Voronoi Diagram of Line Segments

New Features Added to Boost.Polygon

Google Summer of Code 2010

Experience Report

Outline

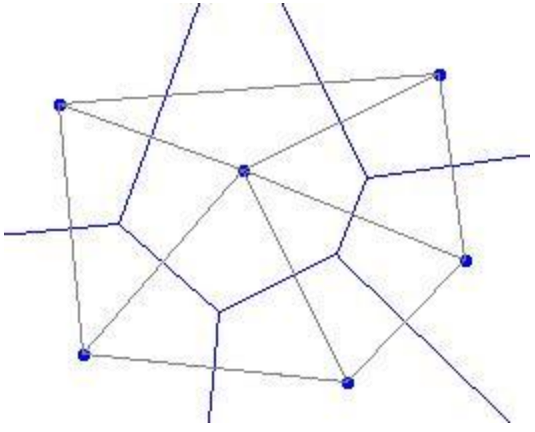
- Voronoi Problem Statement
- Motivation
- Timeline and History of the Project
- Explanation of Fortune’s Algorithm
- Numerical Robustness Problems/Solutions
- Output Examples
- Benchmark Results
- Plans For Integration into Polygon
- GSOC 2011 Edge Concepts Project
- Q&A

Voronoi Diagram of Points

- Given an input set of points called “sites”
- Compute bounded regions called “cells” for each site such the site enclosed by each cell is the closest site to all points within the cell
- Boundaries between cells called “Voronoi edge” are line segments equidistant from two sites
- Intersection of three or more voronoi edges creates a “Vornoi vertex”

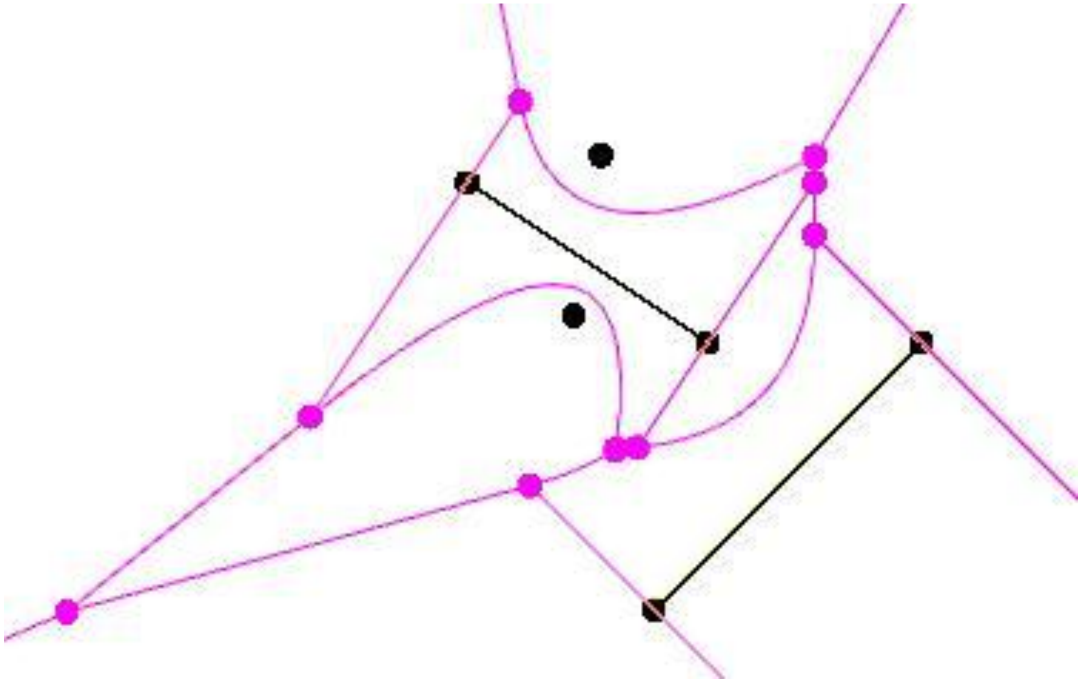
Delaunay Triangulation of Points

- Delaunay triangulation is the dual graph of Voronoi diagram
- Connecting each pair of sites associated with a voronoi edge with a line segment produces Delaunay triangulation



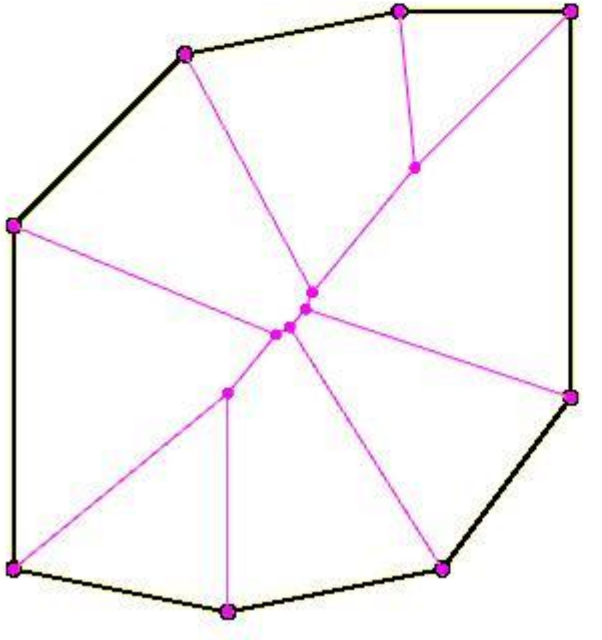
Voronoi Diagram Of Line Segments

- Line segments and points are “sites”
- Voronoi edge equidistant from point and line segments sites is parabolic arc



Medial Axis of Polygon

- The Voronoi Diagram of edges of a polygon in the interior region of the polygon produces the Medial Axis of the polygon



Nearest Neighbor Query

- Optimally find all pairs of closest points in a point set
- Optimally find which polygons in a polygon set enclose which points in a point set
- Optimally find all pairs of closest polygons in a polygon set
- Optimally find which polygons in one polygon set are inside, outside or partially overlapping polygons from another polygon set

One Algorithm to Rule Them All

- Solution to Voronoi Diagram of Line Segments solves also Voronoi Diagram of Points, Delaunay Triangulation of Points, Medial Axis and Nearest Neighbor problems
- A single implementation of Fortune’s sweepline algorithm for Vornoi Diagram of Line Segments allows interfaces for solving all these problems to be added to Boost.Polygon

Motivation

- Voronoi diagrams and the related problems have applications in many fields
  - Physics
  - Data compression
  - VSLI CAD
  - CAM (Computer Aided machinery)
  - GIS (Geospatial Information Systems)
  - Meshing

Timeline and History of the Project

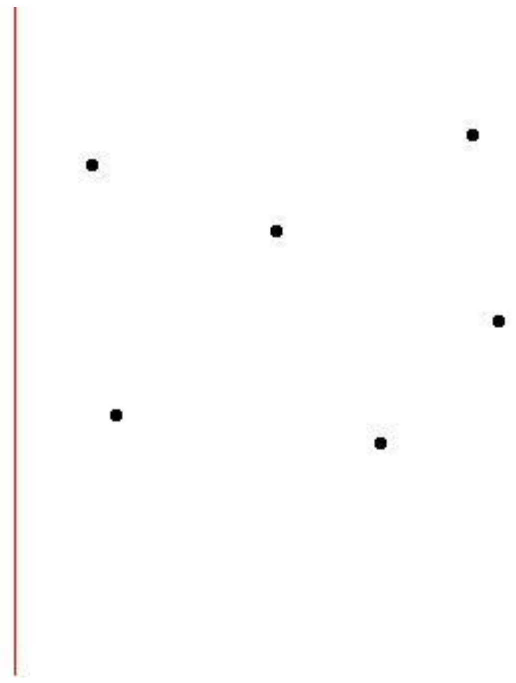
- March 2010
  - Posted computational geometry project Idea
  - Andrii and half a dozen other students expressed interest
- April 2010
  - Four strong student proposals submitted
  - Andrii selected
- May 2010
  - Work started
  - Reasearch of Voronoi Diagram Problem
  - Design of Implementation
- June 2010
  - Polygon released in Boost 1.44
  - Voronoi of points initial implemention
  - Testing of Voronoi of Points
- July 2010 - Midterm
  - QT based Voronoi Diagram visualizer implemented
  - Voronoi of Points Completed with Robust Predicates
  - Voronoi of Segments Started
  - Numerical Robustness Research
- August 2010 – Final
  - Voronoi of segments initial implementation
- September, October, November 2010
  - Voronoi diagram of line segments completed
  - Testing of voronoi of line segments
  - Visualizer updated
  - Refactoring
  - Robustness improvements
  - Performance improvements
- January 2011
  - Boostcon 2011 application

Currently ~4600 lines of library code

Fortune's Algorithm

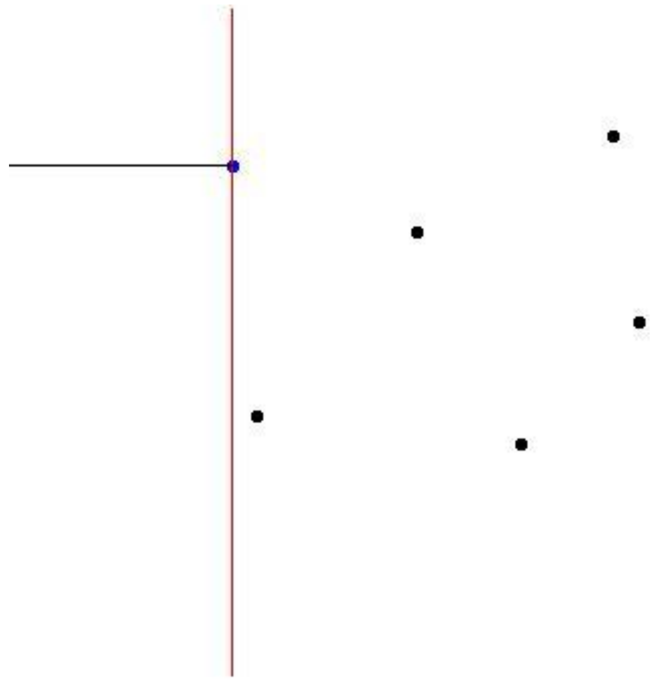
- Describes a sweepline algorithm for solving Voronoi Diagram of Points
- Gives the general idea for extending the algorithm to line segments
- O(n log n) complexity
- Divide an conquer algorithm and randomized incremental construction is also O(n log n)

Fortune’s Algorithm



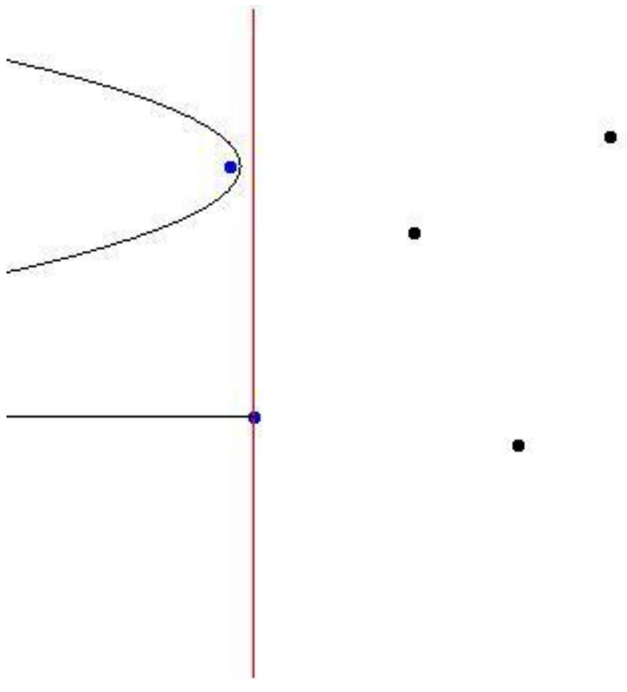
- Start with vertical sweepline at the left

Fortune's Algorithm



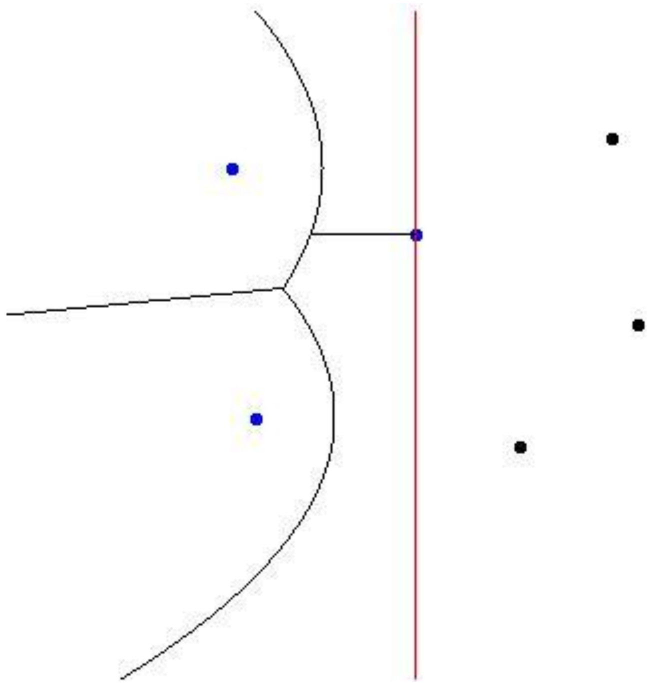
- Sweep until first site is reached
- Sites are sweepline events where work is done

Fortune's Algorithm



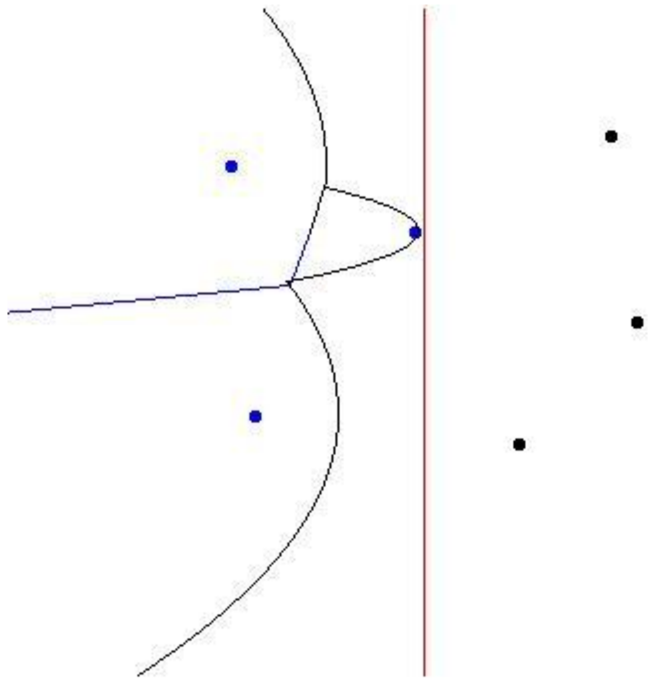
- When a site is reached a parabolic region is opened behind the sweep line
- The parabolic arc is equidistant from the site and the sweepline

Fortune's Algorithm



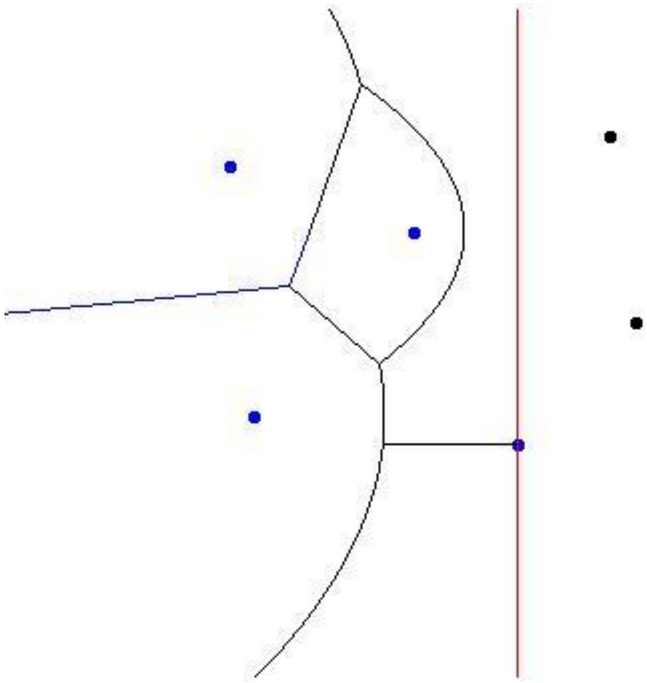
- These parabolic regions form what is called the "beach line"
- The point where two parabolic arcs join on the beach line describe a line segment equidistant from their sites as the sweepline progresses

Fortune’s Algorithm



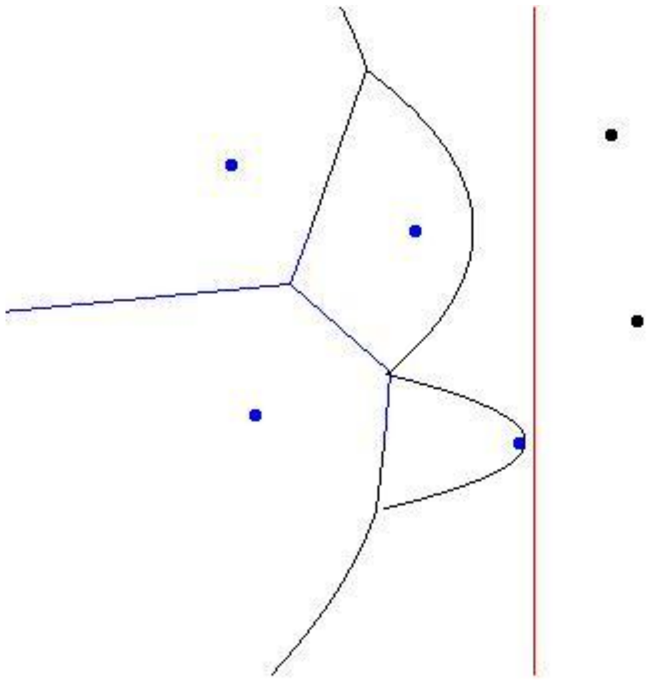
- When three parabolic arcs come together this is called a “circle event” and a Voronoi vertex is formed

Fortune’s Algorithm



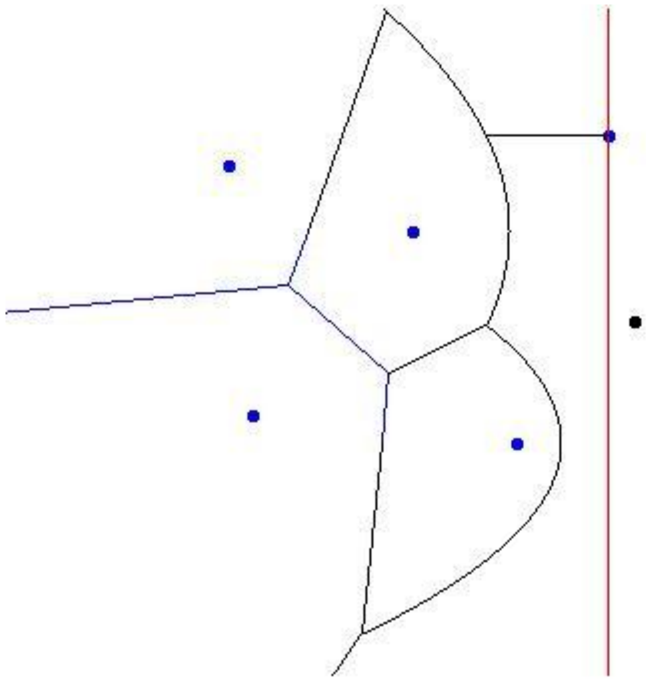
- Sweepline proceeds to process all site events and circle events

Fortune's Algorithm



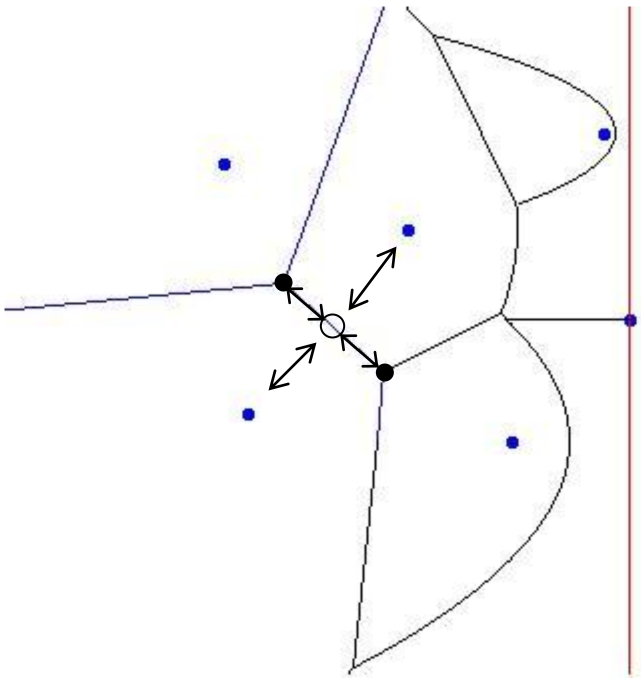
- When vornoi vertices on both sides of a vornoi edge have been processed that edge is output
- The edge is associated with the sites on either side as well as the voronoi vertices on either end

Fortune's Algorithm



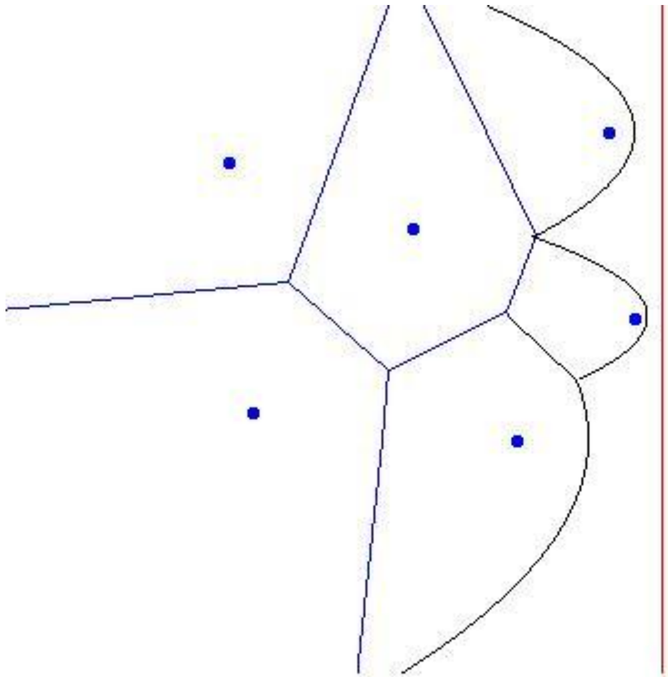
- Insertion of new sites on the beachline is done optimally using a std::map for the beachline data structure

Fortune’s Algorithm



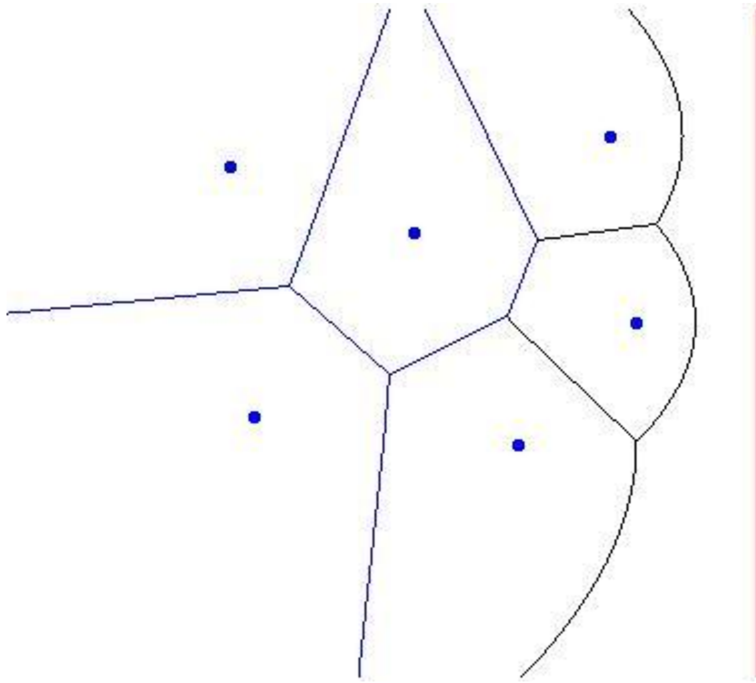
- The output is a planar graph of site, edge and vertex nodes where topological information about the diagram is represented through edges
- Data structure is called a quad-edge

Fortune’s Algorithm



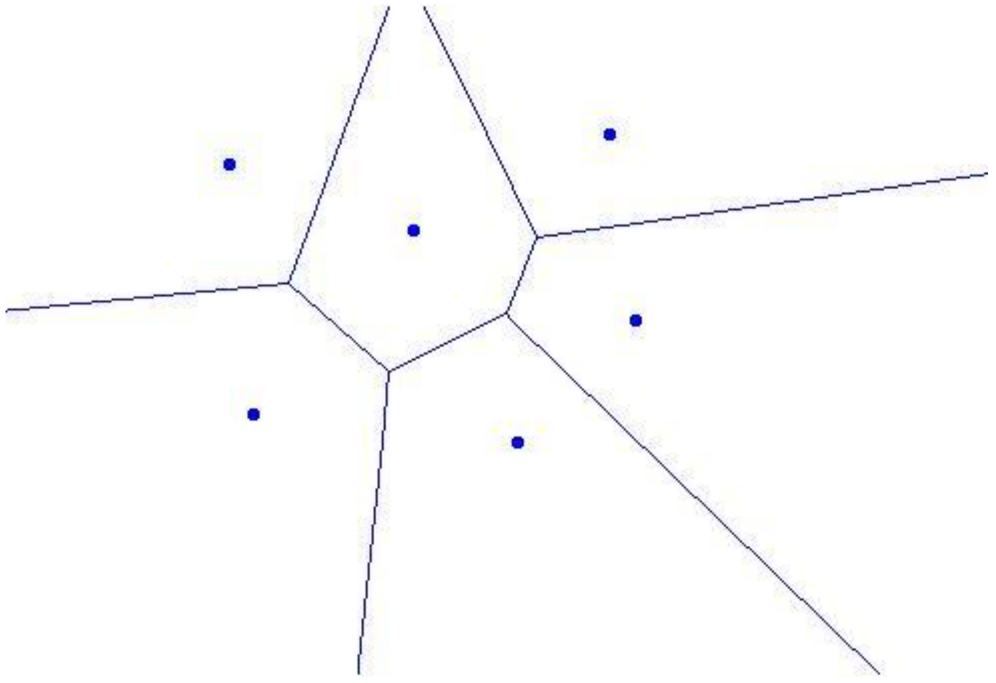
- When a site is no longer associated with active arcs of the beachline its Vornoi cell has been completed

Fortune's Algorithm



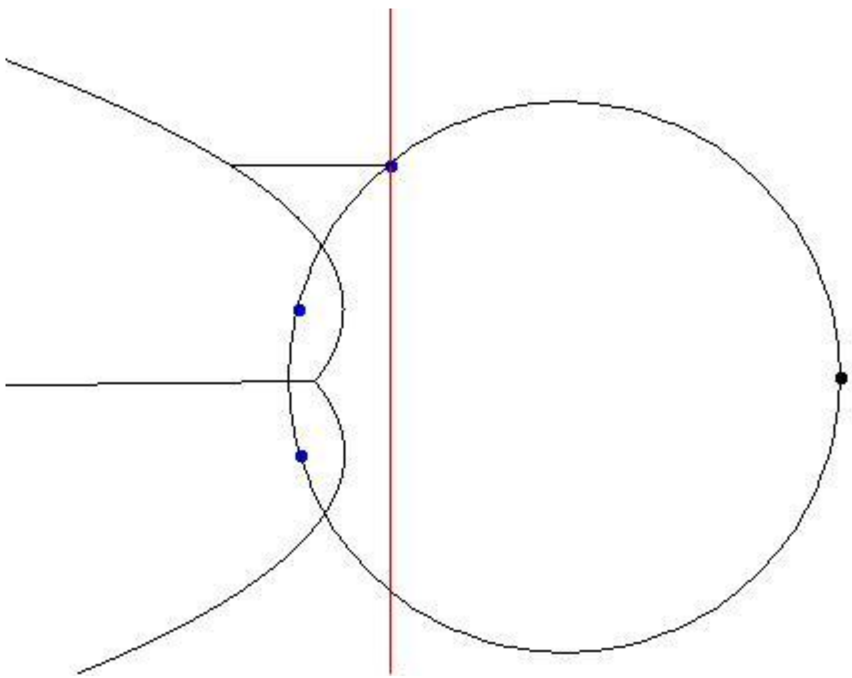
- Once there are no further sites to process the algorithm is complete

Fortune’s Algorithm



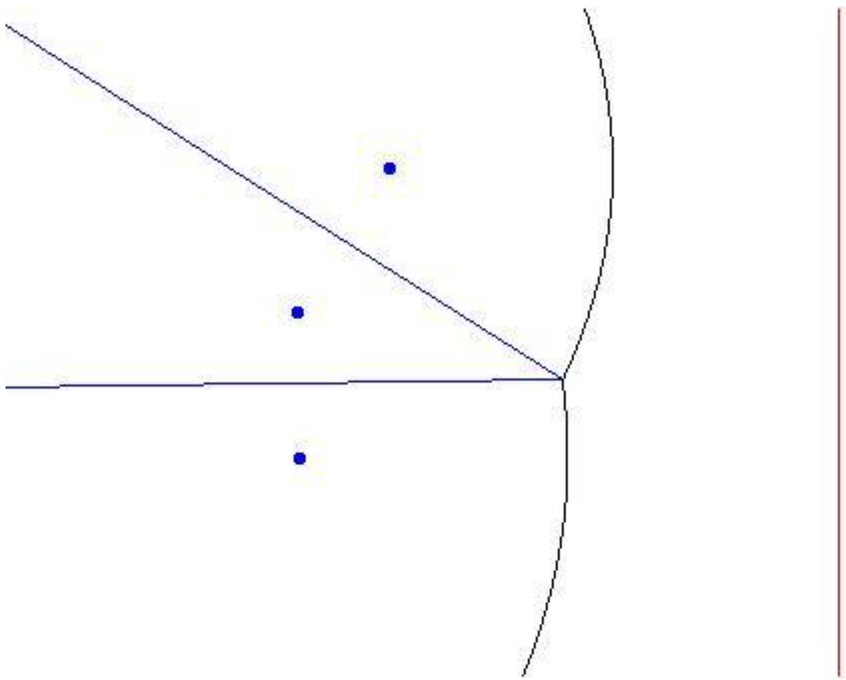
- Voronoi edges on the periphery of the diagram extend to infinity

Fortune's Algorithm



- A circle event is the circle inscribed on three sites
- The event point is the rightmost point of the circle
- When the sweepline reaches the circle event the three input sites associated with it are equidistant from the beachline at a single point, the center of the circle

Fortune's Algorithm



- The beachline at this point is updated by removing one parabolic arc
- The algorithm writes out two Voronoi edges and one Voronoi vertex

Fortune's Algorithm

- The order of circle events is used to maintain a priority queue of circle events yet to be processed
  - It is not a fatal error if the predicate used for the circle event priority queue is not robust but the output topology will be incorrect
- The order of parabolic arcs on the sweepline is maintained by a map
  - It is a fatal error if the predicate used for the sweepline data structure is not robust because the data structure will throw an exception or hang

Fortune's Algorithm for Segments

- Works the same way
- Each end point of the segment is a site as well as a third site for the body of the segment
  - In our case we make two sites for the body, one for each "side" of the line segment
- The portion of the beachline associated with the body of the line segment is a straight line equidistant from the sweep line and the line segment
- Computing beachline events become complex because the three sites involved may be
  - point, point, point
  - point, point, segment
  - point, segment, segment
  - segment, segment, segment
- And solving for the point equidistant from points and segments is a different complex expression involving square roots for each of the four cases
- Computing the relative ordering of circle events in the progress of the sweepline is critically important for correct topology of the output
  - difficult to make numerically robust because you can't compare the computed point if its coordinates contain approximation error

Numerical Robustness

- We assume integer input coordinates and integer output coordinates
- We want to bound the error of the output coordinates of Vornoi vertices to integer rounding error
- We want the topology of the diagram to be correct
- We want the algorithm to be fast, with minimal use of infinite precision arithmetic
- We need a reliable way to deal with irrational values of square roots

Computing Relative Error

- For floating point expressions between arguments of the same sign the relative error is computed as follows

re(A*B) = re(A) + re(B)

re(A/B) = re(A) + re(B)

re(A+B) = max(re(A), re(B))

re(A-B) = (re(A)*A-re(B)*B)/(A-B)

re(sqrt(A)) = re(A) / 2

- Subtraction may result in catastrophic cancellation and arbitrarily large numerical error

How to use Relative Error

- Lets say the sign of a cross product (a*d – b*c) is our predicate
- Let re(a*d) be re1 and re(b*c) be re2
- For a*b and b*c the intervals of error are
  - [a*d*(1-re1), a*d*(1+re1)]
  - [b*c*(1-re2), b*c*(1+re2)]
- If these intervals don't overlap then we know that the sign for a*d – b*c is correct

EPS versus ULP

- Machine epsilon is called EPS and is the maximum error produced by most floating point operations
- ULP (Units in Last Place) is units of precision of the least significant bit of a floating point value
- 0.5 ULP <= 1EPS <= 1 ULP
- So we can safely use units in last place as a proxy for the epsilon interval
- We can tell if two floating point numbers differ only by a value that falls within the ULP error range by unpacking the bits of the IEEE floating point standard by converting to unsigned integer portably using a memcpy
- To compute ULP for floating point operations just add one to the expressions on the previous slide for each operation
- ULP of a floating point expression can be computed off line and used to place a reasonably tight bound on error that can be used to reject the result of a floating point operation as untrustworthy

Relative Error Floating Point Type

- Calculates relative error and floating point result of arithmetic expressions

```
template <typename _fpt>
class robust_fpt {
public:
  typedef _fpt floating_point_type;
  typedef double relative_error_type;
  …
  robust_fpt& operator*=(const robust_fpt &that) {
    this->re_ += that.re_ + ROUNDING_ERROR;
    this->fpv_ *= that.fpv_;
    return *this;
  }
  …
};
```

Boost.Interval

- Boost.Interval can be used to compute tighter bounds on relative error
- Represent a floating point value v as an interval [v, v] and another floating point value u as an interval [u, u]
- Use Boost.Interval to add those intervals with rounding modes: [round_down(u+v), round_up(u+v)]
- round_up and round_down mean manipulating the rounding mode of the floating point unit
- The resulting interval is the relative error interval for u+v
- The round closest behavior that produces machine epsilon error will produce a result that lies in that interval
- Further interval arithmetic accumlates relative error in a similar manner
- Comparison of intervals tells us whether the result of comparing the floating point calculations might be untrustworthy
- This produces a tighter error bound than the ULP method
- The documentation of Boost.Interval doesn’t describe this motivating use case (that I could find)
- We will explore applying Boost.Interval

Fall Back on Infinite Precision

- If floating point may be lying
- Infinite precision is very expensive
- But if good bounds on error are applied the use of infinite precision is very rare
- The result is an algorithm that runs almost as fast as the unreliable floating point algorithm
- This technique is known as lazy exact arithmetic
- It is commonly employed to solve numerical robustness challenges in computational geometry and computing in general

GMP Integration

- Use of GMP will be optional for Voronoi algorithm
  - same to current Polygon library
- Will allow for alternative data types

GMP Performance Problem

- GMP has a C++ wrapper that we use
- GMP runtime can be dominated by allocation/de-allocation time
- You want to recycle your GMP variables
- Allocation of temporaries generated by complex expressions kills your performance
- Writing one arithmetic operation per line isn’t a satisfying solution

Current GMP Performance Solution

- This works to fix the problem
- I hate it because it has static members and isn’t thread safe
- How can we solve this problem better?

```
template <typename mpt, int N>
class mpt_wrapper {
public:
  …
  mpt_wrapper& operator+(const mpt_wrapper& that) const {
      temp_[cur_].m_ = this->m_ + that.m_;
            return temp_[next_cur()];
        }
  …
private:
  static int next_cur() {
    int ret_val = cur_++;
    if (cur_ == N)
      cur_ = 0;
      return ret_val;
    }

    mpt m_;
    static int cur_;
    static mpt_wrapper temp_[N];
};
```

Idea For Better Solution

- I was thinking along these lines

```
template <typename mpt>
class mpt_tmp {
public:
  …
  mpt_tmp& operator+(const mpt_wrapper& that) const {
    return (*this) += that;
  }
  …
private:
  mutable mpt m_;
};

template <typename mpt>
class mpt_wrapper {
public:
  …
  mpt_tmp& operator+(const mpt_wrapper& that) const {
    tmp = m_ + that.m_; return tmp;
  }
  …
private:
  mpt m_;
  mutable mpt_tmp<mpt> tmp;
};
```

SQRT Woes

- If we want to compute a – sqrt(b) and a is greater than zero and almost equal to sqrt(b) then the error will be huge
- If we have several such sub expressions we probably can't trust even the sign bit of the final result
- We need to avoid catastrophic cancellation error

Refactoring By Conjugates

- If we want to have robust floating point evaluation of the expression:

  A*sqrt(a) + B*sqrt(b)
- We need to handle two cases seperately
- A*B >= 0

  A*sqrt(a) + B*sqrt(b)
- A*B < 0

  ((A*A*a)-(B*B*b))/(A*sqrt(a) – B*sqrt(b))
- We multiply numerator and denominator by

  A*sqrt(a) – B*sqrt(b)
- to prevent cancelation

Refactoring By Conjugates

- We can cover all cases of positive and negative factors of square roots up to four square root terms

  A*sqrt(a) + B*sqrt(b) + C*sqrt(c) + D*sqrt(d)

- without cancelation error by using enough conditionals and alternative equivalent expressions
- But five or more cannot be handled
- We are “lucky” that we only have up to four square roots in voronoi diagram of line segments
- We would be luckier not to have square roots at all

Lazy Exact Computations for Voronoi of Points case

- Sweep-line predicate
  - Lazy-exact
  - Floating point approximation with relative error calculation
  - Emulates 65 bit integer arithmetic for exact result
- Circle-event
  - Lazy-exact to within small bounded relative error
  - Uses refactoring by conjugates floating point approximation with relative error calculation
  - Uses wrapped gmp numerical data type for exact result with recycled temporaries and minimal relative error introduced by sqrt

Quick Glance at Code

```
// Find parameters of the inscribed circle that is tangent to three
// point sites.
template <typename T>
static bool create_circle_event_ppp(const site_event<T> &site1,
                                    const site_event<T> &site2,
                                    const site_event<T> &site3,
                                    circle_event<T> &c_event) {
    double dif_x1 = site1.x() - site2.x();
    double dif_x2 = site2.x() - site3.x();
    double dif_y1 = site1.y() - site2.y();
    double dif_y2 = site2.y() - site3.y();
    double orientation = robust_cross_product(dif_x1, dif_y1, dif_x2, dif_y2);
    if (orientation_test(orientation) != RIGHT_ORIENTATION)
        return false;
    robust_fpt<T> inv_orientation(0.5 / orientation, 3.0);
    double sum_x1 = site1.x() + site2.x();
    double sum_x2 = site2.x() + site3.x();
    double sum_y1 = site1.y() + site2.y();
    double sum_y2 = site2.y() + site3.y();
    double dif_x3 = site1.x() - site3.x();
    double dif_y3 = site1.y() - site3.y();
    epsilon_robust_comparator< robust_fpt<T> > c_x, c_y;
    c_x += robust_fpt<T>(dif_x1 * sum_x1 * dif_y2, 2.0);
    c_x += robust_fpt<T>(dif_y1 * sum_y1 * dif_y2, 2.0);
    c_x -= robust_fpt<T>(dif_x2 * sum_x2 * dif_y1, 2.0);
    c_x -= robust_fpt<T>(dif_y2 * sum_y2 * dif_y1, 2.0);
    c_y += robust_fpt<T>(dif_x2 * sum_x2 * dif_x1, 2.0);
    c_y += robust_fpt<T>(dif_y2 * sum_y2 * dif_x1, 2.0);
    c_y -= robust_fpt<T>(dif_x1 * sum_x1 * dif_x2, 2.0);
    c_y -= robust_fpt<T>(dif_y1 * sum_y1 * dif_x2, 2.0);
    epsilon_robust_comparator< robust_fpt<T> > lower_x(c_x);
    lower_x -= robust_fpt<T>(std::sqrt(sqr_distance(dif_x1, dif_y1) *
                                       sqr_distance(dif_x2, dif_y2) *
                                       sqr_distance(dif_x3, dif_y3)), 5.0);
    c_event = circle_event<double>(c_x.dif().fpv() * inv_orientation.fpv(),
                                   c_y.dif().fpv() * inv_orientation.fpv(),
                                   lower_x.dif().fpv() * inv_orientation.fpv());
    bool recompute_c_x = c_x.dif().ulp() >= 128;
    bool recompute_c_y = c_y.dif().ulp() >= 128;
    bool recompute_lower_x = lower_x.dif().ulp() >= 128;
    if (recompute_c_x || recompute_c_y || recompute_lower_x) {
        return create_circle_event_ppp_gmpxx(
            site1, site2, site3, c_event, recompute_c_x, recompute_c_y, recompute_lower_x);
    }
    return true;
}
```

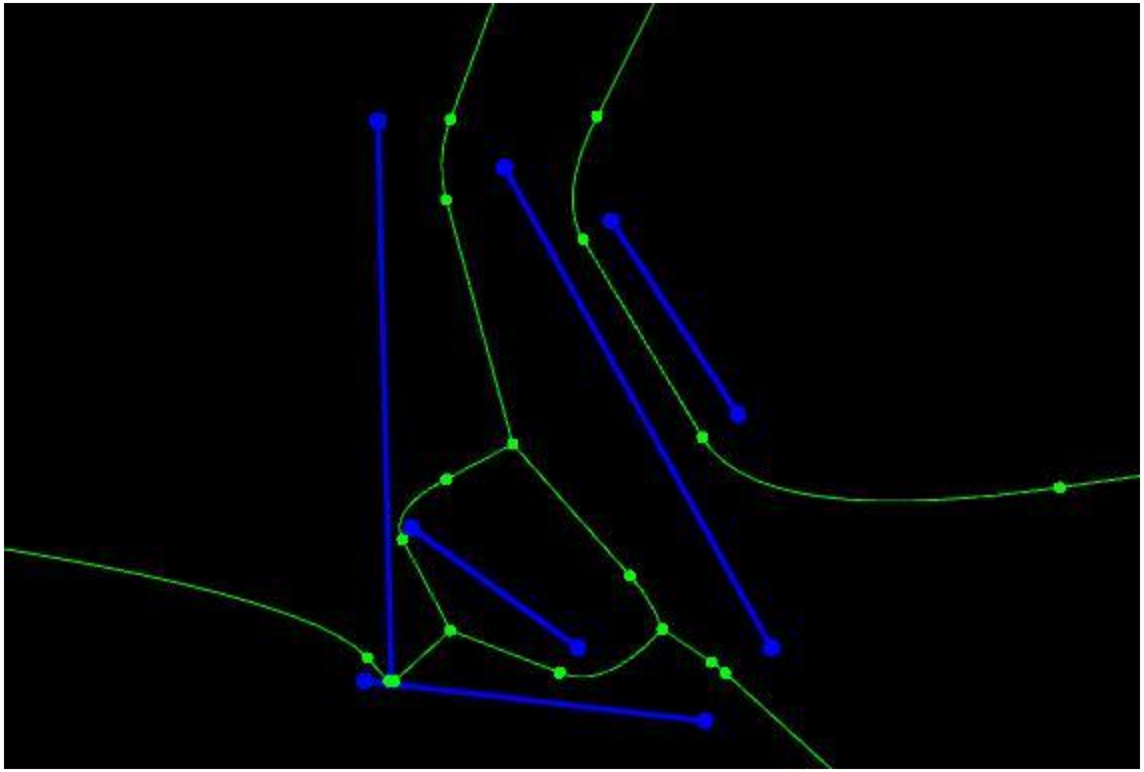
Error Bounded Output

- We are computing the values for the circle event and then comparing these values in the circle event queue
- There is no way to compute the exact values because they contain square root
- Our output is correct to within bounded relative error
- 128 ULP translates into 7 bits in the bottom of double or 2^-45 at integer scale
- I’d rather use long double than double and intend to switch
- If we used long double our relative error would be less than machine epsilon for double
- It may be provable that with a sufficient error bound no miss-ordering of circle events is possible but the numerical analysis is challenging
- Such proof may be possible for floating point input coordinates also

Segment Predicates

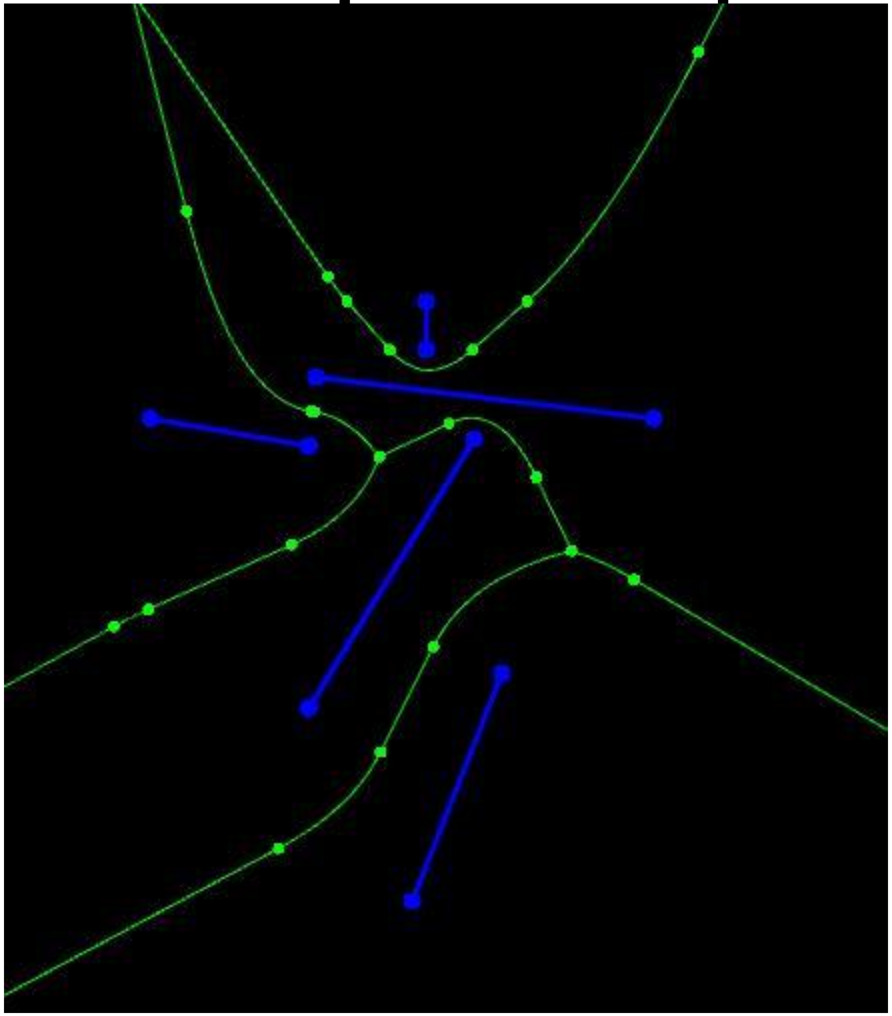
- The comparison of beach line events with line segment sites involved is similar to the points case
- The floating point expressions for each are complicated
- Bound error to many orders of magnitude smaller than integer grid
- Currently use infinite precision arithmetic to reduce error bound to minimal error introduced by square roots operations only
- We use refactoring by conjugates to minimize the error introduced by square roots
- Each needs to be written in floating point and have ULP calculated off line to ensure it is small enough
- May contain up to four square roots in a single expression
- Floating point approximate with relative error computation also implemented but currently not used

Example Output



- Example of diagram of segments with near touch

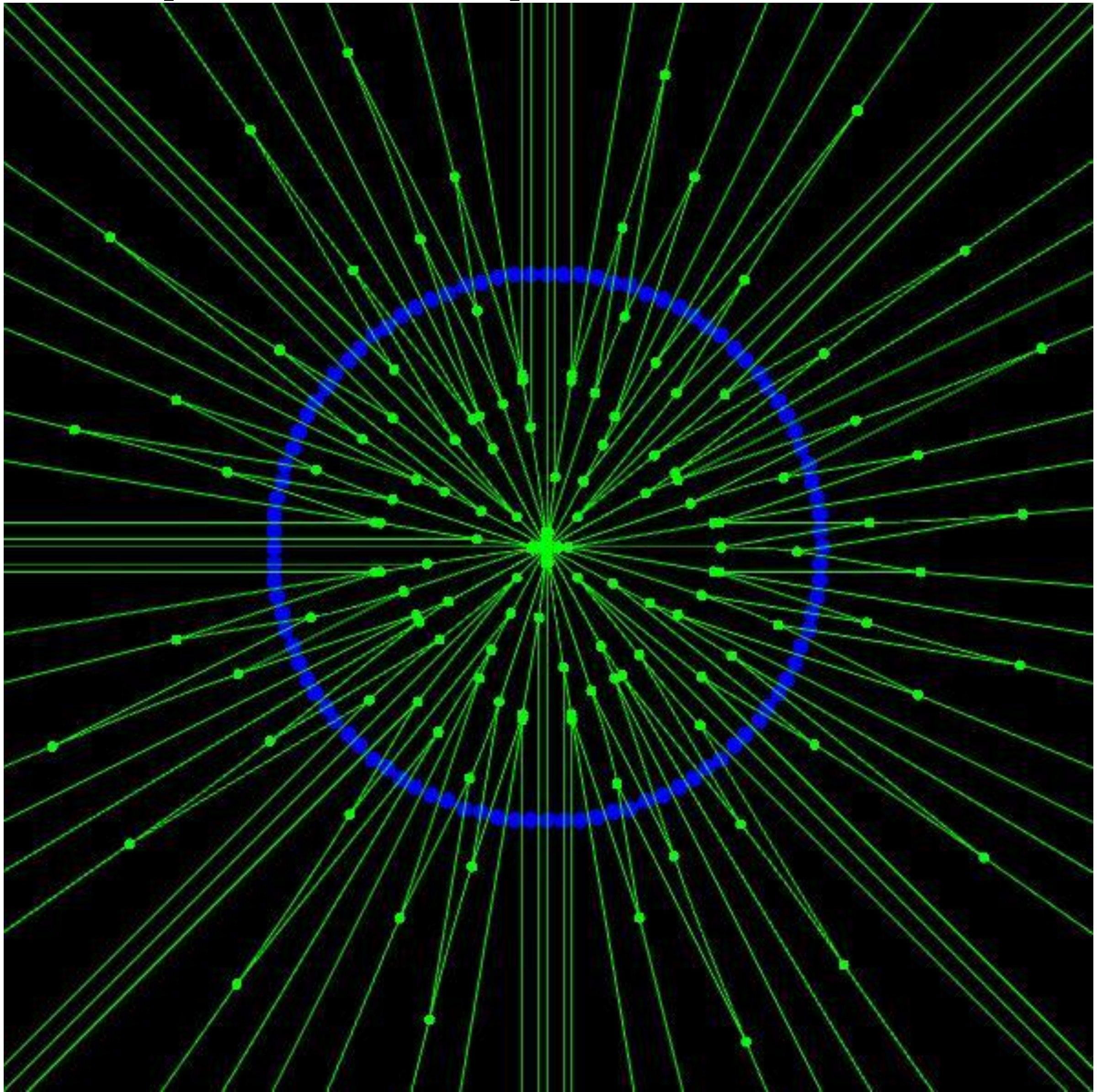
Example Output



- Example of diagram of segments

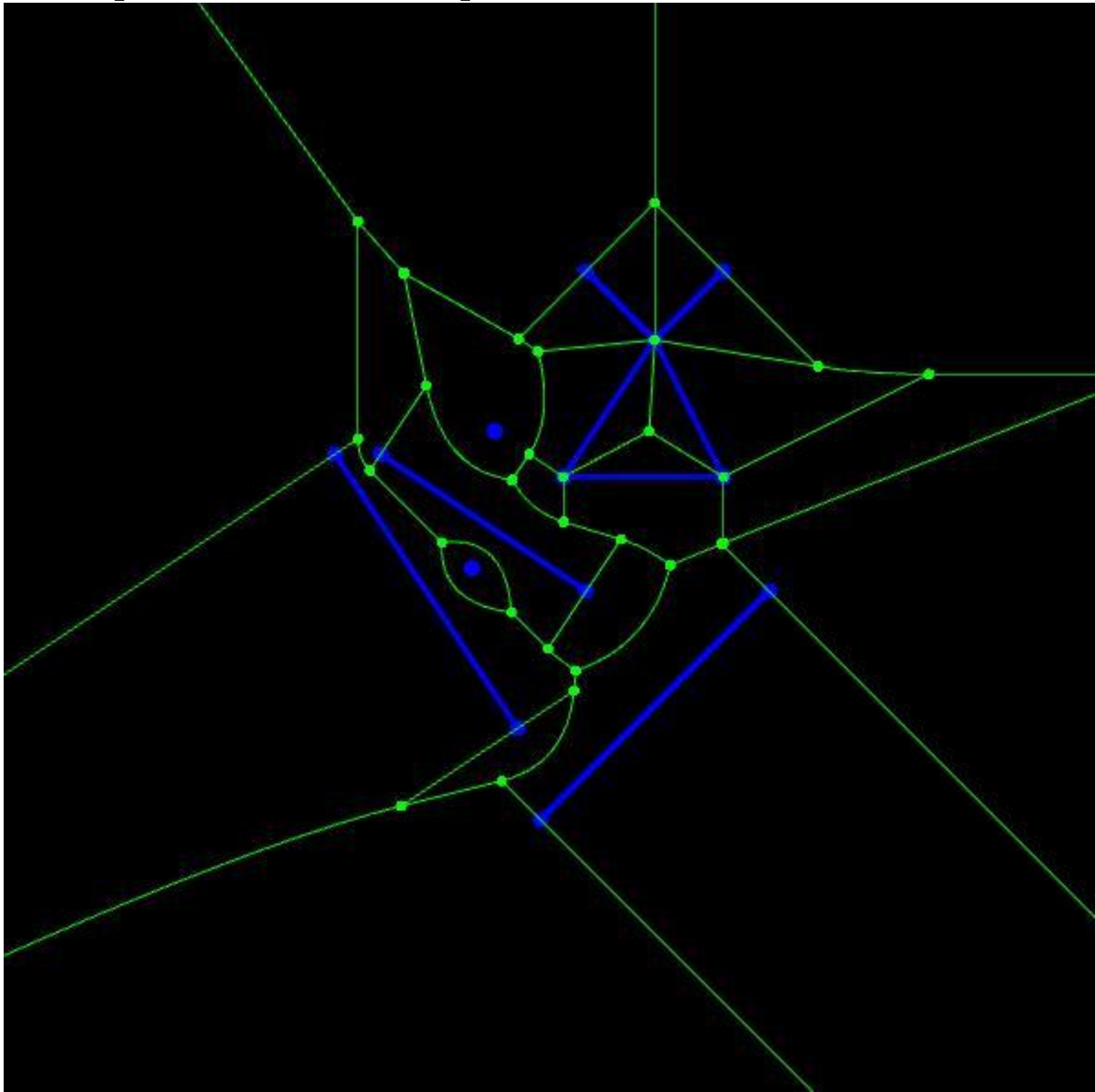
Example Output

- Example of diagram of points
- Co-circular points are the worst case input for Voronoi diagram



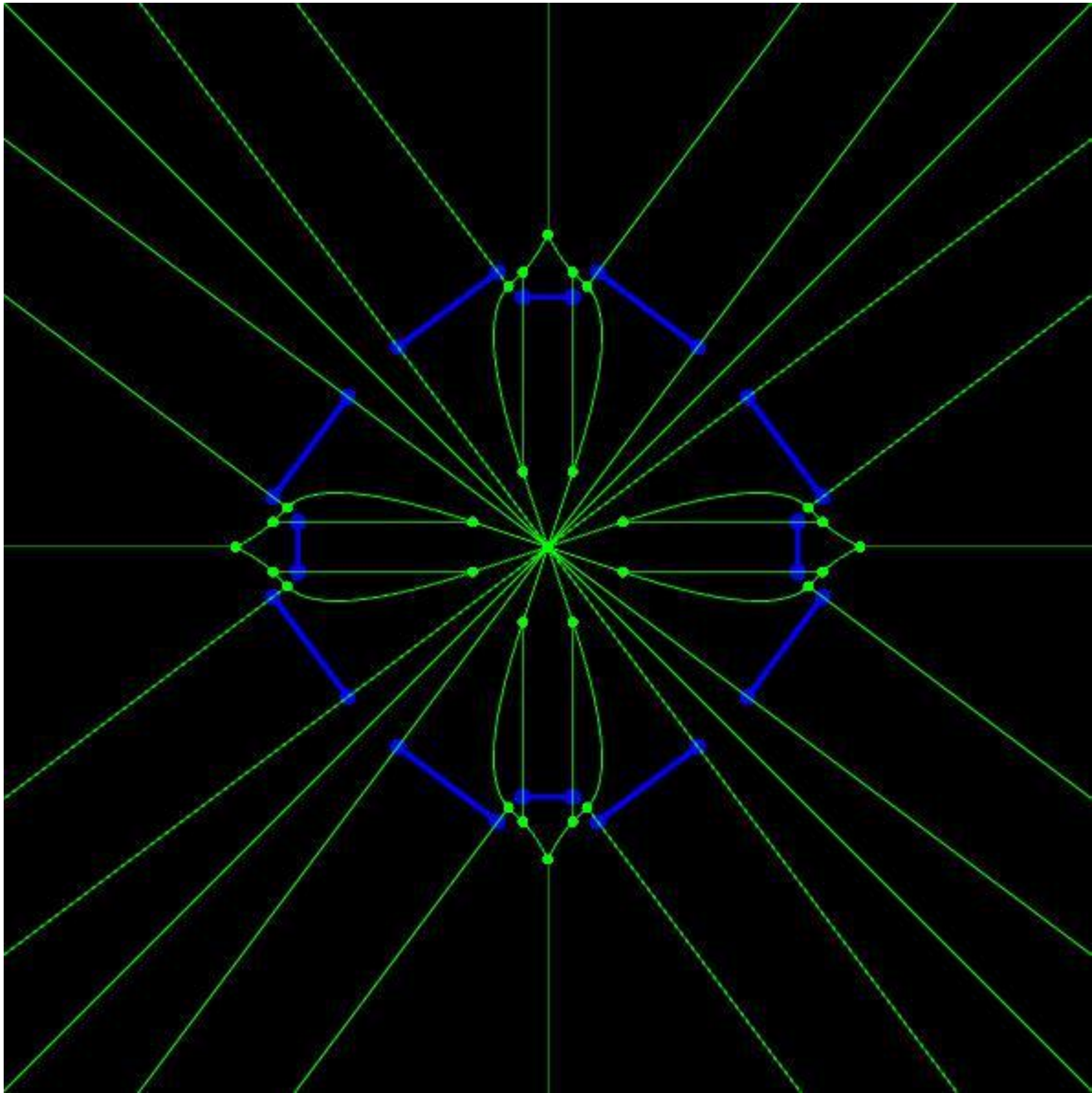
Example Output

- Example diagram of polygon, points and segments



Example Output

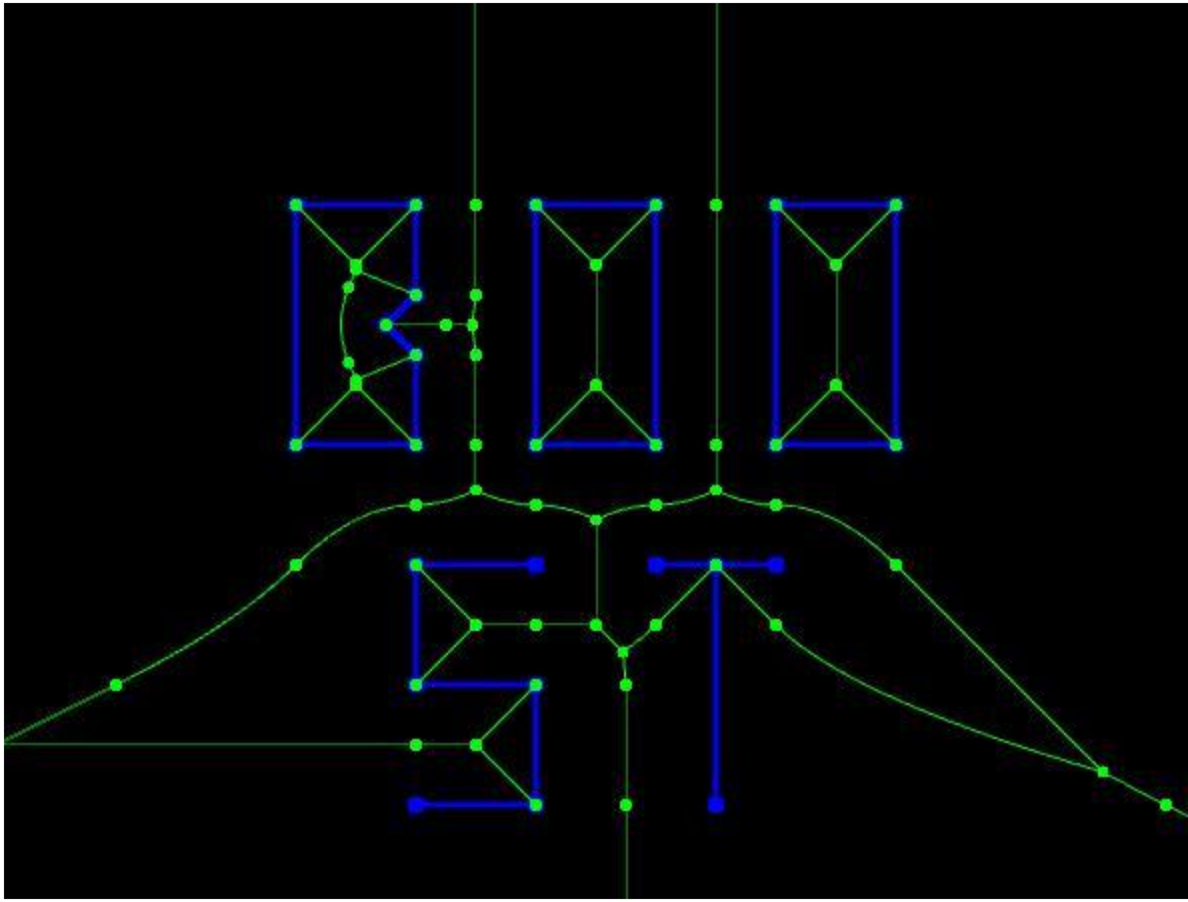
- Co-circular line segments
- Note the high order voronoi vertex in the center



Example Output

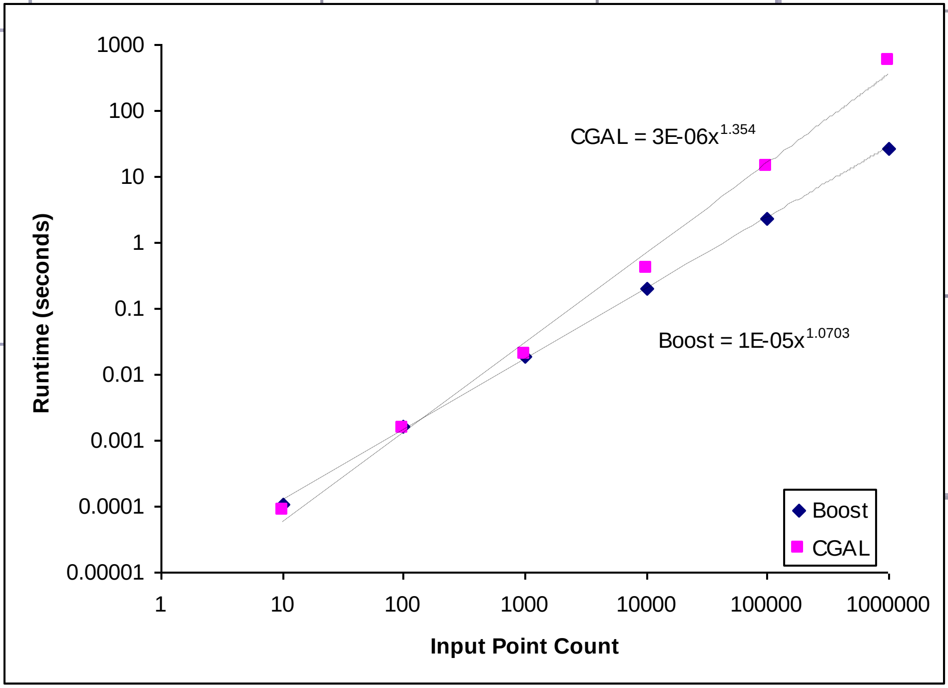
- Diagram of letters:

B, O, O, S, T



Benchmark

- Voronoi diagram of points compared to CGAL
- 22X faster than CGAL at 1,000,000 points



New Boost.Polygon APIs

- Voronoi diagram of Points
  - Populates a container of polygon concept with voronoi cells given a iterator range over point concept

```
void voronoi_diagram(container_type& cells,
  iterator_type begin_points, iterator_type end_points);
```

- Delaunay diagram of Points
  - Populates a container of polygon concept with Delaunay triangles given an iterator range over point concept

```
void delaunay_triangles(container_type& triangles,
  iterator_type begin_points, iterator_type end_points);
```

- Voronoi diagram of Polygon Set
  - Populates a container of polygon concept with voronoi cells given a model of polygon set concept or one of its refinements
  - Uses the threshold provided to segment parabolic arcs into line segments such that the line segments lie no further than threshold from the true curve

```
void voronoi_diagram(container_type& cells, const
  polygon_set_type& polygons, const
  coordinate_distance_type& threshold);
```

GSOC 2011

- Input line segments must not be intersecting is a pre-condition of Voronoi diagram of line segments
- It is also a post condition of the line segment intersection algorithm used in the first stage of polygon clipping implemented already in Boost.Polygon
- Vornoi diagram of line segments needs edge concepts to implement generic interfaces

GSOC 2011

- Implement new generic C++ concepts for
  - Line segment
  - Directed line segment
  - Set of line segments
  - Set of directed line segments
- Directed line segment is a refinement of line segment
- Line segment is a refinement of set of line segments
- Set of directed line segments is a refinement of set of line segments
- Directed line segment is a refinement of set of directed line segments
- Polygon set is a refinement of set of directed line segments
- Expose a concept based user interface to line segment intersection
- Expose a concept based user interface to robust predicates for
  - line segment slope comparison
  - point on above or below line segment
  - whether two line segments intersect
- The user will be able to pass rectangle concept into a generic interface expecting set of line segments concept because rectangle is a refinement of polygon, is a refinement of polygon set, is a a refinement of set of directed line segments is a refinement of set of line segments

GSOC 2011

- New segment concepts in Boost.Polygon will enable new interfaces based on Vornoi Diagram of line segments
- Voronoi diagram of Points
  - Populates a container of line segment concept with voronoi edges given a iterator range over point concept

```
void voronoi_diagram(container_type& edges,
   iterator_type begin_points, iterator_type end_points);
```

- Voronoi diagram of Segments
  - Populates a container of line segment concept with voronoi edges given a iterator range over line segment concept

```
void voronoi_diagram(container_type& edges, iterator_type begin_segments,
   iterator_type end_segments , const coordinate_distance_type& threshold);
```

- Voronoi diagram of Segments
  - Populates a container of line segment concept with voronoi edges given a model of set of line segments concept

```
void voronoi_diagram(container_type& edges, const line_segments_type& sites,
   const coordinate_distance_type& threshold);
```

- Medial Axis of Polygon Set
  - Populates a container of line segment concept with medial axis edges given a model of polygon set concept

```
void medial_axis(container_type& edges, const polygon_set_type& polygons,
   const coordinate_distance_type& threshold);
```

Q&A