Computer Graphics

Prof. Feng Liu

Fall 2018

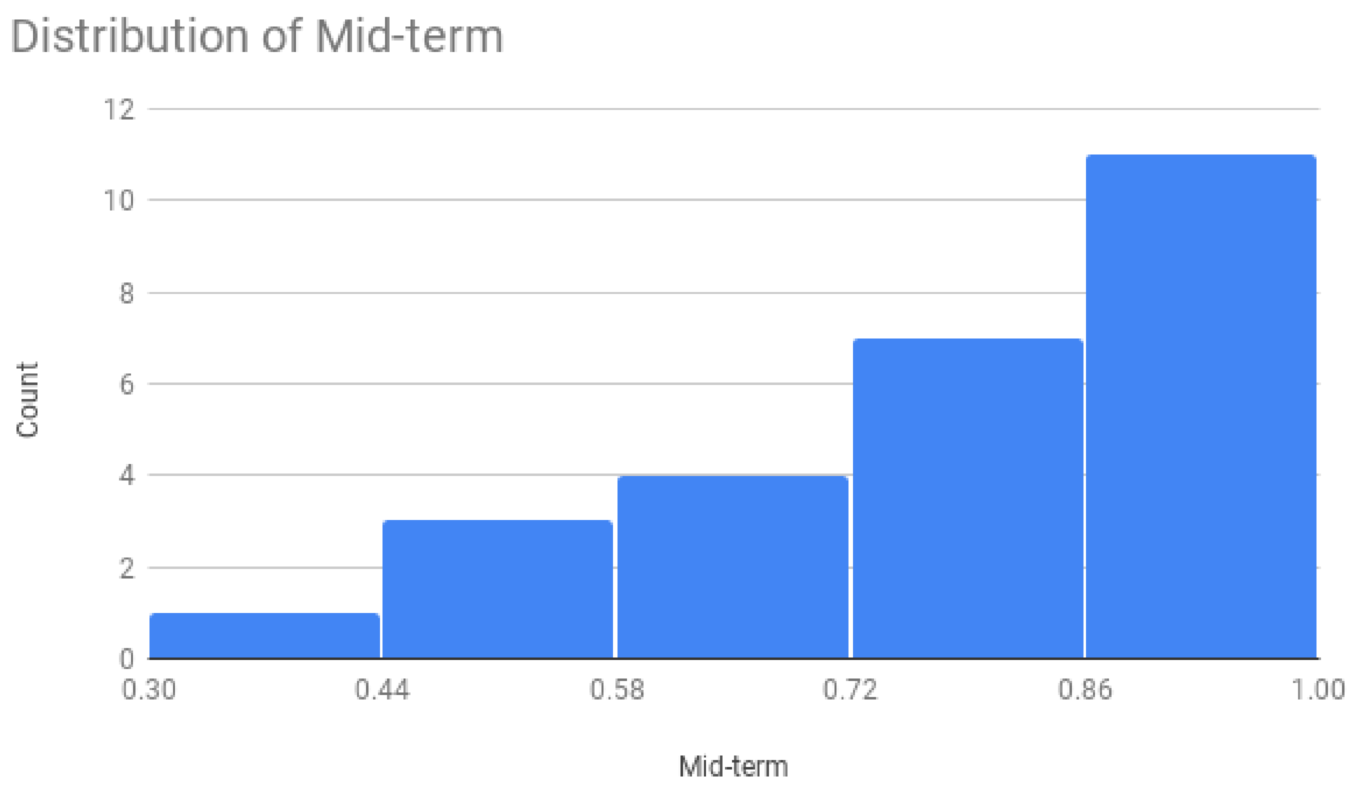
http://www.cs.pdx.edu/~fliu/courses/cs447/

11/13/2018

Last time

- Texture Mapping

Mid-term


Distribution of Mid-term
12
10
8
6
4
2
0
Count
0.30
0.44
0.58
0.72
0.86
1.00
Mid-term

Today

- Mesh and Modeling

The Story So Far

- We’ve looked at images and image manipulation
- We’ve looked at rendering from polygons
- Next major section:
  - Modeling

Modeling Overview

- Modeling is the process of describing an object
- Sometimes the description is an end in itself
  - eg: Computer aided design (CAD), Computer Aided Manufacturing (CAM)
  - The model is an exact description
- More typically in graphics, the model is then used for rendering (we will work on this assumption)
  - The model only exists to produce a picture
  - It can be an approximation, as long as the visual result is good
- The computer graphics motto: "If it looks right it is right"
  - Doesn't work for CAD

Issues in Modeling

- There are many ways to represent the shape of an object
- What are some things to think about when choosing a representation?

Choosing a Representation

- How well does it represent the objects of interest?
- How easy is it to render (or convert to polygons)?
- How compact is it (how cheap to store and transmit)?
- How easy is it to create?
  - By hand, procedurally, by fitting to measurements, …
- How easy is it to interact with?
  - Modifying it, animating it
- How easy is it to perform geometric computations?
  - Distance, intersection, normal vectors, curvature, …

Categorizing Modeling Techniques

- Surface vs. Volume
  - Sometimes we only care about the surface
    - Rendering and geometric computations
  - Sometimes we want to know about the volume
    - Medical data with information attached to the space
    - Some representations are best thought of defining the space filled, rather than the surface around the space
- Parametric vs. Implicit
  - Parametric generates all the points on a surface (volume) by "plugging in a parameter" eg $(\sin\phi \cos\theta, \sin\phi \sin\theta, \cos\phi)$
  - Implicit models tell you if a point in on (in) the surface (volume) eg $x^2 + y^2 + z^2 - 1 = 0$

Techniques

- Polygon meshes
  - Surface representation, Parametric representation
- Prototype instancing and hierarchical modeling
  - Surface or Volume, Parametric
- Volume enumeration schemes
  - Volume, Parametric or Implicit
- Parametric curves and surfaces
  - Surface, Parametric
- Subdivision curves and surfaces
- Procedural models

Polygon Modeling

- Polygons are the dominant force in modeling for real-time graphics
- Why?

Polygons Dominate

- Everything can be turned into polygons (almost everything)
  - Normally an error associated with the conversion, but with time and space it may be possible to reduce this error
- We know how to render polygons quickly
- Many operations are easy to do with polygons
- Memory and disk space is cheap
- Simplicity

What’s Bad About Polygons?

- What are some disadvantages of polygonal representations?

Polygons Aren't Great

- They are always an approximation to curved surfaces
  - But can be as good as you want, if you are willing to pay in size
  - Normal vectors are approximate
  - They throw away information
  - Most real-world surfaces are curved, particularly natural surfaces
- They can be very unstructured
- They are hard to globally parameterize (complex concept)
  - How do we parameterize them for texture mapping?
- It is difficult to perform many geometric operations
  - Results can be unduly complex, for instance

Polygon Meshes

- A *mesh* is a set of polygons connected to form an object
- A mesh has several components, or geometric entities:
  - Faces
  - Edges, the boundary between faces
  - Vertices, the boundaries between edges, or where three or more faces meet
  - Normals, Texture coordinates, colors, shading coefficients, etc
- Some components are implicit, given the others
  - For instance, given faces and vertices can determine edges

Polygonal Data Structures

- Polygon mesh data structures are **application dependent**
- Different applications require different operations to be fast
  - Find the neighbor of a given face
  - Find the faces that surround a vertex
  - Intersect two polygon meshes
- You typically choose:
  - Which features to store explicitly (vertices, faces, normals, etc)
  - Which relationships you want to be explicit (vertices belonging to faces, neighbors, faces at a vertex, etc)

Polygon Soup

- Many polygon models are just lists of polygons

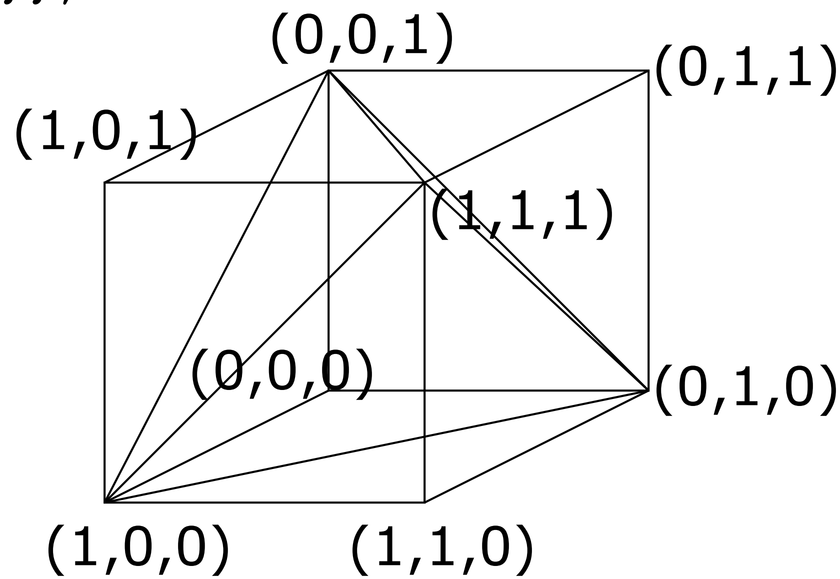
```
struct Vertex {
    float coords[3];
}
struct Triangle {
    struct Vertex verts[3];
}
struct Triangle mesh[n];

glBegin(GL_TRIANGLES)
    for ( i = 0 ; i < n ; i++ )
    {
        glVertex3fv(mesh[i].verts[0]);
        glVertex3fv(mesh[i].verts[1]);
        glVertex3fv(mesh[i].verts[2]);
    }
glEnd();
```

Important Point: OpenGL, and almost everything else, assumes a constant vertex ordering: clockwise or counter-clockwise. Default, and slightly more standard, is counter-clockwise

Cube Soup

```
struct Triangle Cube[12] =
        {{{1,1,1},{1,0,0},{1,1,0}},
         {{1,1,1},{1,0,1},{1,0,0}},
         {{0,1,1},{1,1,1},{0,1,0}},
         {{1,1,1},{1,1,0},{0,1,0}},
         …
        };
```



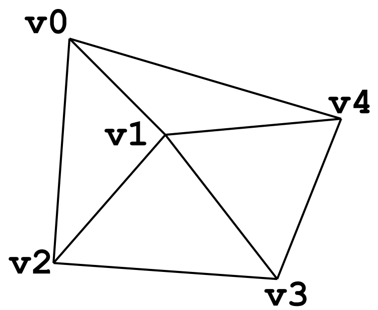
Polygon Soup Evaluation

- What are the advantages?
- What are the disadvantages?

Polygon Soup Evaluation

- What are the advantages?
  - It’s very simple to read, write, etc.
  - A common output format from CAD modelers
  - The format required for OpenGL
- BIG disadvantage: No higher order information
  - No information about neighbors
  - Waste of memory
  - No open/closed information

Vertex Indirection



vertices

v0	v1	v2	v3	v4

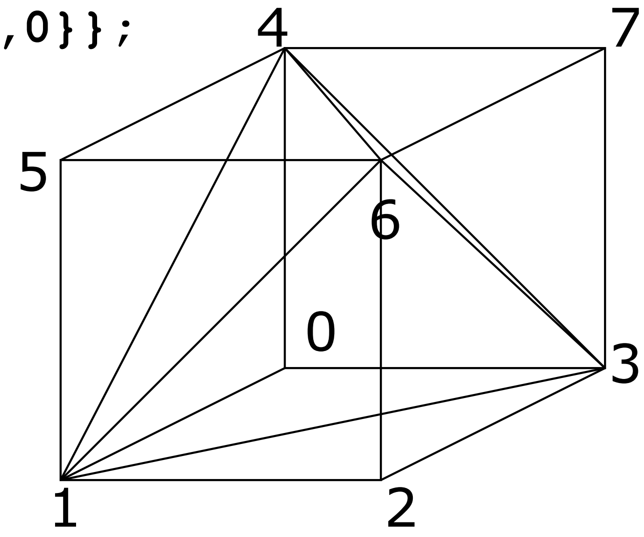
faces

0	2	1	0	1	4	1	2	3	1	3	4

- There are reasons not to store the vertices explicitly at each polygon
  - Wastes memory - each vertex repeated many times
  - Very messy to find neighboring polygons
  - Difficult to ensure that polygons meet correctly
- Solution: Indirection
  - Put all the vertices in a list
  - Each face stores the indices of its vertices
- Advantages? Disadvantages?

Cube with Indirection

```
struct Vertex CubeVerts[8] =
       {{0,0,0},{1,0,0},{1,1,0},{0,1,0},
        {0,0,1},{1,0,1},{1,1,1},{0,1,1}};
struct Triangle CubeTriangles[12] =
       {{6,1,2},{6,5,1},{6,2,3},{6,3,7},
        {4,7,3},{4,3,0},{4,0,1},{4,1,5},
        {6,4,5},{6,7,4},{1,2,3},{1,3,0}};
```



Indirection Evaluation

- Advantages:
  - Connectivity information is easier to evaluate because vertex equality is obvious
  - Saving in storage:
    - Vertex index might be only 2 bytes, and a vertex is probably 12 bytes
    - Each vertex gets used at least 3 and generally 4-6 times, but is only stored once
  - Normals, texture coordinates, colors etc. can all be stored the same way
- Disadvantages:
  - Connectivity information is not explicit

OpenGL and Vertex Indirection

```
struct Vertex {
    float coords[3];
}
struct Triangle {
    GLuint verts[3];
}
struct Mesh {
    struct Vertex vertices[m];
    struct Triangle triangles[n];
}
```

Continued…

OpenGL and Vertex Indirection (v1)

```
glEnableClientState(GL_VERTEX_ARRAY)
glVertexPointer(3, GL_FLOAT, sizeof(struct Vertex),
                  mesh.vertices);
glBegin(GL_TRIANGLES)
    for ( i = 0 ; i < n ; i++ )
    {
        glArrayElement(mesh.triangles[i].verts[0]);
        glArrayElement(mesh.triangles[i].verts[1]);
        glArrayElement(mesh.triangles[i].verts[2]);
    }
glEnd();
```

OpenGL and Vertex Indirection (v2)

```
glEnableClientState(GL_VERTEX_ARRAY)
glVertexPointer(3, GL_FLOAT, sizeof(struct Vertex),
                  mesh.vertices);
for ( i = 0 ; i < n ; i++ )
    glDrawElements(GL_TRIANGLES, 3, GL_UNSIGNED_INT,
                   mesh.triangles[i].verts);
```

- Minimizes amount of data sent to the renderer
- Fewer function calls
- Faster!

Normal Vectors

- Normal vectors give information about the true surface shape
- Per-Face normals:
  - One normal vector for each face, stored as part of face
  - Flat shading
- Per-Vertex normals:
  - A normal specified for every vertex (smooth shading)
  - Can keep an array of normals analogous to array of vertices
  - Faces store vertex indices and normal indices separately
  - Allows for normal sharing independent of vertex sharing

Cube with Indirection and Normals

Vertices:	Normals:	Faces ((vert,norm), …):
(1,1,1)	(1,0,0)	((0,4),(1,4),(2,4),(3,4))
(-1,1,1)	(-1,0,0)	((0,0),(3,0),(7,0),(4,0))
(-1,-1,1)	(0,1,0)	((0,2),(4,2),(5,2),(1,2))
(1,-1,1)	(0,-1,0)	((2,1),(1,1),(5,1),(6,1))
(1,1,-1)	(0,0,1)	((3,3),(2,3),(6,3),(7,3))
(-1,1,-1)	(0,0,-1)	((7,5),(6,5),(5,5),(4,5))
(-1,-1,-1)		
(1,-1,-1)		

Storing Other Information

- Colors, Texture coordinates and so on can all be treated like vertices or normals
- Lighting/Shading coefficients may be per-face, per-object, or per-vertex

Indexed Lists vs. Pointers

- Previous example have faces storing indices of vertices
  - Access a face vertex with:
    `mesh.vertices[mesh.faces[i].vertices[j]]`
  - Lots of address computations
  - Works with OpenGL's vertex arrays
- Can store pointers directly
  - Access a face vertex with:
    `*(mesh.faces[i].vertices[j])`
  - Probably faster because it requires fewer address computations
  - Easier to write
  - Doesn't work directly with OpenGL
  - Messy to save/load (pointer arithmetic)
  - Messy to copy (more pointer arithmetic)

Vertex Pointers

```
struct Vertex {
     float coords[3];
}
struct Triangle {
     struct Vertex *verts[3];
}
struct Mesh {
     struct Vertex vertices[m];
     struct Triangle faces[n];
}

glBegin(GL_TRIANGLES)
     for ( i = 0 ; i < n ; i++ )
     {
          glVertex3fv(*(mesh.faces[i].verts[0]));
          glVertex3fv(*(mesh.faces[i].verts[1]));
          glVertex3fv(*(mesh.faces[i].verts[2]));
     }
glEnd();
```

Next Time

- More Modeling Technologies