# 1.概念术语

## 内存区的类型

- 栈区,存放值类型,引用类型的指针,静态区对象的引用指针,常量区对象的引用指针
- 堆区,存放引用类型对象,可以进行动态分配
- 静态区及常量区,存放静态类,静态成员(静态方法,静态变量),常量的对象本身,由于存在栈内的引用地址,程序在运行时先入栈,因此静态区的常量区对象会持续到程序生命周期结束,届时,静态区和常量区的对象才会被回收,所以应当限制静态类,静态成员,静态变量
- 代码区 存放函数体内的二进制代码

## 装箱拆箱

装箱和拆箱在值类型和引用类型之间架起桥梁,使得值类型和引用类型之间可以互相转换,可以统一考量系统,值类型和引用类型都可以按照对象来处理,但是要尽量避免装箱和拆箱,因为这需要在堆中分配内存,效率低下

```
int i = 0;
System.Object obj = i;//(装箱)将值类型转化成引用类型对象
int j = (int)obj //(拆箱) 将引用类型转化成值类型
```

### 避免装箱拆箱(更耗费性能)

- 使用泛型
- 使用显式装箱来避免隐式装箱

```
//隐式装箱
int x = 0;
ArrayList arr = new ArrayList(3);
arr.Add(x); //1
arr.Add(x); //2
arr.Add(x); //3 //触发三次装箱

//显式装箱
int x = 0;
ArrayList arr = new ArrayList(3);
Object o = x; //1次
arr.Add(o); //不触发
arr.Add(o);
arr.Add(o);
```

# 2.Unity内存管理

## 引擎简述

Unity是一个C++引擎,并不是C#引擎,底层代码全部是由C++写的,除了一些Editor里面的Services可能会用到NodeJS这些网络的语言,Runtime里面用到的每行Unity底层代码都是C++的

Unity实际上分为三层:

- 最底层是Runtime,全是Native C++代码
- 最上层是C#,Unity本身也有一些C#,例如Unity的Editor是用C#写的,还有些Package也是C#写的
- 中间还有一层叫Binding,可以看见很多的.bindings.cs文件(基于C#的binding语言,一开始是Unity自定义的一种语言),这些文件的作用就是把C++和C#联系在一起,为C#层提供所有的API
  因此使用Unity时看见的C# API,都是在Binding层中自定义的 这些文件底层运行的时候还是C++,只是个Wrapper(封装)
  最早用户代码是运行在C#上,是MonoRuntime 现在可以通过IL2CPP将其转成C++代码,所以现在几乎没有纯正的C#在运行了
  Unity的VM(虚拟机:Virtual Machine)依旧还是存在,主要用于跨平台,有了一层VM抽象后,跨平台的工作会容易很多,IL2CPP本质也是VM

## 内存管理简介

Unity内存按照分配方式分为:Native Memory(原生内存)和Managed Memory(托管内存) Native Memory并不会被系统自动管理,需要手动去释放 而Managed Memory的内存管理是自动的,会通过GC来释放

此外Unity在Editor和Runtime下,内存的管理方式是不同的,除了内存大小不同,内存的分配时机以及分配方式也可能不同

例如Asset,在Runtime时,只有Load的时候才会进内存 而Editor模式下,只要打开Unity就会进内存(所以打开很慢) 后续有推出Asset Pipeline 2.0,一开始导入一些基本的Asset,剩下的Asset只有使用的时候才会导入

Unity按照内存管理方式分为:

- 引擎管理内存和用户管理内存
  引擎管理内存即引擎运行的时候分配的一些内存,例如很多的Manager和Singleton,这些内存开发者一般是碰触不到的 用户管理内存也就是开发者开发时使用到的内存,需要 重点注
- Untiy检测不到的内存
  即Unity Profilter无法检查到的内存,例如用户分配的Native内存 比如自己写的Native插件(C++插件)导入Unity,这部分Unity是检测不到的,因为Unity没法分析已编译的C++是如何分配和使用内存的 还有就是Lua,它完全自己管理的,Unity也没法统计到它内部的情况

# 3.Native Memory介绍

## Allocator与Memory Lable

Unity在里面重载了C++的所有分配内存的操作符,例如alloc,new等 每个操作符在被使用的时候要求有一个额外的参数就是Memory Lable,Profilter中查看Memory Detailed里的Name很多就是Memory Label 它指的就是当前的这一块内存内存要分配到哪个类型池里

## GetRuntimeMemory

Unity在底层会用Allocator,使用重载过的分配符分配内存的时候,会根据Memory Lable分配到不同的Allocator池里面 每个Allocator池,单独做自己的跟踪 当要在Runtime去Get一个Memory Lable下面池的时候,可以从对应的Allocator中取,可以从中知道有什么东西,有多少兆

## NewAsRoot

前面提到的Allocator的生成是使用NewAsRoot,生成一个所谓的Memory Island,它下面会有很多的子内存例如一个Shader,当加载一个shader进内存的时候,首先会生成一个shader的Root,也就是Memory Island然后Shader底下的数据,例如Subshader,Pass,Properties等,会作为该Root底下的成员,依次的分配 所以最后统计Runtime的内存时,统计这些Root即可

## 返还操作系统

因为是C++的,所以当delete或free一个内存的时候,会立刻返回给系统 这和托管内存堆不一样,需要GC后才返回

# 4.Managed Memory介绍

## VM内存池

即Mono虚拟机的内存池,内存以Block的形式管理,当一个Block连续6次GC没有被访问到,这块内存会被返回给系统,条件苛刻,比较难触发

## GC

## GC的机制考量

- Throughput(回收能力) 一次GC能收回多少内存
- Pause times(暂停时长) GC时对主线程的影响会多大(卡顿)
- Fragmentation(碎片化) 对整体内存池的碎片化影响多少
- Mutator overhead(额外消耗) GC时的消耗,GC时需要做很多的统计会产生消耗
- Scalability(可拓展性) 拓展到多核多线程会不会有什么

- bugPortability(可移植性) 在不同的平台上是否可以使用

### Boehm

Unity用的Boehm GC,简单粗暴,不分代

- Non-generational(非分代式),即全都堆在一起,因为这样会很快 分代的话就是例如大内存,小内存,超小内存分在不同的内存区域来进行管理(SGen GC的设计思想)
- Non-Compacting(非压缩式),即当有内存被释放的时候,这块区域就空着 而压缩式的会重新排布,填充空白区域,使内存紧密排布

上面的形式就会导致内存碎片化,可能当前的内存并不大的时候,添加一块较大内存时,却没有任何的一个空间放得下(即使整体的空间足够),导致内存扩充很多 因此建议先操作大内存,然后操作小内存

碎片化内存之间空出的内存可能就成为僵尸内存 这种情况实际上并不是内存泄露,因为这些内存并没有被泄露,泄露指这块内存没有任何人可以访问和管理,但实际上这块内存一直在内存池里

IL2CPP GC机制是Unity重新写的,属于一种升级版的Boehm

### Incremental GC

Incremental GC(渐进式GC)

主要解决主线程卡顿的问题,现在进行一次GC主线程被迫要停下来,遍历所有的Memory Island,决定哪些要被GC掉,会造成一定时间的主线程卡顿 Incremental GC把前面暂停主线程的事分帧做了,这样主线程不会出现峰值

## 5.栈和堆

Unity内存中重要的两部分,栈和堆,了解了堆和栈才知道应该如何优化内存,提高性能

### 栈

栈是内存中存储函数和值类型的地方

例如调用一个函数A,会将这个函数体与函数收到的参数放入到堆栈中,若在函数A中调用函数B,同样会把函数B存放到栈中 当函数B运行结束,会将其从栈中移除,然后当A运行结束,把A从栈中移除

因此在看Debug信息的时候,就会发现Log里面能够做到一层层的方法回溯,方便我们查看整体的调用过程,这也就是堆栈回溯

由于是栈的结构,因此不会遇到碎片化或是垃圾收集(GC)的问题 但是可能会碰见栈溢出的问题,比如调用了太多的函数导致一直push东西进栈,占据越来越多的内存空间,导致堆栈溢出

### 堆

堆是内存中另一个区域,要比栈大,所有的引用类型存放在这 创建新对象时会在堆中找到下一个足够存放的空位置存储 当销毁对象后,内存空间不会马上释放出来,而是标记成未使用,之后垃圾收集器会释放这部分空间

对象实例化和摧毁的过程其实很慢,所以要尽可能地避免在堆中分配内存的行为 如果需要的内存比之前已经分配好的还多,在放不下的情况下,堆会膨胀,并且每次都增长两倍,且不会再缩回去,过大的堆就会影响到游戏的性能 当在堆中释放了一些占用空间小的对象,而后添加一些占用空间大的对象时,由于前面释放的空间不足以存放下,就会导致这些空间空出来,使得内存的使用情况就变得不连续,即内存的碎片化,也会降低游戏性能

而前面所提到的GC就是在堆上进行的,每一次GC,都会遍历堆积上所有的对象,找到需要释放的东西,也就是没有被引用的对象,然后将其释放 但是有时候一些错误引用,导致一些希望释放掉的对象没有被GC掉,那么就会造成内存泄漏

假如游戏玩到一半,GC必须要释放数十或数百个游戏对象的内存,这会造成一个负载峰值,要尽量避免这样的负载峰值

## 6.优化 Native Memory

以下内容是和Native Memory相关的,使用不当可能导致Native Memory的增长,可以进行适当的设置和调整来进行性能优化

### Scene

导致Native Memory增长的原因,最常见的就是Scene 因为是C++引擎,所有的实体最终都会反映在C++上,而不会反映在托管堆上 所以当构建一个GameObject的时候,实际上在Unity的底层会构建一个或多个Object来存储这一个GameObject的信息(Component信息等) 所以当一个Scene里面有过多的GameObject存在的时候,Native Memory就会显著的上升,甚至可能导致内存溢出

注:当发现Native Memory大量上升时,可以先着重检查Scene

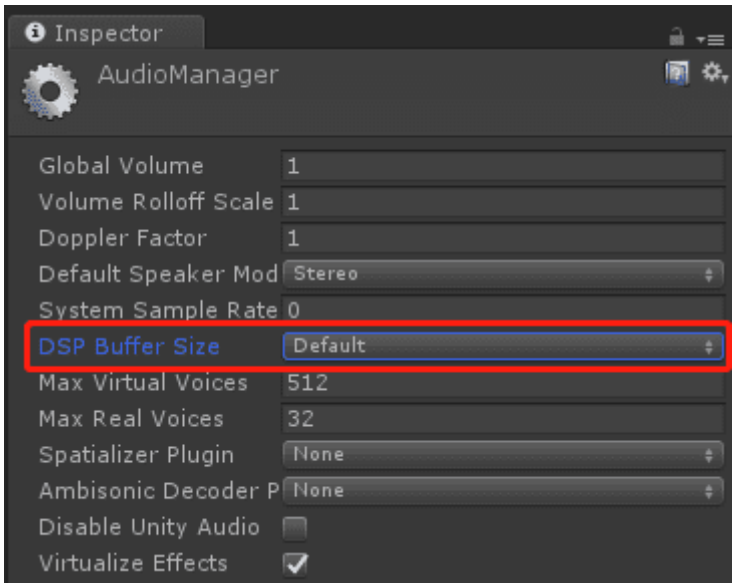
### Audio

### DSP Buffer:

DSP Buffer,是指一个声音的缓冲,当一个声音要播放的时候,需要向CPU去发送指令 如果声音的数据量非常的小,会造成频繁的向CPU发指令,造成IO压力 在Unity的FMOD声音引擎里面,一般会有一个Buffer,当Buffer填充满了才会去向CPU发送一次播放声音的指令

## DSP Buffer大小的设置一般会导致两种问题:

- 设置的值过大会导致声音的延迟,因为填充满需要很多的声音数据,当我们声音数据不大的时候,就会产生延时
- 设置的值太小会导致CPU负担上升,因为会频繁的发送




## DSP Buffer设置

- Force To Mono:这个选项作用是强制单声道,很多声音为了追求质量会设置成双声道,导致声音在包体和内存中,占用的空间加倍,但是95%以上的声音,两个声道是完全一样的数据 因此对声音不是很敏感的项目建议勾选此项,来降低内存的占用
- Compression Format:不同的平台有不同的声音格式的支持,IOS对MP3有硬件支持,Android暂时没有硬件支持 建议IOS适合使用ADPCM和MP3格式,Android适合使用Vorbis格式
- Load Type:决定声音在内存中的存在形态:
- Decompress On Load当audio clip被加载时,解压声音数据适用于小型音频文件(< 200kb)Compressed In Memory声音数据将以压缩的形式保存在内存当中适用于中型音频文件(>= 200kb)Streaming从磁盘读取声音数据适用于大型音频文件,例如背景音
- 注:例如Decompress On Load,要求文件必须小于200kb,因为内部内存管理的问题,如果是大于200kb的文件,那么也还是只会被分配到不足200kb的内存
- Bitrate:可以对音频文件本身进行压缩,降低文件的比特率(bitrate),前提音频品质不会被破坏太严重

## Code Size

代码也是占内存的,需要加载进内存执行 模板泛型的滥用,会影响到Code Size以及打包速度(IL2CPP编译速度,单一一个cpp文件编译的话没办法并行的) 例如一个模板函数有四五个不同的泛型参数(float,int,double等),最后展开一个cpp文件可能会很大 因为实际上C++编译的时候用的所有的Class,所有的Template最终都会被展开成静态类型 因此当模板函数有很多排列组合时,最后编译会得到所有的排列组合代码,导致文件很大

## AssetBundle

TypeTree:Unity前后有很多的版本,不同的版本中很多的类型可能会有数据结构的改变,为了做数据结构的兼容,会在生成数据类型序列化的时候,顺便生成一个叫TypeTree的东西 就是当前这个版本用到了哪些变量,它们对应的数据类型是什么,当进行反序列化的时候,根据TypeTree去做反序列化 如果上一个版本的类型在这个版本没有,那TypeTree里就没有它,所以不会去碰到它 如果有新的的TypeTree,但是在当前版本不存在的话,那要用它的默认值来序列化 从而保证了在不同版本之间不会序列化出错

在Build AssetBundle的时候,有开关可以关掉TypeTree

BuildAssetBundleOptions.DisableWriteTypeTree

当前AssetBundle的使用,和Build它的Unity的版本是一模一样的时候,就可以关闭 这样,一可以减少内存,二AssetBundle包大小会减少,三build和运行时会变快,因为不会去序列化和反序列化TypeTree
压缩方式(Lz4和Lzma):Unity主推Lz4(也就是ChunkBased,BuildAssetBundleOptions.ChunkBasedCompression),Lz4非常快,大概是Lzma的十倍左右,但平均压缩比例比Lzma差30%左右,即包体更大 Lz4的算法开源 Lzma基本可以不用了,因为Lzma解压和读取速度都非常慢,并且内存占比高,因为不是ChunkBased,而是Stream,也就是一次全解压出来 ChunkBased可以逐块解压,每次解压可以重用之前的内存,减少内存的峰值

大小和数量:AssetBundle分两部分,一部分是头(用于索引),一部分是实际的打包的数据部分 如果每个Asset都打成一个AssetBundle,那么可能头的部分比数据还大 官方建议一个AssetBundle,在1-2M,但是现在进入5g时代的话,可以适当加大,因为网络带宽更大了

## Resource

Resource文件夹里的内容被打进包的时候会做一个红黑树(R-B Tree)用做索引,即检索资源到底在什么位置 所以Resource越大,红黑树越大,它不可卸载,并在刚刚加载游戏的时候就会被一直加在内存里,极大的拖慢游戏的启动时间,因为红黑树没有分析和加载完,游戏是不会启动的,并造成持续的内存压力 所以建议不要使用Resource,使用AssetBundle

## Texture

- Upload Buffer:在Unity 的 Quality 里设置如图,和声音的Buffer类似,填满后向GPU push 一次
- Read/Write:没必要的话就关闭,正常情况,Texture读进内存解析完了搁到Upload Buffer里之后,内存里那部分就会delete掉 除非开了Read/Write,那就不会delete了,会在显存和内存里各一份 前面说过手机内存显存通用的,所以内存里会有两份
- Mip Maps:例如UI元素这类相对于相机Z轴的值不会有任何变化的纹理,关闭该选项
- Format:选择合适的Format,可减少占用的空间
- Alpha Source:对于不透明纹理,关闭其alpha通道
- Max Size:根据平台不同,纹理的Max Size设成该平台最小值

- POT:纹理的大小尽量为2的幂次方(POT),因为有些压缩格式可能不支持非2的幂次方的
- 合并:尽量将多张纹理合并成为大图
- 压缩:Android设备运行平台要求支持OpenGL ES 3.0的使用ETC2,RGB压缩为RGB Compressed ETC2 4bits,RGBA压缩为RGBA Compressed ETC2 8bits 需要兼容OpenGL ES 2.0的使用ETC,RGB压缩为RGB Compressed ETC 4bits,RGBA压缩为RGBA 16bits (压缩大小不能接受的情况下,压缩为2张RGB Compressed ETC 4bits)
- IOS设备运行平台要求支持OpenGL ES 3.0的使用ASTC,RGB压缩为RGB CompressedASTC 6x6 block,RGBA压缩为RGBA Compressed ASTC 4x4 block 对于法线贴图的压缩精度较高可以选择RGB CompressedASTC 5x5 block 需要兼容OpenGLES 2.0的使用PVRTC,RGB压缩为PVRTC 4bits,RGBA压缩为RGBA 16bits (压缩大小不能接受的情况下,压缩为2张RGB Compressed PVRTC 4bits)

参考链接:Unity贴图压缩格式设置

### Mesh

- Read/Write:同Texture,若开启,Unity会存储两份Mesh,导致运行时的内存用量变成两倍
- Compression:Mesh Compression是使用压缩算法,将Mesh数据进行压缩,结果是会减少占用硬盘的空间,但是在Runtime的时候会被解压为原始精度的数据,因此内存占用并不会减少 需要注意的是有些版本开了,实际解压之后内存占用大小会更严重
- Rig:如果没有使用动画,请关闭Rig,例如房子,石头这些
- Blendshapes:如果没有用到Blendshapes,也关闭
- Material设置:如果Material没有用到法向量和切线信息,关闭可以减少额外信息

### Assets

和整个的Asset管理有关系,Unity官网上有关于资源管理的文章 移动平台优化实用指南

## 7.优化 Managed Memory

### Destroy与null

用Destroy,别用null,显示的调用Destroy才能真正的销毁掉

### Class和Struct

根据具体使用情况选择Class或Struct 关于Class和Struct的用法和区别

### 减少装箱拆箱操作

例如LINQ和常量表达式以装箱的方式实现,String.Format()也常常会产生装箱操作等

## 对象池

虽然VM自己有内存池,但是我们还是需要自己使用内存池来管理

在游戏程序中,创建和销毁对象事很常见的操作,通常会通过 Instantiate 和 Destroy 方法来实现,如果频繁的进行这些操作,GC的时候会导致负载很重,因为会有大量的已摧毁对象的存在,不仅会造成CPU的负载峰值,还可能导致堆积碎片化 因此我们可以使用对象池来处理这类问题

使用对象池时需要注意,要决定对象池的大小,以及一开始要产生多少数量的对象在池中 因为如果你需要的对象数量多过池中现有的,就必须将对象池变大,扩的太大可能造成浪费,扩的小可能又造成频繁的添加

## 闭包和匿名函数

所有的匿名函数和闭包在C#编IL代码时都会被new成一个Class(匿名class),所以在里面所有函数,变量以及new的东西,都是要占内存的

协程

协程属于闭包和匿名函数的特例,游戏开始启动一个协程直到游戏结束才释放,错误的做法 因为协程只要没被释放,里面的所有变量,即使是局部变量(包括值类型),也都会在内存里 建议用的时候才生产一个协程,不用的时候就丢掉

## 配置表

一个游戏,策划往往会通过excel配置很多的配置表,然后在游戏中加载这些excel来读取其中的数据 但是如果excel数量非常的庞大,最好不要一下子全丢到内存里,建议分关加载等

## 单例

慎用单例,且不要什么都往里放,因为里面的变量会一直占用内存

## Scriptable Objects

假设有一个控制敌人的组件,名叫Enemy,代码如下:

```
public class Enemy : MonoBehaviour {
    public float maxSpeed;
    public float attackRadius;
}
```

这个组件挂载在每个敌人身上,但是其中这两个浮点数(maxSpeed 和 attachRadius)的数值都是不变的那么当场景中存在很多的敌人时,每次生成敌人的时候,这些数据就会重复一份

所以即使所有数据都一样,这两个浮点数还是重复的出现在有此脚本的对象上 所以建议改用Scriptable Objects,这样就只会耗费一组这样数据的内存,代码如下:

```
public class EnemyConfiguration : ScriptableObject {
    public float maxSpeed;
    public float attackRadius;
}
public class Enemy : MonoBehaviour {
    public EnemyConfiguration enemyConfiguration;
}
```

## 变量or属性

通常为了封装安全性,开发时会选择使用属性(getter/setter),而属性本质上是函数的调用,前面提到调用函数时,会在堆栈上分配内存,因此调用属性也是如此 当调用多次时,花费在堆栈中的时间就会增加 当然了,一般来说问题不大,但是如果在使用频繁的循环体中使用属性,可能就需要针对性的优化 可以通过宏命令进行处理,例如在开发时使用属性,发布版本时使用变量,如下:

```
#if DELELOPMENT_BUILD
    int m_health;
    public int health { get => m_health; }
#else
    public int health;
#endif
```

## 缓存一些Hash值

在我们想要在运行时修改动画或者材质的时候,可以使用下面方法来实现

```
animator.SetTrigger("Idle");
material.SetColor("Color", Color.white);
```

这类方法往往也可以通过索引来作为参数,使用字符串只是能显示的更加直观,但是当传递字符串时,程序内部会进行一些处理,频繁调用的话可能就会造成性能的消耗 因此我们可以先找到对应的索引,并将其缓存起来,供后续使用,如下:

```
int idleHash = Animator.StringToHash("Idle");
animator.SetTrigger(idleHash);
int colorId = Shader.PropertyToID("Color");
material.SetColor(colorId, Color.white);
```

### 缓存引用对象

在游戏中会经常查找一些对象,GameObject.Find与其他所有关联的方法,需要遍历所有内存中的游戏对象以及组件,因此在复杂场景中,效率会很低 GameObject.GetComponent,会查询所有附加到GameObject上的组件,组件越多,GetComponent的成本就越高 若使用的是GetComponentInChildren,随着查询变复杂,成本会更高

因此不要多次查询相同的对象或组件,而且查询一次后将其缓存起来,方便后续的使用

## 8.图形(Graphics)的一些优化建议

基本上当Unity渲染游戏图像时,会调用 draw call 来对GPU下指令,让场景能成功渲染 对象,材质和纹理越多,处理起来需要的时间也越多 所以过多的drawcall就会影响游戏的优化,这对于瓶颈在GPU上的游戏影响特别大,也就是我们的游戏已经给GPU太大的压力了

### 使用批处理

我们可以使用批处理来尽量减少drawcall,使用批处理需要满足一些情况,例如,要批处理的对象必须引用一样的材质,并使用相同的纹理(纹理合并在这就很重要),但是使用的模型可以不一样

动态批处理:可以减少对于移动对象的drawcall 只能用于少于900个顶点信息的情况,包含坐标,法线,uv0,uv1,切线 动态批处理每帧评估一次,由CPU负责

静态批处理:即对开启 static 标记的对象做批处理,在构建期完成 适用于绝大部分的静态Mesh,因此任何不会动的对象都应标记为静态的 如果我们在运行时要添加静态对象,可以看一下 StaticBatchUtility.Combine() 的API

有关SRP Batcher可以参考这篇文章:SRP Batcher,Draw Call优化,Shader SRP Batcher compatible

### Cast Shadows

默认情况下,MeshRenderder组件的Cast Shadows是开启的

阴影的渲染可以让游戏的光线增加真实度和深度感,但是某些情况下可能并不需要 在复杂场景中,可能会造成多余的阴影计算,阴影效果最后也看不见

因此若场景有的对象是否有阴影对整体效果没有影响的话,就关闭这个选项 不计算阴影可以省下CPU时间 (具体渲染步骤可以在 Frame Debugger的Shadows.Draw中查看)

### Light Culling Mask

在复杂场景中,许多光线紧靠彼此 根据渲染流程的设置,场景中越多的光照,性能可能就会越差 因此要确保光照只影响特定的对象层(例如专门给角色打光的光源,设置成只影响角色),尤其是多光源和多对象彼此

紧靠时

### 避免使用手机原生分辨率

现在的手机分辨率非常的高,在手机呈现高分辨率可能会影响性能和手机过热的问题 因为会有大量的计算需求,如后期处理 如果游戏本身很耗GPU,高分辨率会恶化这些问题 建议使
用 Screen.SetResolution 来降低游戏预设的解析设置(根据不同的设备来找到一些合适的值),来提高性能

## 9.UI的一些优化建议

UI的隐藏可以使用将其移到Canvas外的方法,而不是SetActive(false)的方法来隐藏

### UI的批处理

如果UI元素会改变数值或是位置,会影响批处理,导致向GPU发送更多的drawcall 因此建议:

- 将更新频率不同的UI放在不同的Canvas上
- 相同Canvas中的UI元素的Z值要相同,这样才不会打断批处理
- 相同Canvas中的UI元素要使用相同的材质和纹理,材质或着色器可以有动态变换(例如一些特效),这不会影响批处理
- 相同Canvas中的UI元素要使用相同裁剪矩阵

### Graphic Raycaster

该组件是用来处理输入事件,默认挂载在每个Canvas上 有时不能互动的对象仍是canvas中的一部分,并附带了该组件,所以当每次鼠标或触控点击时,系统就要遍历所有可能接受输入事件的UI元素,就会造成多次的 "点落在矩形中" 的检查,来判断对象是否该作出反应 在UI很复杂的情况下,这个运算成本就会很高 因此建议确保只有可互动的Canvas才有该组件,节省CPU运行时间

### 全屏UI的处理

游戏中可能会有些全屏UI(例如一些设置界面),会遮挡住场景物体或其他UI元素 然而它们即使被遮挡看不见,CPU和GPU还是会有消耗,因此建议:

- 3D场景完全被遮挡的话,关闭渲染3D场景的摄像机
- 被遮蔽的UI,Disable这些Canvas,注意不是SetActive(false)
- 尽可能的降低帧率,因为这些UI一般不需要频繁刷新

# 10.其他优化

## GameObject的层次结构

某些情况下,场景中的物体可能有很深的嵌套结构,对父节点的GameObject进行坐标转换时,就会产生OnTransformChanged事件,这消息会传递给该GameObject下所有子对象,即使这些对象没有任何渲染组件(也就是我们看不见任何变化),造成一些不必要的转换运算,包括平移,旋转和缩放

此外,较深的结构也会导致在GC时,花费更多的时间在层级结构间遍历

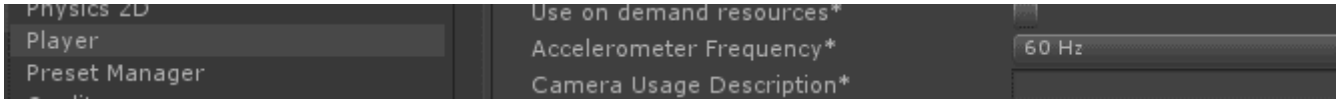
## 避免在Awake和Start中添加大量的逻辑

这对游戏启动很重要,Unity会在Awake和Start方法执行后渲染第一个画面,某些情况可能会导致启动画面或是载入画面需要花更长的时间渲染,因为必须等每个游戏对象都完成Awake和Start的执行 同时若游戏启动时,黑屏太久,提包时可能会被退审

## 删除空的Unity事件

Monobehaviour中的Start,Update这些方法即使是空的,也会带来些微的性能消耗,因此若为空,就删除它们

## Accelerometer Frequency




这个设置在Project Settings->Player->IOS->Other Settings中,这个功能定义Unity从设备读取加速度仪信息的频率,在不需要加速仪的游戏中,将它启动或设置了高于需求的频率,会影响性能表现 因为读取硬件设备信息,会增加CPU的处理时间

## 移动物体

Unity中有许多移动游戏对象的方法,例如 transform.Translate,如果对象需要碰撞判定,需要添加刚体和碰撞体,如果还是使用 transform.Translate 方法,会造成物理引擎整体重新计算,对于复杂的场景,成本可能很高 因此若要移动带有刚体的对象,使用rigidBody.MovePosition,并且要在FixedUpdate方法中执行 建议使用transform.Translate就在Update中执行,使用rigidBody.MovePosition或AddForce方法在FixedUpdate中执行

## 添加组件

在运行时调用AddComponent其实很没效率,尤其在一帧中多次启用这类调用

当添加一个组件的时候,Unity会做下列操作:

- 先看组件有没有DisallowMultipleComponent的设置,如果有,就要去检查是否有同类型的组件已加入
- 然后检查RequireComponent设置是否存在,如果设置了,就代表这个组件需要别的组件同步加入(重复做添加组件的操作)
- 最后调用所有被加入的MonoBehaviour的Awake方法

上述这些步骤都发生在堆上,所以可能会影响性能和增加GC的处理时间

## 数据结构

也就是Array,List和Dictionary等,例如在Array或List中使用索引的成本很低,那么就适合要经常通过索引读取的情况 而要频繁增加和移除对象时,使用Dictionary是最合适的