MoldWing 实时纹理编辑系统

技术实现文档

Real-time Texture Editing Technical Documentation

项目名称	MoldWing - 倾斜摄影三维模型编辑器
文档版本	1.0
模块	M7 - 克隆图章工具
技术栈	C++17, Qt 6, DiligentEngine

目录

1. 系统概述

MoldWing 纹理编辑系统采用屏幕空间投影方式进行纹理编辑，与传统的 UV 展开编辑不同，用户直接在 3D 视图上操作，系统自动将屏幕坐标映射到纹理坐标。这种方式具有以下优势：

- 所见即所得 (WYSIWYG) - 编辑效果实时反映在 3D 模型上
- 无需理解 UV 布局 - 降低用户学习成本
- 保持原生纹理分辨率 - 实际编辑在纹理空间进行
- 支持撤销/重做 - 完整的编辑历史管理

核心组件架构：

组件	职责	关键类
GPU拾取	确定鼠标点击的三角形面	FacePicker
坐标映射	屏幕坐标 → 纹理坐标	ScreenToTextureMapper
编辑缓冲	CPU端纹理数据管理	TextureEditBuffer
GPU更新	将编辑同步到GPU纹理	MeshRenderer
撤销系统	像素级变更记录与恢复	TextureEditCommand

2. 屏幕空间到纹理空间映射

将屏幕上的鼠标点击位置转换为纹理坐标是整个系统的核心。这个过程分为四个步骤：

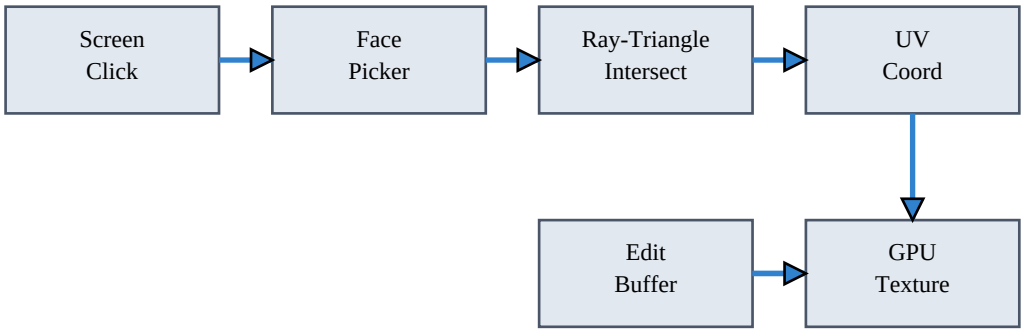

图1: 屏幕到纹理坐标的转换流程

2.1 GPU面拾取 (Face Picking)

使用 GPU 渲染一个特殊的 ID 缓冲区，每个三角形面以其索引值作为颜色输出。通过读取鼠标位置的像素值，可以快速确定点击的是哪个三角形。

Pixel Shader (HLSL):

```
uint faceID = SV_PrimitiveID; // Hardware-provided primitive ID
output.Color = float4(
float(faceID & 0xFF) / 255.0,
float((faceID >> 8) & 0xFF) / 255.0,
float((faceID >> 16) & 0xFF) / 255.0,
1.0);
```

2.2 射线-三角形求交 (Moller-Trumbore 算法)

获得面ID后，需要计算射线与该三角形的精确交点。Moller-Trumbore 算法是最高效的方法：

$$\text{■■■■}: R(t) = O + t \cdot D$$
$$\text{■■■■■}: P = (1-u-v) \cdot V0 + u \cdot V1 + v \cdot V2$$

算法步骤：

1. 计算边向量: E1 = V1 - V0, E2 = V2 - V0
2. 计算叉积: P = D × E2
3. 计算行列式: det = E1 · P
4. 如果 |det| < ε，射线与三角形平行，无交点
5. 计算 T = O - V0
6. 计算重心坐标 u = (T · P) / det
7. 计算 Q = T × E1

8. 计算重心坐标 v = (D · Q) / det

9. 计算距离 t = (E2 · Q) / det

10. 验证: u ≥ 0, v ≥ 0, u + v ≤ 1, t > 0

2.3 重心坐标计算

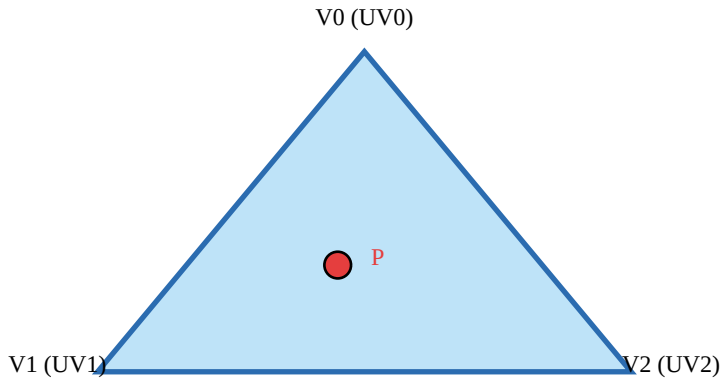

图2: 三角形内任意点P的重心坐标表示

重心坐标 (u, v, w) 满足: P = w·V0 + u·V1 + v·V2，其中 w = 1 - u - v。这些坐标表示点P到各顶点的相对权重，可用于插值任何顶点属性。

2.4 UV坐标插值

使用重心坐标对三个顶点的UV值进行线性插值，得到交点的精确UV坐标:

```
UV_P = w·UV0 + u·UV1 + v·UV2

texX = (int)(UV_P.u × textureWidth)
texY = (int)((1 - UV_P.v) × textureHeight) // V■■■
```

C++ 实现 (ScreenToTextureMapper.cpp):

```
void ScreenToTextureMapper::interpolateUV(
const float* uv0, const float* uv1, const float* uv2,
float baryU, float baryV,
float& outU, float& outV)
{
float baryW = 1.0f - baryU - baryV;
outU = baryW * uv0[0] + baryU * uv1[0] + baryV * uv2[0];
outV = baryW * uv0[1] + baryU * uv1[1] + baryV * uv2[1];
}
```

3. CPU纹理编辑缓冲区

3.1 TextureEditBuffer 设计

TextureEditBuffer 是纹理编辑的核心数据结构，它在 CPU 端维护一份纹理的可编辑副本：

成员	类型	说明
`m_data`	`vector<uint8_t>`	当前编辑数据 (RGBA)
`m_originalData`	`vector<uint8_t>`	原始备份 (用于撤销/橡皮擦)
`m_dirtyRects`	`vector<QRect>`	需要GPU同步的脏区域
`m_width, m_height`	`int`	纹理尺寸
`m_modified`	`bool`	是否有未保存修改

关键API：

```
// Pixel read/write
void setPixel(int x, int y, uint8_t r, g, b, a);
void getPixel(int x, int y, uint8_t& r, g, b, a) const;
void getOriginalPixel(int x, int y, ...) const; // Read original value

// Dirty region management
void markDirty(const QRect& rect);
bool isDirty() const;
void clearDirty();
```

3.2 脏区域追踪

为了优化GPU更新性能，系统追踪哪些区域被修改过。只有脏区域需要上传到GPU：

- 每次 setPixel() 调用会标记对应像素的包围盒为脏
- 多个脏区域可以合并为一个包围矩形 (dirtyBounds())
- GPU更新后调用 clearDirty() 重置状态

4. GPU纹理实时更新

当 CPU 端的编辑缓冲区被修改后，需要将变更同步到 GPU 纹理才能在渲染中显示。DiligentEngine 提供了高效的纹理更新接口：

MeshRenderer::updateTextureFromBuffer():

```
void MeshRenderer::updateTextureFromBuffer(
IDeviceContext* pContext,
int textureIndex,
const TextureEditBuffer& buffer)
{
if (!buffer.isDirty()) return;

// Prepare texture data
TextureSubResData subResData;
subResData.pData = buffer.data();
subResData.Stride = buffer.bytesPerLine();

// Calculate update region
Box updateBox;
QRect dirtyBounds = buffer.dirtyBounds();
updateBox.MinX = dirtyBounds.left();
updateBox.MinY = dirtyBounds.top();
updateBox.MaxX = dirtyBounds.right() + 1;
updateBox.MaxY = dirtyBounds.bottom() + 1;

// Upload to GPU
pContext->UpdateTexture(
m_textures[textureIndex],
0, 0, // mip level, array slice
updateBox,
subResData,
RESOURCE_STATE_TRANSITION_MODE_TRANSITION);
}
```

性能优化： 通过只更新脏区域而非整个纹理，可以显著减少 CPU-GPU 数据传输量。对于 4K 纹理 (4096×4096×4 = 64MB)，一次小型笔刷编辑可能只需要传输几KB数据。

5. 克隆图章算法

克隆图章工具允许用户从纹理的一个位置复制像素到另一个位置，实现纹理修复和复制效果。

5.1 源点选择

用户通过 Alt + 左键点击 设置克隆源位置。系统记录源点的纹理坐标：

```
// DiligentWidget.cpp - Alt+Click handling
if (m_interactionMode == InteractionMode::TextureEdit)
{
m_cloneSourceTexX = result.texX;
m_cloneSourceTexY = result.texY;
m_cloneSourceSet = true;
m_cloneFirstDestTexX = -1; // Reset first dest point
m_cloneFirstDestTexY = -1;
}
```

5.2 偏移计算

首次绘制时，记录目标起始点，计算源与目标的固定偏移：

```
offset = source_position - first_destination_position

// Subsequent painting:
sample_position = current_destination + offset
```

这样在拖拽过程中，源采样位置会跟随目标位置移动，保持相对位置不变。

5.3 像素复制

paintBrushAtPosition() 核心逻辑：

```
void DiligentWidget::paintBrushAtPosition(int texX, int texY)
{
// Calculate offset
if (m_cloneFirstDestTexX < 0) {
m_cloneFirstDestTexX = texX;
m_cloneFirstDestTexY = texY;
}
int offsetX = m_cloneSourceTexX - m_cloneFirstDestTexX;
int offsetY = m_cloneSourceTexY - m_cloneFirstDestTexY;

// Copy pixels within circular brush
for (int dy = -radius; dy <= radius; ++dy) {
for (int dx = -radius; dx <= radius; ++dx) {
if (dx*dx + dy*dy > radius*radius) continue;

int destX = texX + dx, destY = texY + dy;
int srcX = destX + offsetX, srcY = destY + offsetY;

// Read source pixel from original texture
m_editBuffer->getOriginalPixel(srcX, srcY, r, g, b, a);
// Write to destination
m_editBuffer->setPixel(destX, destY, r, g, b, a);
}
}
}
```

6. 撤销/重做系统

系统使用 Qt 的 QUndoStack 框架实现撤销/重做功能。TextureEditCommand 记录每次笔刷操作涉及的所有像素变更：

PixelChange 结构：

```
struct PixelChange {
int16_t x, y; // Pixel position
uint8_t oldR, oldG, oldB, oldA; // Old values
uint8_t newR, newG, newB, newA; // New values
};
```

工作流程：

阶段	操作	说明
beginPaint	■■ TextureEditCommand	准备记录变更
paint	recordPixel(x,y,old,new)	记录每个像素的前后值
endPaint	finalize() + push()	完成命令并压入撤销栈
Ctrl+Z	undo()	恢复所有像素到旧值
Ctrl+Y	redo()	重新应用所有像素新值

GPU同步： 撤销/重做操作会修改 CPU 缓冲区并标记脏区域。通过连接 QUndoStack::indexChanged 信号到 syncTextureToGPU() 槽，确保每次撤销/重做后 GPU 纹理自动更新。

```
connect(m_undoStack, &QUndoStack::indexChanged,
this, [this]() { syncTextureToGPU(); });
```

7. 完整数据流程

以下是一次完整的克隆图章操作的数据流程：

步骤	组件	输入	输出
1	鼠标事件	屏幕坐标 (x, y)	触发处理
2	FacePicker	屏幕坐标	Face ID
3	ScreenToTextureMapper	Face ID + 屏幕坐标	UV坐标
4	UV → Pixel	UV + 纹理尺寸	纹理像素坐标
5	paintBrushAtPosition	目标坐标 + 偏移	修改 EditBuffer
6	TextureEditCommand	像素变更	记录撤销数据
7	updateTextureFromBuffer	脏区域数据	GPU纹理更新
8	渲染循环	更新后的纹理	屏幕显示

关键性能指标：

操作	典型耗时	瓶颈
Face Picking (GPU)	< 1ms	GPU渲染
Ray-Triangle 求交 (CPU)	< 0.1ms	数学计算
像素编辑 (CPU)	< 1ms	内存带宽
GPU纹理更新	1-5ms	PCIe传输
总延迟	< 16ms	保持60FPS交互