

图形学大作业实验报告

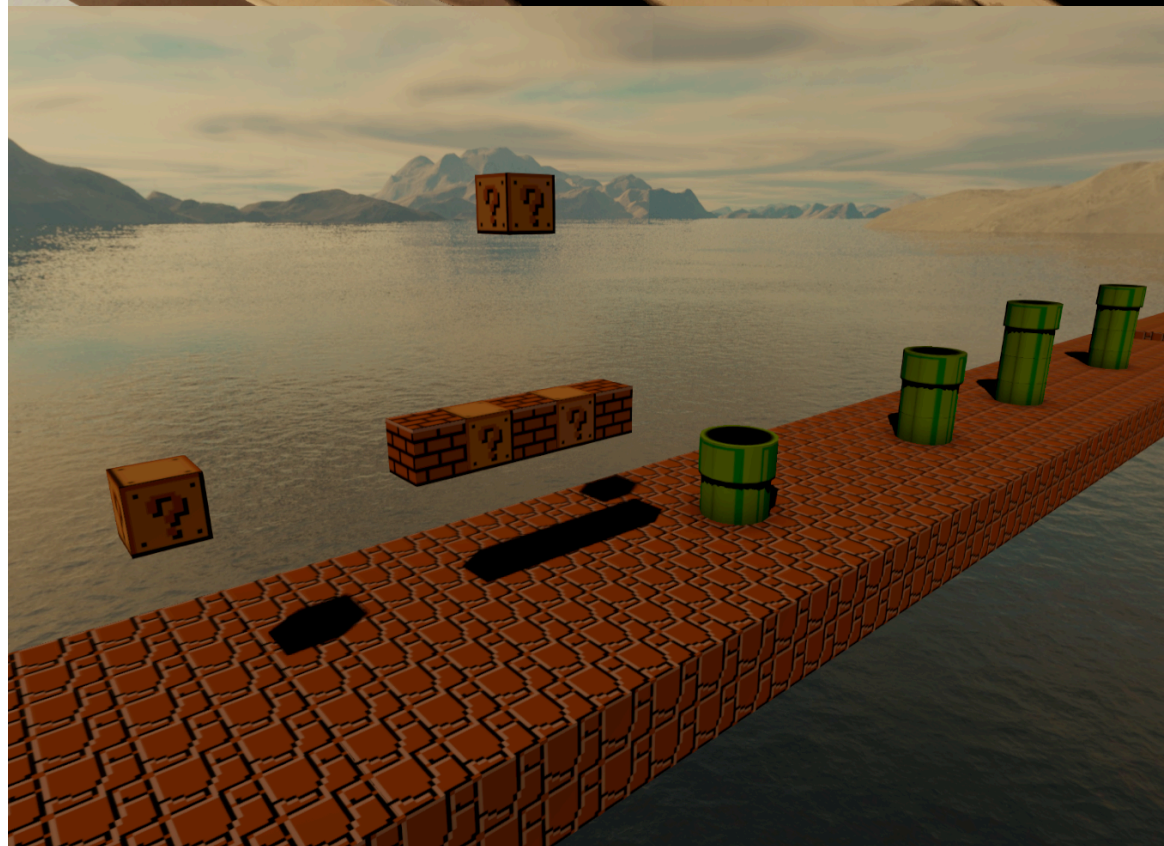
张同学

最终效果

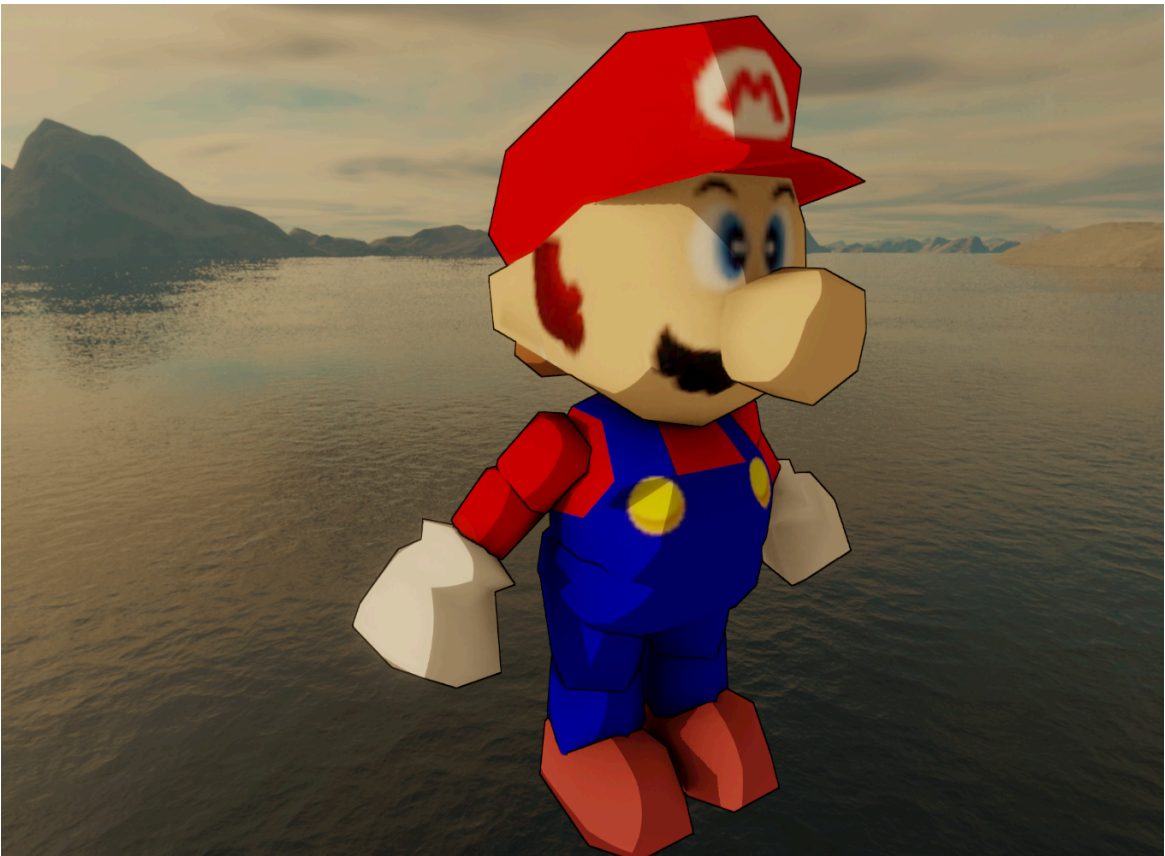
整体效果

处理前：

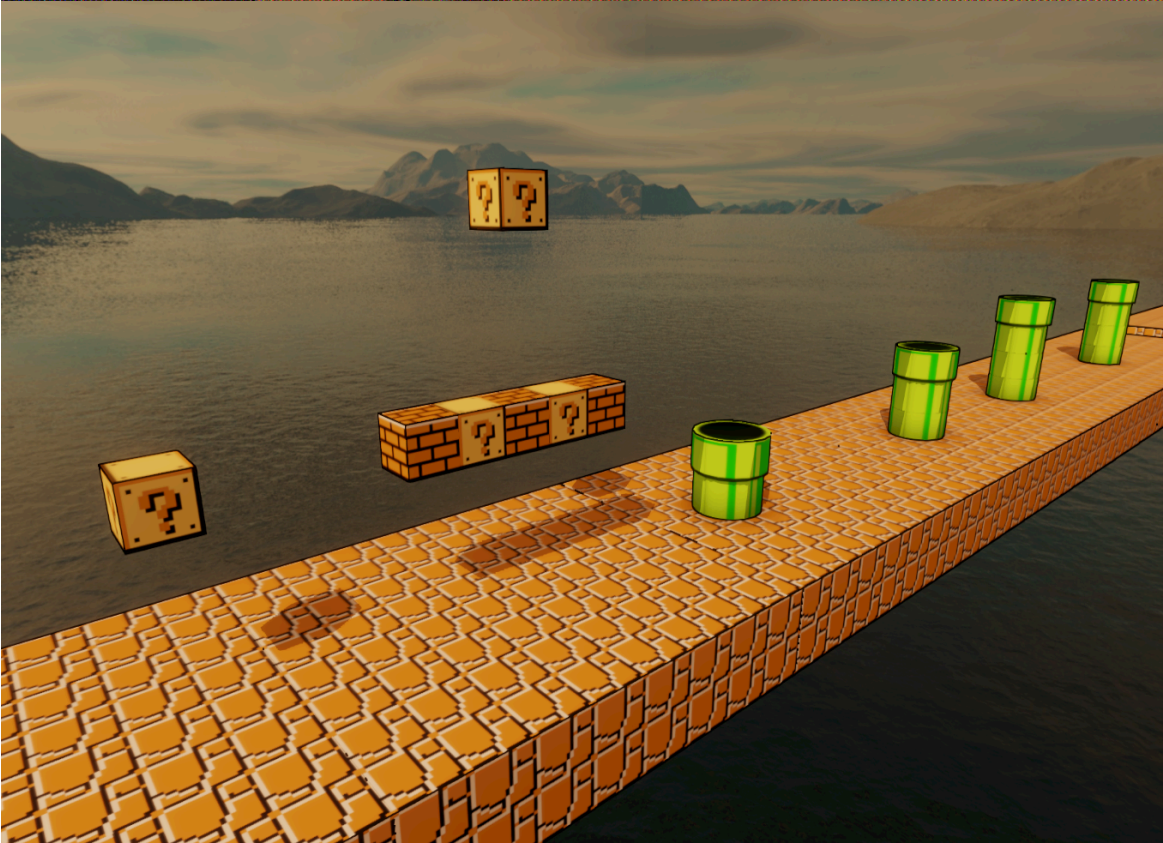


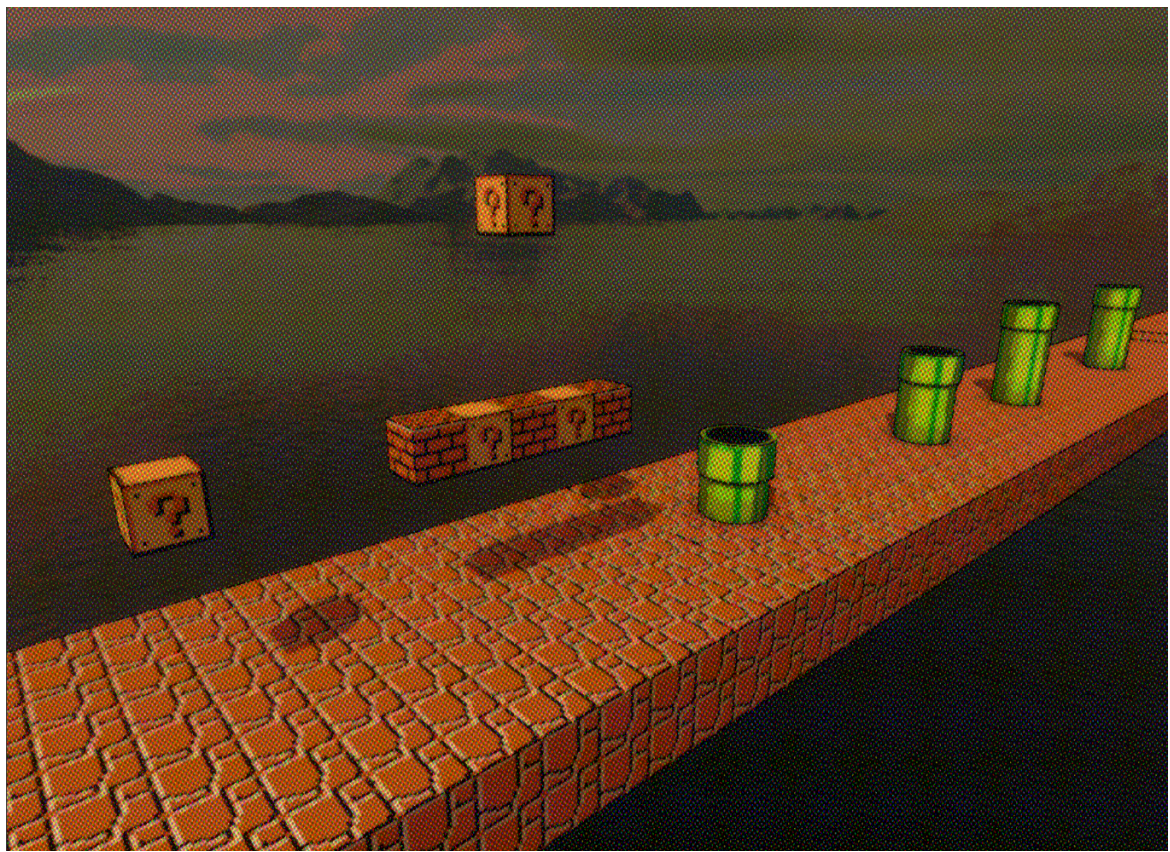


处理后：





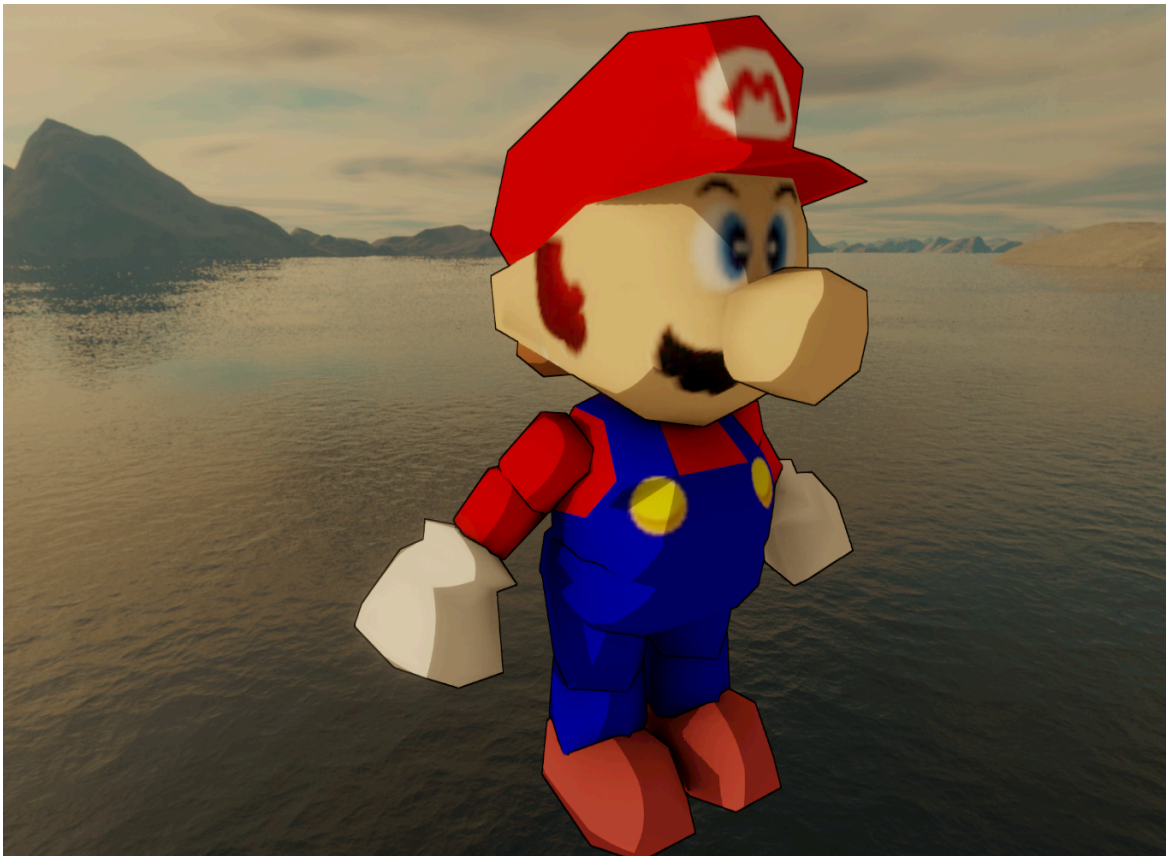




详细对比



Origin



Toon Shading & Outline On



Toon Shading & Outline & Halftone On



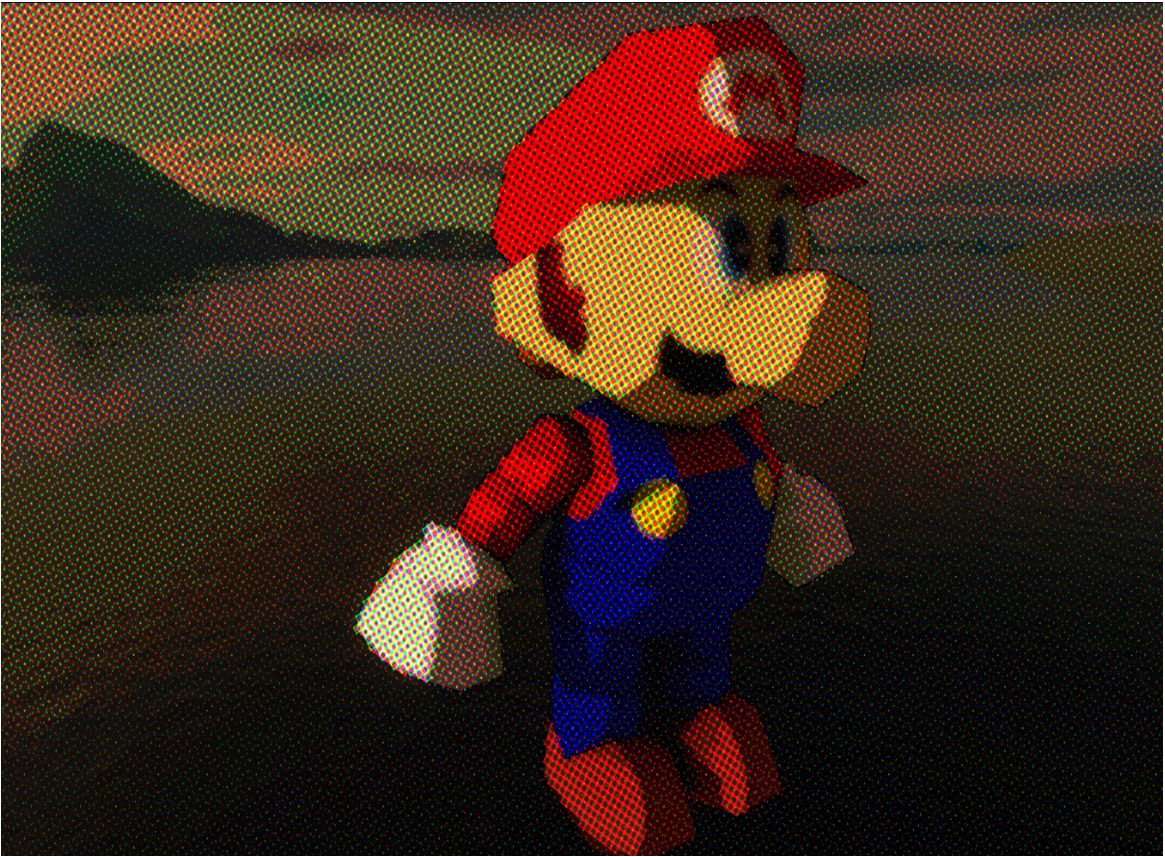
Toon Shading & Outline & Colorful Halftone On



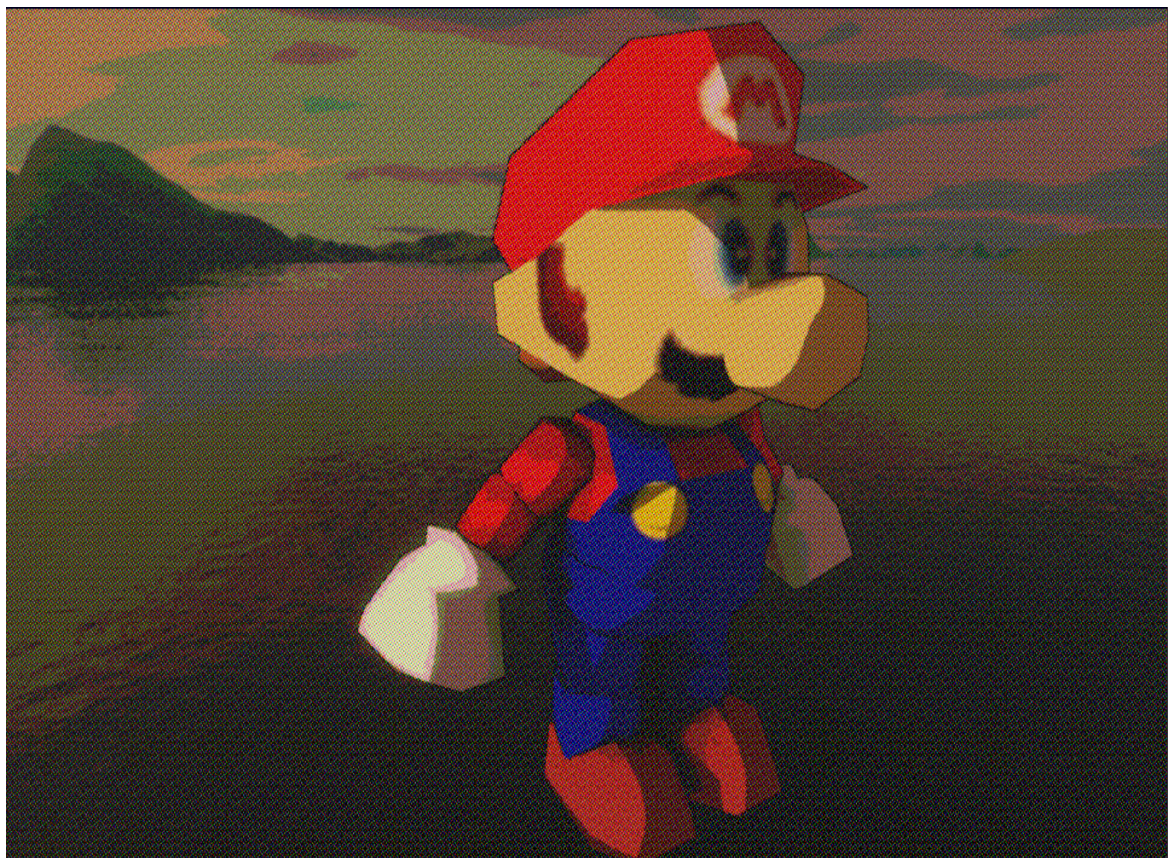
lower Halftone size level



higher Halftone size level



biger Halftone cell size



smaller Halftone cell size

新增功能

一、光照调整方式

把原本调整 x，y，z 三个分量的方式，改为了调整 yaw，pitch 的角度，更符合人类习惯。

核心逻辑

辅助函数：


```

static float3 DirectionFromYawPitch(float yaw_degrees, float pitch_degrees) {
    const float yaw      = Angle::DegreeToRadian(yaw_degrees);
    const float pitch    = Angle::DegreeToRadian(pitch_degrees);
    const float cos_yaw  = std::cos(yaw);
    const float sin_yaw  = std::sin(yaw);
    const float cos_pitch = std::cos(pitch);
    const float sin_pitch = std::sin(pitch);

    return float3(cos_yaw * cos_pitch, sin_pitch, sin_yaw * cos_pitch);
}

static void YawPitchFromDirection(const float3& direction, float& out_yaw_degrees, float& out_pitch_degrees) {
    const float3 normalized_direction = Normalizef(direction);
    out_yaw_degrees = Angle::RadianToDegree(std::atan2(normalized_direction.z, normalized_direction.x));
    out_pitch_degrees =
        Angle::RadianToDegree(std::asin(std::clamp(normalized_direction.y, -1.f, 1.f)));
}

```

用于把 yaw 和 pitch 转成 3D 方向向量，以及把 3D 方向向量转成 yaw 和 pitch

```

if (direction_changed) {
    SceneEditing::SetMainDirectionalLightDirection(
        *scene,
        DirectionFromYawPitch(m_main_directional_light_yaw, m_main_directional_light_pitch)
    );
}

```

如果滑条变化，就把新的 yaw/pitch 转回方向向量，再写回场景里的方向光。

二、二分色阶着色

核心逻辑

```

float GetToonTone(float NoL, float shadow) {
    float light_level = saturate(NoL * shadow);
    return lerp(param.toon_dark_factor, param.toon_light_factor, step(param.toon_threshold, light_level));
}

```

判断当前像素属于亮部还是暗部，并返回对应的亮度系数。


```
float3 AccumulateToonLight(Moer::GLight light, float3 albedo, float3 N, float3 position, float :
...
return albedo * light.intensity * light.color.rgb * tone;
}
```

用于计算一个光源对当前像素的 toon 光照贡献，最终颜色 = 模型本身颜色 * 光源强度 * 光源颜色 * toon明暗系数。

```
for (uint i = 0; i < lighting_data.light_count; i++) {
    Moer::GLight light = light_buf.Load<Moer::GLight>(i);
    toon_color += AccumulateToonLight(light, albedo, N, position, shadow);
}
return float4(max(toon_color, 0.0), 1.0);
```

最后在main函数中进行实际处理：遍历光源并累加 toon_color。

UI以及可调参数

Enable Toon Shading: 控制二分色阶着色的开关

Toon Threshold: 亮部阈值，高于这个值的被认为是亮部

Toon Dark Factor: 暗部亮度系数，越高越亮

Toon Light Factor: 亮部亮度系数，越高越亮

三、角色描边

核心逻辑

```
float SampleEdge(float2 uv)
```

该函数接收一个当前像素的 UV 坐标，返回一个边缘强度。0 表示不是边缘，1 表示是强边缘，中间值表示柔和过渡边缘。

```
float center_depth = TextureHandle(param.depth_tex).Sample2D<float>(uv);
float3 center_normal = normalize(Raster::UnpackNormal(TextureHandle(param.normal_tex).Sample2D<f
```

它首先读取当前像素的深度和法线。


```
edge = max(edge, smoothstep(param.depth_threshold, param.depth_threshold * 2.0, depth_delta));  
edge = max(edge, smoothstep(param.normal_threshold, param.normal_threshold * 2.0, normal_delta));
```

然后循环访问周围像素，检测深度差和法线差，更新边缘强度 edge。

```
float3 color = TextureHandle(param.input_image).Sample2D<float3>(uv);  
float edge = SampleEdge(uv) * param.outline_strength;  
  
color = lerp(color, param.outline_color, edge);  
return float4(color, 1.0);
```

最后在 main 函数中乘一个强度系数，并根据最终的边缘强度来混合颜色。

UI以及可调参数

Enable Outline: 控制角色描边的开关

Outline Color: 描边的颜色

outline_radius: 边缘检测的范围，越大检测的像素数越多

outline_depth_threshold: 深度类型边缘的阈值

outline_normal_threshold: 法线类型边缘的阈值

outline_strength: 描边强度

四、半调

半调是一种常见的印刷风格效果，它用规则排列的离散圆点来模拟连续明暗变化。本项目是这样处理的：先将最终画面划分成许多网格，每个网格根据中心点亮度生成一个圆点，暗处圆点更大，亮处圆点更小或不显示。除此之外，还支持简化的彩色半调效果，用 C、M、Y 三个颜色通道以不同角度叠加出类似彩色印刷的效果。

核心逻辑

```
float2 Rotate(float2 p, float angle) {  
    float s = sin(angle);  
    float c = cos(angle);  
    return float2(c * p.x - s * p.y, s * p.x + c * p.y);  
}
```

该函数用于旋转屏幕坐标。半调网点如果完全横平竖直，看起来会比较像普通滤镜，因此通过旋转坐标系让网点以一定角度排列，更接近漫画或印刷中的网点效果。


```
float Luminance(float3 color) {
    return dot(color, float3(0.2126, 0.7152, 0.0722));
}
```

该函数用于将 RGB 颜色转换为亮度。这里没有简单地对 RGB 求平均，而是使用了符合人眼感知的亮度权重，其中绿色对亮度的贡献最大，蓝色最小。

```
float QuantizeAmount(float amount) {
    uint levels = max(param.levels, 1u);
    if (levels <= 1u) {
        return saturate(amount);
    }

    float level_count = float(levels - 1u);
    return floor(saturate(amount) * level_count + 0.5) / level_count;
}
```

该函数用于将连续的暗度值分成有限的几档，从而让圆点大小产生更明显的层次。levels 越小，圆点大小变化越跳跃；levels 越大，圆点大小变化越平滑。

```
float2 GetCellCenterUv(float2 screen_pos, float angle) {
    float cell_size = max(param.cell_size, 2.0);
    float2 rotated_pos = Rotate(screen_pos, angle);
    float2 cell_id = floor(rotated_pos / cell_size);
    float2 cell_center_rotated = (cell_id + 0.5) * cell_size;
    float2 cell_center_screen = Rotate(cell_center_rotated, -angle);
    return saturate(cell_center_screen * param.inv_resolution);
}
```

这部分用于确定当前像素属于哪个半调网格，并计算该网格中心点的位置。这里不是直接使用当前像素的亮度，而是使用网格中心点亮度来决定整个网格中圆点的大小。这样同一个格子内的圆点半径一致，圆点形状会更稳定，不容易出现变形。

```
float GetDarknessAmount(float3 color) {
    float raw_darkness = 1.0 - saturate(Luminance(color));
    float darkness_low = min(param.darkness_start, param.darkness_end - 0.001);
    float darkness_high = max(param.darkness_end, darkness_low + 0.001);
    return smoothstep(darkness_low, darkness_high, raw_darkness);
}
```


这部分根据中心点颜色计算暗度。亮度越低，raw_darkness 越大。随后使用 darkness_start 和 darkness_end 控制半调响应范围：亮部基本不出点，中间调和暗部逐渐出现更大的点，从而避免整张图像只是均匀套上一层滤镜。

```
float GetDotMask(float2 screen_pos, float angle, float amount) {
    float cell_size = max(param.cell_size, 2.0);
    float2 rotated_pos = Rotate(screen_pos, angle);
    float2 cell_id = floor(rotated_pos / cell_size);
    float2 local = rotated_pos / cell_size - cell_id - 0.5;
    float dist = length(local);

    float max_radius = 0.70710678 * clamp(param.dot_scale, 0.0, 1.5);
    float radius = sqrt(QuantizeAmount(amount)) * max_radius;
    float edge_width = max(fwidth(dist), 0.001);
    return radius > 0.001 ? 1.0 - smoothstep(radius - edge_width, radius + edge_width, dist) : 0;
}
```

这部分计算当前像素到网格中心的距离，并根据暗度计算圆点半径。暗度越大，圆点半径越大。smoothstep 和 fwidth 用于让圆点边缘稍微平滑，减少锯齿和闪烁。

```
float GetShadowGate(float3 color) {
    return param.shadow_only != 0u ? GetDarknessAmount(color) : 1.0;
}
```

该函数用于实现 Shadow Only 模式。开启后，半调效果会再乘一个暗部权重，使亮部尽量保持干净，主要在中间调和暗部显示网点。

```
float3 ApplyMonoHalftone(float3 color, float2 screen_pos) {
    float2 center_uv = GetCellCenterUv(screen_pos, param.angle_radians);
    float3 center_color = TextureHandle(param.input_image).Sample2D<float3>(center_uv);
    float amount = GetDarknessAmount(center_color);
    float dot_mask = GetDotMask(screen_pos, param.angle_radians, amount);
    float gate = GetShadowGate(center_color);
    return lerp(color, param.dot_color, dot_mask * gate * saturate(param.strength));
}
```

这是单色半调模式。它根据网格中心点暗度计算圆点遮罩，然后将原图颜色和 dot_color 混合。暗处圆点更大，亮处圆点更小或没有。


```
float3 ApplyColorHalftone(float3 color, float2 screen_pos) {
    float cyan_angle    = param.angle_radians + 0.2617994;
    float magenta_angle = param.angle_radians + 1.3089969;
    float yellow_angle  = param.angle_radians;

    float cyan_amount    = (1.0 - saturate(cyan_sample.r)) * gate;
    float magenta_amount = (1.0 - saturate(magenta_sample.g)) * gate;
    float yellow_amount  = (1.0 - saturate(yellow_sample.b)) * gate;

    float3 print_color = saturate(float3(1.0 - cyan_mask, 1.0 - magenta_mask, 1.0 - yellow_mask));
    return lerp(color, print_color, saturate(param.strength));
}
```

这是彩色半调模式。它使用简化的 CMY 思路：根据红色通道生成青色点，根据绿色通道生成品红点，根据蓝色通道生成黄色点，并让三个通道使用不同的角度。这样可以模拟彩色印刷中不同颜色网点交错叠加的效果。

```
if (param.color_enabled != 0u) {
    color = ApplyColorHalftone(color, screen_pos);
} else {
    color = ApplyMonoHalftone(color, screen_pos);
}
```

最后在 main 函数中根据 UI 开关选择单色半调或彩色半调，并输出最终颜色。

UI以及可调参数

Enable Halftone: 控制半调后处理的开关

Color Halftone: 控制是否使用彩色 CMY 半调

Shadow Only: 控制半调是否主要作用于暗部，开启后亮部会更干净

Size Levels: 控制圆点大小分成几档，数值越小层次越明显，数值越大变化越平滑

Cell Size: 半调网格大小，数值越大，圆点间距越大，效果越粗

Dot Scale: 圆点最大半径系数，数值越大，暗部圆点越大

Strength: 半调混合强度，数值越大，圆点颜色越明显

Angle: 半调网格旋转角度，用于控制网点排列方向

Darkness Start: 半调开始出现的暗度阈值，越高则只有更暗的区域才出现圆点

Darkness End: 半调达到最大响应的暗度阈值，越低则圆点更容易变大

Dot Color: 单色半调模式下的圆点颜色

对于选题的理解和分析

作为一个爱玩游戏的人，我毫无疑问选择了非真实感渲染这个主题。作业指导里说“非真实感渲染的目标由需求决定，不同场景的需求差别较大”，所以我决定在 UI 上添加可以调节参数的滑条，便于我观察分析。

至于为什么选择“半调”艺术风格，是因为我觉得像素风有点太稀松平常了，而故障风和素描水彩风又感觉不够 cool。而半调这个风格就正符合我的审美。

踩坑记录

- 模型比例过小导致调试困难：**一开始导入的 glTF 模型非常小，摄像机稍微移动就容易飞出视野，最后通过在 glTF 的根节点上加 `scale: [20, 20, 20]` 放大模型（虽然最后没有选择这个模型展示）
- 方向光调节方式不直观：**最初方向光是用 `x/y/z` 三个方向分量调节的，但方向向量需要 `normalize`，所以调一个分量时另外两个分量也会变化，使用体验很差。后来改成 `Yaw/Pitch` 两个角度控制，更符合“调太阳方向”的直觉。
- Toon Shading 暗部过亮：**实现二分色阶后，最开始暗部也显得很亮。原因是 `toon_dark_factor` 本质上是暗部亮度倍率，如果默认值偏高，背光区域也会保留较多亮度。后来通过 UI 暴露 `Toon Dark Factor`，可以根据模型调低暗部亮度，让明暗分界更明显。
- 半调容易像一层普通滤镜：**如果所有区域都套同样密度的网点，半调会像简单覆盖了一层纹理。后来加入 `Darkness Start / Darkness End`，让半调主要根据暗度出现：亮部少出点，暗部点更明显，从而更符合印刷半调的视觉逻辑。

课程总结

课程先介绍几何光学、辐射度量、BRDF 与渲染方程，建立物理渲染的理论基础；再学习光线追踪、反射折射、阴影以及加速结构，并通过分布式光线追踪实现软阴影、景深、运动模糊等效果。这一部分紧跟时代，让我了解了目前这一方面的学术进展。

之后进一步讨论相机模型和材质外观，理解“怎样观察场景”以及“物体为什么看起来像金属、玻璃或粗糙表面”。最后从采样与重建讲到蒙特卡洛积分，学习如何用随机采样近似复杂的渲染积分，在计算成本和画面质量之间取得平衡。尤其是最后讲蒙特卡洛的部分，感觉对我很有帮助，让我了解了“信号”的数学表示，图像和傅里叶变换，以及用蒙特卡洛计算积分的原理。这些知识不仅在图形学上有用，对我理解计算机中的“信号”很有启发，这是一个广泛的通用的概念。

而且，学完这门课，我改变了之前对“渲染”的认识，我以前以为 PBR 没什么用，因为我看到的都是风格化的渲染。但学完后，我认识到 PBR 是所有渲染的基础，能够告诉我们合理的光照位置，反射情况等。

过老师的讲课也很有意思，由浅入深，零基础也完全能听懂并主动参与课堂上的思考。作为一门图形学导入课非常棒。

对于课程的建议就是：BRDF 感觉还是太难了😅，没怎么理解，也不知道怎么用。不过这可能是公式本身的问题，不是课程的问题。

项目地址

<https://github.com/ruirui-ryy/MoerEngine-NPR-Final>

位于 halftone 分支