GDC 13

Computational Geometry

(representation and manipulation)

Graham Rhodes
Senior Software Developer, Applied Research Associates, Inc.

GAME DEVELOPERS CONFERENCE
SAN FRANCISCO, CA
MARCH 25-29, 2013
EXPO DATES: MARCH 27-29
2013

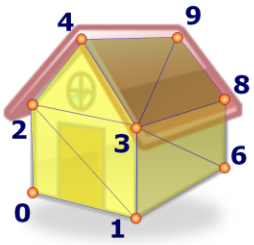
Outline

- Part 1: Explicit boundary representations
  - Half edge data structure
  - Editing geometry on the CPU
- Part 2: Higher order surface modeling
  - Subdivision surfaces using half edge data structure
  - Higher order surfaces on the GPU

Part 1: Explicit Boundary Representations

Some Perspective

- In-game models usually made of meshes
  - Typically triangles
  - Indexed triangle meshes



Vertex Coords	
0	<-.5, 0, 0>
1	<0, 0, 0>
2	<-.5, 0, .5>
3	<0, 0, .5>
4	<-.25, 0, 1>
5	<-.5, 1, 0>
6	<0, 1, 0>
7	<-.5, 1, .5>
8	<0, 1, .5>
9	<-.25, 1, 1>

Triangles
0, 1, 2
1, 3, 2
2, 3, 4
1, 6, 3
3, 6, 8
3, 8, 9
3, 9, 4
...

Is a triangle mesh enough?

- Good for rendering "fixed" geometry
- Maybe not always
  - Inconvenient for editing a mesh
  - Inconvenient for subdivision modeling
  - Inconvenient whenever the triangle connectivity (topology) needs to change

By "fixed" I mean the topology is fixed. Model can be dynamic in that it moves either due to a changing overall transformation matrix or via some per-control vertex transformation such as skinned animation.

Data Structures for Editable Meshes

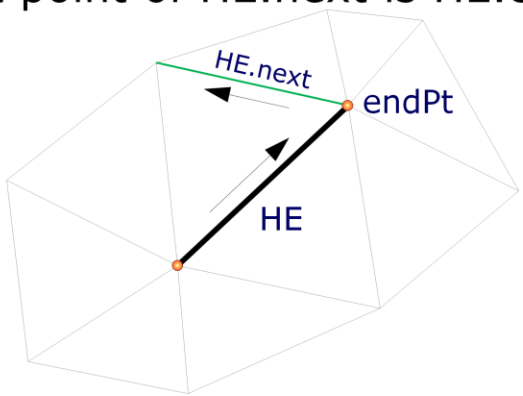
- Manifold
  - Winged Edge (Baumgart, 1972)
  - Half Edge (presented here)
  - Quad edge
- Non-manifold
  - Radial edge
  - Generalized non-manifold
    - (some game DCC tools, CAD, …)

Generalized non-manifold is the type of data structure used in computer aided design software. It completely separates geometry and topology, and is much more rigorous that what we need to be concerned with for games. It is also far more difficult to implement. The complete division between geometry and topology makes this quite non-intuitive.

The open source modeling software, Blender, and other digital content creation tools used in the game industry, are based on non-manifold data topology structures.

Half Edge Data Structure (HEDS)

- HE points to next half edge in traversal direction
- Start point of HE.next is HE.endPt

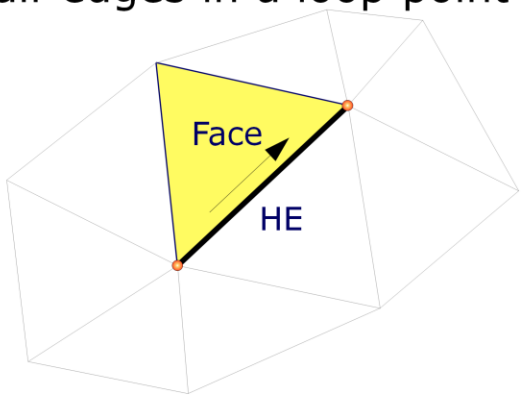



```
struct HalfEdge
{
  HalfEdgeVert *endPt;
  HalfEdge *next;
};
```

Note that for a close polygon, such as a triangle, we can find traverse the polygon's boundary loop simply using HE.next recursively.

Half Edge Data Structure (HEDS)

- HE ***may*** point to a face on its **left** side
- All half edges in a loop point to same face

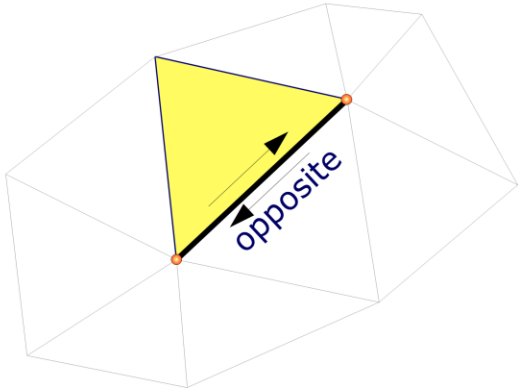



```
struct HalfEdge
{
  HalfEdgeVert *endPt;
  HalfEdge *next;
  HalfEdgeFace *face;
};
```

The truth is, the face is on the left side only depending on viewpoint. If we look at the half edge from a point-of-view where the loop is traversed in a counterclockwise fashion, the face is on the left of the edge….while walking along the edge we would turn towards the left to see the face. If we looked at this same object from behind, the face would appear to be on the right.

Half Edge Data Structure (HEDS)

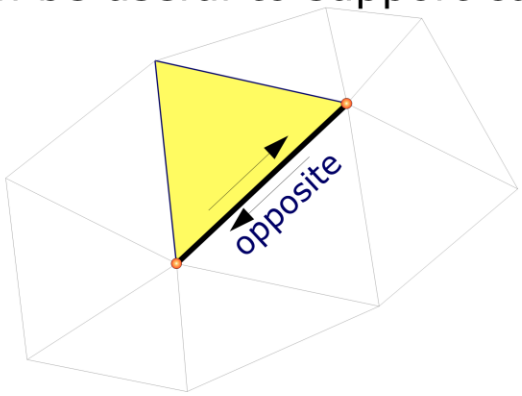
- HE points to its opposite half edge
- Which is attributed as above




```
struct HalfEdge
{
  HalfEdgeVert *endPt;
  HalfEdge *next;
  HalfEdgeFace *face;
  HalfEdge *opposite;
};
```

Half Edge Data Structure (HEDS)

- HE points to its opposite half edge
- It can be useful to support custom data




```
struct HalfEdge
{
  HalfEdgeVert *endPt;
  HalfEdge *next;
  HalfEdgeFace *face;
  HalfEdge *opposite;
  void *userData;
  unsigned char marker;
};
```

Note that due to the orientation flip of the opposite edge, the opposite face has the same orientation as the original half edge's face. Orientation consistency is built into the data structure.

C++ half edge class definitions

```
struct HalfEdge
{
  HalfEdgeVert *endPt;
  HalfEdge *next;
  HalfEdge *opposite;
  HalfEdgeFace *face;
  void *userData;
  unsigned char marker;
};
```

```
struct HalfEdgeFace
{
  HalfEdge *halfEdge;
  unsigned char marker;
};
struct HalfEdgeVert
{
  HalfEdge *halfEdge;
  int index;
  unsigned char marker;
};
```

We are focusing on the half edge, but typical implementations also define special face and vertex data structures. These enable additional traversals that are useful.

The user data could be assigned to the edge, face, and/or vertex. It could store, for example, texture or UV mapping information.

The marker is useful to aid in traversals. For example, if you want to find he constellation of faces around a given starting face, then traverse around the face's loop. For each vertex around the face, find the ring of faces around that vert, but skip any face that has a marker value of 1. For any as-yet-unmarked face, add it to your list, then set marker = 1 for that face. By using the marker in this way, it indicates that you've already visited a face and so it is already in your output list. You can also use this for Boolean type searches. For example, if you want to find faces connected to vert1, but not to vert2, first find the ring of faces around vert2, and set marker to 1. Then find the ring of faces around vert1, skipping

any face with marker == 1. These are simple examples, but it should be clear that marker can enable rather complex selection logic.

The marker can also aid in supporting selection modes. For example, marker == 0 for non-selected items, and marker == 1 for selected items.

You could consider treating the marker as a bitfield, with some bits used for selection, some used to indicate traversal status, some indicating constraints (e.g., crease/corner edge), etc.

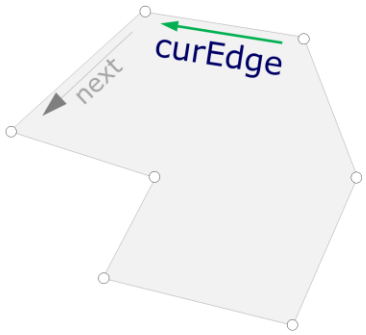
HDS Invariants

- Strict
  - halfEdge != halfEdge->opposite
  - halfEdge != halfEdge->next
  - halfEdge == halfEdge->opposite->opposite
  - startPt(halfEdge) == halfEdge->opposite->endPt
  - There are a few others…
- Implement in code for convenience
  - Vertex == Vertex->halfEdge->endPt

Simple Traversals

Find vertex loop defined by a half edge

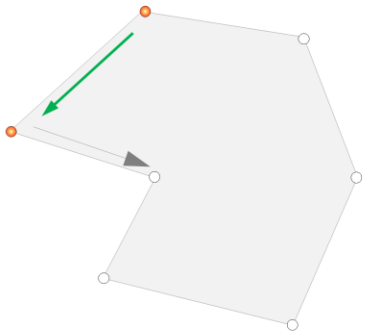
```
IndexList FindVertexLoop(HalfEdge *edge)
{
  IndexList loop;
  HalfEdge *curEdge = edge;
  do {
    loop.push_back(edge.endPt->index);
    curEdge = curEdge->next;
  } while (curEdge != edge);
  return loop;
};
```



Simple Traversals

Find vertex loop defined by a half edge

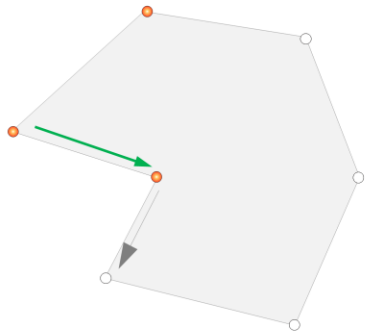
```
IndexList FindVertexLoop(HalfEdge *edge)
{
  IndexList loop;
  HalfEdge *curEdge = edge;
  do {
    loop.push_back(edge.endPt->index);
    curEdge = curEdge->next;
  } while (curEdge != edge);
  return loop;
};
```



Simple Traversals

Find vertex loop defined by a half edge

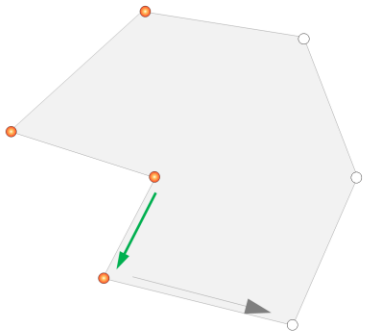
```
IndexList FindVertexLoop(HalfEdge *edge)
{
  IndexList loop;
  HalfEdge *curEdge = edge;
  do {
    loop.push_back(edge.endPt->index);
    curEdge = curEdge->next;
  } while (curEdge != edge);
  return loop;
};
```



Simple Traversals

Find vertex loop defined by a half edge

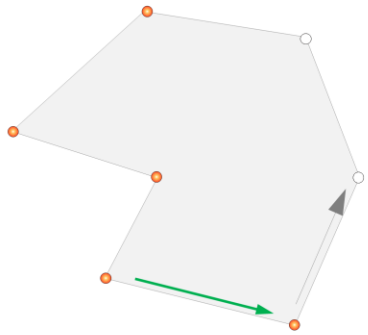
```
IndexList FindVertexLoop(HalfEdge *edge)
{
  IndexList loop;
  HalfEdge *curEdge = edge;
  do {
    loop.push_back(edge.endPt->index);
    curEdge = curEdge->next;
  } while (curEdge != edge);
  return loop;
};
```



Simple Traversals

Find vertex loop defined by a half edge

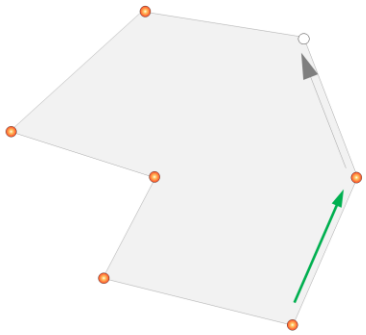
```
IndexList FindVertexLoop(HalfEdge *edge)
{
  IndexList loop;
  HalfEdge *curEdge = edge;
  do {
    loop.push_back(edge.endPt->index);
    curEdge = curEdge->next;
  } while (curEdge != edge);
  return loop;
};
```



Simple Traversals

Find vertex loop defined by a half edge

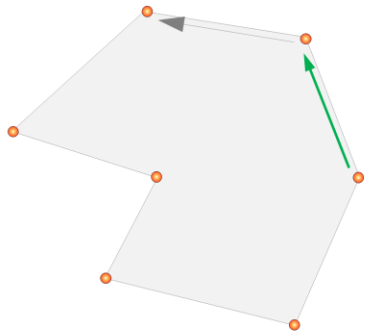
```
IndexList FindVertexLoop(HalfEdge *edge)
{
  IndexList loop;
  HalfEdge *curEdge = edge;
  do {
    loop.push_back(edge.endPt->index);
    curEdge = curEdge->next;
  } while (curEdge != edge);
  return loop;
};
```



Simple Traversals

Find vertex loop defined by a half edge

```
IndexList FindVertexLoop(HalfEdge *edge)
{
  IndexList loop;
  HalfEdge *curEdge = edge;
  do {
    loop.push_back(edge.endPt->index);
    curEdge = curEdge->next;
  } while (curEdge != edge);
  return loop;
};
```

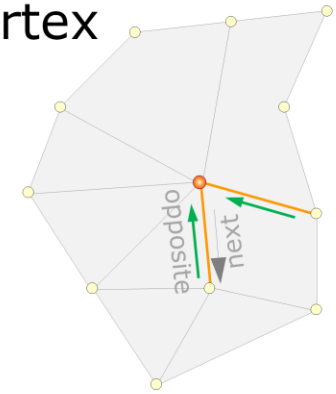


Note that we add each new edge in constant time, so the net cost is O(n), where n is the number of edges in the loop.

Simple Traversals

Find edges of 1-ring around vertex

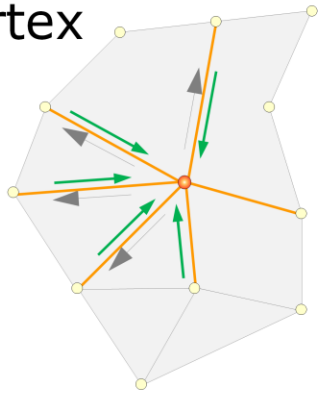
```
EdgeList Find1RingEdges(HalfEdgeVert *vert)
{
  EdgeList ring;
  HalfEdge *curEdge = vert->halfEdge;
  do {
    ring.push_back(curEdge);
    curEdge = curEdge->next->opposite;
  } while (curEdge != vert->halfEdge);
  return ring;
};
```



Simple Traversals

Find edges of 1-ring around vertex

```
EdgeList Find1RingEdges(HalfEdgeVert *vert)
{
  EdgeList ring;
  HalfEdge *curEdge = vert->halfEdge;
  do {
    ring.push_back(curEdge);
    curEdge = curEdge->next->opposite;
  } while (curEdge != vert->halfEdge);
  return ring;
};
```



"Valence" : number of edges in the 1-ring neighborhood of vertex

The collection of faces that immediately touch the vertex of interest is called the "1-ring neighborhood"

Supposed you needed to find all faces connected to a collection of vertices

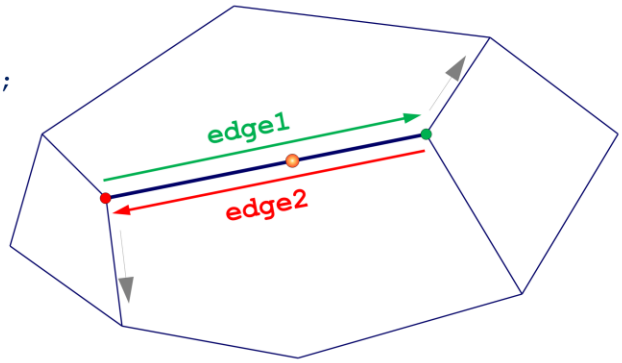
You can use the approach shown here to collect faces for each vertex

Use marker values to avoid collecting a given face more than once

Simple Operations

Split an edge

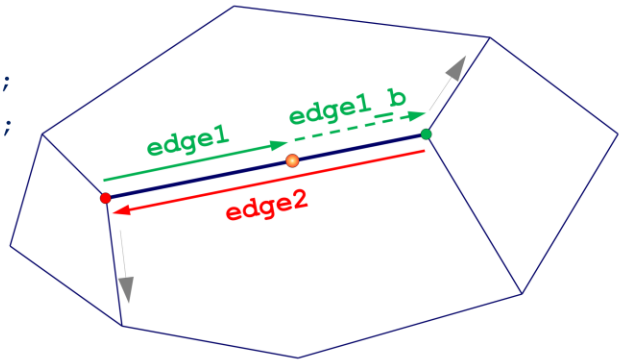
```
HalfEdge edge1;
HalfEdge edge2 = edge1.opposite;
```



Simple Operations

Split an edge

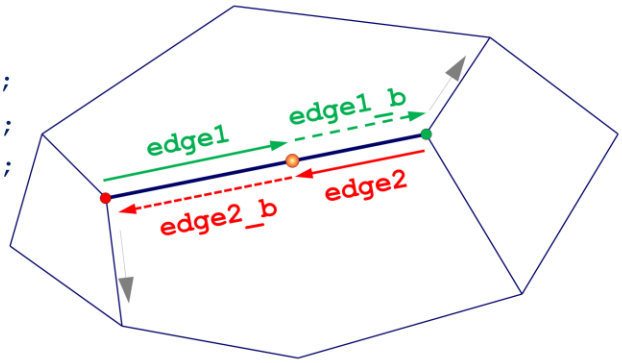
```
HalfEdge edge1;
HalfEdge edge2 = edge1.opposite;
HalfEdge edge1_b = new HalfEdge;

edge1_b.EndPt = edge1.EndPt;
edge1_b.face = edge1.face;
edge1_b.next = edge1.next;
edge1.EndPt = splitVert;
edge1.next = edge1_b;
edge1_b.EndPt.halfEdge = edge1_b;
```



Simple Operations

Split an edge

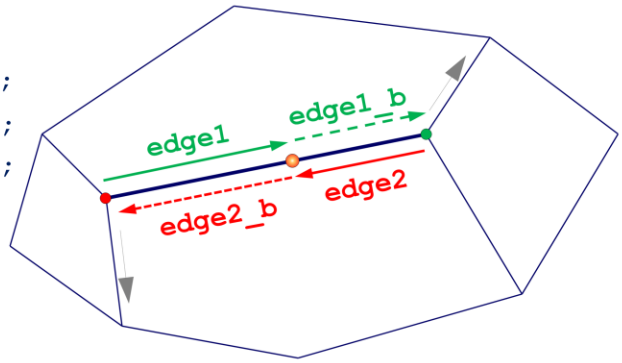
```
HalfEdge edge1;
HalfEdge edge2 = edge1.opposite;
HalfEdge edge1_b = new HalfEdge;
HalfEdge edge2_b = new HalfEdge;

edge2_b.EndPt = edge2.EndPt;
edge2_b.face = edge2.face;
edge2_b.next = edge2.next;
edge2.EndPt = splitVert;
edge2.next = edge2_b;
edge2_b.EndPt.halfEdge = edge2_b;
```



Simple Operations

Split an edge

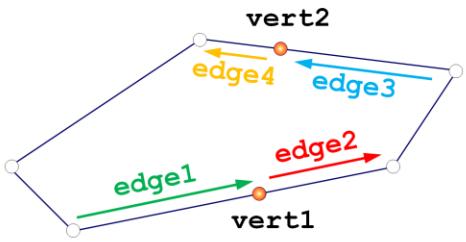
```
HalfEdge edge1;
HalfEdge edge2 = edge1.opposite;
HalfEdge edge1_b = new HalfEdge;
HalfEdge edge2_b = new HalfEdge;

edge2_b.opposite = edge1;
edge2.opposite = edge1_b;
edge1_b.opposite = edge2;
edge1.opposite = edge2_b;
splitVert.halfEdge = edge1;
```



Simple Operations

Split a face

```
HalfEdge edge1 = vert1.halfEdge;
HalfEdge edge2 = edge1.next;
HalfEdge edge3 = vert2.halfEdge;
HalfEdge edge4 = edge3.next;
```



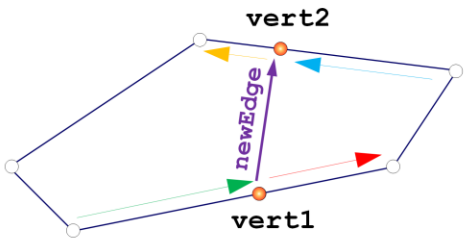
*See speaker notes below slide for an important consideration!

IMPORTANT NOTE: If the face is part of a mesh, then edge1 is not necessarily the only edge whose endPt is vert1. Similarly, edge3 is not necessarily the only edge whose endPt is vert2. **So, in the case of splitting a face in a mesh, it may be necessary to traverse the ring of edges around vert1 (and vert2) to find the edge whose endPt is vert1 (vert2) and whose face is the face of interest.**

Simple Operations

Split a face

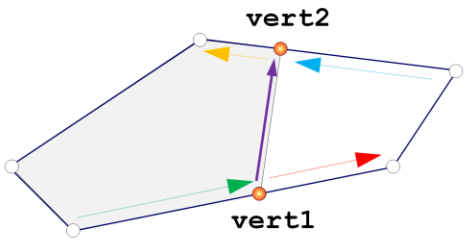
```
HalfEdge edge1 = vert1.halfEdge;
HalfEdge edge2 = edge1.next;
HalfEdge edge3 = vert2.halfEdge;
HalfEdge edge4 = edge3.next;
HalfEdge newEdge = new HalfEdge;

edge1.next = newEdge;
newEdge.next = edge4;
newEdge.face = edge1.face;
newEdge.endPt = vert2;
```



Simple Operations

Split a face

```
HalfEdge edge1 = vert1.halfEdge;
HalfEdge edge2 = edge1.next;
HalfEdge edge3 = vert2.halfEdge;
HalfEdge edge4 = edge3.next;
HalfEdge newEdge = new HalfEdge;
```

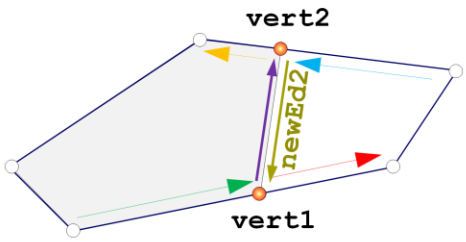



```
edge1.next = newEdge;
newEdge.next = edge4;
newEdge.face = edge1.face;
newEdge.endPt = vert2;
edge1.face.halfEdge = edge1;
```

Simple Operations

Split a face

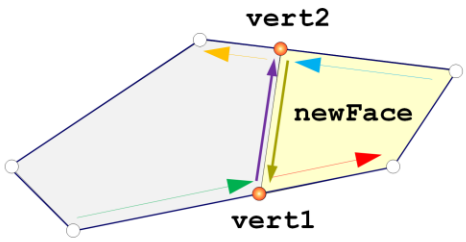
```
HalfEdge newEd2 = new HalfEdge;

newEd2.next = edge2;
newEd2.endPt = vert1;
edge3.next = newEd2;

newEdge.opposite = newEd2;
newEd2.opposite = newEdge;
```



Simple Operations
Split a face

```
newFace = new HalfEdgeFace
newFace.halfEdge = edge2;
```

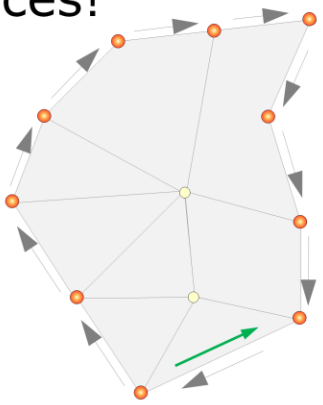
```
HalfEdge *curEdge = edge2;
do {
  curEdge->face = newFace;
  curEdge = curEdge->next;
} while (curEdge != edge2);
```



Pop Quiz!

Find the open boundary vertices!

```
IndexList Boundary;
Boundary =
        FindVertexLoop(startEdge->opposite);
```



(But what if the boundary isn't connected properly?)

*See speaker notes below slide for an important consideration!

Caution! If the outer edges weren't connected properly to begin with, will have to traverse edge rings (see following slides) for each boundary vertex to locate the boundary edges from the inside. This is more expensive. Best to make sure the data structure is properly created and maintained, in order to extract the best performance.

With regard to exterior boundary connectivity, the half edge data structure is difficult to work with when individual triangles or triangle groups touch at a single vertex. The lack of a common edge leads to topological ambiguity with respect to the orientation of the open boundary edges. As a way of visualizing this, consider two triangles that share one vertex, that happen to be coplanar. As you traverse the outside boundary of one triangle, reaching the common vertex, which edge of the other triangle do you move to? You may decide that it is one particular edge, based on a visualization or drawing of the two triangles. You would possibly choose the edge that visually suggests a counterclockwise traversal. This is not necessarily right and not necessarily wrong. Either triangle could be twisted about the common vertex without

changing the topology, and this is where the ambiguity arises. There simply is not one correct choice for the half edge connectivity when one triangle or set of triangles touches another at just a single vertex without a common edge. It is possible to resolve this using geometric (not topological) reasoning, in some cases.

The problem described above can arise when constructing a half edge model from a simple indexed mesh (or polygon soup), even when the model ultimately has no scenario like the one described. To avoid the ambiguity while constructing a half edge model from a simple mesh, it is best to only construct interior edges and faces until you have added all vertices and triangles to the half edge model. In some cases, when you add a new face, your new interior edges will fill in the HE.opposite field of some existing half edge. This is the case whenever you add a new face with an edge that is adjacent to the interior edge of an existing face. Those edges/faces are naturally resolving themselves as being part of the interior of the mesh. In the end, some edges will have HE.opposite == NULL. You can fill those in as the last step, and use vertex 1 ring traversals to fill in the HE.next ordering around the boundary (except, of course, in the case of a single shared vertex.)

What can we do with this?

- Build a navigation mesh
  - Model an underlying terrain or navigation surface
  - "Imprint" obstructions by splitting edges
  - Split faces at obstruction boundaries
  - Remove faces under obstructions by setting face pointer of loop edges to NULL

What can we do with this?

- Traverse navigation mesh
  - Either for player or NPC
  - Knowing which navigation face object is currently ***on***, use Half Edge operations to determine movement

What can we do with this?

- Implement a 3D modeling application
  - Use Half Edge (or other data structure) as underlying data representation
  - Implement face, edge, vertex selection
  - Provide options to locally modify mesh based on selections

What can we do with this?

- Trim a mesh against a cutting plane
  - Step 1: Split edges that cross the plane
  - Step 2: Split faces with 2 or more split edges
  - Step 3: Delete faces on trimmed side of plane
  - Step 4: Find open boundary
  - Step 5: Triangulate the boundary

Intersection of edge and plane

- Plane equation

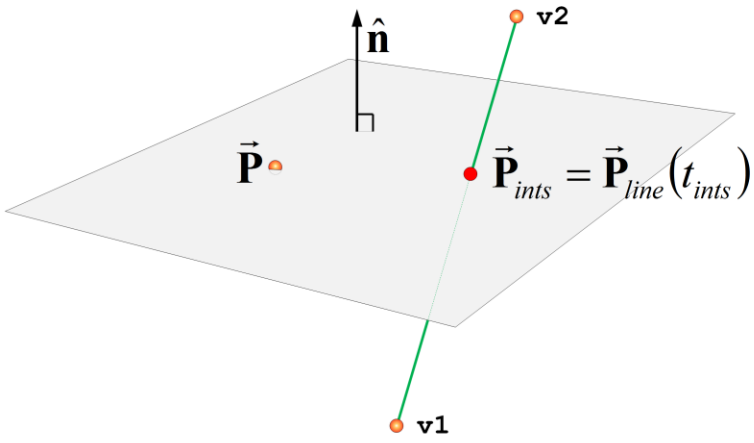
$$\hat{\mathbf{n}} \cdot \vec{\mathbf{P}} = d$$

- Line equation

$$\vec{\mathbf{P}}_{line} = \vec{v}_1 + t\left(\vec{v}_2 - \vec{v}_1\right)$$

- Intersection point

$$t_{\text{ints}} = \frac{(d - \hat{\mathbf{n}} \cdot \vec{v}_1)}{\vec{\mathbf{n}} \cdot \left(\vec{v}_2 - \vec{v}_1\right)}$$

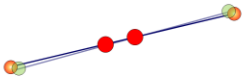


Rendering the geometry

- Convert into GPU friendly format
  - Maintain vector of all vertices during editing
  - Maintain vector of all faces during editing
  - To convert for rendering on modern GPU's:
    - Triangulate all faces
    - Create VBO based on vector of vertices and per-vertex data such as normal, tangent, UV…
    - Create element array buffer based on faces
    - Use graphics API to render

Demo

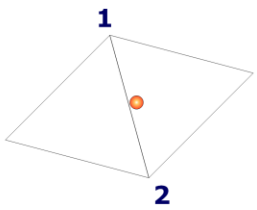
What can go wrong?



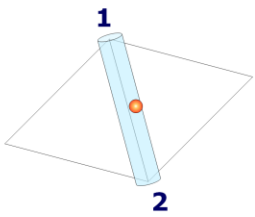
- Tolerance issues
  - Edges not quite collinear
    - Location of intersection point is highly sensitive
  - Points nearly collocated
    - Possible creation of very short/degenerate edges
  - On which side of an edge is a point?
  - Which face near an edge does a ray intersect?

Tolerant Geometry

- Treat edges and points as thick primitives
  - Assign a radius to be used in intersection and proximity testing



Is point on edge?
On left face?
On right face?



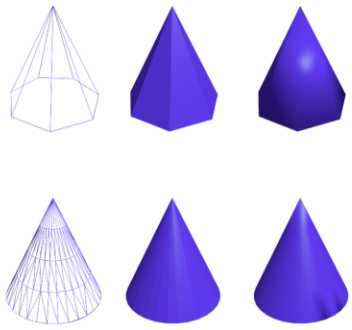
Ambiguous answer depends on:
- Edge from 1->2 or 2->1
- Location

Point is **ON** the edge

Part 2: Higher order surface modeling

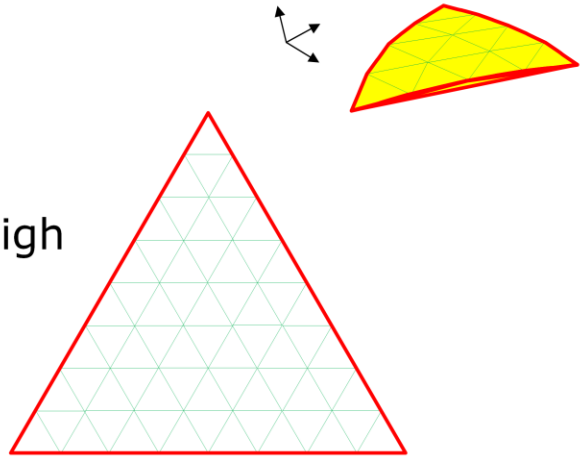
Higher fidelity geometry

- Modern games display models with extraordinary geometric detail
- How can we represent such detail?
- More triangles?
  - Yes, but how? And where?



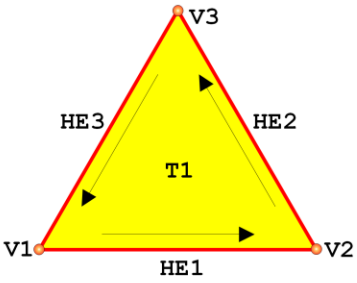
Subdivision surfaces

- Recursively subdivide primitives
- Adjust vertex positions
- Outcome is a smooth, high fidelity surface

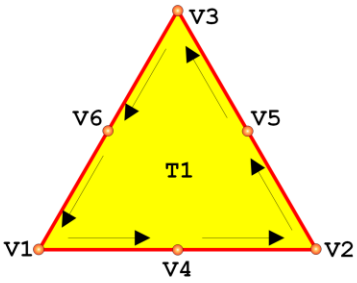


Subdivision surfaces – CPU Approach

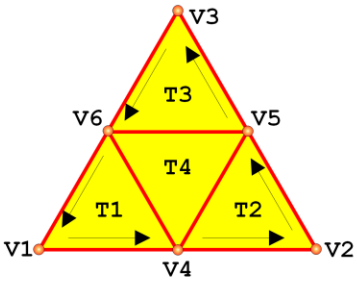
- Planar subdivision using Half Edge data structure



Original



Split All Edges



Triangulate

Subdivision surfaces – CPU Approach

- Smoothing step
  - Adjust vertex positions to smooth surface
- Process
  - Pre-smoothing vertex positions are V^-
  - Create a new vertex array, V^+
  - Apply $V^+ = \mathbf{S}V^-$ **(see slide notes for important details!)**
    - **S**, a local "subdivision matrix," is a coefficient matrix that computes a weighted average at each vertex, based on pre-smoothed locations of neighborhood
    - Weighting factors are a function of a valence(vert)
    - Half edge data structure useful to traverse neighborhood
- Example: Catmull-Clark subdivision surfaces

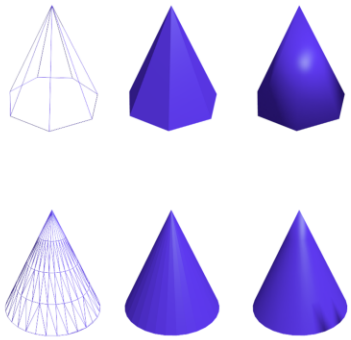
Implementations of subdivision surfaces typically do not apply a global subdivision matrix. A global subdivision matrix, which updates all vertex positions in a single step, is difficult to formulate accurately, in part due to the need to properly handle extraordinary vertices (vertices with a non-standard valence…see references). The global subdivision matrix also makes it more expensive to perform local subdivision refinement, which you might want to apply in a view-dependent level-of-detail application. The reason for the added expense is that a global subdivision matrix would operate on even vertices/edges/faces that are not currently subdivided. Implementations usually smooth the vertices around local neighborhoods, effectively using a local subdivision matrix that is a function of the neighborhood valence. Usually, smoothing is done locally, and in 3 phases: 1) **new** face vertex positions are computed first (if doing face subdivision, which inserts a new vertex into each face…NOT illustrated in this presentation); 2) **new** edge vertex positions, from the edge splits illustrated in this presentation, are computed next using appropriate coefficients and positions of the pre-smoothed corner vertices and new face vertex positions; and, 3) finally, the updated positions of the vertices

that existed before subdivision are computed using the new face and edge vertex positions, based on the valence of the vertices.

If you are interested in implementing subdivision surfaces, please consider reviewing the references and other literature. There is a wealth of information available on theory and implementation schemes.

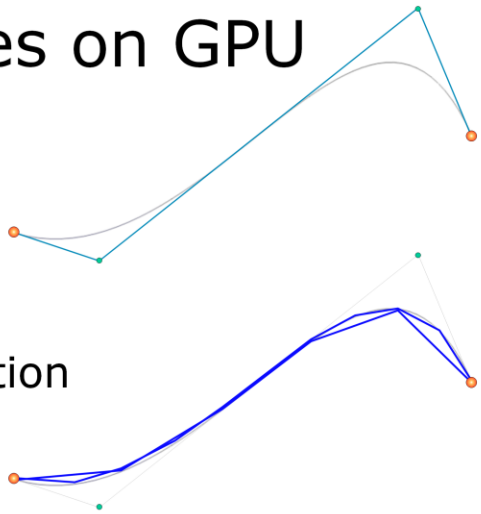
Higher fidelity geometry

- More triangles? Yes, but…
- Large data size can become prohibitive
  - CPU to GPU cost
  - Cost to transform/animate every vertex
  - How can we solve this?



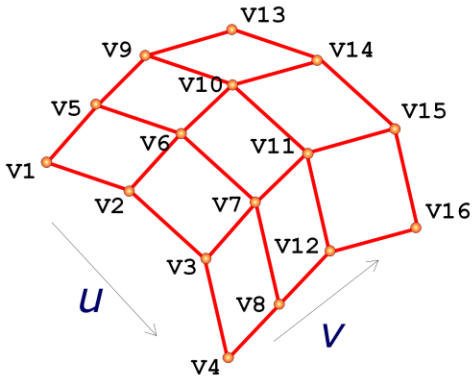
Higher order surfaces on GPU

- Continuously smooth
  - Evaluate at any resolution
- Basic modeling element
  - Control point net
- GPU tessellation and evaluation
  - OpenGL 4/DirectX 11



Rectangular Bicubic Bézier Patch

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{bmatrix} \quad \mathbf{V} = \begin{bmatrix} \mathbf{V}_1 & \mathbf{V}_5 & \mathbf{V}_9 & \mathbf{V}_{13} \\ \mathbf{V}_2 & \mathbf{V}_6 & \mathbf{V}_{10} & \mathbf{V}_{14} \\ \mathbf{V}_3 & \mathbf{V}_7 & \mathbf{V}_{11} & \mathbf{V}_{15} \\ \mathbf{V}_4 & \mathbf{V}_8 & \mathbf{V}_{12} & \mathbf{V}_{16} \end{bmatrix}$$

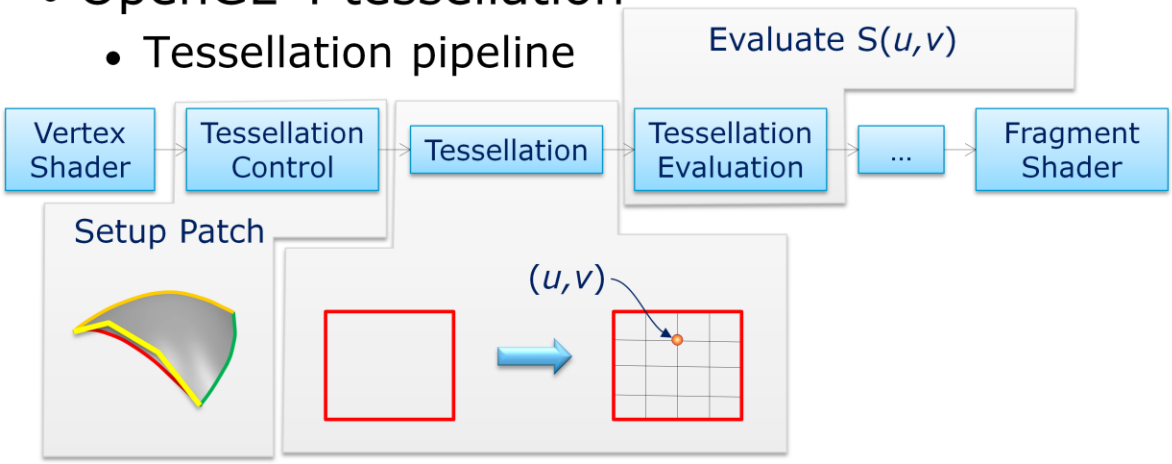


$$\mathbf{P}(u,v) = \begin{bmatrix} 1 & u & u^2 & u^3 \end{bmatrix} \mathbf{M}\mathbf{V}\mathbf{M}^{\mathrm{T}} \begin{bmatrix} 1 \\ v \\ v^2 \\ v^3 \end{bmatrix}$$

Note that you can compute a u direction tangent at a parametric patch point (u,v) by taking the u derivative of the **P**(*u,v*) equation on this slide. And you can compute the v direction tangent (or bitangent) at the same point by taking the v deriviate of **P**(*u,v*). The local surface normal at **P**(*u,v*) is the cross product of the tangent and bitangent. Usually you need to normalize these before use.

Rendering on the GPU

- OpenGL 4 tessellation
  - Tessellation pipeline



Use Half Edge to generate patches

- Given a closed mesh of tris or quads
  - Create a patch per primitive
  - Use half edge to extract full 16x16 (for quads) control net using neighbor information

Demo with code

Final Remarks - Topology

- Why use topological data structures such as HE?
  - Rapid access to complete local neighborhood (1-ring, 2-ring…) of face/vertex/edge
  - Elegant, well-defined interface for editing/modifying mesh
  - Use of markers supports sophisticated selection modes
  - Good support for editing subdivision surfaces

Final Remarks - Topology

- Considerations
  - Topological data structures typically are not GPU friendly
  - Alternatively, could use spatial partition scheme to rapidly modify GPU-friendly vertex/index arrays
    - E.g., use spatial hash or Kd-tree to locate faces/vertices/edges for an edit operation
    - May need to pack or "defragment" arrays periodically to optimize memory usage, since editing operations may necessarily leave portions of linear arrays unused

Final Remarks – Higher Order Surfaces

- Use tessellation sparingly, as it is expensive
- Displacement maps over a GPU-tessellated triangle mesh
  - Alternative to modeling a large number of higher order patches
  - Especially good for games
  - Many game engine art pipelines support this approach

References and Resources

- These slides
  - See http://www.gdcvault.com after GDC
- References for meshes, half-edge
  - http://www.cs.cornell.edu/courses/cs4620/2010fa/lectures/05meshes.pdf (Shirley & Marschner)
  - http://fgiesen.wordpress.com/2012/02/21/half-edge-based-mesh-representations-theory/
    - Nice discussion of invariants
    - See also the followup: http://fgiesen.wordpress.com/2012/03/24/half-edge-based-mesh-representations-practice/
  - http://people.csail.mit.edu/indyk/6.838-old/handouts/lec4.pdf
    - Polygon triangulation

References and Resources

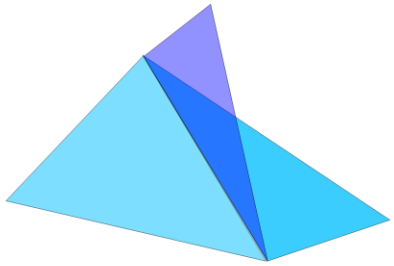
- References for subdivision surfaces
  - Warren, Joe, and Henrik Weimer, Subdivision Methods for Geometric Design, Morgan Kauffman Publishers, 2002
  - Zorin, Denis, et al., "Subdivision for Modeling and Animation," SIGGRAPH 2000 Course Notes, http://www.mrl.nyu.edu/publications/subdiv-course2000/coursenotes00.pdf, 2000
- References for OpenGL tessellation
  - http://prideout.net/blog/?tag=tessellation
- References for robustness issues
  - Christer Ericson, *Real-time Collision Detection*
  - Jonathan Shewchuk's, "Adaptive Precision Floating-Point Arithmetic and Fast Robust Predicates for Computational Geometry"
  - John Hobby, "Practical Segment Intersection with Finite Precision Output" (snap rounding)

Questions?

Backup Slides

Manifold Topology

- Each edge joins exactly two faces
  - Model is said to be watertight
- Open edges that join to one face are allowed
- Modeling operation consistency rules
  - "Invariants"



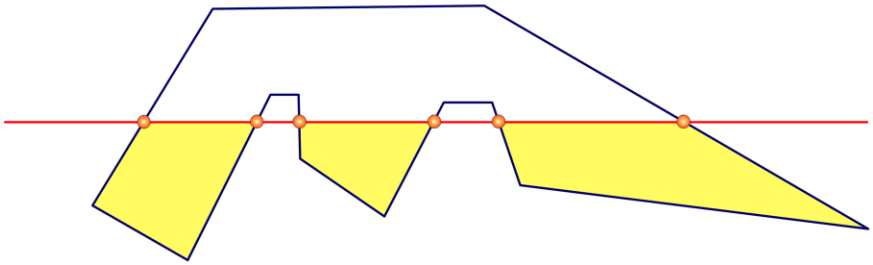
Non-manifold Topology

This is our focus. Simple models with at most two triangles/polygons touching on common edges.

Backup slides: half edge gotchas

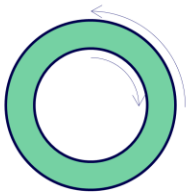
What can go wrong?

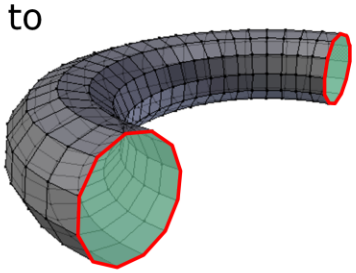
- Be careful when clipping concave face
  - Clipping against a plane can generate multiple loops
  - User marker flags to tag start and stop points
  - Recursively traverse to find ears to clip



What can go wrong?

- Some scenarios produce multiple loops
  - Holes in a face
  - Requires additional triangulation logic
  - Nested loops: auxiliary edge to convert to simple polygon
  - Multiple un-nested loops: locate and triangulate each loop separately



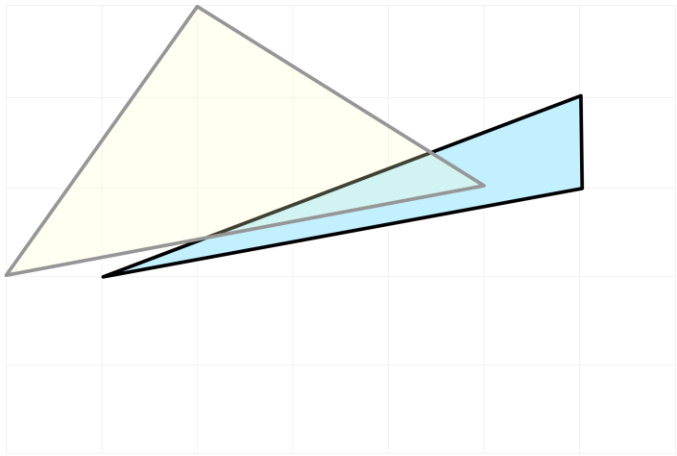


*See speaker notes below slide for an important consideration!

NOTE: It is straightforward to triangulate/cover an open loop that is on a plane. Or one that is approximately planar. If the edges on the loop are not all coplanar, then it is trickier. It may be possible to find some projection plane in which to perform the triangulation connectivity (a plane in which the projection of the edge loop is a simple polygon with all the original edges visible), but a different triangulation will result from different project plane choices. Ultimately, the triangles produced will not be coplanar if the edges were not coplanar, of course.

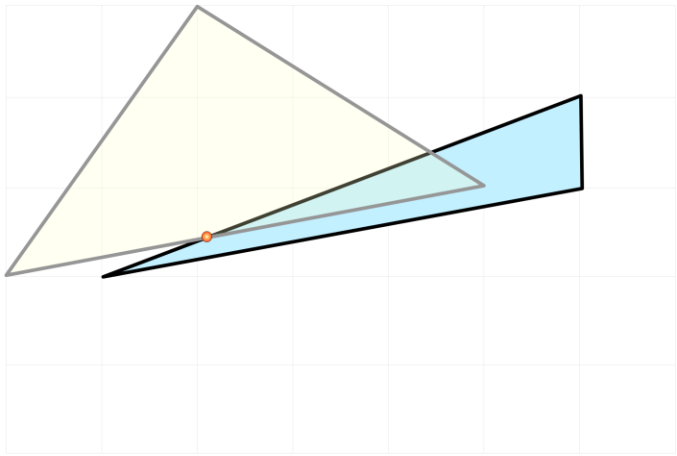
Backup slides: robustness issues

Orientation Inversion



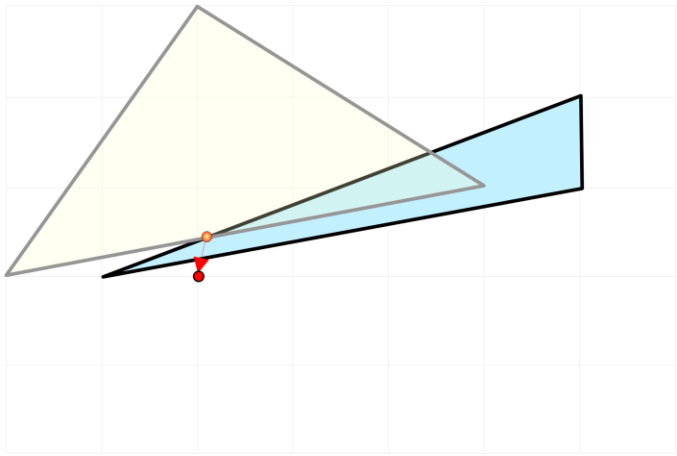
This grid is a portion of the representable floating point numbers. These two triangles are defined by corners that are representable points. Points not lying on the intersection of horizontal and vertical grid lines are not representable. Any unrepresentable number is approximated by a representable number.

Orientation Inversion

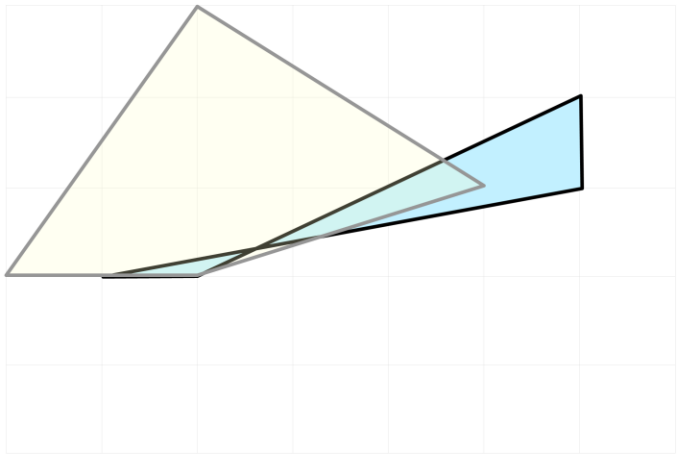


This intersection point is not representable, so the floating point math system will approximate it with the nearest representable number coordinate.

Orientation Inversion

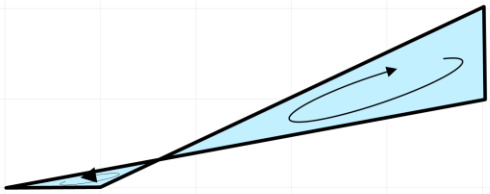


Orientation Inversion

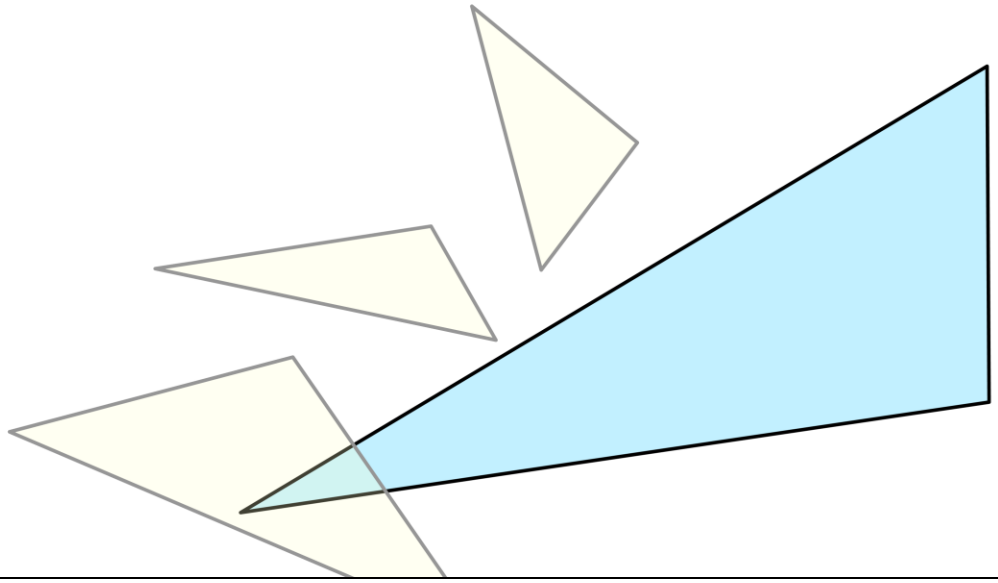


Orientation Inversion

Polygon is no longer simple (it self-intersects) and no longer has a consistent orientation



Cascades of Extraneous Intersections



Here, though the floating point grid is not shown, you will see that a single non-representable intersection point can lead to a chain of intersections that aren't present in the original perfect geometry.

Cascades of Extraneous Intersections