

西安交通大学计算机图形学实验文档

Loop 曲面细分算法的实现

作者：李昊东

组织：计算机图形学课题组

时间：November 6, 2022



目录

第 1 章 Loop 曲面细分	2
1.1 理论：用有限分割操作逼近曲面	2
1.2 数据结构：半边网格	5
第 2 章 实现思路和细节	7
2.1 Scotty3D 中半边网格的实现与操作	7
2.1.1 数据类型	7
2.1.2 网格操作	8
2.2 细分流程	10
2.2.1 4-1 细分	10
2.2.2 位置调整	11
2.3 调试处理几何结构的代码	11
第 3 章 要求与验证	14
3.1 实验要求	14
3.2 实验结果自动测试	15
3.3 向助教寻求帮助	16
参考文献	17

序

这部分实验将在 Scotty3D 中实现 Loop 细分——根据一个“粗糙”模型自动生成“精细”模型的过程。

本文档分为三章：

- **Loop 曲面细分**简单介绍什么是曲面细分、Loop 细分的步骤和半边网格
- **实现思路和细节**则介绍 Scotty3D 中进行几何操作所需的数据类型，并附上一些编程和调试的技巧
- **要求与验证**介绍实验要求和进行自动测试的方法

请大家按顺序认真阅读文档，这可以极大减少实验时遇到的困难。

第 1 章 Loop 曲面细分

1.1 理论：用有限分割操作逼近曲面

问题 1.1 曲面的表示形式

我们测量或设计一个物体时只能描述有限的点，而物体的表面经常是光滑的曲面，这就带来一个问题：如何用有限的点描述一个光滑曲面？

解 良好的描述方法应该满足：

- 曲面的形状应该和给定的点有直观联系，曲面“看起来”就是由这些点决定的。
- 从微分学角度，曲面有一定的连续性，比如导数连续或者曲率连续。
- 如果修改某个点的位置，最好只影响这个点周围区域曲面的形状，而不导致整个曲面变形。

而 B 样条曲面就能满足这些要求。

曲面比较复杂，我们先从曲线说起。

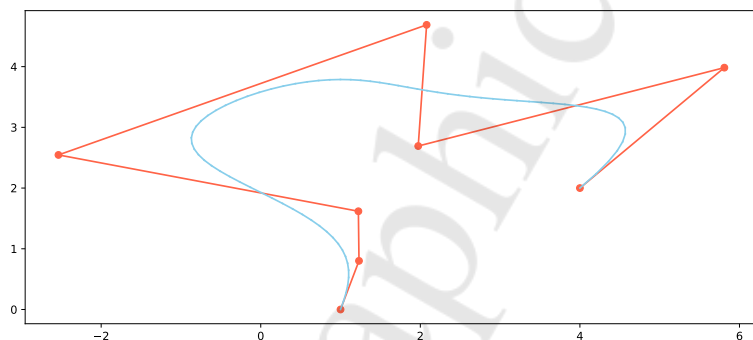


图 1.1: 一条由八个点控制的四阶 B 样条曲线

根据课上学到的知识，B 样条曲线只要达到三阶就是 C^1 连续的（处处可导），并且挪动控制点只修改邻近部分曲线的形状，很适合用来作图。

那么曲面呢？想象一组同阶 B 样条曲线，每一条都有相同个数的控制点。我们知道，B 样条曲线是由参数方程 $\mathbf{p} = \sum_{i=1}^n \mathbf{p}_i B_{i,k}(t)$, $t \in [a, b]$ 描述的，一个参数值 t 对应曲线上的一个点，一组曲线上就有一组点。用这一组点作控制点，又可以得到一条 B 样条曲线。当 t 取遍 $[a, b]$ 上的所有值时，这条曲线扫过的就是一个 B 样条曲面。

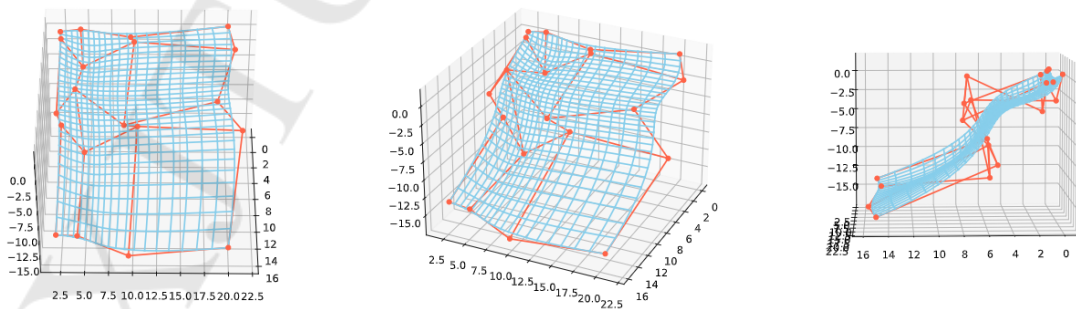


图 1.2: 从不同视角观察一个 B 样条曲面及其控制点

一个方向 k 阶、另一个方向 l 阶的 B 样条曲面参数方程是：

$$p = \sum_{i=1}^m \sum_{j=1}^n p_{i,j} B_{i,p}(u) B_{j,l}(v) \quad (1.1)$$

图形学以处理三角形网格为主，相应的 B 样条曲面也是定义在仿射坐标系上的，形式和上面的方形网格一致。表示和计算这样的基函数都有点麻烦，拟合就更麻烦了。

回忆一下，B 样条基函数 $B_{i,k}(t)$ 是递归定义的，图 1.3 中蓝色的低阶基函数线性组合得到橙色的高阶基函数。

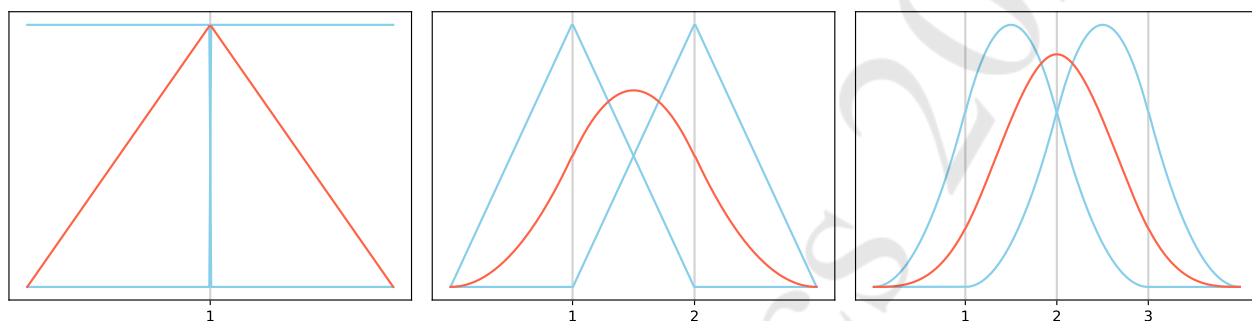


图 1.3: 一至三阶的 B 样条基函数

问题 1.2 更简单的计算方法

既然最终渲染的通常是**直线段**而非曲线，那么曲线（曲面）本身不过是一种中间表示罢了。是不是有更简单的方法，可以不经拟合就求出 B 样条曲线（曲面）的**离散近似**呢？

解 答案是有的，这就是细分曲线（曲面）。

像图 1.4 这样的折线，如果我们想把它的“棱角”打磨得光滑一些，很自然的想法是把凸起的角切平，从而产生更多的线段（边）。

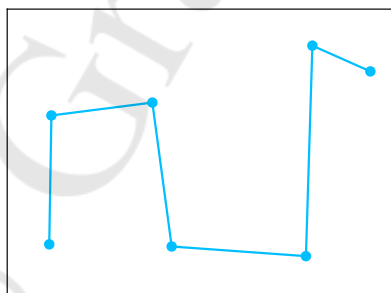


图 1.4: 一条随意的折线

不断重复“把角切平”的过程，就会让多边形越来越光滑。重复的次数趋于正无穷时，直观上就能想象到结果会趋近于一条光滑曲线。

“切角”的数学描述就是增加顶点和边。图 1.5 所示的迭代过程中，新顶点的位置是相邻顶点的线性组合。借助矩阵特征根可以证明，迭代结果收敛到**三次 B 样条曲线**。类似地，用合适的方法给三维网格“切角”，就能任意逼近 B 样条曲面。

细分网格比细分折线要麻烦不少，因为折线可以是一个有序点列，网格顶点却是不能排序的，这产生了复杂的邻接关系和拓扑结构。Loop 细分是一种细分三角网格的方法，可以将任意三角网格细分逼近到网格顶点控制的双向四阶 B 样条曲面 [1] (C^2 连续，也就是导函数处处连续)。

Loop 细分的过程有两步：

1. 将每个三角形三边中点连起来，从而将它划分成四个新三角形。（产生了三个新顶点和三条边）

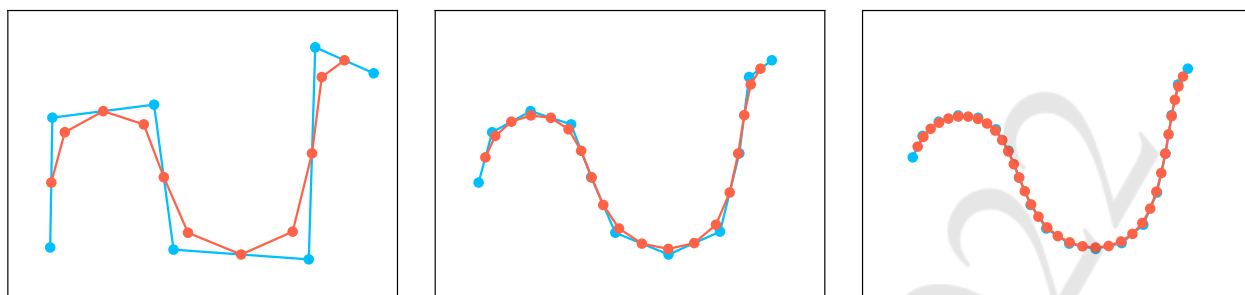


图 1.5: 细分操作的迭代过程

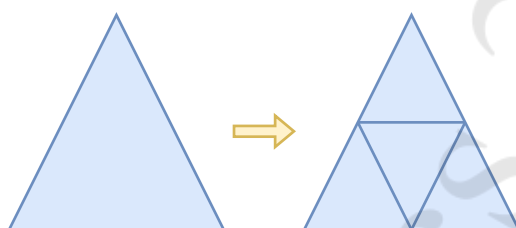


图 1.6: 4-1 细分

2. 调整**所有**顶点的位置，从而改变网格的形状。

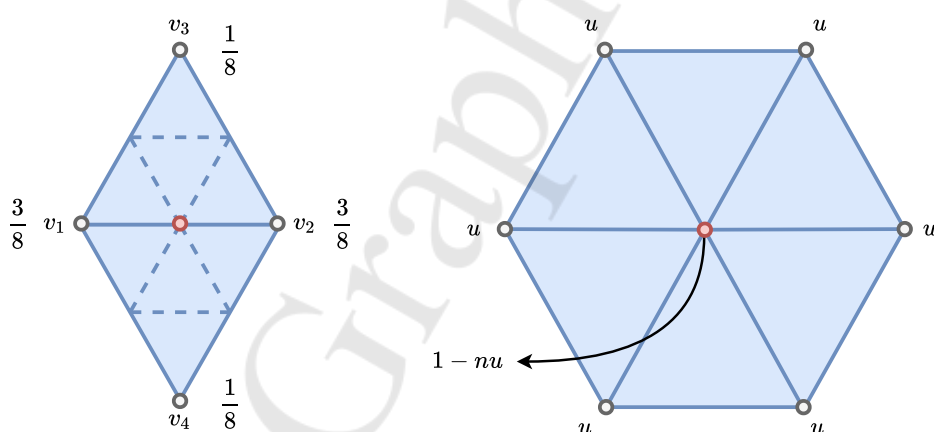


图 1.7: 计算新坐标时各顶点的权重

顶点分为两种：第一步新增的顶点和网格原有的顶点。图 1.7 中实线表示原有的边，虚线表示新增的边。红色顶点是待调整顶点，调整后它们的坐标都是周围顶点**旧**坐标的线性组合，权重如图所示。

计算新增顶点的坐标时，相邻的顶点影响大，相对的顶点影响小：

$$v_{new} = \frac{3}{8}(v_1 + v_2) + \frac{1}{8}(v_3 + v_4) \quad (1.2)$$

原有顶点的坐标则根据其邻接顶点数 n 调整：

$$v_{new} = (1 - nu)v_{old} + u \sum v_{neighbor} \quad u = \begin{cases} \frac{3}{16} & \text{if } n = 3 \\ \frac{3}{8n} & \text{else} \end{cases} \quad (1.3)$$

在一个三角网格上重复这个过程，网格就会变得越来越光滑，趋近于以这个网格为控制点的 **B** 样条曲面。

问题 1.3 顶点权重

如果调整顶点坐标时，把权重系数换一换会发生什么结果？

解 某些权重可能导致细分后产生大量尖锐凸起，并且还有一定的自相似性，也就是导向了分形；另一些则可能导致网格越分越小，最后无限趋近于一个点。正确的 Loop 细分虽然也会导致缩小，但缩小过程是有限的。类似的现象在二维情况（细分曲线）下比较容易证明，有兴趣的同学可以自学 GAMES 102 课程第 8 讲。

1.2 数据结构：半边网格

定义 1.1 (流形网格)

一个网格是流形 (Manifold) 网格，当且仅当它满足 [3]

- 每条边属于一个或两个面片
- 每个顶点所有的邻接面片构成一个扇形面



简单起见，我们的实验过程中只对**封闭流形三角网格**进行细分。直观来看，就是表面平整、没有毛刺和空洞的网格。这种网格还有两个很有用的性质：

性质 封闭流形三角网格的性质

- 所有面片都是三角形
- 每个顶点至少有三个邻接顶点

上面的定义和性质保证了我们进行 Loop 细分第二步时，只有图 1.7 所示的两种可能性，而使用式 1.3 调整原有顶点坐标时不会出现 $n < 3$ 的情况。

Loop 细分第二步调整顶点位置的过程中，无论是新顶点还是原有顶点，都需要访问相邻的顶点来计算新坐标。考虑到顶点数量很多，对邻接关系的访问也会很频繁。

粗一看来，网格中顶点和边的邻接关系很像是图中的邻接关系，似乎用邻接表等数据结构来存储就不错。然而细分操作需要的邻接关系比较多，主要有：

- 由边连接的顶点
- 共用同一条边的三角形面
- 同一个三角形面内的各边

后两种关系主要是**空间位置**带来的，和只考虑拓扑关系的图不太一样，邻接表这样的数据结构就有些不够用。如果网格是流形网格，那我们可以建立它的**半边表示形式**来维护点和边的邻接关系。“半边”就是把一条边拆成来回两条有向边，半边网格以半边间的链接关系为核心。

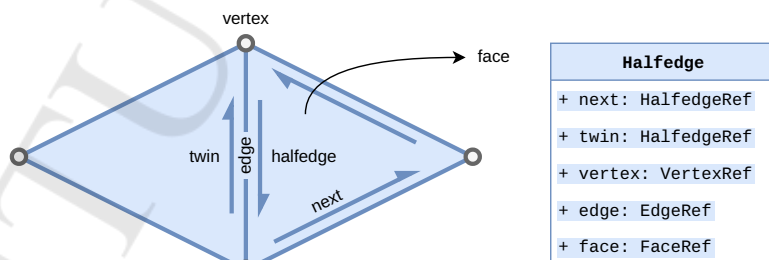


图 1.8: 半边维护的邻接关系

图 1.8 展示了一条半边维护的邻接关系，其中 halfedge 代表这条半边本身，它维护着五个指针：

- next 指向同一个三角形面中的下一条半边
- twin 指向对应同一条边的另一条半边
- vertex 指向这条半边的出发点
- edge 指向这条半边所属的边

- **face** 指向这条半边所属的（三角形）面

另外，每个三角形面上的三条半边顺次首尾相接。

从半边 **halfedge** 出发，迭代遍历 **next** 指针就会绕所在面一圈，最终回到 **halfedge** 本身，而顶点和边都可以在这个遍历过程中找到。如果从 **twin** 开始遍历，则会访问到相邻的面。顶点、边和面都处于从属地位，借助半边来表示：

- 顶点记录从它出发的一条半边
- 边记录属于它的一条半边
- 面记录属于它的一条半边

举个例子，如果我们希望遍历一个顶点的邻接顶点，可以这样做：

1. 找到从它出发的一条半边 **halfedge**
2. **halfedge**->**twin**->**vertex** 就是该顶点的一个邻接顶点
3. 继续遍历下一条从该顶点出发的半边 **halfedge**->**twin**->**next**，再次找到这个顶点记录的半边时停止

这个过程利用了半边维护的两条性质：**twin** 和本身是反向的，并且一定属于和本身不同的面。

第2章 实现思路和细节

笔记 C++ 语法前置要求

- `std::vector` 和 `std::list` 容器的构造、增删、遍历
- 迭代器 (iterator) 和只读迭代器 (const iterator) 的使用
- `std::vector` 和 `std::list` 中何时发生迭代器失效
- 简单了解 `std::optional` 和 `std::variant`

文档中不会专门解释这些语法，未掌握的同学请自行到 zh.cppreference.com 查找 C++ 17 标准文档。

2.1 Scotty3D 中半边网格的实现与操作

2.1.1 数据类型

一个半边网格包含四种基本元素：半边、顶点、边和面，Scotty3D 中相应定义了四个类型。一个半边网格对象用 `std::list`（链表）存储所有的基本元素，用链表的迭代器替代指针：

```
class Halfedge_Mesh
{
public:
    class Vertex;
    class Edge;
    class Face;
    class Halfedge;

    using VertexRef = std::list<Vertex>::iterator;
    using EdgeRef = std::list<Edge>::iterator;
    using FaceRef = std::list<Face>::iterator;
    using HalfedgeRef = std::list<Halfedge>::iterator;

    using VertexCRef = std::list<Vertex>::const_iterator;
    using EdgeCRef = std::list<Edge>::const_iterator;
    using FaceCRef = std::list<Face>::const_iterator;
    using HalfedgeCRef = std::list<Halfedge>::const_iterator;
private:
    std::list<Vertex> vertices;
    std::list<Edge> edges;
    std::list<Face> faces;
    std::list<Halfedge> halfedges;
}
```

形如 `[Element]Ref` 的类型表示指向 `Element` 类型的迭代器，`[Element]CRef` 的类型表示相应的只读迭代器。1 节列举了一条半边应当维护的各种信息，Scotty3D 中都用迭代器实现：


```
class Halfedge
{
public:
    HalfedgeRef& twin() { return _twin; }
```

```

HalfedgeCRef twin() const { return _twin; }
// 其他各私有成员都有相应的函数，这里只展示一个
private:
    HalfedgeRef _twin, _next;
    VertexRef _vertex;
    EdgeRef _edge;
    FaceRef _face;
}

```

每个私有成员都有两个同名的函数用于访问，分别返回迭代器和只读迭代器。**Vertex**、**Edge** 和 **Face** 类型的大致定义与 **Halfedge** 类似，可以用 **halfedge()** 方法访问该对象持有的半边。另外，**Vertex** 类型还有一个 **pos** 记录顶点的坐标，可以直接访问。


 **笔记** 所有位置坐标都是 **Vec3** 类型的变量。**Vec3** 类型定义在 **src/lib/vec3.h** 文件中，由于这个头文件相当短小且清晰，请大家直接阅读该文件了解 **Vec3** 的使用方法。

下面给出遍历某个顶点邻接顶点的例程，其他邻接关系都可以用类似的方法访问。

```

void traverse_adjacent_vertices(VertexRef v)
{
    Halfedge h = v->halfedge();
    do {
        VertexRef neighbor = h->twin()->vertex();
        // 对邻接顶点做一些操作
        h = h->twin()->next();
    } while (h != v->halfedge());
}

```

 **笔记** 一个三维网格少则几百个顶点，多则数万乃至数十万个顶点。而我们的细分一个不太大的网格时，希望能够在在一秒以内完成。为了保证执行效率，请在不需要修改变量值的代码中使用只读迭代器（**[Element]CRef** 类型），这有利于编译器进行优化。

另外，**Halfedge_Mesh** 还提供了 **vertices_begin() / vertices_end()** 等用于遍历某种基本元素的方法。比如我们可以这样遍历所有的边：

```

// edges_end() 返回的是尾后迭代器，必须用 != 比较而不能用 < 比较
for (EdgeRef edge = edges_begin(); edge != edges_end(); ++edge) {
    // 对 edge 做一些操作
}

```

这些方法都有返回只读迭代器的版本，只需在循环中改用 **[Element]CRef** 类型的迭代器即可。

2.1.2 网格操作

我们将网格操作分为两类：

- 局部操作：对少量顶点、边或面进行的操作，只需要访问一两个基本元素及其邻接元素就能完成。
- 全局操作：对整个网格进行的操作，需要遍历大部分甚至全部基本元素才能完成。

用 **Loop** 细分举例：在一条边中点处增加一个新顶点、将一条边分裂成两条边，这是一个局部操作；计算并调整所有顶点的位置，这是一个全局操作。

Loop 细分本身也是一个全局操作，但如果把一些细节单独抽象成局部操作，实现会更清晰、更容易。我们首先介绍一种比较简单的局部操作：翻转边的操作 (**Edge Flip**)。

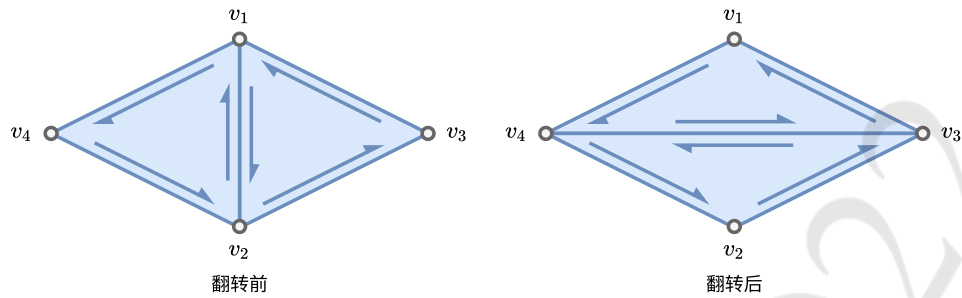


图 2.1: 翻转连接 v_1, v_2 两顶点的边到 v_3, v_4 两顶点

在 Scotty3D 中实现局部操作的基本流程是：

1. 收集所需的基本元素
2. 创建新的基本元素
3. 重新连接这些基本元素
4. 删除无用的元素

根据这个流程，我们给出翻转边操作的代码结构：（翻转过程不需要创建或删除元素）

```
std::optional<Halfedge_Mesh::EdgeRef> Halfedge_Mesh::flip_edge(EdgeRef e)
{
    // 半边
    HalfedgeRef h = e->halfedge();
    HalfedgeRef h_twin = h->twin();
    HalfedgeRef h_2_3 = h->next();
    HalfedgeRef h_3_1 = h_2_3->next();
    HalfedgeRef h_1_4 = h_twin->next();
    HalfedgeRef h_4_2 = h_1_4->next();
    // 顶点
    VertexRef v1 = h->vertex();
    VertexRef v2 = h_twin->vertex();
    VertexRef v3 = h_3_1->vertex();
    VertexRef v4 = h_4_2->vertex();
    // 面
    FaceRef f1 = h->face();
    FaceRef f2 = h_twin->face();

    // 重新连接各基本元素
    h->next() = h_3_1;
    h->vertex() = v4;
    // 其余部分请大家自己完成

    return e;
}
```

在重新连接基本元素时，往往需要对半边进行大量操作，这些操作可以用 `set_neighbors` 替代：

```
HalfedgeRef h;
h->set_neighbors(next, twin, vertex, edge, face);
```

是不是感觉有点冗长？即使是一个局部操作也涉及十个左右的基本元素，并且需要小心地调整它们之间的

连接关系。一旦连接错误，就有可能形状出错（不再是流形网格），甚至是半边的 **twin** 和 **next** 出错（不再是合法网格）。

为了减少调试的痛苦，我们建议大家在编写局部操作代码之前一定要先画图，并把局部操作中访问的基本元素变量名与自己在图中的标记严格对应，再按照图重新连接元素。

2.2 细分流程

2.2.1 4-1 细分

问题 2.1 4-1 细分的实现

1 节介绍了 Loop 细分的两个步骤，不知道大家有没有自己考虑过：新增顶点和边的操作怎么完成？如果按照之前介绍的那样一步将每个三角形分成四个小三角形（也称为 4-1 细分），就会带来两个问题：

- 每个三角形细分前后总共涉及 12 条半边、6 个顶点、9 条边和 4 个面，这么多元素操作起来非常容易混乱。
- 边是两个面片共享的，因此中点也是共享的。细分一个三角形会让一条边变成两条边，这会导致相邻的未细分面片不再是三角形（虽然形状不变，但它可能有四条、五条或者六条边）。

解 为了更容易地实现细分，我们稍微修改一下细分流程，将 4-1 细分的过程再分成两步：

- 将每条原有的边从中点分裂成两条边，并将新顶点（中点）与相对位置的顶点之间新增一条边。
- 旋转所有第一步新增的边（连接新、旧顶点的新增边）。

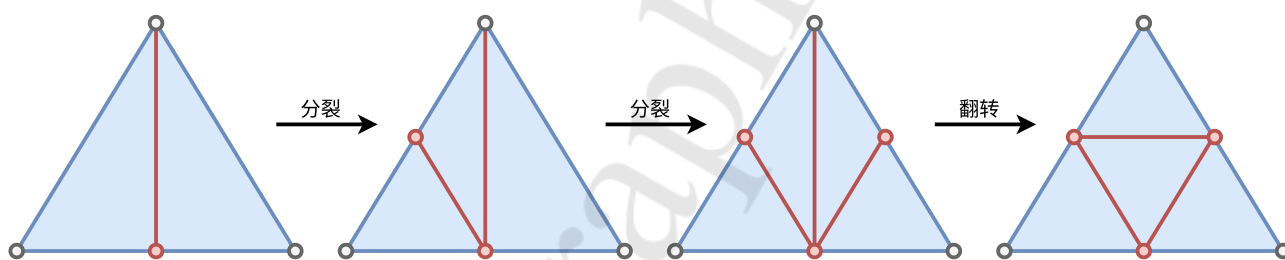


图 2.2: 修改后的 4-1 细分流程

笔记 本次实验中，我们要求将分裂边、翻转边各自实现为一个局部操作。翻转边的操作可以参考图 2.1，而分裂边的操作需要自己画图思考。

在封闭网格中，一条边必然被两个三角形共享，但图 2.2 中我们只画出了一个三角形。在实现分裂操作时，这条边两侧的两个三角形都要分裂，而不能只分裂一个。换言之，一次分裂操作需要将一条边分裂成两条边、将两个三角形各自分裂得到四个三角形。

对照图 2.2 就比较容易理解为什么这样做能实现 4-1 细分：

- 我们每次分裂的总是蓝色边（旧边），并且总是把一个三角形面分成两个三角形面。
- 当一个三角形面的三条旧边都被分裂之后，必然是第三或第四幅图中的样子。
- 第三幅图翻转一条红色边 (Edge Flip) 后必然能转化为第四幅图。

编写代码时请注意如下两点：

- 分裂所有旧边的过程中，不要再次分裂分裂后的边，这会导致无限分裂的死循环。
- 翻转的一定是新边，不要翻转旧边，这会导致网格形状错误。

Edge 类型有一个 `bool` 类型的属性 `is_new` 用于标识新旧，但上面两点中所需要区分的边其实有三种（分裂前的旧边、分裂后的旧边、新边）。请大家自己安排合适的实现方式，尤其注意不要混淆分裂后的旧边和新边。

2.2.2 位置调整

位置调整实现起来反而比较简单，只需要计算旧顶点和新顶点的位置即可。计算顶点位置的过程只需要知道旧顶点的位置，因此在进行 4-1 细分之前计算位置比较方便。

- 遍历每个顶点并访问每个顶点的所有邻接顶点就可以计算出旧顶点的新位置。`Vertex` 类的属性 `new_pos` 用于记录更新后的位置，旧顶点的新位置可以先存放在 `new_pos` 中。同时，应当将这些顶点的 `is_new` 属性标记为 `false`，表示这是旧顶点。
- 遍历每条边并访问其两端和相对的顶点就可以计算出新顶点的位置。`Edge` 类的属性 `new_pos` 用于记录这条边上将来新增顶点的位置。

在新增顶点后，就可以把 `Edge::new_pos` 赋值给新顶点的 `Vertex::pos`；而在进行完 4-1 细分后，就可以把就顶点的 `new_pos` 赋值给 `pos`。

2.3 调试处理几何结构的代码

虽然两年多的学习经历中肯定少不了调试代码的经历，但这里还是有一些建议和工具提供给大家。

首先，`Scotty3D` 在图形界面模式下是一个多线程的程序（虽然处理几何的部分只有一个线程），这会给使用调试器的过程带来一些麻烦。因此我们推荐大家通过输出日志检查程序。头文件 `src/lib/log.h` 中定义了三个输出日志的宏：

- `info` 级别表示运行过程中的正常信息
- `warn` 级别表示运行出错，但不会导致程序崩溃
- `die` 级别表示致命错误，调用该宏将导致程序停止并陷入调试中断

这三个宏的调用格式与 `printf` 一样，并用互斥锁保证了输出日志的过程不会被另外的线程抢占。以下是一些定义在 `Halfedge_Mesh` 类中的方法，可以帮助大家检查自己的半边网格。

`validate()` 方法检查半边网格是否合法（是否保持流形、半边指针是否出错）。每完成一步处理，都可以调用 `validate()` 进行检查。这个函数返回类型是 `std::optional<std::pair<ElementRef, std::string>>`，检查例子如下：

```
// 完成某一步，比如分裂所有旧边后
auto check = validate();
if (check.has_value()) {
    ElementRef element = check.value().first;
    std::string error_msg = check.value().second;
    warn("The mesh is invalid: %s", error_msg.c_str());
    warn("illegal element: %u", id_of(element));
}
```

`ElementRef` 是 `std::variant` 类型（类似于 `union`），可能是指向任意一种基本元素的迭代器。在不知道出错元素类型的情况下，用 `id_of` / `center_of` / `normal_of` 三个方法可以获取这个元素的唯一 ID、中心坐标和法向量。最常用的是基本元素的唯一 ID，这是一个 `unsigned int` 类型的数字，任何两个基本元素（即使是不同类型）的 ID 都是不同的。

请大家自行阅读 `src/geometry/halfedge.cpp` 中 `validate()` 函数的实现，至少理解每一种错误信息的含义，以便检查错误。如果希望进一步从该函数返回的 `ElementRef` 中获取并处理具体的元素，可以用 `std::visit` 配合模板编程分别处理每一种类型：

```
// 假如返回的错误信息是
// A vertex's halfedge does not point to that vertex!
```

表 2.1: 三种基本属性的含义

类型	ID	中心坐标	法向
Vertex	全局唯一	该顶点坐标	顶点法向量
Edge	全局唯一	边中点坐标	相邻两个面法向量的平均
Face	全局唯一	面中心点坐标（所有顶点坐标的算数平均）	所在平面的法向量
Halfedge	全局唯一	无	无

```
// 我希望知道这是哪个点、这条半边又指向了哪个点
// 这时返回的是半边，所以只处理返回 HalfedgeRef 的情况
ElementRef element = check.value().first;
auto processor = [] (HalfedgeRef h) {
    for (VertexRef v = vertices_begin(); v != vertices_end(): ++v) {
        HalfedgeRef h_current = v->halfedge();
        if (h_current == h) {
            warn("The error vertex is %u", v->id());
            warn("Its halfedge points to vertex %u", h->vertex()->id());
        }
    }
};
// overloaded 是一个模板类，定义在 src/lib/mathlib.h 中
// 构造时传入多个可调用对象（比如 lambda 表达式）分别处理每一种可能的类型
std::visit(overloaded{
    processor,
    [] (VertexRef v) {},
    [] (EdgeRef e) {},
    [] (FaceRef f) {}
}, element);
```

以上文介绍的翻转边操作为例，我们可以在 `flip_edge` 函数返回前输出一些辅助信息以检查翻转是否正确：

```
info("---start flipping edge %u---", e->id());
info("(v1, v2) (%u, %u)", v1->id(), v2->id());
info("(v3, v4) (%u, %u)", v3->id(), v4->id());
info("face 1 2 3: %u->%u->%u", f1->halfedge()->vertex()->id(),
    f1->halfedge()->next()->vertex()->id(),
    f1->halfedge()->next()->next()->vertex()->id()
);
info("face 2 1 4: %u->%u->%u", f2->halfedge()->vertex()->id(),
    f2->halfedge()->next()->vertex()->id(),
    f2->halfedge()->next()->next()->vertex()->id()
);
info("---end---");
```

输出可能是这样的：

```
meshedit.cpp:143 [info] ---start flipping edge 97---
meshedit.cpp:158 [info] (v1, v2) (94, 14)
meshedit.cpp:159 [info] (v3, v4) (298, 178)
```

```
meshedit.cpp:183 [info] face 1 2 3: 298->94->178  
meshedit.cpp:187 [info] face 2 1 4: 178->14->298  
meshedit.cpp:188 [info] ---end---
```

通过比较翻转后两个面的顶点是否分别对应 v_1, v_3, v_4 和 v_2, v_3, v_4 可以检查翻转是否正确，并在出错时分辨到底是哪里的连接有问题。

第3章 要求与验证

3.1 实验要求

本实验的目标是实现 Loop 细分，需要大家完成 `src/student/meshedit.cpp` 文件中的三个函数：

表 3.1: 需要填补的函数

函数	功能
<code>Halfedge_Mesh::flip_edge</code>	翻转给定的边 <code>e</code> ，并返回翻转后的边。翻转过程中不能增删任何元素。
<code>Halfedge_Mesh::split_edge</code>	将指定的边 <code>e</code> 分裂成两条边，在新增顶点和 <code>e</code> 相对的顶点之间新增一条边，并返回被分裂的边。分裂过程中只能新增元素，不能删除元素；新增顶点位于 <code>e</code> 两端点中间。
<code>Halfedge_Mesh::loop_subdivide()</code>	完成 Loop 细分，过程中必须调用 <code>flip_edge</code> 和 <code>split_edge</code> 函数完成局部操作，不允许将局部操作写在这个函数里。

除此之外，大家可以任意修改不影响网格数据的函数，尝试 Scotty3D 框架的功能。翻转边和分裂边两种局部操作都可以直接在图形界面中测试，只需要选择一条边并应用 `Model -> flip` 或 `Model -> split` 即可看到操作结果。

打开 Scotty3D 后，大家可以添加简单的物体来验证自己编写的曲面细分程序是否正确，也可以导入 `media` 中的 `.dae` 模型进行细分。以细分一个立方体为例，首先选择 `Layout -> New Object -> Cube -> Add` 创建一个默认的立方体，然后选中立方体并拉近视角，选择 `Model -> Loop` 进行细分。

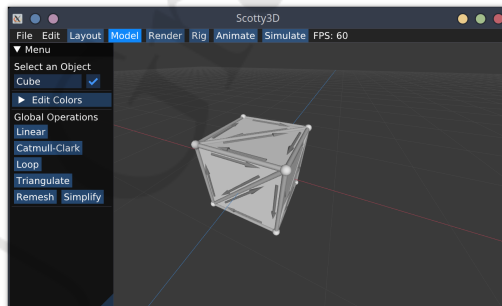


图 3.1: 创建一个边长为 1 的立方体

图 3.2 展示了连续进行三次细分的结果，可以看到物体缩小、表面逐渐光滑。

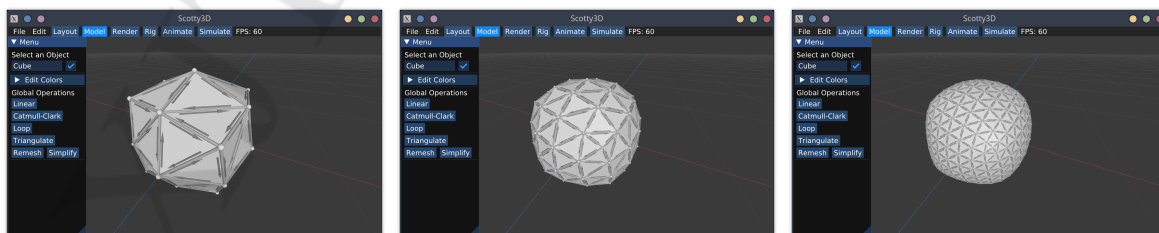



图 3.2: 对立方体进行迭代细分

3.2 实验结果自动测试

为了测试大家完成的代码能否正确地进行细分，我们用四个简单几何体进行测试：立方体、锥体、柱体和环面。通过自动测试是完成本实验的**必要条件**，无法通过自动测试则认定为未能正确实现细分。

 **笔记** 如果你的细分结果看起来毫无问题却无法通过自动测试，欢迎提交该结果，与我们共同建设课程实验。

自动测试程序是一个独立的 CMake 工程 (Project)，依赖 Scotty3D 中的部分文件但并不与 Scotty3D 一起编译。测试代码由一个压缩包 `autotest.zip` 单独提供。请将此压缩包解压到 Scotty3D 目录，形成如下目录结构：

```

Scotty3D
├── autotest
│   └── lab2
│       ├── CMakeLists.txt
│       ├── halfedge.cpp
│       ├── halfedge.h
│       ├── meshedit.cpp
│       ├── objects.inc
│       ├── std
│       ├── test_main.cpp
│       ├── utils.cpp
│       └── utils.h
├── build
├── build_win.bat
├── CMakeLists.txt
├── deps
├── docs
├── LICENSE.txt
├── media
├── README.md
└── src
  
```

请大家将自己完成的三个函数复制到 `autotest/lab2/meshedit.cpp` 中，并**不要修改自动测试目录下任何其他代码**。

在 Windows 平台完成实验的同学，请使用 Visual Studio 直接打开 `lab2` 目录（首页选项中的“打开本地文件夹”），然后生成 `autotest2` 项目并运行之。若之前已经用 Visual Studio 打开了 Scotty3D，请在打开自动测试文件夹之前关闭 Scotty3D 工程，因为自动测试和 Scotty3D 并不共同编译。

在 Linux / Mac OS 平台完成实验的同学，请手动编译并执行测试：

1. 在 `autotest/lab2` 目录下新建 `build` 目录
2. 在 `build` 目录中使用 CMake 构建工程并编译，可执行文件相对于 Scotty3D 根目录的路径应当是 `autotest/lab2/build/autotest`
3. 切换到 `autotest/lab2` 目录下执行 `./autotest` 进行测试

如果你在最后看到了这条消息：

```
[info] all tests passed, congratulations!
```

那么恭喜你通过了所有测试。反之，测试程序将输出一些以 `[warn]` 开头的错误信息，并以这条消息结尾：

```
[warn] test failed
```

表 3.2: 错误信息和相应的解释

错误信息	解释
the result of subdivision is a invalid mesh: [some error messages from the function <code>validate()</code>]	细分后的网格不是流形或不合法，具体错误信息参考之前对 <code>validate()</code> 函数的说明
number of vertices is wrong: [number of vertices] (should be [the answer])	细分后网格顶点数量不正确，应当有 <i>the answer</i> 个顶点
at least one of the vertices has incorrect position	有一些顶点的位置不正确
number of edges is wrong: [number of edges] (should be [the answer])	细分后网格边数量不正确，应当有 <i>the answer</i> 条边
at least one of the edges connects incorrect vertices	有一些边的连接关系不正确

3.3 向助教寻求帮助

大家在遇到无法解决的问题，或希望了解超出文档范围的知识时，欢迎向助教寻求帮助。但助教人数有限且并非全职，而选课的同学很多，所以我们必须尽可能高效地解决问题。为了让我们沟通的过程更像是知识交流而不是调查访谈，希望大家在提问时更多地发挥自己的主观能动性。在寻求帮助时请**务必**遵循这些约定：

- 如果你询问与局部操作相关的问题，请附上自己画的示意图，并保证自己代码中的变量命名与示意图完全一致。
- 如果你询问与框架代码相关的问题，请指出询问的是哪个源文件，并提供所要询问部分的截图，而非拍摄屏幕的结果或整个源代码文件。
- 对问题的描述尽量清晰，问“为什么我的结果不对？”会导致助教一头雾水，而“为什么我做了某处理，检查了某位置，但某一步的输出表明程序还是做错了某种操作？”往往能为双方节省很多时间。

助教在回答问题时经常依赖于一些中间步骤的日志输出，所以如果你已经测试过某些中间变量的值或者输出过某些日志信息，也请尽可能附上这些材料。但是，我们一概不鼓励将未经过滤的材料囫圇扔给助教。除非实在难以区分哪些信息与错误无关，否则请尽量挑选与错误相关的信息。

参考文献

- [1] Charles Loop. “Smooth subdivision surfaces based on triangles”. MA thesis. University of Utah, 1987.
- [2] 刘利刚. *GAMES 102*. URL: http://staff.ustc.edu.cn/~lgliu/Courses/GAMES102_2020/default.html.
- [3] 陈仁杰. 计算机图形学 *Chapter 08: Mesh*. 2022. URL: http://staff.ustc.edu.cn/~renjie/CG_2021S2/default.htm.