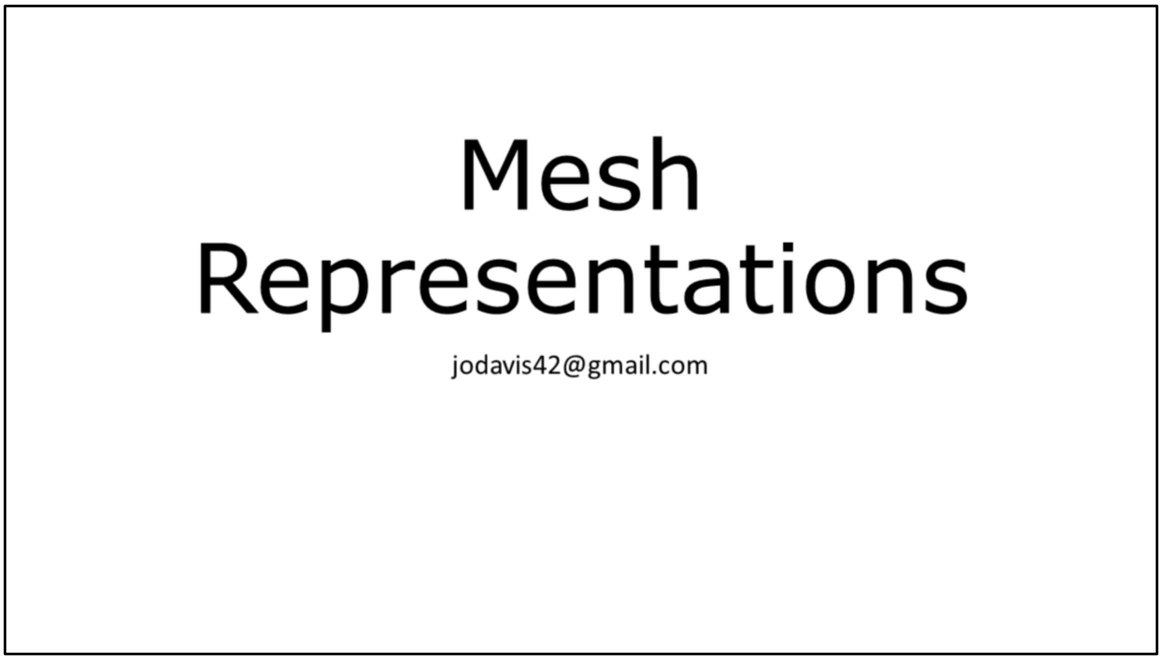
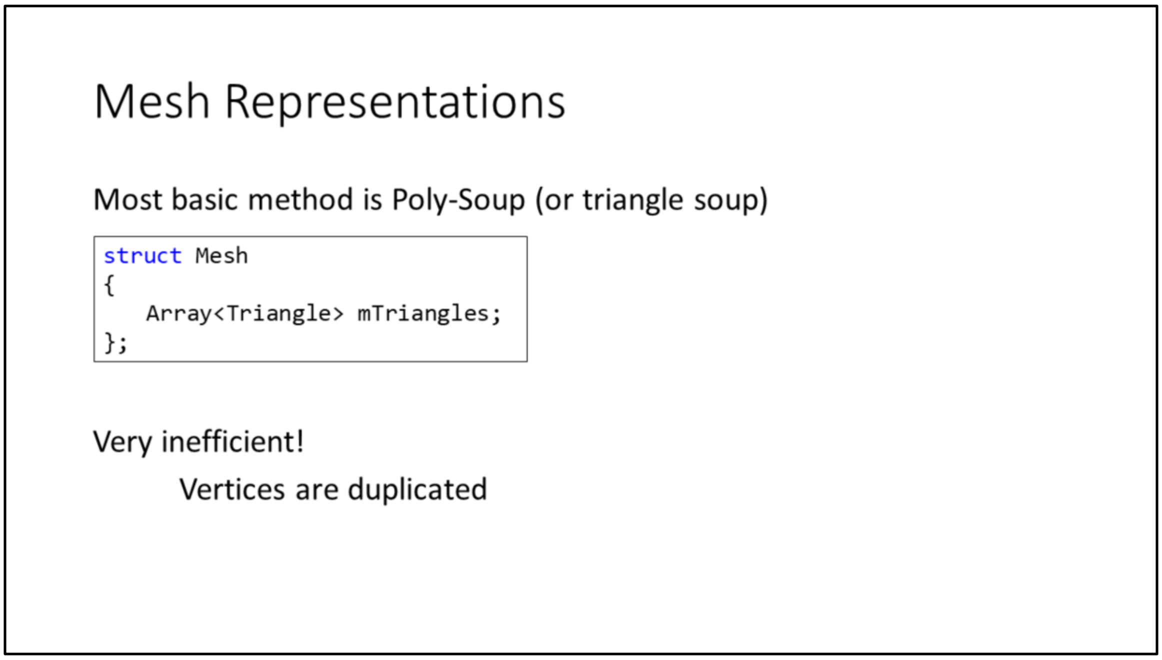


Mesh Representations

jodavis42@gmail.com



Mesh Representations

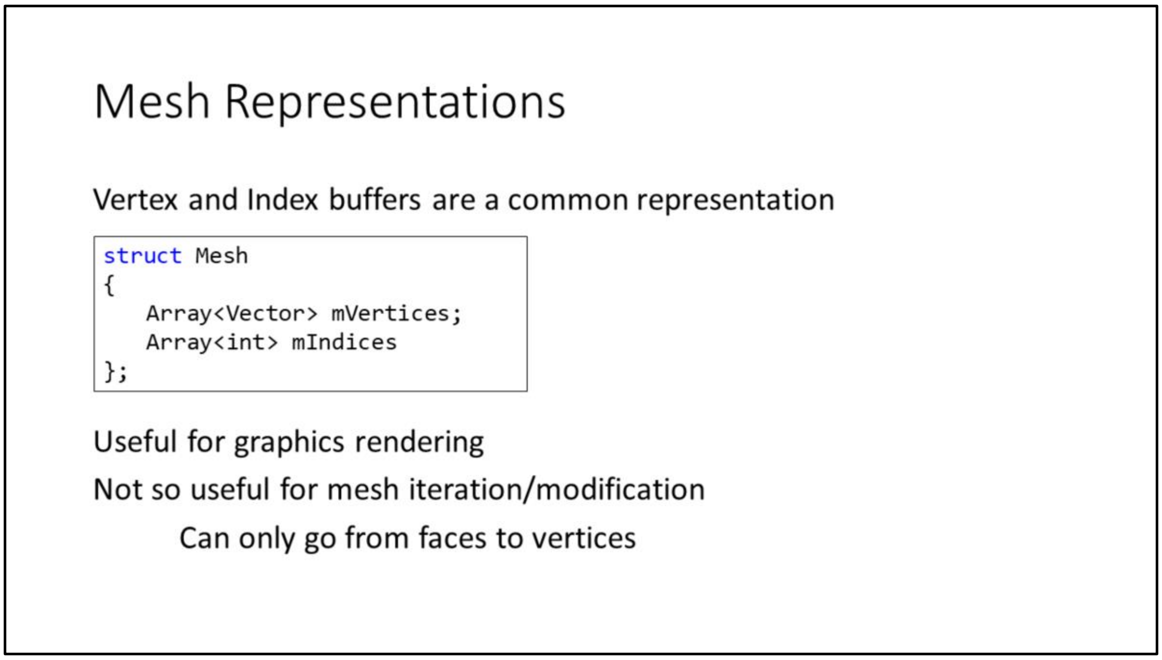
Most basic method is Poly-Soup (or triangle soup)

```
struct Mesh
{
   Array<Triangle> mTriangles;
};
```

Very inefficient!

    Vertices are duplicated

The first most basic method of representing meshes is what is often called poly-soup (although I'm only going to be considering triangles for most of this). Poly-soup is nothing more than an array of triangles where each triangle contains 3 points, each being a vector3.
This is incredibly inefficient as a data structure. For example, in a simple quad the two shared vertices between the two triangle will be stored twice.
So apart from its simplicity, this structure isn't often used.



Mesh Representations

Vertex and Index buffers are a common representation

```
struct Mesh
{
   Array<Vector> mVertices;
   Array<int> mIndices
};
```

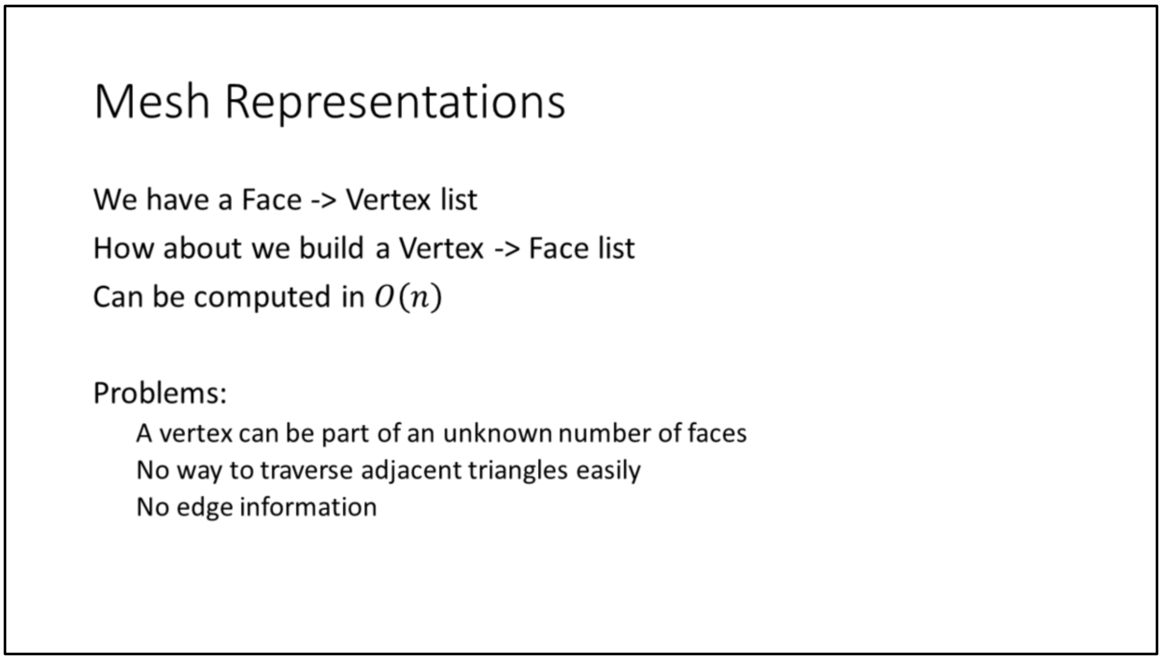
Useful for graphics rendering

Not so useful for mesh iteration/modification

Can only go from faces to vertices

The next mesh representation is perhaps one of the most common ones as it's how graphics cards tend to want their data. In this case if a vertex is duplicated between two triangles then we only need one extra integer, so only 4 extra bytes instead of 12.

While this representation is very memory efficient and even good for graphics, it doesn't lend itself well towards mesh manipulation. We can easily go from a face to its vertices, but not vice-versa.



Mesh Representations

We have a Face -> Vertex list

How about we build a Vertex -> Face list

Can be computed in $O(n)$

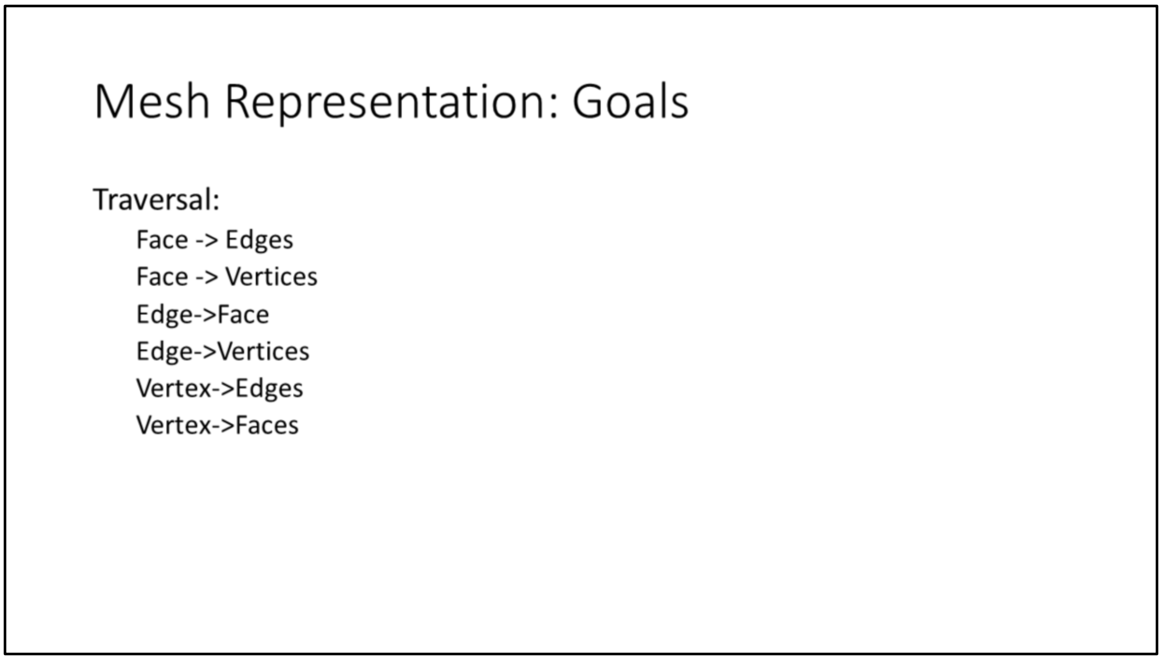
Problems:

- A vertex can be part of an unknown number of faces
- No way to traverse adjacent triangles easily
- No edge information

One potential solution is to try and build a vertex to face list. Unfortunately vertices can have an unbounded number of faces that they're a part of. Because of this a vertex must contain an array or some other form of resizable container.

In either case, this reverse lookup table can be computed in n time by just walking through each face and adding that face to each of its vertices lists.

There are a few core problems with this though, namely that we have no edge information. For example, edges are useful to try and find triangles that are adjacent. We can walk all of the triangles of all of the vertices of a given triangle, but this can produce many more triangles than we want and pruning this list is annoying. Because of these reasons, this approach isn't often used. Instead we tend to want some form of an edge-centric representation.



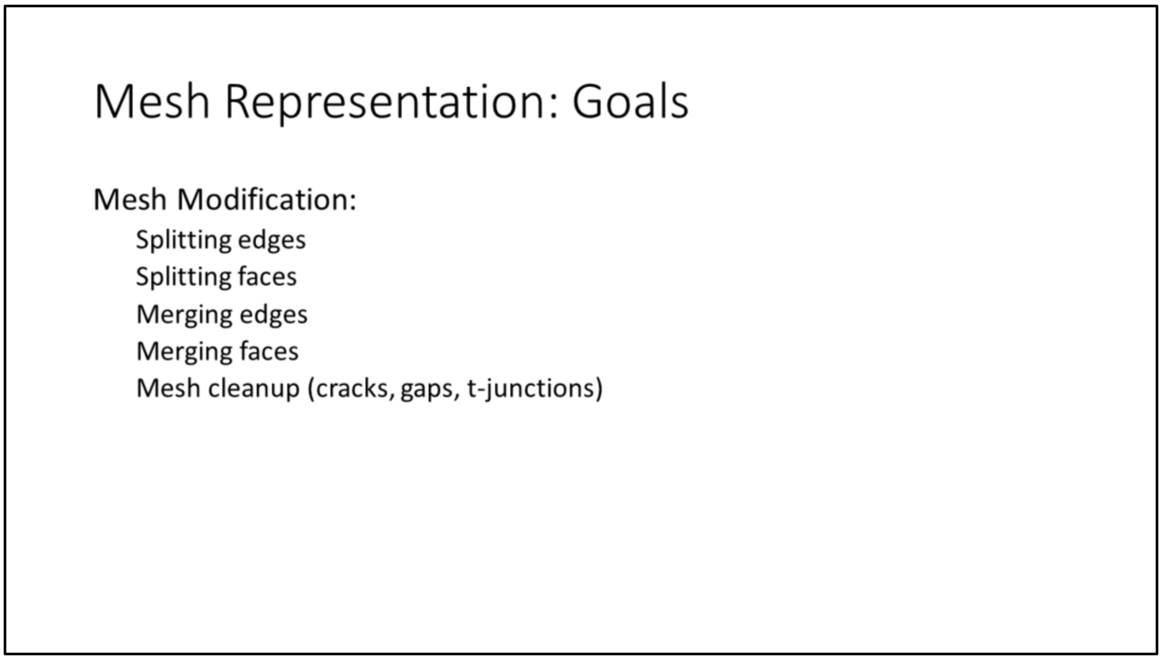
Mesh Representation: Goals

Traversal:

- Face -> Edges
- Face -> Vertices
- Edge->Face
- Edge->Vertices
- Vertex->Edges
- Vertex->Faces

This leads us naturally to look at what it is we want to be able to do with our mesh. When it comes to traversing through the mesh we pretty much want to be able to iterate from any feature to any other adjacent feature in $O(1)$ time.

In particular, we want to go from triangles to edges to vertices and from edges to faces. Sometimes we leave off traversal from a vertex to edges.



Mesh Representation: Goals

Mesh Modification:

- Splitting edges
- Splitting faces
- Merging edges
- Merging faces
- Mesh cleanup (cracks, gaps, t-junctions)

The other important thing for us to be able to do with a mesh is efficiently modify it. There's two main operations that we do when modifying a mesh: splitting and merging. We might be splitting a mesh when we slice it with a plane or something similar. In this case we'll need to efficiently split edges and faces.

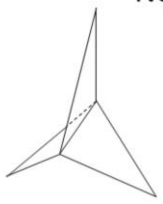
Merging is another common operation for removing redundant or numerical unstable information. In particular, this is often needed to cleanup a mesh for use in things like a physics engine. One such use we'll see later when discussing convex hulls.

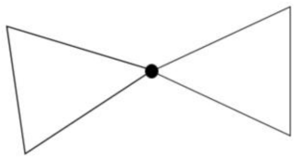
Mesh Representation: Restrictions

Manifold Meshes only:

- Only 2 faces per edge
- Faces incident to a vertex form a closed or open fan

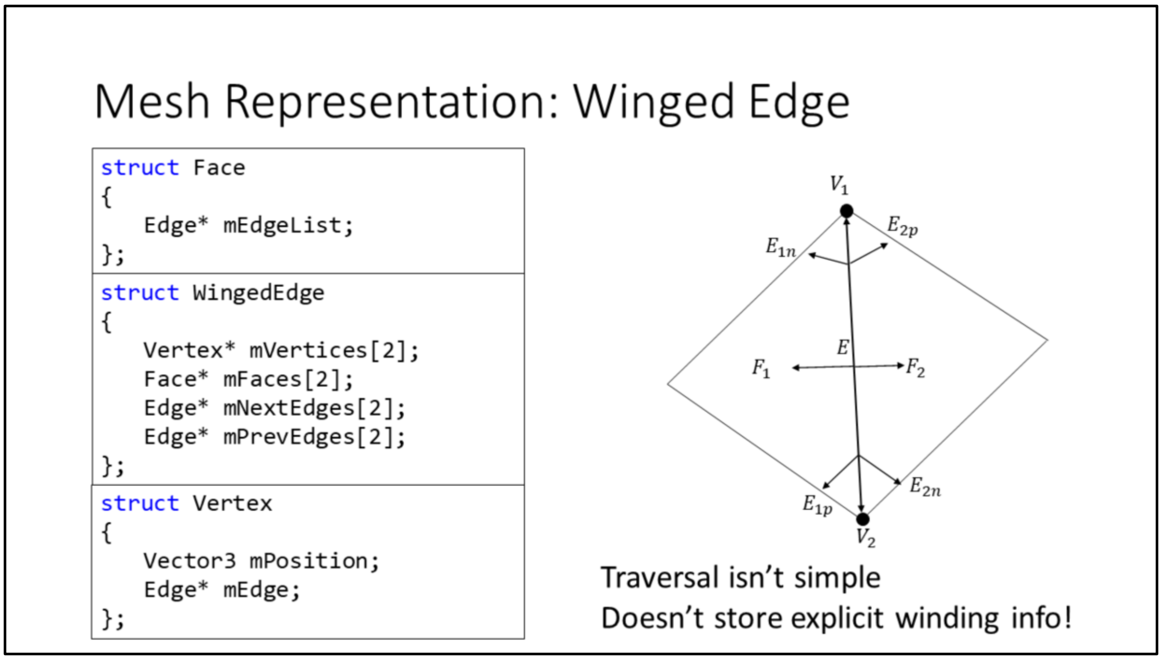
Non-Manifold examples:





One important restriction we place is having what is called a “manifold” mesh. One of the rules of a manifold mesh is that there can only be two triangles per edge. An counter-example to this can be seen on the left. This is the easiest restriction to lift, but we typically make it because it allows us to not have a dynamic array for the number of faces adjacent to an edge.

The other restriction for manifold meshes is harder to explain. All the faces adjacent to a vertex must either form a closed or open fan. What this basically means is that there can’t be triangles connected to a vertex that don’t all connect via some edges. A simple example of this is a bow-tie.

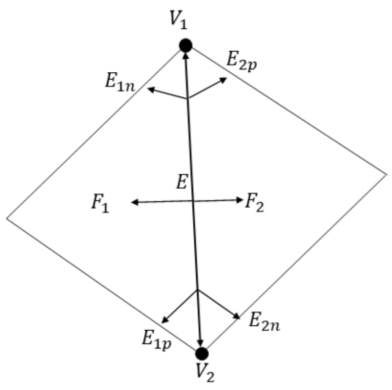


Mesh Representation: Winged Edge

```
struct Face
{
   Edge* mEdgeList;
};
```

```
struct WingedEdge
{
   Vertex* mVertices[2];
   Face* mFaces[2];
   Edge* mNextEdges[2];
   Edge* mPrevEdges[2];
};
```

```
struct Vertex
{
   Vector3 mPosition;
   Edge* mEdge;
};
```

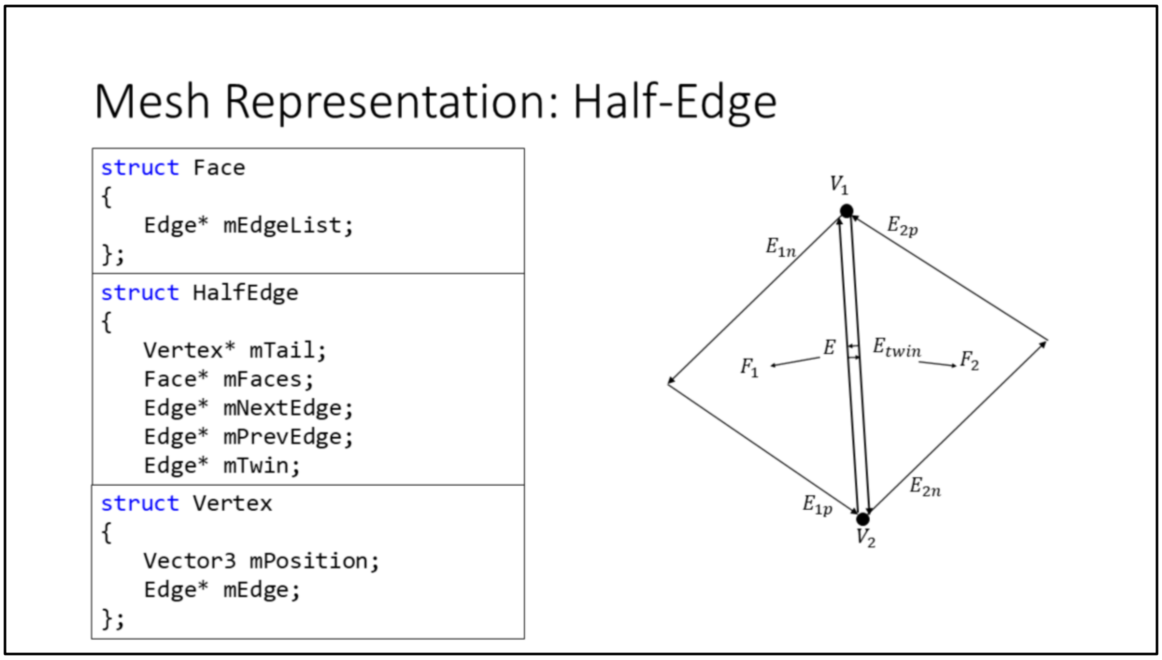


Traversal isn't simple
Doesn't store explicit winding info!

One such representation for meshes is called the winged edge data structure. In this structure an edge contains both faces it's adjacent to, both vertices it's connected to, and the next and previous edges for both faces.

With this traversal we can easily go from a face to an edge to a vertex and from an edge to any face or adjacent faces.

The only real problem with this is that we don't store explicit winding information. We can implicitly store our next and previous such that each triangle's winding order is maintained but traversal isn't simple since a face can only traverse to an adjacent face by checking which one of two faces it is on the edge.

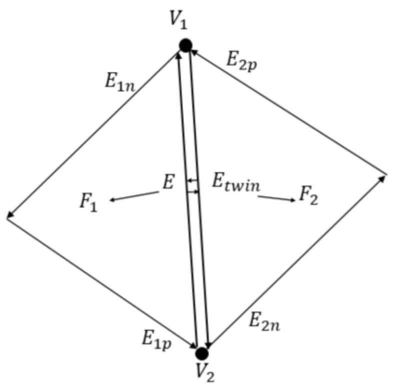


Mesh Representation: Half-Edge

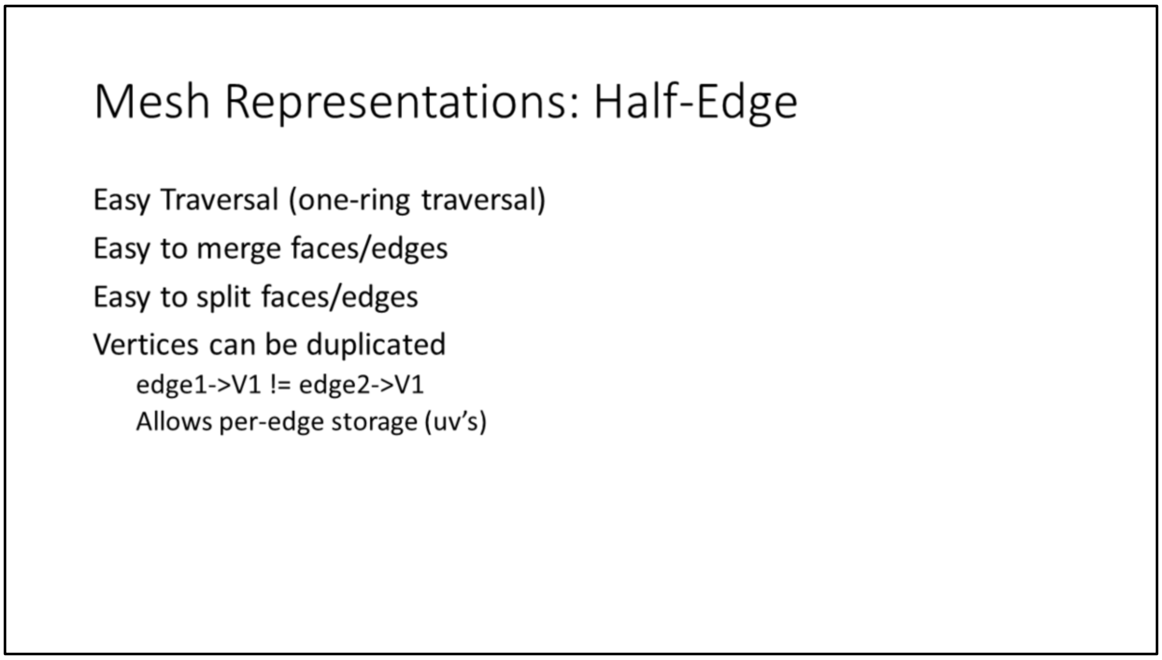
```
struct Face
{
    Edge* mEdgeList;
};
```

```
struct HalfEdge
{
    Vertex* mTail;
    Face* mFaces;
    Edge* mNextEdge;
    Edge* mPrevEdge;
    Edge* mTwin;
```

```
struct Vertex
{
    Vector3 mPosition;
    Edge* mEdge;
};
```



One of the more common mesh representations is the half-edge mesh. The reason it's called the half-edge is because we split our edge into two pieces, one for each face. Each edge then stores a pointer to it's twin edge which allows us to traverse from one face to another.



Mesh Representations: Half-Edge

Easy Traversal (one-ring traversal)

Easy to merge faces/edges

Easy to split faces/edges

Vertices can be duplicated

- edge1->V1 != edge2->V1
- Allows per-edge storage (uv's)

There's a number of useful very useful things about the half-edge mesh.
First of all we can easily traverse from across an entire mesh, especially the one-ring traversal.
It is a little more memory intensive compared to the winged-edge mesh, but only minimally so.
This representation is also very easy to manipulate a mesh with.

Perhaps one of the more interesting benefits is that we can duplicate vertices whenever we want for non-smooth interpolants, for example uv's on a mesh.