# 腾讯光子客户端一面

1.一个数字转换为二进制，其中有多少个1

int count\_ones\_bitwise(int n) {

int count = 0;

unsigned int num = static\_cast<unsigned int>(n); *// 处理负数（转换为无符号数）*

while (num > 0) {

count += num & 1; *// 检查最低位是否为1*

num >>= 1; *// 右移一位*

}

return count;

}

使用位运算

2. 二维空间中三角形，怎么判断一个点是否在其内部？

计算向量AE与向量AB的叉乘，如果每条边计算出的结果的符号相同，那么则表明E在所有边的同一侧，即在三角形内。

# 腾讯天美客户端一面

**1.帧同步和状态同步**

帧同步，客户端上传操作，服务器只负责转发操作，具体的操作在客户端执行。战斗逻辑在本地，会产生外挂。在逻辑帧上传操作，不是渲染帧，可能是几个渲染帧才上传一个。

采用延迟锁步来实现确保操作同步，客户端的操作会延迟几帧才实际执行，在这个过程中会等待所有玩家的操作信息，如果出现丢包那么采用预测来执行如果预测错误那么回退。

状态同步，客户端上传实时操作，服务器计算操作并将结果返回客户端，客户端只是显示层。

回放系统难以实现，需要保存大量数据。且对服务器的压力非常大。

在服务器每隔一段时间就保存一次游戏的完整状态，如果已经出现了不同步的，那么回退到服务器保存的快照重新执行玩家的操作。

[【网络同步】浅析帧同步和状态同步 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/357973435)

<https://zhuanlan.zhihu.com/p/357973435>

**2.map和hash\_map**

map底层使用红黑树o(logn)的时间复杂度，hash\_map使用hash表，查找的时间复杂度为O

0(1)。

红黑树是平衡条件为最长路径不超过最短路径两倍的平衡树，而AVL树是子树高度差不超过2的平衡树。

红黑树使用着色来维持平衡条件，而AVL使用深度来维持。

红黑树最关键的两个条件是红节点不能与红节点相连，根节点到叶节点的不同路径上黑节点的数目相同。

[红黑树详解-CSDN博客](https://blog.csdn.net/hlzs_01/article/details/147254146)

<https://blog.csdn.net/hlzs_01/article/details/147254146>

**3.CPP的编译链接**

编译：预编译，编译，汇编

预编译：执行预处理指令，将引用的头文件完整拷贝到代码中，并将宏定义进行替换。

编译：将预编译产生的代码编译为汇编代码。

汇编：将编译产生的文件汇编为二进制机器语言。

链接：将每一个符号表相对应的符号进行连接。包括静态连接和动态连接。

头文件：只包含函数原型和类的定义，有部分内联函数和模版函数的实现。便于通过编译并为后续的连接提供需要的符号。

库文件：包括函数的实现的具体代码。

[【C语言/C++】C/C++程序的编译+链接\_c语言怎么连接-CSDN博客](https://blog.csdn.net/m0_71580879/article/details/143488527)

<https://blog.csdn.net/m0_71580879/article/details/143488527>

**4. 头文件的函数在cpp中没有实现会报错吗**

不会，如果没有实现那么会在连接阶段去库文件中查找并解析。

**5. 协程及其底层**

协程本质是一个轻量级的用户态线程，由程序进行管理，可以用来实现异步的操作。

C#的协程的底层是一个简单的状态机。C#调用StartCoroutine方法，将IEnumerator对象传入C++层，通过反射来找到对象的moveNext和current方法，然后创建一个Coroutine对象，将这两个方法交给该对象，随后执行Run方法调用moveNext方法一次执行到yield时将对象保存到阻塞链表中，在每一帧都会检测一次阻塞链表中是否有满足条件的coroutine对象，有的话就执行它的Run函数，等待Movenext函数返回false时协程结束。

[Unity/C++/C# 多线程与协程(Coroutine)原理与用法详解\_c# coroutine-CSDN博客](https://blog.csdn.net/m0_73952999/article/details/145962680?utm_medium=distribute.pc_relevant.none-task-blog-2~default~baidujs_baidulandingword~default-8-145962680-blog-116240688.235%5ev43%5epc_blog_bottom_relevance_base2&spm=1001.2101.3001.4242.5&utm_relevant_index=11)

[https://blog.csdn.net/m0\_73952999/article/details/145962680?utm\_medium=distribute.pc\_relevant.none-task-blog-2~default~baidujs\_baidulandingword~default-8-145962680-blog-116240688.235^v43^pc\_blog\_bottom\_relevance\_base2&spm=1001.2101.3001.4242.5&utm\_relevant\_index=11](https://blog.csdn.net/m0_73952999/article/details/145962680?utm_medium=distribute.pc_relevant.none-task-blog-2~default~baidujs_baidulandingword~default-8-145962680-blog-116240688.235%5ev43%5epc_blog_bottom_relevance_base2&spm=1001.2101.3001.4242.5&utm_relevant_index=11)

[反编译C#代码来看看协程是什么 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/35195150)

<https://zhuanlan.zhihu.com/p/35195150>

**6. 移动的实现，用了插值吗**

移动的实现有两种逻辑，增加速度和直接改变位置

直接改变位置：

每帧直接在原本的位置的基础上增加位移

使用Lerp函数，规定移动的总时间和移动的末位置

void Update()

{ //update每帧更新(调用)

time += Time.deltaTime;

//计算当前到开始的时间，每调用一次update就加一次两帧间的时间

t = Mathf.Clamp01(time / totalTime);

//每次调用update重新计算当前的比例系数,clamp钳制到【0，1】区间，相当于映射到【0，1】

transform.position = Vector3.Lerp(start, target, t);

//Lerp的A,B值需要是固定的，t系数值要动态跟随当前时间比例变化，即可实现匀速运动

}

使用Slerp函数，实现弧线运动

void Update()

{

time += Time.deltaTime; //计算当前到开始的时间，每调用一次update就加一次两帧间的时间

t = Mathf.Clamp01(time / totalTime); //钳制到[0，1]

transform.position=Vector3.Slerp(startPos, targetPos, t);

}

上面的两种函数的系数t都是当前的比例以移动的时间比上总的时间就可以实现匀速的运动

通过刚体来设置速度：

rigibody.velocity=new Vector3(speed.x,speed.y,speed.z)

直接在每一帧中设置游戏物体的速度来实现移动

跳跃之类的瞬间速度是rig.addforce()

[unity 实现移动\_unity移动脚本-CSDN博客](https://blog.csdn.net/liangqianjin_1/article/details/145398473)

<https://blog.csdn.net/liangqianjin_1/article/details/145398473>

**7.渲染管线**

应用阶段：传递渲染需要的数据并发出DrawCall指令让GPU进行计算

几何阶段：

顶点着色器：发生MVP矩阵变换以及通过齐次除法得到NDC坐标

裁剪：将不会被摄像机渲染到的点剔除

屏幕映射：将顶点NDC映射到屏幕坐标

光栅化阶段：

三角形设置：将几何阶段得到的点连接三角形

三角形遍历：哪些像素被三角形覆盖，被覆盖的三角形生成一个片元

片元着色器：根据片元的数据计算片元实际的颜色

逐片元操作：通过透明度，深度等一系列测试计算像素的实际颜色。

**8.TCP UDP**

UDP无连接，只需要知道目标的IP和端口号就可以发送。不可靠，没有确认机制，无法得知是否成功传输。

TCP有连接，即使发送失败也会通知进程，有序，流量控制。三次握手，两次只能确定一方的发送接受功能正常，四次挥手。

TCP延迟大于UDP的原因不是在于三次握手的时间开销而是源自于其可靠机制在丢包时的处理会导致后续所有数据发生等待。比如每一帧都接收信息发生丢包后TCP会导致后续几帧都出现延迟但是UDP只会在这个丢包的帧出现问题。

[TCP与UDP协议详解：可靠性、报文格式与区别-CSDN博客](https://blog.csdn.net/qq15035899256/article/details/126073927?ops_request_misc=%257B%2522request%255Fid%2522%253A%252278ed2c9c6869ed8f8f85f6da75c22738%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=78ed2c9c6869ed8f8f85f6da75c22738&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~top_positive~default-1-126073927-null-null.142%5ev102%5epc_search_result_base6&utm_term=TCPUDP&spm=1018.2226.3001.4187)

[https://blog.csdn.net/qq15035899256/article/details/126073927?ops\_request\_misc=%257B%2522request%255Fid%2522%253A%252278ed2c9c6869ed8f8f85f6da75c22738%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request\_id=78ed2c9c6869ed8f8f85f6da75c22738&biz\_id=0&utm\_medium=distribute.pc\_search\_result.none-task-blog-2~all~top\_positive~default-1-126073927-null-null.142^v102^pc\_search\_result\_base6&utm\_term=TCPUDP&spm=1018.2226.3001.4187](https://blog.csdn.net/qq15035899256/article/details/126073927?ops_request_misc=%257B%2522request%255Fid%2522%253A%252278ed2c9c6869ed8f8f85f6da75c22738%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=78ed2c9c6869ed8f8f85f6da75c22738&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~top_positive~default-1-126073927-null-null.142%5ev102%5epc_search_result_base6&utm_term=TCPUDP&spm=1018.2226.3001.4187)

**9.gc了解吗**

标记清除压缩

**10.HTTP和HTTPS协议**

HTTP协议是明文，可以被截取破解。

HTTPS协议在HTTP的基础上增加了加密和身份验证，加密是使用的密钥，对称密钥和非对称密钥。身份验证是验证服务器的数字证书。

[Http 和 Https的区别（图文详解） - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/701521059)

<https://zhuanlan.zhihu.com/p/701521059>

# 腾讯魔方客户端一面

**1.C#和C++中结构体有什么不同？**

C++的类和结构体没有本质的区别，只是在默认访问和继承属性上有区别，类的继承属性和默认访问都是private的。他们主要的区别还是在于用法，结构体用来封装数据，类用来实现复杂的逻辑。

[C++ 中类（class）和结构体（struct）的区别\_c++结构体和类的区别-CSDN博客](https://blog.csdn.net/chenai886/article/details/144132525)

<https://blog.csdn.net/chenai886/article/details/144132525>

C#的结构体进一步发扬了结构体作为轻量级的类的作用，将结构体作为值类型在栈上分配空间，而类的对象在堆上分配空间。

C++的类的对象是值类型，在赋值时进行拷贝赋值。

C#的对象是引用类型，在赋值时进行地址的传递。

C++相较于C#更加底层，所以它的效率更高但是作为一个工具来说C++没有C#现代。

C++开放了更多的功能给程序员，C++认为程序员不会犯错，而C#认为程序员会犯错。

具体来说，C++将内存的管理完全交给程序员，程序员的不当操作会出现内存泄漏，而C#有自带的GC（标记回收压缩）大大降低了内存泄漏的风险。

同时，这两个语言的运行也有较大的不同，C++直接编译为机器语言在本地运行效率更高，而C#需要编译为中间的IL语言在虚拟机上进行运行。

**2.C++11的常见新特性有哪些?**

Atuo关键字，智能指针，Lambda表达式

shared\_ptr的引用计数是线程安全的但是对于shared\_ptr对象本身的修改不是线程安全的，同时对于shared\_ptr引用对象的修改也不是线程安全的， shared\_ptr不会对被引用的对象本身提供任何保障。

Lambda表达式在底层中会被编译器转化为一个重载（）运算符的类对象，捕获列表作为构造函数的参数，参数作为重载函数的参数。



**3.外部排序（归并排序）**

假设内存中能够容纳三个块，那么将两个块作为输入缓存的块，一个块作为输出缓存的块，并进行归并排序，将结果写回硬盘形成由两个块构成的一个归并段，这个段是有序的，随后对所有块进行该操作，得到有序的归并段，随后对两个归并段进行归并排序，先将两个段中分别的低地址的块调入内存进行归并排序，再将高地址的块调入内存进行归并排序，并对后面所有归并段重复该操作，然后文件的每4个块都是有序的，再不断重复上面的过程就得到了有序的文件。

[数据结构（十六）----外部排序-CSDN博客](https://blog.csdn.net/weixin_69884785/article/details/138917062)

<https://blog.csdn.net/weixin_69884785/article/details/138917062>

# 腾讯天美游戏客户端一面

**1.Int类型和long类型的区别**

他们都是用于表示整数类型，只是在内存中占用的大小不同，int占用4个字节而long在不同的系统中长度不同，所以在有跨平台的场合不建议使用该数据类型。

long在windows始终下为4字节，但是在64位的linux和macOS系统中是8字节，在32位时又是4个字节。

**2.Const的用法，修改const的 变量，const的全局常量存放在哪里？**

const表示该值不可以重新赋值，如果没有被取地址那么会被放入符号表，如果被取地址了，那么会给其分配一个内存空间，如果是全局的变量那么会存储到静态存储区，如果是局部的变量那么会被存储到存放到栈区。如果是类成员，那么存储的位置跟随对象，类成员。

[C++内存划分&const常量存储位置\_const存储是栈还是堆-CSDN博客](https://blog.csdn.net/weixin_61256992/article/details/146295634)

<https://blog.csdn.net/weixin_61256992/article/details/146295634>

**3.** **引用类型是否可以做函数的返回类型，使用的时候有哪些需要注意的地方**

可以，最需要注意的点在于返回的被引用的变量的生命周期需要足够长，不然会导致未定义的行为。

对于返回值为临时对象时会有优化，直接将临时对象放在在函数外构造，然后作为参数传递，避免在返回过程中产生的临时对象。这个优化的效果是和右值引用相似的。

**4. 右值引用和移动构造，移动构造函数里面步骤**

移动语义的作用是面对一些即将被销毁的对象不是复制他而是窃取他的资源。右值引用是为了实现移动语义。

移动构造函数和赋值构造函数在函数内部是相同的都是通过传入的参数初始化当前对象，只是在传入参数时有所区别，移动构造函数传参使用右值引用直接剥夺原有资源的所有权，而拷贝构造函数使用左值引用不会剥夺原有资源的所有权。

[【C++ | 移动构造函数】C++11的 移动构造函数 详解及例子代码-CSDN博客](https://blog.csdn.net/wkd_007/article/details/139633287)

<https://blog.csdn.net/wkd_007/article/details/139633287>

**5.** **函数调用栈，栈上面有那些内容。函数的参数入栈顺序是怎么样的，函数的返回值怎么转递**

函数的栈帧就是函数调用过程中在程序的调用栈所开辟的空间，保存函数的参数，函数的返回值，临时变量，以及上下文信息。将参数从左至右压入栈中，函数的返回值一般是通过寄存器的方式来进行传递，但是如果返回值较大则会在调用者的帧栈中单独开辟一块用来保存返回值的空间并将这块空间的地址传递。

栈的速度比堆快，因为栈是在内存空间连续分配，并且以堆栈的形式进行管理，分配与回收只需要进行指针的移动。

堆在内存空间中是非连续分配的，涉及内存查找和碎片整理所以速度较慢。

[C语言：函数栈帧-CSDN博客](https://blog.csdn.net/2401_88251472/article/details/147362419)

<https://blog.csdn.net/2401_88251472/article/details/147362419>

**6.** **析构函数为什么一般定义为虚函数，如果子类只定义一些基础的类型变量，是否会内存泄漏**

在实现运行时多态的过程中，如果未将析构函数定义为虚函数，那么会执行父类的析构函数来释放对象的内存，这样会导致子类增加的内存空间无法被释放造成内存溢出。基础类型的变量不会造成内存泄露，因为其被存放在栈空间上由计算机进行管理。只有当子类使用new或者malloc来分配堆空间时才有可能造成内存的泄露。

**7. 构造函数里面可不可以调用虚函数，如果在构造函数里面调用了虚函数会有什么情况**

可以，只是vptr此时仍指向父类的虚函数表，所以会调用父类的函数导致多态失效。

**8. New操作符除了构造对象还有其他用法吗，new操作符是否可以重载，有什么用途。**

还可以构造数组以及在预先分配的内存空间上构造对象。

可以重载，以实现自定义的内存管理，new操作符本身也就是对malloc的封装，这个封装的过程由我们自己来实现也是可行的。

**9. Static\_cast和dynamic\_cast的区别，转换失败会发生怎么情况**

Static\_cast在编译时进行检查，适用于基本类型的转换和向上转型，可以进行向下转型但是必须保证是正确的，因为其不会进行运行时检查，其效率高于dynamic\_cast但是无法保证类型是绝对安全的。

dynamic\_cast在运行时的进行类型的检查，主要用于向上转型。

static\_cast是在编译时进行检查所以在转换失败时会报出编译错误。而对于dynamic\_cast而言，其在运行时进行检查，在指针转换时转换失败会返回一个空指针，在引用转换失败时会抛出异常。

也就是说static\_cast是在确认安全的时候使用，而dynamic\_cast是在不确定安全的时候使用。

**10.** **ClassA和ClassB没有任何关联，强行转换可以吗，加一些定义就可以转换？比如加一些特殊的构造函数，或者重载一些特殊的运算符。**

直接使用static\_cast和dynamic\_cast进行强制转换是不可行的。有两种常见的实现无关联类的转换，可以自定义一个函数纯逻辑性的进行转换或者重载类型转换运算符。

class ClassA {

public:

int value;

operator ClassB() const {

ClassB b;

b.value = value;

return b;

}

};

ClassA a{42};

ClassB b = a;

**11.** **MAP和unordered\_map关联和时间复杂度**

Map和unordered\_map在实现上实际没有关联，关联可能在于他们都是使用键值对的形式进行的存储，Map底层使用红黑树，其增删改查的时间复杂度是O(logn)，而unordered\_map底层使用Hash表，时间复杂度是O(1)

**12.** **Vector里面的push\_back和empalce\_back的区别**

Push\_back()是将已经构造完成的对象加入容器中，而empalce\_back是在容器的内部进行对象的构造，减少了一次拷贝，效率更高尤其在构造大型对象时更加明显。

**13. Shared\_ptr是不是线程安全的，避免循环引用**

Shared\_ptr本身是线程安全的但是不保证引用的对象的线程安全。对于Shared\_ptr出现的循环引用的问题，我们引入了weak\_ptr来解决。

Shared\_ptr会改变引用对象的生命周期，只有引用归零时生命周期才会结束。而对象销毁时会解除其中所有shared\_ptr的引用。

**14.** **编译链接的过程，链接库有有几种方法**

预编译：处理代码中的预处理指令

编译：将代码编译为汇编代码

汇编：将汇编代码汇编为机器代码

链接：将代码中的符号与符号表进行连接

有两种方式，静态链接和动态链接，静态链接直接将库中的代码复制打包到最后的可执行文件中生成独立的可执行文件；而动态链接会在可执行文件中保留符号，在运行时参照符号表动态链接库文件进行调用。

**15.在两个CPP中分别定义int A，能否通过**

不能，在链接阶段会报重复定义的错误。

链接阶段会去查询每一个.o文件产生的符号表并进行合并，在合并时会发现有两个int a，不知道应该采用哪一个于是出现报错。

可以使用Static将int a局限于本文件，或者使用inline来使两个定义都生效，但是如果这样定义必须完全相同不然违反单一定义原则还是会报错。

**链接阶段（Static Linking）**

* 链接器接收所有目标文件 .o 和静态库 .a。
* 读取并合并符号表，建立全局符号表。
* 检测重复定义和未定义符号错误。
* **符号解析**：未定义符号匹配到静态库或其他目标文件中的定义。
* **地址分配**：为所有代码和数据段分配虚拟地址，合并 .text、.data、.bss 等段。
* **重定位**：修改所有引用符号的机器码或数据地址，替换为分配后的真实地址。
* 从静态库中提取需要的目标文件 .o，其余忽略。
* 将所有代码和数据合并成一个整体。

**16.** **UDP进程用了一个80端口，TCP进程还能使用吗**

可以的，端口号是和协议相绑定的，不同的协议可以使用相同的端口号，但是相同的协议不能。

**17. 输入网址到加载网页内容的过程**

本地解析网址->本地DNS查询IP地址->三次握手建立TCP连接->发送HTTP连接->接收并解析HTML文件->加载子资源

**18.操作系统里面一个多线程的程序怎么进行同步，无锁编程有了解过吗**

目前主要使用的方法有锁和信号量。无锁编程实际上是使用硬件层面提供的原子操作指令来保证线程安全，没有循环等待的过程所以效率较高。

**19.系统调用有了解过吗？主要有什么作用？**

用户态进程使用内核功能的接口，是用户程序访问操作系统内核提供的服务的唯一方式。

线程操作，访问内存和磁盘等面向于硬件的操作我们自己写的程序实际上是没有权限执行的，所以需要借助系统调用来实现。

**20.多线程程序会有几个栈，系统调用里面会有栈切换**

只考虑用户态的话，多线程程序每一个线程都会有一个独立的栈来执行函数，所以线程的数量就是栈的数量，但是如果考虑内核态，那么每一个线程还有一个内核态的栈，也就是栈的数量是线程的两倍。使用系统调用会进入内核态，栈也会从线程的栈进入内核态的栈。

**21.单例模式**

懒汉式在使用时才进行创建，可能会出现多个线程同时使用所以存在线程不安全，而饿汉式在程序启动时就加载好，线程安全但是有一点点内存的浪费。

**22.工厂模式**

工厂模式实际上是对对象创建过程的封装，提供更加简单的对象创作方法，就像语法糖一样。

# 灵犀互娱 平平无奇工作室 游戏客户端开发

**1.C++多态有哪几种实现方式？**

多态有编译时多态和运行时多态两种，编译时多态的是模版，函数重载和运算符重载，而运行时多态是基于虚函数实现的。

**2.C++虚函数**

造函数不可以声明为虚函数，而析构函数则建议声明为虚函数以实现子类内存空间的释放，使用new生成的空间独立的一块空间不是对象的一部分，所以需要在析构函数中速手动释放。构造函数可以调用虚函数，但是由于虚指针没有被初始化仍指向父类的虚表此时会调用父类的虚函数。构造函数不能被声明为虚函数是因为空间没有被初始化，虚指针没有被正确赋值，虚函数执行时需要确定其类型但是构造函数执行时还未确定其类型。

**3.vector中插入元素会发生什么？插入类的元素时是否会调用构造函数？**

会先判断容量是否够，如果够的话插入元素时会导致插入位置后面的所有元素都进行后移导致效率较低，不够会发生整个vector位置的迁移然后再进行插入。插入对象时会调用，如果使用push\_back会在对象时调用一次，拷贝构造的时候调用一次，如果使用enplace\_back的话就只会调用一次。

**4.map元素是否有序**

有序map的底层是使用的红黑树来实现，而红黑树本质是平衡条件更加宽松的二叉平衡树所以他是有序的。

**5.哈希冲突是什么,如何解决哈希冲突**

哈希冲突指的是多个键值被映射到同一个下标，解决哈希冲突可以采用开放地址法或者拉链法。

**6.编译过程，为什么部分符号不会出现在最后的符号表，静态链接发生在哪一个阶段？**

预编译：执行代码中的预处理指令

编译：将代码编译为汇编代码

汇编：将汇编代码汇编为二进制机器语言

连接：解析各目标文件的符号表，解决未定义符号与其他目标文件或者库文件中已定义符号的对应关系，然后生成一张新的符号表，静态链接会将库文件中对应的代码拷贝进最后的可执行文件中生成独立的可执行文件，动态链接会在运行时根据符号表在库中动态加载

两种情况，常见的是使用static来主动避免符号进入全局符号表，static会采用内部链接，另一种情况是编译器优化，未使用的符号会被剔除。

Static的作用：1.让变量只进行一次初始化2.让变量属于类而非某一个具体的对象3.让变量变成内部链接无法被其他文件访问

**7.TCP和UDP**

TCP，稳定，3次握手4次回收，有确认机制，所以延迟较高

UDP，不稳定，不可靠，丢包不管，所以延迟低

**8.帧同步，帧同步要解决的问题**

帧同步，客户端本地计算操作，由服务器分发操作。如果出现丢包会带来较大的影响同时作弊也比状态同步更加容易，延迟较低。

检查到未收到包则等待一段时间，如果没有收到操作那么进行预测使用上一帧或者空操作来代替以让游戏正常进行，同时客户端补发操作，如果预测的操作与实际的操作不同那么进行回滚重新计算。

状态同步，客户端上传操作，由服务器进行计算后分发。服务器压力较大延迟较高。

**9.行为树相较于状态机的优点是什么？有哪些常见的节点？**

结构清晰，扩展性强，能够更加优雅的处理复杂的AI逻辑不会像状态机一样出现状态爆炸。

常见的节点有  
Selector：从左到右执行节点，执行成功时立刻返回

Sequence：从左到右执行节点，每一个都执行成功时才返回成功，任意节点执行失败时停止执行并返回失败

Condition：条件判断，不满足停止执行并立刻返回

Action：实际的逻辑节点，包含游戏物体的实际的运行逻辑

behaviorTree = new SelectorNode(new List<BTNode>

{

new SequenceNode(new List<BTNode>

{

new ConditionNode(IsPlayerInAttackRange),

new ActionNode(AttackPlayer)

}),

new SequenceNode(new List<BTNode>

{

new ConditionNode(IsPlayerVisible),

new ActionNode(MoveTowardsPlayer)

}),

new ActionNode(Patrol)

});

}

**10.MVP变换指的是什么？**

MVP指的是Model，View，Projection，分别是将模型从局部空间移动到世界空间，从世界空间移动摄像机空间，再从摄像机空间移动到裁剪空间

**11.如何实现人物描边**

外扩模型，采用两边渲染，第一次渲染剔除正面，将顶点按法线移动后将模型渲染成描边的颜色形成外壳，第二次剔除背面正常渲染。

**12.如何在100万个数据中找到字典序前1000的数据？**

采用大顶堆来实现，堆顶保存最大的数据，对每一个数据进行一次比较。假如每一次都需要进行插入那么时间的消耗为1，000，000\*log1000 = 10^7在可以接受的范围内用户基本难以感知到延迟

# 米哈游客户端一面

**1.说说什么是线程和进程，他们之间的区别是什么，线程之间的通讯和进程之间的通讯的区别是什么？**

进程是程序的一次运行的实例，操作系统分配资源的最小单位。

线程是程序执行的最小单位，多个线程共享进程的内存资源。

由于线程之间共享资源，所以其可以直接通过信号量和锁来实现，但是进程之间彼此独立所以必须使用操作系统提供的IPC来实现，包括管道和共享内存区之类的。

**2.什么是虚拟内存，主要解决什么问题？**

虚拟内存是一种内存管理技术可以让程序认为自己拥有完整而连续的内存空间，实际上这些内存被存放在物理内存和磁盘中。

主要解决了物理内存有限以及物理内存之间相互干扰的问题

**3.什么时候会出现缺页中断？出现缺页中断时**

当访问的内存块不存在于物理内存中时，常见的就是程序首次访问某块虚拟内存，访问的页已经被替换出了物理内存。

因为用户态实际上是不能进行对内存和磁盘的操作，所以出现缺页中断时会暂停程序随后进入内核态进行异常处理。在异常处理的过程中首先会进行现场以便于完成后继续执行原本的程序，随后判断访问是否合法，如果合法那么查找该页在磁盘中的位置，随后分配物理内存将内存页从磁盘中调如物理内存，然后更新页表恢复现场，程序继续执行。

**4.C++的编译步骤分为几步**

1.预编译：处理预编译指令

2.编译：将源代码编译为汇编代码

3.汇编：将汇编代码汇编为二进制机器语言

4.链接：解析目标文件的符号表，将文件中的未定义符号与其他目标文件或者库文件中的已定义符号建立对应关系。有两种链接，静态链接将代码直接拷贝进可执行文件中生成独立的可执行文件，动态链接在运行时根据符号表动态加载。

**5.STL用过吗？用过哪些?**

Vector，底层为动态数组，C++11之后实际上再大多数的情况下都建议使用vector来替代数组进行使用。

Set，map底层实现一样，为红黑树。

unordered\_set,unordered\_map底层实现都是哈希表。

**6.vector的resize()和reserve()的区别？**resize()是元素层面的修改，而reserve()是修改分配的空间，在增加的元素个数超过原本的容量时会触发reserve的内部来重新分配空间。如果给一个容器分配小于其实际需要空间的空间那么什么都不会发生。

**7.Map和Multimap的底层原理？红黑树的基本概念？红黑树有几个要注意的点？复杂度是多少？**

Map和MultiMap的底层都是使用红黑树，红黑树本质上是平衡条件更加宽松的AVL树，不像AVL要求子树高度差不超过2，只要求根节点到叶子结点的最长路径不超过最短路径的两倍。平衡条件更加宽松也就意味着其对调更加友好。首先根节点必须是黑色，然后红色节点不能接着红色节点，根节点到任意叶子节点路径上的黑色节点数必须相同。

增删改查的时间复杂度都是O(logn)。

**8.多态对你来说意味着什么？编译时多态和运行时多态举例?编程的时候什么时候会用到？**

多态意味着运行时多态和编译时多态，编译时多态有函数重载和模板，运行时多态有虚函数。实现多种敌人时就会使用到运行时多态来根据敌人的具体类型来调用不同的方法，而编译时多态通常用于实现框架性的内容，比如ECS框架中实现对各个组件的统一操作就会使用到模板。

**9.int \*/int&/int /const int 之间可以形成函数重载吗？是直接在编译时报错吗？**

Int\*/int&/int可以形成函数的重载，因为他们本质上是不同的类型，但是int/const int 不能形成函数的重载，他们都是int类型const只是一个修饰词并不会修改变量原本的类型。是直接报错，无法通过编译的语义分析。

**10.指针和引用的区别，初始化有区别吗，指针和引用的对象可以更改吗？**

一个保存对象的地址，一个直接就是对象，指针可以不用初始化但是引用必须进行初始化，指针指向的对象可以更改但是引用的对象不可以更改。

**11.什么是野指针？**

指向的空间不合法的指针。没有初始化或者原本指向的空间被回收。

**12.const int \* 和int\* const 的区别**

Int\* cosnt是指针本身无法修改，也就是只能始终指向一个对象。Cosnt int\* 是无法通过指针修改该对象

**13.智能指针中哪一个用的多一点？如果自己实现一个智能指针你会怎么去实现？除了将对象空间的分配放到构造和析构函数还有什么要注意的地方吗？**

Unique\_Ptr用得多一点，我会创建一个类将需要托管的对象的空间的管理放到对象的构造函数和析构函数中做。重载运算符防止出现二次释放同一资源。

**14.C++的虚函数了解吗？C++如何处理虚函数？虚函数表存的是什么内容？虚函数指针呢？虚函数表存在内存的哪一个位置？**

虚函数是C++中实现运行时多态的核心机制。当类中存在虚函数时，会生成一张虚函数表和一个虚函数指针，调用时通过虚指针查找虚函数表调用实际类型的指针。

虚函数指针保存虚函数表的地址。虚函数表中保存了该类所有虚函数的位置。纯虚函数

Virtual void func() = 0;其作用是强制子类实现该函数，否则子类也会变为无法实例化的抽象类。

**15.Static关键字的用法？**

1.修饰全局变量/函数时会将变量/函数的链接属性修改为内部链接限制作用域为当前文件。

2.修饰类的变量和函数时，修饰的内容会属于整个类而非对象。

3.修饰函数内的局部变量时会将变量存储到静态存储区生命周期延长到整个程序，只会被初始化一次。

**16.设计模式有了解过吗？单例模式实现过吗？如何确定全局只存在一个实例。**

在项目完成的过程中了解设计模式来进行代码结构层面的优化是不可避免的，最常使用的就是单例和观察者，最近实现过的是工厂模式。

单例模式有懒汉和恶汉的两种实现方式，

单例本身是依赖于类本身的，全局只会存在一个类所以也就保证了全局只存在一个实例。

**17.观察者模式有了解过吗？Unity自带的观察者模式有哪些？Unity如何实现观察者模式？**

采用通知而非轮询的方式来触发事件处理。

Unity中UI的eventSystem主体就是使用了观察者模式。

ImputModule监听鼠标等输入事件，通过RayCast来确定当前输入作用的UI元素，EventSystem调用UI元素上实现的事件对应的接口，随后执行具体的逻辑。

**18.C++的类型转换？无关类型是否可以进行转换？**

(int)这种C风格的做法，好用但是不够现代化。

C++更加现代化的写法是使用Static\_Cast()进行值类型和安全的向下转型，Dynamic\_Cast()进行不安全的向下转型，向上转型也可以但是不建议。

无关类型可以进行转换但是C++本身并没有提供相关的方法需要自己实现。单独定义一个函数或者重载类型转换运算符。

# 腾讯光子客户端二面

**1.类的对象大小由什么决定？**

类中非静态成员的大小加根据内存对齐机制多出来的空间。

内存对齐机制：每一个变量的地址偏移都需要是自身大小的整数倍，整个对象大小必须是对齐边界(整个结构体中最大的类型)的整数倍。

**2.虚继承**

避免菱形继承中重复继承基类导致的资源浪费的问题，用得不太多，继承两个基类的话一般使用组合，而且在C#中根本不会出现菱形继承。

**3.inline关键字**

Inline会告知编辑器此处展开函数而非调用，对于多次执行的函数可以优化性能。但是如果函数比较复杂也就是存在递归等内容那么编辑器可能会忽略inline直接调用函数。

# 腾讯光子一面

**1.MVP矩阵**

Model-View-Projection，将模型从模型空间移动到摄像机观察空间，再从摄像机观察空间移动到裁剪空间。

**2.Z-Buffer的作用**

记录深度缓存用于后续逐片元操作时计算遮盖关系。

**3.接口和抽象类的区别**

主要是功能层面的区别，接口相当于没有任何实际内容的规范而抽象类则是实现了部分的实际内容。

**4.透明物体为什么要从后往前渲染**

因为从后向前渲染后导致颜色的叠加错误，因为计算透明物体的渲染公式中需要知道背景的颜色，如果从前向后渲染会导致背景的颜色还不确定。

最终颜色 = 当前像素颜色 × 透明度 + 背后颜色 × (1 - 透明度)

# 游卡客户端一面

**1.XLua和C#的双向的交互？Lua调用C#的参数和返回值是如何传递的呢？C#获取到Lua中的string并修改那么会影响Lua中的string吗？原因是什么？Lua中的Table传递给C#那么C#是否可以修改这个Table吗这个时候Table会影响原本的Lua中的Table？Lua中的闭包指的是什么？**

XLua通过一个Lua虚拟机来实现C#与Lua的通讯，C#通过DOString来加载Lua的脚本，而Lua脚本中的全局变量和函数会被注册到Lua虚拟机中的全局表，这些注册到全局表的元素可以被C#通过键值直接进行访问。

而Lua访问C#可以直接通过命名空间来进行访问有两种访问的方式，一种是通过反射进行访问效率比较低，一种是通过桥接文件进行调用，效率远高于通过反射访问，但是需要提前生成。

参数从C#传递到Lua时会由XLua进行类型的转换，基本类型会直接映射，而List会映射到顺序表，Dictionary会映射到键值表，这一个映射的过程将会被XLua自动执行，而自定义复杂类型需要生成桥接文件才行。

参数和返回值的数据传递都是通过虚拟机的Lua栈来进行的。

Lua的String，Number，Boolean会被当做值类型传递到C#，所以C#只会得到一个复制的副本其修改无法影响Lua中的原始版本。而Table，Function，UserData会被视为引用类型进行传递，C#层修改的就是变量本身。

闭包指的是函数和其保存的运行环境，Lua的闭包机制可以延长外层函数的临时变量的生命周期，避免内层函数访问时出现未定义行为。

**2.C#委托指的是什么？那如果是一个异步调用的话应该是如何实现的呢？多线程之间的同步有哪些方式？那死锁是怎么发生的呢？**

本质是经过封装的函数指针。在多播委托中每一个方法仍然被封装为单播委托，多播委托的对象中有一个List保存所有单播委托。

异步调用主要的关键字是async声明方法为异步方法，await 等待异步方法的完成，Task封装异步方法，作用是让耗时的方法比如寻路和碰撞检测到其他线程去进行，在这个过程中最主要的关注点在于非Unity主线程外的线程都无法使用UnityMonobehaviour的生命周期的内容。

多线程共用资源所以可以直接通过变量或者数据结构来进行同步的操作。

死锁发生的原因是出现了循环等待，导致所有进程都无法执行。

死锁的条件是：

互斥：资源一次只能被一个线程占用

不可剥夺：资源在使用完成前无法被剥夺

请求与保持：在持有资源的时候又去申请其他资源

循环等待：存在资源获取的环状结构

1. **对象池一般是用在哪些地方，作用是什么？**

对象池的作用是避免频繁创建和销毁对象带来的开销，也是空间换时间的一种方法。

**4.之前有用过状态机吗？状态机最主要的几个接口是什么？对于状态突然中断的处理？**

有，状态机最主要的接口是进入，更新，退出。

**5.ECS指的是什么？他们的作用是什么？**

ECS指的是E：entity实体一个标识，C：Conponent组件保存实体相关的数据，S：System执行具体的逻辑操作。

ECS框架最主要的作用避免OOP中数据离散存储带来的读写开销, ECS 通过让组件数据连续存放，利用空间局部性降低查询开销；同时保证缓存中存放的都是同类数据，减少了缓存缺失率，提升了整体性能，在Unity中使用时还能避免对Monobehaviour的过度使用带来的性能开销。

**6.渲染的标准流水线？光栅化的过程？漫反射的数学模型？**

应用阶段：CPU传递渲染数据并发出DrawCall指令

几何阶段：

顶点着色器：计算顶点的光照UV坐标等信息，并将模型从世界空间移动到摄像机观察空间。

投影：将模型从摄像机观察空间移动到裁剪空间

裁剪：将模型不会被渲染的部分进行剔除，并记录深度缓存，得到归一化坐标

屏幕投影：将点映射到屏幕坐标

光栅化阶段：

图元组装：将几何阶段得到的点连接成图元

三角形遍历：检测屏幕上的一个像素是否被一个三角形覆盖，如果被覆盖那么生成一个片元。

片元着色器：通过插值计算每一个片元的数据

逐片元操作：通过深度测试，透明度混合等测试检验一个片元是否会被输出.

传统的光照使用Blinn-phong模型来进行

实际的颜色 = 环境光 + 漫反射强度系数 \* 漫反射强度 + 高光强度系数 \* 高光强度

环境光：背景光线，保证即使没有直射光物体也不会完全表现为黑色

漫反射颜色：通常来源于纹理采样，也就是物体本身的颜色

漫反射强度：KD（漫反射颜色）\* ID（入射光强度）\* max（0，N（法线向量）·L（指向光源的单位向量））

高光颜色：通常提前定义的颜色，高光通常表现为纯白色

高光强度：KS（高光反射颜色）\* IS（入射光强度）\*max（0，N（法线向量）\*L（半程向量））^n（高光指数，越小光照的衰减越慢，光照的范围越大）

半程向量：normalize（L（指向光源的单位向量）\*V（指向观察者的单位向量））

1. **MVC的理解？**

M：Model，保存数据以及对数据的操作，V：ViewUI显示的相关逻辑，C：Controller，Model和View沟通的桥梁。

**8.贝塞尔曲线实现的算法可以简单描述一下吗？**

**9.Lua的Table是什么样的数据结构？里面是具体怎么组织数据的？**

Table是一种十分灵活的数据结构，在Lua中有两种组织数据的方式一种是顺序也就是数组，另一种是键值对也就是字典在XLua的数据转化中，Lua实际上也会被映射为这两种数据类型，Lua的Table可以同时是这两种Table但是在向C#进行映射的时候其中顺序表的部分只能被list接受键值对的部分只能被字典的部分接受。

1. **AB包应该如何进行分包？**

**11.项目中遇到的问题？**

**12.简单描述Shader收集工具？**

**13.C#中闭封包和拆包指的是什么？**

封包指的是将值类型封装为引用类型并保存在堆上的过程，而拆包则是将值类型还原。这个过程中会带来性能的开销，装箱会分配堆内存，拆箱需要类型检查。其作用在于将值类型作为引用类型来使用统一的接口。

# 网易雷火一面

**1.实现碰撞检测的算法**

以Unity为例，碰撞体中存在凸体选项，选择后将会简化模型为凸体此时Unity底层会使用GJK算法和SAT算法来进行检测，如果不选择勾选那么会对每一个三角形面进行检测，这样带来的开销是巨大的并且两个碰撞体都没有选择凸体选项那么Unity不会将他们进行碰撞检测，因为这样会带来巨大到难以承受的开销。

在采用凸体的前提下可以采用两种算法来生成：

GJK算法：计算两个模型当前方向与反方向的最远点的差得到一个单形，如果单形没有覆盖原点，那么取单形最接近圆点的边的法向量为新的方向，如果在迭代过程中不再出现新的最远点那么迭代停止。二维的单形使用三角形来覆盖，三维的单形使用四面体。

SAT算法：计算所有顶点在某一个方向向量上的投影，以长方形包围盒为例，这个方向来源于两个包围盒不平行面的法向量(每个包围盒3条)，以及不平行边两两相交形成的面的法向量(3\*3 = 9条)，则共有15条边要进行检测，那么8\*8\*15共960次计算。如果赠右游戏物体的话带来的开销也也是巨大的，所以需要使用SAP或者四叉树来进行初过滤。

# 途游一面

1. **Lua实现Unity中代码的更新的原理**

XLua实现热更新是通过[Hotfix]属性，这个属性和[LuaCallCSharp]是两个完全无关的属性，在XLua中操作的按钮也不同[Hotfix]是单独的生成热更新按钮而[LuaCallCSharp]是生成桥接文件的按钮。

[Hotfix]本身的实现原理是IL注入程序集，Lua层使用hotfix方法来将所有要替换的Lua函数注册到一张全局表中，而C#层在注入完成后所有带有[Hotfix]属性的方法在执行之前会判断全局表中是否有该函数，如果有那么调用该函数而非原本的C#方法。在替换完成后，这个过程中就存在C#对于Lua的调用，所以需要提前生成桥接文件，这样效率比较高。

[深入理解xLua基于IL代码注入的热更新原理\_luaenv.realstateptr-CSDN博客](https://blog.csdn.net/m1234567q/article/details/134082574)

<https://blog.csdn.net/m1234567q/article/details/134082574>

1. **C#和Lua内存交互的原理**

C#层调用Lua层是使用桥接文件或者反射的方式，桥接文件的形式更快省去了查找的过程，而Lua调用C#则是通过Lua虚拟机的全局表，而在这个过程中的底层实际上都是通过底层的Lua栈。

1. **Unity开发时经常会用到的生命周期方法**

Awake最早执行初始化，当脚本被实例化时执行，只要游戏物体进入场景，那么其上的所有脚本就会被实例化。用来初始化。

Enable在每一次脚本激活时调用，需要满足游戏物体本身是激活的且脚本本身是激活的，在这样的条件下激活游戏物体或者脚本时调用。

Start在第一次Update之前进行调用。要求游戏物体和脚本本身必须激活。

FixedUpdate固定时间进行调用，每1/50秒调用一次，通常将物理相关的操作放到此处执行。

Update每帧进行调用，处理游戏的常规逻辑。

LateUpdate在Update执行完成之后调用，通常用来处理依赖于其他逻辑的操作。

OnDestroy销毁游戏脚本实例时调用，通常用来进行资源的回收。

OnDisabled与Enable相反，在游戏物体被取消激活时调用。

1. **协程的原理**

协程是一个运行在Unity主线程上更加轻量化的用户态线程，本质是一个迭代器对象，通过MoveNext执行下一段代码，通过Current标定当前执行的阶段以及当前继续执行的条件，在执行的过程中遇到yield时将当前函数加入阻塞队列中，每一帧遍历队列通过每一个协程的Current判断当前是否有可以执行的函数。

1. **ECS架构的优势**

主要有两点，在常规项目中，其可以保证所有的组件在内存空间上连续存放，降低查询带来的开销，同时使CPU的缓存均是相同类型的数据，降低缺失率。在Unity项目中还可以降低对Monobehaviour的依赖以优化性能。

1. **分析羊了个羊和黑神话悟空两个游戏**
2. **大致的职业规划**

在未来差不多一年的实习中熟练掌握整个项目的开发流程，在转正后一年的时间内接触项目开发的核心逻辑，在后续的3~5年里参与公司新的核心项目的开发。

# 途游二面

1. **大概讲一讲自己的项目**
2. **C++的Map各自的底层的数据结构，插入时会发生什么动作，哪一些会产生扩容**

Map，底层是红黑树，增删改查的时间复杂度都是O(logn)，在内存中离散存放。

UltimateMap，相当于元素可以重复的Map，底层同样是红黑树。

UnorderedMap，底层是Hash表，使用动态数组实现所以也会出现扩容。

1. **Vector在插入时可能会发生什么操作？**

如果当前的空间无法容纳插入的内容那么会触发扩容机制将整个容器进行移动。

1. **静态多态和动态多态？虚函数介绍一下**

静态多态就是函数重载和模板。

1. **静态多态的偏特化是什么？如何判断一个类中是否存在一个函数**

偏特化是指将模板参数进行约束的特化模板。

判断一个类中是否存在一个函数可以使用concept和require来判断是否满足条件。

Require中函数体的内容可以通过编译那么则会返回true，使用require声明一个约束使用concept来保存一个约束，约束本身也是模板，并且在模板中使用。如果require没有函数体，那么只要()中的值为true便符合模板。

1. **Move和Forward分别的作用**

Move将左值显式转换为右值，而Forward是完美转发，将万能引用进行完整的传递。

1. **写过Shader吗，项目的具体实现**
2. **手撕，保存1~1000的乱序数组，删除其中的一个数字如何通过一次遍历得到缺失的数字**

直接累和然后相减即可。

# 雷火一面A面

**1.template<class T> void f(T){**

**Static int i = 0;**

**Cou<< ++i;**

**}**

**Int main(){**

**f(1);**

**f(1.0);**

**f(1);**

**}**

**输出是什么？**

1 1 2

模板的本质是生成了一个该类型的实例，所以在此处有两个数据类型也就产生了两个实例，而类中的static变量会被存放到静态存储区，是面向于模板实例的，在不同实例之间唯一，所以第一次调用和第三次调用都会式同一个i增加。

**2.** **64位系统中，int\*\* a[4][4]，sizeof(a)占多少字节**

本质还是一个指针数组所以每一个指针的大小为8个字节，那么整体的大小就为4\*4\*8为128个字节。

**3.** **struct A{**

**int a;**

**char b ;**

**int c;**

**char d;**

**}**

**struct p{**

**struct A w[2];**

**stort b;**

**struct A\* p;**

**}**

**4.** **class A{**

**public :**

**A(){printf(“A”);}**

**A(const A&){printf(“b”);}**

**A(A&&){printf(“c”);}**

**A operator = (const A&){printf(“d”);}**

**};**

**class B:public A**

**{**

**public:**

**B():A(){printf(“1”);}**

**B(const B& b):A(b) {printf(“2”);}**

**B(B&& b):A(b){printf(“3”);}**

**B operator = (const B&){printf(“4”);}**

**};**

**int mian()**

**{**

**B b;**

**B m(b);**

**B n=b;**

**B q(std::move(b));  
}**

**5.从1~9这9个数字中随机选出3个不重复的数字，和为奇数的组合有几个？**

和为奇数的情况总共有三个奇数，两个偶数和一个奇数两种情况三个奇数则是C(5,3) = 10；后面一种情况是C(4,2)\*C(5,1) = 30;则总数则为30 + 10 = 40；

**6.有一个A物体和一个B物体，如何判断B物体在A物体的前方？（从向量角度考虑这个问题）**

将两个物体的位置进行相差，然后与方向向量做点积，如果结果大于0则说明B在A的前面，如果结果小于0则说明A在B的前面。

**7.讲一讲Lua中的静态函数和静态变量**

**8.MVC架构的优势和劣势**

优势在于逻辑层和显示层的解耦，让各自的边界更加清晰，易于修改。

劣势在于更加复杂，且跨模块的操作比较麻烦。

# 雷火一面B面

**1.如何判断两个球体是否发生了碰撞，如何获得碰撞点？**

直接计算两个球球心之间的距离是否小于半径。球心坐标加球心连线方向的单位向量乘以球的半径即得到碰撞点的坐标。

**2.如何获得点到射线之间的距离？**

在射线上任取一点，计算该点与目标点的方向向量，然后将该方向向量与射线方向的单位向量进行点乘得到cos，然后乘以该点与目标点的距离即可得到点到射线的距离。

# 腾讯一面

**1.C++和C#的异同点，哪一个编译更快一些，两者的内存分布**

C++会直接编译为机器语言，而C#是编译为中间IL语言在CLR虚拟机上运行，所以C++的编译速度更慢但是运行速度更快但是C#的编译速度更快但是运行速度更慢。

**2.C++哪些存在堆里面，哪些在栈上**

栈通常存放函数的局部变量，函数参数和返回地址这些小型且生命周期较短的变量会被放置到栈上，而堆存放由程序员主动分配的内存。

**3.堆和栈的区别**

栈上数据连续存放，并使用类似于栈的方式进行管理，而堆上的数据是离散存放的。

栈上的数据由计算机自行管理，不会出现内存泄漏，而堆上的数据由程序员自行管理可能会出现内存的泄露。

**4.C#中有类似于指针的操作吗**

函数指针及委托

实际上C#本身可以在unsafe的模式下直接使用指针，跟C++的指针用法相似

**5.C++多态**

多态分为编译时多态和运行时多态，编译时多态的实现主要是模板和函数重载，运行时多态的主要实现是虚函数。

模板有偏特化和完全特化，偏特化有两种，一种是将模板参数直接设置为一个具体的参数，另一个是将模板参数加上制约。

可以使用concept和require来实现判断类中是否存在某一个数据类型。

**6.构造函数和析构函数可以是虚函数吗**

构造函数不能是虚函数，析构函数可以是虚函数以实现子类自定义的内存管理。

**7.游戏中常用的数据结构，数组和链表哪一个遍历的速度更快，为什么连续的空间比不连续的内存空间访问要快。**

数组更快，因为其内存空间连续，两个点，一是寻址的速度更快，二是空间局部性，连续存放的数据会被一次性加载到缓存中，访问时的缺失率更低。

**8.计算机是如何寻址的？**

**9.分页分段机制，为什么要有分页分段机制。计算机如何取到不在内存中的数据？**

将程序的逻辑地址进行分页，同时将内存进行分块，将逻辑页映射到内存的物理块上，分段分页机制的核心作用在于减少难以利用的内存碎片，同时为虚拟内存提供高效的基础。

也就是说此时出现了缺页中断，那么此时系统会进入内核状态并保留现场的状态，同时检查此次访问是否合法，如果合法那么会在辅助表查看当前访问的块在磁盘上的实际位置，并将其调入内存，更新页表等内容随后返回执行。

**10.了解缓存吗**

了解过一些，使用ECS的其中一个目的就是通过将需要访问的数据同时放入缓存以提高访问的性能。

**11.你认为游戏中什么地方使用了多线程，为什么多线程比单线程更快。**

以Unity为例，原生的内容中渲染管线和物理系统是在其他线程上运行的，同时Unity也提供了一些异步的方法，这些方法也是在其他线程上执行。

多线程可以允许多个线程在不同的CPU核心上进行并行运算，显著缩短整体的运算时间。

**12.渲染管线概述，渲染管线中哪些部分在CPU上跑，哪些部分在GPU上跑。**

应用阶段：CPU准备渲染需要的相关的数据并发出DrawCall指令。

几何阶段：

顶点着色器：计算顶点的光照，颜色等信息，同时执行MVP变化将模型从模型空间移动到裁剪空间，同时通过齐次除法得到NDC坐标。

裁剪：将摄像机外不会被渲染的点进行剔除。

屏幕映射：将NDC坐标映射到屏幕坐标。

光栅化阶段：

三角形设置：将顶点着色器得到的点连接成三角形。

三角形遍历：判断屏幕上的一个像素是否被三角形覆盖，如果被覆盖那么生成一个片元。

片元着色器：计算片元实际的颜色信息并写入帧缓存。

逐片元操作：将每一个片元经过深度，透明度等一系列测试后进行输出。

在Unity的渲染管线上只有应用阶段是在CPU上执行，而在我写的C++软光栅渲染器中整个渲染的流程完全在CPU上进行。

# 雷火面试

**1.深度测试的原理**

比较当前片元的深度与深度缓存中值的大小关系，如果当前片元在前面那么将当前片元写入帧缓存并更新深度缓存。

**2.延迟渲染deffershading**

原本是在片元着色器后再进行深度测试，使用延迟渲染的情况下会将深度测试调整到片元着色器之前，避免了大量不会输出的片元参与实际的光照计算，在大量光源的场合明显优于向前计算。

但是需要存储大量的信息，在内存上会具有瓶颈，空间换时间的一个例子。

延迟渲染的时间复杂度为o(n+m)n为渲染模型的时间开销，m为进行光照计算的时间开销。

**3.阴影纹理shadowmap**

在每一个光源的视角计算每一个像素位置深度的最小值并生成一个阴影纹理，随后计算片元在光源视角下的的位置与前面得到的阴影纹理中的值进行比较，如果该片元深度更大那么则该片元在阴影中。

[3D世界中的阴影—ShadowMap原理解析3D世界中的阴影有很多种实现方式，其中 ShadowMap 是比较常用的方案 - 掘金](https://juejin.cn/post/6940078211967483911)

<https://juejin.cn/post/6940078211967483911>

**4.判断线段是否相交**

两种方法，一是写出两条线段的参数方程，然后求解，如果有解那么则相交。另一种是直接通过点乘判断是否平行或者共线，如果没有那么通过叉乘判断4个点是否在同一平面，如果在同一平面那么则视为相交。

**5.EC架构和ECS架构的区别**

EC架构将组件中仍然存在逻辑，到那时ECS架构将所有的逻辑放入System中，组件由纯粹的数据构成。

# 阿里巴巴灵犀互娱-平平无奇工作室

**1.B类型的加载回调函数里如果要加载一个A类型，A类型中又包含一个B类型，这时候应该如何处理。**

可以在实际使用具体对象的时候再进行实例化，而不是在回调的时候再进行实例化。

**2.对于性能优化的理解**

两个方面，从空间角度的优化和从时间角度的优化，实际上在多数的情况下我们不是在优化而是在平衡空间和时间的开销，空间角度的优化就是资源调度策略，什么时候加载资源什么时候卸载资源

**3.C++虚函数的作用**

核心作用是实现运行时多态，也就是在运行时根据对象的实际类型调用函数

**4.C++中什么函数不能被声明为虚函数**

构造函数：C++本身从功能层面就决定了构造函数无法被声明为虚函数。

静态函数：静态函数是面向于类的，没有this指针，无法访问虚函数表，也没有必要访问虚函数表，因为其设计的目的也与多态相悖。

内联函数：虚函数是在运行时确定具体的函数，但是内联函数在编译时就要展开。

**5.解决哈希冲突的方法**

常用的有三种

开放地址法：通过线性探测，二次探测，双重哈希等方法计算下一个地址。

链地址法：哈希表的每一个位置放置一个链表，出现哈希冲突时将新的节点加入链表。

再哈希法：准备多个哈希函数，第一个冲突时使用第二个函数。

**6.红黑树有什么特点，为什么要使用红黑树？**

红黑树相当于是一个平衡条件更加宽松的AVL树，AVL树要求子树高度之差不大于2但是红黑树只要求最高树不超过最低树的两倍。为了实现这个条件红黑树制定了一些规则，所有节点都有一个颜色，红色节点不能与红色节点相连，根节点到不同叶子节点的路径上的黑色节点数相同。

**7.快速排序用到了什么思想？**

分治，将一个大型的问题拆解为多个小问题并递归的进行解决。

**8.lua中有哪些常用的元表**

\_\_index：当访问一个不存在的键值时触发  
\_\_newindex：当给一个不存在的键赋值时触发

\_\_add,\_\_div等：当表参与运算时触发

\_\_call：当表像函数一样被调用时触发

**9.Lua中遍历表的方法有哪些？**

ipairs和pairs，通常使用ipairs来遍历顺序表，pairs来遍历键值表。因为在Lua底层上顺序表本质也是键值表，所以pairs实际上可以遍历所有表，包括键值表，顺序表或者混合表，只是遍历出来的顺序是无序的。

**10.UGUI的底层实现**

UGUI是在3D网格下建立起来的UI系统，它的每个可显示的元素都是以3D模型网格的形式来构建起来的。当UI被实例化时，UGUI首先要做的事就是构建网格。

也就是说当Unity3D制作一个图元，或者一个按钮，或者一个背景时，都会先构建一个方形网格，再将图片放入网格中。可以理解为制造了一个3D模型，用一个网格绑定一个材质球，材质球里存放了要显示的图片。

如果每个元素都会生成一个模型并且绑定一个材质球存入一张图片的话，UI上成千上百个元素就会拥有成千上百个材质球，以及成千上百张图。这样使得引擎在渲染时都需要读取成千上百张图，对每个材质球和网格都进行渲染，这会导致性能开销巨大，drawcall过高，可以简单的理解为一个材质球一个drawcall。(drawcall的原理我们将在后面的章节中介绍)

UGUI也并不是所有的网格和材质球都合并成一个，只有把相同层级的元素，以及相同层级上的拥有相同的材质球参数的才合并在一起。合并成一个网格了就是一个静止的模型了，如果我们移动了任何元素，或者销毁了任何元素，或者改变了任何元素的材质球参数，UGUI则会销毁这个网格，重新构建一个新的。我们设想下，如果我们每时每刻都在移动一个元素的话，UGUI就会不停的拆分合并拆分合并，就会不停的消耗CPU，来使得画面保持应该有的样子。

合并的规则为，同一个Canvas里，相同层级的，相同材质球的元素进行合并，从而减少Drawcall。不过相同层级的概念并不是gameobject 上的节点层级，而是覆盖层级。Canvas说如果两个元素重叠，则可以认为它们是上下层关系，把所有重叠的层级数计算完毕后，第0层的所有元素统一合并，第1层的元素也统一合并，以此类推。

将图片打包为图集就是为了让Unity将多个图片视为一个纹理来进行合批