Unity 项目优化

绪论

在讲优化之前，我希望大家明白几个概念，优化主要是 CPU,GPU,内存的优化，优化方式很多，有些人一提到优化就想到了“Draw Calls”，确实，这是优化的很大的一个点，但是还有很多其他的内容可以进行优化。不要认为“Draw Calls”是全部了，几个重要的概念性知识如下：

Draw Calls：其实就是对底层图形程序（比如：OpenGL ES)接口的调用，以在屏幕上画出东西。所以，是谁去调用这些接口呢？CPU。

Batching：批处理，都知道批处理是干嘛的吧？没错，将批处理之前需要很多次调用（draw calls）的物体合并，之后只需要调用一次底层图形程序的接口就行。听上去这简直就是优化的终极方案啊！但是，理想是美好的，世界是残酷的，一些不足之后我们再细聊

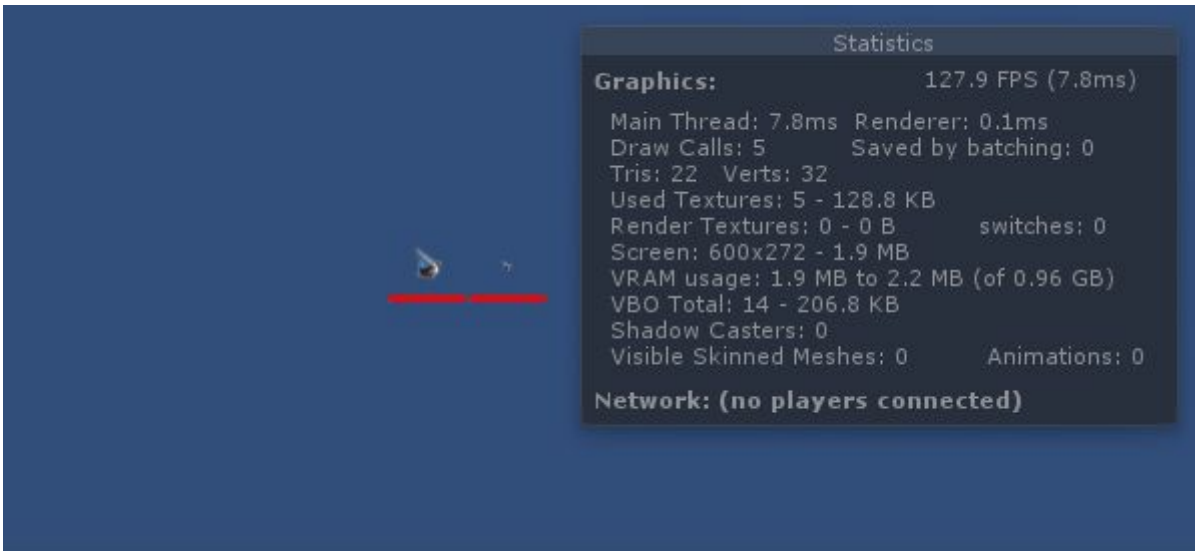
FPS：是图像领域中的定义，是指画面每秒传输帧数，通俗来讲就是指动画或视频的画面数。FPS 是测量用于保存、显示动态视频的信息数量。每秒钟帧数 愈多，所显示的动作就会愈流畅。

1 批处理

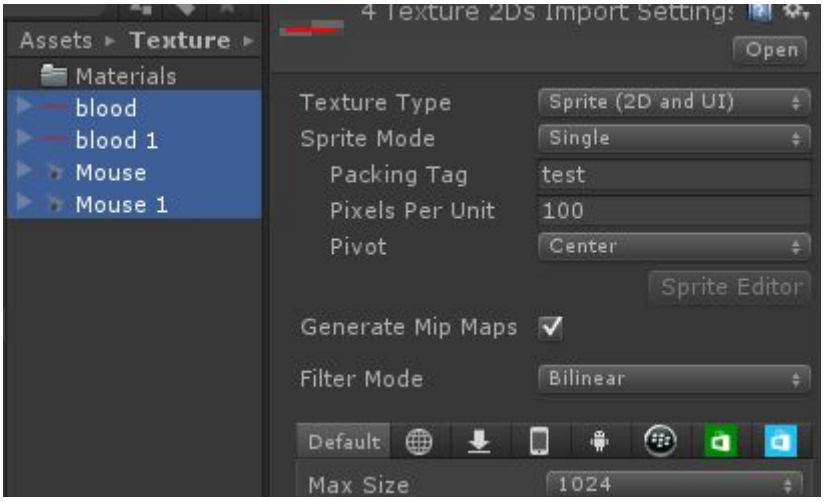
1.1 Unity 中 UGUI 打包图集

大家都知道，Unity 从 4.6 之后的版本都是自带 UGUI 的，非常的好用，之前我们用 NGUI 时有一个打包图集的功能，在 NGUI 中如果不对图集进行打包的话是不能在 NGUI 中使用的，现在的 UGUI 就没有这个限制，其实在 UGUI 中也是有图集的，打包图集，把几张小图合成一张大图，注意：大图尽量不要超过 1024。UGUI 不对图集进行打包也可以使用，为什么要打包图集呢？下面我们来看下打包图集对我们的好处。

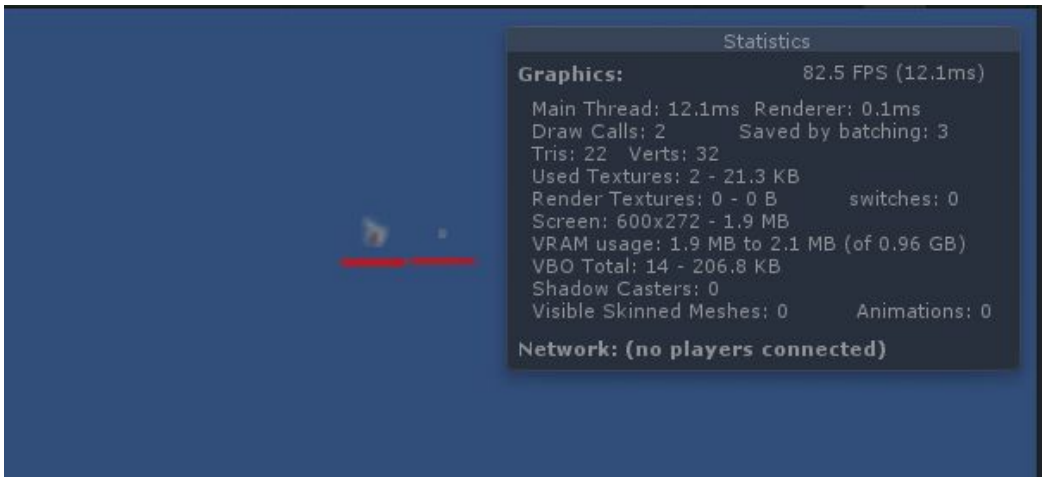
我们有四张小图，全部改成格式为 Sprite（2D and UI）格式，拖进层次视图中，我们看到现在的 Draw Calls 的数量是 5，如下图所示。



场景中没有任何对象的时候，Draw Calls 是 1，加了四张图片，Draw Calls 增加了 4。好，下面开始打包，首先在 Edit - Project Settings - Editor 中找到 Sprite Packer，里面改为“Always Enabled”，这个是调节，是否可以进行打包图集的选项。之后选中想要打包到一起的所有小图，在检视面板中有“Packing Tag”栏，都改成一个自定义的图集名字，并点击 Apply 实施，如下图所示。



这样我们的图集就打包成功了，在菜单栏里面 Window - Sprite Packer 就可以看到我们的图集了，若没有，点击左上角的“Pack”更新一下。之后我们再次运行原来的场景，发现 Draw Call 变成了 2 了，如下图所示。



本来四张图是分开渲染的，现在打包成图集就变成了一次渲染，所以 Draw Calls 从 5 变成了 2。由小及大，这是本文阐述的第一个优化 Draw Calls 的方法。

1.2 静态批处理

何谓静态，就是我们层次视图中那些不动的静态的游戏对象，比如说山，树，石头等，相信大家并不陌生，废话不多说，建立个新场景，里面创建一个 Cube，一个 Sphere，一个 Capsule，这样，场景的 Draw Calls 为 4，是本有的 1 加上我们的三个对象，我们再建立一个 Quad，一个 Cylinder，发现 Draw Call 还是 4，经过测试，发现 Cube，Cylinder，Quad 三者在相同材质下，调用一个接口就可以了，这不是重点。现在的效果如下图

图 1.4

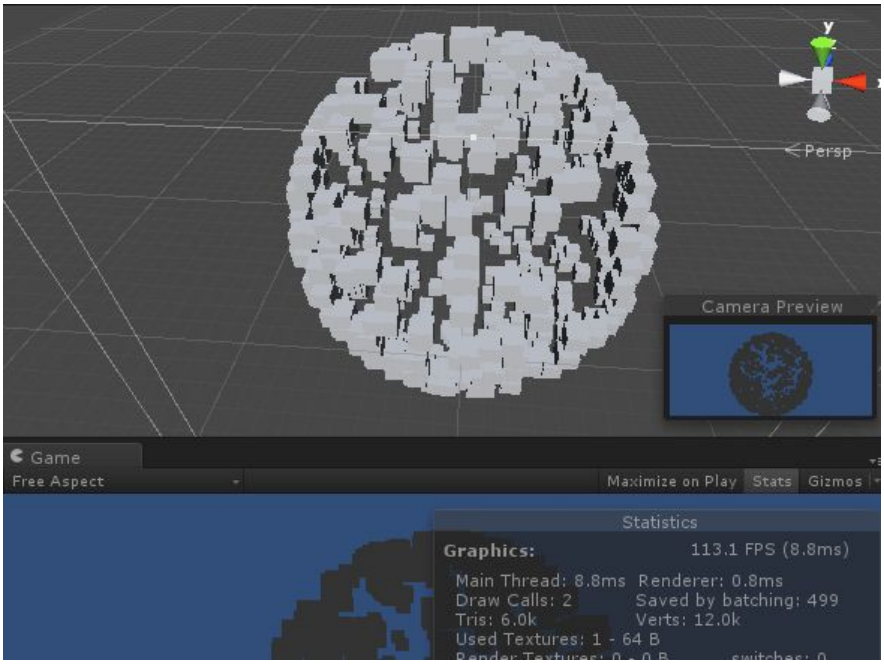
现在我们选中所有物体，在检视面板的右上方，把“Static”打上勾。再次运行场景，发现 Draw Calls 减少为 2，Saved by batching 变成了 4，就是说我们 5 个相同材质的静态物体，可以只调用一次图形接口，注意：必须是同材质。这就是静态批处理的优化效果。

1.3 动态批处理

新建场景，新建脚本“DynamicBatching”写出生成 500 个 Cube 的脚本，如下：

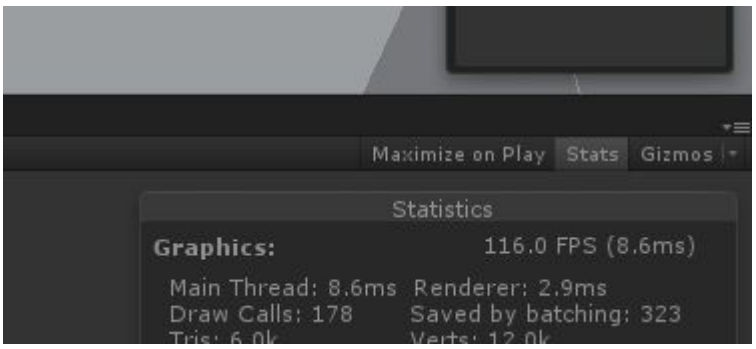
```
for (int i = 0; i < 500; i++)
{
    GameObject cube01 = Instantiate(cube, 10 * Random.onUnitSphere, Quaternion.identity) as GameObject;
}
```

在一个球体范围边上随机生成 200 个 Cube。效果如下图：



所有的 Cube 只调用了一次图形接口，批处理了 499 个。但是，实际上批处理有很多限制，稍不注意就是 Draw Calls 的飞涨，例如，脚本改成和运行效果如下：

```
for (int i = 0; i < 500; i++)
{
    GameObject cube01 = Instantiate(cube, 10 * Random.onUnitSphere, Quaternion.identity) as GameObject;
    if (i < 100)
    {
        cube01.transform.localScale = new Vector3(1 + i, 1 + i, 1 + i);
    }
}
```



这样发现，若是大小不同，是不会批处理的，Draw Calls 涨的飞速。具体动态批处理的限制如下：

1. 批处理动态物体需要在每个顶点上进行一定的开销，所以动态批处理仅支持小于 900 顶点的网格物体。
2. 如果你的着色器使用顶点位置，法线和 UV 值三种属性，那么你只能批处理 300 顶点以下的物体；如果你的着色器需要使用顶点位置，法线，UV0，UV1 和切向量，那你只能批处理 180 顶点以下的物体。
3. 不要使用缩放。分别拥有缩放大小(1,1,1) 和(2,2,2)的两个物体将不会进行批处理。
4. 统一缩放的物体不会与非统一缩放的物体进行批处理。
5. 使用缩放尺度(1,1,1) 和 (1,2,1)的两个物体将不会进行批处理，但是使用缩放尺度(1,2,1) 和(1,3,1)的两个物体将可以进行批处理。
6. 使用不同材质的实例化物体（instance）将会导致批处理失败。
7. 拥有 lightmap 的物体含有额外（隐藏）的材质属性，比如：lightmap 的偏移和缩放系数等。所以，拥有 lightmap 的物体将不会进行批处理（除非他们指向 lightmap 的同一部分）。
8. 多通道的 shader 会妨碍批处理操作。比如，几乎 unity 中所有的着色器在前向渲染中都支持多个光源，并为它们有效地开辟多个通道。
9. 预设体的实例会自动地使用相同的网格模型和材质。

所以，尽量使用静态的批处理。

2 FPS 优化

2.1 垂直同步

FPS 的概念在绪论中已经解释过了，说道 FPS 就一定要说垂直同步，新建一个工程，运行，我们发现，场景中什么都没有，但是 FPS 上不去，就是 80 左右徘徊，这是为什么？这就是因为我们 Unity 中默认开启了垂直同步，进入 Edit - Project Settings - Quality 里面找到 VSync Count，修改为“Don’t Sync”，再运行场景，发现 FPS 提升了非常多。我们简单了解一下垂直同步为什么限制了 FPS 的增长。

如果我们选择等待垂直同步信号（也就是我们平时所说的垂直同步打开），那么在游戏中或许强劲的显卡迅速的绘制完一屏的图像，但是没有垂直同步信号的到

达，显卡无法绘制下一屏，只有等垂直同步的信号到达，才可以绘制。这样 FPS 自然要受到操作系统刷新率运行值的制约。

而如果我们选择不等待垂直同步信号（也就是我们平时所说的关闭垂直同步），那么游戏中作完一屏画面，显卡和显示器无需等待垂直同步信号就可以开始下一屏图像的绘制，自然可以完全发挥显卡的实力。但是不要忘记，正是因为垂直同步的存在，才能使得游戏进程和显示器刷新率同步，使得画面更加平滑和稳定。取消了垂直同步信号，固然可以换来更快的速度，但是在图像的连续性上势必打折扣。这也正是很多朋友抱怨关闭垂直后发现画面不连续的理论原因。

2.2 FPS 的限制

我们都知道 FPS 越高越好，但是其实不然，我们正常 FPS 在 60 左右就很流畅了，为什么要那么高，多余的 FPS 呢，一直以很高的 FPS 运行时非常耗电，耗性能的，所以我们要适当降低 FPS 才能更加优化我们的额项目，我们降低 FPS 的好处有：

1）省电，减少手机发热的情况；

2）能稳定游戏 FPS，减少出现卡顿的情况。

当我们要自己用脚本控制 FPS 时，需要先手动关闭垂直同步，方法在之前讲过。在 ProjectSetting-> Quality 中的 VSync Count 参数会影响你的 FPS，EveryVBlank 相当于 FPS=60，EverySecondVBlank = 30；这两种情况都不符合游戏的 FPS 的话，我们需要手动调整 FPS，首先关闭垂直同步这个功能，然后在代码的 Awake 方法里手动设置 FPS（Application.targetFrameRate = 45;）

3 Update 优化

3.1 OnBecameVisible 和 OnBecameInvisible 方法优化

我们的 Update 是非常耗性能的，尤其在对象非常多的时候，所以我们要尝试对 Update 进行优化：控制摄像机不可见的对象身上的“Update”就不执行。眼见为实，我们建立一个 Cube，再创建个脚本拖到 Cube 上，Update 里面只有一句话就是 print（“1”）；我们会发现这个测试语句一直都在执行，我们在里面再写两个方法，再在 Update 中加入限制，代码如下：

```
    void Start () {
        _isVisible = false;
    }

    // Update is called once per frame
    void Update () {
        if (_isVisible)
        {
            print("1");
        }
    }

    //可见
    private void OnBecameVisible()
    {
        _isVisible = true;
    }
    //不可见
    private void OnBecameInvisible()
    {
        _isVisible = false;
    }
}
```

现在我们，发现，让我们移动 Cube 时，超出了摄像机的视野范围的时候，测试语句就不执行了，这样就是一种对 Update 的优化。但是测试的时候，如若拖动的是摄像机，这两个个回调方法就不会生效。

4 物理引擎优化

这里有两点：

1.Edit - Project Settings - Time 里面有个 Fixed Timestep 属性，这个是控制 FixedUpdate 的执行速率的，默认是 0.02 秒执行一次，我们可以根据需求加大一些，节省 CPU 的消耗。

2.尽量少用网格碰撞器（Mesh Collider），大家都知道，在 Unity 中各种模型的网格是非常复杂的，所以他的碰撞检测在引擎中的实现也是非常复杂的，所以尽量少用。如果不能避免的话，尽量用减少 Mesh 的面片数，或用较少面片的代理体来代替。

5 内存优化

5.1 Update 中不要写多余代码

Update 中如果定义了局部变量，或者实例化了什么对象的话，将会非常的耗内存，所以一些能在方法外部，或者 Start 中处理的就尽量别再 Update 中编写。

5.2 对象池

对象池非常重要，但是在课程中会讲，这里就不细说了，上段代码：

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

17 个引用
public class ObjectPool : MonoBehaviour
{
    // 对象池用字典实现，string存对象名字，ArrayList存对象（在这个例子里是这么存的。分情况，存不同数据类型）    (ArrayList是非泛型，
    private Dictionary<string, List<GameObject>> pool = new Dictionary<string, List<GameObject>>();
    // 单例自身，方便调用类里的方法
    public static ObjectPool instance;

    0 个引用
    void Awake()
    {
        // Awake里面初始化
        instance = this;
    }
```

```
    // 存对象
    7 个引用
    public void ReturnToPool(GameObject o)
    {

        // 对象的名字，就是字典中的key
        string key = o.name;
        //判断当前对象池中有没有该对象池
        // (1) 有该对象池
        if (pool.ContainsKey(key))
        {
            //往对象池中存入对象
            pool[key].Add(o);

        }
        // (2) 没有该对象池
        else
        {
            // 新建一个对象池来存入该对象
            pool[key] = new List<GameObject> { o };
        }
        // 让该对象消失
        o.SetActive(false);

    }
```

```
    9 个引用
    public GameObject GetFromPool(string prefabName, Vector3 p, Quaternion q)
    {
        // 存进去的key为对象名加（Clone）
        string key = prefabName + "(Clone)";
        // o为生成或在对象池中取出的对象
        GameObject o;
        // 判断是否已经有该对象池并且该对象池中的对象数量大于0
        if (pool.ContainsKey(key) && pool[key].Count > 0)
        {
            // 取出一个对象
            o = pool[key][0];
            // 并在对象池中移除它
            pool[key].RemoveAt(0);
            // 设置取出的对象的位置及角度
            o.transform.position = p;
            o.transform.rotation = q;
            o.SetActive(true);

        }
        else
        {
            // 如果想取对象时没有该对象池，那么新生成一个物体，当他销毁的时候对象池就会产生了
            o = Instantiate(Resources.Load(prefabName), p, q) as GameObject;

        }
        //返回这个对象
        return o;
    }
}
```