

一种高效的 SVG 动画自适应渲染算法

周宇轩¹, 刘 丹¹, 乐 俊²

(1 电子科技大学 电子科学技术研究院, 四川 成都 611731; 2 西南电子电信技术研究所, 四川 成都 611731)

摘 要: 针对不同浏览器动画渲染性能差距悬殊的问题设计了一种高效率的自适应的 SVG 动画渲染算法, 该算法通过将动画-算子由多对多归并为一对多形式, 缩减浏览器底层调度次数, 削减系统资源开销. 引入帧率自适应算法, 为没有针对动画渲染进行优化的浏览器提供兼容性实现, 对有优化的浏览器进行渲染效能的进一步提升. 此算法保证了动画渲染性能在不同浏览器上都保持较高的效率以及稳定性, 实验证明, 此渲染算法无论有无针对动画渲染的底层优化, 都能很好地平滑性能抖动, 根据浏览器所能支持的条件达到当前情况最高的效率, 支撑大数量的 SVG 动画渲染.

关键词: 动画渲染; JavaScript; 算法优化; 自适应

中图分类号: TP317

文献标识码: A

文章编号: 1000-7180(2017)08-0093-06

An Efficient Adaptive Rendering Algorithm for SVG Animation

ZHOU Yu-xuan¹, LIU Dan¹, YUE Jun²

(1 Research Institute of Electronic Science and Technology, University of Electronic Science and Technology of China, Chengdu 611731, China; 2 Research Institute of Southwest Electronic Telecommunication Technology, Chengdu 611731, China)

Abstract: In view of the huge gap of the performance for animation rendering between the different browsers, designed a high efficient algorithm for SVG animation rendering, which is to merge the animation and operator from multi to multi to one to multi, and reduced the count of the invoking of the basement of browser, the algorithm, the consumption of system resource. Besides, the algorithm introduced the frame adaptation algorithm, which improved the compatibility for the browsers which is not optimized for animation rendering, and enhanced the performance of browsers which has optimization. so, the algorithm ensured the high efficient and stability between different browsers. Experiments show that the rendering algorithm, whether or not on the browsers which has the optimization of the animation rendering, can smooth the performance jitter, reached the idea efficiency the browsers can ever reached in the current situation, and support the large amount of SVG animation rendering.

Key words: animation rendering; Javascript; algorithm optimization; self-adaption

1 引言

随着当今硬件性能的提升, 浏览器慢慢从简单的渲染表单的应用演变成了一个可以处理视频流, 渲染动画甚至运行整个 web 应用的载体. 人们从仅仅拿浏览器查看文章到通过浏览器观看视频, 网络购物甚至玩网页游戏等等, 浏览器在互联网的浪潮中担当了越来越重要的角色. 与此同时, 浏览器领域

也衍生出了多种 SVG 渲染技术.

目前, 基于 SVG 的动画渲染技术相对来说还有比较大的局限性. 以现有的 SVG 动画渲染技术为例, 帧率方面, 缺乏动态的可自适应调整机制, 当 SVG 动画数量线性增加或者减少时, 帧率往往是由 30 帧到 60 帧或者 60 帧到 30 帧的大幅度陡增或者陡降, 用户体验上欠佳, 不同的浏览器实现方式也不一致, 有的甚至根本没有帧率自适应功能, 在 SVG 动画数量增多或者减少时不能动态调整帧率, 使得

CPU 占用率非常不稳定;此外,由于现阶段 SVG 动画渲染技术标准不统一,各个厂家的浏览器底层实现不完全一致,目前一些公司的浏览器包括谷歌 Chrome 浏览器、Opera、Firefox、火狐等采用的是渲染帧率与浏览器帧率同步的解决方案,开源框架方面提供的解决方案则仍然采用的较为老旧的定时器解决方案,这样就会带来一个问题:一个渲染动画在不同浏览器上或者同一浏览器使用不同框架在渲染效率上差距悬殊。

基于此,设计了一种可根据 SVG 动画执行耗时动态调整帧率且自适应浏览器的 SVG 动画渲染算法,能够对 CPU 占用率削峰填谷,对于没有帧率同步机制的浏览器或者框架也能提供不错的渲染效果. 保证应用在不同标准的浏览器下运行能有较好的稳定性. 优化后的算法与现有实现方式通过测试数据进行对比,改进明显。

2 现有的 SVG 动画渲染技术

一般有三种方式进行实现: setTimeout 函数、 setInterval 函数、使用 css animation 关键字以及 requestAnimationFrame (以后简称 rAF) 底层 API [1]。

2.1 定时器技术

定时器技术是由 setTimeout 以及 setInterval 函数实现的,在指定毫秒数后调用回调函数. 然而,它的执行机制并不是这么简单. 实际上, setTimeout 以及 setInterval 的作用是在指定的毫秒数之后,在得到机会时,将 callback 放入 Event Loop Queue [2]。定时器原理图如图 1 所示。

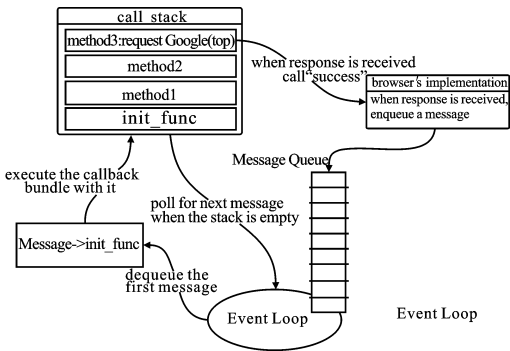


图 1 定时器原理图

当脚本代码全部执行完且栈中的方法都执行完毕弹出后即栈为空时, Event Loop Queue 就会弹出一个事件并执行事件绑定的函数. 以 setTimeout 为例子:

```
console.log("A");
setTimeout(() => {
    console.log("B");
}, 100);
console.log("C");
```

这段代码打印顺序为 A、C、B, 因为打印 B 的语句被压入了 Event Loop Queue, 而 Event Loop Queue 弹出事件并执行回调当且仅当代码全部执行完毕且栈帧为空. setInterval 原理与 setTimeout 原理一致, 只不过 setTimeout 只会在 n 毫秒后会将回调函数压入队列中一次, 而 setInterval 会重复压入队列 [3]。

如何运用这两种机制渲染 SVG 动画也就显而易见了, 通过不断执行的回调函数来改变 SVG 的坐标以及 CSS 属性就能渲染出 SVG 动画了. 然而, 这种方式有一种明显的缺点, 那就是如果同时渲染多个 SVG 对象的话, 回调任务数会呈线性递增, Event Loop 异步机制由操作系统底层作为支持, 涉及到线程的上下文切换, 当回调任务超过一定数目时, 消耗的资源将大大增加 [4]。

2.2 window.requestAnimationFrame

window.requestAnimationFrame 方法会通知浏览器渲染动画并请求浏览器执行特定方法更新动画, 然后继续下一帧的渲染, 其中入参传入一个回调函数, 在重绘下一帧时执行. 具体代码示例如下:

```
var start = null;
var element = document.getElementById("animate");
element.style.position = 'absolute';
function step(timestamp) {
    if (!start) start = timestamp;
    var progress = timestamp - start;
    element.style.left = Math.min(progress/10, 200)
    + "px";
    if (progress < 2000) window.requestAnimationFrame(step);
}
window.requestAnimationFrame(step);
```

此代码的功能为把 id 为 animate 的元素向右移动到 200px 的位置. 可以看到 step 函数每执行一次都会将自身重新设置到 requestAnimationFrame 函数中去, 从而实现不断执行。

rAF 是针对高性能动画渲染进行专门优化的 API, 目前已有多个浏览器支持, 其中包括 IE10 及以上版本、IE Edge、Chrome、Safari、Opera 以及 Firefox 等, 在移动端, IE mobile 10 以上版本以及

IOS6 以上的 Safari 也支持,唯一比较遗憾的是目前安卓上的原生浏览器并不支持 rAF,不过支持 rAF 应该是大势所趋了,安卓版本的 Chrome 16+也是支持 rAF 的^[5].

rAF 比起 setTimeout、setInterval 的优势主要有两点:

- rAF 会将所有对文档对象模型进行的操作进行集中,并在一次回流或者重绘动作中完成所有操作.与此同时,重绘或者回流的频率与浏览器的刷新频率保持一致,一般这个频率保持在 60 帧左右.
- 当元素为不可见或者隐藏的时候以及焦点离开浏览器的时候,rAF 不会对其回流或者重绘,这样能够大量减少 CPU、GPU 以及内存的使用量.

rAF 是一个全局函数,调用后,它会和浏览器的刷新频率进行同步,rAF 接收一个回调函数为入参,当浏览器即将进行刷新时,rAF 会执行这个回调函数,同时,还会将调用回调函数的时间传入回调函数作为入参.由于 rAF 是一次性调用,所以若想达到动画效果,则必须连续不断地调用.调用了 rAF 函数后会返回一个资源标识符,可以将标识符作为参数传入 cancelAnimationFrame 函数来取消 rAF 的回调.实际上,rAF 可以理解为性能优化版本的 setTimeout,其专门为动画渲染打造,不同的是 rAF 不是自己指定回调函数运行的时间,而是跟着浏览器内建的刷新频率来执行回调,这当然就能达到浏览器所能实现动画的最佳效果了^[6].

目前,rAF 还未形成行业标准,各个浏览器实现方式不尽相同,都为各自的私有实现.而对于没有实现 rAF 函数的浏览器,我们退化使用 setTimeout,因为两者在使用方式上几乎相同,所以两者在兼容上并不困难^[7].对于支持 rAF 的浏览器,我们使用 rAF,而不支持我们优雅降级使用传统的 setTimeout.把它们封装一下,就能得到一个统一兼容各大浏览器的 API 了.不过这样缺点也很明显,同一段代码在有无 rAF 支持的浏览器上性能差距会非常大,从而间接造成了应用的不稳定性.同时,60 帧的固定帧率让开发人员缺少了动画的可定制性.

2.3 CSS animation 关键字

实际上 CSS animation 可以称之为原生动画,是既定的,故在可定制方面就没有以上两种方式强,故一般只能实现一些简单的动画.此外,CSS animation 除了小规模动画比较高效外,复杂动画的渲染,性能有较大缺陷^[8].

2.4 总结

三种动画无论在性能上还是可用性上都有比较明显的优缺点,如表 1 所示.

表 1 三种动画的比较

动画类型	优点	缺点
定时器	编写简单,易于理解	动画数目较多时存在性能问题
rAF	支持批渲染,相对定时器效率有明显提升	部分浏览器不支持,会退化到定时器渲染动画,同一应用有潜在的负载抖动.帧率不可定制
CSS animation 关键字	简单,效率高	无法实现较为复杂的动画

3 现有 SVG 动画渲染技术在实际应用方面的缺陷

在多 SVG 动画渲染中,以由于每个 SVG 元素有不同的行为,故不能对 SVG 的路径进行统一运算求解,每条 SVG 对象路径需进行单独运算,以飞行器追踪动画为例,如图 2 所示.

图中每个小飞机都有各自的源节点与目的节点,并在节点间往返运动,并且源节点以及目的节点都会进行移动,此时飞行器实时地根据两点坐标计算轨迹,并修正飞行方向.第二节所介绍的三种技术应用于此场景均有不能接受的缺陷,如果使用定时器方式实现,每个 SVG 绑定一个回调函数进行循环刷新,最终飞行器增加到 10 个左右时,CPU 利用率就已经陡增至 80%左右,且浏览器已出现卡顿.而使用 rAF 则会回到第二节介绍的老问题,向前兼容导致的应用稳定性下降. CSS animation 则无法完成这种复杂的应用场景.

4 SVG 动画渲染优化算法

基于上一节 3 种现有渲染技术所暴露出来的缺点,设计了一种基于闭包以及函数式风格的技术思路,配合 rAF 技术,既保证了应用的稳定性(退化到计时器方式依然能有较好的性能),同时也能很好地实现复杂场景需求.本章将详细阐述基于 rAF 优化的 SVG 动画渲染技术的具体实现以及所采用的技术原理.

4.1 轨迹计算优化

假定现在有 N 个 SVG 元素,选择其中一个记为 $k(1 \leq k \leq N)$,其中 k 的源节点坐标记为 (x_{k0}, y_{k0}) ,目标节点坐标记为 (x_{k1}, y_{k1}) ,当前节点坐标记为

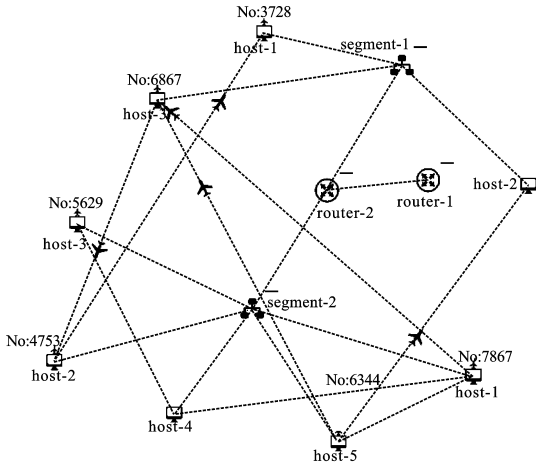


图2 飞行器追踪动画

(x_k, y_k) 。 k 的位移步长记为 ρ_k 。则 k 离目标节点偏移量 D_{xk}, D_{yk} 如式(1)、(2)所示。

$$D_{xk} = x_{k1} - x_k \quad (1)$$

$$D_{yk} = y_{k1} - y_k \quad (2)$$

步长 ρ_k 如式(3)所示, 其中 C 为常数, $f_k(D_{xk}, D_{yk})$ 为 k 与目标节点距离相关的步长函数:

$$\rho_k = f_k(D_{xk}, D_{yk}) * C \quad (3)$$

k 在 x 坐标与 y 坐标的偏移步长 Δ_{xk}, Δ_{yk} 如式(4)、(5)所示。

$$\Delta_{xk} = \rho_k * \frac{D_{xk}}{\sqrt{D_{xk}^2 + D_{yk}^2}} \quad (4)$$

$$\Delta_{yk} = \rho_k * \frac{D_{yk}}{\sqrt{D_{xk}^2 + D_{yk}^2}} \quad (5)$$

k 在下一个时间节点的坐标记为 (x'_k, y'_k) 如式(6)、(7)所示。

$$x'_k = x_k + \Delta_{xk} \quad (6)$$

$$y'_k = y_k + \Delta_{yk} \quad (7)$$

由上述公式可知, 节点 k 的位移仅节点 k 自身以及 k 运动轨迹的源节点与目标节点相关。又由于节点 k 由源节点出发, 故整个运动轨迹可以抽象为 $f_k(x_{k0}, y_{k0}, x_{k1}, y_{k1}, t)$ 其中 $1 \leq k \leq N, t$ 为时间。此时, $x_{k0}, y_{k0}, x_{k1}, y_{k1}$ 也会随时间动态位移, 最终 f_k 如式(8)所示。

$$f_k(x_{k0}, y_{k0}, x_{k1}, y_{k1}, t) \longrightarrow$$

$$f_k(x_{k0} + \theta_0(t), y_{k0} + \eta_0(t), x_{k1} + \theta_1(t), y_{k1} + \eta_1(t), t)$$

由公式可得出如下结论:

(1) 对于 SVG 元素 K , 可以由一个轨迹函数 f_k 进行描述;

(2) 轨迹函数间的入参互相独立, 互不干扰;

(3) K 的轨迹变化仅与源目节点在时间 t_0 时的坐标以及时间 t 有关。

故, 通过函数闭包, 可将 t_0 时刻 K 的源目节点坐标捕获, 并初始化 f_k 运算。利用函数式编程方式, 源目节点的轨迹函数被作为 f_k 的变量即 f_k 作为以 $\theta_0(t), \eta_0(t), \theta_1(t), \eta_1(t)$, 作为运算符的高阶函数来参与到 K 的轨迹运算当中。其中, $\theta_0(t), \eta_0(t), \theta_1(t), \eta_1(t)$ 计算源目节点的动态位移。

由于 K 的轨迹已抽象成以 t 为变量的函数 f_k , 任意多个 SVG 对象的运行轨迹则都能以同样的方式进行抽象, 将 n 个轨迹函数存储在内存当中, 对轨迹函数批量执行便完成了所有 SVG 轨迹在 Δt 时间内的一次坐标变换。此时, 只要不断对轨迹函数组进行调用, 实际上就已经完成多个 SVG 元素的移动动作。配合 rAF 函数或者定时函数, SVG 的动画渲染便成功实现。

由于所有运算动作被归并到一组运算函数式当中, 第二节所介绍的定时函数渲染缺陷将不会在此种实现机制中发生了。因为此时定时回调函数只需要初始化一个, 这一个回调函数将处理所有的运算任务, 而不是初始化和 SVG 数量正相关个回调函数, 每个函数处理一个计算任务, 示意图如图 3 所示。故 rAF 实现动画渲染时即便是浏览器不支持也不会出现巨大的性能抖动了, 此时的性能退化仅跟是否为原生 API 有关。

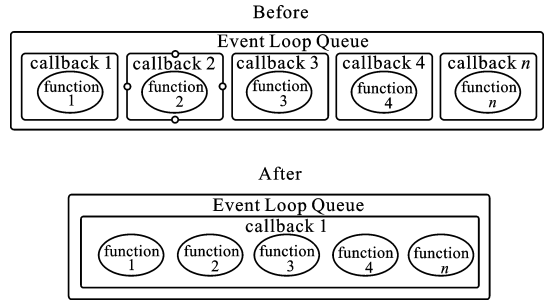


图3 优化前后的 Event Loop Queue

4.2 帧率自适应

帧率自适应算法是平衡 CPU 占用率的一种手段, 算法的目的在于当 SVG 动画数量逐渐增加时, 能根据渲染函数的耗时动态调整帧率, 使 CPU 占用率更加平缓, 可以显著提高应用的稳定性。自适应模型如式(9)~(11)所示。

$$\Theta = \frac{1\ 000}{\xi} \quad (9)$$

$$F = \max(\Theta, -O_{\text{cpu}}(t) * C_0 * \xi + 2C_1) \quad (10)$$

$$\vartheta = \max(\xi, \frac{1\ 000}{F}) - \xi \tag{11}$$

式中,渲染函数总耗时记为 ξ ,单位为毫秒, θ 为最大可达帧率。 $O_{\text{cpu}}(t)$ 为 CPU 为当前时间 t 时的 CPU 占用率, C_0 为 CPU 占用率修正值, C_1 为常数,一般代表人眼能够识别的最低帧,低于此帧率就会感觉到卡顿,一般值为 30。 F 代表自适应后的帧率,单位为帧/每秒。 ϑ 为换算后的延时补偿,单位为毫秒。补偿时间加上函数本身的执行时间为最终的帧与帧的时间差。修正前后,帧率与 SVG 动画数量的关系如图 4 所示。

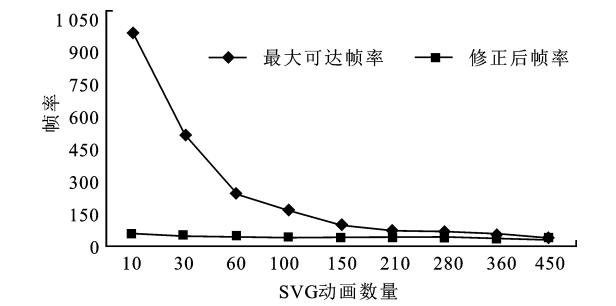


图 4 最大可达帧率与修正后帧率对比

由此可见,帧率修正后渲染函数具有稳定的执行时间,SVG 数量较少时,由于函数执行时间较快,将会添加一定时间的延时补偿,延时补偿是没有 CPU 资源消耗的,故在无 rAF 实现的浏览器上,即使退化成定时器算法,SVG 动画数量较小时,CPU 占用率能够维持在一个稳定值上。

5 性能测试

测试将以第三节介绍的场景作为测试场景,使用三种渲染技术进行对照,以此作为优化后的性能数据的对比。试验将对每种渲染技术进行十组测试,测试的 SVG 动画数量分别为 10、30、60、100、150、210、280、360、450、550。(递增抽样能够更清晰地表示动画数量陡增情况下性能的抖动情况)每个渲染技术每组测试重复 10 次取平均值,保证实验数据的准确性。测试机器为 Retina MacBook Pro 15 寸,CPU 为 2.2 GHz Intel Core i7 四核,满载 CPU 使用率为 400%,内存为 16 GB 1600 MHz DDR3。浏览器使用的 Chrome 54.0.2840.71 (64-bit)。

为了显示方便,rAF 优化后的算法记为 Algorithm*,退化后的优化算法记为 Algorithm-,以此模拟不支持 rAF 的低版本浏览器采用自适应算法的结果,计时器技术记为 EventLoop 以此模拟没有任何优化的结果,测试结果如图 5 所示。

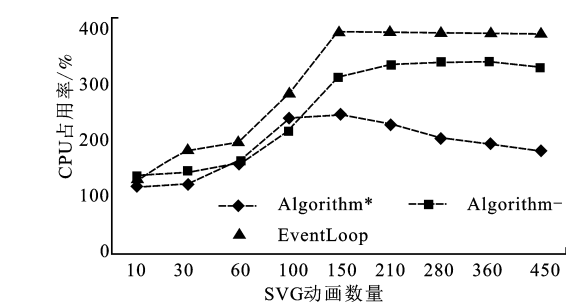


图 5 不同渲染技术的 CPU 占用率

图 5 中,Algorithm* 算法由于经过优化,在 CPU 使用率在 150 个 SVG 动画数时达到峰值,随后逐级递减,原因是随着 SVG 动画数量提升,整体函数调用时间延长,最高可达帧率慢慢小于 Chrome 浏览器本地实现的 rAF 默认帧率,造成了 rAF 每次重绘时部分位移函数并未执行,rAF 同时会采取降帧措施保证 CPU 占用率维持在相对低的水平,在 550 个 SVG 动画时,最高可达帧率保持在 33 帧左右。Algorithm* 退化后的 Algorithm-算法由于缺少图形优化,在最高可达帧率与 Algorithm* 算法相同的同时,CPU 占用率相对 Algorithm* 明显偏高,但仍然能支持超过数百的动画渲染。而未经优化的定时器算法,由于频繁调用线程上下文,在动画数在 150 左右的时候,CPU 占用率已到极限(如果 CPU 最大上线超过 400%,统计曲线将成线性上升趋势)帧率已低至 30 帧以下,由于已达极限,150 个动画后的数据已没有实际意义故未在图中表示。3 种算法的最高可达帧率对比如图 6 所示。

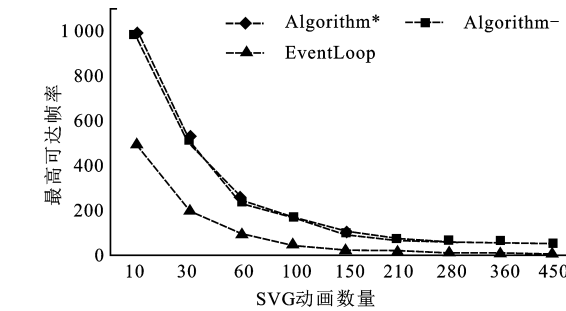


图 6 不同渲染技术的最高可达帧率

我们截取 SVG 动画数量为 150 左右时三种算法 CPU 空闲率以及最高可达帧率做一个统计,如图 7 所示。

在 150 个 SVG 动画时,Algorithm* 动画保证了 97 的最高可达帧率以及 166%左右的 CPU 空闲,而 Algorithm-最高可达帧率同样达到了 97,CPU 空闲率相对偏低点约为 102%。而 EventLoop

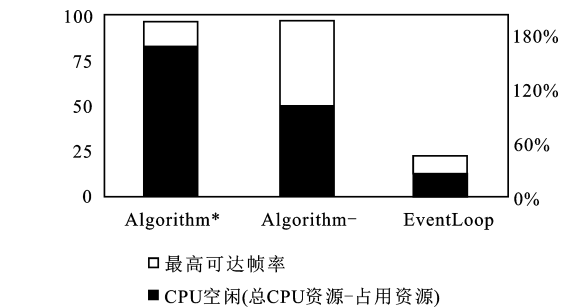


图 7 不同渲染技术 CPU 空闲以及最大可达帧率

帧率此时仅为 23,CPU 空闲率也只有 27%.

6 结束语

优化后的算法在超多动画渲染时(超过 100 个)有着较高的帧率以及低 CPU 占用率,并且在浏览器不支持 rAF 渲染技术的情况下仍然能保证帧率稳定,计时器算法则完全没能力支撑超多动画的应用场景.由此可得出结论,优化后的算法有着良好的运算效率以及稳定性,效率方面由于基于 rAF 进行优化,在 CPU 利用率上并无差别,而 rAF 算法在浏览器不支持的情况下会退化成定时器算法,效率急剧下降,而优化后的算法即便退化后也能在超多动画量的场景下保证稳定的帧率,在页面渲染应用中可以有效地提高应用对于不同浏览器环境下的稳定性.

本算法在具备浏览器帧同步优化的基础上通过减少运算符与回调映射,退化后系统调用相对会少很多,消耗的系统资源会显著减少,能够显著提高动画渲染性能.因此,当动画渲染应用此算法后,在不同浏览器上应用将显著提升应用的稳定性以及适应性.

参考文献:

[1] Kuryanovich E,Shalom S,Goldenberg R, et al. HT-ML5 Games Most Wanted [M] . Apress, 2012; 107-132.

[2] Shankar Aditya Ravi. HTML5 and JavaScript Essentials [M]. Apress, 2012;1-18.

[3] Simpson Kyle. You Don't Know JS: Async & Performance [M] . O'Reilly Media, Inc. , 2015.

[4] Tilkov Stefan, Steve Vinoski. Node. js: Using JavaScript to build high-performance network programs [C] // IEEE Internet Computing . IEEE, 2010: 80.

[5] Percival John. HTML5 Advertising [M]. Apress, 2013; 85-119.

[6] Muenno Atitayaporn, Daranee Hormdee. 3D Web-based HMI with WebGL Rendering Performance [C] // MATEC Web of Conferences. EDP, 2016.

[7] Hudson Shane. JavaScript Creativity. [EB/OL]. [2016-06-12]. <http://www.paulirish.com/2011/requestanimationframe-for-smart-animating/> .

[8] Garaizar Pablo, Miguel A Vadillo, Diego López-de-Ipiña. Presentation accuracy of the Web revisited: animation methods in the HTML5 era [J]. Plos one, 2013, 9(10):e109812.

作者简介:

周宇轩 男,(1991-),硕士研究生.研究方向为网络安全.

刘 丹(通讯作者) 男,(1969-),博士,副教授,硕士生导师.研究方向为网络安全. E-mail:liudan@uestc.edu.cn.

乐 俊 男,(1984-),博士,工程师.研究方向为网络安全.