FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor Thesis in Computer Science

Representation of General Geometric Forms for Humanlike Problem Solving

Wiebke Köpp

FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor Thesis in Computer Science

Representation of General Geometric Forms for Humanlike Problem Solving

Repräsentation allgemeiner geometrischer Formen für menschenähnliche Problemlösealgorithmen

Author:	Wiebke Köpp
Supervisor:	Dr. Alexandra Kirsch
Advisor:	Dr. Alexandra Kirsch
Date:	August 16, 2012

I assure the single handed composition of this bachelor's thesis only supported by declared resources.

Munich, August 16, 2012 Wiebke Köpp

Acknowledgments

First and foremost I would like to thank my advisor, Dr. Alexandra Kirsch for her great support and guidance throughout implementation and writing process and for introducing me to this very interesting field in the first place. Her helpful comments always kept me on track when I wandered to far away from the actual goals of the thesis.

Furthermore, I am very grateful to my family, especially my loving boyfriend and my friends for their advice, support and patience, particularly during the last weeks of this thesis.

Abstract

In general, cutting and packing problems are optimization problems involving optimally placing given small objects into bigger ones. This thesis addresses the so–called cookie–cutting problem, the problem of cutting as many cookies as possible from a given rolled–out dough. One possible approach to solve this problem is using a search algorithm to find an arrangement of cookies within the dough, that contains the maximum number of cookies. Such an algorithm requires a description of the both dough and the shapes to be arranged as input. The data structure representing them needs to provide the possibility to transform a shape, i.e. translate or rotate it and to test for intersection and containment of two shapes. Transformations are needed to find a position, while intersection and containment testing is used to check if that position is admissible.

The objective of this thesis is to find out, if it is better when said representation is based on pixel graphics or when it is predicated on vector graphics. Hence, possible existing representations are examined first and then enhanced where needed. The vector–based approaches comprise the class *Area* and the group of classes associated with the interface *Shape* from the *Java 2D API* and the class *Geometry* from the *JTS Topology Suite*. Two pixel representations partially use the classes *BufferedImage* and *Graphics2D*, but have mostly been self–developed. Subsequently all representation are compared, resulting in the conclusion that vector–based approaches are considerably faster than pixel–based ones. The representations derived from *Shape* and *Geometry* have approximately the same effectiveness, while *Area* is not recommended due long runtimes and high memory consumption.

Zusammenfassung

Diese Arbeit befasst sich mit dem sogenannten Cookie–Cutter Problem (Ausstechformen–Problem), welches der Problemklasse der Packungs– und Zuschnittprobleme angehört. Packungs– und Zuschnittprobleme beschäftigen sich damit, wie eine gewisse Anzahl an kleineren Objekten in einem oder mehreren größeren Objekten möglichst optimal angeordnet oder untergebracht werden können. Beim Cookie–Cutter Problem soll ein Weg gefunden werden, wie aus einem Stück ausgerollten Teig so viele Plätzchen wie möglich ausgestochen werden können. Eine mögliche Lösung für dieses Problem ist das Anwenden eines Suchalgorithmusses zum Finden einer Anordnung von Plätzchen, die maximal viele Plätzchen enthält. Einem solchen Algorthimus müssen sowohl die Form des Teigstücks, als auch die Umrisse der Ausstechformen, die im Teig verteilt werden sollen, bekannt sein. Beiden muss intern durch eine Datenstruktur repräsentiert werden, die das Anwenden von Transformationen erlaubt und zudem Tests bereitstellt, die überprüfen ob zwei Figuren sich überschneiden oder ineinander liegen. Die Tranformationen werden benötigt um eine Position einer Form zu finden. Die Überprüfung von Lagebeziehungen wird verwendet, um zu überprüfen ob eine Position zulässig ist.

Die vorliegende Arbeit soll untersuchen, ob es besser ist, die Figuren durch eine Repräsenation darzustellen, die auf Pixelgraphiken basiert oder eine deren Grundlage Vektorgraphiken sind. Dafür werden bereits bestehende Repräsentationen geometrischer Formen untersucht und wo nötig anschließend erweitert. Unter den Vektor–Repräsentation befinden sich die Klassen *Area* und die dem Interface *Shape* angehörigen Klassen der *Java 2D API*, sowie die Klasse *Geometry* der *JTS Topology Suite*. Zwei Pixel–Repräsentationen werden nahezu vollständig selbst implementiert, nehmen aber die beiden Klassen *BufferedImage* und *Graphics2D* zur Hilfe. Anschließend werden alle Repräsentationen miteinander verglichen. Dabei wird festgestellt, dass vektorbasierte Repräsentationen wesentlich schneller sind als pixelbasierte. Zudem halten sich die von *Shape* und die von *Geometry* abgeleiteten Datenstrukturen in etwa die Wage, während *Area* hauptsächlich aufgrund seiner Unzuverlässigkeit in Bezug auf sowohl Laufzeit als auf Speicherverbrauch schlechter bewertet werden muss.

Contents

1. Introduction

1.1. Motivation

Cutting and packing (C&P) problems are optimization problems that arise in various different application areas, ranging from garment and metal industries to even the private household. While cutting problems deal with minimizing the left over waste from cutting one or more given items into certain pieces, packing problems concern packing several small items into one or more big ones. Increasing scarcity of resources and economic competition motivate companies to use materials and other resources like storage space and means of transportation as effective as possible. Therefore they can benefit from efficient solutions to C&P problems. However, applications can also be found in every household. This thesis addresses the so-called cookie cutting problem [13, p. 173], the problem of cutting as many cookies as possible from a given rolled–out dough.

Today, computers are capable of performing many different tasks very hard to execute by humans, but on the other hand seem to struggle with problems that are relatively easy for humans. Cutting cookies and solving C&P problems in general are examples for such problems, some have even been proven to be NP-complete [11][10]. While humans approach the cookie cutting problem by using spatial reasoning, a computer could use a search algorithm. As a first step to implementing a new algorithm for this problem, it needs to be examined how dough and cookie cutter shapes are best represented internally. The input of the algorithm would be the dough and the shapes to be arranged as either pixel or vector graphics. In order to find a suitable position for a cookie, it also would need to be able to transform a shape, i.e. translate or rotate it. Cookies are obviously not allowed to overlap and have be placed completely within the dough. Therefore, testing for intersection and containment are further functionalities that need to be provided by a representation.

1.2. Objective

This thesis has three major goals. The first goal is to find existing implementations based on both pixel or vector graphics that provide all required functionalities or at least some of them and can therefore be used as a representation for cookies and dough. The second goal is to complete the representations that do not fully meet the requirements. This can also include enhancing or replacing methods already available. Last, all representations have to be compared with each other to finally decide which representation based ob which graphics format is better.

1.3. Thesis Outline

This thesis is structured as follows:

Chapter 2 covers related work and briefly introduces basic principles. The topics addressed in this chapter include packing problems and approaches to solve them, previous work and basic principles of intersection, containment and transformation of shapes and the major concepts of pixel and vector graphics.

Chapter 3 is divided into three sections. The first section review the requirements for each representation. The second section covers representations based on pixel graphics and the third section discusses representations based on vector graphics. Each representation is introduced separately including information on how transformation of one object and intersection testing and containment testing of two objects of the respective representation are performed.

Chapter 4 focuses on implementation details. It shows the overall structure of the program including an explanation of program specifics. Since the implementation partially uses methods provided by different JAVA libraries, this chapter also includes information about those libraries and a detailed overview on which methods have been provided by existing libraries and which functionality has been added.

Chapter 5 evaluates and compares the different representations. It explains the test setting first. Then it shows the results of functionality and runtime tests.

Chapter 6 includes ideas that could be incorporated additionally in order to extend possible applications.

Chapter 7 concludes this thesis.

Appendix A shows how the complexity of test shape has been measured

Appendix B contains the data of all runtime tests

2. Related Work and Fundamentals

The first section of this chapter introduces packing problems in general and shows previously suggested approaches. The following section briefly explains principles of projective geometry needed to transform objects. Computational geometry is a field of computer science concerning the study of geometric algorithms including intersection and containment problems. Therefore, a section about computational geometry is also included in this chapter. The last two sections give a short overview of vector and pixel graphics as all representations of geometric shapes are based on either one of them.

2.1. Packing and Cutting Problems

As stated before, C&P problems have many different application areas. They have been thoroughly researched resulting in an exceedingly high number of publications. As this thesis can hardly cover all different kinds of C&P problems in full width, this section introduces only a few well known problems. Additionally, surveys classifying and describing C&P problems are referenced, as well as solution approaches to problems closely related to the cookie-cutter problem.

One of the earliest surveys on this topic is a paper by Herbert Dyckhoff [7]. His original intention was to introduce a topology for C&P problems by which these problems can be classified, but in doing so, he also names many resources, with the oldest dating back to 1939. His topology is mainly based on the overall dimensionality of the problem, restrictions on how shapes can be positioned and the nature of small and big shapes. [27] refines Dyckhoff's topology and summarizes advances that have been made since Dyckhoff's paper from 1990. The new topology is developed by reformulating Dyckhoff's criteria and introducing new ones. According to [27], the cookie–cutter problem addressed in this thesis, would classify as a two–dimensional cutting stock problem with irregular shapes. The cutting stock problem is defined below along with two other famous problems. It should be noted that all have in common that they involve placing small objects into bigger ones.

Bin Packing Problem: The objective of bin packing problems is to fit all given small items into the big ones, which are referred to as bins. However, the solution has to use as few bins as possible. Industrial applications of this problem are the minimization of storage units in a factory or the number of trucks used for transporting goods.

Cutting Stock Problem: Cutting stock problems deal with finding a strategy for cutting small pieces out of one or more big objects, while minimizing the left over waste. Applications of this problem can be found throughout many different industries, practically everywhere where materials have to be cut into smaller pieces. One example is the cutting of leather. In addition to minimizing the waste, different quality areas have to be considered, which makes the problem even harder to solve.

Knapsack Problem: In this problem, big objects are called knapsacks. Their number and capacity is limited. Small objects have a value, e.g. their size or weight, associated with them. The goal of the knapsack problem is to maximize the total value distributed over all knapsacks. A household application of the knapsack problem is packing items for a vacation and complying with the limitations on weight and quantity of suitcases at the same time.

All three problems exist in many variations, some of which are described in [5]. While this survey mainly focuses on approaches to problems involving rectangles, a later paper [6] by the same authors covers packing of irregular shapes.

[2] is not a survey, but contains a single approach for packing or cutting irregular shapes. The approach is particularly interesting, as unlike many others, the authors choose to represent shapes as pixel images. The motivation behind their approach is to provide a solution that scans the input using a camera and therefore could operate in a fully automated environment without human interference. The proposed algorithm consists of two phases. During the prelayout phase all input shapes are scanned and their respective minimum enclosing bounding rectangles are calculated. The second phase is used to determine optimal positions for all shapes. It is called layout phase. Determining positions for shapes is done iteratively. Each shape is placed using information about the dimensions of the shape, the currently void regions of the original big shape and the resulting packing density at a particular position.

Whelan and Batchelor [28] also address automated packing of arbitrary shapes. However, their paper does not just explain an algorithm to solve packing problems theoretically, but describes a whole system including a component that physically manipulates the objects to be arranged. Therefore, the system consists of two major components. They are called geometric and heuristic packer. The geometric packer is in charge of finding a position for a shape with a given orientation. The heuristic packer proposes such orientations and determines an order in which shapes are best placed into the bigger shape. This paper also includes a detailed description of previous research and names various applications for automated packing.

The paper [12] suggests an agent–based approach. Like the approaches before, it is designed for packing irregular shapes. One major difference is that shapes are represented by neither vector nor pixel graphics, but by agents. The algorithm is based on random initialization of an arrangement and then improving and extending that arrangement iteratively until optimality is reached. The improvement is based on forming relationships between closely positioned agents. Agents gain rewards if they are part of bigger relationship groups, thus are densely packed. This causes agents to stay close together and automatically desire an optimal solution to the problem.

2.2. Projective Geometry

Projective geometry is a field of mathematics, that by defining geometric objects in a projective rather than an euclidean space simplifies many calculations. The projective plane is basically an euclidean plane embedded in the three dimensional space. One of the major concepts of projective space are homogeneous coordinates. They are the projective version of an euclidean coordinate and are derived by transforming an euclidean point $(x, y)^T$ which is then described by three instead of two coordinates. Even though the plane theoretically could be embedded at multiple different positions, an embedding of the plane at $z = 1$ is most common. Therefore the point $(x, y)^T$ has coordinates $(x, y, 1)^T$ in the projective plane. This, together with a projective definition of lines, leads to very interesting principles for intersecting lines and connecting points, but more importantly has a great impact on how the transformation of points can be calculated.

The representation of a transformation within the usual euclidean space depends on the type of transformation. The description of a translation is different from the one of a rotation. Therefore, the composition of multiple transformations results in complicated expressions. This is not the case for the projective plane. Its properties lead to a uniform representation of transformations. All of them can be described by matrices, which is not true in euclidean space, since translation are not linear. Therefore, a composition of arbitrary transformations in projective geometry is equal to a simple matrix multiplication, which is much easier to compute than having to handle multiple different representations at the same time. Table 2.1 shows the results for different transformations of $(x, y)^T$ in the euclidean and the projective plane.

This principle plays a major role in transforming shapes in vector as well as pixel graphics. A great resource for an introduction to projective geometry is the recently published book *Perspectives on Projective Geometry* [19]. The topic discussed above is addressed in chapter 3.

2.3. Computational Geometry

Computational Geometry is a scientific discipline that deals with efficient algorithms and data structures for solving geometric problems. Questions that can be answered by using computational geometry tools include the following:

- Given a set of points, which pair of points are closest to each other?

Transformation	**Euclidean Plane**	**Projective Plane**
Rotation	$\begin{pmatrix}\cos(\alpha) & -\sin(\alpha)\\ \sin(\alpha) & \cos(\alpha)\end{pmatrix}\begin{pmatrix}x\\ y\end{pmatrix}$	$\begin{pmatrix}\cos(\alpha) & -\sin(\alpha) & 0\\ \sin(\alpha) & \cos(\alpha) & 0\\ 0 & 0 & 1\end{pmatrix}\begin{pmatrix}x\\ y\\ 1\end{pmatrix}$
Scaling	$\begin{pmatrix}s & 0\\ 0 & s\end{pmatrix}\begin{pmatrix}x\\ y\end{pmatrix}$	$\begin{pmatrix}s & 0 & 0\\ 0 & s & 0\\ 0 & 0 & 1\end{pmatrix}\begin{pmatrix}x\\ y\\ 1\end{pmatrix}$
Translation	$\begin{pmatrix}x\\ y\end{pmatrix}+\begin{pmatrix}t_x\\ t_y\end{pmatrix}$	$\begin{pmatrix}1 & 0 & t_x\\ 0 & 1 & t_y\\ 0 & 0 & 1\end{pmatrix}\begin{pmatrix}x\\ y\\ 1\end{pmatrix}$

Table 2.1.: Comparison of transformations in euclidean and projective plane

- Given a set of line segments, do any of the segments intersect?
- Given a point and a polygon, does the polygon contain the point?
- Given a set of points, what is the convex hull of those points, thus the smallest convex polygon containing all points?

A great resources for an extensive overview on both problems and solutions in computational geometry can be found in [26]. [17] is a chapter of the *Handbook of Discrete and Computational Geometry* concerning geometric intersection problems. It refers to many different approaches including the ones that will be applied to the intersection problem of arbitrary shapes in Chapter 3.

2.4. Pixel Graphics

In computer graphics, there are two major principles to represent images [14, ch. 2]. One principle is to represent an image by a pixel or raster graphic, vector graphics, as the other one, will be introduced within the subsequent section. Raster graphics are basically matrices of fixed height and width, where each matrix position or pixel contains information about the color at that position. Since colors are usually represented as a sequence of bits, raster graphics are often also referred to as bitmaps. The length of such a bit sequence depends on the color model, thus the range of colors, for an image. The simplest type of raster graphics are black–and–white images, as each pixel can only have one of two values. A value of 0 represents a black pixel, a value of 1 a white one. Black–and–white bitmaps suffice for representing geometric objects, since the only interesting property of a pixel is if it is covered by the shape or not. In general, pixel graphics have three major drawbacks. The first are aliasing effects. These effects occur since representing something originally continuous by something with discrete steps will always involve some kind of discretization. Figure 2.1 shows an arrow represented by pixel images in three different resolutions. Saving an image in one of the resolutions on the left will make it almost impossible to derive the same information contained in the image on the right without additional information or processing. The second closely related drawback is that raster graphics cannot be scaled without loss of information. Similarly, rotating a shape represented by a pixel graphic will result in a slightly different shape due to newly introduced aliasing effect. Third, saving an image at a high resolution can lead to very big files. Despite these disadvantages, pixel images still play a very important role in computer graphics. One reason is the simplicity of the bitmap data structure. Another, possibly even more important cause is that monitors, printers and digital cameras depend on pixel images. Common file formats for raster graphics are BMP, GIF, JPG, PNG and TIFF.

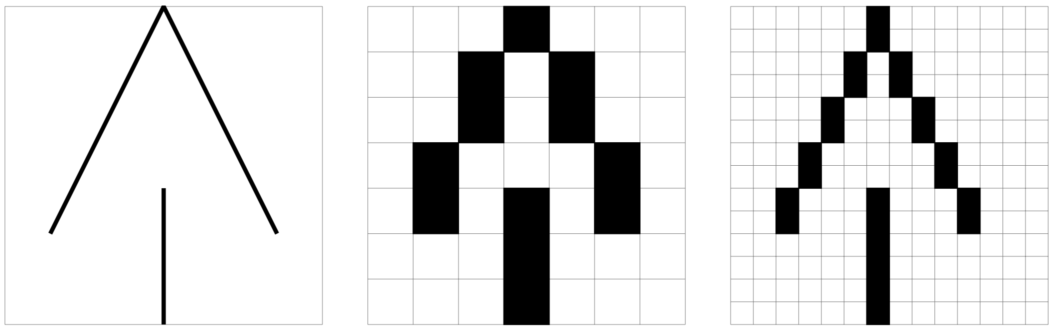


Figure 2.1.: An arrow drawn as raster graphics in three different resolutions (see [14, p. 9])

2.5. Vector Graphics

In contrast to the discrete representation of pixel graphics, vector graphics constitute a more mathematical description of shapes. The most commonly used format for vector graphics are scalable vector graphics (SVG). SVG is a XML–based language used to describe two–dimensional graphics by structured text. It has been has been standardized by World Wide Web Consortium, who describe the standard in [4]. XML files are structured by using a defined set of elements and attributes within those elements specific to the language used. For SCV, these elements include geometric shapes, text and elements for grouping other elements. SVG provides a number of basic shape elements along with the possibility to represent arbitrary shapes as a sequence of two– and three–dimensional bézier paths and lines or a combination of basic shapes. These basic shapes are circles, ellipses, lines, polygons, polylines and rectangles. The following four basic shape elements will play a major role in the vectorbased approach of this thesis and are therefore defined below:

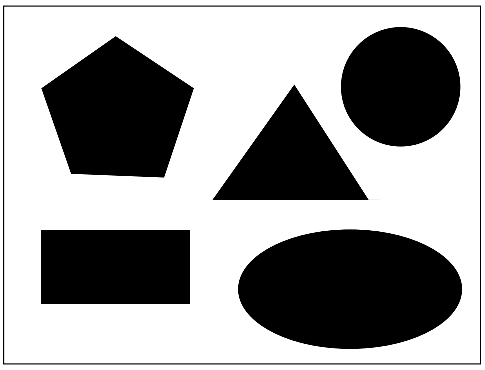
circle element: a circle with its center point and its radius as attributes

ellipse element: an ellipse given by its center point and the radii in x and y direction

polygon element: a polygon defined by a given number of points

rectangle element: a rectangle with its upper left corner, its width and its height as attributes

The additional **path element** can describe arbitrary shapes and contains a sequence of bézier paths and lines as attributes. A sequence consists of points and commands defining the type of the following segment. Figure 2.2 shows an example for a SVG graphic and the corresponding code The path element contains only the commands `M` (MoveTo), `L` (LineTo) and `Z` (ClosePath). The other two most common commands are `C` for cubic and `Q` for quadratic bézier curves.



```
<svg width="640" height="480" xmlns="http://www.w3.org/2000/svg">
 <g>
  <title>A combination of different shapes</title>
  <circle cx="533" cy="108" r="80" fill="black"/>
  <ellipse cx="465" cy="380" rx="150" ry="80" fill="black"/>
  <polygon points="215,230 90,225 50,110 150,40 255,110"
   fill="black"/>
  <path d="m 490,260 l -100,-155 l -110,155 l 225,0 l -15,0 z"
   fill="black"/>
  <rect x="50" y="300" width="200" height="100" fill="black"/>
 </g>
</svg>
```

Figure 2.2.: A combination of different shapes in a vector graphic

3. Approach

The previous chapter discussed basic principles of pixel and vector graphics. This chapter introduces five different representations that meet the requirements mentioned in Section 1.1. Two representations are based on pixel graphics. They are described in Sections 3.2.1 and 3.2.2. The three vectorbased representations can be found in Section 3.3.1, 3.3.2 and 3.3.3. Prior to illustrating individual representations, Section 3.1 reviews defined requirements. All representations somehow depend on existing libraries as stated within the respective section. However, for a detailed itemization of used libraries and added functionalities see Section 4.2.

3.1. Common Functionalities of all Representations

The objective of this thesis is to find representations of geometric shapes that can be used by a search algorithm solving the cookie–cutter problem. In general, search problems are defined by at least three things: Firstly, a state space including an initial state and one or more goal states. Secondly, the actions available in one state and their results and third a goal test determining if a state is the goal [20, ch. 3]. Given the problem definition various different search algorithms can be applied to a problem. In case of the cookie–cutter problem a possible problem definition includes a set of shapes with their positions as the state space and the empty set as the initial state. Actions are adding a new shape and transforming a shape and are available in every state. The goal is to pack as many shapes as possible into the dough. Shapes are not allowed to overlap or to leave the inside of the dough. Therefore, the goal test has to include testing if none of the shapes intersect and if all shapes are contained by the dough. Altogether, this leads to requiring representations of geometric shapes to implement a way to both transform shapes and test for intersection and containment.

The shape representations presented in this thesis provide the following functionalities:

Translation: translates a shape by given x and y values

Rotation around the object center: rotates a shape around the center of its bounding box by a given angle θ.

Rotation around the picture center: rotates a shape around the center of the image canvas by a given angle θ.

Scaling with the object center as fix point: scales a shape with a given factor without moving the center of its bounding box

Scaling with the picture center as fix point: scales a shape with a given factor without moving the picture center

Intersection test: checks if a shape intersects a given other shape

Containment test: checks if a shape contains a given other shape

Both scaling types are technically not needed for the cookie–cutter problem and only given for completeness. Also, rotation around the picture center is not very intuitive from the human perspective as a human will usually rotate things in place.

3.2. Pixel Graphics

3.2.1. PixelImage

The representation PixelImage contains three variables: the original image, the PixelImage has been initialized with, the current image and the current transformation. In terms of libraries, PixelImage uses transformation functionalities of Java2D. For general information on images in JAVA see [14, ch. 4].

Bounding Box Calculation

A pixel image does not contain information about whether it describes a circle, a polygon or a snowman, but only whether a certain pixel is black or white. By calculating the bounding box of all black pixels, at least some information about the dimension and location of the shape within a pixel image can be gained. The information about location and dimension is needed for all required functionalities and therefore the calculation needs to be very efficient. The following section describes three possible ways to calculate the bounding box of all black pixels. For pixel images, bounding boxes are defined by minimum and maximum row and minimum and maximum column, where a black pixel can be found within the image data matrix.

The first bounding box calculation (Bounds1) starts with traversing the image from the top left corner in order to find the minimum row. Then, the image is traversed from the bottom right corner to find the maximum row. Minimum and maximum column are calculated by scanning the image from left and right towards the middle omitting the top and bottom stripes that have already been scanned. An example of the calculation steps for this method can be seen in Figure 3.1

The second bounding box calculation (Bounds2) takes advantage of the fact that shapes used in cookie baking are usually very compact. That means, when the minimum row

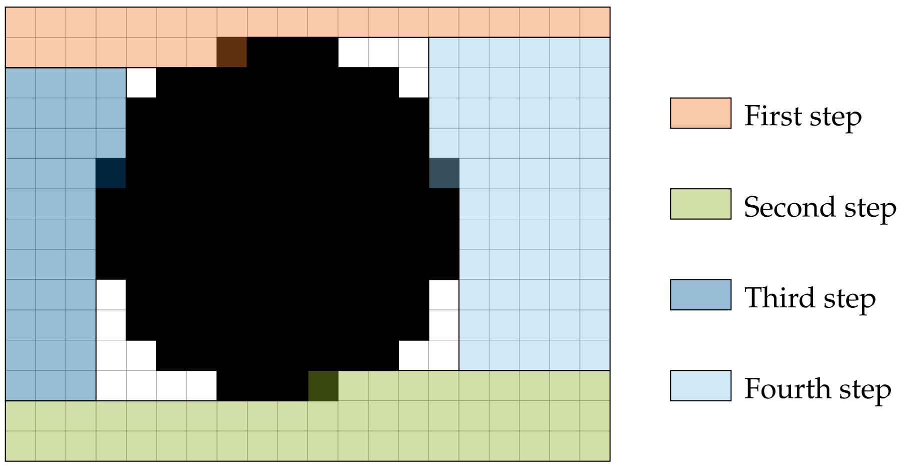


Figure 3.1.: Calculation steps for Bounds1

pixel has been found, it is very likely that the column number of this pixel is close to either the minimum or maximum column number. The first two steps of Bounds2 are exactly the same as the first two steps of Bounds1. At that point, minimum row, maximum row and their respective column numbers are known. The next step uses the minimum of both column numbers and scans the image to the left, starting one column before found column number. Top and bottom rows that have been already scanned are omitted. Scanning stops when one column contains only white pixels. Then, the column before was the last containing a black pixel and therefore is the minimum column. The same principle is used in the last step. It takes the maximum of the two column numbers found at the beginning and scans to the right until a completely white column occurs.

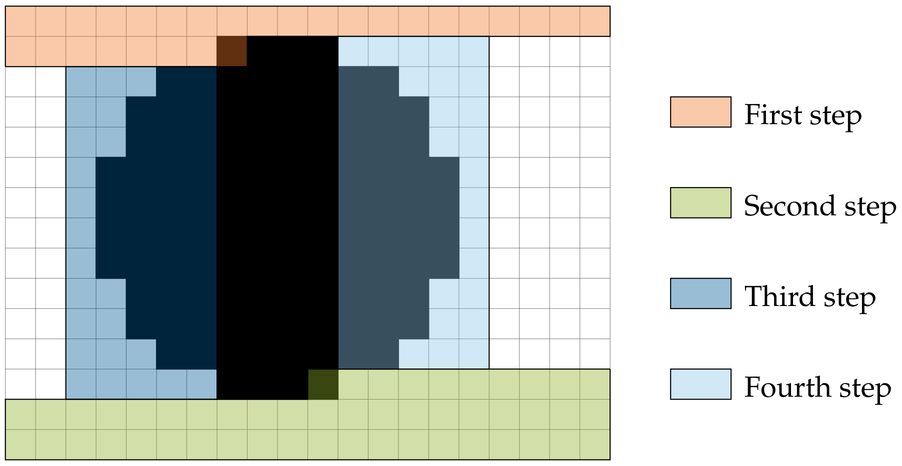


Figure 3.2.: Calculation Steps for Bounds2

The third and last bounding box calculation (Bounds3) uses a approach inspired by breadth first search, starting with one black pixel as a seed. The seed pixel is the first black pixel found in the image. It is searched for as done in the two methods above, by traversing the image from the top. Once this seed pixel is found, all eight neighboring pixels are examined and added to a queue if they are black. The next iteration uses the next pixel from the queue using a first-in-first-out strategy and repeats the process, while making sure that only pixels that have not been visited before are added. Meanwhile, maximum row, minimum column and maximum column are updated. The minimum row number does not have to be updated, as it has already been found at the beginning. The search is repeated until the queue does not contain any pixels, meaning that no further black neighbors have been found.

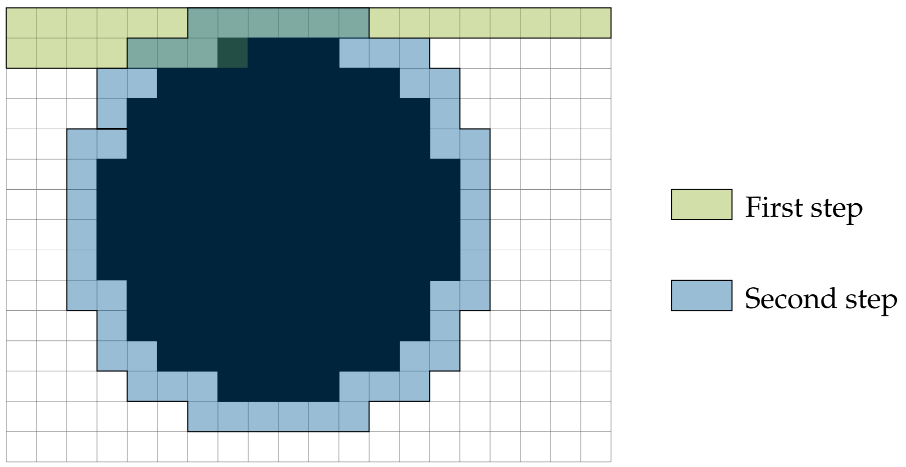


Figure 3.3.: Calculation steps for Bounds3

Transformation

Pixel images are transformed by applying a transformation to the image canvas and then filling the canvas according to the contents of the old image. Applying a longer sequence of transformations to an image and always using the image from the previous step leads to images that have only little similarity with the original shape. They would be useless for further intersection and containment testing. These effects occur due to permanent rounding of pixel positions when transforming images. Figure 3.4 shows how an image of a snowflake is affected by rotating it 180 times by 2°. In consequence, PixelImage does not save only the current image, but additionally saves original image and current transformation. That way, when a PixelImage is translated two steps are executed. First, the transformation is updated by multiplying new and prior transformation matrices. Then, this transformation is applied to the original image and the result is saved to the current image. This leaves original image unchanged and updates the current image.

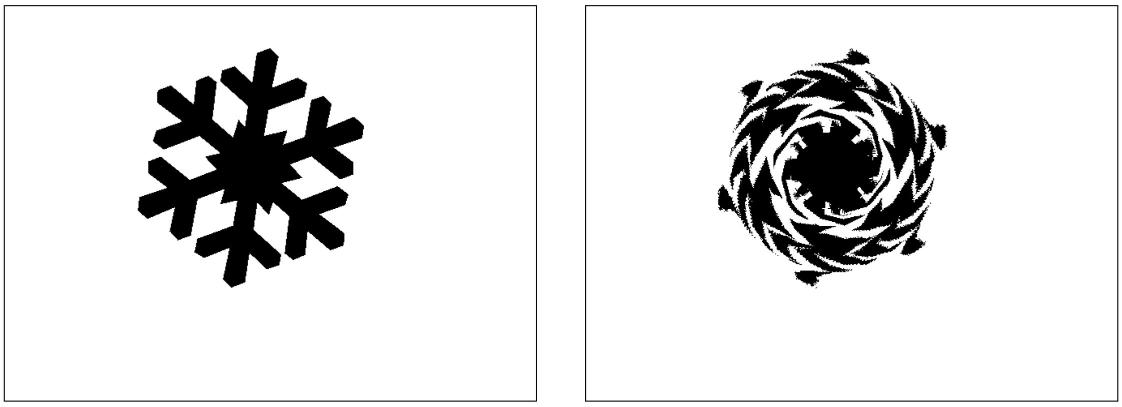

(a) Original Image **(b)** Rotated around 360° in 2° steps

Figure 3.4.: Image transformation using the previous results

Intersection

Testing if pixelbased shapes intersect is comprised by two steps. An intersection can only exist if the bounding boxes of the two shapes intersect. Thus, the first step is to test for bounding box intersection. Two bounding boxes do not intersect if the minimum row value of either one of the shapes is bigger than the minimum row value of the other one and the same holds for columns [9, ch. 4]. In case of a bounding box intersection, the second step is to scan both images within the part where the bounding boxes intersect. During the scanning process, the values of each pixel in both images are examined. If a pair of two black pixels is found during scanning an intersection of the shapes exists, otherwise it does not. An illustration of the process can be found in Figure 3.5.

Containment

Determining if one shape contains another one is done very similar to the intersection process. A shape can only contain another shape if it contains its bounding box. Bounding box containment for shapes A and B, where A is the shape potentially containing the other shape, can be checked as follows. The minimum rows and columns of A have to be smaller than $B's$ respective values, while maximum rows and columns of A have to be bigger than $B's$ respective values. If the test is positive, the bounding box of B is scanned completely, while looking for pixels where B is black and A is white. If such a pixel is found, the containment test for A and B is negative. Figure 3.6, shows an example with a circle (A) and a rectangle (B), where the bounding box test yields a positive result. However, the overall test is negative due to the one pixel located in the lower left corner of the rectangles bounding box.

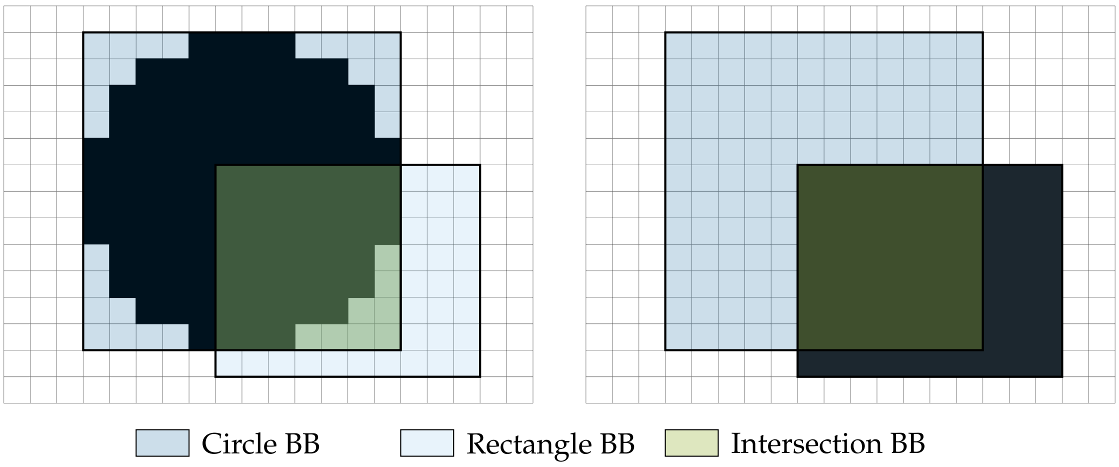


Figure 3.5.: Intersection test for two pixel images

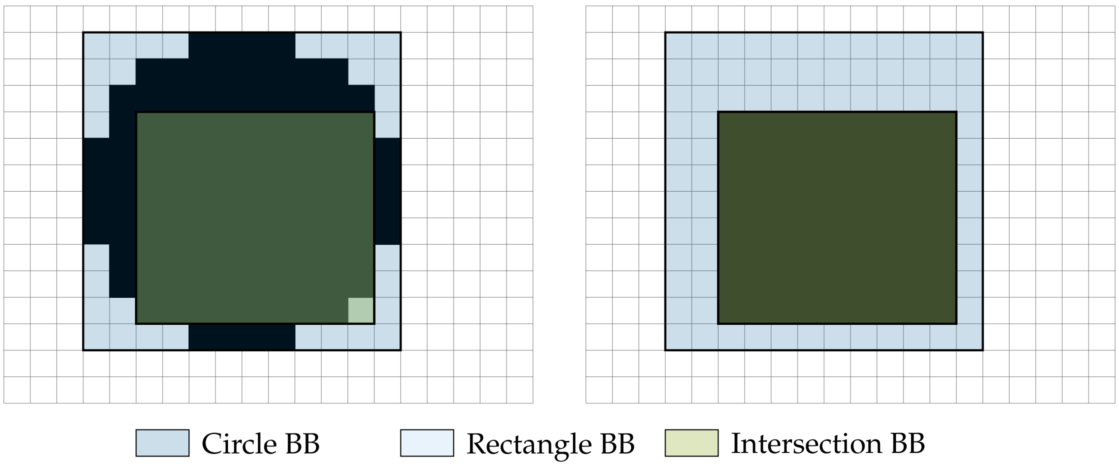


Figure 3.6.: Containment test of two pixel images

3.2.2. PixelImageEnhanced

The representation PixelImageEnhanced is a special kind of PixelImage that has further variables. In addition to original image, current image and current transformation, PixelImageEnhanced saves the current bounding box as well as the current object center, thus the center of the bounding box. That way, bounding boxes do not have to be recalculated every time an intersection or containment test occurs. On the other hand, that implies that transformations must include and update of bounding box and object center.

3.3. Vector Graphics

3.3.1. Area

The representation Area is a class that is part of the Java2D library. Basic principles about areas and other classes from Java2D are described in [14, ch.2]. More detailed information can be found in [15]. The main purpose of the area class is to provide a possibility to recursively perform constructive area geometry, which is why area was chosen as one of the representations. The term constructive area geometry is derived from its 3D version, named constructive solid geometry. Both are methods to construct complex geometric objects by combining multiple simpler objects using boolean operations. In 3D, those simpler objects can for example be spheres, cuboids, polyhedra or other previously combined objects. The corresponding shapes in 2D are circles, rectangles and polygons. Area provides the following four methods to combine two areas:

Union of two areas: The result contains the whole area covered by the two original areas.

Intersection of two areas: The result contains only the area where the two original areas intersect.

Subtraction of one area from another: The result contains the first area excluding the intersection of both areas.

Exclusive–Or of two areas: The result contains the whole area covered by the two original areas excluding the part where they intersect.

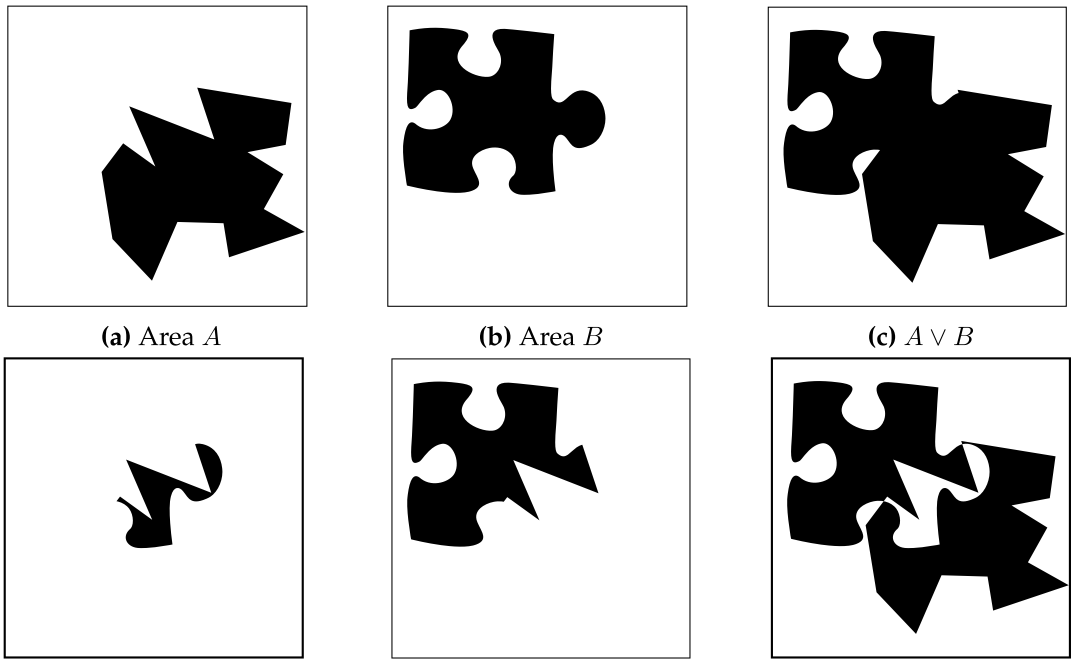
Figure 3.7 shows the result of those methods for two areas.

Transformation

Areas are internally saved as a path consisting of lines and bézier curve segments. A transformation of an area is therefore a transformation of all defining segments, which is essentially a transformation of all coordinates defining each segment.

Intersection

The class area itself does not provide methods to perform intersection and containment tests, but the boolean operators explained above can be used to derive information about the relative position of two areas. One possible way to test if two areas intersect is to actually compute the resulting area of the intersection and check if this new area is empty or not.



(a) Area A **(b)** Area B **(c)** $A \vee B$

(d) $A \wedge B$ **(e)** $A \wedge \neg B$ **(f)** $A \oplus B$

Figure 3.7.: Boolean operations on two areas

Containment

Determining if one area contains another one is done in a very similar way. First, the intersection of the two areas is calculated. Assuming we want to know if an area A contains an area B, we have to check if the intersection area exactly equals area B. If that is the case, A completely contains B.

3.3.2. Shape2D

The representation Shape2D consists of a collection of representations, one for each of the basic shapes in SVG and a general shape that is defined by a path. The following itemization specifies individual shapes and gives details on the variables they contain. The representation Shape2D is mainly based on classes and methods provided by the Java2D library especially classes associated to the interface Shape.

Circle: Java2D does not provide a separate class for circles. A circle therefore has to be defined by an ellipse, where both radii are the same.

Ellipse: Ellipses are defined by either an ellipse object of Java2D or a general Path. This depends on whether the ellipse is axis–aligned or not, as Java2D ellipses only represent axis–aligned ellipses. Ellipses always have to be initialized with an axis–aligned version, even if the intended ellipse is not. Then, a rotation leading to the desired ellipse has to be applied subsequently to initialization. Ellipses therefore contain variables for both an ellipse object and a path. Additionally, a boolean value indicates if the ellipse is currently axis–aligned or not. This value is used by intersection and containment tests to always make sure that the correct representation is used.

Polygon: Java2D only provides a class for polygons defined by integers. As all other shapes use doubles, polygons are represented by a path containing exclusively line segments

Rectangle: The same principles explained for ellipses above, also apply to rectangles. Rectangles therefore contain a Java2D rectangle, a path and a boolean value indicating if the rectangle is currently axis–aligned or not.

Free shape: Free shapes contain three variables. The first is a general path with both line and bézier segments. The second is a boolean value indicating if a shape can additionally be represented by a combination of the 4 basic shapes above. Optionally, the combination of basic shapes is saved as an array of Shape2D.

Transformation

One of the major drawbacks of Java2D is that, even though it does provide methods for transforming shapes, these methods always convert shapes into general paths regardless of which type the shape originally had. Since intersection and containment testing use specific characteristics of the original shapes, these have to be preserved. This means that the build–in transformation methods can not be used directly for all shapes.

Circles are transformed by transforming center and radius separately and then creating a new circle from the result. The center point has to be transformed within all transformation methods except in the case of scaling around the object center which only affects the radius. The radius only changes within both scaling methods but not for translation and rotation.

The description of all shapes part of the representation Shape2D, explained that Ellipses and Rectangles are both shapes that can only be saved as Java2D ellipse and rectangle objects when they are axis–aligned, and have to be saved as general paths otherwise. Within the cookie–cutter problem, the only transformations used are translations and rotations. A sequence of only translations and rotations has the property that the cumulated rotation portion can be read directly off the first two values in the first and second row of the transformation matrix. The following equation shows this for just one rotation and one translation.

$$\begin{pmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) & \cos(\alpha)t_x - \sin(\alpha)t_y \\ \sin(\alpha) & \cos(\alpha) & \sin(\alpha)t_x + \cos(\alpha)t_y \\ 0 & 0 & 1 \end{pmatrix}$$

Hence, it can be determined directly from the matrix if the originally axis–aligned ellipse or the originally axis–aligned rectangle has been rotated around either $90°$, $180°$, $270°$ or $360°$ and is therefore again axis–aligned. Those rotations correspond to the following four transformation matrices, where x and y denote values derived from multiplying matrices as shown above:

$$\begin{pmatrix} 0 & -1 & x \\ 1 & 0 & y \\ 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} -1 & 0 & x \\ 0 & -1 & y \\ 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 & 1 & x \\ -1 & 0 & y \\ 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 & x \\ 0 & 1 & y \\ 0 & 0 & 1 \end{pmatrix}$$

Altogether, that leads to following mechanism for transforming ellipses and rectangles. Translating a shape consists of translating the path of the shape and the ellipse or rectangle object if the shape is currently axis–aligned. Translation of axis–aligned shapes is done in a similar manner to the translation of circles. Rotating a shape consist of rotating the path and then checking if the overall transformation matrix is equal to one of the above. If that is the case, the axis–aligned shape objects are updated, by calculating parameters from the path. Additionally the transformation matrix is set to the identity, since the old values are not needed anymore and only make future computations more complicated. Updated ellipses are derived by using all points of the path except control points for bézier segments to calculate both radii and the center. Rectangles are updated by simply setting them to the bounding box of the path after rotation. When the transformation methods are used in a context that allows scaling and a scaling is applied when the shape is not axis–aligned, the shape can only be represented by a path from that point on, since recognizing the rotation portion of a matrix involving scaling and rotation is not possible.

Polygons and free shapes are both defined by paths in their original version. Therefore, the provided transformations methods can be used to transform them.

Intersection

As already mentioned in sections about other representations a test for bounding box intersection is useful to avoid unnecessary costly computations. Therefore, all intersection tests start by determining if the two bounding boxes intersect. Despite two exceptions, the subsequent more precise intersection test for two Shape2D objects is based on either one of two algorithms. The first algorithm tests for intersection between one of the two round basic shapes, circle and ellipse and any other shape and the second handles polygon intersection, sometimes including a approximation of curves to polygons.

The intersection between two circles is one of said exceptions. Determining if two circles intersect is done by calculating the distance between the two center points and checking if that distance is smaller than the sum of the two radii. The other exception concerns the intersection of axis–aligned rectangles and any other shape. It does not use one of two algorithms, but a method provided by Java2D. This is the only intersection method directly provided by Java2D and it is implemented differently for each shape.

The first algorithm, dealing with intersection tests for circles and ellipses with arbitrary other shapes is based on closest point calculations. The other shape has to be composed of only line segments. Therefore, for intersecting a free shape and a circle or ellipse, the free shape has to be approximated by a polygon. The same applies to the case where two ellipses or a circle and ellipse are intersected. One important part of the algorithm is calculating the closest point to a given point on a line segment. This is done by projecting the point onto the line defined by the endpoint of the segments. In some cases, the projected point is not directly on the line segment, but left or right of the segment. Then, the demanded closest point on the line segment is the closer endpoint of the segment. The exact calculation can be found in chapter 5 of [9]. Figure 3.8 shows the results of projecting a point C onto a line segment and then choosing the correct point if the projected point is not on the line segment itself.

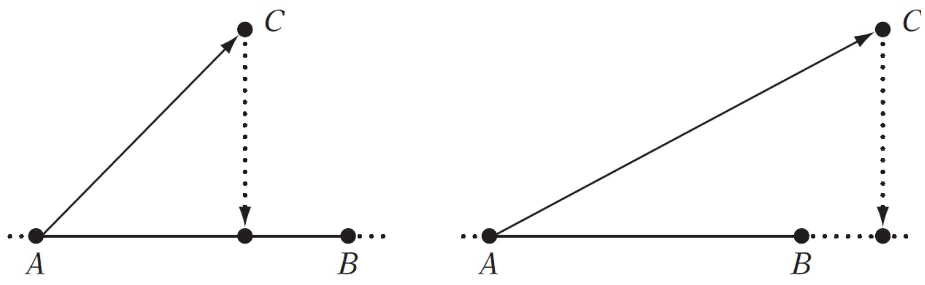


Figure 3.8.: Closest point on a line segment, see [9]

The algorithm for intersection testing of circles and other shapes works as follows. First, the other shape is approximated by a polygon. This is done by a method provided by Java2D that flattens a path in a way that the newly introduced vertices are never more than a given distance away from the original path. This results in a path solely consisting of line segments. The next step is to iterate over all those line segments. For each line segment, the closest point on that line segment to the center of the circle is calculated. The procedure simultaneously keeps track of the point of all closest points with minimum distance to the center. After iterating through all segments the point with minimum distance to the center is exactly the closest point on the other approximated shape. If that point is inside the circle, the circle must intersect the line segment that point is located on. Therefore the last step consists of checking if the distance from found point to the center is less than the radius of the circle.

The procedure above has to be slightly modified for ellipses. Since ellipses do not equally spread from one point like circles, but are defined by two different radii, just calculating the one closest point could result in a point that is in fact closer to the center of the ellipse, but is not inside the ellipse while a different point inside the ellipse is further away. Therefore, the procedure for ellipses also iterates trough all segments, but saves all points within the distance of the bigger one its two radii. Then, all found points are tested for containment. In addition to this modification non axis–aligned additionally require an extraction of radii and center point.

The second algorithm also involves an approximation for shapes that are not already polygons. It is an algorithm for intersecting two polygons. The inspiration for it comes from an algorithm for finding intersections of an arrangement of line segments in a plane. It was first proposed by Michael I. Shamos and Dan Hoey in [22]. The first step of this algorithm is exactly the same as for the previous procedure. The only difference is fthat both shapes are converted into polygons. Next, the both sets of line segments are sorted into a list of line segments. The criterion used for sorting is the smaller point of the two points defining a line segment. Smaller denotes a smaller x value in this case. Altogether this results in two lists of line segments each starting with the segment that contains the leftmost point of that shape as an endpoint. Then, the first list of segments is traversed by iterating though the second list of segments for each segment while looking for intersection between line segments. In order to avoid performing intersection tests for line segments that can not intersect due to their position, intersection testing is only performed for segments where at least part of the segment is located within the x–coordinate interval of the current segment from the first list. Figure 3.9 shows an arrangement of line segments. The blue segment the segment that intersection is checked for at that point. The x–coordinate interval of this segment is represented by the rectangle ranging from one end point to the other. If an intersection of line segments is detected, the overall intersection test is positive. Additionally, two other points are taken into account when intersecting two line segments. The line intersection test provided by Java2D does not only return true if an intersection point can be found, but also when the line segments are coincident or when a line segment contains an endpoint of the other one. In both cases, that could mean that the two polygons only touch in one segment or vertex. Therefore, each segment intersection test also tests for the two special cases. If one of the cases is found, the intersection continues to search for explicit intersections, but keeps in mind that a touching point or segment has been found. If none of the following tests is positive, the overall result will still be positive, even though the polygons could only touch from the outside.

The procedure above is used for intersecting all polygonal shapes, thus polygons and rectangles that are not axis–aligned, but also free shapes. A separate testing algorithm for free shapes would have to involve the intersection of bézier segments. Even though various algorithms for the intersection of bézier curves are available, e.g. beziér clipping [21] or the beziér subdivision algorithm [16], all of them are somehow based reducing bézier curve intersection to line intersection. This is exactly what the algorithm above does. Therefore, a different algorithm for free shape would only use a different approximation technique. Alternatively, the one already provided can be used, resulting in the same procedure for intersection testing of polygons, rectangles and free shapes with any other shapes.

The only test not covered so far is intersection testing for combined free shapes and any

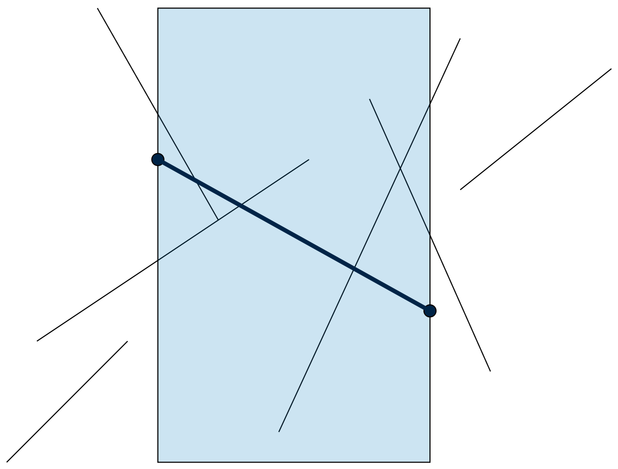


Figure 3.9.: Arrangement of line segments and part scanned for one line segment

other shape. This is done by simply iterating though all shapes the combined free shape consists of, looking for an intersection with the other shape separately.

Containment

An exact test for finding out if one Shape2D completely contains a different Shape2D seems to be a difficult task compared to intersection testing. Literature on this topic is rare and mostly focusses on polygons. Therefore, containment testing for this representation uses a rather simple approach.

One exception is the containment test for two circles. If a circle A contains a circle B can easily be checked in two steps. First, it is tested if A contains the center of B. In case of a positive result, it needs to be checked if A contains the whole range of B. This is done by comparing the distance between the two center points to the result of subtraction the radius of B from the radius of A.

All other shapes use an approach based on approximation the potentially contained shape as a polygon and point containment. The concept is the same for all shapes and only differs in the point containment test used by the respective shape. For two arbitrary shapes A and B and testing if A contains B the calculation steps are as follows. First, a bounding box containment test for A and B is conducted to avoid unnecessary calculations. If this test is positive, B is approximated by a polygon except when it already is a polygon. Then all vertices of the polygon are extracted. The next step is to test if shape A contains the extracted vertices. Testing if a shape contains a point is done differently for each shape. For circles, checking if the distance between center and point is smaller than the radius suffices. The point containment problem for axis–aligned ellipses can be reduced to the circle case by transforming the ellipse. For axis–aligned rectangles the x and y coordinate of the point have to be within the interval. In all other cases the result can be determined by using a ray casting algorithm [23], [26, pp. 41–67]. Figure 3.10 shows the basic principle for a polygon and two points P and Q. A horizontal ray originating at the respective point is casted to the right. Every time the ray crosses a segment, a counter that has been set to 0 at the beginning, is incremented. If the result is an even number the containment test yields a negative result, otherwise it is positive.

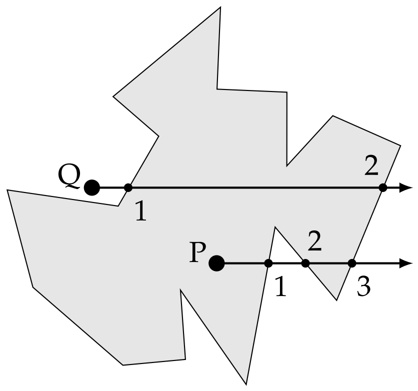


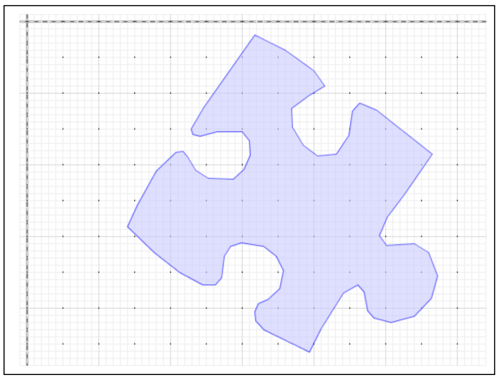
Figure 3.10.: Point–in–polygon testing by ray casting

Another exception are general shapes in their combined version. In their case, even the approach above is not applicable for determining if a combined shape contains an arbitrary other shape. Therefore, the combined version of free shapes uses bounding box containment in order to provide at least hint for the exact result.

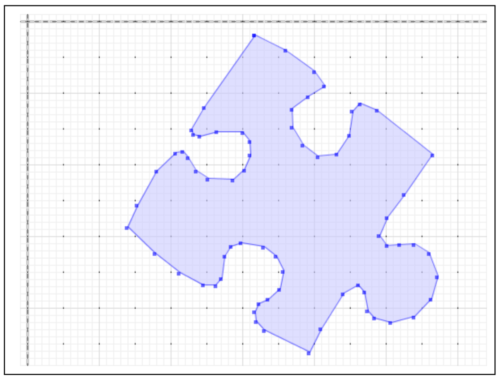
3.3.3. Geometry

The specialty of this representation is that all shapes are approximated by polygons. That way, none of the shapes needs special treatment leading to an overall simplification of all methods. Since shapes not purely made up of line segments have to be approximated in some way to become a polygon, shapes in this representation look slightly different from the other two vector based representations. However, the approximation is still exact enough that the human eye can barely distinguish the polygon from the original curved shape.

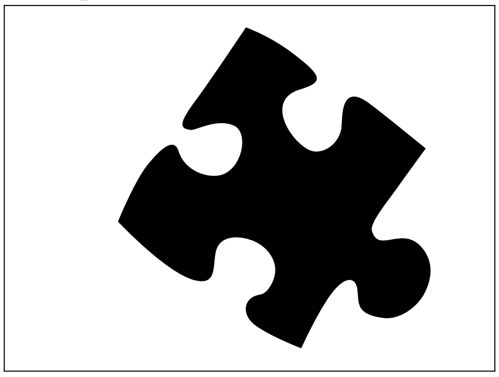
Geometry is a class that is part of the library JTS Java Topology Suite. More information on this library can be found in Chapter 4 as well as in the Technical Specifications Document of the library [24].



(a) Geometry without points



(b) Geometry with points



(c) Shape2D representation

Figure 3.11.: Screenshot of a geometry with points and without points and the Shape2D representation of the same shapes

Transformation

The Transformation of a Geometry is very similar to the one described in the previous sections about transformation of vectorbased shapes. Since a Geometry is entirely defined by its vertices, the transformation of a Geometry is simply the transformation of these vertices.

Intersection and Containment

Intersection and containment tests for two geometries are both based on calculating the Dimensionally Extended 9-Intersection Matrix (DE-9IM) of two geometries. This matrix contains the dimensions of the resulting geometries when interior, boundary and exterior of two geometries are intersected pair-wise. The result of an intersection or containment test is based on the values within the matrix. The DE-9IM was first introduced by Zhong, Jing, Chen and Wu in [29]. The matrix refines the 9-Intersection Matrix, which was first defined by Egenhofer and Herring [8].

In general, intersecting two geometric objects of at most two dimensions can only yield a limited number of different geometric objects. The function $dim(x)$ below defines the dimension of x for the four cases occurring in intersection for objects with two or less dimensions. x stands for the result object of the intersection.

$$dim(x) = \begin{cases} -1 \text{ (or F)} & \text{if } x = \emptyset \\ 0 & \text{if } x \text{ contains only points} \\ 1 & \text{if } x \text{ contains at least one line and no polygons} \\ 2 & \text{if } x \text{ contains at least one polygon} \end{cases}$$

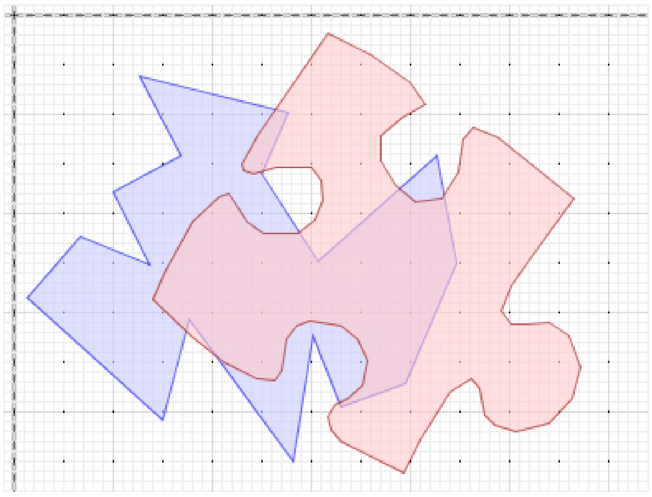
The definition of the (DE-9IM) for two geometries X and Y is defined as follows:

$$\text{DE-9IM}(X,Y) = \begin{pmatrix} dim(\mathrm{I}(X) \cap \mathrm{I}(Y) & dim(\mathrm{I}(X) \cap \mathrm{B}(Y) & dim(\mathrm{I}(X) \cap \mathrm{E}(Y) \\ dim(\mathrm{B}(X) \cap \mathrm{I}(Y) & dim(\mathrm{B}(X) \cap \mathrm{B}(Y) & dim(\mathrm{B}(X) \cap \mathrm{E}(Y) \\ dim(\mathrm{E}(X) \cap \mathrm{I}(Y) & dim(\mathrm{E}(X) \cap \mathrm{B}(Y) & dim(\mathrm{E}(X) \cap \mathrm{E}(Y) \end{pmatrix} \tag{3.1}$$

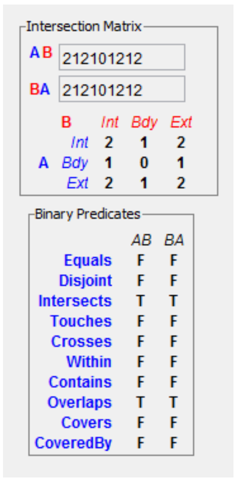
I(A) denotes the interior, B(A) the boundary and E(A) the exterior of a geometry A. Within JTS, determining the nine values is based on building a topology graph [24]. Topology graphs express the relationship of nodes and lines of one geometry to a different geometry. The graph contains all nodes and lines of both geometries as well as all intersection nodes between the lines of the input geometries. Nodes and lines are then labeled with their relationship to both geometries. For topology graphs with two geometries every part of the graph has one label for each geometry. Labels of a node, for example, have one value from the set $\{Interior, Boundary, Exterior\}$. Assuming the topology graph for two disjoint geometries X and Y is build and N is vertex of X than the labels of N would be $Boundary$ for X and $Exterior$ for Y. Once the whole topology graph is build, its labels are used to determine the values of the DE-9IM.

Intersection and containment testing for two geometries X and Y both start with calculating the DE-9IM matrix for the two geometries. Then, it has to be determined if the matrix matches a certain pattern. For a positive intersection test the matrix has to match one of the following patterns:

$$\begin{pmatrix} T & * & * \\ * & * & * \\ * & * & * \end{pmatrix}, \begin{pmatrix} * & T & * \\ * & * & * \\ * & * & * \end{pmatrix}, \begin{pmatrix} * & * & * \\ T & * & * \\ * & * & * \end{pmatrix}, \begin{pmatrix} * & * & * \\ * & T & * \\ * & * & * \end{pmatrix}$$



(a) Geometry without points



Intersection Matrix

AB 212101212

BA 212101212

	B	Int	Bdy	Ext
	Int	2	1	2
A	Bdy	1	0	1
	Ext	2	1	2

Binary Predicates

	AB	BA
Equals	F	F
Disjoint	F	F
Intersects	T	T
Touches	F	F
Crosses	F	F
Within	F	F
Contains	F	F
Overlaps	T	T
Covers	F	F
CoveredBy	F	F

(b) Result

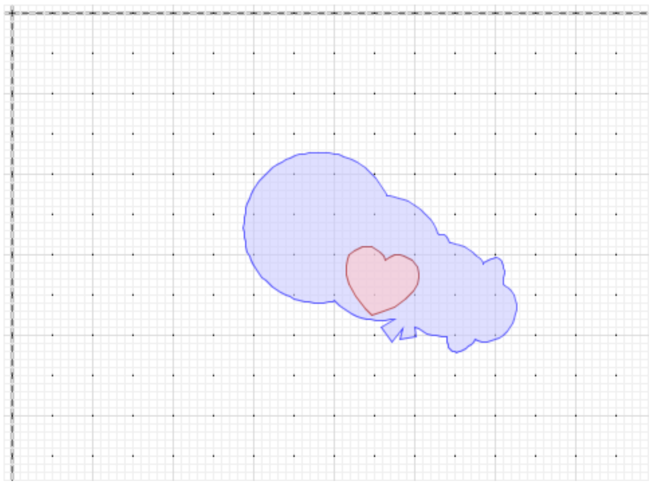
Figure 3.12.: Picture of a jigsaw piece intersecting a concave polygon and the result of the DE-9IM calculation

T denotes that this element has to be $\in \{0, 1, 2\}$, $*$ means that this element can have an arbitrary value. An example for a positive intersection test is given in Figure 3.12.

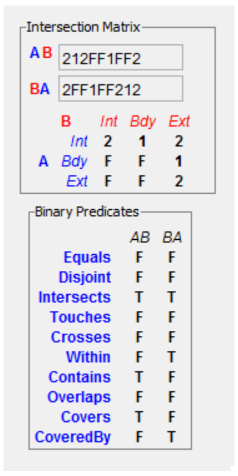
The pattern for a positively testing if X contains Y looks as follows:

$$\begin{pmatrix} T & * & * \\ * & * & * \\ F & F & * \end{pmatrix}$$

Figure 3.13 illustrates an example for a positive containment test.



(a) Geometry without points



Intersection Matrix

AB 212FF1FF2

BA 2FF1FF212

A \ B	Int	Bdy	Ext
Int	2	1	2
Bdy	F	F	1
Ext	F	F	2

Binary Predicates

	AB	BA
Equals	F	F
Disjoint	F	F
Intersects	T	T
Touches	F	F
Crosses	F	F
Within	F	T
Contains	T	F
Overlaps	F	F
Covers	T	F
CoveredBy	F	T

(b) Result

Figure 3.13.: Picture of a snowman containing a heart and the result of the DE-9IM calculation

4. Implementation

This chapter contains two sections concerning implementation details. The first section explains the structure of the whole program: the representations and the infrastructure needed for comparing representations. The second section contains information about the libraries that have been used within both representations and testing. Additionally, an overview on used and added functionalities is given.

4.1. Structure of the Program

The previous chapter introduced five different representations for general geometric shapes. In order to compare these representations with each other and to test implementations in a structured way, additional classes are introduced. Every newly added class either relates to one of the basic SVG shapes or a version for free shapes. Altogether, the following five classes are added: CircleObj, EllipseObj, FreeShapeObj, PolygonObj and RectangleObj. Each class contains an instance of the five representations. CircleObj for example comprises an Area, a CircleShape which is the Shape2D representation of a circle, a Geometry and the two pixelbased representations PixelImage and PixelImageEnhanced. The super class of the five classes is the abstract ShapeObj. It summarizes all methods commonly used by the five classes. The commonly used methods are the same methods that each of the representations is required to have. However, instead of executing a method for just one representation, the methods in ShapeObj calculate the results for all five representations simultaneously.

Initializing the classes inheriting from ShapeObj sometimes requires a conversion between representations to ensure that all representations actually represent the same geometric shape. For example, PixelImage and PixelImageEnhanced are usually initialized by converting the Shape2D or Area representation. All conversions are handled by a class named Conversion. This class also supports multiple saving methods, like saving a PixelImage as a PNG, a Shape2D or Area as an SVG or a Geometry as its textual description. The class Conversion is part of a package including multiple other classes used by ShapeObj or one of the representations. Transformation, Intersection and Conclusion contain everything related to the respective topic that is used by multiple classes. Transformation, for example provides methods for computing transformation matrices for all transformation methods. Thus each transformation within the representation includes a call to method of this class. The class Settings only contains the width and height of the image canvas. This ensures that canvas size and picture center are the same for all representations. Altogether, the program has the structure shown in Figure 4.1.

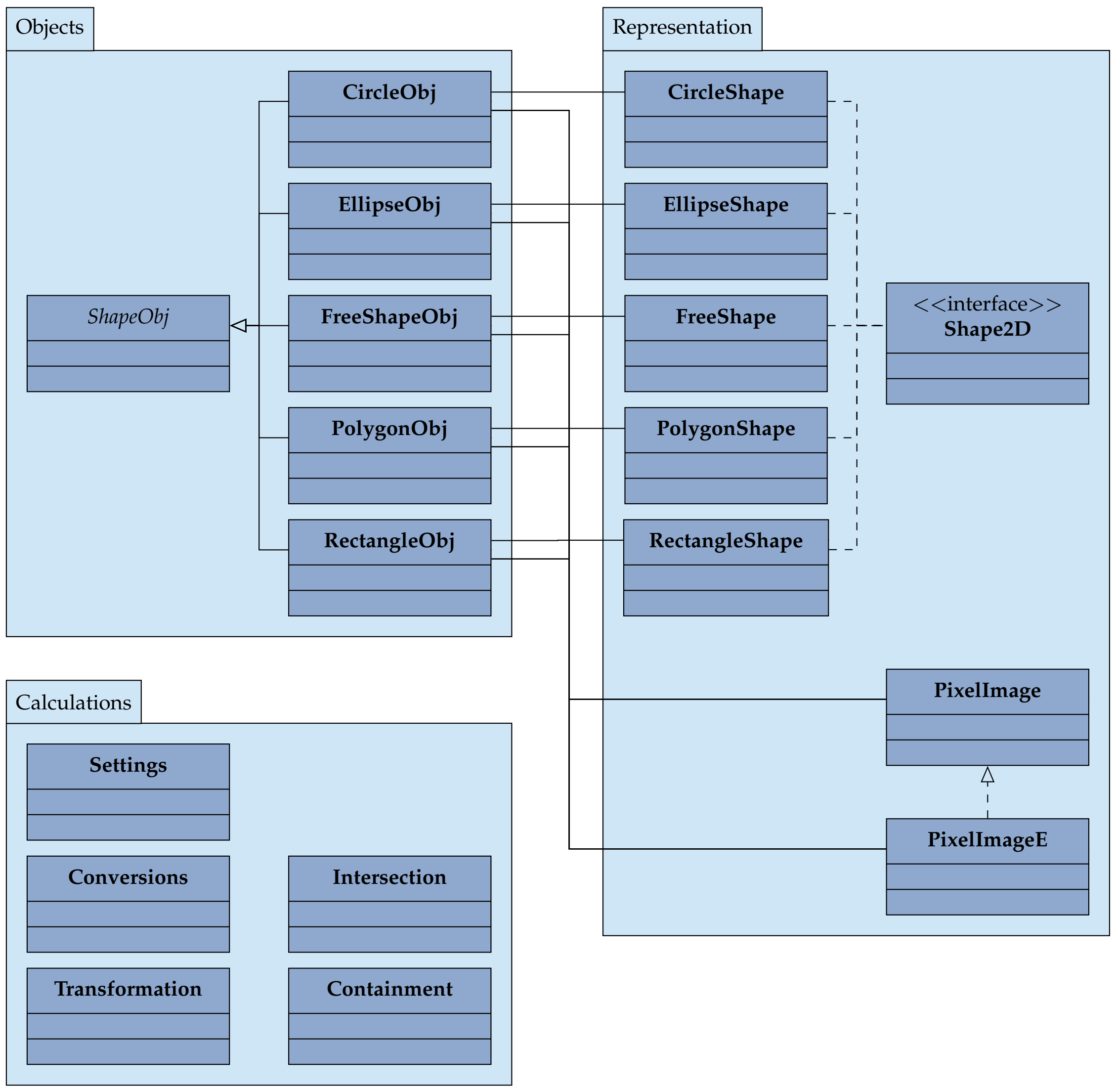


Figure 4.1.: Class diagram for the implementation

4.2. Overview of Used Libraries and Added Functionality

Implementing both representations and testing is partially based on libraries. In total, three libraries are used, two of which have been mentioned before. The third is mainly used in testing. This section contains a few notes on those libraries. Additionally Table 4.1 shows which functionalities have already been available within a library and which had to be implemented additionally to complete the required functionalities.

The library that has been used the most is Java2D. Java2D has originally been designed for displaying and drawing images and shapes, not to compute. Therefore, some methods are not accurate or display a behavior that does not affect drawing and displaying, but does affect transformations, intersection testing and containment testing. One example are the methods to calculate bounding boxes. They only guarantee to return an axis–aligned rectangle that fully contains the shape, but not that this rectangle is necessarily the tightest one. The reason is that for complex areas or shapes computation costs can be very high. Returning a slightly different bounding box obviously affects rotations around the object center, as the center can also be somewhere else. This makes it harder to compare the different representations, especially since it is not predictable how much the bounding box is different from the actual one or when this effect occurs at all. Another example is the combination of areas. When two areas are combined using one of the boolean combination methods like union or intersection, Area sometimes inserts unwanted vertices. There are two cases for unwanted vertices. In one case, a group of about four vertices located seemingly completely random is added. These vertices are not connected to the rest of the path, thus appended by a MoveTo command and are very close together. The other case are vertexes that appear within the path and are within the range of about 10^{-3} pixel away from another point. This is a known bug of Java2D, but has not been fixed yet. It mainly causes problems for the conversion of shapes. JTS, for example seems not able, to compute a polygon from points that close together. Additionally this could also be the explanation for other effects occurring with Area. These effects will be discussed in Chapter 5.

The JTS Topology Suite is a library that contains many fundamental spatial algorithms. The major goal of the design of JTS is to provide complete, consistent and robust implementations of algorithm that are also fast. Aside from testing if two geometries intersect or contain each other ,multiple other relation queries can be answered. Additionally, geometries in JTS are not limited to just polygons. It also provides methods for convex hull computations or geometric simplification and many other algorithms classified as computational geometry applications. This thesis uses only geometries containing one polygon, but JTS supports several other types of geometries. Amongst others, a Geometry can also represent a single point, multiple points, multiple connected line segments, multiple polygons or a mixture of all spatial data types.

Batik SVG Toolkit is the library that has not been mentioned before, but is also used within the implementation. This library is capable of displaying, generating and manipulating SVG graphics. It supports conversion from Area and Shape2D objects to SVG. Therefore it is used to save these in order to visualize internal procedures and check test results. An introduction on Batik and the way it can be used can be found in [1].

Representation	**Library**	**Usage**	**Added Functionality**	**Sources**
PixelImage	Java2D	Transformation of an image	Intersection with another pixel image, Containment of another image, Bounding box calculation	[9], [14]
Area	Java2D	Union of multiple areas, Transformation of an area, Intersection of two areas, Bounding box calculation	Combination of different methods to perform intersection and containment tests	[15]
Shape2D	Java2D	Transformation of path objects, Intersection of an arbitrary shape with a rectangle, Containment of a rectangle for an arbitrary shape, Containment of a point for an arbitrary shape, Flattening of a path	Special handling of transformations for circles, ellipses and rectangles, Extraction of points, lines and curves from a path, Intersection point of lines, Closest Point on a line segment, Test for touching and coincidence of line segments, Intersection test for multiple line segments	[9], [15], [16], [26], [21], [22], [23]
Geometry	JTS	Transformation of a geometry, Intersection with another geometry, Containment of another geometry	Creation of geometries by combining multiple methods of JTS, Conversion between different classes for affine transformations	[8], [18], [29], [24]

Table 4.1.: Usage of libraries and added functionality in each representation

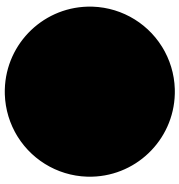
5. Evaluation

The following chapter evaluates and compares the implemented representations. It focuses mostly on the runtime and functionality aspects of transformation, intersection test and containment test methods. Additionally, a complexity measure is taken into account in order to determine if a correlation between runtime and complexity exists. The first section of this chapter describes the test setting, especially the shapes used in testing. The second section compares the three methods for bounding box calculation in pixel images. Subsequently, the results of runtime and functionality tests are shown. The chapter concludes with a discussion of the results.

5.1. Test Environment

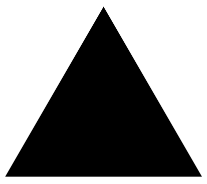
The data used for all tests is based on a fixed set of shapes. The set consists of twenty shapes that can be classified according to the different types of shapes in vector graphics. Ten of the test shapes represent one of the basic shapes, i.e either a circle, an ellipse, a rectangle or a polygon. The other ten shapes can not be represented by a single basic shape. Five of them are constructed by combining multiple basic shapes and the other five use a path of bézier curves. All test shapes are shown in Figure 5.1 with name and type noted below the respective shape. Types are abbreviated with B standing for basic, C for combined and P for path. The shapes are ordered by a complexity measure based on both the points needed to define the path of a shape and a shape's combination of basic shapes if available. The measure is explained elaborately in Appendix A.

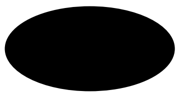
Each runtime test uses a certain number of randomly created instances, which will be mentioned within the section about the respective test. The instances are not drawn randomly from the set of shapes, but the number of instances is the same for every test shape and the randomness refers to size, orientation and position of an instance. Furthermore, tests are carried out separately for different object sizes. Test objects can be of either small, medium or big size. The classification into these categories is done by specifying the minimum and maximum area of an objects bounding box in relation to the total picture size, which is set to 640×480 pixels in all test cases. The following percentage intervals are used, to bound the size of shapes:

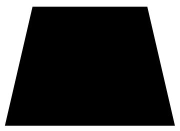

(a) A Circle (B)

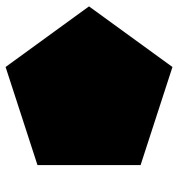
(b) A Square (B)

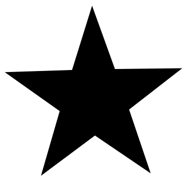
(c) A Rectangle (B)

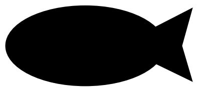

(d) A Triangle (B)

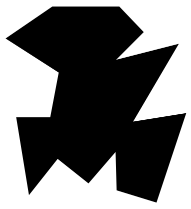

(e) An Ellipse (B)

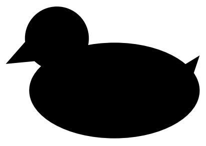

(f) A Trapezoid (B)

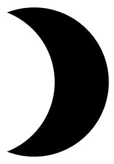

(g) A Pentagon (B)

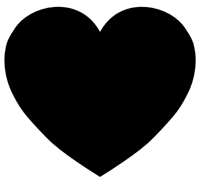

(h) A Star (B)

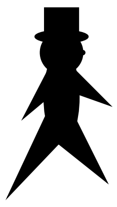

(i) A Fish (C)


(j) A Polygon (B)

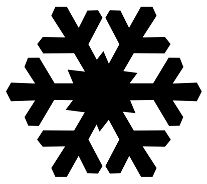

(k) A Bird (C)


(l) A Moon (P)

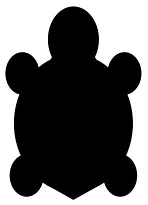

(m) A Heart (P)

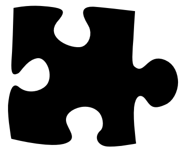

(n) A Man (C)

(o) A Snowman (C)


(p) A Snowflake (B)

(q) A Tree (P)


(r) A Turtle (C)


(s) A Jigsaw Piece (P)

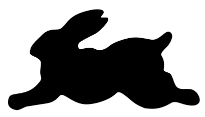

(t) A Bunny (P)

Figure 5.1.: Test shapes ordered by complexity

$$\%_{totalA} = \begin{cases} \in [1.3, 26.6] & \text{for small shapes} \\ \in [26.6, 66.6] & \text{for medium sized shapes} \\ \in [66.6, 100] & \text{for big shapes} \end{cases}$$

Each test object is first initialized with a version of the shape that is at most 100 pixels in both height and width. Then, a random scaling, a random rotation and a random translation are applied to the object. The scale factor depends on the chosen test size. The interval of possible factors is determined by calculating which factor would be needed to reach the smallest and biggest bounding box area allowed for the desired size. After the scale factor has been randomly selected from that interval and has been applied to the original object, the rotation angle can be generated without such restrictions. However, in some cases a rotation can lead to an object that does not fit the chosen image canvas. In such a case the object is reinitialized and the selection of scale factor and rotation angle is repeated. Since the maximum size of an object is restricted to be smaller than the image canvas this procedure will always find suitable parameters. Last, the object is translated in both x and y direction, ensuring that the translation does not lead to an object outside the image.

Once the desired number of objects are generated, the objects are saved in order to reuse them for all tests. The tests are run using a computer with *4 GB RAM* and an *Intel Core 2 Duo P8600* processor with *2.40 GHz*. Even when the number of running processes is kept at a minimum, runtime can still be influenced by other processes running on the computer. Therefore, every test was run multiple times using exactly the same data. Values that are more than three standard variations above the mean of one test series are rejected and not part of the final evaluation.

5.2. Comparison of Different Methods for Bounding Box Calculation

Section 3.2.1 explained three methods to calculate the bounding box of a geometric object represented by a pixel image. In oder to compare those methods, the bounding boxes for 1000 objects of each size are calculated. The average runtimes for those calculations are given in Figure 5.2. The results for each method are displayed separately. Figure 5.3 additionally shows how often the image data of an object has to be accessed for successfully finding the bounding box.

The runtime of Bounds1 and Bounds2 declines with increasing shape size, while the runtime of Bounds3 increases. The number of accesses to the image data display the same behavior, as do minimum and maximum runtime (see Tables B.1 and B.2). Altogether, both figures show that Bounds2 seems to be the best option for all sizes. Therefore, Bounds2 will be used for further testing.

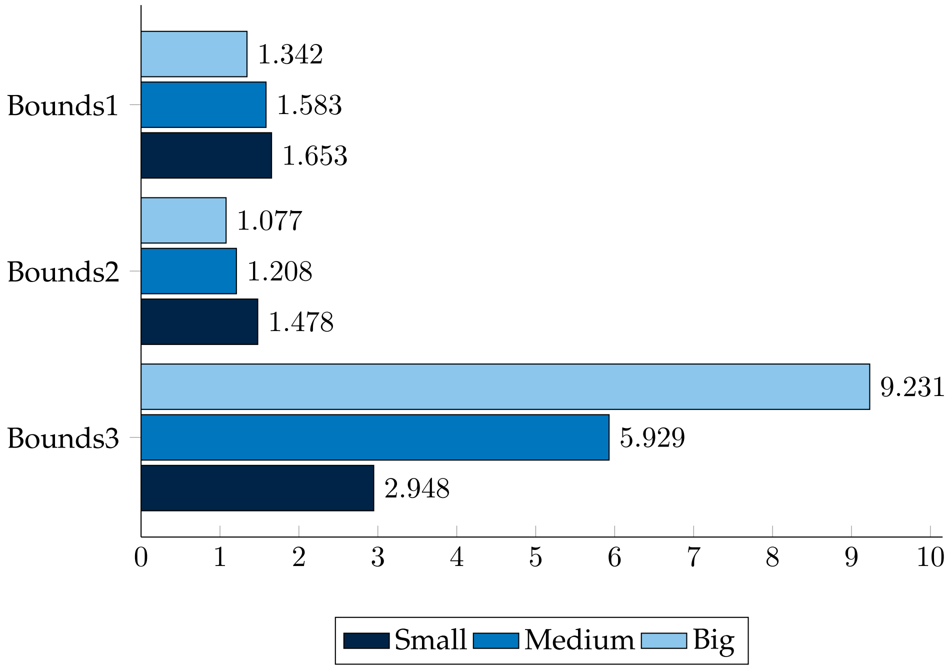


Figure 5.2.: Runtime of different bounding box calculations (measured in ms)

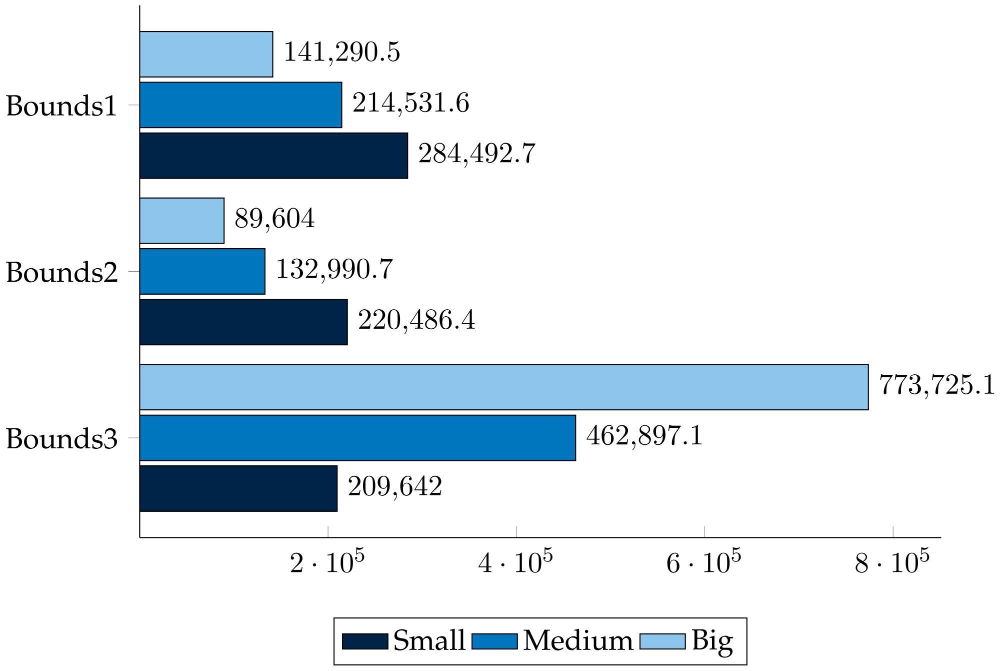


Figure 5.3.: Number of accesses to the image data

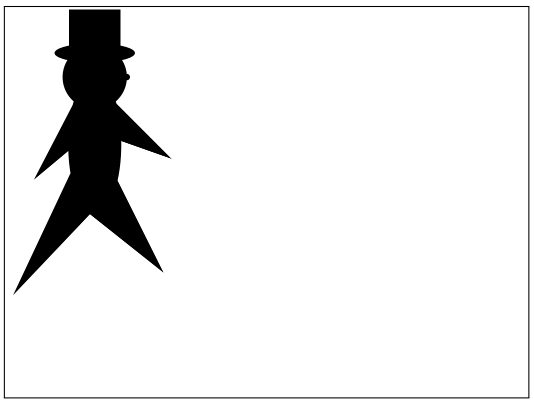
5.3. Comparison of Common Functionalities

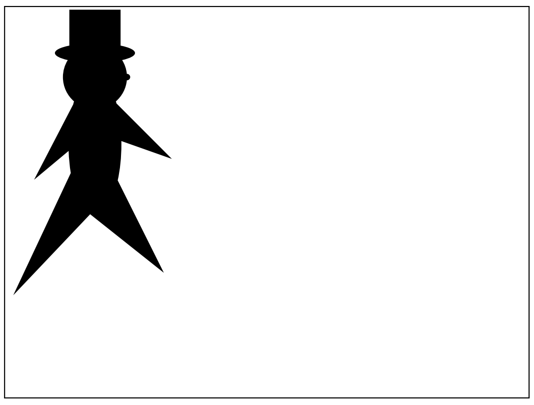
This section covers the evaluation of the three major functionalities provided by all five representations. The first subsection treats transformation, the second subsection intersection and the third containment. All three functionalities are tested using the same 200 objects of small, medium and big size. However, cookies to be arranged in one rolled out dough do not necessarily have to be the same size. In consequence, intersection and containment tests also include a test for objects of different sizes.

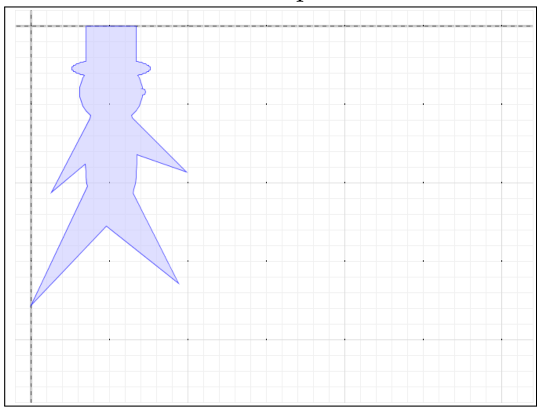
5.3.1. Transformation

Before comparing the transformation methods of the different representations in terms of runtime, it has to be shown that the methods actually yield the expected result for all representations. The following figures display the result of applying a sequence of all possible transformations to the representation of one of the test objects. The test shape Man (Figure 5.1n) contains all basic shapes and is therefore used as an example here. The representation PixelImageEnhanced has been omitted in the pictures since it is transformed the same way as PixelImage. Figure 5.4 shows the original image of one instance of the Man, located at the upper left corner of the Image. Figure 5.5 displays the Man after translating him by 10 pixels in x direction and 100 pixels in y direction. The following figure results from rotating the Man around his object center by 30 degrees. Figure 5.7 contains the images produced by scaling the Man by a factor of 1.2, using the center of his bounding box as the anchor point. Then, the shape is rotated around the picture center by 90 degrees. Last, the Man is scaled by 0.8 using the picture center as the anchor point. The result can be seen in Figure 5.9.

The runtime comparison for the transformation methods is based an applying various different transformations to the 200 test objects and measuring the elapsed time along the way. The test uses only translation, rotation around the object center and scaling around the object center. Rotation and scaling with the picture center as the fixed point are omitted since they only differ from the other rotation and scaling methods in the fixed point. First, a sequence of five translation is applied to all objects 25 times. The following translation vectors are used: $(-2, -2)$, $(4, 4)$, $(-5, 1)$, $(7, -7)$ and $(-4, 4)$. One iteration of the sequence is equal to the identity transformation, meaning that the position before and after the sequence are the same. Therefore, applying the sequence multiple times is essentially the same as reinitializing the objects and carrying out the test again. Then, the objects are rotated around their centers in steps of approximately 2.8 degrees ($\pi/64$ in radians). This is done 128 times, altogether leading to original and resulting position to be the same again. The third step is to repeatedly apply a sequence of scalings with scale factors 1.25, 0.4, 2.8 and 0.8. Last, the shapes are transformed by a mixture of transformations, rotations and scalings in arbitrary order.


(a) Shape


(b) Area


(c) Screenshot of the Geometry

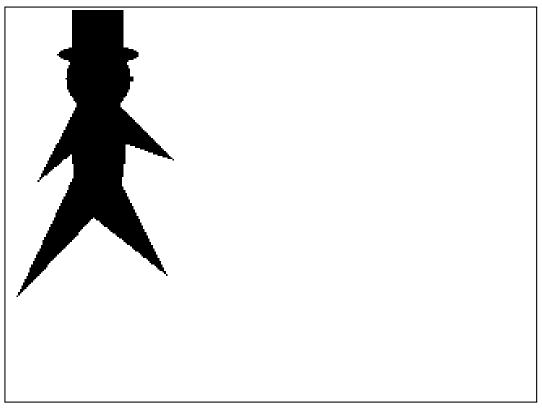
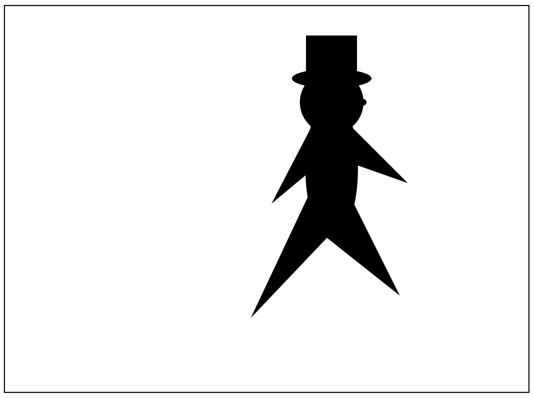
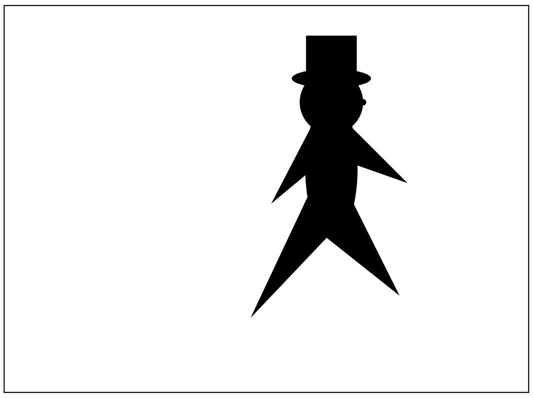
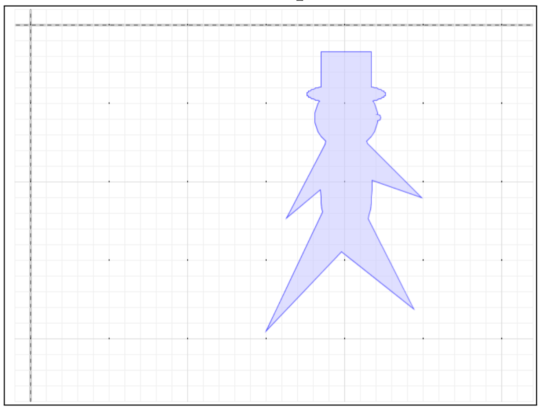

(d) Pixel Image

Figure 5.4.: Original Man in all representations


(a) Shape


(b) Area


(c) Screenshot of the Original Geometry

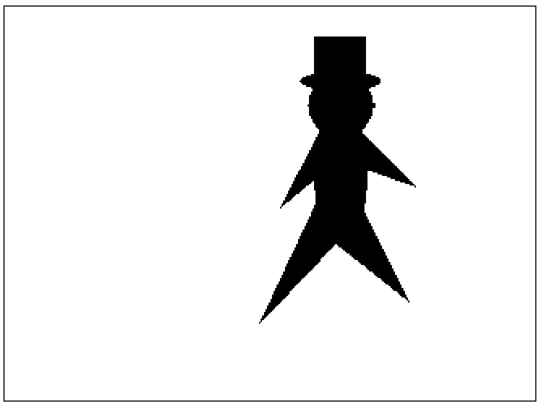
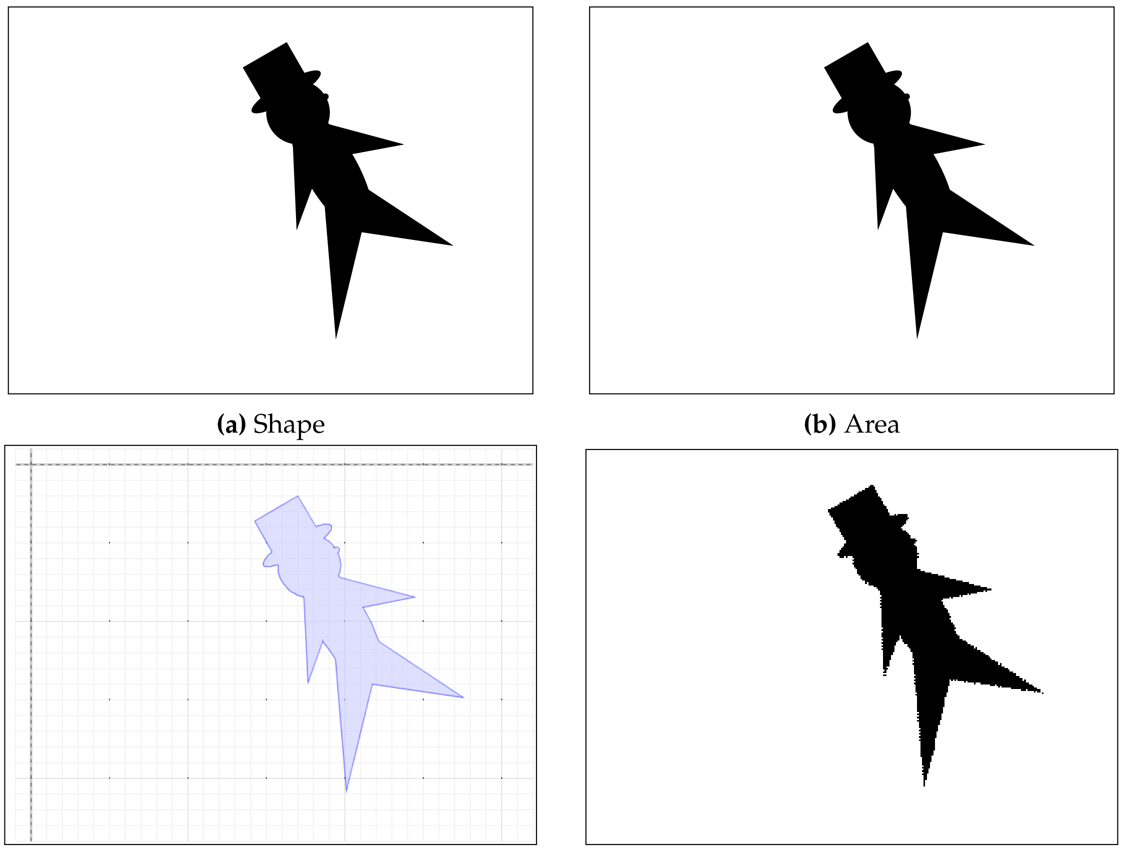

(d) Pixel Image

Figure 5.5.: Translated Man in all representations



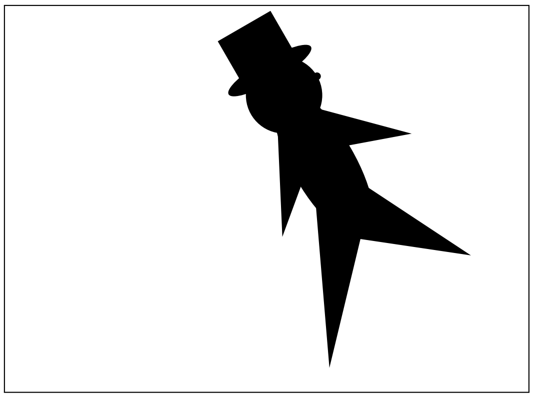
(a) Shape

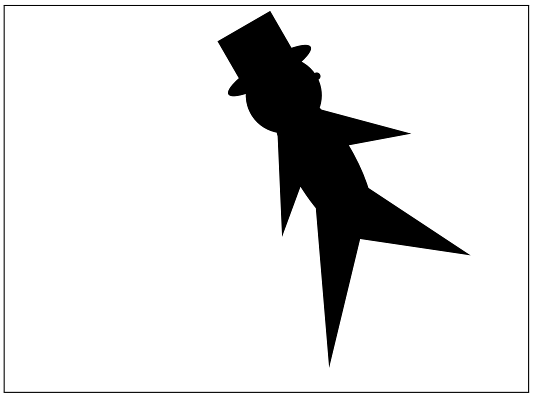
(b) Area

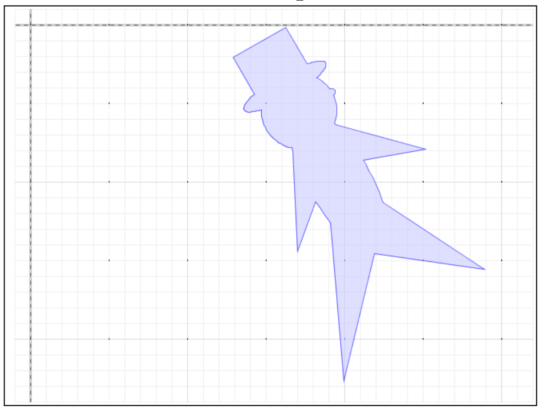
(c) Screenshot of the Geometry

(d) Pixel Image

Figure 5.6.: Man rotated around his object center in all representations


(a) Shape


(b) Area


(c) Screenshot of the Geometry

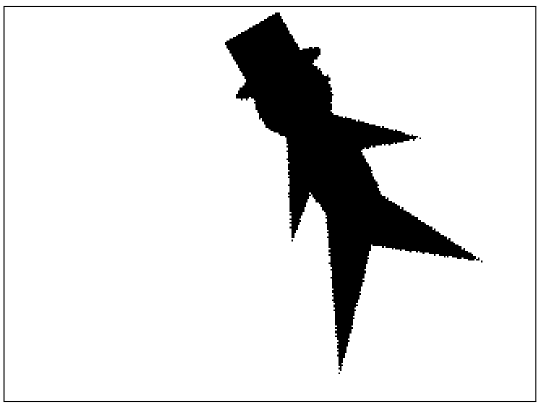
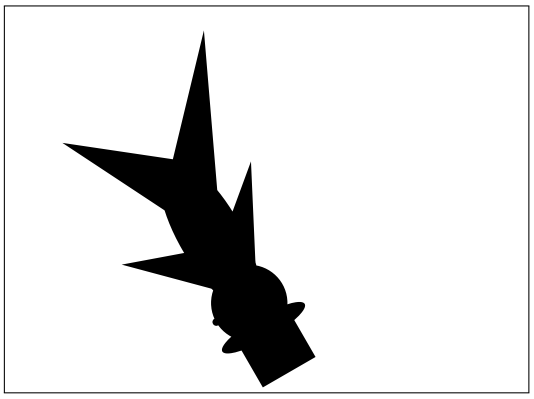
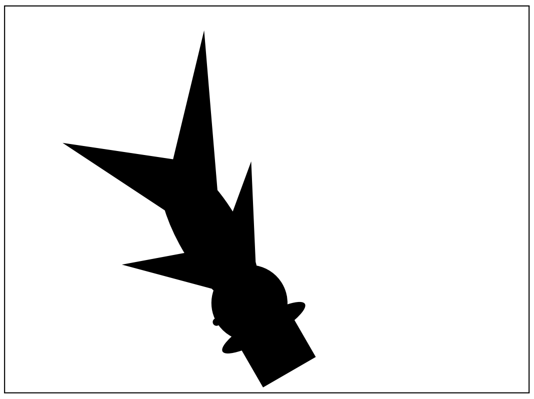
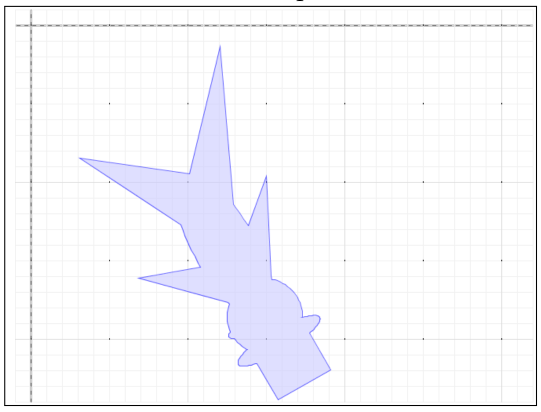

(d) Pixel Image

Figure 5.7.: Scaled Man with object center as anchor point in all representations


(a) Shape


(b) Area


(c) Screenshot of the Geometry

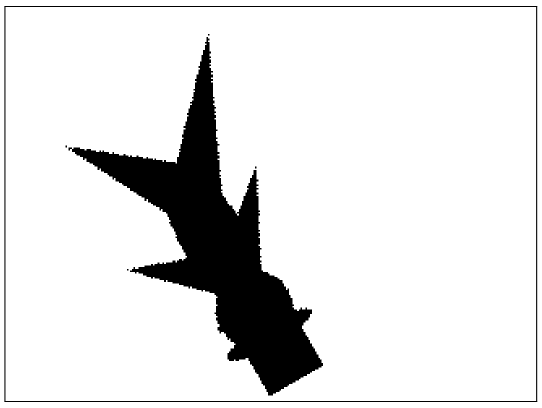
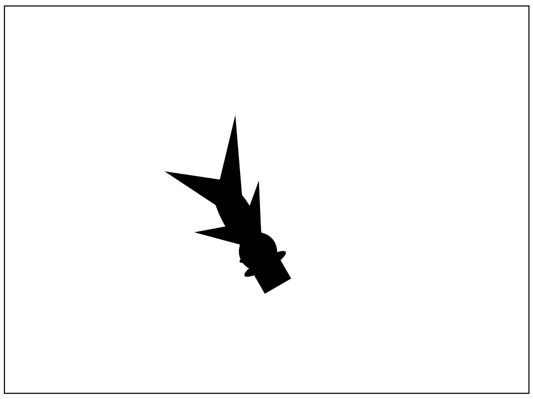
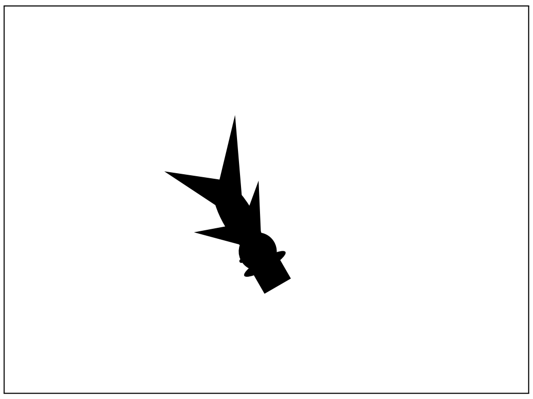
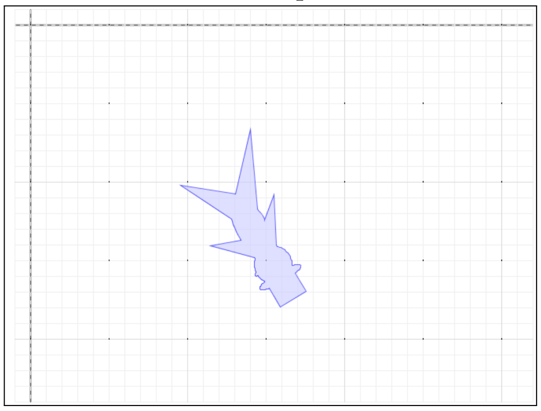

(d) Pixel Image

Figure 5.8.: Man rotated around the picture center in all representations


(a) Shape


(b) Area


(c) Screenshot of the Geometry

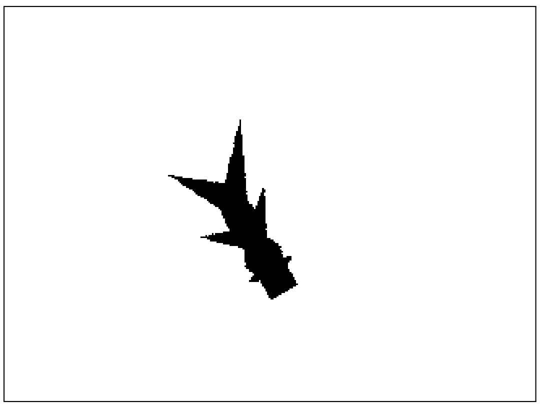

(d) Pixel Image

Figure 5.9.: Scaled Man with the picture center as anchor point in all representations

The following figures only display the runtime for objects of medium size, as the results are very similar. Tables B.3 – B.14 show the results for small and big sizes. The tables for medium sizes are also included in Appendix B because they additionally contain the runtimes for each test shape separately. Figure 5.10 shows the results for translating shapes. Shape2D includes combined shapes in their path representation and Shape C. contains the combined version. Therefore, results for Shape C. only exist for shapes that actually are combined. Shape2D and Shapes C. are the two fastest representations with only negligible differences between test shapes. The same applies to Geometry representations which can also be translated rather quickly. Area is a little slower and the only one where more significant differences between test shapes can be observed. Bunny, Jigsaw, Man, Snowflake, Snowman and Turtle need about four times longer than shapes like Circle, Rectangle or Star. The representations based on pixel images are the slowest with averages of almost 3 ms for PixelImage and just above 4 ms for PixelImageEnhanced.

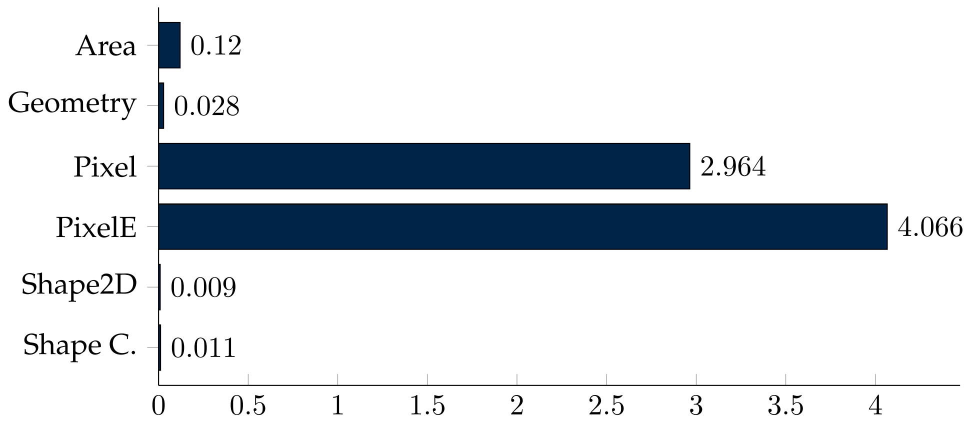


Figure 5.10.: Runtime of translating test shapes (measured in ms)

Figure 5.11 shows the differences in runtime between all representations for rotations. It has to be noted, that the value for Area is not the average value for all shapes. As can be seen in Table B.8 the average runtimes for Bird and Man are significantly higher than for all other shapes. The averages are that high due to four shapes where one rotation suddenly takes multiple seconds instead of milliseconds and the following rotations stay at that level. Therefore, the mean is also around those high numbers causing them to not being thrown out by the standard variation process. The same phenomena also occurs for rotating small and big shapes. For small shapes there are three such outliers, for big shapes there are two. The value used for Area in the picture is the one that can be derived when the four shapes with odd behavior are omitted. Then, the results are approximately the same as for the translation. The order from fastest to slowest are equal, as is the diversity of runtimes for Area. Only the difference between pixel and vector representations is much bigger and the two pixel representation do not differ as much.

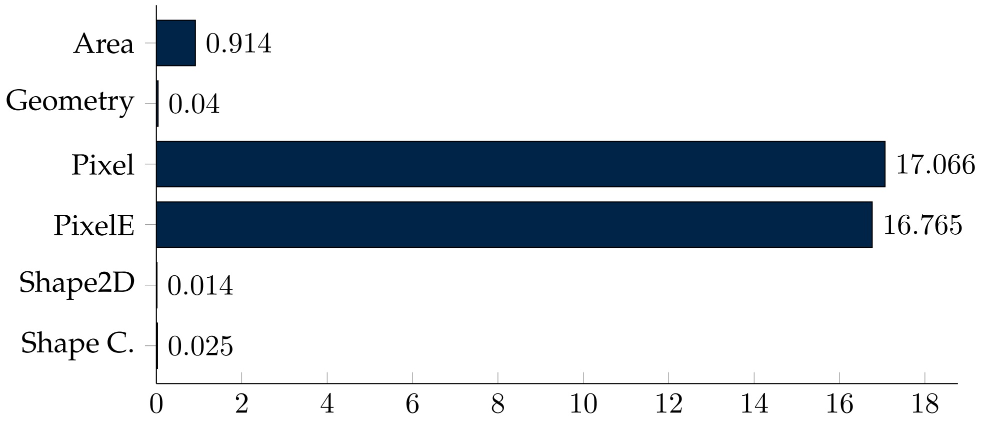


Figure 5.11.: Runtime of rotating test shapes (measured in ms)

The same differences as in translation and rotation can be witnessed in scaling (Figure 5.12) and mixed transformation (Figure 5.13) as well. In conclusion, the transformation of Shape2D is the fastest, followed by Geometry and then Area. PixelImage and PixelImageEnhanced are the slowest representation and need nearly the same time, except for the translation case. The only representation where runtime seems to depend on the type of shape is Area.

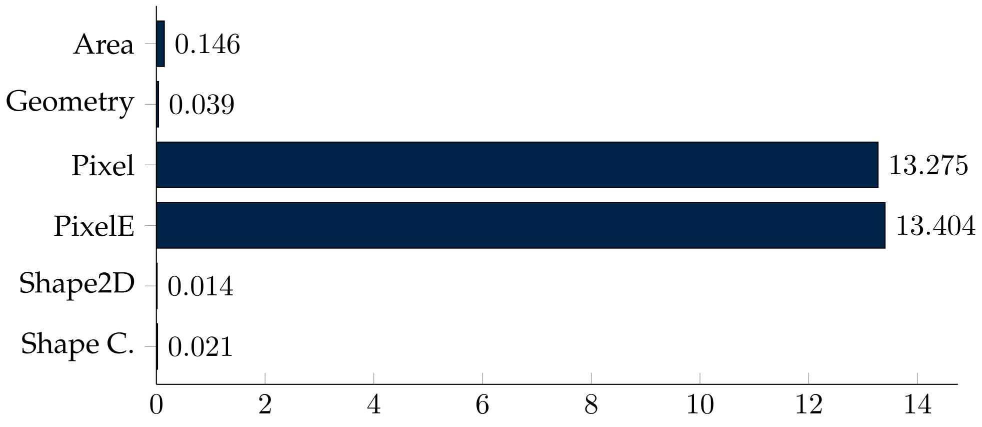


Figure 5.12.: Runtime of scaling test shapes (measured in ms)

5.3.2. Intersection

The intersection test data uses the same 200 test shapes that have already been created for testing the transformation methods. The 200 objects are intersected with each other leading to $40,000$ intersections per size category in total. The figures displayed in this section

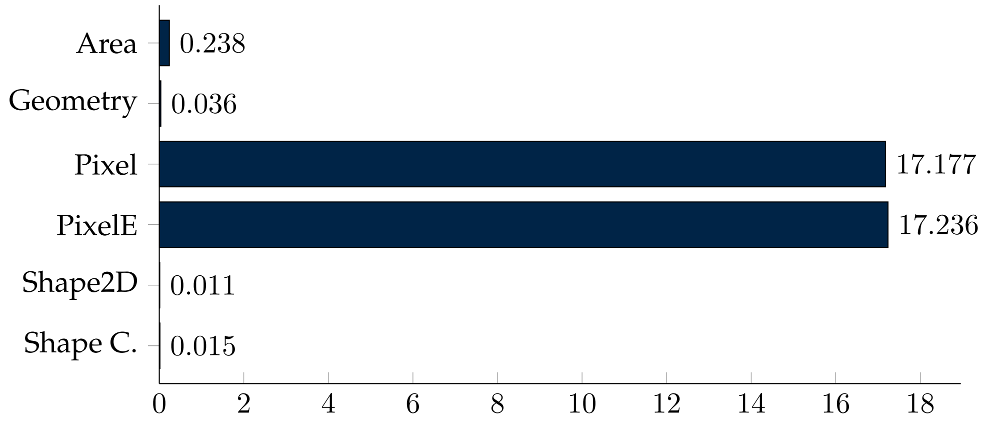


Figure 5.13.: Runtime of applying arbitrary transformations (measured in ms)

only show the results for mixed shapes since in that case the ratio between positive and negative intersection tests is the closest to being balanced. For small shapes, the majority of the shapes do not intersect. For medium the opposite holds true and for big shapes there almost do not exist any negative intersection tests. The test results for all sizes can be found in Tables B.15, B.17 (small), B.19, B.21 (medium), B.23, B.25 (big), B.27, and B.29 (mixed). Figure 5.14 illustrates the intersection test runtimes. The table includes a result for only using the bounding box of the two shapes. This result is denoted by Shape. The results for combined shapes are shown separately in Figure 5.15.

Intersecting just the bounding boxes of the Shape2D representation is the fastest intersection test, followed by Geometry, Shape2D and Area with only little differences between them. Again the two pixel representations are the slowest. PixelImageEnhanced is about 2–3ms faster than PixelImage. Additionally, intersection for shapes that can be combined by basic Shape2D objects is faster than intersection of their path versions. Generally, determining that two shapes do not intersect is faster than the opposite case. Analyzing the results of one individual representation shows that for Area the same variations as found for transformation also apply for intersections. Additionally, for Geometry the value for Snowflake is slightly higher than all other values, but the difference is still not very big. In contrast to the result of transformation tests, Shape2D and Shape C. show approximately the same variation as Area does. Both pixel image representations as well as the bounding box test do not depend on the type of the shape. Table 5.1 shows how many intersection tests have been positive for each shape. Since Shape uses only bounding box containment, its value obviously has to be higher. Altogether, there were $26,010$ cases were all representations agreed on intersection. In $11,004$ cases all representations agreed on non-intersection and in $2,638$ cases only bounding box intersection tests were positive.

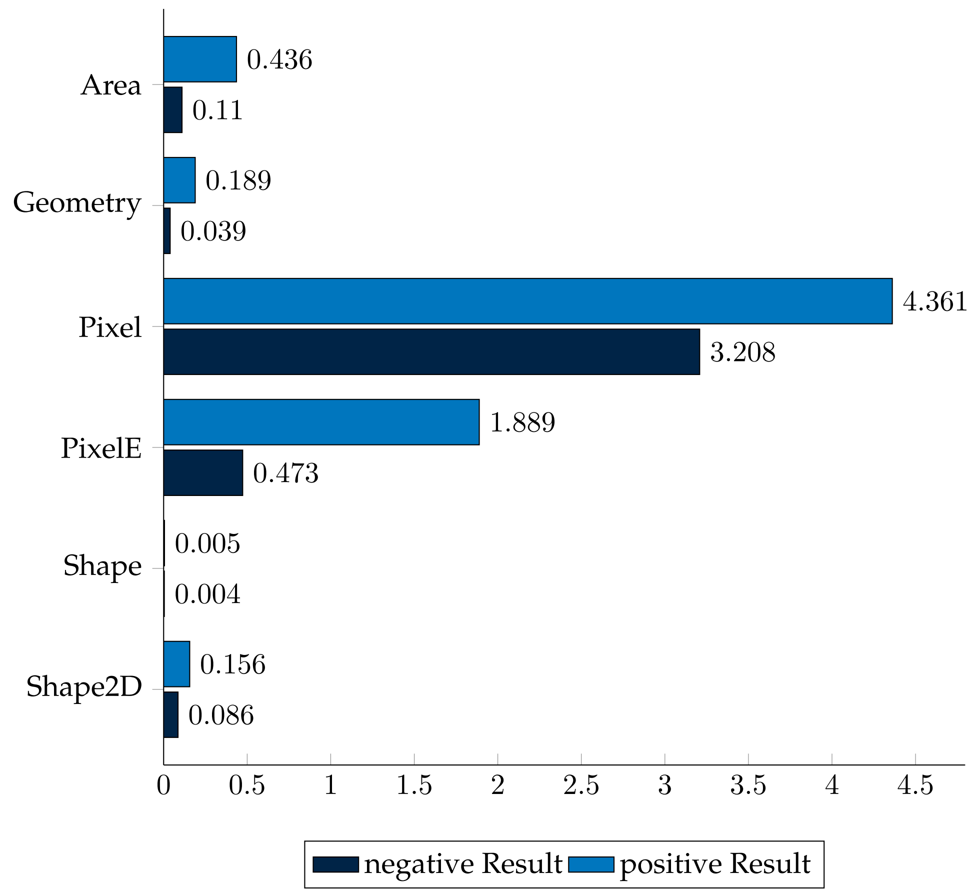


Figure 5.14.: Runtime of intersection tests (measured in ms)

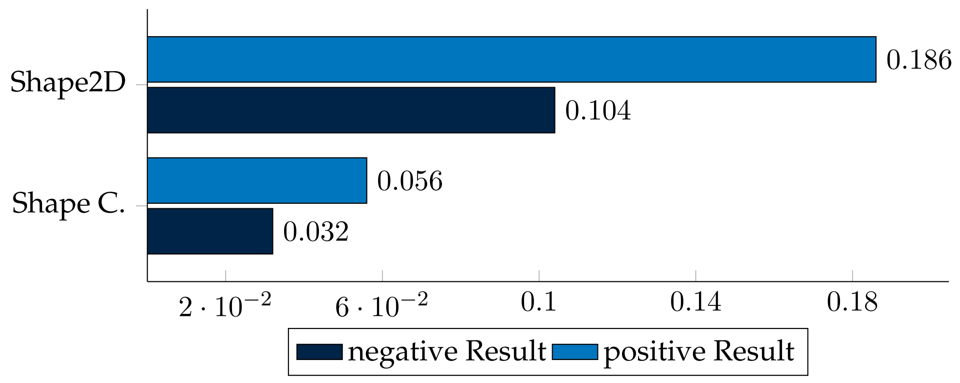


Figure 5.15.: Runtime of intersection tests limited to combined shapes (measured in ms)

Shape:	**Area**	**Geometry**	**Pixel**	**PixelE.**	**Shape**	**Shape2D**	**Shape C.**
Nr:	$26,316$	$26,284$	$26,264$	$26,264$	$28,995$	$26,306$	$11,116$

Table 5.1.: Number of positive intersection tests for each representation

5.3.3. Containment

Just as intersection testing before containment testing reuses the same 200 test shapes from transformation testing. The test traverses all 200 objects and for each object checks if it contains an object of the same set of 200 objects. This leads to $40,000$ containment tests. Similar to the section about intersection testing, the figures displayed in this section only show the results for object of mixed size. The main reason is that positive results for for shapes of roughly the same size are very unusual. For big objects, for example around 200 containments are positive for all representations except the bounding box containment test, which yields a higher number of positive results. However, these 200 results are mostly made up by tests for self containment. The same observation can also be made for objects of small and medium size. The test results for all sizes can be found in Tables B.16, B.18 (small), B.20, B.22 (medium), B.24, B.26 (big), B.28, and B.30. Figure 5.16 contains the runtime result for containment tests. The results for combined shapes are illustrated separately in Figure 5.17

The results are partially very similar to the ones for intersection testing. Testing for bounding box containment is faster than anything else and is followed by Shape2D and then Geometry. In contrast to intersection testing a visible difference between Shape2D and Geometry exists for positive test results. Additionally the difference between Area and the other representations is much bigger for Shape2D, Geometry and Shape. Area almost reaches times that are needed to determine containment for two PixelImageEnhanced objects. Tests for PixelImageEnhanced are much faster than the ones for PixelImage, especially if the test result if negative. As for comparing the runtimes of different shapes of one individual representation, Area displays the same behavior as in all other methods before. Geometry has two shapes with slightly higher runtimes. These are Man and Snowflake. All other representations show no significant deviation. Table 5.2 shows how many containment tests have been positive for each shape. Since Shape and Shape C. use only bounding box containment, the numbers obviously have to be higher. Altogether, there were 580 cases were all representations agreed on containment. In $36,642$ cases all representations agreed on non-intersection and in $2,446$ cases only bounding box intersection tests were positive.

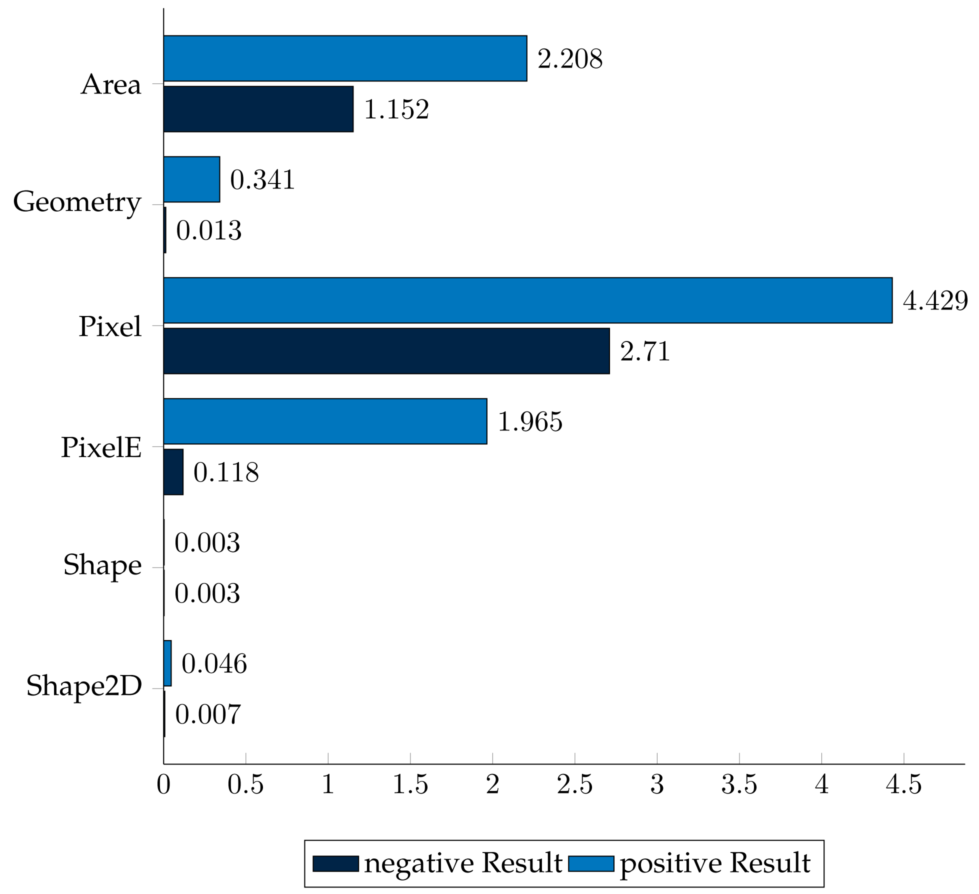


Figure 5.16.: Runtime of containment tests (measured in ms)

5.4. Discussion

This section discusses reasons for the results illustrated in the previous sections.

The results for calculating the bounding box of a pixel image showed that Bounds1 and Bounds2 get faster with increasing object size, while Bounds3 gets slower. This can easily be explained by looking at the parts of an image each of these methods has to scan. Bounds1 and Bounds2 mostly scan the part of the image that does not contain black pixels. Therefore, with increasing object size and thereby increasing number of black pixels the portion of the image that has to be scanned declines. The opposite holds for Bounds3. Bounds3 scans the part of the image that does contain black pixels. Thus, for big object, almost the whole image has to be scanned.

The results for transforming, intersecting and checking containment for PixelImage and PixelImageEnhanced exactly fulfill the expectations. As translating is the only method

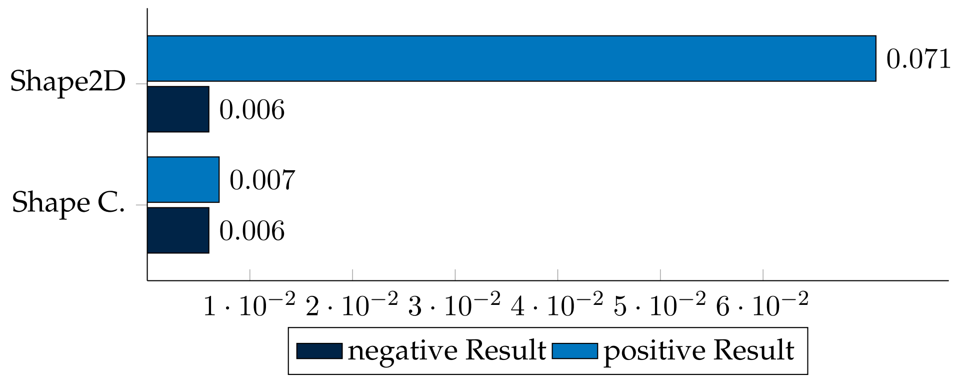


Figure 5.17.: Runtime of containment tests limited to combined shapes (measured in ms)

Shape:	Area	Geometry	Pixel	PixelE.	Shape	Shape2D	Shape C.
Nr:	781	825	845	845	3,338	669	1050

Table 5.2.: Number of containment tests for each representation

where PixelImage does not have to calculate it bounding box, it is about $1ms$ faster, approximately the time needed to calculate a bounding box. For the other two presented transformation methods the bounding box of the PixelImage has to be calculated in order to derive the object center. PixelImage does not have to calculate the bounding box at the beginning, since the object center is available without calculations, but has to update the bounding box after transformation. Therefore both PixelImage and PixelImageEnhanced need about the same time for rotation and scaling around the object center. For intersection and containment test PixelImage is notably faster since the bounding boxes of both shapes do not have to be calculated at the beginning.

The phenomena that the rotation of Areas sometimes suddenly takes much longer than the average time for all other shapes, especially shapes of the same type occurs only for combined shapes. Therefore, it possibly has something to do with the bug that has already been mentioned in Chapter 4.

For several reasons testing for intersection or containment of two objects does not always reach the same result for all representations. The first reason is the non-accurate bounding box calculation for Areas and Shape2D objects discussed in Chapter 4. The second reason is discretization, as done when initializing PixelImage and PixelImageEnhanced and the third reason is that some methods involve approximation. However, all cases where results contradict each other, are cases where a slightly different position of one shape already leads to a different result. Figure 5.18 shows four such cases for intersection testing, whereas Figure 5.19 displays four cases for containment testing.

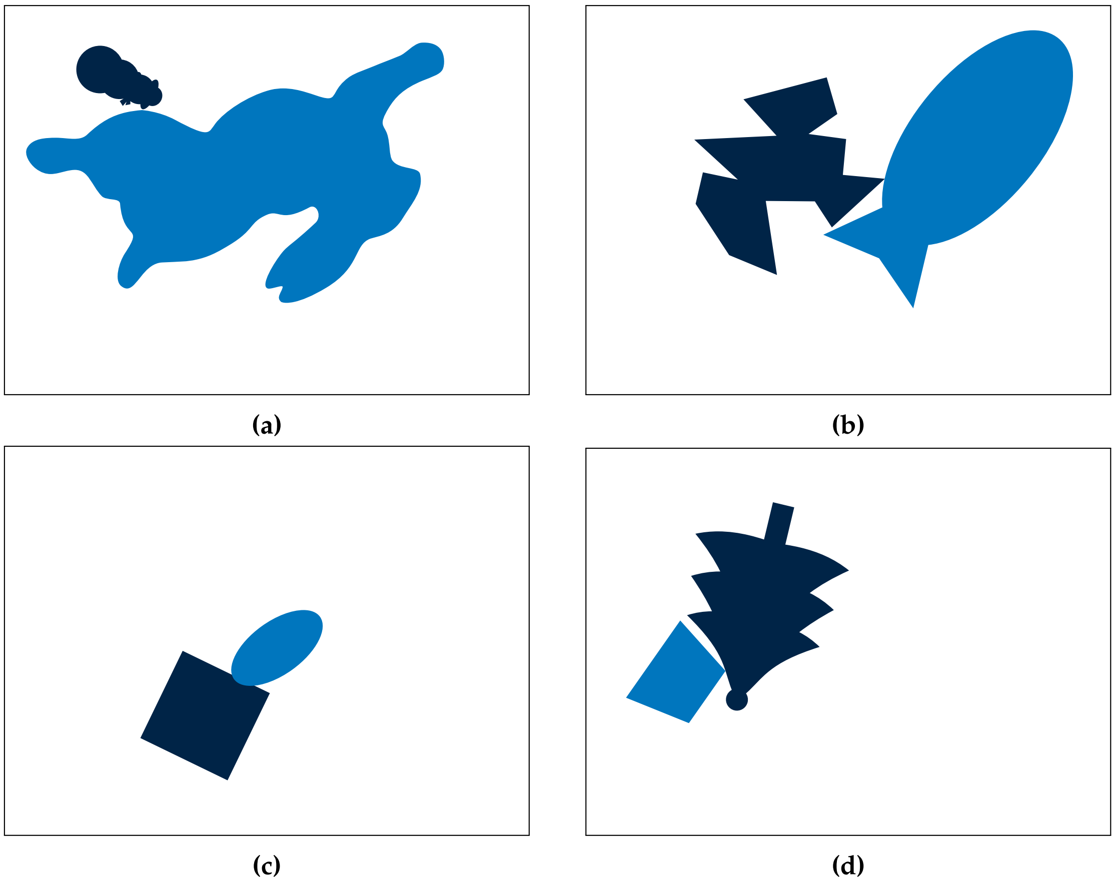
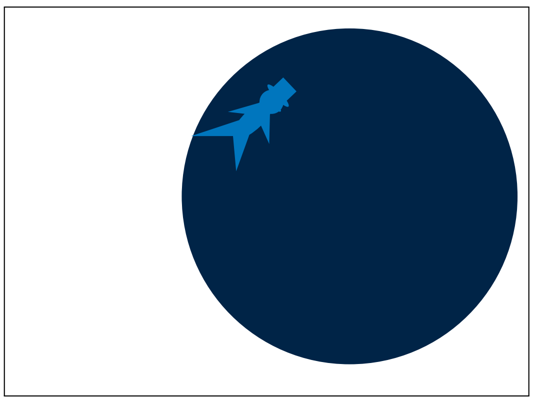
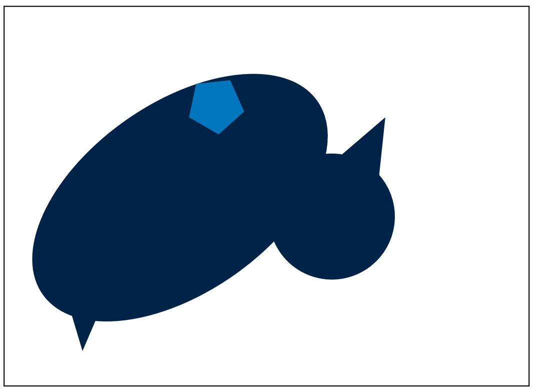


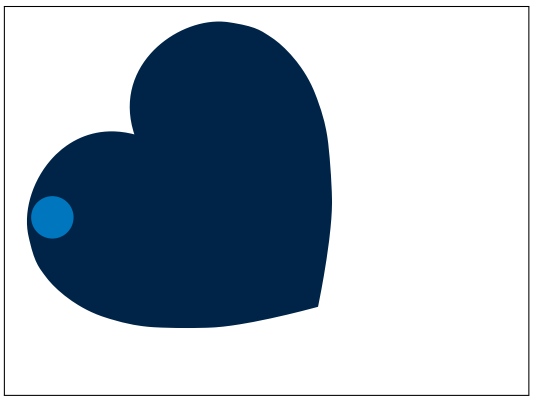
Figure 5.18.: Examples for intersection tests that yield different results



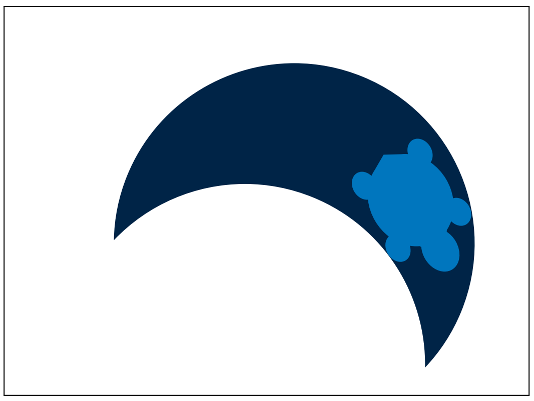
(a)



(b)



(c)



(d)

Figure 5.19.: Examples for containment tests that yield different results

6. Outlook

This chapter contains a collection of ideas for improving and extending the program.

The results of Chapter 5 suggest that Shape2D is one of the more promising representations examined in this thesis. Additionally, if available, the combined version of a shape should be preferred as its intersection test is more effective and transformation methods need only a negligible amount of time longer. However, the current implementation of combined shapes has two major disadvantages. First, testing if a combined shape contains an arbitrary other shape is done by bounding box intersection testing which is not very accurate. Therefore, choosing only the combined shape instead of the path representation is infeasible. In order to profit from the advantages of combined shapes and still provide accurate containment tests, both path and combined version of a shape would have to be kept updated. On the other hand that would lead to longer execution times for transformations. This basically only leaves the possibility of always using the path version of a shape, even if it could be combined by basic shapes. The second disadvantage of combined shapes is, that they can only represent the union of basic shapes. Other ways for combining shapes like the intersection or difference of shapes are not provided. That way, shapes that can normally be created by using constructive area geometry like the moon in Figure 5.11, can not be represented by combined shapes. Determining if an implementation of additional methods for combining shapes within the current program structure is possible without using very complex data structures and calculations methods ought to be the subject of further investigations.

Some frequently used cookie–cutter shapes contain holes. As none of the test shapes contained holes, there do not exists any test results on whether holes are treated correctly within intersection and containment testing. For PixelImage, Area and Geometry everything should work exactly the same. However, at least the polygon intersection presumes that polygons are always made up of one sequence of line segments. Therefore another topic for possible further research is to find out how the possibility of holes can be incorporated into the current Shape2D implementation.

A possible search algorithm solving the cookie–cutter problem currently has only transformation, intersection and containment methods available to find a goal. Other additional methods could possibly help to reach the goal faster. One idea is to modify intersection testing in a way that results are not either true or false, but in case of a positive intersection test, the method proposes a direction in which the current shape should be moved in order to resolve the intersection. Another idea is to provide a method that given multiple shapes issues a warning if a shape is transformed to a position very close to a different shape. Optimally this method would also including the direction where this other shape is positioned, so that the moving shape can avoid moving in the that direction. The multiple shapes given to that method would be the shapes that have already been positioned in case of the search algorithm.

Once a suitable solving algorithm for the cookie–cutter problem is found, a possible application for that algorithm could be letting a robot bake cookies or using the algorithm to recommend cookie–cutter position based on photos, possibly taken by a smart phone, of both dough and desired cookie cutter. Both applications use visual input by a camera, leading to shapes that are at least at the beginning represented by pixel images. An interesting question to answer is which representation should be used in such a case. in order to extract the shapes, the images have to be processed for both representations. Using a vectorbased representation would additionally require a vectorization of the shape found in an image. This vectorization could be so costly that it is more convenient to use a pixelbased representation.

7. Conclusion

The objective of this thesis was to find different representations for general geometric shapes where each representation provides methods to transform a shape and perform intersection and containment tests for two shapes. Representations of geometric shapes are used to solve C&P problems, including the cookie–cutter problem. Besides many other approaches, C&P problems can be solved by search algorithms, which require exactly those functionalities. In general, geometric shapes can either be saved using a vector graphic or a pixel graphics file format. Intuitively, calculating transformations and relative positioning of shapes should correlate with the way they are saved. Therefore, the search for representations focused on approaches based on vector or pixel graphics.

An elaborate research lead to five different representations already providing parts of the demanded procedures. Two of the representations are based on pixel graphics and three are based on vector graphics. In order to obtain only representations that completely fulfill the requirements, several functionalities had to be added. For pixel based representations the added functionalities are bounding box calculation, as well as intersection and containment tests. For vector based representations completing demanded methods lead to implementing the intersection of two polygons, the calculation of the closest point on one shape to a given other point and combining newly added and already available methods.

Detailed testing of all representations revealed that in general, vectorbased representations are considerably more effective than pixelbased approaches. Even though the representation Area yields similar results as the other two vectorbased approaches, a few isolated cases occurred where both runtime and memory consumption were much higher than in the average case. This phenomena could not fully be explained. In conclusion, either one of the representations Shape2D or Geometry is recommended. Future applications of the results of this thesis could include automated cookie–baking by robots or the implementation of an augmented reality based recommender system for cookie–cutter positions within private households.

A. Complexity Measure for Test Shapes

Measuring the complexity of a shape is a useful tool in shape recognition. Therefore, various different approaches can be found in literature. Chen and ′s complexity measure in [3] is based on distance and angle entropy, perceptual smoothness and randomness of a shape. The measure in [25] on the other hand incorporates the complexity of the shape's boundary, the difference between the shape and its convex hull and the symmetry of a shape. For simplicity reasons and due to the nature of the program, the complexity measure suggested here is built on analyzing the number of segments a shape can be constructed from by examining the shape's path definition and the number and kind of shapes by looking at the shape's composition of the four basic shapes circle, rectangle, ellipse and polygon.

Test Shape	# Line Segments	# Bézier Curves	Total	Rank
Triangle	3	0	3	1
Square	4	0	4	2
Rectangle	4	0	4	2
Trapezoid	4	0	4	2
Circle	0	4	4	3
Ellipse	0	4	4	3
Pentagon	5	0	5	4
Moon	0	6	6	5
Fish	4	4	8	6
Star	10	0	10	7
Heart	0	13	13	8
Bird	4	11	15	9
Polygon	17	0	17	10
Tree	16	3	19	11
Snowman	6	20	26	12
Man	11	17	28	13
Jigsaw	0	28	28	14
Turtle	3	27	30	15
Bunny	0	38	38	16
Snowflake	98	0	98	17

Table A.1.: Test shapes ordered by number of segments

The first measure ranks all test shapes according their total number of segments. Segments can either be lines or bézier curves. A circle, for example, if defined by a path of 4 bézier curves, while a rectangle if constructed from 4 line segments. Since line segments are less complex than bézier curves, two shapes with an equal number of total segments are ranked by classifying the shape with the lower number of bézier curves as less complex. Table A.1 shows the resulting order for all test shapes introduced in Chapter 5.

Test Shape	**#●**	**#■**	**#⬬**	**#⬟**	**#☁**	**Total**	**Rank**
Circle	1	0	0	0	0	1	1
Rectangle	0	1	0	0	0	1	2
Square	0	1	0	0	0	1	2
Ellipse	0	0	1	0	0	1	3
Polygon	0	0	0	1	0	1	4
Pentagon	0	0	0	1	0	1	4
Snowflake	0	0	0	1	0	1	4
Star	0	0	0	1	0	1	4
Trapezoid	0	0	0	1	0	1	4
Triangle	0	0	0	1	0	1	4
Fish	0	0	1	1	0	2	5
Bird	1	0	1	2	0	3	6
Man	2	1	2	2	0	7	7
Snowman	3	1	1	3	0	8	8
Turtle	0	0	7	1	0	8	9
Bunny	0	0	0	0	1	1	10
Heart	0	0	0	0	1	1	10
Jigsaw	0	0	0	0	1	1	10
Moon	0	0	0	0	1	1	10
Tree	0	0	0	0	1	1	10

Table A.2.: Test shapes ordered by shapes they are composed of

The second measure first ranks all test shapes by the total number of basic shapes they are composed of, then by the types of those basic shapes. Shapes that can not be defined by a composition of basic shapes, but can only be described by a generic path, are ranked last. Besides introducing a new complexity measure, the authors of [3] conducted a user study trying to find out if their measure has any correlation to the human perception of complexity. Since the user study suggests that it does, the results from their paper are used in the following to order basic shapes by complexity. The paper states, that circles are less complex than rectangles which themselves are simpler than polygons. Ellipses were not part of the test shapes shown in evaluation, but since the distance and angle entropy of ellipses should be higher then the one of rectangles, and in most cases lower then the one of polygons, ellipses are ranked between rectangles and polygons. Table A.2 contains the resulting ranks.

The final complexity rank of a shape is calculated by simply adding both previously mentioned ranks and ordering them starting with the lowest rank first. This leads to the following order with the rank given in parentheses after the respective shape name:

1. Circle (4)
2. Square (4)
3. Rectangle (4)
4. Triangle (5)
5. Ellipse (6)
6. Trapezoid (6)
7. Regular Pentagon (8)
8. Star (11)
9. Fish (11)
10. Concave Polygon (14)
11. Bird (15)
12. Moon (15)
13. Heart (18)
14. Man (20)
15. Snowman (20)
16. Snowflake (21)
17. Tree (21)
18. Turtle (24)
19. Jigsaw Piece (24)
20. Bunny (26)

B. Data of Runtime Tests

The following tables show the results of runtime tests with all times measured in ms.

Object Size	Method	Min	Max	Average
Small	Bounds1	1.361	4.050	1.653
	Bounds2	1.090	3.760	1.478
	Bounds3	1.492	8.764	2.947
Medium	Bounds1	1.116	3.835	1.583
	Bounds2	0.889	3.107	1.208
	Bounds3	1.517	15.550	5.929
Big	Bounds1	0.905	3.501	1.342
	Bounds2	0.880	2.983	1.077
	Bounds3	4.491	22.532	9.231

Table B.1.: Runtime of bounding box calculations

Object Size	Method	Min	Max	Average
Small	Bounds1	214727	305647	284492.7
	Bounds2	104927	284489	220486.4
	Bounds3	14380	480067	209642.0
Medium	Bounds1	85749	264126	214531.6
	Bounds2	16374	216735	132990.7
	Bounds3	36654	937911	462897.1
Big	Bounds1	16208	202585	141290.5
	Bounds2	10962	178571	89604.0
	Bounds3	331970	1484979	773725.1

Table B.2.: Number of accesses to the image data for bounding box calculations

Test Shape	Area	Geometry	Pixel	PixelE.	Shape2D	Shape C.
Bird	0.193	0.053	2.958	4.218	0.009	0.014
Bunny	0.404	0.033	2.940	4.190	0.009	–
Circle	0.033	0.057	2.952	4.228	0.008	–
Polygon	0.046	0.015	2.945	4.201	0.009	–
Ellipse	0.038	0.046	2.945	4.251	0.009	–
Fish	0.051	0.042	2.942	4.242	0.009	0.009
Heart	0.064	0.018	2.942	4.256	0.009	–
Jigsaw	0.250	0.027	2.948	4.218	0.009	–
Man	0.222	0.040	2.945	4.163	0.009	0.015
Moon	0.065	0.016	2.942	4.226	0.009	–
Rectangle	0.022	0.012	2.944	4.166	0.010	–
Pentagon	0.023	0.012	2.943	4.225	0.008	–
Snowflake	0.277	0.034	2.954	4.208	0.009	–
Snowman	0.175	0.054	2.954	4.237	0.009	0.010
Square	0.022	0.012	2.951	4.253	0.009	–
Star	0.034	0.013	3.092	4.248	0.009	–
Trapezoid	0.022	0.012	2.964	4.207	0.008	–
Tree	0.132	0.018	2.964	4.246	0.009	–
Triangle	0.021	0.012	2.956	4.227	0.008	–
Turtle	0.283	0.053	2.957	4.257	0.009	0.010
Total	0.118	0.029	2.957	4.223	0.009	0.011

Table B.3.: Runtime of translating test shapes of small size

Test Shape	Area	Geometry	Pixel	PixelE.	Shape2D	Shape C.
Bird	1.212	0.068	17.059	16.755	0.013	0.031
Bunny	4.706	0.046	17.017	16.766	0.016	–
Circle	0.372	0.067	17.106	16.764	0.008	–
Polygon	0.085	0.024	16.932	16.607	0.017	–
Ellipse	0.376	0.056	17.176	16.883	0.025	–
Fish	289.950 (0.321)	0.052	16.960	16.648	0.011	0.025
Heart	0.504	0.027	16.624	16.274	0.012	–
Jigsaw	3.676	0.037	17.124	16.808	0.013	–
Man	1.685	0.050	16.992	16.677	0.013	0.036
Moon	0.477	0.025	16.559	16.218	0.011	–
Rectangle	0.039	0.021	16.657	16.378	0.019	–
Pentagon	0.040	0.021	16.474	16.200	0.016	–
Snowflake	0.406	0.045	17.042	16.719	0.016	–
Snowman	1.895	0.066	16.898	16.585	0.013	0.019
Square	0.038	0.021	16.797	16.513	0.019	–
Star	0.063	0.023	17.143	16.859	0.011	–
Trapezoid	0.042	0.021	16.933	16.637	0.011	–
Tree	0.792	0.027	16.658	16.344	0.012	–
Triangle	0.042	0.021	16.842	16.555	0.011	–
Turtle	2.712	0.064	16.438	16.156	0.013	0.017
Total	15.521 (0.984)	0.039	16.874	16.570	0.014	0.026

Table B.4.: Runtime of rotating test shapes of small size, for Area the runtimes without the 3 outliers are given in parenthesis

Test Shape	**Area**	**Geometry**	**Pixel**	**PixelE.**	**Shape2D**	**Shape C.**
Bird	0.227	0.067	13.499	13.614	0.014	0.026
Bunny	0.451	0.043	13.445	13.254	0.016	–
Circle	0.056	0.069	13.486	13.310	0.017	–
Polygon	0.066	0.024	13.439	13.286	0.017	–
Ellipse	0.062	0.059	13.495	13.310	0.019	–
Fish	0.079	0.052	13.464	14.034	0.011	0.016
Heart	0.094	0.027	13.461	14.034	0.011	–
Jigsaw	0.293	0.037	13.378	13.977	0.012	–
Man	0.229	0.050	13.378	13.996	0.012	0.029
Moon	0.091	0.025	13.463	14.067	0.011	–
Rectangle	0.039	0.021	13.430	13.259	0.018	–
Pentagon	0.040	0.021	13.463	13.306	0.016	–
Snowflake	0.300	0.046	13.471	13.259	0.015	–
Snowman	0.213	0.066	13.498	14.063	0.012	0.018
Square	0.039	0.021	13.502	13.326	0.018	–
Star	0.054	0.023	13.491	13.323	0.011	–
Trapezoid	0.039	0.021	13.438	13.261	0.011	–
Tree	0.169	0.027	13.466	14.058	0.011	–
Triangle	0.038	0.020	13.444	13.279	0.011	–
Turtle	0.314	0.064	13.417	14.016	0.012	0.016
Total	0.144	0.039	13.457	13.600	0.014	0.021

Table B.5.: Runtime of scaling test shapes of small size

Test Shape	Area	Geometry	Pixel	PixelE.	Shape2D	Shape C.
Bird	0.319	0.061	17.203	17.372	0.011	0.014
Bunny	0.849	0.035	16.994	17.182	0.012	–
Circle	0.094	0.067	17.472	17.403	0.011	–
Polygon	0.074	0.023	17.399	17.305	0.011	–
Ellipse	0.093	0.055	17.514	17.425	0.015	–
Fish	0.111	0.050	17.158	17.348	0.010	0.012
Heart	0.138	0.025	17.004	17.212	0.011	–
Jigsaw	0.672	0.035	17.163	17.412	0.011	–
Man	0.425	0.048	17.091	17.207	0.011	0.021
Moon	0.149	0.023	17.082	17.295	0.010	–
Rectangle	0.036	0.019	17.287	17.190	0.015	–
Pentagon	0.037	0.019	17.294	17.214	0.010	–
Snowflake	0.362	0.043	17.354	17.300	0.012	–
Snowman	0.413	0.062	17.095	17.355	0.011	0.016
Square	0.036	0.019	17.316	17.237	0.015	–
Star	0.056	0.021	17.396	17.315	0.010	–
Trapezoid	0.036	0.019	17.203	17.104	0.010	–
Tree	0.294	0.025	17.008	17.129	0.011	–
Triangle	0.039	0.019	17.025	16.940	0.010	–
Turtle	0.580	0.061	16.999	17.137	0.012	0.014
Total	0.239	0.036	17.205	17.256	0.011	0.015

Table B.6.: Runtime of transforming test shapes of small size using a mixture of translation, rotation and scaling

Test Shape	Area	Geometry	Pixel	PixelE.	Shape2D	Shape C.
Bird	0.154	0.053	2.965	4.078	0.009	0.014
Bunny	0.428	0.035	2.955	4.025	0.009	–
Circle	0.033	0.056	2.955	4.074	0.010	–
Polygon	0.045	0.015	2.950	4.034	0.009	–
Ellipse	0.034	0.045	2.956	4.123	0.010	–
Fish	0.052	0.041	2.953	4.085	0.008	0.009
Heart	0.069	0.017	2.961	4.084	0.009	–
Jigsaw	0.259	0.027	2.957	4.052	0.009	–
Man	0.208	0.039	2.974	4.078	0.009	0.015
Moon	0.064	0.015	2.952	4.033	0.009	–
Rectangle	0.021	0.012	2.955	4.017	0.011	–
Pentagon	0.022	0.012	2.954	4.090	0.008	–
Snowflake	0.281	0.033	2.956	4.077	0.009	–
Snowman	0.223	0.054	3.094	4.063	0.009	0.010
Square	0.021	0.012	2.949	4.018	0.009	–
Star	0.034	0.013	2.949	4.080	0.008	–
Trapezoid	0.021	0.012	2.953	4.025	0.008	–
Tree	0.136	0.017	2.963	4.122	0.009	–
Triangle	0.020	0.012	2.951	4.102	0.008	–
Turtle	0.277	0.052	2.969	4.065	0.009	0.010
Total	0.120	0.028	2.964	4.066	0.009	0.011

Table B.7.: Runtime of translating test shapes of medium size

Test Shape	Area	Geometry	Pixel	PixelE.	Shape2D	Shape C.
Bird	336.996(1.232)	0.072	17.386	17.069	0.014	0.029
Bunny	4.281	0.046	17.323	17.002	0.016	–
Circle	0.370	0.077	16.988	16.738	0.008	–
Polygon	0.084	0.028	17.110	16.838	0.015	–
Ellipse	0.367	0.056	16.829	16.577	0.025	–
Fish	0.312	0.051	17.117	16.820	0.017	0.020
Heart	0.508	0.027	16.867	16.567	0.015	–
Jigsaw	3.608	0.037	17.184	16.924	0.013	–
Man	150.620 (1.607)	0.050	17.194	16.747	0.013	0.034
Moon	0.479	0.025	17.053	16.752	0.011	–
Rectangle	0.038	0.021	16.998	16.693	0.019	–
Pentagon	0.039	0.021	17.111	16.789	0.016	–
Snowflake	0.405	0.044	16.933	16.655	0.018	–
Snowman	1.725	0.065	16.974	16.668	0.012	0.023
Square	0.038	0.021	16.962	16.659	0.019	–
Star	0.063	0.023	17.092	16.818	0.011	–
Trapezoid	0.041	0.021	17.298	17.009	0.011	–
Tree	0.794	0.027	16.981	16.685	0.012	–
Triangle	0.043	0.021	16.908	16.584	0.011	–
Turtle	2.474	0.064	16.948	16.652	0.013	0.016
Total	25.278 (0.913)	0.040	17.066	16.765	0.014	0.025

Table B.8.: Runtime of rotating test shapes of medium size, for Area the runtimes without the 4 outliers are given in parenthesis

Test Shape	Area	Geometry	Pixel	PixelE.	Shape2D	Shape C.
Bird	0.189	0.068	13.267	13.193	0.013	0.025
Bunny	0.475	0.044	13.267	13.066	0.016	–
Circle	0.057	0.068	13.283	13.128	0.018	–
Polygon	0.065	0.025	13.234	13.082	0.016	–
Ellipse	0.057	0.059	13.335	13.141	0.019	–
Fish	0.081	0.052	13.273	13.860	0.011	0.014
Heart	0.099	0.027	13.279	13.859	0.011	–
Jigsaw	0.303	0.037	13.270	13.845	0.012	–
Man	0.246	0.050	13.250	13.827	0.012	0.029
Moon	0.089	0.025	13.230	13.828	0.011	–
Rectangle	0.039	0.021	13.227	13.051	0.018	–
Pentagon	0.040	0.021	13.308	13.134	0.016	–
Snowflake	0.301	0.046	13.299	13.107	0.015	–
Snowman	0.233	0.065	13.255	13.839	0.012	0.018
Square	0.039	0.021	13.249	13.062	0.018	–
Star	0.054	0.023	13.319	13.139	0.011	–
Trapezoid	0.039	0.021	13.261	13.080	0.010	–
Tree	0.173	0.027	13.309	13.914	0.011	–
Triangle	0.038	0.021	13.313	13.130	0.010	–
Turtle	0.320	0.064	13.280	13.853	0.012	0.016
Total	0.146	0.039	13.275	13.404	0.014	0.021

Table B.9.: Runtime of scaling test shapes of medium size

Test Shape	Area	Geometry	Pixel	PixelE.	Shape2D	Shape C.
Bird	0.301	0.061	17.145	17.347	0.011	0.014
Bunny	0.874	0.035	17.038	17.228	0.012	–
Circle	0.094	0.066	17.302	17.214	0.011	–
Polygon	0.076	0.023	17.402	17.285	0.010	–
Ellipse	0.094	0.054	17.279	17.202	0.014	–
Fish	0.108	0.049	17.097	17.309	0.010	0.011
Heart	0.139	0.025	17.066	17.267	0.011	–
Jigsaw	0.685	0.034	17.161	17.368	0.011	–
Man	0.403	0.047	17.079	17.296	0.011	0.020
Moon	0.160	0.023	17.058	17.255	0.010	–
Rectangle	0.036	0.019	17.237	17.140	0.015	–
Pentagon	0.037	0.019	17.331	17.271	0.010	–
Snowflake	0.359	0.042	17.306	17.231	0.012	–
Snowman	0.395	0.062	17.124	17.314	0.011	0.015
Square	0.036	0.019	17.191	17.098	0.015	–
Star	0.056	0.021	17.255	17.182	0.010	–
Trapezoid	0.037	0.019	17.307	17.214	0.010	–
Tree	0.278	0.025	16.975	17.197	0.011	–
Triangle	0.038	0.019	17.057	16.987	0.010	–
Turtle	0.607	0.061	17.120	17.311	0.012	0.014
Total	0.238	0.036	17.177	17.236	0.011	0.015

Table B.10.: Runtime of transforming test shapes of medium size using a mixture of translation, rotation and scaling

Test Shape	Area	Geometry	Pixel	PixelE.	Shape2D	Shape C.
Bird	0.168	0.053	2.960	3.983	0.009	0.015
Bunny	0.451	0.034	2.959	3.939	0.009	–
Circle	0.033	0.057	2.963	3.952	0.011	–
Polygon	0.047	0.015	2.966	3.970	0.008	–
Ellipse	0.039	0.046	2.974	4.016	0.009	–
Fish	0.053	0.041	2.958	4.025	0.009	0.008
Heart	0.062	0.018	2.946	3.972	0.009	–
Jigsaw	0.254	0.027	2.950	3.924	0.009	–
Man	0.347	0.039	2.957	4.187	0.009	0.015
Moon	0.069	0.015	2.967	3.955	0.009	–
Rectangle	0.021	0.012	2.957	3.986	0.011	–
Pentagon	0.022	0.012	2.966	3.965	0.008	–
Snowflake	0.279	0.034	2.956	3.954	0.009	–
Snowman	0.256	0.053	2.956	3.977	0.009	0.009
Square	0.021	0.012	2.944	3.886	0.009	–
Star	0.035	0.013	3.070	4.008	0.009	–
Trapezoid	0.021	0.012	2.946	4.013	0.008	–
Tree	0.120	0.017	2.950	3.966	0.009	–
Triangle	0.020	0.011	2.952	3.997	0.008	–
Turtle	0.371	0.052	3.064	3.966	0.009	0.009
Total	0.132	0.029	2.968	3.982	0.009	0.012

Table B.11.: Runtime of translating test shapes of big size

Test Shape	Area	Geometry	Pixel	PixelE.	Shape2D	Shape C.
Bird	312.362 (1.422)	0.075	17.679	17.415	0.014	0.029
Bunny	4.412	0.049	17.393	17.081	0.016	–
Circle	0.373	0.079	17.358	17.061	0.008	–
Polygon	0.084	0.026	17.413	17.069	0.015	–
Ellipse	0.371	0.056	17.409	17.098	0.025	–
Fish	0.311	0.051	17.400	17.081	0.017	0.022
Heart	0.504	0.027	17.062	16.730	0.017	–
Jigsaw	3.626	0.037	17.354	17.026	0.013	–
Man	436.962 (1.606)	0.050	17.361	17.165	0.012	0.036
Moon	0.512	0.024	17.013	16.726	0.011	–
Rectangle	0.039	0.021	17.279	16.964	0.019	–
Pentagon	0.040	0.021	17.217	16.911	0.016	–
Snowflake	0.411	0.044	17.159	16.907	0.017	–
Snowman	1.555	0.065	17.173	16.927	0.012	0.025
Square	0.039	0.021	17.253	16.929	0.019	–
Star	0.064	0.023	17.275	16.967	0.011	–
Trapezoid	0.041	0.021	17.380	17.077	0.010	–
Tree	0.825	0.027	17.440	17.142	0.012	–
Triangle	0.042	0.021	17.198	16.930	0.010	–
Turtle	2.482	0.064	17.340	17.041	0.013	0.017
Total	38.433 (0.923)	0.040	17.307	17.011	0.014	0.026

Table B.12.: Runtime of rotating test shapes of big size, for Area the runtimes without the 2 outliers are given in parenthesis

Test Shape	Area	Geometry	Pixel	PixelE.	Shape2D	Shape C.
Bird	0.201	0.068	13.245	13.287	0.013	0.025
Bunny	0.498	0.044	13.228	13.057	0.016	–
Circle	0.057	0.068	13.218	13.051	0.019	–
Polygon	0.066	0.025	13.223	13.022	0.017	–
Ellipse	0.063	0.058	13.250	13.041	0.019	–
Fish	0.081	0.052	13.227	13.797	0.011	0.014
Heart	0.092	0.027	13.191	13.753	0.011	–
Jigsaw	0.299	0.036	13.189	13.763	0.012	–
Man	0.365	0.050	13.298	13.889	0.012	0.030
Moon	0.095	0.024	13.152	13.716	0.011	–
Rectangle	0.039	0.021	13.208	13.009	0.019	–
Pentagon	0.040	0.021	13.190	12.985	0.016	–
Snowflake	0.302	0.045	13.199	12.984	0.014	–
Snowman	0.278	0.065	13.194	13.767	0.012	0.018
Square	0.039	0.021	13.150	12.934	0.017	–
Star	0.054	0.023	13.249	13.046	0.011	–
Trapezoid	0.039	0.021	13.180	12.982	0.011	–
Tree	0.155	0.027	13.209	13.773	0.012	–
Triangle	0.038	0.021	13.220	13.026	0.011	–
Turtle	0.386	0.064	13.216	13.787	0.012	0.016
Total	0.158	0.039	13.212	13.331	0.014	0.021

Table B.13.: Runtime of scaling test shapes of big size

Test Shape	Area	Geometry	Pixel	PixelE.	Shape2D	Shape C.
Bird	0.296	0.060	17.082	17.271	0.011	0.014
Bunny	0.859	0.035	17.066	17.217	0.012	–
Circle	0.093	0.066	17.284	17.167	0.011	–
Polygon	0.074	0.023	17.286	17.186	0.010	–
Ellipse	0.095	0.054	17.291	17.183	0.014	–
Fish	0.108	0.049	17.112	17.298	0.010	0.011
Heart	0.147	0.025	16.973	17.155	0.011	–
Jigsaw	0.688	0.034	17.078	17.241	0.011	–
Man	0.495	0.047	17.124	17.335	0.011	0.020
Moon	0.159	0.022	16.999	17.178	0.010	–
Rectangle	0.036	0.019	17.364	17.128	0.015	–
Pentagon	0.037	0.019	17.346	17.062	0.010	–
Snowflake	0.363	0.042	17.323	17.100	0.012	–
Snowman	0.462	0.062	17.118	17.240	0.011	0.015
Square	0.036	0.019	17.267	17.096	0.015	–
Star	0.058	0.021	17.344	17.193	0.010	–
Trapezoid	0.037	0.019	17.312	17.140	0.010	–
Tree	0.279	0.025	17.121	17.248	0.011	–
Triangle	0.038	0.019	17.259	17.130	0.010	–
Turtle	0.610	0.061	17.119	17.254	0.012	0.014
Total	0.247	0.036	17.194	17.191	0.011	0.015

Table B.14.: Runtime of transforming test shapes of big size using a mixture of translation, rotation and scaling

ID	Area		Geometry		Pixel		PixelE.		Shape		Shape2D	
	no	yes	no	yes	no	yes	no	yes	no	yes	no	yes
Bird	0.180	0.485	0.028	0.209	3.003	4.414	0.281	1.825	0.010	0.011	0.064	0.144
Bunny	0.269	0.810	0.027	0.210	2.976	4.384	0.241	1.827	0.010	0.011	0.108	0.265
Circle	0.070	0.300	0.021	0.200	2.980	4.432	0.232	1.820	0.009	0.010	0.015	0.046
Polygon	0.076	0.229	0.022	0.179	2.946	4.416	0.221	1.823	0.009	0.011	0.029	0.094
Ellipse	0.077	0.325	0.023	0.190	3.015	4.468	0.243	1.820	0.009	0.011	0.017	0.056
Fish	0.081	0.295	0.020	0.200	2.959	4.453	0.199	1.825	0.009	0.011	0.029	0.108
Heart	0.077	0.339	0.015	0.166	2.926	4.449	0.144	1.822	0.009	0.011	0.029	0.127
Jigsaw	0.122	0.693	0.027	0.212	3.001	4.405	0.242	1.826	0.010	0.012	0.087	0.227
Man	0.178	0.470	0.050	0.225	3.167	4.389	0.494	1.830	0.011	0.011	0.075	0.132
Moon	0.110	0.397	0.025	0.161	3.011	4.435	0.266	1.823	0.009	0.011	0.054	0.127
Rectangle	0.070	0.172	0.027	0.156	3.003	4.369	0.314	1.821	0.009	0.011	0.026	0.075
Pentagon	0.067	0.163	0.016	0.151	2.916	4.436	0.172	1.819	0.009	0.011	0.018	0.076
Snowflake	0.174	0.530	0.043	0.328	3.000	4.422	0.272	1.825	0.010	0.011	0.110	0.217
Snowman	0.125	0.502	0.025	0.236	2.978	4.431	0.228	1.825	0.010	0.011	0.067	0.176
Square	0.065	0.175	0.017	0.152	2.924	4.416	0.171	1.810	0.009	0.011	0.017	0.080
Star	0.076	0.173	0.027	0.163	3.047	4.421	0.300	1.810	0.009	0.011	0.029	0.084
Trapezoid	0.066	0.185	0.025	0.155	2.981	4.358	0.288	1.812	0.009	0.011	0.024	0.080
Tree	0.132	0.430	0.020	0.184	2.913	4.399	0.173	1.811	0.010	0.011	0.035	0.140
Triangle	0.065	0.188	0.019	0.156	2.920	4.378	0.199	1.811	0.009	0.011	0.018	0.079
Turtle	0.165	0.659	0.021	0.251	2.911	4.409	0.157	1.814	0.011	0.011	0.046	0.195
Total	0.112	0.377	0.025	0.195	2.977	4.412	0.239	1.820	0.010	0.011	0.044	0.126

Table B.15.: Runtime of intersection tests for test shapes of small size

ID	Area		Geometry		Pixel		PixelE.		Shape		Shape2D	
	no	**yes**	**no**	**yes**	**no**	**yes**	**no**	**yes**	**no**	**yes**	**no**	**yes**
Bird	0.615	3.526	0.009	1.537	2.729	4.446	0.029	1.870	0.008	0.008	0.010	0.068
Bunny	0.898	11.121	0.015	0.170	2.741	4.429	0.048	1.877	0.009	0.009	0.012	0.069
Circle	0.443	1.777	0.007	0.933	2.733	4.479	0.015	1.837	0.008	0.009	0.009	0.029
Polygon	0.470	0.094	0.010	0.297	2.736	4.528	0.043	1.903	0.008	0.008	0.011	0.027
Ellipse	0.476	1.478	0.007	1.074	2.764	4.541	0.015	1.859	0.008	0.008	0.010	0.065
Fish	0.343	1.309	0.008	1.309	2.765	4.530	0.026	1.861	0.008	0.008	0.010	0.063
Heart	0.364	2.416	0.007	0.334	2.775	4.553	0.018	1.849	0.009	0.009	0.010	0.047
Jigsaw	0.716	14.098	0.010	0.888	2.757	4.464	0.044	1.884	0.009	0.009	0.011	$\emptyset$ [a]
Man	0.609	4.570	0.011	1.498	2.716	4.432	0.056	1.939	0.009	0.009	0.011	$\emptyset$
Moon	0.420	1.864	0.007	0.275	2.752	4.534	0.027	1.881	0.009	0.008	0.010	0.051
Rectangle	0.367	1.129	0.009	0.152	2.716	4.379	0.055	1.862	0.008	0.009	0.011	0.050
Pentagon	0.309	0.490	0.006	0.139	2.737	4.487	0.019	1.843	0.008	0.008	0.010	0.034
Snowflake	0.814	0.676	0.010	1.489	2.735	4.515	0.033	1.899	0.009	0.009	0.012	$\emptyset$
Snowman	0.536	4.962	0.007	1.538	2.744	4.486	0.020	1.861	0.009	0.009	0.010	0.068
Square	0.339	0.374	0.006	0.136	2.744	4.527	0.014	1.845	0.008	0.009	0.010	0.040
Star	0.291	0.066	0.008	0.212	2.770	4.564	0.032	1.874	0.008	0.008	0.011	$\emptyset$
Trapezoid	0.482	0.993	0.009	0.158	2.724	4.452	0.051	1.854	0.008	0.008	0.011	0.051
Tree	0.435	4.087	0.007	0.401	2.749	4.515	0.021	1.868	0.009	0.009	0.010	0.044
Triangle	0.374	0.039	0.008	0.125	2.734	4.523	0.033	1.886	0.008	0.008	0.010	0.025
Turtle	0.515	9.053	0.006	2.005	2.759	4.504	0.021	1.860	0.009	0.009	0.011	0.067
Total	0.491	2.689	0.008	0.698	2.744	4.487	0.031	1.865	0.009	0.009	0.011	0.046

Table B.16.: Runtime of containment tests for test shapes of small size

[a] $\emptyset$ stands for: None of the containment tests was positive

ID	**Shape2D**		**Shape C.**	
	no	**yes**	**no**	**yes**
Bird	0.064	0.144	0.022	0.043
Bunny	0.189	0.279	0.042	0.138
Circle	0.017	0.058	0.013	0.028
Polygon	0.039	0.116	0.016	0.032
Ellipse	0.021	0.065	0.016	0.039
Fish	0.029	0.108	0.016	0.040
Heart	0.043	0.144	0.015	0.053
Jigsaw	0.117	0.246	0.035	0.111
Man	0.075	0.132	0.031	0.068
Moon	0.065	0.150	0.023	0.055
Rectangle	0.028	0.099	0.014	0.026
Pentagon	0.021	0.103	0.012	0.026
Snowflake	0.121	0.226	0.030	0.096
Snowman	0.067	0.176	0.023	0.042
Square	0.022	0.098	0.012	0.025
Star	0.041	0.106	0.015	0.031
Trapezoid	0.027	0.098	0.013	0.024
Tree	0.050	0.156	0.018	0.062
Triangle	0.024	0.097	0.013	0.025
Turtle	0.046	0.195	0.020	0.057
Total	0.055	0.144	0.021	0.051

Table B.17.: Runtime of intersection tests for test shapes of small size limited to combined shapes

ID	**Shape2D**		**Shape C.**	
	no	**yes**	**no**	**yes**
Bird	0.010	0.068	0.012	0.013
Fish	0.010	0.063	0.010	0.011
Man	0.011	$\emptyset$	0.012	0.014
Snowman	0.010	0.068	0.012	0.014
Turtle	0.011	0.067	0.012	0.014
Total	0.010	0.067	0.012	0.013

Table B.18.: Runtime of containment tests for test shapes of small size limited to combined shapes

ID	Area		Geometry		Pixel		PixelE.		Shape		Shape2D	
	no	**yes**	**no**	**yes**	**no**	**yes**	**no**	**yes**	**no**	**yes**	**no**	**yes**
0	0.160	0.524	0.094	0.198	3.333	4.046	1.024	1.788	0.011	0.011	0.336	0.173
1	0.344	0.870	0.077	0.206	3.018	4.016	0.748	1.794	0.012	0.011	0.426	0.307
2	0.088	0.354	0.049	0.184	2.934	4.050	0.575	1.780	0.009	0.011	0.029	0.042
3	0.095	0.283	0.073	0.183	3.117	4.019	0.792	1.791	0.010	0.011	0.081	0.103
4	0.089	0.357	0.057	0.177	2.978	4.099	0.651	1.780	0.010	0.011	0.037	0.056
5	0.107	0.385	0.067	0.182	3.032	4.069	0.740	1.794	0.010	0.011	0.173	0.137
6	0.107	0.366	0.059	0.162	3.050	4.052	0.692	1.787	0.010	0.011	0.177	0.150
7	0.165	0.823	0.082	0.208	3.142	4.034	0.804	1.792	0.010	0.011	0.319	0.269
8	0.173	0.529	0.081	0.221	3.103	4.030	0.818	1.800	0.012	0.011	0.148	0.160
9	0.185	0.493	0.106	0.163	3.533	4.027	1.278	1.802	0.011	0.011	0.460	0.163
10	0.086	0.213	0.087	0.159	3.362	4.008	1.083	1.795	0.010	0.011	0.077	0.091
11	0.087	0.192	0.073	0.155	3.236	4.072	0.897	1.787	0.010	0.011	0.062	0.090
12	0.211	0.654	0.129	0.329	3.219	4.054	0.856	1.791	0.012	0.011	0.239	0.205
13	0.191	0.587	0.112	0.216	3.459	4.027	1.153	1.795	0.011	0.011	0.289	0.201
14	0.082	0.201	0.044	0.155	2.830	4.008	0.512	1.788	0.010	0.011	0.043	0.091
15	0.087	0.231	0.111	0.167	3.672	4.080	1.323	1.801	0.010	0.011	0.099	0.097
16	0.091	0.223	0.107	0.161	3.562	4.026	1.284	1.798	0.010	0.011	0.089	0.090
17	0.173	0.490	0.102	0.184	3.429	4.088	1.066	1.796	0.011	0.012	0.307	0.165
18	0.081	0.197	0.072	0.157	3.259	4.087	0.899	1.805	0.010	0.011	0.060	0.089
19	0.179	0.675	0.094	0.220	3.249	4.044	0.923	1.800	0.012	0.011	0.183	0.222
Total	0.141	0.432	0.084	0.189	3.214	4.046	0.888	1.793	0.011	0.011	0.179	0.145

Table B.19.: Runtime of intersection tests for test shapes of medium size

ID	**Area**		**Geometry**		**Pixel**		**PixelE.**		**Shape**		**Shape2D**	
	no	**yes**	**no**	**yes**	**no**	**yes**	**no**	**yes**	**no**	**yes**	**no**	**yes**
0	1.602	4.251	0.009	2.043	2.372	4.294	0.023	2.032	0.008	0.008	0.011	∅
1	2.354	15.478	0.013	∅	2.325	4.233	0.016	2.106	0.009	0.010	0.012	∅
2	1.303	1.100	0.008	1.760	2.369	4.288	0.015	2.006	0.009	0.009	0.010	0.024
3	1.408	0.090	0.012	0.314	2.358	4.316	0.047	2.098	0.008	0.009	0.013	∅
4	1.236	1.041	0.008	1.681	2.410	4.366	0.012	2.004	0.009	0.008	0.010	∅
5	1.495	1.417	0.017	1.503	2.462	4.468	0.099	2.132	0.008	0.008	0.012	∅
6	1.261	3.161	0.008	0.391	2.376	4.314	0.017	2.021	0.009	0.009	0.010	0.025
7	2.442	15.291	0.011	0.871	2.367	4.280	0.037	2.086	0.009	0.009	0.011	∅
8	1.533	5.004	0.017	1.482	2.402	4.362	0.082	2.149	0.009	0.009	0.012	∅
9	1.409	1.772	0.010	0.288	2.349	4.322	0.036	2.110	0.009	0.009	0.011	∅
10	1.210	0.083	0.015	0.137	2.380	4.293	0.086	2.133	0.009	0.009	0.012	0.048
11	1.038	0.039	0.008	0.148	2.380	4.376	0.011	2.011	0.008	0.008	0.011	∅
12	2.396	0.675	0.011	1.475	2.374	4.359	0.025	2.061	0.009	0.010	0.013	∅
13	1.731	7.266	0.011	2.160	2.372	4.308	0.043	2.085	0.009	0.009	0.011	∅
14	1.232	0.511	0.014	0.139	2.391	4.211	0.091	2.007	0.009	0.009	0.012	0.043
15	1.025	0.064	0.009	0.212	2.391	4.399	0.027	2.080	0.008	0.009	0.012	∅
16	1.295	0.110	0.017	0.137	2.418	4.323	0.114	2.098	0.008	0.009	0.013	0.036
17	1.439	4.427	0.008	0.419	2.396	4.367	0.013	2.027	0.009	0.009	0.010	∅
18	0.890	0.034	0.008	0.125	2.390	4.364	0.020	2.037	0.008	0.008	0.011	∅
19	1.957	10.297	0.011	2.107	2.372	4.310	0.037	2.096	0.009	0.009	0.012	∅
Total	1.513	3.178	0.011	0.841	2.383	4.318	0.042	2.064	0.009	0.009	0.011	0.035

Table B.20.: Runtime of containment tests for test shapes of medium size

ID	**Shape2D**		**Shape C.**	
	no	**yes**	**no**	**yes**
Bird	0.336	0.173	0.065	0.040
Bunny	0.871	0.376	0.141	0.148
Circle	0.039	0.052	0.019	0.026
Polygon	0.111	0.133	0.030	0.032
Ellipse	0.053	0.066	0.036	0.041
Fish	0.173	0.137	0.045	0.042
Heart	0.339	0.183	0.044	0.059
Jigsaw	0.574	0.311	0.143	0.135
Man	0.148	0.160	0.067	0.076
Moon	0.788	0.204	0.121	0.072
Rectangle	0.107	0.120	0.027	0.024
Pentagon	0.080	0.120	0.022	0.025
Snowflake	0.233	0.235	0.062	0.086
Snowman	0.289	0.201	0.083	0.042
Square	0.072	0.119	0.019	0.023
Star	0.121	0.127	0.037	0.030
Trapezoid	0.125	0.120	0.036	0.023
Tree	0.462	0.212	0.059	0.068
Triangle	0.088	0.118	0.022	0.025
Turtle	0.183	0.222	0.065	0.062
Total	0.237	0.173	0.060	0.053

Table B.21.: Runtime of intersection tests for test shapes of medium size limited to combined shapes

ID	**Shape2D**		**Shape C.**	
	no	**yes**	**no**	**yes**
Bird	0.011	$\emptyset$	0.012	0.013
Fish	0.012	$\emptyset$	0.011	0.011
Man	0.012	$\emptyset$	0.013	0.014
Snowman	0.011	$\emptyset$	0.013	0.014
Turtle	0.012	$\emptyset$	0.013	0.014
Total	0.012	$\emptyset$	0.012	0.013

Table B.22.: Runtime of containment tests for test shapes of medium size limited to combined shapes

ID	**Area**		**Geometry**		**Pixel**		**PixelE.**		**Shape**		**Shape2D**	
	no	**yes**	**no**	**yes**	**no**	**yes**	**no**	**yes**	**no**	**yes**	**no**	**yes**
Bird	0.261	0.694	0.142	0.208	3.794	3.887	1.950	1.813	∅	0.011	0.943	0.180
Bunny	0.380	1.089	0.170	0.213	3.553	3.865	1.651	1.827	0.012	0.012	3.588	0.320
Circle	0.110	0.501	0.122	0.193	3.904	3.846	1.947	1.792	∅	0.010	0.076	0.025
Polygon	0.148	0.379	0.140	0.192	3.946	3.878	2.011	1.815	∅	0.011	0.140	0.109
Ellipse	∅	0.539	∅	0.189	∅	3.903	∅	1.807	∅	0.011	∅	0.037
Fish	∅	0.498	∅	0.191	∅	3.933	∅	1.816	∅	0.011	∅	0.144
Heart	0.161	0.450	0.119	0.168	3.746	3.880	1.721	1.806	0.010	0.011	0.426	0.171
Jigsaw	0.316	1.076	0.171	0.222	3.834	3.830	1.877	1.809	∅	0.011	0.430	0.295
Man	∅	0.808	∅	0.231	∅	3.995	∅	1.850	∅	0.011	∅	0.158
Moon	0.212	0.585	0.123	0.168	3.749	3.873	1.788	1.838	0.011	0.011	0.222	0.169
Rectangle	∅	0.248	∅	0.160	∅	3.897	∅	1.804	∅	0.011	∅	0.096
Pentagon	0.092	0.257	0.122	0.163	3.930	3.862	1.960	1.804	∅	0.011	0.120	0.099
Snowflake	0.392	0.896	0.274	0.348	3.841	3.868	1.905	1.813	∅	0.011	4.633	0.203
Snowman	∅	0.787	∅	0.227	∅	3.882	∅	1.816	∅	0.011	∅	0.209
Square	∅	0.254	∅	0.160	∅	3.804	∅	1.803	∅	0.011	∅	0.099
Star	0.110	0.298	0.127	0.175	3.975	3.943	1.975	1.822	∅	0.011	0.126	0.103
Trapezoid	∅	0.278	∅	0.167	∅	3.848	∅	1.809	∅	0.011	∅	0.097
Tree	∅	0.621	∅	0.195	∅	3.880	∅	1.815	∅	0.011	∅	0.171
Triangle	0.081	0.263	0.121	0.164	4.011	3.915	1.988	1.816	∅	0.011	0.120	0.096
Turtle	0.521	0.994	0.164	0.234	3.973	3.864	2.049	1.810	∅	0.011	1.769	0.255
Total	0.234	0.576	0.139	0.198	3.764	3.883	1.807	1.814	0.011	0.011	0.984	0.152

Table B.23.: Runtime of intersection tests for test shapes of big size

ID	Area		Geometry		Pixel		PixelE.		Shape		Shape2D	
	no	**yes**	**no**	**yes**	**no**	**yes**	**no**	**yes**	**no**	**yes**	**no**	**yes**
Bird	2.405	4.076	0.009	2.051	2.189	4.362	0.024	2.268	0.009	0.008	0.011	$\emptyset$
Bunny	3.355	16.678	0.021	$\emptyset$	2.219	4.315	0.092	2.331	0.010	0.010	0.013	$\emptyset$
Circle	2.132	1.061	0.008	2.033	2.141	4.263	0.013	2.223	0.009	0.009	0.010	0.014
Polygon	2.097	0.094	0.012	0.305	2.191	4.412	0.050	2.341	0.009	0.009	0.013	$\emptyset$
Ellipse	2.182	1.214	0.009	1.683	2.193	4.429	0.019	2.302	0.009	0.008	0.012	$\emptyset$
Fish	2.127	1.421	0.023	1.486	2.363	4.590	0.163	2.416	0.008	0.008	0.015	$\emptyset$
Heart	1.972	3.166	0.008	0.392	2.172	4.314	0.011	2.211	0.009	0.008	0.010	$\emptyset$
Jigsaw	3.735	15.244	0.016	0.882	2.183	4.305	0.085	2.354	0.009	0.009	0.012	0.029
Man	2.688	7.321	0.024	1.470	2.392	4.630	0.150	2.374	0.009	0.009	0.013	$\emptyset$
Moon	1.854	1.915	0.012	0.286	2.186	4.347	0.065	2.329	0.009	0.009	0.012	$\emptyset$
Rectangle	1.970	0.887	0.012	0.137	2.232	4.456	0.067	2.313	0.009	0.009	0.012	0.058
Pentagon	1.678	0.039	0.008	0.149	2.158	4.326	0.019	2.263	0.008	0.008	0.012	$\emptyset$
Snowflake	4.091	0.669	0.013	1.481	2.164	4.393	0.041	2.340	0.009	0.009	0.014	$\emptyset$
Snowman	2.865	7.969	0.012	2.145	2.203	4.408	0.051	2.340	0.009	0.009	0.012	$\emptyset$
Square	1.893	0.472	0.016	0.137	2.185	4.263	0.110	2.284	0.009	0.009	0.012	0.051
Star	1.387	0.065	0.008	0.212	2.224	4.434	0.019	2.250	0.008	0.008	0.012	$\emptyset$
Trapezoid	1.917	0.284	0.025	0.138	2.318	4.453	0.210	2.412	0.008	0.008	0.015	0.055
Tree	2.258	4.052	0.007	0.420	2.162	4.295	0.011	2.229	0.009	0.008	0.010	$\emptyset$
Triangle	1.357	0.033	0.007	0.124	2.195	4.383	0.012	2.234	0.008	0.008	0.012	$\emptyset$
Turtle	3.299	10.421	0.010	2.128	2.175	4.328	0.035	2.294	0.009	0.009	0.012	$\emptyset$
Total	2.363	3.618	0.013	0.907	2.212	4.383	0.062	2.305	0.009	0.009	0.012	0.030

Table B.24.: Runtime of containment tests for test shapes of big size

ID	**Shape2D**		**Shape C.**	
	no	**yes**	**no**	**yes**
Bird	0.943	0.180	0.146	0.036
Bunny	∅	0.366	∅	0.156
Circle	∅	0.025	∅	0.024
Polygon	∅	0.143	∅	0.029
Ellipse	∅	0.037	∅	0.035
Fish	∅	0.144	∅	0.036
Heart	0.213	0.212	0.166	0.064
Jigsaw	∅	0.327	∅	0.145
Man	∅	0.158	∅	0.071
Moon	0.264	0.204	0.170	0.078
Rectangle	∅	0.130	∅	0.020
Pentagon	∅	0.133	∅	0.022
Snowflake	∅	0.238	∅	0.080
Snowman	∅	0.209	∅	0.035
Square	∅	0.134	∅	0.021
Star	∅	0.137	∅	0.027
Trapezoid	∅	0.131	∅	0.021
Tree	∅	0.209	∅	0.067
Triangle	∅	0.129	∅	0.023
Turtle	1.769	0.255	0.214	0.072
Total	0.737	0.181	0.169	0.052

Table B.25.: Runtime of intersection tests for test shapes of big size limited to combined shapes

ID	**Shape2D**		**Shape C.**	
	no	**yes**	**no**	**yes**
Bird	0.010	∅	0.012	0.012
Fish	0.010	∅	0.011	0.011
Man	0.011	∅	0.013	0.015
Snowman	0.011	∅	0.013	0.013
Turtle	0.011	∅	0.013	0.013
Total	0.011	∅	0.012	0.012

Table B.26.: Runtime of containment tests for test shapes of big size limited to combined shapes

ID	Area		Geometry		Pixel		PixelE.		Shape		Shape2D	
	no	**yes**	**no**	**yes**	**no**	**yes**	**no**	**yes**	**no**	**yes**	**no**	**yes**
Bird	0.126	0.503	0.030	0.202	3.164	4.443	0.359	1.892	0.004	0.005	0.080	0.180
Bunny	0.351	0.943	0.060	0.201	3.408	4.384	0.685	1.899	0.005	0.005	0.344	0.330
Circle	0.072	0.372	0.028	0.179	3.124	4.266	0.355	1.873	0.003	0.005	0.013	0.035
Polygon	0.075	0.287	0.037	0.180	3.189	4.346	0.447	1.887	0.003	0.005	0.047	0.103
Ellipse	0.079	0.380	0.033	0.178	3.190	4.360	0.455	1.882	0.003	0.005	0.019	0.049
Fish	0.087	0.346	0.041	0.179	3.232	4.380	0.503	1.897	0.004	0.005	0.116	0.146
Heart	0.077	0.346	0.026	0.158	3.092	4.362	0.348	1.882	0.003	0.005	0.071	0.164
Jigsaw	0.126	0.786	0.046	0.207	3.261	4.327	0.512	1.910	0.004	0.005	0.229	0.309
Man	0.182	0.522	0.054	0.216	3.363	4.431	0.587	1.884	0.005	0.005	0.097	0.171
Moon	0.123	0.468	0.038	0.161	3.288	4.399	0.525	1.895	0.003	0.005	0.107	0.168
Rectangle	0.065	0.198	0.033	0.154	3.170	4.408	0.455	1.907	0.003	0.005	0.028	0.091
Pentagon	0.063	0.195	0.026	0.153	3.123	4.346	0.377	1.883	0.003	0.005	0.025	0.093
Snowflake	0.184	0.675	0.064	0.335	3.114	4.309	0.435	1.891	0.005	0.005	0.203	0.216
Snowman	0.130	0.552	0.041	0.219	3.191	4.354	0.450	1.875	0.004	0.005	0.111	0.207
Square	0.058	0.205	0.039	0.152	3.215	4.287	0.544	1.876	0.003	0.005	0.031	0.091
Star	0.068	0.237	0.044	0.165	3.313	4.387	0.594	1.900	0.003	0.005	0.047	0.096
Trapezoid	0.062	0.193	0.026	0.153	3.090	4.406	0.371	1.893	0.003	0.005	0.023	0.090
Tree	0.119	0.454	0.042	0.184	3.195	4.370	0.494	1.885	0.004	0.005	0.064	0.168
Triangle	0.064	0.206	0.037	0.153	3.256	4.367	0.512	1.881	0.003	0.005	0.030	0.088
Turtle	0.155	0.677	0.055	0.224	3.264	4.341	0.585	1.886	0.005	0.005	0.184	0.258
Total	0.110	0.436	0.039	0.189	3.208	4.361	0.473	1.889	0.004	0.005	0.086	0.156

Table B.27.: Runtime of intersection tests for test shapes of mixed sizes

ID	Area		Geometry		Pixel		PixelE.		Shape		Shape2D	
	no	**yes**	**no**	**yes**	**no**	**yes**	**no**	**yes**	**no**	**yes**	**no**	**yes**
Bird	1.003	2.405	0.007	0.950	2.770	4.647	0.059	1.979	0.003	0.003	0.006	0.051
Bunny	1.986	9.714	0.022	0.164	2.736	4.463	0.157	2.008	0.003	0.004	0.009	0.064
Circle	1.222	2.232	0.008	0.275	2.622	4.434	0.081	1.998	0.003	0.003	0.005	0.038
Polygon	1.090	0.294	0.016	0.267	2.729	4.435	0.159	2.003	0.002	0.003	0.008	0.035
Ellipse	0.930	1.501	0.006	0.491	2.655	4.454	0.052	1.937	0.002	0.002	0.005	0.046
Fish	0.904	1.446	0.012	0.713	2.683	4.530	0.118	2.001	0.002	0.002	0.006	0.046
Heart	1.022	1.219	0.007	0.178	2.674	4.472	0.069	1.969	0.003	0.003	0.005	0.042
Jigsaw	2.087	8.447	0.034	0.510	2.813	4.557	0.310	2.074	0.003	0.003	0.010	0.044
Man	1.058	6.458	0.009	1.487	2.740	4.517	0.071	1.968	0.003	0.003	0.006	$\emptyset$
Moon	0.953	1.569	0.011	0.205	2.736	4.447	0.119	1.918	0.003	0.003	0.006	0.045
Rectangle	0.644	0.609	0.007	0.145	2.653	4.453	0.060	1.951	0.002	0.003	0.005	0.034
Pentagon	0.766	1.396	0.006	0.139	2.638	4.437	0.055	1.939	0.002	0.002	0.006	0.040
Snowflake	2.102	0.701	0.036	1.571	2.721	4.625	0.208	2.183	0.003	0.003	0.010	0.024
Snowman	1.113	3.576	0.010	0.843	2.713	4.355	0.083	1.892	0.003	0.003	0.006	0.072
Square	1.014	1.790	0.015	0.136	2.710	4.318	0.168	1.960	0.002	0.003	0.007	0.043
Star	0.846	0.472	0.016	0.194	2.742	4.659	0.162	2.104	0.002	0.003	0.008	0.042
Trapezoid	0.705	1.277	0.007	0.138	2.729	4.381	0.073	1.911	0.002	0.002	0.006	0.040
Tree	1.135	2.157	0.010	0.237	2.698	4.373	0.087	1.906	0.003	0.003	0.005	0.053
Triangle	0.754	1.126	0.011	0.144	2.736	4.395	0.119	1.906	0.002	0.002	0.007	0.039
Turtle	1.679	3.978	0.016	0.608	2.691	4.324	0.142	1.927	0.003	0.003	0.008	0.089
Total	1.152	2.208	0.013	0.341	2.710	4.429	0.118	1.965	0.003	0.003	0.007	0.046

Table B.28.: Runtime of containment tests for test shapes of mixed sizes

ID	Shape2D		Shape C.	
	no	yes	no	yes
Bird	0.080	0.180	0.025	0.038
Bunny	0.385	0.382	0.124	0.169
Circle	0.018	0.048	0.009	0.019
Polygon	0.064	0.130	0.015	0.029
Ellipse	0.024	0.061	0.017	0.035
Fish	0.116	0.146	0.030	0.041
Heart	0.090	0.193	0.021	0.058
Jigsaw	0.264	0.349	0.083	0.160
Man	0.097	0.171	0.041	0.082
Moon	0.129	0.205	0.036	0.075
Rectangle	0.037	0.117	0.010	0.021
Pentagon	0.026	0.117	0.008	0.019
Snowflake	0.287	0.258	0.053	0.107
Snowman	0.111	0.207	0.036	0.035
Square	0.046	0.116	0.013	0.018
Star	0.056	0.125	0.014	0.026
Trapezoid	0.028	0.119	0.008	0.020
Tree	0.081	0.208	0.031	0.072
Triangle	0.042	0.115	0.010	0.019
Turtle	0.184	0.258	0.052	0.068
Total	0.104	0.186	0.032	0.056

Table B.29.: Runtime of intersection tests for test shapes of mixed sizes limited to combined shapes

ID	Shape2D		Shape C.	
	no	yes	no	yes
Bird	0.006	0.051	0.005	0.007
Fish	0.006	0.046	0.004	0.005
Man	0.006	$\emptyset$	0.006	0.009
Snowman	0.006	0.072	0.006	0.008
Turtle	0.008	0.089	0.006	0.008
Total	0.006	0.071	0.006	0.007

Table B.30.: Runtime of containment tests for test shapes of mixed sizes limited to combined shapes

List of Figures

List of Tables

Bibliography

[1] Budi Kurniawan Alexander Kolesnikov and and Paul Deck. *Java Drawing with Apache Batik: A Tutorial*. Brainysoftware, 2006.

[2] Alexandros Bouganis and Murray Shanahan. A Vision–Based Intelligent System for Packing 2–D Irregular Shapes. *IEEE Transactions on Automation Science and Engineering*, 4(3):382–394, 2007.

[3] Yinpeng Chen and Hari Sundaram. Estimating Complexity of 2D Shapes. In *2005 IEEE 7th Workshop on Multimedia Signal Processing*, pages 1–4, 2005.

[4] Erik Dahlström, Patrick Dengler, Anthony Grasso, Chris Lilley, Cameron McCormack, Doug Schepers, and Jonathan Watt. Scalable Vector Graphics (SVG) 1.1 (Second Edition). W3C Recommendation, 2011. Available online at : `http://www.w3.org/TR/2011/REC-SVG11-20110816/`; (last checked August 8, 2012).

[5] Kathryn A. Dowsland and William B. Dowsland. Packing Problems. *European Journal of Operational Research*, 56(1):2–14, 1992.

[6] Kathryn A. Dowsland and William B. Dowsland. Solution Approaches to Irregular Nesting Problems. *European Journal of Operational Research*, 84(3):506–521, 1995.

[7] Harald Dyckhoff. A Typology of Cutting and Packing Problems. *European Journal of Operational Research*, 44(2):145–154, 1990.

[8] Max J. Egenhofer and John R. Herring. Categorizing Binary Topological Relationships between Regions, Lines, and Points in Geographic Databases. Technical report, University of Maine, Dept. of Surveying Engineerin, Orono, ME, 1992.

[9] Christer Ericson. *Real-Time Collision Detection*. Morgan Kaufmann Series in Interactive 3D Technolog. Morgan Kaufman, San Francisco, CA, 1st edition, 2005.

[10] Robert J. Fowler, Mike Paterson, and Steven L. Tanimoto. Optimal Packing and Covering in the Plane are NP-Complete. *Information Processing Letters*, 12(3):133–137, 1981.

[11] Michael Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, NY, 1st edition, 1979.

[12] Ramin Halavati, Saeed B. Shouraki, Mahdieh Noroozian, and Saman H. Zadeh. Optimizing Allocation of Two Dimensional Irregular Shapes using an Agent Based Approach. In *The 4th World Enformatika Conference (WEC'05)*, pages 24–26, 2005.

[13] Paul Hoffmann. *The Man Who Loved Only Numbers: The Story of Paul Erdos and the Search for Mathematical Truth.* Forth Estate, London, 1st edition, 1999.

[14] Frank Klawonn. *Introduction to computer graphics: Using Java 2D and 3D.* Springer, London; New York, 2nd edition, 2012.

[15] Jonathan Knudsen. *Java 2D Graphics.* O'Reilly Media, Bejing, 1st edition, 1999.

[16] Jeffrey M. Lane and Richard F. Riesenfeld. A Theoretical Development for the Computer Generation and Display of Piecewise Polynomial Surfaces. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2(1):35–46, 1980.

[17] David M. Mount. *Handbook of Discrete and Computational Geometry*, chapter 38, pages 857–876. CRC Press LLC, Boca Raton, FL, 2nd edition, 2004.

[18] Open GIS Consortium, Inc. OpenGIS Simple Features Specification For SQL - Revision 1.1. OpenGIS project Document 99-049, 1999. Available online at : `http://www.opengeospatial.org/standards/sfs/`; (last checked August 8, 2012).

[19] Jürgen Richter-Gebert. *Perspectives on Projective Geometry: A Guided Tour Through Real and Complex Geometry.* Springer, Berlin; Heidelberg, 1st edition, 2011.

[20] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach.* Pearson Education, 3rd edition, 2009.

[21] Thomas W. Sederberg and Tomoyuki Nishita. Curve intersection using bézier clipping. *Computer-Aided Design*, 22(9):538–549, 1990.

[22] Michael I. Shamos and Dan Hoey. Geometric Intersection Problems. In *Proceedings of the 17th Annual Symposium on Foundations of Computer Science, Houston, Texas*, pages 208–215, 1976.

[23] M. Shimrat. Algorithm 112: Position of Point Relative to Polygon. *Communications of the ACM*, 5(8):434, 1962.

[24] Vivid Solutions. JTS Topology Suite Technical Specifications - Version 1.4, 2003. Available online at: `http://www.vividsolutions.com/JTS/bin/JTS%20Technical%20Specs.pdf`; (last checked August 8, 2012).

[25] Hongjiang Su, Ahmed Bouridane, and Danny Crookes. Scale Adaptive Complexity Measure of 2D Shapes. In *18th International Conference on Pattern Recognition, 2006 - ICPR (2)*, pages 134–137, 2006.

[26] Franco P. Preparata und Michael Ian Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, 1985.

[27] Gerhard Wäscher, Heike Haußner, and Holger Schumann. An Improved Typology of Cutting and Packing Problems. *European Journal of Operational Research*, 183(3):1109–1130, 2007.

[28] Paul F. Whelan and Bruce G. Batchelor. Automated packing systems – a systems engineering approach. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, 26(5):533–544, September 1996.

[29] Zhinong Zhong, Ning Jing, Luo Chen, and Qiu-Yun Wu. Representing topological relationships among heterogeneous geometry-collection features. *Journal of Computer Scence and Technology*, 19(3):280–289, 2004.