# 消隐算法工程说明文档

## 1. 编程环境

本工程的主体使用了Visual Studio 2017作为编程IDE，使用语言为C++，除标准库外，使用了OpenCV 3.4.8版本用于显示和存储图像。

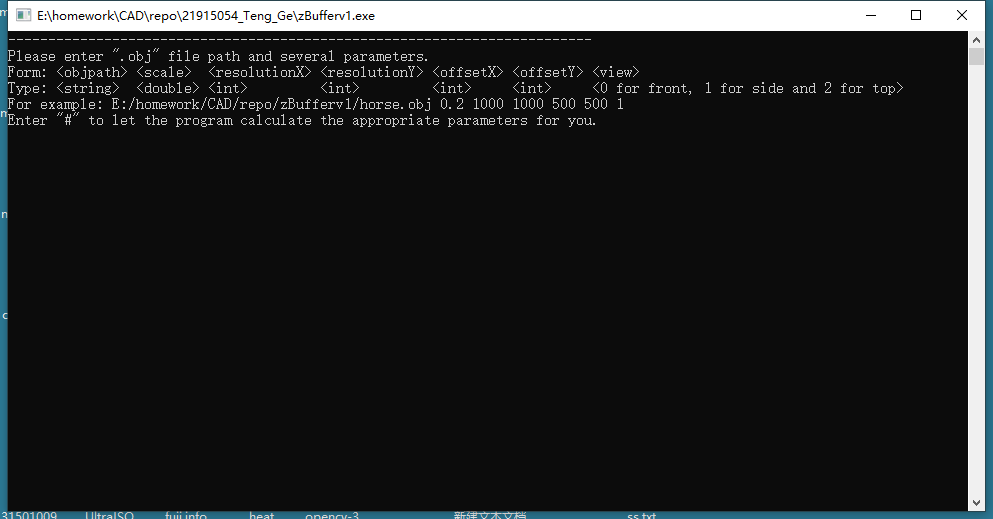
另外，使用Python写了一个生成随机面片的程序用于测试算法性能。（这个脚本与算法无关，仅用于生成测试用OBJ文件）。

## 2. 用户界面使用说明

本软件使用命令行与用户进行交互。（因时间原因没能学习并开发一个图形UI，因为我以前没怎么接触过图形UI的编程，甚至连C++也是边写边学的，所以在算法实现上花了很长时间）。

支持OBJ格式模型文件的输入。

**步骤1**：打开软件后首先可以看到如下图所示的提示：



此时有两种选择，一种是按提示上的格式输入OBJ文件的路径，然后再输入一系列参数用于定制渲染的效果，参数间用空格分开（跳转**步骤2-1**）。另一种则是直接输入“#“，此时算法会根据读入的OBJ里顶点的坐标分布，自动计算合适的参数（跳转**步骤3**）。

以下对各参数的意义作简要解释：

objpath：OBJ文件的路径，请注意里面**不要带空格**。

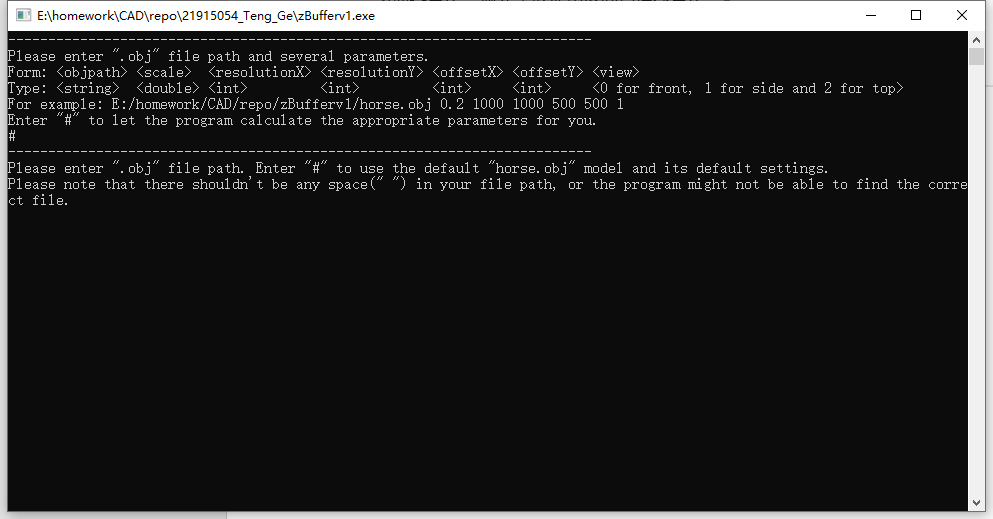
resolutionX，resolutionY：X和Y方向上的画布大小。

scale，offsetX，offsetY：用于对模型进行缩放，并放置到画布内。假设OBJ中定义的模型，其顶点坐标范围可能是[x1, x2], [y1, y2]。而我们的画布的空间范围却是

[0, resolutionX], [0, resolutionY]。因而引入这三个参数对其进行调整。其中scale用于调整模型的放大倍率，即每个顶点的坐标值均会乘以scale。两个offset则是对每个坐标放大后的顶点进行X和Y方向上的平移，即每个X坐标会加上offset X，Y坐标类似，最终使其落在我们需要的空间范围内。

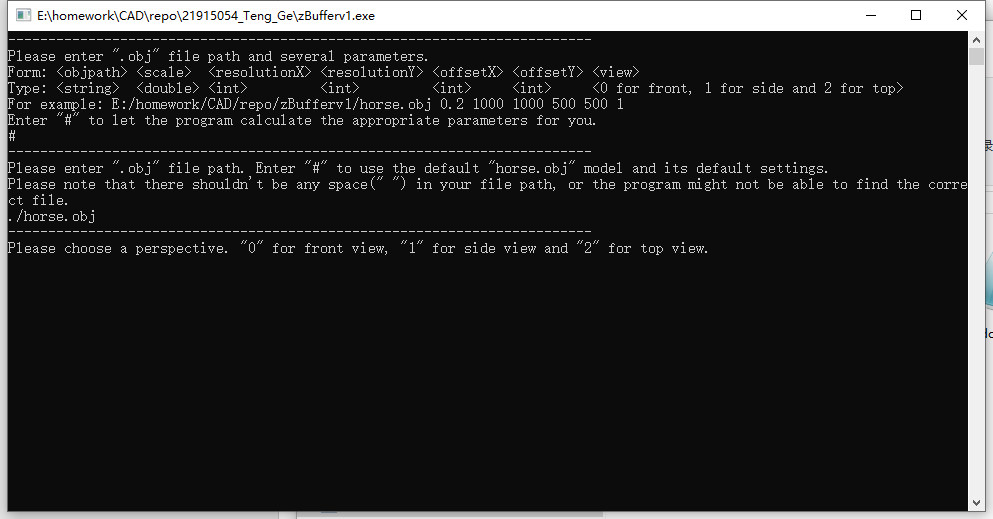
View：该参数用于调整绘制的视角，主要被用于验证渲染的正确性。目前提供了三视图视角（因为这个实现起来很方便），其中0代表正视图，1代表侧视图，而2代表俯视图。

**步骤2-1**：倘若在第一步中输入了“#“，则会弹出如下提示：

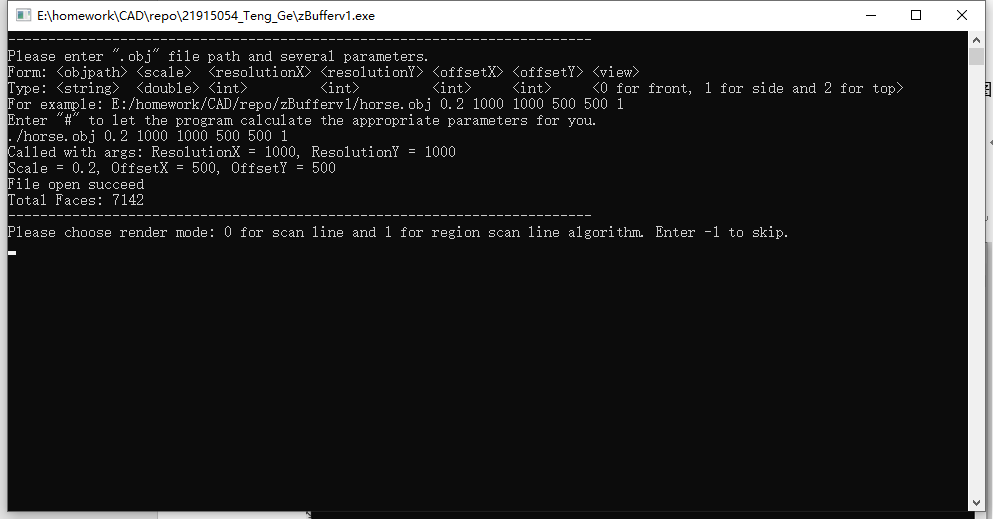


此时依旧有两种选择，一种是输入“#“，则将会调用随程序附带的“horse.obj”以及它的默认设定（跳转**步骤3**）。另一种则是手动输入想要渲染的OBJ文件路径（跳转**步骤2-2**）。

**步骤2-2**：完成OBJ文件路径输入后，将弹出如下提示，此时需要选择渲染的镜头角度，具体参考**步骤1**中的描述。（跳转**步骤3**）

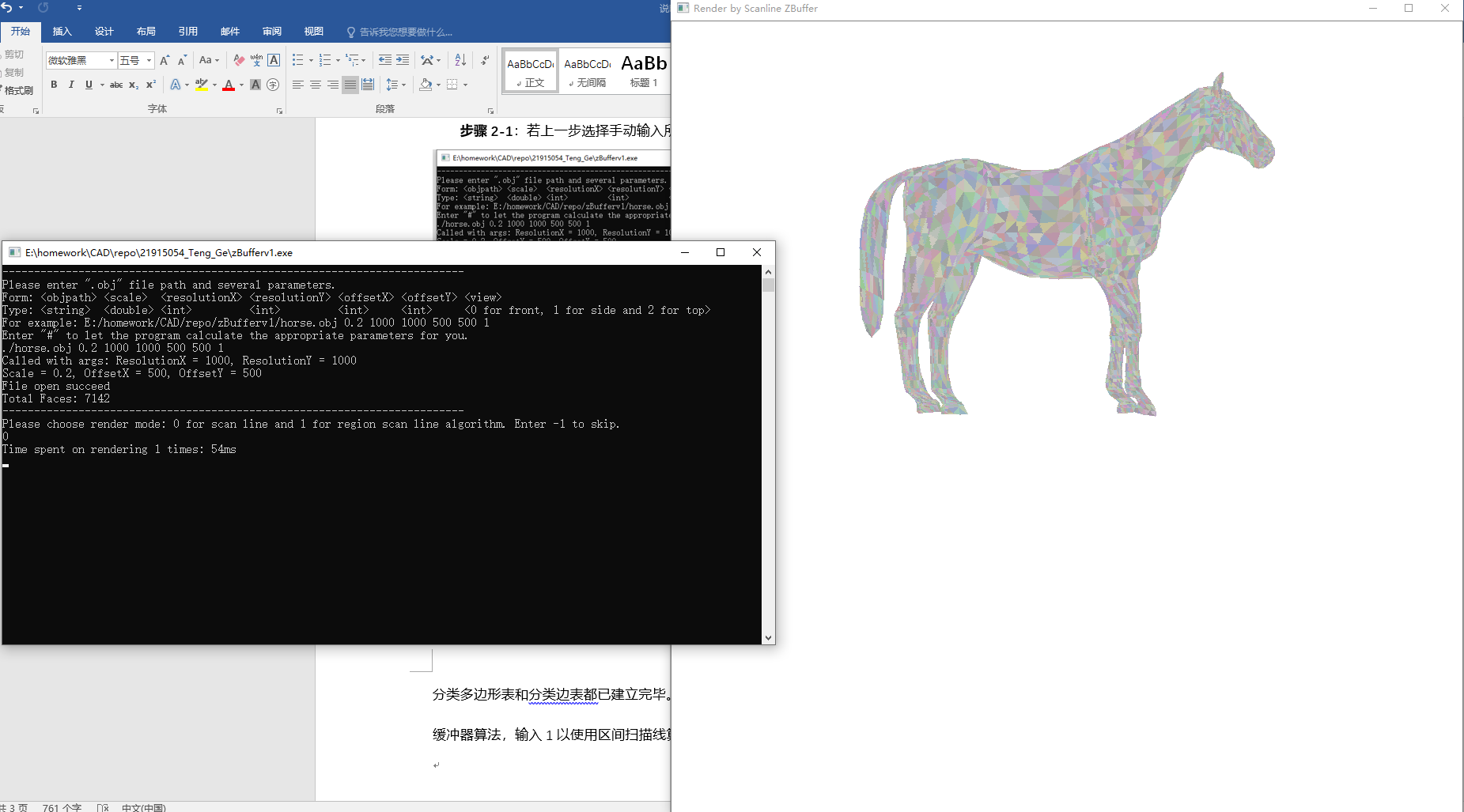


**步骤3**：完成参数配置后，会弹出如下提示：



一般如果没有其他错误提示，且Total Faces数目不为零，则说明OBJ读取正确。此时分类多边形表和分类边表都已建立完毕。接下来需要选择渲染算法：输入0以使用扫描线Z缓冲器算法，输入1以使用区间扫描线算法，在前面参数输入错误或前面步骤出错的情况下，也可以输入-1以跳过渲染，进入下一轮循环。

按下回车后将弹出渲染结果，左侧将输出渲染所用时间，以毫秒为单位：



关闭弹出的图像或尝试按下回车即可开始下一轮循环。

**常见错误及可能的原因**：

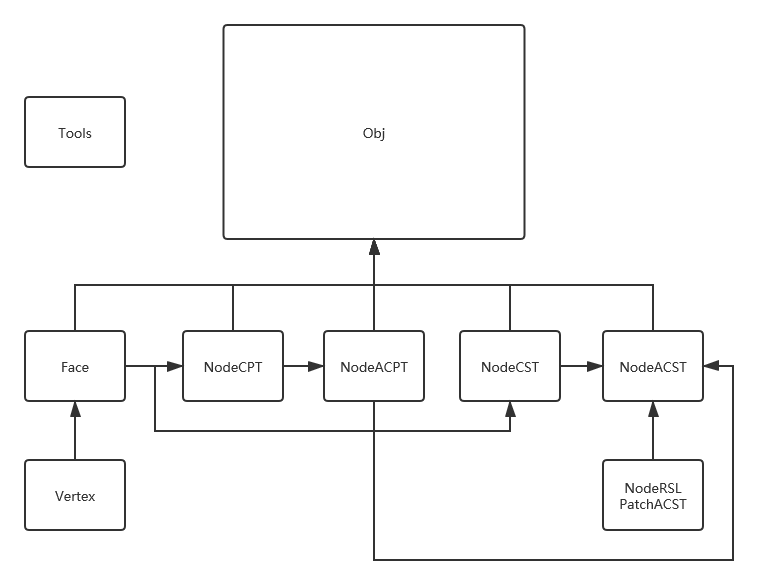
1. 提示“File does not exist!”，一般是因为OBJ文件路径输入错误导致的。请注意路径中**不要带空格**，使用“/”分隔路径中的文件夹。

2. 提示“Vertex out of screen!”一般是参数配置错误导致的。应该是有顶点在经过缩放和平移变换后超过了画布范围。可以尝试使用“#”，让程序自动生成合适的参数。

3. 画面绘制错误。目前程序的功能还很局限，因此在一些输入情况下可能出现绘制错误。两种算法都不能处理由边数量大于等于5的多边形建模的模型，其会导致模型绘制不完整。上色采用的是随机上色，所以两次绘制的结果可能面片颜色不同。区间扫描线算法不能处理多边形贯穿的情况，会导致类似下图（左）所示的绘制错误：

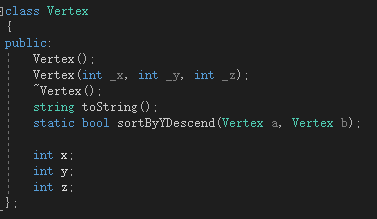
## 3. 数据结构和算法说明



上图所示的是本工程的所有类的结构关系图，其中箭头方向A->B表示A类为B类的附属，B类的建立依赖A类，或A类的对象被包含于B类。

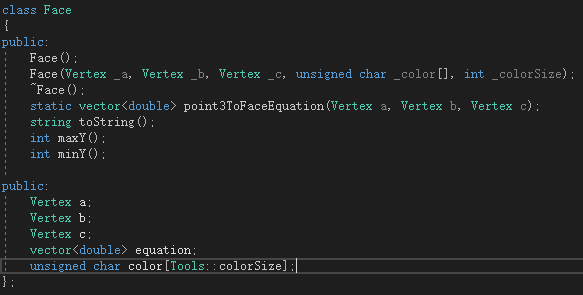
接下来对各个类进行简要的介绍：

**Vertex：**



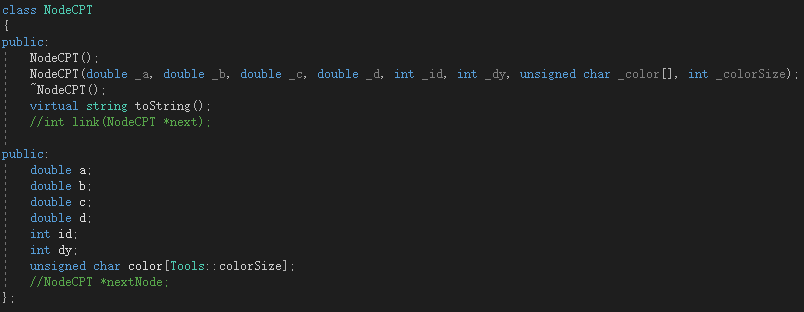
Vertex类表示一个顶点，在读取OBJ文件时直接创建其对象，提供暂存每个顶点坐标的功能。提供一个用于排序的参数函数，和一个用于调试的输出为字符串的toString()函数。

**Face**：



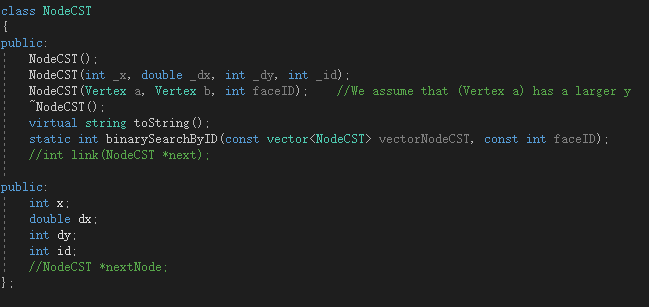
Face类表示一个面片，由颜色数组，平面方程参数和三个顶点组成（4个顶点的面片被我简单粗暴地拆成了两个3顶点面片，虽然导致效率降低，但简化了实现）。也在读取OBJ文件时创建其对象，用于暂存面片信息。

**NodeCPT**：



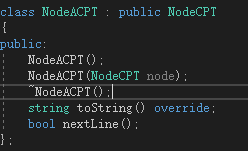
CPT是Classified Polygon Table的简称，表示分类多边形表的一个节点。相对应的分类多边形表就是vector<vector<NodeCPT>>。实现上和老师PPT上没什么区别。

**NodeCST**：



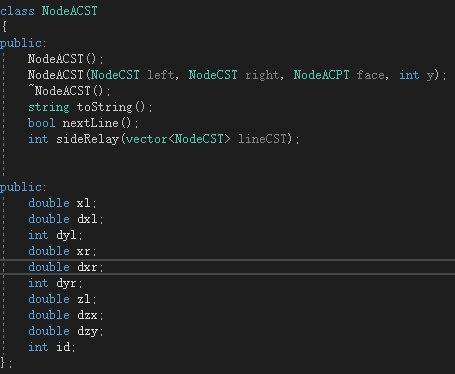
CST是Classified Side Table的简称，表示分类边表的一个节点。数据结构上和老师的PPT上一致，提供由两个顶点和4个参数两种构造方式。提供一个根据ID二分搜索的方法。

**NodeACPT**：



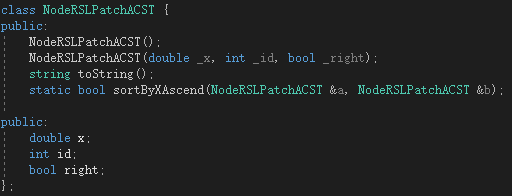
A是Active的意思，所以这个类表示活化边表的一个节点。因为和NodeCPT具有相同的数据结构，因而直接继承了NodeCPT类。在此基础上重写了toString()字符串化函数并新增了一个nextLine()函数用于在进入下一条扫描线时更新该节点的参数值。

**Node ACST**：



NodeACST表示活化边表的节点，其由一个活化多边形表节点和两个对应的分类边表节点构造。数据结构与老师PPT上的一致。函数方面提供了一个nextLine()更新函数，功能与NodeACPT中的一致，一个sideRelay()函数，用于在扫描过程中边对中的一条边结束而另一条边仍在的情况下，到分类边表CST中找到相应的边来替换已结束的边。

**NodeRSLPatchACST**：

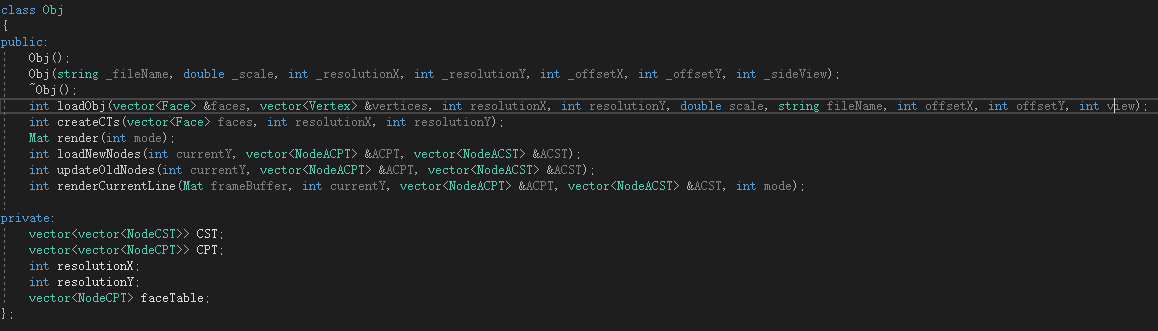


全称是Node Region Scan Line Patch ACST。意为，活化边表-在使用区间扫描线算法时-新增的补丁-的一个节点。因为要求两种算法要使用相同的基本数据结构，但我不太确定哪些数据结构算是“基础的”，所以在实现区间扫描线算法的时候，我干脆重用了所有扫描线ZBuffer算法的数据结构（包括活化多边形表和活化边表），只新增了这个补丁用于加速算法。

这个类的每个对象和活化边表中的一条边对应，其的成员变量有三个：一个double x用于表示扫描线和该边交点的x坐标，一个int id用于表示该边对应的多边形编号，一个bool right变量表示该边是对应活化边表边对中的左边还是右边。

通过打上这个补丁，可以更快速地提供区间扫描线算法中，由当前交点寻找下一个交点的功能，即PPT 23页中的next\_item()功能。

**Obj**：

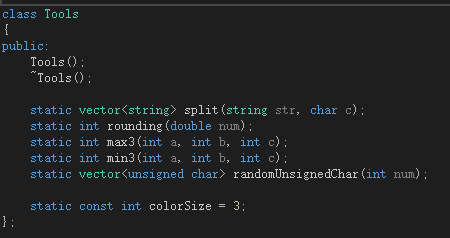


这是整个软件的主类，每个Obj对象对应一个OBJ模型。其成员变量包括一张活化多边形表CPT和一张活化边表CST，另外还有定义画布大小的resolutionX和resolutionY，最后是一个为加速区间扫描线算法加入的冗余结构faceTable，其本质上是一个按照id排序的多边形表，用于快速通过id获得多边形的平面方程和颜色。

每一个Obj对象由一个OBJ模型文件和一系列输入参数构造。构造大致分为两个步骤，第一步调用函数loadObj()读取OBJ文件，并建立vector<Face>过渡结构暂存数据。第二步调用createCTs()函数，通过vector<Face>建立分类多边形表，分类边表和冗余结构faceTable。

除构造用函数外，Obj对象提供一个主要函数接口render(int mode)。其功能是根据分类边表和分类多边形表绘制出图像。Mode参数用于选择绘制算法，0表示扫描线zBuffer，1表示区间扫描线算法。Render的绘制过程对于每一条扫描线来说分为三步，updateOldNodes()用于更新活化表中的节点，包括去除绘制结束的节点。loadNewNodes()使得活化表载入该行的新节点。renderCurrentLine()利用当前的活化多边形表和活化边表的内容绘制当前行的图像。对于两种算法，他们共用了相同的updateOldNodes()和loadNewNodes()函数，只在renderCurrentLine()函数中有所不同。

**Tools**：



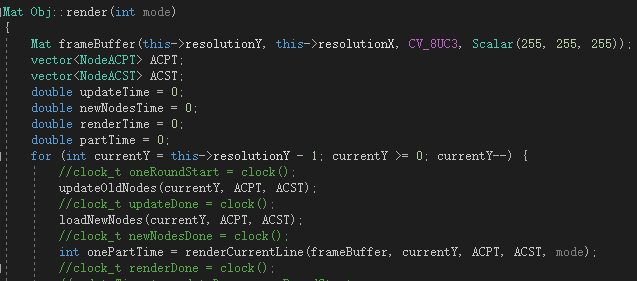
Tools是一个工具类，提供了一些通用的工具函数，包括字符串分割split()，四舍五入rounding()，三个元素求最大最小值，生成随机颜色值等。

## 4. 加速

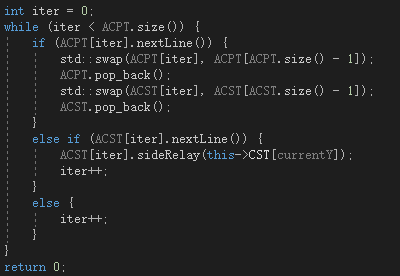
加速方面主要采用了两种思路。其一是利用冗余的数据结构，用空间换取时间。其二是利用数据的有序性或无序性从而节约不必要的计算。

因为在测算时间时我只计算了render()函数的执行时间，作为渲染时间，因此建立分类多边形表和分类边表以及之前的步骤我没有进行很细致的优化，主要优化了维护活化表和绘制的部分。

### 4.1 更新已在活化表中的节点



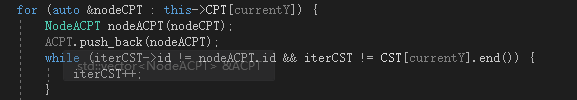
接下来按照render()调用的三个函数的顺序分别进行说明：



首先是更新已经存在于活化表中的节点，这里可以看到，由于在将节点放入或移出活化表的过程中，我们保持了两个表对应位置元素的相关，即两张活化表同一个下标的元素拥有同一个id（出自同一个面片）。因此我们**在更新时可以用同一个下标访问两张表**，因而节约了查找的时间。

更新时可能遇到三种情况，其一是多边形dy < 0（if内部的代码），因此我们需要同时将该多边形对应的ACPT节点和ACST节点同时移出表（**同时移出表可以省去更新ACST对应节点的时间**），可以看到这里删除节点用的是将节点换到尾端，然后pop\_back()的方法，因为我们并不在意这些节点在表中的顺序，只要他们表间相对位置一样就行。**所以用这种方法可以以O(1)的复杂度实现移除**。其二是若活化多边形表（ACPT）节点更新后dy >= 0，此时需要对活化边表（ACST）进行更新，更新完之后如果发现其中一边的dy = 0，则此时需要对这条边进行“接续“。接续的方法是回到分类边表（CST）中对应行取出同id的节点对ACST进行更新，这里因为当初建立CST时每一行都是按照id排序的，因此可以**用二分查找O(logN)找到相应的节点**。其三是以上两种情况均没有发生，则iter++更新后一对节点。

### 4.2 加载新活化表节点

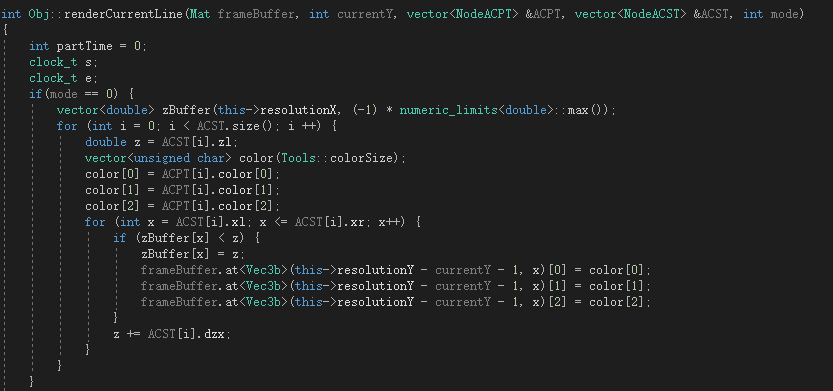


然后是loadNewNodes()部分，这一部分比较简单，唯一值得注意的是因为分类多边形表CPT和分类边表CST的每一行都是按id排序的（因为建表时是按id顺序执行插入的）。因此我们可以**用两个迭代器轮流向后搜索的方式进行查找**，从而以O(n)的时间复杂度实现功能。

### 4.3 绘制当前行

最后时renderCurrentLine()部分，这一部分两种算法不相同，因此我对他们分别进行说明。

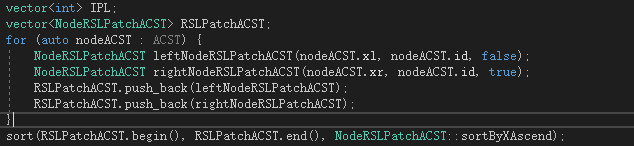
#### 4.3.1 扫描线Z Buffer算法



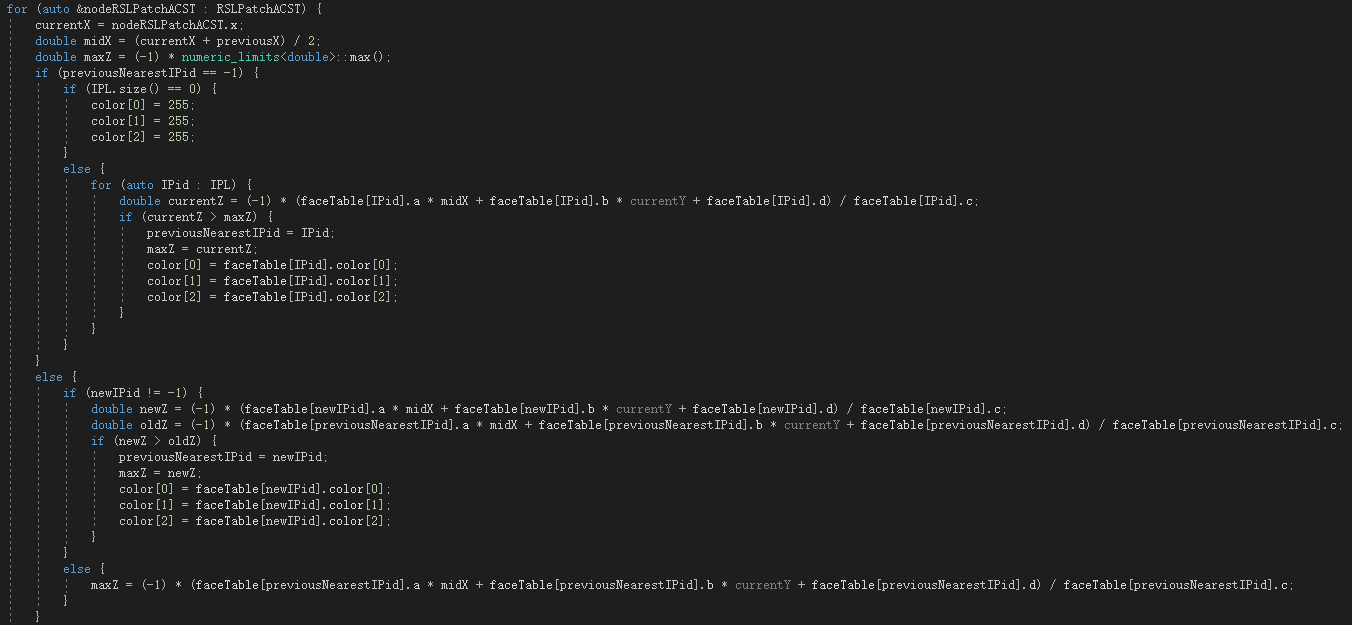
首先是扫描线ZBuffer算法，这个比较简单，就是两个for循环，先对每个边对，再对每条边，这里利用了ACST和ACPT的对应关系来快速地取出边对对应的颜色值。

#### 4.3.2 区间扫描线算法

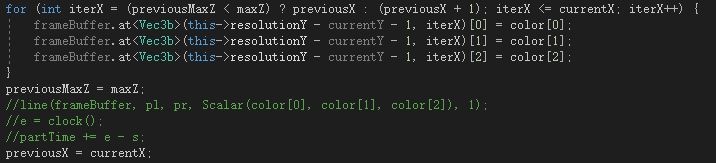
这一块比较复杂，也是我花了最多时间优化的地方。最后10000面片从4秒优化到了200多毫秒。



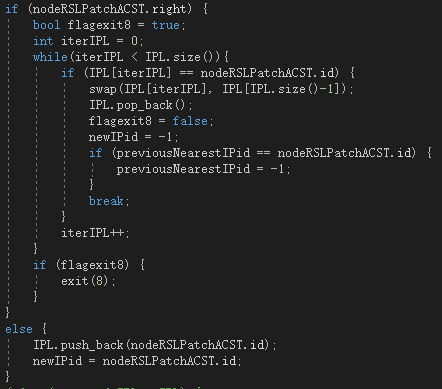
首先是建立ACST的补丁，这个补丁主要用于提供快速的从一个交点获得右侧下一个交点的功能（PPT伪代码中的next\_item功能）。为此需要对其根据x坐标值排序，这里排序的顺序是，先按x坐标排，如果x相等则边对中的左边在前。通过排序可以以O(NlogN)的复杂度完成该功能，而不排序就需要O(N2)的复杂度。



然后是决定某一个线段所看见的多边形颜色的部分，这一部分本来用了取出所有标记为in的多边形进行比较的方法，后来参考PPT进行了逻辑上的优化。现在代码逻辑如下：每次记录上一次可见的多边形ID。如果经过了一个交点，如果该交点是由边对的右边产生的（离开多边形），那么判断该边对是不是属于上次可见的多边形，如果不是，那么沿用上次的多边形，如果是，那么重新比较所有进入了的多边形。**而如果该交点是由边对的左边产生的（进入多边形），那么比较上次可见多边形和新进入的多边形的深度，以决定当前可见的多边形颜色。这里取颜色使用了faceTable的冗余结构，可以以O(1)的复杂度根据id取颜色。**



然后是绘制颜色部分，这里还是手工一个一个像素填颜色，尝试用过OpenCV的画直线函数，但是发现效率极其低下，因而放弃了。



最后是更新IPL的部分，这里也用了交换再pop\_back()的方法，因为我们并不在意IPL中元素的顺序。IPL是一个仅包含进入了的多边形ID的vector。