Computing Set Operations on Simple Polygons Using Binary Space Partition Trees

Simon Nordved Madsen, 20101916

Rasmus Hallenberg-Larsen, 20102247

Master's Thesis, Computer Science

Aarhus University
July 2015

Advisor: Gerth Stølting Brodal

Co-Advisor: Peyman Afshani

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Part I

PRELIMINARIES

ABSTRACT

In this thesis we show how it can be worthwhile to save the Boundary Representation of polygons even if a Binary Space Partitioning tree has been built. The reason being that one algorithm (INC_SET_OP [5]), which rebuilds the relevant parts of the scene, can in some cases run in $O(n)$ time with a smaller constant than another algorithm (MERGE [4]) that merges Binary Space Partitioning trees. Not only can it run faster but in some cases it builds smaller trees than the merging algorithm, which has subroutines minimizing the resulting tree. INC_SET_OP may build smaller trees because it cuts away irrelevant line segments, as it still operates on Boundary Representations. This ability does however also mean that it has the risk of having to build an entire structure from scratch, which has the expected running time of $O(n^2 \log n)$ for a tree of size $O(n \log n)$.

ACKNOWLEDGEMENTS

We would like to give a big *Thank You* to our advisor Peyman Afshani for keeping us on track, helping us through any problems we've had and providing us relevant feedback when needed. We would like to also thank our friends, families and girlfriends for supporting us through writing this thesis. The office next door, Lukas Walther, Mathies Christensen and Thomas Sandholt, deserves a *Thank You* for relevant discussions and comments concerning our progress. Especially Thomas Sandholt for giving valuable and constructive criticism and feedback.

Thank you.

Rasmus & Simon
Aarhus, July 2015

CONTENTS

Part II

CONTENT

1 INTRODUCTION

Having a binary partitioning of space is important in many computational geometry problems such as collision detection, visibility etc. Binary Space Partitioning allows us to use binary search in space.

In this thesis we focus on polygons partitioning space and enabling such polygons to become dynamic by combined them with other polygons using binary operators, such as intersection and union. We show how two different algorithms for resolving such operations can give different results and even differ in asymptotic running time on certain scenes. One algorithm [4] uses only Binary Space Partitioning trees (BSP-trees) as input making the raw data, in form of a Boundary Representation (BREP), obsolete. The other algorithm [5] still uses a BREP of a polygon as one of its inputs. We argue that saving the BREP in some cases can be beneficial, even if a BSP-tree has been built. The reason for this being that the algorithm that uses the BREP is able to outperform the merging algorithm not only in speed but sometimes also in size of the resulting tree.

We present related work in Chapter 2. In Chapter 3 we go through the theory for building BSP-trees as presented in [5]. Next we will compare a BSP-tree, built from a polygon, with a regular set. Polygons can be represented as regular sets, which means that the operations applied to regular sets can also be applied to polygons and this is the basic idea behind merging BSP-trees. We move on to describe, in further detail, the behavior and mechanics of the two algorithms as well as give an analysis on the running time. We will also in this section describe how the structure of the scene can have an impact on the running time for both algorithms.

In Chapter 4 we describe the implementation details for our algorithms. We focus on details that are not described in the theory as well as some optimisations for MERGE that are mentioned in [4]. In this chapter we will also describe any relevant environment details that we used, such as compiler optimisations as well as the usage of a machine ϵ in order to deal with numerical instability.

In Chapter 5 we compare both algorithm through experiments and discuss the difference in their output. We explain how test data is generated and analyse the necessity of the subroutines of MERGE. After this we use special cases to illustrate how the algorithms differ. Finally we will use larger test data to show how the algorithms behave on different representations of data as well as inputs demonstrating the worst case running time of the algorithms.

In the last chapter we present our conclusion and talk about ideas for future work.

Appendices A and B are found at the end of the thesis and contain pseudo code for the algorithms used and additional experimental results.

2

RELATED WORK

Thibault and Naylor in 1987 developed a method for combining polyhedra using binary operations [5]. This was done by having a Binary Space Partitioning tree (BSP-tree) in which new polyhedra could be inserted. A polyhedron would be represented as a Boundary Representation (BREP). A Binary Space Partitioning is the partitioning of space using lines. Each of these partitioning lines are nodes in the BSP-tree. A BREP is a way to represent shapes by defining their boundary with lines that have a direction. A more detailed explanation of these can be found in 3.1 and 3.2.2. The algorithm INC_SET_OP, that they developed, took as input a BSP-tree, a Binary Operation and a BREP and output a BSP-tree. Depending on the operation, the BREP is split by the lines already saved in the BSP-tree. As the algorithm splits the BREP the now partitioned lines are recursively split by the children of the root of the BSP-tree. When the recursion reaches a leaf it will, depending on the operation, either build a new BSP-tree from any line segments passed down or simply return the leaf.

In [5] they also describe a way of building a BSP-tree from a BREP, BUILD_BSP. A line segment is picked and extended to a line, which partitions the remaining line segments. If a line segment is split, then a new segment is partitioned into both sides. The splitting line represents a node in the BSP-tree and its children will then be created from the recursion of the same procedure in both the left and right side of the partitioning. Finally when no lines exists in a side, an IN- or OUT-leaf is made, depending on the orientation of the line segment in the original BREP. A more detailed explanation of the building of a BSP-tree can be seen in section 3.1.1.

In 2008 Lysenko et al. created an algorithm for doing binary operations on two BSP-trees while also potentially making the resulting BSP-tree smaller than that of contemporary merging algorithms[4]. Their algorithm MERGE takes as input two BSP-trees and output a BSP-tree. Their algorithm recursively traverses the nodes of one BSP-tree and inserts in the leafs of this BSP-tree. However, the insertion is done by recursing in the other BSP-tree and then applying the binary operation on the two leafs. It uses linear programming to refrain from doing binary operations on regions that are disjoint. The linear programming is a way to avoid the need of performing difficult tree partitioning and polygon cutting used by other current merging methods. Finally they use a subroutine, COLLAPSE, that replaces mutually equal subtrees with pointers to already existing subtrees.

A more detailed explanation and analysis of INC_SET_OP and MERGE can be found in sections 3.2 and 3.3 respectively.

3 THEORY

3.1 BINARY SPACE PARTITIONING OF POLYTYPES

We want to represent polytypes with binary space partitioning trees (BSP-tree). BSP-trees allows spatial information about a scene to be accessed quickly. This is useful in rendering, such as a front-to-back rendering from a viewpoint. As with all structures, a BSP-tree takes time to build and we want this building time to enable faster operations on the structures than the data itself allows. Specifically, we want the BSP-tree to enable us in using binary operators between polytypes. The motivation being that it will enable building more complex polygons by combining smaller, simpler polygons with binary operators or that animation of a scene is possible by inserting and removing polygons. This theory section is restrained to 2 dimensional examples and explanations since our implemented algorithms also operates in the 2 dimensional space. In section 3.1.1 we explain how a polygon partitions space and how to build a BSP-tree from these partitions. We will then go on to explain how a BSP-tree can be merged with a boundary representation (BREP) of a polygon, in section 3.2. And finally, given two BSP-trees we show how to merge these trees given a binary operation, such as UNION or INTERSECTION.

3.1.1 *Binary Space Partitioning Trees*

A BSP-tree is a recursive partitioning of a d-dimensional space. As earlier mentioned we will only discuss the case of 2 dimensions. A common strategy for building a BSP-tree is to use the technique known as *auto-partitioning*: The input line segments are extended to lines. These lines are called the partitioning lines. A line segment from the input that intersects a partitioning line is split in two. This process is recursively applied in each half of the space. Each partitioning line has an exterior and interior direction. The normal of a line segment points towards the exterior. Having recursively partitioned with each line segment, each node v in the BSP-tree represents a partitioning of the space $R(v)$ as seen in Figure 1.

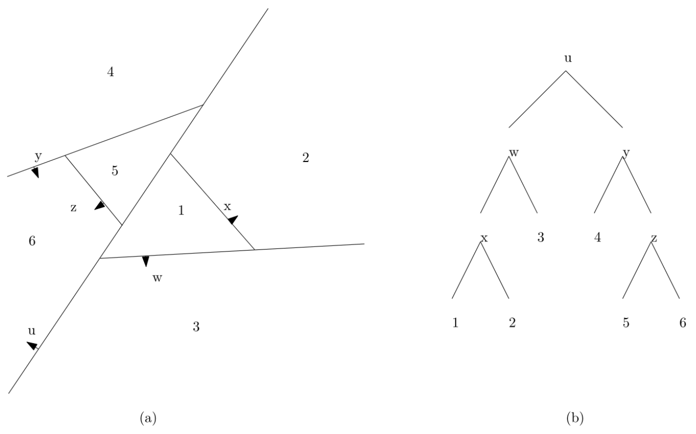


Figure 1.: In (a) we see the 2D space partitioned. (b) Shows the BSP-tree created from the partition shown in (a).

Given a finished tree each leaf will correspond to an unpartitioned subspace. In figure 1, these leaves are labeled $1-6$ in figure 1(a) and each represent the corresponding cell in figure 1(b).

For any line

$$L = (x_1, x_2)|a_1x_1 + a_2x_d + a_3 = 0$$

The subspace will be partitioned into two halves;

$$L^+ = (x_1, x_2)|a_1x_1 + a_2x_2 + a_3 > 0$$

and

$$L^- = (x_1, x_2)|a_1x_1 + a_2x_2 + a_3 < 0$$

There exists now for each v a subspace $R(v)$ and three sets in this subspace. Consider the set $R(v) \cap L_v^+$, which is represented by the right subtree of v. In 2 dimensions this set is the set of points on the normals side of the splitting line. On the other side of the line is the set $R(v) \cap L_v^-$ represented by the left subtree of v. Finally we have $R(v) \cap L_v$ which is the line for v and other lines laying on the line for v. Given any leaf we can traverse up through the tree and the lines on the route from the leaf to the root of the tree will define all the partitioning lines that enclose this subspace. Not all of these subspaces will be entirely enclosed, as an example see all the exterior regions in figure 1. It is possible for an exterior region to not be enclosed by partitioning lines, but interior regions will always be enclosed by a solid and continuous border. To better illustrate this we will describe these subspaces as *regular sets* in section 3.1.2.

Running time

The naive approach selects line segment from a list as they are ordered. A bad permutation of these line segments can result in a partitioning with a lot of splits, thus leading to a large tree. To avoid such a bad permutation we randomize the input list before building the BSP-tree. The time it takes to build a BSP-tree depends on the size of the resulting tree which in turn depends on the amount of cuts made while partitioning. We start by analysing the amount of fragments generated by one line, leading to the total number of fragments. From this we analyse the expected time and space requirements for the algorithm of building a BSP-tree.

Let s_i be a segment from the input. If another segment s_j is to be cut by the line $l(s_i)$, extended from s_i, then some conditions must hold. Firstly s_i must be selected before s_j. Secondly $l(s_i)$ has to actually intersect s_j. Thirdly no other line segment s_k that blocks the view of s_j from s_i must have been selected between s_i and s_j. With this we define a distance function. The distance function $dist_{s_i}(s_j)$ is the number of segments intersected by $l(s_i)$ between s_i and the line segment s_j.

$$dist_{s_i}(s_j) = \begin{cases} \text{number of segments intersected} & \\ \text{by } l(s_i) \text{ in between } s_i \text{ and } s_j & \text{if } l(s_i) \text{ intersects } s_j \\ +\infty & \text{Otherwise} \end{cases}$$

The probability that $l(s_i)$ intersects s_j is dependent on the distance function $dist_{s_i}(s_j)$, whics smaller when the distance increases. As we already mentioned, s_i must come before s_j and s_j must be selected before any of the segments that blocks s_j from s_i. We get the following inequality:

$$P[l(s_i) \ cuts \ s_j] \leq \frac{1}{dist_{s_i}(s_j) + 2}$$

, where $+2$ are the lines s_i and s_j themselves. This is an inequality because other extended lines can block $l(s_i)$ from intersecting with s_j. The expected total number of cuts generated from selecting s_i as splitting line, is the sum of this inequality for all other line segments:

$$E[\# \text{ cuts made by } s_i] \leq \sum_{j \neq i} \frac{1}{dist_{s_i}(s_j) + 2}.$$

Because there are at most two segments with the same distance and the distance is increasing, then we get the following:

$$\begin{aligned} E[\# \text{ cuts made by } s_i] &\leq \sum_{j \neq i} \frac{1}{dist_{s_i}(s_j) + 2} \\ &\leq 2 \sum_{k=0}^{n-2} \frac{1}{k+2} \\ &\leq 2 \ln n. \end{aligned}$$

Having n segments, the total number of cuts generated by all segments is at most $2n \ln n$. The total number of fragments must be our starting segments added to these cuts: $n + 2n \ln n$, giving the bound $O(n \log n)$.

This bound on the expected number of fragments means that we can determine an expected running time, as the running time was dependent on the size of the output tree. The running time is linear in the number of starting segments n and it will get smaller at each recursive call, since we partition the space. The number of recursive calls depends on how many segments existing at each call, which again depends on the number of cuts/generated segments. Meaning that the expected running time is:

$$O(n \cdot \#generated\_segments)$$

Since the number of expected cuts generated by all n lines is $O(n \log n)$, the expected number of recursive calls is also bounded by $O(n \log n)$. This means that the expected running time of building a BSP-tree of size $O(n \log n)$ is $O(n^2 \log n)$ [3]. Although given $\frac{n}{2}$ vertical and horizontal lines, the building time will in worst case be at least $\Omega(n^2)$.

3.1.2 *Regular Sets*

The d-dimensional subspace can be described as a metric space. We define a metric space like in [1] as (W, d) where W is the set and d is a distance function for two points in W;

$$d : W\ x\ W \to R$$

Each set has an interior, an exterior and a boundary. Before looking further at the definitions of these, we define the neighborhood of a point in a metric set as in [1].

Definition 3.1.1. *In a metric space (W,d) a set X is a neighborhood of a point x if there exists an open ball with centre x and radius* $r > 0$*, such that*

$$B_r(x) = \{w \in W | d(w, x) < r\}$$

is contained in X.

Simply put; if in 2D a circle with any radius larger than 0 is drawn around x, then this circle must be entirely inside X for X to the a neighborhood of X.

Returning to the subspace as a metric set and the description of this set as interior, exterior and boundary. The exterior of a set can be defined as the complement of the set. To Distinguish interior points from boundary points we define the interior of X, $i(X)$, as the union of all interior points;

Definition 3.1.2. *A point x of W is an interior point of a subset X of W if X is a neighborhood of x.*

Not all points contained in the set will be interior points of the set, but all interior points will by definition be in the set. All points in the set that do not fall under the interior point definition can be defined as a boundary point. The boundary of X, $b(X)$, is the union of all boundary points defined as;

Definition 3.1.3. *A point x of W is a boundary point of a subset X of W if each neighborhood of x intersects both X and* $c(X)$*, where* $c(X)$ *is the complement of X.*

Figure 2 is an illustration of two such points.

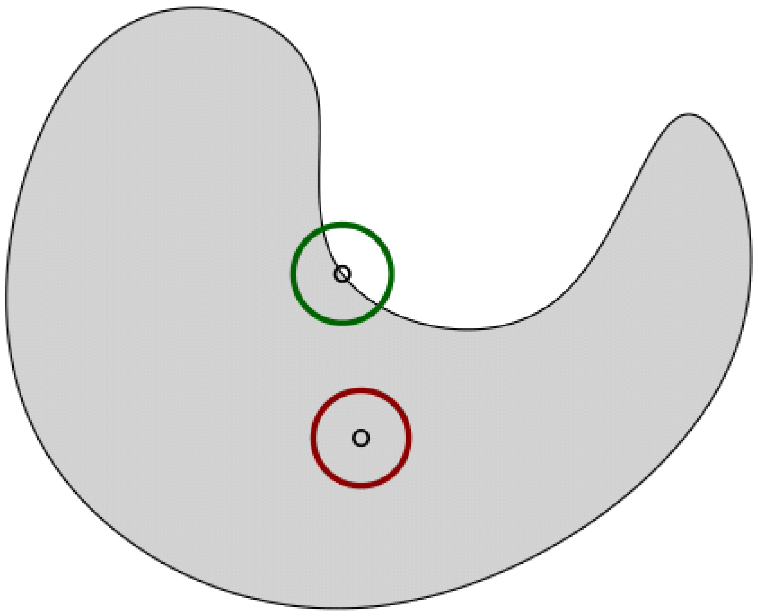


Figure 2.: A regular set with boundary point (green) and interior (red)

Limit points also exists for the set. Limit points can be both in and not in the set, but all limit points of a set will be in the set if the set is closed [1]. And since we are using the set theory to describe polytypes we only use closed sets. For closed sets limit points will be equal to boundary points.

Definition 3.1.4. *A point x is a limit point of a subset X of a metric space (W, d) if each neighborhood of x contains at least a point of X different from x*

Another restriction on a metric set is a regular set. A subset X of a metric space (W, d), is a regular set if

$$X = k(X)$$

Where k is the closure of a set. The closure of a set is the union of the boundary and the interior. The closure of a set is a regularization, where the set is made similar to a solid model with a well defined border between the set and its complement, the border being included in the set.

A polyhedron is a regular set since the interior and boundary is exactly the polyhedron. Were we to build a BSP-tree for any polytype the boundary of the polyhedron would be stored in the nodes of the tree and each leaf would represent a part of the exterior or a part of the interior of the polyhedron. By convention, all normals points away from the interior and with this we can label each leaf as IN or OUT. The union of all IN-cells and the lines stored at the nodes on the route from such a leaf to the root would be equal to the regular set of the polyhedron. Since we can now say that a polyhedron is equal to a regular set, we are able to use the set operations of a regular set on a polygon.

3.1.3 *Regular Operations*

We have not defined the properties of regular sets as a non variant for a BSP-tree that has been built from a polygon. We would also like them to hold for any BSP-tree that is a product of a binary operation between two polygons. For this reason only the regularized set operations are possible, the intersection of two regular sets, the union of two regular sets, subtracting two regular sets and the complement of a regular set. Regular sets are closed under these operations. These operations are denoted by

$$\cap^*, \cup^*, -^*, \text{and } \sim^*$$

In order to evaluate these operations on regular sets we use expression simplification in a geometric setting. But firstly consider the complement $\sim^* S$ of a regular set S. The complement is easily formed by complementing all cell values and changing the direction of the normal for each boundary. Thus making the interior the exterior and vice versa. Complementing a BSP-tree is done by turning all IN-leaves into OUT-leaves and vice versa, and swapping all left and right children.

As for the other operations, we want to utilise expression simplification and to obtain that at least one operand is homogeneous. Given $S_1 op S_2$ we simplify by partitioning the space such that each region R_i is either in the exterior or interior of both sets. Having a BSP-tree we are given such a partitioning and we want to recursively traverse the tree, evaluating only when one operand is a leaf. As an example; consider the operation $S \cup^* v$, where v is an IN-leaf. This evaluates to IN, since there is nothing in this cell not already in the interior. A more detailed overview is found in table 1.

3.2 SET OPERATIONS ON BSP-TREES AND BREPS

Now given a representation of a polygon as a BSP-tree and a way to use binary operations on BSP-trees, we can begin dis-

op	Left Operand	Right Operand	Result
$\cup^*$	S	IN	IN
	S	OUT	S
	IN	S	IN
	OUT	S	S
$\cap^*$	S	IN	S
	S	OUT	OUT
	IN	S	S
	OUT	S	OUT
$-^*$	S	IN	OUT
	S	OUT	S
	IN	S	$\sim^*$ S
	OUT	S	OUT

Table 1.: Overview of the results from applying binary set operators

cussing how to combine polygons. We look at an algorithm that incrementally resolves a binary operation between the current BSP-tree and a new polygon - represented as a BREP (detailed in 3.2.2). Given a BSP-tree we should be able to combine it with a BREP as per the table in 3.1.3. An obvious advantage of this is that when constructing a scene one does not have to construct a complex BREP in order to represent a complex polygon, as long as the correct building blocks exists. This also gives the ability to animate a scene. *Constructive Solid Geometry* is an example where you use simple building blocks to represent more complex scenes and it is further discussed in 3.2.1.

3.2.1 *Constructive Solid Geometry (CSG)*

In [2, p. 557] a Constructive Solid Geometry (CSG) is described as a representation, where simple primitives are combined by means of Boolean set operations that are included directly in the representation.

The primitives are stored at the leaves and the operators are stored at the nodes. A combined object is then represented at the root of a tree. Boolean operations, rotations and translations can all be stored at the nodes. The order of left and right children is taken into account since Boolean operators normally are not commutative. An example of a CSG can be seen in figure 3, which also shows that a CSG is not unique since the same object can be modelled in several ways.

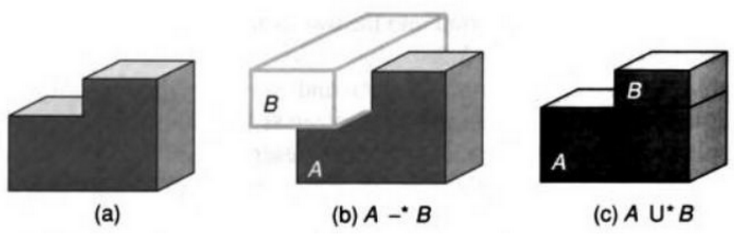


Figure 3.: (a) shows a CSG, where (b) and (c) shows two different ways to produce it. Figure taken from [2].

3.2.2 *Boundary Representation (BREP)*

A Boundary Representation is a method used to represent shapes using the boundary of the shape. Any solid polygon can be represented by connecting line segments creating a split between the interior and the exterior of the solid. Each of these segments will have a direction defined from a starting point and an ending point. An example of a BREP can be seen in figure 4, where start points are red dots and ending points are arrow heads. It is important that the starting point of line segment i is the exact ending point of line segment $i-1$. The final line segment must end where the first line segment began, creating an unbroken and solid path of straight lines.

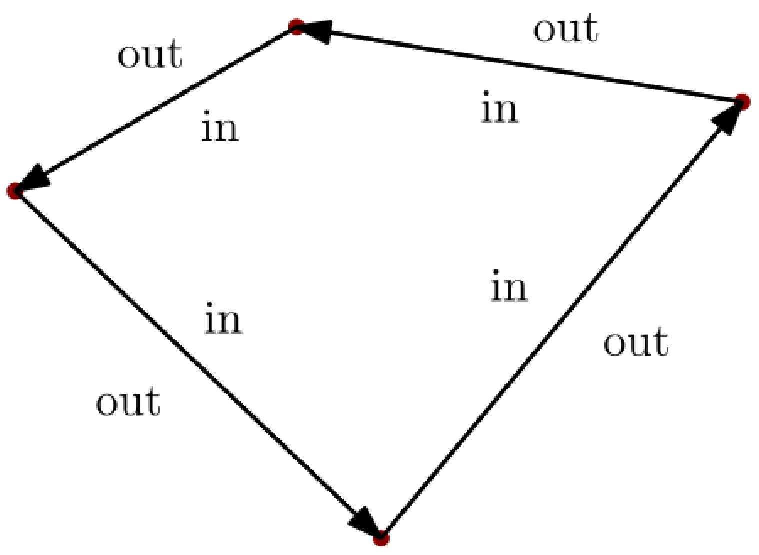


Figure 4.: BREP of a trapezium.

This is in coherence with the idea of representing polygons as regular sets. If each element has a direction is it possible to say that the interior is opposite of the normals direction and thus the definitions are equal.

3.2.3 *Binary Set Operators*

As we saw in 3.1.3, we work with three binary set operators:

1. UNION: $A \cup^* B$
2. INTERSECTION: $A \cap^* B$
3. SUBTRACTION: $A -^* B$

, here * refers to the regular operator.

Instead of having three different operators, we take advantage of the fact that the SUBTRACTION operator can be induced from the INTERSECTION and COMPLEMENT operators:

$$A -^* B \Leftrightarrow A \cap^* (\sim^* B)$$

This allows us to only have two binary operators and to have only commutative operators. We have to do the translation from using the SUBTRACTION operator to using the INTERSECTION operator, before we for instance swap operands (more on this in section 3.3.1). In 3.2.4, if our operator is SUBTRACTION, we COMPLEMENT our BREP and change the operator to INTERSECTION as the first step. SUBTRACTION is handled in a similar way by MERGE described in more details in 3.1.3.

3.2.4 *The Incremental Set Operation Algorithm*

Given a BSP-tree and a BREP we want to be able to include this BREP in the same BSP-tree, representing the resulting polygon. This is, in INC_SET_OP, done by recursively traversing down the BSP-tree. At each recursion we check if the current node is a leaf or a node. If it is a leaf we use the table in 3.1.3, e.g. if we are at an IN-leaf and the operation is UNION, then we simply return an IN-leaf. However, if the result is not equal to a leaf, then we must build a new BSP-tree of the line segments of the BREP that are still relevant in this leaf. This is illustrated in figure 5

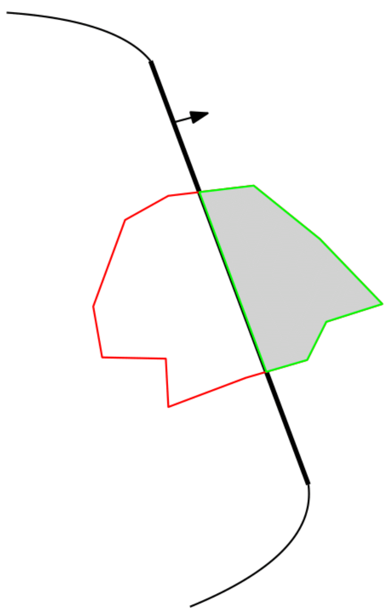


Figure 5.: Only the darker region is to replace the old OUT-leaf

In order for INC_SET_OP to know which lines to include in the iterations, it will need to partition them with the current partitioning line. If the recursion reaches a node, it partitions the lines of the BREP with the partitioning line saved on the node. After such a partitioning INC_SET_OP now goes through each side of the partition recursively. If no line segment from the BREP exists on one side of the partition, then this partitioning line must lie in the interior or the exterior of the BREP. This is essential to the algorithm. If a set line segments from the BREP are exclusively outside of interior of the BSP-tree, then they can be ignored. A subroutine for determining this, the *In/Out Test*, essentially tells INC_SET_OP to either ignore this part, because the BREP doesn't add anything new in this region, or replace with a new IN- or OUT-leaf because the BREP actually contains this area.

In/Out Test

The In/Out test is used to determine if an empty area represents an IN-region or an OUT-region of the BREP. Because we might not have any line segments lying coincident with our splitting line L, we cannot just use the normals orientation of those line segments. Instead, we have to look at the other side and do some calculations in order to determine whether an area is the interior or the exterior of the BREP. Without loss of generality assume that our left region is empty.

We start by finding the point b from our BREP in the non-empty, that is closest to our splitting line H_v. We then pick a point p that lies on our line represented by our node v and our splitting line H_v. We now have two cases as illustrated in Figure 6.

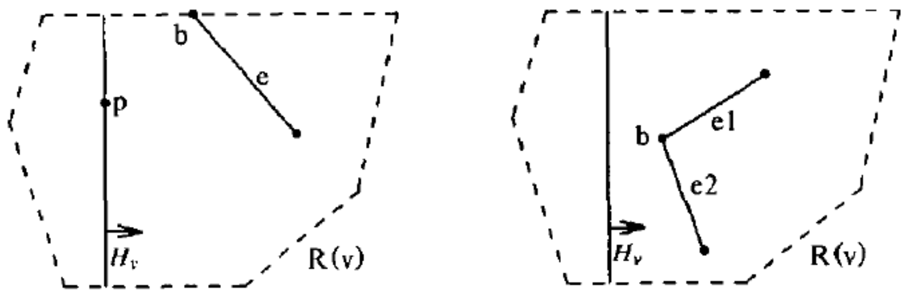


Figure 6.: (left)Closest point b is on the boundary of $R(v)$. (right) Closest point b is in the interior of $R(v)$. Figure taken from [5].

If our point b lies on the boundary of $R(v)$ an edge e to another point in the interior of $R(v)$ can be found. If we expand the edge e to a line H_e, we can now determine if our empty left region is representing an IN- or an OUT-region by comparing with a point p on H_v in Figure 6. If the point p lies in the exterior of H_e it means that the empty left region is an OUT-region as e defines a left most edge of its polygon. Otherwise the empty left side must be the interior of the polygon as e defines a right most edge.

If our point b instead lies in the interior of $R(v)$, b will now be connected to two other points by edges $e1$ and $e2$. Again we expand these edges to lines H_{e1} and H_{e2}. If $e1 \subset H_{e2_{left}}$ and $e2 \subset H_{e1_{left}}$ we have local convexity an the empty left region is therefore an OUT-region. This situation can be seen in figure 7(right). If the line segments are instead in each others right side, then local concavity exists and the empty left side must be inside the polygon. This is illustrated in Figure 7(left).

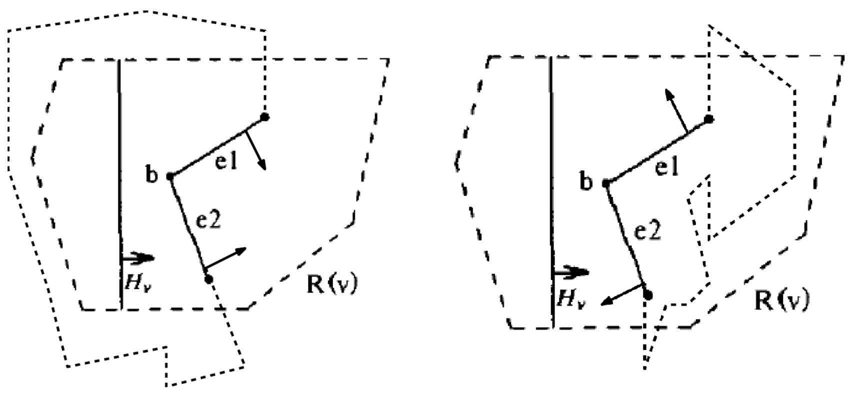


Figure 7.: (left)Left region is interior. (right) left region is exterior

Running Time Analysis

Analysing the running time of INC_SET_OP is not easy as it depends highly on how the lines of the two polygons intersect in the scene. One possibility is that the entire BREP lies within one leaf of the BSP-tree. This means that INC_SET_OP only has to call BUILD_BSP *once*, but this one call will be of the entire BREP. The running time of this BUILD_BSP call will depend on the structure of the BREP. Furthermore, INC_SET_OP in this case has to check for partitions h times, where h is the length of the route from root to the containing leaf. Another possibility is that the partitioning lines partition the BREP in such a way that each leaf only contains a constant number of lines. This means that each call to BUILD_BSP is made with a constant number of line segments. A single BUILD_BSP run with constant line segments can be made in constant time. This means that all the parts of the tree are built in time linear in the number of leaves. For a worst case scenario; consider a large convex polygon and n vertical lines and n horizontal lines inside this polygon. These lines can be connected in a way to form a polygon but for the sake of the argument, consider only the vertical and horizontal lines. If the lines are alternating in direction(parallel, anti-parallel), this will create $\frac{n}{2}$ vertical and horizontal bars. We let the operation used be INTERSECTION. All of these will now have to be built in the single IN-leaf of the large convex polygon. These bars will create $\frac{n}{2}^2$ squares, which will have to be leaves in the resulting tree. Meaning that in the worst case scenario the running time is at least $\Omega(n^2)$.

INC_SET_OP can be split into two phases; it must partition the BREP with the partitioning lines in the BSP-tree and it must build the remaining line segments (if there are any) in the leaves. A BSP-tree with size $2V$ has $V + 1$ leaves and V nodes. The algorithm will at most have V partitioning operations and $V + 1$ building operations. Note that depending on the operation used when merging, the building part of the algorithm takes place in only IN- or OUT-leaves.

When the line $l(v)$ for node v partitions B line segments of the BREP then the number of lines passed on to the children v_r and v_l must be at most B. The time spent on partitioning the BREP into the leaves will be worst case $O(V \cdot B)$. Combined with the above, this means that if INC_SET_OP has the worst case partitioning and the polygon that is partitioned is the grid described above, then the total worst case running time will be $O(n^3)$

Given a partitioning of the BREP by all the lines of the BSP-tree, then at most $V + 1$ builds must happen in the leaves. The time that such a build can take is dependent on the partitioning. We know from the running time section of 3.1.1 that a tree with size $O(n \log n)$ can be built in expected $O(n^2 \log n)$ time. Here $O(n \log n)$ is the expected number of fragments created by n line segments. The time it takes to build the remaining fragments depends not only on the number of fragments, but also on their structure. It could be the case that each build is done in constant time or it could be in the expected bound for building n lines, $O(n^2 \log n)$.

All in all the time spent in INC_SET_OP is entirely dependent on the structure of the scene and not so much on the number of lines. There is a possibility that this algorithm will run in linear time, which we will see many times in the experiments. The following case illustrates what can make it run so fast. At the root node we partition the $O(B)$ lines in such a way that $O(B)$ line segments are put in the right side and $O(1)$ are put in the left side. If the entire tree exists in the left side and the right side just is an OUT-leaf (and we have intersection meaning that these $O(B)$ line segments are discarded), then the rest of the partitioning will happen on $O(1)$ line segments. This case is visualized in Figure 8.

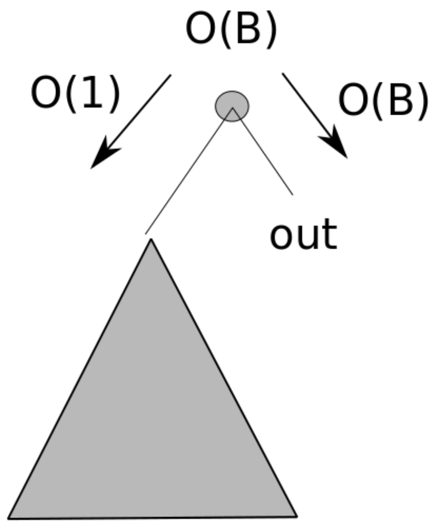


Figure 8.: A case that makes INC_SET_OP run in linear time.

This leads to both $O(V)$ time for all the partitioning, but also $O(V)$ time for all the build calls, leading to a total running time of $O(V)$ time.

3.3 MERGING OF BSP-TREES

Instead of using a BREP to represent a new polygon, then a BSP-tree of the polygon could be used. Removing the BREP entirely from the merging process allows an algorithm to ignore the In/Out test as this has already been resolved by the building of the second BSP-tree. MERGE does this by using linear programming to determine if it is to continue the recursion, or if the region is in fact infeasible because of the currently selected splitting lines.

3.3.1 *The Merging Algorithm*

MERGE, [4]), like the one described in section 3.2.4, recursively traverses one tree until it reaches a leaf. During this traversal it maintains the two current trees, the operation and a stack of partitioning lines. In each recursion some early termination checks are done.
The first check is to determine, using linear programming, if the current region is infeasible. If so, then the recursion stops and an empty node is returned as the result. The parent node is then replaced by the child that is not an empty node. Note that if one child is empty then the other can not be. This is

because the partitioning lines are added to the linear program with directions and in the two children, the directions are opposite each other. Such two lines define the entirety of $R(v)$ and both sub regions cannot be empty since we are partitioning a non-empty region. The theory behind linear programming and how it is relevant to this problem is described in further detail in section 3.3.3.
The second early termination check is to if the leaf in the first tree is irrelevant to the operation. IN-leaves with UNION as the operator needs no recursion on the other tree. This happens in two cases, where the trees are A and B.

1. The operator is UNION and A is an IN-leaf.

2. The operator is INTERSECTION and A is an OUT-leaf.

In the first case, where we have UNION, we always return an IN-leaf. This is because, no matter what our B is, we will always end up with an IN-leaf. The other case is where we have INTERSECTION as our operator. If A ever becomes an OUT-leaf, we simply return an OUT-leaf. This is because intersection between an OUT-leaf and anything always becomes OUT.

If MERGE reaches a leaf in the first tree then it will start the recursion for the second tree. The recursion in the second tree will be as in the first tree, only here the stack of partitioning lines contains lines from both trees. Note that the early termination check for infeasibility only occurs when traversing the second tree as the lines in a BSP-tree will have a feasible solution. If a leaf has been reached in the second tree, without any early termination, then MERGE performs the operation on these leaves.

In [4] it is suggested to always insert in to the smallest tree. However as we shall see in section 5.3.3 more caution has to be taken when swapping. MERGE also performs *collapse operations* while it is running. These operations create smaller trees with pointers to equal subtrees instead of having equal subtrees existing more places in the tree. While these operations are not needed for the correctness of the algorithm, it does however lead to a performance boost. The next section provide more information about the collapsing subroutine. Pseudo code for MERGE can be found in Appendix A.

3.3.2 *Collapsing*

One important subroutine of MERGE is to collapse the trees, using COLLAPSE. The motivation being to create smaller trees as the result of an operation. After a series of operations on BSP-trees the trees may grow to be very large as each new splitting line potentially creates n new line segments. Ideally we would like pointers to identical trees as seen in Figure 9.

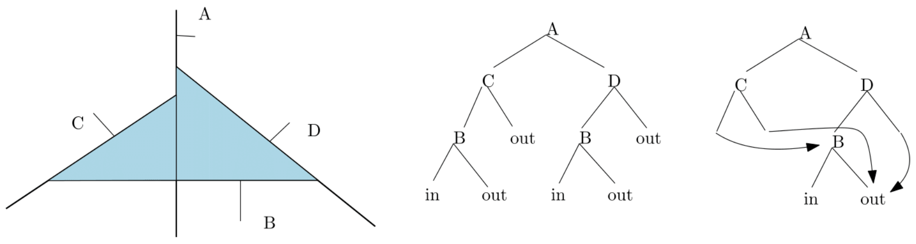


Figure 9.: A geometric space with a simple BSP-tree and finally the collapse of this tree. Arrows indicate pointers to similar subtrees.

For replacing subtrees with pointers to similar subtrees, a comparison between trees is necessary. For this we define an indexing function *id* on the set of all BSP-trees. Such that for two trees $A = \{A_h, A^+, A^-\}$ and $B = \{B_h, B^+, B^-\}$, A_h being the subhyperplane, or splitting line, with A^+ and A^- being the right and left subtrees, $id[A] = id[B]$ iff

$$A_h = B_h \quad , \quad id[A^-] = id[B^-] \quad \text{and} \quad id[A^+] = id[B^+],$$

where equality between two leaves are handled by simply comparing the type of leaves. This is not entirely enough to compress trees, we need one more constraint

$$id[h_i, B, B] = id[B].$$

This constraint enables the collapsing on more levels. This constraint states that if a node has children with the same id, then the id of the node is equal to that of the children. Effectively this means that we can remove nodes that hold no information. An example could be a node with two OUT-leaves as children. Then this can be collapsed to simply being an OUT-leaf.

To calculate the index of each subtree in the BSP-tree, we traverse the tree bottom up. In order to avoid multiple tests we will utilize a *visit map* that will keep track of visited subtrees by maintaining a map with a hashed version of the tree as key and the actual tree as value. The hash of a tree is a 3-tuple of integers

$$hash[IN] = (0, 1, 0)$$

$$hash[OUT] = (0, 2, 0)$$

$$hash[h_i, B^+, B^-] = (i, id[B^+], id[B^-]),$$

where i denotes the i'th splitting line selected by BUILD_BSP.

Initially we will will have visited no subtrees and the visit map will be empty, but as it fills up with visited subtrees we can compare the hashing of a subtree with those already visited and replace the subtree with a pointer to that which has the same hashing. This means that, if there is a collision in the hashing then it is exactly because the two subtrees are equal and they can be collapsed.

It is important to maintain the *visit* and *id* between subsequent calls. This will affect the running time as seen in subsection 3.3.2. In section 5.3.2 we will test the effects of collapsing trees to see that it does in fact yield smaller trees without any asymptotic increase to running time.

With this collapsing subroutine we are able to remove duplicated sub tress, but it does not completely reduce the size of the tree as it cannot remove sections of the tree that describes infeasible regions. In order to avoid these dead regions as seen in Figure 10 we, in subsection 3.3.3, look at the problem as a feasibility test in linear programming. Here we use the linear programming method as suggested in [3].

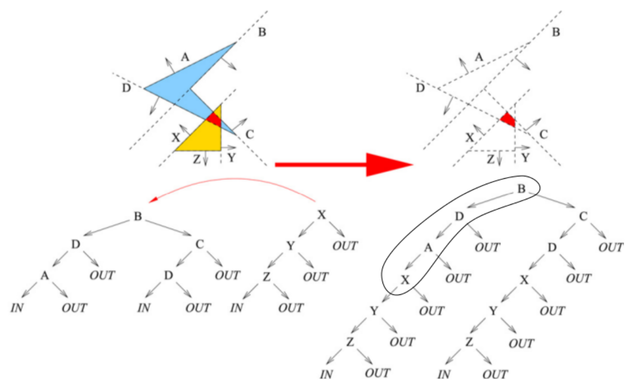


Figure 10.: (left)A scene it's two BSP-trees. (right) The BSP-trees naively being merged without any feasibility test. The four nodes BDAX represents an infeasible area, which our feasibility test takes care of. Figure taken from [4].

Analysis of the collapsing

COLLAPSE is not necessary for MERGE to return a correct BSP-tree. However, it does yield smaller trees which saves memory and ultimately leads to a performance boost.

Tree size

At the very least, our tree size is cut in half since all the leaf nodes can be compressed to only two (one for the OUT-leaf and one for the IN-leaf). All equal subtrees will also be compressed as seen in Figure 9 and 11.

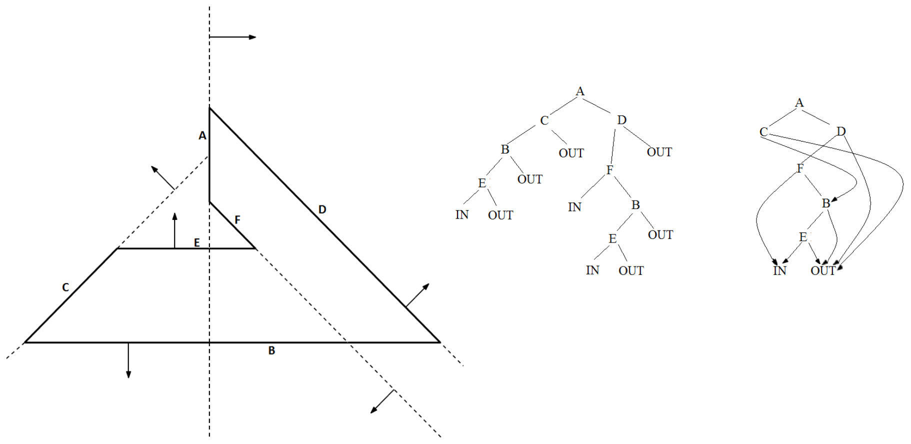


Figure 11.: A geometric space with a simple BSP-tree and finally the collapse of this tree. Arrows indicate pointers to similar subtrees.

Running time
Since COLLAPSE only visits each node at most once, a single run of the COLLAPSE will run in $O(n)$ time, where n is the number of nodes. We maintain the *visit map* and *id* throughout subsequent calls and the algorithm returns immediately, if an already collapsed tree is found. This means that the running time amortizes over several calls to an optimal $O(1)$ time. The hashing function will never result in a collision between two BSP-trees which do not represent the same set [4].

3.3.3 *Linear Programming*

In linear programming we search for a specific solution given a set of linear constraints and an objective function. A linear programming problem is typically given as follows:

$$\begin{array}{ll} \text{Maximize} & c_1x_1 + c_2x_2 + \cdots + c_dx_d \\ \text{Subject to} & a_{1,1}x_1 + \cdots + a_{1,d}x_d \leq b_1 \\ & a_{2,1}x_1 + \cdots + a_{2,d}x_d \leq b_2 \\ & \vdots \\ & a_{n,1}x_1 + \cdots + a_{n,d}x_d \leq b_n \end{array}$$

.

Here a_i, b_i and c_i are real numbers, d is the dimension of the linear problem, n the number of constraints and the function we wish to maximize is called the objective function.

In general the constraints can be seen as half-spaces in $\mathbb{R}^d$ and in our 2D case these would be lines, where only points on one side of the line satisfies the constraint. To satisfy all the constraints a point must lie in the intersecting region of all the lines. This region is called the *feasible* region. If no such region exists, the linear program is called *infeasible*. Since the feasible region is an infinite set of points, the objective function defines which one point that maximizes the solution.

For the general linear programming problem, many algorithms exists. One such algorithm is the Simplex Algorithm which can be found in [3]. For a 2D problem this algorithm is not very efficient. The idea for our algorithm in 2D is to add a constraint one by one, check for feasibility and then maintain the optimal solution for the succeeding problems.

Let H be the set of constraints to a linear problem given as hyperplanes, where h_i denotes the i'th constraint. H_i denotes the set of the first i constraints. C_i denotes the feasible region defined by the constraints in H_i. Let v_i be the optimal point in C_i.

When we add a new constraint, two things can happened:

Lemma 1. *Let $1 \leq i \leq n$, and let C_i and v_i be defined as already mentioned. We now have two cases:*

1. $v_{i-1} \in C_i$

2. $v_{i-1} \notin C_i$

Case1: *v_i is simply updated by $v_i = v_{i-1}$. If we are in case 2 either $C_i = \emptyset$ or $v_i \in l_i$, where l_i is the line bounding h_i.*

Proof. The optimal point, v_{i-1} before adding the constraint h_i also exists in the new region C_i after adding h_i, since $C_i = C_{i-1} \cap h_i$. The point is also still optimal since $C_i \subseteq C_{i-1}$. And thus $v_i = v_{i-1}$

Case 1 can be seen visually put in Figure 12. Here adding h_5, does not lead to an update of the optimal point v_i.

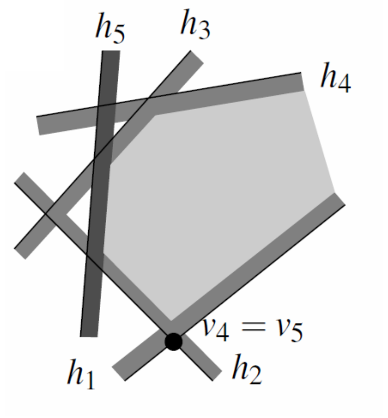


Figure 12.: Adding constraint h_5, which is not more strict. Figure taken from [3].

Case2: We add a more strict constraint and two things can again happen. First of all the constraint can make the entire linear program infeasible, so $C_i = \emptyset$. This is seen in Figure 13, where adding h_6 makes the entire problem infeasible.

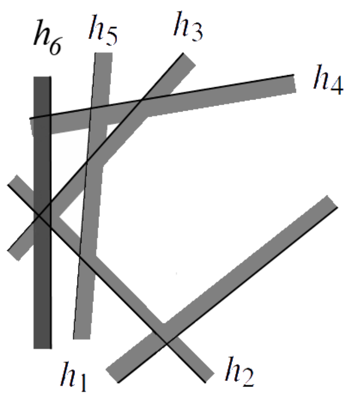


Figure 13.: Adding constraint h_6, which causes the LP to become infeasible

Let $v_i \notin h_i$. Now, suppose for contradiction that C_i is not empty and that v_i is not on the line l_i for h_i. Consider the line segment from v_i to v_{i-1}. Clearly $v_{i-1} \in C_{i-1}$ and, since C_i is a subset of C_{i-1}, then v_i must also be in C_{i-1}. So since both end points on the line segment $v_i \rightarrow v_{i-1}$ are in C_{i-1} and C_{i-1} is convex from it's constraints - then the line segment is also inside C_{i-1}.

Since v_{i-1} is the optimal point in C_{i-1} and an objective function f_c is linear, then $f_c(p)$ must increase monotonically along the line segment $v_i \rightarrow v_{i-1}$ as p moves from v_i to v_{i-1} since v_{i-1} was a better solution than v_i, since v_i is more restricted.

Now let q be the intersection between the line segment and the line l_i. This intersection must exist since $v_{i-1} \notin h_i$ and $v_i \in C_i$, meaning that v_{i-1} and v_i are on different sides of l_i.

Since the line segment is inside C_{i-1} then the point q must be in C_i, meaning that it is potential optimal point. And since the objective function increases on the line segment, then $f_c(q) > f_c(v_i)$ which contradicts that v_i is the optimal point in C_i. Instead the point q, on the line l_i, must be the optimal point. □

Figure 14 illustrates this case, where h_6 is a new, more strict constraint.

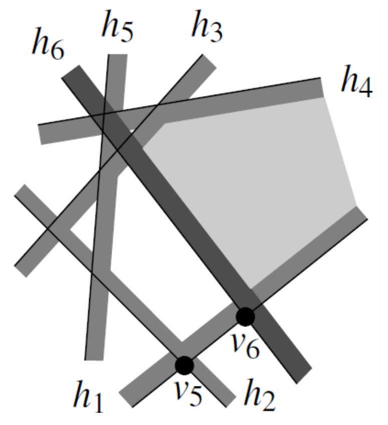


Figure 14.: Adding constraint h_6, which is more strict. Figure taken from [3].

Finding new optimal point

As we saw in the previous section, we have two cases of how to update the new optimal point. Case 1, where the old point exists in the new feasible region, is easy. Here we simply check our new constraint up against our current optimal point and check if it still satisfies all constraints. If we however reach case two and we need to update the optimal point then another approach is needed.

Since case 2 in Lemma 1 states that this new point has to lie on the line l_i that specifies the new constraint h_i, we have a 1D linear program problem. If l_i is not vertical, we can parametrize it by its x-coordinate. If l_i however is vertical, we can instead use the y-coordinate. In the non-vertical case we would like to find all the x-coordinates of the intersection points between l_i and the lines defining $h \in H_i$. If no intersection point can be found, i.e. the lines are parallel, either any point on l_i is satisfied by h or no point is. If all points are satisfied, we simply ignore the constraint, since it does not provide any restrictions to the new optimal point. If no points are satisfied a violation is found and we can terminate the algorithm and report infeasible.

In the other case, where we have an intersection point, we need to find out if that point defines a right bounding point, ie. the constraint h has its feasible region to the left or if it is a left bounding point. When adding a new intersection point, we always keep track of the maximum left bounding point and the minimum right bounding point, since they define the strictest constraint on either side. We define $x_{left} =$

$max(x_{left}, intersect(l_i, h)), x_{right} = min(x_{right}, intersect(l_i, h))$ where $h \in H_i$. The feasible region on l_i is found in the interval $[x_{left}, x_{right}]$. An example is shown is Figure 15.

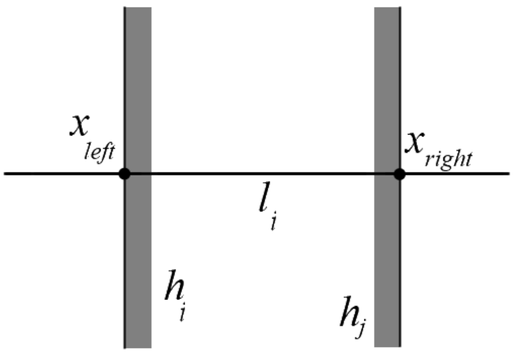


Figure 15.: The 1D problem, where $[x_{left}, x_{right}]$ is still a feasible region

If $x_{left} > x_{right}$ at some point, we can report that the entire LP problem is infeasible. This comes from the fact that the new optimal point has to lie on the line l_i, but no point can be found on this line that fulfills all the constraints in H_i. An example of an infeasible 1D problem can be seen in Figure 16.

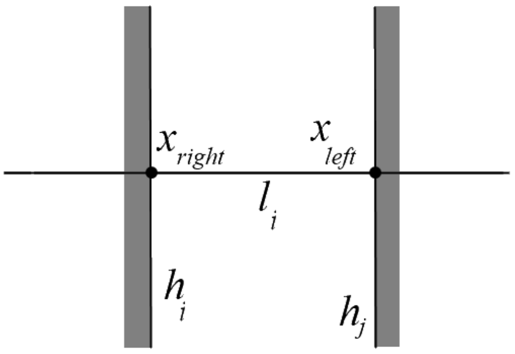


Figure 16.: The 1D problem, where $[x_{left}, x_{right}]$ does not exists.

If all constraints has been added and $x_{left} < x_{right}$, then the new optimal point is either x_{left} or x_{right} depending on the objective function.

Since we are not interested in a particular solution, but rather if one exists, our objective function is arbitrarily picked so we just have *any* point that satisfies the current constraints.

Analysis of the Linear Program Solver

The worst case behaviour for our LP_SOLVER is when we have no early-out termination. With no early termination there is no infeasibility and the solver has to run to completion. The worst case scenario happens when a newly added constraint is a more strict constraint (where we need to solve the 1D problem). Our algorithm has to run through all constraints so LP_SOLVER runs in at least $O(n)$ time. But for every constraint, we need to solve the 1D problem, which takes $O(i)$ time, since it checks all the previous constraints. LP_SOLVER takes $\sum_{i=1}^{n} O(i) = O(n^2)$ time in the worst case.

The problem with the naive incremental algorithm is that it always adds the constraints in a deterministic fashion, making it possible for a given configuration to always run in the worst case $O(n^2)$ time, because we need to solve the 1D problem in *every* iteration.

So how do we find the correct order to add the constraints? - We choose a random ordering of the constraints. The running time now completely depends on the ordering that has been chosen. So instead of analysing the running time of the algorithm, we analyze the *expected* running time. If a random permutation of the constraints can be done in $O(n)$ time before running the algorithm, we end up with an expected randomized running time for the entire algorithm of $O(n)$.

Proof. The running time of the LP_SOLVER must be the total running time of solving the 1D problem for each constraint. We know that a 1D problem for i constraints can be found in $O(i)$ time. As explained earlier there is a chance, when adding a constraint, that we do not need to solve the problem in 1D for this constraint. This happens when $v_{i-1} \in h_i$, v_{i-1} being the previous solution and h_i being the newly added constraint. This means that when adding a constraint, then it either takes no time, since no new solution is needed to be found in the line of this constraint or that $O(i)$ time is spent searching for the 1D solution. Let X_i be a random variable that indicates whether or not the added constraint requires a new solution to be found. We define $X_i = 1$ if $v_{i-1} \notin h_i$ and $X_i = 0$ otherwise. Now, the total time spent adding all $\{h_1, h_2 \ldots h_n\}$ is

$$\sum_{i=1}^{n} O(i) \cdot X_i$$

In order to bound the expected value of this sum we use the *Linearity of expectation* which says that the expected value of a sum of random variables is the sum of the expected value of those random variables. Even when the random variables are dependent, the linearity holds. This means that the expected time for solving all 1D linear programs is

$$E[\sum_{i=1}^{n} O(i) \cdot X_i] = \sum_{i=1}^{n} O(i) \cdot E[X_i]$$

To find the expected value of the random variable assume that the algorithm has found v_n to be the optimal solution. This v_n must lie in C_n and must be defined by at least two of the constraints. Removing the last added constraint h_n the region must be C_{n-1}. If v_n is not a vertex in C_{n-1} that is extreme in the direction $\vec{c}$ then adding constraint h_n must have meant that a new solution has been found at this step and that h_n is one of the lines defining v_n. Since the constraints are added in random order then h_n is a random constraint from the set of n constraints. And since the vertex v_n is defined by at least two lines then the probability of h_n defining v_n is at most $\frac{2}{n}$. The probability is largest when exactly two lines define v_n. The same argument hold for any complete subset $\{h_1 \ldots h_i\}$ of the n constraints. And so the expected value of the random variable X_i is at most $\frac{2}{i}$ [3]. Now, with the expected value of X_i, we define the total expected running time of solving a linear program with n constraints as

$$\sum_{i=1}^{n} O(i) \cdot \frac{2}{i} = O(n)$$

□

3.3.4 *Running Time Analysis for MERGE*

Consider MERGE without COLLAPSE. If a resulting BSP-tree from MERGE has n nodes and height h, we know that MERGE must run in at least $\Omega(n)$ time, because it constructs its result incrementally. If we then, after MERGE is done, collapse the resulting BSP-tree it will have no impact on the total running time of MERGE, because the collapsing can be done in $O(n)$

time as well. Using COLLAPSE within MERGE will therefore not affect the asymptotic running time. Consider now a set of n vertical bars intersected by a set of n horisontal bars. This creates n^2 squares, meaning that the running time of MERGE is at least $\Omega(n^2)$ worst case. Here, the COLLAPSE subroutine will also not affect the running time of MERGE asymptotically.

If we look at the pseudo code for MERGE, we see that we have a tree called T_{right} and a tree called T_{left}. Both of these trees cannot be NULL, since they represent a partition of a non-empty region. A partition of a non-empty region will lead to at least a non-empty sub region in one of the sides. Because both T_{right} and T_{left} cannot be NULL at the same time, MERGE is not called more than twice as the size of the final tree, which is h. This means that the number of calls to MERGE is $\Theta(n)$.

The number of constraints given to our LP_SOLVER is at most the height of our resulting tree. This is reached if no infeasibility is found before we reach the bottom. We can have fewer constraints, but we cannot exceed the height of the tree and hence the number of constraints is $O(h)$. The time it takes to run our feasibility test is then $O(LP(h))$. As we saw in section 3.3.3 the expected running time for our LP_SOLVER was $O(n)$ time, meaning that in our case it would take $O(h)$ time.

In total we end up with a running time of $O(nh)$, but as we will see in the optimization step of section 4.3.2, this time can be further improved to a running time of $O(n)$. It is done by taking advantage of the fact that the constraints can be added one by one, in an incremental behaviour. This ultimately leads to a reduction of the cost, for testing the linear programming feasibility, to a constant.

4 IMPLEMENTATION

In this section we describe the algorithms implemented, the subroutines needed for the algorithms along with the environment in which we implemented the algorithms. In section 4.1 we will discuss BUILD_BSP, how to select a splitting line and the structure of the BSP-tree. The algorithm is discussed in further detail in [5], and pseudo code of our implementation can be found in Appendix A. In section 4.2 we will go into the implementation of INC_SET_OP that merges a BSP-tree with a BREP. A detailed description of this algorithm can also be found in [5], along with pseudo code in Appendix A. Next, in section 4.3, we discuss MERGE that merges two BSP-trees. We focus on the implementation of LP_SOLVER, the usage of subroutines along with optimisations in more detail than in the article [4]. As with the other algorithms we have written our own pseudo code, that can be found in Appendix A. Finally in section 4.4 we will discuss general implementation and environment details.

All code are implemented in C++11, using smart pointers.

4.1 BUILDING A BSP-TREE

For building our BSP-trees we used the method from [5, p. 155]. The pseudo code can be found in Appendix A, where we assume a random permutation has already been applied to the line segments. It takes a heuristic for choosing a splitting line, and a BREP in the form of oriented line segments to generate a BSP-tree according to the splitting heuristic and the set of line segments. The BREPs that we represent are solid representations, meaning that they do not have any gaps.

The orientation of the line segments can of course be represented in many ways, but our approach was to define a line segment as two points and a normal, pointing towards the exterior of our solid polygon.

The pseudo code is quite vague in *how* to partition the line segments in F with a splitting line H, but also how to even choose the splitting line. A further discussion upon how to chose a splitting line can be found in section 4.1.2.

4.1.1 *Partitioning of Line Segments*

When a splitting line has been found, we need to partition our line segments in F into three lists; F_{left}, F_{right} and F_{same}. We do this by going through the line segments one by one and determining how to partition a specific line segment. This partition leaves two cases, of which one is easy to handle and the other needs a bit more tampering.

Because we work with oriented line segments, our splitting line also need to have a direction. We did this by defining a line from a normal and an origin, both being vectors. The origin and normal are being used both to define the position of the line, but also to define the direction because the normal again is pointing towards the exterior of the polygon.

Case1 is where both end-points of a line segment lies entirely on one side of the splitting line. In other words, the splitting line does *not* cut the line segment. In this case we simply have to determine on which side of the splitting line our points are. Here we used the fact that we had a normal and an origin to produce algorithm 2 seen in Appendix A.

Case2 is a bit more tricky, because we here have an end-point on either side of the splitting line. We here extend our line segment to a line, and find the intersection point between those two lines using the determinant. Depending on which point are on which side of the splitting line, two new segments are made. The first segment is between the start point of the line segment, and the intersection point with the splitting line. The second segment is between the same intersection point and the endpoint of the line segment. The two new segments are then put in F_{left} and F_{right} accordingly.

When the partitioning of the line segments are done, the rest of the algorithm just continues recursively down in F_{left} and F_{right}. Line segments that lie *on* the partitioning line is put in a separate list F_{same}, which is stored on the node v, where $l(v)$ is the partitioning line at node v.

4.1.2 *Selecting Splitting Lines*

For selecting the splitting line when building a BSP-tree, we use the method called *auto partitioning*. This means that the splitting lines has to be an extension of the line segment given by the BREP. To select what line to use, we used the idea from 2DRandomBSP(S) in [3] , where we pick a random line segment each time. This does not give us a minimum tree, not even expected, but tends to give a balanced tree with expected size $n + 2n \ln n$. A perfect balanced tree is a tree that has the minimum possible maximum height for the leaf nodes. Because we use auto partition the worst case depth, is when we have a convex figure. Here everything will always be put in one side of the tree and the other side will be either an OUT- or an IN-leaf.

An improvement here, could be to implement different heuristics of what line segment to choose. It could for example be worth finding the line segment that splits the fewest other line segments. To greedily always pick the line segment with fewest splits would however not be an solution to finding the optimal size BSP-tree, since a choice earlier in the process affects the cost later.

Another heuristic could be to find a segment that makes a "good" split, that is a split where each side of the line contains approximately the same number of segments. This heuristic tends to create balanced trees when there is a possibility because each side of the recursion gets approximately the same size subproblem. Since we are using the auto partition method, where we simply extend the line segments into splitting lines, a BSP-tree of a convex polygons cannot be made better by using this heuristic. In this heuristic we also first need to *find* this "good" split, which in worst case is a search through every line each time. We have not been experimenting with this heuristic.

4.2 SET OPERATIONS ON BSP-TREES AND BREPS

The INC_SET_OP is basically a complex version of BUILD_BSP. It uses many of the subroutines from BUILD_BSP, such as the partitioning of line segments as seen in 4.1.1, but also BUILD_BSP itself. It does however also come with an extension in the form of the In/Out test, which implementation details will be briefly discussed in the next subsection.

The pseudo code for INC_SET_OP can be found in Appendix A.

4.2.1 *In/Out Test*

The implementation of the In/Out test is basically as described in the theory. We start by finding the closest point b to our splitting line. We then determine if b is on the boundary of $R(v)$ by going through the other points in the interior of $R(v)$ and see if they connect to b. If two connecting points exists then both an $e1$ and an $e2$ edge must exist in $R(v)$. Otherwise only one edge e is found. After that we simply do the checks as in theory, but instead of having a point p on our line, we simply check that the whole line is on one or the other side of the edge e.

4.3 MERGING OF BSP-TREES

In this section we describe the implementation details used in the Improved Binary Space Partition merging. We start by talking about COLLAPSE and some implementation details related to this subroutine. After this we discuss the implementation details regarding the LP_SOLVER.

4.3.1 *Collapse*

As we saw in section 3.3.2, a single run of COLLAPSE is done in worst case linear time and the amortized time is $O(1)$ because we keep track of the trees seen between subsequent calls to COLLAPSE.

In practice we did this by having three key factors. First of all we had a *count* which was used as the *id* of unseen nodes. This count was initialized with a 0 and when we met an unseen node, we would set the id of that node equal to the current count and increment the count.

Second of all we had an *idMap* that held a connection between a node and the id of that node.

Lastly we had the map *visit*, which was a map from a hashing of a node to the actual node. This hashing of a node is a key part of COLLAPSE since we, instead of having to compare two whole trees, could just compare their hashing value; a 3-tuple as seen in section 3.3.2.

If we hash a node and look it up in the visit map, a match will mean that a node with same splitting line and same subtree is already found in the tree. This node can simply be returned. The tree now only contain nodes that can be pointed to by several other nodes, and thereby having more than one parent. We do no longer have a binary tree, but instead a directed acyclic graph. We can however still traverse it as usual if we start top down. The pseudocode of COLLAPSE can be seen in Appendix A.

4.3.2 *Linear Program Solver*

We've implemented a simple algorithm where we give a list of constraints, to the solver (LP_SOLVER) and it returns either true or false (feasible/not-feasible).

In the case where no previous solution is given to the linear program the solver starts by shuffling the vector of lines giving a better expected running time. After that it will select a starting line, on which a point will be selected. This point is used as our initial solution point. Now, it iterates over the remaining constraint lines and for each line l_i we solve the 1D linear program, where we want to find the maximum left boundary point $left_{max}$ and the minimum right boundary point $right_{min}$. We do this by finding intersection points with all previous lines one by one. For each intersection point between our new constraint l_i and a previous constraint defined by l_j, we find out whether it is a left boundary point or a right boundary point. We then update $left_{max}$ and $right_{min}$. If $left_{max}$ ever gets larger than $right_{min}$, we terminate with false. To find out whether we have a left boundary point or a right boundary point we first look at if our new line constraint l_i is vertical or not. If it is a vertical constraint, we solve the 1D problem along the y-axis, otherwise the x-axis. If the constraint is not vertical, we take a point that is a little left (closer to 0 along x-axis) on our new constraint line l_i. We then figure out if this new point is in the feasible or infeasible region of the constraint h_j defined by the line l_j. If that point is in the feasible area, the constraint must be a right boundary bound and vice versa. A detailed description of how LP_SOLVER is optimised by using the previous solution can be found in 4.3.2.

We know from subsection 3.3.3 that if no intersection point can be found(the lines l_i and l_j are parallel) either all points on l_i are feasible, or no point is. To find out which case we are in, we pick an arbitrary point on l_i and checks if it is in the feasible or infeasible region of h_j defined by l_j. If we are in the infeasible region we simply terminate with false. Else we simply update the current optimal point to be the arbitrary point chosen on l_i.

This leaves us with two early out conditions. 1) The lines are parallel and the new point is in the infeasible area. 2) The maximum left boundary point has a larger x/y (depending on whether or not we have a vertical new constraint) than the minimum right boundary point. When all constraints have been checked and no violation has been found, we simply return true as in the constraints defines a feasible area.

The pseudo code for the LP_SOLVER can be found in [3].

Optimisations for the Linear Program Solver

The algorithm LP_SOLVER is in itself correct and optimised enough for the purpose of this study, but we can make use of the environment that we are in to make it faster.

Taking Advantage of Closed Polygons

A polygon has by definition a feasible region, the interior. If LP_SOLVER is run on a feasible set of lines, then it will run in $O(n^2)$ time. This motivates us to never check for feasibility as long as the lines are from the same polygon. When MERGE reaches a leaf in the first tree, A, it will swap it with the second tree, B and begin traversing this tree. Because of this, the first feasibility check is needed when B becomes a leaf, right after the first swap. This assumption is also safe when using the heuristic for swapping the trees when B is smaller. If $height(B) < height(A)$ then surely any subtree of B will also satisfy this. Meaning that the swap of trees will only happen when a leaf is reached or before any traversal. Were we to use heuristics that swapped if B was larger than A, in height or total number of nodes, then the swap could occur at subtrees and the assumption would not hold.

Figure 17 shows where the first feasibility test is run (red), which lines are on the stack (grey) and the subsequent calls (green) run in $O(1)$ time.

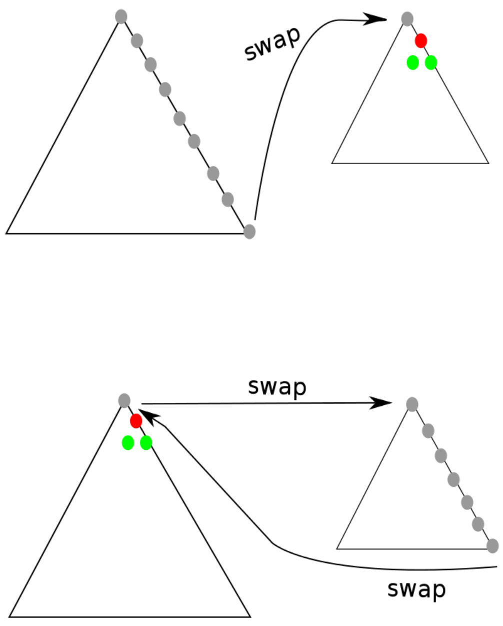


Figure 17.: Top: the traversal of nodes if no swap heuristic is used. Bottom: the traversal of nodes if a swapping heuristic that will swap with trees with lower height is used.

Save Feasible Solution

The second optimisation that we implemented was to save the solution for a linear program for the next problems. This is safe because any subsequent calls to the solver will be with only one new line to the previous set of lines. If we save the previous solution then we can quickly determine if the new line is a more strict constraint or not, and in the case where it is, find the new solution on the new line. This means that instead of treating each set of lines as a new problem, then we actually treat the entire set of lines from both polygons as one big problem and only solve it for one line at the time. This means that we instead of using $O(n)$ expected time at each recursive call in MERGE, spent expected $O(1)$ if there is a previous solution and expected $O(n)$ if there isn't.

Looking at the two optimisations together, then the first call to LP_SOLVER will be when we reach a leaf in A, swap the trees and insert the root of the new A on the stack. LP_SOLVER will spend $O((h+1)^2)$ time if the root does not make the problem infeasible, where h is the height from this leaf to the root. And it will spend expected linear time if it is not feasible. In the case of the problem still being feasible LP_SOLVER continues the recursion of the the new A tree and each call to LP_SOLVER will here only take expected $O(1)$ time not depending on it being feasible or not.

4.3.3 *Reduction of Trees*

It is possible that when we find an infeasible region that the sibling of this infeasible is an OUT-leaf. This means that we overwrite their parent to be this OUT-leaf. If the sibling of the original parent is also and OUT-leaf, then we will now have a node with two children, both being an OUT-leaf. It is mentioned how to handle this in [4] using COLLAPSE. We chose to handle it outside of the collapsing subroutine by simply overwriting the parent of the two OUT-leaves with an OUT-leaf. This is correct because the line simply partitioned an OUT-region into two, which is redundant information. The reverse case with IN-leafs are treated in the same way.

4.4 ENVIRONMENT

4.4.1 *Data Types and Structures*

When representing a BSP-tree we chose to have an implicit structure of nodes. Each node has 3 pointers to other nodes. One to the parent, the left and the right child. The root has a null pointer as parent and leaves have null pointers as children. We chose to have this implicit structure because it is easily mutable and when collapsing trees it enables us to destroy the binary properties of the tree, potentially having the same subtree in several places, but all being the same place in memory. Each node also has a type, where internal nodes all have the NODE type and each leaf has either an IN or an OUT type. This enables us to query if the current node is a leaf before accessing any children. Finally each node has a vector of pointers to line segments stored. These represent the line segments that are all coincident and create the splitting line selected at this node.

Line segments are simply 2 vectors denoting the start and end point of the line segment and a normal. The normal is calculated from the start and end points so they are easily flipped by switching the end and start point. We've implemented a vector type where the usual operators, $\cdot, +, -$ etc. are overwritten by the corresponding vector operations. For each line segment a line can be created on the form $y = a \cdot x + b$. Horizontal and vertical lines are handled by special cases.

4.4.2 *Numerical Instability*

Because we worked with double precision, some care had to be taken. The lack of exact precision, would for instance some times lead us to be unable to correctly determine if a point was *on* a line or just very close to it. To work with this numerical instability we chose to use a machine epsilon. This means that we do not work with absolute precision, but do allow some slack as seen in the following:

```
isEqual(a,b){
    if(abs(a-b) < EPSILON)
        return true;
}
```

Here the size of ϵ determines how much slack we allow. The value of ϵ was something that we adjusted along the way, so that errors were minimized.

4.4.3 *Smart Pointers*

We chose to implement smart pointers while implementing the different algorithms. They enable us to have a pseudo garbage collector that deletes pointers as soon as they are not used anymore. This means that we do not have to worry about memory leaks when testing our algorithms on larger input. And when recursing into trees, creating new, flipped line segments at each recursion and overwriting children it is hard to keep in mind what to delete and what not to delete. We did encounter problems with memory leaks as we started implementing the algorithms as we used standard C++ pointers, and switching to smart pointers fixed that problem. This is of course only a problem because we used C++ to implement the algorithms and because we were not able to manually control the memory correctly. We figured that the overhead of creating smart pointers outweighed the downside of running the algorithms inside the java virtual machine. We also preferred C++ because it is a fast, type strong language and because we are able to illustrate scenes using OpenGL.

4.4.4 *Compiler Optimisation*

When using linear programming to determine feasibility in MERGE, a lot of the computational power is used to compute intersections between lines. As the amount of lines can get very large, then it is important that this computation is as effective as possible. The biggest issue in this intersection computation is the division when calculating the determinant. This can be optimised to multiplying with the inverse, but instead of manually optimising all the important computations, we use the O2 compiler optimisations for the C++ compiler. These optimisations will compile any arithmetic to be as optimal as possible. We chose to use O2 optimisation as we could sometimes see small decreases in speed when using the O3 flag.

4.4.5 *Visualisation*

For illustrating our polygons we parse our line segments to 2D vertices for OpenGL, they are drawn using the libraries GLFW and GLEW. For debugging, we printed the trees out using a textual printer.

5 EXPERIMENTS & ANALYSIS

In this chapter we will show our results of running experiments on different inputs, given our implementation of INC_SET_OP ($BSPT \times BREP \rightarrow BSPT$) and MERGE($BSPT \times BSPT \rightarrow BSPT$). We will in this chapter analyse the running time of the algorithms and the height and the size of the resulting tree.

Because of COLLAPSE, that is described in section 3.3.2, the tree size needs to be calculated in a special way. COLLAPSE will make a BSP-tree into a Directed Acyclic Graph (DAG), so to count the size of the tree we go down through the tree and count unique pointer addresses to memory. This means that if a node v is being pointed to by several parent nodes, it will only be counted once. By calculating the tree size in this fashion we are also able to give the size of a normal uncollapsed BSP-tree, as every address of a node in such a tree will be unique. We are not adding the leaves to the size of the tree, so the size of a tree will be the total number of unique nodes.

The height of a tree is measured by comparing all routes from the leaves to the root and the longest route is the height of a tree.

The running time is the time it takes to do the actual merging operation. If it a nested operation, i.e. multiple polygons being merged, then we measure the accumulated time of the merging operations. We want to examine the difference between using MERGE on two BSP-trees and INC_SET_OP on the same polygons represented as a BSP-tree and a BREP. The time used for building the BSP-trees is therefore not included in the running time.

All above measurements are compared to the number of segments in the total scene, meaning in all polygons to be merged. To calculate this we simply accumulate the size of the BREPs to be merged. Note that two scenes with the same amount of line segments can produce very different results, as building the

BSP-tree and both of the algorithms for combining polygons are dependent on the structure of the scene.

5.1 COMPUTER SPECIFICATIONS

All the tests are run on a PC running Ubuntu 14.10. The PC has 3.7 GiB of RAM and the processor is an Intel Core i3 CPU M 330 @ 2.13GHz x 4.

5.2 GENERATING TEST DATA

For testing the algorithms we chose to use simple polygons, with the exception of parallel lines. A simple polygon is a closed polygon with no self intersecting lines and well defined edge between interior and exterior. A simple polygon has to be drawn without lifting the pencil, so a donut-form is for instance not allowed. Such a shape can however be represented as a small circular polygon subtracted from a larger circular polygon. Since our results from the two algorithms will always be BSP-trees, which *can* represent a polygons with holes, we will allow such results. The reasoning behind choosing simple polygons as input is the equality between simple polygons and regular sets as explained in section 3.1.2. Having only simple polygons and using only regular set operations ($\cap^*, \cup^*, -^*$), we ensure that the resulting tree is also a polygon. This invariant can be broken when using $\cap^*$ or $-^*$ if the operation separates the result into two separate simple polygons. To avoid this we never create a scene in which this would happen.

All of the polygons created can be transformed using the transformation matrix to give better flexibility when creating a scene. Given a point (x, y) it can be rotated, scaled and translated according to the following matrices

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \cdot \begin{bmatrix} cos(\theta) & sin(\theta) & 0 \\ -sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \cdot \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \Delta_x & \Delta_y & 1 \end{bmatrix}$$

5.2.1 *Convex Polygons*

For creating convex polygons we had special case generation for triangles and squares. In order to generate a random convex polygon we, given a center coordinate, generated n random points on the circle with radius r from this center point. The random points on the circle are generated within a angular range depending on the number of points to be generated. As an example; if there are to be four points on the circle then each point is generated within four separated angles of $\frac{1}{2}\pi$ radians. We did this in order to create a convex polygon with a large interior. As a special case we also made a generator that produces n vertical or horizontal lines. The first two lines will have normals pointing away from the space between them, so will the third and forth line and so on. This means that $\frac{n}{2}$ IN-sections are created in every other spacing. Examples of convex polygons as well as squares and lines are illustrated in Figure 18.

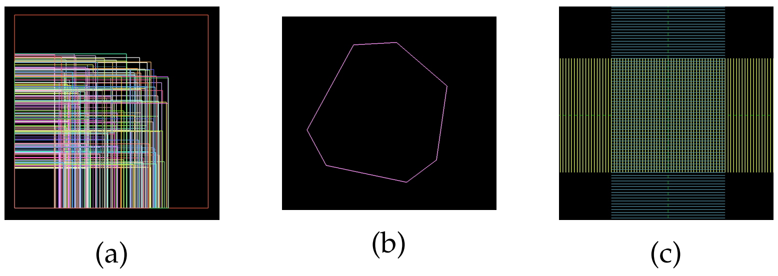


Figure 18.: Convex polygons where (a) is squares, (b) is a random convex polygon and (c) is multiple horizontal and vertical lines.

5.2.2 *Random Polygons*

In order to generate a random polygon we define an x-range and an y-range within which we generate n points according to a uniformly random distribution. Having these points we sort them on the x-coordinate and find the maximum and minimum. Between the maximum and minimum we draw a line and partition the points into those above the line and those below. These two partitions are then connected, in the upper from lowest x-coordinate to highest x-coordinate and reverse in the lower. The two partitions are then connected using the minimum and the maximum points. Having sorted the two partitions we ensure that there are no overlapping lines in either partition. Since we connect through points that are between the partitions, then these lines will also not intersect anything. This gives us a random polygon. If the range is small and the number of points to be generated is large, then the upper and lower partitions will consists of lines that are close to vertical. Generally these polygons contain a lot of *jaggies* or *spikes*, which we will analyse the consequences of in section 5.4.1. To simulate these jaggies created by random polygons we also made a polygon generator that could generate n jaggies on the side of a rectangle. These figures are illustrated in Figure 19.

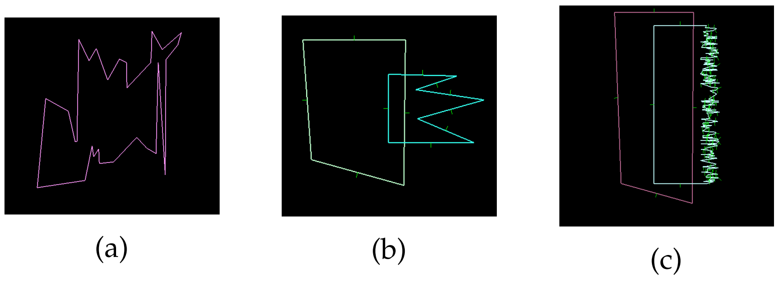


Figure 19.: A random polygon is shown in (a) and simulations of the jaggies, both few (b) and many (c).

5.2.3 *Sierpinski Curve as a Polygon*

We also created a generator for very dense polygons. For this we used the Sierpinski Curve which is a fractal of a line. Given a line with length l and an order i, then the Sierpinski algorithm creates a fractal of this line. For each order the line is divided in three recursively. Within these divisions the smaller lines are turned 60 degrees alternating clockwise and counter-clockwise. Figure 20 are squares where the edges have been made into a Sierpinski curve of increasing order.

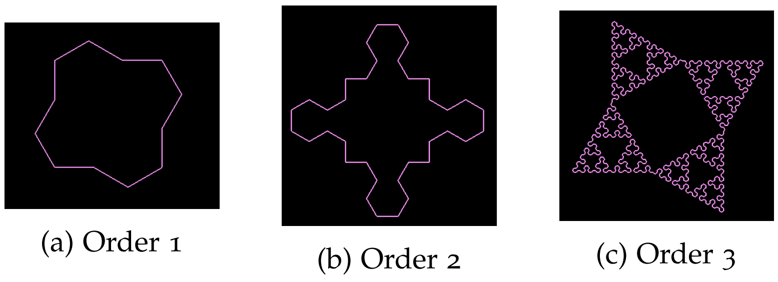


Figure 20.: Different orders of the Sierpinski curve.

These polygons create n very dense regions. The problem with these polygons is that they give rise to numerical instabilities, see section 4.4.2. We handled this by adjusting the ϵ in our double comparator for the scene. It was, however, not always possible to find an ϵ that would not lead to errors, so the scene would have to be changed a bit until no errors occurred. The numerical errors arise because of the many line segments that lie on the same splitting line.

5.3 SUBROUTINE USAGE

In this section we will analyse and discuss the subroutines LP_SOLVER, COLLAPSE and SWAP used by MERGE and try reasoning about how often they should be used. In our tests we did not implement a heuristic for swapping. Instead we simply set MERGE to swap the arguments or not. The reasoning behind this is that it would affect the running time of MERGE as we would have to measure the size of the tree. Alternatively we could, at each node, have maintained the size of its subtree. We wanted to test the scenes with swapping enabled and disabled, so we simply ran the tests twice, swapping manually.

5.3.1 *Linear Programming*

In this section we will look at the impacts of using LP_SOLVER in MERGE. We will show how tree dimensions and running time is affected by the Linear Program Solver. We will try to justify, why we always have LP_SOLVER turned *on* in all other experiments. The trade-off between having it on or off is that MERGE might miss out on some early-out terminations compared to the extra time it takes to do LP_SOLVER. The theory behind the Linear Program Solver, can be found in section 3.3.3.

The Scene

The scene used for this experiment is less than 10 random simply polygons, with between 3 and 10 edges each. Each polygon have the same origin point, meaning we could use INTERSECTION without having to worry about ending up with a simple OUT-leaf as output.

Results

The first thing we measured was the size of the resulting tree and the maximum height of the tree. The result of this measurement is shown in figure 21.

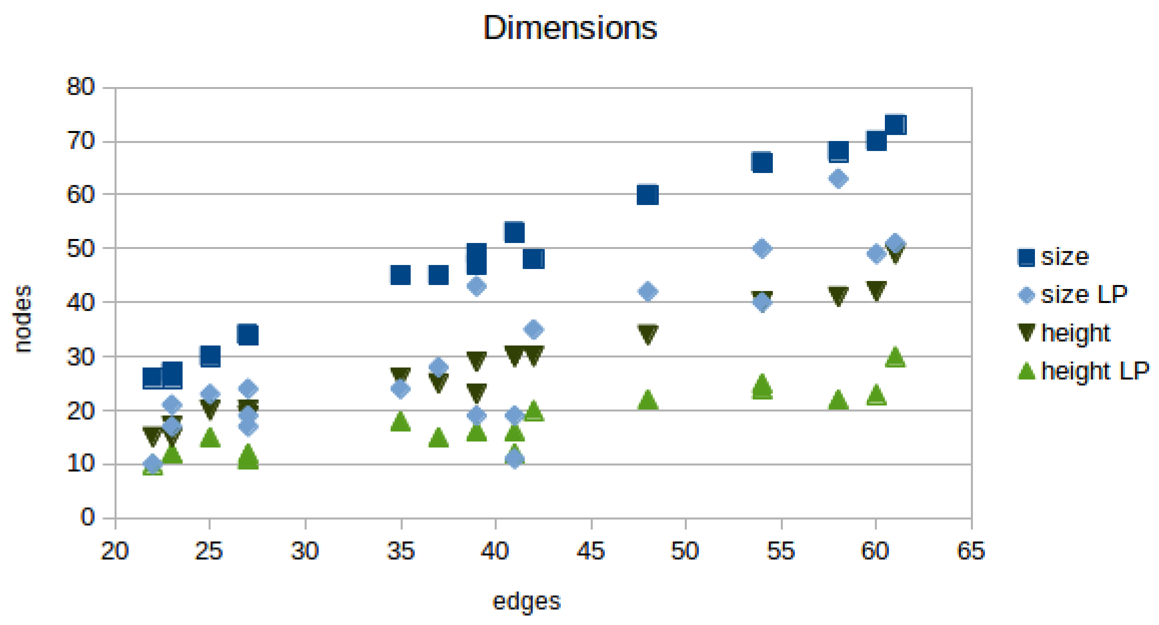


Figure 21.: Dimensions with and without linear feasibility solver on random input.

The results obviously shows that more edges from the input also leads to a bigger tree. The results also show a difference in size and height with LP_SOLVER enabled or disabled. All measure points where LP_SOLVER was enabled are lower than where LP_SOLVER was disabled.

The other thing we measured was the running time of the same experiments. Here again an obvious fact can be induced - larger input leads to longer running time. The results also show, as seen in figure 22, that a clear time advantage is coming from enabling LP_SOLVER. When LP_SOLVER was off(blue) the time it took to make the merge operations was drastically longer than when it was on(red).

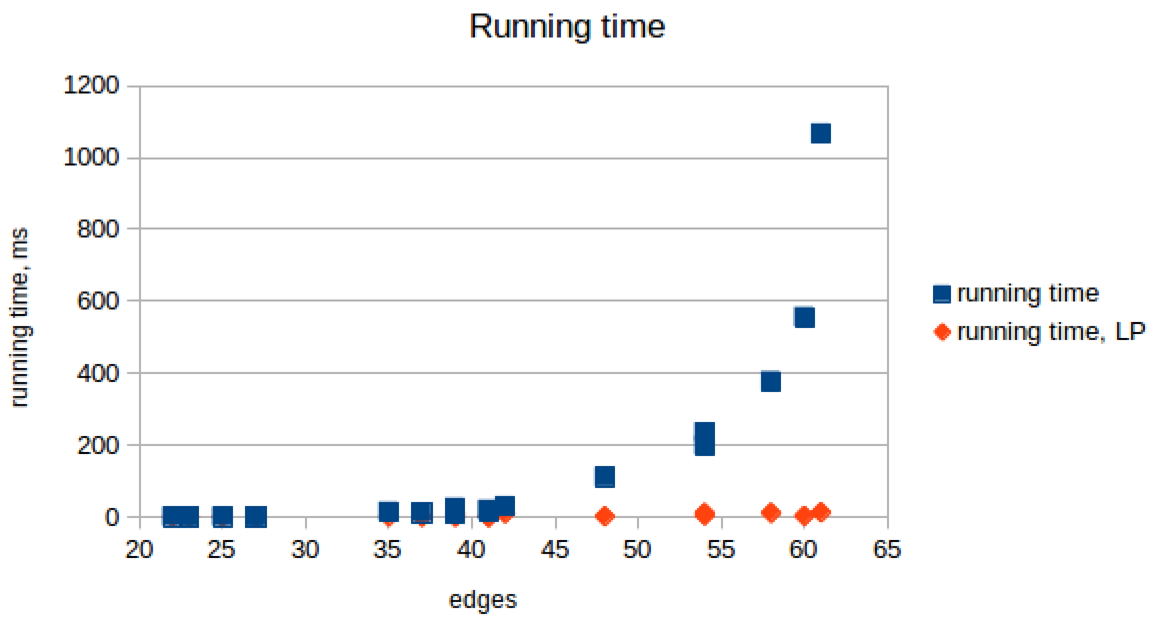


Figure 22.: Running time with and without linear feasibility solver on random input.

Analysis

Because our experiments with LP_SOLVER uses random input, it says something about the expected case. When looking at the theory from section 3.3.3, but also the running time analysis of MERGE in section 3.3.4 it is clear that our results are in line with the theory. We see a clear running time optimization when having LP_SOLVER turned on. This of course comes from the fact that the recursion can be early terminated, combined with optimizations made to our LP solver, as seen in section 4.3.2. That our tree also gets smaller comes from the fact that infeasible regions are left out of the resulting tree. An example of this can be seen in figure 10, where LP is disabled. The region defined by nodes BDAX is actually infeasible. With LP_SOLVER enabled the resulting tree would only consist of CDXYZ. In the pseudo code, it again comes from the fact that R is infeasible and we then return NULL. This leads to a smaller tree, when we return either T_{left} or T_{right}.

We only ran the LP_SOLVER on random data, which we have shown here. Whether or not the LP_SOLVER is still useful for different input is not something that we have experimented with. If a scene would have no infeasible regions then enabling

LP_SOLVER would mean that for every IN- or OUT-leaf there would be a call to LP_SOLVER. The stack of lines in each of these calls would always produce a feasible problem and the running time for each of these calls would be $O(n^2)$. After these calls LP_SOLVER would run in constant time as it has a previous solution. Consider now the structure of such a scene and let the operation be INTERSECTION. For everything to be infeasible the two polygons would have to be exactly equal. For the UNION operation the second polygon would have to be the complement. We do not produce such scenes and for any other scene, infeasible regions are to be expected.

With the results we have, we cannot justify that we should *not* be using LP_SOLVER. It simply has no disadvantages, and it only gets better time wise as the input size increases. Even though our experiment with the LP_SOLVER are using randomized input and therefore only says something about the expected case, we will be running the remainder of our experiments, where many also uses random polygons, with our LP_SOLVER turned *on*.

5.3.2 *Collapsing*

COLLAPSE is applied to the return value of any subtree. It is important for the running time of MERGE that this subroutine is not leading to too much of an overhead. As explained in further detail in 3.3.2 the tree is collapsed by replacing similar subtrees with a pointer in all but one case. This makes the tree noticeably smaller in memory. MERGE on a collapsed tree still has to traverse into all subtrees uniquely as the splitting lines on the stack will be different for each subtree. In this test we want to justify the use of collapse on each and every return value. We are interested in the effect in just one merging step and not several chained operations. If the running time has not changed in a drastic way, but the trees are smaller, then there is no reason to *not* use collapse in chained operations. This will make the final output smaller in memory and there is a chance of simply creating pointers to similar subtrees leading to even further memory saving and also faster collapsing.

The Scene

We create two random simple polygons with 3 to 40 edges and find the INTERSECTION of these polygons. The INTERSECTION exists as the polygons have the same origin point. The INTERSECTION of that scene is then computed with and without using collapse.

Results

First we look at the dimensions of the resulting tree. It should be noted that the leaves are not counted in the size of a tree as the leaves are always collapsed to only being one IN- and one OUT-leaf.

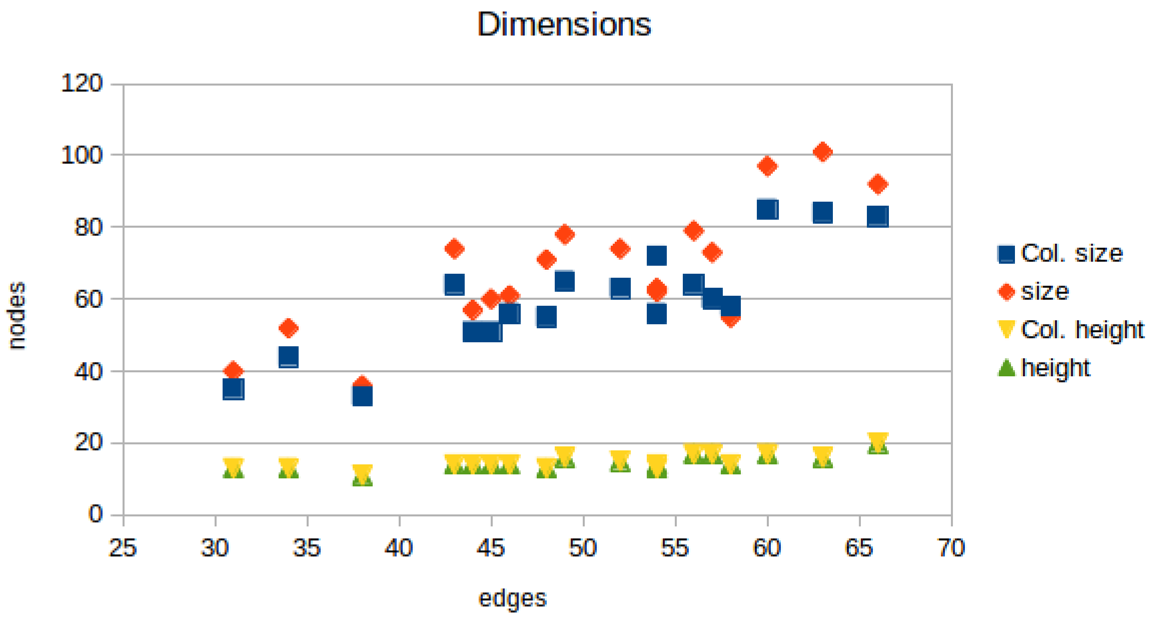


Figure 23.: Dimensions with and without collapse on random input.

As figure 23 shows, then the height of the tree does not change (shown in yellow and green). The size of the tree is measured by counting the number of unique memory addresses in the tree and this number is clearly smaller for the collapsed trees (blue) than the trees that are not collapsed (red). We do however, in some cases, have the same size.

The second part of this experiment was to see if COLLAPSE made a large impact on the running time of the algorithm.

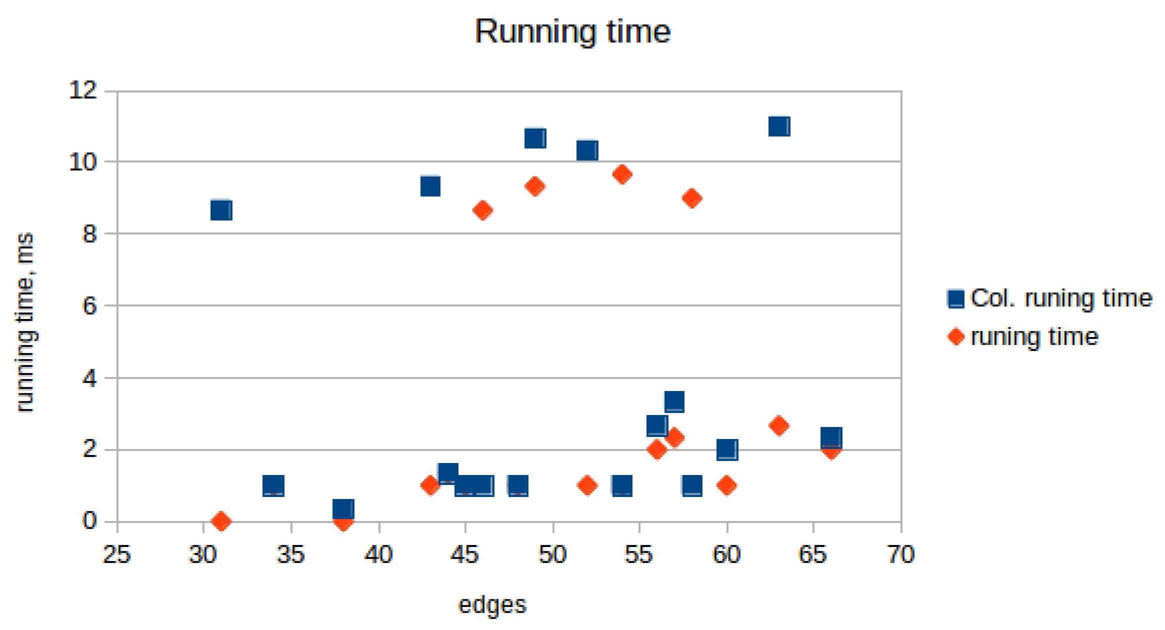


Figure 24.: Running time with and without collapse on random input.

Figure 24 for the running time of MERGE, shows that in most cases MERGE with COLLAPSE enabled is slower, in some cases the running time is the same and there are even some cases where MERGE is faster with COLLAPSE enabled. Which case is better does not seem to be connected to the input size. This suggest that the result is more related to the structure of the input scene more than it is to the size of the input. The clustering of the data points into two (around 10 ms and around 1 ms) is not something that we are able to explain. It could be something about the resulting trees actually being equal even though the amount of edges in the scene are different. This could come from a lot of irrelevant edges that does not contribute to the resulting BSP-tree.

Analysis

It makes sense that the height of the tree does not decrease as we measure the height by traversing from the root to all leaves and find the longest route. This number does not take into consideration if the leaf is only a pointer or not. The reasoning behind this way of measuring the height is that MERGE traverses down in the same way.

In most of the test runs, the size of the tree decrease when we use COLLAPSE. The decrease is never extreme and always within $\sim$ 20%. This is supporting the theory that it is only smaller subtrees that are collapsed. When two or more similar subtrees exists, then they must have the same splitting line as the root and the same nodes as left and right children. This is not something happening many places. Since it is not always possible to find similar subtrees then the size of the tree should not always decrease. But even if the size of the nodes is the same, then the number of leaves will still have been reduced to two unique addresses. Meaning that the memory usage will always be smaller and at least be halved.

The running time of MERGE should not be dominated by the running time of COLLAPSE. The running time of MERGE is expected to run in $O(n)$ time and COLLAPSE runs in $O(n)$, and when saving the visit maps for subsequent calls it will run in amortized constant time (as seen in 3.3.2). This means that only the overhead and the constant of the running time for COLLAPSE should affect the total running time of MERGE. The total running time will only expected run in linear time, and if this happens then clearly figure 24 states that COLLAPSE will not affect the running time as much as the input structure and the splitting line selection will.

So even though COLLAPSE does not give an optimal reduction of the subtree and can only be used for labeled scenes, where lines that are split maintain the same label then the running time will not asymptotically affect the running time of MERGE. It will in the very worst case only reduce the memory usage by half when reducing the number of unique leaves to two. For all coming experiments we will enable COLLAPSE.

5.3.3 *Heuristics for Swapping*

The result of merging two BSP-trees should always represent the same IN-regions. However, it is not guaranteed that the trees will be the same or even that the time taken to create them is the same. In Lysenko et. al ([4]) it is suggested to use the heuristic of always merging into the shortest tree. We argue that this is not always the case and one should be careful with using this heuristic. In this section we will not show any experiments, rather explain and analyse the consequences of swapping the arguments in MERGE.

Consider as an example figure 25, where a chain of merging operations have already happened. Three convex polygons have been merged using UNION to produce the binary space partitioning illustrated by the five IN-regions denoted 1 to 5 and the ten OUT-regions, all labeled OUT. Given a new BSP-tree, the blue triangle, we now want to UNION the grey polygon with this.

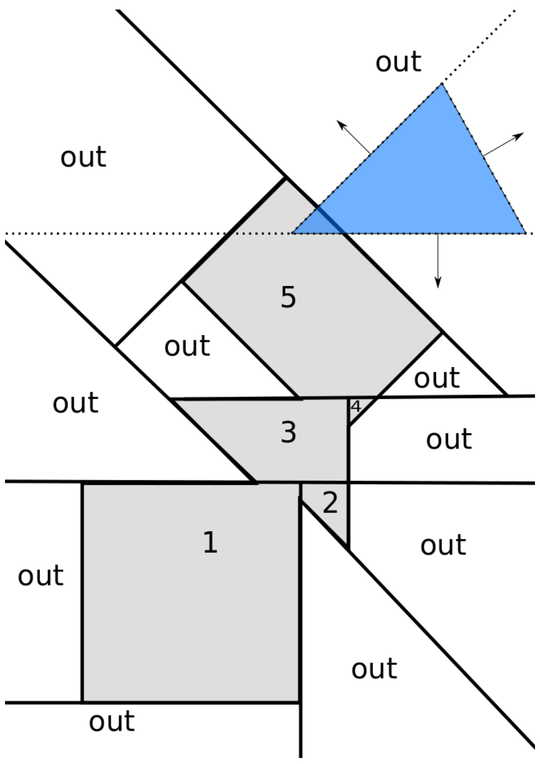


Figure 25.: Regions in two BSP-trees that are to be merged.

Firstly consider the first argument to be the grey polygon. This is against the heuristic suggested in [4]. MERGE will want to recurse to the OUT-regions of the grey polygon and potentially insert new IN-regions of the blue polygon. As the illustration shows, only one OUT-region intersects the IN-region shown in blue. All other OUT-regions are non-intersecting with this IN-region. When merging polygons, four scenarios are possible:

1. Two IN-regions overlap

2. Two OUT-regions overlap

3. An A.IN-region overlaps a B.OUT-region

4. An A.OUT-region overlaps a B.IN-region

Given UNION we are not interested in A's IN-regions. When reaching such a leaf in the tree, MERGE simply returns this leaf as a result. If the leaf in A is an OUT-region we will need to recurse on the second tree to either report infeasibility, if the two regions are non-intersecting, or to perform a leaf operation. The worst case, in running time, is when we perform leaf operations, because it means a full traversal of one tree and then the other. The best case is when we do not have to traverse the second tree. Using LP_SOLVER it is possible that the traversal of the second tree is cut short.

Now consider again figure 25. We will have to do leaf operations for all the intersecting OUT-regions. But the only place where we will do a leaf operation between an OUT-region in A and an IN-region in B is the upper right OUT-region. None of the other OUT-regions of A intersects the blue triangle. Did we swap the arguments, then consider the OUT-region created by the bottom line of the triangle. In this leaf we would have to recurse down the BSP-tree of the grey polygon and do leaf operations for each of the five IN-regions.

So to summarise, having the grey polygon as the first argument means one leaf operation between an OUT-region in A and an IN-region in B. All other leaf operations are between two OUT-regions. The overlapping OUT-regions will not change when swapping the two BSP-trees, but the number of IN-regions inside an OUT-region is now five instead of one. In general table 2 shows the scenarios when merging BSP-trees.

A	**B**	
UNION		
IN	IN	*A is IN, so no recursion in B is needed*
IN	OUT	*A is IN, so no recursion in B is needed*
OUT	OUT	*MERGE in B until infeasebility is reached or the leaf operation returns OUT*
OUT	IN	*This will never be infeasible, so the recursion continues in B until the leaf operation*
INTERSECTION		
IN	IN	*This will never be infeasible, so the recursion continues in B until the leaf operation*
IN	OUT	*MERGE recurses in B until infeasibility is reached or the leaf operation returns OUT*
OUT	OUT	*A is OUT, so no recursion in B is needed*
OUT	IN	*A is OUT, so no recursion in B is needed*

Table 2.: Scenarios when merging.

The table does not show the total number of IN or OUT leaves, but rather the consequence of two regions overlapping. Looking again at figure 25 then having the grey polygon as A means that there are five IN-OUT overlaps and one OUT-IN overlap. Swapping the polygons will also swap these cases. The table tells us that swapping can have a more drastic effect on the running time when using UNION as a best case scenario becomes a worst case scenario and vice versa. For INTERSECTION SWAP will swap a best case scenario for a scenario where early termination is possible.

5.4 SPECIAL CASE STUDIES

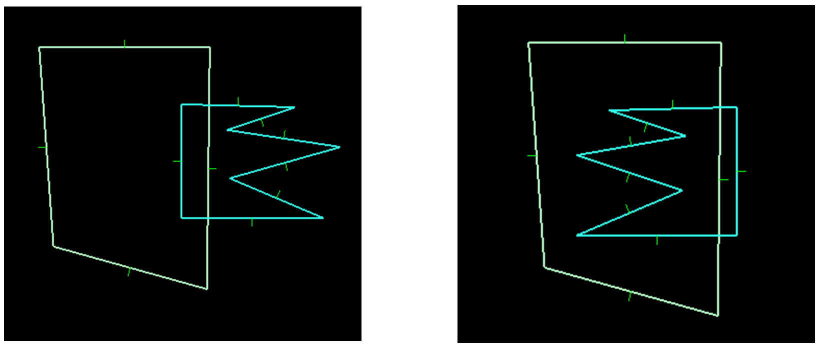
In this section we experiment with how MERGE and INC_SET_OP handles special cases. These special cases produce very different results for the size of the resulting trees as well as having a large impact on the running time of the algorithms. In the first test we examine how MERGE can sometimes produce a much larger tree than the tree created from using INC_SET_OP. In the second test we see how MERGE is much faster at reporting an empty region, than INC_SET_OP is.

5.4.1 *Storing Redundant Line Segments*

In our testing and development of the algorithms, we found that often the size of the resulting tree coming from MERGE would be much larger than the tree which INC_SET_OP produced. This would sometimes happen even if the region they described was fairly simple and sometimes even convex. This means that MERGE has a major disadvantage at certain input types. If it happens in a chain of merge operations, then the resulting tree will become larger for each merge, and the next recursion will have to traverse more nodes. We found that this occurred when one or more polygons in the scene had many *jaggies* or *spikes*.

The Scene

To give a better understanding of what happens in these cases we produced simple scenes with a convex polygon and a *jagged* polygon as seen in figure 26a and figure 26b. For both of these scenes we found the INTERSECTION, UNION and SUBTRACTION (where the convex polygon was the first argument).


(a) An image of the scene with jaggies outside the convex figure. The green lines indicate direction of normals (pointing towards exterior)

(b) An image of the scene with jaggies inside the convex figure. The green lines indicate direction of normals (pointing towards exterior)

Figure 26.

Results

Running the two scenes with all three operations gave us six different output trees for each algorithm. The table shows us that the main difference in size occurs when the jaggies are outward pointing and we are calculating the INTERSECTION, when the jaggies point inward and we take the UNION and finally when the jaggies are outward and we subtract the jagged polygon from the convex polygon. There is also a difference in size in the other three cases, albeit smaller.

	Size	**Height**		**Size**	**Height**
INTERSECTION	**Jagged**			**Rev. Jagged**	
Inc. Op.	7	8	*Inc. Op.*	10	10
Merge	9	10	*Merge*	12	10
UNION					
Inc. Op.	10	9	*Inc. Op.*	7	6
Merge	11	10	*Merge*	11	7
SUBTRACTION					
Inc. Op.	7	8	*Inc. Op.*	10	10
Merge	9	10	*Merge*	12	10

Table 3.: Dimensions for trees built.

Analysis

To explain what happens in the above scenes consider Figure 27. The partitioning lines for the jagged polygon are shown as dotted lines and the different regions are denoted either IN or OUT. The darker area is the INTERSECTION between the two polygons.

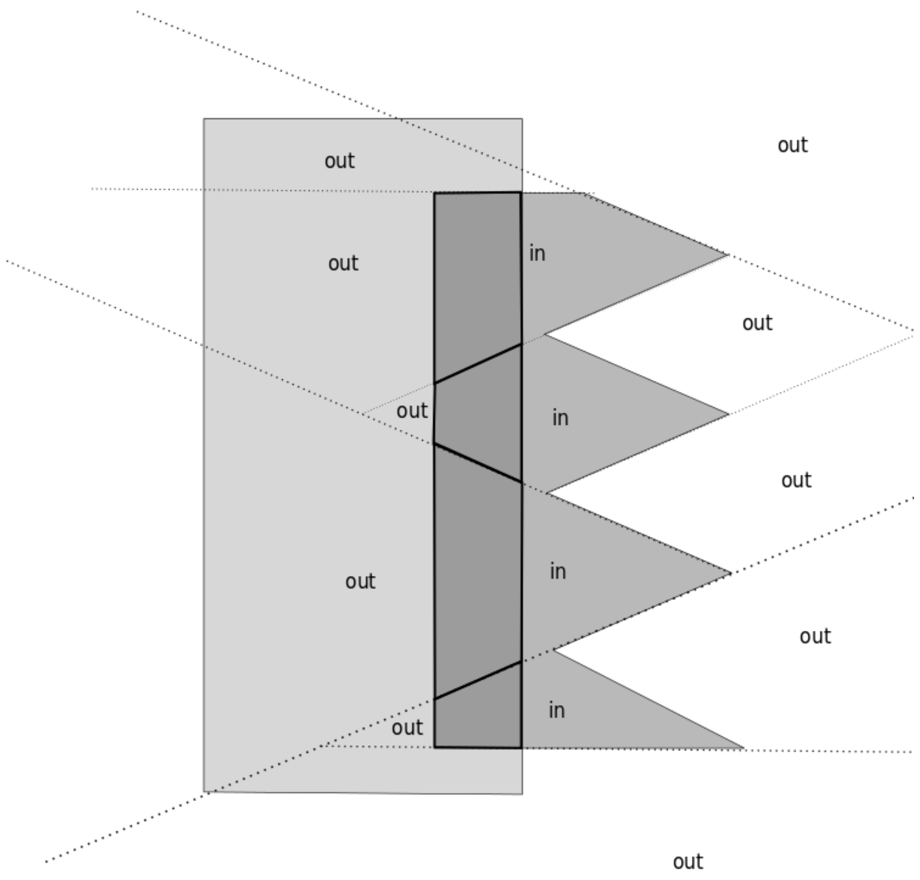


Figure 27.: Figure showing the underlying regions of the jagged figure, highlighting the problem with MERGE combined with jagged.

Notice in the figure above that the IN-regions of the jagged polygon all intersect the single IN-region of the convex polygon. This in short means that when running MERGE, it has to merge this IN-region with each of the four from the jagged polygon. This creates four new IN-regions denoted by the bolder black lines in Figure 27. Note that it does not cut the line segments. The resulting tree of this operation is therefore multiple sub trees inserted in the tree for the convex polygon - one for each new IN-region. The reason that this problem does not occur in INC_SET_OP is that it does not consider the jagged polygon as a binary space partition, but as a BREP. Because of this it is able to cut the new lines from the jagged polygon and only represent the new IN-region using only the original tree and the three extra lines needed to close off the darker IN-region. The exact same applies when we have the reversed scene, where the jaggies are inside the convex polygon and UNION is used as the operation. Here INC_SET_OP is able to represent a much smaller tree, where MERGE also represents the extra redundant in-regions.

To further show this problem we produced a scene where the jagged polygon had many jaggies and therefore would produce a very large number of IN-regions. An example of such a seen can be seen in 28.

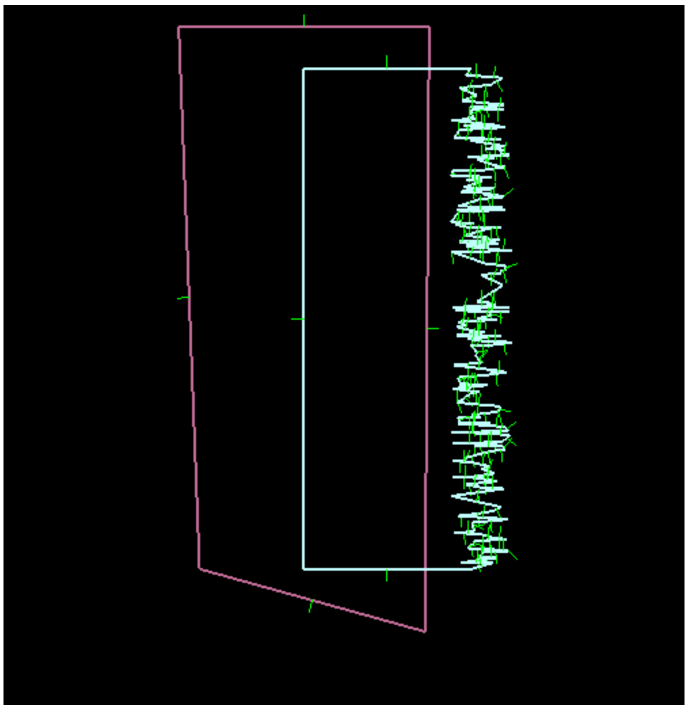


Figure 28.: An image of the scene with 200 jaggies outside the convex figure. The green lines indicate direction of normals (pointing towards exterior).

We increased the number of jaggies while using INTERSECTION and UNION of the polygons. According to table 3 we should not see a difference in UNION, but the difference in INTERSECTION should grow as the number of jaggies does.

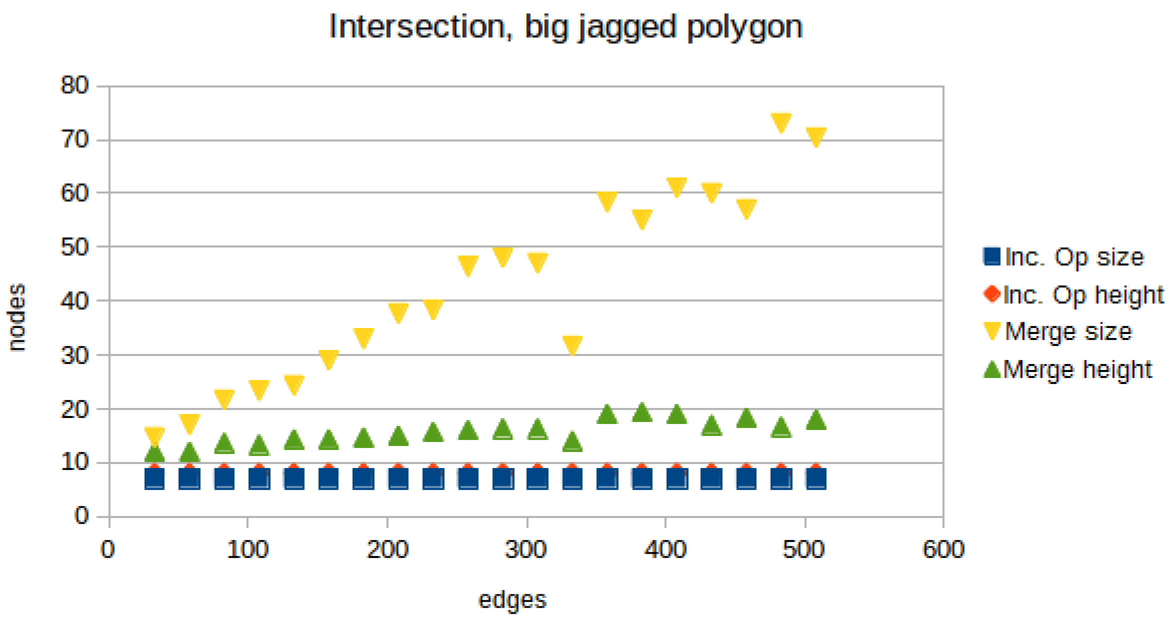


Figure 29.: Dimensions with increasing number of jaggies using INTERSECTION.

The graph for the INTERSECTION supports the above statements and the results produced from the small scenes. The size of the merged tree (yellow) grows as the number of jaggies increases, whereas the size of the old tree stays the same. The height of the two trees stays the same. The reason for the growing size is because of the many new IN-regions. The reason that the height doesn't change can be found in the structure of this new tree. Imagine, like in figure 27, that the overlapping region is split up into many small convex regions. When inserting all these regions, they must all be inserted in the subtree that is the left child of the node with the line of the convex polygon that is intersecting the jagged tree. This is because they are all on the opposite side of the normal for this line. When selecting the root of this tree there is a high probability that it will be close to the middle of the intersecting region, splitting all the new IN-regions in half. And as this probability continues a very balanced tree is produced to describe the IN-regions.

Consider again figure 27. If we are not interested in finding the INTERSECTION, but rather the UNION of the two polygons, then this region cannot be expressed as only one region. INC_SET_OP also has to include the jaggies, as it will have to build a new subtree for each of these. This is also what we see when looking at the results of this in figure 30. Here the dimensions of the two polygons are close to being the same for equal input.

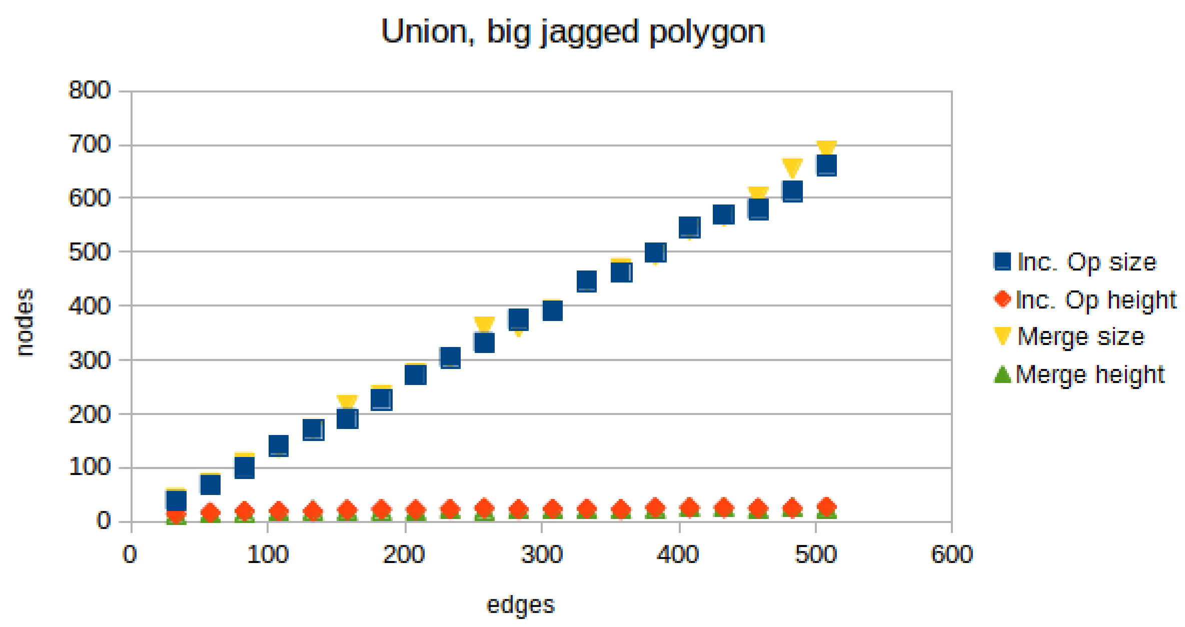


Figure 30.: Dimensions with increasing number of jaggies using UNION.

Having very jagged polygons can be very damaging on the size of the tree produced by MERGE, but this can also happen for INC_SET_OP. It all depends on the structure of the input. The problem being that, when it happens to only MERGE it is because MERGE includes a lot of redundant information to express a simple IN-region. What should be a single IN-region is actually split into many IN-regions and this can produce even more problems with chained operations. Here each extra IN- or OUT-leaf means that the algorithm has to create recursive calls all the way down to even more leaves. Depending on the operation, this could mean that it has to merge with the other tree for each IN leaf. This is not something that we found a way to solve and we believe it to be one of the reasons for why MERGE can be slower than INC_SET_OP.

5.4.2 *Empty Intersection*

In this section we will show a special strong side about MERGE, which shows in practice. We show how MERGE can compute the resulting tree in $O(n)$ time with a very small constant, where INC_SET_OP also does it in $O(n)$ time, but with a much larger constant.

The Scene

The scene consist of one large convex polygon(The shape does not matter as long as it covers all internal shapes entirely), and n smaller polygons inside this large convex polygon. The n smaller polygons are distributed randomly inside a square that is contained within the large convex polygon. The scene can be seen in figure 31.

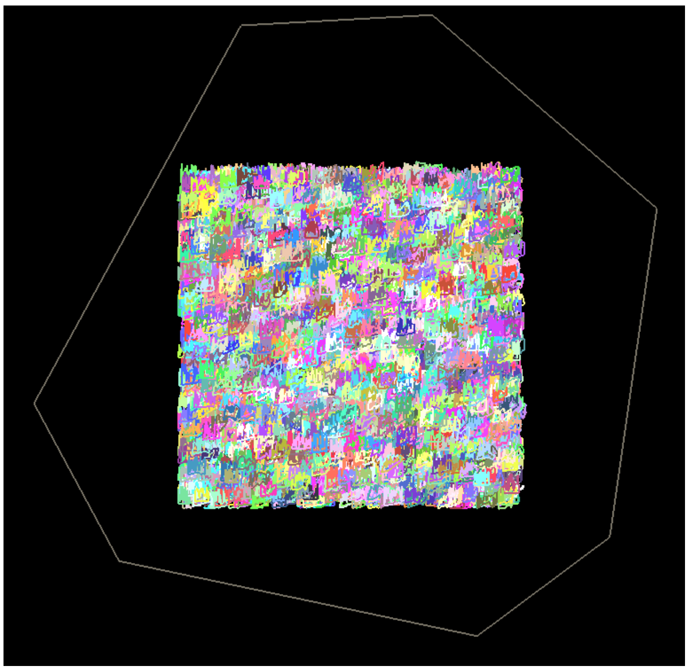


Figure 31.: Empty INTERSECTION scene.

The idea is now that we use INTERSECTION on this scene, starting with the large convex polygon and one of the smaller polygons. We then use the resulting BSP-tree from this operation and make another INTERSECTION on the next small polygon. As we can see in the scene image, not all polygons share a region. This means that we ultimately end up with an OUT-leaf, because we end up intersecting two disjoint polygons, represented with their BSP-trees. When we first have an OUT-leaf intersecting with anything, we always end up with an OUT-leaf again.

Results

The results of running this experiments with increasing n polygons inside the convex polygon can be seen in figure 32. The figure shows an logarithmic scale, with a linear regression made to INC_SET_OP.

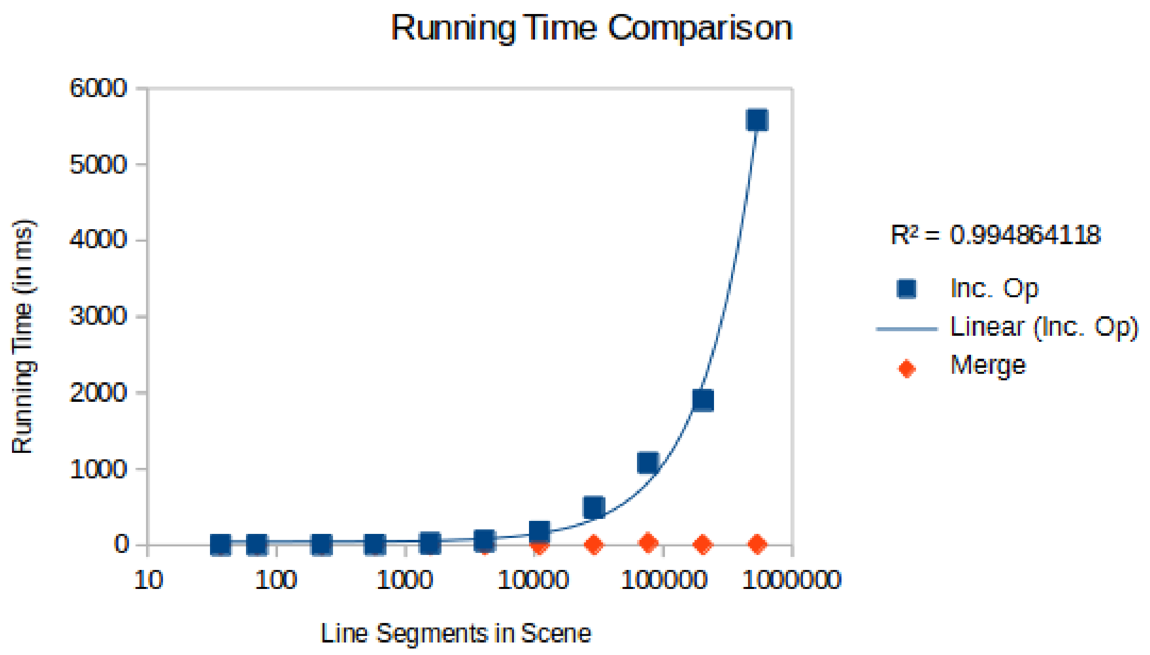


Figure 32.: Empty INTERSECTION running time, with increasing number of segments. Data is presented on a logarithmic scale.

As we can see in the graph, MERGE is very efficient compared to INC_SET_OP. MERGE is already from the very beginning very fast compared to INC_SET_OP. As we increase the input size to over 1000 line segments, the difference really becomes apparent. INC_SET_OP continues in its linear fashion and so does MERGE, but MERGE has a much lower constant resulting in a drastic performance boost in practice.

Analysis

To understand what makes MERGE so fast, we need to look at the second base case of MERGE seen in section 3.3.1. When MERGE intersects two disjoint polygons, it will end up with an OUT-leaf. When we then next time have the OUT-leaf as our first argument to MERGE, the second base case is hit. Because

A is a leaf, we swap our arguments *A* and *B*. After that we check the following:

- The operator is INTERSECTION and B is an OUT-leaf.

Because we hit this case, we simply end up returning an OUT-leaf, without having to look at the other tree at all. This leads to very fast constant work for each MERGE call, leading to the resulting linear time for all INTERSECTION operations with very small constant.

INC_SET_OP also has constant work at each operation, but here it needs to do a little more work. The first operation between the large polygon and a small polygon goes well, resulting in a BSP-tree that is larger than the original BSP-tree representing the large polygon. However when we reach a BREP that is disjoint from the polygon from last iteration, we will fail every In/Out test, resulting in a smaller tree only containing out leaves. When it then next time runs down, it will always just return out leaves, when meeting out leaves and always failing the In/Out test. The In/Out test is explained in section 3.2.4. As we can see in table 4, the first table shows the resulting trees after one operation. After more than one operation the resulting tree actually stays constant in size and depth. This supports the claim that each operation is done in constant time for both MERGE and INC_SET_OP after that point. The efficiency of MERGE comes from the fact that each of these constant time operations, in practice, is faster.

	Inc. Op.	**Merge**
Size	8	0
Depth	9	1

	Inc. Op.	**Merge**
Size	52	44
Depth	18	18

Table 4.: Tree size after (top) one iteration and (bottom) iterations thereafter.

5.5 DATA REPRESENTATION EXPERIMENTS

In the following subsections we look at different types of scenes and the results coming from the different ways of representing the data. At first we will look at a very dense scene, where we have a very large polygon made from a Sierpinski fractal.

We then make operations on smaller polygons. Next a scene with only two polygons of increasing size is investigated. We measure the running time of the two algorithms, MERGE and INC_SET_OP, as the size of the two polygons increase. Lastly we look at the CSG way of building a scene from simple elements. In this scene a lot of small polygons will form a more complex polygon using nested UNION operations.

5.5.1 *Dense Scene*

As already mentioned, this experiment with few operations. A large and very dense polygon is used as the first BSP-tree so that the scene will already be split into many small cells.

The Scene

The scene consist of a large polygon made from a septagon with a Sierpinski arrowhead curve of order 5 on each edge. This polygon is a large closed polygon which consists of many small regions. The two small polygons are either convex or simple polygons each with between 10 and 20 edges. In this experiment we use INTERSECTION and UNION as operations. The Sierpinski is always the BSP-tree argument given to INC_SET_OP. In MERGE we run the tests with and without swapping the arguments.

An example of the scene can be seen in figure 33.

Results

The results of running the experiment can be found in table 5. MERGE always produces a smaller tree, when running INTERSECTION, but INC_SET_OP is almost always faster than merging. It is also always the case that MERGE is faster when swapping the arguments, but UNION with MERGE produces smaller trees when not swapping the arguments.

Analysis

Because both INC_SET_OP and MERGE has a BSP-tree representation of the Sierpinski polygon, both algorithms will have problems with many small regions as described in section 5.4.1. This means that the size of the trees should be at about the same size in the INTERSECTION case, but because of COLLAPSE and the

	Using Swap		
INTERSECTION	**simple + convex**	**2 simple**	**2 convex**
edges	1741	1735	1736
Inc. Op. size	135	89.67	543.67
height	25	22.67	21.33
time	2.33	18.33	2.33
Merge size	63.33	31.33	475
height	38	12.33	36.33
time	14.33	4.33	105
UNION			
edges	1740	1739	1738
Inc. Op. size	1292	1587	1190
height	22	25	23
time	3	10.67	2.33
Merge size	1322	1806	1208
height	29	30	31
time	164	168.67	115.67
	Not using Swap		
INTERSECTION	**simple + convex**	**2 simple**	**2 convex**
edges	1734	1733	1736
Inc. Op. size	206.67	170.33	547.67
height	22.33	22	23
time	1.33	2.67	1.33
Merge size	143.67	154.33	529
height	21.33	26	25.33
time	108	113.67	313.33
UNION			
edges	1733	1735	1736
Inc. Op. size	1249	1607	1151
height	24	25	22
time	18	1.33	2
Merge size	1175	1613	1112
height	26	28	24
time	275.33	238	276

Table 5.: Results from merging larger polygons

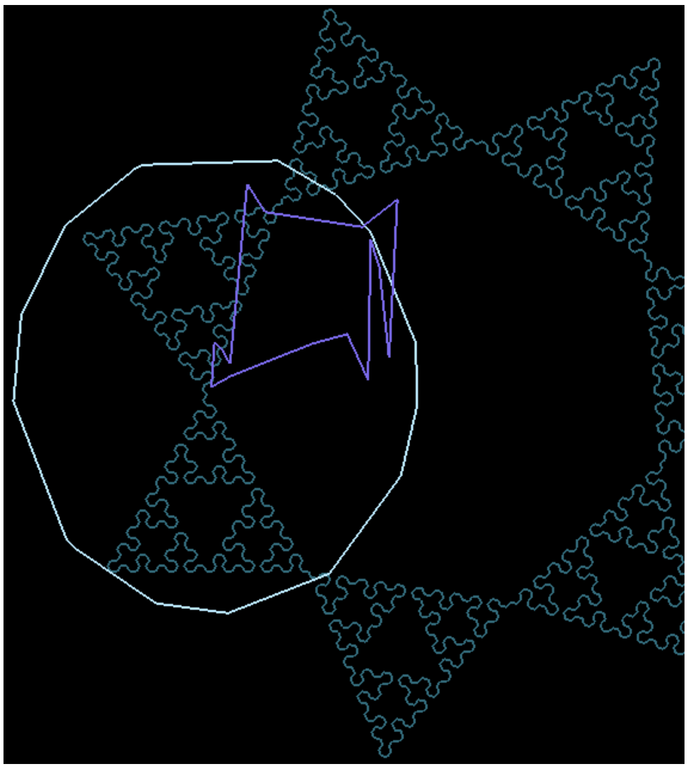


Figure 33.: Dense scene

linear program solver in MERGE, this algorithm will produce smaller trees as seen in the results.

Because of the many regions, INC_SET_OP will split the other polygons up in constant amount of line segments for each cell, which it can then build in constant time. This is what makes INC_SET_OP run in linear time and even faster than MERGE because it has a smaller constant.

In this particular scene it is valuable to swap the input parameters for MERGE. It affects both the running time, but also the tree size when inserting the large Sierpinski polygon into the smaller polygons. The reason for this, must be that we get a better structure of the scene, with better overlapping regions as described in table 2. If we also choose to swap the arguments for INC_SET_OP, then it would be very slow and actually slower than MERGE. INC_SET_OP would have to build the entire Sierpinski tree (UNION case) or at least a part of it(INTERSECTION) which contains enough line segments for the building process to not take constant time.

5.5.2 *Two Random Polygons*

In this experiment we are looking at two random polygons of increasing size. We are here focusing on the running time of the two algorithms.

The Scene

In the scene we have two random polygons of increasing size. The polygons are created in the fashion described in section 5.2.2. The operation used was INTERSECTION. As the amount of edges for each polygon is increased, so is the bounding box for the polygon in order to lessen the amount of pseudo parallel lines.

Results

The result of running this experiment with increasing size random polygons can be found in figure 34.

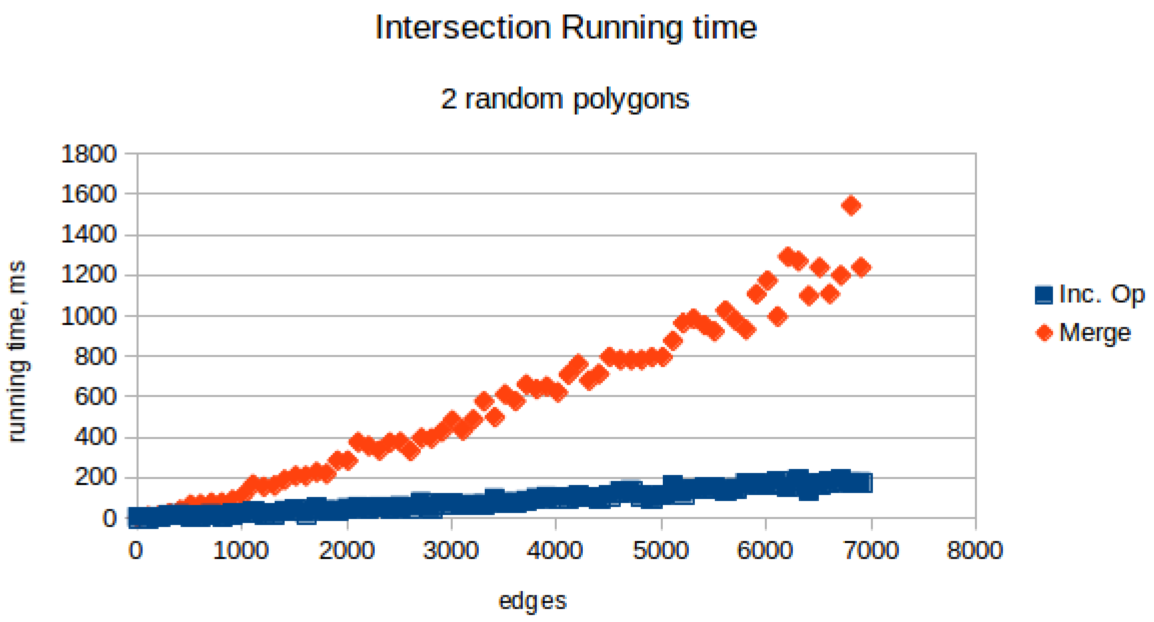


Figure 34.: Graph showing running time of running both algorithms with two random polygons of increasing size with INTERSECTION.

Both algorithms here run in a linear time, with MERGE having the biggest constant and thereby being the slower one.

Analysis

We have many small regions, because of the extremely thin triangles of the random polygons and this is something INC_SET_OP can benefit from. This will spread the line segments out into many regions, making a single region containing constant line segments. This will make INC_SET_OP run in constant time, an as we can see even faster than MERGE. This constant can come from mainly the simpleness of INC_SET_OP. In INC_SET_OP we basically just have an extension to BUILD_BSP with an In/Out test. MERGE has a lot more subroutines which, with a suboptimal implementation, could lead to a larger constant.

5.5.3 *Modular Scene*

In the experiments in this section, we look at many operations, but with small polygons. We examine the creation of a complex polygon by using many smaller polygons. We also look at another scene where we have one large convex polygon that has few edges where we subtract a lot of small polygons, creating a resulting polygon like the original big polygon, just with many holes in it. We do not look at INTERSECTION here, because it is hard to create a scene where the many polygons will all have an intersecting area. Because SUBTRACTION is implemented as complementing the second tree and then INTERSECTION, one could argue that the SUBTRACTION scene also represents INTERSECTION. We also investigate how swapping of the arguments can affect size and running time of MERGE.

The Scene

The first scene that we have is the scene where we UNION many small polygons. We start by having one simply polygon down in the left corner of our scene, and then we simply add a new polygon by creating a new random simple polygon and then translating it to its new position. The new position is chosen so it overlaps the top right corner of the current rightmost polygon. In this way we create a closed chain of polygons perfect for UNION.

An example of the scene can be seen in figure 35.

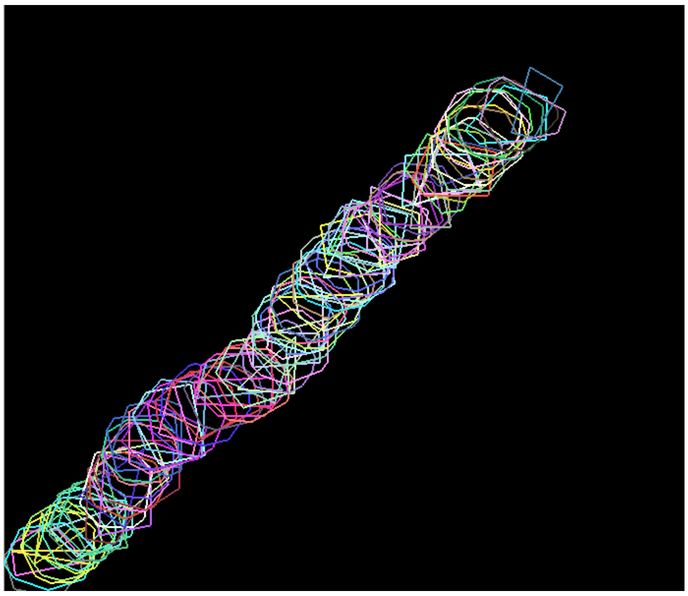


Figure 35.: Modular scene with UNION

The second scene we have is the scene used with SUBTRACTION. In this scene we have one large convex polygon, and an increasing number of smaller polygons. These polygons are places randomly around the scene. If a polygon is places completely outside of the large convex polygon, then it simply wont contribute to an altering of the resulting polygon.

However, some care does need to be exercised when placing these random polygons. The placement of the polygons could lead to a disjoint resulting polygon, because a number of polygons could cut chunks of the original convex polygon away. To avoid this issue, we made the convex polygon large, the smaller polygons small and kept the amount of small polygons within some limit. We also repeatedly manually looked at the scene to ensure that we had a valid scene.

An example of the modular scene with SUBTRACTION can be seen in figure 36.

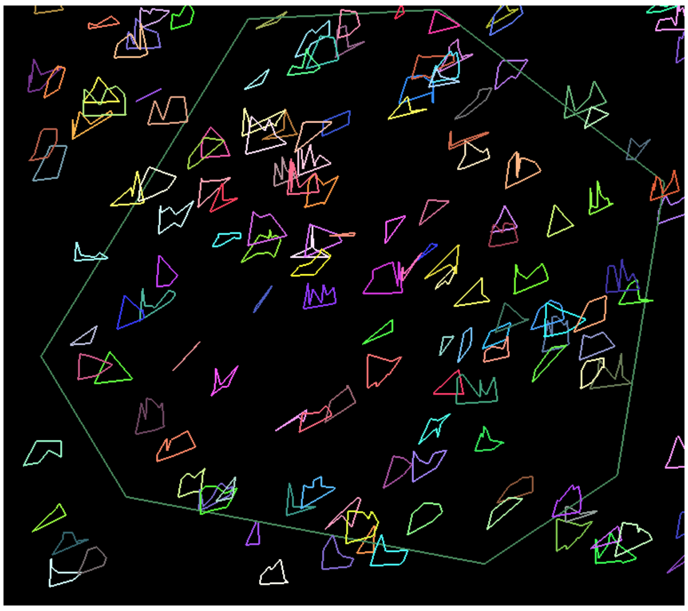


Figure 36.: Modular scene with SUBTRACTION.

Results

We ran the two experiments with increasing sizes of the the scene by increasing the amount of polygons. We also ran the experiments with and without swapping the input arguments to MERGE. In the UNION scene this mean that we either inserted a new small polygon into the chain or reverse. Not that the swapping of arguments in merge takes place after $A \ -^* \ B$ is transformed to $A \cap^* B$ meaning that swapping the arguments does not change the outcome of the algorithm.

The results of running these experiments can be found in table 6.

SUBTRACTION	**Using swap**		
Edges	646	979	1309
Inc. Op. size	541.33	1091	1404.67
height	32.33	34	35.67
time	47.33	81.33	109
Merge size	2572	5094	7613
height	40.33	42.67	48
time	6338.33	20313.7	42755.7
UNION			
Edges	666	967	1300
Inc. Op. size	702.33	1224.67	1598.67
height	87.33	114.67	157.33
time	226	421	730.33
Merge size	1275.67	1777.33	2502.67
height	55.67	47.67	51.67
time	3602	8478.33	15883.7
SUBTRACTION	**Not using swap**		
Edges	646	979	1309
Inc. Op. size	544.33	1096.67	1402
height	31.67	34.67	35.33
time	57.33	96.33	115
Merge size	586.67	1133.67	1478.33
height	35	40.67	38.67
time	2732	7498	13548.3
UNION			
Edges	666	967	1300
Inc. Op. size	711.67	1216.33	1621
height	88.67	115.33	150
time	218.33	521	715.67
Merge size	836.67	1138.67	1566.67
height	54.33	60.33	96.67
time	3695.67	7753	15050.3

Table 6.: Results from merging many smaller polygons.

The results from this experiment show that swapping the input arguments can have a big effect on both the running time, but also the resulting tree size when using MERGE. For the SUBTRACTION scene it affects both the running time and the size of the tree. In the UNION scene only the tree size is affected. The results also show that, when not using swap, MERGE produces a resulting tree at almost the same size as INC_SET_OP.

The results does however also show that INC_SET_OP is always faster than MERGE at running these experiments.

Analysis

Swapping the arguments in the SUBTRACTION case is clearly not beneficial. The explanation can be found in table 2. When we subtract a small polygon it actually is the polygon with inverse normals intersected with the larger polygon. This means that the small polygon represents IN-regions in the entire scene except inside the polygon. In the SUBTRACTION scene the IN-region INTERSECTION with OUT-regions are all the cases where other holes exists in the complementary exterior of the current hole. In all these overlapping regions the algorithm either terminates from infeasibility or continues to a leaf operation. If we however swap the input arguments we get the reversed case. Here we will have a lot of OUT-regions intersecting the big IN-regions from the new polygon. This is the best case when the operation is INTERSECTION.

As for the UNION scene; when not swapping we insert the IN-region of the small polygon a few places in the BSP-tree of the larger chained polygon. If we swap we have to insert the IN-regions of the large chained polygon that overlaps the OUT leaves of the new small polygon into these OUT-leaves. That the running time is about the same when swapping and not swapping can come from the fact that there might be around the same amount of IN-regions overlapping OUT-regions, swapped or not. So the small polygon's OUT-regions overlap about as many IN-regions from the chained polygon as the small polygons one IN-region overlaps with the chained polygons OUT-regions.

5.6 SIZE AND OPERATIONS EXPERIMENTS

In this section we examine how the algorithms behave on a random number of randomly sized polygons as well as a large number of horizontal and vertical line segments. Note that our random polygons are made using our random generator and that as the size of such a polygon increases, so does the probability of having almost vertical lines and very narrow triangles as IN-cells. In the first experiment we compare the algorithms on n random polygons and compare the running time. The next experiment will be merging an increasing number of randomly sized squares and finally we examine how INC_SET_OP behaves when having to build all of a worst case BREP.

5.6.1 *Random Number of Operations and Random Size*

This test is a test of how the algorithms behave on increasingly large random input data. We want to examine how this affects the constant in the asymptotically linear running time of MERGE, with both SWAP enabled and disabled. Another reason for this test is to see if the running time of INC_SET_OP remains linear on random data.

The Scene

We create the same scene for both the INTERSECTION and UNION operations. With the same origin n polygons are generated with a random number of edges. As each of these polygons have the same origin and they all have the same bound on their size, then an INTERSECTION will with high probability exist. For UNION they will not be disjoint in the space, so it makes sense to UNION them. Below are two examples of how the scenes could look for these tests. As for SUBTRACTION we simply generate a large convex polygon and many small polygons uniformly distributed in the space. This has the potential to create illegal output, a separated polygon, so we keep the number of small polygons low, so that it will not happen.

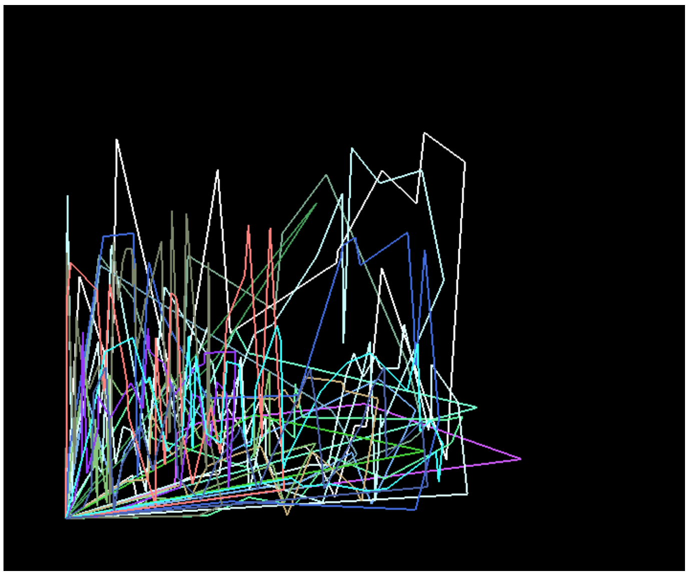


Figure 37.: A scene of size 387 where INTERSECTION is applied.

Notice how it, in the above example, is hard to see any common area for the polygons. But even if it does not exist, then it is still valid input as the result off all operations would be an OUT-leaf.

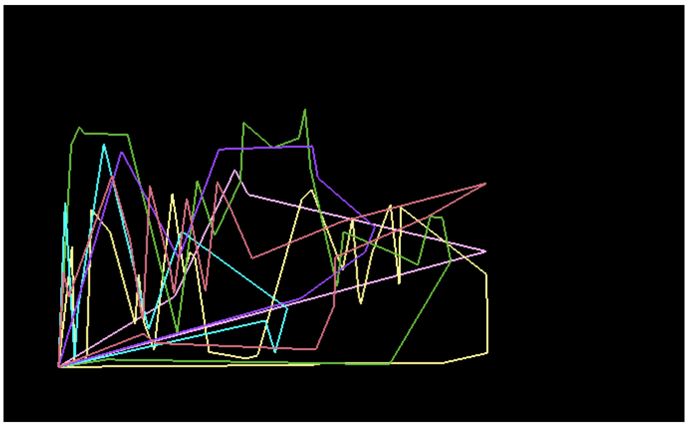


Figure 38.: A scene of size 103 where UNION is applied.

Results

We measured both the running time of the algorithms as well as the dimensions of the resulting tree. These results can be seen in figure 39, 40 and 41.

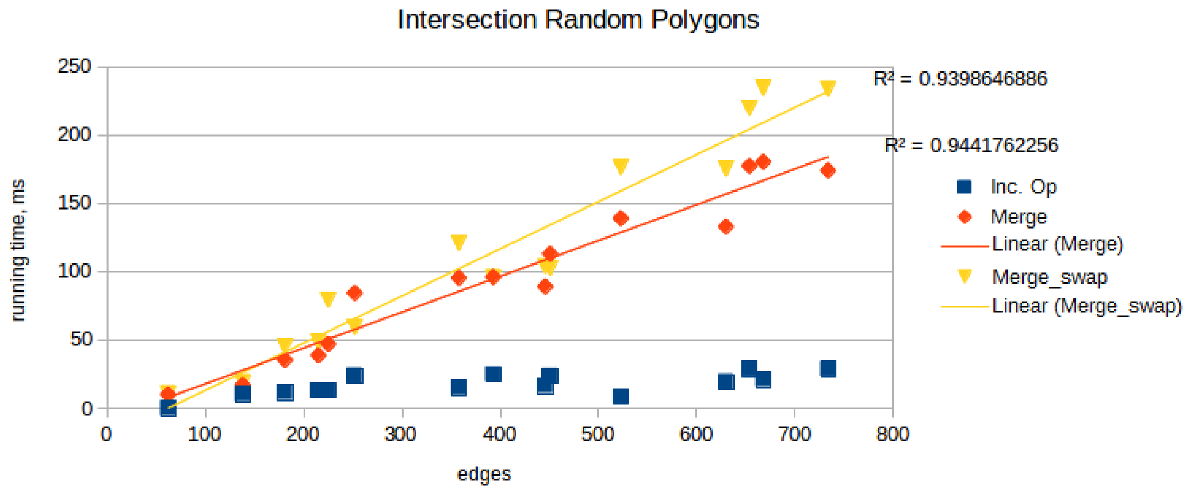


Figure 39.: Running time

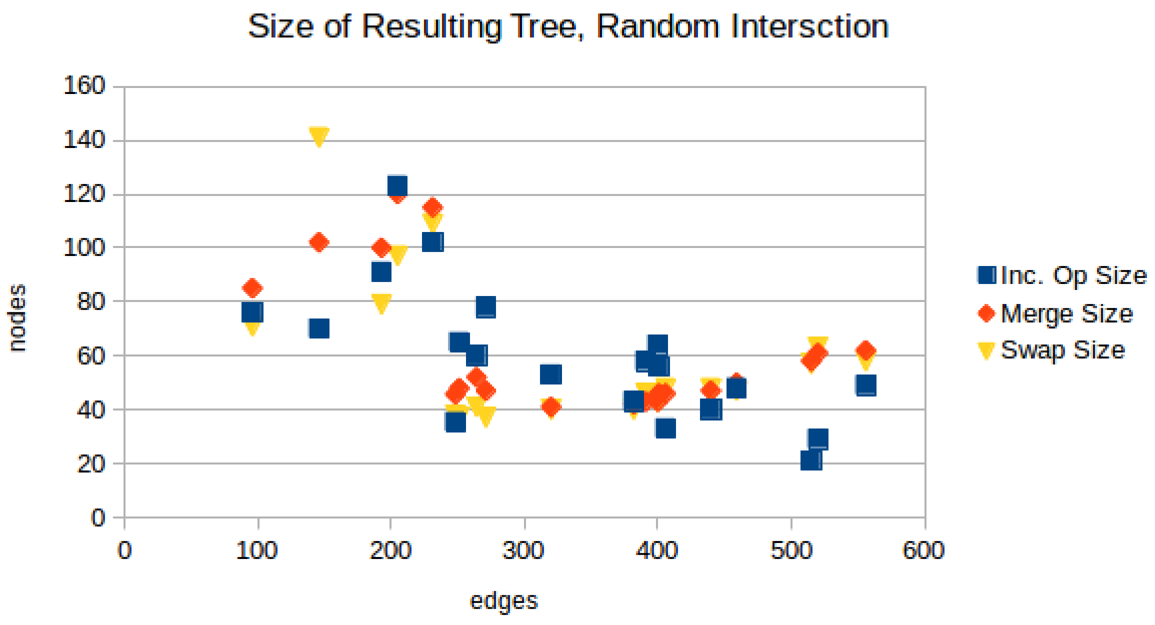


Figure 40.: Size

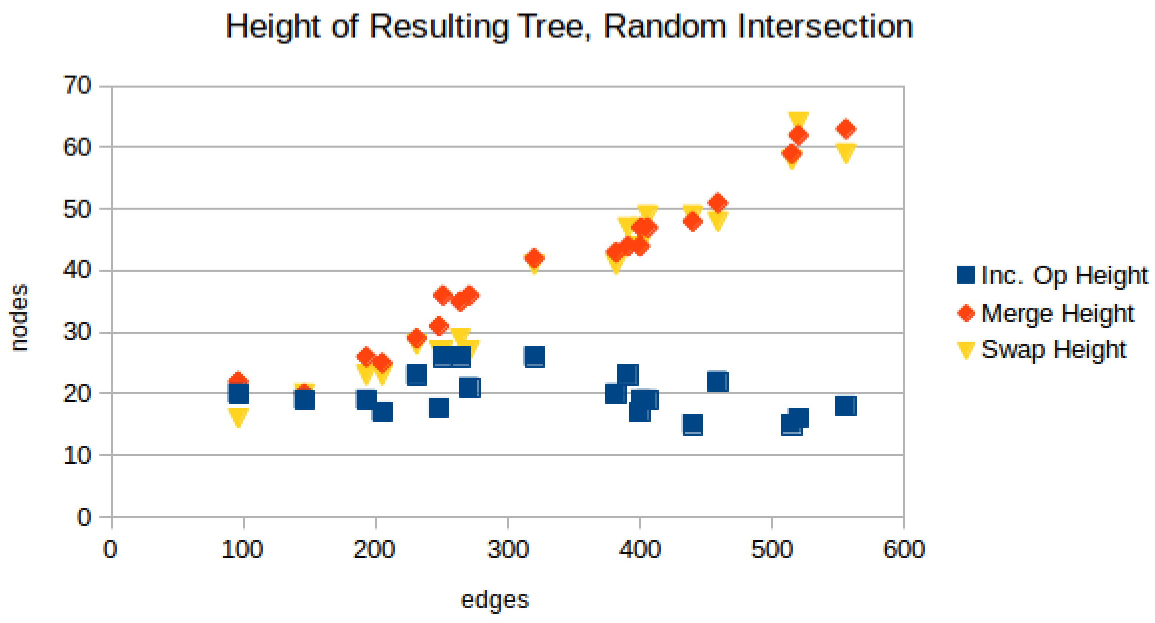


Figure 41.: Height

The running time for MERGE is clearly linear with and without swapping the arguments. The running time for INC_SET_OP is low, compared to MERGE and it seems to be running in linear time as well. The size of the tree is decreasing with more edges while the height of the MERGE result grows linearly in the amount of lines. The results for when running with UNION can be found in figure 42, 43 and 44.

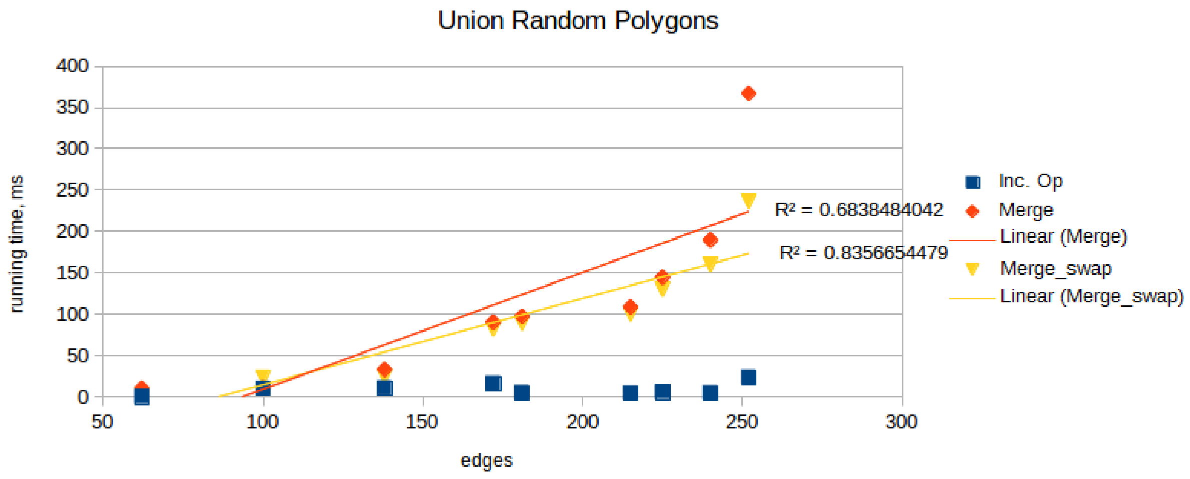


Figure 42.: Running time

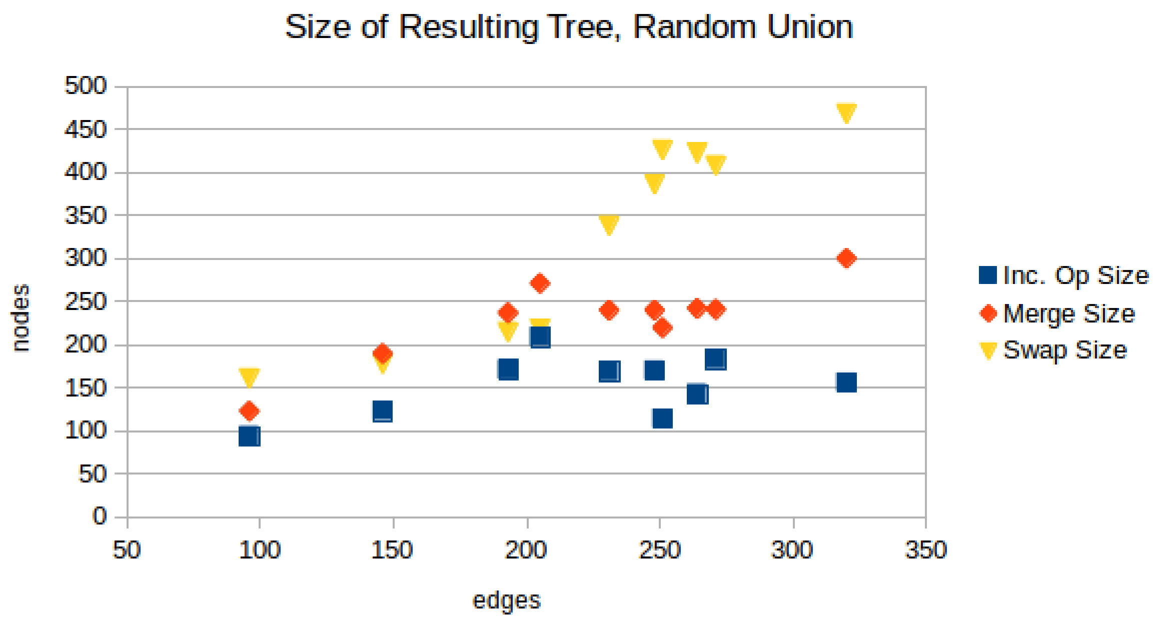


Figure 43.: Size

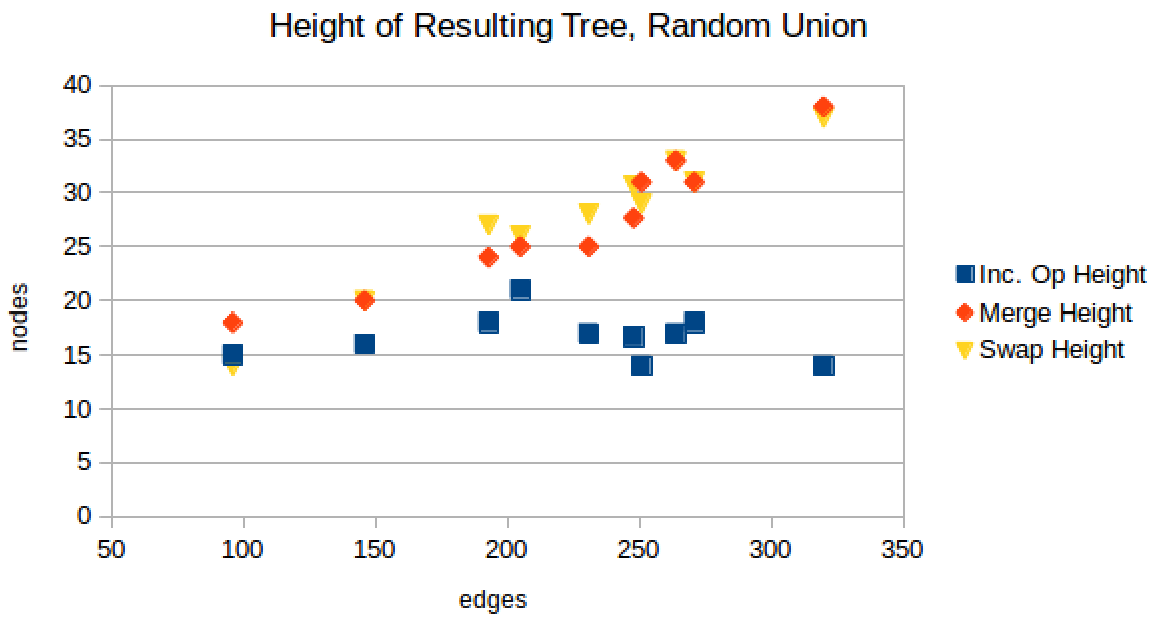


Figure 44.: Height

The running time for MERGE when the operation is UNION seems to be faster when swapping the arguments. All running times, like in the INTERSECTION case, looks to be linear. There is a noticeable increase in running time for the last measurement, which is visible in both algorithms. As the tests are run on the same input, then the structure of the scene could simply result in a larger running time in general.

The results for when the operation is SUBTRACTION can be found in Appendix B.

Analysis

Looking at the running time graph for the INTERSECTION case, it is clear that all algorithms run in linear time. The difference in running time between swapping and not is constant and the extra time spent could be because of MERGE not having as many early termination instances in the feasibility case. According to the jagged special case, INC_SET_OP should perform much better in size and running time, if it can build each IN-region in constant time. This is especially visible in the height of the tree, which doesn't increase as we insert more and more random polygons into the scene.

When finding the UNION of the random polygons it seems that swapping makes the MERGE slower, which is probably caused by a better ratio of best to worst cases than when swapping - see the section about swapping 5.3.3. Again the jagged scenario causes the tress to become bigger for MERGE than those of INC_SET_OP. It is interesting to note that the size of the tree when swapping is much larger than when not swapping, which we believe to be because the algorithm, when swapping, has to insert the many jaggies outside of the smaller random polygon, whereas when not swapping the number of regions that are actually outside the bigger polygon will be smaller if not non existing.

5.6.2 *Parallel Lines in Form of Many Squares*

In this section we will experiment with many parallel lines. The lines will be inserted only one square at the time, so the input for any operation will be the previous result and a square.

The Scene

To create the scenes for the UNION operation, we first create a big square and then, with some offset, n randomly sized squares. An example can be seen in Figure 45. This is also the scene used for SUBTRACTION.

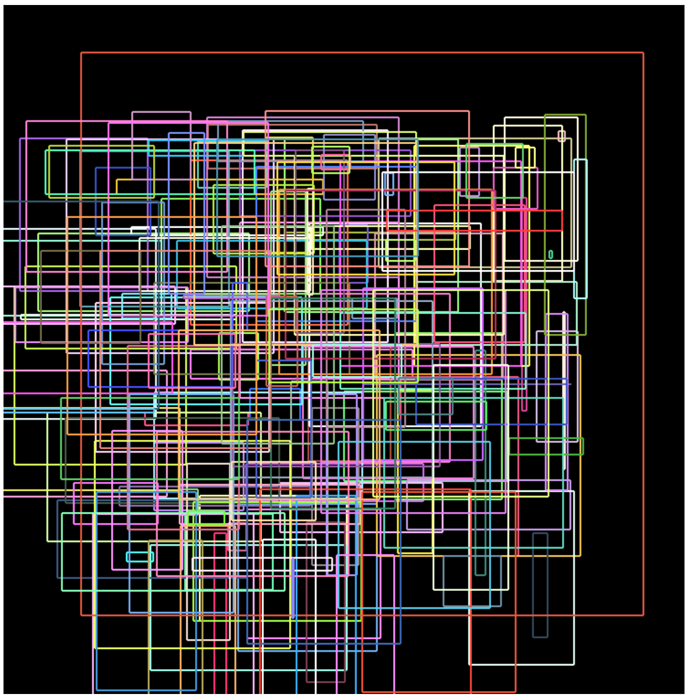


Figure 45.: UNION Square Scene.

For the INTERSECTION we created a large containing square as well, but we did not offset the n-squares inside. This means that the intersection will be the smallest square visible in the lower left corner in Figure 46.

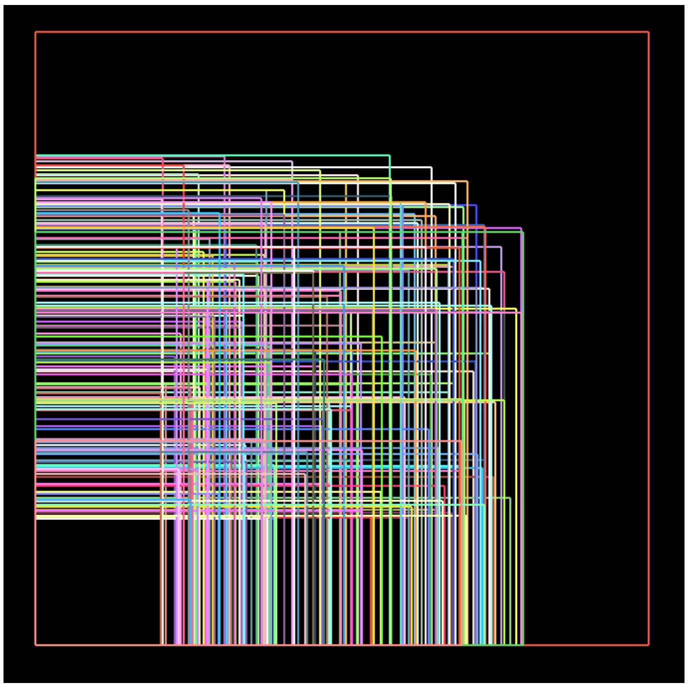


Figure 46.: INTERSECTION square scene.

Results

The results for the INTERSECTION of squares can be found in Figure 47. The tendency line for MERGE is, when not swapping, linear. But when swapping the polygons the tendency line resembles a power function more.

In Figure 48 are the results from UNION. It is clear that here, MERGE is linear both with and without swapping. The constant factor is, however, much lower when not swapping.

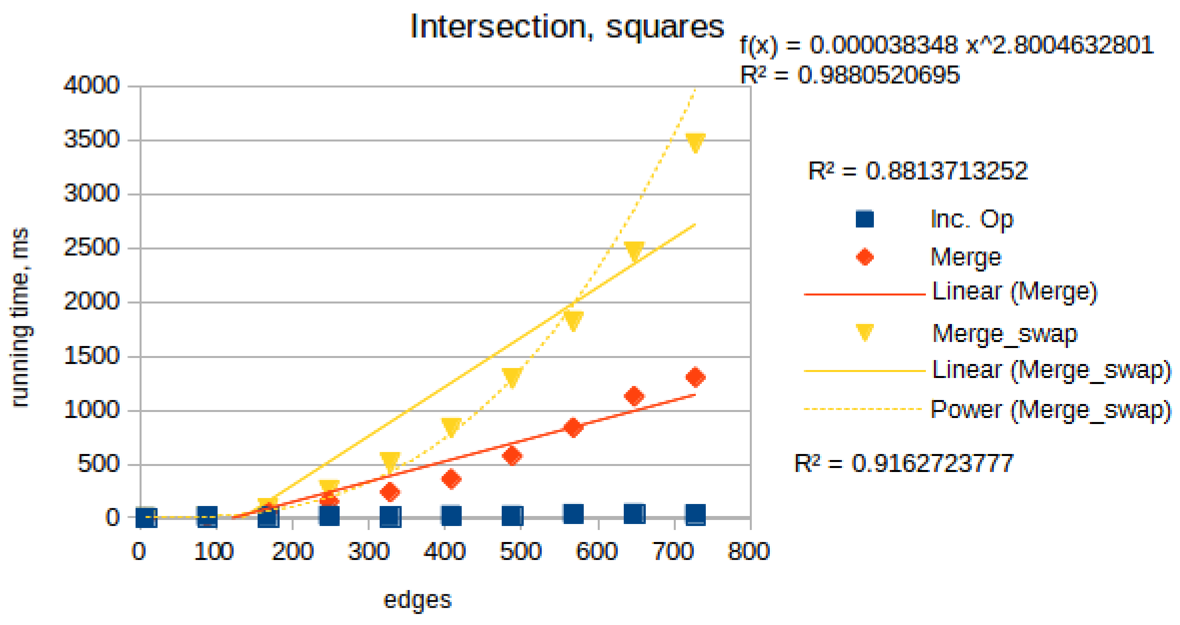


Figure 47.: Running time of intersecting squares scene with increasing edges. Shown with and without swapping the inputs

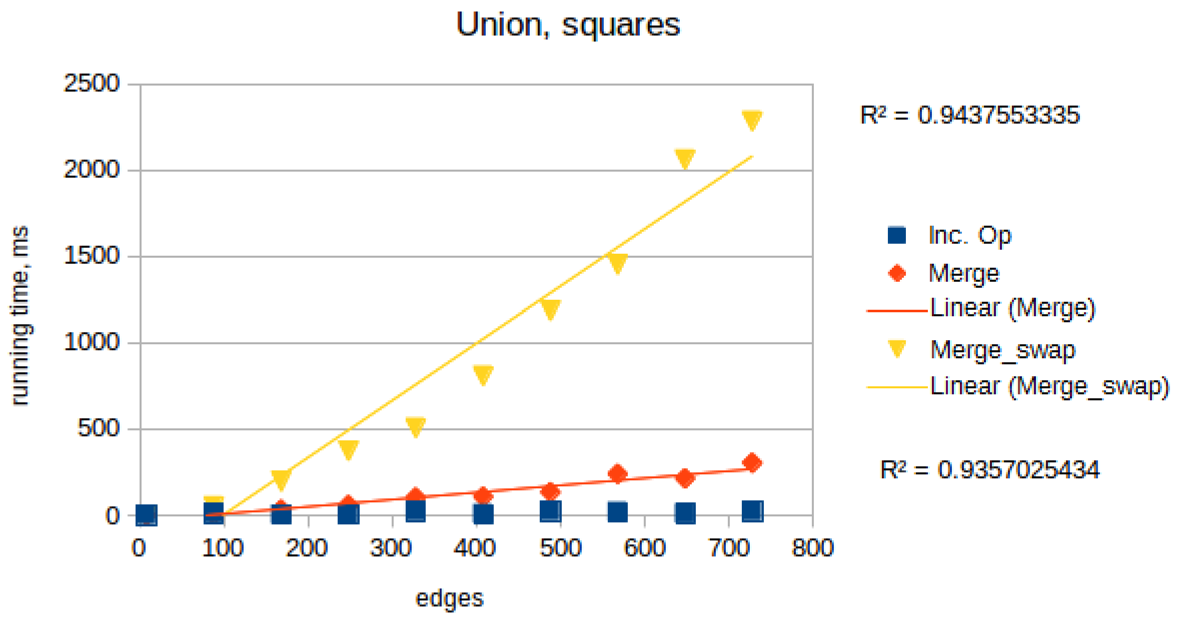


Figure 48.: Running time of using UNION on the squares scene with increasing edges. Shown with and without swapping the inputs.

The results for SUBTRACTION are similar to those of UNION and can be found in Appendix B.

Analysis

The first thing to notice in the INTERSECTION scene is the difference in running time for MERGE. Swapping or not makes the difference between $O(n)$ and $O(n^2)$ running time. We expect that the reason for the $O(n^2)$ running time is that when having an intersection of squares, then the BSP-tree for this scene will resemble a grid. Having the grid as the right argument in an INTERSECTION means that MERGE has to do leaf operations for each of the IN-regions in the pseudo grid that are within the IN-region of the new square. Not having the BSP-trees swapped for the operations has the effect that only the one IN-region in the pseudo grid performs leaf operations with the one IN-region in the new square. This is in coherence with our explanation of the dangers of swapping in 5.3.3 and the theory for the worst case running time for MERGE. In this scene INC_SET_OP seems to be running in linear time. This is because the only time INC_SET_OP will build anything is when the new square is inserted within the current intersection. In any other case the INC_SET_OP will cut the new square out and not spend any time building.

In the scene where we wish to find the UNION of the many squares, swapping also yields a slower running time. Unlike the INTERSECTION scene it does not result in a squared running time. To explain this, consider having performed n operations and a new square is to be inserted. If this square is the first argument, meaning that we have swapped, then for each of it's OUT-regions it will have to search for IN-regions in the more complex tree. This tree does not resemble a grid like in the INTERSECTION case, as only convex squares are added on the border of the current UNION of polygons, which is why we don't have worst case running time. There are still a lot of unnecessary leaf operations between OUT-regions and IN-regions. If we had not swapped these would be overlapping as IN-regions and OUT-regions which requires no further recursion in the second tree.

5.6.3 *Operations on Parallel Lines*

In these experiments we wanted to test the behavior of the algorithms when the input was not restricted to be polygons, but simply vertical or horizontal lines. In the first test the BREPs are n vertical lines and n horizontal lines. In the second test we created a convex polygon that surrounds $\frac{n}{2}$ vertical and $\frac{n}{2}$ horizontal lines.

The Scene

The first scene (figure 49) was generated simply by creating the lines. Strictly speaking none of the BREPs are polygons, but the space is still binary partitioned. The green lines in the scene are the normals of the line and each spacing between lines where there is an orthogonal green line is an OUT-region.

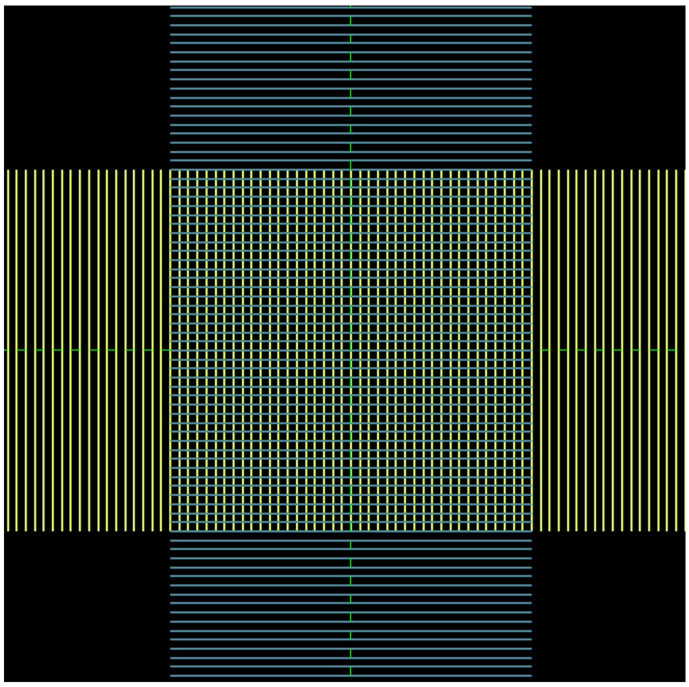


Figure 49.: n vertical lines intersected with n horizontal lines.

The second scene was the same as above but the lines were put in the same BREP and then a convex polygon was created as a bounding polygon for the intersections of the lines. This scene is only run on INC_SET_OP.

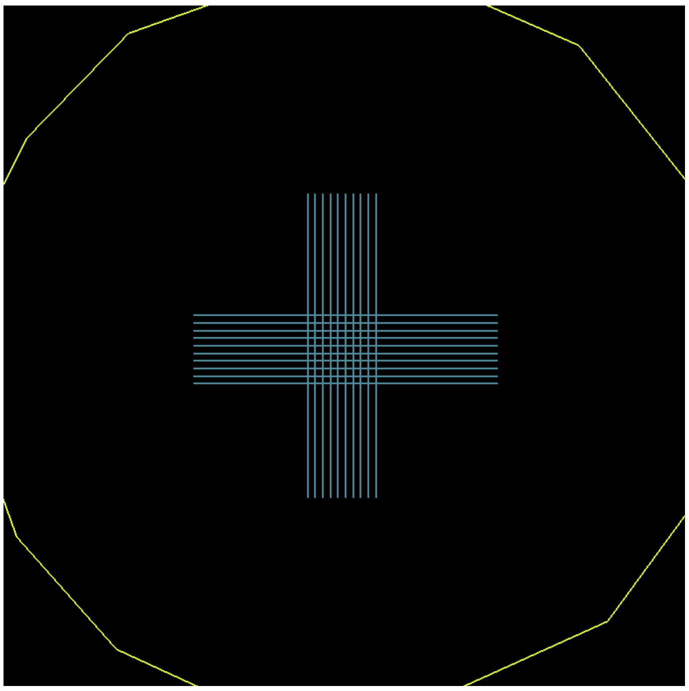


Figure 50.: A convex polygon intersected with vertical lines and horizontal lines.

Results

In figure 51 the running time of the first scene is displayed. MERGE runs in $O(n^2)$ time and INC_SET_OP runs in linear time. There seems to be a bump in the graph around input sizes of 1000 line segment, but as the rest of the point seem to lie almost perfectly on the line, then we assume that these outlier points must be measurement uncertainty or the computer having to do extra jobs.

In figure 52 and 53 two very different results are shown. Both are for the second scene, but in the first we, as in all previous tests, used auto partitioning when selecting the splitting line in BUILD_BSP. In the second graph we simply selected the line from the head of the list of line segments. We interpret the running times as being $O(n^2)$ and $O(n^3)$ respectively.

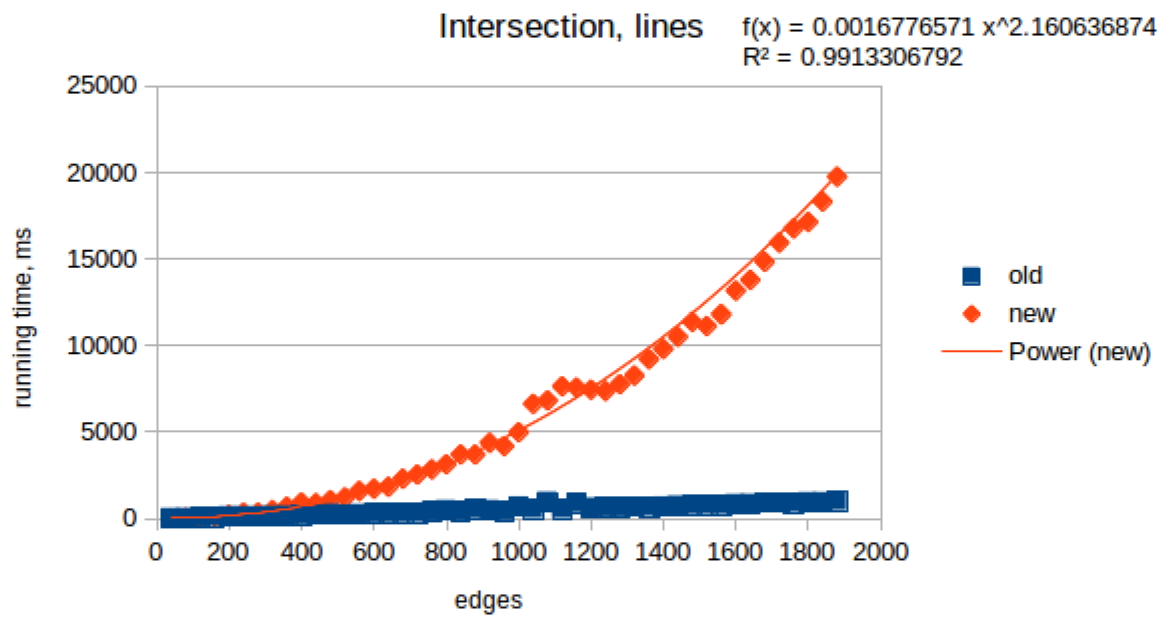


Figure 51.: Running time of line scene with increasing edges. Shown with and without swapping the inputs.

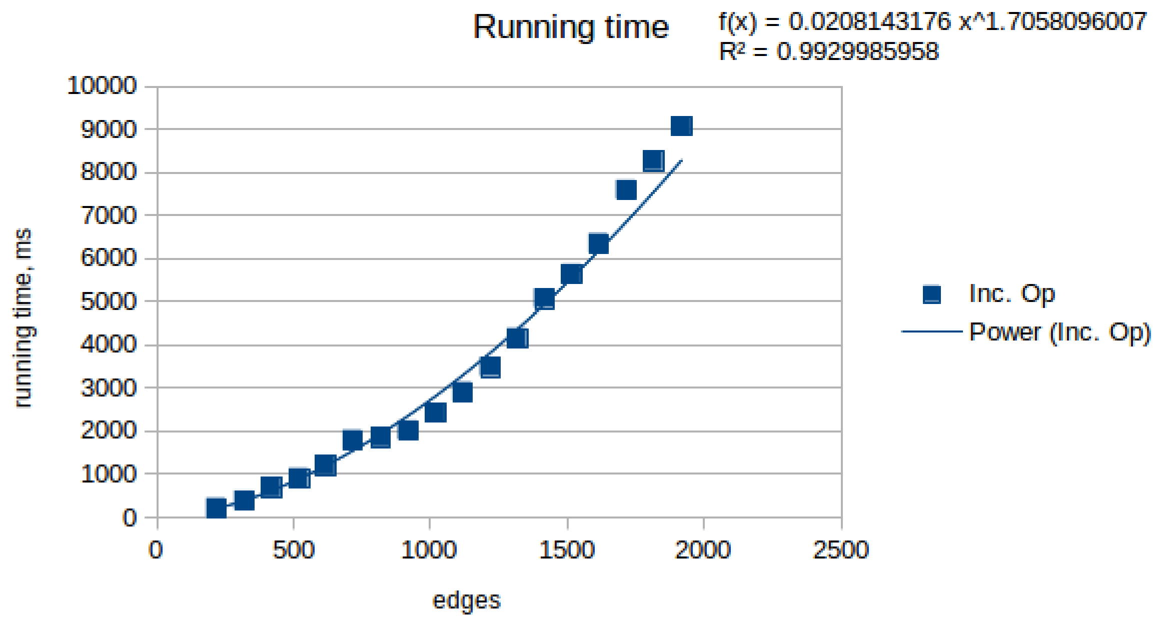


Figure 52.: Running time when using auto partition.

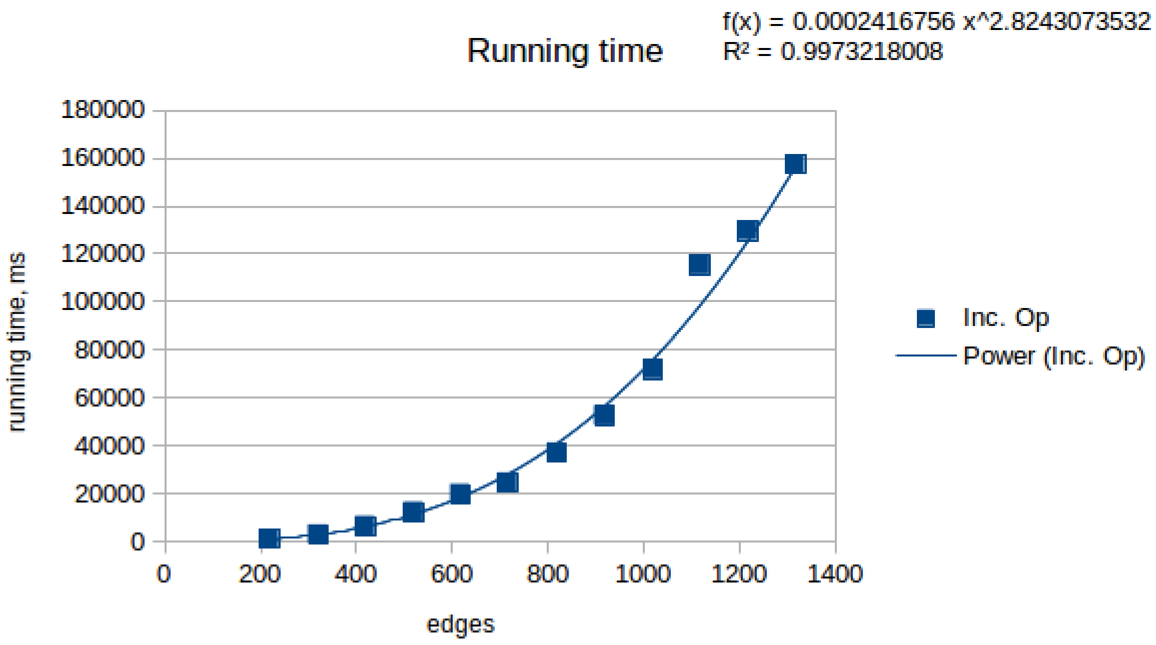


Figure 53.: Running time when using naive splitting line selection.

Analysis

The scene in figure 49 is the exact case described as worst case scenario in [4] for MERGE, where the worst case running time is at least $\Omega(n^2)$. So it makes sense that the running time is $O(n^2)$. As for INC_SET_OP, it seems to be running in linear time. Leaving out MERGE and running this scene for much larger input also suggests a linear running time for INC_SET_OP, see figure 56 in Appendix B. This suggests that for each of the n horizontal bars created, it builds the intersecting vertical bars in constant time. Figure 52 and 53 does however show that INC_SET_OP not always runs in linear time. If the scene forces it to build more than a constant amount of lines in a leaf, then the running time is bound by this building time. We know that BUILD_BSP spends $O(n \cdot \#total\_cuts)$ time when building a BSP-tree. In these cases the total number of cuts is dependent on the selection of splitting lines. If the line segments, in figure 50, are selected in a bad way, then n^2 cuts will be created giving a total running time of $O(n^3)$ for the building process alone. If the line segments however are chosen in random order, only expected n cuts are made and thereby $O(n^2)$ running time, explained the results from figure 52 and 53.

6 CONCLUSION

In this thesis we found that it is possible for INC_SET_OP to run in linear time. With our implementations it is even faster than MERGE, if the input structure allows it. During our experiments we found that the running time of INC_SET_OP is very input dependent. This is because it basically is an extension of BUILD_BSP. This, in turn, is also what gives it the potential to be faster than MERGE. MERGE is less dependent on the structure of the scene as the partitioning lines will not split more lines. But if a BSP-tree is built on a bad scene, MERGE will still suffer. A case of this is when the BSP-tree resembles a grid or when the scene forces MERGE to include redundant lines, which it cannot handle and therefore create larger trees. INC_SET_OP is better at cutting away these redundant lines which results in smaller trees. If the BSP-tree splits the BREP into small sections where only a constant amount are to be included in the resulting BSP-tree, then INC_SET_OP will run very fast. If all lines are relevant then MERGE will create the smallest trees of the two algorithms. In our tests we did not include the huge decrease in size from collapsing all leaves, but this is of course something that should be taken into account when looking at MERGE as a whole.

We can also conclude that INC_SET_OP is not always linear. We showed that if the worst case scenario for BUILD_BSP is put inside a convex polygon, then the running time for INC_SET_OP was bounded by this running time. This suggests that if the building process of INC_SET_OP is anything but constant in each leaf or that it is linear in a constant amount of leaves, then it is much slower than the expected $O(n)$ running time of MERGE.

Our experiments suggest that if you have a BREP of a polygon then it can be useful to store that information. In almost all of our cases INC_SET_OP was superior to MERGE. The MERGE algorithm does however run in linear time and can produce smaller trees than INC_SET_OP if no redundant lines are represented in the resulting BSP-tree. If the only information that exists is BSP-trees, making INC_SET_OP impossible to use, then MERGE is a competent algorithm as it runs in linear time and uses subroutines that will always make the resulting tree smaller, at least in the number of leaves.

6.1 FUTURE WORK

A clear improvement to the work done in this thesis would be to also implement the work in higher dimensions than two. Here BSP-tree representations are important to have, as a lot of graphical computations are made on these instead of on the actual scene. It would be exciting to see if the INC_SET_OP could outperform MERGE in in higher dimensions as well.

Further work would also be trying out other Linear Programming algorithms than the one we used. Since we used a basic algorithm, a more sophisticated LP algorithm implementation could lead to a performance boost in practice. An investigation into whether or not it could be possible to reduce the size of trees by eliminating the redundant lines stored by MERGE in the scene containing *jaggies*, would also be interesting.

In general it would be interesting to see just how much we could improve the results by optimizing the code in MERGE. It could be the case that smart pointers was not the best way to deal with memory and that a more manual pointer handling would be more efficient. In our work we did not focus heavily on optimizing our code so a decrease in running time for all the algorithms could be found.

BIBLIOGRAPHY

[1] Aristides A. G. Requicha, R. B. T. Mathematical foundations of constructive solid geometry: General topology of closed regular sets. *Techinal Memorandum 27* (March 1978).

[2] James D. Foley, Andries van Dam, S. K. F. J. F. H. *Computer Graphics: Principles and Practice*, 2 ed. Addison-Wesley Professional, 1996.

[3] Mark de Berg, Otfried Cheong, M. v. K. M. O. *Computational Geometry: Algorithms and Application*, 3 ed. Springer, 2008.

[4] Mikola Lysenko, Roshan D'Souza, C.-K. S. Improved binary space partition merging. *Computer-Aided Design 40* (2008), 1113–1120.

[5] William C. Thibault, B. F. N. Set operations on polyhedra using binary space partition trees. *SIGGRAPH '87 21*, 4 (July 1987), 153–162.

Part III

APPENDIX

A

APPENDIX A: ALGORITHMS

Algorithm 1 BUILD_BSP(Set-of-linesegments F)

```
Data: Set-of-linesegments F
Result: BspNode root
choose splitting line H that embeds a line segment of F
BspNode node = new BspNode(H)
partition line segments of F with H into $F_{right}$, $F_{left}$, $F_{same}$
node.addSegment($F_{same}$)
if $F_{left}$.isEmpty() then
    node.left = "in"
else
    node.left = BUILD_BSP($F_{left}$)
end
if $F_{right}$.isEmpty() then
    node.right = "out"
else
    node.right = BUILD_BSP($F_{right}$)
end
return node;
```

Algorithm 2 getDirectionOfPoint(Point p)

```
Data: Point p
Result: int ($< 0$ left $\|$ $> 0$ right $\|$ 0 on line)
double m = normal.getX()
double n = normal.getY()

double a = origin.getX()
double b = origin.getY()

double x = point.getX()
double y = point.getY()

return $m \cdot (x - a) + n \cdot (y - b)$
```

Algorithm 3 INC_SET_OP(Operation OP ; BspNode v ; Set-of-linesegments B)

```
Data: Operation OP ; BspNode v ; Set-of-linesegments B
Result: BspNode mergedTree
if $v$ is a leaf then
    if $OP = \cup^*$ then
        if $v = IN$ then
            return v;
        else
            return BUILD_BSP(B);
        end
    end
    if $OP = \cap^*$ then
        if $v = IN$ then
            return BUILD_BSP(B);
        else
            return v;
        end
    end
else
    partition line segments of B with $H_v$ into $B_{right}$, $B_{left}$, $B_{same}$
    if $B_{left}$ is empty then
        status = $Test\_InOut(H_v, B_{same}, B_{right}$;
        if $OP = \cup^*$ then
            if status = IN then
                (* Overriding old v.left*)
                v.left = new IN-leaf;
            else
                (* do nothing about exterior in union*)
            end
        end
        if $OP = \cap^*$ then
            if status = IN then
                (* do nothing about interior in intersection*)
            else
                (* Overriding old v.left*)
                v.left = new OUT-leaf;
            end
        end
    else
        v.left = INC_SET_OP$(OP, v.left, B_{left})$;
    end
    if $B_{right}$ is empty then
        (* Similar to $B_{left}$ is empty*)
    else
        v.right = INC_SET_OP$(OP, v.right, B_{right})$;
    end
end
```

Algorithm 4 MERGE(BspNode A,B ; Operation OP ; Stack<SplittingLines> R)

```
Data: BspNode A,B ; Operation OP ; Stack<SplittingLines> R
Result: BspNode mergedAndCollapsedNode
if R is infeasible then
    return NULL;
else if A and B are leaves then
    return COLLAPSE(A OP B);
else if A is a leaf or heuristic_swap(A,B) then
    swap(A,B);
end
if IN OP B = OUT OP B then
    return COLLAPSE(IN OP B);
end
R.push($A_h$)
$T_{right}$ = MERGE($A_{right}$, B, OP, R);
r.pop();
R.push($A_h^C$)
$T_{left}$ = MERGE($A_{left}$, B, OP, R);
r.pop();
if $T_{right}$ = NULL then
    return $T_{left}$;
else if $T_{left}$ = NULL then
    return $T_{right}$;
end
Node mergedNode = Node($A_h$, $T_{right}$, $T_{left}$);
return COLLAPSE(mergedNode);
```

Algorithm 5 COLLAPSE(BspNode root)

```
Data: BspNode root
Result: BspNode collapsedTree
if id[root] is set then
    return root;
end
if root is not leaf then
    root = node($root_{splitline}$, COLLAPSE($root_{right}$), COLLAPSE($root_{left}$));
    if id[$root_{right}$] = id[$root_{left}$] then
        return $root_{right}$;
    end
end
if hash[root] $\in$ visit then
    return visit[hash[root]];
end
id[root] = count;
count++;
visit[hash[root]] = root;
return root;
```

B

APPENDIX B: FURTHER RESULTS

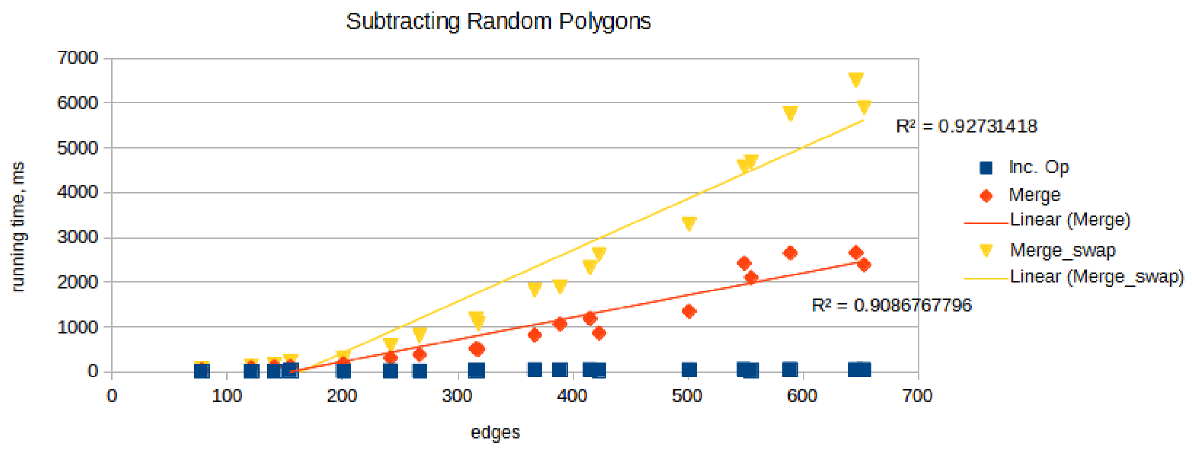


Figure 54.: Running time of substracting random polygons. Shown with and without swap.

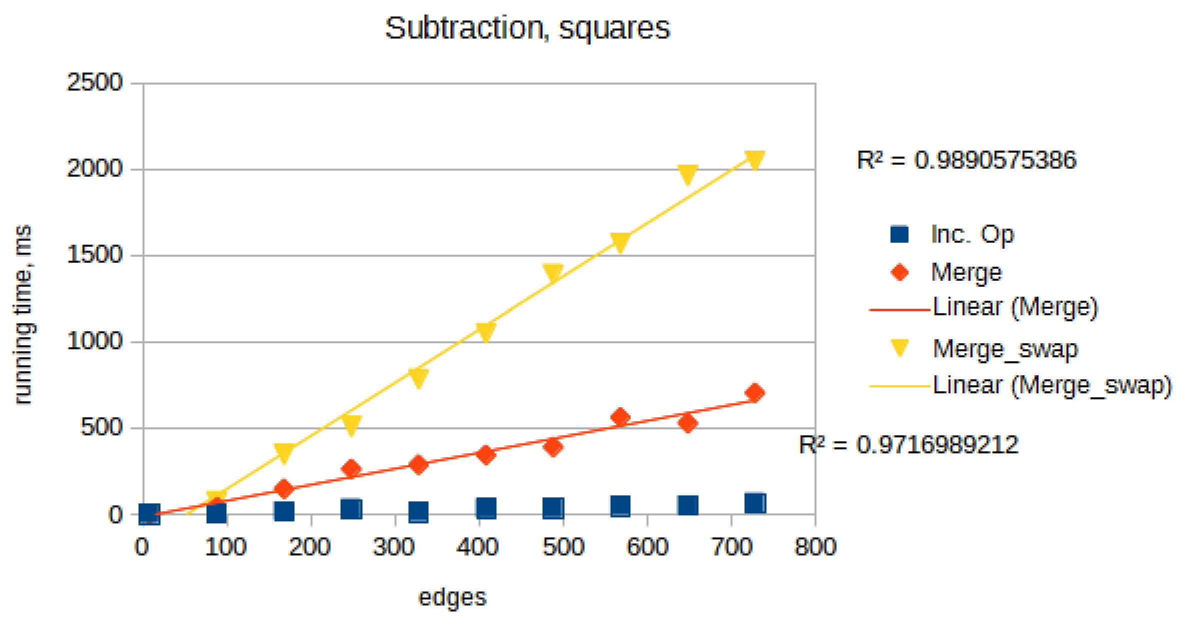


Figure 55.: Running time of subtracting squares scene with increasing edges. Shown with and without swapping the inputs

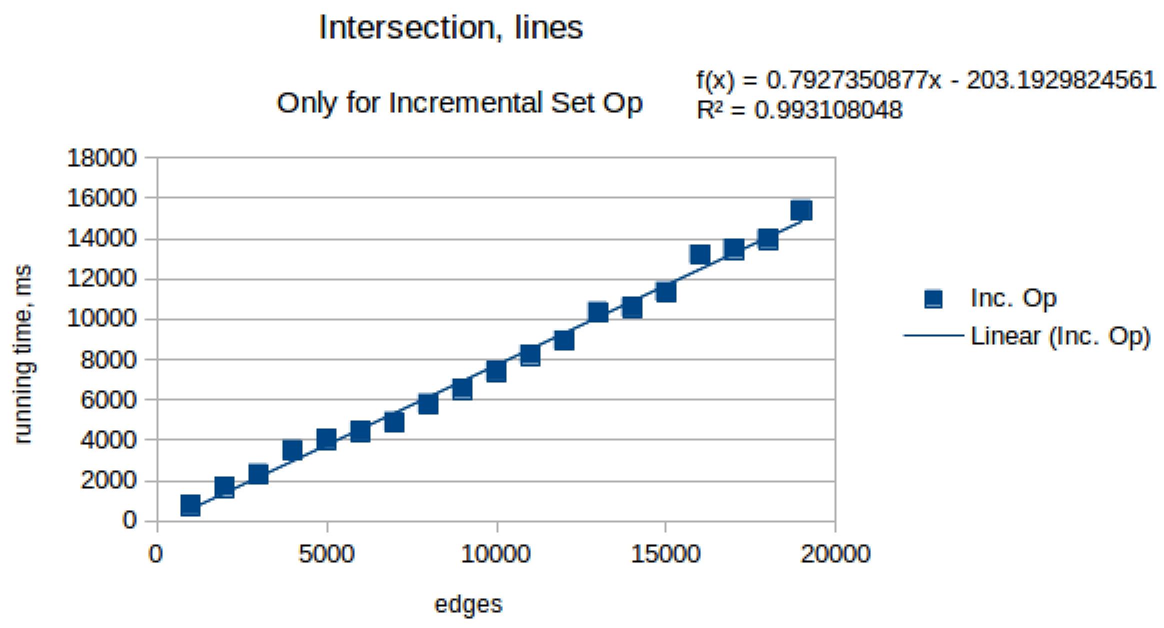


Figure 56.: Running time for intersection of lines, larger input