# 消隐算法实验结果报告

## 1. 功能测试

功能实现情况：

实现了扫描线Z缓冲算法的所有功能，实现了区间扫描线的大多数功能，但没有实现贯穿处理和利用多条扫描线间的深度连贯性进行优化。

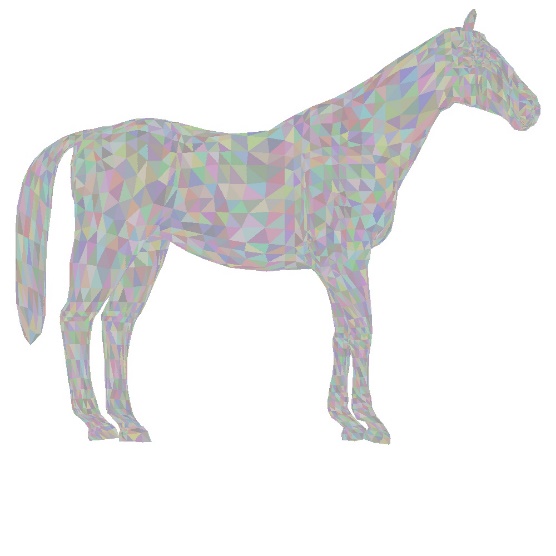
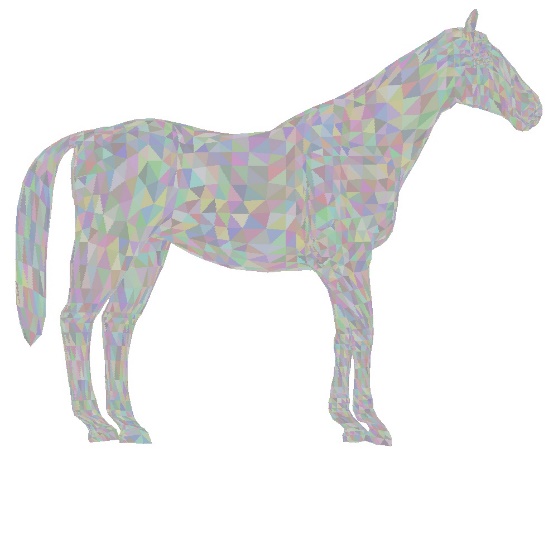
两种算法共用了扫描线Z缓冲器算法的所有数据结构，区间扫描线算法只是额外添加了一些简单的加速用数据结构。

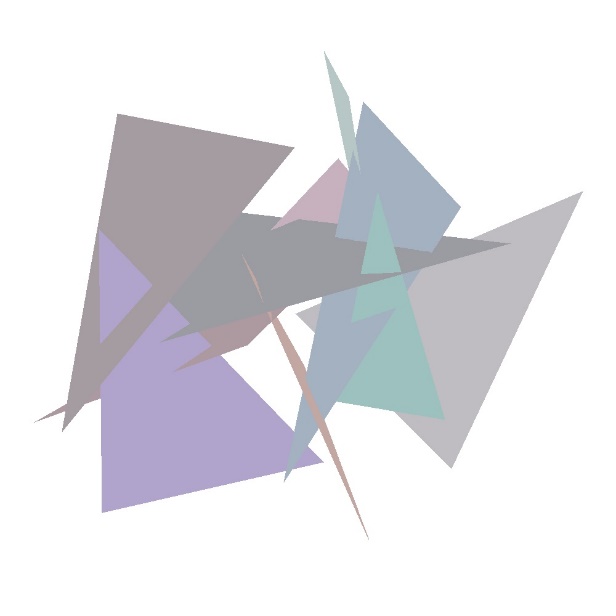
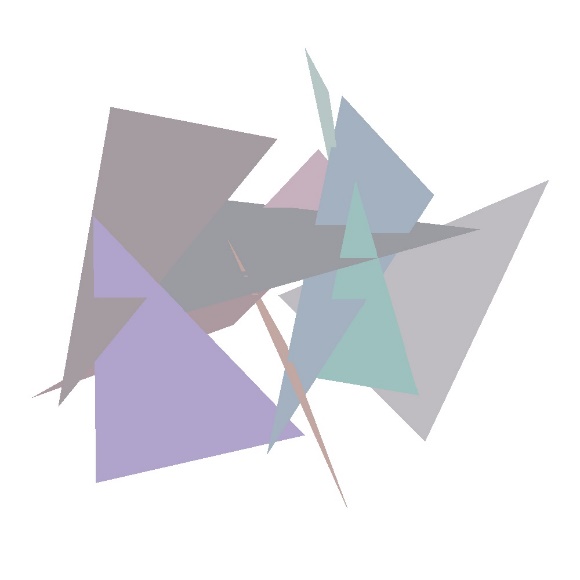
实现了一个简单（但还算好用？）的命令行UI。

支持三视图三个角度的绘制。

支持OBJ模型的导入，支持四边形和三角形面片的建模（其他形状暂时没有遇到，因而不支持）。四边形面片的绘制通过将其切割成两个三角面片完成，有待优化。

下图右侧为扫描线Z Buffer算法绘制的结果，左侧为区间扫描线绘制的结果。可以看到区间扫描线算法在绘制贯穿图形时会出错。



## 2. 性能测试

性能方面，我采用了生成一定数量的随机面片的方式来对算法性能进行测试。详细生成代码见OBJFile/Random/generate\_random\_obj.py。测试的范围是render()函数的运行时间，即从分类多边形表和分类边表建立完成开始，到绘制出矩阵形式的图像所消耗的时间。为消除随机误差，采用了绘制100次取平均的方法。在测试中使用了4种面片数量，2种面片大小，其中大面片由700范围内取随机三点获得，小面片由100范围内取随机三点获得。

测试环境为Inter® Core™ i5-7500 CPU @ 3.40GHz。

测试结果如下：

|  |  |  |
| --- | --- | --- |
| 面片数量 | 扫描/ms | 区间/ms |
| 10小 | 6.00 | 8.09 |
| 100小 | 9.54 | 18.95 |
| 1000小 | 44.83 | 38.03 |
| 10000小 | 329.53 | 227.07 |
| 10大 | 14.39 | 13.10 |
| 100大 | 52.55 | 29.58 |
| 1000大 | 333.64 | 167.24 |
| 10000大 | 2929.43 | 3307.22 |

通过上方图表可以看出，区间扫描线算法**在面片较小且数量较多，或面片较大且数量不是很多时拥有较高的效率**。

为了解这种现象的成因，我们可以对这两种算法的复杂度进行简单的分析：

为计算方便，我们假设所有面片投影和画布都是正方形且平均分布。假设面片的边长平均为S（或更简化一点，均为S），画布边长为M，面片数为N。那么可以计算得：平均每条扫描线穿过了K = N×S/M个面片。

因此扫描线ZBuffer算法的复杂度易得，为K×S×M，即**N×S2**。

而区间扫描线算法较为复杂。就我目前的实现来说，首先排序部分的复杂度约为2K×log(2K)，计算可见多边形部分的复杂度比较难算，其应当小于等于每次经过右端点时，当时的IPL中元素个数的和（最差情况为等于，此时每次经过的右端点都属于当时最顶层的多边形）。我们可以通过简化计算得，其复杂度为：N2(2S>M), N×S(2S<M)，计算过程见最后。绘制颜色部分总复杂度为M，因为每个点差不多只画一次。最后的更新IPL部分的复杂度即为计算可见多边形最差情况的复杂度。因而最终的复杂度为2K×log(2K)+M+C，其中

C= N2(2S>M), N×S(2S<M)。当给定S/M比时，复杂度为

**N2(2S>M), N×logN+N×S(2S<M)**。

由两个算法的复杂度可以看出：

**当面片较小的时候，得益于NlogN的排序复杂度，区间扫描线算法即使在N比较大的情况下也有相当良好的性能。要等到logN超过S排序才会变为主要耗时的因素。因而例如对于边长S为20的面片，需要有100万以上的面片才会使得算法的复杂度受到排序复杂度NlogN比较明显的影响，而在这之前复杂度始终靠近N×S，因而比扫描线ZBuffer算法的N×S2要快。在面片很少的时候，则是因为区间扫描线额外建立了一些数据结构而导致其耗时比较长。**

**而当面片较大时，区间扫描线的算法复杂度为N2，因而受面片数量影响很大，当面片数超过一定数量后，其耗时上升得就比N×S2的扫描线ZBuffer快很多了。**

以下附区间扫描线部分复杂度的计算过程（因为公式很多所以就手写了，可能不是很严密，但是和实验结果大致相符）：

