# 剑三性能管理器：一个资源受限平台上的性能自平衡系统

# SO3 PERFORMANCE MANAGER - A PERFORMANCE SELF-BALANCE SYSTEM ON A RESOURCE-LIMITED PLATFORM

## 版本历史

[2014-06-27] 首次成文，由零碎的开发笔记整理而来，仅供参考

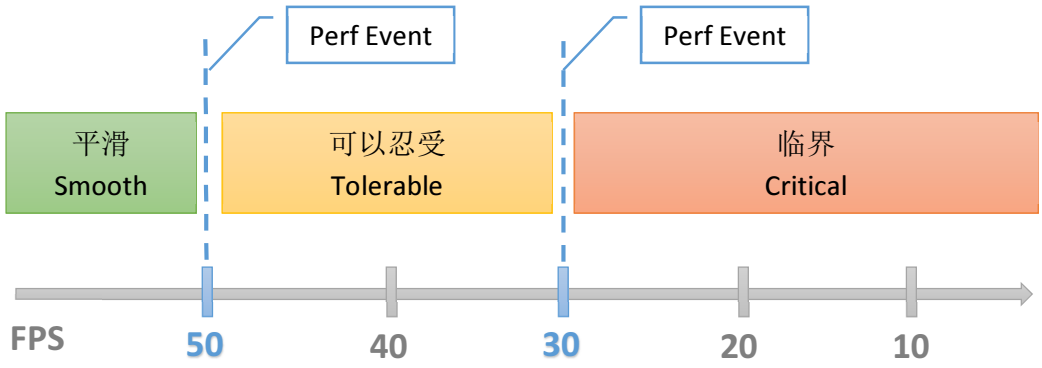
## 目录

基本原理和思路

## 基本原理：性能监控，分档和全局事件响应

一句话概括：检测系统当前运行帧数（fps），根据系统当前表现，动态做出反应，调整各个子系统的负载以降低压力，提高帧数。当系统恢复到健康平滑状态时，将负载逐步还给系统。

具体地说，我们将系统分为三档：“平滑(smooth) / 可以忍受(tolerable) / 临界(critical)” 目标帧率分别是 "50+ / 30-50 / 30-"，如下图所示：




当性能出现较大变动（跨档变动）时，向整个系统发送性能事件，每个子系统可以根据自己的情况响应该事件。（如性能持续恶化时，调整 NPC / 玩家的显示数量，释放较低优先级的贴图，调整渲染数目和植被的数量）

目前仅考虑对玩家影响最大的帧数指标，以后可逐步加入 MEM/IO/GPU 等的运行情况。随着系统对自身运行环境了解的越充分，对特定子系统的调节也就越有针对性。

## 这么做有啥牺牲？

玩家视野范围内一些相对不重要，或本来用于丰富场景的东西会被有选择地干掉（或简化）。干掉（及简化）的原则是（重要性依次降低）：

0) 当前系统的性能
1) 目标与玩家的距离
2) 目标对性能的影响程度
3) 目标对画面效果的提升程度

列举一些肉眼可见的会受影响的东西：

（具体的有角色蒙皮及渲染，装备及特效的渲染，草，树，水，小物件，地表，场景特效，后处理，等等，看情况策略可能略有不同）

## 这么做有啥好处？

[性能] 平均帧数的提高。并倾向于稳定在某一档。

[针对用户] 系统设置智能化。即使是高级用户也只需指定“画质优先”还是“性能优先”即可（小白直接用默认就行），程序会根据自己的运行情况自行动态地去调整。这样能够在不同地图，不同的情境下跑出不同的参数集，程序总是倾向于自动找到当下最优的方案，换句话说，小白用户知道系统会替他操心就可以了。（我们也可以采集大量的用户运行数据，这些运行数据比干巴巴的硬件列表更有用——举个栗子，如果是根本不懂维护系统的小白，好硬件一样会跑出卡游戏）

[针对开发者] 易于扩展。任意一个子系统，只要提供一个响应函数 OnPerfEvent()，即可对系统的性能事件做出反应，提供相应的调整。慢慢地整个系统就能变得有伸缩性，优化的职责也清晰地分摊到了开发具体模块的同事。可以缓解“优化永远跟在研发屁股后面追”的窘境。

## 其他注意事项

注意有一个性能事件是警告(Warning)。什么是警告呢？假设系统性能正在缓慢地下降，在程序判断出就快要跌到下一档之前，预先以每 3 秒一次的频率向整个系统发出警告，这时，如果各个子系统积极响应，妥善调整，系统就有很大的机会在进一步的恶化（掉到下一档）之前恢复到之前良好的状态。这比真的跌破后再去调整要好很多，对用户来说，“从没感觉到卡过”比“先变卡后恢复了”，体验要好得多，也会有效地缓解系统颠簸（就是在两档之间不断波动）。

# 第一版(V1)的系统设计和功能实现

前文的“基本原理和思路”是我在实现第一版(v1)之前，在草稿上涂画下的简略设计方案。性能监控部分的实现没什么好说的，代码很直白一看就懂。这里我主要说一下如何利用对性能事件的响应，来控制系统在活跃角色（视野内的玩家和 NPC）上的负载。

## 分页系统（PAGE IN / PAGE OUT）

我第一个想到的就是跟“操作系统对虚拟内存的使用方式”一样去处理。

当内存紧张时，操作系统会把那些较少访问的内存分页交换到硬盘上去，腾出物理内存来满足那些活跃的进程的需要。这里也是一样，当服务器同步过来较多的玩家和NPC，超出了（较低配置的）客户端机器的负载能力的时候，我们也可以把部分“相对而言不那么重要”的非活跃角色交换出去，达到降低系统负载，改善流畅度的目的。

交换出去的角色实际上只是隐藏了，内存数据都还在，随时可以恢复，以免加重 IO 负担。

如何定义在玩家的视野范围内，哪些是活跃玩家/NPC，哪些是非活跃角色呢？这就是仁者见仁智者见智的问题了，熟悉游戏逻辑的程序员能写出比我靠谱得多的逻辑来衡量一个玩家/NPC 角色的活跃度。

我目前的判断标准就是以下几点（其中 3 和 4 还没做）：

- 距离玩家的远近。（离玩家越远就越有可能被交换出去）
- 玩家比 NPC 活跃度更高。（有任务或其他交互的 NPC 可适当提高）
- 战斗状态的角色比非战斗状态活跃度更高
- 运动的角色比静止不动的角色活跃度更高

分页交换（Page swapping）的强度可以按照当前的性能评估等级来定。性能越差，Paging Out 可以越激进。而 Paging In 则不受当前性能程度的影响（具体原因后面会分析）。

## 优先级缓存队列

当有角色被交换出去的时候，我们使用一个优先级队列来缓存他们。随着玩家的移动，队列里的角色不断会被更新，直到满足一些条件（通常是玩家走得足够近）就可以被交换回来了（或由于下线，离开，玩家的远去而直接丢弃）。

以下这些情况发生时，可以考虑把队列里的角色立刻交换回来：

- 该角色发生了跟玩家有关的事件。
- 该角色向角色方向移动。

## “急进缓出”（GREEDY IN, GENTLE OUT）

请注意，在我的实现中，Page In 和 Page Out 不是在同一批次完成的。之所以把 Paging In 跟 Paging Out 独立开来，是为了达到“急进缓出”的效果，也就是说，交换出去总是即时生效的，而交换回来是分阶段逐步进行的。“急进”是为了在尽可能短的时间内迅速对性能变化做出反应，而“缓出”则是为了当性能恢复时，逐步将负载偿还

给系统。避免一下子塞给系统太多的负载，性能指标又迅速下跌。也可以减少系统在两档之间颠簸的情况。

## 第一版(V1)的问题和思考

第一版基本实现之后，我遇到了一些问题，这些问题出现得很及时，促使我思考如何去改善现有系统。这些问题主要有三点：

### 第一，缺乏细粒度控制

- Page In / Page Out 的特点是“非黑即白”，也就是要么出去要么进来，没有过渡，缺乏更细粒度的控制，对视觉效果影响过大。

### 第二，性能控制逻辑跟游戏逻辑互相影响，不够正交

- 分页系统的隐藏和显示直接使用了逻辑层的隐藏显示功能，实际上干扰了正常的游戏逻辑（举个栗子，游戏逻辑隐掉了某个角色，但分页系统又会把它给交换回来，改变了游戏的本来逻辑）

### 第三，系统的运算不够直观，出了 BUG 不好调试

- 随着时间的推移，逻辑的复杂化，分页系统的行为会越来越不直观，出了 bug 很不好查。比如有时候一个角色通过很复杂的规则判定后被交换出去了，你明知道这个是 bug，但比较难以推敲它是如何发生的。

### 结论：影响架构的根本性问题

深入地思考就能发现，这三点都不是“对现有系统修修补补”就能改善或缓解的问题，他们是影响架构的根本性问题。

## 第二版(V2)的设计和实现

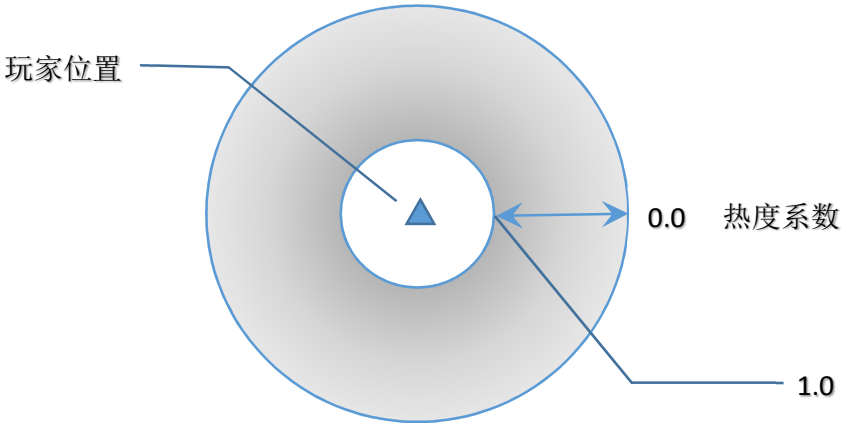
如前所述，我在实现第一版过程中遇到的问题，不是程度性问题而是根本性问题。要想彻底解决这些问题，唯有重新设计一途。那么如何设计才能既保住优点又改掉缺点呢？

答案就是所谓的“热度环系统”。

## 热度环系统 “HEAT RING”

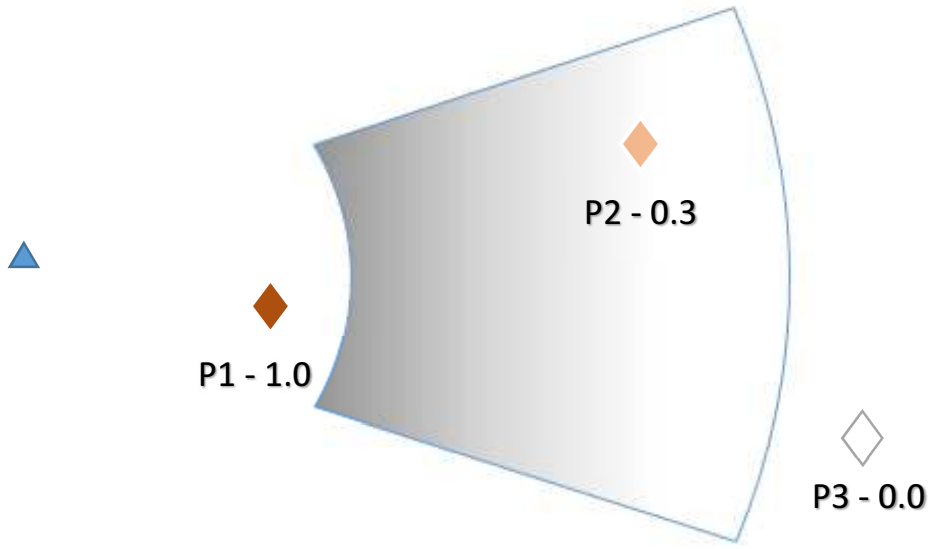
先说一下，“热度环”这个名词是我现编的，只是为了更形象地说明这个解决方案。

直接上图吧，比文字要直观，应该能省不少力气。




如图中所示，对当前玩家而言，总是存在这样一个环：环的外圈以外是他不关心的世界（热度系数为 0.0），环的内圈以内是他最为关心的区域（热度系数为 1.0），而环本身作为一个由 0.0 到 1.0 之间平滑过渡的灰色缓冲区域存在。这个环状过渡区域就是所谓的“热度环”。

通过下面的实例，可以看到热度环在性能管理系统中是怎么起作用的。




从图上可以看出，随着距离远近的不同，角色的热度值在 0.0 到 1.0 之间变化。当角色的热度值是 0.0 时，我们可以完全隐藏这个角色；当热度值是 1.0 时，可以认为这个角色完全可见；当位于过渡区域时，我们可以有选择地简化或移除一些部件/装备/特效，来达到更细粒度控制的目的。

目前角色身上（从性能角度值得去考虑的）细粒度控制的组件如下表所示：

- 本体模型
- PhysX 驱动的装备
- 装备武器
- 装备武器特效
- 本体特效
- 姓名板，气泡，任务图标等

## 热力环系统在系统架构上的关键性作用

热力环系统不仅在功能上，可用来实现更平滑的过渡，而且在系统架构上更是第二版(v2)的核心组件。

这是因为，通过热力值这个非常单纯的 0.0-1.0 的值，我们实现了性能管理器的全局调度算法（全局性能控制）和角色内部性能控制（单体性能控制）的解耦。

有了这个值的存在，我们可以随意改进全局的调度算法（就是这个值是怎么算出来的），而完全不用操心角色身上的具体实现（就是这个值是怎么被使用的）；反过来，也可以随意改进角色内部的性能粒度的控制，而不用担心外部影响。

更重要的是，随着游戏的开发，角色身上的组件可能会新增和改变，通过热力值的传递，我们把这些改变对性能系统的影响隔离在了角色类的内部，避免了牵一发而动全身。

附 [2014-06-26] 开发日志，更多信息供参考

> *之前一直没想清楚，哪些逻辑在 KRLPEHandler_RemoteCharacter 那边实现，哪些逻辑在单个的角色 KRLRemoteCharacter 这边实现。*
>
> *刚刚想清楚了，给 KRLRemoteCharacter 添加了一个重要的函数*
>
> *DWORD RefreshPerformanceStatus(float factor); // [0.0f, 1.0f]*
>
> *这个函数用来把一个 0.0-1.0 的值翻译成各种性能标记打到角色对应的部件上*
>
> *这样一来，“RemoteCharacter 内部逻辑结构” 就和“性能管理器”之间解耦了*
>
> *以后 RemoteCharacter 内部改进，添加或细化更多的性能控制选项，性能管理器都不必知道；而性能管理器的算法改进，比如实现了更智能的 page in/out，RemoteCharacter 这边也不必知道*

## 在 REMOTE CHARACTER 类内新增专门的性能标记

这个性能标记主要是用来解决第一版的第二个问题，将性能控制逻辑跟游戏本身的逻辑从物理上隔离开，避免相互干扰，实现系统的正交化。

摘录该变量的注释如下

```
/*
--- 性能状态标记 ---
这里记录的状态是性能管理器关心的设置。

一些约定：
1. 比现有的游戏逻辑内对应的设置（如 m_eShowLevel）的优先级要低。也就是说，只有在游戏里被确认开启的情况下，
    才会进一步考虑这里的设置。这样的物理隔离是为了从根源上避免与已有逻辑互相影响。
2. 这些设置应该是非侵入的，只影响视觉表现，不应修改RL内各种对象之间的逻辑关系，以免破坏已有的游戏逻辑的假定。
    比如是否使用空模型（Empty Model），是否有 Attach / Bind 的关系，等等。
*/
```

## V2 对 V1 出现的各种问题的解决情况

热度环解决了“缺乏细粒度控制”的问题，同时改善了系统架构。

专门增加的性能标记解决了“性能控制跟游戏逻辑控制相互干扰”的问题

热度环系统的实现及其简单，核心代码不到 30 行，大幅简化了原有系统（核心部分 150 行）而且很直观，只要把任意一个角色的热力值打印到屏幕上即可随时监视，调试无压力。

## 其他的考虑和备注

### 消除性能管理器本身对系统性能的冲击

有一些专门考虑是为了消除性能管理器本身的性能压力。

首先，整个管理器不需要那么高的实时性，因此它不是每帧更新而是每秒更新的。

但是如果每次更新的时候，所有角色都过一遍，更新量太大也会吃不消，所以每个角色维护一个单独的更新时间戳，系统每隔 0.1 秒检查一次，更新那些到期了的角色。这是为了分摊更新的压力。

如下图所示：