Meshes: Memory Formats

Siddhartha Chaudhuri  http://www.cse.iitb.ac.in/~cs749
Recap: Polygon Meshes

- A mesh is a **discrete sampling** of a surface (vertices), plus **locally linear** approximations (simple polygons).
- A mesh is a **graph**.
Today

• How do we store a mesh?
  • In RAM
  • On disk
Closed/Open, Connected/Disconnected

- **Closed**: No boundary edges (adjacent to a single face)

- **Connected**: Same definition (and algorithm) as for a graph

These shapes are individually connected, but not connected to each other.
Manifold vs Non-Manifold

- A mesh is manifold if
  1) Every edge is adjacent to 1 or 2 faces
  2) The faces around every vertex form a closed or open fan
Orientable vs Non-Orientable

- Two adjacent faces are compatible if their vertices wind the same way (both counter-clockwise or both clockwise) around their boundaries.
  - In other words, if their boundaries traverse the shared edge in opposite directions.
- A mesh is orientable if all pairs of adjacent faces are compatible.
Storing a mesh in RAM

• What might we need?
  • Fast *iteration* (over vertices, faces, edges...)
  • Fast graph *traversal*
    – Jump from element to adjacent element, e.g. edge to neighboring faces
  • Stored *attributes* (normals, colors, texture coordinates, features...)
  • Efficient use of *space*
  • Other considerations, e.g. caching adjacent elements in nearby memory locations
A simple memory format

vertices = {
    { 0.0, 0.0, 0.0 },
    { 0.0, 0.0, 1.0 },
    { 0.0, 1.0, 0.0 },
    { 0.0, 1.0, 1.0 },
    { 1.0, 0.0, 0.0 },
    { 1.0, 0.0, 1.0 },
    { 1.0, 1.0, 0.0 },
    { 1.0, 1.0, 1.0 },
};

quads = {
    { 0, 2, 6, 4 },
    { 0, 1, 3, 2 },
    { 2, 3, 7, 6 },
    { 4, 6, 7, 5 },
    { 0, 4, 5, 1 },
    { 1, 5, 7, 3 },
};

In practice, maybe a vector<Vec3>

References to vertex list

In practice, maybe a vector<array<long, 4>>
A simple memory format

vertices = {
    { 0.0, 0.0, 0.0 },
    { 0.0, 0.0, 1.0 },
    { 0.0, 1.0, 0.0 },
    { 0.0, 1.0, 1.0 },
    { 1.0, 0.0, 0.0 },
    { 1.0, 0.0, 1.0 },
    { 1.0, 1.0, 0.0 },
    { 1.0, 1.0, 1.0 },
    { 2.0, 0.5, 0.5 }
};

quads = {
    { 0, 2, 6, 4 },
    { 0, 1, 3, 2 },
    { 2, 3, 7, 6 },
    { 4, 6, 7, 5 },
    { 0, 4, 5, 1 },
    { 1, 5, 7, 3 }
};

triangles = {
    { 7, 8, 6 },
    { 5, 8, 7 },
    { 4, 8, 5 },
    { 6, 8, 4 }
};


Pros and Cons

- **Fast iteration, good for rendering**

  ```
  glBegin(GL_QUADS);
  for (size_t i = 0; i < quads.size(); ++i)
    for (size_t j = 0; j < 4; ++j) {
      Vec3 const & v = vertices[quads[i][j]];
      glVertex3f(v.x, v.y, v.z);
    }
  glEnd();
  ```

- **Directly maps to GPU vertex and index buffer formats**

- **Compact use of space**
  - Higher-degree polys are usually rare and can be stored in separate list (such as a `vector< vector<long> >`)

- **Bad for traversal**
  - How would you go from a vertex to its neighbors?
  - How would you go from a vertex to its adjoining faces?
Adjacencies

• Let's explicitly store the graph structure
  • Every **vertex** will store its incident faces and edges
  • Every **edge** will store its two endpoints, and its adjoining faces
  • Every **face** will store its vertices and edges

```cpp
class Vertex {
  Vec3 position;
  Vec3 normal;
  list<Face *> faces;
  list<Edge *> edges;
};

class Edge {
  double length;
  Vertex * endpoints[2];
  list<Face *> faces;
};

class Face {
  Vec3 normal;
  list<Vertex *> vertices;
  list<Edge *> edges;
};

(Constructors, accessors and other functions omitted)

Mesh = [ list<Vertex>, list<Edge>, list<Face> ]
```
Pros and Cons

- **Fast iteration**
  - ... over any standard subset of elements (all vertices, or vertices around a face, or edges at a vertex...)

- **Great for traversal**
  - Can go from any element to its adjoining elements (of any type) in $O(1)$ time

- **Ok use of space**
  - Typically a constant-factor overhead

- Such adjacency-heavy representations are good for geometric algorithms
Analysis of storage overhead

• For a manifold surface
  • Each edge has (at most) two adjacent faces
    – ... so #edge-face incidences \(\leq 2E\)
  • Number of vertices around a face = number of edges around the face
    – ... so #vertex-face incidences \(\leq 2E\)
  • Each edge has two endpoints
    – ... so #edge-vertex incidences \(\leq 2E\)
• So total overhead of the adjacency information = \(O(E)\)
  – ... = \(O(V + F)\), for small genus
Euler-Poincaré formula

- For a closed polygonal mesh with $V$ vertices, $E$ edges and $F$ faces
  
  $$V - E + F = \chi$$

- $\chi$ is the Euler characteristic of the surface
  
  - For a closed, connected, orientable 2-manifold, $\chi = 2(1 - g)$
  
  - $g$ is the genus of the surface
    
    - Number of holes/handles
    - More formally, the number of cuttings along simple closed loops on the surface that do not disconnect it
Euler Characteristic

Platonic solids (and other convex polyhedra): $g = 0, \chi = 2$

Sphere: $g = 0, \chi = 2$

Two spheres: $\chi = 4$

Torus: $g = 1, \chi = 0$

Double torus: $g = 2, \chi = -2$

Triple torus: $g = 3, \chi = -4$

Möbius strip: $\chi = 0$

Klein bottle: $\chi = 0$

$g = 0, \chi = 2$ (if you don't model the alimentary canal)
Euler-Poincaré formula

• For a closed polygonal mesh with \( V \) vertices, \( E \) edges and \( F \) faces, \( V - E + F = \chi \)

• For small genus/characteristic, gives \( E \approx V + F \)

• Consider a closed manifold mesh with only triangles
  • Each edge borders two faces, each face borders 3 edges
    • ... so \( 2E = 3F \)
    • ... and plugging this into the formula, \( V = E/3 + \chi \)
  • Hence, the vertex, edge and face counts are all (asymptotically) the same, for fixed characteristic
    • \( O(V) = O(E) = O(F) \)
Space-efficient adjacencies

• Can we store adjacencies more efficiently?
  • Yes! For manifold, orientable surfaces, we can store the graph with a constant storage overhead *per-element* (no arbitrary-size lists)
  • ... without changing the complexity of traversal
Half-Edge Data Structure
aka Winged Edge Data Structure, aka Doubly-Connected Edge List (DCEL)

- Only for manifold, orientable surfaces
- Instead of an edge, store two opposing half-edges linked to each other
  - Every half-edge links to its twin
- For every vertex, store one half-edge exiting it
  - The half-edge also links back to this source vertex
- For every face, store one half-edge on its boundary that traverses it counter-clockwise
  - The half-edge links back to this adjacent face
  - ... and also to the next half-edge along the boundary of the same face

```cpp
class Vertex {
    Vec3 position;
    HalfEdge * half_edge;
};
class HalfEdge {
    HalfEdge * twin;
    HalfEdge * next;
    Vertex * source;
    Face * face;
};
class Face {
    HalfEdge * half_edge;
};
```
Traversal Building Blocks

- The tip of a half-edge $E$
  - $E \rightarrow \text{twin} \rightarrow \text{source}$
- The boundary of a face
  - Follow the next pointer
- From a face $F$ to an adjacent face
  - $F \rightarrow \text{half\_edge} \rightarrow \text{twin} \rightarrow \text{face}$
- All edges at a vertex $V$
  - Start from $V \rightarrow \text{half\_edge}$, follow $\text{twin} \rightarrow \text{next}$
- ...
Storing a mesh on disk

- **OBJ**: a simple and common file format
  - Plain text, easy to hand-review and edit if needed
  - Also see: OFF, PLY, STL

`cube.obj`

Vertex positions, one per line

List of vertex indices for each face, one face per line (indices are 1-based)

(Many more tags not listed here, see https://en.wikipedia.org/wiki/Wavefront_.obj_file http://www.martinreddy.net/gfx/3d/OBJ.spec)