学号	姓名	论文规范性（10）	问题分析与调研（30）	方案创新性（20）	实验结果分析与讨论（40）	结课论文总成绩（100）
21301129	刘美何	8	24	17	36	85

参考文献有些少

北京交通大学

计算机图形学期末论文

抗锯齿与纹理失真修复技术的研究与应用

Research and Application of Anti-aliasing and Texture Distortion Correction Techniques

学　　院：软件学院

专　　业：软件工程

学生姓名：刘美何

学　　号：21301129

北京交通大学

2024 年 6 月

摘要

在图形学中表示在光栅化三角形的时候，会发生走样现象。常见的走样有锯齿、摩尔纹等，走样影响图像的真实性和视觉体验，使用纹理来增强渲染图像的真实感和细节时会发生纹理失真的现象。

本文针对上述问题展开研究，分别采用多重采样抗锯齿技术（MSAA）和多级渐远纹理（Mipmap）技术进行改进和修复：

（1）使用多重采样抗锯齿技术(MSAA)，成功改善了几何物体边缘的锯齿现象。MSAA通过增加多个子采样点并根据覆盖度加权计算颜色值，有效减少了锯齿边缘，提升了图像边缘的平滑度和质量。

（2）使用多级渐远纹理(Mipmap)技术来修复因纹理映射而导致的失真现象。Mipmap技术通过预先生成不同分辨率的纹理贴图，并根据视角和距离动态选择适当的纹理级别，从而减少了远处纹理贴图的细节损失和失真，改善了图像整体的视觉一致性和真实感。

通过这些技术的研究与应用，加深了对抗锯齿和纹理失真问题的理解，在实际应用中有效改善了图像的显示效果和用户的视觉体验。

关键词：抗锯齿；纹理失真；反走样

目　　录

1 引言

图形学把屏幕抽象为一个二维数组，数组中的每一个元素是一个像素，屏幕是一个典型的光栅成像设备。光栅化是将物体投影在屏幕上的图形，依据像素打散，每一个像素中填充不同的颜色。在图形学中，三角形网格构成了物体表面结构，使用采样将三角形填充到屏幕像素中。三角形采样利用像素的中心对屏幕空间进行采样，判断每个像素的中心是否在三角形内部。走样，表示失去原来的样子。在图形学中表示在光栅化三角形的时候，得到的结果与原本变样了。常见的走样有锯齿、摩尔纹、车轮效应。 锯齿和摩尔纹都是在空间或者位置上采样产生的问题，车轮效应是在时间上采样产生的问题。

通过为每个顶点添加颜色可以增加图形的细节，如果想让图形看起来更真实，必须有足够多的顶点，指定足够多的颜色，会产生很多额外开销。纹理可以用来添加物体的细节，在一张图片上插入非常多的细节，可以让物体非常精细而不用指定额外的顶点。为了能够把纹理映射到三角形上，需要指定三角形的每个顶点各自对应纹理的哪个部分，每个顶点关联一个纹理坐标，用来标明从纹理图像的哪个部分采样。纹理坐标不依赖于分辨率，可以是任意浮点值，需要将纹理像素映射到纹理坐标。大的物体上应用低分辨率的纹理会这样就会导致一个纹理像素要对应多个屏幕像素，导致纹理失真走样。纹理过小并使用最近邻纹理过滤的话，会使得纹理渲染结果变得有颗粒感。但是如果纹理过大的话，也会有问题，近处纹理会产生锯齿，而远处纹理会产生摩尔纹。

其中锯齿现象和摩尔纹现象对视觉效果产生负面影响，为了削弱这种影响，本文对于抗锯齿和纹理失真修复的技术展开研究与应用。

2 相关工作介绍

抗锯齿的基本思想主要有两种，一是提高分辨率达到增加采样点的目的， 二是区域采样，将锯齿部分的像素看作有限的区域，对区域进行采样[1]。下面介绍几种三角形的抗锯齿手段。超级采样(SSAA)，超级采样将分辨率成倍的提高，在显示器上呈现时再进行缩放，该技术特点在于采样过程，超级采样的做法忽略了实际应用中的成本问题，使得该算法的性价比较低。虽然在原理上 SSAA 拥有最理想的精度，但是所需要绘制的图像千变万化，SSAA 在处理质量上的优势不具有竞争力。多重采样(MSAA)，多重采样是为了解决超级采样中消耗资源过大的问题设计出来的一种采样方式。主要目的是在减小硬件资源占用，降低成本的前提下，尽可能的改善图形质量。自适应抗锯齿(AAA)自适应抗锯齿是一种基于硬件支持的抗锯齿技术，该技术由 ATI 公司提出，作为 MSAA 的一种补充技术。AAA 可以处理 MASS 处理不了的 Alpha 材质，如纹理较小的一些物体，从而提高图像的整体质量。AAA 对 Alpha 材质进行多重采样或者超级采样，这样相对于全屏使用超级采样的计算量小，但可以得到非常优良的图像质量。覆盖采样抗锯齿（CSAA），该技术引入了一个全新的概念：一种表示覆盖的取样。CSAA 在像素的计算、深度和帧缓存值的处理方式等依然与多重采样保持一致，在此基础上增加了覆盖采样。通过存储一个二进制的 Coverage 值，从颜色值、深度值和缓存值分离出覆盖范围，对取样技术进一步的优化。具有针对性的提高覆盖采样的信息， 在计算消耗几乎为 0 的前提下，得到的抗锯齿效果却更加的出众。CSAA 可以看作是 MSAA 的进一步扩展，只增加了少量的性能消耗，该技术的应用前景广阔[2]。

3 方法描述

本次实验采用的抗锯齿方法是多重采样(MSAA)抗锯齿技术，多重采样采用的主要技术手段有以下两点：（1）边缘抗锯齿 多重采样只对 Z-Buffer 和 Stencil-Buffer 中的数据进行处理，即只对多边形的边缘采用抗锯齿技术。对于一个多边形的边缘像素，均有多个采样点，需要判断采样点是否均属于一个多边形的边缘，确定采样点属于哪个物体的边缘像素。多重采样通过对图形边缘采用抗锯齿代替超级采样中对图形全部进行抗锯齿的方法，处理的数据大幅减小，对资源的消耗降低。（2）用像素计算结果 对于一个 1*1 的像素块，以像素中心点的坐标作为该像素的坐标点，对其进行颜色值的计算，在该像素包含的区域内，采样点的颜色值与中心点坐标的颜色值保持一致，这一做法为了避免增加采样点而引入额外的计算量。虽然为了判别图形边缘处的像素而增加了采样点，但不会产生提高计算量的结果，这么一来对于硬件的负担大大减轻，抗锯齿的效率也得到了提高。 MSAA 开启时，分辨率 x4，即 100*100 的屏幕，要准备 200*200 的数据深度， 每个像素的 4 个采样点有各自的颜色值、 深度，一次把整一帧所有的三角形画完， 按照采样级别去做 Z 深度测试，最终像素的颜色是 4 个采样点的均值，通过 Bilinear resolve 得到 100*100 的图像。

针对纹理失真走样采用的修复方式是多级渐远纹理（Mipmap），多级渐远纹理是一系列的纹理图像，后一个纹理图像是前一个的二分之一。这些缩小版本存储在一个单独的纹理数组中，每个级别称为一个“Mipmap 级别”。多级渐远纹理的理念是：距观察者的距离超过一定的阈值，使用不同的多级渐远纹理，即最适合物体的距离的那个。多级渐远纹理有以下优点：1.减少走样和失真： 通过选择合适的 Mipmap 级别，可以有效减少放大时的失真和锯齿效应，同时在远距离或小尺寸时保持良好的纹理细节。2.节省性能和资源： 使用 Mipmaps 可以避免不必要的计算和内存消耗，因为只需加载和使用适当分辨率的纹理级别。

4 实验设置

本次实验在第二次实验的基础上，应用 MSAA 抗锯齿技术，减少实验中创建的原创三维模型几何物体的边缘处的锯齿状，提高视觉上的质量和平滑度。

在 OpenGL 中使用 MSAA，需要使用一个能在每个像素存储大于 1 个颜色值的颜色缓冲，因为多重采样需要为每个采样点都储存一个颜色，叫做多重采样缓冲(Multisample Buffer)。具体有以下两种实现方式。

第一种实现方式是窗口创建一个多重采样缓冲：

```
glfwWindowHint(GLFW SAMPLES, 4);
glEnable(GL MULTISAMPLE);
```

第二种实现方式是使用多重采样纹理附件：

```
glGenFramebuffers(1, &m_uiFrameBuffer);
glBindFramebuffer(GL_FRAMEBUFFER, m_uiFrameBuffer);

glGenTextures(1, &m_uiColorAttachment);
glBindTexture(GL_TEXTURE_2D_MULTISAMPLE, m_uiColorAttachment);
glTexImage2DMultisample(GL_TEXTURE_2D_MULTISAMPLE,4, GL_RGB, uiWidth, uiHeight, GL_TRUE);
glBindTexture(GL_TEXTURE_2D_MULTISAMPLE, 0);
glFramebufferTexture2D(GL_FRAMEBUFFER,GL_COLOR_ATTACHMENT0,GL_TEXTURE_2D_MULTISAMPLE, m_uiColorAttachment, 0);
glGenRenderbuffers(1, &m_uiFrameBufferObject);
glBindRenderbuffer(GL_RENDERBUFFER, m_uiFrameBufferObject);
glRenderbufferStorageMultisample(GL_RENDERBUFFER,4,GL_DEPTH24_STENCIL8, uiWidth, uiHeight);
glBindRenderbuffer(GL_RENDERBUFFER, 0);
glFramebufferRenderbuffer(GL_FRAMEBUFFER,GL_DEPTH_STENCIL_ATTACHMENT, GL_RENDERBUFFER, m_uiFrameBufferObject);
```

还原多重采样图像

```
glBindFramebuffer(GL_READ_FRAMEBUFFER, m_spMSAAInfo->Source);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, m_spMSAAInfo->Target);
glBlitFramebuffer(0,0,m_spMSAAInfo->SourceWidth, m_spMSAAInfo->SourceHeight, 0,0,m_spMSAAInfo->TargetWidth,m_spMSAAInfo->TargetHeight,GL_COLOR_BUFFER_BIT, GL_NEAREST);
```

在 OpenGL 中使用 Mipmap，包括下面两个步骤：1.加载纹理时生成 Mipmaps 2.修改纹理加载函数。对实验二中涉及纹理数据绑定的函数进行修改：

```
void MeshPainter::bindObjectAndData(TriMesh* mesh, openGLObject& object, const std::string& texture_image, const std::string& vshader, const std::string& fshader) {
    // 初始化各种对象

    std::vector<glm::vec3> points = mesh->getPoints();
    std::vector<glm::vec3> normals = mesh->getNormals();
    std::vector<glm::vec3> colors = mesh->getColors();
    std::vector<glm::vec2> textures = mesh->getTextures();

    // 创建顶点数组对象
#ifdef __APPLE__      // for MacOS
    glGenVertexArraysAPPLE(1, &object.vao);         // 分配 1 个顶点数组对象
    glBindVertexArrayAPPLE(object.vao);         // 绑定顶点数组对象
#else                 // for Windows
    glGenVertexArrays(1, &object.vao);      // 分配 1 个顶点数组对象
    glBindVertexArray(object.vao);      // 绑定顶点数组对象
#endif

    // 创建并初始化顶点缓存对象
    glGenBuffers(1, &object.vbo);
    glBindBuffer(GL_ARRAY_BUFFER, object.vbo);
    glBufferData(GL_ARRAY_BUFFER,
        points.size() * sizeof(glm::vec3) +
        normals.size() * sizeof(glm::vec3) +
        colors.size() * sizeof(glm::vec3) +
        textures.size() * sizeof(glm::vec2),
        nullptr, GL_STATIC_DRAW);

    // 绑定顶点数据
    glBufferSubData(GL_ARRAY_BUFFER, 0, points.size() * sizeof(glm::vec3), points.data());
    // 绑定颜色数据
```

```
        glBufferSubData(GL_ARRAY_BUFFER, points.size() * sizeof(glm::vec3), colors.size() * sizeof(glm::vec3), colors.data());
        // 绑定法向量数据
        glBufferSubData(GL_ARRAY_BUFFER, (points.size() + colors.size()) * sizeof(glm::vec3), normals.size() * sizeof(glm::vec3), normals.data());
        // 绑定纹理数据
        glBufferSubData(GL_ARRAY_BUFFER, (points.size() + normals.size() + colors.size()) * sizeof(glm::vec3), textures.size() * sizeof(glm::vec2), textures.data());

        object.vshader = vshader;
        object.fshader = fshader;
        object.program = InitShader(object.vshader.c_str(), object.fshader.c_str());

        // 将顶点传入着色器
        object.pLocation = glGetAttribLocation(object.program, "vPosition");
        glEnableVertexAttribArray(object.pLocation);
        glVertexAttribPointer(object.pLocation, 3, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0));

        // 将颜色传入着色器
        object.cLocation = glGetAttribLocation(object.program, "vColor");
        glEnableVertexAttribArray(object.cLocation);
        glVertexAttribPointer(object.cLocation, 3, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(points.size() * sizeof(glm::vec3)));

        // 将法向量传入着色器
        object.nLocation = glGetAttribLocation(object.program, "vNormal");
        glEnableVertexAttribArray(object.nLocation);
        glVertexAttribPointer(object.nLocation, 3,
            GL_FLOAT, GL_FALSE, 0,
            BUFFER_OFFSET((points.size() + colors.size()) * sizeof(glm::vec3)));

        // 将纹理坐标传入着色器
        object.tLocation = glGetAttribLocation(object.program, "vTexture");
```

```
        glEnableVertexAttribArray(object.tLocation);
        glVertexAttribPointer(object.tLocation, 2,
            GL_FLOAT, GL_FALSE, 0,
            BUFFER_OFFSET((points.size() + normals.size() + colors.size()) *
sizeof(glm::vec3)));

        // 获得矩阵位置
        object.modelLocation = glGetUniformLocation(object.program, "model");
        object.viewLocation = glGetUniformLocation(object.program, "view");
        object.projectionLocation = glGetUniformLocation(object.program, "projection");

        object.shadowLocation = glGetUniformLocation(object.program, "isShadow");

        // 读取纹理图片路径
        object.texture_image = texture_image;

        // 创建纹理的缓存对象
        glGenTextures(1, &object.texture);
        glBindTexture(GL_TEXTURE_2D, object.texture);

        // 调用 stb_image 生成纹理
        int width, height, nrChannels;
        stbi_set_flip_vertically_on_load(true); // 如果需要，翻转图片
        unsigned char *data = stbi_load(object.texture_image.c_str(), &width, &height,
&nrChannels, 0);
        if (data)
        {
            GLenum format;
            if (nrChannels == 1)
                format = GL_RED;
            else if (nrChannels == 3)
                format = GL_RGB;
            else if (nrChannels == 4)
                format = GL_RGBA;
```

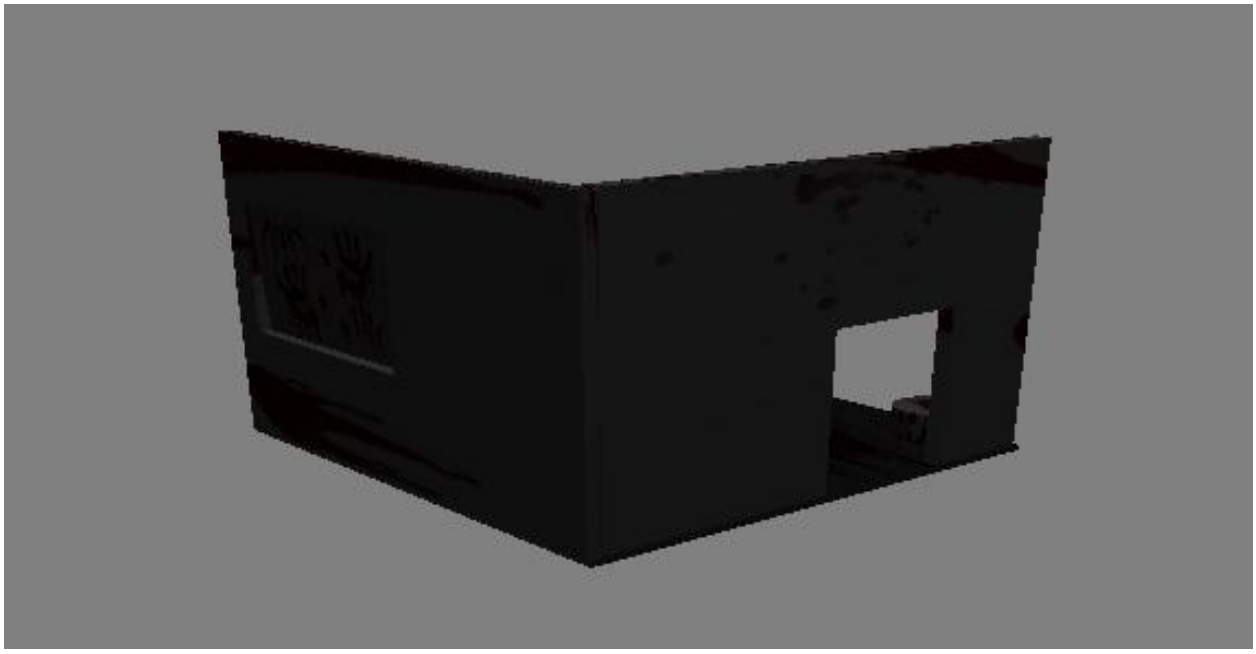
```
            glTexImage2D(GL_TEXTURE_2D, 0, format, width, height, 0, format,
GL_UNSIGNED_BYTE, data);
            glGenerateMipmap(GL_TEXTURE_2D); // 生成 Mipmaps
        }
        else
        {
            std::cerr << "Failed to load texture: " << object.texture_image << std::endl;
        }
        stbi_image_free(data);

        // 设置纹理参数
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_LINEAR);

        // Clean up
        glUseProgram(0);
    #ifdef __APPLE__
        glBindVertexArrayAPPLE(0);
    #else
        glBindVertexArray(0);
    #endif
    }
```

5 实验结果与分析

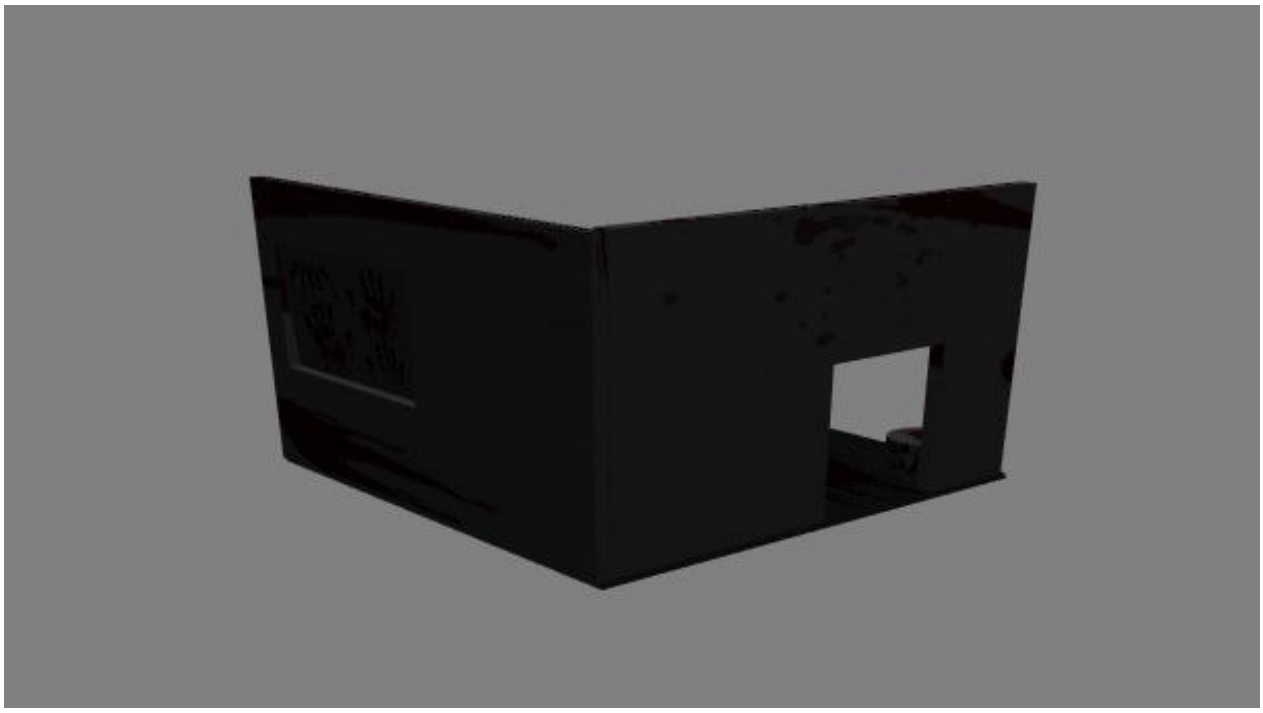
1.MSAA 抗锯齿

(1)是否应用抗锯齿技术结果对比

应用 MSAA 抗锯齿技术前：场景三鬼屋边缘可见明显锯齿

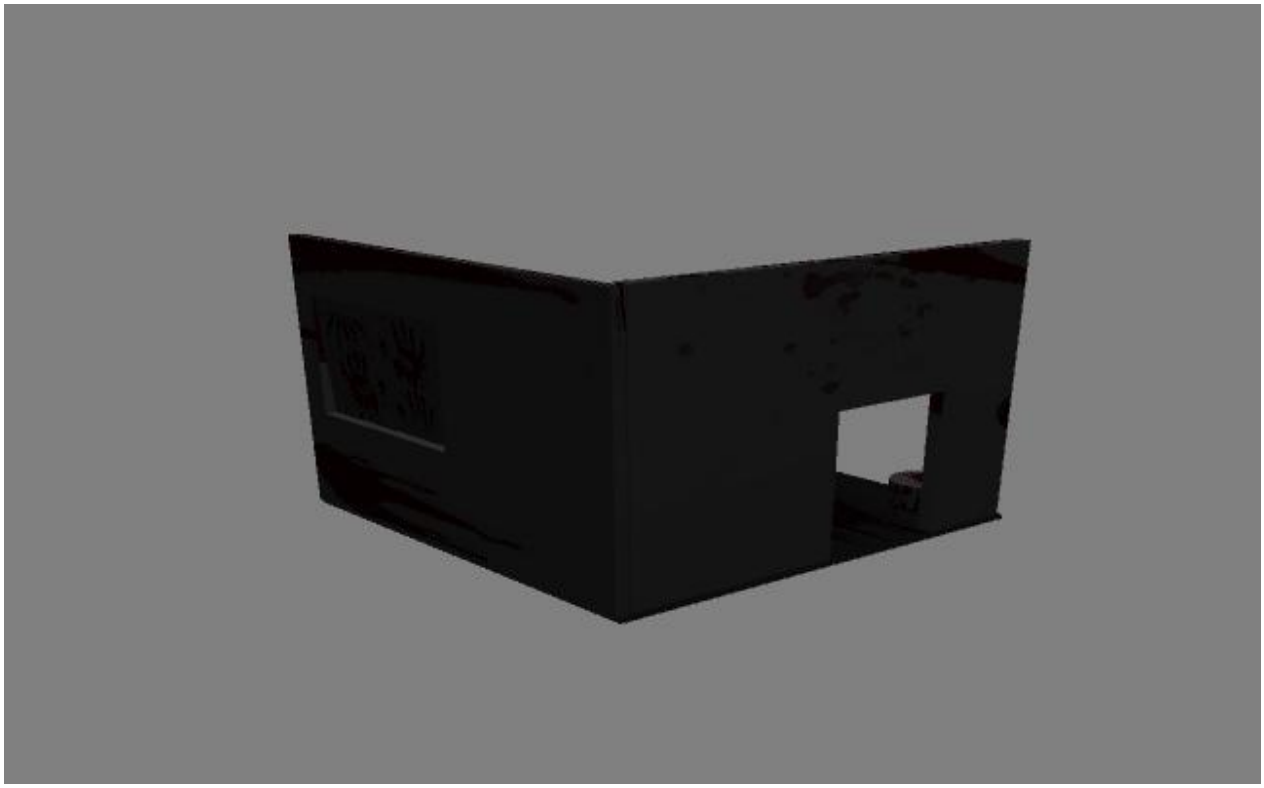


应用 MSAA 抗锯齿技术（采样点设置为 8）后：鬼屋边缘锯齿得到了有效平滑

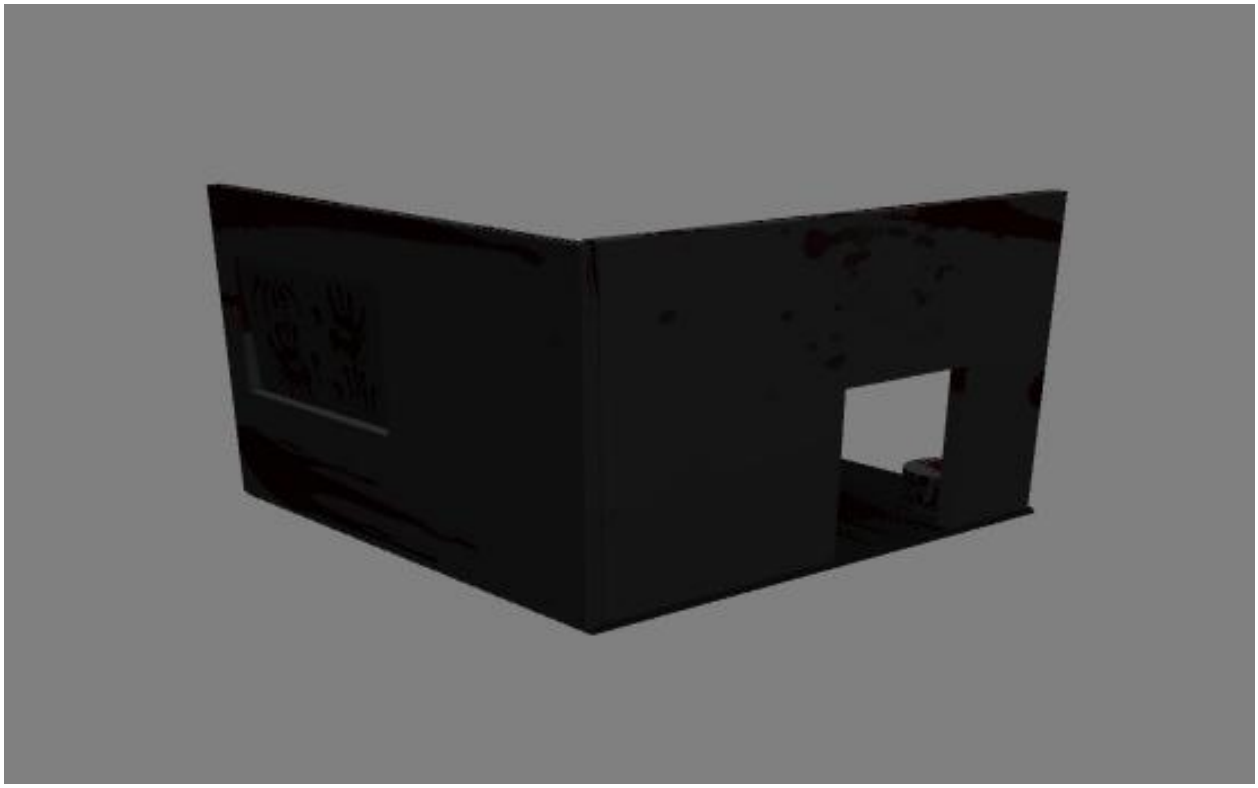


(2)不同数量采样点设置结果对比

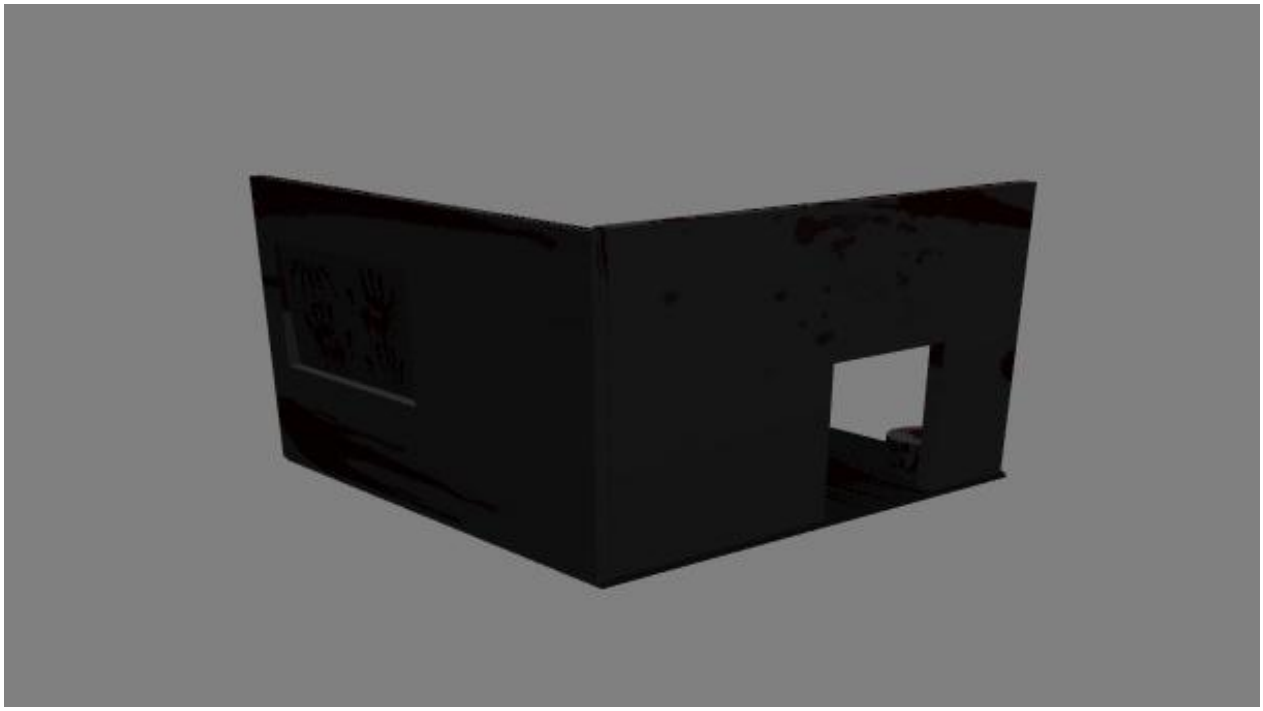
应用 MSAA 抗锯齿技术（采样点设置为 2）



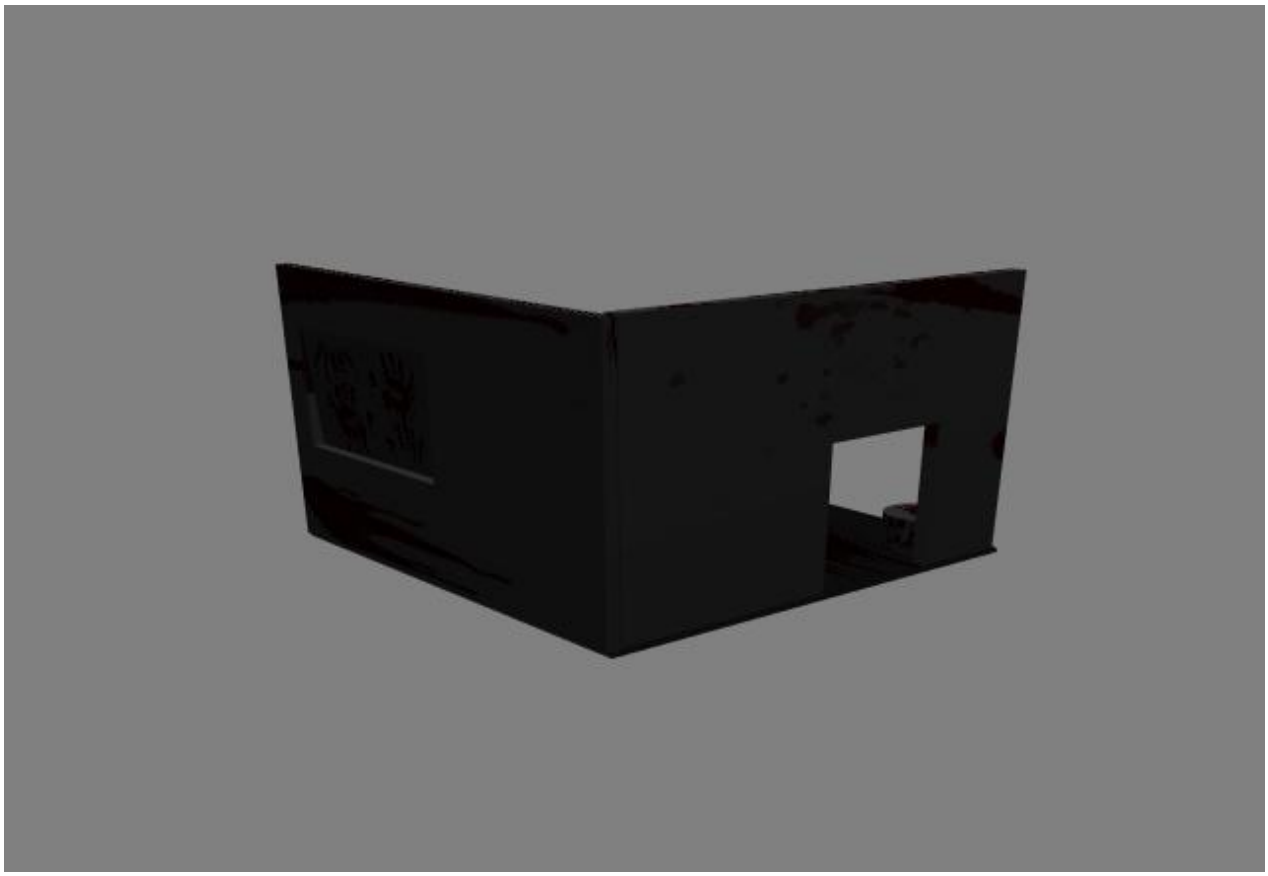
应用 MSAA 抗锯齿技术（采样点设置为 4）



应用 MSAA 抗锯齿技术（采样点设置为 8）



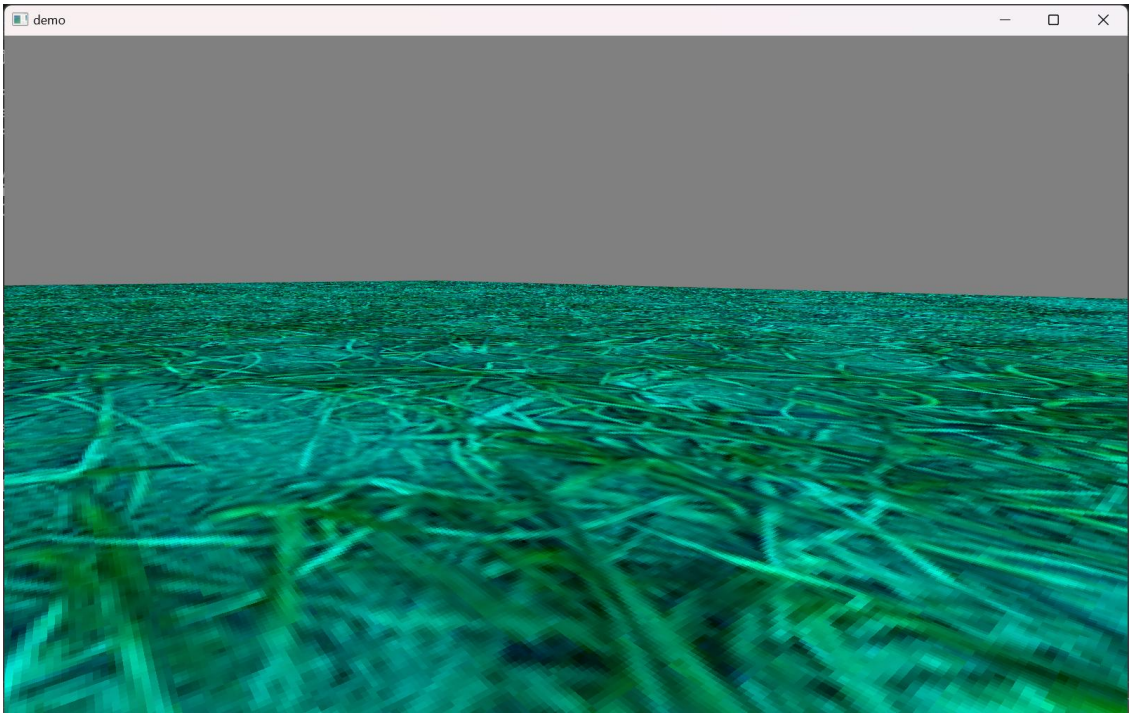
应用 MSAA 抗锯齿技术（采样点设置为 16）



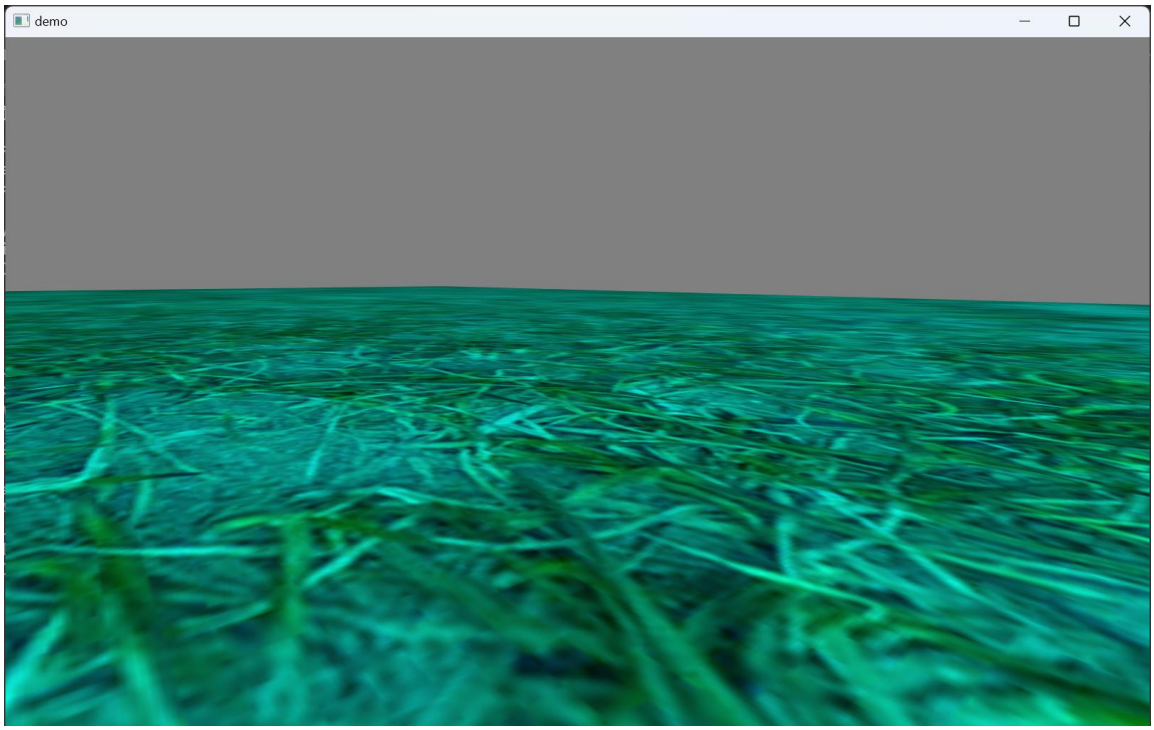
观察实验结果发现，从不使用 MSAA 的抗锯齿技术，即采样点为 1，到使用 2、4、8、16，随着采样点的增加，鬼屋边缘平滑度上升，视觉效果更加流畅，更逼近真实的物理世界的效果。

2.多级渐远纹理（Mipmap）

应用 MSAA 抗锯齿技术前：场景一中的草地近处纹理产生了锯齿，而远处纹理产生了摩尔纹



应用 MSAA 抗锯齿技术后：近处锯齿和远处摩尔纹都得到了改善



6 结论

计算机图形学处理的是数字图像，数字图像由模拟图像转化而来，以像素为基本元素。采样就是用离散的像素模拟原本连续的数据。三角形光栅化的过程中，涉及到三角形采样，利用像素模拟三角形的形状，在纹理采样的过程中，对于一张纹理，即一个二维图片，也是有很多个像素组成的，在小纹理大像素的情形中，一个纹理像素对应多个屏幕像素，导致纹理失真走样；在大纹理小像素的情形中，当纹理过大时，每个屏幕上的像素实际上代表了一个相对较大的区域，而这个区域内可能存在多个纹理细节。然而，由于每个像素只能显示一种颜色，这就导致了近处的纹理细节无法被完全表达出来，因此在渲染时会出现锯齿状的效果。而由于透视投影会产生近大远小的效果，远处纹理会包含更多的纹理细节，但是，这些丰富的细节却只能用一个像素来呈现，采样率更低，显然会使得远处纹理细节会撕裂的更厉害，所以就会产生摩尔纹。上述的锯齿的出现和纹理走样中的大纹理小像素都是来自于采样率的不足，改善采样率不足最直观的解决办法就是超采样，提高采样频率，超采样虽然能解决锯齿和摩尔纹问题，但计算成本是太高。在抗锯齿中，MSAA 通过增加采样点的方式用子采样点决定像素的遮盖度，根据覆盖度来加权计算所有子采样点的颜色值。在解决纹理失真中，由于一个采样点不能代表纹理细节，可以将纹理区域内的所有像素进行求和求平均来代替当前像素的纹理。Mipmap 算法解决的就是快速查询纹理平均像素值的问题。可以发现两种方法都是为了解决采样率不足的问题进行平均操作，具有相同的底层原理，对于以后解决图形学中其他走样问题的解决提供了很好的思路。

参考文献

[1]张宇剑.图形处理器三角形光栅化和抗锯齿算法研究与实现[D].西安电子科技大学,2018.

[2] Wu X. An efficient antialiasing technique[J]. 1991, 25(4):143-152.