

# 西安交通大学计算机图形学实验文档

## 几何处理部分

作者：李昊东 李晶

组织：计算机图形学课题组

时间：November 16, 2023



# 目录

<b>第 1 章 基于半边的局部操作</b>	<b>2</b>
1.1 实验内容	2
1.2 指导和要求	2
1.2.1 几何操作	2
1.2.2 半边网格	4
1.2.3 实现局部操作	5
1.2.4 检查和调试	7
1.2.5 要求	8
1.3 提交和验收	9
<b>第 2 章 Loop 曲面细分</b>	<b>10</b>
2.1 实验内容	10
2.2 指导和要求	10
2.2.1 曲面细分	10
2.2.2 网格的整体性质：封闭与流形	13
2.2.3 实现思路和细节	13
2.2.4 调试	14
2.2.5 要求	14
2.3 提交和验收	15
<b>第 3 章 基于 QEM 的曲面简化</b>	<b>16</b>
3.1 实验内容	16
3.2 指导和要求	16
3.2.1 用贪心思想简化网格	16
3.2.2 二次误差度量	16
3.2.3 实现思路和细节	18
3.2.4 要求	19
3.3 实验结果	19
3.4 提交和验收	19
<b>第 4 章 各向同性重网格化</b>	<b>20</b>
4.1 实验内容	20
4.2 指导和要求	20
4.2.1 评判网格的性质	20
4.2.2 重构过程	20
4.2.3 实现思路和细节	21
4.2.4 要求	21
4.3 实验结果	21
4.4 提交和验收	22
<b>第 5 章 对四边形网格的支持</b>	<b>23</b>
5.1 实验内容	23

---

5.2 指导和要求 . . . . .	23
5.3 提交和验收 . . . . .	23
<b>参考文献</b>	<b>24</b>


# 序

这份文档是计算机图形学几何处理部分选做实验的介绍。在这部分实验中，你需要练习使用半边网格进行几何操作，首先实现三种修改局部结构的局部操作，然后实现三种处理 mesh 整体的全局操作。在完成几何处理部分之后，你将得到一个可以对三角形网格进行曲面细分、曲面简化和重网格化的几何编辑器。

这部分文档总共包含五个选做实验，分别是

- 基于半边的局部操作：实现对指定边的分裂、翻转和坍塌三种局部操作。
- Loop 曲面细分：将一个低面数网格自动细分成高面数网格。
- 基于 QEM 的曲面简化：自动减少网格的面数，得到更简单的网格。
- 各向同性重网格化：调整网格上的边和顶点，使面片的分布和形状更加均匀。
- 四边形网格：让 Dandelion 支持加载、预览和编辑四边形网格。

最后一个实验是挑战任务，我们相信它给你带来的收获将远超一般的实验，但它的难度同样很大。请根据自己的精力、兴趣和理解情况，慎重选择自己要做的实验。<sup>1 2</sup>

 **笔记** 在开始做实验之前，请确保自己至少知道以下的 C++ 语法特性是什么：

- 表示“可以是空值”的 `std::optional` 类型
- 替代 C 风格 `NULL` 宏的 `nullptr` 常量
- `vector`, `set`, `map` 和 `unordered_map` 这几种常用 STL 容器

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

---

<sup>1</sup>这份文档使用 `ElegantBook` 模板编写，按 CC BY-NC-SA 4.0 协议发布。

<sup>2</sup>封面是同一个模型细分前后的对比，自左至右依次为原始输入、细分一次、两次和三次的结果。

# 第 1 章 基于半边的局部操作

基于半边的局部操作（编号 2.7）是半边网格的基础实验，这个实验将介绍如何使用半边网格进行几何操作，并为后续的全局几何处理打下基础。

## 1.1 实验内容

理解半边网格在几何操作上的便利性，实现边分裂、边坍塌、边折叠三种局部操作。

## 1.2 指导和要求

### 1.2.1 几何操作

什么是几何操作？通常来说，只要是改变模型形状的操作都可以算是几何操作，而修改模型形状的过程一般就是建模的过程，所以“几何”这个领域的知识与“建模”这个实际任务往往联系非常紧密。

与人工修整形状的过程不同，自动调整形状的过程需要相应的数学模型作为指导，而这部分实验中我们使用的是离散曲面，也就是将 mesh 视作是对某个理想光滑曲面的离散近似。这种数学模型的好处在于可以迁移光滑曲面上的微分几何知识，从而将去噪、平滑、误差分析等理论应用到 mesh 处理的过程中。

但 mesh 毕竟不是一个光滑曲面，也没有简单的参数形式或隐函数形式，因此我们实际操纵 mesh 时往往不会直接使用导数、梯度等工具，而是用一些局部操作 (local operation) 完成局部求值或局部修改的过程。比如通过分裂操作增加局部精度，或通过坍塌操作降低局部的精度。本次实验中要实现的三种局部操作如图 1.1, 1.2 和 1.3 所示。

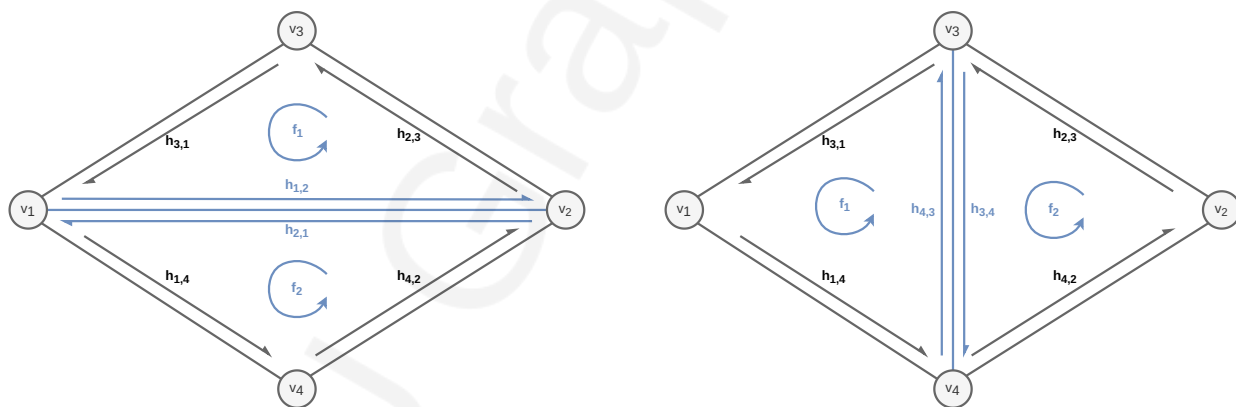


图 1.1: 边翻转操作（蓝色表示被修改的元素）

边翻转操作会逆时针旋转 mesh 的一条边，将其连接到另外两个顶点上，图 1.1 中是在三角形网格上翻转一条边的过程。翻转操作既可以改变顶点间的连接关系，也可以改变面片的形状，但不会增删任何元素。

边分裂操作会将一条边分裂成两条边并新增一个顶点，然后将新增顶点与边端点之外的其他顶点连接起来。图 1.2 是在三角形网格上分裂一条边的过程。分裂操作可以产生新的顶点、边和面片，进而用于细分曲面。

边坍塌操作会将一条边“折叠”成一个顶点，如果这条边邻接的两个面片都是三角形，它也会导致这两个面片坍塌成两条边，如图 1.3 所示。坍塌操作可能会删除面片，因而可以用来简化曲面。

我们将这些操作称为“局部操作”是因为它们只需要网格上局部区域的信息就能完成，这个“局部”通常是某个顶点或边的邻域。与局部操作相对，需要以某种形式遍历整个网格的操作称为全局操作 (global operation)。

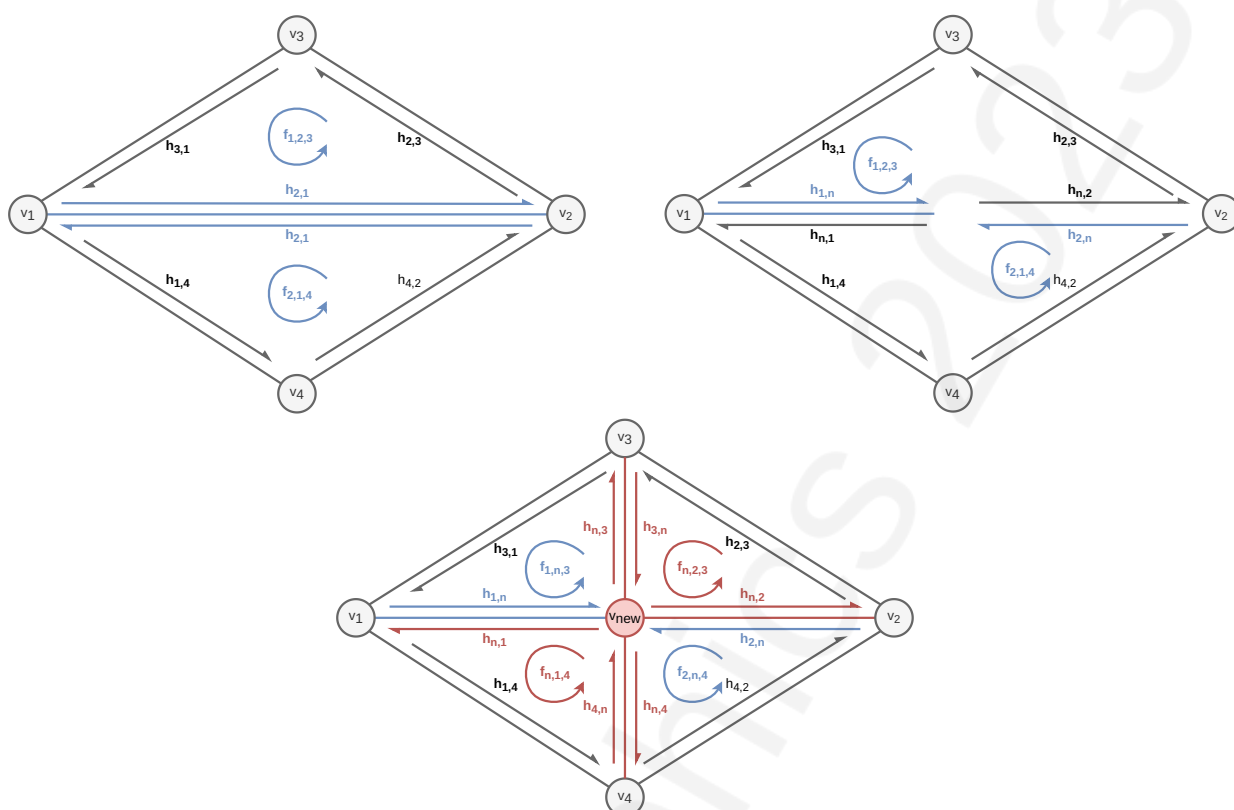


图 1.2: 边分裂操作 (蓝色表示被修改的元素, 红色表示增删的元素)

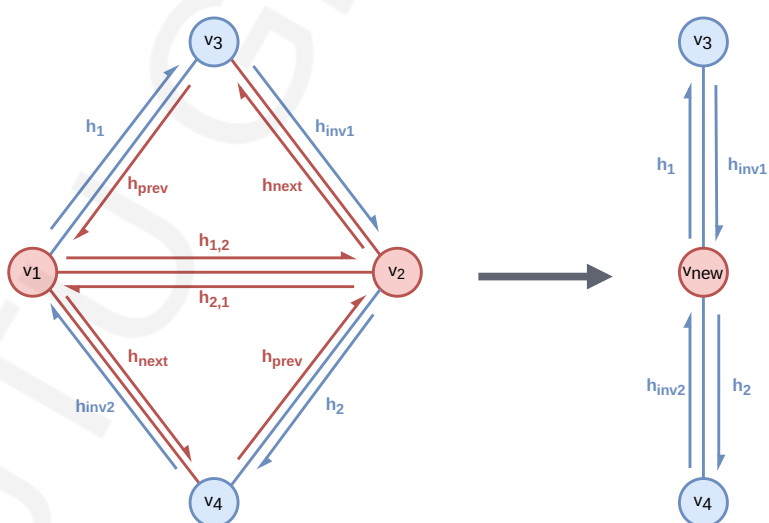


图 1.3: 边坍塌操作 (蓝色表示被修改的元素, 红色表示增删的元素)

，这部分后续的几个实验都是全局操作。

## 1.2.2 半边网格

从局部操作的示意图中，我们不难看出它们往往需要根据顶点和边的邻接关系访问相邻的几何元素，这种访问行为在几何处理算法中非常普遍。

最直接也是最经典的 mesh 存储方式是数组与索引的组合：所有顶点的坐标被存储到一个数组中，每条边存储两个顶点索引（下标）、每个面片存储若干顶点索引。这种存储方式很节省内存，要遍历所有内容也很方便。但索引并不能指示顶点、边和面片之间的邻接关系，想要沿着几何上相邻的元素“游走”是非常困难的。如果你希望遍历与顶点  $v$  相邻的所有顶点，你将不得不依次检查 mesh 中的每个顶点是否与  $v$  相邻，这样做的时间成本实在太大了。

引入一种虚拟的几何元素“半边”（halfedge），再用大量的指针维护几何上的邻接关系，就得到了半边网格。它的基本几何元素有四种：半边、顶点、边和面片。半边网格用链表存储各种几何元素，而且在一条链表的基础上新添加了多种链接关系。下面我们简要介绍 Dandelion 中实现的半边网格。

“半边”是有方向的边，每条边由两条方向相反的半边组成。每个半边结构体 Halfedge 保存有六个指针：

- `inv` 是同一条边上反方向的半边，反向的反向就是自身。
- `prev` 和 `next` 是一个面片环 (face loop) 上的前一条和下一条半边。沿着面片法线的反方向看去，所有半边组成一个逆时针的环。
- `from`, `edge` 和 `face` 依次是半边的起点、所在边和所在面。

这些指针的指向如图 1.4 所示。

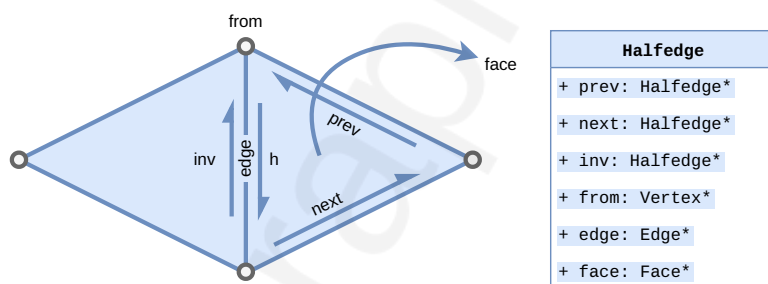


图 1.4: 半边与其他元素的链接关系

每个顶点、边和面片也都保存着指向半边的指针：

- 顶点保存从自己出发的任意一条半边。
- 边保存从属于自己的任意一条半边。
- 面片保存从属于自己的任意一条半边。

Dandelion 用了一个头文件和多个源文件实现半边网格和几何操作：函数的声明和类型的定义都位于 `halfedge.h` 文件中；`Halfedge`, `Vertex`, `Edge` 和 `Face` 这四种基本类型的类方法则在各自的源文件中实现；所有局部操作和全局操作都在 `meshedit.cpp` 中。在几何处理部分的实验中，你需要填写的正是定义在 `meshedit.cpp` 的函数。

**开发者文档：**几何处理中包含对 `Halfedge`, `Vertex`, `Edge` 和 `Face` 这几个类的介绍，请在开始写代码之前大致阅读一遍，这将有助于你了解哪些问题可以直接从开发者文档上找到答案。

启动 Dandelion，加载 `cube.obj` 并进入建模模式 (model mode) 就可以看到用箭头表示的半边（如图 1.5 左侧所示）。直接点击某条半边就可以选中它，此时点击 `Local Operations` 一栏下的 `Inverse`, `Next`, `Previous` 等按钮就可以在这条半边连接的元素之间切换（如图 1.5 右侧所示）。

场景中的顶点、边和面片也都是可以选中的，选中后会显示相应类型元素可以执行的局部操作，例如切换

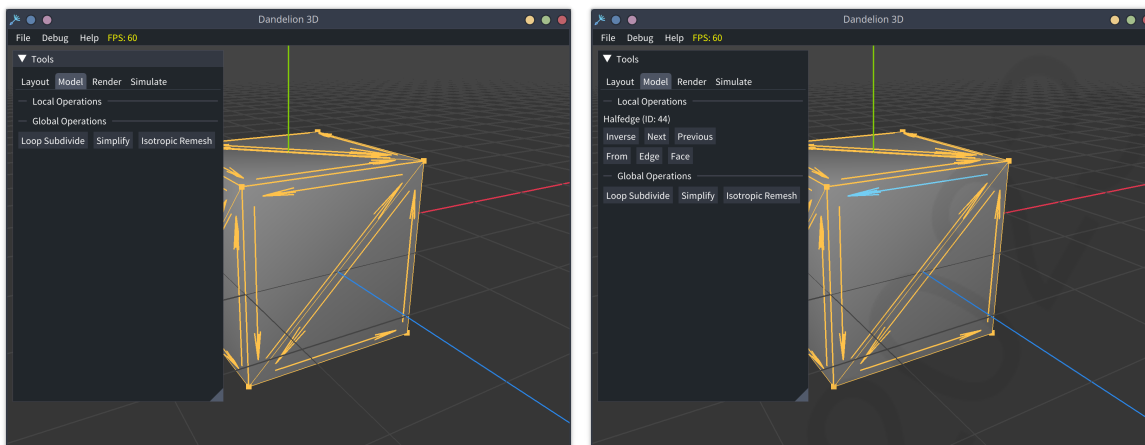


图 1.5: 建模模式下看到的半边网格

到半边或调整位置等。此时我们选中一条边执行翻转操作，效果如图 1.6 所示。用类似的方法执行分裂或坍塌操作，结果也会直接显示在界面上。当你在代码中实现了这三个操作后，可以用这种方法来验证自己的实现是否正确。

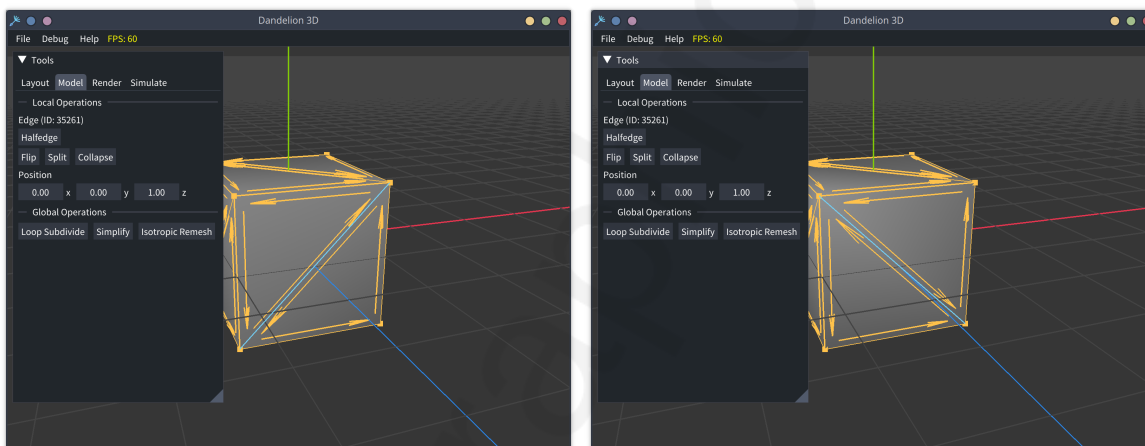


图 1.6: 一条边翻转前和翻转后的对比

### 1.2.3 实现局部操作

半边网格在遍历邻接关系上具有明显的优势，例如我们可以这样遍历一个顶点的  $\mathcal{N}_1$  邻域（也叫 1-ring neighborhood，即所有直接相邻的顶点）：

```
void traverse_1_ring(Vertex* v)
{
    Halfedge* h = v->halfedge;
    do {
        Vertex* neighbor = h->inv->from;
        h = h->inv->next;
    } while (h != v->halfedge);
}
```



这段代码利用了两个性质：`h->inv` 和 `h` 必定属于不同的面片、任一条半边的 `next` 必然以它的终点为起点。通过反复在 `inv` 和 `next` 间切换，我们通常可以在  $O(1)$  的时间复杂度内遍历一个顶点的所有邻接点（除非这个顶点与其他所有顶点都邻接）。

利用这些邻接关系实现局部操作的流程一般是这样的：

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

根据这个流程和图 1.1 中的步骤，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）

```
optional<Edge*> HalfedgeMesh::flip_edge(Edge* e)
{
    // 要用到的半边
    Halfedge* h      = e->halfedge;
    Halfedge* h_inv  = h->inv;
    Halfedge* h_2_3  = h->next;
    Halfedge* h_3_1  = h_2_3->next;
    Halfedge* h_1_4  = h_inv->next;
    Halfedge* h_4_2  = h_1_4->next;
    // 要用到的顶点
    // v1 and v2 are vertices along the edge
    Vertex* v1 = h->from;
    Vertex* v2 = h_inv->from;
    // v3 and v4 are vertices opposite the edge
    Vertex* v3 = h_3_1->from;
    Vertex* v4 = h_4_2->from;
    // 要用到的面片
    Face* f1   = h->face;
    Face* f2   = h_inv->face;

    // 重新连接各基本元素
    h->next     = h_3_1;
    h->prev     = h_1_4;
    h->from     = v4;
    // 其余部分请自己完成

    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：

```
Halfedge* h;
h->set_neighbors(next, prev, inv, from, 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 `inv` 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。

分裂一条边的过程需要新增一些元素(见图 1.2),这时你应该使用名为 `HalfedgeMesh::new_[element]` 的函数，其中 `[element]` 可以是任何一种基本元素：（切勿直接使用 `new` 运算符创建新元素）

```
Halfedge* h = new_halfedge();
Vertex* v = new_vertex();
Edge* e = new_edge();
Face* f = new_face();
```

相对地，坍塌一条边的过程需要删除一些元素（见图 1.3），这时你应该使用 `HalfedgeMesh::erase` 函数，我们提供了删除每种元素类型的重载：

```
erase(h);
erase(v);
erase(e);
erase(f);
```

同样，切勿直接使用 `delete` 运算符删除元素。这样做极有可能在运行时产生访存错误，进而产生 `segment fault`, `core dump` 之类的问题。

图 1.1, 1.2 和 1.3 中只展示了最理想的情况，而你在实现这三个局部操作时应该额外注意一些特殊情况：

- 大多数边都处于 `mesh` 内部，被两个面片共享。如果某个 `mesh` 不封闭，就有可能存在一些位于 `mesh` 边界的边。从图 1.1 中可以看出边翻转操作需要一个“菱形”结构，而边界上的边根本不具备这种结构。对于它们而言，翻转操作不是良定义的，也不应该执行。
- 分裂一条边界上的边并没有问题，但此时图 1.2 中的  $v_3$  与  $v_4$  之一不存在，操作过程中应该考虑这个情况。
- 坍塌一条边界上的边也是可以的，但此时这条边只有一侧是有面片的，操作过程中不应该删除两个面片。

上面的情况都来自边界上的边，不难想到：它们一侧的半边存在而另一侧的半边不存在，而边界上半边的 `inv` 指针似乎也应该是空指针。但让半边持有空指针是比较危险的，一旦某处忘记判断边界就解引用 `inv` 指针，就会产生非法访存导致的错误，轻则程序崩溃，重则不知不觉修改了某处的数据。为了避免这种问题，我们将每个边界环视作一个虚假的“面片”，并且也为这个面片建立半边，这样以来每条半边的 `inv` 指针都必定非空，至少不会导致访存问题了。

Dandelion 中，`Halfedge`、`Edge` 和 `Face` 类都有与边界判定相关的属性或方法，分别是：

- `Halfedge::is_boundary`，表示它是否是边界的一部分。
- `Edge::on_boundary`，表示它是否在边界上。
- `Face::is_boundary`，表示它是否为虚假的边界面片 (virtual face loop)。

请点击链接，在开发者文档上查看进一步的解释。

## 1.2.4 检查和调试

由于 Dandelion 的函数调用关系比较复杂，我们更推荐你使用日志检查程序运行状况并进行调试。在前置实验中你已经使用过 `spdlog` 日志库，在 Dandelion 的代码中使用方式也一样。我们已经预先为大部分类创建了 `logger`，日志同时输出到终端和 `dandelion.log` 文件中。

还是以翻转操作为例，我们可以在 `flip_edge` 函数返回之前输出一些结构信息来检查翻转后元素间的链接关系：

```
logger->trace("---start flipping edge {}---", e->id);
// 假如将 e 的两个端点赋值给 v1 和 v2，将两个相对位置的点赋值给 v3 和 v4
logger->trace("(v1, v2) ({{, {{})", v1->id, v2->id);
logger->trace("(v3, v4) ({{, {{})", v3->id, v4->id);
// 修改各种指针
logger->trace("face 1 2 3: {{->{{->{{", f1->halfedge->from->id,
            f1->halfedge->next->from->id,
            f1->halfedge->next->next->from->id);
logger->trace("face 2 1 4: {{->{{->{{", f2->halfedge->from->id,
            f2->halfedge->next->from->id,
            f2->halfedge->next->next->from->id);
logger->trace("---end---");
```

一次正确的翻转可能会输出这样的信息：

```
[Halfedge Mesh] [trace] ---start flipping edge 49---
[Halfedge Mesh] [trace] (v1, v2) (3, 1)
[Halfedge Mesh] [trace] (v3, v4) (5, 4)
[Halfedge Mesh] [trace] face 1 2 3: 5->3->4
[Halfedge Mesh] [trace] face 2 1 4: 4->1->5
[Halfedge Mesh] [trace] ---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  $v_2, v_3, v_4$ ，我们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。

上面的日志涉及到一个新的属性 `id`，这代表**全局唯一 ID**，是用于区分基本元素的重要工具。每个半边、顶点、边或面片都有自己的 ID，并且相互不会重复（不同的半边一定有不同的 ID，而且半边的 ID 一定和顶点的不同，以此类推）。分配 ID 的过程始终递增，如果你删除了某个元素，它的 ID 就不会再次出现。

如果你将上面的语句添加到代码中，你会发现程序并没有输出相应的日志。这是因为所有 `logger` 默认的日志级别是 `debug`（如果你用 `Release` 模式编译，那么默认日志级别是 `info`），你需要在菜单栏的 `Debug -> Global Logging Level` 中选择 `Trace`，这些日志才会被输出。

**问题 1.1** 为什么不直接将这些信息输出为 `debug` 甚至 `info` 级别的日志呢？这是因为日志级别有自身的含义。一般 `info` 表示正常运行的记录，`debug` 则是额外的调试信息。

**解** 直观来看，通常 `info` 或 `debug` 级别都不会在短时间内输出数万行的日志，而以后全局操作时会非常频繁地调用局部操作，很容易产生成千上万次调用。`trace` 通常是用于追踪执行过程的“最详细级别”，将每次局部操作时的检查信息设置为这个级别是比较合适的。否则，一些关键信息可能会被海量的输出淹没。如果你确实需要追踪过程，只需要临时开启 `trace` 级别输出就可以了。

### 1.2.5 要求

我们已经写好了构造半边网格的代码，请填写 `geometry/meshedit.cpp` 中的三个函数：

- `HalfedgeMesh::flip_edge`，要求考虑传入的边在 `mesh` 边界上的情况。

- `HalfedgeMesh::split_edge`，不要求考虑边界边。
- `HalfedgeMesh::collapse_edge`，要求考虑传入的边在 mesh 边界上的情况。

完成后，请载入 `cow.dae` 文件，进入建模模式并进行旋转、分裂、坍塌操作各三次，整个过程请不要改变视角。图 1.7 是一个可供参考的例子，展示了九次局部操作产生的变化，你的结果不必与它完全相同。

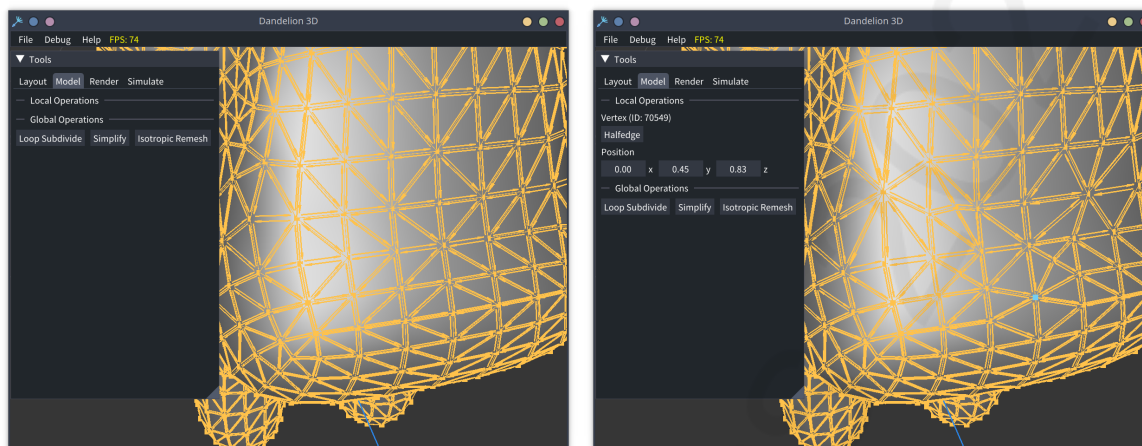


图 1.7: 局部操作进行前（左）与进行后（右）对比

## 1.3 提交和验收

要提交的结果截图有两张，分别是局部操作进行前和进行后的截图（参考图 1.7）；其他内容按照标准要求即可。

验收时请现场演示三种局部操作，完成一个局部操作得 4 分，两个得 7 分，三个得 10 分。所有处理不封闭 mesh 边界的情况，只需要说明思路，不需要执行测试。

## 第 2 章 Loop 曲面细分

Loop 曲面细分（编号 2.8）这个实验中，你将细分一个三角形网格，从而得到更精细、光滑的模型。

### 2.1 实验内容

理解 Loop 曲面细分的基本原理和适用条件，完成对封闭流形网格的曲面细分操作。

### 2.2 指导和要求

#### 2.2.1 曲面细分

##### 问题 2.1 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。

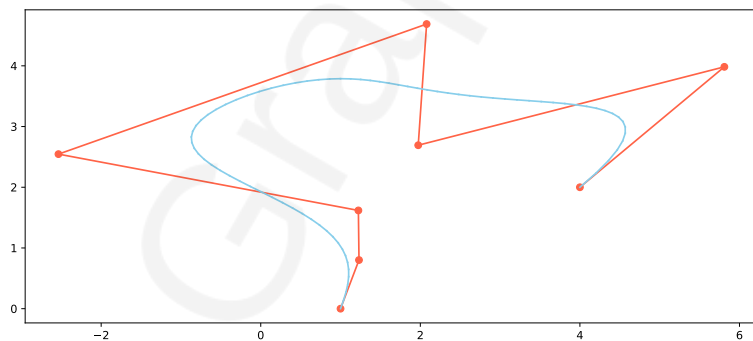


图 2.1: 一条由八个点控制的四阶 B 样条曲线

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。

一个方向  $k$  阶、另一个方向  $l$  阶的 B 样条曲面参数方程是：

$$\mathbf{p} = \sum_{i=1}^m \sum_{j=1}^n \mathbf{p}_{i,j} B_{i,k}(u) B_{j,l}(v)$$

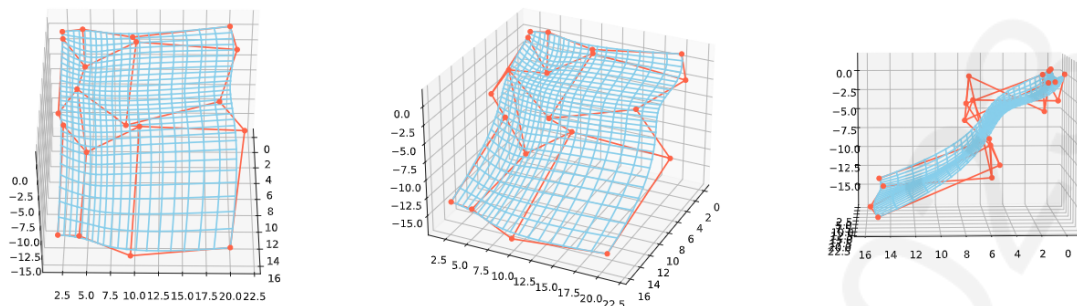


图 2.2: 从不同视角观察一个 B 样条曲面及其控制点

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 2.3 中蓝色的低阶基函数线性组合得到橙色的高阶基函数。

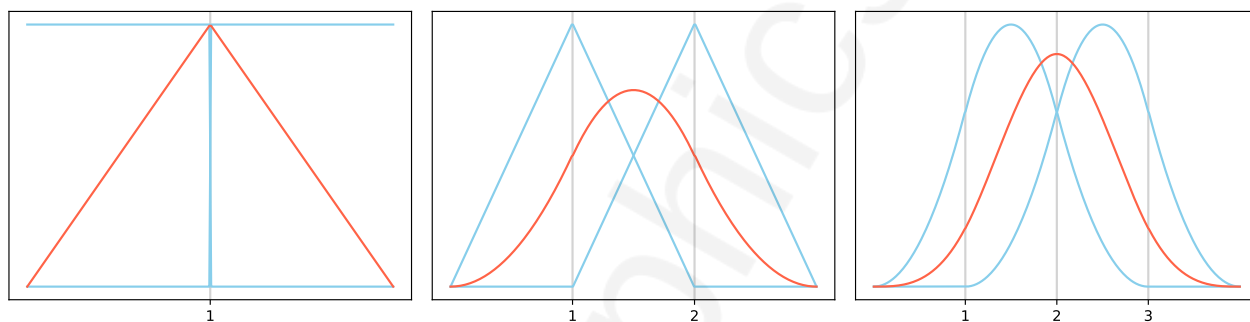


图 2.3: 一至三阶的 B 样条基函数

### 问题 2.2 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 2.4 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）。

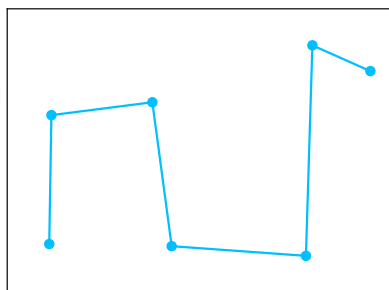


图 2.4: 一条随意的折线

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 2.5 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。

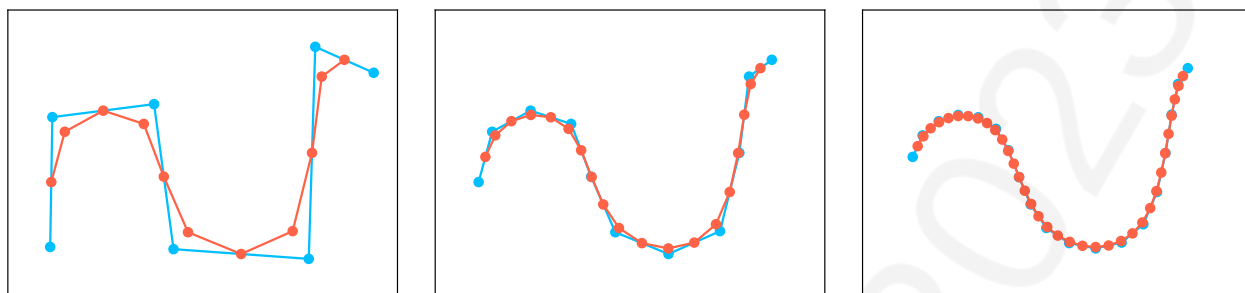


图 2.5: 细分操作的迭代过程

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** [2] ( $C^2$  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)

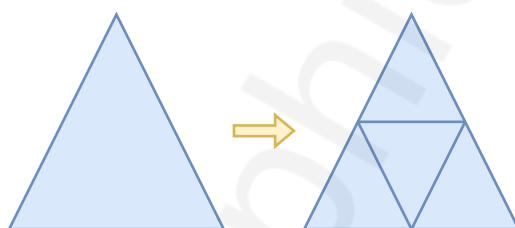


图 2.6: 4-1 细分

2. 调整**所有顶点**的位置，从而改变网格的形状。

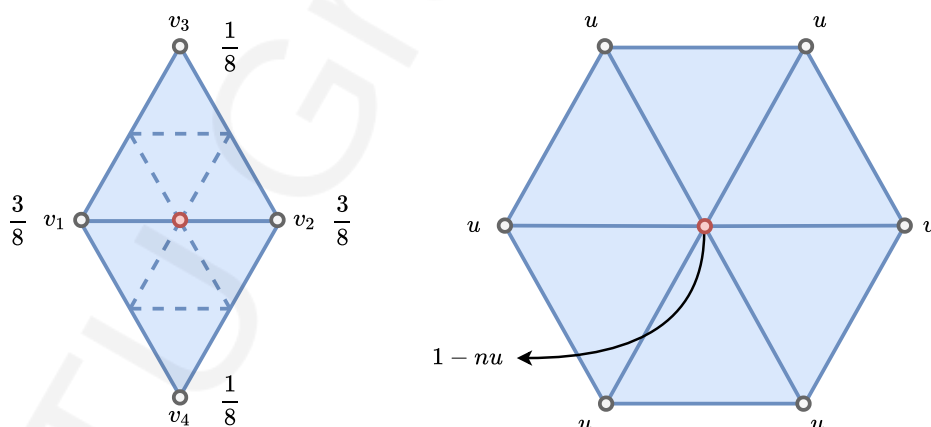


图 2.7: 计算新坐标时各顶点的权重

顶点分为两种：第一步新增的顶点和网格原有的顶点。图 2.7 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：

$$v_{new} = \frac{3}{8}(v_1 + v_2) + \frac{1}{8}(v_3 + v_4) \quad (2.1)$$

原有顶点的坐标则根据其邻接顶点数  $n$  调整：

$$v_{new} = (1 - nu)v_{old} + u \sum v_{neighbor} \quad u = \begin{cases} \frac{3}{16} & \text{if } n = 3 \\ \frac{3}{8n} & \text{else} \end{cases} \quad (2.2)$$

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 2.3 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。

## 2.2.2 网格的整体性质：封闭与流形

### 定义 2.1 (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 2.7 所示的两种可能性，而使用式 2.2 调整原有顶点坐标时不会出现  $n < 3$  的情况。如果你对更普适的 Loop 细分算法（允许网格不封闭、允许指定某些边不改变）感兴趣，可以阅读 C. Loop 的论文原文 [2]。

### 2.2.3 实现思路和细节

曲面细分是一个全局操作，实现它需要遍历所有顶点（或边、面片等），但之前我们并没有提到应该怎么做。

Dandelion 将每一种基本元素存储于一个双链表中，双链表类型的说明请参考[开发者文档：LinkedList](#)。要遍历一个双链表中的元素，我们就要沿着 `next_node` 指针移动：

```
for (Vertex* v = vertices.head; v != nullptr; v = v->next_node) {
    // 对顶点 v 进行某些操作
}
```

遍历边或面片的过程非常接近，只要改为遍历 `edges` 或 `faces` 即可。

解决了遍历问题，接下来就是实现 4-1 细分和位置调整这两步了。

### 问题 2.4 4-1 细分的实现

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？如果想要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：

1. 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。
2. 旋转所有第一步新增的边（连接新、旧顶点的新增边）。

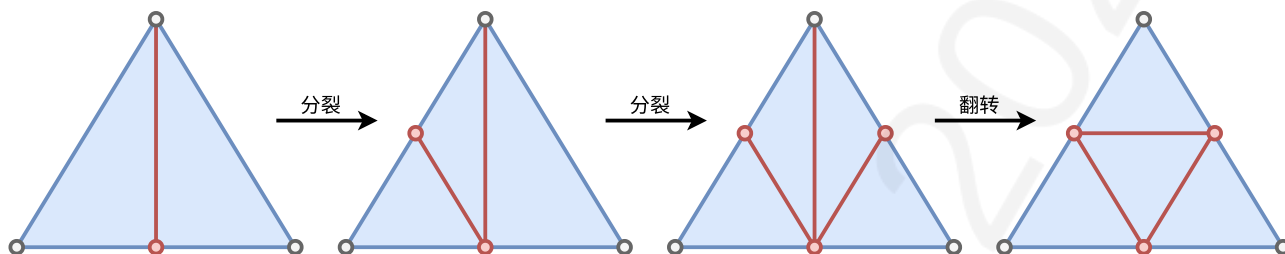


图 2.8: 修改后的 4-1 细分流程

**笔记** 在封闭网格中，一条边必然被两个三角形共享，但图 2.8 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 2.8 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` 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。

在局部操作（实验 2.7）中，你已经实现了翻转和分裂操作，可以直接用在本次实验中。

## 2.2.4 调试

我们之前已经讲过输出日志，也同样鼓励你在实现 Loop 细分时输出级别合适的日志。除此以外，这里再介绍一个检查工具：`HalfedgeMesh::validate` 函数。这个函数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，都可以调用它进行检查。如果检查出了问题，它会以 `error` 级别输出日志指出错误内容。

## 2.2.5 要求

本实验需要填写 `geometry/meshedit.cpp` 中的 `HalfedgeMesh::loop_subdivide` 函数。填写完成后加载 `cube.obj` 进入建模模式，点击 `Loop Subdivide` 按钮即可进行一次细分。图 2.9 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。

你需要继续加载 `cow.dae` 和 `bunny.obj` 进行细分，验证程序的正确性。

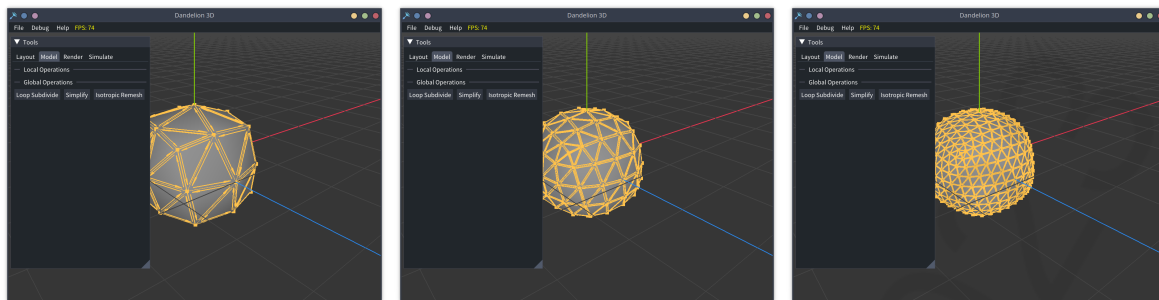


图 2.9: 对立方体进行迭代细分

## 2.3 提交和验收

要提交的结果截图有三张：细分四次的 *cube.obj*、细分两次的 *cow.dae* 和细分一次的 *bunny.obj*。

所有的细分测试必须依次完成，中途只允许删除物体（选中后按 `delete` 键）不能退出程序，因此日志文件中应当包含全部细分过程输出的日志。测试时日志级别设置为 `debug`。

验收时需要现场演示细分操作，正确实现 4-1 细分可以得到 10 分，正确调整顶点位置可以再得到 5 分，共计 15 分。

## 第3章 基于 QEM 的曲面简化

基于 QEM 的曲面简化 (编号 2.9) 这个实验中, 你将实现一种曲面简化算法, 可以减少一个网格中顶点、边和面片的数量, 让网格更加简单。

### 3.1 实验内容

理解使用边坍塌操作简化网格的过程, 尤其是如何以误差度量结果为依据进行贪心选择的过程, 并实现曲面简化算法。

### 3.2 指导和要求

#### 3.2.1 用贪心思想简化网格

在局部操作 (实验 2.7) 中, 你已经实现了能减少面片数量的边坍塌操作。如果我们连续坍塌若干条边, 网格的面数也会在这个过程中不断减少, 从而达到简化网格的目的。

**问题 3.1** 用边坍塌操作简化网格时的关键问题:

- 决定坍塌哪些边
- 决定坍塌之后新顶点的位置

整体上, “简化一个网格并使结果尽可能接近原始网格”是个全局优化问题, 求解的难度很高。

**解** 我们用贪心思想来近似解决这个优化问题: 首先赋予每条边一个“代价”, 代表坍塌它之后产生的误差是多少; 然后每次从所有边中选择代价最小的边坍塌, 直到面数达到目标为止。

二次误差度量 (Quadric Error Metric, QEM) 是一种估算代价的方法, 使用这种方法可以同时估计出每条边最佳的位置 (新顶点位置) 和这一点到原网格的距离 (坍塌代价)。在简化过程中, 我们维护一个优先队列来高效地选取代价最小的边。

#### 3.2.2 二次误差度量

**问题 3.2** 如何衡量简化后的网格与原始网格之间的误差呢?

**解** 每次坍塌将一条边变为一个点, 因此这个“误差”可以取点  $\mathbf{x}$  (坍塌后) 到网格 (坍塌前) 的距离。

进一步地, 坍塌操作后新增的顶点的位置是受限的。假如坍塌之前边的两个端点分别是  $v_i, v_j$ , 那么  $\mathbf{x}$  一定在  $v_i, v_j$  的连线上。假设  $\mathcal{N}_{1,f}(v)$  表示任意顶点  $v$  的所有邻接面片, 在一次坍塌操作后, 只有  $\mathcal{N}_{1,f}(v_i) \cup \mathcal{N}_{1,f}(v_j)$  中的面片发生变化 (被删去或变形)。这意味着我们不必度量  $\mathbf{x}$  到整个网格的距离, 只要度量  $\mathbf{x}$  到  $\mathcal{N}_{1,f}(v_i) \cup \mathcal{N}_{1,f}(v_j)$  的距离就能衡量一次坍塌操作引入的误差。

##### 定义 3.1 ( $\mathcal{N}_1$ 邻域)

计算几何领域常用某环邻域 (也就是  $x$ -ring neighborhood) 来代表一个顶点周围的几何元素。最小的邻域是一环邻域 (1-ring neighborhood), 表示某个顶点周围“一圈”的范围。顶点  $v$  的 1-ring neighborhood 通常记作  $\mathcal{N}_1(v)$ 。

为了书写方便, 我们在后续的文档中用  $\mathcal{N}_{1,v}$  表示  $\mathcal{N}_1$  中的顶点、 $\mathcal{N}_{1,f}$  表示  $\mathcal{N}_1$  中的面片。一个顶点的  $\mathcal{N}_1$  邻域如图 3.1 所示。

首先从构造点到三角形的距离开始。当点离三角形不太远时, 点到三角形的距离可以近似为点到平面的距

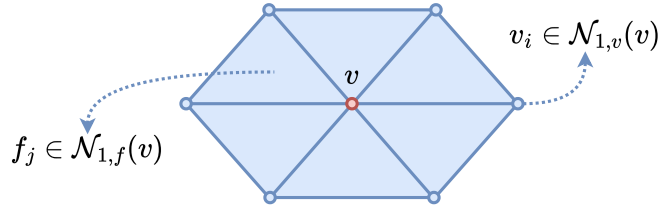


图 3.1: 顶点  $v$  的  $\mathcal{N}_1$  邻域示意图，图中所有蓝色部分都属于  $\mathcal{N}_1$  邻域。

离：设平面法向为  $\mathbf{N}$ ，点  $\mathbf{x}$  沿平面法向在平面上的投影为  $\mathbf{p}$ ，那么点  $\mathbf{x}$  到该平面的距离就是  $\text{dist}(\mathbf{x}) = \mathbf{N} \cdot (\mathbf{x} - \mathbf{p})$ ，如图 3.2 所示。

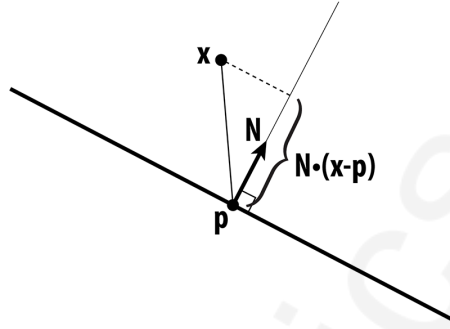


图 3.2: 点到平面的距离

将点和法向的坐标拓展为齐次坐标后，这个距离可以改写成两个四维向量的点积：

$$\begin{aligned} \text{dist}(\mathbf{x}) &= \mathbf{u} \cdot \mathbf{v} \\ \mathbf{u} &= (\mathbf{x}, 1) \\ \mathbf{v} &= (\mathbf{N}, -\mathbf{N} \cdot \mathbf{p}) \end{aligned} \quad (3.1)$$

式 3.1 计算的是有向距离，但我们只关心远近，不关心方向，所以用这个距离的平方更合适：

$$(\mathbf{u} \cdot \mathbf{v})^2 = \mathbf{u}^T \mathbf{v} \mathbf{v}^T \mathbf{u} = \mathbf{u}^T (\mathbf{v} \mathbf{v}^T) \mathbf{u} \quad (3.2)$$

式 3.1 中定义的  $\mathbf{u}$  就是  $\mathbf{x}$  的齐次坐标，因此式 3.2 可以看作一个二次型，这个二次型的系数矩阵就是  $K_f = \mathbf{v} \mathbf{v}^T$ 。对任意的点  $\mathbf{x}$ ，一个恒定的系数矩阵  $K_f$  决定了  $\mathbf{x}$  到这个平面的距离，因此  $K_f$  是一个体现点-面距离的特征。方便起见，我们称之为面片的二次误差矩阵。

用二次误差矩阵估计  $\mathbf{x}$  到  $\mathcal{N}_{1,f}(v_i) \cup \mathcal{N}_{1,f}(v_j)$  的距离就得到式 3.3，用于度量一次坍塌引入的误差总和（注意，误差矩阵用于计算误差，它不是误差本身）。

$$K = \sum_{f \in \mathcal{N}_{1,f}(v_i) \cup \mathcal{N}_{1,f}(v_j)} K_f \quad (3.3)$$


但集合求并计算起来比较麻烦，我们可以再做一些近似处理，用式 3.4 代替式 3.3：

$$K \approx \sum_{f \in \mathcal{N}_{1,f}(v_i)} K_f + \sum_{f \in \mathcal{N}_{1,f}(v_j)} K_f = K_i + K_j \quad (3.4)$$

最终，将以  $v_i, v_j$  为端点的边坍塌到  $\mathbf{x}$  处带来的误差就是  $K = K_i + K_j$ 。

按照式 3.4 计算二次误差的好处在于：我们可以预先计算好每个顶点的  $K$  并放在一个哈希表中，在坍塌边时查找端点的  $K$ 、在坍塌后更新新增顶点的  $K$ ，这比每次遍历  $\mathcal{N}_{1,f}(v_i) \cup \mathcal{N}_{1,f}(v_j)$  更高效。

按照式 3.2, 将边坍塌到点  $\mathbf{x}$  的二次误差就是  $\mathbf{x}^T K \mathbf{x}$ 。按照多元函数求极值的方法, 对这个二次型求导并令导数为零可得  $K \mathbf{x} = \mathbf{0}$ , 解这个线性方程组就得到了最优坍塌位置。

 **笔记** 可以验证  $K$  是正定矩阵, 所以这里解出的极值点一定是极小值而非极大值。

### 3.2.3 实现思路和细节

实现一个贪心算法的过程不外乎计算代价、排序、选择、更新代价。本次实验中计算代价的过程就是求每条边的二次误差矩阵  $K$ , 而  $K$  由两个顶点的  $K_i, K_j$  累加得到, 每个顶点的  $K_i$  又是由若干面片的  $K_f$  求和得到。所以我们按照面片、顶点、边的顺序逐步累加, 就能求出每条边的  $K$ 。为此, 我们维护了两个哈希表 (用 `std::unordered_map` 实现):

- `face_quadrics` 是从 `Face*` 到 `Matrix4f` 的映射
- `vertex_quadrics` 是从 `Vertex*` 到 `Matrix4f` 的映射

初始化过程的第一步是根据每个面片的法向量 `f->normal()` 求出它的二次误差矩阵, 并更新 `face_quadrics[f]`; 然后累加求出每个顶点的二次误差矩阵, 更新 `vertex_quadrics[v]`。

在贪心的过程中, 我们每次都要选择代价最小的边, 也就需要将边动态地排序, 所以要用优先队列存储边。每次坍塌一条边后, 原先与它端点相邻的所有边都会变形, 它们的二次误差矩阵也都会改变。因此, 这个优先队列要支持高效地查找和更新元素, 最常用的堆实现不能满足要求, 这里我们用 `std::set` (二叉查找树) 实现优先队列。Dandelion 使用一个辅助类型 `EdgeRecord` 代替 `Edge` 参与排序, 它包含了坍塌代价和最优位置信息, 请阅读[开发者文档: HalfedgeMesh::EdgeRecord 结构体](#)了解它的定义。

初始化的第二步是为每条边构造一个 `EdgeRecord`, 然后加入到两个数据结构中:

- `edge_records` 是一个哈希表, 将 `Edge*` 映射到对应的 `EdgeRecord`
- `edge_queue` 是用于动态排序的优先队列

完成初始化后算法进入循环, 每次从优先队列中取出代价最小的边 `edge_queue.begin()` 并坍塌它。坍塌结束后, 更新队列中所有邻接边的 `EdgeRecord`。每坍塌一次减少两个面片, 面数减少到最初的 1/4 时退出循环。

在坍塌一条边时需要注意: 坍塌操作是不安全的, 一次不加检查的坍塌可能会破坏网格的流形性质。图 3.3 中, 两个蓝色的面在坍塌后重叠在了一起。

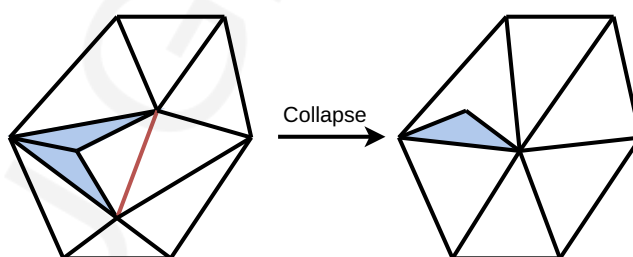


图 3.3: 破坏流形性质的坍塌操作

为了避免这种情况出现, 我们必须在坍塌前检查两个端点公共邻接顶点的数量, 当且仅当两个端点  $v_i, v_j$  公共邻接顶点数等于 2 时 (满足式 3.5 表示的条件), 这次坍塌才是安全的。图 3.3 中两端点的公共邻接顶点数为 3, 所以坍塌后会产生重叠的面片。

$$|\mathcal{N}_{1,v}(v_i) \cap \mathcal{N}_{1,v}(v_j)| = 2 \quad (3.5)$$

### 3.2.4 要求

你需要填写两个函数：

- 在 `HalfedgeMesh::simplify` 中实现曲面简化算法的主要流程
- 在 `HalfedgeMesh::EdgeRecord` 的构造函数中计算一条边的二次误差矩阵、误差值和最优坍塌位置

另外，你还需要在 `HalfedgeMesh::collapse_edge` 函数中增加判断安全性的代码，当坍塌操作不满足式 3.5 给出的条件时拒绝坍塌。本实验只需要简化封闭流形网格，不需要处理退化情况。

实现简化算法后，我们要求你加载 *cow.dae* 并连续简化三次，再加载 *bunny.obj* 并简化一次。

## 3.3 实验结果

如果你正确地实现了简化算法，那么在建模模式下点击 *Simplify* 按钮后，模型的面数会减少，但形状应该与原先比较接近。图 3.4 展示了简化两次的例子。

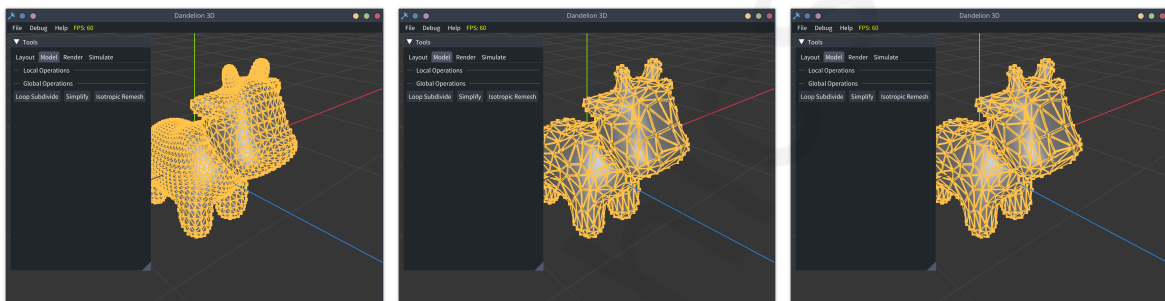


图 3.4: 将 *cow.dae* 连续简化两次的效果

## 3.4 提交和验收

需要提交的结果截图有两张：简化三次的 *cow.dae* 和简化一次的 *bunny.obj*，其他内容按照标准即可。

所有的简化测试必须依次完成，中途只允许删除物体（选中后按 `delete` 键）不能退出程序，因此日志文件中应当包含全部简化过程输出的日志。测试时日志级别设置为 `debug`。

验收时，你需要现场演示简化模型，并回答相关的问题。受到具体实现的影响，你的简化的结果并不一定与示例完全一致，但必须能保持物体原有的形状，简化正确可得 15 分。



## 第4章 各向同性重网格化

各向同性重网格化（编号 2.10）这个实验中，你将实现一种“重网格化”（remeshing）算法，将不规整的网格重构得更整齐。

### 4.1 实验内容

了解一些判断网格性质优劣的标准，熟练掌握三种局部操作并清楚每种操作带来的副作用，实现各向同性重网格化算法。

### 4.2 指导和要求

#### 4.2.1 评判网格的性质

不同的应用任务关注不同的网格性质，对于渲染或几何处理而言，一个面片均匀分布的三角形网格往往是比较好的。

什么是“均匀分布”呢？这个问题没有统一的答案，但有两条共识是被大多数人认可的：

- 没有非常“尖锐”的三角形。如果一个三角形的某个顶角非常小，那么它就很倾向于退化成一条线。渲染这样的面片时，可能会因为浮点舍入误差在图像上产生断裂、黑边等缺陷；修改这样的面片时，也容易因为近似平行的边、过小的面积而进行错误的几何操作，产生不良的几何结构。
- 没有度很大的顶点。一个顶点的“度”通常定义为邻接的面片（或顶点）数量，如果某个顶点的度非常大，那么修改它的位置会改变很多个面片的形状，而我们一般希望对顶点的修改只影响少数几个面片；如果某个顶点的度很大，那么它周围“一圈”面片很可能包含许多尖锐的顶角（比如一个二十棱锥有二十个侧面，每个侧面都是一个非常尖锐的三角形），翻转某条邻接边可能导致形状凹陷、面片大小不均等等不好的结果。

Botsch 等人在论文中 [1] 提出了一种比较理想的情况：

- 每个三角形尽可能接近等边三角形
- 每个顶点的度尽可能接近 6

以这种状态为目标重构网格的过程就是各向同性重网格化（isotropic remeshing），原文称 area-equalizing remeshing。

#### 4.2.2 重构过程

整个算法按照上面的标准重构网格，每一遍重构可以分为四步：

1. 分裂特别长的边
2. 坍塌特别短的边
3. 通过翻转一些边让顶点的度更加平均
4. 微调顶点的位置，使之更接近  $N_1$  邻域顶点的中心

连续重构几遍之后，网格就会变得比较“均匀”。在本次实验中，我们统一重构五遍。

在重构前，我们首先要决定“特别长”和“特别短”的标准。设网格中所有边的平均长度是  $L$ ，那么这个算法会分裂长度大于  $\frac{4}{3}L$  的边、坍塌长度小于  $\frac{4}{5}L$  的边。

回顾一下图 1.1 中翻转边的过程，翻转之后两个端点  $v_1, v_2$  的度减 1 而对向的两个顶点  $v_3, v_4$  的加 1。算法的目标是让所有顶点的度尽可能平均且接近 6，所以这四个顶点的评价指标就是

$$d = |\text{degree}(v_1) - 6| + |\text{degree}(v_2) - 6| + |\text{degree}(v_3) - 6| + |\text{degree}(v_4) - 6|$$

如果翻转某条边可以让  $d$  减小，那我们就翻转它。

在分裂、坍塌、翻转完成后，我们还可以微调顶点位置，如图 4.1 所示。

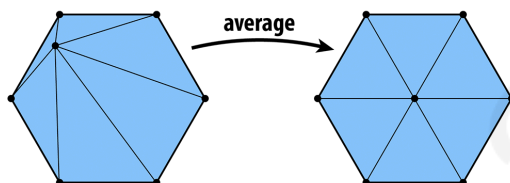


图 4.1: 调整顶点位置（拉普拉斯平滑）的过程示意

但图 4.1 只画出了平面上的情况，三维空间与之有所不同。如果某个顶点与其邻接点构成一个棱锥，那么直接将它移动到这个中心就会把棱锥“抹平”，在很大程度上改变模型的形状，这是不好的。各向同性重网格化算法采取了两项措施限制形变幅度：

- 减小移动的幅度。设  $\mathbf{p}$  和  $\mathbf{c}$  分别代表顶点位置和中心位置，重构算法不会一次性将顶点移动到  $\mathbf{c}$ ，而是将它移动到  $\mathbf{p} + w(\mathbf{c} - \mathbf{p})$ 。在本实验中，我们取  $w = 1/5$ 。
- 只沿着切线方向移动。如果某个顶点是棱锥的尖角，那么它的法向一定比较接近棱锥凸起的方向，沿着切向移动自然可以避免“抹平”棱锥。记  $\mathbf{v} = \mathbf{c} - \mathbf{p}$ ，顶点处的法线为  $\mathbf{N}$ ，重构时用  $\mathbf{v} - (\mathbf{N} \cdot \mathbf{v}) \cdot \mathbf{N}$  代替  $\mathbf{v}$  作为移动方向，即可在切向移动顶点。

### 4.2.3 实现思路和细节

Dandelion 提供了 `Vertex::neighborhood_center` 函数用于计算一个顶点  $\mathcal{N}_1$  领域内邻接点的中心位置，你可以直接使用它。由于计算中心位置依赖于平滑前的顶点位置，在进行平滑时你不应该直接改变顶点的坐标 (`Vertex::pos`)，而应该暂时将新的坐标赋值给 `Vertex::new_pos` 属性，在全部计算完后再将每个顶点的 `new_pos` 赋值给 `pos`。

按照前述步骤写代码时要注意：后面的步骤不能与之前的步骤冲突。坍塌一条边会导致邻接边的长度增加，如果坍塌后邻接边的长度大于  $\frac{4}{3}L$ ，那就不应该坍塌这条边。请仔细考虑几种局部操作之间的影响，否则重构结果可能会非常混乱，甚至直接导致程序死循环、崩溃等。

### 4.2.4 要求

你需要实现 `src/geometry/meshedit.cpp` 中的 `HalfedgeMesh::isotropic_remesh` 函数。

实现重网格化算法后，你需要加载 `cow.dae` 并进行一次重网格化。如果你有兴趣，也可以尝试重网格化 `bunny.obj` 和 `dragon2.dae`（后者大约有 36 万个三角形面，请使用 Release 模式编译后再尝试）。如果你实现了细分或简化算法，我们也鼓励你尝试重网格化细分（或简化）之后的结果，探究网格的面数与形状对重网格化有什么影响。本实验只要求对封闭流形网格进行重网格化处理。

## 4.3 实验结果

如果你正确地实现了各向同性重网格化算法，那么进入建模模式点击 *Isotropic Remesh* 按钮后应该会看到模型表面的三角形被“平均”了，如图 4.2 所示。



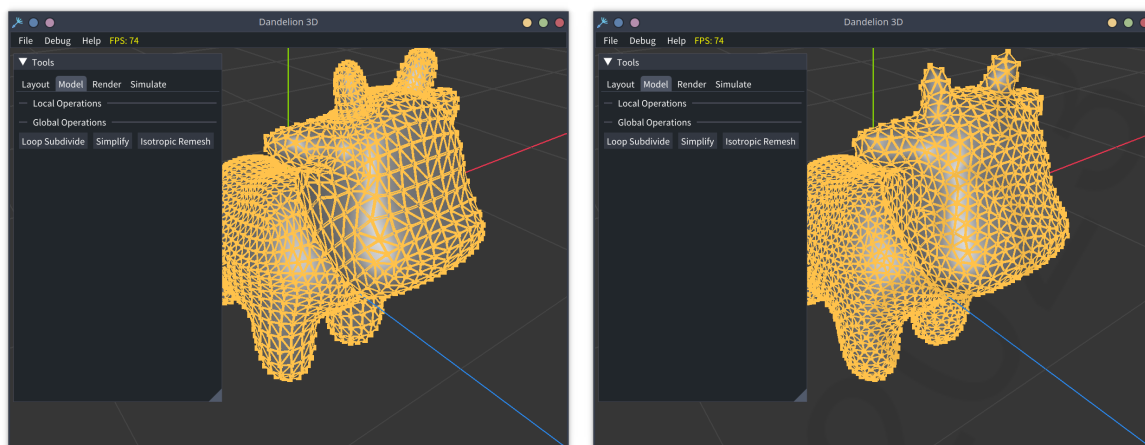


图 4.2: 重网格化 *cow.dae* 的结果示例

## 4.4 提交和验收

需要提交的截图只有一张：重网格化 *cow.dae* 的结果，其他按标准提交即可。

验收时需要现场演示重网格化并回答相关问题。如果你能够正确地分裂过长的边，可以得 5 分；如果能正确坍缩过短的边，可以再得 5 分；如果能正确地翻转边并平滑顶点，可以再得 10 分，总计 20 分。

## 第 5 章 对四边形网格的支持

使 Dandelion 支持四边形网格（编号 2.11）是一个挑战任务，你需要为框架的加载、存储、渲染和几何处理部分添加对四边形网格 (quad-mesh) 的支持。

### 5.1 实验内容

在渲染与几何处理过程中，最常用、性质也最好的是三角形网格，但人工建模时四边形网格更方便美术工作者调整形状，因此很多模型资源是四边形网格（严格来说，是三角形与四边形混合的网格）。

目前 Dandelion 中存储和渲染网格所用的是同一个数据结构（参阅[开发者文档：GL::Mesh 结构体](#)），它采用数组加索引的方式存储网格，并且只能存储三角形网格。当程序进入建模模式时，会为选中的物体构造一个半边网格实例（参阅[开发者文档：HalfedgeMesh 类](#)）。根据建模模式下修改操作的类型，半边网格会将修改结果通过局部更新或全局更新的方式同步到对应的网格。

你的任务是添加基本的四边形网格支持（但这已经需要很多工作），这很可能会花费你上百个小时。我们乐于为愿意付出的同学提供帮助，但选择时请谨慎考虑。

### 5.2 指导和要求

由于 OpenGL 只能渲染三角形，你必须大幅度改变现有的数据结构：增加存储网格的通用数据结构（支持三角形和四边形混合存储）来替代 `GL::Mesh` 存储真正的网格数据，并且将每个四边形面映射为两个三角形面以送入 OpenGL API 进行渲染。由于建模模式会修改网格数据，你还需要在底层渲染数据、网格数据、半边网格数据之间进行同步，保证三者的一致性。

半边网格本身完全可以支持任意多边形网格，但之前实现的局部操作只考虑了三角形网格的情况。为此你需要将三个局部操作拓展到四边形网格，并对操作四边形网格时可能出现的退化情况进行检查。几何处理部分实现的三个全局操作都是只接受三角形网格的算法，所以你需要为 `HalfedgeMesh` 类增加检查自身是否为三角形网格的功能，以便这些函数在开头检查当前网格是否含有四边形。

在建模模式下拾取网格的顶点、边和面片的操作是通过射线求交实现的，而这部分代码也只考虑了三角形的情况。你需要实现射线与四边形求交（或调整射线-mesh 的求交过程，使之与底层渲染数据中的三角形求交）、修改拾取操作的判断逻辑，才能正确地选中物体或几何基本元素。

本实验没有详细的指导，请参考开发者文档并阅读源代码来完成实验。实验过程中如果遇到任何问题，欢迎与助教沟通。

### 5.3 提交和验收

本实验只需要提交修改后的源代码和 `CMakeLists.txt` 文件。

在验收时，请现场演示四边形网格的加载和编辑功能，并向助教解释你实现这个过程的思路。如果你能够在布局模式下加载并显示四边形网格，可以得到 25 分；如果你能够在建模模式下移动四边形网格的顶点、进行局部操作，可以再得到 25 分，总计 50 分。

如果你的代码质量比较优秀，我们会将你的代码并入 Dandelion 并将你列入 Dandelion 贡献者列表中。根据你的意愿，我们可以在 GitHub 仓库页面、开发者文档或实验文档上留下你的名字。

## 参考文献

- [1] Mario Botsch and Leif Kobbelt. “A remeshing approach to multiresolution modeling”. In: *Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*. 2004, pp. 185–192.
- [2] Charles Loop. “Smooth subdivision surfaces based on triangles”. MA thesis. University of Utah, 1987.
- [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](http://staff.ustc.edu.cn/~renjie/CG_2021S2/default.htm).