B-rep for Triangle Meshes

Gino van den Bergen
Dtecta

To the memory of

Jan Paul van Waveren (1977-2017)

B-reps in Games

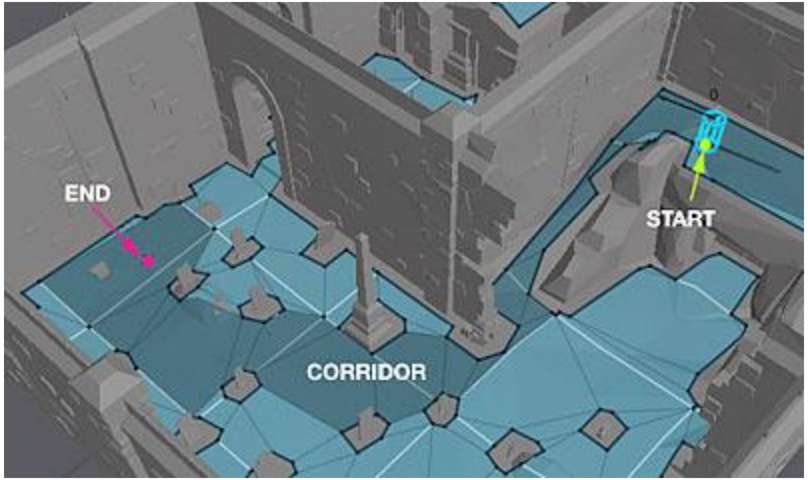
- Mesh cutting, e.g. Woodcutting in *Farming Simulator 15* and *Farming Simulator 17*.

B-reps in Games

- Incremental hull computation in *Quickhull* and *Expanding Polytope Algorithm* (EPA).

B-reps in Games

- Pathfinding on a navigation mesh.



Triangle Mesh

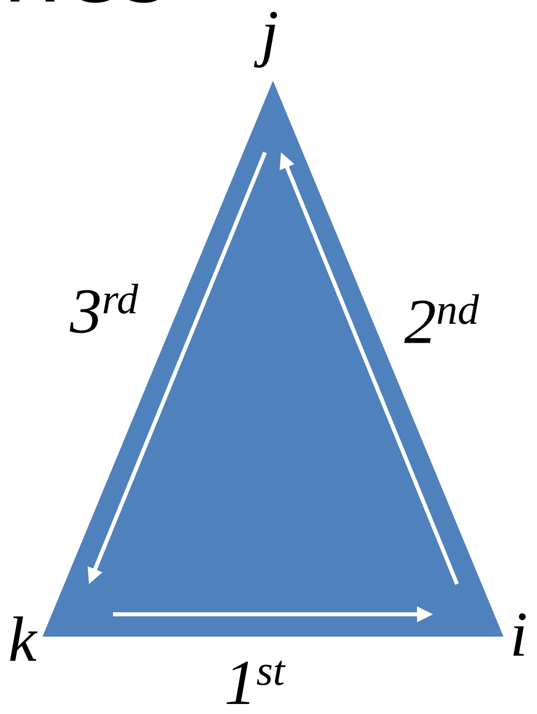
- Commonly stored as two arrays:
  - Array of vertices (xyz, uv, normals, etc.)
  - Array of triplets of indices into the vertex array.
- Finding neighboring vertices / adjacent faces involves O(n) search.

Boundary Representation

- A boundary representation (B-rep) offers O(1) retrieval of neighboring features.
- Examples of B-reps for polygon meshes are *winged-edge* and *half-edge* structure.
- Winged-edge-type structures are not the best choice for triangle meshes.

B-rep for Triangle Meshes

- A triangle given by index triplet (i, j, k) has its edges identified by:
  - 1st edge: (k, i)
  - 2nd edge: (i, j)
  - 3rd edge: (j, k)



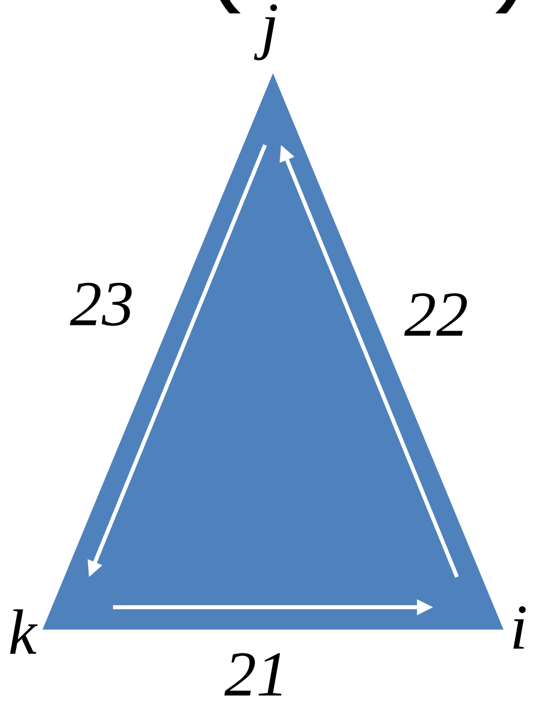
B-rep for Triangle Meshes (cont.)

- A B-rep triangle stores combined indices to its three adjacent half-edges.
- A (half-)edge is identified by a zero-based face index *f* and a one-based edge index *e* (1, 2, or 3).
- The combined half-edge index *h* is:

$$f * 4 + e.$$

B-rep for Triangle Meshes (cont.)

- Example: Suppose face index is 5, then half-edge indices are resp. 21, 22, and 23



B-rep for Triangle Meshes (cont.)

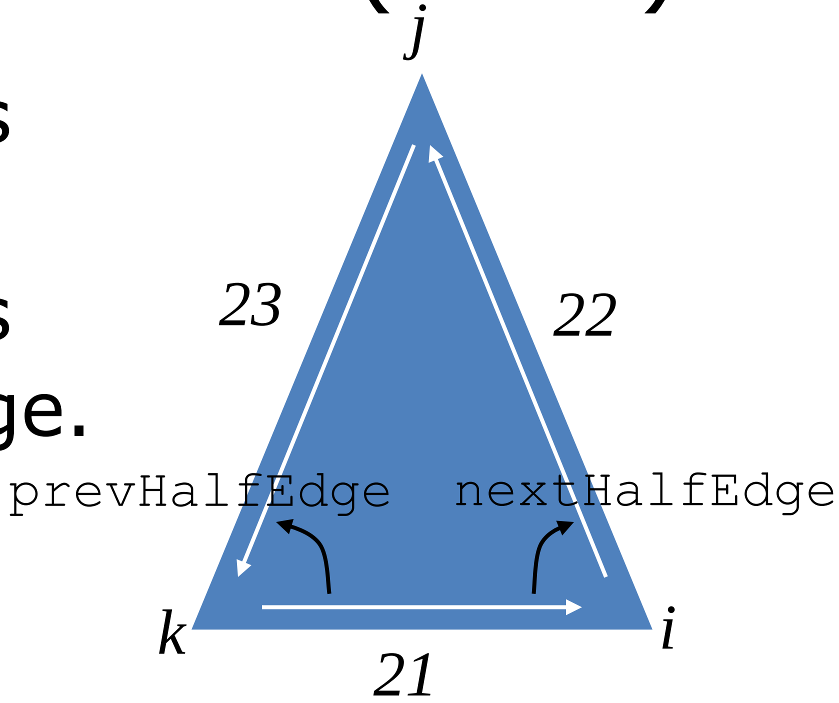
- Why don't we use a zero-based edge index and store h as $f * 3 + e$?
- Decomposition of h into f and e requires an integer division. Integer division by a power of two is cheaper using right shift.
- Rationale for one-based edge index follows…

B-rep for Triangle Meshes (cont.)

```
struct HalfEdge
{
    Index end; // end vertex index
    Index opp; // opposite half-edge
};
```

B-rep for Triangle Meshes (cont.)

- `nextHalfEdge`: returns next (CCW) half-edge.
- `prevHalfEdge`: returns previous (CW) half-edge.



B-rep for Triangle Meshes (cont.)

```
Index nextHalfEdge(Index h)
{
    ++h;
    return (h & 3) != 0 ? h : h - 3;
}
```

B-rep for Triangle Meshes (cont.)

```
Index prevHalfEdge(Index h)
{
    --h;
    return (h & 3) != 0 ? h : h + 3;
}
```

B-rep for Triangle Meshes (cont.)

- Note that no modulo (%) is used. Modulo of 3 involves an integer division.
- No branch either. Conditional expression (?:) will use conditional move (CMOV).
- One-based edge index requires comparison with zero. `(h & 3) != 0` is slightly cheaper than `(h & 3) != 3`.

B-rep for Triangle Meshes (cont.)

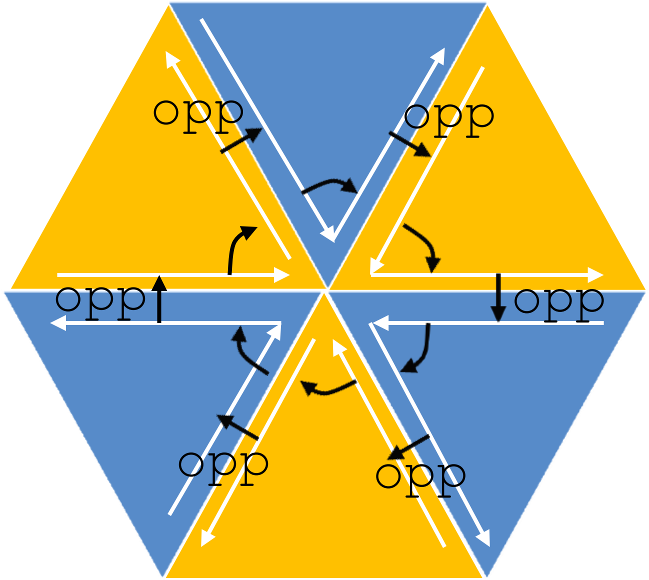
```
struct Face
{
    Index flags; // flag bits
    Index matId; // material ID
    HalfEdge edges[3]; // half-edges
};
```

B-rep for Triangle Meshes (cont.)

- We make sure that `sizeof(Face) == sizeof(HalfEdge) * 4`,
- And store all faces in a single array (`std::vector`) attribute `faces`.
- Then, `opp` can be used as an index into

```
reinterpret_cast<HalfEdge*>(&faces[0])
```

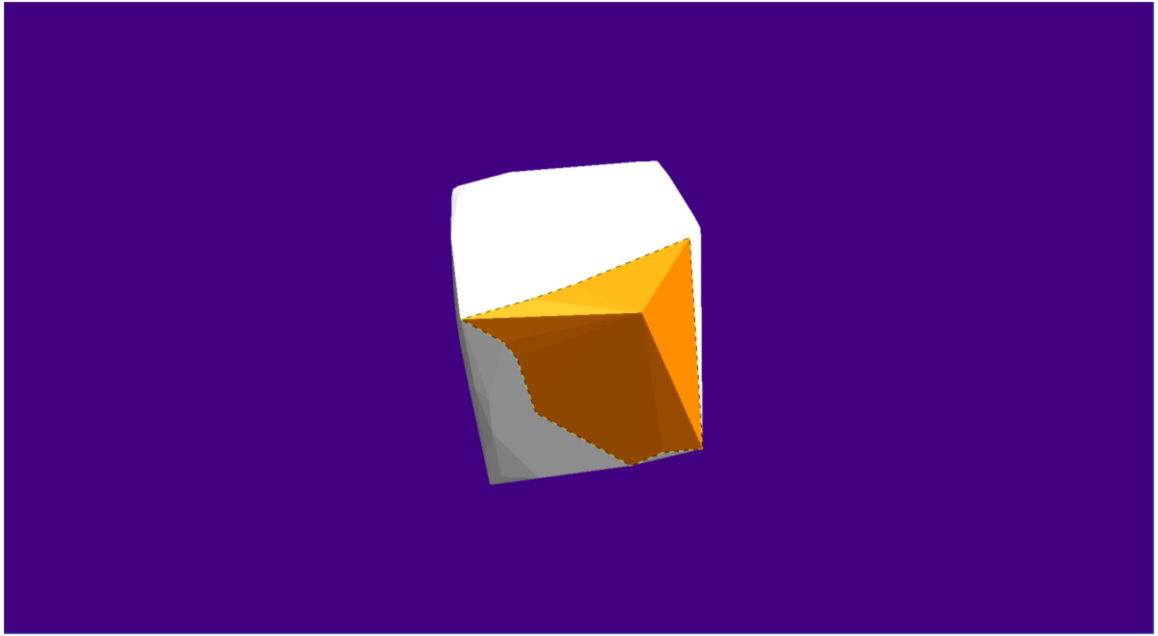
Incoming Half-Edges



Incoming Half-Edges (cont.)

```
Index h = first;
do
{
	…
	h = edgeAt(nextHalfEdge(h)).opp;
}
while (h != first);
```

Convex Silhouette



Convex Silhouette

```
void silhouetteMain(Index f, Vector3 p)
{
    faces[f].flags |= VISIBLE;
    for (Index e = 1; e != 4; ++e)
    {
        silhouette(edgeAt(f * 4 + e).opp, p);
    }
}
```

Convex Silhouette

```
void silhouette(Index h, Vector3 p)
{
    if ((faces[h / 4].flags & VISIBLE) == 0 &&
        faces[h / 4].isVisibleFrom(p))
    {
        faces[h / 4].flags |= VISIBLE;
        silhouette(edgeAt(nextHalfEdge(h)).opp, p);
        silhouette(edgeAt(prevHalfEdge(h)).opp, p);
    }
```

Quickhull

- Computes a B-rep for the convex hull of a point cloud.
- Pick three non-collinear points and form a B-rep by welding the triangle’s front and back.
- Enclose remaining points by forming a polyhedral cone (teepee) to the current B-rep’s silhouette for each point.

Quickhull (cont.)

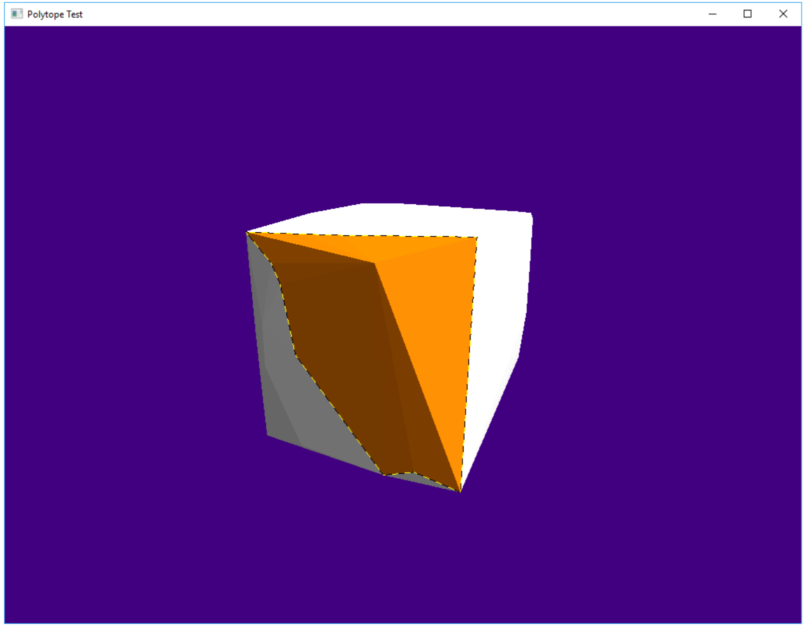
- Distribute set of points over faces based on containment in each face’s outside-half-space.
- For each face having outside-points, pick the point furthest from its face’s plane.
- Compute silhouette from this point and form a polyhedral cone.
- Repeat until all points are contained.

Quickerhull

- Maintain a priority queue of faces that have outside-points using the distance to the furthest point as priority.
- The face with furthest point goes first.
- Prioritizing results in fewer expansions and speeds up computations roughly by a factor of three.

Quickhull versus Quickerhull

- Demo



References

- Baumgart. *A polyhedron representation for computer vision*. Proc. AFIPS (1975)
- Rossignac, Safonova, Szymczak. *3D Compression Made Simple: Edgebreaker on a Corner-Table*. Proc. SMI (2001)
- Barber, Dobkin, Huhdanpaa. *The quickhull algorithm for convex hulls*. *ACM Transactions on Mathematical Software.* **22** (4): 469–483. (1996)
- Van den Bergen. *Collision Detection in Interactive 3D Environments.* Morgan Kaufmann Publishers (2003)

Thanks!

Check me out on

- Web: www.dtecta.com
- Twitter: @dtecta
- GitHub: https://github.com/dtecta