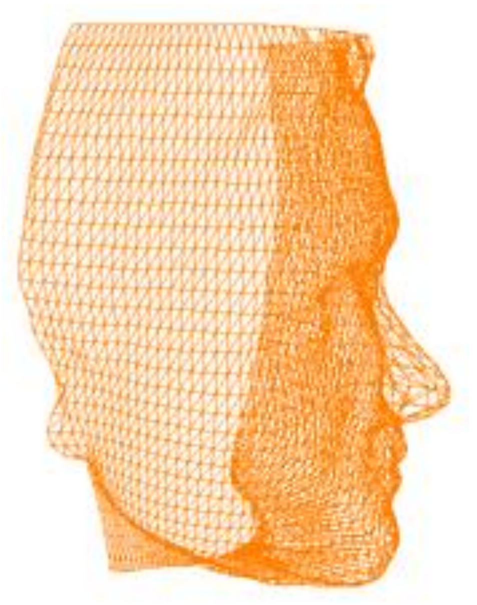
COSC422 Advanced Computer Graphics



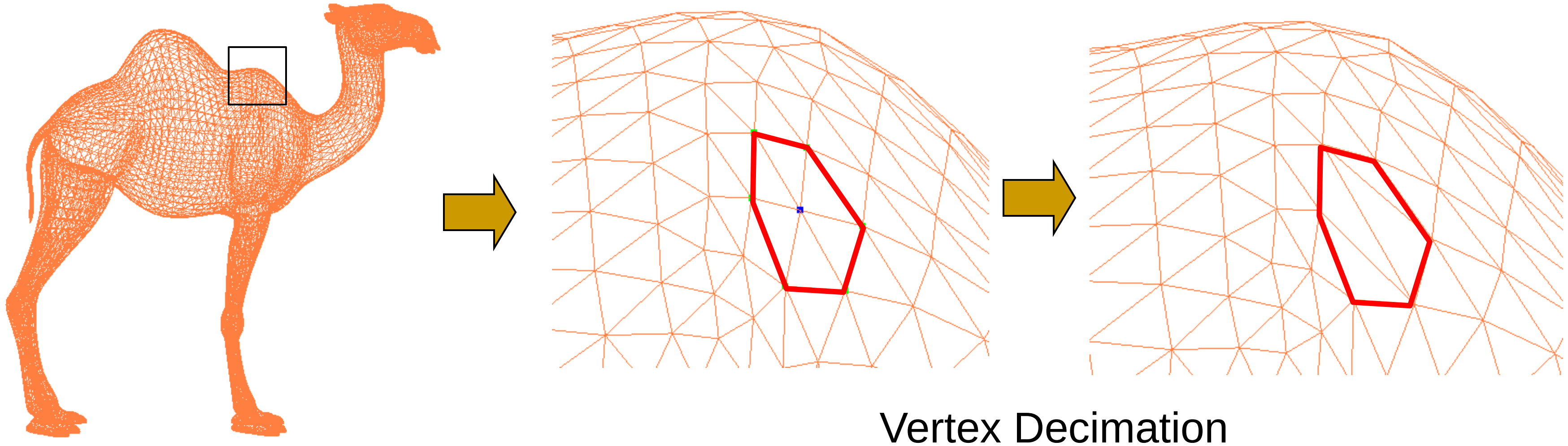
7 Mesh Processing

Semester 2
2021

R. Mukundan (mukundan@canterbury.ac.nz)
Department of Computer Science and Software Engineering
University of Canterbury, New Zealand.

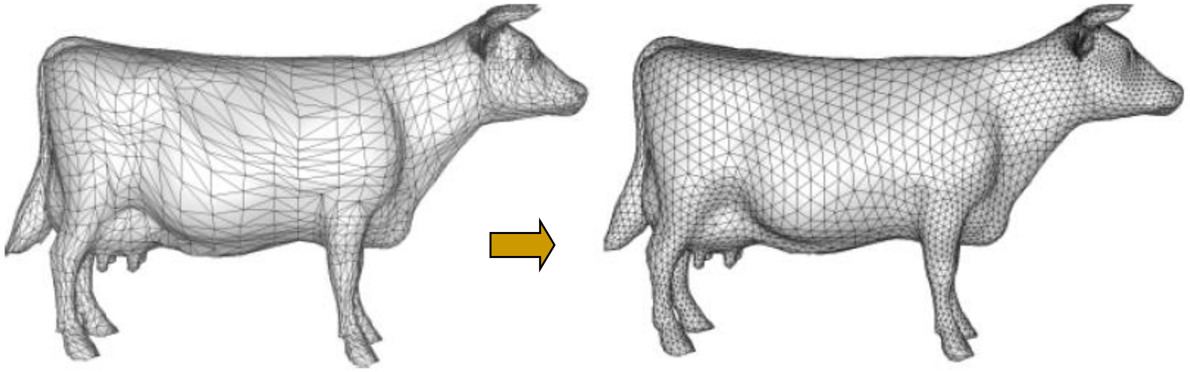
3D Mesh Processing

- Detailed geometric models use highly complex meshes.
- Mesh models will often need to be modified: Eg., Sculpting and repair, simplification, subdivision.
- Mesh algorithms require efficient data structures for performing **local operations around vertices**.

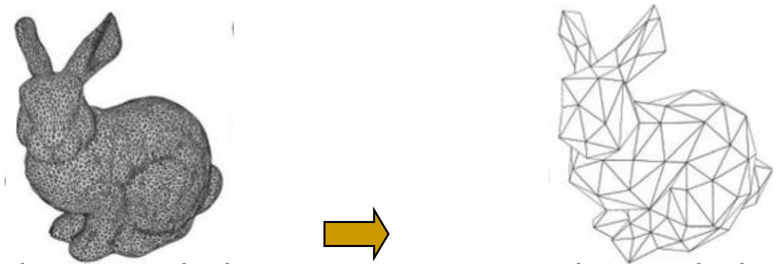

Vertex Decimation

Mesh Processing Algorithms

- Remeshing:

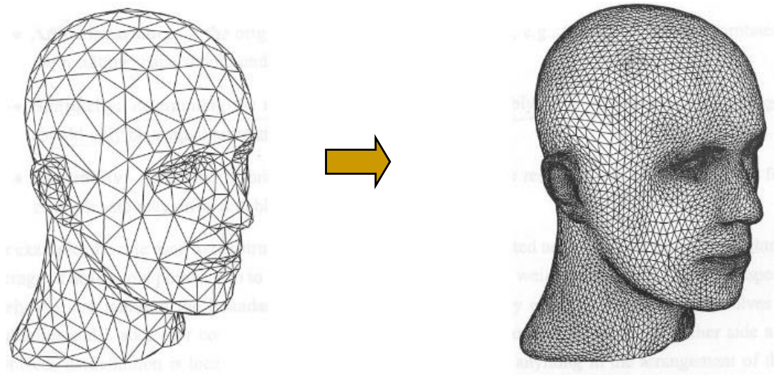


- Mesh Simplification



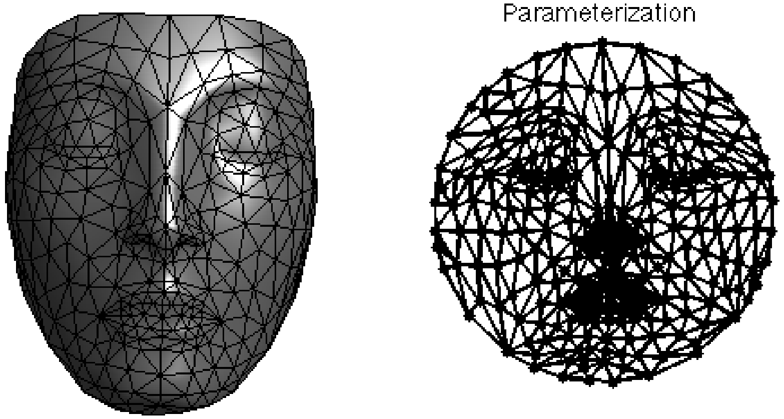
(69451 polys) (251 polys)

- Mesh Subdivision



Mesh Processing Algorithms

- Mesh Parameterization

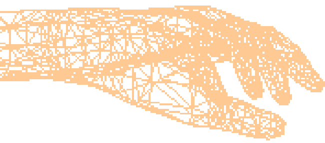


- Mesh Deformation/Morphing

3D Mesh Files

3D mesh files are usually created by content designers.

- Popular 3D mesh editing tools: 3D Studio Max, AutoCAD, Maya, Blender, Zbrush, MakeHuman
- Several mesh file formats exist: OFF, OBJ, PLY, 3DS, Blend, B3D, DXF, LWO, STL, X3D, X, GLTF
- OFF, OBJ, PLY are editable text files that can be easily parsed, while others have complex structures.
- A mesh file may store several things: verts, polys, normals, textures, materials, lights, matrices, bones…
- 3D graphics application development often requires 3D model loaders!



Object File Format (OFF)

The simplest ASCII mesh file format containing only the most basic information.

We cannot store texture coordinates, normals or material definitions in an OFF file.

Always begins with keyword OFF

Number of vertices, polygons and edges

```
OFF
8 6 0
-0.5 -0.5 0.5
0.5 -0.5 0.5
-0.5 0.5 0.5

4 0 1 3 2
4 2 3 5 4
4 4 5 7 6
```

Vertex list:
Vertex coordinates x, y, z

Face list:
No. of vertices per face (4=quad), Vertex indices

Wavefront Object Format (OBJ)

A versatile file format that can store several mesh related attributes.

```
# sample mesh file
v 1 1 1
v 1 1 -1
vt 0 0.5
vt 0.2 0.2
vn -0.15 0.12 0.8
vn 0.9 -0.01 0.012
f 1 3 4 2
f 2/1 4/2 3/3
f 5/3/8 9/6/4/ 2/2/1
f 8//1 6//2  3//4
```

comment line

Vertex coords

Texture coords

Normal components

Face elements v/t/n

Note: The starting value of indices is 1.

Ref: http://paulbourke.net/dataformats/obj/

OBJ File: Material Definition

Materials are stored in external files with .mtl extension and referenced from the OBJ file.

```
# sample mesh
o cube
mtlib cube.mtl
v 1 1 1
v 1 1 -1
usemtl  red
f 1 3 4 2
f 2/1 4/2 3/3
usemtl  green
f 5/3/8 9/6/4/ 2/2/1
f 8//1 6//2  3//4
```

User defined object name – ignored.

Material file name

Material name

Material name

The default material colour is white.

OBJ Material File (MTL)

The material file contains the definitions for each named material in the OBJ file

cube.mtl :

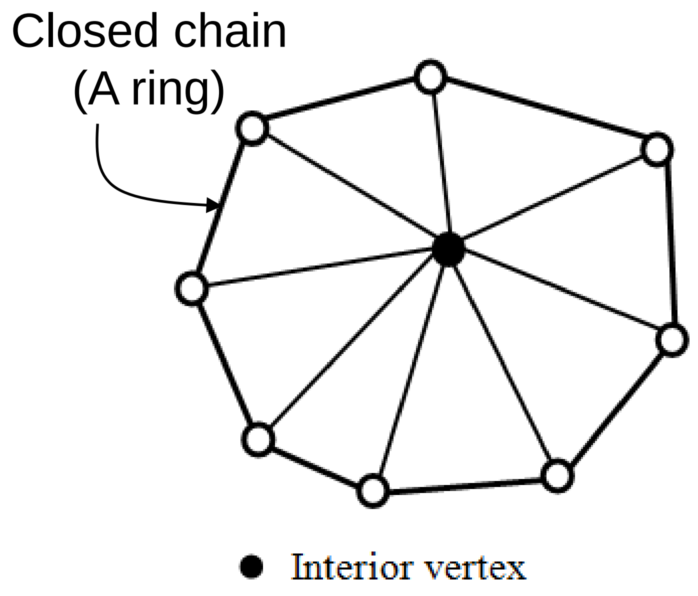
```
newmtl green
   Ka 0.1 0.1 0.1
   Kd 0 1 0
   Ks 0.0000 0.0000 0.0000
   Ns 0.0000
newmtl red
   Ka 0.1 0.1 0.1
   Kd 1 0 0
   Ks 1 1 1
   Ns 10.0000
```

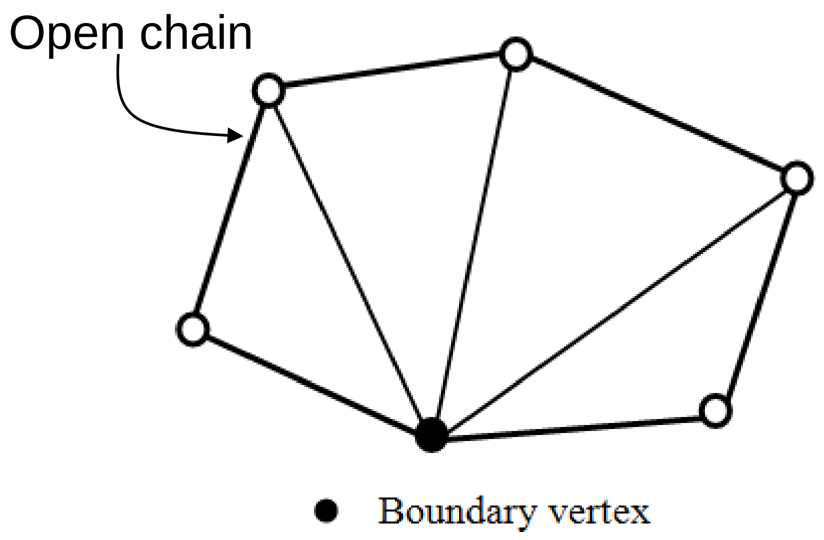
- Ambient material property
- Diffuse material property
- Specular material property
- Phong’s exponent

Polygonal Manifolds

A polygonal manifold mesh (or simply, a manifold mesh) satisfies two conditions:

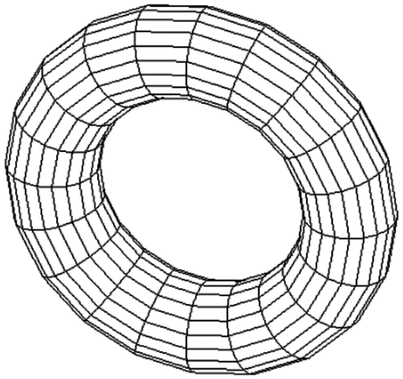
- No edge is shared by more than two faces,
- The faces sharing any vertex can be ordered in such a way that their vertices excluding the shared vertex form a simple chain. The chain can be either closed or open.

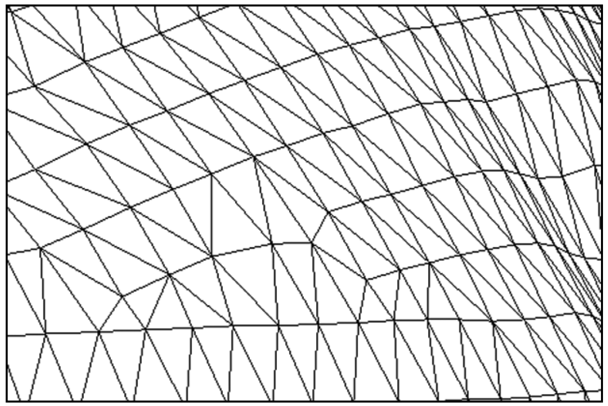




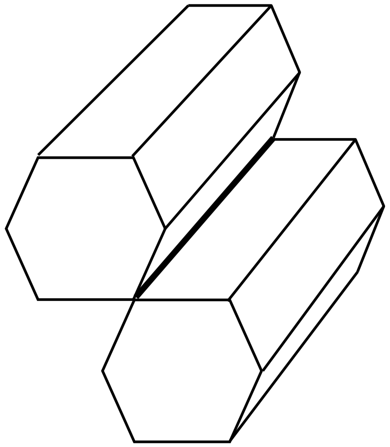
Polygonal Manifolds

Manifold Meshes

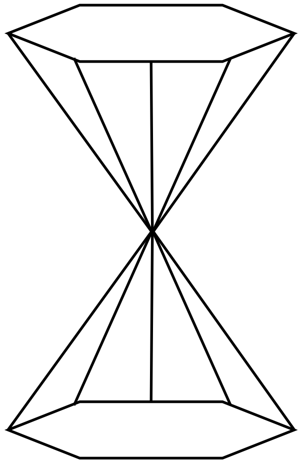




Non-manifold Meshes



An edge shared by more than two faces



The neighbours of a vertex do not form a single chain

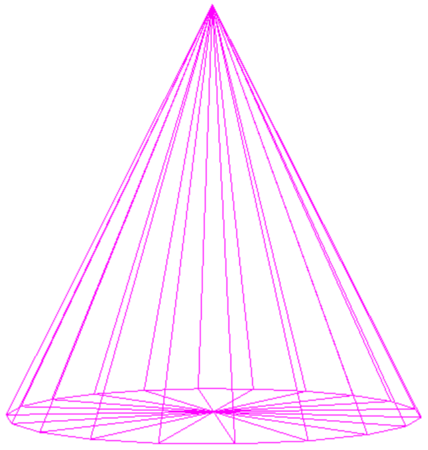
Euler-Poincaré Formula

Assumption: Polygonal manifold mesh.

V = number of vertices, F = number of faces,

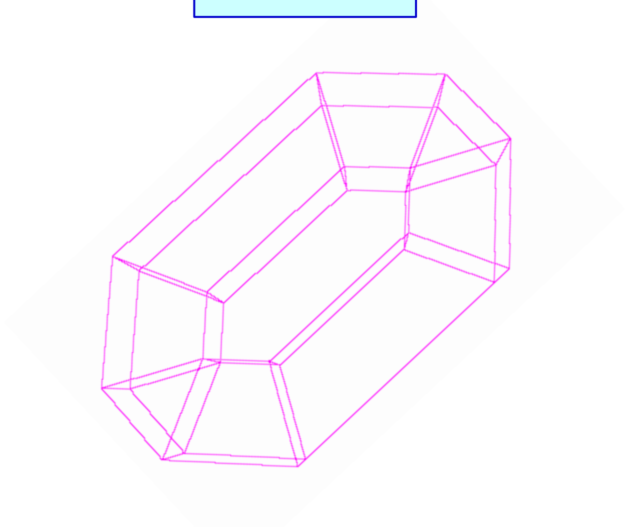
E = number of edges, g = genus: **V + F – E = 2- 2g**

Genus 0



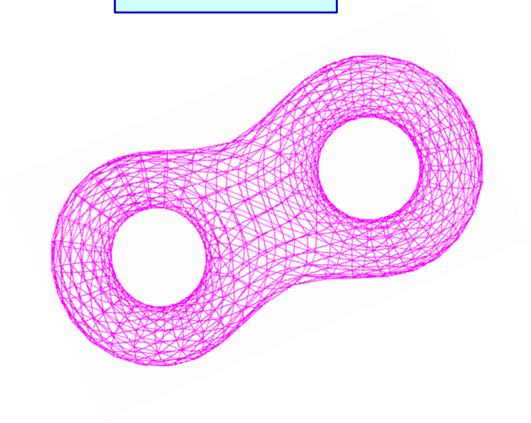
V = 22
F = 40
E = 60
V + F - E = 2

Genus 1



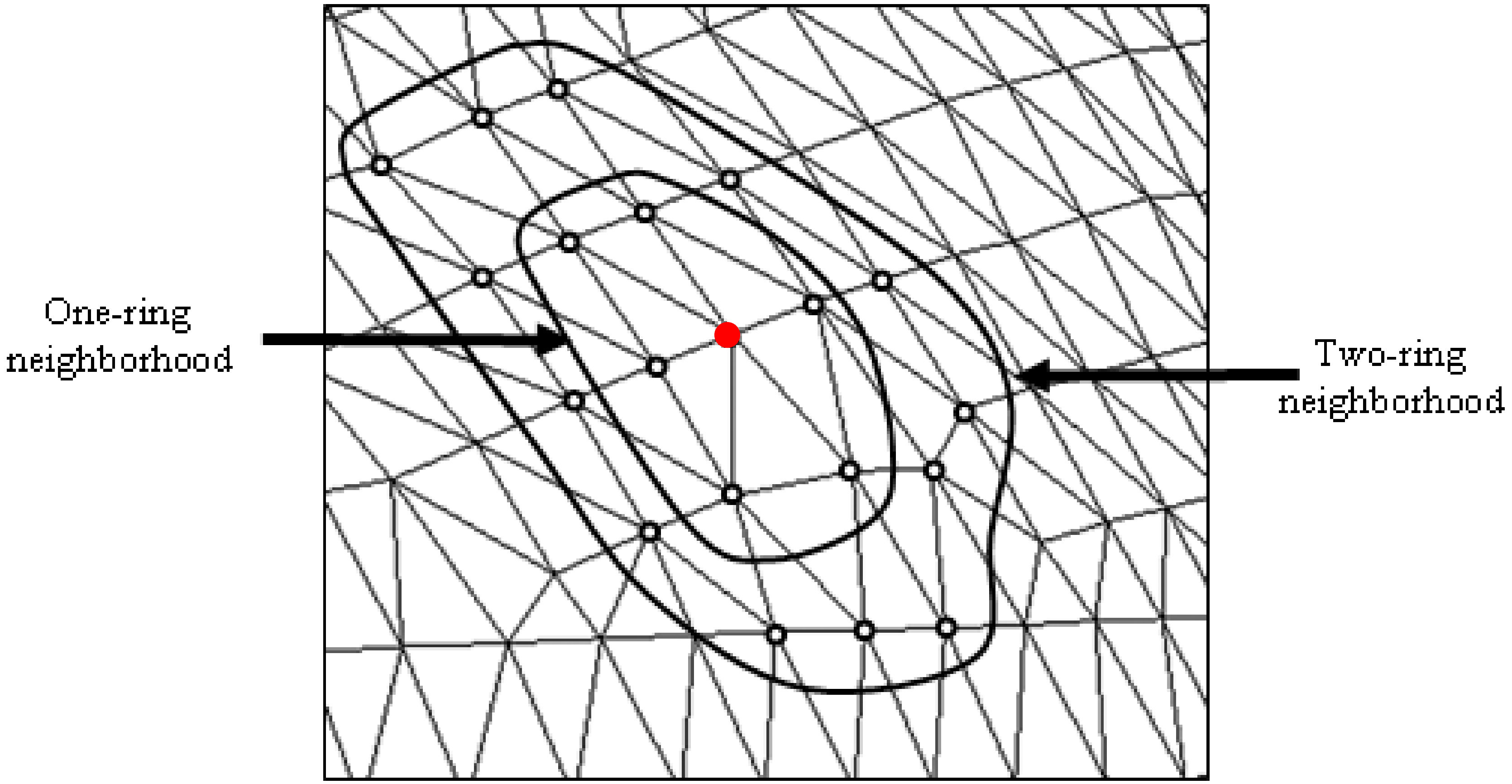
V = 28
F = 56
E = 84
V + F - E = 0

Genus 2



V = 766
F = 1536
E = 2304
V + F - E = -2

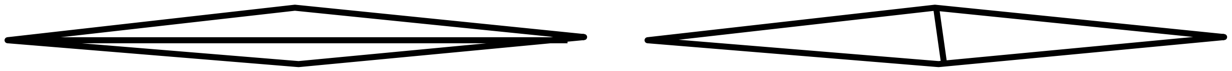
Ring Neighbourhoods



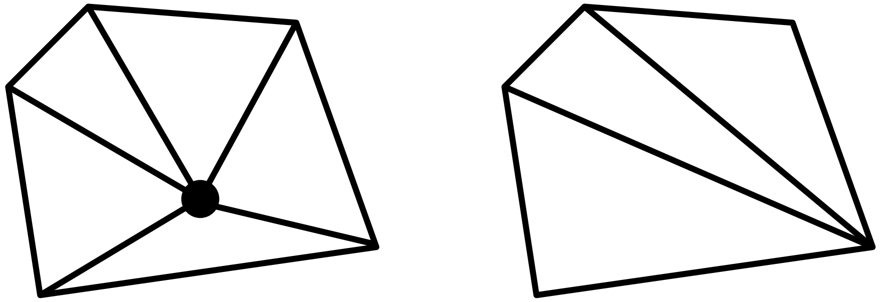
One- ring neighbourhood: The set of adjacent vertices to a given vertex.
Two-ring neighbourhood: The union of one-ring neighbourhoods of adjacent vertices.

Common Mesh Operations

Edge flipping



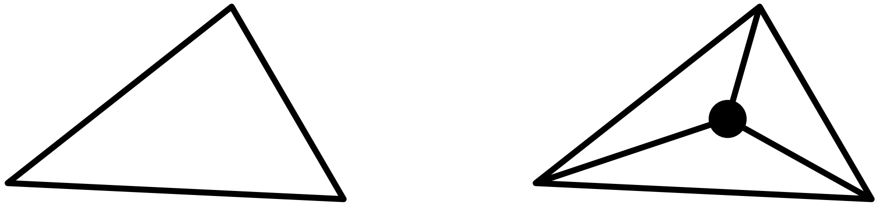
Vertex removal and re-triangulation (usually for mesh simplification)



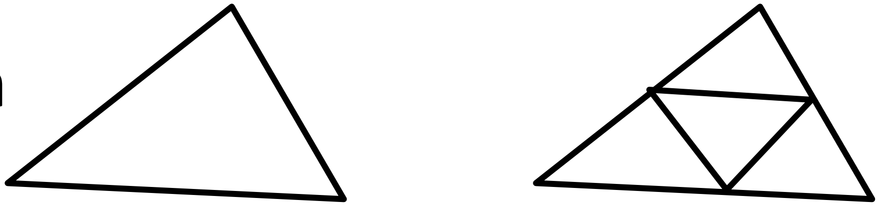
Vertex addition

or

Triangle subdivision



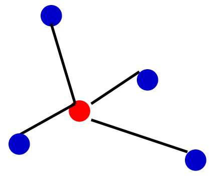
(usually for mesh subdivision)



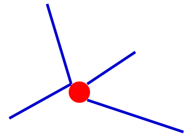
Adjacency Queries

Mesh elements: V: Vertices, E: Edges, F: Faces

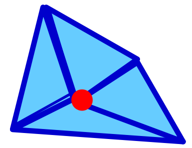
- V $\rightarrow$ V: Given a vertex, find all vertices that are adjacent to it (one-ring neighbourhood).



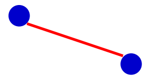
- V $\rightarrow$ E: Given a vertex, find all edges that are incident at that vertex.



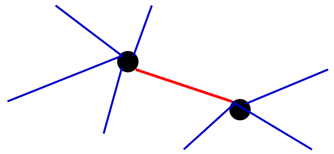
- V $\rightarrow$ F: Given a vertex, find all faces that share the vertex.



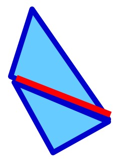
- E $\rightarrow$ V: Given an edge, find the vertices that form its end points.



- E $\rightarrow$ E: Given an edge, find its neighbouring edges.

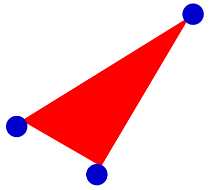


- E $\rightarrow$ F: Given an edge, find its two bordering faces.

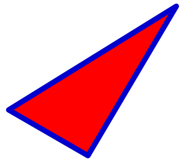


Adjacency Queries

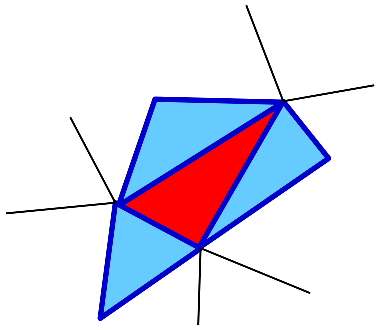
- F $\rightarrow$ V: Given a face, find all its vertices. (Directly obtained from the face list)



- F $\rightarrow$ E: Given a face, find all its edges.



- F $\rightarrow$ F: Given a face, find all neighbouring faces.



Mesh Data Structures

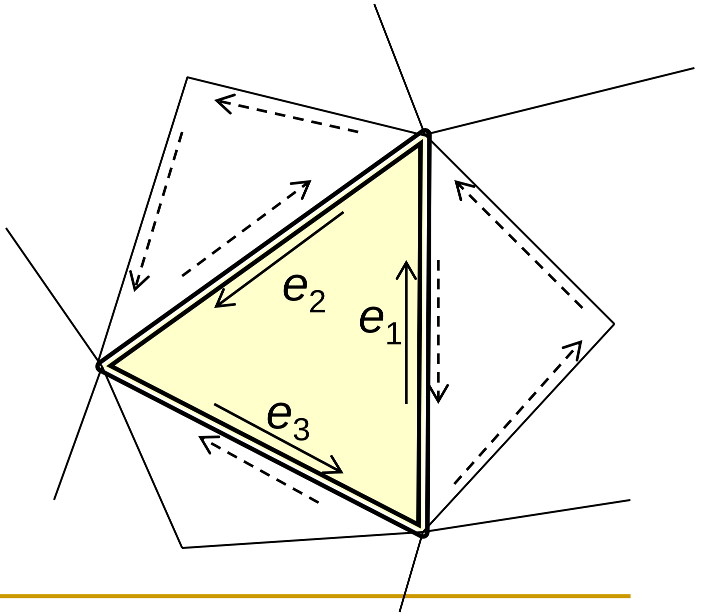
- Winged Edge Data Structure (Baumgart, 1975)
- **Half-Edge Data Structure** [HEDS] (Eastman, 1982)
- Split-Edge Data Structure
- Corner Data Structure
- QuadEdge Data Structure (Guibas and Stolfi, 1985)
- FacetEdge Data Structure (Dobkin and Laszlo, 1987)

…

Mesh data structures are designed to efficiently perform local mesh search in the **neighbourhood of a vertex** without having to traverse the whole mesh.

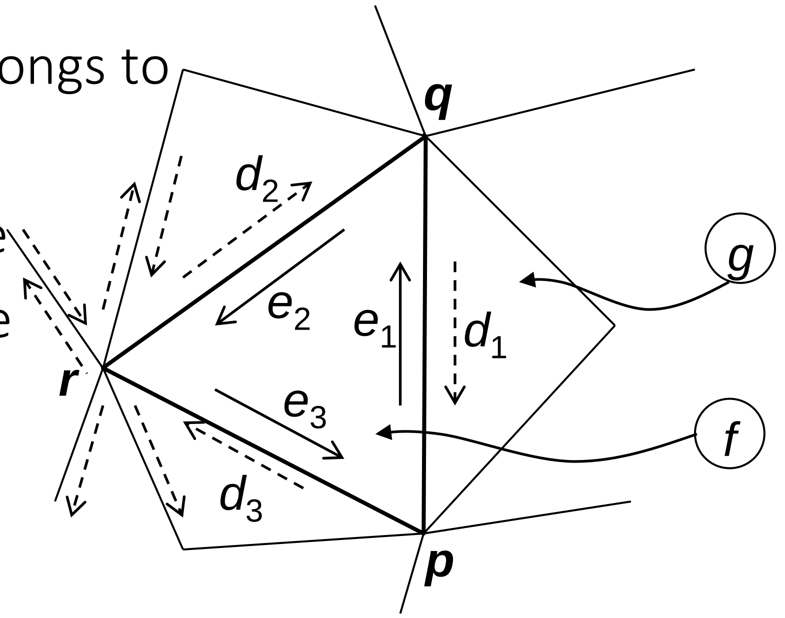
Half-Edge Data Structure

- Each edge is divided into two **directed half-edges**
- Each half-edge belongs to a single face.
- **Each triangle has exactly 3 half-edges**. The total number of half-edges in a triangle mesh is exactly three times the number of triangles.
- The half-edges always have a counter-clockwise ordering on each face.
- Written as “halfedge”.



Halfedge Data Structure

- The data structure consists of three components
  - **Vertex:** Each vertex points to *one of the* outgoing halfedges.
  - **Face:** Each face points to *one of its* halfedges
  - **Halfedge:** Each halfedge contains pointers to neighbouring elements:
    - A pointer to the unique vertex it points to
    - A pointer to the unique face it belongs to
    - A pointer to the next halfedge
    - A pointer to the previous halfedge
    - A pointer to the opposite halfedge



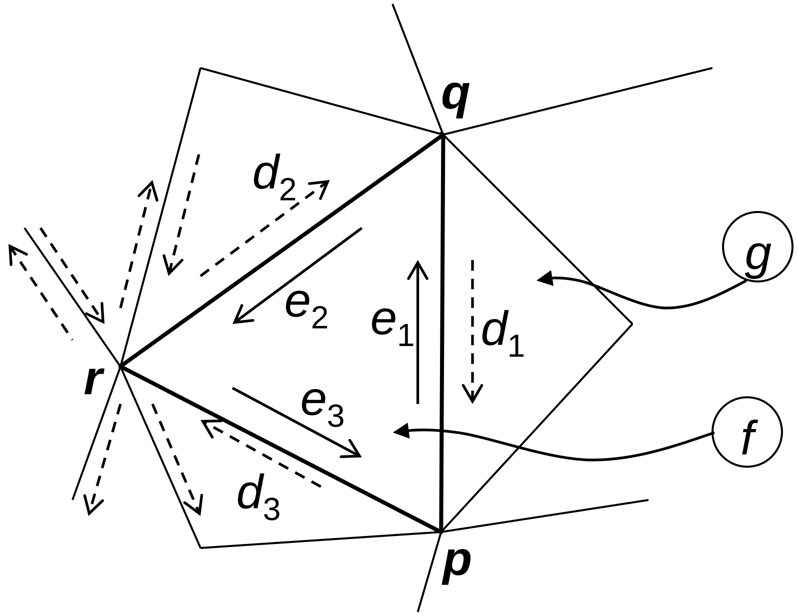
Half-Edge Data Structure: Example

```
struct HE_edge
{
   HE_vert* vert;   //The vertex the edge points to
   HE_face* face;   //The face the edge belongs to
   HE_edge* next;   //The next halfedge
   HE_edge* prev;   //The previous halfedge (optional)
   HE_edge* pair;   //The opposite halfedge
}

struct HE_vert
{
   HE_edge* edge;  //An outgoing halfedge from the vertex
   float x, y, z;
}

struct HE_face
{
   HE_edge* edge;  //A halfedge belonging to the face
}
```

Half-Edge Data Structure

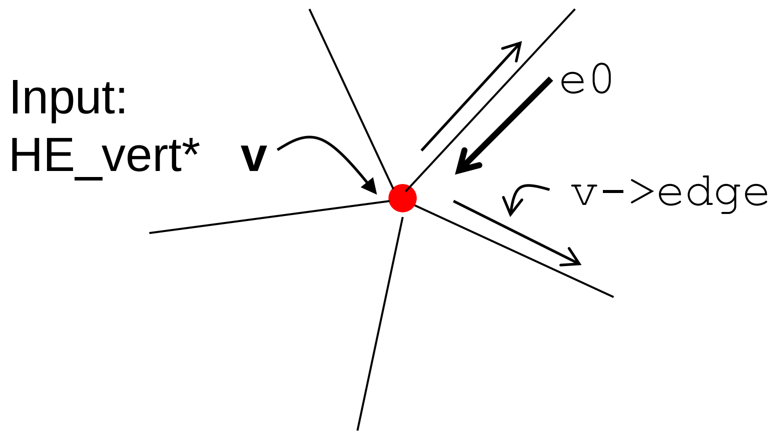
```
HE_edge *e1, *e2, *e3, *d1, *d2, *d3;
HE_vert *p, *q, *r;
HE_face *f, *g;

e1->vert = q;
e1->face = f;
e1->next = e2;
e1->pair = d1;

r->edge = e3;
p->edge = e1;
f->edge = e2;
g->edge = d1;
```

$\left.\begin{array}{l} \\ \\ \\ \\ \end{array}\right\}$ Unique



Half-Edge Data Structure

Query V→E
(Slide 15)



Anticlockwise enumeration of incident halfedges at a vertex.

```
HE_edge *e0 = v->edge->prev;
HE_edge *edge = e0;
do
{
  output(edge);
  edge = edge -> pair -> prev;
} while (edge != e0);
```

Clockwise enumeration of incident halfedges at a vertex.

```
HE_edge *e0 = v->edge->prev;
HE_edge *edge = e0;
do
{
  output(edge);
  edge = edge -> next -> pair;
} while (edge != e0);
```

Simple Mesh Operations

- Retrieving end-points of an `edge`:

Query E→V

```
HE_vert* vert1 = edge->vert;
HE_vert* vert2 = edge->pair->vert;
```

- Retrieving the two faces that border an `edge`:

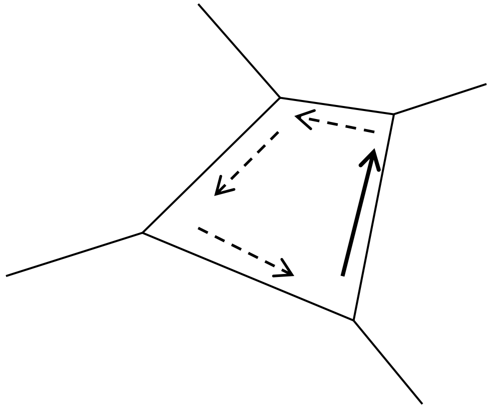
Query E→F

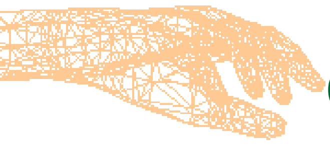
```
HE_face* face1 = edge->face;
HE_face* face2 = edge->pair->face;
```

- Retrieving vertices of a `face`:

Query F→V

```
HE_edge* edge = face->edge;
do{
    output(edge->vert);
    edge = edge->next;
  } while (edge!= face->edge);
```

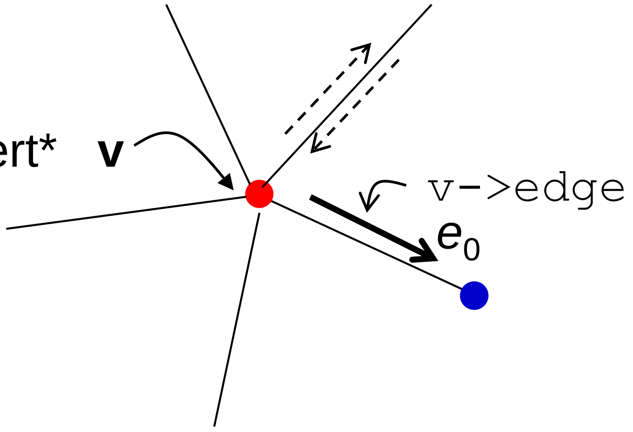




One-Ring Neighbourhood

Query V→V
(Slide 15)

Input:
HE_vert* **v**



```
HE_edge *e0 = v->edge;
HE_edge *edge = e0;
do
{
   output(edge->vert);
   edge = edge -> prev -> pair;
} while (edge != e0);
```

Mesh Processing Software

- MeshLab:

  http://www.meshlab.net/

- PMP: Polygon Mesh Processing: v1.0 Feb 2019

  https://www.pmp-library.org/

- CGAL: Computational Geometry Algorithms Library:

  https://www.cgal.org/

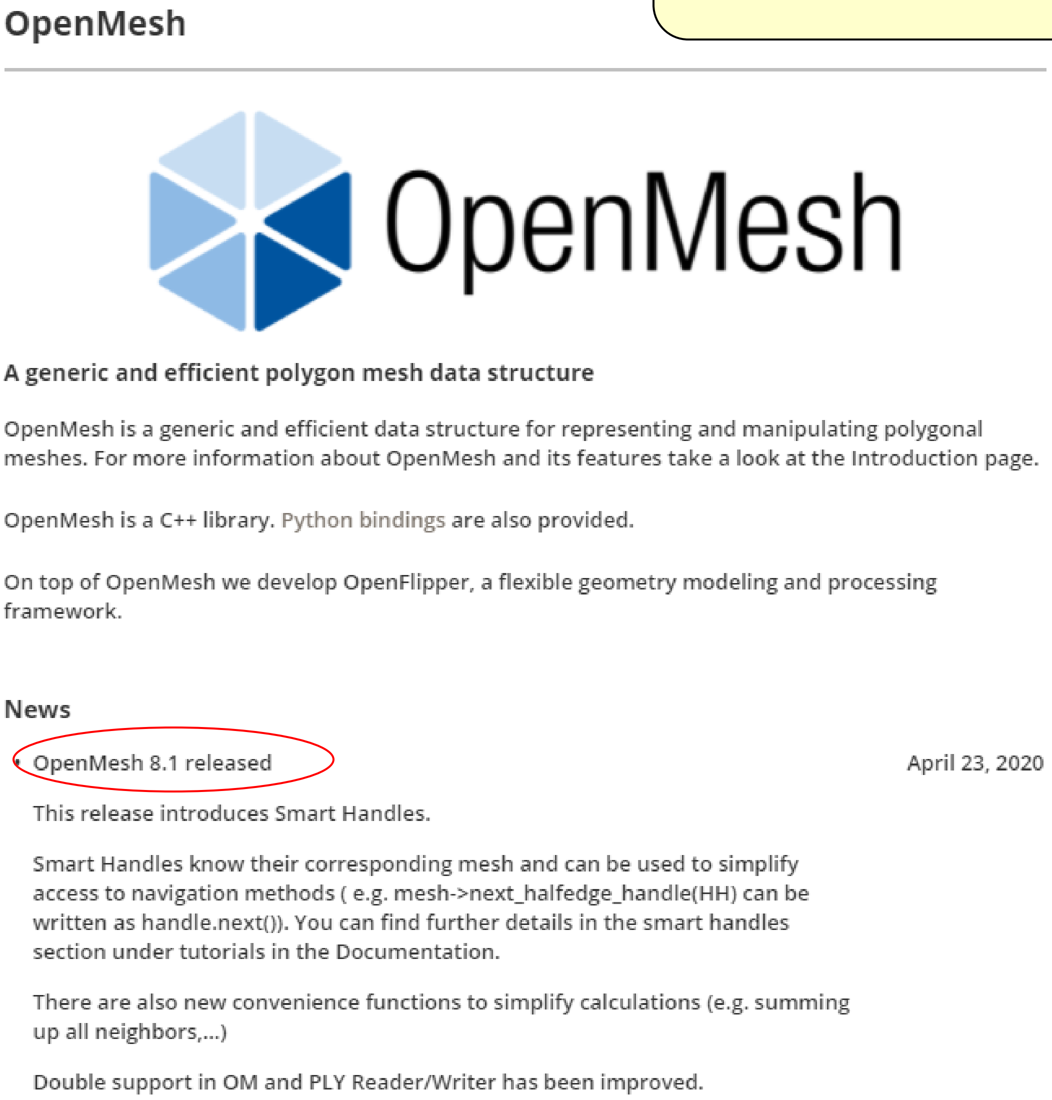
- **OpenMesh**:

  https://www.openmesh.org/

OpenMesh

https://www.openmesh.org/

Supported Mesh File Formats:
OFF, OBJ, PLY, STL



OpenMesh

A generic and efficient polygon mesh data structure

OpenMesh is a generic and efficient data structure for representing and manipulating polygonal meshes. For more information about OpenMesh and its features take a look at the Introduction page.

OpenMesh is a C++ library. Python bindings are also provided.

On top of OpenMesh we develop OpenFlipper, a flexible geometry modeling and processing framework.

News

- OpenMesh 8.1 released — April 23, 2020

  This release introduces Smart Handles.

  Smart Handles know their corresponding mesh and can be used to simplify access to navigation methods (e.g. mesh->next_halfedge_handle(HH) can be written as handle.next()). You can find further details in the smart handles section under tutorials in the Documentation.

  There are also new convenience functions to simplify calculations (e.g. summing up all neighbors,...)

  Double support in OM and PLY Reader/Writer has been improved.

https://www.openmesh.org/intro/

OpenMesh provides the following features:

- Representation of arbitrary polygonal (the general case) *and* pure triangle meshes (providing more efficient, specialized algorithms)
- Explicit representation of vertices, halfedges, edges and faces.
- Fast neighborhood access, especially the one-ring neighborhood (see below).
- Highly customizable :
  - Choose your coordinate type (dimension and scalar type)
  - Attach user-defined elements/functions to the mesh elements.
  - Attach and check for attributes.
  - Attach data at runtime using dynamic properties.

In addition we provide some sample applications that demonstrate the usage of OpenMesh:

- Mesh Smoothing.
- Mesh Decimation.
- Qt integration.

The halfedge data structure

Polygonal meshes consist of geometry (vertices) and topology (edges, faces). Data structure for polygonal meshes mainly differ in the way they store the topology information. While face-based structures lack the explicit representation of edges, and edge-based structures loose efficiency because of their missing orientation of edges, halfedge-based structures overcome this disadvantages. The halfedges (that result in splitting the edges in two *oriented* parts) store the main connectivity information:

- one vertex
- one face
- the next halfedge

OpenMesh: Initialization

- General polygonal manifold mesh

```
#include <OpenMesh/Core/Mesh/PolyMesh_ArrayKernelT.hh>
typedef OpenMesh::PolyMesh_ArrayKernelT<> MyMesh;
MyMesh mesh;
...
OpenMesh::IO::read_mesh(mesh, “cube.off")
```

- Triangle mesh

```
#include <OpenMesh/Core/Mesh/TriMesh_ArrayKernelT.hh>
typedef OpenMesh::TriMesh_ArrayKernelT<> MyMesh;
MyMesh mesh;
...
OpenMesh::IO::read_mesh(mesh, “cube.off")
```

cube.off

```
OFF
8 12 0
-1  1 -1
-1 -1 -1
 1 -1 -1
 1  1 -1
-1  1  1
-1 -1  1
 1 -1  1
 1  1  1
3 0 3 1
3 3 2 1
3 4 5 7
3 5 6 7
3 0 1 5
3 4 0 5
3 7 6 2
3 3 7 2
3 0 4 7
3 0 7 3
3 5 1 2
3 5 2 6
```

OpenMesh: Basic Types, Functions

```
OpenMesh::Vec3f p = { 1, 2, 3 };       //a point or a vector
OpenMesh::Vec3f n = { 0, 1, 0 };       //a point or a vector
MyMesh::Point q = { 10, 20, 30 };      //a point
MyMesh::Normal m = { 0.6, 0.8, 0};     //a vector
```

Basic Operations:

```
len = p.length();
glVertex3fv(p.data());
glNormal3fv(n.data());
d = n1 | n2;    //dot product
m = n1 % n2;    //cross product
float x = p[0],  y = p[1],  z = p[2];   //component access
```

Number of vertices: `mesh.n_vertices()`

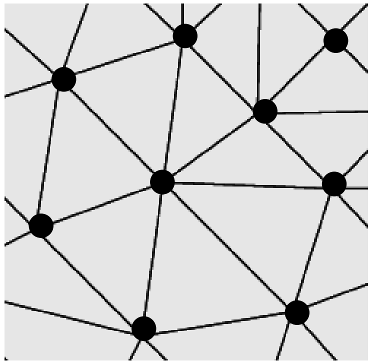
Number of faces: `mesh.n_faces()`

Number of edges: `mesh.n_edges()`

See Slide 12

Traversing a Mesh: Vertex Iterator

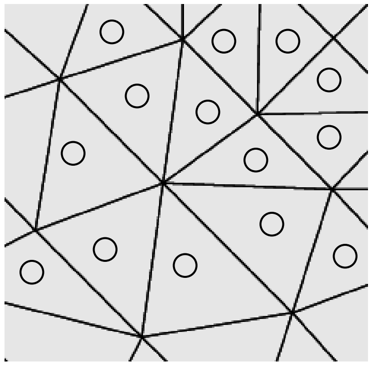
- A vertex iterator is used to visit all vertices of a mesh. Dereferencing a vertex iterator gives a handle (reference) to the current vertex
- Using a vertex handle, we can find the position and other available attributes of a vertex.



```
MyMesh::VertexIter vit;        //Vertex iterator
MyMesh::VertexHandle veH;      //Vertex handle
MyMesh::Point p;
int index;
for (vit=mesh.vertices_begin(); vit!=mesh.vertices_end();  vit++)
{
   veH = *vit;              //or,  veH = vit.handle();
   p = mesh.point(veH);     //vertex coordinates
   index = veH.idx();       //vertex index
}
```

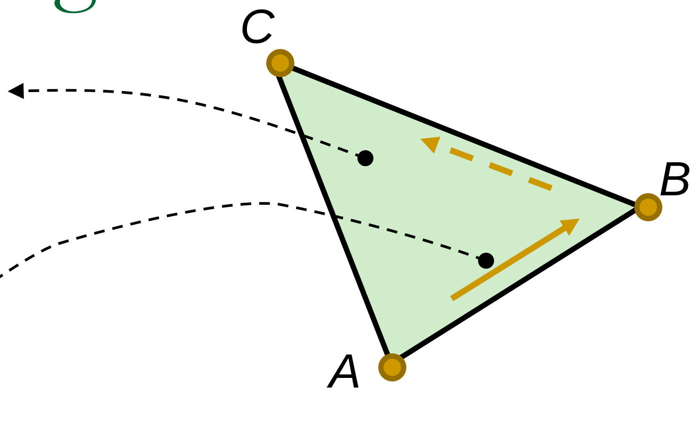
Traversing a Mesh: Face Iterator

- A face iterator allows us to visit all faces of a mesh. Dereferencing a face iterator gives a handle (reference) to the current face.
- Using a face handle, we can find the properties associated with that face.



```
MyMesh::FaceIter fit;           //Face iterator
MyMesh::FaceHandle faH;         //Face handle
MyMesh::Normal n;
int index;
for (fit = mesh.faces_begin();  fit != mesh.faces_end();  fit++)
{
   faH = *fit;
   n = mesh.normal(faH);   //Face normal, if available
   index = faH.idx();      //Face index
}
```

Processing a Triangle



```
OpenMesh::FaceHandle faH  = *fit;
```

```
OpenMesh::HalfedgeHandle heH = mesh.halfedge_handle(faH);
```

Handle for Vertex A:

```
    OpenMesh::VertexHandle veH1 = mesh.from_vertex_handle(heH);
```

Handle for Vertex B:

```
    OpenMesh::VertexHandle veH2 = mesh.to_vertex_handle(heH);
```

Handle for Vertex C:

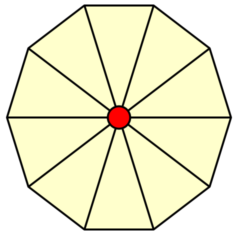
```
    OpenMesh::VertexHandle veH3 = mesh.opposite_vh(heH);
```

Handle for the next halfedge (dashed line):

```
OpenMesh::HalfedgeHandle heH2 = mesh.next_halfedge_handle(heH);
```

OpenMesh: Valence

- Vertex valence (or degree):
  - = number of halfedges from the vertex
  - = number of halfedges to the vertex
  - = number of one-ring neighbours of the vertex
  - `int nedges = mesh.valence(veH);`
  - Note: Vertex valence is **not** the number of faces sharing a vertex. E.g., see next slide on Boundary Elements.



Vertex valence = 10

- Face valence:
  - = number of vertices of the face
  - `int nvert = mesh.valence(faH);`
  - For a triangle mesh, nvert = 3.

/ determine if triangle or quad

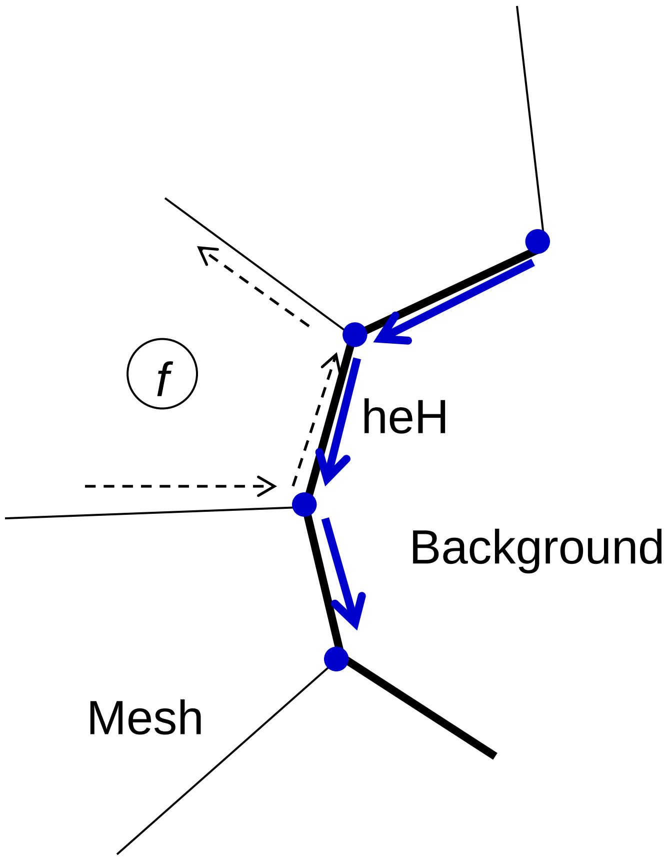
Useful for determining the primitive type

Boundary Elements

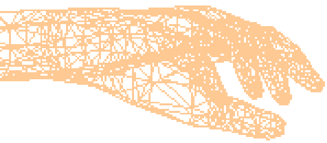
- A boundary half-edge does not belong to a face. Closed meshes do not have boundary edges.
- If a half-edge is a boundary edge, it next and previous half-edges are also a boundary edges.
- All boundary vertices satisfy the property

  `mesh.is_boundary(veH)  == true`
- All boundary half-edges satisfy the property

  `mesh.is_boundary(heH)  == true`



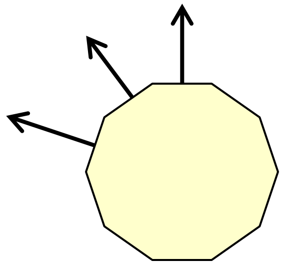
→ checking if boundary edge



OpenMesh: Normals

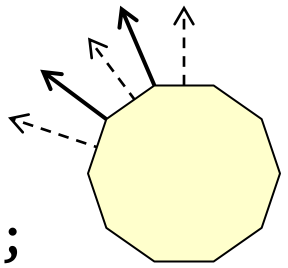
- Getting face normals:

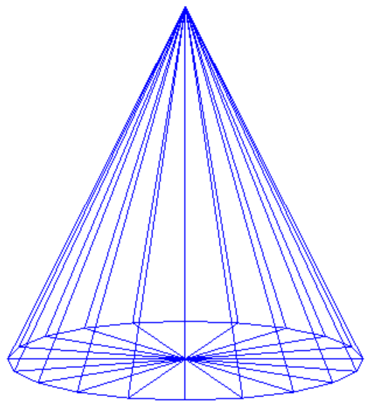
```
  if (!mesh.has_face_normals())
      mesh.request_face_normals();
   mesh.update_face_normals();
  ...
  normFace = mesh.normal(faH);
```

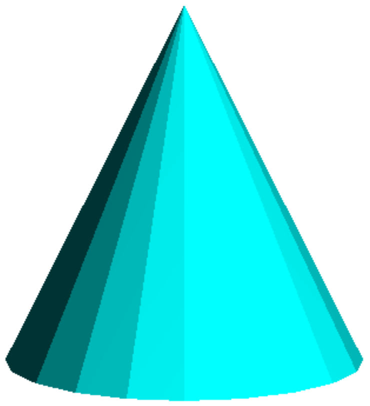


- Getting vertex normals:

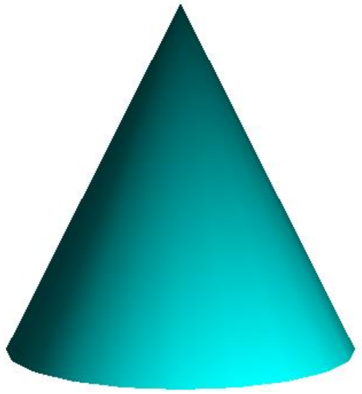
```
if (!mesh.has_vertex_normals())
      mesh.request_vertex_normals();
   mesh.update_face_normals();
  mesh.update_vertex_normals();
  ...
  normVert = mesh.normal(veH);
```







Using face normals



Using vertex normals

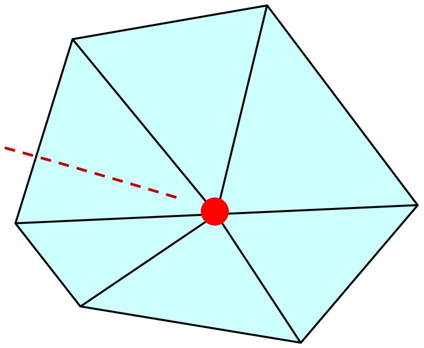
OpenMesh: Circulators

Circulators are used to traverse the local neighbourhood of a mesh item. Example,

- Find all faces around a given vertex:

Query V→F

```
MyMesh::VertexFaceIter vfit;

for(vfit = mesh.vf_iter(veH); vfit;  vfit++)
{
    faH = *vfit;
        ...
}
```



Note: `vfit` returns false when all faces around the vertex have been visited. You may also use the function `vfit.is_valid()` to know if all faces around the vertex have been enumerated.

OpenMesh: Circulators

	Circulator	
All one-ring neighours of a vertex:	VertexVertexIter	V→V
All outgoing halfedges from a vertex:	VertexOHalfedgeIter	V→E
All incident halfedges on a vertex:	VertexIHalfedgeIter	
All vertices of a given face:	FaceVertexIter	F→V
All halfedges belonging to a face:	FaceHalfedgeIter	F→E
All adjacent faces of a given face:	FaceFaceIter	F→F

Ref:

https://www.graphics.rwth-aachen.de/media/openmesh_static/Documentations/OpenMesh-4.0-Documentation/a00026.html

Rendering a Mesh using Circulators

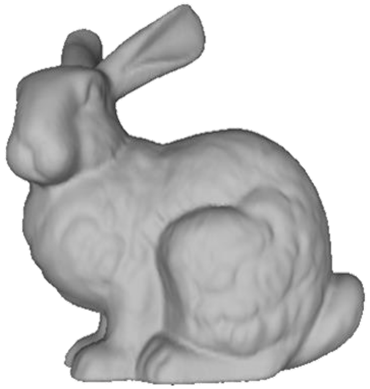
```
for (fit = mesh.faces_begin(); fit != mesh.faces_end(); fit++)
{
        feH = *fit;                     //Get face handle
        col = mesh.color(feH);          //Get face color
        glColor3ubv(col.data());
        norm = mesh.normal(feH);        //Get face normal
        glNormal3fv(norm.data());

        nvert = mesh.valence(feH); //Find polygon type
        if  (nvert == 3)     glBegin(GL_TRIANGLES);
        else if (nvert == 4) glBegin(GL_QUADS);
        else continue;
        for(fvit = mesh.fv_iter(feH); fvit.is_valid(); fvit++)
        {
                p = mesh.point(*fvit);
                glVertex3fv(p.data());
        }
        glEnd();
}
```

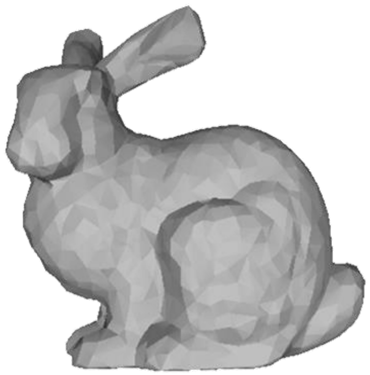
Face Iterator

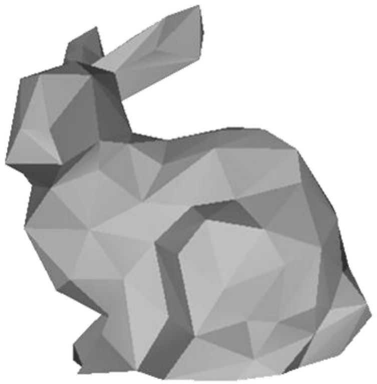
Face-Vertex Iterator

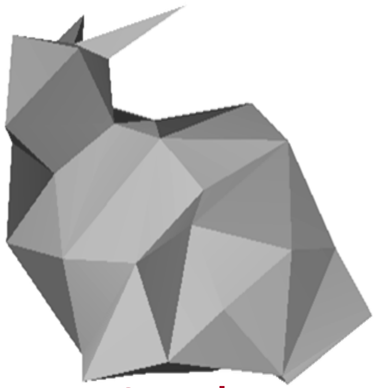
Mesh Processing: Applications

Mesh Simplification


69451 polys


2502 polys

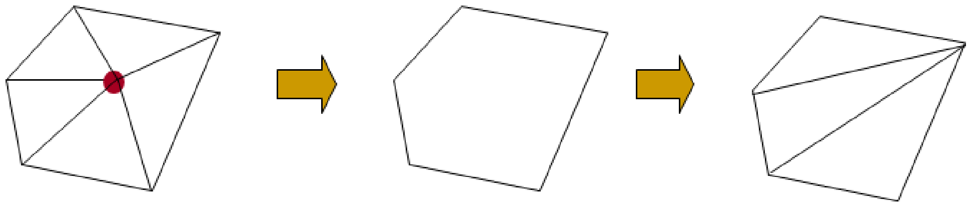

251 polys


76 polys

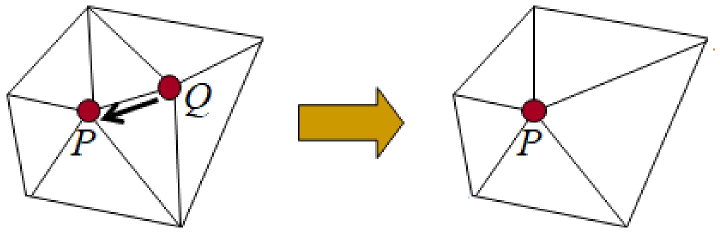
- Commonly used for creating level-of-detail representations.
- Primary goals:
  - Geometrical shape characteristics must be preserved.
  - Topology must not be significantly altered.

Mesh Simplification

- Vertex Decimation Algorithm
  - Select a vertex from the mesh based on an error metric (see next slide).
  - Remove the vertex with incident edges, and re-triangulate the resulting hole.



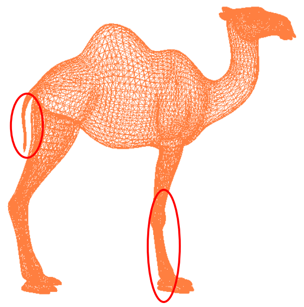
- Edge Collapse Algorithm
  - Move a vertex towards its adjacent vertex, collapsing an edge



Error Metric

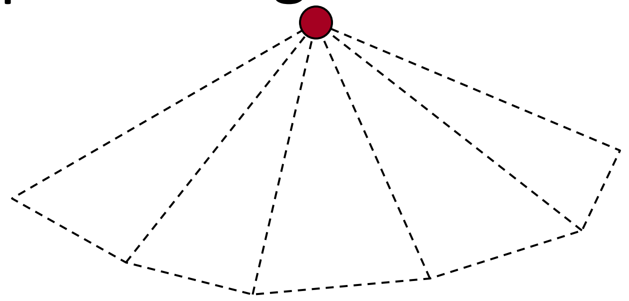
- We require an error metric that measures the amount of error introduced in the simplified mesh at each step.
  - Cost function: Heuristic based on **curvature**, **planarity**, **area** etc.
  - Selection: Modify the mesh where the cost function is minimum.
- Vertices can be flagged as locked, so that they will not be touched by the simplification algorithm:

```
mesh.status(veH).set_locked(true);
```

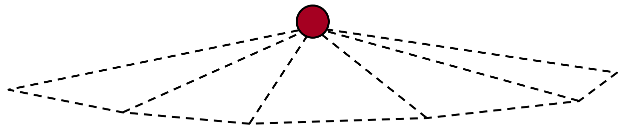


Mesh Planarity Metric

The mesh planarity metric measures how closely a mesh surface segment around a vertex approximates a planar surface. Vertices are commonly deleted from nearly planar regions.

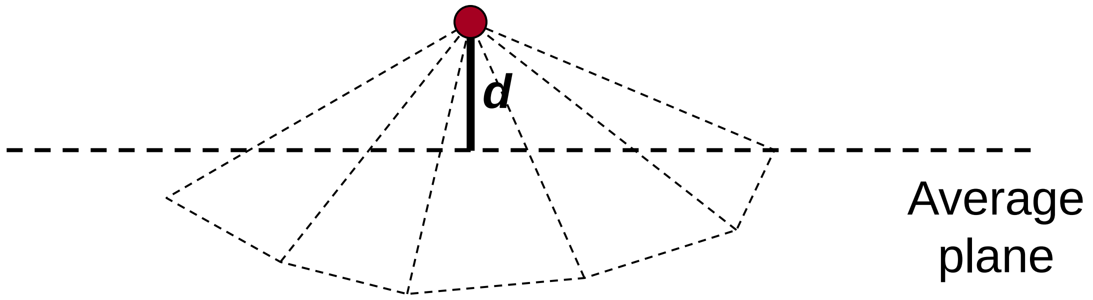


High cost



Low cost
(Nearly planar)

The distance *d* of the vertex from an average plane is used as the value of this metric.



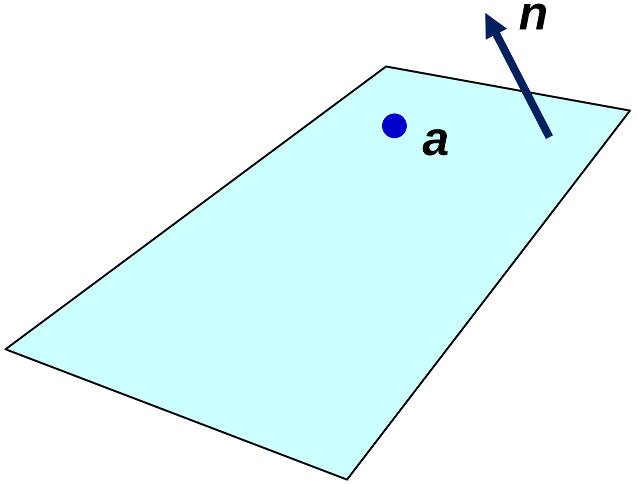
Equation of a Plane

The plane passing through a point ***a*** = (x_a, y_a, z_a) and having a unit normal vector ***n*** = (x_n, y_n, z_n) is given by the equation:

$$(\boldsymbol{p}-\boldsymbol{a})\bullet\boldsymbol{n} = 0,$$

where ***p*** = (x, y, z) is any point on the plane.

The above equation can be expanded as a linear equation in x, y, z:



$$(x - x_a)\, x_n + (y - y_a)\, y_n + (z - z_a)\, z_n = 0$$

This equation simplifies to

$$x_n x + y_n y + z_n z + C = 0$$

Signed Distance from a Plane

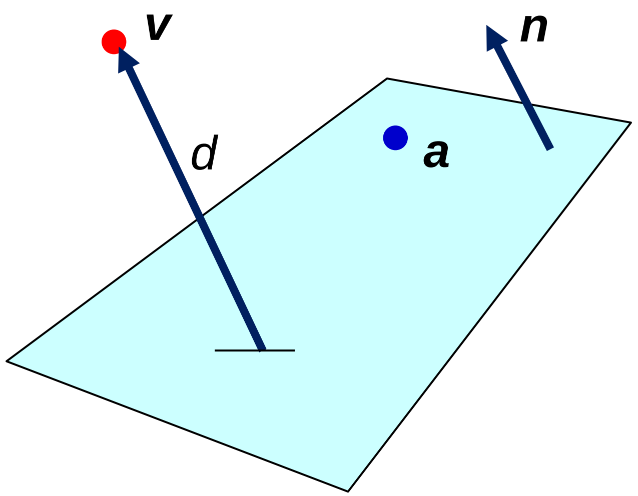
The signed distance of any vertex ***v*** = (x_v, y_v, z_v) from the plane is given by

$d = (\boldsymbol{v} - \boldsymbol{a}) \bullet \boldsymbol{n}$

where, ***n*** is a unit vector.

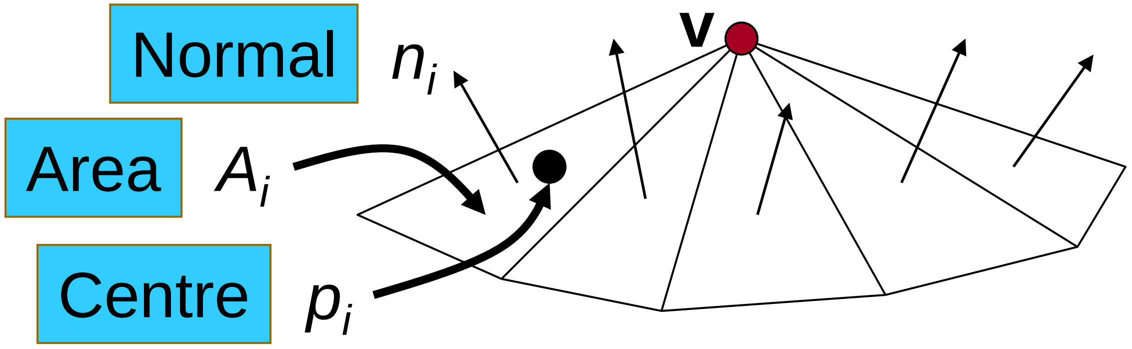
$$d = x_n x_v + y_n y_v + z_n z_v + C$$

- If the vertex ***v*** lies on the plane, *d*=0.
- *d* > 0 if ***v*** is an out-of-plane point on the side given by ***n***
- *d* < 0 if ***v*** is an out-of-plane point on the side given by –***n***



Computing the Planarity Metric

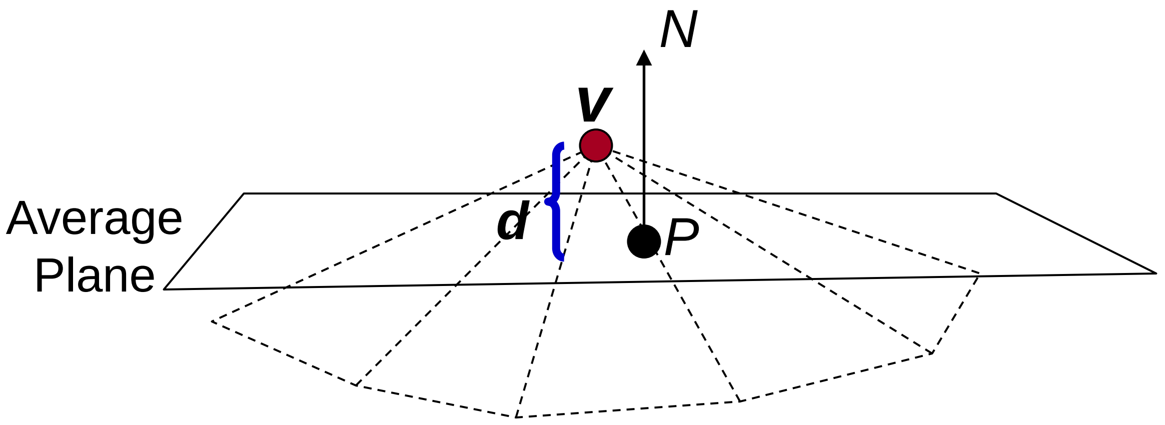
- For each vertex ***v*** of the mesh, obtain the set of triangles sharing the vertex (using a vertex-face iterator)
- Obtain the vertices of each face (using a face-vertex iterator). Using the vertices, compute the centre and normal vector.
- Take the weighted average of the centres of the triangles to get a point *P*, and the weighted average of the normal vectors to get a vector *N*. The point *P* and the normal vector *N* form an <u>average plane.</u>



Area-weighted average normal

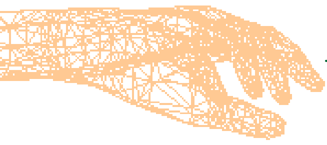
$$N = \frac{\sum_i A_i n_i}{\sum_i A_i} \qquad \hat{N} = \frac{N}{|N|} \qquad P = \frac{\sum_i A_i p_i}{\sum_i A_i}$$

Error Metric for Vertices

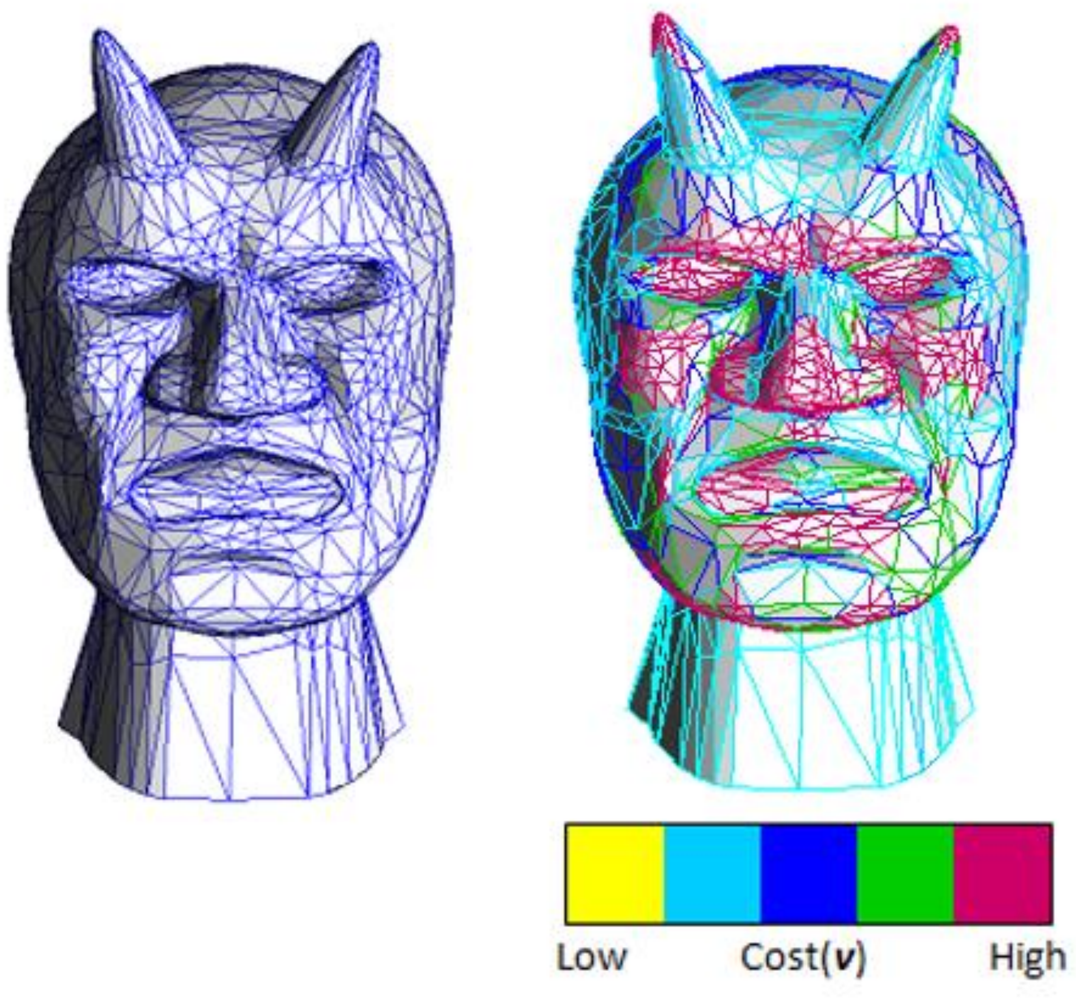


The absolute value of the signed distance of the vertex ***v*** from the **average plane** can be used as an error metric representing the local curvature of the surface.

Cost(***v***) = abs(*d*)

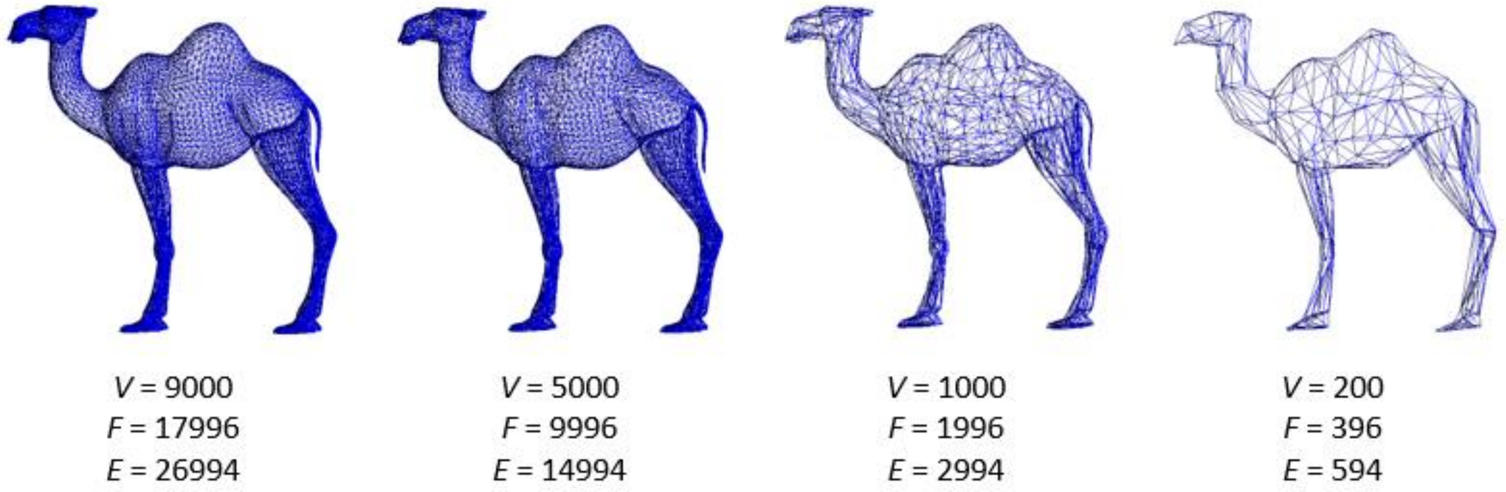


Planarity Metric: Visualization

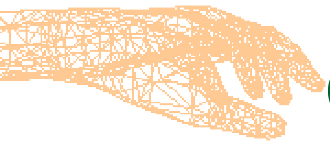


Vertex Decimation: Example

```
typedef OpenMesh::TriMesh_ArrayKernelT<>  MyMesh;
typedef OpenMesh::Decimater::DecimaterT<MyMesh>  MyDecimater;
MyMesh mesh;

OpenMesh::IO::read_mesh(mesh, "Camel.off")
decimater.add(hModQuadric);
decimater.initialize();
...
decimater.decimate_to(tverts[iter]);    //{5000, 1000, 200}
mesh.garbage_collection();
```

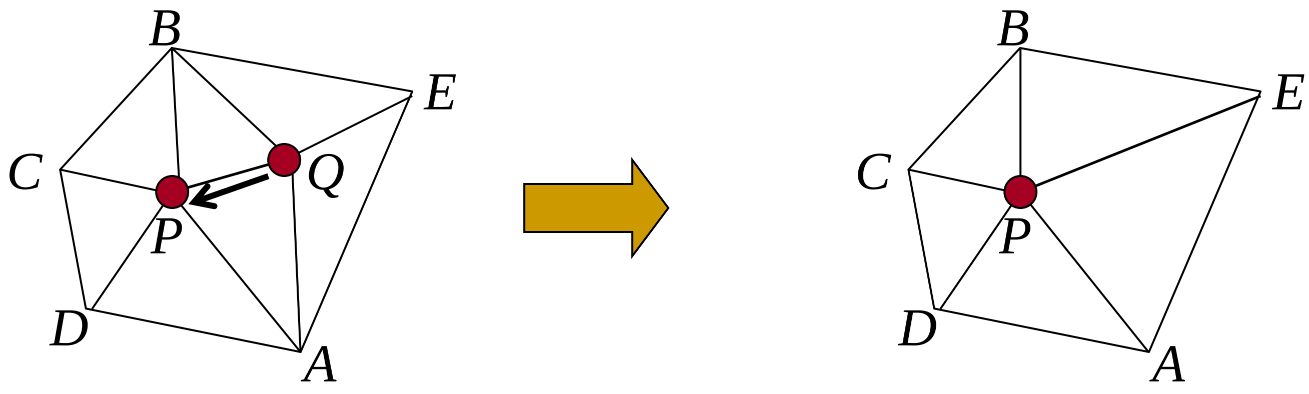


$V = 9000$	$V = 5000$	$V = 1000$	$V = 200$
$F = 17996$	$F = 9996$	$F = 1996$	$F = 396$
$E = 26994$	$E = 14994$	$E = 2994$	$E = 594$



Collapsing Edges

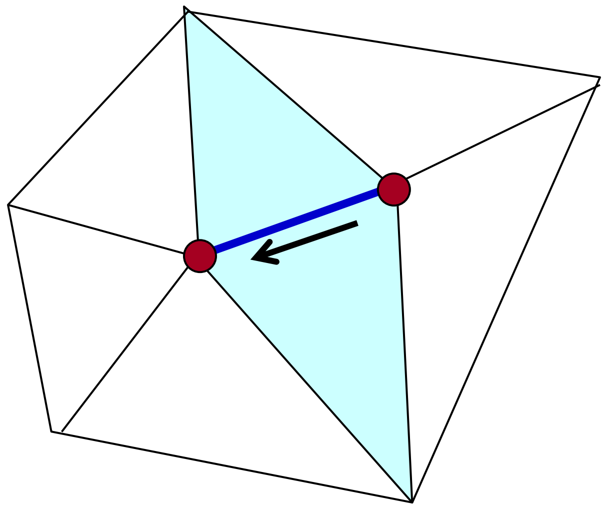
- Collapsing a halfedge moves its “from_vertex” (*Q*) to its “to_vertex” (*P*)
- An edge collapse operation reduces the number of vertices by one, and the number of triangles by 2.
- OpenMesh function: `mesh.collapse(heH);`



Note: Edges AQ, BQ, PQ are deleted. Triangles BPQ, APQ are also deleted.

Error Metric for Edges

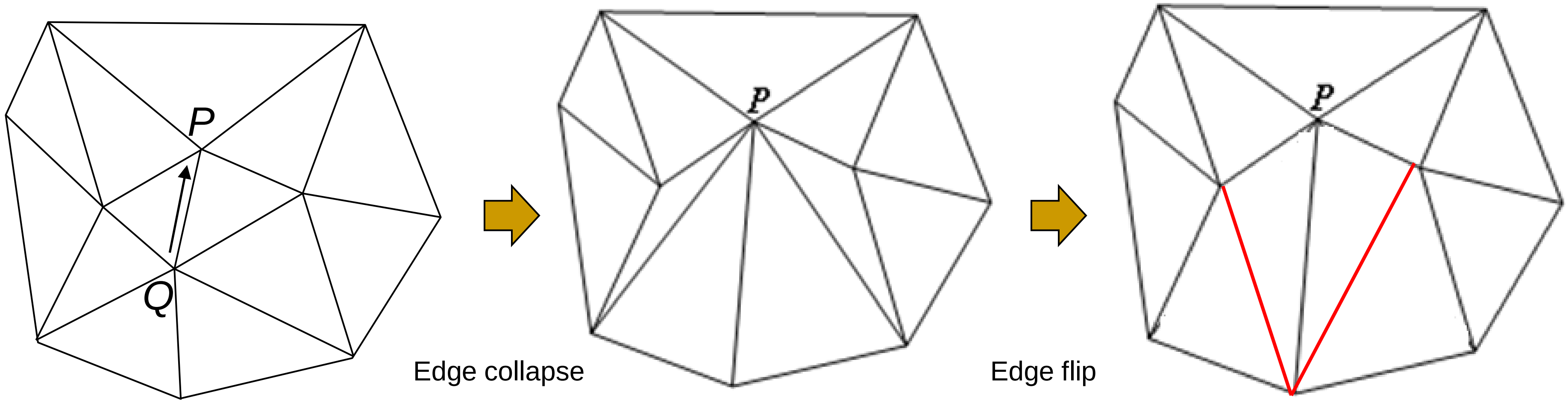
- An edge may be collapsed if
  - The dihedral angle between the bordering faces is small
  - The edge is small



- A linear combination of the dihedral angle between the two triangles bordering the edge and the length of the edge can be used as an error metric:

$$\text{Cost}(P, Q) = k_1 \underbrace{\cos^{-1}(\boldsymbol{m}_1 \bullet \boldsymbol{m}_2)}_{\text{Small angle}} + k_2 \underbrace{|P - Q|}_{\text{Small edge length}}$$

Edge Collapse Operation



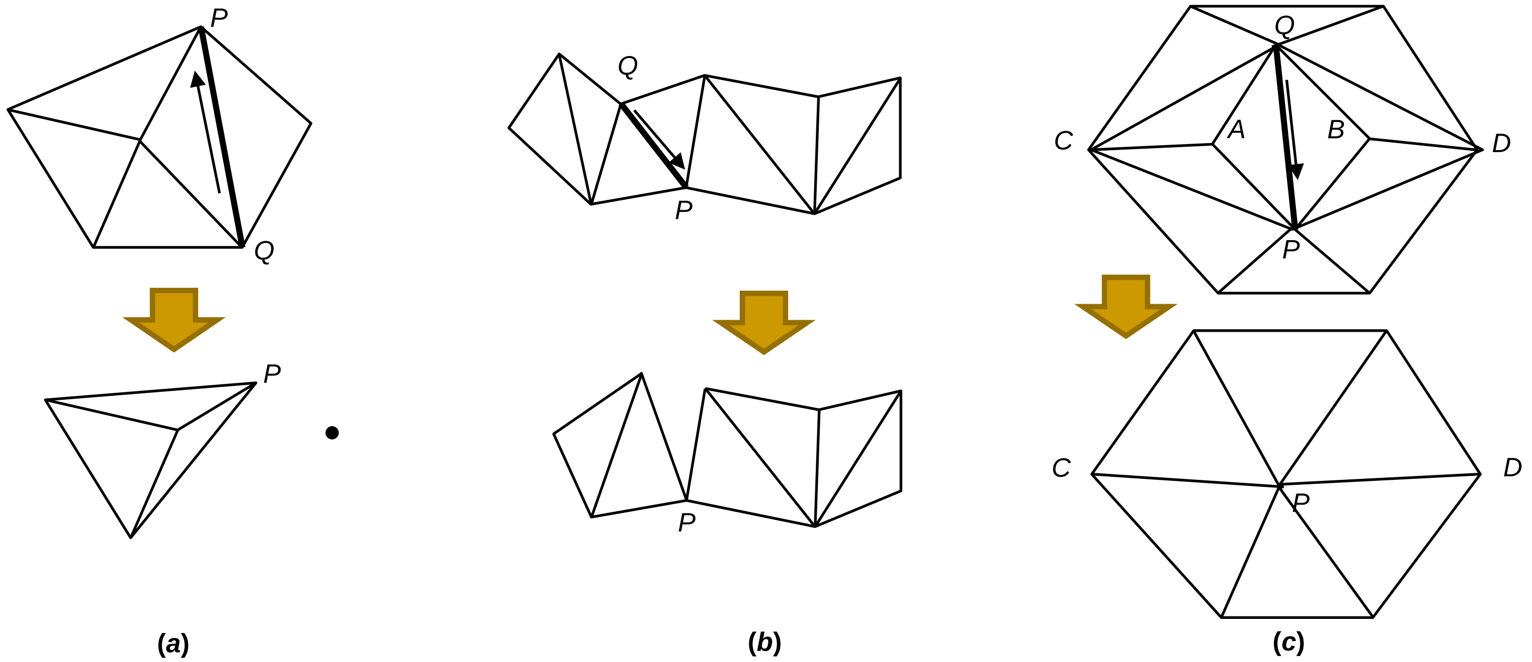
Sliver triangles: An edge collapse operation usually results in several "thin" triangles.

Mesh repair: Some edges will need to be flipped to get an **angle-optimal** triangulation.

OpenMesh function : `mesh.flip(heH);`

Edge Collapse Operation

Invalid edge collapse operations:



(a). The edge or its pair belongs to a triangle whose other two edges are boundary edges. Removing this edge creates an isolated vertex.

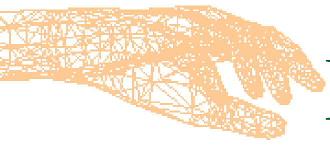
(b). Both vertices of the edge are boundary vertices, but the edge is not a boundary edge.

(c). The intersection of the one-ring neighbourhoods of vertices *P* and *Q* normally contains only the opposite vertices *A*, *B* of an edge. In this case, the intersection contains the points *A*, *B*, *C* and *D*. Removing this edge may create overlapping polygons.

OpenMesh: `mesh.is_collapse_ok();`

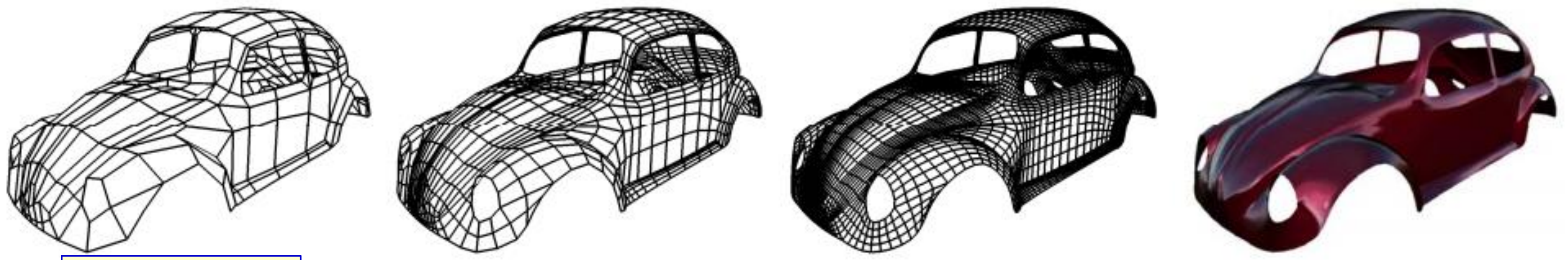
Subdivision Surfaces

- Iteratively subdivides a mesh, creating a smooth surface as the limit of a sequence of successive refinements.
- Extensively used in games and animation design for modelling complex objects with smooth surfaces, that are otherwise difficult to model using parametric curves/surfaces, splines etc.
- Two main classes of subdivision algorithms:
  - Surface **interpolation** methods
  - Surface **approximation** methods



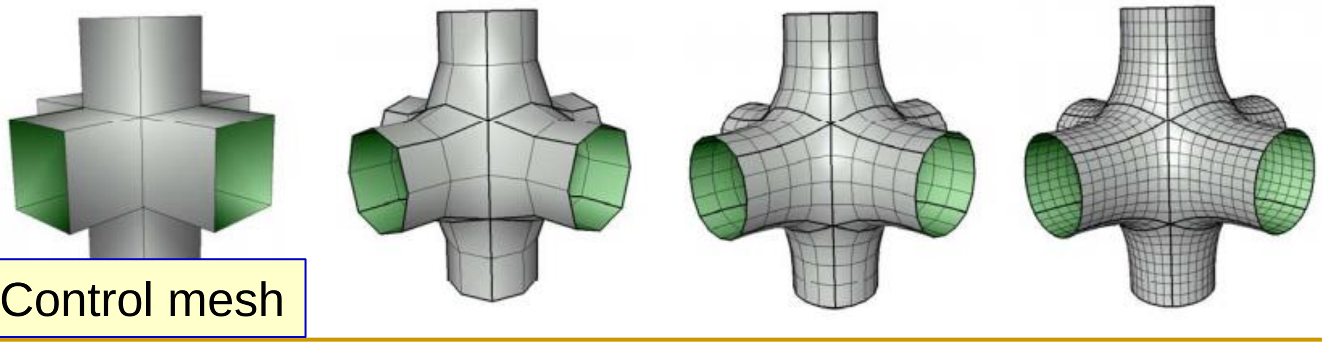
Interpolation Surfaces

- A low polygon mesh is used as the **base mesh** that defines the required shape of the final mesh.
- In each iteration, the mesh is subdivided and the locations of new vertices are computed using a weighted combination of a set of existing neighbouring vertices.
- No vertex is moved once it is computed. In particular, the base mesh's vertices are not altered.

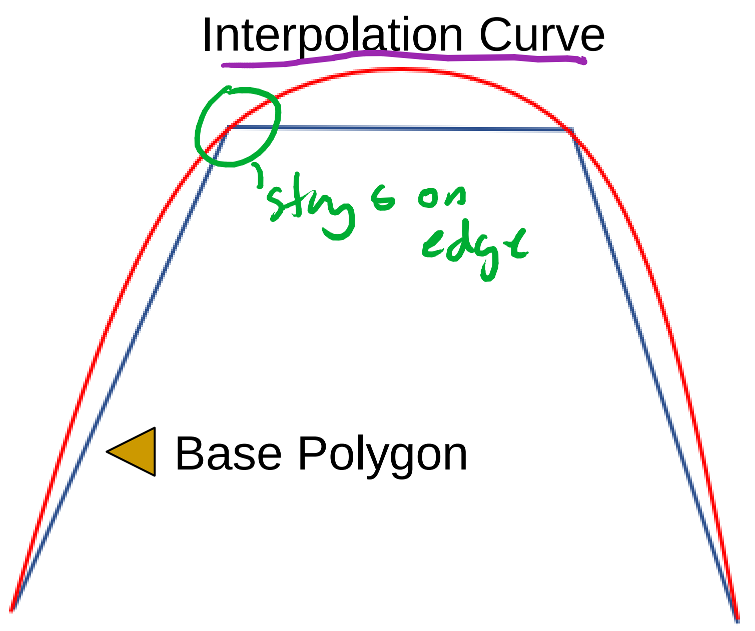


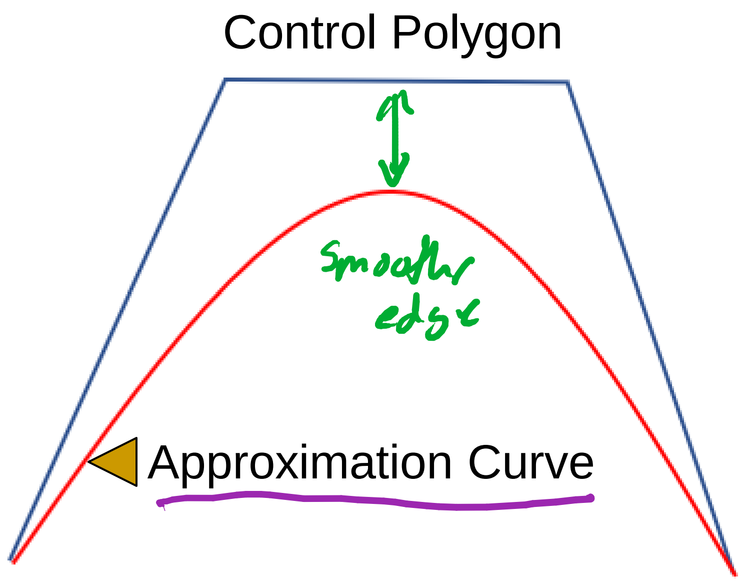
Approximation Surfaces

- A low polygon mesh is used as the **control mesh** for the final mesh. The generated mesh gives only a smooth approximation surface.
- In each iteration, the mesh is subdivided and the new vertices computed using a weighted combination of existing neighbouring vertices.
- Existing vertices are then modified using a local averaging step. The shape of the subdivided surface tends to a limiting surface.

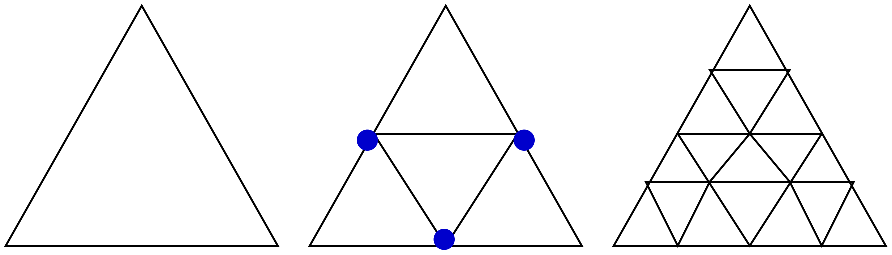


Interpolation vs Approximation (2D)

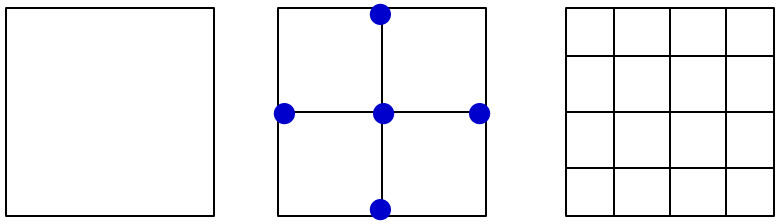




Mesh Subdivision



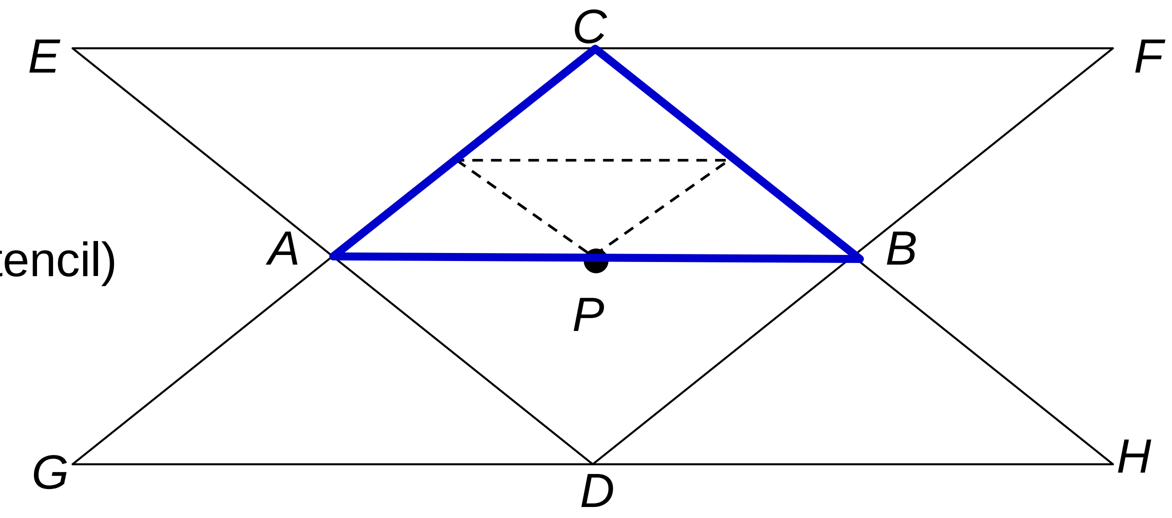
- Triangle Mesh Subdivision:
  - Bisects every edge by inserting a new vertex between every pair of adjacent vertices, increasing the number of triangles by a factor of 4 in each step.
  - Creates vertices of valence 6.
- Quad Mesh Subdivision
  - Bisects every edge by inserting a new vertex between every pair of adjacent vertices, and adds a new vertex for each face, increasing the number of quads by a factor of 4 in each step.
  - Creates vertices of valence 4.

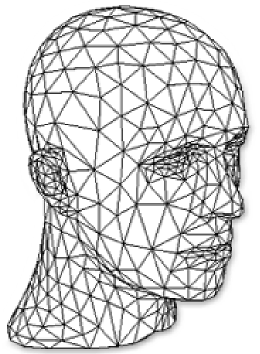


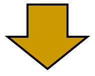
Interpolation: Butterfly Algorithm

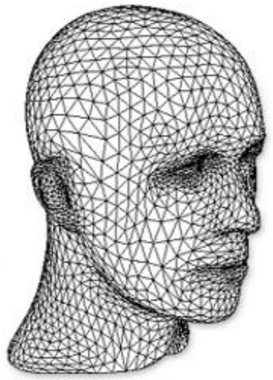
- Uses a subdivision of **triangle meshes**.
- Transforms a new vertex using a convex combination of vertices in the neighbourhood.

(8-Point Stencil)





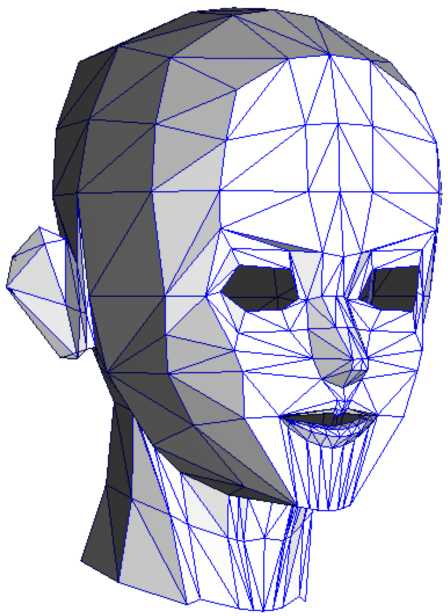


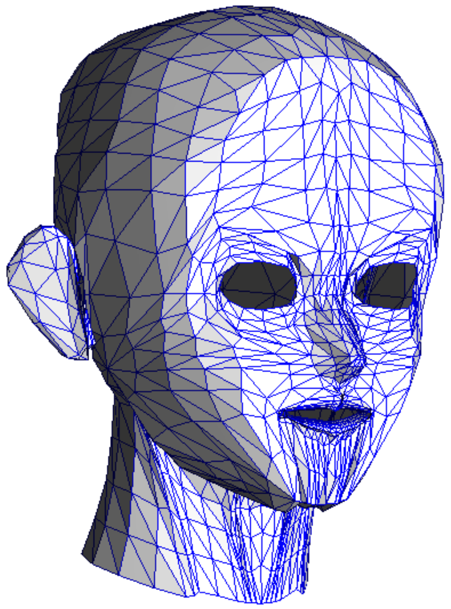


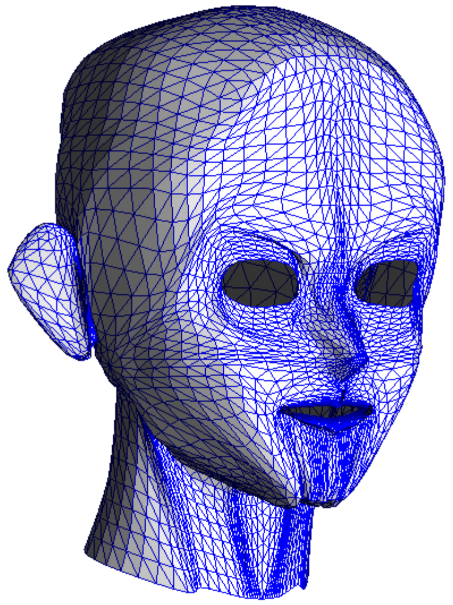
Current triangle: *ABC*. The edge *AB* is subdivided to get the midpoint *P*. This point is transformed using the following formula:

$$P = \left(\frac{1}{2}\right)A + \left(\frac{1}{2}\right)B + \left(\frac{1}{8}\right)C + \left(\frac{1}{8}\right)D - \left(\frac{1}{16}\right)E - \left(\frac{1}{16}\right)F - \left(\frac{1}{16}\right)G - \left(\frac{1}{16}\right)H$$

Butterfly Algorithm: OpenMesh


Base mesh
504 triangles


Iteration 1
2016 triangles

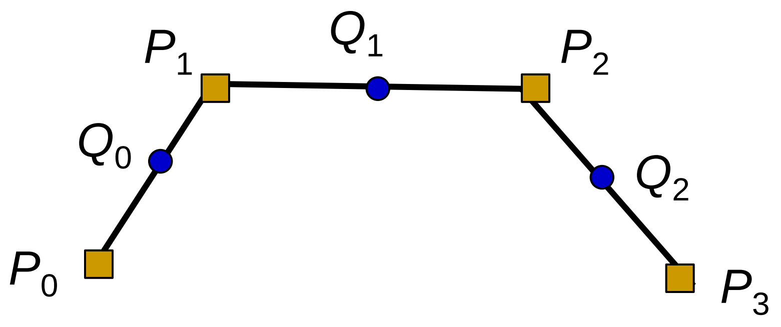

Iteration 2
8064 triangles

```
#include <OpenMesh/Tools/Subdivider/Uniform/ModifiedButterFlyT.hh>
OpenMesh::Subdivider::Uniform::ModifiedButterflyT<MyMesh> butterfly;
...
butterfly.attach(mesh);
butterfly(niter);  //Number of iterations
butterfly.detach();
mesh.update_normals();
```

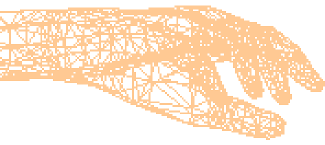
Subdivision Curves For Approximation

- An iterative refinement of a control polygon (in 2D) can be made to converge to a parametric curve.
- Subdivision is a 2-step process
  - Topological split: New points are added as shown on Slide 57, and their positions updated using neighbouring vertices (similar to the Butterfly Algorithm)
  - Local Averaging/Smoothing: Existing points are also transformed using their current positions and the locations of their closest new neighbours.

Step 1: Computing new vertices Q_i

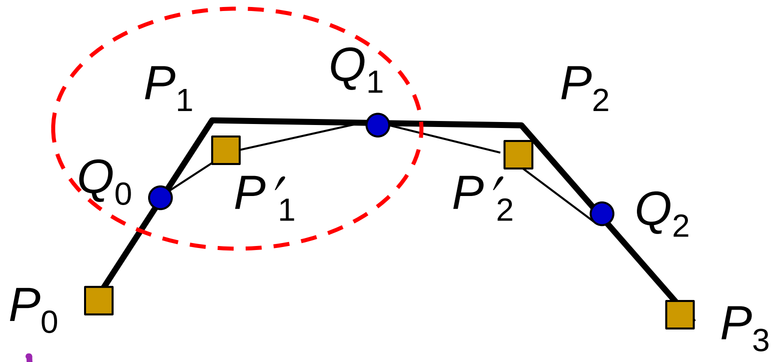


$$Q_0 = (P_0 + P_1)/2$$
$$Q_1 = (P_1 + P_2)/2$$
$$Q_2 = (P_2 + P_3)/2$$



Subdivision Curves for Approximation

Step 2:
Shifting existing points.
(Local averaging)



assigning weights to limiting curve

$$P'_1 = \left(\frac{1}{4}\right)Q_0 + \left(\frac{1}{2}\right)P_1 + \left(\frac{1}{4}\right)Q_1$$

$$P'_2 = \left(\frac{1}{4}\right)Q_1 + \left(\frac{1}{2}\right)P_2 + \left(\frac{1}{4}\right)Q_2$$

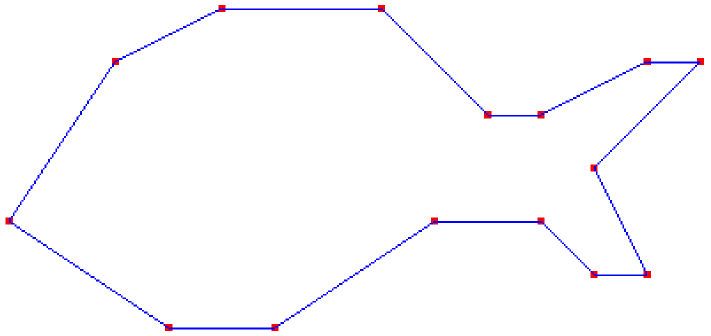
Smoothing Equation

$$\Rightarrow$$

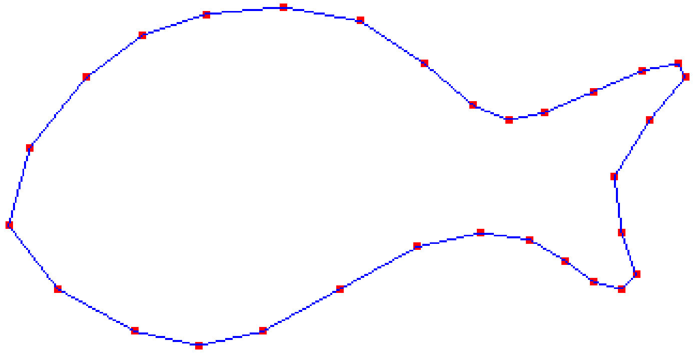
$$P'_1 = \left(\frac{1}{8}\right)P_0 + \left(\frac{6}{8}\right)P_1 + \left(\frac{1}{8}\right)P_2$$

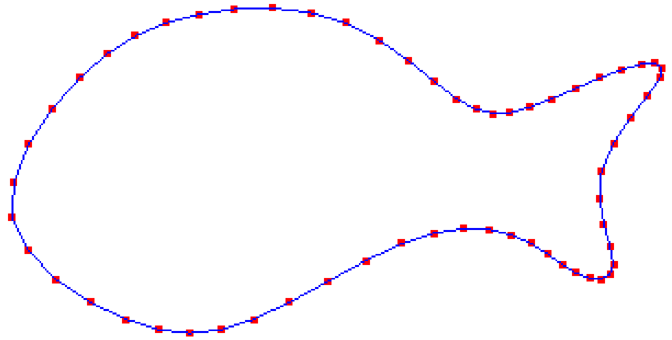
$$P'_2 = \left(\frac{1}{8}\right)P_1 + \left(\frac{6}{8}\right)P_2 + \left(\frac{1}{8}\right)P_3$$

Subdivision Curves for Approximation (2D)


15 points

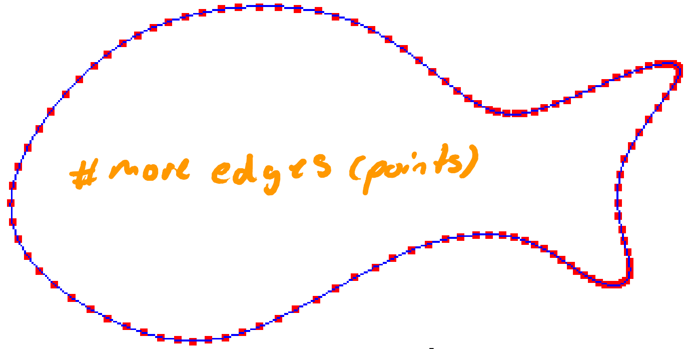
Control polygon


30 points


60 points

$\longleftrightarrow$

Not that much difference


more edges (points)

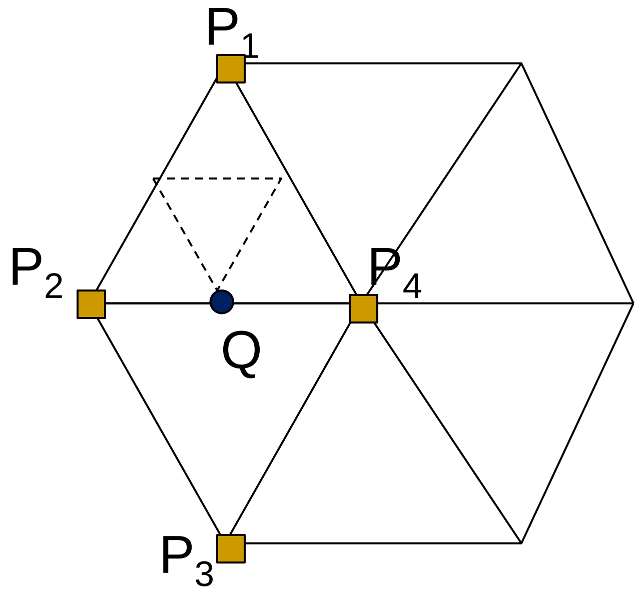
120 points

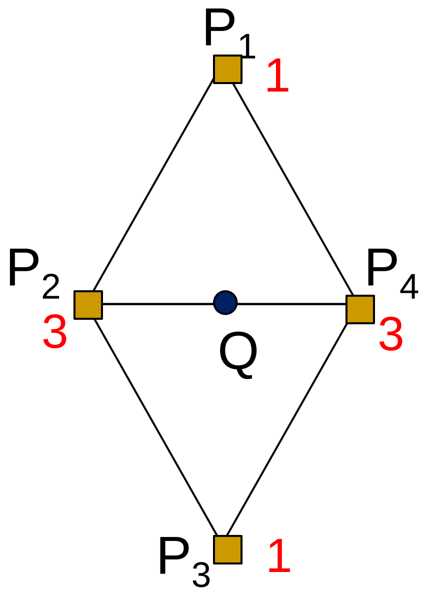
Limiting Curve

Charles-Loop Subdivision

- Extension of the previous method for a triangle mesh.
- (Step 1: Computing new points): A new point is added on every edge, and their positions computed using a 4-point stencil.

Step 1: Computing new points

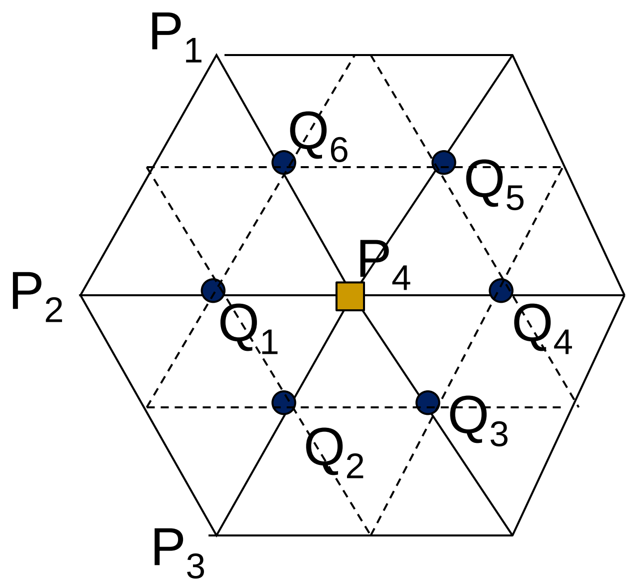




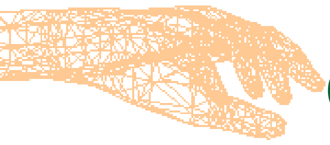
$$\boxed{Q = \frac{P_1 + 3P_2 + P_3 + 3P_4}{8}}$$

Charles-Loop Subdivision

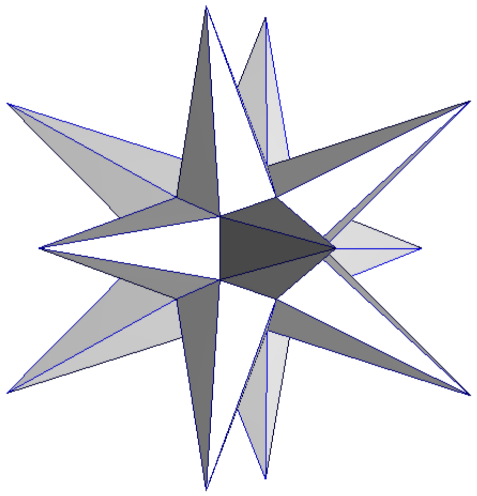
- (Step 2: Local averaging): Existing points are transformed based on their current position and the locations of their closest new neighbours.

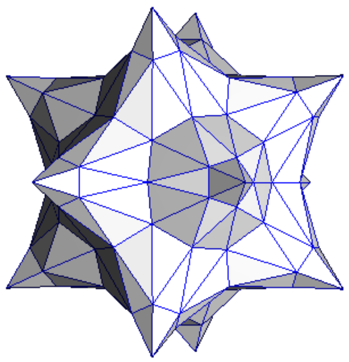


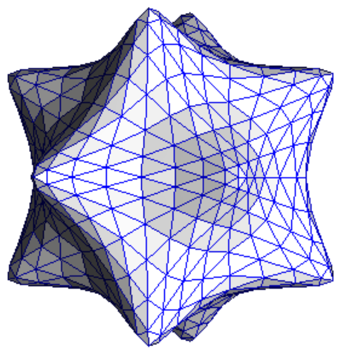
$$P_4' = \left(\frac{1}{10}\right)(Q_1 + Q_2 + Q_3 + Q_4 + Q_5 + Q_6) + \left(\frac{4}{10}\right)P_4$$

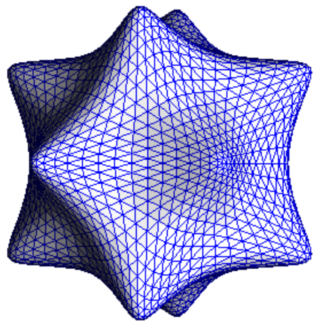


Chales-Loop Subdivision: OpenMesh


Original mesh
60 triangles


Iteration 1
240 triangles


Iteration 2
960 triangles

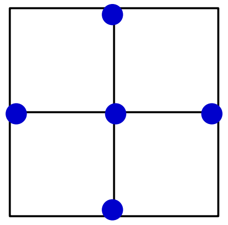

Iteration 3
3840 triangles

Limiting Surface

```
#include <OpenMesh/Tools/Subdivider/Uniform/LoopT.hh>
OpenMesh::Subdivider::Uniform::LoopT<MyMesh> loop;
...
loop.attach(mesh);
loop(niter);  //Number of iterations
loop.detach();
mesh.update_normals();
```

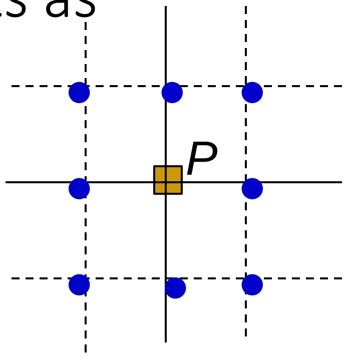
Catmull-Clark Subdivision

- An approximation method suitable for <u>quad meshes</u>
- Step 1: Computing new points:
  - Add a new face point at the centre of each face.
  - For each edge, add a new edge point by taking the average of the two end points and the new adjacent face points.



- Step 2: Local averaging:
  - Move existing vertices (*P*) using neighboring points as follows:

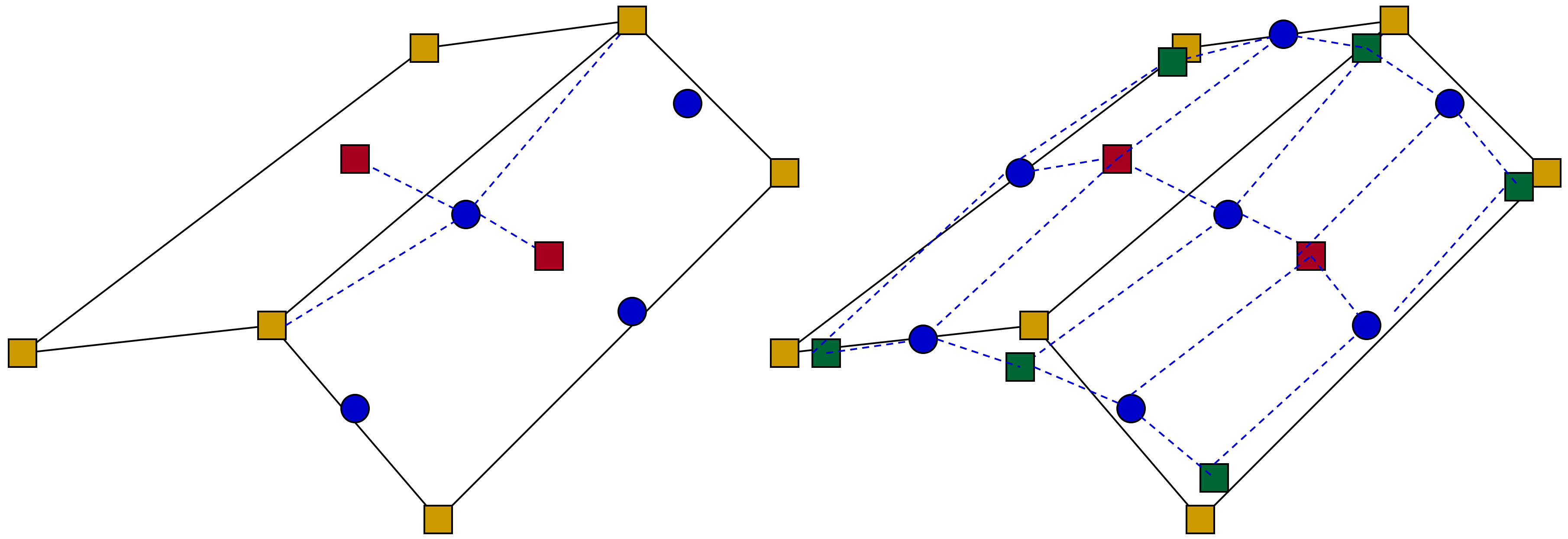
$$\frac{1}{n}\left(F+2R+(n-3)P\right)$$



    - *F*: Average of new face points surrounding *P*.
    - *R*: Average of midpoints of edges through *P*.
    - *n*: Number of edges that share the old vertex *P*.

Catmull-Clark Subdivision

smoother surface



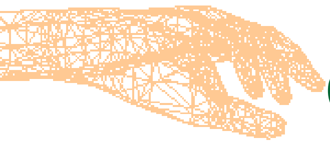
🟨 Original Vertex

🟥 New Face Point

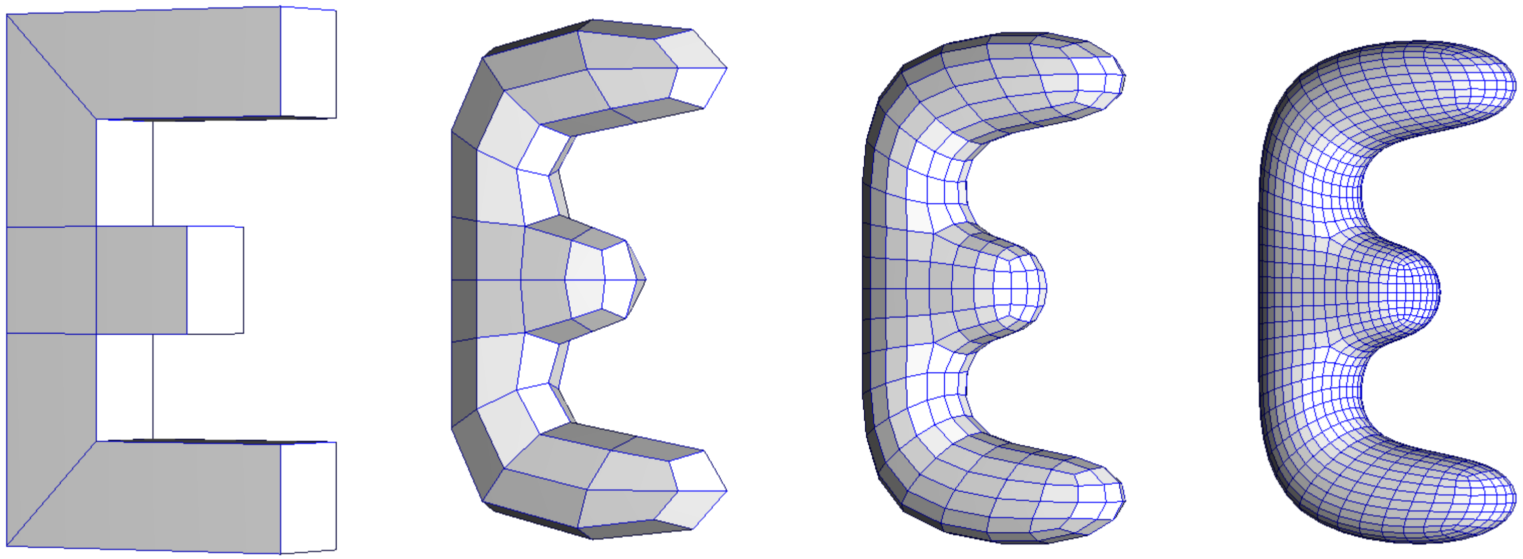
● New Edge Point

🟩 Modified Vertex

- - - - - Subdivided Mesh



Catmull-Clark Subdivision: OpenMesh



Original mesh	Iteration 1	Iteration 2	Iteration 3
26 quads	104 quads	416 quads	1664 quads

```
#include <OpenMesh/Tools/Subdivider/Uniform/ CatmullClarkT.hh>
OpenMesh::Subdivider::Uniform:: :CatmullClarkT<MyMesh> catmull;
...
catmull.attach(mesh);
catmull(niter);  //Number of iterations
catmull.detach();
mesh.update_normals();
```

Summary

- Mesh processing is fun!
- Many complex mesh shapes can be created using subdivision tools.
- Mesh decimation algorithms are used primarily for creating multiple levels of detail
- OpenMesh is a versatile mesh processing library that can be used for
  - Approximation (Chales-Loop, Catmull-Clark)
  - Interpolation (Buttefly)
  - Decimation (Edge collapse)
  - Conversion (Read-write, Triangulation)
  - Visualization