Robust Incremental Polygon Triangulation for Fast Surface Rendering

Subodh Kumar
Department of Computer Science
Johns Hopkins University
Baltimore, MD 21218, USA
subodh@cs.jhu.edu

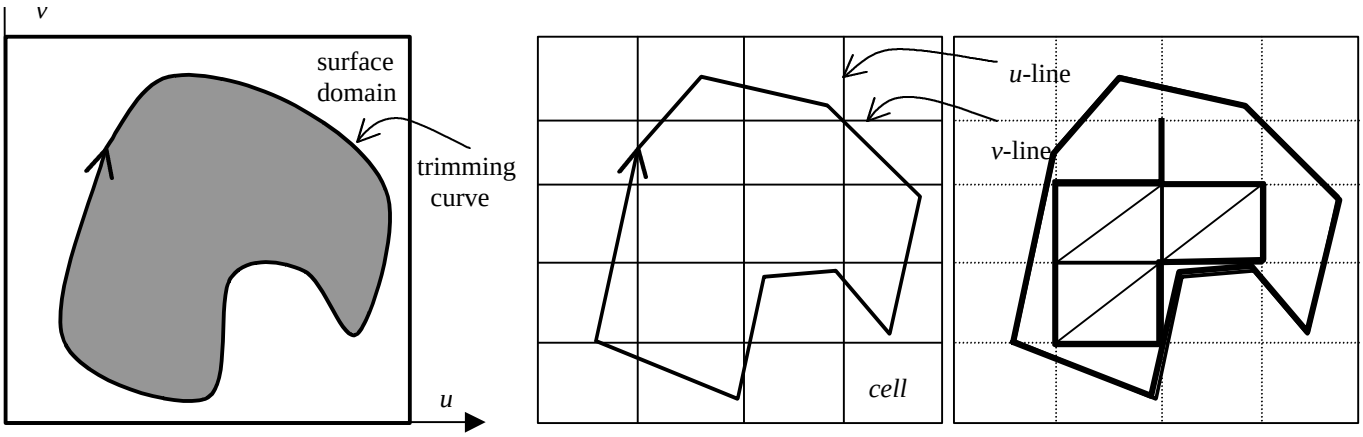
Abstract: This paper presents a simple, robust and practical, yet fast algorithm for triangulation of trimmed Bézier surfaces. Points on the surface are input to our algorithm by a sampling scheme. A set of polygons (more generically, a planar straight-line graph, or PSLG) is formed from these samples, which are then triangulated. We also show how to update the triangulation when the samples, and hence the polygons, are updated. The output of the algorithm is a set of triangle strips. The algorithm avoids long and thin triangles. In addition, it also detects if the sampling of the trimming curve forms any non-simple polygons and corrects the triangulation by adding more samples. We report an implementation of the algorithm and its performance on extensive simulation.

Keywords: Surface rendering, CAD, Triangulation, Polygon, PSLG, Computational geometry.

1 Introduction

View dependent surface triangulation is a popular technique for interactive display and walkthroughs of large geometric models like those of ships and submarines. Such real-time inspection in a virtual environment provides the sense of space and is crucial for simulation based design. It can reduce the time and cost of manufacturing by reducing the need for full-scale mockups. Such view-dependent techniques enable fast rendering by generating more polygons only the region of high detail close to the viewer and less in other regions. However, such schemes require re-sampling, and re-triangulation of surfaces every frame. To facilitate these operations, we present an incremental triangulation algorithm for parametric surfaces that allows addition and deletion of samples. The algorithm for re-sampling is presented elsewhere [KML96,Kum96].

Our system uses the Bézier representation as its basic parametric primitive. In particular, trimmed Bézier and Non-uniform Rational B-spline (NURBS) forms are widely used to represent complex models in engineering and other domains. A number of techniques have recently been proposed for sampling and tessellation of trimmed Bézier surfaces [RHD89,AES94,LC93,PR95,KML96]. In particular, [KM95] proposes an incremental triangulation algorithm. Our algorithm allows *incremental updates* as well and is comparatively more efficient in terms of worst case complexity, expected complexity, and observed behavior. Additionally, in the trapezoidation step, we have incorporated *simplicity detection* at almost no extra cost. Consequently, our implementation is much *robust*. We also avoid producing long and thin triangles. Furthermore, unlike most recent algorithms, we directly generate *triangle strips*, which are more efficient to render than triangles are on current graphics systems. Several asymptotically efficient polygon triangulation algorithms are known [CI84,Cha91,CTV89,FM84,Sei91] but most are difficult to implement and they do not ensure triangle quality. Recently researchers have shown how to ensure triangle quality using Steiner points [BDE92,BE92,Mit93,MSR94]. For example, in two dimensions, it is possible to triangulate a polygon using triangles with angles at most $7/8\pi$ using $O(n^2 \log n)$ Steiner points in time $O(n^2 \log^2 n)$ [Mit93]. Unfortunately, these Steiner points greatly increase the number of triangles generated. Moreover, Steiner points need to be added on polygon edges. In our application such on-edge Steiner points can result in cracks [KML96] and must be avoided. At the same time our polygons include strategically placed vertices and an enclosed grid, which helps us generate *good-quality triangles* on average without having to pay the extra cost for it. (Note that inclusion of grid vertices does not reduce the complexity of the polygon-triangulation problem.) The rest of this paper is organized as follows. In section 2, we describe the application and provide the problem description. Section 3 describes the curve-tracing step that is used to construct the planar straight-line graph (PSLG). In this paper, we present the algorithm mainly in terms of polygons but all steps and proves hold for all PSLGs. In general, we triangulate PSLGs but quickly reduce them to polygons. Section 4 discusses PSLG triangulation. In section 5, we describe our data structures for efficient point location and present the incremental algorithm. Finally, we provide the implementation results in section 6 and conclude in section 7.



(a) **(b)** **(c)**

Figure 1

2 Problem Description

A tensor product parametric surface, $S(u,v)$, is defined by a vector function over the domain $(u,v) = [0,1] \times [0,1]$. Optionally, a closed sequence of trimming curve, $C_i(t)$, each defined over the domain $t = [0,1]$ may be defined on the domain of S (see Fig. 1). For brevity, we will refer to the sequence as a single trimming curve, C. The trimming curve restricts the domain of S. Depending on the orientation of the curve, we either discard the part of domain of S enclosed by it or that enclosing it. In this paper, we only consider the trimming curves that comprise a single connected component (other than the regular 0-1 domain boundary). Reducing multiple components to this case follows the same principle as described in [KM95]. Before triangulation, the surface and the trimming curves are uniformly sampled. The sampling is performed for every frame in an interactive graphics simulation. Since the sampling density for a surface does not change much in consecutive frames, it is more efficient to modify the triangulation of the previous frame than to re-generate the whole triangulation anew. For a discussion of sampling techniques, please refer to [KM95,KML96]. This paper is concerned with producing a triangulation of these sampled points. In particular:

- The initial input to the algorithm is a grid of points and a closed polygonal chain (Fig. 1(b)) on the *u*-*v* domain. These input points are uniformly distributed on the domain and the screen-space distance between adjacent points is bounded. In addition, an update involving an addition or deletion of one of the grid-lines or a sample on the curve may be performed. While multiple changes on a surface can be simultaneously handled, it is easier to describe the algorithm in terms of a single update at a time.

- The output is a triangulation such that no triangles lie outside the domain restricted by the trimming curve's tessellation. In addition, no edge of the triangles generated may be greater than twice the maximum distance between adjacent input samples.

- Long and thin triangles are undesirable for smooth shading. However, since the input to our triangulation routine is generated by a uniform sampling, the points are usually well distributed. Thus, an exact Delaunay triangulation is not necessary. Using simple heuristics, we are able to obtain fat triangles in practice.

- Since triangle strips can be displayed much more efficiently than a list of triangles, the output of our algorithm is a set of triangle-strips. We follow the OpenGL convention for these strips. We avoid triangle fans, since fans tend to produce higher degree vertices in the triangulation. We keep the number of vertices adjacent to a given vertex in the triangulation small in order to lower the cost of incremental updates to the triangulation (see Section 5).

- No Steiner points may be added on the trimming polygon (i.e., the sampled trimming curve), since doing it independently of adjacent surfaces can lead to cracks in the tessellation of the model. Additionally, we do not introduce Steiner points elsewhere on the domain either, thus saving the extra triangles and keeping the point location query simple.

- Note that trimming curves are non self-intersecting. However, a sparse sampling of the trimming curve can still produce non-simple polygons. While uncommon, non-simple polygons do occur and can cause most triangulation algorithms to fail. Instead of assuming that the input polygon is simple, which can require very high sampling density, we check for such cases at little additional cost and thence make the polygon simple by increasing the sampling *only* when necessary.

In our application domain, a significant number of instances of the algorithm are executed every frame. The size of each instance is relatively small. Thousands of surfaces are re-triangulated per frame. (We ran several experiments to characterize sampling sizes in typical surface-model walkthroughs.The number of surface samples range mostly from 4 to 100, the number of curve samples range mostly from 5 to 100 and the number of points in a non-empty cell ranges

from 1 to 5.) Hence, in addition to simplicity and robustness, the constants of complexity of the triangulation algorithm are crucial to interactive performance. The space requirement per surface is not quite critical, but it is prohibitive to maintain a large data structure for each surface across all frames. To speed up point location operations (required for incremental updates), we use the natural partitioning provided by the grid sampling of the surfaces domain and discard all other auxiliary data structures, which are re-constructed every time they are needed. Before we describe the construction of this data structure, we introduce some notation. Without loss of generality, assume that the curve is specified clock-wise and the part of the domain to be triangulated is enclosed by the trimming curve.

Definitions: The polygon corresponding to the tessellated trimming curve consists of points p_i (see Fig. 1). The surface grid points are denoted by $g_{ij.}$ We also denote a vertical grid line (a sequence of sample points) by u_i and horizontal lines by v_i. i and j range from 0 to the corresponding sampling sizes. We also refer to the u and v coordinates of a point, p, on the domain as its *u-value* or $u(p)$ and *v-value* or $v(p)$, respectively. Each rectangle formed by four adjacent grid points is called a *cell*. The part of the domain between two consecutive grid lines is called a *strip*.

3 Polygon Tracing

The surface grid provides a natural partition of the domain. We trace the trimming polygon, i.e. process p_is in polygon

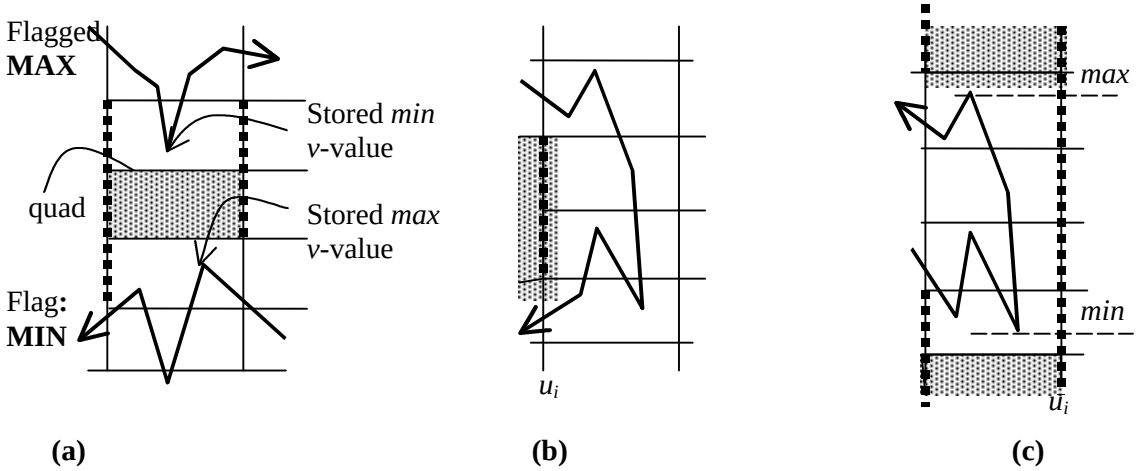


Figure 2: Curve Tracing (Segments of the PSLG are indicated by dots**)**

order, assigning each point p_i to the grid cell it lies in, $cell(p_i)$. The tracing step generates the PSLG that we seek to triangulate (Fig. 1(c) shows an example in thick lines.). During this process, we also construct auxiliary data useful for triangulation. In addition, we compute all intersections of the polygon with u-lines. For each cell, we keep a list of intersections of the polygon with its left boundary in increasing order of their v-values. We discard all degenerate intersections, i.e. if a polygon segment is collinear with a u-line or a v-line, we remove all corresponding grid points from consideration. We store the following information during curve-tracing:

For each intersection, I, of the polygon segment, p_i-$p_{i+1,}$ with a u-line, u_j:

- Mark I as **MIN**, if $u(p_i) > u(p_{i+1})$. Mark as **MAX** otherwise. ($u(p_i) \neq u(p_{i+1})$ as degenerate intersections are not allowed — only grid points and lines that lie strictly inside the trimming polygon are included in the PSLG)
- Store $v(I)$, the v-value of the intersection. (Keep multiple intersection with a cell boundary sorted by v-value.)
- Store $Maximum(v(I))$ and $minimum(v(I))$ attained by the polygon in the u-strip containing p_i. These bound the quads.
- Store a pointer to $p_{i+1.}$ For each cell's left and right u-boundaries we maintain a linked list of all intersections with that boundary. Note that in the worst case a cell could have $O(n)$ intersections with a polygon with n vertices. However, since the cells and polygons follow the same sampling rule and highly winding trimming curves are tough to generate, the number of intersections of most cell boundaries is small, if not 0 or 1.

If the entire trimming polygon lies within the same cell, no intersections are detected. This case does occur in practice, especially for surfaces with small on-screen area or with degree 1×1. It means the polygon lies within the same cell. In such cases, typically the number of points on the polygon is small as well. If that is not the case, additional grid lines may be included solely for curve tracing, thus ensuring that the number of points in a cell is still small.

We generate the quads and the PSLG using what amounts to a modified sweep line algorithm. We find **MIN-MAX** pairs on each u-line and a corresponding **MIN-MAX** pair on its adjacent u-line. (see Fig. 2(a)). The pairs on each line are available in the sorted order. Note that adjacent **MIN-MIN** or **MAX-MAX** pairs indicate non-simple polygon. The matching of pairs on adjacent line (to obtain a strip of quads) is as simple as matching the i^{th} pair on both lines, except the two special cases shown in Figs. 2(b) and 2(c), when polygon chain turns back to intersect the current line instead of the

adjacent line. For case (b), we generate an extra pair on the *u*-line u_i, discard it (for matching purposes). For case (c), we insert an extra **MIN-MAX** pair on u_i for matching.

4 Triangulation

Our basic triangulation scheme is based on trapezoidation [CI84,Sei91]. The basic idea of this technique is demonstrated

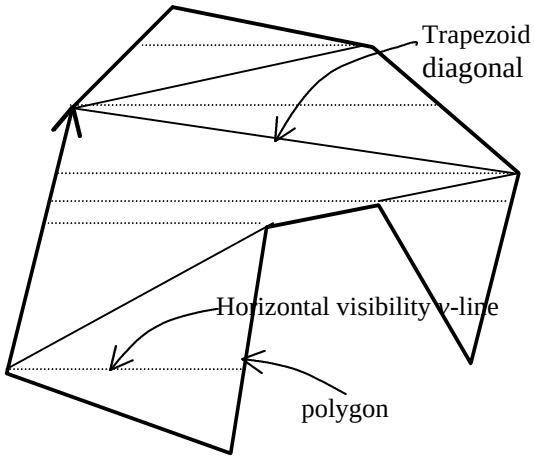
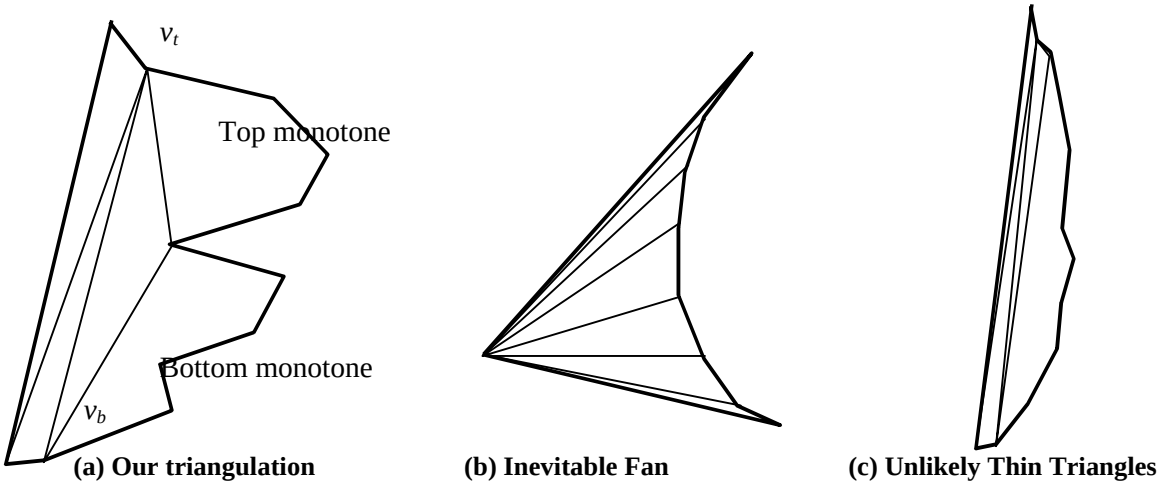


Figure 3: Trapezoidation

in Figure 3. Trapezoidation of a PSLG (shown in thick solid lines) is obtained by drawing horizontal rays (i.e. dashed lines parallel to the *u* axis) at each vertex of the graph limited in both directions by the first segment (or vertex) the ray intersects. The PSLG segments and horizontal lines form a set of trapezoids. The diagonals (shown in thin solid lines) of these trapezoids that connect two vertices of the PSLG partition the PSLG into a set of uni-monotone polygons. Uni-monotone polygons consist of a single *v*-monotone chain and another line-segment. For a discussion and proofs, we refer the reader to [Sei91,FM84]. It can be shown (we omit the proof here) that the line-segments mentioned above, call them monotone segments, are all PSLG segments, and thus small in length for our application. We will exploit this fact while triangulating these monotone polygons.

4.1 Monotone Triangulation

A number of simple O(*n*) algorithms for triangulating monotone polygons have been proposed [FM84,GJPT78] and



(a) Our triangulation **(b) Inevitable Fan** **(c) Unlikely Thin Triangles**

Figure 4: Triangulating Monotone Polygons

implemented [NM95]. However, all of them tend to produce triangle fans and long and thin triangles, both undesirable properties for our purpose. We propose another algorithm, equally efficient in practice that produces better triangle strips. Our approach is motivated partly by [HM83] and [RR94]. Designed for uni-monotone polygons, our algorithm is much simpler and more efficient. We use a *u*-tree. A *u*-tree maintains all the local *u*-minima and can be constructed in O(*n*). The invariant for a *u*-tree node is as follows: it stores the vertex with the minimum *u*-value of all its children. All vertices above it (i.e. with higher *v*-value) are kept in its left sub-tree and all vertices below it are kept in the right sub-tree. Fig. 4(a) illustrates the basic idea of tree construction. A *u-tree* can be constructed incrementally in a single pass over the *n* vertices of the polygon in O(*n*) + O(k^2), where k is the number of local *u*-minima. Processing the minima in a random

order reduces the expected cost to O($n + k \log k$). However, k is typically less than 3-4 and processing the vertices in the polygon order is sufficient in practice. Once the *u*-tree is constructed, we produce the triangle strips in O(n) time as follows:

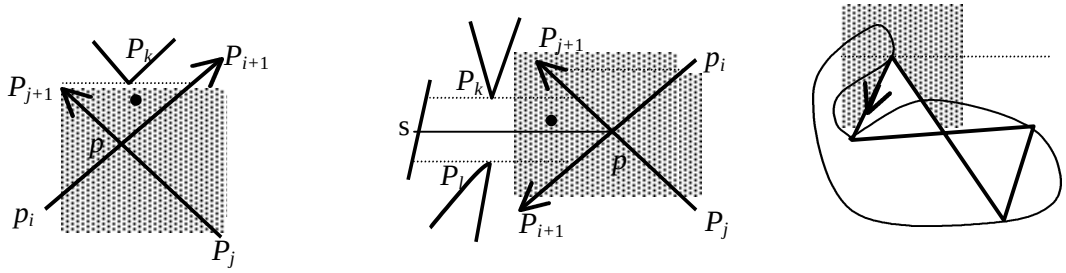
- Maintain pointers to the current root v_m, current top, v_t, and bottom, v_b, vertices of the chain
- While v_t and v_b both have lower *u*-value than v_m does:

  Add the one with lower *u*-value, say v_t, to the strip, Replace v_t by the next vertex on the polygon
- Otherwise,
  - Add v_t, v_m and v_b to the current strip and output it.
  - Diagonals v_t -v_m and v_b - v_m subdivide the polygon into two *v*-monotone polygons, the left sub-tree of the current *u*-tree corresponds to the top polygon and the right sub-tree corresponds to the bottom polygon. Proceed recursively (Fig. 4(a)).

Note that the procedure above uses a diagonal between v_t and v_b only if both lie to the left of the minimum *u*-valued inflection vertex between them and thus are visible to each other. Due to this advancing front like technique, high degree triangles are less likely to occur. Further, $u(v_t) - u(v_b)$ is small, as the corresponding monotone segment is short (see Fig. 4(c) for an example). Hence long and thin triangles of the kind shown in Fig. 4(c) do not occur. However, thin triangles can be generated due to horizontal trapezoidation, if two trapezoids vertically adjacent to each other are both thin (see Fig. 6(a)). This is a general shortcoming of our scheme since we avoid skinny triangles only during the second phase: monotone polygon triangulation. While it may be possible to devise an algorithm not based on trapezoidation, we have found the trapezoidation scheme to be very robust. It fails only if the input polygon is non-simple or almost non-simple. Hence, it is more appropriate to implement special cases for such (rare) thin monotone polygons generated by trapezoidation. In practice, we avoid diagonalizing thin trapezoids and thus obtain more than a single u-monotone chain. An extension of the monotone triangulation algorithm described above works for this case. Note also that sometimes fans are inevitable as shown in Fig. 4(b) – no other triangulations exist.

4.2 Simplicity Detection

Although it is not common for a polygon input to the triangulation algorithm to be non-simple, if left undetected, it can cause the triangulation to fail and display to become invalid. However, we pose the simplicity detection in terms of horizontal visibility lines used in the trapezoidation. Subsequently, a minor modification to the trapezoidation algorithm helps us detect if edges p_i-p_{i+1} and p_j-p_{j+1} of the polygon intersect. If they do, we compute extra samples on the curve between $t(p_i)$ and $t(p_{i+1})$ and between $t(p_{i+1})$ and $t(p_{j+1})$ and retry. Since polygons are rarely non-simple, the extra cost of sampling and iterating is acceptable. Simplicity check is straightforward after realizing that the horizontal line corresponding to some vertex of a non-simple polygon is *inconsistent*. We define inconsistency as follows:



(a: Case I) **(b: Case II)** **(c)**

Figure 5: Simplicity Verification

Definitions: A point q is *visible* to point p if the line segment pq does not intersect the given PSLG. Segment s is visible to p if a points on s is visible from p. Point p is on the *interior* with respect to an oriented segment p_i-p_{i+1} visible to p, if it lies on the right hand side of p_i-p_{i+1}. Also on the interior are all points r on the line segment pq, where q is the point on p_i-p_{i+1} such that $v(q) = v(p)$. A point is *inconsistent* if it is on the interior with respect to some edge of the polygon while on the exterior with respect to another. If a point is not *inconsistent* we call it *consistent*.

Clearly, each non-simple polygon has inconsistent points on its boundary. In fact:

Theorem: one of the polygon vertices, p_i, must be inconsistent, if the polygon is non-simple.

Proof: Note that a vertex is inconsistent if any point on the horizontal line through it is inconsistent. To sketch a proof, consider two edges p_i-p_{i+1} and p_j-p_{j+1} that intersect. There are four possible cases to consider. Two are shown in Figs. 5(a)

and 5(b), the other two are symmetric. Neighboring interior points are shown shaded for each edge. The black dots show examples of inconsistent points.

CASE I: If either p_{j+1} or p_{j+1} is horizontally visible from the other edge, we have found an inconsistent vertex. Otherwise, there must exist occluding edges and vertices. The vertex with the minimum v-value greater than $v(p)$, p being the point of intersection, must be visible to both p_i-p_{i+1} and p_j-p_{j+1} and is hence inconsistent.

CASE II: Consider the segment s, that is horizontally visible from p. If both the end points of s are visible from p, one of these must be inconsistent. Otherwise, there must exist minima and maxima vertices p_k and p_l on either side of the horizontal visibility line through p. (One of these points may lie on s.) If s is oriented upwards, p_k is inconsistent, otherwise, p_l is inconsistent. Strictly speaking, the argument above holds only if any given segment has only one intersection with the rest of the polygon and if no segments are horizontal. However, by using transitivity, the first restriction may be removed and by rotating the input (or topologically sorting it), the second restriction may be removed.

Thus, simplicity detection can be performed while constructing the horizontal visibility line for trapezoidation. The only remaining operation is to verify that the two ends of the visibility line are both in the exterior or both in the interior. (As a special case, for the minimum and maximum v-valued vertices, if the visibility line is locally in the interior, there must exist other segments visible along this visibility line. Fig. 5(c) shows a case in point with the corresponding smooth curve overlaid.)

5 Incremental Update

Before we explain the incremental update, we need to consider the data structure for triangulation. Since the number of surface patches can be quite large, we keep the size of data-structure cached per patch small. In addition to the triangle strips, for each *v-strip* on the domain, we maintain a list of triangle strips intersecting that *v-strip*, i.e., we keep a pointer to the first vertex of each entering triangle strip. In addition, if one of the strip-edges is also an edge of the trimming polygon, we mark it so. This structure is similar to the one proposed in [MSZ96], which allows point locations, in expected $O(n^{1/3})$ time. However, we do not maintain adjacencies between strips and cannot "walk" from a random triangle. We directly walk the appropriate strips. Another difference is that [MSZ96] maintains all adjacencies explicitly; we easily infer those from the strips. It is possible for us to retain pointers between adjacent strips as well, however since the number of strips crossing a cell is typically small, we do not lose much performance by explicitly searching for adjacent strips in any cell.

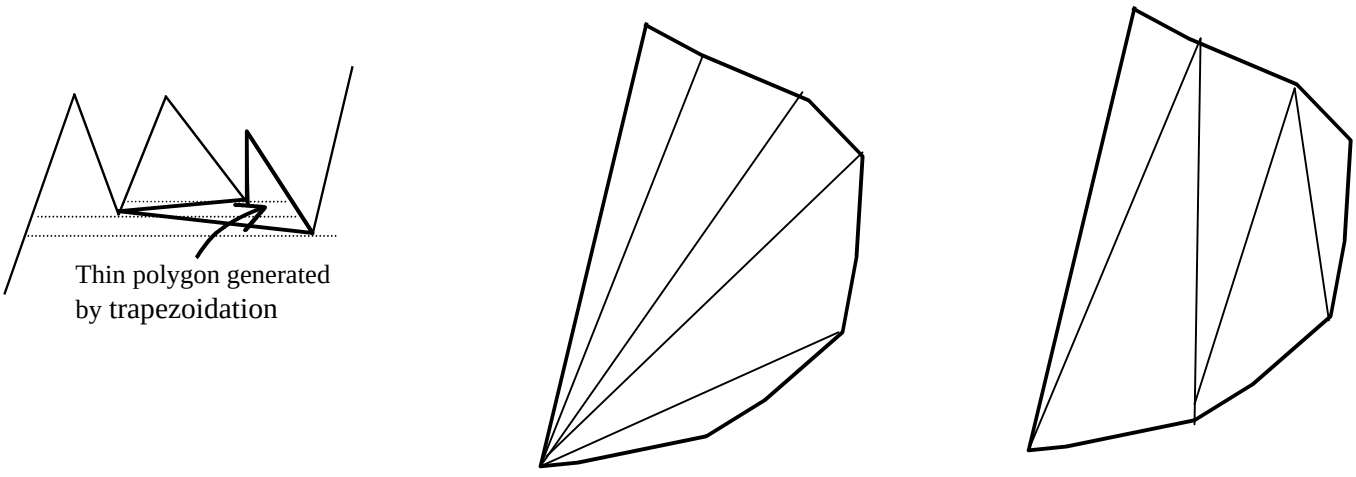
The types of updates to our triangulation is limited for our application (see [KM95,KML96] for details):

i) A segment p_i-p_{i+1} may be replaced by p_iq and qp_{i+1} (and vice versa), call these update segments.

ii) A grid line u_i (or v_i) could be added or deleted: call it the update line.

The incremental triangulation has the following main steps:

- Re-trace to compute new PSLG
- Delete triangles intersected by the update features (segment or line)
- Re-triangulate the hole



(a) Thin Trapezoids **(b) Triangle Fan (long and thin)** **(c) Our Triangle Strips**

Figure 6

The first two steps need some explanation. For all update types, we first need to determine if any quads are added or deleted. Thus, we need to re-trace the polygon. However, we need only re-trace the new segments in case (i). No other segments may change. In case (ii), we must retrace all segments intersected by the update line. By searching for strips in all cells intersected by the update line, we can determine the polygon edges that bound the edges intersected by the line. We only need to re-trace between these pairs of bounds. Any quads introduced by this re-tracing are diagonalized. A quad intersected by the new feature is deleted, if it contains an intersection with the trimming polygon, otherwise it is bisected. Similarly if the deletion of a feature results in the deletion of an intersection, a new quad is generated or two quads are merged. The quads added or deleted in the previous step are also included in the set of update features. Parts of a grid line that result in a quad bisection are deleted.

To locate an update feature in the current triangulation, we start randomly in each triangle-strip crossing the corresponding *v-strip* and walk across triangle boundaries until all triangles adjacent to the feature are located. A strip intersected by an update feature is split into two or more (disjoint) pieces. Each intersected triangle is deleted from the strip. We re-triangulate this hole (which is again a PSLG) using the algorithm described in Section 4.

6 Implementation and Performance

We augmented the publicly available implementation of Seidel's triangulation algorithm [NM95] with our algorithm and plugged it into a Bezier surface rendering system. We performed a suite of tests on a variety of Bezier surfaces on a Silicon Graphics Indigo 2 system with Maximum Impact graphics. We compared our results with those of [NM95] by plugging that in. [NM95] uses the algorithm by [FM84] for monotone triangulation. In practice, our implementation results in less than 10% slow-down in one-time triangulation. (Here we compare the times for the first time triangulation performing no incremental triangulation.) For the cost of this slowdown, our system generates far better triangles, and, more importantly, it never fails – the code does not crash and the result is always correct. Our triangulation is of significantly better quality: the smallest angles goes up from 11.3 degrees to 33.7 degrees on average. The degree of our triangulation is better as well (see Table 1). Furthermore, we generate triangle strips obtaining a rendering speed-up of more than 60% over triangles. A surface-patch needs re-triangulation in less than 20% of the frames on average, and is just re-rendered 80% of the time. The speed-up in triangulation obtained by using an incremental technique over one triangulating from scratch every time sampling changes is almost 90%. Also, note that our implementation is more robust due to the simplicity check. We have performed millions of triangulations without any failure using our implementation.

Model	Num. Patches	Triangulation time[1]	Minimum angle[1]	Avg. Degree[1]	Triang. time[2,3]	Triang. Time[2,4]	Minimum angle[2]	Avg. Degree[2]	Total frame time[2]
Brakehub	560	32	10.5	9.2	33	14	33.5	1.8	24
Torpedo	1201	59	16.1	11.1	62	28	34.2	2.1	42
Pivot	4101	61	14.9	10.8	68	38	31.1	2.2	66
TorpRoom	17032	131	8.4	14.4	143	81	25.3	3.9	110

1. [NM95] implementation: No simplicity check, skinny polygons, generated triangles
2. Includes simplicity check, fat polygons, triangle strip generation
3. First instance of the triangulation
4. Subsequent updates

Table 1: Performance of our Triangulation algorithm on an SGI Octane with 195MHz R10k. (Statistics are per frame averaged over more than 5,000 frames. Time is in milliseconds, angle is in degrees.)

7 Conclusion

We have presented a simple, robust, efficient and incremental algorithm for triangulating points on a surface. This includes both the generation of the polygons and their triangulation. To contrast our method of polygon generation with that of [KM95], that approach is cell based and attempts to find polygons around each cell. In spite of the higher overhead of this search, [KM95]'s method results in similar sized polygons. Furthermore, we do not need to handle a large number of special cases, nor do we need the clean-up stage, which can be quite slow. In addition to being more efficient, our algorithm directly produces triangle strips and generates better quality triangles. However, due to strip splitting procedure, the strips tend to become fragmented after a while. Currently we perform complete re-triangulation periodically. A slightly more complicated approach could avoid splitting strips by adding extra vertices in the middle. Our algorithm also verifies that a polygon is simple at little additional cost. While infrequent, if left undetected, a non-simple polygon can cause a system to crash. While it is possible to extend our incremental algorithm to perform constrained Delaunay triangulation, we believe the cost does not justify the minor benefit. One

limitation of our system is its independence from the surface parameterization. We only guarantee triangle quality in the domain. Thus our triangulations may still be long and skinny for severely skewed parameterizations. In our experience most surface models do not suffer from such skews. Our algorithm works well in the common cases.

References

[AES93] S.S. Abi-Ezzi and L.A. Shirman.The scaling behavior of viewing transformations. *IEEE Computer Graphics and Applications*, 13(3):48—54, 1993.

[AES94] S. Abi-Ezzi and S. Subramaniam, Fast Dynamic Tessellation of Trimmed NURB Surfaces, *Computer Graphics Forum, 13(3),* 107—126, 1994. (Eurographics '94.)

[BDE92] M. Bern, D. Dobkin and D. Eppstein. Triangulating polygons without large angles. *Proc.* 8th *Annual ACM Symp. Comput. Geom.*, pages 222—231, 1992.

[BE92] M. Bern and D. Eppstein. Polynomial-size nonObtuse triangulation of polygons. *Proc.* 7th *Annual ACM Symp. Comput. Geom.*, pages 342—350, 1991.

[CH90] C. Cherfils and F. Hermeline. Diagonal swap procedures and characterizations of 2d-delaunay triangulations. *Math. Modeling and Num. Analysis* , 24(5):613—625, 1990.

[Cha91] B. Chazelle. Triangulating a simple polygon in linear time. *Discrete Comput. Geom.* , 6:485—524, 1991.

[CI84] B. Chazelle and J. Incerpi. Triangulation and shape-complexity. *ACM Trans. Graph.* , 3(2):135—152, 1984.

[CTV89] K. Clarkson, R. E. Tarjan, and C. J. Van Wyk . A fast Las Vegas algorithm for triangulating a simple polygon. *Discrete Comput. Geom.* , 4:423—432, 1989.

[EKA84] M. Edahiro, I. Kokubo, and Ta. Asano. A new point-location algorithm and its practical efficiency: comparison with existing algorithms. *ACM Trans. Graph.* , 3:86—109, 1984.

[FM84] A. Fournier and D. Montuno. Triangulating simple polygons and equivalent problems. *ACM Transactions on Graphics* , 3:153—174, 1984.

[GJPT78] M. R. Garey, D. S. Johnson, F. P. Preparata, and R. E. Tarjan. Triangulating a simple polygon. *Inform. Process. Lett.* , 7:175—179, 1978.

[HM83] S. Hertel and K. Mehlhorn. Fast triangulation of simple polygons. In Proc. 4th Internat. Conf. Found. Comput. Theory , volume 158 of Lecture Notes in Computer Science , pages 207—218. Springer-Verlag, 1983.

[KM95] S. Kumar and D. Manocha. Efficient rendering of trimmed NURBS surfaces. *Computer-Aided Design* , 27(7):509—521, July 1995.

[KML96] S. Kumar, D. Manocha, and A. Lastra. Interactive display of large NURBS models. *IEEE Transactions on Visualization and Computer Graphics* , 2(4):323—336, Dec 1996.

[Kum96] S. Kumar. Interactive Display of Parametric Spline Surfaces. *Ph.D. Thesis*, University of North Carolina, 1996.

[LC93] W.L. Luken and Fuhua Cheng. Rendering trimmed NURB surfaces. *Computer science research report* 18669(81711), IBM Research Division, 1993.

[Mit93] S. Mitchell. Refining a triangulation of a planar straight-line graph to eliminate large angles. *Proc.* 34th *Annual IEEE Symposium on Foundation of Computer Science (FOCS).* Pages 583—591, 1993.

[MSR94] M. Bern, S. Mitchell and J.Rupert. Linear-Size nonobtuse triangulation of polygons. . *Proc.* 10th *Annual ACM Symp. Comput. Geom.*, pages 221—230, 1994.

[MSZ96] E. M"ucke, I. Saias, and B. Zhu. Fast randomized point location without preprocessing in two- and three-dimensional Delaunay triangulations. In *Proc. 12th Annual ACM Symp. Comput. Geom.* , pages 274—283, 1996.

[NM95] A. Narkhede and D. Manocha. Fast polygon triangulation based on Seidel's algorithm. In A. Paeth, editor, *Graphics Gems V*, Academic Press, 1995.

[PR95] L. Piegl and A. Richard. Tessellating trimmed NURBS surfaces. *Computer Aided Geometric Design* , 27(1):16—26, 1995.

[RHD89] A. Rockwood, K. Heaton, and T. Davis. Real-time rendering of trimmed surfaces. *ACM Computer Graphics*, 23(3):107—117, 1989. (SIGGRAPH Proceedings).

[RR94] R. Ronfard and J. Rossignac. Triangulating multiply-connected polygons: A simple, yet efficient algorithm. *Comput. Graphics Forum* , 13(3):C281—C292, 1994.

[Sei91] R. Seidel. A simple and fast randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Computational Geometry Theory & Applications* , 1(1):51—64, 1991.