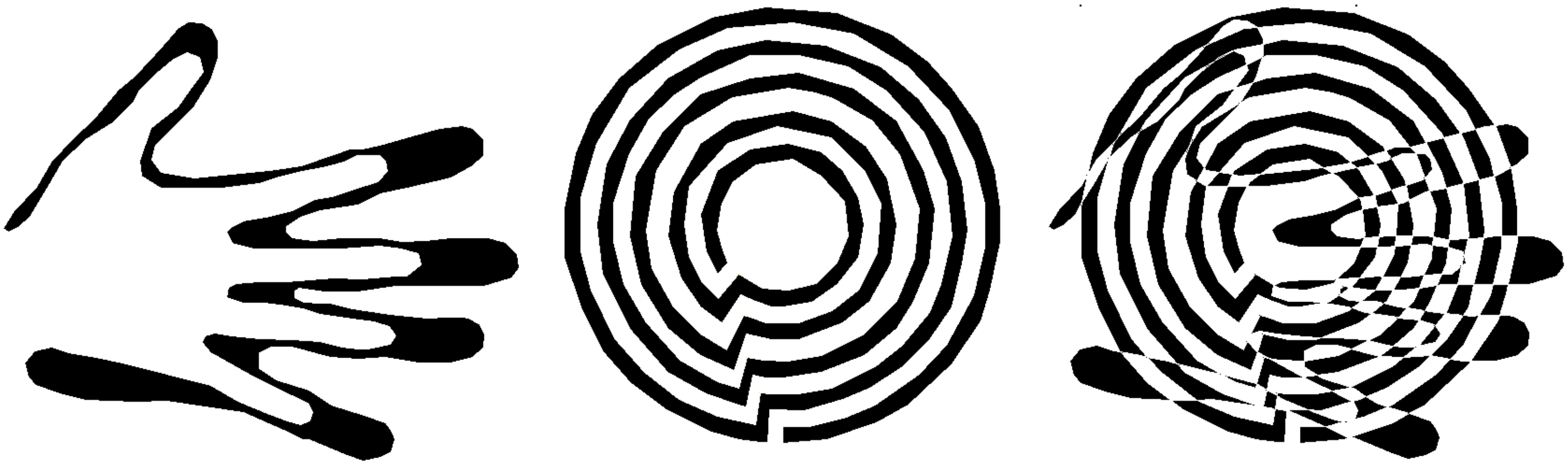


GTL Geometry Template Library

-for stl-like polygon manipulation

Lucanus Simonson, Gyuszi Suto

Intel Corporation

Overview

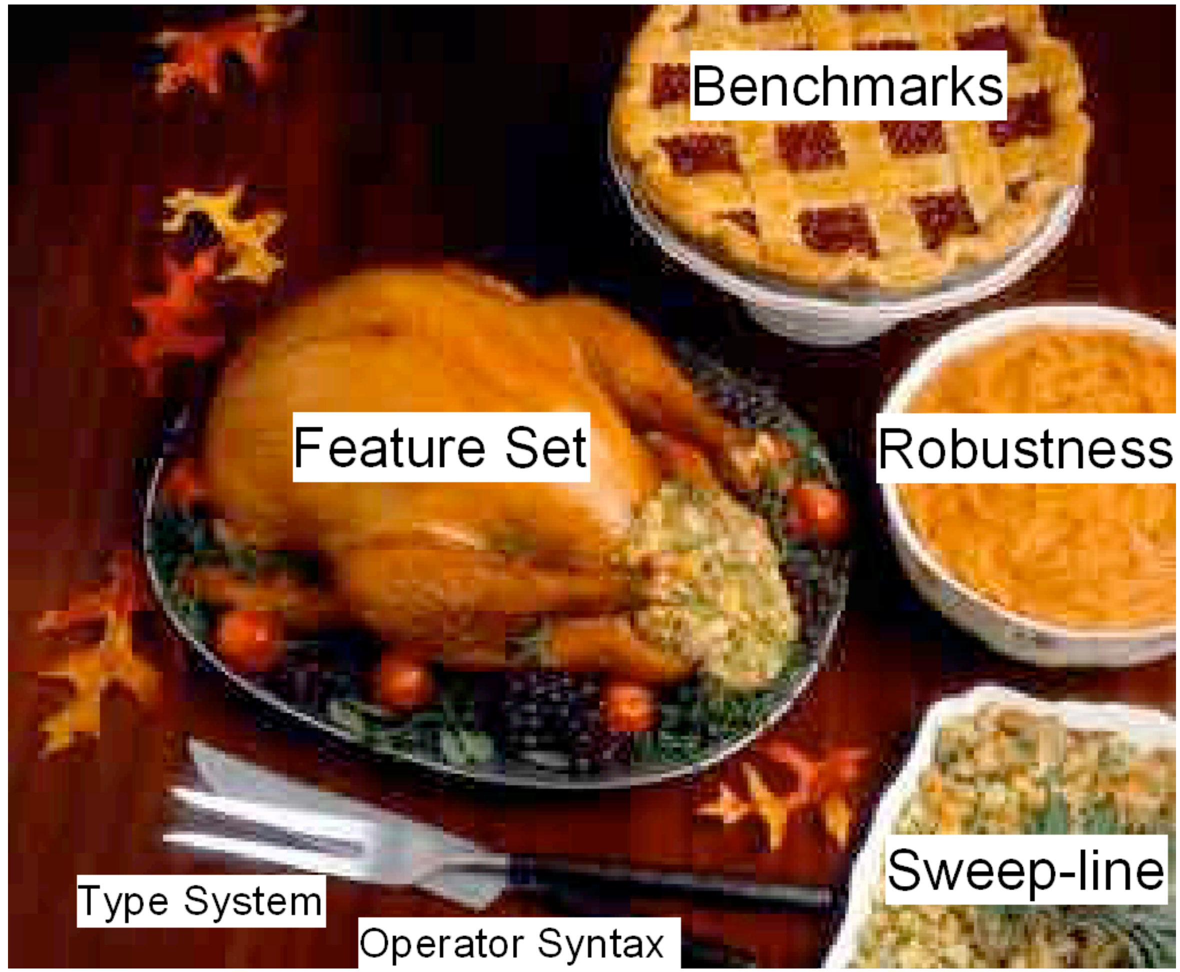
- Intel badly needed high performance algorithms for planar polygon manipulation
  - I implemented them
- We have 2D Cartesian geometry
  - Coordinate, Interval, Point, Rectangle, Polygon, Polygon Set
  - Library of concepts for each
- Many generic functions that operate on conceptual types
  - API strives for symmetry, consistency and simplicity
- Some pretty heavy weight algorithms under the hood
- 3 man years and 30kloc

Introduction

- Implemented goofy template argument inheritance type system and Manhattan geometry features
- Request for interest from boost in 2007
  - Discussed the design on boost dev list
  - Found out the design was bad and needed to be redone the boost way
    - Thank you Joel Guzman
- Added 45 degree geometry features
- After six months of work we got permission from Intel to release under boost license
  - Discussed the code on the boost dev list
  - Got a lot of feedback on specific design considerations
- Rewrote the interfaces to be more generic by using tag dispatching
  - Got more feedback on design considerations from boost, especially refinement
- Re-rewrote the interfaces to be more generic still and based on SFINAE
- Added arbitrary-angle geometry features
  - Got feedback on arbitrary-angle algorithms and robustness considerations from boost
    - Thank you Fernando Cacciola
- Ported new SFINAE interfaces to MSVC9
  - Thank you Steven Watanabe
- The library now looks more like Joel said it should back in 2007
  - We may pursue formal review this year
- Deployed library to internal users who are using it now to create the next generation of silicon fabrication process technology and microprocessors

Agenda

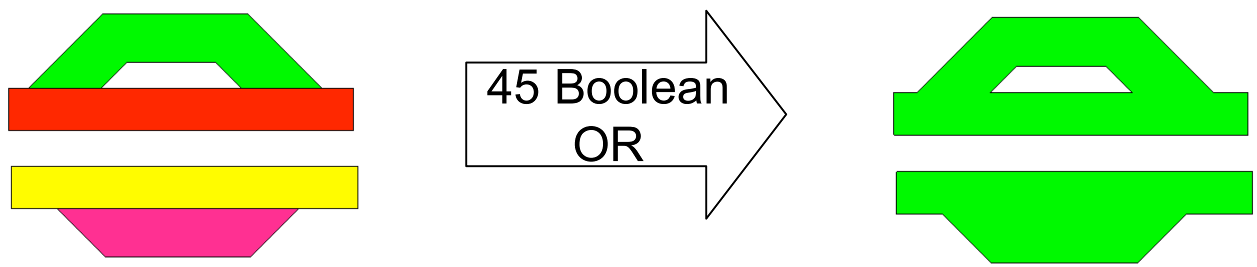
- GTL Feature Set
- Benchmark Comparisons
- Generic Sweep-line Booleans Algorithm
- Numerical Robustness
- Geometry Concepts Type System
- Booleans Operator Syntax



Primary GTL Feature

- Boolean operations on sets of polygons
  - Manhattan

  - 45-degree



  - Arbitrary Angle (XOR)

Using Booleans

```
void clip_and_subtract(polygon_set& d,
           polygon a, polygon b, rectangle c) {
  d = (a & c) - b;
}
```

- Productive operator syntax
- Clip polygon `a` against bounding box `c`, then subtract polygon `b`, storing the result in polygon set `d`
- Takes longer to say than to type
- No try/catch and no memory management

Details Of Booleans

- No preconditions placed on input polygons

Details Of Booleans

- No preconditions placed on input polygons
  - Open/closed semantic for last vertex

Details Of Booleans

- No preconditions placed on input polygons
  - Open/closed semantic for last vertex
  - Winding direction conventions not enforced

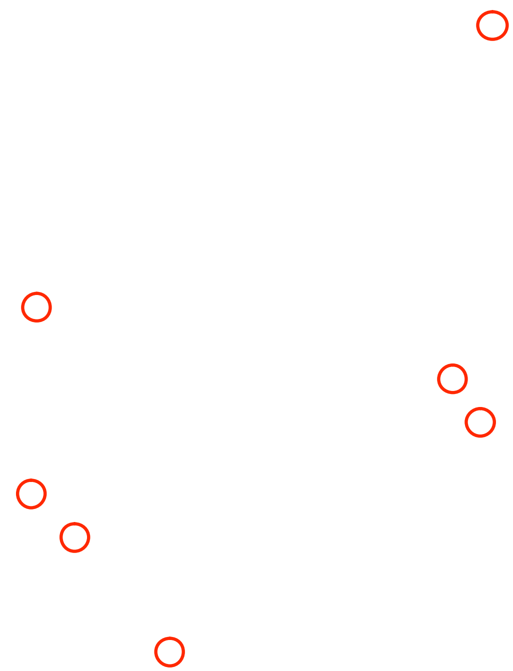
?

Details Of Booleans

- No preconditions placed on input polygons
  - Open/closed semantic for last vertex
  - Winding direction conventions not enforced
  - Input polygons may be
    - self touching

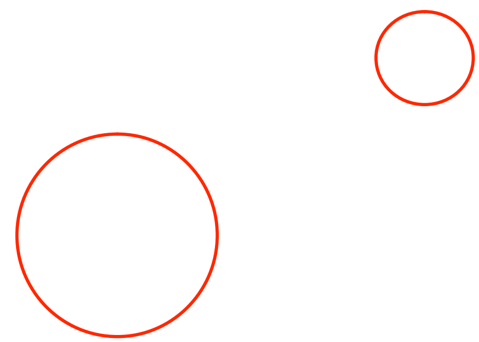
Details Of Booleans

- No preconditions placed on input polygons
  - Open/closed semantic for last vertex
  - Winding direction conventions not enforced
  - Input polygons may be
    - self touching
    - self intersecting



Details Of Booleans

- No preconditions placed on input polygons
  - Open/closed semantic for last vertex
  - Winding direction conventions not enforced
  - Input polygons may be
    - self touching
    - self intersecting
    - self overlapping

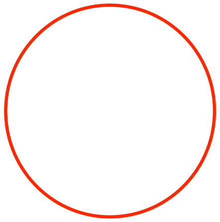


Details Of Booleans

- No preconditions placed on input polygons
  - Open/closed semantic for last vertex
  - Winding direction conventions not enforced
  - Input polygons may be
    - self touching
    - self intersecting
    - self overlapping
  - Correctly handles duplicate/co-linear points

Details Of Booleans

- No preconditions placed on input polygons
  - Open/closed semantic for last vertex
  - Winding direction conventions not enforced
  - Input polygons may be
    - self touching
    - self intersecting
    - self overlapping
  - Correctly handles duplicate/co-linear points
  - Correctly handles zero degree angles and polygons that degenerate to lines and points

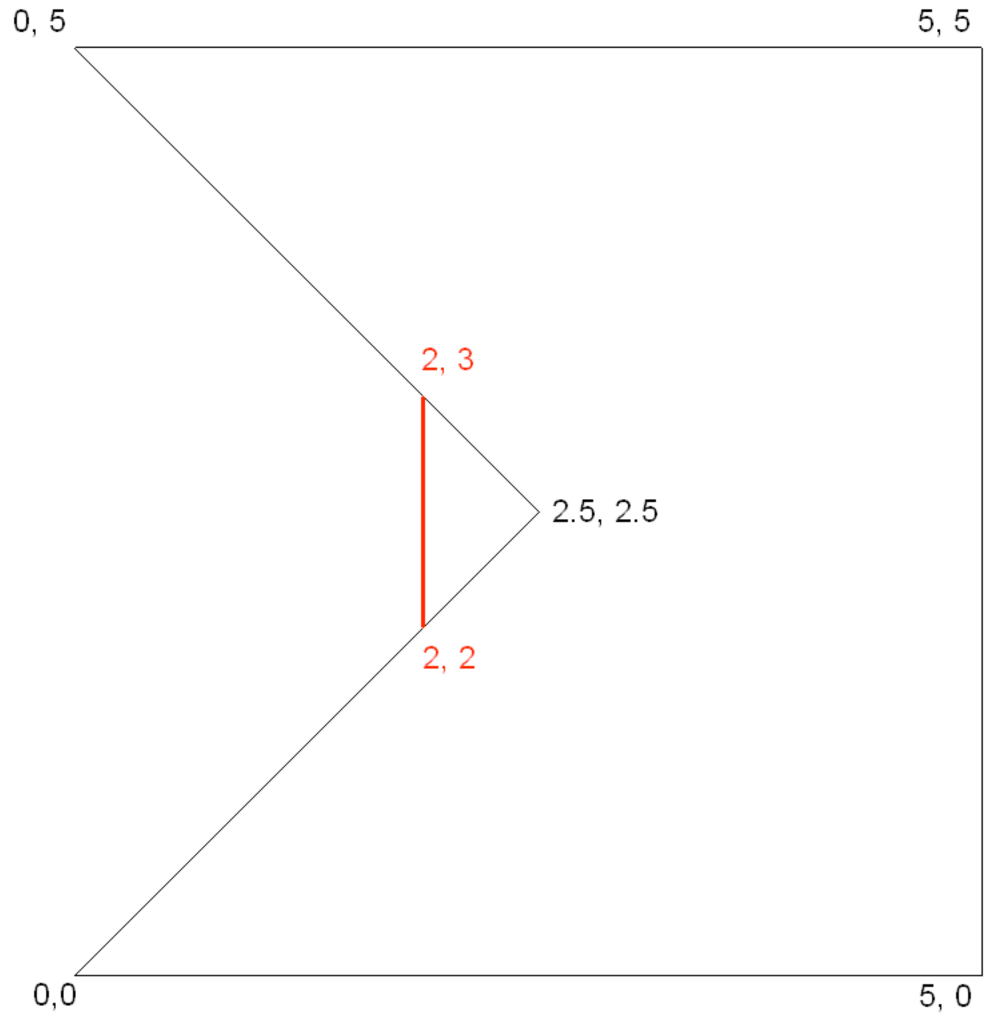


Details Of Booleans

- No preconditions placed on input polygons
  - Open/closed semantic for last vertex
  - Winding direction conventions not enforced
  - Input polygons may be
    - self touching
    - self intersecting
    - self overlapping
  - Correctly handles duplicate/co-linear points
  - Correctly handles zero degree angles and polygons that degenerate to lines and points
  - To produce a clean result

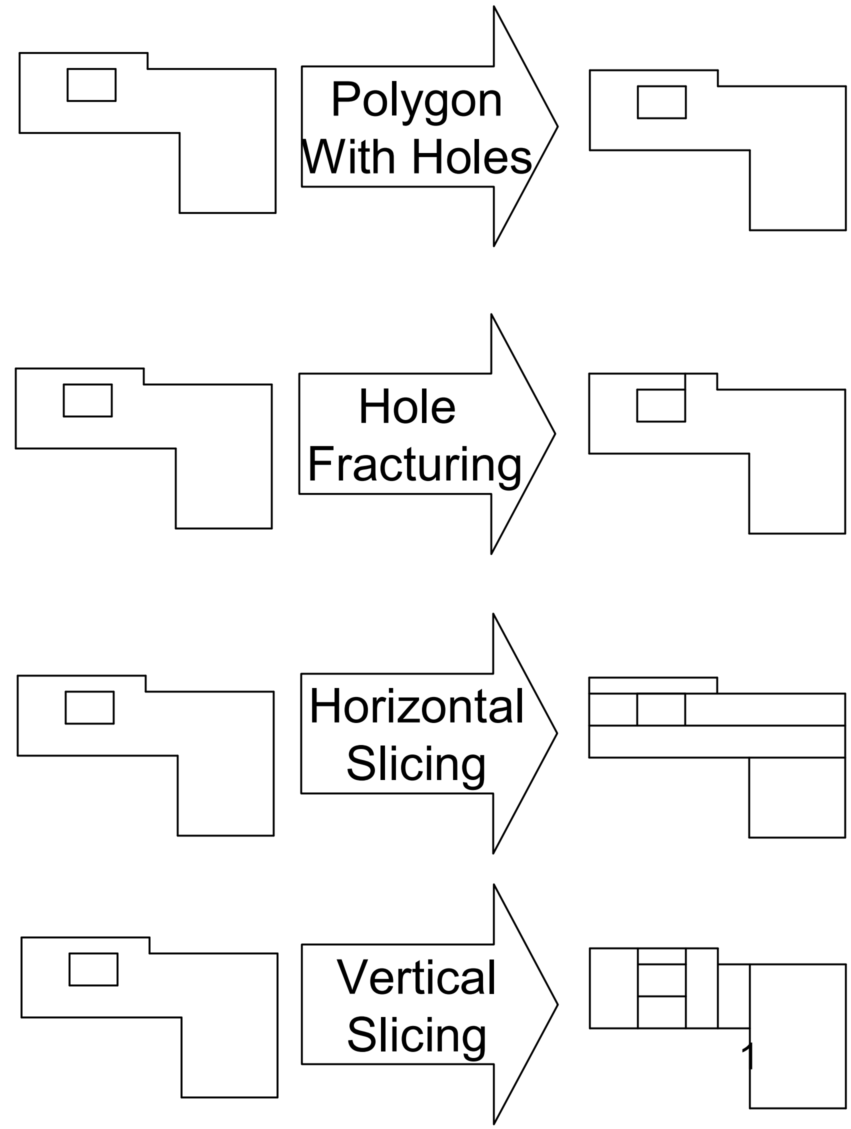
Details of 45-degree Booleans

- Preserve 45-degree nature of geometry at output
- Handle off-grid intersections by inserting an edge to approximate the output region



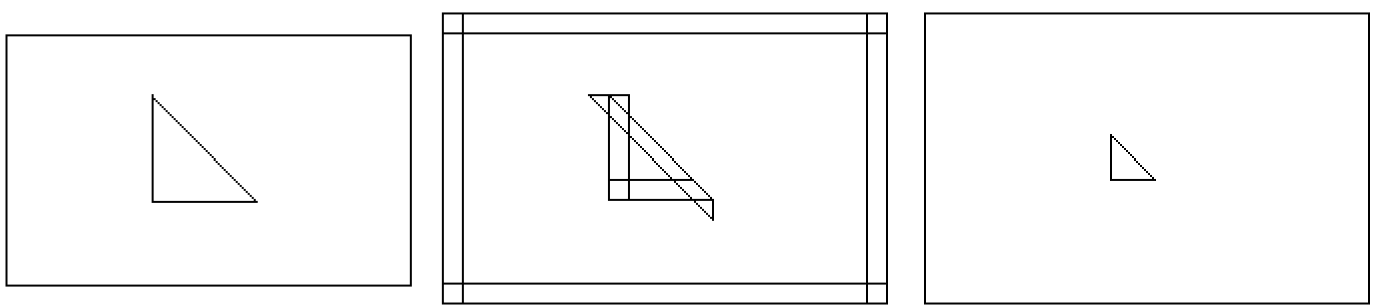
Boolean Operation Output Modes

- Manhattan Booleans
  - Polygons with lists of holes
  - Keyhole holes to outer polygon
  - Horizontal and vertical sliced rectangle tiling
- 45-degree Booleans
  - Polygon with lists of holes
  - Keyhole holes to outer polygon
  - Vertical sliced trapezoid tiling
- Arbitrary-angle Booleans
  - Polygon with lists of holes
  - Keyhole holes to outer polygon



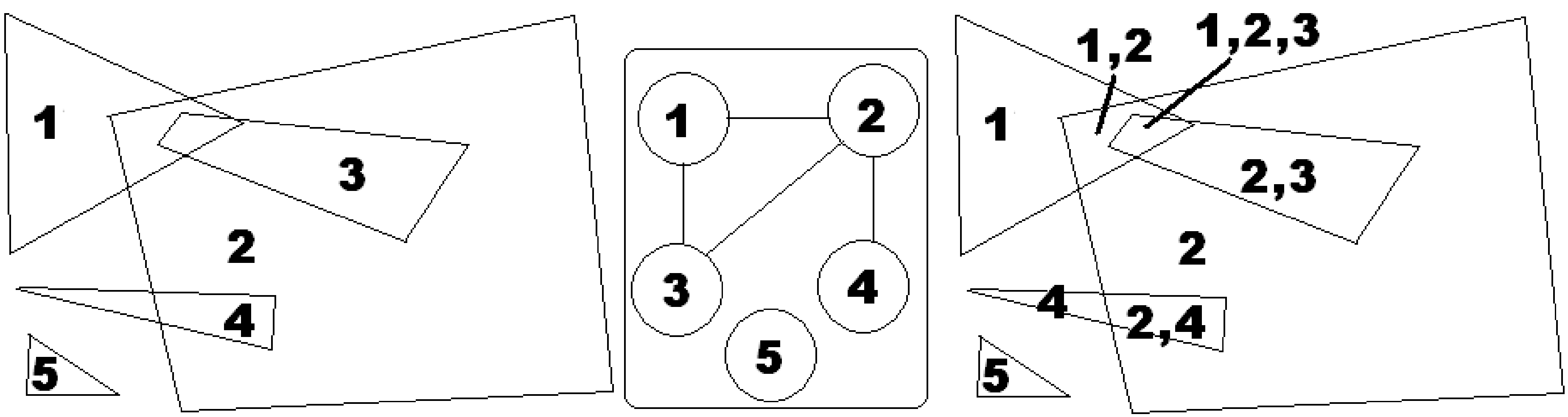
Polygon Buffering/Resizing/Offsetting

- Manhattan
  - Uniform resizing
  - Resizing by different amount in each of the four directions
  - Optionally leave corners unfilled
- 45-Degree
  - Uniform resizing
  - Preserve original topology or cut off acute angled corners at resizing distance
  - Snapping options for moving 45-degree edges

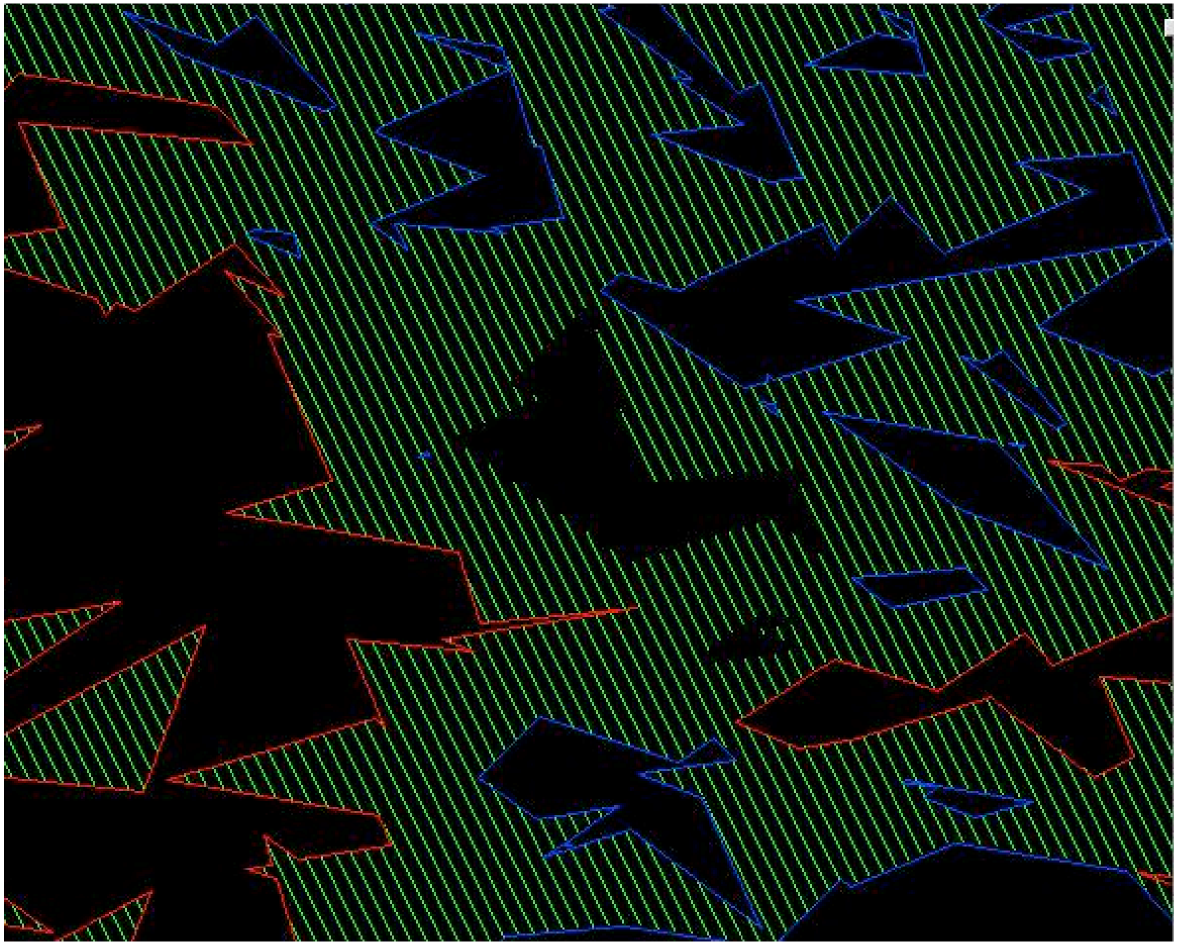


Many More Features

- Rectangle query tree
- Maximum enclosed rectangle in Manhattan polygon
- Connectivity Extraction
- Property Merge/Map Overlay
- Etc.

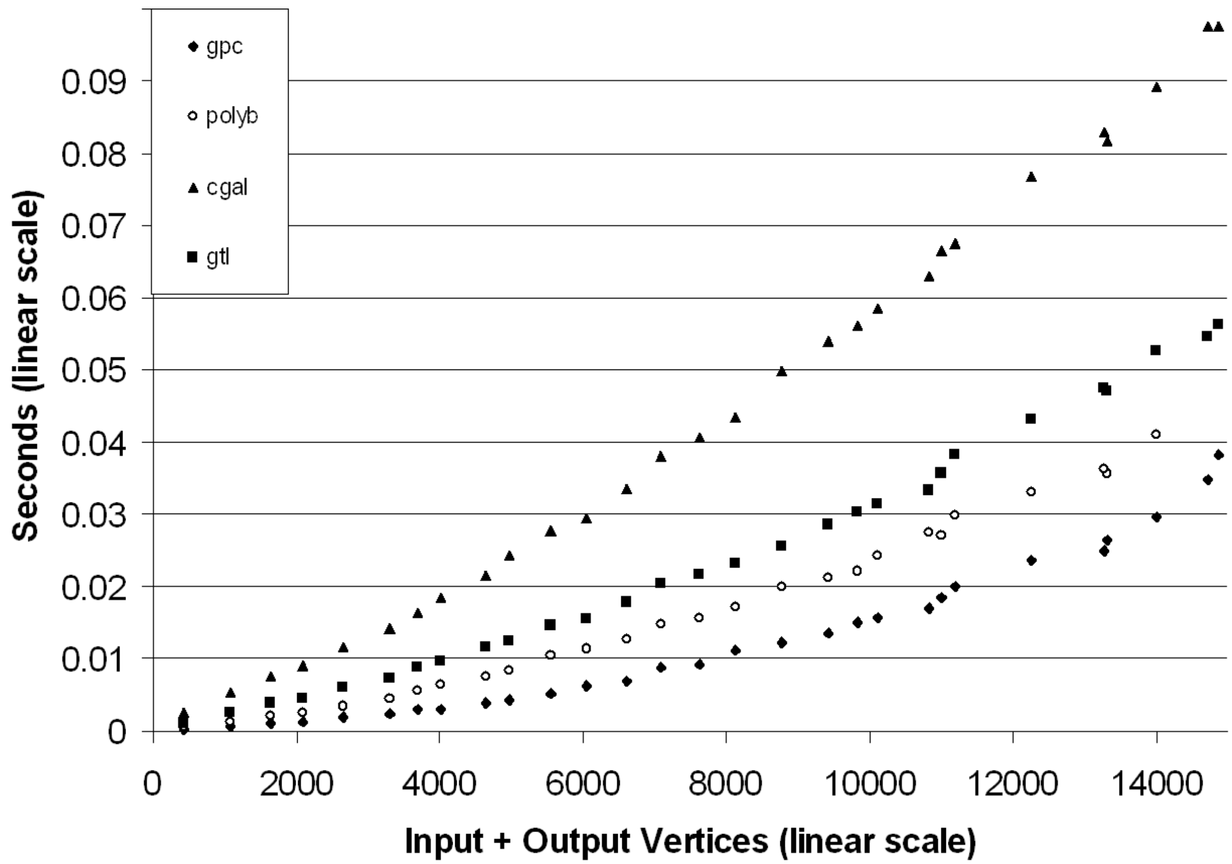


Small Arbitrary-angle Input Benchmark Comparison



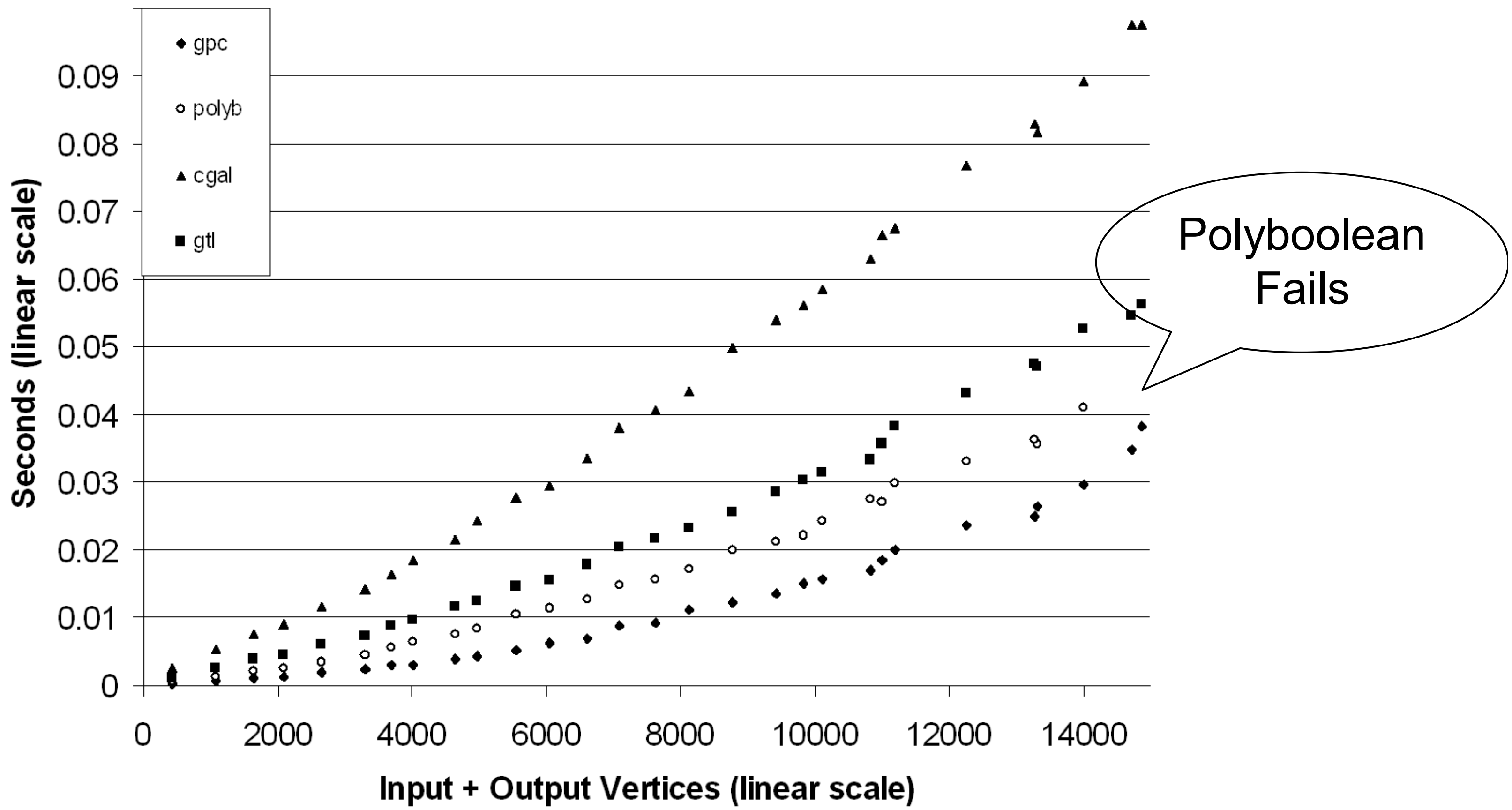
- Runtime for intersection operation

Small Arbitrary-angle Input Benchmark Comparison



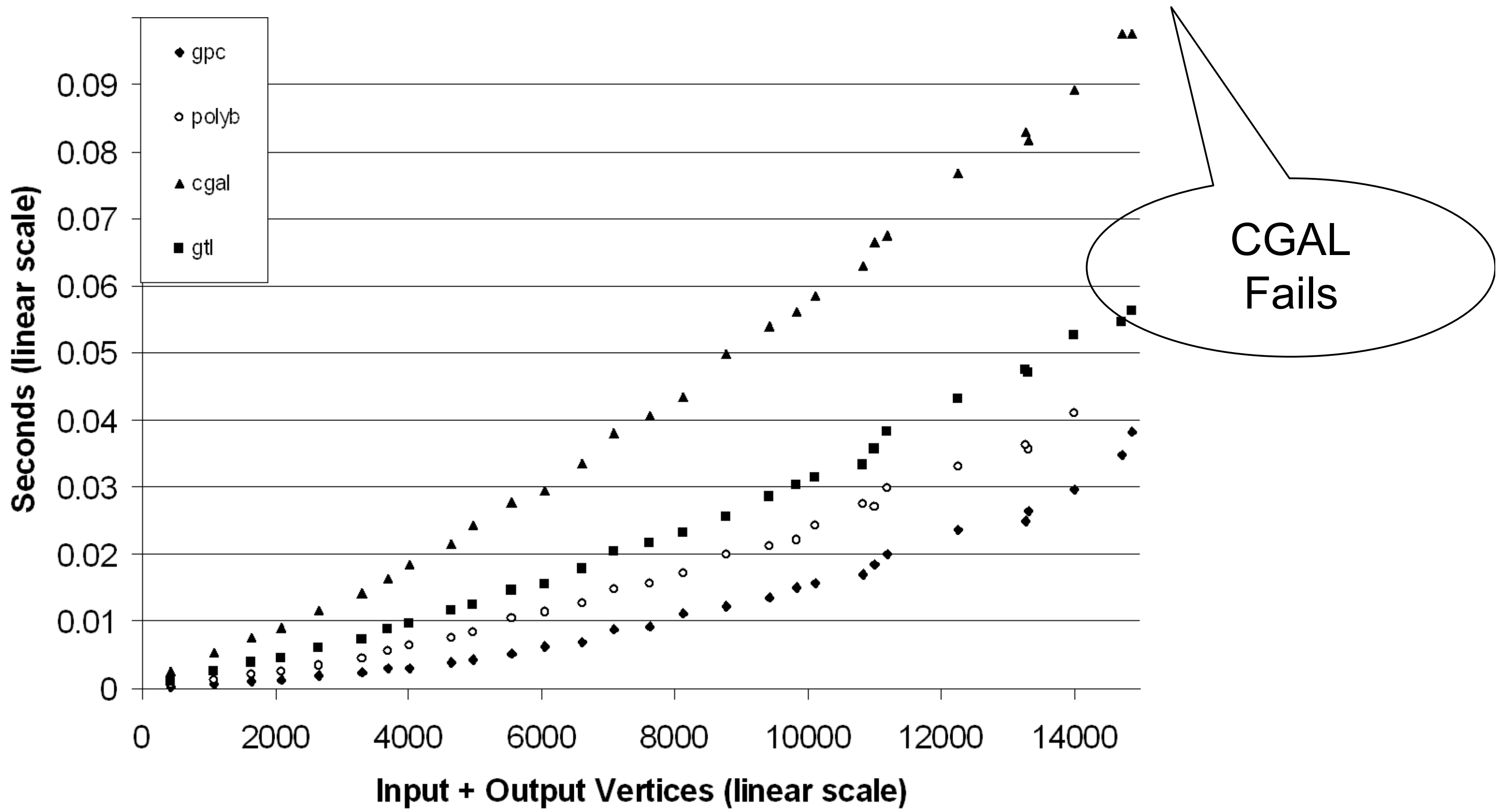
- Runtime for intersection operation

Small Arbitrary-angle Input Benchmark Comparison



- Runtime for intersection operation

Small Arbitrary-angle Input Benchmark Comparison



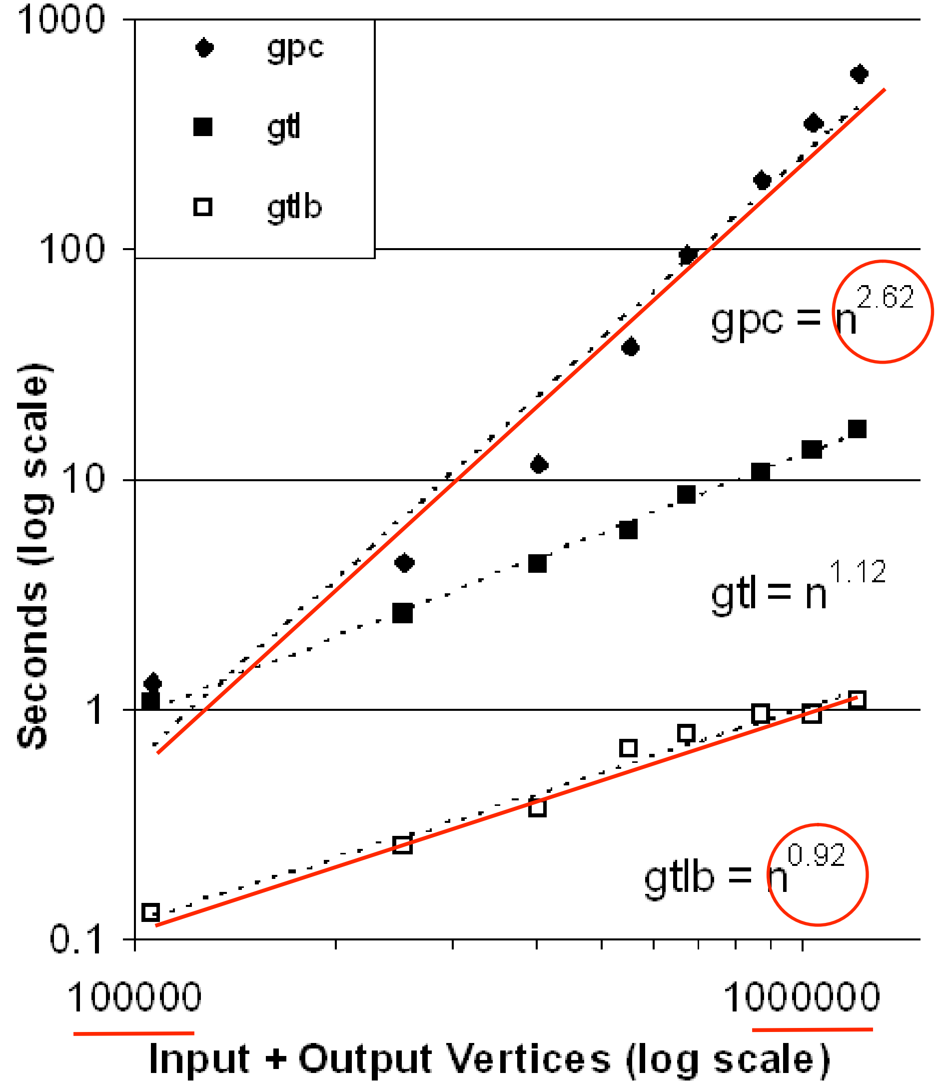
- Runtime for intersection operation

Small Arbitrary-angle Input Benchmark Comparison

- Runtime for intersection operation

Large Scale Arbitrary-angle Performance Comparison

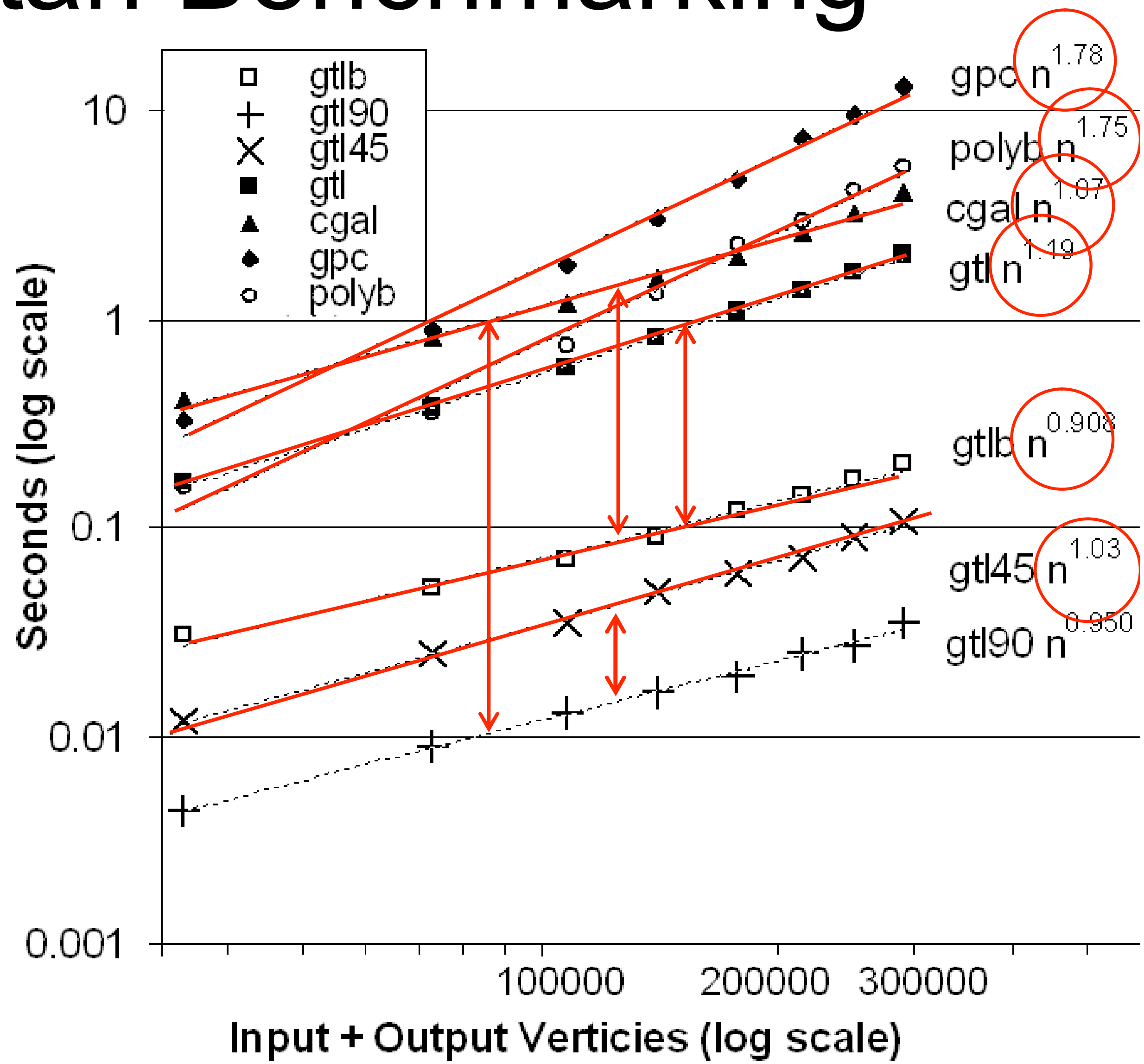
- One to two orders of magnitude larger than previous benchmark
- Though fastest for small inputs, GPC does not scale well
- gtlb excludes line segment intersection
- Core Boolean is n log n, Intel micro-architecture accelerates processing of large vectorss



Manhattan Benchmarking

- 100X performance delta between optimal gtl 90-degree algorithm and general algorithms
- gtl 45-degree Boolean is optimal
- Core arbitrary angle Boolean (gtlb) is optimal
- gtl arbitrary angle Boolean is slightly suboptimal due to line segment intersection
- CGAL is optimal, but has a high constant factor
- GPC and PolyBoolean both scale sub-optimally
- Optimal is:

  near linear O(n log n) runtime



Benchmarking Conclusions about GTL

- GTL arbitrary-angle Booleans is near optimal
- Performance of GTL arbitrary-angle Booleans is middle-of-road for small inputs
- Performance of GTL arbitrary-angle Booleans is best in class for large inputs
- Performance of GTL could be improved by up to 10X with further work on the arbitrary-angle Booleans
- If you have 45-degree or Manhattan polygons gtl provides 50X and 100X performance advantage over cgal

Observations on GPC, CGAL and PolyBoolean

- We found at least two different bugs in PolyBoolean
- We found one bug in CGAL
- GPC and PolyBoolean have very difficult to use C-style APIs
- GPC and PolyBoolean cannot merge multiple overlapping polygons in one step
- GPC and PolyBoolean both have $O(n^{1.5} \log n)$ line segment intersection algorithms (sort all edges that intersect sweep-line at every x)
- PolyBoolean has O(n * m * k) algorithm to determine which polygons contain which holes (n polygons, m holes, k points per polygon), which is O(n^2) in the worst case
- CGAL requires that overlapping polygons be merged before being an input to a Boolean, but can do that itself

Observations About Preconditions

- CGAL throws an “Precondition Violated” exception if an input polygon is self intersecting/overlapping or has “closed” semantic at last vertex
- PolyBoolean returns a “bad input polygon” error code if an input polygon is self intersecting/overlapping has zero area or is a hole with no enclosing polygon
- Both PolyBoolean and CGAL inform the user the input is bad when a bug in their algorithms leads to a fatal error
- GPC produces garbage output when input polygons are self intersecting/overlapping
- GTL has no preconditions and produces correct output in all cases

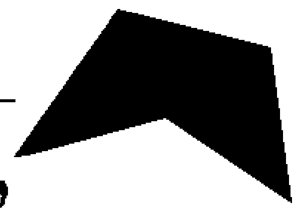
Generic Sweep-line Algorithm

- Sweep-line algorithms for polygon clipping is a tradition that goes back to 1979
- Sweep-line is the best known method for line segment intersection
- GTL implements different sweep lines for Manhattan, 45-degree and general case
- GTL Booleans sweep-lines are parameterized to allow them to perform multiple operations

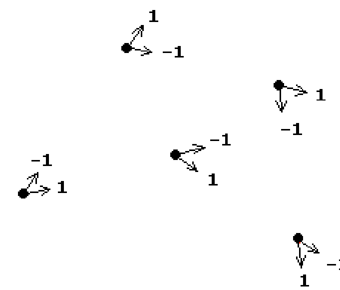
Better Booleans through Calculus

- We use the same algorithm for Manhattan, 45-degree and general polygon Booleans

$$\frac{d}{dx}\frac{d}{dy}$$

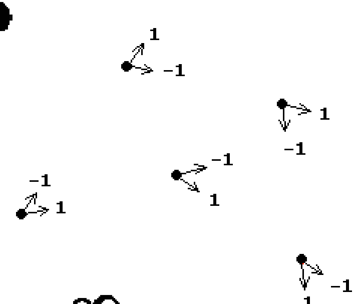


$$=$$

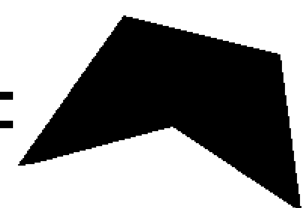


- We will explain how it works in the Manhattan case first, then how we generalize it

$$\int_{x=-\infty}^{\infty}\int_{y=-\infty}^{\infty}$$

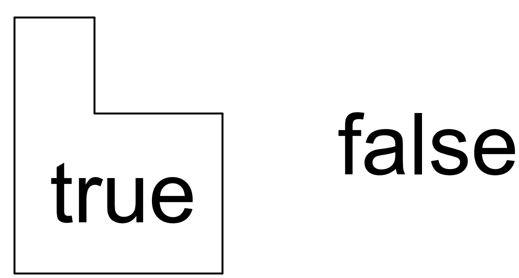


$$=$$



Boolean Polygon Model

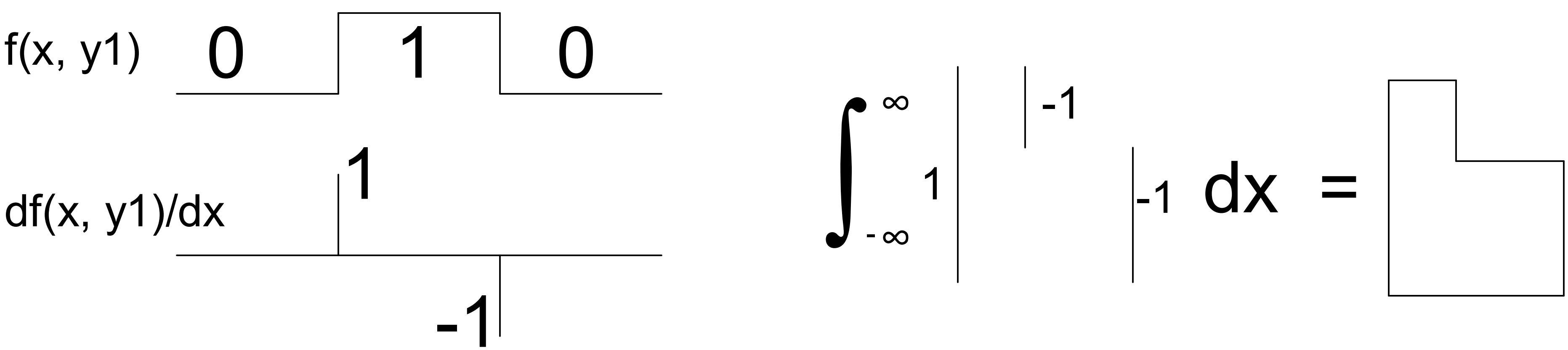
- We define a polygon as a two dimensional Boolean function
  - Function evaluates to true inside the polygon
  - Function evaluates to false outside the polygon



inside_polygon = f(x, y)

Math With Polygon Model

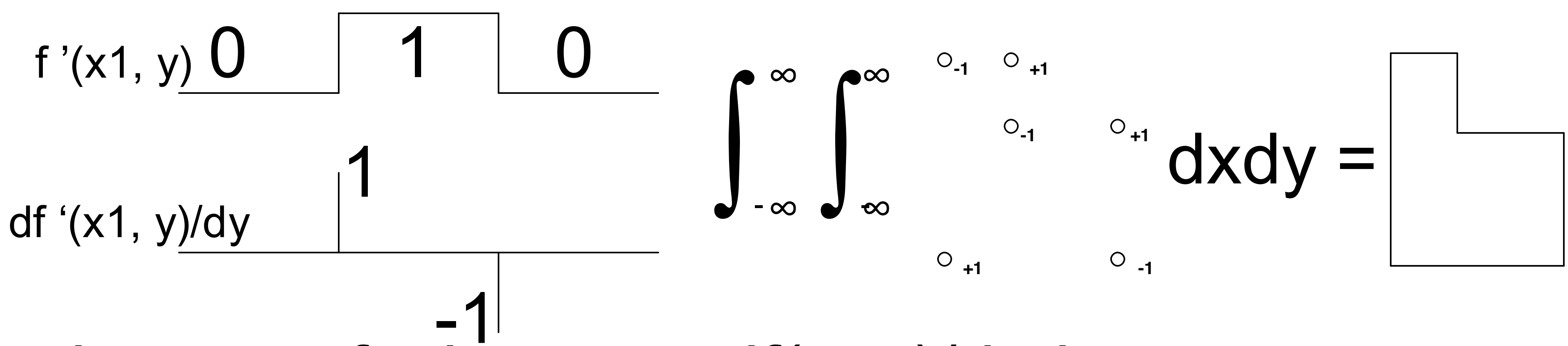
- Because the Polygon is now modeled mathematically…
- We can manipulate it with calculus
- The derivative with respect to x of the polygon function is the change in polygon count as we cross its vertical edges
- In one dimension the polygon looks like a step function at its vertical edges
- Derivative of a step function is an impulse with area of one
- Summing changes in polygon count from left to right (scanline) performs an integration over the df/dx to produce the original polygon



changing_polygon_count = df(x, y)/dx

The Great Thing About Math

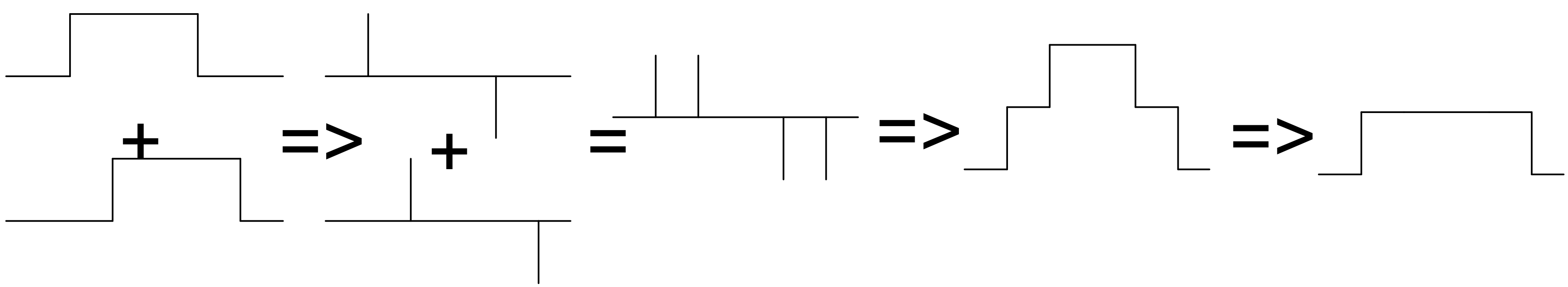
- If it works once, it will work a second time
- The derivative with respect to y of the d/dx of polygon function f is the change in the change in polygon count with respect to x as we enter and leave its vertical edges in the y dimension
- In the y dimension d f/dx (vertical edges) looks like a step function
- Derivative of a step function is an impulse with area of one
- Summing changes in y of changes in x from low to high y integrates the function and produces changes in x (edges) that can be integrated left to right to produce polygons



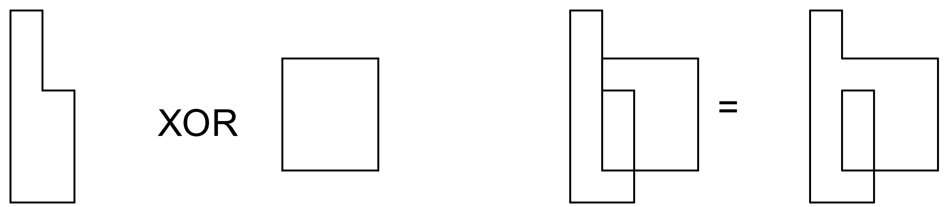
change_of_change = df(x, y)/dxdy

1D Boolean OR Operation Example

- We want a data model for polygons that can provide the input for sweep-line and be constructed from n polygon verticies in O(n log n) time
- If you want to sum two piece-wise linear functions (continuous)
  - you can take the derivative of each (discreet)
  - combine their derivatives in linear time by merging (sum any overlapping values)
  - and then integrate by summing from low to high (in linear time)
- The math is what allows the boolean algorithm to achieve optimal time complexity
  - All we do is sort vertices, but you have to carry the dxdy values along with them so that the meaning of the vertices is retained

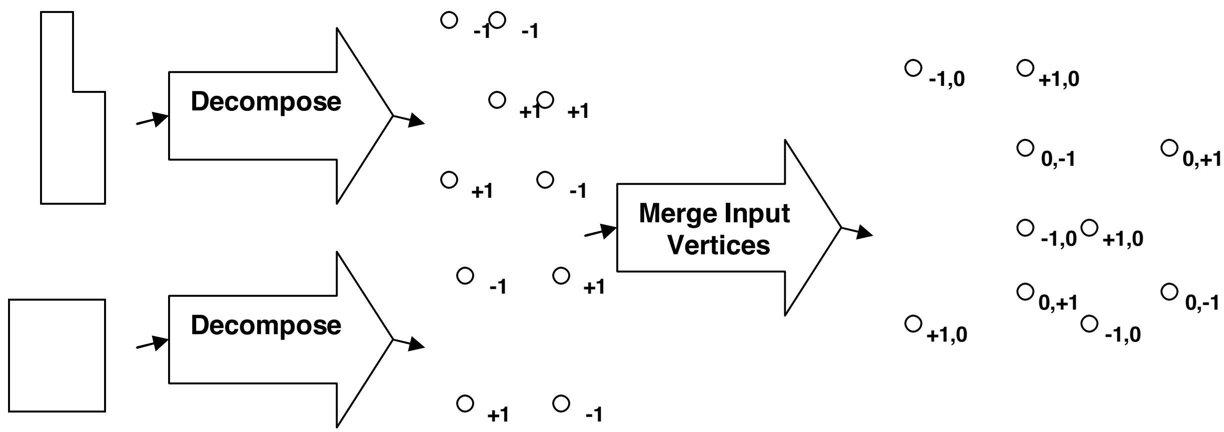


2D, Two Layer Boolean XOR Example



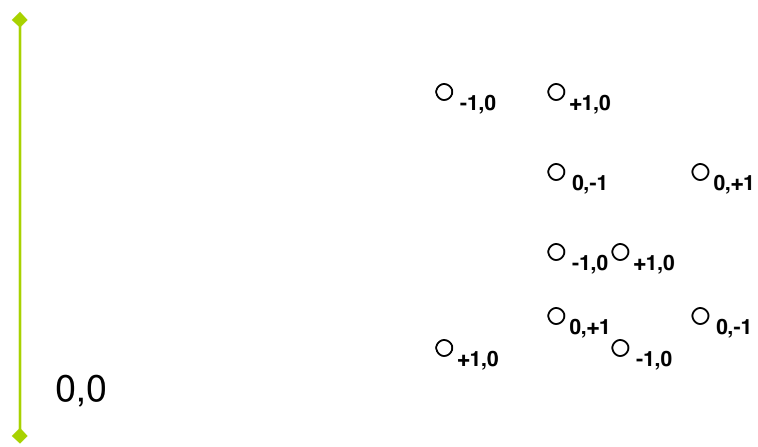
- XOR an L shape with a rectangle

XOR Example



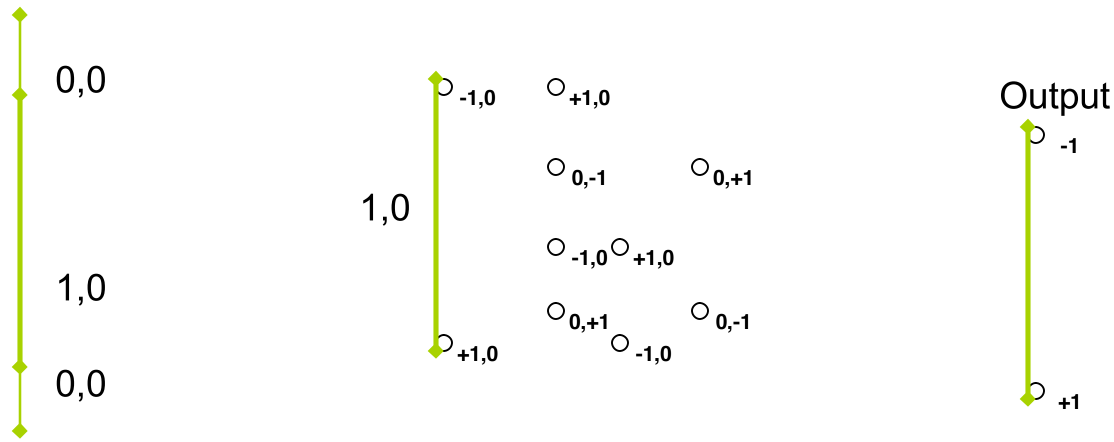
- Preprocess input polygons into a merged, sorted sequence of change on y of change on x of polygon intersection count
- Decomposition is linear, sort is n log n, merge is linear

XOR Example



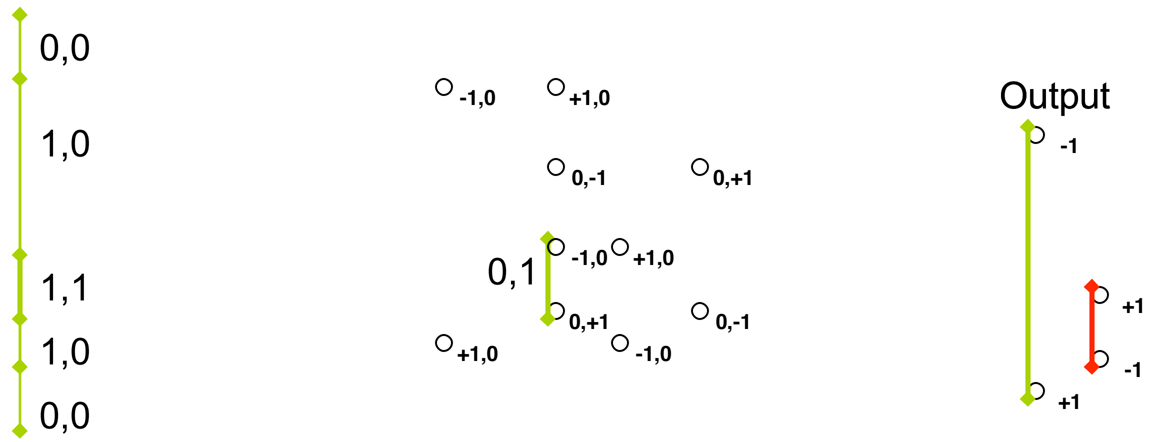
- Sweep-line data structure initialized to a single interval from -infinity to +infinity with intersection count of zero for each input layer

XOR Example



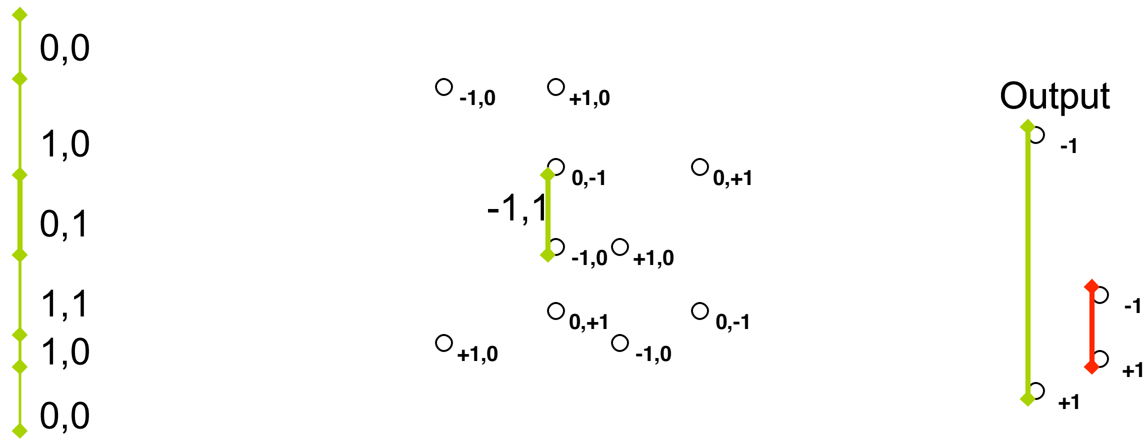
- Intersect first input interval of intersection count change on x against sweep-line data structure of intersection count intervals
- Intersection count changes from zero to one on layer1 on that interval
- 0 xor 0 = false, 1 xor 0 = true, output a left edge because Boolean logic changed from false to true

XOR Example



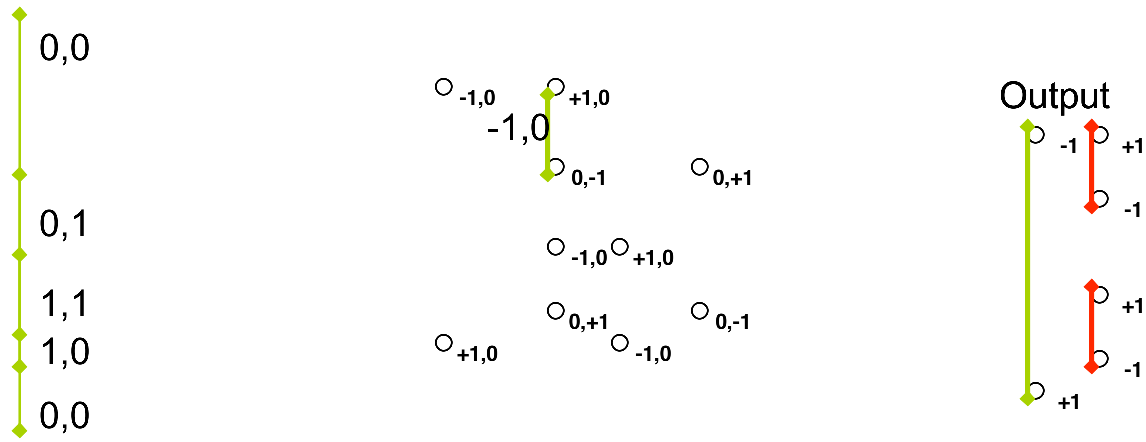
- Intersect second input interval against sweep-line data structure
- Intersection count changes from zero to one for layer2 on that interval
- 1 xor 0 = true, 1 xor 1 = false, so output a right edge because Boolean logic has changed from true to false

XOR Example



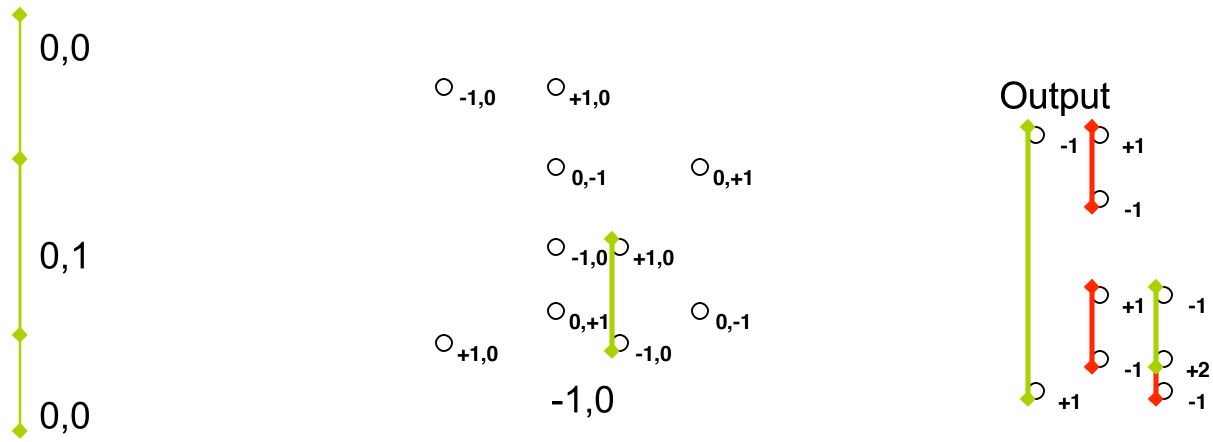
- Intersect third input interval against sweep-line data structure
- Intersection count changes from one to zero for layer1 on that interval
- 1 xor 0 = false, 0 xor 1 = false, so no output

XOR Example



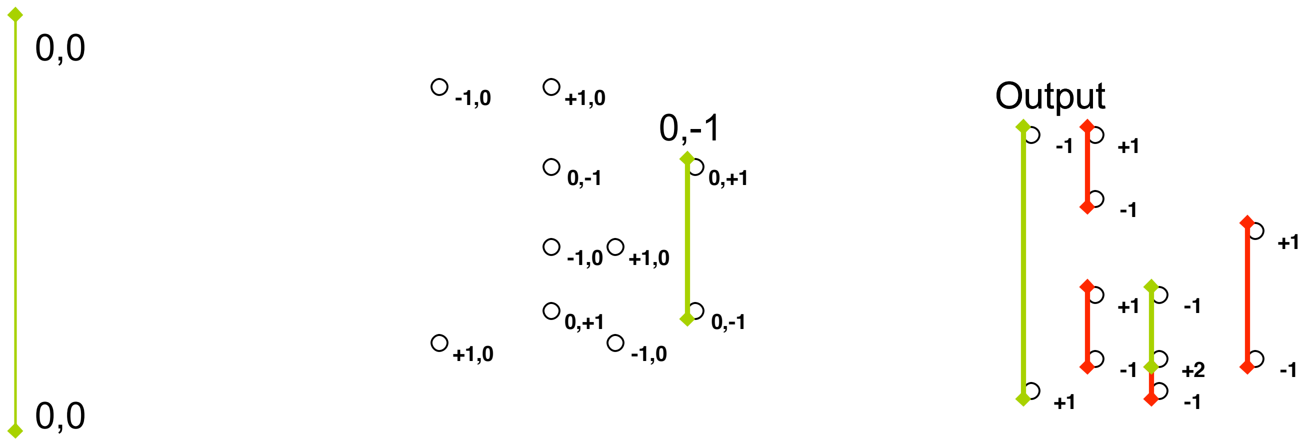
- Intersect fourth input interval against sweep-line data structure
- Intersection count changes from one to zero for layer1 on one interval
- 1 xor 0 = true, 0 xor 0 = false, so output a right edge because Boolean logic has changed from true to false

XOR Example



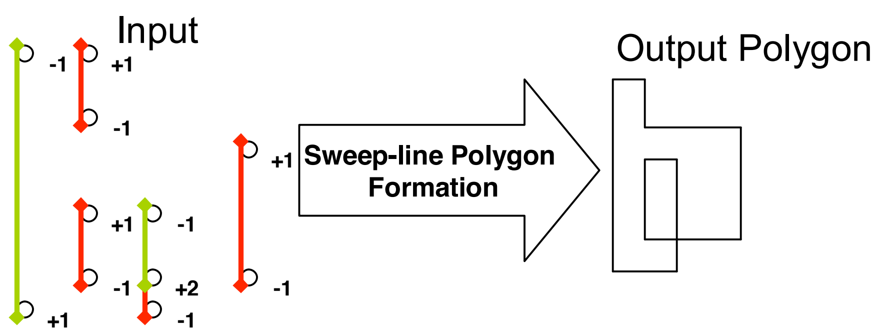
- Intersect fifth input interval against sweep-line data structure
- Intersection count changes from one to zero for layer1 on two intervals
- 1 xor 0 = true, 0 xor 0 = false, so output a right edge for the first interval
- 1 xor 1 = false, 0 xor 1 = true, so output a left edge for the second interval

XOR Example



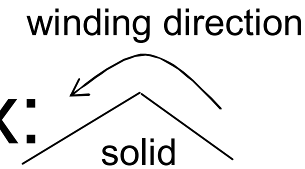
- Intersect sixth input interval against sweep-line data structure
- Intersection count changes from one to zero for layer2 on one interval
- 0 xor 1 = true, 0 xor 0 = false, so output a right edge

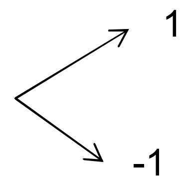
XOR Example

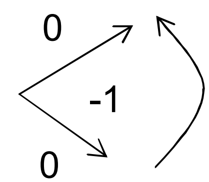


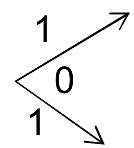
- Sweep-line Polygon Formation produces output polygon
- Could be done in the same pass as the xor
- Leaving it in the derivative form allows direct input to a subsequent Boolean

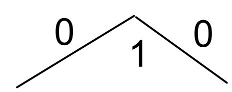
Generalizing The Algorithm

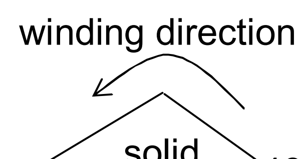
- We want the derivative of this vertex: 

- We apply d/dx and d/dy
- To get a result in terms of θ: 

- We sweep the θ from low to high: 

- As we integrate wrt. y: 

- And finally integrate wrt. x: 

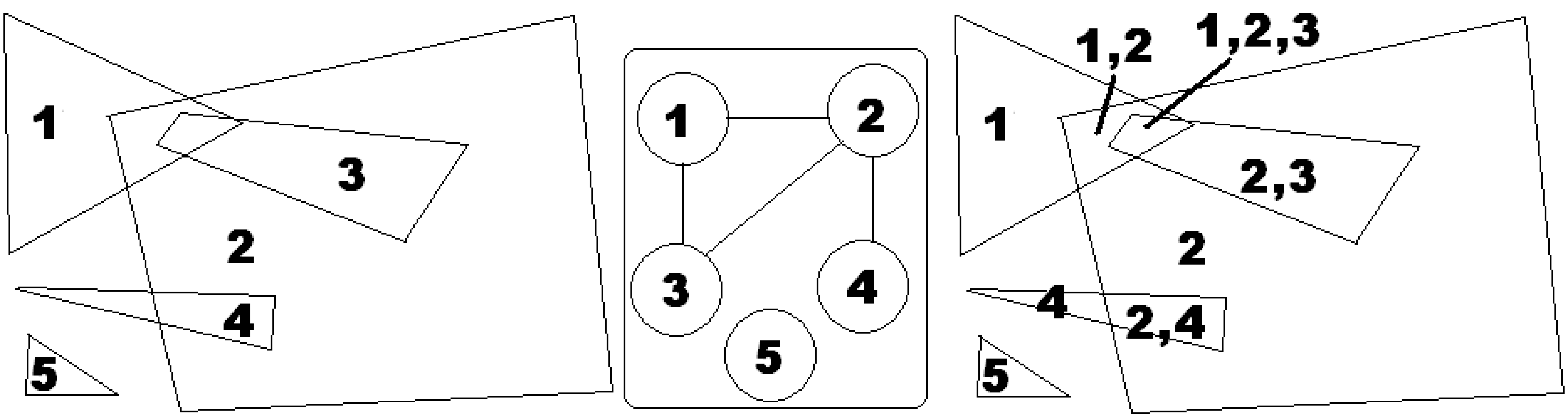
- To which we assign counter clockwise winding and output partial polygon: 

The Algorithm Requires No Preconditions

- The great thing about math is that it's general
- Every special case is just another instance of the general case
- Every case that breaks other algorithms is handled implicitly and correctly

Taking Things One Step Further

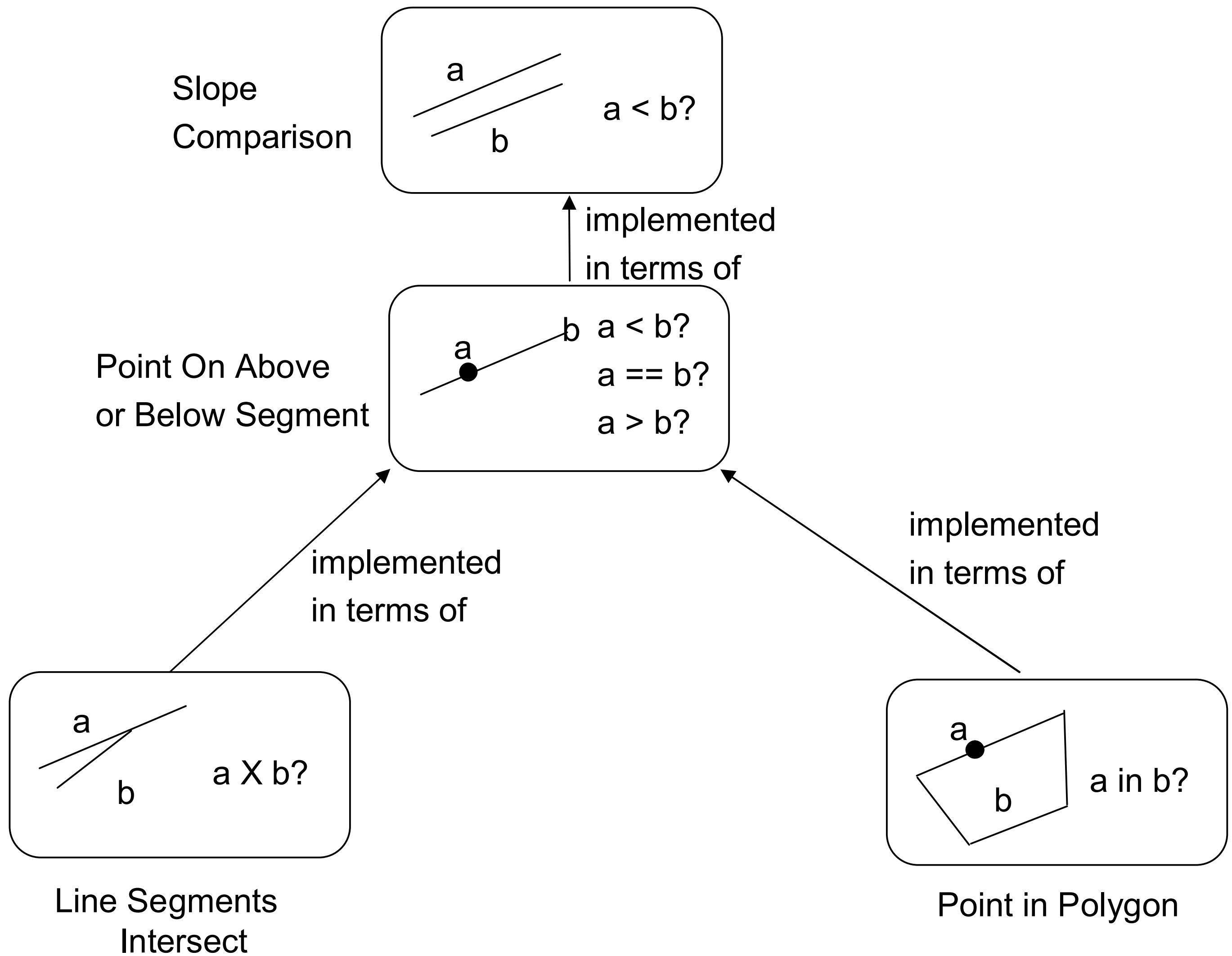
- The Booleans algorithm is parameterized
- N layer operations are implemented with a single pass of the same algorithm
- Is used to provide connectivity extraction / spatial map join and property merge / map overlay



Robustness

- Strategies employed by GTL are provably robust for all cases
  – 100% robust--not just “works for all the cases we’ve tried”
- A firm guarantee of 100% numerical robustness is a very comforting feature
- PolyBoolean fails to find polygons that enclose some holes because its point-in-polygon calculation is not numerically robust

Robust Predicate Primitives



Robust Comparison of Slope

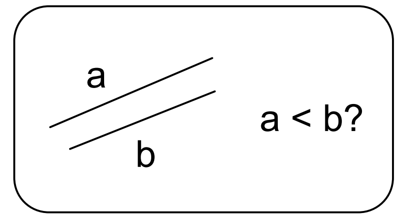
Segment 1: (x11,y11) to (x12, y12)

Segment 2: (x21,y21) to (x22, y22)

Slope1: (y12 - y11) / (x12 - x11)

Slope2: (y22 - y21) / (x22 - x21)

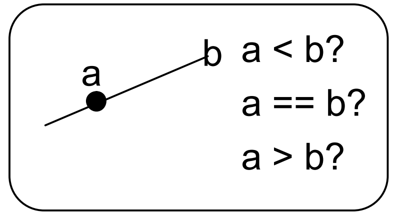
Slope1 < Slope2 iff (y12 - y11)(x22 - x21) < (x12 - x11)(y22 - y21)



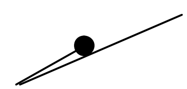
- Cross multiplication avoids integer truncation of division
- Requires 65 bits for signed 32 bit integer coordinates
  - Use long double, multi-precision, SSE quad word, or unsigned 64 bit integer with sign computed separately

Robust Comparison Of Point and Line Segment

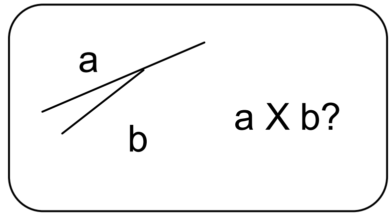
Point On Above or Below Segment



- Make a 2nd segment from one end of the segment to the point
- Compare slopes

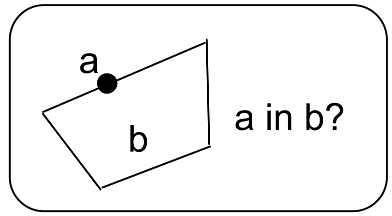


Robust Line Segment Intersection Check



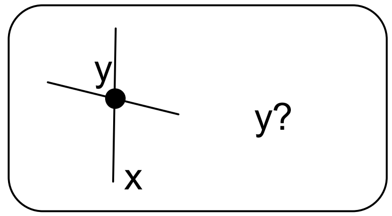
- Compute whether the two ends of each segment are on, above or below the other segment
- Both points of one segment on the same side of the other means no intersection

Robust Point In Polygon Predicate



- For all edges which contain the x value of the point within their x interval
  - Accumulate the sum of such edges the point is above
- The point is inside if the sum is odd

Robust Calculation of Slope Intercept

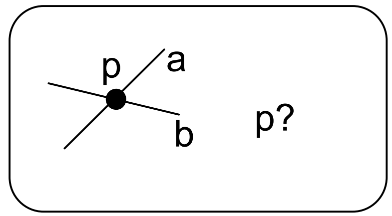


- Apply GMP multi-precision rational and compute exact result
- To compare two slope intercepts

```
//Segment 1: (x11,y11) to (x12, y12)
//Segment 2: (x21,y21) to (x22, y22)
y1 < y2 iff
 (x22 - x21)((x - x11)(y12 - y11) + y11(x12 - x 11)) <
 (x12 - x11)((x - x21)(y22 - y21) + y21(x22 - x 21))
```

(requires 97 bits of precision)

Robust Calculation of Line Segment Intersection Point

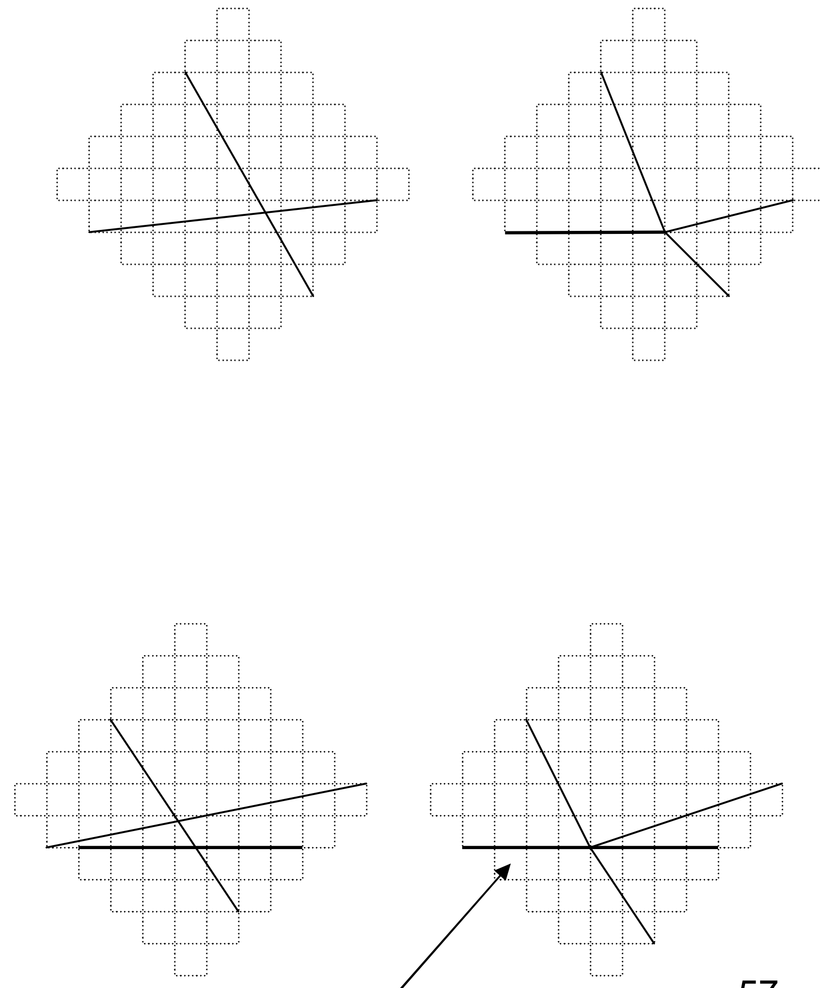


- Apply GMP multi-precision rational and compute exact result.

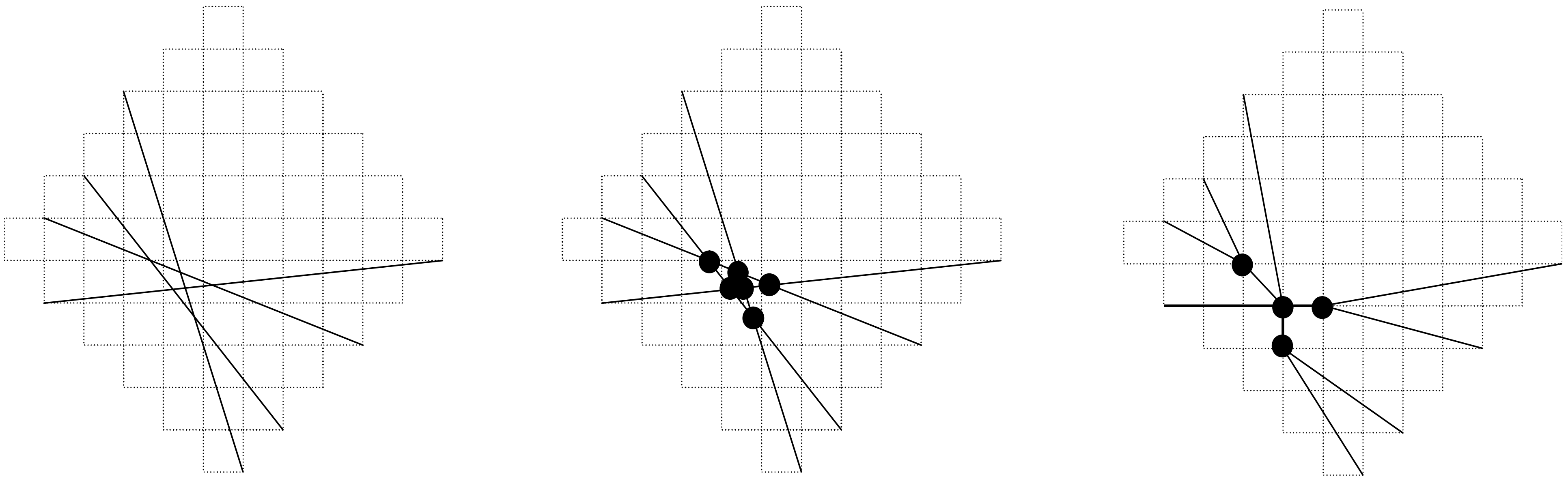
```
//Segment 1: (x11,y11) to (x12, y12)
dx1 = x12 - x11;  dy1 = y12 - y11;
//Segment 2: (x21,y21) to (x22, y22)
dx2 = x22 - x21;  dy2 = y22 - y21;
x = (x11 * dy1 * dx2 – x21 * dy2 * dx1 +
     y21 * dx1 * dx2 - y11 * dx1 * dx2) /
     (dy1 * dx2 - dy2 * dx1);
y = (y11 * dx1 * dy2 - y21 * dx2 * dy1 +
     x21 * dy1 * dy2 - x11 * dy1 * dy2) /
     (dx1 * dy2 - dx2 * dy1);
```

Robust Snapping of Non-Integer Intersection Points to Grid

- Truncate down and to left
- Causes Edges to move slightly
- Moving edges may introduce artifacts
- Non overlapping edges may become parallel and overlap



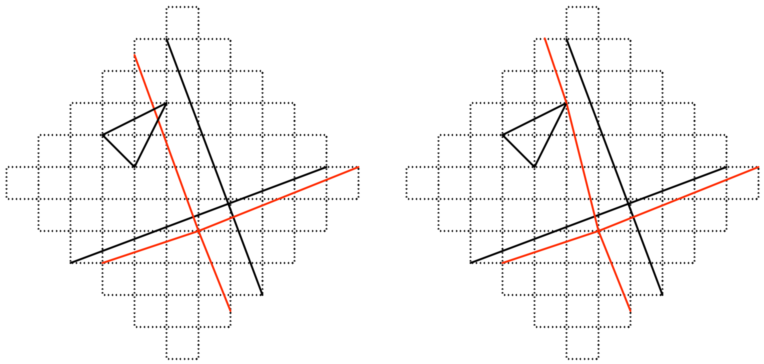
Intersection Clusters



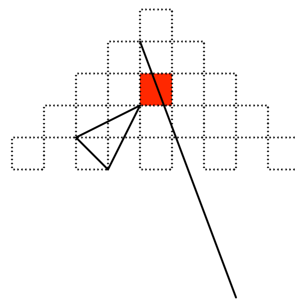
- Multiple intersection points within the same unit grid are merged

Intersections Creating Intersections

- When long edges are moved by integer truncation of intersection point
- Very close geometry may be intersected
- Intersect segments with very close vertices



- Sufficient to check the upper right grid for line segments



Acceptable vs. Unacceptable Artifacts

- An artifact is unacceptable
  - if it causes any line segments to intersect other than at their end points
  - if it causes a closed cycle in the input to become open at the output
- Inserting vertices on line segments and merging vertices are acceptable
- We insert vertices and merge vertices to snap to integer grid robustly

What code that uses GTL looks like

```
void foo(list<CPolygon>& result,
         const list<CPolygon>& a,
         const list<CPolygon>& b) {
  CBoundingBox domainExtent;
  gtl::extents(domainExtent, a);
  result += (b & domainExtent) ^ (a - 10);
}
```

- Two lines of code in the example invoke five different GTL algorithms
- Arguments passed into functions are not GTL data types
- The code is maximally concise, yet easy to read
- Clip `b` to the bounding box of `a`, XOR that with `a` shrunk by ten then merge into `result`
- Details of memory management for intermediate results are abstracted away from the use of algorithms
- Such code is easy to write and easy to maintain

C++ Concepts-based Type System

- GTL allows application data types to be arguments to its API
- You can check if your point type lies inside your polygon type with a call to GTL contains() passing in your point and your polygon

```
gtl::contains(my_polygon, my_point);
```

- This is accomplished by use of a C++ Concepts-based statically polymorphic type system
- This is much more convenient than copying your polygon into a GTL polygon data type first

C++ Traits

- GTL accesses your geometry types through type traits that you must provide
- These traits map your implementation of a geometry object to GTL's concept of how a such geometry behaves

```
template <typename T>
struct point_traits {
typedef T::coordinate_type coordinate_type;
coordinate_type get(const T& p, orientation_2d orient) {
  return p.get(orient);
}
template <typename T>
struct point_mutable_traits {
  void set(const T& p, orientation_2d orient,
           coordinate_type value) {
    p.set(orient, value);
  }
  T construct(coordinate_type x, coordinate_type y) {
    return T(x, y); }
};
```

C++ Concepts Overloading

- GTL functions that expect a polygon check whether the input data type is registered as a polygon and will not instantiate if the check fails
- A different gtl function with the same name can instantiate if the data type turns out to be registered as a rectangle, or a point
- The mechanism for doing this is called substitution failure is not an error (SFINAE)

```
template <typename T> struct is_integer {};
template <>
struct is_integer<int> { typedef int type; };
template <typename T> struct is_float {};
template <>
struct is_float<float> { typedef float type; };

template <typename T>
typename is_int<T>::type foo(T input);
template <typename T>
typename is_float<T>::type foo(T input);
```

foo() would be ambiguous, but both return types cannot be instantiated with the same type. Failure to instantiate the return type is not a syntax error.

Concept Refinement

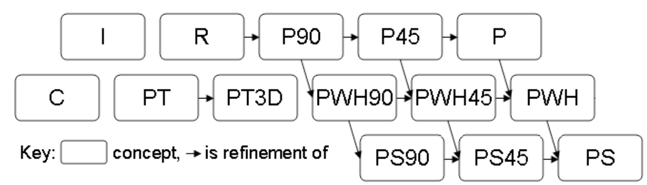
- A rectangle is a refinement of the concept of a polygon
  - A rectangle narrows-down the definition of polygon to four sided, 90-degree angles
- A function that requires read only access to a polygon can always work on a rectangle
  - A polygon is a generalization of a rectangle
- A function that requires write-access to a polygon cannot work on a rectangle
  - A rectangle cannot store a polygon

```
struct polygon_concept {};
struct rectangle_concept {};
template <typename T>
struct is_a_polygon_concept{};
template <> struct is_a_polygon_concept<rectangle_concept> {
  typedef gtl_yes type; };
```

GTL Refinement Relationships

- GTL assign() function
  - copies data between objects of the same conceptual type
  - copies data from a refinement to a more general conceptual type
  - instantiates for each of the 49 legal combinations
  - requires only one overload definition per concept type
  - each overload protected by SFINAE concept check

Concept	Abbreviation
coordinate_concept	C
interval_concept	I
point_concept	PT
point_3d_concept	PT3D
rectangle_concept	R
polygon_90_concept	P90
polygon_90_with_holes_concept	PWH90
polygon_45_concept	P45
polygon_45_with_holes_concept	PWH45
polygon_concept	P
polygon_with_holes_concept	PWH
polygon_90_set_concept	PS90
polygon_45_set_concept	PS45
polygon_set_concept	PS



Concept Casting

- A Manhattan polygon is a refinement of a general polygon
- Given a general polygon and the certainly that it contains only Manhattan data
  - GTL view_as<polygon_90_concept>() can allow that polygon to be legally passed to functions expecting a Manhattan polygon
- This is useful when general objects are used by applications to model several specific kinds of data

Booleans Operator Syntax

- GTL overloads the C++ bit-wise logical operators &|^ and the subtraction operator -
- They perform Boolean AND, OR, XOR and AND-NOT (SUBTRACT)
- They work with any polygons, rectangles, vectors or lists of polygons or rectangles and the GTL polygon-set data types

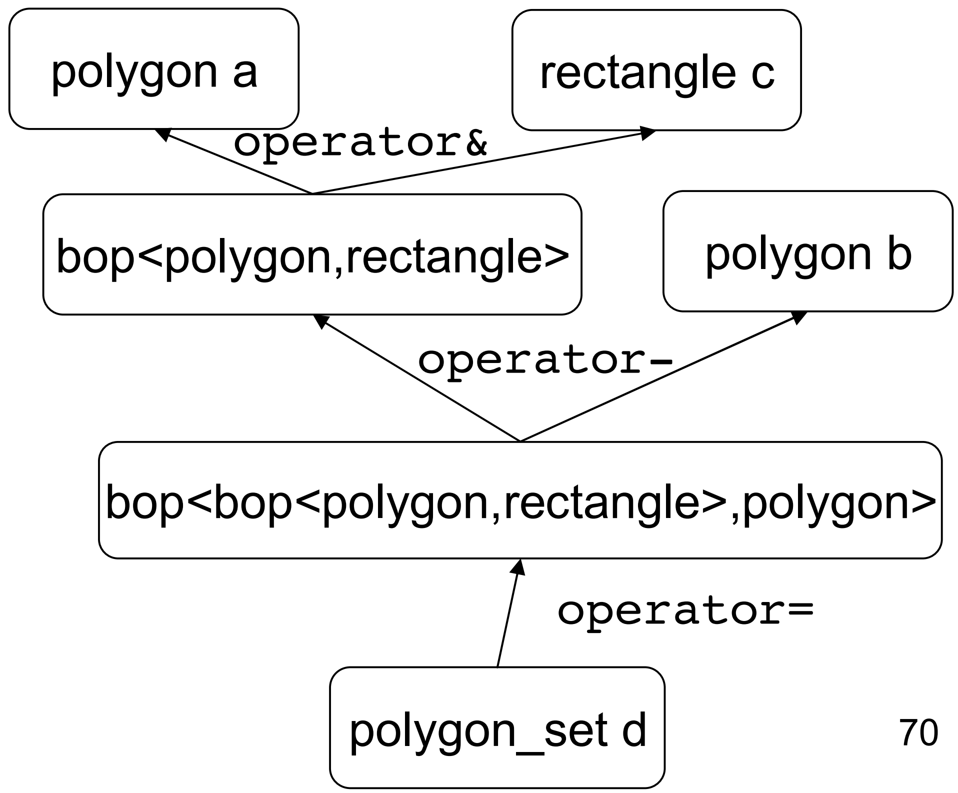
GTL Booleans Operator Templates

- C++ requires that operators return their result by value
- The return value of a GTL Boolean operator function call is an operator template
- The operator template stores references to the arguments and defers the operation until the result is requested
- In this way the operation is performed after the operator template is returned by the operator function

Operator Templates

```
void clip_and_subtract(polygon_set& d,
           polygon a, polygon b, rectangle c) {
  d = (a & c) - b;
}
```

- When chaining operator templates they cache references to each other and build an expression tree
- When the final result is requested the expression is evaluated and the result is produced
- This avoids unnecessary copying of intermediate results



MSVC SFINAE limitation

- SFINAE works in MSVC for the simple cases
- Order of template instantiation in MSVC depends on type of template
  - compile time constant vs. by type
- Substitution failure of a nested template is an error in MSVC
- The only way to get reliable SFINAE behavior out of MSVC is to use enable_if with compile time logic expressions
- It took two weeks of work to port the code from EDG/gcc compatibility to MSVC

EDG SFINAE Bug

- An unnamed enum type cannot be referred to in the template definition when instantiating a template on that type
- STL uses unnamed enum types with arithmetic operators
- Substituion of my generic operators for the unnamed STL enum types should fail
- A bug in older versions of EDG frontend produces a syntax error instead of SFINAE if the template references it in the definition
- Currently fixed in the version of EGD used by the new icc11

EDG Bug Workaround

- If substitution of a nested template parameter fails before EDG tries to instantiate the template that would refer to the unnamed enum type no syntax error is generated
- EDG supports nested SFINAE, of course
- I provide an intermediate meta-function with preprocessor macros in its definition that results in nested SFINAE except when compiled by MSVC to work around both bugs

```
template <typename T> struct gtl_if {
#ifdef WIN32
  typedef gtl_no type;
#endif
};
template <> struct gtl_if<gtl_yes> { typedef gtl_yes type; };
```