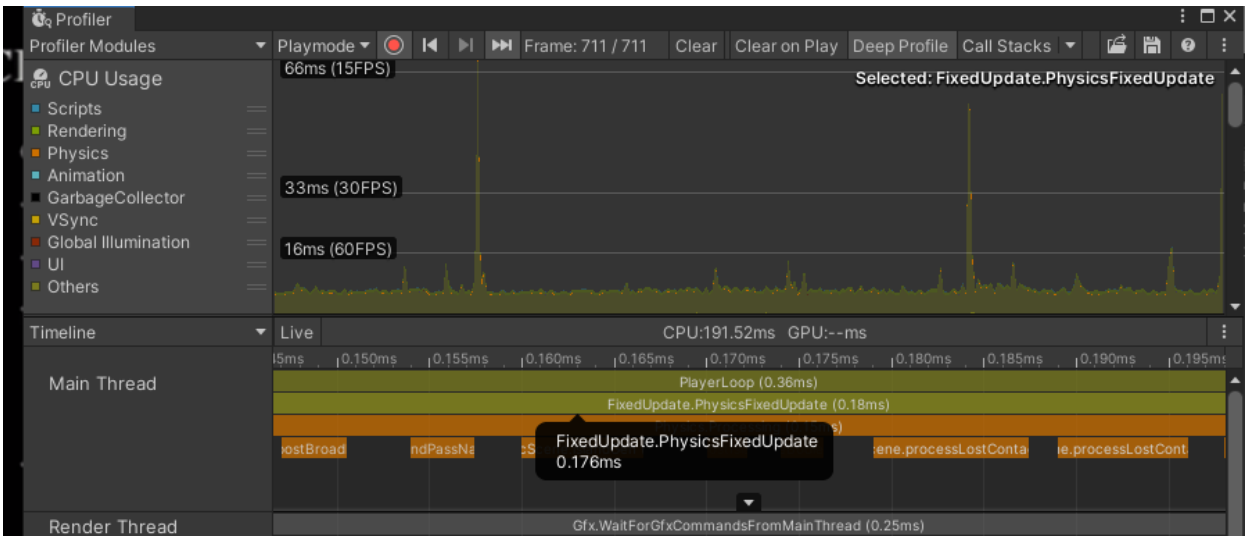
CPU

C# 不良的代码带来的GC Alloc



脚本

1.序列化和反序列化的时候会产生 一些GC（json\xml\protobuffer...）

2.string 字符串拼接，频繁的字符串拼接用StringBuilder

3.unity api带“s”的，返回数组的，GetComponents,Object.name,Sprite.vertices

4.Debug.Log , Warning(不注意的化 开销可能会很大)

5.Camera.main--->Object.FindWithtag("MainCamera"),应该在start函数里用FindWithTag缓存起来

6.Physics.RaycastAll------>Physics.RaycastNonAlloc(用Physics的话，记得加RaycastNonAlloc)

Physics.RaycastAll：投射一条光线并返回所有碰撞，也就是投射光线并返回一个RaycastHit[]结构体。

Physics2D.RaycastNonAlloc：　光线投射不分配 ：发射一定长度的射线，可以穿过碰撞器，返回遇到的碰撞器数目。

UGUI

Canvas Rebuild Mesh(重建)

GrapicRayCaster(接受射线)

动静分离（例子： canvas 有一个遥感，遥感是变化的，解决方式，遥感单独用一个Canvas）

Layout

SetDirty遍历会带来高消耗

RectTransform.Anchor代替LayoutGroup

缓存UI对象

图集

分类方式

　　按照功能模块区分

按照UI区分

减少一个UIPrefab引用的图集数量

图集不同影响合批批次

引擎特性

static Batching(静态合批)

内存换CPU性能

游戏运行时不要频繁调用StaticBatchingUtility.Combine

Dynamic Batching(动态合批)

动态批处理仅支持顶点数小于900的网格物体。

使用不同材质的实例化物体将会导致批处理失败。

拥有lightmap的物体不会进行批处理，除非他们指向lightmap的同一部分。

预制体的实例化会自动使用相同的网格模型和材质

drawcall并不是决定引擎渲染性能的关键

Batch数量不是唯一的因素 关注setPass的数量（材质，sharder发生改变之后，这个开销才是最大的）

GUI Instancing（树和草）的稳定性不是很好，大多数项目还是没有用到

动画用插件Animator Instancing进行优化（割草类游戏，观众等超多人使用）

https://zhuanlan.zhihu.com/p/36896547

关掉Animator？？？

MeshColloider的开销很大，尽量用BoxCollider，如果要用的化最好用简模

GPU

Rendering

顶点数量

适当的减面，剔除被遮挡部分

LOD技术

美术制作

减少零散的不见

合并材质和贴图

手机mmorpg纹理的内存占用量60~70左右

MipMap

提升采样的品质，降低命中的好事，IO速度

UI，例子特效（AlphaBlend）

控制半透明区域大小

控制面积

去掉纹理上多余的透明部分

Mesh.alpha=0依然被渲染

Terrian

分割多块地形

插件：Terrain Slicing&Dynamic Loading Kit

接缝：继承接口，解决黑缝

其他

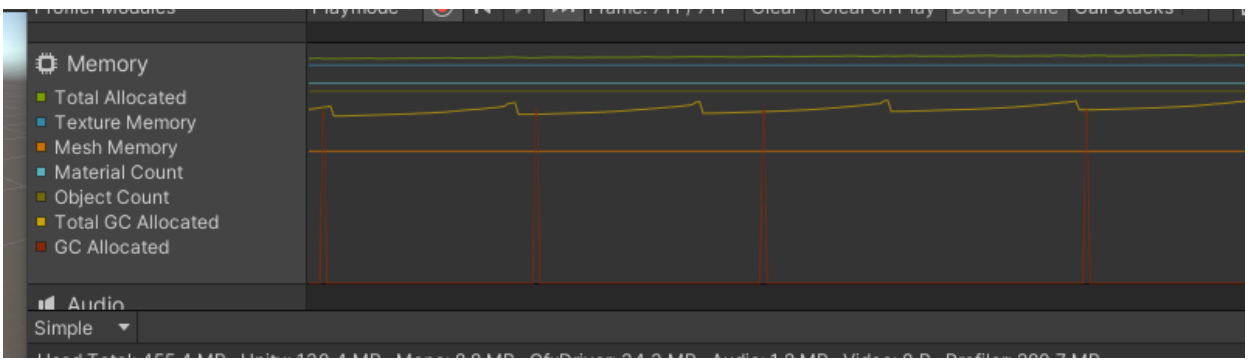
控制例子发射器的数量

控制例子发射数量Max Particles

控制Camera数量（camera的自身操作时很多的）

特效不一定都是通过ParticleSystem实现（mesh实现）

Memory



mesh 一般大型游戏的mesh占20~30m左右

重要的参数Read/WriteEnabled

meshCollider的用低模

动态加载mesh，MeshUploadMeshData是否释放CPU内存

Texture

60-70m

AnimationClip 4 0m

加载时的策略

一次性全部加载

分类加载

AssetBundle 2m以下

自行管理AB和Assets对象的引用计数

卸载主Bundle依赖与依赖Bundle

繁殖依赖资源的Assets出现记忆体泄漏