# RectMesh构建

BR顶点

TL顶点

BR顶点

TR顶点

BL顶点

TL顶点

注：

1. 暗色face是虚拟的，只是为了保证HalfEdge逻辑的完整性，边界情况无须判空逻辑
2. 为了可视化，这里将虚face做了翻转，其实就2个虚face，同一颜色的是同一个虚face
3. 边界4条edge的oppositeEdge都是虚edge
4. 红色矩形为ConstraintShape，该shape有4条ConstraintSegment

BR顶点

TL顶点

4个face（2实+2虚），所有face都是逆时针序，具体信息如下：

|  |  |  |
| --- | --- | --- |
| **face** | **edge** | **isReal** |
| fTL\_BL\_TR | eBL\_TR | true |
| fTR\_BL\_BR | eTR\_BL | true |
| fTL\_BR\_BL | eBR\_BL | false |
| fTL\_TR\_BR | eTL\_TR | false |

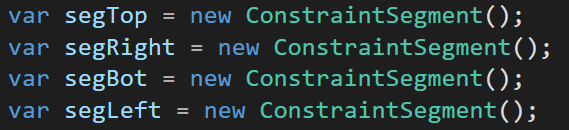
总共12条edge（10实+2虚，只有对角线2条edge是虚的，8条约束边+4条非约束边），具体信息如下：

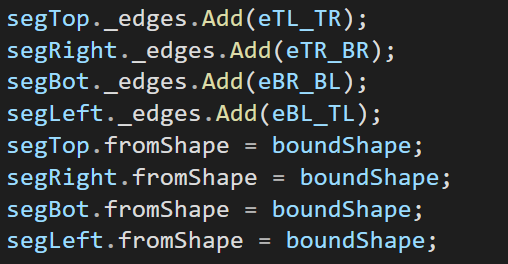
（注：nextLeftEdge是edge逆时针方向的下一个相连的edge）

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **edge** | **originVertex** | **oppositeEdge** | **nextLeftEdge** | **leftFace** | **isReal** | **isConstrained** |
| eTL\_TR | vTL | eTR\_TL | eTR\_BR | fTL\_TR\_BR | true | false |
| eTR\_TL | vTR | eTL\_TR | eTL\_BL | fTL\_BL\_TR | true | false |
| eTR\_BR | vTR | eBR\_TR | eBR\_TL | fTL\_TR\_BR | true | true |
| eBR\_TR | vBR | eTR\_BR | eTR\_BL | fTR\_BL\_BR | true | true |
| eBR\_BL | vBR | eBL\_BR | eBL\_TL | fTL\_BR\_BL | true | true |
| eBL\_BR | vBL | eBR\_BL | eBR\_TR | fTR\_BL\_BR | true | true |
| eBL\_TL | vBL | eTL\_BL | eTL\_BR | fTL\_BR\_BL | true | true |
| eTL\_BL | vTL | eBL\_TL | eBL\_TR | fTL\_BL\_TR | true | true |
| eTR\_BL | vTR | eBL\_TR | eBL\_BR | fTR\_BL\_BR | true | false |
| eBL\_TR | vBL | eTR\_BL | eTR\_TL | fTL\_BL\_TR | true | false |
| eTL\_BR | vTL | eBR\_TL | eBR\_BL | fTL\_BR\_BL | false | false |
| eBR\_TL | vBR | eTL\_BR | eTL\_TR | fTL\_TR\_BR | false | false |

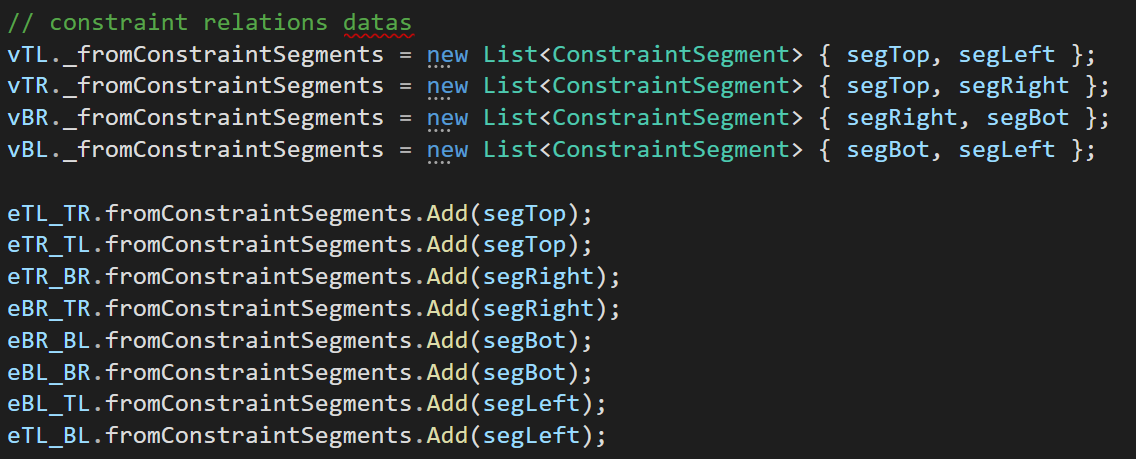
红色矩形为ConstraintShape，该shape有4条ConstraintSegment（顺时针序？）

（1）构建Segement和Edge关系（1对N关系）

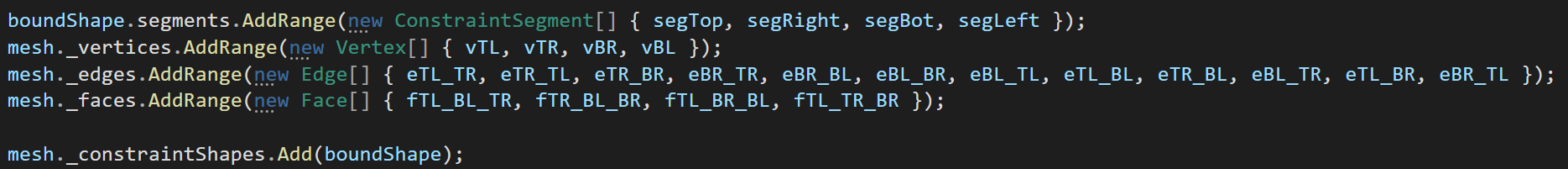




（2）关联Vertex和Edge和Segment，方便删除Segment使用，1个顶点对应2个segment，1个edge对应1个segment



构建ConstraintShape和Mesh数据



# Mesh. insertConstraintSegment

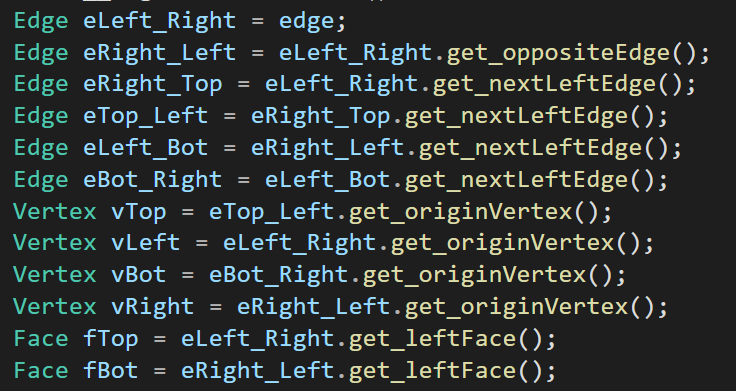
## Mesh. insertVertex

### Geom2D.locatePosition

基于传入的坐标点（x，y）找到和当前Mesh的相交信息（可能是Vertex，Edge，Face这3种情况）

1. 如果找到的是Vertex，直接返回该顶点即可，insertVertex完成
2. 如果找到的是Edge，则执行Mesh.splitEdge函数，将edge分割
3. 如果找到的是Face，则执行Mesh.splitFace函数，将face分割

## Mesh.splitEdge



vLeft

vRight

vTop

vBot

新的顶点命名为vCenter

fTop

fBot

edge

**vCenter**

如图edge（属于fTop）被1分为2，两个相邻face被分成4个face。新增的Vertex，Edge，Face以及原来的4个邻接（如图灰色）需要重新构建HalfEdge关系

1. 新增1个Vertex：vCenter
2. 新增8个Edge：eTop\_Center，eCenter\_Top，eBot\_Center，eCenter\_Bot，eLeft\_Center，eCenter\_Left，eRight\_Center，eCenter\_Right
3. 新增4个Face：fTopLeft，fLeftBot，fBotRight，fRightTop

构建HalfEdge关系如下：

vCenter.setDatas((fTop.get\_isReal()) ? eCenter\_Top : eCenter\_Bot);

vCenter.\_pos.X = x;

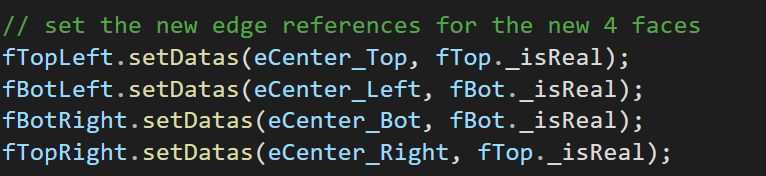
vCenter.\_pos.Y = y;

Geom2D.projectOrthogonaly(ref vCenter.\_pos, eLeft\_Right);

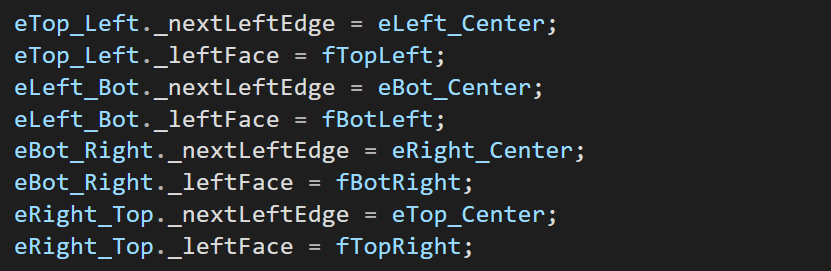
（1）以Center为起点的Edges：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **edge** | **originVertex** | **oppositeEdge** | **nextLeftEdge** | **leftFace** | **isReal** | **isConstrained** |
| eCenter\_Top | vCenter | eTop\_Center | eTop\_Left | fTopLeft | fTop.\_isReal | false |
| eTop\_Center | vTop | eCenter\_Top | eCenter\_Right | fRightTop | fTop.\_isReal | false |
| eCenter\_Left | vCenter | eLeft\_Center | eLeft\_Bot | fLeftBot | eRight\_Left.\_isReal | eRight\_Left.\_isConstrained |
| eLeft\_Center | vLeft | eCenter\_Left | eCenter\_Top | fTopLeft | eLeft\_Right.\_isReal | eLeft\_Right.\_isConstrained |
| eCenter\_Bot | vCenter | eBot\_Center | eBot\_Right | fBotRight | fBot.\_isReal | false |
| eBot\_Center | vBot | eCenter\_Bot | eCenter\_Left | fLeftBot | fBot.\_isReal | false |
| eCenter\_Right | vCenter | eRight\_Center | eRight\_Top | fTopRight | eLeft\_Right.\_isReal | eLeft\_Right.\_isConstrained |
| eRight\_Center | vRight | eCenter\_Right | eCenter\_Bot | fBotRight | eRight\_Left.\_isReal | eRight\_Left.\_isConstrained |

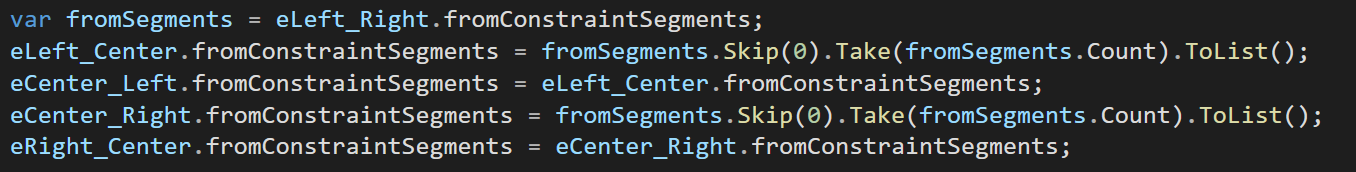
（2）建立Face和Edge关系：



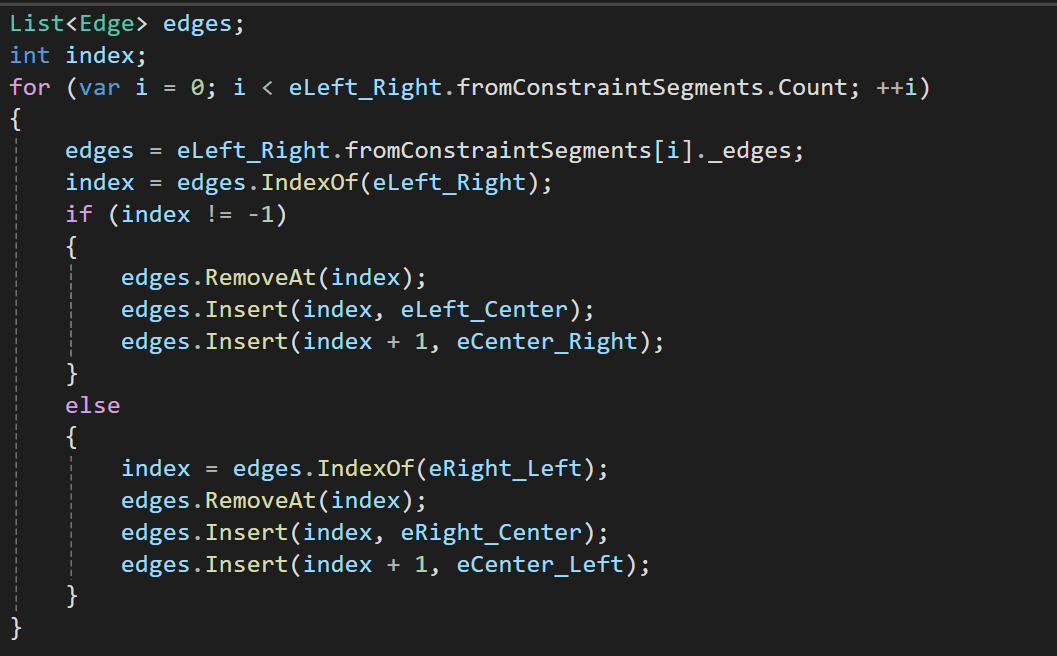
（3）更新原有Edges的关系：



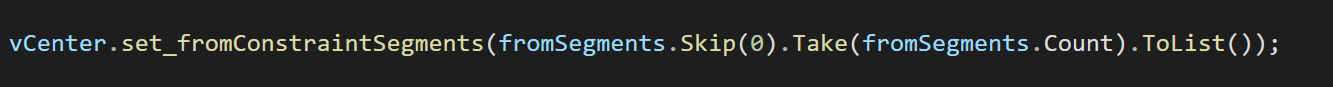
1. 如果eLeft\_Right是约束边，还需处理下面几点：
   1. 将eLeft\_Right的fromConstraintSegments传递到2个新的edge



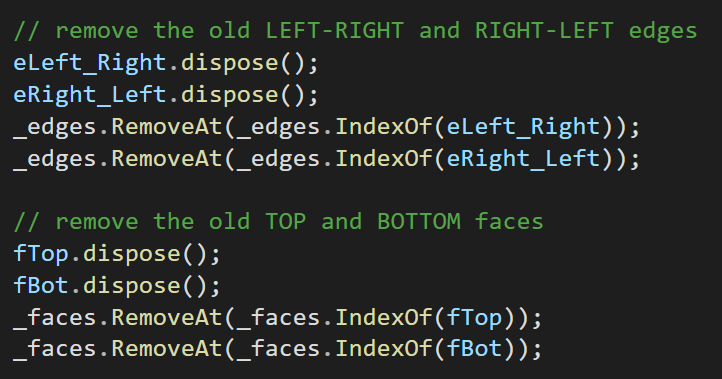
* 1. 更新eLeft\_Right的fromSegments中edge，即替换eLeft\_Right为2个新edge（注意：不需要处理eRight\_Left的fromSegments，因为这里存的是Segement的对象引用，eLeft\_Right和eRight\_Left肯定是公用同一个Segment，所以这里统一处理了）



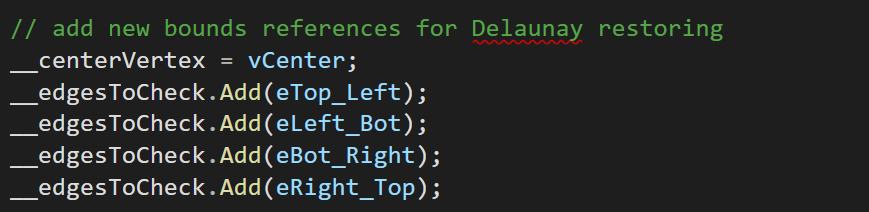
* 1. 将eLeft\_Right的fromConstraintSegments传递到新的vCenter顶点



1. 将老的eLeft\_Right和e Right\_Left边，以及老的fTop和fBot面删除掉



1. 因为2个Face被分为4个Face了，最小最大约束被破坏，所以要从老的4条边开始递归检查，如果不满足就走交换对角边操作（参考Mesh. restoreAsDelaunay函数）



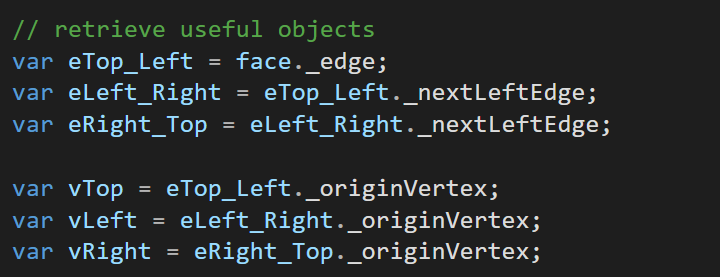
## Mesh.splitFace

vTop

vLeft

vRight

**vCenter**

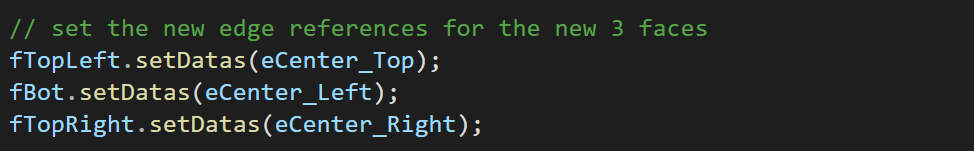


如图，原始1个face被新顶点vCenter分成3个face

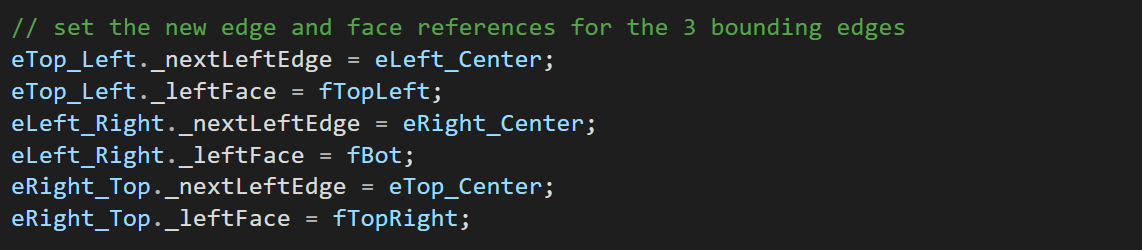
1. 新增6个Edges：eTop\_Center，eCenter\_Top，eLeft\_Center，eCenter\_Left，eRight\_Center，eCenter\_Right

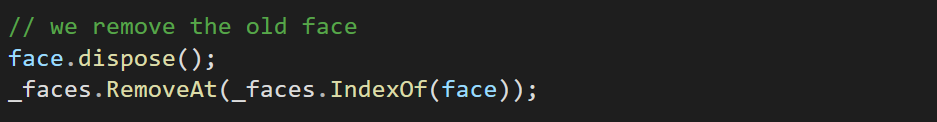
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **edge** | **originVertex** | **oppositeEdge** | **nextLeftEdge** | **leftFace** | **isReal** | **isConstrained** |
| eTop\_Center | vTop | eCenter\_Top | eCenter\_Right | fTopRight | true | false |
| eCenter\_Top | vCenter | eTop\_Center | eTop\_Left | fTopLeft | true | false |
| eLeft\_Center | vLeft | eCenter\_Left | eCenter\_Top | fTopLeft | true | false |
| eCenter\_Left | vCenter | eLeft\_Center | eLeft\_Right | fBot | true | false |
| eRight\_Center | vRight | eCenter\_Right | eCenter\_Left | fBot | true | false |
| eCenter\_Right | vCenter | eRight\_Center | eRight\_Top | fTopRight | true | false |

1. 新增3个face

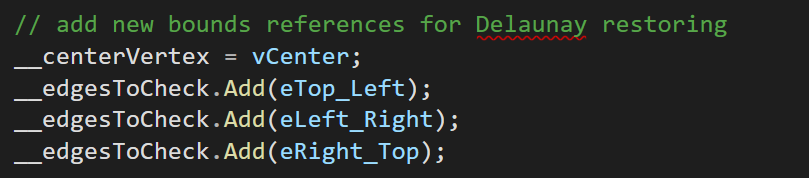


1. 更新老的face的3条edge（外边）对应关系，并移除老的face





1. 破坏最小最大约束，需要递归遍历edge检查相邻四边形是否需要交换对角edge



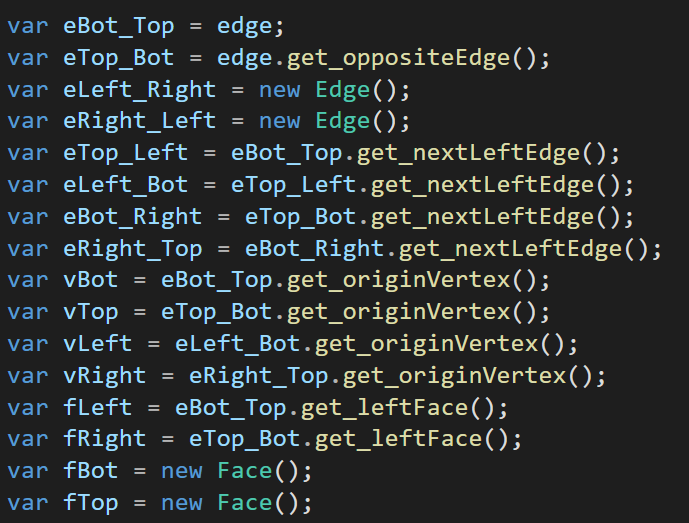
## Mesh.flipEdge

vTop

vLeft

vRight

**edge**



vBot

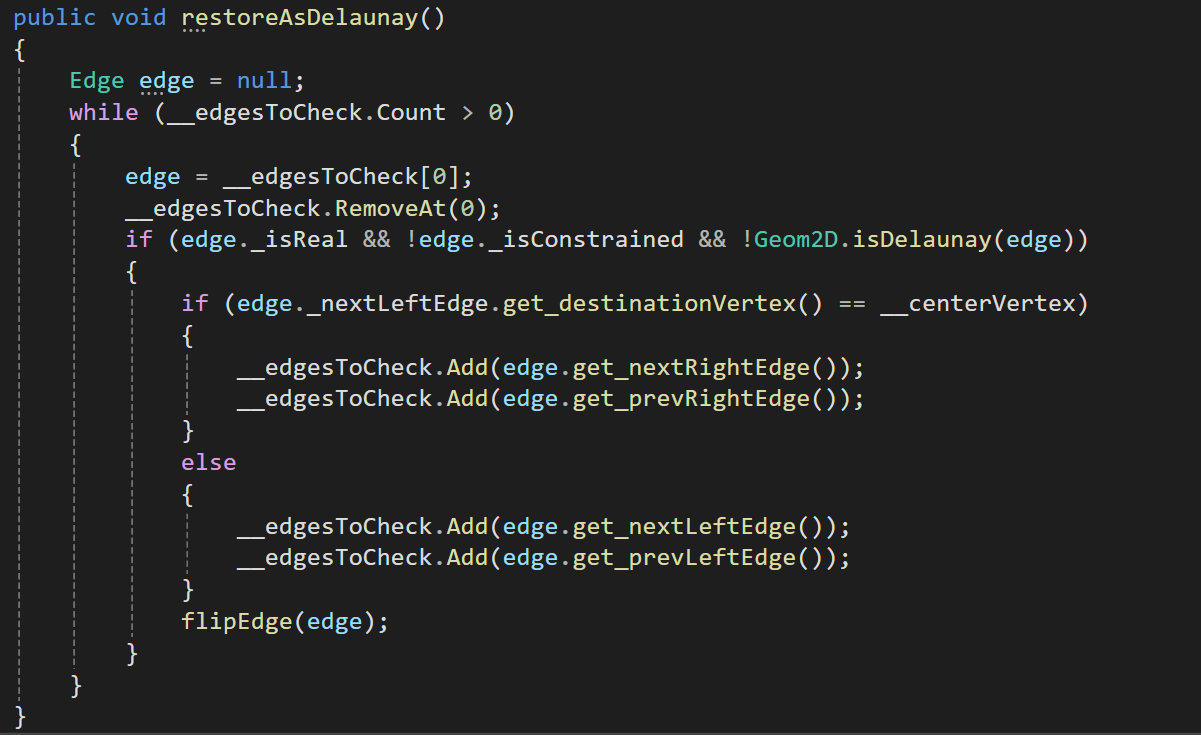
fLeft

fRight

目标：构造新eLeft\_Right边，将四边形分割成上下2个face

## Mesh. restoreAsDelaunay

前面的splitEdge和splitFace函数都会改变对应face的内部结构，所以需要从初始几个edge开始递归遍历，检查相邻的四边形是不是满足最小最大约束，不满足就要调用Mesh.flipEdge交换对角edge



**edge**

如图，假设当前遍历到edge（红色），对其相邻face构造的四边形（虚线包起来的）检查isDelaunay条件，如果不满足需要flipEdge，同时要把oppsiteFace对应的另外2条edge加入到待检查队列（这里同步检查destinationVeretx == centerVertex判断是否是第1层递归）

prevRightEdge

nextRightEdge

**edge**

prevLeftEdge

nextLeftEdge

递归第1层Face，添加nextRightEdge和prevRightEdge，因为加nextLeftEdge和prevLeftEdge，递归方向错了

递归第2层Face，添加nextLeftEdge和prevLeftEdge

## insertConstraintSegment完整流程

算法和核心就是依次迭代找segment穿过edge

（1）如果该edge是非约束边，且正常相交（非正常相交看下面的情况），则找到交点，并用交点

### 基于Bounds裁剪Segment

**0**

**1**

**2**

**3**

**4**

**5**

**6**

**7**

**8**

将空间分成9个区域，0号区域是RectMesh区域，其他区域是要被裁剪区域，如图输入线段(x1, y1) – (x2, y2)，经过了8,0,6三个区域，被裁剪后为 (newX1, newY1) – (newX2, newY2)，这里默认如果有裁剪，线段方向是从下到上，如果没有裁剪，则使用默认输入方向，所以最后的segment方向不是确定的顺时针或逆时针顺序

(x1, y1)

(x2, y2)

(newX1, newY1)

(newX2, newY2)

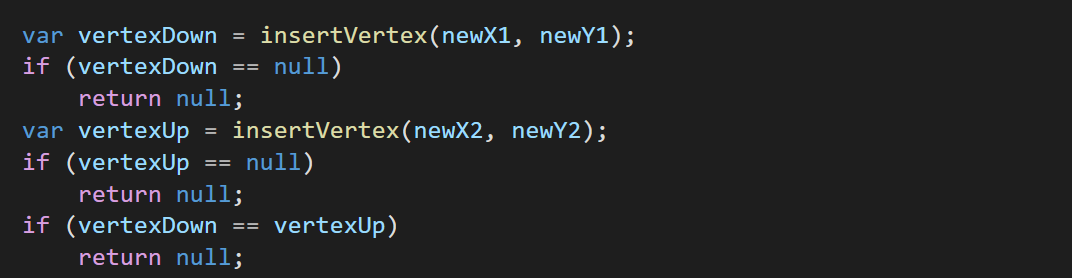
xmin

xmax

ymin

ymax

### 插入(newX1, newY1) 和 (newX2, newY2)两个顶点



### 以vertexDown作为起点开始迭代，找到所有和(newX1, newY1) - (newX2, newY2)线段相交的Edges

vertexDown

vertexUp

#### currObject当前是Vertex对象

vertexDown

vertexUp

如图，依次遍历从Vertex发出去的edges（FromVertexToOutgoingEdges迭代器），当前edge记为currEdge，vertexDown 到vertextUp的edge记为tempEdgeDownUp：

1. 检查currEdge.get\_destinationVertex() == vertexUp条件，如果满足，则直接将该edge标记为约束edge，并添加到segment的edge列表中，**迭代结束**
2. 如果currEdge.get\_destinationVertex()到tempEdgeDownUp边的距离很近，则将该edge加入到segement的edges列表，并更新vertexDown和currObject为currEdge.get\_destinationVertex()，**迭代继续**
3. 以上2种情况都不满足，则继续重新遍历从Vertex发出去的edges，检查currEdge.get\_nextLeftEdge()是否和tempEdgeDownUp是否相交，如果相交则更新currObject对象，进入下一轮迭代
   1. 如果currEdge是约束边，则需要将该约束边1分为2

vertexDown

vertexUp

currEdge（红色实线）是约束边，则将该edge分成2个约束边，并把vertexDown更新为新的交点，currObject更新为vertexDown继续迭代

此edge加到leftBoundingEdges（非约束边情况）

此edge加到rightBoundingEdges（非约束边情况）

* 1. 如果currEdge不是约束边，则将该约束边添加到intersectedEdges列表（这些edge是要被删除的），并把Prev和Next边加到rightBoundingEdges和leftBoundingEdges列表中，后续删除intersectedEdges需要重新三角化需要用到，currObject更新为currEdge的oppsiteEdge，开始下一次的迭代

#### currObject当前是Edge对象

vertexDown

vertexUp

currEdge

edgeLeft

1. 如果edgeLeft.get\_destinationVertex() == vertexUp，则把edgeLeft和edgeLeft.get\_nextLeftEdge()分别加到rightBoundingEdges和leftBoundingEdges，并将boundingEdges构成的封闭多边形三角化，**结束迭代**
2. edgeLeft.get\_destinationVertex()到tempEdgeDownUp距离非常近，先把当前迭代结果处理掉（跟步骤1一样），然后重置状态，并将vertexDownh和currObject设置为edgeLeft.get\_destinationVertex()，**重新开始迭代过程**
3. 如果上面2种情况都不满足，则将edgeLeft和tempEdgeDownUp的交点找到，并用该交点（记为currVertex）将edgeLeft分割
   1. 如果edgeLeft是约束边，约束边被1分为2，遍历currVertex发出去的edges，将其正确的填充到rightBoundingEdges和leftBoundingEdges列表中（就是找到如图黄色封闭多边形），三角化，vertexDown和currObject更新为currVertex，并清空状态，重新迭代

vertexDown

vertexUp

currEdge

edgeLeft

* 1. 如果edgeLeft不是约束边，则处理和上面Vertex情况3-b一样处理，并继续迭代

vertexDown

vertexUp

currEdge

edgeLeft

vertexDown

vertexUp

currEdge

edgeLeft

1. 上面情况3是和edgeLeft相交情况，但也有可能是和edgeLeft.get\_nextLeftEdge()相交情况，处理逻辑和情况3基本一样

vertexDown

vertexUp

currEdge

edgeLeft