内存：

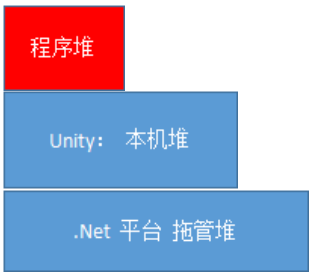
1，静态区　只实例化一次

2，堆区　new出来的静态区（全局区）：静态变量和全局变量的存储区域是一起的，一旦静态区的内存被分配，静态区的内存直到程序全部结束之后才会被释放　引用类型

3，栈区　参数 声明的　值类型的

4，代码区

内存结构：



程序堆：

1，程序员写的代码

2，Dll

优化：

System.data　system.sqlite　system.xml.

IOS:　17M+　android :　10M+

1，　选.NET 2.0　Subset

2，　Stripping Level”表示从build的库中剥离的力度　每一个剥离选项都将从打包好的库中去掉一部分内容你需要保证你的代码没有用到这部分被剥离的功能，选为“Use micro mscorlib”的话将使用最小的库

3，

自己写的代码：

1,string　尽量　stringbuilder　　string每加一次就会重新分配一块内存 释放之前那块内存，产生垃圾　造成内存浪费||||字符串相加减stringBulider相当于一个list 进行字符串相加减
字符串加减法的话就用stringbuilder
2,foreach　　for
每一次foreach的时候就会new一个指针出来 指向下一个 foreach的遍历的时候会释放很多的垃圾

本机堆：
Unity 所占的内存。

Unity引擎进行申请和操作的地方，比如贴图，音效，关卡数据等

- 资源：纹理、网格、音频等等
- GameObject和各种组件。
- 引擎内部逻辑需要的内存：渲染器，物理系统，粒子系统等等

1，　当你加载完成一个Unity的scene的时候，scene中的所有用到的asset（包括Hierarchy中所有GameObject上以及脚本中赋值了的的材质，贴图，动画，声音等素材），都会被自动加载（这正是Unity的智能之处）。
2，　本中对资源的引用。大部分脚本将在场景转换时随之失效并被回收，但是，在场景之间被保持的脚本不在此列（通常情况是被附着在DontDestroyOnLoad的GameObject上了）。而这些脚本很可能含有对其他物体的Component或者资源的引用，这样相关的资源就都得不到释放

优化：

1，尽量减少在Hierarchy对资源的直接引用，而是使用Resource.Load的方法，在需要的时候从硬盘中读取资源，在使用后用Resource.UnloadAsset()和Resources.UnloadUnusedAssets()尽快将其卸载掉

2，尽量减少代码的耦合和对其他脚本的依赖是十分有必要的

托管堆：

.NET 平台 托管对象 。 最后引用计数　没有引用了　GC 释放 。

```
GameObject    tmpObj =      new GameObject();

tmpObj = null ;
```

优化：

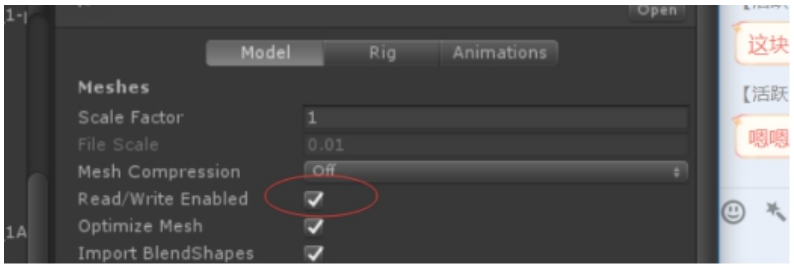
1， 减少对GameObject的Instantiate()和Destroy()调用

2， 在合适的时候，也可以手动地调用System.GC.Collect()来建议系统进行一次垃圾回收

3， 多用 缓存池

Fbx：

1,是一个我们做这个模型导入要使用FBX文件，我们见到很多项目它的模型导入会把Read/Write Enable打开，这里强调一下，这个设置打开一下会导致内存翻倍。

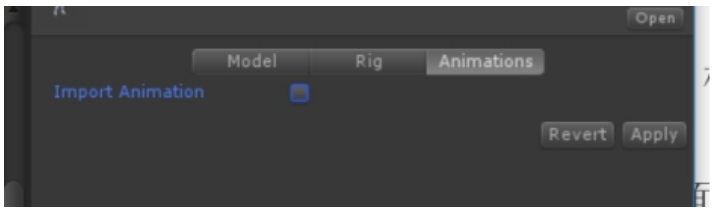


2， 模型导入有三个选项卡，其中一个是动画相关的选项卡，我们默认不是None，即使FBX里面没有动画数据的话，也会默认添加一个Animator的组件，会增加了内存的占用，建议大家检查一下导入设置。

压缩时间线 和精度 ：

https://blog.csdn.net/alexander_xfl/article/details/66975570

动画系统时间线：



3， NormalMap:

我们FBX导入的时候，如果不使用一些Normalmap等特性，你如果开了Normal Tangent导入也都是浪费了内存空间，我们一般都是在项目开始初期决定用不用Normal Map，如果不用的话以后可以把这个关闭掉

数量规格：
顶点数量： 手机 5万 流畅 3--4万
NPC : 700 主角色： 2000-3000
300 1500*10 = 1.5万
300*6*3*2
36*3= 1万多

动态物体
a.控制面片数量 300-2000面片
b.控制Skinned Mesh Render 为1个
c.控制材质数量为1-3个
d.控制骨骼数量 小于30根
静态物体
a.控制网格顶点数量 小于500个
b.标记为Static
c.Static Batching
d.Animation组件：如果不需要，尽量不要附加
研究课题： 网格合并 。减少DC 。
共用一个材质球 ： 顶点数组1 顶点数组2

GameObject:

gameObject.tag == “Player”

1，gameObject.tag 会在内部循环调用对象分配的标签属性以及拷贝额外的内存，推荐使用gameObject.CompareTag("XXX")来代替.tag。

2，使用ObjectPool对象池来管理对象，避免频繁的Instance，Destroy。

Audio:

1,格式：
对于音频设置方面，我们建议iOS使用**mp3**，安卓采用**vorbis**

2,采样率 ：
然后还有音频的一个采用率，一般采用率20k就够了。 手机 8K 够了
3,声道：
音频文件还要考虑是单声道还是双声道，一般情况下用单声道效果是足够的，我们把它压缩成单声道也可以减少内存的占用。

4,Load:

我们建议对这些小的音频文件采用Decompress On Load这种模式，因为对小的音频文件的话，你在加载的时候解压缩一次，这样播放的时候不需要重新解压缩，如果每次解压缩会导致手机发烫、续航减少等等问题

Shader:

1，做了Shader编译，我们引擎提供一个功能是Shader.WarmupAllShaders，推荐的使用方式是游戏场景加载完之后，可以调用这个API，把你场景当中引用的Shader帮你预先编译一下，这样游戏过程当中再使用的时候就不会再去调用编译的操作，这样就避免你游戏过程当中产生Shader编译导致的卡顿。

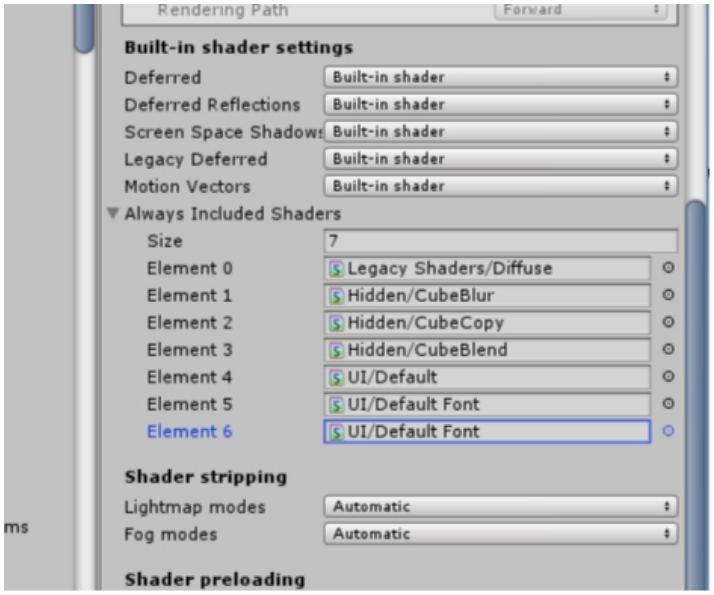
2,精度 调低 Float4 à 换成 fixed4 float : 32 half: 16 fixed: 12

3,少使用的函数：pow,sin,cos,导数 对数等

4, 只使用mobile组里面的那些预置shader

AssetBundle shader :

打包 android shader 丢失 ：



Resources ：

现在Unity官方不再推荐大家使用Resources文件夹，我们未来可能把这个功能关闭掉也是有可能的，因为你把资源放在这里面的话，它是有几个缺点的，第一个缺点你游戏启动的时候，比如说我们手机上面启动这个游戏，它第一步会把Resources文件夹内的文件构建一个索引，这样你后面再动态加载资源的时候，他可以在这里面查找这个文件，这样做导致游戏启动比较耗时，会发现启动黑屏的时间很长，这样对玩家来说这样的体验不太好。

另外，构建索引会占用你更多系统的内存资源。所以说我们建议以后大家不要使用Resources文件夹

GUI:

首先，OnGUI是**Unity**老的**UI**系统，现在不建议使用**OnGUI**。另外就是第三方的插件NGUI，也是有它的不足。因为它使用C#开发，会导致堆栈的内存分配，运行时会导致内存的操作。

RenderTexture:

我们还比较常见的一个问题就是现在手机游戏对画面要求越来越高了，过去两年前很少见到手机游戏用后期特效，现在很多游戏把后期特效打开，后期特效其实有多个问题，首先它的GPU消耗比较高，另外一个消耗是会导致你的内存消耗非常大，因为每个特效都有可能分配你一个全屏幕大小的一个RenderTexture。而RenderTexture因为要用来实时渲染，所以是不能压缩，不像普通的游戏纹理可以使用ETC或者PVRTC格式压缩。所以它对内存的占用非常高，我们经常看到有的游戏后期特效分配了几十兆甚至更高的内存。

图片：

Png ： rgba

JPG: rgb

1,alpha

我们发现过很多次正常不应该出现的情况，就是有些不需要Alpha通道的纹理在导入时保留了Alpha通道。我们建议大家都检查一下，把这些不必要的Alpha通道都关闭掉。

2,格式:

Android:

Android设备就没有那么好的运气了。由于硬件平台不统一，每个厂商的GPU可能使用不同的纹理压缩格式。所以还是老老实实用PNG比较好。

Etc 1: 没有透明通道的 图片 。兼容所有的机器。

ETC2 : 带有透明通道。 高端机 。

IOS:

当然是用pvr格式。

pvr是iOS设备的图形芯片 PowerVR 图形 支持的专用压缩纹理格式。它在PowerVR图形芯片中效率极高，占用显存也小。

性能对比可以看这里：In Depth iOS & Cocos2D Performance Analysis with Test Project。

物理：

1, 真实的物理（刚体）很消耗，不要轻易使用，尽量使用自己的代码模仿假的物理

2.对于投射物不要使用真实物理的碰撞和刚体，用自己的代码处理
3.不要使用mesh collider
4.在edit->project setting->time中调大FixedTimestep（真实物理的帧率）来减少cpu损耗

FixedUpdate()

C#:

命名：

```
Class     Msg   ;


Msg   msgDict ;
```

1,尽量减少函数调用栈，用x = (x > 0 ? x : -x);代替x = Mathf.Abs(x)

2,Struct 赋值 ：

Unity里的Struct的赋值是通过String.memcpy实现的。优化可以考虑把每个变量单独赋值，而不是使用Struct的整体赋值，它的性能有可能提升，建议测试对比一下性能有没有变化，再选择使用哪一种方式。

```
Struct Person
{
    Int age ;
  Float weight ;
}
Person tmpPeroson ;
tmpPerson.age=10;
```

a.对于有的函数，则可以每个几帧 执行一次
b.通过使用InvokeRepeating函数实现定时重复调用，比如，启动0.5s后每隔1s执行一次DoSomeThing
c.尽量少使用临时变量，特别是在Update() OnGUI()等实时调用的函数中
d.定时进行垃圾回收
e.优化数学运算，尽量避免使用float，而用int，特别是在手机游戏中，尽量少用复杂的数学函数，比如sin，cos等函数。改除法为乘法，比如0.5f*a而不是a/2.0f

编译：

如system.data.dll，因为Unity在5.5升级了C#的编译器，编译检查会比过去更严格。你可以检查一下，重新拷贝这个文件，或者更新一下这个dll文件。

游戏标准：

1. 首先是安卓2g低端机型要流畅运行在20到25fps，内存不超过350mb
2. IOS1g低端机型流畅运行20到25fps，内存不超过300mb
3. 国内游很多1g安卓。还有场景加载速度，启动过场场景速度尽量控制在5秒以内。还有优化是对性能和效果的权衡，这点是非常重要的，不能只考虑性能而不讲究美术效果，或者反过来只追求美术效果，玩都不能玩的话，游戏也是不成功的。

垂直同步：

GPU--> 显示器

垂直同步其实是这么回事：
电脑的每显示一张画面都分为两个过程完成，一个是cpu和显卡把所要显示的数据计算出来，另一个是显示器把这些数据写入到屏幕上去。这两步都是需要时间的，并且两个过程可以同时完成（因为具体实现这两个过程的硬件是相对独立的）。
但往往两个过程所花时间不一样，比如每幅画面的第一个过程只需要5毫秒，而第二个过程需要20毫秒，这个时候就有个问题：如果新的一幅图片已经计算完成了，但显示器中前一幅画面还没有画完该怎么办？
一种方法是是让cpu和显卡先等着，等显示器把之前一幅画画完再说（这就是开启垂直同步），很显然这样帧数就取决于显示器最快能画多快了（液晶一般是每秒60帧），而cpu、显卡再好帧数也不能提升，它们只能干等着；
另一种方法是不管显示器画完没，显卡强制用新数据覆盖掉旧的，这样帧数就提升了（也就是关闭垂直同步），不过这只是表面的，你最终看到的是显示器屏幕上的内容，实际帧数还是被显示频率限制死了，多出来的那些画面等于cpu和显卡白算了，这些多运算出来的帧可能还没来得及输出到显示器就又被更新了。

所以如果你显卡和CPU很好的话，游戏帧速80+，最好打开垂直同步，这样画面更平稳，而且可以省下一下运算资源应对突发大场景，预防帧速剧烈波动。

如果帧速低于60+，大家可以考虑关掉垂直同步——这时候不存在显卡等显示器的问题，而毕竟垂直同步多了一些判断过程，关掉可以略微提升帧数。

IOS 联机查看性能:

Instruments：

http://www.cocoachina.com/ios/20150225/11163.html

android 联机查看：

查看 log:

```
color f4 rem
adb logcat | findstr "Unity"
```

查看 profiler 参数说明 ：

http://www.cnblogs.com/zhaoqingqing/p/5059479.html

android 连接真机看：

http://blog.sina.com.cn/s/blog_5b6cb9500101ehz0.html

人员配置

RPG 游戏 半年
6 人左右

MMORPG 一年 半左右

前端： 10 人 后端： 5--10 策划： 10 人 美术 ： 20人

棋牌

3—4 人 半年

Lua c# 通信原理： 栈 。

C# --> Lua： Lua 表 key value

Lua--> C# : C#： 类

Int

age ;

Bool

sex ;

Void * 任意类型的指针。

Person tmpPerson = Void *

tmpPerson.age =10

Lua 遇到的坑：

1、不支持continue，因为没有这个关键字，有时候写代码有些麻烦
3、一个较大的数调用tostring()之后，再调用tonumber()，得到的值不是原来的值了。从代码可以看出只能容下14位小数位，所以有时候数字会被截断变小。
#define LUA_NUMBER_FMT "%.14g"
#define lua_number2str(s,n) sprintf((s), LUA_NUMBER_FMT, (n))

4、函数调用层次、参数个数等等经常会有一个最大数的限制，如果不小心超过了。。。。。
TextAsset
5、LUA的TABLE分为数组和HASH两个部分，很多的库函数是针对把TABLE当作严格的数组时起作用的。如果误用的话，结果通常是不准确的。 len # for 循环 代替
6、LUA 5.1的变量默认是全局的，很容易在函数中的变量没有加上local而覆盖全局的变量，而导致全局数据变化，逻辑不正确。 local tmp
7、LUA代码执行如果出现错误会停止执行函数后面的代码，会导致需要严格控制的流程只执行了一部分，产生逻辑错误。例如在网游中购买商品的过程中，如果先给物品，再扣钱。代码在给完物品后出现错误，就不会执行扣除金钱的操作，导致可在无限地利用这个漏洞刷物品。 对于这类情况，一般是先检测金钱数是否够，条件是否满足，先扣钱，再给物品，并记录下日志。这样如果出现错误，玩家损失了金钱，但是可以找客服进行解决。

http://blog.csdn.net/xoyojank/article/details/12762909

Lua 内存泄漏：

弱引用：
https://www.cnblogs.com/chenny7/p/4050259.html

Lua 内存泄漏：
https://blog.csdn.net/yxtxiaotian/article/details/51475856

Lua:

myTable ={}

myTable.obj = GameObject.Find(“”)

C#:

Destroy()

Lua:
myTable.obj = nil

解决内存泄漏问题：

弱表。

table的弱引用类型是通过其元表中的__mode字段来决定的。这个字段的值应为一个字符串：
如果包含'k'，那么这个table的key是弱引用的；
如果包含'v'，那么这个table的value是弱引用的；

```
a = {1,4, name='cq'}
setmetatable(a, {__mode='kv'})
```

PB 原理：

https://blog.csdn.net/carson_ho/article/details/70568606

Int   age = 4 ;

Byte : age= 4
Short   age = 513 ;

0000 0000　　0000 0000　　0000 0000　　0000 0100

0000 0000
&
1111 1111

0000 0000

SVN 用法：保证每个成员 每台机器上的代码 资源 都是一致的。

Server:　　专门一台机器 。

Commit update update

第一次 由主程 上传 框架 到代码仓库 。

Assets:
ProjectSetting:

成员管理：
1，创建人员 设置用户名 和密码
2，创建组 。

创建代码仓库：
1，设置访问成员

https://192.168.109.1:8443/svn/TestTwo/

客户端： 第一次从服务器上拿东西 CheckOut .

增 ：

1,add
2,commit

其它的用户：
1,update

删：
1， Svn delete
2， commit

改：

1,Update
2,Commit

遇到冲突 ： 保留别人 备份自己的 最后修改完了 提交代码。

不要两个人 操作同一个东西 。

回滚版本：

1，首先找到对应的版本
2，将版本里面的代码 备份 变成新的资源
3，提交 。

SVN 被锁住的时候：

Clean up

SVN :下载地址
链接：https://pan.baidu.com/s/1efJTSujh6f7W12nNXzRVLA
提取码：qze0
复制这段内容后打开百度网盘手机App，操作更方便哦