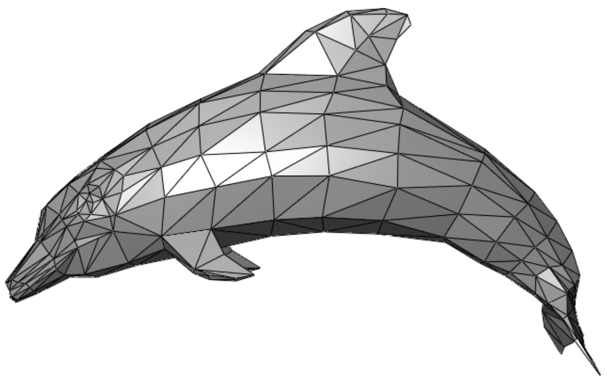
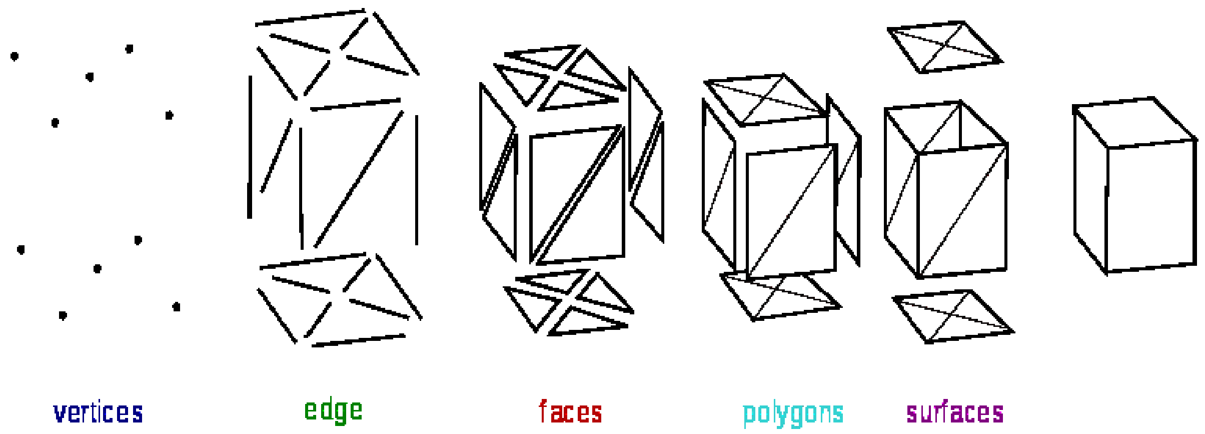
Polygon mesh

A **polygon mesh** or unstructured grid is a collection of vertices, edges and faces that defines the shape of a polyhedral object in 3D computer graphics and solid modeling. The faces usually consist of triangles, quadrilaterals or other simple convex polygons, since this simplifies rendering, but may also be composed of more general concave polygons, or polygons with holes.


Example of a triangle mesh representing a dolphin.

The study of polygon meshes is a large sub-field of computer graphics and geometric modeling. Different representations of polygon meshes are used for different applications and goals. The variety of operations performed on meshes may include Boolean logic, smoothing, simplification, and many others. Network representations, "streaming" and "progressive" meshes, are used to transmit polygon meshes over a network. Volumetric meshes are distinct from polygon meshes in that they explicitly represent both the surface and volume of a structure, while polygon meshes only explicitly represent the surface (the volume is implicit). As polygonal meshes are extensively used in computer graphics, algorithms also exist for ray tracing, collision detection, and rigid-body dynamics of polygon meshes.

Elements of Mesh Modeling



Objects created with polygon meshes must store different types of elements. These include vertices, edges, faces, polygons and surfaces. In many applications, only vertices, edges and either faces or polygons are stored. A renderer may support only 3-sided faces, so polygons must be constructed of many of these, as shown in Figure 1. However, many renderers either support quads and higher-sided polygons, or are able to triangulate polygons to triangles on the fly, making it unnecessary to store a mesh in a triangulated form. Also, in certain applications like head modeling, it is desirable to be able to create both 3- and 4-sided polygons.

A **vertex** is a position along with other information such as color, normal vector and texture coordinates. An **edge** is a connection between two vertices. A **face** is a closed set of edges, in which a *triangle face* has three edges, and a *quad face* has four edges. A **polygon** is a set of faces. In systems that support multi-sided faces, polygons and faces are equivalent. However, most rendering hardware supports only 3- or 4-sided faces, so polygons are represented as multiple faces. Mathematically a polygonal mesh may be considered an unstructured grid, or undirected graph, with addition properties of geometry, shape and topology.

Surfaces, more often called **smoothing groups**, are useful, but not required to group smooth regions. Consider a cylinder with caps, such as a soda can. For smooth shading of the sides, all surface normals must point horizontally away from the center, while the normals of the caps must point in the +/-(0,0,1) directions. Rendered as a single, Phong-shaded surface, the crease vertices would have incorrect normals. Thus, some way of determining where to cease smoothing is needed to group smooth parts of a mesh, just as polygons group 3-sided faces. As an alternative to providing surfaces/smoothing groups, a mesh may contain other data for calculating the same data, such as a splitting angle (polygons with normals above this threshold are either automatically treated as separate smoothing groups or some technique such as splitting or chamfering is automatically applied to the edge between them). Additionally, very high resolution meshes are less subject to issues that would require smoothing groups, as their polygons are so small as to make the need irrelevant. Further, another alternative exists in the possibility of simply detaching the surfaces themselves from the rest of the mesh. Renderers do not attempt to smooth edges across noncontiguous polygons.

Mesh format may or may not define other useful data. **Groups** may be defined which define separate elements of the mesh and are useful for determining separate sub-objects for skeletal animation or separate actors for non-skeletal animation. Generally **materials** will be defined, allowing different portions of the mesh to use different shaders when rendered. Most mesh formats also suppose some form of **UV coordinates** which are a separate 2d representation of the mesh "unfolded" to show what portion of a 2-dimensional texture map to apply to different polygons of the mesh.

Representations

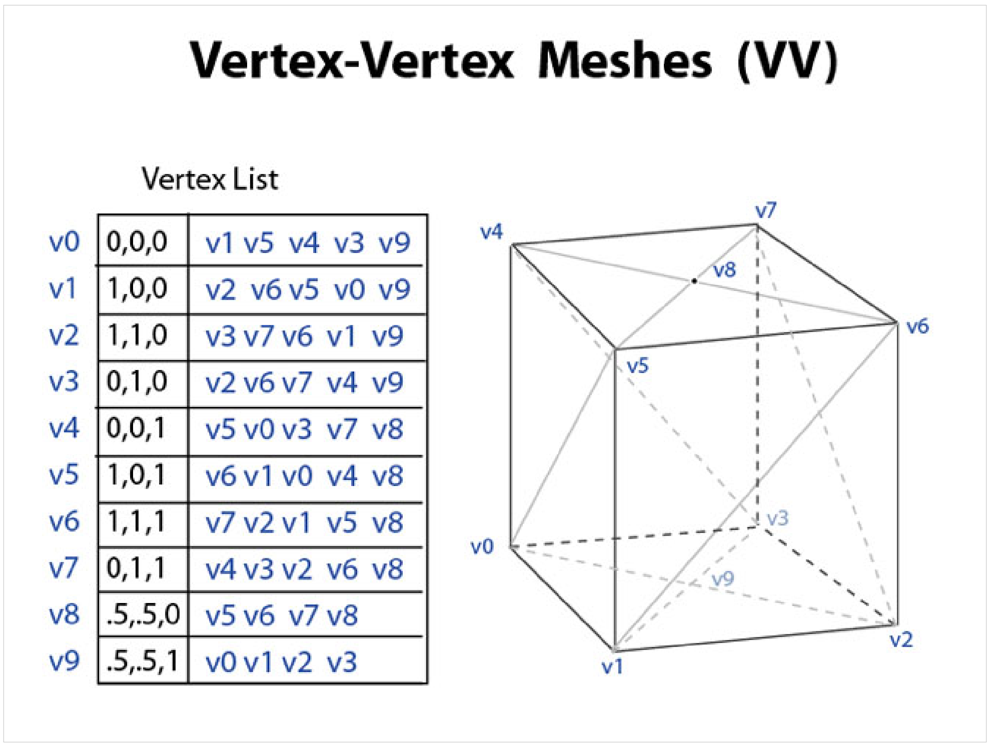
Polygon meshes may be represented in a variety of ways, using different methods to store the vertex, edge and face data. These include:

- Face-vertex meshes: A simple list of vertices, and a set of polygons that point to the vertices it uses.
- Winged-edge meshes, in which each edge points to two vertices, two faces, and the four (clockwise and counterclockwise) edges that touch it. Winged-edge meshes allow constant time traversal of the surface, but with higher storage requirements.
- Half-edge meshes: Similar to winged-edge meshes except that only half the edge traversal information is used.
- Quad-edge meshes, which store edges, half-edges, and vertices without any reference to polygons. The polygons are implicit in the representation, and may be found by traversing the structure. Memory requirements are similar to half-edge meshes.
- Corner-tables, which store vertices in a predefined table, such that traversing the table implicitly defines polygons. This is in essence the "triangle fan" used in hardware graphics rendering. The representation is more compact, and more efficient to retrieve polygons, but operations to change polygons are slow. Furthermore, corner-tables do not represent meshes completely. Multiple corner-tables (triangle fans) are needed to represent most meshes.
- Vertex-vertex meshes: A "VV" mesh represents only vertices, which point to other vertices. Both the edge and face information is implicit in the representation. However, the simplicity of the representation allows for many efficient operations to be performed on meshes.

Each of the representations above have particular advantages and drawbacks, further discussed in Smith (2006).[1]

The choice of the data structure is governed by the application, the performance required, size of the data, and the operations to be performed. For example, it is easier to deal with triangles than general polygons, especially in computational geometry. For certain operations it is necessary to have a fast access to topological information such as edges or neighboring faces; this requires more complex structures such as the winged-edge representation. For hardware rendering, compact, simple structures are needed; thus the corner-table (triangle fan) is commonly incorporated into low-level rendering APIs such as DirectX and OpenGL.

Vertex-vertex meshes



Vertex-Vertex Meshes (VV)

Vertex List

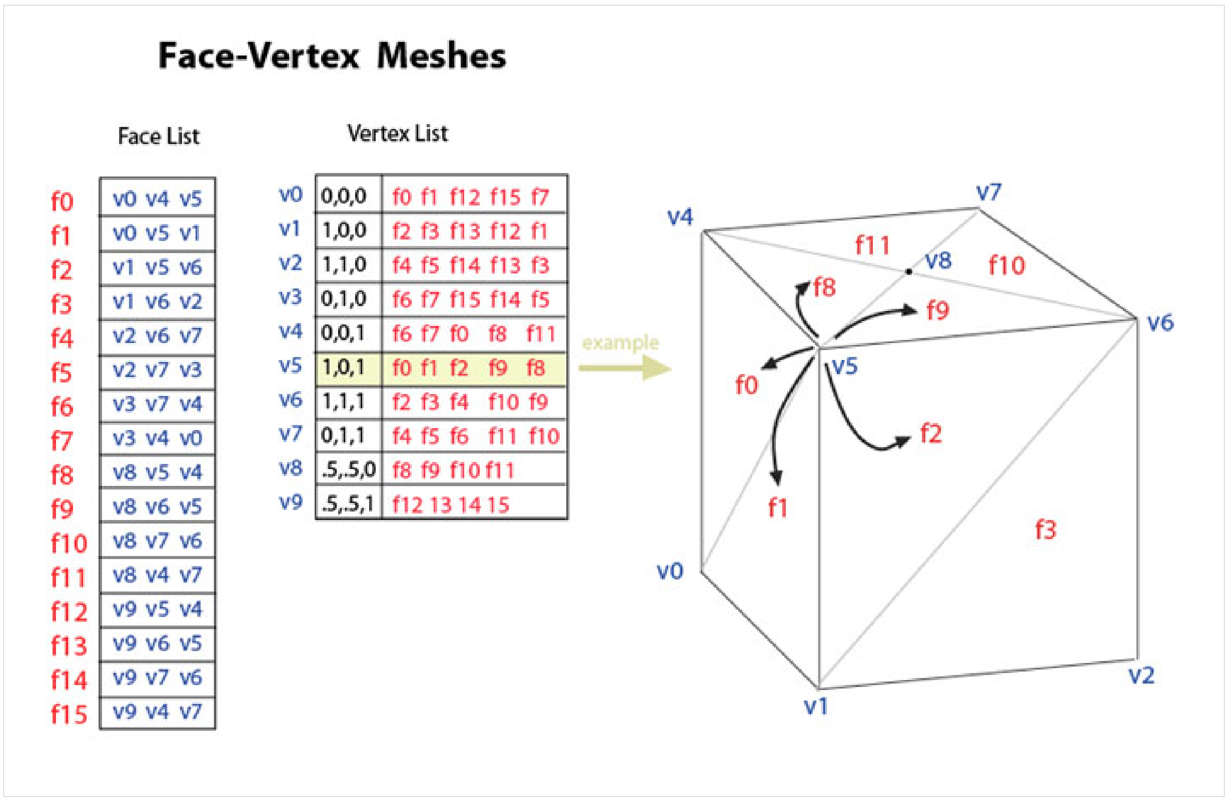
v0	0,0,0	v1 v5 v4 v3 v9
v1	1,0,0	v2 v6 v5 v0 v9
v2	1,1,0	v3 v7 v6 v1 v9
v3	0,1,0	v2 v6 v7 v4 v9
v4	0,0,1	v5 v0 v3 v7 v8
v5	1,0,1	v6 v1 v0 v4 v8
v6	1,1,1	v7 v2 v1 v5 v8
v7	0,1,1	v4 v3 v2 v6 v8
v8	.5,.5,0	v5 v6 v7 v8
v9	.5,.5,1	v0 v1 v2 v3

Vertex-vertex meshes represent an object as a set of vertices connected to other vertices. This is the simplest representation, but not widely used since the face and edge information is implicit. Thus, it is necessary to traverse the data in order to generate a list of faces for rendering. In addition, operations on edges and faces are not easily accomplished.

However, VV meshes benefit from small storage space and efficient morphing of shape. Figure 2 shows the four-sided cylinder example represented using VV meshes. Each vertex indexes its neighboring vertices. Notice that the last two vertices, 8 and 9 at the top and bottom center of the "box-cylinder", have four connected vertices rather than five. A general system must be able to handle an arbitrary number of vertices connected to any given vertex.

For a complete description of VV meshes see Smith (2006).[1]

Face-vertex meshes



Face-Vertex Meshes

Face List

f0	v0 v4 v5
f1	v0 v5 v1
f2	v1 v5 v6
f3	v1 v6 v2
f4	v2 v6 v7
f5	v2 v7 v3
f6	v3 v7 v4
f7	v3 v4 v0
f8	v8 v5 v4
f9	v8 v6 v5
f10	v8 v7 v6
f11	v8 v4 v7
f12	v9 v5 v4
f13	v9 v6 v5
f14	v9 v7 v6
f15	v9 v4 v7

Vertex List

v0	0,0,0	f0 f1 f12 f15 f7
v1	1,0,0	f2 f3 f13 f12 f1
v2	1,1,0	f4 f5 f14 f13 f3
v3	0,1,0	f6 f7 f15 f14 f5
v4	0,0,1	f6 f7 f0 f8 f11
v5	1,0,1	f0 f1 f2 f9 f8
v6	1,1,1	f2 f3 f4 f10 f9
v7	0,1,1	f4 f5 f6 f11 f10
v8	.5,.5,0	f8 f9 f10 f11
v9	.5,.5,1	f12 13 14 15

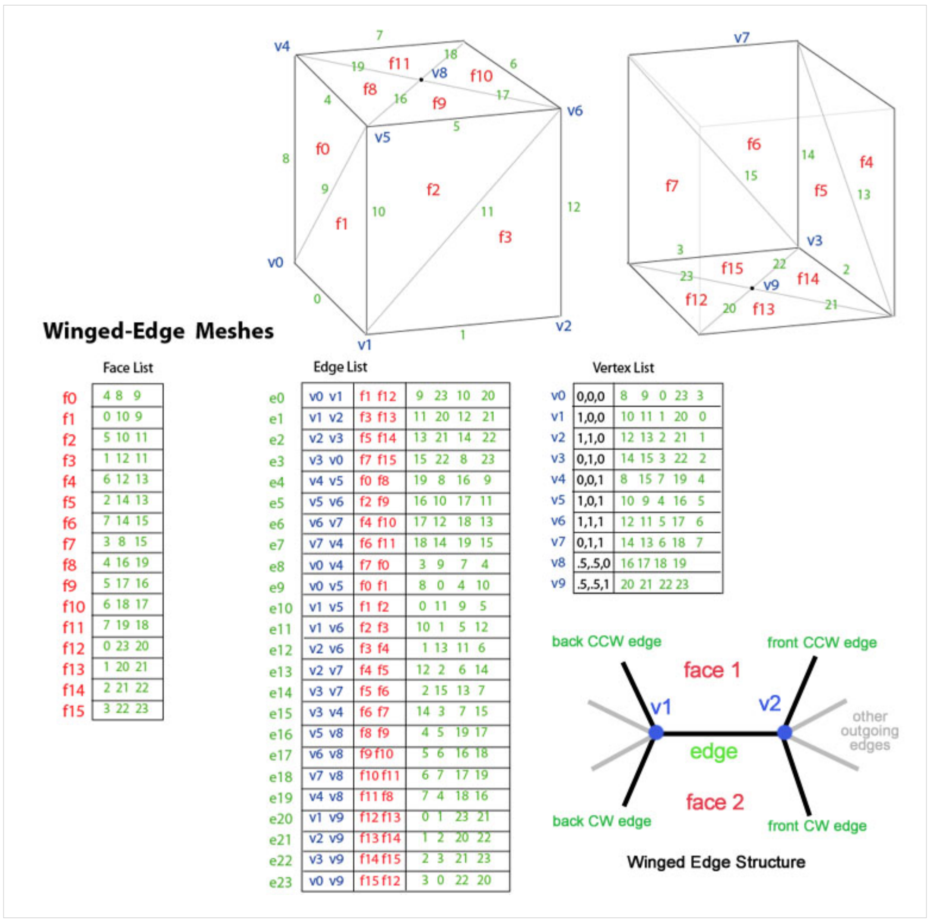
Face-vertex meshes represent an object as a set of faces and a set of vertices. This is the most widely used mesh representation, being the input typically accepted by modern graphics hardware.

Face-vertex meshes improve on VV-mesh for modeling in that they allow explicit lookup of the vertices of a face, and the faces surrounding a vertex. Figure 3 shows the "box-cylinder" example as an FV mesh. Vertex v5 is highlighted to show the faces that surround it. Notice that, in this example, every face is required to have exactly 3 vertices. However, this does not mean every vertex has the same number of surrounding faces.

For rendering, the face list is usually transmitted to the GPU as a set of indices to vertices, and the vertices are sent as position/color/normal structures (in the figure, only position is given). This has the benefit that changes in shape, but not geometry, can be dynamically updated by simply resending the vertex data without updating the face connectivity.

Modeling requires easy traversal of all structures. With face-vertex meshes it is easy to find the vertices of a face. Also, the vertex list contains a list of faces connected to each vertex. Unlike VV meshes, both faces and vertices are explicit, so locating neighboring faces and vertices is constant time. However, the edges are implicit, so a search is still needed to find all the faces surrounding a given face. Other dynamic operations, such as splitting or merging a face, are also difficult with face-vertex meshes.

Winged-edge meshes



Face List

f0	4 8 9
f1	0 10 9
f2	5 10 11
f3	1 12 11
f4	6 12 13
f5	2 14 13
f6	7 14 15
f7	3 8 15
f8	4 16 19
f9	5 17 16
f10	6 18 17
f11	7 19 18
f12	0 23 20
f13	1 20 21
f14	2 21 22
f15	3 22 23

Edge List

e0	v0 v1	f1 f12	9 23 10 20
e1	v1 v2	f3 f13	11 20 12 21
e2	v2 v3	f5 f14	13 21 14 22
e3	v3 v0	f7 f15	15 22 8 23
e4	v4 v5	f0 f8	19 8 16 9
e5	v5 v6	f2 f9	16 10 17 11
e6	v6 v7	f4 f10	17 12 18 13
e7	v7 v4	f6 f11	18 14 19 15
e8	v0 v4	f7 f0	3 9 7 4
e9	v0 v5	f0 f1	8 0 4 10
e10	v1 v5	f1 f2	0 11 9 5
e11	v1 v6	f2 f3	10 1 5 12
e12	v2 v6	f3 f4	1 13 11 6
e13	v2 v7	f4 f5	12 2 6 14
e14	v3 v7	f5 f6	2 15 13 7
e15	v3 v4	f6 f7	14 3 7 15
e16	v5 v8	f8 f9	4 5 19 17
e17	v6 v8	f9 f10	5 6 16 18
e18	v7 v8	f10 f11	6 7 17 19
e19	v4 v8	f11 f8	7 4 18 16
e20	v1 v9	f12 f13	0 1 23 21
e21	v2 v9	f13 f14	1 2 20 22
e22	v3 v9	f14 f15	2 3 21 23
e23	v0 v9	f15 f12	3 0 22 20

Vertex List

v0	0,0,0	8 9 0 23 3
v1	1,0,0	10 11 1 20 0
v2	1,1,0	12 13 2 21 1
v3	0,1,0	14 15 3 22 2
v4	0,0,1	8 15 7 19 4
v5	1,0,1	10 9 4 16 5
v6	1,1,1	12 11 5 17 6
v7	0,1,1	14 13 6 18 7
v8	.5,.5,0	16 17 18 19
v9	.5,.5,1	20 21 22 23

Introduced by Baumgart 1975, **winged-edge meshes** explicitly represent the vertices, faces, and edges of a mesh. This representation is widely used in modeling programs to provide the greatest flexibility in dynamically changing the mesh geometry, because split and merge operations can be done quickly. Their primary drawback is large storage requirements and increased complexity due to maintaining many indices. A good discussion of implementation issues of Winged-edge meshes may be found in the book *Graphics Gems II*.

Winged-edge meshes address the issue of traversing from edge to edge, and providing an ordered set of faces around an edge. For any given edge, the number of outgoing edges may be arbitrary. To simplify this, winged-edge meshes provide only four, the nearest clockwise and counter-clockwise edges at each end. The other edges may be traversed incrementally. The information for each edge therefore resembles a butterfly, hence "winged-edge" meshes. Figure 4 shows the "box-cylinder" as a winged-edge mesh. The total data for an edge consists of 2 vertices (endpoints), 2 faces (on each side), and 4 edges (winged-edge).

Rendering of winged-edge meshes for graphics hardware requires generating a Face index list. This is usually done only when the geometry changes. winged-edge meshes are ideally suited for dynamic geometry, such as subdivision surfaces and interactive modeling, since changes to the mesh can occur locally. Traversal across the mesh, as might be needed for collision detection, can be accomplished efficiently.

See Baumgart (1975) for more details.[2]

Render dynamic meshes

Winged-edge meshes are not the only representation which allows for dynamic changes to geometry. A new representation which combines winged-edge meshes and face-vertex meshes is the **render dynamic mesh**, which explicitly stores the vertices of a face (like FV meshes), the faces of a vertex (like FV meshes), and the faces and vertices of an edge (like winged-edge).

Render dynamic meshes require slightly less storage space than standard winged-edge meshes, and can be directly rendered by graphics hardware since the face list contains an index of vertices. In addition, traversal from vertex to face is implicit (constant time), as is from face to vertex. RD meshes do not require the four outgoing edges since these can be found by traversing from edge to face, then face to neighboring edge.

RD meshes benefit from the features of winged-edge meshes by allowing for geometry to be dynamically updated.

See Tobler & Maierhofer (WSCG 2006) for more details.[3]

Summary of mesh representation

Operation		Vertex-vertex	Face-vertex	Winged-edge	Render dynamic
V-V	All vertices around vertex	Explicit	V → f1, f2, f3, ... → v1, v2, v3, ...	V → e1, e2, e3, ... → v1, v2, v3, ...	V → e1, e2, e3, ... → v1, v2, v3, ...
E-F	All edges of a face	F(a,b,c) → {a,b}, {b,c}, {a,c}	F → {a,b}, {b,c}, {a,c}	Explicit	Explicit
V-F	All vertices of a face	F(a,b,c) → {a,b,c}	Explicit	F → e1, e2, e3 → a, b, c	Explicit
F-V	All faces around a vertex	Pair search	Explicit	V → e1, e2, e3 → f1, f2, f3, ...	Explicit
E-V	All edges around a vertex	V → {v,v1}, {v,v2}, {v,v3}, ...	V → f1, f2, f3, ... → v1, v2, v3, ...	Explicit	Explicit
F-E	Both faces of an edge	List compare	List compare	Explicit	Explicit
V-E	Both vertices of an edge	E(a,b) → {a,b}	E(a,b) → {a,b}	Explicit	Explicit
Flook	Find face with given vertices	F(a,b,c) → {a,b,c}	Set intersection of v1,v2,v3	Set intersection of v1,v2,v3	Set intersection of v1,v2,v3
Storage size		V*avg(V,V)	3F + V*avg(F,V)	3F + 8E + V*avg(E,V)	6F + 4E + V*avg(E,V)
		Example with 10 vertices, 16 faces, 24 edges:			
		10 * 5 = 50	3*16 + 10*5 = 98	3*16 + 8*24 + 10*5 = 290	6*16 + 4*24 + 10*5 = 242

Figure 6: summary of mesh representation operations

In the above table, *explicit* indicates that the operation can be performed in constant time, as the data is directly stored; *list compare* indicates that a list comparison between two lists must be performed to accomplish the operation; and *pair search* indicates a search must be done on two indices. The notation *avg(V,V)* means the average number of vertices connected to a given vertex; *avg(E,V)* means the average number of edges connected to a given vertex, and *avg(F,V)* is the average number of faces connected to a given vertex.

The notation "V → f1, f2, f3, ... → v1, v2, v3, ..." describes that a traversal across multiple elements is required to perform the operation. For example, to get "all vertices around a given vertex V" using the face-vertex mesh, it is necessary to first find the faces around the given vertex V using the vertex list. Then, from those faces, use the face list to find the vertices around them. Notice that winged-edge meshes explicitly store nearly all information, and other operations always traverse to the edge first to get additional info. Vertex-vertex meshes are the only representation that explicitly stores the neighboring vertices of a given vertex.

As the mesh representations become more complex (from left to right in the summary), the amount of information explicitly stored increases. This gives more direct, constant time, access to traversal and topology of various elements but at the cost of increased overhead and space in maintaining indices properly.

Figure 7 shows the connectivity information for each of the four technique described in this article. Other representations also exist, such as half-edge and corner tables. These are all variants of how vertices, faces and edges index one another.

As a general rule, face-vertex meshes are used whenever an object must be rendered on graphics hardware that does not change geometry (connectivity), but may deform or morph shape (vertex positions) such as real-time rendering of static or morphing objects. Winged-edge or render dynamic meshes are used when the geometry changes, such as in interactive modeling packages or for computing subdivison surfaces. Vertex-vertex meshes are ideal for efficient, complex changes in geometry or topology so long as hardware rendering is not of concern.

Other representations

Streaming meshes store faces in an ordered, yet independent, way so that the mesh can be transmitted in pieces. The order of faces may be spatial, spectral, or based on other properties of the mesh. Streaming meshes allow a very large mesh to be rendered even while it is still being loaded.

Progressive meshes transmit the vertex and face data with increasing levels of detail. Unlike *streaming meshes*, progressive meshes give the overall shape of the entire object, but at a low level of detail. Additional data, new edges and faces, progressively increase the detail of the mesh.

Normal meshes transmit progressive changes to a mesh as a set of normal displacements from a base mesh. With this technique, a series of textures represent the desired incremental modifications. Normal meshes are compact, since only a single scalar value is needed to express displacement. However, the technique requires a complex series of transformations to create the displacement textures.

File formats

Polygonal meshes can be stored in a number of file formats:

- FBX
- 3DS
- Collada
- DXF
- Obj
- PLY
- STL
- VRML
- X3D
- .C4D

See also

- Wire-frame model
- Euler operator
- B-rep
- Simplex
- Triangulation (advanced geometry)
- Manifold (a mesh can be manifold or non-manifold)

References

[1] Colin Smith, On Vertex-Vertex Meshes and Their Use in Geometric and Biological Modeling, http://algorithmicbotany.org/papers/smithco.dis2006.pdf

[2] Bruce Baumgart, Winged-Edge Polyhedron Representation for Computer Vision. National Computer Conference, May 1975. http://www.baumgart.org/winged-edge/winged-edge.html

[3] Tobler & Maierhofer, A Mesh Data Structure for Rendering and Subdivision. 2006. (http://wscg.zcu.cz/wscg2006/Papers_2006/Short/E17-full.pdf)

External links

- Weisstein, Eric W., " Simplicial complex (http://mathworld.wolfram.com/SimplicialComplex.html)" from MathWorld.
- Weisstein, Eric W., " Triangulation (http://mathworld.wolfram.com/Triangulation.html)" from MathWorld.

Article Sources and Contributors

Polygon mesh *Source*: http://en.wikipedia.org/w/index.php?oldid=404173523 *Contributors*: ALoopingIcon, Andreas Kaufmann, BenFrantzDale, Berland, Chadernook, Chrschn, Cobi, Cornellier, Crahul, EthanL, Evakuate, Furrykef, Giftlite, HannesJvV, Happyrabbit, Hasanisawi, Havemann, Hymek, Jengelh, Jeodesic, Kevin, Lobsterbake, MaSt, Mackseem, Me Three, Mr.stilt, Orderud, Rchoetzlein, Reyk, Ronz, Siddhant, Some Old Man, SoylentGreen, Sterrys, Tamfang, Tetracube, Tom Edwards, Tomas e, Tomruen, Victorbabkov, Waldir, Wwlin, Wykypydya, Δ, 45 anonymous edits

Image Sources, Licenses and Contributors

File:Dolphin triangle mesh.png *Source*: http://en.wikipedia.org/w/index.php?title=File:Dolphin_triangle_mesh.png *License*: Public Domain *Contributors*: German

File:mesh overview.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Mesh_overview.svg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: User:Lobsterbake

File:mesh vv.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:Mesh_vv.jpg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Berland, Rchoetzlein

File:mesh fv.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:Mesh_fv.jpg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Berland, Rchoetzlein

File:mesh we2.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:Mesh_we2.jpg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Berland, Rchoetzlein

License