Meshing in Fixed Dimension in near Optimal Work and Time Sequential and Parallel

Benoit Hudson
Gary Miller
Todd Phillips

Carnegie Mellon University

Ohio
November 15, 2007

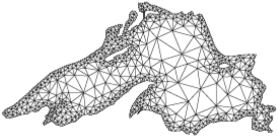
What is Meshing?

- Begin with a geometric domain (Features)

What is Meshing?

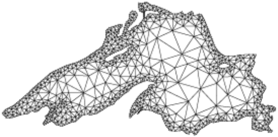
- Begin with a geometric domain (Features)
- Decompose into Simple Pieces
  Quadrilaterals, Triangles, Hexahedra, Tetrahedra

What is Meshing?



- Begin with a geometric domain (Features)
- Decompose into Simple Pieces
  Quadrilaterals, Triangles, Hexahedra, Tetrahedra
- Applications for Physical Simulation or Graphics

What is Meshing?



- Begin with a geometric domain (Features)
- Decompose into Simple Pieces
  Quadrilaterals, Triangles, Hexahedra, Tetrahedra
- Applications for Physical Simulation or Graphics

What good are meshes?

They are used to represent functions

- Temperature, pressure, and velocity

What good are meshes?

They are used to represent functions

- Temperature, pressure, and velocity
- A surface – Mickey Mouse

What good are meshes?

They are used to represent functions

- Temperature, pressure, and velocity
- A surface – Mickey Mouse
- Continuous or better and discontinuous at boundaries

What good are meshes?

They are used to represent functions

- Temperature, pressure, and velocity
- A surface – Mickey Mouse
- Continuous or better and discontinuous at boundaries
- Geometric data structure – Intel chip

Who Uses Meshes?

- Everyone uses meshes!

Who Uses Meshes?

- Everyone uses meshes!
- No One uses meshes!

Who Uses Meshes?

- Everyone uses meshes!
- No One uses meshes!
- Meshes are missing in many physical simulations

Who Uses Meshes?

- Everyone uses meshes!
- No One uses meshes!
- Meshes are missing in many physical simulations
- Many people go to amazing ends not to mesh.

What do People say about the meshing problem?

- Half the people say the problem is solved.

What do People say about the meshing problem?

- Half the people say the problem is solved.
- The other half say the problem is impossible

The Static Meshing Problem

Meshing Algorithm Requirements:

- Guarantees on Element Quality

The Static Meshing Problem

Meshing Algorithm Requirements:

- Guarantees on Element Quality
- Conform to Input Features

The Static Meshing Problem

Meshing Algorithm Requirements:

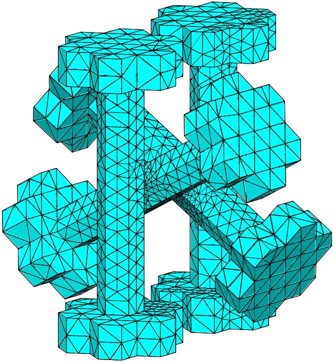
- Guarantees on Element Quality
- Conform to Input Features
- Guarantees on Output Size

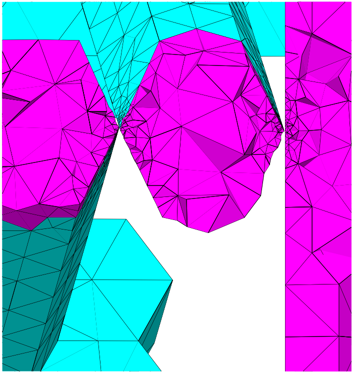
The Static Meshing Problem

Meshing Algorithm Requirements:

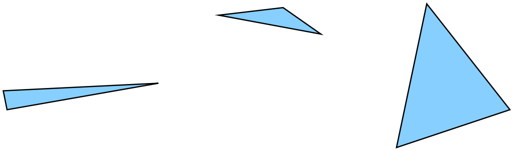
- Guarantees on Element Quality
- Conform to Input Features
- Guarantees on Output Size
- Efficient Runtime and Space Usage

A Simple Example

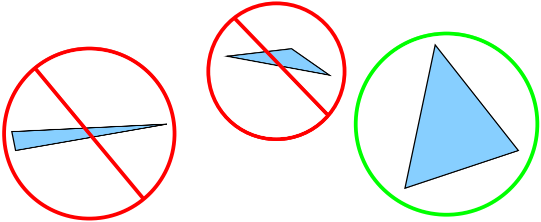




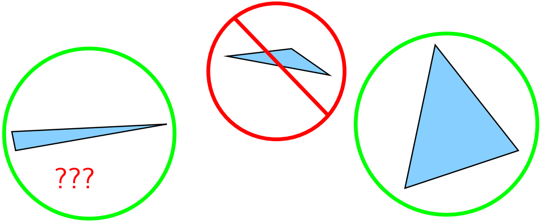
Skinny Elements Bad, Round Elements Better



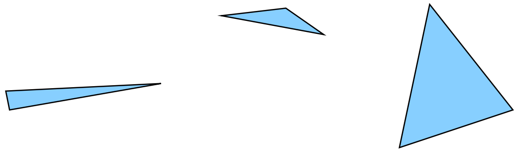
Skinny Elements Bad, Round Elements Better



Skinny Elements Bad, Round Elements Better

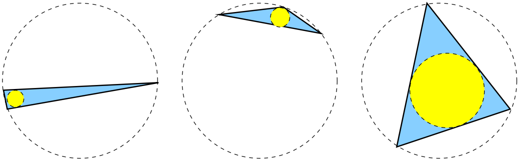


Determining Element Quality



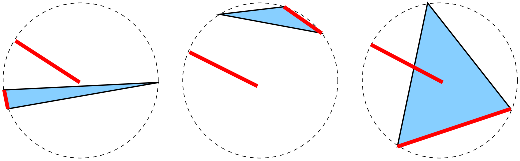
- What does it mean to be round?

Determining Element Quality



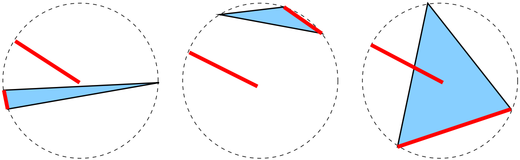
- What does it mean to be round?
- Bounded Aspect Ratio

Determining Element Quality



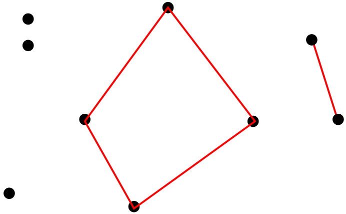
- What does it mean to be round?
- Bounded Aspect Ratio
- Bounded Radius-Edge Ratio

Determining Element Quality



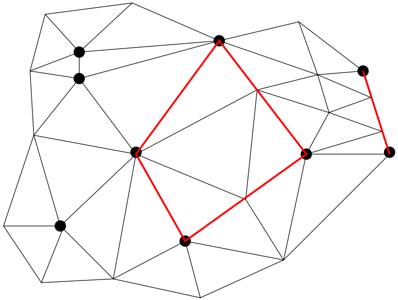
- What does it mean to be round?
- Bounded Aspect Ratio
- Bounded Radius-Edge Ratio
- Input Parameter Determines "Good"

Topologically Conforming Meshes



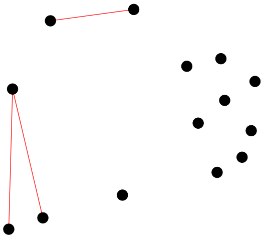
- In 2 Dimensions, Features are Vertices and Edges

Topologically Conforming Meshes



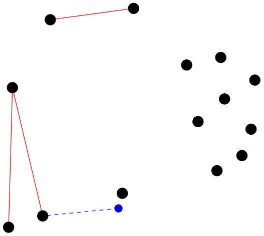
- In 2 Dimensions, Features are Vertices and Edges
- Mesh Must Contain Features (Topologically Conform)

Local Feature Size (lfs)



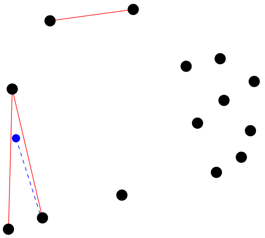
- $\mathrm{lfs}(x)$ = distance to second nearest disjoint feature

Local Feature Size (lfs)



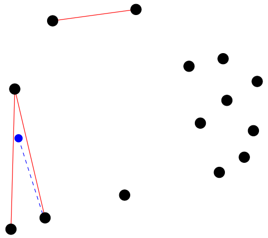
- $\mathrm{lfs}(x) =$ distance to second nearest disjoint feature

Local Feature Size (lfs)



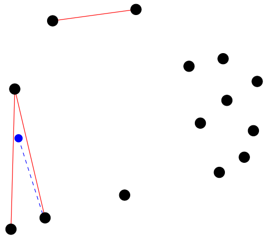
- $\mathrm{lfs}(x)$ = distance to second nearest disjoint feature

Local Feature Size (lfs)



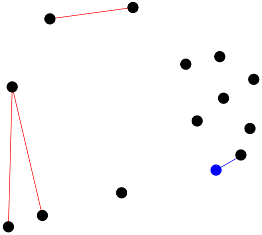
- $\mathrm{lfs}(x) =$ distance to second nearest disjoint feature
- For feature sets of only vertices, we can ignore “disjoint”

Local Feature Size (lfs)



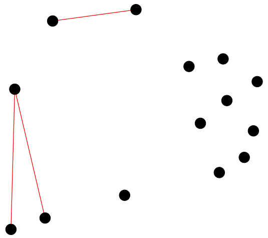
- $\text{lfs}(x) =$ distance to second nearest disjoint feature
- For feature sets of only vertices, we can ignore "disjoint"
- If v is a Vertex?

Local Feature Size (lfs)



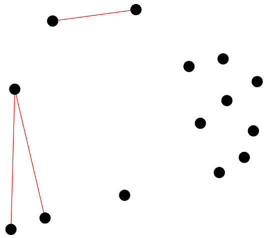
- $\text{lfs}(x) =$ distance to second nearest disjoint feature
- For feature sets of only vertices, we can ignore "disjoint"
- If v is a Vertex? $NN(v)$

Geometrically Conforming Mesh



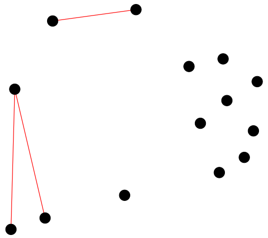
- We could think of just conforming to the lfs

Geometrically Conforming Mesh



- We could think of just conforming to the lfs
- $|E| \in O(\mathrm{lfs}(V_1)), O(\mathrm{lfs}(V_2))$ or perhaps just $|E| \in O(\mathrm{lfs})$

Geometrically Conforming Mesh



- We could think of just conforming to the lfs
- $|E| \in O(\mathrm{lfs}(V_1)), O(\mathrm{lfs}(V_2))$ or perhaps just $|E| \in O(\mathrm{lfs})$
- Critical definition for analysis.

Mesh Size Lower Bound

Theorem: Given a set of input features, *any* geometrically conforming mesh with good with bounded aspect ratio elements, the number of vertices must be:

$$\Omega\left(\int_D \frac{1}{\mathrm{lfs}^d(\mathbf{x})} d\mathbf{x} \right)$$

Note: A bounded radius-edge mesh maybe smaller

$O(1)$-Approximations to Optimal Size

In general, if we guarantee that:

$$|E| \in \Omega(\mathrm{lfs})$$

then the number of vertices is:

$$O\left(\int_D \frac{1}{\mathrm{lfs}^d(\mathbf{x})} d\mathbf{x} \right)$$

So we have a constant factor approximation to an Optimal Size Mesh.

$O(1)$-Approximations to Optimal Size

In general, if we guarantee that:

$$|E| \in \Omega(\mathrm{lfs})$$

then the number of vertices is:

$$O\left(\int_D \frac{1}{\mathrm{lfs}^d(\mathbf{x})} d\mathbf{x} \right)$$

So we have a constant factor approximation to an Optimal Size Mesh.

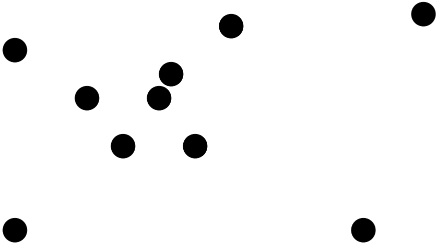
After post processing to remove slivers (Li & Teng)

Notations and Runtime

Size of Input (Number of Features):	n
Size of Output (Points):	m
Constant Dimension:	d
Spread of Input:	L/s

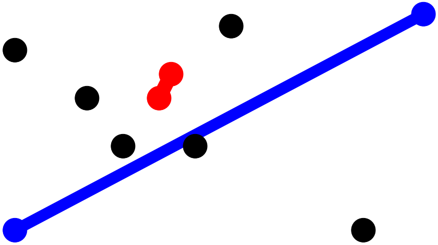
Notations and Runtime

Size of Input (Number of Features):	n
Size of Output (Points):	m
Constant Dimension:	d
Spread of Input:	L/s



Notations and Runtime

Size of Input (Number of Features):	n
Size of Output (Points):	m
Constant Dimension:	d
Spread of Input:	L/s



Notations and Runtime

Size of Input (Number of Features):	n
Size of Output (Points):	m
Constant Dimension:	d
Spread of Input:	L/s

- Optimal Runtime is $O(n \log n + m)$ (Sorting Lower Bound)

Notations and Runtime

Size of Input (Number of Features):	n
Size of Output (Points):	m
Constant Dimension:	d
Spread of Input:	L/s

- Optimal Runtime is $O(n \log n + m)$ (Sorting Lower Bound)
- We can obtain is $O(n \log L/s + m)$
  Optimal if Spread $\in O(n^k)$

Notations and Runtime

Size of Input (Number of Features):	n
Size of Output (Points):	m
Constant Dimension:	d
Spread of Input:	L/s

- Optimal Runtime is $O(n \log n + m)$ (Sorting Lower Bound)
- We can obtain is $O(n \log L/s + m)$
  Optimal if Spread $\in O(n^k)$
- More like $O(d!(n \log L/s + m))$, maybe $O(k^d(n \log L/s + m))$

Remainder of Talk

- Review existing algorithms with size/shape/conformal guarantees

Remainder of Talk

- Review existing algorithms with size/shape/conformal guarantees
- A New Algorithm: **Sparse Voronoi Refinement** (SVR)

Remainder of Talk

- Review existing algorithms with size/shape/conformal guarantees
- A New Algorithm: **Sparse Voronoi Refinement** (SVR)
  - Size and Quality Guaranteed

Remainder of Talk

- Review existing algorithms with size/shape/conformal guarantees
- A New Algorithm: **Sparse Voronoi Refinement** (SVR)
  - Size and Quality Guaranteed
  - Near-Optimal Runtime

Remainder of Talk

- Review existing algorithms with size/shape/conformal guarantees
- A New Algorithm: **Sparse Voronoi Refinement** (SVR)
  - Size and Quality Guaranteed
  - Near-Optimal Runtime
  - Conforms to Features

Remainder of Talk

- Review existing algorithms with size/shape/conformal guarantees
- A New Algorithm: **Sparse Voronoi Refinement** (SVR)
  - Size and Quality Guaranteed
  - Near-Optimal Runtime
  - Conforms to Features
- Bird's-Eye View of Runtime Proof

Runtime Efficient Meshing Algorithms

- Quadtree 2D points $O(n \log n + m)$ *[BEG93]* (edges??)

Runtime Efficient Meshing Algorithms

- Quadtree 2D points $O(n \log n + m)$ *[BEG93]* (edges??)
- Parallel 2D Refinement *[STU02]* $O(\log^2 n)$

Runtime Efficient Meshing Algorithms

- Quadtree 2D points $O(n \log n + m)$ *[BEG93]* (edges??)
- Parallel 2D Refinement *[STU02]* $O(\log^2 n)$
- 3D Structured Octrees *[MV99]* $O(n \cdot m)$

Runtime Efficient Meshing Algorithms

- Quadtree 2D points $O(n \log n + m)$ *[BEG93]* (edges??)
- Parallel 2D Refinement *[STU02]* $O(\log^2 n)$
- 3D Structured Octrees *[MV99]* $O(n \cdot m)$
- 2D *[M04]* $O((n \log n + m) \log n)$

Runtime Efficient Meshing Algorithms

- Quadtree 2D points $O(n \log n + m)$ *[BEG93]* (edges??)
- Parallel 2D Refinement *[STU02]* $O(\log^2 n)$
- 3D Structured Octrees *[MV99]* $O(n \cdot m)$
- 2D *[M04]* $O((n \log n + m) \log n)$
- 2D points *[HU05]* $O(n \log n + m)$–Off-Centers

Runtime Efficient Meshing Algorithms

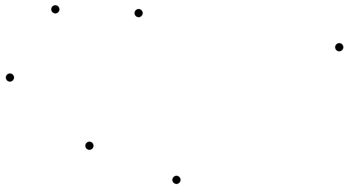
- Quadtree 2D points $O(n \log n + m)$ *[BEG93]* (edges??)
- Parallel 2D Refinement *[STU02]* $O(\log^2 n)$
- 3D Structured Octrees *[MV99]* $O(n \cdot m)$
- 2D *[M04]* $O((n \log n + m) \log n)$
- 2D points *[HU05]* $O(n \log n + m)$–Off-Centers
- This is all assuming $L/s \in poly(n)$

Main Result

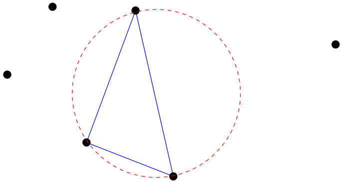
Theorem

Bounded aspect ratio meshing in any fixed dimension in $O(n \log L/s + m)$ work and parallel time $O(\log n \log L/s)$.

The Delaunay Mesh

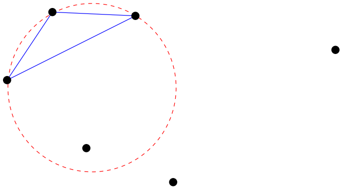


The Delaunay Mesh



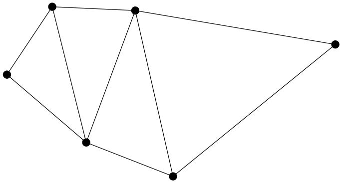
- Empty Circumball Property

The Delaunay Mesh



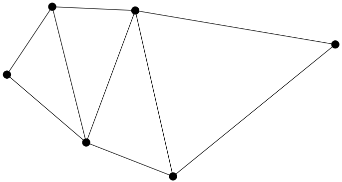
- Empty Circumball Property

The Delaunay Mesh



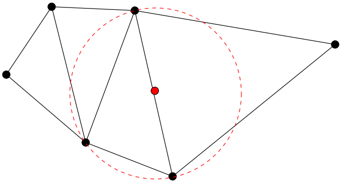
- Empty Circumball Property
- Delaunay Triangles Give a Triangulation (Tetrahedralization)

The Delaunay Mesh



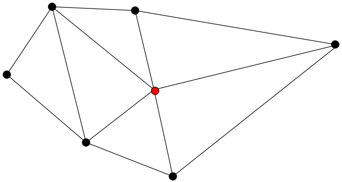
- Empty Circumball Property
- Delaunay Triangles Give a Triangulation (Tetrahedralization)
- By keeping balls empty we can insure conformity.

The Delaunay Mesh



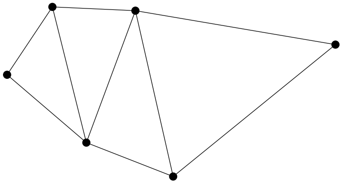
- Empty Circumball Property
- Delaunay Triangles Give a Triangulation (Tetrahedralization)
- By keeping balls empty we can insure conformity.

The Delaunay Mesh

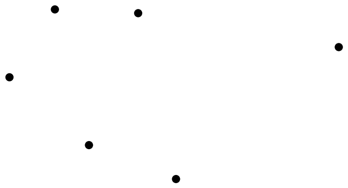


- Empty Circumball Property
- Delaunay Triangles Give a Triangulation (Tetrahedralization)
- By keeping balls empty we can insure conformity.

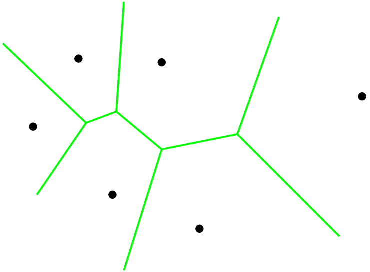
Voronoi Diagrams



Voronoi Diagrams

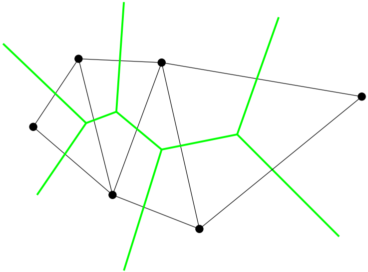


Voronoi Diagrams



- Nearest Neighbor Partition

Voronoi Diagrams

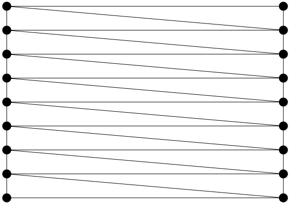


- Nearest Neighbor Partition
- Dual to the Delaunay Triangulation

Delaunay Refinement Algorithm

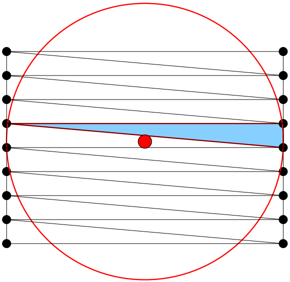
- Obtain the Delaunay Triangulation
- While there are poor elements (`Clean` move)
  - Destroy a Poor Quality Element by Inserting the Circumcenter
  - Update the Delaunay

Delaunay Refinement Algorithm



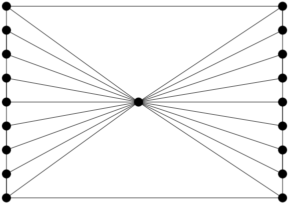
- Obtain the Delaunay Triangulation
- While there are poor elements (`Clean` move)
  - Destroy a Poor Quality Element by Inserting the Circumcenter
  - Update the Delaunay

Delaunay Refinement Algorithm



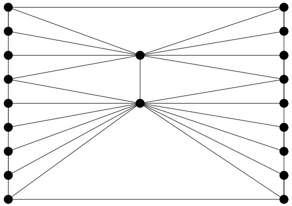
- Obtain the Delaunay Triangulation
- While there are poor elements (`Clean` move)
  - Destroy a Poor Quality Element by Inserting the Circumcenter
  - Update the Delaunay

Delaunay Refinement Algorithm



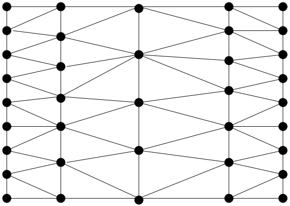
- Obtain the Delaunay Triangulation
- While there are poor elements (`Clean` move)
  - Destroy a Poor Quality Element by Inserting the Circumcenter
  - Update the Delaunay

Delaunay Refinement Algorithm



- Obtain the Delaunay Triangulation
- While there are poor elements (`Clean` move)
  - Destroy a Poor Quality Element by Inserting the Circumcenter
  - Update the Delaunay

Delaunay Refinement Algorithm

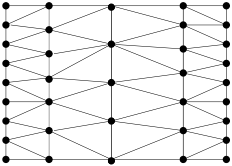


- Obtain the Delaunay Triangulation
- While there are poor elements (`Clean` move)
  - Destroy a Poor Quality Element by Inserting the Circumcenter
  - Update the Delaunay

Incremental Delaunay Refinement Algorithms

- Skinny triangles really happen in real examples!

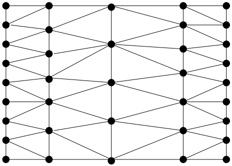
Ruppert's Algorithm Guarantees



- Theorem (Ruppert): This terminates with

$$|E| \in \Omega(\mathrm{lfs})$$

Ruppert's Algorithm Guarantees

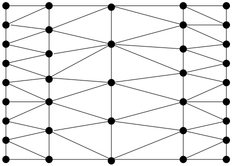


- Theorem (Ruppert): This terminates with

$$|E| \in \Omega(\mathrm{lfs})$$

- By Design: All output elements have quality guarantees

Ruppert's Algorithm Guarantees



- Theorem (Ruppert): This terminates with

$$|E| \in \Omega(\mathrm{lfs})$$

- By Design: All output elements have quality guarantees
- Nontrivial Fact: The output size is $O(1)$-Optimal.

Runtime Concerns

- Good Average-Case Runtime Maybe?

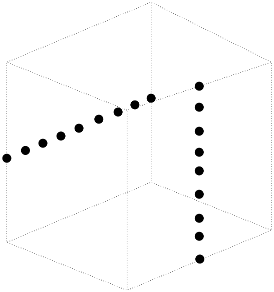
Runtime Concerns

- Good Average-Case Runtime Maybe?
- Bounded below by time to obtain the Delaunay triangulation. Therefore: worst case is: $\Omega(n^{\lceil d/2 \rceil})$

Runtime Concerns

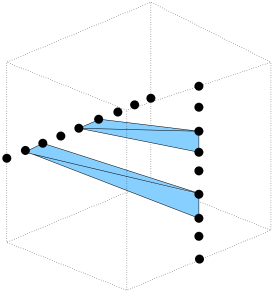
- Good Average-Case Runtime Maybe?
- Bounded below by time to obtain the Delaunay triangulation. Therefore: worst case is: $\Omega(n^{\lceil d/2 \rceil})$
- Thus 3-D space/time is $\Omega(n^2)$

$\Theta(n^2)$ Configurations Can Happen in Practice



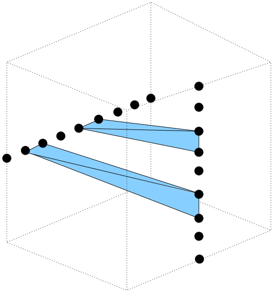
- Arises due to skew edges

$\Theta(n^2)$ Configurations Can Happen in Practice



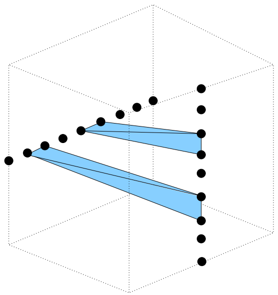
- Arises due to skew edges
- Delaunay Connectivity has all Vertical/Horizontal pairs: $(n/2)^2$

$\Theta(n^2)$ Configurations Can Happen in Practice



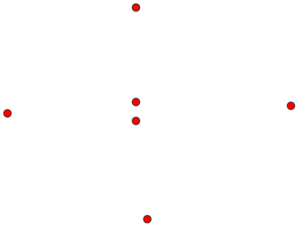
- Arises due to skew edges
- Delaunay Connectivity has all Vertical/Horizontal pairs: $(n/2)^2$
- Never actually contained in Final Output Mesh

$\Theta(n^2)$ Configurations Can Happen in Practice



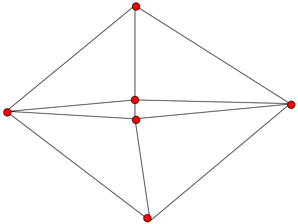
- Arises due to skew edges
- Delaunay Connectivity has all Vertical/Horizontal pairs: $(n/2)^2$
- Never actually contained in Final Output Mesh
  - *How can we avoid creating such intermediate structures?*

Two Competing Goals



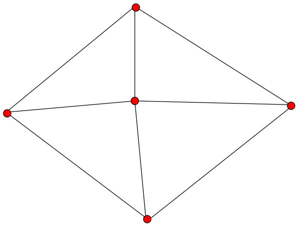
- Opposing Goals of Quality and Conformity Create Work

Two Competing Goals



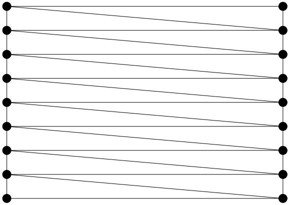
- Opposing Goals of Quality and Conformity Create Work

Two Competing Goals



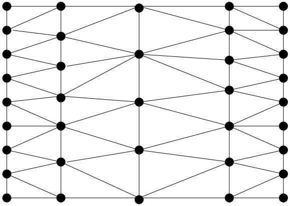
- Opposing Goals of Quality and Conformity Create Work

Two Competing Goals



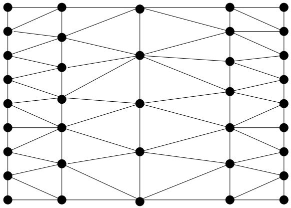
- Opposing Goals of Quality and Conformity Create Work
- Ruppert's Algorithm: Always Conforming, Gradually Quality

Two Competing Goals



- Opposing Goals of Quality and Conformity Create Work
- Ruppert's Algorithm: Always Conforming, Gradually Quality

Two Competing Goals

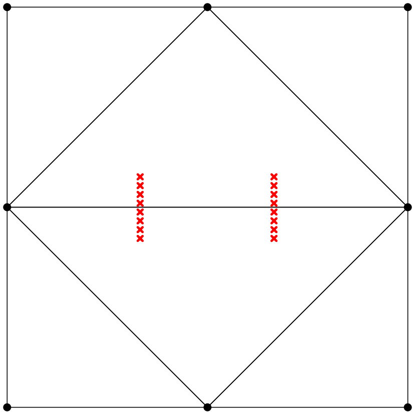


- Opposing Goals of Quality and Conformity Create Work
- Ruppert's Algorithm: Always Conforming, Gradually Quality
- **SVR Main Idea:** Always Quality, Gradually Conforming

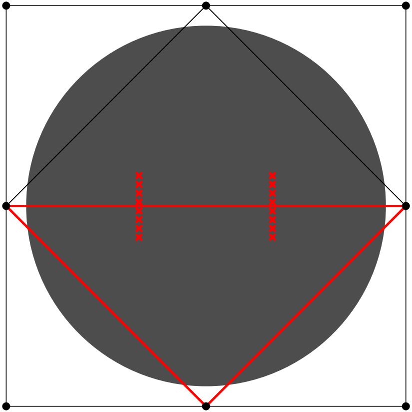
SVR in Abstract

- Outer Loop Invariant: Mesh Is Quality
- While Mesh is not Conforming
    - Try to Conform a Little Bit More
    - *While* Mesh is not Quality
        - Destroy Poor Quality Element (Insert it's CC, Update Delaunay)

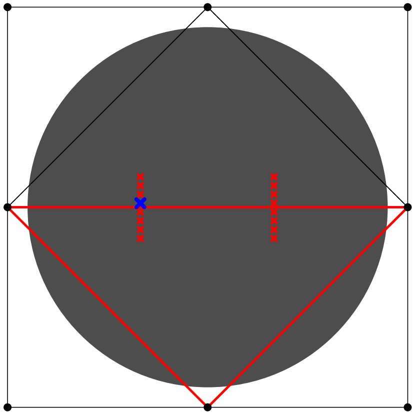
SVR in Action



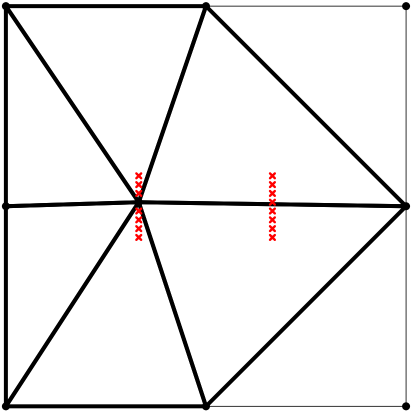
SVR in Action



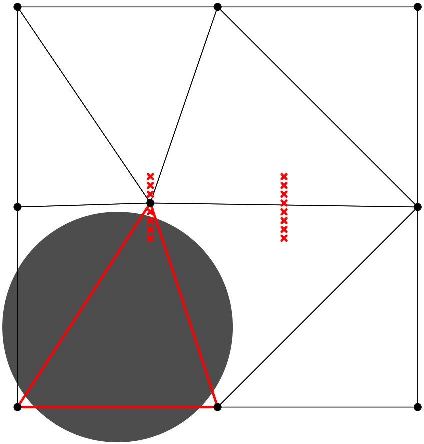
SVR in Action



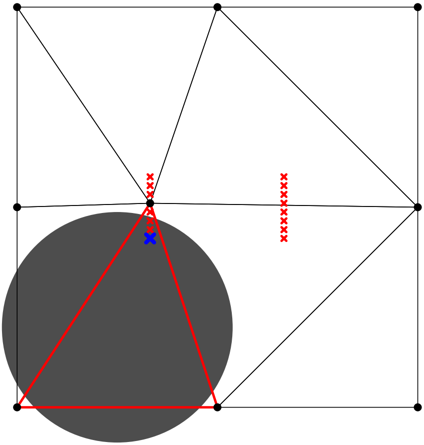
SVR in Action



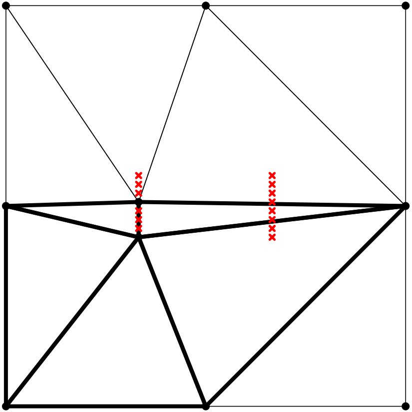
SVR in Action



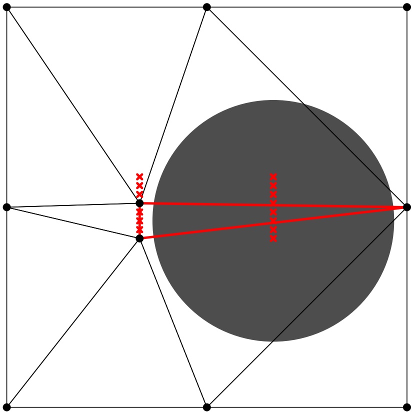
SVR in Action



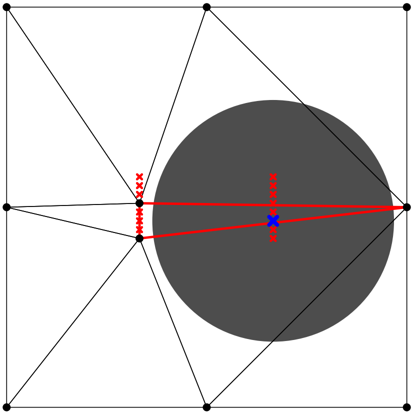
SVR in Action



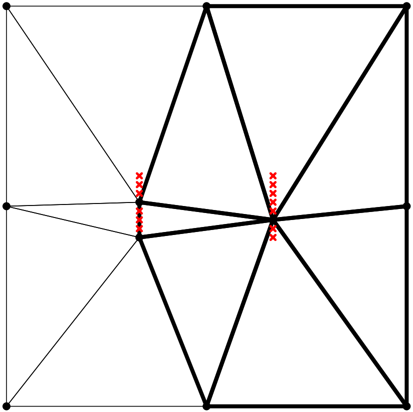
SVR in Action



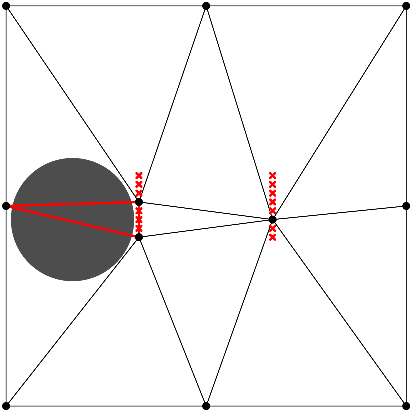
SVR in Action



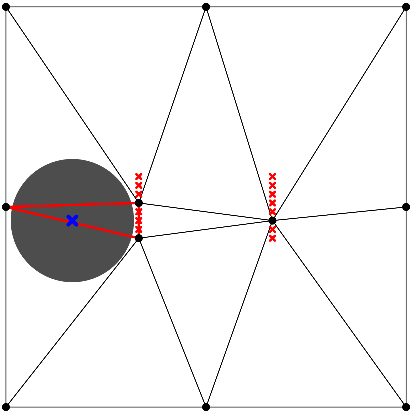
SVR in Action



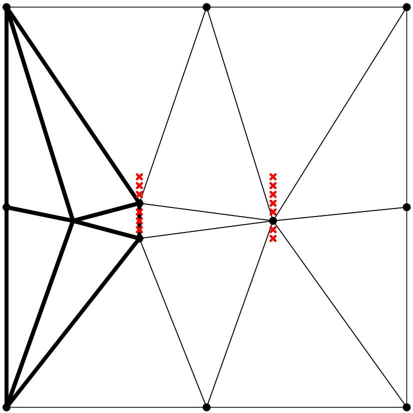
SVR in Action



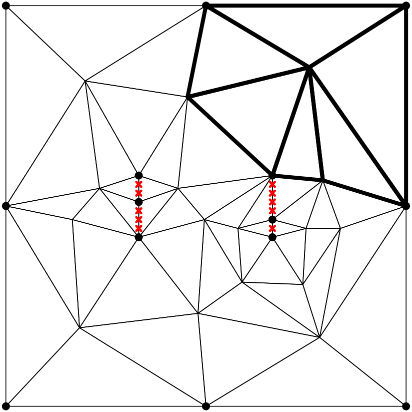
SVR in Action



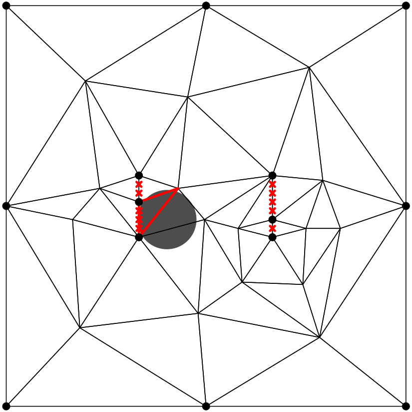
SVR in Action



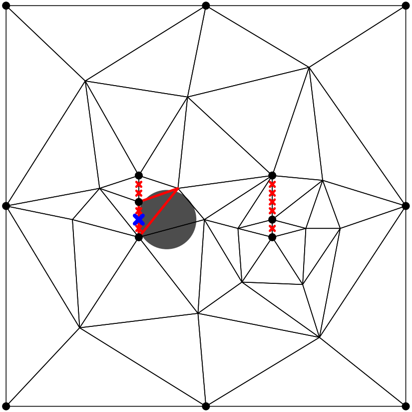
SVR in Action



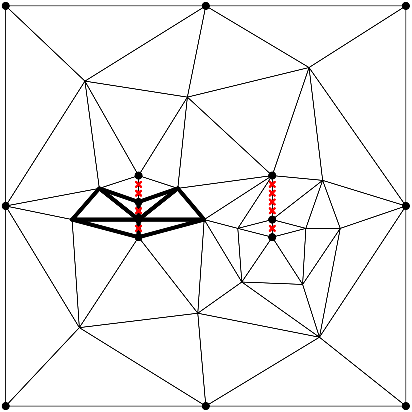
SVR in Action



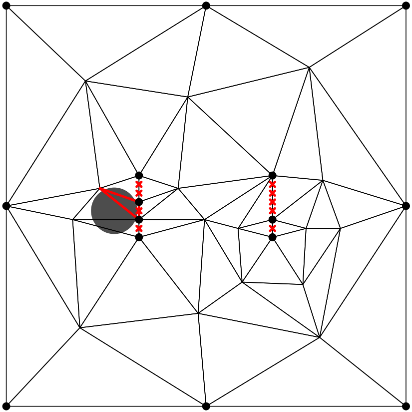
SVR in Action



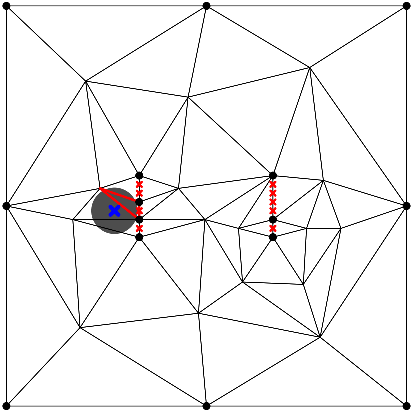
SVR in Action



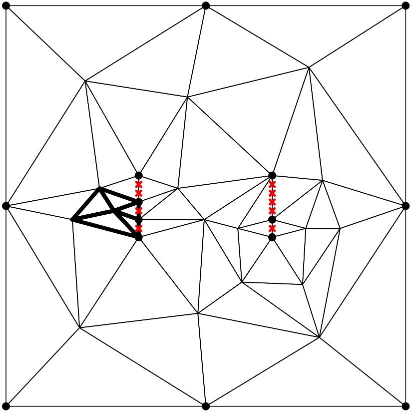
SVR in Action



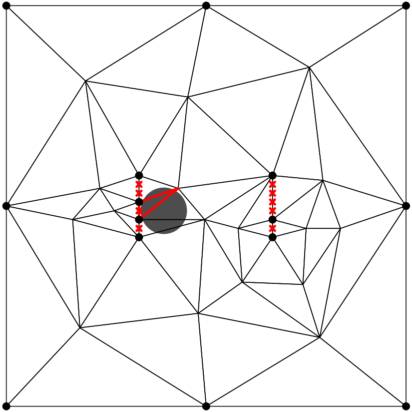
SVR in Action



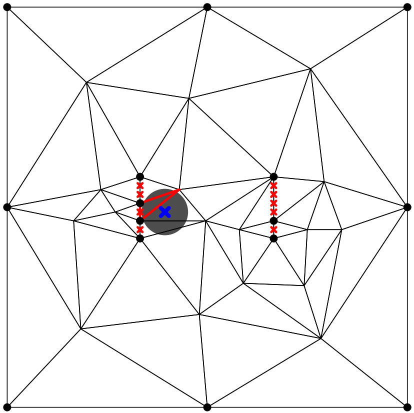
SVR in Action



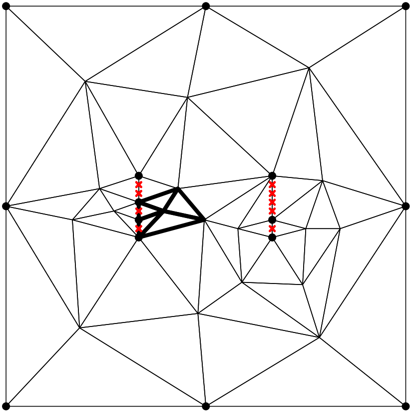
SVR in Action



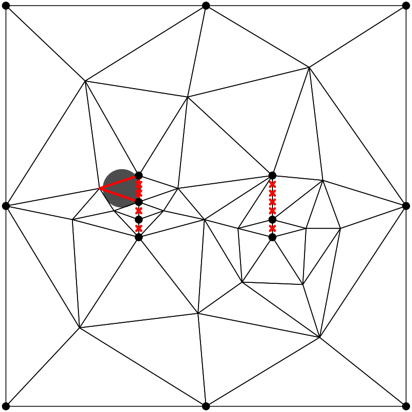
SVR in Action



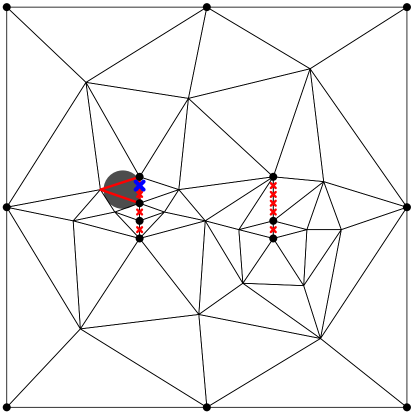
SVR in Action



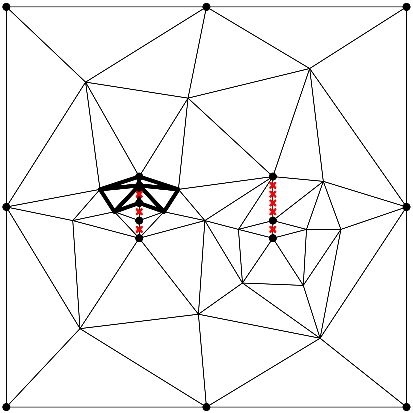
SVR in Action



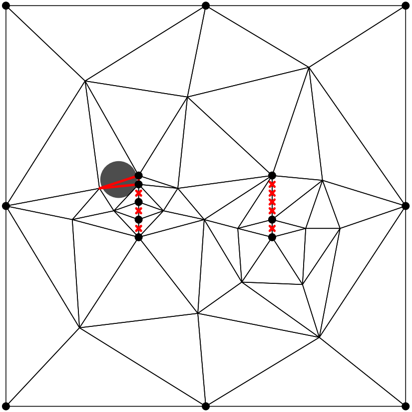
SVR in Action



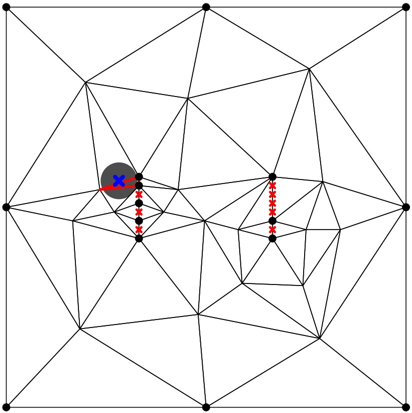
SVR in Action



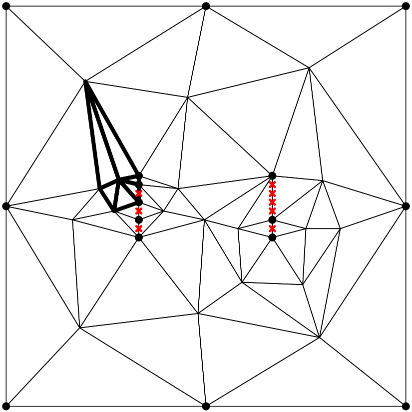
SVR in Action



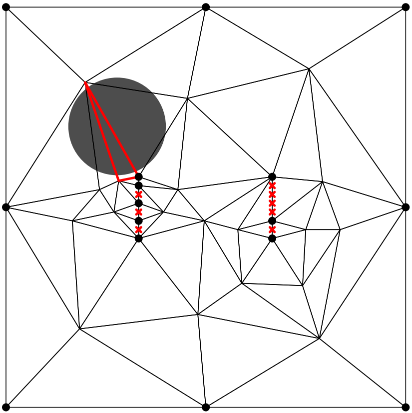
SVR in Action



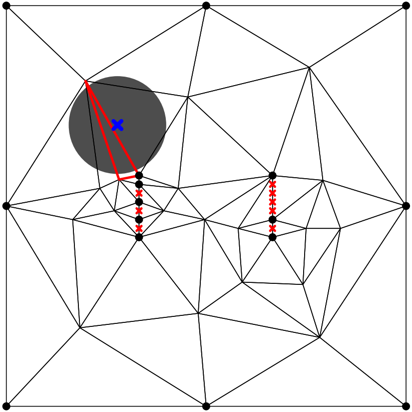
SVR in Action



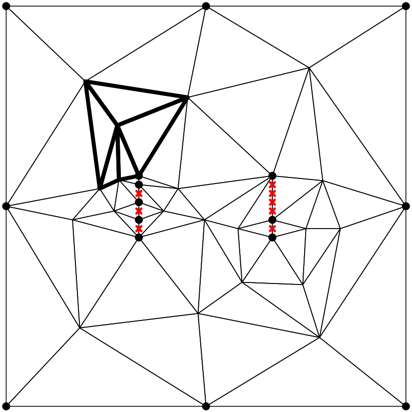
SVR in Action



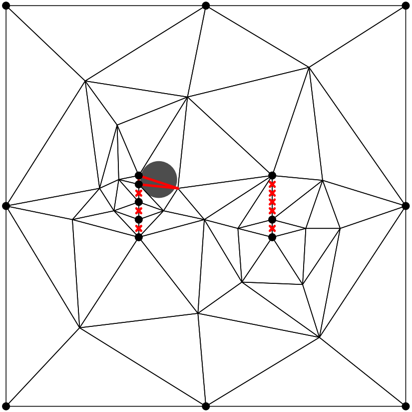
SVR in Action



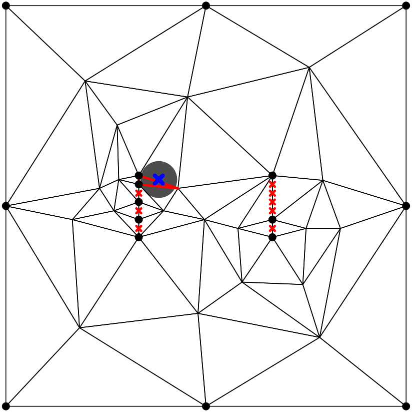
SVR in Action



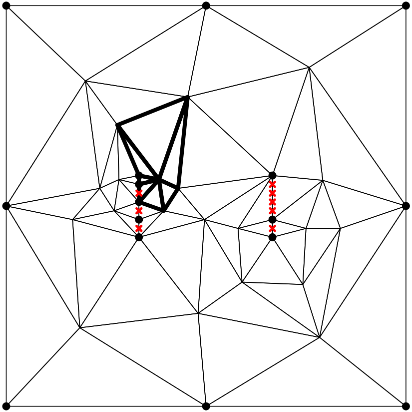
SVR in Action



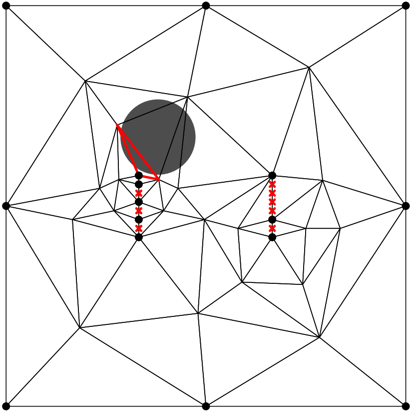
SVR in Action



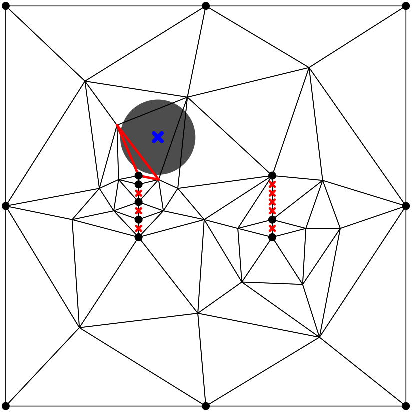
SVR in Action



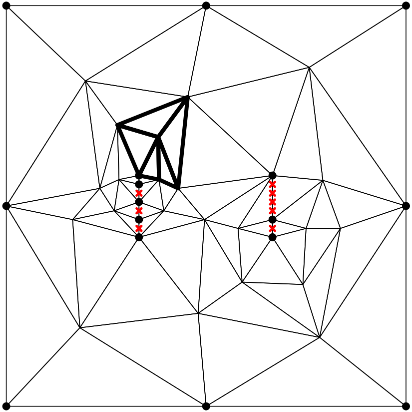
SVR in Action



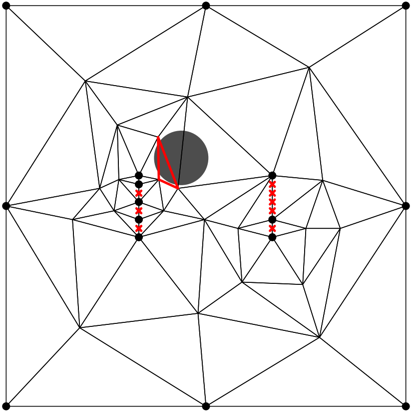
SVR in Action



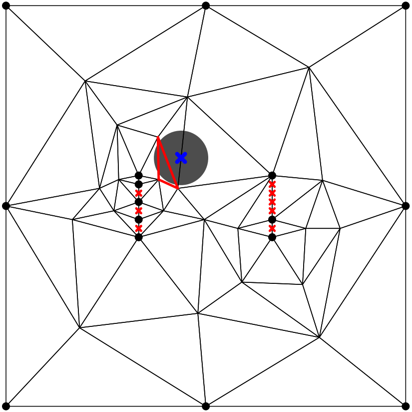
SVR in Action



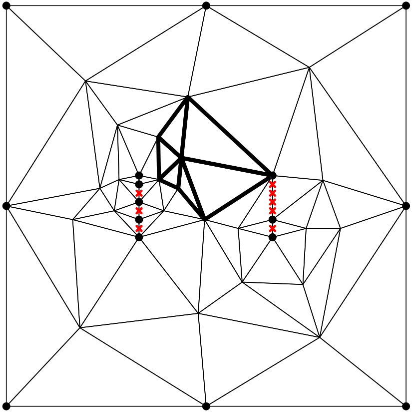
SVR in Action



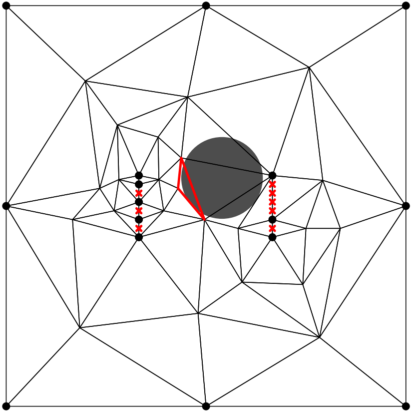
SVR in Action



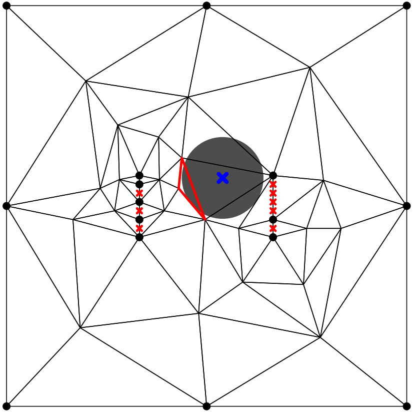
SVR in Action



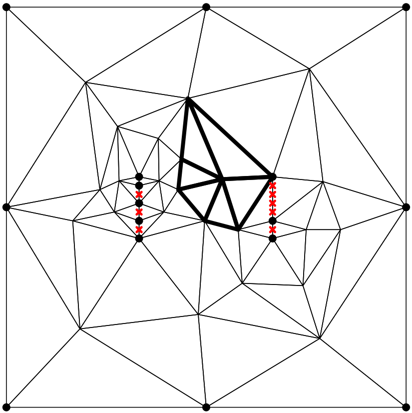
SVR in Action



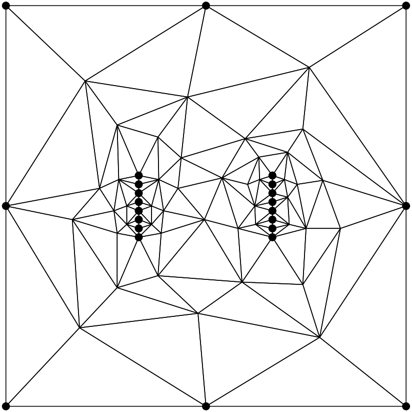
SVR in Action



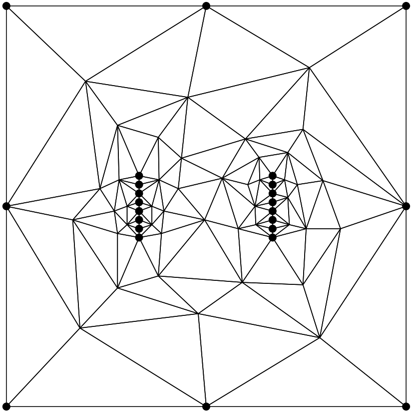
SVR in Action



SVR in Action

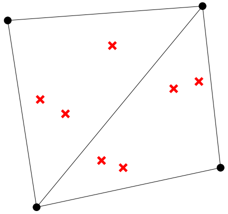


SVR in Action



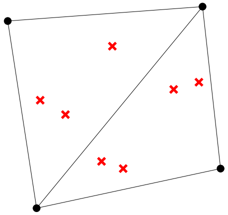
Maintaining Quality, Gradually Conform
Gradual Mesh Size Decrease

Try to Conform a Little Bit More . . .



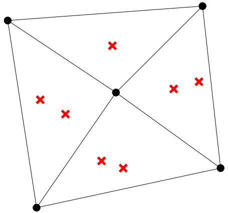
- **Break** Move

Try to Conform a Little Bit More . . .



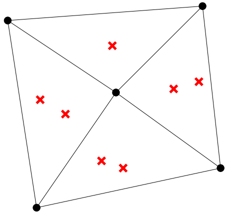
- **Break** Move
- Pick some cell that contains uninserted points still doesn't conform

Try to Conform a Little Bit More . . .



- **Break** Move
- Pick some cell that contains uninserted points still doesn't conform
- Try to insert furthest corner of the cell

Try to Conform a Little Bit More . . .



- **Break** Move
- Pick some cell that contains uninserted points still doesn't conform
- Try to insert furthest corner of the cell
- **Eagerly** keep track of where I still need to conform:

The Priority Queue for SVR

- Cell-Queue (Tet)

The Priority Queue for SVR

- Cell-Queue (Tet)
- Cells in Queue

The Priority Queue for SVR

- Cell-Queue (Tet)
- Cells in Queue
  - Bad-Aspect-Ratio Cells (Clean Move)

The Priority Queue for SVR

- Cell-Queue (Tet)
- Cells in Queue
  - Bad-Aspect-Ratio Cells (Clean Move)
  - Cells containing uninserted points (Break Move)

The Priority Queue for SVR

- Cell-Queue (Tet)
- Cells in Queue
  - Bad-Aspect-Ratio Cells (Clean Move)
  - Cells containing uninserted points (Break Move)
- Process Cell in Cell-Queue with
  TRY-TO-INSERT(furthest point of Cell)

The Priority Queue for SVR

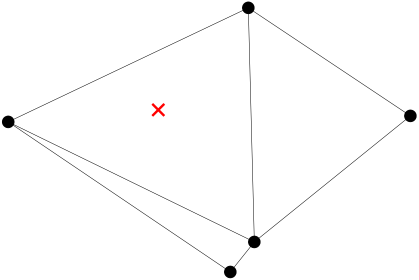
- Cell-Queue (Tet)
- Cells in Queue
  - Bad-Aspect-Ratio Cells (Clean Move)
  - Cells containing uninserted points (Break Move)
- Process Cell in Cell-Queue with **TRY-TO-INSERT(furthest point of Cell)**
- Priority clean moves first

Inserting Points

TRY-TO-INSERT(P) IF $\exists$ "nearby" uninserted point Q THEN add Q ELSE P

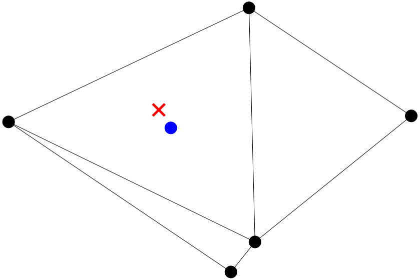
Priority Queue: Clean before Breaks

Conflicts Between Goals



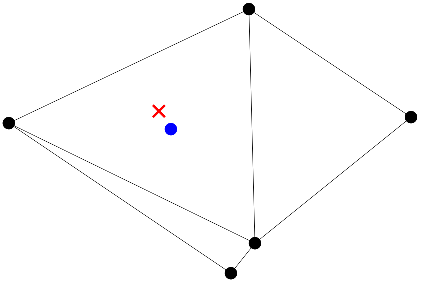
- Notice the Break Move need not do any conforming!

Conflicts Between Goals



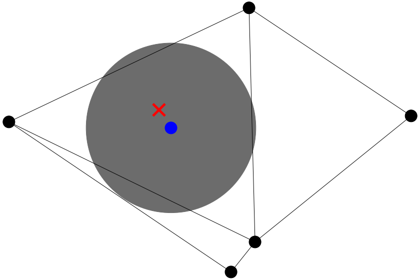
- Notice the Break Move need not do any conforming!
- Whenever we *Destroy Element*, we might need to **yield**

Conflicts Between Goals



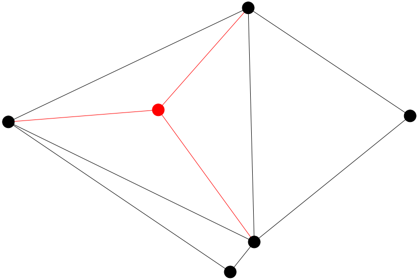
- Notice the Break Move need not do any conforming!
- Whenever we *Destroy Element*, we might need to **yield**
- If a Queue Point is *relatively close*, insert that instead

Conflicts Between Goals



- Notice the Break Move need not do any conforming!
- Whenever we *Destroy Element*, we might need to **yield**
- If a Queue Point is *relatively close*, insert that instead

Conflicts Between Goals



- Notice the Break Move need not do any conforming!
- Whenever we *Destroy Element*, we might need to **yield**
- If a Queue Point is *relatively close*, insert that instead

Conflicts Between Goals

- Notice the Break Move need not do any conforming!
- Whenever we *Destroy Element*, we might need to **yield**
- If a Queue Point is *relatively close*, insert that instead
- Reasoning behind the Eagerness of the Conformity Queue

Termination Guarantee

This yielding is enough to give us termination with

$$|E| \in \Omega(\mathrm{lfs})$$

By design, we have output with quality elements and conforming, hence we output an $O(1)$-Optimal Mesh.

Termination Guarantee

This yielding is enough to give us termination with

$$|E| \in \Omega(\mathrm{lfs})$$

By design, we have output with quality elements and conforming, hence we output an $O(1)$-Optimal Mesh.
Were we successful in avoiding the bad intermediate stages?

Sparse Voronoi Refinement

- A Re-Scheduled Version of a Traditional Incremental Meshing Algorithm.

Sparse Voronoi Refinement

- A Re-Scheduled Version of a Traditional Incremental Meshing Algorithm.
- Yielding procedure can be varied

Sparse Voronoi Refinement

- A Re-Scheduled Version of a Traditional Incremental Meshing Algorithm.
- Yielding procedure can be varied
  - Yielding less often is faster

Sparse Voronoi Refinement

- A Re-Scheduled Version of a Traditional Incremental Meshing Algorithm.
- Yielding procedure can be varied
  - Yielding less often is faster
  - Yielding more often is closer to original schedule (better mesh size guarantee).

Insuring Conforming by Maintaining empty Balls

- Each Edge is meshed into segments and protective balls.

Insuring Conforming by Maintaining empty Balls

- Each Edge is meshed into segments and protective balls.
- Each Face is meshed into triangles and protective balls.

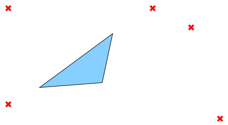
Insuring Conforming by Maintaining empty Balls

- Each Edge is meshed into segments and protective balls.
- Each Face is meshed into triangles and protective balls.

Balls and Multiple Meshes

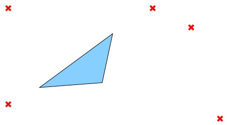
- In the Queue, we add protective *Balls* around each feature.

Balls and Multiple Meshes



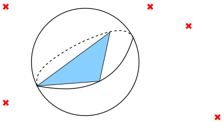
- In the Queue, we add protective *Balls* around each feature.
- These get handled just like conforming to points (0-dimensional balls)

Balls and Multiple Meshes



- In the Queue, we add protective *Balls* around each feature.
- These get handled just like conforming to points (0-dimensional balls)
- Add one operation, to subdivide a Ball

Balls and Multiple Meshes

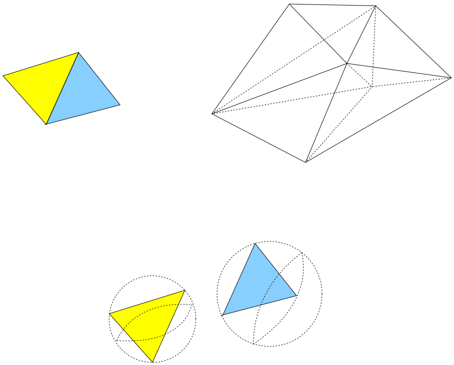


- In the Queue, we add protective *Balls* around each feature.
- These get handled just like conforming to points (0-dimensional balls)
- Add one operation, to subdivide a Ball
- Maintain a Lower-Dimensional Mesh/Subdivision of Each Feature

Balls and Multiple Meshes

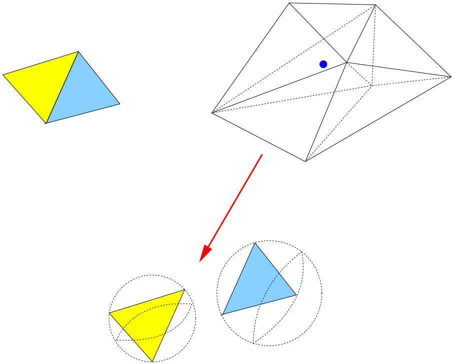
- In the Queue, we add protective *Balls* around each feature.
- These get handled just like conforming to points (0-dimensional balls)
- Add one operation, to subdivide a Ball
- Maintain a Lower-Dimensional Mesh/Subdivision of Each Feature
- Lower Dimensional Meshes Recursively have their own conformity queues.

Handling Features



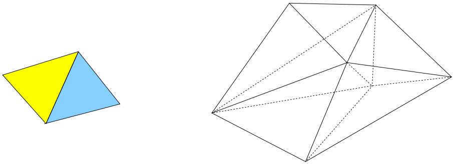
3D Mesh, Queue of Uninserted Features, 2D Mesh

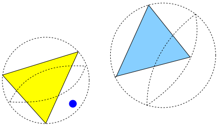
Handling Features



3D Mesh Wants to Insert a Point

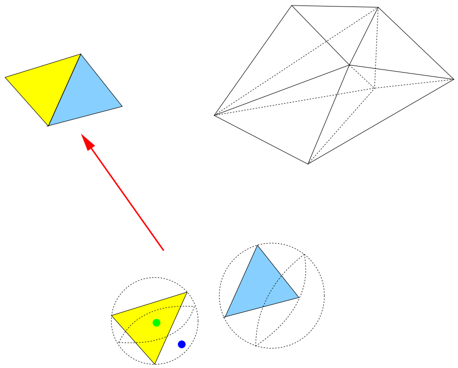
Handling Features





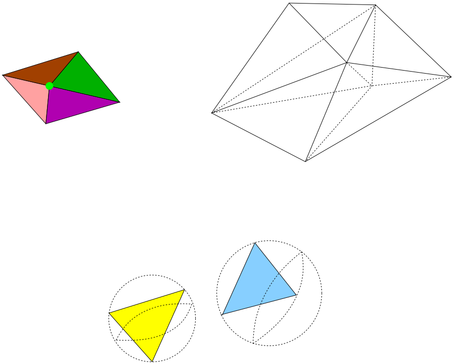
Does it Encroach on Any Balls on the Queue?

Handling Features



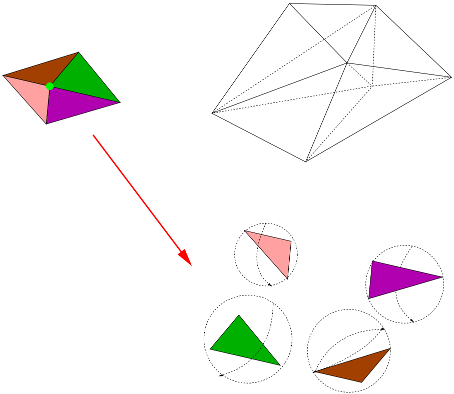
Yield to a lower Dimensional Insertion

Handling Features



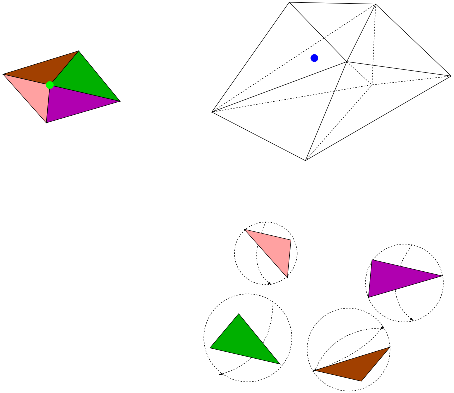
Perform an Insertion in the Lower Dimensional Mesh

Handling Features



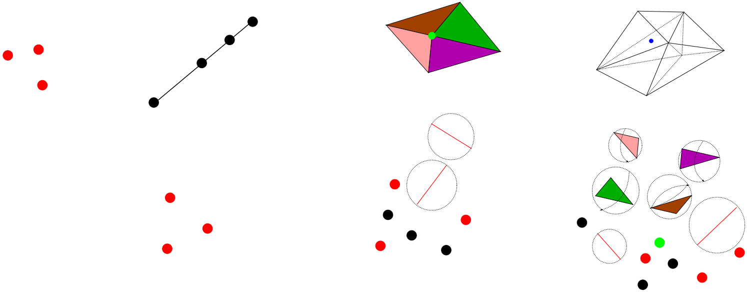
Update the Higher Dimensional Queue

Handling Features



Try Again

Handling Features



In General, Meshes and Queues at Every Level

Cells Points and Balls

Abstract Objects:

- Cell: A Voronoi cell of an inserted point

Structures:

Cells Points and Balls

Abstract Objects:

- Cell: A Voronoi cell of an inserted point
- Point: An uninserted input or Steiner point

Structures:

Cells Points and Balls

Abstract Objects:

- Cell: A Voronoi cell of an inserted point
- Point: An uninserted input or Steiner point
- Ball: A protective Voronoi ball

Structures:

Cells Points and Balls

Abstract Objects:

- Cell: A Voronoi cell of an inserted point
- Point: An uninserted input or Steiner point
- Ball: A protective Voronoi ball

Structures:

- For each Cell a list of Points in it

Cells Points and Balls

Abstract Objects:

- Cell: A Voronoi cell of an inserted point
- Point: An uninserted input or Steiner point
- Ball: A protective Voronoi ball

Structures:

- For each Cell a list of Points in it
- For each Cell a list of Balls intersecting it.

Cells Points and Balls

Abstract Objects:

- Cell: A Voronoi cell of an inserted point
- Point: An uninserted input or Steiner point
- Ball: A protective Voronoi ball

Structures:

- For each Cell a list of Points in it
- For each Cell a list of Balls intersecting it.
- For each Ball a list of Cells intersecting it.

Cells Points and Balls

Abstract Objects:

- Cell: A Voronoi cell of an inserted point
- Point: An uninserted input or Steiner point
- Ball: A protective Voronoi ball

Structures:

- For each Cell a list of Points in it
- For each Cell a list of Balls intersecting it.
- For each Ball a list of Cells intersecting it.
- For each Point a list of Cells containing it.

Always Quality Mesh

- Outer Loop Invariant: Mesh Is Quality
- Until Mesh is Conforming
  - Try to Conform a Little Bit More
  - Until Mesh is Quality
    - Destroy Poor Quality Element (Insert it's CC)

Always Quality Mesh

- Outer Loop Invariant: Mesh Is Quality
- Until Mesh is Conforming
  - Try to Conform a Little Bit More
  - Until Mesh is Quality
    - Destroy Poor Quality Element (Insert it's CC)
- Always Have Quality at the Outer Loop

Always Quality Mesh

- Outer Loop Invariant: Mesh Is Quality
- Until Mesh is Conforming
  - Try to Conform a Little Bit More
  - Until Mesh is Quality
    - Destroy Poor Quality Element (Insert it's CC)

- Always Have Quality at the Outer Loop
- Our worry is that sometime during the Inner Loop, we could reach a poor state

Always Quality Mesh

- Outer Loop Invariant: Mesh Is Quality
- Until Mesh is Conforming
  - Try to Conform a Little Bit More
  - Until Mesh is Quality
    - Destroy Poor Quality Element (Insert it's CC)

- Always Have Quality at the Outer Loop
- Our worry is that sometime during the Inner Loop, we could reach a poor state
- In Fact, we always have a "Weak-Quality" bound.

Overall Runtime

- We have the Weak-Quality Invariant

Overall Runtime

- We have the Weak-Quality Invariant
- Want to get $O(n \log L/s + m)$ runtime

Overall Runtime

- We have the Weak-Quality Invariant
- Want to get $O(n \log L/s + m)$ runtime
- Split:
  $O(m)$ time Building/Maintaining the mesh
  $O(n \log L/s)$ time maintaining the Conformity Queue

Quality Gives Degree Bound

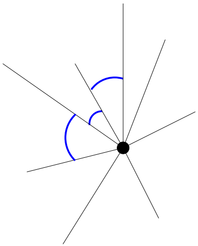
- **Theorem:** *[MTTW96]* Every vertex in a good radius-edge mesh has constant degree.

Quality Gives Degree Bound

- **Theorem:** *[MTTW96]* Every vertex in a good radius-edge mesh has constant degree.
- SVR is always updating a **Sparse** Mesh.

Quality Gives Degree Bound

- **Theorem:** *[MTTW96]* Every vertex in a good radius-edge mesh has constant degree.
- SVR is always updating a **Sparse** Mesh.



Sparse Mesh Updating

- New Vertices Are Constant Degree After Insertion

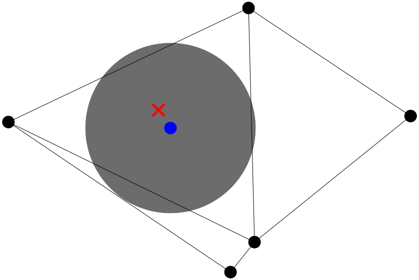
Sparse Mesh Updating

- New Vertices Are Constant Degree After Insertion
- Each Insertion Took Constant Work

Sparse Mesh Updating

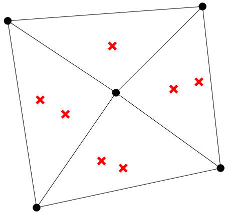
- New Vertices Are Constant Degree After Insertion
- Each Insertion Took Constant Work
- Total mesh construction work is $O(m)$.

Point Location Events



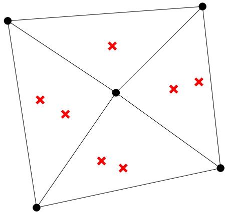
- Two types of Events:
  - Look for Someone to Yield To

Point Location Events



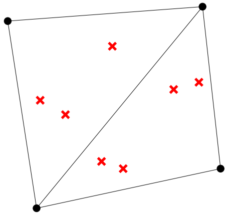
- Two types of Events:
  - Look for Someone to Yield To
  - Relocation after a mesh insertion
- Cost is Queue points handled

Point Location Events



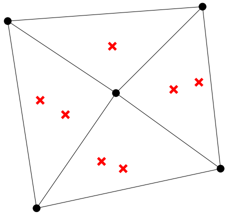
- Two types of Events:
  - Look for Someone to Yield To
  - Relocation after a mesh insertion
- Cost is Queue points handled
- Two types happen at the "same time" with the "same cost"

Work Per Event



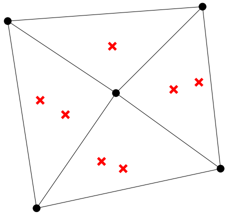
- One Event could take large work, many queue points handled. (Naively $O(mn)$)

Work Per Event



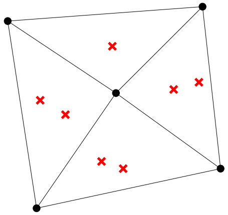
- One Event could take large work, many queue points handled. (Naively $O(mn)$)

Work Per Event



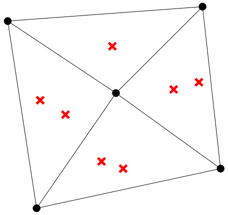
- One Event could take large work, many queue points handled. (Naively $O(mn)$)
- **Amortized Analysis**

Work Per Event



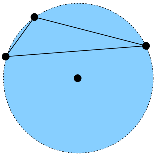
- One Event could take large work, many queue points handled. (Naively $O(mn)$)
- **Amortized Analysis**
- Charge Event Work to the queue points involved (k events per queue point)

Work Per Event



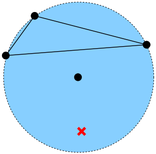
- One Event could take large work, many queue points handled. (Naively $O(mn)$)
- **Amortized Analysis**
- Charge Event Work to the queue points involved (k events per queue point)
- Total Work: $O(nk)$

Bounding k



- Geometric "Scale" r of the insertion of some vertex v

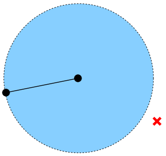
Bounding k



- Geometric "Scale" r of the insertion of some vertex v
- **Theorem:** In a quality mesh, if an insertion affects a queue point q, then:

$$r \in \Omega(||v, q||)$$

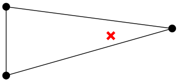
Bounding k



- Geometric “Scale” r of the insertion of some vertex v
- **Theorem:** In a quality mesh, if an insertion affects a queue point q, then:

$$r \in \Omega(||v, q||)$$

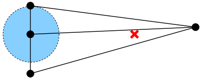
Bounding k



- Geometric "Scale" r of the insertion of some vertex v
- **Theorem:** In a quality mesh, if an insertion affects a queue point q, then:

$$r \in \Omega(||v, q||)$$

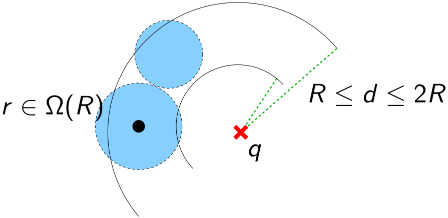
Bounding k



- Geometric "Scale" r of the insertion of some vertex v
- **Theorem:** In a quality mesh, if an insertion affects a queue point q, then:

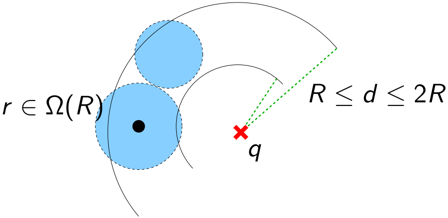
$$r \in \Omega(||v, q||)$$

A Packing Argument



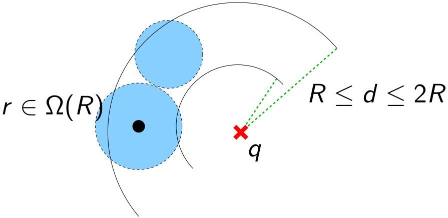
- Radius Doubling Annulus around q

A Packing Argument



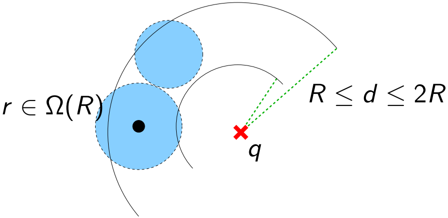
- Radius Doubling Annulus around q
- Balls in Each Annulus are $\Omega(R)$, Essentially Disjoint

A Packing Argument



- Radius Doubling Annulus around q
- Balls in Each Annulus are $\Omega(R)$, Essentially Disjoint
- Volume of Annulus is $O(R^d)$, Volume of Event is $\Omega(R^d)$

A Packing Argument



- Radius Doubling Annulus around q
- Balls in Each Annulus are $\Omega(R)$, Essentially Disjoint
- Volume of Annulus is $O(R^d)$, Volume of Event is $\Omega(R^d)$
- $O(1)$ Events per Annulus affecting q

Total Point Location Time

- How many total annulii around q? Largest is L, smallest?

Total Point Location Time

- How many total annulii around q? Largest is L, smallest?
- $|E| \in \Omega(\mathrm{lfs})$, thus $|E| \in \Omega(s)$

Total Point Location Time

- How many total annulii around q? Largest is L, smallest?
- $|E| \in \Omega(\mathrm{lfs})$, thus $|E| \in \Omega(s)$
- $\log L/s$

Total Point Location Time

- How many total annulii around q? Largest is L, smallest?
- $|E| \in \Omega(\mathrm{lfs})$, thus $|E| \in \Omega(s)$
- $\log L/s$
- $O(n \log L/s)$ total work to maintain Conformity Queue

Intersection Sizes

Theorem

Suppose $\mathcal{V}$ bded aspect ratio Voronoi diagram and B is a ball with no points of $\mathcal{V}$ in its interior then B intersects a bded number cells.

False: Need center of B is in convex closure of points of $\mathcal{V}$.

Intersection Sizes

Intersection Sizes

- **Theorem**

  Over life of SVR #cells containing an input point $O(\log L/s)$.

Intersection Sizes

- **Theorem**

  Over life of SVR #cells containing an input point $O(\log L/s)$.

Intersection Sizes

- **Theorem**

  Over life of SVR #cells containing an input point $O(\log L/s)$.

- **Theorem**

  Over life of SVR #cells containing an Steiner point $O(1)$.

Intersection Sizes

- **Theorem**

  Over life of SVR #cells containing an input point $O(\log L/s)$.

- **Theorem**

  Over life of SVR #cells containing an Steiner point $O(1)$.

Intersection Sizes

- **Theorem**

  Over life of SVR #cells containing an input point $O(\log L/s)$.

- **Theorem**

  Over life of SVR #cells containing an Steiner point $O(1)$.

- **Theorem**

  Over life of SVR #cells intersecting an original ball $O(\log L/s)$.

Intersection Sizes

- **Theorem**

  Over life of SVR #cells containing an input point $O(\log L/s)$.

- **Theorem**

  Over life of SVR #cells containing an Steiner point $O(1)$.

- **Theorem**

  Over life of SVR #cells intersecting an original ball $O(\log L/s)$.

Intersection Sizes

- **Theorem**

  Over life of SVR #cells containing an input point $O(\log L/s)$.

- **Theorem**

  Over life of SVR #cells containing an Steiner point $O(1)$.

- **Theorem**

  Over life of SVR #cells intersecting an original ball $O(\log L/s)$.

- **Theorem**

  Over life of SVR #cells intersecting a created ball $O(1)$.

Overall Runtime Bound

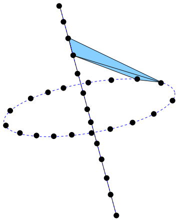
- $O(n \log L/s + m)$

Overall Runtime Bound

- $O(n \log L/s + m)$
- Notice: $O(m)$ Optimal Space Usage because of Sparsity

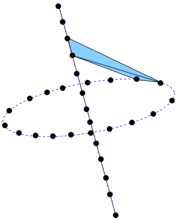
Research Implementation (3D Point Sets)

Quadratic Delaunay Example
Problem Size: $n = 1000$



Research Implementation (3D Point Sets)

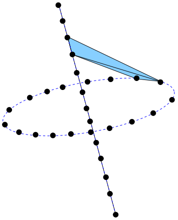
Quadratic Delaunay Example
Problem Size: $n = 1000$



Algorithm	Lifetime	MaxTets	Worst Degree	Output
Pyramid	1.3M Tets	1.0M	1000 (84 avg.)	14K V, 87K Tets
SVR	308K Tets	81K	39 (24 avg.)	13K V, 81K Tets

Research Implementation (3D Point Sets)

Quadratic Delaunay Example
Problem Size: $n = 1000$



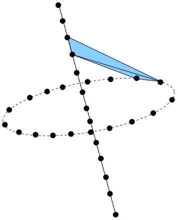
Algorithm	Lifetime	MaxTets	Worst Degree	Output
Pyramid	1.3M Tets	1.0M	1000 (84 avg.)	14K V, 87K Tets
SVR	308K Tets	81K	39 (24 avg.)	13K V, 81K Tets

$n = 10000$, Pyramid runs out of Memory (1.25 Gig Laptop)

SVR outputs 774K Tets, sees 2.7M Lifetime, worst case degree is 41

Research Implementation (3D Point Sets)

Quadratic Delaunay Example
Problem Size: $n = 1000$



Algorithm	Lifetime	MaxTets	Worst Degree	Output
Pyramid	1.3M Tets	1.0M	1000 (84 avg.)	14K V, 87K Tets
SVR	308K Tets	81K	39 (24 avg.)	13K V, 81K Tets

$n = 10000$, Pyramid runs out of Memory (1.25 Gig Laptop)
SVR outputs 774K Tets, sees 2.7M Lifetime, worst case degree is 41

Conclusions

- New Meshing Algorithm
  Element Shape / Output Size / Conformity Guarantees.

Conclusions

- New Meshing Algorithm
  Element Shape / Output Size / Conformity Guarantees.
- Runtime Analysis: $O(n \log L/s + m)$

Conclusions

- New Meshing Algorithm
  Element Shape / Output Size / Conformity Guarantees.
- Runtime Analysis: $O(n \log L/s + m)$
- Reasonable to Implement

Future Work

- Better language to handle 3D geometry, eg, Trimmed Nurbs

Future Work

- Better language to handle 3D geometry, eg, Trimmed Nurbs
- Competitive output size algorithms for small input angles

Future Work

- Better language to handle 3D geometry, eg, Trimmed Nurbs
- Competitive output size algorithms for small input angles
- Meshing for dirty geometries

Future Work

- Better language to handle 3D geometry, eg, Trimmed Nurbs
- Competitive output size algorithms for small input angles
- Meshing for dirty geometries
- Settle Tet verses Hex meshing issues

Future Work

- Better language to handle 3D geometry, eg, Trimmed Nurbs
- Competitive output size algorithms for small input angles
- Meshing for dirty geometries
- Settle Tet verses Hex meshing issues
- Better handling of slivers

Future Work

- Better language to handle 3D geometry, eg, Trimmed Nurbs
- Competitive output size algorithms for small input angles
- Meshing for dirty geometries
- Settle Tet verses Hex meshing issues
- Better handling of slivers
- Replacing runtime term $\log L/s$ with $\log n$

Thanks!