Polygon Meshes

Def: a collection of polygons that form a surface that may or may not be closed.

The surface is generally an approximation, but it can be an exact representation:

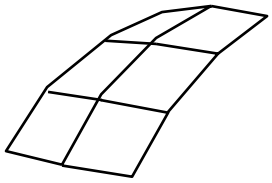
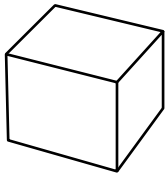


Fig 4.1 A general polygon mesh representing an approximation of a surface.

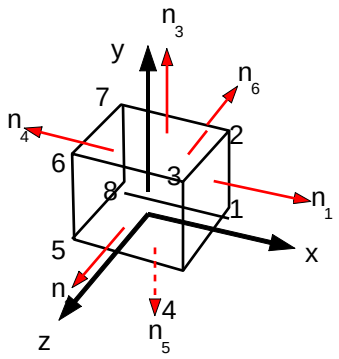


A polyhedron representing an exact object.

A polyhedron is a polygon mesh that encloses a finite amount of space.

Defining a Polygon Mesh

Start by defining all the vertices.



Vertex Index	x	y	z
1	1	0	0
2	1	1	0
3	1	1	1
4	1	0	1
5	0	0	1
6	0	1	1
7	0	1	0
8	0	0	0

Next, we could define each *face (polygonal patch on the polyhedron)* and list, for each face, all of its vertices. This results in a redundant structure that wastes space. A more efficient approach defines each face as a collection of indices into the vertex list.

Face Index	Vertices (indices into vertex list)
1	1,2,3,4
2	3,6,5,4
3	3,2,7,6
4	5,6,7,8
5	1,4,5,8
6	8,7,2,1

Next, we would define a list of normals for each face. In the case of the box, it is assumed to have flat sides of the polygonal mesh would have 6 distinct normals.

Normal Index	x	y	z
1	1	0	0
2	0	0	1
3	0	1	0
4	-1	0	0
5	0	-1	0
6	0	0	-1

Now, the faces would be defined as a list of indices into the vertex list and a list of normals into the normal list (one normal for each vertex in the face). In the case of the box, the same normal is used for each vertex in a face. The face list would then look like:

Face Index	Vertices for each face (indices into vertex list)	Corresponding normals (indices into the normal list)
1	1,2,3,4	1,1,1,1
2	3,6,5,4	2,2,2,2
3	3,2,7,6	3,3,3,3
4	5,6,7,8	4,4,4,4
5	1,4,5,8	5,5,5,5
6	8,7,2,1	6,6,6,6

Thus for a polygonal mesh, we end up with three lists (or three objects):

Vertex list: supplies geometric information
Normal list: supplies orientation information
Face list: supplies connectivity or topological information

Our author has made these three lists into three objects. Since every vertex in a face has an index into the vertex list and the normal list, he defines a VertexID object that contains a vertex index and the corresponding normal index. For example, in the first face in the table above, the vertex with the index 1 is has a normal with the index 1 and so on. The class is defined as:

```
class VertexID{
         public:
                  int vertIndex;              // index of this vertex in the vertex list
                  int normIndex;              // index of this vertex's normal
};
```

The second class is called Face and is used to represent one face of the mesh. It contains two data members – one keeps the number of vertices in a face (nVerts) and the second keeps a list of VertexID objects to maintain a face (vert) with vertices and corresponding normals. The default constructor assumes the list is empty. The destructor dynamically deletes the list of vertices and sets nVert to 0.

```
class Face{
  public:
                  int nVerts;        // number of vertices in this face
                  VertexID * vert;   // the list of vertex and normal indices
                  Face(){nVerts = 0; vert = NULL;}      // constructor
                  ~Face(){delete[] vert; nVerts = 0;}   // destructor
};
```

The last object manages the entire polygonal mesh. It contains a list of points that correspond to a vertex list (pt) and a list of normals that correspond to the normal list (norm). Finally it contains a list of faces that corresponds to the face list (face). It contains a constructor, destructor, a method to read mesh data from a file, and a method to draw the mesh after it has been read.

```
class Mesh{
   private:
                  int numVerts;               // number of vertices in the mesh
                  Point3* pt;                 // array of 3D vertices
                  int numNorms;               // number of normal vectors for the mesh
                  Vector3 *norm;              // array of normals
                  int numFaces;               // number of faces in the mesh
                  Face* face;                 // array of face data

   public:
                  Mesh();                     // constructor
                  ~Mesh();                    // destructor

                  // to read in a filed mesh
                  int readmesh(char * fileName);

                  // use OpenGL to draw this mesh
                  void draw();

};
```

The implementation of the constructor, destructor, readmesh and draw appear below

```
//The default constructor for the mesh class
//sets the number of vertices, normals, and
//faces to 0.
Mesh::Mesh()
{
         numVerts=0;
         numNorms=0;
         numFaces=0;
         norm=NULL;
         pt=NULL;
         face=NULL;
}

//The Mesh destructor releases all the space
//allocated to the mesh and sets the number
//of vertices, normals, and faces back to 0.
Mesh::~Mesh()
{
         delete[] pt;
         numVerts=0;
         delete[] norm;
         numNorms=0;
         delete[] face;
         numFaces=0;
}


//This function reads face information from a data file.
//The name of the file is passed to the function through
//the argument list
int Mesh:: readmesh(char * fileName)
{
         //open the file and check for file failure
         fstream infile;
         infile.open(fileName, ios::in);
         if(infile.fail()) return -1; // error - can't open file
         if(infile.eof())  return -1; // error - empty file

         //the file is OK so read the number of vertices,
         //normals, and faces.
         infile >> numVerts >> numNorms >> numFaces;

         //create arrays to hold the vertices, nomrmals,
         //and faces.
         pt = new Point3[numVerts];
         norm = new Vector3[numNorms];
         face = new Face[numFaces];

         //check that enough memory was found:
         if( !pt || !norm || !face)return -1;
```

```
          //read the vertices
          for(int p = 0; p < numVerts; p++)
          {
                    infile >> pt[p].x >> pt[p].y >> pt[p].z;

          }

          //read the normals
          for(int n = 0; n < numNorms; n++)
          {
                    infile >> norm[n].x >> norm[n].y >> norm[n].z;
          }

          //read the faces
          for(int f = 0; f < numFaces; f++)
          {
                    infile >> face[f].nVerts;
                    face[f].vert = new VertexID[face[f].nVerts];
                    for(int i = 0; i < face[f].nVerts; i++)
                              infile >> face[f].vert[i].vertIndex
                    >> face[f].vert[i].normIndex;
          }
          return 0; // success
}

//Draw the mesh.  Each face of the object is drawn
//using a different material property.
void Mesh:: draw()
{
          //set up the beginning material properties
          GLfloat mat_diffuse[] = {0.6, 0.6, 0.6, 1.0};
          GLfloat mat_specular[] = {1.0, 1.0, 1.0, 1.0};
          GLfloat mat_shininess[] = {50.0};
          glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, mat_specular);
          lMaterialfv(GL_FRONT_AND_BACK, GL_SHININESS, mat_shininess);

          //loop through the faces of the object
          for(int f = 0; f < numFaces; f++)
          {
                    //adjust the diffuse material property slightly for each face
                    mat_diffuse[2]=0.0 + f * .1; mat_diffuse[1] = 0.0 + .02 * f;
                    glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, mat_diffuse);

                    //draw the face as a filled polygon
                    glBegin(GL_POLYGON);
                      for(int v = 0; v < face[f].nVerts; v++) // for each one..
                      {
                                //find the next normal and vertex
                                int in = face[f].vert[v].normIndex ;   // index of this normal
                                int iv =  face[f].vert[v].vertIndex ;  // index of this vertex

                                //inform OpenGL of the normal and vertex
                                glNormal3f(norm[in].x, norm[in].y, norm[in].z);
                                glVertex3f(pt[iv].x, pt[iv].y, pt[iv].z);
                      }
                    glEnd();
          }
}
```

To use these classes to read a mesh and draw the corresponding shape, the following code would be used.

```
          //Create a polygonal mesh
          Mesh mymesh;

          //read the mesh from the file
          mymesh.readmesh("BarnMeshFile.txt");

          //draw the mesh
          mymesh.draw();
```

For a complete example, see the 3D mesh example on my web site.