Unity Game Developer Postion Interview

deepwaterooo

February 9, 2018

Contents

1 [Unity] 面试题整理 100 题

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

1.1 相机

1.1.1 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。

1.1.2 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。

1.1.3 **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。

1.1.4 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。

1.2 光源

1.2.1 : **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

1.2.2 写光照计算中的 **diffuse** 的计算公式

- 实际光照强度 I= 环境光 (Iambient) + 漫反射光 (Idiffuse) + 镜面高光 (Ispecular);
- 环境光：Iambient= Aintensity* Acolor; (Aintensity 表示环境光强度，Acolor 表示环境光颜色)
- 漫反射光：Idiffuse = Dintensity*Dcolor*N.L; (Dintensity 表示漫反射强度，Dcolor 表示漫反射光颜色，N 为该点的法向量，L 为光源向量)
- 镜面反射光：Ispecular = Sintensity*Scolor*$(R.V)^n$; (Sintensity 表示镜面光照强度，Scolor 表示镜面光颜色，R 为光的反射向量，V 为观察者向量，n 称为镜面光指数)

1.2.3 写出光照计算中的 **diffuse** 的计算公式

- diffuse = Kd x colorLight x max(N*L,0);
- Kd - 漫反射系数、colorLight - 光的颜色、N - 单位法线向量、L - 由点指向光源的单位向量,
- 其中 N 与 L 点乘，如果结果小于等于 0，则漫反射为 0。

1.2.4 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。

1.3 物理碰撞、刚体等

1.3.1 **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 IsTrigger = 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 IsTrigger = 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

1.3.2 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。

1.3.3 **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。

1.3.4 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

1.3.5 **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

1.3.6 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。

1.4 3D 数学

1.4.1 四元数有什么作用?

- 对旋转角度进行计算时用到四元数

1.4.2 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）

1.4.3 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

1.4.4 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

1.4.5 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

1.4.6 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。

1.4.7 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

1.5 GUI

1.5.1 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。

1.5.2 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

1.5.3 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。

1.5.4 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。

1.5.5 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

1.5.6 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

1.6 生命周期

1.6.1 **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

1.6.2 **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!

1.6.3 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。

1.6.4 **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。
  - 纹理帖图：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。

1.6.5 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。

1.6.6 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。

1.6.7 简述 SkinnedMesh 的实现原理

1.7 渲染

1.7.1 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。

1.7.2 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。

1.7.3 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）

1.7.4 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

1.7.5 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。

1.7.6 Unity 的 Shader 中，Blend SrcAlpha OneMinusSrcAlpha 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）

1.7.7 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，以产生波光玲玲的效果。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

1.7.8 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。

1.7.9 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

1.7.10 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。

1.7.11 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。

1.8 其它

1.8.1 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，同事 prefab 打包的内容简化了导出的操作，便于团队的交流。

1.8.2 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

1.8.3 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。

1.8.4 物体自身旋转使用的函数?

- Transform.Rotate()

1.8.5 **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()

1.8.6 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

1.8.7 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。

1.8.8 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

1.8.9 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。

1.8.10 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。

1.8.11 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）

1.8.12 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
}
```

1.8.13 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

1.8.14 **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。

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

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

1.8.16 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。

1.8.17 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。

1.8.18 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

1.8.19 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。

1.8.20 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。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，把资源达成几个小的 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。

1.8.21 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/LOAssetFramework.git

1.9 与 Mobile 端交互

1.9.1 Unity 和 Android 与 iOS 如何交互?

1.9.2 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。

1.9.3 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
- 16filldrawcall 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

1.9.4 你用过哪些插件?

1.10 语言基础、OOP

1.10.1 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。

1.10.2 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

1.10.3 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。

1.10.4 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：对该类和其派生类公开
- internal：只能在包含该类的程序集中访问该类
- protected internal：protected + internal

1.10.5 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

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

1.10.6 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。

1.10.7 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

1.10.8 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)。

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

- 使用 Destroy() 方法;

1.10.10 能用 foreach 遍历访问的对象需要实现 __

- IEnumerable；GetEnumerator

1.10.11 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。

1.10.12 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[1](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：

```
int iLength=strlen(strcpy(strA,strB));
// or
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。

1.10.13 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。

1.10.14 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。

1.10.15 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。

[1]DEFINITION NOT FOUND.

- 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。

1.10.16 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

1.10.17 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。

1.10.18 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。
  - STATread：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。

1.10.19 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 Responsiblity 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 Pricinple，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。

1.10.20 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。

1.10.21 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'});

1.10.22 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。

1.10.23 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。

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

- 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

1.10.25 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。

1.10.26 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。

1.10.27 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.
    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 (waitTime : float) {
    // suspend execution for waitTime seconds
    // 暂停执行 waitTime 秒
    yield WaitForSeconds (waitTime);
    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。

1.10.28 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

1.10.29 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。

1.10.30 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。

1.10.31 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

1.10.32 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；

1.10.33 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。

1.10.34 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。

1.10.35 .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。

1.10.36 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 Pinvoke 完成一些底层操作

1.10.37 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