GPU-Based Parallel Algorithm for Computing Point Visibility Inside Simple Polygons

Ehsan Shoja[a,*], Mohammad Ghodsi[a,b,**]

[a]*Department of Computer Engineering, Sharif University of Technology, Tehran, Iran*
[b]*Institute for Research in Fundamental Sciences (IPM), Tehran, Iran*

Abstract

Given a simple polygon P in the plane, we present a parallel algorithm for computing the visibility polygon of an observer point q inside P. We use chain visibility concept and a bottom-up merge method for constructing the visibility polygon of point q. The algorithm is simple and mainly designed for GPU architectures, where it runs in $O(\log n)$ time using $O(n)$ processors. This is the first work on designing a GPU-based parallel algorithm for the visibility problem. To the best of our knowledge, the presented algorithm is also the first suboptimal parallel algorithm for the visibility problem, that can be implemented on existing parallel architectures. We evaluated a sample implementation of the algorithm on several large polygons. The Experiments indicate that our algorithm can compute the visibility of a polygon having over 4M points in tenth of a second on an NVIDIA GTX 780 Ti GPU.

Keywords: Visibility Polygon, Visibility Chain, Parallel Processing, GPU Processing, CUDA

1. Introduction

Finding visibility region of an observer is one of the old and well-known problems in computational geometry and computer science [1, 2]. The visibility problem has many subproblems according to the type of the polygon, observer, and observation. In this paper, we will mainly focus on the visibility of a point inside simple polygons. Although there are practical optimal solutions for this problem that have linear bounds on a single processor [3, 4, 5], unfortunately there are just a few parallel algorithms for this problem [6, 7, 8] which are either impractical or not good enough for new architectures.

From beginning of the 21st century, due to energy-consumption and heat-dissipation issues, the increase of clock frequency was no longer a good option. At 2003, parallel architectures made their ways into our personal computers, as multicore CPUs and manycore GPUs. Since then, the race of floating point performance has been led by manycore GPUs, and year by year the performance gap between these two architectures has been increased in the favor of manycore GPUs. This large performance gap has motivated developers to move the computationally intensive part of their program to GPUs for execution, and thus exploit the parallel computing capabilities of them. Until 2006, using graphic chips for parallel computing was very difficult and required special techniques called general purpose computing on graphics processing units (GPGPU). With the advent of CUDA in 2007 by NVIDIA [9] GPU programming becomes more popular.

Today, many programs in different fields, including computational chemistry, molecular dynamics, computational fluid dynamics, computational finance, etc. have been implemented on GPU. The field of computational geometry, is not an exception. Several parallel algorithms have been proposed for computing convex hull [10, 11, 12], Voronoi diagram [13, 14, 15], and Delaunay triangulation [16, 17] of a set of points on GPU. In this paper, we are going to address another important problem in this field: The visibility problem. This problem is also used in many other fields of computer science, including robotics, computer vision and graphics.

The present work is mainly inspired by the one in [6], where Atallah et al. give an elegant parallel algorithm for the point visibility problem on the EREW-PRAM computational model. Although the algorithm presented by Atallah et al. [6] is a theoretically optimal solution on PRAM models, it is not suitable for real parallel systems. The reason is that, the algorithm itself is very complicated and also makes use of some other involved algorithms such as parallel merge sort [18] and linear visibility [3, 4, 5]. Therefore, in order to be able to apply a similar algorithm on real parallel architectures especially on restricted ones like GPUs, we have to make some core modifications to the algorithm.

The rest of the paper is organized as follows. We first give the notations, preliminaries and also our core visibility definition in section 2. An overview of the approach, is sketched in section 3. Section 4 gives some insights to the chain visibility concept and the related possible cases that could occur during the algorithm. Section 5 proposes the algorithm itself, gives further solutions for the special cases left behind and analyses the complexity of the resulted algorithm. In section 6, the experimental results are presented and discussed, and finally in section 7, conclusions and future work are presented.

*Corresponding author
**Principal corresponding author
Email addresses: shoja@ce.sharif.edu (Ehsan Shoja), ghodsi@sharif.edu (Mohammad Ghodsi)

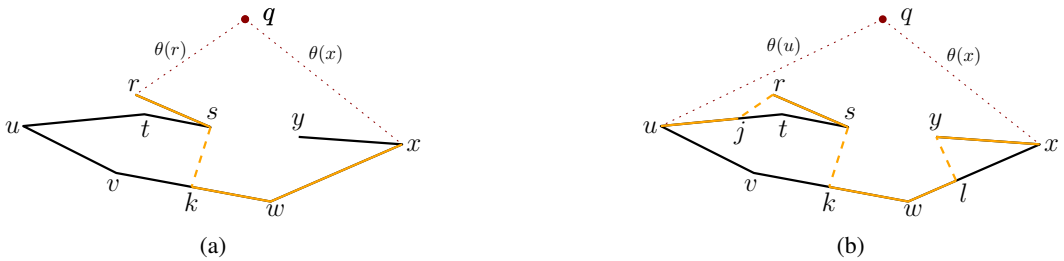


Figure 1: Visibility Chain $Vis(C)$ from point q, (a) The new definition used in this paper, and (b) The definition from [6] by Atallah et al.

2. Preliminaries

Let $P = (v_0, v_1, ..., v_n)$ denote a simple polygon in the plane, where $v_0, v_1, ..., v_n$ are a consecutive list of vertices in counterclockwise order (otherwise we can easily convert it to a counterclockwise order list), and $v_0 = v_n$. Each vertex has two fields representing the x and y coordinates of it in the plane. The edge connecting v_i to v_{i+1}, will be denoted by e_i or v_iv_{i+1}. For any two points u and w in the plane, we will show the line segment connecting these two points with $\overline{uw}$ or $\overline{wu}$. A polygonal chain C, is a connected series of edges or line segments. Each chain C, has a length denoted by $|C|$ that is the number of edges it contains. Our definitions and assumptions are mainly based on the one's defined at the work of Atallah et al. [6]. However, in contrast to the Algorithm of Atallah et al. that only works on open chains our algorithm only works on closed chains (polygons).

Definition 1. *Visibility Chain* We define the visibility chain of C from q, as the portion of the chain C that is visible from point q, with respect to the existing **counterclockwise edges** to block its visibility. In other words we assume that only counterclockwise edges of the polygon are opaque objects. We denote the visibility chain of C by $Vis(C)$.

This definition is different from the definition given in [6]. Figure 1 shows the difference between these two definitions. The purpose of this definition is to simplify the visibility chain concept required to be handled at each merge operation. Indeed, using this definition, has two major advantages: First, the resulted visibility portion at each stage would respect the existing order of the corresponding chain in polygon (compare figures 1a and 1b). Second, the merging chains would only block each other at their merging point (except for the case that the chains have two common intervals like the one represented in figure 12). As a result of the first advantage, we can handle merge operations more easily because visibility chains can be expressed by an ordered range from a starting visible point to an ending visible point, and if one chain blocks a portion of other chain, that portion would be ordered too (e.g. we can say all vertices and edges having greater or lower index than an identified boundary index is not visible anymore). Because of the second advantage, we do not need to consider the other ends of each chain when merging (even for the special case mentioned).

Assuming that, clockwise edges are not visible, would also have the same effect but, considering them as not opaque has other advantages too. It will reduce the number of possible cases that may occur during merge operations (These cases will be discussed in section 4). In definition 1, we consider clockwise edges from the beginning as invisible and transparent edges, this is because of the monotonicity of the visibility polygon.

The Visibility chain $Vis(C)$ is a monotone chain with respect to point q, that is, any rays emitting from q, would intersect $Vis(C)$ in at most one point or would be tangent to a line segment on it. An edge of $Vis(C)$ that is not contained in an edge of C is called an extra segment of $Vis(C)$ or window of point q. Every window of point q are collinear with it. The closest point of the window to point q, is called the *base* of the window and is a vertex of the polygon. The other vertex is called the *end* of the window and necessarily is not a vertex of the polygon. Every window will be denoted by $w(base, end)$. Each window separates the polygon into two regions, the visible and invisible regions with respect to that window. The invisible region of the window is called a *pocket* and will be shown by $pocket(base, end)$. A window is called a left (right) window if the corresponding pocket lies to the left (right) of it. We would call the base vertex of a window the *blocking vertex* of the vertices and edges inside its pocket.

3. An Overview of the Approach

We assign each edge of the polygon to a thread, so each thread at first has a chain of length one. Each thread can tell whether its edge is visible or not, according to the direction of its edge with respect to point q. After this, at each stage threads would double the length of their chain (but still working on the same edge as before) by merging and cooperating with other threads. This would continue until we compute the visibility of the whole polygon.

As mentioned, each thread has been given an edge with its two endpoints. So, thread with index i holds the edge e_i (v_iv_{i+1}). In order to represent all edges of the polygon we would need n threads, from index 0 to $n-1$. The chain index, for each thread can be evaluated by $(thread\ index)/|C|$. At each stage of the algorithm, we merge two consecutive chains (the chain with the even index and the odd chain next to it). While merging two consecutive chains, each thread related to these chains would update the visibility state of their edges and vertices.

The length of chains, double at each stage, so we need $\lceil \log n \rceil$ stages to compute $Vis(P)$. Another stage is also needed to merge the last chain with itself at vertex v_0. (This is not necessary if v_0 is both visible and convex.)

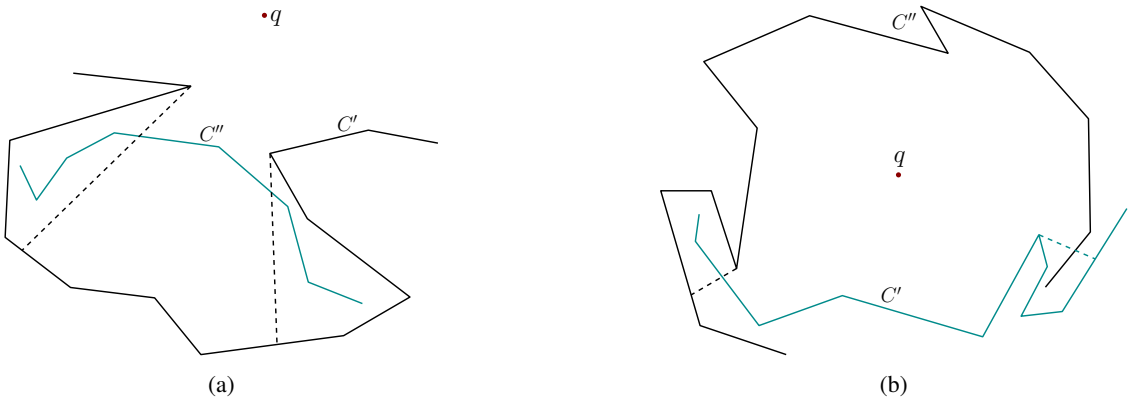


Figure 2: Visibility chains according to the new definition that also have two intersections

4. Visibility Chains and Their Intersections

In the previous section, we gave an overall description of merging chains and computing their visibilities. In this section and the next one, we will study the merging operation in more details.

According to the assumption, polygon P is simple and thus sub-chains of P do not cross each other. But this is not necessarily true for visibility chains. The fact that, visibility chains could cross each other, makes the merging operation even harder. Therefore, it is necessary to consider the number of possible intersections and the way they happen between the two merging chains.

Atallah et al.'s Lemma 4.1. *If C' and C'' are two subchains of C that are disjoint except that they may share one endpoint, then there are at most two intersections between $Vis(C')$ and $Vis(C'')$. Furthermore, if $Vis(C')$ and $Vis(C'')$ have two intersections and $I(C') \cap I(C'')$ consists of two disjoint intervals, then there is exactly one intersection in each such interval. If $Vis(C')$ and $Vis(C'')$ have two intersections and $I(C') \cap I(C'')$ consist of one interval, then one of $I(C')$ or $I(C'')$ contains the other.*

Proof. Although, definition 1 for visibility chain is different from the one given by Atallah et al. [6], but the windows occurring in the new definition are just a subset of all windows occurring in the original definition. This is due the fact that, only the windows made by clockwise edges, are not considered in the new definition. Hence, the proof of this lemma is given at [6]. Figure 2 shows that, the visibility chains of the same two cases at [6] also have two intersections in the new definition. □

Theorem 4.2. *Assume C_1 and C_2 are two consecutive subchains of chain C that, except a common endpoint they share (v_x), are disjoint. The visibility chains $Vis(C_1)$ and $Vis(C_2)$ can have at most one **visible** intersection per each common interval.*

Proof. According to lemma 4.1, if we have two subchains of a simple chain that at most share a common endpoint, then the visibility chain of these two subchains could not have more than two intersections. Now if the visibility chains cross each other only at one point, then the theorem holds.

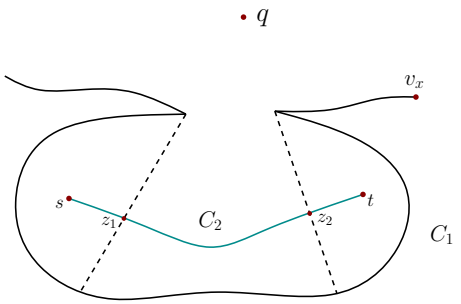


Figure 3: The case where $Vis(C_1)$ and $Vis(C_2)$ have two intersections in a common interval

For the case where subchains have two intersections, according to lemma 4.1 one of the situations depicted in figure 2a or figure 2b could occur. In figure 2b, $Vis(C')$ and $Vis(C'')$ have only one intersection per each common interval. Therefore, figure 2a, is the only case where $Vis(C')$ and $Vis(C'')$ have two intersections inside a common interval. So, in order to prove the correctness of the theorem it is sufficient to show that cases like figure 2a would never occur.

We assume that $Vis(C_1)$ and $Vis(C_2)$ have two intersections like the way shown in figure 3. ($Vis(C_2)$ crosses two extra segments of $Vis(C_1)$ at points z_1 and z_2.) We know that C_1 and C_2 are connected at vertex v_x. Hence, we should connect this vertex to one of the vertices s or t using a chain of line segments.

It is much easier to prove this theorem according to chain visibility definition of Atallah et al. Figure 4 shows the cases where the common vertex v_x is connected to vertex s or vertex t. In figure 4a, v_x is connected to s, so the subchain of C_2 from s to t is no longer visible. Hence, $Vis(C_1)$ and $Vis(C_2)$ cross each other just at point z'. In figure 4b, v_x is connected to t, but this makes an outer window $w(e, f)$ which blocks the subchain of C_2 from f to t. So again, $Vis(C_1)$ and $Vis(C_2)$ cross each other only at one point (this time at z_1).

In the same way, we can prove the theorem considering the new definition for visibility chains. Figure 5 shows the cases where the common vertex v_x is connected to either vertex s or vertex t.

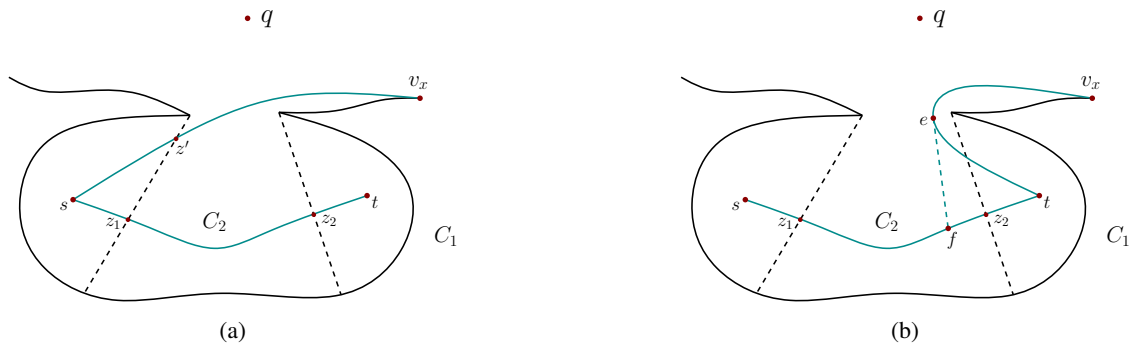


Figure 4: Considering the definition of Atallah et al. for visibility chains, two consecutive visibility chains could not have more than one intersection in a common interval

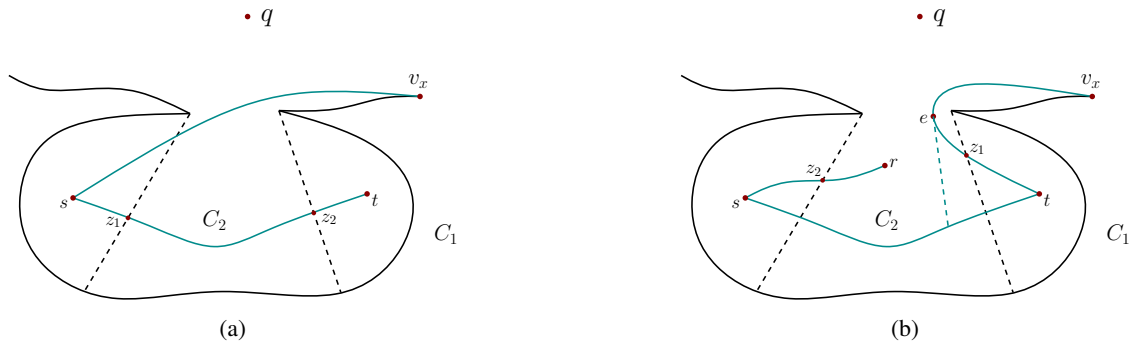


Figure 5: Considering definition 1 for visibility chains, two consecutive visibility chains could not have more than one visible intersection in a common interval

In figure 5a, v_x is connected to s. In this case, the subchain from v_x to s is transparent (because it is clockwise) and the subchain from s to t is visible. Therefore, visibility chains $Vis(C_1)$ and $Vis(C_2)$ cross each other at points z_1 and z_2, but none of these points are visible from q. So, we can easily ignore them. In figure 5b, v_x is connected to t. In this case, the subchains from e to t and from s to r are visible. Therefore, visibility chains $Vis(C_1)$ and $Vis(C_2)$ cross each other at points z_1 and z_2 as shown in the figure. But, from these two intersection points, only z_2 is visible to q. Thus, the theorem holds. □

Let $I(C_1)$ and $I(C_2)$ be the angular interval of chains C_1 and C_2. Clearly, $I(C_1) \cap I(C_2)$ consists of no more than two disjoint intervals. Moreover, due to chain's connectivity, the endpoints of intervals are the endpoints of chains. So, C_1 and C_2 and therefore $Vis(C_1)$ and $Vis(C_2)$ would have at most two common intervals, and these two intervals occur at the endpoints of chains.

Two chains could block or cross each other at their common interval(s). Thus, in order to find out whether two chains block each other or not, it is sufficient to check the visibility state of their endpoints. This is much easier for two consecutive chains with a common endpoint. From now on, we assume that the chains C_1 and C_2 could only block each other at their merging interval. This seems to be sufficient because the other endpoints will finally meet each other. But unfortunately this is not true for the case where two chains intersect at different intervals (*e.g.* figure 2b). We will consider this case separately in section 5.1.

As stated earlier, two consecutive chains C_1 and C_2 would block each other only if they have a common angular interval other than $\theta(v_x)$. If two chains, have an empty common interval (just the angle $\theta(v_x)$), then necessarily $\theta(v_x)$ is one of the two endpoints of $I(C_1)$ and $I(C_2)$. Figure 6 illustrates two possible cases, where $I(C_1)$ and $I(C_2)$ have empty common interval. For the case displayed in figure 6a, vertex v_x is visible before merging and would remain visible afterwards, and for the case

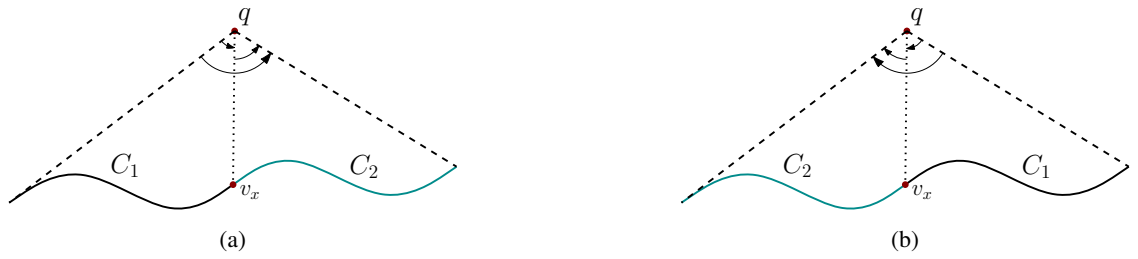


Figure 6: Two possible cases, for consecutive chains C_1 and C_2 with an empty common interval

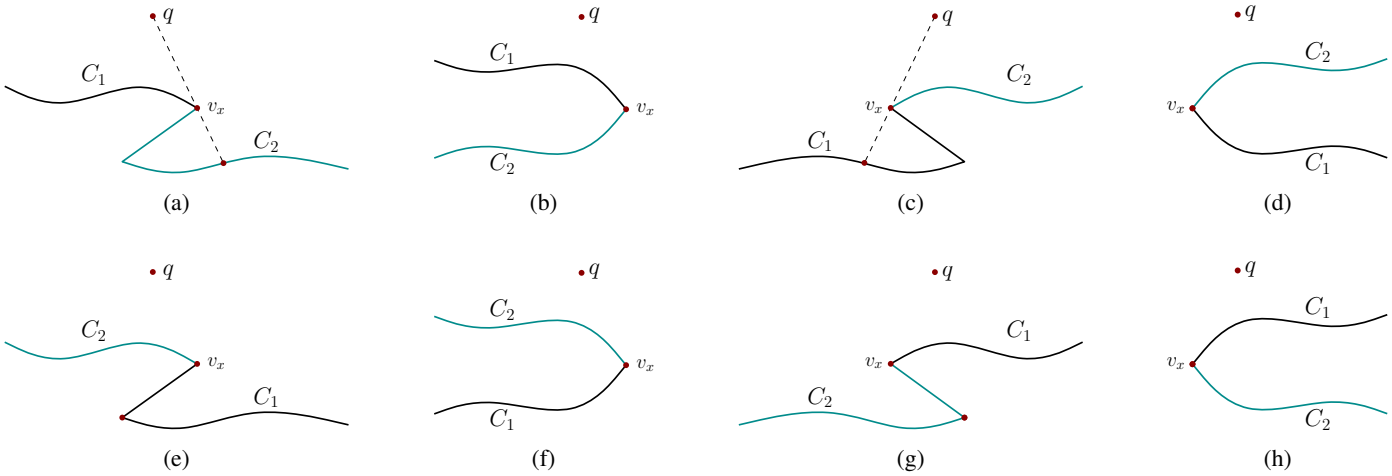


Figure 7: Illustrating case 1 and its subcases

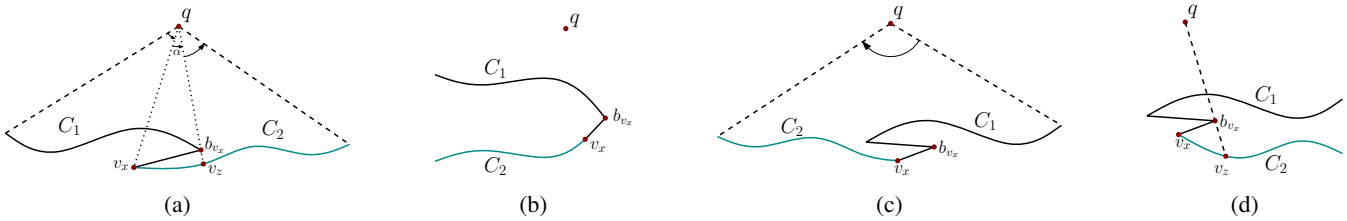


Figure 8: Illustrating case 2 and its subcases

displayed in figure 6b, vertex v_x is not visible before merging (because it is on a clockwise edge) and would remain the same afterwards. Therefore, this case does not have any effects on the visibility state of the merging chains. As a result, we can assume that the chains are merged, and just do double the length of chains.

If the common interval of C_1 and C_2 was not empty, then one of the cases below has definitely occurred.

Case 1. Vertex v_x is not covered in any of the chains C_1 and C_2, but both chains partially or completely locate on the same side of line qv_x and thus share a common interval. Figure 7 shows different subcases of this case.

Case 1.1. Chain C_1 is counterclockwise and over chain C_2 near merging vertex v_x. Figures 7a and 7b represent this subcase. In figure 7a, chain C_1 blocks chain C_2 partially, and in figure 7b, chain C_1 blocks chain C_2 completely.

Case 1.2. This case is symmetric to case 1.1 with the roles of C_1 and C_2 being interchanged. Figures 7c and 7d represent this subcase.

Case 1.3. Chain C_2 is clockwise and over chain C_1 near merging vertex v_x. In this case, C_2 does not block C_1. Figures 7e and 7f represent this subcase.

Case 1.4. This case is symmetric to case 1.3 with the roles of C_1 and C_2 being interchanged. Figures 7g and 7h represent this subcase.

Case 2. Vertex v_x is blocked by a subchain in C_1 or C_2, and the chains $Vis(C_1)$ and $Vis(C_2)$ do not intersect. Figure 8 illustrates the cases where v_x is blocked by chain C_1. The cases where v_x is blocked by C_2 is symmetric.

Case 2.1. Blocker of v_x (vertex b_{v_x}), is not covered by a clockwise subchain (figures 8a and 8b). As described earlier, b_{v_x} the blocker of vertex v_x, is the base vertex of the outermost window containing v_x. In figure 8a, b_{v_x} blocks the pocket according to window $w(b_{v_x}, v_z)$, and in figure 8b, b_{v_x} blocks the chain C_2 completely.

Case 2.2. Blocker of v_x (vertex b_{v_x}), is covered by a clockwise subchain (figures 8c and 8d). In figure 8c, b_{v_x} blocks the whole of C_2, and in figure 8d, b_{v_x} just blocks the pocket according to window $w(b_{v_x}, v_z)$. Note that the figure 8c is the same as figure 8b, and the figure 8d is the same as figure 8a. From now on, we will only consider the cases that are unique.

Case 3. In both chains C_1 and C_2, v_x is blocked, besides $Vis(C_1)$ and $Vis(C_2)$ do not intersect (figure 9). Figure 9 illustrates the cases where C_1 blocks C_2. The cases where C_2 blocks C_1 is symmetric. The blocker of v_x in chain C_1 is represented by b'_{v_x} and in chain C_2 is represented by b''_{v_x}.

Case 3.1. One of the blockers b'_{v_x} or b''_{v_x}, is blocked by the other chain (b'_{v_x} by C_2 and b''_{v_x} by C_1). In this case, one of the blockers is a true and the other is a false blocker. Figures 9a and 9b represent the cases where b''_{v_x} is a false blocker. The cases where b'_{vx} is a false blocker is symmetric. Knowing the coordinates of the true blocker we could easily compute the coordinates of vertex v_z and the respective window.

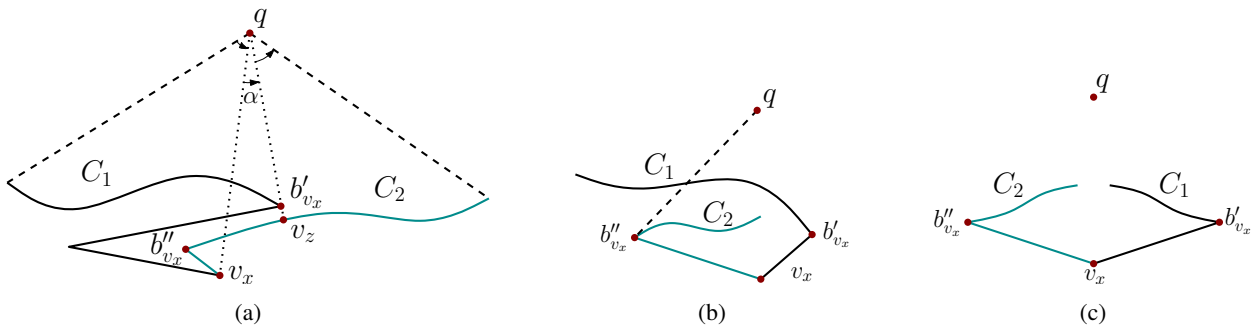


Figure 9: Illustrating case 3 and its subcases

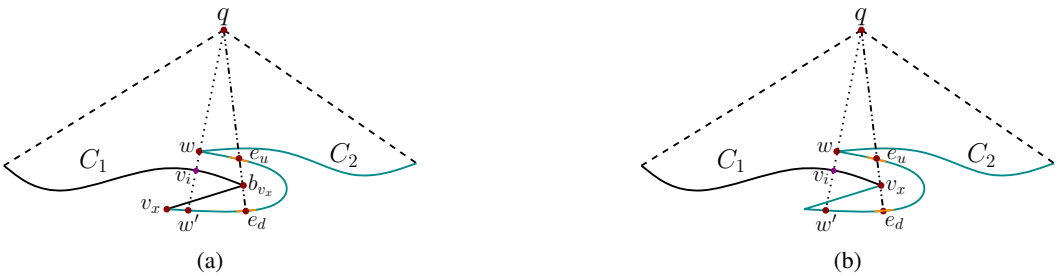


Figure 10: Illustrating case 4

Case 3.2. None of the blockers b'_{v_x} and b''_{v_x}, is blocked by the chain. Figure 9c represents the only possible case. Notice that this case is just like the case of figure 9b, with the difference that b''_{v_x} is not blocked anymore. Unfortunately, in this case we could not recognize the true blocker. In section 5.1 we will give more detail about this situation and how to handle it.

Case 4. Vertex v_x is blocked in one of the chains C_1 or C_2, and the visibility chains $Vis(C_1)$ and $Vis(C_2)$ have an intersection. Intersection occurs when b_{v_x} (or v_x when it is not blocked) of one chain crosses a window of another chain. This could not occur in cases where v_x is not blocked and also it is not a blocking vertex (1.3, 1.4). Note that a vertex could not cross a window without blocking a subchain of it. It also could not occur in the cases where b_{v_x} or v_x, blocks the other chain completely (the cases represented by figures 7b, 7d, 8b, 8c). Besides, when a chain is completely blocked, no visible chain would be left to cross. Thus, the only cases which could intersect are the ones represented in figure 10.

Case 5. Vertex v_x is blocked in both of the chains C_1 and C_2, in addition the visibility chains $Vis(C_1)$ and $Vis(C_2)$ have an intersection. As stated in case 4, an intersection could not occur in a case where b'_{v_x} or b''_{v_x}, blocks the other chain completely. Thus, the cases illustrated as figures 9b and 9c could not cross each other. Figure 11 represents the case where b'_{v_x} crosses a window of chain $Vis(C_2)$. The case where b''_{v_x} crosses a window of $Vis(C_1)$ is similar.

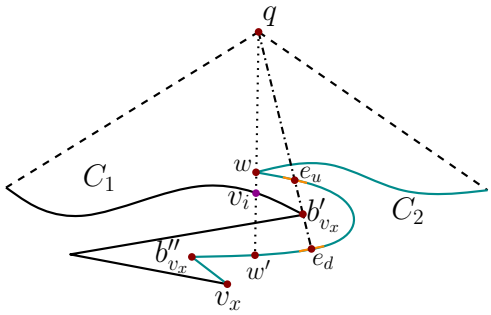


Figure 11: Illustrating case 5

5. The Proposed Algorithm

In the previous section we considered all the possible cases where two merging chains could block each other. Now, in this section we will discuss the main algorithm and its stages. Mainly, our algorithm consists of $\lceil \log n \rceil$ merging stages. Each stage itself consists of four steps. These steps are required for threads to collaboratively compute the merged visibility chain. The communication between threads are handled using the consecutive writes to and reads of the same global memory address. The steps, are designed in a way to make sure the required communication between threads is correctly handled.

A Brief description of each step is as follows: In step one, we will check the existence of an intersection or a false blocker. In step two, we will find the base vertex of the window that is intersected (if there is an intersection at all). In step three, we will find the end vertex of the window (if intersected or blocked) and we will denote it as boundary vertex. Finally, in step four, according to the index of boundary vertex, we will decide about the visibility state of each vertex and edge in the chain. In other words, the first two steps of the algorithm are devoted to identifying the real blocker of the chain. And the second two is devoted to computing the window (if existed) and the blocked

subchain.

Now, we will describe each step in more detail. Note that at each stage, we denote the tail thread of even chain (the chain with even index), and the head thread of odd chain as the candidate thread of the corresponding chains.

Step 1. Each thread of the current chain, identifies the blocker of v_x in the other chain (if there exist such blocker). This could be done by checking the blocker of the head or the tail vertex of that chain, depending whether its *chain index* is odd or even. If v_x is not blocked, we consider v_x itself as the blocker. Having the coordinates of the blocker (b_{v_x}), we check whether it is a false blocker or not. According to definition, b_{v_x} is a false blocker if it is covered by a counterclockwise edge. Thus, each thread, checks whether its edge covers b_{v_x} or not. If such a thing happens, the corresponding thread would set a flag of candidate thread which indicates the occurrence of false blocker situation. Likewise, in order to check whether b_{v_x} crosses a window, we check the existence of two edges in a pocket that falls below and over b_{v_x}. In order to do this in parallel, each thread checks whether its blocked edge intersect the line qb_{v_x}. If such a case happens, every counterclockwise edge below b_{v_x} set one of the *intersection detection* flags of its blocker, and every clockwise edge that covers b_{v_x}, also set the other *intersection detection* flag of its blocker. These two edges are represented as e_u and e_d in figures 10 and 11. Except a special case that will be discussed in section 5.1, the blocker of these two edges (w in figures 10 and 11), is the base vertex of the window that has been crossed.

Step 2. Assume there is an intersection between two visibility chains. Let w denote the base vertex of the window that has been crossed. We know that both *intersection detection* flags of w should have been set at step 1. Thus, at this step, each vertex checks its both *intersection detection* flags. If both flags of a vertex have been set, then that vertex is either the base vertex of the intersected window (like the case in figures 10 and figure 11) or it is the blocker of the current chain (like the case in figure 9a with two blockers). If w is not the same vertex that blocks v_x in current chain, then definitely we have an intersection and w is the base of the intersected window. Now we have to set w as the new blocker, and we have to inform other threads too. We do this by setting the *intersection* flag of the candidate thread. On the other hand, if w is the current chain's blocker, then it has the other chain's blocker in its pocket. Hence, in this case, we should invalidate the blocker (b_{v_x}) of the other chain.

Step 3. At this step, threads in each chain could be in one of these three cases:

Case 1. If the *intersection* flag of the current chain (whether it is odd or even) has been set, then for odd (even) chains, the vertices and edges before (after) w should be set as blocked. (Note that w is now accessible as b_{v_x}.) For this, if the current chain is even, then each thread in it compares its index to that of vertex w, and if its index was greater, then it resets the *visibility* flag of its vertex and edge to zero. Otherwise, if the current chain is odd, each thread with smaller index compared to w's, would reset its visibility flags to zero.

Case 2. If the *intersection* flag of other chain has been set, then we have to locate the point v_z, which is the intersection point and also the boundary vertex of visible and blocked regions. To do this, each visible edge e_i checks to see whether it has an intersection with the line qw or not. Notice that there is only one such edge (actually if we have $|bv_z| < |qv_z|$). After computing the intersection point v_z and updating the visibility state of its edge and vertices, the intersected thread stores its index at the candidate thread to inform others.

Case 3. If the *intersection* flag has not been set for both chains (if the chains intersect), then we have to check if the chains block each other. Like the way we described in step 1, we find the blocker of other chain (b_{v_x} for cases 2 and 3 and v_x for blocking subcases of case 1). Now, in the same way as the previous case, we find v_z and store its index at candidate thread's address.

Step 4. In this step, we finally decide the visibility state of the chains that were not determined yet (cases 2 and 3 of step 3). Threads with larger index than $index(v_z)$ (index of boundary vertex) in even chains, and threads with smaller index than $index(v_z)$ in odd chains, update the visibility state of their edges and vertices to not visible. They also set b_{v_x} (v_x in some cases) as their blocker.

5.1. Dealing with Special Cases

In this subsection, we consider some of the cases and situations ignored in the previous sections for simplicity. Below we describe each case and provide algorithmic solutions for each of them.

- **The case where we have two common intervals, each with an intersection (figure 12):** During considering possible cases and also during the description of the algorithm, we ignored this case. And we assumed that it does not happen. Here in this section we will consider this case thoroughly, and we will give possible solutions to handle this case. The first solution is to consider two intersections during the algorithm one occurring when b_{v_x} is inside a pocket and the other when b_{v_e} is inside a pocket (v_e is the first vertex of even chain or the last vertex of the odd chain). This solution is not good enough and will double the work of the algorithm and it also may cause some unconsidered situations. The other simple solution is to make sure that either this case would not happen or it would not affect the final visibility polygon.

  Notice that this condition could only happen when b_{v_x} and b_{v_e} are inside a pocket. Thus, neither the blockers (b_{v_x} and b_{v_e}) and nor their corresponding vertices (v_x and v_e) are visible. To handle this case, we choose to make sure that this case does not happen unless it does not affect $Vis(P)$. We do this by choosing a visible point over polygon as the point v_0. (This requires rotating the polygon to make this point as the new v_0.) Let L_q be a half-line from q to its right that crosses the polygon at points $(c_0, c_1, \ldots, c_k)$. We compute the new v_0 by doing a minimum reduction on x-coordinates of c_is. After doing this,

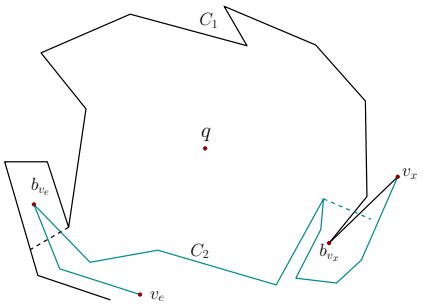


Figure 12: The case with two common intervals

the special case mentioned, could never happen for the last two chains (because v_0 the starting point of the first chain will be visible). In other words, after finding a visible v_0, ignoring this case would never cause a problem.

- **The case where we have two true blockers (figure 9c):** Unfortunately in this case we could not precisely find out which one of the blockers is true and which one is false. In order to handle this we assume that we have two kinds of blockers: temporary and permanent blockers. Permanent blockers are the base vertex of the windows whose their end vertex has been found (when b_{v_x} blocks the other chain partially). Temporary blockers are those which their window has not been find or in other words those who blocks the chain completely. We have two kinds of temporary blockers the left and right temporary blockers. A temporary blocker is considered left blocker if it blocks the odd chain completely. In the same way, a temporary blocker is considered right blocker if it blocks the even chain completely. Thus, a vertex could have both left and right blockers (like vertex v_x in figure 9c). This goes on until finally we could recognize which one is the true blocker and which one is the false blocker. Figure 13 shows the cases where b'_{v_x} becomes the true blocker. (The cases where b''_{v_x} becomes the true blocker is similar.)

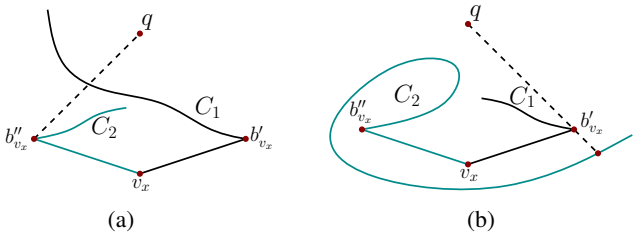


Figure 13: Illustrating the situation where we can finally detect the true blocker (here b'_{v_x})

Unfortunately, we are not done yet. Consider a case where we have two true blockers like before (now we can call them *temporary blockers*), but this time one or both of the ending vertices (v_e) are also temporarily blocked in their own chains (see figure 14 where v_e is temporarily blocked by b'_{v_e}).

Now, merging the chains would result in two temporary

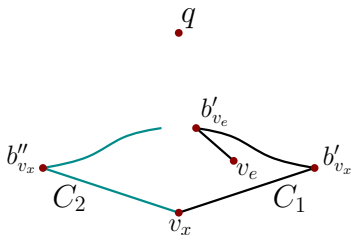


Figure 14: The case where we have two temporary blockers for vertex v_e

blockers for vertex v_e. Normally we replace the old temporary blocker with the new one, but in this case we do not for sure know whether the new temporary blocker (b''_{v_x} for figure 14) is a true blocker or not. For example in figure 14, if we set b''_{v_x} as the temporary blocker of vertex v_e, we may not be able to find the window resulted by vertex b'_{v_e}. On the other hand, if we set b'_{v_e} as the temporary blocker of v_e, we would find the window, but we will lose the information about vertex b''_{v_x} which might be important. In order to solve this problem, we will use two temporary blockers for each chain (it is easy to show that this is sufficient). So, in case of figure 14, we will have both b''_{v_x} and b'_{v_e} as temporary blockers. (b''_{v_x} blocks the subchain starting right after b''_{v_x} and ending at the vertex b'_{v_e}, similarly b'_{v_e} blocks the subchain starting right after b'_{v_e} and ending at the vertex v_e). Now we are able to find the window and also remember the old temporary blocker. Although adding another temporary blocker would make the implementation a little bit more intricate, but it would solve this special case completely.

- **The case where we could not detect the intersection point correctly:** In section 4 when considering possible cases, we assumed that $w = b_{e_d} = b_{e_u}$ in cases 4 and 5 (see figures 10 and 11). Unfortunately, this does not hold for the case when both chains have a sawtooth-like shape given at figure 15a. According to the algorithm, we would consider this situation as case 5. Thus, b''_{v_x} will be chosen as the true blocker, but there would be no visible edge to be intersected by qb''_{v_x} and also covered by b''_{v_x}. Figure 15b shows the subchain that will be computed as the visibility chain. Because of these wrongly computed chains we may later have two intersections for a window. However, we could have at most two intersections and both in the same direction, so we can easily handle this with choosing the one with minimum distance. (We have used a modification of atomicMin to do this, but it can be done without it too.) Thus, we can say these wrongly computed visibility chains do not affect other chains and their visibility state. Therefore, we ignore this case during merging stages, and we left them to be handled at a refinement phase at the end of the algorithm. To be more specific, after doing a compaction of visible vertices, we have to do a refinement on them and remove those that are not visible.

  After doing the first compaction, we check whether this

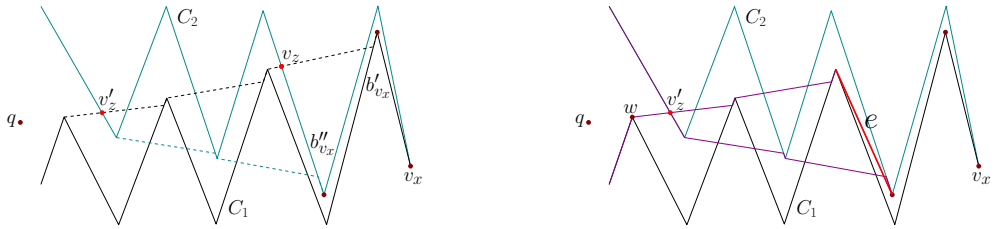


Figure 15: The case where we could not detect the intersection point correctly

case has happened or not. Notice that, edge e in figure 15b is a clockwise edge. Thus, in order to find out if this case has occurred or not it is sufficient for each edge to check the direction of its edge according to point q. If we have such an edge, then we have to refine the polygon. To do so we do another compaction over window edges plus clockwise edges. This is so that we could reduce the number of active threads. Now again we assign each edge to a thread.

As shown in figure 15b, the window relating to vertex w is the only window that crosses a counterclockwise edge. Thus, the thread of vertex w could find the real end vertex v_z, by searching through a monotone chain from e to w. Actually, this requires that each window knows its predecessor and successor clockwise edges (which can be done easily by broadcasting each e to all the windows on its both sides until reaching another clockwise edge). After finding v_z, we could either compact the polygon again or just use a jumping parameter to jump from w to v_z when outputting the polygon.

5.2. Algorithm Complexity

The presented algorithm mainly consists of five phases. This indicates that analyzing the complexity of each phase is a prerequisite for computing the complexity of whole algorithm.

1. *Finding v_0:* As stated earlier, we can find a visible v_0 by doing a minimum reduction on the x-coordinates of c_is (intersection points between the boundary of polygon and half-line L_q). Parallel reduction can be done in $O(\log n)$ time using $O(n/\log n)$ processors on GPUs [19].
2. *Rotation:* Having the index of v_0, each thread could compute its new index in $O(1)$ time. Thus, the work of this phase is also $O(n)$.
3. *Visibility Computation:* As described, this phase of the algorithm consists of $O(\log n)$ iterations, where each iteration itself consists of four constant time steps. This phase is executed by $O(n)$ threads and thus the work done during its execution is of order of $O(n \log n)$.
4. *Compaction:* Same as phase 1, we could compact the visible vertices of the polygon in $O(\log n)$ time with $O(n/logn)$ processors using available algorithms for compaction on GPU [20, 21, 22].
5. *Refinement:* The refinement phase consists of another compaction, which can be united with previous phase's compaction. Let n_w and n_c respectively indicate the number of window edges and clockwise edges of the latest visibility chain computed. Using $O(n_w)$ processors (note that $n_c = O(n_w)$), the rest of the phase, including the broadcasting and searching steps, would take $O(\log n)$ time.

Let us assume p is the number of cores in the device. The complexity of the algorithm follows from phase 3 (visibility computation phase) which with $O(n \log n)$ computational work dominates all other phases. Thus, the overall complexity of the whole algorithm is $O((n/p) \log n)$.

6. Experimental Results

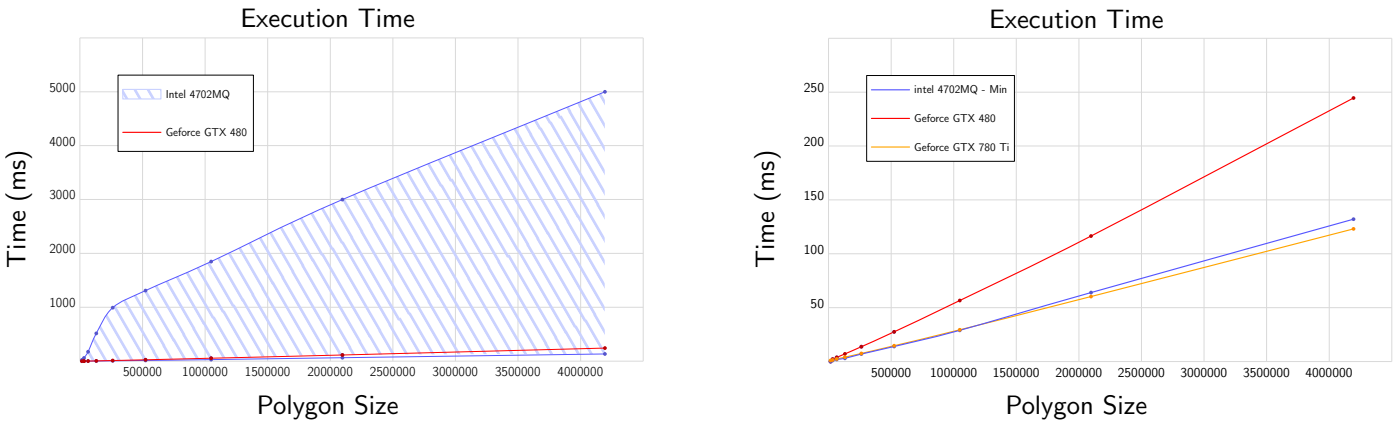
For the sake of evaluation, we implemented the algorithm using the CUDA programming model on two different NVIDIA devices with technical specifications given in tables 1 and 2. In order to check and prove the correctness of the algorithm, we have executed the algorithm on 5000 instances distributed over a hundred random polygon with the size of 16384 (50 query points for each polygon), generated by the RPG package[1] [23]. On the other hand, to evaluate the performance of the algorithm, we also generated and used some random polygons with different sizes of power two. More precisely for each size class, we have taken the average of at least 10 executions on 10 different instances of polygons and points. Better yet, for the sake of comparison, we implemented the serial algorithm provided by Joe and Simpson [5]. (We did this because we could not find a performance-wise implementation of point visibility algorithm for simple polygons.)

Table 1: Technical Specifications of Geforce GTX 480

Number of CUDA Cores	448
Frequency of CUDA Cores	1401 MHz
Single Precision Floating Point Performance	1345 GFLOPS
Total Dedicated Memory	1536 MB GDDR5
Memory Speed	3696 MT/s
Memory Bandwidth	177.4 GB/sec

Figure 16a compares the running time of our parallel algorithm on NVIDIA GTX 480 with the implemented serial algorithm on the CPU on hand. Note that the running time of

[1] http://www.cosy.sbg.ac.at/~held/projects/rpg/rpg.html



(a) (b)

Figure 16: Performance results

Table 2: Technical Specifications of Geforce GTX 780 Ti

Number of CUDA Cores	2880
Frequency of CUDA Cores	876 MHz
Single Precision Floating Point Performance	5046 GFLOPS
Total Dedicated Memory	3072 MB GDDR5
Memory Speed	7000 MT/s
Memory Bandwidth	336 GB/sec

the serial algorithm is closely relevant to the size of output, so instead of a simple line we have a range corresponding to execution time. Finally, to show the scalability and advantage of the proposed algorithm, we evaluated the performance of the algorithm this time on a better and more recent device (NVIDIA GTX 780 Ti). Figure 16b compares the running time of our parallel algorithm with the minimum execution time we have recorded for CPU (on polygons where almost whole polygon is blocked by some few edges). Figure 17 also represents the visibility polygons of some points computed inside a sample polygon with 2048 vertices.

7. Conclusion and Future Works

We have presented a GPU-based parallel algorithm for computing the visibility of a point inside simple polygons, which is the first algorithm for computing the visibility problem on manycore architectures. Similar to the algorithm of Atallah et al. [6] we used the chain visibility concept and their merging to compute the visibility polygon. However, these concepts have been used in a different way and with different definitions. The proposed algorithm has $\frac{p}{\log n}$ speedup over corresponding sequential algorithms. Although the algorithm is not theoretically optimal, yet unlike the previous works, it is quite simple and fitting for implementation purposes especially on manycore

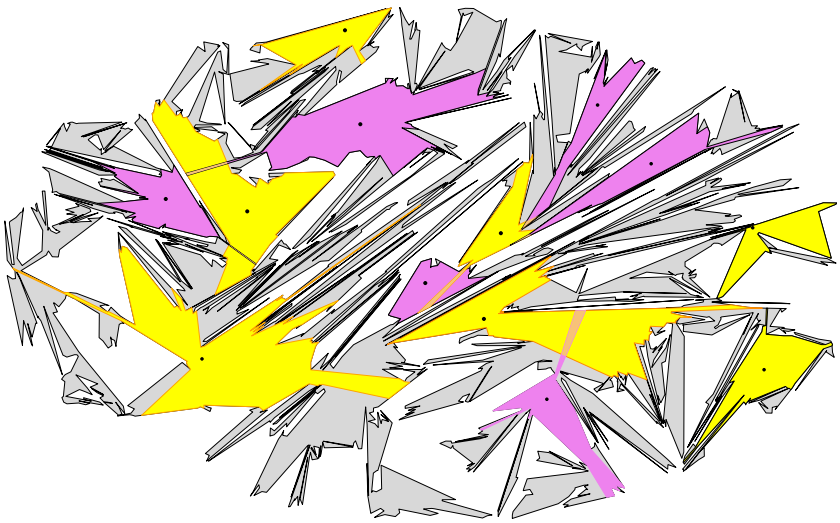


Figure 17: Visibility polygons of some sample points inside a polygon with 2048 vertices

architectures with lightweight threads.

For more coarse-grained architectures (like multicore systems) we could assign each thread a chain of size $O(n/p)$, instead of an edge. As a result, we could compute the visibility polygon in $O(n/p + \log p \log n)$ time using p processors.

We believe the algorithm presented in this section could be easily extended to other related problems. We are currently investigating the weak visibility and more generally x-ratio visibility variant of the problem, which we believe could be computed in the same bounds. The other extensions could be considering the problem on higher dimensions (e.g. assigning each thread a face instead of an edge) or investigating other related problems and applications.

References

[1] Asano T, Ghosh SK, Shermer TC. Visibility in the plane. Handbook of computational geometry 2000;:829–76.

[2] Ghosh SK. Visibility algorithms in the plane. New York, NY, USA: Cambridge University Press; 2007.

[3] ElGindy H, Avis D. A linear algorithm for computing the visibility polygon from a point. Journal of Algorithms 1981;2(2):186–97.

[4] Lee DT. Visibility of a simple polygon. Computer Vision, Graphics, and Image Processing 1983;22(2):207–21.

[5] Joe B, Simpson RB. Corrections to Lee's visibility polygon algorithm. BIT Numerical Mathematics 1987;27(4):458–73.

[6] Atallah MJ, Wagener H, Chen DZ. An optimal parallel algorithm for the visibility of a simple polygon from a point. J ACM 1991;38(3):515–32.

[7] Atallah MJ, Chen DZ. Optimal parallel hypercube algorithms for polygon problems. IEEE Transactions on Computers 1995;44(7):914–22.

[8] Guha S. Optimal mesh algorithms for proximity and visibility problems in simple polygons*. Parallel Algorithms and Applications 1998;13(2):167–85.

[9] NVIDIA C. NVIDIA CUDA programming guide (version 1.0). NVIDIA: Santa Clara, CA 2007;.

[10] Tang M, Zhao J, Tong R, Manocha D. Gpu accelerated convex hull computation. Computers & Graphics 2012;36(5):498–506.

[11] Srungarapu S, Reddy DP, Kothapalli K, Narayanan PJ. Fast two dimensional convex hull on the GPU. 2011.

[12] Stein A, Geva E, El-Sana J. Cudahull: Fast parallel 3D convex hull on the GPU. Computers & Graphics 2012;36(4):265–71.

[13] Rong G, Tan T. Jump flooding in GPU with applications to Voronoi diagram and distance transform. In: Proceedings of the 2006 symposium on Interactive 3D graphics and games. I3D '06; New York, NY, USA: ACM; 2006, p. 109–16.

[14] Fischer I, Gotsman C. Fast approximation of high-order Voronoi diagrams and distance transforms on the GPU. Journal of Graphics, GPU, and Game Tools 2006;11(4):39–60.

[15] Yuan Z, Rong G, Guo X, Wang W. Generalized voronoi diagram computation on gpu. In: Voronoi Diagrams in Science and Engineering (ISVD), 2011 Eighth International Symposium on. IEEE; 2011, p. 75–82.

[16] Rong G, Tan T, Cao T, Stephanus . Computing two-dimensional Delaunay triangulation using graphics hardware. In: Proceedings of the 2008 symposium on Interactive 3D graphics and games. I3D '08; New York, NY, USA: ACM; 2008, p. 89–97.

[17] Qi M, Cao T, Tan T. Computing 2D constrained Delaunay triangulation using the GPU. In: Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games. I3D '12; New York, NY, USA: ACM; 2012, p. 39–46.

[18] Cole R. Parallel merge sort. SIAM Journal on Computing 1988;17(4):770–85.

[19] Harris M. Optimizing parallel reduction in cuda. http://docsnvidiacom/cuda/samples/6_Advanced/reduction/doc/reductionpdf 2007;.

[20] Harris M, Sengupta S, Owens JD. Parallel prefix sum (scan) with cuda. GPU Gems 2007;3(39):851–76.

[21] Sengupta S, Harris M, Garland M. Efficient parallel scan algorithms for gpus. NVIDIA, Santa Clara, CA, Tech Rep NVR-2008-003 2008;(1):1–17.

[22] Merrill D, Grimshaw A. Parallel scan for stream architectures. University of Virginia, Department of Computer Science, Charlottesville, VA, USA, Technical Report CS2009-14 2009;.

[23] Auer T, Held M. Heuristics for the generation of random polygons. In: Proceedings of the 8th Canadian Conference on Computational Geometry. Carleton University Press. ISBN 0-88629-307-3; 1996, p. 38–43.