ZEN AND THE ART OF MULTI PRECISION ALGORITHMS DESIGN

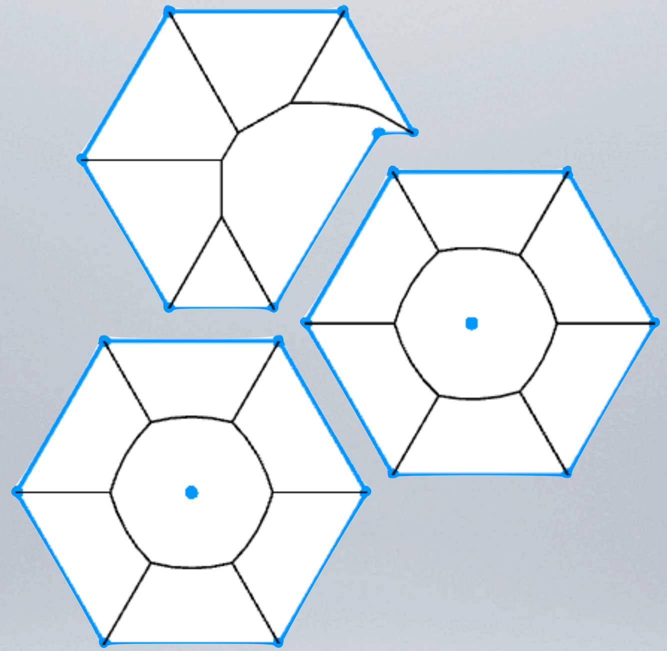
Andrii Sydorchuk (Google Corp.),
Simonson Lucanus (Intel Corp.)

TARGETS

- Defining notion of the algorithm design and implementation quality.
- Lay the foundation for the robust, language independent, multi precision computations.
- Covering techniques to enhance implementation performance and reliability.
- Covering techniques to speed up development process.
- Familiarizing community with the **Boost.Polygon Voronoi** library.

AGENDA

- Voronoi
- Algorithm Design
    - Robustness
    - Efficiency
    - Extensibility
    - Usability
- Summary
- Q&A



VORONOI

VORONOI DIAGRAM

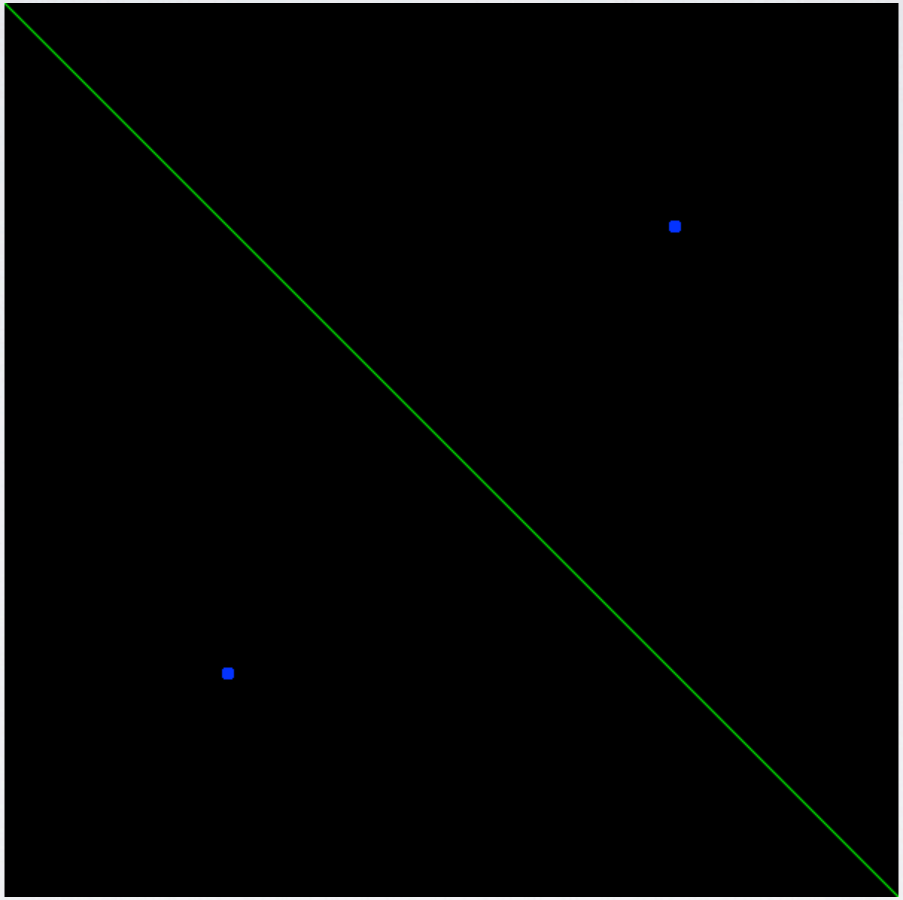
Computational geometry concept named in honor of the Ukrainian mathematician Georgi Voronoi.

Represents partition of the space onto regions, with bounds determined by the distances to a specified family of objects.

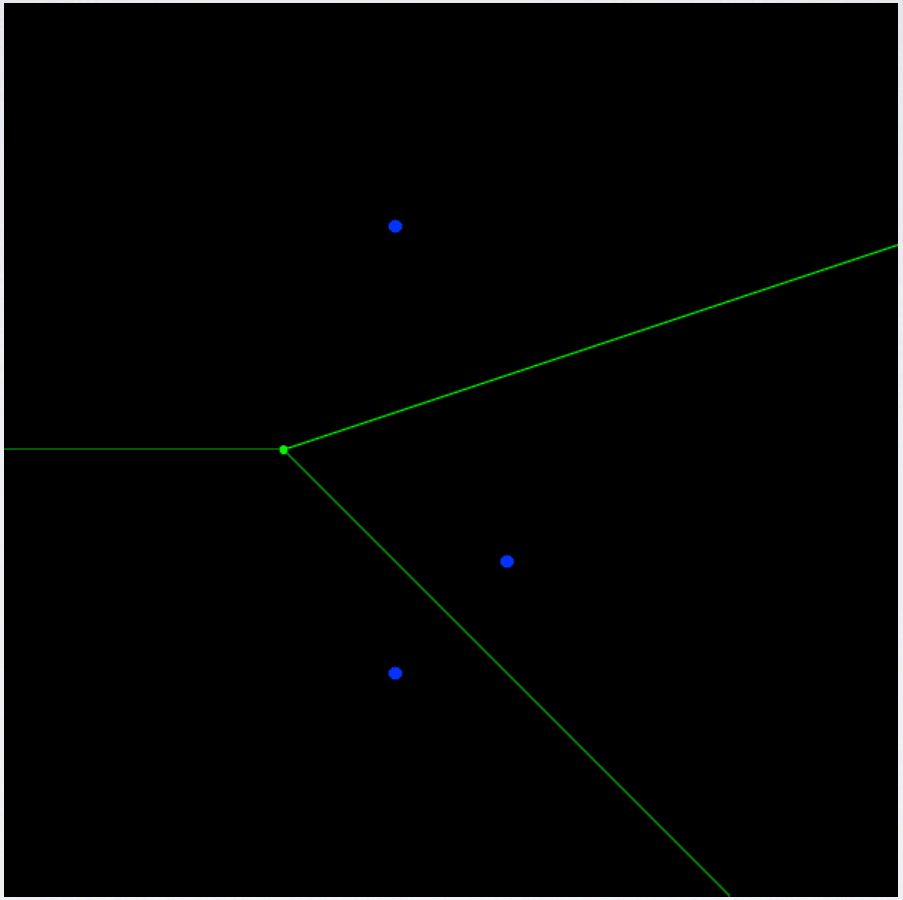
NARROWING DEFINITION

Space	2D
Distance	Euclidean
Input objects	Points & segments

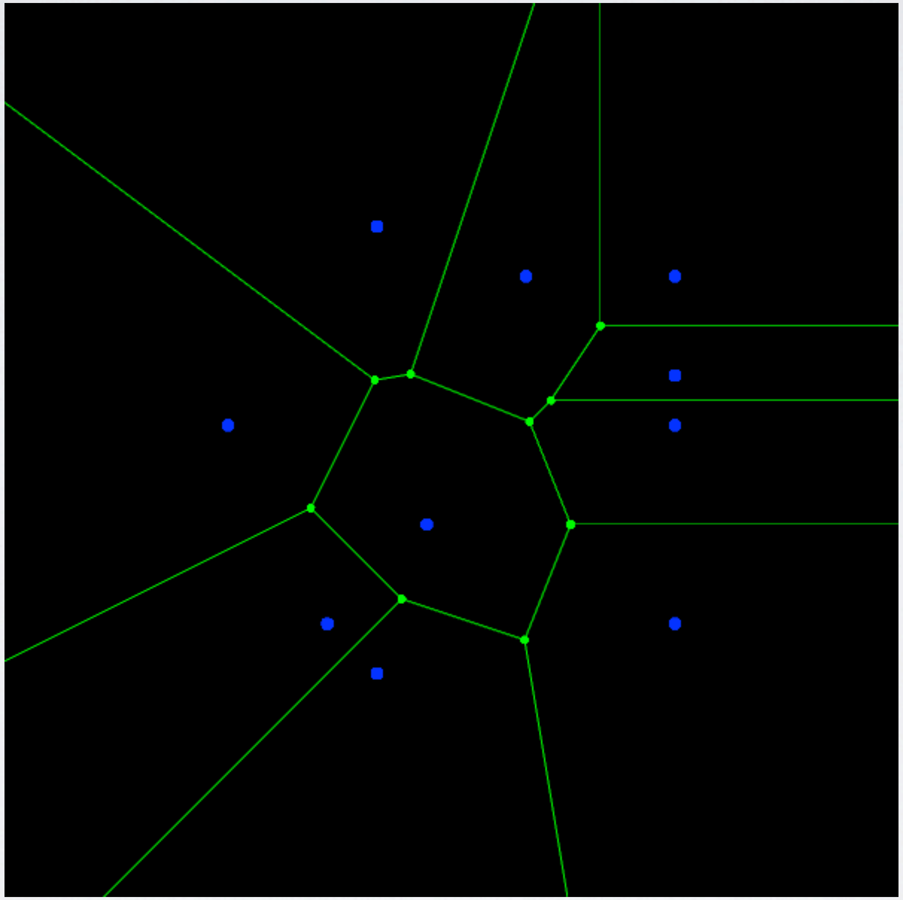
EXAMPLE (2 POINTS)



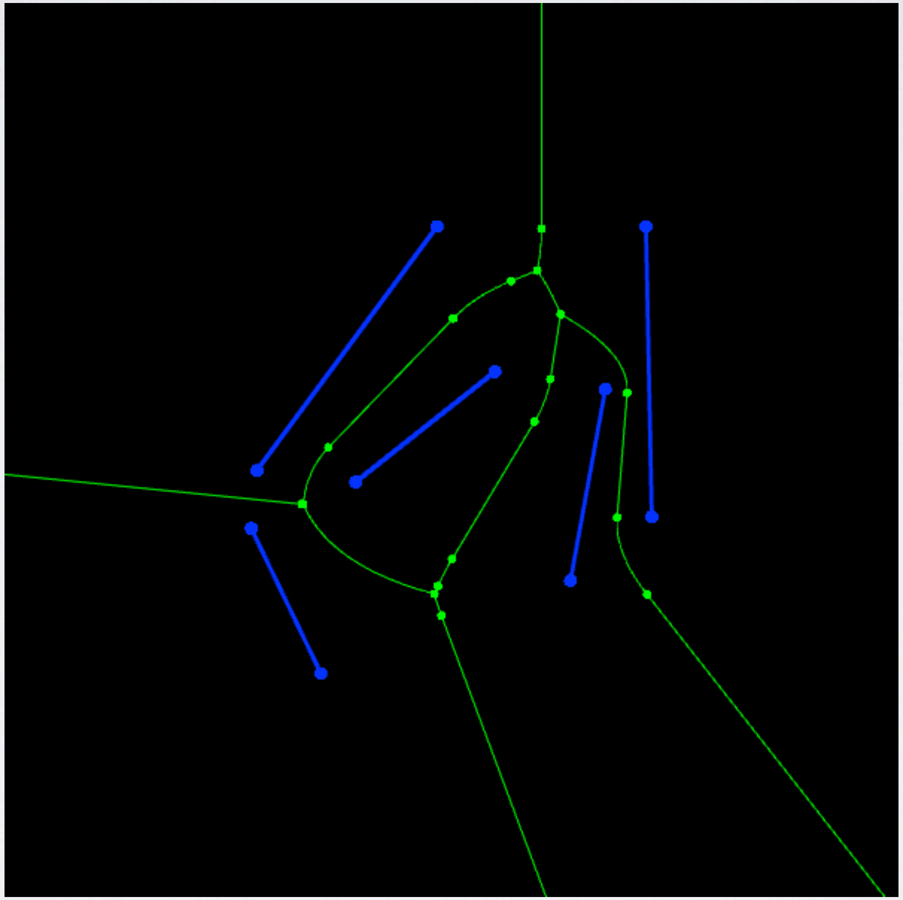
EXAMPLE (3 POINTS)



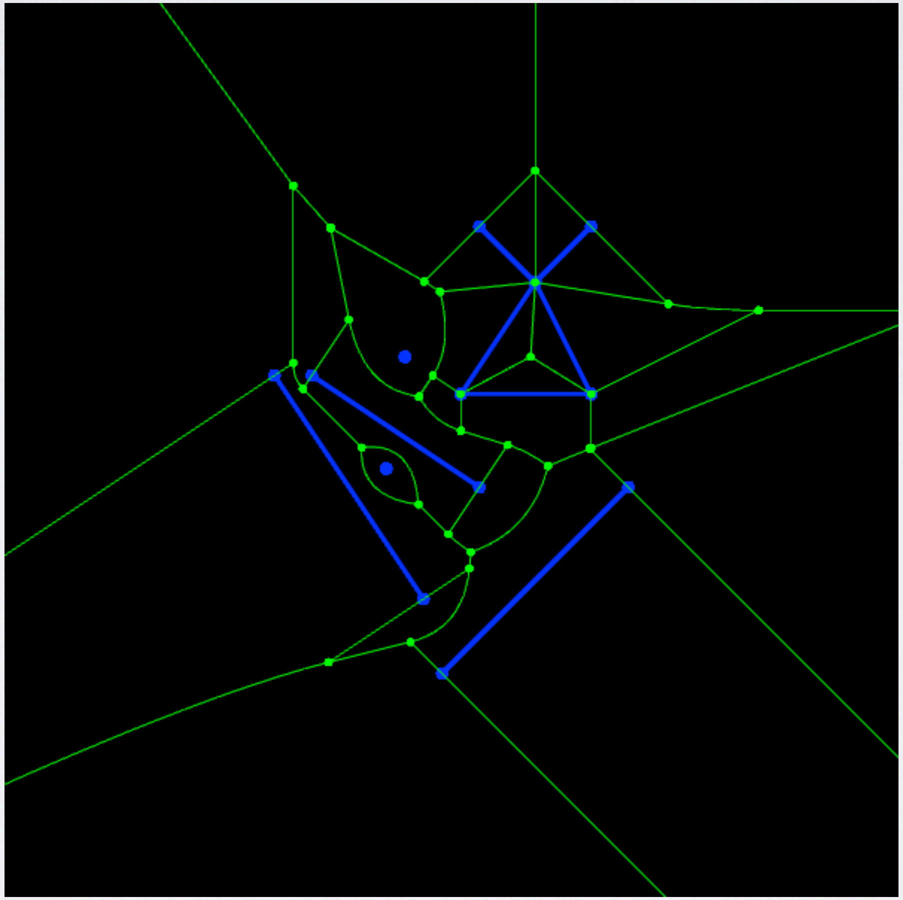
EXAMPLE (10 POINTS)



EXAMPLE (5 SEGMENTS)



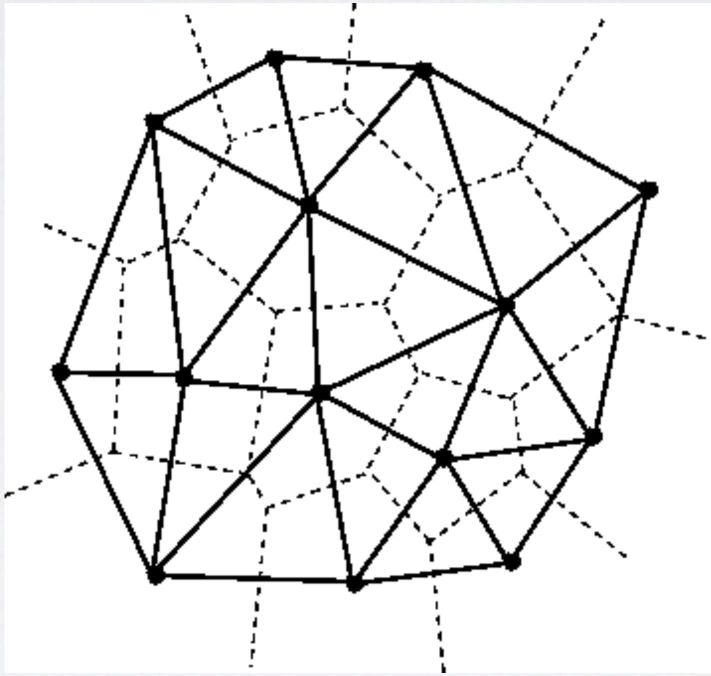
EXAMPLE (MIXED)



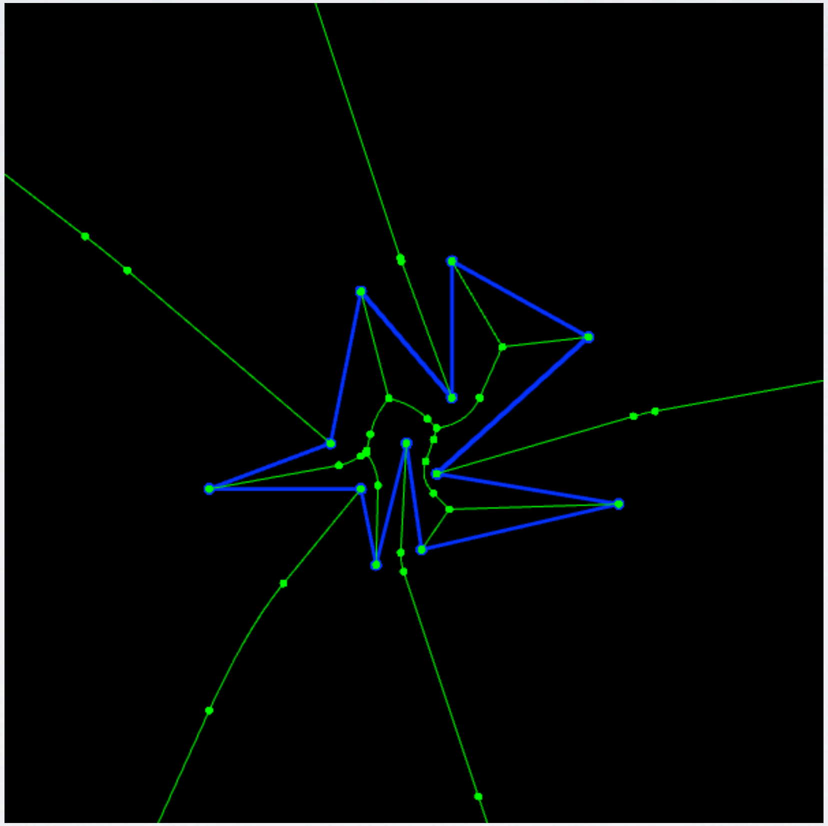
APPLICATION AREAS

Application Fields	Anthropology, Archeology, Astronomy, Biology, Ecology, Forestry, Cartography, Crystallography, Chemistry, Finite Element Analysis, Geography, Geology, Geometric Modeling, Marketing, Mathematics, Metallurgy, Meteorology, Pattern Recognition, Physiology, Robotics, Statistics and Data Analysis, Zoology
Computational Geometry	Delaunay Triangulation Medial Axis Knuth's Post Office Problem Closest Pair All Nearest Neighbors Euclidean Minimum Spanning Tree Largest Empty Circle Enumerating Inter Point Distances

VORONOI DIAGRAM & DELAUNAY TRIANGULATION

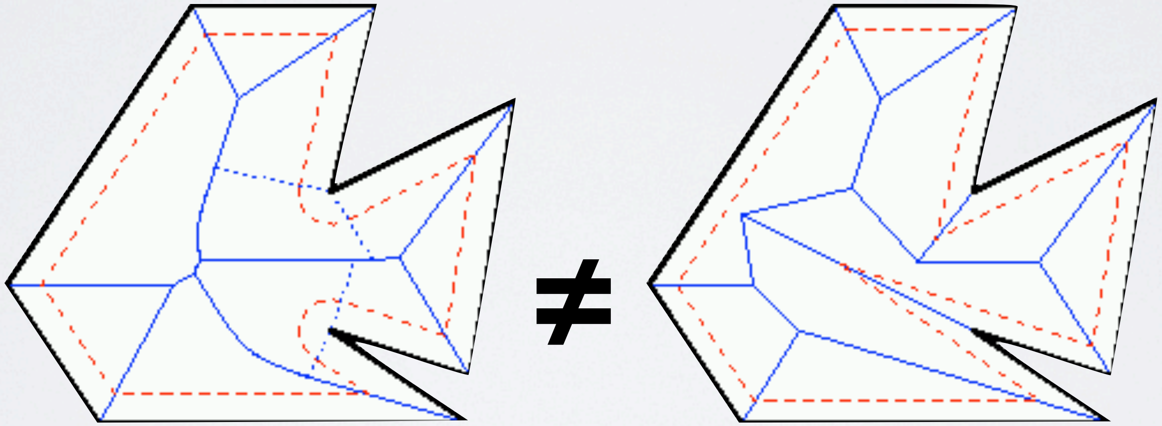


VORONOI DIAGRAM & MEDIAL AXIS



MEDIAL AXIS VS STRAIGHT SKELETON

Medial Axis	Straight Skeleton



MOTIVATION

- Application areas vary from Archeology to Zoology.
- The number of known robust implementations even for Voronoi of points could be counted on fingers.
- Existing libraries have quite complex user interface and usually include dependencies on such libraries as GMPXX, MPFR.
- No public library that implements Voronoi of segments and just three known commercial: CGAL (1500$), LEDA (1600$), Vroni (price not known).

CHALLENGES

- Just a few noteworthy articles relevant to the topic.
- Nontrivial algorithm with many corner cases.
- Minor info on the Voronoi of points implementation.
- No info on the Voronoi of segments implementation.
- Very careful handling of the numeric computations required.

ROBUSTNESS

OWNERSHIP VALIDATION

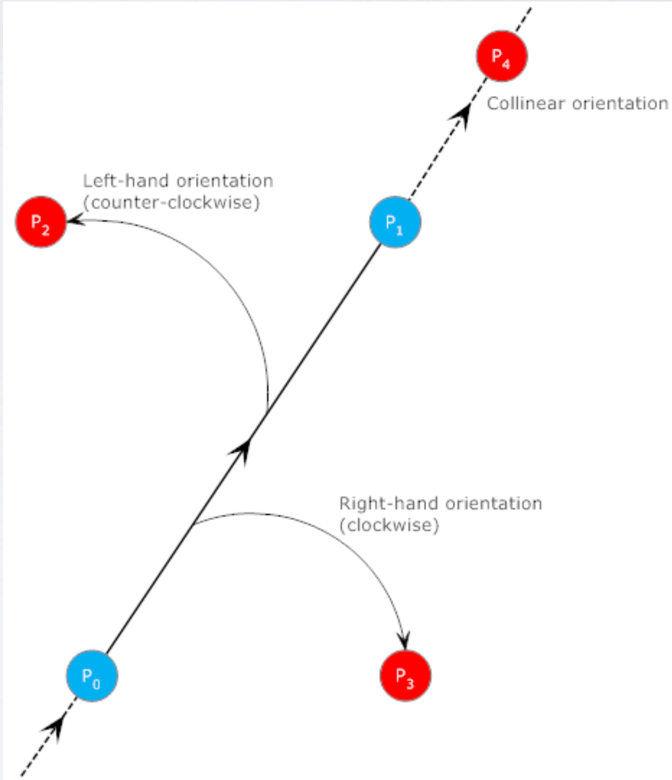
```
#include <iostream>
#include <numeric>
#include <vector>

void validate_ownership(const std::vector<double>& ownership) {
    double total = std::accumulate(ownership.begin(), ownership.end(), 0.0);
    if (total <= 100.0)
      std::cout << "Ownership validation passed!" << std::endl;
    else
      std::cout << "Invalid ownership!" << std::endl;
}

int main() {
    std::vector<double> ow1 = {0.2, 49.2, 50.6};
    std::vector<double> ow2 = {49.2, 0.2, 50.6};
    std::vector<double> ow3 = {49.2, 50.6, 0.2};
    validate_ownership(ow1);
    validate_ownership(ow2);
    validate_ownership(ow3);
   return 0;
}
```

OWNERSHIP VALIDATION

```cpp
#include <iostream>
#include <numeric>
#include <vector>

void validate_ownership(const std::vector<double>& ownership) {
   double total = std::accumulate(ownership.begin(), ownership.end(), 0.0);
   if (total <= 100.0)
     std::cout << "Ownership validation passed!" << std::endl;
   else
     std::cout << "Invalid ownership!" << std::endl;
}

int main() {
   std::vector<double> ow1 = {0.2, 49.2, 50.6};
   std::vector<double> ow2 = {49.2, 0.2, 50.6};
   std::vector<double> ow3 = {49.2, 50.6, 0.2};
   validate_ownership(ow1);  // "Ownership validation passed!"
   validate_ownership(ow2);  // "Ownership validation passed!"
   validate_ownership(ow3);  // "Invalid ownership!"
  return 0;
}
```

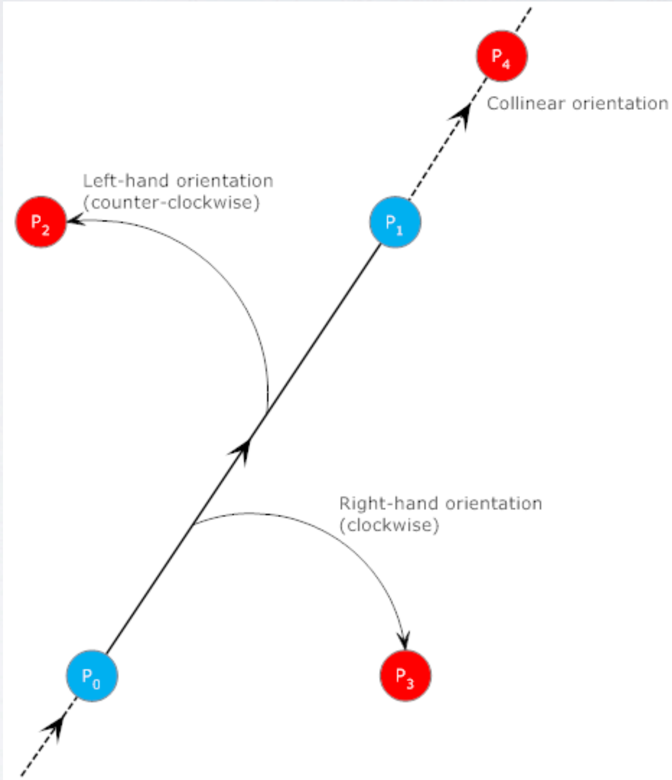
ORIENTATION TEST

```
#include <iostream>

int orientation(double dx1, double dy1,
                double dx2, double dy2) {
   double lhs = dx1 * dy2;
   double rhs = dx2 * dy1;
   if (lhs == rhs)
       return 0;
   return (lhs < rhs) ? -1 : 1;
}

int main() {
   double v = (1 << 30);
   int or = orientation(v, v+1, v-1, v);
   std::cout << or << std::endl;
   return 0;
}
```



ORIENTATION TEST

```
#include <iostream>

int orientation(double dx1, double dy1,
                double dx2, double dy2) {
   double lhs = dx1 * dy2; // Equal 2^60
   double rhs = dx2 * dy1; // Equal 2^60
   if (lhs == rhs)
       return 0;
   return (lhs < rhs) ? -1 : 1;
}

int main() {
   double v = (1 << 30);
   int or = orientation(v, v+1, v-1, v);
   std::cout << or << std::endl; // Prints 0!
   return 0;
}
```



ROBUST ORIENTATION TEST

```
#include <iostream>

int robust_orientation(double dx1_, double dy1_, double dx2_, double dy2_) {
  MPFPT dx1(dx1_), dy1(dy1_), dx2(dx2_), dy2(dy2_);
  MPFPT lhs = dx1 * dy2; // Equal 2^60
  MPFPT rhs = dx2 * dy1; // Equal 2^60 - 1
  if (lhs == rhs)
    return 0;
  return (lhs < rhs) ? -1 : 1;
}

int main() {
  double v = (1 << 30);
  int or = robust_orientation(v, v+1, v-1, v);
  std::cout << or << std::endl; // Prints 1!
  return 0;
}
```

NUMERIC TYPES

INTEGER INTEGRAL TYPES	FLOATING-POINT TYPES

Think of an integer as of a single value:
`int64 a = 1E18;`
`int64 b = 1;`
`int c = a + b; // 1E18 + 1`
`// c = expected result`
	Think of a floating-point type as of a range of values:
	`double a = 1E18;`
	`double b = 1;`
	`double c = a + b; // 1E18`
	`// c - delta <= expected result <= c + delta`
	`// c - c * EPS <= expected result`
	`// expected_result <= c + c * EPS`

The step between the neighboring two integers is always the same and equal to one. Number of representable integers is the same for interval [1, 2] and [3,4].	The step between the neighboring two doubles exponentially grows with their magnitude. Number of representable floating-point values is different for interval [1,2] and [3,4].

RELATIVE ERROR ARITHMETIC

Operation	Relative error
`A+B, A*B >= 0`	max(re(A), re(B))
`A-B, A*B > 0`	(re(A)*B+re(B)*A) / |A-B|
`A*B`	re(A) + re(B)
`A/B`	re(A) + re(B)
`sqrt(A)`	0.5 * re(A)

ROBUST FPT

```
template <typename _fpt>
class robust_fpt {
public:
  typedef _fpt floating_point_type;
  typedef _fpt relative_error_type;

  // Rounding error is at most 1 EPS.
  static const relative_error_type ROUNDING_ERROR;

  robust_fpt operator+(const robust_fpt &that) const;
  robust_fpt operator-(const robust_fpt &that) const;
  robust_fpt operator*(const robust_fpt &that) const;
  robust_fpt operator/(const robust_fpt &that) const;
  robust_fpt sqrt() const

private:
  floating_point_type fpv_;
  relative_error_type re_;
};
```

ROBUST ORIENTATION TEST

```
#include <iostream>

int mp_robust_orientation(double dx1_, double dy1_, double dx2_, double dy2_) {
  MPFPT dx1(dx1_), dy1(dy1_), dx2(dx2_), dy2(dy2_);
  MPFPT lhs = dx1 * dy2; // Equal 2^60
  MPFPT rhs = dx2 * dy1; // Equal 2^60 - 1
  if (lhs == rhs)
    return 0;
  return (lhs < rhs) ? -1 : 1;
}

int robust_orientation(double dx1_, double dy1_, double dx2_, double dy2_) {
  robust_fpt<double> dx1(dx1_), dy1(dy1_), dx2(dx2_), dy2(dy2_);
  robust_fpt<double> lhs = dx1 * dy2; // Equal 2^60
  robust_fpt<double> rhs = dx2 * dy1; // Equal 2^60
  if (could_be_equal(lhs, rhs))
    return mp_robust_orientation(dx1_, dy1_, dx2_, dy2_);
  return (lhs < rhs) ? -1 : 1;
}

int main() {
  double v = (1 << 30);
  int or = robust_orientation(v, v+1, v-1, v);
  std::cout << or << std::endl; // Prints 1!
  return 0;
}
```

CANCELLATION ERROR

```
Expression: A - B + C, A ~ B, C << B

re(A-B+C)  = max(re(A-B), re(C))
           = re(A-B)
           = |B*re(A) + A*re(B)| / |A-B|
           = INF

re(A-B+C)  = re(C-B+A)
           = max(re(C-B), re(A))
           ~= max(re(B), re(A))
```

ROBUST DIFFERENCE

```
template <typename T>
class robust_dif {
  public:
    robust_dif(const T &value) :
        positive_sum_((value>0)?value:0),
        negative_sum_((value<0)?-value:0) {}

    T dif() const {
        return positive_sum_ - negative_sum_;
    }
  private:
    T positive_sum_;
    T negative_sum_;
};

// (A - B) * (C - D) = (A * C + B * D) - (A * C + B * D)
template<typename T>
robust_dif<T> operator*(const robust_dif<T>& lhs, const robust_dif<T>& rhs);
template<typename T>
robust_dif<T> operator*(const robust_dif<T>& lhs, const T& val);
template<typename T>
robust_dif<T> operator*(const T& val, const robust_dif<T>& rhs);

// (A - B) / C = (A / C - B / C)
template<typename T>
robust_dif<T> operator/(const robust_dif<T>& lhs, const T& val);
```

ROBUST SQRT EVALUATION

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

In general case sqrt produces irrational values, even arbitrary precision types won't be able to compute this expression in a robust way. That's why math. transformation is required.

Example with two square roots:

1) A * B >= 0, compute expression as it is:
C = A*sqrt(a) + B*sqrt(B)

2) A * B < 0, multiply by conjugate.
C = (A*A*a - B*B*b) / (A*sqrt(a) - B*sqrt(b));
A*(-B) > 0, so we can compute it in a robust way.

CONSIDER MOVING TO INTEGER INPUT COORDINATES

- It's much harder to ensure robustness for floating-point input types. Consider following example:

  ```
  A = 1E-100;
  B = 1E+100;
  C = A + B;
  ```

  C would consume a lot of memory.

- 32bit integer grid is enough to sample the whole area of Mars within 0.5 centimeter precision.

- Scaling and snapping to the integer grid would imply smaller relative error comparing to the one produced by measuring devices.

EFFICIENCY

KNOW YOUR ALGORITHM

ALGORITHM COMPLEXITY

Complexity	Running time	Example
constant	O(1)	Hash map lookup
logarithmic	O(log n)	Binary search
linear	O(n)	Max element
linearithmic	O(n log n)	Heap sort
quadratic	O(n^2)	Bubble sort
cubic	O(n^3)	Matrix multiplication
exponential	O(2^n)	Traveling salesman with DP

CHOOSING ALGORITHM

Algorithm	Complexity	Advantages	Disadvantages
Sweep-line	n*log(n)	Efficiency, generic interface	No major
Incremental	n*log(n)	Real time construction	Performance slowdown for large input data sets
Divide & Conquer	n*log(n)	Parallel computing	Complex merging step

HYBRID ALGORITHMS (SGI STL UNSTABLE SORT)

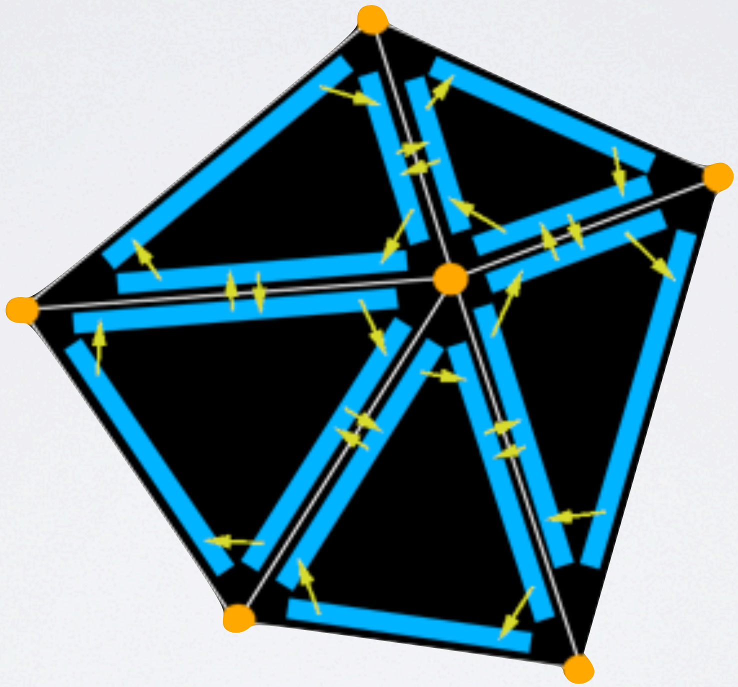
Introsort	Default sorting option
Heapsort	If recursion depth is exceeded
Insertion sort	For the small buckets

KNOW YOUR DATA STRUCTURES

STL COVERS 95% OF USAGE CASES

Container	Access	Type	Reallocation
vector	Random	Sequence	TRUE
list	Bidirectional	Sequence	FALSE
deque	Random	Sequence	TRUE
set (map)	Bidirectional	Associative	FALSE
unordered set (map)	Forward	Associative	TRUE
queue	No access	Adaptive	Depends
stack	No access	Adaptive	Depends
priority_queue	No access	Adaptive	Depends

VORONOI GRAPH STRUCTURE



STD::VECTOR & STD::LIST

```
template <typename T, typename TRAITS = voronoi_diagram_traits<T> >
class voronoi_diagram {
public:
  typedef typename TRAITS::coordinate_type coordinate_type;
  typedef typename TRAITS::point_type point_type;
  typedef typename TRAITS::cell_type cell_type;
  typedef typename TRAITS::vertex_type vertex_type;
  typedef typename TRAITS::edge_type edge_type;

  typedef std::list<cell_type> cell_container_type;
  typedef std::list<vertex_type> vertex_container_type;
  typedef std::list<edge_type> edge_container_type;

  /* Public methods */
private:
  cell_container_type cells_;
  vertex_container_type vertices_;
  edge_container_type edges_;
};
```

STD::VECTOR & STD::LIST

```
template <typename T, typename TRAITS = voronoi_diagram_traits<T> >
class voronoi_diagram {
public:
  typedef typename TRAITS::coordinate_type coordinate_type;
  typedef typename TRAITS::point_type point_type;
  typedef typename TRAITS::cell_type cell_type;
  typedef typename TRAITS::vertex_type vertex_type;
  typedef typename TRAITS::edge_type edge_type;

  typedef std::vector<cell_type> cell_container_type;
  typedef std::vector<vertex_type> vertex_container_type;
  typedef std::vector<edge_type> edge_container_type;

  void reserve(int num_sites);
  /* Public methods */
private:
  cell_container_type cells_;
  vertex_container_type vertices_;
  edge_container_type edges_;
};
```

MISCELLANEOUS

- Bitset (STL) & dynamic bitset (Boost)
- Disjoint data sets (Boost)
- Static array (Boost, C++11)
- Circular buffer (Boost)
- Unordered associative containers (Boost, C++11)
- Spatial index (Boost)

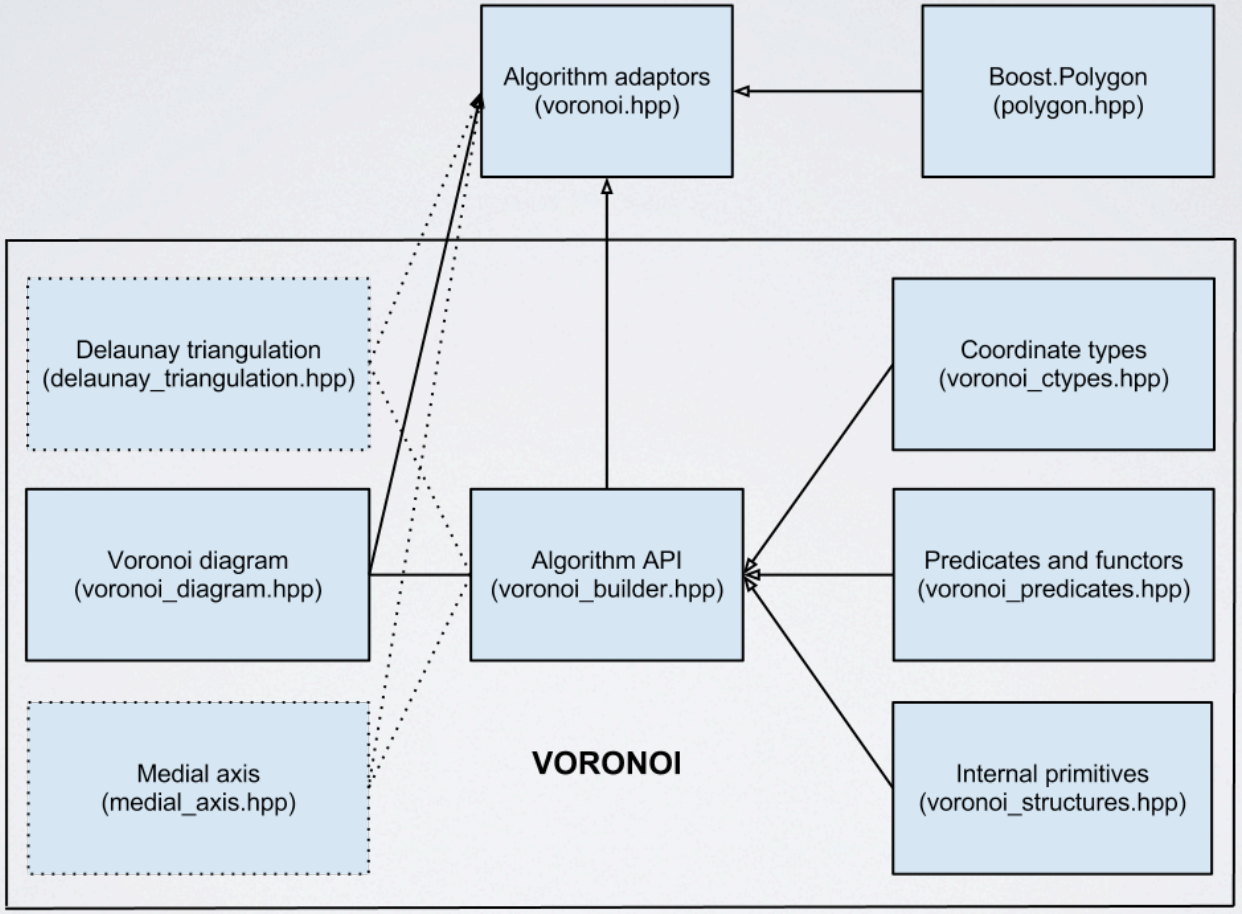
KNOW YOUR TYPES

COMPARING FLOATING-POINT TYPES

```
Result operator()(fpt64 a, fpt64 b, unsigned int maxUlps) const {
    uint64 ll_a, ll_b;

    // Reinterpret double bits as 64-bit signed integer.
    std::memcpy(&ll_a, &a, sizeof(fpt64));
    std::memcpy(&ll_b, &b, sizeof(fpt64));

    // Positive 0.0 is integer zero. Negative 0.0 is 0x8000000000000000.
    // Map negative zero to an integer zero representation - making it
    // identical to positive zero - the smallest negative number is
    // represented by negative one, and downwards from there.
    if (ll_a < 0x8000000000000000ULL)
        ll_a = 0x8000000000000000ULL - ll_a;
    if (ll_b < 0x8000000000000000ULL)
        ll_b = 0x8000000000000000ULL - ll_b;

    // Compare 64-bit signed integer representations of input values.
    // Difference in 1 Ulp is equivalent to a relative error of between
    // 1/4,000,000,000,000,000 and 1/8,000,000,000,000,000.
    if (ll_a > ll_b)
        return (ll_a - ll_b <= maxUlps) ? EQUAL : LESS;
    return (ll_b - ll_a <= maxUlps) ? EQUAL : MORE;
}
```

EXTENSIBILITY

MODULARIZATION



GENERIC PROGRAMMING VS INHERITANCE

Inheritance	Generic programming
• Run time polymorphism • Virtual function table storage • 6-50% time virtual call overhead	• Compile time polymorphism • Increased code size • Increased compilation time

GENERIC PROGRAMMING VS INHERITANCE

```
template <typename T>
class site_event {
public:
  site_event(const point_type &point) :
      point0_(point), point1_(point) {}
  site_event(const point_type &point1, const point_type &point2) :
      point0_(point1), point1_(point2) {}

  coordinate_type x0() const;
  coordinate_type y0() const;
  coordinate_type x1() const;
  coordinate_type y1() const;
  /* Other public methods follow. */
private:
  point_type point0_;
  point_type point1_;
  unsigned int site_index_;
};
```

TRAITS & DEFAULT TEMPLATE ARGUMENTS

```
// From voronoi_ctypes.hpp
template <typename T>
struct voronoi_ctype_traits;

template <>
struct voronoi_ctype_traits<int32> {
  typedef int32 int_type;
  typedef int64 int_x2_type;
  typedef uint64 uint_x2_type;
  typedef extended_int<64> big_int_type;
  typedef fpt64 fpt_type;
  typedef extended_exponent_fpt<fpt_type> efpt_type;
  typedef ulp_comparison<fpt_type> ulp_cmp_type;
  typedef type_converter_fpt to_fpt_converter_type;
  typedef type_converter_efpt to_efpt_converter_type;
};

// From voronoi_builder.hpp
template <typename T,
          typename CTT = detail::voronoi_ctype_traits<T>,
          typename VP = detail::voronoi_predicates<CTT> >
class voronoi_builder;
```

ADAPTORS

```
template <typename T, typename Predicate>
class ordered_queue {
public:
  ordered_queue();
  bool empty() const;
  const T &top() const;
  T &push(const T &e);
  void pop();
  void clear();

private:
  typedef typename std::list<T>::iterator list_iterator_type;

  struct comparison {
    bool operator() (list_iterator_type it1, list_iterator_type it2) const {
      return cmp_(*it1, *it2);
    }
    Predicate cmp_;
  };

   std::priority_queue< list_iterator_type,
                        std::vector<list_iterator_type>,
                        comparison > c_;
  std::list<T> c_list_;
};
```

BUILDER PATTERN (DIRECTOR)

```
template < typename T,
           typename CTT = detail::voronoi_ctype_traits<T>,
           typename VP = detail::voronoi_predicates<CTT> >
class voronoi_builder {
  /* ... */

  template <typename OUTPUT>
  void construct(OUTPUT *output) {
    output->builder()->reserve(site_events_.size());
    /* Process site/circle events.  */
    output->builder()->build();
  }

  /* ... */
};
```

BUILDER PATTERN (CONCRETE BUILDER)

```
template <typename T, typename TRAITS = voronoi_diagram_traits<T> >
class voronoi_diagram {

  class voronoi_diagram_builder {
    public:
      void reserve(size_t num_sites) { vd_->reserve(num_sites); };
      void build() { vd_->build(), vd_ = NULL };
      /* Other site processing methods */
    private:
      voronoi_diagram *vd_;
   };
   voronoi_diagram_builder *builder() const;

  /* Other public methods */

private:
  friend class voronoi_diagram_builder;
  voronoi_diagram_builder builder_;

   void reserve(int num_sites);
   void build();
  /* Other site processing methods */
};
```

FUNCTION OBJECTS

```
template <typename Point>
class point_comparison_predicate {
public:
  typedef Point point_type;

  bool operator()(const point_type &lhs, const point_type &rhs) const;
};

template <typename Site, typename Circle>
class event_comparison_predicate {
  public:
    typedef Site site_type;
    typedef Circle circle_type;

    bool operator()(const site_type &lhs, const site_type &rhs) const;
    bool operator()(const site_type &lhs, const circle_type &rhs) const;
    bool operator()(const circle_type &lhs, const site_type &rhs) const;
    bool operator()(const circle_type &lhs, const circle_type &rhs) const;

  private:
    ulp_cmp_type ulp_cmp;
    to_fpt_converter to_fpt;
};
```

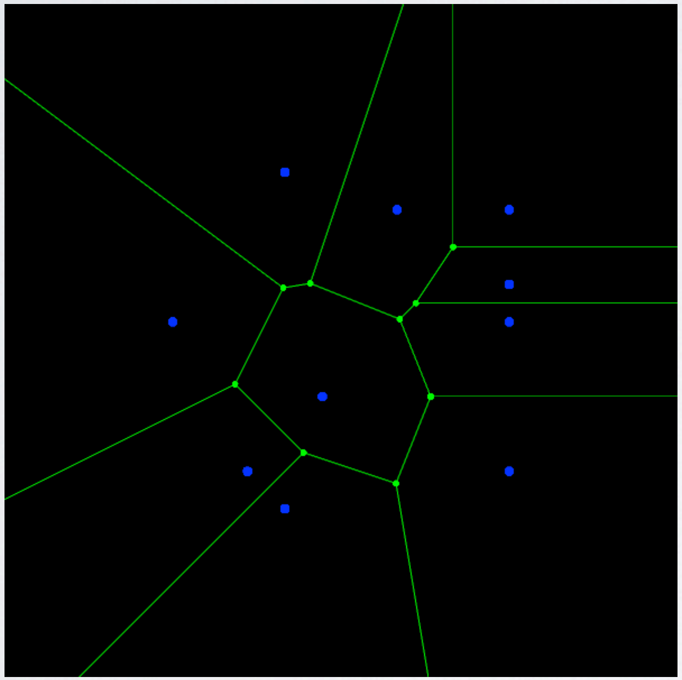
MAINTENANCE

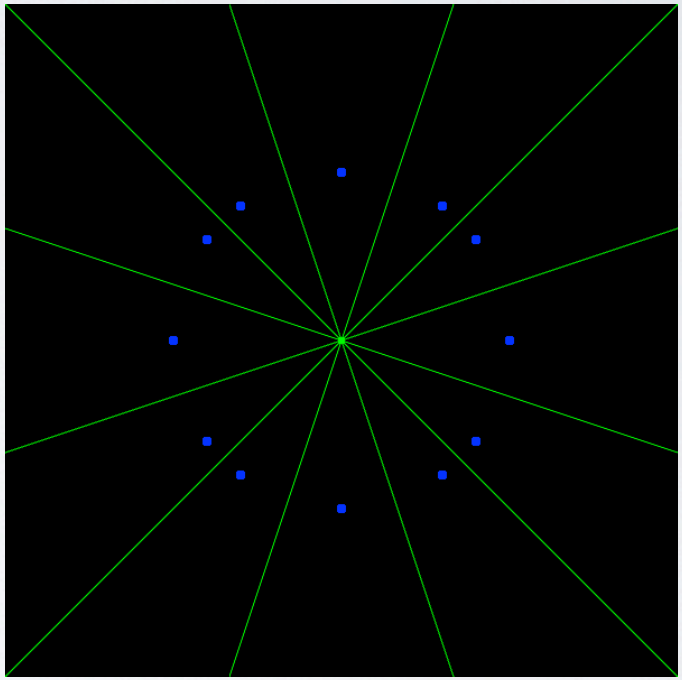
Debugging	Helper classes, loggers, assertions, bug localization
Testing	Unit testing, coverage testing, regression testing, performance testing
Readability	Leave comments, follow style guides
Documentation	If you are not the single library user, this would be required anyway

TESTING

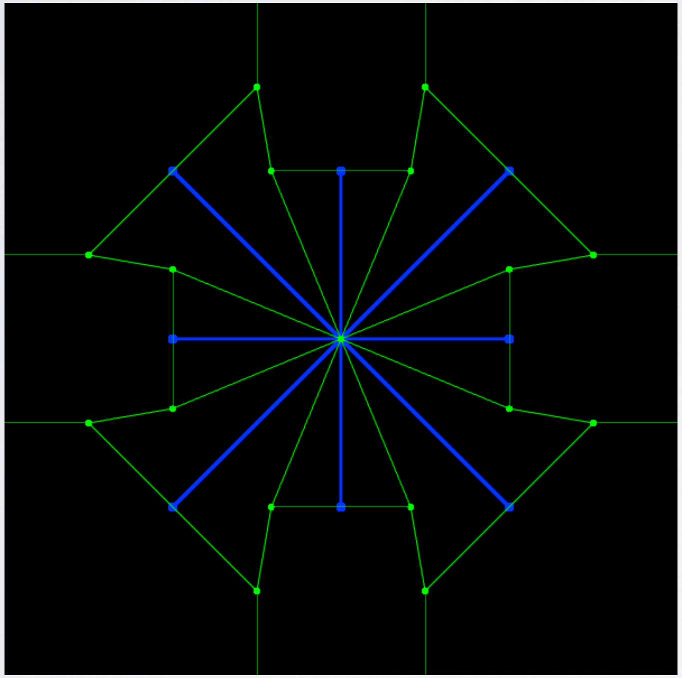
- Unit tests provide a good way of bug localization
- Check branch code coverage, especially within math. functions
- Random tests don't handle corner cases
- Make sure that random tests are random
- Validate output of random tests
- Use benchmark tests to identify performance regressions

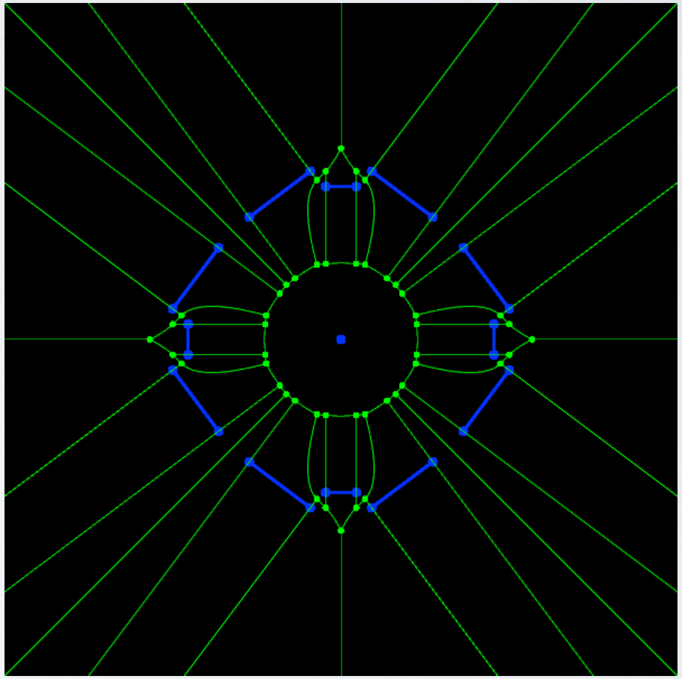
RANDOMNESS VS DETERMINISM





CORNER CASES





USABILITY

SIMPLE & INTUITIVE INTERFACE STEP 1

```
#include <boost/polygon/voronoi_builder.hpp>
#include <boost/polygon/voronoi_diagram.hpp>
using namespace boost::polygon;

template <typename Container>
void render_voronoi(const Container& input) {
  typedef detail::voronoi_ctype_traits<int> ctype_traits;
  typedef detail::voronoi_predicates<ctype_traits> predicates;
  typedef voronoi_diagram_traits<double> vd_traits;

  voronoi_builder<int, ctype_traits, predicates> vb;
  voronoi_diagram<double, vd_traits> vd;

  for (int i = 0; i < input.size(); ++i)
    vb.insert_point(x(input[i]), y(input[i]));

  vb.construct(&vd);
  // Rendering code follows.
}
```

DEFAULT TEMPLATE ARGUMENTS

std::priority_queue	`template < class T, class Container = vector<T>, class Compare = less<typename Container::value_type> > class priority_queue;`
std::set	`template < class Key, class Compare = less<Key>, class Allocator = allocator<Key> > class set;`
voronoi_builder	`template < typename T, typename CTT = detail::voronoi_ctype_traits<T>, typename VP = detail::voronoi_predicates<CTT> > class voronoi_builder;`
voronoi_diagram	`template < typename T, typename TRAITS = voronoi_diagram_traits<T> > class voronoi_diagram;`

SIMPLE & INTUITIVE INTERFACE STEP 2

```
#include <boost/polygon/voronoi_builder.hpp>
#include <boost/polygon/voronoi_diagram.hpp>
using namespace boost::polygon;

template <typename Container>
void render_voronoi(const Container& input) {
  voronoi_builder<int> vb;
  voronoi_diagram<double> vd;

  for (int i = 0; i < input.size(); ++i)
    vb.insert_point(x(input[i]), y(input[i]));

  vb.construct(&vd);
  // Rendering code follows.
}
```

PUBLIC FUNCTIONS

```
template <typename PointIterator, typename VD>
void construct_voronoi_points(PointIterator first,
                              PointIterator last,
                              VD *vd)
```

```
template <typename SegmentIterator, typename VD>
void construct_voronoi_segments(SegmentIterator first,
                                SegmentIterator last,
                                VD *vd)
```

```
template <typename PolygonIterator, typename VD>
void construct_voronoi_polygons(PolygonIterator first,
                                PolygonIterator last,
                                VD *vd)
```

SIMPLE & INTUITIVE INTERFACE STEP 3

```
#include <boost/polygon/voronoi>
using namespace boost::polygon;

template <typename Container>
void render_voronoi(const Container& input) {
  voronoi_diagram<double> vd;

  construct_voronoi_points(input.begin(), input.end(), &vd);
  // Rendering code follows.
}
```

SFINAE & MPL

```
template <typename PointIterator, typename VD>
typename enable_if<
  typename gtl_if<
    typename is_point_concept<
      typename geometry_concept<
        typename std::iterator_traits<PointIterator>::value_type
      >::type
    >::type
  >::type,
  void
>::type
construct_voronoi(PointIterator first, PointIterator last, VD *vd);
```

```
template <typename SegmentIterator, typename VD>
typename enable_if<
  typename gtl_if<
    typename is_segment_concept<
      typename geometry_concept<
        typename std::iterator_traits< SegmentIterator >::value_type
      >::type
    >::type
  >::type,
  void
>::type
construct_voronoi(SegmentIterator first, SegmentIterator last, VD *vd);
```

SFINAE & MPL

```
template <typename PointIterator, typename VD>
void construct_voronoi(PointIterator first, PointIterator last, VD *vd);
```

```
template <typename SegmentIterator, typename VD>
void construct_voronoi(SegmentIterator first, SegmentIterator last, VD *vd);
```

```
template <typename PolygonIterator, typename VD>
void construct_voronoi(PolygonIterator first, PolygonIterator last, VD *vd);
```

SIMPLE & INTUITIVE INTERFACE STEP 4

```
#include <boost/polygon/voronoi>
using namespace boost::polygon;

template <typename Container>
void render_voronoi(const Container& input) {
  voronoi_diagram<double> vd;

  construct_voronoi(input.begin(), input.end(), &vd);
  // Rendering code follows.
}
```

C++11

```
template <typename PointIterator>
voronoi_diagram<double>
construct_voronoi(PointIterator first, PointIterator last) {
  default_voronoi_builder builder;
  default_voronoi_diagram diagram;
  insert(first, last, &builder);
  builder.construct(&diagram);
  return diagram;
}
```

```
template <typename SegmentIterator>
voronoi_diagram<double>
construct_voronoi(SegmentIterator first, SegmentIterator last);
```

```
template <typename PolygonIterator>
voronoi_diagram<double>
construct_voronoi(PolygonIterator first, PolygonIterator last);
```

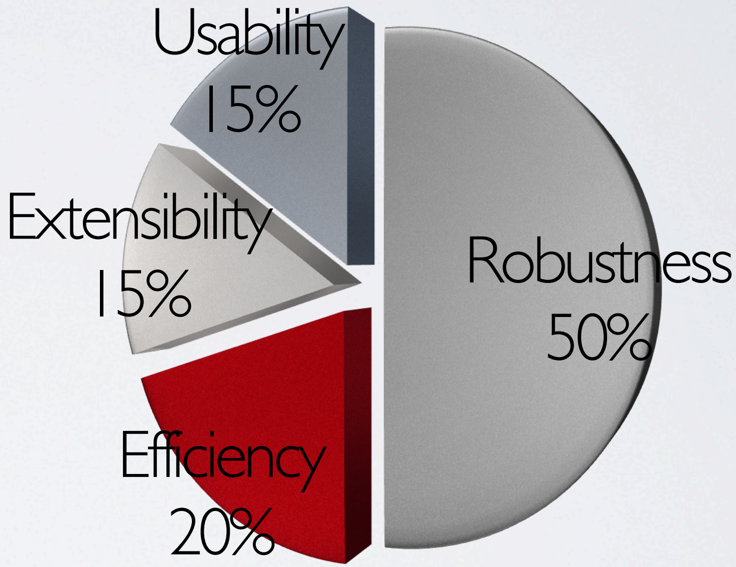
SIMPLE & INTUITIVE INTERFACE STEP 5

```
#include <boost/polygon/voronoi>
using namespace boost::polygon;

template <typename Container>
void render_voronoi(const Container& input) {
  voronoi_diagram<double> vd =
      construct_voronoi(begin(input), end(input));
  // Rendering code follows.
}
```

RECAP

DEVELOPMENT TIME

Aspect	Time
Robustness	12 months
Efficiency	5 months
Extensibility	3.5 months
Usability	3.5 months
Total	24 months



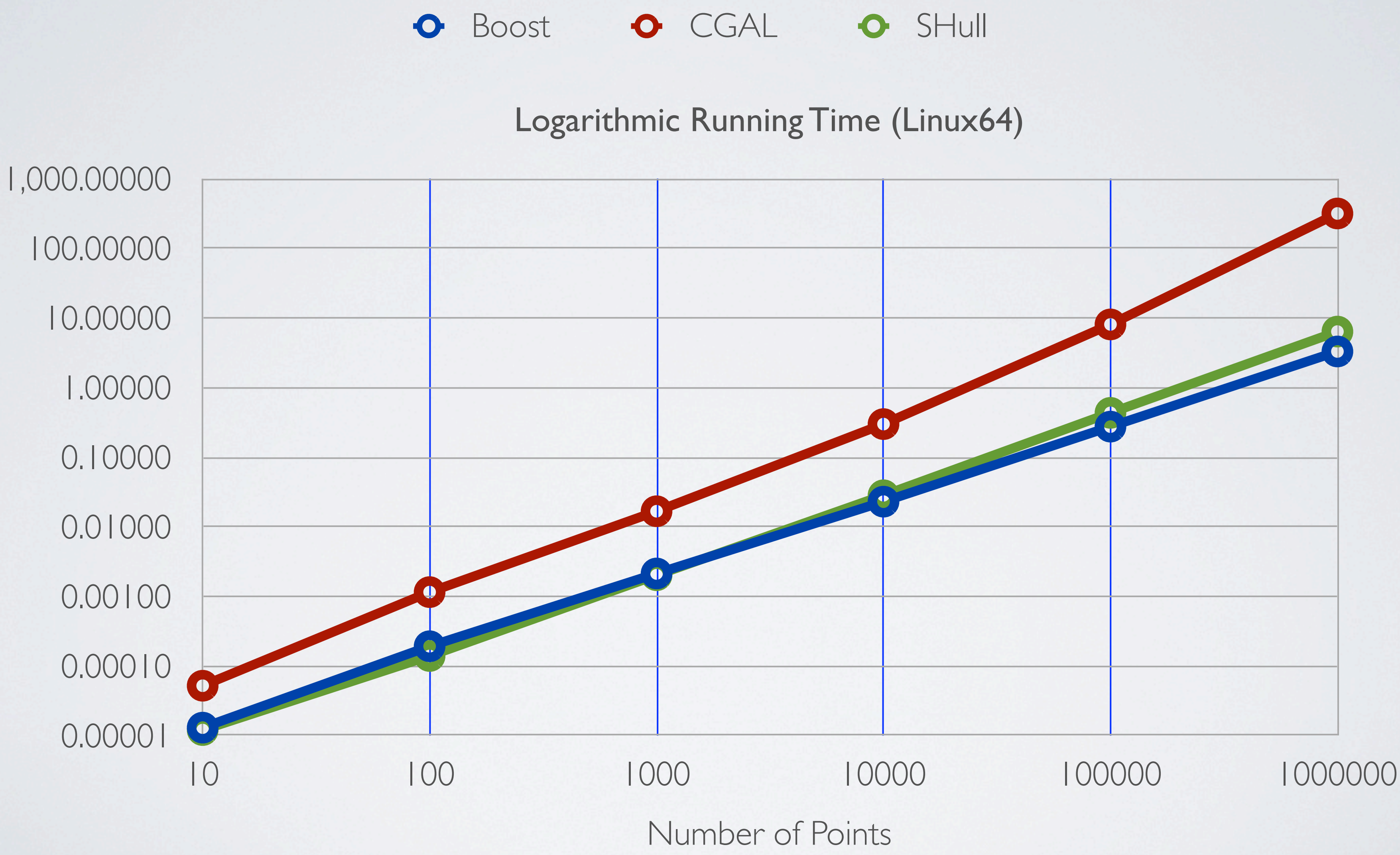
BENCHMARKS

Target	Configuration

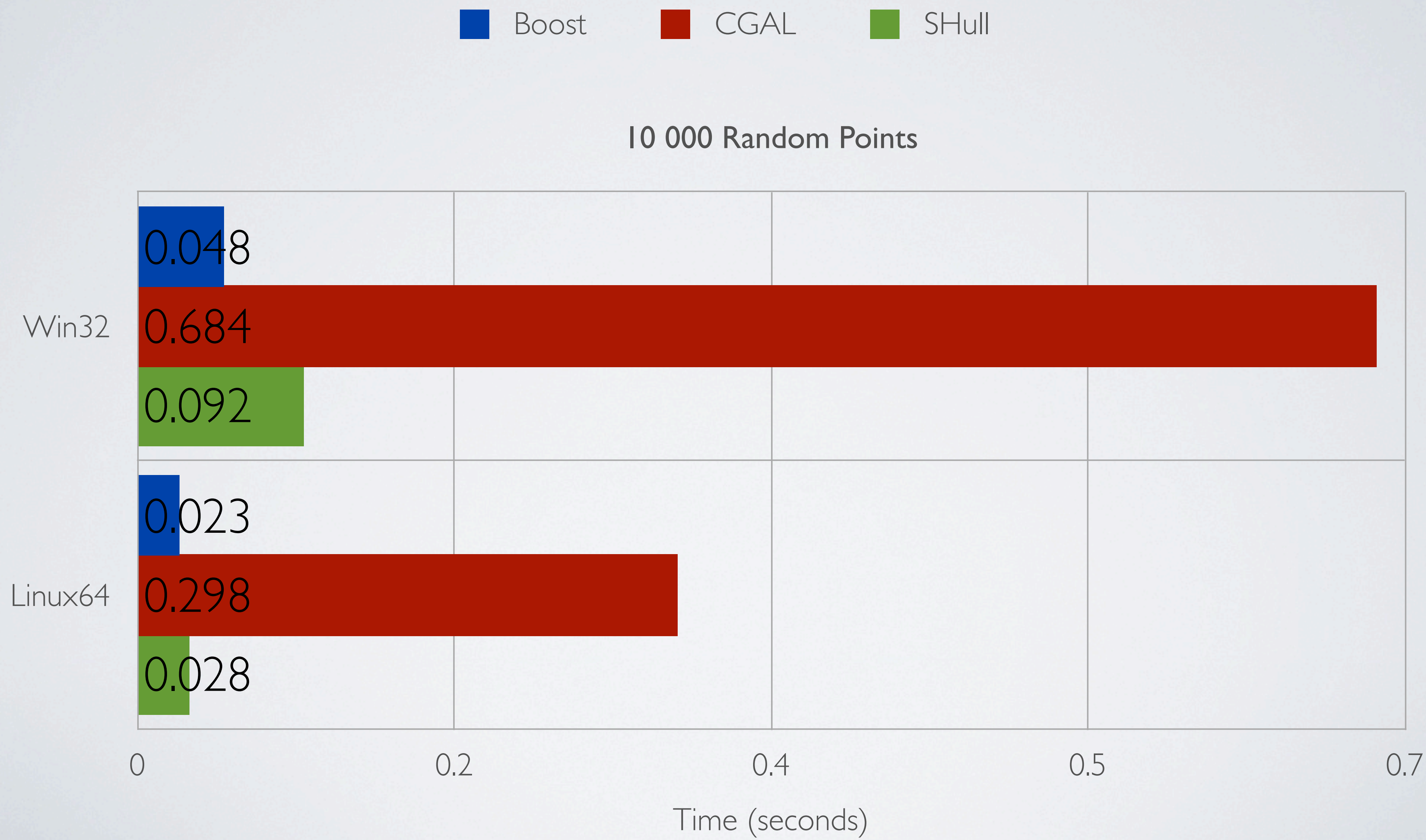
Win32	`System: CPU i5-7600 2.8 GHz, 4Gb RAM`
	`OS: Windows 7 Professional 32 bit`
	`Compiler: MSVC-9.0`

Linux64	`System: CPU i5-7600 2.8 GHz, 4Gb RAM`
	`OS: Ubuntu 11.10 64 bit`
	`Compiler: gcc 4.6.1`

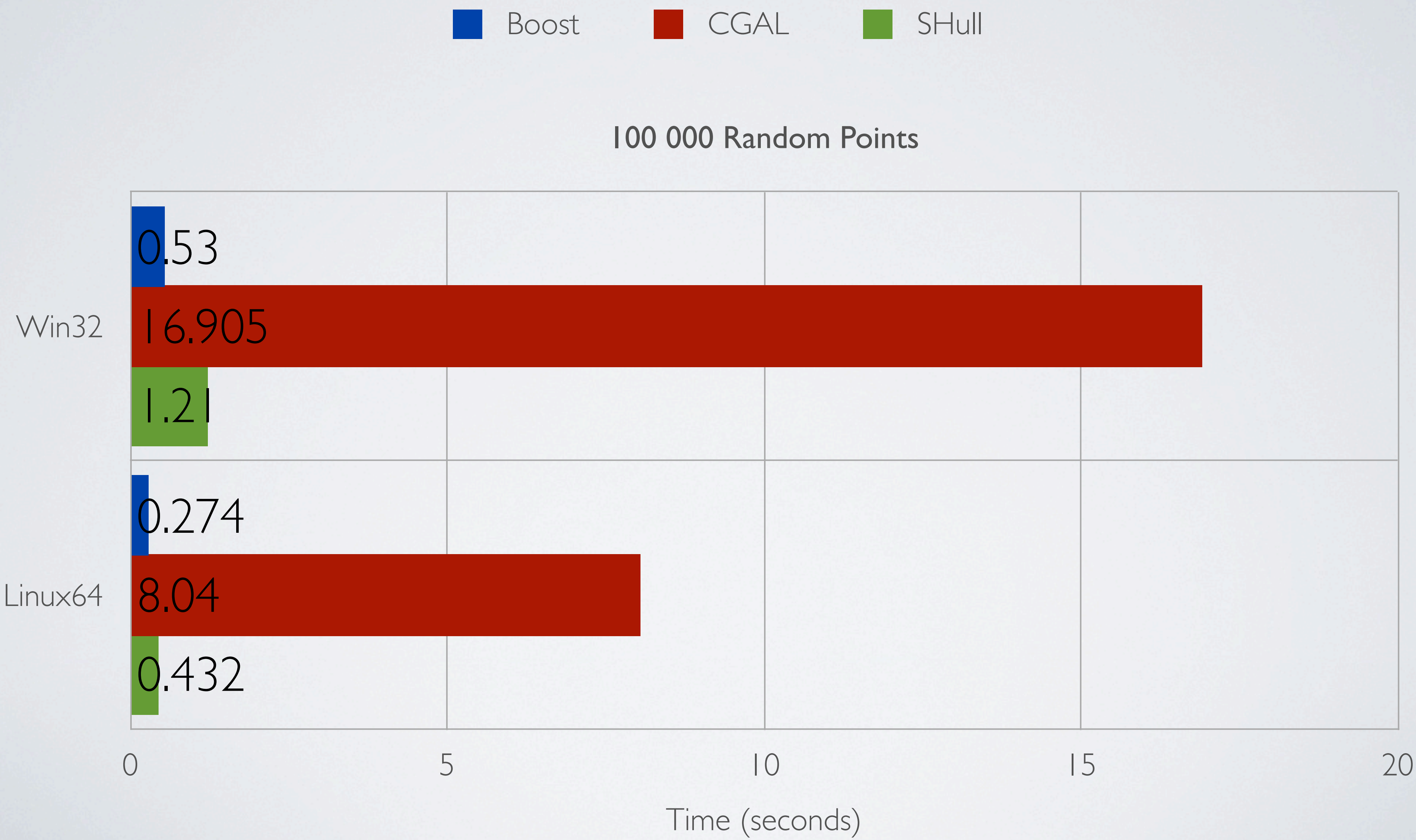
BENCHMARKS


Boost
CGAL
SHull
Logarithmic Running Time (Linux64)
1,000.00000
100.00000
10.00000
1.00000
0.10000
0.01000
0.00100
0.00010
0.00001
10
100
1000
10000
100000
1000000
Number of Points

BENCHMARKS


Boost
CGAL
SHull
10 000 Random Points
Win32
0.048
0.684
0.092
Linux64
0.023
0.298
0.028
0
0.2
0.4
0.5
0.7
Time (seconds)

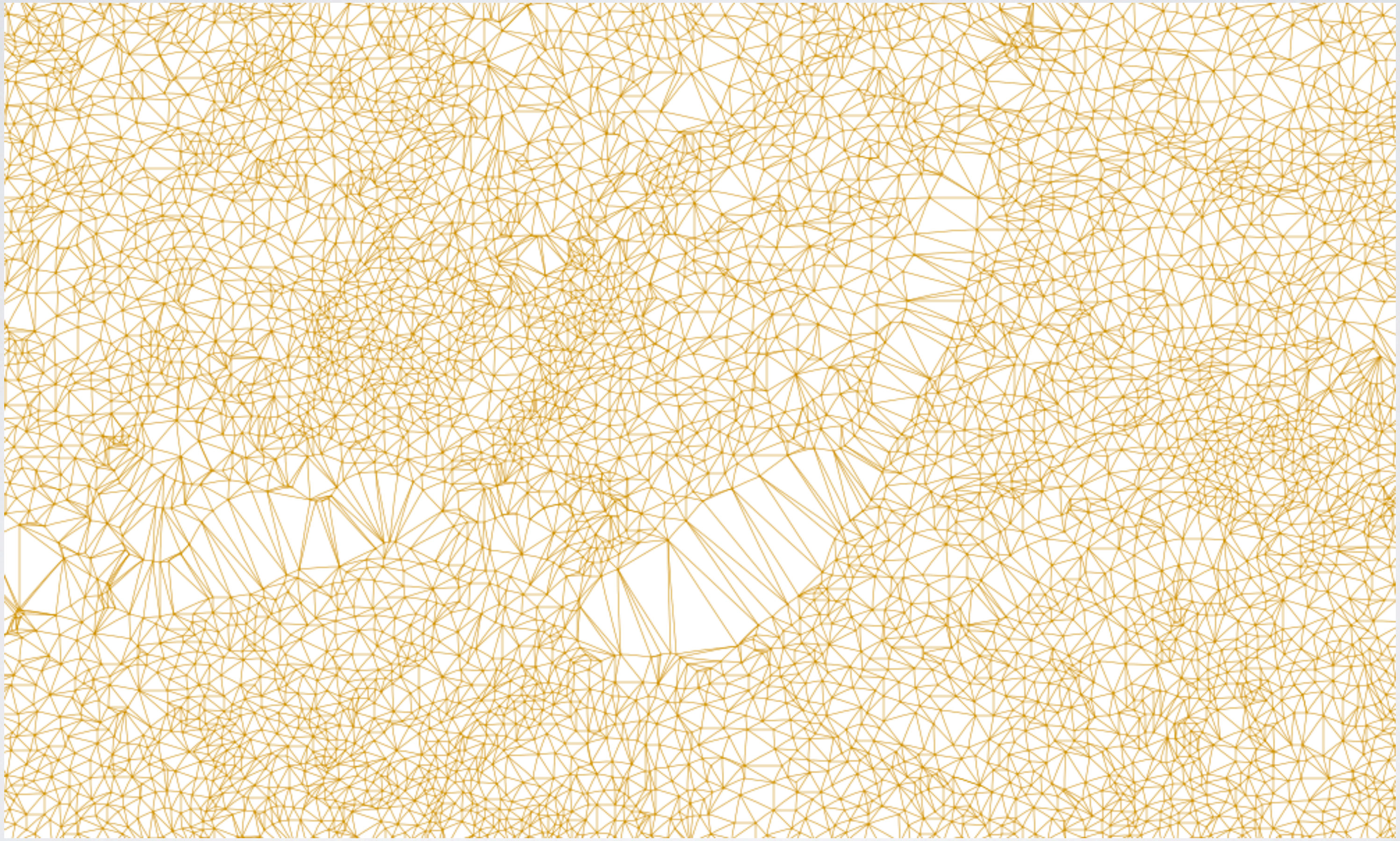
BENCHMARKS



Boost
CGAL
SHull
100 000 Random Points
Win32
0.53
16.905
1.21
Linux64
0.274
8.04
0.432
0
5
10
15
20
Time (seconds)

USER EXPERIENCE

(RENDERED BY **PHIL ENDECOTT**)



Q&A