Efficient Polygonal Intersection Determination with Applications to Robotics and Vision

Christohper E. Smith and Hanspeter Schaub

Simulated Reprint from

IEEE/RSJ International Conference on Intelligent Robots and Systems

Edmonton, Alberta, Canada, Aug. 26, 2005

Efficient Polygonal Intersection Determination with Applications to Robotics and Vision

Christopher E. Smith
Dept. of Electrical and Computer Engineering
The University of New Mexico
MSC01 1100
1 University of New Mexico
Albuquerque, NM 87131-0001

Hanspeter Schaub
Dept. of Aerospace and Ocean Engineering
Virginia Tech University
215 Randolph Hall
Blacksburg VA, 24061

Abstract

Several robotic and computer vision applications depend upon the efficient determination of polygonal self- and mutual-intersection checking. The commonly used algorithms for intersection checking rely upon static geometric primitives, such as lines and vertices. When these geometric primitives are dynamic, that is moving or changing shape, these algorithms become inefficient due to repeated actions that do not utilize topological features of the primitives. In this paper we present a novel algorithm for line segment intersection checking that builds a query structure and then updates the structure using previously computed topological data. We exploit the fact that the amount of model deformation is limited during any single iteration, yielding a relatively small bookkeeping cost to maintain the query structure. The result is an algorithm whose asymptotic runtime complexity in the expected case is better than competing methods. We then suggest an extension of this work into higher dimensions (polytope intersection for 3-D and higher).

1. Introduction

Intersection checking is a crucial activity for many robotic and computer vision applications. Many potential techniques have been developed to perform intersection checks in collections of line segments and polygons; however these algorithms rely upon the geometric primitives (line segments and vertices) being static. In applications where geometric primitives are allowed to deform, these algorithms must be restarted from scratch after each deformation. This leads to gross inefficiencies that can result in asymptotic runtime complexity higher than simple brute force algorithms.

In particular, we begin with an application of intersection checking derived from our previous computer vision work in active deformable models (snakes, see Figure 1for an example image) [5][7]. Our particular energy functional in these models requires that we perform self-intersection checking. In other forms of these models, self- and mutual-intersection checking becomes necessary when multiple snakes are used topologically adapt to apparent splits and merges in tracked targets [8] (See Figure 2). Merges occur when two (or more) snakes mutually-intersect and the individual snakes are merged into a single model (as with the red and green snakes). Splits occur when a snake self-intersects and the result is two (or more) snakes evolving from the original (the blue snake shown before a split occurs).

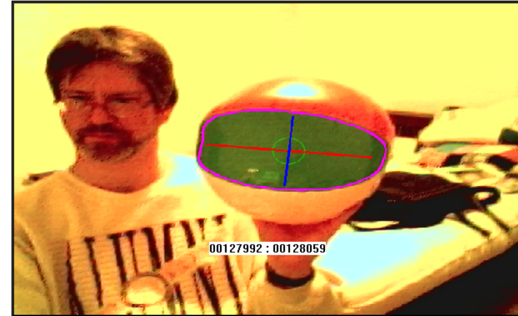


Figure 1 A snake tracking a beach ball section

Since snakes are often applied to dynamic objects in video streams (see Figure 9 for a sequence from a mobile-manipulation application), there is a strong real-time constraint placed upon the models. If intersection checking is inefficient, then it jeopardizes the real-time requirement of the system.

To address this problem, we investigated the work on intersection checking that has previously

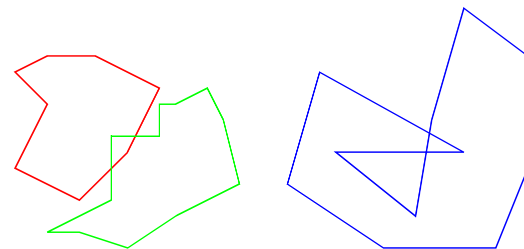


Figure 2 Mutual (left) and self (right) intersections

appeared in active deformable model and in computational geometry literature. Lin and Gottschalk [3] provide an excellent overview of many of the techniques from robotics, computational geometry, and computer graphics. The technique most closely related to ours is the technique of Cohen, et al. presented in [1]. This technique uses projected, axis aligned bounding boxes to drive colision detection among “close” polytopes. The projection reduces the dimensionality of the query by one. That is, intersection among 3-D polytopes will occur after projection of the bounding boxes into 2-D. While efficient, our particular application has further topological knowledge that allows us to improve upon the expected time for a collision query from $O(n \log n + K)$ for the method of Cohen et al. to $O(n + K)$ where n is the number of rectangles (i.e. bounding boxes) and K is the number of intersecting rectangles.

Starting from a brute force algorithm, we modified the algorithm to be more efficient. In doing so, we uncovered some topological constraints that suggested a more efficient method might exist. Using ideas from both computational geometry and our own optimized brute force approach, we developed a new, efficient method for determining line segment intersections that out performs existing algorithms in our application domain. We then looked at extending the algorithm to higher dimensions (beyond 2-D) that have potential in other application areas. In all these cases, when the geometric primitives are deformable (e.g. allowed to move and change shape) that our algorithm becomes significantly more efficient than alternative approaches.

2. Prior Work

Prior work in intersection checking has been almost exclusively performed under the auspices of computational geometry. A brute force technique exists that simply tests each possible pair of line segments to determine if the segments intersect. This algorithm is clearly $\Theta(n^2)$ where n is the number of line segments. This is not very efficient and generally is not used in computational geometry literature.

In general, efficient line segment intersection checking is based upon a class of algorithms known as sweep-line or sweep-plane algorithms [6]. Sweep algorithms are equivalent to sorting and therefore require $\Theta(n \log n)$ comparisons to determine which of n line segments intersect (or if any intersections occur). The basic sweep algorithms lexicographically sorts the line segment endpoints and uses a priority queue to store the $2n$ sorted endpoints. The algorithm then only performs at most $2n + K$ work where K is the total number of intersections. This gives us a $\Theta(n \log n)$ algorithm to determine if a polygon is simple (does not self-intersect). Since the control points of a snake are in constant motion, the algorithm needs to re-sort the endpoints after every control point update, making a naive application of this technique $\Theta(n^2 \log n)$ for a single iteration of the snake algorithm (that is, each control point gets updated once).

It is clear that the naive sweep algorithm is rechecking segments that need not be rechecked and re-sorting the entire set of segments when only two segments (the two that have the updated control point as an endpoint) need be checked. Several authors have noted this and produced algorithms that are variations of a technique called sweep-and-prune [3]. These techniques do not naively re-sort the entire list, giving an improvement over this approach. These improvements still yield runtimes over our target of $O(n + K)$ where K is commonly a constant, yielding $O(n)$ runtime.

A technique specifically adopted for use in snake algorithms relies upon graphical line drawing to identify line segment intersections [2]. Basically, a graphical bitplane is initialized to zeros and then each line segment is drawn into the bitplane using a unique segment id number as the pixel color value. When an attempt is made to draw onto a pixel whose value is already nonzero, then an intersection has occurred between the segment currently being drawn and the segment whose id is stored in the pixel. The major problem with this algorithm is that aliasing (stair stepping) of the graphical line segments can incorrectly report both false positives and false negatives (see Figure 3 where the red represents one snake, the blue the other, and the black represents a detected intersection). The false positives

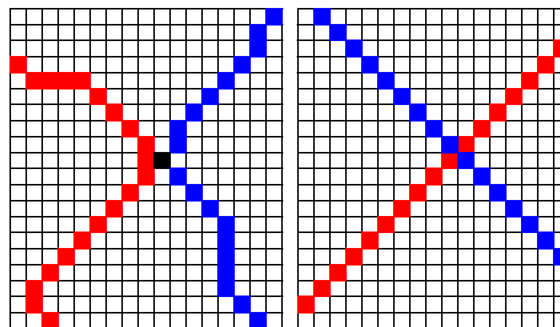


Figure 3 A false positive and a false negative case

can be easily handled by performing a confirmation intersection check, but the false negatives cannot be confirmed. In order to eliminate false negatives, another algorithm must be used, destroying the advantages of this method.

One technique to eliminate this problem involves drawing lines with a width of two pixels to eliminate the false negatives; however this can significantly increase the number of false positives. Overall, this algorithm is substandard for line segment intersection applications with dynamic geometric primitives.

3. More Efficient Intersection Checking

Improved Brute Force

Initially, we used the brute force technique to determine whether or not intersections exist. While inherently inefficient, the algorithm is simple and, as discussed in the previous section, was better than the naive application of a sweep algorithm.

The method for determining whether or not a pair of line segments intersect is computationally expensive relative to other basic operations in the intersection checking algorithm. The most common method uses, for a single pair of segments, four dot-products and multiple coordinate comparisons [4]. It is therefore beneficial to reject testing certain pairs if simple comparisons can rule out the possibility of an intersection. We used a popular bounding-box-overlap method to reject many of the pairs in our snake application. One simply determines the bounding box for each line segment that is to be checked and if these bounding boxes overlap each other the segment pair is tested for intersection. Degenerate cases can force this technique to run slower than the unmodified brute force algorithm; however, such degenerate cases are virtually impossible to produce in real-world data.

3.1 The New 2-D Algorithm

The sweep technique, while not suited to changing geometric primitives, has desirable aspects that are useful in intersection checking.

We used sweep algorithms as a template for the construction of a query structure that will be used in our algorithm. When a snake is initially created, we know that the snake is a simple polygon and therefore we only need to build the underlying query structure. The structure is comprised of two lexicographical sorts (in both X and Y) of the

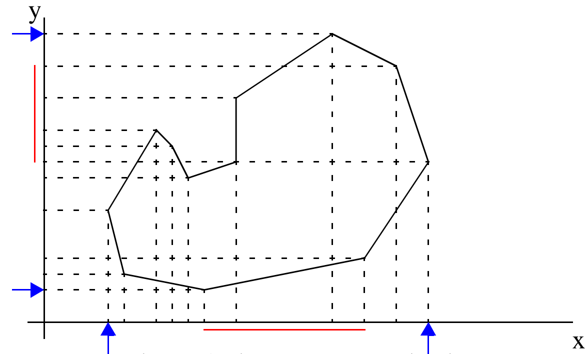


Figure 4 Line segment projections

snake control points. This can also be thought of as projecting the line segments onto the X- and Y-coordinate axes (see Figure 4).

During the structure's construction, basic topological features are calculated and stored in the structure to improve algorithm efficiency. Specifically, these topological features include the greatest spans of any segment in X and in Y (shown in red in Figure 4), the minimum and maximum X coordinates over all the control points, and the minimum and maximum Y coordinates over all the control points (all marked with blue arrows). Once built, the query structure can be directly accessed through pointers stored in the snake control point structures for intersection checking and for updating. Figure 5 shows how a snake control point node exists within the lexicographically sorted lists and within the snake model itself. The lexicographical lists can be conceptualized as doubly-linked lists while the snake itself is represented as a doubly-linked ring.

When a control point's location is updated, the two segments that have the control point as an endpoint are used to access the query structure. The basic idea is to search in both the increasing and decreasing directions of the lexicographical sort to determine candidate segments to test for potential intersection. The candidate segments are determined similar to the bounding box test used in our modified brute force algorithm.

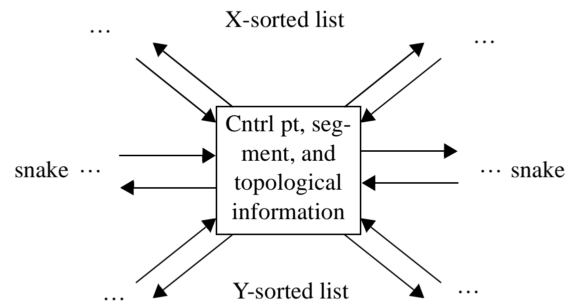


Figure 5 A snake node's relationship to the query structure

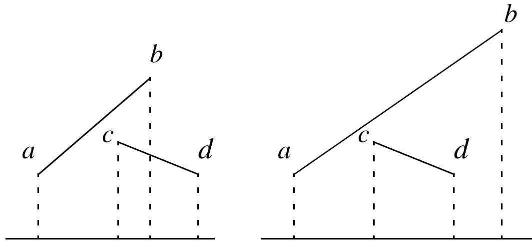


Figure 6 The two cases for line segment overlap

During such a test, only one of the two lexicographically sorted lists needs to be used, so the axis with the largest difference between the minimum and maximum coordinate values for that axis. This heuristic is based upon the assumption that the axis with this largest difference will also be the axis with the fewest overlaps among the projected line segments.

The distance from the control point that must be searched is determined by the greatest span of any segment for the axis used in the sort. For instance, two line segments might intersect if, when lexicographically sorted, they overlap (akin to bounding box overlap) in the sort. Figure 6 shows the two cases where line segments will overlap in the lexicographical sort. Consider that control point c has been updated. Since we know the greatest span of any segment projected into the coordinate axis, one only needs to search half of that distance below c and half of that distance above d. This will guarantee that any segment that lexicographically overlaps $\overline{cd}$ will have an endpoint encountered during the search process. This occurs regardless of whether the overlap is the left or the right case in Figure 6.

While the query structure can be built in $\Theta(n^2)$, this is a one-time cost and the efficiency actually depends upon the number of line segments that are tested and the updates to the structure. In our snake application, the snake must be reparameterized to maintain validity of the energy model. To address this, the control point spacing is bounded by both a minimum and a maximum length. Because of this, and in part due to the limits placed upon control point motion during a single iteration, we can guarantee that control points will move only small distances during any update. When a move occurs, the point may be required to move inside the lexicographically sorted lists to maintain the query structure; however, small motion will generally mean that the number of positions in the lists that the control point must be moved will be $\Theta(1)$ in the expected case. It is this topological knowledge that allows us to reduce our runtime.

Similarly, given any snake other than severely degenerate cases, the number of line segments that will need to be checked will be $\Theta(1)$ expected case for a single control point update.

In both instances (updates and tests) the expected case runtime is $\Theta(1)$ for each control point, the overall expected case runtime for an entire snake iteration (where each control point is updated once) will cost $\Theta(n)$.

Worst case runtime is much worse, reaching a theoretical limit of $O(n^2)$. Heuristically, we choose either the X or the Y list by picking the one with the greatest span measured by $maxW - minW$ where W is either X or Y. This limits our exposure to degenerate cases, but does not eliminate them. In fact, a true degenerate case that produces an $O(n^2)$ runtime is virtually impossible to obtain for our application. It would require a snake that looks like the one pictured in Figure 7. This case requires that every line segment overlaps every other segment in both the X and the Y directions. In practice, we have never witnessed any case that did not exhibit the expected case behavior.

The most difficult part of maintaining the query structure is keeping the topological data updated. The maximum and minimum values in each ordinal direction are relatively easy to derive during a single snake iteration; however the greatest segment span in X and in Y are more difficult to derive. Upon a control point update, the spread of the two changed segments must be checked to see if they exceed the current maximums. If they do, then the greatest spread must be updated for use through the end of the current iteration. At the same time, new greatest spread data must be collected only from those segments that have been updated during the current iteration. Once the iteration ends, this new data replaces the old spread data and will in turn be used for the next iteration. These updates have a constant runtime, but the housekeeping requires careful attention to the topological data and its current validity.

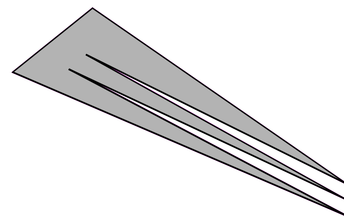


Figure 7 A degenerate snake

3.2 Experimental Validation

3.3 Extension to Higher Dimensions

The basic ideas in our 2-D algorithm can be extended to 3-D (and higher) geometric models. The limiting factor will be maintaining nearly uniform geometric primitive size. In 2-D, this was taken care of via the spacing constraint placed upon the snake by reparameterization requirements. In 3-D, the basic geometric primitive becomes a polygon and the lexicographical sorts must be maintained along three axes (X, Y, and Z). For two polygons to have a potential intersection, the polygons' spreads must overlap in each of the three ordinal directions. Since the intersection tests rely upon knowing the greatest spread over all polygons in the model and the constant time performance relies upon these spreads to be relatively uniform across all the polygons, again we must enforce reparameterization of the model.

Fortunately, most 3-D models already require reparameterization. For instance, finite element meshes generally enforce minimum and maximum triangle areas, forcing reparameterization when a triangle in the mesh either becomes too large or too small.

The extension to 3-D opens up new applications in areas such as collision detection and graphical modeling. The same technique can be extended arbitrarily; however, the utility and efficiency lessen as dimensions are added to the model space. Basically, the overhead incurred to maintain the multiple lexicographical lists (one for each dimension) overwhelms the gains in efficiency produced from the use of the query structure to identify candidate geometric primitives for further intersection tests. In general, the number of applications in robotics and vision are greatly reduced for dimensions higher than three.

4. Experimental Evaluation

We have tested our improved intersection checking using a simple snake application built from the OpenCV image processing libraries. The brute force intersection checking exhibited significant slowdown when the number of snake control points reached >80. This is typical of a $\Theta(n^2)$ algorithm. With our new intersection algorithm, snakes with >200 control points continued to run at the frame rate of the camera being used (a USB webcam). These tests were performed on a Pentium III laptop with a 700Mhz clock.

Code profiling on our improved algorithm showed that using snakes with a high number of control points, the vast majority (>90%) of the machine cycles were used by the Tcl/Tk interface on our demonstration application. This was a completely unexpected result given that the improved algorithm still has a worst case performance that is the same as the brute force algorithm. We then benchmarked our results by timing the intersection code.

To verify our analysis of the expected case runtimes, we conducted two sets of experiments. One set used the brute-force method on simple object tracking cases where relative object size determined the number of snake control points.

In Figure 8 we see the time taken (blue data points) by the brute-force method when checking a single line segment against the entire polygon. The plot shows time in microseconds (vertical axis) vs. number of control points (horizontal axis) with multiple trials for each size of object. The plot shows the characteristic linear growth with respect to the number of control points. Remember that this is the time taken for a single segment to be checked, so that the time to check the entire polygon follows the predicted $O(n^2)$. We then conducted the same set of object tracking trials using our improved algorithm. These results are plotted in red. The time remains constant even when intersections are induced by occlusion of the object being tracked. This results in our predicted $O(n+K)$ complexity where K is a constant, thus reducing the expression to $O(n)$.

5. Conclusions

We have presented a novel line segment intersection checking algorithm that has a significantly lower expected case runtime than popular alternative algorithms. The methods works well with dynamic geometric primitives and can easily be extended to higher dimensional geometric models. Our results show a significant decrease in the runtime of our selected example application of active deformable models (e.g. snakes) allowing snakes of well over 100 control points (in a single snake or combined when running multiple snakes

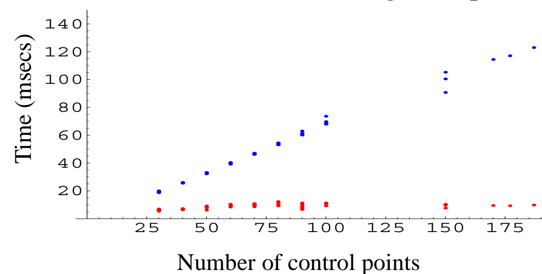


Figure 8 Comparison of intersection runtimes

simultaneously) without significant system slowdown.

6. Acknowledgements

This work has been supported in part by the U.S. Department of Energy under Grant No. DE-FG04-95EW55151 issued to the Manufacturing Engineering Program at the University of New Mexico, by the Sandia National Laboratories University Research Program (SURP), and the Department of Electrical and Computer Engineering at the University of New Mexico.

7. References

[1] J. Cohen, M. Lin, D. Manocha, and K. Ponamgi, "I-COLLIDE: an interactive and exact collision detection system for large-scaled environments," *Proceedings of the ACM Symposium on Interactive 3D Graphics*, 1995.

[2] J. Ivins, "Statistical snakes: active region models," *Ph.D. Thesis*, University of Sheffield, 1996.

[3] M. Lin and S. Gottschalk, "Collision detection between geometric models: a survey," *Proceedings of the IMA Conference on Mathematics of Surfaces*, 1998.

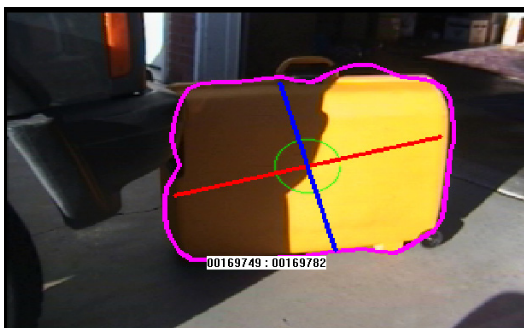
[4] J. O'Rourke, Computational Geometry in C, Cambridge University Press, Cambridge, UK, 1998.

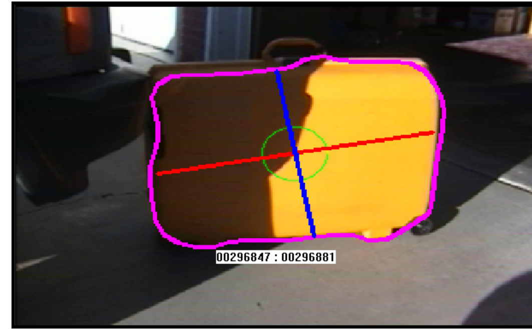
[5] D. Perrin and C. Smith. "Rethinking classical internal forces for active contour models," *Proceedings of the IEEE International Conference on Computer Vision and Pattern Recognition*, 2001.

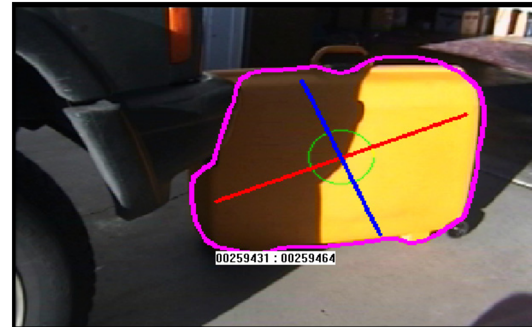
[6] F. Preparata and M. Shamos, Computational Geometry: An Introduction, Springer-Verlag, new York, NY, 1985.

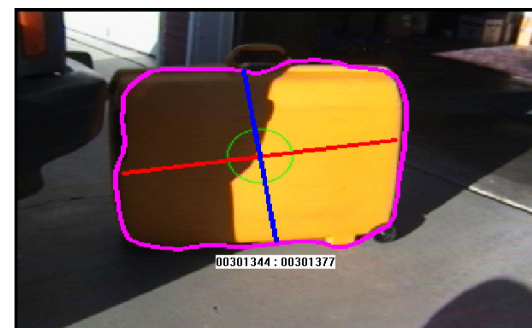
[7] H. Schaub and C. Smith, "Color snakes for dynamic lighting conditions on mobile manipulation platforms," *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2003.

[8] S. Stoeter and N. Papanikolopoulos, "Closed Dynamic Contour Models that Split and Merge," *Proceedings of the IEEE International Conference on Robotics and Autom*ation, 2004.









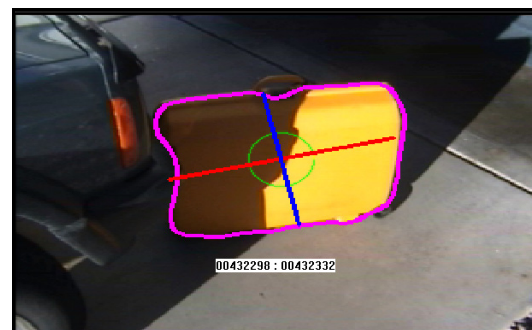


Figure 9 An image sequence from a mobile-manipulator security application