* **代码优化**

1. Stringbuilder是动态的，针对字符串有大量的修改，可以使用；string的修改会申请新的内存
2. 访问枚举器就会动态申请内存，造成GC，所以不能大量使用foreach嵌套，除非是自己实现的可枚举类型，进行了优化，常见的都是下面这样

public Enumerator GetEnumerator() {

return new Enumerator(this);

}

1. 数组内存连续
2. LIst的效率不高，add方法会在容量不够的时候new新的数组，容量双倍扩容，可以在new的时候传入容量，减少GC，remove并不会缩小内存，使用array.copy来移动覆盖数组，内部的大部分算法是线性算法，sort是快速排序，可以自己实现一个数据结构来优化，总的来说list的兼容性好，但是内部没有优化，线程不安全
3. Dictionary内部由两个数组构成int[]和entity[]，由哈希函数完成地址构建（计算出的key的hashcode就是数组的地址），由拉链法冲突解决方式，遍历冲突的链表key值比较来确定位置，效率上也是存在扩容GC，在数量不够时和hash碰撞过高时扩容，移除并不会减少内存，线程不安全，查找比遍历节点（o(n)复杂度）的效率要高
4. c#的Linklist为双向链表，持有前后节点和当前链表，要删除一个节点，就是将前一个节点的next指向自己的next，自己的next置空
5. 浮点数精度问题，先等的比较方法采用相减和精度对比，可以乘以系数计算完之后再除以系数确保计算结果准确，可以自己扩展采用两个整数来表示整数位和小数位，分别实现相应的操作符和数据转换的扩展
6. Event和delegate，event是对委托的进一步封装，隐藏了赋值操作，避免前面的委托链条被替换
7. 装箱拆箱：当通用类型作为参数或者返回值比如object，在使用的时候会存在装拆操作，会申请新的对象，造成GC，可以使用泛型，或者继承的方式来避免装拆箱
8. 编码的好坏：编码的经验，计算机原理的理解，算法的理解运用，算法追求高效，常用的算法排序和搜索
9. 快速排序 二分排序，左右分块，找出一个参考值，一般选最左边元素，遍历将小于参考值的元素顺序填入左边，大于的放到右边，几次迭代直到左右相等。
10. 快速排序的优化：参考值的选取很关键，如果选的偏大或者偏小，更坏的情况选的是最大最小值，效率更低，优化方法，随机数选取中轴数，头中尾先排序再选则中轴数，小区间使用插入排序，元素小于等于8个效率高，和中轴数相等的集中到一起，缩小分割区域
11. 堆排序 基于完全二叉树实现
12. 搜索算法，依赖好的数据结构，找出特定的元素或者具有共同特征的一组数据
13. 二分查找法：需要保证数组有序，二分查找注意区间的写法，和左右边界的查找（目标的重复的闭区间），二叉树，红黑树，B树等等
14. 字符串造成GC，可以使用字符串缓存机制，一个对应一个id存到dictionary，或者使用unsafe代码使用指针直接赋值，字符串的length存在遍历char[]数组，所以频繁使用length消耗挺大，equal判断相等先判断内存地址，再看长度，再遍历字节
15. 内存分为代码段内存（指令内存），数据段内存（动态数据也就是堆内存，静态数据），栈内存，类的方法都是编译后存到代码段内存
16. 协程原理：基于枚举器实现，yield关键字会自动创建枚举器类，yield return 返回的是current的值，然后在unity的loop中根据current的值判断是否要执行movenext

* **系统架构**

1. 在软件系统架构中，架构承担了解决项目从研发到上线运营的方案。评估特性：承载力（稳定性），可扩展，易用，可伸缩，容错力错误感知
2. 架构思想 抽象 分层 分治 衍化（不断修改进步）

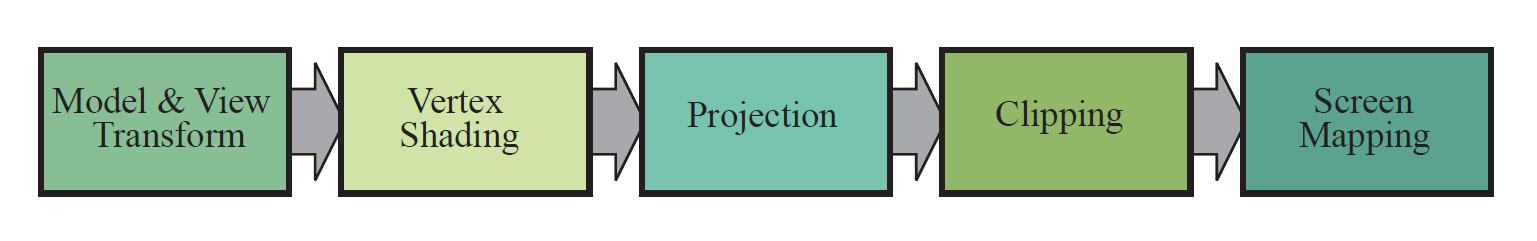
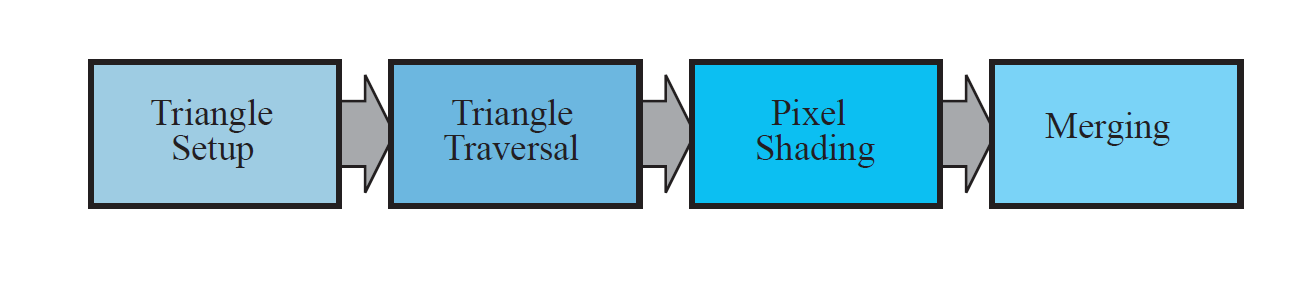
* **UGUI源码和优化总结**

1. 结构： 事件数据模块（eventdata） 输入检测模块（inputmodule）射线检测模块（GraphicRaycaster）事件处理系统（eventsystem）
2. 架构UI框架：UI管理类，UI基类，输入响应（继承接口，或者绑定事件），自定义组件（UI动画，按钮声音，无限滚动列表等）
3. UI动静分离和拆分，资源的预加载（提前加载到内存，或者提前实例化隐藏了）
4. UI图集的优化：alpha分离，主要是在ios上压缩格式带透明通道的压缩后模糊（UGUI已经实现了，不分离会因为图集的压缩导致显示的错误），分离后就要使用两张图来渲染
5. 字体拆分 节省内存，提高加载的速度
6. 滚动列表的优化（使用对象池动态创建，不要一次全部创建出来）
7. 改变颜色就要去改变顶点的颜色，就需要重新合并mesh，改变image的alpha就会生成新的材质球，如果每帧都改变alpha会一直申请材质球效率低下，优化方案是自己创建一个UI的材质球，赋值给UI组件，单独来修改材质的颜色
8. 界面打开和关闭的优化：预加载，关闭时隐藏打开再展示（会触发mesh重构），移除屏幕（降低GPU的消耗，CPU不会），设置layout渲染层级
9. 导入图片后的属性设置，unity的图片资源导入后就转化为unity自己的图片格式，写入权限会在内存中存储双倍，所以不会修改的话关闭（unity默认关闭），UI可以关闭mipmap（选了sprite后默认就没有mipmap选项），减小内存占用

* **内存优化**

1. 托管内存优化，减少GC，资源占用的是native内存（就是内存条），使用.net memory profiler工具查找内存泄露（内存得不到回收，占用的内存越来越多）
2. 采用资源加载队列，限制每次加载数量，做到很好的缓冲，update中检测是否需要去加载，加载完成后再通知去实例化，实例化时也要注意数量的控制，对于加载的资源管理起来，引用计数为0 时去卸载，如果所在的ab引用计数也为0，同时卸载，为了避免频繁的卸载，可以使用卸载倒计时缓冲
3. 打包AB时要做好颗粒度的分配，过大相当于都打在了一起，内存占用过大，过细导致相当于没有打，频繁的加载卸载

* **渲染优化**

1. 理解opengl和directx只是图形渲染API不同的实现，应用程序调用=》API=》系统=》显卡驱动=》显卡渲染，unity又把调用API的接口封装，我们只管绘制
2. 我们可以描述说渲染管线是一系列数据处理的过程，这个过程最终的目的是将应用程序的数据经过计算最终输出到帧缓存上最后输出到屏幕。渲染管线从接受到渲染命令后开始，分几个阶段处理了这些数据，这几个阶段分别是应用阶段，几何阶段，光栅化阶段，经过这几个阶段处理最终输出到屏幕上
3. 应用阶段：准备渲染数据，包括mesh，贴图，材质，光源，摄像机；提交数据前的优化操作：批处理，视锥剔除（包围盒是否在相机视锥体内），遮挡剔除
4. 渲染命令队列，所有的drawcall都是加入命令队列，即我们通常所说的 Draw call 太多时GPU的工作效率比较差。其根本原因就是 CPU 处理大量的渲染命令的效率比较低，所以要合批
5. 几何阶段：几何阶段分拆成了，顶点着色器、曲面细分着色器、细分计算着色器、几何着色器、图元装配、裁剪6个节点；曲面细分着色器、细分计算着色器、几何着色器，这三个着色器是非必须着色器，很多手机设备上的GPU并没有这几个功能。细分着色器包括曲面细分着色器和细分计算着色器会使用面片来描述一个物体的形状，并且增加顶点和面片数量使得模型外观更加平顺。几何着色器则允许自定义增加和创建新的图元，这是唯一一个能自定义增加新图元的着色器；在归一化坐标后进行裁剪
6. 光栅化阶段：光栅化、片元着色器、逐片元操作；光栅化：三角形设置和三角形遍历确定哪些像素被覆盖，进行顶点的插值，得到片元数据；片元着色器：确定片元如何绘制
7. 逐片元操作：逐片元操作(Per-Fragment Operations)是OpenGL的说法，在DirectX称为输出合并阶段(Output-Merger)，其实只是说法不同而已包含的内容都是相同的，它包括了，剪切测试(Scissor test)、多重采样的片元操作、alpha测试、模板测试(Stencil Test)、深度测试(Depth Test)、混合(blending)、以及最后还有个逻辑操作
8. 帧缓冲显示和绘制采用双缓冲的模式，前后缓冲置换指针
9. 渲染顺序：多个相机 ：相机的depth小先画，同一个相机内：材质的render mode（shader的render queue）,同一个queue的按深度值排序（Unity3D规定2500以下的索引号，排序规则以根据摄像机的距离由近到远顺序渲染（不透明的物体，遮挡剔除了远的绘制），2500索引号以上的渲染队列标号则相反，排序规根据摄像机的距离由远到近顺序渲染排列），对于两个半透明的物体有交叉时，可能渲染顺序出错需要手动控制render queue
10. Alpha test 比较alpha的值，进行clip裁剪，多用于绘制树和草，使用面片加贴图alpha test即可实现小的缝隙 的效果
11. 因为深度测试是发生在片元着色器之后的，所以被剔除的片元之前也参与了计算，造成大量的浪费，硬件上的优化方法是Early-Z（todo需要理解），在片元着色器计算之前，提前进行了深度测试，但是如果使用clip丢弃片元或者修改深度值，会导致失效
12. Mipmap层级计算：在OpenGL中Mipmap的计算就依赖于片元中的uv求偏导值，片元所映射的uv范围越大，计算出来的Mipmap层级越高，纹理贴图选取的分辨率就越小
13. 纹理采样滤波方式：nearest（就近），liner（biliner，triliner），anisotropic（各向异性）
14. 阴影投射：屏幕空间阴影投射（screen space shadow map），当Unity3D在渲染时会首先在当前Shader中找到LightMode为ShadowCaster的Pass，如果没有则会在Fallback指定的Shader中继续寻找，如果没有则无法产生阴影
15. 全局光照=间接光+直接光\*阴影系数
16. 动态合批：遍历模型，符合条件的模型合并网格，统一提交渲染；条件限制比较苛刻，目的是为了减少cpu的合批负载
17. 静态合批：物体必须标记为静态，离线计算网格合并，降低了drawcall，但是合并的网格会一起加载，消耗了更多的内存
18. 动态和静态合批必须要相同材质球实例，即材质参数一致，对于不同参数的实例相同mesh的物体，使用GPU Instance技术，在具体的材质上勾选，例如不同的旋转，缩放，颜色，从图形调用接口上来说 GPU Instancing 调用的是 OpenGL 和 DirectX 里的多实例渲染接口，通过instance id来获取数据数组的数据；注意不适用于skinmesh，并且贴图不同也不支持，如果开启instance，合批失败将没有动态合批，注意测试，有时候效率会很低，并且最好是连续绘制时有效
19. Unity的occlusion culling，需要bake遮挡数据，物体可以动态也可静态，相机要勾选遮挡剔除
20. GPU上的multisampling和antialiasing，这些技术在opengl中实现，我们只是选择开还是关，或者设置一些参数，消耗只有gpu，对cpu不影响
21. UnityShader的编译：在UnityEditor中类似Jit及时编译，构建着色器对象，构建着色器程序，绑定着色器对象，链接对象形成着色器程序，然后执行，如果是build player，会将所有用到的shader编译好，有些可能在editor中没有用到，用不到的shader variants会被strip剥离
22. Shader的变体：另外一个指令 shader\_feature 也可以设置预编译宏，与 multi\_compile 的区别是 shader\_feature 不会将没有被使用到的Shader(变体)打包进包内，因此 shader\_feature 更适合材质球的关键字指定预编译内容，因为Unity3D只生成和编译被使用的预编译情况，而 multi\_compile 更适合全局Shader指定关键字，因为它会把所有组合都编译一遍，无论有没有用到
23. Projector应用：

平面阴影 ：不依赖地形，自身投影到想要的任何平面，将其渲染出来

贴花decal：利用深度信息计算图片投影：相当于动态贴花，可以移动动态更新贴花效果

* **游戏AI**

1. 状态机：用来处理简单的状态转变比较方便，逻辑的处理放在各个状态内部
2. 行为树：composite node复合节点、decorator node装饰节点、condition node条件节点、action node行为节点，条件和行为节点作为叶子节点，也被称为behaviour node表现节点，组合和装饰节点成为决策节点
3. 非典型性AI：庄家式AI（赌博式先甜后苦始终骗你钱）、可演算式AI（结果唯一）、博弈式AI（搜索所有可能的下一步，计算更多步）

* **网络层**

1. 如果是由客户端间歇性的发起无状态的查询，并且偶尔发生延迟是可以容忍，那么使用HTTP/HTTPS吧。

如果客户端和服务器都可以独立发包，但是偶尔发生延迟可以容忍（比如：在线的纸牌游戏，许多MMO类的游戏），那么使用TCP长连接吧。

如果客户端和服务器都可以独立发包，而且无法忍受延迟（比如：大多数的多人动作类游戏，以及少部分MMO类游戏），那么使用UDP吧。

1. http面向连接，三次握手开启，四次挥手断开
2. Tcp数据的接收和处理会存在内存的争夺，要使用lock，数据接收要缓冲队列一个接收，一个处理，使用双缓冲效率更高，在数据交换时lock
3. Tcp数据的发送确定好数据的协议，比如使用json还是protocol buffer，接收队列中进行压缩处理，确定数据协议关键点：那种格式，数据包体的最小化，有校验能力（md5校验，奇偶校验，crc循环冗余校验），加密（异或，密钥，RSA，非对称加密（双端密钥））
4. Tcp断线检测，本身是有断线检测，但是不可靠，建立心跳检测协议，客户端每隔一段时间发送心跳包，客户端发送心跳回应包，当检测到一段时间内没有收到数据，表示连接断开

* **寻路优化**

1. A星寻路采用贪婪算法获取路径，但不是最短路径，时间复杂度nlogn
2. 选取附近点，计算到目标点的最短距离，依次操作，具体算法中会存在排序
3. 长路径的优化方案，可以选取一些特定的点，离线计算好寻路路径，然后在离线数据中找到最近的起点终点，然后再动态导航到选择的起点终点
4. 排序算法优化，使用最小堆（待学习）
5. 可以使用int值记录点是否被当前寻路使用过，及每次寻路+1，和寻路次数比较相等，表示使用过，省去了在每次寻路前，对所有节点的状态初始化