Contents

1 前言

Unity 是个普及度很高拥有大量开发者的游戏开发引擎，其提供的 *Unity* 编辑器可以快速的开发移动设备游戏，并且通过编辑器扩展可以很容易开发出项目需要的辅助工具，但是 *Unity* 提供性能调试工具非常简陋，功能简单并且难以使用，如果项目出现性能问题，定位起来相当花时间，并且准确率很低，一定程度上靠运气。

Profiler

目前 Unity 随包提供的只有 *Profiler* 工具，里面聚合 *CPU*、*GPU*、内存、音频、视频、物理、网络等多个维度的性能数据，但是我们大部分情况下只是用它来定位卡顿问题，也就是主要 *CPU* 时间消耗 (图1)。

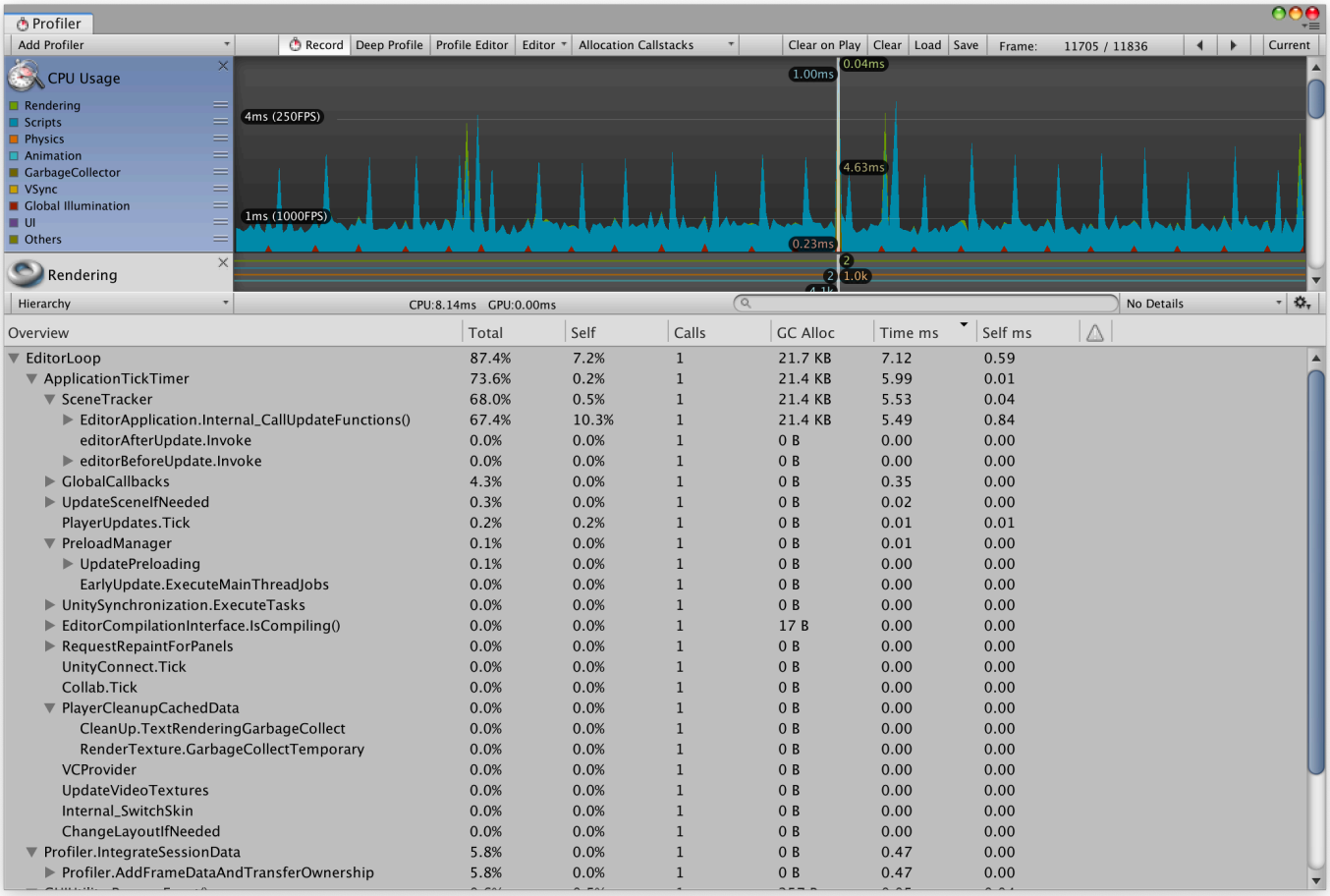


Figure 1: Unity 性能调试工具 Profiler

在 *CPU* 的维度里面，可以看到当前渲染帧的函数调用层级关系，以及每个函数的时间开销以及调用次数等信息，但是这个工具同一时间只能处理 **300** 帧左右的数据，如果游戏帧率 30，那么只能看到过去 **10** 秒的信息，并且需要我们一直盯着图表看才有机会发现意外的丢帧情况，这种设计非常的不友好，违反正常人的操作习惯，因为通常情况下如果我要调试游戏内战斗过程的性能开销，首先我要像普通玩家那样安安静静的玩一把，而不是分散出大部分精力去看一个只有 10 秒历史的滚动图表。这种交互带来两个明显的问题，

- 由于分心去看 *Profiler*，导致不能全心投入游戏，从而不能收集正常战斗过程的性能数据
- 为了收集数据需要像正常玩家那样打游戏，不能全神关注 *Profiler* 图表，从而不能发现/查看所有的性能问题

上面两个情形相互排斥，鱼与熊掌不可兼得。从这个角度来看，*Profiler* 不是一个好的性能调试工具，苛刻的操作条件导致我们很难发现性能问题，想要通过 *Profiler* 定位所有的性能问题简直是痴人说梦。

MemoryProfiler

Unity 还提供另外一个内存分析工具 MemoryProfiler(图2)

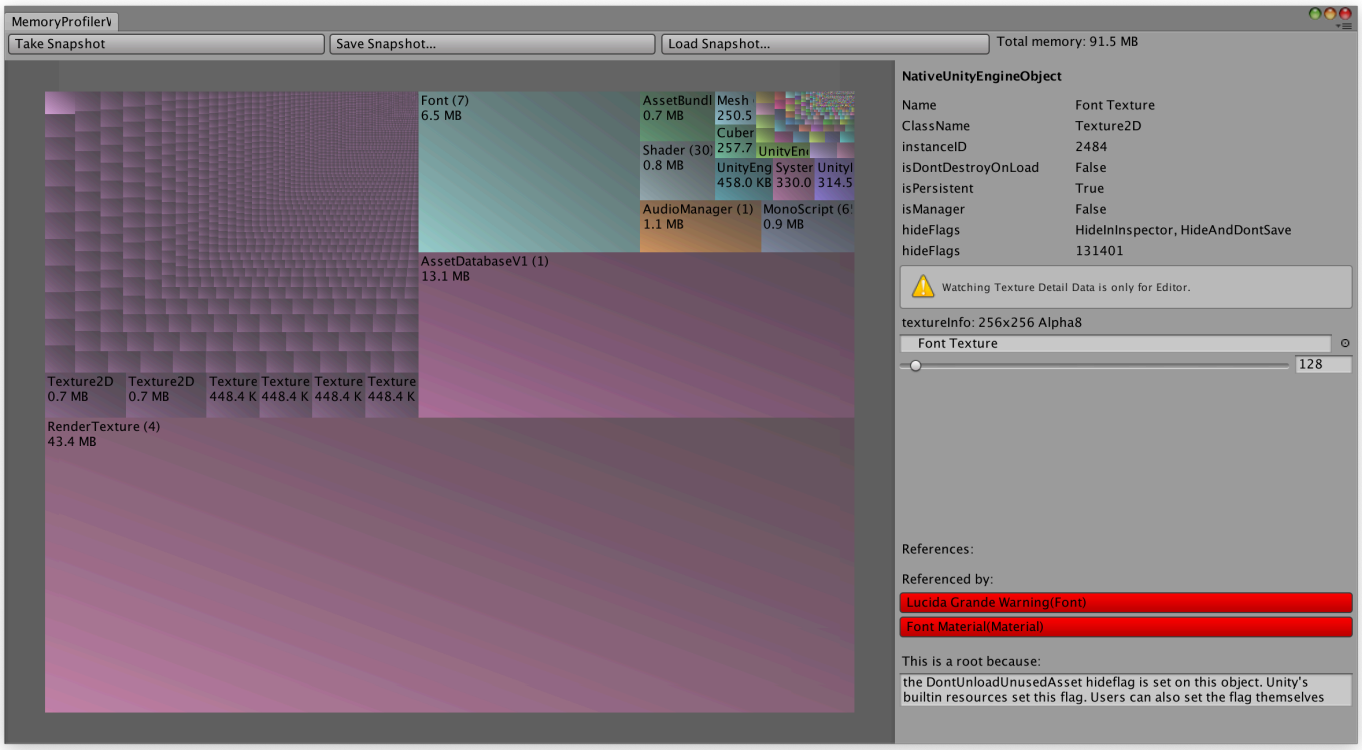


Figure 2: Unity 内存调试工具 MemoryProfiler

在这个界面的左边彩色区域里，*MemoryProfiler* 按类型占用总内存大小绘制对应面积比例的矩阵图，第一次看到还是蛮酷炫的，*Unity* 是想通过这个矩阵图向开发者提供对象内存检索入口，但是实际使用过程中问题多多。

- 内存分析过程缓慢
- 在众多无差别的小方格里面找到实际关心的资源很难，虽然可以放大缩小，但感觉并没有提升检索的便利性
- 每个对象只提供父级引用关系，无法看到完整的对象引用链，容易在跳转过程中迷失
- 引擎对象的引用和 *IL2CPP* 对象的引用混为一谈，让使用者对引用关系的理解模糊不清
- 没有按引擎对象内存和 *IL2CPP* 对象内存分类区别统计，加深使用者对内存使用的误解

*MemoryProfiler*源码托管在 *Bitbucket*，但是从最后提交记录来看，这个内存工具已经超过 **2** 年半没有任何更新了，但是这期间 *Unity* 可是发布了好多个版本，想想就有点后怕。

Unity Technologies / Unity / MemoryProfiler

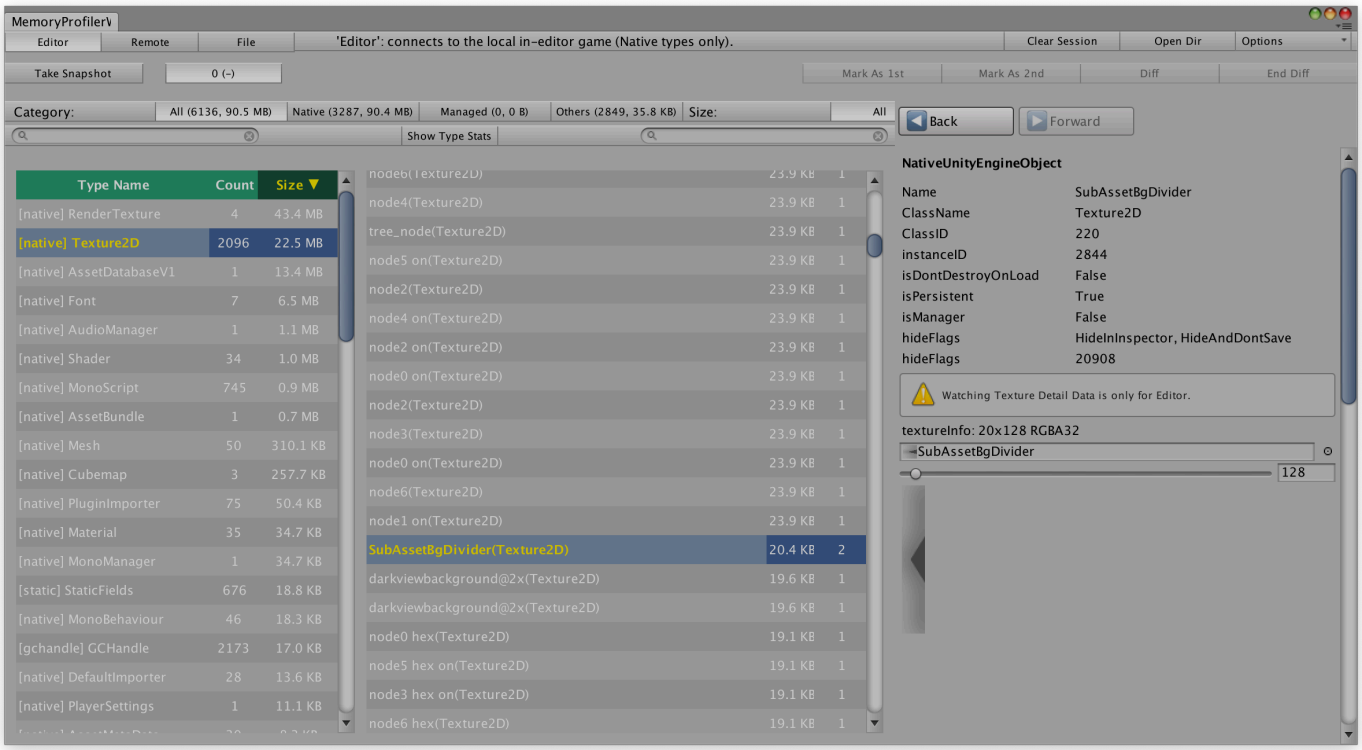
Commits

All branches

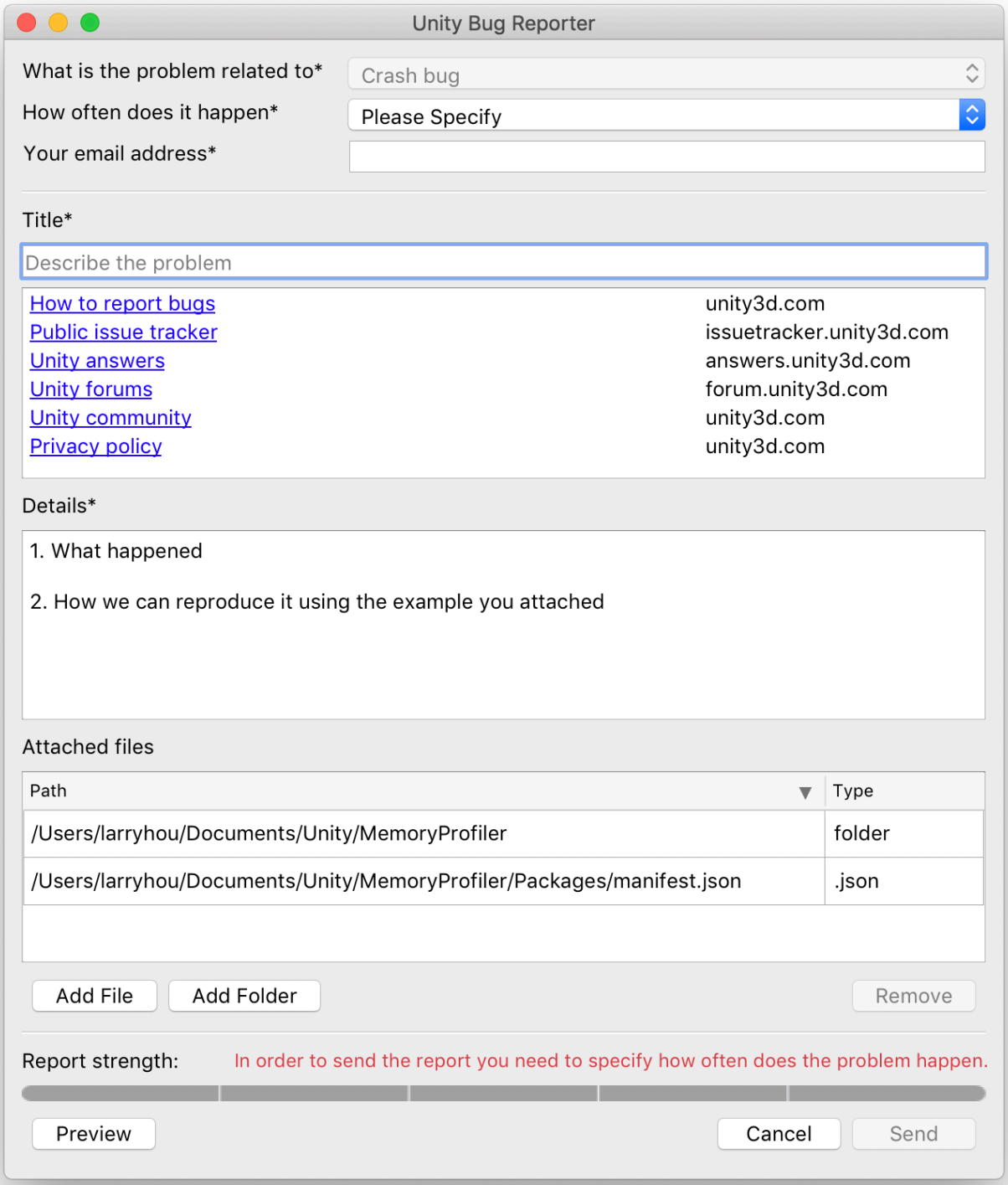
Find commits

Author	Commit	Message	Date	Builds
Lukasz Paczko...	e8317df M	Merged in lukaszunity/include-mono-support-in-readme (pull request ...	2017-10-13	
Lukasz Paczko...	abd35a7	Close branch lukaszunity/include-mono-sup... lukaszunity/includ...	2017-10-13	
Lukasz Paczko...	8322cc4	Use Mono .NET 3.5 wording instead of 2.0 in ... lukaszunity/includ...	2017-09-27	
Lukasz Paczko...	11f359d	Include Mono support in README lukaszunity/includ...	2017-09-27	
Tautvydas Žilys	4b2f619 M	Merged in flassari/memoryprofiler/zero-positive-fix (pull request #30)...	2017-07-29	
Ari Arnbjörnss...	7e8a974	Treat zero as a positive number zero-positive-fix	2017-07-27	
Tautvydas Žilys	a0d5a31 M	Merged in aghogg/memoryprofiler/classid_remove (pull request #31) ...	2017-07-29	
Ashley Hogg	c3d91ad	Removed ClassID field from the object inspector, s... classid_remove	2017-07-20	
Tautvydas Žilys	7126ded M	Merged in aghogg/memoryprofiler/64bit_size_fix (pull request #32) C...	2017-07-29	
Ashley Hogg	6478c5d	Changed size members to 64-bit longs, to fix over... 64bit_size_fix	2017-07-20	

有热心开发者也忍受不了 *Unity* 这缓慢的更新节奏，干脆自己动手基于源码在github上更新优化，并更改了检索的交互方式。



不过这也只是在 *MemoryProfiler* 的基础上增加检索的便利性，跟理想的检索工具还有很大差距，虽然在内存的类别上做了相对 *MemoryProfiler* 更加清晰的区分，但是没有系统化的重构设计，内存分析过程依然异常缓慢，甚至会在分析过程中异常崩溃。

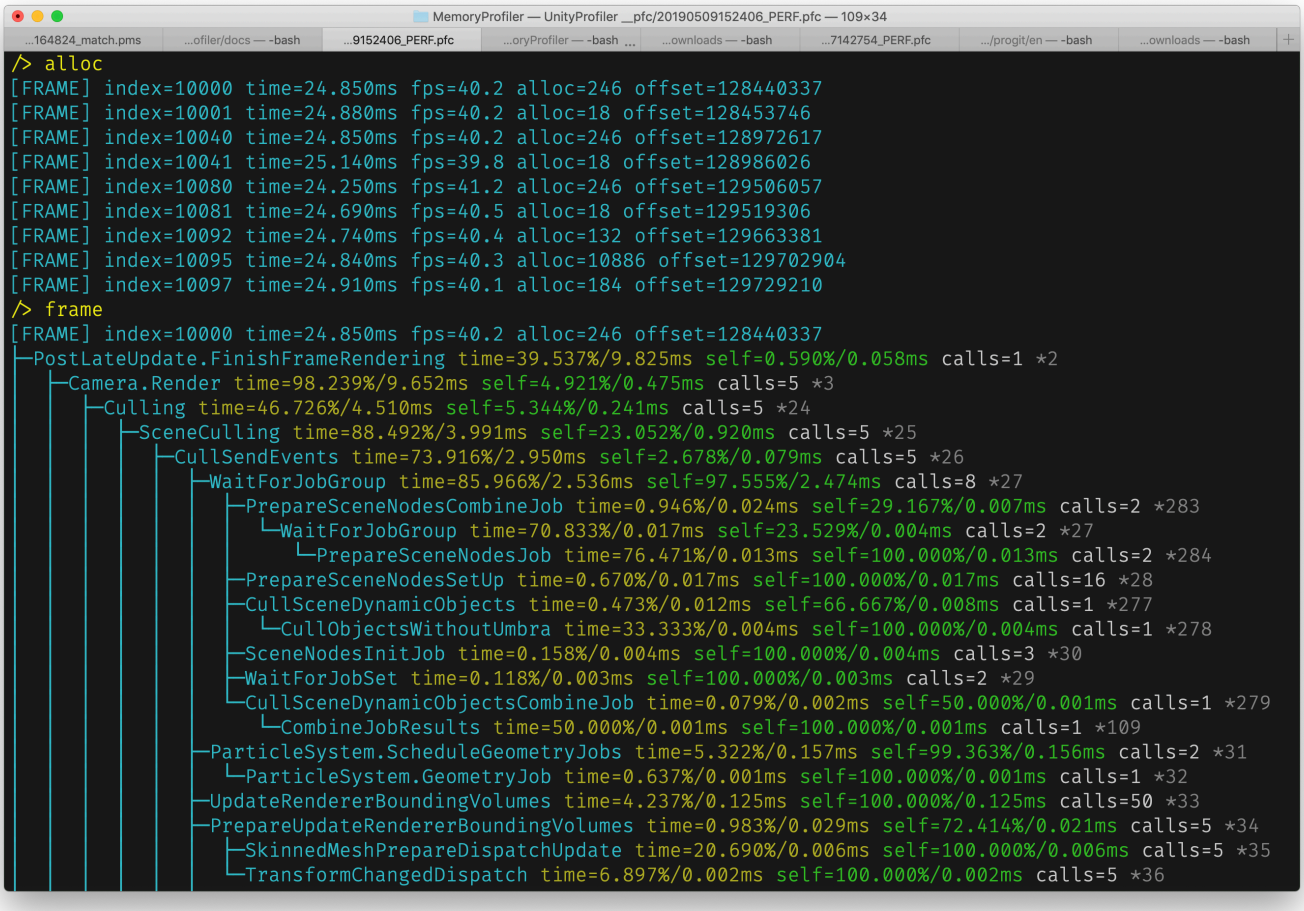


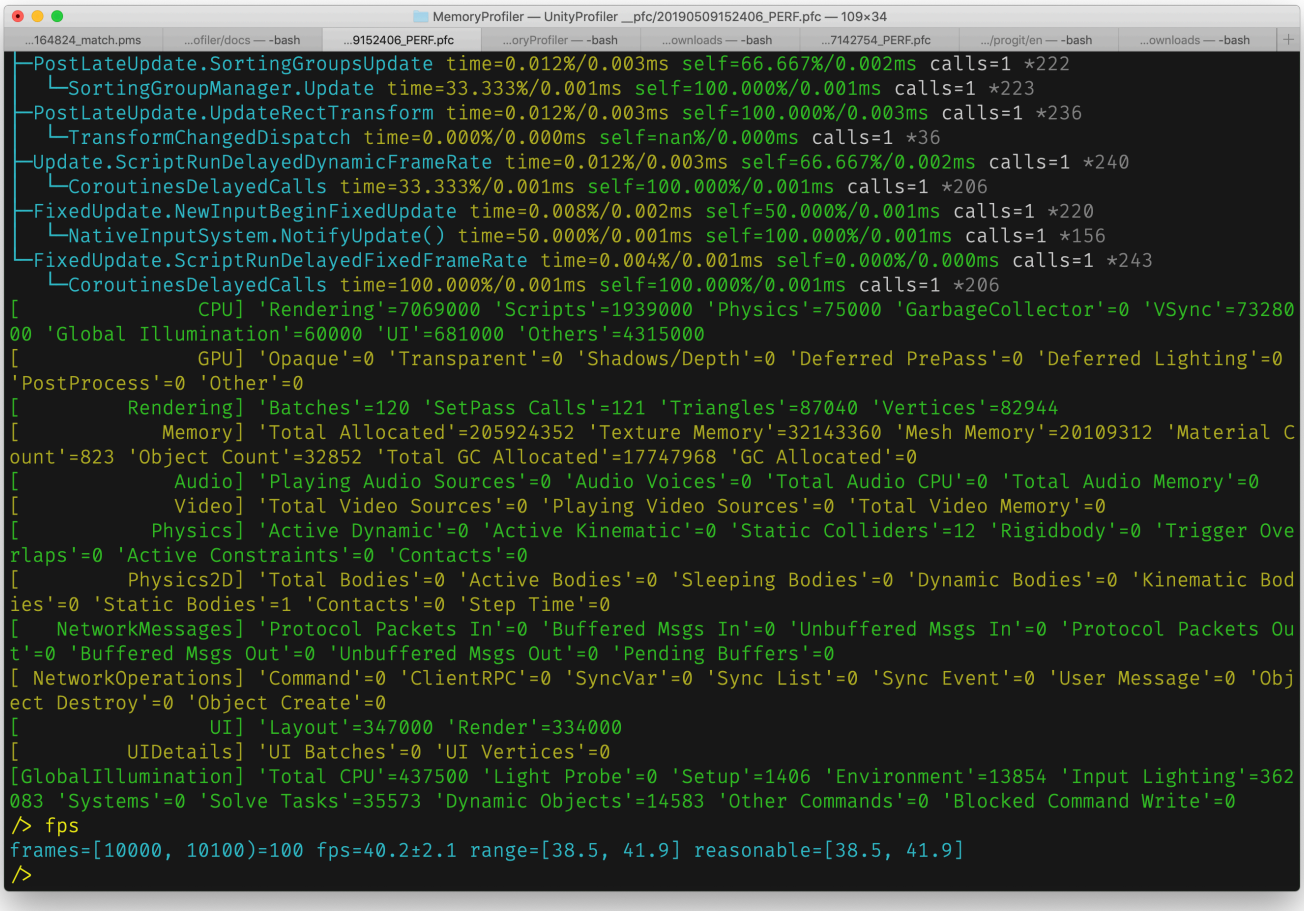
基于 *Unity* 性能调试工具现实存在问题，我开发了 UnityProfiler 和 MemoryCrawler 两款工具，分别替代 *Profiler* 以及 *MemoryProfiler* 进行相同领域的性能调试，这两个款均使用纯 *C++* 实现，因为经过与 *C#*、*Python* 对比后发现 *C++* 有绝对的计算优势，可以非常明显提升性能数据分析效率和稳定性。但是这两个款工具都没有可视化交互界面，不过并不影响分析结果的查看，它们都内置命令行模式的交互方式，并提供了丰富的命令，可以对性能数据做全方位的分析定位。

2 UnityProfiler

2.1 简介

UnityProfiler 以 *Unity* 引擎自带的 *Profiler* 工具生成的性能数据为基础，提供多种维度的工具来帮助发现性能问题。该工具前期预研阶段使用 *Python* 测试逻辑，最终使用 *C++* 实现。由于是基于 *Unity* 的原生接口获取数据，所以需要保证 *Profiler* 工具打开后能看到性能采集界面，真机调试确保按照官方文档正确配置。





2.2 原理

Unity 编辑器提供的 *Profiler* 调试工具，有多个维度的性能数据，我们比较常用的就是查看 *CPU* 维度的函数调用开销。这个数据可以通过 *Unity* 未公开的编辑器库 *UnityEditorInternal* 来获取，鉴于未公开也谈不上查阅官方文档来获取性能数据采集细节，所以需要通过反编译查看源码才能知道其实现原理：构造类 *UnityEditorInternal.ProfilerProperty* 对象，调用 *GetColumnAsSingle* 方法来获取函数调用堆栈相关的性能数据。

```
var root = new ProfilerProperty();
root.SetRoot(frameIndex, ProfilerColumn.TotalTime, ProfilerViewType.Hierarchy);
root.onlyShowGPUSamples = false;

var drawCalls = root.GetColumnAsSingle(ProfilerColumn.DrawCalls);
samples.Add(sequence, new StackSample
{
    id = sequence,
    name = root.propertyName,
    callsCount = (int)root.GetColumnAsSingle(ProfilerColumn.Calls),
    gcAllocBytes = (int)root.GetColumnAsSingle(ProfilerColumn.GCMemory),
    totalTime = root.GetColumnAsSingle(ProfilerColumn.TotalTime),
```

```
    selfTime = root.GetColumnAsSingle(ProfilerColumn.SelfTime),
});
```

除了函数堆栈方面的开销，*Unity* 还有渲染、物理、*UI*、网络等其他维度的数据，这些数据要通过另外一个接口来获取。

```
for (ProfilerArea area = 0; area < ProfilerArea.AreaCount; area++)
{
    var statistics = metadatas[(int)area];
    stream.Write((byte)area);
    for (var i = 0; i < statistics.Count; i++)
    {
        var maxValue = 0.0f;
        var identifier = statistics[i];
        ProfilerDriver.GetStatisticsValues(identifier, frameIndex, 1.0f,
                                           provider, out maxValue);
        stream.Write(provider[0]);
    }
}
```

本工具基于以上接口把采集到的数据保存为 *PFC* 格式，该格式为自定义格式，使用了多种算法优化数据存储，比 *Unity* 编辑器录制的原始数据节省 **80%** 的存储空间，同时用 *C++* 语言编写多种维度的性能分析工具，可以高效率地定位卡顿问题。

2.3 命令手册

2.3.1 alloc

alloc *[frame_offset][=0] [frame_count][=0]*

参数	可选	描述
frame_offset	是	指定起始帧，相对于当前帧区间第一帧的整形偏移量
frame_count	是	指定帧数量

alloc 可以在指定的帧区间内搜索所有调用 *GC.Alloc* 分配内存的渲染帧。

```
/> alloc 0 1000
[FRAME] index=2 time=23.970ms fps=41.7 alloc=10972 offset=12195
[FRAME] index=124 time=25.770ms fps=38.8 alloc=184 offset=1326925
[FRAME] index=127 time=24.870ms fps=40.2 alloc=10972 offset=1359192
[FRAME] index=250 time=25.740ms fps=38.8 alloc=184 offset=2682771
[FRAME] index=253 time=24.690ms fps=40.5 alloc=10972 offset=2715142
```

如果 *frame_offset* 和 *frame_count* 留空，alloc 将所有可用帧作为参数进行条件搜索。

2.3.2 info

无参数，查看当前性能录像的基本信息。

```
frames=[1, 44611)=44610 elapse=(1557415446.004, 1557416582.579)=1136.574s
fps=39.9±12.8 range=[1.3, 240.6] reasonable=[27.2, 52.5]
```

2.3.3 frame

frame *[frame_index] [stack_depth][=0]*

参数	可选	描述
frame_index	**否**	指定帧序号
stack_depth	是	指定函数调用堆栈层级，默认完整堆栈

frame 可以查看指定渲染帧的详细函数调用堆栈信息，见下图。

```
/> info
frames=[1, 44611)=44610 elapse=(1557415446.004, 1557416582.579)=1136.574s fps=39.9±12.8 range=[1.3, 240.6] re
asonable=[27.2, 52.5]
/> frame 200 1
[FRAME] index=200 time=26.030ms fps=38.4 alloc=0 offset=2144297
├─Initialization.PlayerUpdateTime time=59.516%/15.492ms self=0.123%/0.019ms calls=1 *0
├─PostLateUpdate.FinishFrameRendering time=16.881%/4.394ms self=1.434%/0.063ms calls=1 *2
├─Update.ScriptRunBehaviourUpdate time=6.185%/1.610ms self=0.311%/0.005ms calls=1 *75
├─PreLateUpdate.ScriptRunBehaviourLateUpdate time=2.144%/0.558ms self=0.717%/0.004ms calls=1 *110
├─PostLateUpdate.EnlightenRuntimeUpdate time=1.398%/0.364ms self=1.923%/0.007ms calls=1 *67
├─PostLateUpdate.ProfilerEndFrame time=0.999%/0.260ms self=1.923%/0.005ms calls=1 *126
├─PreUpdate.SendMouseEvents time=0.715%/0.186ms self=2.151%/0.004ms calls=1 *143
├─PostLateUpdate.PlayerUpdateCanvases time=0.699%/0.182ms self=2.747%/0.005ms calls=1 *137
├─PostLateUpdate.PlayerEmitCanvasGeometry time=0.642%/0.167ms self=2.994%/0.005ms calls=1 *133
├─PreLateUpdate.ParticleSystemBeginUpdateAll time=0.611%/0.159ms self=1.887%/0.003ms calls=1 *119
├─PostLateUpdate.UpdateAllRenderers time=0.592%/0.154ms self=20.130%/0.031ms calls=1 *108
├─PostLateUpdate.UpdateCustomRenderTextures time=0.315%/0.082ms self=7.317%/0.006ms calls=1 *147
├─FixedUpdate.ScriptRunBehaviourFixedUpdate time=0.257%/0.067ms self=7.463%/0.005ms calls=1 *150
├─PostLateUpdate.PlayerSendFrameStarted time=0.161%/0.042ms self=23.810%/0.010ms calls=1 *154
├─PreUpdate.PhysicsUpdate time=0.134%/0.035ms self=8.571%/0.003ms calls=1 *165
├─FixedUpdate.ScriptRunDelayedTasks time=0.131%/0.034ms self=20.588%/0.007ms calls=1 *163
├─EarlyUpdate.NewInputBeginFrame time=0.119%/0.031ms self=58.065%/0.018ms calls=1 *168
├─EarlyUpdate.UpdatePreloading time=0.104%/0.027ms self=18.519%/0.005ms calls=1 *171
```

在函数堆栈的底部，还有当前帧其他性能指标数据，主要有 *CPU*、*GPU*、*Rendering*、*Memory*、*Audio*、*Video*、*Physics*、*Physics2D*、*NetworkMessages*、*NetworkOperations*、*UI*、*UIDetails*、*GlobalIllumination* 等 13 类别，一共 77 个细分性能指标。

```
├─PostLateUpdate.UpdateRectTransform time=0.015%/0.004ms self=50.000%/0.002ms calls=1 *236
└─Update.ScriptRunDelayedDynamicFrameRate time=0.015%/0.004ms self=50.000%/0.002ms calls=1 *240
[               CPU] 'Rendering'=3757000 'Scripts'=2073000 'Physics'=83000 'GarbageCollector'=0 'VSync'=15473
000 'Global Illumination'=357000 'UI'=289000 'Others'=1796000
[               GPU] 'Opaque'=0 'Transparent'=0 'Shadows/Depth'=0 'Deferred PrePass'=0 'Deferred Lighting'=0
'PostProcess'=0 'Other'=0
[         Rendering] 'Batches'=21 'SetPass Calls'=19 'Triangles'=0 'Vertices'=0
[            Memory] 'Total Allocated'=58493952 'Texture Memory'=7758848 'Mesh Memory'=122880 'Material Count
'=28 'Object Count'=4817 'Total GC Allocated'=6246400 'GC Allocated'=0
[             Audio] 'Playing Audio Sources'=0 'Audio Voices'=0 'Total Audio CPU'=0 'Total Audio Memory'=0
[             Video] 'Total Video Sources'=0 'Playing Video Sources'=0 'Total Video Memory'=0
[           Physics] 'Active Dynamic'=0 'Active Kinematic'=0 'Static Colliders'=0 'Rigidbody'=0 'Trigger Over
laps'=0 'Active Constraints'=0 'Contacts'=0
[         Physics2D] 'Total Bodies'=0 'Active Bodies'=0 'Sleeping Bodies'=0 'Dynamic Bodies'=0 'Kinematic Bod
ies'=0 'Static Bodies'=1 'Contacts'=0 'Step Time'=0
[   NetworkMessages] 'Protocol Packets In'=0 'Buffered Msgs In'=0 'Unbuffered Msgs In'=0 'Protocol Packets Ou
t'=0 'Buffered Msgs Out'=0 'Unbuffered Msgs Out'=0 'Pending Buffers'=0
[ NetworkOperations] 'Command'=0 'ClientRPC'=0 'SyncVar'=0 'Sync List'=0 'Sync Event'=0 'User Message'=0 'Obj
ect Destroy'=0 'Object Create'=0
[                UI] 'Layout'=45000 'Render'=244000
[         UIDetails] 'UI Batches'=0 'UI Vertices'=0
[GlobalIllumination] 'Total CPU'=189063 'Light Probe'=0 'Setup'=3334 'Environment'=33646 'Input Lighting'=788
02 'Systems'=0 'Solve Tasks'=27656 'Dynamic Objects'=25834 'Other Commands'=0 'Blocked Command Write'=0
```

每次执行 frame 都会记录当前查询的帧序号，如果所有参数留空，则重复查看当前帧数据。

2.3.4 next

next *[frame_offset][=1]*

参数	可选	描述
frame_offset	是	相对当前帧的偏移

next 命令相当于按照指定帧偏移量修改当前帧序号同时调用 frame 命令生成帧数据。

2.3.5 prev

prev *[frame_offset][=1]*

参数	可选	描述
frame_offset	是	相对当前帧的偏移

prev 命令相当于按照指定帧偏移量修改当前帧序号同时调用 frame 命令生成帧数据。

2.3.6 func

func *[rank][=0]*

参数	可选	描述
rank	是	指定显示排行榜前 *rank* 个数据

func 在当前可用帧区间内，按照函数名统计每个函数的时间消耗，并按照从大到小的顺序排序，*rank* 参数可以限定列举范围，默认列举所有函数的时间统计。

```
/> func 1
26.27%   1276.54ms #200     ████████████████████████ WaitForTargetFPS *1
/> func 5
26.27%   1276.54ms #200     ████████████████████████ WaitForTargetFPS *1
 6.80%    330.49ms #200     ██████ Profiler.CollectMemoryAllocationStats *128
 4.31%    209.44ms #1818    ████ RenderForward.RenderLoopJob *7
 3.89%    188.95ms #909     ████ SceneCulling *25
 3.11%    151.03ms #9071    ███ WaitForJobGroup *27
```

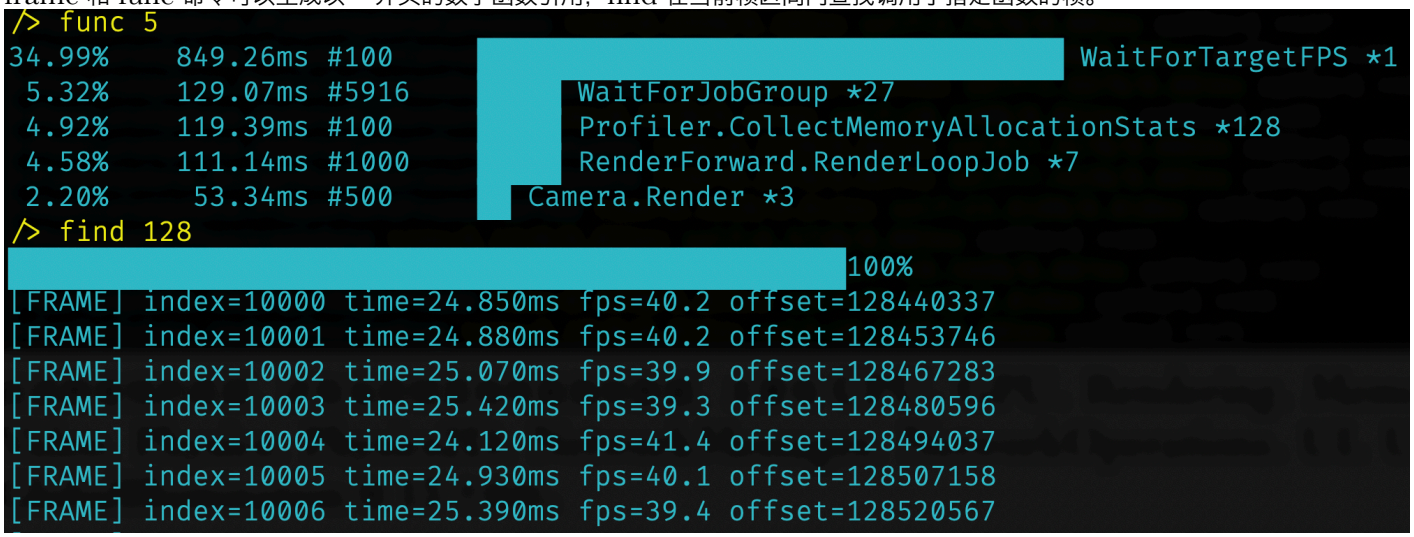
第一列表示函数时间消耗百分比，第二列表示时间消耗的总毫秒数，第三列表示函数调用的总次数，最后一列以 * 开头的数字表示函数引用。

2.3.7 find

find *[function_ref]*

参数	可选	描述
function_ref	**否**	指定函数引用

frame 和 func 命令可以生成以 * 开头的数字函数引用，find 在当前帧区间内查找调用了指定函数的帧。



2.3.8 list

list *[frame_offset][=0] [frame_count][=0]*

参数	可选	描述
frame_offset	是	指定始帧，相对于当前帧区间第一帧的整形偏移量
frame_count	是	指定帧数量

list 列举指定范围的帧基本信息，如果所有参数留空则列举当前帧区间的所有帧信息。

```
/> list 0 10
[FRAME] index=20000 time=24.900ms fps=40.2 offset=261033153
[FRAME] index=20001 time=25.230ms fps=39.6 offset=261046018
[FRAME] index=20002 time=24.530ms fps=40.8 offset=261059203
[FRAME] index=20003 time=24.840ms fps=40.2 offset=261072356
[FRAME] index=20004 time=24.880ms fps=40.2 offset=261084797
[FRAME] index=20005 time=24.900ms fps=40.2 offset=261097310
[FRAME] index=20006 time=25.270ms fps=39.6 offset=261110143
[FRAME] index=20007 time=24.470ms fps=40.9 offset=261122944
[FRAME] index=20008 time=25.340ms fps=39.5 offset=261135865
[FRAME] index=20009 time=24.400ms fps=41.0 offset=261148698
[SUMMARY] fps=40.2±1.6 range=[39.5, 41.0] reasonable=[39.5, 41.0]
```

该工具同时在所有帧数据底部生成 *fps* 统计数据。

2.3.9 meta

meta 查看性能指标索引，包含 *CPU*、*GPU*、*Rendering*、*Memory*、*Audio*、*Video*、*Physics*、*Physics2D*、*NetworkMessages*、*NetworkOperations*、*UI*、*UIDetails*、*GlobalIllumination* 等 13 类别，以及类别属性一共 77 个细分性能指标，每个性能指标由类别和类别属性两个索引确定，比如 *Scripts* 由 0-1 确定，该命令用来为 stat 和 seek 提供输入参数。

2.3.10 lock

lock *[frame_index][=0] [frame_count][=0]*

参数	可选	描述
frame_index	是	起始帧序号
frame_count	是	锁定相对于起始帧的帧数量

lock 参数留空恢复原始帧区间，一旦锁定帧区间，其他除 info 命令以外的其他命令均在该区间执行相关操作。

```
/> lock 10000 20
frames=[10000, 10020)
/> list
[FRAME] index=10000 time=24.850ms fps=40.2 offset=128440337
[FRAME] index=10001 time=24.880ms fps=40.2 offset=128453746
[FRAME] index=10002 time=25.070ms fps=39.9 offset=128467283
[FRAME] index=10003 time=25.420ms fps=39.3 offset=128480596
[FRAME] index=10004 time=24.120ms fps=41.4 offset=128494037
[FRAME] index=10005 time=24.930ms fps=40.1 offset=128507158
[FRAME] index=10006 time=25.390ms fps=39.4 offset=128520567
[FRAME] index=10007 time=24.590ms fps=40.7 offset=128533880
[FRAME] index=10008 time=24.560ms fps=40.7 offset=128547161
[FRAME] index=10009 time=24.900ms fps=40.2 offset=128560474
[SUMMARY] fps=40.2±1.9 range=[39.3, 41.4] reasonable=[39.3, 41.4]
```

2.3.11 stat

stat *[profiler_area] [property]*

参数	可选	描述
profiler_area	**否**	类别索引，meta 命令生成一级索引
property	**否**	类别属性索引，meta 命令生成的二级索引

stat 在当前帧区间按照参数指标进行数学统计，给出 99.87% 置信区间的边界值，以及均值和标准差信息。

```
/> stat 0 1
[CPU][Scripts] mean=1874400.000±316545.565 range=[1582000, 2965000]
reasonable=[1582000, 2269000]
```

range 表示当前帧区间 *Scripts* 时间消耗的最小值和最大值，单位是纳秒 [1 毫秒 =1000000 纳秒]，*reasonable* 表示按照 3 倍标准差剔除极大值后的合理取值范围，超出该范围的值应该仔细检查，因为按照统计学在正态分布里面 3 倍标准差可以覆盖 99.87% 的数据。

2.3.12 seek

seek *[profiler_area] [property] [value] [predicate][=>]*

参数	可选	描述
profiler_area	**否**	类别索引，meta 命令生成一级索引
property	**否**	类别属性索引，meta 命令生成二级索引
value	**否**	临界值
predicate	是	> 大于临界值、= 等于临界值、< 小于临界值三种参数

seek 按照参数确定的指标进行所搜比对，默认列举大于临界值的帧信息，可以通过 *predicate* 选择大于、等于和小于比对方式进行过滤帧数据。

```
/> stat 0 1
[CPU][Scripts] mean=1874400.000±316545.565 range=[1582000, 2965000]
reasonable=[1582000, 2269000]
/> seek 0 1 2269000
[FRAME] index=10012 time=23.880ms fps=41.9 offset=128599965
```

调用该命令前建议先用 stat 对性能指标进行简单数学统计，然后根据最大值或者最小值搜索可能存在性能问题的渲染帧。

2.3.13 fps

fps *[value] [predicate][=>]*

参数	可选	描述
value	**否**	临界值
predicate	是	> 大于临界值、= 等于临界值、< 小于临界值三种参数

```
/> fps
frames=[20000, 20100)=100 fps=40.2±1.2 range=[39.2, 41.5] reasonable=[39.2, 41.2]
/> fps 41.2 >
[FRAME] index=20066 time=24.080ms fps=41.5 offset=261880027
```

当参数留空时，fps 统计当前帧区间的帧率信息，指定临界值后，则默认过滤大于临界值的帧数据，可以通过 *predicate* 选择大于、等于和小于比对方式进行过滤帧数据。

2.3.14 help

显示帮助信息。

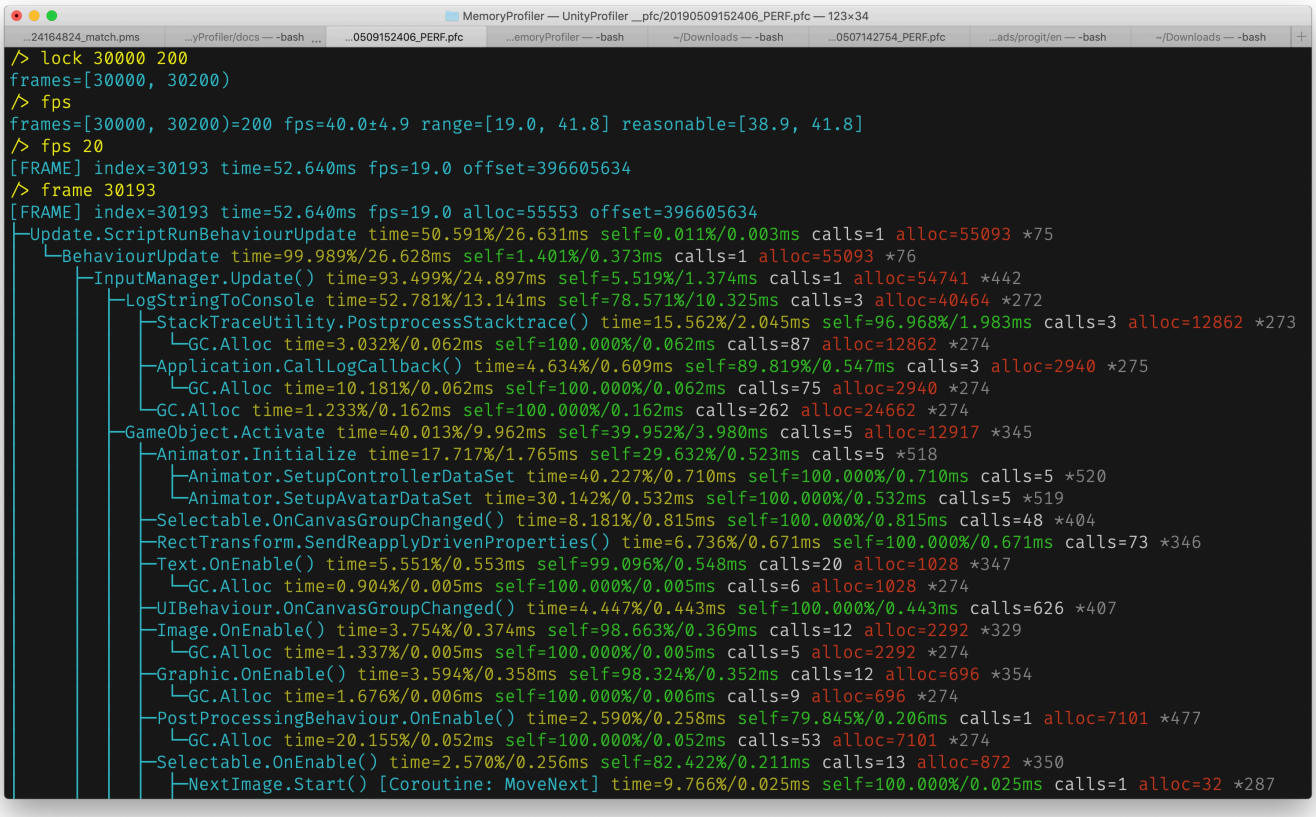
```
/> help
alloc [FRAME_OFFSET] [FRAME_COUNT] 搜索申请动态内存的帧
frame [FRAME_INDEX] 查看帧时间消耗详情
 func 按照方法名统计时间消耗
 find [FUNCTION_NAME_REF]* 按照方法名索引查找调用帧
 list [FRAME_OFFSET] [±FRAME_COUNT] [+⊢] 列举帧简报 支持排序(+按fps升序 -按fps降序)输出 默认
不排序
 next [STEP] 查看后STEP[=1]帧时间消耗详情
 prev [STEP] 查看前STEP[=1]帧时间消耗详情
 meta 查看性能指标参数
 lock [FRAME_INDEX] [FRAME_COUNT] 锁定帧范围
 stat [PROFILER_AREA] [PROPERTY] 统计性能指标
 seek [PROFILER_AREA] [PROPERTY] [VALUE] [>⊨|<] 搜索性能指标满足条件(>大于VALUE[默认] =等于VA
LUE <小于VALUE)的帧
 info 性能摘要
  fps [FPS] [>⊨|<] 搜索满足条件(>大于FPS =等于FPS <小于FPS[默认])的帧
 help 帮助
 quit 退出
```

2.3.15 quit

退出当前进程。

2.4 使用案例

2.4.1 追踪渲染丢帧



1. 使用 lock 锁定大概帧区间 $[30000, 30000+1000)$
2. 使用无参数 fps 对当前帧区间生成针对 *fps* 统计数据
3. 从数值区间 *range=[19.0, 41.8]* 可以看出，有渲染帧的 *fps* 低于阈值 20，调用 `fps 20` 把 *fps* 值低于 20 的所有帧列出来，从搜索结果可以看出第 **30193** 帧的 *fps* 等于 19.0
4. 使用 `frame 30193` 查看目标帧的详细数据，可以发现当前帧有内存申请开销，并且 *InputManager.Update()* 函数占用了将近 25 毫秒的时间，其中 *LogStringToConsole*(打印日志) 和 *GameObject.Activate* 占用了大部分时间开销，更多信息可以在自堆栈里面查看。

2.4.2 追踪动态内存分配

2.5 小结

3 MemoryCrawler

3.1 简介

3.2 原理

3.3 命令手册

3.3.1 read

3.3.2 load

3.3.3 track

3.3.4 str

3.3.5 ref

3.3.6 uref

3.3.7 REF

3.3.8 UREF

3.3.9 kref

3.3.10 ukref

3.3.11 KREF

3.3.12 UKREF

3.3.13 link

3.3.14 ulink

3.3.15 show

3.3.16 ushow

3.3.17 find

3.3.18 ufind

3.3.19 type

3.3.20 utype

3.3.21 stat

3.3.22 ustat

3.3.23 list

3.3.24 ulist