GAMES101 Report

范昊翀

commit:ef7fa4e

1 Lab 1

Lab 1 中主要内容是对三维空间中的点进行平移,旋转,投影操作,我通过补全了 **utils.rs** 中的函数实现了 **Lab 1** 中要求实现的效果。按照 Guideline 中要求填写矩阵,设置旋转轴为 $v = (1.0, 1.0, 1.0)$ 实现效果如下:

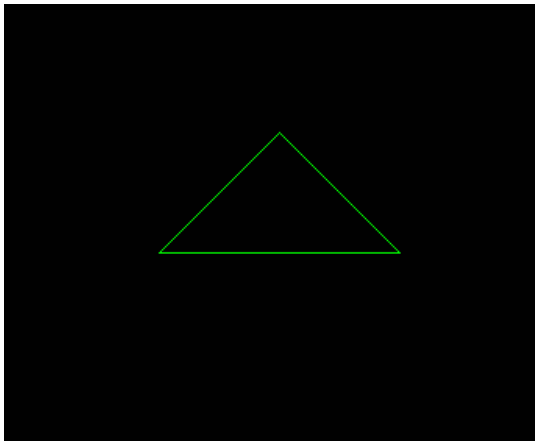

图 1: 未旋转的三角形

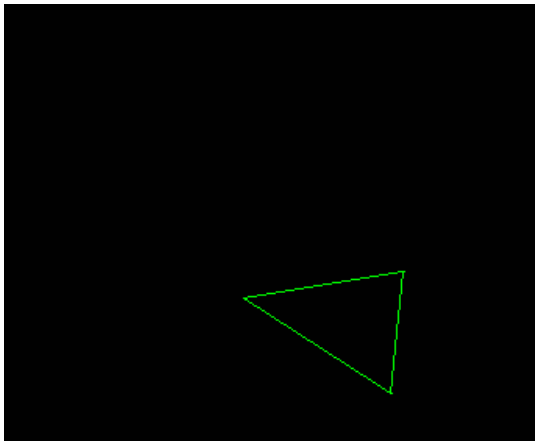

图 2: 顺时针旋转后的三角形

2 Lab 2

在 **Lab 2** 中，模型的位置保持不变，我们需要渲染三个带有深度与颜色信息的三角形，其基本思路是遍历屏幕上所有像素，如果这个像素的中心点在这个三角形内，则把这个像素的颜色设为这个三角形的颜色。当然，我们可以优化这个遍历的进程，只搜索三角形 xy 坐标范围内的所有像素，可以节省渲染的时间，虽然这在 **Lab 2** 中的表现并不明显，但后续在 **Lab 3** 中需要渲染上千个小三角形时，优化后的差异相当明显。

2.1 Z-buffering 算法

在平移旋转变换之后，我们把摄像机放在了原点，并且看向负 z 轴方向。为了模拟三角形之间由于深度产生的互相遮挡，我们对每一个像素的位置记录一个当前离摄像机最近的三角形（实际上，我们记录的是离摄像机最近的颜色）。

在我的实现中，我并没有改变 **clear** 函数中将 depth 初始化为 INF 的操作，因此我把 z 值映射到 depth 值的过程做了一些调整，每次发现 depth 变小，则更新它。并且，z（默认为负）越小，dep 越大，这样在不改变代码结构情况下正确实现了 Z-buffering 算法（尽管这么做有些丑陋，并不符合直觉）。

在实现了深度的记录之后，我们便可以渲染出第一张成果图：

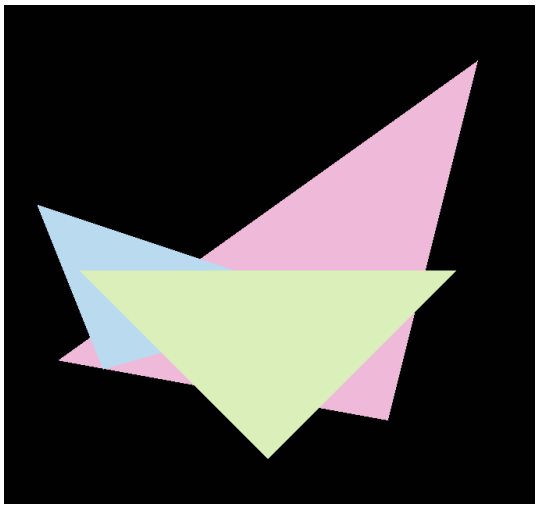

图 3: 无抗锯齿的渲染图

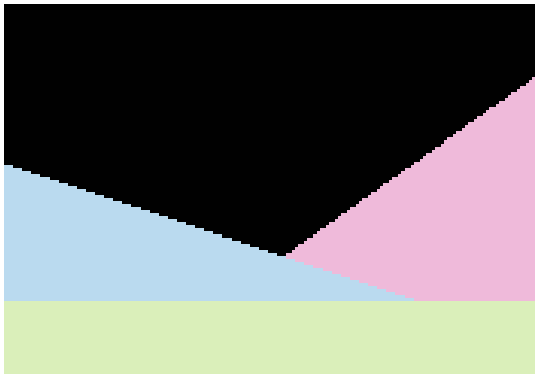

图 4: 局部的锯齿

由于三角形的边缘是斜线，正如 Guideline 中所说的那样，在采样的过程中，由于我们的采样步长是以像素为单位设定的，因此其边缘一定不会是平滑的，这便出现了如图 4 中所展现的锯齿状边缘。

2.2 MSAA 抗锯齿

因此，一个自然的思路是让采样的细度变得更加精细，这需要我们增加一个像素内采样点的个数，判断这些采样点是否在三角形内部，随后按照“在三角形内部的点”占总采样点的比例，确定这个像素中应该填入的颜色。

应用了以上较为 naive 的多采样策略之后，我们发现，渲染出的图片边缘出现了一些我们并不想要的黑边。这是因为在边缘处，有小部分采样点在离摄像机较近的三角形的内部，然而大部分采

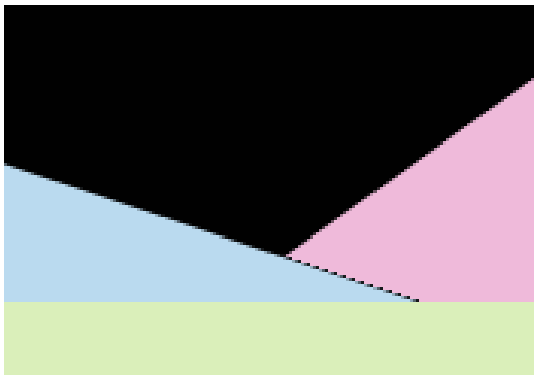

图 5: Naive 的多采样策略

样点落在了三角形外部，他们的颜色会被记为 $(0.0, 0.0, 0.0)$ 。此时，取平均后，结果会非常接近 $(0.0, 0.0, 0.0)$，在 RGB 中，这是接近于黑色的颜色。

因此，为了避免这个问题，我们需要判定每个子像素分别归属的三角形。我们通过记录每个子像素的深度和颜色来实现这一点。修复黑边后，局部效果如下：（写在这里：我在写报告的时候才突然发现，我的 MXAA 实现成了 SSAA，之前的 naive 方法是类似 MSAA 的思想，但是需要判断边缘）

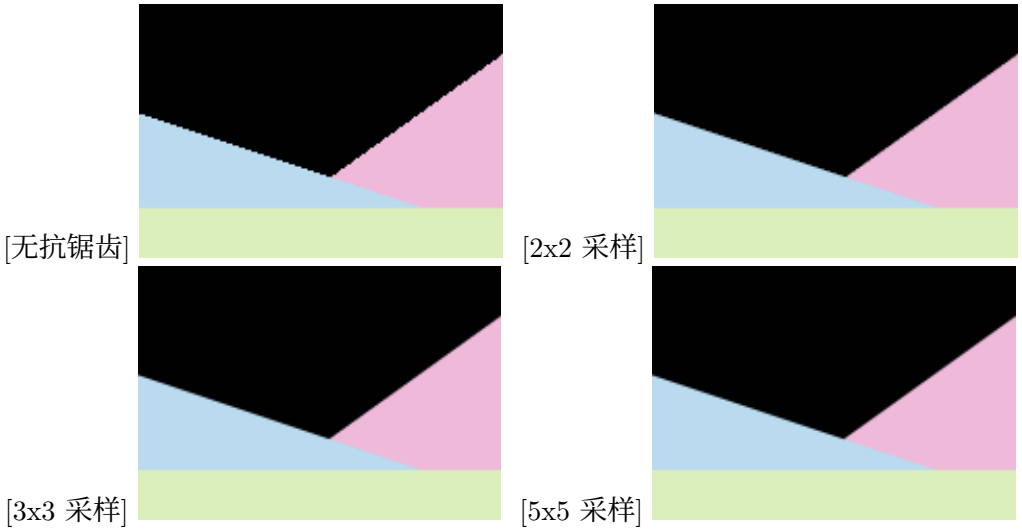

图 6: 不同采样细度下的效果对比

可以看出，在进行了 2x2 采样时，其观感就比无抗锯齿时改善了很多了，而在进一步提升采样点个数后，从人眼的角度来说，优化空间不是很大。

当然，更多的采样点意味着更慢的渲染速度，下表列出了各个采样细度下，渲染一个三角形所需要的时间：

	首次渲染用时 (ms)	后续每次渲染用时 (ms)
2x2 采样	319	337
3x3 采样	679	669
5x5 采样	1878	1819

表 1: 各个采样细度下渲染所需时间

通过数据，我们可以很容易地看出，渲染一次的时间与采样点个数成线性关系。

2.3 TAA 抗锯齿

另一种抗锯齿的思路是进行插值，对于边缘像素，我们通过随机偏移每次取样点，保证这一像素的颜色能够整合其周围的颜色信息。而对于内部像素，偏移采样并不会对其渲染造成什么影响。并且，由于每次渲染只进行一个点的采样，因此理论上其渲染耗时与无抗锯齿时不会有什么区别。我们只需要让它迭代几个回合，达到稳定状态即可。

这里为了确保达到稳态，进行了 100 次迭代（实际上在个位数次数的迭代后人眼就无法分辨了）。然而，从局部来看，锯齿效果仍然相对明显，或许需要把插值的系数调整的更大一些。总的来说，TAA

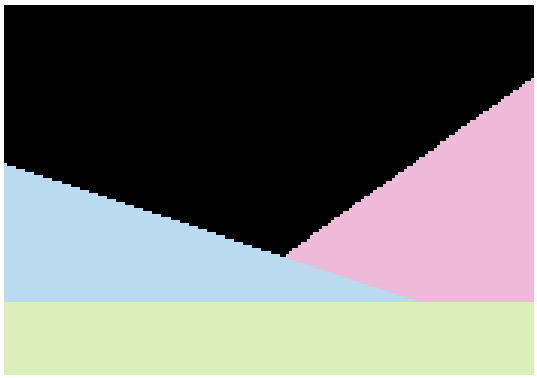

图 7: 插值系数为 0.2,100 次迭代后局部效果

的效果不如 MSAA 那么好，而且 MSAA 使用无锯齿四倍时空开销就能做出不错的结果，而 TAA 在同样也需要多次迭代的情况下，效果不够显著。

3 Lab 3

Lab 3 的指示给的不是非常明确, 但是跟随代码里的注释以及官方课件, 并借助强大的 **Copilot** 的力量，仍然是成功完成了任务要求。

为了验证我在 **Lab 1** 中三个投影矩阵的正确性，我首先渲染了一个不带有任何颜色，纹路和高光的 **牛牛**（一开始渲染出来的**牛牛**转了 180 度，我通过修正投影矩阵把**牛牛**重新摆正了），并调用 **Normal shader** 渲染了一个彩色**牛牛**:

图 8: 剪影牛牛

图 9: 彩色牛牛

在 **Phong shader** 和 **Texture shader** 中，**牛牛**的上色来源于三个部分：环境光，反射高光以及漫反射光，套用公式分别计算即可，两者唯一的区别是 **Texture shader** 的颜色并不固定，随在模型上的位置变化而不同，这取决于表面贴图的颜色。

图 10: 巧克力牛牛

图 11: 原装牛牛

再一次地，为了验证旋转矩阵的正确性，我调整了相机视角与位置，继续渲染原装**牛牛**:

图 12: 牛牛屁股

图 13: 牛牛侧颜

对于剩下的两个渲染模式，**Bump shader** 通过表面材质颜色的变化率，判断法向方向，从而在表面装出纹理的样子。**Displacement shader** 通过强行对顶点做位移，做出实际的表面材质。

图 14: 菌子牛牛

图 15: 文物牛牛

下面是两个失败品的展示，在渲染 **Phong shader** 时，我在反光时忘记除以光源离反射点的距离，因此反光效果变得过于强烈。另一个模型是只算了反射光，没有算其他的光线，因此图中只剩下了黑白两色。

图 16: 闪亮牛牛

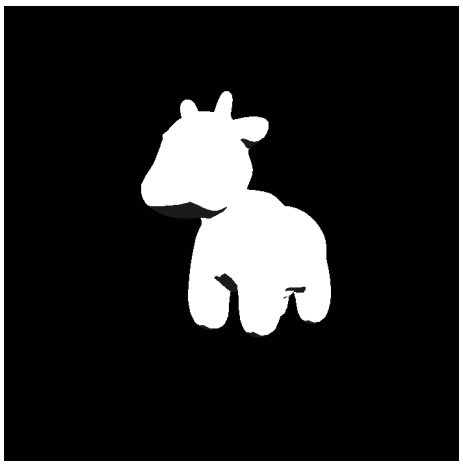

图 17: 后现代牛牛

此外，我尝试了 **Optional** 的内容，自己下载了网上的免费的人体模型，尝试渲染，但是在不同地方出现了若干 Runtime Error，考虑到时间有限，这次的 GAMES101 就先做到这里了。