图形学实验报告

实验 3：Mesh Subdivision

霍超凡

实验内容

- 阅读源码并使它通过编译，
- 实现 Loop Subdivision 方法和 butterfly interpolating subdivision 方法，
- 使用提供的 obj 文件测试、对比这两个方法。

算法原理

关于这两个算法课上没有听清楚，PPT 又太精炼，原始论文太难懂，我自己找到一些博客把这个算法搞明白了。

Loop Subdivision

Loop Subdivision 细分算法包含三个步骤：插入边点、移动原始顶点和重新划分网格，最终使网格内的一个三角形细分成四个小三角形。

1. 插入边点，在插入边点这个步骤中，在每一条边上插入一个顶点，这个顶点的位置通过如下规则取得，如果这条边位于网格的边界或者是 crease 边，则新加入的点v_{anchor}为这条边两个端点v_1, v_2的中点，即

$$v_{anchor} = \frac{1}{2}(v_1 + v_2)$$

如果这条边位于网格的内部，那么这条边一定会邻接两个三角形，新插入的点为这两个三角形四个顶点的加权平均，即

$$v_{anchor} = \frac{3}{8}v_1 + \frac{3}{8}v_2 + \frac{1}{8}v_3 + \frac{1}{8}v_4$$

2. 移动原始顶点，根据第一步加入的边点重新计算原始顶点的位置，这个顶点v_{old}和该顶点的相邻边点加权求和得到该点新的位置v_{new}，如果这个顶点位于边界，

$$v_{new} = \frac{3}{4}v_{old} + \frac{1}{8}v_1 + \frac{1}{8}v_2$$

如果这个顶点位于网格的内部，

$$v_{new} = (1 - k\beta)v_{old} + \sum_{\mathrm{i}=1}^{\mathrm{k}} \beta v_i$$

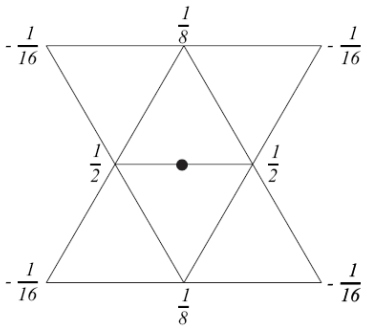
其中，k为该顶点相邻边点的个数，$\beta = \frac{1}{\mathrm{k}}\left(\frac{5}{8} - \left(\frac{3}{8} - \frac{1}{4}\cos\frac{2\pi}{k}\right)^2\right)$。

3. 重新划分网格，使用前两步得到的顶点重新划分网格，使网格内的每一个三角形变成三个小三角形。

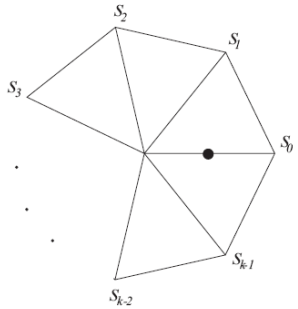
Butterfly Interpolating Subdivision

这个方法和 Loop subdivision 方法相似，只包含两大步骤：插入边点和重新划分网格。插入边点这个步骤按照边两个顶点的度数分成四种情况，

（1） $degree(v_1)=6, degree(v_2)=6$，则按照下图掩码计算边点的位置。

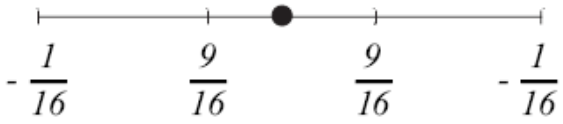


（2） $degree(v_1)=6, degree(v_2)\neq 6$或者$degree(v_2)=6, degree(v_1)\neq 6$，则按照下图掩码计算边点的位置。



（3） $degree(v_1)\neq 6, degree(v_2)\neq 6$，则按照规则（2）计算改变两个端点的位置，然后取平均。

（4） 边位于边界或者是 crease 边，则按照下图掩码继续计算。



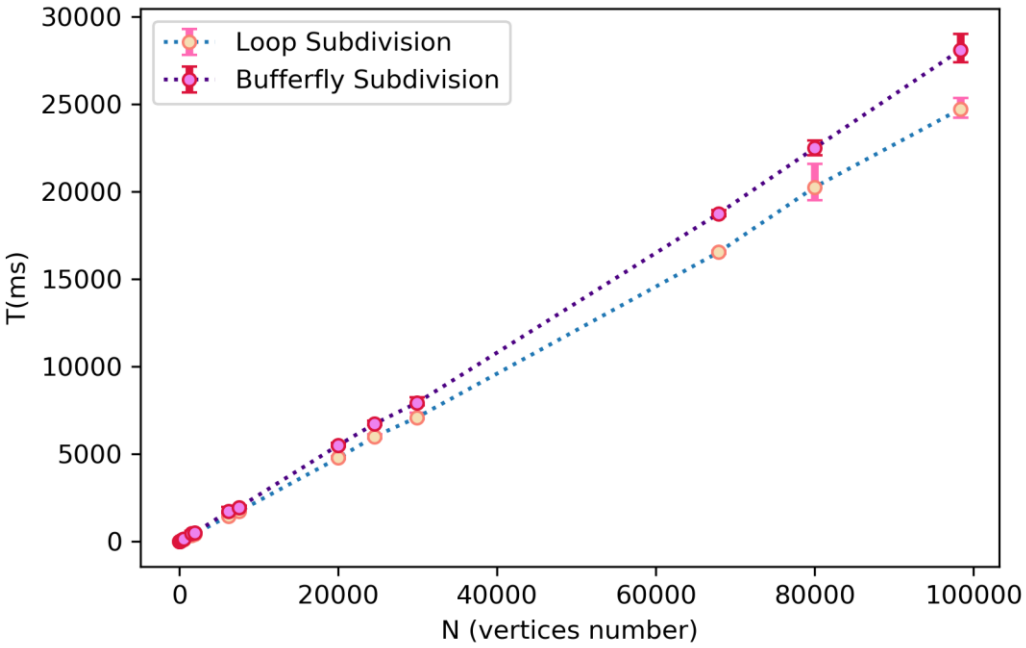
关于算法实现的具体细节，由于涉及到大量代码，我把他们放到了附录中。

最终结果

算法效率

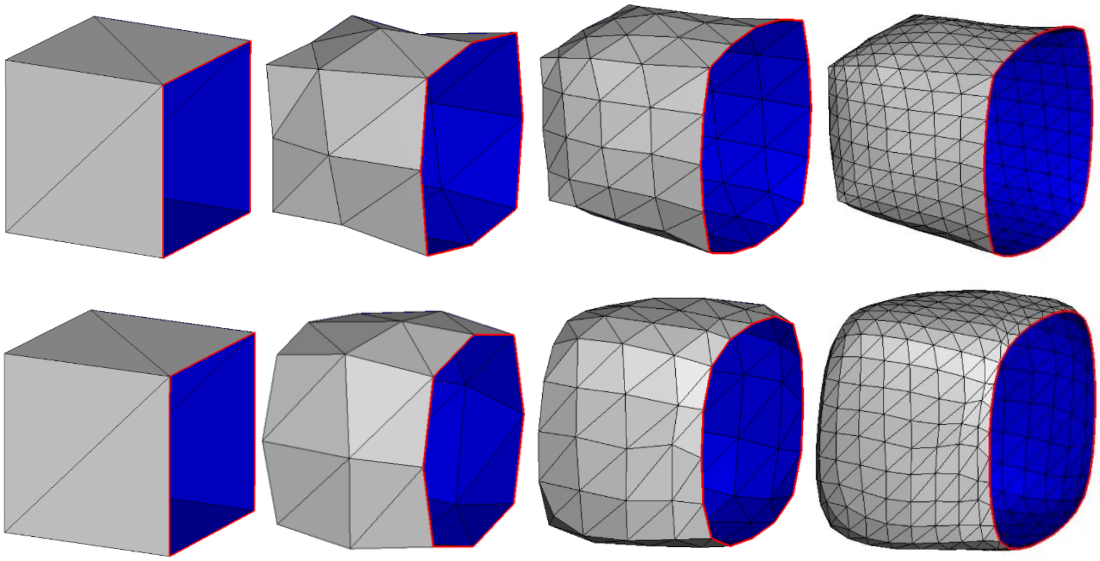
在编写完两个算法后，我测试了这两个算法运行时间效率并做了对比，实验中所测得的

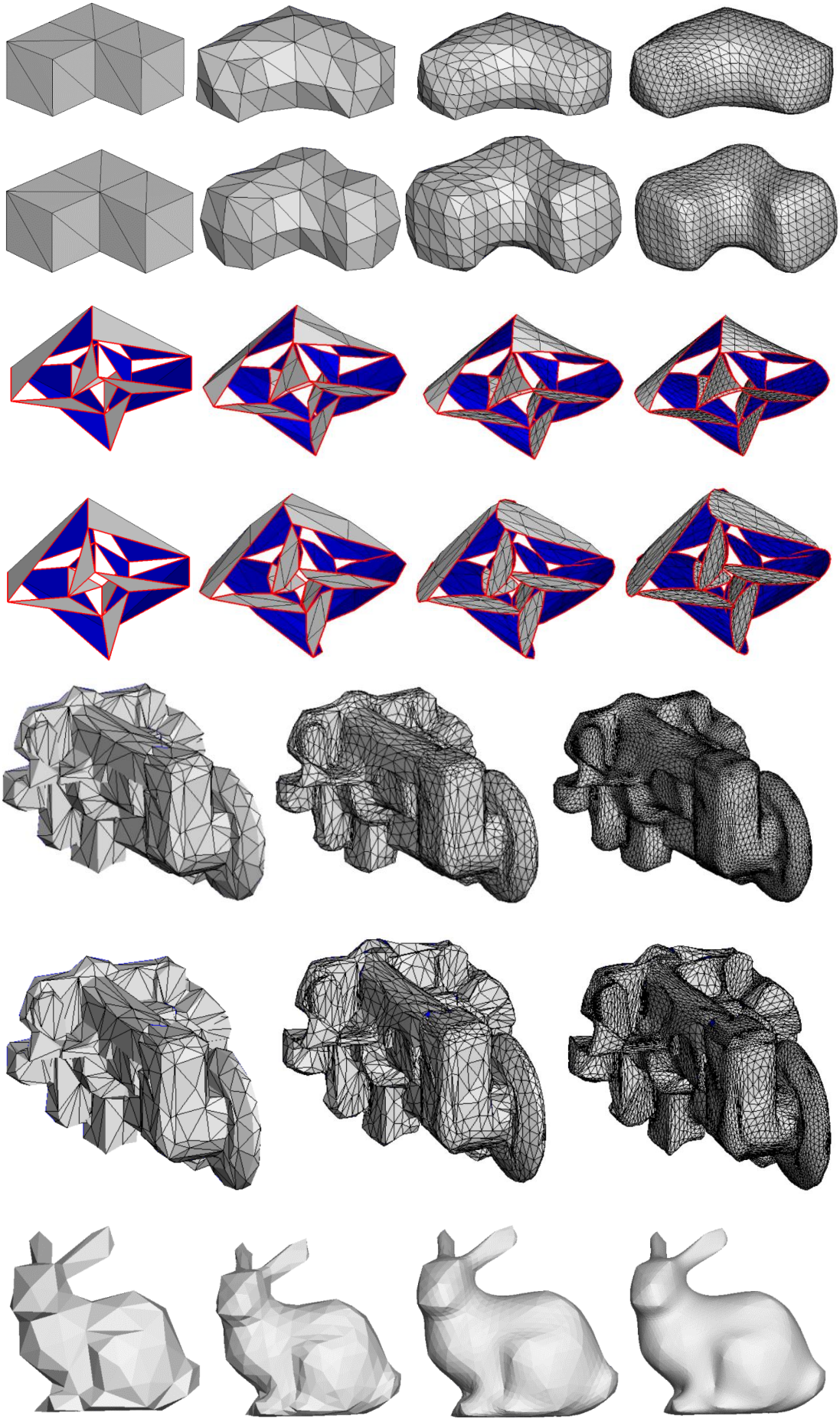
数据是依赖我所实现的方法，我不能保证以下结果真正符合这两个算法的效率。取不同的模型测试算法，计算模型中顶点的个数和进行一次划分所用的时间，相同模型重复测量 5 次时间，最终结果取平均。以模型中顶点个数N为横坐标，算法细分一次所耗费的时间为纵坐标，得到下图结果，从结果中我们可以看到我们这两个算法相对顶点数 N 的时间复杂度为$O(n)$，均为线性时间复杂度，loop subdivision 计算量稍小一点，Butterfly Subdivision 中计算边点位置时需要遍历顶点的所有相邻顶点，case B 的权重计算也比较大，而 loop subdivision 计算边点和新的顶点的位置比较简单，所以 loop subdivision 的效率要略高于 butterfly subdivision。

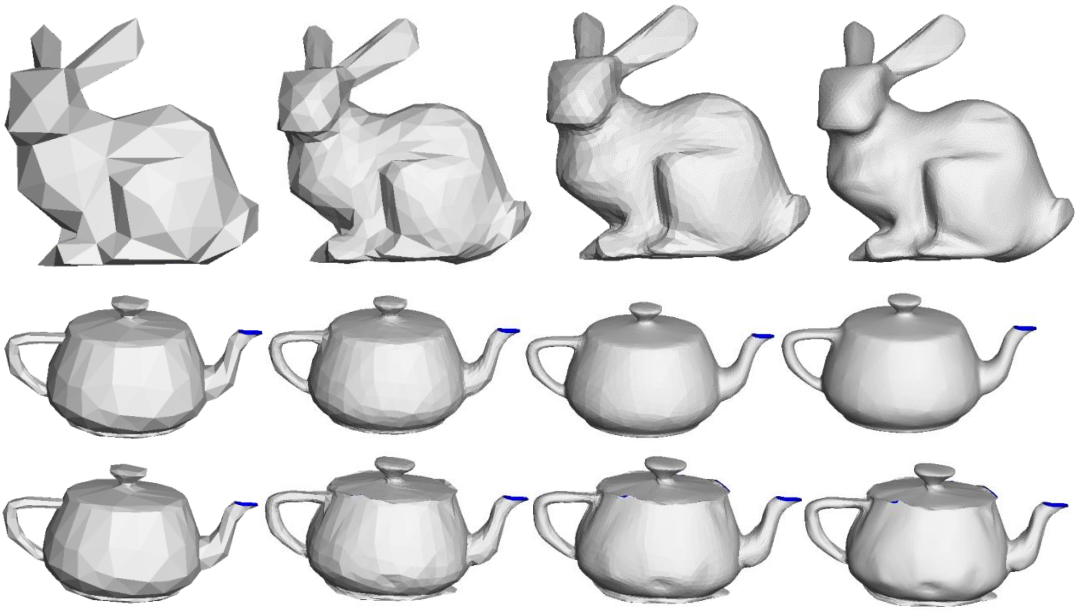


网格模型细化

偶数行为 Butterfly Interpolating Subdivision，奇数行为 Loop Subdivision。







通过这几组模型细化的结果，可以看到 Butterfly Interpolating Subdivision 细化的模型棱角更加明显，而 Loop Subdivision 细化的模型轮廓非常平滑、模糊。

参考资料

[1] Loop 细分曲面（loop subdivision）详解，https://blog.csdn.net/McQueen_LT/article/details/106136324

[2] 改进的蝴蝶算法的详细介绍，https://www.cnblogs.com/yezhangxiang/archive/2011/04/10/2011284.html

[3] 半边数据结构与网格细分算法 Loop subdivision，https://blog.csdn.net/fengluod/article/details/83821348

附录：算法实现细节

数据结构

实验所提供的代码中 mesh 采用的是课堂中所讲的半边结构，这种数据结构包含两个部分，一是顶点集合$\text{Vertices} = \{v_1, v_2, \ldots, v_n\}$，另外一部分是顶点之间边角的关系，存储顶点间指针所使用的数据结构是 map 映射，顶点对到边的映射$\text{Edges}: (v_1, v_2) \to e_{12}$，三角形索引到三角形指针的映射$\text{Triangles}: i \to t_i$，其中数据结构$e_{12}$包含了两个顶点$v_1, v_2$的指针、反向边$e_{21}$、下一条边$e_{23}$以及改变所属的三角形，三角形$t_i$数据结构只包含了三角形其中一条边的指针。在编程前我们不需十分熟悉这几种数据结构之间的关系，mesh 类还提供了组织这些数据结构的方法，比如往顶点集合加入新顶点的方法 addVertex，加入三角形的方法 addTriangle，移除三角形的方法 removeTriangle，注意到 mesh 并没有直接提供增删边关系的方法，增删边点关系的功能被封装在处理三角形的那两个函数里，在实现细化方法时，我们将会频繁使用这几种方法。

Loop Subdivision 方法

之前已经提及 Loop Subdivision 方法的步骤，步骤被划分成三大步：插入边点、修改原始顶点、重新建立网格，Loop Subdivision 方法如下所示，

```
void Mesh::LoopSubdivision()
{
    printf("Subdivide the mesh! (Default: Loop subdivision) \n");
    addEdgeAnchor();
    moveOldVertices();
    splitTriangles();
}
```

加入边内锚点

在 Loop Subdivision 的第一步中，我们需要往每一条边内插入一个新点，使用代码提供的 vertex_parents 映射来建立两个顶点和刚刚插入边点之间的映射关系，

```
void Mesh::addEdgeAnchor()
{
    for (edgeshashtype::iterator iter = edges.begin(); iter != edges.end(); iter++)
{
        Edge* e = iter->second;
        Vertex* p1 = e->getStartVertex();
        Vertex* p2 = e->getEndVertex();

        Vertex* anchor = getChildVertex(p1, p2);
        if (anchor == NULL) {
```

```
                anchor = addLoopAnchor(e);
                setParentsChild(p1, p2, anchor);
            }
        }
    }
```

addEdgeAnchor 中遍历 Mesh 内的所有边，计算边点，边点的计算分成两种情况：该边属于内部边、该边位于边界，两种不同的情况分别对应不同的插值计算方式，计算边点被封装在 getChildVertex 函数内，这里不再展示。得到边点之后再使用 setParentChild 标记这个刚刚被加入但还没有被划分三角形的顶点，注意到正向边和反向边共用相同的顶点，在加入边点的时候，要判断时候之前是否由于遍历反向边而已经加入了该点。

移动旧有顶点

原始代码中并没有提供遍历顶点的方法，我们需要在遍历顶点的过程中，既能看到顶点也能找到点边关系，我们可以遍历边或者遍历三角形，这里在移动旧有顶点时，遍历的三角形，这里之所以选择遍历三角形而不选择遍历边是因为遍历三角形会减少重复遇到同一个地点的次数，具体函数如下，

```
void Mesh::moveOldVertices()
{
    int n = numVertices();
    bool* done = new bool[n];
    for (int i = 0; i < n; i++) {
        done[i] = false;
    }

    triangleshashtype::iterator iter = triangles.begin();
    for (; iter != triangles.end(); iter++) {
        Triangle* t = iter->second;

        Vertex* p = (*t)[0];
        if (!done[p->getIndex()]) {
            moveLoopVertex(p, t->getEdge());
            done[p->getIndex()] = true;
        }

        p = (*t)[1];
        if (!done[p->getIndex()]) {
            moveLoopVertex(p, t->getEdge()->getNext());
            done[p->getIndex()] = true;
        }

        p = (*t)[2];
```

```
        if (!done[p->getIndex()]) {
            moveLoopVertex(p, t->getEdge()->getNext()->getNext());
            done[p->getIndex()] = true;
        }
    }

    delete[] done;
}
```

为了防止重复遍历同一个点多次，我们使用一个 bool 数组来记录处理过的点，通过遍历网格中所有的三角形，分别对三角形三个顶点应用 moveLoopVertex 来修改顶点的坐标，顶点坐标的修改分成两种情况，之前也已经说过，这里代码不再展示。

重新建立三角形

重新建立三角形这个步骤要逐个遍历网格中的所有三角形，将每个三角形按照之前已经计算出的边点和被修改的顶点划分成四个小三角形，将这四个小三角形插入到网格中，并删除原有的大三角形，具体代码如下，

```
void Mesh::splitTriangles()
{
    std::vector<Triangle*> todo;
    triangleshashtype::iterator iter = triangles.begin();
    triangleshashtype::iterator end = triangles.end();
    for (; iter != end; iter++) {
        Triangle* t = iter->second;
        todo.push_back(t);
    }
    int num_triangles = todo.size();
    for (int i = 0; i < num_triangles; i++) {
        Triangle* t = todo[i];
        Vertex* p1 = (*t)[0];
        Vertex* p2 = (*t)[1];
        Vertex* p3 = (*t)[2];
        Vertex* anchor1 = getChildVertex(p1, p2);
        Vertex* anchor2 = getChildVertex(p2, p3);
        Vertex* anchor3 = getChildVertex(p3, p1);
        addTriangle(p1, anchor1, anchor3);
        addTriangle(p2, anchor2, anchor1);
        addTriangle(p3, anchor3, anchor2);
        addTriangle(anchor1, anchor2, anchor3);
        removeTriangle(t);
    }
}
```

Butterfly Interpolating Subdivision 方法

该方法分成两大步骤，插入新的边点，再 remesh，方法如下，

```
void Mesh::BufferflySubdevision()
{
    printf("Subdivide the mesh! (Buferfly subdivision) \n");
    interpolatePoints();
    splitTriangles();
}
```

该方法中的 splitTriangles 和 Loop Subdivision 算法中的完全相同，interpolatePoints 方法比较复杂，复杂在于需要分成许多情况，下面将详细介绍。

插入边点

该方法仍然像 Loop Subdivision 方法那样处理，逐步遍历网格中的所有边，计算新的边点，建立边点和两个顶点的父子关系，方法如下所示，

```
void Mesh::interpolatePoints()
{
    for (edgeshashtype::iterator iter = edges.begin(); iter != edges.end(); iter++)
{
        Edge* e = iter->second;
        Vertex* neighbor1a = e->getStartVertex();
        Vertex* neighbor1b = e->getEndVertex();

        if (getChildVertex(neighbor1a, neighbor1b) != NULL) {
            continue;
        }

        if (e->getOpposite() == NULL) { // for case (d), boundary and crease
            Vertex* interpolatedVertex = bufferflyCaseD(e);
            setParentsChild(neighbor1a, neighbor1b, interpolatedVertex);
        }
        else {
            std::vector<Vertex*> neighbors_a;
            getNeighbors(e, neighbors_a);

            std::vector<Vertex*> neighbors_b;
            getNeighbors(e->getOpposite(), neighbors_b);

            Vertex* interpolatedVertex = NULL;
            if (neighbors_a.size() == 6 && neighbors_b.size() == 6) {
                interpolatedVertex = bufferflyCaseA(e, neighbors_a, neighbors_b);
```

```
            }
            else if (neighbors_a.size() != 6 && neighbors_b.size() == 6) {
                interpolatedVertex = bufferflyCaseB(e, neighbors_a, neighbors_b);
            }
            else if (neighbors_a.size() == 6 && neighbors_b.size() != 6) {
                interpolatedVertex = bufferflyCaseB(e->getOpposite(), neighbors_b,
neighbors_a);
            }
            else if (neighbors_a.size() != 6 && neighbors_b.size() != 6) {
                interpolatedVertex = bufferflyCaseC(e, neighbors_a, neighbors_b);
            }
            setParentsChild(neighbor1a, neighbor1b, interpolatedVertex);
        }
    }
}
```

该方法遍历所有边，先找到边中两个顶点所有的邻近点，后按照临近点的个数划分成四种情况，针对不同的情况有不同的计算插值方式，这里不再展示代码，最后这里只说一下寻找所有临近顶点这个比较复杂的方法。

遍历所有邻接边以及顶点

在遍历所有邻接边及顶点时，我们需要考虑两种情况，一种是所有邻接边没有一个是边界边，另外一种情况是，存在属于边界边的邻接边，第一种情况比较好处理，我们只需要按照正常的指针走向即可遍历完所有的邻接边，在遍历第二种情况的邻接边时，中间会出现中断的情况，我们还需要反向在遍历剩余的边。下面给出了代码，

```
void Mesh::getNeighbors(Edge* e, std::vector<Vertex*>& neighbors)
{
    Edge* e_iter = e;
    neighbors.push_back(e_iter->getEndVertex());
    while (e_iter->getOpposite() != NULL &&
        e_iter->getOpposite()->getNext() != e) {
        e_iter = e_iter->getOpposite()->getNext();
        neighbors.push_back(e_iter->getEndVertex());
    }

    if (e_iter->getOpposite() == NULL) {
        std::vector<Vertex*> _neighbors;
        e_iter = e->getNext()->getNext();
        _neighbors.push_back(e_iter->getStartVertex());
        while (e_iter->getOpposite() != NULL) {
            e_iter = e_iter->getOpposite()->getNext()->getNext();
            _neighbors.push_back(e_iter->getStartVertex());
```

```
18              }
19              for (int i = _neighbors.size() - 1; i >= 0; i--) {
20                  neighbors.push_back(_neighbors[i]);
21              }
22          }
23      }
```

代码的第一个 while 循环中，按照正常的顺序遍历邻接边，在遍历的过程中要注意停止条件，运行完第一个 while 循环后，e_iter 有两种情况，如果遍历了一圈，那么 e_iter 的下一条邻接边将会重新回到 e，如果为 e_iter 的反向边为空，这说明我们到了边界，我们还需要反向重新遍历剩下的边，第二个 while 为反向遍历剩余边的代码，遍历时要注意指针的指向和终止条件。

以上就是 Loop Subdivision 和 Butterfly Interpolating Subdivision 的代码，其中有些简单的函数没有展示，欢迎查看实验所附的源码，在编码过程中，把一个较为复杂的函数分成了多个子过程，代码逻辑清晰，可读性增强，但是这不能保证效率不受损失，在编完所有代码后，试图对方法做出优化，可是能力有限，又受思维定势的影响，想不出更好的编码方案。