**基于法向估计算法的游戏模型贴图优化方法研究与实现**

# 摘 要

法线贴图一直是游戏开发的重要技术，能够在低多边形表面上伪造细致的光影，开销低且效果好，应用十分广泛。法线贴图的原理是高模表面法线投射到低模上，记录在一张纹理中，算法过程是光线投射并计算相交，然后对交点进行重心坐标插值。然而，重心坐标插值被限制在一个三角形的范围，法线的平滑根本上是一种近似，这表明在极端情况下点法线的估算会有问题。

本文用主成分分析的方法计算得高模表面的特征，取最小特征值所在的方向作为法线方向，用协方差矩阵的逆对采样到的高模表面法线进行偏扰，提出了一种基于点邻域而非三角形图元的估算高模法向的方法，将估算得到的法线输出成法线贴图，可以在保留明显结构的同时模糊消除偏离期望的细节，明显提升渲染质量。本文使用 OpenGL 的 Java 绑定编写了一个包含重新生成模型法切线、生成体素并向其中存入图元、常规重心坐标插值烘焙、PCA分析等模块的法线烘焙工具，设计并实现了渲染到纹理、浏览模型、解码法线贴图等其他辅助功能。

本文提供了一种在烘焙器里实时解决烘焙缺陷的高效方案，这种新的烘焙方法能够移除常规方法无法处理的不平整瑕疵，提升模型的视觉效果，使美术流程更加自由。

**关键词：**光线投射；主成分分析；OpenGL；法线贴图；表面法向估计

**Baking Method Based on Fast Grid-based Ray-tracing**

# Abstract

Normal mapping has always been an important technology for game development. It can fake detailed light and shadow on low-polygon surfaces. It has low overhead and good effect, and is widely used. The principle of normal mapping is to project the normal of the high-poly surface onto the low-poly and record it in a texture. The algorithm process is to project rays and calculate intersections, and then interpolate the intersections in barycentric coordinates. However, barycentric coordinate interpolation is limited to the range of a triangle, and the smoothing of normals is essentially an approximation, which shows that in extreme cases, the estimation of point normals will be problematic.

This paper uses the principal component analysis method to calculate the features of the high-poly surface, takes the direction of the minimum eigenvalue as the normal direction, and uses the inverse of the covariance matrix to perturb the sampled high-poly surface normal. A method for estimating the high-poly normal based on point neighborhood rather than triangle primitives is proposed. The estimated normal is output as a normal map, which can blur and eliminate details that deviate from the expected while retaining the sharp features, significantly improving the rendering quality. This article uses the Java binding of OpenGL to write a normal baking tool that includes modules such as regenerating model normal tangents, generating voxels and storing primitives in them, conventional barycentric coordinate interpolated baking, PCA analysis, etc. Other auxiliary functions such as rendering to texture, browsing models, and decoding normal maps are designed and implemented.

This article provides an efficient solution to solve baking defects in real time in the baker. This new baking method can remove uneven defects that cannot be handled by conventional methods, improve the visual effect of the model, and make the art process more free.

**Keywords:** Ray Casting; Primary Component Analysis; OpenGL; Texture Baking; Surface Normals Estimation

**目 录**

[1 绪论 5](#_Toc29003)

[1.1 研究背景及意义 5](#_Toc11386)

[1.2 法线贴图的原理 7](#_Toc31483)

[1.2.1 法线贴图的渲染方式 7](#_Toc12222)

[1.2.2 法线贴图的编码方式 7](#_Toc3918)

[1.2.3 法线贴图的制作方式 8](#_Toc6917)

[1.3 国内外研究现状 8](#_Toc19673)

[1.4 本文主要工作 10](#_Toc31711)

[2 技术分析 11](#_Toc5881)

[2.1 JOGL介绍 11](#_Toc11290)

[2.2 在片元着色器中访问顶点 12](#_Toc23705)

[2.3 在片元着色器中光线投射 12](#_Toc20965)

[2.3.1 光线投射算法 12](#_Toc6746)

[2.3.2 NDC坐标系 13](#_Toc20959)

[2.3.3 全屏四边形 13](#_Toc6870)

[2.4 使用FBO渲染到纹理 13](#_Toc19147)

[2.4.1 渲染到纹理 13](#_Toc19984)

[2.4.2 帧缓冲 13](#_Toc12228)

[2.5 实现基本的法线烘焙器 14](#_Toc6482)

[2.5.1 流行的加速结构 14](#_Toc32031)

[2.5.2 流行的相交算法 15](#_Toc19203)

[2.5.3 与法线有关的其他实现细节 16](#_Toc9572)

[2.6 用协方差矩阵偏转法向量 16](#_Toc31301)

[3 系统分析](#_Toc32517) **[错误！未定义书签。](#_Toc32517)**

[3.1 系统总体分析](#_Toc6226) **[错误！未定义书签。](#_Toc6226)**

[3.2 需求分析](#_Toc176) **[错误！未定义书签。](#_Toc176)**

[3.3 系统运行流程图](#_Toc27124) **[错误！未定义书签。](#_Toc27124)**

[4 系统实现 19](#_Toc18814)

[4.1 体素 19](#_Toc4493)

[4.1.1 体素的设计 19](#_Toc6986)

[4.1.2 多维数组的一维表示 20](#_Toc22150)

[4.1.3 如何在GLSL中获取体素包括的三角形 20](#_Toc24810)

[4.1.4 数值稳定性 21](#_Toc32037)

[4.2 平滑法线 22](#_Toc26415)

[4.2.1 平滑法线 22](#_Toc23059)

[4.2.2 向量的哈希 23](#_Toc26316)

[4.2.3 顶点的合并 24](#_Toc26400)

[4.2.4 坐标映射法线的哈希表 24](#_Toc11929)

[4.2.5 重新索引顶点 25](#_Toc2815)

[4.3 生成切线 25](#_Toc30926)

[4.3.1 施密特正交化 25](#_Toc5137)

[4.3.2 切线的选择 26](#_Toc2852)

[4.3.3 左右手坐标系 27](#_Toc9862)

[4.4 基础变换 27](#_Toc261)

[4.5 加速结构 28](#_Toc8264)

[4.6 相交算法 31](#_Toc22575)

[4.7 法向估算 32](#_Toc18839)

[5 系统测试与优化 34](#_Toc23413)

[5.1 基础烘焙功能 34](#_Toc8006)

[5.1.1 启动软件 34](#_Toc9349)

[5.1.2 不同层级 35](#_Toc11238)

[5.1.3 烘焙贴图 36](#_Toc29912)

[5.1.4 观察低模 36](#_Toc20340)

[5.1.5 对象空间法线 37](#_Toc26429)

[5.1.6 烘焙效果评价 37](#_Toc4413)

[5.2 对追踪方向的测试 38](#_Toc18075)

[5.3 优化访问图元的方式 39](#_Toc13844)

[5.4 实际应用测试 40](#_Toc31710)

[6 总结与展望 42](#_Toc25874)

[参考文献](#_Toc21546) **[错误！未定义书签。](#_Toc21546)**

[致 谢](#_Toc14199) **[错误！未定义书签。](#_Toc14199)**

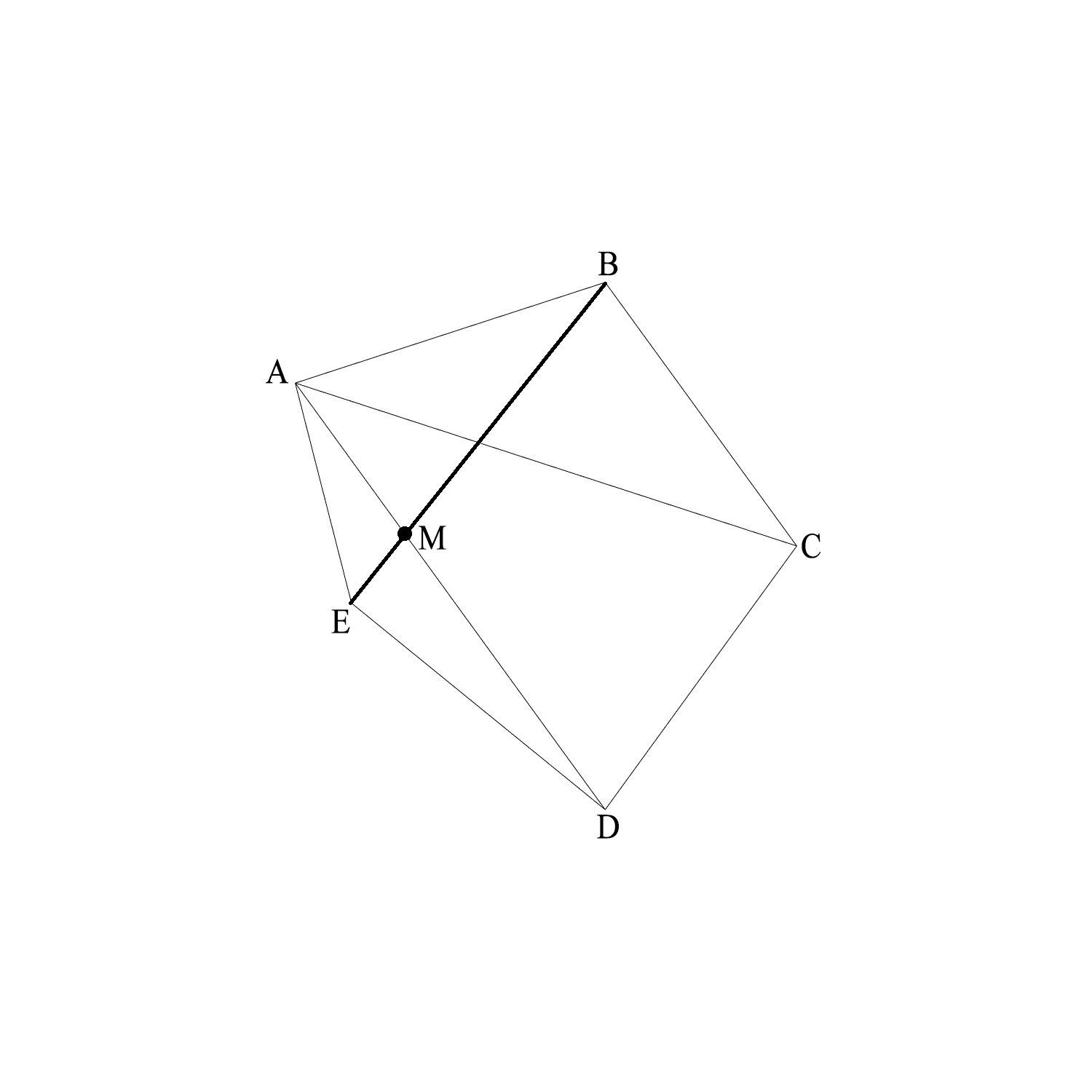
# **1 绪论**

## **1.1 研究背景及意义**

在游戏产业中，法线贴图使用非常广泛[1]。法线贴图简单，且高性能，其原理是将高面数模型（highpoly）的表面细节投射在低面数模型（lowpoly）上，能够用低性能损耗得到细致的光照效果。然而，多边形模拟的表面是有限制的，插值算法根本上是一种近似，这表明在极端情况下点法线的估算会有问题。

在计算机图形学的实际应用中，所有多边形最终都会被分解为三角形[2]，因为现代图形硬件是针对三角形优化的。这种分解过程称为三角化，它确保了任何形式的多边形网格都可以被有效地渲染。因此，三角网格可以被视为多边形网格的一种只包含一个三角形的特殊形式，三角形是图形硬件最基本的处理单元。

高模拓扑结构会极大地影响光线追踪时光子对三角形进行重心坐标插值的结果。

****

**图1-1 拓扑分析**

三角重心坐标插值算法知且仅知当前三角形的三个顶点，然而一个多边形平面实际上可以细分成多个三角形。图1-1展示的五边形可以分为，但是对五边形内任意一点插值的时候，实际上是将该点置于这三个三角形其一之中插值。因此，譬如连接EB交AD于M，要对EM线段上的一点插值时，它落在该五边形内，但算法并不是对五边形进行多边形插值，而是对进行插值。

然而仅仅知道A、E、D这三个顶点的数据，它不知道B点的数据。从直觉上来说，M点应该具备E和B两点按权重的混合，这个权重可以是EM：MB的长度比，但重心坐标插值算法强制要求E点在边AD上权重为0，因此事实上M点上E点的权重为0。这是因为算法要避免分界线上产生明显接缝。如若不然，接缝的成因是AD在的一侧取到不为0的E点权重，但在的一侧取到为0 的E点权重，因为对E是未知的。

限制E点在AD上权重为0，带来明显且严重的视觉瑕疵，因为E点的份额会在短短的线段EM上迅速且剧烈地消耗至0，仿佛线段AD将这个多边形劈成了两片。E点到线段AD的距离越近，E点的份额就越是无法沿着线段EB向多边形的另一侧传播。在三角形上取任何一点做重心坐标插值，仅仅能保证它是三角形所知的三点的均匀混合，但不是整个多边形上所有的点的均匀混合，除非是将三角面图元重构为N-多边形图元，采用多边形插值的方式来采样多边形上的一点。

**本文的研究内容是如何使得图元能够访问空间内的顶点，基于邻域计算各最近点在其上的影响权重。**虽然高模明确地定义了表面，但本文对光线的每个落点重新拟合表面，本质上是一种法向估算方法。

传统方法在重心坐标插值的时候只知当前三角形的三个顶点。本文提出的算法则在空间里搜寻邻近点，计算中心，依据各点到中心的差值构造协方差矩阵，将任意的法线输入变换到成分空间，再按照各成分的方差大小重新变换回世界空间，使变换后的结果尽可能向方差最小的方向靠拢。

类似的方法经常用于点云处理。本文与点云处理的区别是，高模法线是已知的，协方差矩阵用来偏转法线，而无需进行分解。

在点云处理当中，点云数据是离散的，彼此之间不相连的。如果顶点上只带有关于位置坐标的信息，那么在不同时间采样到的两个位于同一坐标的点无法分出彼此的区别，但实际上它们尽管位置相同，表达的却可能是两种表面，因此在点云处理的时候，需要用最小二乘法拟合一个最能描述k-邻域（周围k个最近点）的平面，使得周围点到平面的方差和最小，这个平面最好地描述了点的特征。曲率和法线被称作特征描述符，来表示当对点采样的时候，实际上采样到怎么样的平面。一种最简单的估算表面法线的方法是主成分分析。

在主成分分析的时候，从原数据当中抽象出一系列新的向量，形成一个正交坐标系。坐标系的其中一轴是整个点云数据最大的差异分布方向，特征值最大，可以看作是拟合面的切线方向；将点云投影到垂直于该轴的平面上，找一条向量使得投影的差异分布最大，形成坐标系的第二条轴，特征值其次；剩下的那一轴即点云差异分布最小，可以看作是表面法线。

实际测试表明，本文所提出的方法同样有不小的局限性，依赖于大量的采样点才能获得良好效果，在先前提出的问题上，同样难以为低多边形生成良好的插值。然而，在采样点数量足够多的时候，本文的算法能够高效地分析出点云特征，以近似去噪的方式抹平原高模表面的鼓包等，可能是细分平滑或雕刻笔触造成的偏离期望的瑕疵。在高模制作的过程中，常见的问题是布线不符合实际上的结构，导致边缘锯齿化严重，但本文提出的方法可以高效抹除这些瑕疵。此外Dynamesh会产生大量的连接5边或6边的极点，导致光影细节错误，但按高模的特征进行模糊可以消除这些细节上的差异。

## **1.2 法线贴图的原理**

### **1.2.1 法线贴图的渲染方式**

法线决定物体的光照。在进行光照的时候，平面的亮度和光线照射的角度有关。照射光线越垂直，平面反馈的信息就会越亮。如果一个平面法线方向都是一致的，平面所反馈的亮度信息也是一样的，从而在视觉上就呈现出平面性。但是如果这个平面法线信息做一些改动，使他们朝向尽不相同，那么计算机渲染就会反馈出这个平面的亮度明暗不一，从而给人一种视觉上凹凸不平的感觉。如果模型表面法线储存的信息发生变化，模型的光照效果也会随之变化。

法线贴图是对单一的面添加细节，这个过程实际上是重心坐标插值的过程，由硬件执行，处于光栅化和片元着色之间。在程序上的实现过程是，首先将离散的顶点组装成图元，然后根据图元对屏幕的覆盖范围计算被覆盖到的像素，每个像素都知道该图元的构成，因此可以访问三个顶点计算得到插值，片元着色器可以比喻为将一个任意的输入图元进行微分，微分结果可以精细到超出图元级别，进入像素级别。纹理采样时，微分的对象是顶点的UV坐标，如此可以得到图元上任一点的UV坐标，随后将该坐标扩展到纹理的实际尺寸，取整，经过纹理过滤，即可找到纹理上的一点。因此，法线贴图理论上应该能完全覆盖低模的法线，但实际上这很难做到，因为虽然法线贴图的工作原理只与纹理有关，但大部分法线贴图是定义在切线空间中的。

### **1.2.2 法线贴图的编码方式**

切线空间法线用低模本身的法线来编码。切线空间法线在纹理中储存的是基向量的系数。向量空间中任意一个元素，都可以唯一地表示成基向量的线性组合。切线、副切线和法线组成了一组正交基，空间中的任意向量都可以按照平行四边形法则表示为基向量分解的形式，在法线贴图中，储存的是系数a、b、c，因此可以称切线空间法线贴图是对原始法线的“扰动”。切线空间坐标到世界空间坐标的变换指的是将系数a、b、c代入回公式，把T、B、N向量表示的任意向量还原到世界空间，于是，同理可得y、z。

### **1.2.3 法线贴图的制作方式**

烘焙（Baking）是计算机图形学中经常使用的术语，在游戏开发和传统的非实时渲染中广泛运用。烘焙指的是将3D几何体的表面特征保存到低模即将使用的2D纹理文件的过程，可以烘焙的内容包括环境光遮蔽（AO），法线，顶点颜色，位置，曲率等，或者是这些属性组合产生的光照和材质效果。烘焙的载体是2D图像平面，通过纹理坐标系与3D空间的位置产生一一对应关系。

法线烘焙的首要任务是在给定低模上一点的条件下找到它在高模上的对应一点，这种投射是通过从工作模型表面投射光线到参照模型上完成的。

## **1.3 国内外研究现状**

Xnormal、Fornos、Marmoset等纹理烘焙工具都可以传递法线信息。Knald从高度图推导法线信息。Substance也集成了烘焙功能。此外，也有纹理烘焙器是Unity和Unreal引擎的插件，比如BNAO。Blender、Houdini等流行DCC软件也有纹理烘焙插件。这些软件或插件虽然流行且高效，但本文研究的内容目前并没有其他软件或插件做过。

Fornos GPU Texture Baking Tool（2018）是一个开源的纹理烘焙器，采用C++语言编写，支持AO、法线、厚度、高度等信息的烘焙。本文在一些设计思路和具体实现上参考了它的设计。它的做法是将模型UV在CPU中光栅化到一张2D纹理上，随后按照纹理尺寸运行计算着色器，将光线追踪的结果写入纹理。本文为了节省编写CPU光栅器的时间，直接将代码运行在常规渲染管线的片元着色器当中，光栅化由GPU硬件完成。

常规的烘焙手段的精度太高，以至于完全复刻高模的瑕疵。本文旨在提出一种在烘焙过程中，在采样高模表面时，能够模糊并去除高模表面较小的凹凸噪声的方法。为了做到这一点，需要估算并重新生成法线。

重新设计模型的法线不是常规游戏模型制作流程的一部分，但并不罕见，通常可以在卡通渲染管线中看到。在卡通渲染中，需要模型的法线近似球状，忽略事实上的面朝向和结构，来获得更好的，更一体的光影，模拟光滑且连续的体积，使得零散的树叶面片能够有彼此之间相关联的融球状光影表现，或使得人体模型分出工整且大块的亮暗面。这是一种对本文有参考价值的技术，通常称作球形法线、包裹式法线或用户法线，以表明它不是一般手段计算的，而是专门为用户的特定需求调制的。该功能有相当多的工具实现，比如User Normal Translator，国内也有人在Blender上实现。

另外对本文的研究有帮助的是广义重心坐标插值，比如均值坐标插值。这类技术可以用于图像变形，也可以解决三角形重心坐标插值的对角线问题，但仅适用于平面。多边形插值将可知信息从三角形的三个顶点拓展到N-多边形顶点，N-多边形是N个点形成的凹或凸多边形。虽然三角形重心坐标能够正确插值得到位置坐标，但对于本身不规则的其他信息，比如法线和颜色，是无法均匀混合的，但这在广义重心坐标插值下得到了解决。Kai Hormann的论文Barycentric Coordinates for Arbitrary Polygons in the Plane介绍了几种为N-多边形建立重心坐标并进行插值的方式，其渲染结果很具有参考价值。

脱离三角形对散点进行采样的需求常见于流体模拟和流体渲染。流体是一个研究广泛的领域。计算流体动力学的步骤分为三步，首先是模拟得出物理属性，然后提取可渲染的表示，最后是基于物理的渲染，第二阶段是对本文有参考意义的。在流体模拟中，每个粒子搜索固定半径内的邻近粒子，计算出它们给予的密度和压强，更新自己下一时刻的运动状态。在流体渲染的过程中，更新了位置坐标的流体粒子本身携带密度信息，因此空间中某一处的值可以表达为附近点的密度的混合，如果算得密度大于某个阈值，就说明此处位于流体内部，从而可以重建表面。

传统的基于粒子的流体渲染方式是累加径向基函数的结果并提取等值面，但这种方法产生球形伪影，这在Yu和Turk的各向异性方法中得到了很好的解决[3]。本文注重光线步进作为提取表面的方式。光线步进一般指体渲染，用于渲染3D纹理定义的烟雾和云团，光线步进也可以用于提取流体的表面特征。Xiao等人将将粒子扩展成球体，将光线的起点放在球体表面上，来跳过从摄像机到球体表面的空白部分[4]；Wu等人提出了一种基于二值网格跳过低密度区域，并分类飞溅液体和聚集液体来减少光线步进的次数的优化方式[5][6]。



图1-2 在Houdini的测试

Yu和Turk的各向异性方式使用了各向异性核来处理向量，可以创造相当平滑的表面。各向异性的方向和缩放通过测试每个粒子的邻近空间粒子密度得到，如图1-2所示，椭球的朝向与粒子的空间分布有关，椭球的尺寸和聚集程度有关。各向异性表示为协方差矩阵，协方差描述张量之间的差异分布。因为需要计算协方差，所以首先需要计算粒子的平均位置，作为粒子集的期望，这个位置的得出也是径向基函数插值的过程。

各向异性方法最早由Dinh等人提出以处理点云数据。对于空间中的给定点，相邻点的特征通过PCA分析来表征，以得到粒子分布的主成分轴，将此信息用于拉伸表面来拟合粒子的实际分布，可以保留边缘和尖角等特征。Yu和Turk与流体表面重建结合使用的各向异性方法启发了Xiao等人和Wu等人估算表面法线的方法[4][5][6]，基本思路是对所求点执行PCA分析，以最小特征值对应的特征向量作为法线方向，本文受这些研究的启发，采用了类似的方法，但在细节上有所差异，不对协方差矩阵进行特征分解。

从粒子集中估算法线是点云处理领域常见且得到充分研究的问题，常见的算法有PCA回归，Voronoi，移动最小二乘法等。PCA回归对邻域执行主成分分析得出切平面，每一个邻域点上也可能添加了高斯权重，如Pauly等人的方法，然后用切平面的法线代表点法线。PCA算法拟合平面，假设点云的采样数据处处光滑，点附近的局部曲面可用平面拟合，用此平面代替该点的法向量。

## **1.4 本文主要工作**

本文编写了一个渲染器，本文实现的主要功能都以可替换的着色器的形式存在，法线烘焙是一种着色器。本文的主要工作及创新点如下：

本文首先将高模看做多边形几何，重心坐标插值得到交点，在计算得到射线和平面的相交关系后，记录相交位置，将相交位置输入给重心坐标插值函数，对法线插值。

随后将高模看作点云，以光线和平面的交点为中心，遍历周围的体素里的顶点，分析主成分，此时借助先前插值得到的法线信息排除相反的面，然后计算协方差矩阵。得到协方差矩阵后，将先前插值得到法线变换到成分空间，再以方差为系数变换回世界空间，和原法线相比，新法线偏转向切平面的法线方向，最后通过烘焙的方式将新法线传递到低模纹理中。

本文采用寻找空间内邻近的点的方式，这种方式比为每个顶点构建拓扑上相连的点邻域耗时更少，效率更高，且发挥GPU的并行优势，使得此算法可以应用于高多边形。

本文提出一种在法线烘焙的时候对高模点云进行法向估算的方法，渲染得到比一般重心坐标插值算法的结果更准确，更连续，更符合模型形态的法线。主成分分析可以模糊法线贴图上不显著的小结构，消除伪影。

通常法线的定义与顶点实际位置相关，移动顶点也会造成法线偏移。本文提出的法向估计算法在不改变高模实际顶点位置的前提下修改了捕获到的法线。这么做的好处是着色器按纹理贴图的像素数量运行，不需要关心高模实际的顶点数量。在顶点量极大的时候，处理模型是比处理图像更消耗时间和空间的。

# **2 技术分析**

## **2.1 JOGL介绍**

本系统采用Java和JOGL编写。OpenGL是应用广泛的图形API，具有跨平台，跨语言性，能够实现高性能高质量的渲染。OpenGL 是一套标准化接口。标准化接口能够提供必要的抽象层，以支持设备和平台的无关性。

JOGL是OpenGL的Java语言绑定。语言绑定允许将一种编程语言的代码与另一种编程语言的代码相互交互和调用。这意味着开发人员可以使用一种编程语言编写的库或模块，并在另一种编程语言中使用它们，不同编程语言之间可以实现相互调用，并共享各自的资源和功能。OpenGL有着多种不同的语言绑定形式。

由于 OpenGL 驱动版本众多，它大多数函数的位置都无法在编译时确定下来，需要在运行时查询。开发者需要在运行时对每个可能使用的函数获取地址并将其保存在一个函数指针中供以后使用，这个过程复杂且繁琐。这些功能已经有库来提供，比如GLAD。

例程是系统对外提供的功能接口的集合，JOGL提供了访问OpenGL例程的功能，这通过GLAutoDrawable 接口实现。GLAutoDrawable能够访问GL对象以调用OpenGL例程，提供了GLEventListener 回调机制来执行渲染，使用 display方法强制OpenGL同步地渲染。

此外，窗口操作在每个系统上都是不一样的，OpenGL 有目的地将这些操作抽象出去。用户不得不自己创建窗口，定义 OpenGL 上下文以及处理用户输入。OpenGL上下文指的是，在应用程序调用任何OpenGL的指令之前，需要安排首先创建一个OpenGL的上下文。这个上下文是一个非常庞大的状态机，保存了OpenGL中的各种状态，这也是OpenGL指令执行的基础。OpenGL的函数不管在哪个语言中，都是面向过程的，本质上都是对OpenGL上下文这个庞大的状态机中的某个状态或者对象进行操作。GLUT，GLFW等库为此设计，节省了书写用户接口相关代码的时间，提供渲染用的窗口和上下文。

OpenGL 最核心的功能是向使用者提供管线编程功能，通过使用GLSL语言对渲染管线进行编程，可以在 GPU 上编写灵活的着色器程序，以控制图形渲染过程中的各个阶段[7]。

JOGL为 Java 开发者提供了访问 OpenGL功能的接口，允许 Java 程序利用 OpenGL 的强大图形渲染能力，使得在 Java 平台上创建高性能、交互式的 2D 和 3D 图形应用程序成为可能。JDK本身不包含三维图形开发组件，JOGL提供了Java开发三维领域的能力，适合有 OpenGL 开发经验的开发人员使用。

## **2.2 在片元着色器中访问顶点**

着色器储存块（Shader Storage Buffer Object，SSBO）是一类缓冲对象，储存在逐顶点、逐像素调用中不会改变的常量。着色器可以在逐顶点、逐像素的调用中向SSBO写入。SSBO具有极大的容量，因此通常用在光线追踪程序中用来输入顶点。SSBO是可以读写的，这意味着着色器程序可以向SSBO写入内容，在fornos法线烘焙器中，作者即使用了SSBO来接收计算着色器的运算结果。在SSBO存在以前，要在片元着色阶段读取几何信息，是通过纹理缓冲对象（TBO）做到的。TBO的容量更小，适用性不如专门为GPU通用目的计算服务的SSBO高。

一种使用SSBO的方法被称为Programmable vertex pulling，用SSBO取代了VBO的位置，Attribute则被手动对结构体的编程取代。在给定VS运行次数之后，可以通过gl\_VertexID对SSBO进行遍历。这种方式的优点在于省去了繁琐的Attribute绑定步骤，不用预先计算步长。

在GLSL中，Interface block是一系列输入输出的集合，它有一些特殊的语法，譬如管线之间通信的大名、实例的小名、名称的全局定义。声明SSBO的语法和Interface block的语法相同。在SSBO的声明中，可以用一个变长数组来划分数据，这个变长数组必须声明在所有其他定长成员之后。这个变长数组可以是结构体的数组，所有剩下的数据被划分为若干组，符合结构体的成员声明。

## 2.3 在片元着色器中光线投射

### **2.3.1 光线投射算法**

光线追踪算法从摄像机镜头向场景中发射光线，测试光线与场景内的物体计算交点，但光线追踪总是涉及反射、折射、材质属性，法线烘焙只取光线追踪的第一步，在现在这种算法更多地被称为光线投射。光线投射是光线追踪的前身，基本思路是一致的，从摄像机出发，从每个像素向场景投射光线，对所有几何体测试是否存在碰撞，如果有，则确定交点。这种算法也常见于碰撞检测。光线投射的主要研究内容是，在加速结构和各种相交算法的帮助下更高效快速地找到相交的几何体。

### **2.3.2 NDC坐标系**

将渲染输出显示到屏幕上时，存在的问题是设备之间的屏幕尺寸是不统一的，因此有NDC（归一化设备坐标）坐标系。NDC坐标系是和设备无关的标准化坐标，用于将不同分辨率的屏幕坐标统一到同一通用空间。模型经过世界-视图-投射矩阵变换（MVP矩阵）的变换后进入裁剪空间（Clip Space），将三个轴上超过-W~W范围的顶点裁剪，随后执行透视除法，进入NDC坐标系规定的(-1,-1,-1)~(1,1,1)范围，所有该范围之外的顶点都是不显示的。但是模型也可以不经过MVP变换直接进入NDC空间，比如一个非常小的物件，它可以直接显示在NDC中，由于屏幕是二维的，这时就相当于观察模型在XY平面上的轮廓。模型也可以直接定义在NDC空间中，比如Unity引擎在执行Blit操作时就是渲染了一个全屏三角形，然后在三角形上运行着色器，着色器从某一源纹理采样，在渲染的过程中可以对它施加影响以进行全屏后处理；使用RenderDoc可以看到该全屏三角形的实际样貌。

### **2.3.3 全屏四边形**

全屏四边形（Fullscreen Quad）是一个特殊的几何形状，它覆盖了整个渲染纹理（通常是屏幕或帧缓冲区）的面积。这个四边形的顶点坐标通常被设置为屏幕的四个角落，使得它能够填充整个视图。全屏四边形提供了一种简单而高效的方式来在屏幕上渲染内容。在通用计算着色器被引入之前，OpenGL上的光线追踪算法依赖全屏四边形。全屏四边形占据NDC空间中的(-1,-1,0)~(1,1,0)空间。模型纹理坐标系的范围在(0,0)~(1,1)之间。因此只需要对模型的UV作算术就可以转化到NDC空间，铺满屏幕。额外注意事项：NDC空间存在深度，但所有UV岛的深度都是0；实际上在FS执行完毕后还会执行深度检测，如果UV有重叠的，那么后渲染的片元会被前者覆盖（不通过LESS检测）或者覆盖后者（通过LEQUAL检测）。

## **2.4 使用FBO渲染到纹理**

### **2.4.1 渲染到纹理**

渲染到纹理（Render to Texture）允许开发者将场景的一部分图像拷贝到一张纹理图，重新应用到场景中的某一物体上,这种技术最经典的使用案例就是可交互雪/沙和在游戏内绘画。在本系统中，首先渲染了光线追踪的结果，而后将该图像应用在模型表面，以产生仿高模的光照效果，这也是渲染到纹理的一种应用方式。

### **2.4.2 帧缓冲**

帧缓冲对象（FrameBufferOjbect）提供自定义的渲染目标。渲染目标允许程序将片元着色器的输出绘制到纹理上，而不是屏幕上，随后渲染的场景就可以作为纹理应用在其他地方。帧缓冲对象拥有一系列指针，这些指针指向储存实际颜色数据的纹理对象，这些指针被称作Attachment，通过glDrawBuffer函数绑定，这也是为什么该函数的第二个输入是一个整型缓冲。实际储存数据的是帧缓冲对象引用的纹理对象，因此只有对纹理对象执行读写，才可以导出颜色数据。根据OpenGL文档，读写纹理对象的方法如下：**glReadBuffer** 函数指定颜色缓冲区，该颜色缓冲区作为后续**glReadPixels** 和**glCopyPixels** 命令的源，读取到ByteBuffer后利用Java.awt的BufferedImage输出。

## **2.5 实现基本的法线烘焙器**

### **2.5.1 流行的加速结构**

流行的加速结构有很多，比如BVH、八叉树、kd-tree、均匀网格。不仅是光线追踪需要这些加速结构，它们也是碰撞检测的核心内容。加速结构可以排除大量一定不会与光线相交的三角形，这样只需要检测可能与光线相交的一小部分三角形就足够。

Fornos GPU Texture Baking Tool（2018）的加速结构构建为BVH，这是一种划分物体的算法，阅读代码可以推测，程序先自上而下，再自下而上，对每个节点的子树按从左往右的顺序遍历，若在左子树没有通过AABB相交测试，则跳到右子树，如果依然没有通过相交测试，则跳到上一级的右子树，不断往上提升层次，直到完全跳出，或者在另外的分支中找到相交。因为BVH是层次结构，只需要判断上一级AABB是否通过相交测试，如果没有，就意味着无需再寻找它的子集，应该向平级寻找。

GPU Gem3提到了均匀网格，这是一种划分空间的算法，把模型放进尺寸统一的体素中。本系统使用均匀网格作为划分方法，因为均匀网格方法适合用于查找单一网格上的顶点，尤其是低模和高模大致占用相同的空间的时候。对于本文的情况，它也很适合用来查询空间内的点。这种方法比记录拓扑邻域更快。

低模和高模占用差不多一致的空间意味着可以跳过大量用于填充模型表面到总体AABB之间的空隙的体素，将算法的时间复杂度限制在非常低的量级上。但是实际应用编写中发现存在问题，光线发射点可能在高模表面之下，或者处在超过迭代深度的高空，这些都导致无法查询到高模表面上对应的位置。这里适合应用GPU Zen2提到的Offset Map，这也是八猴的特性之一，可以对低模表面任意一点确定包裹框的高度。

GPU Gem3提到了3D DDA直线算法。这里使用的直线算法要求在均匀网格中延长光路并计算所有经过的体素。因此，与屏幕光栅化时使用的直线算法不同，用于体素的直线算法不评估并筛选更靠近的体素，它统计经过的所有体素。Tmax标志光线从哪里离开体素，Tmax的最小分量就是光线最先碰到的坐标轴，在这个坐标轴上步进1找到紧邻的下一个体素，再重复这个过程。该方法由Woo等人发明。

### **2.5.2 流行的相交算法**

检测3D空间中光线和三角形的相交性的算法有很多种，比如Möller-Trumbore 交集算法、Baldwin-Weber算法、Woob-Benthin-Wald水密算法、Havel-Herout 算法等许多。

求解相交性的常规方法从法线和平面垂直出发，作出等式，A是三角形上一点，通常取三角形的某一顶点。解得。将t代回直线的参数方程即可找到交点，随后再判断该点是否在三角形内，判断点在三角形内的方法也有许多，比如算，，面积的绝对值的和；或者沿某一时针方向作叉乘，异号则表明在三角形外；或者使用参数方程，其中满足。

Möller-Trumbore 交集算法是使用最广泛的一种求交算法，也是本系统使用的方法。它的基础是直线和平面的相交，对此作等式，可以变形为，如此可以将t、u、v看做是在重心空间的坐标，利用克莱默法则求解得到t、u、v，其中t代入直线的参数方程计算交点，u、v可以用来进一步判断点是否在三角形内。

Baldwin-Weber方法可以看做是前者的变形，也是基于线性无关的一组3维基向量，其中两者是三角形的边，另一者是平面法线最大分量所在的轴。这么做可以保证第三向量无法用另外两个向量的参数方程表示，即在3D 中非共面的三个向量是线性独立的，或者说如果不存在包含N个向量的（N-1）维子空间，那么它们是线性独立的。Baldwin-Weber方法首先提出一个4x4矩阵，矩阵的前两列是三角形的边向量，第三列是平面法线最大分量所在的轴，第四列是三角形的其中一个顶点。任意定义在重心空间的坐标在矩阵变换后即化成平面参数方程的经典形式。这个矩阵因此被称为重心空间到世界空间的变换矩阵。随后，Baldwin计算了矩阵的逆矩阵，这样可以将任何世界空间向量变换到重心空间。最后，他用这种方法将光线变换到重心空间，很方便地计算出了t、u、v三个关键变量。

### **2.5.3 与法线有关的其他实现细节**

平滑法线是一种常见的需求，通常出自于卡渲描边的需要，因为不平滑的法线使得三角形在挤出后炸开。平滑法线最常见的做法是用键值对归纳一个点的所有法线，然后归一化。法线可以按权重混合。Blender的加权法线算法可以对面积、角度或两者之积加权。Maya对面积和角度之积加权。对面积加权能使得面积越大的面越能吸引法线向自己靠拢。

切线空间法线总是需要TBN矩阵编码，但TBN矩阵的计算方式是不一定的。目前通用的标准是MikkTSpace，流行的应用都提供了对MikkTSpace的支持。任何人也都可以编写自己的切线空间生成方式，只要保持在应用程序中一致，即可获得正确效果，比如XNormal允许用户编写自己的切线空间，Maya、3DMax也有自己的切线空间计算方式。本系统参考了LearnOpenGL将纹理坐标系作为切线和副切线的方法，但是实际计算中只需要计算切线，而副切线可以通过叉乘获得。

Mikkelsen解释了切线空间编码解码的问题：关于切线空间法线贴图的一个常见误解是，这种表示方法在某种程度上与资产无关。然而，从高分辨率表面采样/捕获法线，然后将其变换到切线空间，更像是一种编码。因此，要反转最初捕获的法线场，用于解码的变换必须与用于编码的变换准确相反。这带来了一个问题，因为切线空间生成没有实现标准。每个法线贴图烘焙器都使用不同的实现，此外，也没有关于如何使用插值帧将法线变换到切线空间的标准。法线贴图烘焙器与用于渲染的像素着色器之间的这种不匹配而产生的数学错误会导致着色接缝。这些不必要的硬边在模型被光照/着色时会变得可见。它们会浪费艺术家/设计师的时间，并让他们感到沮丧。

Polycount详细地解释了程序角度的认知如何辅助构建美术工作流。出于切线空间的不确定性，像Handplane、NSpace这样的软件被开发出来用于将对象空间法线结合模型信息转化到切线空间，允许更加同步的工作流；但是现在MikkTSpace已经受到广泛支持，对象空间法线贴图显得不再必要。

## **2.6 用协方差矩阵偏转法向量**

空间中点最根本的表示是一个相对于原点的坐标值，但是这不足以将点彼此区分开，需要附加一些额外的数据，才能从点构建出正确的表面。假设有两个点彼此非常靠近，但是法线方向相反，分别落在一薄片的正面和背面，如果点上仅具有坐标信息，是无法将这两点区分开的。

在点云处理中，法线和曲率被称为表面描述符或几何描述符，表示围绕点的局部空间关系。表面描述符也称为特征，意为将两点区分开的特征。计算点的特征的方法是将点周围固定半径内的或k个邻近点包含进来，据此拟合出一个表面。根据表面法线的方向不同，顶点的歧义性就得到抵消。顶点的特征实际上是假想表面的特征，分布在同一个平面附近的点有类似的特征。顶点的特征表示的是邻近粒子的位置信息。（方便起见，以下称高模的点为粒子）

要决定一个顶点的法线，可以近似成拟合平面的问题。该平面是一个切平面，点云在切线方向上分布有最大的方差，在平面的法线方向上分布最紧凑。

在流体表面重建中，Yu和Turk提出了一种各向异性方法，各向异性矩阵**Gj**描述每个粒子j的邻近密度分布，本质上是个描述三维分布的协方差矩阵，如果粒子就在平面附近，那么密度会在法线方向上快速衰减，但切线方向衰减的慢，所以矩阵应该沿切线方向拉伸平滑核，沿法线方向压缩平滑核。在尖锐特征上，密度会在好几个方向上同时剧烈下降，矩阵应该收缩平滑核，使得能够捕获到尖锐特征。

在本文中，声明i表示某个片元，每个片元都代表光线与高模的一个交点，声明Pi是光线与高模表面的交点坐标；本文采用的公式是，其中是校正后的高模法线，着色器的输出；是校正前的高模法线，通过三角形重心坐标插值得到；是协方差矩阵的逆。

协方差矩阵有一种直观的无求和符号的表示方式，其中m是以Pi为中心，半径h内的所有粒子的总数。h是一个标量，代表搜索半径。M是一个矩阵，样本数是m，维度是3。

声明j表示周围的某个粒子，Pj表示该粒子的坐标。那么上述M矩阵的每一列都代表一个三维向量Vj，，Vj是Pj均值中心化的结果，是均值，也是样本的期望值，的设置方法是不固定的。

协方差矩阵的这种无求和符号的分解形式可以直接用于矩阵和向量的运算。，其中A是一个列向量，表示空间中的任意向量。

不过写成这种形式还是不够直观，且M矩阵不是方阵，Gi无法求逆，因此可以转换思路，将Gi写成特征分解的形式。，其中U是正交矩阵，每一列都是特征向量，特征向量之间彼此正交，这使得U-1=UT。D是一个对角矩阵，对角线上的元素是协方差矩阵的特征值。协方差矩阵的特征分解形式可以很方便地求逆，，与唯一的差别在于D矩阵求逆。因为D是一个对角矩阵，所以它的逆矩阵很方便地就是原对角线上元素的倒数。

根据特征值分解的结果，协方差矩阵乘向量可以写作，按照矩阵乘法的结合律，右边等于。

U的每一列都是特征向量，称协方差矩阵的三个特征向量为u，v，w，那么，这是向量点乘的形式，空间中的任意向量A分别与u，v，w点乘。

上式与D左乘的结果是，对角阵的每个对角元素会与矩阵对应行的元素相乘。，其中，，是D矩阵对角线上的元素。

上式与U左乘，可以得到最终结果：

更直观地，更符合代码习惯地，可以将上述公式写作：

这是一个空间变换公式，对于任意输入的世界空间向量A，首先在UT的变换下转到局部空间，得到三个坐标，，，随后在U的变换下，将局部坐标系变换回世界空间。

在主成分分析中，数据散布方差最大的方向称作数据的主成分。协方差矩阵的特征值是该方向的方差，特征向量是该方向。如果u是数据的主成分向量，那么方差比其他系数都大。在这个情况下，可以理解成，当局部空间变换回世界空间时，基向量并非单位向量，而是要乘上方差的大小，因此把局部向量变换回世界空间后它会向方差最大的方向靠拢，相比变换到局部空间前的原向量，新向量产生了偏转。

如果把协方差矩阵取逆，唯一产生变化的是系数。因为对角阵的逆是原元素的倒数，原先最大的系数在逆矩阵中取到最小，因此偏转得到的向量更排斥方差最大的方向，反而是趋近于方差最小的方向，即数据的法线方向。

如果输入的A本身就是重心坐标插值得到的法线，在经过逆矩阵变换后，法线就像数据集坐标定义的法线方向靠拢。当数据集采样个数够大的时候，细微的凹凸不平等瑕疵就被抹除了。

相比于其他PCA分析的方法，本文不需要分解特征值，免除了着色器内部的大量开销。根据上述公式，可以证明，即使不需要知道特征值和特征向量，直接将协方差矩阵的逆左乘法线向量，也能够执行PCA分析，只是最后得到的法线稍微会偏移一些，因为是借助系数来改变各分量之间的占比，但在特征值系数的辅助下，可以让输出结果基本上靠近法线。

在公式中，是重心坐标插值得到的法线，是协方差矩阵。可以通过这种方式计算协方差矩阵，W是计算径向向量对协方差的影响多少的各向同性平滑核，是以坐标点i为中心的邻近点的加权平均位置，是邻居点k的坐标，是坐标点i的坐标。与一般协方差矩阵的差别仅仅在于加权。

以i为中心的，周围粒子的加权平均位置的计算是，W是同之前使用的相同的平滑核函数，它实际上是一个将模型整体平滑收缩的函数，以便程序能够正确处理边缘尖锐特征。Yu和Turk的方法同时也在渲染的时候偏移了粒子位置，以让粒子分布得更加均匀。在本文的情况下，没有必要修改高模的实际顶点位置，因为在烘焙结束后，只有法线信息是保留的。

# 4 系统实现

系统实现阶段的目标是具体实现系统的功能，在这个阶段需要处理导入的模型，构建体素数据结构，向GPU传入缓冲，编写基本的烘焙算法，在基本烘焙算法的基础上扩展成PCA算法，此外，还需要编写各种辅助功能，比如在视口中观察模型，加载外部着色器文件，制定切线空间等。导入模型的时候，本系统使用了Assimp的JVM移植，这里面包括了glm包，可以直接进行向量数学运算。

## 4.1 体素

### **4.1.1 体素的设计**

体素是对空间的粗略划分方式，就像像素是图像的储存和定位的方式一样，体素将空间内一定距离之内的所有点看做是一个实体。体素化的过程是无数这样的实体组成了轮廓大致相似的新模型。在本文的情况下，体素将一定范围内的图元统一到一个体素对象中，在光线追踪的时候，光线首先测试这些粗略地模拟了原模型空间划分的体素，因为体素对原模型的空间划分只是粗略的模拟，所以程序很快就能抛弃大量不可能相交的图元，之后只对一小部分图元进行相交测试。本程序总是建议提高体素的密度，因为要得知坐标对应的体素，对坐标取整即可，查找体素的算法的时间复杂度是O（1），不随体素数量的变化而变化，但太大量的体素会导致难以预览，潜在地拖慢程序运行速度（要求光线迭代更多次才能到达表面），并且占用空间。

在本系统中，首先为高模计算整体的边界框，然后以体素的边长为除数将边界框划分为三维的若干个体素，得到每个方向上的体素的数量，数量之间可以不一致。

随后确定边界框左下角第一个体素的中心坐标start，将其作为起始，每次步进单元格尺寸的二分之一。

只需要记录每个体素的中心位置，在已知单元格尺寸的前提下，就可以推算体素的边界框。这么设计是为了能够输入顶点着色器，因为系统设计的要求包括提供用户浏览体素线框的能力。在光栅化渲染时，可以在视窗中观察体素的网格，这是通过几何着色器实现的。顶点着色器的运行次数是散点的总数，输入的顶点流是一系列体素立方体的中心。之后，散点输入到几何着色器（GS）中，在这里，以散点的空间坐标为中心，围绕中心程序化地生成线段，线段首尾相连形成长宽高相等的立方形线框。

### **4.1.2 多维数组的一维表示**

体素在计算机中实际的尺寸方式是一维的，按一定规律排列成一维数组。将三维体素放进一维数组的方式应该保持一致。

数组的定位方式是int id = x + y\*xcount + z\*ycount\*xcount; 其中 x ∈ [0 , xcount−1] , y ∈ [0 , ycount−1] , z ∈ [0 , zcount − 1]；x、y、z是范围内的空间中任意一点对应的体素的三维id。最终得到的一维id只与体素的三维id有关，与遍历顺序无关；寻找周围的体素的方法是将三维id沿某方向位移一格，将新的位置代入等式，得到新的一维id。

### **4.1.3 如何在GLSL中获取体素包括的三角形**

根据上文所述，三角形的体素化是未经删减的。因为位于三角形面的AABB内，一个完全未经过三角形面的体素可能包含了对它的引用。一个体素上可能引用了多个三角形，无论它们是否经过该体素。在构建体素的情况下，因为三角形的AABB包括多个体素，所以同一三角形的索引拷贝在多个体素中，而这些体素又含有对其他三角形的索引，结果是为了查询例如包含三角形2，4，5，6，7的体素，必须将这5个索引分配给该体素，而假如有另一个体素包含三角形2，4，5，12，13，三角形2，4，5索引被重复拷贝到另一处体素中。

综上所述，为了构造适合体素访问的一维索引数组，无可避免地占用远超实际三角形数量的内存，这块内存被分为体素数量个分区，每个分区记录被某个特定体素引用的所有三角形。这些分区是按自然数顺序排序的，分区的尺寸就是三角形索引的数量，任意分区的起点可以写作它之前所有分区的尺寸的和，其终点则是在起点的基础上加上自己的尺寸。体素结构体的GLSL代码：

|  |
| --- |
| struct Grid{  float bbmin[3];  float bbmax[3];  float start;  float end;  }; |

需要注意的是，SSBO不一定能够准确对齐到结构体的成员声明，这是由于SSBO在内存里的布局方式会将vec3类型对齐到vec4，占用16字节而非12字节，导致错位访问。可选的解决方案是让顶点在CPU中提前为第四位填充0，或者使用更紧凑的std430数组布局，在std430中，数组不被强制对齐到16字节，而是保留原有的尺寸。

要设计正确读取SSBO的结构体，必须清晰地知晓SSBO的内存布局规律。本文利用Nvidia的NsightGraphics工具，结合RenderDoc，确定了SSBO的内存布局，并最终设计了正确的对齐方式。

要储存在SSBO中的，除体素结构体之外，还要额外传入包括所有体素的全局BoundingBoxMin和BoundingBoxMax、单元格尺寸、XYZ每个方向上的体素数量，这些一共占10个浮点数的大小，随后用不定长数组归纳剩下的数据，每一组都是一个体素结构体，由代表左下角和右上角的两个向量，还有标识三角形索引起始位置的两个浮点型表示的整数组成。要让GLSL正确理解SSBO，必须使用对应的类型来区分这些紧密排布在内存中的数据，因为CPU和GPU通信的时候只能将数据流传递过去，无法共享这些变量名。程序员需要保证两方面对数据流的理解方式一致。Vertex Attribute就是这样工作的，程序员首先需要在CPU中声明一类顶点属性的尺寸和步长，比如法线占据vec3个空间，而如果在法线之前储存了uv，法线的步长就是vec2，这样才能让GPU理解某个区域的数据对应什么属性。

虽然三轴上体素的数量，内存里的起末点等数据理论上应该是int类型，但是为了方便，也就是只需要绑定一个SSBO就能导入所有这些数据，采取的方法是将它们都合并在同一个缓冲中。这个缓冲由于也同时包括vec3类型而必须是floatBuffer，虽然在JAVA中将int类型存入floatBuffer的过程包括自动的类型转化过程，但Interface block在shader里对缓冲进行读取的时候不包括自动的类型转化，硬件会尝试以整型格式来解析浮点类型的二进制数据，导致错误的结果。

### **4.1.4 数值稳定性**

存储在计算机中的小数具有精度问题，因为小数的二进制表示方法本质上是对0~1区间作二叉树以不断逼近所求小数。譬如0.2的二进制表示为0.001100110011…...，是无限循环小数。计算机中用于存储该数字的空间是有限的，因此它在计算机上存储的实际值不为0.2。计算机在舍入二进制数字时，遵循的原则是如果末尾是1就是进位，这导致存储的实际值可能大于字面值。IEEE 754浮点数标准的本质是将数字用科学计数法表示，因此随着指数不同，小数点位置不固定，即会浮动。但是Mantissa的容量是有限的，如果消耗太多用来表示整数部分，小数的精度会下降。浮点数的精度有多重不确定性。

尽管科学计数法同样可以精确地表示整数，但这受到指数位数的制约。IEEE754浮点标准表达int32整数的时候，存在精度问题。浮点数的有效位由Mantissa决定，这是23位，但是int32有32位精度，浮点数无法表示缺少的9位，除非它们正好都是0。这就是整数转化成浮点数时的严重舍入误差，导致数值跳跃，但这类问题通常不在考虑范围内，因为大部分模型的尺寸都不会导致指数大于23（）。类似的问题是开放大世界游戏导致浮点数的精度全部用来表示整数部分而没有留给小数的精度。如果要解决此类问题，可以设置数个区划中心，然后计算模型的相对坐标。

浮点数加减的时候，指数更小的会向更大的指数对齐，抛弃多余的位数，在左边填充0，这同样导致舍入误差。

本文采用的解决方法是添加容差、约束上下限、优化体素画线的算法。

## **4.2 平滑法线**

### **4.2.1 平滑法线**

一个三角形上的三个点总是在同一平面上。取这平面内的任意两条向量，叉乘得到的即是平面的法线。由此可见，法线是对平面定义的，而单独的点并不存在真正意义上的法线。但是顶点着色器对点运行，而片元着色器是对点的重心坐标插值结果运行。因此有必要将面级别的法线转化到点级别。

平坦法线指的是将三角面的法线均一分配给它的三个顶点，无视面之间的连续性，也称面法线；平滑法线指的是如此不断将三角形的法线分配给顶点，最终合并所有处于相同世界坐标的顶点，统计其上所有法线的平均值，即得平滑法线，也称顶点法线。

这种算法没有考虑划分多边形产生的非结构性的对角线。引擎总是在三角形上运行，因此期望输入的网格都是三角化的。三角化网格会产生额外的面，这些面处在同一个平面上，但是被重复计算，重复的权重添加到共用的顶点上，造成偏移。

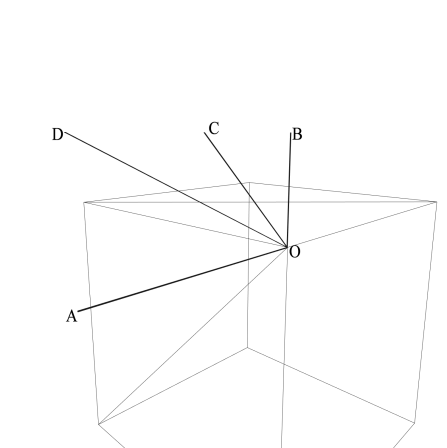


图4-1 重复的面权重

比如有一个连接了三个三角面的顶点O，如图4-1所示，法线的长度是2，因为它连接了两个三角形，累加了两份相同的面法线；法线的长度是1，因为它只连接了一个三角形。平滑得到的错误法线用表示，期望得到的形状用表示。

对于此种情况，给出的解决方案是计算点O参与构成的三角形中角O的角度，对面法线进行角度的加权，的长度是，的长度是，平均得到，将角度纳入考虑因素可以很好地抵消掉重复的贡献。

在本文的情况下，需要对高低模都进行平滑。这是因为Zbrush导出的高模没有光滑组，所以需要手动重建光滑组；低模需要平滑的原因在于光线方向和TBN矩阵，不平滑的光追方向会留下大量黑边，同时实践表明平坦法线构建的TBN矩阵容易留下接缝。

### **4.2.2 向量的哈希**

哈希映射是键和值之间的映射，其中键被哈希函数转化成地址，地址大部分时候都是独一无二的，以键的内容和类型为特征，换句话说，哈希值是键的特征表示，因此哈希函数在加密和完整性校验领域也有作用。以查找为应用目的的哈希算法要做到输出的结果足够平均，尽可能降低冲突的概率，这样一来，每个键都一一对应于一个地址，而如果将值储存在该地址上，就可以用时间复杂度为O(1)的哈希函数找到任意大小的哈希表中的任意一个值，而无需逐个遍历和比较，适合超大体量的数据的处理。虽然哈希函数应该尽可能保证输出的分布的均匀性，但哈希值和物理地址有着本质的区别，在数据查找和比对时，通常希望两个输入即使分配在不同的地址上，如果它们的内容相同，那么就输出同样的哈希值。

Java GLM为Vec3实现了HashCode函数，阅读.class文件的字节码，参照JVM Spec 2.6.2，按照运算符将操作数栈里最顶端的两个值出栈，运算后再将结果入栈的规则，可以看出其算法是31 \* 31 \* Float.hashCode(X) + 31 \* Float.hashCode(Y) + Float.hashCode(Z)。

其中Float.hashCode 是java.lang.Float的内置方法，其原理是将浮点数二进制直接读成整型。利用Online Hex Converter和Floating Point to Hex Converter可以得知Float.hashCode将浮点数以Big Endian (ABCD) 规则转化成Uint32，举例：浮点数1.0000f的十六进制表示为0x3f800000，等于整数1065353216。因此，可以得出结论：如果两个Vec3值完全相同，即X、Y、Z分别相等，那么，由于float的哈希函数只与float本身的二进制有关，两个完全相同的float的哈希值也相同，这两个Vec3的哈希值也是完全相同的。

这种恒定的等价关系对于设计至关重要。因为Assimp导入时会炸开模型上的每一个点，相当于将点的数据原封不动地拷贝了多份。只要这些拷贝体都指向同一个哈希值，就可以将这些炸开的点合并起来。

### **4.2.3 顶点的合并**

顶点可以认为是包括一个世界空间坐标Vec3，一个法线Vec3，一个颜色Vec3，一个UV Vec2的结构体，而一个点连接多个面，作为面之间的共享点，比如正方形的八个角，每个顶点都连接着三个面，而法线的根本计算方法是取面的两条向量算面法线，因此一个点上可能有三种不同的法线。如果一个点上有不同的法线、颜色或UV，那么点实际上就会被炸开，分成多个点，每个图元索引不同的点，产生不同的渲染效果。

模型的储存和处理的本质是点处理，只有点才能确定面，点定义了面在哪个平面上以及三角形的边界。将面级别的法线属性转移到点上，是模型处理的常见操作。

作为多个面的共享点的顶点，其同时承担了这些面朝向各异的法线。DCC软件通常会在交互层将这些点合并成一个并提供给用户，来保证操作上的流畅和方便。这是逻辑上的点，但物理内存中的点可能数量更多。图形渲染的时候，点被限定只能拥有一条法线，依据法线数量新增多个处在同一世界空间坐标的不同点，它们之间的区别就是承担不同朝向的法线，图元会索引这些新建的点。

渲染的过程是在点之间插值的过程，如果图元索引的是同一个顶点，它们承担的法线朝向一样，插值的结果是曲面，否则会产生接缝，因为接近该点时每个面的临界值都不一样。

平滑法线是一条令曲面看起来更有说服力的法线，其选择方法是不固定的，但应该让结果看起来尽可能符合直觉，因此有面积加权和角度加权等算法。

能够将这些仅法线不同的点合并成一个的原因，就在于它们享有相同的位置，因此哈希表的键选择为位置向量。

### **4.2.4 坐标映射法线的哈希表**

这张哈希表在Vec3和Vec3之间建立映射关系，它的用途是在合并顶点的时候计算最终这唯一且独特的一点应该拥有怎么样的法线，该计算是基于合并前的所有在该位置上的点的法线的。

在合并顶点的过程中需要对哈希表进行增改查。需要一个函数Get，它的参数是键，返回键对应的值，这个Get函数应该处理该键尚不存在时的情况，以及该键已经存在时的情况。若键尚不存在，则函数返回Vec3(0,0,0)，但在这个过程中，并没有向哈希表写入该键值对，函数只是在查询无果的情况下自行设置了一个值。

在三角形遍历循环内，遍历的对象的三角形的三个点，每到达一个点，便将它的坐标传给Get函数，检查是否存在相同坐标的顶点。

无论Get函数是否查询到了对应的键值对，函数都会返回一个Vec3代表查询的法线结果。随后，程序根据当前顶点的角度对三角形的平坦法线加权，这样会得到一个模不为1的向量。这个向量与先前返回的Vec3相加，把当前键所对应的值设置为加法结果，依然不归一化，因为该非1的模长最终可以用来调整法线在某个方向上的权重。

整个过程就是把当前三角形的法线传递到了顶点上，这个顶点可能原本就拥有其他三角形的法线信息。如果先前返回的是零向量，那么此时是在哈希表中为该顶点新建了一条记录。

### **4.2.5 重新索引顶点**

由于合并了顶点，三角形图元在合并前所引用的顶点在合并后的顶点数组中找不到，以及易位，因此在合并顶点的过程中，需要重建顶点索引。

这张坐标映射索引的哈希表，在所有这些坐标对应的顶点被写入之前，表内是空无一物的，而每写入一个顶点，哈希表的尺寸都会+1。因此，可以直接用哈希表的尺寸来作为当前点的索引，这正好相当于当前点在表中的位置。比如在空表情况下，新增的顶点的索引是0（尺寸为0）；如果向已经有一条记录的表中添加一个顶点，那么该新顶点的索引是1（尺寸为1）。新建的索引正好是顶点按放入顺序在表内的自然数位置。

## **4.3 生成切线**

### **4.3.1 施密特正交化**

法线垂直于切平面。平坦法线的切平面就是三角形本身，但求平滑法线的切平面有一定难度，因为若干组原本互相垂直的法切线的平均法切线不一定再相互垂直。

导致丧失正交性的另一个因素在于片元着色器的插值。当光栅化的时候，屏幕对被三角形覆盖的像素计算重心坐标插值，将三角形的三个顶点的属性线性混合到三角形内任意一点，原本相互垂直的两条向量在插值之后不一定仍然保留有正交关系。

当且仅当时，矩阵TBN是正交矩阵，正交矩阵的逆矩阵是它的转置。非正交的TBN矩阵导致无法正确进行逆变换，产生错误的结果。一种常见的强制正交化的方法是施密特正交化（Gram-Schmidt process），代码如下，具体思路是将两个向量拉到同一海拔上，这时这两向量相减的结果就在切平面上：

|  |
| --- |
| vec3 sTa = normalize(SmoothTangent);  vec3 sNo = normalize(SmoothNormal);  float LengthTN = dot(sTa, sNo);  vec3 alignedN = LengthTN \* sNo;  vec3 newT = sTa - alignedN;  mat3 oTBN = mat3(normalize(newT), normalize(cross(newT, sNo)), sNo); |

### **4.3.2 切线的选择**

对三角形定义一个切平面，三角形上的任何一点都可以用切平面的一组二维基向量表示。切线的选择问题实际上就是这组基向量的选择问题。

切线空间法线可以理解成高模法线相对低模法线的偏移，这份偏移量可以写成，因此如果T、B、N定义得不同，而解码同一份偏移量数据，重构的世界空间法线也不同。

一个切平面上有无数条候选切线，切线空间法线只有在解码切线选择得与编码切线一致的时候才能被正确转换到世界空间。从高分辨率模型表面采样的切线转换到切线空间的过程近似于编码，用于解码回原始向量的变换矩阵是编码矩阵的逆矩阵。

切线与副切线的计算与纹理坐标并没有必然联系，事实上三角形的一条边就可以直接作为切线，只是无法保证顶点的顺序在其他软件里也是一样的。采用纹理坐标的好处是纹理坐标在不同的软件中保持一致，这样能够在不同的软件中得到统一的切线空间。切线在CPU中计算，并作为顶点属性（Vertex Attribute）输入着色器，因为着色器只能计算得到平坦切线（利用几何着色器或片元求导），GPU的并行性使它无法计算平滑切线。

已知该三角形三个顶点的位置坐标：, 以及对应的纹理坐标：。 定义三角形的两条边为，对应的纹理坐标差值：。有如下关系式：**，可解得T、B向量。**

**解得的T、B向量是垂直于三角形的平坦切线。实践证明平坦切线虽然更容易在不同应用中统一，即只需要在几何着色器或片元着色器中计算面法线，就能正确解码，但会造成面之间存在细微接缝，如图4-2所示。因此需要加权平均成平滑切线。本文按角加权。**

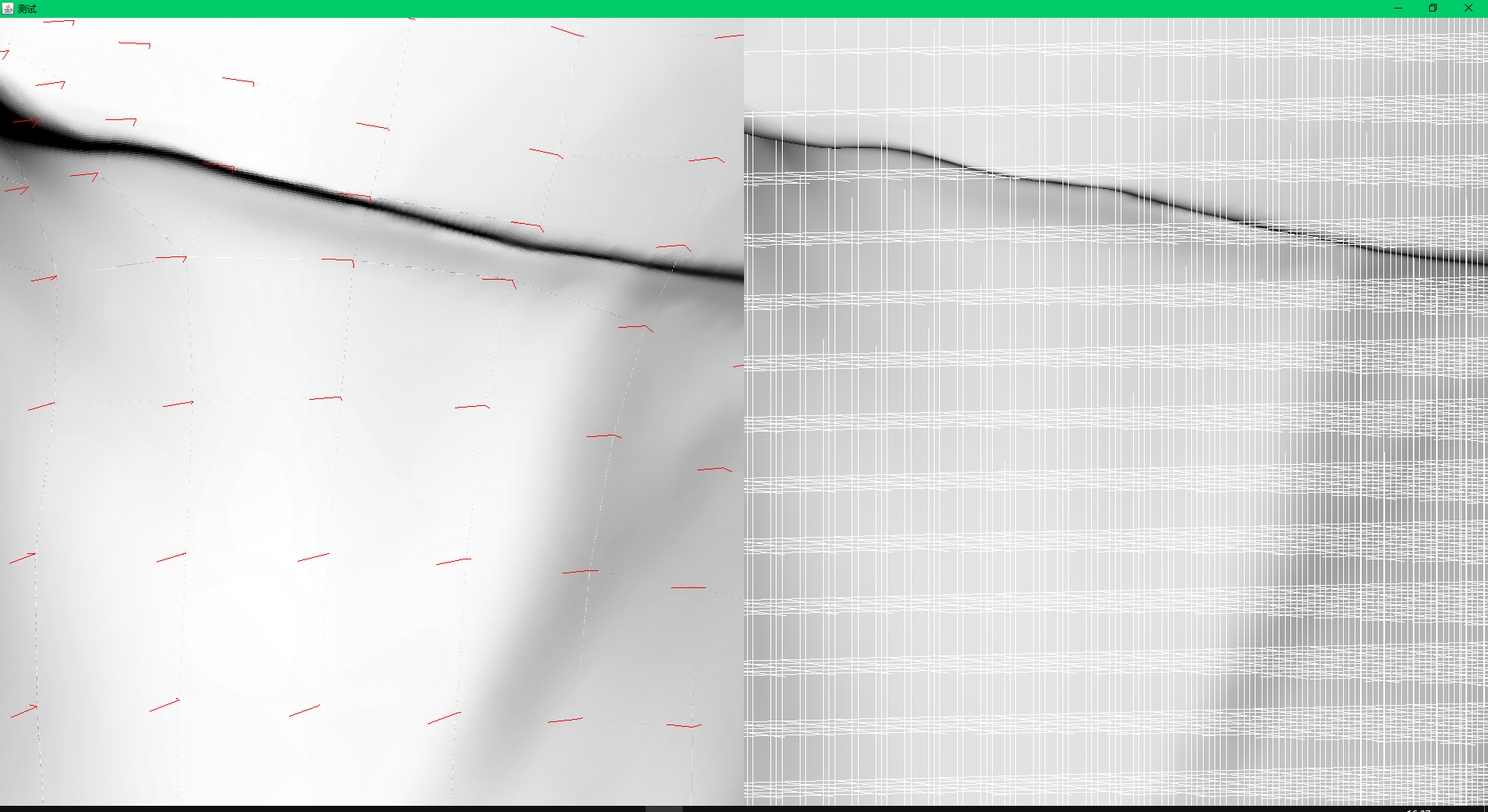
****

图4-2 面之间的接缝

**参数也可以不取纹理坐标，甚至可以取常量，只要E0和E1不平行，就不影响切线和副切线的计算，但法线贴图的观感会比较奇怪，但是不影响光照的计算，因为切线空间只要保持在应用程序中一致，即保证编码和解码的一致性，就可获得正确效果。**

**切线在多边形上的插值同样服从5.2节所描述的规律，但由于编码和解码的一致性，最终渲染效果不带对角线瑕疵，尽管法线贴图上可以明显看到有对角线瑕疵。**

### ****4.3.3 左右手坐标系****

**副切线可以通过得到，没有必要作为额外的顶点属性输入。在这里存在左右手坐标系的区别，左右手坐标系的唯一区别就在于副切线前是否带有负号。在编码和解码过程中，副切线的符号性要统一。在实际应用中，用户应该通过反转法线贴图的y通道来反转坐标轴方向。**

## **4.4 基础变换**

本系统也做了模型预览的功能。模型的预览功能的一般做法是通过三个矩阵WorldMatrix, ViewMatrix, ProjectionMatrix完成。WorldMatrix储存模型的平移、旋转、缩放变换信息，每次运行VS着色器，都会做世界矩阵乘法，将模型从局部空间变换到世界空间，如果WorldMatrix已经被应用在输入的数据上，这些变换就变成顶点的属性，乘上单位世界矩阵后和原来的效果相同，这就是Maya的“冻结变换”的原理。ViewMatrix是摄像机的旋转的逆矩阵，用于将世界空间变换到摄像机空间，这可以理解为以摄像机为参考系世界如何反向旋转，计算得到的Z就是场景的深度。ProjectionMatrix提供透视焦点、正确的裁剪范围和深度曲线，本身是一系列相似三角形运算的总结，因此分母总是代表距离，距离内的顶点会落入NDC空间。

## **4.5 加速结构**

文章采用的加速结构为均匀网格，它将空间划分为个大小均一的立方体，如图4-3所示。本应用提供了在视口中观察体素线框的支持，这样用户可以更直观地检查体素网格的大小是否合适，本文建议设置的体素大小容纳30个左右的顶点。图4-3所用的是开发前期使用的测试模型，是无数小球的阵列，整体上稍微倾斜了一些，用于测试光线是否被阻挡以及能否正确步进深度。

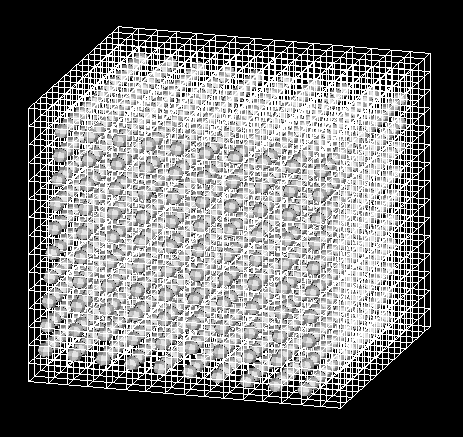
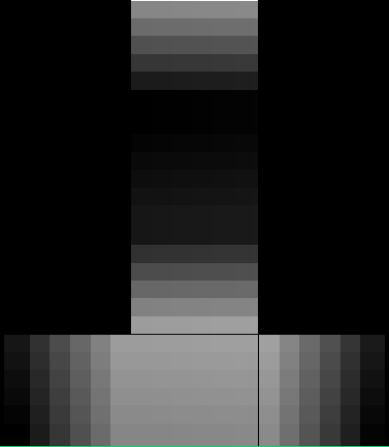
 

图4-3 几何着色器预览的体素 图4-4 片元着色器中的体素（以不同亮度代表）

图4-4展示了光线投射时识别到的体素，可以看到它们具有明显的规律，呈现有方向性的渐变。这一特性在调试时非常有用。图形调试具有难度，光线追踪调试起来更困难，但是选择合适的变量可以立体化数据，直观了解空间的划分结构，大幅降低调试难度。

体素的建立方式是，需要对高模的每一个三角形图元进行计算，检查它落在哪一个立方体内，在立方体中建立对其的引用。

将三角形放入体素的方法类似于立体空间的光栅化，遍历三角形，根据三个顶点算出最小值和最大值，共同组成轴对齐包围盒（AABB）。有了AABB之后也就知道了每个轴上需要遍历的体素从哪里开始，在哪里结束。遍历所覆盖的所有体素，向体素写入该三角形的索引。

在体素中建立图元索引的Java伪代码如下所示：

|  |
| --- |
| vec3 count = vec3(xcount,ycount,zcount);  vec3 boundmin = min(min(pos1, pos2), pos0);//pos0, pos1, pos2是当前三角形的三个点  vec3 boundmax = max(max(pos1, pos2), pos0);  vec3 distLeft = boundmin - Grids.GlobalMin;  vec3 distRight = boundmax – Grids.GlobalMin;  ivec3 idLeft = clamp(floor(distLeft/Grids.UniformGridSize),0,count-1);  ivec3 idRight = clamp(floor(distRight/Grids.UniformGridSize),0,count-1);  for X from idLeft.x to idRight.x{  for Y from idLeft.y to idRight.y{  for Z from idLeft.z to idRight.z{  voxels.get(X+Y\*xcount+Z\*xcount\*ycount).add(i);//i是当前三角形的索引  }  }  } |

Slab Method用于求解射线和立方体相交的问题。在本文的情况下，立方体即空间中的体素，射线即从低模包裹框的位置向内发射的光线，由于体素都是轴对齐的立方体，可以用此方法分析。射线的参数方程为，代表射线进入立方体的位置，代表射线穿透立方体后离开的位置。射线进入立方体的时候，一定穿过超平面XY、ZY、XZ，顺序不一定，但一定是穿过最后一个超平面才真正进入立方体，因此求最大的t；射线离开立方体的时候，一旦穿过第一个超平面就算作离开立方体，因此求最小的t。对于射线平行于某一坐标轴的情况，按照AABB的角点在Origin的哪一侧，分别产生正无限和负无限，与正无限求较小值和与负无限求较大值都会返回有意义的分量。t值的求法如下：

|  |
| --- |
| vec3 VecTmax = (bbmax - Orig)/Dir;  vec3 VecTmin = (bbmin – Orig)/Dir;  //这两行代码保证Tmin<Tmax，且负无限被强制求最大值，正无限被强制求最小值  VecTmax = max(VecTmin, VecTmax);  VecTmin = min(VecTmin, VecTmax);  float Tmax = min(VecTmax.x, min(VecTmax.y,VecTmax.z));  float Tmin = max(VecTmin.x, max(VecTmin.y,VecTmin.z)); |

将所有体素的集合视作一个巨大的立方体，对该立方体使用Slab Method分析，分析结果即为光路进入立方体的位置，在这个位置上它与某一体素发生碰撞。为了具体确定该体素，其代码如下。

|  |
| --- |
| vec3 VTmin = (globalmin - Orig)/Dir;  vec3 VTmax = (globalmax - Orig)/Dir;  VTmin = min(VTmax, VTmin);  float Tmin = max( VTmin.x ,max( VTmin.y,VTmin.z) );  vec3 where = Orig + Dir \* Tmin;//where可能因为精度误差而略小于global min  vec3 limit = vec3(xcount-1,ycount-1,zcount-1);//floor(50)=50，但实际上想要49  ivec3 where\_id;  ivec3 shootAbroad = max(ivec3(0), ivec3(min(floor((where - globalmin)/ unitsize),limit)));  ivec3 shootInside = max(ivec3(0), ivec3(min(floor((Orig - globalmin)/ unitsize),limit)));  where\_id = LeftIsLess(Orig,globalmin) && LeftIsLess(globalmax,Orig) ? shootInside : shootAbroad; |

这里使用一个三目运算符判断发射源Origin是否在总体AABB之外，如果是，则查找最初进入的体素。如果否，则直接找到Orig所在的体素，即查找低模坐落在哪个体素中，对该体素内的图元进行检测。

低模作为光子发射源的追踪方法也在Woo等人的论文中提到：初始化阶段首先识别射线原点所在的体素，如果射线原点在网格之外，则找到射线进入网格的点，并取其相邻的体素。

在使用此方法前，本文总是计算shootAbroad，但这种方法很快被证明受限制于迭代次数，不同的模型大小和体素大小会导致高模表面到总体AABB之间的空隙被不同数量的空体素填充，这个数量会相当大，不仅在剧烈降低算法性能的同时，还产生两个问题：

(1)内置设定的迭代深度上限可能小于到高模表面的实际距离，导致光子未产生碰撞便退出迭代；

(2)当要追踪的模型表面被其他凸起结构遮挡时，光子产生的最近的碰撞不在目标表面上。

该算法可以理解成作直线与总体AABB相交的数学问题，它的优势在于它不需要额外提供包裹框，也不以输入的位置为发射源，当输入的位置落在高模表面之下时，依然能够正确追踪到目标表面；然而该算法只在简单的流体空间上工作良好，无法适应复杂的实际情况。

使用低模作为光子发射源后，由于通常高模和低模在空间上高度相关，往往只需要找一次Origin所在的体素而无需再继续步进射线，就能得到光线在高模上的交点。这样做大幅度优化了光子迭代次数过多的问题，并且避免产生迭代距离小于实际距离而找不到相交的情况。

如果用户能够确信要追踪的模型不会在任何地方发生自遮挡，又不想专门设置包裹框，可以将 shootInside替换为 shootAbroad，这能解决一些空缺问题。

在定位到与之相交的体素后，遍历体素中包括的所有三角形，用光线方向矢量测试与每个三角形相交与否。如果没有产生相交，则步进找下一个体素。本文最初采用的步进方法是计算光线射出体素时的Tmax值，将该值略微扩大，表明光线略微超出该体素的边界，那么再对该新位置取floor，就得到新格子的id。代码如下：

|  |
| --- |
| float E = 0.0001;  vec3 NextgridEntry = Orig + Dir\*(Tmax+E);  vec3 id = floor((NextgridEntry – globalmin) / unitsize);  startX = int(id.x); startY = int(id.y); startZ = int(id.z); |

取不同的E值带来不同的渲染瑕疵。当E过大时，光线可能跳过体素。当E过小时，可能是出于对齐指数时的舍入误差，光线原地不动。同时，实践表明，这个算法不可避免地产生微量噪点。

因此，本文优化了光线步进的算法。新的算法参考了3D DDA算法，3D DDA算法也是体素查询时通用的方法，光线沿Tmax的最小分量所表示的坐标轴不断锯齿状运动，在空间中经过紧邻的体素，不会产生跳跃，也不会原地停滞。在运动过程中不保存旧的Tmax，而是在抵达下一个体素后，重新计算光线离开该体素的Tmax，随后在重复Tmax的最小值分类讨论过程。这么设计的初衷是为了减少float运算时累积的误差，虽然优化效果并不明显。在使用此算法后，渲染出图不再有噪点。代码如下：

|  |
| --- |
| if(GVTmax.x<GVTmax.z){  if(GVTmax.x<GVTmax.y){  startX += stepX;  }  else{  startY += stepY;  }  }  else{  if(GVTmax.z<GVTmax.y){  startZ += stepZ;  }  else{  startY += stepY;  }  } |

## **4.6 相交算法**

给出光线方向和起点，程序需要计算光线会在何处与平面相交。算法基本思路是，把上文中提到的三角形被扩展成无限的平面，测试射线和平面的相交，然后判断相交点是否落在三角形外，如果不是，则算作射线与一个三角形存在相交。统计射线与全部三角形的相交结果，找到距离光线起点最近的交点。

在相交测试的过程中储存当前三角形的id，在得到交点后，对id所指向的三角形做重心坐标插值。该三角形id也用于获取邻接三角形的信息。

三角形的三个点P1，P2，P3必然在一个平面内，依据此三角形可以做出平面的参数方程，取任意的a，b值，可以在平面内取到任意一点P。经过光线起点O，方向为D的直线的参数方程是，取任意的n值，可以在直线上取到任意一点P’。Möller-Trumbore 交集算法联立这两公式，写作线性方程组的形式。利用克莱默法则列出公式，将行列式写成向量混合积的格式，运用轮换对称性化简成如下代码所示的公式，解得t、u、v，其中t代入直线的参数方程计算交点，u、v可以用来进一步判断点是否在三角形内。测试光线与平面的交点以及交点是否在三角形外的GLSL代码如下：

|  |
| --- |
| vec3 e1 = C-A;  vec3 e2 = B-A;  vec3 res = Orig-A;  double det = dot(e1,cross(e2,Dir));  double deta = dot(res,cross(e2,Dir));  double detb = dot(e1,cross(res,Dir));  double deti = dot(e1,cross(e2,res));  double a = deta/det;  double b = detb/det;  double c = -deti/det;  vec3 IntersectedAt = OrigPS + float(c)\*Dir;  bool IsInside = false;  if(a>=0. && a<=1. && b>=0. && b<=1. && a+b<=1.){  IsInside = true;  } |

## **4.7 法向估算**

首先需要建立容纳高模顶点的体素。体素只是帮助快速寻找顶点的工具，实际上的运算过程将每个体素内的全部顶点加起来综合考虑。构建点体素的过程如之前一样，但是不需要额外记录包围盒，因为给定相同的跨度和尺寸，体素的分布都是相同的，标记包含面的体素的序号也能够是标记同样空间上的点的体素的序号，但是要注意，同样序号的体素，可能包含面图元但不包含点图元，除非这个体素刚好在三角形的三个顶点上。这个特性会导致产生网格状的斑驳，因为每个像素都对应低模上的一点，而低模上的一点对应高模上的光线交点，但交点大部分时间都落在三角形的中央，很少会靠近边缘。当交点靠近边缘的时候，在交点所在的体素内直接就能找到粒子，但对于落在中心的情况，则在当前体素内找不到粒子，这就导致渲染的结果呈现大量散点色块。扩大搜索半径可以解决这个问题。

点的体素化可以很直观地做到，首先把计算点相对边界框的偏移，然后把坐标除以体素大小，再向下取整到第一个整数，这是体素的边界或中心位置，最后记录下该体素[8]。

在循环粒子前，还需要计算重心坐标，就如常规的重心坐标插值着色器一样，这是为了获得足以代表当前点的法线。因为每个片元都应该代表一条不同的法线，一个三角形上却有三个顶点也就是三条法线，需要用一种足够有说服力的方法来选择需要变换的初始法线。

循环粒子的方法类似之前在CPU端将图元放入体素的过程，只不过这次要体素化的对象是球体，程序会检测半径内的粒子。流程和之前一样，找到左下角和右上角，遍历体素。这个过程对每个检测到的交点都会进行两次，第一次找到平均坐标，第二次根据平均坐标计算矩阵。球体的半径越大，找的体素就越多，这会影响运行速度，因此程序设置了一个阈值，如果半径超过体素尺寸的15倍，就直接返回。

接下来这部分参考了Yu和Turk的各向异性算法，对每个粒子执行的第一部分的操作是算出平均值。首先找到它到交点的距离，如果距离小于搜索半径，则进行接下来的操作。这个限制的存在原因是必须保证权重是0~1之间的数，否则会着色错误，这也是为什么不能将距离判定写作体素的中心之差，因为体素的中心不能准确代表粒子的位置。在确定距离小于半径后，检测粒子的法线，如果和交点的法线（重心坐标插值得到）的夹角大于等于90度，就将权重设为0，这是为了避免背面也会影响权重。

计算平均值的详细代码如下：

|  |
| --- |
| float weight\_function = 1;  vec3 weight\_pos = where;  for(int a = x\_begin; a <= x\_end; a++){          for(int b = y\_begin; b <= y\_end; b++){              for(int c = z\_begin; c <= z\_end; c++){                  VertexSection which = sections[a + b \* xcount + c \* xcount \* ycount];                  for(int i= which.start;i< which.end;i++){                      VertexStruct pX = vertices[vertexref[i]];                      vec3 X = vec3(pX.pos[0],pX.pos[1],pX.pos[2]);                      float mag = length(X - where);                      if(mag<=SearchRadius){                          vec3 angle = vec3(pX.normal[0], pX.normal[1], pX.normal[2]);                          float angle\_weight = dot(angle,p\_normal);                          float weight = 1 - pow(mag / (SearchRadius), 3);                          weight \*= clamp(angle\_weight ,0,1);                          weight\_function += weight;                          weight\_pos += weight \* X;                      }                  }              }          }  }  vec3 mean\_Pos = weight\_pos / weight\_function; |

在获得平均位置之后，对每个粒子执行第二部分操作。重复如上所示的循环和条件分支，用相同的方法再次计算权重，构造加权协方差矩阵，代码如下：

|  |
| --- |
| for(int a = int(x\_begin); a <= int(x\_end); a++){  for(int b = int(y\_begin); b <= int(y\_end); b++){  for(int c = int(z\_begin); c <= int(z\_end); c++){          VertexSection which = sections[a + b \* int(xcount) + c \* int(xcount) \* int(ycount)];              for(int i= int(which.start);i< int(which.end);i++){              VertexStruct pX = vertices[vertexref[i]];                  vec3 X = vec3(pX.pos[0],pX.pos[1],pX.pos[2]);                  float mag = length(X - where);                  if(mag<=SearchRadius){                  vec3 angle = vec3(pX.normal[0], pX.normal[1], pX.normal[2]);                    float angle\_weight = dot(angle,p\_normal);                       float weight = 1 - pow(mag / (SearchRadius), Power);                       weight \*= clamp(angle\_weight,0,1);                       vec3 dist = X - mean\_Pos;                       covariance += mat3(dist\*dist.x\*weight,                               dist\*dist.y\*weight,                               dist\*dist.z\*weight);                  }               }  }  }  }  covariance /= weight\_function;  covariance = inverse(covariance); |

整个算法过程由两个多重循环组成，对性能负担是非常大的。构造n维协方差矩阵的过程，实际上是统计向量的n个分量之间的二维协方差。二维协方差定义如下，k是附近粒子总数。轮流从x，y，z中取出两个分量，统计重复的，有9种组合方式，形成对称的协方差矩阵，对角线是各个维度上的方差。weight控制当前粒子与均值之间的差异对最终统计的协方差作多少贡献。

****5**** 系统测试与优化

**系统测试是软件开发流程中的关键步骤，由于图形开发的抽象性和难调试性，其重要性不言而喻。系统测试旨在确保系统的在不同输入和环境下的可靠性、稳定性，通过不断解决系统测试中出现的问题来提升功能的全面性，并提升设计要求的符合性，以确保系统各模块功能完善、运行稳定，并满足既定设计要求。通过全面且细致的系统测试。**

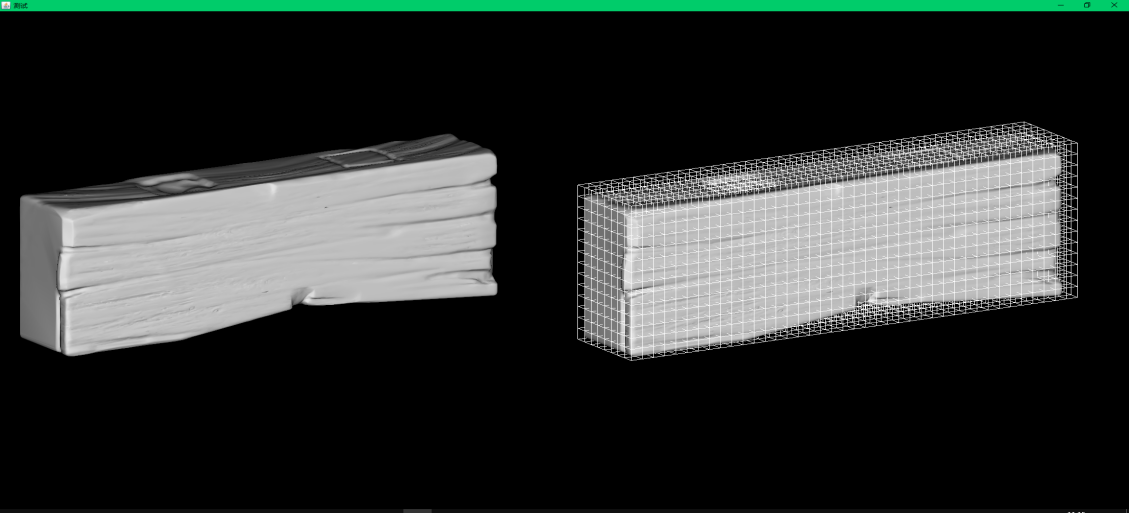
**在这部分，本文进行了四种测试。首先是测试基本的交互功能和基本的法线烘焙功能，然后是测试不同输入对结果的影响，这两步测试综合优化了基本的烘焙器框架。PCA重建法线的烘焙方法建立在基本的烘焙器框架之上。第三步修改了获取空间中邻域的方式，第四步是针对完善的PCA烘焙系统进行测试。**

## ****5.1 基础烘焙功能****

### ****5.1.1 启动软件****

**启动软件后，指定模型路径，完成GL初始化，初始化的过程中模型被加载进来，同时生成体素和平滑法线，如图5-1所示。初始化之后，窗口就开始进行渲染循环，在这个过程中用户可以在视窗中检查平滑后的高模，检查体素对高模的包裹情况，并且用鼠标和键盘移动模型，旋转摄像机，或者放大缩小模型，以适应不同缩放程度的模型。左侧显示的是高模本身，右侧是带包裹框的高模，两者都是平滑着色。平滑着色下，多边形本身的边缘痕迹被隐藏，模拟出连续的曲面。**

**用户可以使用上下箭头放大屏幕图像，这将提供能清楚的视野，让用户可以近距离观察体素是如何穿过模型表面，以及哪些顶点会被归到该体素中，这可以让用户更好地理解法线烘焙背后的逻辑。**

********

**图5-1 浏览模型和体素**

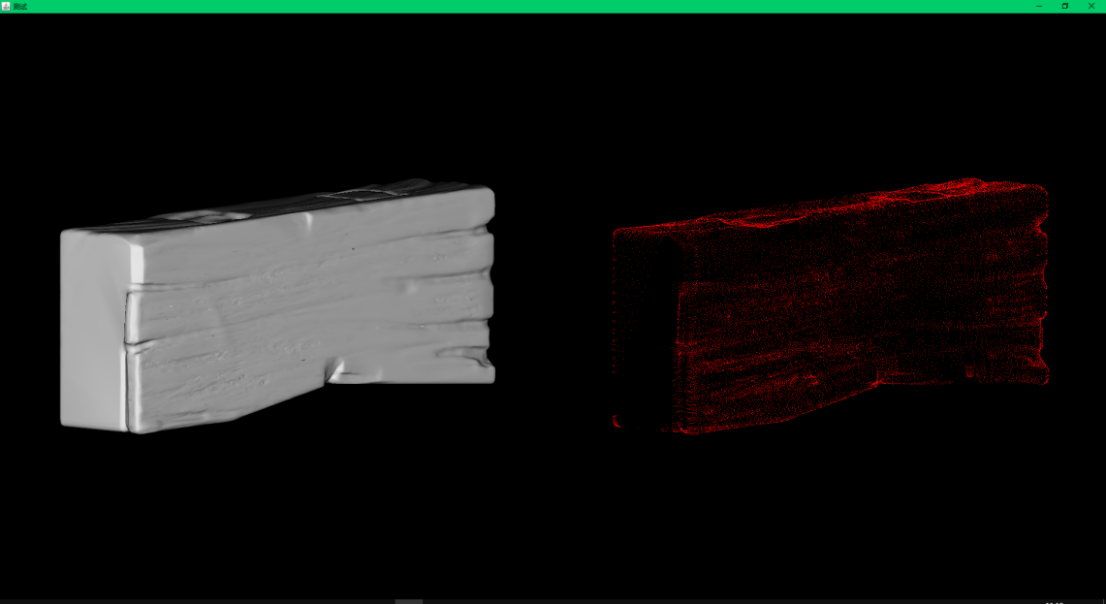
### ****5.1.2 不同层级****

**用户可以选择从点、线、面层级来观察高模。图5-2显示的是仅显示点的模型。左侧是默认渲染状态，右侧只显示点层级，隐藏了线框、面和体素。体素中如果包括太多顶点，运行速度会大幅降低，因此用户应该观察体素大小和点密度来选择恰当的体素尺寸，并在选择了合适的尺寸之后开始烘焙。**

**用户可以在渲染循环中随时更改显示的点的尺寸，渲染器将及时更新。这是因为JoGL让Display方法始终处于运行状态，因此能够及时响应到渲染状态的变化。**

**用户同样可以绑定自定义着色器，比如通过在顶点上储存法线，渲染可以响应光照变化的点集。模型缩放得较小的时候，带光照的点集看起来就像完整的模型。**

**用户可以选择是否将面渲染成线，当启用时，硬件在对面图元执行背面剔除后将多边形光栅化成线框，这提供非常清晰的视图，让用户更好地观察模型的拓扑。**



**图5-2 浏览模型的不同图元层级**

### ****5.1.3 烘焙贴图****

**错误展开的UV可能导致光线追踪时间大量增长，几乎死循环，因此烘焙前需要检查UV。用户可以进入到UV模式，这将告诉程序开始停止渲染高模预览，而是开始渲染低模的UV展开图，用户可以检查是否存在重叠或者变形，并且返回到DCC软件中进行修改。**

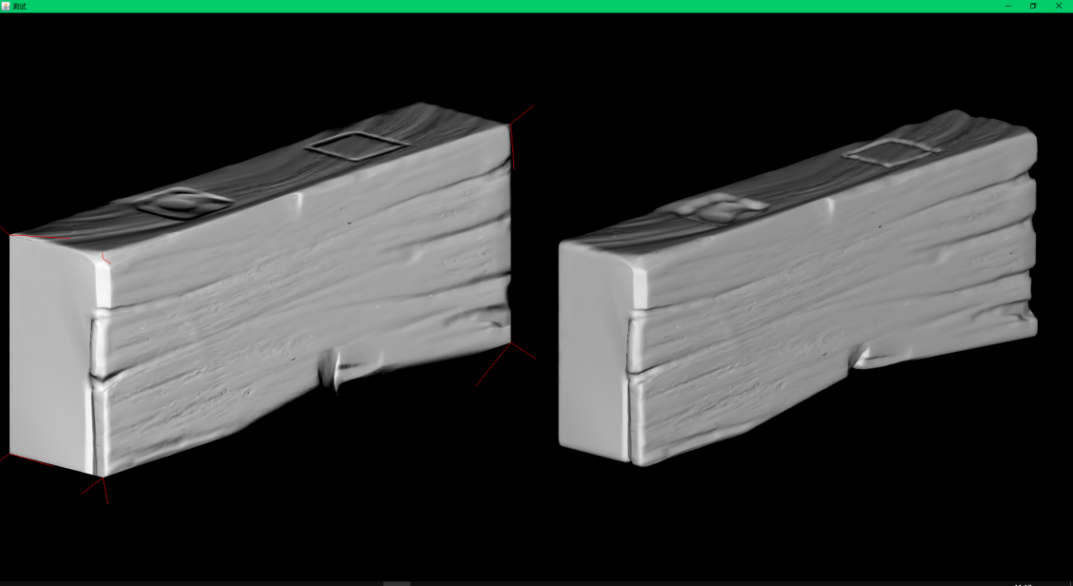
**当用户决定烘焙输出的时候，FPS动画器会被暂停，避免产生卡顿。根据用户划分的体素大小，烘焙可能会很快，也可能需要一点时间。烘焙的结果会在视窗中提供预览，这是低分辨率的。用户可以选择是否在表面附加线框，观察具体对应到低模的哪里。实际纹理文件导出在根目录下，分辨率更高，用户可以直观地看到哪里存在瑕疵，并且做出对应的修改。**

**测试过程中常见的问题是包裹框在高模内部，这时候可以简单地将追踪方向设置为完全经过体素，能够简单地解决问题。**

### ****5.1.4 观察低模****

**用户可以切换显示模式，切换后，视窗左边将展示应用法线贴图后低模的样子。如果用户没有选择贴图路径，软件会提示。默认的路径就在根目录下，与输出的法线贴图同名。用户可以应用自己的法线贴图，只需要修改路径。**

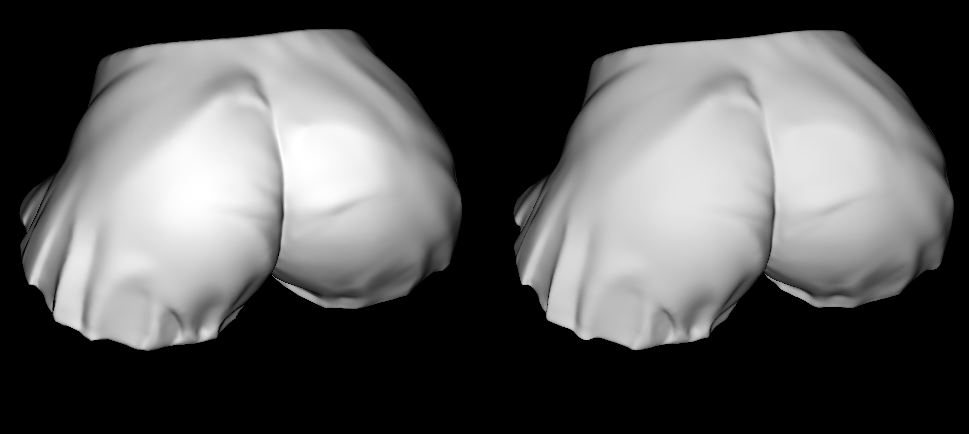
**在烘焙完成法线后，用户可以关闭右侧的体素显示，清晰地观察高模的光线特征，检查法线贴图是否还原了高模的光影。在显示低模的时候，同时也会显示顶点的切线空间，用户可以选择隐藏。图5-3同时显示应用了法线贴图的低模和平滑渲染的高模，可以看到两者的细节极其相似。**

****

**图5-3 观察效果**

### ****5.1.5 对象空间法线****

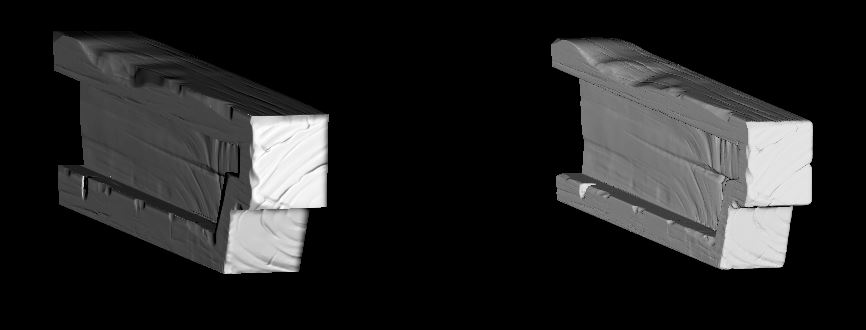
**本工具也支持导出世界空间法线，用户可以简单地切换并选择导出世界空间法线。世界空间法线是彩色的，因为它直接储存世界空间的高模法线信息。本渲染器也提供对世界空间法线的支持，能够正确显示，如图5-4所示。世界空间法线的好处是不需要关心低模的法线信息，但弊端是难以压缩，难以复用。一个常见的误解是世界空间法线无法运动或者变形，也不能镜像，但这些都可以通过在着色器里提供额外支持而克服。使用对象空间法线的一个显而易见的好处就是完全忽略了低模的着色方式，比如当应用对象空间法线贴图的时候，对低模进行细分，不影响法线的计算。当应用法线贴图的时候，细分低模会导致切线变化，使得法线发生扭转。对象空间法线和细分修改器的结合使用有助于在保留完全相同的结构光影的同时柔化轮廓边缘，这是切线空间法线贴图做不到的。**

****

**图5-4 对象空间法线**

### ****5.1.6 烘焙效果评价****

法线贴图的效果如图5-5所示，低模（左）充分地记录了高分辨率模型（右）的表面法线属性，在光照下呈现类似的光影效果。低模呈现的光影效果有些微不同之处，这是因为法线贴图不会更改顶点位置，模型轮廓不会被改变，所以表面的微结构没有遮挡关系，从侧面依然能看到明显的低模痕迹，这是法线贴图无法解决的问题，但是从光影的划分，结构凹凸起伏上而言，烘焙的法线效果近似于高分辨率模型的平滑光照效果。渲染结果没有光线提前结束迭代产生的杂色，没有光线提前碰撞导致的错位，没有精度浮动导致的噪点。该法线烘焙系统具备应用意义，基本算法的正确性得到验证，表示本文可以展开下一步更深入的研究。



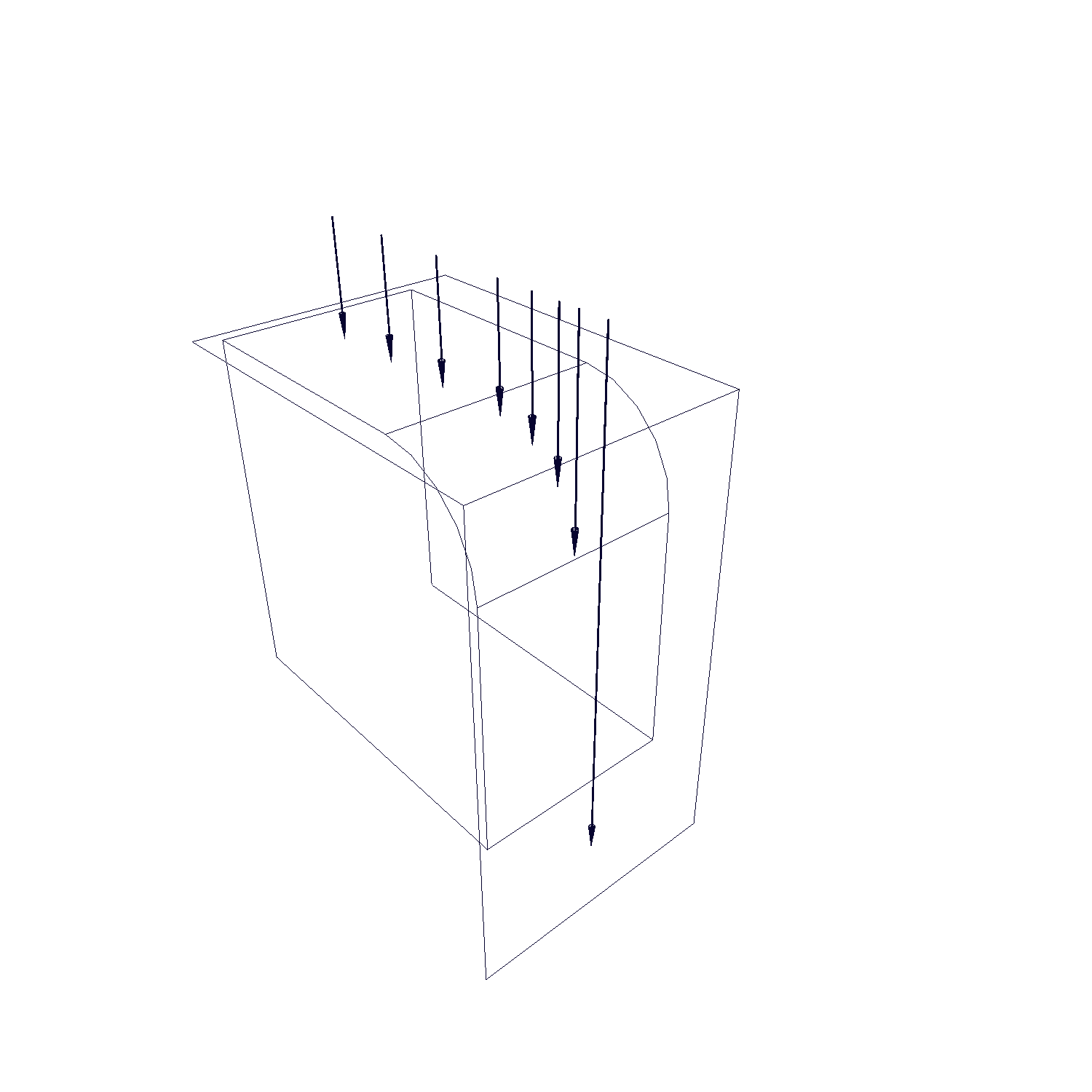
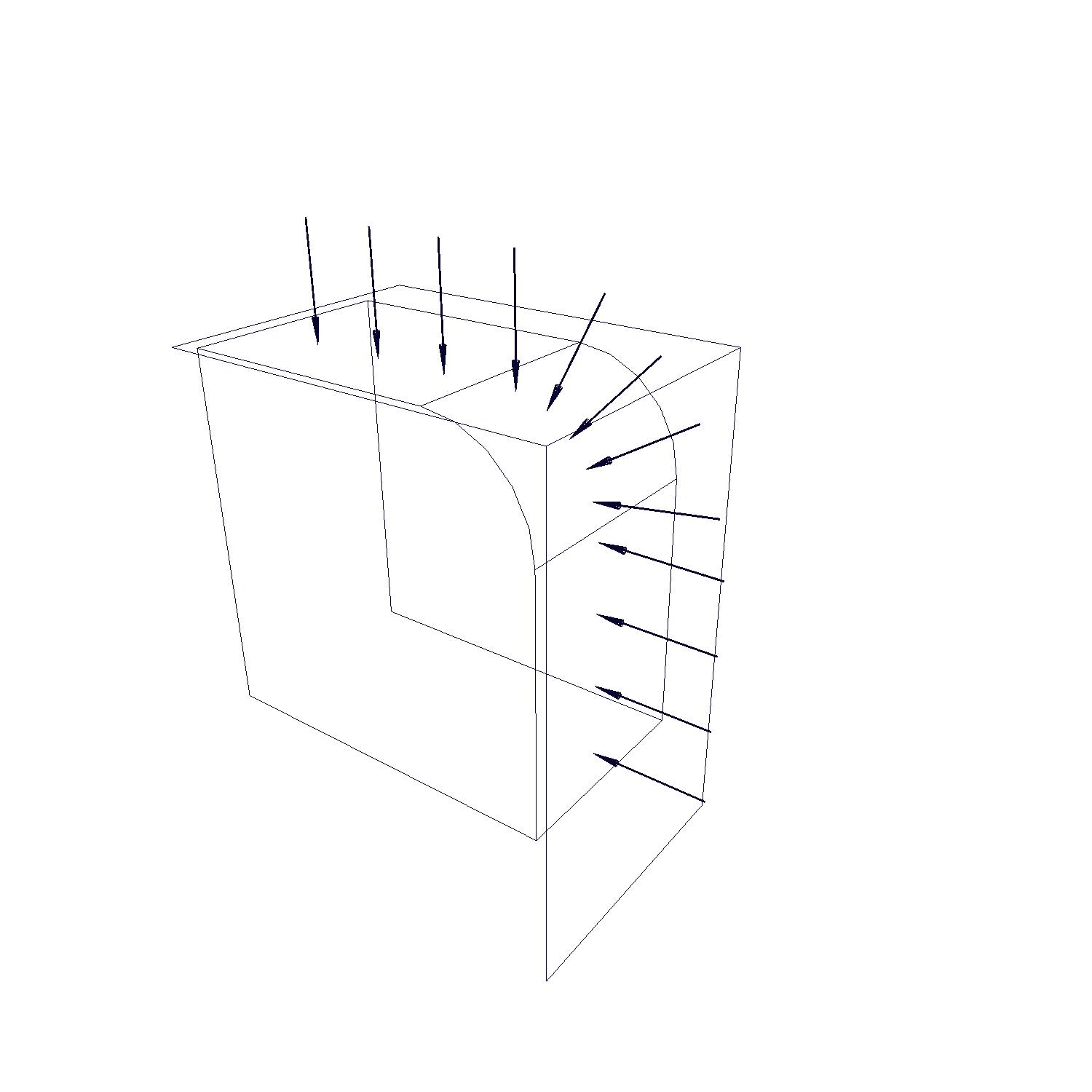
**图5-5 测试模型**

## ****5.2 对追踪方向的测试****

**在实际测试的基础上，本文总结了垂直于低模表面的光追方向的缺陷：低模通常会膨胀，如果光线的发射点位于膨胀超出高模体积的部分，那么它发射的光线将永不膨胀，导致UV的这个部位是黑色，在实际运行结果上，为贴图四周添加了黑框。这种现象背后的原理可以被形象地理解为，最后一条射线没有落在需要追踪的模型上，由于光的直线传播，它一路深入直到离开所有体素，或者直到迭代次数耗尽，这使得低模对应的表面是黑色，如图5-6所示。**

**平行于平坦法线的光线会造成交错错位和射空。交错错位指的是，两个垂直的表面各自发射与各自垂直的光线，结果形成十字交错，分别把错误的空间捕获到自己的表面上；平行于平滑法线的光线导致遮挡错位和鱼眼变形。遮挡错位指的是，光线从错误的角度射入，被其他结构阻挡。鱼眼变形最明显的例子就是把立方体盒子的八个角设为顶点法线，对立方体内部的物体追踪，得到的结果有明显的变形。**

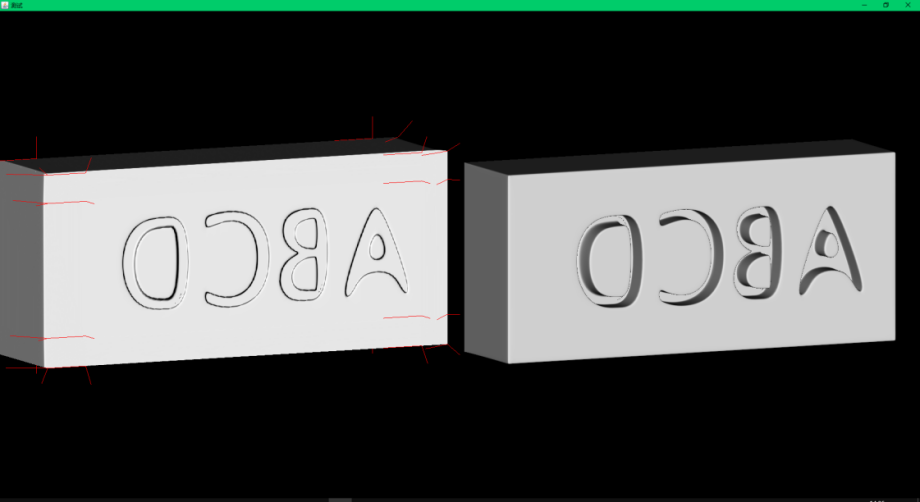
**光线方向的影响可以形象地比喻为，将低模包裹在高模表面的时候，低模上任意一点如何映射到高模上任意一点。光线是一点到另一点的映射关系，也可以称之为影响的是程序如何将低模包裹在高模上。在实际应用中，光线总应该捕获高模表面，产生平滑过渡，如图5-7所示。虽然这种方式可以正确捕获表面，用户也需要有意识地避免高模和低模之间产生太大的空隙，因为法线贴图只是对光影的模拟，无法真正改变轮廓。**

** **

**图5-6 不能正确捕获体积 图5-7 正确捕获体积**

**通常希望的效果是仅平滑边缘，而中心处平坦，这可以通过卡线来实现，如图5-8里的铭牌。正确的追踪效果如图5-8所示。**

**卡线是指通过在边缘上添加额外的线来约束模型平滑细分后顶点的偏移量。卡线的意义在于（1）获得更光滑更连续的曲面效果；（2）在提高分辨率的同时保持轮廓。卡线能够约束平滑范围，强制光线按垂直表面的方向追踪，这导致当环线过于逼近边缘时，模型重现图5-6的黑边问题。**

****

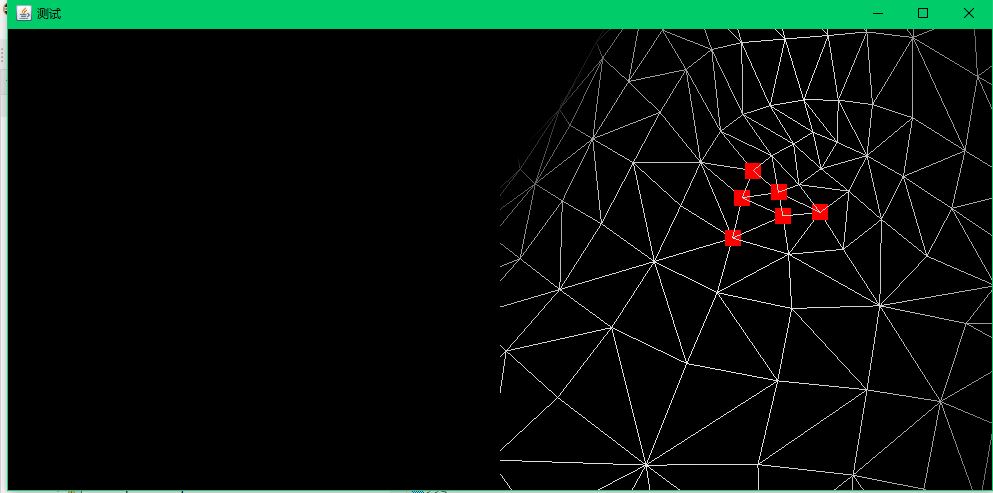
**图5-8 垂直于表面捕获**

## ****5.3 优化访问图元的方式****

**在最初设计系统的时候，考虑到实际的模型已经确定好了拓扑之间的连接性，并不能被简单地看作是三维重建中的点云或者流体渲染中的粒子，然后像元球一样将一些靠近的顶点融合起来，所以采用的是沿拓扑结构方向寻找邻接点的方法，如图5-9所示。**

**如果只是简单地按球半径搜索，很容易使得实际上不相连的，具有完全相反的法线方向的顶点错误地混合在空间距离更近的顶点上，比如茶壶的壁和它的手柄。**

**为了找到一个面在拓扑上的点邻域，本系统最初的设计是储存点的邻接点，建立了点索引面，面索引点的关系，每个点都记录它参与构成了哪些面，逐一迭代这些面，找到构成面的顶点，再重复该过程，最终记录初始面对点邻域的映射关系，这些点在拓扑上是首尾相连的。在光线投射的时候，找到相交的三角形后，就能直接用一个for语句逐个循环这个三角形映射的点集。**

****

**图5-9 邻接顶点**

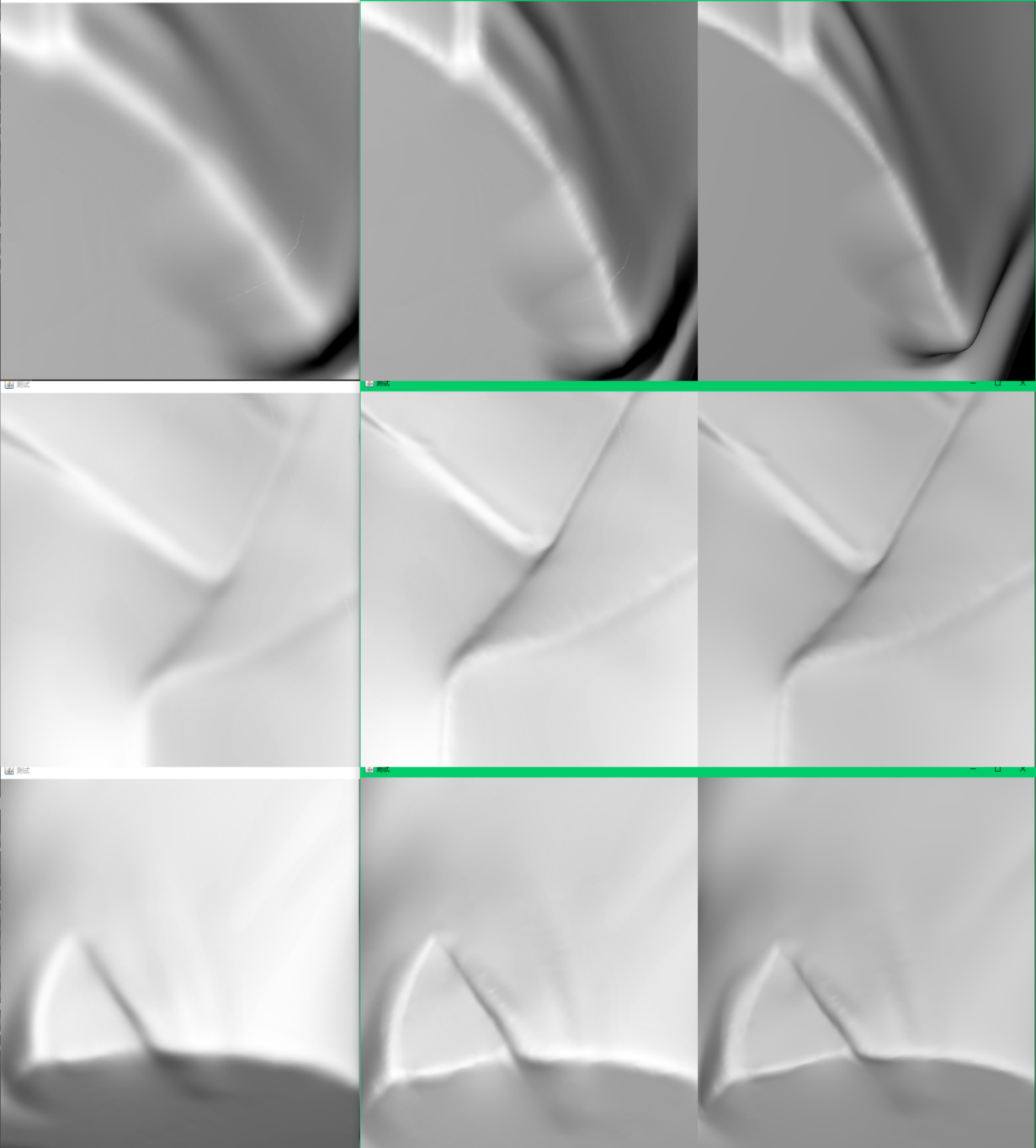
**虽然在GPU中的运行速度较快，但在CPU中构建邻域的过程非常耗时，该方法具有很大的局限性，无法应用于高面数模型，但对于低模而言，倒不如直接在GPU里遍历所有的点，完全跳过构建邻域的步骤。**

**在CPU端记录每个顶点的拓扑邻域过于低效，耗时太久，不断有重复访问已经经过的面的无效操作，而且找到的顶点经常重合故需要合并处理，这种迭代方式完全无法在高多边形上运行。因此本文最终不采用拓扑方案，而是将所有顶点放在均匀网格中，按球半径搜索邻近点，用法线方向排除反面，并成功将该算法用在高面数模型上，这么做可以支持搜索远多于先前的邻近粒子，而只会有短暂的未响应时间，充分发挥了GPU的并行能力，大幅度提高了运行速度。**

## ****5.4 实际应用测试****

**在常规重心坐标插值法烘焙的法线贴图中，贴图很完整地保留了高模具有的种种瑕疵。观察5-11和5-12右边高模的光影分布，这就等同于常规方法烘焙的法线上带有的瑕疵，但本文尝试在烘焙的过程中刻意地重新校正高模的法线，以消除模型自带的瑕疵。本文的研究很成功。**

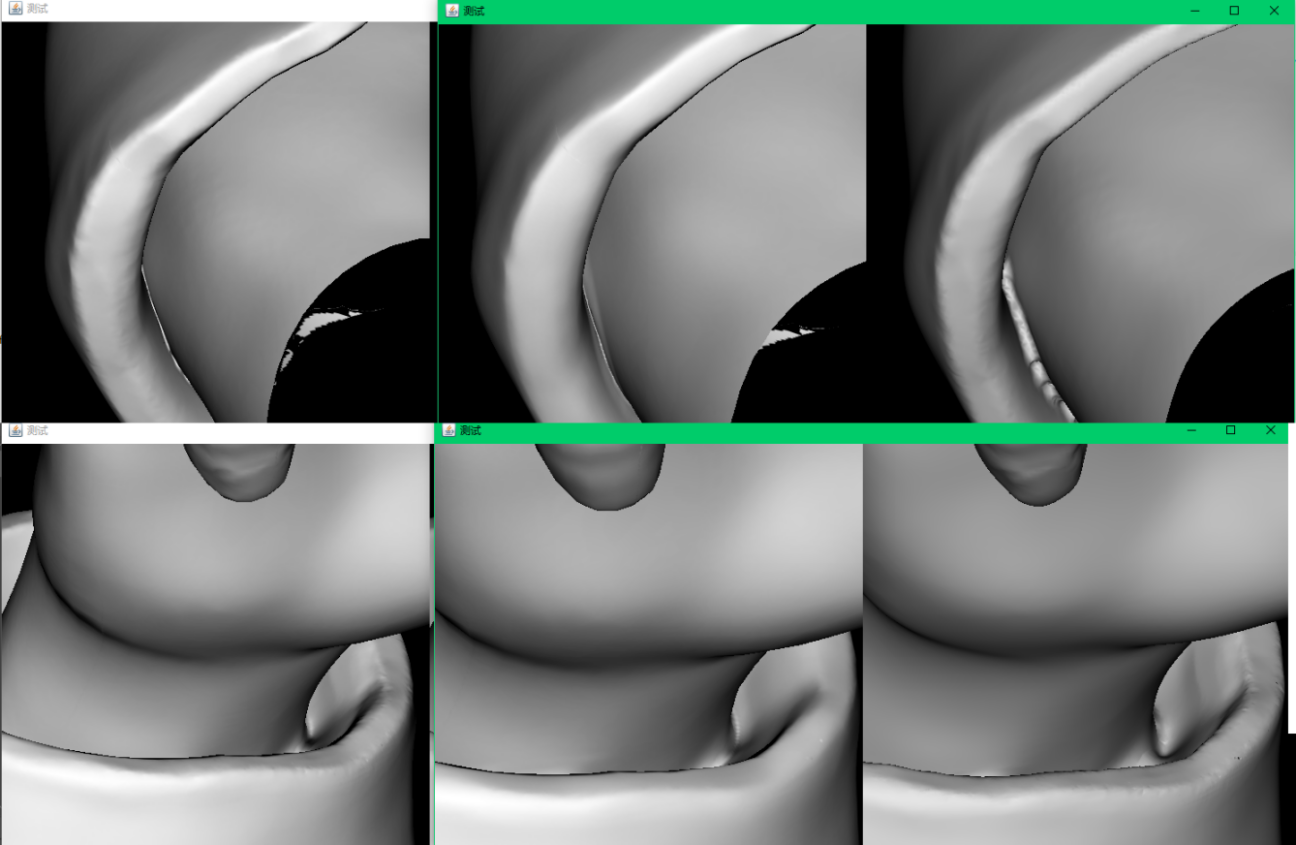
**Zbrush导出的模型不可避免地总是有各种瑕疵，比如不连续的笔触、Dynamesh导致的扭结、不顺着雕刻笔触走的拓扑产生的四边形拉伸。本系统能够在法线烘焙的过程中，以一种符合模型造型的方式模糊法线，能够保留清晰边缘，而在模糊的过程中，能消去瑕疵。**

****

**图5-10消除瑕疵**

**图5-10展示的是本文的方法与传统重心坐标插值烘焙方法的对比。每一行都显示了一部分瑕疵，第一行和第三行的图像显示的瑕疵是不连续的笔触使得模型表面产生鱼鳞焊一样的连续瑕疵，第二行的图像展示的是在平滑之后保留清晰边缘的能力。左边显示的是应用了带有PCA分析的重建法线烘焙方法后的法线着色，中间是重心坐标插值的结果，右边的是高模本身的形态，以平滑着色的方式渲染。可以观察到，重心坐标插值算法的精度非常高，和高模本身的光影几乎一样，但这也导致如实复制瑕疵，导致表面不再工整，在一些地方产生扭曲，但在PCA烘焙的结果里，瑕疵已经被消除掉了，而模型的整体光影还保持着形状。**

**图5-11展示了本文的方法如何消除更严重的瑕疵。每一行都截取了模型上不同的部分，左边是重心坐标插值的结果，中间是PCA分析的结果，右边是高模本身的形态。可以看到模型本身有着严重的不连续问题，而在PCA分析之后已经全部看不到了。这些问题是较低的面数和不沿着结构排布的拓扑导致的，这样的问题在不改变拓扑的前提下是无法改变的，但重拓扑的流程过于繁琐，本文提出的PCA方法就非常适合在烘焙软件中实时地，简易且快速地消除这种缺陷，产生更好的视觉效果。**

****

**图5-11消除更严重的瑕疵**

6 总结与展望

本文章研究了法线贴图背后的原理，使用OpenGL的Java绑定编写了包含实验性法线烘焙功能的渲染器。本文提出了一种全新的，富有创造性的对法线进行加权和模糊的方式，提供了一种在烘焙器里实时解决烘焙缺陷的高效方案。

**目前该系统的缺憾是无法手动限制要模糊的区域，这可以通过贴图来交互地指定。如果本系统能够限制模糊的范围和强度，就能允许用户单独把具有明显瑕疵的地方模糊掉，而保持剩下的大部分区域依然清晰，这将极大地提升烘焙法线的效率。这项功能需要在后续的更新中逐步实现。**

**系统有两个可以进一步研究的方向。（1）各向异性不仅能用于平滑表面，它的本质很适合用来强化差异，可以在本系统的框架上继续改进，使用户能够单独拉大某个区域的法线对比度，这可能支持更有创造性的风格化渲染；（2）目前系统要求高模输入具有相当高和平均的点密度， 在大部分情况下，这不是问题，因为无论是Dynamesh还是Zremesher就能产生符合要求的点云，但是如果高模经过减面处理，测试表面平滑效果就差了很多，因此有必要研究怎么在低密度下依然准确地捕获形状。**