基于几何指导的手绘风格三维场景NPR渲染——实验报告

本实验设计并实现了一套基于几何缓冲区（G-Buffer）的非真实感渲染（NPR）系统。不同于传统的二维图像滤镜，本系统在光栅化阶段提取三维场景的深度图（Depth Map）与法线图（Normal Map），以此作为几何指导信息，驱动素描、水彩、卡通及油画四种风格的生成。此外，系统引入了**Reinhard色彩迁移算法**与**基于深度的背景合成技术**，实现了从单一模型渲染到完整艺术场景构建的流程闭环。

NRP

非真实感渲染（Non-Photorealistic Rendering, NPR）旨在模拟艺术媒介的表现形式。传统基于图像处理（Image-space）的方法往往忽略了场景的三维结构，导致渲染结果缺乏体积感或轮廓断裂。本实验旨在通过提取三维模型的几何特征（深度与法线），构建一个"几何指导"的渲染管线，在保持低计算成本的同时，实现具有精确三维结构特征的手绘风格渲染。

系统架构和方法

本系统采用延迟着色（Deferred Shading）的思想，将渲染流程分为几何阶段、风格化阶段风格化迁移阶段。

1. 几何缓冲区的生成

   系统首先通过基于扫描线的光栅化算法（Rasterization）处理 `.obj` 格式的三维网格数据。在此过程中，不进行光照计算，而是生成两张核心特征图：

   - **深度图 (Depth Map):** 记录每个像素点到视平面的距离，用于后续的笔触强度控制和阴影模拟。

   - **法线图 (Normal Map):** 记录每个像素点的表面朝向向量 (n_x, n_y, n_z)，用于精确捕捉物体表面的曲率变化。
   - **实现细节:** 使用 `Numba` 加速的重心坐标插值算法计算片元属性。

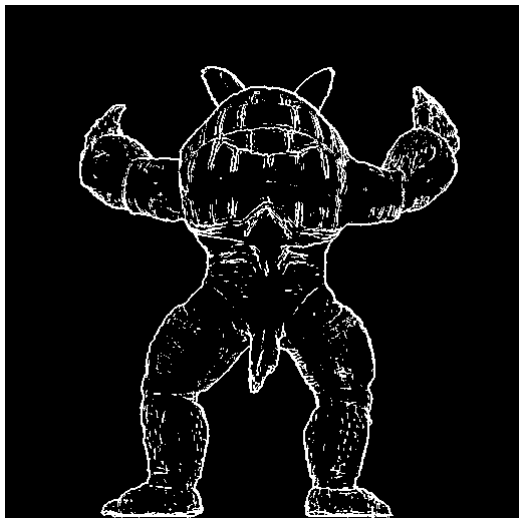
```
# 基于光栅化的几何缓冲区生成核心算法
@jit(nopython=True)
def rasterize_triangles(vertices, faces, image_size):
    # ... (省略初始化代码)
```

```
    # 遍历所有三角形面片
    for i in range(len(faces)):
        # ... (省略顶点投影和包围盒计算)

        # 遍历包围盒内的像素
        for y in range(min_y, max_y + 1):
            for x in range(min_x, max_x + 1):
                # 计算重心坐标 (Barycentric Coordinates)
                # ... (省略重心坐标计算公式)

                # 如果点在三角形内
                if w0 >= 0 and w1 >= 0 and w2 >= 0:
                    # 插值计算深度值 z
                    z = w0 * v0[2] + w1 * v1[2] + w2 * v2[2]

                    # Z-Buffer 测试：保留更近的片元
                    if z > depth_map[y, x] or depth_map[y, x] == 0:
                        depth_map[y, x] = z
                        # 将面法线写入法线缓冲区
                        normal_map[y, x, :] = face_normals[i, :]

    return depth_map, normal_map
```

2. 几何指导的轮廓提取

为了模拟手绘风格中的硬边缘（轮廓线），本实验摒弃了仅基于颜色梯度的Canny算子，改为基于几何信息的边缘检测：

$$Edge = \alpha \cdot Sobel(Depth) + \beta \cdot Sobel(Normal)$$



- 对深度图应用 Sobel 算子提取**深度不连续边界**（物体轮廓）。
- 对法线图应用 Sobel 算子提取**法线不连续边界**（物体内部折痕）。
- **代码实现:**

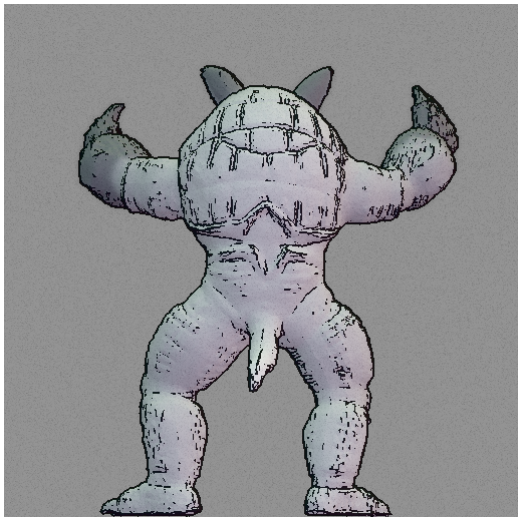
```
    # 基于深度与法线的几何边缘检测
    def detect_edges(self, depth_map, normal_map, threshold=0.1):
        # 1. 深度图边缘检测 (捕捉物体轮廓)
        depth_edges = cv2.Sobel(depth_map, cv2.CV_32F, 1, 0, ksize=3)
        depth_edges += cv2.Sobel(depth_map, cv2.CV_32F, 0, 1, ksize=3)
```

```
    # 2. 法线图边缘检测 (捕捉物体内部折痕)
    normal_x = cv2.Sobel(normal_map[:, :, 0], cv2.CV_32F, 1, 0, ksize=3)
    normal_y = cv2.Sobel(normal_map[:, :, 1], cv2.CV_32F, 0, 1, ksize=3)
    normal_z = cv2.Sobel(normal_map[:, :, 2], cv2.CV_32F, 1, 0, ksize=3)
    # 计算法线梯度的模长
    normal_edges = np.sqrt(normal_x ** 2 + normal_y ** 2 + normal_z **
2)

    # 3. 融合几何信息
    edges = depth_edges + normal_edges * 0.5

    # 二值化输出
    edge_map = (edges > threshold).astype(np.uint8)
    return edge_map
```

3. 风格化渲染

   系统利用提取的 G-Buffer 数据实现了四种风格算法：

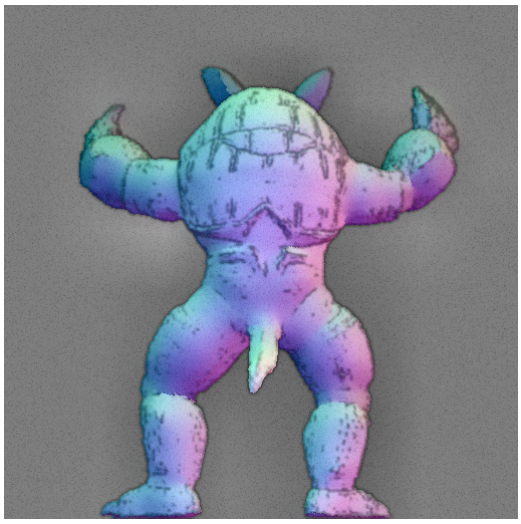
   1. **素描风格 (Sketch):**
      - **纹理:** 使用高斯噪声模拟纸张颗粒感。
      - **几何指导:** 利用 **深度图** 控制阴影涂抹（Shading），深度值大的区域笔触更重；利用 **边缘图** 勾勒轮廓。
      - **合成:** `Result = (PaperNoise * EdgeMap * DepthShading)`。



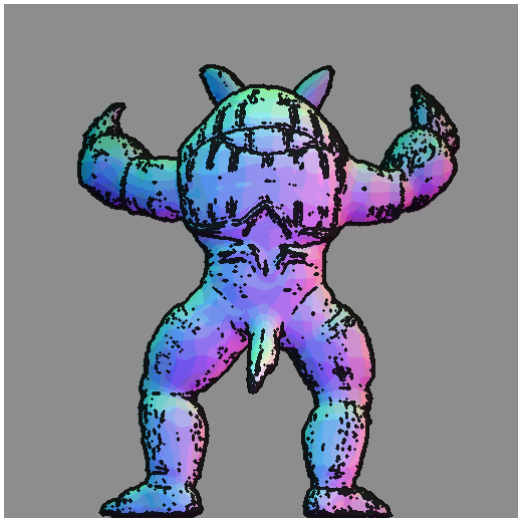
```
# 几何指导的素描风格合成
def apply_sketch_style(self, color_map, edge_map, depth_map,
strength=0.9):
    # ... (省略噪声生成部分)

    # 1. 利用深度图生成阴影 (Shading via Depth)
    # 只有深度较大（较远/较暗）的区域才加重阴影
    shading = (depth_map * 255).astype(np.uint8)
    _, dark_areas = cv2.threshold(shading, 100, 255,
cv2.THRESH_BINARY_INV)

    # 2. 几何边缘叠加
    # edge_map: 1是线，0是背景 -> 转换为乘法掩码
    edges = (1 - edge_map) * 255
```

```
    # 3. 最终合成：底纹 x 轮廓 x 深度阴影
    final_sketch = cv2.multiply(base_sketch / 255.0, edges / 255.0)

    # 再次叠加深度阴影增强体积感
    shadow_factor = 1.0 - (dark_areas / 255.0 * 0.3)
    final_sketch = final_sketch * shadow_factor

    return result_rgb
```

2. **水彩风格 (Watercolor):**
    - **抽象化:** 使用均值漂移滤波（Mean Shift Filtering）对色彩进行平滑，消除高频纹理，形成色块感。
    - **几何指导:** 利用边缘图模拟"湿边效应"（Wet Edges），即在轮廓处叠加加深的半透明层。
    - **湍流模拟:** 叠加低频噪声模拟水彩颜料在纸张上的不均匀扩散。

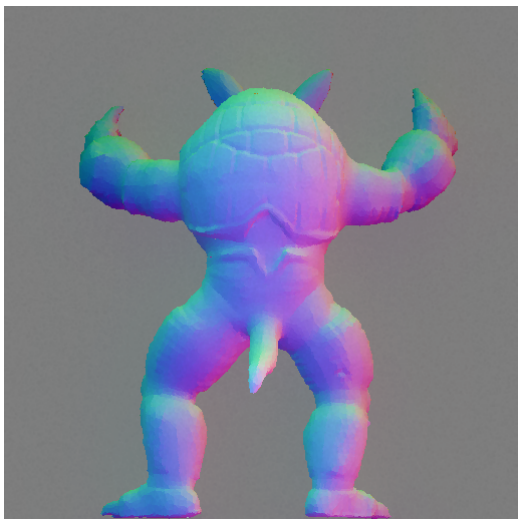


3. **卡通风格 (Cartoon/Cel-Shading):**
    - **双边滤波:** 在保持边缘清晰的同时平滑颜色。
    - **色阶量化:** 将连续的RGB色彩空间离散化（Quantization），例如将 256 色阶降维至 8 色阶，形成明显的明暗交界线。
    - **轮廓增强:** 叠加膨胀处理后的加粗边缘图。



4. **油画风格 (Oil Painting):**

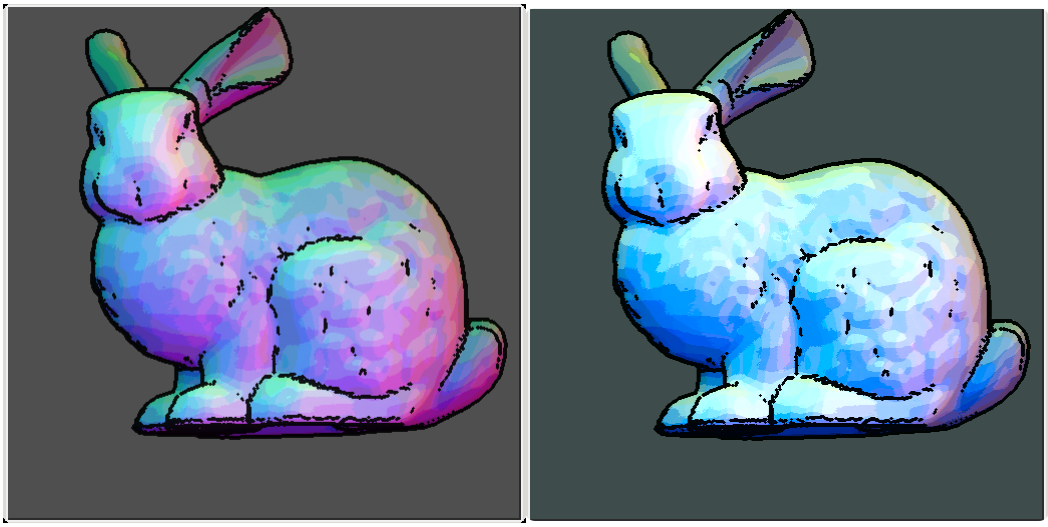
- **笔触模拟:** 采用保边滤波器（Edge Preserving Filter）产生类似油画笔触的涂抹感。
- **画布纹理:** 叠加生成的画布纹理图层，并提升HSV空间中的饱和度分量。



4. 基于统计匹配的色彩迁移

   为了赋予渲染结果特定的艺术氛围（如模拟梵高的星空色调），本实验实现了基于 **Reinhard** 统计匹配的全局色彩迁移算法。该算法不需要复杂的神经网络训练，而是利用色彩空间的统计特征进行快速映射。

   - **色彩空间转换:** 将源图像（渲染结果）和参考图像（艺术画作）从 RGB 空间转换到 **LAB 色彩空间。**LAB 空间将亮度（L）与色度信息（A, B）分离，且各通道之间的相关性极低，适合独立处理。
   - **统计特征映射:** 分别计算源图像 S 和参考图像 R 在各通道上的均值（μ）和标准差（σ）。通过线性变换，强行将源图像的色彩分布拉伸至与参考图像一致：$C_{out} = \frac{C_{in}-\mu_S}{\sigma_S} \cdot \sigma_R + \mu_R$



```
def reinhard_color_transfer(source, reference):
    """
    Reinhard色彩迁移算法核心实现
    Args:
        source: 原始NPR渲染结果 (RGB)
        reference: 目标艺术风格参考图 (RGB)
    """
    # 1. 转换到 LAB 空间 (解耦亮度和色彩)
    source_lab = cv2.cvtColor(source, cv2.COLOR_RGB2LAB).astype(np.float32)
    reference_lab = cv2.cvtColor(reference,
cv2.COLOR_RGB2LAB).astype(np.float32)
```

```
    # 2. 计算统计量 (均值和标准差)
    mean_src = source_lab.mean(axis=(0, 1))
    std_src = source_lab.std(axis=(0, 1))
    mean_ref = reference_lab.mean(axis=(0, 1))
    std_ref = reference_lab.std(axis=(0, 1))

    # 3. 应用线性变换 (色彩分布对齐)
    # 减去源均值 -> 缩放标准差比率 -> 加上参考均值
    result_lab = (source_lab - mean_src) * (std_ref / std_src) + mean_ref

    # 4. 裁剪并转回 RGB
    result_lab = np.clip(result_lab, 0, 255).astype(np.uint8)
    result = cv2.cvtColor(result_lab, cv2.COLOR_LAB2RGB)

    return result
```

5. 基于深度遮盖的背景合成

   为了将孤立的3D模型融入完整的图像背景中，本系统利用光栅化阶段生成的深度图作为完美的 Alpha 通道进行图像合成。

   - **精确遮罩生成:** 不同于基于色度键（绿幕）的抠图技术，本系统直接拥有几何真值。通过深度阈值检测，可以生成像素级精确的前景遮罩（Mask）。
   - **无伪影合成:** 由于遮罩直接源自几何投影，能够彻底避免传统图像处理中常见的边缘溢色（Color Spill）问题。

   算法逻辑：

   1. 检测深度图中的非零区域作为前景区域，生成二值化 Alpha 通道 $\alpha \in \{0, 1\}$。
   2. 利用线性插值公式进行混合：$I_{final} = I_{foreground} \cdot \alpha + I_{background} \cdot (1 - \alpha)$

```
def composite_with_background(foreground, depth_map, background):
    """
    基于几何深度的背景合成
    """
    h, w = depth_map.shape
    # 调整背景尺寸以匹配前景
    bg_resized = cv2.resize(background, (w, h))

    # 1. 生成 Alpha 遮罩
    # 利用光栅化产生的深度图，depth > 0.001 即为确定的模型区域
    # 这种方式比基于颜色的抠图更精确，无边缘杂色
```

```
    alpha = (depth_map > 0.001).astype(np.float32)

    # 扩展维度以匹配 RGB 通道 (H, W, 1)
    alpha = np.expand_dims(alpha, axis=2)

    # 2. Alpha 混合
    # 前景 * Alpha + 背景 * (1 - Alpha)
    result = foreground * alpha + bg_resized * (1 - alpha)

    return result.astype(np.uint8)
```

实验总结

本系统成功构建了一个完整的 NPR 渲染管线。通过结合底层的**几何光栅化**（生成深度/法线）与顶层的**图像处理算法**（Reinhard 色彩迁移、风格化滤镜），从而实现了从风格迁移的手绘风格三维场景NPR渲染。实验结果表明，几何指导信息对于保持物体结构、正确施加阴影以及精确的背景合成至关重要。