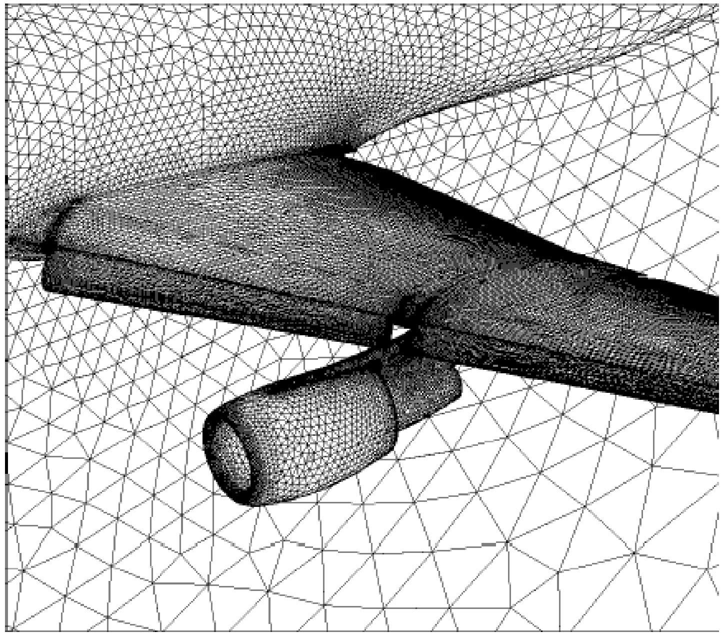
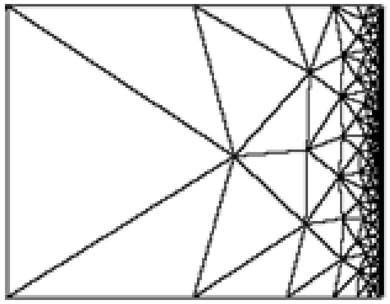
Mesh Generation

Steve Rotenberg
CSE291: Physics Simulation
UCSD
Spring 2019

Mesh Generation

- In order to perform elaborate solid mechanics simulations, we require complex tetrahedral (or hexahedral) meshes
- The production of these meshes falls in the subject of *mesh generation* and is an entire field of its own
- Usually, one wants to start with a surface description of a model (perhaps as triangles or NURBS surfaces) and build a tetrahedral model of the internal volume
- There may be quality restrictions on the tetrahedra. For example, there may be requirements that they be within a certain range of sizes and it is often best if they are as close to equilateral as possible
- Sometimes, we want the mesh to be *graded* meaning that we want more detail in a particular area of interest and we want a smooth gradation of detail between the higher and lower detailed areas





Mesh Generation

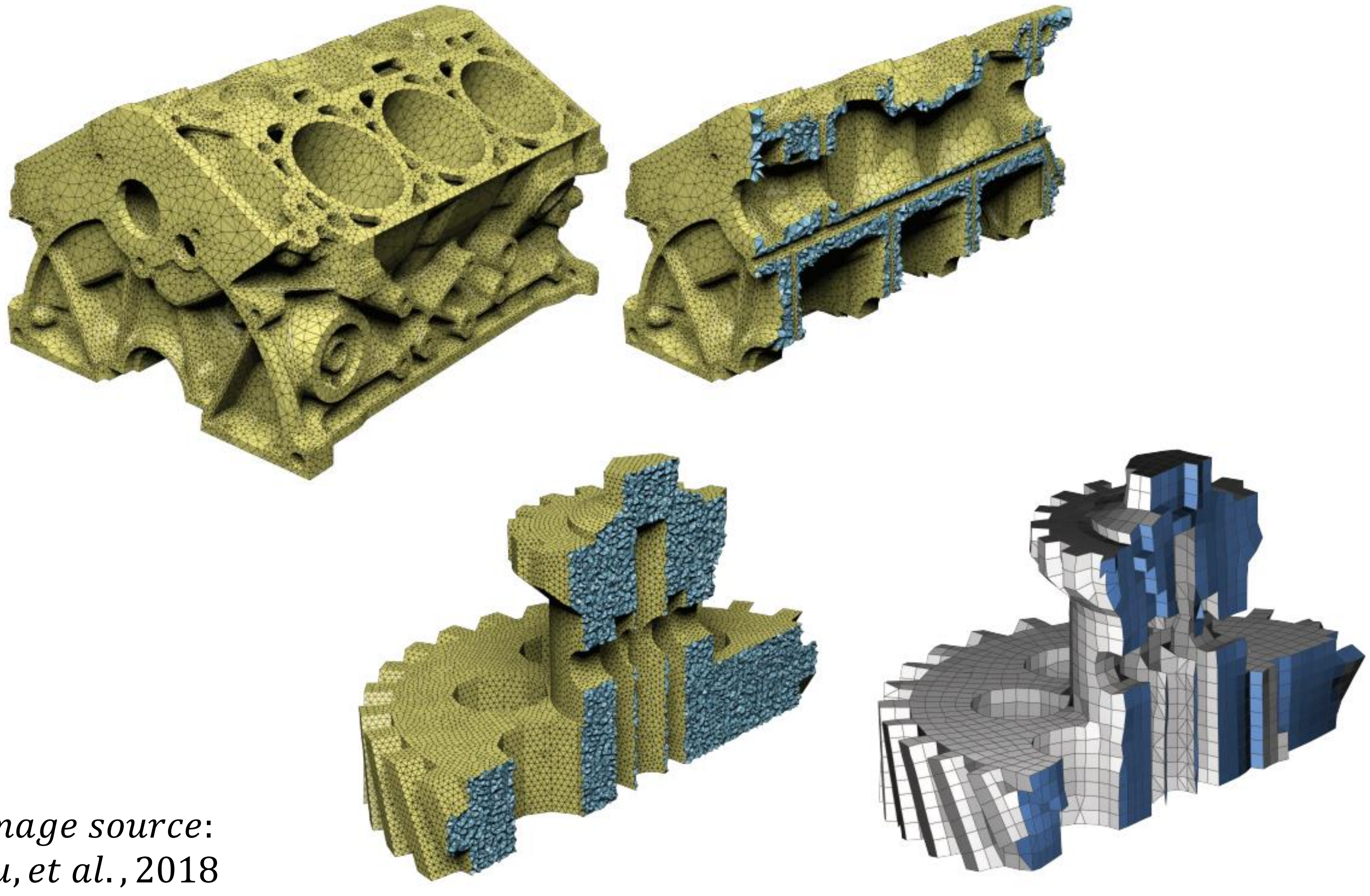
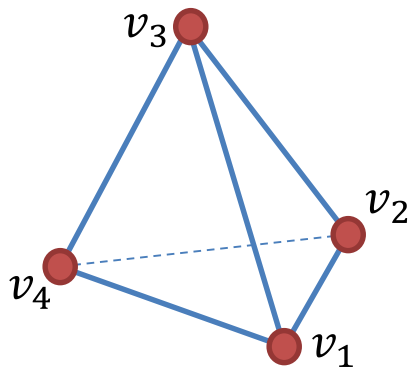


Image source:
Hu, et al., 2018

Mesh Representation

Vertex Numbering

- Let’s assume we have a mesh of tetrahedra that we are loading from a file or generating with a meshing algorithm
- At a minimum, we would represent a model as an array of position vectors and an array of tetrahedra, each defined with 4 vertices, which are just integers that index into the array of positions
- It is customary to define triangle vertices in a counter-clockwise order, so we will define our tetrahedron such that the first 3 vertices define one of the triangle faces in counter-clockwise order, and the fourth vertex would be the remaining one



Mesh Data Structures

- Obviously, there are many different ways one can design the data structures for the mesh
- If one is building a general purpose simulation toolkit, it is likely that there would be a desire to be able to use meshes for different types of physics problems
- In other words, one could use the same mesh generation technology for fluid dynamics, solid mechanics, electromagnetics, etc.
- For general systems, it makes a lot of sense to decouple the mesh system from the physics system
- For simpler systems that are only focused on a single type of problem, it may not be necessary

Mesh Data Structure

- Finite element modeling is computationally demanding, as it is often desirable to model fine scale details and thus requires lots of elements
- Therefore, the data structures should be designed with performance taken into consideration
- If one is using C++, a mesh element (like a tetrahedron) would require some way to reference the nodes. Two common options would be by pointer or by index
- An integer index is probably a better choice for several reasons:
  - Allows separate arrays for node data (positions, velocities, etc. can be stored in separate arrays, indexed by the integer). This tends to be better for caching performance
  - Allows one to use a std::vector to store node data that can dynamically grow if the number of nodes increases. Pointers are bad here because the vector may need to reallocate, invalidating any pointers
  - Easier to load/save
- In this way, a mesh element like a tetrahedron really reduces to just 4 integers. We could create a custom data structure for it that stores these, or we could just store one big array of integers in a mesh data structure

Mesh Class

- Perhaps the simplest way to store a mesh of tetrahedra is:

```
class Mesh {
public:
        void AddNode(const glm::vec3 &node);
        void AddTetrahedron(uint n0,uint n1,uint n2,uint n3);
private:
        std::vector<glm::vec3> Nodes;
        std::vector<uint> Tetrahedra;          // Number of tetrahedra is size() / 4
};
```

- Note: uint is an “unsigned int”, or 32-bit positive integer from 0 to ~4.3 billion

Mesh Class

- This is actually a very good approach. It may not be the most object oriented way to go, but can be very beneficial for performance purposes and one can always build a nice interface on top of this
- Also, if there is a desire to mix element types, one could add more integer arrays:

```
std::vector<uint> Segments;
std::vector<uint> Triangles;
std::vector<uint> Hexahedra;
```

Geometric & Topological Queries

- Let’s say we start with a valid tetrahedral mesh, stored as in the previous slide
- There are some basic query operations we might want to do with the mesh (where we want to obtain some type of information)
- We might want to obtain boundary information such as:
  - All of the triangles that make up the surface of the model
  - All of the segments that make up sharp edges on the boundary
  - All of nodes that are sharp corners
- We might also want to obtain connectivity information such as
  - For each tetrahedron, we might want to know the index of the neighboring tetrahedron across each of the 4 faces
  - For each vertex, we may want to know which tetrahedra connect to it
- We might also want to compute geometric properties such as
  - Tetrahedral volumes
  - Tetrahedral coordinate frames
  - Triangle normals
- There are good algorithms for precomputing and tabling these things, most running in linear O(n) time

Mesh Queries

- It’s nice to separate things like this into their own class to avoid the Mesh class itself from getting too big

```
class MeshQuery {
public:
        // Boundary queries
        void GetBoundaryTriangles(std::vector<uint> &btris, const Mesh &m);
        etc.

        // Connectivity queries
        void GetFaceNeighbors(std::vector<uint> &fns , const Mesh &m);
        etc.

        // Geometric queries
        void ComputeVolumes(std::vector<float> &vols , const Mesh &m);
        etc.
};
```

Physics on Meshes

- To use a mesh for some type of physics simulation, we need to associate additional properties to the mesh, such as
  - Node properties: velocity, mass, force…
  - Tetrahedral properties: material, undeformed shape, plastic tensor…
  - Triangle properties: surface friction…
- In other words, **we want to be able to associate arbitrary properties** (integers, real numbers, vectors, and matrices/tensors) **to geometric components** (nodes, edges, triangles, tetrahedra)
- For caching and computational performance, it is often better to store each of these things in its own array. The size of the array will match the number of the associated geometry type (i.e., force vectors would be stored as an array of 3D vectors where the size of the array is the number of nodes in the mesh)
- Again, this leads to an architecture that is still object oriented, but in a different way than one might initially expect

Elasticity Example

```
class ElasticSystem : public PhysicsSystem {
		Mesh M;

		// Materials
		std::vector<Material> Material;

		// Node properties
		std::vector<float> Mass;
		std::vector<glm::vec3> MaterialCoord;
		std::vector<glm::vec3> Velocity;
		std::vector<glm::vec3> Force;

		// Element properties
		std::vector<uint> MaterialIndex;
		std::vector<glm::mat3> InverseR;
		std::vector<float> Volume;
};
```

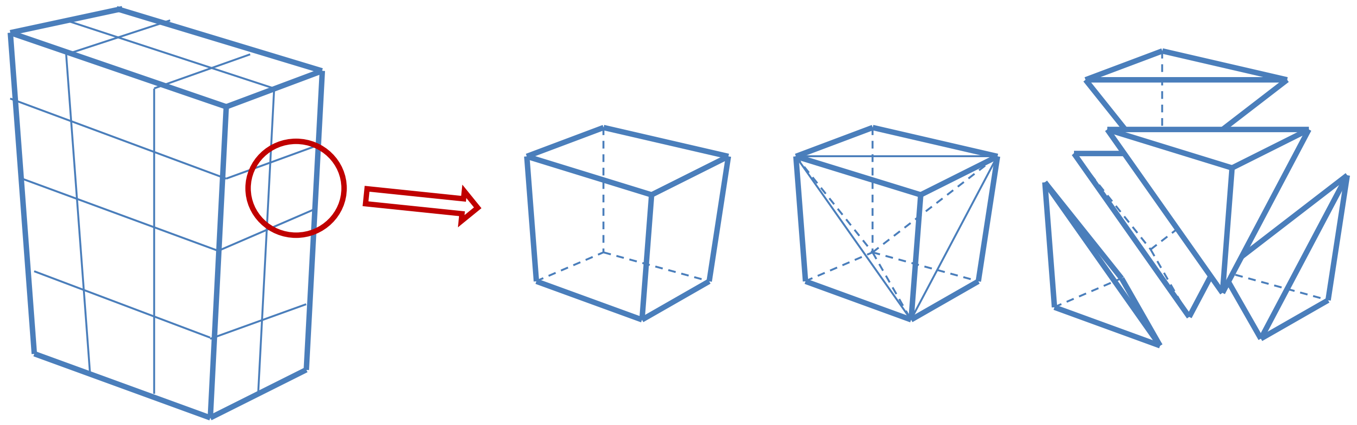
SIMD Computation

- For optimal (CPU) performance, its actually better to separate the *x*, *y*, and *z* components of the vectors & matrices into their own float arrays
- This is known as *structure of arrays* form as opposed to *array of structures*
- This way, one can take maximal advantage of SIMD (single instruction, multiple data) instructions and process several elements or nodes at once
- For example, modern Intel CPUs all have the AVX512 instruction set that operates on 512-bit vectors containing 16 floats (or 8 doubles)
- Using these, one could process 16 elements/nodes at a time on a single core
- Do this in parallel on an 8-core CPU and you can get nearly optimal performance
- Caching behavior is also nice as arrays are largely processed in sequential order and predictable enough to take advantage of pre-caching as well
- One can also use GPU architectures for large-scale physics simulations

Mesh Generation

Cube Lattice

- A simple way to start off is to build a deformable box out of a lattice (3D grid) of box-like cells
- Each cell is built out of 5 tetrahedra: 4 of them from corners of the cell and 1 final one in the middle



Mesh Generation

- The more general approach to mesh generation is to take a triangular boundary mesh as input and construct an internal tetrahedral volume mesh as output
- There are various issues to consider relating to the input and output mesh properties

Input Mesh Quality

- We will assume that the input mesh is defined as a set of triangles
- It might start as higher order curved surfaces (NURBS, subdivision surfaces…), but these would first be tessellated into triangles
- There are several possible issues one might run into with triangle meshes though, and these have to be considered, such as:
  - T-intersections
  - Non-manifold topology
  - Interpenetrations

Output Mesh Quality

- We would almost always require that the output mesh is build from right-side-out tetrahedra with no interpenetrations, and no bad (non-manifold) topology
- Sometimes, it is required that the output mesh retain all of the vertices and triangles of the input mesh. Other times, it can just be close (i.e., within some geometric tolerance)
- There might be additional restrictions on the output mesh related to element size and quality
- For example, it is often desirable that the elements are as close to equilateral as possible to improve the accuracy of the physics
- It is often a problem to have nearly flat or other nearly degenerate shapes

Tetrahedron Quality

- Element quality can affect computational stability and accuracy, and the closer elements are to equilateral in their undeformed state, the better
- There are various measures of tetrahedron quality and various ways to identify ‘bad’ ones
- “What is a Good Linear Element? Interpolation, Conditioning, and Quality Measures”, Shewchuk, 2002, has a detailed discussion on this

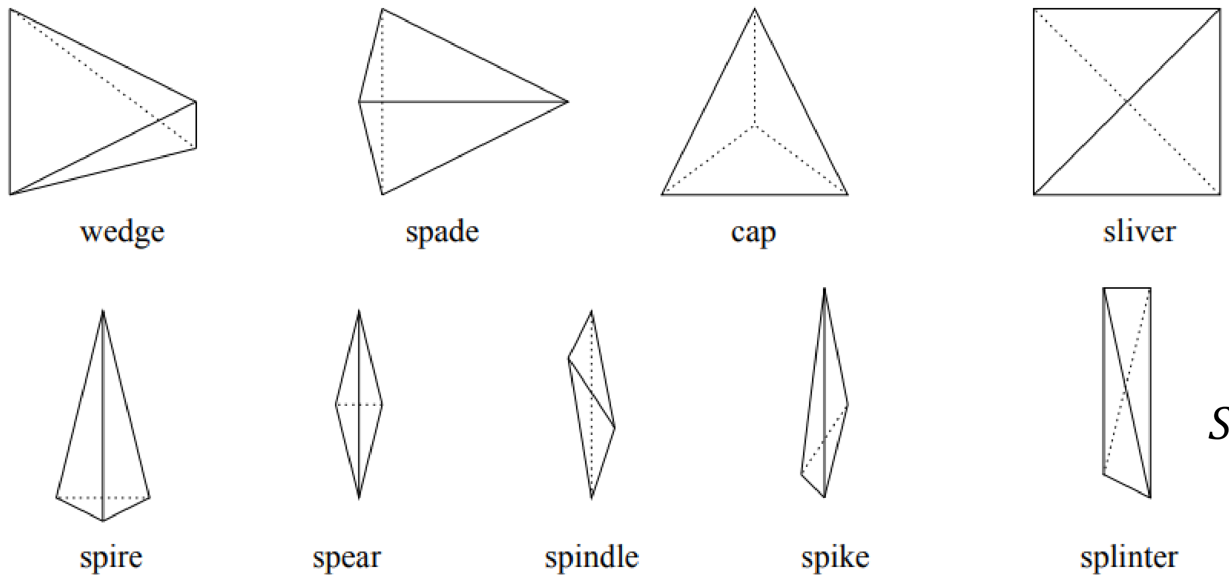


Image source Shewchuk, 2012

Figure 1.19: A taxonomy of skinny tetrahedra, adapted from Cheng, Dey, Edelsbrunner, Facello, and Teng [29].

Tetrahedralization

- Meshing algorithms usually start by building an initial tetrahedralization of the input points
- Some algorithms enforce the boundary triangles into the tetrahedralization at the time of initial construction, and other algorithms do this as a second phase after the initial construction
- Later, we will look at an example of this process in 2D and then discuss its generalization into 3D

Mesh Refinement

- After the initial tetrahedralization is constructed around the surface points, there will usually be many long, thin slivers throughout the model
- The algorithms then proceed by inserting additional points within the volume and adjusting the connectivity of the mesh accordingly
- Often, the points are inserted so as to force re-arranging the worst tetrahedra in the mesh
- Point insertion continues until all tetrahedra are within desired quality tolerances
- These are *incremental mesh refinement* algorithms
- These can also be used dynamically within a simulation to add new computational points where the physics requires additional accuracy

Mesh Smoothing

- *Mesh smoothing* algorithms start with an input tetrahedral mesh and adjust the interior points so as to improve the average quality of the tetrahedra
- A pure mesh smoothing algorithm would only move points and not adjust the element connectivity, but some hybrid smoothing algorithms might combine node movement with connectivity adjustment

Dynamic Meshing

- Several problems can be modeled by constructing an initial mesh and using it throughout the entire simulation
- More complex problems may involve the mesh changing over time in various ways
  - Adaptive meshing: adding/removing points & readjusting connectivity to adapt to accuracy requirements
  - Fracture: this may involve dynamically splitting elements and nodes
  - Moving mesh: allowing the underformed version of the mesh to change, as with the arbitrary Lagrangian-Eulerian method for handling large plastic deformations

Applied Geometry Lab

- A lot of good meshing, geometry processing, and physics simulation papers have come from Matthieu Desbrun’s Applied Geometry lab at Caltech:

- http://www.geometry.caltech.edu/pubs.html

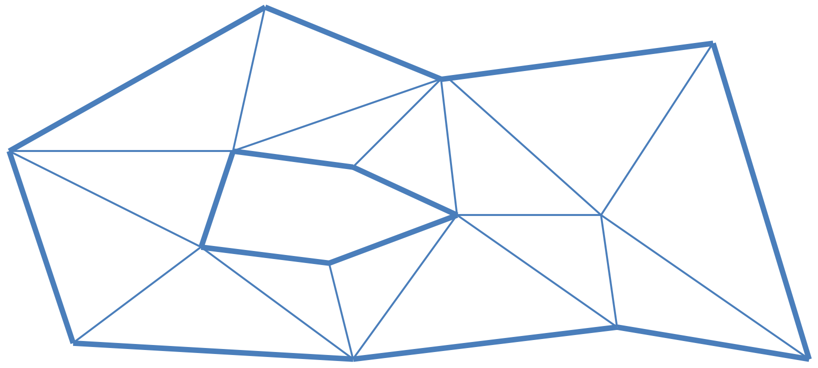
Delaunay Triangulation

Delaunay Triangulations

- We will discuss a classic concept in mesh generation that many algorithms are based on, namely the 2D Delaunay triangulation
- They are named after Boris Delaunay for his 1934 work on the subject
- These have been extensively researched and are a good place to start when studying meshing algorithms
- The concepts of 2D Delaunay triangulations can be extended into 3D as a basis for tetrahedralization algorithms
- Jonathan Richard Shewchuk has done a lot of work on 2D and 3D Delaunay mesh generation and refinement. Some classics include:
  - "Lecture Notes on Delaunay Mesh Generation", Shewchuk, 2012
  - "Delaunay Refinement Algorithms for Triangular Mesh Generation", Shewchuk, 2001
  - "Delaunay Refinement Mesh Generation", Shewchuk, 1997 (Ph.D. thesis)
  - "Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates", Shewchuk, 1997
  - See more on his page: http://www.cs.cmu.edu/~jrs/jrspapers.html

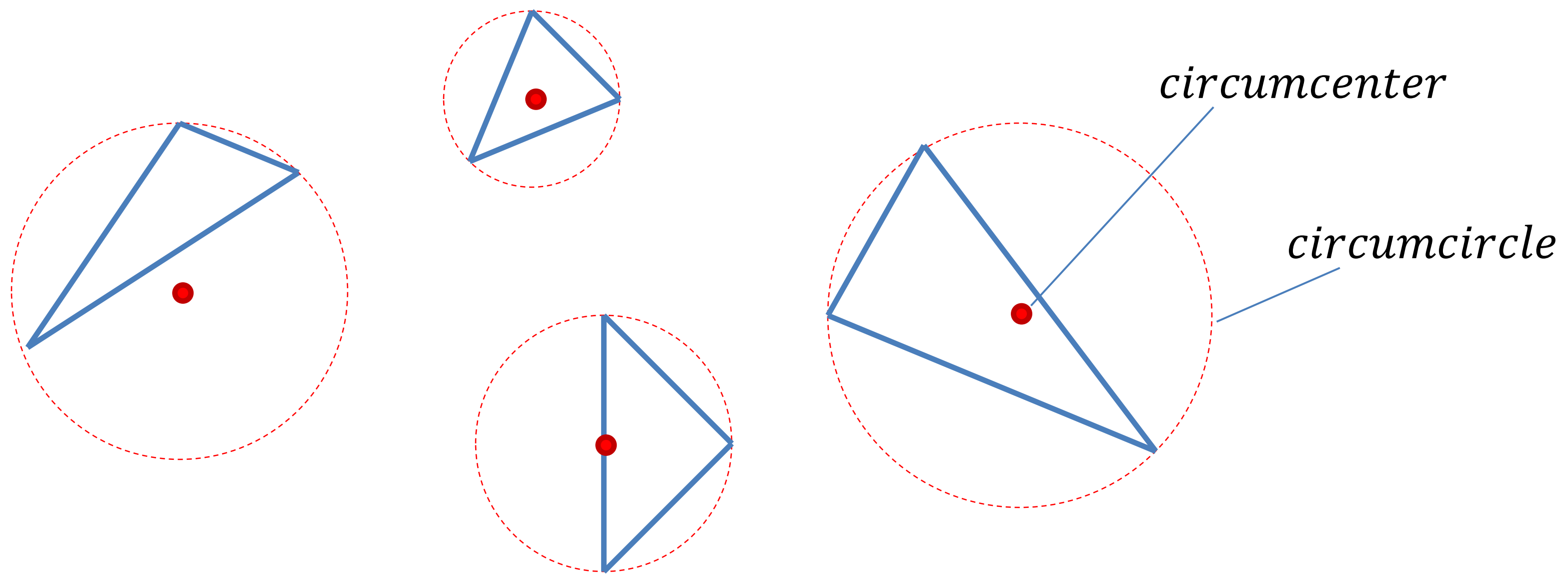
Planar Triangulations

- We define a *planar triangulation* as a set of 2D points and a set of counter-clockwise, non-overlapping triangles
- The outer boundary forms a polygon that does not have to be convex and there may be additional polygonal shaped holes removed from the interior
- Some of the points may be on the interior and others on the boundary



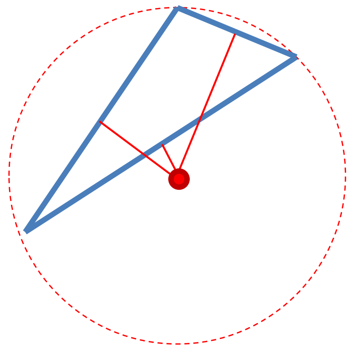
Circumcircle

- The *circumcircle* or *circumscribed circle* of a triangle is the unique circle that passes through all three vertices
- The *circumcenter* of a triangle is the center point of the triangle's circumcircle

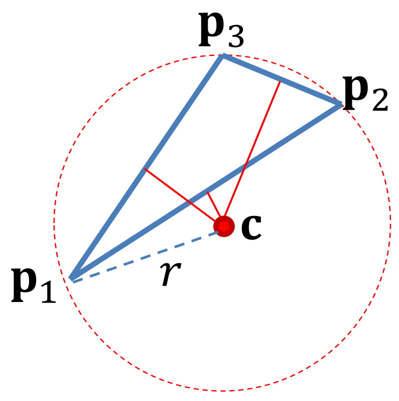


Circumcircle

- If we draw a line perpendicular to each edge and passing through the midpoint, the lines will intersect at the circumcenter
- Therefore, we can intersect any two of these lines to compute the circumcenter
- The radius of the circumcircle will be the distance from this point to any vertex
- We wish to take a triangle defined by points $\mathbf{p}_1$, $\mathbf{p}_2$, and $\mathbf{p}_3$ and find the circumcenter point $\mathbf{c}$ and radius r



Circumcircle



$$\mathbf{s} = \mathbf{p}_2 - \mathbf{p}_1$$
$$\mathbf{t} = \mathbf{p}_3 - \mathbf{p}_1$$

$$d = 2(\mathbf{s}_x \mathbf{t}_y - \mathbf{s}_y \mathbf{t}_x)$$

$$\mathbf{u}_x = \frac{1}{d}\left[\mathbf{t}_y \left(\mathbf{s}_x{}^2 + \mathbf{s}_y{}^2\right) - \mathbf{s}_y \left(\mathbf{t}_x{}^2 + \mathbf{t}_y{}^2\right)\right]$$

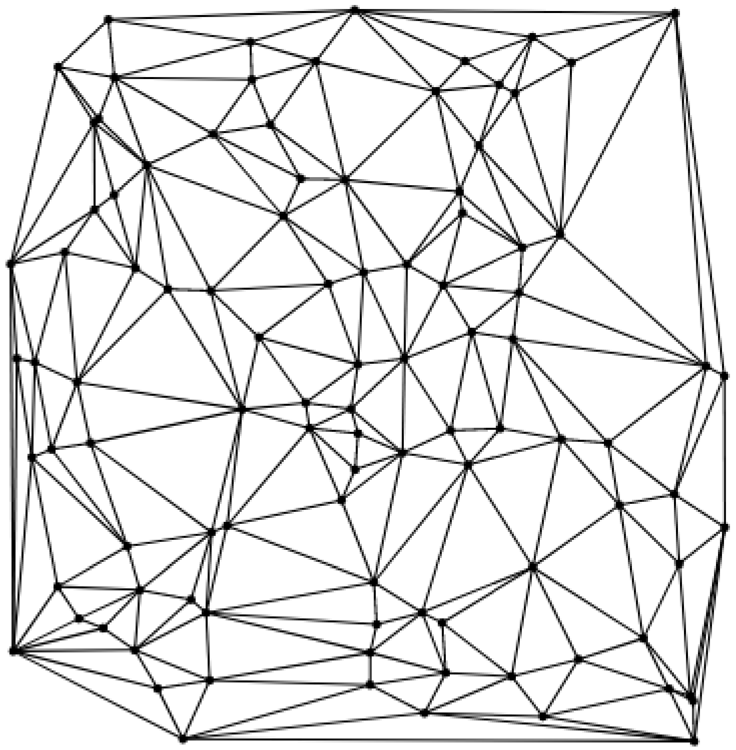
$$\mathbf{u}_y = \frac{1}{d}\left[\mathbf{s}_x \left(\mathbf{t}_x{}^2 + \mathbf{t}_y{}^2\right) - \mathbf{t}_x \left(\mathbf{s}_x{}^2 + \mathbf{s}_y{}^2\right)\right]$$

$$r = \|\mathbf{u}\| = \sqrt{\mathbf{u}_x{}^2 + \mathbf{u}_y{}^2}$$
$$\mathbf{c} = \mathbf{u} + \mathbf{p}_1$$

Delaunay Property

- The three points of a triangle are on the boundary of its circumcircle
- If no other points in the triangulation lie within a triangle’s circumcircle, then that triangle is said to have the *Delaunay property*
- If all triangles in the triangulation have the Delaunay property, then it is a *Delaunay triangulation*

Delaunay Triangulation



$Delaunay\ triangulation\ of\ 100\ random\ points$

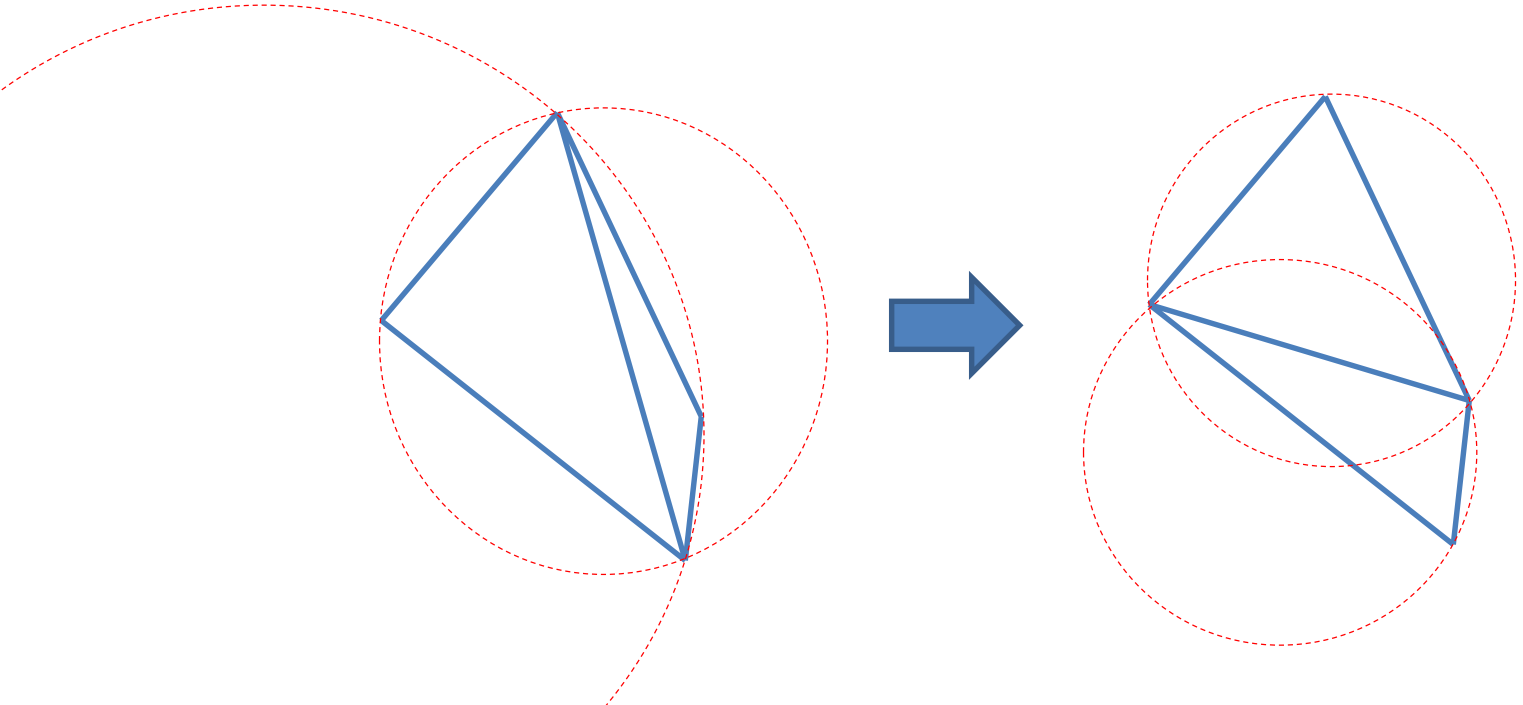
$source: Wikipedia$

Delaunay Triangulations

- Some important & provable properties of Delaunay triangulations include:
  - Every *d*-dimensional point set has a Delaunay triangulation (may have more than one due to symmetries)
  - The Delaunay triangulation **maximizes the minimum angle** of all triangles (Note: it does not necessarily minimize the sum of the edge lengths)
  - The union of all the simplexes form a convex polygon (polyhedron…) around all of the points, called a *convex hull*
  - The number of simplexes in a *d*-dimensional Delaunay triangulation of *n* points is on the order of $O(n^{d/2})$

Edge Flip

- We can convert a non-Delaunay triangulation to a Delaunay triangulation by performing a sequence of edge flips



Edge Flipping

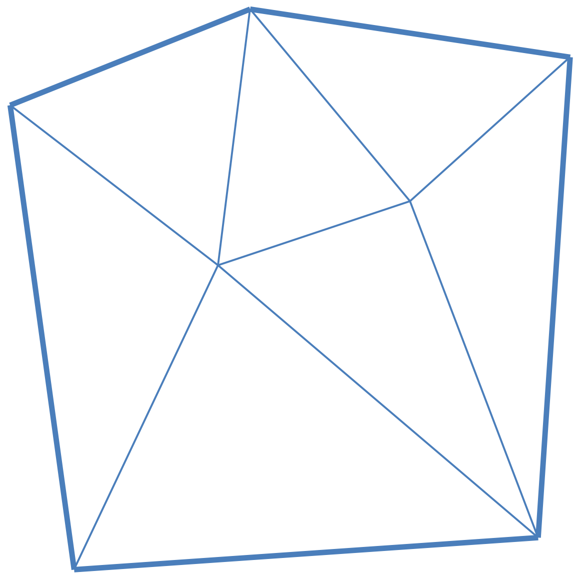
- Several Delaunay triangulation algorithms are built on edge flipping
- A basic method inputs a non-Delaunay triangulation and produces Delaunay triangulation
- It tests each of the edges connecting two triangles and flips it if necessary (i.e., if the connecting triangles are non-Delaunay and the quadrilateral formed by them is convex)
- This proceeds until there are no more edges that require flipping
- At worst case, it can take O(n^2) flips, but usually will require much less

Incremental Lawson's Algorithm

- One can also use an incremental edge flipping method called *Lawson's algorithm* to construct Delaunay triangulations from scratch
- It is an incremental algorithm that inserts one point at a time and maintains the Delaunay property as it goes
- It starts by initializing a large square from two triangles - much larger than the bounding box of the set of points so that every added point is within an existing triangle
- When a point is inserted, we find the triangle containing the point and split it into three new triangles
- We then examine each new triangle and see if it violates the Delaunay property. If it does, we perform the edge flip
- The algorithm propagates outward to neighboring triangles recursively until no more violations are found
- After all points are inserted, the final triangulation is trimmed from the large outer square by removing all triangles connected to the corners
- Note: named for Charles Lawson, 1977

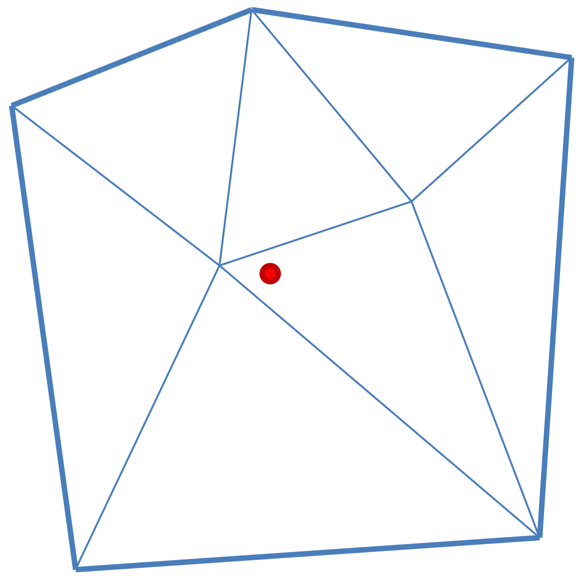
Lawson’s Algorithm

- We start with a valid Delaunay triangulation



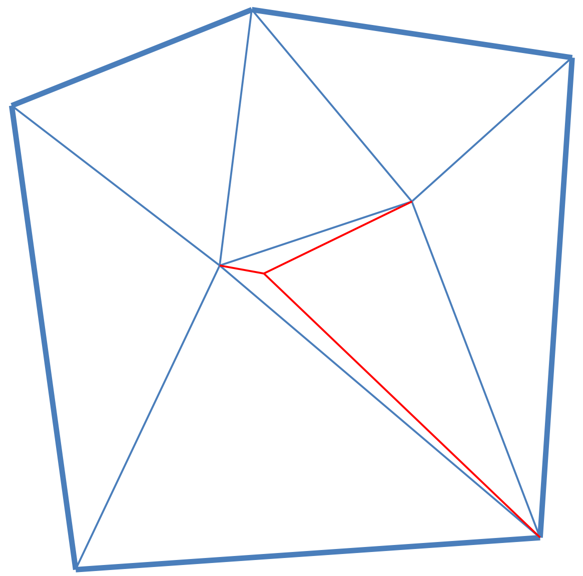
Lawson’s Algorithm

- Insert the new point



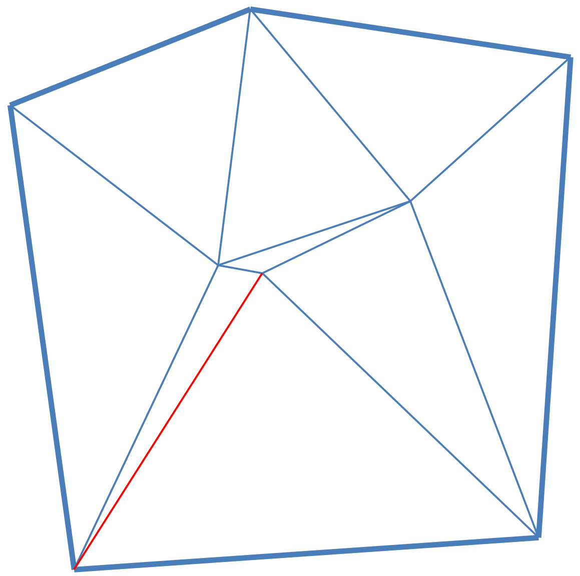
Lawson’s Algorithm

- Split the triangle containing the point into 3 new triangles



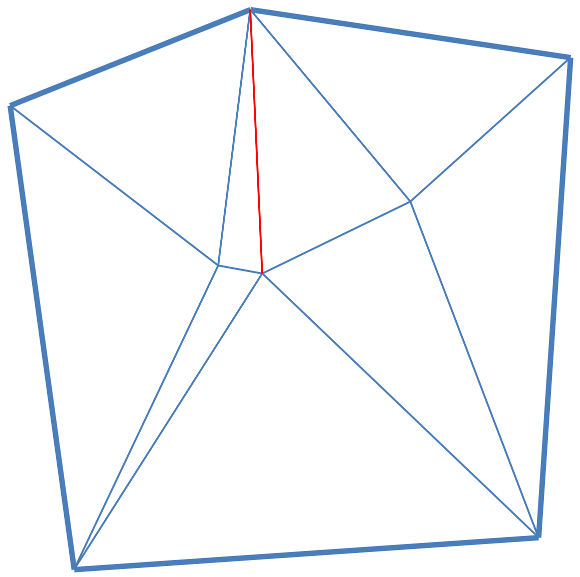
Lawson's Algorithm

- Flip any neighboring edges as needed



Lawson’s Algorithm

- Flip edges until we have a valid Delaunay triangulation



Algorithm Performance

- In a large triangulation, we can expect each new point insertion to cause a relatively small number of edge flips
- Some insertions will not require any and others may require over 10, but on average, there will be just a few
- The performance of the point insertion is therefore roughly constant for a single point, assuming we know which triangle the point falls within
- Finding this triangle is actually the slower part of the algorithm, and different approaches exist to optimize it
- Overall, building a complete Delaunay triangulation with the incremental Lawson's algorithm should perform better than $O(n^2)$
- There are numerous other algorithms for building Delaunay triangulations and many will outperform this one
- Still, this algorithm and related edge flipping approaches are very general purpose and can be used for building initial meshes, refining mesh quality, dynamically adapting meshes during simulation, fixing distorted meshes, and more

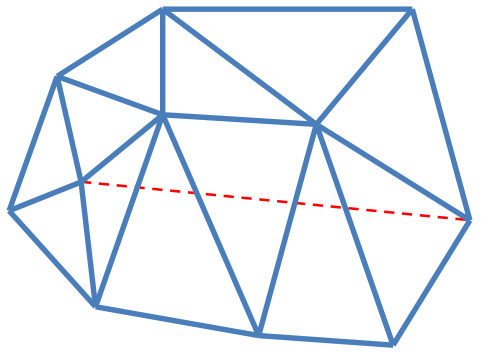
Algorithm Robustness

- It's worth mentioning a few things about robustness of meshing algorithms
- Floating point accuracy can be a big issue in computational geometry algorithms and care must be taken to ensure that floating point roundoff doesn't cause them to fail
- Often, geometric algorithms require making decisions based on some geometric measurement (i.e., is the point above or below some plane). These decisions need to be made in a consistent way and are often crucial to the success of the algorithm.
- It is common to try to design an algorithm to rely on a small number of these geometric *predicates*, and to implement each of them as carefully as possible, sometimes resorting to the use of *exact arithmetic* when necessary
- Computational geometry is an entire subject of its own, and the design of robust algorithms is central to it

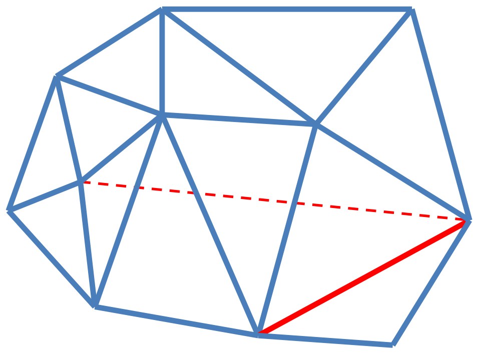
Constrained Delaunay

- A *constrained Delaunay triangulation* is a triangulation that enforces the existence of certain specified line segments input to the algorithm
- It may not be perfectly Delaunay in most cases, but it should be as close as possible
- The input to a 2D constrained triangulation is called a *planar line segment graph* (PLSG) and contains a set of points and line segments connecting the points. One would expect that there would not be any intersecting line segments, but if there were, they could be pre-processed by having new points inserted at the intersection, etc.

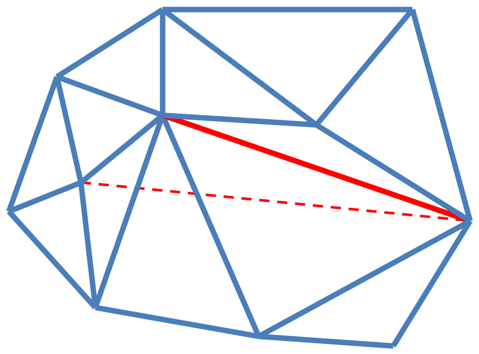
Constraint Enforcement



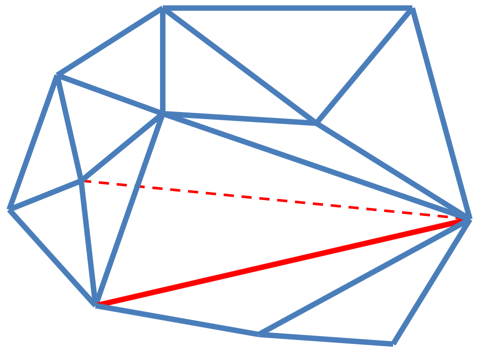
1. *initial state*



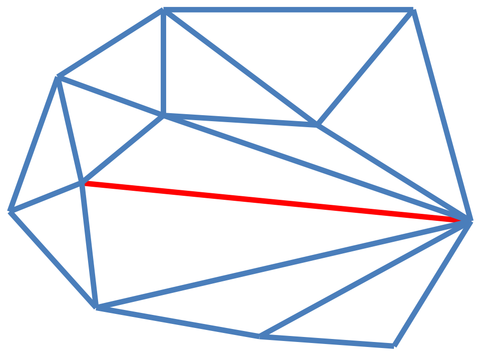
2. *force edge flip*



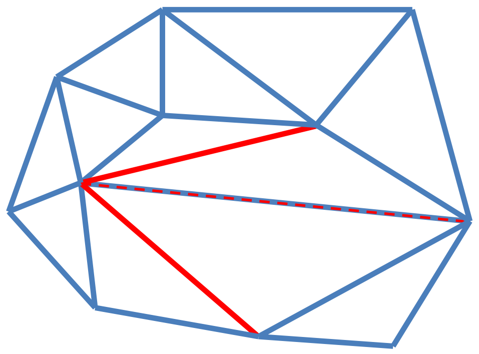
3. *force edge flip*



4. *force edge flip*



5. *final forced flip*



6. *clean up Delaunay where possible*

Void Removal

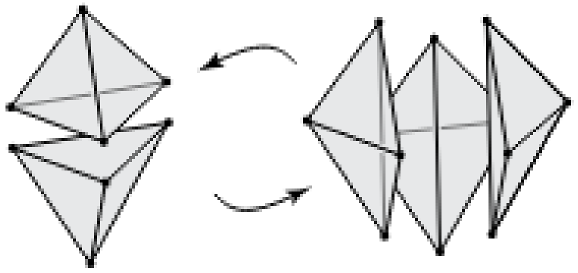
- Often, Delaunay algorithms build a triangulation over all of the input points
- This may include areas that are supposed to be removed, like interior holes
- It is common for algorithms to wait until the end and then remove these by recursively propagating out from boundary triangles
- This may also apply to the exterior triangles if the algorithm started with a large square

Advanced Delaunay Algorithms

- Lawson’s algorithm is a reasonable foundation, but many other approaches exist
- For example, *advancing front* approaches involve presorting the inserted points along one or more axes or by spatial proximity and incrementally growing the mesh
- Delaunay triangularizations and tetrahedralizations are key computational geometry algorithms used in physics and many other fields of computation
- They are a well studied problem and many different algorithms and implementations exist, such as versions optimized for parallel systems, GPUs, etc.

Extension to 3D

- We looked at 2D constrained Delaunay triangulation using Lawson's algorithm with constraint enforcement
- These concepts can be extended into 3D
- The 2D circumcircle can be extended to a 3D *circumsphere*, etc.
- The edge flip concept can also be generalized into *n*-dimensions
- The 3D extension of Lawson's algorithm can get stuck in cyclic loops and isn't guaranteed to terminate, but one can make some minor modifications to fix this
- Unlike the 2D version, the 3D version can run into geometric situations that actually require insertion of additional points to mesh correctly, but this is a manageable problem as well



Mesh Refinement & Smoothing

Delaunay Refinement

- Delaunay algorithms often begin by triangulating the input set of points, which usually only exist on the boundary
- In order to maintain uniform element size and quality, it is necessary to insert additional points inside the volume
- Delaunay refinement algorithms start with a poor quality mesh and insert new points, maintaining the Delaunay property, and improving mesh quality
- A classic algorithm for this is Ruppert’s algorithm

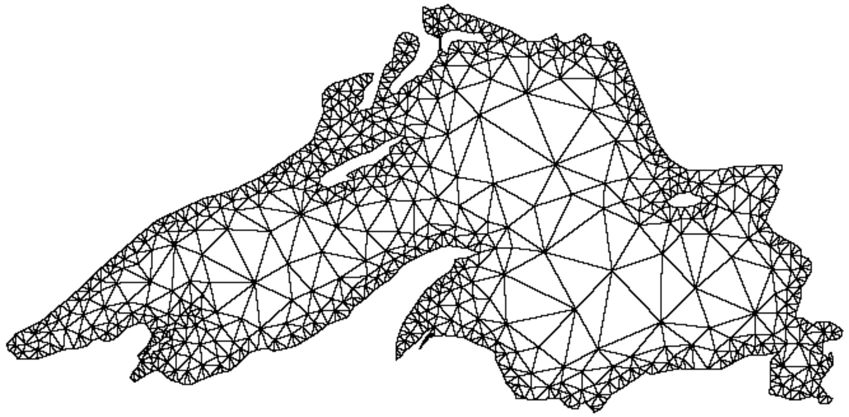
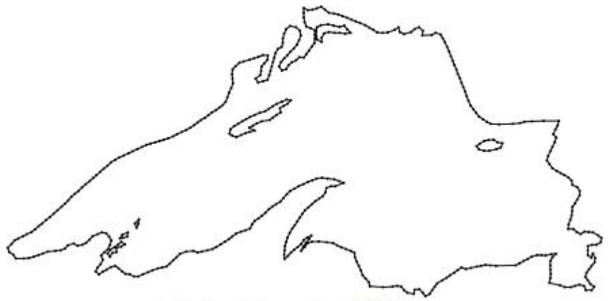


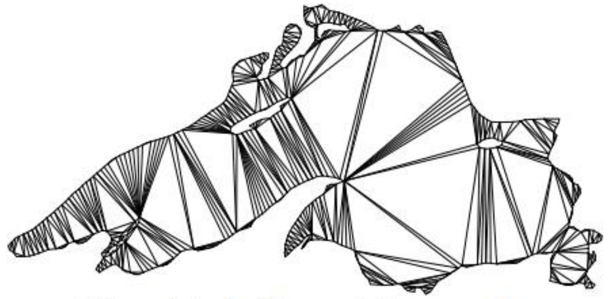
Image source Shewchuk, 2012

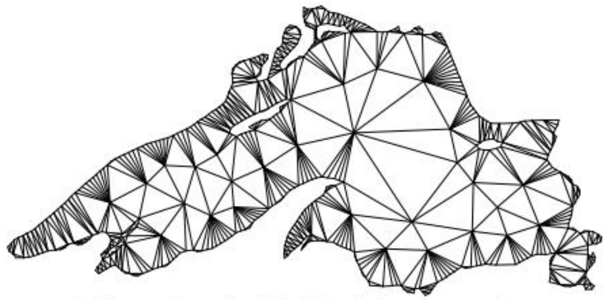
Ruppert’s Algorithm

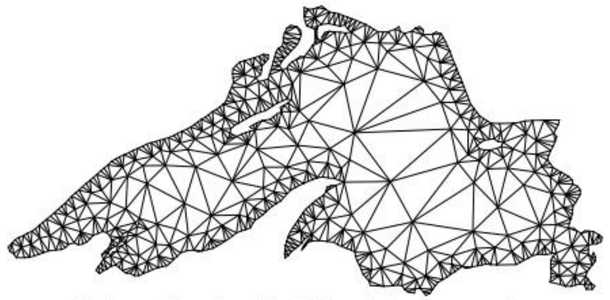
- Ruppert’s algorithm takes a constrained Delaunay mesh as input and adds new points in order to refine the mesh so that no triangle has an angle less than some input tolerance θ_{min}
- The basic idea starts by putting all triangles that violate the tolerance on a sorted queue
- Working from the top of the list, points are inserted at the circumcenter of the worst offending triangle (using Lawson’s or similar incremental point insertion)
- If the new point is within the circumcircle of a constraint edge, the edge is split instead of inserting the point
- As new triangles are formed from edge flips, they are tested for violation of the tolerance and added to the queue if necessary
- The algorithm terminates when the queue is empty (and as long as $\theta_{min} < 34.3°$)
- J.R.Shewchuk made several refinements to the algorithm to handle cases where input data had sharp angles below the tolerance

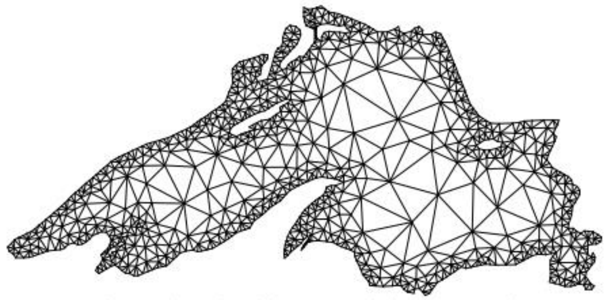
Ruppert's Algorithm


Lake Superior PSLG.


Triangulated with no minimum angle.


Triangulated with 5° minimum angle.


Triangulated with 15° minimum angle.


Triangulated with 25° minimum angle.

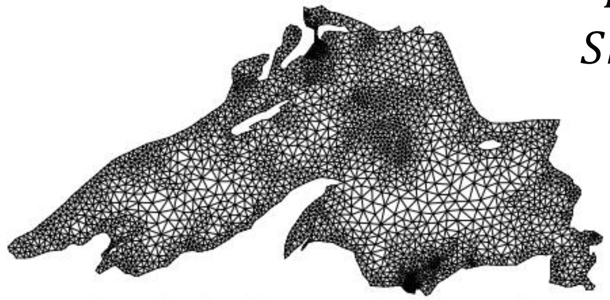
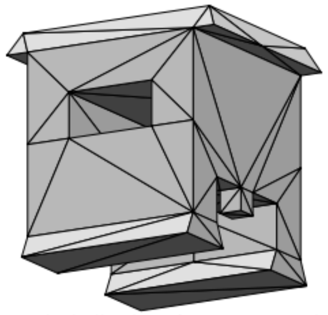

Triangulated with 34.2° minimum angle.

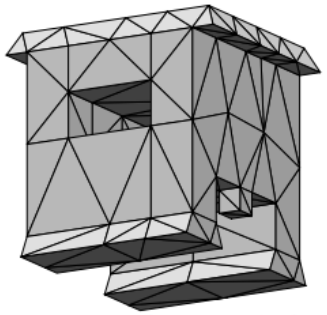
Image source
Shewchuk, 2012

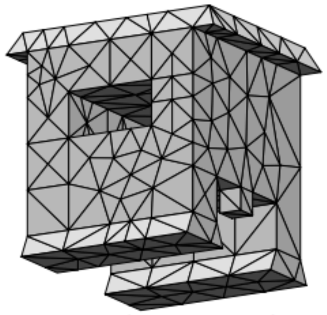
3D Delaunay Refinement

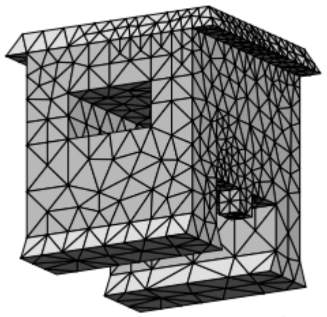
- The mesh refinement process can be extended to 3D as well
- Shewchuk did a lot of work in this area
- See his papers listed on the earlier slide for an overview of 2D Delaunay triangulation, Ruppert’s algorithm, and 3D extensions

3D Delaunay Refinement


Initial tetrahedralization after segment and facet recovery. 71 vertices, 146 tetrahedra.


$B = 2.095$, $\theta_{\min} = 1.96°$, $\theta_{\max} = 176.02°$, $h_{\min} = 1$, 143 vertices, 346 tetrahedra.


$B = 1.2$, $\theta_{\min} = 1.20°$, $\theta_{\max} = 178.01°$, $h_{\min} = 0.743$, 334 vertices, 1009 tetrahedra.


$B = 1.07$, $\theta_{\min} = 1.90°$, $\theta_{\max} = 177.11°$, $h_{\min} = 0.369$, 1397 vertices, 5596 tetrahedra.

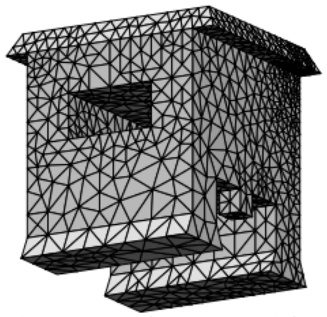

$B = 1.041$, $\theta_{\min} = 0.93°$, $\theta_{\max} = 178.40°$, $h_{\min} = 0.192$, 3144 vertices, 13969 tetrahedra.

Image source Shewchuk, 2012

Mesh Smoothing

- Mesh smoothing algorithms move nodes of a mesh to improve the quality
- Some will also update the connectivity
- Some hybrid approaches combine smoothing and refinement
- There are many smoothing approaches including:
  - “Interleaving Delaunay Refinement and Optimization for Practical Isotropic Tetrahedron Mesh Generation”, Tournois, Wormser, Alliez, Desbrun, 2009
  - “Variational Tetrahedral Meshing”, Alliez, Cohen-Steiner, Yvinec, Desbrun, 2005

Mesh Smoothing

- One simple method from the “Variational Tetrahedral Meshing”, 2005 paper computes a new position for each vertex based on the average of connected tetrahedra circumcenters, weighted by the tetrahedral volumes

$$\mathbf{x}'_i = \frac{1}{|\Omega_i|} \sum_{T_j \in \Omega_i} |T_j| \, \mathbf{c}_j$$

- Where $\mathbf{x}'_i$ is the new position for vertex i, Ω_i is the set of tetrahedra connecting to vertex i, $\mathbf{c}_j$ is the circumcenter of tetrahedron T_j, and $|T_j|$ and $|\Omega_i|$ are the volumes of the tetrahedron T_j or set Ω_i
- They also extend this to handle graded meshes and more

Adaptive Meshing

- Adaptive meshing refers to enhancing the mesh on the fly during the simulation in order to improve detail in areas that require it
- In can involve dynamically adding or removing detail as necessary (similar to adaptive time stepping integrators increasing or decreasing the time step to achieve accuracy goals)
- We can base the decision to subdivide an element on Taylor series estimations of the error induced by the mesh approximation
- Variables assigned to geometry created in this process need to be assigned by interpolation from nearby points

Tetrahedral Meshing in the Wild

“Tetrahedral Meshing in the Wild”

- We will take a look at the recent paper by Hu, Zhou, Gao, Jacobson, Zorin, and Panozzo, 2018
- There have been many papers on tetrahedral meshing over many years
- This is a very state of the art paper that attempts to address many of the limitations that previous algorithms run into when running on actual real-world data
- The paper is posted in the “Papers” section on the class web page

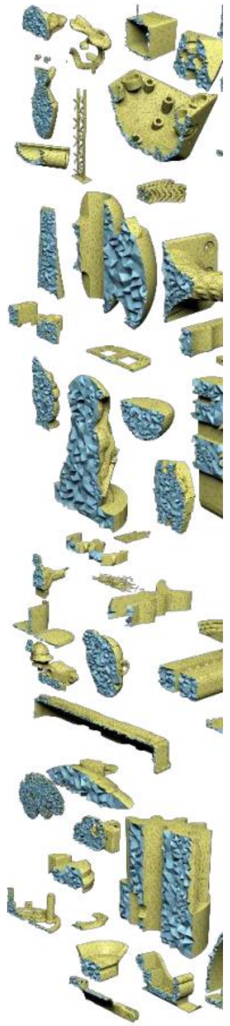


Image source:
Hu, et al., 2018

Tetrahedralization

- Previous approaches to the problem often would run into trouble when the input model was less than perfect
- This would include problems such as:
  - Non-manifold topology
  - Interpenetrations

Key Features

From the paper:

- “We consider the input as fundamentally imprecise, allowing deviations from the input within user-defined envelope of size ϵ;
- We make no assumptions about the input mesh structure, and reformulate the meshing problem accordingly;
- We follow the principle that robustness comes first (i.e., the algorithm should produce a valid and, to the extent possible, useful output for a maximally broad range of inputs), with quality improvement done to the extent robustness constraints allow.
- While allowing deviations from the input, which is critical both for quality and performance, we aim to make our algorithm conservative, using the input surface mesh as a starting point for 3D mesh construction, rather than discarding its connectivity and using surface sampling only.”

Input & Output

- The input is a ‘triangle soup’ which refers to a collection of triangles with no restrictions on connectivity or self-intersection
- The also require two constants: a geometric tolerance ϵ and a target edge length ℓ

- They output an ‘approximately constrained tetrahedralization’ that:
  - Contains an approximation of the input triangles to within tolerance ϵ
  - Has no inverted elements
  - Enforces all edge lengths to be less than ℓ
  - Optimizes mesh quality within these constraints

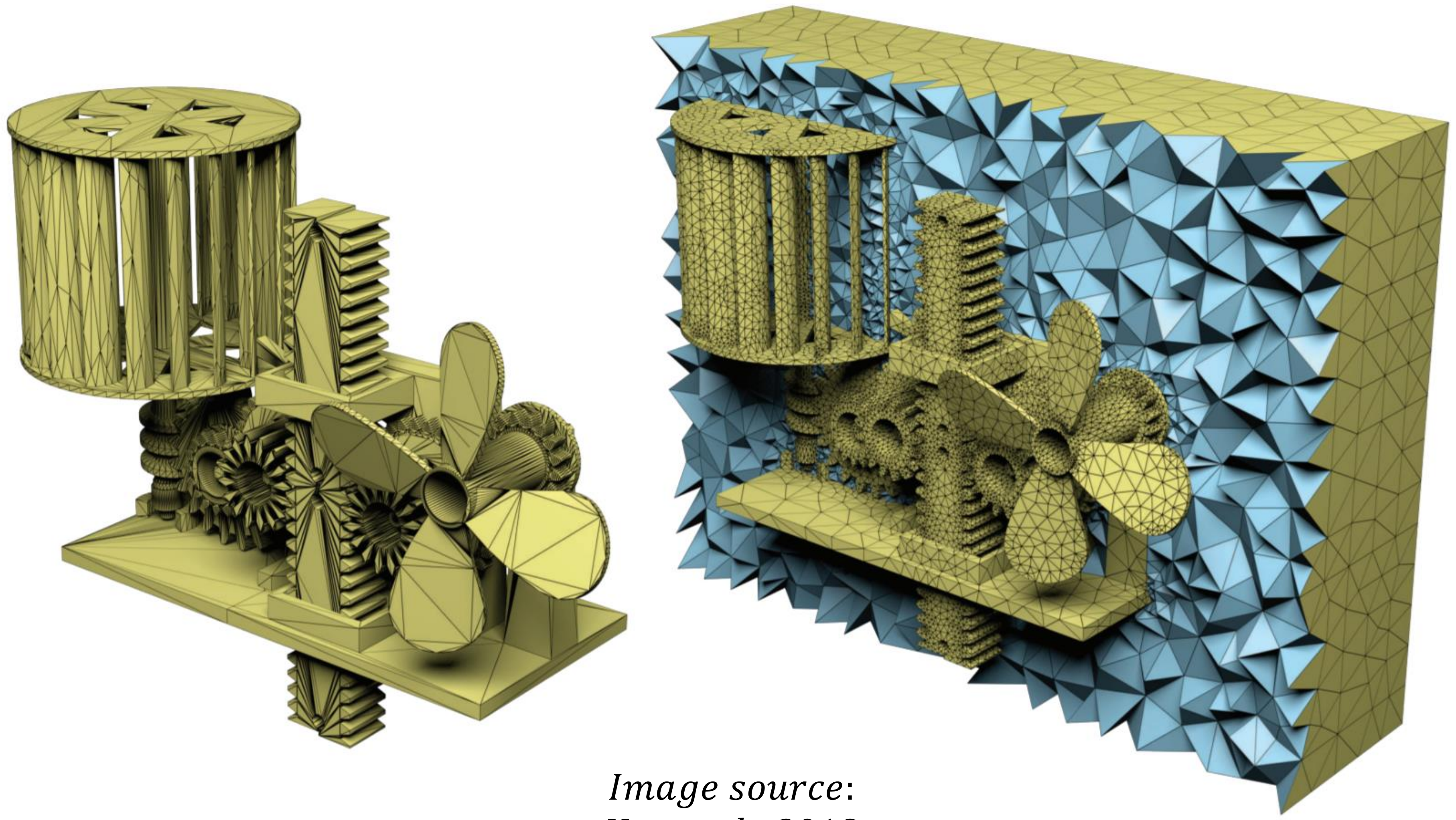
Approach

- The first phase of their algorithm builds a BSP tree (binary space partition) based on the input points, and tetrahedralizes the convex leaf volumes
- It is implemented using exact rational arithmetic, where numbers are represented as a/b where a and b are arbitrary length integers
- They also have a nice method for enforcing that triangles from the input are represented in the tetrahedralization to within the geometric tolerance and is tolerant of cracks and other imperfections

- In the second phase, they improve the quality of the mesh through a process that combines local operations such as edge splitting, edge collapsing, face swapping, and vertex smoothing
- The second phase also converts the data to regular floating point while retaining geometric robustness

- After the mesh is refined, they extract the final tetrahedral with a void removal process that is tolerance of non-manifold input geometry

"Tetrahedral Meshing in the Wild"



$Image\ source$:
$Hu, et\ al., 2018$