UPPSALA
UNIVERSITET

PROJECT REPORT

Implementation of a fast and efficient concave hull algorithm

Emil Rosén
Emil Jansson
Michelle Brundin

Project in Computational Science

Januari 31, 2014

DEPARTMENT OF INFORMATION TECHNOLOGY

Contents

Abstract

The calculation of convex and concave hulls to sets of points in two dimensional space is a problem found in many different areas. For calculating a convex hull many known algorithms exist, but there are fewer for calculating concave hulls. In this project we have developed and implemented an algorithm for calculating a concave hull in two dimensions that we call the Gift Opening algorithm. The idea is to first calculate the convex hull and then convert the convex hull into a concave hull. The convex hull can be calculated with any known algorithm. We implemented and compared Gift Wrapping and Divide and Conquer for this purpose. We found the performance of Divide and Conquer to be better and used that in our final prototype. We also developed and implemented the part that converts the convex hull to a concave hull. The performance and scaling of the algorithm is very good, mainly due to the optimizations we did to the algorithm and implementation. The concave hull for data sets with over 10^7 points can be calculated within a few seconds on an Intel Core 2 duo PC. The algorithm is also very reliable and produces a good quality concave hull even when the geometry is very odd or when the density of the points is varying.

1. Introduction

1.1 Introduction

Finding the concave or convex hull for a set of points is a very general problem that can be useful in many different situations. It can be used to define the shape of a set or find minimal area that encloses the set. It can be applied in many different areas, for example pattern recognition and image processing [1].

In this report we will describe our solution to the problem of finding a concave hull. We will talk about and compare the Gift-Wrapping and Divide and Conquer approach to calculating the convex hull and see how they perform on different test cases. We will also describe the Gift-Opening algorithm that we developed that converts a convex hull to a concave hull.

We also tried an approach described in [2] based on delaunay triangulation but abandoned the implementation because it was too slow.

1.2 Aim

The goal of this project was to implement an algorithm that calculates the concave hull for a set of points in two dimensions. This can be done by either researching and testing known algorithms or by developing a new algorithm. Another goal was to parallelize the algorithm as much as possible to run it on multi-core CPU or GPU.

The algorithm should be stable and must be able to handle any kind of set while giving a good quality resulting hull. This means that the hull should be one single polygon, that there are no crossed segments, and that all points are contained within the hull. In addition to this, the most important aspect of the algorithm is that is should be fast on big input sets. Calculations on sets of size 10^7 should be completed within 10 seconds. Also, from a user perspective, the algorithm must be easy to use, meaning that no additional parameters, except for the input set, should be required from the user.

2. The Concave Hull

A convex hull of a set of points is the uniquely defined shape that minimizes the area that contain all the points, without having any angle that exceed 180 degrees between two neighbouring edges, as seen in *Figure 2.1a*. To find the convex hull of a set of points is a well-known problem studied in detail and there are many existing algorithms.

A concave hull of a set of points could be defined as the shape which minimizes the area of the containing shape, but allowing any angle between the edges. Usually we don't want the minimal area since that can easily distort a figure, see *Figure 2.1d*. As seen in *Figure 2.1b* or *Figure 2.1c* we rather want a hull which is sufficiently concave, which in turn will make the definition of the searched for hull ambiguous. All sets of shapes between the convex hull, and the one that completely minimizes the area, would be considered concave.

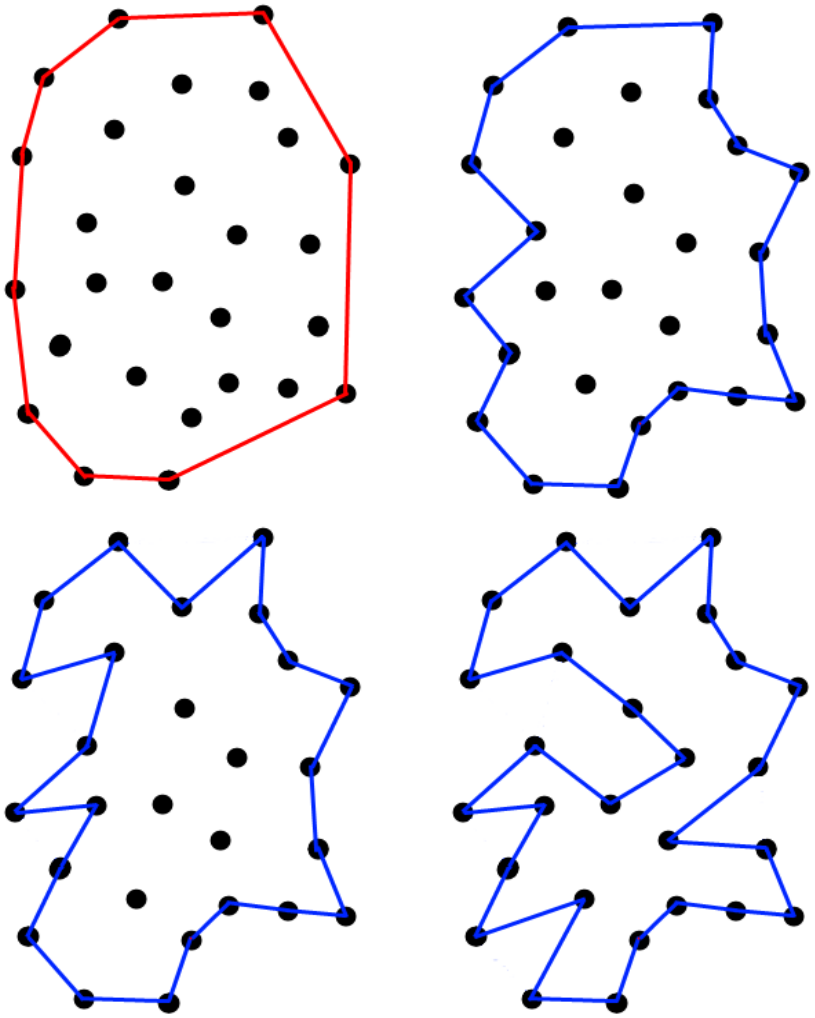

Figure 2.1a: Convex hull of a set of points (Top left). No angle between two neighbouring edges have angle greater than 180 degrees.
Figure 2.1b: One concave hull (Top right) of a set of points. The area is smaller than the convex hull but it does not minimize the area.
Figure 2.1c: A slightly "more" concave hull (Lower left).
Figure 2.1d: Even more concave hull, distorted. In bigger sets the concave hull can get even more distorted.

In this project we want a concave hull that defines and outlines the general shape of the set of points. We want the hull to be relatively smooth, but still show as much detail as possible in the set. From the four hulls in *Figure 2.1*, the hull in *Figure 2.1b* would probably be the the preferred solution. It outlines the shape of the set, while not being as distorted as *Figure 2.1c* and *2.1d*.

3. Gift opening

We developed our own algorithm to calculate the concave hull by first calculating the convex hull using some well known algorithm. The convex hull is then iteratively opened up to create the concave hull.

3.1 Gift wrapping

One way to calculate the convex hull is by using gift wrapping [3]. One point on the boundary is chosen, most simply a point with minimum or maximum x or y coordinate. The angle is measured between the selected point and all other points in the dataset. The point which have the best angle will be chosen as the next boundary point. This procedure will continue until the starting point is reached.

This algorithm has complexity $O(nh)$, where n is the number of points and h is the number of points on the boundary. This means it is fast to calculate most convex hulls but will be very slow if all point are on the convex boundary.

This algorithm can also easily be parallelized by starting the algorithm at each point with maximum or minimum x and y coordinates, dividing the problem into four different subproblems.

3.2 Divide and conquer

The other convex hull algorithm that was tested is an algorithm based on divide and conquer, as described in [4], but optimized since we are only interested in the convex hull. To find the hull, the points are first sorted in, for example, lexicographic order on the x coordinate using quicksort.

All points are divided into sets of one or two neighbouring (according to the sorted axis) points. The points are then merged together creating an edge between them. The resulting subset of edges are merged together to create subsets of convex hulls, where the points on the boundary know about their clockwise and counterclockwise neighbouring boundary points. This goes on

recursively until all subsets have been merged together into one convex hull.

The merge phase is the costly phase of the algorithm. We have to find the common lower and upper tangent to merge the two hulls together. To find the common tangents a technique called walking, see *Figure 3.1*, is used. First, the point with the biggest x coordinate from the left hull and the one with the smallest x coordinate from the right hull are chosen, since we know the edge between them will not cross any edges in the two boundaries and that they are both on the boundary. To find the upper tangent we walk one step up to the left, meaning we draw an edge between the point counterclockwise of the starting point on the left hull.

This walking procedure continues to walk up on both the left and right side until taking one more step on boths sides will create an edge that would cross an edge on the boundary of either subset. We can measure this by taking the two dimensional cross product between the last edge and the new edge after each step and checking the sign of the product, taking into consideration if we are walking up or down and if we are walking on the right or left side. If we can't take a step in either direction, the upper common tangent is found.

The same procedure is performed when finding the lower tangent but with the exception that we walk down instead of up.

In case the cross product would give zero, the two edges are collinear. In the implementation zero is always a valid step, this is necessary since there probably are points that are collinear. Therefore, to prevent infinite loops if all points in two subsets are collinear, each step is also checked to see if we returned to the origin point on each polygon, in which case the step is invalid.

Both the merge phase and the quicksort are branched into two subsets in each recursive call. Each recursive call can then be put on its own thread, optimally only up to a certain level so we don't get that much overhead by using too many threads. Additionally, finding the upper and lower common tangents can be done on two separate threads.

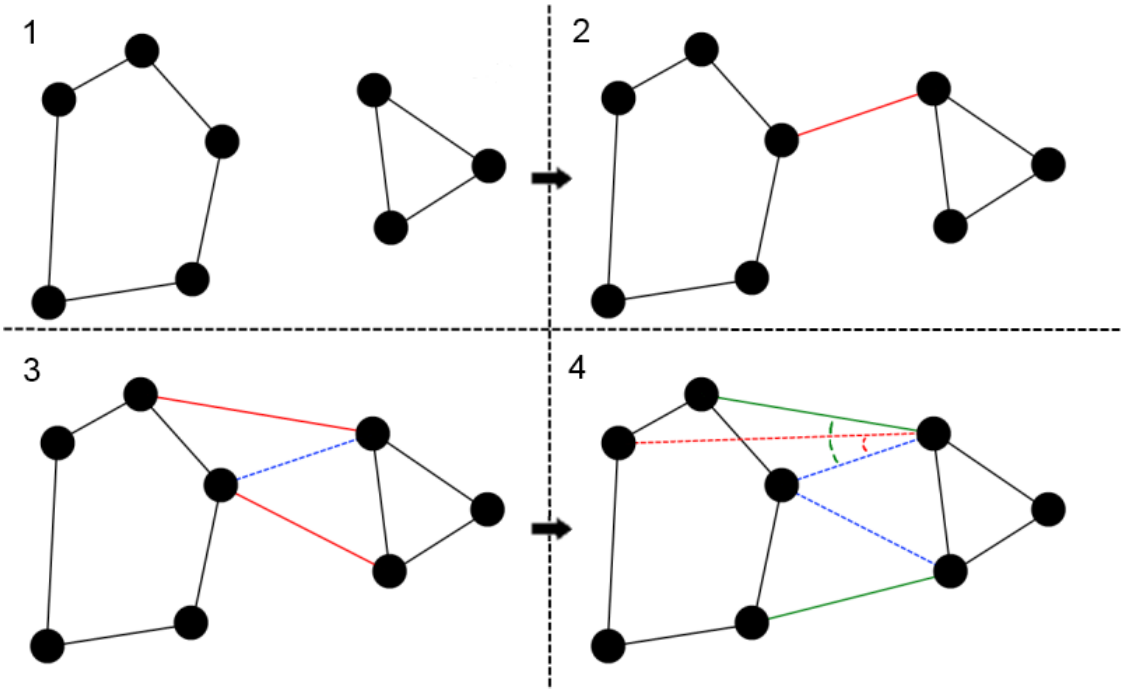

Figure 3.1a: Example of two convex polygons to be merged together. Each point on the boundary knows about their clockwise and counterclockwise neighbouring boundary point.
Figure 3.1b: We choose the point with the biggest x value from the left hull and the one with the smallest from the right hull. This is simple since the points are sorted in that order.
Figure 3.1c: We start to walk up and down to find the lower and upper common tangent.
Figure: 3.1d: We need to take one more step to find the lower common tangent (lower green). If we take one more step to the left when we walk up we see that the edge is not valid (red dotted edge) and that the line between that edge and the old edge (upper green) is smaller. We know the next point would only be valid if it were above the last point, giving a bigger angle. This test is fast if a two dimensional cross product is used, then we only need to check the sign of the cross product. We would take a step on the right side and see if it is valid, but it is not so the upper common tangent is found.

3.3 Concave "opening" phase

The concave phase is inspired by [2] and starts with adding all boundary edges to a list where they are sorted by length. The longest edge from the list is then selected and removed from the list. All points are then searched to select the best candidate to add to the boundary between the endpoints of the selected edge. The search criteria for the points are that the largest angle from the new point to the end points of the old edge should be as small as possible, see *Figure 3.2*. This criteria ensures that no point will end up outside the hull. If a point ends up outside the boundary when this selected point is added to the boundary, this means that both the angles of the outside point are lower than those of the selected point. This can not happen since we selected the point with the smallest

maximum angle.

A test is then performed to make sure that the hull will still be a valid single polygon if the point is added and the selected edge is removed. The test consists of checking that the point isn't already on the boundary and the new edges do not intersect any other edges. If the hull is still a polygon then two new edges, between the new point and the points on the old edge, are added to the list. If the hull isn't a polygon then the selected edge is added to another list that contains the final boundary. This is then repeated until all edges in the list have a length below a max length threshold.

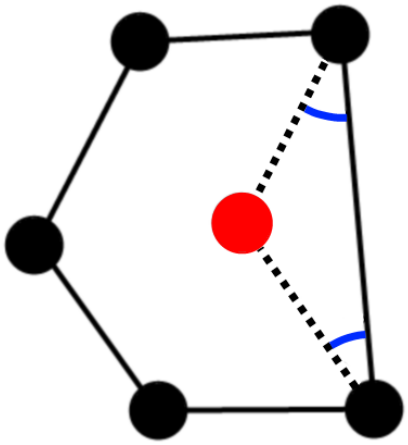

Figure 3.2: The angles (blue) between the longest edge and the new edges (dotted lines) that would be created if a point (red) would be added to the boundary are shown. The largest of these two will be used when trying to find out how good it would be to add this point to the boundary.

The algorithm can be altered to give a concave hull that is more likely to look like what is probably desired by adding a requirement when testing if a point should be added or not. The requirement is that the angles seen in *Figure 3.2* for the best point should be smaller than 90°. Without this check there is a risk that concave bends with very sharp angles are added to the hull. This is illustrated in *Figure 3.3*.

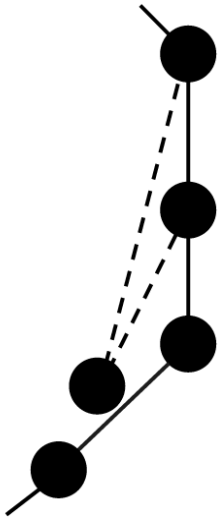

Figure 3.3: Illustration of what could have happened if we allowed too big angles when adding new points to the hull.

Since the density of the points may vary, the max length threshold of an edge required to split it is calculated locally. It is calculated by measuring the density of points in areas nearby to the edge. For practical reasons we use the boxes described in *section* 4.2 as the areas. One of the points of the edge is chosen and the density of points in neighbouring boxes to that point is measured. The density is measured by counting the number of points in that box and dividing with the area. The box with the highest density is then selected and by assuming that the points are equally spread out, the expected distance from a point to it's neighbours can be calculated. This value multiplied with a constant is compared to the edge length to decide if the edge should be removed. Since it is not exactly defined how concave the calculated hull should be, this constant can be used to vary the concaveness of the hull. The lower this constant is, the more concave the hull will be and it will have more and smaller concave bends. With a higher constant the hull will be less concave and not show as much detail.

3.3.1 Pseudocode of the concave phase

```
Data: list A with edges for the convex hull
Result: list B with edges for a concave hull

Sort list A after the length of the edges;

while list A is not empty
      Select the longest edge e from list A;
      Remove edge e from list A;
      Calculate local maximum distance d for edges;

      if length of edge is larger than distance d
            Find the point p with the smallest maximum angle a;

            if angle a is small enough and point p is not on the boundary
                  Create edges e2 and e3 between point p and endpoints of edge e;

                  if edge e2 and e3 don't intersect any other edge
                        Add edge e2 and e3 to list A;
                        Set point p to be on the boundary;

      if edge e2 and e3 was not added to list A
            Add edge e to list B;
```

3.3.2 Parallelisation

Since the edges are opened up depending on the length it is not straightforward to parallelize it. We did a simple parallelisation where we divided all the edges into several subsets, each run on a separate thread, and then merge the results in the end. The subsets will not be entirely independent since they all have to share a list of edges to check for intersecting edges.

4 Optimizations

To improve the performance of the algorithm we implemented some optimizations and also parallelized some parts of the algorithm.

4.1 Linearized angle

The Gift Opening algorithm relies heavily on calculating angles between edges, which is a very costly operation. However, we never need the absolute value of an angle, we only need to find the relative size of angles so we can compare which angle is the biggest. This means that we can linearize the angle function.

Instead of calculating an angle based on the unit circle, we calculate it from a "unit" square, see *Figure 4.1*. The angle is then, in our implementation, between 0 – 8 instead of between 0 – 2π.

To calculate this "angle", each side in the square is first ordered clockwise. Which side of the square the edge belongs to is determined by which side the edge would intersect, ie determined by the slope of the edge. For this to work the sign of the slope need to be consistent through all edges, so in our implementation the points on the boundary are stored in clockwise order.

In the case two edges intersect two different sides of the square, to determine which edge have the biggest angle is simply to determine which side is further clockwise. For example, an edge intersecting the bottom side will have an angle between 1 – 3 while an edge intersecting the left side will have have an angle between 3 – 5 and will thus be bigger.

If on the other hand two edges are on the same side, we will have to measure the magnitude of the slope rather than which side it intersects. This is done by mapping the minimum and maximum valid slope values for the current side and the edge slope value to a value between the corners of the current side. This will make the angle to increase from zero to eight in clockwise order and we can measure two angles against each other, the sign of their relative size will still be valid as if a regular angle would be used.

A reference angle is also used to give a basis for the measurements. Using a reference angle of zero mean that we are as using an edge which points from the centre of the square to the right and is perpendicular to the right side of the square as a reference. All angles which will be measured are then calculated against this reference edge.

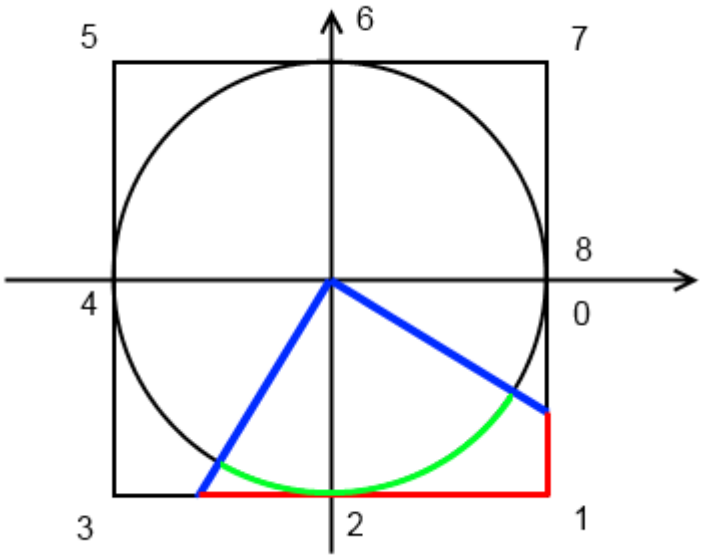

Figure 4.1: Linearized angle. Instead of using a circle to determine the angle a square can be used since its only the relative size of the angles' that matter. First, which sides of the square the edges(blue) cross will be calculated, then both edges are given a value by interpolating between the two corners of the sides that was crossed. The linearized angle will be the difference between the edge values which is equivalent to the length of the length of the red line. The normal angle value would have been equivalent with the length of the green line.

4.2 Space partitioning

To decrease the amount of points to search for we sort each point into an equidistant two dimensional grid. This can substantially decrease the amount of points we need to search through during the concave phase. The sort is only $O(n)$ operations since each point directly can be mapped to a box.

In the Gift Opening algorithm, when searching for a potential new point on the boundary while removing an edge, we will only consider points which are in boxes which lie in between the x or y coordinates for the points of the edge. when a potential candidate is found out of these points we don't have to continue the search. In the case we do not find a good potential point, we extend the search with one more layer of boxes in both x and y direction, and search through the new points. The search continues to be extended until we either find a potential point or the size of the searched area is considered to be big enough. The searched area is considered big enough if the amount of extended layers of boxes reaches a certain percent of the length of the longest axis, where the length is measured in number of boxes. See *Figure 4.2* for example.

The sizes of the boxes are determined by the maximum and minimum x and y coordinates and the total amount of points.

This method works well as long as the dataset is somewhat uniform. If most of the points are unluckily placed so that most of them are contained in the same box then the method will not work so well.

The partitioning was parallelized by simply dividing all points into different subsets, partition all subsets and then merge the result.

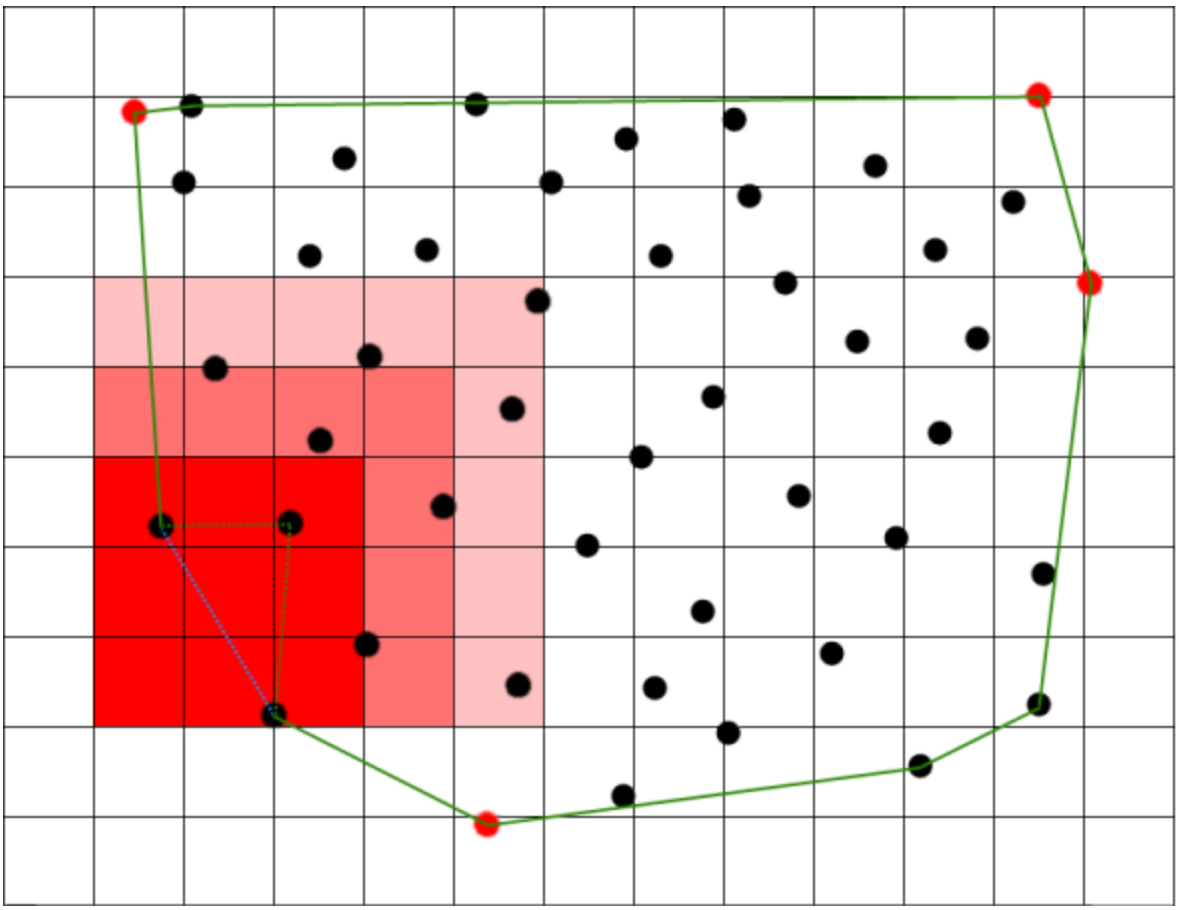

Figure 4.2: The green lines represent the convex boundary and the blue dotted line to the bottom left represent the edge we will check if it can be opened up. Only the points in the dark red area will be considered. The point we found seem suitable, which means we don't have to extend the search. If the polygon would still be valid the point is chosen and the blue dotted line will be removed and replaced by the two green dotted lines to the new point.

In case the point would not be a good candidate or if none were found, we would extend the search by one step as seen in the lighter red boxes. This search will continue to extend if no good candidates are found until a preset maximum length is reached.

In the real implementation the longest edges will be checked first and the grid will be aligned with the maximum and minimum (red) points.

By searching the boxes this way there is no risk that a point ends up outside of the hull due to one or more boxes have not been searched. The requirement to make sure that do not happen can be seen in *Figure 4.3*.

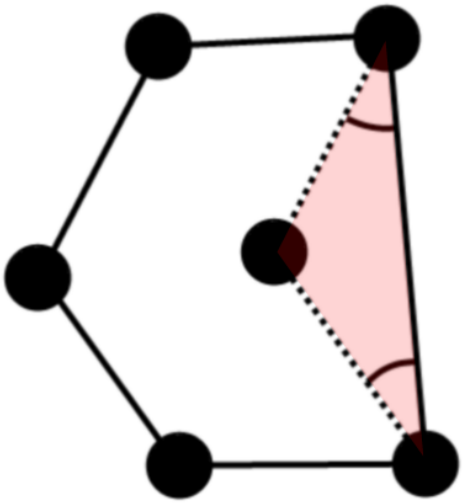

Figure 4.3: To prevent a point to end up outside the boundary the Area(pink), that will change from being inside to outside of the boundary, needs to have been searched to make sure that there is no point there. A point being inside that area is equivalent to both of the angles, formed between the old edge and lines from the edges endpoints an untested point, being smaller compared to the point selected to be added to the boundary.

4.3 Culling

Using the equidistant grid we can create an approximate convex boundary of the boxes in the grid. Since the amount of boxes are much smaller than the amount of points, creating this convex boundary is fast in comparison. Using this boundary we can then remove uninteresting points in the calculation of the convex hull.

To create this boundary, from all boxes in the grid which contain at least one point, we will add a point to a new list of points. The coordinate of these points will simply be the coordinate of the box in the grid, in our implementation that will be the lower left corner of each box. The points will be run through the same

convex hull algorithm as previously used and will create a approximate boundary of boxes.

To make sure no points will be left out we will create a new smaller boundary called the buffer boundary using the approximated boundary. This is simply done by decreasing the size of the boundary by one step in the direction of the normal vector of each face. The buffer boundary may however not be a convex boundary and can even have intersecting edges. We loop through the boundary until we removed all invalid edges and get out final convex buffered approximated boundary.

Only the points which are in boxes outside of the boundary can be potential candidates for the real convex boundary and are the only points that need to be considered when calculating the convex boundary. See *Figure 4.4*.

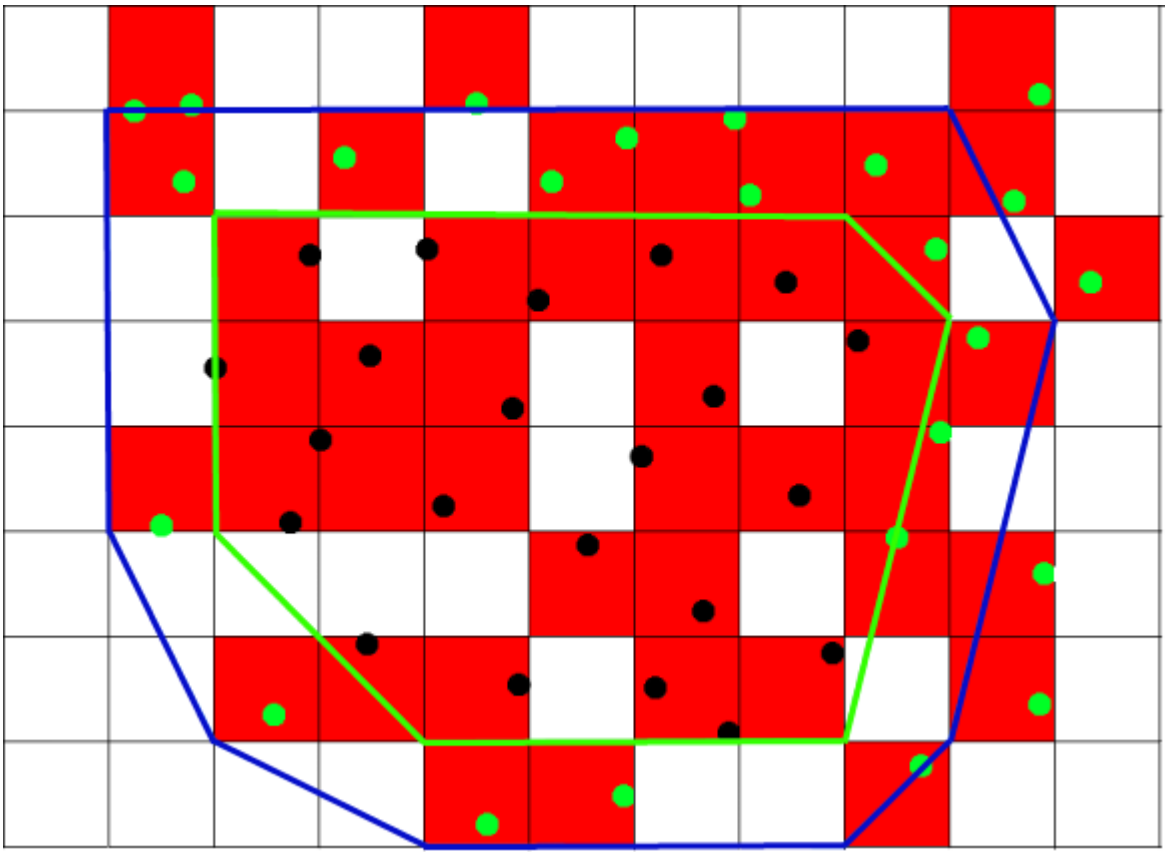

Figure 4.4: The green points are the only points that are potential candidates for the convex hull. The red boxes are boxes with at least one point and creates an approximated boundary (blue) where the lower left corner of each box is considered a point. The green line is a buffed boundary created from the approximated boundary to make sure we don't miss any potential points.

4.4 Quadrilateral culling

The culling algorithm described in *section 4.3* may fail if, for example, all points are placed in one big box. To improve the performance we apply a simple culling method after the first one to remove at least some points in case the first one would fail.

Using the points with minimum and maximum x and y coordinates we can create a quadrilateral, as seen in *Figure 4.5*. All interior points of the quadrilateral can be removed when calculating the convex hull since interior points of any convex hull, including subsets, could never be on the convex boundary. The amount of points that are removed depends heavily on the position of the minimum and maximum points, it can either remove all interior points or not a single point.

This algorithm loops once over all the points, hence its execution time depends on the number of points, *n*. In the case the culling algorithm in *section 4.3* does not fail, *n* would hopefully be significantly reduced and the added overhead from this part would be minimal. In case *n* is still big, the complexity is still lower than of the convex part and we would still get a speedup.

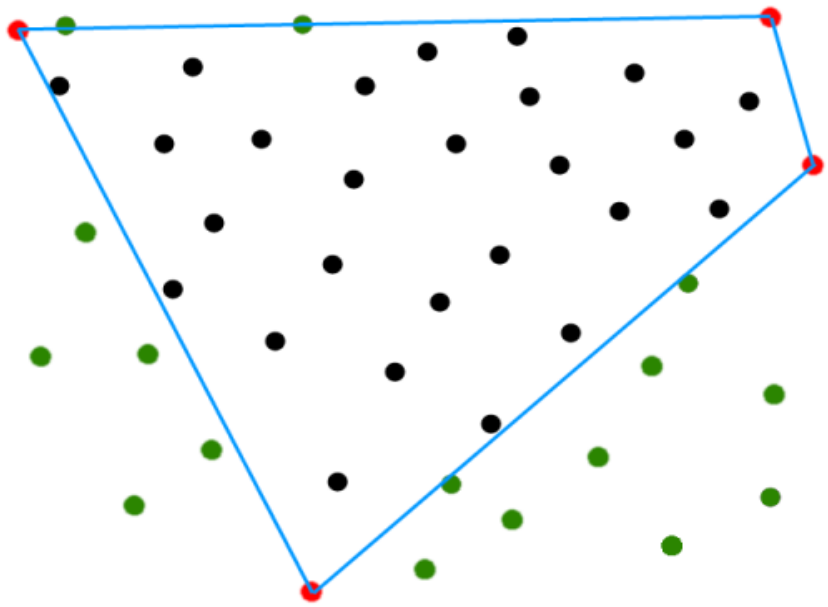

Figure 4.5: Only the green points need to be considered when calculating the convex hull. The maximum and minimum points (red) creates a quadrilateral and all points in the inside can be removed.

5. Concave Delaunay triangulation

During our project we also tried to implement a concave hull finding algorithm based on delaunay triangulation as described in [2].

The method first creates a convex delaunay triangulation and then iteratively removes edges that are smaller than a certain threshold. However, while the concave phase of this algorithm was deemed fast and the resulting hull was very high quality the creation of the delaunay triangulation was very slow. Therefore we decided to abandon the delaunay approach and focus on the gift opening algorithm.

To calculate the delaunay triangulation we implemented a random incremental insert according to [5], sorted incremental insert and a divide and conquer method using flipping according to [6].

6 Results

We performed all tests on an Intel core 2 duo E6850 processor at 3.0 GHz running 64-bit Windows 7 as operating system, unless stated otherwise. The code was compiled with the mingw g++ compiler [7] with optimization flag -O3.

To test the performance of the different algorithms we used several test cases. Some of the data sets were provided to us and and some were created by us by randomizing points on different two dimensional geometries. *Figure 6.1* (1) shows the set "*300k*" that contains 300 000 points. Set (2) is "*Australia*" and contains 1 000 000 points. Sets (3) to (7) are randomly created by us and can contain any number of points. Set (3) is "*circle"*, (4) is "*crescent"*, (5) is "*square"*, (6) is "*hourglass"*, and (7) is "*star"*.

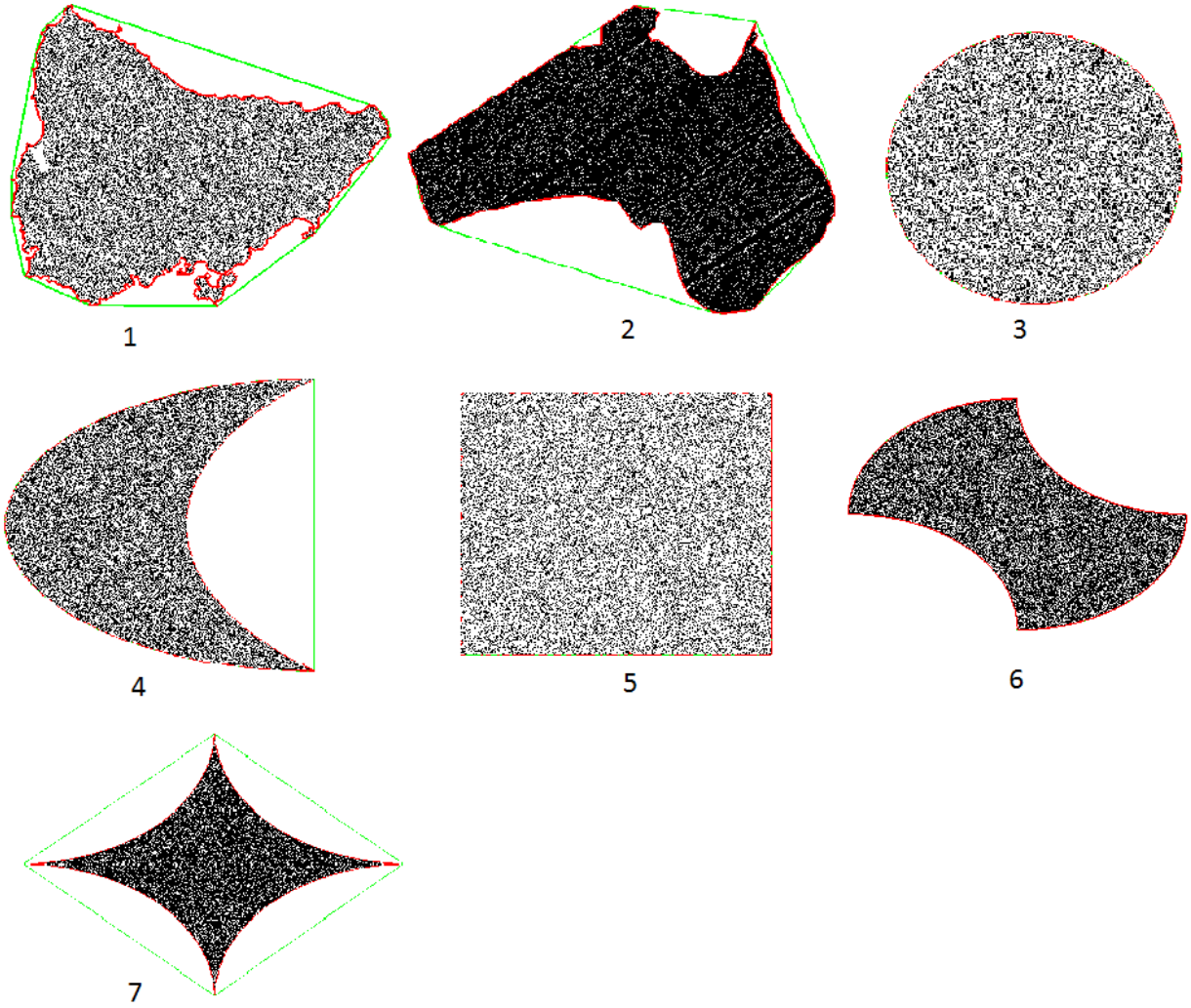

Figure 6.1: The test cases used when evaluating the algorithms. (1): "300k", (2): "Australia" (3): "circle", (4): "crescent", (5): "square", (6): "hourglass", (7): "star"

6.1 Gift-Wrapping and Divide and Conquer

Since part of our algorithm is to calculate the convex hull we tested the performance of both the Gift-Wrapping and the Divide and Conquer algorithms to see what algorithm was faster. We tested the algorithms on all our test cases and in *Figure 6.2* the results for the *circle*, *star, square* and *Australia* data sets are shown. In this case we used 4 000 000 points in the circle, star and square.

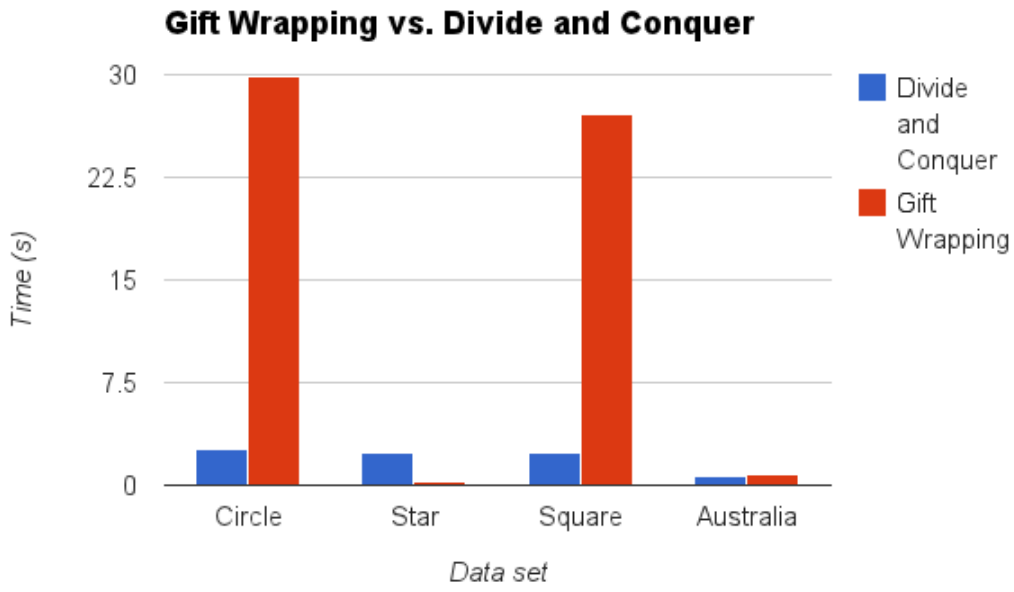

Figure 6.2: Performance of the Gift Wrapping and Divide and Conquer algorithms on the circle, star, square and Australia test cases. In the circle, star and square cases 4 000 000 points were used.

As can be seen, the performance of Divide and Conquer is consistent over the different geometries while Gift Wrapping show very inconsistent performance. On the *circle* and the *square* the performance is very bad compared to Divide and Conquer, while on the *star*, performance is much better than Divide and Conquer. In the *Australia* test case the performance is comparable, with Divide and Conquer being a little bit faster.

We choose to use the Divide and Conquer algorithm in our final implementation because the performance is consistent over different geometries and in most cases better than Gift Wrapping. The performance of Gift Wrapping is too dependant on the geometry of the data set, and is not suitable for an implementation where you want reliable performance. The culling optimizations that we have implemented is also more effective on Divide and Conquer than on Gift Wrapping. This is because Divide and Conquer has a complexity of $nlog(n)$ while Gift Wrapping has a complexity of nh, and since the optimization reduces the number of points n, it will help the performance of the divide and conquer method more.

6.2 Linearized angle

We tested how the performance of the linearized angle function compared to the performance of the atan2 function in c++ standard library. In *Figure 6.3* we can see both the total time and the time of the concave part when running the *crescent* test case.

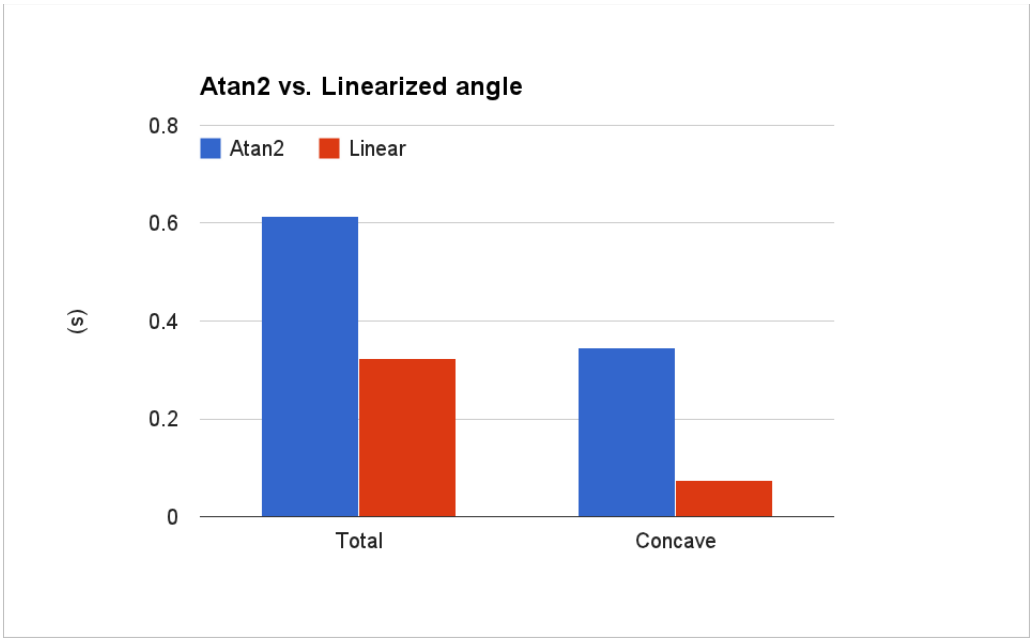

Figure 6.3: The performance of the atan2 function in c++ and the linearized angle function.

The total execution time is decreased two times when using the linearized angle instead of the atan2 function. Because the concave part is the predominant part of the total execution time we can see a big improvement in overall performance by using the linearized angle.

6.3 Space partitioning

We tested the box sort space partitioning optimization on the *crescent* test case with 1 000 000 points. In *Figure 6.4* the execution time with and without the optimization is shown. Note that the vertical axis is cut and not continuous.

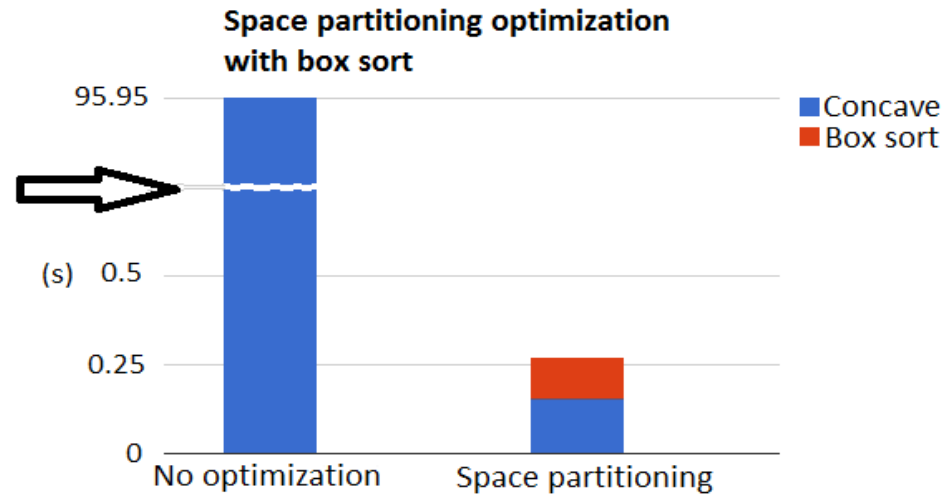

Figure 6.4: The execution time on the crescent test case, 1 000 000 points, with and without space partitioning optimization.

The optimization reduces the execution time from about 96s down to 0.25s which is a huge performance improvement.

6.4 Culling boundary

We tested the effectiveness of the culling boundary optimization by checking how many points it would remove when running on different test cases. In *Figure 6.5* you can see how many points the sets contained before and after the culling.

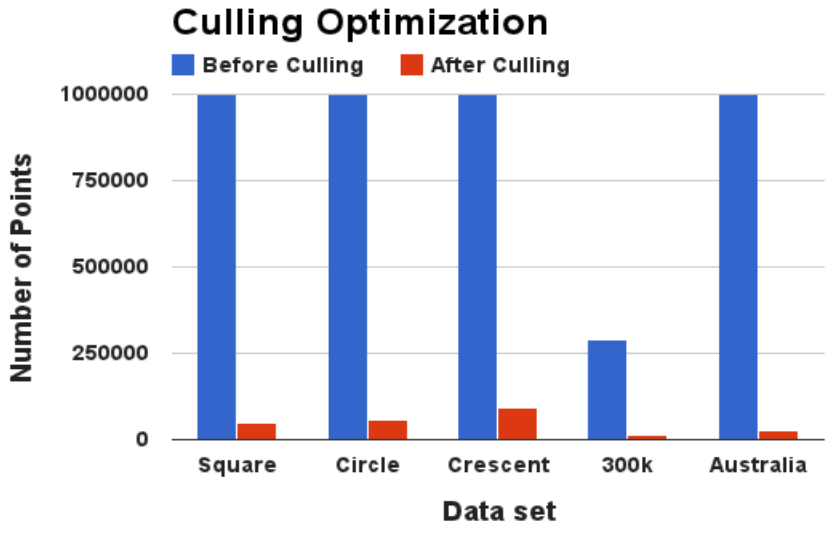

Figure 6.5: The number of points in the sets before and after the culling optimization.

The culling was very effective, on the sets with 1 000 000 points only a few thousand points remained after the culling. This increases the performance of the convex part significantly.

6.5 Optimal box size

In the space partitioning optimization the size of the boxes affects the performance. We therefore tested what box size that gave the best performance. In *Figure 6.6* you can see the execution time of different parts of the code when running on the *crescent* test case with 1 000 000 points.

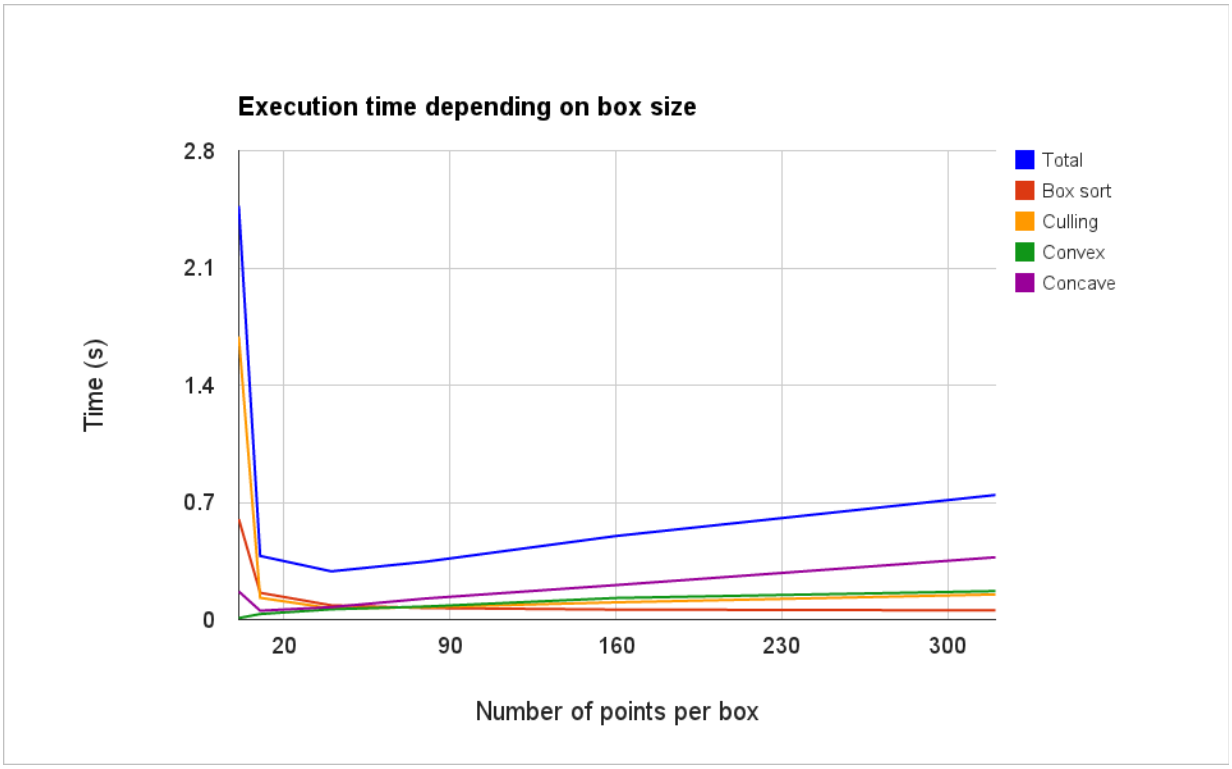

Figure 6.6: Execution time as a function of the number of points per box.

The space partitioning gives the best performance when there is about 40 points per box. We received similar results for all the other test cases.

6.6 Concave threshold modifier constant

The concave threshold modifier constant decides how concave the calculated hull will be. In *Figure 6.7* we can see how the constant effects the result. A have constant 2, B is 5 and C is 10.

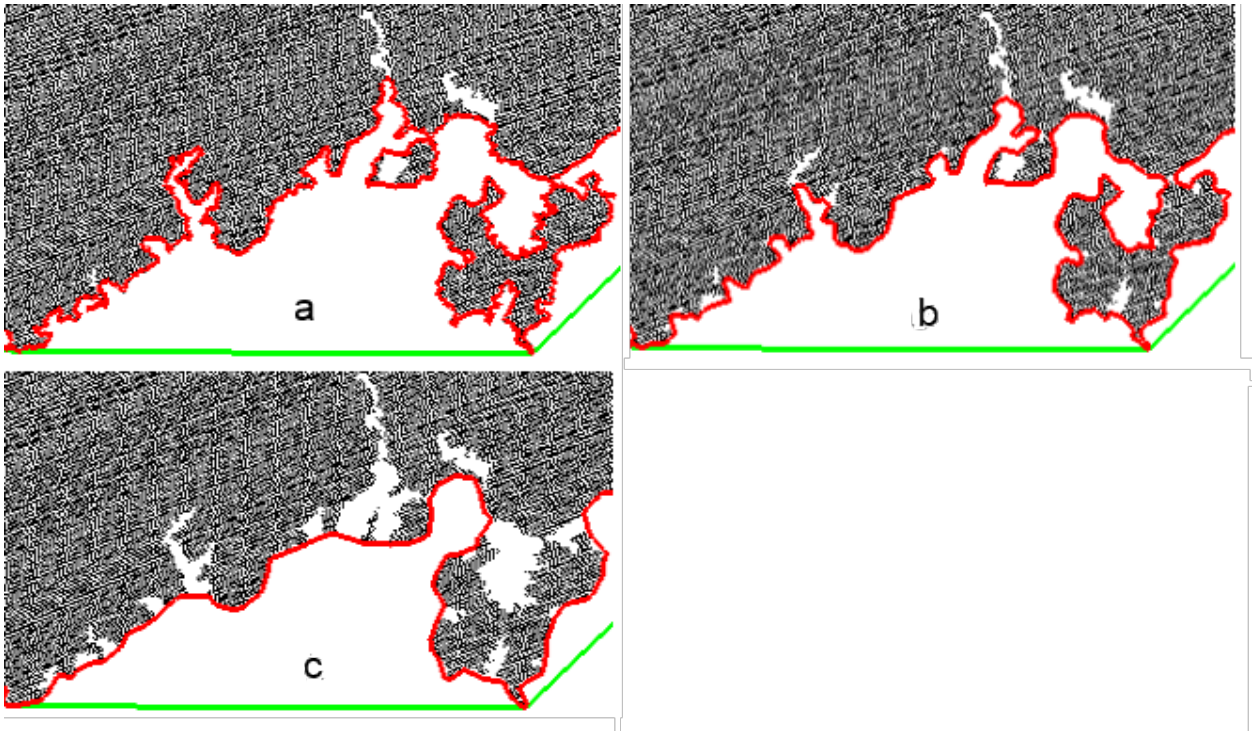

Figure 6.7: The concave and convex hull for a part of the "300k" data set. In (a) the concave threshold modifier constant is 2, in (b) it is 5 and in (c) it is 10.

As we can see, a higher constant makes the calculated hull less concave, a lower constant makes it more concave. From these results and from the other test cases we think that the algorithms gives good concave hulls when the constant is between 5 and 10. If the constant is over 10 the hull will not be concave enough, and if the constant is close to 2 or below, the calculated hull will risk being too concave. In our implemented prototype the constant is set to 8.

6.7 Parallel performance

We tested the parallel performance on some parts of the code. The computer used for the parallel tests were an Intel i5-450M computer that has 2 physical CPU cores and 4 threads with hyperthreading, running Windows 7.

First we tested the parallel performance of the concave phase of the algorithm and the space partitioning. We used the star test case with 10 000 000 points, since that geometry has four big concave parts, and ran the algorithm with 1, 2 and 4 threads. The results can be seen in *Figure 6.8* and as can be seen, the concave part is roughly two times faster when using two physical cores while only getting a minor speedup if also using the two additional virtual cores. The

space partitioning only get a minor speedup when using two physical cores and using the two virtual cores make it a bit slower. In both the concave part and the space partitioning, the reason we see a bigger speedup when using the two physical cores than when using the two virtual cores may be because it is limited of heavy memory access and the fact that the two virtual does not add any additional physical memory.

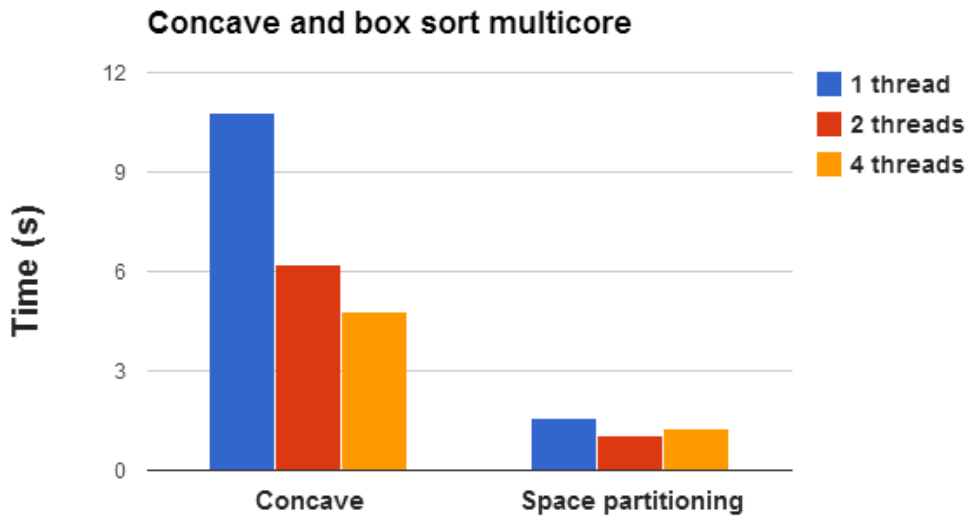

Figure 6.8: Execution time depending on number of threads, star test case with 10 000 000 points

The convex part was also parallelized. *Figure 6.9*, show the result when parallelizing the quicksort phase and the merge phase. The test was on the square test case with 1 000 000 points. As you can see we got a good speedup on quicksort when running on 2 and 4 threads. On the other hand, we did not see any speedup on the merge phase, and the merge accounts for a bigger part of the convex phase.

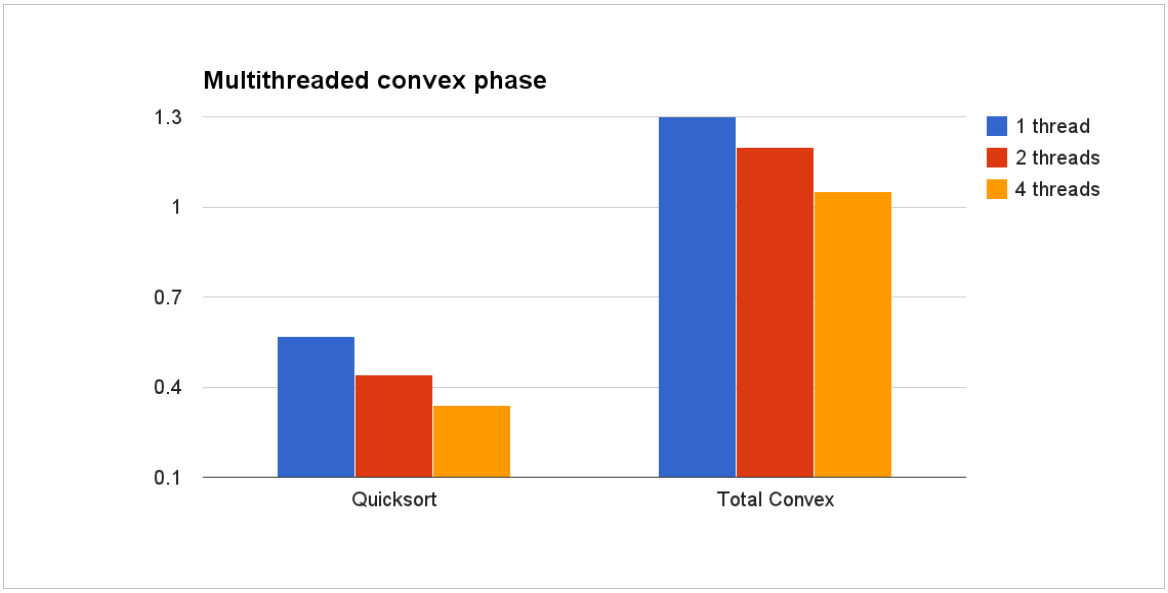

Figure 6.9: The execution time of quicksort and the entire convex phase on 1, 2 and 4 threads.

6.8 Performance

We tested the performance of the algorithm and how the performance scales with the number of points by running the algorithm on the "hourglass" test case with between 1 000 000 and 16 000 000 points. The total execution time was measured as well as the execution time of different parts of the code. In *Figure 6.10* these results are shown.

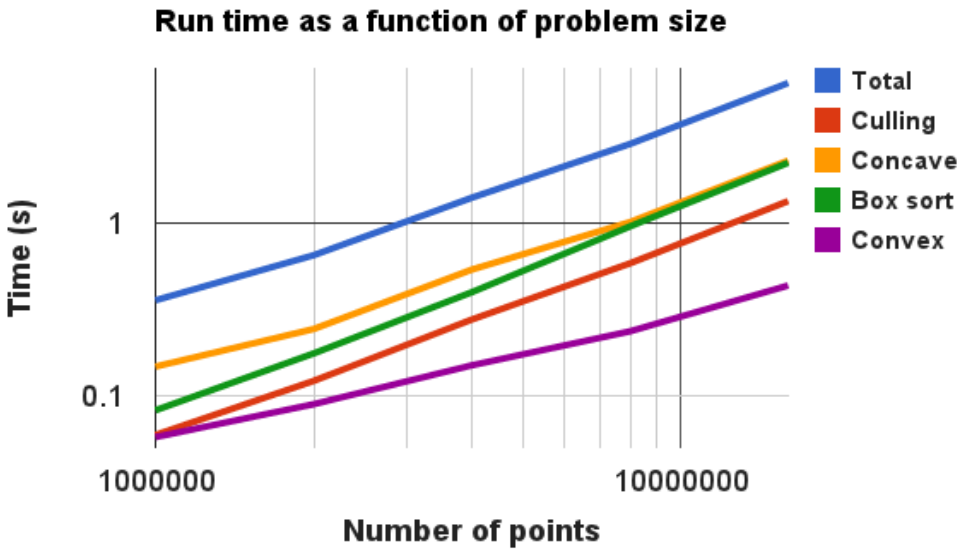

Figure 6.10: The execution time as a function of the number of points.

We can see that, even with 16 000 000 points in the input data set, the execution time is under 10s. We can also see that the concave phase and the box sort are the two parts of the algorithm that take the longest time. The scaling of the performance to the number of points is very good, at 1 000 000 points the total execution time is 0.36s - at 10 000 000 points it is just over 3s.

We know that the convex phase has a complexity of *nlog(n)* [4] and that the box sort has a complexity of *n*. The exact complexity of the concave phase can not easily be derived, but from these results we can reason that the complexity is higher than *n* and also higher than the convex phase, but not close to n^2. In that case it means that the concave phase is the limiting part for the overall complexity of the algorithm.

6.9 Robustness

To ensure that the algorithm produces a good concave hull we tested the algorithm on many different test cases with varying geometry. The algorithm produced good concave hulls for all the test cases in *Figure 6.1*, where we can see the calculated concave hull in red and the convex hull in green. In addition to these test cases, the algorithm can also handle datasets with varying density, as shown in *Figure 6.11*. Here the density of points is not constant, but the algorithm still manages to calculate a good concave hull.

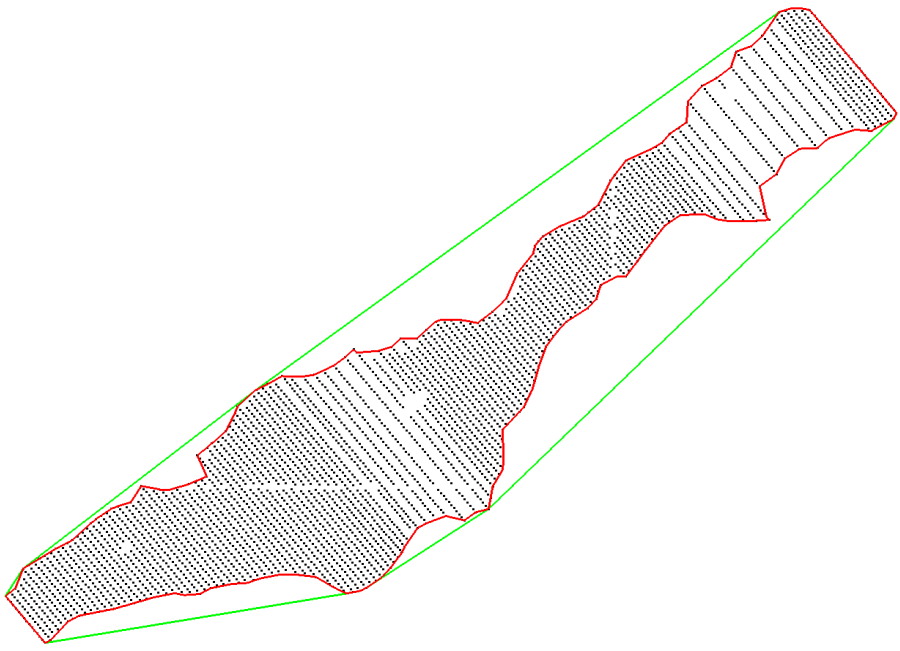

Figure 6.11: The concave (red) and convex (green) hull to a data set with varying point density.

To show that the algorithm can handle extreme cases we created a data set consisting of many different clusters of points. This can be seen in *Figure 6.12*. It is not very likely that you want to find one single concave hull to a set like this, but it shows that the algorithm can handle data sets that have a very odd geometry and still calculate good hulls.

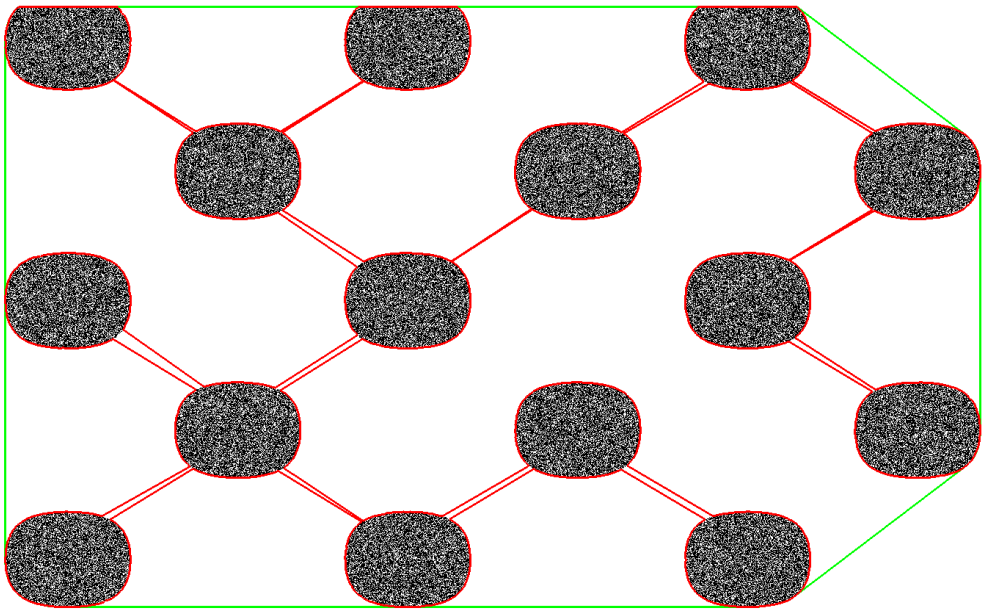

Figure 6.12: The concave (red) and convex (green) hull to a set with points in clusters.

6.10 Constants and parameters

In the Gift Opening algorithm some constants are needed. We attempted to have as many as possible be automatically calculated from the input set. Two constants were tweaked according to what gave the best results for all the test cases and best performance, see *section 6.5* and *section 6.6.* No input parameters are required by the user. From the results we determined that the constants defined in *Table 1* work well with all test cases we tried.

Table 1: Recommended values of parameters and values used in our implementation

Parameter	Interval	Our choice
Points per box	40 - 80	40
Concave Threshold Modifier	5 - 10	8

7. Discussion

7.1 Algorithm

One problem with the Gift Opening algorithm is that the time required to run the code will depend a lot on the geometry. When the number of edges of the concave hull increases then the time increases since all edges needs to be tested to see if the should be splitted into smaller parts. Longer edges take longer time compared to smaller edges since more points needs to be controlled to see if they should be added to the boundary between the edges endpoints. A bad scenario that can be seen in *Figure 7.1* is when there is a long straight line of points nearly parallel to an edge with one endpoint that is part of the line. When the algorithm is going to try to split the edge, it is then going to pick the point in the line next to the edges endpoint meaning one of the new edges are going to be almost as long as the old edge. This will be problematic when the edge is long and the line contains many points.

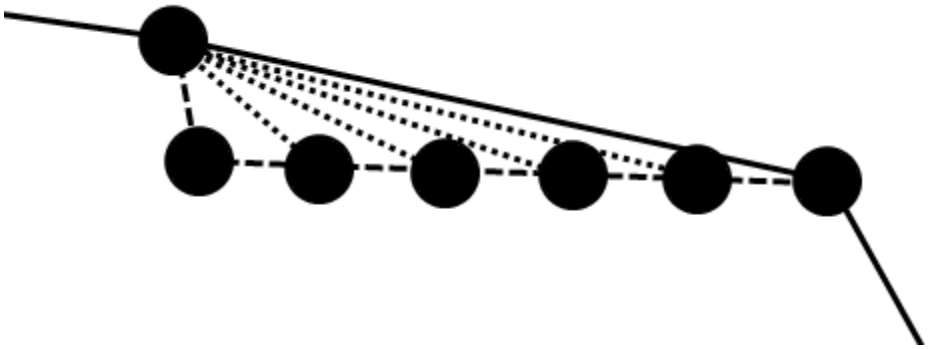

Figure 7.1: Problematic case when a line points are close to an edge. The solid line are the convex hull, the dashed line the final concave hull and the dotted lines are edges that are created during the concave part and splitted into smaller edges.

One advantage with the Gift Opening algorithm is that It is possible to interrupt the algorithm during the concave phase. This can be used to stop the algorithm after it has run for a certain amount of time. The result after interrupting the algorithm will still be a correct concave hull, but less concave and with longer edges than if the algorithm would have continued. This could be useful to avoid long runtimes due to the geometry of the problem. Another advantage is that the algorithm first calculates the convex hull, and then the concave hull. This means that when you calculate the concave hull with this algorithm you also get

the convex hull without any additional computation.

There is a tradeoff when deciding what boxes to search in when trying to find points in the concave part. By searching fewer boxes, that part of the code will run faster but there is a risk that it will miss a point that would have been better to add to the boundary. By searching more boxes the reverse happens. It is very important that there is no possibility for a point to end up outside of the boundary due to a box not being searched, see *Figure 4.3*.

7.2 GPU parallelization

During the project we had enough trouble to parallelize the algorithm using the CPU and decided to not try to extend it to GPU parallelization. The biggest improvements we had on the performance came from optimizations rather than parallelism so we decided to focus on that instead. But according to [8], quicksort performed well on a GPU and the merge phase of the convex phase was parallelizable to the same extent even that part may be able to be parallelized on the GPU.

The concave phase should be more troublesome since to get a good concave hull we want to remove the biggest edges before removing smaller edges, making parallelisation hard.

Using the Delaunay approach may still be good since there are some efficient Delaunay triangulations using the GPU, as described in [9]. The concave part will still have the same problems but the concave part of the Delaunay triangulation is faster than the concave part of the gift opening algorithm.

7.3 Three dimensions

We haven’t made any attempts to implement a solution in three dimensions but we see no reason for why it shouldn't be possible to implement it. There are known solutions to generate a concave hull in three dimensions, see [10].

Gift Opening will need the following modifications to work in three dimensions. Edges needs to be triangular areas containing three corner points. Another measurement than length needs to be used for edges eg. the area of the triangle or the total length of the sides of the triangle. Angles needs to be calculated in another way when trying to find the best point eg. measure the angle between the triangle to lines between the tested point to the triangles corners.

7.4 Improvements

In the concave phase, an edge is only removed if its length is long enough compared to the local closest distance between two points times a constant. This distance could be calculated using a local mean value between all closest points in neighbouring boxes or in other ways that may prove to give better results.

Currently the amount of boxes is determined by the axis which has the largest distance between it's maximum and minimum coordinates. Let say the points are divided unto a very thin rectangle, all boxes in the space partitioning would be on one long line. This would make the culling algorithm to fail, for example, and could pose other problems.

The space partitioning is an equidistant grid. There is a possibility that using boxes with different size would be able to improve the performance by for example not creating many boxes where there are only a few points and the opposite for areas with many points. This would make it more robust for non-uniform geometries.

In the implementation of the Divide and Conquer in the convex phase we did not see any speedup on the merge phase. This is likely due to our implementation, and not due to the algorithm itself. Therefore the implementation can be improved to achieve a better parallel speedup on that part. This would in turn lead to a significant speedup on the convex phase which would mean a small, but noticeable, speedup on the entire algorithm.

8. Conclusions

From the results we conclude that the Gift Opening algorithm is effective and fast. It can handle very big input data sets (order 10^7) and calculate the concave hull within a few seconds (<10). This is very good since one of the most important goals in this project was to implement a fast algorithm. The performance is also very consistent on different input set geometries. The algorithm is also very reliable. It can handle inputs sets with odd geometries and still give a good result. No matter how the input set it shaped the algorithm will never fail to give a result. It can also handle input sets where the density of points is varying. From all of this we draw the conclusion that the algorithm satisfies all the goals and constraints that were set in the beginning of the project.

In the project we have also seen that the optimizations made to the implementation of the algorithm have been very successful and a key factor to the good performance we have seen from the algorithm. From this we draw the conclusion that both making changes to the algorithm and making optimizations to the implementation of the algorithm is very important in order to get a good performance.

As a final conclusion we think that this has been a successful project. We have developed and implemented a prototype of an algorithm that satisfies all the project goals within the set timeframe for the project.

References

[1] Wikipedia
URL: http://en.wikipedia.org/wiki/Convex_hull 2013-01-16

[2] Matt Duckham, Lars Kulik, Mike Worboys, Antony Galton. 11 january 2008. Efficient generation of simple polygons for characterizing the shape of a set of points in the plane
URL: http://www.geosensor.net/papers/duckham08.PR.pdf 2013-01-16

[3] Wikipedia
URL: http://en.wikipedia.org/wiki/Gift_wrapping_algorithm 2013-01-16

[4] D. T. Lee, and B. J. Schachter. February 1980. Two Algorithms for Constructing a Delaunay Triangulation
URL:
http://www.personal.psu.edu/cxc11/AERSP560/DELAUNEY/13_Two_algorithms_Delaunay.pdf

[5] Faniry Harijaona Razafindrazaka. 22 May 2009. Delaunay Triangulation Algorithm and Application to Terrain Generation
URL: http://users.aims.ac.za/~faniry/documents/faniry.pdf

[6] Sang-Wook Yang, Young Choi, Chang-Kyo Jung. June 2011. A divide-and-conquer Delaunay triangulation algorithm with a vertex array and flip operations in two-dimensional space

[7] MinGW
URL: http://www.mingw.org/ 2013-01-16

[8] Daniel Cederman, Philippas Tsigas. January 2008. A Practical Quicksort Algorithm for Graphics Processors
URL: http://www.cse.chalmers.se/research/group/dcs/TechReports/gpuqsort.pdf

[9] ASHWIN NANJAPPA. 2012. Delaunay triangulation in R^3 on the GPU
URL: http://ash.daariga.com/papers/gdel3d-thesis.pdf

[10] Jin-Seo Park, Se-Jong Oh. 22 February 2011. A New Concave Hull Algorithm and Concaveness Measure for n-dimensional Datasets
URL: http://www.iis.sinica.edu.tw/page/jise/2012/201205_10.pdf