Simple yet robust triangulation for polygons containing multiple holes for real-time 3D industrial applications

Triangulación simple y robusta para polígonos con múltiples orificios en aplicaciones industriales 3D de tiempo real

Yuriy Kotsarenko

Afterwarp Interactive
yunkot@gmail.com

KEYWORDS:

Polygon triangulation, Ear clipping algorithm, Polygon with holes, Real-Time visualization

ABSTRACT

In industrial applications such as CAD modeling, manufacture or automated systems, it is common to work with data models that are represented by polygonal shapes, or models that produce polygonal shapes out of complex geometry defined by lines and curves. However, this data has to be converted to a triangular mesh before it can be processed and/or rendered by the GPU. Existing solutions that generate triangular mesh out of polygonal shapes either do not support holes or have limitations on how many holes can be present at the same time. Most modern advanced solutions need considerable effort to implement, debug and maintain, which involves significant development costs.

In this work, an alternative solution is proposed, which is relatively simple to implement yet is sufficiently robust to handle all possible input scenarios handling any number of holes, or inner polygons in an outer polygon, assuming that polygons do not intersect each other or themselves, and makes no assumptions about the winding order of polygon vertices. The proposed solution involves initial pre-processing work, merging inner polygons into an outer polygon, and then performing polygon triangulation using one of the two proposed variations of an Ear Clipping algorithm. The proposed solution is shown to handle practically an unlimited number of holes in polygon independently of its shape. Quality and performance comparison between two techniques is also provided and discussed, along with the images of a real life CAD application, which implements the proposed solution.

PALABRAS CLAVE:

Triangulación de polígonos, Algoritmo ear-clipping, Polígono con agujeros, Visualización en tiempo real

RESUMEN

En aplicaciones industriales tales como el modelado CAD, la industria de fabricación o los sistemas automatizados, es común trabajar con modelos de datos que representan contornos poligonales, o con los modelos que general polígonos a partir de una geometría compleja que se define mediante líneas y curvas. Sin embargo, para que estos datos pueden ser procesados y/o rendereados por GPU, los datos se tienen que convertir en una serie de triángulos. Las soluciones existentes que general triángulos a partir de contornos poligonales no son capaces de trabajar con orificios o se limiten a un cierto número de orificios que pueden estar presentes al mismo tiempo. La mayoría de las soluciones modernas avanzadas requieren de un esfuerzo considerable para su implementación, depuración y mantenimiento, lo que a su vez produce ciertas implicaciones importantes en cuestión de costos de desarrollo.

En este trabajo, una solución alternativa se propone, siendo relativamente fácil de implementar y al mismo tiempo suficientemente robusta para manejar cualquier posible entrada de datos, incluyendo cualquier número de orificios e incluso cualquier número de polígonos internos, siempre y cuando los polígonos no se cruzan entre sí ni se auto-cruzan a sí mismos. La solución propuesta tampoco depende de un orden determinado de los vértices del polígono, solo necesita que este orden sea consistente. La solución propuesta involucra un pre-procesamiento inicial, donde los polígonos internos se integran a los polígonos externos. Una vez hecho esto, se hace una triangulación de polígonos utilizando una de las dos variaciones de técnicas propuestas que se basan en algoritmo de "Ear Clipping". En este trabajo se muestra que la solución propuesta puede trabajar con el número de orificios prácticamente ilimitados independientemente de su forma. Este trabajo también incluye una comparativa de evaluación de calidad y desempeño entre las dos técnicas propuestas, incluyendo unas imágenes de una aplicación CAD que utiliza estas soluciones de manera práctica.

• **Recibido: 7 de agosto de 2023** • **Aceptado: 18 de noviembre de 2023** • **Publicado en línea: 1 de febrero de 2024**

DOI: 10.30973/progmat/2024.16.1/2

1. INTRODUCTION

In industrial applications such as construction, manufacture or automated systems, it is common to work with complex objects, diagrams and/or obstacles, which are often described in terms polygonal shapes. In other situations, an object or a model may be described in terms of primitives such curves and arcs, which can then be approximated to a polygon using certain degree of accuracy by subdividing the primitive into line segments of finite length. The resulting polygons may have irregular, strange looking shapes and may even contain holes, which would represent irregular polygon themselves. One such example could be a real-time Computer Aided Design (CAD) application that works with vector and/or 3D graphics has drawings and text represented as a series of curves forming closed shapes with holes. Another example would be an electronic circuit design software, working with a data model of printed circuit board plate with holes, interconnections and electronic components. In order to present the information visually using Graphics Processing Unit (GPU), both applications would need to convert their representations into triangular meshes.

As a result, it is very common in the industry to deal with a problem of converting one or more polygonal shapes into a series of triangles. This is called *triangulation* and can be formally defined as the decomposition of a polygonal area into a set of triangles [1][2]. Depending on how many polygons need to be triangulated, their shape, complexity and whether or not they have holes, the triangulation process may vary greatly from being trivial to quite difficult problem with a large number of possible solutions, some of which could be satisfactory for a given particular situation, whereas others would be unacceptable.

For the purposes of this work, a polygon is defined by a series of points, where each pair of consecutive vertices, along with first and last ones, are connected by edges. As long as edges do not intersect each other except at their respective vertices and each vertex shares exactly two edges, such polygon is called a *simple polygon*; otherwise, it is called a *complex polygon*.

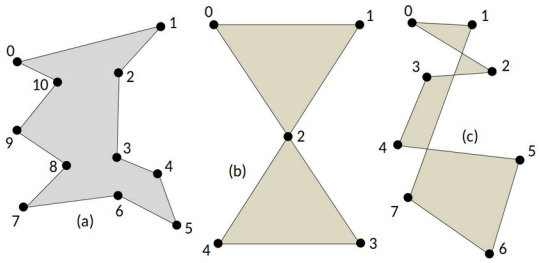

Figure 1. A simple polygon (a) and two complex polygons: (b) and (c).

On Figure 1, three different polygons are shown. The first polygon (*a*) is simple because it fulfills aforementioned conditions. The second polygon (*b*) is not simple because its vertex at the center is shared by more than two edges, and the last polygon (*c*) is not simple because there are multiple intersections between the edges at points that are not vertices. Also, this last polygon presents additional challenges because it requires generation of new vertices where edges intersect and such newly generated vertices themselves will also be shared by more than two edges. In the scope of this work, only simple polygons will be considered. In polygon on Figure 1(*a*), vertices {0, 1, 4, 5, 7, 9} are considered *convex*, whereas vertices {2, 3, 6, 8, 10} are *reflex* (also known as *concave*). A vertex is considered *convex*, when its interior angle is smaller than π radians and the line segment between its neighboring vertices lies completely inside the polygon; otherwise, the vertex is considered *reflex*. A polygon with n vertices may have up to n-2 vertices that are reflex and up to n vertices that are *convex*. If a simple polygon has all vertices *convex*, such polygon is called *convex polygon*. A triangulation of a polygon with n vertices produces n-2 triangles [3]. A convex polygon

can be trivially triangulated by choosing a vertex and forming edges from that vertex to all other vertices of the polygon (also called "fan triangulation" or "tri-fanning")[1]. Furthermore, a simple polygon with only one *reflex* vertex can also be trivially triangulated as long as all edges are formed from the *reflex* vertex.

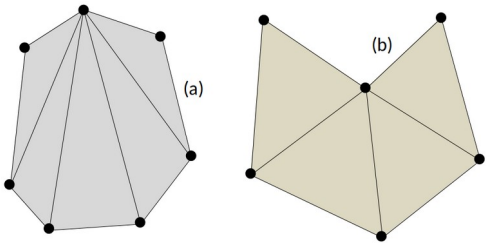

Figure 2. A convex polygon (a) and non-convex polygon (b) with only one reflex vertex, both trivially triangulated.

A simple polygon with two or more *reflex vertices* can be triangulated iteratively using a so-called *Ear Clipping* algorithm. An *ear* of a polygon is defined as a triangle formed by three consecutive vertices, the middle of which is a *convex* vertex, also called *tip of the ear*, with two sides being the edges of the polygon and third side being completely inside the polygon. It has a property that no polygon vertices are contained within its triangle other than the triangle vertices themselves. According to Meister's theorem, any simple polygon with at least four vertices without holes has at least two *ears* [4]. *Ear Clipping* algorithm works by iterating through the consecutive vertices, identifying whether they are *convex* or *reflex*, and if a vertex is found to be *convex*, it is tested to fulfill the ear property. Once an *ear* has been identified, the triangle formed by the *ear tip* and its neighbors is added to the final list of triangles, and the *tip of the ear* itself is removed from the polygon. A newly formed smaller polygon will continue to meet the "two ears condition". The process is then repeated until there is only one triangle remaining. A naïve implementation of *Ear Clipping* algorithm has time complexity of $O(n^3)$, but it can be optimized to run in $O(n^2)$ time [3][5] and with certain assumptions can be improved further to have $O(k \cdot n)$ time complexity, where k-1 is number of *reflex* vertices in polygon [6]. A limitation of this algorithm is that it does not handle polygons with holes, let alone multiple polygons with holes that may contain other filled polygons inside. Solutions to support holes with *Ear Clipping* algorithm have been described in varying degrees of detail. For instance, both Mei et al. [7] and Eberly [8] propose merging inner and outer polygons via coincident edges. However, there are many undocumented caveats when dealing with polygons and their holes having parallel edges, or vertices that end up being collinear when connected, as well as other edge cases that produce self-intersecting polygons that *Ear Clipping* algorithm cannot handle.

There are other, more advanced algorithms that can triangulate polygons, some of which can handle complex polygons with holes and self-intersections, most notable being *Sweep Line* based algorithms [9] and algorithms such as *Constrained Delaunay Triangulation* [10], among others. However, these algorithms require more advanced data structures, while being more difficult to implement, debug and maintain, which implies higher development and maintenance costs. Also, these algorithms similarly to *Ear Clipping* algorithm, handle edge cases with varying degree of success, but due to increased complexity, the triangulation problems may be very difficult to debug and correct. Therefore, for triangulation, *Ear Clipping* algorithm was chosen for this work because of its efficiency, accuracy and low memory requirements [11], while most effort was focused on solving the problem with supporting polygons that can have an unlimited number of holes inside.

1 Fan triangulation, although being very simple to achieve, may not necessary produce a quality triangulation as it may produce triangles with very sharp edges, also called *silver triangles*. However, depending on polygon's shape and vertex positions, such triangles might be unavoidable unless new vertices are created during the triangulation process, which may have certain design consequences and implications, which are out of the scope of this work.

2. TRIANGULATION

For this work, two variations of an *Ear Clipping* algorithm were developed:

a) A high-performance approach, and
b) Quality approach.

A simple data structure consisting of a circular double-linked list with custom memory allocator is used to store vertex indices and their classification, which is stored linearly in memory for all of the inner and outer polygons. This improves CPU cache locality, which helps to boost overall performance. Each polygon has a pointer to an element in the circular double-linked list. The linked-list nodes are pre-allocated and when deleted, are simply marked as unused. The deallocation may occur after triangulation is finished, or the allocated memory can be re-used for future invocations. Polygon vertices are provided as a separate array of coordinates. Both of the approaches involve the following steps:

1. For each of the input polygons, populate its corresponding vertex indices into the nodes of circular double-linked list.
2. For each polygon, iterate through all nodes and classify all vertices as convex or reflex: for vertices that are found to be neither, the corresponding nodes are removed from the double-linked list and added to global list of removed vertex indices. Later on, such list can optionally be used to eliminate these vertices from polygons themselves with posterior remapping of resulting triangle indices.
3. For each polygon, iterate through all nodes to search for a right-most vertex (that is, vertex with highest x coordinate). If multiple vertices share the same x coordinate, then a vertex with lower y coordinate is chosen.
4. Once a right-most vertex for each polygon is found, check whether it is classified as convex or reflex. If it is classified as convex, then set polygon's winding order as positive. Otherwise, set polygon's winding order as negative and flip all its vertex classifications from convex to reflex and vice-versa.

After performing the aforementioned steps, a winding order is known for each of the polygons, duplicate and collinear vertices have been excluded from processing and all vertices have been classified. If there is a single polygon, then triangulation can be performed immediately and the algorithm terminates. Otherwise, inner polygons representing the holes have to be identified by doing polygon in polygon containment test, and iteratively merged into the outer polygon one at a time. After this, a single polygon will remain, the triangulation of which can then be performed. The concrete steps depend on the approach taken. For high-performance approach (*a*), the steps are the following:

1. Set walking position to start (or any existing node) in circular-double linked list.
2. If there is only one *reflex* vertex left, perform tri-fanning using the remaining reflex vertex as origin and terminate.
3. If no *reflex* vertices are remaining (or only three vertices left), perform tri-fanning using the current vertex as origin and terminate.
4. Continue advancing from the given position until a *convex* vertex is found, that was not previously marked as "*not an ear*". If the whole list has been cycled once and no such vertex was found, the polygon is likely not simple, so algorithm terminates with error.
5. Test the found *convex* vertex to see if it is an *ear*. If it is not, mark it as "*not an ear*" and go to step 4.
6. *Clip the ear* by generating indices for the appropriate triangle and removing the *ear tip* from the list of nodes and advancing the current position appropriately. Check both neighbors if they were classified as *reflex* and if so, re-classify them, otherwise check if they were marked as "*not an ear*" and if so, delete this remark, so they will have to be *ear*-tested again next time.

7. Go to step 2.

As it can be seen from the aforementioned steps, once an *ear* has been identified, it is clipped immediately, so there is no need to store a list of ears. In fact, a circular double-linked list can also be avoided altogether if there is no need to support holes, as the vertices can be iterated by advancing a set of three indices {*previous*, *current*, *next*}. For many practical data sets, this algorithm reduces the problem complexity quickly to a trivial triangulation, while also being cache-friendly due to data linearity and close proximity of nodes in memory.

For quality approach (*b*), the steps are the following:

1. Walk through all nodes and test each *convex* vertex to see if it is an *ear*: if so, calculate its interior angle, otherwise, mark it as a "*not an ear*".
2. If there are three vertices left, add them to a list of triangles and terminate.
3. Search for an ear (that is, a *convex* vertex that is not marked as "*not an ear*") with the smallest interior angle. If there is none, the polygon is likely not simple, so algorithm terminates with error.
4. *Clip the ear* by generating indices for the appropriate triangle and removing the *ear tip* from the list of nodes. Re-classify both neighbors to see if they are *convex* or *reflex*. If a neighbor has become *convex*, or has previously been marked as "*not an ear*", re-test it to be an *ear* and if so, re-calculate its interior angle; otherwise mark it as "*not an ear*".
5. Go to step 2.

The quality approach uses an improved *Ear Clipping* algorithm described by Mei et al. [7], which produces higher-quality triangulations at the expense of performance. Although the amortized worst-case time complexity is still $O(n^2)$, all ears have to be tested before any clipping is to be performed, plus a search for an ear with smallest interior angle is also performed iteratively in the loop, leading to $O(n^2)$ time complexity even for *convex* polygons. The difference in resulting triangulation is illustrated below.

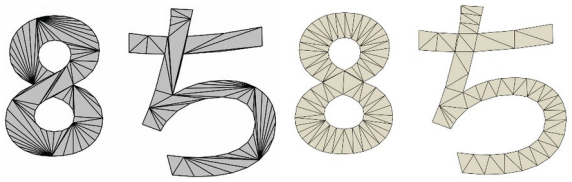

Figure 3. A digit "8" and a Japanese letter "Chi" triangulated using (left) high-performance and (right) quality approaches.

On Figure 3, four polygons are triangulated at the same time using both techniques: a letter "8" is described by an outer polygon and two inner polygons representing its holes (all merged into a single *master* polygon), whereas the Japanese letter "chi" is described by a single polygon. The triangles resulting from the triangulation using *quality approach* are generally more desirable in practice because they have less sharp corners than the resulting triangles of high-performance approach. Triangles with sharp corners, or *silver triangles*, can have precision issues in some calculations due to their higher slope. However, as far as rendering is concerned, both look visually equivalent: if the lines on Figure 3 would not be visible, the results from both techniques would look exactly the same pixel-wise.

3. HOLE MANAGEMENT

One of the limitations of traditional Ear Clipping algorithm is that it does not support holes. However, assuming that inner polygons describing the holes are specified in reverse winding order[2] of the outer polygon, it is possible to combine inner polygons into outer polygon by connecting two directly visible vertices from inner to outer polygon, which would form two coincident edges [8]. This results in a single polygon, which can then be triangulated. The process of choosing two mutually visible vertices.

2 A winding order refers to how the vertices are specified in 2D space, either clockwise or counter-clockwise.

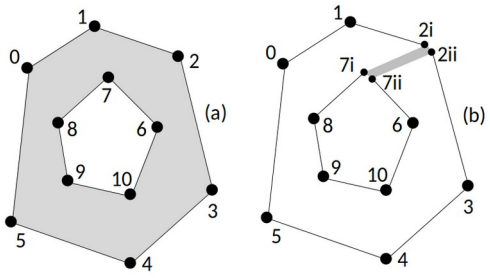

Figure 4. An outer polygon with an inner polygon representing a hole, both merged in to form a single polygon (a). The second polygon (b) illustrates how the actual connection is made by using two coincident edges.

As it can be seen on the above figure, an outer polygon formed by a group of vertices [0, 5] is merged with an inner polygon formed by a group of vertices [6, 10]: two additional edges $\{2_i, 7_i\}$ and $\{2_{ii}, 7_{ii}\}$ are generated by creating a duplicate pair of vertices 2 and 7. A new polygon has vertices $\{0, 1, 2, 2_i, 7_i, 8, 9, 10, 6, 7_{ii}, 2_{ii}, 3, 4, 5\}$. Since the inner polygon had its vertices specified in counter-clockwise order, whereas outer polygon had vertices specified clockwise, the newly created polygon continues to have the same winding as before, which can be verified by walking through the list of vertices sequentially on Figure 4(b). Therefore, the newly formed polygon can be triangulated by Ear Clipping algorithm. Generally speaking, once all holes have been identified and merged into the outer polygon, as long as the integration process has been performed correctly, the following triangulation should be successful. The integration process is crucial, as incorrect merging would produce a self-intersecting polygon, which would make it unsuitable for triangulation.

In order to merge inner polygon hole, it is important to identify a pair of vertices: one from inner polygon and one from outer polygon, where connection is to be made. A required property of such pair of vertices is that they have to be *mutually visible*. *Mutually visible vertices* can be defined as pair of vertices from two different polygons that can be connected by a line segment and such line segment will not intersect any other vertices or edges from both polygons. Finding closest visible vertex between two polygons is a well-studied problem with solutions having sequential time complexity no worse than $O(n \log n)$ and up to $O(n)$ depending on types of polygons involved [12][13][14][15][16]. In their recent work, Mei et al. [7] use a simple approach by iterating through all vertex pairs between inner and outer polygon, leading to a time complexity between $O(n^2)$ and $O(n^3)$ depending on the actual implementation.

In this work, most focus was given on a solution proposed by David Eberly [8], but with certain clarifications in the procedure:

1. From the inner polygon, take right-most vertex *M* with highest x value[3], which was previously calculated as part of initial preparation work.
2. Calculate intersections between all edges of outer polygon that are located to the right of *M* (that is, at least one of the edge's vertices has x bigger than x_M) and a horizontal ray coming from vertex *M*. Select point *I* among all intersections calculated this way to be the closest visible point to *M* on this ray.
3. If *I* is a vertex of the outer polygon, then *M* and *I* are mutually visible and the algorithm terminates.
4. Otherwise, *I* is an interior point of the edge in outer polygon. Select *P* to be one of the vertices that has highest x value. If both edge vertices have same x value, then choose vertex that has lowest y value.
5. Test the *reflex* vertices of the outer polygon, excluding *P* if it happens to be reflex, that are on the right side of the vertex *M*, to see if they are within the triangle formed by {*M*, *I*, *P*}. If all of them are strictly outside the triangle, then *M* and *P* are mutually visible and the algorithm terminates.
6. Otherwise, select one *reflex* vertex *R* among aforementioned ones that lies inside the triangle {*M*, *I*, *P*} that minimizes the angle between {*M*, *I*} and {*M*, *R*}. If there

3 The choice of min/max and axis is arbitrary: instead of maximum X value, it can be maximum Y, or minimum on either of the axes. For instance, Wijeweera et al. [11] uses maximum Y, which is also valid: it just requires changing all steps in this work analogously.

are multiple vertices with a similar angle within certain threshold, then select vertex that is closest to M. Vertices R and M would be mutually visible and the algorithm terminates.

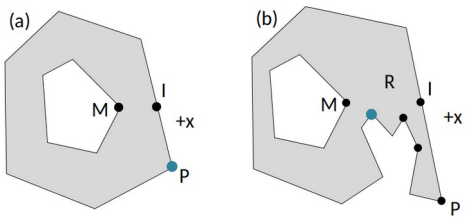

Figure 5. a) Right-most vertex P on intersected edge is directly visible and b) Right-most vertex P is not directly visible, but there are three reflex vertices inside triangle {M, I, P}, one of which (drawn in color) is determined to be mutually visible with M.

The aforementioned approach is relatively simple to implement and quite efficient. It requires n_1 steps calculating intersections with edges of the outer polygon and n_2 steps testing reflex vertices, leading to an amortized time complexity of O(n) for any types of polygons involved. In addition, a right-most vertex that was previously found to determine the winding order can also be reused for merging, without the need of an additional search.

3. HANDLING MULTIPLE HOLES

When a polygon has multiple holes, they can be merged iteratively one at a time using aforementioned approach, choosing inner polygon with a right-most vertex that has highest x coordinate each time. In case two or more inner polygons have right-most vertex with the same x coordinate, then the one is chosen that has lowest y coordinate first.

This iterative approach would not work, however, for many common situations such as shown on Figure 6: first, a polygon with right-most vertex A is merged into the outer polygon, with closest visible vertex determined to be P as shown on image (*a*). This would split vertex P into P_1 and P_2, and vertex A into A_1 and A_2; second, a polygon with right-most vertex B is merged into the outer polygon, the closest intersection point will lie on both coincident edges {P_1, A_1} and {A_2, P_2}. If an intersection on edge {A_2, P_2} is chosen, which would select P_2 as closest visible vertex, then the resulting connection with produce a self-intersecting polygon.

Furthermore, as it can be seen on an illustrative image Figure 6(*b*), the correct closest intersection point should definitely lie on edge {P_1, A_1}, but since both edges are coincident, this cannot be determined numerically. A solution to this problem requires detecting coincident edges during intersection calculation, which should be trivial, as both edges would refer to the same vertex indices in the polygon, except in opposite order. If an intersection with a second coincident edge is detected, then a *signed triangle area* should be calculated the hole's right-most vertex and two vertices of the edge[4] and the preference should be given to the edge, that has the sign of *signed triangle area* matching the winding order of the outer polygon. In other words, as in case of Figure 6(*b*), point B must be located *to the right* of intersecting edge, in which case it would be {P_1, A_1} (if one is standing directly at point P_1 looking in the direction of point A_1); accordingly, in case of edge {A_2, P_2}, the point B will appear on left side.

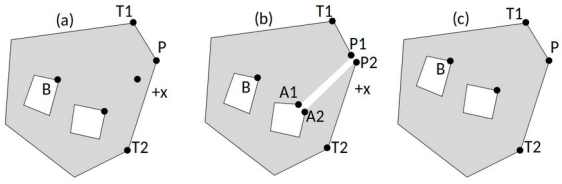

Figure 6. A hole with right-most vertex B is merged into outer polygon, which had previously hole with right-most vertex A (a) and the same situation (b), where two coincident edges are separated for illustrative purposes and (c), where intersection happens directly at vertex P.

However, there is another situation, which also requires attention: if an intersection like on Figure 6(c) would happen directly at vertex

4 This *signed triangle area* is also used to calculate whether a particular vertex is *convex* or *reflex*.

P, there would be actually four intersecting edges: $\{T_1, P_1\}$, $\{P_1, A_1\}$, $\{A_2, P_2\}$ and $\{P_2, T_2\}$. Two of these edges would give point P_1 as visible vertex, whereas other two would give point P_2. In other words, edges $\{P_1, A_1\}$ and $\{A_2, P_2\}$ will be handled according to the strategy described above, but the other two would still result in an ambiguous situation that could potentially produce a self-intersection. A solution to this problem is to detect when another edge has intersection point within certain minimum threshold of the current intersection (in other words, very similar x coordinate), then choose an edge that minimizes the angle between an edge in question and a line segment between right-most vertex and the intersection point. In case of current situation, this would give preference to coincident edges $\{P_1, A_1\}$ and $\{A_2, P_2\}$, which would be resolved according to strategy above (which should have higher preference priority than the angle test as both edges would produce the same angle). It can be observed on Figure 6(*c*), using the minimum interior angle criteria, edge $\{P, T_2\}$ will always be the least preferred because it will always have interior angle higher than the one of coincident edges. If point T_1 is moved so edge $\{T_1, P\}$ will have smallest interior angle, then it would still result in proper duplicate vertex P (P_1 in previous image) to be selected as mutually visible vertex.

4. RESULTS AND DISCUSSION

In this work, both of the approaches have been implemented in C++ and integrated into a commercial framework for development of industrial CAD applications. The triangulation with holes provided by both of the approaches after testing with existing data sets seems to be very robust: as long as inner and outer polygons do not intersect each other and/or themselves, the techniques succeed with an accurate triangulation, even when working exclusively with 32-bit floating-point data types. The solutions proposed in this work were developed during the debug process of the initially prototyped technique, which was failing for many test cases. However, with the proposed solutions, the technique satisfies all the requirements for production use in industrial environments. As the number of data nodes can be known based on the number of vertices and number of polygons, the working memory can be pre-allocated and then re-used for all consecutive invocations, which rules out any potential memory fragmentation issues, so the final application can be left running for an indeterminate amount of time without the need of restart. Some of the triangulation results are shown below.

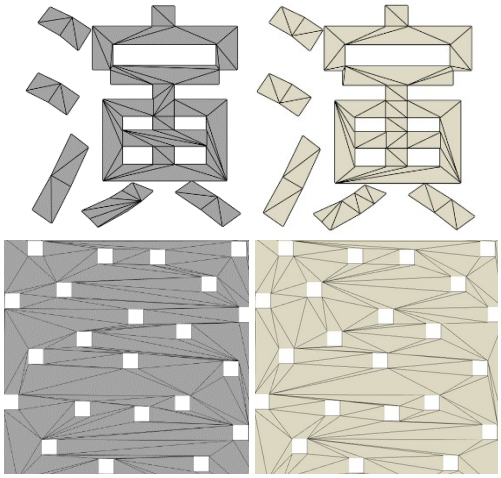

Figure 7. Chinese letter “Yǎn” and a rectangle with many randomly placed rectangular holes triangulated using high-performance (dark gray) and quality approaches (beige). .

As it can be seen on the above Figure 7, not all situations benefit from additional performance costs involved with the *quality* approach. When very simple polygons are used such as triangles or rectangles and the techniques do not produce any new vertices, there are too few existing vertices to choose from, so the resulting triangulation will produce sharp triangles no matter what approach is used. Therefore, while a Japanese letter “Chi” from Figure 3 contains smooth curves and benefits from quality approach, a Chinese letter “Yǎn” from above image has mostly flat contours and results in a

satisfactory triangulation even with high-performance approach.

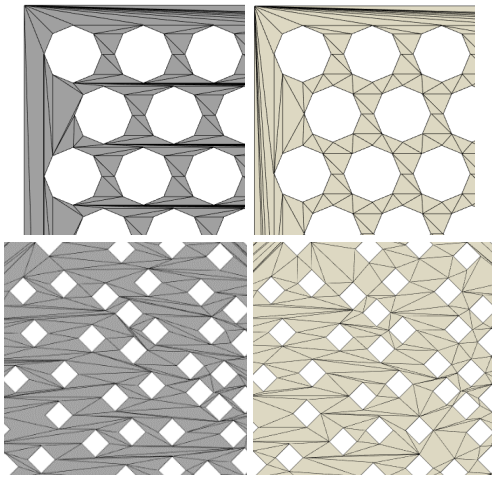

Figure 8. A rectangle with two different types of holes triangulated using high-performance (dark gray) and quality approaches (beige).

A spatial orientation of the rectangles does not seem to affect the situation as it can be seen on the above Figure 8 on last two images. However, as number of vertices increase, the *quality* approach starts to produce much better triangulation results as it can be seen on the first two images from the aforementioned figure.

Table 1. Performance benchmarks between high-performance and quality techniques from four different data sets.

Variant	Polygons	Vertices	Triangles	Speed (triangles/sec)
High-perf	2	16	16	**24,242,424**
Quality				**11,034,482**
High-perf	70	1778	1692	**14,461,538**
Quality				**4,327,365**
High-perf	604	11344	10721	**6,742,767**
Quality				**3,089,625**
High-perf	106	6724	6932	**110,912**
Quality				**41,633**

A couple of sample performance benchmarks between two variants used in this work were performed, involving four different data sets, to give an idea of expected throughput. The application was compiled using GNU GCC compiler version 11.2 using optimization level 3 and executed on a machine with AMD Ryzen 3950X processor at stock speeds, with 128Gb of DDR4 RAM running at 3600 Mhz. The tests were done using only a single core. A first data set was a simple letter "A" from Segue UI font. A second data set was a portion of "Lorem ipsum" text. A third data set was a big portion of text from Japanese version of "Lorem ipsum" text and a fourth data set used a rectangle with many high-quality circles inside similar to those visible on Error: no se encontró el origen de la referencia. As it can be seen from the results shown on Table 1, high-performance variant for the most common polygon shapes produces a whooping several millions of triangles per second. As for the fourth data set, it is unique because more than 99% of its vertices are *reflex* (all except for the 4 points at rectangle's corners), which results in a perfect worst-case scenario for both algorithm variants, which due to their $O(n^2)$ worst-case time complexity struggle in this specific scenario.

The following images are taken as screenshots from an actual CAD application, which uses the technique to produce 3D meshes using an extrusion technique: once a triangular mesh has been produced, any unused vertices that were identified as duplicate or collinear, the nodes of which were initially removed, are excluded from the polygons. Following this, side triangles are generated by iterating through the polygons and calculating the appropriate normals. An angle between two consecutive edges is used to determine if vertex normal can be re-used between two consecutive rectangular sections consisting of two triangles and if not, a triangle normal is used for "flat" sides.

Figure 9. A screenshot of the proposed solution used in a real CAD application, which includes a rectangle with many circular holes as well as 3D text extruded from a triangulated mesh.

The resulting image that can be seen on the Figure 9 using either of the techniques proposed in this work is visually indistinguishable, even when a color gradient is produced by using different vertex colors calculated according to their spatial position, when rendered on GPU. However, this may not always be the case, for instance, when performing a software rasterization using limited precision arithmetic or doing per-vertex lighting.

Figure 10. A screenshot of the proposed solution used in a real CAD application, which shows a phrase in Japanese language rendered as a 3D text model, generated by using extrusion from a triangulated mesh.

Finally, as it can be seen on the above Figure 10, non-trivial character glyphs such as those from Japanese language, containing multiple holes, can be triangulated accurately, producing a high-quality visualization of the resulting 3D text.

5. CONCLUSION AND FUTURE WORK

It is common in the area of industrial applications, for visual real-time software to work with data models that can be described using polygonal shapes. However, many algorithms that run on GPU expect the data to be provided in form of triangles. Therefore, the polygons need to be converted into triangles using a process called *triangulation*. Many existing solutions exist that enable polygon triangulation: simpler ones cannot handle holes at all, while most modern solutions require advanced data structures, requiring significant implementation and debugging effort, which increases development costs.

In this work, an alternative solution based and improved upon an *Ear Clipping* algorithm is proposed, with two variations of the technique that can meet different requirements based on speed and quality. It supports complex shapes defined in form of an outer polygon and one or more inner polygons that describe its holes, assuming that none of the polygons intersect each other or themselves. The proposed solution supports any number of holes in polygon and, as it has been shown in the experiments, produces accurate results even when implemented exclusively using 32-bit floating-point arithmetic.

The proposed solution has been integrated into an existing commercial framework for development of industrial applications with 3D visual content. This work includes images from a real CAD application that uses both variations of the proposed techniques to produce an actual 3D meshes that can be rendered directly on the GPU.

Future work involves further improvements in the proposed solution to reduce number of arithmetic operations and increase real-time performance. This is especially important when the polygon to be triangulated has large number of *reflex* vertices, triggering worst-case scenario with time complexity of $O(n^2)$. One area of interest for further investigation is to leverage trapezoidal decomposition such as described by Seidel [17] for the aforementioned use-cases, which significantly improves time complexity and could potentially increase the real-time performance significantly.

6. ACKNOWLEDGMENTS

This work has been made in memory of my mother, Dra. Prof. Svitlana Koshova (1941 – 2022), who has always been an inspiration for me as a brilliant scientist, a good colleague and a best mother I could ever have. She has been very supportive both in my professional work and in in real life, providing guidance

and advice always at the right moment and the right place. Dra. Koshova is missed by her sons and grandsons. May she rest in peace.

REFERENCES

[1] Garey, M.R., Johnson, D.S., Preparata, F.P., Tarjan, R.E. Triangulating a simple polygon. *Information Processing Letters*, 1978, 7(4), 175-179. doi: 10.1016/0020-0190(78)90062-5.

[2] Berg, M., Cheong, O., Kreveld, M., Overmars, M. *Computational Geometry*. Berlin: Springer, 2008. doi: 10.1007/978-3-540-77974-2.

[3] O'Rourke, J. *Computational Geometry in C. Cambridge*: Cambridge University Press, 1998. doi: 10.1017/CBO9780511804120.

[4] Meisters, G.H. Polygons Have Ears. *The American Mathematical Monthly*. 1975, 82(6), 648-651. doi: 10.1080/00029890.1975.11993898.

[5] ElGindy, H., Everett, H., Toussaint, G. Slicing an ear using prune-and-search. *Pattern Recognition Letters*. 1993, 14(9), 719-722. doi: 10.1016/0167-8655(93)90141-Y.

[6] Kong, X., Everett, H., Toussaint, G. The Graham scan triangulates simple polygons. *Pattern Recognition Letters*, 1990, 11(11). 713-716. doi: 10.1016/0167-8655(90)90089-K.

[7] Mei, G., Tipper, J.C., Xu, N. Ear-Clipping Based Algorithms of Generating High-Quality Polygon Triangulation. In *2012 Int. Conf. on Information Technology and Software Engineering*. LNEE, 2012, 212, 979-988. doi: 10.1007/978-3-642-34531-9_105.

[8] Eberly, D. Triangulation by Ear Clipping. *Geometric Tools*. Nov. 18, 2002. Accessed August 4, 2023 from https://www.geometrictools.com/Documentation/Documentation.html

[9] Preparata, F.P., Shamos, M.I. *Computational Geometry: An Introduction. New York*, NY: Springer, 1985. doi: 10.1007/978-1-4612-1098-6.

[10] Chew, L.P. Constrained Delaunay triangulations. *Algorithmica*, 1989, 4, 97-108. doi: 10.1007/BF01553881.

[11] Wijeweera, K.R., Kodituwakku. S.R. Accurate, Simple and Efficient Triangulation of a Polygon by Ear Removal. *Ceylon Journal of Science*, 2016, 45(3), 65-76. doi: 10.4038/cjs.v45i3.7402.

[12] McKenna, M., Toussaint, G.T. Finding the minimum vertex distance between two disjoint convex polygons in linear time. *Computer & Mathematics with Applications*. 1985, 11(12), 1227-1242. doi: 10.1016/0898-1221(85)90109-9.

[13] Toussaint, G.T. An optimal algorithm for computing the minimum vertex distance between two crossing convex polygons. *Computing*. 1984, 32, 357-364. doi: 10.1007/BF02243778.

[14] Aggarwal, A., Moran, S. Shor, P.W. Suri. S. Computing the minimum visible vertex distance between two polygons. *In Workshop on Algorithms and Data Structures* (WADS 1989). LNCS, 1989, 382, 115-134. doi: 10.1007/3-540-51542-9_11.

[15] Amato, N.M. An optimal algorithm for finding the separation of simple polygons. In *Workshop on Algorithms and Data Structures* (WADS 1993). LNCS, 1993, 709, 48-59. doi: 10.1007/3-540-57155-8_235.

[16] Wang, C., Chan, E.P.F. Finding the minimum visible vertex distance between two non-intersecting simple polygons. *Second Annual Symposium on Computational Geometry*. ACM, 1986, 34-42. doi: 10.1145/10515.10519.

[17] Seidel, R. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Computational Geometry*, 1991, 1(1), 51-64. doi: 10.1016/0925-7721(91)90012-4.

ACERCA DEL AUTOR

Dr. Yuriy Kotsarenko is a researcher, developer, and entrepreneur with a doctoral degree in Computer Sciences from 2011. He has authored more than a dozen scientific articles and holds multiple registered trademarks and copyrighted materials. Yuriy is a founder and operator of Afterwarp Interactive, a consulting firm specializing in providing products and services, particularly robust real-time visual applications for the industrial sector. His primary research interests lie in massively parallel processing on GPUs, GPGPU techniques and 3D visualization.