Detecting and Removing Islands in Graphics-Rendering-Based Computations of Lower Envelopes of Plane Slabs

Kamran Safdar
Fachbereich Computerwissenschaften
Universität Salzburg, A-5020 Salzburg, Austria
ksafdar@cosy.sbg.ac.at

ABSTRACT

Geometric algorithms which make use of graphics rendering often require manipulation and adaption of the pixel map of the lower envelope of plane slabs. The complex interaction of the slab geometries may give rise to isolated portions in the pixel map ("islands") which need to be discarded and patched appropriately. Such problems may occur, for instance, when attempting to compute multiplicatively-weighted Voronoi diagrams or straight skeletons in 2D by means of graphics rendering of the lower envelope of plane slabs in 3D. This paper presents general algorithms for detection, labeling, and removal of islands in an input pixel map. Removal, here, means recovery of the correct portions of the pixel map in lieu of the islands. The presented island detection algorithm requires only a constant number of passes over the input pixel map without any dependence on the number of input sites being processed by the geometric algorithm. This paper introduces the concept of black lists for the removal of islands and explains how the presented approach can cope with stacked islands, with no need for looping over the stack of islands for recovery of the correct pixel map underneath it. The discussion is concluded by experimental results obtained with the implementation of the presented algorithms.

Keywords: black list, FILM, GPGPU, islands, lower envelope, pixel map, plane slabs, rendering-based computation, SIR, STIR, straight skeleton, weighted Voronoi diagram.

1 INTRODUCTION

1.1 Motivation

Rendering specially crafted distance functions or slab geometries and projecting their lower envelope is a well-known approach for obtaining graphics-rendering-based solutions of geometric problems. For instance, the use of graphics hardware for computing discretized versions of Voronoi diagrams of point sites in 2D was shown already several years ago in early versions of the OpenGL Programming Guide [WND97]. Roughly, the Voronoi diagram of a set of point sites in 2D is a partition of the plane into individual regions, the so-called Voronoi regions, with exactly one region per site, such that every region is given by the loci of points closer to its defining site than to any of the other point sites. At every input point p an upright circular cone is constructed above the xy-plane such that its rotation axis is parallel to the z-axis and such that its apex coincides with the input point. If the envelope of the cone forms an angle of 45^o with the xy-plane then the set of all points with distance r from p can be identified with the intersection of the cone with the plane $z = r$. If every cone is colored uniquely then a discretized Voronoi diagram of the input points can be obtained by rendering the lower envelope of all cones via a parallel projection in the direction of $+z$, which is readily accomplished on a modern graphics processing unit (GPU).

Hoff et al. [HCK+99] extended this concept to discretized Voronoi diagrams of more general primitives in 2D and 3D. We give reference to [Hae90, Den03b, Den03a, Yam05b, Yam05a, CT05, VR05, FG06, LZC09] for other illustrative examples of how the fast rendering and interpolation capabilities of a GPU have been employed to solve various problems of a geometric nature.

Generally speaking, in all such rendering-based solutions every input site is assigned a unique color identity which is used as the color of the slab drawn for this site. Of-course, the geometries are designed in a way that they leave no portion of the scene uncovered thus constituting a correct diagram of the geometric structure being discretized. Moreover, all such solutions rely on the parallel projection of the lower envelope of these slab geometries. The mutual interactions of the geometries often is very complex as the slabs may penetrate

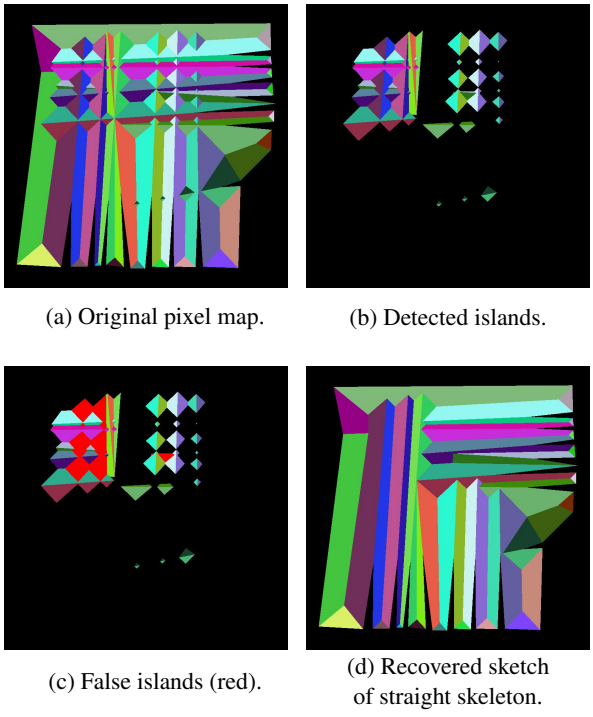
each other at multiple locations. From a computational point of view a problem occurs when a slab geometry reappears, in the projected image of the scene, after penetrating somewhere. These reappearing portions of the slab geometries project to areas in the pixel map which we call *islands*. In simple terms, an island is a set of pixels which bear the same color identity as some site but lie isolated from that site in the image. Formally, we define an island pixel as a pixel from which the projection of the corresponding site cannot be reached by traversing only identically colored neighbors.

While the standard Voronoi diagram of a set of point sites in 2D is indeed given precisely by the lower envelope of a set of cones in 3D and, thus, can be obtained easily by graphics rendering, this approach is not directly applicable to multiplicatively-weighted Voronoi diagrams: For multiplicatively-weighted Voronoi diagrams one has to employ cones with different inclinations, which implies that the Voronoi diagram may no longer corresponds to the projection of the lower envelope of the cones to the *xy*-plane. Rather, the complex interaction of the cone geometries may give rise to isolated portions in the pixel map ("islands") that may need to be taken care of appropriately. Similar problems arise when one attempts to compute the straight skeleton of a simple polygon by means of graphics rendering of the lower envelope of certain plane slabs.

It is important to know here that a few of the isolated portions may not actually be islands, but true features of the geometric structure being approximated. Such islands are referred to as *false islands*. Existence of false islands is understandable as for example, a multiplicatively-weighted Voronoi diagram can actually have disjoint Voronoi regions. Or, while approximating a straight skeleton, false islands may appear in the pixel map because of certain true islands isolating them from their corresponding slabs and all of them collectively forming a large complex island. Removal of the "island" tags from such false-islands purely depends on the geometric structure being approximated.

The false islands which appear while computation of the discrete straight skeleton are efficiently dealt by the proposed "stir" algorithm as it utilizes the outline of an island for recovery of the correct portions of the diagram underneath it. And, in case of a false island, the site corresponding to the false island participates in forming the outline of the complex island containing this false island. The reason for this is that a false island is actually a portion of the correct diagram and, therefore, its leading part exists in the outline of the complex island which contains it. The removal of this complex island is carried out in a way that the false island portions remain intact and, hence, the actual diagram is recovered. Figure 1 illustrates an example of such false islands (highlighted in red) while sketching the straight skeleton of a simple polygon.

In case of a multiplicatively-weighted Voronoi diagram, a false island pixel may be dealt by appropriately comparing its distance to the corresponding site and its distance to the sites which form the outline of the complex island containing this false island. The depth value of each pixel, here, is proportional to the distance of this pixel to its corresponding site in terms of its weight.



(a) Original pixel map. (b) Detected islands.

(c) False islands (red). (d) Recovered sketch of straight skeleton.

Figure 1: False islands encountered while approximating straight skeleton of a simple polygon.

1.2 Main Contribution

The identification and, more importantly, removal of islands is a prerequisite for being able to extend rendering-based computations to geometric structures where the normal projection of the lower envelope of a suitably defined set of plane slabs does not suffice to reveal a discretized approximation of the structure. This paper highlights this problem and presents remedies for its resolution.

The main contributions of the presented work are:

1. A robust algorithm for the detection of islands in the rendering of a lower envelope of plane slabs.
2. Introduction of the concepts of black lists, and outlines of islands for recovering correct portions of the geometric structure hidden underneath the islands.
3. An efficient approach to deal with multiply stacked islands.

To put the problem of island detection and removal in such a pixel map in an actual geometric context, we briefly discuss the construction of a multiplicatively-weighted discretized Voronoi diagram of point sites using the cones method (by Hoff et al. [HCK+99]), and our own graphics-rendering-based technique for computing a discretized straight skeleton of a simple polygon.

1.3 Witnessing islands while constructing weighted discrete Voronoi diagrams

This discussion refers to the well-known cones method [HCK+99] for the construction of discretized Voronoi diagrams in 2D; the authors claimed that it is easily extendable to multiplicatively-weighted Voronoi diagrams. For simplicity, let us suppose that the input contains only point sites. The distance functions, cones in this case, are multiplicatively weighted [OBS92] and thus have variable angles. Furthermore, consider the case as illustrated in Figure 2 in which sites a, b, and d, have the same weight, however, site c has a higher weight and thus a higher base angle of the corresponding cone drawn over it. Due to this wider angle, this cone at site c (shown in green color in figs. 2 to 4) is responsible for three islands in the cross-sectional area under consideration. These islands are illustrated by dash-dotted-green lines in fig. 2. The dashed lines represent the non-visible sides of the cones. The dotted-red, dotted-blue, and dotted-purple portions, in the same figure, represent the actually correct but hidden parts of the diagram due to islands.

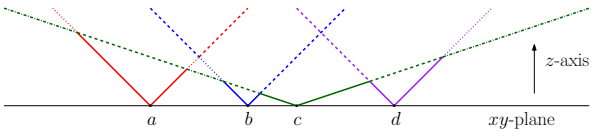


Figure 2: Islands arising due to site c.

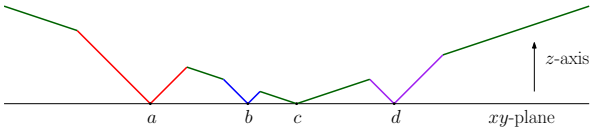


Figure 3: Visible lower envelope containing islands.

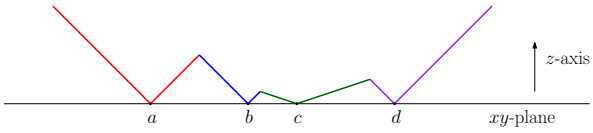
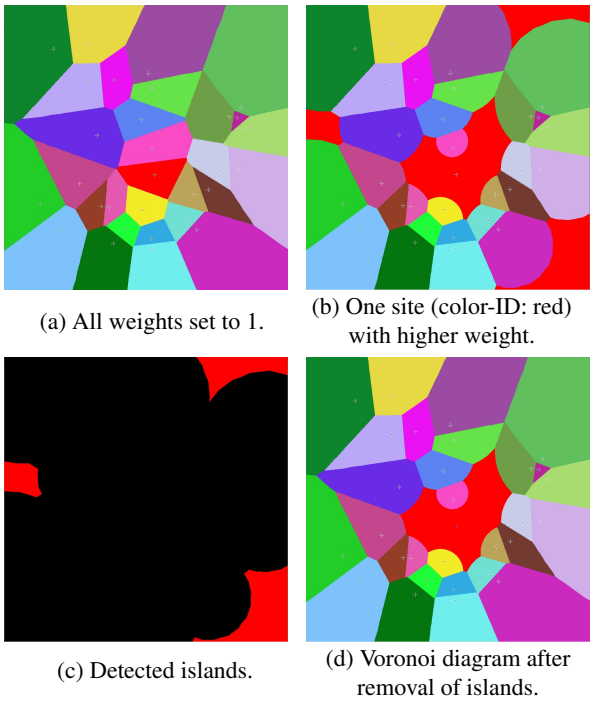


Figure 4: Lower envelope after removal of islands.

Similarly, if various sites carry different weights many islands may pop-up in the rendered pixel map which need to be taken care of. Figure 5 illustrates another example of islands experienced in a pixel map while computing a multiplicatively weighted discrete Voronoi diagram.



(a) All weights set to 1.

(b) One site (color-ID: red) with higher weight.

(c) Detected islands.

(d) Voronoi diagram after removal of islands.

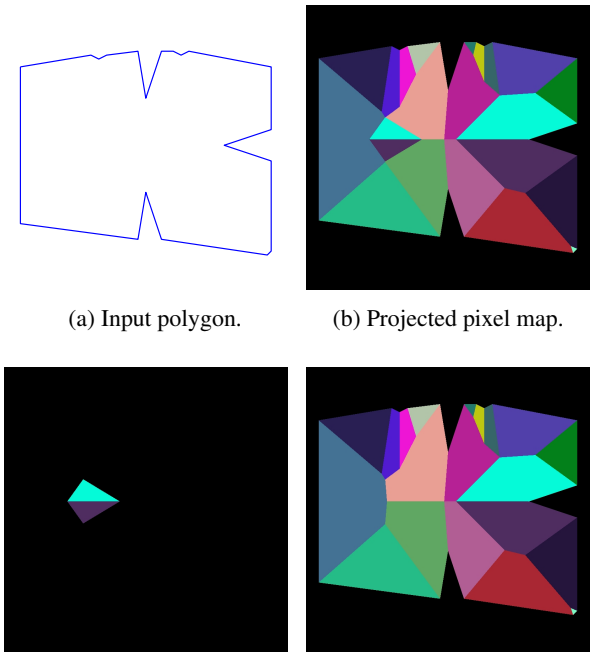
Figure 5: An example of islands appearing in the projection of the lower envelope of weighted distance functions while computation of discrete Voronoi diagram.

1.4 Experiencing islands while approximating straight skeleton of a polygon

Skeletons, being important structures for representation of 2D objects based on their topological characteristics, are of much interest in many geometry applications. The term "straight skeleton" was first tossed by Aichholzer et al. in 1995 [AAAG95], who are also the first ones to give an algorithm for computing the straight skeletons for interior of simple polygons. The straight skeleton of a polygon is defined by the set of lines traced out by its vertices during a shrinking process in which the edges of the polygon are moved inwards towards its interior, such that, at any instance of time, every shrinking edge is parallel to the original edge and the orthogonal distance between every shrinking and the corresponding original edge is the same. While this shrinking process, if any vertex passes over a non-adjacent edge, the polygon is split into two and the shrinking process is continued in each individually. The lines traced are known as *arcs* and their endpoints which are not vertices of the polygon are known as *nodes*. The arcs, the nodes, and the vertices of the polygon collectively define a graph embedded in the interior of the polygon which is its *straight skeleton*.

A straight skeleton is composed of only straight line segments and is closely related to, but not same as, the Voronoi-based medial axis offsetting scheme as the latter may contain circular arcs in addition to the straight lines. And, this fact makes straight skeletons advantageous over any other skeleton type as processing straight lines turns out to be less complex than handling some curved constructs.

Suppose we are given, as input, the n vertices of a simple polygon $\mathscr{P}$. Assuming $\mathscr{P}$ to lie on the $z = 0$ plane of the 3D euclidean space and using the general convention of assuming height of a point to be its z-value, certain slabs are drawn over each vertex. Each of these n vertices is assigned a unique color identity which is used as the color for its corresponding slab. The exterior of $\mathscr{P}$ contains background color of the scene since all n geometries grow from the edges towards the interior of $\mathscr{P}$. Moreover, the mutual interaction of these slabs is extremely complex as they penetrate each other at multiple locations. Hence, islands may exist in the parallel projection of the lower envelope of these slabs. These islands must be removed to achieve an accurate sketch of the straight skeleton of $\mathscr{P}$. Figure 6 shows an example island occurring due to the interaction of slabs constructed over the vertices of an input polygon.



(a) Input polygon. (b) Projected pixel map.

(c) Detected island. (d) Straight skeleton sketch.

Figure 6: An example of island in the projection of the lower envelope of slab geometries.

2 DETECTION AND REMOVAL OF ISLANDS – A GENERAL APPROACH

The approach is to process the parallel projection of the lower envelope of slabs, which are drawn by the employed geometric structure approximation technique, by detecting, uniquely identifying, and removing all islands in it. The projection is saved as a color pixel map and island detection is carried out on it as the first step. If any islands are encountered then the island labeling algorithm is invoked which labels every island making it uniquely identifiable. Subsequent to this, the pixel map is fed to the island removal procedure which generates a correct diagram patch for every island, and appropriately stitches these patches in the pixel map in place of the corresponding islands. If no islands are detected in the first step then this indicates that the output of the employed geometric technique is already correct and requires no further processing.

2.1 Detecting the islands

To detect islands in the projected pixel map, an island-boolean-map having resolution the same as that of the pixel map is maintained. The approach is to mark all pixels as island pixels first and then start changing their boolean flags based on the colors and flags of their neighborhood. Here, it is assumed that the input sites are projected adjacent to the background color of the scene (as in case of a polygon with background color on its exterior). Or, if this is not the case then the sites are projected in background color while still keeping the color identities of their corresponding slabs intact. In the latter case, the projections of sites are recovered back to their original colors later after island detection.

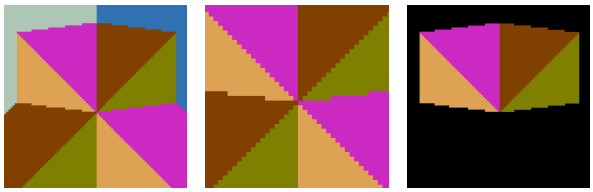
A pixel is marked as a non-island pixel only if its color is the background color of the scene, or if any of its neighboring pixels has the background color, or if any of its neighboring pixels has a color same as the pixel under question and that neighbor has already been marked as a non-island pixel. This leads to the island detection algorithm which iterates over the whole pixel map from one end to the other updating the pixel flags and leaving behind the actual islands. This continues until no pixel flag is updated during a pass of the pixel map.

The assumption of sites being projected in, or adjacent to, background color not only guarantees the generality of island detection but also ensures the island detection to be fail-safe. This is because, the islands are assumed as not to have any connectivity to their corresponding sites via their identically colored neighboring pixels and that the islands are detected based on their relationship to the background of the scene. Moreover, islands may be classified into two kinds based on their relationship to the sites responsible for their occurrence. If an island is due to only one site, it is called a *simple island*. And, if it is due to more than one sites then it is referred as a *complex island*. Islands due to different sites but lying immediately adjacent to each other are regarded as a single complex island. The sites responsible for existence of islands are called *bad sites*. In other words,

a bad site is the one a portion of whose corresponding slab is projected as an island in the pixel map.

2.1.1 *Four-neighbors versus Eight-neighbors testing*

Defining the term *neighbor* is of crucial importance for accurate island detection. Two candidate schemes, namely four-neighbors and eight-neighbors are the natural candidates for adoption. Considering only four neighbors (left, right, top and bottom) of a pixel to be its actual "neighbors" wins over the the other alternate in cases where some pixels of an island have diagonal connectivity to the non-island portions of their respective slabs. These island pixels are actually part of the island but are missed by the eight-neighbor-testing scheme as it assumes them as connected to their corresponding slabs and marks them as non-islands. However, the four-neighbor-testing is able to detect such pixels correctly and marks them as islands since it does not perform any diagonal connectivity checks. For an illustration of this fact, see Figure 7 which is taken from one of the experiments approximating the straight skeleton of a polygon.



(a) Portion of pixel map. (b) A zoomed portion of fig(a). (c) Detected island.

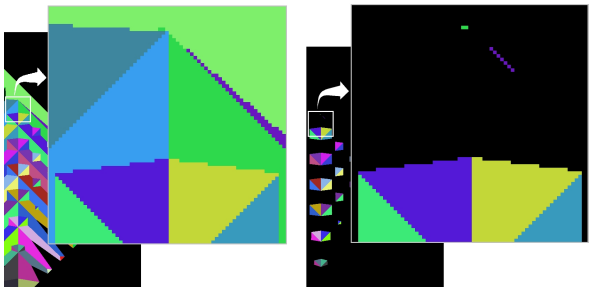
Figure 7: An island detected using four-neighbor connectivity testing.

Thus, eight-neighbors testing is not a good choice for adoption regardless of its early connectivity detection as compared to its counterpart. This leaves us with the four-neighbors testing scheme which works perfectly as it is capable of handling the cases which the eight-neighbors testing scheme fails to take care of.

The four-neighbors testing mechanism, as a drawback, requires an extra pass over the input pixel map for removal of single pixel thick areas which are actually not islands but are marked as islands due to the lack of diagonal-connectivity testing. These areas may occur as diagonally connected strips of single pixels. For an example, see Figure 8 which is taken from one of the experiments approximating the straight skeleton of a polygon.

2.1.2 *Orders of processing the pixel map*

The order of processing the pixel map in subsequent iterations of the island detection algorithm is extremely important as it can greatly effect the overall number of iterations required to accurately detect the islands.



(a) Portion of the original pixel map. (b) Detected islands.

Figure 8: Single pixel thick islands detected using the four-neighbor testing scheme without an extra pass.

Moreover, since immediate neighbors of every pixel are to be analyzed, the island detection algorithm should process the pixel map within the bounding box (second pixel in the second row –to– second-last pixel in the second-last row).

Let us consider the processing of a pixel map starting from the lower left corner of the box area inside it and moving row by row towards the top right corner. All non-island pixels which are direct neighbors of the background color, or ones which have a connectivity to it via their left or bottom neighbors will be correctly marked in the very first pass of the pixel map. For each subsequent pass, a one-pixel thick layer of the not yet marked non-island pixels, adjacent to already marked non-island pixels, will pass the non-island-pixel test, thus, leaving behind the actual island pixels. Similarly, if the pixel map is processed from the top right corner to the bottom left corner, a large number of non-island pixels having connectivities to the background color via their top or right neighbors pass the non-island test in the very first iteration.

Thus, altering the order of processing the pixel map at every subsequent iteration greatly boosts the detection of islands. To achieve maximum performance, we apply a four-way processing of the pixel map. That is, processing in the following order: First Iteration – bottom-left-to-top-right, Second Iteration – top-right-to-bottom-left, Third Iteration – bottom-right-to-top-left, Fourth Iteration – top-left-to-bottom-right; and then repeating this order for any subsequent iterations.

Experiments on various datasets using different processing orders (one-way, two-way, four-way) prove the four-way-four-neighbor testing to be the best choice. For details, please see section 3. Two sample runs of the island detection algorithm using four-way-four-neighbor testing are illustrated in Figure 9.

2.2 Labeling the islands

One of the island removal techniques presented in this paper requires unique identification of every island in

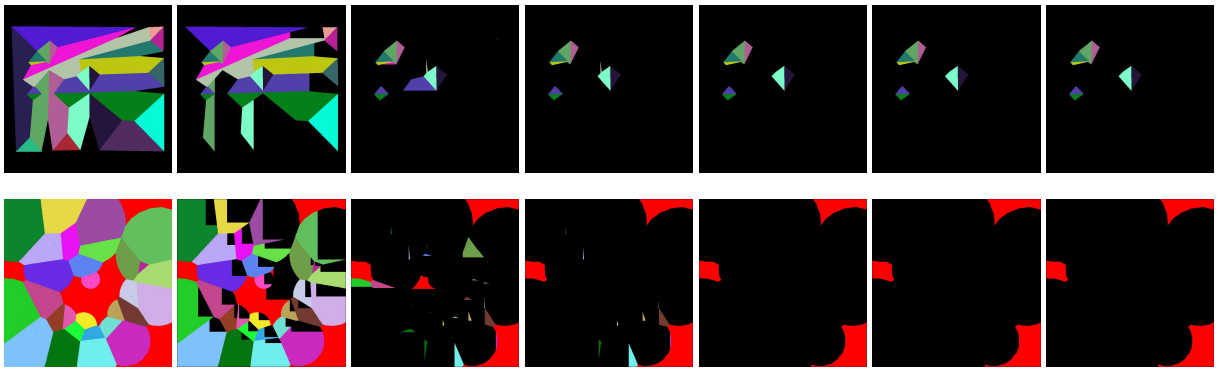
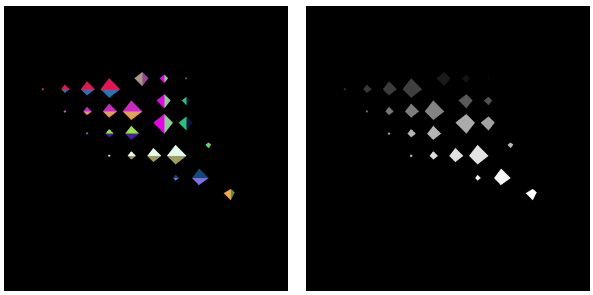


Figure 9: Progress of each iteration (left-to-right) of the four-way-four-neighbor testing for island detection in example pixel maps (left most in both rows) projected during approximation of the straight skeleton of a simple polygon having twenty vertices (first row), and the weighted Voronoi diagram of thirty randomly generated point sites (second row). The right most image in both rows contains the detected islands.

the pixel map thus laying the basis of the basic requirement for an island labeling algorithm. Moreover, in some later parts of the geometric solution being computed, the overall technique may also require information relating some particular island(s) for which we must be able to uniquely identify and distinguish it/them.

Let us assume that island detection has already been applied to the input pixel map and some islands do exist which need to be labeled. The island labeling algorithm maintains an island-label-map having resolution the same as that of the pixel map. It begins by initializing the island-label-map to all zeros – zero is regarded as the non-island label. The algorithm parses the island-boolean-map and considers only island pixels. For every such pixel, it generates a new label and assigns it to this pixel by recording this label at the pixel's position in the island-label-map. Along with this, it compares the labels of the neighboring pixels. The eight-neighbor scheme is adopted here for defining the term "neighbor". This boosts the performance as the connectivity information does not effect a pixel's island flag and, rather, lets the determination of labels of the diagonal neighbors beforehand. Now, if a neighbor of the current pixel has a non-zero label, and this label is smaller than that of the current pixel, then the algorithm sets the current pixel's label same as that of the neighbor pixel just tested.

Looping the island-boolean-map and the island-label-map with this simple testing scheme assigns unique labels to all islands. Thus, the underlying idea is similar to the standard connected component labeling techniques, that is, to assign a temporary label to each island pixel and update it depending on whether its immediate neighbor pixels also belong to the same island. As an example of the labeling output, Figure 10 illustrates an island map and the corresponding island-label-map with labels converted to the levels of gray.



(a) Sample island-map.

(b) The corresponding island-label-map.

Figure 10: An example grid of island labels with labels converted to the levels of gray.

Similar to the island detection, different orders of processing of the island-label-map can be adopted for labeling the islands. Application of a two-way processing of the pixel map ensures a generally speedy approach, since the eight neighbor scheme is employed here.

2.3 Removing the islands

Removal of islands is the most important step towards computation of the geometric structure being approximated. This section presents two island removal algorithms which can be adopted depending upon the types of the islands detected, and the requirements of the problem being addressed. If no complex island is detected in the pixel map then the first algorithm is used which operates per bad site and hence has a worst case complexity of the number of input sites (n) times the resolution of the input pixel map. This algorithm is named *sir* abbreviating "Simple Island Removal".

In worst scenarios, if more than one bad sites exist, it is not unlikely to have islands which are stacked over each other in a way that the top islands hide the ones underneath them. In such cases, removal of the visible islands in the pixel map may result in popping up of new islands in their places thus leaving the prob-

lem of island removal unresolved. The second island removal algorithm is capable of handling such stacked islands without requiring any extra looping over each stack of islands for their removal, hence, keeping the overall worst case complexity under controlled bounds. This variant of island removal is referred to as *stir* abbreviating "Stacked Island Removal". Thus, if removal of stacked islands is essential, or if any complex islands are encountered in the pixel map, then "stir" is used which operates per island and has a higher theoretical worst case complexity than "sir".

Both algorithms, "sir" and "stir", rely on a simple data structure which is called a *black list*. As the name suggests, it is a list of sites whose corresponding geometries are temporarily banned from being displayed in the scene based on certain criteria. The criteria, of-course, relates to involvement in creation of islands. In short, a black list is a sorted list of unique barred bad-sites. Moreover, both of these algorithms rely on the basic philosophy of patching the correct portions of the diagram in place of the islands. Temporarily black listing a bad site makes its geometry to be shutoff thus also shutting-off the islands due to it, and making the diagram underneath them visible on the lower envelope. These correct portions are patched in place of the islands in pixel map. The patch operations have been implemented as a parse of the pixel map while updating pixels relating to the island being processed. One could restrict the processing of entire pixel map in both "sir" and "stir" to bounding boxes of the islands being removed. This, however, does not really pay-off in real practice because of the overheads involved in computing and saving these bounding boxes.

2.3.1 Simple island removal – sir

The "sir" simply iterates on all bad sites, shutting-off their corresponding slabs one-by-one, and recovering the portions of diagram underneath the islands due to each of them. The worst case complexity of "sir" is thus $O(r \cdot w \cdot h)$, where r is the number of bad sites ($r < n$), w is the width, and h is the height of the pixel map. Considering $m = w \cdot h$ to be the resolution of the pixel map, the worst case time complexity bounds to $O(r \cdot m)$. The pseudo-code of "sir" is outlined in algorithm 1.

2.3.2 Stacked island removal – stir

It is straightforward to see that the slabs projected immediately adjacent to an island are the ones which actually constitute the correct portion of the diagram underneath it. Thus, black listing all bad sites except those participating in forming the *outline* of an island removes any possibility of more islands being stacked underneath it. The "stir" relies on this concept and utilizes the outline information for providing a general approach to remove stacked and complex islands. The pseudo-code of "stir" is outlined in algorithm 2.

Require: $pixelMap[\ ]$ = the pixel map to process; $islandBoolMap[i] = true$ if pixel i is an island pixel; $noOfIslands$ = total number of islands encountered; $noOfBadSites$ = total number of bad sites.
Ensure: $pixelMap[\ ]$ = correct pixel map with no islands

```
1:  if ( noOfIslands > 0 ) then
2:      for badSite = 1 → noOfBadSites do
3:          blackList ← badSite, and re-render the scene
4:          tempPixelMap ← framebuffer
5:          for y = 1 → imageHeight do
6:              for x = 1 → imageWidth do
7:                  pix = (x,y)
8:                  if ( (islandBoolMap[pix] = true) AND
                    (pixelMap[pix] = colorID(badSite)) ) then
9:                      pixelMap[pix] ← tempPixelMap[pix]
10:                     islandBoolMap[pix] ← false
11:         blackList ← NULL
```

Algorithm 1: The core *sir* algorithm

Require: $pixelMap[\ ]$ = the pixel map to process; $islandBoolMap[i] = true$ if pixel i is an island pixel; $islandLabelMap[i] > 0$ if pixel $i \in$ island "$islandLabelMap[i]$"; $noOfIslands$ = total number of islands encountered; $noOfBadSites$ = total number of bad sites; $island.outline$ = set of bad sites present in the outline of $island$; $island.label$ = the unique label of $island$;
Ensure: $pixelMap[\ ]$ = correct pixel map with no islands

```
1:  if ( noOfIslands > 0 ) then
2:      for island = 1 → noOfIslands do
3:          for badSite = 1 → noOfBadSites do
4:              if ( badSite ∉ island.outline ) then
5:                  blackList ← badSite
6:          re-render the scene
7:          tempPixelMap ← framebuffer
8:          for y = 1 → imageHeight do
9:              for x = 1 → imageWidth do
10:                 pix = (x,y)
11:                 if ( islandLabelMap[pix] = island.label ) then
12:                     pixelMap[pix] ← tempPixelMap[pix]
13:                     islandBoolMap[pix] ← false
14:                     islandLabelMap[pix] ← 0
15:         blackList ← NULL
```

Algorithm 2: The core *stir* algorithm

The worst case complexity of "stir" is $O(i \cdot (r + w \cdot h))$, where i is the number of islands processed, r is the number of bad sites ($r < n$), w is the width, and h is the height of the pixel map. Considering $m = w \cdot h$ to be the resolution of the pixel map, the worst case time complexity bounds to $O(i \cdot (r + m))$.

Experiments prove that adopting "stir" in place of "sir" actually boosts performance when the number of input sites is fairly large. This happens because the graphics card has to render lesser number of geometries in each iteration as compared to rendering all geometries except one every time. Hence, the algorithm which appears theoretically worse, performs actually better on large datasets. Additionally, this adoption ensures removal of stacked islands as a positive side effect.

2.3.3 Handling dependency chains of islands

It is interesting to note that an island may exist due to the existence of another island in the pixel map. And

this may also be true for that source island. Thus, it is possible for a pixel map to contain chains of dependencies of islands. Hence, it may become necessary to remove some other islands before a certain island is removed. An example of a dependency chain of islands is illustrated in Figure 11. Some islands in such a chain are actually the true features ("false" islands) of the diagram being computed and appear isolated because of the other "true" islands in the chain which isolate them from their corresponding slabs. A straightforward application of "sir" in such cases may not give the desired results. Moreover, the "stir" algorithm may require more than one runs in such cases. As in the first run, it will remove the root of this chain, and in the subsequent runs the reset of the chain will be processed. Furthermore, if the dependency chain forms a cycle then the "stir" requires a run of the "sir" algorithm, prior to its application, to remove one of the islands in the chain in order to break the cycle.

Such situations can be avoided by marking the "false" islands, which form links in these chains, as non-islands prior to the application of any island removal algorithm. We name this process as *film* abbreviating "False Island Marking". Hence, the application of "film" before removing islands lets us efficiently handle the dependency chains of islands. Here we use the term *participants* to refer to the individual islands constituting the complex island which is a dependency chain.

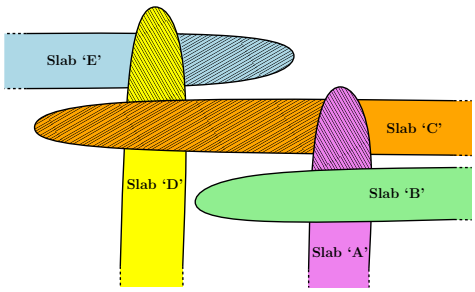


Figure 11: An illustration of a dependency chain of islands with true islands shown in falling tiling pattern, and false islands highlighted in rising tiling pattern.

False island marking – *film* All participants of a complex island are, by default, assumed to be "true" islands. The algorithm processes all complex islands in the pixel map and classifies their participants as *possibly-false*, *false*, and *true*. The classification is performed based on the relationship of a participant to the outline of the complex island being processed. First, all participants of the complex island being processed are analyzed. If the color of a participant is same as that of an outline constituent then that participant is marked as "possibly-false". As a next step, the outline adjacent to every "true" participant is checked for existence of a constituent having color same as a "possibly-false" participant of this complex island. If found then that "possibly-false" participant is marked as a "false" participant. This is done because when this "true" participant will be removed then the participant which is now marked as "false" will actually get connected to its corresponding slab thus forming a portion of the diagram being processed (for instance, see the false island corresponding to slab 'C' in Figure 11). Following this, the outline information of this newly marked "false" participant is checked for existence of a constituent having color same as that of any other "possibly-false" participant of this complex island. If found then that participant is marked as a "true" participant. This is because, as this newly marked "false" participant is to stay in the diagram, it will isolate this newly marked "true" participant from its corresponding slab (for example, see the island portion corresponding to slab 'D' in Figure 11). If all participants of a complex island are marked as "false" then all of them are actually "true" participants and must be handled accordingly. Finally, the boolean flags and labels of all "false" participant pixels are appropriately updated.

The application of "film" provides three benefits: First, it breaks the cycles in the dependency chains of islands. Second, it ensures the accuracy and integrity of the diagram. And third, it reduces the count of participants constituting the complex island, hence, reducing the number of pixels to be processed for its removal.

2.4 Salient features of the algorithms

Generality For detection and labeling of islands, no assumption is made with respect to the input primitives as these algorithms purely operate on a discrete grid which does not grow or shrink depending upon any factors. The island detection and removal approach is generally applicable to all graphics rendering based algorithms as islands are one of the major challenges towards their completeness, and rendering slabs and projecting their lower envelope is a basic feature of all such techniques.

Simplicity These algorithms can be easily implemented on the available graphics systems as from/to GPU data transfer functionalities are vastly supported. Moreover, these techniques do not have any special case arising which requires special handling.

Robustness Our algorithms use the features of the projected pixel map to efficiently handle highly sophisticated problems due to islands. For instance, the "stir" algorithm is capable of handling stacked islands without requiring any extra looping over each stack thus giving highly efficient throughput.

3 RESULTS

The statistics discussed in this section have been recorded while conduct of our experiments using a standard PC with Intel Core i7-2600 CPU clocked at

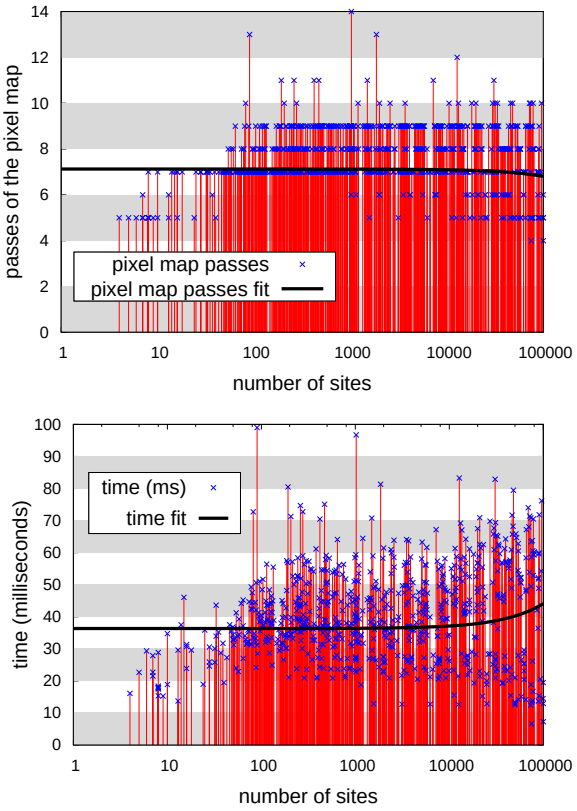


Figure 12: Plots of data relating sample runs of island detection using the four-way-four-neighbor testing scheme.

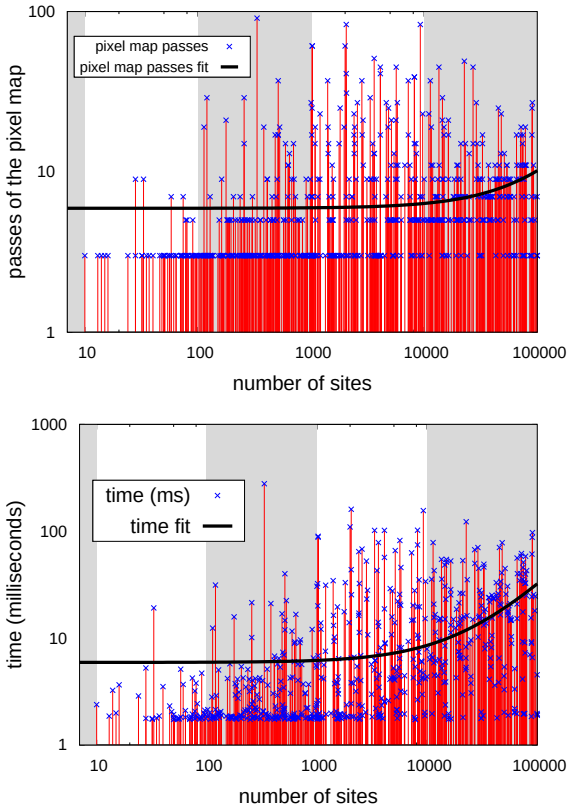


Figure 13: Plots of data relating sample runs of island labeling using the two-way-eight-neighbor testing scheme.

3.4GHz processor, and NVIDIA GeForce GTX 460 graphics card. The results presented here are based on tests with a pixel map resolution of 1000×1000. Handling datasets with more than a hundred-thousand point sites is not much meaningful with this resolution as this increases the chance of many very closely spaced sites being approximated by a single pixel. Therefore, the analytical data has been restricted to be bounded in the range $[1, 100000]$ for the number of input sites. Moreover, due to lack of space, the statistics relating weighted Voronoi diagram computation have been omitted. The presented results relate to the experiments approximating the straight skeletons. The Table 1, and Figures 12 & 13, summarize some information from the benchmarks.

It is evident from the figures presented in Table 1 that the four-way-four-neighbor testing scheme for island detection is a robust algorithm. This is also supported by the plots shown in Figure 12. Experiments show that, while detecting islands using this scheme, the number of passes of the pixel map do not grow with an increase in the number of sites being processed. Rather, this count of passes follows a constant value. Similarly, it can be observed in the second plot shown in Figure 12 that the increase in the time consumed by this island detection algorithm variant also does not exhibit any significant growth with respect to the number of input sites. A very slight increase in the time consumption occurs when the number of input sites crosses several tens of thousands and approaches the hundred-thousand limit. This is because a majority of pixels now pass the island test and require their island boolean flags to be updated.

Variant	Passes of pixel-map	Time (ms) consumed
1-way; 8-neighbor	455	3610.790
1-way; 8-neighbor; interlaced[1]	612	2771.825
1-way; 4-neighbor	612	2780.503
2-way; 8-neighbor	225	231.160
2-way; 8-neighbor; interlaced[1]	519	1848.751
4-way; 8-neighbor	14	104.622
4-way; 4-neighbor	14	99.104

Table 1: Peaks of data relating sample runs of various variants of the island detection algorithm applied to same datasets.

[1] Odd and even rows of pixels are processed in alternate iterations, similar to the interlaced raster scanning for scan-conversion of an image.

Figure 13 presents the plots of data relating sample runs of the island labeling algorithm using the two-way-eight-neighbor testing scheme. The least-squares line fit both for the number of passes of the pixel map, and for the time consumed by the labeling algorithm, with respect to the number of input sites, follows a constant value until the size of the input reaches an order of 10^4. A slight increase is observed in both of these trends beyond this level. The reason for this ascent is the increase in the number of island pixels requiring their label information to be updated. The increase in this case can observed slightly more significant than the one in case of island detection. This is because every island pixel now requires eight comparisons for getting its label information updated in contrast to the earlier case where every island pixel required four comparisons for having its island boolean flag modified.

4 CONCLUDING REMARKS

This paper highlights a very challenging problem (existence of islands) faced by geometric algorithms which make use of the graphics rendering and interpolation capabilities of a GPU. An efficient and general solution to this problem is proposed. The presented algorithms can be applied in combination with any such geometric technique thus ensuring its conformance to the task. The concept of black listing bad sites has been introduced and efficiently utilized for recovering correct diagram patches in lie of the islands. An incremental application of this patchwork on the pixel map transforms it into the required final diagram of the geometric structure being computed. The proposed approach ensures the accuracy of a patch by avoiding blacklisting of the sites which actually form the correct portion of the diagram in replacement of the corresponding island. This also certifies removal of all islands which may be stacked under this island at no extra processing cost. Moreover, the proposed algorithms also serve as a few steps towards the first ever GPU-based attempt to approximate the straight skeleton of a simple polygon.

ACKNOWLEDGEMENTS

This work was supported by the Higher Education Commission (HEC), Pakistan, and the Universität Salzburg, Austria.

REFERENCES

[AAAG95] O. Aichholzer, F. Aurenhammer, D. Alberts, and B. Gärtner. A novel type of skeleton for polygons. *Journal of Universal Comp. Sc.*, 1(12):752–761, 1995.

[CT05] J. Champagne and W. Tang. Real-time simulation of crowds using Voronoi diagrams. In *Theory and Practice of Comp. Graphics*, pages 195–201, Canterbury, United Kingdom, 2005. Eurographics Association.

[Den03a] M. O. Denny. *Algorithmic Geometry via Graphics Hardware*. PhD thesis, Universität des Saarlandes, Saarbrücken, 2003.

[Den03b] M. O. Denny. Solving geometric optimization problems using graphics hardware. *Comp. Graphics Forum*, 22(3):441–452, 2003.

[FG06] I. Fischer and C. Gotsman. Fast approximation of high-order Voronoi diagrams and distance transforms on the GPU. *Journal of Graphics, GPU, and Game Tools*, 11(4):39–60, 2006.

[Hae90] P. Haeberli. Paint by numbers: Abstract image representations. In *SIGGRAPH '90: Proc. of the 17th annual Conf. on Comp. Graphics and Interactive Techniques*, pages 207–214, New York, NY, USA, 1990. ACM.

[HCK+99] K. E. Hoff, T. Culver, J. Keyser, M. Lin, and D. Manocha. Fast computation of generalized Voronoi diagrams using graphics hardware. In *SIGGRAPH '99: Proc. of the 26th Annual Conf. on Comp. Graphics and Interactive Techniques*, pages 277–286, New York, NY, USA, 1999. ACM.

[LZC09] C. L. Li, G. Zhou, and C. W. Chan. A graphical approach to approximate offset computation. In *Proc. of the 6th Intl. Conf. on Comp. Graphics, Imaging and Visual.*, CGIV '09, pages 217–221, Washington, DC, USA, Aug 2009. IEEE Comp. Society.

[OBS92] A. Okabe, B. Boots, and K. Sugihara. *Spatial tessellations: Concepts and applications of Voronoi diagrams*. John Wiley & Sons, Inc., New York, NY, USA, 1992.

[VR05] M. Vona and D. Rus. Voronoi toolpaths for PCB mechanical etch: Simple and intuitive algorithms with the 3D GPU. In *IEEE Intl. Conf. on Robotics and Automation*, pages 2759–2766, 2005.

[WND97] M. Woo, J. Neider, and T. Davis. *OpenGL Programming Guide (2nd ed.): The Official Guide to Learning OpenGL Version 1.1*. Addison-Wesley Longman Pub. Co., Inc., Boston, MA, USA, 1997.

[Yam05a] O. Yamamoto. An acceleration technique for the computation of Voronoi diagrams using graphics hardware. In *ICCSA (Part 1): Intl. Conf. on Computational Sc. and its Appl.s*, volume 3480 of *Lecture Notes in Comp. Sc.*, pages 786–795. Springer Berlin / Heidelberg, 2005.

[Yam05b] O. Yamamoto. Fast computation of three-dimensional convex hulls using graphics hardware. *Japan Journal of Industrial and Applied Mathematics*, 22:291–310, 2005.