实现细节

BVH实现细节

对于BVH的实现，在分割的时候我选择的是使用平分元素的方法进行分割。而为了避免每次排序带来的开销，我是用了C++标准库的nth_element用于找出深度在中间的元素从而避免了每次分割都进行排序。除此之外，需要注意的是排序所对比的值应该为三角形的z值的最大值（离视点最近的值）。

```
else{
    // 按照z轴分割 平分
    int mid = (start + end) / 2;
    std::nth_element(&triangle_info[start], &triangle_info[mid], &triangle_info[end - 1] + 1, [](const BVHTri
        return a.bounds.max_z > b.bounds.max_z;
    });// 对中间位置排序
    node -> InitInterior(RecursiveBuild(triangle_info, start, mid, ordered_prims), RecursiveBuild(triangle_in
```

构建完BVH后，借助深度优先搜索找到所有叶节点提前存储起来。

```
void FindLeafNode(BVHNode *root){
    if(root == nullptr)return;
    if(root -> n_triangles > 0){
        // 叶节点
        leaf_nodes.push_back(root);
    }else{
        // 非叶节点 深度优先遍历
        FindLeafNode(root -> children[0]);
        FindLeafNode(root -> children[1]);
    }
}
```

Scanline zbuffer实现细节

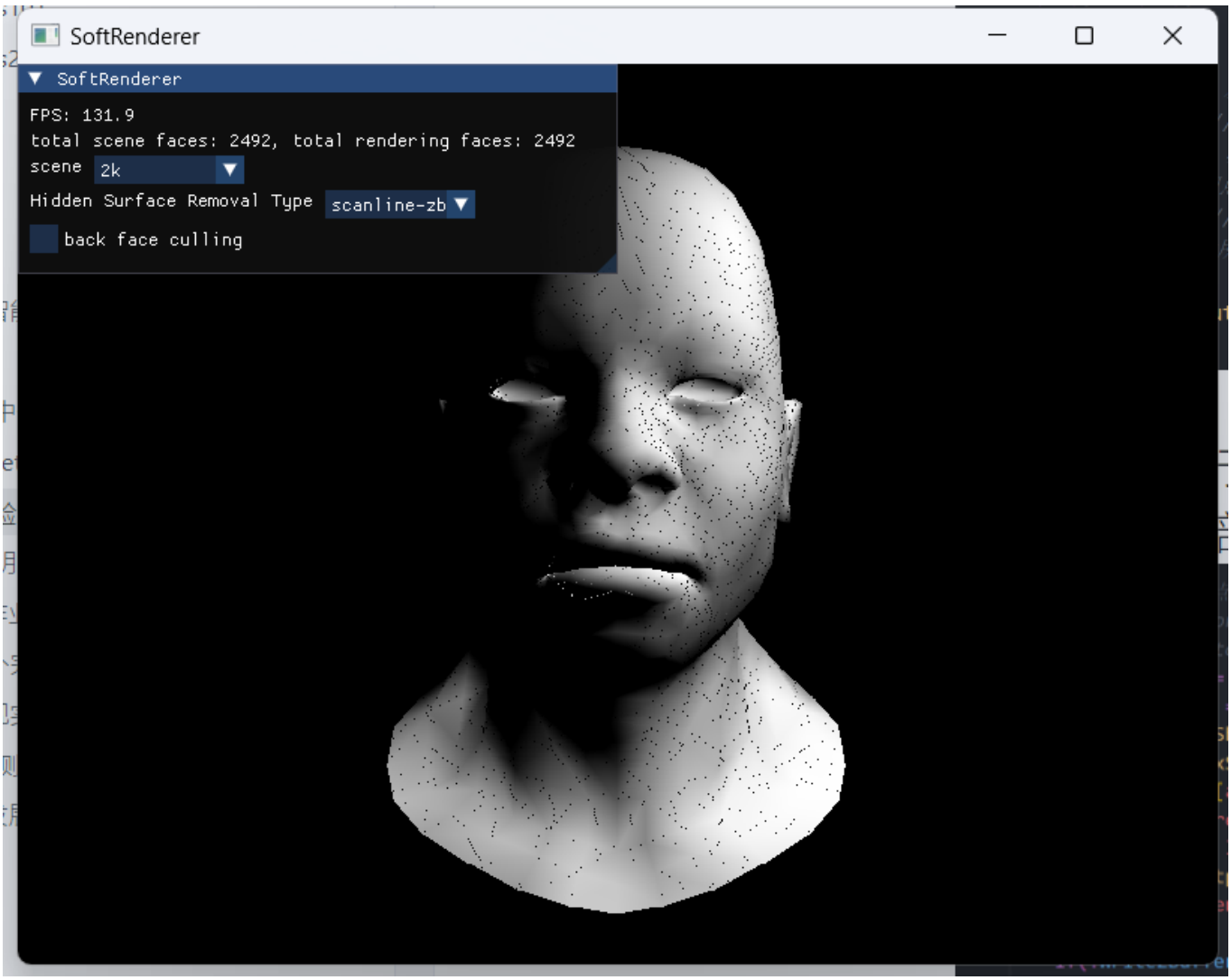
对于Scanline zbuffer，原本的假设是同一个面片里面的所有像素都是相同的颜色。而我的实现中三角形的每个像素是将三个顶点的属性插值后使用偏远着色器进行着色，于是我在Scanline zbuffer中也将着色模式统一为上述方法。在具体的数据结构中额外加入了对应的属性。

```
// 活化边类
class ActiveEdge{
public:
    ActiveEdge(float xl, float xr, float dxl, float dxr, int dyl, int dyr, float zl, float dzx, float d
    xl(xl), xr(xr), dxl(dxl), dxr(dxr), dyl(dyl), dyr(dyr), zl(zl), dzx(dzx), dzy(dzy), id(id){}
    ActiveEdge(){}

    float xl, xr; // 左右交点的x坐标 当前
    float dxl, dxr; // 左右交点边上两相邻扫描线交点的x坐标之差
    int dyl, dyr; // 以和左右交点所在边相交的扫描线数为初值 以后向下每处理一条扫描线减1
    float zl; // 左脚点深度值
    float dzx, dzy; // 沿扫描线向右（下）走过一个像素时，多边形所在平面的深度增量。
    int id; // 交点对所在的多边形编号

    VertexShaderOutput l_vs_output[2], r_vs_output[2]; // 左右边的顶点信息
};
```

而在插值时由于z值也能够插值得到，故此处的zl、dzx、dzy并没有被使用。光栅化部分如下：

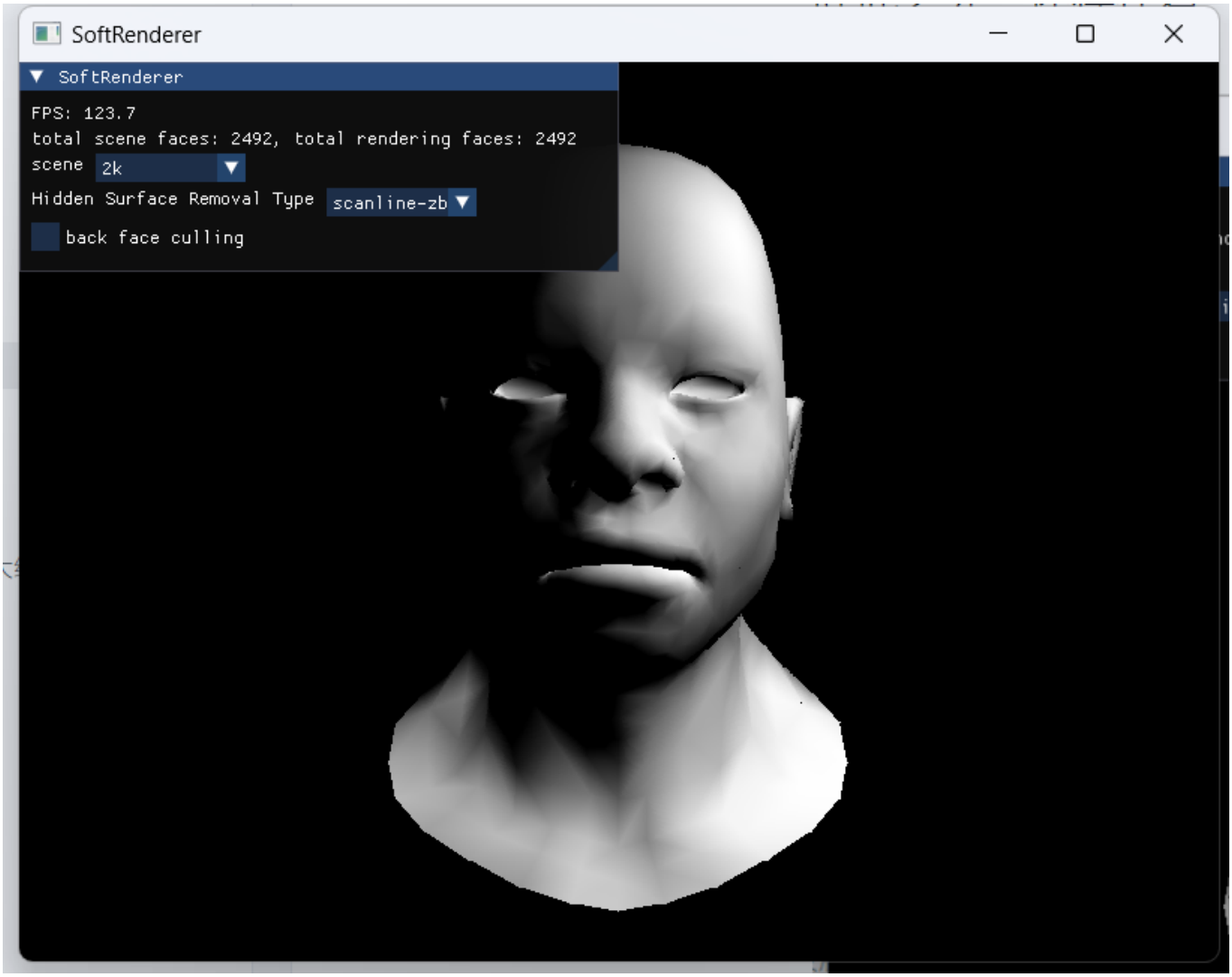
```
    }
    // 插值得到两个边界点的属性
    // float left_factor = (active_edge.l_vs_output[0].screen_position(0) - active_edge.xl) / (active_edge.l_vs_out
    // float right_factor = (active_edge.r_vs_output[0].screen_position(0) - active_edge.xr) / (active_edge.r_vs_ou
    float left_factor = 1 - active_edge.dyl / (active_edge.l_vs_output[0].screen_position(1) - active_edge.l_vs_out
    float right_factor = 1 - active_edge.dyr / (active_edge.r_vs_output[0].screen_position(1) - active_edge.r_vs_ou
    auto left = VertexShaderOutput::Lerp(active_edge.l_vs_output[0], active_edge.l_vs_output[1], left_factor);
    auto right = VertexShaderOutput::Lerp(active_edge.r_vs_output[0], active_edge.r_vs_output[1], right_factor);
    auto shader = tris[active_edge.id].shader;
    int len = right.screen_position(0) - left.screen_position(0) + 1.5;
    for(int j = 0; j < len; j ++){
        VertexShaderOutput v = VertexShaderOutput::Lerp(left, right, j * 1.f / len);
        int x = v.screen_position(0) - 0.5;
        int y = i;
        if(!WriteZBuffer(x, y, v.screen_position(2)))continue;
        Vec3f color = shader -> FragmentShader(v);
        y = height - y - 1;
        framebuffer[3 * (y * width + x)] = color(0);
        framebuffer[3 * (y * width + x) + 1] = color(1);
        framebuffer[3 * (y * width + x) + 2] = color(2);
    }
```

除此之外，还存在着一些精度累计误差问题，有时会导致某个像素没有被绘制：



对此我的做法是在光栅化时多光栅化一个像素，可以大大改善这一现象。

```
int len = right.screen_position(0) - left.screen_position(0) + 1.5;
```



背面剔除

由于主要的时间消耗是在光栅化阶段，于是我使用了背面剔除，在顶点变换后在ndc空间根据片元法向量和观察方向的夹角判断是否光栅化三角形：

```
bool Renderer::BackFaceCulling(VertexShaderOutput &v1, VertexShaderOutput &v2, VertexShaderOutput &v3) {
    // 使用NDC坐标
    auto ndc_v1 = v1.ndc_position, ndc_v2 = v2.ndc_position, ndc_v3 = v3.ndc_position;
    Vec3f t1 = {ndc_v2(0) - ndc_v1(0), ndc_v2(1) - ndc_v1(1), ndc_v2(2) - ndc_v1(2)};
    Vec3f t2 = {ndc_v3(0) - ndc_v1(0), ndc_v3(1) - ndc_v1(1), ndc_v3(2) - ndc_v1(2)};
    Vec3f normal = t1.cross(t2).normalized();
    Vec3f view = {0, 0, 1}; // ndc空间观察方向
    return view.dot(normal) < 0;
}
```

效率对比与分析

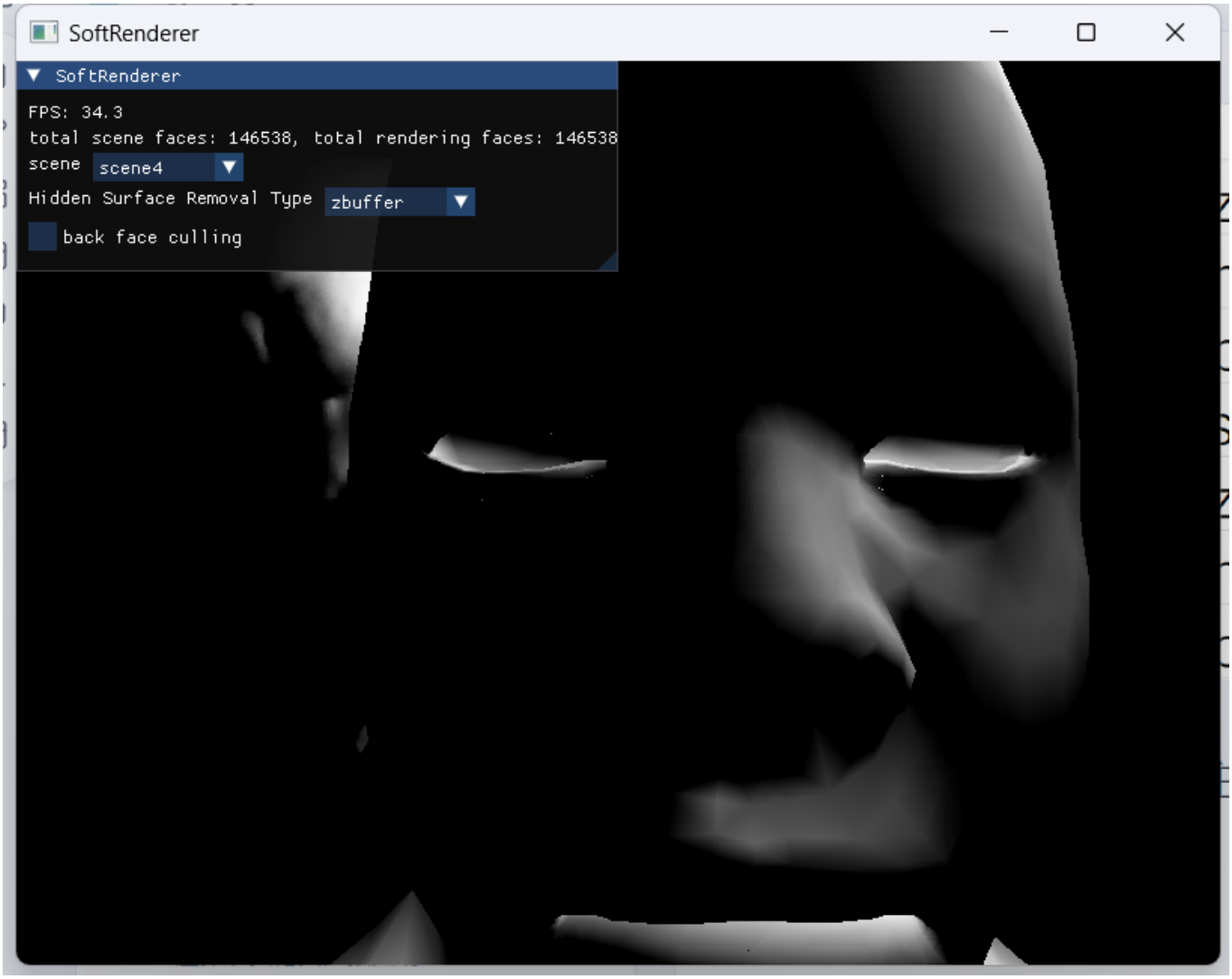
此处选择2k、15k、144k的模型进行光栅化，在800 * 600的窗口下，以稳定后的帧率为指标，得到的结果如下：

	2k	15k	144k
zbuffer	162.1	77	16.7
hiz	68.6	43	11.4
bvh+hiz	72.5	45.3	13.4
scanline	125.2	19.8	0.8
zbuffer+背面剔除	207.7	105.8	23.4
hiz+背面剔除	78.4	53.1	16.4
bvh+hiz+背面剔除	80.2	56.8	18

主要结果如下：

1. 在面片越来越多的情况下，scanline zbuffer的时间换空间策略带来的影响越来越明显，从2k的优于hiz与bvh+hiz到15k和144k的全面落后。
2. 背面剔除能够带来可观的帧率提升
3. hiz、bvh+hiz和scanline zbuffer都比不过初始的zbuffer，通过观察实际面片渲染数量发现，**在只渲染一个物体的情况下，遮挡关系简单，没有实现快速拒绝。**

因此，为了测试hiz和bvh+hiz的效果，我构造了一个明显遮挡关系的场景：



使用了一个大的人头遮挡住后面的bunny物体，此处有两个物体，绘制顺序会对hiz的效果产生影响。于是我对比了从远到近绘制（对应scene4）以及从近到远绘制（对应scene5），对比渲染fps，得到结果如下：

	scene4	scene5
zbuffer	13.2	15.8
hiz	9	14.1
bvh+hiz	16.4	16.6
scanline	0.8	0.8
zbuffer+背面剔除	18.7	21.8
hiz+背面剔除	12.4	17
bvh+hiz+背面剔除	18.8	19.2

结果分析如下：
在有复杂遮挡的场景中，bvh+hiz能够实现超越普通zbuffer的性能，而且不在意绘制顺序。而hiz则更取决于面片的就只顺序，在从远到近绘制时明显不同普通zbuffer，在从近到远绘制时和普通zbuffer接近。

下面给出实际渲染面片（表格内容为实际渲染面片数量）的结果：

总面片:146538		
	scene4	scene5
zbuffer	146538	146538
hiz	146386	44582
bvh+hiz	44424	44424
scanline	146538	146538
zbuffer+背面剔除	61178	61178
hiz+背面剔除	61115	19678
bvh+hiz+背面剔除	19652	19652

总结

本次作业主要是实现了四种不同的zbuffer，从结果来看，scanline zbuffer作为一种时间换空间的做法，在如今的内存越来越便宜的情况下显得表现不佳，但其思想值得借鉴。而hiz对绘制顺序比较敏感，结合了bvh后在有复杂遮挡关系的场景效率可以超过普通的zbuffer。

本次作业的收获一方面是在于对于这些算法的细节理解更加深刻，同时还对图像光栅化过程有了更加深刻的认识。在改进方向上，有以下改进方向：

- 借助多线程或者cuda进行并行化计算
- 优化代码架构，减少模块耦合

- 使用更加高级的着色方式