

Study

Study

自我注意，自我遵守

1. 数据结构

1.1 链表相关算法

1. 链表的基础知识

2. 打印链表

1. 题目：

2. 题解

3. 倒数第k个节点

1. 题目

2. 题解

4. 反转链表

1. 题目

2. 题解

5. 合并两个排序的链表

1. 题目

2. 题解

6. 复杂链表的复制

1. 题目

2. 题解

7. 两个链表的第一个公共节点

1. 题目

2. 题解

8. 删除链表的节点 d

1. 题目

2. 题解

9. 环形链表

1. 题目

2. 解决方案

10. 二叉树转链表

1. 题目描述

2. 解题思路

1.2 栈与队列

1. 栈与队列的基础知识

2. 用两个栈实现队列

1. 题目描述

2. 解题思路

3. 包含main函数的栈

4. 栈的压入、弹出序列

5. 翻转单词顺序

6. 浮动窗口的最大值

7. 表达式括号匹配检测

1. 问题说明

2. 解决方案

8. LRU缓存策略

1. 题目描述

2. LRU的解决思路

1.3 图相关算法

1. 图的基础知识

2. 矩阵中的路径(BFS)

3. 机器人的运动范围(DFS)

4. 有向无循环图(DAG)

1.4 树相关算法

1. 二叉树

1. 基础概念
2. 二叉树遍历
3. 重建二叉树
4. 从上往下打印二叉树
5. 二叉树的镜像
6. 二叉树中和为某一值的路径
7. 二叉树的深度
8. 序列化二叉树
9. 二叉树第K大节点
10. 最近公共父节点
11. 最小堆的创建

2. 二叉搜索树

1. 二叉搜索树基础知识
2. 二叉搜索树的遍历

3. 平衡二叉树

1. 平衡二叉树基础知识

1.5 查找算法

1. 查找基础知识

2. 二分查找
3. 顺序查找
4. 插值查找

1.6 排序算法

1. 排序基础知识

2. 冒泡排序
3. 桶排序
4. 堆排序
5. 归并排序
6. 快速排序

1.7 动态规划

1. 动态规划基础知识

2. 连续子数组的最大和
3. 正则表达式匹配

1.8 数组以及字符串

1. 全排列数组

1. 题目
2. 思路

2. 两数之和

1. 题目
2. 思路

3. 回文字符串的判断

1. 题目描述
2. 思路

2. C++知识

2.1 虚函数的理解

1. 面向对象的三个特征

2. 多态

1. 动态多态
2. 静态多态
3. 虚函数的原理

3. C++多态下的内存布局

1. 空类
2. 普通类(不包含静态变量)
3. 普通类(包含静态成员与普通函数)
4. 普通类(加虚函数)
5. 单一的一般继承(子类带虚函数)
6. 单一的一般继承(带成员函数、虚函数、虚函数重写)

- 7. 单一的虚拟继承(包含成员变量、虚函数、虚函数覆盖、派生类新增虚函数)
 - 8. 总结
4. public private protected三种继承区别
- 1. public公有继承
 - 2. private私有继承
 - 3. protected保护继承
- 2.2 反射机制
- 2.3 内存对齐
- 1. 什么是内存对齐
 - 2. 内存对齐规则
- 2.4 线程相关
- 1. 什么是线程
 - 1.1 什么是进程，什么是线程
 - 1.2 进程与线程的区别
 - 1.3 线程的几种状态
 - 3. 什么是自旋锁
 - 3.1 概念
 - 3.2 CAS操作实现自旋锁
 - 4. 什么是乐观锁和悲观锁
 - 4.1 乐观锁
 - 4.2 悲观锁
 - 5. 多线程
 - 1. 基础知识
 - 2. 按序打印
 - 3. 打印零与奇偶数
 - 4. H2O生成
 - 5. 交替打印字符串
 - 6. 哲学家进餐
- 2.5 C++中基础知识
- 1. new与malloc的区别
 - 1.1 申请内存所在位置
 - 1.2 返回类型安全性
 - 1.3 内存分配失败时的返回值
 - 1.4 是否需要指定内存大小
 - 1.5 是否调用构造、析构函数
 - 2. vector与list的区别
 - 2.1 vector相关
 - 2.2 list相关
 - 2.3 map与unordered_map的区别/插入和删除的复杂度
 - 3. C++修饰符
 - 1. const
 - 2. static
 - 3. volatile
 - 4. explicit(显示)关键字
 - 5. extern
 - 4. 智能指针
 - 1. Shared_ptr
 - 2.unique_ptr
 - 3. Weak_ptr
 - 5. 深拷贝与浅拷贝
 - 1. 普通类型对象的拷贝
 - 2. 类对象的拷贝
 - 3. 浅拷贝
 - 4. 深拷贝
 - 6. 宏定义与inline函数
 - 1. 宏定义
 - 2. inline函数
- 2.6 C++ 设计模式

设计模式原则

1. 单一职责原则(SRP, Single Responsibility Principle)
2. 里氏替换原则(LSP, Liskov Substitution Principle)
3. 依赖倒置原则(DIP, Dependence Inversion Principle)
4. 接口隔离原则(ISP, Interface Segregation Principle)
5. 迪米特原则(LoD, Law of Demeter)
6. 开放封闭原则(OCP, Open Close Principle)

1. 适配器模式

1. 概念
2. 类适配器
3. 对象适配器
4. 注意要点
5. 使用场景

2. 桥接模式

1. 概念
2. 具体的例子

3. 观察者模式

1. 概念
2. 具体的例子

4. 单例模式

1. 概念
2. 具体的例子
3. 注意的事情

2.7 内存管理

- 1.

3. 渲染架构

1. ECS设计模式

4. 渲染算法

4.1 OpenGL ES 2.0与3.0的区别

4.1.1 概述

4.1.2 OpenGL ES 3.0新功能

1. Occlusion Queries:遮挡查询
2. Transform Feedback:变换反馈
3. Instanced Rendering:实例化渲染
4. Multiple Render Targets:多目标渲染
5. 新的纹理功能
6. 同步信号和栅栏
7. 着色器新增的功能

4.2 四元数、欧拉角、矩阵的区别与联系

1. 矩阵欧拉角四元数优缺点

1. 比对表格
2. 适用地方

1. 矩阵

1. 转置与逆矩阵
2. 2D旋转
3. 3D旋转
4. 法线与矩阵的关系

2. 四元数

3. 欧拉角

4. 相互转换

5. 施密特正交化(Gram-Schmidt process)

6. 矩阵的行空间、列空间解释

1. 线性组合
2. 行空间
3. 列空间

7. 相机矩阵的推导

8. 其次坐标的几何意义

4.3 渲染管线

1. 整体渲染管线流程
 2. 渲染管线具体步骤
 1. 顶点缓冲区/数组对象
 2. 顶点着色器
 3. 图元装配
 4. 光栅化
 5. 片元着色器
 6. 像素归属测试
 7. 剪裁测试
 8. 模板/深度测试
 9. 混合
 10. 抖动
 3. 总结
- 4.4 渲染中的混合操作
1. OpenGL ES中混合介绍
 1. 混合运算公式
 2. 混合范围
 3. 源颜色与源因子、目标颜色与目标因子
 4. 常量混合
 5. 混合运算符的设置
 2. 颜色缓冲区与深度缓冲区的介绍
 1. 颜色缓冲区
 2. 深度缓冲区
 3. 半透明混合为什么会出现问题
 4. 如何解决半透明渲染问题
 1. 次序渲染
 2. 次序无关半透明深度剥离
 3. 总结
- 4.5 Z-Fighting问题的出现以及处理
1. Z-Fighting问题的描述
 2. 为什么会产生Z-fighting现象
 - 2.1 原因1
 - 2.2 原因2
 3. 从透视投影矩阵计算的角度解释以及如何避免
 - 3.1 从矩阵计算的角度看待问题
 - 3.2 如何避免z-fight问题
 4. 深度值的非线性到线性的转化
- 4.6 FBO相关问题
- 4.7 延迟渲染
- 4.8 次序无关半透明渲染
1. 基础理论
 - 1.1 概述
 - 1.2 基础概念
 - 1.3 渲染管线
 2. 深度剥离实现
- 4.9 柏林噪声、细胞噪声
1. Value Noise
 1. 理论
 2. 效果及原因
 2. Perlin Noise
 1. 理论
 2. 说着色器实现
 3. 细胞噪声
 1. 理论
 2. 着色器实现
- 4.10 Bilnn_Phone/Phone光照模型（点光源、聚光灯、平行光）
- 4.11 多光源渲染问题
- 4.12 法线贴图

- 1. 切线空间
 - 2. 切线副法线的计算
 - 2.1 在CPU侧计算
 - 2.2 在着色器中计算
 - 3. 法线贴图的生成
 - 4.13 视察贴图（三种）
 - 4.14 三角剖分
 - 4.15 抗锯齿处理
 - 1. FXAA快速近似抗锯齿
 - 1. 原理
 - 2. 着色器实现
 - 2. MSAA多重采样抗锯齿
 - 1. 问题
 - 2. 多重采样抗锯齿
 - 3. 总结
 - 4.16 空间管理算法
 - 4.17 PBR渲染管线
 - 4.18 IBL
 - 4.19 阴影贴图、阴影体
 - 4.20 SSAO环境光遮蔽
 - 1. SSAO概述
 - 1.1. SSAO解决什么问题
 - 1.2. SSAO背后的原理
 - 2. SSAO技术实现
 - 2.1. 概述技术实现步骤
 - 2.2 延迟渲染
 - 2.3. 遮蔽因子计算
 - 3. SSAO的技术流派
 - 1. 六种SSAO的技术图
 - 4.21 实时渲染中的软阴影技术
 - 4.22 多viewport渲染问题
- ## 5. 图形后期渲染
- 5.1 Bloom曝光
 - 5.2 DOF景深
 - 5.2 HDR高光渲染
 - 5.3 运动模糊
 - 5.4 室外阴影渲染PSSM(平行分割阴影图)
 - 5.5 体积云
 - 5.6 体积光
- ## 6. 图像渲染
- 6.1 高斯模糊
 - 1. 一维高斯模糊
 - 1. 基础原理
 - 2. 着色器实现
 - 2. 二维高斯模糊
 - 2. 着色器实现
 - 6.2 双线性插值
 - 1. 基础原理
 - 2. 着色器实现
 - 6.3 高\低通滤波器(中值滤波、巴特沃斯滤波、高斯低通滤波)
 - 1. 理想高\低通滤波器
 - 1. 原理
 - 2. 着色器实现
 - 2. 中值滤波
 - 1. 原理
 - 2. 着色器实现
 - 3. 巴特沃斯(Butterworth)滤波
 - 1. 原理

- 2. 着色器实现
- 4. 高斯低通滤波
 - 1. 原理
 - 2. 着色器实现
- 6.4 饱和度、对比度、曝光、色阶调整
 - 1. 饱和度
 - 1. 原理
 - 2. 着色器实现
 - 2. 对比度
 - 1. 原理
 - 2. 着色器实现
 - 3. 曝光
 - 1. 原理
 - 2. 着色器实现
- 4. 图像增强--USM锐化
 - 1. 原理
 - 2. 着色器实现
- 6.5 图像描边 (索贝尔、拉普拉斯、罗伯特、普洱瑞斯)
 - 1. 图像描边的本质
 - 1.1 定义
 - 1.2 像素与其八领域矩阵表示
 - 2. 索贝尔算子
 - 2.1 定义与作用：用来对图像进行边缘检测
 - 2.2 着色器实现
 - 3. 罗伯特算子
 - 3.1 罗伯特算子特点
 - 3.2 着色器实现
 - 4. 拉普拉斯算子
 - 1. 导数求导公式
 - 2. 微分公式
 - 3. 拉普拉斯边缘检测
 - 4. 着色器实现
 - 5. prewitt(普洱瑞斯算子)
 - 1. 原理
 - 2. 卷积模版
 - 3. 着色器实现
- 6.6 非真实感着色
 - 1. 油画相关算法
 - 1. 图像油画处理
 - 2. Kuwahara滤波进行油画处理
 - 2. 钢笔画
 - 3. 卡通着色
- 6.7 老电影
 - 1. 理论
 - 2. 着色器实现及效果
- 6.8 最小二乘法
 - 1. 最小二乘概念
 - 1.概念
 - 2. 微积分处理
 - 3. 线性代数处理
 - 2. 图像领域使用MLS
 - 1. 概述
 - 2. MLS变换类型
 - 3. CPU侧MLS变形
- 6.9 颜色查找表原理
 - 1. 理论知识
 - 2. 着色器实现
- 6.10 美颜相关算法

1. 全局磨皮算法
2. 美白
 - 1. 使用logarithmic Curve函数
7. 渲染优化
 - 1. 针对移动端TBDR架构GPU特性的渲染优化
8. 高等数学与线性代数

字节跳动---游东

自我注意，自我遵守

1. 不能在私下嚼舌头根，不去议论任何同事的事情
2. 诚实守信，不摆资历，不拉帮结派
3. 愤怒是跟自己的愚蠢是成正比的，如果不想让自己变成愚蠢的人，那就不要愤怒
4. 专注于自己的工作效率，工作质量，工作的上下游事情
5. 努力提升自己的专业技能，加强自己的自律性管理，提升自己行业、人生认知
6. 踏实专注于一件事情，不必为了一些小人而徒增烦恼

1. 数据结构

1.1 链表相关算法

1. 链表的基础知识

2. 打印链表

1. 题目：

输入一个链表的头节点，从尾到头反过来返回每一个节点的值(用数组返回)

```
输入:head = [1,3,2]
输出: [2,3,1]
```

2. 题解

```
#include <vector>
using namespace std;
struct ListNode
{
    int val;
    ListNode* next;
    ListNode(int x): val(x), next(nullptr)
    {
    }
};
class Solution
{
public:
    vector<int> res;
    //使用algorithm算法中的reverse反转res
    vector<int> reversePrint(ListNode* head)
    {
        while(head){
```

```

        res.push_back(head->val);
        head = head->next;
    }
    reverse(res.begin(),res.end());
    return res;
}
//入栈法
vector<int> reversePrint1(ListNode* head)
{
    stack<int> s;
    //入栈
    while(head){
        s.push(head->val);
        head = head->next;
    }
    //出栈
    while(!s.empty()){
        res.push_back(s.top());
        s.pop();
    }
    return res;
}
//递归
vector<int> reversePrint2(ListNode* head)
{
    if(head == nullptr)
        return res;
    reversePrint2(head->next);
    res.push_back(head->val);
    return res;
}
//改变链表结构
vector<int> reversePrint3(ListNode* head)
{
    ListNode *pre = nullptr;
    ListNode *next = head;
    ListNode *cur = head;
    while(cur){
        next = cur->next;//保存当前结点的下一个节点
        cur->next = pre;//当前结点指向前一个节点，反向改变指针
        pre = cur;//更新前一个节点
        cur = next;//更新当前结点
    }
    while(pre){//上一个while循环结束后，pre指向新的链表头
        res.push_back(pre->val);
        pre = pre->next;
    }
    return res;
}
};

```

3. 倒数第k个节点

1. 题目

给定一个链表，删除链表的倒数第n个节点，并且返回链表的头节点

给定一个链表: 1->2->3->4->5, 和 n = 2
当删除了倒数第二个节点后, 链表变为: 1->2->3->5

说明:

给定的n保证是有效的

进阶:

能尝试使用一趟扫描实现吗?

2. 题解

1. 基本方法(两次遍历): 先遍历整个链表, 求出长度len, 要删除倒数第n个节点, 也就是顺数N = len - n + 1个节点, 于是再一次遍历即可
2. 一次遍历方法
 - 设置一个哑节点, 用来保证可以删除头节点;
 - 设置两个节点p, q, 让两个节点之间的距离为n(假设p不动, q往后移动)
 - 当q->next不为空时, pq两个节点一起往后移动, 直到q->next为空则停止
 - 将p->next的节点进行删除, 就是题目的答案

```
class Solution
{
public:
    ListNode* removeNthFromEnd(ListNode* head, int n)
    {
        ListNode* dummy = new ListNode(0);
        if(!head)
            return head;
        ListNode* p = dummy;
        ListNode* q = dummy;
        for(int i = 0; i < n; i++)
        {
            q = q->next;
        }
        while(q->next)
        {
            p = p->next;
            q = q->next;
        }
        //删除p->next的节点
        p->next = p->next->next;
        ListNode* res = dummy->next;
        delete dummy; //删除哑节点
        return res;
    }
};
```

4. 反转链表

1. 题目

反转一个链表

输入: 1->2->3->4->5->nullptr
输出: 5->4->3->2->1->nullptr

进阶

可以使用迭代或者递归地反转链表

2. 题解

1. 方法一：迭代

在遍历链表时，将当前节点的next指针改为指向前一个元素。由于节点没有引用其上一个节点，因此必须事先存储其前一个元素。在更改引用之前，还需要另一个指针来存储下一个节点。不要忘记在最后返回新的头引用

```
ListNode* reverseList(ListNode head)
{
    ListNode* prev = nullptr;
    ListNode* curr = head;
    while(curr != nullptr)
    {
        ListNode* nextTemp = curr->next;
        curr->next = prev;
        prev = curr;
        curr = nextTemp;
    }
    return prev;
}
```

复杂度：

- 时间复杂度:O(n),假设n时列表的长度，时间复杂度是O(n)
- 空间复杂度:O(1)

2. 方法二：递归

递归版本稍微复杂一些，其关键在于反向工作。假设列表的其余部分已经被反转，现在我该如何反转它前面的部分？

假设列表为：

$$n_1 - > \dots - > n_{k-1} - > n_k - > n_{k+1} - > \dots > n_m - > nullptr$$

若从节点 n_{k+1} 到 n_m 已经被反转，而我们正处于 n_k

$$n_1 - > \dots - > n_{k-1} - > n_k - > n_{k+1} < - \dots < - n_m < - nullptr$$

我们希望 n_{k+1} 的下一个节点指向 n_k

所以， $n_k -> \text{next} -> \text{next} = n_k$

要小心的是 n_1 的下一个必须指向 $nullptr$ ，如果忽略了，你的链表可能回产生循环，如果使用大小为的2的链表测试代码，则可能捕获此错误

```
public ListNode* reverseList(ListNode* head)
{
    if(head == nullptr || head->next == nullptr)
        return head;
    ListNode* p = reverseList(head->next);
    head->next->next = nullptr;
    head->next = nullptr;
    return p;
}
```

复杂度分析：

- 时间复杂度:O(n), 假设n是列表的长度, 那么时间复杂度为O(n)
- 空间复杂度:O(n), 由于使用递归, 将会使用隐式栈空间, 递归深度可能会达到n层

5. 合并两个排序的链表

1. 题目

将两个升序链表合并为一个新的升序链表并返回, 新链表由通过拼接给定的两个链表的所有节点组成的

```
输入: 1->2->4, 1->3->4
输出: 1->1->2->3->4->4
```

2. 题解

- 递归实现
- 新链表不需要构造新的节点
 - 终止条件: 两条链表分别名为l1和l2, 当l1和l2为空或l2为空时结束
 - 返回值: 每一层调用都返回排序好的链表头
 - 本级递归内容: 如果l1的val值更小, 则将l1->next与排序好的链表头相接, l2同理
 - O(m + n), m为l1的长度, n为l2的长度

```
struct ListNode
{
    int val;
    ListNode* next;
    ListNode(int x):
        val(x)
    {
    }
};

class Solution
{
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2)
    {
        if(l1 == nullptr)
            return l2;
        if(l2 == nullptr)
            return l1;
        if(l1->val < l2->val)
        {
            l1->next = mergeTwoLists(l1->next, l2);
            return l1;
        }
        if(l1->val >= l2->val)
        {
            l2->next = mergeTwoLists(l1, l2->next);
            return l2;
        }
    }
};
```

递归解决方法

```
ListNode* mergeTwoLists(ListNode* l1, ListNode* l2)
```

```

{
    ListNode* head = new ListNode(-1);
    ListNode prev = head;
    while(l1 != nullptr && l2 != nullptr)
    {
        if(l1->val < l2->val)
        {
            prev->next = l1;
            l1 = l1->next;
        }else{
            prev->next = l2;
            l2 = l2->next;
        }
        prev = prev->next;
    }
    //特殊情况,比如某个链表已经为空
    prev->next = l1 == nullptr?l2:l1;
}

```

6. 复杂链表的复制

1. 题目

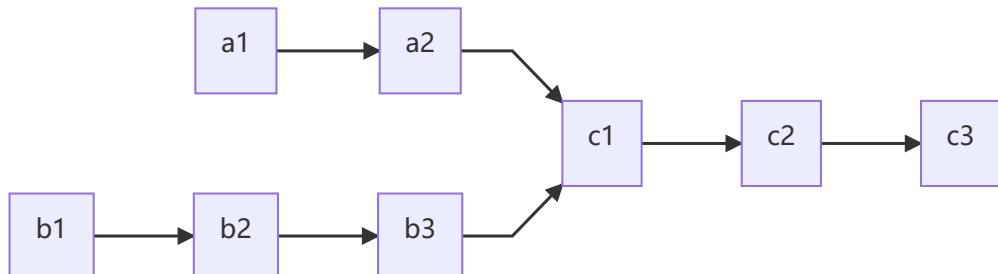
2. 题解

7. 两个链表的第一个公共节点

1. 题目

输入两个链表，找出它们的第一个公共节点

如下所示在节点c1处开始相交：



注意：

- 如果两个链表没有交点，返回`nullptr`
- 在返回结果后，两个链表仍需保持原有的结构
- 假定整个链表结构中咩有循环
- 程序尽量满足 $O(n)$ 时间复杂度， $O(1)$ 空间复杂度

2. 题解

使用两个指针`node1`与`node2`分别指向两个链表的`headA`与`headB`节点，然后怒同时分别逐节点遍历，当`node1`到达链表`headA`的末尾时，重新定位到链表`headB`头节点，当`node2`到达链表`headB`的末尾时，重新定位到链表`headA`的头节点

```

ListNode* getIntersectionNode(ListNode* headA, ListNode* headB)
{
    ListNode* node1 = headA;
    ListNode* node2 = headB;
    while(node1 != node2)
    {
        node1 = node1 != nullptr? node1->next:headB;
        node2 = node2 != nullptr? node2->next:headA;
    }
    return node1;
}

```

复杂度分析：

- 时间复杂度:O(M+N)
- 空间复杂度:O(1)

8. 删除链表的节点 d

1. 题目

给定单链表的头指针和一个要删除的节点的值，定义一个函数删除该节点

返回删除后的链表的头节点

示例

```

输入:head = [4,5,1,9], val = 5
输出:[4,1,9]

```

解释：给定你链表中值为5的第二个节点，那么在调用了你的函数之后，该链表应变为4->1->9

说明：

- 题目保证链表中节点的值互不相同
- 若使用c或者c++语言，不需要free或delete被删除的节点

2. 题解

```

ListNode* deleteNode(ListNode* head, int val)
{
    if(head->val == val)
        return head->next;
    ListNode* pre = head;
    ListNode* cur = head->next;
    while(cur != nullptr || cur->val != val)
    {
        pre = cur;
        cur = cur->next;
    }
    if(cur != nullptr)
        pre->next = cur->next;
}

```

9. 环形链表

1. 题目

给定一个链表，判断链表中是否有环

为了表示给定链表中的环，我们使用整数pos来表示链表尾连接到链表中的位置(索引从0开始)。如果pos是-1，则在该链表中没有环

示例：

```
输入: head = [3,2,0,-4],pos = 1  
输出: true  
解释: 链表中有一个环, 其尾部连接到第二个节点
```

2. 解决方案

想象一下，有两个速度不同的跑步者，如果他们在直路上行驶，快跑者将首先到达目的地。但是如果，他们在圆形的跑道上跑步，那么快跑者如果继续跑步就会追上慢跑者。

- 如果没有环，快指针将停在链表的末尾
- 如果有环，快指针最终将与慢指针相遇

慢指针每次移动一步，快指针每次移动两步，如果环的长度为M，经过M次迭代后，快指针肯定会多环绕一周，并赶上慢指针

空间复杂度：O(1)

时间复杂度：O(n)

代码：

```
linkNode* judgeRingNode(linkNode* root)  
{  
    if(nullptr == root || nullptr == root->next)  
    {  
        return 0;  
    }  
    linkNode* low = root;  
    linkNode* quick = root;  
    int ringNodeId = 0;  
    while(low && quick && quick->next)  
    {  
        low = low->next;  
        quick = quick->next->next;  
        if(low == quick)  
        {  
            low = root;  
            while(low != quick)  
            {  
                low = low->next;  
                quick = quick->next;  
                printf("how Many steps.\n");  
                ringNodeId++;  
            }  
            printf("ring node id is:%d\n",ringNodeId);  
            return low;  
        }  
    }  
    return nullptr;  
}
```

10. 二叉树转链表

1. 题目描述

给定一个二叉树，将它原地展开为链表

2. 解题思路

1. 转换递归时记住先转换右子树，在转左子树。
2. 需要记录一下右子树转换完之后链表的头节点在哪里。
3. 没有next指针，直接拿right当作next指针，那么left指针我们赋值为null就可以了。

代码：

```
class Solution
{
private:
    TreeNode* prev = nullptr;
public:
    void flatten(TreeNode* root)
    {
        if(root == nullptr) return;
        flatten(root->right); //先转换右子树
        flatten(root->left);
        root->right = prev; //右子树指向链表的头
        root->left = nullptr; //左子树置为空
        prev = root; //当前节点为链表头
    }
};
```

1.2 栈与队列

1. 栈与队列的基础知识

说明：std::deque底层实现，为什么要分段连续？好处是什么

2. 用两个栈实现队列

1. 题目描述

2. 解题思路

```
#pragma once
#include <iostream>
#include <stack>
class QueueDoubleStack
{
public:
    QueueDoubleStack(){}
    ~QueueDoubleStack(){}
    void push(int value)
    {
        mainStack.push(value);
    }
    int pop()
    {
        if(cacheStack.empty() && mainStack.empty())
            return -1;
        int value;
```

```

    if(!cacheStack.empty())
    {
        value = cacheStack.top();
        cacheStack.pop();
    }else{
        while(!mainStack.empty())
        {
            int temp = mainStack.top();
            cacheStack.push(temp);
            mainStack.pop();
        }
        value = cacheStack.top();
        cacheStack.pop();
    }
    return value;
}
private:
stack<int> mainStack;
stack<int> cacheStack;
};

```

3. 包含main函数的栈

4. 栈的压入、弹出序列

5. 翻转单词顺序

6. 浮动窗口的最大值

7. 表达式括号匹配检测

1. 问题说明

假设数学表达式中允许包含两种括号:圆括号"()"和方括号"[]", 嵌套顺序任意

正确的嵌套模式: (())、[(())]

正确的表达式例: (a + b)[c * (d - e)]

错误的嵌套模式: [())、(()]

比如, 在处理表达式a时

```
4 + (2 + 8) *[5/(9 - 7)]
```

有以下步骤

1. 检测到第一个括号"("
2. 检测到第二个括号")", 说明子表达式"4 + (2 + 8)"已完成匹配
3. 检测到第三个括号"[
4. 检测到第四个括号"(", 与3中的括号不匹配, 但由于同是左括号, 可以继续匹配
5. 检测到第五个括号")", 由括号的作用可知, 后来的括号比先来的括号优先级高, 因此与4中括号匹配
6. 检测到第六个括号"]", 由于原来优先级更高的括号已经完成, 因此与3中的括号匹配, 致此所有括号匹配完成

2. 解决方案

成功匹配的条件

每一个检测到的括号与已检测到的优先级最高的括号都匹配

匹配失败的条件

- 检测到与已检测到的优先级最高的括号不匹配的括号
- 扫描完整个表达式，还是有已检测到的括号没有完成匹配

代码实现：

```
string left = "{[(";
string right = "}]";
bool judge(string str)
{
    stack<char> s;
    if(str.length() == 0)
        return true;
    int il = -1;
    int ir = -1;
    ir = right.find(str[0]);
    if(il >= 0)
        return false;
    for(int i = 0; i < str.length(); i++)
    {
        char chr = str[i];
        il = left.find(chr);
        ir = right.find(chr);
        if(il >= 0)//左括号
        {
            s.push(chr);//入栈
        }else if(ir >= 0)//右括号
        {
            if(!s.empty() && s.top() == right[ir])
                s.pop();
            else
                return false;
        }
    }
    return s.empty();
}
```

8. LRU缓存策略

1. 题目描述

缓存机制有不同的缓存调度方法，常见的就是FIFO、LRU、LFU等策略

1. FIFO(First in First out):先进先出，淘汰最先进来的数据，最迟进来的被最迟删除，符合队列的性质
2. LRU(Least recently used):最近最少使用，删除最近不使用的数据
3. LFU(Least frequently used):最近使用次数最少的，删除使用次数最少的数据

2. LRU的解决思路

1. 使用双向链表，使用头尾两个卫兵
2. 使用哈希表，使查找删除的时间复杂度为O(1)

代码实现：

```
#pragma once
```

```
#include <stdlib.h>
#include <iostream>
#include <unordered_map>
using namespace std;
struct LRUNode
{
    int key;
    LRUNode* prev;
    LRUNode* next;
    LRUNode(int x):
        prev(nullptr),
        next(nullptr),
        key(x)
    {}
};
class LRU
{
public:
    LRU(int capacity)
    {
        total_ = 0;
        capacity_ = capacity;
        LRUNode* l1 = new LRUNode(-1);
        LRUNode* l2 = new LRUNode(-1);
        head_ = l1;
        tail_ = l2;
        head_->next = tail_;
        tail_->prev = head_;
    }
    ~LRU()
    {
        LRUNode* temp;
        while (head_)
        {
            temp = head_->next;
            delete head_;
            head_ = nullptr;
            head_ = temp;
        }
        printf("释放链表的内存空间。\\n");
    }
    int get(int key)
    {
        if(dic.find(key) == dic.end())
            return -1;
        //在哈希表中查找到所需要的数值
        LRUNode* node = dic[key].second;
        //将该节点从链表中摘除出来，并确保链表是一条链
        node->prev->next = node->next;
        node->next->prev = node->prev;
        //将该节点移动到链表的头节点
        node->next = tail_;
        node->prev = tail_->prev;
        tail_->prev->next = node;

        tail_->prev = node;
        return dic[key].first;
    }
}
```

```

void put(int key, int value)
{
    if(dic.find(key) != dic.end())
    {
        dic[key].first = value;
        LRUNode* node = dic[key].second;
        //从当前位置摘除，并将当前节点的上下节点链接到一起
        node->prev->next = node->next;
        node->next->prev = node->prev;
        //将摘除后的节点链接到尾节点，以尾节点为最新访问的节点
        node->next = tail_;
        tail_->prev = tail_->prev;
        tail_->prev->next = node;

        tail_->prev = node;

    }else{
        //假如缓存容器已满，那么就删除一直未被访问的头节点
        if(total_ == capacity_)
        {
            LRUNode* node = head_->next;
            dic.erase(node->key);
            head_->next = node->next;
            node->next->prev = head_;
            delete node;
            total_--;
        }
        //将最新的值组装放入到哈希表中，删除与查找都是时间复杂度是O(1)
        LRUNode* node = new LRUNode(key);
        dic[key] = make_pair(value, node);
        //将最新的节点添加到链表的尾部
        node->next = tail_;
        tail_->prev->next = node;
        node->prev = tail_->prev;
        tail_->prev = node;
        total_++;
    }
}

private:
    int total_;
    int capacity_;
    LRUNode* head_;
    LRUNode* tail_;
    unordered_map<int,pair<int,LRUNode*>> dic;
};

```

1.3 图相关算法

1. 图的基础知识
2. 矩阵中的路径(BFS)
3. 机器人的运动范围(DFS)
4. 有向无循环图(DAG)

1.4 树相关算法

1. 二叉树

1. 基础概念

Huffman编码使用了二叉树的结构

- 二叉树包含左子树与右子树

满二叉树: 对于一颗二叉树，如果每一个非叶子结点都存在左右子树，并且二叉树中所有的叶子结点都在同一层中，这样的二叉树称为满二叉树。

完全二叉树: 对于一颗具有n个节点的二叉树按照层次编号，同时，左右子树按照先后左右编号，如果编号为i的节点与同样深度的满二叉树中编号为i的节点在二叉树中的位置完全相同，则这颗二叉树称之为完全二叉树。

- 二叉树的存储结构

链式存储结构

顺序存储结构

2. 二叉树遍历

```
struct TreeNode
{
    int data;
    struct TreeNode* left;
    struct TreeNode* right;
};
```

- 前序遍历

首先遍历根节点，其次遍历左孩子，在遍历右孩子，按照如此顺序遍历整棵树

```
//前序遍历
void pre_order(TreeNode* p)
{
    if(p != nullptr)
    {
        printf("%d\n", p->data);
        pre_order(p->left);
        pre_order(p->right);
    }
}
```

- 中序遍历

首先遍历左子树，其次遍历父节点，最后遍历右子树，按照如此顺序遍历整棵树

```
//中序遍历
void in_order(TreeNode* p)
{
    if(p != nullptr)
    {
        in_order(p->left);
        printf("%d\n", p->data);
        in_order(p->right);
    }
}
```

- 后序遍历

首先遍历左子树，其次遍历右子树，最后遍历父节点，按照如此顺序遍历整棵树

```
//后序遍历
void post_order(TreeNode* p)
{
    if(p != nullptr)
    {
        post_order(p->right);
        post_order(p->left);
        printf("%d\n", p->data);
    }
}
```

- 层次遍历

需要用到链表存储每一层的节点，同时，遍历完一个节点，将其左右子节点添加进链表中。

```
//层次遍历
void lever_order(TreeNode* p)
{
    //使用队列
    list<TreeNode*> t;
    if(p != nullptr)
    {
        t.push_back(p);
    }
    while(t.size() > 0)
    {
        printf("%d\n", t.front()->data);
        if(t.front()->left != nullptr)
        {
            t.push_back(t.front()->left);
        }
        if(t.front()->right != nullptr)
        {
            t.push_back(t.front()->right);
        }
        t.pop_front();
    }
}
```

3. 重建二叉树

1. 题目

二叉树是非线性结构，每个结点会有零个、一个或两个孩子结点，一个二叉树的遍历序列不能决定一棵二叉树，但某些不同的遍历序列组合可以唯一确定一棵二叉树。

可以证明，给定一棵二叉树的**前序遍历序列**和**中序遍历序列**可以唯一确定一棵二叉树的结构，给定一棵二叉树的**后序遍历序列**和**中序遍历序列**也可以唯一确定一棵二叉树的结构。

注意：这还有一个条件：**二叉树的任意两个结点的值都不相同。**

算法如下：

- 用前序序列的第一个结点（后序序列的最后一个结点）作为根结点
- 在中序序列中查找根结点的位置，并以此为界将中序序列划分为左、右两个序列（左、右子树）

- 根据左、右子树的中序序列中的结点个数，将前序序列（后序）去掉根结点后的序列划分为左、右两个序列，它们分别是左、右子树的前序（后序）序列
- 对左、右子树的前序（后序）序列和中序序列递归地实施同样方法，直到所得左、右子树为空

2.题解

```
#include <iostream>
#include <vector>
using namespace std;

// Definition for binary tree
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}

};

/* 先序遍历第一个位置肯定是根节点node,
中序遍历的根节点位置在中间p，在p左边的肯定是node的左子树的中序数组，p右边的肯定是node的右子树的中序数组
另一方面，先序遍历的第二个位置到p，也是node左子树的先序子数组，剩下p右边的就是node的右子树的先序子数组
把四个数组找出来，分左右递归调用即可
*/
class Solution
{
public:
    struct TreeNode* reConstructBinaryTree(vector<int> pre, vector<int> vin)
    {
        int in_size = vin.size(); // 获取序列的长度
        if (in_size == 0)
            return NULL;
        // 分别存储先序序列的左子树，先序序列的右子树，中序序列的左子树，中序序列的右子树
        vector<int> pre_left, pre_right, in_left, in_right;
        int val = pre[0]; // 先序遍历第一个位置是根节点node，取其值
        // 新建一个树结点，并传入结点值
        TreeNode* node = new TreeNode(val);

        // p用于储存中序序列中根节点的位置
        int p = 0;
        for (p; p < in_size; ++p)
        {
            if (vin[p] == val)
                break; // 找到即跳出for循环
        }
        for (int i = 0; i < in_size; ++i)
        {
            if (i < p)
            {
                // 建立中序序列的左子树和前序序列的左子树
                in_left.push_back(vin[i]);
                pre_left.push_back(pre[i + 1]); // 前序第一个为根节点，+1从下一个开始记录
            } else if (i > p) {
                // 建立中序序列的右子树和前序序列的右子树
                in_right.push_back(vin[i]);
                pre_right.push_back(pre[i + 1]);
            }
        }
        node->left = reConstructBinaryTree(pre_left, in_left);
        node->right = reConstructBinaryTree(pre_right, in_right);
    }
};
```

录

```

        pre_right.push_back(pre[i]);
    }
}

//取出前序和中序遍历根节点左边和右边的子树
//递归，再对其进行上述所有步骤，即再区分子树的左、右子树数，直到叶节点
node->left = reConstructBinaryTree(pre_left, in_left);
node->right = reConstructBinaryTree(pre_right, in_right);
return node;
}
};

//test=====后序递归遍历二叉树
void Posorder(TreeNode* &T)
{
    if (T){//当结点不为空的时候执行
        //左右中
        Posorder(T->left);
        Posorder(T->right);
        cout << T->val;
        printf("T->val:%d\n", T->val);
    }else{
        T = NULL;
    }
}
int main()
{
    vector<int>pre{1,2,4,7,3,5,6,8};
    vector<int>vin{4,7,2,1,5,3,8,6};
    Solution T;
    TreeNode* node=T.reConstructBinaryTree(pre, vin);
    //测试---输出后续遍历
    printf("后续遍历为:\n");
    Posorder(node);
    system("pause");
}

```

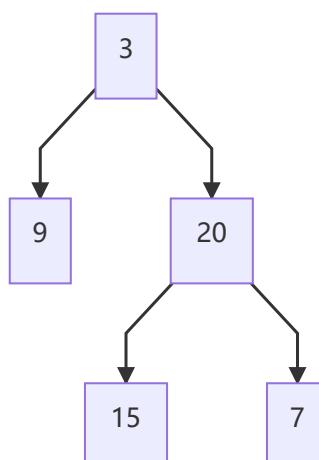
4. 从上往下打印二叉树

1.题目

从上到下打印出二叉树的每个节点，同一层的节点按照从左到右的顺序打印

例如：

给定二叉树:[3,9,20,null,null,15,7]



返回：

[3,9,20,15,7]

提示：

节点总数<=1000

2. 题解

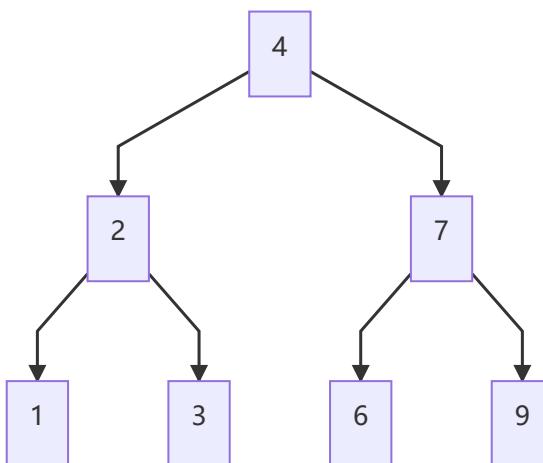
```
class Solution
{
public:
    vector<int> levelOrder(TreeNode* root)
    {
        queue<TreeNode*> q;
        vector<int> ans;
        if(root == nullptr)
            return ans;
        while(!q.empty())
        {
            TreeNode* e = q.front();
            q.pop();
            ans.push_back(e->val);
            if(e->left)
                q.push(e->left);
            if(e->right)
                q.push(e->right);
        }
        return ans;
    }
};
```

5. 二叉树的镜像

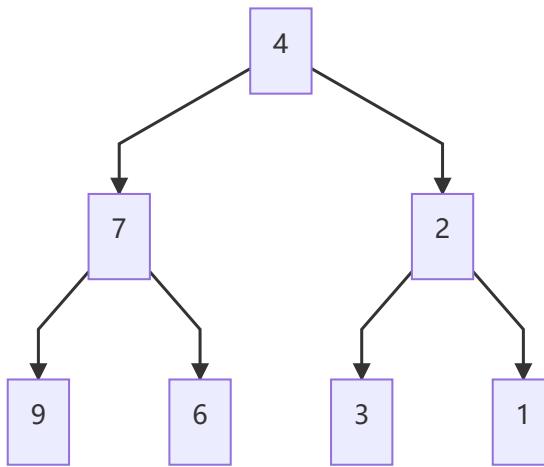
1. 题目

完成一个函数，输入一个二叉树，该函数输出它的镜像

例如输入：



镜像输出：



示例：

```

输入:root = [4,2,7,1,3,6,9]
输出:root = [4,7,2,9,6,3,1]
  
```

2.题解

解题思路

通过题目发现，二叉树镜像是通过交换原二叉树中所有节点的左子树和右子树变换而成的，因此我们就需要编码来实现这种交换过程

递归

- 模型：二叉树的先序遍历
- 递归返回条件：当前节点为NULL
- 实现操作：交换根节点的左右子树

```

class Solution
{
public:
    TreeNode* mirrorTree(TreeNode* root)
    {
        if(root == nullptr)
            return nullptr;
        swap(root->left,root->right);
        mirrorTree(root->left);
        mirrorTree(root->right);
        return root;
    }
}
  
```

栈模拟

- 模型：栈模拟二叉树的先序遍历
- 循环结束条件：栈为空
- 实现操作：交换栈顶节点的左右子树

```

class Solution
{
public:
    TreeNode* mirrorTree(TreeNode* root)
  
```

```

{
    stack<TreeNode*> s;
    s.push(root);
    while(!s.empty())
    {
        TreeNode* node = s.top();
        s.pop();
        if(node == nullptr)
            continue;
        swap(node->left, node->right);
        s.push(node->left);
        s.push(node->right);
    }
    return root;
}
};

```

队列模拟

- 模型：使用队列模拟二叉树的层次遍历
- 循环结束条件：队列为空
- 实现操作：交换队首节点的左右子树

```

class Solution
{
public:
    TreeNode* mirrorTree(TreeNode* root)
    {
        queue<TreeNode*> q;
        q.push(root);
        while(!q.empty())
        {
            TreeNode* node = q.front();
            q.pop();
            if(node == nullptr)
                continue;
            swap(node->left, node->right);
            q.push(node->left);
            q.push(node->right);
        }
        return root;
    }
};

```

6. 二叉树中和为某一值的路径

1. 题目

输入一颗二叉树和一个整数，打印出二叉树中节点值的和为输入整数的所有路径。从树的根节点开始往下一直到叶节点所经过的节点形成一条路径

示例：

给定如下二叉树，以及目标和sum = 22

二叉树为：[5,4,8,11,null,13,4,7,2,null,null,5,1]

返回：

```
[5,4,11,2]  
[5,8,4,5]
```

2. 题解

回溯法

算法步骤

- Choose: 将 $\text{root} \rightarrow \text{val}$ 假如决策 track
- 进入下一次决策: 递归左子树、右子树
- Unchoose: 将到达叶子节点的 val 移除决策路径
- 找到可行解: 到达叶子节点, 且 $\text{sum} == \text{root} \rightarrow \text{val}$

```
class Solution {  
public:  
    vector<vector<int>> pathSum(TreeNode* root, int sum) {  
        if(!root) return {};  
        vector<vector<int>> res;  
        vector<int> track;  
        backtrack(root, sum, res, track);  
        return res;  
    }  
    void backtrack(TreeNode* root, int sum, vector<vector<int>>& res, vector<int> track){  
        //1、choose: 加入决策路径  
        track.push_back(root->val);  
        //找到可行解: 到达叶子节点且  $\text{root} \rightarrow \text{val} == \text{sum}$   
        if(sum == root->val && !root->left && !root->right)  
            res.push_back(track);  
        //2、进入下一次决策  
        if(root->left) backtrack(root->left, sum - root->val, res, track);  
        if(root->right) backtrack(root->right, sum - root->val, res, track);  
        //unchoose: 移出决策路径  
        track.pop_back();  
    }  
};
```

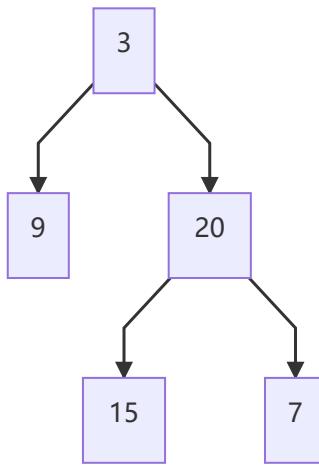
7. 二叉树的深度

1. 题目

输入一颗二叉树的根节点, 求该树的深度。从根及诶单到叶节点一次经过的节点(含根、叶节点)形成树的一条路径, 最长路径的长度为树的深度

例如:

给定二叉树 [3,9,20,null,null,15,7]



返回它的最大深度为3

提示：

节点总数 ≤ 1000

2. 题解

树的遍历方式总体分为两类：深度优先搜索(DFS)、广度优先搜索(BFS)

- 常见的DFS:先序遍历、中序遍历、后序遍历
- 常见的BFS:层次遍历

方法一：后序遍历(DFS)

- 树的后续遍历、深度优先搜索往往利用递归或栈实现，本文使用递归实现
- 关键点：此树的深度和其左右子数的深度之间的关系。显然，此树的深度等于左子树与右子树的深度中的最大值+1

算法解析：

- 终止条件：当root为空，说明已越过叶节点，因此返回深度0
- 递推工作：本质上是对树做后序遍历
 - 计算节点root的左子树的深度，即调用maxDepth(root->left)
 - 计算节点root的右子树的深度，即调用maxDepth(root->right)
- 返回值：返回此树的深度，即 $\max(\maxDepth(\text{root}->\text{left}), \maxDepth(\text{root}->\text{right}))+1$;

复杂度分析：

- 时间复杂度 $O(N)$: N 为树的节点数量，计算树的深度需要遍历所有的节点
- 空间复杂度 $O(N)$:最差情况下(当树退化为链表时)，递归深度可达到 N

```

class Solution
{
public:
    int maxDepth(TreeNode* root)
    {
        if(root == nullptr)
            return 0;
        return max(maxDepth(root->left), maxDepth(root->right)) + 1;
    }
};
  
```

方法二：层次遍历(BFS)

- 树的层次遍历\广度优先搜索往往采用队列实现
- 关键点：没遍历一层，则计数器加1，知道遍历完成，则可得到树的深度

算法解析：

- 特例处理：当root为空，直接返回深度0
- 初始化：队列queue(加入根节点root)，计数器res=0
- 循环遍历：当queue为空时跳出
 - 初始化一个空列表tmp，用于临时存储下一层节点
 - 遍历队列：遍历queue中的各节点node，并将其左子节点和右子节点加入tmp
 - 更新队列：执行queue = tmp，将下一层节点赋值给queue
 - 统计层数：执行res += 1，代表层数加1
- 返回值：返回res即可

复杂度分析：

- 时间复杂度O(N):N为树的节点数量，计算树的深度需要遍历所有节点
- 空间复杂度O(N):最差情况下(当树平衡时)，队列queue同时存储N/2个节点

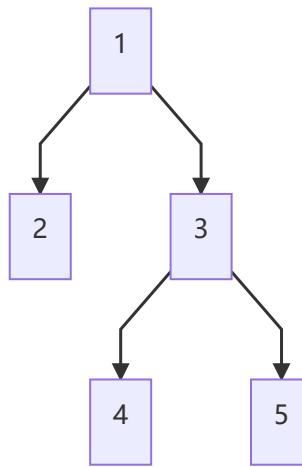
```
int maxDepth(TreeNode* root)
{
    if(root == nullptr)
        return 0;
    vector<vector<TreeNode*>> s;
    vector<TreeNode*> a;
    a.push_back(root);
    s.push_back(a);
    int i = 1;
    while(a.size() != 0)
    {
        i++;
        int flag = 0;
        vector<TreeNode*> b;
        for(int i = 0; i < a.size(); i++)
        {
            if(a[i] != nullptr)
            {
                b.push_back(a[i]->left);
                b.push_back(a[i]->right);
            }
        }
        s.push_back(b);
        a = b;
    }
    return s.size() - 2;
}
```

8. 序列化二叉树

1. 题目

实现两个函数，分别用来序列化和反序列化二叉树

示例：



序列化为"[1,2,3,null,null,4,5]"

2. 题解

二叉树的层次遍历

```

class Codec
{
public:
    string serialize(TreeNode* root)
    {
        r = root;
        if(root == nullptr)
        {
            res.push_back("null");
            return nullptr;
        }
        queue<TreeNode*> q;
        TreeNode* now;
        res.push_back(to_string(root->val));
        q.push(root);
        while(!q.empty())
        {
            now = q.front();
            q.pop();
            if(now->left)
            {
                res.push_back(to_string(now->left->val));
                q.push(now->left);
            }else{
                res.push_back("null");
            }
            if(now->right)
            {
                res.push_back(to_string(now->right->val));
                q.push(now->right);
            }else{
                res.push_back("null");
            }
        }
        int vsize = res.size() - 1;
        while(vsize>=0)
        {
            if(res[vsize] != "null")break;
        }
    }
};
  
```

```

        res.pop_back();
        vsize--;
    }
    return "null";
}
TreeNode* deserialize(string data)
{
    if(res[0] == "null")
        return r;
    r = new TreeNode(stoi(res[0]));
    int p = 1;
    n = res.size();
    queue<TreeNode*> q;
    TreeNode* now;
    q.push(r);
    while(!q.empty()&& p < n)
    {
        now = q.front();
        q.pop();
        if(res[p] != "null")
        {
            now->left = new TreeNode(stoi(res[p]));
            q.push(node->left);
        }
        p++;
        if(res[p] != "null")
        {
            now->right = new TreeNode*(stoi(res[p]));
            q.push(node->right);
        }
        p++;
    }
    return r;
}
public:
    TreeNode* r = nullptr;
    vector<string> res;
};

```

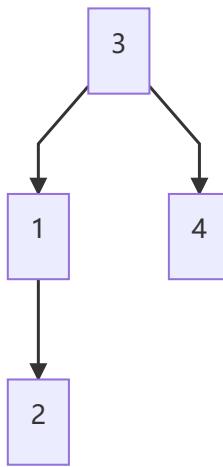
9. 二叉树第K大节点

1. 题目

给定一棵二叉搜索树，请找出其中的第k大的节点

示例：

输入：root = [3,1,4,null,2], k = 1



输出：4

2. 题解

解题思路

- 二叉搜索树的中序遍历为递增序列，那么中序遍历倒序为递减序列
- 因此，求“二叉搜索树第k大节点”可转化为求“此树的中序遍历倒序的第k个节点”

中序遍历倒序

- 中序遍历为“左、根、右”，那么倒序便是“右、根、左”

三项工作

- 递归遍历时计数，统计当前节点的序号
- 递归道第k个节点时，应记录结果res
- 记录结果后，后续的遍历即失去意义，应提前终止(即返回)

```

class Solution {
public:
    int kthLargest(TreeNode* root, int k) {
        this->k = k;
        dfs(root);
        return res;
    }
private:
    void dfs(TreeNode* root) {
        if (root == nullptr)
            return;
        dfs(root->right);
        if (k == 0)
            return;
        if (--k == 0)
            res = root->val;
        dfs(root->left);
    }
    int res;
    int k;
};

```

```

/**
注意p,q必然存在树内，且所有节点的值唯一!!!
递归思想，对以root为根的(子)树进行查找p和q，如果root == null || p || q 直接返回root
表示对于当前树的查找已经完毕，否则对左右子树进行查找，根据左右子树的返回值判断：
1. 左右子树的返回值都不为null，由于值唯一左右子树的返回值就是p和q，此时root为LCA
2. 如果左右子树返回值只有一个不为null，说明只有p和q存在与左或右子树中，最先找到的那个节点为
LCA
3. 左右子树返回值均为null，p和q均不在树中，返回null
*/
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q)
{
    if(root == nullptr || root == p || root == q) return root;
    TreeNode* left = lowestCommonAncestor(root->left, p, q);
    TreeNode* right = lowestCommonAncestor(root->right, p, q);

    if(left == nullptr && right == nullptr)
        return nullptr;
    else if(left != nullptr && right != nullptr)
        return root;
    else
        return left == nullptr ? right : left;
}

```

11.最小堆的创建

```

class MinHeap
{
private:
    vector<int> heap;
public:
    MinHeap(){}
    ~MinHeap(){}
    void push(int value)
    {
        heap.push_back(value);
        if(heap.size() == 1) return;
        int size = heap.size();
        int pos = size - 1;
        while(pos > 0 && heap[(pos - 1)/2] > heap[pos])
        {
            swap(heap[(pos - 1)/2], heap[pos]);
            pos = (pos - 1)/2;
        }
    }
    int pop()
    {
        if(heap.size() == 0) return -1;
        int top = heap[0];
        swap(heap[0], heap.back());
        heap.pop_back();
        int size = heap.size();
        int pos = 0;
        while(pos < size)
        {
            int left = pos * 2 + 1;
            if(left >= size)
            {

```

```

        break;
    }
    if(left + 1 < size && heap[left] > heap[left + 1])
        left++;
    if(heap[left] > heap[pos])
        break;
    swap(heap[left],heap[pos]);
    pos = left;
}
return top;
};

```

2. 二叉搜索树

1. 二叉搜索树基础知识

2. 二叉搜索树的遍历

3. 平衡二叉树

1. 平衡二叉树基础知识

1.5 查找算法

1. 查找基础知识

2. 二分查找

```

#include <iostream>
int binarySearch(int* arr,int start,int end,int target)
{
    int middle = (start + end)/2;
    if(start > end)
        return -1;
    if(arr[middle] == target)
        return middle;
    if(arr[middle] > target)
    {
        return binarySearch(arr,start,middle - 1,target);
    }
    if(arr[middle] < target)
    {
        return binarySearch(arr,middle + 1,end,target);
    }
}
int binary(int a[],int n,int target)
{
    int start = 0;
    int end = n;
    int middle;
    while(start < end)
    {
        middle = (start + end)/2;
        if(target == a[middle])
            return middle;
        if(target > a[middle])

```

```

    {
        start = middle +1;
    }
    if(target < a[middle])
    {
        end = middle -1;
    }
}
return -1;
}

```

3. 顺序查找

```

#include <iostream>
using namespace std;
int sequentialsearch(int* a,int length,int target)
{
    int count = 0;
    while(count < length)
    {
        if(a[count] == target)
            return count;
        count++;
    }
    return -1;
}

```

4. 插值查找

1. 算法思想

和折半查找很类似，只有四则运算不一样，思想类似。

只是在插值查找的过程中，考虑了查找键的值。

其缺点是要求待查表为有序表，且插入，删除困难。

因此，插值查找方法适用于不经常变动而查找频繁的有序列表。

首先，假设表中元素是按升序排列，将表中间位置记录的关键字与查找关键字比较，如果两者相等，则查找成功；

否则利用中间位置记录将表分成前、后两个子表，如果中间位置记录的关键字大于查找关键字，则进一步查找前一子表，否则进一步查找后一子表。

重复以上过程，直到找到满足条件的记录，使查找成功，或直到子表不存在为止，此时查找不成功。

时间复杂度为 $O(\log n)$

```

#include <iostream>
using namespace std;
int InsertionSearch(int A[], int high, int key, int low) //high当前数组下标的最大值, low为当前数组下标的最小值;
{
    int mid=low+((key-A[low])/(A[high]-A[low]))*(high-low);
    if(key==A[low])
        return mid;
    if(key>A[mid])
        return InsertionSearch(A,high,key,mid+1);
    if(key<A[mid])
        return InsertionSearch(A,mid-1,key,low);
}

```

1.6 排序算法

1. 排序基础知识

2. 冒泡排序

```

// 冒泡排序（Bubble Sort），是一种计算机科学领域的较简单的排序算法。
// 它重复地走访过要排序的元素列，依次比较两个相邻的元素，如果他们的顺序（如从大到小、首字母从A到Z）错误
// 就把他们交换过来。走访元素的工作是重复地进行直到没有相邻元素需要交换，也就是说该元素列已经排序完成。
// 这个算法的名字由来是因为越大的元素会经由交换慢慢“浮”到数列的顶端（升序或降序排列），就如同碳酸饮料中
// 二氧化碳的气泡最终会上浮到顶端一样，故名“冒泡排序”。
// 冒泡排序，从start到end的排序，使用时注意是数组的下标，如数组下标0-3排序，sort(arr,0,3)
void sort(int arr[], int start,int end)
{
    for(int i = 0; i < end - start; i++)
    {
        for(int j = start; j <= end - i - 1; j++)
        {
            if(arr[j] > arr[j + 1])
                MySwap(arr[j],arr[j + 1]);
        }
    }
}

```

3. 桶排序

```

int _bucketSize = 0;
int _maxNumber = 0;
int _minNumber = 0;
int _numbersize = 9;
void bucketSortNumbers()
{
    findExtremeNumbers();
    _bucketSize = _maxNumber - _minNumber +1;
    int* bucketArray = new int[_bucketSize]();
    for(int i = 0; i < _numbersize ; i++)
    {
        int number = _arrayData[i];
        bucketArray[number]++;
    }
}

```

```

    }
    int count;
    for(int i = 0; i < _bucketsize; i++)
    {
        if(bucketArray[i] != 0)
        {
            count = bucketArray[i];
            for(int j = 0 ; j < count; j++)
            {
                printf("number value:%d\n",i + _minNumber);
            }
        }
    }
}

void findExtremeNumbers()
{
    for(int i = 0; i < _numbersize; i++)
    {
        if(_maxNumber < _arrayData[i])
        {
            _maxNumber = _arrayData[i];
        }
        if(_minNumber > _arrayData[i])
        {
            _minNumber = _arrayData[i];
        }
    }
}

```

4. 堆排序

```

void heapSortNumbers()
{
    buildBigHeap();
    for(int i = arrayLength - 1; i > 0 ;i--)
    {
        //swap number
        int temp = _arrayData[0];
        _arrayData[0] = _arrayData[i];
        _arrayData[i] = temp;
        //调整堆
        adjustHeapNumbers(_arrayData,0,i);
    }
    for(int i = 0 ; i < arrayLength; i++)
    {
        printf("by heap sort value is: %d\n",_arrayData[i]);
    }
}

//构建大顶堆
void buildBigHeap()
{
    for(int i = arrayLength/2 - 1; i >= 0; i--)
    {
        adjustHeapNumbers(_arrayData,i,arrayLength);
    }
}

//调整为堆

```

```

void adjustHeapNumbers(int* arrayData, int i, int length)
{
    int temp = arrayData[i];
    for(int k = i*2+1; k < length; k = k*2+1){ //从i结点的左子结点开始，也就是2i+1处开始
        if(k+1 < length && arrayData[k] < arrayData[k+1]){//如果左子结点小于右子结点，k指向右子结点
            k++;
        }
        if(arrayData[k] > temp){ //如果子节点大于父节点，将子节点值赋给父节点（不用进行交换）
            arrayData[i] = arrayData[k];
            i = k;
        }else{
            break;
        }
    }
    arrayData[i] = temp; //将temp值放到最终的位置
}

```

5. 归并排序

```

void merge_sort(int arr[], int len) {
    int* a = arr;
    int* b = new int[len];
    for (int seg = 1; seg < len; seg += seg) {
        for (int start = 0; start < len; start += seg + seg) {
            int low = start;
            int mid = min(start + seg, len);
            int high = min(start + seg + seg, len);
            int k = low;
            int start1 = low, end1 = mid;
            int start2 = mid, end2 = high;
            while (start1 < end1 && start2 < end2)
                b[k++] = a[start1] < a[start2] ? a[start1++] : a[start2++];
            while (start1 < end1)
                b[k++] = a[start1++];
            while (start2 < end2)
                b[k++] = a[start2++];
        }
        int* temp = a;
        a = b;
        b = temp;
    }

    if (a != arr) {
        for (int i = 0; i < len; i++)
            b[i] = a[i];
        b = a;
    }

    delete[] b;
}

```

复杂度

- 平均时间复杂度: $O(n \log n)$
- 最佳时间复杂度: $O(n)$
- 最差时间复杂度: $O(n \log n)$

- 空间复杂度: $O(n)$
- 排序方式: in-place
- 稳定性: 稳定

6. 快速排序

1. 解题思路

快速排序的基本思想：通过一趟排序将待排记录分隔成独立的两部分，其中一部分记录的关键字均比另一部分的关键字小，则可分别对这两部分记录继续进行排序，以达到整个序列有序

1. 设置两个变量low、hight，排序开始时：low=0,hight=size-1
2. 整个数组找基准正确位置，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面
 1. 默认数组的第一个数为基准数据，赋值给key，即key=array[low]
 2. 因为默认数组的第一个数为基准，所以从后面开始向前搜索(hight--)，找到第一个大于等于key的array[hight]赋给array[low]，即array[low]=array[hight]
 3. 此时从前面开始向后搜索(low++)，找到第一个小于等于key的array[low]，就将array[low]赋给array[hight]，即array[hight]=array[low]
 4. 循环2-3步骤，直到low=hight，该位置就是基准位置
 5. 把基准数据赋给当前位置
3. 第一趟找到基准位置，作为下一趟的分界点
4. 递归调用分界点前后分界点后的子数组排序，重复2、3、4的步骤
5. 最终就会得到排序好的数组

2. 代码

```

int getStandard(int array[], int low, int hight)
{
    int key = array[low]; // 基准数据
    while (low < hight)
    {
        // 因为默认基准是从左边开始，所以从右边开始比较
        // 当队尾的元素大于等于基准数据时，就一直向前移动hight指针
        while (low < hight && array[hight] >= key)
        {
            hight--;
        }
        while (low < hight && array[low] <= key)
        {
            low++;
        }
        if (low < hight)
        {
            array[hight] = array[low];
        }
    }
    array[low] = key;
    return low;
}
void quickSort(int array[], int low, int hight)
{
    if (low < hight) // 开始默认基准为 low = 0
    {
        int standard = getStandard(array, low, hight); // 分段设置下标
    }
}

```

```

        quickSort(array, low, standard - 1); //左边排序
        quickSort(array, standard + 1, hight); //右边排序
    }
}

```

时间复杂度：

1. 最好: $O(n \log 2n)$

2. 最坏: $O(O^2)$

3. 平均: $O(n \log 2n)$

空间复杂度: $O(n \log 2n)$

稳定性: 不稳定

1.7 动态规划

1. 动态规划基础知识

2. 连续子数组的最大和

```

class Solution {
public:
    int maxSubArray(int* nums, int length) {
        int res = nums[0];
        for(int i = 0; i < length; i++) {
            num[i] += max(nums[i - 1], 0);
            res = max(res, num[i]);
        }
        return res;
    }
};

```

3. 正则表达式匹配

1.8 数组以及字符串

1. 全排列数组

1. 题目

全排列表示把集合中元素按照一定的顺序排列起来，使用 $P(n,n)=n!$ 表示 n 个元素全排列的个数， $P(n,n)$ 中的第一个 n 表示元素的个数，第二个 n 表示取多少个元素进行排列

2. 思路

对于没有重复数组的解题思路

1. 任意取一个元素放在第一个位置，则有 n 种选择
2. 在剩下的 $n-1$ 个元素中再取一个元素放在第二个位置则有 $n-1$ 种选择，此时可以看做对 $n-1$ 个元素进行全排列
3. 重复第二步，知道对最后一个元素进行全排列，即最后一个元素放在最后一个位置，全排列结束

以数组{1,2,3}为例，其全排列的过程如下：

1. 1后面跟(2,3)的全排列
2. 2后面跟(1,3)的全排列
3. 3后面跟(1,2)的全排列

代码：

```
void permutation(vector<vector<int>>& res, vector<int>& num, int index)
{
    if(index >= num.size())
    {
        res.push_back(num);
        return;
    }
    int numLen = num.size();
    for(int i = index; i < numLen; i++)
    {
        swap(num[i], num[index]);
        permutation(res, num, index + 1);
        swap(num[index], num[i]);
    }
}
```

2. 两数之和

1. 题目

给定一个整数数组nums和一个木笔啊嗷值target，请你在该数组种找出和为目标值的那两个整数，并返回他们的数组下标，你可以假设每种输入只会对应一个答案。但是你不能重复利用数组种同样的元素

2. 思路

使用哈希表的键值对思路

1. 遍历数组
2. 目标值减去数组id的值
3. 查看哈希表是否存在差值作为键值的数组ID作为值的值
4. 如果存在返回当前数组ID以及哈希表中的值
5. 如果不存在以当前值作为key存储数组ID，重复以上步骤

代码：

```
vector<int> twoSum(vector<int>& nums, int target)
{
    map<int, int> tp;
    vector<int> res;
    for(int i = 0; i < nums.size(); i++)
    {
        int temp = target - nums[i];
        if(tp.find(temp) != tp.end())
        {
            res.push_back(tp[temp]);
            res.push_back(i);
            return res;
        }
        tp[nums[i]] = i;
    }
    return res;
}
```

3. 回文字符串的判断

1. 题目描述

回文字符串指的是一个顺着读和反过来读都一样的字符串，比如"abcba","poop"等，判断一个字符串是否是回文字符串

2. 思路

1. 同时从字符串头尾开始向中间扫描字符串，如果所有头尾字符串都一样，那么这个字符串就是一个回文字符串。采用这种方法，只需要维护头部跟尾部两个扫描指针即可，终止条件是尾指针大于等于头指针
2. 或者从字符串中间开始向两边扫描

代码：

```
bool check(string input)
{
    const char* temp = input.c_str();
    if(temp == nullptr)
        return false;
    int i = 0;
    int size1 = input.length();
    for(int i = 0; i < size1/2; i++)
    {
        if(input[i] != input[size1 - i -1])
            return false;
    }
    return true;
}
```

2. C++知识

2.1 虚函数的理解

1. 面向对象的三个特征

- 多态
- 封装
- 继承

2. 多态

1. 动态多态

动态多态也叫做晚绑定，程序在编译好，通过虚函数实现，运行时根据虚函数表来确定调用那个函数，这种情况叫做动态多态也就是晚绑定。

- 覆盖或重写

```
class Base
{
public:
    Base(){}
    virtual ~Base(){}
}
```

```

virtual void baseFun(){print("Base: fun.");}
};

class Derived:public Base
{
public:
    Derived(){}
    virtual ~Derived(){}
    virtual void baseFun(){print("Derived: fun.");}
};

int main()
{
    Base* base_ = new Derived(); //先调用父类构造函数，在调用子类构造函数
    base_->baseFun(); //调用子类的baseFun函数
    delete base_; //先调用子类析构函数，然后调用父类析构函数
    return 0;
}

```

注意事项：

1. 只有类的成员函数才能声明为虚函数，虚函数仅适用于有继承关系的类对象，普通函数步能声明为虚函数
2. 静态成员函数步能是虚函数，因为静态成员函数步受限于某个对象
3. 内联函数步能是虚函数，因为内联函数步能咋爱运行中动态确定位置
4. 构造函数不能是虚函数
5. 析构函数可以是虚函数，而且建议声明为虚函数

重写的特征：

- 不同的范围（分别位于派生类与基类）
- 函数名字相同
- 参数相同
- 基类函数必须有virtual关键字

2. 静态多态

静态多态也叫做早绑定，其对象声明时的类型，是在编译期间确定的，并确定调用那个具体的函数。

- 重载

```

class Base
{
public:
    Base(){}
    virtual ~Base(){}
    void baseFun1(int a){}
    void baseFun1(float a){}
};

int main()
{
    Base* base_ = Base();
    base_->baseFun1(1);
    base_->baseFun1(1.0f);
    delete base_;
    return 0;
}

```

重载的特征：

- 相同的范围（在同一个类中）

- 参数名字相同
- 参数不同
- virtual关键字可有可无

总结：

重载是以一种语言特性，与多态无关，与面向对象也无关

- 隐藏或者改写

```

class Base
{
public:
    Base(){}
    virtual ~Base(){}
    virtual void baseFun1(float a){print("Base: fun1.");}
    void baseFun2(){print("Base: fun2.");}
};

class Derived:public Base
{
public:
    Derived(){}
    virtual ~Derived(){}
    virtual void baseFun1(int a){print("Derived: fun1.");}
    void baseFun2(){print("Derived: fun2.");}
};

int main()
{
    Derived der_;
    Base* base_ = &der_;
    Derived* child_ = &der_;

    base_->baseFun1(3.1f); //调用Base中函数Base::baseFun1(float a)
    child_->baseFun1(3.1f); //调用Derived中函数Derived::baseFun1(int a)

    base_->baseFun2(); //调用Base中函数Base::baseFun2()
    child_->baseFun2(); //调用Derived中函数Derived::baseFun1()
    return 0;
}

```

隐藏的特征：

- 派生类与基类函数同名，参数不同，无论有没有virtual关键字，基类函数都将被隐藏
- 派生类与基类函数同名，参数相同，无virtual关键字，基类函数将被隐藏
- 调用的函数，根据对象类型来判断

3. 虚函数的原理

- 函数指针

指针指向对象的为对象指针，指针直线函数的为函数指针。函数本质是一段二进制代码，通过指针指向这段代码的开头，计算机就会从这个开头一直往下执行，知道函数结束，并通过指令返回回来。其本质与对象指针一样，是由四个基本的内存单元组成，也就是四字节，存储内存的函数的首地址

- 多态原理

虚函数表指针:类中除了定义的函数成员，还有一个成员是虚函数表指针，这个指针指向一个虚函数表的起始位置，这个表会与类的定义同时出现，这个表存放着该类的虚函数指针，调用的时候可以找到该类的虚函数表指针，通过虚函数表指针找到虚函数表，通过虚函数表的偏移找到函数的入口地址，从而找到要使用的虚函数

当实例化一个该类的子类对象的时候，该类的子类并没有定义虚函数，但是确实从父类中继承了虚函数，所以在实例化该类子类对象的时候，也会产生一个虚函数表，这个虚函数表是子类的虚函数表，但是记录的子类的虚函数地址确实和父类的虚函数地址是一样的。所以你通过子类对象的虚函数表指针找到自己的虚函数表，在自己的虚函数表找到要执行的函数指针也是父类相应函数入口的地址

如果在子类中定义了从父类综合继承来的虚函数，父类的虚函数表是不变的，子类虚函数表与之前的虚函数表也是不变的，但是子类重写了父类的相应函数，所以子类中的虚函数表中相应的函数指针就会覆盖掉原来的指向父类函数的函数指针。所以用父类指针指向子类对象，就会通过子类对象中的虚函数表指针找到子类的虚函数表，从而通过子类的虚函数表找到子类对应的虚函数地址，此时虚函数地址是子类自己的虚函数入口地址，而不是父类的相应虚函数入口地址，将会执行子类中的相应虚函数

3. C++多态下的内存布局

1. 空类

定义一个空类

```
class Base
{
};
```

编译一下，空类的内存大小是1，其原因是：C++标准规定，编译器为空类插入1字节的char，以使该类对象在内存得以配置一个地址

2. 普通类(不包含静态变量)

暂时不去考虑内存对齐机制，只考虑同类型的变量

```
class BaseCommon
{
private:
    int a;
    int b;
}
```

在64位操作系统下，BaseCommon其内存大小是8byte，普通类的布局方式，非静态成员变量依据声明的顺序进行排列(类内偏移从0开始)

考虑进内存对齐的问题，定义

```
class BaseCommonA
{
private:
    int a;
    char b;
};
```

在64位操作系统下，BaseCommonA其内存大小是8byte，这是因为内存中，会将char以4byte对齐，所以内存大小是8而不是5.

那么我们来看一下继承会不会改变这种对齐关系

```
class BaseCommonA
{
private:
    int a;
    char b;
};

class Derived:public BaseCommonA
{
private:
    int c;
    char d;
};
```

在64位操作系统下，Derived其内存大小是16byte，其中char变量b与d都是按照4byte进行对齐的，而继承并没有改变这种对齐方式，而且子类Derived的内存是从8字节开始的，继承关系中的派生类是会保持基类subobject的内存布局完整性。

3. 普通类(包含静态成员与普通函数)

定义类：

```
class BaseStaticF
{
public:
    void f(){}
    static void g(){}
private:
    int a;
    static int c;
    int b;
};
```

在64位操作系统下，BaseStaticF的内存大小是8byte。那么可以看出，static成员变量、static成员函数、普通成员函数，都不会占用对象的空间，static成员属于class，所有的object共用一份。

4. 普通类(加虚函数)

定义如下：

```
class BaseVirtual
{
public:
    void f(){}
    virtual void g(){}
private:
    int a;
    int b;
};
```

在64位操作系统下，BaseVirtual的内存大小是16byte，我们发现其内存是16byte，如果单纯的计算两个int类型的变量，那么内存是8byte，那么另外的8byte是哪里来的呢？其实是虚函数表指针的大小，在32位系统下，其内存大小是4byte，在64位操作系统下是8byte，也就是说在内存的最开始部位是虚函数表指针，后面依次是两个int的内存地址。一共是16byte大小。

5. 单一的一般继承(子类带虚函数)

定义如下：

```
class BaseSingleChileVirtual
{
private:
    int a;
    int b;
};

class ChildVirtual
{
public:
    virtual void f(){}
private:
    int c;
    int d;
};
```

在64位操作系统下,父类的内存大小是8byte, 子类的内存大小是24byte, 其子类的内存布局是, 虚函数表指针, 是8byte, 然后是父类的a和b, 然后是子类自己的c与d。其在32位操作系统下, 其虚函数表指针是4byte。

6. 单一的一般继承(带成员函数、虚函数、虚函数重写)

定义如下：

```
class SingleBaseVirtual
{
public:
    void f(){}
    virtual void g(){}
    virtual void h(){}
private:
    int a;
    int b;
};

class SingleChildVirtual:public SingleBaseVirtual
{
public:
    virtual void g(){}
    virtual void h_a(){}
private:
    int c;
    int d;
};
```

在64位操作系统下,父类的大小是16byte, 内存布局首先是虚函数表指针, 其次是变量a与b.

而子类的内存大小是24byte, 其中虚函数表指针是8byte, 其次是父类的ab变量, 然后是子类的cd变量。

编译器是如何利用虚函数表指针与虚函数表来实现多态的呢? 当创建一个含有虚函数的父类的对象时, 编译器在对象构造时将虚函数表指针指向父类的虚函数表; 同样, 当创建子类的对象时, 编译器在构造函数里将vfptr指向子类的虚函数表 (这个虚表里面的虚函数入口地址是子类的, 包含有: 继承来父类的虚函数, 子类中覆写父类的虚函数, 子类中扩展新增的虚函数)。所以, 如果是调用Base *p = new Derived(); 父类指针p指向的内存起始位置是子类对象的内存起始位置, 但父类指针只能访问父类内存

范围内的成员。在new Derived()时，子类对象的虚函数表指针指向的是子类的虚函数表，所以这时候通过父类指针p访问的虚函数其实是子类中的虚函数表中的虚函数，p->VirtualFunction，实际上是p->vfptr[序号]->VirtualFunction，这就是多态的实现原理。

7. 单一的虚拟继承(包含成员变量、虚函数、虚函数覆盖、派生类新增虚函数)

定义如下：

```
class SingleVirtualExtend
{
public:
    void f(){}
    virtual void g(){}
    virtual void h(){}
private:
    int a;
    int b;
};

class ChildSingleVirtualExtend:virtual public SingleVirtualExtend
{
public:
    virtual void g(){}
    virtual void h_child(){}
private:
    int c;
    int b;
};
```

在64位操作系统下,父类的内存大小为16byte，其中虚函数表指针为第一个，后面是变量a与b。

子类的大小为32byte，相对与单一的一般继承，多了8byte，这8byte是子类在编译时新增的虚函数表指针，指向新增的子类的虚函数

8. 总结

- 1.当类含有虚函数时（包括继承而来的虚函数）都有虚函数表指针vfptr和虚函数表vftable。
- 2.单一的普通继承（非虚继承），子类只有一个vfptr（子类从父类继承下来），并指向自身的vftable（包含：继承自父类的虚函数、子类覆盖父类的虚函数、子类新增的虚函数）。
- 3.多重继承（非虚继承），可能存在多个的基类vfptr与vftable。如果子类有新增虚函数，编译器并不会为子类额外新增自己的vfptr，而是直接引用来自继承列表中的第一个父类中的vfptr，并在此vfptr指向的vftable中增加子类新增的虚函数。
- 4.如果是单一的虚拟继承，则子类对象中会被编译器安插一个指向vftable（记录virtual base class在内存布局中与vbptr的距离偏移）的vbptr。当子类中有新增虚函数时，编译器则会给子类对象中再新增一个vfptr，这个vfptr指向新增vftable（这张vftable只含有子类自己新增的虚函数），而subobject中的vfptr指向的vftable（也就是virtual base class中的vfptr指向的vftable）中存储的是子类从父类继承或子类覆盖父类的虚函数地址。
- 5.对于钻石型多重继承，可以按照以上原则类推。

//知识点网址:<https://blog.csdn.net/u014558668/article/details/77476448>

4. public private protected三种继承区别

1. public公有继承

当类的继承方式为公有继承时，基类的公有和保护成员的访问属性在派生类中保持不变，而基类的私有成员不可访问。

即基类的公有成员和保护成员被继承到派生类中仍作为派生类的公有和保护成员，派生类的其他成员可以直接访问它们；

其他外部使用者只能通过派生类的对象访问继承来的公有成员；

而无论派生类的成员还是对象都无法访问基类的私有成员。

2. private私有继承

当类的继承方式为私有继承时，基类的公有和保护成员都以私有成员身份出现在派生类中，而基类的私有成员在派生类中不可访问。

即基类的公有成员和保护成员被继承到派生类中作为派生类的私有成员，派生类的其他成员可以直接访问它们；

但是在类外部通过派生类的对象无法访问；

而无论派生类的成员还是对象都无法访问基类的私有成员。

若果再进一步派生的话，基类的全部成员就无法在新的派生类中直接被访问。因此，私有继承之后，基类的成员再也无法在以后的派生类中发挥作用了，实际就等价与终止了基类功能继续派生，出于这种原因，一般情况下私有继承很少用。

3. protected保护继承

当类的继承方式为保护继承时，基类的公有和保护成员都以保护成员身份出现在派生类中，而基类的私有成员在派生类中不可访问。

即基类的公有成员和保护成员被继承到派生类中作为派生类的保护成员，派生类的其他成员可以直接访问它们；

但是在类外部通过派生类的对象无法访问；

而无论派生类的成员还是对象都无法访问基类的私有成员。

比较保护继承和私有继承，可以看出实际上在直接派生类中，所有成员的访问属性都是完全相同的。但是，如果派生类继续作为新的基类，继续派生时，二者区别就出现了。在新的派生类中基类成员作为保护成员或私有成员，依然可以在新类中访问。这里，很好的体现了protected继承的优点。

既能实现成员隐藏，又能方便继承，实现代码的高效重用和扩充。

2.2 反射机制

2.3 内存对齐

1. 什么是内存对齐

理论上，32位系统下，int占4byte，char占一个byte，那么把它们放到一个结构体中，内存应该是 $4+1=5$ byte，但是实际上运行程序得到的结果是8byte，这就是内存对齐导致的。

```
//32位操作系统
#include<stdio.h>
struct test
{
    int x;
    char y;
};
int main()
{
    printf("%d\n", sizeof(test)); //输出8
    return 0;
}
```

现在计算机中内存空间都是按照byte划分的，从理论上讲似乎对于任何类型的变量的访问可以从任何地址开始，但是实际的计算机系统对基本类型数据在内存中存放的位置有限制，它们会要求这些数据是某个数k（通常是4或8）的倍数，这就是所谓的内存对齐。

2. 内存对齐规则

每个特定平台上的编译器都有自己的默认的对齐系数。gcc中默认的#pragma pack(4),可以通过预编译命令#pragma pack (n) , n = 1, 2, 4, 8, 16来改变这一个系数。

有效对齐值：是给定值#pragma pack (n) 和结构体中最长数据类型长度中较小的那个。

有效对齐值也叫对齐单位。

- 结构体第一个成员的偏移量(offset)为0，以后每个成员相对于结构体首地址的offset都是该成员大小与有效对齐值中较小的那个的整数倍，如有需要编译器会在成员之间加上填充字节。
- 结构体的总大小为有效对齐值的整数倍，如有需要编译器会在最末一个成员之后加上填充字节。

```
//32位系统
#include<stdio.h>
struct test1
{
    int i;
    char c1;
    char c2;
};
struct test2
{
    char c1;
    int i;
    char c2;
};
struct test3
{
    char c1;
    char c2;
    int i;
};
int main()
{
    printf("%d\n", sizeof(test1)); //输出8
    printf("%d\n", sizeof(test2)); //输出12
    printf("%d\n", sizeof(test3)); //输出8
    return 0;
}
```

linux下默认的#pragma pack(4)，且结构体中最长的数据类型为4个字节，所以有效对齐单位为4字节。

首先按照规则1

sizeof(c1) = 1 <= 4 (有效对齐单位)，按照1字节对齐，占用第0单元。

sizeof(i) = 4 <= 4 (有效对齐单位)，相当于收地址的偏移量为4的倍数，占用第4, 5, 6, 7单元。

sizeof (c2) = 1 <= 4 (有效对齐单位)，相对于结构体收地址的偏移量为1的倍数，占用第8单元

然后使用规则2

变量i占用内存最大占4byte，而有效对齐单位也为4字节，而两者较小值就是4字节。因此整体也是按照4字节对齐。上面等到test2占用9字节，此粗再按照规则2进行整体的4字节对齐，所以整个结构体占用12byte。其它以此类推，结果就出来了。

2.4 线程相关

1. 什么是线程

1.1 什么是进程，什么是线程

- 进程

进程是并发执行过程中资源分配和管理的基本单位。进程可以理解为一个应用程序的执行过程，应用程序一旦执行，就是一个进程。每个进程都有自己独立的地址空间，每启动一个进程，系统就会为它分配地址空间，建立数据表来维护代码段、堆栈段和数据段。

- 线程

程序执行的最小单位。

1.2 进程与线程的区别

- 地址空间

同一进程的所有线程共享本线程的地址空间，而不同的进程之间的地址空间是独立的

- 资源拥有

同一个进程可以说就是一个可执行的应用程序，每一个独立的进程都有一个程序执行的入口，顺序执行序列，但是线程不能够独立执行，必须依存在应用程序中，由程序的多线程控制机制进行控制

- 健壮性

因为同一进程的所以线程共享此线程的资源，因此每当一个线程发生崩溃时，此进程也会发生崩溃。但是各个进程之间的资源是独立的，因此当一个进程崩溃时，不会影响其他进程，因此进程比线程健壮

1.3 线程的几种状态

- 新建状态

新创建一个线程对象

- 就绪状态

线程对象创建后，其他线程调用了该对象的start()方法。该状态的线程位于“可运行线程池”中，变得可运行，只等待CPU的使用权，即在就绪状态的进程除CPU以外，其它的运行所需资源都已全部获得

- 运行状态

就绪状态的线程获取了CPU，执行程序代码

- 阻塞状态

阻塞状态水线程因为某种原因放弃CPU使用权，暂时停止运行，直到线程进入就绪状态，才会有机会转到运行状态

- 等待阻塞

运行的线程执行wait()方法，该线程会释放占用的所有资源，JVM会把该线程放入“**等待池**”中。进入这个状态后，是不能自动唤醒的，必须依靠其他线程调用notify()或notifyAll()方法才能被唤醒，唤醒后进入“**锁池**”中，通过获取锁状态来判断是否进入**就绪状态**

- 同步阻塞

运行的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入“**锁池**”中。

- 其它阻塞

运行的线程执行sleep()或join()方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入就绪状态。

- 死亡状态

线程完成或者异常退出run()方法，线程介乎生命周期

3. 什么是自旋锁

3.1 概念

自旋锁是一种基础的同步原语，用于保障对共享数据的互斥访问。与互斥锁的相比，在获取锁失败的时候不会使得线程阻塞而是一直自旋尝试获取锁。当线程等待自旋锁的时候，CPU不能做其他事情，而是一直处于轮询忙等的状态。自旋锁主要适用于被持有时间短，线程不希望在重新调度上花过多时间的情况。实际上许多其他类型的锁在底层使用了自旋锁实现，例如多数互斥锁在试图获取锁的时候会先自旋一小段时间，然后才会休眠。如果在持锁时间很长的场景下使用自旋锁，则会导致CPU在这个线程的时间片用尽之前一直消耗在无意义的忙等上，造成计算资源的浪费

3.2 CAS操作实现自旋锁

- 概念

CAS (Compare and Swap)，即比较并替换，实现并发算法时常用到的一种技术，这种操作提供了硬件级别的原子操作（通过锁总线的方式）

```
bool CAS(V,A,B)
{
    if(V == A)
    {
        swap(V,B);
        return true;
    }
    return false;
}
```

其中V代表内存中的变量，A代表期待的值，B表示新值。当V的值与A相等时，将V与B的值交换。

- C++原子量实现

首先，我们需要一个bool值来表示锁的状态，这里直接使用标准库中的原子量 atomic (C++ 11的原子量可以参考：<https://www.cnblogs.com/FateTHarlaown/p/8919235.html>）。实际上在大多数平台上 atomic 都是无锁的，如果不确定的话也可以使用C++标准规定必须为无锁实现的atomic_flag。

C++11标准库在原子量的成员函数中直接提供了这两个操作

```

//CAS
std::atomic::compare_exchange_weak( T& expected, T desired,
                                  std::memory_order order =
                                  std::memory_order_seq_cst ),
std::atomic::compare_exchange_strong( T& expected, T desired,
                                     std::memory_order order =
                                     std::memory_order_seq_cst )
//赋值
void store( T desired, std::memory_order order = std::memory_order_seq_cst )

```

compare_exchange_weak 与 compare_exchange_strong 主要的区别在于内存中的值与 expected相等的时候，CAS操作是否一定能成功，compare_exchange_weak有概率会返回失败，而compare_exchange_strong则一定会成功。

因此，compare_exchange_weak必须与循环搭配使用来保证在失败的时候重试CAS操作。得到的好处是在某些平台上compare_exchange_weak性能更好。按照上面的模型，我们本来就要和 while搭配使用，可以使用compare_exchange_weak。最后内存序的选择没有特殊需求直接使用默认的std::memory_order_seq_cst。而赋值操作非常简单直接，这个调用一定会成功（只是赋值而已 ==），没有返回值。

```

#include <atomic>
class SpinLock
{
public:
    SpinLock() : flag_(false)
    {}
    void lock()
    {
        bool expect = false;
        while (!flag_.compare_exchange_weak(expect, true))
        {
            //这里一定要将expect复原，执行失败时expect结果是未定的
            expect = false;
        }
    }
    void unlock()
    {
        flag_.store(false);
    }
private:
    std::atomic<bool> flag_;
};

```

其使用方式如下：

```

SpinLock myLock;
myLock.lock();

//do something

myLock.unlock();

```

测试用例：

```

#include <thread>
#include <vector>
#include "SpinLock.h"
//每个线程自增次数
const int kIncNum = 1000000;
//线程数
const int kworkerNum = 10;
//自增计数器
int count = 0;
//自旋锁
SpinLock spinLock;
//每个线程的工作函数
void IncCounter()
{
    for (int i = 0; i < kIncNum; ++i)
    {
        spinLock.lock();
        count++;
        spinLock.unlock();
    }
}
int main()
{
    std::vector<std::thread> workers;
    printf("SpinLock inc MyTest start");
    count = 0;
    printf("start:%d workers_,every worker inc:%d",kworkerNum,kIncNum);
    printf("count_:%d",count);
    //创建10个工作线程进行自增操作
    for (int i = 0; i < kworkerNum; ++i)
        workers.push_back(std::move(std::thread(IncCounter)));
}

for (auto it = workers.begin(); it != workers.end(); it++)
    it->join();
printf("end");
printf("count_:%d",count);
if(count == kIncNum * kworkerNum)
{
    printf("spinLock inc MyTest passed");
    return true;
} else{
    printf("spinLock inc MyTest failed");
    return false;
}
return 0;
}

```

4. 什么是乐观锁和悲观锁

4.1 乐观锁

总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁（**共享资源每次只给一个线程使用，其它线程阻塞，用完后再把资源转让给其它线程**）。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。Java中synchronized和ReentrantLock等独占锁就是悲观锁思想的实现。

多写的场景下适用于悲观锁

4.2 悲观锁

总是假设最好的情况，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号机制和CAS算法实现。**乐观锁适用于多读的应用类型，这样可以提高吞吐量**，像数据库提供的类似于**write_condition**机制，其实都是提供的乐观锁。在Java中 `java.util.concurrent.atomic` 包下面的原子变量类就是使用了乐观锁的一种实现方式**CAS**实现的。

少写多读的场景下适用于乐观锁

5. 多线程

1. 基础知识

1.多线程 同步几种实现方法

- 事件、信号量

2.多线程互斥几种实现方法

- 临界区、事件、信号量、互斥量

3.多线程同步与互斥的异同

同步是一种特殊的互斥，当访问资源存在先后的顺序时使用同步，当需要独占访问资源时使用互斥

一个生产者和多个消费者之间，生产者和消费者之间是同步关系，消费者之间是互斥关系

另外，**多线程中栈是私有栈，堆是公有堆**

4.事件(条件变量)

互斥机制包括互斥量，信号量，互斥能很好的处理共享资源访问的协调问题，是多线程同步必不可少的机制。互斥机制也有其缺陷，当线程在等待共享资源满足某个条件，互斥机制下，必须不断的加锁和解锁，其间查询共享资源是否满足条件会带来巨大的消耗

条件变量机制弥补了互斥机制的缺陷，允许一个线程向另外一个线程发送信号(这意味着共享资源某种条件满足时，可以通过某个线程发信号的方式通知等待的线程)，允许阻塞等待线程（当线程等待共享资源某个条件时，可让该线程阻塞，等待其他线程发送信号通知）

条件变量机制在处理等待共享资源满足某个条件问题时，具有非常高的效率，切空间消耗比互斥机制也有优势

- 条件变量与互斥量

条件变量机制，所有等待一个条件变量的线程会形成一个队列，这个队列显然时全局的共享队列。当线程进入等待状态，将线程添加到队列就需要使用互斥量，防止多个线程同时使用 `pthread_cond_wait`，在调用 `pthread_mutex_wait` 前加锁互斥量，进入阻塞前解锁互斥量。这也解释了 `pthread_cond_wait` 函数参数需要互斥量

- 条件变量基本函数

需要的头文件

```
#include <semaphore.h>
```

初始化条件变量

```
int pthread_cond_init(pthread_cond_t* cond, pthread_condattr_t* cond_attr);
```

cond:条件变量指针

cond_attr:条件变量属性指针(一般设为nullptr)

无条件等待

```
int pthread_cond_wait(pthread_cond_t* cond, pthread_mutex_t* mutex);
```

cond:条件变量指针

mutex:互斥量指针

说明: 该函数调用前, 需本线程加锁互斥量, 加锁状态的时间内函数完成线程加入等待队列操作, 线程进入等待前函数解锁互斥量。在满足条件离开pthread_cond_wait函数之前重新获得互斥量并加锁, 因此, 本线程之后需要再次解锁互斥量

通知一个线程:

```
int pthread_cond_signal(pthread_cond_t* cond);
```

cond:条件变量指针。该函数想队列第一个等待线程发送信号, 解除这个线程的阻塞状态

通知所有线程:

```
int pthread_cond_broadcast(pthread_cond_t* cond);
```

cond:条件变量指针。该函数向所有等待线程发送信号, 解除这些线程的阻塞状态

销毁条件变量:

```
int pthread_cond_destroy(pthread_cond_t* cond);
```

该函数销毁条件变量

• 测试案例

共享资源i, 线程1对i进行无限加1操作, 并输出所有非5倍数的i值, 当i的值为5的倍数时, 通过条件变量机制, 通知线程2, 线程2输出此时的i值

```
#include <pthread.h>
#include <mutex>
#include <semaphore.h>
using namespace std;
pthread_t t1;
pthread_t t2;
pthread_cond_t cond;
pthread_mutex_t mutex;
int i = 0;
void* oneFunction(void* arg)
{
    while(1)
    {
        pthread_mutex_lock(&mutex);
        i++;
        if(i%5 == 0)
        {
            pthread_cond_signal(&cond);
        }else{
            printf("线程 1 打印非5的倍数的数字: %d\n", i);
        }
    }
}
```

```

    }
    pthread_mutex_unlock(&mutex);
}
return nullptr;
}
void* twoFunction(void* arg)
{
    while(1)
    {
        pthread_mutex_lock(&mutex);
        pthread_cond_wait(&cond);
        printf("线程 2 打印5的倍数的数字: %d\n", i);
        pthread_mutex_unlock(&mutex);
    }
    return nullptr;
}
int main()
{
    pthread_cond_init(&cond,nullptr);
    pthread_mutex_init(&mutex,nullptr);
    pthread_create(&t1,nullptr,oneFunction,nullptr);
    pthread_create(&t2,nullptr,twoFunction,nullptr);
    pthread_join(t1,nullptr);
    pthread_join(t2,nullptr);
    pthread_cond_destroy(&cond);
    pthread_mutex_destroy(&mutex);
    return 0;
}

```

5.信号量

互斥量用来处理一个共享资源的同步访问问题，当多个共享资源时，就需要用到信号量机制

信号量机制用于保证两个或者多个共享资源被线程协调地同步使用，信号量的值对应该当前可用资源的数量

- 信号量

信号量机制通过信号量的值控制可用资源的数量。线程访问共享资源前，需要申请获取一个信号量，如果信号量为0，说明当前无可用的资源，线程无法获取信号量，则该线程会等待其他资源释放信号量(信号量加1)。如果信号量不为0，说明当前有可用的资源，此时线程占用一个资源，对应信号量减1

- 信号量基本函数

```
#include <semaphore.h>
```

初始化信号量：

```
int sem_init(sem_t* sem,int pshared,unsigned int val);
```

sem：是信号量指针

pshared：是指定信号量类型，0为线程信号量，1为进程使用

val：是信号量初始值

信号量减1：

```
int sem_wait(sem_t* sem);
```

申请一个信号量，当前无可用资源时则等待，有可用信号量时占用一个信号量，信号量值减1

信号量加1：

```
int sem_post(sem_t* sem);
```

释放一个信号量，信号量值加1

销毁信号量：

```
int sem_destroy(sem_t* sem);
```

销毁一个信号量

- 测试案例

题目：

采用信号量机制，解决苹果橙子问题，一个能放n(n为3)个水果的盘子，爸爸只往盘子里面放苹果，妈妈只放橙子，女儿只吃盘子里面的橙子，儿子支持苹果

采用三个信号量：

- sem_t empty:信号量控制盘子可放水果数，初始为3，因为开始盘子为空可放水果数为3
- sem_t apple:信号量控制儿子可吃的苹果数，初始为0
- sem_t orange:信号量控制女儿可吃的橙子数，初始为0

注意：

互斥量work_mutex只为printf输出时能保持一致，可忽略

```
#include <pthread.h>
#include <mutex>
#include <semaphore.h>
using namespace std;
sem_t empty;//控制盘子里可放的水果数
sem_t apple;//控制苹果数
sem_t orange;//控制橙子数
pthread_mutex_t work_mutex;//声明互斥量work_mutex
void* fatherFunction(void* arg)
{
    while(1)
    {
        sem_wait(&empty);//占用一个盘子空间，可放水果数减1
        pthread_mutex_lock(&work_mutex);//加锁
        printf("father set a apple.\n");
        sem_post(&apple);//释放一个apple信号，可吃苹果数加1
        pthread_mutex_unlock(&work_mutex);//解锁
    }
    return nullptr;
}
void* matherFunction(void* arg)
{
    while(1)
    {
        sem_wait(&empty);//占用一个盘子空间，可放水果数减1
        pthread_mutex_lock(&work_mutex);//加锁
        printf("mather set a orange.\n");
    }
}
```

```

    sem_post(&orange); //释放一个orange信号量，可吃水果加1
    pthread_mutex_unlock(&work_mutex); //解锁
}
return nullptr;
}

void* sonFunction(void)
{
while(1)
{
    sem_wait(&apple); //占用一个苹果信号量，可吃苹果数减1
    pthread_mutex_lock(&work_mutex); //加锁
    printf("son eat a apple.\n");
    sem_post(&empty); //吃了一个苹果，释放一个盘子空间，可放水果数加1
    pthread_mutex_unlock(&work_mutex); //解锁
}
return nullptr;
}

void* daughterFunction(void* arg)
{
while(1)
{
    sem_wait(&orange);
    pthread_mutex_lock(&work_mutex);
    printf("daughter eat a orange.\n");
    sem_post(&empty);
    pthread_mutex_unlock(&work_mutex);
}
return nullptr;
}

int main()
{
pthread_t father;
pthread_t mother;
pthread_t son;
pthread_t daughter;
sem_init(&empty, 0, 3);
sem_init(&apple, 0, 0);
sem_init(&orange, 0, 0);
pthread_mutex_init(&work_mutex, nullptr);
pthread_create(&father, nullptr, fatherFunction, nullptr);
pthread_create(&mother, nullptr, motherFunction, nullptr);
pthread_create(&son, nullptr, sonFunction, nullptr);
pthread_create(&daughter, nullptr, daughterFunction, nullptr);

pthread_join(father, NULL);
pthread_join(mother, NULL);
pthread_join(son, NULL);
pthread_join(daughter, NULL);
sem_destroy(&empty);
sem_destroy(&apple);
sem_destroy(&orange);
return 0;
}

```

6.临界区

7.互斥量

互斥量是一种线程同步对象，互斥的含义是同一时刻只能有一个线程获得互斥量，一个互斥量对应一个共享资源，互斥量状态：解锁状态意味着共享资源可用，加锁状态意味着共享资源不可用

一个线程需要使用共享资源时，使用pthread_mutex_lock申请，当互斥量为解锁状态，则占用互斥量，并给互斥量加锁，占用资源(互斥量为加锁状态，其他线程不能使用互斥量并等待互斥量变为解锁状态)；如果互斥量为加锁状态，则线程等待，直到互斥量为解锁状态(其他线程使用完共享资源后会解锁互斥量，释放资源)

- 互斥量基本函数

头文件：

```
#include <pthread.h>
```

初始化互斥量：

```
int pthread_mutex_init(pthread_mutex* mutex, const pthread_mutexattr_t* mutexattr);
```

第一个参数为互斥量指针，第二个参数为互斥量属性指针(一般设为nullptr)该函数将按照互斥量属性对互斥量进行初始化

互斥量加锁：

```
int pthread_mutex_lock(pthread_mutex* mutex);
```

申请一个互斥量并对其进行加锁，使该互斥量对其他线程不可用，让其他线程互斥量等待

互斥量解锁：

```
int pthread_mutex_destroy(pthread_mutex* mutex);
```

销毁一个不再需要的互斥量，释放系统资源

- 测试案例

```
#include <pthread.h>
#include <mutex>
#include <semaphore.h>
using namespace std;
pthread_t t1;
pthread_t t2;
char share[10];
pthread_mutex_t work_mutex;
void* oneFunction(void* arg)
{
    while(1)
    {
        char* p = share;
        pthread_mutex_lock(&work_mutex);
        for(int i = 0; i < 9; i++)
        {
            *p = 'a';
            p++;
        }
    }
}
```

```

    p++;
    *p = '\0';
    printf("线程 1 share是: %s\n",share);
    pthread_mutex_unlock(&work_mutex);
}
return nullptr;
}
void* twoFunction(void* arg)
{
    while(1)
    {
        char* p = share;
        pthread_mutex_lock(&work_mutex);
        for(int i = 0; i < 9;i++)
        {
            *p = 'e';
            p++;
        }
        p++;
        *p = '\0';
        printf("线程 2 share是: %s\n",share);
        pthread_mutex_unlock(&work_mutex);
    }
    return nullptr;
}
int main()
{
    pthread_mutex_init(&work_mutex, nullptr); //初始化互斥量
    pthread_create(&t1,nullptr,oneFunction,nullptr);
    pthread_create(&t2,nullptr,twoFunction,nullptr);
    pthread_mutex_destroy(&work_mutex); //销毁互斥量
    return 0;
}

```

2. 按序打印

1. 题目

提供一个类

```

class Foo
{
public:
    void one()
    {
        printf("one ");
    }
    void two()
    {
        printf("two ");
    }
    void three()
    {
        printf("three ");
    }
};

```

三个不同的线程将会共用一个Foo实例

- 线程A将会调用one()方法
- 线程B将会调用two()方法
- 线程C将会调用three()方法

示例

```
输入:[1,2,3]
输出:"one two three"
解释:
有三个线程会被异步启动
输入[1,2,3]表示线程A将会调用 one() 方法, 线程B将会调用two()方法, 线程C将会调用three()方法
正确的输出是"one two three"
```

2.题解

使用lock_guard：当mutex没上锁时，就会上锁。mutex上锁时，会等待，超出作用域后会析构解锁

```
#include <pthread.h>
#include <mutex>
using namespace std;
class Foo
{
public:
    Foo()
    {
        lock1.lock();
        lock2.lock();
    }
    void one()
    {
        printf("one ");
        lock1.unlock();
    }
    void two()
    {
        lock_guard<mutex> guard1(lock1);
        printf("two ");
        lock2.unlock();
    }
    void three()
    {
        lock_guard<mutex> guard2(lock2);
        lock_guard<mutex> guard1(lock1);
        printf("three ");
    }
private:
    mutex lock1;
    mutex lock2;
};
```

使用互斥信号量

```
#include <mutex>
#include <pthread.h>
class Foo1
```

```

{
public:
    Foo1()
    {
        pthread_mutex_init(&mutex1,nullptr);
        pthread_mutex_init(&mutex2,nullptr);
        pthread_mutex_lock(&mutex1);
        pthread_mutex_lock(&mutex2);
    }
    void one()
    {
        printf("one ");
        pthread_mutex_unlock(&mutex1);
    }
    void two()
    {
        pthread_mutex_lock(&mutex1);
        printf("two ");
        pthread_mutex_unlock(&mutex2);
    }
    void three()
    {
        pthread_mutex_lock(&mutex2);
        printf("three ");
        pthread_mutex_unlock(&mutex2);
    }
private:
    pthread_mutex_t mutex1;
    pthread_mutex_t mutex2;
};

```

3. 打印零与奇偶数

1. 题目

假设有这么一个类：

```

class zeroEvenOdd
{
public:
    ZeroEvenOdd(int n){...}//构造函数
    void zero(printNumber){...}//仅打印出0
    void even(printNumber){...}//仅打印偶数
    void odd(printNumber){...}//仅打印出奇数
};

```

相同的一个ZeroEvenOdd类实例将会传递给三个不同的线程：

- 线程A将调用zero(), 它只输出0
- 线程B将调用even(), 它只输出偶数
- 线程C将调用odd(), 它只输出奇数

每个线程都有一个printNumber方法来输出一个证书。请修改给出的代码以输出证书序列
010203040506..., 其中虚列的长度必须为 $2n$

示例：

输入:n = 2

输出:"0102"

说明:三条线程异步执行,其中一个调用zero(),另一个线程调用even(),最后一个线程调用odd()。正确输出为"0102"

2.题解

解法: 信号量

思路: 输出0的线程是主控,第一次输出0后,让输出奇数的线程工作,同时输出偶数的线程休眠。第二次输出0后,让输出偶数的线程工作,同时输出奇数的线程休眠,如此,每次输出0后,偶数和奇数线程交替工作。

用信号量进行线程间的互动,信号量wait和post操作原语的应用

```
#include <semaphore.h>
class zeroEvenOdd
{
public:
    ZeroEvenOdd(int n)
    {
        sem_init(&mzero,0,1);
        sem_init(&meven,0,0);
        sem_init(&modd,0,0);
    }
    void zero(function<void(int)> printNumber)
    {
        for(int i = 0; i < n ;i++)
        {
            sem_wait(&mzero);
            printNumber(0);
            if(i & 1)
            {
                sem_post(&modd);
            }else{
                sem_post(&meven);
            }
        }
    }
    void even(function<void(int)> printNumber)//偶数
    {
        for(int i = 2;i < n; i += 2)
        {
            sem_post(&meven);
            printNumber(i);
            sem_post(&mzero);
        }
    }
    void odd(function<void(int)> printNumber)//奇数线程
    {
        for(int i = 1;i <= n;i += 2)
        {
            sem_wait(&modd);
            printNumber(i);
            sem_post(&mzero);
        }
    }
}
```

```
private:  
    sem_t mzero;  
    sem_t meven;  
    sem_t modd;  
};
```

在这里稍微说一下，奇偶性的判断

对于一个整数数字，只需要判断二进制中最后一个是否为0与1，如果二进制位中最后一个数字为0，那么该数字为偶数(除去0整数)，如果最后一个数字为1那么为奇数

那么我将整数数字与1进行按位与运算，就能清晰的判断出，最后与计算后的是0或非0

4. H2O生成

1. 题目

现在有两种线程，氢与氧，目标是组织这两种线程来产生水分子

存在一个屏障使得每个线程必须等候知道一个完整的水分子能够被产生出来

氢和氧线程会被分别给予releaseHydrogen和releaseOxygen方法来允许它们突破屏障

这些线程应该三三成组突破屏障并能立即组合产生一个水分子

必须保证产生一个水分子锁需线程的结合必须发生在下一个水分子产生之前：

换句话说：

- 如果一个氧线程到达屏障时没有氢线程到达，它必须等候知道两个氢线程到达
- 如果一个氢线程到达屏障时没有其他线程到达，它必须等候知道一个氧线程和另一个氢线程到达

书写满足这些限制条件的氢、氧线程同步代码

示例1：

```
输入："HOH"  
输出："HHO"  
解释："HOH"和"OHH"依然都是有效解
```

示例：

```
输入："OOHHHH"  
输出："HHOHHO"  
解释："HOHHHO"， "OHHHHO"， "HHOHOH"， "HOHHOH"， "OHHHOH"， "HHOOHH"， "HOHOHH" 和  
"OHHOHH" 依然都是有效解
```

限制条件：

- 输入字符串的总长将会是 $3n$ ，取值范围为1到50
- 输入字符串中的"H"，总数将会是 $2n$
- 输入字符串中的>O"，总数将会是 n

2. 题解

```
#include <semaphore.h>  
class H2O  
{  
public:  
    H2O()
```

```

{
    sem_init(&h_limit, 0, 2);
    sem_init(&o_limit, 0, 1);
}
void hydrogen(function<void()> releaseHydrogen)
{
    sem_wait(&h_limit);
    releaseHydrogen();
    count_h++;
    if(count_h == 2)
    {
        count_h = 0;
        sem_post(&o_limit);
    }
}
void oxygen(function<void()> releaseOxygen)
{
    sem_wait(&o_limit);
    releaseOxygen();
    sem_post(&h_limit);
    sem_post(&h_limit);
}
private:
    int count_h = 0;
    sem_t h_limit;
    sem_t o_limit;
};

```

5. 交替打印字符串

1. 题目

编写一个可以从 1 到 n 输出代表这个数字的字符串的程序，但是：

- 如果这个数字可以被 3 整除，输出 "fizz"
- 如果这个数字可以被 5 整除，输出 "buzz"
- 如果这个数字可以同时被 3 和 5 整除，输出 "fizzbuzz"

例如：

```
当 n = 15, 输出: 1, 2, fizz, 4, buzz, fizz, 7, 8, fizz, buzz, 11, fizz, 13, 14,
fizzbuzz
```

假设有这么一个类：

```

class FizzBuzz
{
public:
    FizzBuzz(int n){...}
    void fizz(printFizz){...}
    void buzz(printBuzz){...}
    void fizzBizz(printFizzBuzz){...}
    void number(printNumber){...}
};

```

实现一个有四个线程的多线程版FizzBuzz，同一个FizzBuzz实例会被如下四个线程使用：

- 线程A将调用 `fizz()` 来判断是否能被 3 整除，如果可以，则输出 `fizz`
- 线程B将调用 `buzz()` 来判断是否能被 5 整除，如果可以，则输出 `buzz`
- 线程C将调用 `fizzbuzz()` 来判断是否同时能被 3 和 5 整除，如果可以，则输出 `fizzbuzz`
- 线程D将调用 `number()` 来实现输出既不能被 3 整除也不能被 5 整除的数字

2. 题解

```
#include <semaphore.h>
#include <functional>
#include <thread>
using namespace std;
class FizzBuzz
{
public:
    FizzBuzz(int n)
    {
        this->n = n;
        cur = 0;
        sem_init(&sem_fizz, 0, 0);
        sem_init(&sem_buzz, 0, 0);
        sem_init(&sem_fizz_buzz, 0, 0);
        sem_init(&sem_num, 0, 1);
    }
    void fizz(function<void()> printFizz)
    {
        while(cur <= n)
        {
            sem_wait(&sem_fizz);
            if(curr > n)break;
            printFizz();
            sem_post(&sem_num);
        }
    }
    void buzz(function<void()> printBuzz)
    {
        while(cur <= n)
        {
            sem_wait(&sem_buzz);
            if(curr > n)break;
            printBuzz();
            sem_post(&sem_num);
        }
    }
    void fizzBuzz(function<void()> printFizzBuzz)
    {
        while(cur <= n)
        {
            sem_wait(&sem_fizz_buzz);
            if(curr > n)break;
            printFizz();
            sem_post(&sem_num);
        }
    }
    void number(function<void(int)> printNumber)
    {
        while(++cur <= n)
        {
```

```

    sem_wait(&sem_num);
    if(cur % 3 == 0 && cur % 5 == 0){
        sem_post(&sem_fizz_buzz);
    }else if(cur % 3 == 0){
        sem_post(&sem_fizz);
    }else if(cur % 5 == 0){
        sem_post(&sem_buzz);
    }else{
        printNumber(cur);
        sem_post(&sem_num);
    }
}

sem_post(&sem_fizz);
sem_post(&sem_buzz);
sem_post(&sem_fizz_buzz);
}

private:
int n;
int cur;
sem_t sem_fizz;
sem_t sem_buzz;
sem_t sem_fizz_buzz;
sem_t sem_num;
};

int main(int argc, char** argv){
    FizzBuzz fizzBuzz(15);
    std::function<void()> printFizz = [](){printf(" fizz ");};
    std::function<void()> printBuzz = [](){printf(" buzz ");};
    std::function<void()> printFizzBuzz = [](){printf(" fizzbuzz ");};
    std::function<void(int)> printNum = [](int x){printf(" %d ", x);};
    std::thread th[4];
    th[0] = std::thread(&FizzBuzz::fizz, &fizzBuzz, printFizz);
    th[1] = std::thread(&FizzBuzz::buzz, &fizzBuzz, printBuzz);
    th[2] = std::thread(&FizzBuzz::fizzbuzz, &fizzBuzz, printFizzBuzz);
    th[3] = std::thread(&FizzBuzz::number, &fizzBuzz, printNum);
    for(auto &ts : th) {
        if(ts.joinable()) ts.join();
    }
    return 0;
}

```

6. 哲学家进餐

五个哲学家共用一张圆桌，在桌子上有五只碗和五只筷子，它们的生活的方式是交替的进行思考和进餐。平时，一个哲学家进行思考，饥饿时便试图取用其左右最靠近他的筷子，只有他拿到筷子时才能进餐。进餐完毕，放下筷子继续思考

分析：

放在桌子上的筷子时临界资源，在一段时间内只允许一位哲学家使用，为了实现对筷子的互斥访问，可以用一个信号量表示筷子，由这五个信号量构成信号量数组

```

semaphore chopstick[5] = {1,1,1,1,1};
while(1)
{
    //哲学家饥饿时，总是先拿起左边的筷子，再拿右边的筷子
    wait(chopstick[i]);
    wait(chopstick[(i + 1)%5]);
    //吃饭
    //当哲学家进餐完成后，总是先放下左边的筷子，在放下右边的筷子
    signal(chopstick[i]);
    signal(chopstick[(i + 1)% 5]);
}

```

上述代码可以保证不会有两个相邻的哲学家同时进餐，但却可能会引起死锁的情况，假如五位哲学家同时饥饿而都拿起的左边的筷子，就会使五个信号量chopstick都为0，当他们试图去拿右手边的筷子时，都将无筷子而陷入无限期的等待

解决方案1：

至多只允许四个哲学家同时进餐，以保证至少有一个哲学家能够进餐，最终总会释放出他所使用过的两支筷子，从而可使更多的哲学家进餐。定义信号量count，只允许4个哲学家同时进餐，这样就能保证至少有一个哲学家可以就餐

```

semaphore chopstick[5]={1,1,1,1,1};
semaphore count=4; // 设置一个count，最多有四个哲学家可以进来
void philosopher(int i)
{
    while(true)
    {
        think();
        wait(count); //请求进入房间进餐 当count为0时 不能允许哲学家再进来了
        wait(chopstick[i]); //请求左手边的筷子
        wait(chopstick[(i+1)%5]); //请求右手边的筷子
        eat();
        signal(chopstick[i]); //释放左手边的筷子
        signal(chopstick[(i+1)%5]); //释放右手边的筷子
        signal(count); //离开饭桌释放信号量
    }
}

```

解决方案2：

仅当哲学家的左右两支筷子都可用时，才允许他拿起筷子进餐。可以利用AND型信号量机制实现，也可以利用信号量的保护机制实现。利用信号量的保护机制实现的思想是通过记录型信号量mutex对取左侧和右侧筷子的操作进行保护，使之成为一个原子操作，这样可以防止死锁的出现。描述如下

1.用记录型信号量实现：

```

semaphore mutex = 1; // 这个过程需要判断两根筷子是否可用，并保护起来
semaphore chopstick[5]={1,1,1,1,1};
void philosopher(int i)
{
    while(true)
    {
        /* 这个过程中可能只能由一个人在吃饭，效率低下，有五只筷子，其实是可以达到两个人同时吃饭 */
        think();

```

```

    wait(mutex); // 保护信号量
    wait(chopstick[(i+1)%5]); // 请求右手边的筷子
    wait(chopstick[i]); // 请求左手边的筷子
    signal(mutex); // 释放保护信号量
    eat();
    signal(chopstick[(i+1)%5]); // 释放右手边的筷子
    signal(chopstick[i]); // 释放左手边的筷子
}
}

```

2.用AND型信号量实现：

```

semaphore chopstick[5]={1,1,1,1,1};
do{
    //think()
    Swait(chopstick[(i+1)%5],chopstick[i]);
    //eat()
    Ssignal(chopstick[(i+1)%5],chopstick[i]);
}while(true)

```

解决方案3：

规定奇数号的哲学家先拿起他左边的筷子，然后再去拿他右边的筷子；而偶数号的哲学家则先拿起他右边的筷子，然后再去拿他左边的筷子。按此规定，将是1、2号哲学家竞争1号筷子，3、4号哲学家竞争3号筷子。即五个哲学家都竞争奇数号筷子，获得后，再去竞争偶数号筷子，最后总会有一个哲学家能获得两支筷子而进餐

```

semaphore chopstick[5]={1,1,1,1,1};
void philosopher(int i)
{
    while(true)
    {
        think();
        if(i%2 == 0) //偶数哲学家，先右后左。
        {
            wait (chopstick[(i + 1)%5]) ;
            wait (chopstick[i]) ;
            eat();
            signal (chopstick[(i + 1)%5]) ;
            signal (chopstick[i]) ;
        }
        else //奇数哲学家，先左后右。
        {
            wait (chopstick[i]) ;
            wait (chopstick[(i + 1)%5]) ;
            eat();
            signal (chopstick[i]) ;
            signal (chopstick[(i + 1)%5]) ;
        }
    }
}

```

2.5 C++中基础知识

1. new与malloc的区别

1.1 申请内存所在位置

- new操作符从自由存储区上为对象分配内存空间

自由存储区是c++基于new操作符的一个抽象概念，通过new操作符进行内存申请，该内存即为自由存储区

自由存储区是否能在堆上动态分配内存，取决于operator new的实现细节，自由存储区不仅可以是堆，还可以是静态存储区，这都看operator new在那里为对象分配内存

- malloc函数从堆上动态分配内存

堆事操作系统中的术语，是操作系统所维护的一块特殊内存，用于程序的内存动态，c语言使用malloc从堆上分配内存，使用free释放对应内存

1.2 返回类型安全性

- new操作符内存分配成功时，返回的是对象类型的指针，类型严格与对象匹配，无须进行类型转换，因此new是符合类型安全性的操作符
- malloc内存分配成功则是返回void*，需要通过强制类型转换将void *指针转换成我们需要的类型，类型安全很大程度上可以等价于内存安全，类型安全的带哪不会试图访问自己没被授权的内存区域

1.3 内存分配失败时的返回值

- new内存分配失败时，会抛出bad_alloc异常，它不会返回null

```
try
{
    int* a = new int();
}catch(bad_alloc){
    ...
}
```

我们应该检测new操作符时的状态，如果new失败就会抛出异常

- malloc分配内存失败时返回null

```
int *a = (int *)malloc( sizeof( int ) );
if(NULL == a)
{
    ...
}
else
{
    ...
}
```

1.4 是否需要指定内存大小

- new操作符

使用new操作符申请内存分配时无须指定内存块的大小，编译器会根据类型信息自行计算

- malloc操作符

需要显示的支出所需内存的尺寸

1.5 是否调用构造、析构函数

- new操作符 分配对象步骤

1. 调用operator new函数（对于数组时operator new[]）分配一块足够大的，原始的未命名的内存空间以便存储特定类型的对象
 2. 编译器运行相应的构造函数以构造对象，并为其传入初值
 3. 对象构造完成后，返回一个指向该对象的指针
 4. delete操作符，调用析构函数，编译器调用operator delete(或operator delete[])函数释放内存空间
- malloc操作符

不会调用c++自定义类型与标准库中凡是需要构造/析构的类型的构造函数与析构函数

2. vector与list的区别

2.1 vector相关

- vector和数组类似，拥有一段连续的内存空间，并且起始地址不变能够高效的进行随机访问，时间复杂度是O(1)
- 因为内存空间是连续的，所以在进行插入和删除操作时，会造成内存块的拷贝，时间复杂度是O(n)
- 当内存空间不够时，会重新申请一块内粗暴空间并进行内存拷贝

2.2 list相关

- list是由双向链表实现的，因此内存空间是不连续的。
- 只能通过指针访问数据，所以list的随机存取非常没有效率，时间复杂度为o(n);
- 但由于链表的特点，能高效地进行插入和删除。

2.3 map与unordered_map的区别/插入和删除的复杂度

- map

map内部实现了一个红黑树，该结构具有自动排序的功能，因此map内部所有的元素都是有序的，红黑树的每一个节点都代表着map的一个元素，因此map的查找、删除、添加操作都是相当于对于红黑树进行这样的操作，故红黑树决定了map的效率。

有序性，这是map结构最大的优点，其元素的有序性在很多应用中都会简化很多的操作。红黑树，内部实现一个红黑书使得map的很多操作在lg n lg n的时间复杂度下就可以实现，因此效率非常的高。

- 红黑树

- Unordered_map内部实现哈希表，其内部元素排列时杂乱的、无序的，查找速度非常的快，但是哈希表的建立比较耗费时间

- 哈希表

3. C++修饰符

1. const

- 作用

1. 修饰变量，说明该变量不可以被改变
2. 修饰指针，分为指向常量的指针和指针常量。常量指针：指向常量;指针常量：指针是常量
3. 常量引用，经常用于形参类型，即避免了拷贝，又避免了函数对值的修改
4. 修饰成员函数，说明该成员函数内不能修改成员变量

- 具体例子

```
#include <iostream>
class TestConst
{
public:
```

```

TestConst():a_(0){}//构造函数
TestConst(int x):a_(x){}//初始化列表

virtual ~TestConst(){}
//const 可用于对重载函数的区分
int getValue()//普通成员函数
int getValue() const//常成员函数，不得修改类中的任何数据成员的值
private:
const int a_;//常对象成员，只能在初始化列表赋值
};

void function()
{
//对象
TestConst a_;//普通对象，可以调用全部成员函数、更新常成员变量
const TestConst b_;//常对象，只能调用常成员函数
const TestConst* p_ = &a_;//常指针
const TestConst &q_ = a;//常引用
//指针
char greeting[] = "HelloLiuQian";
char* p1 = greeting; //指针变量，指向字符数组变量
const char* p2 = greeting; //指针变量，指向字符串数组常量
char* const p3 = greeting; //常指针，指向字符串数组变量
const char* const p4 = greeting;//常指针，指向字符数组常量
}
//函数
void function1(const int var);//传递过来的参数在函数内不可变
void function2(const char* var);//传递过来的变量指针，指向内容为常量
void function3(char* const var);//参数指针为常指针
void function4(const int& var);//引用参数在函数内为常量
//函数返回值
const int function5();//返回一个常数
const int* function6();//返回一个指向常量的指针变量，const int* p = function6();
int* const function7();//返回一个指向变量的常指针，使用 int* const p = function7();

```

2. static

- 作用

- 修饰普通变量，修改变量的存储区域和生命周期，使变量存储在静态区，在 main 函数运行前就分配了空间，如果有初始值就用初始值初始化它，如果没有初始值系统用默认值初始化它。
- 修饰普通函数，表明函数的作用范围，仅在定义该函数的文件内才能使用。在多人开发项目时，为了防止与他人命名空间里的函数重名，可以将函数定位为 static。
- 修饰成员变量，修饰成员变量使所有的对象只保存一个该变量，而且不需要生成对象就可以访问该成员。
- 修饰成员函数，修饰成员函数使得不需要生成对象就可以访问该函数，但是在 static 函数内不能访问非静态成员。

3. volatile

```
volatile int i = 10;
```

- volatile关键字是一种类型修饰符，用它声明的类型变量表示可以被某些编译器未知因素(操作系统、硬件、其它线程等)更改。所以使用volatile告诉编译器不应该对这样的对象进行优化

- volatile关键字声明的变量，每次访问时都必须从内存中取出值(没有被volatile修饰的变量，可能由于编译器的优化，从CPU寄存器中取值)
- const可以是volatile(如只读的状态寄存器)
- 指针可以是volatile

4. explicit(显示)关键字

- explicit修饰构造函数时，可以防止隐式转换和肤质初始化
- explicit修饰转换函数时，可以防止隐式转换，但按语境转换除外

具体例子

```

struct A
{
    A(int a){}
    operator bool() const {return true;}
};

struct B
{
    explicit B(int b){}
    explicit operator bool()const{return true}
};
void doA(A a){}
void doB(B b){}
int main()
{
    A a1(1);//OK: 直接初始化
    A a2 = 1;//OK: 复制初始化
    A a3{ 1 };//OK: 直接列表初始化
    A a4 = { 1 };//OK: 复制列表初始化
    A a5 = (A)1;//OK: 允许 static_cast 的显式转换
    doA(1);//OK: 允许从 int 到 A 的隐式转换
    if (a1);//OK: 使用转换函数 A::operator bool() 的从 A 到 bool 的隐式转换
    bool a6 (a1);//OK: 使用转换函数 A::operator bool() 的从 A 到 bool 的隐式转换
    bool a7 = a1;//OK: 使用转换函数 A::operator bool() 的从 A 到 bool 的隐式转换
    bool a8 = static_cast<bool>(a1);//OK : static_cast 进行直接初始化

    B b1(1);//OK: 直接初始化
    B b2 = 1;//错误: 被 explicit 修饰构造函数的对象不可以复制初始化
    B b3{ 1 };//OK: 直接列表初始化
    B b4 = { 1 };//错误: 被 explicit 修饰构造函数的对象不可以复制列表初始化
    B b5 = (B)1;//OK: 允许 static_cast 的显式转换
    doB(1);//错误: 被 explicit 修饰构造函数的对象不可以从 int 到 B 的隐式转换
    if (b1);//OK: 被 explicit 修饰转换函数 B::operator bool() 的对象可以从 B 到 bool 的按语境转换
    bool b6(b1);//OK: 被 explicit 修饰转换函数 B::operator bool() 的对象可以从 B 到 bool 的按语境转换
    bool b7 = b1;//错误: 被 explicit 修饰转换函数 B::operator bool() 的对象不可以从 B 到 bool 的按语境转换
    bool b8 = static_cast<bool>(b1); // OK: static_cast 进行直接初始化
    return 0;
}

```

5. extern

一、告知编译器：当extern与“c”一起使用的时候，就是告诉编译器，下面的函数或者变量以C语言的方式编译。这里主要是因为一方面我们可以使用C语言写成的项目运用到C++中，另一方面由于C++支持重载而C不支持，这就导致了C++在编译的时候，C++的函数名会和参数一起被编成函数名，而C只是函数名。所以在链接的时候，找不到我们定义的那个函数。

二、就是共享：extern可以置于变量或者函数前，以标示变量或者函数的定义在别的文件中，提示编译器遇到此变量和函数时在其他模块中寻找其定义。此外extern也可用来进行链接指定。

4. 智能指针

1. Shared_ptr

- 概念

智能指针的原理是，接受一个申请好的内存地址，构造一个保存在栈上的智能指针对象，当程序退出栈的作用域范围后，由于栈上的变量自动被销毁，智能指针内部保存的内存也就被释放掉了（除非将智能指针保存起来）。

shared_ptr使用引用计数，每一个shared_ptr的拷贝都指向相同的内存。每使用他一次，内部的引用计数加1，每析构一次，内部的引用计数减1，减为0时，删除所指向的堆内存。shared_ptr内部的引用计数是安全的，但是对象的读取需要加锁。

2.unique_ptr

3. Weak_ptr

5. 深拷贝与浅拷贝

1. 普通类型对象的拷贝

2. 类对象的拷贝

3. 浅拷贝

4. 深拷贝

6. 宏定义与inline函数

1. 宏定义

1. 概述

预处理命令可以改变程序设计环境，提高编程效率，它们并不是c语言本身的组成部分，不能直接对它们进行编译，必须在对程序进行编译之前，先对程序中这些特殊的命令进行“预处理”。经过预处理后，程序就不在包括预处理命令了，最后在由编译程序对预处理之后的源程序进行编译处理，得到可供执行的目标代码。c语言提供的预处理功能有三种，分别为爱宏定义，文件包含和条件编译

2. 宏定义的基础语法

宏定义在c语言源程序中允许用一个标识符来表示一个字符串，称之为宏或宏体，被定义为宏的标识符称之为宏名。在预编译处理时，对程序中所有出现的宏名，都用宏定义中的字符串去代替，这称之为宏替换或宏展开

宏定义是由源程序中的宏定义命令完成的，宏替换是由预处理程序自动完成的

c语言中，宏分为有参和无参两种。无参宏的宏名后不带参数，其定义的一般形式为：

```
#define 标识符 字符串
```

“#”表示这是一条预处理命令

“define”：宏定义命令

“**标识符**”：宏定义的宏名

“**字符串**”：可以是常熟、表达式、格式串等，符号常量

```
//不带参数的宏定义  
#define MAX 10  
//带参数的宏定义  
#define M(y) y * y + 3 * y  
//宏调用  
k = M(5);
```

3. 宏定义的优点

- 方便程序的修改
- 提高程序的运行效率

函数调用会占用运行时间(分配单元，保存现场，值传递，返回),每次执行都要载入，而宏只占编译时间，宏替换会替换掉程序中所用宏的地方，这样，不需要占用运行时间

4. 宏定义的缺点

- 会增加编译后源码所占的内存
- 宏定义是不可以debug的，而且是预编译处理，所以不会检查宏定义的正确性
- 宏定义过多的嵌套，影响可读性、维护性
- 宏的定义很容易产生二义性，如：定义#define S(a) (a)*(a)，那么调用S(a++)，则宏展开为(a++) * (a++),并且在不同的编译器下会有不同编译结果
 - 算符优先级问题

```
#define Multiply(x,y) x * y  
//调用示例  
Multiply(1,2); //宏展开后: 1 * 2  
Multiply(1+2,2); //宏展开后: 1 + 2 * 2  
//总结: 优先级明显出错了  
//解决方法: 对宏体和被引用的每个参数加括号, 就能避免这个问题  
#define Multiply(x,y) ((x) * (y))
```

▪ 分号吞噬问题

```
#define foo(x) bar(x);baz(x)  
//调用示例  
if(!feral)  
    foo(wolf);  
//宏展开后为  
if(!feral)  
    bar(wolf);  
baz(wolf);  
//解决方法: 通过使用do{}while(0)能够解决上述问题  
#define foo(x) do{bar(x);baz(x);}while(0)  
if(!feral)  
    foo(wolf);  
else  
    bin(wolf);  
//宏展开后为  
#define foo(x) do{ bar(x);baz(x); }while(0)  
if(!feral)  
    do{bar(x);baz(x);}while(0);
```

```
else  
bin(wolf);
```

■ 宏参数重复调用

```
#define min(x,y) ((x)<(y)?(x):(y))  
//调用示例  
min(x+y, foo(z));  
//宏展开后  
((x + y)<foo(z))?(x + y):(foo(z))  
//foo(z)有可能会被重复调用了两次，如若果foo是不可重入的(foo内修改了全局或静态  
变量)，程序  
//会产生逻辑错误
```

■ 对自身的递归调用

```
#define foo(4 + foo)  
//按照前面的理解，该宏会一直展开下去，但是预处理器采取的策略是只展开一次，但是不  
推荐，  
//可读性太差
```

5. 宏函数中的特定语法

1. 在宏体中，如果宏参数前加一个#，那么在宏体扩展的时候，宏参数会被扩展成字符串的形式

```
#include <stdio.h>  
#define Psqr(x) printf("The Square of "#x" is %d.\n",((x)*(x)))  
#define Psqr2(x) printf("The Square of %s is %d.\n",#x,((x)*(x)))  
//调用示例  
int test = 5;  
Psqr(test);//The Square of test is 25  
Psqr(test);//The Square of test is 25
```

2. 和##运算符一样，“##”运算符可以用于类函数宏的替换部分。粘合两个语言符号为单个语言符号

3. 可变宏：...和__VA_ARGS__

```
#define PR(...)printf(__VA_ARGS__)
```

2. inline函数

内联函数和宏的区别在于，宏是由预处理器对宏进行替代。而内联函数是通过编译器控制实现的，内联函数是真正的函数，只有在需要用到的时候，内联函数像宏一样展开，所以取消了函数的参数压栈，减少了调用的开销。可以像调用函数一样来调用内联函数，而不必担心会产生类似宏的一些问题。

内联函数有一定的局限性，就是函数中执行代码不能太多了，如果内联函数体过大，一般的编译器会放弃内联方式，而采用普通的方式调用函数。这个时候，内联函数就和普通函数执行效率一样了。

2.6 C++ 设计模式

设计模式原则

1. 单一职责原则(SRP, Single Responsibility Principle)
2. 里氏替换原则(LSP, Liskov Substitution Principle)

3. 依赖倒置原则(DIP, Dependence Inversion Principle)

4. 接口隔离原则(ISP, Interface Segregation Principle)

5. 迪米特原则(LoD, Law of Demeter)

6. 开放封闭原则(OCP, Open Close Principle)

1. 适配器模式

1. 概念

适配器模式把一个类的接口转换成客户端所期待的另一种接口，从而使原本接口不匹配而无法在一起工作的两个类能够在一起工作

适配器模式有类适配器和对象适配器两种模式

2. 类适配器

- 目标角色(Target):客户所期待的接口
- 源角色(Adapter):需要适配的类
- 适配器角色(Adapter):把源接口转换成目标接口

实现例子如下：

```
#include <iostream>
using namespace std;
class Target
{
public:
    virtual void request(){};;
};

class Adaptee
{
public:
    void specificRequest()
    {
        printf("called specificRequest()");
    }
};

class Adapter:public Adaptee,public Target
{
public:
    void request()
    {
        this->specificRequest();
    }
};

int main()
{
    Target* t = new Adapter();
    t->request();
    delete t;
    return 0;
}
```

3. 对象适配器

- 目标角色(Target):客户所期待的接口。目标可以是具体的或抽象的类，也可以是接口。

- 源角色(Adapter):需要适配的类
- 适配器角色(Adapter):通过在内部包装 (Wrap) 一个Adaptee对象，把源接口转换成目标接口

实现例子如下:

```
#include <iostream>
using namespace std;
class Target
{
public:
    virtual void request() {};
};

class Adaptee
{
public:
    void specificRequest()
    {
        printf("called specificRequest()");
    }
};

class Adapter:public Target
{
public:
    Adapter()
    {
        try
        {
            adaptee_ = new Adaptee();
        }catch(bad_alloc)
        {
            printf("create Adaptee point object failure");
        }
    }

    virtual ~Adapter()
    {
        if(adaptee_ != nullptr)
            delete adaptee_;
    }

    void request()
    {
        adaptee_->specificRequest();
    }
private:
    Adaptee* adaptee_;
};

int main()
{
    Target* t = new Adapter();
    t->request();
    delete t;
    return 0;
}
```

4. 注意要点

- Adapter模式主要是复用现存的类，但是接口又与复用环境要求不一致的情况，遗留代码复用、类库迁移方面非常有用

- Adapter模式，分为对象适配器与类适配器，但是类适配器采用多继承，一不小心就会产生菱形继承的问题，建议采用对象适配器的对象组合这种手段
- Adapter模式要求尽量面向接口编程，后期很方便适配

5. 使用场景

- 使用现有的类，但是现有类接口不符合系统需要
- 建立一个重复使用的类，
- 需要改变多个已有子类的接口

2. 桥接模式

1. 概念

桥接模式，将抽象部分与它的实现部分分离，使它们都可以独立地变化。这里实现指的是抽象类和它的派生类用来实现自己的对象，也就是说实现系统可能有多角度分类，每一个分类都有可能变化，那么就把这种多角度分离出来让它们独立变化，减少它们之间的耦合。

2. 具体的例子

```
#include <iostream>
using namespace std;

class Implementor
{
public:
    virtual void Operation() = 0;
    virtual ~Implementor(){}
};

class ConcreteImplementorA : public Implementor
{
public:
    void Operation() {
        printf("ConcreteImplementorA");
    }
};

class ConcreteImplementorB : public Implementor
{
public:
    void Operation()
    {
        printf("ConcreteImplementorB");
    }
};

class Abstraction
{
protected:
    Implementor* implementor;
public:
    void setImplementor(Implementor* im)
    {
        implementor = im;
    }
    virtual void Operation()
    {
```

```

        implementor->Operation();
    }
    virtual ~Abstraction(){}
};

class RefinedAbstraction : public Abstraction
{
public:
    void operation()
    {
        implementor->Operation();
    }
};

int main()
{
    Abstraction* r = new RefinedAbstraction();
    ConcreteImplementorA* ca = new ConcreteImplementorA();
    ConcreteImplementorB* cb = new ConcreteImplementorB();
    r->setImplementor(ca);
    r->Operation();
    r->setImplementor(cb);
    r->Operation();

    delete ca;
    delete cb;
    delete r;
    return 0;
}

```

3. 观察者模式

1. 概念

定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。它还有两个别名，依赖(Dependents)，发布-订阅(Publish-Subscribe)。

2. 具体的例子

```

//观察者
class Observer
{
public:
    Observer() {}
    virtual ~Observer() {}
    virtual void Update() {}

};

//博客
class Blog
{
public:
    Blog() {}
    virtual ~Blog() {}
    void Attach(Observer *observer) { m_observers.push_back(observer); } //添加观察者
    void Remove(Observer *observer) { m_observers.remove(observer); } //移除观察者
    void Notify() //通知观察者
    {
        list<Observer*>::iterator iter = m_observers.begin();
        for(; iter != m_observers.end(); iter++)
            (*iter)->Update();
    }
};

```

```

    }
    virtual void SetStatus(string s) { m_status = s; } //设置状态
    virtual string GetStatus() { return m_status; } //获得状态
private:
    list<Observer*> m_observers; //观察者链表
protected:
    string m_status; //状态
};

//具体博客类
class BlogCSDN : public Blog
{
private:
    string m_name; //博主名称
public:
    BlogCSDN(string name): m_name(name) {}
    ~BlogCSDN() {}
    void SetStatus(string s) { m_status = "CSDN通知：" + m_name + s; } //具体设置
    //状态信息
    string GetStatus() { return m_status; }
};

//具体观察者
class ObserverBlog : public Observer
{
private:
    string m_name; //观察者名称
    Blog *m_blog; //观察的博客，当然以链表形式更好，就可以观察多个博客
public:
    ObserverBlog(string name,Blog *blog): m_name(name), m_blog(blog) {}
    ~ObserverBlog() {}
    void Update() //获得更新状态
    {
        string status = m_blog->GetStatus();
        cout << m_name << "-----" << status << endl;
    }
};

int main()
{
    Blog *blog = new BlogCSDN("SmileSFL");
    Observer *observer1 = new ObserverBlog("aaa", blog);
    blog->Attach(observer1);
    blog->SetStatus("设计模式C++实现--观察者模式");
    blog->Notify();
    delete blog; delete observer1;
    return 0;
}

```

4. 单例模式

1. 概念

单例模式(Singleton Pattern, 也称为单件模式)，使用最广泛的设计模式之一。其意图是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，该实例被所有程序模块共享。

2. 具体的例子

```
// 懒汉式单例模式
```

```

class Singleton
{
private:
    Singleton() { }
    static Singleton * pInstance;
public:
    static Singleton * GetInstance()
    {
        if (pInstance == nullptr)
            pInstance = new Singleton();
        return pInstance;
    }
};

// 线程安全的单例模式
class Singleton
{
private:
    Singleton() { }
    ~Singleton() { }
    Singleton(const Singleton &);
    Singleton & operator = (const Singleton &);
public:
    static Singleton & GetInstance()
    {
        static Singleton instance;
        return instance;
    }
};

```

3. 注意的事情

单例模式，如果运用不得当极其容易造成内存泄露问题，所以一定要小心谨慎。所以在开发中，尽量使用设计模式与上下文思想来规避这个单例的模式

2.7 内存管理

1.

3. 渲染架构

1. ECS设计模式

4. 渲染算法

4.1 OpenGL ES 2.0与3.0的区别

4.1.1 概述

OpenGL ES 3.0在2012年8月发布向前兼容OpenGL ES 2.0，同时也与OpenGL 4.3兼容，OpenGL ES 3.0版本在发布时的实际版本是3.0.5。

4.1.2 OpenGL ES 3.0新功能

1. Occlusion Queries: 遮挡查询

遮挡查询通俗来讲，就是OpenGL提供了API来查询某些物体的包围盒子是否被遮挡了，然后返回遮挡查询的结果，我们拿到这些查询的结果后，就可以知道，那些物体需要渲染，那些物体不需要渲染，可以提升性能。

那么我们就需要给渲染的物体提供一个包围盒子提交给OpenGL的API进行判断，然后我们拿到遮挡查询的结果，来判断是否渲染真正的物体。

使用：

```
void initial()
{
    //生成查询对象ID, n查询对象的数量 ids存储查询对象的数组
    glGenQueries(GLsizei n,GLuint* ids);
}

void release()
{
    //n:要删除的查询对象的数量 ids:要删除的查询对象的数组
    //删除id, 回收资源
    glDeleteQueries(GLsizei n,GLuint* ids);
}

void draw()
{
    //开启遮挡查询
    //target取值
    //GL_ANY_SAMPLES_PASSED
    //GL_ANY_SAMPLES_PASSED_CONSERVATIVE(性能较高, 精度低)
    //GL_TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN
    //id知名查询对象的名字
    glBeginQuery(GLenum target,GLuint id);
    drawBoundingBox();
    //结束遮挡查询
    //target与GLBeginQuery中的target一样
    glEndQuery(GLenum target);
    //绘制真实物体阶段
    //查询遮挡查询的结果
    //pname可取值为: GL_QUERY_RESULT, GL_QUERY_RESULT_AVAILABLE
    //params:如果pname的值为GL_QUERY_RESULT, params为通过了深度测试的片段的数量, 如果值为0, 则表示这个物体被完全遮挡
    //如果噢pname的为GL_QUERY_RESULT_AVAILABLE, 则该方法用来检测遮挡查询是否完成
    //param的值为GL_TRUE或者GL_FALSE
    glGetQueryObjectuiy(GLuint id,GLenum pname,GLuint* params);
    //根据查询结果, 判断是否渲染真正物体
    drawRealObject();
}

void drawBoundingBox(){...}
void drawRealObject(){...}
```

2. Transform Feedback:变换反馈

- 概述

变换反馈是在OpenGL ES 3.x渲染管线中，顶点处理阶段结束之后，图元装配和光栅化之前的一个步骤。变换反馈可以重新捕获即将装配为图元(点、线段、三角形)的顶点，然后你将它们的部分或者全部属性传递到缓存对象。其可以将顶点着色器的处理结果输出，并且可以多个输出。

- 如何使用

- 设置变换反馈变量

```

void initialShader()
{
    glAttachShader(program, vertexShaderHandle);
    glAttachShader(program, fragShaderHandle);
    //我们获取变换反馈的变量需要在glLinkProgram之前，因为我们并不需要
    //顶点着色器之后的处理结果
    GLChar const* varyings[] = {"outPos", "outTex"};
    //GL_INTERLEAVED_ATTRIBS:将顶点输出变量捕获到一个缓冲区中
    //GL_SEPARATE_ATTRIBS:将每一个顶点输出变量捕获到它自己的缓冲区周边
    glTransformFeedbackVaryings(program, sizeof(varyings) /
        sizeof(varyings[0]), varyings, GL_INTERLEAVED_ATTRIBS);
    glLinkProgram(program);
}

```

- 创建变换反馈缓冲区

```

void createTBufferCach()
{
    //创建变换反馈的缓冲区buffer
    glGenBuffers(1, &transFeedBufId);
    //绑定变换反馈缓冲区buffer
    glBindBuffer(GL_TRANSFORM_FEEDBACK_BUFFER, transFeedBufId);
    //该步骤是设置缓存的大小，其设置的大小是一个三维向量与一个二维向量，一共6个项
    //点，大小自然是(3+2)*6*sizeof(GLfloat)
    glBindBufferData(GL_TRANSFORM_FEEDBACK_BUFFER, (3 + 2) *
        6 * sizeof(GLfloat), NULL, GL_STATIC_READ);
    //释放绑定
    glBindBuffer(GL_TRANSFORM_FEEDBACK_BUFFER, 0);
}

```

- 创建变换反馈对象，并绑定缓冲区

```

void createTObject()
{
    //创建变换反馈缓冲区对象
    glGenTransformFeedbacks(1, &transFeedbackObjId);
    //绑定变换反馈缓冲区对象
    glBindTransformFeedback(GL_TRANSFORM_FEEDBACK, transFeedbackObjId);
    //绑定变换反馈缓冲区
    glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 0, transFeedBufId);
    glBindTransformFeedback(GL_TRANSFORM_FEEDBACK, 0);
    glBindBuffer(GL_TRANSFORM_FEEDBACK_BUFFER, 0);
}

```

- 启动变换反馈，在绘制结束后停止变换反馈

```

void render()
{
    glviewport(0, 0, screenW, screenH);
    glUseProgram(program);
}

```

```

glBindVertexArray(vaoId);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D,imageTextureId);
glUniform1i(textureLocation,0);
glBindTransformFeedback(GL_TRANSFORM_FEEDBACK,transFeedbackobjId);
glBeginTransformFeedback(GL_TRIANGLES);
glDrawArrays(GL_TRIANGLES,0,6);
glEndTransformFeedback();
glBindTransformFeedback(GL_TRANSFORM_FEEDBACK,0);
glBindTexture(GL_TEXTURE_2D,GL_NONE);
glBindVertexArray(GL_NONE);
}

```

- 读取或者送到真正的渲染侧

```

void readTFBuffer()
{
    glBindBuffer(GL_TRANSFORM_FEEDBACK_BUFFER,transFeedBufId);
    void* rawData= glMapBufferRange(GL_TRANSFORM_FEEDBACK_BUFFER,0
                                    ,(3+2)*6*sizeof(GLfloat),GL_MAP_READ_BIT);
    for(int i = 0; i < 6; i++)
    {
        printf("buffer outPos[%d]= [%f,%f,%f],outTex[%d]=[%f,%f]",
               i,p[i * 5],p[i * 5+1],p[i * 5+2],i,p[i * 5+3],p[i * 5+4]);
        glUnmapBuffer(GL_TRANSFORM_FEEDBACK_BUFFER);
        glBindBuffer(GLTRANSFORM_FEEDBACK_BUFFER,0);
    }
}

```

3. Instanced Rendering:实例化渲染

- 概述

实例化是一种连续执行多条相同渲染命令的方法。并且每个命令所产生的渲染结果会有轻微的差异。其通过对具有相同顶点数据的几何体，赋予不同的空间位置、颜色或者纹理等特征，来创建不同的实例对象。

- 渲染API

- 针对每一个实例绘制一遍网格
 - void glDrawArraysInstanced(GLenum **mode**,GLint **first**,GLsizei **count**,GLsizei **instanceCount**);
 - void glDrawElementsInstanced(GLenum **mode**,GLsizei **count**,GLenum **type**,const GLvoid* **indices**,GLsizei **instanceCount**);
 - **mode**:绘制的图元类型，可取值为
GL_POINTS、GL_LINES、GL_LINELIST、GL_LINE_LOOP、GL_TRIANGLES、
GL_TRIANGLE_STRIP、GL_TRIANGLE_FAN
 - **first**:起始顶点索引
 - **count**:要绘制的顶点数量
 - **type**:指明indices中储存的元素索引类型，其可取值为
 - GL_UNSIGNED_BYTE,GL_UNSIGNED_SHORT,GL_UNSIGNED_INT
 - **indices**:指向元素数组的位置

- **instanceCount**:要绘制的实例数量
- 访问实例的数据
 - 通过glVertexAttribDivisor方法---实例化数组的方法
 - **index**:顶点属性索引
 - **divisor**:指定index位置的属性更新之间传递的实例数量（每隔多少个实例更换一次纹理、颜色等属性）
 - 当没有使用glVertexAttribDivisor或者divisor为0时，对每个顶点读取一次顶点属性。如果divisor为1时，则每个图元实例读取一次顶点属性（逐实例读取一个顶点属性）
 - 通过着色器内建变量gl_InstanceID作为缓冲索引来访问（该uniform数量有上限，所以大量实例化时须用实例化数组法）。

• 如何使用

- 着色器

```
//vertexShader
attribute vec3 position;
attribute vec3 offset;
attribute vec2 texCoord;
varying vec2 vTexCoord;
void main()
{
    gl_Position = vec4(position + offset, 1.0);
    vTexCoord = texCoord;
}
```

```
//fragmentShader
uniform sample2D image;
varying vec2 vTexCoord;
void main()
{
    gl_FragColor = texture2D(image, vTexCoord);
}
```

- 简略代码示例

```
void setVBO()
{
    GLfloat vertices[] = {...};
    vboId = createVBO(GL_ARRAY_BUFFER, GL_STATIC_DRAW,
sizeof(vertices), vertices);
    glEnableVertexAttribArray(glGetAttribLocation(_program, "position"));
    glVertexAttribPointer(glGetAttribLocation(_program, "position"), 3,
GL_FLOAT,
                           GL_FALSE, sizeof(GLfloat)*5, NULL);
    glEnableVertexAttribArray(glGetAttribLocation(_program, "texcoord"));
    glVertexAttribPointer(glGetAttribLocation(_program, "texcoord"), 2,
GL_FLOAT,
                           GL_FALSE, sizeof(GLfloat)*5,
NULL+sizeof(GL_FLOAT)*3);
}

void createTexture(){...}
```

```

void setOffset()
{
    GLfloat vertices[] = {
        0.1f, -0.1f, 0.0f,
        0.7f, -0.7f, 0.0f,
        1.3f, -1.3f, 0.0f,
    };
    offsetVBO = createVBO(GL_ARRAY_BUFFER, GL_STATIC_DRAW,
    sizeof(vertices),
                    vertices);
    glEnableVertexAttribArray(glGetAttribLocation(_program, "offset"));
    glVertexAttribPointer(glGetAttribLocation(_program, "offset"), 3,
    GL_FLOAT,
                    GL_FALSE, 0, NULL);
}

void render()
{
    glClearColor(1.0, 1.0, 1.0, 1.0);
    glClear(GL_COLOR_BUFFER_BIT);
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, _texture);
    glUniform1i(glGetUniformLocation(_program, "image"), 0);
    // 每次绘制之后，对offset进行1个偏移
    glVertexAttribDivisor(glGetAttribLocation(_program, "offset"), 1);
    glDrawArraysInstanced(GL_TRIANGLES, 0, _vertCount, 3);
}

```

4. Multiple Render Targets:多目标渲染

- 概述

MRT允许一次渲染到多个颜色缓冲区中。这样我们就可以将纹理体育场、法线贴图以及深度贴图渲染到一个帧缓冲对象。后面的渲染拿到这些数据，在进行下一步操作处理，所以该技术与延迟渲染很契合，所以常常搭配延迟渲染来处理灯光等浪费性能的渲染问题。OpenGL3.x中实现都支持的颜色附着的最小数量为4。

- 渲染API

- glGenFramebuffers和glBindFramebuffer命令初始化帧缓冲区对象
 - void glGenFramebuffers(GLsizei n,GLuint* framebuffers);
 - void glBindFramebuffer(GLenum target,GLuint framebuffer);
- glGenTextures和glBindTexture命令初始化纹理
 - void glGenTextures(GLsizei n,GLuint* textures);
 - void glBindTexture(GLenum target,GLuint texture);
- glFramebufferTexture2D或者glFramebufferTextureLayer命令将相关的纹理绑定到FBO
 - void glFramebufferTexture2D(GLenum target,GLenum attachment,GLenum textarget,GLuint texture,GLint level);
 - void glFramebufferTextureLayer(GLenum target,GLenum attachment,GLuint texture,GLint level,GLint layer);
- 使用glDrawBuffers用于指定接受颜色值的多个缓冲区，buffers是缓冲区枚举类型的数组
 - void glDrawBuffers(GLsizei n,const GLenum* bufs);
- 片段着色器中可以生成多个输出，与自己定义的缓冲区个数对应起来

```
layout(location = 0) out vec4 fragData0;
layout(location = 1) out vec4 fragData1;
layout(location = 2) out vec4 fragData2;
layout(location = 3) out vec4 fragData3;
```

OpenGL3.0实现都支持的颜色附着的最小数量为4，可以通过以符号常量
GL_MAX_COLOR_ATTACHMENTS为参数的glGetIntegerv查询颜色附着的最大数量。

5. 新的纹理功能

- 新的纹理格式
 - **Floating point textures**:浮点纹理，32位的浮点数据。新浮点格式：GL_R11F_G11F_B10F，红绿蓝通道各11位，蓝色通道10位
 - **3D Textures**:3D纹理。用(s,t,r)形式进行访问，r坐标选择3D纹理中需要采样的切片，(s,t)坐标用于读取每个切片的2D贴图
 - **Depth Textures**:深度纹理。每个像素值分别存储的是highp(32位浮点数)类型的深度值
 - **Vertex Textures**:顶点纹理。可以在VertexShader里检索的纹理
 - **NPOT Textures**:NPOT纹理。纹理贴图的宽高非2次幂
 - **R/RG Textures**:R/RG纹理。GL_RED单通道和GL_RG双通道的纹理
 - **Immutable Textures**:不可变纹理。加载纹理前指定纹理的格式和大小，进行一致性、内存的检查之后，纹理的格式和尺寸就不能更改
 - **2D Array Textures**:2D数组纹理。用(s,t,r)坐标访问，s、t代表二维纹理坐标，r值代表选择读取那一张纹理的数据
 - **LOD**:多细节层次。根据距离视点远近使用不同精度级别的模型
 - **MipMap**:多级纹理。根据距离视点远近来使用不同精细级别的纹理，消除最近采样时所产生的伪像
 - MipMap的思路是建立一个图像链称为mipmap链，这个mipmap链以原始图像开始，后续的每个图像的每个维度都是它上一个图像大小的一半，直到链底部的一个1X1大小的纹理。在mipmap链生成时需要用一些过滤方式对上一级图片进行过滤。
 - **Seamless Cube maps**:无缝立方体贴图。立方体贴图时一种通过三维空间坐标获取像素值的贴图方式。以三维立方体中心为原点，通过(s,t,r)坐标来采样纹理的颜色，在2.0中线性过滤核心落到立方体边缘时，过滤会只发生在单个面上，会造成伪像。在3.0中，如果过滤核心跨越立方体的一个面以上，核心将会从覆盖的每个面获取样本，以在立方体各面的边缘形成更平滑的过滤
 - **Sample Objects**:采样器对象。一个采样器可以被多个纹理单元共享，修改对象时可同时改变所有与之关联的纹理单元绑定到纹理对象
 - **ASTC**:自适应可伸缩纹理压缩。支持高动态范围成像(HDR)和三维压缩，OpenGL ES3.2版本开始支持该功能
- 新的纹理压缩方式

概念：

在OpenGL ES 2.0中，核心规范不定义任何压缩的纹理图像格式。因此，许多供应商提供了许多特定于硬件的纹理压缩扩展，但它们并不兼容。OpenGL ES 3.0引入了所有供应商必须支持的标准纹理压缩格式，其中ETC2和EAC (ETC2和EAC，爱立信纹理压缩，Ericsson Texture Compression，是由Khronos支持的开放标准，在移动平台中广泛采用。它是一种为感知质量设计的有损算法) 被作为OpenGL ES 3.0的标准纹理压缩格式。其中glCompressedTexImage2D用于加载2D纹理和立方图压缩图像数据。glCompressedTexImage3D用于加载2D纹理数组。

ETC2/EAC不支持3D纹理。 (只支持2D纹理和2D纹理数组, glCompressedTexImage3D可以用于加载供应商专用的3D纹理压缩格式)。

ETC2比OpenGL ES 2.0所用的ETC1压缩质量更高, 而且支持透明, 在Android设备上不需要打不同纹理格式的包了。

- **ETC2**

ETC2以向后兼容的方式来扩展ETC1, 它提供更高质量的RGB和RGBA压缩方式。在OpenGL ES 3.0中使用ETC2需要选择以下命令 (编解码器) :

GL_COMPRESSED_RGB8_ETC2 - 压缩RGB888数据, 即ETC1的后续数据。

GL_COMPRESSED_RGBA8_ETC2_EAC - 在完全支持alpha的情况下压缩RGBA8888数据 (支持透明)

GL_COMPRESSED_RGB8_PUNCHTHROUGH_ALPHA1_ETC2 - 压缩RGB8数据, 其中像素为完全透明或完全不透明。

- **EAC**

EAC的构建原理与ETC1 / ETC2相同, 但用于单通道或双通道数据。OpenGL ES 3.0中包含以下四个EAC编解码器:

GL_COMPRESSED_R11_EAC - 一个无符号数据的通道

GL_COMPRESSED_SIGNED_R11_EAC - 一个有符号数据的通道

GL_COMPRESSED_RG11_EAC - 两个无符号数据的通道

GL_COMPRESSED_SIGNED_RG11_EAC - 两个有符号数据的通道

在OpenGL ES中通过glCompressedTexImage2D代码来进行压缩。

- **用法**

- void glCompressedTexImage2D(GLenum target, GLint level, GLenum internalformat, GLsizei width, GLsizei height, GLint border, GLsizei imageSize, const GLvoid * data);
- void glCompressedTexImage3D(GLenum target, GLint level, GLenum internalformat, GLsizei width, GLsizei height, GLsizei depth, GLint border, GLsizei imageSize, const GLvoid * data);
 - **target**: 指定纹理目标, 可选GL_TEXTURE_2D、GL_TEXTURE_CUBE_MAP或GL_TEXTURE_2D_ARRAY
 - **internalFormat**: 纹理的压缩格式, 可选上文中GL_COMPRESSED_R11_EAC等
 - **width**: 以像素数表示的图像宽度
 - **height**: 以像素数表示的图像高度
 - **depth**: 以像素数表示的图像高度 (或2D纹理数组的切片数量)
 - **border**: 为兼容OpenGL保留, 在ES中总是取为0
 - **imageSize**: 以字节数表示的图像大小
 - **ata**: 包含图像的实际压缩像素数据, 必须能够容纳imageSize个字节

6. 同步信号和栅栏

- **概述**

同步信号和栅栏为应用程序提供了通知GPU在一组OpenGL ES操作执行完成之前先等待, 然后再将更多执行命令送入队列的手段。

- **渲染API**

- **创建同步对象glFenceSync**

```
GLsync glFenceSync( GLenum condition, GLbitfield flags);
```

- **condition**: 指定向同步对象发送信号必须符合的条件:
 - GL_SYNC_GPU_COMMANDS_COMPLETE
- **flags**: 指定控制同步对象行为的标志按位组合, 当前必须为0
- 阻塞GL的服务器端知道如变换反馈等OpenGL ES的操作执行完成


```
void glWaitSync( GLsync sync, GLbitfield flags, GLuint64 timeout);
```

 - **sync**: 指定等待其状态的同步对象
 - **lags**: 指定控制命令刷新行为的位域, 可选的值有:
GL_SYNC_FLUSH_COMMANDS_BIT
 - **timeout**: 指定等待同步对象获得信号的超过时间
- 删除同步对象


```
void glDeleteSync( GLsync sync);
```

 - **sync**: 指定需要删除的同步对象

7. 着色器新增的功能

- 概述

支持统一块 (用大括号包含矩阵等成员的uniform声明) 32位整数和32位浮点运算3.1版本还支持了Compute Shader完全用于计算任意信息的着色器, 可用于渲染, 也可用于与绘制三角形和像素无关的任务等功能

- 几何着色器

几何着色器将可以修改图元的类型和个数。当输出的图元减少或者不输出时, 实际上起到了剪裁图形的作用, 当输出的图元类型改变或者输出更多图元时起到了产生和改变图元的作用

- 曲面细分着色器

思路是导入一个多边形数量很少的模型, 之后将组成此模型的所有三角面细分成更小的三角面。其分为三部分

- 细分控制着色器 (TCS, Tessellation Control Shader)
- 图元生成 (PG, Primitive Generator) ----固定功能阶段
- 细分曲面计算着色器 (TES, Tessellation Evaluation Shader)

总结: TCS确定patch, PG计算坐标, TES模拟顶点着色器进行逐顶点操作

- UBO: 统一缓冲区对象

- 概述

UBO(统一缓冲区对象, OpenGL ES 3.1新功能),是用来存储uniform数据的缓冲对象, 通过它我们可以在程序之间或者shader之间共享uniform

UBO需要shader中声明uniform变量块, 采用这套机制的原因是:

如果程序中包含了多个着色器, 并且这些着色器使用了相同的uniform变量, 那么我们就需要为每个着色器分别管理这些变量。uniform变量是在程序链接时产生, 所以uniform变量的location会随着着色器的不同而发生变化。因此这些变量的数据必须重新产生, 然后用到新的location上。

而uniform block正是为了使在着色器间共享uniform数据变得更加容易而设计的。有了uniform block, 我们可以创建一个缓冲区对象来存储这些uniform变量的值, 然后将缓冲区对象绑定到uniform block上。当着色器程序改变的时候, 只需将同样的缓冲区对象绑定到新的着色器中与之相关的block即可。

在使用时, 可以通过使用glBufferData, glBufferSubData, glMapBufferRange和glUnmapBuffer来修改uniform缓冲对象内容, 不再使用glUniformXXX的方法。

在着色器中，一般使用默认的std140 uniform block布局，否则需要查询获得字节偏移和跨距，以便在uniform缓冲对象中设置uniform数据。std 140布局使用有OpenGL ES3.0明确规范定义的布局规范来进行特定打包，因此，使用std140布局，可以在不同的OpenGL ES3.0之间实现共享uniform block。

```
layout(std140)uniform lightBlock
{
    vec3 lighDirection;
    vec3 lighPosition;
};
```

■ 渲染API解释

■ 获得uniform block名字

```
void glGetActiveUniformBlockName(GLuint program,GLuint index, GLsizei
bufSize,GLsizei* length,GLchar* blockName);
```

- Program: program对象
- index:要查询的索引
- bufSize:名字的字符数
- Length:如果这个值不是空，会被写入uniform名字的字符数(不包括终止字符)
- name:被写入uniform的名字，最多有bufSize个字符，以终止字符结尾

■ 获得uniform block的其他属性

```
void glGetActiveUniformBlockiv(GLuint program,GLuint index,GLenum
pname,GLint* params);
```

- Program:program对象
- index:要查询的索引
- pname:要查询的属性
- Params: 查询结果

■ 根据名字获得uniform block index

```
GLuint glGetUniformLocation(GLuint program,const GLchar* blockName)
```

- Program: program对象
- blockName: uniform block的名字

■ 将uniform block index和program中的一个binding point进行绑定：

```
void glUniformBlockBinding(GLuint program,GLuint blockIndex,GLuint
blockBinding);
```

- Program: program对象
- blockIndex:uniform block index
- blockBinding: uniform缓冲对象绑定点

■ 将一个uniform buffer object 和这个binding point绑定：

```
void glBindBufferRange(GLenum target,GLuint index,GLuint buffer,GLintptr
offset,GLSizeiptr size);
```

```
void glBindBufferBase(GLenum target,GLuint index,GLuint buffer);
```

- Target: GL_UNIFORM_BUFFER,GL_TRANSFORM_FEEDBACK_BUFFER
- Index: 绑定索引
- buffer: 缓冲对象
- offset: 缓冲对象的起始偏移字节数
- size:能从缓冲对象读取或者写入缓冲对象的数据量

○ 二进制格式的着色器代码

- 概述

在OpenGL ES 2.0中可以使用二进制格式存储着色器代码，但是必须在运行时链接到程序中，在3.0中完全链接过的二进制文件可以保存为离线的形式，并提供相关接口，不需要离线工具就可以使用二进制代码，减少了加载时间

- 渲染API

- 检索program Binaries

```
void glGetProgramBinary(GLuint program, GLsizei bufSize,GLsizei*  
length,GLenum binaryFormat,GLvoid* binary);
```

- Program:program对象
 - bufSize:可能被写入到binary的字节最大值
 - length:写入binary的字节数
 - binaryFormat:binary格式
 - binary:binary数据

- 检索完成之后，可以将其保存到文件系统，或者调用下面方法读回到OpenGL ES

```
void glProgramBinary(GLuint program,GLenum binaryFormat,const GLvoid*  
binary,GLsizei length);
```

- Program:program对象
 - binaryFormat:binary格式
 - binary:binary数据
 - length:写入binary的字节数

4.2 四元数、欧拉角、矩阵的区别与联系

1. 矩阵欧拉角四元数优缺点

1. 比对表格

任务/性质	旋转矩阵	欧拉角	四元数
在坐标系间	能	不能	不能
连接或者增量旋转	能，比四元数慢、矩阵蠕变	不能	能，比矩阵快
差值	不能	能，万向锁	slerp平滑差值
易用程度	难	易	难
在内存或文件中存储	9个数	3个数	4个数
对给定方位的表达式是否唯一	是	不是，有多个	不是，有两种
是否导致非法	矩阵蠕变	任意三个数都可以构成欧拉角	误差积累

2. 适用地方

- 欧拉角最容易使用，当需要为世界中的物体指定方位时，欧拉角能简化人机交互
包括直接的键盘输入方位、代码中指定方位、调试中测试
- 坐标系之间转换向量，那么使用矩阵。可以把旋转矩阵保存为其它数据，使用时转换为矩阵。

- 需要大量保存方位数据时，使用欧拉角或四元数。欧拉角将少占用25%的内存，但它在转换矩阵时要稍微慢一些。如果动画数据需要嵌套坐标系之间的连接，四元数是最好的选择
- 平滑的插值只能使用四元数完成

1. 矩阵

1. 转置与逆矩阵

- 转置矩阵

将矩阵的行列互换得到的新矩阵称之为转置矩阵，转置矩阵的行列式不变

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix}^T = \begin{bmatrix} a_{00} & a_{10} & a_{20} & a_{30} \\ a_{01} & a_{11} & a_{21} & a_{31} \\ a_{02} & a_{12} & a_{22} & a_{32} \\ a_{03} & a_{13} & a_{23} & a_{33} \end{bmatrix}$$

转置矩阵运算性质为：

1. $(M^T)^T = M$
2. $(M + N)^T = M^T + N^T$
3. $(kM)^T = kM^T$
4. $(M * N)^T = M^T * N^T$

- 逆矩阵

存在矩阵M以及矩阵N，假如 $M * N = \text{矩阵I}$ (I为单位矩阵)，那么矩阵M和矩阵N互为逆矩阵

逆矩阵有一个很大的作用就是还原变换。假设M与N互为逆矩阵，那么 $(M * N * \text{齐次坐标A})$ 得到还是原来的其次坐标A。从仿射空间来看，仿射空间A在经过矩阵M变换到仿射空间B，那么仿射空间B经过M的逆矩阵N变换就还原成了仿射空间A

逆矩阵的运算性质为：

2. 2D旋转

3. 3D旋转

4. 法线与矩阵的关系

2. 四元数

3. 欧拉角

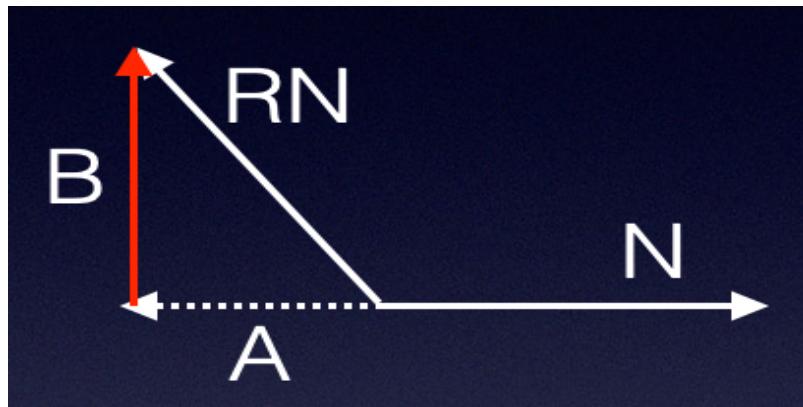
4. 相互转换

5. 施密特正交化(Gram-Schmidt process)

在线性代数中，如果内积空间上的一组向量能够张成一个子空间，那么这一组向量就称为这个子空间的一个基。

<https://blog.csdn.net/baimafujinji/article/details/6479143>

对于二维空间中，求两个向量之间的施密特正交化



如上图中，我们呢可以根据已知的向量RN以及N求出A与B向量

按照向量内积的定义， $\text{dot}(\text{RN}, \text{N})$ 那么我们可以求出A向量与RN向量的夹角

那么 $A = \text{dot}(\text{RN}, \text{N}) * \text{RN}$

那么根据向量的减法的定义向量 $B = \text{RN} - A$

最终我们得到， $B = \text{RN} - \text{dot}(\text{RN}, \text{N}) * \text{RN}$

那么我们根据已经垂直的向量B与N，可以求出另外一个轴，当然，如果我们愿意，确实是完全可以求出来一个三维的坐标系的

6. 矩阵的行空间、列空间解释

1. 线性组合

2. 行空间

3. 列空间

7. 相机矩阵的推导

我们设定三个参数，相机位置CameraPos、Lookat以及up向量计算出相机矩阵

<https://blog.csdn.net/popy007/article/details/5120158>

8. 其次坐标的几何意义

4.3 渲染管线

1. 整体渲染管线流程

图形渲染开发，首先牢靠的基础知识是必须的，也是必要的。那么首先要了解的一个基础的东西，就是渲染管线。渲染管线是整个渲染的流程，每一个步骤都是很重要，需要掌握并且熟悉的，下面说一下渲染管线的整体的流程。

顶点缓冲区\数组对象-----顶点着色器-----图元装配-----光栅化-----片元着色器-----像素归属测试-----剪裁测试-----模版测试-----深度测试-----混合-----抖动-----前往帧缓冲

2. 渲染管线具体步骤

1. 顶点缓冲区/数组对象

OpenGL解释是客户端空间，通俗来讲，顶点缓冲区是一个地址空间，也可以说是拷贝空间。其具体作用就是将工程师组装好的顶点属性数据拷贝到该内存空间，然后GPU将数据读取过来以供使用，类似于一个中转站。

2. 顶点着色器

顶点着色器是实现了顶点操作的通用性可编程方法，比如可以在顶点着色器进行坐标空间的转换等。

顶点着色器中接收并处理顶点数据转换为片元数据输入到片元着色器。在3.0中可以在图元装配与光栅化之前将数据输出到缓冲区中，这种技术被称为映射缓冲。

3. 图元装配

图元装配其分为两个步骤，第一步骤是几何对象转化为图元，第二步骤，对图元进行优化。将点或者三角形或者四边形等几何对象切分成连续的图元，以供着色使用。优化就是对已经切分成连续的图元进行判断，是否是在视景体之内，视景体之外的舍弃，保留视景体之内的图元，然后就是图元本身背面或者正面，保留正面，舍弃背面。

在优化中，进行视景体剪裁之后，图元的坐标已经转化为屏幕坐标了。

4. 光栅化

图元装配完后，几何对象转化为图元。但是图元比像素大太多了，远远达不到着色的阶段。那么就将这些图元在进行切分，切分到更加接近像素大小的图元。OpenGLES称这些更小的图元为片元。当然这个阶段的片元，其属性包括：屏幕坐标、颜色、纹理坐标等。

5. 片元着色器

片元着色器实现了对片元操作的通用性编程方法。接收光栅化处理后的二维数组片段，对其二维数组片段进行处理，然后输出颜色、深度、模版等着色数据。

6. 像素归属测试

OpenGLES内部会进行上下文的判断，当前上下文只显示当前上下文渲染的着色数据

7. 剪裁测试

确定像素是否作为OpenGLES状态的一部分剪裁矩形范围内，如果该片段位置处于剪裁区域之外，那么将会被舍弃。

8. 模板/深度测试

该测试会在输入片段的模版以及深度上进行，来确定该片段数据是否被拒绝

9. 混合

将新生成的片段颜色值与保存在帧缓冲区位置的颜色数据按照混合公式与参数进行混合

10. 抖动

可用于最小化因为使用有限精度在帧缓冲区保存颜色值而产生的伪像素

3. 总结

OpenGLES3.x将Alpha测试和逻辑操作剔除掉了，它们存在与OpenGL 2.0和OpenGLES1.x中。而不需要Alpha测试的原因是，因为片元着色器可能会因为Alpha测试而抛弃片段，因此Alpha测试可以在片元着色器中编程实现。

4.4 渲染中的混合操作

1. OpenGLES中混合介绍

1. 混合运算公式

混合是发生在深度测试之后，抖动之前的操作。其片段通过了深度测试，其颜色属性将与颜色缓冲区存在颜色进行混合的操作。OpenGLES提供了一套混合的API以供开发者使用，其中开发者可以遵照混合运算符来进行混合

$\$C_{final}$:是最终的颜色混合值

$\$F_{\{source\}}$:源目标因子，也可以称之为 α 因子
 $\$C_{\{source\}}$:源目标颜色，当前渲染的物体的片元的颜色值
 $\$F_{\{destination\}}$:目标因子， β 因子
 $\$C_{\{desination\}}$:目标颜色，颜色缓冲中的颜色
Op: 混合运算符

2. 混合范围

- API函数

```
glEnable(GL_BLEND);glDisable(GL_BLEND);
```

这两个函数是开启与关闭混合的API，它主要的作用是告诉管线，混合的范围是那里

- 伪代码

```
glEnable(GL_BLEND);
a.draw();
b.draw();
c.draw();
glDisable(GL_BLEND);
```

告诉管线，abc三个物体进行颜色混合。

3. 源颜色与源因子、目标颜色与目标因子

- API函数

```
glBlendFunc(GLenum factor,GLenum defector);
```

该函数是指定混合系数，设置源目标因子、目标因子

```
glBlendFuncSeparate(GLenum srcRGB,GLenum dstRGB,GLenum srcAlpha,GLenum dstAlpha);
```

前面两个参数是指定RGB的源因子与目标因子

后面两个参数是指定Alpha的源因子与目标因子

4. 常量混合

- API函数

```
glBlendColor(GLfloat red,GLfloat green,GLfloat blue,GLfloat alpha);
```

其参数是常量混合颜色值

按照设置混合参数带有(\$R_c\$, \$G_c\$, \$B_c\$, \$A_c\$)的混合方式进行混合，此函数被称之为自定义混合，按照程序开发设定的否混合颜色进行混合，但是其需要搭配特定非几个混合参数使用

5. 混合运算符的设置

设置完源颜色、源因子与目标颜色、目标因子后。就需要设置运算符了，让其按照指定的运算符进行组合。默认运算符是GL_FUNC_ADD。GL_FUNC_SUBTRACT运算符从输入片段值中减去帧缓冲中的换算值，而GL_FUNC_REVERSE_SUBTRACT是反过来的

```
glBlendEquation(GLenum mode);
```

Mode:有效值为GL_FUNC_ADD、GL_FUNC_SUBTRACT、
GL_FUNC_REVERSE_SUBTRACT GL_MIN GL_MAX

```
glBlendEquationSeparate(GLenum modeRGB,GLenum modeAlpha);
```

modeRGB:为红绿蓝分量设置混合运算符

modeAlpha：指定Alpha分量混合运算符

2. 颜色缓冲区与深度缓冲区的介绍

1. 颜色缓冲区

颜色缓冲区由前台缓冲区与后台缓冲区组成

前台缓冲区负责显示，后台缓冲区负责构造下一个图像

颜色缓冲区有三个分量，用于存储红、绿和蓝分量以及可选信息的alpha分量的存储

场景中渲染的物体最终成像的像素颜色会存储到该缓冲区中，如果开启了混合并且深度测试通过，则输入片段的颜色值将会与颜色缓冲区的颜色值进行混合，然后混合颜色覆盖颜色缓冲区的值

2. 深度缓冲区

记录每个着色器片段与视点最近的距离值，对于每个新的输入片段，将其与视点的距离和存储值比较。默认情况下，如果输入的片度的深度值小于深度缓冲区中保存的值，则输入片段的深度值代替保存在深度缓冲区的值

3. 半透明混合为什么会出现问题

我们先来说一下，混合操作是在深度缓冲区测试之后。当片段在深度缓冲区测试通过之后，才会与颜色缓冲区保存的该片段位置的颜色进行组合。这样，如果一个片段在深度缓冲区测试没有通过，是会被丢弃掉的，那么也无法与颜色缓冲区的保存了该片段位置的颜色进行组合。

那么对于不透明物体来说，这样的渲染是没有问题的，在同一屏幕坐标位置的与视点距离近的片段遮挡与视点距离远的。渲染出来的结果是没有问题的。

深度缓冲区测试打开、混合打开

比如说，不透明物体与半透明物体在一起渲染的时候。

如果我们先渲染半透明物体。如果屏幕半透明的片段距离视点的距离比不透明的片段视点的距离小，半透明物体的片段的深度值已经写入深度缓冲区，那么不透明的片段在经过深度缓冲区测试的时候，会被丢弃掉，这是深度缓冲区测试的规定。但是我们想看到是透过半透明区域看到不透明的物体。所以这从视觉感受来讲，渲染已经出错了。

如果我们先渲染不透明物体。不透明片段的深度值将被写入深度缓冲区，不透明片段的颜色值将被写入颜色缓冲区。然后渲染半透明物体，半透明片段的距离视点距离比不透明视点小，那么将替代深度缓冲区的保存的不透明的片段距离视点的距离值，半透明片段的颜色值也将替换掉颜色缓冲区中保存的不透明物体的片段值。那么这也是一个错误的渲染结果。

这就是半透明与不透明物体渲染在一起渲染的时候，开启深度缓冲区测试后常常出现的错误的渲染结果，但是又不得不开启深度缓冲区测试。

简而言之，如果想要保证一个不透明物体的正确渲染，那么我们是需要开启深度缓冲区测试的，为什么，因为每一个复杂的3维物体，其本身也是有深度层次的。如果不开启深度缓冲区测试，那么不透明的物体就会渲染出一个错误的结果。那么我们该如何根据深度缓冲区测试与混合的理论知识，达到半透明不透明渲染的正确的显示那？

4. 如何解决半透明渲染问题

1. 次序渲染

首先我们来说一下深度缓冲区测试理论知识，其实深度缓冲区测试，具体来说是分为两个比较重要的东西。一个是当前片段位置的片段与视点的距离值与深度缓冲区保存的当前片段位置的片段与视点的距离的比对，一个是是否将当前片段位置的片段距离视点的距离写入深度缓冲区记录（其中也包括是否替换颜色缓冲区的片段颜色值）。

上面我们已经分析了半透明渲染具体出现的问题以及背后的理论知识，这里笔者直接给出一个解决方案，然后在分析该解决方案为什么可以。

我们首先开启深度缓冲区测试与混合，先渲染不透明物体，然后在关闭深度缓冲区测试的写入操作，渲染半透明物体，关闭深度缓冲区测试与混合然后得到正确的渲染结果。

原理：当前不透明片段位置的片段距离视点的距离会被写入深度缓冲区，然后将当前不透明片段位置的片段颜色写入颜色缓冲区，然后渲染半透明物体，其深度写入是关闭的，那么当前位置不透明片段距离视点的距离会与深度缓冲区保存的不透明的片段距离视点的距离比对，片段测试通过，但是不会将当前片段位置的半透明片段的距离视点的距离替换掉深度缓冲区保存的不透明片段距离视点的距离，也不会将颜色缓冲区中不透明片段的值替换掉。那么深度缓冲区测试结束，那么开始进行混合，就是当前位置的半透明片段的颜色值与颜色缓冲区保存的当前片段位置的不透明片段颜色值进行混合，得到正确的渲染结果。

这个过程，如果按照pass来定义的话，那么是经历了两个pass的处理，这是有点浪费性能的，但是能够得到正确的半透明的渲染效果。并且这种渲染技术手段是跟不透明与半透明渲染的顺序有关系，所以称之为半透明的次序渲染。

2. 次序无关半透明深度剥离

有一个单独的章节会介绍该技术手段

3. 总结

半透明渲染还有别的方式，比如说生成一张半透明物体的深度图，以此来作为参考值来对不透明物体进行渲染，或者半透明与半透明进行渲染混合，都是可以的。其背后的原理都是保证不透明物体渲染的正确，然后将半透明的物体与不透明或者半透明物体进行一个混合，以此来达到正确的渲染结果。

4.5 Z-Fighting问题的出现以及处理

1. Z-Fighting问题的描述

我们都了解，在渲染三维场景的时候，我们一般都是采用透视投影矩阵，这样三维场景会产生近大远小的效果，很符合现实中人眼观看世界的情况。但是渲染场景时经常会遇到场景中离摄像机较远的物体会有闪烁的现象，而场景中离摄像机比较近的地方反而会很少会产生物体闪烁的现象。

这是为什么那？

如果我们采用了正交投影矩阵后，闪烁的问题会极大的解决，这又是为什么那？

2. 为什么会产生Z-fighting现象

2.1 原因1

场景中渲染多个三维物体的时候，几个三维物体的摆放位置很接近，导致咋爱深度缓冲测试的时候，会产生精度的误差，然后会导致几个物体之间的片段值有的时候是a通过，有的时候是b通过，导致交替显示这几个物体的颜色值，然后那就会产生闪烁疯现象。

2.2 原因2

采用透视投影矩阵渲染的场景，其深度缓冲区存储的深度值，是ndc空间中的深度值。而ndc空间的深度值是经由透视空间转换过来的，ndc空间的深度值与透视空间的深度值并非是我们想象中的线性转化的，而是非线性转化的。大家都知道，透视空间转换到ndc空间会有一步透视线除法，是除以z。这样就会导致，离视点越近的物体的片段深度值是越精确的，离视点越远的物体的片段值其深度值是越不精确

的，这样也会导致z-fighting现象。

而采用正交投影矩阵渲染场景，其变换是线性的，为什么？因为其投影空间转换为ndc空间的时候采用的是透视除法是除以1，所以其片段的深度值是线性的，这样除非你把两个物体设置额位置非常接近，否则是产生不了z-fighting这种现象额。

那么为什么透视投影矩阵渲染场景，在投影空间转换到ndc空间时，透视除法需要除以z值那？

3. 从透视投影矩阵计算的角度解释以及如何避免

3.1 从矩阵计算的角度看待问题

三维空间的物体在经历的图元转配以后，会经历剪裁与淘汰这两步操作。其中在剪裁中经历坐标系转换的步骤是：

模型坐标系-----世界坐标系-----相机坐标系-----投影坐标系-----标准设备空间-----视口变换-----光栅化阶段

投影坐标系转换到标准化设备空间时，需要将xyz坐标转化到-1到1之间，也可以说标准设备空间(NDC)是一个立方体空间。其借助于矩阵，矩阵的一个几何意义就是将一个物体从一个坐标系转化到另外一个坐标系。先看一下完整的透视矩阵：

其中n表示near也就是近平面，f表示远平面，t表示视口top，b表示视口bottom，r表示视口right，l表示视口left

那么现在使用坐标(Xe,Ye,Ze)表示在相机坐标系下的坐标，坐标(Xc,Yc,Zc)表示在投影坐标系下的坐标，坐标(Xndc,Yndc,Zndc)表示标准设备空间坐标，其转换关系如下：

经过复杂的数学计算，得到的数学结果是：

从该数学公式中，坐标系的数学转化的最终的标准设备空间下的坐标值是与Ze成反比的。这就是为什么距离摄像机近的，其深度精确，距离摄像机远的其深度不精确。

另外，透视投影矩阵是利用相似三角形来计算的，其会将Ze的值带入计算，在进入NDC空间后，在深度上是要进行归一化处理的，所以其归一化的Wc与Ze相等的。正交投影矩阵是没有相似三角形的，所以其Wc是为1.

3.2 如何避免z-fight问题

- 从数学层面上来解决这个问题，其中，最常见的就是减小近平面与远平面之间的绝对距离，可以做到一定程度上的减少z-fighting问题。
- 根据物体距离摄像机的距离，在渲染时，进行计算，避免距离摄像机较远位置的物体相距太近
- 使用OpenGL ES中内部提供的API，多边形偏移，但是会造成性能的浪费，可结合空间管理算法与LOD算法使用

4. 深度值的非线性到线性的转化

在着色器的api中，`gl_fragCoord`其保存的z值是非线性的深度值，是将NDC空间下片段的深度值从-1到1转化到0-1之间的值。

在延迟渲染灯光或者SSAO的时候，我们想用这个值，那么直接使用非线性的深度值来计算，那么得到的结果显然是不正确的，那么就需要将这个非线性的值转化到线性的值，这样才能得到正确的结果。

从上面的Zn的计算公式来看，显然可以做到，其计算公式为：

这是一个相等的数学公式，简单的变换，就可以得到Ze的值，也就是摄像机空间下的深度值，拿到这个深度值，就可以进行后续的很多事情。

4.6 FBO相关问题

4.7 延迟渲染

4.8 次序无关半透明渲染

1. 基础理论

1.1 概述

笔者最近通过某搜索引擎检索了一把半透明渲染，说实话，有些写的比较粗略，看的模模糊糊的，有的写的比较细致，但是还是没有看懂。整的笔者自己都懵了。所以笔者决定深入浅出的说一下这个半透明渲染。当然还有别的半透明渲染的方法，笔者在这里只是单纯的聊一下Depth-Peeling这个技术手段，如有不妥欢迎拍砖。

1.2 基础概念

深度剥离处理手段，通过对物体的多次渲染，以跟观察者的距离从近到远的顺序，每次剥离出一层，之后将剥离出来的颜色再按从后向前的正确顺序进行透明混合，从而保证得到正确的颜色结果。但是缺点是，效率不是很高，假如半透明的物体后面有n个物体，需要跟半透明进行混合，那得进行n次渲染，这个是不是感觉有点性能拖累了。所以在此基础延伸出了双向深度剥离，说实话，这个名字好洋气。具体到理论上就是，两个方向剥离，一个从前往后，一个从后往前。相当于，以前一个方向干活，现在两个方向。所以效率就提高了，这里不在详谈双向深度剥离，只谈深度剥离。

1.3 渲染管线

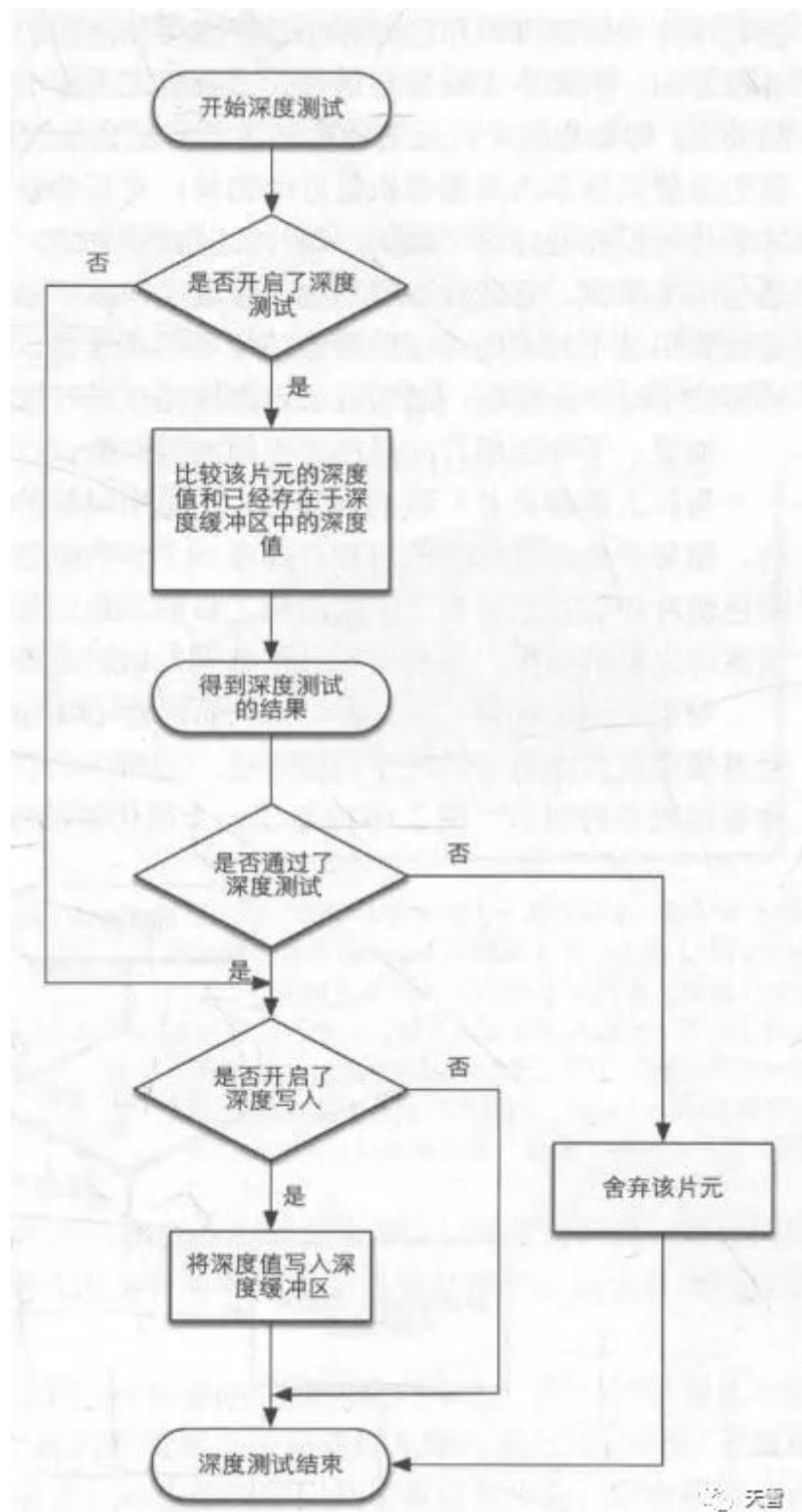
笔者在这里又要老生常谈一把渲染管线了，但是这次，笔者只是单纯的说一下片元之后的管线顺序，读者不必觉得啰嗦（其实我觉得蛮啰嗦的）。

以OpenGL ES2.0/3.x管线为例（再次特殊说明一下，alpha测试只存在于OpenGL ES1.x与OpenGL2.0，不要在把alpha测试给混了，笔者以脑袋做担保）

片元着色器->像素归属测试->剪裁测试->模版测试->**深度测试**->混合->抖动->送进帧缓冲。

注意加粗的地方，我们所说的东西，就是类似于干了这个家伙干的事情，但是具体来讲有点小区别。

现在先来说说这个加粗的家伙，深度测试到底干了点啥。



© 天雪

而我们要干的就是类似于深度测试的操作，只不过是在片元着色器进行的编程化操作。

这个深度测试，按照笔者的理解，也就是两个关键点

一个是根据当前渲染的深度与深度缓冲区的深度进行比对，

一个是深度缓冲区的深度是否让当前深度值进行覆盖，也就是是否重置深度缓冲区的值。

那么深度测试完成后，会变成什么样子那？就是把场景中z值最小的点输出到屏幕上去了。

先加深深度测试印象，下面说到深度剥离的时候，读者可能就明白，这不就是编程版本的深度测试嘛（笔者是这么想滴）。

2. 深度剥离实现

首先不透明渲染一遍，得到一层FBO深度，然后交给下一片元着色器进行深度比对，然后再生成FBO的深度信息，在交给下一层使用。如此循环，得到正确的结果。

当然其中会有颜色的blend，这个下一篇章细说。

简而言之，就是先对半透明来一次fbo得到深度信息，作为第二次物体渲染在着色器的深度比对值，然后通过进行颜色混合，不通过的那就只能discard了，第三次拿第二次的深度值进行对比。如此循环，在每一次深度剥离的同时进行颜色的混合。

其实也比较容易理解，先确定半透明物体的深度，如果深度比半透明的低，那么半透明就被不透明遮挡了，就不需要显示了，如果深度比半透明小，那就要透过半透明看到不透明。

现在说完了深度剥离，其实每一步骤深度剥离完成后，需要进行混合，就是半透明物体之后的物体要显示出来，那就得跟半透明的颜色进行混合。其实具体操作就是这个样子。

笔者到现在已经将深度剥离的技术方案说完了，其实，细心的读者自己心里面已经很明白了，渲染管线真的很重要，理解好渲染管线，真的可以做非常多的事情。

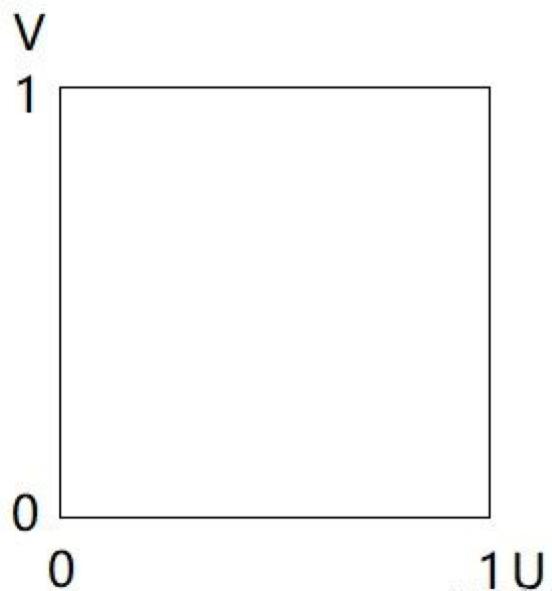
4.9 柏林噪声、细胞噪声

1. Value Noise

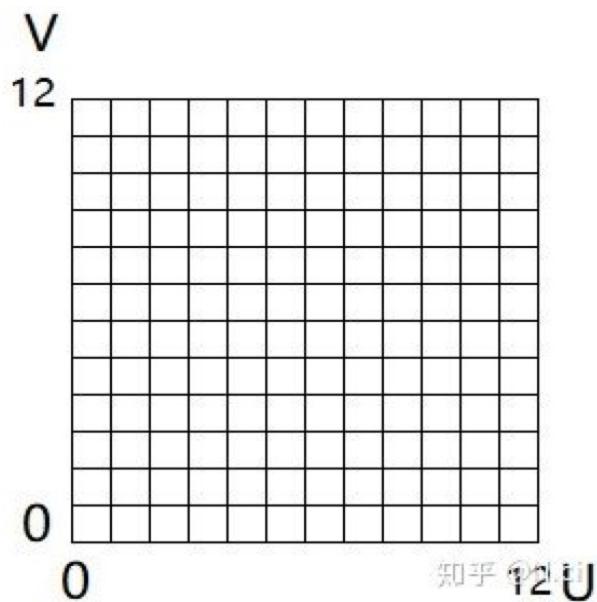
1.理论

- 纹理坐标

纹理坐标左下角为坐标原点。



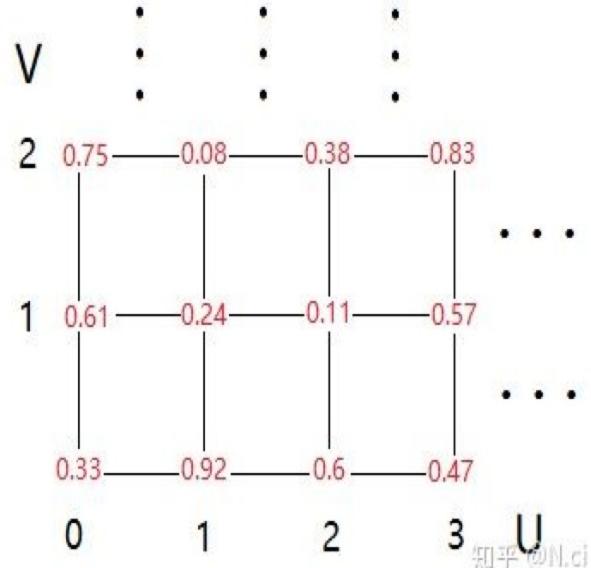
- UV缩放



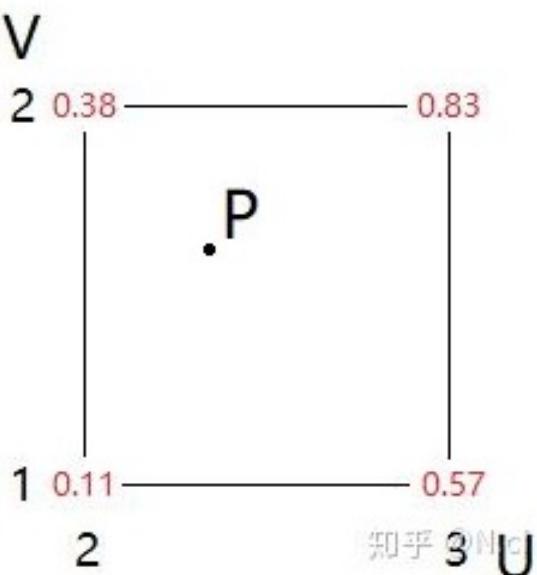
- 随机值

在网格中的每个交点都计算出来一个随机值，用于Value噪音值的计算。

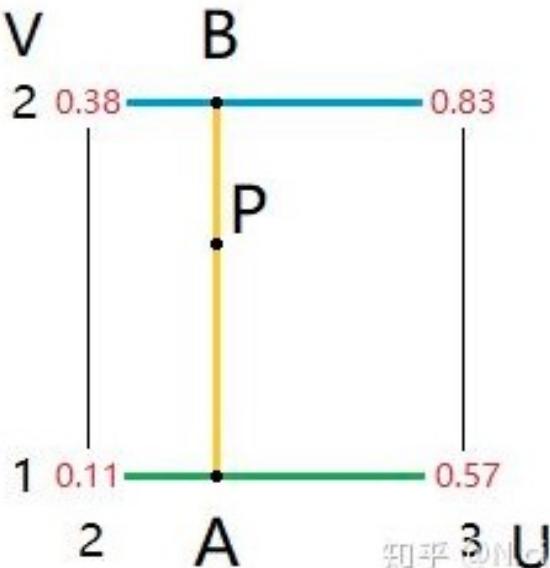
- 比如说用哈希函数来计算随机值。



- 假如取值UV(2,3,1,6)位置点



- 根据p值的坐标，对当前的这一个晶格的下侧 (2, 1) 到 (3, 1) 求一个插值A，在对上侧 (2, 2) 到 (3, 2) 求一个插值B，最后在AB之间齐聚插值P。



- 插值方法

线性插值：

$$A = 0.11 * 0.7 + 0.57 * 0.3 = 0.248$$

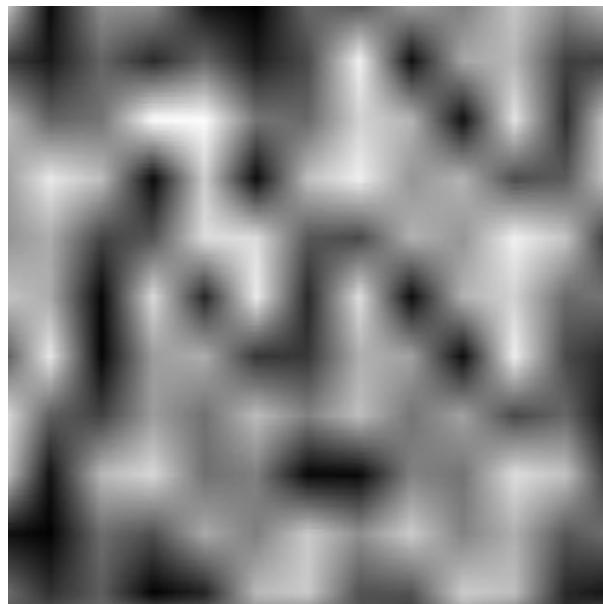
$$B = 0.38 * 0.7 + 0.83 * 0.3 = 0.515$$

$$P = 0.248 * 0.4 + 0.515 * 0.6 = 0.4082$$

使用该值作为最后的灰度值，便是最终的Value Nose。

注意 Δ ：当然也可以换成其他的插值方式来进行插值，不同的插值首发会产生不同的效果。

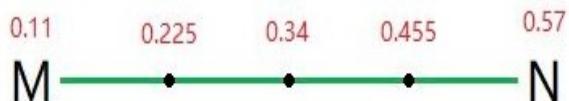
2. 效果及原因



晶格之间的过渡是有梯度的，过渡并不太顺滑。

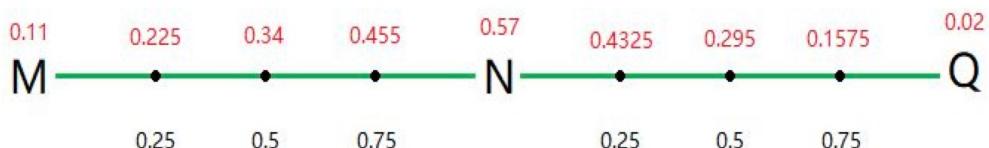
- 原因

Value Nose插值方式是这个样子：

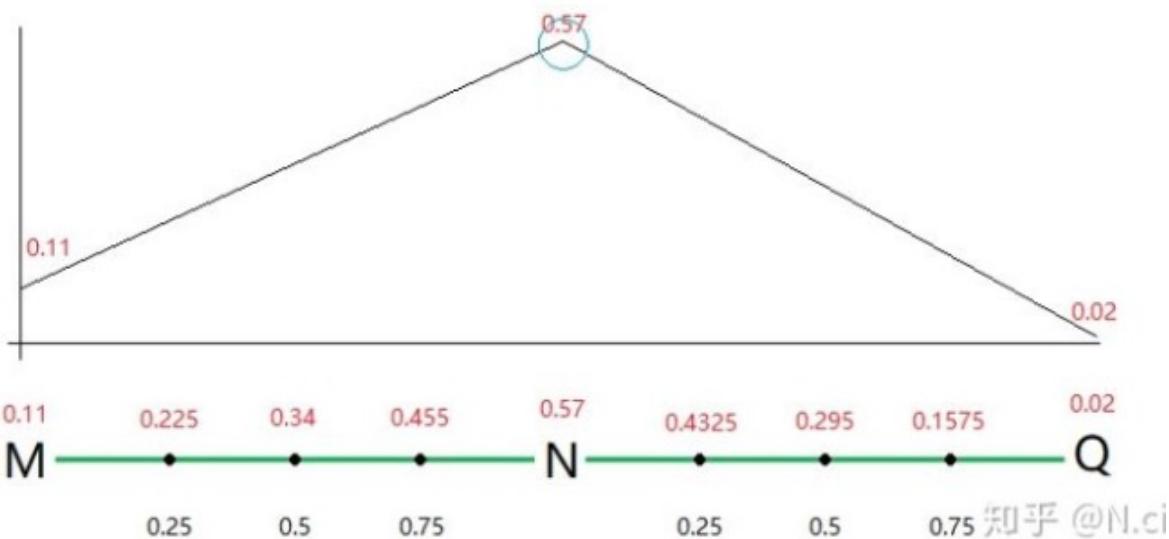


0.25 0.5 0.75 知乎 @N.ci

从M到N，插值的灰度均匀递增。我们假设后面还有一个节点Q，并且Q的灰度值为0.002



如果将这些值形成一个连续的图像



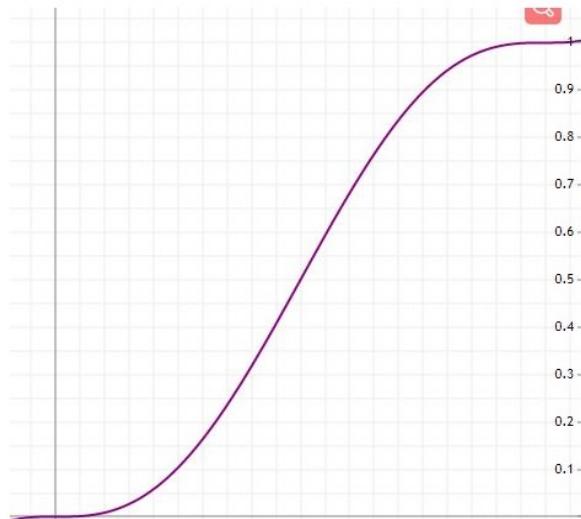
因此，大家可以采用另外的一个平滑的插值函数来计算插值的因子，就可以达到平滑的效果。

2. Perlin Noise

1. 理论

- 插值因子的计算

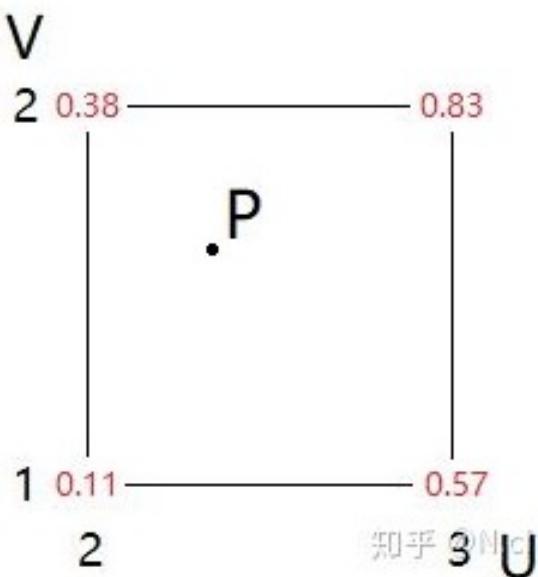
柏林噪声之父的噪声插值因子的计算为 $y= 6 * \text{pow}(x,5) - 15 * \text{pow}(x,4) + 10 * \text{pow}(x,3)$ 。连续性图片表示。



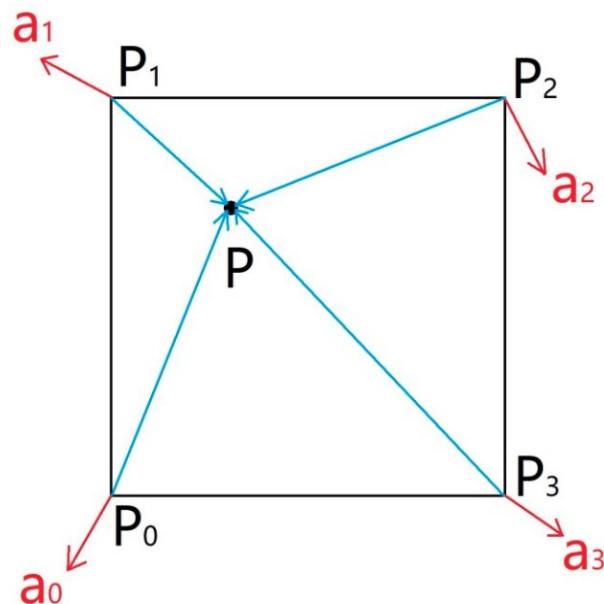
$y = 6x^5 - 15x^4 + 10x^3$ 在区间 $[0, 1]$ 下的演算结果

插值因子是相当平滑的，比函数 $y=x$ 这种平滑了太多。

- 生成晶格。



- Perlin Noise的晶格中的计算稍微不同。



红色的向量可以使用hash函数来随机生成。

而蓝色的向量是四个角指向p点位置的向量

每个位置对对应的红蓝向量求点积。

最后按照Value的插值方式，只不过换为Perlin Noise的插值因子.

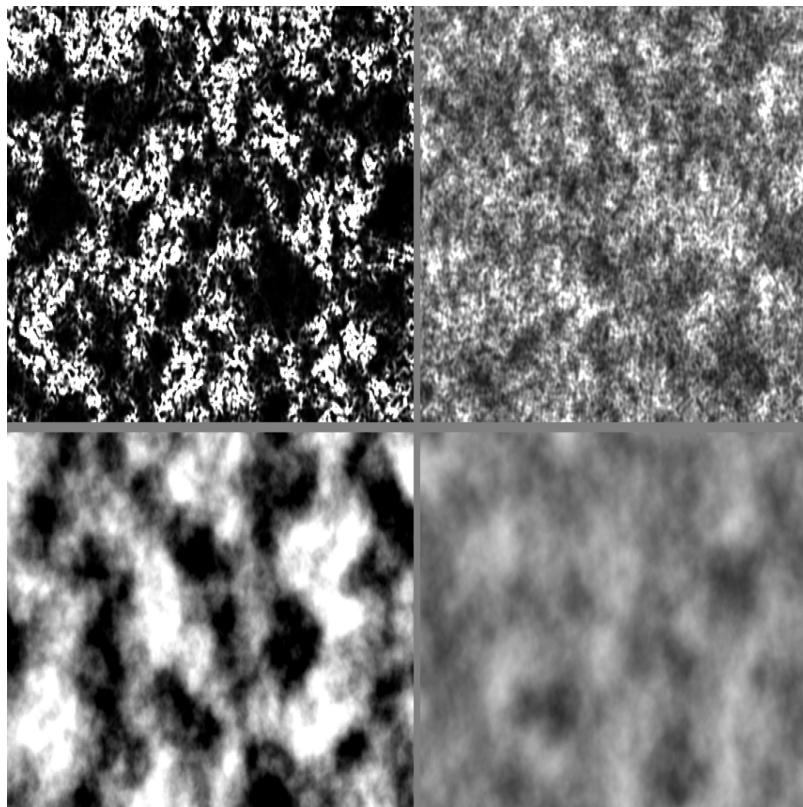
2. 说着色器实现

- 着色器实现

```
vec2 hash(vec2 p)
{
    p = vec2(dot(p, vec2(127.1, 311.7)), dot(p, vec2(269.5, 183.3)));
    return -1. + 2.*fract(sin(p+3.)*53758.5453123);
}

float perlinNoise(vec2 p)
{
    vec2 pos = floor(p); //向下取整
    vec2 f = fract(p); //向上取整
    vec2 u = f * f * f * (6. * f * f - 15. * f + 10.);
    //四个角指向随机坐标的向量
    vec2 v0 = f - vec2(0., 0.);
    vec2 v1 = f - vec2(1., 0.);
    vec2 v2 = f - vec2(0., 1.);
    vec2 v3 = f - vec2(1., 1.);
    //四个角上的随机的向量
    vec2 j0 = hash(pos + vec2(0., 0.));
    vec2 j1 = hash(pos + vec2(1., 0.));
    vec2 j2 = hash(pos + vec2(0., 1.));
    vec2 j3 = hash(pos + vec2(1., 1.));
    //点乘的结果
    float d0 = dot(j0, v0);
    float d1 = dot(j1, v1);
    float d2 = dot(j2, v2);
    float d3 = dot(j3, v3);
    //bottom的x方向插值
    float bx = mix(d0, d1, u.x);
    //top的x方向插值
    float tx = mix(d2, d3, u.x);
    //y方向的插值
    float result = mix(bx, tx, u.y);
    return result;
}
```

- 效果



3. 细胞噪声

1. 理论

2. 着色器实现

4.10 Bilnn_Phone/Phone光照模型（点光源、聚光灯、平行光）

4.11 多光源渲染问题

4.12 法线贴图

1. 切线空间

切线空间，是基于物体表面的空间，是在3D渲染领域众多坐标系之一。其一个重要的用途是法线映射(Normal Mapping)。其与视差映射(Parallax Mapping)、位移贴图(Displacement Mapping)等属于Bump Mapping范畴。相对于后两种，后两种方法可以提供更加真实逼真的表面凹凸感。

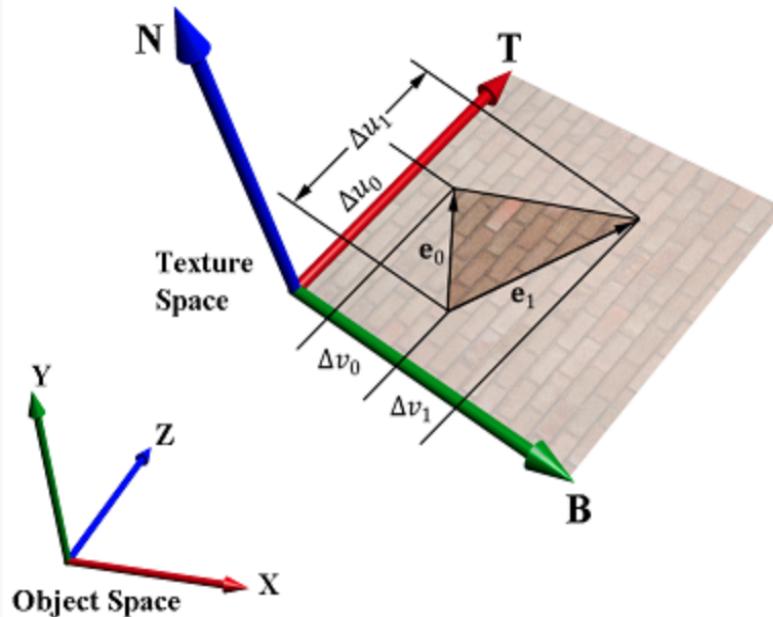
对于3D渲染来说，有模型空间、世界空间、相机空间、投影空间、标准设备空间、屏幕空间。对于每一种空间，3D渲染都会有对应的转换矩阵，将一个空间坐标系转换到另外一个空间坐标系。往微观上面来说，有切空间，微元空间，切空间是基于模型的表面来建立的一种坐标系，对于微元空间，是建立在原子级别的坐标系，但是对于原子级别的坐标系，我们并没有对应的矩阵来转换因为太小了，计算量巨大，即使有也得不偿失。所以针对于微元空间，3D渲染借助于一些数学以及物理学公式来建立模型算子来估算一些我们想要得到的计算结果。

但是对于表面空间，其计算量与最终得到的收益来衡量，其最终得到的收益是大于计算量的，所以3D渲染建立了一个坐标系，并构建了对应的坐标系系统。

切空间中，3D渲染规定，切空间由tangent(切线T)轴，bitangent(副法线B)轴，法线轴(N)组成的切空间坐标系。其中规定切线轴(T)与纹理坐标U增长方向一致，副法线轴与纹理坐标V增长方向一致，法线轴由法线贴图中读取得到。这就是切空间坐标系TBN，基于物体表面空间建立的一种坐标系。

2. 切线副法线的计算

2.1 在CPU侧计算



该图表示的是一个切空间坐标系，已知三角形三个顶点\$V_0\$、\$V_1\$、\$V_2\$，以及纹理坐标\$T_0(u_0, v_0)\$、\$T_1(u_1, v_1)\$、\$T_2(u_2, v_2)\$。

定义三角形的两条边为：

那么对应使用纹理坐标表示这个两条边差值：

那么我们如下关系式：

其中T、B表示切线与副法线，因为我们知道T与纹理坐标U增长方向一致，B与纹理坐标V增长方向一致。

我们将上纹理坐标与位置坐标的关系公式用矩阵表示：

将\$E_0\$、\$E_1\$，T、B拆分为分量模式：

对等式进行交换：

根据矩阵相关知识，我们知道 $A=\left[\begin{matrix} a & b \\ c & d \end{matrix}\right]$ 的逆矩阵为 A^{-1} ：

所以以上公式可以进一步表示为：

该公式是最终的计算公式，我们知道，其中，\$t_1\$、\$t_2\$、\$b_1\$、\$b_2\$与\$E_0\$、\$E_1\$都是已知变量，我们可以很轻松的计算出来TB的值。需要注意的是计算出来的向量的长度小于1，需要规范化一下才使用。

而我们计算出来的TB是基于单个三角形的，但是一般法线贴图是基于每个顶点进行。对于每个顶点所在的所有的三角形的对应的切空间值进行求平均，就可以得到该顶点的切空间的值。

对于切空间来说，我们知道TBN是相互垂直的，只要我们知道其中两个变量轴，就可以叉乘得到第三个轴，组装成TBN矩阵，进行后续的计算。

2.2 在着色器中计算

- 原理

在OpenGL ES 3.x的着色器中，增加了偏导数函数，偏导数函数（HLSL中的ddx和ddy，GLSL中的dFdx和dFdy）是片元着色器中的一个用于计算任何变量基于屏幕空间坐标的变化率的指令（函数）

其实在三角形栅格化期间，GPU并行执行时求像素块中的变量的差值计算的。数学计算模型为：

我们将之对应到CPU中的计算上来看，是否就是我们定义两条边的差值公式。那么按照上面的计算公式，我们就可以计算出TB向量了。

- 着色器中实现

```
in vec2 vTexCoords;
in vec3 vNormals;
in vec3 vPositions;
uniform sampler2D normalMap;
out vec4 fragColor;
void main()
{
    vec3 tangentNormal = texture(normalMap, vTexCoords).xyz * 2.0 - 1.0;
    vec3 E1 = dFdx(vPositions);
    vec3 E2 = dFdy(vPositions);
    vec3 t1b1 = dFdx(vTexCoords);
    vec3 t2b2 = dFdy(vTexCoords);
    vec3 T = normalize(E1 * t2b2.t, E2 * t1b1.t);
    vec3 B = -normalize(cross(N, T));
    vec3 N = normalize(vNormals);
    mat3 TBN = mat3(T, B, N);
    fragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

从切空间渲染法线贴图，还是在世界空间渲染法线贴图，看自己的选择。

3. 法线贴图的生成

- 生成法线贴步骤

- 转换为灰度图像

初始输入图像记做O,通过转换，将输入图像转换为灰度图像，因为对于生成法线贴图来说，色彩在生成的过程中时无用的。把图像作为一个顶点网格抽象出来，每个顶点代表图像上的一个像素点，每个顶点的z置等于灰度图像的值，而不是它的RGB值。

- 求图像的偏导数

图像从数学层面解释，其是一个2D函数，我们记做 $F(x,y)$ 表示，它将空间中的坐标与像素值建立关联，但是这个函数不是一个连续函数，因为它仅仅在空间中某些离散坐标点有定义。如果描述这个抽象出来的顶点网格都有法线并且是连续的。那么就需要对这个2D函数进行偏导数，以求取各个抽象出来的顶点值的关联或者是差值来作为描述顶点网格之间凹凸连续变化。其计算的数学公式为：

其中 F 函数，是我们抽象记做的 $F(x,y)$ 函数， $\frac{\partial F}{\partial x}$ 代表的是偏导数，也就是说，对图像灰度值进行偏导数。换句话说，就是求像素灰度值之间的变化率，因为函数 F 只能表示离散的像素之间的关系，并不能表示两个像素之间变化的值大小，偏导数却可以做到

- 归一化

图像存储只能存储到0到1之间的值，而我们计算的却是矢量，它的范围是-1到1之间，所以需要转换到0到1之间，转换公式为：

RG是红绿通道，而没有转换Z值，因为法线量总是面向z轴的正方向，所以不需要映射

4.13 视察贴图（三种）

4.14 三角剖分

4.15 抗锯齿处理

1. FXAA快速近似抗锯齿

1. 原理

2011年NVIDIA发布的一款基于后期处理的抗锯齿处理技术。主要是先对边缘像素的探测，之后区分RGB和明暗信息，将对比最为明显的像素取出，进行后处理，完成之后将纹理重新覆盖，从而实现抗锯齿效果。

其实现的原理类似于硬件多重采样抗锯齿操作

原理步骤为：

1. 首先将图像转变为亮度图像
2. 使用描边算子进行描边计算，得到边缘的值
3. 计算最大的亮度值与最小的亮度值，如果其亮度差值的计算大于规定阙值，则进行边缘模糊计算
4. 对x方向的亮度差值进行计算，对y方向的亮度差值进行计算，比对两个方向的亮度差值大小，以确定锯齿边缘的走向是垂直还是水平
5. 按照x或者y方向进行不同大小的步进进行计算亮度值，根据不同的亮度值判定选择不同的x或者y方向的模糊均值操作
6. 最后控制模糊的次数，分为四次模糊两次模糊，八次模糊等来控制边缘锯齿的控制

2. 着色器实现

代码：

```
const vec2 imageSize = vec2(400.,400.);  
const float FXAA_Reduce_Min = 1.0/128.0;  
const float FXAA_Reduce_Mul = 1.0/8.0;  
const float FXAA_Span_Max = 4.0;  
const vec3 cFXAAParams = vec3(1.0,0.5,0.75);  
vec4 FXAA_2(sampler2D tex2D,vec2 uv,vec2 imageSize,vec3 fxaaparams,float  
f_r_min,float f_r_mul,float f_s_max)  
{  
    vec2 posoffset = vec2(1.0/imageSize.x,1.0/imageSize.y) * fxaaparams.x;  
    vec3 rgbNW = texture(tex2D,uv + vec2(-posoffset.x,-posoffset.y)).xyz;  
    vec3 rgbNE = texture(tex2D,uv + vec2(posoffset.x,-posoffset.y)).xyz;  
    vec3 rgbSW = texture(tex2D,uv + vec2(-posoffset.x,posoffset.y)).xyz;  
    vec3 rgbSE = texture(tex2D,uv + vec2(posoffset.x,posoffset.y)).xyz;  
    vec3 rgbM = texture(tex2D,uv).xyz;  
  
    vec3 luma = vec3(0.299,0.587,0.114);  
    //S为南 N为北 E为东 W为西  
/*  
        NW(左上)           NE(右上)  
            center  
        SW(左下)           SE(右下)  
*/  
    float lumaNW = dot(rgbNW,luma); //左下角  
    float lumaNE = dot(rgbNE,luma); //  
    float lumaSW = dot(rgbSW,luma);  
    float lumaSE = dot(rgbSE,luma);  
    float lumaM = dot(rgbM, luma);  
  
    float lumaMin = min(lumaM,min(min(lumaNW,lumaNE),min(lumaSW,lumaSE)));  
    float lumaMax = max(lumaM,max(max(lumaNW,lumaNE),max(lumaSW,lumaSE)));  
    if(((lumaMax - lumaMin) / lumaMin) >= fxaaparams.y)
```

```

{
    vec2 dir;
    dir.x = -(lumaNW + lumaNE - (lumaSW + lumaSE));
    dir.y = ((lumaNW + lumaSW) - (lumaNE + lumaSE));

    float dirReduce = max((lumaNW + lumaNE+lumaSW+lumaSE)*(0.25 *
f_r_mul),f_r_min);

    float rcpDirmin = 1.0/(min(abs(dir.x),abs(dir.y) + dirReduce));
    dir = min(vec2(f_s_max,f_s_max),max(vec2(-f_s_max,-f_s_max),dir *
rcpDirmin)) * (1.0/imagesize);
    dir *= fxaarParams.z;
    vec3 rgbA = (1.0/2.0) * (texture(tex2D,uv + dir * (1.0/3.0 - 0.5)).xyz +
        texture(tex2D,uv + dir * (2.0/3.0 - 0.5)).xyz);
    vec3 rgbB =rgbA*(1.0/2.0)+(1.0/4.0)*(texture(tex2D,uv+dir * (0.0/3.0 -
0.5)).xyz
+texture(tex2D,uv + dir * (3.0/3.0 - 0.5)).xyz);
    float lumaB = dot(rgbB,luma);
    vec3 rgbOut;
    if(lumaB < lumaMin || lumaB >lumaMax)
        rgbOut = rgbA;
    else
        rgbOut = rgbB;
    return vec4(rgbout,1.0);
}else{
    return vec4(rgbM,1.0);
}
return vec4(0.0,0.0,0.0,1.0);
}

```

2.MSAA多重采样抗锯齿

1. 问题

光栅器是定点着色器通用性编程于顶元装配之后的所有的算法与过程的总和。光栅化会将一个图元的相关数据转化为一系列的片段值，其结果是一个二维的片段数组。其片段数组与图元的顶点数据是多对一的关系，而片段数据与屏幕上的像素是一对多的关系，这个关系并非是恒定不变的，其取决于硬件设备的像素分辨率大小。

理论上，如果光栅化所产生的片段大小是小于屏幕分辨率下的像素大小的，那么由小的片段值去组织一个三角形或者其它图元，那么肯定会有非常严整的边缘效果。就像我们在地面上画一个三角形，我们拿一些非常小的正方形去做填充，那么，正方形越小，其填充出来的边缘便会越平滑。

那现在这个屏幕像素，就是相当于我们理论上最小的正方形，而我们手里拿的正方形是大于这个理论上的最小的正方形的，那么，无论我们怎么去填充，其最终的边缘，肯定是会存在于锯齿的。

2. 多重采样抗锯齿

现在的问题是，在目前的硬件基础上，无法达到片段是比像素更小的情况下。我们就换另外一个思路去解决这个问题。

原来，一个像素点是有一个采样点的。手机硬件在判别三角形是否覆盖了当前像素点的时候，会去看三角形是否覆盖了这个像素的采样点位。单采样点，其规定，像素矩形的中心位置是采样点位。假如这个三角形没有覆盖这个像素采样点，那么当前像素的颜色是不会进入片段着色器中的计算当中的，这样就会带来锯齿。

我们无法解决上面所说的片段大小比像素大小更小的问题。那么就只能通过改变原来覆盖了一部分的像素的颜色值来达到人眼分辨不出来的效果。

多重采样其实是从这个理论上出发的，那么我们将在光栅器之后，给每个像素增加n个采样点。其采样运算还是经历一次，换句话说，每个像素还是只是经历一次片段计算过程，然后把三角形覆盖住的像素中一些采样点进行着色。假如一个像素有四个采样点，其中一个被三角形所覆盖，其它三个采样点未被覆盖。那将是一个有值采样点数据与其它三个无值采样点的和的平均值作为当前像素的颜色值。

从一定程度上来讲，这么做的结果是，内存是会变大的，因为多了很多采样点位来存储数据。

另一个方面，原来不参与片元着色器着色计算的像素，也会参与进来，也无疑是增加了计算量与内存的使用量

上面所说的问题，其实不单单存在与颜色的多个采样点，另外的深度以及模版测试也能够使用多个采样点。

还需要说明一点的是，多重采样缓冲的图像是不能直接用于其它的运算的。我们可以将多重采样缓冲区的数据用过OpenGL ES的glBlitFrameBuffer将它复制到另外一个普通的帧缓冲中，那么就将图像缩小或者还原了。

3. 总结

- 正常的像素是有一个居于像素矩形中心位置的采样点位的
- 多重采样抗锯齿，是在光栅化之后，一个像素矩形会拥有多个采样点，但是只会经历一次片元着色器的计算，其被覆盖住的采样点，会被片元着色器计算进行赋值
- 最终像素的颜色是多个采样点的均值颜色，片段着色器会处理这件事情
- 多重采样缓冲区的数据，是不能直接用于其他计算的，可以将它从多重采样帧缓冲中拷贝到另外一个帧缓冲中

4.16 空间管理算法

4.17 PBR渲染管线

说明：渲染方程、Cook-Torrance, BRDF, BSDF, BTDF, GGX分布函数、Fresnel、PBR材质

4.18 IBL

4.19 阴影贴图、阴影体

4.20 SSAO环境光遮蔽

1. SSAO概述

1.1. SSAO解决什么问题

对于光照，渲染将之分为环境光、漫反射光、镜面反射光三个光照分量，对于其中漫反射、镜面反射光，渲染使用了基于物理的光线模拟，以此来达到近似真实世界中的漫反射与镜面反射的效果。对于渲染中模拟光散射的环境光，是固定不变的光照常量。但是在真实世界中，光线会以任意方向进行散射，它的强度不是一成不变的，因此灯光渲染中即使使用了pbr技术，但是某些特殊的地方，比如褶皱、墙角以及孔洞等非常靠近墙面在现实中应该变暗的地方，但是渲染中并没有变暗，SSAO就是解决这种问题而产生的。

1.2. SSAO背后的原理

对于平铺四边形上的每一个片段，根据周边深度值计算出来一个遮蔽因子。这个遮蔽因子会被用来估算环境光照分量的大小，遮蔽因子越大，那么环境光照分量越小，反之环境光越大。而遮蔽因子的计算是通过采集片段周围半球核心的多个深度样本，并和当前片段深度值比对而得到的。高于片段深度值样本的个数就是我们想要的遮蔽因子。

2. SSAO技术实现

2.1. 概述技术实现步骤

从总体来看，SSAO的技术实现，需要几个非常关键的技术步骤，如下：

1. 延迟渲染空间渲染中的数据，输出法线量、位置数据、深度值以及纹理着色数据
2. 遮蔽因子(SSAO)计算
 1. 生成随机的法线半球的采样核心
 2. 生成随机向量，使随机核心转动
 3. SSAO着色器
3. 进行blur计算
4. 进行延迟光照计算，并把影响因子用到环境光照中

2.2 延迟渲染

将3D场景中的物体通过多重渲染输出，我们将3D场景中数据的位置、法线、深度以及纹理着色数据输出到对应的颜色缓冲里面，其中对于位置以及法线的颜色缓冲，我们采用RGB16F的参数，来确保数据的准确以及不被规格化。

2.3. 遮蔽因子计算

- 法线半球

沿着表面法线生成大量的样本以供当前位置的深度值比对。但对于每个表面法线方向生成采样核心，这是一件很困难而且很费力气的操作。那么在切空间中生成采样核心，这将是一件很轻松的事情。我们知道切空间其实就是3维物体的表面空间，非常契合我们要在表面做计算的一些事情。法线量将指向z轴正方向，假如z轴有负值，那么采样核心就会变成一个球，而不是半球了，这样就会有一半的采样核心在表面里面，就会造成孤岛危机中所有的东西都是灰蒙蒙的这种不真实的感觉。

另外，我们想让采样核心靠近真正的片段周围，那么我们采用一个加速插值函数：

```
float lerp(float a, float b, float f)
{
    return a+f*(b-a);
}
```

- 随机核心转动

单纯的一个随性性质的采样球，并不能很完美的，其采样的样本只能是一个在随机球里面的一个样本，这样并不能准确的估算出来当前真实片段的被遮蔽饿的情况。那么我们随机生成一个旋转的向量，来旋转这个随机采样半球，并多次采样样本进行比对，那么对于当前真实片段是否被遮蔽就会有一个很准确的估算。

- SSAO着色器

该步骤计算出来的环境光遮蔽因子，是下一个光照pass所需要的数据，那么对该次计算将输出到颜色缓冲中，以供下一次光照pass进行使用。

那么首先，我们需要将第一步骤的延迟渲染所输出的位置、法线、深度以及纹理着色器数据使用起来。

其次我们还要使用法线半球的法线来使用(记住这个随机法线是在切空间下定义随机出来的)，还有扰动法线的随机向量使用。

首先创建一个正交基，切线每一次都会根据我们随机出来的法线量值变动。根据TBN矩阵的正交性，我们可以求出来副法线，以此构造出来TBN矩阵用做后面将随机转动的向量转出切空间。

然后检查当前样本深度值，是否大于存储的深度值，如果大于那么添加到最终的遮蔽因子上。

```

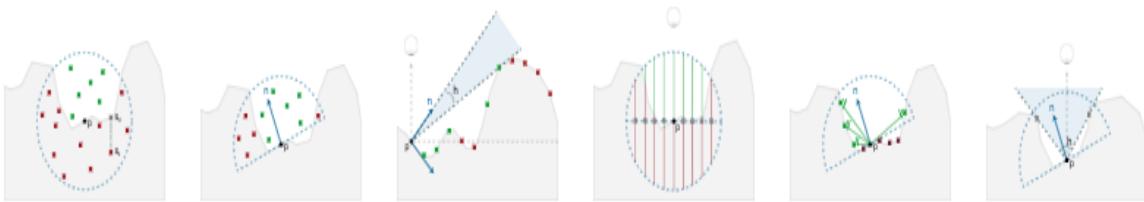
#version 300 es
precision mediump float;
uniform sampler2D gPosition;
uniform sampler2D gNormal;
uniform sampler2D texNoise;
uniform mat4 projection;
uniform vec3 samples[64];
int kernelsize = 64;
float radius = 0.4;
float bias = 0.025;
//屏幕的宽高
const vec2 noiseScale = vec2(1280.0/4.0, 720.0/4.0);
in vec2 vTexCoord;
out vec4 FragColor;
void main()
{
    //获取视图空间下的位置坐标, w为深度值
    vec3 fragPos = texture(gPosition, vTexCoord).xyz;
    //获取视图空间下的法线数据
    vec3 normal = normalize(texture(gNormal, vTexCoord).rgb);
    vec3 randomVec = normalize(texture(texNoise, vTexCoord *
noiseScale).xyz);
    // create TBN change-of-basis matrix: from tangent-space to view-space
    vec3 tangent = normalize(randomVec - normal * dot(randomVec, normal));
    vec3 bitangent = cross(normal, tangent);
    mat3 TBN = mat3(tangent, bitangent, normal);
    // iterate over the sample kernel and calculate occlusion factor
    float occlusion = 0.0;
    for(int i = 0; i < kernelsize; ++i)
    {
        // get sample position
        vec3 samplep = TBN * samples[i]; // from tangent to view-space
        samplep = fragPos + samplep * radius;

        // project sample position (to sample texture)
        vec4 offset = vec4(samplep, 1.0);
        offset = projection * offset; // from view to clip-space
        offset.xyz /= offset.w; // perspective divide
        offset.xyz = offset.xyz * 0.5 + 0.5; // transform to range 0.0 - 1.0
        // get sample depth //get depth value of kernel sample
        float sampleDepth = texture(gPosition, offset.xy).z;
        // range check & accumulate
        float rangeCheck = smoothstep(0.0, 1.0, radius / abs(fragPos.z -
sampleDepth));
        occlusion += (sampleDepth >= samplep.z + bias ? 1.0 : 0.0) *
rangeCheck;
    }
    occlusion = 1.0 - (occlusion / float(kernelsize));
    //occlusion
    FragColor = vec4(occlusion,occlusion,occlusion,1.0);
}

```

3. SSAO的技术流派

1.六种SSAO的技术图



从左到右依次为：

1. CryEngine 2 AO
2. 星际争霸二 AO
3. 基于Ray的机遇Horizon的AO
4. 体积模糊 AO
5. 炼金术 AO
6. 虚幻引擎4 AO

下面简单介绍这六种江湖上流传的SSAO技术流派中的优缺点：

借鉴于英文论文：<http://frederikaalund.com/a-comparative-study-of-screen-space-ambient-occlusion-methods/>

4.21 实时渲染中的软阴影技术

4.22 多viewport渲染问题

5. 图形后期渲染

5.1 Bloom曝光

5.2 DOF景深

5.2 HDR高光渲染

5.3 运动模糊

5.4 室外阴影渲染PSSM(平行分割阴影图)

5.5 体积云

5.6 体积光

6. 图像渲染

6.1 高斯模糊

1. 一维高斯模糊

1. 基础原理

正太分布的密度函数叫做高斯模糊，其一维形式是

其中 μ 是x的均值， σ 是x的方差。因为计算平均值的时候，中心点就是原点，所以 μ 等于0。其公式变为如下：

2. 着色器实现

```
//该算法是游戏引擎中通用的高斯模糊着色器。这是一维的高斯模糊模糊求解方式
// Adapted: http://callumhay.blogspot.com/2010/09/gaussian-blur-shader-gls1.html
vec4 GaussianBlur(int blurKernelsize, vec2 blurDir, vec2 blurRadius, float
sigma, sampler2D texSampler, vec2 texCoord)
{
    int blurKernelsizeHalfSize = blurKernelsize / 2;

    // Incremental Gaussian Coefficent Calculation (See GPU Gems 3 pp. 877 -
889)
    vec3 gaussCoeff;
    gaussCoeff.x = 1.0 / (sqrt(2.0 * PI) * sigma);
    gaussCoeff.y = exp(-0.5 / (sigma * sigma));
    gaussCoeff.z = gaussCoeff.y * gaussCoeff.y;

    vec2 blurVec = blurRadius * blurDir;
    vec4 avgValue = vec4(0.0);
    float gaussCoeffSum = 0.0;

    avgValue += texture2D(texSampler, texCoord) * gaussCoeff.x;
    gaussCoeffSum += gaussCoeff.x;
    gaussCoeff.xy *= gaussCoeff.yz;

    for (int i = 1; i <= blurKernelsizeHalfSize; i++)
    {
        avgValue += texture2D(texSampler, texCoord - float(i) * blurVec) *
gaussCoeff.x;
        avgValue += texture2D(texSampler, texCoord + float(i) * blurVec) *
gaussCoeff.x;

        gaussCoeffSum += 2.0 * gaussCoeff.x;
        gaussCoeff.xy *= gaussCoeff.yz;
    }

    return avgValue / gaussCoeffSum;
}
```

2. 二维高斯模糊

1. 基础原理

那么我们可以根据一维的高斯函数，可以推导出来二维的高斯函数为：

根据高斯函数，我们可以计算出来高斯模糊中，我们所需要的半径内的各个像素对于中心点的像素权重值。

2. 着色器实现

```
//这是高斯模糊的二维求解模式，可以动态求解3*3、5*5、7*7、9*9的半径的模糊
const float PI = 3.14159265f;
vec4 GaussianBlur(int radius, float sigma, sampler2D texSampler, vec2
texCoord, vec2 imageSizeStep)
{
```

```

float sigmasquare = sigma * sigma;
float gaussCoeff1 = 0.5/(PI * sigmasquare);
float gaussCoeff2 = 0.0;
float gaussCoeffSum = 0.0;
vec4 avgValue = vec4(0.0);
int halfRadius = radius/2;
for(int i = 0;i < radius; i++)
{
    for(int j = 0; j < radius; j++)
    {
        float stepX = float(-halfRadius + i);
        float stepY = float(halfRadius - j);
        vec2 stepCoord = texCoord + imageSizeStep * vec2(stepX,stepY);
        gaussCoeff2 = gaussCoeff1 * exp(-(stepX * stepX + stepY * stepY) *
0.5/sigmasquare);
        avgValue += texture(texSampler,stepCoord) * gaussCoeff2;
        gaussCoeffSum += gaussCoeff2;
    }
}
vec4 result = clamp(avgValue/gaussCoeffSum,0.0,1.0);
return result;
}

```

6.2 双线性插值

1. 基础原理

双边滤波的核函数是空间域核与像素域核的综合结果：在图像的平坦区域，像素变化很小，对应的像素范围权重接近于1，此时空间域权起重要作用，相当于进行了高斯模糊，在图像的边缘区域，像素值变化很大，像素范围域权重变大，从而保持了边缘的信息。

其中 σ_d 与 σ_r 表示的是平滑的参数，一个是空域参数，一个是值域参数。其中 $I(i,j)$ 与 $I(k,l)$ 表示的在坐标 (i,j) 与 (k,l) 坐标处像素的值，然后计算的权重公式为：

其中 I_D 是降低在坐标 (i, j) 处的噪声。

2. 着色器实现

```

vec3 BilateralFilter(sampler2D cmap,vec2 uv,float r,float ss,float sc,vec2
images)
{
    int d = int(r);
    float ss2 = 2.0 * ss * ss;
    float sc2 = 2.0 * sc * sc;
    float i = uv.x;
    float j = uv.y;
    float weightSum = 0.0;
    vec3 filtervalue = vec3(0.0);
    vec3 centerCol = texture(cmap,uv).rgb;

    for(int k = -d; k <= d;k++)
    {
        for(int l = -d; l <= d; l++)
        {
            vec2 curUV = uv + vec2(k,l) * vec2(1.0/images.x,1.0/images.y);
            vec3 curCol = texture(cmap,curUV).rgb;
            //值域距离计算
            float value_Square = dot(curCol - centerCol,curCol - centerCol);
        }
    }
}

```

```

    //空间域距离计算
    float distance_Square = distance(curUV, uv) * float(d);
    float weight = exp(-1.0 * (distance_Square/ss2 + value_Square/sc2));
    weightSum += weight;
    filterValue += (weight * curCol);
}
}
filterValue = filterValue/weightSum;
return filterValue;
}

```

6.3 高\低通滤波器(中值滤波、巴特沃斯滤波、高斯低通滤波)

1. 理想高\低通滤波器

1. 原理

- 高通滤波器：让高频信息通过，过滤低频信息；低通滤波相反
- 低通滤波器模版

其中\$D_0\$表示通带半径，\$D(u,v)\$是到频谱中心的距离(欧式距离)，计算公式如下：

M和N表示频谱图像的大小，(M/2,N/2)即为频谱中心

- 高通滤波器

理想的高通滤波器与此相反，1减去低通滤波器模版即可

2. 着色器实现

```

vec4 LowAndHighPass(bool flag, float d0, int radius, float
distanceNormalizaFactor, sampler2D texSampler, vec2 texCoord, vec2 step)
{
    int halfRadius = radius/2;
    vec4 centerColor = texture(texSampler, texCoord);
    vec4 colorValue;
    float radiusValue;
    for(int i = 0; i < radius; i++)
    {
        for(int j = 0; j < radius; j++)
        {
            float stepX = float(-halfRadius + i);
            float stepY = float(halfRadius - j);
            vec2 stepCoord = texCoord + step * vec2(stepX,stepY);
            vec4 tempColor = texture(texSampler,stepCoord);
            float disFromCentralColor = distance(centerColor,tempColor)
            * distanceNormalizaFactor;
            if(disFromCentralColor <= d0)
            {
                disFromCentralColor = 1.0;
            }else{
                disFromCentralColor = 0.0;
            }
            if(!flag)/0 是低通滤波 1 是高通滤波
            {
                disFromCentralColor = 1.0 - disFromCentralColor;
            }
            colorValue += tempColor * disFromCentralColor;
            radiusValue += disFromCentralColor;
        }
    }
}

```

```

        }
    }
    return colorValue/radiusValue;
}

```

2. 中值滤波

1. 原理

- 前言

信号处理时经常要做的一件事就是滤波，其中线性滤波器比如FIR、IIR等类型都是研究的比较透彻的，实际使用中也有很好的效果。但是有时我们遇到的信号的噪声比较顽固，比如说电子信号中的爆米花噪声患有图像处理中的椒盐噪声，用普通的线形滤波只能将其压低，而无法彻底消除，这时一些非线性滤波器就表现出来优势了。中值滤波在图像领域中用的比较多，其实这种滤波器也可以用于一维信号，有时甚至能起到意想不到的效果。

- 基本原理

假如一个信号是平缓变化的，那么某一点的输出值可以用这点的某个大小的邻域内的所有值的统计中值来代替。这个邻域在信号处理邻域称之为窗。窗开的越大，输出的结果就越平滑，但也可能会把我们有用的信号特征给抹掉。所以窗的大小要根据实际的信号和噪声特性来确定。通常我们会选择窗的大小使得窗内的数据个数为奇数个，这样才会有唯一的中间值。

- 边界值的处理

- 不处理边界的数据
- 将边界值重复
- 前后补0
- 在边界时缩小窗的大小

2. 着色器实现

```

vec4 median(sampler2D texSampler, vec2 xy, vec2 stepSize)
{
    vec4 c = texture(texSampler, xy);
    vec4 median;
    float x1 = stepSize.x;
    float y1 = stepSize.y;
    float buff = xy.x +1.;
    xy.x-= x1;
    xy.y-= y1;
    float array[9];
    float array1[9];
    float array2[9];
    float r = 0.;
    float g = 0.;
    float b = 0.;
    for (int i =0; i < 9; i++)
    {
        c = texture(texSampler, xy);
        array[i] = c.r;
        r = r + c.r;
        array1[i] = c.g;
        g = g + c.g;
        array2[i] = c.b;
        b = b + c.b;
        if (xy.x - buff == 0.)
        {

```

```

        xy.x-= 2.*x1;
        xy.y+= y1;
    }
    else
        xy.x+=x1;
}
for (int k = 0; k < 9; k++)
{
    float sum = array[k];
    for (int h = k+1; h < 9; h++)
    {
        float sum1 = array[h];
        if (sum > sum1)
        {
            float r = array[k];
            array[k] = array[h];
            array[h] = r;
        }
    }
}
for (int q = 0; q < 9; q++)
{
    float sum = array1[q];
    for (int p = q+1; p < 9; p++)
    {
        float sum1 = array1[p];
        if (sum > sum1)
        {
            float r = array1[q];
            array1[q] = array1[p];
            array1[p] = r;
        }
    }
}
for (int k = 0; k < 9; k++)
{
    float sum = array2[k];
    for (int h = k+1; h < 9; h++)
    {
        float sum1 = array2[h];
        if (sum > sum1)
        {
            float r = array2[k];
            array2[k] = array2[h];
            array2[h] = r;
        }
    }
}
median;
median.r = array[4];
median.g = array1[4];
median.b = array2[4];
return median;
}

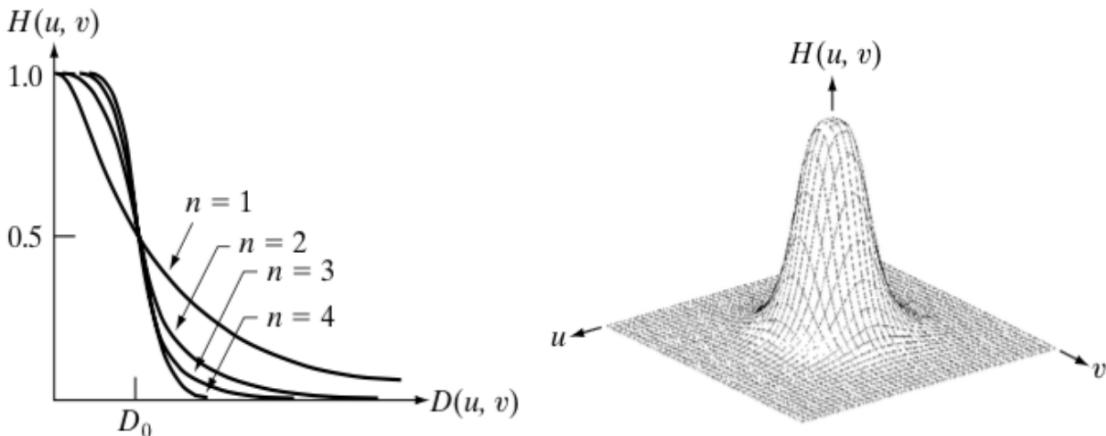
```

3. 巴特沃斯(Butterworth)滤波

1. 原理

- 滤波函数

如果从函数图的走向来看，用幂数n可以改变滤波器的形状，n越大，则该滤波器越接近于理想滤波器



1减去低通滤波器模版即可得到高通滤波器模版

2. 着色器实现

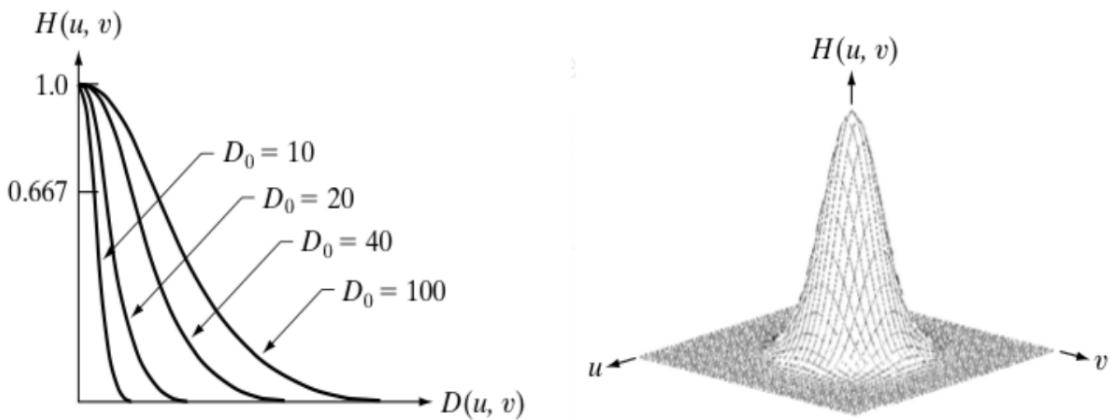
```
vec4 ButterWorth(bool flag, int radius, float D0, float n, float
distanceNormalizaFactor, sampler2D texSampler, vec2 texCoord, vec2 step)
{
    int halfRadius = radius/2;
    vec4 centerColor = texture(texSampler, texCoord);
    vec4 colorValue;
    float radiusValue;
    for(int i = 0; i < radius; i++)
    {
        for(int j = 0; j < radius; j++)
        {
            float stepX = float(-halfRadius + i);
            float stepY = float(halfRadius - j);
            vec2 stepCoord = texCoord + step * vec2(stepX, stepY);
            vec4 tempColor = texture(texSampler, stepCoord);
            float disFromCentralColor = distance(centerColor, tempColor)
                * distanceNormalizaFactor;
            disFromCentralColor = min(1.0 / (1.0 + pow((disFromCentralColor/D0),
                2.0 * n)), 1.0);
            if(!flag) // 0 是低通滤波 1 是高通滤波
            {
                disFromCentralColor = 1.0 - disFromCentralColor; // 高通滤波
            }
            colorValue += tempColor * disFromCentralColor;
            radiusValue += disFromCentralColor;
        }
    }
    return colorValue/radiusValue;
}
```

4. 高斯低通滤波

1. 原理

- 高斯低通滤波器函数为：

函数走向：



- 高斯高通滤波

1减去低通滤波模版即可得到高通滤波模版

2. 着色器实现

```

vec4 Gauss(bool flag, int radius, float D0, float n, float
distanceNormalizaFactor, sampler2D texSampler, vec2 texCoord, vec2 step)
{
    int halfRadius = radius/2;
    vec4 centerColor = texture(texSampler, texCoord);
    vec4 colorvalue;
    float radiusvalue;
    for(int i = 0; i < radius; i++)
    {
        for(int j = 0; j < radius; j++)
        {
            float stepX = float(-halfRadius + i);
            float stepY = float(halfRadius - j);
            vec2 stepCoord = texCoord + step * vec2(stepX,stepY);
            vec4 tempColor = texture(texSampler,stepCoord);
            float disFromCentralcolor = distance(centerColor,tempColor)
                * distanceNormalizaFactor;
            disFromCentralcolor = min(exp(-disFromCentralcolor*
                disFromCentralcolor/(2*D0*D0)),1.0);
            if(!flag)//0 是低通滤波 1 是高通滤波
            {
                disFromCentralcolor = 1.0 - disFromCentralcolor;//高通滤波
            }
            colorvalue += tempColor * disFromCentralcolor;
            radiusValue += disFromCentralcolor;
        }
    }
    return colorvalue/radiusValue;
}

```

6.4 饱和度、对比度、曝光、色阶调整

1. 饱和度

1. 原理

- 饱和度即色彩的纯度，R、G、B三个分量，如果它们之间的值相差很大，如RGB(255,0,0),则画面是很鲜明的红色，如果它们的值趋于相同，如(100,100,100)，那图像看起来是比较灰的，只有亮度信息而没有色彩信息。

- 计算步骤
 - 提取图像亮度
 - 构造灰度校准向量
 - 调节饱和度

$\$inputColor\{rgb\}$ 代表的是RGB三个分量的差异程度，而 $greyScaleColor$ 代表RGB三个分量的趋同程度， $saturation$ 调节它们所占的比重，当 $saturation < 1$ 时， $greyScaleColor$ 灰度信息比重变大，图像饱和度减少，当 $saturation > 1$ 时， $\$inputColor\{rgb\}$ 比重变大，RGB的差异程度变大，图像色彩更加鲜明

2. 着色器实现

```
vec4 saturationColor(sampler2D sTexture, vec2 uv)
{
    vec4 originColor = texture(sTexture, uv);
    //求原图像亮度
    vec3 luminanceWeighting = vec3(0.2125, 0.7154, 0.0721);
    float luminance = dot(originColor.rgb, luminanceWeighting);
    //构造灰度校准向量
    vec3 greyScaleColor = vec3(luminance);
    float saturation = 1.2;
    vec3 saturationColor = mix(greyScaleColor, originColor.rgb, saturation);
    return vec4(saturationColor, originColor.a);
}
```

2. 对比度

1. 原理

- 对比度公式

由对比度公式，对比度调节会改变输入颜色和颜色中间值0.5的距离。当 $contrast=1$ (默认值)， $\$outColor\{rgb\} = \$inputColor\{rgb\}$ ，也就是原图，当 $contrast > 1$ 时，输出的颜色会远离中间值0.5，向两边靠拢，到图像的效果就是“亮的更亮，暗的更暗”，反之当 $contrast < 1$ 时，输出值会靠近中间值0.5，图像上所有的像素的亮度都会趋于相同

2. 着色器实现

```
vec4 contrastColor(sampler2D sTexture, vec2 uv)
{
    vec4 originColor = texture(sTexture, uv);
    float contrast = 1.9;
    vec4 contrastColor = vec4((originColor.rgb - vec3(0.5) *
                                contrast +
                                vec3(0.5)), originColor.w);
    return contrastColor;
}
```

3. 曝光

1. 原理

- 计算公式

2. 着色器实现

```

vec4 exposureColor(sampler2D sTexture, vec2 uv)
{
    vec4 originColor = texture(sTexture, uv);
    float exposure = -0.5;
    vec4 exposureColor = vec4(originColor.rgb *
pow(2.0, exposure), originColor.w);
}

```

4. 图像增强--USM锐化

1. 原理

USM锐化是用来锐化图像边缘的，它是通过调整图像边缘细节的对比度，并在边缘的两侧生成一条亮线一条暗线，使画面整体更加清晰

USM锐化公式描述是很麻烦的，这里说一下实现的步骤依次是什么

- 备份图像原数据
- 按照给定半径进行高斯模糊
- 用原图像与高斯模糊后的像素相减，形成一个差值
- 将差值乘以数量
- 将差值部分分为正负两部分，取绝对值
- 正负差值分别减去给定的阈值
- 原像素加上正负差值减去负差值，锐化完毕

2. 着色器实现

代码：

```

const vec2 imageSize = vec2(300.0, 300.0);
const float PI = 3.14159265f;
const float numCount = 2.0; //边的数量
const float threshold = 0.03;
//这是高斯模糊的二维求解模式，可以动态求解3*3、5*5、7*7、9*9的半径的模糊
vec4 GaussianBlur(int radius, float sigma, sampler2D texSampler, vec2
texCoord, vec2 imageSizeStep)
{
    float sigmaSquare = sigma * sigma;
    float gaussCoeff1 = 0.5 / (PI * sigmaSquare);
    float gaussCoeff2 = 0.0;
    float gaussCoeffSum = 0.0;
    vec4 avgValue = vec4(0.0);
    int halfRadius = radius / 2;
    for(int i = 0; i < radius; i++)
    {
        for(int j = 0; j < radius; j++)
        {
            float stepX = float(-halfRadius + i);
            float stepY = float(halfRadius - j);
            vec2 stepCoord = texCoord + imageSizeStep * vec2(stepX, stepY);
            gaussCoeff2 = gaussCoeff1 * exp(-
(stepX*stepX+stepY*stepY)*0.5/sigmaSquare);
            avgValue += texture(texSampler, stepCoord) * gaussCoeff2;
            gaussCoeffSum += gaussCoeff2;
        }
    }
    vec4 result = clamp(avgValue / gaussCoeffSum, 0.0, 1.0);
}

```

```

        return result;
    }

//图像增强的方法，该方法是USM方法
vec4 USMSharpen(sampler2D tex2D, vec2 uv)
{
    vec4 finalColor;
    vec4 posColor;//正数值
    vec4 negColor;//负数绝对值
    vec4 gColor = GaussianBlur(7, 3.0, tex2D, uv, 1.0 / imageSize);
    vec4 originColor = texture(tex2D, uv);
    vec4 diffColor = originColor - gColor;
    if(diffColor.r < 0.0)
    {
        negColor.r = abs(diffColor.r * numCount) - threshold;
    }else{
        posColor.r = diffColor.r * numCount - threshold;
    }
    if(diffColor.g < 0.0)
    {
        negColor.g = abs(diffColor.g * numCount) - threshold;
    }else{
        posColor.g = diffColor.g * numCount - threshold;
    }
    if(diffColor.b < 0.0)
    {
        negColor.b = abs(diffColor.b * numCount) - threshold;
    }else{
        posColor.b = diffColor.b * numCount - threshold;
    }
    finalColor = vec4(originColor.rgb + posColor.rgb -
negColor.rgb, originColor.a);
    return finalColor;
}

```

6.5 图像描边（索贝尔、拉普拉斯、罗伯特、普洱瑞斯）

1. 图像描边的本质

1.1 定义

根据微积分的定义在二维离散函数中推导出来的。核心目的是得到像素点与其相邻像素的灰度值变化情况，并通过这种变化来增强图像。原始定义的梯度知识灰度值变化的度量工具。

1.2 像素与其八领域矩阵表示

x方向:\$Z_{-1}\$---->\$Z_7\$

y方向： \$Z_{-1}\$-->\$Z_3\$

根据图像梯度定义：

$G_x = Z_8 - Z_5$

$G_y = Z_6 - Z_5$

显然我们还可以使用另外的变化来度量灰度的变化，比如

$Z_9 - Z_5$ 、 $Z_1 - Z_5$ 等

梯度本质作用，是找到某像素与其相邻像素的灰度差值，并放大这种差值，利用这种差值来达到图像增强的效果，所以梯度计算方法可以有多种的，最后的结果是要达到度量梯度变化就可以了

- 比如罗伯特交叉梯度算子

$$G_x = Z_9 - Z_5$$

$$G_y = Z_8 - Z_6$$

- 拓展到8领域

$$G_x = (Z_7 + Z_8 + Z_9) - (Z_1 + Z_2 + Z_3)$$

$$G_y = (Z_3 + Z_6 + Z_9) - (Z_1 + Z_4 + Z_7)$$

增加距离距离的权重，那么就是索贝尔算子（与中心点Z5更近的点Z3, Z4, Z6, Z8的权重为2，其它对角线上的权重为1）

2. 索贝尔算子

2.1 定义与作用：用来对图像进行边缘检测

本质上是一离散性差分算子，用来运算图像亮度函数的灰度之近似值。在图像的任何一点使用此算子，将会产生对应的灰度矢量或是其法矢量。

- Sobel卷积因子为：

G_x 也就是x方向的因子为

-1	0	+1
-2	0	+2
-1	0	+1

G_y 也就是y方向的因子为

+1	+2	+1
0	0	0
-1	-2	-1

该算子包含两组3*3的矩阵，分别为横向以及纵向，将之与图像做平面卷积，即可分别得出横向以及纵向的亮度差分近似值。如果以A代表原始图像， G_x 及 G_y 分别代表和行以及纵向边缘检测的图像灰度值，其公式如下

其中 $f(a,b)$ ，表示图像 (a,b) 点的灰度值

图像的每一个像素的横向及纵向灰度值通过一下公式结合，来计算该点灰度值的大小

通常，为了提高效率，使用不开平方的近似值

如果梯度 G 大于某一阈值，则认为该点 (x, y) 为边缘点。

Sobel算子更具像素点上下左右临近灰度加权差，在边缘处达到极值这一现象检测边缘。对噪声具有平滑作用，提供较为准确的边缘方向信息，边缘定位精度不够高，**当对精度要求不是很高时**，是一种较为常用的边缘检测方法。

如果要求整体x和y方向的图像梯度，只需要将 G_x 与 G_y 相加即可。

2.2 着色器实现

```
vec4 sobelSharpen(sampler2D texSamp, vec2 uv, vec2 step)
```

```

{
    vec4 sum;
    vec4 s0 = texture(iChannel1, vec2(uv.x - step.x, uv.y + step.y));
    vec4 s1 = texture(iChannel1, vec2(uv.x, uv.y + step.y));
    vec4 s2 = texture(iChannel1, vec2(uv.x + step.x, uv.y + step.y));
    vec4 s3 = texture(iChannel1, vec2(uv.x - step.x, uv.y));
    vec4 s4 = texture(iChannel1, vec2(uv.x, uv.y));
    vec4 s5 = texture(iChannel1, vec2(uv.x + step.x, uv.y));
    vec4 s6 = texture(iChannel1, vec2(uv.x - step.x, uv.y - step.y));
    vec4 s7 = texture(iChannel1, vec2(uv.x, uv.y - step.y));
    vec4 s8 = texture(iChannel1, vec2(uv.x + step.x, uv.y - step.y));
    /* GX          GY
     * -1 0 +1      +1 +2 +1
     * -2 0 +2      0 0 0
     * -1 0 +1      -1 -2 -3
    */
    vec4 gx = s2 * 1.0 + s5 * 2.0 + s8 * 1.0 - (s0 * 1.0 + s3 * 2.0 + s6 * 1.0)
    + s4;
    vec4 gy = s0 * 1.0 + s1 * 2.0 + s2 * 1.0 - (s6 * 1.0 + s7 * 2.0 + s8 * 1.0)
    + s4;
    sum = gx ;
    return sum;
}

```

3. 罗伯特算子

3.1 罗伯特算子特点

Robert算子被应用到图像增强总的锐化，其作为一阶微分算子Robert计算简单，对细节的反应敏感
其边缘检测的作用是提供边缘候选点，可以提供相对较细的边缘

- Gx 与Gy的表示

-1	0	0	-1
0	1	1	0

3.2 着色器实现

```

vec4 robertSharpen(sampler2D texSamp, vec2 uv, vec2 step)
{
    vec4 sum;
    vec4 s0 = texture(texSamp, vec2(uv.x,uv.y));
    vec4 s1 = texture(texSamp, vec2(uv.x + step.x,uv.y));
    vec4 s2 = texture(texSamp, vec2(uv.x,uv.y - step.y));
    vec4 s3 = texture(texSamp, vec2(uv.x + step.x,uv.y - step.y));
    vec4 gx = 2.0 * s0 - 1.0 * s3;
    vec4 gy = 1.0 * s1 - 1.0 * s2;
    sum = gx + gy;
    return sum;
}

```

4. 拉普拉斯算子

1. 导数求导公式

2. 微分公式

3. 拉普拉斯边缘检测

- 概念

拉普拉斯算子是最简单的各同性微分算子，具有旋转不变性。二维图像的拉普拉斯变换是各向同性的二阶导数，定义为：

我们都知道，求导是求的切线的斜率，假如定义坐标值(x,y)到坐标值(\$x_0\$,\$y_0\$)的灰度图像中的灰度值为函数f(x,y)的值，那么函数f(x,y)就可以表示一副二维图像，也就是 $y = f(x, y)$ 函数。

那么在这二维图像任何一个坐标我都可以求解到图像的灰度值大小 $f(x, y)$ 。如果对其进行一阶求导，那么我就可以得到 $f(x, y)$ 函数的跃升的位置。那么我们做二次导数之时，一阶导数出现跃升的地方则会出现零点现象，并会出现从正到负的。那么其零点就是作为边缘的判断点，这就是拉普拉斯算子的数学推论。

现实中为了更好的进行数字图像处理，该方程表示为离散形式：

从前面我们假设的 $f(x, y)$ 代表的二维图像的灰度值表示，那么该公式我们可以拓展为：

$0 * f(x-1, y+1)$	$1 * f(x, y+1)$	$0 * f(x+1, y+1)$
$1 * f(x-1, y)$	$-4 * f(x, y)$	$1 * f(x+1, y)$
$0 * f(x-1, y-1)$	$1 * f(x, y-1)$	$0 * f(x+1, y-1)$

其还有另外一种扩展模版：

$1 * f(x-1, y+1)$	$1 * f(x, y+1)$	$1 * f(x+1, y+1)$
$1 * f(x-1, y)$	$-8 * f(x, y)$	$1 * f(x+1, y)$
$1 * f(x-1, y-1)$	$1 * f(x, y-1)$	$1 * f(x+1, y-1)$

拉普拉斯算子可以增强图像边缘变化剧烈的位置，平缓减弱变换缓慢的变化区域。因此可以使用拉普拉斯算子对原图像灰度图像进行处理，然后再与原图像叠加，，叠加公式由下式表示：

4. 着色器实现

```
vec4 laplacianSharpen(sampler2D texSamp, vec2 uv, vec2 step)
{
    vec4 sum;
    vec4 s4 = texture(texSamp, uv);
    vec4 s0 = texture(texSamp, vec2(uv.x - step.x, uv.y + step.y));
    vec4 s1 = texture(texSamp, vec2(uv.x, uv.y + step.y));
    vec4 s2 = texture(texSamp, vec2(uv.x + step.x, uv.y + step.y));
    vec4 s3 = texture(texSamp, vec2(uv.x - step.x, uv.y));
    vec4 s5 = texture(texSamp, vec2(uv.x + step.x, uv.y));
    vec4 s6 = texture(texSamp, vec2(uv.x - step.x, uv.y - step.y));
    vec4 s7 = texture(texSamp, vec2(uv.x, uv.y + step.y));
    vec4 s8 = texture(texSamp, vec2(uv.x + step.x, uv.y - step.y));
    vec4 Gx = 0. * s0 - 1. * s1 + 0. * s2 - 1. * s3 + 5. * s4 - 1. * s5 + 0. * s6 - 1. * s7 + 0. * s8;
    vec4 Gy = -1. * s0 - 1. * s1 - 1. * s2 - 1. * s3 + 9. * s4 - 1. * s5 - 1. * s6 - 1. * s7 - 1. * s8;
    sum = Gx;
    // sum = Gy;
    // sum = Gx + Gy;
    return sum;
}
```

5. prewitt(普洱瑞斯算子)

1. 原理

- prewitt算子是一种一阶微分算子，利用像素点上下左右邻点灰度差，在边缘处达到极值检测边缘，对噪声具有平滑的作用。

其原理是在图像空间利用两个方向模板与图像进行邻域卷积来完成，这两个方向模板一个检测水平边缘，一个检测垂直边缘

对比roberts算子，prewitt算子对噪声有抑制作用，抑制噪声的原理是通过像素平均，因此噪声较多的图像处理比较好，但是像素平均相当于对图像的低通滤波，所以prewitt算子对边缘的定位却不如roberts算子

2. 卷积模版

- 水平方向

-1	0	1
-1	0	1
-1	0	1

- 垂直方向

1	1	1
0	0	0
-1	-1	-1

3. 着色器实现

```
vec4 prewittSharpen(sampler2D texSamp, vec2 uv, vec2 step)
{
    vec4 sum;
    vec4 s4 = texture(texSamp, uv);
    vec4 s0 = (texture(texSamp, vec2(uv.x - step.x, uv.y + step.y)));
    vec4 s1 = (texture(texSamp, vec2(uv.x, uv.y + step.y)));
    vec4 s2 = (texture(texSamp, vec2(uv.x + step.x, uv.y + step.y)));
    vec4 s3 = (texture(texSamp, vec2(uv.x - step.x, uv.y)));
    vec4 s5 = (texture(texSamp, vec2(uv.x + step.x, uv.y)));
    vec4 s6 = (texture(texSamp, vec2(uv.x - step.x, uv.y - step.y)));
    vec4 s7 = (texture(texSamp, vec2(uv.x, uv.y + step.y)));
    vec4 s8 = (texture(texSamp, vec2(uv.x + step.x, uv.y - step.y)));
    /* prewitt 普瑞维特 算子
        GX          GY
        -1 0 +1      +1 +1 +1
        -1 0 +1      0  0  0
        -1 0 +1      -1 -1 -1
        G(x,y) = max(|GX|, |GY|);
    */
    vec4 gx = -(s0 + s3 + s6) + (s2 + s5 + s8) + 1.0 * s4;
    vec4 gy = (s0 + s1 + s2) - (s6 + s7 + s8) + 1.0 * s4;
    sum = gx;
    return sum;
}
```

6.6 非真实感着色

1. 油画相关算法

1. 图像油画处理

- 原理

西方绘画之一的油画，主要的特点是色彩丰富含蓄，不过多追求细节表现，显示比较明显的边缘效果。在图像处理领域，是通过像素周边的权重滤波来模拟油画的特点。其算法的主要步骤为三步骤：

- 图像灰度化

首先将图像进行灰度化操作

- 像素散列

按照0-255灰度像素值，来划分n个区间，判断r半径中，像素值处于灰度区间最多的区间，并将这些真实的像素值（RGB）累加

- 将累加的这些像素值除以像素的个数，最终得到油画的颜色值

- 着色器实现

```
//radius 是半径
//texCoord 是uv采样坐标
//imageSizeStep 偏移的单位步进
//texSampler 纹理
vec4 oilcolor(int radius, vec2 texCoord, vec2 imageSizeStep, sampler2D texSampler)
{
    int halfRadius = radius/2;
    int raSq = radius * radius;
    int iCoCount = 10;
    vec4 numColor[iCoCount];
    int numIndex[iCoCount];
    //计算出最大的灰度区间值是那个
    for(int i = 0; i < radius; i++)
    {
        for(int j = 0; j < radius; j++)
        {
            float stepX = float(-halfRadius + i);
            float stepY = float(halfRadius - j);
            vec2 stepCoord = texCoord + imageSizeStep * vec2(stepX, stepY);
            vec4 origColor = texture(texSampler, stepCoord);
            int index = int(luminance(origColor) * float(iCoCount));
            numIndex[index] += 1;
            numColor[index] += origColor;
        }
    }
    //统计计算
    int maxCount = -1;
    int maxIndex = 0;
    for(int k = 0; k < iCoCount; k++)
    {
        if(numIndex[k] > maxCount){
            maxCount = numIndex[k];
            maxIndex = k;
        }
    }
    vec4 finalcolor = numColor[maxIndex]/float(maxCount);
```

```
    return vec4(finalcolor.rgb,1.0);
}//效果比较好的是，半径radius=9
```

2. Kuwahara滤波进行油画处理

- 原理

Kuwahara滤波是一种非线性的平滑滤波，它可以在平滑图像的同时保留图片边缘原有的性质。Kuwahara滤波是使用四个卷积来分别统计像素的左上、左下、右上和右下四个方向的像素。统计半径为radius，每次统计的像素数量为 $(radius + 1) * (radius + 1)$ ，统计四次，也就是四个方向。在每个卷积核的统计过程中，记录每个块所对应的均值与方差，通过这两个参数来衡量卷积核内颜色值的总体波动情况。对于有着相似颜色的卷积核，对应的方差波动较小，对于颜色相差较大的卷积核，方差差距较大。由于图片的边缘颜色值差异都比较大，所以方差波动较大，因此我们在处理边缘像素点的时候会倾向于选择图片内侧或外侧的卷积颜色值，以此来保留边缘的效果。

统计完成够，我们要比较四对数组，均值数组与方差数组的波动情况，记录总体波动最小的颜色值作为像素点的输出。公式如下：

该算法的遍历次数是 $4 * (radius + 1)^2$ 次，是比上面的算法计算量多一些的。

- 着色器实现

```
vec4 kuwaharaC(int radius, vec2 texCoord, vec2 stepSize, sampler2D textureSamp)
{
    vec2 offset[4];
    offset[0] = vec2(-radius, -radius);
    offset[1] = vec2(-radius, 0);
    offset[2] = vec2(0, -radius);
    offset[3] = vec2(0, 0);
    vec3 m[4]; //均值
    vec3 s[4]; //方差
    for (int k = 0; k < 4; ++k)
    {
        m[k] = vec3(0.0);
        s[k] = vec3(0.0);
    }
    float n = float((radius + 1) * (radius + 1));
    for (int k = 0; k < 4; k++)
    {
        for (int i = 0; i <= radius; i++)
        {
            for (int j = 0; j <= radius; j++)
            {
                vec2 nT = vec2(i, j) + offset[k];
                vec3 c = texture(textureSamp, texCoord + nT * stepSize).rgb;
                m[k] += c;
                s[k] += c * c;
            }
        }
    }
    vec4 finalColor;
    float min = 1e+2;
    for (int ik = 0; ik < 4; ik++)
    {
        m[ik] /= n;
        s[ik] = abs(s[ik]/n - m[ik]* m[ik]);
        float res = s[ik].r + s[ik].g + s[ik].b;
        if(res < min)
```

```

    {
        min = res;
        finalColor.rgb = m[ik].rgb;
    }
}

return vec4(finalColor.rgb,1.0);
}

```

- 结论

Kuwahara滤波算法由于卷积朝向和像素局部朝向相同，也就是轴对齐卷积核(Axis-aligned-Kernels)，这样的过滤方式可能会导致得到的图片有方形形状的感觉，一种解决方案是使用有向性滤波(Directional Kuwahara Filter)和各向异性滤波(Anisotropic Kuwahara Filter)。

2. 钢笔画

3. 卡通着色

//

6.7 老电影

1. 理论

- 对于旧电影后处理效果，可以采用简单的棕褐色着色与锐化滤波器相结合，并且可以加上粒子效果进行划痕和渐晕的处理

2. 着色器实现及效果

- 《孢子》中旧电影效果像素着色器代码如下

```

vec4 oldFilm(sampler2D originTex,sampler2D noiseTex,vec2 uv,vec2 imageSize)
{
    vec3 kSepiaRGB = vec3(0.8,0.5,0.3);
    float offsetX = 2.0;
    float offsetY = 2.0;
    float kNoiseTile = 1.0;//gpu pro is 5.0
    float kNoiseScale = 0.18;

    vec4 sourceColor = texture(originTex,uv);
    vec4 noiseColor = texture(noiseTex,uv);
    //sharpen filter
    vec4 tap0 = texture(originTex, uv + vec2(0.0, -offsetY)/imageSize);
    vec4 tap1 = texture(originTex, uv + vec2(0.0, offsetY)/imageSize);
    vec4 tap2 = texture(originTex, uv + vec2(-offsetX,0.0)/imageSize);
    vec4 tap3 = texture(originTex, uv + vec2(offsetX,0.0)/imageSize);
    sourceColor = 5.0 * sourceColor - (tap0 + tap1 + tap2 + tap3);

    vec4 converter = vec4(0.23,0.66,0.11,0.0);
    float bwColor = dot(sourceColor,converter);
    vec3 sepia = kSepiaRGB * bwColor;
    vec3 outColor = sepia * kNoiseTile + noiseColor.xyz * kNoiseScale;
    return vec4(outColor,sourceColor.a);
}

```

6.8 最小二乘法

1. 最小二乘概念

1. 概念

线性最小二乘，是统计学中用来进行线性回归分析，线性拟合的方法。给定一些离散的数据点位，来确定一条直线，使得该直线与这些点位的方差和最小，这就是最小二乘法。

2. 微积分处理

假定笛卡尔坐标系中给定三个离散数据点位(1, 1),(2, 2),(2, 3),确定一条直线，使得直线与点位之间的方差和最小

先设定一条直线函数 $y = kx + b$ ，那么方差和为：

展开来看：

如果要求方差和最小，那么就是求函数 $f(k,b)$ 的最小值。 $f(k,b)$ 是关于 k, b 的多元函数，多元函数求极值问题，一般而言，我们都是采用对 k 与 b 分别求偏导数。

令 $\frac{\partial f}{\partial k}(k,b) = \frac{\partial f}{\partial b}(k,b) = 0$ ，则解的 $k = \frac{1}{2}, b = \frac{3}{2}$, 则直线方程为 $y = \frac{1}{2}x + \frac{3}{2}$

3. 线性代数处理

我们仍然设直线 $y = kx + b$ ，则要求直线 y 拟合三个点(1,1),(2,1),(3,1)就是求以下线性方程组的解。

其矩阵形式 $Ax=b$ 为：

显然 $Ax = b$ 是无解，而确定 k,b 的过程就是求线性方程 $Ax = b$ 最优解的过程，使得 Ax 向量最靠近 b 向量。即 $\|Ax - b\|$ 小，而 $\|Ax - b\|$ 即为误差 e 。要求最优解则需将 b 向量投影到 A 矩阵的列空间中去，得到对应投影向量 \vec{b} ，从而线性方程变为 $A^T A \vec{x} = A^T b$ ，解得 $\vec{x} = \left[\begin{matrix} \frac{1}{2} \\ \frac{3}{2} \end{matrix} \right]$ 。因此 $k = \frac{1}{2}$, $b = \frac{3}{2}$ ，直线方程为 $y = \frac{1}{2}x + \frac{3}{2}$

线性代数中，规定，假如 $Ax=b$ 有唯一解，那么其充要条件是，向量 b 必然存在于矩阵 A 的列空间中

$A^T(A\vec{x} - b) = 0$ 其中 $(A\vec{x} - b)$ 代表的几何意义便是求解向量 b 投影到 A 矩阵的投影向量。

//<https://zhuanlan.zhihu.com/p/62694878>

2. 图像领域使用MLS

1. 概述

图像处理领域paper 《Image Deformation Using Moving Least Squares》 (<http://faculty.cs.tamu.edu/u/schaefer/research/mls.pdf>)利用移动最小二乘的原理实现了图像的相关变形，并且这片论文的引用率非常之高，可以称之为图像变形算法的经典，是Siggraph中的论文。其中刚性变换的效果是最好的。

该方法由用户指定图像中的控制点位以及控制点位所希望的目标点位，即控制点位，目标点位。假设 p 为原图像中控制点的位置， q 为目标点位。那么移动最小二乘法来为原图像上的每个像素点 v 构建相应的仿射变换函数，并通过该变换来计算得到图像变形后的位置：

其中 w_i 的表达式为：

其中 p_i 表示控制点位， v 表示网格像素点 v ，即网格像素坐标值， a 是一个参数

仿射变换 $L(v)$ 由两部分组成 $L(v) = Mv + T$ ，其中 M 为线性转换矩阵， T 为平移量。事实上将最小化表达式对变量 T 求偏导后可以得到 T 的表达式 $T = q^* - p^* M$, 其中 p^* 与 q^* 表示为：

于是仿射变换可以化简为 $\text{I}_v(x) = (x - p)M + q$ ，而最小化表达式可以变化为：

其中 $\text{vec}\{p_i\}$, $\text{vec}\{q_i\}$ 为：

其中最小二乘法并没有对转换矩阵M进行条件限制，如果对其进行添加限制后，就能得到不同形式的转换矩阵M。

2. MLS变换类型

• 仿射变换

仿射变换是利用经典正规方程对最小化表达式直接求解得到的结果：

有了旋转矩阵的表达式后，我们可以得到变形的表达式：

由于我们式通过控制 q (目标点)来实现图像变形的，而 p (控制点)是固定不变的，因此其中大部分内容可以预先计算并保存，从而提高运算速度，重写变形表达式如下：

其中 A_j 为：

由于仿射变换包含非均匀缩放，因此其效果不是很好。

• 相似变换

相似变形是仿射变换的一个特殊子集，它的变形效果只包含平移、旋转和均匀缩放，限制转换矩阵M满足

根据该条件得到相似变换的转换矩阵M为：

其中：

与仿射变换一样，提取可以预先计算的部分后得到变形表达式为：

其中 A_i 为：

• 刚性变换

刚性变换不含任何缩放效果，进一步限制转换矩阵使其满足

可得到刚性变换的表达式：

其中 $\text{vec}\{f_r(v)\}$ 为：

其中的 A_i 与相似变换中的 A_i 表达式相同。

3. CPU侧MLS变形

• 界面网格化(顶点以及UV坐标)

根据设定的行列计算出来顶点坐标以及uv坐标

• 计算控制点位

对单独的脸颊、眉毛、眼睛、鼻子以及下巴计算控制点位 p 以及目标点位 q

• MLS计算控制点位控制下的网格化的顶点(或者UV坐标)

根据控制点位 p 以及目标点位 q 来计算行列网格化后的顶点或者uv坐标

• 进行顶点、UV卷绕

将mls计算后的顶点或者uv，进行卷染按照三角形卷染方式，或者索引法组装索引数组

• 进行渲染

送入管线进行渲染

6.9 颜色查找表原理

1. 理论知识

颜色查找表有64个正方形，每个小正方形存着64*64的运算结果，对于颜色RGB(x,y,z)，先用Z值计算出来那个小正方形，再用xy读取对应的结果

- 用蓝色计算正方形的位置，得到quad1和quad2，整个图片的小格子的位置记为wow格子
- 根据红色值和绿色值计算wow格子内具体的像素小格子坐标，并加上wow的格子坐标值，以此来得到整体坐标
- 根据计算出来的两个整体uv坐标采样颜色查找表，并使用蓝色值进行mix操作

2. 着色器实现

```
vec4 lookupColor(sampler2D originTex,sampler2D lookTex,vec2 uv,float mixRadio)
{
    vec4 textureColor=texture(originTex,uv);
    float blueColor=textureColor.b*.63.; // 蓝色部分[0, 63] 共64种
    //第一个正方形的位置，假如blueColor=22.5，则y=22/8=2,x=22-8*2=6,
    //即是第2行第6个正方形；(因为y是纵坐标)
    vec2 quad1;
    quad1.y=floor(floor(blueColor)*.125);
    quad1.x=floor(blueColor)-(quad1.y*8.);
    //第二个正方形的位置，同上。注意x、y坐标的计算，还有这里用int值也可以，但是为了效率使用float
    vec2 quad2;
    quad2.y=floor(ceil(blueColor)*.125);
    quad2.x=ceil(blueColor)-(quad2.y*8.);

    vec2 texPos1;// 计算颜色(r,b,g)在第一个正方形中对应位置
    texPos1.x=(quad1.x*.125)+stepSize+(.125*textureColor.r);
    texPos1.y=(quad1.y*.125)+stepSize+(.125*textureColor.g);
    vec2 texPos2;// 同上
    texPos2.x=(quad2.x*.125)+stepSize+(.125*textureColor.r);
    texPos2.y=(quad2.y*.125)+stepSize+(.125*textureColor.g);

    vec4 newColor1=texture(lookTex,vec2(texPos1.x,texPos1.y)); // 正方形1的颜色值
    vec4 newColor2=texture(lookTex,vec2(texPos2.x,texPos2.y)); // 正方形2的颜色值
    vec4 newColor=mix(newColor1,newColor2,fract(blueColor)); // 根据小数点的部分进行
    mix
    vec4 blendColor=mix(textureColor,vec4(newColor.rgb,textureColor.w),mixRadio);
    return blendColor;
}
```

6.10 美颜相关算法

1. 全局磨皮算法

磨皮是多种算法合起来的综合的技术解决方案。其中包括去噪、平滑、保存边缘等操作。

- 去噪

中值滤波、高斯低通滤波、巴特沃斯滤波等降噪算法首先对原图像进行降噪记为降噪图I

- 高反差保留

采用很好的可以保留图像边缘的双边滤波算法，对降噪图进行处理，记为双边滤波图，其公式为：

其中H表示高反差保留图，F表示双边滤波图，I表示为降噪图

- 高斯模糊

对高反差保留图H进行高斯模糊，记为模糊图Y，其滤波的半径会直接影响肤质，半径越大，质感越强，但是太大反而会减弱磨皮的效果

- 图层混合

将模糊图与降噪图像进行混合，得到最终的磨皮图像z。其公式可以为：

其中Op为不透明度

2. 美白

1. 使用logarithmic Curve函数

- 原理

美白主要是使皮肤变白变亮，如果拥有一个合适的映射表，满足使得原图在色阶上有所增强，并在亮度两端增强的稍弱，中间稍强，则效果也会不错。图像增强论文《A Two-Stage Contrast Enhancement Algorithm for Digital Images》中有一个公式是这个样子：

其中 $w(x,y)$ 表示输入图像数据， $v(x,y)$ 表示输入结果， β 表示调节参数， β 越大，美白程度越大

- 着色器实现

```
• vec4 logwhiten(float beta,vec4 originColor)
{
    return log(originColor * (beta - 1.) + 1.)/log(beta);
}
```

7. 渲染优化

1. 针对移动端TBDR架构GPU特性的渲染优化

8. 高等数学与线性代数