02: Mesh Data Structures

Dr. Hamish Carr

Data Structures

- Based on identifying the requirements:
  - topological – mesh assumptions
  - algorithmic – what operations we will need
  - resource – how much memory / bandwidth
  - complexity – preprocessing / runtime cost
- Throughout, we will assume IEEE floats (4B)

Topological Requirements

- Meshes are manifold (if possible)
- If not, the boundary is clearly marked
- Meshes are triangulations
  - Easily enforced – subdivide polygons
- Sometimes generalise to quads
- Almost never use larger mesh elements

Algorithmic Requirements

- Operations will include:
  - Rendering
  - GPU transfer
  - Iteration for mesh processing
  - Insertion / deletion / modification

Operations

a. access to vertices, edges, faces

b. iteration over vertices, edges, faces (any order)

c. access to endpoints of edge

d. access to faces for each edge

e. oriented traversal of edges (walk around face)

f. oriented traversal of vertices (ditto) c,e

g. oriented traversal of edges (around vertex) d,e

h. oriented traversal of faces (ditto) d,g

Example: Smooth Normals

- We'll use this as a running example
- For each vertex
  - *average* the adjacent normals
  - to produce an approximate vertex normal
- Strictly speaking, don't need to iterate around
  - We just need to iterate over adjacent faces
  - Any order will do

Resource Requirements

- Memory footprint is the key issue
- It also influences bandwidth
  - Memory transfer to GPU for rendering
  - Not all operations are on GPU
- Ideal language these days is C++
  - Because it enforces the POD rule

Runtime Polymorphism

- OO languages have *runtime* polymorphism
  - decides which function to call at runtime
  - needs to store information on object class
  - typically done with trap table pointer
  - pointer to details of class in memory
  - object name, method base addresses, &c.

Trap Table Penalty

- This trap table pointer is on CPU
  - it's *totally* useless on GPU
- If you have a class representing a point
  - `Point3 { float x, y, z; NULL *trapTablePtr;}`
  - 12B for your payload, 4B or 8B for the pointer
- GPU bandwidth goes up by 33% or 67%
- Unless you strip it out first (i.e. copy!)

POD Loophole

- If you have no runtime polymorphism
  - i.e. no virtual methods *at all*
  - you don't need the trap table pointer
- C++ guarantees not to create one
  - So you can copy your data straight to GPU
- And arrays are even better!
  - One memory copy for the entire array

Complexity Requirements

- Algorithms should be O(n) if possible
- Insertions / deletions should be O(1) cost
- Iterations should also be O(1) cost
- O(n lg n) should be reserved for preprocessing
- O(lg n) is the maximum runtime lookup cost

Mesh Data Structures

- Face
- Indexed Face
- Winged-Edge
- Half-Edge
- Directed Edge

Face Data Structure

- Simplest approach:
  - Each face consists of 3 (xyz) points
  - Render uses glVertex() or equivalent
  - Most operations iterate over faces
  - Cannot (usually) enforce constraints
  - Attributes (normals, &c.) need to be repeated
- Colloquially called *triangle/polygon soup*

Face Operations

- Reading / rendering is easy (but not fast)
- Vertices stored 6x, prone to matching errors
- Cannot iterate around vertices
- Cannot (easily) cross edge to next face
- Can iterate over vertices 6x, edges 2x
  - by iterating through faces
- Can iterate over, around faces
- No per vertex/edge storage

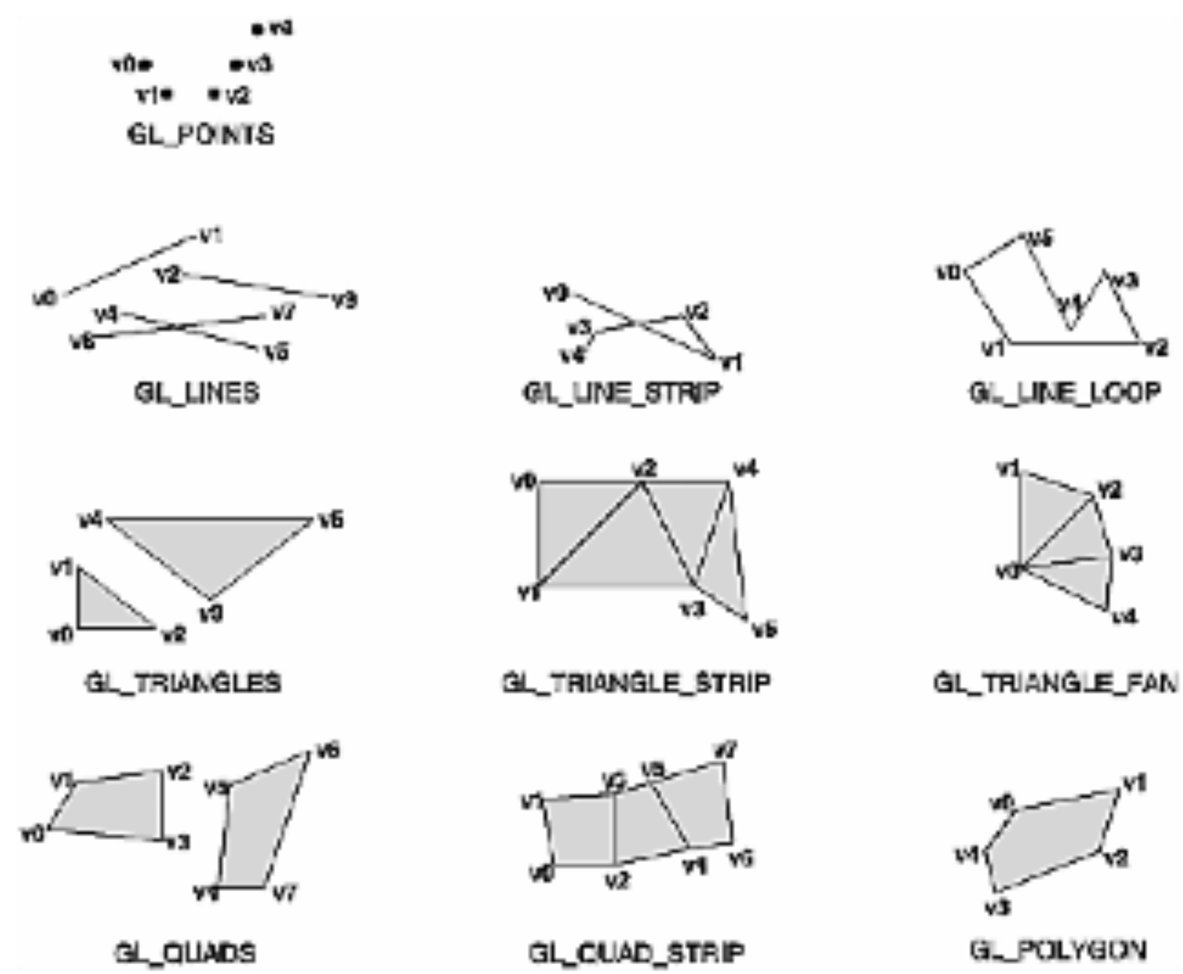
Smooth Normals- Face

```
// FACE DATA STRUCTURE
for (each face f)
  { // loop through faces
  // start with a zero vector
  normal = zero
  for (each vertex v on face f)
    { // loop through vertices on face
    // loop through all faces (including this one)
    for (each face f1)
      { // loop through faces again
      for (each vertex v1 on face f1)
        { // loop through vertices on faces
        // if it's the same vertex (look out for roundoff errors!)
        if (v1 == v)
          { // vertex match
          compute normal n_f1 for face f1
          normal += n_f1
          } // vertex match
        } // loop through vertices on faces
      } // loop through faces again
  // note: we don't need to keep the degree or divide through
  // because we're going to normalise anyway!
  normalise(normal)
  } // loop through faces
```

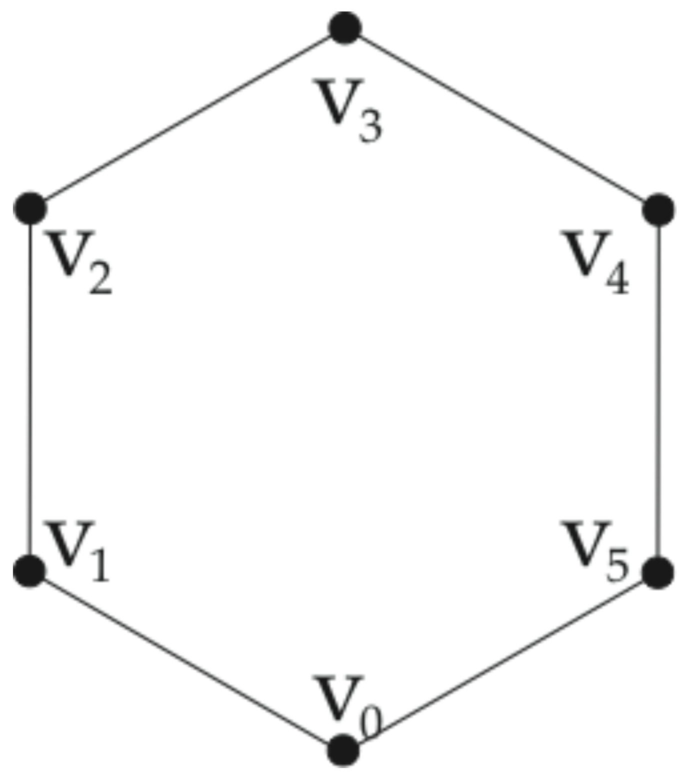
- Hopelessly inefficient – $O(v^2)$
- Recomputes each normal six times

(Old) Face Improvements

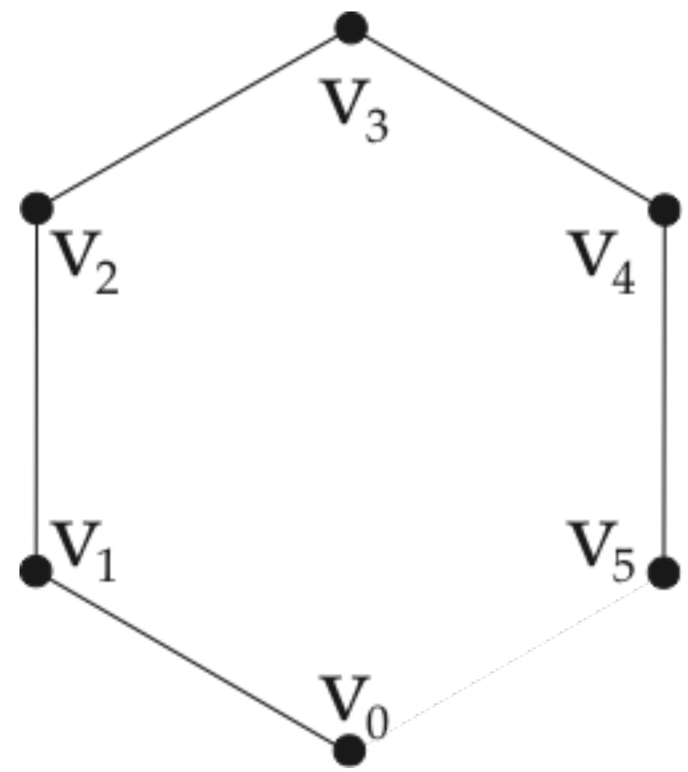
- Display lists – save bandwidth, cache on GPU
- Implicit vertex reuse
  - Line Strips & Loops
  - Triangle Fans & Strips
  - Quads, Quad Strips
  - Polygons



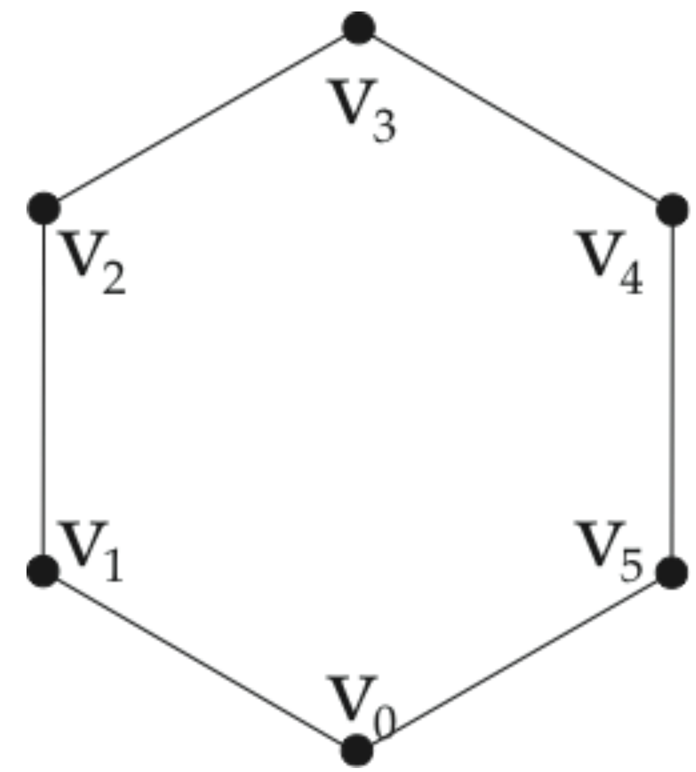
Lines, Strips & Loops



GL_LINES:
v0v1
v1v2
v2v3
v3v4
v4v5
v5v0
2 v/e

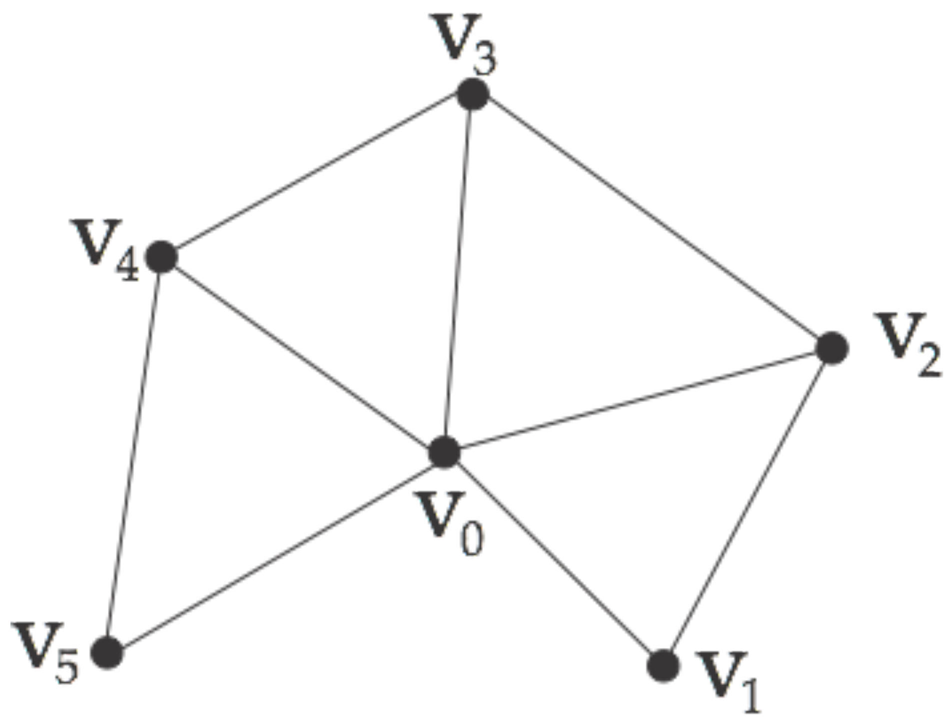


GL_LINE_STRIP:
v0v1
(v1)v2
(v2)v3
(v3)v4
(v4)v5
~ 1 v/e

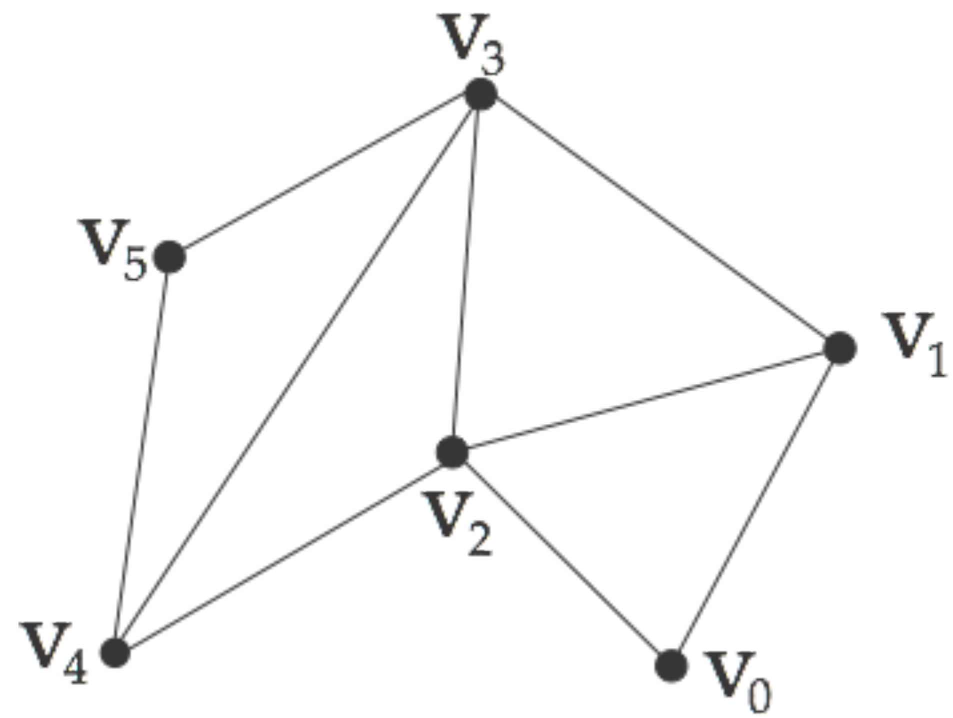


GL_LINE_LOOP:
v0v1
(v1)v2
(v2)v3
(v3)v4
(v4)v5
(v5)(v0)
~ 1 v/e

Triangle Fans/Strips



GL_TRIANGLE_FAN:
v0v1v2
(v0)(v2)v3
(v0)(v3)v4
(v0)(v4)v5
~ 1.16 v/e
(degree 6 vertex)



GL_TRIANGLE_STRIP:
v0v1v2
(v2)(v1)v3
(v2)(v3)v4
(v4)(v3)v5
~ 1 v/e
(for long strips)
alternate CW/CCW
NP-hard to find them
easier on regular grids

Face Costs

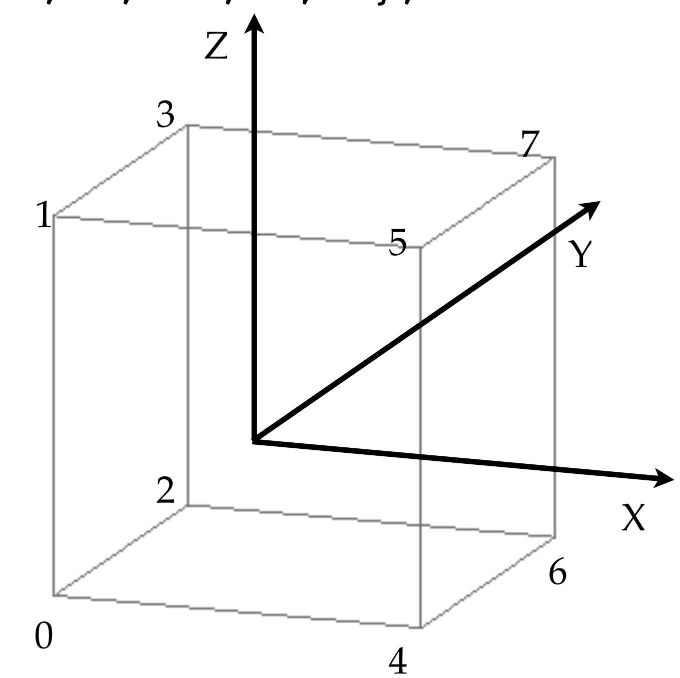
- Memory / bandwidth cost:
  - 3 x 3 x 4B per face = 36B
  - f ~ 2v, so 72B per vertex
- Every glVertex() call has overhead
  - so no advantage for batched operations
- Horrendously inefficient operations
- But easy and lowest common denominator

Indexed Face Data Structure

- Store a *single* array of vertices / attributes
- Store their indices (or pointers) for each face
  - Pointers are no longer cheaper
  - *Most* PUs have 1-op array lookup
- Render uses vertex arrays / VBOs
- But operations aren't so easy

Vertex Arrays

```
GLint vertices[8][3] = {  -1, -1, -1,   -1, -1,  1,   -1,  1, -1,   -1,  1,  1,
                           1, -1, -1,    1, -1,  1,    1,  1, -1,    1,  1,  1};
GLint normals[6][3] =  {  -1,  0,  0,    0, -1,  0,    0,  0, -1
                           1,  0,  0,    0,  1,  0,    0,  0,  1};
unsigned int triangles[36] = {  0, 1, 3,   0, 3, 2,   0, 1, 4,   1, 4, 5,
                                0, 2, 4,   2, 6, 4,   5, 4, 6,   5, 6, 7,
                                2, 3, 7,   2, 7, 6,   1, 5, 7,   1, 7, 3};

glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_INT, 0, vertices);
glEnableClientState(GL_NORMAL_ARRAY);
glNormalPointer(3, GL_INT, 0, normals);

// this runs it's own loop internally
glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_INT, indices);

// turn it off when we're done
glDisableClientState(GL_VERTEX_ARRAY);
glDisableClientState(GL_NORMAL_ARRAY);
```



- Replace individual glVertex() calls
- Pass *entire arrays* to library
- Push loop through vertices into library call

Vertex Buffer Objects

- Same idea, but data transferred to VRAM
- I.e. we cache the data & save the bandwidth
- Which means memory management:
  - allocation: glGenBuffers, glDeleteBuffers
  - binding: glBindBuffer
  - transfer: glBufferData
- And more is moving onto GPU with Vulkan

Indexed Face Operations

- Reading / rendering is easy & fast
- Cannot iterate around vertices
  - But can iterate over them
- Cannot (easily) swap faces across edge
- Can iterate over, around faces
- No per edge storage
- Overall weakness – edge operations

Smooth Normals – Indexed Face

```
// INDEXED FACE DATA STRUCTURE
// initialise the normals to zeros for each vertex
for (each vertex v)
  normal[v] = zero

// now loop through the faces
for (each face f)
  { // loop through faces
  // start with a zero vector
  compute normal n_f for face f
  // loop through face's vertices
  for (each vertex v on face f)
    { // per vertex
    // add the face normal to the vertex normal
    normal[v] += n_f
    } // per vertex
  } // loop through faces

// now normalise them all
for (each vertex v)
  normalise(normal[v])
```

- Much better – O(v) computation
- But it cannot iterate *around* vertices yet

Reading Polygon Soup

```
int main(int argc, char **argv)
    { // main()
    std::vector<Cartesian3> raw_vertices;
    while (std::cin.good())
        { // loop to read
        Cartesian3 next;
        std::cin >> next.x >> next.y >> next.z;
        raw_vertices.push_back(next);
        } // loop to read

    // now we loop to set vertex IDs for each existing vertex
    std::vector<long> vertexID(raw_vertices.size(), -1);

    // set the initial vertex ID
    long nextVertexID = 0;

    // loop through the vertices
    for (long vertex = 0; vertex < raw_vertices.size(); vertex++)
        { // vertex loop
        // first see if the vertex already exists
        for (long other = 0; other < vertex; other++)
            { // per other
            if (raw_vertices[vertex] == raw_vertices[other])
                vertexID[vertex] = vertexID[other];
            } // per other
        // if not set, set to next available
        if (vertexID[vertex] == -1)
            vertexID[vertex] = nextVertexID++;
        } // vertex loop
```

Writing Indexed Faces

```
// the first four lines will be skipped completely
std::cout << "#" << std::endl;
std::cout << "# Created for Leeds COMP 5821M Autumn 2019" << std::endl;
std::cout << "#" << std::endl;
std::cout << "#" << std::endl;
std::cout << "# Surface vertices=" << nextVertexID << " faces="
          << raw_vertices.size()/3 << std::endl;
std::cout << "#" << std::endl;

// id of next vertex to write
long writeID = 0;

// loop to write the vertices out
for (long vertex = 0; vertex < raw_vertices.size(); vertex++)
    { // per vertex
    // if it's the first time found
    if (writeID == vertexID[vertex])
        { // first time found
        // print out the coordinates
        std::cout << "Vertex " << writeID << " " << raw_vertices[vertex].x << " "
                  << raw_vertices[vertex].y << " " << raw_vertices[vertex].z << std::endl;
        // increment the ID number
        writeID++;
        } // first time found
    } // per vertex

// loop to write the face vertices out
for (long face = 0; face < raw_vertices.size()/3; face++)
    { // per face
    std::cout << "Face " << face << "  " << vertexID[3 * face] << " "
              << vertexID[3 * face + 2] << " " << vertexID[3 * face + 1] << std::endl;
    } // per face

// done
return 0;
} // main()
```

Indexed Face Costs

- Memory / bandwidth cost:
  - 3 x 4B per vertex= 12B
  - f ~ 2v, so 6 x 4B indices per vertex
  - Total: 36B / vertex
- Efficient rendering
- Default in practice: OBJ files / VBOs
- But limited in its operations

Improvements Needed

- Mesh *processing* requires more operations
- In particular, iterating in *cyclic* order
- This allows us to preserve topology
  - e.g. testing 2-manifold conditions
  - can't (easily) do this with indexed face
- We will need to track vertices, faces, edges
  - Each will need index + attribute storage

Face-Based Structures

- Each vertex stores:
  - position 12B
  - 1 face index (first) 4B 16B/vert
- Each face stores:
  - 3 vertex indices 12B
  - 3 neighbours (face indices) 12B 24B/face
    - (2 faces / vertex) 48B /vert
- Total: 64B / vert
-

Smooth Normals – Face-Based

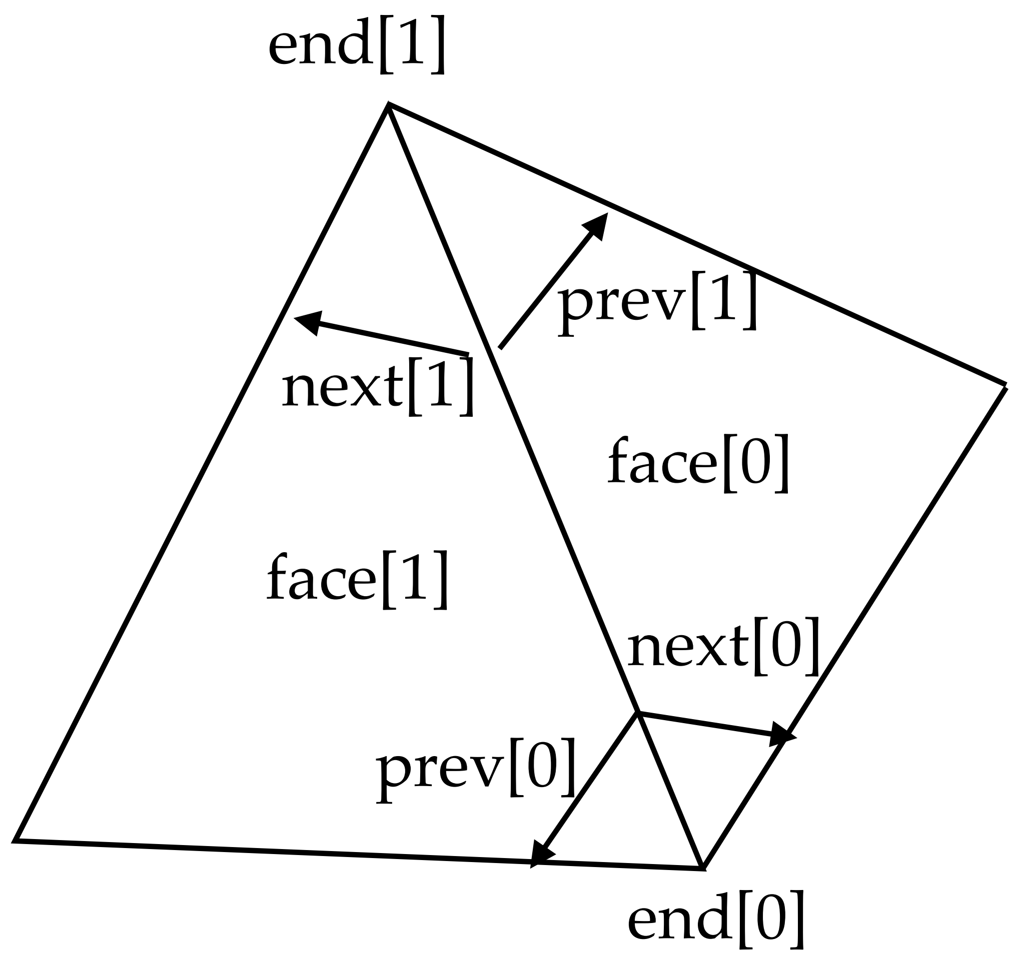
```
// FACE-BASED DATA STRUCTURE
// loop through vertices
for (each vertex v)
  { // for each vertex
  // initialise normal to zero
  normal[v] = zero
  // start loop control variable at first face
  face = first[v]
  do { // do loop around neighbours
     compute normal n_f for face f
     normal[v] += n_f
     // now find which neighbour to use
     if (faceV[0] == v)
       face = neighbour[0]
     else if (faceV[1] == v)
       face = neighbour[1]
     else if (faceV[2] == v)
       face = neighbour[2]
     } // do loop around neighbours
  while (face != first[v])

  // now normalise
  normalise(normal[v])
  } // for each vertex
```

- Iteration around vertices now possible
- But lots of if statements
- Starts failing with non-triangular faces

Edge-Based Structures

- These start *explicitly* representing edges
- And using them to store cyclic edge loops
- Each edge points to the next/previous edge
- At each end, and for each face
- Paradigm is the *winged edge* (Baumgart, 1972)

Winged Edge



UNIVERSITY OF LEEDS

Winged Edge Structures

- Each vertex stores:
  - position 12B
  - 1 edge index (first) 4B 16B/vertex
- Each edge stores:
  - 2 vertex indices 8B
  - 2 face indices 8B
  - 2 edge indices (next) 8B
  - 2 edge indices (prev) 8B 32B/edge
    - (3 edges / vertex) 96B/vertex
- Each face stores:
  - 1 edge index (first) 4B
    - (2 faces / vertex) 8B /vert
- Total: 120B / vert

Smooth Normals – Winged Edge

```
// WINGED-EDGE DATA STRUCTURE
// loop through vertices
for (each vertex v)
  { // for each vertex
  // initialise normal to zero
  normal[v] = zero
  // start loop control variable at first edge
  edge = firstEdge[v]
  do { // do loop around neighbours
     // retrieve the corresponding face
     face = edge.face[0]
     // note: each vertex will need retrieval through edge
     // there are some further optimisations available here
     compute normal n_f for face f
     normal[v] += n_f
     // now step forward
     if (edge == firstEdge[end[0]])
         edge = edge.next[0]
     else
         edge = edge.next[1]
     } // do loop around neighbours
  while (edge != firstEdge[v])

  // now normalise
  normalise(normal[v])
  } // for each vertex
```

- This is cleaner, and handles arbitrary meshes
- Face can store vertex IDs as well (+24B/v)
- Still need if statements to track things

Half-Edge Structures

- Break winged-edges into a pair of half-edges
  - Mantyla 1988, Kettner 1999
- Each half edge has:
  - vertex, face, next, prev
  - ID of paired half-edge
    - But store them in pairs
    - Use XOR on bit 0 to flip between them
- Same cost as winged-edge, simpler code

Smooth Normals – Half Edge

```
// HALF-EDGE DATA STRUCTURE
// loop through vertices
for (each vertex v)
  { // for each vertex
  // initialise normal to zero
  normal[v] = zero
  // start loop control variable at first edge
  halfedge = firstHalfEdge[v]
  do { // do loop around neighbours
     // retrieve the corresponding face
     face = halfedge.face[0]
     // note: each vertex will need retrieval through edge
     // there are some further optimisations available here
     compute normal n_f for face f
     normal[v] += n_f
     // now step forward
     edge = edge.next
     } // do loop around neighbours
  while (edge != firstHalfEdge[v])

  // now normalise
  normalise(normal[v])
  } // for each vertex
```

- This is nice, but still memory-inefficient
- We need to reduce the per edge costs

Directed Edge Structures

- Campagna 1998
- Stores the directed-edges *per triangle*
- I.e. in groups of three
- Assumes arithmetic cheap, memory not
- Uses modular arithmetic (%3) instead of XOR
- And uses it for implicit storage of topology

Conversion

- Each directed edge d is no. i on face f:
  - $d = 3f + i$ $\quad f = h/3$ $\quad i = h\%3$
    - d is directed face index
    - f is face index
    - i is 0, 1, or 2 in CCW order
- $next = 3f + (i + 1) \% 3$
- $prev = 3f + (i + 2) \% 3$

Smooth Normals – Directed Edge

```
// DIRECTED-EDGE DATA STRUCTURE
// loop through vertices
for (each vertex v)
  { // for each vertex
  // initialise normal to zero
  normal[v] = zero
  // start loop control variable at first edge
  halfedge = firstHalfEdge[v]
  do { // do loop around neighbours
     // retrieve the corresponding face
     face = halfedge / 3
     whichEdge = halfedge % 3
     // note: each vertex will need retrieval through edge
     // there are some further optimisations available here
     compute normal n_f for face f
     normal[v] += n_f
     edge = 3 * face + (whichEdge + 1) % 3
     } // do loop around neighbours
  while (edge != firstHalfEdge[v])

  // now normalise
  normalise(normal[v])
  } // for each vertex
```

- Right, let's look at the cost

Directed Edge Structures

- Each vertex stores:
  - position — 12B
  - 1 edge index (first) — 4B — 16B/vertex
- Each directed edge stores:
  - 1 vertex index — 4B
  - 1 paired directed edge index — 4B — 8B/directed edge
    - (6 directed edges / vertex) — 48B / vertex
- Each face stores:
  - nothing
- Total: — 64B/ vertex

Implementation

- We will do this C-style, not OO
- so we get that efficiency as well
- the vertices are stored in a vertex array
- and the faces have vertex indices in an array
  - since directed edges are stored per face
  - their vertex indices are *also* the face's list

Code (Finally)

```
// data structure for directed-edge processing
class Mesh
    { // Mesh
    public:
    std::vector<Point3D> position;                    // xyz
    std::vector<indexType> firstDirectedEdge;         // per vertex
    std::vector<indexType> faceVertices;              // doubles as "directed edge to"
    std::vector<indexType> otherHalf;                 // pairs the directed edges

    void Render()
        { // Render()
        // use the arrays for vertex array calls
        // VBO's are similar, but a bit messier
        glEnableClientState(GL_VERTEX_ARRAY);
        glVertexPointer(3, GL_INT, 0, position);
        glDrawElements(GL_TRIANGLES, faceVertices.size(), GL_UNSIGNED_INT, faceVertices);
        glDisableClientState(GL_VERTEX_ARRAY);
        } // Render()


    }; // Mesh
```

- After all that mess, it's surprisingly simple
- But it's not perfect

Directed Edge Comments

- A boundary edge does not have a pair
  - So store a negative index
- Similar structure for quads is easy to construct
  - but we can't mix the two
- Libraries exist (OpenMesh, CGAL, &c.)
- But most people roll their own
  - Because their *attributes* vary