//网络

//内存

//平台

//效率

工具层 （编辑）

功能层 （逻辑 输入 相机 角色控制 动画 物理 渲染 网络 IO 内存管理 其他）

资源层

核心层 （创建线程 内存分配 容器创建 数学模块）

平台层

只能上调下，不能下跨上

工具：

1. 蓝图，材质编辑器
2. DCC Digital Content Creation数字内存创造物 的 导入导出工具器

功能：

1. 功能顺序 + tick
2. 多线程

资源：

1. 需要转换成最佳的格式 （图片：DTS GPU纹理）
2. 资源互相的关系的文件 + GUID资源唯一号
3. 资产管理器Handle系统（返回你要的东西）
4. 资源内存生命周期 + 延迟加载

核心：

1. 数学效率
2. 数据结构（防止内存碎片 ， STL 不太行）
3. 内存管理 （pool，数据集中，顺序读取，批量增删）

平台：

1. 文件系统
2. 图形库（虚函数分库）
3. 提供的新硬件设备

初始化和回收：

1. 问题：默认构造析构导致执行次序不可预测
2. 方法1：静态单例化管理器，管理器内嵌到小管理器，并且不使用构造和析构函数，独立启动和终止函数 （影响开闭原则）
3. 方法2：全局优先级队列
4. 方法3：构建依赖图，自动计算优先级，自动按顺序初始化

内存管理：

1. 问题：malloc/free & new/delete 在内存随机分配，管理非常慢（用户到内核的上下文切换）
2. 法则：少堆分配，不在紧凑循环中堆分配
3. 方法1：自制栈分配器（或双端），安排一个指针指栈顶 （游戏关卡）
4. 方法2：池分配器，以元素为单位申请N倍空间，用链表包含所有元素，分配时拆断下一个元素，回收时插入，直接用地址（偏移）作为指针 （链表只有两个元素，后面元素就是一大片内存）
5. 方法3：对齐的分配器，总是2的幂为单位，使用掩码（单位-1），直接和内存AND得到余多少，最后加上差的部分（单位-余多少）。 这个差的值可以用多分配的内存来记录
6. 方法4：单帧分配器，用方法1，但每一帧自动全部清除
7. 方法5：双缓冲分配器，两个单帧交错，安心使用结果

内存碎片：

1. 问题：多次分配&释放后，内存不连续，分配失败
2. 方法1：堆栈或池分配器
3. 方法2：碎片整理，将已分配内存逐一往低位移动，并更新指针（或使用Handle或smart ptr）
   1. Handle：是句柄表的索引，句柄表的每个元素是一个指针 ？
   2. Smart ptr：智能指针是类， ？
   3. 运用某些库时内存不能被定位，要独立出来不整理
   4. 限制每帧可整理次数，分摊成本

自定义容器：

1. 可以完全掌控，优化，定制，无第三方依赖，自制并发结构
2. STL慢，内存大，多动态分配，游戏机编译有差异 （少量内存才用 ）
3. Boost lib太多，许可问题
4. STL port 方便各种编译器
5. 字典，二叉树，用键值排序（二分法搜索）
6. 散列表，键值按mod运算存储在表中（有可能碰撞）
   1. 开放散列表，碰撞时存在同一个位置
   2. 闭合散列表，碰撞时用规则算法探查下一个空位，大小最好为质数
   3. 探查法，线性探查（找下一个，或前后交替找），二次探查（前后交替）
   4. 散列法，把键转换为整数，再mod运算得到索引（越好越少碰撞）
   5. 字符串散列法，LOOKUP3,CRC32,MD5等

字符串处理：

1. 问题：动态分配，本地化
2. 字符串类可能产生多次拷贝等开销，（建议少用，引用传递）
3. 唯一标识符
4. 用散列表储存字符串集合
5. 本地化，字符编码，数据库管理所有字符

引擎配置：

1. 选项的保存，可配置性，每个开发者自己的设置

I/O系统：

1. C的I/O API : fopen()为有缓冲功能，open()则为无缓冲功能
2. 封装I/O API，可以多平台，简化调用，衍生功能
3. C的I/O都是同步的，程序必须等待函数返回
4. asyncOpen() asyncReadFile() asyncWait() 为异步，可以加入回调函数响应
5. 异步处理要注意优先权，先读取优先权高的

资源管理：

1. 资源调节管道，转换为引擎格式，调节参数的记录
2. 资源数据库，增删改查，磁盘移动，交叉引用，保存历史版本，多样化搜索联想
3. 三步： exporter原始数据读入引擎，compiler压缩改造，linker依赖资源的互相链接
4. 运行时：生命周期，处理复合资源，交叉引用，内存安排，
5. 目录组织，多路径导致磁盘需要寻道，合并成单一文件（免费的ZIP）更高效
6. 唯一标识符GUID
7. 注册表（字典），保证内存中无重复资源
8. 生命周期，永久，关卡期间，用完就丢（小短片），串流（背景音乐）
   1. 引用计数，先下一个关卡的资源引用加1，再上一个关卡资源引用减1，再导入卸载资源
9. 内存管理，堆分配造成碎片，栈分配限制，池分配需要分割成组块并冗余，
   1. 自由大小组块分配器：链表管理未用组块，组块内带栈分配器（一般用于同生命周期的资源 关卡，使得组块内也可以继续分配空间），！释放还是组块整体释放！
10. 复合资源，保证交叉引用
    1. 方法1，指针只能用于运行时
    2. 方法2，内部引用时，GUID表，保存引用，查找获取指针
    3. 方法3，内部引用时，指针修正表，序列化时内存不连续但相互引用的对象保存到文件时变得连续，文件内保存一个表，储存每个对象的起始偏移值，文件内对象也保存引用对象的该偏移值。P302
    4. 方法4，外部引用时，用GUID表或指针修正表同时，需保存对方的路径，文件内建立交叉引用表，导入时扫描该表。 内存也有主查表记录和获取已有的引用资源。
11. 导入后初始化，C需要一个表对应到初始函数，C++则（类 多态 虚函数 工厂等）

动态数组：

1. C的静态数组很友好
2. 动态数组，需要更多时分配更大的空间再复制过去 （效能慢，旧位置的内存碎片）
3. 外露式链表，节点含指针指向元素 每次都是新分配的节点（适合使用池分配器分配节点）
   1. Template<typename t>

Struct Link

{

Link <t>\* m\_pPrev;

Link <t>\* m\_pNext;

t\* m\_pT;

}

1. 侵入式链表，节点嵌在元素中，无需动态分配节点，或者从节点派生出元素类（但不能像指针一样被共享在多个链表中）
2. 循环链表，简化插入删除逻辑

Modern C++ Design

游戏循环：

1. 消息泵，先处理来自windows的消息，再处理游戏循环
2. 回调驱动框架，lib要求完善回调函数或衍生类的虚函数
3. 事件更新机制，一个队列保存收到的事件，再在周期中按位置处理事件

时间线：

1. 使用上一帧的时间差来作为下一帧的执行参数有可能导致物理系统步进多次花更多时间
2. 方法1：使用连续几帧的平均时间差
3. 方法2：锁帧率，可以使物理系统使用最佳时间差
4. 多线程获取的时间会不同
5. 当暂停等，帧时间过长时，设置最大帧时间值1/60s等

并行：

1. 方法1：SIMD指令，多个数据同时运算
2. 方法2：分叉/汇合fork/join，
3. 方法3：引擎子系统，主系统控制和同步，子系统执行有隔离性的功能（渲染，物理，动画，音频）
4. 方法4：任务模型，任务（数据和代码组）队列，多个处理器取出处理。
   1. 异步，提交后过段时间等结果继续执行（可提前一帧提交任务）

网络架构：

1. client-server model 客户端输入/渲染，服务器逻辑
2. Peer to peer 每个机器只有特定对象的管辖权

世界编辑器：

1. 组块的构建和管理 定义世界组块，填入动态和静态元素
2. 可视化编辑 （自制渲染引擎，作为DCC工具插件，直接在游戏内调）
3. 导航功能（编辑时摄像机的快捷移动）
4. 选取功能
5. 图层功能
6. 对象的属性操作
7. 辅助（透视，安放，对齐辅助）
8. 特殊对象类型（粒子，灯，线条，声源，体积，导航网格）
9. 加载和保存功能
10. 资产管理工具

游戏性基础系统：

1. 在世界安放游戏对象
2. 关卡管理和串流（大世界加载和预加载）
3. 更新实时对象 / 动态产生和销毁
4. 消息和事件处理
5. 脚本
6. 目标和游戏流程管理
7. 对象能够按需获得底层引擎服务（粒子，物理，等等）
8. 随时间变化
9. 新增游戏对象
10. 唯一标识符
11. 对象的查询和引用
12. 状态机
13. 网络同步
14. 存档功能 （对象状态的存储，运行时类型识别RTTI/反射：获得类及其属性和方法，/抽象构造：直接先构建实例再从硬盘将数据序列化到内存）

1.工具方 2.运行时

以对象为中心：UE4

1. 所有物体都有具体对应的类型。对象中含有各种属性
2. 深且宽的继承关系 （虚函数的定义容易混淆 ，类型分类的标准写死 ）

多重继承，致命钻石：一个类型最多一个有祖父的类，其他都是独立的类（mix-in类），

改善1：冒泡效应，将新功能往基类送，导致基类越来越大

改善2：is-a合成为一个，新类型作为类成员，必须同时构造和销毁

改善3：has-a有一个，将类的功能全部拆分为独立类型（组件），再拥有他们

改善4：纯组件模型，没有根类型，组件上拥有实例化的id，用id将组件联系在一起（通信困难）

以属性为中心：（关系数据库的一列，类似纯组件模型）

1. 属性类含有表，记录各个对象的该属性的值
2. 属性间互相通信实现行为 / 专属脚本控制对象
3. 优点:1更好的内存管理，2轻松定义游戏对象（对象都是在属性表加一个id的事）
4. 缺点:1 属性关系和行为难维持实现 ，2 debug难

储存格式：

1. 二进制对象映像：需要处理指针和虚函数表
2. 序列化：1.序列化只导出可重建的数据 2.反序列化时使用数据重建
   1. 格式：XML/自定义二进制/JSON
   2. 基类带 serialize\_out() serialize\_in() 或 反射系统 通用序列化
   3. C++反射系统：
      1. 宏或模板抽取反射数据（或直接编写结构体），重载虚函数返回反射数据
      2. 利用硬编码的数据表，字符串映射到实例化函数， 工厂构建
3. 生成器：在世界组块文件内简单描述对象（是什么，位置旋转缩放，等）
   1. 抽象对象则以Schema 类型文本 保存内部的属性名称类型和初始值，甚至存在继承关系。在世界编辑器中表现这些属性出来供设计者调整。
   2. 游戏程序再另外实现该对象的构建和行为

游戏加载：读取资产+管理内存

1. 关卡加载：一片内存，堆栈分配器，基础资源LSR压箱底关卡资源加载一个释放一个 （世界不能无缝）
2. 阻隔室：一大内存为关卡，一小内存为关卡过渡区，异步卸载旧关卡加载新关卡，而玩家在阻隔室继续打打杀杀
3. 串流：三片内存三个区域ABC，看不到A时释放加载D，看不到B时释放加载E
   1. 或者组块分配器，将资产切割为相同大小的数据块，按需加载卸载 （可以动态化分配内存空间）
   2. 判断：每个组块为凸体积，并包含一个表表明内存该有哪些组块，求并集加载和卸载组块

对象的内存管理：

1. 载入组块后立即生成所有动态资源，随后不得创建销毁 （但要求对象多了休眠隐形的状态，并且无法无限制构建对象）
2. 动态管理
   1. 为每个对象类型安排一个池分配器
   2. 多个倍数大小的池分配器
   3. 内存重定位

世界组块文件：

1. 静态元素的完整描述（特殊位置，背景，碰撞，线条，等）
2. 动态对象的初始信息

游戏存档：

1. 动态游戏对象的状态信息（覆盖世界组块文件内的初始信息）
2. 存储点 / 随处可存

对象引用：

1. 指针，（忘记释放，忘设空，设了无效地址）
2. 智能指针
   1. 需要重载\* ->解引用 + - ++ -- 算术运算
   2. 能在想释放时获取引用计数std::shared\_ptr<T>::use\_count
   3. 不要自己开发智能指针 ，直接使用Boost /STL
3. 句柄，（多一层控制指针的获取处理， 也方便全局查询）
   1. 全局句柄表（指针类型的数组）
   2. 每个对象有唯一标识符和句柄表的索引
   3. 对象构造内生成标识符和将自己保存到句柄表的空位中
   4. 对象句柄类（代替指针 供外部调用） 保存有同样的唯一标识符和索引（检查），有获取对象函数以返回对象的指针

对象查询：

1. Hash表/二叉树 以标识符为键 存储游戏对象指针或句柄
2. 预先以多个条件排序出多个链表
3. 光线投射，体积碰撞
4. 四叉树/八叉树/kd树 等空间邻近算法，以区域/半径搜索

实时更新游戏对象：

1. 最直接：
   1. 每个对象有update()
   2. 一个单例类带链表管理所有动态物体
   3. 每个update()将深入请求更新引擎子系统！！（业余了）
2. 批次更新
   1. 对象update()中只调整属性
   2. 子系统遍历对象属性真正更新（把共同的作业（渲染，动画等）组成一个批次更新）
   3. 子系统内存统一，少重复运算，少再分配，允许对象间和系统间的依赖计算
   4. 子系统间更新时，对象的数据也可能需要被多次更新
      1. 用多个链表管理每个更新阶段需要更新的对象（耦合稍强）
      2. 或者 回调！ 登记自己每次的更新函数，子系统执行前先遍历对应的表更新对象（当然这些表取一个删一个）
   5. 桶式更新，对象间有依赖时，将依赖关系排成多颗树，从根往下，一层一层往下更新这些树
      1. 注意对象依赖处理时，依赖对象的状态是用旧的还是新的，旧的应在树的高层，新的在低层。或者干脆不要修改对象的状态，保存到新的缓存中，保留新旧状态
3. 并行更新
   1. 对象间并行（对象只能使用消息机制传递互相信息/加锁/状态缓存）
4. 与并行的子系统对接
   1. 多以任务形式发布作业，而不是自己执行
   2. 不是循环遍历发布任务，而是一大批任务一起批发布（子系统也需开发批处理功能）
   3. 尽早请求任务， 决定等待时间

事件与消息泵：

1. 静态的后期绑定：直接调用对方的函数（要求对象都继承于一个基类，该基类还得有所有的可能事件处理！）
2. 动态的后期绑定：事件/消息封装为对象（类变量包含类型和参数）
   1. 事件可以继承，对象只需要一个处理事件的虚函数
   2. 事件可以一个发到多个对象上
   3. 对象可以转发事件
   4. 事件类型：
      1. 全局枚举（硬编码，没封装，不兼容旧存档）
      2. 字符串 /字符散列标识（名称冲突需要额外的系统管理检测）
   5. 事件参数：提供可能对接收者有用的信息
      1. 直接继承基类事件（硬编码）
      2. Variant泛类结构体（Union内存，内部含type标识类型）
         1. 事件包含泛类数组（要求了对象知道参数顺序）
         2. 或者用map散列表字符串对应值（键值对）
   6. 事件处理：
      1. 在对象中switch
      2. 需要强制转换参数类型
   7. 职责链：关系图，对象间的联系？事件处理返回bool 告诉还需不需要转发？
   8. 登记关注：减少多播广播
      1. 每个事件类型维护一个对象链表/每个对象维护一个关注事件的位数组
      2. 或查询机制（借子系统查询需要发送的对象 碰撞 爆炸范围等）
   9. 事件队列 ：对象中带有事件队列（某些也可以无视队列立即处理）
      1. 事件应在合理时机被处理
      2. 可以往未来投递事件（入队时需要指定送达时间）
      3. 同类型/时间的事件需要优先次序
      4. 事件系统更复杂，需要深复制事件至队列，需要管理内存，调试更难
      5. 但立即处理也可能耗尽调用堆栈
   10. 数据驱动事件传递系统（设计师diy事件）
       1. 界面挑挑挑选选选配置数据（还需要连线功能以触发对方的特定事件）
       2. 脚本编写

脚本：（1数据式:读入内存引擎使用 2运行时脚本:在引擎上下文执行）

1. 直译式，引擎带虚拟机VM，可快速看效果
2. 比如
   1. Quake雷神之锤
   2. UnrealScript UE类C++,可以派生，latent函数等待触发再执行，界面显示变量，标记后自动网络复制
   3. Lua 变量无类型 值有类型
   4. Python 鸭子类型，函数接口决定对象类型（无需继承），有这个函数就是一家人
   5. Pawn 状态机
3. 架构
   1. 回调脚本：引擎执行用户的回调函数
   2. 事件处理器脚本：同上
   3. 脚本扩充对象/类型：继承/组合聚合
   4. 组件/属性脚本：新组件 新属性
   5. 脚本驱动引擎：游戏对象由脚本编写，某些组件才调用引擎代码
   6. 脚本驱动游戏：引擎只是库 供脚本调用
4. 功能：（实现游戏性的功能）
   1. 原生函数的接口
      1. 双方通信
         1. 引擎按函数名称用虚拟机编译对应的字符码执行脚本函数（可能是遍历指令执行）
         2. 脚本语言可能是维护一个表来映射调用引擎的功能函数
   2. 脚本中对游戏对象的引用
      1. 字符串散列句柄映射对象指针，脚本中获取对象，当做参数调用引擎功能函数
   3. 脚本中接受处理事件 （引擎中对象可以获取脚本的事件处理函数）
   4. 脚本中发事件（事件以字符串命名，脚本间发送接收）
   5. 面向对象脚本：允许脚本有自己的参数和属性方法 （原生类可以被映射入脚本作为变量使用）
   6. 有限状态机：游戏对象有多个状态，对应各个更新和事件处理函数
   7. 多线程脚本：多个脚本同时执行 / 合作式多任务？（具备休息等待函数和唤醒函数）
   8. 游戏流程脚本：有限状态机，全局数据检测控制以及流程控制。

编译式语言：需要编译成对应硬件的机器码

直译式语言： 编译成字节码，无需再编译（VM的机器码，但VM需要提供对各种平台的兼容）

命令式语言：指令序列组成，每一个执行操作/读写 C++

声明式语言：只描述操作? HTML

函数式语言：无状态，函数到函数到函数，除结果外不改变系统（多线程） F#

过程式语言：程序/函数，改变内存

面向对象式语言：类包含程序/函数管理操作自己的内存

反射式语言：运行时可以获得数据的类型和详细的属性

收集下第三方工具库和各种引擎（DCC digital content creation数字内容创作）的特点

Blender：建模

Houdini：特效

Maya：离线动画

面试题：

超大型整数，数据结构如何设计？相加溢出怎么处理？

 forward+

纹理贴图的压缩算法

屏幕空间反射

使用过 Git 吗？如何排查 200 个 commit 中的 bug ？

模型面数太多 程序化生成

伽马矫正 伽马空间

 MicroGBuffer

虚拟纹理

有没有无锁线程同步方式

LRU页面置换算法

如何在编译期获取变量类型信息

变换矩阵如何将各分量进行分解

三维空间中光线和三角面片求交

光栅化步骤，世界坐标转到屏幕坐标的过程中需要插值，如何保证插值的正确性

神仙面试：

Q：compute shader 剔除结果需要回读到 cpu 吗？能直接绘制吗？

可以不直接回读，在 GPU 直接执行绘制命令

需要图形 API 达到一定版本才支持，比如 DX12 的 ExecuteIndirect

Q：说一说 GPU Driven 能实现哪些功能？具体怎么实现的？

遮挡剔除，先生成低分辨率深度图，然后将物体 bounding box 和深度图做对比，如果被挡住则不用绘制。

这里当时回答的时候没有说清楚，低分辨率深度图的获取主要分为两个流派，一种是 cpu 做软光栅化，另一种是回读上一帧的深度图

Q：移动端、PC 端 GPU 在显存、架构上有何不同？为什么移动端采用这种解决方案？

移动端不分内存显存，但是 cpu 和 gpu 无法交叉访存。渲染方面使用 tile base，每次渲染一小块屏幕区域。依次将每个 tile 的数据调入小容量、低功耗的高速存储器（Tile Memory）进行光栅化、着色等计算

Q：PBR 贴图格式需要注意什么？

albedo 可以 sRGB 格式，而 roughness 和 metallic 需要保存在线性空间？？？

Q：compute shader 使用时会有什么问题，需要注意什么？

执行某些算法时 cpu 需要向 gpu 提交大量数据，内存 -显存的数据迁移造成开销

注：这里没答好，感觉面试官想问线程同步、硬件特性、wrap 优化性能这方面的。因为每个 GPU SIMD 的特性，每个 wrap 内所有线程执行一样的指令，所以难以处理分支指令。一般是两边都走然后各取结果所以效率低

知道 HDR 吗？谈谈 HDR 和 Bloom 的关系 为何在 Bloom 时候要采样多级 mipmap？

普通像素的范围在 0~255，而 HDR 能够模拟物理世界中正确的光照，比如环境贴图中亮度上万的像素。使用 HDR 在物理上能够得出正确的颜色，但是屏幕显示不了那么亮，需要做一个色调映射再输出。

两种方法：用横竖两次高斯模糊、或者 down sample 出一串 mip 然后再 up sample 回屏幕

如果采样单级 mip，因为硬件 Bilinear 滤波是 2x2 的盒子，最终扩出的 bloom 结果会是方块形状的，很丑。叠加多级 mip 的结果，方块感没那么强

注：这里又没答好，用低分辨率的 mip 还有一个好处就是 bloom 扩散的范围会非常大。而高斯模糊的效果要想做到与前者同样大的扩散范围，需要更大的 filter size 也带来了更高的开销

////////////////////////////////////////////////////

SSAO:

在摄像机坐标系下利用z轴判断随机点和原点的前后遮挡关系

随机点需要转换到NDC投影中找对应视图的坐标数据

坐标系转换要注意齐次坐标缩放归一

1. 将坐标转换到摄像机局部坐标系，并需要缩放归一？

float4 viewpostion = mul(float4(gbufferd.xyz, 1.0), gView);

viewpostion.xyz /= viewpostion.w;

1. 利用噪声图构建随机方向，和该法线一起构建随机切线空间的转换矩阵

float3 normal = normalize(mul(gbufferb.xyz, gView));

float3 randomVector = NoiseMap.Sample(gsamLinearWrap, pIn.texcoord \* noiseScale).xyz;

float3 tangent = normalize(randomVector - normal \* dot(randomVector, normal));

float3 bitangent = cross(normal, tangent);

float3x3 transformMat = float3x3(tangent, bitangent, normal);

1. 利用随机方向数组获取切线空间的随机点

float3 samplePos = mul(kernelOffsets[i], transformMat);

samplePos = samplePos \* radius + centerDepthPos;

1. 转换到屏幕空间获取对应坐标，并转换到摄像机局部坐标系

float4 offset = mul(float4(samplePos, 1.0), gProj);

offset.xy /= offset.w;

float4 offsetposition = GBuffer[3].Sample(gsamPointClamp, float2(offset.x \* 0.5 + 0.5, -offset.y \* 0.5 + 0.5));

offsetposition = mul(float4(offsetposition.xyz, 1.0), gView);

offsetposition.xyz /= offsetposition.w;

float sampleDepth = offsetposition.z;

1. 判断遮挡和叠加

//距离 大于 半径 则趋近于0 表示未受遮挡

float rangeCheck = smoothstep(0.0, 1.0, radius / abs(centerDepthPos.z - sampleDepth));

//如果采样点在自己前面则不加，否则按照角度叠加遮挡

occlusion += rangeCheck \* step(sampleDepth, samplePos.z) \* nDotS;

}

//反向遮挡值

occlusion = 1.0 - (occlusion / kernelSize);

float finalOcclusion = pow(occlusion, power);

