Geometry Template Library for STL-like 2D Operations

Lucanus Simonson
Intel Corporation
2200 Mission College Blvd.
Santa Clara, CA 95054-1549
1 (408) 765-8080
lucanus.j.simonson@intel.com

Gyuszi Suto
Intel Corporation
2200 Mission College Blvd.
Santa Clara, CA 95054-1549
1 (408) 765-8080
gyuszi.suto@intel.com

ABSTRACT
There is a proliferation of geometric algorithms and data types with no existing mechanism to unify geometric programming in C++. The geometry template library (GTL) provides geometry concepts and concept mapping through traits as well as algorithms parameterized by conceptual geometric data type to provide a unified library of fundamental geometric algorithms that is interoperable with existing geometric data types without the need for data copy conversion. Specific concepts and algorithms provided in GTL focus on high performance/capacity 2D polygon manipulation, especially polygon clipping. The application-programming interface (API) for invoking algorithms is based on overloading of generic free functions by concepts. Overloaded generic operator syntax for polygon clipping Booleans (see Figure 1) and the supporting operator templates are provided to make the API highly productive and abstract away the details of algorithms from their usage. The library was implemented in Intel Corporation to converge the programming of geometric manipulations in C++ while providing best in class runtime and memory performance for Booleans operations. This paper discusses the specific needs of generic geometry programming and how those needs are met by the concepts-based type system that makes the generic API possible.

Categories and Subject Descriptors
I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling – *Curve, surface, solid and object representations.*

General Terms
Algorithms, Performance, Design, Reliability, Standardization.

Keywords
C++ concepts, geometry, polygon clipping, data modeling, library design.

1. INTRODUCTION
1.1 Problem Statement and Motivation
Traditional object-oriented design leads to type systems that employ a common base class to integrate new types into an existing system. This leads to monolithic software design with shared dependencies on the base class, creating barriers of incompatibility between different software systems when they cannot share a common base class. Integrating library functionality into such a software system typically requires wrapping the library API with functions that perform implicit data copy conversion or that data copy conversion be performed as an explicit step before the library API can be used. This leads to code and memory bloat in applications and to libraries that are hard to use. The problem presents itself clearly in geometry. An application will model objects that have a geometric nature as well as application-specific attributes such as names, weights, colors etc. These object models are geometric entities, but are typically not syntactically compatible with a geometric library unless there is code coupling through some base geometric class. Often an unnecessary copy conversion of data model to geometric object is required to satisfy syntactic requirements of a geometry library API. For example, to use CGAL [8] library algorithms a polygon that depends on CGAL header files must be declared and data copied into it before that data can be passed into the algorithm. Eliminating this artificial incompatibility and allowing a geometry library's API to be used directly and non-intrusively in an application with its native object model is the design goal for GTL's interfaces. This allows application level programming tasks to benefit from the thoughtful design of a productive and intuitive set of library APIs.

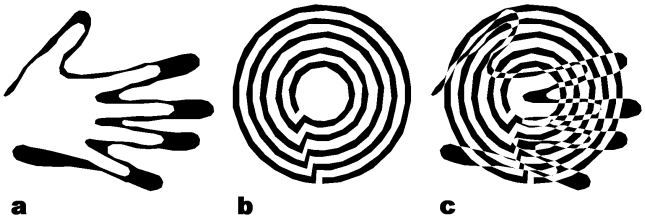

Figure 1. Booleans example: disjoint-union (a xor b) is c.

1.2 C++ Concepts Address the Problem
Computational geometry is the ideal domain to apply C++ Concepts [10] based library design. At a conceptual level, geometric objects are universally understood. Existing geometry codes coming from different sources are not inter-compatible due largely to syntactic rather than semantic differences. These syntactic differences can be easily resolved by concept mapping through C++ traits. However, there are minor semantic differences that need to be comprehended in order to implement a truly generic geometry concept.

Boostcon'09, May 3–8, 2009, Aspen, Colorado, USA.

A generic geometry library must also parameterize the numerical coordinate data type while at the same time providing numerically robust arithmetic. Doing this requires more than simply making the coordinate data type a template parameter everywhere, but also looking up parameterized coordinate type conversions, handling the differences between integer and floating point programming, and making sparing use of high-precision data types to allow robust calculations. The generic geometry library must define a type system of concepts that includes a coordinate concept and provide an API based upon it that is both intuitive and productive to use yet maintainable and easily extensible.

We will compare and contrast different proposed approaches to generic geometry type systems in Section 2. Section 3 will present the approach used in GTL to implement a geometry concepts API and explain its advantages over other proposals. Operator templates and the details of the operator based API for polygon set operations (intersection, union, etc.) will be presented in Section 4. In Section 5 we will present a generic sweep-line algorithmic framework, explain the principles behind our implementation of sweep-line and how they are reflected in the requirements placed on its template parameters. Numerical issues faced and solutions for numerical robustness problems implemented in our library will be discussed in Section 6. A performance comparison with several open source computational geometry libraries will be presented in Section 7 and closing remarks in Section 8.

2. Generic Geometry Approaches

There are several well known generic programming techniques that are applicable to the design of a generic geometry library API. The most important is C++ traits, which provides the necessary abstraction between the interface of a geometry object and its use in generic code. In combination with traits, other generic programming techniques have been proposed in the design of generic geometry libraries including: tag dispatching, static asserts and substitution-failure-is-not-an-error (SFINAE) template function overloading.

2.1 Free Functions and Traits

The conventional way to implement a generic API is with template functions declared within a namespace. This allows arbitrary objects to be passed directly into template functions and accessed through their associated traits. There is, however, one problem with this approach when used for computational geometry. How to specify what kind of geometric entity a given template parameter represents? The simplest solution is to name the function such that it is clear what kind of geometry it expects. This prevents generic function name collisions and documents what expectation is placed upon the template parameter. However, it leads to overly long function names, particularly if two or more kinds of geometry are arguments of a function. Such generic functions do not lend themselves to generic programming at a higher level of abstraction. Consider the `center(T)` function. If we name it variously `rectangle_center(T)` for the rectangle case and `polygon_center(T)` for polygons we cannot abstract away what kind of geometry we are dealing with when working with their center points. As an example, a generic library function that computes the centroid of an iterator range over geometry objects using their center points weighted by their area would require two definitions that differ only by the names of functions that compute center and area. As we add triangle and polygon-with-holes and circle to the generic library, the drawback of not being able to abstract away what kind of geometry is being worked with in library code as well as user code becomes painfully obvious.

2.2 Concepts and Static Assert

A static assert generates a syntax error whenever a concept check fails. A boost geometry library proposal from Brandon Kohn [5] employs `boost::static_assert` on top of generic free functions and traits. This approach to C++ concepts when applied to computational geometry improves the API primarily by producing more intelligible syntax errors when the wrong type is passed into an API. It does not solve the problem of not being able to abstract away the geometric concept itself because it still relies on functions having different names for different concepts to prevent function name collisions.

2.3 Tag Dispatching Based Concepts

A series of boost geometry library proposals from Barend Gehrels and Bruno Lalande [2] have culminated into a tag dispatching based API where a generic free function that looks up tag types for the objects passed in to disambiguate otherwise identical dispatch functions. These dispatch functions are wrapped in structs to emulate partial specialization of dispatch functions by specializing the struct. Partial specialization of template functions is not legal C++ syntax under the C++03 standard.

```
namespace dispatch {
  template <typename TAG1, typename TAG2,
            typename G1, typename G2>
  struct distance {};

  template <typename P1, typename P2>
  struct distance<point_tag, point_tag, P1, P2> {
    static typename distance_result<P1, P2>::type
    calculate(const P1& p1, const P2& p2) {…};
  };

  template <typename P, typename L>
  struct distance<point_tag,linestring_tag,P,L> {
    template<typename S>
    static typename distance_result<P1, P2>::type
    calculate(const P& point, const L& linestr);
}

template <typename G1, typename G2>
typename distance_result<G1, G2>::type
distance(const G1& g1, const G2& g2) {
  return
  dispatch::distance< tag<G1>::type,
    tag<G2>::type, G1, G2>::calculate(g1, g2);
}
```

This approach solves the name collision problem. It allows one `center()` function, for example, to dispatch to different implementations for various rectangle, circle, polygon conceptual types and abstract the concept away when working with objects that share the characteristic that they have a center. However, it is hard to generalize about concepts in a tag dispatching API because concept tags need to be explicit in the declaration of dispatch functions. For all combinations of tags that could satisfy a generic function, a definition of a dispatch function that accepts those tags must be provided. For `center()` this is merely one for each concept, but for a function such as `distance()` it is all pairs and becomes cumbersome once the number of concepts in the

system exceeds just a handful. The number of such dispatch functions needed to implement an API explodes if some other abstraction is not used, such as multi-stage dispatch. Another way to achieve that additional abstraction is inheritance sub-typing of tags, while SFINAE provides a third.

3. GTL's Approach to Generic Geometry

Empty concept `structs` are defined for the purposes of meta-programming in terms of concepts and are analogous to tags used in tag dispatching. Sub-typing relationships between concepts (concept refinements) are implemented by specializing meta-functions that query for such.

```
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; };
```

Even with concepts inherited from each other (for tag dispatching purposes, for instance) such meta-functions would still be convenient for SFINAE checks because inheritance relationships are not easily detected at compile time. The use of `boost::is_base_of` could obviate the need for these meta-functions in GTL.

Traits related to geometry concepts are broken down into mutable and read-only traits structs. A data type that models a concept must provide a specialization for that concept's read-only traits or conform to the default traits definition. It should also do the same for the mutable traits if possible.

GTL interfaces follow a geometric programming style called isotropy, where abstract ideas like orientation and direction are program data. Direction is a parameter to function calls rather than explicitly coded in function names and handled with flow control. The access functions in the traits of a point data type therefore defines one `get()` function that accepts a parameter for horizontal or vertical axis component rather than separate `x()` and `y()` access functions.

```
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); }
};
```

A data type that models a refinement of a concept will automatically have read only traits instantiate for the more general concept based upon the traits of the refinement that data type models. The programmer need only provide concept mapping traits for the exact concept their object models and it becomes fully integrated into the generic type system.

Concept checking is performed by looking up the concept associated with a given object type by meta-function `geometry_concept<T>` and using that along with pertinent concept refinement relationships through compile time logic to produce a yes or no answer for whether a given function should instantiate for the arguments provided or result in SFINAE behavior in the compiler. This allows generic functions to be overloaded in GTL. The two generic functions `foo()` in the example code below differ only by return type, but are not ambiguous because their return types cannot both be instantiated for the same template argument type. While SFINAE generic function overloading is quite powerful and flexible, the compiler support for it is currently inconsistent, requiring significant effort and knowledge of compiler idiosyncrasies and their implications in order to produce portable code.

```
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);
```

3.1 Geometry Concepts Provided by GTL

GTL provides geometry concepts that are required to support planar polygon manipulation. A summary of these concepts can be found in Table 1.

Table 1. GTL Concepts

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

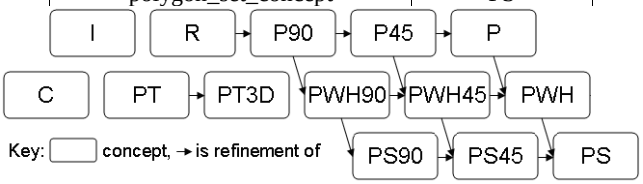

Figure 2. GTL Concept Refinement Diagram

Concept refinement relationships in GTL are shown in Figure 2, with concepts labeled by the abbreviations listed in Table 1. GTL provides algorithms that have been optimized for Manhattan and 45-degree VLSI layout data, and concepts specific to these restricted geometries are named with 90 and 45.

A polygon set in our terminology is any object that is suitable for an argument to a polygon set operation (intersection, union, disjoint union, etc.) A vector of polygons is a natural and convenient way to define such an object. Vectors and lists of objects that model polygon and rectangle concepts are automatically models of polygon sets concepts. A user can define the traits for their polygon data type, register it as a `polygon_concept` type by specializing `geometry_concept<T>` and immediately begin using vectors of those polygons as arguments to GTL APIs that expect objects that model `polygon_set_concept`. GTL also provides data structures for polygon set objects that store the internal representation suitable for consumption by the Booleans algorithms.

3.2 Generic Functions Provided

It is very important to make use of the concept refinement definition of parent concept traits with child concept objects to allow a complete and symmetric library of generic functions to be implemented in a manageable amount of code. O(n) generic functions defined for O(m) conceptual types can allow O(n * m) function instantiations that all operate on distinct conceptual types. A good example of this is the `assign()` function that copies the second argument to the first and is provided in lieu of a generic free assignment operator, which is not legal C++ syntax. The `assign()` function can be called on any pair of objects where the second is the same conceptual type as the first or a refinement of the first conceptual type. GTL allows roughly fifty, functionally correct and semantically sensible, instantiations of `assign()` that accept distinct pairs of conceptual types. There is, however, only one SFINAE overload of the generic assign function for each of thirteen conceptual types. No nonsensical combination of concepts passed to `assign()` is allowed to compile and the syntax error generated is simply "no function assign that accepts the arguments..."

The `assign()` function alone turns GTL into a Rosetta-stone of geometry data type conversions, but the library also provides a great many other useful functions such as area, perimeter, contains, distance, extents etc. Because of the extensible design, it is very feasible to add new functions and concepts over time that work well with the existing functions and concepts.

3.3 Bending the Rules with view_as

Sometimes use of GTL APIs with given types would be illegal because the of a conceptual type mismatch, yet the programmer knows that some invariant is true at runtime that the compiler cannot know at compile time. For example, that a polygon is a rectangle, or degenerate. In such cases, the programmer might want to view an object of a given conceptual type as if it were a refinement of that conceptual type. In such cases the programmer can concept-cast the object to the refined concept type with a `view_as` function call. A call to `view_as` provides read only access to the object through the traits associated with the object. For example, some algorithms may be cheaper to apply on concepts that place restrictions on the geometry data through refinement because they can safely assume certain invariants. It is much faster to compute whether a rectangle fully contains a polygon than it is to compute whether a polygon fully contains a polygon. Rather than construct a rectangle from the polygon we can simply view the polygon as a rectangle if we know that to be the case at runtime.

```
if(is_rectilinear(polygon) &&
   size(polygon) == 4) {
  //polygon must be a rectangle
  //use cheaper O(n) algorithm
  return contains(view_as<
    rectangle_concept>(polygon), polygon2);
} else {
  //use O(n log n) Booleans-based algorithm
  return contains(polygon, polygon2);
}
```

The ability to perform concept casting, concept refinement and overload generic functions by concept type results in a complete C++ concepts-based type system.

4. Booleans Operator Templates

The Booleans algorithms are the core algorithmic capability provided by GTL. An example of a Boolean XOR operation on two polygons is shown in Figure 1. The geometry concepts and concept based object model are focused on providing mechanisms for getting data in and out of the core Booleans in the format of the user's choosing. This enables the user to directly make use of the API provided by GTL for invoking these algorithms on their own data types. This novel ability to make use of library APIs with application data types motivates us to provide the most productive, intuitive, concise and readable API possible. We overload the C++ bit-wise Boolean arithmetic operators to perform geometric Boolean operations because it is immediately intuitive, maximally concise, highly readable and productive for the user to apply.

4.1 Supported Operators

A Boolean operator function call is allowed by the library for any two objects that model a geometry concept for which an area function call makes sense. These include the `operator&` for intersection, `operator|` for union, `operator^` for disjoint-union and `operator-` for the and-not/subtract Boolean operation. Self-assignment versions of these operators are provided for left hand side objects that model the mutable polygon set concepts, which are suitable to store the result of a Boolean. Also supported for such objects are `operator+`/`operator-` when the right hand side is a numeric for inflate/deflate, known as offsetting or buffering operations. There is no complement operation because the ability to represent geometries of infinite extent is not expected of application geometry types. Nor is such an operation truly needed when `object ^ rectangle` with a suitably large rectangle is equivalent for practical purposes.

4.2 Operator Templates Definition

To avoid the unnecessary copying of potentially large data structures as the return value of an operator function call that must return its result by value, the return value of GTL Boolean operators is an operator template. The operator template caches references to the operator arguments and allocates storage for the result of the operation, which remains empty initially, allowing the copy of the operator template to be lightweight when it is returned by value. The operator template lazily performs the Boolean operation, storing the output only when first requested.

Operator templates are expected to be temporaries when operators are chained. For instance `(a + b) - c` produces an operator template as the result of `a + b`, passes that into `operator-` and another operator template is returned by `operator-`. Only later when the result of that `operator-` is requested will both the Booleans be performed as the operator templates recursively perform lazy evaluation of the expression. Because the user is not expected to refer to the operator templates by type, but instead use them only as temporaries, there is little danger of the arguments going out of scope before the expression is evaluated.

4.3 Exemplary User Code

The combination of operator templates with the C++ concepts based type system leads to the ability to write exemplary user code using the library. For instance, in an application that defines its own `CBoundingBox` and `CPolyon`, the following GTL based code snippet becomes possible:

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

The application of five GTL library algorithms is accomplished in only two lines of code while the design intent of the code is clear and easy to read. This is with application rather than library data types and no performance is sacrificed for data copy to satisfy the syntactic requirements of library interfaces or the operator semantics of C++ that require return by value. This abstracts away the low-level details of the algorithms and allows the user to program at a higher level of abstraction while at the same time preserving the optimality of the code produced.

5. Generic Sweep-line for Booleans

A common way to implement Booleans is to first intersect polygon edges with an algorithm such as Bentley Ottmann [1]. After line segment intersection, new vertices are commonly introduced on edges where intersections were identified along with crosslinks that stitch the input polygons together into a graph data structure. The graph data structure is then traversed and a rules-based algorithm ensures that interior edges are not traversed. Traversing exterior edges yields closed polygons. [12] This traditional algorithm has several problems. The graph data structure is expensive to construct, expensive to store and expensive to traverse. When the graph is traversed to output polygons the winding direction can be used to identify holes, but no information stored within the graph helps to associate those holes to the outer polygons, requiring that additional computation be performed if that information is needed. The algorithm leads to complex implementations of rule logic because it requires that degeneracy be handled explicitly with logic, making it challenging to achieve a reliable implementation of the algorithm.

A much better approach to Booleans is the application of sweep-line to identify interior edges. GTL provides a generic sweep-line algorithm framework that is used to implement line segment intersection, Booleans and related algorithms such as physical connectivity extraction.

5.1 Better Booleans through Calculus

Our Booleans algorithm differs from the traditional approaches found in the literature. The algorithm most closely resembles [11] in that it can perform polygon clipping and line segment intersection with a single pass of sweep-line. In our problem formulation we model a polygon as a mathematical function of two variables x and y such that for all x/y points inside the polygon the function returns one, and for all points outside the polygon the function returns zero. This view of a polygon is useful because it allows us to reason about the problem mathematically.

If we consider a mathematical function of two variables, we can apply the partial derivative with respect to each of those variables, which provides the points at which the function value changes and the directions and magnitudes in which it is changing. Because our geometry is piece-wise linear this reduces the two dimensional definition of the polygon function to a collection of zero dimensional quantities at its vertices that are directed impulses with magnitude of positive or negative one.

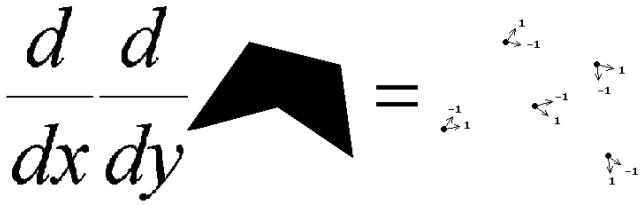

Figure 3. Derivative of a polygon

Integrating with respect to x and y allows us to reconstruct the two dimensional polygon function from these zero dimensional derivative quantities.

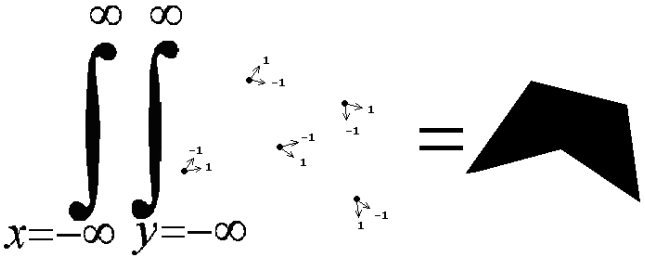

Figure 4. Integrating polygon-derivative reproduces polygon

This integration with respect to x and y in mathematical terms is analogous to programmatically sweeping from left to right and from bottom to top along the sweep-line and accumulating partial sums. Because the polygons are piecewise linear this summation is discreet rather than continuous and is therefore computationally simple. What this mathematical model for calculus of polygons allows us to do is superimpose multiple overlapping polygons by decomposing them into vertex objects that carry data about direction and magnitude of change along the edges that project out of those vertices. Because these vertices are zero-dimensional quantities they can be superimposed simply by placing them together in a collection, trivially sorting them in the order they are to be scanned and summing any that have the same point in common. When scanned, their magnitudes are summed (integrated) onto intervals of the sweep-line data structure. The sweep-line data structure should ideally be a binary tree that

provides amortized log(n) lookup, insertion and removal of these sums, keyed by the lower bound of the interval (which of course changes as the sweep-line moves.) Each such interval on the sweep-line data structure stores the count of the number of polygons the sweep-line is currently intersecting along that interval. Notably, the definition allows for counts to be negative. A union operation is performed by retaining all edges for which the count above is greater than zero and the count below is less than or equal to zero or visa-versa. Vertical edges are a special case because they are parallel to our sweep-line but are easily handled by summing them from bottom to top as we progress along the sweep-line.

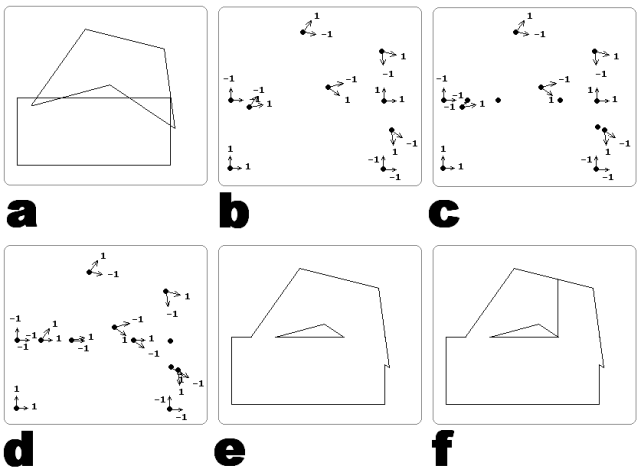

Figure 5. Sequence of Boolean OR (union) operation

The sequence of steps to perform a Boolean OR (union) operation on two polygons is shown in Figure 5. The two input polygons are shown overlapping in Figure 5 a. They are decomposed into their derivative points as shown in Figure 5 b. Line segment intersection points are inserted as shown in Figure 5 c. These intersection points carry no derivative data quantities because no change in direction of edges takes place at intersection points. The result of a pass of sweep-line to remove interior points through integration and output updated derivative quantities is shown in Figure 5 d. Note that it is the same data-format as the input shown in Figure 5 b and is in fact the derivative of the output polygons. This facilitates the chaining together of multiple Booleans operations without the need to convert to and from polygons in between. Note that one point in Figure 5 d. has no derivative vector quantities assigned to it. That point is collinear with the previous and next point in the output polygon and therefore doesn't represent a change in direction of edges. It is retained because preserving the topology of collinear points in the output is a requirement for some meshing algorithms that their input polygons be "linearly consistent." Such collinear points can be trivially discarded if undesired. A final pass of sweep-line can either integrate the output polygon derivative from Figure 5 d to form polygons with holes as shown in Figure 5 e or keyhole out the holes to the outer shells as shown in Figure 5 f. It is possible to perform line segment intersection, interior point removal and form output polygons in a single pass of sweep-line. We break it down into separate steps for convenience. The computation required for interior point removal, updating of derivative quantities and formation of output polygons increases the computational complexity of sweep-line based generalized line segment intersection such as that described by [1] by only a constant factor whether performed as a single pass or separated into multiple passes. The algorithm presented here is therefore optimal because it is well known that polygon clipping is bounded by the complexity of line segment intersection, as can be trivially proven because line segment intersection could be implemented with our polygon-clipping algorithm.

The output polygons can contain holes, and the input polygons can likewise contain holes. Moreover, the output holes can be associated to their outer shells as additional data available in the output or geometrically by keyholing. The output can easily be obtained as the non-overlapping trapezoid decomposition of polygons sliced along the sweep-line orientation similar to [11]. All of these polygonal forms of output are legal inputs to the algorithm, and it is closed both on the polygon domain as well as the polygon derivative domain meaning that it consumes its own output. The other advantage of this algorithm over the traditional previous polygon clipping algorithms is that it correctly handles all degeneracy in inputs implicitly with the same logic path that handles the normal case. Our algorithm reduces the complex logic of categorizing states to simple arithmetic applied while scanning. It is robust to negative polygon counts (holes outside of shells), high order overlaps of intersections and edges, co-linear and duplicate points, zero length edges, zero degree angles and self-intersecting/self-overlapping polygons, all by simply applying the same calculus of summing derivative values that are easily computed by inspecting each polygon vertex. To our knowledge this polygon-derivative data-modeling and algorithm for polygon clipping has not appeared in past literature and is novel.

5.2 Generic Booleans Algorithmic Framework

The scanning of geometry for a Boolean in GTL performs integration with respect to x and y of changes in counts of the number of polygons from left-to-right/bottom-to-top. The sweep-line data structure stores the current count of the number of polygons that overlap intervals of the sweep-line. We employ the stl map for our sweep-line data structure using a similar technique as described in [9] to implement a comparison functor that depends upon the position of the sweep-line. The count data type stored as the value of the map element is a template parameter of the sweep-line algorithm. It is required to be addable, and generally conform to the integral behaviors. An integer is a valid data type for the count and is used to implement unary Boolean operations. A pair of integers can be used to implement binary Boolean operations such as intersection. A map of property value to count can be used to perform sweep-line on an arbitrary number of input geometry "layers" in a single pass. Other template parameters include an output functor, output data structure and of course the coordinate data type.

```
template <typename coordinate_type>
struct boolean_op {
  template <typename count, typename output_f>
  struct sweep_line {
    template <output_c, input_i>
    void scan(output_c& o, input_i b, input_i e);
  };
};
```

The generic algorithm takes care of all the details of intersecting polygon edges and summing counts while the output functor, count data type and output data structure control what is done with that information. In this way, the algorithm can be adapted to perform multiple operations with minimal effort. The seven simple Booleans supported by GTL employ output functors that differ only in the logic they apply to the count data type.

```
//intersect
count[0] > 0 && count[1] > 0;
//union
count[0] > 0 || count[1] > 0;
//self-union
count > 0
//disjoint-union
(count[0] > 0) ^ (count[1] > 0)
//subtract
(count[0] > 0) && !(count[1] > 0)
//self-intersect
(count > 1)
//self-xor
(count % 2)
```

If the logic applied by these output functors to the count results in true on one side of an edge and false on the other then that edge is exterior and appended to the output data structure. If partial polygons are stored as part of the count data structure in the sweep-line tree then the output functor can construct output polygons.

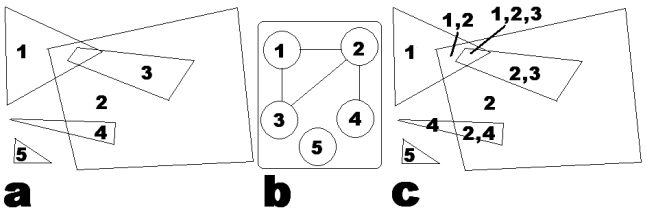

Figure 6. Connectivity Extraction and Property Merge

Also implemented with the generic Booleans framework are property merge and connectivity extraction. By using a map of property to polygon count as the data type for the counts stored on the sweep-line and appropriate output functor and output data structure the connectivity graph of n nodes of polygonal inputs can be computed in a single pass of the algorithm to provide a solution to the spatial overlay join problem. An example of the output of this algorithm for the geometry in Figure 6 a. is shown in Figure 6 b. Similarly, the geometry of all unique combinations of overlap between n polygonal inputs can be computed and output by the property merge output functor to a map of polygon sets keyed by sets of property values. An example of the output of property merge for the geometry in Figure 6 a. is shown in Figure 6 c. The property merge algorithm is a generalization of two input Boolean operations to n inputs to solve the n-layer map overlay problem. The generic algorithm can be easily adapted to implement other sweep-line based algorithms including domain specific algorithm such as capacitance estimation.

5.3 Offsetting/Buffering Operations

In addition to Booleans, GTL also provides the capability to offset polygons by "inflating" or "deflating" them by a given resizing value. Polygons grow to encompass all points within the resizing distance of their original geometry. If the resizing distance is negative, polygons shrink. This implies that circular arcs be inserted at protruding corners when a polygon is resized. Such circular arcs are segmented to make the output polygonal. Other options for handling such corners include inserting a single edge instead of an arc, simply maintaining the original topology or leaving the corner region unfilled. The resizing operations are accomplished by a union operation on the original polygons with a collection of trapezoids constructed from their edges of width equal to the resizing distance and with polygons at the corners generated based on the two adjacent edge trapezoids. An example of the shapes created from the input 45-degree geometry in Figure 7 a is shown in Figure 7 b and the result of the union between those shapes and the original to create the output geometry of an inflate operation is shown in Figure 7 c. Deflate is accomplished by substituting subtraction for union.

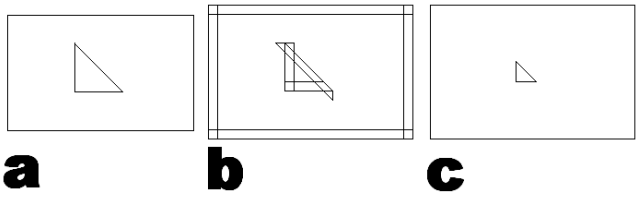

Figure 7. Resize Example: inflate of polygon with hole

6. Numerical Robustness

There are three problems in integer arithmetic that must be overcome to implement generalized line segment intersection for polygon clipping. These are integer overflow, integer truncation of fractional results and integer snapping of intersection points. Overflow and truncation of fractional results makes computing the result of very innocent looking algebraic expressions all but impossible with built-in integer types. The common practice of resorting to floating point arithmetic in these cases is clearly not suitable because the error it introduces is even more problematic.

Intersection points must be snapped to the integer grid at the output of the algorithm. However, snapping the intersection point causes a small lateral movement of the line segments it is inserted on. This movement can cause a line segment to cross to the other side of a vertex than was at the case in the input, introducing new intersections. If these new intersections have not yet been reached by the forward progress of the line segment intersection sweep-line, they might be handled naturally by the algorithm, however, it is just as likely they are introduced prior to the current position of the sweep-line and the algorithm will not have the opportunity to handle them during its forward progress.

A choice about how to handle spurious intersection points introduced by intersection point snapping must be made. It is impossible to both output the idealized "correct" topology of intersected line segments and at the same time output fully intersected line segments with their end points on the integer grid with the property that no two line segments intersect except at their end points. The invariant that output line segments not intersect except at their end points is crucial because this invariant is a requirement of algorithms that would consume the output. Topologically, the important consideration for polygon clipping is that the output edges describe closed figures. Violating this invariant would, at best, cause polygons to be "dropped" during subsequent execution and, at worst, result in undefined behavior.

It is obvious that merging of vertices and the insertion of new vertices are both topological changes that preserve the property of the network that all closed cycles remain closed. These topological changes are allowed to occur as the result of snapping intersection points because we choose to enforce the invariant that no line segments at the output intersect except at their end points.

6.1 Solving Overflow and Truncation

Overflow is easy to handle if the appropriate data types are available. Thirty-two bit can be promoted to sixty-four and sixty-four bit can be promoted to multi-precision integer. However, in generic code it becomes impossible to be explicit about when to cast and what to cast to. The same algorithm might be applied on several different coordinate data types when instantiated with different template parameters. We provide indirect access to the appropriate data types through coordinate traits, a coordinate concept and a meta-function: `high_precision_type<T>`. The coordinate traits allow the lookup of what data type to use for area, difference, Euclidean distance, etc. The coordinate concept is used to provide algorithms that apply these data types correctly to ease the burden of common operations such as computing the absolute distance between two coordinate values in one-dimensional space. The high precision type is used where built-in data types would not be sufficient. It defaults to long double, which is the highest precision built-in data type, but still potentially insufficient. By specializing for a specific coordinate data type such as integer, a multi-precision rational such as the gmp mpq type [3] can be specified. This can be done outside the GTL library itself, making it easy to integrate license encumbered numerical data types with GTL and its boost license without the need for the GTL code itself to depend on license encumbered header files.

Handling integer truncation of fractional results can be done either by applying the high-precision type (preferably a multi-precision rational) or by algebraic manipulation to minimize the need for division and other operations that may produce factional results. Some examples of this are distance comparison, slope comparison and intersection point computations. When comparing the distances between two points it is not necessary to apply the square root operation because that function is monotonic. When comparing slopes we use the cross-product as a substitute for the naïve implementation. This avoids division and produces reliable results when performed with integer data types of sufficient precision. Comparing intersection coordinates can also use the cross product to avoid division because computing the intersection point of two line segments can be algebraically manipulated to require only a single division operation per coordinate, which is performed last.

```
//Segment 1: (x11,y11) to (x12, y12)
//Segment 2: (x21,y21) to (x22, y22)
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);
```

6.2 Solving Spurious Intersections

Non-integer intersection points need to be snapped to the integer grid in the output. We snap each intersection point to the integer grid at the time it is identified. We do this by taking the floor of the fractional value. Integer truncation is platform dependent, but frequently snaps toward zero, which is undesirable because it is not uniformly consistent. Because the integer grid is uniform, the distance a point can be snapped by taking the floor is bounded to a 1x1 unit integer grid region. Our current approach differs from the similar approach described by John Hobby [4] in that he rounds to the nearest integer.

Because the distance a segment can move is bounded, it is predictable. That means that we can predict the distance a segment might move due to a future intersection event and handle any problems that would cause pro-actively in the execution of line segment intersection. There are two types of intersection artifacts created by snapping. The first is caused when a line segment moves laterally and crosses a vertex, causing intersection with edges that would not otherwise be intersected. The second is when an output line sub-segment is lengthened by snapping and its end point crosses a stationary line segment. The second case is functionally equivalent to the first since it doesn't matter whether a point moves to cross an edge or an edge moves to cross a point. Both can be handled by the same strategy so we'll focus on the case of the line segment moving in the description of our strategy. That strategy relies upon the following lemma: all artifacts take place only when a vertex lies within a distance of a line segment bounded by the max distance an intersection point can be snapped. This lemma can be trivially proven because the distance that segments can move is bounded and it is obviously impossible for two non-intersecting line segments to cross each other without one first crossing an end-point of the other. Moreover, since the direction of snapping is known to be always downward, it follows that a vertex can only be crossed by a line segment if that line segment intersects the 1x1 integer unit grid box with that vertex in its lower left corner. In these cases, we intersect the line segment with those vertices pro-actively such that if a future intersection causes the line segment to move, the output topology cannot contain spurious intersection artifacts due to that event. Because the vertex is intersected and not other edges, no additional line segment intersections need be introduced and no propagation of intersection artifacts through the network can take place. This method in known as snap-rounding and has been much discussed in the literature. [4]

Given an algorithm that finds intersections between line segments, it is easy to find intersections with 1x1 integer grid boxes at segment end-points and snapped-intersection points by modeling them as several tiny line segments called a widget. Any line segment that intersects the unit grid box will intersect at least one of the segments of the widget shown in Figure 8.

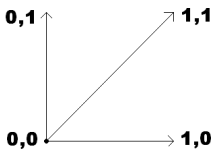

Figure 8. Example: Vertex/Segment Intersection Widget

Importantly, intersection events are detected by the algorithm based on only the input line segments geometry and never that of

the intersected line segments it has produced. Otherwise, numerical error could propagate forward in cascading increased severity to reach arbitrarily large magnitudes. If such were the case, no assurance of numerical robustness could be reasonably made.

If the snapping direction is uniform it can be arranged so that vertices snap forward in the scanning direction allowing evaluation of the widget to be performed by the same sweep-line that finds the intersections. This is our intention. However, currently our arbitrary angle polygon Booleans apply a much simpler line segment intersection algorithm implemented to validate the sweep-line version of robust line segment intersection, which is still a work in progress. It compares all pairs of line segments that overlap in their x coordinate range and all vertices and snapped intersection points with all segments that overlap with the x coordinate of those points. It has O(n^2) worst case runtime complexity, but in the common case it has expected runtime of $O(n^{3/2})$ and, in practice, performs nearly as well as the expected O(n log n) runtime of the optimal algorithm, making its use for even quite large input data sets highly practical.

The combination of handling overflow and applying rational data types to overcome truncation errors with the strategy for mitigating errors introduced by intersection point snapping allows 100% robust integer line segment intersection. The algorithm approximates output intersection points to within one integer unit in x and y and may intersect line segments with points that lie within one integer unit in x and y. This approximates the ideal "correct" output to the extent practical with integer coordinates. The algorithm could be enhanced to round to closest integer grid point when snapping intersections and make intersecting segments to nearby vertices predicated upon whether it later becomes necessary to do so. As a practical matter, however, these measures would result in very little benefit to accuracy. That benefit, and more, can be more easily obtained by scaling up the input and applying higher precision integer arithmetic, if necessary, which is easily accomplished using GTL.

7. Experimental Results

We benchmarked our own GTL polygon clipping Booleans algorithm against the GPC [7], PolyBoolean [6] and CGAL [8] polygon clippers. We benchmarked the three GTL algorithms, Manhattan, 45-degree, and general Booleans against all three. These benchmarks were performed on a two-package, 8 core, 3.0 GHz Xenon with 32 GB of RAM, 32 KB of L1 cache and 6 MB L2 cache. Hyper-threading was disabled. None of the algorithms tested were threaded and all ran in the same process. We compiled the benchmark executable using gcc 4.2.2 with O3 and finline-limit=400 optimization flags.

Inputs consisted of small arbitrary triangles, arbitrarily distributed over square domains of progressively increasing size. Runtimes measured were the wall-clock execution time of the intersection operation on geometry contained within two such domains. The overlapping triangles in each domain had to be merged first with GTL to make them legal inputs for the other three libraries' Boolean operations. For the Manhattan (axis-aligned rectilinear) benchmark we used small arbitrary rectangles instead of triangles.

Results of our benchmarking are shown in Figures 9, 10 and 11. Note that in Figure 11 the last two data points for PolyBoolean are absent. PolyBoolean suffered from unexplained crashes as well as erroneously returning an error code due to a bug in its computation of whether a hole is contained within a polygon. This prevented PolyBoolean from successfully processing large data sets. CGAL had a similar problem that prevented it from processing data sets larger than those in Figure 11. We conclude that this is a bug in CGAL because both GTL and GPC were always successful in processing the same polygons. This issue with GCAL was observed regardless of which kernel was employed.

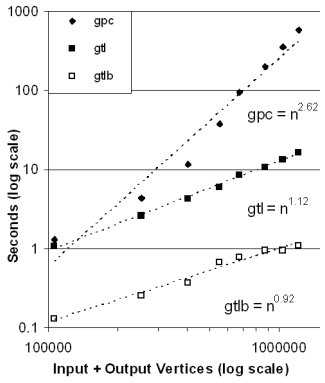

Figure 9. GPC/GTL Scalability Comparison

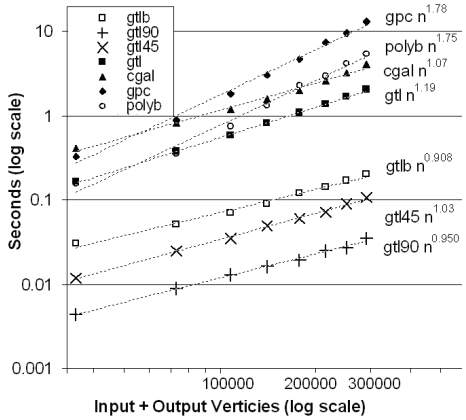

Figure 10. Rectilinear Scalability Comparison

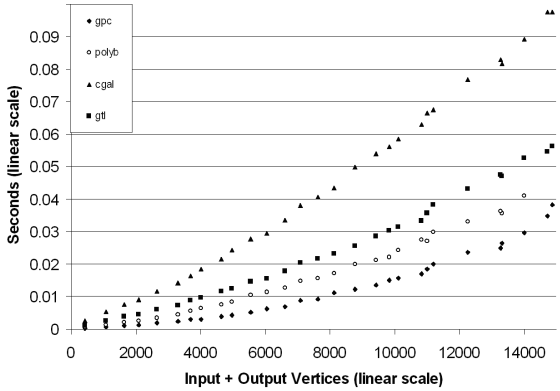

Figure 11. Small Scale Performance Comparison

All libraries performed roughly within the range of 2X faster to 2X slower than GTL for the small inputs shown in Figure 11. We feel that such small constant factor variation is not significant since it could likely be remedied by profile-guided performance tuning of implementation details. We did not apply empirical complexity measurement on the data sets in the General Performance plot because non-linearity in micro-architecture performance as memory footprints start to exceed L1 cache size renders such analysis on very small input sizes faulty.

While successful at processing all inputs, the GPC library's runtime scaled sub-optimally for large inputs, as can be seen in Figure 10. The empirical runtime complexity of GPC from that plot is $n^{2.6}$, which can be clearly seen in its steep slope relative to GTL. We were unable to measure CGAL or PolyBoolean for this benchmark because of the bugs that effectively prevented them from processing inputs larger than those shown in Figure 11. Also in Figure 9 we show the portion of GTL runtime spent in the core Boolean sweep as gtlb. Note that the runtime of GTL is dominated by the currently suboptimal line segment intersection, which we plan on eventually rectifying by integrating line segment intersection into the core Boolean sweep at a constant factor overhead.

All libraries were successful in processing large-scale Manhattan polygon inputs. There is a 100X variability in runtimes, however, as can be seen in Figure 10. The Manhattan Booleans algorithm in GTL is labeled gtl90 in the figure, and the 45-degree Booleans algorithm is labeled gtl45. Note that the 45-degree algorithm is optimal, computing line segment intersection in the same sweep as the Boolean operation, and performs within a small constant factor of the similar 90-degree algorithm. Again, we show the portion of the general Booleans algorithm labeled gtlb. We believe that when upgraded with optimal line segment intersection the general Booleans algorithm could perform closer to the gtlb curve than the current performance, which is labeled gtl. GPC and PolyBoolean both turn in suboptimal $n^{1.8}$ runtime scaling in this benchmark. CGAL appears to be optimal for this benchmark, scaling at a near linear $n^{1.07}$. Frequently we have observed O(n log n) algorithms will have an empirical scaling factor of less than one for input ranges that are modest in size, as we see in both log-log plots for gtlb as well as for gtl90. This is because the micro-architecture has advanced features such as speculative memory pre-fetch that become more effective as input vector sizes grow. However, it clearly demonstrates that empirical scaling observations must be interpreted cautiously when drawing conclusions about algorithmic complexity and optimality. Our review of GPC and PolyBoolean code lead us to believe that their line segment intersection algorithms should perform at around $n^{1.5}\log n$ on the test data we generated. Our conclusion that they are suboptimal is not based upon empirical data alone.

8. Conclusion

Our C++ Concepts based API for planar polygon manipulations makes these powerful algorithms readily accessible to applications developers. Improvements in our Booleans algorithm over prior work frees users of that API from the hassles of accommodating library restrictions and conventions imposed upon input geometries, while the C++ Concepts based API frees them from syntactic restrictions on how the algorithms may be applied. Because our library compares favorably with similar open-source libraries, in terms of both performance and feature set, while providing a superior API based upon generic programming techniques, we feel that it is a good candidate for acceptance into boost and plan to pursue review this year.

9. ACKNOWLEDGMENTS

Our thanks to Fernando Cacciola for technical guidance and editorial review and to Intel for supporting our work.

10. REFERENCES

[1] Bentley, J.L., Ottmann, T.A. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers, 9*, (C-28), 643-647.

[2] Gehrels, B., Lalande, B. Generic Geometry Library, 2009. Retrieved February 17 2009, from boost: https://svn.boost.org/svn/boost/sandbox/ggl

[3] GMP Gnu Multi-Precision Library, 2009. Retrieved August 9, 2008, from gmplib.org: http://gmplib.org

[4] Hobby, J. Practical segment intersection with finite precision output. Technical Report 93/2-27, Bell Laboratories (Lucent Technologies), 1993.

[5] Kohn, B. Generative Geometry Library, 2008. Retrieved July 22, 2008, from boost: http://www.boostpro.com/vault/index.php?action=downloadfile&filename=generative_geometry_algorithms.zip&directory=Math - Geometry&

[6] Leonov, M. PolyBoolean, 2009. Retrieved March 15, 2009, from Complex A5 Co. Ltd.: http://www.complex-a5.ru/polyboolean/index.html

[7] Murta, A. GPC General Polygon Clipper library, 2009. Retrieved March 15, 2009, from The University of Manchester: http://www.cs.man.ac.uk/~toby/alan/software/

[8] Pion, S. CGAL 3.3.1, 2007. Retrieved October 10, 2008, from CGAL: http://www.cgal.org

[9] Ruud, B. Building a Mutable Set, 2003. Retrieved March 3, 2009, from Dr. Dobb's Portal: http://www.ddj.com/cpp/184401664

[10] Dos Reis, G. and Stroustrup, B. 2006. Specifying C++ concepts. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, USA, January 11 - 13, 2006). POPL '06. ACM, New York, NY, 295-308. DOI= http://doi.acm.org/10.1145/1111037.1111064

[11] Vatti, B. R. 1992. A generic solution to polygon clipping. *Commun. ACM* 35, 7 (Jul. 1992), 56-63. DOI= http://doi.acm.org/10.1145/129902.129906

[12] Weiler, K. 1980. Polygon comparison using a graph representation. In *Proceedings of the 7th Annual Conference on Computer Graphics and interactive Techniques* (Seattle, Washington, United States, July 14 - 18, 1980). SIGGRAPH '80. ACM, New York, NY, 10-18. DOI= http://doi.acm.org/10.1145/800250.8074