Mesh representations and data structures

Luca Castelli Aleardi
Shared vertex representation
Half-edge DS
Winged edge
Triangle based DS
Corner Table
Shared vertex representation

class Point{
    double x;
    double y;
}

grouped information

class Vertex{
    Point p;
}

combinatorial information

class Face{
    Vertex[] vertices;
}

Memroy cost

$3 \times f = 6n$

Size (number of references)

Queries/Operations

- List all vertices or faces
- Test adjacency between $u$ and $v$
- Find the 3 neighboring faces of $f$
- List the neighbors of vertex $v$

for each face (of degree $d$), store:
- $d$ references to incident vertices
for each vertex, store:
- 1 reference to its coordinates

easy to implement
quite compact
not efficient for traversal

for each face (of degree $d$), store:
- $d$ references to incident vertices
for each vertex, store:
- 1 reference to its coordinates

easy to implement
quite compact
not efficient for traversal

for each face (of degree $d$), store:
- $d$ references to incident vertices
for each vertex, store:
- 1 reference to its coordinates

easy to implement
quite compact
not efficient for traversal

for each face (of degree $d$), store:
- $d$ references to incident vertices
for each vertex, store:
- 1 reference to its coordinates

easy to implement
quite compact
not efficient for traversal

for each face (of degree $d$), store:
- $d$ references to incident vertices
for each vertex, store:
- 1 reference to its coordinates

easy to implement
quite compact
not efficient for traversal

for each face (of degree $d$), store:
- $d$ references to incident vertices
for each vertex, store:
- 1 reference to its coordinates

easy to implement
quite compact
not efficient for traversal

for each face (of degree $d$), store:
- $d$ references to incident vertices
for each vertex, store:
- 1 reference to its coordinates

easy to implement
quite compact
not efficient for traversal

for each face (of degree $d$), store:
- $d$ references to incident vertices
for each vertex, store:
- 1 reference to its coordinates

easy to implement
quite compact
not efficient for traversal

for each face (of degree $d$), store:
- $d$ references to incident vertices
for each vertex, store:
- 1 reference to its coordinates

easy to implement
quite compact
not efficient for traversal

for each face (of degree $d$), store:
- $d$ references to incident vertices
for each vertex, store:
- 1 reference to its coordinates

easy to implement
quite compact
not efficient for traversal

for each face (of degree $d$), store:
- $d$ references to incident vertices
for each vertex, store:
- 1 reference to its coordinates

easy to implement
quite compact
not efficient for traversal

for each face (of degree $d$), store:
- $d$ references to incident vertices
for each vertex, store:
- 1 reference to its coordinates

easy to implement
quite compact
not efficient for traversal

for each face (of degree $d$), store:
- $d$ references to incident vertices
for each vertex, store:
- 1 reference to its coordinates

easy to implement
quite compact
not efficient for traversal

for each face (of degree $d$), store:
- $d$ references to incident vertices
for each vertex, store:
- 1 reference to its coordinates

easy to implement
quite compact
Half-edge data structure: polygonal (orientable) meshes

\[ f + 5 \times h + n \approx 2n + 5 \times (2e) + n = 32n + n \]
Size (number of references)

```
class Halfedge{
    Halfedge prev, next, opposite;
    Vertex v;
    Face f;
}
```

```
class Vertex{
    Halfedge e;
    Point p;
}
```

```
class Face{
    Halfedge e;
    }
```
Half-edge data structure: polygonal (orientable) meshes

\[ 3 \times h + n \approx 3 \times (2e) + n = 18n + n \]

Size (number of references)

```
class Point{
    double x;
    double y;
}
```

genermic information

```
class Halfedge{
    Halfedge prev, next, opposite;
    Vertex v;
    Face f;
}
```

combinatorial information

```
class Vertex{
    Halfedge e;
    Point p;
}
```

```
class Face{
    Halfedge e;
}
```
Half-edge data structure: efficient traversal

```java
public int vertexDegree(Vertex<X> v) {
    int result = 0;
    Halfedge<X> e = v.getHalfedge();

    Halfedge<X> pEdge = e.getNext().getOpposite();
    while (pEdge != e) {
        pEdge = pEdge.getNext().getOpposite();
        result++;
    }

    return result + 1;
}

public int degree() {
    Halfedge<X> e, p;
    if (this.halfedge == null) return 0;

    e = halfedge; p = halfedge.next;
    int cont = 1;
    while (p != e) {
        cont++;
        p = p.next;
    }

    return cont;
}
```
Half-edge data structure: efficient traversal

```java
public int vertexDegree(Vertex<X> v) {
    int result=0;
    Halfedge<X> e=v.getHalfedge();
    Halfedge<X> pEdge=e.getNext().getOpposite();
    while(pEdge!=e) {
        pEdge=pEdge.getNext().getOpposite();
        result++;
    }
    return result+1;
}

public int degree() {  
    Halfedge<X> e,p;
    if(this.halfedge==null) return 0;
    e=halfedge; p=halfedge.next;
    int cont=1;
    while(p!=e) {
        cont++;
        p=p.next;
    }
    return cont;
}
```
Half-edge data structure: efficient traversal

```java
public int vertexDegree(Vertex<X> v) {
    int result=0;
    Halfedge<X> e=v.getHalfedge();

    Halfedge<X> pEdge=e.getNext().getOpposite();
    while(pEdge!=e) {
        pEdge=pEdge.getNext().getOpposite();
        result++;
    }

    return result+1;
}

public int degree() {
    Halfedge<X> e,p;
    if(this.halfedge==null) return 0;

    e=halfedge; p=halfedge.next;
    int cont=1;
    while(p!=e) {
        cont++;
        p=p.next;
    }

    return cont;
}
```
Half-edge data structure: polygonal manifold meshes

can we represent them?

*yes*
Triangle based DS: for triangle meshes

(used in CGAL)

for each triangle, store:
- 3 references to neighboring faces
- 3 references to incident vertices

for each vertex, store:
- 1 reference to an incident face

\[(3 + 3) \times f + n = 6 \times 2n + n = 13n\]

Size (number of references)
Triangle based DS: mesh traversal operators

The data structure supports the following operators:

\[
\begin{align*}
& v = \text{vertex}(\Delta, i) \\
& \Delta = \text{face}(v) \\
& i = \text{vertexIndex}(v, \Delta) \\
& g_0 = \text{neighbor}(\Delta, i) \\
& g_1 = \text{neighbor}(\Delta, \text{ccw}(i)) \\
& g_2 = \text{neighbor}(\Delta, \text{cw}(i)) \\
& z = \text{vertex}(g_2, \text{faceIndex}(g_2, \Delta))
\end{align*}
\]

we can turn around a vertex, by combining the operators above

we can locate a point, by performing a walk in the triangulation
Triangle based DS: mesh update operators

The data structure supports the following operators:
- `removeVertex(v)`
- `splitFace(f)`
- `edgeFlip(e)`

The data structure is modifiable and all these operators can be performed in $O(1)$ time.