



# Lecture 11: 网格和流形

SSE315: 计算机图形学  
Computer Graphics

---

陈壮彬

软件工程学院

chenzhb36@mail.sysu.edu.cn

# Course roadmap

## 光栅化 Rasterization

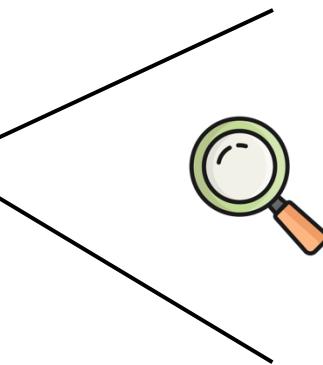
计算机图形学介绍  
基于采样的光栅化  
空间变换  
纹理映射、深度和透明度

## 几何 Geometry

几何介绍  
曲线与曲面

## 材质与光线 Materials and Lighting

## 动画 Animation



几何介绍  
曲线与曲面



# 几何 (Geometry) 回顾

口现实世界几何的种类繁多且复杂

口需要复杂、精细的表示方法

口有两大类

- 隐式的 (**Implicit**) – 测试某个点是否在几何上
- 显式的 (**Explicit**) – 直接给出点的位置

口每一类都有很多具体的表示方法

**Geometry**



# Today's topics

□ 曲面 (surface) 的概念

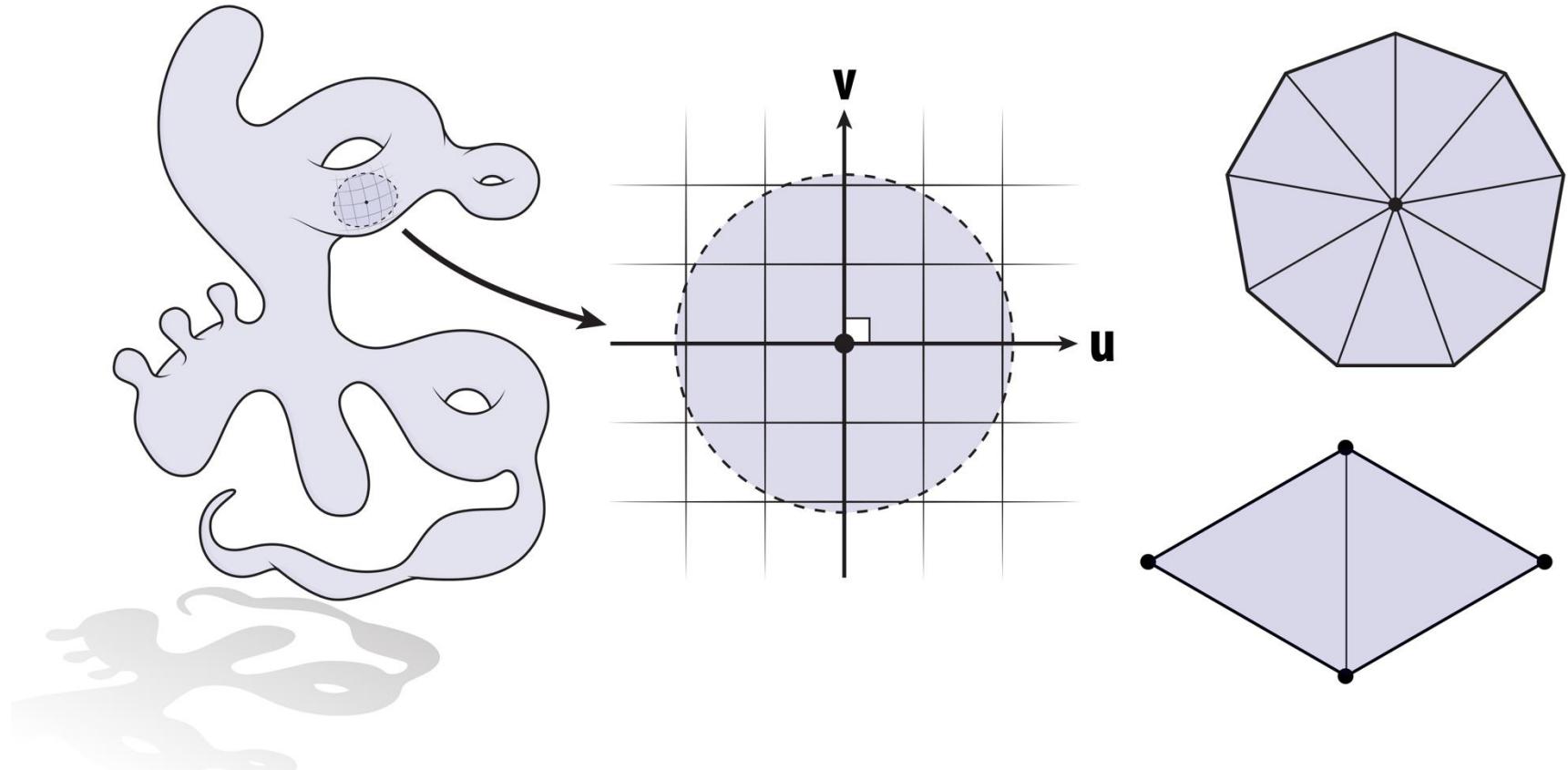
- 曲面是什么?
- 有什么特性?

□ 多边形网格 (polygon meshes) 的具体细节

- 如何编码曲面?
- 如何处理几何物体?

# 流形假设 Manifold assumption

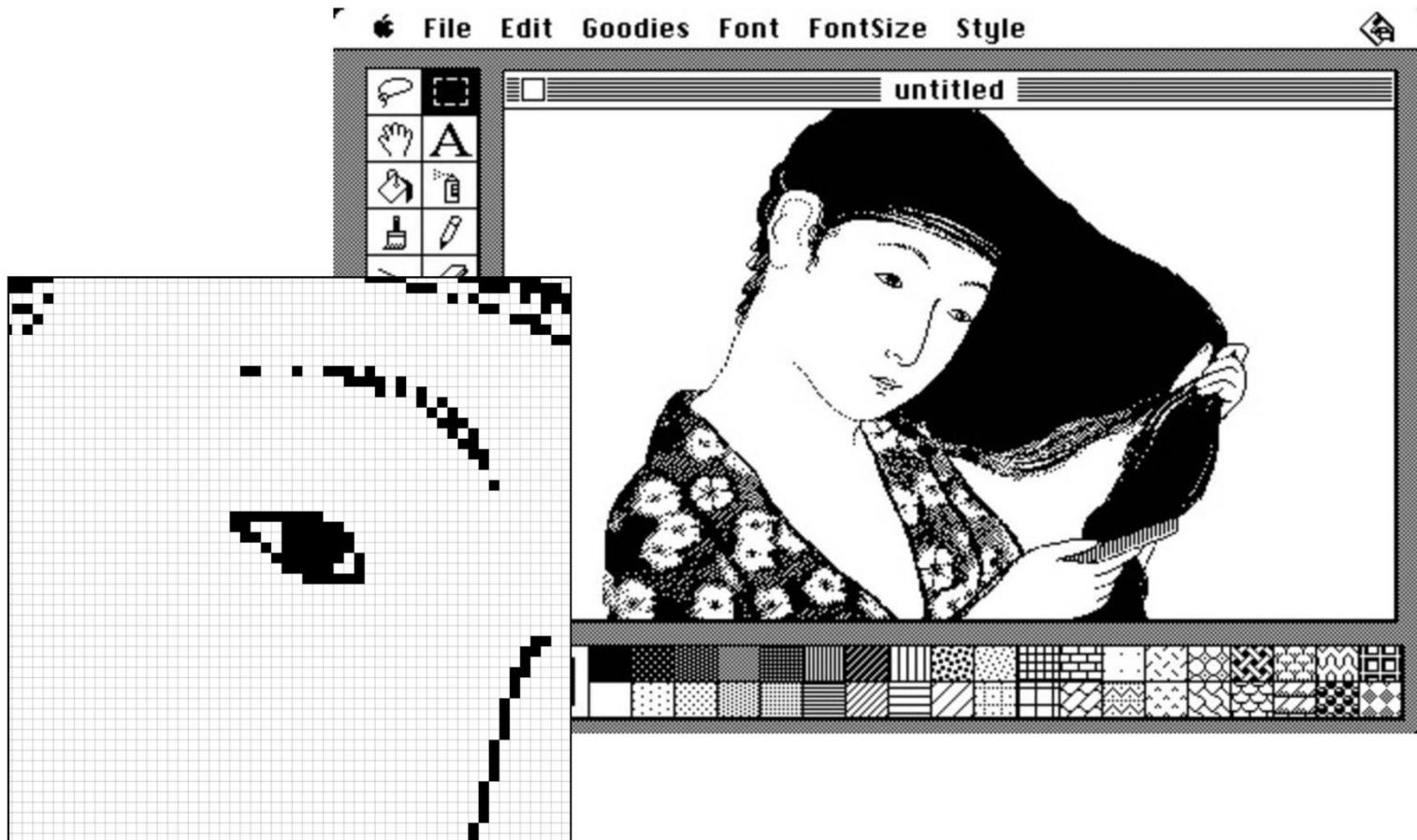
- 几何一个非常重要的概念：**流形 manifold**
- 尽管很有用，但一开始可能很难理解它的动机是什么
- 所以，我们先回顾一些更熟悉的例子 – 2D 图像



# 位图图像 (Bitmap Images)

为了编码图像，我们使用一个规则的像素网格模型

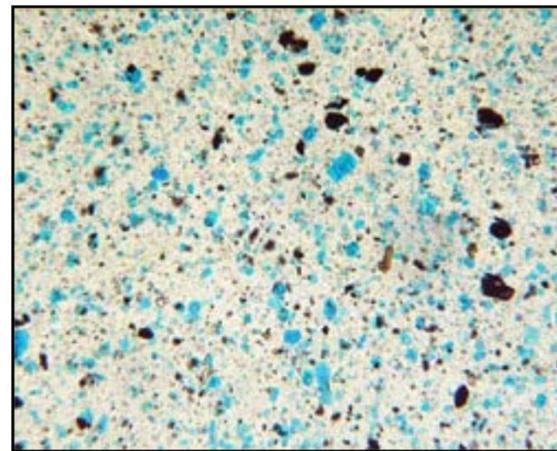
- 每个像素是方形的，且有一定的属性 (颜色，深度...)



但图像从根本上并不是由小方块组成的



Goyō Hashiguchi, *Kamisuki* (ca 1920)

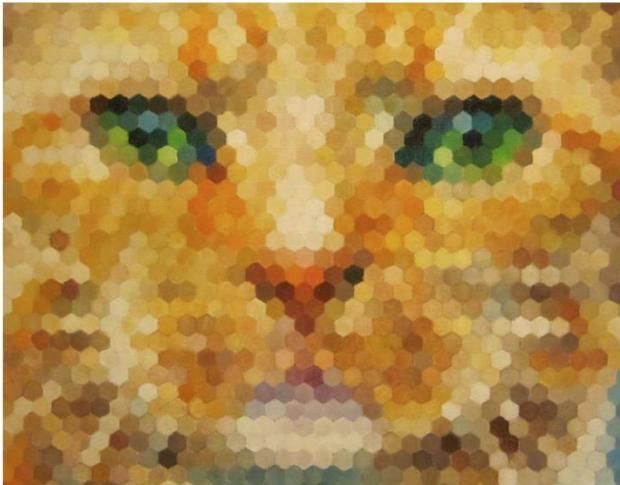


photomicrograph of paint

# 那我们为什么还使用正方形网格？

□ 有其他非常多的方式铺满平面

- 正六边形 (也有人做出这样的计算机，但未推广)
- 不规则多边形
- 正六边形+正方形+三角形
- ...



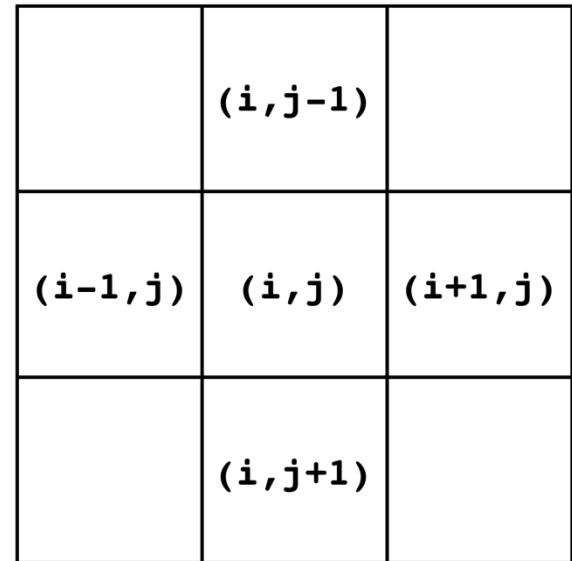
# 规则网格让事情变得更简单

□一个原因：简单及高效 (simplicity / efficiency)

- 比如总是有四个邻居
- 容易索引 (index) 和过滤 (filter)
- 存储简单：数字列表

□另一个原因：一般性 (generality)

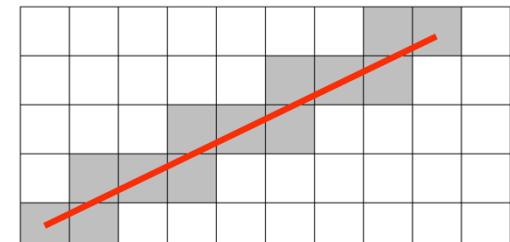
- 能够编码几乎所有图形



□规则网格是否总是图像最好的编码方式呢？

- No, 比如受各向异性 (anisotropy) 影响，不能捕捉边缘...
- 但总的来说还是一个很好的编码选择

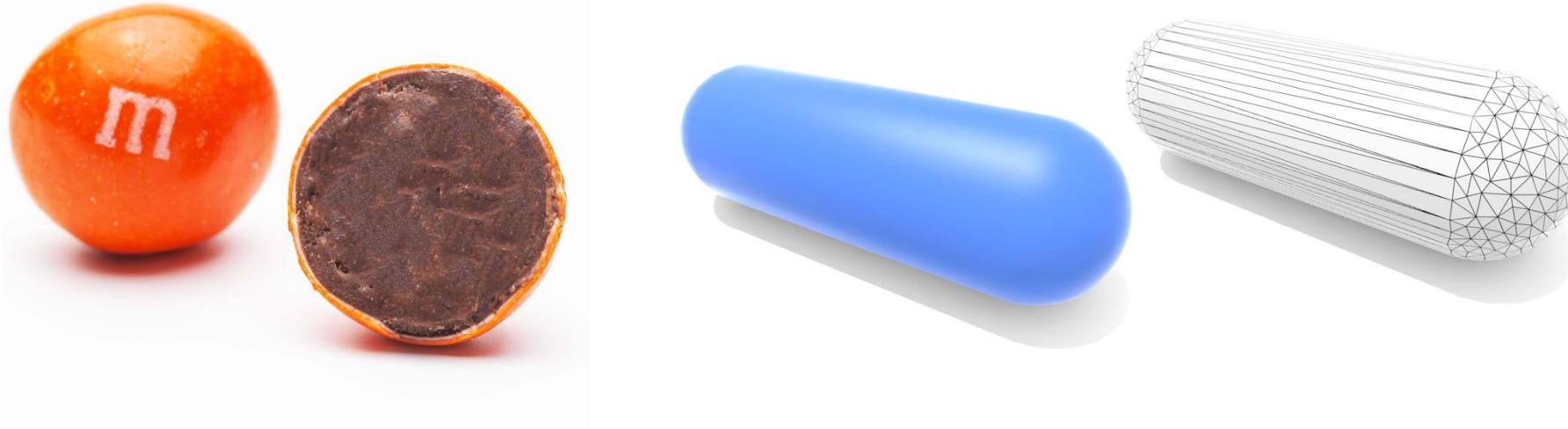
□几何中有类似的情况，需要权衡



我们如何编码曲面  
(surfaces)?

# 平滑的表面

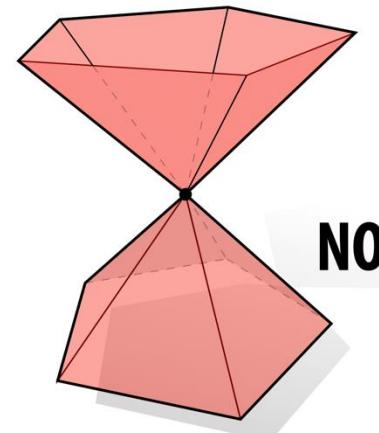
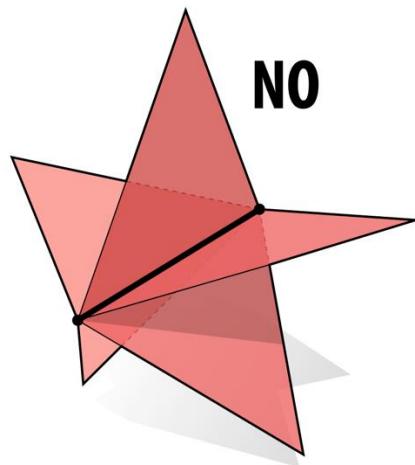
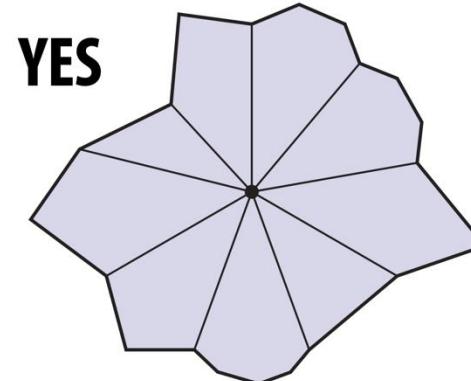
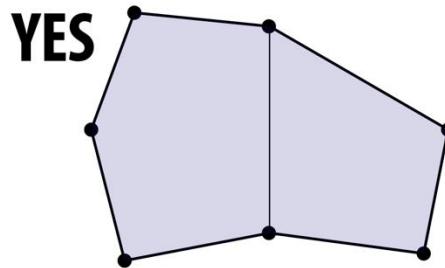
- 直观地说，表面是物体的边界 (boundary) 或外壳 (shell)
- 想想糖果的壳，而不是内部的巧克力
- 表面的另一个重要概念：流形 (manifold)
  - 流形几何具有良好的特性
  - 怎么判断多边形网格是否为流形？



# 流形多边形网格 (manifold polygon mesh)

判断多边形曲面是否为流形，只需检查两个简单的条件：

1. 每条“内部”边只包含在两个多边形中
2. 每个“内部”点周围包含一圈循环的多边形



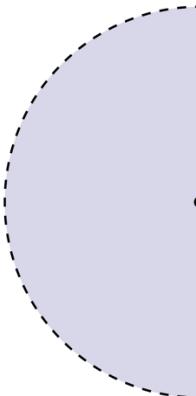
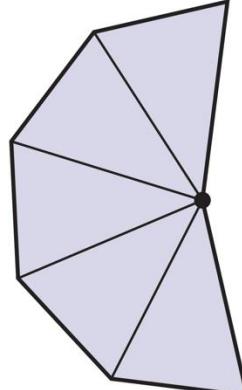
# 形状的边界 (boundary) 呢？

□ 边界是曲面的“终点”

□ 例如，裤子上的腰部和脚踝

□ 在局部，边界看起来像半个磁盘

□ 整体上看，每个边界都形成一个循环



YES

□ 多边形网格

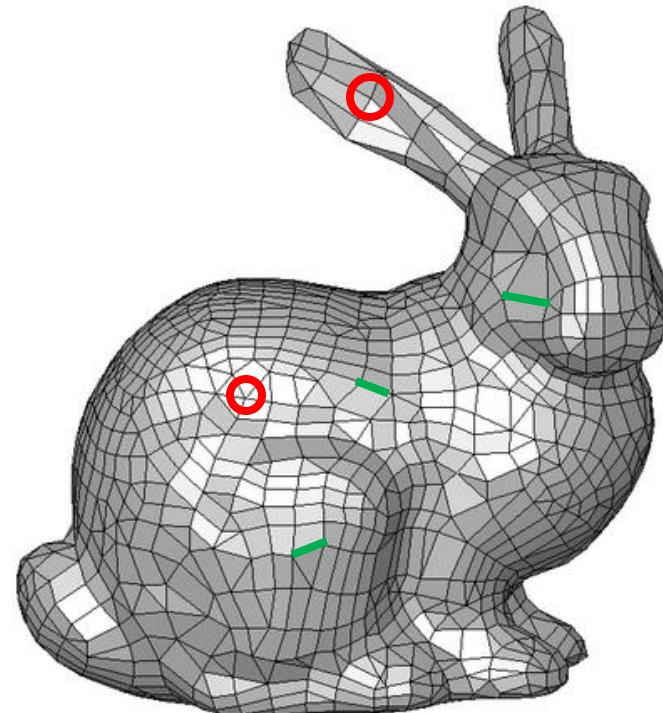
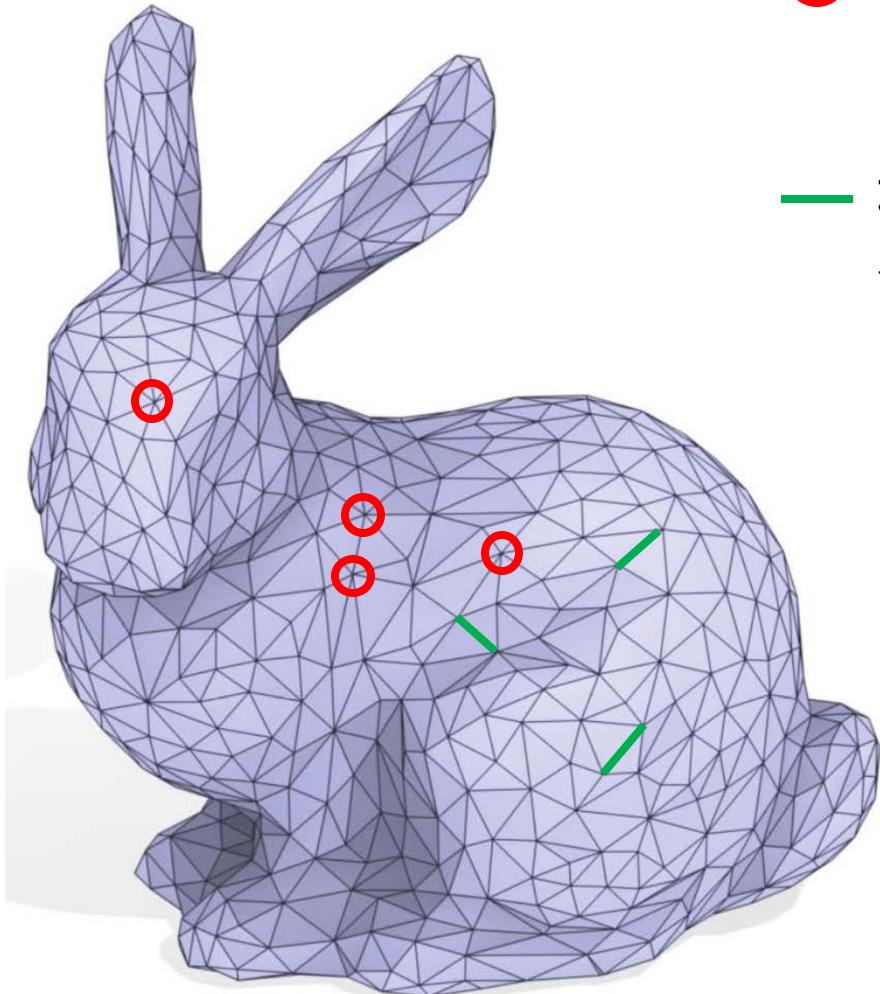
- 每个“边界”边仅包含在一个多边形中
- 每个“边界”点周围包含成一组(而不是一圈)的多边形



# 流形多边形网格

○ 标识“内部”点，被与其相连的多边形环绕一圈

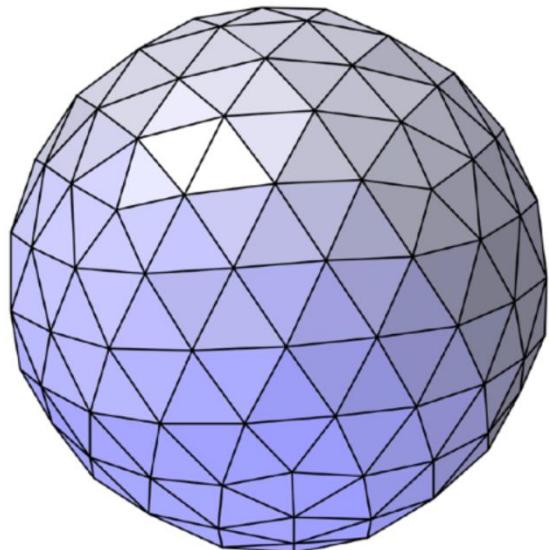
— 标识“内部”边，仅包含在两个多边形中



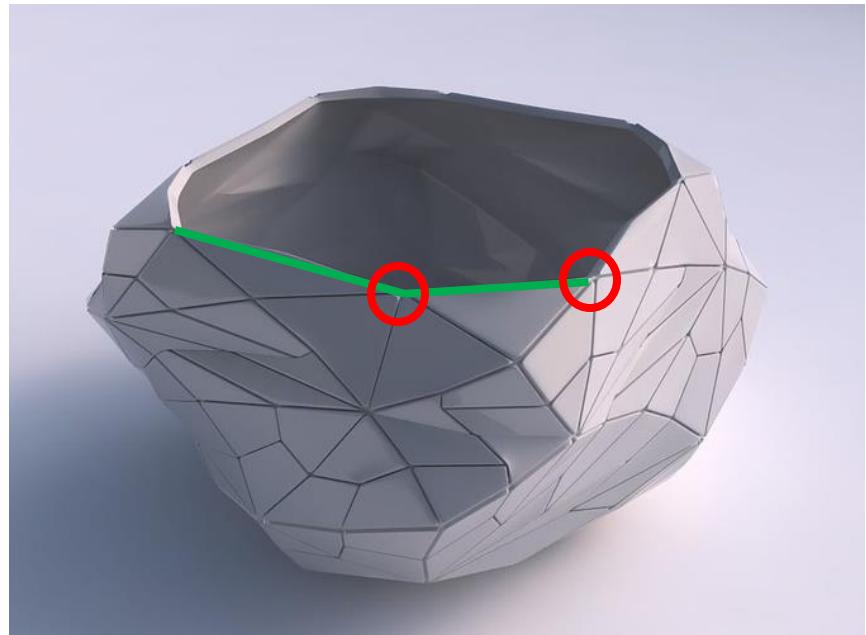
其他多边形网格也一样

# 流形多边形网格

并不是每个网格都有边界



无边界



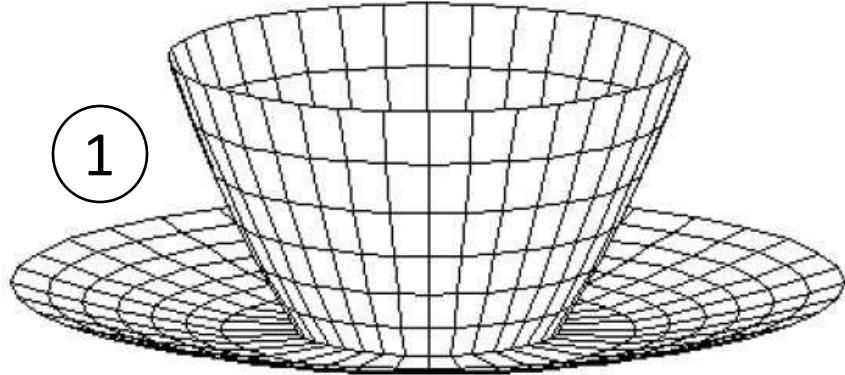
有边界

○ 标识“边界”点，被与其相连的  
多边形环绕，但未构成一圈

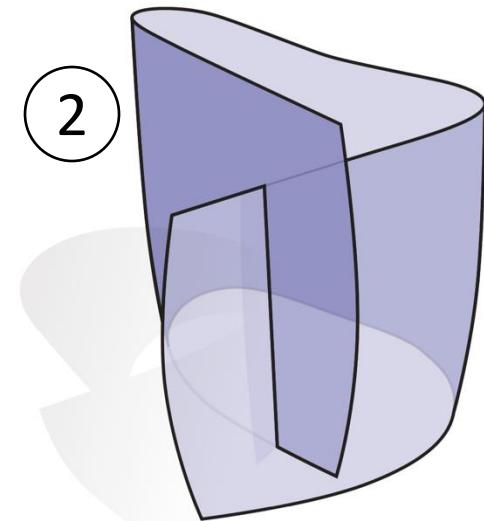
— 标识“边界”边，仅包含在  
一个多边形中

# 流形与非流形 (manifold vs. nonmanifold)

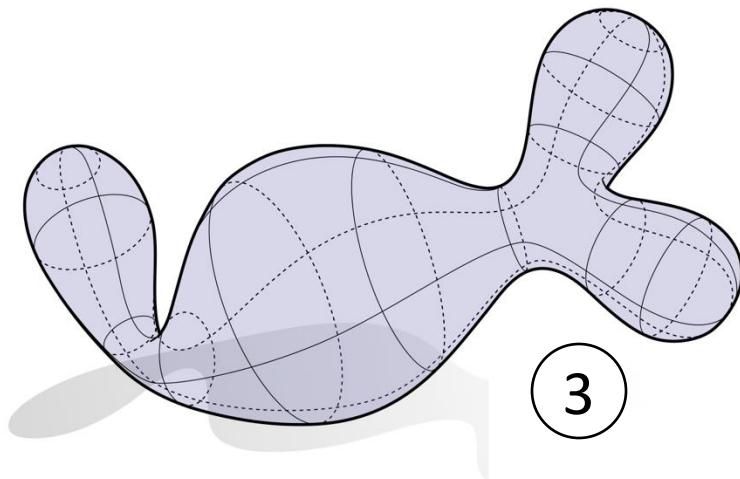
□下列哪些几何图形是流形的



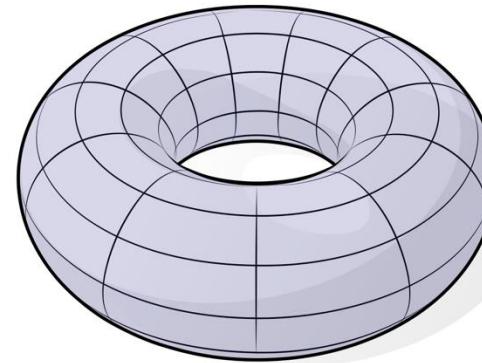
1



2



3



4

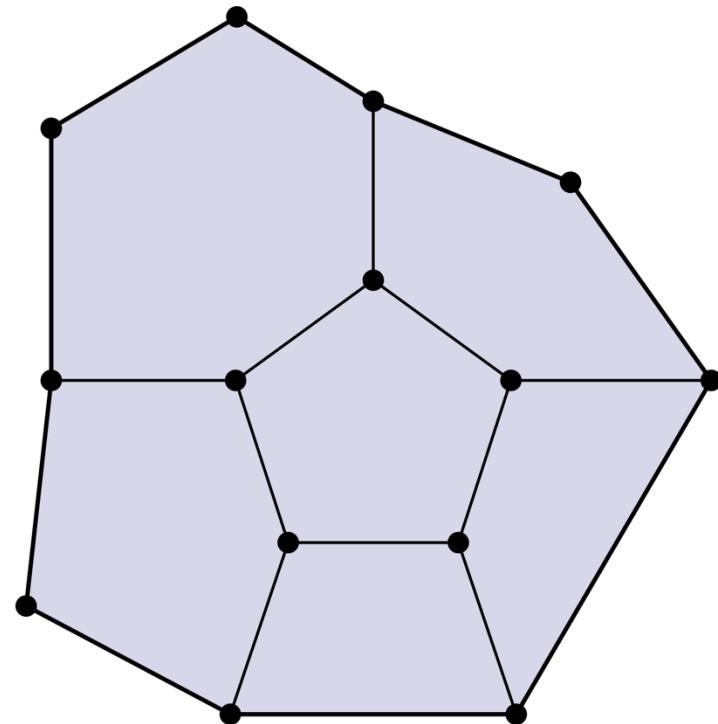
# 流形假设有什幺用？

# 让事情变得更简单！

□与图片的动机相同：

- 对几何的结构做一些假设，以保持数据结构/算法简单高效
- 同时，在许多常见的情况下，不会限制我们对几何的处理

	$(i, j-1)$	
$(i-1, j)$	$(i, j)$	$(i+1, j)$
	$(i, j+1)$	



我们如何编码这些数据？

# 热身：存储数字

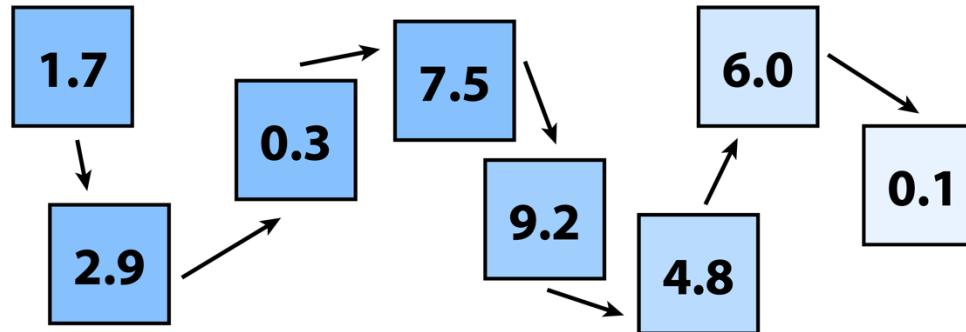
□ Q：我们可以使用哪些数据结构来存储数字列表？

□ 一个方案：使用数组（恒定查找时间、一致的访问）



□ 但是难以删除和插入数据

□ 其他方案：使用链表（线性查找时间、不一致的访问）



□ Q：为什么要处理链接列表？

□ A：我们可以很容易地在任何地方插入数据...

# 存储多边形网格

## □最基本的想法

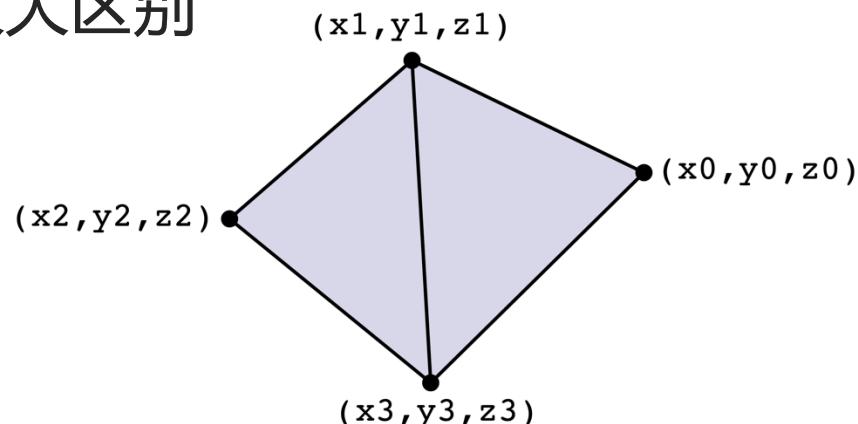
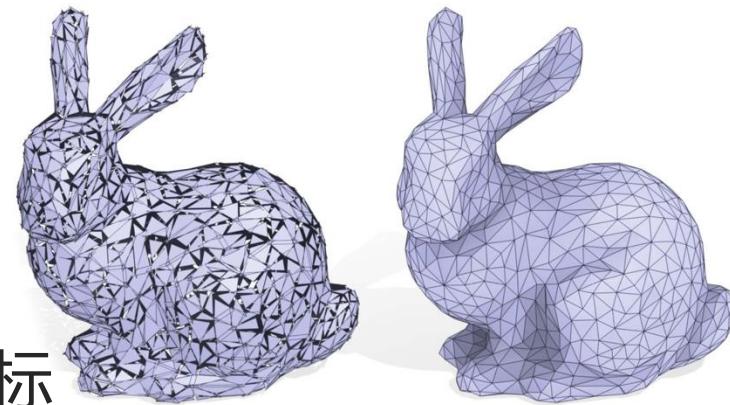
- 对于每个三角形，只需存储三个坐标
- 没有其他有关连接的信息
- 与点云 (point cloud) 没有太大区别

## □Pros

- 超级简单

## □Cons

- 存储了冗余的数据
- 除了在屏幕上简单地绘制网格外，做不了其他什么事
- 需要稀疏的数据结构 (spatial data structures) 来查找邻居



$x_0, y_0, z_0$	$x_1, y_1, z_1$	$x_3, y_3, z_3$
$x_1, y_1, z_1$	$x_2, y_2, z_2$	$x_3, y_3, z_3$

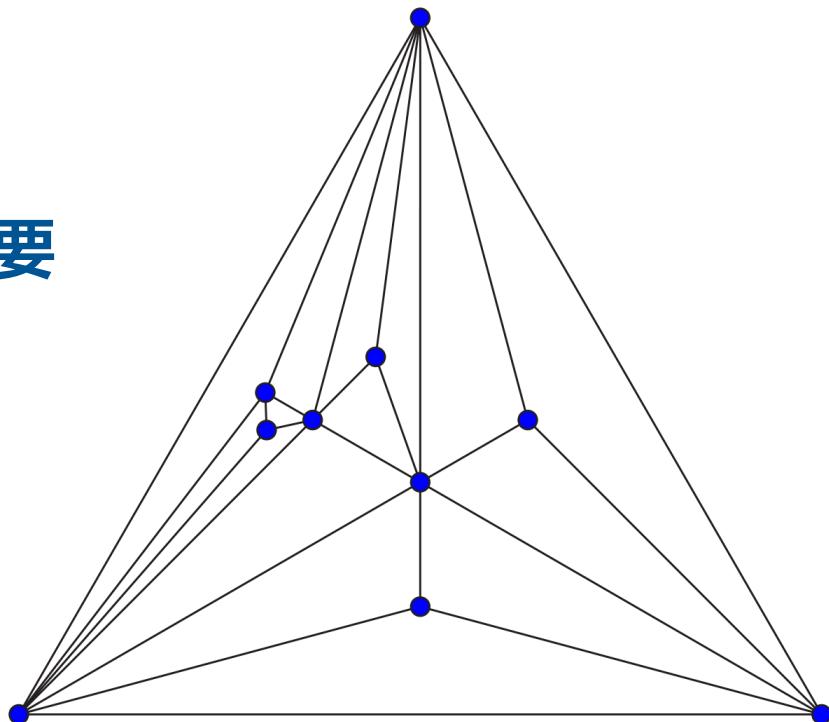
# 几何的连接性/邻居信息

□ 连接信息是指节点、边、面之间如何连接

- 点和点的连接
- 点和边的连接
- 边和面的连接
- ...

**□ 连接信息在在几何处理中很重要**

- 网格简化
- 表面细化
- 表面规则化
- 纹理映射
- 导航
- 拓扑操作
- ...

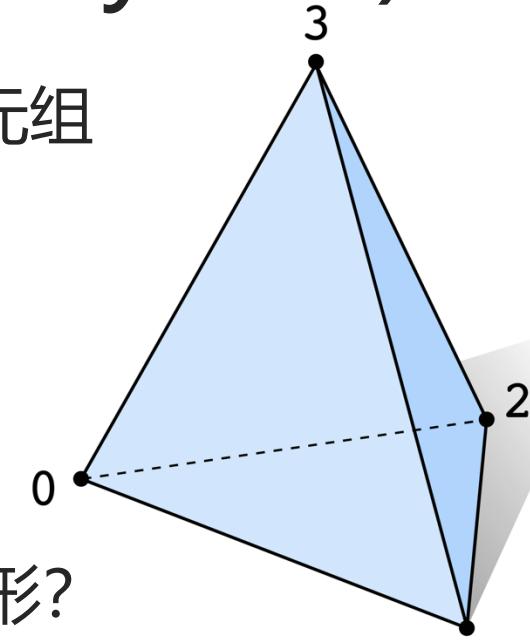


# 邻接表 Adjacency list (Array-like)

□ 存储坐标的三元组  $(x, y, z)$  以及索引的三元组

□ 比如四面体：

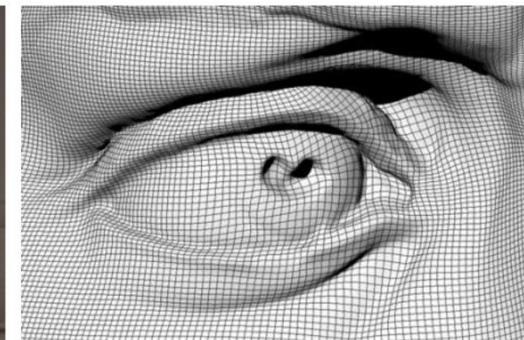
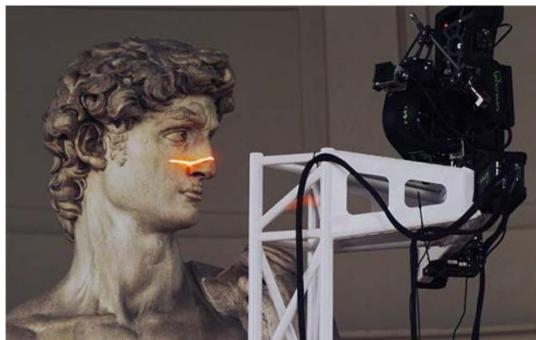
	VERTICES			POLYGONS		
	x	y	z	i	j	k
0:	-1	-1	-1	0	2	1
1:	1	-1	1	0	3	2
2:	1	1	-1	3	0	1
3:	-1	1	1	3	1	2



□ Q：我们如何找到所有接触顶点 2 的多边形？

□ 现在考虑一个更复杂的网格：

$\sim 1 \text{ billion polygons}$ <sup>1</sup>



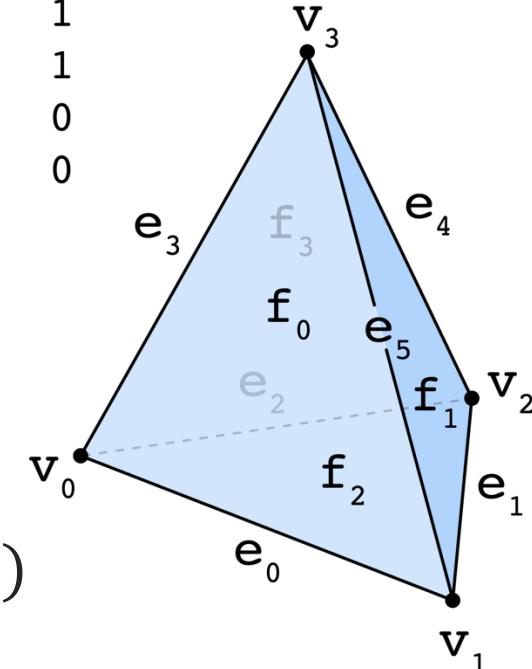
□ 查找相邻多边形非常昂贵！

# 关联矩阵 Incidence matrices

- 如果我们知道邻居是谁，为什么不存储一个邻居列表？
- 可以通过**关联矩阵**对所有邻居信息进行编码
- 例如，四面体：**VERTEX↔EDGE**      **EDGE↔FACE**

	v0	v1	v2	v3		e0	e1	e2	e3	e4	e5
e0	1	1	0	0	f0	1	0	0	1	0	1
e1	0	1	1	0	f1	0	1	0	0	1	1
e2	1	0	1	0	f2	1	1	1	0	0	0
e3	1	0	0	1	f3	0	0	1	1	1	0
e4	0	0	1	1							
e5	0	1	0	1							

- 1 意味着**接触**；0 表示**不接触**
- 使用**稀疏矩阵**而不是存储大量的 0
- 存储成本仍然很大，但现在查找邻居是  $O(1)$
- 很难更改连接，因为我们使用了固定索引
- 除了简单，另外的好处是**网格不必是流形的**



# 稀疏矩阵数据结构

□ 我们如何实际存储“稀疏矩阵”？

□ 很多可能的数据结构：

- **从 (row, column) 到 value 的关联数组**

- 易于查找/设置条目，速度快（例如哈希表）
- 难以进行矩阵运算（例如乘法）

- **链表数组（每行一个）**

- 概念简单
- 存取时间慢，数据存取不连贯

- **压缩列格式 – 打包列表中的条目**

- 难以添加/修改条目
- 快速进行实际矩阵运算

□ 在实践中：通常使用“简单”的数据结构存储数据，转换为其他格式进行计算

0	1	2
0	4	2
1	0	0
2	0	7

(row,col) val

(0,0)	->	4
(0,1)	->	2
(1,2)	->	3
(2,1)	->	7

(col,val) (col,val)

row 0:	(0,4)	→	(1,2)
1:	(2,3)		
2:	(1,7)		

values 4,2,7,3

row indices 0,0,2,1

cumulative  
# entries  
by column

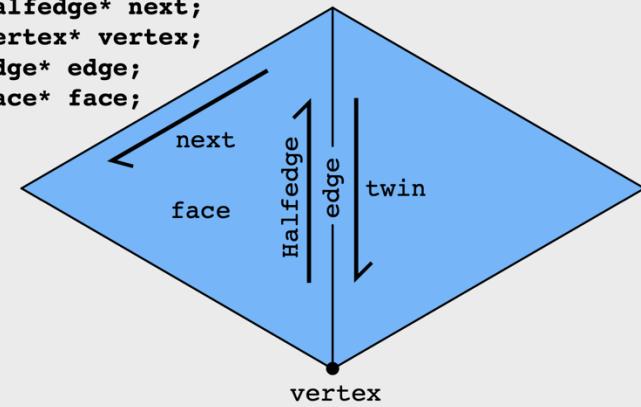
# 半边 Halfedge 数据结构 (类似链表)

□ 基本理念：存储一些邻居的信息

□ 不需要详尽的清单；只需要几个关键指针 pointer (**就能让我们存储/计算所有网格信息**)

□ 关键思想：两个半边缘充当网格元素之间的“粘合剂”

```
struct Halfedge
{
    Halfedge* twin;
    Halfedge* next;
    Vertex* vertex;
    Edge* edge;
    Face* face;
};
```



```
halfedge
edge
struct Edge
{
    Halfedge* halfedge;
};
```

```
struct Face
{
    Halfedge* halfedge;
};
```

```
vertex
halfedge
struct Vertex
{
    Halfedge* halfedge;
};
```

**每个 halfedge 存储很多信息**

□ 每个顶点、边和面仅指向它的其中一个半边

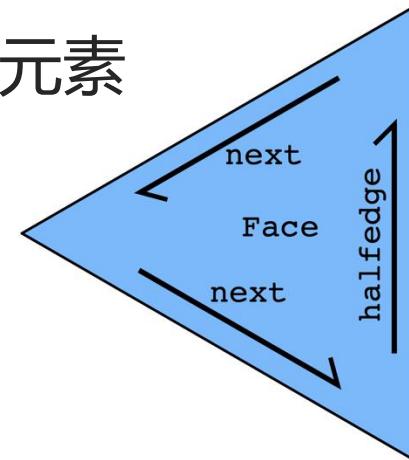
# 半边简化了网格遍历 mesh traversal

□ 使用 twin 和 next 指针在网格中移动

□ 使用 vertex、edge 和 face 指针来抓取元素

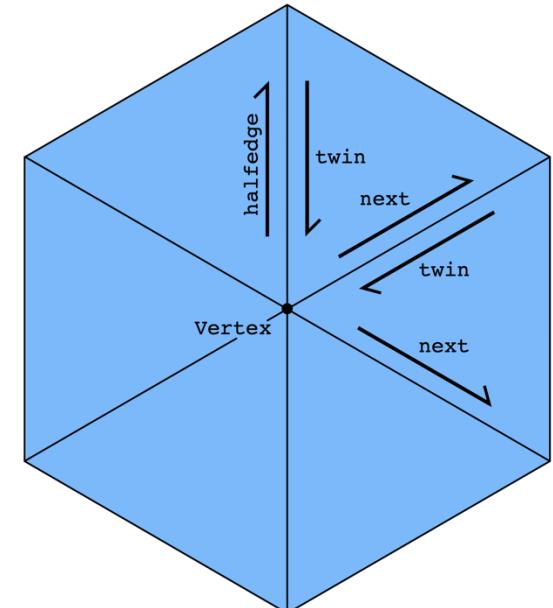
□ **示例1：访问一个面的所有顶点**

```
Halfedge* h = f->halfedge;
do {
    h = h->next;
    // do something w/ h->vertex
}
while( h != f->halfedge );
```



□ **示例2：访问一个顶点接触的所有邻居**

```
Halfedge* h = v->halfedge;
do {
    h = h->twin->next;
}
while( h != v->halfedge );
```



□ 注意：只有当网格是流形时才有意义！

# 半边连通性总是流形的

□ 考虑一个简化的半边缘数据结构

□ 仅需要以下条件，就能得到一个流形多边形网格

指向自己的  
的指针

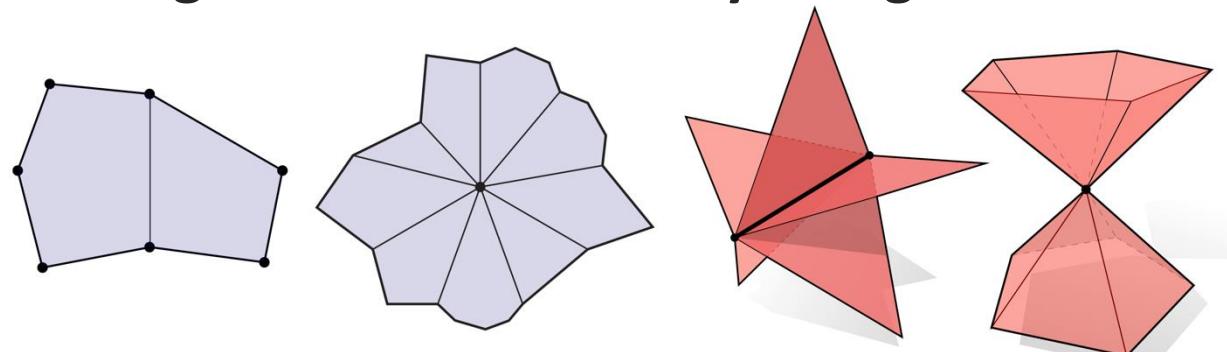
```
struct Halfedge {  
    Halfedge *next, *twin;  
};
```

```
twin->twin == this  
twin != this  
every he is someone's "next"
```

□ Keep following next, and you'll get faces

□ Keep following twin and you'll get edges

□ Keep following next->twin and you'll get vertices



□ 因此，从半边的构造来看，不能编码红色的非流形几何图形

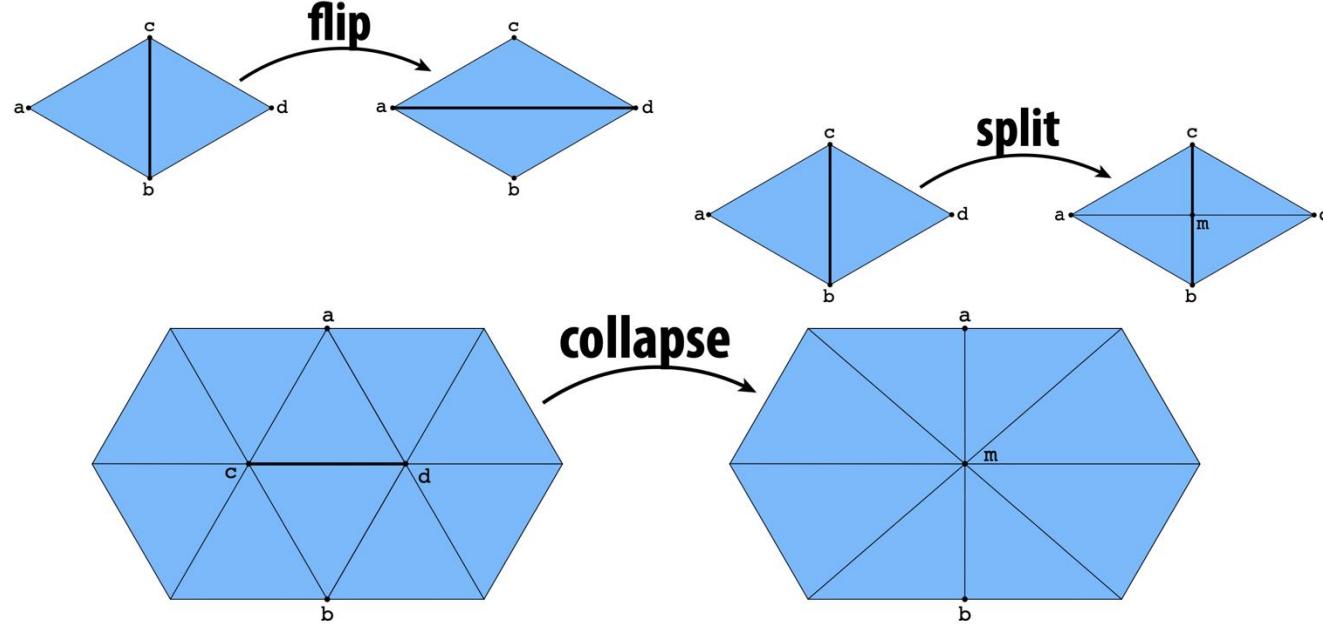
半边数据结构有什么  
优势/作用？

# 半边网格易于编辑

口链表的关键功能：便于插入/删除元素

口半边网格（本身也类似链表）也类似

口例如，对于三角形网格，有几个原子操作：

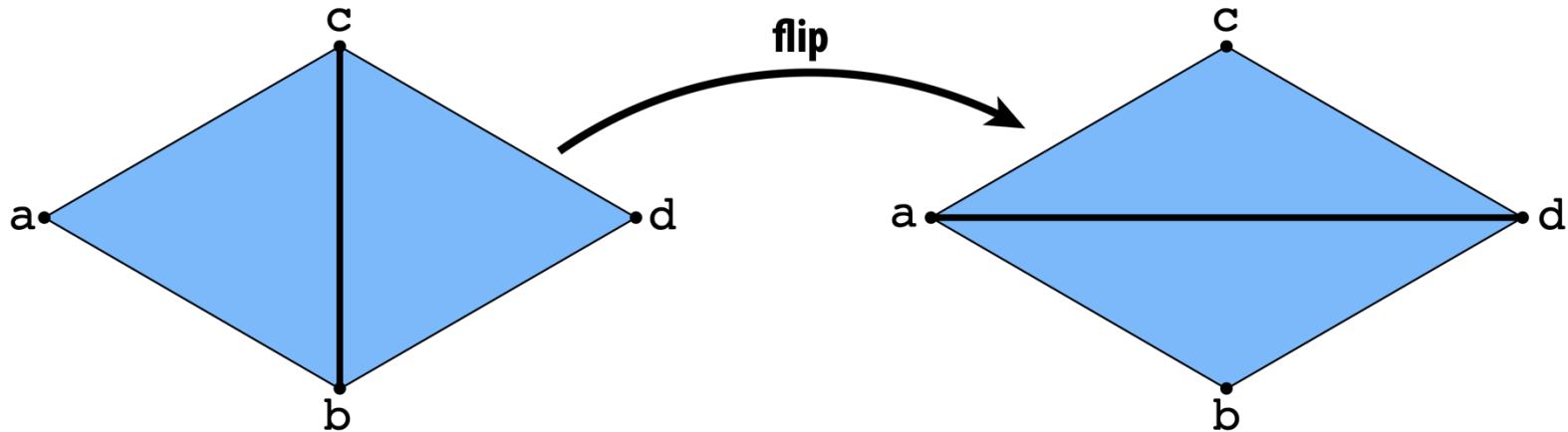


口怎么做？分配/删除元素；重新分配指针

口必须小心保存流形性 manifoldness

# 边翻转 edge flip (三角形)

□ 三角形  $(a, b, c)$  和  $(b, d, c)$  变成  $(a, d, c)$  和  $(a, b, d)$



□ 指针重新分配的长列表 (edge->halfedge = ...)

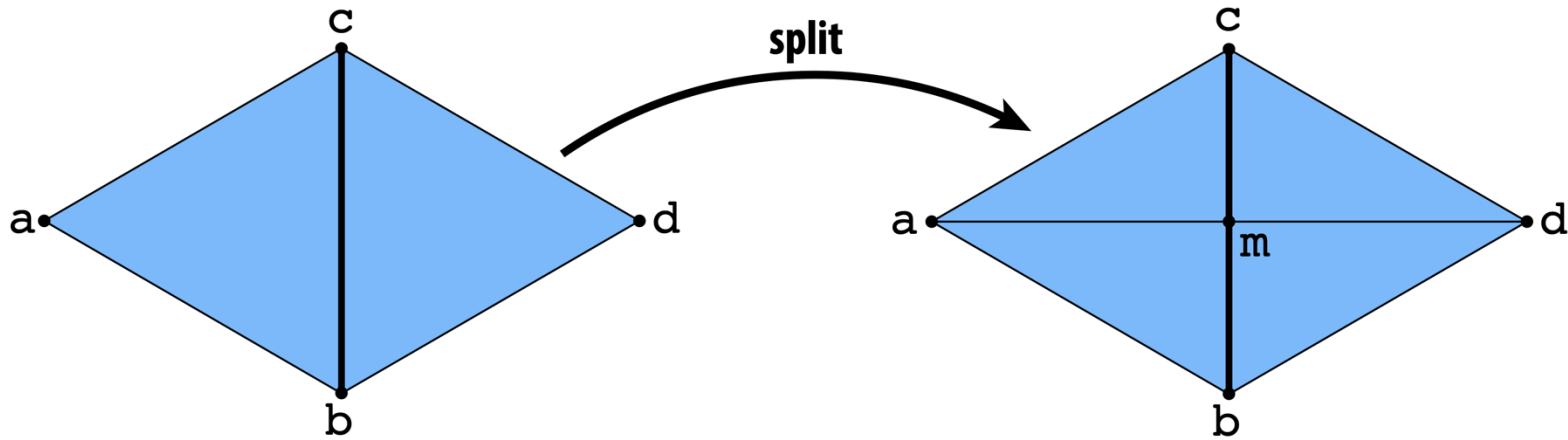
□ 没有创建/销毁任何元素

□ Q：如果我们翻转两次会发生什么？

□ A：会回到原来的结构

# 边拆分 edge split (三角形)

□ 连接边 (c,b) 的中点 m 得到四个三角形



□ 这一次，必须添加新元素

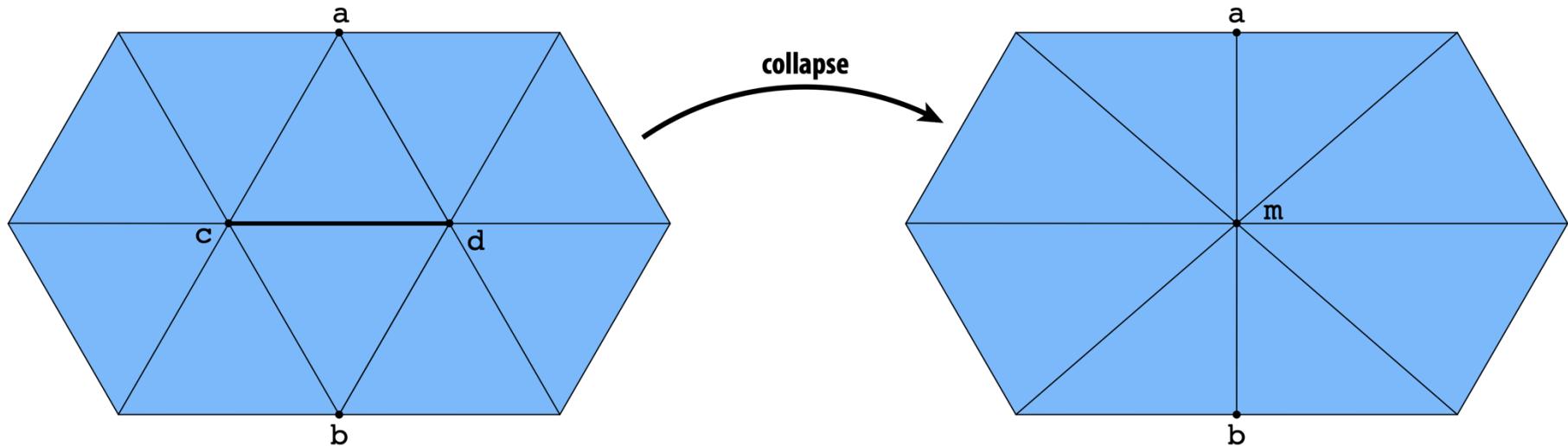
□ 大量的指针重新分配

□ Q：我们能“逆转”这种操作吗？

□ A：可以，但需要标记新增的元素

# 边折叠 edge collapse (三角形)

□ 将边  $(b, c)$  替换为单个顶点  $m$  :



□ 必须删除元素

□ 同样有很多指针重分配!

□ Q: 我们如何在邻接列表中完成上述操作?

□ 是否有其他方式进行上述操作? 比如不同的数据结构?

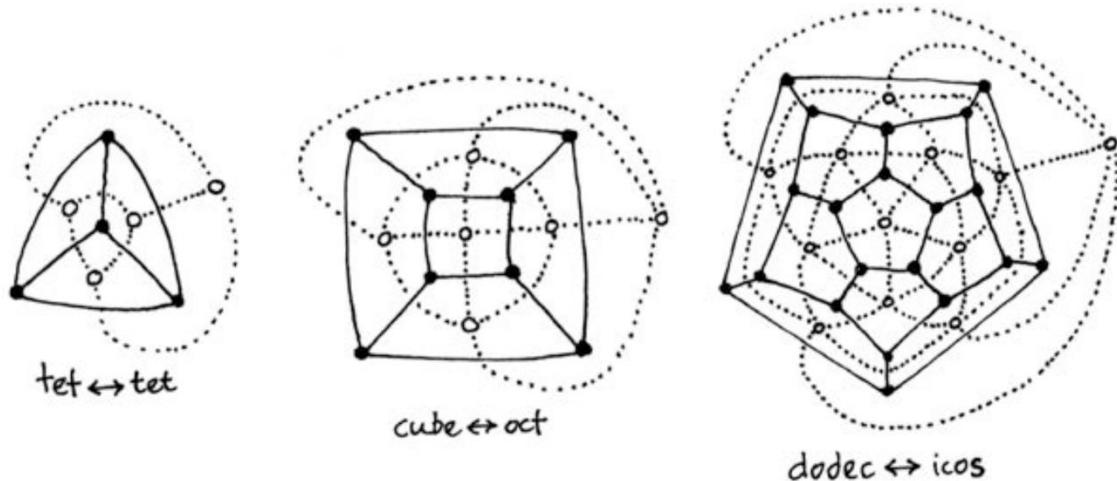
# 半边的替代方案

口很多类似的数据结构

- Winged edge
- Corner table
- Quadedge
- ...

Paul Heckbert

quadedge code - <http://bit.ly/1QZLHos>



口类似半边，存储本地邻居的信息

口与简单多边形列表有类似的权衡

- **CONS**: 附加存储/指针，不连贯的内存访问
- **PROS**: 更好地访问单个元素，直观地遍历局部元素

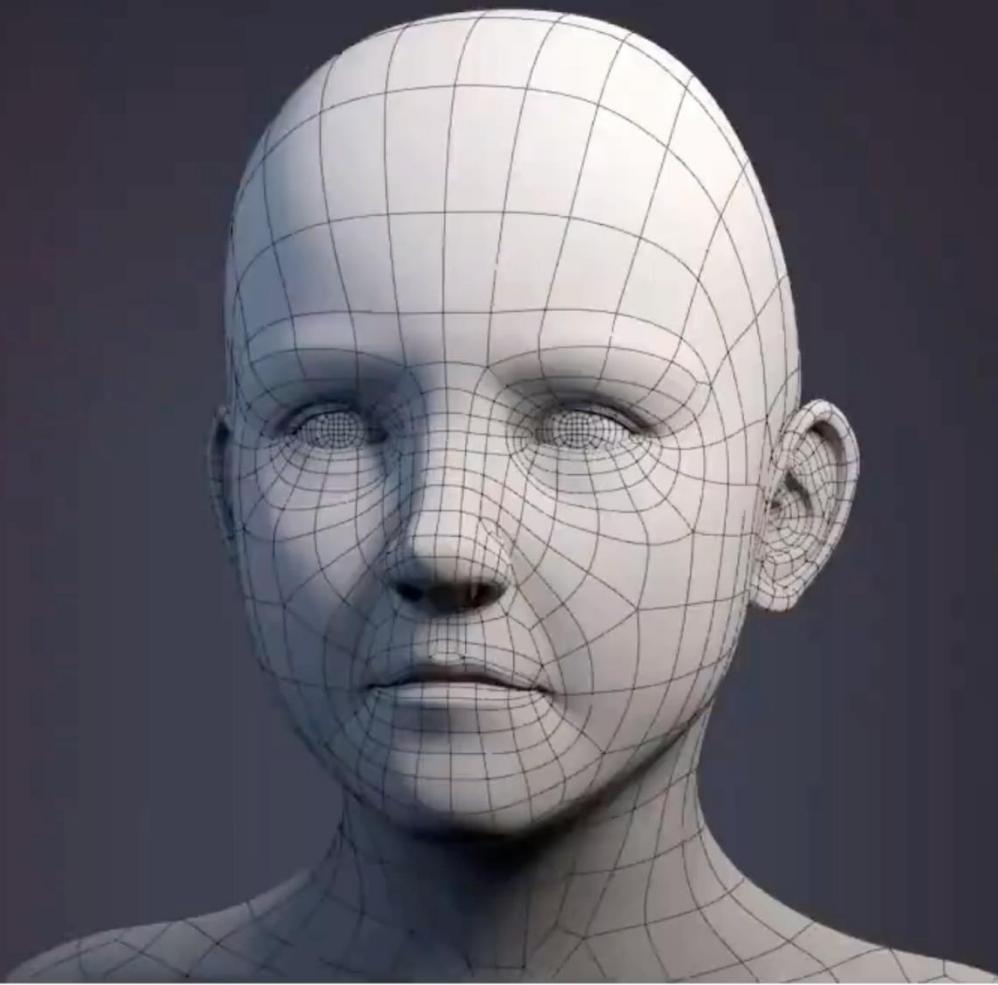
# 不同多边形网格数据结构的比较

	Adjacency List	Incidence Matrices	Halfedge Mesh
<b>constant-time neighborhood access?</b>	NO	YES	YES
<b>easy to add/remove mesh elements?</b>	NO	NO	YES
<b>nonmanifold geometry?</b>	YES	YES	NO

**Conclusion: pick the right data structure for the job!**

我们能用这些花哨的新数据  
结构做些什么？

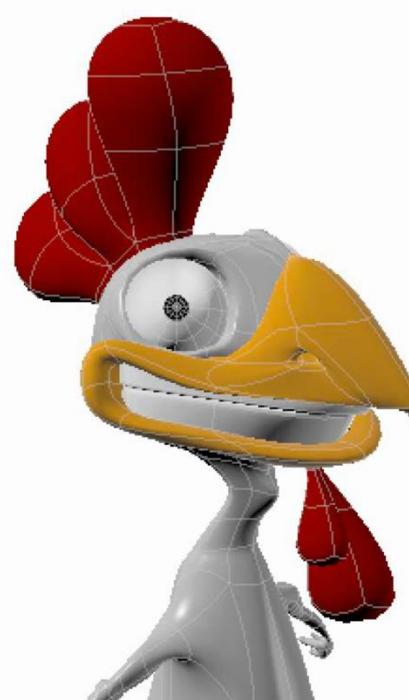
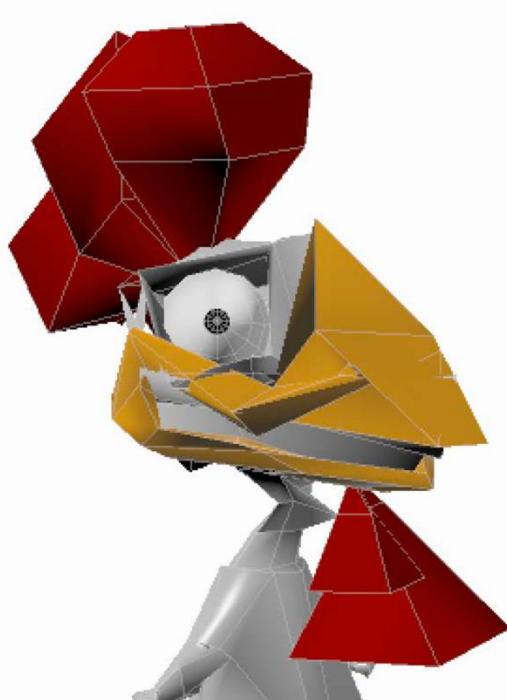
# 细分建模 Subdivision modeling



# 细分建模 Subdivision modeling

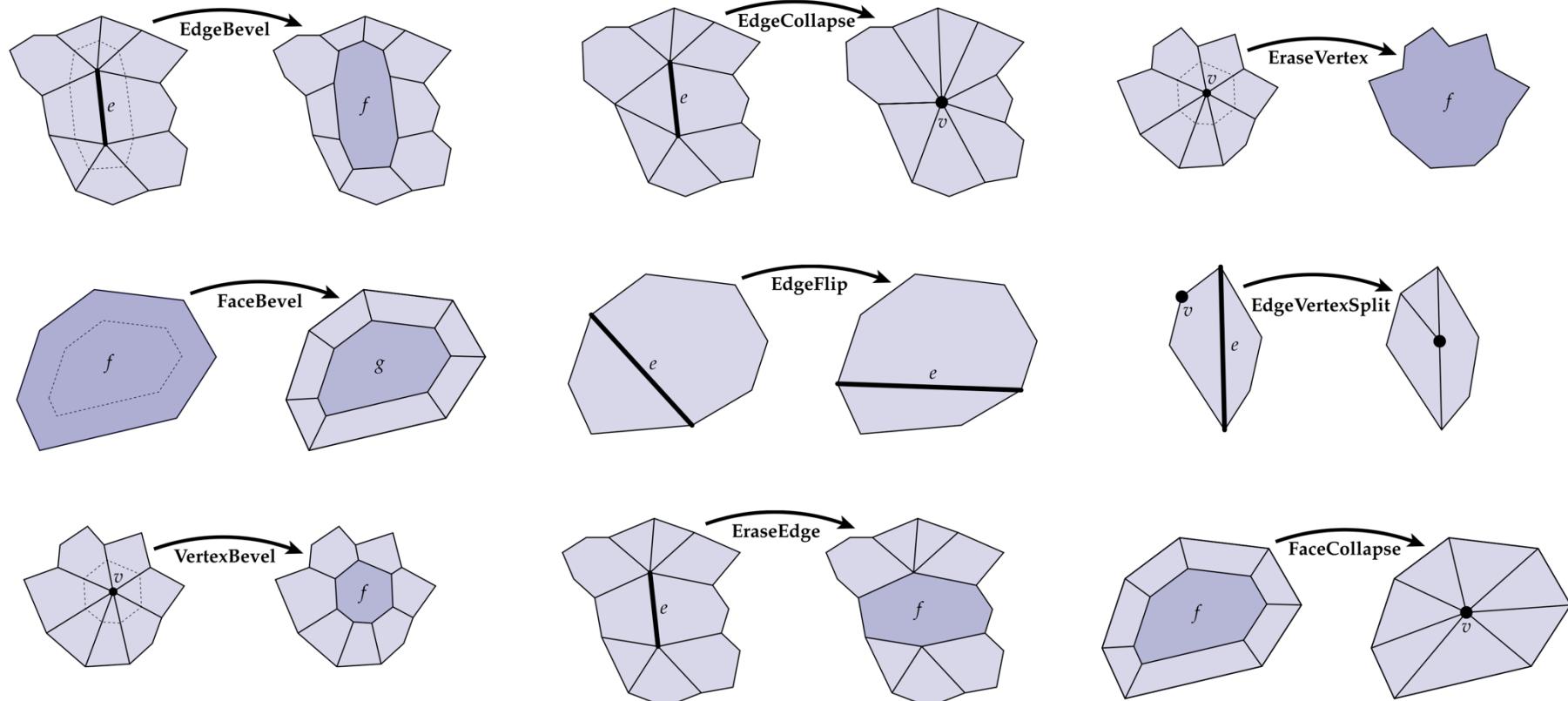
□ 现代三维工具中常见的建模范式：

- 粗糙的“控制笼 (control cage)”
- 执行局部操作以控制/编辑形状
- 通过全局的细分过程确定最终曲面



# 细分建模 – 局部操作

对于一般的多边形网格，我们有许多对建模有用的局部网格操作：

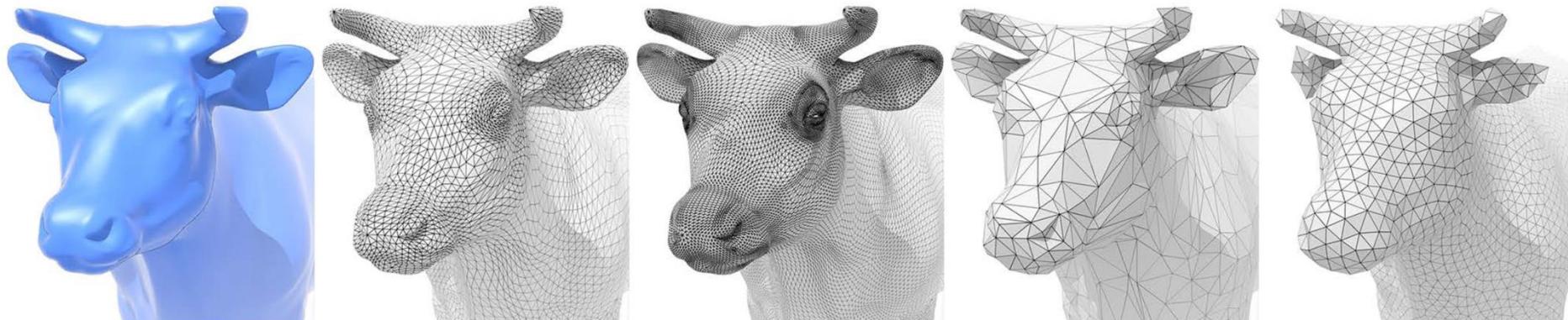


**...and many, many more!**

# 几何网格处理

扩展传统的数字信号处理 (音频/视频等) 以处理几何信号：

- 上采样 / 下采样 / 重采样 / 滤波 ...
- 走样 (重建的表面给出 “错误印象” )



原始网格

网格细分

网格简化

网格规则化



中山大學 软件工程学院  
SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

谢谢

陈壮彬  
软件工程学院  
[chenzhb36@mail.sysu.edu.cn](mailto:chenzhb36@mail.sysu.edu.cn)