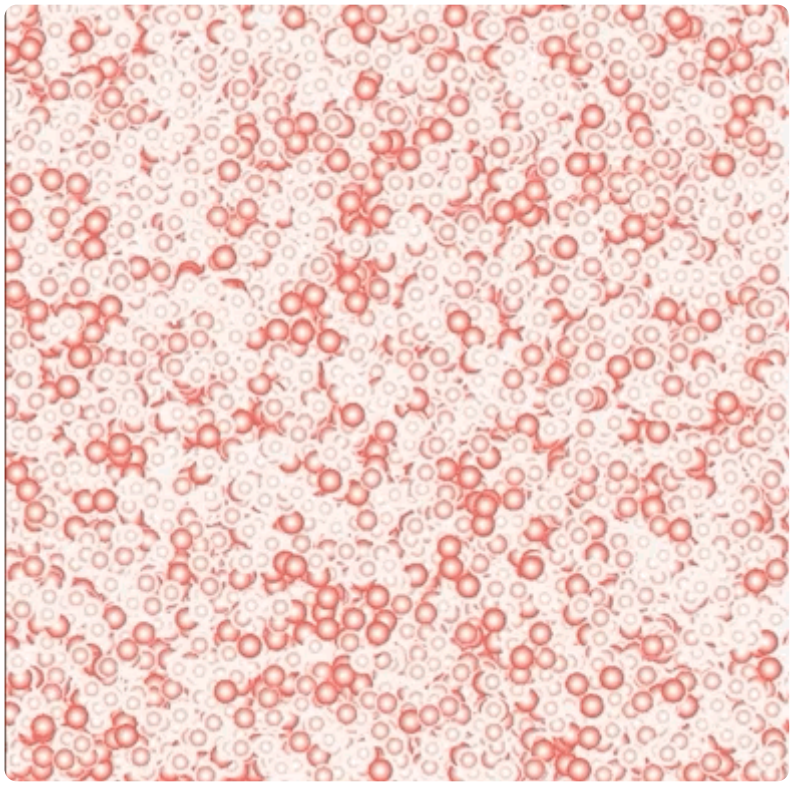


SpriteJS 3.0 结构

与 SpriteJS 1.0 和 SpriteJS 2.0 采用缓存机制优化性能不同，SpriteJS 3.0 默认采用 WebGL 渲染，因此使用了批量渲染的优化策略，我们在性能篇中讲过这种策略，在绘制大

量几何图形时，它能够显著提升 WebGL 渲染的性能。

由于发挥了 GPU 并行计算的能力，在大批量图形绘制的性能上，SpriteJS 3.0 的性能大约是 SpriteJS 2.0 的 100 倍。此外，SpriteJS 3.0 支持了多线程渲染，可避免 UI 阻塞，从而进一步提升性能。



SpriteJS 3.0 绘制5万个地理信息点，60fps帧率

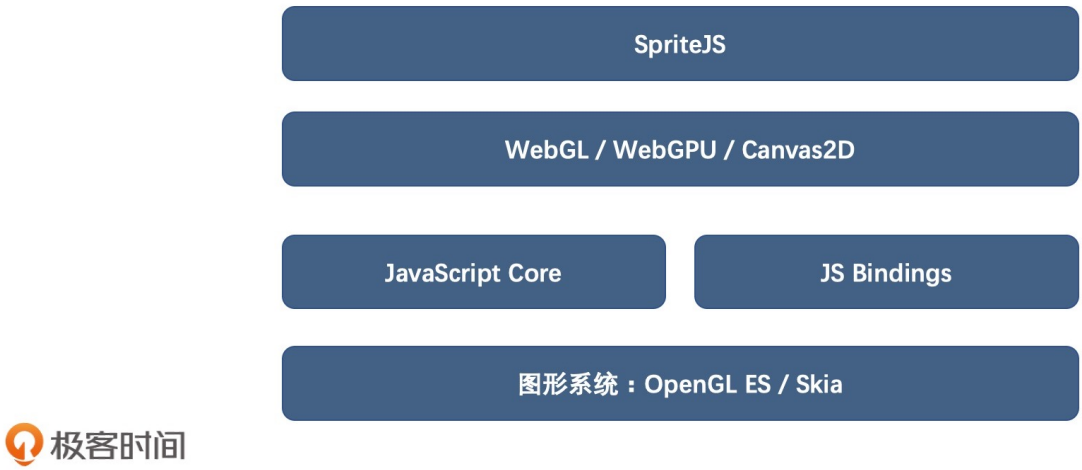
总之，SpriteJS 3.0 随着性能的优化，已经成为一个纯粹的可视化渲染引擎了，但在我看来它仍然有些问题：

1. 性能优化得不够极致，数据压缩和批量渲染没有做到最好；
2. JS 的矩阵运算还是不够快，计算性能有提升空间；
3. 因为考虑到兼容性的问题，所以我采用了 Canvas2D 的降级，这让 JavaScript 包仍然有些大；
4. 3D 能力不够强，与 ThreeJS 等主流 3D 引擎仍有差距。

SpriteJS 的未来版本（2020 年~2021 年）

今年下半年，我开始设计 SpriteJS 4.0。这一次，我打算把它打造成一个更纯粹的图形系统，让它可以做到真正跨平台，完全不依赖于 Web 浏览器。

下面是 SpriteJS 4.0 的结构图，它的底层将采用 OpenGL ES 和 Skia 来渲染 3D 和 2D 图形，中间层使用 JavaScript Core 和 JS Bindings 技术，将底层 Api 通过 JavaScript 导出，然后在上层适配层实现 WebGL、WebGPU 和 Canvas2D 的 API，最上层实现 SpriteJS 的 API。



SpriteJS 4.0 体系结构

根据这个设计，SpriteJS 4.0 将对浏览器完全没有依赖，同时依然可以通过 Web Assembly 方式运行在浏览器上。这样 SpriteJS 4.0 会成为真正跨平台的图形系统，可以以非常小的包集成到其他系统和原生 App 中，并且达到原生应用的性能。

在这一版，我还会全面优化 SpriteJS 的内存管理、矩阵运算和多线程机制，力求渲染性能再上一个台阶，最终能够完全超越现在市面上的任何主流的图形系统。

要点总结

在 SpriteJS 1.0 中，我们追求的是和 DOM 一致的 API，能够使用 D3.js 结合 SpriteJS 来绘制可视化图表到 Canvas，从而提升性能。到了 SpriteJS 2.0，我们追求跨平台能力和一些强大的功能扩展，比如文字排版和布局系统。而到了 SpriteJS 3.0，我们决定回归到渲染引擎本质，追求极致的性能发挥 GPU 的能力，并支持 3D 渲染。再到今年的 SpriteJS 4.0，我打算把它打造成更纯粹的图形系统，让它的渲染能力和性能最终能够超越目前市面上的主流图形系统。

总的来说，在 SpriteJS 1.0 到 4.0 的设计发展过程中，包含了我对整个图形系统架构的思考和取舍。我希望通过我今天的分享，能够帮助你理解图形系统和渲染引擎的设计，也期待在你设计其他系统和平台的时候，它们能给你启发。

课后思考

最后，请你试着回想你曾经接触过的可视化项目，如果用 SpriteJS 来实现它们会不会有更好的效果呢？欢迎把你的思考和答案写在留言区，我们一起讨论。

看了我给 SpriteJS 未来版本定下的目标，你有没有心动呢？SpriteJS 是一个开源项目，如果你学完这门课，也想参与进 SpriteJS 的开发，那我非常欢迎你成为一名 SpriteJS 开发者，为我们提交 PR、贡献代码。

好了，今天的内容就到这里，我们下节课见！

推荐阅读

1. D3.js
2. SpriteJS
3. Mesh.js

提建议

更多课程推荐

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 加餐一 | 作为一名程序员，数学到底要多好?

下一篇 用户故事 | 非前端开发，我为什么要学可视化?

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。