

Unity Game Developer Postion Interview Prepare

deepwaterooo

September 13, 2018

Contents

1 Unity 优化	1
1.1 Shader 着色器	1
1.2 光源	1
1.3 碰撞器	2
1.4 贴图纹理	3
1.5 脚本	4
1.6 组件	5
1.7 NGUI	5
1.8 顶点数	6
1.9 材质	6
1.10 特效	6
1.11 模型物体	6
1.12 粒子系统	7
1.13 DrawCalls:	7
1.14 Draw Call Batching	7
1.15 Static Batching 静态批处理	8
1.16 Dynamic Batching 动态批处理	9
2 Unity UI/GUI/UGUI/NGUI	11
2.1 Unity3D 快速实现 UI 架构设计一	11
2.2 NGUI 与 UGUI 最详细对比	11
2.3 关于 Unity 中的 UGUI 优化，你可能遇到这些问题	12
2.3.1 界面制作	12
2.3.2 网格重建	15
2.3.3 界面切换	16
2.3.4 加载相关	16
2.3.5 字体相关	18
3 Object Oriented Design/Programming and Design Patterns	18
3.1 设计模式的几个原则	18
3.1.1 原则 1: 单一职责原则	19
3.1.2 原则 2: 里氏替换原则	21
3.1.3 原则 3: 依赖倒置原则	23
3.1.4 原则 4: 接口隔离原则	25
3.1.5 原则 5: 迪米特法则	28
3.1.6 原则 6: 开闭原则	31
3.1.7 如何去遵守这六个原则	31
3.2 Unity 中各种设计模式的实践与运用	31
3.2.1 Gang of Four Patterns in Unity (23 种 GOF 设计模式的 Unity 实现)	32
3.2.2 Game Programming Patterns in Unity (《游戏编程模式》的 Unity 实现)	33
3.3 单例模式 (Unity3D/C#)	33
3.3.1 【单例概述】	33
3.3.2 【C# 中的单例模式】	33
3.3.3 【Unity 中的伪单例模式】	36
3.3.4 【为什么要用单例】	39
3.3.5 【总结】	39

3.4	Unity 中常用的几种设计模式	39
3.4.1	单例模式	39
3.4.2	观察者模式	40
3.4.3	迭代器模式	40
3.4.4	访问者模式	41
3.5	设计模式之观察模式	42
3.5.1	模式中的角色	42
3.5.2	模式解读	43
3.5.3	模式总结	44
3.6	C# 设计模式学习笔记之建造者模式 (Builder)	45
3.7	C# 设计模式学习笔记之工厂设计模式	50
4	Camera 相关	54
4.1	Camera Shake in Unity	54
4.2	Unity 中做放大镜效果	55
4.3	Unity 3D- 摄像机 Clear Flags 和 Culling Mask 属性用途详解	57
4.4	[Unity 基础] 对 Camera 组件属性的一些理解	57
4.5	Unity 的 camera 组件	57
4.6	unity Camera 属性详解	57
5	Lights Lighting related	57
5.1	渲染管线中的 Rendering Path	57
5.1.1	Forward Rendering Path(前向渲染)	58
5.1.2	Deferred Shading Rendering Path(延迟渲染)	59
5.1.3	Lagacy Deferred Lighting Rendering Path(延迟光照)	60
5.1.4	Vertex Lit Rendering Path(顶点光照)	60
5.1.5	Unity3D 对几种渲染管线的统一处理	61
5.1.6	结束语	62
5.2	Unity5 Lighting 面板说明书	62
5.2.1	Object 面板	62
5.2.2	Scene 面板	64
5.2.3	Stats 面板	68
5.3	Unity3D Mesh 学习笔记 – 创建 MeshBuilder	70
5.4	深入了解光照贴图	71
6	Unity ShaderLab 学习总结	71
6.1	Shader 基础	71
6.1.1	SubShader 的 Tag	71
6.1.2	Pass	72
6.1.3	Shader 中的数据类型	74
6.1.4	ShaderLab 中的 Matrix	74
6.1.5	ShaderLab 中各个 Space 的坐标系	75
6.2	Shader 形态	76
6.2.1	Shader 形态之 1: 固定管线	76
6.2.2	Shader 形态之 2: 可编程 Shader	76
6.2.3	Shader 形态之 3: SurfaceShader	77
6.2.4	Shader 形态之 4: Compiled Shader	78
6.3	Unity 渲染路径 (Rendering Path) 种类	80
6.3.1	概述	80
6.3.2	渲染路径的内部阶段和 Pass 的 LightMode 标签	80
6.4	移动设备 GPU 架构简述	81
6.4.1	Part 1 - Frame Pipelining	81
6.4.2	Part 2 - Tile-based Rendering	81
6.4.3	Part 3 - The Midgard Shader Core	81
6.4.4	Part 4 - The Bifrost Shader Core	81
6.5	References	81
6.5.1	Implementing a Loading Bar in Unity	81
6.5.2	shader	82
6.5.3	Unity 动画	82

7	Unity3D - 物理引擎	82
7.1	Unity3D - 物理引擎之刚体部件 (Rigidbody) 详解	82
7.2	Unity3D Rigidbody 详解	83
7.2.1	属性	83
7.2.2	详细描述	83
7.2.3	Use The Right Size	84
7.2.4	Hints	84
7.3	Unity EventSystem 详解 (Unity Version 5.5.1)	85
7.3.1	Unity EventSystem	85
7.3.2	Message System (改进的消息系统)	85
7.3.3	Input Modules	86
7.3.4	Supported Events (支持的输入事件)	86
7.3.5	Raycasters (射线们)	87
7.3.6	Practice - 练习和测试	87
8	Nav Mesh	88
8.1	Navigation	88
8.1.1	Object: 物体参数面板	88
8.1.2	Bake: 烘培参数面板	89
8.2	Nav Mesh Agent: 导航组建参数面板	90
9	基类 MonoBehaviour/自带函数以及脚本执行的生命周期	90
9.1	MonoBehaviour 的生命周期:	90
9.2	编辑器 (Editor)	91
9.3	第一次场景加载 (First Scene Load)	91
9.4	第一帧更新之前 (Before the first frame update)	91
9.5	在帧之间 (In between frames)	91
9.6	更新顺序 (Update Order)	91
9.7	渲染 (Rendering)	91
9.8	协程程序 (Coroutines)	92
9.9	销毁 (When the Object is Destroyed)	92
9.10	退出游戏 (When Quitting)	92
9.11	官方给出的脚本中事件函数的执行顺序如下图	93
10	Unity Study Notes	95
10.1	Unity3D 重要模块的类图	95
10.2	GameObject 和 Component	95
10.3	3D 基础知识	96
10.3.1	点乘与叉乘	96
10.3.2	Quaternion	97
10.4	map 贴图	97
10.4.1	normalmap	97
10.4.2	Bump Map 凹凸贴图	97
10.5	NavMesh	97
10.6	粒子系统	97
10.7	references	98
10.8	comments	98
10.9	【Unity3D 基础教程】给初学者看的 Unity 教程 (一)	98
10.9.1	Unity 3d (just so that don't occupy a chapter)	98
10.9.2	重要类及其关系	98
10.9.3	Input 输入	99
10.9.4	Time	99
10.9.5	Physics 和 Transform	99
10.10	【Unity3D 基础教程】给初学者看的 Unity 教程 (二): 所有脚本组件的基类 – MonoBehaviour 的前世今生	100
10.10.1	引子	100
10.10.2	MonoBehaviour 的生命周期	100
10.10.3	MonoBehaviour 的那些坑	100

10.11 【Unity3D 基础教程】给初学者看的 Unity 教程 (三): 通过制作 Flappy Bird 了解 Native 2D 中的 Sprite, Animation	101
10.12 【Unity3D 基础教程】给初学者看的 Unity 教程 (四): 通过制作 Flappy Bird 了解 Native 2D 中的 Sprite, Animation	101
10.13 【Unity3D 基础教程】给初学者看的 Unity 教程 (五): 详解 Unity3d 中的协程 (Coroutine)	101
10.13.1 为什么需要协程	101
10.13.2 协程是什么	101
10.13.3 协程怎么用?	101
10.13.4 协程是怎么工作的	101
10.13.5 协程是如何实现延时的?	102
10.13.6 还有些更好玩的?	103
10.14 【Unity3D 基础教程】给初学者看的 Unity 教程 (六): 理解 unity 的新 GUI 系统 (UGUI)	104
10.15 【Unity3D 基础教程】给初学者看的 Unity 教程 (七): 在 unity 中构建健壮的单例模式 (Singleton)	104
10.15.1 为什么需要单例模式	104
10.15.2 单例的设计原则	105
10.15.3 设计单例的基类	105
10.15.4 设计单例的管理类	105
10.15.5 如何拓展新的单例	106
10.16 场景视图 (Scene View) 导航	107
10.16.1 上下左右箭头键	107
10.16.2 使用手形工具	107
10.16.3 漫游模式	107
10.16.4 场景小图示 (Scene Gizmo)	107
10.17 定位游戏对象 (GameObjects)	107
10.17.1 聚焦游戏对象	107
10.17.2 移动、旋转和缩放	108
10.17.3 小图示显示切换器 (Gizmo Display Toggles)	108
10.17.4 单位对齐	108
10.17.5 表面对齐	108
10.17.6 查看旋转	108
10.17.7 顶点对齐	108
10.18 视图模式	109
10.18.1 绘图模式 (Draw Mode)	109
10.18.2 渲染模式 (Render Mode)	109
10.18.3 场景照明、游戏覆盖和试听模式	109
10.19 小图示和图标显示控制	109
10.19.1 图标选择器 (Icon Selector)	109
10.19.2 显示和隐藏图标与小图示	110
10.20 iOS	110
10.21 Unity 动画	110
10.21.1 shader	110

11 Unity C# Functions code practice

111

12 [Unity] 面试题整理 100 题	113
12.1 相机	113
12.1.1 移动相机动作在哪个函数里, 为什么在这个函数里?	113
12.1.2 在场景中放置多个 Camera 并同时处于活动状态会发生什么?	114
12.1.3 Unity 中, 照相机的 Clipping Planes 的作用是什么? 调整 Near、Fare 两个值时, 应该注意什么?	114
12.1.4 将 Camera 组件的 ClearFlags 选项选成 Depth only 是什么意思? 有何用处?	114
12.1.5 Camera.depth	114
12.2 光源	114
12.2.1 : Unity 提供了几种光源, 分别是什么	114
12.2.2 写光照计算中的 diffuse 的计算公式	114
12.2.3 写出光照计算中的 diffuse 的计算公式	114
12.2.4 实时点光源的优缺点是什么?	114
12.3 物理碰撞、刚体等	114
12.3.1 Unity3D 中的碰撞器和触发器的区别?	114

12.3.2 物体发生碰撞的必要条件	115
12.3.3 CharacterController 和 Rigidbody 的区别?	115
12.3.4 在物体发生碰撞的整个过程中, 有几个阶段, 分别列出对应的函数三个阶段	115
12.3.5 Unity3d 的物理引擎中, 有几种施加力的方式, 分别描述出来	115
12.3.6 射线检测碰撞物的原理是?	115
12.4 3D 数学	115
12.4.1 四元数有什么作用?	115
12.4.2 简述四元数 Quaternion 的作用, 四元数对欧拉角的优点?	115
12.4.3 U3D 中用于记录节点空间几何信息的组件名称, 及其父类名称	115
12.4.4 向量的点乘、叉乘以及归一化的意义?	115
12.4.5 矩阵相乘的意义及注意点	115
12.4.6 什么是 DrawCall? DrawCall 高了又什么影响? 如何降低 DrawCall?	116
12.4.7 如何在 Unity3D 中查看场景的面试, 顶点数和 Draw Call 数? 如何降低 Draw Call 数?	116
12.5 GUI	116
12.5.1 Unity 和 cocos2d 的区别	116
12.5.2 为何大家都在移动设备上寻求 U3D 原生 GUI 的替代方案	116
12.5.3 请简述如何在不同分辨率下保持 UI 的一致性	116
12.5.4 请简述 OnBecameVisible 及 OnBecameInvisible 的发生时机, 以及这一对回调函数的意义?	116
12.5.5 简述 NGUI 中 Grid 和 Table 的作用?	116
12.5.6 请简述 NGUI 中 Panel 和 Anchor 的作用	116
12.6 生命周期	116
12.6.1 Unity3d 脚本从唤醒到销毁有着一套比较完整生命周期, 请列出系统自带的几个重要的方法。	116
12.6.2 OnEnable、Awake、Start 运行时的发生顺序? 哪些可能在同一个对象周期中反复的发生?	116
12.6.3 物理更新一般放在哪个系统函数里?	117
12.6.4 GPU 的工作原理	117
12.6.5 什么是渲染管道?	117
12.6.6 MeshRender 中 material 和 sharedmaterial 的区别?	117
12.6.7 简述 SkinnedMesh 的实现原理	117
12.7 渲染	117
12.7.1 Unity3D Shader 分哪几种, 有什么区别?	117
12.7.2 alpha blend 工作原理	117
12.7.3 两种阴影判断的方法、工作原理。	118
12.7.4 阴影由两部分组成: 本影与半影	118
12.7.5 阴影分为两种: 自身阴影和投射阴影	118
12.7.6 Unity 内置了几个 RenderQueue 的字面值:	118
12.7.7 Unity 的 Shader 中, Blend SrcAlpha 1 - SrcAlpha 这句话是什么意思?	118
12.7.8 简述水面倒影的渲染原理	118
12.7.9 请问 alpha test 在何时使用? 能达到什么效果?	119
12.7.10 有 A 和 B 两组物体, 有什么办法能够保证 A 组物体永远比 B 组物体先渲染?	119
12.7.11 Vertex Shader 是什么, 怎么计算?	119
12.7.12 MipMap 是什么, 作用?	119
12.7.13 半透明物体的渲染:	119
12.7.14 不透明物体的渲染:	119
12.7.15 裁剪区域的渲染:	120
12.7.16 蒙皮网格渲染	120
12.7.17 蒙皮网格渲染如何解决:	120
12.7.18 粒子系统渲染:	120
12.7.19 粒子系统渲染的优化建议:	120
12.7.20 图像后处理:	120
12.7.21 开启多线程后, UI 渲染不稳定的原因?	121
12.7.22 实时阴影: 两个地方会增大实时阴影的耗时:	121
12.8 其它	121
12.8.1 简述 prefab 的用处	121
12.8.2 使用 unity3d 实现 2d 游戏, 有几种方式?	121
12.8.3 什么叫做链条关节?	121
12.8.4 物体自身旋转使用的函数?	121

12.8.5 Unity3d 提供了一个用于保存和读取数据的类 (PlayerPrefs), 请列出保存和读取整形数据的函数	121
12.8.6 请描述为什么 Unity3d 中会在组件上出现数据丢失的情况	121
12.8.7 请描述游戏动画有哪几种, 以及其原理?	121
12.8.8 如何安全的在不同工程间安全地迁移 asset 数据? 三种方法	122
12.8.9 什么叫动态合批? 跟静态合批有什么区别?	122
12.8.10 什么是 LightMap?	122
12.8.11 当一个细小的高速物体撞向另一个较大的物体时, 会出现什么情况? 如何避免?	122
12.8.12 请写出求斐波那契数列任意一位的值得算法	122
12.8.13 什么是里氏代换元则?	122
12.8.14 Mock 和 Stub 有何区别?	122
12.8.15 如何让已经存在的 GameObject 在 LoadLevel 后不被卸载掉?	122
12.8.16 将图片的 TextureType 选项分别选为 Texture 和 Sprite 有什么区别	122
12.8.17 问一个 Terrain, 分别贴 3 张, 4 张, 5 张地表贴图, 渲染速度有什么区别? 为什么?	122
12.8.18 使用动态字体时是否会生成字符纹理	123
12.8.19 为什么 dynamic font 在 unicode 环境下优于 static font	123
12.8.20 动态加载资源的方式? (有时候也问区别, 具体请百度)	123
12.8.21 动态加载资源的方式?	123
12.9 与 Mobile 端交互	123
12.9.1 Unity 和 Android 与 iOS 如何交互?	123
12.9.2 LOD 是什么, 优缺点是什么?	123
12.9.3 UNITY3d 在移动设备上的一些优化资源的方法	123
12.9.4 你用过哪些插件?	124
12.10 语言基础、OOP	124
12.10.1 请简述 ArrayList 和 List 之间的主要区别。	124
12.10.2 请简述 ArrayList 和 List<Int> 的主要区别	124
12.10.3 请简述 sealed 关键字用在类声明时与函数声明时的作用。	124
12.10.4 请简述 private, public, protected, internal 的区别。	124
12.10.5 反射的实现原理?	125
12.10.6 简述一下对象池, 你觉得在 FPS 里哪些东西适合使用对象池?	125
12.10.7 请简述 GC (垃圾回收) 产生的原因, 并描述如何避免?	125
12.10.8 如何优化内存?	125
12.10.9 如何销毁一个 UnityEngine.Object 及其子类?	125
12.10.10 能用 foreach 遍历访问的对象需要实现 __	125
12.10.11 简述 StringBuilder 和 String 的区别?	125
12.10.12 已知 strcpy 函数的原型是:	126
12.10.13 堆和栈的区别?	126
12.10.14 Heap 与 Stack 有何区别?	127
12.10.15 值类型和引用类型有何区别?	127
12.10.16 C# 中所有引用类型的基类是什么	127
12.10.17 结构体和类有何区别?	127
12.10.18 C# 中四种访问修饰符是哪些? 各有什么区别?	127
12.10.19 请说出 4 种面向对象的设计原则, 并分别简述它们的含义。	128
12.10.20 请描述 Interface 与抽象类之间的不同	128
12.10.21 下列代码在运行中会产生几个临时对象?	128
12.10.22 下列代码在运行中会发生什么问题? 如何避免?	128
12.10.23 在编辑场景时将 GameObject 设置为 Static 有何作用?	129
12.10.24 Unity3D 是否支持写成多线程程序? 如果支持的话需要注意什么?	129
12.10.25 什么是协同程序?	129
12.10.26 Unity3D 的协程和 C# 线程之间的区别是什么?	129
12.10.27 协同程序的执行代码是什么? 有何用处, 有何缺点?	129
12.10.28 C# 中的排序方式有哪些?	130
12.10.29 ref 参数和 out 参数是什么? 有什么区别?	130
12.10.30 C# 的委托是什么? 有何用处?	130
12.10.31 概述序列化:	130
12.10.32 概述 c# 中代理和事件?	130
12.10.33 TCP/IP 协议栈各个层次及分别的功能	130
12.10.34 客户端与服务器交互方式有几种?	130

12.10.35Net 与 Mono 的关系?	131
12.10.3C# 和 C++ 的区别?	131
12.10.3简述 Unity3D 支持的作为脚本的语言的名称	131
13 第一部分	131
14 第二部分	140
15 C++	143

1 Unity 优化

- DrawCalls: 控制电脑平台上 DrawCalls 几千个之内, 移动平台上 DrawCalls 几百个之内
- Verts: PC 平台的话保持场景中显示的顶点数少于 300W, 移动设备的话少于 10W, 一切取决于你的目标 GPU 与 CPU。
- 需要注意的是:
 - 如果在 Profiler 下的 GPU 中显示的 RenderTexture.SetActive() 占用率很高的话, 那么可能是因为你同时打开了编辑窗口的原因, 而不是 U3D 的 BUG。

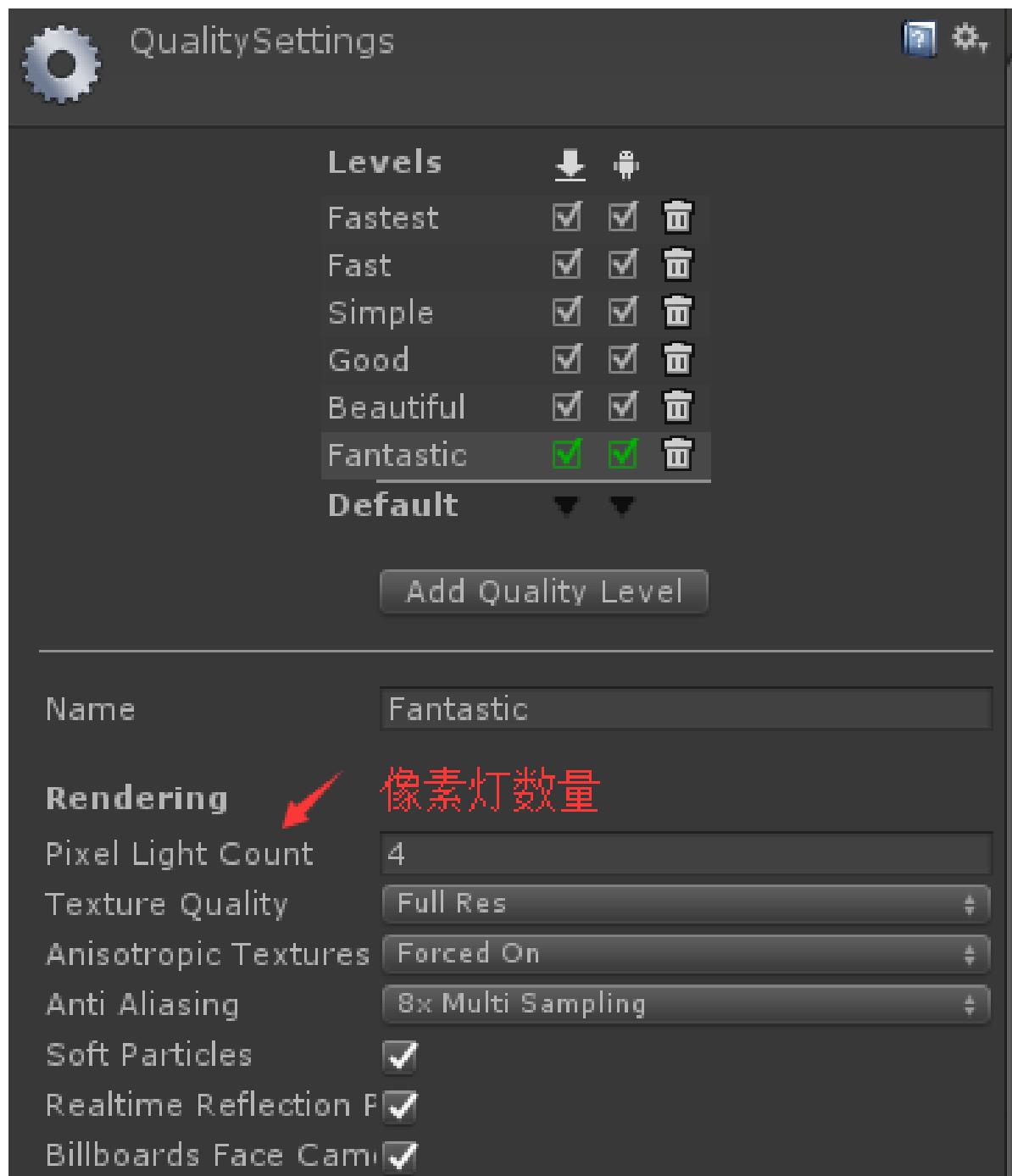
1.1 Shader 着色器

- (1) 有些着色器可能是处理器密集型的, 因此最好为材质指定移动设备专用的着色器。将着色器从 Diffuse 修改为 Mobile/Diffuse。
- (2) shader 中用贴图混合的方式去代替多重通道计算。
- (3) shader 中注意 float/half/fixed 的使用。
- (4) shader 中不要用复杂的计算 pow,sin,cos,tan,log 等。
- (5) shader 中越少 Fragment 越好。
- (6) 自己写的 shader 请注意复杂操作符计算, 类似 pow,exp,log,cos,sin,tan 等都是很耗时的计算, 最多只用一次在每个像素点的计算。不推荐你自己写 normalize, dot, inversesqrt 操作符, 内置的肯定比你写的好。
- (7) 需要警醒的是 alpha test, 这个非常耗时。
- (8) 浮点类型运算: 精度越低的浮点计算越快。
 - 在 CG/HLSL 中: float : 32 位浮点格式, 适合顶点变换运算, 但比较慢。
 - * half: 16 位浮点格式, 适合贴图和 UV 坐标计算, 是 highp 类型计算的两倍。
 - * fixed: 10 位浮点格式, 适合颜色, 光照, 和其他。是 highp 格式计算的四倍。

1.2 光源

- (1) 最好使用平行光, 点光源和聚光灯消耗资源比较大
- (2) 限制灯光使用数量, 尽可能不用灯光。动态灯光更加不要了。
- (3) Lightmapping 烘焙灯光, 为场景添加光源时要考虑一下, 因为有渲染开销。如果你以前做过着色器编程, 你会知道为了支持动态光源的渲染, 要付出额外的代价。每个光源都需要渲染对象, 根据对象使用的着色器、材质计算最终的光源效果, 这个计算开销很大。尽可能的在渲染之前就将光源细节“烘焙 (bake)”到对象的纹理中。”烘焙”是使用静态光源效果的渲染方式, 可以实现相同的视觉效果, 而无需额外的计算开销。
- (4) 实时阴影技术非常棒, 但消耗大量计算。为 GPU 和 CPU 都带来了昂贵的负担
- (5) 灯光的 Shadow Type 只对 PC 平台有效, 也就是说在移动平台是没有阴影效果的 (亲测), 另外软阴影更为昂贵, 耗资源!!!

- (6) light 的 Render Mode 下的 Auto 是根据附近灯光的亮度和当前质量的设置 (Edit ->Project Settings -> Quality) 在运行时确定, Not Important 为顶点渲染, Important 为像素渲染 (更耗资源), 但是像素渲染能够实现顶点渲染不能实现的效果, 比如实时阴影, 因此必须权衡前后照明质量和游戏速度。像素灯的实际数量可以在质量设置 (Edit -> Project Settings -> Quality) 中的进行设置。



1.3 碰撞器

- (1) 通常, 碰撞器根据复杂度排序, 对象越复杂, 使用这个对象的性能开销越大。有可能的话, 用盒子或者球体 (Box/Sphere) 来封装对象, 这样碰撞器的计算最少。不要用网格碰撞器 (Mesh Collider)。
- (2) 注意碰撞体的碰撞层, 不必要的碰撞检测请舍去。

1.4 贴图纹理

- (1) 可以把图像纹理或者其它资源共享使用, 尽量避免透明, 可以使用填充黑色

- (2) 尝试用压缩贴图格式，或用 16 位代替 32 位。图片压缩将降低你的图片大小（更快地加载更小的内存跨度 (footprint)），而且大大提高渲染表现。压缩贴图比起未压缩的 32 位 RGBA 贴图占用内存带宽少得多。
 - (3) 之前 U3D 会议还听说过一个优化，贴图尽量都用一个大小的格式 (512 * 512, 1024 * 1024)，这样在内存之中能得到更好的排序，而不会有内存之间空隙。
 - (4) MIPMAPs，跟网页上的略缩图原理一样，在 3D 游戏中我们为游戏的贴图生成多重纹理贴图，远处显示较小的物体用小的贴图，显示比较大的物体用精细的贴图。这样能更加有效的减少传输给 GPU 中的数据。但同时也会增加内存的大小，自己根据项目来权衡利弊
 - (5) 如果你做了一个图集是 1024X1024 的。此时你的界面上只用了图集中的一张很小的图，那么很抱歉 1024X1024 这张大图都需要载入你的内存里面，1024 就是 4M 的内存，如果你做了 10 个 1024 的图集，你的界面上刚好都只用了每个图集里面的一张小图，那么再次抱歉你的内存直接飙 40M。意思是任何一个 4096 的图片，不管是图集还是 texture，他都占用 $4 \times 4 = 16M$ ？
 - (6) IOS 平台使用 PVRTC 压缩纹理。Android 平台使用 ETC1 格式压缩。均可以减至 1/4 的内存大小，优化非常明显！目前主流的 Android 机型基本都支持 ETC1 格式压缩。但 ETC1 只能支持非 Alpha 通道的图片压缩。所以一般把 Alpha 通道图分离出来，绘制到 GPU 显存时，a 值从 Alpha 图里获取，无 Alpha 通道的图就可以使用 ETC1 压缩。
 - (7) 设置不透明贴图的压缩格式为 ETC 4bit，因为 android 市场的手机中的 GPU 有多种，每家的 GPU 支持不同的压缩格式，但他们都兼容 ETC 格式
 - 对于透明贴图，我们只能选择 RGBA 16bit 或者 RGBA 32bit。
 - (8) 减少 FPS，在 ProjectSetting -> Quality 中的 VSync Count 参数会影响你的 FPS，EveryVBlank 相当于 FPS=60，EverySecondVBlank = 30；
 - 如果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 (VSync = Vertical Sync) 这个功能，即设置 VSync 为 Don't Sync
 - 然后在代码的 Awake 方法里手动设置 FPS
- 1 Application.targetFrameRate = 45;
- 降低 FPS 的好处：
 - * 1) 省电，减少手机发热的情况；
 - * 2) 能够稳定游戏 FPS，减少出现卡顿的情况。
 - (9) 当我们设置了 FPS 后，再调整下 Fixed timestep 这个参数，这个参数在 ProjectSetting->Time 中，目的是减少物理计算的次数，来提高游戏性能。
 - (10) 尽量少使用 Update LateUpdate FixedUpdate，这样也可以提升性能和节省电量。多使用事件（不是 SendMessage，使用自己写的，或者 C# 中的事件委托）。
 - (11) 待机时，调整游戏的 FPS 为 1，节省电量。
 - (12) 图集大小最好不要高于 1024，否则游戏安装之后、低端机直接崩溃、原因是手机系统版本低于 2.2、超过 1000 的图集无法读取导致。2.2 以上没有遇见这个情况。注意手机的 RAM 与 ROM、小于 512M 的手机、直接放弃机型适配。
 - (13) 不同设备要使用不同的纹理大小，尤其是 UI 和大型背景中的纹理。《Shadow Blade》使用的是通用型模板，但如果在启动时检测到设备大小和分辨率，就会载入不同资产。
 - (14) 远处的物体绘制在 skybox 上

1.5 脚本

- (1) 如果你不需要运行某一个脚本，那么就禁用它。不管它多少的小，或者出现的很少，但每一个处理都需要占用时间。
- (2) 不要留着未实现的 Update, FixedUpdate 等方法，用不到就删除，不然会执行，消耗时间！
- (3) 移除代码中的任何字符串连接，因为这会给 GC 留下大量垃圾。使用 StringBuilder 链接字符串

- (4) 用简单的”for” 循环代替”foreach” 循环。由于某些原因，每个”foreach” 循环的每次迭代会生成 24 字节的垃圾内存。一个简单的循环迭代 10 次就可以留下 240 字节的垃圾内存。
- (5) 更改我们检查游戏对象标签的方法。用”if (go.CompareTag (“Enemy”))” 来代替”if (go.tag == “Enemy”)”。

```

1 if (go.CompareTag ("Enemy")) {}
2 // 来代替
3 // if (go.tag == "Enemy")

```

– 在一个内部循环调用对象分配的标签属性以及拷贝额外内存，这是一个非常糟糕的做法。

- (6) 不使用 LINQ 命令，因为它们一般会分配中间缓器，而这很容易生成垃圾内存。
- (7) 修改代码以免依赖”ControllerColliderHit” 回调函数。这证明这些回调函数处理得并不十分迅速。
- (8) 要谨慎评估触发器的”onInside” 回调函数，在我们的项目中，我们尽量在不依赖它们的情况下模拟逻辑。
- (9) 注意是否有多余的动画脚本，模型自动导入到 U3D 会有动画脚本，大量的话会严重影响消耗 CPU 计算。
- (10) 尽量避免每帧处理，可以每隔几帧处理一次

```

1 void Update() {
2     if (Time.frameCount % 5 == 0) {
3         DoSomeThing();
4     }
5 }

```

- (11) 尽量避免使用 float，而使用 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，例如：使用 x*0.5f 而不是 x/2.0f。
- (12) 避免使用

```

1 for (int i = 0; i < myArray.Length; i++)

```

– 而应该这样

```

1 int length = myArray.Length;
2 for (int i = 0; i < length; i++)

```

- (13) 少使用临时变量，特别是在 Update OnGUI 等实时调用的函数中定义临时变量。
- (14) 协同是一个好方法。可以使用协同程序来代替不必每帧都执行的方法。（还有 InvokeRepeating 方法也是一个好的取代 Update 的方法）。
- (15) 不要使用 SendMessage 之类的方法，他比直接调用方法慢了 100 倍，你可以直接调用或通过 C# 的委托来实现。
- (16) 操作 transform.localPosition 的时候请小心，移动 GameObject 是非常平常的一件事情，以下代码看起来很简单：

```

1 transform.localPosition += new Vector3 ( 10.0f * Time.deltaTime, 0.0f, 0.0f );

```

– 但是小心了，假设上面这个 GameObject 有一个 parent，并且这个 parent GameObject 的 localScale 是 (2.0f, 2.0f, 2.0f)。你的 GameObject 将会移动 20.0 个单位/秒。

– 因为该 GameObject 的 world position 等于：

```

1 Vector3 offset = new Vector3( my.localPosition.x * parent.lossyScale.x,
2                               my.localPosition.y * parent.lossyScale.y,
3                               my.localPosition.z * parent.lossyScale.z );
4 Vector3 worldPosition = parent.position + parent.rotation * offset;

```

– 换句话说，上面这种直接操作 localPosition 的方式是在没有考虑 scale 计算的时候进行的，为了解决这个问题，unity3d 提供了 Translate 函数，所以正确的做法应该是：

```
1 transform.Translate ( 10.0f * Time.deltaTime, 0.0f, 0.0f );
```

- (17) 减少固定增量时间，将固定增量时间值设定在 0.04-0.067 区间（即，每秒 15-25 帧）。您可以通过 Edit -> Project Settings -> Time 来改变这个值。这样做降低了 FixedUpdate 函数被调用的频率以及物理引擎执行碰撞检测与刚体更新的频率。如果您使用了较低的固定增量时间，并且在主角身上使用了刚体部件，那么您可以启用插值办法来平滑刚体组件。
- (18) 减少 GetComponent 的调用使用，GetComponent 或内置组件访问器（transform）会产生明显的开销。您可以通过一次获取组件的引用来自避免开销，并将该引用分配给一个变量（有时称为“缓存”的引用）。

```
1 Transform myTransform ;  
2 void Awake () {  
3     myTransform = transform;  
4 }
```

- (19) 同时，在某些可能的情况下，您也可以使用结构（struct）来代替类（class）。这是因为，结构变量主要存放在栈区而非堆区。因为栈的分配较快，并且不调用垃圾回收操作，所以当结构变量比较小时可以提升程序的运行性能。但是当结构体较大时，虽然它仍可避免分配/回收的开销，而它由于“传值”操作也会导致单独的开销，实际上它可能比等效对象类的效率还要低。
- (20) 使用 GUILayout 函数可以很方便地将 GUI 元素进行自动布局。然而，这种自动化自然也附带着一定的处理开销。您可以通过手动的 GUI 功能布局来避免这种开销。此外，您也可以设置一个脚本的 useGUILayout 变量为 false 来完全禁用 GUI 布局：

```
void Awake () {  
    useGUILayout = false;  
}
```

- (21) 最小化碰撞检测请求（例如 ray casts 和 sphere checks），尽量从每次检查中获得更多信息。
- (22) 在 edit->project setting->time 中调大 FixedTimestep（真实物理的帧率）来减少 cpu 损耗
- (23) 尽量不要动态的 instantiate 和 destroy object，使用 object pool
- (24) 尽量不要再 update 函数中做复杂计算，如有需要，可以隔 N 帧计算一次
- (25) 不要使用内置的 onGUI 函数处理 gui，使用其他方案，如 NGUI

1.6 组件

- (1) 尽可能的使用简单组件—如果你不需求功能较多的组件，那么就自己去实现它避免一起使用大量系统组件。比如，CharacterController 是一个很废资源的组件，那么最好使用刚体来定义自己的解决方案。
- (2) 面对性能更弱的设备，要用 skinned mesh 代替 physics cloth。cloth 参数在运行表现中发挥重要作用，如果你肯花些时间找到美学与运行表现之间的平衡点，就可以获得理想的结果。
- (3) 在物理模拟过程中不要使用 ragdolls（布娃娃系统），只有在必要时才让它生效。
- (4) 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

1.7 NGUI

- (1) NGUI 中所有 Panel 都有一个 Depth 值影响着他下面的所有挂件。如果你正在创建一个使用多个窗口的复杂 UI，通常最好的做法是每个窗口有一个 UIPanel。请确认你的 panel 不会拥有相同的 depth 值。如果这个值是一样的，为了保证绘制顺序，draw call 将会开始频繁分割，这将导致产生比平常更多的 draw call。

1.8 顶点数

- (1) 尽量减少顶点数

1.9 材质

- (1) 尽可能共用材质。这样便可以减少 DrawCall，引擎可以进行其批处理！
- (2) 如果你需要通过脚本来控制单个材质属性，需要注意改变 Renderer.material 将会造成一份材质的拷贝。因此，你应该使用 Renderer.sharedMaterial 来保证材质的共享状态。
- (3) 有一个合并模型材质不错的插件叫 Mesh Baker

1.10 特效

- (1) 如果不需要别用雾效 (fog)
- (2) 要找到美学/性能之间的平衡，就免不了许多粒子效果的迭代。减少发射器数量并尽量减少透明度需求也是一大挑战。

1.11 模型物体

- (1) 不要有不必要的三角面。面片数最好控制在 300~2000 面片
- (2) UV 贴图中的接缝和硬边越少越好。
 - 需要注意的是，图形硬件需要处理顶点数和硬件报告说的并不一样。不是硬件说能渲染几个点就是几个点。模型处理应用通常展示的是几何顶点数量。例如，一个由一些不同顶点构成的模型。在显卡中，一些集合顶点将会被分离 (split) 成两个或者更多逻辑顶点用作渲染。如果有法线、UV 坐标、顶点色的话，这个顶点必须会被分离。所以在游戏中处理的实际数量显然要多很多。
- (3) LOD (Level Of Detail) 是很常用的 3D 游戏技术了，其功能理解起来则是相当于多重纹理贴图。在以在屏幕上显示模型大小的比例来判断使用高或低层次的模型来减少对 GPU 的传输数据，和减少 GPU 所需要的顶点计算。
- (4) 摄像机分层距离剔除 (Per-Layer Cull Distances)：为小物体标识层次，然后根据其距离主摄像机的距离判断是否需要显示。
- (5) 遮挡剔除 (Occlusion Culling) 其实就是当某个物体在摄像机前被另外一个物体完全挡住的情况，挡住就不发送给 GPU 渲染，从而直接降低 DRAW CALL。不过有些时候在 CPU 中计算其是否被挡住则会很耗计算，反而得不偿失。
- (6) 将不需要移动的物体设为 Static，让引擎可以进行其批处理。
- (7) 用单个蒙皮渲染、尽量少用材质、少用骨骼节点、移动设备上角色多边形保持在 300~1500 内（当然还要看具体的需求）、PC 平台上 1500~4000 内（当然还要看具体的需求）。
 - 角色的面数一般不要超过 1500，骨骼数量少于 30 就好，越多的骨骼就会越多的带来 CPU 消耗，角色 Material 数量一般 1~2 个为最佳。
- (8) 导入 3D 模型之后，在不影响显示效果的前提下，最好打开 Mesh Compression。Off, Low, Medium, High 这几个选项，可酌情选取。
- (9) 避免大量使用 unity 自带的 Sphere 等内建 Mesh，Unity 内建的 Mesh，多边形的数量比较大，如果物体不要求特别圆滑，可导入其他的简单 3D 模型代替。
- (10) 每个角色尽量使用一个 Skinned Mesh Renderer，这是因为当角色仅有一个 Skinned Mesh Renderer 时，Unity 会使用视锥型可见性裁剪和多边形网格包围体更新的方法来优化角色的运动，而这种优化只有在角色仅含有一个 Skinned Mesh Renderer 时才会启动。
- (11) 对于静态物体顶点数要求少于 500，UV 的取值范围不要超过 (0,1) 区间，这对于纹理的拼合优化很有帮助。
- (12) 不需要的 Animation 组件就删掉

1.12 粒子系统

- 粒子系统运行在 iPhone 上时很慢，怎么办？因为 iPhone 拥有相对较低的 fillrate。如果您的粒子效果覆盖大部分的屏幕，而且是 multiple layers 的，这样即使最简单的 shader，也能让 iPhone 傻眼。我们建议把您的粒子效果 baking 成纹理序列图。然后在运行时可以使用 1-2 个粒子，通过动画纹理来显示它们。这种方式可以取得很好的效果，以最小的代价。
- 自带地形：地形高度图尺寸小于 257，尽量使用少的混合纹理数目，尽量不要超过 4 个，Unity 自带的地形时十分占资源的，强烈建议不要使用，自己制作地形，尽量一张贴图搞定
 - drawcall 是啥？draw：绘制，call：调用，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。那么，是谁去调用这些接口呢？CPU。
 - fragment 是啥？经常有人说 vf 啥的，vertex 我们都知道是顶点，那 fragment 是啥呢？说它之前需要先说一下像素，像素各位应该都知道吧？像素是构成数码影像的基本单元呀。那 fragment 呢？是有可能成为像素的东西。啥叫有可能？就是最终会不会被画出来不一定，是潜在的像素。这会涉及到谁呢？GPU。
 - batching 是啥？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（drawcall）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。
 - 内存的分配：记住，除了 Unity3D 自己的内存损耗。我们可是还带着 Mono 呢啊，还有托管的那一套东西呢。更别说你一激动，又引入了自己的几个 dll。这些都是内存开销上需要考虑到的。
 - CPU 方面：

* 上文中说了，drawcall 影响的是 CPU 的效率，而且也是最知名的一个优化点。但是除了 drawcall 之外，还有哪些因素也会影响到 CPU 的效率呢？让我们一一列出暂时能想得到的：

- (1) DrawCalls
- (2) 物理组件（Physics）
- (3) GC（什么？GC 不是处理内存问题的嘛？匹夫你不要骗我啊！不过，匹夫也要提醒一句，GC 是用来处理内存的，但是是谁使用 GC 去处理内存的呢？）
- (4) 当然，还有代码质量

1.13 DrawCalls：

- 前面说过了，DrawCall 是 CPU 调用底层图形接口。比如有上千个物体，每一个的渲染都需要去调用一次底层接口，而每一次的调用 CPU 都需要做很多工作，那么 CPU 必然不堪重负。但是对于 GPU 来说，图形处理的工作量是一样的。所以对 DrawCall 的优化，主要就是为了尽量解放 CPU 在调用图形接口上的开销。所以针对 drawcall 我们主要的思路就是每个物体尽量减少渲染次数，多个物体最好一起渲染。所以，按照这个思路就有了以下几个方案：

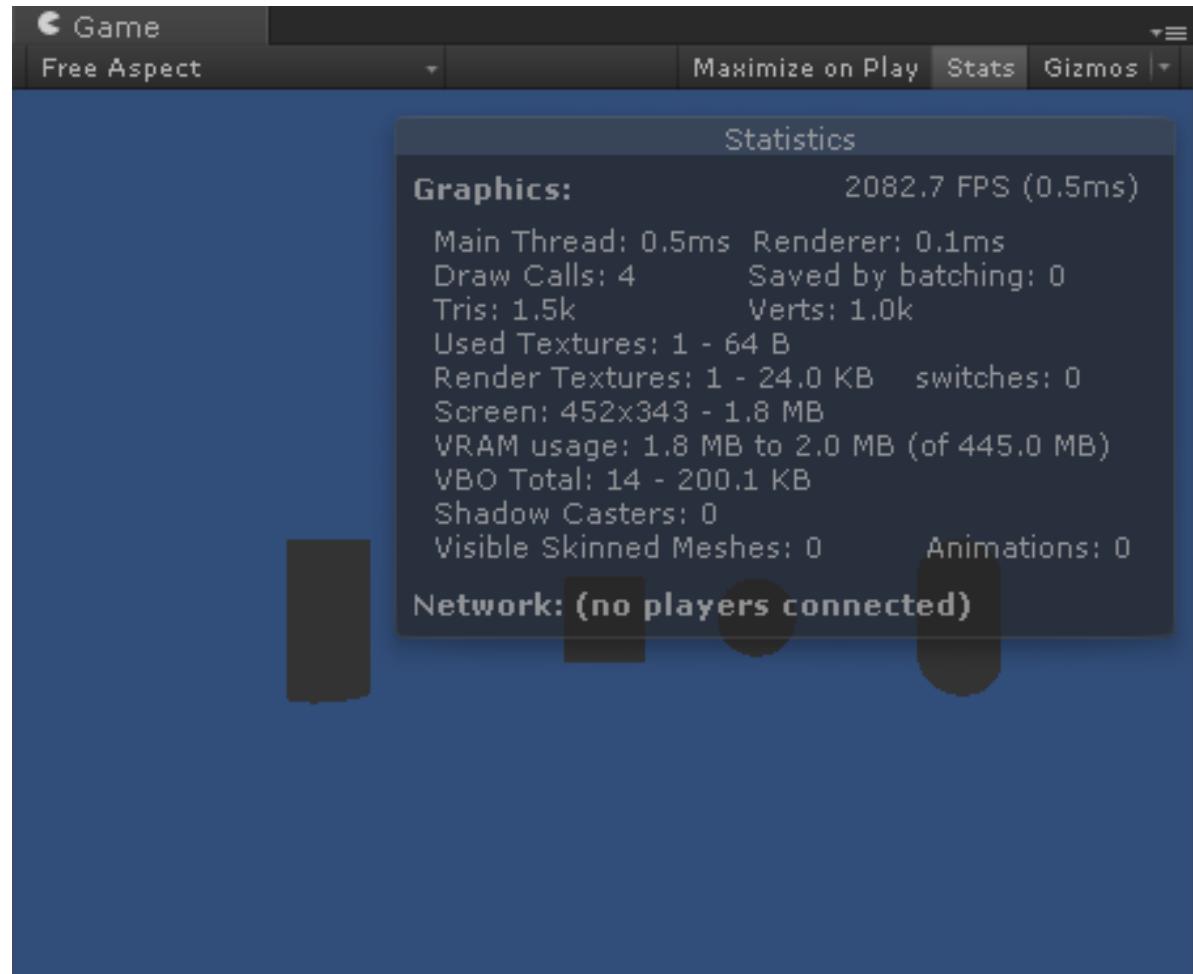
- 使用 Draw Call Batching，也就是描绘调用批处理。Unity 在运行时可以将一些物体进行合并，从而用一个描绘调用来渲染他们。具体下面会介绍。
- 通过把纹理打包成图集来尽量减少材质的使用。
- 尽量少的使用反光啦，阴影啦之类的，因为那会使物体多次渲染。

1.14 Draw Call Batching

- 首先我们要先理解为何 2 个没有使用相同材质的物体即使使用批处理，也无法实现 Draw Call 数量的下降和性能上的提升。
- 因为被“批处理”的 2 个物体的网格模型需要使用相同材质的目的，在于其纹理是相同的，这样才可以实现同时渲染的目的。因而保证材质相同，是为了保证被渲染的纹理相同。
- 因此，为了将 2 个纹理不同的材质合二为一，我们就需要进行上面列出的第二步，将纹理打包成图集。具体到合二为一这种情况，就是将 2 个纹理合成一个纹理。这样我们就可以只用一个材质来代替之前的 2 个材质了。
- 而 Draw Call Batching 本身，也还会细分为 2 种。

1.15 Static Batching 静态批处理

- 看名字，猜使用的情景。
- 静态？那就是不动的咯。还有呢？额，听上去状态也不会改变，没有“生命”，比如山石，楼房校舍啥的。那和什么比较类似呢？嗯，聪明的各位一定觉得和场景的属性很像吧！所以我们的场景似乎就可以采用这种方式来减少 draw call 了。
- 那么写个定义：只要这些物体不移动，并且拥有相同的材质，静态批处理就允许引擎对任意大小的几何物体进行批处理操作来降低描绘调用。
- 那要如何使用静态批来减少 Draw Call 呢？你只需要明确指出哪些物体是静止的，并且在游戏中永远不会移动、旋转和缩放。想完成这一步，你只需要在检测器（Inspector）中将 Static 复选框打勾即可！
- 至于效果如何呢？
- 举个例子：新建 4 个物体，分别是 Cube，Sphere，Capsule，Cylinder，它们有不同的网格模型，但是也有相同的材质（Default-Diffuse）。
- 首先，我们不指定它们是 static 的。Draw Call 的次数是 4 次，如图：



- 我们现在将它们 4 个物体都设为 static，在来运行一下：



- 如图, Draw Call 的次数变成了 1, 而 Saved by batching 的次数变成了 3。
- 静态批处理的好处很多, 其中之一就是与下面要说的动态批处理相比, 约束要少很多。所以一般推荐的是 draw call 的静态批处理来减少 draw call 的次数。那么接下来, 我们就继续聊聊 draw call 的动态批处理。

1.16 Dynamic Batching 动态批处理

- 有阴就有阳, 有静就有动, 所以聊完了静态批处理, 肯定跟着就要说说动态批处理了。首先要明确一点, Unity3D 的 draw call 动态批处理机制是引擎自动进行的, 无需像静态批处理那样手动设置 static。我们举一个动态实例化 prefab 的例子, 如果动态物体共享相同的材质, 则引擎会自动对 draw call 优化, 也就是使用批处理。首先, 我们将一个 cube 做成 prefab, 然后再实例化 50 次, 看看 draw call 的数量。

```
for (int i = 0; i < 50; i++) {  
    GameObject cube;  
    cube = GameObject.Instantiate(prefab) as GameObject;  
}
```

- draw call 的数量:

Statistics

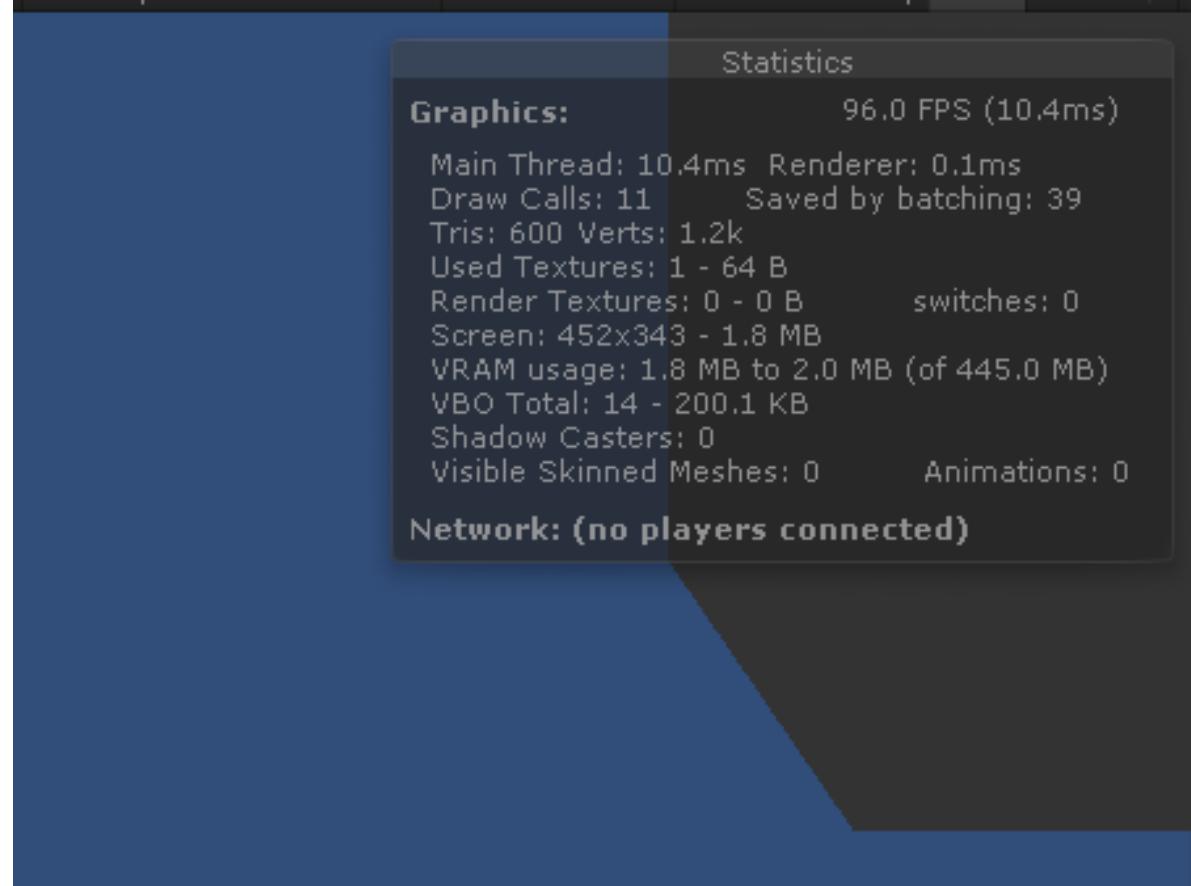
Graphics: 102.7 FPS (9.7ms)

Main Thread: 9.7ms Renderer: 0.1ms
Draw Calls: 1 Saved by batching: 49
Tris: 600 Verts: 1.2k
Used Textures: 1 - 64 B
Render Textures: 0 - 0 B switches: 0
Screen: 452x343 - 1.8 MB
VRAM usage: 1.8 MB to 2.0 MB (of 445.0 MB)
VBO Total: 14 - 200.1 KB
Shadow Casters: 0
Visible Skinned Meshes: 0 Animations: 0

Network: (no players connected)

- 可以看到 draw call 的数量为 1，而 saved by batching 的数量是 49。而这个过程中，我们除了实例化创建物体之外什么都没做。不错，unity3d 引擎为我们自动处理了这种情况。
- 但是有很多童靴也遇到这种情况，就是我也是从 prefab 实例化创建的物体，为何我的 draw call 依然很高呢？这就是匹夫上文说的，draw call 的动态批处理存在着很多约束。下面匹夫就演示一下，针对 cube 这样一个简单的物体的创建，如果稍有不慎就会造成 draw call 飞涨的情况吧。
- 我们同样是创建 50 个物体，不同的是其中的 10 个物体，每个物体的大小都不同，也就是 Scale 不同。

```
for (int i = 0; i < 50; i++) {  
    GameObject cube;  
    cube = GameObject.Instantiate(prefab) as GameObject;  
    if (i / 10 == 0) {  
        cube.transform.localScale = new Vector3(2 + i, 2 + i, 2 + i);  
    }  
}
```



2 Unity UI/GUI/UGUI/NGUI

2.1 Unity3D 快速实现 UI 架构设计一

- <https://blog.csdn.net/jxw167/article/details/72057771>

2.2 NGUI 与 UGUI 最详细对比

- <http://www.u3dc.com/archives/412>

- 1. ugui 的 ui 根目录为 canvas (画布), ngui 则是 uiroot。在命名上官方似乎更贴合想象力。
- 2. 在屏幕自适应方面, ugui 为 render mode。ngui 则为 scaling style。
- 3. anchor (锚点) 的使用方式差不多, 都是用来固定位置, 在可视化方面, ugui 的花瓣锚点真不太好调。
- 4. ngui 灵活性不是一般的高, 随意创建一个 sprite, 加了 boxcollider, 它就可以是按钮、滑动条……
- 5. ugui 的 sprite 的切图功能真心不错。ngui 使用图集不能直接拖拉 (毕竟是三方插件) 略不方便。
- 6. ngui 的 tween 动画功能很省心, 无需额外定义代码, 使用封装好的脚本就可以实现一些简单动画, 叠加脚本甚至能实现相对复杂的动画效果。
- 最后, 强大的网友分享了一张比较全面的对比图 (点击图片放大):

ngui 与 ugui 优缺点及性能比较

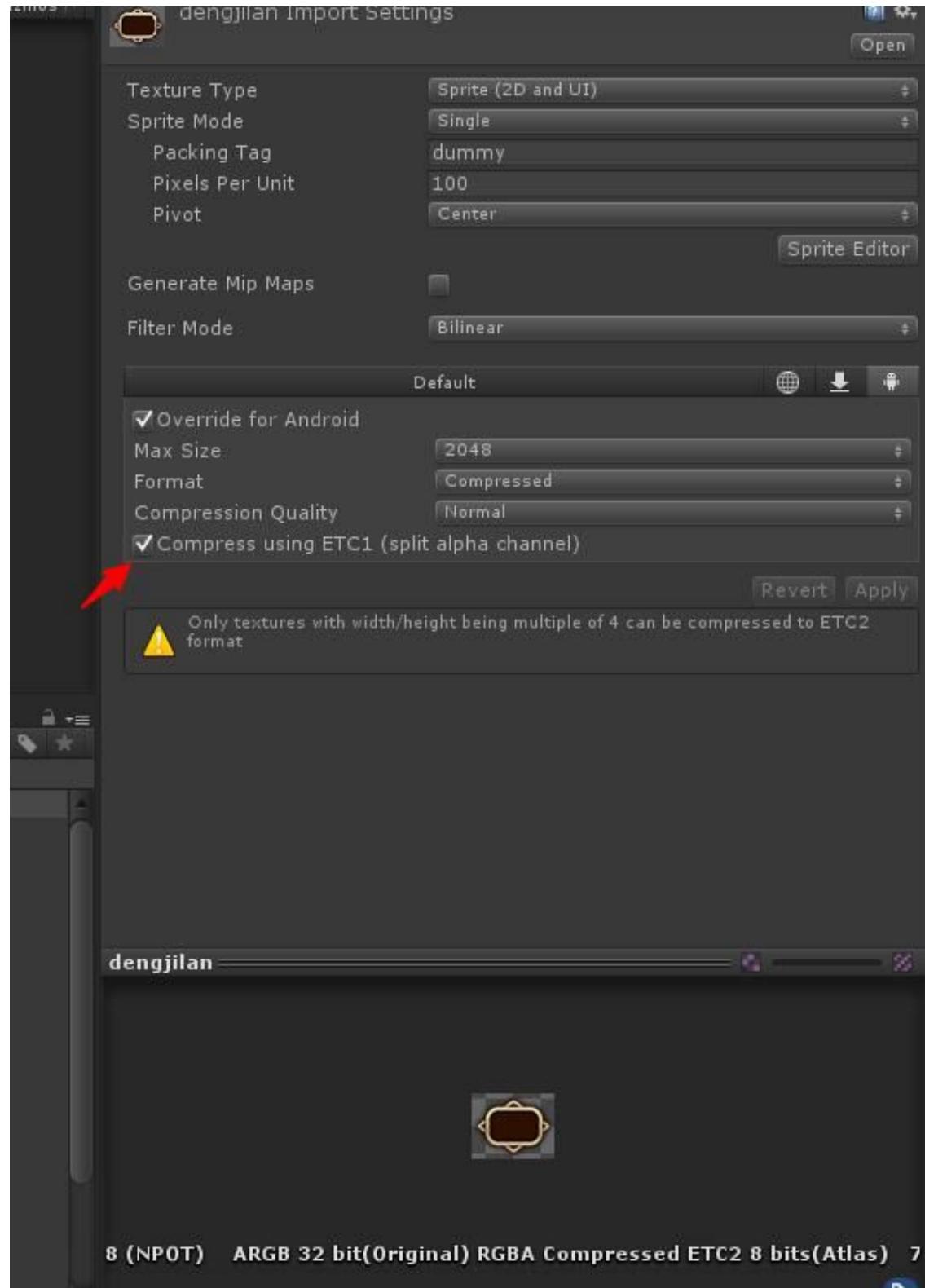
	ngui	ugui	对比结果
同一界面性能对比	1. drawcall:7~9 2. Used Textures:16.3 MB 3. VRAM usage :11.4MB~25.4MB 4. 发布后APK大小:19.8MB	1. drawcall:11~13 2. Used Textures:13.5 MB 3. VRAM usage :9.0MB~22.6MB 4. 发布后APK大小:18.5MB	同样界面(其中图片资源约为1MB), ugui的DrawCall高2个, 但是内存等其他明显减小, 安装包也小1.3MB
自适应	1. PixelPerfect 完美像素, 直接显示设定好的像素。当屏幕高度低于minimum Height时按比例缩小, 当屏幕高度大于maximum Height时按比例扩大。 2. FixedSize 按比例缩放, 在设定好的基础上, 直接按比例缩放。 3. FixedSizeOnMobiles 合体版, android和ios为FixedSize方式, 其它按照PixelPerfect方式。	1. 有三种renderMode: overlay: 始终在最前面并充满屏幕, 不可设置大小。可设置canvas层级显示的顺序。 camera: 始终充满屏幕可设置摄像机, 不可设置大小, 不一定在最前可设置层级显示的顺序。 worldSpace: 可设置大小, 可设置层级。可用于人物血条等的制作	ugui的更好用
text文本	1. 若需要的中文字体较少, 可以减少最终安装包的大小。 2. 对中文支持不好, 需要用fontMaker, BMFont等工具制作成图集, 比较麻烦。 3. 无法支持动态字体的图文混排。	1. 可直接导入ttf字体, 无需制作成图集。支持富文本格式, 方便易用。 2. 支持动态字体的图文混排。 3. 若需要的中文字体比较少, 则会加大最终安装包的大小。 4. 图文混编需要自己编写	ugui的性能更强大, 对动态字体中文等支持更好, 但是有些功能需要自己写组件
图集方面	1. 使用图片前必须要打成图集。 2. 可设置图集大小。	1. 可直接使用小图, 发布时自动打成图集, 或者你也可以自己设置图集。 2. 可设置图集大小 可以使用material	ugui的更友好, 但是无法在一个图集中动态获取其中的一个小图 无太大区别。
image			
深度排序	1、相同panel 相同atlas sprite受depth控制 2、相同panel 不同atlas sprite受z轴控制(同时受1的影响) 3、不同panel 相同atlas sprite受z轴控制(同时受1的影响) 4、不同panel 不同atlas sprite受z轴控制(同时受1的影响)	在Hierarchy视图中, 越在下方显示越靠前, 可用SetSiblingIndex 和 GetSiblingIndex等来设置深度	ugui更复杂 无太大区别, 都可使用委托等
事件系统			
Tween动画	有常用的一些缓动效果。	无	ngui自带缓动, ugui需用插件, 如 tween, tween等 

2.3 关于 Unity 中的 UGUI 优化, 你可能遇到这些问题

- https://blog.uwa4d.com/archives/QA_UGUI-1.html

2.3.1 界面制作

- UGUI 里的这个选项, 应该是 ETC2 拆分 Alpha 通道的意思, 但是在使用中并没起作用? 请问有没有什么拆分的标准和特别要求呢?



- 据我们所知，alpha split 的功能最初只对 Unity 2D 的 Sprite (SpriteRenderer) 有完整的支持，而 UI 的支持是在 Unity 5.4 版本之后的。建议大家在 Unity 5.4 版本以后的 UGUI 中尝试该功能。
2. 在 UI 界面中，用 Canvas 还是用 RectTransform 做根节点更好？哪种方法效率更高？
- Canvas 划分是个很大的话题。简单来说，因为一个 Canvas 下的所有 UI 元素都是合在一个 Mesh 中的，过大的 Mesh 在更新时开销很大，所以一般建议每个较复杂的 UI 界面，都自成一个 Canvas(可以是子 Canvas)，在 UI 界面很复杂时，甚至要划分更多的子 Canvas。同时还要注意动态元素和静态元素

的分离，因为动态元素会导致 Canvas 的 mesh 的更新。最后，Canvas 又不能细分的太多，因为会导致 Draw Call 的上升。我们后续将对 UI 模块做具体的讲解，尽请期待。

3. UWA 性能检测报告中的 Shared UI Mesh 表示什么呢？

- Shared UI Mesh 是在 Unity 5.2 版本后 UGUI 系统维护的 UI Mesh。在以前的版本中，UGUI 会为每一个 Canvas 维护一个 Mesh（名为 BatchedMesh，其中再按材质分为不同的 SubMesh）。而在 Unity 5.2 版本后，UGUI 底层引入了多线程机制，而其 Mesh 的维护也发生了改变，目前 Shared UI Mesh 作为静态全局变量，由底层直接维护，其大小与当前场景中所有激活的 UI 元素所生成的网格数相关。
- 一般来说当界面上 UI 元素较多，或者文字较多时该值都会较高，在使用 UI/Effect/shadow 和 UI/Effect/Outline 时需要注意该值，因为这两个 Effect 会明显增加文字所带来的网格数。

4. 在使用 NGUI 时，我们通常会将很多小图打成一个大的图集，以优化内存和 Draw Call。而在 UGUI 时代，UI 所使用的 Image 必须是 Sprite；Unity 提供了 SpritePacker。它的工作流程和 UGUI Atlas Paker 有较大的差别。在 Unity Asset 中，我们压根看不到图集的存在。问题是：

- 1. SpritePacker 大概的工作机制是什么样的？
- 2. 如果 Sprite 没有打包成 AssetBundle，直接在 GameObject 上引用，那么在 Build 时 Unity 会将分散的 Sprite 拼接在一起么？如果没有拼接，那 SpritePacker 是不是只会优化 Draw Call，内存占用上和不用 SpritePacker 的分离图效果一样？
- 3. 如果将 Sprite 打成 AssetBundle，AssetBundle 中的资源是分散的 Sprite 吗？如果不是，不同的 AssetBundle 中引用了两张 Sprite，这两张 Sprite 恰好用 SpritePacker 拼在了一起，是不是就会存在两份拼接的 Sprite 集？
- 4. 如果想使用 NGUI Atlas Packer 的工作流程，该如何去实现？
 - 简单来说，UGUI 和 NGUI 类似，但是更加自动化。只需要通过设定 Packing Tag 即可指定哪些 Sprite 放在同一个 Atlas 下。
 - * 可以通过 Edit -> Project Settings -> Editor -> Sprite Packer 的 Mode 来设置是否起效，何时起效（一种是进入 Play Mode 就生效，一种是 Build 时才生效）。所以只要不选 Disabled，Build 时就会把分散的 Sprite 拼起来。
 - * 可以认为 Sprite 就是一个壳子，实际上本身不包含纹理资源，所以打包的时候会把 Atlas 打进去。如果不用依赖打包，那么分开打两个 Sprite 就意味各自的 AssetBundle 里都会有一个 Atlas。
 - * 可以通过第三方工具（如 Texture Packer）制作 Atlas，导出 Sprite 信息（如，第 N 个 Sprite 的 Offset 和 Width, Height 等），然后在 Unity 中通过脚本将该 Atlas 转成一个 Multiple Mode 的 Sprite 纹理（即一张纹理上包含了多个 Sprite），同时禁用 Unity 的 Sprite Packer 即可。
 - 两种做法各有利弊，建议分析一下两种做法对于自身项目的合适程度来进行选择。

5. 在 Unity 5.x 版本下，我们在用 UGUI 的过程中发现它把图集都打进了包里，这样就不能自动更新了，请问图集怎么做自动更新呢？

- 在 Unity 5.x 中 UGUI 使用的 Atlas 确实是不可见的，因此无法直接将其独立打包。但我们建议，可以把 Packing Tag 相同的源纹理文件，打到同一个 AssetBundle 中（设置一样的 AssetBundle Name），从而避免 Atlas 的冗余。同时这样打包可以让依赖它的 Canvas 的打包更加自由，即不需要把依赖它的 Canvas 都打在一个 AssetBundle 中，在更新时直接更新 Atlas 所在的 AssetBundle 即可。

6. ScrollRect 在滚动的时候，会产生 Canvas.SendwillRenderCanvases，有办法消除吗？

- ScrollRect 在滚动时，会产生 OnTransformChanged 的开销，这是 UI 元素在移动时触发的，但通常这不会触发 Canvas.SendWillRenderCanvases。
- 如果观察到 Canvas.SendWillRenderCanvases 耗时较高，可以检查下 ScrollRect 所在的 Canvas 是否开启了 Pixel Perfect 的选项，该选项的开启会导致 UI 元素在发生位移时，其长宽会被进行微调（为了对其像素），而 ScrollRect 中通常有较多的 UI 元素，从而产生较高的 Canvas.SendWillRenderCanvases 开销。因此可以尝试关闭 Pixel Perfect 看效果是否可以接受，或者尝试在滚动过程中暂时关闭 Pixel Perfect 等方式来消除其开销。

2.3.2 网格重建

1. 我在 UGUI 里更改了 Image 的 Color 属性，那么 Canvas 是否会重建？我只想借用它的 Color 做 Animation 里的变化量。
 - 如果修改的是 Image 组件上的 Color 属性，其原理是修改顶点色，因此是会引起网格的 Rebuild 的（即 Canvas.BuildBatch 操作，同时也会有 Canvas.SendWillRenderCanvases 的开销）。而通过修改顶点色来实现 UI 元素变色的好处在于，修改顶点色可以保证其材质不变，因此不会产生额外的 Draw Call。
2. Unity 自带的 UI Shader 处理颜色时，改 *Color*？
 - 在 UI 的默认 Shader 中存在一个 Tint Color 的变量，正常情况下，该值为常数 (1,1,1)，且并不会被修改。如果是用脚本访问 Image 的 Material，并修改其上的 Tint Color 属性时，对 UI 元素产生的网格信息并没有影响，因此就不会引起网格的 Rebuild。但这样做因为修改了材质，所以会增加一个 Draw Call。
3. 能否就 UGUI Batch 提出一些建议呢？是否有一些 Batch 的规则？
 - 在 UGUI 中，Batch 是以 Canvas 为单位的，即在同一个 Canvas 下的 UI 元素最终都会被 Batch 到同一个 Mesh 中。而在 Batch 前，UGUI 会根据这些 UI 元素的材质（通常就是 Atlas）以及渲染顺序进行重排，在不改变渲染结果的前提下，尽可能将相同材质的 UI 元素合并在同一个 SubMesh 中，从而把 DrawCall 降到最低。而 Batch 的操作只会在 UI 元素发生变化时才进行，且合成的 Mesh 越大，操作的耗时也就越大。
 - 因此，我们建议尽可能把频繁变化（位置，颜色，长宽等）的 UI 元素从复杂的 Canvas 中分离出来，从而避免复杂的 Canvas 频繁重建。
4. 我用的是 UGUI Canvas，Unity 5.3.4 版本，请问如何查看每次 Rebuild Batch 影响的顶点数，Memory Profiler 是个办法但是不好定位。
 - 由于 Unity 引擎在 5.2 后开始使用 Shared UI Mesh 来存储 UI Mesh，所以确实很难查看每次 Rebuild 的 UI 顶点数。但是，研发团队可以尝试通过 Frame Debugger 工具对 UI 界面进行进一步的查看。
5. 动静分离或者多 Canvas 带来性能提升的理论基础是什么呢？如果静态部分不变动，整个 Canvas 就不刷新了？
 - 在 UGUI 中，网格的更新或重建（为了尽可能合并 UI 部分的 DrawCall）是以 Canvas 为单位的，且只在其中的 UI 元素发生变动（位置、颜色等）时才会进行。因此，将动态 UI 元素与静态 UI 元素分离后，可以将动态 UI 元素的变化所引起的网格更新或重建所涉及到的范围变小，从而降低一定的开销。而静态 UI 元素所在的 Canvas 则不会出现网格更新和重建的开销。
6. UWA 建议“尽可能将静态 UI 元素和频繁变化的动态 UI 元素分开，存放于不同的 Panel 下。同时，对于不同频率的动态元素也建议存放于不同的 Panel 中。”那么请问，如果把特效放在 Panel 里面，需要把特效拆到动态的里面吗？
 - 通常特效是指粒子系统，而粒子系统的渲染和 UI 是独立的，仅能通过 Render Order 来改变两者的渲染顺序，而粒子系统的变化并不会引起 UI 部分的重建，因此特效的放置并没有特殊的要求。
7. 多人同屏的时候，人物移动会使得头顶上的名字 Mesh 重组，从而导致较为严重的卡顿，请问一下是否有优化的办法？
 - 如果是用 UGUI 开发的，当头顶文字数量较多时，确实很容易引起性能问题，可以考虑从以下几点入手进行优化：
 - 尽可能避免使用 UI/Effect，特别是 Outline，会使得文本的 Mesh 增加 4 倍，导致 UI 重建开销明显增大；
 - 拆分 Canvas，将屏幕中所有的头顶文字进行分组，放在不同的 Canvas 下，一方面可以降低更新的频率（如果分组中没有文字移动，该组就不会重建），另一方面可以减小重建时涉及到的 Mesh 大小（重建是以 Canvas 为单位进行的）；
 - 降低移动中的文字的更新频率，可以考虑在文字移动的距离超过一个阈值时才真正进行位移，从而可以从概率上降低 Canvas 更新的频率。

2.3.3 界面切换

- 游戏中出现 UI 界面重叠，该怎么处理较好？比如当前有一个全屏显示的 UI 界面，点其中一个按钮会再起一个全屏界面，并把第一个 UI 界面盖住。我现在的做法是把被覆盖的界面 SetActive(False)，但发现后续 SetActive(True) 的时候会有 GC.Alloc 产生。这种情况下，希望既降低 Batches 又降低 GC Alloc 的话，有什么推荐的方案吗？
 - 可以尝试通过添加一个 Layer 如 OutUI，且在 Camera 的 Culling Mask 中将其取消勾选（即不渲染该 Layer）。从而在 UI 界面切换时，直接通过修改 Canvas 的 Layer 来实现“隐藏”。但需要注意事件的屏蔽，禁用动态的 UI 元素等等。
 - 这种做法的优点在于切换时基本没有开销，也不会产生多余的 Draw Call，但缺点在于“隐藏时”依然还会有一定的持续开销（通常不太大），而其对应的 Mesh 也会始终存在于内存中（通常也不太大）。
 - 以上的方式可供参考，而性能影响依旧是需要视具体情况而定。

- 如图，我们在 UI 打开或者移动到某处的时候经常会观测到 CPU 上的冲激，经过进一步观察发现是因为 Instantiate 产生了大量的 GC。想请问下 Instantiate 是否应该产生 GC 呢？我们能否通过资源制作上的调整来避免这样的 GC 呢？如下图，因为一次性产生若干 MB 的 GC 在直观感受上还是很可观的。

Hierarchy	CPU:426.49ms GPU:0.00ms	Frame Debugger											
Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms	Object	Total	Self	Calls	GC Alloc	Time ms	Self ms
BehaviourUpdate	98.7%	0.0%	1	5.6 MB	420.99	0.05	et_name	0.0%	0.0%	1	17.3 KB	0.08	0.08
GameMain.Update()	98.5%	71.1%	1	5.6 MB	420.15	303.51	et_name	0.0%	0.0%	1	17.3 KB	0.07	0.07
LogStringToConsole	8.4%	6.8%	4	200.1 KB	35.93	29.14	et_name	0.0%	0.0%	1	17.3 KB	0.05	0.05
Instantiate	8.1%	0.0%	1	2.4 MB	34.85	0.00	et_name	0.0%	0.0%	1	17.3 KB	0.06	0.06
Instantiate.Awake	3.5%	1.2%	1	1.2 MB	15.14	5.42	et_name	0.0%	0.0%	1	17.3 KB	0.05	0.05
MaskableGraphic.OnEnable	1.0%	1.0%	106	388.6 KB	4.29	4.29	et_name	0.0%	0.0%	1	17.3 KB	0.08	0.08
Text.OnEnable()	0.7%	0.7%	59	0.6 MB	3.11	3.11	et_name	0.0%	0.0%	1	17.3 KB	0.05	0.05
HyperText.OnEnable()	0.1%	0.1%	8	99.1 KB	0.75	0.75	et_name	0.0%	0.0%	1	17.3 KB	0.05	0.05
RectTransform.SendReap	0.1%	0.1%	408	0 B	0.43	0.43	et_name	0.0%	0.0%	1	17.3 KB	0.06	0.06
Animator.Initialize	0.0%	0.0%	9	0 B	0.30	0.16	et_name	0.0%	0.0%	1	17.3 KB	0.06	0.06
ContentSizeFitter.OnEnable	0.0%	0.0%	52	0 B	0.26	0.26	et_name	0.0%	0.0%	1	17.3 KB	0.05	0.05
Mask.OnEnable()	0.0%	0.0%	1	11.1 KB	0.22	0.22	et_name	0.0%	0.0%	1	17.3 KB	0.05	0.05
Selectable.OnEnable()	0.0%	0.0%	29	0.6 KB	0.11	0.11	et_name	0.0%	0.0%	1	17.3 KB	0.06	0.06
HyperTextQuad.OnEnable	0.0%	0.0%	4	11.8 KB	0.10	0.10	et_name	0.0%	0.0%	1	17.3 KB	0.05	0.05
Graphic.OnRectTransformrr	0.0%	0.0%	265	10.4 KB	0.03	0.03	et_name	0.0%	0.0%	1	17.3 KB	0.05	0.05
LayoutGroup.OnRectTran	0.0%	0.0%	2	1.4 KB	0.01	0.01	et_name	0.0%	0.0%	1	17.3 KB	0.05	0.05
UIBehaviour.Awake()	0.0%	0.0%	290	0 B	0.01	0.01	et_name	0.0%	0.0%	1	17.3 KB	0.05	0.05
BaseMeshEffect.OnEnable	0.0%	0.0%	56	0 B	0.01	0.01	et_name	0.0%	0.0%	1	17.3 KB	0.05	0.05
LayoutGroup.OnEnable()	0.0%	0.0%	2	0 B	0.00	0.00	et_text	0.0%	0.0%	1	16.7 KB	0.05	0.05
ScrollRect.OnEnable()	0.0%	0.0%	1	0 B	0.00	0.00	et_text	0.0%	0.0%	1	16.7 KB	0.04	0.04
LayoutElement.OnEnable	0.0%	0.0%	1	0 B	0.00	0.00	et_num	0.0%	0.0%	1	16.1 KB	0.06	0.06
UIA_BaseComponent.On	0.0%	0.0%	27	0 B	0.00	0.00	et_text	0.0%	0.0%	1	16.1 KB	0.04	0.04
CanvasRenderer.OnTrans	0.0%	0.0%	1667	0 B	0.00	0.00	et_text	0.0%	0.0%	1	16.1 KB	0.05	0.05
ContentSizeFitter.OnRect	0.0%	0.0%	17	0 B	0.00	0.00	et_text	0.0%	0.0%	1	16.1 KB	0.04	0.04
Selectable.Awake()	0.0%	0.0%	29	0 B	0.00	0.00	et_text	0.0%	0.0%	1	16.1 KB	0.05	0.05
ScrollRect.OnRectTransfo	0.0%	0.0%	1	0 B	0.00	0.00	et_word_01	0.0%	0.0%	1	16.1 KB	0.04	0.04
Slider.OnRectTransformD	0.0%	0.0%	2	0 B	0.00	0.00	et_cost	0.0%	0.0%	1	14.9 KB	0.03	0.03
UIBehaviour.OnRectTran	0.0%	0.0%	66	0 B	0.00	0.00	et_word	0.0%	0.0%	1	14.9 KB	0.04	0.04
Instantiate.Copy	3.0%	2.1%	1	0.8 MB	12.99	9.13	et_story	0.0%	0.0%	1	14.9 KB	0.04	0.04
Instantiate.Produce	1.5%	1.0%	1	429.5 KB	6.70	4.57	et_lock	0.0%	0.0%	1	10.2 KB	0.04	0.04

- 准确的说这些 GC Alloc 并不是由 Instantiate 直接引起的，而是因为被实例化出来的组件会进行 OnEnable 操作，而在 OnEnable 操作中产生了 GC，比如以上图中的函数为例：
- 上图中的 Text.OnEnable 是在实例化一个 UI 界面时，UI 中的文本（即 Text 组件）进行了 OnEnable 操作，其中主要是初始化文本网格的信息（每个文字所在的网格顶点，UV，顶点色等等属性），而这些信息都是储存在数组中（即堆内存中），所以文本越多，堆内存开销越大。但这是不可避免的，只能尽量减少出现次数。
- 因此，我们不建议通过 Instantiate/Destory 来处理切换频繁的 UI 界面，而是通过 SetActive(true/false)，甚至是直接移动 UI 的方式，以避免反复地造成堆内存开销。

2.3.4 加载相关

- UGUI 的图集操作中我们有这么一个问题，加载完一张图集后，使用这个方式获取其中一张图的信息：assetBundle.Load (subFile, typeof (Sprite)) as Sprite; 这样会复制出一个新贴图（图集中的子图），不知道有什么办法可以不用复制新的子图，而是直接使用图集资源。

Name	Memory	Ref count	Referenced By:
30312	256.2 KB	1	
40056	256.2 KB	1	
30308	256.2 KB	1	
SpriteAtlasTexture-512x512-fmt47	256.2 KB	19	
SpriteAtlasTexture-512x512-fmt47	256.2 KB	4	
10009	192.2 KB	4	
5100601	192.2 KB	5	
5100101	192.2 KB	4	
5100101	192.2 KB	1	
10009	192.2 KB	1	
30314	192.2 KB	1	
5100601	192.2 KB	1	
11020	192.2 KB	1	
11018	192.2 KB	1	
I18NLogo_xcqy	139.7 KB	1	
Font Texture	128.2 KB	2	
SpriteAtlasTexture-512x256-fmt47	128.2 KB	23	
SpriteAtlasTexture-512x256-fmt47	128.2 KB	1	
SpriteAtlasTexture-256x512-fmt47	128.2 KB	1	
playertipsBG	98.1 KB	1	
Bg	86.2 KB	1	
winbg	72.2 KB	1	
zhiyin5	64.2 KB	1	
zhiyin4	64.2 KB	1	
SpriteAtlasTexture-256x256-fmt47	64.2 KB	5	
SpriteAtlasTexture-256x256-fmt47	64.2 KB	1	
SpriteAtlasTexture-256x256-fmt47	64.2 KB	1	
SpriteAtlasTexture-256x256-fmt47	64.2 KB	1	
SpriteAtlasTexture-256x256-fmt47	64.2 KB	2	
SpriteAtlasTexture-256x256-fmt47	64.2 KB	14	
3_0	60.2 KB		
2_0	60.2 KB		
4_0	60.2 KB		

- 经过测试，这确实是 Unity 在 4.x 版本中的一个缺陷，理论上这张“新贴图（图集中的子图）”是不需要的，并不应该加载。因此，我们建议通过以下方法来绕过该问题：
 - 在 assetBundle.Load (subFile, typeof (Sprite)) as Sprite; 之后，调用 Texture2D t = assetBundle.Load (subFile, typeof (Texture2D)) as Texture2D;
 - Resources.UnloadAsset(t);
 - 从而卸载这部分多余的内存。

2. 加载 UI 预制的时候，如果把特效放到预制里，会导致加载非常耗时。怎么优化这个加载时间呢？

- UI 和特效（粒子系统）的加载开销在多数项目中都占据较高的 CPU 耗时。UI 界面的实例化和加载耗时主要由以下几个方面构成：
- 纹理资源加载耗时
 - UI 界面加载的主要耗时开销，因为在其资源加载过程中，时常伴有大量较大分辨率的 Atlas 纹理加载，我们在之前的 Unity 加载模块深度分析之纹理篇有详细讲解。对此，我们建议研发团队在美术质量允许的情况下，尽可能对 UI 纹理进行简化，从而加快 UI 界面的加载效率。

资源名称	生命周期(场景数)	内存占用	数量峰值	高度	宽度	格式	Mipmap 数量
Lightmap-4_comp_light	2	4.0 MB	1	1024	1024	RGB24	11
Lightmap-1_comp_light	2	4.0 MB	1	1024	1024	RGB24	11
Atlas_Alpha1	2	2.7 MB	1	1024	1024	RGBA4444	11
Ground1_B	2	1.3 MB	1	1024	1024	ETC_RGB4	11
Ground6_B	2	1.3 MB	1	1024	1024	ETC_RGB4	11
ground3	2	1.3 MB	1	1024	1024	ETC_RGB4	11
Ground5	2	1.3 MB	1	1024	1024	ETC_RGB4	11
stamps	2	1.3 MB	1	1024	1024	ETC2_RGB8	11
Atlas_Nature1	2	682.8 KB	1	1024	1024	ETC_RGB4	11
Atlas_Nature1_nmp	2	682.8 KB	1	1024	1024	ETC_RGB4	11

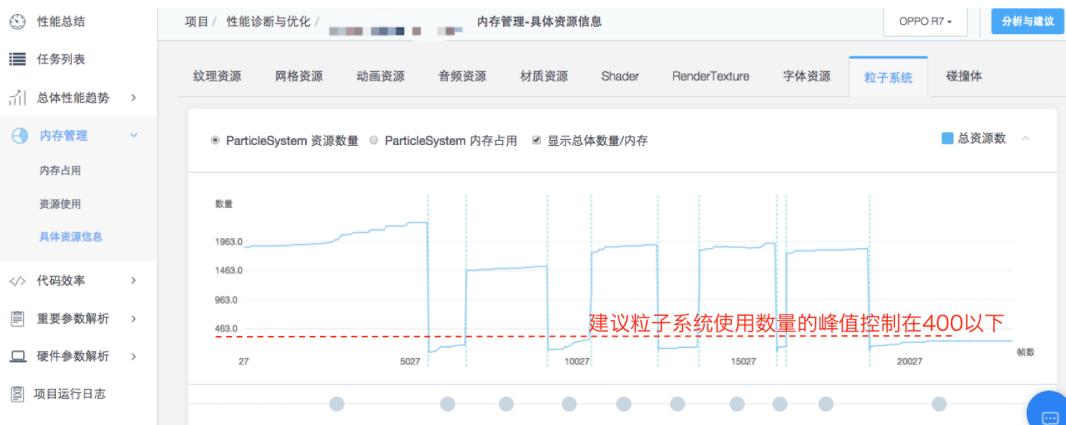
清除选中

« < 1 2 3 4 5 ... 21 > »

• UI 网格重建耗时

- UI 界面在实例化或 Active 时，往往会造成 Canvas (UGUI) 或 Panel (NGUI) 中 UIDrawCall 的变化，进而触发网格重建操作。当 Canvas 或 Panel 中网格量较大时，其重建开销也会随之较大。

- UI 相关构造函数和初始化操作开销
 - 这部分是指 UI 底层类在实例化时的 ctor 开销，以及 OnEnable 和 OnDisable 的自身开销。
- 上述 2 和 3 主要为引擎或插件的自身逻辑开销，因此，我们应该尽可能避免或降低这两个操作的发生频率。我们的建议如下：
 - 在内存允许的情况下，对于 UI 界面进行缓存。尽可能减少 UI 界面相关资源的重复加载以及相关类的重复初始化；
 - 根据 UI 界面的使用频率，使用更为合适的切换方式。比如移进移出或使用 Culling Layer 来实现 UI 界面的切换效果等，从而降低 UI 界面的加载耗时，提升切换的流畅度。
 - 对于特效（特别是粒子特效）来说，我们暂时并没有发现将 UI 界面和特效耦合在一起，其加载耗时会大于二者分别加载的耗时总和。因此，我们仅从优化粒子系统加载效率的角度来回答这个问题。粒子系统的加载开销，就目前来看，主要和其本身组件的反序列化耗时和加载数量相关。对于反序列化耗时而言，这是 Unity 引擎负责粒子系统的自身加载开销，开发者可以控制的空间并不大。对于加载数量，则是开发者需要密切关注的，因为在我们目前看到的项目中，不少都存在大量的粒子系统加载，有些项目的数量甚至超过 1000 个，如下图所示。因此，建议研发团队密切关注自身项目中粒子系统的数量使用情况。一般来说，建议我们建议粒子系统使用数量的峰值控制在 400 以下。



3. 我有一个 UI 预设，它使用了一个图集，我在打包的时候把图集和 UI 一起打成了 AssetBundle。我在加载生成了 GameObject 后立刻卸载了 AssetBundle 对象，但是当我后面再销毁 GameObject 的时候发现图集依然存在，这是什么情况呢？

- 这是很可能出现的。unload(false) 卸载 AssetBundle 并不会销毁其加载的资源，是必须对其调用 Resources.UnloadAsset，或者调用 Resources.UnloadUnusedAssets 才行。关于 AssetBundle 加载的详细解释可以参考我们之前的文章：你应该知道的 AssetBundle 管理机制。

2.3.5 字体相关

1. 我在用 Profiler 真机查看 iPhone App 时，发现第一次打开某些 UI 时，Font.CacheFontForText 占用时间超过 2s，这块主要是由什么影响的？若 iPhone5 在这个接口消耗 2s 多，是不是问题很大？这个消耗和已经生成的 RenderTexture 的大小有关吗？

- Font.CacheFontForText 主要是指生成动态字体 Font Texture 的开销，一次性打开 UI 界面中的文字越多，其开销越大。如果该项占用时间超过 2s，那么确实是挺大的，这个消耗也与已经生成的 Font Texture 有关系。简单来说，它主要是看目前 Font Texture 中是否有地方可以容下接下来的文字，如果容不下才会进行一步扩大 Font Texture，从而造成了性能开销。

3 Object Oriented Design/Programming and Design Patterns

3.1 设计模式的几个原则

- <http://www.cnblogs.com/zhaqingqing/p/4288454.html>
- 原则 1：单一职责
- 原则 2：里氏替换原则（子类扩展但不改变父类功能）
- 原则 3：依赖倒置原则

- 原则 4：接口隔离原则
- 原则 5：迪米特法则（最少知道原则）
- 原则 6：开闭原则

3.1.1 原则 1：单一职责原则

- 说到单一职责原则，很多人都会不屑一顾。
- 因为它太简单了，稍有经验的程序员即使从来没有读过设计模式、从来没有听说过单一职责原则，在设计软件时也会自觉的遵守这一重要原则，因为这是常识。
- 在软件编程中，谁也不希望因为修改了一个功能导致其他的功能发生故障。
- 而避免出现这一问题的方法便是遵循单一职责原则。
- 虽然单一职责原则如此简单，并且被认为是常识，但是即便是经验丰富的程序员写出的程序，也会有违背这一原则的代码存在。
- 为什么会出现这种现象呢？因为有职责扩散。所谓职责扩散，就是因为某种原因，职责被分化成了更细的职责。

1. 用一个类描述动物呼吸这个场景

```

1  class Animal {
2      public void breathe(string animal) {
3          Debug.Log(animal + " 呼吸空气");
4      }
5  }
6  public class Client {
7      Animal animal = new Animal();
8      void Start() {
9          animal.breathe(" 牛");
10         animal.breathe(" 羊");
11         animal.breathe(" 猪");
12     }
13 }
14 //运行结果：
15 //牛呼吸空气
16 //羊呼吸空气
17 //猪呼吸空气

```

2. 当需求变动

- 程序上线后，发现问题了，并不是所有的动物都呼吸空气的，比如鱼就是呼吸水的。
- 修改时如果遵循单一职责原则，需要将 Animal 类细分为陆生动物类 Terrestrial，水生动物 Aquatic，代码如下：

```

1  class Terrestrial {
2      public void breathe(String animal) {
3          Debug.Log(animal + " 呼吸空气");
4      }
5  }
6  class Aquatic {
7      public void breathe(String animal) {
8          Debug.Log(animal + " 呼吸水");
9      }
10 }
11 public class Client {
12     public static void main(String[] args) {
13         Terrestrial terrestrial = new Terrestrial();
14         terrestrial.breathe(" 牛");

```

```

15     terrestrial.breathe(" 羊");
16     terrestrial.breathe(" 猪");
17     Aquatic aquatic = new Aquatic();
18     aquatic.breathe(" 鱼");
19 }
20 }
21 //运行结果:
22 //牛呼吸空气
23 //羊呼吸空气
24 //猪呼吸空气
25 //鱼呼吸水

```

3. 改动量小的方法

- 我们会发现如果这样修改花销是很大的，除了将原来的类分解之外，还需要修改客户端。
- 而直接修改类 Animal 来达成目的虽然违背了单一职责原则，但花销却小的多，代码如下：

```

1 class Animal {
2     public void breathe(String animal) {
3         if (" 鱼" == animal) {
4             Debug.Log((animal + " 呼吸水"));
5         }
6         else {
7             Debug.Log((animal + " 呼吸空气"));
8         }
9     }
10 }
11 public class Client {
12     public static void main(String[] args) {
13         Animal animal = new Animal();
14         animal.breathe(" 牛");
15         animal.breathe(" 羊");
16         animal.breathe(" 猪");
17         animal.breathe(" 鱼");
18     }
19 }

```

4. 隐患

- 可以看到，这种修改方式要简单的多。
- 但是却存在着隐患：有一天需要将鱼分为呼吸淡水的鱼和呼吸海水的鱼，
- 则又需要修改 Animal 类的 breathe 方法，而对原有代码的修改会对调用“猪”“牛”“羊”等相关功能带来风险，
- 也许某一天你会发现程序运行的结果变为“牛呼吸水”了。
- 这种修改方式直接在代码级别上违背了单一职责原则，虽然修改起来最简单，但隐患却是最大的。

5. 另一种修改方式

```

1 class Animal {
2     public void breathe(String animal) {
3         Debug.Log(animal + " 呼吸空气");
4     }
5     public void breathe2(String animal) {
6         Debug.Log(animal + " 呼吸水");
7     }
8 }
9 public class Client {
10     public static void main(String[] args) {
11         Animal animal = new Animal();
12         animal.breathe(" 牛");

```

```

13     animal.breathe(" 羊");
14     animal.breathe(" 猪");
15     animal.breathe2(" 鱼");
16 }
17 }
```

- 可以看到，这种修改方式没有改动原来的方法，而是在类中新加了一个方法，这样虽然也违背了单一职责原则，
- 但在方法级别上却是符合单一职责原则的，因为它并没有动原来方法的代码。这三种方式各有优缺点，
- 那么在实际编程中，采用哪一中呢？
- 其实这真的比较难说，需要根据实际情况来确定。
- 我的原则是：只有逻辑足够简单，才可以在代码级别上违反单一职责原则；只有类中方法数量足够少，才可以在方法级别上违反单一职责原则。

6. 遵循单一职责原的优点有

- 可以降低类的复杂度，一个类只负责一项职责，其逻辑肯定要比负责多项职责简单的多；
- 提高类的可读性，提高系统的可维护性；
- 变更引起的风险降低，变更是必然的，如果单一职责原则遵守的好，当修改一个功能时，可以显著降低对其他功能的影响。
- 需要说明的一点是单一职责原则不只是面向对象编程思想所特有的，只要是模块化的程序设计，都适用单一职责原则。

3.1.2 原则 2：里氏替换原则

1. 名字的由来

- 肯定有不少人跟我刚看到这项原则的时候一样，对这个原则的名字充满疑惑。
- 其实原因就是这项原则最早是在 1988 年，由麻省理工学院的一位姓里的女士（Barbara Liskov）提出来的。
- 简单来说的话，就是当我们使用继承时，遵循里氏替换原则。

2. 定义

- 注：类 B 继承类 A 时，除添加新的方法完成新增功外，尽量不要重写父类 A 的方法，也尽量不要重载父类 A 的方法。
- 继承包含这样一层含义：父类中凡是已经实现好的方法（相对于抽象方法而言），实际上是在设定一系列的规范和契约，
- 虽然它不强制要求所有的子类必须遵从这些契约，但是如果子类对这些非抽象方法任意修改，
- 就会对整个继承体系造成破坏。而里氏替换原则就是表达了这一层含义。
- 继承作为面向对象三大特性之一，在给程序设计带来巨大便利的同时，也带来了弊端。
- 比如使用继承会给程序带来侵入性，程序的可移植性降低，增加了对象间的耦合性，如果一个类被其他的类所继承，
- 则当这个类需要修改时，必须考虑到所有的子类，并且父类修改后，
- 所有涉及到子类的功能都有可能会产生故障。

3. 继承的风险

- 那就让我们一起看看继承的风险，如下：

```

1 class A {
2     public int func1(int a, int b) {
3         return a - b;
4     }
5 }
6 public class Client {
7     void Start() {
```

```

8     A a = new A();
9     Debug.Log("100-50=" + a.func1(100, 50));
10    Debug.Log("100-80=" + a.func1(100, 80));
11 }
12 }
13 // 运行结果
14 // 100-50=50
15 // 100-80=20

```

4. 需求变动

- 后来，我们需要增加一个新的功能：完成两数相加，然后再与 100 求和，由类 B 来负责。
- 即类 B 需要完成两个功能：
- 两数相减。
- 两数相加，然后再加 100。
- 由于类 A 已经实现了第一个功能，所以类 B 继承类 A 后，只需要再完成第二个功能就可以了，代码如下

```

1 class B:A {
2     public int func1(int a, int b) {
3         return a + b;
4     }
5     public int func2(int a, int b) {
6         return func1(a, b) + 100;
7     }
8 }
9 public class Client {
10    private void Start() {
11        B b = new B();
12        Debug.Log("100-50=" + b.func1(100, 50));
13        Debug.Log("100-80=" + b.func1(100, 80));
14        Debug.Log("100+20+100=" + b.func2(100, 20));
15    }
16 }
17 // 类 B 运行结果
18 // 100-50=150
19 // 100-80=180
20 // 100+20+100=220

```

5. 影响了正常的功能

- 我们发现原本运行正常的相减功能发生了错误。
- 原因就是类 B 在给方法起名时无意中重写了父类的方法，造成所有运行相减功能的代码全部调用了类 B 重写后的方法，造成原本运行正常的功能出现了错误。
- 在本例中，引用基类 A 完成的功能，换成子类 B 之后，发生了异常。
- 在实际编程中，我们常常会通过重写父类的方法来完成新的功能，这样写起来虽然简单，
- 但是整个继承体系的可复用性会比较差，特别是运用多态比较频繁时，程序运行出错的几率非常大。
- 如果非要重写父类的方法，比较通用的做法是：原来的父类和子类都继承一个更通俗的基类，原有的继承关系去掉，采用依赖、聚合、组合等关系代替。

6. 里氏替换原则通俗的来讲就是

- 子类可以扩展父类的功能，但不能改变父类原有的功能。它包含以下 4 层含义：
- 1. 子类可以实现父类的抽象方法，但不能覆盖父类的非抽象方法。
- 2. 子类中可以增加自己特有的方法。
- 3. 当子类的方法重载父类的方法时，方法的前置条件（即方法的形参）要比父类方法的输入参数更宽松。

- 4. 当子类的方法实现父类的抽象方法时，方法的后置条件（即方法的返回值）要比父类更严格。
- 看上去很不可思议，因为我们会发现自己编程中常常会违反里氏替换原则，程序照样跑的好好的。
- 所以大家都会产生这样的疑问，假如我非要不遵循里氏替换原则会有什么后果？
- 后果就是：你写的代码出问题的几率将会大大增加。

3.1.3 原则 3：依赖倒置原则

1. 定义

- 高层模块不应该依赖低层模块，二者都应该依赖其抽象；抽象不应该依赖细节；细节应该依赖抽象。
- 以抽象为基础搭建起来的架构比以细节为基础搭建起来的架构要稳定的多。
- 抽象指的是接口或者抽象类，细节就是具体的实现类，使用接口或者抽象类的目的是制定好规范和契约，而不去涉及任何具体的操作，把展现细节的任务交给他们的实现类去完成。

2. 依赖倒置原则核心思想

- 依赖倒置原则的核心思想是面向接口编程，我们依旧用一个例子来说明面向接口编程比相对于面向实现编程好在什么地方。

3. 情景举例

- 场景是这样的，母亲给孩子讲故事，只要给她一本书，她就可以照着书给孩子讲故事了。代码如下：

```

1  class Book {
2      public String getContent() {
3          return " 很久很久以前有一个阿拉伯的故事.....";
4      }
5  }
6  class Mother {
7      public void narrate(Book book) {
8          Debug.Log(" 妈妈开始讲故事");
9          Debug.Log(book.getContent());
10     }
11 }
12 public class Client {
13     void Start() {
14         Mother mother = new Mother();
15         mother.narrate(new Book());
16     }
17 }
18 // 运行结果:
19 // 妈妈开始讲故事
20 // 很久很久以前有一个阿拉伯的故事.....

```

4. 需求变动

- 运行良好，假如有一天，需求变成这样：不是给书而是给一份报纸，让这位母亲讲一下报纸上的故事，报纸的代码如下：

```

1  class Newspaper {
2      public String getContent() {
3          return " 林书豪 38+7 领导尼克斯击败湖人.....";
4      }
5  }

```

- 这位母亲却办不到，因为她居然不会读报纸上的故事，这太荒唐了，只是将书换成报纸，居然必须要修改 Mother 才能读。
- 假如以后需求换成杂志呢？换成网页呢？
- 还要不断地修改 Mother，这显然不是好的设计。
- 原因就是 Mother 与 Book 之间的耦合性太高了，必须降低他们之间的耦合度才行。

5. 抽象的接口

- 我们引入一个抽象的接口 IReader。
- 读物，只要是带字的都属于读物：

```

1  interface IReader {
2      String getContent();
3  }

• Mother 类与接口 IReader 发生依赖关系，而 Book 和 Newspaper 都属于读物的范畴，
• 他们各自都去实现 IReader 接口，这样就符合依赖倒置原则了，代码修改为：

1  interface IReader {
2      String getContent();
3  }
4  class Newspaper : IReader {
5      public String getContent() {
6          return " 林书豪 17+9 助尼克斯击败老鹰.....";
7      }
8  }
9  class Book : IReader {
10     public String getContent() {
11         return " 很久很久以前有一个阿拉伯的故事.....";
12     }
13 }
14 class Mother {
15     public void narrate(IReader reader) {
16         Debug.Log(" 妈妈开始讲故事");
17         Debug.Log(reader.getContent());
18     }
19 }
20 public class Client {
21     public static void main(String[] args) {
22         Mother mother = new Mother();
23         mother.narrate(new Book());
24         mother.narrate(new Newspaper());
25     }
26 }
27 // 运行结果
28 // 妈妈开始讲故事
29 // 很久很久以前有一个阿拉伯的故事.....。
30 // 妈妈开始讲故事
31 // 林书豪 17+9 助尼克斯击败老鹰.....。

```

这样修改后，无论以后怎样扩展 Client 类，都不需要再修改 Mother 类了。

- 这只是一个简单的例子，实际情况中，代表高层模块的 Mother 类将负责完成主要的业务逻辑，一旦需要对它进行修改，引入错误的风险极大。
- 所以遵循依赖倒置原则可以降低类之间的耦合性，提高系统的稳定性，降低修改程序造成的风险。
- 采用依赖倒置原则给多人并行开发带来了极大的便利，
- 比如上例中，原本 Mother 类与 Book 类直接耦合时，Mother 类必须等 Book 类编码完成后才可以进行编码，因为 Mother 类依赖于 Book 类。
- 修改后的程序则可以同时开工，互不影响，因为 Mother 与 Book 类一点关系也没有。
- 参与协作开发的人越多、项目越庞大，采用依赖导致原则的意义就越重大。
- 现在很流行的 TDD 开发模式就是依赖倒置原则最成功的应用。

6. 在实际编程中，我们一般需要做到如下 3 点

- 1. 低层模块尽量都要有抽象类或接口，或者两者都有。
- 2. 变量的声明类型尽量是抽象类或接口。使用继承时遵循里氏替换原则。
- 3. 依赖倒置原则的核心就是要我们面向接口编程，理解了面向接口编程，也就理解了依赖倒置。

3.1.4 原则 4：接口隔离原则

1. 定义

- 客户端不应该依赖它不需要的接口；一个类对另一个类的依赖应该建立在最小的接口上。
- 将臃肿的接口 I 拆分为独立的几个接口，类 A 和类 C 分别与他们需要的接口建立依赖关系。也就是采用接口隔离原则。
- 举例来说明接口隔离原则：

2. 未遵循接口隔离原则的设计

- 类图 1
- 这个图的意思是：类 A 依赖接口 I 中的方法 1、方法 2、方法 3，类 B 是对类 A 依赖的实现。
- 类 C 依赖接口 I 中的方法 1、方法 4、方法 5，类 D 是对类 C 依赖的实现。
- 对于类 B 和类 D 来说，虽然他们都存在着用不到的方法（也就是图中红色字体标记的方法），但由于实现了接口 I，所以也必须要实现这些用不到的方法。

3. 示例代码

- 对类图不熟悉的可以参照程序代码来理解，代码如下：

```
1 //接口
2 interface I {
3     void method1();
4     void method2();
5     void method3();
6     void method4();
7     void method5();
8 }
9
10 class A {
11     public void depend1(I i) {
12         i.method1();
13     }
14     public void depend2(I i) {
15         i.method2();
16     }
17     public void depend3(I i) {
18         i.method3();
19     }
20 }
21
22 class B : I {
23     public void method1() {
24         Debug.Log("类 B 实现接口 I 的方法 1");
25     }
26     public void method2() {
27         Debug.Log("类 B 实现接口 I 的方法 2");
28     }
29     public void method3() {
30         Debug.Log("类 B 实现接口 I 的方法 3");
31     }
32     //对于类 B 来说，method4 和 method5 不是必需的，但是由于接口 A 中有这两个方法，
33     //所以在实现过程中即使这两个方法的方法体为空，也要将这两个没有作用的方法进行实现。
34     public void method4() { }
35     public void method5() { }
36 }
37
38 class C {
39     public void depend1(I i) { }
```

```

40     i.method1();
41 }
42     public void depend2(I i) {
43         i.method4();
44     }
45     public void depend3(I i) {
46         i.method5();
47     }
48 }

49
50 class D : I {
51     public void method1() {
52         Debug.Log(" 类 D 实现接口 I 的方法 1");
53     }
54     //对于类 D 来说, method2 和 method3 不是必需的, 但是由于接口 A 中有这两个方法,
55     //所以在实现过程中即使这两个方法的方法体为空, 也要将这两个没有作用的方法进行实现。
56     public void method2() { }
57     public void method3() { }
58     public void method4() {
59         Debug.Log(" 类 D 实现接口 I 的方法 4");
60     }
61     public void method5() {
62         Debug.Log(" 类 D 实现接口 I 的方法 5");
63     }
64 }

65
66 public class Client {
67     void Start() {
68         A a = new A();
69         a.depend1(new B());
70         a.depend2(new B());
71         a.depend3(new B());
72         C c = new C();
73         c.depend1(new D());
74         c.depend2(new D());
75         c.depend3(new D());
76     }
77 }

```

- 可以看到, 如果接口过于臃肿, 只要接口中出现的方法, 不管对依赖于它的类有没有用处, 实现类中都必须去实现这些方法, 这显然不是好的设计。
- 如果将这个设计修改为符合接口隔离原则, 就必须对接口 I 进行拆分。

4. 遵循接口隔离原则的设计

- 在这里我们将原有的接口 I 拆分为三个接口, 拆分后的设计如图 2 所示:
- 类图 2

5. 示例代码

- 照例贴出程序的代码, 供不熟悉类图的朋友参考:

```

1 interface I1 {
2     void method1();
3 }
4
5 interface I2 {
6     void method2();
7     void method3();
8 }
9

```

```

10 interface I3 {
11     void method4();
12     void method5();
13 }
14
15 class A {
16     public void depend1(I1 i) {
17         i.method1();
18     }
19     public void depend2(I2 i) {
20         i.method2();
21     }
22     public void depend3(I2 i) {
23         i.method3();
24     }
25 }
26
27 class B : I1, I2 {
28     public void method1() {
29         Debug.Log(" 类 B 实现接口 I1 的方法 1");
30     }
31     public void method2() {
32         Debug.Log(" 类 B 实现接口 I2 的方法 2");
33     }
34     public void method3() {
35         Debug.Log(" 类 B 实现接口 I2 的方法 3");
36     }
37 }
38
39 class C {
40     public void depend1(I1 i) {
41         i.method1();
42     }
43     public void depend2(I3 i) {
44         i.method4();
45     }
46     public void depend3(I3 i) {
47         i.method5();
48     }
49 }
50
51 class D : I1, I3 {
52     public void method1() {
53         Debug.Log(" 类 D 实现接口 I1 的方法 1");
54     }
55     public void method4() {
56         Debug.Log(" 类 D 实现接口 I3 的方法 4");
57     }
58     public void method5() {
59         Debug.Log(" 类 D 实现接口 I3 的方法 5");
60     }
61 }

```

- 接口隔离原则的含义是：建立单一接口，不要建立庞大臃肿的接口，尽量细化接口，接口中的方法尽量少。
- 也就是说，我们要为各个类建立专用的接口，而不要试图去建立一个很庞大的接口供所有依赖它的类去调用。
- 本文例子中，将一个庞大的接口变更为 3 个专用的接口所采用的就是接口隔离原则。
- 在程序设计中，依赖几个专用的接口要比依赖一个综合的接口更灵活。

- 接口是设计时对外部设定的“契约”，通过分散定义多个接口，可以预防外来变更的扩散，提高系统的灵活性和可维护性。
- 说到这里，很多人会觉得接口隔离原则跟之前的单一职责原则很相似，其实不然。
- 其一，单一职责原则原注重的是职责；而接口隔离原则注重对接口依赖的隔离。
- 其二，单一职责原则主要是约束类，其次才是接口和方法，它针对的是程序中的实现和细节；
- 而接口隔离原则主要约束接口，主要针对抽象，针对程序整体框架的构建。

6. 注意几点

- 采用接口隔离原则对接口进行约束时，要注意以下几点：
- 1. 接口尽量小，但是要有限度。对接口进行细化可以提高程序设计灵活性是不争的事实，但是如果过小，则会造成接口数量过多，使设计复杂化。所以一定要适度。
- 2. 为依赖接口的类定制服务，只暴露给调用的类它需要的方法，它不需要的方法则隐藏起来。只有专注地为一个模块提供定制服务，才能建立最小的依赖关系。
- 3. 提高内聚，减少对外交互。使接口用最少的方法去完成最多的事情。
- 运用接口隔离原则，一定要适度，接口设计的过大或过小都不好。设计接口的时候，只有多花些时间去思考和筹划，才能准确地实践这一原则。
-

3.1.5 原则 5：迪米特法则

1. 定义

- 一个对象应该对其他对象保持最少的了解
- 类与类之间的关系越密切，耦合度越大，当一个类发生改变时，对另一个类的影响也越大。
- 因此，尽量降低类与类之间的耦合。
- 自从我们接触编程开始，就知道了软件编程的总的原则：低耦合，高内聚。
- 无论是面向过程编程还是面向对象编程，只有使各个模块之间的耦合尽量的低，才能提高代码的复用率。
- 低耦合的优点不言而喻，但是怎么样编程才能做到低耦合呢？那正是迪米特法则要去完成的。

2. 最少知道原则

- 迪米特法则又叫最少知道原则，最早是在 1987 年由美国 Northeastern University 的 Ian Holland 提出。
- 通俗的来讲，就是一个类对自己依赖的类知道的越少越好。也就是说，对于被依赖的类来说，无论逻辑多么复杂，都尽量地将逻辑封装在类的内部，对外除了提供的 public 方法，不对外泄漏任何信息。
- 迪米特法则还有一个更简单的定义：只与直接的朋友通信。首先来解释一下什么是直接的朋友：
- 每个对象都会与其他对象有耦合关系，只要两个对象之间有耦合关系，我们就说这两个对象之间是朋友关系。
- 耦合的方式很多，依赖、关联、组合、聚合等。其中，我们称出现成员变量、方法参数、方法返回值中的类为直接的朋友，
- 而出现在局部变量中的类则不是直接的朋友。也就是说，陌生的类最好不要作为局部变量的形式出现在类的内部。

3. 违反迪米特法则的设计

- 举一个例子：有一个集团公司，下属单位有分公司和直属部门，现在要求打印出所有下属单位的员工 ID。
- 先来看一下违反迪米特法则的设计。

```

1 //总公司员工
2 class Employee {
3     private String id;
4     public void setId(String id) {
5         this.id = id;
6     }

```

```
7     public String getId() {
8         return id;
9     }
10 }
11
12 //分公司员工
13 class SubEmployee {
14     private String id;
15     public void setId(String id) {
16         this.id = id;
17     }
18     public String getId() {
19         return id;
20     }
21 }
22
23 class SubCompanyManager {
24     public List<SubEmployee> getAllEmployee() {
25         List<SubEmployee> list = new List<SubEmployee>();
26         for (int i = 0; i < 100; i++) {
27             SubEmployee emp = new SubEmployee();
28             //为分公司人员按顺序分配一个 ID
29             emp.setId(" 分公司" + i);
30             list.Add(emp);
31         }
32         return list;
33     }
34 }
35
36 class CompanyManager {
37     public List<Employee> getAllEmployee() {
38         List<Employee> list = new List<Employee>();
39         for (int i = 0; i < 30; i++) {
40             Employee emp = new Employee();
41             //为总公司人员按顺序分配一个 ID
42             emp.setId(" 总公司" + i);
43             list.Add(emp);
44         }
45         return list;
46     }
47     public void printAllEmployee(SubCompanyManager sub) {
48         List<SubEmployee> list1 = sub.getAllEmployee();
49         foreach (SubEmployee e in list1) {
50             Debug.Log(e.getId());
51         }
52         List<Employee> list2 = this.getAllEmployee();
53         foreach (Employee e in list2) {
54             Debug.Log(e.getId());
55         }
56     }
57 }
58
59 public class Client {
60     void Start() {
61         CompanyManager e = new CompanyManager();
62         e.printAllEmployee(new SubCompanyManager());
63     }
64 }
```

- 现在这个设计的主要问题出在 CompanyManager 中，根据迪米特法则，只与直接的朋友发生通信，
- 而 SubEmployee 类并不是 CompanyManager 类的直接朋友（以局部变量出现的耦合不属于直接朋友），从逻辑上讲总公司只与他的分公司耦合就行了，
- 与分公司的员工并没有任何联系，这样设计显然是增加了不必要的耦合。

4. 修改后的代码

- 按照迪米特法则，应该避免类中出现这样非直接朋友关系的耦合。修改后的代码如下：

```

1  class SubCompanyManager {
2      public List<SubEmployee> getAllEmployee() {
3          List<SubEmployee> list = new List<SubEmployee>();
4          for (int i = 0; i < 100; i++) {
5              SubEmployee emp = new SubEmployee();
6              //为分公司人员按顺序分配一个 ID
7              emp.setId(" 分公司" + i);
8              list.Add(emp);
9          }
10         return list;
11     }
12     public void printEmployee() {
13         List<SubEmployee> list = this.getAllEmployee();
14         foreach (SubEmployee e in list) {
15             Debug.Log(e.getId());
16         }
17     }
18 }
19
20 class CompanyManager {
21     public List<Employee> getAllEmployee() {
22         List<Employee> list = new List<Employee>();
23         for (int i = 0; i < 30; i++) {
24             Employee emp = new Employee();
25             //为总公司人员按顺序分配一个 ID
26             emp.setId(" 总公司" + i);
27             list.Add(emp);
28         }
29         return list;
30     }
31     public void printAllEmployee(SubCompanyManager sub) {
32         sub.printEmployee();
33         List<Employee> list2 = this.getAllEmployee();
34         foreach (Employee e in list2) {
35             Debug.Log(e.getId());
36         }
37     }
38 }
```

5. 主要的变化

- 修改后，为分公司增加了打印人员 ID 的方法，总公司直接调用该方法来打印，从而避免了与分公司的员工发生耦合。

6. 总结

- 迪米特法则的初衷是降低类之间的耦合，由于每个类都减少了不必要的依赖，因此的确可以降低耦合关系。
- 但是凡事都有度，虽然可以避免与非直接的类通信，但是要通信，必然会通过一个“中介”来发生联系，例如本例中，
- 总公司就是通过分公司这个“中介”来与分公司的员工发生联系的。

- 过分的使用迪米特原则，会产生大量这样的中介和传递类，导致系统复杂度变大。
- 所以在采用迪米特法则时要反复权衡，既做到结构清晰，又要高内聚低耦合。

3.1.6 原则 6：开闭原则

1. 定义

- 一个软件实体如类、模块和函数应该对扩展开放，对修改关闭
- 在软件的生命周期内，因为变化、升级和维护等原因需要对软件原有代码进行修改时，可能会给旧代码中引入错误，也可能使我们不得不对整个功能进行重构，并且需要原有代码经过重新测试。
- 因此，当软件需要变化时，尽量通过扩展软件实体的行为来实现变化，而不是通过修改已有的代码来实现变化。
- 闭原则是面向对象设计中最基础的设计原则，它指导我们如何建立稳定灵活的系统。开闭原则可能是设计模式六项原则中定义最模糊的一个了，
- 它只告诉我们对扩展开放，对修改关闭，可是到底如何才能做到对扩展开放，对修改关闭，并没有明确的告诉我们。
- 以前，如果有人告诉我“你进行设计的时候一定要遵守开闭原则”，我会觉得他什么都没说，但貌似又什么都说了。因为开闭原则真的太虚了。
- 在仔细思考以及仔细阅读很多设计模式的文章后，终于对开闭原则有了一点认识。
- 其实，我们遵循设计模式前面 5 大原则，以及使用 23 种设计模式的目的就是遵循开闭原则。

2. 如何遵守

- 也就是说，只要我们对前面 5 项原则遵守的好了，设计出的软件自然是符合开闭原则的，这个开闭原则更像是前面五项原则遵守程度的“平均得分”，
- 前面 5 项原则遵守的好，平均分自然就高，说明软件设计开闭原则遵守的好；
- 如果前面 5 项原则遵守的不好，则说明开闭原则遵守的不好。
- 其实，开闭原则无非就是想表达这样一层意思：用抽象构建框架，用实现扩展细节。
- 因为抽象灵活性好，适应性广，只要抽象的合理，可以基本保持软件架构的稳定。
- 而软件中易变的细节，我们用从抽象派生的实现类来进行扩展，当软件需要发生变化时，我们只需要根据需求重新派生一个实现类来扩展就可以了。
- 当然前提是我们的抽象要合理，要对需求的变更有前瞻性和预见性才行。

3.1.7 如何去遵守这六个原则

- 对这六个原则的遵守并不是是和否的问题，而是多和少的问题，也就是说，我们一般不会说有没有遵守，而是说遵守程度的多少。
- 任何事都是过犹不及，设计模式的六个设计原则也是一样，制定这六个原则的目的并不是要我们刻板的遵守他们，而需要根据实际情况灵活运用。
- 对他们的遵守程度只要在一个合理的范围内，就算是良好的设计。
- 如果大家对这六项原则的理解跟我有所不同，欢迎指正

3.2 Unity 中各种设计模式的实践与运用

- <https://blog.csdn.net/u010019717/article/details/41172783>
- <https://www.ctolib.com/Unity-Design-Pattern.html>
- 23 design patterns.

Creational Patterns	1. Abstract Factory 抽象工厂	创建几个相似的类的一个实例
	2. Builder 生成器	分离对象构造与它的表示
	3. Factory Method 工厂方法	创建几个派生类的一个实例
	4. Prototype 原型	要复制或克隆一个完全初始化的实例
	5. Singleton 单件	一个类只能运行一个实例可以存在
Structural Patterns	6. Adapter 适配器	不同的类的接口相匹配
	7. Bridge 桥接	从其实现分离对象接口
	8. Composite 复合	简单和复合对象的树形结构
	9. Decorator 装饰者	动态添加到对象的责任
	10. Facade 外观	一个表示整个子系统的单个类
	11. Flyweight 享元	细粒度的实例用于高效共享
	12. Proxy 代理服务器	一个表示另一个对象的对象
Behavioral Patterns	13. Chain of Resp. 责任链模式	一连串的对象之间传递请求的一种方式
	14. Command 命令	将命令请求封装为一个对象
	15. Interpreter 解释器	方法包含程序中的语言元素
	16. Iterator 迭代器	按顺序访问集合中的元素
	17. Mediator 中介者	定义简化的类之间的通信
	18. Memento 备忘录	捕获和还原对象的内部状态
	19. Observer 观察者	一种方式通知到类数目的变化
	20. State 状态	在其状态改变时，改变一个对象的行为
	21. Strategy 策略	封装在类的内部算法
	22. Template Method 模板方法	推迟算法到子类的确切步骤
	23. Visitor 访问者	对一类没有改变定义新的操作

3.2.1 Gang of Four Patterns in Unity (23 种 GOF 设计模式的 Unity 实现)

1. Behavioral Patterns 行为型模式 Command Pattern 命令模式

- State Pattern 状态模式
- Observer Pattern 观察者模式
- Chain of Responsibility Pattern 责任链模式
- Mediator Pattern 中介者模式
- Interpreter Pattern 解释器模式
- Iterator Pattern 迭代器模式
- Memento Pattern 备忘录模式
- Strategy Pattern 策略模式
- Template Method Pattern 模板方法模式
- Visitor Pattern 访问者模式

2. Structural Patterns 结构型模式

- Adapter Pattern 适配器模式
- Bridge Pattern 桥接模式
- Composite Pattern 组合模式
- Decorator Pattern 装饰模式
- Facade Pattern 外观模式
- Flyweight Pattern 享元模式
- Proxy Pattern 代理模式

3. Creational Patterns 创建型模式

- Prototype Pattern 原型模式
- Singleton Pattern 单例模式
- Abstract Factory Pattern 抽象工厂模式
- Builder Pattern 建造者模式
- Factory Method Pattern 工厂方法模式

3.2.2 Game Programming Patterns in Unity (《游戏编程模式》的 Unity 实现)

- Subclass Sandbox Pattern 子类沙盒模式
- Type Object Pattern 类型对象模式
- Component Pattern 组件模式
- Event Queue Pattern 事件队列模式
- Game Loop Pattern 游戏循环模式
- Service Locator Pattern 服务定位器模式
- Data Locality Pattern 数据局部性模式
- Dirty Flag Pattern 脏标记模式
- Object Pool Pattern 对象池模式

3.3 单例模式 (Unity3D/C#)

3.3.1 【单例概述】

- 定义：单例，顾名思义，单个实例，即应用单例模式的类有且只有一个实例对象，并提供一个全局访问点来供其他类与对象访问。
 - (1) 有且只有一个实例对象。
 - 有实例对象说明该类不是抽象类；只有一个实例对象表示不能随时随地的 new 一个该类的对象出来（这不是对开发者的约定，而是代码层面上的约定，即如果你这样做了，编辑器会提示你错误），即该类的构造函数是 private 级别，只能在类的内部构建实例对象；
 - (2) 提供全局访问点使其他类与对象访问。
- 这说明应用单例模式的类在提供实例访问方法时，该方法应该是静态的。
- 有了如上的分析，就可以很轻松的创建单例类了，代码并不是很复杂。为什么要分成“C# 中的单例模式”和“Unity 中的伪单例模式”，前者指 C# 语言本身，后者指在 Unity 的 Mono 框架下。还有一点要说的，下面所有所有的例子都只是说明单例如何创建，不是说类中就这么点东西。。。--

3.3.2 【C# 中的单例模式】

- 在单例模式中，根据应用情况的不同，也有着不同的实现方式。先把统一的访问方式写出来，这种访问方式应用于该命题下述的所有模式。

```
1 namespace CSharpTest {
2     class Program {
3         static void Main(string[] args) {
4             Singleton s = Singleton.GetInstance();
5         }
6     }
7 }
```

1. 饿汉模式

- 意思是，在该类装载时构建类的单例。（类的装载可以粗浅的理解为发生在程序启动时，Main 之前）也就是说，这个单例跟你什么时候用，是否要用无关，只要运行程序，这个单例就存在了。这样做的坏处是如果程序初始化时要载入的资源过多时显然这种方式又提高了加载的负担，其次如果没有使用到的话也浪费了内存。

```
1 namespace CSharpTest {
2     public class Singleton {
3         // 私有化构造函数，使得外部无法构造类的实例
4         private Singleton() { }
5     }
6 }
```

```

6 // 定义实例对象时便创建实例
7     private static Singleton _instance = new Singleton();
8
9     // 提供全局访问点
10    public static Singleton GetInstance() {
11        return _instance;
12    }
13}
14}

```

2. 懒汉模式

- 意思是，在该类的单例被使用时构造类的单例。它相比饿汉模式更加灵活，所以应用更为广泛。

(a) 基本模式（单线程模式）

- 在单线程中，只需做如下定义：

```

1 namespace CSharpTest {
2     public class Singleton {
3         //私有化构造函数 使得外部无法构造类的实例
4         private Singleton() { }
5
6         //定义一个空的单例对象
7         private static Singleton _instance;
8
9         //提供全局访问点
10        public static Singleton GetInstance() {
11            //第一次访问时会创建实例
12            if (_instance == null)
13                _instance = new Singleton();
14            return _instance;
15        }
16    }
17}

```

(b) 多线程模式

- 在多线程的程序中，构造单例的方式要发生什么变化呢？我们来依据单线程模式构造单例的代码来分析：全局访问方法内提供了如下的判断条件：

```

1 if (_instance == null)
2     _instance = new Singleton();

```

- 这会引发什么问题？以两个线程情况为例。当两个线程运行到这里时，可能线程 1 刚经过判断还没创建实例时，线程 2 就也已经通过判断要创建实例了，这会造成两个线程都创建了实例，这就违背了我们单例模式的初衷。所以我们要对其进行“加锁”，进行争用条件判断。即谁先来的谁先访问，我访问的时候你不许访问，我访问完了你再访问。

- 实现思路如下：

- 这里使用一个辅助对象（必须是引用类型）充当锁，当多个线程同时访问 GetInstance 方法时，第一个进来的锁定该对象，这时，其他线程遇到锁时会挂起等待，当这个线程执行完锁定代码块时解锁，这时第二个进来的线程再锁，解锁之后第三个进来的线程再锁。。。依次类推。这样就避免了多线程访问同一对象时会引发的风险。此例中的风险便是创建多个实例。

```

1 namespace CSharpTest {
2     public class Singleton {
3
4         //私有化构造函数 使得外部无法构造类的实例
5         private Singleton() { }
6
7         //定义一个空的单例对象
8         private static Singleton _instance;
9
10        //辅助对象
11        private static object obj = new object();

```

```

12
13     //提供全局访问点
14     public static Singleton GetInstance() {
15         //加锁，此时其他线程挂起，等待上锁的那个线程执行完事
16         lock (obj) {
17             //第一次访问时会创建实例
18             if (_instance == null)
19                 _instance = new Singleton();
20         }
21         //运行完代码块就解锁了，其他线程此时可以进入
22         return _instance;
23     }
24 }
25 }
```

这边对线程做个说明。通常我们学习编程基础时都是单线程模式。当我们开启第二条线程时，两条线程的运行是各自独立，处理各自的逻辑，他们基本上是同时运行的。可能上述例子会有个疑问，为什么可以同时通过判断而不能同时加锁呢？这涉及到两个问题。判断与锁的区别多线程的执行顺序。

- (1) 判断与锁的区别。
- 判断中，只要满足条件即可执行相应的代码块，并无其他限制；锁是当一个访问者进入锁的代码块之后马上加锁，其他访问者只能等前一个访问者出来后才能进去，当然，无论谁进去都会马上加锁。
- (2) 多线程的执行顺序。
- 这里做两个合理的猜想。一是多个线程各自独立，只是执行的快慢有微小差别，这种速度差别能使一个线程刚通过判断语句还没创建实例时，另外的线程也通过了判断语句；二是多个线程的确是同时过来的，但是在锁之前会出现顺序之分可能是底层的处理机制，因为每个线程都是有自己的标识的，当遇到琐时线程管理器会自动为多个线程分配优先顺序，保证他们有序申请锁定。

(c) 优化多线程模式

- 多线程模式主要解决的问题是当单例未创建时，多个线程同时访问 GetInstance 方法造成单例的多次创建。但现在的解决方案显然是有问题的。首先，单例没创建时，多线程是否会同时访问我们是不清楚的；在单例已经创建时，我们再去访问 GetInstance 方法时其实只需判断 `instance == null`。
- 在分析中很显然提出了解决的办法，加个判断就好了～

```

1  namespace CSharpTest {
2      public class Singleton {
3
4          //私有化构造函数 使得外部无法构造类的实例
5          private Singleton() { }
6
7          //定义一个空的单例对象
8          private static Singleton _instance;
9
10         //辅助对象
11         private static object obj = new object();
12
13         //提供全局访问点
14         public static Singleton GetInstance() {
15
16             //后续再访问时只要判断实例是否为 null 就行了
17             //不为 null 直接返回 _instance
18             //只有未创建时才会启动锁的功能
19             if (_instance == null) {
20
21                 //加锁，此时其他线程挂起，等待上锁的那个线程执行完事
22                 lock (obj)
23                 {
24                     //第一次访问时会创建实例
25                     if (_instance == null)
26                         _instance = new Singleton();
27                 }
28             }
29         }
30     }
31 }
```

```

28         //运行完代码块就解锁了，其他线程此时可以进入
29     }
30     return _instance;
31 }
32 }
33 }
```

3.3.3 【Unity 中的伪单例模式】

- 以下模式的前提都是单场景。下述的单例模式都是伪单例模式。
- Unity 实际是脚本编程，基于 Mono 框架，类默认继承自 MonoBehaviour 可以直接附加到物体上作为组件，组件所在的物体就是这个脚本类的对象，它提供了一种除了 new 之外新的对象构建方式。
- 将脚本类应用于单例模式通常是想应用例如 Update、Start 等 Message 方法，或者应用其组件化的特性在编辑器中设置脚本的成员等等。基本套路是脚本指定给物体上，获取单例使用 FindObjectOfType 方法，这也解释了为什么只能单场景使用，因为场景中的物体会随着场景变更而销毁，而脚本依附在物体上面也会被销毁。

1. 基本模式

```

1  using UnityEngine;
2  public class Singleton : MonoBehaviour {
3
4      //不写也无妨，创建继承自 MonoBehaviour 的类使不允许的
5      //虽然不会报错而是产生警告，但仍不可直接 new
6      //因为其作为组件来使用，继承关系如下
7      //Object->Component->Behaviour->MonoBehaviour->Singleton
8      private Singleton() { }
9      private static Singleton _instance;
10
11     public static Singleton GetInstance() {
12         if (_instance == null) {
13             Debug.Log("Create singleton...");
14             _instance = GameObject.FindObjectOfType<Singleton>();
15         }
16         return _instance;
17     }
18 }
```

2. 复用模式

- 可能有多个类都需要应用单例模式，它们用于处理不同的逻辑块。为每个类都写一个提供单例的创建方式显然太低效率了，那就直接写个泛型来剥离开创建单例的代码吧！

(a) 静态类

- 用来创建实例的 SingletonStatic 类：

```

1  using UnityEngine;
2
3  // FindObjectOfType 方法的泛型参数必须继承自 Object 类，所以这里对 T 要进行约束
4  public static class SingletonStatic<T> where T : MonoBehaviour {
5      private static T _instance;
6      public static T GetInstance() {
7          if (_instance == null) {
8              Debug.Log("Create " + typeof(T).ToString() + " singleton...");
9              _instance = GameObject.FindObjectOfType<T>();
10             if (_instance == null)
11                 Debug.LogError("Class of " + typeof(T).ToString() + " not found!");
12         }
13         return _instance;
14     }
15 }
```

- 需要应用单例模式的两个类， SingletonClass1 类和 SingletonClass2 类：

```

1  using UnityEngine;
2  public class SingletonClass1 : MonoBehaviour {
3      private SingletonClass1() { }
4      public int myInt = 2;
5  }
6
7  using UnityEngine;
8  public class SingletonClass2 : MonoBehaviour {
9      private SingletonClass2() { }
10     public int myInt = 5;
11 }
```

- 用来访问单例的测试类， TestClass 类：

```

1  public class TestClass : MonoBehaviour {
2      void Awake () {
3          SingletonClass1 s1 = SingletonStatic<SingletonClass1>.GetInstance();
4          SingletonClass2 s2 = SingletonStatic<SingletonClass2>.GetInstance();
5          Debug.Log(s1.myInt);
6          Debug.Log(s2.myInt);
7          Debug.Log(s1.myInt);
8          Debug.Log(s2.myInt);
9      }
10 }
```

- 除了静态类，将这三个脚本分别指定给不同的对象，运行查看 Console 面板：

- 这里写图片描述

- 可以看到两个类的单例都实例化了一次。很多人会有疑问，应用泛型会不会导致另外一个类型创建实例时会覆盖掉之前类型的实例，经过这样的测试我们发现这样的担忧完全是不必要的。

(b) 继承抽象类

- 继承抽象类的原理其实与静态类比较相似，这里直接给出父类，应用单例模式类，以及测试类的代码。

- 父类 SingletonBase 类：

```

1  using UnityEngine;
2
3  public abstract class SingletonBase<T> : MonoBehaviour where T : MonoBehaviour {
4      private static T _instance;
5      public static T GetInstance() {
6          if (_instance == null) {
7              Debug.Log("Create " + typeof(T).ToString() + " singleton...");
8              _instance = GameObject.FindObjectOfType<T>();
9              if (_instance == null)
10                  Debug.LogError("Class of " + typeof(T).ToString() + " not found!");
11          }
12          return _instance;
13      }
14 }
```

- 应用单例模式的类 SingleClass1 类：

```

1  using UnityEngine;
2  public class SingletonClass1 : SingletonBase<SingletonClass1> {
3      private SingletonClass1() { }
4
5
6  // 测试类 TestClass 类：
7  using UnityEngine;
8  public class TestClass : MonoBehaviour {
9      void Awake () {
10         SingletonClass1 s1 = SingletonClass1.GetInstance();
```

```
11 }
12 }
```

3. 为什么称为伪单例

- 假设应用单例模式的类（脚本）的名称为 SingletonClass：

(a) (一) 根本问题

- 无法避免脚本挂在多个物体上，因为 SingletonClass 会继承 MonoBehaviour 类。虽然我们在任何时候访问 SingletonClass 对象都是同一个，但是这不代表场景中这个对象是唯一的。说白了就是当脚本挂在物体上时已经是个实例了，FindObjectOfType 方法只是去找到其中一个实例，并不是在创造独一无二。每个实例都会执行 Monobehaviour 中的 Message 方法（Start、Update 这些）。
- 总结是当你同样的脚本挂在两个物体上的时候这个脚本类的对象就不唯一了，且没有办法阻止脚本挂在物体上除非不继承 MonoBehaviour 类。那既然不需要 MonoBehaviour 类，何不写成标准 C# 中的真单例模式呢？

(b) (二) 衍生问题

- 前面说到这样的伪单例只适合单场景，其实使用 Object 类的静态方法 DontDestroyOnLoad 方法可以将对象加载到内存中，只有整个程序结束的时候才会被清除。但这样做又会引发新的问题。这里做个演示，将“继承抽象类”例子中的代码修改至如下所示：

```
1 using UnityEngine;
2
3 public abstract class SingletonBase<T> : MonoBehaviour where T : MonoBehaviour {
4     private static T _instance;
5
6     public static T GetInstance() {
7         if (_instance == null) {
8             Debug.Log("Create " + typeof(T).ToString() + " singleton...");
9             _instance = GameObject.FindObjectOfType<T>();
10            //创建完实例后使其不会因场景切换被销毁
11            Object.DontDestroyOnLoad(_instance);
12            if (_instance == null)
13                Debug.LogError("Class of " + typeof(T).ToString() + " not found!");
14        }
15        return _instance;
16    }
17 }
```

- 新建一个场景，两个场景都添加个按钮，点击按钮能来回切换场景。这里单例的物体名称为 SingletonClass1，测试类所在物体叫 TestClass，该场景为 Scene1，新创建场景为 Scene2。现在我们从 Scene1 运行，点击按钮切换到 Scene2，再点击按钮切换回 Scene，资源面板显示如图所示：
- 这里写图片描述
- 出现了两个实例！这是因为我们将其加载到内存中时它已经不属于场景本身了，而场景初始化的时候会创建预制的资源，这就导致了我们再次回到场景时，出现了两个 SingletonClass1。这进一步的违背单例模式的初衷。

4. 为什么 C# 中没写“复用模式”？

- 上面介绍过，Unity 中脚本挂在物体上，我们构建所谓的单例是找到这个物体，并不是创建对象的方式；而 C# 中都是用 new 关键字创建对象的形式来构造单例。假设我们构造了这样一个泛型类，来看看具体的写法：

```
1 namespace CSharpTest {
2     public class Singleton<T> where T : new() {
3
4         //私有化构造函数 使得外部无法构造类的实例
5         private Singleton() { }
6
7         //定义实例对象时便创建实例
8         private static T _instance;
```

```

10 //提供全局访问点
11 public static T GetInstance() {
12     if (_instance == null)
13         return new T();
14     return _instance;
15 }
16 }
17 }
```

- 与最开始例子给出的代码的不同之处，除了所有的类型都写成了泛型 T 以为，还有很关键的一点，我们对 T 的类型进行了约束，约束 T 类型必须含有 public 级别的构造函数。问题就在于这里。我们为了应用单例模式的类不被随意创建，会将其构造函数设为 private 级，这就造成了冲突，导致需要应用单例模式的类无法作为该泛型类的类型参数。

3.3.4 【为什么要用单例】

- 比如我们玩游戏，游戏的目标是把三个任务都完成就可以通关。在游戏内部机制中，应该是没完成一个任务就通知管理器该任务已经完成，此时游戏管理器就是一个单例，这样当游戏管理器检测到三个任务都完成时才会通知玩家游戏通关。如果每个任务都创建了一个游戏管理器，那么这个游戏是不可能通关的——每个游戏管理器中只有一个任务被完成的记录啊！这些任务记录并没有集中到同一个游戏管理器中。此时是一定要使用单例模式的。

3.3.5 【总结】

- 举了很多例子，单例的性质已经很清晰了。但现在还有的疑问可能是，既然基继承自 MonoBehaviour 写的单例是伪单例，为什么还要一一列举出来呢？存在必有道理。MonoBehaviour 的确提供了极大地便利，很多开发者在 Unity 中都会用这样的伪单例形式，的确可以这样用，而且对于某些需求，这样做会极大的提高开发效率，但是要跟真正的单例模式区分开，不是说死记硬背一种设计模式，而是掌握其核心思想，更加安全高效的开发才是最重要的。

3.4 Unity 中常用的几种设计模式

- <https://blog.csdn.net/swj524152416/article/details/52931422>
- 23 种设计模式，实在是太多了，而且其中有一些看着还貌似差不多，让人很费解，好不容易理解了每一种设计模式的含义，并且看了一堆伪代码之后，高高兴兴的合上了书本去玩几把 LOL，赢了几把之后脑袋里关于那 23 种设计模式的概念就剩下 80% 了，然后接下来的每日工作中，基本写代码的时候也用不到啊！老板催着让你做功能，你就还哪里记得去使用设计模式啊，就开始乱写吧，日复一日，23 种设计模式基本就和你拜拜了，再见了。
- 其实呢，在游戏开发中，我们能够在 Unity 中看见的，和我们经常使用的也就是那么几种，在其他软件设计中，同样也就是经常用这么几种，那些“备忘录”模式，“责任链”模式等等，基本上不用，下面我们就说说我们常用的这几种吧：“单例模式”，“观察者模式”，“迭代器模式”，“访问者模式”。（顺便插一句，那些各种工厂模式，我就不多说了，都是很容易理解的。）

3.4.1 单例模式

- 概念很简单，保证一个类仅有一个实例，并提供一个访问它的全局访问点。我就提供两段代码就好了，在游戏当中，有两种类，一种是不继承 MonoBehaviour，另外一种是继承它的，首先看不继承的，

```

1 Class Singleton {
2     Static MySingleton;                                // 单件对象，全局唯一的。
3
4     Static Instance() {
5         if (MySingleton == null)
6             MySingleton = new MySingleton();
7         return MySingleton;
8     }          // 对外暴露接口
9 }
10
11 // 下面来看继承自 MonoBehaviour 的类，
```

```

12 Class Singleton : MonoBehavior {
13     Static MySingleton;
14     Static Instance() {
15         return MySingleton;
16     }
17     void Awake() {
18         MySingleton = this;
19     }
20 }

```

- 直接在游戏开发中这么使用就可以了。

3.4.2 观察者模式

- 概念：它将对象与对象之间创建一种依赖关系，当其中一个对象发生变化时，它会将这个变化通知给与其创建关系的对象中，实现自动化的通知更新。
- 在游戏开发中，比如 UI 上有一个下拉的 List，我选择了其中的每一项，都会导致 UI 界面的变化，比如我选择“强化”，对应出现强化装备的界面；我选择“镶嵌”，就会出现镶嵌的界面。
- 伪代码如下：

```

1 Class Subject {
2     // 对本目标绑定一个观察者 Attach( Observer );
3     // 解除一个观察者的绑定 DeleteAttach( Observer );
4     // 本目标发生了变化了，通知所有的观察者，但没有传递改动了什么
5     Notify() {
6         For ( ...遍历整个ObserverList ...) { pObserver ->Update(); }
7     }
8     // 对观察者暴露的接口，让观察者可获得本类有什么变动           GetState();
9 }
10 // 观察者/监听者类
11 Class Observer {
12     // 暴露给对象目标类的函数，当监听的对象发生了变动，则它会调用本函数通知观察者
13     Void Update () {
14         pSubject ->GetState();
15     // 获取监听对象发生了什么变化
16         TODO: DisposeFun();
17     // 根据状态不同，给予不同的处理
18     }
19 }

```

- 这个很好理解了吧！

3.4.3 迭代器模式

- 我们就拿 C# 中的迭代器为例直接说明即可，既能了解了迭代器模式的概念，有了解了 C# 中迭代器是如何实现的。
- 迭代器模式是设计模式中行为模式 (behavioral pattern) 的一个例子，他是一种简化对象间通讯的模式，也是一种非常容易理解和使用的模式。简单来说，迭代器模式使得你能够获取到序列中的所有元素而不用关心其类型是 array, list, linked list 或者是其他什么序列结构。这一点使得能够非常高效的构建数据处理通道 (data pipeline)–即数据能够进入处理通道，进行一系列的变换，或者过滤，然后得到结果。事实上，这正是 LINQ 的核心模式。
- 在.NET 中，迭代器模式被 Ienumerator 和 IEnumerable 及其对应的泛型接口所封装。如果一个类实现了 IEnumerable 接口，那么就能够被迭代；调用 GetEnumerator 方法将返回 Ienumerator 接口的实现，它就是迭代器本身。迭代器类似数据库中的游标，他是数据序列中的一个位置记录。迭代器只能向前移动，同一数据序列中可以有多个迭代器同时对数据进行操作。

- 在 C#1 中已经内建了对迭代器的支持，那就是 foreach 语句。使得能够进行比 for 循环语句更直接和简单的对集合的迭代，编译器会将 foreach 编译来调用 GetEnumerator 和 MoveNext 方法以及 Current 属性，如果对象实现了 IDisposable 接口，在迭代完成之后会释放迭代器。但是在 C#1 中，实现一个迭代器是相对来说有点繁琐的操作。C#2 使得这一工作变得大为简单，节省了实现迭代器的不少工作。

```

1 public System.Collections.IEnumerator GetEnumerator() {
2     for (int i = 0; i < 10; i++) {
3         yield return i;
4     }
5 }
6 static void Main() {
7     ListClass listClass1 = new ListClass();
8     foreach (int i in listClass1) {
9         System.Console.Write(i + " ");
10    } // Output: 0 1 2 3 4 5 6 7 8 9
11 }

```

3.4.4 访问者模式

- 当我们希望对一个结构对象添加一个功能时，我们能够在不影响结构的前提下，定义一个新的对其元素的操作。
- 例如场景管理器中管理的场景节点，是非常繁多的，而且种类不一，例如有 Ogre 中的 Root, Irrlicht 中就把摄像机，灯光，Mesh，公告版，声音都做为一种场景节点，每个节点类型是不同的，虽然大家都有共通的 Paint(), Hide() 等方法，但方法的实现形式是不同的，当我们外界调用时需要统一接口，那么我们很可能需要这样的代码

```

1 Hide (Object) {
2     if (Object == Mesh) HideMesh();
3     if (Object == Light) HideLight();
4     // ...
5 }

```

- 此时若我们需要增加一个 Object 新的类型对象，我们就不得不对该函数进行修正。而我们可以这样做，让 Mesh, Light 他们都继承于 Object，他们都实现一个函数 Hide()，那么就变成

```

1 Mesh::Hide( Visitor ) {
2     Visitor.Hide(Mesh);
3 }
4 Light::Hide(Visitor ) {
5     Visitor.Hide(Light);
6 }

```

- 意思就是说，Mesh 的隐藏可能涉及到 3 个步骤，Light 的隐藏可能涉及到 10 个步骤，这样就可以在每个自己的 Visitor 中去实现每个元素的隐藏功能，这样就把很多跟元素类本身没用的代码转移到了 Visitor 中去了。
- 每个元素类可以对应于一个或者多个 Visitor 类。比如我们去银行柜台办业务，一般情况下会开几个个人业务柜台的，你去其中任何一个柜台办理都是可以的。我们的访问者模式可以很好付诸在这个场景中：对于银行柜台来说，他们是不变的，就是说今天和明天提供个人业务的柜台是不需要有变化的。而我们作为访问者，今天来银行可能是取消费流水，明天来银行可能是去办理手机银行业务，这些是我们访问者的操作，一直是在变化的。

- 伪代码如下：

```

1 // 访问者基类
2 Class Visitor {
3     Virtual VisitElement( A ){ ... };
4     ^~I// 访问的每个对象都要写这样一个方法
5     Virtual VisitElement( B ){ ... };
6 }

```

```

8 // 访问者实例 A
9 Class VisitorA {
10     VisitElement( A ){ ... };
11     // 实际的处理函数
12     VisitElement( B ){ ... };
13     // 实际的处理函数
14 }
15
16 // 访问者实例 B
17 Class VisitorB {
18     VisitElement( A ){ ... };
19     // 实际的处理函数
20     VisitElement( B ){ ... };
21     // 实际的处理函数
22 }
23
24 // 被访问者基类
25 Class Element {
26     Virtual Accept( Visitor );
27     // 接受访问者
28 }
29
30 // 被访问者实例 A
31 Class ElementA {
32     Accecp( Visitor v ){ v-> VisitElement(this); };
33     // 调用注册到访问者中的处理函数
34 }
35
36 // 被访问者实例 B
37 Class ElementB {
38     Accecp( Visitor v ){ v-> VisitElement(this); };
39     // 调用注册到访问者中的处理函数
40 }

```

3.5 设计模式之观察模式

- <http://www.newbieol.com/information/1766.html>
- 有时被称作发布/订阅模式，观察者模式定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象，这就是观察模式。那么今天的 Unity3d 教程我们就来讲讲它。
- 在 Unity3d 中，有时被称作发布/订阅模式，观察者模式定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象。这个主题对象在状态发生变化时，会通知所有观察者对象，使它们能够自动更新自己。下面我们的 Unity3d 教程就来讲讲它：
- 解决的问题
- 将一个系统分割成一个一些类相互协作的类有一个不好的副作用，那就是需要维护相关对象间的一致性。我们不希望为了维持一致性而使各类紧密耦合，这样会给维护、扩展和重用都带来不便。观察者就是解决这类的耦合关系的。

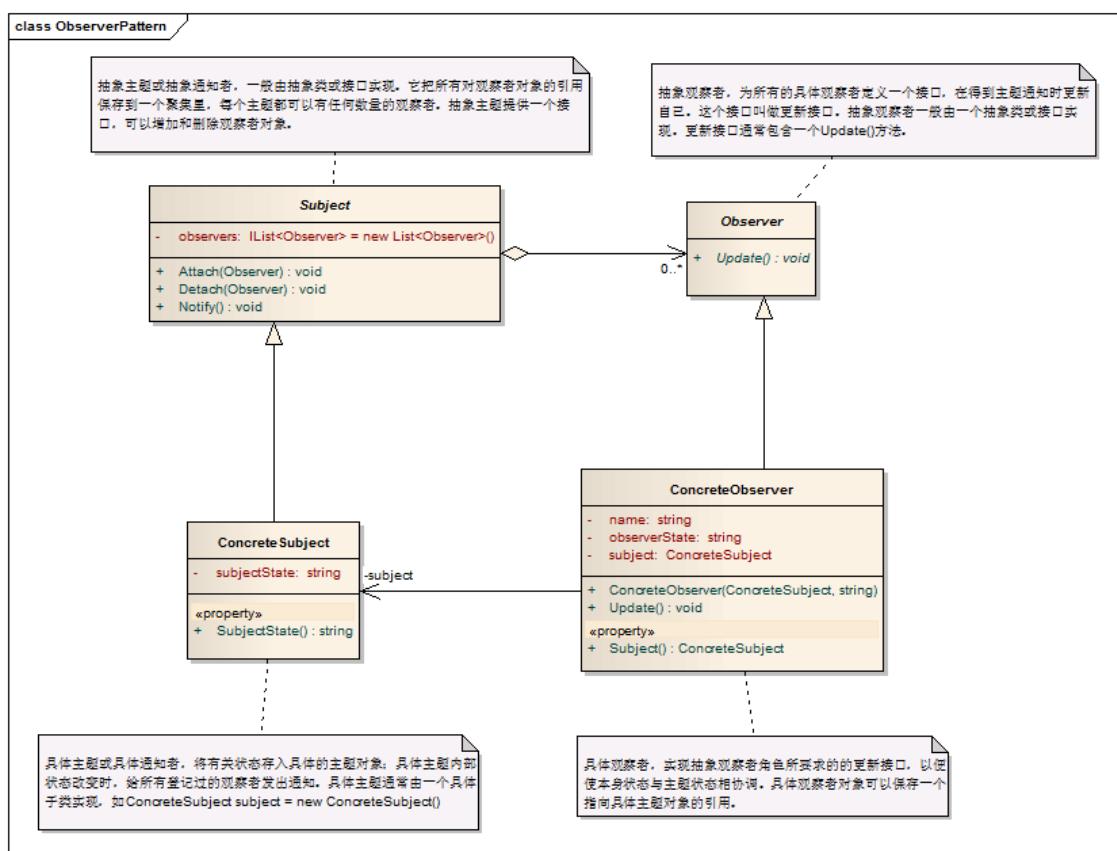
3.5.1 模式中的角色

- 抽象主题 (Subject): 它把所有观察者对象的引用保存到一个聚集里，每个主题都可以有任何数量的观察者。抽象主题提供一个接口，可以增加和删除观察者对象。
- 具体主题 (ConcreteSubject): 将有关状态存入具体观察者对象；在具体主题内部状态改变时，给所有登记过的观察者发出通知。
- 抽象观察者 (Observer): 为所有的具体观察者定义一个接口，在得到主题通知时更新自己。

- 具体观察者 (ConcreteObserver): 实现抽象观察者角色所要求的更新接口, 以便使本身的状态与主题状态协调。

3.5.2 模式解读

- 观察者模式的类图



- Unity3d 教程: 设计模式之观察模式

- 观察者模式的代码

```

1 // 抽象主题类
2 public abstract class Subject {
3     private IList observers = new List();
4     ^^^// 增加观察者
5     public void Attach(Observer observer) {
6         observers.Add(observer);
7     }
8     ^^^// 移除观察者
9     public void Detach(Observer observer) {
10        observers.Remove(observer);
11    }
12     ^^^// 向观察者(们)发出通知
13     public void Notify() {
14         foreach (Observer o in observers) {
15             o.Update();
16         }
17     }
18 }
19
20 // 抽象观察者类，为所有具体观察者定义一个接口，在得到通知时更新自己
21 public abstract class Observer {
22     public abstract void Update();

```

```

23 }
24
25 // 具体观察者或具体通知者，将有关状态存入具体观察者对象；在具体主题的内部状态改变时，给所有登记
26 public class ConcreteSubject : Subject {
27     private string subjectState;
28     ^^I// 具体观察者的状态
29     public string SubjectState {
30         get { return subjectState; }
31         set { subjectState = value; }
32     }
33 }
34
35 // 具体观察者，实现抽象观察者角色所要求的更新接口，已是本身状态与主题状态相协调
36 public class ConcreteObserver : Observer {
37     private string observerState;
38     private string name;
39     private ConcreteSubject subject;
40
41     ^^I// 具体观察者用一个具体主题来实现
42     public ConcreteSubject Subject {
43         get { return subject; }
44         set { subject = value; }
45     }
46     public ConcreteObserver(ConcreteSubject subject, string name) {
47         this.subject = subject;
48         this.name = name;
49     }
50     ^^I// 实现抽象观察者中的更新操作
51     public override void Update() {
52         observerState = subject.SubjectState;
53         Console.WriteLine("The observer's state of {0} is {1}", name, observerState);
54     }
55 }

```

- 客户端代码

```

1 class Program {
2     static void Main(string[] args) {
3         ^^I// 具体主题角色通常用具体类来实现
4         ConcreteSubject subject = new ConcreteSubject();
5         subject.Attach(new ConcreteObserver(subject, "Observer A"));
6         subject.Attach(new ConcreteObserver(subject, "Observer B"));
7         subject.Attach(new ConcreteObserver(subject, "Observer C"));
8         subject.SubjectState = "Ready";
9         subject.Notify();
10        Console.Read();
11    }
12 }

```

- 运行结果
- Unity3d 教程：设计模式之观察模式

3.5.3 模式总结

1. 优点 观察者模式解除了主题和具体观察者的耦合，让耦合的双方都依赖于抽象，而不是依赖具体。从而使得各自的变化都不会影响另一边的变化。
2. 缺点
 - 依赖关系并未完全解除，抽象通知者依旧依赖抽象的观察者。

3. 适用场景

- 当一个对象的改变需要给变其它对象时，而且它不知道具体有多少个对象有待改变时。
- 一个抽象某型有两个方面，当其中一个方面依赖于另一个方面，这时用观察者模式可以将这两者封装在独立的对象中使它们各自独立地改变和复用。
-

3.6 C# 设计模式学习笔记之建造者模式（Builder）

- <http://www.58kaifa.com/article/177>
- 建造者模式（Builder）概述：
 - 在构造一个对象的时候，往往有很多复杂的过程和次序，例如建造一个机器人，那要先造头，还是要先造身子，这就关乎到一个制造工序的问题。其实建造者模式（Builder）和工厂模式很接近的，但是建造模式提供了一个更加细粒度的对象的建造过程。
- 定义：
 - 建造者模式（Builder）将复杂的结构与其表示分离，使得同样的构建过程可以创建不同的表示。
- 原型模式应用：
 - 在软件系统中，有时候面临一个复杂对象的创建工作，该对象通常由各个部分子对象用一定的算法构成，或者按一定的步骤组合而成；这些算法和步骤是稳定的，而构成这个对象的子对象却经常由于需求的变化而不断变化。
- 假设要组装一台电脑，它的组装过程基本是不变的，都可以由主板、CPU、内存等按照某个稳定方式组合而成。然而主板、cpu、内存等零件本身是可能多变的。将内存等这种易变的零件与电脑的其他部件分离，实现解耦合，则可以轻松实现电脑不断升级。
- 建造模式结构图：
 - 建造模式参与者：
 - Builder（抽象建造者）：给出一个抽象接口，以规范产品对象的各个组成成分的建造。这个接口规定要实现复杂对象的哪些部分的创建，并不涉及具体的对象部件的创建。
 - ConcreteBuilder（建造者）：实现 Builder 接口，针对不同的商业逻辑，具体化负责对象的各部分的创建，在建造过程完成后，提供产品的实例。
 - Director（导演类）：调用具体建造者来创建复杂对象的各个部分，在导演类中不急具体产品的信息，只复杂保证对象各个部分完整创建或按某种顺序创建。
 - Product（产品类）：
 - 表示被构造的复杂对象。ConcreteBuilder 创建该产品的内部表示并定义它的装配过程
 - 包含定义组成部件的类，包括将这些部件装配成最终产品的接口
 - 在建造者模式中，Director 规定了创建一个对象所需要的步骤和次序，Builder 则提供了一些列完成这些步骤的方法，ConcreteBuilder 给出了这些方法的具体实现，是对象的直接创建者。
- 建造者模式结构实现：

```
1 // 产品类
2 public class Product {
3     private List<string> _parts = new List<string>();
4     public void Add(string part) {
5         _parts.Add(part);
6     }
7     public void Show() {
8         Console.WriteLine("Product Parts");
9         foreach (string part in _parts) {
10            Console.WriteLine(part);
```

```
11     }
12 }
13 }
14
15 // 抽象建造者类
16 public abstract class Builder {
17     public abstract void BuildPartA();
18     public abstract void BuildPartB();
19     public abstract Product GetResult();
20 }
21
22 // 建造者 1
23 public class ConcreteBuilder1 : Builder {
24     private Product _product = new Product();
25     public override void BuildPartA() {
26         _product.Add("PartA");
27     }
28     public override void BuildPartB() {
29         _product.Add("PartB");
30     }
31     public override Product GetResult() {
32         return _product;
33     }
34 }
35
36 // 建造者 2
37 public class ConcreteBuilder2 : Builder {
38     private Product _product = new Product();
39     public override void BuildPartA() {
40         _product.Add("Partx");
41     }
42     public override void BuildPartB() {
43         _product.Add("PartY");
44     }
45     public override Product GetResult() {
46         return _product;
47     }
48 }
49
50 // 导演类
51 public class Director {
52     public void Construct(Builder builder) {
53         builder.BuildPartA();
54         builder.BuildPartB();
55     }
56 }
57
58 // 客户端
59 class Client {
60     static void Main(string[] args) {
61         Director director = new Director();
62         Builder b1 = new ConcreteBuilder1();
63         Builder b2 = new ConcreteBuilder2();
64
65         director.Construct(b1);
66         Product p1 = b1.GetResult();
67         p1.Show();
68
69         director.Construct(b2);
70         Product p2 = b2.GetResult();
71         p2.Show();
72 }
```

- ```

69 }
70 }
71 }

• 建造者模式实例
• 建造小人，要求：小人必须包括，头，身体，手和脚。现在系统要包括的分为胖子和瘦子：

```

```

1 // 产品类
2 class Product {
3 private List<string> _part = new List<string>();
4 public void Add(string part) {
5 _part.Add(part);
6 }
7 public void Show() {
8 foreach (string part in _part) {
9 Console.WriteLine(part);
10 }
11 }
12 }
13 // 抽象建造者
14 public abstract class Builder {
15 public abstract void BuildHead();
16 public abstract void BuildBody();
17 public abstract void BuildHand();
18 public abstract void BuildFeet();
19 public abstract Product GetResult();
20 }
21 // 胖子建造
22 public class FatPBuilder : Builder {
23 private Product _product;
24 public override void BuildHead() {
25 _product.Add(" 胖子头");
26 }
27 public override void BuildBody() {
28 _product.Add(" 胖子身体");
29 }
30 public override void BuildHand() {
31 _product.Add(" 胖子手");
32 }
33 public override void BuildFeet() {
34 _product.Add(" 胖子脚");
35 }
36 public override Product GetResult() {
37 return _product;
38 }
39 }
40 // 瘦子建造
41 public class ThinPBuilder : Builder {
42 private Product _product;
43 public override void BuildHead() {
44 _product.Add(" 瘦子头");
45 }
46 public override void BuildBody() {
47 _product.Add(" 瘦子身体");
48 }
49 public override void BuildHand() {
50 _product.Add(" 瘦子手");
51 }
52 public override void BuildFeet() {

```

```

53 _product.Add(" 瘦子脚");
54 }
55 public override Product GetResult() {
56 return _product;
57 }
58 }
59 // 导演类
60 public class Director {
61 public void Construct(Builder builder) {
62 builder.BuildHead();
63 builder.BuildBody();
64 builder.BuildHand();
65 builder.BuildFeet();
66 }
67 }
68 // 客户端类
69 public class Client {
70 static void Main(string[] args) {
71 Director _director = new Director();
72 Builder b1 = new FatPBuilder();
73 Builder b2 = new ThinPBuilder();
74 _director.Construct(b1);
75 Product p1 = b1.GetResult();
76 p1.Show();
77 }
78 }

```

- 汽车生产

```

1 // 车辆产品类
2 public class Vehicle {
3 private string _vehicleType;
4 private Dictionary<string, string> _parts = new Dictionary<string, string>();
5
6 public Vehicle(string vehicleType) {
7 this._vehicleType = vehicleType;
8 }
9
10 public string this[string key] {
11 get { return _parts[key]; }
12 set { _parts[key] = value; }
13 }
14
15 public void Show() {
16 Console.WriteLine("\n-----");
17 Console.WriteLine("Vehicle Type:{}", _vehicleType);
18 Console.WriteLine("Frame:{0}", _parts["frame"]);
19 Console.WriteLine("Engine:{0}", _parts["engine"]);
20 Console.WriteLine("#Wheels{0}", _parts["wheels"]);
21 Console.WriteLine("#Doors:{0}", _parts["doors"]);
22 }
23 }
24
25 // 汽车制作抽象类
26 public abstract class VehicleBuilder {
27 protected Vehicle vehicle;
28 public Vehicle Vechicle {
29 get { return Vechicle; }
30 }

```

```
31 public abstract void BuildFrame();
32 public abstract void BuildEngine();
33 public abstract void BuildWheels();
34 public abstract void BuildDoors();
35 }
36
37 // 摩托车制作类
38 public class MotorCycleBuilder : VehicleBuilder {
39 public MotorCycleBuilder() {
40 vehicle = new Vehicle("MotorCycle");
41 }
42 public override void BuildFrame() {
43 vehicle["Frame"] = "MotorCycle Frame";
44 }
45 public override void BuildEngine() {
46 vehicle["engine"] = "500 cc";
47 }
48 public override void BuildWheels() {
49 vehicle["wheels"] = "2";
50 }
51 public override void BuildDoors() {
52 vehicle["doors"] = "0";
53 }
54 }
55
56 // 踏板车类制作类
57 public class ScooterBuilder : VehicleBuilder {
58 public ScooterBuilder() {
59 vehicle = new Vehicle("Scooter");
60 }
61 public override void BuildFrame() {
62 vehicle["Frame"] = "Scooter Frame";
63 }
64 public override void BuildEngine() {
65 vehicle["engine"] = "50 cc";
66 }
67 public override void BuildWheels() {
68 vehicle["wheels"] = "2";
69 }
70 public override void BuildDoors() {
71 vehicle["doors"] = "0";
72 }
73 }
74
75 // 商店类
76 public class Shop {
77 public void Construct(VehicleBuilder vb) {
78 vb.BuildFrame();
79 vb.BuildEngine();
80 vb.BuildWheels();
81 vb.BuildDoors();
82 }
83 }
84
85 // 客户端类
86 class Client {
87 static void Main(string[] args) {
88 VehicleBuilder builder;
```

```

89 Shop shop = new Shop();
90
91 builder = new ScooterBuilder();
92 shop.Construct(builder);
93 builder.Vehicle.Show();
94
95 builder = new MotorCycleBuilder();
96 shop.Construct(builder);
97 builder.Vehicle.Show();
98 }
99 }
```

## 建造者模式应用分析

- 需要生成的产品对象有复杂的内部结构
- 需要生成的产品对象的属性相互依赖，建造者模式可以强迫生成顺序
- 在对象创建过程会使用到系统中的一些其他对象，这些对象在产品对象的创建过程中不易得到
- 建造者模式特点：
- 建造者模式的使用使得产品的内部表对象可以独立地变化。使用建造者模式可以使客户不必知道产品内部组成的细节
- 每一个 builder 都相对独立，而与其他 builder 无关
- 可使对构造过程更加精细控制
- 将构建代码和表示代码分开
- 建造者模式的缺点在于难于应付分步步骤构造算法的需求变动。
- ConcreteBuilder)，用来实现抽象制造类中的操作，并进行现象内容的操作。（可定义多个，继承抽象制造者（Builder）类）
- Director），用来实现操作的流程。
- ConcreteBuilder)，在 ConcreteBuilder 中我们根据 Builder 里定义操作来进行组装，在这里我们可以在操作中定义我们 CPU 使用什么型号，主板用什么型号，内存用的时多打内存等等。

## 3.7 C# 设计模式学习笔记之工厂设计模式

- <http://www.58kaifa.com/article/171>
- 模式概述：
- 工厂方法（Factory Method）模式又成为工厂模式，属于类的创建型模式。在工厂方法模式中，父类负责定义创建对象的公共接口，子类负责生产具体的对象，这样做的目的是将类的实例化操作延迟到子类中完成，即由子类决定究竟应该实例化哪个类。
- 定义：
- 工厂方法模式定义一个永远创建对象的接口，让子类决定实例化哪一个类。工厂方法模式是以一个类的实例化延迟到其子类。
- Factory Method 模式用于在不指定待创建对象的具体类的情况下创建对象。
- Factory Method 模式的主要意图是隐藏对象创建的责任性。Client 通常不指定要创建的具体类，Client 将面向接口或抽象类进行编码，让 Factory 类负责创建具体的类型。通常 Factory 类有一个返回抽象类或接口的静态方法。Client 通常提供某种信息让 Factory 类使用提供的信息来确定创建并返回哪个子类。
- 将创建子类的责任抽象出来的好处是允许 Client 完成无需考虑依赖类是如何创建。这遵守依赖倒置原则（DIP）。Factory Method 模式另外一个好处是把负责对象创建的代码集中起来，如果需要修改对象生产方式，可以轻松定位并更新，而不会影响到依赖它的代码。

• 在面向对象编程中，一般方法是用一个 new 操作符产生一个对象的实例。但是在一些情况下，用 new 操作符直接生产对象会带来一些问题。首先，要使用 new 运算符创建一个对象必须清楚所要创建的对象的类信息，包括类名、构造函数等，而又时并不现实。其次许多类型的对象创建需要一系列的步骤，可能需要计算或取得对系那个的初始设置，选择生产那个对象实例，或在需要的对象之前必须生产一些辅助功能的对象。在这些情况下，新对象的创建就是一个过程，而不是一个简单的操作。

- 工厂方法模式结构：

- 工厂模式参与者：

- Product: 抽象的产品角色，定义工厂方法所创建的对象接口

- ConcreteProduct: 具体 Product 角色，实现 Product 接口

- Factory: 抽象的工厂角色，声明工厂方法，返回一个 Product 类型的对象

- Factory 可以定义一个工厂方法的默认实现，返回一个默认的 ConcreteProduct 对象。可以调用工厂方法创建一个 Product 对象。

- ConcreteFactory: 具体的工厂角色，创建具体 Product 的子工厂，重写工厂方法以返回一个 ConcreteProduct 实例。

- 工厂方法模式结构实现：

```
1 // 工厂模式结构实现
2 // 抽象产品类
3 public class Product {
4 }
5 // 具体产品类 A
6 public class ConcreteProductA : Product {
7 }
8 // 具体产品类 B
9 public class ConcreteProductB : Product {
10 }
11 // 抽象工厂类
12 public abstract class Factory {
13 public abstract Product CreateProduct();
14 }
15 // 具体工厂 A
16 public class ConcreteFactoryA : Factory {
17 public override Product CreateProduct() {
18 return new ConcreteProductA();
19 }
20 }
21 // 具体工厂 B
22 public class ConcreteFactoryB : Factory {
23 public override Product CreateProduct() {
24 return new ConcreteProductB();
25 throw new NotImplementedException();
26 }
27 }
28 // 客户端类
29 public class Client {
30 static void Main(string[] args) {
31 Factory[] factories = new Factory[2];
32 factories[0] = new ConcreteFactoryA();
33 factories[1] = new ConcreteFactoryB();
34
35 foreach (Factory factory in factories) {
36 Product product = factory.CreateProduct();
37 Console.WriteLine("Created{0}", product.GetType().Name);
38 }
39 }
```

```
39 }
40 }
41 // 工厂模式应用 1：扩展刷卡处理
42 // 在简单工厂模式应用中我们谢了刷卡处理的应用，但是突然公司要添加新的卡种，
43 // 就可以很使用工厂模式来做，只要增加卡的处理类和生产卡处理类的工厂。
44
45 // 抽象产品类
46 abstract class BankCardHandle {
47 public abstract void HandleProcess();
48 }
49 // 具体产品类 VISA，继承抽象产品类
50 class VisaHandle : BankCardHandle {
51 public override void HandleProcess() {
52 Console.WriteLine("Visa 卡处理中..");
53 }
54 }
55 // 具体产品类 Master，继承抽象产品类
56 class MasterCardHandle : BankCardHandle {
57 public override void HandleProcess() {
58 Console.WriteLine("Master 卡处理中..");
59 }
60 }
61 // 抽象工厂类
62 public abstract class HandleFactory {
63 public abstract BankCardHandle CreateBankCardHandle();
64 }
65 // 具体工厂类，继承抽象工厂类
66 public class VisaFactory:HandleFactory {
67 public override BankCardHandle CreateBackCardHandle() {
68 return new VisaHandle();
69 }
70 }
71 // 具体工厂类，继承抽象工厂类
72 public class MasterFactory:HandleFactory {
73 public override BankCardHandle CreateBackCarHandle() {
74 return new BankCardHandle();
75 }
76 }
77 // 客户端调用
78 class Client {
79 public static void Main(string[] args) {
80 //实例化工厂
81 HandleFactory visaFacotry = new VisaFactory();
82 HandleFactory masterFactory = new MasterFactory();
83 //创建卡类
84 BankCardHandle vf = visaFacotry.CreateBackCarHandle();
85 BankCardHandle mf = masterFactory.CreateBackCarHandle();
86 //新添加的卡种
87 HandleFactory unionFactory = new UnionPayCardFactory();
88 BankCardHandle nf = unionFactory.CreateBackCarHandle();
89 }
90 }
91 // 新添加的卡的特性
92 public class UnionPayCardHandle : BankCardHandle {
93 public override void HandleProcess() {
94 Console.WriteLine(" 银联卡处理中..");
95 }
96 }
```

```
97 // 新添加的卡的工厂
98 public class UnionPayCardFactory : HandleFactory {
99 public override BankCardHandle CreateBankCardHandle() {
100 return new UnionPayCardHandle();
101 }
102 }
```

- 工厂模式应用：
- 扩展刷卡应用

```
1 // KFC 生产:
2 // 应用实例: KFC 生产模式
3 ///抽象产品类
4 public abstract class KFCFood {
5 public abstract void Display();
6 }
7 // 具体产品类 鸡腿
8 public class Chicken : KFCFood {
9 public override void Display() {
10 Console.WriteLine(" 鸡腿 +1");
11 }
12 }
13 // 具体产品类鸡翅
14 public class Wings : KFCFood {
15 public override void Display() {
16 Console.WriteLine(" 鸡翅 +1");
17 }
18 }
19 // 抽象工厂类
20 public abstract/interface class KFCFactory {
21 public abstract KFCFood CreateFood();
22 }
23 }
24 // 具体工厂类 鸡腿工厂
25 public class ChikenFactory:KFCFactory {
26 public override KFCFood CreateFood() {
27 return new Chicken();
28 }
29 }
30 // 具体工厂类 鸡翅工厂
31 public class WingsFactory:KFCFactory {
32 public override KFCFood CreateFood() {
33 return new Wings();
34 }
35 }
36 // 客户端类
37 public class Client {
38 public static void Main(string[] args) {
39 //创建鸡腿和鸡翅工厂
40 KFCFactory chichen = new KFCFactory();
41 KFCFactory wings = new KFCFactory();
42 //生产鸡腿
43 KFCFood food1 = chichen.CreateFood();
44 food1.Display();
45
46 KFCFood food2 = chichen.CreateFood();
47 food1.Display();
48 //生产鸡翅
49 KFCFood food3 = wings.CreateFood();
```

- ```

50         food3.Display();
51     }
52 }

• 工厂模式方法应用分析:
• 1、工厂模式使用情形:
• 当一个类不知道它所必须创建的对象的类信息的时候
• 当一个类希望由它来指定它所创建的对象的时候
• 当类将创建对象的职责委托给多个辅助子类中的某一个，希望将哪一个辅助之类时代理者这以信息局部化的时候
• 2、工厂模式特点
• 使用工厂方法在一个类的内部创建对象通常比直接创建对象更灵活
• 工厂方法模式通过面向对象的手法，将所要创建的具体对象的创建工作延迟到子类，从而提供了一种扩展的策略，较好的解决了紧耦合的关系
• 工厂方法模式遵守依赖倒置原则 (DIP);
• 总结:
• 相对于简单工厂模式而言，工厂方法模式的核心是一个抽象工厂类，而简单工厂模式把核心放在一个具体工厂类上。在工厂模式中，子工厂与产品往往具有平行的等级结构，他们之间一一对应。
• 就上一节玩具工厂来说，我们知道有一台机器可以设定多个模式，那么可以多买几台机器，每台机器的设定不一样，比如，这一是生产小熊用的，另一台是生产汽车用的，当我们需要那种玩具的时候就用相对的机器进行生产。可以工厂模式就是多个简单工厂模式的综合。

```

4 Camera 相关

4.1 Camera Shake in Unity

- <http://www voidcn com/article/p-yfmjbkxr-pu.html>

```

1 var originPosition:Vector3;
2 var originRotation:Quaternion;
3 var shake_decay: float;
4 var shake_intensity: float;;
5 function OnGUI () {
6     if (GUI.Button (Rect (20,40,80,20), "Shake")) {
7         Shake();
8     }
9 }
10 function Update(){
11     if(shake_intensity > 0){
12         transform.position = originPosition + Random.insideUnitSphere * shake_intensity;
13         transform.rotation = Quaternion(
14             originRotation.x + Random.Range(-shake_intensity,shake_intensity)*.2,
15             originRotation.y + Random.Range(-shake_intensity,shake_intensity)*.2,
16             originRotation.z + Random.Range(-shake_intensity,shake_intensity)*.2,
17             originRotation.w + Random.Range(-shake_intensity,shake_intensity)*.2);
18         shake_intensity -= shake_decay;
19     }
20 }
21 function Shake(){
22     originPosition = transform.position;
23     originRotation = transform.rotation;
24     shake_intensity = .3;
25     shake_decay = 0.002;
26 }

```

4.2 Unity 中做放大镜效果

- <http://www.voidcn.com/article/p-qyltseeo-ry.html>
- 其实和小地图都差不多了。还是要借助另一个相机
- 目的: 这篇文章的主要目的是要给你一个想法如何做放大镜效果。
- 在 unity 中可以简单的实现放大镜效果啊. 那么现在就来一步一步实现这个:
- 创建一个摄像机对象, 设置 projection 类型为 perspective 或者 orthographic.
- 设置相机的 orthographicSize 或者 fieldOfView (依赖于相机的 projection 类型).
- 设置其 pixelrect . 例如如果您想要在你鼠标位置显示放大镜和其大小是 100 x 100 , 然后设置 pixelrect 为:

```
magnifyCamera.pixelRect = new Rect (Input.mousePosition.x - 100f / 2.0f, Input.mousePosition.y
```

- 设置相机的位置。例如如果你想在你的鼠标位置显示放大镜效果, 那么设置相机的位置为 mousePosition 世界点。
- 下面的 C# 脚本将创建一个 MagnifyGlass, 并将它移动到 mousePosition 位置。
- MagnifyGlass 脚本: 添加到一个空的游戏对象。

```
1  using UnityEngine;
2  using System.Collections;
3
4  public class MagnifyGlass : MonoBehaviour {
5      private Camera magnifyCamera;
6      private GameObject magnifyBorders;
7      private LineRenderer LeftBorder, RightBorder, TopBorder, BottomBorder; // Reference for borders
8      private float MGOX, MGOY; // Magnify Glass Origin X and Y position
9      private float MGWidth = Screen.width/5f, MGHeight = Screen.height/5f; // Magnify glass size
10     private Vector3 mousePos;
11
12     void Start () {
13         createMagnifyGlass ();
14     }
15
16     void Update () {
17         // Following lines set the camera's pixelRect and camera position at mouse position
18         magnifyCamera.pixelRect = new Rect (Input.mousePosition.x - MGWidth / 2.0f, Input.mousePosition.y - MGHeight / 2.0f, MGWidth, MGHeight);
19         mousePos = getWorldPosition (Input.mousePosition);
20         magnifyCamera.transform.position = mousePos;
21         mousePos.z = 0;
22         magnifyBorders.transform.position = mousePos;
23     }
24
25     // Following method creates MagnifyGlass
26     private void createMagnifyGlass() {
27         GameObject camera = new GameObject("MagnifyCamera");
28         MGOX = Screen.width / 2f - MGWidth/2f;
29         MGOY = Screen.height / 2f - MGHeight/2f;
30         magnifyCamera = camera.AddComponent<Camera>();
31         magnifyCamera.pixelRect = new Rect(MGOX, MGOY, MGWidth, MGHeight);
32         magnifyCamera.transform.position = new Vector3(0,0,0);
33         if (Camera.main.isOrthoGraphic) {
34             magnifyCamera.orthographic = true;
35             magnifyCamera.orthographicSize = Camera.main.orthographicSize / 5.0f; // + 1.0f;
36             createBordersForMagnifyGlass ();
37         } else {
38             magnifyCamera.orthographic = false;
```

```

39     magnifyCamera.fieldOfView = Camera.main.fieldOfView / 10.0f; //3.0f;
40 }
41 }
42
43 // Following method sets border of MagnifyGlass
44 private void createBordersForMagnifyGlass() {
45     magnifyBorders = new GameObject ();
46     LeftBorder = getLine ();
47     LeftBorder.SetVertexCount(2);
48     LeftBorder.SetPosition(0,
49         new Vector3(getWorldPosition(new Vector3(MGOX,MGOY,0)).x,
50             getWorldPosition(new Vector3(MGOX,MGOY,0)).y-0.1f,
51             -1));
52     LeftBorder.SetPosition(1,
53         new Vector3(getWorldPosition(new Vector3(MGOX,MGOY+MGHeight,0)).x,
54             getWorldPosition(new Vector3(MGOX,MGOY+MGHeight,0)).y+0.1f,
55             -1));
56     LeftBorder.transform.parent = magnifyBorders.transform;
57     TopBorder = getLine ();
58     TopBorder.SetVertexCount(2);
59     TopBorder.SetPosition(0,
60         new Vector3(getWorldPosition(new Vector3(MGOX,MGOY+MGHeight,0)).x,
61             getWorldPosition(new Vector3(MGOX,MGOY+MGHeight,0)).y,
62             -1));
63     TopBorder.SetPosition(1,
64         new Vector3(getWorldPosition(new Vector3(MGOX+MGWidth,MGOY+MGHeight,0)).x,
65             getWorldPosition(new Vector3(MGOX+MGWidth,MGOY+MGHeight,0)).y,
66             -1));
67     TopBorder.transform.parent = magnifyBorders.transform;
68     RightBorder = getLine ();
69     RightBorder.SetVertexCount(2);
70     RightBorder.SetPosition(0,
71         new Vector3(getWorldPosition(new Vector3(MGOX+MGWidth,MGOY+MGWidth,0)).x,
72             getWorldPosition(new Vector3(MGOX+MGWidth,MGOY+MGWidth,0)).y+0.1f,
73             -1));
74     RightBorder.SetPosition(1,
75         new Vector3(getWorldPosition(new Vector3(MGOX+MGWidth,MGOY,0)).x,
76             getWorldPosition(new Vector3(MGOX+MGWidth,MGOY,0)).y-0.1f,
77             -1));
78     RightBorder.transform.parent = magnifyBorders.transform;
79     BottomBorder = getLine ();
80     BottomBorder.SetVertexCount(2);
81     BottomBorder.SetPosition(0,
82         new Vector3(getWorldPosition(new Vector3(MGOX+MGWidth,MGOY,0)).x,
83             getWorldPosition(new Vector3(MGOX+MGWidth,MGOY,0)).y,
84             -1));
85     BottomBorder.SetPosition(1,
86         new Vector3(getWorldPosition(new Vector3(MGOX,MGOY,0)).x,
87             getWorldPosition(new Vector3(MGOX,MGOY,0)).y,
88             -1));
89     BottomBorder.transform.parent = magnifyBorders.transform;
90 }
91
92 // Following method creates new line for MagnifyGlass's border
93 private LineRenderer getLine() {
94     LineRenderer line = new GameObject("Line").AddComponent<LineRenderer>();
95     line.material = new Material(Shader.Find("Diffuse"));
96     line.SetVertexCount(2);

```

```

97     line.setWidth(0.2f, 0.2f);
98     line.setColors(Color.black, Color.black);
99     line.useWorldSpace = false;
100    return line;
101 }
102 private void setLine(LineRenderer line) {
103     line.material = new Material(Shader.Find("Diffuse"));
104     line.SetVertexCount(2);
105     line.setWidth(0.2f, 0.2f);
106     line.setColors(Color.black, Color.black);
107     line.useWorldSpace = false;
108 }
109
110 // Following method calculates world's point from screen point as per camera's projection
111 public Vector3 getWorldPosition(Vector3 screenPos) {
112     Vector3 worldPos;
113     if(Camera.main.isOrthographic) {
114         worldPos = Camera.main.ScreenToWorldPoint (screenPos);
115         worldPos.z = Camera.main.transform.position.z;
116     } else {
117         worldPos = Camera.main.ScreenToWorldPoint (new Vector3 (screenPos.x, screenPos.y, screenPos.z));
118         worldPos.x *= -1;
119         worldPos.y *= -1;
120     }
121     return worldPos;
122 }
123 }
```

4.3 Unity 3D— 摄像机 Clear Flags 和 Culling Mask 属性用途详解

- <http://www.voidcn.com/article/p-pwbltogg-pu.html>

4.4 [Unity 基础] 对 Camera 组件属性的一些理解

- <http://www.voidcn.com/article/p-gtfuejmb-uz.html>

4.5 Unity 的 camera 组件

- <http://www.voidcn.com/article/p-mcxkifby-cb.html>

4.6 unity Camera 属性详解

- <http://www.voidcn.com/article/p-enwwkrfr-hh.html>

5 Lights Lighting related

5.1 渲染管线中的 Rendering Path

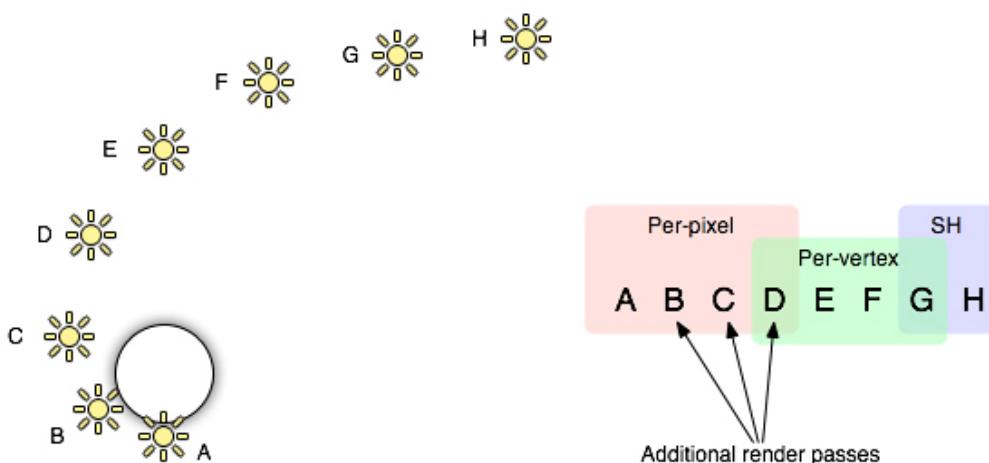
- <https://yq.aliyun.com/articles/49406>
- Unity3D 中，目前保存了 4 种 Rending Path.
 - 1、Deferred Shading Rendering Path(延迟渲染)
 - 2、Forward Rendering Path(前向渲染)
 - 3、Lagacy Deferred Lighting Rendering Path(延迟光照)
 - 4、Vertex Lit Rendering Path(顶点光照)

5.1.1 Forward Rendering Path(前向渲染)

- 官方文档: <http://docs.unity3d.com/Manual/RenderTech-ForwardRendering.html>
- 传统的前向着色渲染方式是一种简单粗暴的方式。就是一次性将一个物体绘制完再绘制下一个。光照计算也是如此。前向着色最主要的问题就是光照问题。当光源过多的时候，就需要进行处理。而 Unity3D 在光源处理方面，也结合了像素光照 (per-pixel lighting)，顶点光照 (per-vertex lighting)，球谐光照 (Spherical Harmonics lighting) 等多种方案，其复杂程度可见一斑。

1. Unity3D 中前向渲染的光源策略

- 前向着色渲染由于效率和寄存器限制等原因。不可能老老实实地去处理所有的光源。因此，需要根据光源的远近，范围，重要程度等因子决定计算方式。
 - 1、如果一个光源的 RenderMode 没有设置为重要 (Important)，那么它将永远是逐顶点或者球谐光照计算方式。
 - 2、最亮的 (Brightest) 方向光源会做为逐像素光照计算。
 - 3、如果一个光源的 RenderMode 设置为重要 (Important)，那么它将会采用逐像素光照计算方式
 - 4、如果上面的结果中的用于逐像素计算的光源数目小于了 Project Settings->Quality 面板中的 Pixel Light Count 值。为了减少亮度误差，有一些光源会被当作像素光源计算 (PS: 非像素光源无法完全计算阴影关系。会导致场景偏亮)。
 - 5、Unity3D 对顶点光照，像素光照做了最大限制。分别为 4 个。而场景中不可能只有这么多光源。因此，使用了一个优先级判定。



– 左：光源位置图

– 右：光源分配图

- 从图中我们可以看到。D 参与了像素和顶点光照。G 参与了顶点光照和球谐光照。这样的处理应该是为了做一个光照信息的平滑衔接，不至于变得很突兀。

2. Unity3D 中前向渲染的光照的计算过程

- 前向渲染中，光照计算被分成了多个 PASS。通常它们由 1*FarwardBase Pass + N*FarwardAdd Pass 组成。可以看出，光源的数目在前向渲染管线中是一个致命的效率杀手。
 - 1、Base Pass 提交一个逐像素方向光和所有的逐顶点光照和球谐光照。
 - 2、Additional Pass 提交一个逐像素的光照计算。如果一个物体受更多的光源影响。那么就会有许多 PASS 需要进行绘制。
 - * 注：在 Shader 中。可以指定 FarwardBase 来关掉 Additional 功能。
 - * 注：在前向渲染中，只能有一个方向光实时阴影。

3. Unity3D 中的球谐光照

- 球谐光照是一个 CPU 光照算法。它可以处理大规模的光源信息，非常适合一些很小的动态点光源。但是它存在以下几个问题。

- 1、球谐光照是基于物体顶点的算法，并非像素级。这就意味着它不能使用一些像素级的效果增强手段。比如法线贴图。
- 2、基于效率的考虑，球谐光照的更新频率非常低。如果光源移动过快，就会穿邦。
- 3、球谐光照是基于全局空间的计算，当一个面离点光源或者聚光灯很近时，效果是错的。

5.1.2 Deferred Shading Rendering Path(延迟渲染)

- 官方文档: <http://docs.unity3d.com/Manual/RenderTech-DeferredShading.html>

1. 延迟渲染技术简介

- 延迟渲染技术已经是一个成熟且稳定的技术。已经广泛用于各种商业引擎中。它主要解决的是当光源数目过多时带来的复杂光照计算的开销。在传统前向渲染中，假设一个场景中有 M 个物体，N 个光源。那么理论上进行的光照计算次数为 $M \times N$ 。而 DS 管线可以使其变为 $M + N$ 。
- 但是 DS 管线并非万能的。有以下情况需要注意

2. 延迟渲染技术缺陷

- 1、DS 需要多渲染目标 (MRT) 的支持、深度纹理、双面模板缓存。
 - 这个特性，需要显卡支持 SM3.0 的 API。2006 年以后的 PC 显卡应该不成问题。比如 GeForce 8xxxx, AMD Radeon X2400, Intel G45 以后的显卡。
- 2、DS 管线会消耗更多的显存，用于缓存 G-Buffer。同时，会要求显卡拥有更大的位宽 (bit counts)。
 - 在手机上目前（2016 年）还不实现。
- 3、DS 管线无法处理半透明物体。
 - 半透明物体需要退回到传统前向渲染中进行
- 4、硬件抗锯齿 (MSAA) 无法使用。
 - 只能使用一些后期算法，如 FXAA 等。
- 5、这句话中的内容目前未测试，故无法准确体会其中的意思：There is also no support for the Mesh Renderer's Receive Shadows flag and culling masks are only supported in a limited way. You can only use up to four culling masks. That is, your culling layer mask must at least contain all layers minus four arbitrary layers, so 28 of the 32 layers must be set. Otherwise you will get graphical artefacts.

3. 延迟渲染技术在 Unity3D 中的实现方案

- Unity3D 中，延迟渲染管线分为两个阶段进行。G-Buffer 阶段和光照计算 (Lighting) 阶段。

4. G-Buffer 阶段

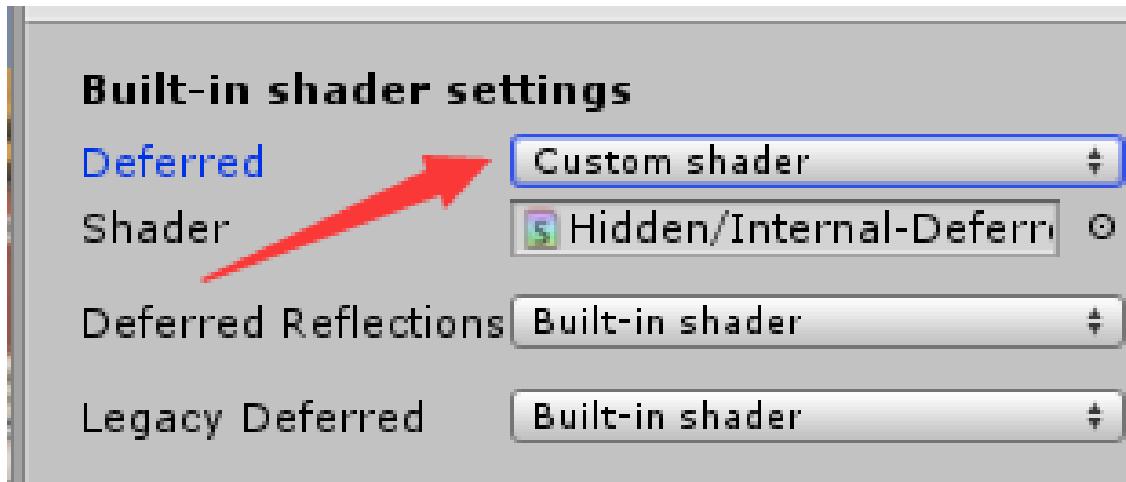
- Unity3D 渲染所有的非透明对象到各个 RT 中，RT 的内容分布见 G-Buffer 内容。Unity3D 将各个 RT 做成了全局变量，方便 Shader 中进行操作。像这样: CameraGBufferTexture0 .. CameraGBufferTexture3

5. 光照计算阶段

- 这个阶段的主要目的就是根据 G-Buffer 的内容进行光照计算。由于是屏幕空间的计算，显然要比之前的前向渲染来得容易得多。
 - 注：实时阴影的计算是在光照计算之前的。DS 管线并不能减少实时阴影的开销。应该怎么整还得怎么整。最后，每个物体受到的实时阴影的影响会叠加到 RT3 中。一些特殊的曝光效果，Lightmap 光影贴图效果等都会进入 RT3。参见 G-Buffer 内容
- G-Buffer 内容
 - • RT0, ARGB32 format: Diffuse color (RGB), occlusion (A).
 - • RT1, ARGB32 format: Specular color (RGB), roughness (A).
 - • RT2, ARGB2101010 format: World space normal (RGB), unused (A).
 - • RT3, ARGB2101010 (non-HDR) or ARGBHalf (HDR) format: Emission + lighting + lightmaps + reflection probes buffer.
 - • Depth+Stencil buffer.
 - 注：为了减少显存和渲染开销。当前场景的摄像机开启了 HDR 模式时，RT3 将不会被创建。而是直接使用摄像机的 HDR RT。

6. 自定义 DS 管线

- Unity3D 中，标准的 DS 管线是 Standard 系列。如果你想修改某一个部分。你可以新建一个自己的 Shader。然后替换掉默认的即可。
- 替换的位置在 Edit->Project Settings->Graphics 面板中。把 Deferred 属性下拉框变成 Custom，就可以进行替换操作了。



5.1.3 Lagacy Deferred Lighting Rendering Path(延迟光照)

- 官方文档: <http://docs.unity3d.com/Manual/RenderTech-DeferredLighting.html>
- 技术文章: <http://www.realtimerendering.com/blog/deferred-lighting-approaches/>
 - 注: 从 5.0 开始, DL 渲染已经不被 Unity3D 推荐, Unity3D 更推荐大家新项目使用 DS 管线方式。因为 DL 方式不好实现基于 PBR 着色的 Standard 材质, 以及场景反射。
 - 注: 如果摄像机设置为了正交投影。会强制退回到前向渲染
 - 注: DL 也不会降低实时阴影的开销
- 和 DS 渲染管线一样, DL 的出现同样是为了解决光照计算的复杂度问题。然后不同的是, DL 仅仅把光照计算拿出去了。简单说来, DL 管线工作流程如下。
 - 步骤一、渲染场景中的对象, 输出光照计算需要的 RT (深度、法线、高光信息)
 - 步骤二、使用上面的 RT 进行光照计算
 - 步骤三、再次渲染场景中的对象, 并与计算机来的光照信息结合。
- 不难看出, DL 相比 DS 而言, 不需要大量的 G-Buffer 支持。甚至不需要 MRT 的支持。但是对深度图的要求是必须的。如果遇上无法访问深度 BUFFER 的情况。那就需要做一次 Pre-Depth Pass 渲染。
- 比起 DS 而言, DL 由于不需要 MRT 的支持。在硬件特性需求和位宽上, 少了许多开销。
 - 1、SM3.0
 - 2、PC:GeForce FX、Radeon X1300、Intel 965 / GMA X3100
 - 3、Mobile: 所有支持 OpenGL ES 3.0 的设备, 部分支持 OpenGL ES 2.0 的设备。

5.1.4 Vertex Lit Rendering Path(顶点光照)

- 官方文档: <http://docs.unity3d.com/Manual/RenderTech-VertexLit.html>
 - 注: 这不是一个通用的技术名词, 只有 Unity3D 中有。
 - 注: 主机平台这个不顶用。
 - 注: 不支持实时阴影和高精度高光计算。
- Vertex Lit Rendering 就是指, 所有的光照计算都通过顶点进行。而在 Unity3D 中, 它的主要目的是为了支持那些没有可编程管线的设备。VL 也提供了几种方法, 用于支持不同的材质类型。

- 1、Vertex 使用 Blinn-Phong 进行光照处理，针对没有 lightmap 的对象使用。
- 2、VertexLMRGBM 针对 lightmap 贴图使用 RGBM 加密（PC 和主机平台）的对象，没有额外的光照计算。
- 3、VertexLMM 针对 lightmap 贴图使用 double-LDR 加密（移动平台）的对象，没有额外的光照计算。

5.1.5 Unity3D 对几种渲染管线的统一处理

- 在上面我描述中，我们发现，不管哪一种管线。都会涉及到几个部分。
 - 1、光照计算
 - 2、透明渲染
 - 3、阴影计算
 - 4、图像输出。
- 那 Unity3D 又是如何对这些进行统一流程控制的呢。我们看下面的图就明白了。

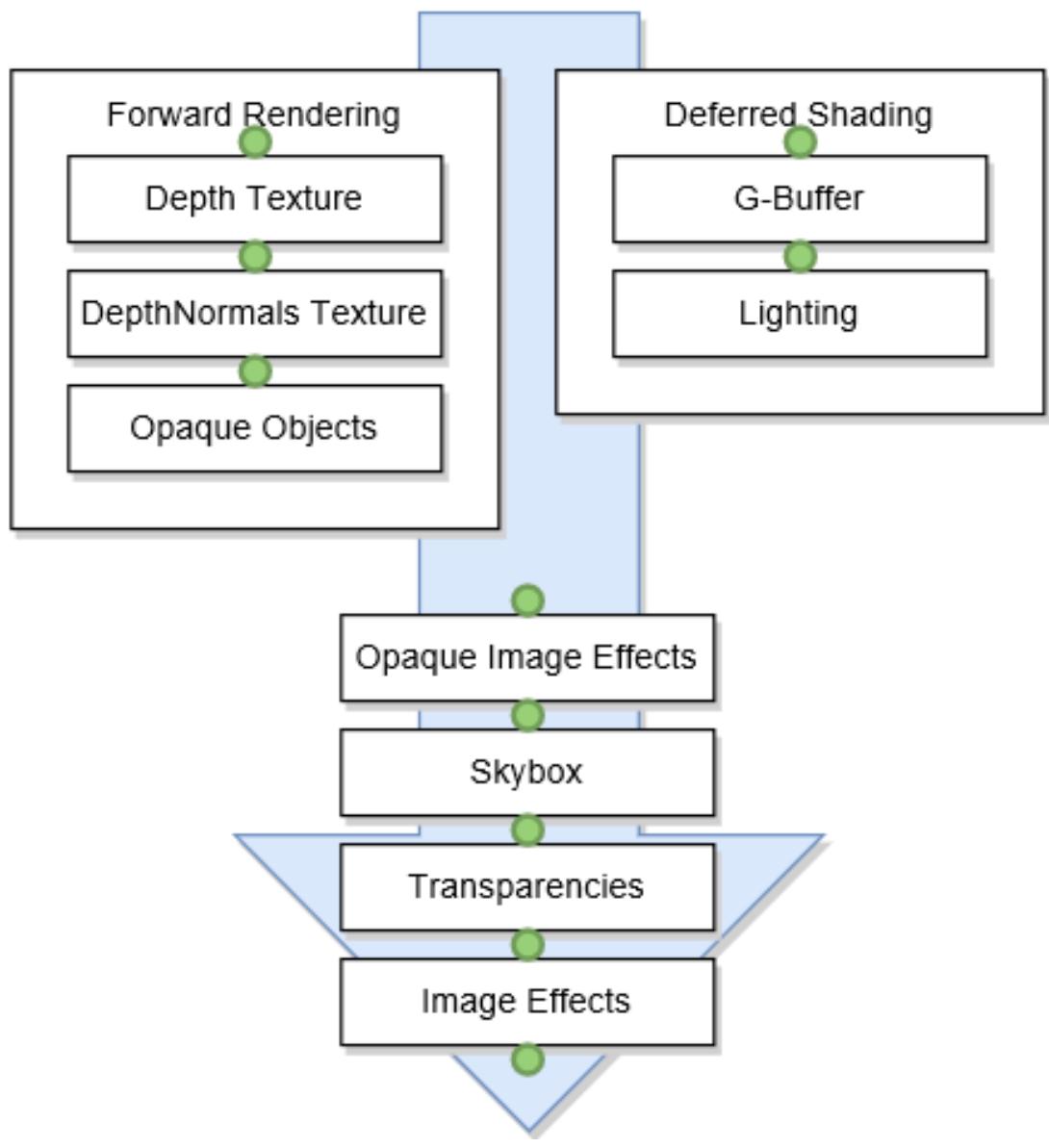


Figure 1: Unity3D 渲染管线图

- 从图中我们可以看到。Unity3D 的传统渲染管线和延迟渲染管线，在非透明物体渲染和光照阶段是分离的。当处理完以后，紧接着处理非透明图像效果-> 天空盒-> 透明物体渲染-> 后期效果。

5.1.6 结束语

- 总的来说，Unity3D 的渲染管线还算稳定和易用，其的渲染管线包含了常见的三种方案。最主要目的还是在想着光照计算和效果显示的处理。同时保持对上层透明。其 Graphics Commander Buffer 提供了扩展管线的能力。而 Graphics 设置面板里，也提供了自定义延迟管线 Shader 的功能。然而也有诸多的美中不足，比如，正交摄像机模式下，不能使用延迟渲染技术。延迟光照下无法实现 PBR 和实时环境反射抽取等等。

5.2 Unity5 Lighting 面板说明书

- <https://blog.csdn.net/u010026134/article/details/53673015>

5.2.1 Object 面板

- Lightmap Static: 当前所选物体是否有勾上 Lightmap Static。如果该物体没有勾上 Lightmap Static，仍可用光线探针（Light Probe）来照亮。
- Scale In Lightmap: 该值影响当前物体对应 lightmap 上的像素数。默认值 1 代表像素数只取决于该物体表面区域（也就是说，各区域像素数相同）。该值越大（大于 1），像素数越多；该值越小（小于 1），像素数越少（lightmap 分辨率）。调整该值，有利于优化 lightmap，使重点细节区得到更精确的照射。举个例子，场景是带平坦黑色墙壁的某独立建筑，此时用小一点儿的 lightmap scale（小于 1）会更合适；若场景是一堆五颜六色的摩托车，用大点儿的值更合适。
- Important GI: 勾上，代表告诉 Unity 当前选择物体的光反射/放射会以显眼的方式影响其它物体，不让 Unity 优化该光源效果。
- Advanced Parameters: 当前选择物体的高级 lightmap 设置（可选择或创建）。
- Preserve UVs: 是否保护 UV 坐标（又可以叫“是否接受 UV 优化”）。为了提高性能，Unity 会重新计算实时光照贴图的 UV 坐标。而在重新计算时，有时会对原 UV 坐标连续性判断出错。举个例子，把锐角错误判断成曲面。若勾上“保护 UV 坐标”，则保留原有 UV 的效果；若不勾，则 Unity 会基于已烘培 UV，计算实时光照贴图 UV，加入临近 Charts，尽可能使 lightmap 紧凑。计算过程和下方 max distance, max angle 有关。实时 Charts 被半像素的边界包围，保证渲染时不会发生遗漏。
- Auto UV Max Distance: 如果 Charts 之间的 worldspace 距离比该值小，则简化 Charts。
- Auto UV Max Angle: 如果 Charts 之间的角度比该值小，则简化 Charts。
- Ignore Normals: 在实时全局光照检测 Charts 时，请勿比较法线（就是说这里要打勾）。手动编辑 UV 坐标时，可勾上该选项，以避免出现 Chart 分离问题。
- Min Chart Size: 这里涉及到 Unity 全新光照系统 Enlighten 的缝合特性，该特性使 Chart 平滑缝合（连接在一起（比如说球体和圆柱体）。该特性需要每个 Chart 的每条边的方向数据，而方向数据是以块（block）单位存储的，则每个可缝合 Chart 最少需要 2*2 块。如果不需精确缝合效果（比如说台阶模型），可选择 2（Minimum）。

Lighting

Object

Scene

Lightmaps

Scene Filter:



Mesh Renderer



Lightmap Static

Scale In Lightmap 1

Important GI

Advanced Parameters

Preserve UVs

Auto UV Max Distance 0.5

Auto UV Max Angle 89

Ignore Normals

Min Chart Size 4 (Stitchable)

▼ Baked Lightmap

Lightmap Index 65535

Tiling X 1

Tiling Y 1

Offset X 0

Offset Y 0

▼ Realtime Lightmap

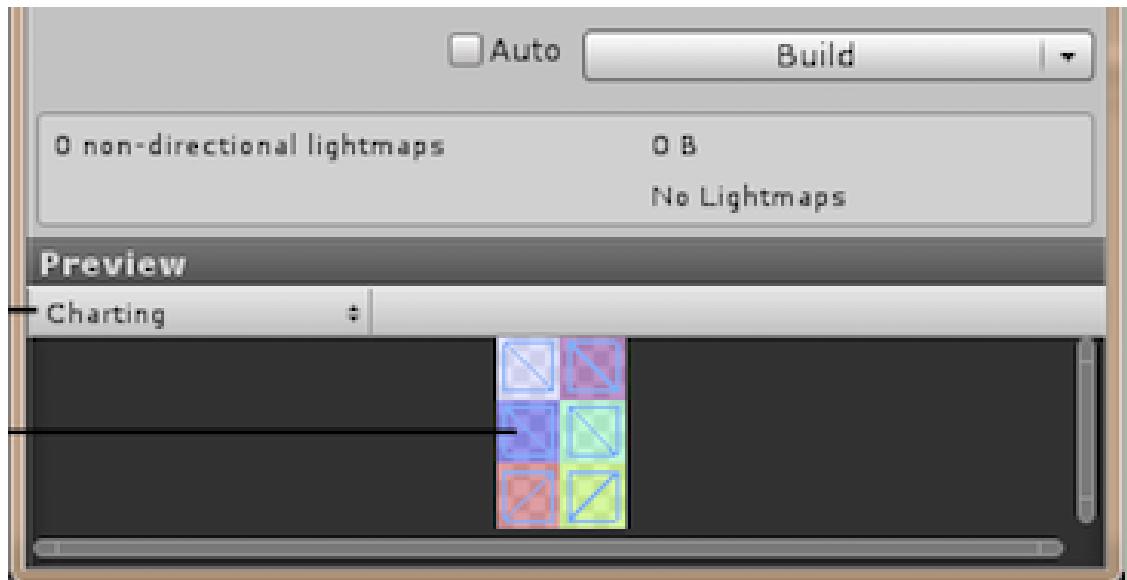
Instance Hash 1ae895faad701493ade03597e633f047

Geometry Hash 8d5465c2e27239985cf06ad5cb0ab729

Instance Resolution 8x12

System Hash d87205c005be798193e0cb06f4ef14fa

System Resolution 8x12



- Charting: 这个彩图就是 Chart，里面的蓝色线框就是当前选择模型的 UV (也可在 Scene 视图的 UV Charts 模式下浏览)。



5.2.2 Scene 面板

- Environment Lighting: 环境光照
 - Sun: 当使用天空盒时，可在此设置代表太阳的平行光（或者其它够大够远的照亮场景的光源）。如果设置为 None，则假定场景中最亮的平行光代表太阳。
 - Ambient Source: 环境光源
 - Ambient Intensity: 环境光亮度
 - Ambient GI: 指定环境光的全局光照模式
 - Reflection Source: 反向源，使用天空盒（天空盒也是 Cubemap）或其它自定义 Cubemap。如果不需反射，请选自定义 Cubemap 但不赋值，这样就不会生成反射探针（Reflection Probe）。
 - Resolution: Cubemap 分辨率
 - Compression: 是否需要压缩反射探针（Reflection Probe）
 - Reflection Intensity: 在反射物上反射源（天空盒或自定义 Cubemap）的可见程度
 - Reflection Bounces: 反弹反射。在场景中用反射探针（Reflection Probe）捕捉该反射。该值决定了反射探针所检测物体之间的来回反射反弹次数。如果该值为 1，则只有初始反射（即来自上述 Reflection Source 的 Skybox/自定义 Cubemap 的反射）。

Lighting

[Object](#)[Scene](#)[Lightmaps](#)

▼ Environment Lighting

Skybox	<input checked="" type="radio"/> Default-Skybox	
Sun	None (Light)	
Ambient Source	Color	
Ambient Color		
Ambient GI	Baked	
Reflection Source	Skybox	
Resolution	128	
Compression	Auto	
Reflection Intensity		1
Reflection Bounces		1

▼ Precomputed Realtime GI

Realtime Resolution	2	texels per unit
CPU Usage	Low (default)	

▼ Baked GI

Baked Resolution	30	texels per unit
Baked Padding	2	texels
Compressed	<input checked="" type="checkbox"/>	
Ambient Occlusion	<input checked="" type="checkbox"/>	
Max Distance	0.25	
Indirect		1
Direct		0
Final Gather		
Atlas Size	1024	

Light Probes

Add Direct Light

▼ General GI

Directional Mode	Directional	
------------------	-------------	--

- Precomputed Realtime GI: 预处理实时全局光照

- Realtime Resolution: 实时 lightmap 分辨率, 每世界坐标单位多少纹素, 通常每单位一纹素就已经有不

错的效果，如果是地形或大型物体，请适量调低该值。

- CPU Usage: 运行时最终光照计算的 CPU 占用

- Baked GI: 烘培式全局光照

- Baked Resolution: 烘培式 lightmap 分辨率，每世界坐标多少纹素
- Baked Padding: 在 lightmap 上图形之间相距有多少纹素
- Compressed: 是否压缩烘培式 lightmap (压缩后的 lightmap 可能会产生伪影)
- Indirect Resolution: (只在 Precomputed Realtime GI 没有勾时才出现) 计算间接照明分辨率。该数值等同 Precomputed Realtime GI 中的 Realtime Resolution。
- Ambient Occlusion: 环境光遮蔽区表面的相对亮度值。该值越大，遮蔽区和无遮蔽区对比越鲜明
- Final Gather: ”最终收集”。当勾上时，在同分辨率已烘培 lightmap 中计算最终光反弹量。能提高 lightmap 可视性，但需要耗费额外的烘培时间。
- Atlas Size: 整张 lightmap 贴图的纹素大小
- Light Probes: 光线探针
- Add Direct Light: 在光线探针中加入平行光。如果整个场景都用烘培式照明，但又需要有动态物体照明，请勾上该选项。如果场景实时照明，则光线探针只能放间接光

- General GI: 一般全局光照

- Directional Mode: 通过设置 lightmap 的 directional 模式，可以存储物体表面每一点的主导入射光信息。在 directional 模式下，会生成第二张 lightmap 来存储主导入射光信息。Directional Specular 模式下，主要是配合漫反射法线贴图，甚至是高光反射法线贴图工作。Directional 模式需要两倍空间存储额外的 lightmap 数据，而 Directional Specular 需要四倍存储空间和两倍纹理内存
- Indirect Intensity: 间接照明显亮度值
- Bounce Boost: 反弹（物体表面之间光反弹）增量
- Default Parameters: 当前场景的默认 Lightmap 设置（可选择或创建）

General GI

Directional Mode

Directional



Directional lightmaps cannot be decoded on SM2.0 hardware nor when using GLES2.0. They will fallback to Non-Directional lightmaps.

Indirect Intensity

1

Bounce Boost

1

Default Parameters

Default-Medium

+

View

Fog

Fog Color



Fog Mode

Exponential Squared

+

Density

0.04



Fog does not affect opaque objects in Deferred Shading. Use Global Fog image effect.

Other Settings

Halo Texture

None (Texture 2D)

○

Halo Strength

0.5

Flare Fade Speed

3

Flare Strength

1

Spot Cookie

Soft

○

 Auto

Build

▼

9 directional lightmaps: 9x1024x1024px

24.0 MB

Preview

- Fog: 雾效

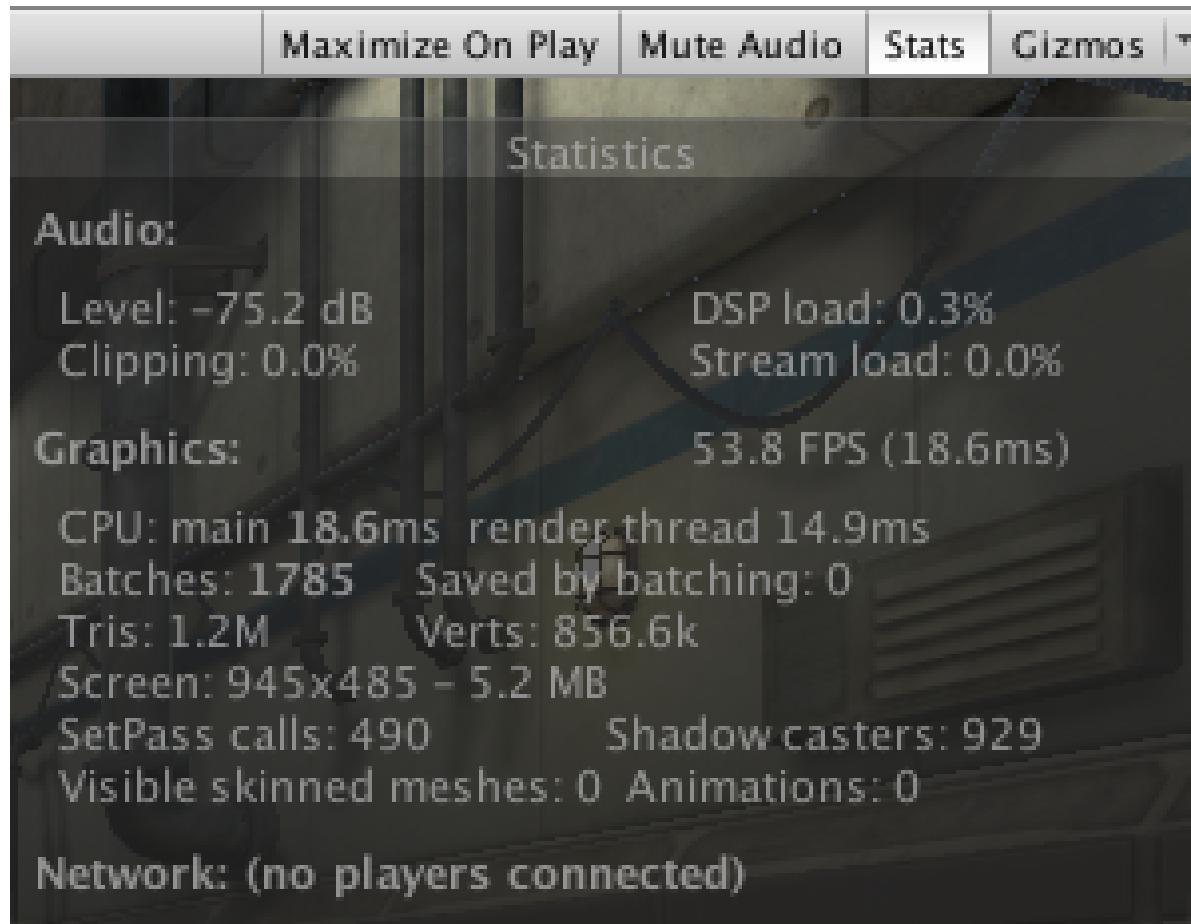
- Fog Color: 雾的颜色 (如果渲染路径选择延迟渲染，则雾效不起作用)
- Fog Mode: 雾效累积模式 (线性累积、指数累积、指数平方累积)
- Density: 雾效密度

- Other Settings: 其它设置

- Halo Texture: 光晕贴图
- Halo Strength: 光晕能见度
- Flare Fade Speed: 镜头光晕淡出时间
- Flare Strength: 镜头光晕能见度
- Spot Cookie: 聚光灯 (Spot Light) 所用剪影 (Cookie)

5.2.3 Stats 面板

- <http://www.voidcn.com/article/p-kegvverb-ha.html>
- Unity5 的 Statistics 上的统计信息和 Unity4 有一些区别,
- Statistics 窗口, 全称叫做 Rendering Statistics Window, 即渲染统计窗口 (或渲染数据统计窗口),
- 窗口中罗列出关于渲染、声音、网络状况等多种统计信息, 下面详细的解释一下这些项的意义。



- FPS
- FPS(Time per frame andFPS):frames per seconds 表示引擎处理和渲染一个游戏帧所花费的时间, 该数字主要受到场景中渲染物体数量和 GPU 性能的影响, FPS 数值越高, 游戏场景的动画显示会更加平滑和流畅。一般来说, 超过 30FPS 的画面人眼不会感觉到卡, 由于视觉残留的特性, 光在视网膜上停止后人眼还会保持 1/24 秒左右的时间, 因此游戏画面每秒帧数至少要保证在 30 以上。
- 另外, Unity 中的 FPS 数值仅包括此游戏 Scene 里更新和渲染的帧, 编辑器中绘制的 Scene 和其它监视窗口的进程不包括在内。
- CPU
 - CPU: 获取到当前占用 CPU 进行计算的时间绝对值, 或时间点, 如果 Unity 主进程处于挂断或休眠状态时, CPU time 将会保持不变。
- Render thread
 - Render thread:GPU 渲染线程处理图像所花费的时间, 具体数值由 GPU 性能来决定
- Batches
 - Batches: 即 Batched Draw Calls, 是 Unity 内置的 Draw Call Batching 技术。

- 首先解释下什么叫做“Draw call”，CPU 每次通知 GPU 发出一个 glDrawElements (OpenGL 中的图元渲染函数) 或者 DrawIndexedPrimitive (DirectX 中的顶点绘制方法) 的过程称为一次 Draw call，一般来说，引擎每对一个物体进行一次 DrawCall，就会产生一个 Batch，这个 Batch 里包含着该物体所有的网格和顶点数据，当渲染另一个相同的物体时，引擎会直接调用 Batch 里的信息，将相关顶点数据直接送到 GPU，从而让渲染过程更加高效，即 Batching 技术是将所有材质相近的物体进行合并渲染。
- 对于含有多个不同 Shader 和 Material 的物体，渲染的过程比较耗时，因为会产生多个 Batches。每次对物体的材质或者贴图进行修改，都会影响 Batches 里数据集的构成。因此，如果场景中有大量材质不同的物体，会很明显的影响到 GPU 的渲染效率。这里说几点关于 Batches 优化相关的方案
- 虽然 Unity 引擎自带 Draw Call Batching 技术，我们也可以通过手动的方式合并材质接近的物体；
- 尽量不要修改 Batches 里物体的 Scale，因为这样会生成新的 Batch。
- 为了提升 GPU 的渲染效率，应当尽可能的在一个物体上使用较少的材质，减少 Batches 过多的开销；
- 对于场景中不会运动的物体，考虑设置 Static 属性,Static 声明的物体会自动进行内部批处理优化。

- Verts 和 Tris

- Verts: 摄像机视野 (field of view) 内渲染的顶点总数。
- Tris: 摄像机视野 (field of view) 内渲染的三角面总数量。
 - 关于 Tris 和 Verts, 突然想到一些问题，这里需要多嘴说几句：
 - Camera 的渲染性能受到 Draw calls 的影响。之前说过，对一个物体进行渲染，会生成相应的 Draw call，处理一个 Draw Call 的时间是由它上面的 Tris 和 Verts 数目决定。尽可能得合并物体，会很大程度的提高性能。举个很简单例子，比如场景一有 1000 个不同的物体，每个物体都有 10 个 Tris；场景二中有 10 个不同的物体，每个物体有 1000 个 Tris。在渲染处理中，场景一中会产生 1000 个 Draw Calls，它的渲染时间明显比场景二慢。
 - Unity stats 视图中的 Tris 和 Verts 并不仅仅是视锥中的梯形内的 Tris 和 Verts，而是 Camera 中 field of view 所有取值下的 tris 和 verts，换句话说，哪怕你在当前 game 视图中看不到这个 cube，如果当你把 field of view 调大到 179 过程中都看不到这个 cube，stats 面板才不会统计，GPU 才不会渲染，否则都会渲染，而且 unity 不会把模型拆分，这个模型哪怕只有 1 个顶点需要渲染，unity 也会把整个模型都渲染出来。（参考自 Mess 的《Unity Camera 组件部分参数详解》）
 - 之前有童鞋问过我，新建一个空的场景，里边没有添加任何物体，为什么 Status 面板上显示有 1.7k Tris 以及 5.0kVerts。这是因为空的场景自带默认的天空盒。
 - 点击 Windows—Lighting 打开 Lighting 下的 Scene 面板，把 Skybox 里的材质设为空
 - 删掉它，你就会发现 Tris 和 Verts 都变为 0 了（以 Unity 5.5.0 为例）

- Screen

- Screen: 获当前 Game 屏幕的分辨率大小，后边的 2.1MB 表示总的内存使用数值。

- SetPass calls

- SetPass calls: 在 Unity 4.x 和 3.x 原来的 Stats 面板的第一项是“Draw calls”，然而到了 Unity5.X 版本，Stats 上没有了“Draw calls”，却多出来一项“SetPass calls”。
- 比如说场景中有 100 个 gameobject，它们拥有完全一样的 Material，那么这 100 个物体很可能被 Unity 里的 Batching 机制结合成一个 Batch。所以用“Batches”来描述 Unity 的渲染性能是不太合适的，它只能反映出场景中需要批处理物体的数量。那么可否用“Draw calls”来描述呢？答案同样是不适合。每一个“Draw calls”是 CPU 发送给 GPU 的一个渲染请求，请求中包括渲染对象所有的顶点参数、三角面、索引值、图元个数等，这个请求并不会占用过多的消耗，真正消耗渲染资源的是在 GPU 得到请求指令后，把指令发送给对应物体的 Shader，让 Shader 读取指令并通知相应的渲染通道 (Pass) 进行渲染操作。
- 假设场景中有 1 个 gameobject，希望能显示很酷炫的效果，它的 Material 上带有许多特定的 Shader。为了实现相应的效果，Shader 里或许会包含很多的 Pass，每当 GPU 即将去运行一个 Pass 之前，就会产生一个“SetPass call”，因此在描述渲染性能开销上，“SetPass calls”更加有说服力。

- Shadow casters

- Shadow casters: 表示场景中有多少个可以投射阴影的物体，一般这些物体都作为场景中的光源。

- visible skinned meshed

– visible skinned meshed: 渲染皮肤网格的数量。

- Animations

- Animations: 正在播放动画的数量。

- 其它

- 目前渲染统计窗口的参数就只有这些，如果你想了解更多的渲染信息，可以打开 Unity 的 Profiler 窗口（右键—AddTab—Profiler），这儿你可以获取到更多的渲染数据，比如“Draw Calls”、“VBO Totals”、“VBO Uploads”等等，还能实时观察 CPU、内存和音频的使用情况。

5.3 Unity3D Mesh 学习笔记 – 创建 MeshBuilder

- <http://www.voidcn.com/article/p-ubvbgmhq-pe.html>

```
1  using UnityEngine;
2  using System.Collections;
3  using System.Collections.Generic;
4
5 // 实现了 MeshBuilder 的类，会使得我们生成 mesh 的时候犯的错误更少一些
6 public class MeshBuilder {
7
8     ^~Iprivate List<Vector3> m_Vertices = new List<Vector3>();
9     ^~Ipublic List<Vector3> Vertices {get{return m_Vertices;}}
10
11    private List<Vector3> m_Normals = new List<Vector3>();
12    ^~Ipublic List<Vector3> Normals {get{return m_Normals;}}
13
14    private List<Vector2> m_UVs = new List<Vector2>();
15    ^~Ipublic List<Vector2> UVs {get{return m_UVs;}}
16
17    private List<int> m_indexs = new List<int>();
18
19    ^~Ipublic void AddTriangle(int index0, int index1, int index2) {
20        m_indexs.Add(index0);
21        m_indexs.Add(index1);
22        m_indexs.Add(index2);
23    }
24
25    public Mesh CreateMesh() {
26        Mesh mesh = new Mesh();
27        mesh.vertices = m_Vertices.ToArray();
28        mesh.triangles = m_indexs.ToArray();
29        if (m_Normals.Count == m_Vertices.Count)
30            mesh.normals = m_Normals.ToArray();
31        if (m_UVs.Count == m_Vertices.Count)
32            mesh.uv = m_UVs.ToArray();
33        mesh.RecalculateBounds();
34        return mesh;
35    }
36}
37
38 public class MeshBuilderTest : MonoBehaviour {
39
40     ^~Iprivate float m_Length = 1f;
41     ^~Iprivate float m_Width = 1f;
42
43     ^~Ipublic void Start() {
44         TestMeshBuilder();
45     }
46}
```

```

46
47  ^~Ipublic void TestMeshBuilder() {
48      MeshBuilder meshBuilder = new MeshBuilder();
49
50      //Set up the vertices and triangles:
51      meshBuilder.Vertices.Add(new Vector3(0.0f, 0.0f, 0.0f));
52      meshBuilder.UVs.Add(new Vector2(0.0f, 0.0f));
53      meshBuilder.Normals.Add(Vector3.up);
54
55      meshBuilder.Vertices.Add(new Vector3(0.0f, 0.0f, m_Length));
56      meshBuilder.UVs.Add(new Vector2(0.0f, 1.0f));
57      meshBuilder.Normals.Add(Vector3.up);
58
59      meshBuilder.Vertices.Add(new Vector3(m_Width, 0.0f, m_Length));
60      meshBuilder.UVs.Add(new Vector2(1.0f, 1.0f));
61      meshBuilder.Normals.Add(Vector3.up);
62
63      meshBuilder.Vertices.Add(new Vector3(m_Width, 0.0f, 0.0f));
64      meshBuilder.UVs.Add(new Vector2(1.0f, 0.0f));
65      meshBuilder.Normals.Add(Vector3.up);
66
67      meshBuilder.AddTriangle(0, 1, 2);
68      meshBuilder.AddTriangle(0, 2, 3);
69
70      MeshFilter filter = this.gameObject.AddComponent<MeshFilter> ();
71      if (filter != null) {
72          filter.sharedMesh = meshBuilder.CreateMesh();
73      }
74
75      MeshRenderer meshRender = this.gameObject.AddComponent<MeshRenderer> ();
76      Shader shader = Shader.Find ("Diffuse");
77      meshRender.sharedMaterial = new Material (shader);
78  }
79 }

```

5.4 深入了解光照贴图

- <https://my.oschina.net/u/2874878/blog/736785>

6 Unity ShaderLab 学习总结

6.1 Shader 基础

6.1.1 SubShader 的 Tag

```

Shader "ShaderLab Tutorials/TestShader" {
    SubShader {
        {
            Tags { "Queue"="Geometry+10" "RenderType"="Opaque" }
            //...
        }
    }
}

```

SubShader 内部可以有标签（Tags）的定义。Tag 指定了这个 SubShader 的渲染顺序（时机），以及其他的一些设置。

1. “RenderType” 标签。Unity 可以运行时替换符合特定 RenderType 的所有 Shader。Camera.RenderWithShader 或 Camera.SetReplacementShader 配合使用。Unity 内置的 RenderType 包括：

- “Opaque”：绝大部分不透明的物体都使用这个；

- ”Transparent”：绝大部分透明的物体、包括粒子特效都使用这个；
- ”Background”：天空盒都使用这个；
- ”Overlay”：GUI、镜头光晕都使用这个；

用户也可以定义任意自己的 RenderType 这个标签所取的值。

应注意，Camera.RenderWithShader 或 Camera.SetReplacementShader 不要求标签只能是 RenderType，RenderType 只是 Unity 内部用于 Replace 的一个标签而已，你也可以自定义自己全新的标签用于 Replace。

比如，你为自己的 ShaderA.SubShaderA1（会被 Unity 选取到的 SubShader，常为 Shader 文件中的第一个 SubShader）增加 Tag 为”Distort”=”On”，然后将”Distort”作为参数 replacementTag 传给函数。此时，作为 replacementShader 实参的 ShaderB.SubShaderB1 中若有也有一模一样的”Distort”=”On”，则此 SubShaderB1 将代替 SubShaderA1 用于本次渲染。

具体可参考 Rendering with Replaced Shaders <https://docs.unity3d.com/Manual/SL-ShaderReplacement.html>

2. ”Queue” 标签。定义渲染顺序。预制的值为

- ”Background”。值为 1000。比如用于天空盒。
- ”Geometry”。值为 2000。大部分物体在这个队列。不透明的物体也在里面。这个队列内部的物体的渲染顺序会有进一步的优化（应该是从近到远，early-z test 可以剔除不需经过 FS 处理的片元）。其他队列的物体都是按空间位置的从远到近进行渲染。
- ”AlphaTest”。值为 2450。已进行 AlphaTest 的物体在这个队列。
- ”Transparent”。值为 3000。透明物体。
- ”Overlay”。值为 4000。比如镜头光晕。
- 用户可以定义任意值，比如”Queue”=”Geometry+10”
- ”ForceNoShadowCasting”，值为”true” 时，表示不接受阴影。
- ”IgnoreProjector”，值为”true” 时，表示不接受 Projector 组件的投影。

另，关于渲染队列和 Batch 的非官方经验总结是，一帧的渲染队列的生成，依次决定于每个渲染物体的：

- Shader 的 RenderType tag,
- Renderer.SortingLayerID,
- Renderer.SortingOrder,
- Material.renderQueue (默认值为 Shader 里的”Queue”) ,
- Transform.z(ViewSpace) (默认为按 z 值从前到后，但当 Queue 是”Transparent”的时候，按 z 值从后到前)。

这个渲染队列决定了之后（可能有 dirty flag 的机制?）渲染器再依次遍历这个渲染队列，“同一种”材质的渲染物体合到一个 Batch 里。

6.1.2 Pass

```
Shader "ShaderLab Tutorials/TestShader" {
    SubShader {
        Pass {
            {
                //...
            }
        }
    }
}
```

一个 SubShader (渲染方案) 是由一个个 Pass 块来执行的。每个 Pass 都会消耗对应的一个 DrawCall。在满足渲染效果的情况下尽可能地减少 Pass 的数量。

1. Pass 的 Tag

```

1 Shader "ShaderLab Tutorials/TestShader" {
2     SubShader {
3         Pass {
4             Tags{ "LightMode"="ForwardBase" }
5             //...
6         }
7     }
8 }
9 }
```

和 SubShader 有自己专属的 Tag 类似，Pass 也有 Pass 专属的 Tag。

其中最重要 Tag 是“LightMode”，指定 Pass 和 Unity 的哪一种渲染路径（“Rendering Path”）搭配使用。除最重要的 ForwardBase、ForwardAdd 外，这里需额外提醒的 Tag 取值可包括：

- Always，永远都渲染，但不处理光照
- ShadowCaster，用于渲染产生阴影的物体
- ShadowCollector，用于收集物体阴影到屏幕坐标 Buff 里。
- 其他渲染路径相关的 Tag 详见下面章节“Unity 渲染路径种类”。
- 具体所有 Tag 取值，可参考 ShaderLab syntax: Pass Tags。

2. FallBack

```

1 Shader "ShaderLab Tutorials/TestShader"{
2     SubShader { Pass {} }
3
4     FallBack "Diffuse" // "Diffuse"即Unity预制的固有Shader
5     // FallBack Off //将关闭FallBack
6 }
```

当本 Shader 的所有 SubShader 都不支持当前显卡，就会使用 FallBack 语句指定的另一个 Shader。FallBack 最好指定 Unity 自己预制的 Shader 实现，因其一般能够在当前所有显卡运行。

3. Properties

```

1 Shader "ShaderLab Tutorials/TestShader"
2 {
3     Properties {
4         _Range ("My Range", Range (0.02,0.15)) = 0.07 // sliders
5         _Color ("My Color", Color) = (.34, .85, .92, 1) // color
6         _2D ("My Texture 2D", 2D) = "" {} // textures
7         _Rect("My Rectangle", Rect) = "name" {}
8         _Cube ("My Cubemap", Cube) = "name" {}
9         _Float ("My Float", Float) = 1
10        _Vector ("My Vector", Vector) = (1,2,3,4)
11
12        // Display as a toggle.
13        [Toggle] _Invert ("Invert color?", Float) = 0
14        // Blend mode values
15        [Enum(UnityEngine.Rendering.BlendMode)] _Blend ("Blend mode", Float) = 1
16        //setup corresponding shader keywords.
17        [KeywordEnum(Off, On)] _UseSpecular ("Use Specular", Float) = 0
18    }
19
20    // Shader
21    SubShader{
22        Pass{
23            //...
24            uniform float4 _Color;
25            //...
```

```

26     float4 frag() : COLOR{ return fixed4(_Color); }
27     //...
28     #pragma multi_compile __ _USESPECULAR_ON
29 }
30 }
31
32 //fixed pipeline
33 SubShader {
34     Pass{
35         Color[_Color]
36     }
37 }
38 }
```

- Shader 在 Unity 编辑器暴露给美术的参数，通过 Properties 来实现。
- 所有可能的参数如上所示。主要也就 Float、Vector 和 Texture 这 3 类。
- 除了通过编辑器编辑 Properties，脚本也可以通过 Material 的接口（比如 SetFloat、SetTexture 编辑）
- 之后在 Shader 程序通过 [name]（固定管线）或直接 name（可编程 Shader）访问这些属性。
- 在每一个 Property 前面也能类似 C# 那样添加 Attribute，以达到额外 UI 面板功能。详见 Material-PropertyDrawer.html。

6.1.3 Shader 中的数据类型

有 3 种基本数值类型：float、half 和 fixed。

这 3 种基本数值类型可以再组成 vector 和 matrix，比如 half3 是由 3 个 half 组成、float4x4 是由 16 个 float 组成。

- float：32 位高精度浮点数。
- half：16 位中精度浮点数。范围是 [-6 万, +6 万]，能精确到十进制的小数点后 3.3 位。
- fixed：11 位低精度浮点数。范围是 [-2, 2]，精度是 1/256。

数据类型影响性能

- 精度够用就好。
- 颜色和单位向量，使用 fixed
- 其他情况，尽量使用 half（即范围在 [-6 万, +6 万] 内、精确到小数点后 3.3 位）；否则才使用 float。

6.1.4 ShaderLab 中的 Matrix

当提到“Row-Major”、“Column-Major”，根据不同的场合，它们可能指不同的意思：

- 数学上的，主要是指矢量 V 是 Row Vector、还是 Column Vector。引用自 [Game Engine Architecture 2nd Edition, 183]。留意到 V 和 M 的乘法，当是 Row Vector 的时候，数学上写作 VM，Matrix 在右边，Matrix 的最下面一行表示 Translate；当是 Column Vector 的时候，数学上写作 MtVt，Matrix 在左边并且需要转置，Matrix 最右面一列表示 Translate。
- 访问接口上的：Row-Major 即 MyMatrix[Row][Column]、Column-Major 即 MyMatrix[Column][Row]。HLSL/C 的访问接口都是 Row-Major，比如 MyMatrix¹ 返回的是第 3 行；GLSL 的访问接口是 Column-Major，比如 MyMatrix¹ 返回的是第 3 列。
- 寄存器存储上的：每个元素是按行存储在寄存器中、还是按列存储在寄存器中。需要关注它的一般情况举例是，float2x3 的 MyMatrix，到底是占用 2 个寄存器（Row-Major）、还是 3 个寄存器（Column-Major）。在 HLSL 里，可以通过 #pragmapack_{matrixrowmajorcolumnmajor}

上述情况，互不相干。

然后，ShaderLab 中，数学上是 Column Vector、访问接口上是 Row-Major、存储上是（尚未查明）。

¹DEFINITION NOT FOUND.

6.1.5 ShaderLab 中各个 Space 的坐标系

一般情况下，从 Vertex Buff 输入顶点到 Vertex Shader，

- 该顶点为左手坐标系 Model Space 中的顶点 vInModel，
- 其用 w=1 的 Homogenous Coordinates (故等效于 Cartesian Coordinates) 表达 $vInModel = \text{float4}(xm, ym, zm, 1)$ ；
- $vInWorld = \text{mul}(\text{Object2World}, vInModel)$ 后，得出左手坐标系 World Space 中的 vInWorld，其为 w=1 的 Homogenous Coordinates (故等效于 Cartesian Coordinates) $vInWorld = \text{float4}(xw, yw, zw, 1)$ ；
- $vInView = \text{mul}(\text{UNITY_MATRIX_V}, vInWorld)$ 后，得出右手坐标系 View Space 中的 vInView，其为 w=1 的 Homogenous Coordinates (故等效于 Cartesian Coordinates) $vInWorld = \text{float4}(xv, yv, zv, 1)$ ；
- $vInClip = \text{mul}(\text{UNITY_MATRIX_P}, vInView)$ 后，得出左手坐标系 Clip Space 中的 vInClip，其为 w 往往不等于 1 的 Homogenous Coordinates (故往往不等效于 Cartesian Coordinates) $vInClip = \text{float4}(xc, yc, zc, wc)$ ；

设 r、l、t、b、n、f 的长度绝对值如下图：

注意 View Space 中摄像机前方的 z 值为负数、-z 为正数。则 GL/DX/Metal 的 Clip Space 坐标为：

- GL:
 - $xc = (2nx + rz + lz) / (r - l)$;
 - $yc = (2ny + tz + bz) / (t - b)$;
 - $zc = (-fz - nz - 2nf) / (f - n)$;
 - $wc = -z$;
- DX/Metal:
 - $xc = (2nx + rz + lz) / (r - l)$;
 - $yc = (2ny + tz + bz) / (t - b)$;
 - $zc = (-fz - nf) / (f - n)$;
 - $wc = -z$;
- $vInNDC = vInClip / vInClip.w$ 后，得出左手坐标系 Normalized Device Coordinates 中的 vInNDC，其为 w=1 的 Homogenous Coordinates (故等效于 Cartesian Coordinates) $vInNDC = \text{float4}(xn, yn, zn, 1)$ 。xn 和 yn 的取值范围为 [-1,1]。
 - GL: $zn = zc / wc = (fz + nz + 2nf) / ((f - n)z)$;
 - DX/Metal: $zn = zc / wc = (fz + nf) / ((f - n)z)$;
 - 在 Unity 中，zn 的取值范围可以这样决定：
 - 如果 $\text{UNITY_REVERSEDZ_zn}[\text{UNITY_NEARCLIPVALUE}, 0]$ ，即 [1,0]
 - 如果 $\text{UNITY_REVERSEDZ_zn}[\text{UNITY_NEARCLIPVALUE}, 1]$
 - 如果 $\text{SHADER_API_D3D9}/\text{SHADER_API_D3D119X} [0,1]$
 - 否则，即 OpenGL 情况，即 [-1,1]

* 如果 $\text{UNITY_REVERSEDZ_zn}[\text{UNITY_NEARCLIPVALUE}, 0]$ ，即 [1,0]

* 如果 $\text{UNITY_REVERSEDZ_zn}[\text{UNITY_NEARCLIPVALUE}, 1]$

· 如果 $\text{SHADER_API_D3D9}/\text{SHADER_API_D3D119X} [0,1]$

· 否则，即 OpenGL 情况，即 [-1,1]

```
1 v2f vert (appdata v)
2 {
3     v2f o;
4     o.vertex = mul(UNITY_MATRIX_MVP, v.vertex);
5     // 1 、 2 、 3 是等价的，和 4 是不等价的
6     // 因为是 M 在左、 V 在右，所以是 Column Vector
7     // 因为是 HLSL/CG 语言，所以是访问方式是 Row-Major
8     o.rootInView = mul(UNITY_MATRIX_MV, float4(0, 0, 0, 1)); // 1
9     o.rootInView = float4(UNITY_MATRIX_MV[0].w, UNITY_MATRIX_MV[1].w, UNITY_MATRIX_MV[2].w, UNITY_MATRIX_MV[3].w); // 2
10    o.rootInView = UNITY_MATRIX_MV._m03_m13_m23_m33; // 3
11    // o.rootInView = UNITY_MATRIX_MV[3]; // 4
```

```

12     return o;
13 }
14 }
15
16 fixed4 frag (v2f i) : SV_Target
17 {
18     // 因为是ViewSpace是右手坐标系，所以当root在view前面的时候，z是负数，所以需要-z才能正确
19     fixed4 col = fixed4(i.rootInView.x, i.rootInView.y, -i.rootInView.z, 1);
20     return col;
21 }
22
23 struct appdata
24 {
25     float4 vertex : POSITION;
26 };
27 struct v2f
28 {
29     float4 rootInView : TEXCOORD0;
30     float4 vertex : SV_POSITION;
31 };

```

6.2 Shader 形态

6.2.1 Shader 形态之 1：固定管线

固定管线是为了兼容老式显卡。都是顶点光照。之后固定管线可能是被 Unity 抛弃的功能，所以最好不学它、当它不存在。特征是里面出现了形如下面 Material 块、没有 CGPROGRAM 和 ENDCG 块。

```

Shader "ShaderLab Tutorials/TestShader"
{
    Properties {
        _Color ("My Color", Color) = (.34, .85, .92, 1) // color
    }

    // Fixed Pipeline
    SubShader
    {
        Pass
        {
            Material{
                Diffuse [_Color]
                Ambient [_Color]
            }

            Lighting On
        }
    }
}

```

6.2.2 Shader 形态之 2：可编程 Shader

```

Shader "ShaderLab Tutorials/TestShader"
{
    Properties {}

    SubShader
    {
        Pass
        {
            // ... the usual pass state setup ...
        }
    }
}

```

```

CGPROGRAM
// compilation directives for this snippet, e.g.:
#pragma vertex vert
#pragma fragment frag

// the Cg/HLSL code itself
float4 vert(float4 v:POSITION) : SV_POSITION{
    return mul(UNITY_MATRIX_MVP, v);
}
float4 frag() : COLOR{
    return fixed4(1.0, 0.0, 0.0, 1.0);
}
ENDCG
// ... the rest of pass setup ...
}

}

}

```

- 功能最强大、最自由的形态。
- 特征是在 Pass 里出现 CGPROGRAM 和 ENDCG 块
- 编译指令 #pragma。详见官网 Cg snippets。其中重要的包括：

编译指令	示例/含义
#pragma vertex name	
#pragma fragment name	替换 name，来指定 Vertex Shader 函数、Fragment Shader 函数。
#pragma target name	替换 name（为 2.0、3.0 等）。设置编译目标 shader model 的版本。
#pragma only_renderers name name ...	
#pragma exclude_renderers name name...	#pragma only_renderers gles gles3, #pragma exclude_renderers d3d9 d3d11 opengl, 只为指定渲染平台（render platform）编译

- 引用库。通过形如 #include "UnityCG.cginc" 引入指定的库。常用的就是 UnityCG.cginc 了。其他库详见官网 Built-in shader include files。
- ShaderLab 内置值。Unity 给 Shader 程序提供了便捷的、常用的值，比如下面例子中的 UNITY_MATRIX_MVP 等 built-in values。
- Shader 输入输出参数语义 (Semantics)。在管线流程中每个阶段之间（比如 Vertex Shader 阶段和 FragmentShader 阶段之间）的输入输出参数，通过语义字符串，来指定参数的含义。常用的语义包括：COLOR, SV_Position, TEXCOORD[n] 等 HLSL Semantic (由于是 HLSL 的连接，所以可能不完全在 Unity 里可以使用)。
- 特别地，因为 Vertex Shader 的输入往往是管线的最开始，Unity 为此内置了常用的数据结构：

数据结构	含义
appdata_base	vertex shader input with position, normal, one texture coordinate.
appdata_tan	vertex shader input with position, normal, tangent, one texture coordinate.
appdata_full	vertex shader input with position, normal, tangent, vertex color and two texture coordinates.
appdata_img	vertex shader input with position and one texture coordinate.

6.2.3 Shader 形态之 3: SurfaceShader

```

Shader "ShaderLab Tutorials/TestShader"
{
    Properties { }

    // Surface Shader
    SubShader {
        Tags { "RenderType" = "Opaque" }

```

```

CGPROGRAM
#pragma surface surf Lambert
struct Input {
    float4 color : COLOR;
};
void surf (Input IN, inout SurfaceOutput o) {
    o.Albedo = 1;
}
ENDCG
}
FallBack "Diffuse"
}

```

- SurfaceShader 可以认为是一个光照 Shader 的语法糖、一个光照 VS/FS 的生成器。减少了开发者写重复代码的需要。
- 在手游，由于对性能要求比较高，所以不建议使用 SurfaceShader。因为 SurfaceShader 是一个比较“通用”的功能，而通用往往导致性能不高。
- 特征是在 SubShader 里出现 CGPROGRAM 和 ENDCG 块。（而不是出现在 Pass 里。因为 SurfaceShader 自己会编译成多个 Pass。）
- 编译指令是：

```
#pragma surface surfaceFunction lightModel [optionalparams]
```

- surfaceFunction：surfaceShader 函数，形如 void surf (Input IN, inout SurfaceOutput o)
- lightModel：使用的光照模式。包括 Lambert（漫反射）和 BlinnPhong（镜面反射）。
- 也可以自己定义光照函数。比如编译指令为 #pragma surface surf MyCalc
- 在 Shader 里定义 half4 LightingMyCalc (SurfaceOutput s, 参数略) 函数进行处理（函数名在签名加上了“Lighting”）。
- 详见 Custom Lighting models in Surface Shaders
- 你定义输入数据结构（比如上面的 Input）、编写自己的 Surface 函数处理输入、最终输出修改过后的 SurfaceOutput。SurfaceOutput 的定义为

```

1 struct SurfaceOutput {
2     half3 Albedo; // 纹理颜色值 (r, g, b)
3     half3 Normal; // 法向量(x, y, z)
4     half3 Emission; // 自发光颜色值(r, g, b)
5     half Specular; // 镜面反射度
6     half Gloss; // 光泽度
7     half Alpha; // 不透明度
8 };

```

6.2.4 Shader 形态之 4: Compiled Shader

点击 a.shader 文件的“Compile and show code”，可以看到该文件的“编译”过后的 ShaderLab shader 文件，文件名形如 Compiled-a.shader。

其依然是 ShaderLab 文件，其包含最终提交给 GPU 的 shader 代码字符串。

先就其结构进行简述如下，会发现和上述的编译前 ShaderLab 结构很相似。

```

// Compiled shader for iPhone, iPod Touch and iPad, uncompressed size: 36.5KB
// Skipping shader variants that would not be included into build of current scene.
Shader "ShaderLab Tutorials/TestShader"
{
    Properties {...}
    SubShader {

```

```

// Stats for Vertex shader:
//      gles : 14 avg math (11..19), 1 avg texture (1..2)
//      metal : 14 avg math (11..17)
// Stats for Fragment shader:
//      metal : 14 avg math (11..19), 1 avg texture (1..2)
Pass {
    Program "vp" // vertex program
    {
        SubProgram "gles" {
            // Stats: 11 math, 1 textures
            Keywords{...} // keywords for shader variants ("uber shader")

            //shader codes in string
            "
            #ifdef VERTEX
            vertex shader codes
            #endif
            "

            // Note, on gles, fragment shader stays here inside Program "vp"
            #ifdef FRAGMENT
            fragment shader codes
            #endif
            "
        }

        SubProgram "metal" {
            some setup
            Keywords{...}

            //vertex shader codes in string
            "..."

        }
    }

    Program "fp" // fragment program
    {
        SubProgram "gles" {
            Keywords{...}
            "// shader disassembly not supported on gles" //(because gles fragment shader
        }

        SubProgram "metal" {
            common setup
            Keywords{...}

            //fragment shader codes in string
            "..."

        }
    }

    ...
}

```

6.3 Unity 渲染路径 (Rendering Path) 种类

6.3.1 概述

开发者可以在 Unity 工程的 PlayerSettings 设置对渲染路径进行 3 选 1:

- Deferred Lighting, 延迟光照路径。3 者中最高质量地还原光照阴影。光照性能只与最终像素数目有关, 光源数量再多都不会影响性能。
- Forward Rendering, 顺序渲染路径。能发挥出 Shader 全部特性的渲染路径, 当然也就支持像素级光照。最常用、功能最自由, 性能与光源数目 * 受光照物体数目有关, 具体性能视乎其具体使用到的 Shader 的复杂度。
- Vertex Lit, 顶点光照路径。顶点级光照。性能最高、兼容性最强、支持特性最少、品质最差。

6.3.2 渲染路径的内部阶段和 Pass 的 LightMode 标签

每个渲染路径的内部会再分为几个阶段。

然后, Shader 里的每个 Pass, 都可以指定为不同的 LightMode。而 LightMode 实际就是说:“我希望这个 Pass 在这个 XXX 渲染路径的这个 YYY 子阶段被执行”。

1. Deferred Lighting

渲染路径内部子阶段	对应的 LightMode	描述
Base Pass	"PrepassBase"	渲染物体信息。即把法向量、高光度到一张 ARGB32 的纹理上, 把深度信息保存在 Z-Buff 上。
Lighting Pass	无对应可编程 Pass	根据 Base Pass 得出的物体信息, 在屏幕坐标系下, 使用 BlinnPhong 光照模式, 把光照信息渲染到 ARGB32 的光照信息纹理上 (RGB 表示 diffuse 颜色值、A 表示高光度)
Final Pass	"PrepassFinal"	根据光照信息纹理, 物体再渲染一次, 将光照信息、纹理信息、发光信息最终混合。LightMap 也在这个 Pass 进行。

2. Forward Rendering

渲染路径内部子阶段	对应的 LightMode	描述
Base Pass	"ForwardBase"	渲染: 最亮一个的方向光光源 (像素级) 和对应的阴影、所有顶点级光源、LightMap、所有 LightProbe 的 SH 光源 (Sphere Harmonic, 球谐函数, 效率超高的低频光)、环境光、自发光。
Additional Passes	"ForwardAdd"	其他需要像素级渲染的光源

注意到的是, 在 Forward Rendering 中, 光源可能是像素级光源、顶点级光源或 SH 光源。其判断标准是:

- 配制成“Not Important”的光源都是顶点级光源和 SH 光源
- 最亮的方向光永远都是像素级光源
- 配置成“Important”的都是像素级光源
- 上面 2 种情况加起来的像素级光源数目小于“Quality Settings”里面的“Pixel Light Count”的话, 会把第 1 种情况的光源补为额外的像素级光源。

另外, 配置成“Auto”的光源有更复杂的判断标注, 截图如下:

具体可参考 Forward Rendering Path Details。

3. Vertex Lit

渲染路径内部子阶段对应的 LightMode 描述 Vertex “Vertex” 渲染无 LightMap 物体 VertexLMRGBM “VertexLMRGBM” 渲染有 RGBM 编码的 LightMap 物体 VertexLM “VertexLM” 渲染有双 LDR 编码的 LightMap 物体

4. 不同 LightMode 的 Pass 的被选择

一个工程的渲染路径是唯一的，但一个工程里的 Shader 是允许配有不同 LightMode 的 Pass 的。在 Unity，策略是“从工程配置的渲染路径模式开始，按 Deferred、Forward、VertexLit 的顺序，搜索最匹配的 LightMode 的一个 Pass”。比如，在配置成 Deferred 路径时，优先选有 Deferred 相关 LightMode 的 Pass；找不到才会选 Forward 相关的 Pass；还找不到，才会选 VertexLit 相关的 Pass。再比如，在配置成 Forward 路径时，优先选 Forward 相关的 Pass；找不到才会选 VertexLit 相关的 Pass。

6.4 移动设备 GPU 架构简述

《The Mali GPU: An Abstract Machine》系列以 Arm Mali GPU 为例子给出了全面的讨论，现简述如下：

6.4.1 Part 1 - Frame Pipelining

- Application/Geometry/Fragment 三阶段组成，三者中最大才是瓶颈
- OpenGL 的同步 API 是个“illusion”，事实上是 CommandQueue（直到遇到 Fence 会被强制同步），以减少 CPU/GPU 之间的互相等待
- Pipeline Throttle，为了更低的延迟，当 GPU 累积了多帧（往往是 3 帧，以 eglSwapBuffers() 或 Present() 来区分帧）的 Command 时，OS 会通过 eglSwapBuffers() 或 Present() 来阻塞 CPU 让其进入 idle，从而防止更多后续 Command 的提交

6.4.2 Part 2 - Tile-based Rendering

- tile-based deferred rendering (Wiki, PowerVR/Mali/Adreno) 是重要的概念。其将 Fragment 一帧处理多个比如 16x16 的单元，并为 Shader 集成一个小但快的 cache，从而大幅避免 Shader 和主内存之间带宽消耗（电量消耗）

6.4.3 Part 3 - The Midgard Shader Core

- GPU 包含数个（当前常见为 4-8 个）Unified Shading Core，可动态分配用于 Vertex Shader、Fragment Shader 或 Compute Kernel
- 每个 Unified Shader Core 包含数个（当前常见为 2 个）用于 SIMD 计算的运算器 Arithmetic Pipeline (A-pipe)，1 个用于纹理采样的 Texutre Pipeline (T-pipe)，1 个用于非纹理类的内存读写的 Load/Store Pipeline (LS-pipe) 比如顶点属性写读、变量访问等
- 会进行 Early-ZS 测试尝试减少 Overdraw（依赖于渲染物体提交顺序由前至后）
- Arm 的 Forward Pixel Kill 和 PowerVR 的 Hidden Surface Removal 做到像素级别的 Overdraw 减少（不用依赖于渲染物体提交顺序由前至后）
- 当 Shader 使用 discard 或 clip、在 Fragment Shader 里修改深度值、半透明，将不能进行 Early-ZS，只好使用传统的 Late-ZS

6.4.4 Part 4 - The Bifrost Shader Core

- 2016 年的新型号，对架构作出了优化

6.5 References

6.5.1 Implementing a Loading Bar in Unity

- <http://www.alanzucconi.com/2016/03/30/loading-bar-in-unity/>
-
-
-

6.5.2 shader

- http://blog.csdn.net/poem_qianmo/article/details/40723789 【浅墨 Unity3D Shader 编程】之一夏威夷篇：游戏场景的创建 & 第一个 Shader 的书写
- <http://www.jianshu.com/p/7b9498e58659> Unity ShaderLab 学习总结
- 猫都能学会的 Unity3D Shader 入门指南（一）<https://onevcat.com/2013/07/shader-tutorial-1/>
- Youtube: <https://www.youtube.com/watch?v=hDJQXzajiPg> (包括 part1-6)。视频是最佳的入门方式没有之一，所以墙裂建议就算不看下文的所有内容，都要去看一下 part1。
- 书籍：《Unity 3D ShaderLab 开发实战详解》
- 表面着色器的写法 9 种：<http://www.unity.5helpyou.com/2381.html>
- 在 unity 向量空间内绘制几何 (2): 球面—重构《黑客帝国》的‘上帝机器’，Deus Ex Machina http://blog.csdn.net/liu_if_else/article/details/51554940

6.5.3 Unity 动画

- http://www.360doc.com/content/13/0225/17/10941785_267831975.shtml
- 用好 Lua+Unity，让性能飞起来—LuaJIT 性能坑详解 <https://zhuanlan.zhihu.com/p/26528101>

7 Unity3D - 物理引擎

7.1 Unity3D - 物理引擎之刚体部件 (Rigidbody) 详解

- 在虚拟世界中，任何物体都是没有活力的，要想变的真实，Rigidbody 是必不可少的组件，下面介绍 Rigidbody 的各个属性：
 - Mass: 质量
 - Drag: 阻力，对象在运动时遇到的空气阻力，0 表示没有空气阻力，好比丢一个东西出去，如果这个东西没有阻力的话，则会一直不停的向你丢的方向所运动。
 - Angular Drag: 角阻力也称扭矩力，扭矩力是使物体发生转动的一种特殊的力矩。
 - Use Gravity: 使用重力，开启后会受到重力影响。
 - Is Kinematic: 是否开启动力学，开启后不在受物理引擎的影响，只能通过 Transform 属性来操作。此属性一般用来模拟平台的移动，或带有铰链关节链接刚体的动画。
 - Interpolate: 插值，用于控制刚体运动抖动的情况，有以下三种值可选。
 - None: 没有插值
 - Interpolate: 内插值，基于前一帧的 Transform 来平滑此次的 Transfomr。
 - Extrapolate: 外插值，基于下一帧的 Transform 来平滑此次的 Transform。
 - Collision Detection: 碰撞检测，用于控制避免高速运动的对象穿过其他对象而未发生碰撞。
 - Discrete: 离散碰撞检测（默认值），与场景中其他多有碰撞体进行碰撞检测。
 - Continuous: 连续碰撞检测，此模式对物理性能会有很大影响，如果不需要对快速运动的对象进行碰撞检测，就是用离散模式。
 - Continuous Dynamic: 连续动态碰撞检测模式。
 - Constraints: 约束，控制对刚体运动的约束。
 - Freeze Position: 冻结位置，刚体对象在世界坐标系中的 X,Y,Z, 轴方向上的移动将无效
 - Freeze Rotation: 冻结旋转
- 刚体会使对象在物理引擎下运动，真实模拟一个物体在现实世界中受到力后的行为。
- 通常如果使用了刚体来操作游戏对象时没必要再对其 Transform 进行操作。

7.2 Unity3D Rigidbody 详解

- 刚体能让你的游戏对象被物理引擎所控制，它能通过受到推力和扭力来实现真实的物理表现效果。所有游戏对象必须包含刚体组件来实现重力、通过脚本施加力、或者与其他对象进行交互，这一切都通过 NVIDIA 的 PhysX 物理引擎来实现。

7.2.1 属性

- Mass: 质量，单位为 Kg，建议不要让对象之间的质量差达到 100 倍以上
- Drag: 空气阻力，为 0 表示没有阻力，infinity 表示立即停止移动
- Angular Drag: 扭力的阻力，数值意义同上
- Use Gravity: 是否受重力影响
- Is Kinematic: 是否为 Kinematic 刚体，如果启用该参数，则对象不会被物理所控制，只能通过直接设置位置、旋转和缩放来操作它，一般用来实现移动平台，或者带有 HingeJoint 的动画刚体
- Interpolate: 如果你的刚体运动时有抖动，尝试一下修改这个参数，None 表示没有插值，Interpolate 表示根据上一帧的位置来做平滑插值，Extrapolate 表示根据预测的下一帧的位置来做平滑插值
- Freeze Rotation: 如果选中了该选项，那么刚体将不会因为外力或者扭力而发生旋转，你只能通过脚本的旋转函数来进行操作
- Collision Detection: 碰撞检测算法，用于防止刚体因快速移动而穿过其他对象
- Constraints: 刚体运动的约束，包括位置约束和旋转约束，勾选表示在该坐标上不允许进行此类操作

7.2.2 详细描述

- 刚体让你的游戏对象处于物理引擎的控制之下，这打开了实现真实碰撞，各种连接类型，以及其他各种效果的大门。通过给刚体施加外力来移动它，与以前的通过设置其位置变换来移动它有非常大的不同。通常情况下，你不会同时操作刚体和变换，你只会使用其中之一。
- 这两者之间最大的差异在于力 (Forces) 的使用，刚体能接受推力和扭力，变换不可以。变换同样可以实现位置变化与旋转，但这与通过物理引擎来实现是不一样的。给刚体施加力来移动他的时候同时也会影响对象的变换数值，这也是为什么只能使用这两者之一的原因，如果同时直接操作了刚体的变换，那么在执行碰撞和其他操作的时候会出问题。
- 你必须显示的将刚体组件添加到你的游戏对象上，通过菜单项 Component -> Physics -> Rigidbody 即可添加，之后对象就处于物理引擎控制之下了，他会受到重力的影响而下落，也能够通过脚本来受力，不过你可能还需要添加一个 Collider 或者 Joint 来让他表现的更像你所期望的。

1. Parenting

- 当一个对象处于物理引擎控制之下，他的运动将会与其父对象的移动半独立开。如果你移动任意的父对象，他们将会拉动刚体子对象，然而，刚体在重力及碰撞影响下还会下落。

2. Scripting

- 控制刚体的方法主要是通过脚本来施加推力和扭力，通过在刚体对象上调用 AddForce() 和 AddTorque() 方法。再次注意，当你使用物理引擎来控制刚体的时候，不要直接操作对象的变换数值。

3. Animation

- 在某些时候，主要是创建纸娃娃效果的时候，你可能需要在动画与物理控制之间进行切换。你可以将刚体设置为 IsKinematic，当设置为 Kinematic 模式，它将不再受到外力影响。这时你只能通过变换方式来操作对象，但是 Kinematic 刚体还会影响其他刚体，但他自己不会再受物理引擎控制。比如，连在 Kinematic 刚体上的 Joints 还会继续影响连接的另一个非 Kinematic 刚体，同时也能够给其他刚体产生碰撞力。

4. Colliders

- 碰撞体是另一类必须手动添加的组件，用来让对象能够发生碰撞。当两个刚体接触到一起的时候，除非两个刚体都设置了碰撞属性，否则物理引擎是不会计算他们的碰撞的。没有碰撞体的刚体在进行物理模拟的时候将会简单的穿过其他刚体。

5. Composed Colliders

- 由多个基本的碰撞体对象组合而成，扮演一个独立的碰撞体对象。当你有一个复杂的模型，而你又不能使用 Mesh Collider 的时候就可以使用组合碰撞体。

6. Continuous Collision Detection

- CCD 用来防止快速移动的物体穿过其他对象。
- 当使用默认的离散式碰撞检测时，如果前一帧时对象在墙这一面，下一帧时对象已到了墙另一面，那么碰撞检测算法将检测不到碰撞的发生，你可以将该对象的碰撞检测属性设置为 Continuous，这时碰撞检测算法将会防止对象穿过所有的静态碰撞体，设置为 Continuous Dynamic 将还会防止穿过其他也设置为 Continuous 或者 Continuous Dynamic 的刚体。
- CCD 只支持 Box, Sphere 和 Capsule 的碰撞体。

7.2.3 Use The Right Size

- 当使用物理引擎的时候，游戏对象的大小比刚体的质量更重要。如果你发现刚体的行为不是你所期望的，比如移动的太慢，漂浮，或者不能正确的进行碰撞，尝试一下修改你的模型的缩放值。Unity 的默认单位是 1 unit = 1 米，物理引擎的计算也是按照这个单位来的。比如，一个摩天大楼的倒塌与一个由积木搭成的玩具房子的倒塌是完全不一样的，所以，不同大小的对象在建模时都应该按照统一的比例。
- 对于一个人类角色模型来说，他应该有 2 米高。可以创建一个 Box 来作为参照物，默认的 Box 为 1 米，所以一个角色应该是 Box 的两倍高。
- 当然，你也可以通过修改导入模型的缩放来调整比例，如果你不能直接修改模型本身的话。在 Project 面板中选中模型，调整其 Importer 属性，注意不是变换里的缩放。
- 如果你的游戏需要你实例化具有不同缩放值的对象，你也可以调整变换里的缩放值，但是物理引擎来创建这个对象的时候会额外多做一点工作，这可能会引起一点性能问题。
- 这个问题不会太严重，但性能显然会比上面两种方法低。
- 同样要注意的是，non-uniform scales 也会引起一些问题，如果这个对象具有父对象的话。基于以上原因，尽可能的在制作模型的时候就按照 Unity 的比例来建模。

7.2.4 Hints

- 两个刚体的相对质量决定他们在碰撞的时候将会如何反应。
- 给刚体设置更大的质量并不会让它下降的更快，如果要实现这个目的，使用 Drag 参数。
- 低的阻力值使得对象看起来更重，高的阻力值使对象看起来更轻。
- 典型的 Drag 值介于 0.001(固体金属) 到 10(羽毛) 之间。
- 如果你想同时使用变换和物理来控制对象，那么给他一个刚体组件并将其设置为 Kinematic
- 如果你通过变换来移动对象，同时又想收到对象的碰撞消息，那么必须给他一个刚体组件。
- Mass (质量):
 - 学过物理的同学们都知道的吧，质量越大，惯性越大。这里的单位可以自己统一规定，但是官方给出的建议是场景中的物体质量最好不要相差 100 倍率以上。估计是防止两个质量相差太大的物体碰撞后会产生过大的速度，从而影响游戏性能吧。Drag (阻力): 这里指的是空气阻力，当游戏物体收到某个作用力的时候，这个值越大越难移动。如果设置成无限的话，物体会立即停止移动。Angular Drag (角阻力):
- 同样指的是空气阻力，只不过是用来阻碍物体旋转的。如果设置成无限的话，物体会立即停止旋转。Use Gravity (使用重力):
- 勾选了这个项，游戏对象就会受到重力影响。Is Kinematic (是否动态):
- 勾选这个选项会使游戏对象不受物理引擎的影响，但这不等同于没有刚体组件。这通常用于需要用动画控制的刚体，这样就不会因为惯性而影响动画了。Interpolate (差值类型): 如果看到刚体移动的时候一直抽风或者运动的不是很平滑，可以选择一种平滑方式:

- None (无差值): 不使用差值平滑。
- Interpolate (差值): 根据上一帧来平滑移动。
- Extrapolate (推算): 根据推算下一帧物体的位置来平滑移动。Collision Detection (碰撞检测方式):
- Discrete (离散): 默认的碰撞检测方式。但若当物体 A 运动很快的时候, 有可能前一帧还在 B 物体的前面, 后一帧就在 B 物体后面了, 这种情况下不会触发碰撞事件, 所以如果需要检测这种情况, 那就必须使用后两种检测方式。
- Continuous (连续): 这种方式可以与有静态网格碰撞器的游戏对象进行碰撞检测。
- Continuous Dynamic (动态连续): 这种方式可以与所有设置了 2 或 3 方式的游戏对象进行碰撞检测。Freeze Position/Rotation (冻结位置/旋转):
- 可以对物体在 X、Y、Z 三个轴上的位置/旋转进行锁定, 即使受到相应的力也不会改变, 但可以通过脚本来修改。
- 最后顺便再提一下恒力组件 (Constant Force), 由于比较容易理解我就不做详细介绍了。一共有 4 个参数, 分别是 Force/Relative Force (世界/相对作用力)、Torque/Relative Torque (世界/相对扭力)。这些参数代表了附加在刚体上的 XYZ 轴方向恒力的大小, 另外还要注意必须是刚体才可以添加恒力。有兴趣可以自己尝试一下给物体一个 Y 轴方向的力, 物体就会像火箭一样飞向天际, 哈哈。

7.3 Unity EventSystem 详解 (Unity Version 5.5.1)

- <https://www.jianshu.com/p/229d9abc7bd9>
- 起因: 想使用 UGUI 给项目设计一个万能拖拽系统 (2d-3d, 3d-3d, 3d-2d), 做了快两天了, 完成度 99%, 但就是有点小 BUG 无法解决, 天气又冷, 烦恼。
- 静下心来, 觉得还是对 unity 的事件系统不熟悉, 今晚坐下来读了读官网文档 (终于), 收获不小, 写在这里, 也算是一个记录:

7.3.1 Unity EventSystem

- Message System
- Input Modules
- Supported Events
- Raycasters

7.3.2 Message System (改进的消息系统)

- 基本上可以看成是以前 SendMessage 的升级版。
- 使用方法 (照抄官网):

– step1. 声明一个接口, 继承自 IEventSystemHandler

```

1  public interface ICustomMessageTarget : IEventSystemHandler {
2      // functions that can be called via the messaging system
3      void Message1();
4      void Message2();
5 }
```

– step2. 实现这个接口, 把这个脚本挂在某个物体上, 这里假设为物体 AAA

```

1  public class CustomMessageTarget : MonoBehaviour, ICustomMessageTarget {
2      public void Message1() {
3          Debug.Log ("Message 1 received");
4      }
5      public void Message2() {
6          Debug.Log ("Message 2 received");
7      }
8 }
```

- step3. 在任何脚本中使用 ExecuteEvents 静态类发送 Message，来执行接口中定义的方法

```
1 //target should be AAA  
2 ExecuteEvents.Execute<ICustomMessageTarget>(target, null, (x,y) => x.Message1());
```

* 注意：step3 里的 Execute 泛型方法，有 3 个参数，第一个参数是发送 message 的 gameobject 对象，只有当对象上有 IEventSystemHandler 实现类的时候才可以，这个例子中自然就是 AAA 物体。

* 还要注意：ExecuteEvents 静态类还有其他方法：## Static Functions

- EventSystems.ExecuteEvents.CanHandleEvent: Can the given GameObject handle the IEventSystemHandler of type T.
- EventSystems.ExecuteEvents.Execute: Execute the event of type T : IEventSystemHandler on GameObject.
- EventSystems.ExecuteEvents.ExecuteHierarchy: Recurse up the hierarchy calling Execute<T> until there is a GameObject that can handle the event.
- EventSystems.ExecuteEvents.GetEventHandler: Traverse the object hierarchy starting at root, and return the GameObject which implements the event handler of type <T>
- EventSystems.ExecuteEvents.ValidateEventData: Attempt to convert the data to type T. If conversion fails an ArgumentException is thrown.

- 名字解释都比较直白，就不翻译了。比如那个 EventSystems.ExecuteEvents.ExecuteHierarchy，是递归寻找适合的 gameobject，并执行方法。
- 说实话，比以前的 SendMessage 科学了不少，以前只能在 Hierarchy 里上下求索，现在是有目的的寻找了。
- 但.... 我看来也就仅此而已了，SendMessage 我在实际工作中从来都没用过，这个估计也不会用。为什么？有了 System.Action 谁还用这个...

7.3.3 Input Modules

- 此部分略，大致就是 unity 支持所有的输入方式，包括键盘啦，手柄啦，触摸啦等等，balabala...

7.3.4 Supported Events (支持的输入事件)

- 这部分就比较重要了，unity 事件系统支持以下 17 种输入事件

事件接口	含义
IPointerEnterHandler	pointer 进入
IPointerExitHandler	pointer 离开
IPointerDownHandler	pointer 按下
IPointerUpHandler	pointer 抬起
IPointerClickHandler	pointer 按下和抬起
IInitializePotentialDragHandler	可拖拽物体被发现，可用来初始化一些变量
IBeginDragHandler	开始拖拽
IDragHandler	拖拽中
IEndDragHandler	拖拽结束时 (when)
IDropHandler	拖拽结束位置 (where)
IScrollHandler	鼠标滚轮
IUpdateSelectedHandler	选中物体时，持续发送
ISelectHandler	物体变为被选择
IDeselectHandler	物体变为取消选择
IMoveHandler	物体移动 (左右上下等)
ISubmitHandler	submit (提交) 按钮按下
ICancelHandler	cancel (取消) 按钮按下

- 注意：这里的“pointer”可以是鼠标、touch 等一切 unity 支持的类型
- 那也就意味着，我们终于可以在 PC 和移动端共用一套代码了

7.3.5 Raycasters (射线们)

- 这也是另外一个重要的点：决定了 unity 对何种输入方式进行响应：

射线类型	含义
Graphic Raycaster	UI 使用
Physics 2D Raycaster	2D 物体组件使用，如 BoxCollider2D 等
Physics Raycaster	3D 物体使用 (UI 其实也能使用)

7.3.6 Practice - 练习和测试

- 我们来建个简单的场景：

- 场景中增加一个空物体，命名为 EventSystem，添加 EventSystem 组件，点击组件上的 Add Default Input Modules 按钮
- 场景中的摄像机上，添加 Physics Raycaster 组件
- 场景中建立一个 3d 的 Cube，和一个 2d 的 image
- 将以下脚本拖给 Cube 和 Image：

```
1  using UnityEngine;
2  using UnityEngine.EventSystems;
3  public class SupportedEventsTest : MonoBehaviour,
4      IPointerEnterHandler, IPointerExitHandler, IPointerDownHandler, IPointerUpHandler,
5      IPointerClickHandler, IInitializePotentialDragHandler, IBeginDragHandler, IDragHandler,
6      IEndDragHandler, IDropHandler, IScrollHandler, IUpdateSelectedHandler,
7      ISelectHandler, IDeselectHandler, IMoveHandler, ISubmitHandler, ICancelHandler
8  {
9      public void OnBeginDrag(PointerEventData eventData) {
10         Debug.Log("OnBeginDrag");
11     }
12     public void OnCancel(BaseEventData eventData) {
13         Debug.Log("OnCancel");
14     }
15     public void OnDeselect(BaseEventData eventData) {
16         Debug.Log("OnDeselect");
17     }
18     public void OnDrag(PointerEventData eventData) {
19         Debug.Log("OnDrag");
20     }
21     public void OnDrop(PointerEventData eventData) {
22         Debug.Log("OnDrop");
23     }
24     public void OnEndDrag(PointerEventData eventData) {
25         Debug.Log("OnEndDrag");
26     }
27     public void OnInitializePotentialDrag(PointerEventData eventData) {
28         Debug.Log("OnInitializePotentialDrag");
29     }
30     public void OnMove(AxisEventData eventData) {
31         Debug.Log("OnMove");
32     }
33     public void OnPointerClick(PointerEventData eventData) {
34         Debug.Log("OnPointerClick");
35     }
36     public void OnPointerDown(PointerEventData eventData) {
37         Debug.Log("OnPointerDown");
38     }
39     public void OnPointerEnter(PointerEventData eventData) {
40         Debug.Log("OnPointerEnter");
```

```

41 }
42     public void OnPointerExit(PointerEventData eventData) {
43         Debug.Log("OnPointerExit");
44     }
45     public void OnPointerUp(PointerEventData eventData) {
46         Debug.Log("OnPointerUp");
47     }
48     public void OnScroll(PointerEventData eventData) {
49         Debug.Log("OnScroll");
50     }
51     public void OnSelect(BaseEventData eventData) {
52         Debug.Log("OnSelect");
53     }
54     public void OnSubmit(BaseEventData eventData) {
55         Debug.Log("OnSubmit");
56     }
57     public void OnUpdateSelected(BaseEventData eventData) {
58         Debug.Log("OnUpdateSelected");
59     }
60 }

```

- 运行游戏，我们可以看到，3d 和 2d 物体都可以相应事件系统，这是由于我们给摄像机添加了 Physics Raycaster 组件。如果你换成 Graphic Raycaster，那 Cube 是不会响应的。

8 Nav Mesh

8.1 Navigation

8.1.1 Object: 物体参数面板

- Navigation Static: 勾选后表示该对象参与导航网格的烘培。
- OffMeshLink Generation: 勾选后可跳跃 (Jump) 导航网格和下落 (Drop)。

Max Slope

Step Height

0.4

Generated Off Mesh Links

Drop Height

0

Jump Distance

0

Advanced

Min Region Area

2

Width Inaccuracy %

100%

Height Inaccuracy %

100%

Height Mesh



Reset

Clear

Bal

- Radius: 具有代表性的物体半径，半径越小生成的网格面积越大。
- Height: 具有代表性的物体的高度。
- Max Slope: 斜坡的坡度。
- Ste Height: 台阶高度。
- Drop Height: 允许最大的下落距离。
- Jump Distance: 允许最大的跳跃距离。
- Min Region Area: 网格面积小于该值则不生成导航网格。
- Width Inaccuracy: 允许最大宽度的误差。
- Height Inaccuracy: 允许最大高度的误差。
- Height Mesh: 勾选后会保存高度信息，同时会消耗一些性能和存储空间。

8.2 Nav Mesh Agent：导航组建参数面板

- Radius: 物体的半径
- Speed: 物体的行进最大速度
- Acceleration: 物体的行进加速度
- Angular Speed: 行进过程中转向时的角速度。
- Stopping Distance: 离目标距离还有多远时停止。
- Auto Traverse Off Mesh Link: 是否采用默认方式度过链接路径。
- Auto Repath: 在行进某些原因中断后是否重新开始寻路。
- Height: 物体的高度。
- Base Offset: 碰撞模型和实体模型之间的垂直偏移量。
- Obstacle Avoidance Type: 障碍躲避的表现登记, None 选项为不躲避障碍, 另外等级越高, 躲避效果越好, 同时消耗的性能越多。
- Avoidance Priority: 躲避优先级。
- NavMesh Walkable: 该物体可以行进的网格层掩码。

9 基类 MonoBehaviour/自带函数以及脚本执行的生命周期

- Awake -> OnEnable -> Start ->-> FixedUpdate -> Update -> LateUpdate -> OnGUI -> OnDisable -> OnDestroy

9.1 MonoBehaviour 的生命周期：

- MonoBehaviour 是 Unity 中所有脚本的基类, 如果你使用 JS 的话, 脚本会自动继承 MonoBehaviour。如果使用 C# 的话, 你需要显式继承 MonoBehaviour。
- 在我们使用 MonoBehaviour 的时候, 尤其需要注意的是它有哪些可重写函数, 这些可重写函数会在游戏中发生某些事件的时候被调用。我们在 Unity 中最常用到的几个可重写函数是这几个:
 - Awake: 当一个脚本实例被载入时 Awake 被调用。我们大多在这个类中完成成员变量的初始化, 执行一次。
 - Start: 仅在 Update 函数第一次被调用前调用, 只执行一次。因为它是在 Awake 之后被调用的, 我们可以把一些需要依赖 Awake 的变量放在 Start 里面初始化。同时我们还大多在这个类中执行 StartCoroutine 进行一些协程的触发。要注意在用 C# 写脚本时, 必须使用 StartCoroutine 开始一个协程, 但是如果使用的是 JavaScript, 则不需要这么做。
 - Update: 当 MonoBehaviour 启用时, 其 Update 在每一帧被调用。
 - FixedUpdate: 当 MonoBehaviour 启用时, 其 FixedUpdate 在每一固定帧被调用。生命周期中可以被执行多次。FixedUpdate 函数适合调用 Rigidbody 逻辑。
 - OnEnable: 当对象变为可用或激活状态时此函数被调用。可执行多次, 每次激活对象时对象上 MonoBehaviour 上脚本会调用一次。

```
1 gameObject.SetActive(false); // 先隐藏对象
2 gameObject.SetActive(true); // 显示对象
3 // 或
4 enabled = false; // 先关闭启用
5 enabled = true; // 开启启用
6 // 都会立马会执行 OnEnable 函数
7 // 函数里适合放适配的逻辑
```

- OnDisable: 当对象变为不可用或非激活状态时此函数被调用。
- OnDestroy: 当 MonoBehaviour 将被销毁时, 这个函数被调用。

9.2 编辑器 (Editor)

- Reset: Reset 函数被调用来初始化脚本属性当脚本第一次被附到对象上，并且在 Reset 命令被使用时也会调用。
 - 编者注：Reset 是在用户点击 Inspector 面板上 Reset 按钮或者首次添加该组件时被调用。Reset 最常用于在见识面板中给定一个默认值。

9.3 第一次场景加载 (First Scene Load)

- 这些函数会在一个场景开始（场景中每个物体只调用一次）时被调用。
- Awake: 这个函数总是在任何 Start() 函数之前一个预设被实例化之后被调用，如果一个 GameObject 是非激活的 (inactive)，在启动期间 Awake 函数是不会被调用的直到它是活动的 (active)。
- OnEnable: 只有在对象是激活 (active) 状态下才会被调用，这个函数只有在 object 被启用 (enable) 后才会调用。这会发生在 MonoBehaviour 实例被创建，例如当一个关卡被加载或者一个带有脚本组件的 GameObject 被实例化。
- 注意：当一个场景被添加到场景中，所有脚本上的 Awake() 和 OnEnable() 函数将会被调用在 Start()、Update() 等它们中任何函数被调用之前。自然的，当一个物体在游戏过程中被实例化时这不能被强制执行。

9.4 第一帧更新之前 (Before the first frame update)

- Start: 只要脚本实例被启用了 Start() 函数将会在 Update() 函数第一帧之前被调用。
 - 对于那些被添加到场景中的物体，所有脚本上的 Start() 函数将会在它们中任何的 Update() 函数之前被调用，自然的，当一个物体在游戏过程中被实例化时这不能被强制执行。

9.5 在帧之间 (In between frames)

- OnApplicationPause: 这个函数将会被调用在暂停被检测有效的在正常的帧更新之间的一帧的结束时。在 OnApplicationPause 被调用后将会有额外的一帧用来允许游戏显示显示图像表示在暂停状态下。

9.6 更新顺序 (Update Order)

- 当你在跟踪游戏逻辑和状态，动画，相机位置等的时候，有几个不同的事件函数你可以使用。常见的模式是在 Update() 函数中执行大多数任务，但是也有其它的函数你可以使用。
- FixedUpdate: FixedUpdate 函数经常会比 Update 函数更频繁的被调用。它一帧会被调用多次，如果帧率低它可能不会在帧之间被调用，就算帧率是高的。所有的图形计算和更新在 FixedUpdate 之后会立即执行。当在 FixedUpdate 里执行移动计算，你并不需要 Time.deltaTime 乘以你的值，这是因为 FixedUpdate 是按真实时间，独立于帧率被调用的。
- Update: Update 每一帧都会被调用，对于帧更新它是主要的负载函数。
- LateUpdate: LateUpdate 会在 Update 结束之后每一帧被调用，任何计算在 Update 里执行结束当 LateUpdate 开始时。LateUpdate 常用为第三人称视角相机跟随。

9.7 渲染 (Rendering)

- OnPreCull: 在相机剔除场景前被调用。剔除是取决于哪些物体对于摄像机是可见的，OnPreCull 仅在剔除起作用之前被调用。
- OnBecameVisible/OnBecameInvisible: 当一个物体对任意摄像机变得可见/不可见时被调用。
- OnPreRender: 在摄像机开始渲染场景之前调用。
- OnRenderObject: 在指定场景渲染完成之后调用，你可以使用 GL 类或者 Graphics.DrawMeshNow 来绘制自定义几何体在这里。
- OnPostRender: 在摄像机完成场景渲染之后调用。
- OnRenderImage(Pro Only): 在场景徐然完成之后允许屏幕图像后期处理调用。

- OnGUI: 为了响应 GUI 事件, 每帧会被调用多次 (一般最低两次)。布局 Layout 和 Repaint 事件会首先处理, 接下来处理的是通过 Layout 和键盘/鼠标事件对应的每个输入事件。
- OnDrawGizmos: 用于可视化的绘制一些小玩意在场景视图中。

9.8 协同程序 (Coroutines)

- 正常的协同程序更新是在 Update 函数返回之后运行。一个协同程序是可以暂停执行 (yield) 直到给出的依从指令 (YieldInstruction) 完成, 写成的不同运用:
 - yield: 在所有的 Update 函数都已经被调用的下一帧该协程将持续执行。
 - yield WaitForSeconds: 一段指定的时间延迟之后继续执行, 在所有的 Update 函数完成调用的那一帧之后。
 - yield WaitForFixedUpdate: 所有脚本上的 FixedUpdate 函数已经执行调用之后持续。
 - yield WWW: 在 WWW 下载完成之后持续。
 - yield StartCoroutine: 协同程序链, 将会等到 MuFunc 函数协程执行完成首先。

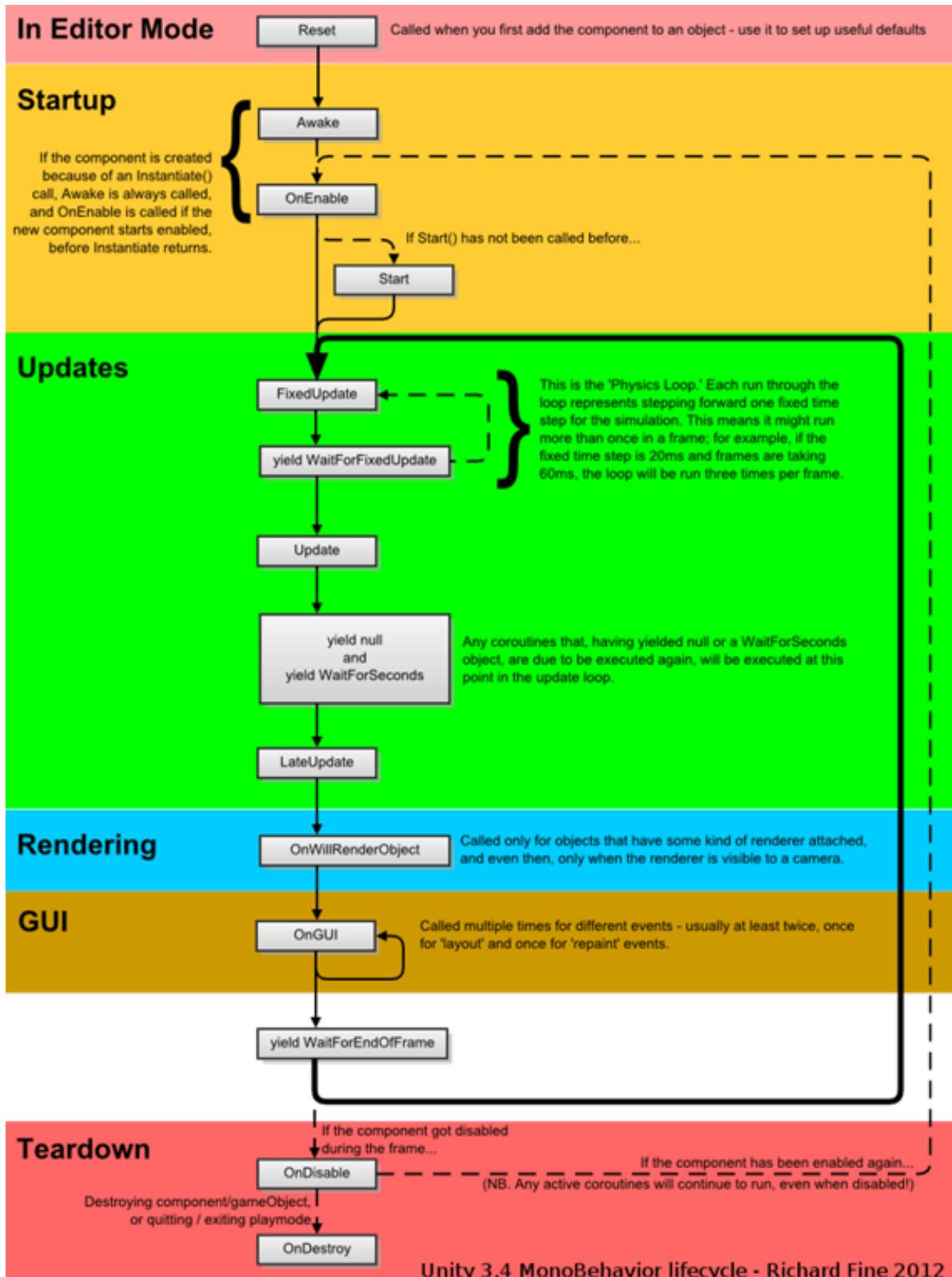
9.9 销毁 (When the Object is Destroyed)

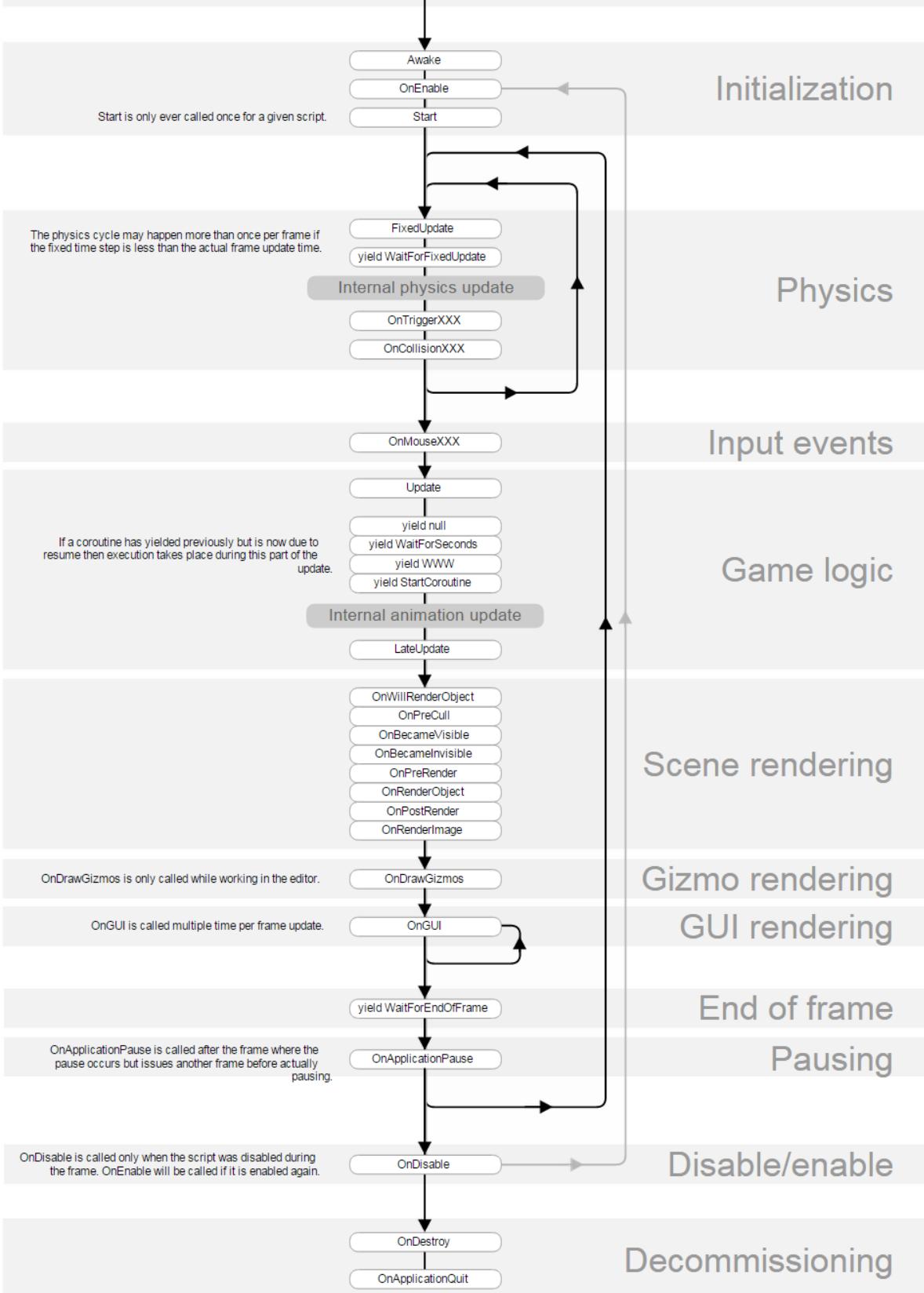
- OnDestroy: 这个函数会在一个对象销毁前一帧调用, 会在所有帧更新一个对象存在的最后一帧之后执行, 对象也许会响应 Object.Destroy 或一个场景关闭时被销毁。

9.10 退出游戏 (When Quitting)

- 这些函数会在你场景中所有的激活的物体上调用:
- OnApplicationQuit: 这个函数在应用退出之前的所有游戏物体上调用, 在编辑器 (Editor) 模式中会在用户停止 PlayMode 时调用, 在网页播放器 (web player) 中会在网页视图关闭时调用。
- OnDisable: 当行为变为非启用 (disable) 或非激活 (inactive) 时调用。

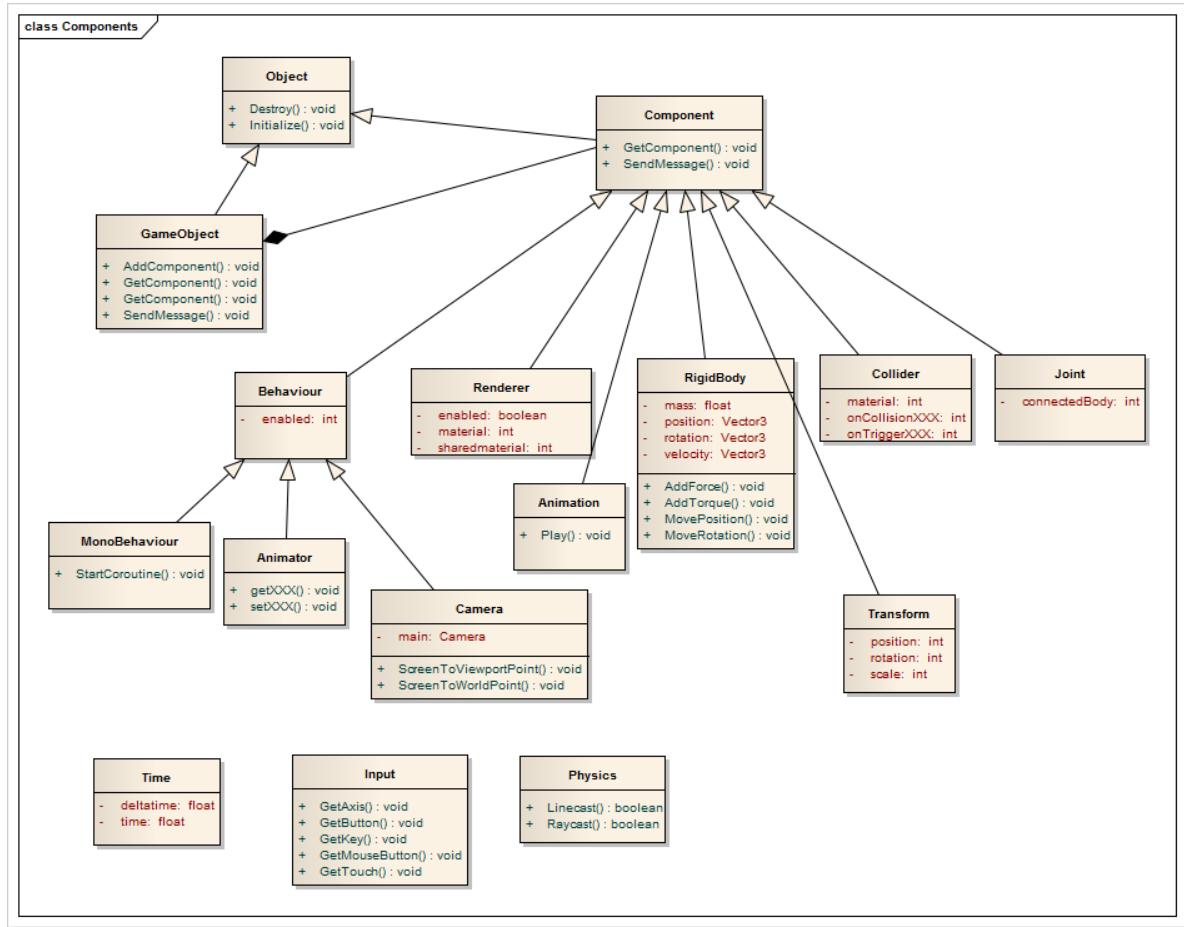
9.11 官方给出的脚本中事件函数的执行顺序如下图





10 Unity Study Notes

10.1 Unity3D 重要模块的类图



10.2 GameObject 和 Component

- 由于 Unity 是一个 Component-Based 的游戏引擎，所以游戏中所有的物体都是一个 **GameObject**，为了给这个 **GameObject** 附加各种各样的属性，所以我们引入了 **Component** 这个概念。
- GameObject** 是由 **Component** 组合而成的，**Component** 的生命周期和 **GameObject** 息息相关。一旦 **GameObject** 的 `Destroy` 方法，它的子对象和对应的所有 **Component** 都会被销毁，同时，我们也可以一次只销毁一个单独的 **Component**。
- Component** 有如下这些种类，我制作了一张表格来记录它们的用途：

Component	作用
Rigidbody 刚体	<p>刚体使物体能在物理控制下运动。</p> <p>刚体可通过接受力与扭矩，使物体像现实方式一样运动。</p> <p>任何物体想要受重力影响，受脚本施加的力的作用，或通过 NVIDIA PhysX 物理引擎来与其他物体交互，都必须包含一个刚体组件。</p>
Collider 碰撞器	<p>碰撞器是另一种组件，它和刚体一起，来使碰撞发生。</p> <p>如果两个刚体撞在一起，物理引擎将不会计算碰撞除非它们包含一个碰撞器组件。没有碰撞器的刚体，会在物理模拟中相互穿透。</p>
Renderer 渲染器	<p>渲染器是使物体显示在屏幕上。</p> <p>对于任何游戏物体或组件它的渲染器可以通过渲染器属性来访问，使用这个类可访问任意物体，网格或粒子系统的渲染器。</p> <p>渲染器可以禁用，使物体不可见（见 enabled）。并且材质可以通过它访问和修改（见 material）。</p>
AudioSource 音频源	<p>音频源(Audio Source)在场景中播放音频剪辑(Audio Clip)。</p> <p>如果音频剪辑(Audio Clip)是一个 3D 剪辑，音频源是在一个给定的位置，并会随距离衰减这样的方式进行播放。</p>
Animation 动画	<p>animation 组件用于播放动画。</p> <p>可以指定动画剪辑到动画组件并从脚本控制动画播放。在 Unity 的动画系统基于权重并且支持动画融合，叠加动画，动画混合，标签和 完全控制动画播放的各个方面。</p> <p>如果想播放一个简单的动画，可以使用 <code>Animation.Play</code>； 如果想在动画之间交叉淡入，可以使用 <code>Animation.CrossFade</code>；</p>
Animator 动画控制器	<p>Animator 组件声明了一个 Animator 控制器，用来设置角色上的行为。这些包括状态机、融合树和通过脚本控制的事件。</p>

10.3 3D 基础知识

10.3.1 点乘与叉乘

1. 点乘

- $S = U$ 点乘 $V = U$ 的模乘以 V 的模再乘以 $\cos(UV$ 之间的角度) ($a \cdot b = |a||b|\cos(\theta)$)
- 这样得出来的是一个标量 S ，是没有方向的值。但是根据这个值与 0 的比较，我们可以获得 U , V 两标量之间的关系。
 - 如果 $= 0$ ，那么向量 u 、 v 相等。
 - 如果 > 0 ，那么向量 u 、 v 之间的夹角小于 90 度。
 - 如果 < 0 ，那么向量 u 、 v 之间的夹角大于 90 度。

2. 叉乘

- 叉积（英语：Cross product）是一种在向量空间中向量的二元运算。与点积不同，它的运算结果是一个向量而不是一个标量。两个向量的叉积写作 $a * b$ ，也称作外积（英语：Outer product）或向量积（英语：Vector product）。叉积与原来的两个向量都垂直。

$$\begin{aligned} S &= U \text{ 叉乘 } V \\ &= [(UyVz - UzVy), \\ &\quad (UzVx - UxVy), \\ &\quad (UxVy - UyVx)], \end{aligned}$$

- 我们从中可以看出，我们叉乘得到的不再是一个无方向标量值。而是一个向量。那么这个向量是什么样的向量呢。
 - 我们应该去记住，两个向量叉乘得到的是一个同时垂直于这两个向量的一个新向量。

10.3.2 Quaternion

- 我们用四元组来表示旋转，一个四元组包含四个分量 x, y, z, w 。假设一个旋转的旋转轴是 $axis$, 旋转角度是 $theta$ 。那么对应的四元组 q 如下。

```
1 q.x = sin(theta / 2) * axis.x;  
2 q.y = sin(theta / 2) * axis.y;  
3 q.z = sin(theta / 2) * axis.z;  
4 q.w = cos(theta / 2);
```

10.4 map 贴图

10.4.1 normalmap

10.4.2 Bump Map 凸凹贴图

• 10 •

10.5 NavMesh

- <http://liweizhaolili.blog.163.com/blog/static/16230744201271225812998/>

2

10.6 粒子系统

1

10.7 references

- 入门 1-5 http://www.cnblogs.com/neverdie/p/How_To_Learn_Unity3D.html
- Unity Manual <http://docs.unity3d.com/Manual/index.html>
- <https://unity3d.com/cn/learn>
- <http://www.cnblogs.com/neverdie/>
- <http://subject.manew.com/learn/index.html>
- 摄像机 raycast 拉近与拉远 momo <http://www.xuanyusong.com/archives/1991>
-
-

10.8 comments

10.9 【Unity3D 基础教程】给初学者看的 Unity 教程（一）

10.9.1 Unity 3d (just so that don't occupy a chapter)

- 熟练者，关注 Unity 圣典和 Unity User Manual，在这一阶段，要把在第一阶段所忽略的内容进行选择性的补充学习。
- 进阶者，关注 Unity 社区，Unity Answers，Unity Wiki 和知乎的 Unity 板块，在这一阶段，要对 Unity 的各种细节问题，优化，底层原理和新的技术方案都要进行思考和学习。在以上几个模块中，知乎的 Unity 板块尤其值得关注，干货满满而且都是中文，建议通读。看书。对知识点进行查漏补缺，我个人用的是《unity 5.x 从入门到精通》。
- 在进阶者这一阶段，你才可以对一些中间件进行学习，具体学习什么样的 Asset，还要由你的项目需求决定，不过无论如何还是推荐学习这几个 Asset：Behavior Designer (AI)，DoTween (Tween 动画)，PlayMaker (可视化编程)，Shader Forge (可视化的 Shader 编写) 和 Elementals (粒子特效)。
- 如果你还觉得不过瘾的话，就可以尝试反编译一些市面上流行的 Unity3D 游戏来获取代码，毕竟真实生产环境中的代码才是最值得深入研究的，在这部分我还是要推荐啪啪三国的代码，相当整洁。
- 个人觉得就 Unity 学习，掌握几个很重要的点，
 1. 基本编程语言功底，C#、js、数据结构、算法
 2. Unity 资源流原理 (Unity 如何处理资源关系的，mate、library、prefab 之间的关系 dll 如何被引用等等)，基于这些关系去构建自己的资源管理结构。
 3. 做好对象生命周期管理 (利于管理内存、利于更加灵活结构化)
 4. 善用 unity 文档和 answer
 5. 我是很不赞同去看网络上的 demo 来学习 Unity 的，Untiy 的思想是可以用任意脚本对象去构建一个项目，这是非常方便非常灵活的，这是个很大的优点，然而这对于新手来说是很可怕的缺点，就像往一个容器里放入很多散落的个体，然而在项目这些个体又是项目有引用关系的，那么个体之间的偶合关系处理的不好那这就会成一个项目很大的问题。然而目前网络上的 demo，大部分都是不够结构化不够框架化的初级 demo。于初学者很悲催的一点是，如果一开始你从这些 demo 上去理解 Unity，你就会进入完全基于实现的误区。很多问题都靠挂载脚本来解决，于产品级项目这样的思想是很可怕的，会严重影响产品迭代速度、管理成本以及时间成本。甚至很多教学视频往往也是把新手引入了这个误区。
- Model-View-Controller (MVC) 是一种组合设计模式，它体现了一种关注点分离 (Separation of concerns, SoC) 的思想。MVC 主要把逻辑层和表现层进行了解耦，将一个问题划分成了不同的关注点。增强了应用的稳定性，易修改性和易复用性。

10.9.2 重要类及其关系

10.9.3 Input 输入

- Unity 支持，键盘，操纵杆和游戏手柄输入。
- 当创建时，每个项目都具有下面的默认输入轴：
 - Horizontal and Vertical are mapped to w, a, s, d and the arrow keys.
 - 水平和垂直被映射到 w, a, s, d 键和方向键
 - Fire1, Fire2, Fire3 are mapped to Control, Option (Alt), and Command, respectively.
 - Fire1, Fire2, Fire3 被分别映射到 Ctrl, Option (Alt) 和 Command 键
 - Mouse X and Mouse Y are mapped to the delta of mouse movement.
 - Mouse X 和 Mouse Y 被映射到鼠标移动增量
 - Window Shake X and Window Shake Y is mapped to the movement of the window.
 - Window Shake X 和 Window Shake Y 被映射到窗口的移动

10.9.4 Time

- Time 类是 Unity 中的一个全局变量，它记载了和游戏相关的时间，帧数等数据。
- Time 类包含一个非常重要的变量叫 deltaTime. 这个变量包含从上次调用 Update 或 FixedUpdate 到现在的时间 (根据你是放在 Update 函数还是 FixedUpdate 函数中).(另注: Update 每帧调用一次)
- 依照上面的例子，使得物体在一个匀速的速度下旋转，不依赖帧的速率，如下：

```
1 using UnityEngine;
2 using System.Collections;
3
4 public class example : MonoBehaviour {
5     void Update() {
6         transform.Rotate(0, 5 * Time.deltaTime, 0);
7     }
8 }
```

- 当然了，在使用 Time 这个类的时候，我们也要记住使用各种各样的 Lerp (线性插值) 函数来减少自己的工作量，在 Unity3D 中，Vector3, Vector2, Color 等类都提供了相应的 Lerp 函数给我们调用。

10.9.5 Physics 和 Transform

- Physics 类是 Unity 重的一个工具函数类，它主要提供了 Linecast 和 Raycast 两种射线投射方式。
 - 其中 Linecast 是以投射的起始位置和终止位置为参数，来判断这个投射有没有和某个 Collider 发生了碰撞。
 - 而 Raycast 则是以投射的起始位置和投射方向为参数，来判断这个投射有没有和某个 Collider 发生了碰撞。
- 相应的实例可以看下面的这一段程序：

```
1 using UnityEngine;
2 using System.Collections;
3
4 public class Example : MonoBehaviour {
5     void Update() {
6         // 使用 Raycast
7         Vector3 fwd = transform.TransformDirection(Vector3.forward);
8         if (Physics.Raycast(transform.position, fwd, 10))
9             print("There is something in front of the object!");
10        // 使用 Linecast
11        Transform target;
12        if (!Physics.Linecast(transform.position, target.position))
13            ProcessData.AndDoSomeCalculations();
14    }
15 }
```

- 在 Physics 这个模块中，有三个 Component 是至关重要的，分别是 Rigidbody, Collision, Joint。在新的版本中，又引入了 Rigidbody2D, Collision2D, Joint2D 这些 Component 来处理 2D 中的 Physics 事件。
- 这三个类都是处理物理相关的事件的，那么它们有什么区别呢？
 - Rigidbody 是作为一个受力物体而存在的，所以可以向一个 Rigidbody 施加 Force (力), Drag (阻力)。同时 Rigidbody 还有 velocity (速度), mass (质量), position (位置), 旋转 (rotation) 等等。
 - Collider 是为了处理物理中的碰撞事件而出现的类，就像上面表格中所说的，如果没有 Collider，两个 Rigidbody 之间是无法发生碰撞的。对同一个 GameObject 可以绑定多个 Collider 构建更加复杂的碰撞体结构。
 - * Collider 另外一个很值得注意的就是我们可以为 Collider 设置 material，即 Collider 的物理材质。物理材质用于调整摩擦力和碰撞单位之间的反弹效果。
 - * 当发生碰撞时，会触发毁掉函数 OnCollisionEnter, OnCollisionStay, OnCollisionExit 等等。这几个函数与 OnTriggerXXX 的区别会在接下来的博客中提到。
 - Joint 用于连接两个 Rigidbody，当 Joint 断掉的时候会触发 OnJointBreak 的回调函数。

10.10 【Unity3D 基础教程】给初学者看的 Unity 教程 (二): 所有脚本组件的基类 – MonoBehaviour 的前世今生

10.10.1 引子

- 上一次我们讲了 GameObject, Component, Time, Input, Physics，其中 Time, Input, Physics 都是 Unity 中的全局变量。GameObject 是游戏中的基本物件。GameObject 是由 Component 组合而成的，GameObject 本身必须有 Transform 的 Component，这也加深了我们对 GameObject 的理解，即 GameObject 是游戏场景中真实存在，而且有位置的一个物件。
- 但是我们怎么操纵这个 GameObject 呢？这就需要引入脚本组件了，也就是今天讲的 MonoBehaviour

10.10.2 MonoBehaviour 的生命周期

- MonoBehaviour 是 Unity 中所有脚本的基类，如果你使用 JS 的话，脚本会自动继承 MonoBehaviour。如果使用 C# 的话，你需要显式继承 MonoBehaviour。
- 在我们使用 MonoBehaviour 的时候，尤其需要注意的是它有哪些可重写函数，这些可重写函数会在游戏中发生某些事件的时候被调用。我们在 Unity 中最常用到的几个可重写函数是这几个：
 - Awake：当一个脚本实例被载入时 Awake 被调用。我们大多在这个类中完成成员变量的初始化
 - Start：仅在 Update 函数第一次被调用前调用。因为它是在 Awake 之后被调用的，我们可以把一些需要依赖 Awake 的变量放在 Start 里面初始化。同时我们还大多在这个类中执行 StartCoroutine 进行一些协程的触发。要注意在用 C# 写脚本时，必须使用 StartCoroutine 开始一个协程，但是如果使用的是 JavaScript，则不需要这么做。
 - Update：当 MonoBehaviour 启用时，其 Update 在每一帧被调用。
 - FixedUpdate：当 MonoBehaviour 启用时，其 FixedUpdate 在每一固定帧被调用。
 - OnEnable：当对象变为可用或激活状态时此函数被调用。
 - OnDisable：当对象变为不可用或非激活状态时此函数被调用。
 - OnDestroy：当 MonoBehaviour 将被销毁时，这个函数被调用。
- 下面用一张图来更形象地说明一下这几个类的在 MonoBehaviour 的生命周期中是如何被调用的：

10.10.3 MonoBehaviour 的那些坑

- 私有 (private) 和保护 (protected) 变量只能在专家模式中显示。属性不被序列化或显示在检视面板。
- 不要使用命名空间 (namespace)
- 记得使用缓存组件查找，即在 MonoBehaviour 的长远方法中经常被访问的组件最好在把它当作一个私有成员变量存储起来。
- 在游戏里经常出现需要检测敌人和我方距离的问题，这时如果要寻找所有的敌人，显然要消耗的运算量太大了，所以最好的办法是将攻击范围使用 Collider 表示，然后将 Collider 的 isTrigger 设置为 True。最后使用 OnTriggerEnter 来做攻击范围内的距离检测，这样会极大提升程序性能。

- 10.11 【Unity3D 基础教程】给初学者看的 Unity 教程（三）：通过制作 Flappy Bird 了解 Native 2D 中的 Sprite, Animation
- 10.12 【Unity3D 基础教程】给初学者看的 Unity 教程（四）：通过制作 Flappy Bird 了解 Native 2D 中的 Sprite, Animation
- 10.13 【Unity3D 基础教程】给初学者看的 Unity 教程（五）：详解 Unity3d 中的协程（Coroutine）

10.13.1 为什么需要协程

- 在游戏中有许多过程（Process）需要花费多个逻辑帧去计算。
 - 你会遇到“密集”的流程，比如说寻路，寻路计算量非常大，所以我们通常会把它分割到不同的逻辑帧去进行计算，以免影响游戏的帧率。
 - 你会遇到“稀疏”的流程，比如说游戏中的触发器，这种触发器大多数时候什么也不做，但是一旦被调用会做非常重要的事情（比图说游戏中自动开启的门就是在门前放了一个 Empty Object 作为 trigger，人到门前就会触发事件）。
- 不管什么时候，如果你想创建一个能够历经多个逻辑帧的流程，但是却不使用多线程，那你就需要把一个任务来分割成多个任务，然后在下一帧继续执行这个任务。
- 比如，A* 算法是一个拥有主循环的算法，它拥有一个 open list 来记录它没有处理到的节点，那么我们为了不影响帧率，可以让 A* 算法在每个逻辑帧中只处理 open list 中一部分节点，来保证帧率不被影响（这种做法叫做 time slicing）。
- 再比如，我们在处理网络传输问题时，经常需要处理异步传输，需要等文件下载完毕之后再执行其他任务，一般我们使用回调来解决这个问题，但是 Unity 使用协程可以更加自然的解决这个问题，如下边的程序：

```
1 private IEnumerator Test() {  
2     WWW www = new WWW(ASSESS_URL);  
3     yield return www;  
4     AssetBundle bundle = www.assetBundle;  
5 }
```

10.13.2 协程是什么

- 从程序结构的角度来讲，协程是一个有限状态机，这样说可能并不是很明白，说到协程（Coroutine），我们还要提到另一样东西，那就是子例程（Subroutine），子例程一般可以指函数，函数是没有状态的，等到它 return 之后，它的所有局部变量就消失了，但是在协程中我们可以在一个函数里多次返回，局部变量被当作状态保存在协程函数中，直到最后一次 return，协程的状态才被清除。
- 简单来说，协程就是：你可以写一段顺序的代码，然后标明哪里需要暂停，然后在下一帧或者一段时间后，系统会继续执行这段代码。

10.13.3 协程怎么用？

- 一个简单的 C# 代码，如下：

```
1 IEnumerator LongComputation() {  
2     while(someCondition) {  
3         /* 做一系列的工作 */  
4         // 在这里暂停然后在下一帧继续执行  
5         yield return null;  
6     }  
7 }
```

10.13.4 协程是怎么工作的

- 注意上边的代码示例，你会发现一个协程函数的返回值是 IEnumerator，它是一个迭代器，你可以把它当成指向一个序列的某个节点的指针，它提供了两个重要的接口，分别是 Current（返回当前指向的元素）和 MoveNext()（将指针向前移动一个单位，如果移动成功，则返回 true）。IEnumerator 是一个 interface，所以你不用担心的具体实现。

- 通常，如果你想实现一个接口，你可以写一个类，实现成员，等等。迭代器块（iterator block）是一个方便的方式实现 IEnumerator 没有任何麻烦-你只是遵循一些规则，并实现 IEnumerator 由编译器自动生成。
- 一个迭代器块具备如下特征：
 - 返回 IEnumerator
 - 使用 yield 关键字
- 所以 yield 关键词是干嘛的？它声明序列中的下一个值或者是一个无意义的值。如果使用 yield x (x 是指一个具体的对象或数值) 的话，那么 moveNext 返回为 true 并且 current 被赋值为 x，如果使用 yield break 使得 moveNext() 返回 false。
- 那么我举例如下，这是一个迭代器块：

```

1 public void Consumer() {
2     foreach(int i in Integers()) {
3         Console.WriteLine(i.ToString());
4     }
5 }
6 public IEnumerable<int> Integers() {
7     yield return 1;
8     yield return 2;
9     yield return 4;
10    yield return 8;
11    yield return 16;
12    yield return 16777216;
13 }
```

- 注意上文在迭代的过程中，你会发现，在两个 yield 之间的代码只有执行完毕之后，才会执行下一个 yield，在 Unity 中，我们正是利用了这一点，我们可以写出下面这样的代码作为一个迭代器块：

```

1 IEnumerator TellMeASecret() {
2     PlayAnimation("LeanInConspiratorially");
3     while(playingAnimation)
4         yield return null;
5
6     Say("I stole the cookie from the cookie jar!");
7     while(speaking)
8         yield return null;
9
10    PlayAnimation("LeanOutRelieved");
11    while(playingAnimation)
12        yield return null;
13 }
```

- 然后我们可以使用下文这样的客户代码，来调用上文的程序，就可以实现延时的效果。

```

1 IEnumerator e = TellMeASecret();
2 while(e.MoveNext()) {
3     // do whatever you like
4 }
```

10.13.5 协程是如何实现延时的？

- 如你所见，yield return 返回的值并不一定是有意义的，如 null，但是我们更感兴趣的是，如何使用这个 yield return 的返回值来实现一些有趣的效果。
- Unity 声明了 YieldInstruction 来作为所有返回值的基类，并且提供了几种常用的继承类，如 WaitForSeconds（暂停一段时间继续执行），WaitForEndOfFrame（暂停到下一帧继续执行）等等。更巧妙的是 yield 也可以返回一个 Coroutine 真身，Coroutine A 返回一个 Coroutine B 本身的时候，即等到 B 做完了再执行 A。下面有详细说明：

- Normal coroutine updates are run after the Update function returns. A coroutine is a function that can suspend its execution (yield) until the given YieldInstruction finishes. Different uses of Coroutines:
 - yield; The coroutine will continue after all Update functions have been called on the next frame.
 - yield WaitForSeconds(2); Continue after a specified time delay, after all Update functions have been called for the frame
 - yield WaitForFixedUpdate(); Continue after all FixedUpdate has been called on all scripts
 - yield WWW Continue after a WWW download has completed.
 - yield StartCoroutine(MyFunc); Chains the coroutine, and will wait for the MyFunc coroutine to complete first.

- 实现延时的关键代码是在 StartCoroutine 里面, 以为笔者也没有见过 Unity 的源码, 那么我只能猜想 StartCoroutine 这个函数的内部构造应该是这样的:

```

1 List<IEnumerator> unblockedCoroutines;
2 List<IEnumerator> shouldRunNextFrame;
3 List<IEnumerator> shouldRunAtEndOfFrame;
4 SortedList<float, IEnumerator> shouldRunAfterTimes;
5
6 foreach(IEnumerator coroutine in unblockedCoroutines) {
7     if(!coroutine.MoveNext())
8         // This coroutine has finished
9         continue;
10
11    if(!coroutine.Current is YieldInstruction) {
12        // This coroutine yielded null, or some other value we don't understand; run it next
13        shouldRunNextFrame.Add(coroutine);
14        continue;
15    }
16
17    if(coroutine.Current is WaitForSeconds) {
18        WaitForSeconds wait = (WaitForSeconds)coroutine.Current;
19        shouldRunAfterTimes.Add(Time.time + wait.duration, coroutine);
20    } else if(coroutine.Current is WaitForEndOfFrame) {
21        shouldRunAtEndOfFrame.Add(coroutine);
22    } else /* similar stuff for other YieldInstruction subtypes */
23 }
24 unblockedCoroutines = shouldRunNextFrame;

```

- 当然了, 我们还可以为 YieldInstruction 添加各种的子类, 比如一个很容易想到的就是 yield return new WaitForSeconds("GameOver") 来等待某个消息的触发, 关于 Unity 的消息机制可以参考这篇文章: 【Unity3D 技巧】在 Unity 中使用事件/委托机制 (event/delegate) 进行 GameObject 之间的通信 (二) : 引入中间层 NotificationCenter。

10.13.6 还有些更好玩的?

- 第一个有趣的地方是, yield return 可以返回任意 YieldInstruction, 所以我们可以在这里加上一些条件判断:

```

1 YieldInstruction y;
2
3 if (something)
4     y = null;
5 else if (somethingElse)
6     y = new WaitForEndOfFrame();
7 else
8     y = new WaitForSeconds(1.0f);
9
10 yield return y;

```

- 第二个，由于一个协程只是一个迭代器块而已，所以你也可以自己遍历它，这在一些场景下很有用，例如在对协程是否执行加上条件判断的时候：

```

1 IEnumarator DoSomething() {
2     /* ... */
3 }
4
5 IEnumarator DoSomethingUnlessInterrupted(){
6     IEnumarator e = DoSomething();
7     bool interrupted = false;
8     while(!interrupted) {
9         e.MoveNext();
10        yield return e.Current;
11        interrupted = HasBeenInterrupted();
12    }
13 }
```

- 第三个，由于协程可以 yield 协程，所以我们可以自己创建一个协程函数，如下：

```

1 IEnumarator UntilTrueCoroutine(Func fn) {
2     while (!fn())
3         yield return null;
4 }
5
6 Coroutine UntilTrue(Func fn) {
7     return StartCoroutine(UntilTrueCoroutine(fn));
8 }
9
10 IEnumarator SomeTask() {
11     /* ... */
12     yield return UntilTrue(() => _lives < 3);
13     /* ... */
14 }
```

10.14 【Unity3D 基础教程】给初学者看的 Unity 教程（六）：理解 unity 的新 GUI 系统（UGUI）

- UGUI 是 Unity 在 4.6 中引入的新的 GUI 系统，与传统的中间件 NGUI 相比，这套新 GUI 系统有几个核心亮点：
 - 放弃了 Atlas 的概念，使用 Packing Tag 的方式进行图集的规划
 - 放弃了 depth 来确定 UI 显示层级的概念，使用 Hierarchy 的 SiblingIndex 来确定先是层级
 - 使用 RectTransform 来代替以前 NGUI 中的 UIAnchor，分辨率适配更加简单
 - 使用 LayoutGroup 代替以前的 NGUI 中的 Grid，实现了更灵活的布局
- UGUI 的类继承结构还是挺复杂的，如果你想对此有了解的话，可以参考我的两个类图：
 - Unity GUI 链接：UnityEngine.UI 系统基础类架构图 <https://www.processon.com/view/link/55421d9>
 - Unity Event System 链接：UnityEngine Event & EventSystem 系统架构图 <https://www.processon.com/view/55421efce4b01376faa9dbfe>

10.15 【Unity3D 基础教程】给初学者看的 Unity 教程（七）：在 unity 中构建健壮的单例模式（Singleton）

10.15.1 为什么需要单例模式

- 游戏中需要单例有以下几个原因：
 - 我们需要在游戏开始前和结束前做一些操作，比如网络的链接和断开，资源的加载和卸载，我们一般会把这部分逻辑放在单例里。

- 单例可以控制初始化和销毁顺序，而静态变量和场景中的 GameObject 都无法控制自己的创建和销毁顺序，这样就会造成很多潜在的问题。
- Unity3D 的 GameObject 需要动态创建。而不是固定在场景里，我们需要使用单例来创建 GameObject。
- Unity3D 的场景中的各个 GameObject 需要从单例中存取数据。

10.15.2 单例的设计原则

- 在设计单例的时候，我并不建议采取延迟初始化的方案，正如云风所说：对于单件的处理，采用静态对象和惰性初始化的方案，简直就是 C++ 程序员的陋习。**Double Checked Locking is broken**，相信很多人都读过了。过于依赖语法糖，通常就会造成这种结果。其实让程序有明显的初始化和退出阶段，是很容易被规划出来的。把单件 (singleton) 的处理放在正确的时机，以正确的次序来处理并非难事。
- 我们应该在程序某处明确定义单例是否被初始化，在初始化执行完毕后再执行正常的游戏逻辑
 - 尽量避免多线程创建单例带来的复杂性
 - 在某处定义了一定的初始化顺序后，可以在游戏结束的时候按照相反的顺序销毁这些单例

10.15.3 设计单例的基类

- 在 Unity 中，我们需要一个基类来为所有单例的操作提供统一的接口，同时，我们还要让所有单例继承 MonoBehaviour，只有这样才能让单例自由使用协程这一特性。
- 基类设计如下，

```

1  using System;
2  using UnityEngine;
3
4  namespace MiniWeChat {
5      [RequireComponent(typeof(GameRoot))]
6
7      public class Singleton<T> : MonoBehaviour where T : Singleton<T> {
8          private static T _instance;
9
10         public static T GetInstance() {
11             return _instance;
12         }
13
14         public void SetInstance(T t) {
15             if (_instance == null) {
16                 _instance = t;
17             }
18         }
19
20         public virtual void Init() {
21             return;
22         }
23
24         public virtual void Release() {
25             return;
26         }
27     }
28 }
```

10.15.4 设计单例的管理类

- 除了设计基类之外，还需要设计一个让所有基类初始化和销毁的类，我们把这个类叫做 GameRoot，并且把它绑定在一个名为 GameRoot 的 GameObject 上，并且把这个 GameObject 放在游戏进入的 Main 场景中。
- GameRoot 类设计如下，

```

1 namespace MiniWeChat {
2     public class GameRoot : MonoBehaviour {
3         private static GameObject _rootObj;
4         private static List<Action> _singletonReleaseList = new List<Action>();
5
6         public void Awake() {
7             _rootObj = gameObject;
8             GameObject.DontDestroyOnLoad(_rootObj);
9             StartCoroutine(InitSingletons());
10        }
11
12        /// <summary>
13        /// 在这里进行所有单例的销毁
14        /// </summary>
15        public void OnApplicationQuit() {
16            for (int i = _singletonReleaseList.Count - 1; i >= 0; i--) {
17                _singletonReleaseList[i]();
18            }
19        }
20
21        /// <summary>
22        /// 在这里进行所有单例的初始化
23        /// </summary>
24        /// <returns></returns>
25        private IEnumerator InitSingletons() {
26            yield return null;
27            // Init Singletons
28        }
29
30        private static void AddSingleton<T>() where T : Singleton<T> {
31            if (_rootObj.GetComponent<T>() == null) {
32                T t = _rootObj.AddComponent<T>();
33                t.SetInstance(t);
34                t.Init();
35                singletonReleaseList.Add(delegate() {
36                    t.Release();
37                });
38            }
39        }
40
41        public static T GetSingleton<T>() where T : Singleton<T> {
42            T t = _rootObj.GetComponent<T>();
43            if (t == null) {
44                AddSingleton<T>();
45            }
46            return t;
47        }
48    }
49 }

```

10.15.5 如何拓展新的单例

- 有了以上两个类之后，当我们需要新创建一个类的时候，就可以继承 `Singleton<T>` 来创建新的单例，重写 `Init` 和 `Release` 方法，同时在 `GameRoot` 的 `InitSingleton` 方法的适当顺序执行 `AddSingleton<T>` 方法即可。

10.16 场景视图 (Scene View) 导航

场景视图 (Scene View) 是你的交互式沙箱。你可以使用场景视图 (Scene View) 选择和放置环境、玩家、相机、敌人和所有其他游戏对象 (GameObjects)。

在场景视图 (Scene View) 中调动和操纵对象是 Unity 最重要的一些功能，因此，能够迅速使用它们至关重要。

场景视图 (Scene View) 有一个导航控件集，可帮助你快速高效地四处移动。

10.16.1 上下左右箭头键

您可以使用箭头键 (Arrow Keys) 在场景中四处移动，就像“走路”穿过一样。向上和向下箭头在相机面向的方向向前或向后移动相机。

左右箭头横向平移视图。按住 Shift 键和一个箭头键，可更快地移动。

10.16.2 使用手形工具

你还可以发现手形工具 (Hand Tool) (快捷键: Q) 的功用，特别是使用单键鼠标时。选择手形工具 (Hand tool) 后：

场景视图 (Scene View) 的右上角是场景小图示 (Scene Gizmo)，显示场景相机 (Scene Camera) 的当前定位，让你快速改变视角。

按住 Shift 键可提高移动和缩放的速率。

10.16.3 漫游模式

漫游模式可让您像许多第一人称视角游戏一样来浏览场景视图 (Scene View)。

单击并按住鼠标右键进入漫游模式。

现在您可以使用鼠标将视图四处移动，使用 WASD 键向前后左右移动，使用 Q 和 E 键向上或向下移动。

按住 Shift 键可移动得更快。

漫游模式专为透视图模式而设计。在等距模式中，按住鼠标右键并移动鼠标，则会旋转相机。

10.16.4 场景小图示 (Scene Gizmo)

场景视图 (Scene View) 右上角是场景小图示 (Scene Gizmo)，它显示了场景视图相机 (Scene View Camera) 当前视角方向，可让你快速修改视角。

小图示

(Gizmo) 的每个彩色“支脚”代表一根几何轴。

你可以点击任何方向杆，将场景视图相机 (Scene View Camera) 更改为该方向。点击场景小图示 (Scene Gizmo) 的中央或其下方的文本，可在等距模式和透视图模式之间切换。

你也可以常常按下 Shift 键并点击场景小图示 (Scene Gizmo) 的中央，以获得“好”的透视图，其查看场景的视角位于侧面略上方。

10.17 定位游戏对象 (GameObjects)

构建游戏时，会在游戏世界中放置许多不同的对象。此时，使用工具栏 (Toolbar) 中的转换工具 (Transform Tools)，可转换、旋转及缩放单个游戏对象。

每个游戏对象的周围都有相应的小图示 (Gizmo)。可使用鼠标操作任何 (Gizmo) 轴来修改游戏对象 (GameObjects) 的转换组件 (Transform Component)，也可以在检视器 (Inspector) 中转换组件 (Transform Component) 的数字字段中直接输入值。

10.17.1 聚焦游戏对象

先将场景视图的相机 (Scene View Camera) 焦距一个对象，或者在层级视图中选择一个游戏对象 (GameObject)，然后再将鼠移到场景视图上操纵该对象，此时视图将以对象为中心移动，这点非常有用。

选择任何游戏对象 (GameObject) 并按 F 键，这样场景视图 (Scene View) 和枢轴点会以所选对象为中心，该操作称为“框选 (Frame Selection)”。

10.17.2 移动、旋转和缩放

移动、旋转和缩放是场景视图 (Scene View) 导航中的关键操作，因此，Unity 提供了几个备选方法，以在执行时提供最大方便。

以下是对一些要点的概述：

按住鼠标右键进入漫游 (Flythrough) 模式。

用鼠标和方向键（向上和向下要加上 Q 和 E 键）进入第一视角视图导航。

选择任何游戏对象 (GameObject) 并按 F 键。这将确定所选对象在场景视图的中心位置及轴点。

使用箭头键在 X/Z 平面上移动。

按住 Alt 键和鼠标左键拖动，使相机围绕当前轴点旋转。

按住 Alt 键和鼠标滚轮拖动，拖动场景视图 (Scene View) 相机。

按住 Alt 键和鼠标右键拖动，缩放场景视图 (Scene View)。此效果与滚动鼠标滚轮的效果相同。

点击并拖动小图示 (Gizmo) 中心，以同时在所有轴上操纵对象。

在转换 (Translate) 小图示的中心有三个小正方形，可用来在相同平面（即可同时移动两个轴，第三个轴保持不动）拖动对象。

如果你有一个三键鼠标，则可以点击鼠标中键来调整最后调整的轴（该轴变成黄色），而无需直接点击该轴。

使用缩放工具时要小心，因为非统一缩放，可能会导致子对象的缩放比例失衡。

10.17.3 小图示显示切换器 (Gizmo Display Toggles)

小图示显示切换器 (Gizmo Display Toggles) 用来定义任何变换小图示 (Transform Gizmo) 的位置。

可使用

它在不同视图模式中查看场景：纹理、线框、RGB、夸张等等。还可以看（听）场景视图 (Scene View) 中的游戏照明、游戏元素及声音。

定位时，居中 (Center) 将小图示 (Gizmo) 定位在对象范围的中心。枢轴 (Pivot) 将小图示 (Gizmo) 定位在网格 (Mesh) 的实际枢轴点。

旋转时，局部 (Local) 将相对于对象的旋转保持小图示 (Gizmo) 的旋转。全局 (Global) 强制小图示 (Gizmo) 为世界空间的方向。

10.17.4 单位对齐

使用转换工具 (Translate Tool) 拖动任何小工具轴 (Gizmo Axis) 时，你可以按住控制键，以对齐设置 (Snap Settings) 中定义的增量。

你可以使用菜单编辑-> 对齐设置...更改单位对齐所用的单位距离：

10.17.5 表面对齐

使用转换工具 (Translate Too) 在中心拖动时，可以按住 Shift 键，这可将对象与任何碰撞体 (Collider) 的交叉点对齐。使对象精确定位变得更迅速。

10.17.6 查看旋转

使用旋转工具 (Rotate Tool) 时，可以按住 Shift 将对象朝向任何碰撞体 (Collider) 表面上的一点旋转。这可使对象相对于另一个对象的定向变得简单。

10.17.7 顶点对齐

你可以使用一个称为顶点对齐 (Vertex Snapping) 的功能来轻松组装你的游戏世界。该功能可以让你取给定网格的任何顶点，然后用鼠标将该顶点放到你所选的任何其他网格的任何顶点上。使用该功能可以快速地组装你的游戏世界。例如，你可以在高精度竞速游戏中放置赛道，在网格的顶点添加动力道具。

使用顶

点对齐 (Vertex Snapping) 组装赛道。

在 Unity 中使用顶点对齐非常简单，只需按照以下步骤操作：

1. 选择您要操纵的网格，确保变换工具 (Transform Tool) 处于活动状态。
2. 长按 V 键以激活顶点对齐模式。
3. 将光标移到您想用作枢轴点的网格的顶点上。
4. 光标在想要的顶点上时按住鼠标左键，然后将网格拖到另一个网格的任何其他顶点旁边。

5. 对结果满意时，松开鼠标按键和 V 键。

6. Shift-V 用来切换该功能。

您可以将顶点与顶点对齐、顶点与表面对齐、枢轴与顶点对齐。

10.18 视图模式

场景视图 (Scene View) 控制条可让您选择查看场景的各种选项及控制是否启用灯光和音频。这些控制只影响开发过程中的场景视图，对构建的游戏无影响。

10.18.1 绘图模式 (Draw Mode)

第一个下拉菜单选择哪种绘图模式 (Draw Mode) 将用来描绘场景。

纹理: 显

示表面及其可见纹理。

线框: 用线框绘制网格。

纹理-线框: 显示带纹理且有线框覆盖的网格。

渲染路径: 使用颜色代码显示每个对象的渲染路径: 绿色表示延时光照、黄色表示正向渲染，红色表示顶点光明。

光照贴图分辨率: 在场景上覆盖棋盘格以显示光照贴图的分辨率。

10.18.2 渲染模式 (Render Mode)

下一个下拉菜单选择四种渲染模式 (Render Modes) 的哪种模式用于渲染场景。

RGB: 渲染具有正常颜色的对象的场景。

Alpha: 用 alpha 渲染颜色。

过度绘制 (Overdraw): 将对象作为透明的“剪影”渲染。透明颜色累加，这样很容易就可以找出一个对象绘制在另一个对象上的位置。

Mipmap: 使用颜色代码显示理想的纹理尺寸: 红色表示纹理大于所需尺寸 (在当前距离和分辨率下); 蓝色表示纹理可以更大。当然，理想的纹理尺寸取决于游戏运行时的分辨率及相机可以离特定表面多近。

10.18.3 场景照明、游戏覆盖和试听模式

下拉菜单的右侧有三个按钮，可控制场景表示的其他方面。

第一个

按钮确定是否使用默认光照方案或实际添加到场景中的灯光来照亮视图。默认灯光方案初始时使用，但当添加第一个灯光时会自动更改。第二个按钮控制是否在场景视图中渲染天空盒和 GUI 元素，也显示和隐藏布局网格。第三个按钮开启/关闭场景中的音频源。

10.19 小图标和图标显示控制

小图标和图标有几个显示选项，可以用来在开发过程中减少混乱和改善场景的视觉清晰度。

10.19.1 图标选择器 (Icon Selector)

使用图标选择器 (Icon Selector)，可以轻松为游戏对象 (GameObjects) 和脚本设置自定义图标，这些图标将在场景视图 (Scene View) 和检视器 (Inspector) 中使用。要更改一个游戏对象 (GameObject) 的图标，只需在检视器 (Inspector) 中点击其图标。可用类似方法更改脚本资源的图标。在图标选择器 (Icon Selector) 中，标签图标 (Label Icon) 是一种特殊的图标。这类图标将在场景视图 (Scene View) 中作为文本标签显示，使用游戏对象 (GameObject) 的名称。内置组件 (Components) 的图标不能更改。

注意：一个资源的图标更改后，该资源将标记为“已修改”，因此由版本控制系统 (Revision Control Systems) 取回。

下面的图演示如何为游戏对象 (GameObject) 选择一个图标：

下面的图演示如何为脚本选择一个图标：

10.19.2 显示和隐藏图标与小图标

单独组件的小图示的可见性取决于组件在检视器中是否被展开或折叠（即折叠的组件不可见）。然而，您可以使用小图示 (Gizmos) 下拉菜单同时展开或折叠给定类型的每个组件。当场景中有大量小图示和图标时，这是减少视觉混乱的有效方法。

要显示当前小图示和图标的状态，单击场景 (Scene) 或游戏视图 (Game View) 控制条中的小图示 (Gizmos)。这里的切换用来设置哪种图标和小图示可见。

注意，脚本 (Scripts) 部分显示的脚本为拥有自定义图标或让 `OnDrawGizmos ()` 或 `OnDrawGizmosSelected ()` 函数执行的脚本。

小图示 (Gizmos) 下拉菜单，显示图标和小图示的可见状态

图标缩放 (Icon Scaling) 滑动条可以用来调节图标在场景中显示的大小。滑动条在最右端时，图标常常按实际尺寸画出。否则，图标将会根据其距场景视图相机的距离来缩放（尽管显示尺寸设有上限，以避免屏幕混乱）。

10.20 iOS

- <http://www.jianshu.com/p/8472ba0f2bb6>
 - https://github.com/qinjx/30min_guides/blob/master/ios.md iOS 开发 60 分钟入门
 - http://blog.startry.com/2015/07/24/iOS_EnvWithXcconfig/ iOS 开发必备 - 环境变量配置 (Debug & Release)
 - <https://www.zhihu.com/question/22000647> 零基础如何学习 iOS 开发?
 - <https://github.com/Aufree/trip-to-iOS> iOS 学习资料整理
 -
 -
 -
 -
 -
 -

10.21 Unity 动画

10.21.1 shader

•
•
•
•
•

11 Unity C# Functions code practice

```
using UnityEngine;
using System.Collections;

public class ActiveObjects : MonoBehaviour {

    // bgn Physics

    // rigidbody.AddForce(Vector3 direction & magnitude, [optional]Mode of force being used);
    // ForceMode default to ForceMode.Force
    void OnMouseDown() {
        // AddForce() can be used in FixedUpdate() to apply a constant force
        // 4 ForceModes: Acceleration, Force, Impulse, VelocityChange
        // Acceleration & VelocityChange are NOT affected by mass
        // Force & Impulse are affected by mass
        rigidbody.AddForce(-transform.forward * 1000, ForceMode.Force);
        rigidbody.useGravity = true;
    }

    // rigidbody.AddTorque(Vector3 torque, ForceMode mode);
    // rigidbody.AddTorque(Vector3 as axis to apply torque around, [optional]Type of Force to a
    // 4 ForceModes: Acceleration, Force, Impulse, VelocityChange
    // Acceleration & VelocityChange are NOT affected by mass
    // Force & Impulse are affected by mass
    // AddTorque() is significantly affected by Angular drag,
    // if Angular drag is increased, it will make difficult for gameObject to torque
    public float amount = 50f;
    void FixedUpdate() {
        float h = Input.GetAxis("Horizontal") * amount * Time.deltaTime;
        float v = Input.GetAxis("Vertical") * amount * Time.deltaTime;
        rigidbody.AddTorque(transform.up * h, ForceMode.);
        rigidbody.AddTorque(transform.right * v);
    }

    // bgn Scripts
    public GameObject myObject;
    //int [] myInt = new int[5];
    int [] myInt = new int[] {1, 2, 3, 4, 5}; // default is private for C3, public for javascript
    public GameObject players;
    void Start() {
        Debug.Log("Active Self: " + myObject.activeSelf);
        Debug.Log("myObject.activeInHierarchy: " + myObject.activeInHierarchy);
        gameObject.SetActive(false);

        myInt[0] = 12;
    }
}
```

```

players = GameObject.FindGameObjectsWithTag("Player");
for (int i = 0; i < players.Length; i++) {
    Debug.Log("Player number " + i + " is named " + players[i].name);
}
}

public float speed = 8f;
public float countdown = 3f;
void Update() {
    countdown -= Time.deltaTime;
    if (countdown <= 0.0f)
        light.enabled = true;
    if (Input.GetKey(KeyCode.RightArrow))
        transform.position += new Vector3(speed * Time.deltaTime, 0f, 0f);
}

public GameObject other;
void Update() {
    if (Input.GetKey(KeyCode.Space)) {
        // Destroy(gameObject, delayTime);
        // remove entire gameObject
        Destroy(other, 3); // 3 second delayTime
        // remove components
        Destroy(gameObject.GetComponent<MeshRenderer>());
    }
}
// Activate Deactivate
void Update() {
    if (Input.GetKey(KeyCode.Space))
        myLight.enabled = !myLight.enabled;
}

public GameObject otherGameObject;
private AnotherScript anotherScript;
private YetAnotherScript yetAnotherScript;
private BoxCollider boxCol;
void Awake() {
    anotherScript = GetComponent<AnotherScript>();
    yetAnotherScript = otherGameObject.GetComponent<YetAnotherScript>();
    boxCol = otherGameObject.GetComponent<BoxCollider>();
}
void Start() {
    boxCol.size = new Vector3(3, 3, 3);
}

private Vector3 newPosition;
private float newIntensity;
public float smooth = 2;
void Awake() {
    newIntensity = light.intensity;
}
void Update() {
    PositionChanging();
    IntensityChanging();
}
void PositionChanging() {
    Vector3 positionA = new Vector3(-5, 3, 0);
}

```

```

Vector3 positionB = new Vector3(5, 3, 0);
if (Input.GetKeyDown(KeyCode.Q))
    newPosition = positionA;
if (Input.GetKeyDown(KeyCode.E))
    newPosition = positionB;
// Vector3.Lerp(from Vector3, to Vector3, time float)
transform.position = Vector3.Lerp(transform.position, newPosition, Time.deltaTime * smooth);
}

void IntensityChanging() {
    float intensityA = 0.5f;
    float intensityB = 5f;
    if (Input.GetKeyDown(KeyCode.A))
        newIntensity = intensityA;
    if (Input.GetKeyDown(KeyCode.D))
        newIntensity = intensityB;
    // Mathf.Lerp(from float, to float, time float)
    light.intensity = Mathf.Lerp(light.intensity, newIntensity, Time.deltaTime * smooth);
}

// Similarly, Color.Lerp(from Color, to Color, time float)
light.color = Color.Lerp(light.color, newColor, Time.deltaTime * smooth);
}

void Update(){
    if (Input.GetKeyDown(KeyCode.R))
        gameObject.renderer.material.color = Color.red;
}

public float speed = 10f;
void Update(){
    // transform.Rotate(Axis around which to rotate, amount to rotate by)
    transform.Rotate(Vector3.up, speed * Time.deltaTime); // (0, 1, 0)
    if (Input.GetKey(KeyCode.UpArrow))
        transform.Translate(Vector3.forward * speed * Time.deltaTime); // (0, 1, 0)
    if (Input.GetKey(KeyCode.DownArrow))
        transform.Translate(-Vector3.forward * speed * Time.deltaTime); // (0, 1, 0)
}

// VectorA: (x, y, z)
// VectorB: (x, y, z)
// Vector3.Dot(VectorA, VectorB) // Unity function
// (Ax * Bx) + (Ay * By) + (Az * Bz) = Dot Product

// Cross product, L 于原来的两个向量
// Vector3.Cross(VectorA, VectorB) // Unity function
}

```

12 [Unity] 面试题整理 100 题

- <http://gad.qq.com/article/detail/18272>
- <http://gad.qq.com/article/detail/20995> 100 题

12.1 相机

12.1.1 移动相机动作在哪个函数里，为什么在这个函数里？

- LateUpdate，是在所有的 Update 结束后才调用，比较适合用于命令脚本的执行。官网上例子是摄像机的跟随，都是所有的 Update 操作完才进行摄像机的跟进，不然就有可能出现摄像机已经推进了，但是视角里还未有角色的空帧出现。

12.1.2 在场景中放置多个 Camera 并同时处于活动状态会发生什么?

- 游戏界面可以看到很多摄像机的混合。

12.1.3 Unity 中, 照相机的 Clipping Planes 的作用是什么? 调整 Near、Fare 两个值时, 应该注意什么?

- 剪裁平面。从相机到开始渲染和停止渲染之间的距离。

12.1.4 将 Camera 组件的 ClearFlags 选项选成 Depth only 是什么意思? 有何用处?

- 仅深度, 该模式用于对象不被裁剪。

12.1.5 Camera.depth

- 还有一个很常用的调整渲染顺序的选项: 摄像机深度。摄像机深度是优先级最高的选项, 深度值越大, 物体层级越高。常用的比如 NGUI 就是用一个单独的相机 (深度值比渲染场景的相机大) 来渲染 UI, 这样就保证 UI 在所有东西上方。

12.2 光源

12.2.1 : Unity 提供了几种光源, 分别是什么

- 四种。
 - 平行光: Directional Light
 - 点光源: Point Light
 - 聚光灯: Spot Light
 - 区域光源: Area Light

12.2.2 写光照计算中的 diffuse 的计算公式

- 实际光照强度 $I = \text{环境光} (I_{\text{ambient}}) + \text{漫反射光} (I_{\text{diffuse}}) + \text{镜面高光} (I_{\text{specular}});$
- 环境光: $I_{\text{ambient}} = A_{\text{intensity}} * \text{Acolor};$ ($A_{\text{intensity}}$ 表示环境光强度, Acolor 表示环境光颜色)
- 漫反射光: $I_{\text{diffuse}} = D_{\text{intensity}} * \text{Dcolor} * \mathbf{N} \cdot \mathbf{L};$ ($D_{\text{intensity}}$ 表示漫反射强度, Dcolor 表示漫反射光颜色, \mathbf{N} 为该点的法向量, \mathbf{L} 为光源向量)
- 镜面反射光: $I_{\text{specular}} = S_{\text{intensity}} * \text{Scolor} * (\mathbf{R} \cdot \mathbf{V})^n;$ ($S_{\text{intensity}}$ 表示镜面光照强度, Scolor 表示镜面光颜色, \mathbf{R} 为光的反射向量, \mathbf{V} 为观察者向量, n 称为镜面光指数)

12.2.3 写出光照计算中的 diffuse 的计算公式

- $\text{diffuse} = K_d \times \text{colorLight} \times \max(\mathbf{N} \cdot \mathbf{L}, 0);$
- K_d - 漫反射系数、 colorLight - 光的颜色、 \mathbf{N} - 单位法线向量、 \mathbf{L} - 由点指向光源的单位向量,
- 其中 \mathbf{N} 与 \mathbf{L} 点乘, 如果结果小于等于 0, 则漫反射为 0。

12.2.4 实时点光源的优缺点是什么?

- 可以有 cookies - 带有 alpha 通道的立方图 (Cubemap) 纹理。点光源是最耗费资源的。

12.3 物理碰撞、刚体等

12.3.1 Unity3D 中的碰撞器和触发器的区别?

- 碰撞器是触发器的载体, 而触发器只是碰撞器身上的一个属性。
 - 当 $\text{IsTrigger} = \text{false}$ 时, 碰撞器根据物理引擎引发碰撞, 产生碰撞的效果, 可以调用 `OnCollisionEnter/Stay/Exit` 函数;
 - 当 $\text{IsTrigger} = \text{true}$ 时, 碰撞器被物理引擎所忽略, 没有碰撞效果, 可以调用 `OnTriggerEnter/Stay/Exit` 函数。
- 如果既要检测到物体的接触又不想让碰撞检测影响物体移动或要检测一个物件是否经过空间中的某个区域这时就可以用到触发器

12.3.2 物体发生碰撞的必要条件

- 两个物体都必须带有碰撞器 Collider，其中一个物体还必须带有 Rigidbody 刚体。

12.3.3 CharacterController 和 Rigidbody 的区别？

- Rigidbody 具有完全真实物理的特性，而 CharacterController 可以说是受限的 Rigidbody，具有一定的物理效果但不是完全真实的。

12.3.4 在物体发生碰撞的整个过程中，有几个阶段，分别列出对应的函数三个阶段

- OnCollisionEnter
- OnCollisionStay
- OnCollisionExit

12.3.5 Unity3d 的物理引擎中，有几种施加力的方式，分别描述出来

- rigidbody.AddForce
- rigidbody.AddForceAtPosition

12.3.6 射线检测碰撞物的原理是？

- 射线是 3D 世界中一个点向一个方向发射的一条无终点的线，在发射轨迹中与其他物体发生碰撞时，它将停止发射。

12.4 3D 数学

12.4.1 四元数有什么作用？

- 对旋转角度进行计算时用到四元数

12.4.2 简述四元数 Quaternion 的作用，四元数对欧拉角的优点？

- 四元数用于表示旋转
- 相对欧拉角的优点：
 - 能进行增量旋转
 - 避免万向锁
 - 给定方位的表达方式有两种，互为负（欧拉角有无数种表达方式）

12.4.3 U3D 中用于记录节点空间几何信息的组件名称，及其父类名称

- Transform 父类是 Component

12.4.4 向量的点乘、叉乘以及归一化的意义？

- 1) 点乘描述了两个向量的相似程度，结果越大两向量越相似，还可表示投影
- 2) 叉乘得到的向量垂直于原来的两个向量
- 3) 标准化向量：用在只关心方向，不关心大小的时候

12.4.5 矩阵相乘的意义及注意点

- 用于表示线性变换：旋转、缩放、投影、平移、仿射
- 注意矩阵的蠕变：误差的积累

12.4.6 什么是 DrawCall? DrawCall 高了又什么影响? 如何降低 DrawCall?

- Unity 中, 每次引擎准备数据并通知 GPU 的过程称为一次 Draw Call。DrawCall 越高对显卡的消耗就越大。
降低 DrawCall 的方法:
 - Dynamic Batching
 - Static Batching
- 高级特性 Shader 降级为统一的低级特性的 Shader。

12.4.7 如何在 Unity3D 中查看场景的面试, 顶点数和 Draw Call 数? 如何降低 Draw Call 数?

- 在 Game 视图右上角点击 Stats。降低 Draw Call 的技术是 Draw Call Batching

12.5 GUI

12.5.1 Unity 和 cocos2d 的区别

- Unity3D 支持 C#、javascript 等, cocos2d-x 支持 c++、Html5、Lua 等。
- cocos2d 开源并且免费
- Unity3D 支持 iOS、Android、Flash、Windows、Mac、Wii 等平台的游戏开发, cocos2d-x 支持 iOS、Android、WP 等。

12.5.2 为何大家都在移动设备上寻求 U3D 原生 GUI 的替代方案

- 不美观, OnGUI 很耗费时间, 使用不方便

12.5.3 请简述如何在不同分辨率下保持 UI 的一致性

- NGUI 很好的解决了这一点, 屏幕分辨率的自适应性, 原理就是计算出屏幕的宽高比跟原来的预设的屏幕分辨率求出一个对比值, 然后修改摄像机的 size。

12.5.4 请简述 OnBecameVisible 及 OnBecameInvisible 的发生时机, 以及这一对回调函数的意义?

- 当物体是否可见切换之时。可以用于只需要在物体可见时才进行的计算。

12.5.5 简述 NGUI 中 Grid 和 Table 的作用?

- 对 Grid 和 Table 下的子物体进行排序和定位

12.5.6 请简述 NGUI 中 Panel 和 Anchor 的作用

- 只要提供一个 half-pixel 偏移量, 它可以让一个控件的位置在 Windows 系统上精确的显示出来 (只有这个 Anchor 的子控件会受到影响)
- 如果挂载到一个对象上, 那么他可以将这个对象依附到屏幕的角落或者边缘
- UIPanel 用来收集和管理它下面所有 widget 的组件。通过 widget 的 geometry 创建实际的 draw call。没有 panel 所有东西都不能够被渲染出来, 你可以把 UIPanel 当做 Renderer

12.6 生命周期

12.6.1 Unity3d 脚本从唤醒到销毁有着一套比较完整的生命周期, 请列出系统自带的几个重要的方法。

- Awake —> Start —> Update —> FixedUpdate —> LateUpdate —> OnGUI —> Reset —> OnDisable —> OnDestroy

12.6.2 OnEnable、Awake、Start 运行时的发生顺序? 哪些可能在同一个对象周期中反复的发生?

- Awake -> OnEnable -> Start
- OnEnable 在同一周期中可以反复地发生!

12.6.3 物理更新一般放在哪个系统函数里？

- FixedUpdate，每固定帧绘制时执行一次，和 Update 不同的是 FixedUpdate 是渲染帧执行，如果你的渲染效率低下的时候 FixedUpdate 调用次数就会跟着下降。
 - FixedUpdate 比较适用于物理引擎的计算，因为是跟每帧渲染有关。
 - Update 就比较适合做控制。

12.6.4 GPU 的工作原理

- 简而言之，GPU 的图形（处理）流水线完成如下的工作：(并不一定是按照如下顺序)
 - 顶点处理：这阶段 GPU 读取描述 3D 图形外观的顶点数据并根据顶点数据确定 3D 图形的形状及位置关系，建立起 3D 图形的骨架。在支持 DX8 和 DX9 规格的 GPU 中，这些工作由硬件实现的 Vertex Shader（顶点着色器）完成。
 - 光栅化计算：显示器实际显示的图像是由像素组成的，我们需要将上面生成的图形上的点和线通过一定的算法转换到相应的像素点。把一个矢量图形转换为一系列像素点的过程就称为光栅化。例如，一条数学表示的斜线段，最终被转化成阶梯状的连续像素点。
 - 纹理贴图：顶点单元生成的多边形只构成了 3D 物体的轮廓，而纹理映射（texture mapping）工作完成对多变形表面的帖图，通俗的说，就是将多边形的表面贴上相应的图片，从而生成“真实”的图形。TMU（Texture mapping unit）即是用来完成此项工作。
 - 像素处理：这阶段（在对每个像素进行光栅化处理期间）GPU 完成对像素的计算和处理，从而确定每个像素的最终属性。在支持 DX8 和 DX9 规格的 GPU 中，这些工作由硬件实现的 Pixel Shader（像素着色器）完成。
 - 最终输出：由 ROP（光栅化引擎）最终完成像素的输出，一帧渲染完毕后，被送到显存帧缓冲区。
- 总结：GPU 的工作通俗的来说就是完成 3D 图形的生成，将图形映射到相应的像素点上，对每个像素进行计算确定最终颜色并完成输出。

12.6.5 什么是渲染管道？

- 是指在显示器上为了显示出图像而经过的一系列必要操作。渲染管道中的很多步骤，都要将几何物体从一个坐标系中变换到另一个坐标系中去。
- 主要步骤有：
 - 本地坐标 -> 视图坐标 -> 背面裁剪 -> 光照 -> 裁剪 -> 投影 -> 视图变换 -> 光栅化。

12.6.6 MeshRender 中 material 和 sharedmaterial 的区别？

- 修改 sharedMaterial 将改变所有物体使用这个材质的外观，并且也改变储存在工程里的材质设置。
- 不推荐修改由 sharedMaterial 返回的材质。如果你想修改渲染器的材质，使用 material 替代。

12.6.7 简述 SkinnedMesh 的实现原理

12.7 渲染

12.7.1 Unity3D Shader 分哪几种，有什么区别？

- 表面着色器的抽象层次比较高，它可以轻松地以简洁方式实现复杂着色。表面着色器可同时在前向渲染及延迟渲染模式下正常工作。
- 顶点片段着色器可以非常灵活地实现需要的效果，但是需要编写更多的代码，并且很难与 Unity 的渲染管线完美集成。
- 固定功能管线着色器可以作为前两种着色器的备用选择，当硬件无法运行那些酷炫 Shader 的时，还可以通过固定功能管线着色器来绘制出一些基本的内容。

12.7.2 alpha blend 工作原理

- Alpha Blend 实现透明效果，不过只能针对某块区域进行 alpha 操作，透明度可设。

12.7.3 两种阴影判断的方法、工作原理。

- 本影和半影：
 - 本影：景物表面上那些没有被光源直接照射的区域（全黑的轮廓分明的区域）。
 - 半影：景物表面上那些被某些特定光源直接照射但并非被所有特定光源直接照射的区域（半明半暗区域）
- 工作原理：从光源处向物体的所有可见面投射光线，将这些面投影到场景中得到投影面，再将这些投影面与场景中的其他平面求交得出阴影多边形，保存这些阴影多边形信息，然后再按视点位置对场景进行相应处理得到所要求的视图（利用空间换时间，每次只需依据视点位置进行一次阴影计算即可，省去了一次消隐过程）

12.7.4 阴影由两部分组成：本影与半影

- a. 本影：景物表面上那些没有被光源直接照射的区域（全黑的轮廓分明的区域）
- b. 半影：景物表面上那些被某些特定光源直接照射但并非被所有特定光源直接照射的区域（半明半暗区域）
- 求阴影区域的方法：做两次消隐过程
 - 一次对每个光源进行消隐，求出对于光源而言不可见的区域 L；
 - 一次对视点的位置进行消隐，求出对于视点而言可见的面 S；

shadow area = L ∩ S

12.7.5 阴影分为两种：自身阴影和投射阴影

- 自身阴影：因物体自身的遮挡而使光线照射不到它上面的某些可见面
 - 工作原理：利用背面剔除的方法求出，即假设视点在点光源的位置。
- 投射阴影：因不透明物体遮挡光线使得场景中位于该物体后面的物体或区域受不到光照照射而形成的阴影
 - 工作原理：从光源处向物体的所有可见面投射光线，将这些面投影到场景中得到投影面，再将这些投影面与场景中的其他平面求交得出阴影多边形，保存这些阴影多边形信息，然后再按视点位置对场景进行相应处理得到所要求的视图（利用空间换时间，每次只需依据视点位置进行一次阴影计算即可，省去了一次消隐过程）若是动态光源此方法就无效了。

12.7.6 Unity 内置了几个 RenderQueue 的字面值：

RenderQueue	值	说明
Background	1000	这个渲染队列最先渲染，一般用于渲染背景
Geometry(默认值)	2000	这个渲染队列是大多数物体的默认队列，用于渲染不透明物体
AlphaTest	2450	使用了 AlphaTest 的物体在这个队列渲染，当所有的不透明物体都渲染完了再渲染这个队列
Transparent	3000	在 Geometry 和 AlphaTest 之后、从后往前渲染，所有的半透明物体都应该在这里渲染
Overlay	4000	在之前的所有渲染队列都渲染完了之后渲染，比如镜头光晕

12.7.7 Unity 的 Shader 中，Blend SrcAlpha 1 - SrcAlpha 这句话是什么意思？

- 作用就是 Alpha 混合。
- 公式：最终颜色 = 源颜色 * 源透明值 + 目标颜色 (1 - 源透明值)

12.7.8 简述水面倒影的渲染原理

- 原理就是对水面的贴图纹理进行扰动，以产生波光粼粼的效果。用 shader 可以通过 GPU 在像素级别作扰动，效果细腻，需要的顶点少，速度快

12.7.9 请问 alpha test 在何时使用？能达到什么效果？

- Alpha Test，中文就是透明度测试。简而言之就是 V&F
- shader 中最后 fragment 函数输出的该点颜色值（即上一讲 frag 的输出 half4）的 alpha 值与固定值进行比较。
- Alpha Test 语句通常于 Pass{} 中的起始位置。
- Alpha Test 产生的效果也很极端，要么完全透明，即看不到，要么完全不透明。

12.7.10 有 A 和 B 两组物体，有什么办法能够保证 A 组物体永远比 B 组物体先渲染？

- 把 A 组物体的渲染对列大于 B 物体的渲染队列

12.7.11 Vertex Shader 是什么，怎么计算？

- 顶点着色器是一段执行在 GPU 上的程序，用来取代 fixed pipeline 中的 transformation 和 lighting，Vertex Shader 主要操作顶点。
- Vertex Shader 对输入顶点完成了从 local space 到 homogeneous space (齐次空间) 的变换过程，homogeneous space 即 projection space 的下一个 space。在这其间共有 world transformation, view transformation 和 projection transformation 及 lighting 几个过程。

12.7.12 MipMap 是什么，作用？

- MipMapping：在三维计算机图形的贴图渲染中有常用的技术，为加快渲染进度和减少图像锯齿，贴图被处理成由一系列被预先计算和优化过的图片组成的文件，这样的贴图被称为 MipMap。

12.7.13 半透明物体的渲染：

- Unity 性能详解：渲染模块性能 <http://gad.qq.com/article/detail/34971>
- 整个半透明的渲染，包括树、草、UI；
- 如果使用的是 NGUI，看下 Mesh.CreateVBO 是不是很高，因为有时候会达到 3%、5% 甚至百分之十几；如果比较高，说明 NGUI 出现问题，因为这块都是有 UIPanel.LateUpdate 引起的；说明 NGUI 在不停地重建 UI 的 Mesh；
- Unity4.X 会有 ParticleSystem.RenderSingle，是说粒子系统在 CPU 端的开销；
- BatchRenderer.Flush：对应 API 的 Flush，对模型进行渲染；(Unity5.3 之后版本才有)
- MeshRenderer.Render：与半透明网格的渲染相关（花花草草，特效及其面片、UI）；
- Mesh.CreateVBO：如果数值太高，说明 UI 存在很大的问题；
- 在使用 NGUI 时，UI 网格发生重建，重建时创建一些新的网格，对应在渲染里面，会引起 UIPanel.LateUpdate 的调用；
- ParticleSystem.ScheduleGeometryJobs：对粒子系统的计算与处理；
- ParticleSystem.SubmitVBO：粒子系统渲染的 Drawcall；

12.7.14 不透明物体的渲染：

- MeshRenderer.Render：场景模型的渲染；
- BatchRenderer.Add (MeshSkinned.Render)：蒙皮网格的渲染；

12.7.15 裁剪区域的渲染:

- Culling: 根据视锥体进行裁剪, 一般耗时在 15%~20% 之间, MMO 项目场景中 GameObject 数量越多, Culling 耗时越多, 一般优化空间不大;
- Uniy4.X 为 Culling, Unity5.X 是 Scene Culling;
- 根据视锥体的锥状体和里面所有物体的包围盒进行求交, 若果有交点, 就会进入到渲染的 PipeLine 里面去;
- XXX.OnPreCull 跟所做的图像的后处理相关;

12.7.16 蒙皮网格渲染

- Unity 性能详解: 渲染模块性能
- 找出骨骼数 (BoneWeights) 大于 0 的, 一般降低面片数就可以降低 Update 以及渲染方面的开销;

12.7.17 蒙皮网格渲染如何解决:

- 1: 减少 Skinned Mesh 面片数;
- 2: 某些特定场合可以降低 Draw Call (使用 MeshBaker 插件);
 - (将蒙皮网格合成大的网格, 比如有 15 个同种的怪物或士兵, 将 15 个怪物使用同一个骨架);
 - 但是这种方式在动态增减 Skinned Mesh 时, 会造成较大的开销; 且只能带来 Drawcall 的下降, 对于 Animator、MeshSkinnedUpdate 等 CPU 的开销无太大帮助)

12.7.18 粒子系统渲染:

- Unity 性能详解: 渲染模块性能
- Unity5.3.0 版本之前, 粒子系统的渲染在 CPU 的主线程中都会有一定开销,
- Unity5.3 之后粒子系统被 Unity 挪到了子线程进行渲染, 不再卡主线程;
- 粒子系统使用越多, 覆盖面积越大时, 开销越大;

12.7.19 粒子系统渲染的优化建议:

- 1: 简化粒子系统、减小屏幕的覆盖面积;
- 2: 尝试升级到 Unity5.3 版本之后

12.7.20 图像后处理:

- Camera.ImageEffects: 图像或屏幕的后处理;
 - RenderPipeline.OnRenderImage: 在做图像的后处理;
 - MeshRenderer.RenderStaticBatch 和 MeshRenderer.Render: 对场景中物体的渲染;
 - Shadows.RenderShadowmap: 开启了实时阴影的渲染 (只是画 Shadowmap 的耗时), 其实还会在开启 Shadowmap 时, 光源会作为另外一个 Camera 对场景进行第二次的渲染, 这次渲染的耗时也会放到 Shadows.RenderShadowmap 中, 实际的开销高于这里显示的;
 - 优化方案:
 - 1: 使用 Asset Store 中 Mobile Ready 的插件 (DOF、Blur 等);
 - 2: 在中低端机器上适度使用复杂特效; (简化特别复杂的或直接进行关闭)
 - 3: 在安卓端开启多线程渲染功能 (默认不开启);
- * 但是开启后会带来新的问题, 如下图所示 (对不透明物体的渲染走势波动太大):

12.7.21 开启多线程后，UI 渲染不稳定的原因？

- A: NGUI 和 UGUI 的 UI Mesh 随着动静没有分离或其他因素会产生网格的重建，网格重建后，半透明会非常之高；修复网格重建后，半透明渲染就能降下来；
- 影响多线程渲染的有：
 - 1: 动静没有分离，导致网格重建；
 - 2: Drawcall 太高；

12.7.22 实时阴影：两个地方会增大实时阴影的耗时：

- 1: Shadowmap 的分辨率；2048*2048 还是 1024*1024 等；
 - 分辨率越高生成 Shadow 的效果越好，但带来的性能的开销就越大；
 - Built-in Shadow Map 分辨率比较高时，耗时高；
- 2: Shadow Receiver 网格的数据量：形成 Shadow 时，会有一个接触面，网格的数据量决定了渲染阴影时的开销；
 - Shadow Receiver 接收阴影的网格面片，面片越多，CPU 占用高；

12.8 其它

12.8.1 简述 prefab 的用处

- 在游戏运行时实例化，prefab 相当于一个模板，对你已经有的素材、脚本、参数做一个默认的配置，以便于以后的修改，同时 prefab 打包的内容简化了导出的操作，便于团队的交流。

12.8.2 使用 unity3d 实现 2d 游戏，有几种方式？

- 1. 使用本身的 GUI、UGUI
- 2. 把摄像机的 Projection(投影) 值调为 Orthographic(正交投影)，不考虑 z 轴；
- 3. 使用 2d 插件，如：2DToolKit、NGUI

12.8.3 什么叫做链条关节？

- Hinge Joint，可以模拟两个物体间用一根链条连接在一起的情况，能保持两个物体在一个固定距离内部相互移动而不产生作用力，但是达到固定距离后就会产生拉力。

12.8.4 物体自身旋转使用的函数？

- Transform.Rotate()

12.8.5 Unity3d 提供了一个用于保存和读取数据的类 (**PlayerPrefs**)，请列出保存和读取整形数据的函数

- PlayerPrefs.SetInt()
- PlayerPrefs.GetInt()

12.8.6 请描述为什么 Unity3d 中会发生在组件上出现数据丢失的情况

- 一般是组件上绑定的物体对象被删除了

12.8.7 请描述游戏动画有哪几种，以及其原理？

- 主要有关节动画、骨骼动画、单一网格模型动画（关键帧动画）。
 - 关节动画：把角色分成若干独立部分，一个部分对应一个网格模型，部分的动画连接成一个整体的动画，角色比较灵活，Quake2 中使用这种动画；
 - 骨骼动画，广泛应用的动画方式，集成了以上两个方式的优点，骨骼按角色特点组成一定的层次结构，有关节相连，可做相对运动，皮肤作为单一网格蒙在骨骼之外，决定角色的外观；
 - 单一网格模型动画由一个完整的网格模型构成，在动画序列的关键帧里记录各个顶点的原位置及其改变量，然后插值运算实现动画效果，角色动画较真实。

12.8.8 如何安全的在不同工程间安全地迁移 asset 数据？三种方法

- 1. 将 Assets 和 Library 一起迁移
- 2. 导出包 package
- 3. 用 unity 自带的 assets Server 功能

12.8.9 什么叫动态合批？跟静态合批有什么区别？

- 如果动态物体共用着相同的材质，那么 Unity 会自动对这些物体进行批处理。动态批处理操作是自动完成的，并不需要你进行额外的操作。
- 区别：动态批处理一切都是自动的，不需要做任何操作，而且物体是可以移动的，但是限制很多。静态批处理：自由度很高，限制很少，缺点可能会占用更多的内存，而且经过静态批处理后的所有物体都不可以再移动了。

12.8.10 什么是 LightMap？

- LightMap：就是指在三维软件里实现打好光，然后渲染把场景各表面的光照输出到贴图上，最后又通过引擎贴到场景上，这样就使物体有了光照的感觉。

12.8.11 当一个细小的高速物体撞向另一个较大的物体时，会出现什么情况？如何避免？

- 穿透（碰撞检测失败）

12.8.12 请写出求斐波那契数列任意一位的值得算法

```
static int Fn(int n) {  
    if (n <= 0)  
        throw new ArgumentOutOfRangeException();  
    if (n == 1 || n == 2)  
        return 1;  
    return checked(Fn(n - 1) + Fn(n - 2)); // when n>46 memory will overflow  
}
```

12.8.13 什么是里氏代换元则？

- 里氏替换原则 (Liskov Substitution Principle LSP) 面向对象设计的基本原则之一。里氏替换原则中说，任何基类可以出现的地方，子类一定可以出现，作用方便扩展功能能

12.8.14 Mock 和 Stub 有何区别？

- Mock 与 Stub 的区别：
 - Mock：关注行为验证。细粒度的测试，即代码的逻辑，多数情况下用于单元测试。
 - Stub：关注状态验证。粗粒度的测试，在某个依赖系统不存在或者还没实现或者难以测试的情况下使用，例如访问文件系统，数据库连接，远程协议等。

12.8.15 如何让已经存在的 GameObject 在 LoadLevel 后不被卸载掉？

```
void Awake() {  
    DontDestroyOnLoad(transform.gameObject);  
}
```

12.8.16 将图片的 TextureType 选项分别选为 Texture 和 Sprite 有什么区别

- Sprite 作为 UI 精灵使用
- Texture 作用模型贴图使用。

12.8.17 问一个 Terrain，分别贴 3 张，4 张，5 张地表贴图，渲染速度有什么区别？为什么？

- 没有区别，因为不管几张贴图只渲染一次。

12.8.18 使用动态字体时是否会生成字符纹理

- 在使用动态字体时，Unity 不会先生成字符纹理

12.8.19 为什么 **dynamic font** 在 **unicode** 环境下优于 **static font**

- Unicode 是国际组织制定的可以容纳世界上所有文字和符号的字符编码方案。
- 使用动态字体时，Unity 将不会预先生成一个与所有字体的字符纹理。当需要支持亚洲语言或者较大的字体的时候，若使用正常纹理，则字体的纹理将非常大。

12.8.20 动态加载资源的方式？(有时候也问区别，具体请百度)

- 1. 通过 Resources 模块，调用它的 load 函数：可以直接 load 并返回某个类型的 Object，前提是把这个资源放在 Resource 命名的文件夹下，Unity 不管有没有场景引用，都会将其全部打入到安装包中。Resources.Load();
- 2. 通过 bundle 的形式：即将资源打成 asset bundle 放在服务器或本地磁盘，然后使用 WWW 模块 get 下来，然后从这个 bundle 中 load 某个 object。AssetBundle
- 3. 通过 AssetDatabase.loadasset：这种方式只在 editor 范围内有效，游戏运行时没有这个函数，它通常是在开发中调试用的【AssetDatabase 资源数据库】
- 区别：Resources 的方式需要把所有资源全部打入安装包，这对游戏的分包发布（微端）和版本升级（patch）是不利的，所以 unity 推荐的方式是不用它，都用 bundle 的方式替代，把资源分成几个小的 bundle，用哪个就 load 哪个，这样还能分包发布和 patch，但是在开发过程中，不可能没更新一个资源就打一次 bundle，所以 editor 环境下可以使用 AssetDatabase 来模拟，这通常需要我们封装一个 dynamic resource 的 loader 模块，在不同的环境下做不同实现。
- 动态资源的存放
 - 有时我需要存放一些自己的文件在磁盘上，例如我想把几个 bundle 放在初始的安装里，unity 有一个 streaming asset 的概念，用于提供存储接口的访问。我们需要在编辑器建立一个 StreamingAssets 名字的文件夹，把需要我们放在客户磁盘上的动态文件放在这个文件夹下面，这样安装后，这些文件会放在用户磁盘的指定位置，这个位置可以通过 Application.streamingAssetsPath 来得到。

12.8.21 动态加载资源的方式？

- 1. Resources.Load();
- 2. AssetBundle
- Unity5.1 版本后可以选择使用 Git:
- <https://github.com/applexiaohao/L0AssetFramework.git>

12.9 与 Mobile 端交互

12.9.1 Unity 和 Android 与 iOS 如何交互？

12.9.2 LOD 是什么，优缺点是什么？

- LOD(Level of detail) 多层次细节，是最常用的游戏优化技术。它按照模型的位置和重要程度决定物体渲染的资源分配，降低非重要物体的面数和细节度，从而获得高效率的渲染运算。

12.9.3 UNITY3d 在移动设备上的一些优化资源的方法

- 1. 使用 assetbundle，实现资源分离和共享，将内存控制到 200m 之内，同时也可以实现资源的在线更新
- 2. 顶点数对渲染无论是 cpu 还是 gpu 都是压力最大的贡献者，降低顶点数到 8 万以下，fps 稳定到了 30 帧左右
- 3. 只使用一盏动态光，不是用阴影，不使用光照探头
- 4. 剪裁粒子系统 (粒子系统是 cpu 上的大头)

- 5. 合并同时出现的粒子系统
- 6. 自己实现轻量级的粒子系统
- 7. 把不需要跟骨骼动画和动作过渡的地方全部使用 animation，控制骨骼数量在 30 根以下 (animator 也是一个效率奇差的地方)
- 8.animator 出视野不更新
- 9. 删除无意义的 animator
- 10.animator 的初始化很耗时（粒子上能不能尽量不用 animator）
- 11. 除主角外都不要跟骨骼运动 apply root motion
- 12. 绝对禁止掉那些不带刚体带包围盒的物体（static collider ）运动
- 13 每帧递归的计算 finalalpha 改为只有初始化和变动时计算
- 14 去掉法线计算
- 15 不要每帧计算 viewsize 和 windowsize
- 16 filldrawcall 时构建顶点缓存使用 array.copy
- 17. 代码剪裁：使用 strip level ， 使用.net2.0 subset
- 18. 尽量减少 smooth group
- 19. 给美术定一个严格的经过科学验证的美术标准，并在 U3D 里面配以相应的检查工具
- ? NUGI 的代码效率很差，基本上 runtime 的时候对 cpu 的贡献和 render 不相上下

12.9.4 你用过哪些插件？

12.10 语言基础、OOP

12.10.1 请简述 ArrayList 和 List 之间的主要区别。

- ArrayList 存在不安全类型。ArrayList 会把所有插入其中的数据都当做 Object 来处理。装箱拆箱的操作（费时），
- List 是接口，ArrayList 是一个实现了该接口的类，可以被实例化。

12.10.2 请简述 ArrayList 和 List<Int> 的主要区别

- ArrayList 存在不安全类型
 - ArrayList 会把所有插入其中的数据都当做 Object 来处理
 - 装箱拆箱的操作
- List 是接口，ArrayList 是一个实现了该接口的类，可以被实例化

12.10.3 请简述 sealed 关键字用在类声明时与函数声明时的作用。

- 类声明时可防止其他类继承此类，在方法中声明则可防止派生类重写此方法。

12.10.4 请简述 private, public, protected, internal 的区别。

- public：对任何类和成员都公开，无限制访问
- private：仅对该类公开
- protected：对该类及其派生类公开
- internal：只能在包含该类的程序集中访问该类
- protected internal：protected + internal

12.10.5 反射的实现原理?

- 审查元数据并收集关于它的类型信息的能力。
- 实现步骤:
 - 导入 using System.Reflection;
 - Assembly.Load("程序集"); // 加载程序集, 返回类型是一个 Assembly

```
1 foreach (Type type in assembly.GetType()) {  
2     string t = type.Name;  
3 } // 3.. 得到程序集中所有类的名称  
4 Type type = assembly.GetType(" 程序集. 类名 "); // 4. 获取当前类的类型  
5 Activator.CreateInstance(type); // 5. 创建此类型实例  
6 MethodInfo mInfo = type.GetMethod(" 方法名 "); // 6. 获取当前方法  
7 mInfo.Invoke(null, 方法参数); // 7.
```

12.10.6 简述一下对象池, 你觉得在 FPS 里哪些东西适合使用对象池?

- 对象池是就存放需要被反复调用资源的一个空间, 比如游戏中要常被大量复制的对象, 子弹, 敌人, 以及任何重复出现的对象。

12.10.7 请简述 GC (垃圾回收) 产生的原因, 并描述如何避免?

- GC 回收堆上的内存
- 避免:
 - 1) 减少 new 产生对象的次数
 - 2) 使用公用的对象 (静态成员)
 - 3) 将 String 换为 StringBuilder

12.10.8 如何优化内存?

- 有很多种方式, 例如
 - 1. 压缩自带类库;
 - 2. 将暂时不用的以后还需要使用的物体隐藏起来而不是直接 Destroy 掉;
 - 3. 释放 AssetBundle 占用的资源;
 - 4. 降低模型的片面数, 降低模型的骨骼数量, 降低贴图的大小;
 - 5. 使用光照贴图, 使用多层次细节 (LOD), 使用着色器 (Shader), 使用预设 (Prefab)。

12.10.9 如何销毁一个 UnityEngine.Object 及其子类?

- 使用 Destroy() 方法;

12.10.10 能用 foreach 遍历访问的对象需要实现 __

- IEnumerable; GetEnumerator

12.10.11 简述 StringBuilder 和 String 的区别?

- String 是字符串常量。
- StringBuffer 是字符串变量, 线程安全。
- StringBuilder 是字符串变量, 线程不安全。
- String 类型是个不可变的对象, 当每次对 String 进行改变时都需要生成一个新的 String 对象, 然后将指针指向一个新的对象, 如果在一个循环里面, 不断的改变一个对象, 就要不断的生成新的对象, 所以效率很低, 建议在不断更改 String 对象的地方不要使用 String 类型。
- StringBuilder 对象在做字符串连接操作时是在原来的字符串上进行修改, 改善了性能。这一点我们平时使用中也许都知道, 连接操作频繁的时候, 使用 StringBuilder 对象。

12.10.12 已知 strcpy 函数的原型是：

```
char * strcpy(char * strDest, const char * strSrc);
```

- 1. 不调用库函数，实现 strcpy 函数。
- 2. 解释为什么要返回 char *

```
char * strcpy(char *strDest, const char *strSrc) {  
    if ((strDest == NULL) || (strSrc == NULL)) // [1]  
        throw "Invalid argument(s)"; // [2]  
    char *strDestCopy = strDest; // [3]  
    while ((*strDest++ = *strSrc++) != '\0'); // [4]  
    return strDestCopy;  
}
```

- 错误的做法：

- //不检查指针的有效性，说明答题者不注重代码的健壮性。
- //检查指针的有效性时使用 ((!strDest)||(!strSrc)) 或 (!strDest&&!strSrc)，说明答题者对 C 语言中类型的隐式转换没有深刻认识。在本例中 char * 转换为 bool 即是类型隐式转换，这种功能虽然灵活，但更多的是导致出错概率增大和维护成本升高。所以 C++ 专门增加了 bool、true、false 三个关键字以提供更安全的条件表达式。
- //检查指针的有效性时使用 ((strDest==0)|| (strSrc==0))，说明答题者不知道使用常量的好处。直接使用字面常量（如本例中的 0）会减少程序的可维护性。0 虽然简单，但程序中可能出现很多处对指针的检查，万一出现笔误，编译器不能发现，生成的程序内含逻辑错误，很难排除。而使用 NULL 代替 0，如果出现拼写错误，编译器就会检查出来。
- //return new string("Invalid argument(s)");，说明答题者根本不知道返回值的用途，并且他对内存泄漏也没有警惕心。从函数中返回函数体内分配的内存是十分危险的做法，他把释放内存的义务抛给不知情的调用者，绝大多数情况下，调用者不会释放内存，这导致内存泄漏。
- //return 0;，说明答题者没有掌握异常机制。调用者有可能忘记检查返回值，调用者还可能无法检查返回值（见后面的链式表达式）。妄想让返回值肩负返回正确值和异常值的双重功能，其结果往往是两种功能都失效。应该以抛出异常来代替返回值，这样可以减轻调用者的负担、使错误不会被忽略、增强程序的可维护性。
- //忘记保存原始的 strDest 值，说明答题者逻辑思维不严密。
- //循环写成 while (*strDest++=*strSrc++);，同²(B)。
- //循环写成 while (*strSrc!='\0') *strDest++=*strSrc++;，说明答题者对边界条件的检查不力。循环体结束后，strDest 字符串的末尾没有正确地加上'\0'。
- 返回 strDest 的原始值使函数能够支持链式表达式，增加了函数的“附加值”。同样功能的函数，如果能合理地提高的可用性，自然就更加理想。
- 链式表达式的形式如：

```
1   int iLength=strlen(strcpy(strA,strB));  
2   // or  
3   char * strA=strcpy(new char[10],strB);
```

- 返回 strSrc 的原始值是错误的。其一，源字符串肯定是已知的，返回它没有意义。其二，不能支持形如第二例的表达式。其三，为了保护源字符串，形参用 const 限定 strSrc 所指的内容，把 const char * 作为 char * 返回，类型不符，编译报错。

12.10.13 堆和栈的区别？

- 栈通常保存着我们代码执行的步骤，如在代码段 1 中 AddFive() 方法，int pValue 变量，int result 变量等等。
- 而堆上存放的则多是对象，数据等。（译者注：忽略编译器优化）
- 我们可以把栈想象成一个接着一个叠放在一起的盒子。当我们使用的时候，每次从最顶部取走一个盒子。栈也是如此，当一个方法（或类型）被调用完成的时候，就从栈顶取走（called a Frame，译注：调用帧），接着下一个。堆则不然，像是一个仓库，储存着我们使用的各种对象等信息，跟栈不同的是他们被调用完毕不会立即被清理掉。

²DEFINITION NOT FOUND.

12.10.14 Heap 与 Stack 有何区别?

- 1.heap 是堆， stack 是栈。
- 2.stack 的空间由操作系统自动分配和释放， heap 的空间是手动申请和释放的， heap 常用 new 关键字来分配。
- 3.stack 空间有限， heap 的空间是很大的自由区。

12.10.15 值类型和引用类型有何区别?

- 1. 值类型的数据存储在内存的栈中；引用类型的数据存储在内存的堆中，而内存单元中只存放堆中对象的地址。
- 2. 值类型存取速度快，引用类型存取速度慢。
- 3. 值类型表示实际数据，引用类型表示指向存储在内存堆中的数据的指针或引用
- 4. 引用类型继承自 System.Object，值类型继承自 System.ValueType，同时，值类型也隐式继承自 System.Object。
- 5. 栈的内存分配是自动释放；而堆在.NET 中会有 GC 来释放。
- 6. 值类型的变量直接存放实际的数据，而引用类型的变量存放的则是数据的地址，即对象的引用。
- 7. 值类型变量直接把变量的值保存在堆栈中，引用类型的变量把实际数据的地址保存在堆栈中。

12.10.16 C# 中所有引用类型的基类是什么

- 引用类型的基类是 System.Object 值类型的基类是 System.ValueType
- 值类型也隐式继承自 System.Object

12.10.17 结构体和类有何区别?

- 结构体是一种值类型，而类是引用类型。（值类型、引用类型是根据数据存储的角度来分的）
- 就是值类型用于存储数据的值，引用类型用于存储对实际数据的引用。
- 那么结构体就是当成值来使用的，类则通过引用来对实际数据操作。

12.10.18 C# 中四种访问修饰符是哪些？各有什么区别?

- 1. 属性修饰符
- 2. 存取修饰符
- 3. 类修饰符
- 4. 成员修饰符。
- 属性修饰符：
 - Serializable：按值将对象封送到远程服务器。
 - STAThread：是单线程套间的意思，是一种线程模型。
 - MATAThread：是多线程套间的意思，也是一种线程模型。
- 存取修饰符：
 - public：存取不受限制。
 - private：只有包含该成员的类可以存取。
 - internal：只有当前命名空间可以存取。
 - protected：只有包含该成员的类以及派生类可以存取。
- 类修饰符：

- abstract: 抽象类。指示一个类只能作为其它类的基类。
- sealed: 密封类。指示一个类不能被继承。理所当然，密封类不能同时又是抽象类，因为抽象总是希望被继承的。
- 成员修饰符：
 - abstract: 指示该方法或属性没有实现。
 - sealed: 密封方法。可以防止在派生类中对该方法的 override (重载)。不是类的每个成员方法都可以作为密封方法，必须对基类的虚方法进行重载，提供具体的实现方法。所以，在方法的声明中，sealed 修饰符总是和 override 修饰符同时使用。
 - delegate: 委托。用来定义一个函数指针。C# 中的事件驱动是基于 delegate + event 的。
 - const: 指定该成员的值只读不允许修改。
 - event: 声明一个事件。
 - extern: 指示方法在外部实现。
 - override: 重写。对由基类继承成员的新实现。
 - readonly: 指示一个域只能在声明时以及相同类型的内部被赋值。
 - static: 指示一个成员属于类型本身，而不是属于特定的对象。即在定义后可不经实例化，就可使用。
 - virtual: 指示一个方法或存取器的实现可以在继承类中被覆盖。
 - new: 在派生类中隐藏指定的基类成员，从而实现重写的功能。若要隐藏继承类的成员，请使用相同名称在派生类中声明该成员，并用 new 修饰符修饰它。

12.10.19 请说出 4 种面向对象的设计原则，并分别简述它们的含义。

- 1) 单一职责原则 (The Single Responsibility Principle, 简称 SRP): 一个类，最好只做一件事，只有一个引起它的变化。
- 2) 开放 - 封闭原则 (The Open - Close Principle, 简称 OCP): 对于扩展是开放的，对于更改是封闭的。
- 3) Liskov 替换原则 (The Liskov Substitution Principle, 简称 LSP): 子类必须能够替换其基类。
- 4) 依赖倒置原则 (The Dependency Inversion Principle, 简称 DIP): 依赖于抽象。
- 5) 接口隔离原则 (The Interface Segregation Principle, 简称 ISP): 使用多个小的专门的接口，而不要使用一个大的总接口。

12.10.20 请描述 Interface 与抽象类之间的不同

- 抽象类表示该类中可能已经有一些方法的具体定义，但接口就是公共 (刚刚?) 只能定义各个方法的界面，不能具体的实现代码在成员方法中。
- 类是子类用来继承的，当父类已经有实际功能的方法时该方法在子类中可以不必实现，直接引用父类的方法，子类也可以重写该父类的方法。
- 实现接口的时候必须要实现接口中所有的方法，不能遗漏任何一个。

12.10.21 下列代码在运行中会产生几个临时对象？

```
string a = new string("abc");
a = (a.ToUpper() + "123").Substring(0, 2);
```

- 其实在 C# 中第一行是会出错的 (Java 中倒是可行)。应该这样初始化: string b = new string(new char[]{'a','b','c'});

12.10.22 下列代码在运行中会发生什么问题？如何避免？

```
List<int> ls = new List<int>(new int[] { 1, 2, 3, 4, 5 });
foreach (int item in ls)
{
    Console.WriteLine(item * item);
    ls.Remove(item);
}
```

- 会产生运行时错误，因为 foreach 是只读的。不能一边遍历一边修改。

12.10.23 在编辑场景时将 GameObject 设置为 Static 有何作用？

- 设置游戏对象为 Static 将会剔除（或禁用）网格对象当这些部分被静态物体挡住而不可见时。因此，在你的场景中的所有不会动的物体都应该标记为 Static。

12.10.24 Unity3D 是否支持写成多线程程序？如果支持的话需要注意什么？

- 参考：Unity3D 教程：Unity3D 中的多线程及使用多线程 <http://www.manew.com/3821.html>
- 仅能从主线程中访问 Unity3D 的组件，对象和 Unity3D 系统调用
- 支持：如果同时你要处理很多事情或者与 Unity 的对象互动小可以用 thread，否则使用 coroutine。
- 注意：C# 中有 lock 这个关键字，以确保只有一个线程可以在特定时间内访问特定的对象

12.10.25 什么是协同程序？

- 在主线程运行时同时开启另一段逻辑处理，来协助当前程序的执行。换句话说，开启协程就是开启一个可以与程序并行的逻辑。可以用来控制运动、序列以及对象的行为。

12.10.26 Unity3D 的协程和 C# 线程之间的区别是什么？

- 多线程程序同时运行多个线程，而在任一指定时刻只有一个协程在运行，并且这个正在运行的协同程序只在必要时才被挂起。
- 除主线程之外的线程无法访问 Unity3D 的对象、组件、方法。
- Unity3d 没有多线程的概念，不过 unity 也给我们提供了 StartCoroutine（协同程序）和 LoadLevelAsync（异步加载关卡）后台加载场景的方法。StartCoroutine 为什么叫协同程序呢，所谓协同，就是当你在 StartCoroutine 的函数体里处理一段代码时，利用 yield 语句等待执行结果，这期间不影响主程序的继续执行，可以协同工作。

12.10.27 协同程序的执行代码是什么？有何用处，有何缺点？

```
function Start() {
    // - After 0 seconds, prints "Starting 0.0"
    // - After 0 seconds, prints "Before WaitAndPrint Finishes 0.0"
    // - After 2 seconds, prints "WaitAndPrint 2.0"
    // 先打印"Starting 0.0" 和"Before WaitAndPrint Finishes 0.0" 两句,2 秒后打印"WaitAndPrint 2.0"
    print ("Starting " + Time.time );
    // Start function WaitAndPrint as a coroutine. And continue execution while it is running
    // this is the same as WaintAndPrint(2.0) as the compiler does it for you automatically
    // 协同程序 WaitAndPrint 在 Start 函数内执行，可以视同于它与 Start 函数同步执行。
    StartCoroutine(WaitAndPrint(2.0));
    print ("Before WaitAndPrint Finishes " + Time.time );
}

function WaitAndPrint (waitForTime : float) {
    // suspend execution for waitForTime seconds
    // 暂停执行 waitForTime 秒
    yield WaitForSeconds (waitForTime);
    print ("WaitAndPrint "+ Time.time );
}
```

- 作用：一个协同程序在执行过程中，可以在任意位置使用 yield 语句。yield 的返回值控制何时恢复协同程序向下执行。协同程序在对象自有帧执行过程中堪称优秀。协同程序在性能上没有更多的开销。
- 缺点：协同程序并非真线程，可能会发生堵塞。

12.10.28 C# 中的排序方式有哪些？

- 选择排序
- 冒泡排序
- 快速排序
- 插入排序
- 希尔排序
- 归并排序

12.10.29 ref 参数和 out 参数是什么？有什么区别？

- ref 和 out 参数的效果一样，都是通过关键字找到定义在主函数里面的变量的内存地址，并通过方法体内的语法改变它的大小。
- 不同点就是输出参数必须对参数进行初始化。
- ref 参数是引用，out 参数为输出参数。

12.10.30 C# 的委托是什么？有何用处？

- 委托类似于一种安全的指针引用，在使用它时是当做类来看待而不是一个方法，相当于对一组方法的列表的引用。
- 用处：使用委托使程序员可以将方法引用封装在委托对象内。然后可以将该委托对象传递给可调用所引用方法的代码，而不必在编译时知道将调用哪个方法。与 C 或 C++ 中的函数指针不同，委托是面向对象，而且是类型安全的。

12.10.31 概述序列化：

- 序列化简单理解成把对象转换为容易传输的格式的过程。比如，可以序列化一个对象，然后使用 HTTP 通过 Internet 在客户端和服务器端之间传输该对象

12.10.32 概述 c# 中代理和事件？

- 代理就是用来定义指向方法的引用。
- C# 事件本质就是对消息的封装，用作对象之间的通信；发送方叫事件发送器，接收方叫事件接收器；

12.10.33 TCP/IP 协议栈各个层次及分别的功能

- 网络接口层：这是协议栈的最低层，对应 OSI 的物理层和数据链路层，主要完成数据帧的实际发送和接收。
- 网络层：处理分组在网络中的活动，例如路由选择和转发等，这一层主要包括 IP 协议、ARP、ICMP 协议等。
- 传输层：主要功能是提供应用程序之间的通信，这一层主要是 TCP/UDP 协议。
- 应用层：用来处理特定的应用，针对不同的应用提供了不同的协议，例如进行文件传输时用到的 FTP 协议，发送 email 用到的 SMTP 等。

12.10.34 客户端与服务器交互方式有几种？

- socket 通常也称作“套接字”，实现服务器和客户端之间的物理连接，并进行数据传输，主要有 UDP 和 TCP 两个协议。Socket 处于网络协议的传输层。
- http 协议传输的主要有 http 协议和基于 http 协议的 Soap 协议（web service），常见的方式是 http 的 post 和 get 请求，web 服务。

12.10.35 .Net 与 Mono 的关系?

- mono 是.net 的一个开源跨平台工具，就类似 java 虚拟机，java 本身不是跨平台语言，但运行在虚拟机上就能够实现了跨平台。.net 只能在 windows 下运行，mono 可以实现跨平台编译运行，可以运行于 Linux, Unix, Mac OS 等。

12.10.36 C# 和 C++ 的区别?

- 简单的说：C# 与 C++ 比较的话，最重要的特性就是 C# 是一种完全面向对象的语言，而 C++ 不是。
- 另外 C# 是基于 IL 中间语言和.NET Framework CLR 的，在可移植性，可维护性和强壮性都比 C++ 有很大的改进。
- C# 的设计目标是用来开发快速稳定可扩展的应用程序，当然也可以通过 Interop 和 Pinvole 完成一些底层操作

12.10.37 简述 Unity3D 支持的作为脚本的语言的名称

- Unity 的脚本语言基于 Mono 的.NET 平台上运行，可以使用.NET 库，这也为 XML、数据库、正则表达式等问题提供了很好的解决方案。
- Unity 里的脚本都会经过编译，他们的运行速度也很快。这三种语言实际上的功能和运行速度是一样的，区别主要体现在语言特性上。
- JavaScript、C#、Boo

13 第一部分

- http://blog.csdn.net/qq_26270779/article/details/53609069

- 以下哪一个选项不属于 Unity 引擎所支持的视频格式文件 (D)
 - A. 后缀为 mov 的文件
 - B. 后缀为 mpg 的文件
 - C. 后缀为 avi 的文件
 - D. 后缀为 swf 的文件
- Unity 引擎使用的是左手坐标系还是右手坐标系 (A)
 - A. 左手坐标系
 - B. 右手坐标系
 - C. 可以通过 ProjectSetting 切换右手坐标系
 - D. 可以通过 Reference 切换左手坐标系
- 什么是导航网格 (NavMesh) (B)
 - A. 一种用于描述相机轨迹的网格
 - B. 一种用于实现自动寻址的网格
 - C. 一种被优化过的物体网格
 - D. 一种用于物理碰撞的网格
- 下列选项中有关 Animator 的说法错误的是? (D)
 - A.Animator 是 Unity 引擎中内置的组件
 - B. 任何一个具有动画状态机功能的 GameObject 都需要一个 Anim 组件
 - C. 它主要用于角色行为的设置，包括 StateMachine、混合树 BlendTrees 以及同通过脚本控制的事件
 - D.Animator 同 Animation 组件的用法是相同的
- Application.loadLevel 命令为 (A)

- A. 加载关卡
 - B. 异步加载关卡
 - C. 加载动作
 - D. 加载动画
- 下列选项中，关于 Transform 组件的 Scale 参数描述正确的是 (A)
 - A.Transform 组件的 Scale 参数不会影响 ParticleSystem 产生粒子的大小
 - B.Transform 组件的 Scale 参数不会影响 GUITexture 的大小
 - C. 添加 Collider 组件后的 GameoObject，其 Collider 组件的尺寸不受 Transform 组件的 Scale 参数影响
 - D. 添加 Rigidbody 组件后的物体，大小将不再受 Transform 组件中 Scale 参数的影响
- 在 Unity 中的场景中创建 Camera 时，默认情况下除了带有 Transform、Camera、GUILayer、Flare Layer 组件之外，还带有以下哪种组件 (C)
 - A.Mouse Look
 - B.FPS Input Controller
 - C.Audio Listener
 - D.Character Motor
- 如果将一个声音剪辑文件从 Project 视图拖动到 Inspector 视图或者 Scene 视图中的游戏对象上，该游戏对象会自动添加以下哪种组件 (C)
 - A.Audio Listener
 - B.Audio Clip
 - C.Audio Source
 - D.Audio Reverb Zone
- 下列叙述中有关 Prefab 说法错误的是哪一项 (B)
 - A.Prefab 是一种资源类型
 - B.Prefab 是一种可以反复使用的游戏对象
 - C.Prefab 可以多次在场景进行实例
 - D. 当一个 Prefab 添加到场景中时，也就是创建了它的一个实例
- 关于 MonoBehaviour.LateUpdate 函数描述错误的是：(B)
 - A. 当 MonoBehaviour 类被启用后，每帧调用一次
 - B. 常被用于处理 Rigidbody 的更新
 - C. 在所有 Update 函数执行后才能被调用
 - D. 常被用于实现跟随相机效果，且目标物体的位置已经在 Update 函数中被更新
- 下列哪个函数不属于碰撞事件 (C)
 - A.OnCollisionEnter
 - B.OnCollisionExit
 - C.OnCollisionUpdate
 - D.OnCollisionStay
- 以下关于 MonoBehaviour.OnGUI() 的描述错误的是 (D)
 - A. 如果 MonoBehaviour 没有被启用，则 OnGUI 函数不会被调用
 - B. 用于绘制和处理 GUI events
 - C. 每帧可能会被绘制多次，每次对应于一个 GUI event
 - D. 每帧被调用一次

- 以下哪组摄像机中 Normalized View Port Rect 的数值设置可以使摄像机显示的画面位于 1280*720 分辨率的屏幕画面右上角 (D)
 - A.X=640,Y=360,W=640,H=360
 - B.X=640,Y=0,W=640,H=360
 - C.X=0,Y=0,W=0.5,H=0.5
 - D.X=0.5,Y=0.5,W=0.5,H=0.5
- 在 Unity 引擎中, Collider 所指的是什么 (D)
 - A.collider 是 Unity 引擎中所支持的一种资源, 可用作存储网格信息
 - B.Collider 是 Unity 引擎中内置的一种组件, 可用对网格进行渲染
 - C.Collider 是 Unity 引擎中所支持的一种资源, 可用作游戏对象的坐标转换
 - D.Collider 是 Unity 引擎中内置的一种组件, 可用作游戏对象之间的碰撞检测
- 以下关于 WWW.LoadFromCacheOrDownload 描述正确的是 (C)
 - A. 可被用于将 Text Assets 自动缓存到本地磁盘
 - B. 可被用于将 Resource 自动缓存到本地磁盘
 - C. 可被用于将 Asset Bundles 自动缓存到本地磁盘
 - D. 可被用于将任意格式的 Unity 资源文件自动缓存到本地磁盘
- 如何实现加载外部视频并播放?
 - 外部视频文件: 目前测试仅支持 ogg 格式 (设置网络获取视频时, 必须将 MIME 设置.ogg 为 application/octet-stream)
 - 场景设置: MainCamera 上添加 AudioSource 脚本; 播放物体上 (如 Plane) 添加 MovieTest 脚本
 - MovieTest 脚本:

```

using UnityEngine;
using System.Collections;

public class MovieTest : MonoBehaviour {
    //视频纹理
    protected MovieTexture movTexture;
    AudioClip audio;
    AudioSource AudioSource1;
    void Start() {
        StartCoroutine(DownLoadMovie());
    }
    void OnGUI() {
        if (GUILayout.Button(" 播放/继续"))
            //播放/继续播放视频
            if (!movTexture.isPlaying)
                movTexture.Play();
                AudioSource1.Play();
        }
        if (GUILayout.Button(" 暂停播放"))
            //暂停播放
            movTexture.Pause();
            AudioSource1.Pause();
        }
        if (GUILayout.Button(" 停止播放"))
            //停止播放
            movTexture.Stop();
            AudioSource1.Stop();
    }
}

```

```

} Ienumerator DownLoadMovie() {
    WWW www = new WWW ("http://127.0.0.1/Wildlife.ogg"); //file://" + Application.dataPath
    yield return www;
    movTexture = www.movie;
    //获取主相机的声源
    AudioSource1 = Camera.main.GetComponent(typeof(AudioSource)) as AudioSource;
    //获取视频的声音设置到声源上
    AudioSource1.clip = movTexture.audioClip;
    audio = AudioSource1.clip;
    //设置当前对象的主纹理为电影纹理
    renderer.material.mainTexture = movTexture;
    //设置电影纹理播放模式为循环
    movTexture.loop = true;
}
}

```

- 游戏对象 B 是游戏对象 A 的子物体，游戏对象 A 经过了旋转，请写出游戏 B 围绕自身的 Y 轴进行旋转的脚本语句，以及游戏对象 B 围绕世界坐标的 Y 轴旋转的脚本语句。
 - 绕世界坐标旋转：transform.Rotate (transform.up*speed*Time.deltaTime);
 - 绕自身 Y 轴旋转：transform.Rotate (Vector.up*speed*Time.deltaTime);
- Unity 中用过哪些插件？具体功能
 - FXMaker, 制作粒子特效；NGUI, 制作 2D 界面；EasyTouch, 摆杆；shaderForge, 制作 shader；Itween, 制作动画；
- 当删除 Unity 工程 Assets 目录下地 meta 文件时会导致什么？为什么？
 - 会导致在场景中游戏对象看不到，或者报错，材质找不到资源。多人协作的时候会导致资源的重复产生。因为每个资源文件都对应一个.meta 文件，这个.meta 文件中的 guid 就是唯一标识这个资源的。材质就是通过这个 guid 来记录自己使用了那些资源，而且同一个资源的 guid 会因为不同的电脑而不同，所以当你上传了丢失了.meta 文件的资源的时候，到了别人的机器上就会重新产生 guid，那个这个资源就相当于垃圾了。
- 频繁创建 GameObject 会降低程序性能为什么？怎么解决？
 - 频繁创建游戏对象，会增加游戏的 Drawcall 数，降低帧率，GPU 会一直在渲染绘制。可以通过对象池来管理对象：当需要创建一个游戏对象时，先去对象池中查找一下对象池中是否存在没有被正在使用的对象，如果说有的话直接使用这个对象，并把它标记为正在使用，没有话就创建一个，并把它添加到池中，然后标记为使用中。一个游戏对象使用完毕的时候，不要销毁掉，把它放在池中，标记为未使用。
- 关于 Vector3 的 API，以下说法正确的是（BC）
 - A.Vector3.normalize 可以获取一个三维向量的法线向量
 - B.Vector3.magnitude 可以获取一个三维向量的长度
 - C.Vector3.forward 与 Vector3(0,0,1) 是一样的意思
 - D.Vector3.Dot(向量 A, 向量 B) 是用来计算向量 A 与向量 B 的叉乘
- 以下哪个函数在游戏进入新场景后会被马上调用（B）
 - A.MonoBehaviour.OnSceneWastLoaded()
 - B.MonoBehaviour.OnSceneEnter()
 - C.MonoBehaviour.OnLevelEnter()
 - D.MonoBehaviour.OnLevelWastLoaded()
- 采用 Input.mousePosition 来获取鼠标在屏幕上的位置，以下表达正确的是（C）
 - A. 左上角为原点 (0, 0)，右下角为 (Screen.Width, Screen.Height)
 - B. 左下角为原点 (0, 0)，右下角为 (Screen.Height, Screen.Width)

- C. 左下角为原点 (0, 0), 右上角为 (Screen.Width, Screen.Height)
 - D. 左上角为原点 (0, 0), 右下角为 (Screen.Height, Screen.Width)
- 如何通过脚本来删除其自身对应的 Gameobject (A)
- A.Destroy(gameObject)
 - B.this.Destroy()
 - C.Destroy(this)
 - D. 其他三项都可以
- 某个 GameObject 有一个名为 MyScript 的脚本, 该脚本中有一个名为 DoSomething 的函数, 则如何在该 Gameobject 的另外一个脚本中调用该函数? (A)
- A.GetComponent().DoSomething()
 - B.GetComponent
 - C.GetComponent().Call("DoSomething")
 - D.GetComponent
- Animator.CrossFade 命令作用是: (B)
- A. 动画放大
 - B. 动画转换
 - C.Update()
 - D.OnMouseButton()
- OnEnable,Awake,Start 运行时的发生顺序? (A)
- A.Awake->OnEnable->Start
 - B.Awake->Start->OnEnable
 - C.OnEnable-Awake->Start
 - D.Start->OnEnable->Awake
- 以下选项中, 正确的是 (D)
- A.Mathf.Round 方法作用是限制
 - B.Mathf.Clamp 方法作用是插值
 - C.Mathf.Lerp 方法作用是四舍五入
 - D.Mathf.Abs 方法作用是取得绝对值
- 以下选项中, 将游戏对象绕 Z 轴逆时针旋转 90 度 (C)
- A.transform.rotation = Quaternion.Euler(0,0,90)
 - B.transform.rotation = Quaternion.Angle(0,0,90)
 - C.transform.Rotate(new Vector3(0,0,90))
 - D.transform.Rotate(new Vector3(90,0,0))
- public static function InitializeServer(connections:int,listenPort:int,useNat:bool):NetworkConnectionError;
解释一下函数, 参数以及返回值的意思。
- 初始化服务器。connections 是允许的入站连接或玩家的数量, listenPort 是要监听的端口, useNat 设置 NAT 穿透功能。如果你想要这个服务器能够接受连接使用 NAT 穿透, 使用 facilitator, 设置这个为 true。如果有错误会有返回错误。
- 请写出以下函数的含义和运算结果

```

1  delegate b Func<a, b>(a a1);
2  static void Main(string[] args) {
3      Func<int, bool> mFunc = x => x == 5;
4      Console.WriteLine(mFunc(6));
5  }

```

- false, 就是定义一个 delegate, 返回值类型为 b, 有一个参数, 类型为 a。

- 编写一个函数, 输入一个 32 位整数, 计算这个整数有多少个 bit 为 1.

```
1 uint BitCount (uint n) {
2     uint c = 0; // 计数器
3     while (n > 0) {
4         if ((n & 1) == 1) // 当前位是 1
5             ++c; // 计数器加 1
6         n >>= 1; // 移位
7     }
8     return c;
9 }
```

- 某游戏中的装备系统有 16 种附加属性, 每种附加属性使用一个 32 位的 ID 表示 (比如 10001 表示加人物 hp 的附加属性, 10002 表示加人物 mp 的附加属性), 一件装备最多有 4 个附加属性, 请写一个程序输出所有附加属性的组合。
- 请实现如下函数, 在 Unity 中有一副骨骼树, 请使用递归方式与非递归方式实现先序遍历, 在 Unity 的 Console 输出所有骨骼名。

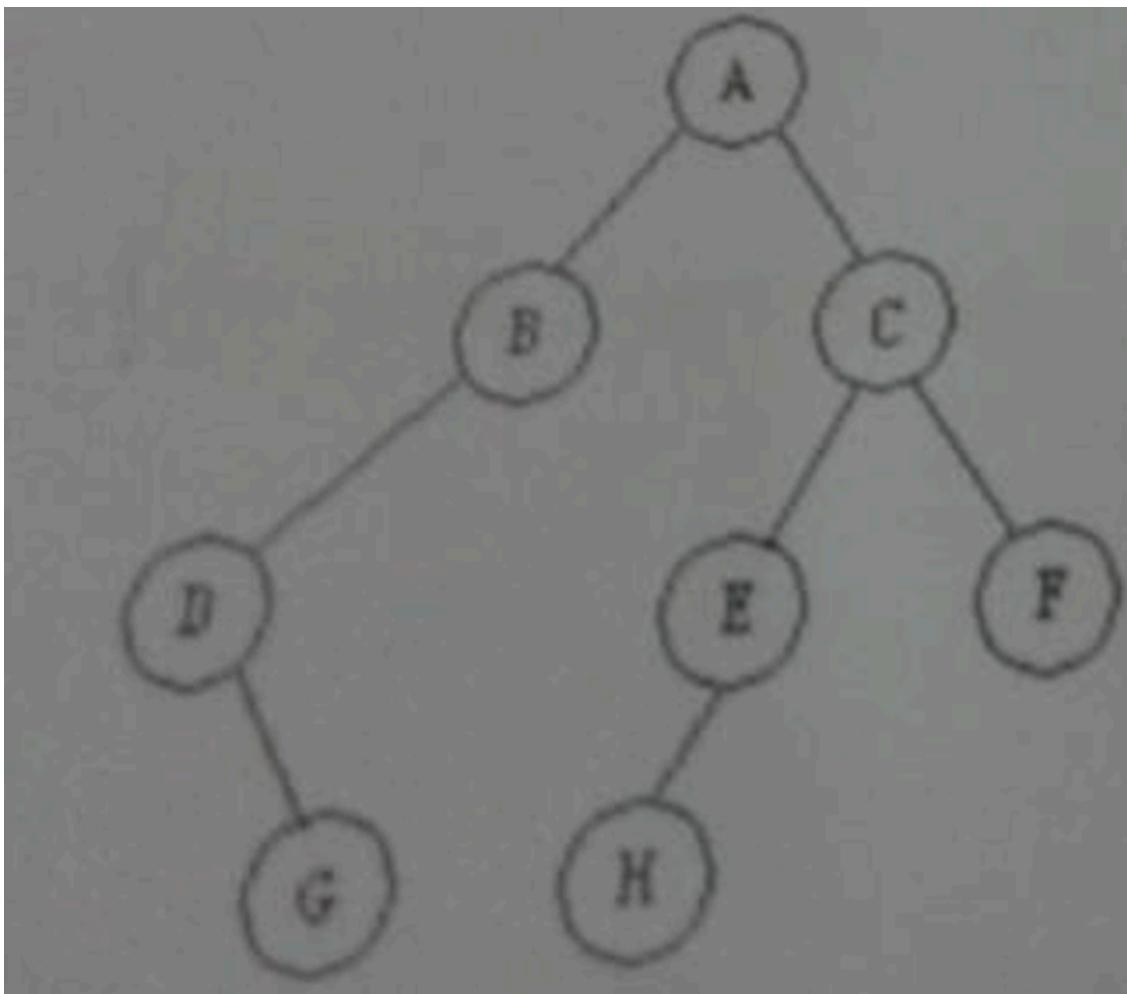
- 可能用到的函数

```
1 public Transform GetChild [int index];
2 public int Transform.childCount
3 public void OutputTree(Transform root) {}
```

- 简要解释下数据库中 ACID 的含义。

- ACID 是指在可靠数据库管理系统 (DBMS) 中, 事务所具有的四个特性: 原子性 (Atomicity)、一致性 (Consistency)、隔离性 (Isolation)、持久性 (Durability)。
- 原子性是指事务是一个不可分割的工作单位, 事务中的操作要么都发生, 要么都不发生。
- 一致性是指在事务开始之前和事务结束以后, 数据库的完整性约束没有被破坏。这是说数据库事务不能破坏关系数据的完整性以及业务逻辑上的一致性。例如: 对银行转帐事务, 不管事务成功还是失败, 应该保证事务结束后 ACCOUNT 表中 aaa 和 bbb 的存款总额为 2000 元。
- 隔离性多个事务并发访问时, 事务之间是隔离的, 一个事务不应该影响其它事务运行效果。这指的是在并发环境中, 当不同的事务同时操纵相同的数据时, 每个事务都有各自的完整数据空间。由并发事务所做的修改必须与任何其他并发事务所做的修改隔离。事务查看数据更新时, 数据所处的状态要么是另一事务修改它之前的状态, 要么是另一事务修改它之后的状态, 事务不会查看到中间状态的数据。
- 持久性, 意味着在事务完成以后, 该事务所对数据库所作的更改便持久的保存在数据库之中, 并不会被回滚。即使出现了任何事故比如断电等, 事务一旦提交, 则持久化保存在数据库中。

- 32 位整数 256 和 255 按位异或后的结果是 (511)
- unix/linux 系统将所有的 I/O 设备模型化为文件,c 语言中的 (stdin) ,(stdout) 和 (stderr) 分别表示标准输入, 标准输出, 标准错误。
- k 层二叉树最多有 $2^k - 1$ 个结点。
- 中缀算式 $(8 + x * y) - 2x / 3$ 的后缀算式是
- 对以下二叉树进行前序遍历的结果是 (ABDGCEHF)



- 写出一个 C 类 ip 地址（192.168.1.105），该 ip 地址一个合法的网格掩码是（255.255.255.224）。
- c/c++ 程序运行时有堆内存与栈内存之分，请写一个语句在堆中分配一个整数：(int a = new int(4)), 在栈内存中分配一个整数：(int a = 5)。
- 简述从 C/C++ 源代码生成可执行文件所经历的几个过程和每个过程中所做的事情。

预编译-》编译-》链接-》执行

- 简述 TCP 建立连接的过程，最好能画出时序图。

– 三次握手；

- 有一个 c 语言函数

```

1 unsigned int getN(unsigned int v){
2     v--;
3     v |= v >> 1;
4     v |= v >> 2;
5     v |= v >> 4;
6     v |= v >> 8;
7     v |= v >> 16;
8     v++;
9     return v;
10 }
```

请问这个函数的功能是什么？

- 返回的结果是 v 或者 $2^{\lfloor \log_2 v \rfloor} + 1$ （比如 125 返回 128, 128 返回的就是 128）
- 列出 c 语言中 static 关键字的用法。

- static 关键字至少有下列几个作用：

- * (1) 函数体内 static 变量的作用范围为该函数体，不同于 auto 变量，该变量的内存只被分配一次，因此其值在下次调用时仍维持上次的值；
- * (2) 在模块内的 static 全局变量可以被模块内所用函数访问，但不能被模块外其它函数访问；
- * (3) 在模块内的 static 函数只可被这一模块内的其它函数调用，这个函数的使用范围被限制在声明它的模块内；
- * (4) 在类中的 static 成员变量属于整个类所拥有，对类的所有对象只有一份拷贝；
- * (5) 在类中的 static 成员函数属于整个类所拥有，这个函数不接收 this 指针，因而只能访问类的 static 成员变量。

- 解释一下 Unity3D 中的协程 (coroutine) 是什么？并举例说明

- 在主线程运行的同时开启另一段逻辑处理，来协助当前程序的执行，协程很像多线程，但是不是多线程，Unity 的协程实在每帧结束之后去检测 yield 的条件是否满足。

- 物体自旋转使用的函数叫什么？物体绕某点旋转使用函数叫什么？

- 物体自旋转函数 transform.Rotate，物体绕某点旋转函数：transform.RotateAround

- 使用 prefab 预制物体有什么好处？

- 1.Prefab 一个重要的优势就是编辑 prefab 资源后，场景中所有使用 Prefab 克隆的游戏对象将全部使用新编辑的资源，无需一个一个的给场景中的对象赋值。
 - 2. 当游戏中需要频繁创建一个物体时，使用 Prefab 能够节省内存。
 - 3. 当你在一个场景中增加一个 Prefabs，你就实例化了一个 Prefabs。

- 设有如下关系表 R1 和 R2

- R1(NO, NAME, SEX, AGE, CLASS)

- R2(NO, SUBJECT, SCORE)

- 主关键字是 NO，其中 NO 为学号，NAME 为姓名，SEX 为性别，AGE 为年龄，CLASS 为班号，SUBJECT 为科目，SCORE 为成绩。写出实现下列功能的 SQL 语句。查找学号为 20 的学生的姓名，科目，成绩。

- SELECT NAME,SUBJECT,SCORE FORM R1 INNER JOIN R2 ON R1.NO=R2.NO WHERE R1.NO = 20

- 描述多人联网中位置的同步和聊天通讯的实现方法，并编程写出如何实现多人协同中交互操作。（交互操作例如，血值为 100 的两个角色人物可以相互射击双方，射击一次减去血值 10，当血值为 0 的时候证明已经死亡）

- 当实例化一个 prefab 对象，prefab 对象上面挂载了一个继承 MonoBehaviour 的脚本

- 1. 请问这个脚本分别会按顺序调用哪些函数，并列出哪些函数是反复进去的

- 2. 请问当这个实例化对象先调用了 SetActive(false)，然后又调用了 SetActive(true)，那么这个脚本依次会调用哪些函数方法？

- 3. 如果在脚本的 Awake () 函数中，调用了 this.gameObject.AddComponent()(PS:GameController 为另一个继承 Mono 的脚本类)，请问此时脚本函数的调用顺序是否发生变化？

- if(go.CompareTag("Enemy")) 和 if (go.tag == "Enemy") 两种判断方法哪种是合适的，为什么？

- 第一种合适，因为第二种会占用更多内存。

- DestroyImmediate 和 Destroy 的区别是？

- DestroyImmediate 销毁对象的时候，会立即释放资源。Destroy 只是从该场景销毁，但是还在内存当中。

- 详细说明 Unity 中资源加载的方法，以及他们的区别？

- 1. 通过 Resources 模块，调用它的 load 函数：可以直接 load 并返回某个类型的 Object，前提是需要把这个资源放在 Resource 命名的文件夹下，Unity 不管有没有场景引用，都会将其全部打入到安装包中。

- 2. 通过 bundle 的形式：即将资源打成 asset bundle 放在服务器或本地磁盘，然后使用 WWW 模块 get 下来，然后从这个 bundle 中 load 某个 object。

- 从代码角度上如何进行内存优化？
- 分别写出 Invoke 和协程的几种用法？
- 以下函数的功能是计算斐波那契数列的第 n 项，请填空

```
1 int func(int n) {  
2     if(n<=2) return 1;  
3     return n + func(n-1);  
4 }
```

- C 语言中宏定义中若有多行，可以使用字符 (\) .
- C 语言中 32 位整数能表达的最小的数是- 2^{31} ~ $2^{31}-1$.
- 使用（点乘）向量运算可以很方便地计算出三维空间中两个单位向量之间夹角的 cos 值。
- 类 unix 系统中某文件的权限为：drw-r-r-，用八进制数值形式表示该权限为 (411)，首位 d 代表目录（文件夹）
- 若有如下程序段，其中 s, a, b, c 均已定义为整型变量，且 a, c 均已赋值 (c 大于 0)

```
1 s = a;  
2 for(b = 1; b <= c; b++)  
3     s = s + 1;
```

则与上述程序段功能等价的赋值语句是 (B)

- A.s = a + b
- B.s = a + c
- C.s = s + c
- D.s = b + c
- 简述 static 和 const 关键字的作用

- static 关键字至少有下列几个作用：

- * (1) 函数体内 static 变量的作用范围为该函数体，不同于 auto 变量，该变量的内存只被分配一次，因此其值在下次调用时仍维持上次的值；
- * (2) 在模块内的 static 全局变量可以被模块内所用函数访问，但不能被模块外其它函数访问；
- * (3) 在模块内的 static 函数只可被这一模块内的其它函数调用，这个函数的使用范围被限制在声明它的模块内；
- * (4) 在类中的 static 成员变量属于整个类所拥有，对类的所有对象只有一份拷贝；
- * (5) 在类中的 static 成员函数属于整个类所拥有，这个函数不接收 this 指针，因而只能访问类的 static 成员变量。

- const 关键字至少有下列几个作用：

- * (1) 欲阻止一个变量被改变，可以使用 const 关键字。在定义该 const 变量时，通常需要对它进行初始化，因为以后就没有机会再去改变它了；
- * (2) 对指针来说，可以指定指针本身为 const，也可以指定指针所指的数据为 const，或二者同时指定为 const；
- * (3) 在一个函数声明中，const 可以修饰形参，表明它是一个输入参数，在函数内部不能改变其值；
- * (4) 对于类的成员函数，若指定其为 const 类型，则表明其是一个常函数，不能修改类的成员变量；
- * (5) 对于类的成员函数，有时候必须指定其返回值为 const 类型，以使得其返回值不为“左值”。

- 用你熟悉的语言及你认为最简洁的方法书写计算 $s = 1!+2!+3!+\cdots+num!$ 的代码。num 为输入，s 为输出。0 代表阶乘 $3! = 1 * 2 * 3$

```

1 Console.ReadLine(num)
2 int s = 0;
3 for(int i = 1; i <= num; i++) {
4     s += JieCheng(num);
5 }
6 public int JieCheng(int num) {
7     if(num < 0) {
8         Console.WriteLine("error");
9         return;
10    }
11    if(num <=1) {
12        return 1;
13    } else {
14        return num * JieCheng(num - 1)
15    }
16 }

```

- 用你熟悉的语言从一个字符串中去掉相连的重复字符，例如原字符串“adffjkljaalkjhl”变为“adfjkljalkjhl”

```

1 int GetResult(char[] input, char[] output) {
2     int i, j, k = 0;
3     int flag;
4     int length;
5     if(input == null || output == null) {
6         return -1;
7     }
8     length=strlen(input); //求数组的长度
9     for(i = 0; i<length; i++) {
10         flag = 1;
11         for(j = 0; j < i; j++) {
12             if(output[j] == input [i])
13                 flag = 0;
14         }
15         if(flag)
16             output [k++] = input [i];
17     }
18     printf(" 最终的字符串为: ");
19     output[k] = '\0';
20     for(int m = 0; m < output.Length; m++) {
21         print (output [m]);
22     }
23     return 0;
24 }

```

14 第二部分

- 哪种实时光源是 Unity 中没有的? (D)
 - A. 点光源
 - B. 方向光
 - C. 聚光灯
 - D. 日光灯
- 如何在 Unity 中创建地形系统? (D)
 - A.Terrain->Create Terrain
 - B.Component->Create Terrain
 - C.Asset->Create Terrain

- D.Windows->Create Terrain
- 以下哪种操作步骤可以在场景中添加“Wind Zone”?(B)
 - A.Terrain->Wind Zone
 - B.GameObject->Create other->Wind Zone
 - C.Component->Physics->Wind Zone
 - D.Assets->Create->Wind Zone
- 在 Unity 编辑器中创建一个 Directional Light, 以下步骤正确的是? (B)
 - A.Edit->Rendering Setting->Directional Light
 - B.GameObject->Create Other->Directional Light
 - C.Component->Rendering->Directional Light
 - D.Assets->Directional Light
- 下列哪一项不属于 Camera 中的“Clear Flags”? (D)
 - A.Skybox
 - B.Solid Color
 - C.Depth Only
 - D.Background
- 以下哪种脚本语言是 Unity 编辑器所不支持的? (D)
 - A.Javascript
 - B.C#
 - C.Boo
 - D.Perl
- 对于 Prefab, 以下说法错误的是? (D)
 - A.Prefab 资源可以在项目中多次重复使用
 - B.由 Prefab 实例出的 GameObject, 其在 Hierarchy 试图中表现为蓝色
 - C.Prefab 上的组件信息一经改变, 其实例出的 GameObject 也会自动改变
 - D.实例出的 GameObject 上的组件信息一经改变, 其对应出的 Prefab 也会自动改变
- 下面哪种做法可以打开 Unity 的 Asset Store? (A)
 - A.Windows->Asset Store
 - B>Edit->Asset Store
 - C.File->Asset Store
 - D.Assets->Asset Store
- 在哪个面板中可以修改物体的空间属性, 如位置, 朝向, 大小等 (B)
 - A.Project
 - B.Inspector
 - C.Hierarchy
 - D.Toolbar
- 如何为一个 Asset 资源设定一个 Label, 从而能够方便准确的搜索到? (D)
 - A.在 Project 窗口中选中一个 Asset, 右键->Create->Label
 - B.在 Project 窗口中选中一个 Asset, 右键->Add Label
 - C.在 Project 窗口中选中一个 Asset, 在 Inspector 窗口中点击添加 Label 的图标
 - D.在 Project 窗口中选中一个 Asset, 在 Inspector 窗口中点击按钮“Add Label”

- Mecanim 系统中，Body Mask 的作用是? (D)
 - A. 指定身体的某一部分是否参与骨骼动画
 - B. 指定身体的某一部分是否参与物理模拟
 - C. 指定身体的某一部分是否可以输出骨骼信息
 - D. 指定身体的某一部分是否参与渲染
- 以下哪种操作步骤可以打开 Unity 编辑器的 Lightmapping 视图? (C)
 - A.File->Lightmapping
 - B.Assets->Lightmapping
 - C.Windows->Lightmapping
 - D.Component->Lightmapping
- 下列关于光照贴图，说法错误的是? (C)
 - A. 使用光照贴图比使用实时光源渲染要快
 - B. 可以降低游戏内存消耗
 - C. 可以增加场景真实感
 - D. 多个物体可以使用同一张光照贴图
- 如何为物体添加光照贴图所使用的 UV?(B)
 - A. 不用添加，任何时候都会自动生成
 - B. 更改物体导入设置，勾选“Generate Lighting UVs”
 - C. 更改物体导入设置，勾选“Swap UVs”
 - D. 更改物体导入设置，在 UVs 选项中选择“Use Lightmaps”
- 在哪个模块下可以修改 Render Path? (A)
 - A.Camera
 - B.Light
 - C.Render Settings
 - D.Project Setting->Quality
- 以下哪项技术下不是目前 Unity 所支持的 Occlusion Culling 技术? (D)
 - A.PVS only
 - B.PVS and dynamic objects
 - C.Automatic Portal Generation
 - D.Dynamic Only
- 关于 Vector3 的 API, 以下说法正确的是? (C)
 - A.Vector3.normalize 可以获取一个三维向量的法线向量
 - B.Vector3.magnitude 可以获取一个三维向量的长度
 - C.Vector3.forward 与 Vector3 (0, 0, 1) 是一样的
 - D.Vector3.Dot(向量 A, 向量 B) 是用来计算向量 A 与向量 B 的叉乘
- 下列那些选项不是网格层属性的固有选项? (B)
 - A.Default
 - B.Walkable
 - C.Not Walkable
 - D.Jump
- 写出你对游戏的理解及游戏在生活中的作用，对 Unity3D 软件理解最深入的地方。

15 C++

- 请写代码打印 100 之内的素数，讲求效率（请做你的解法的效率分析）
- 求 m,n 的最大公约数
- 输入 10 个字符串，打印出其中重复的字符串以及重复的次数
- 请画图例（UML 最好），给出 windows 下的文件目录的设计模式
- 用 OO 表示狼吃羊羊吃草
- 什么是 subversion? 它与 vss,cvs 的区别在哪？或者有什么优势？
- 什么是 wiki，关于程序项目的 wiki 你使用过哪些？wiki 对你有什么帮助吗？wiki 与程序文档的差别在哪？
- 什么是 tdd? 你使用过吗？tdd 的关键在哪？跟传统的单元测试相比，有什么优越性？
- 什么是单元测试？你用过哪些单元测试工具？他们的区别和好处各有哪些？你主要倾向于哪一种？
- 什么是编程规范？你倾向于什么样的规范？他的好处在哪？
- 什么是 mfc？你经常使用那些 mfc 类？那么为什么很多人不主张使用 mfc？
- 什么是头文件依赖？你注意过这些问题吗？你注意过编译的时间吗？你怎么改进编译时间？
- 什么是面向对象？你在哪些方面用过面向对象？带来了什么好处？又有什么弊端？
- 什么是接口编程.com，他带来了什么好处？适用于什么地方？
- 什么是设计模式？使用设计模式有什么好处？列举你使用过的各种设计模式知识：
- 一寸山河一寸血，_____
- 抗战历时 _____