

# **The Camera Placement Problem – An art gallery problem variation**

Master Thesis  
Axis Communications AB  
Department of Computer Science, LTH

Mikael Pålsson  
Joachim Ståhl

February 2008

# Abstract

Choosing the correct location for surveillance cameras in a camera installation is a complex problem. The layout of the location, the resolution and angle of view of the cameras and the required image quality all have to be considered. By providing software that helps a camera installer to plan camera locations his job will be made easier and the quality of the installation will improve.

The goal of this master thesis is to find an algorithm that solves the variation of the art gallery problem we have chosen to refer to as the camera placement problem. Promising algorithms are implemented in prototype software and evaluated. The goal is to find an algorithm at least being able to handle:

- Limited field of view of cameras.
- Limited effective range of cameras.
- Obstacles limiting the line of sight.
- A coverage threshold.

First we investigate a very well-known algorithm, the 3-Color algorithm, and conclude that it does not fit our needs. We then move on to explaining how an algorithm of the authors own design, called the rectangular algorithm, works. We test two different strategies of camera placement within the rectangular algorithm, referred to as the basic and the greedy strategy. Since the 3-Color algorithm produced poor results the majority of the evaluation is focused on the advantages and shortcomings of the rectangular algorithm and comparison of these two camera placement strategies.

Finally we conclude that both the basic and the greedy version of the rectangular algorithm fulfill the basic requirements of the algorithm listed above. The greedy strategy generally produces better results than the basic one but do so at approximately twice the time needed by the basic strategy. The greedy strategy is also the version of the rectangular algorithm that is easiest to modify, bringing us to the conclusion that the greedy rectangular algorithm is the best of the evaluated algorithms for solving the camera placement problem.

## **Acknowledgements**

We would like to thank our LTH supervisor Christos Levcopoulos for his ideas and for his general support during this thesis. We would also like to thank our supervisor at Axis Communications AB, Hans Olsen, for all the help he has provided us with. Finally we would like to thank Axis Communications AB and all its employees for making this master thesis possible.

# Table of Contents

|       |                                                                 |    |
|-------|-----------------------------------------------------------------|----|
| 1     | Introduction.....                                               | 4  |
| 2     | Background.....                                                 | 5  |
| 2.1   | Motivation.....                                                 | 5  |
| 2.2   | Definition and earlier research.....                            | 5  |
| 2.3   | Camera placement limiters .....                                 | 7  |
| 2.4   | Problem description .....                                       | 8  |
| 3     | Algorithms .....                                                | 10 |
| 3.1   | General requirements of the algorithms.....                     | 10 |
| 3.2   | The 3-Color algorithm .....                                     | 10 |
| 3.2.1 | Polygon triangulation.....                                      | 11 |
| 3.2.2 | Vertex coloring .....                                           | 11 |
| 3.3   | Rectangular algorithm.....                                      | 12 |
| 3.3.1 | Finding rectangles in orthogonal polygons.....                  | 13 |
| 3.3.2 | Covering rectangles with cameras .....                          | 14 |
| 3.3.3 | Placing cameras .....                                           | 16 |
| 3.3.4 | Removing redundant cameras.....                                 | 17 |
| 4     | The Prototype Software .....                                    | 18 |
| 4.1   | Program survey .....                                            | 18 |
| 4.1.1 | Defining the room polygon.....                                  | 18 |
| 4.1.2 | Setting camera field of view and effective range.....           | 19 |
| 4.1.3 | Predefining cameras.....                                        | 20 |
| 4.1.4 | Choosing algorithm settings .....                               | 21 |
| 4.1.5 | The solution view.....                                          | 21 |
| 4.1.6 | Editing solutions .....                                         | 22 |
| 4.2   | Architecture.....                                               | 23 |
| 4.3   | Design choices and motivations.....                             | 24 |
| 5     | Evaluation .....                                                | 26 |
| 5.1   | Initial observations.....                                       | 26 |
| 5.2   | The quality of a solution and individual camera positions ..... | 27 |
| 5.3   | Algorithm characteristics.....                                  | 28 |
| 5.3.1 | 3-Color algorithm.....                                          | 28 |
| 5.3.2 | Rectangular algorithm.....                                      | 29 |
| 5.4   | Algorithm solution comparisons.....                             | 29 |
| 5.5   | Boundary analysis of camera to vertex count.....                | 31 |
| 5.6   | Impact of design choices.....                                   | 32 |
| 5.7   | Possible improvements .....                                     | 32 |
| 5.7.1 | Algorithm modeling improvements.....                            | 32 |
| 5.7.2 | Software improvements .....                                     | 33 |
| 6     | Conclusions.....                                                | 35 |
| 7     | References.....                                                 | 36 |
| 8     | Appendix A: Wordlist.....                                       | 37 |
| 9     | Appendix B: Walkthrough of algorithms .....                     | 38 |
| 10    | Appendix C: Program commands.....                               | 42 |

# 1 Introduction

Installing surveillance cameras today is a craftsmanship. Factors such as camera angle, distance to object, lighting and coverage are all calculated by hand since there is no formalized method for finding a good set of camera positions. In addition to environmental constraints and camera hardware limitations there are legal aspects and regulations that will further complicate camera placement. Currently it is up to the individual camera installer to, based on a mix of experience and mathematics; choose a suitable spot for each camera. Since this is very time consuming when dealing with large installations, and because it sometimes will result in inaccurate camera positions, there is a need for a tool that can create example solutions automatically. This thesis will try to find a solution to this camera placement problem and create a prototype program testing this solution.

The first goal of the thesis is to find and analyze a set of algorithms suitable for solving the camera placement problem. Camera limitations, environmental constraints and other factors with an impact on camera placement will also need to be identified in this phase of the thesis. The second goal is to develop prototype software for automatic camera placement that implements the most promising algorithms found in the problem studies. Algorithm accuracy, time efficiency and limitations will be analyzed, discussed and presented in this report.

This thesis contains an introduction to the problem family our camera placement problem belongs to, the art gallery problems. A definition of the original art gallery problem and other concepts needed for describing our specific variation of the art gallery problem are included. The most promising algorithms, the 3-Color algorithm and a rectangular algorithm, are described, analyzed and evaluated in both a mathematical and a practical context. Possible software and algorithm modeling improvements are also discussed. The created prototype software is showcased, explaining design choices and architecture. Joachim is responsible for the 3-Color and greedy rectangular algorithm analysis, the post algorithm fixes and the mathematical evaluation of the rectangular algorithm, including worst case camera boundry and time complexity analysis. Mikael is responsible for the basic rectangular algorithm, the camera placement limiters analysis, the prototype program design and the practical evaluation of the rectangular algorithm.

## 2 Background

### 2.1 Motivation

Today there is no known algorithm that can find the optimal number of cameras, given a representation of a room. This may not cause problems when placing cameras in simple environments such as a single rectangular shaped room or hallway but in a more complex environment, for example in a train station, a lot of calculations need to be made to ensure a set of camera positions that gives acceptable camera coverage and resolution of the monitored objects. The camera placement factors, among others camera angle, lighting and coverage all need to be calculated by hand, meaning that the success of the camera placement is highly dependant on the experience and skill of the camera installer. Today, simple software and spread sheets are used as aids when installing cameras in more complex environments. A program providing accurate and dynamic camera placement management in a graphical user interface would certainly prove to be a useful tool assuming the program results are plausible, that is, equal to or better than those derived by hand.

### 2.2 Definition and earlier research

The camera placement problem is a visibility problem part of a group of problems referred to as “Art gallery problems” [6]. A number of definitions are needed to set up the parameters of the problem:

**Definition 2.1: Simple Polygon**

A simple polygon  $P = (v_0, v_1, \dots, v_n)$  is a closed plane figure whose boundary  $bd(P)$  is composed of straight line edges  $(v_i, v_{i+1})$ ,  $i = 0, 1, \dots, n$ ,  $v_{n+1} = v_0$ , and where no two non-consecutive edges intersect. The points  $v_0, v_1, \dots, v_n$  are referred to as the vertices of  $P$ .

**Definition 2.2: Simple polygon with holes**

$P$  may contain holes. These are then represented as simple polygons without holes enclosed by  $bd(P)$ . The term *simple polygon* is reserved for simple polygons without holes. If the polygon may contain holes the term *simple polygon with holes* is used.

**Definition 2.3: Visibility in a polygon**

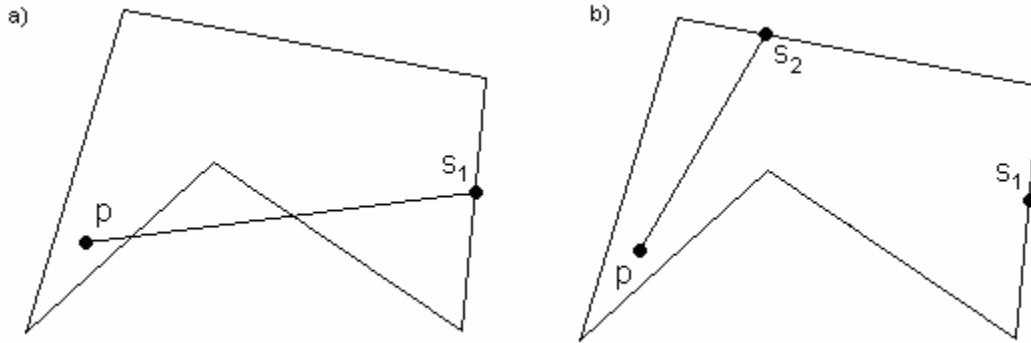
Two points  $p$  and  $q$  of polygon  $P$  are said to be visible from each other if the line segment from  $p$  to  $q$  lies completely in  $P$  as shown in figure 2.2.1a and b.

The Art gallery problem itself was first defined in 1973 by Victor Klee and can be formulated in the following way:

**Definition 2.4: The art gallery problem**

How many guards need to be placed to guard the interior of an art gallery? The art gallery is here represented by a simple polygon and the guards by points placed in a set  $S$ . When

the guards are placed in a way that any point  $p$  inside the polygon is visible from at least one guard the problem is solved.



**Figure 2.2.1a:** A room without full coverage since a point  $p$  can be found that is not seen by  $S_1$ . **Figure 2.2.1b:** After introducing a new guard  $S_2$  full coverage is achieved. Note that the set  $S$  is not optimal since  $S_1$  is redundant.

When the art gallery problem is defined as described above, finding the optimal solution will be NP-complete, this because a suggested solution can be verified in polynomial time while finding the optimal solution can only be done by testing all possible candidates [5]. It has been proven that it is always enough and at times required to use  $\lfloor n/3 \rfloor$  guards to guard the entire polygon, where  $n$  is the number of vertices in the polygon. There is however no guarantee that the number of guards is optimal when using this approach. So far no one has been able to construct an algorithm that gives the optimal solution regardless of polygon shape. Therefore restrictions are usually put on the definition of the problem and on the shape of the polygon making it possible to solve the problem in polynomial time. The restrictions chosen differ from application to application meaning there are as many variations of the art gallery problem and its solutions as there are applications. Variations include moving guards and limited range of guards, used in robotics navigation and in sensor coverage problems respectively. Other variations may focus solely on guarding the walls of the polygon, placing the guards in the center of the room [3]. The book “Art Gallery Theorems and Algorithms” by O’Rourke covers the art gallery problem well [6]. It contains algorithms that find the  $\lfloor n/3 \rfloor$  guards that are needed to guarantee full coverage in  $O(n \log(n))$  time<sup>1</sup> and discusses some of the variations of the problem.

The camera placement problem is as earlier mentioned a variation of the art gallery problem:

**Definition 2.5: The camera placement problem**

How many cameras are needed to guard the area inside a room? The room is represented by a simple polygon with holes. Guards are represented by cameras with a limited field of view and effective range. Coverage is calculated similar to in definition 2.3 but with the

<sup>1</sup> It has, since the book was published, been proved that this can be done in  $O(n)$  time [4].

addition that a point in the polygon only can be seen from a camera if it is within the effective camera cone.

Definition 2.5 covers the first four limiters in the next section. The other six limiters are excluded from the formal definition since they only impose further limitations on the first four limiters instead of on the actual camera placement problem. Because of this they will be considered secondary demands and not critical to the success of a camera placement algorithm.

Note that the restrictions added by the camera placement problem do nothing to improve the computational situation of the problem. It is still NP-complete since no simplifications made affect the computational complexity of the problem, making it very time consuming to find an optimal solution.

## 2.3 Camera placement limiters

The restrictions mentioned in previous sections are only a few examples of the limitations that separate the camera placement problem from the original art gallery problem. In this specific variation of the problem several other factors will also need to be taken into account. This section will define and discuss those factors.

**Field of view:** A camera is limited to a certain field of view, which depends on lens type and image sensor size. Vertical and horizontal fields of view are separated in the camera specifications since they are dependant of the height and width of the image sensor respectively. A normal value for the horizontal field of view is around  $50^\circ$  but for specialized cameras it will range up to a  $110^\circ$ .

**Line of sight:** The line of sight is limited by the effective range of the camera. In addition to the camera hardware limitations the line of sight is also, as defined in the original art gallery problem, blocked by walls and other obstacles as for example pillars and shelves.

**Coverage:** Coverage is defined as the percentage of the polygon area covered by the camera cones. The importance of coverage differs from application to application since while some applications need close to 100% coverage, others focus on covering a few selected areas of interest. Supermarkets for example, might choose 100% coverage threshold to detect shoplifting while a bank instead only place cameras with facing the doors and the cashiers to monitor the bank transactions.

**Resolution:** There are often requirements on how small an object monitored is allowed to be on the resulting footage, which effectively limits the range of the camera. Resolution is, similar to coverage, dependant on the application. Using a camera to count people in an open area would only require limited resolution since only a few pixels is needed to differentiate a person from the surroundings. Facial recognition on the other hand would require higher resolution. Once a lower bound for the size of the object monitored on the resulting footage has been decided, it is possible to calculate the effective range of the camera provided the lens type and the sensor resolution is known.

**Height:** The downwards angle of the camera have an impact on the camera cone. Cameras used for facial recognition may function poorly if the angle between the camera and the face of a person is too large, while other applications where people are counted or traffic situations are monitored not are affected to the same degree by the height of the camera.

**Lighting:** When placing cameras the lighting must be considered. High contrasts caused by for example windows or spotlights must be taken into account since the image quality of the footage of a camera placed in such an area will suffer. Lack of lighting will cause other problems since it would require the camera to have better, more expensive, optics and a light sensitive sensor better suited for these conditions.

**Camera facing:** Not only is the total coverage of a camera important, the facing of the camera is also relevant. Some combinations of cameras will be more favorable than others even if they all have 100% total coverage. For instance, cameras are often placed facing doors or other entrances when identification of people entering is desired. Sometimes the facing of a camera is more important than full coverage, especially if surveillance tasks have been defined.

**Surveillance tasks:** A surveillance task is an action that is performed at a predefined place in the monitored area. An example would be the activity around an ATM. The Swedish police recommend that there is at least one camera per task monitored [8]. This often means that the individual camera coverage will overlap or leave gaps. Usually this does not matter since it still results in an acceptable solution with the correct camera focus area.

**Legal aspects:** Camera placement is regulated by laws and licenses [7]. Separate licenses are needed for monitoring for example the interior and exterior of a store. This means that if a license for monitoring only the interior of the store is given, it is not allowed to have cameras with a view of the area outside the store. There are also areas such as bathrooms and changing rooms where camera surveillance is forbidden.

**Vandalism:** In environments where vandalism of cameras is probable, for example in train stations or prisons, it is common to require each camera to be seen by at least one other camera. This requirement minimizes the camera dead zone, making it impossible to vandalize a camera without being seen by at least one other camera.

## 2.4 Problem description

This thesis consists of two parts:

- Find a set of algorithms and methods suitable for solving the camera placement problem, while taking camera placement limiters into account.
- Develop prototype software implementing the chosen algorithms and methods.

Initially focus will be directed at examining existing algorithms that solve variations of the art gallery problem similar to our specific variant. The goal will be to find an already

existing algorithm that can be used, with few modifications, to solve the camera placement problem. If no such algorithm can be found, a new algorithm will be developed that can handle as many of the constraints in section 2.3 as possible.

The minimum algorithm requirement is that it is able to produce acceptable coverage with stationary cameras that have limited field of view and effective range without an unacceptable amount of overlap between cameras. The algorithm should also be able to handle polygons with holes because there normally are objects partially blocking the field of view in environments monitored by cameras. Another requirement is that it should be possible to introduce additional constraints from section 2.3 at later stages of development.

The most promising algorithm candidates will be implemented. Since the prototype software is not part of any time critical system the focus will not be time efficiency, but rather on handling as many of the limiters as possible while generating a set of camera positions of acceptable quality.

After the algorithms have been implemented a comparison of the algorithm results in regard to limiters, time complexity, solution quality and other factors will be made.

## 3 Algorithms

The art gallery problem is NP-complete, which means an algorithm is needed to approximate guard positions. First a well-known algorithm, the so called 3-Color algorithm is described. It is often used for solving art gallery problem variations close to the general definition. After that a rectangular algorithm is described, the authors own approach to finding an acceptable solution for the variation of the art gallery problem, defined as the camera placement problem.

### 3.1 General requirements of the algorithms

An algorithm that solves the camera placement problem should at a minimum be able to handle the following:

- Limited field of view of cameras.
- Limited effective range of cameras.
- Obstacles limiting the line of sight, also called polygon holes.
- A coverage threshold.

These points reflect the first four of the camera placement limiters described in section 2.3. The other limiters should be considered secondary and should only be added if an algorithm looks promising and permits it.

The camera placement problem in this thesis is modeled strictly as a two dimensional problem. A reason for this is that a third dimension would add a lot of complexity to the algorithm and not be of any use in most room polygons. Our opinion is that the thesis time is better used focusing on other features with a greater impact on the algorithm result. The main drawback with this limitation is that the algorithms will perform worse in areas with floors at more than one level, such as for example train stations where one might need to monitor both the track area and the platform area.

### 3.2 The 3-Color algorithm

In 1975 Chvátal established a theorem nowadays known as “Chvátal’s Art Gallery Theorem” [6]. It states that  $\lfloor n/3 \rfloor$  number of guards is needed to guard a polygon with  $n$  vertices. It is however occasionally possible to guard a polygon with fewer guards, depending on the polygon shape. The theorem makes two simplifications; guards have a 360° field of view and infinite effective range. The algorithm used to prove this theorem is referred to as the 3-Color algorithm.

The idea behind the 3-Color algorithm is that only one guard is needed to guard a triangle; therefore polygon triangulation is used to divide the polygon into triangles. If a polygon vertex is shared by many triangles, putting a guard in that vertex is enough to guard all of those triangles. Vertex coloring is used to choose the  $\lfloor n/3 \rfloor$  guards needed for guaranteed coverage.

### 3.2.1 Polygon triangulation

Polygon triangulation is a method used to divide a polygon into a set of triangles. There are several ways of solving a triangulation problem, the one used here is known as ear trimming. A vertex and its two adjacent neighbors form an ear if a line can be drawn between the neighbors without crossing any other edge in the polygon. The algorithm uses this approach to find ears and, when found, trims them from the polygon. If the algorithm fails to find an ear in one vertex it moves on to the next. The algorithm starts in one of the polygon vertices and is repeated until only one triangle remains.

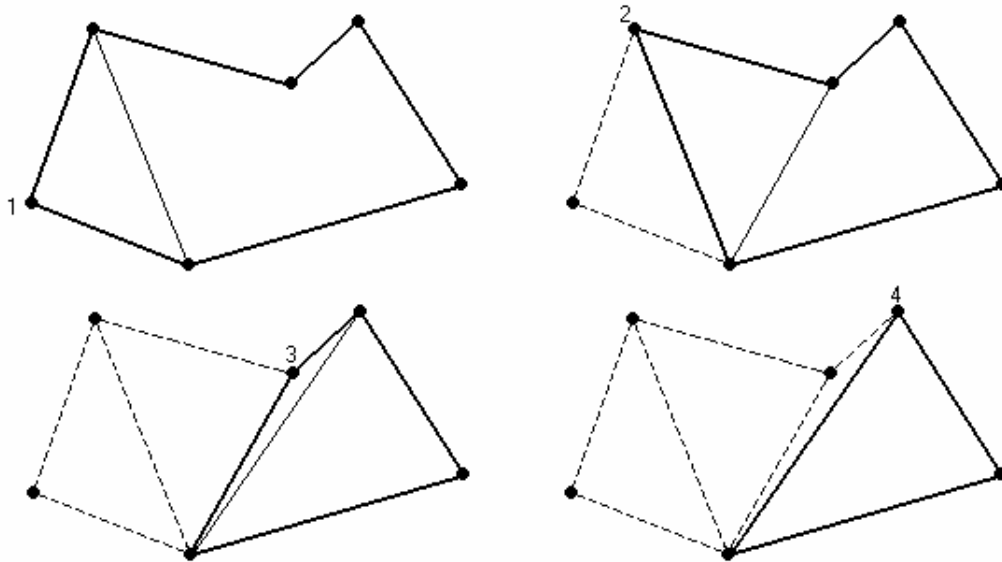


Figure 3.2.1: A polygon triangulated clockwise starting in point 1. When an ear is found it is trimmed, shown by the dashed lines.

Triangulating a polygon with holes is possible by, for example, extending a line from the polygon hole to the main polygon. Then the hole can be considered a part of the polygon and then it is possible to triangulate using the same method as earlier. There is seldom one unique triangulation of a polygon. Depending on which order the vertices of the polygon are handled the triangulation algorithm may produce different sets of diagonals.

### 3.2.2 Vertex coloring

The triangulation results in a triangle grid which is colored with three different colors, in the example below red, green and blue. The vertices are colored so that every triangle has one vertex of each color.

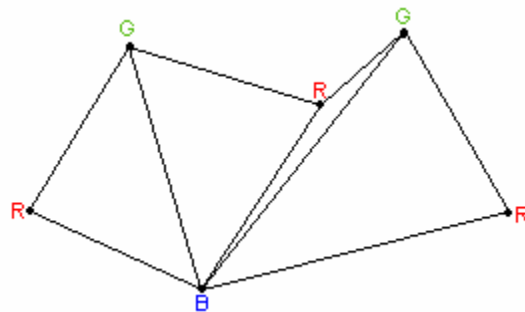


Figure 3.2.2: A polygon with red, green and blue vertices.

The coloring algorithm is quite trivial. One random triangle in the grid is colored. After that, adjacent triangles can be colored since they share two vertices with the first one, and only miss one color. When all vertices have been colored in this manner, the least used color is chosen, blue in the figure below, and guards are put in those vertices. Since a guard placed in a vertex guards all triangles including that vertex, full coverage is achieved.

### 3.3 Rectangular algorithm

The rectangular algorithm has an approach very similar to the 3-Color algorithm. Both divide the polygon area into small pieces and place guards, in this case cameras, which guarantee coverage on a local scale. When all the pieces are added together full coverage on a global scale is achieved. However, instead of directly dividing the polygon into triangles as the 3-Color algorithm does, this algorithm tries to find the minimal set of unique rectangles that cover the polygon. These rectangles are then split into triangles and one camera is placed at one end of each triangle to achieve full coverage. In a way this method can be viewed as a form of triangulation where maximization of triangle size within the rectangle is the goal. For details see the pseudo code of the rectangular algorithm steps in Appendix B.

The rectangular algorithm is divided into three steps:

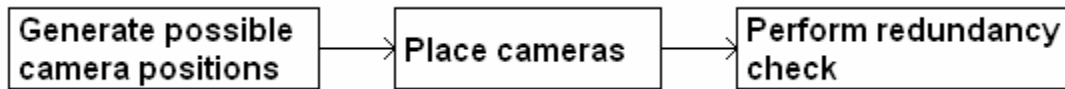


Figure 3.3.1: Workflow of the rectangular algorithm.

- 1) **Generate possible camera positions:** This is done by dividing the room polygon into rectangles, using the corners as a possible camera position. This method is described in section 3.3.1 and 3.3.2.
- 2) **Place cameras:** Two different camera placement strategies are described in section 3.3.3. The goal of both strategies is to find a subset of the generated camera positions that solve the camera placement problem.
- 3) **Perform redundancy check:** Since both strategies optimize the problem locally, per rectangle, it may be possible to find redundant cameras after the subsolutions have been combined together. This task is covered in section 3.3.4.

The obvious drawback with this algorithm is that it can only be used on polygons with orthogonal angles. In agreement with the supervisor of this master thesis it was decided that this was an acceptable limitation, since many rooms are shaped in this way, assuming that the algorithm was designed in a way that it could at later stages be improved to allow arbitrary shapes. A definition of the orthogonal polygon is needed to formalize the algorithm:

**Definition 3.1: Orthogonal Polygon**

An orthogonal polygon is a polygon that has a  $90^\circ$  angle between the two edges that intersect at each vertex. The term *orthogonal polygon* refers to an orthogonal polygon

that is also simple. *Orthogonal polygon with holes* is used to refer to simple polygons with holes where both the main polygon and its holes are orthogonal.

### 3.3.1 Finding rectangles in orthogonal polygons

The goal of the rectangle finding algorithm is to find a set of rectangles that combined together cover the room polygon. The rectangles and consequently the triangles are maximized, considering area, but they are not allowed to enclose one another since redundancy must be avoided. The shape of the rectangles is limited by the camera field of view, the effective range and shape of the room polygon. This will often result in overlap between the rectangles, but it is allowed since it can be used to further minimize the number of cameras used in later stages of the algorithm.

Rectangles are found by sweeping through the room polygon, using the leftmost vertical edge as starting point, searching for other vertical edges. When a vertical edge with the inside of the polygon on the right side is found it is first extended vertically in both directions if possible and then used as a base for a rectangle. If the vertical edge has the inside of the polygon to their left instead it is used to close previously found bases to form rectangles where it is possible. If the base and the closing edge do not have matching start and end point new bases may have to be introduced. By using this strategy all relevant, unique, rectangles will be found when the rightmost vertical edge is reached regardless if the polygon contains holes or not.

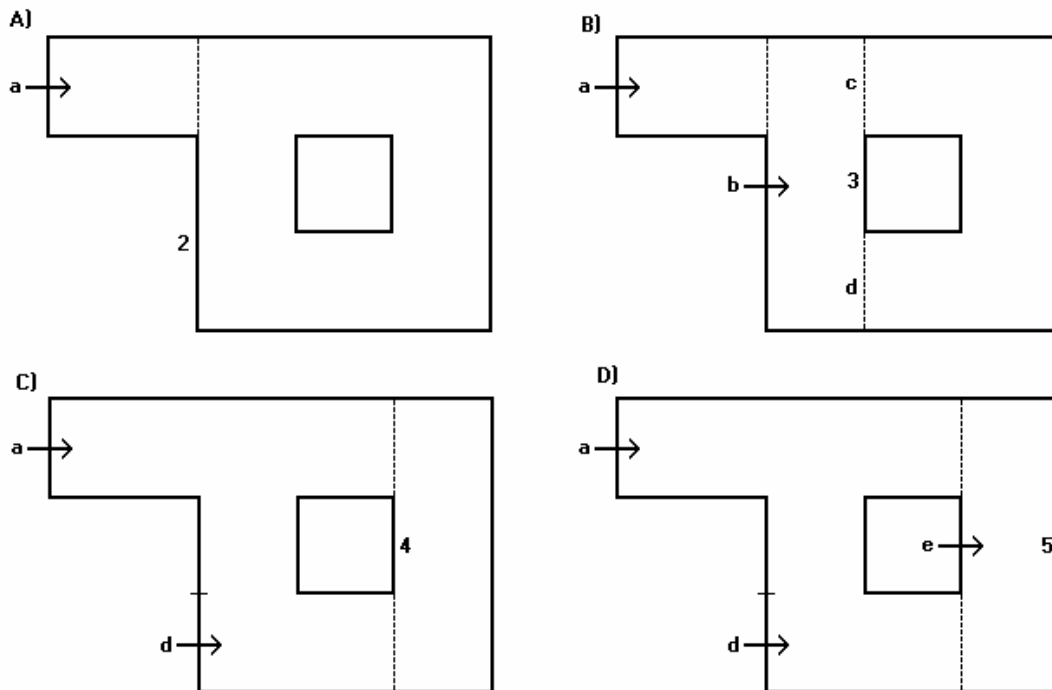


Figure 3.3.2: How to find rectangles in an orthogonal polygon.

An example of the sequence of actions in the find-rectangle algorithm explaining figure 3.3.2:

- A) Add the leftmost edge as base *a* to the list of bases and let it sweep into the polygon until reaching the x-coordinate of edge 2. Since this edge does not overlap with base *a* no rectangle can be created, instead edge 2 is extended vertically to the horizontal edge and added to the list as base *b*.
- B) The sweep continues from base *a* and *b* separately until reaching edge 3 where a rectangle is created from base *b* since this edge interferes with its sweep. Two new bases, *c* and *d*, are created here but *c* is ignored since it is fully enclosed by base *a*, which has yet to become a rectangle. Base *b* can now be removed from the list.
- C) The sweep continues from base *a* and *d*. Since no rectangle can be created when reaching edge 4 it is extended in both vertical directions and added to the list as base *e*.
- D) The bases *a*, *d* and *e* all continue until the final edge 5 is reached where all the bases left in the list creates one rectangle each and are removed from the list. The resulting rectangles can be seen in figure 3.3.3.

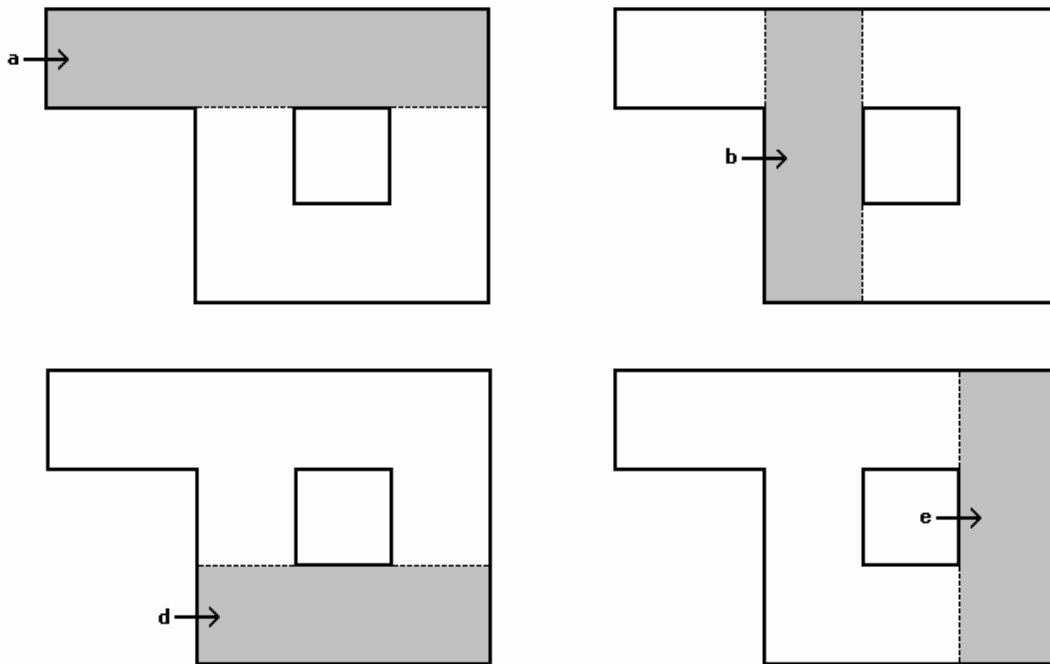


Figure 3.3.3: Rectangles found by each of the sweeps.

### 3.3.2 Covering rectangles with cameras

The number of cameras needed to cover a rectangle depends on the camera field of view and effective range. If a camera model has over  $90^\circ$  field of view it would be enough to place one camera per rectangle, assuming the length of the rectangle diagonal is not greater than the camera range. Most cameras only have a  $50^\circ$  field of view though, which means that a minimum of two cameras per rectangle are needed for full coverage.

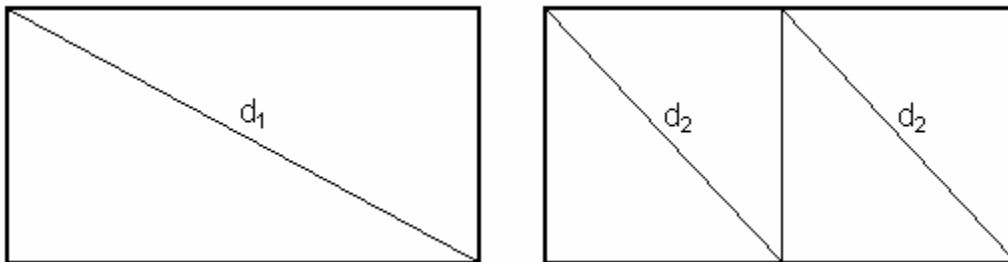
The camera positions are found by drawing the diagonal of the rectangles. This splits the rectangle into two triangles guarded by one camera each. The cameras are placed in the corner with the smallest angle in each of these triangles. The facing of the camera is found by letting one of the camera cone sides align to the wall.



**Figure 3.3.4:** Two cameras are needed to fully cover this rectangle. The dark green areas are covered by only one camera cone while there is an overlap of the two camera cones in the light green area.

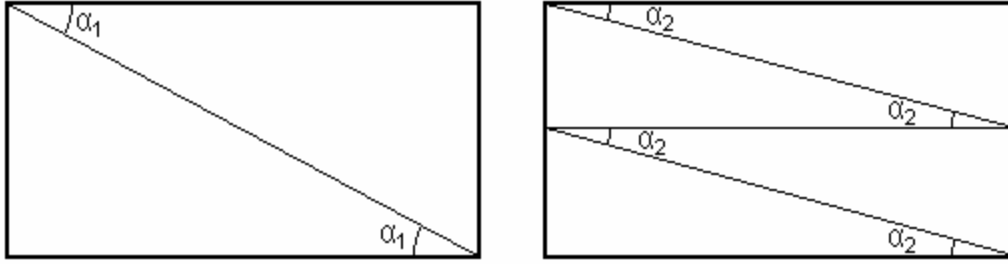
As shown in the above figure there will be some overlap in the coverage of each camera cone if the camera field of view is larger than the angle of the triangle corner where it is positioned. In addition to the previous mentioned advantages of overlap it also makes the setup more robust, a little protection against the facing being a few degrees off during installation.

If a resulting rectangle has a diagonal length greater than the effective camera range it is necessary to split the rectangle as shown in figure 3.3.5



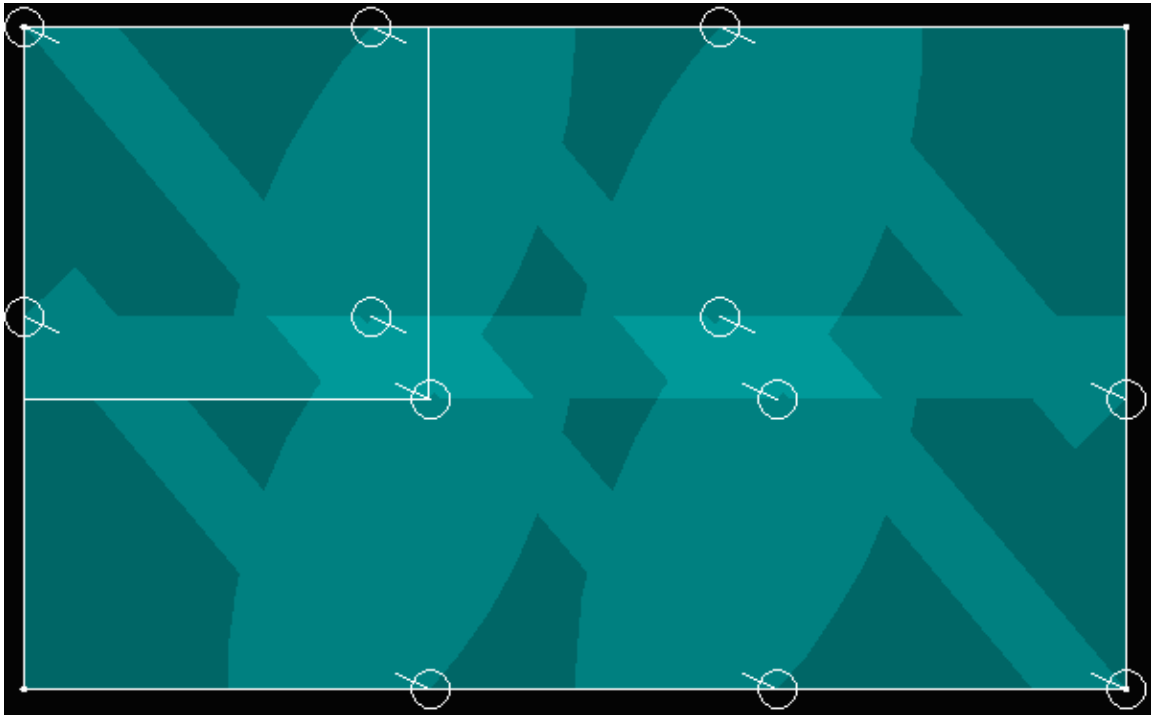
**Figure 3.3.5:** The length of the diagonal  $d$  is reduced by splitting the rectangle breadthwise.

A split is also necessary if a resulting triangle does not have a corner with an angle smaller than the camera field of view. figure 3.3.6 shows how to reduce the angle  $\alpha$  by performing a lengthwise split of the rectangle.



**Figure 3.3.6: The angle  $\alpha$  is reduced by splitting the rectangle lengthwise into two pieces.**

Note that the breadthwise split must be done first because it increases the angle of the triangle corner. The lengthwise split on the other hand decreases both the diagonal length and the relevant angles. It may also be necessary to divide the rectangle into more than two subrectangles if it is very large or if the camera field of view is very small. If the division into subrectangles results in overlap between the rectangles it is evenly divided between all of them. These two techniques guarantee that enough cameras are placed to achieve full coverage in any rectangle.



**Figure 3.3.7: A rectangle split into 6 subrectangles with 2 cameras guarding each one. The upper leftmost subrectangle is outlined. Note that the rectangles will overlap because their size is determined by the shape of the camera cone.**

### 3.3.3 Placing cameras

After a set of unique rectangles has been found the camera positions are established. In this thesis two different strategies for placing cameras in rectangles have been tested.

### **3.3.3.1 Basic strategy**

First the most basic strategy was tested. It divides each rectangle into two triangles along the diagonal from the upper left corner to the lower right corner. One camera is then placed in the upper left corner observing the upper triangle and another one is placed in the lower right corner responsible for the lower triangle with their camera cones aligned to the side of the rectangle to maximize effective camera cone usage, as shown in figure 3.3.4.

### **3.3.3.2 Greedy strategy**

The second strategy takes a greedy approach to the problem. This strategy uses both diagonals to divide the rectangles into triangles simultaneously, meaning it treats all four corners of each rectangle as possible camera positions.

The strategy works iteratively, calculating the increased coverage of each possible camera position in each step, adding the position with best coverage increase to the solution. Camera positions connected to a wall and not already occupied by a camera are prioritized. This procedure is repeated until the desired level of coverage is achieved or if adding additional cameras do not improve coverage.

### **3.3.4 Removing redundant cameras**

Since the rectangle finding algorithm is designed to maximize the rectangle size, limited only by camera specifications and the shape of the room polygon, there will almost certainly be an overlap of the effective camera cones in the final camera set. Therefore it is possible to further minimize the number of cameras by performing a final cleanup after the algorithm has finished, searching for redundant cameras.

First the coverage is calculated, one time for each camera, for a set without that camera. Then the camera with the least impact on coverage is removed if the resulting coverage is large enough to satisfy the chosen coverage threshold. Cameras at positions not connected to any of the room polygon edges are prioritized for removal because the actual camera cone is better utilized with this approach. For positions in the middle of a room a dome camera would be more suitable since that would utilize the camera field of view in a better way, but unfortunately that type of cameras are not in the scope of this thesis. The search for redundant cameras is repeated until no more cameras can be removed.

## 4 The Prototype Software

### 4.1 Program survey

The created software provides graphical and mathematical support for solving camera placement problems. The user defines the room polygon and chooses camera model settings. Then the program generates a solution of the chosen algorithm type, based on that input. Since the quality of the solution is difficult to approximate with an algorithm it is possible for the user to edit the solution camera properties. It is also possible to add cameras before running the algorithm, indicating areas that must be monitored from a certain point of view.

This section will go through the process of creating a camera placement problem solution by entering a room polygon, choosing camera and algorithm settings. Available program commands are covered in Appendix C.

#### 4.1.1 Defining the room polygon

The room polygon is created from user input. The user chooses a starting point and then proceeds with adding the polygon vertices. After closing the main polygon the user may add additional polygons inside the main polygon representing obstacles.

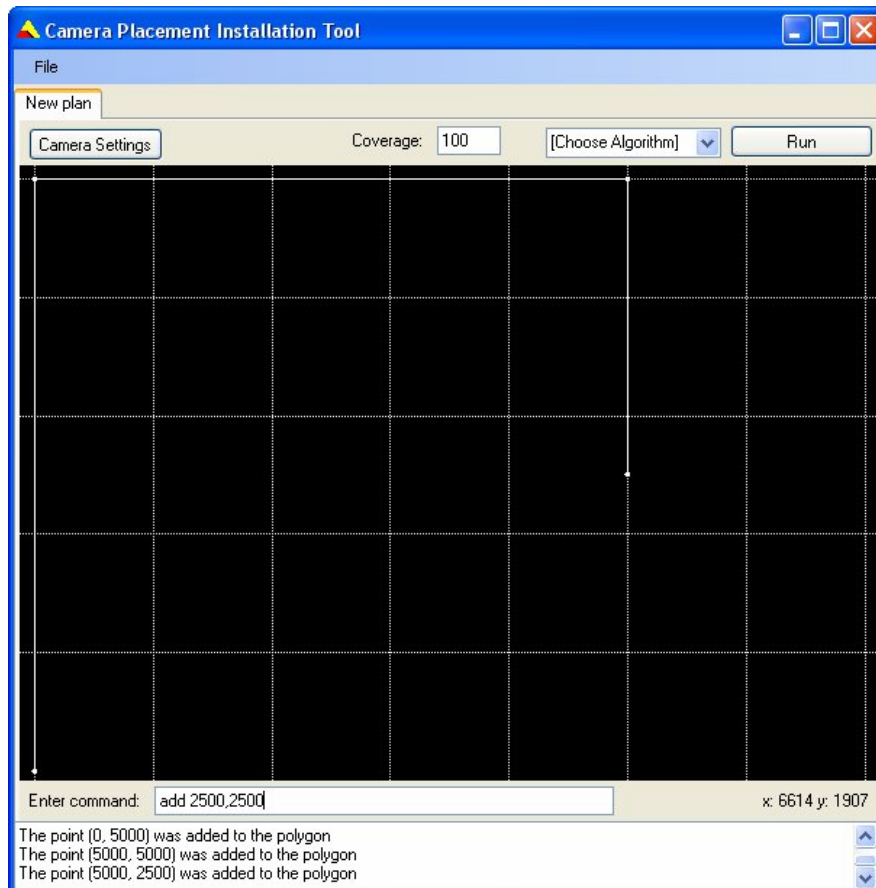


Figure 4.1.1: Entering the polygon vertices.

### 4.1.2 Setting camera field of view and effective range

The user can either set camera field of view and effective range manually, by choosing the custom camera alternative, or set the following properties in the camera settings window:

- Camera model
- Lens model
- Image sensor model
- Size of the object monitored
- Minimum percentage of the object height on the resulting image

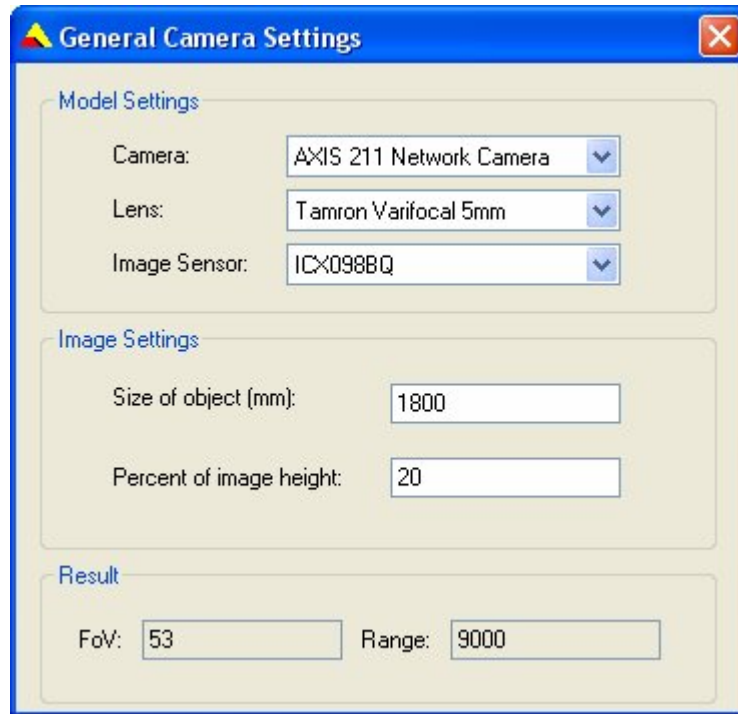


Figure 4.1.2: The general camera settings dialogue window.

Camera field of view and effective range are calculated from the camera specification properties combined with the user input as follows:

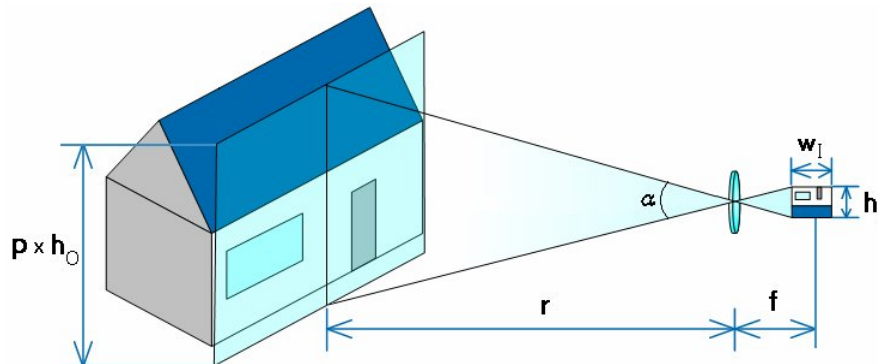


Figure 4.1.3: The relation between the variables in the field of view and effective range formulas.

$$\alpha = 2 \arctan\left(\frac{w_l}{2f}\right) \quad r = \frac{f \times h_o}{h_l \times p}$$

- $\alpha$  = Camera field of view
- $r$  = Camera effective range
- $f$  = Lens focal length
- $w_l$  = Image sensor width
- $h_l$  = Image sensor height
- $h_o$  = Object height
- $p$  = Minimum percentage of the object height on the resulting image

### 4.1.3 Predefining cameras

It is possible to predefine cameras before executing the algorithm. The dialogue window for adding cameras is very similar to the camera settings window, but with additional fields for coordinates and angle, measured counter-clockwise from the positive x-axis. These cameras are guaranteed to be part of the solution since the algorithm will not manipulate them.

**Add New Camera**

**Model Settings**

Camera:

Lens:

Image Sensor:

**Image Settings**

Size of object (mm):

Percent of image height:

**Result**

FoV:  Range:

**Positioning**

X:  Angle:

Y:

OK Cancel

Figure 4.1.4: The add camera dialogue window.

#### 4.1.4 Choosing algorithm settings

Coverage threshold and algorithm type are the last things to set before generating a solution. The coverage threshold is a parameter that controls what percentage of the room polygon that needs to be covered by the cameras. There are three algorithms to choose from: 3-color, basic rectangle and greedy rectangle. For detailed information about the algorithms, see section 3. Note that an orthogonal polygon is required for the rectangular algorithms to produce a solution.

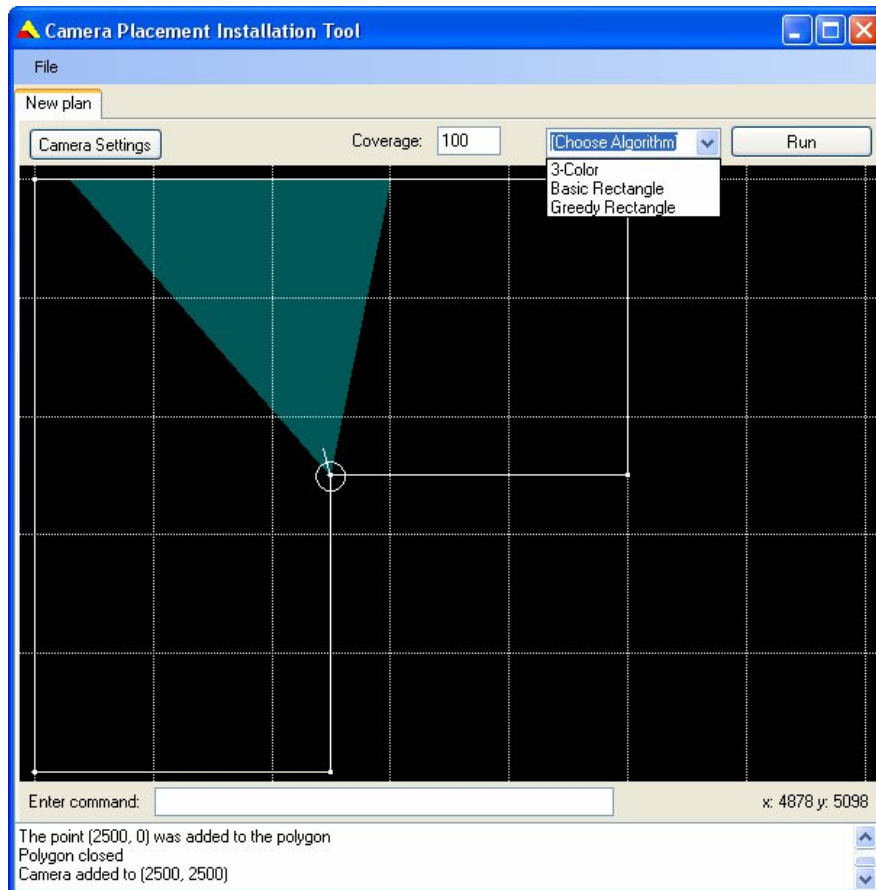


Figure 4.1.5: Choosing algorithm in the input view after one camera has been predefined.

#### 4.1.5 The solution view

If the algorithm produces a result the solution view is shown. The room representation along with the cameras generated by the algorithm is shown in the window. The camera visibility cones are displayed as green areas, brighter areas showing overlap. The camera field of view and effective range in addition to the number of cameras in the solution and the estimated coverage is shown on top of the view.

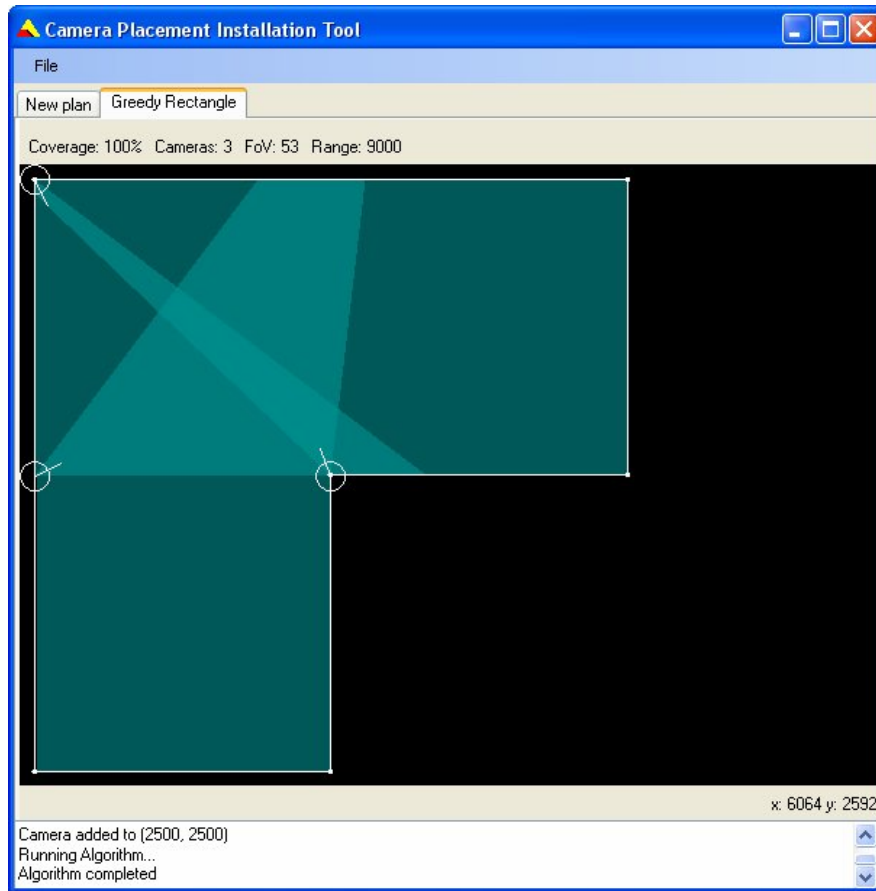


Figure 4.1.6: A solution generated by the greedy rectangle algorithm using one predefined camera.

#### 4.1.6 Editing solutions

In the solution view it is possible to view and edit camera properties by clicking on the camera symbols. In addition to the dialogue described in section 4.1.3 there is a “Copy to input” button, which copies the selected camera to the input view to be used as a predefined camera when the algorithm is executed the next time. This allows the user to find a solution in an iterative manner. There are also controls for browsing through the solution cameras. Changes to the camera properties are applied directly to the solution when pressing the “OK” or the “Apply changes” button.

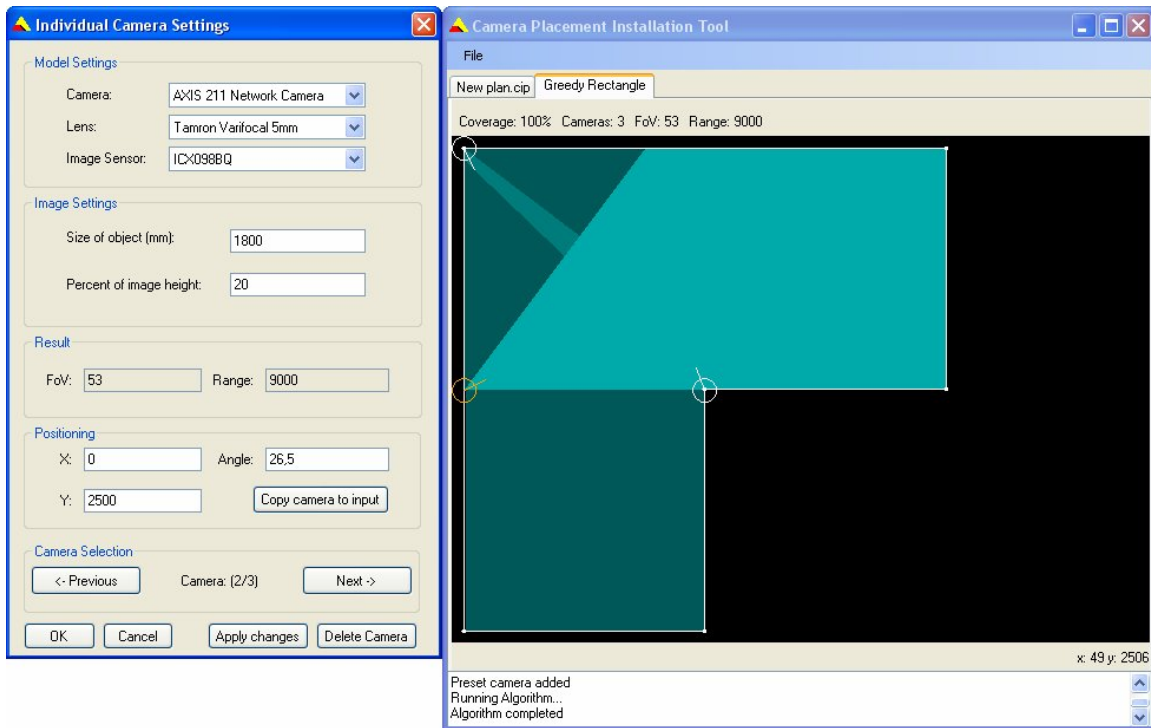


Figure 4.1.7: Editing cameras in the solution view.

## 4.2 Architecture

The prototype software is divided into three modules with clear areas of responsibility: the User Interface, the Data Handling and the Algorithm module. In addition to the classes described below a number of utility and geometric operation classes are used. Those were excluded to simplify the class diagram and since they are not of interest when discussing the general architecture.

### User Interface

- **MainForm:** Handles basic window commands and tab functions.
- **InputUserControl:** Represents the input tab. Contains a command line, algorithm selection and a paint area.
- **SolutionUserControl:** Represents a solution tab. Contains a paint area and an algorithm summary.
- **CSPopupForm:** Represents a camera settings popup used both for general camera settings, individual camera details and for adding new cameras.

### Data Handling

- **DataHandler:** Acts as connection point for all data in the program. Used by the user interface to fetch the camera model database, read/write the plan files and to access the algorithm module.
- **FileHandler:** Handles all file read/write operations. Converts from and to the XML-structures used by the files.

- **XMLCameraDatabase:** Represents the camera model database. Contains data such as camera model name, lens name, lens focal length, image sensor name and image sensor dimensions.
- **Cfg:** Contains the polygon and result objects. The polygon object contains the representation of a room and the result objects have a list of camera objects correspondent to those shown in a solution tab.

### Algorithm

- **AlgorithmController:** The main access point of the algorithm module. Contains methods that control the algorithms.
- **IAlgorithm:** Defines an algorithm. ThreeColorAlg, BasicRectAlg and GreedyRectAlg all implement this interface. Details of the implemented algorithms can be found in Appendix B: Walkthrough of algorithms.

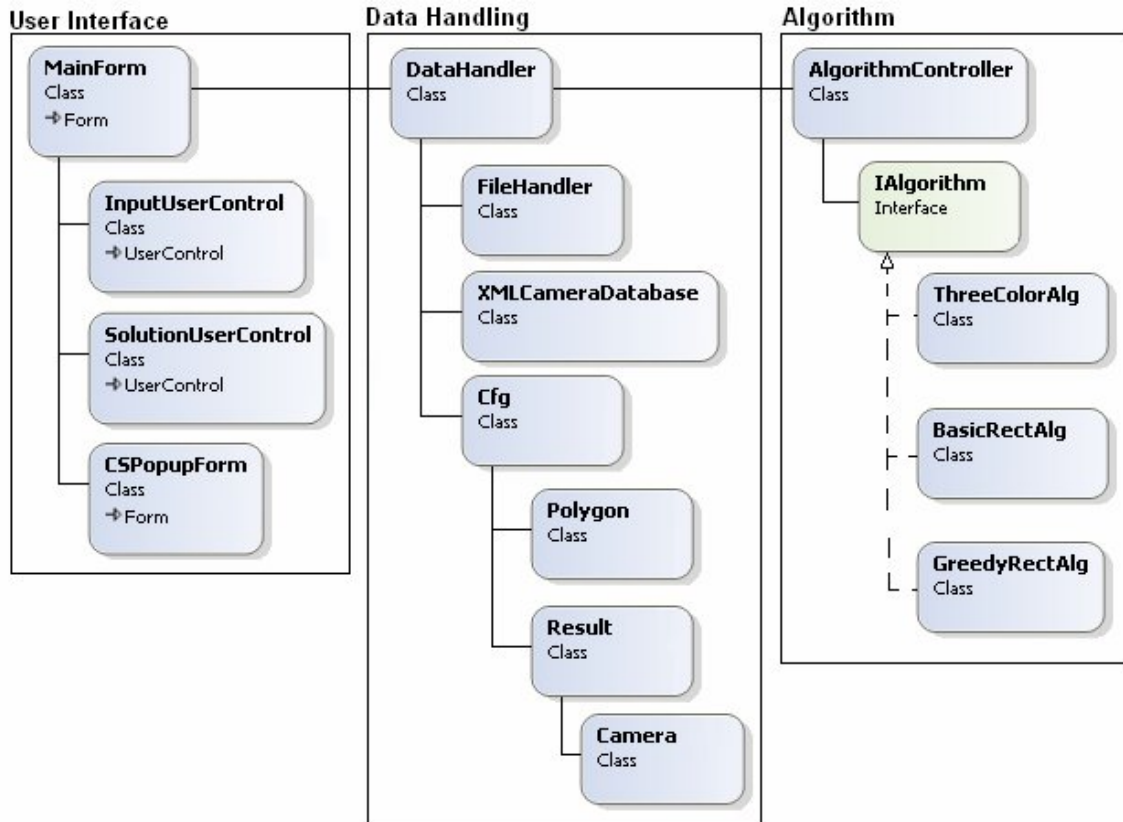


Figure 4.2.1: The class relation diagram.

## 4.3 Design choices and motivations

Since we did not have much faith in the 3-Color algorithm we chose to implement it in the simplest way possible. We found an implementation of the algorithm on an internet community homepage called *The Code Project* [2] that uses a triangulation algorithm with a time complexity of  $O(n^3)$ . We chose to modify this implementation instead of writing our own version from scratch using the better  $O(n \log(n))$  algorithm, because we

did not want to spend a lot of time implementing something we did not believe in. For the same reasons the coloring is done in  $O(n^2)$  instead of  $O(n)$ . We reasoned that if the 3-Color algorithm produced better results than anticipated we could always implement better versions of these algorithms where time complexity might become an issue. This however was not necessary since the 3-Color algorithm, as expected, generated rather unimpressive results as described in section 5.3.1.

Both of the rectangular strategies<sup>2</sup> have a time complexity of  $O(n^4s)$  where:

- $n$  is the number of vertices of the room polygon.
- $s$  is the number of sample points in the coverage algorithm.

When implementing the rectangular algorithm we used the same strategy as in the 3-Color case. The aim was to have a working rectangular algorithm as quickly as possible to conclude if the algorithm had any potential. The bottleneck in the current rectangular implementation is the coverage calculation algorithm. It depends on a large number of sample points in the room polygon, from where visibility checks are made. The number of points grows with the total area of the room polygon, making the algorithm very time consuming for large polygons. In consultation with our supervisor we decided that it was more important to add additional functionality than to improve the time complexity of the coverage algorithm.

The software is written in a modular way to simplify the replacement and improvement of the different modules. We have also chosen to save our files in an XML-structure to simplify further development and integration with other software. For the same reasons we are using an XML-structure in our camera model database.

---

<sup>2</sup> See Appendix B: Walkthrough of algorithms

## 5 Evaluation

### 5.1 Initial observations

The 3-Color algorithm is the most common method used to solve art gallery problems but despite that it was clear to us early on that it would not be sufficient to solve our specific problem variation. The original form of the algorithm does not take any of the limiters described in section 2.3 into account and our attempts to modify it all ended in undesired algorithm results such as several cameras in one position or triangle sides longer than the effective range of the camera.

A lot of effort was put into finding a more suitable algorithm, resulting in the rectangular algorithm described in section 3.3. The rectangular algorithm was mainly explored because rooms are most commonly rectangular or can be divided into rectangles. Only two cameras will ever be needed to achieve local coverage in a rectangle because of the camera field of view and effective range both can be compensated for by splitting the rectangle into two or more subrectangles as described in section 3.3.2.

At first glance the rectangular algorithm can seem to function in a similar way to the 3-Color algorithm as they both ultimately divide the polygon into triangles, but there are a few important differences:

- Overlap between triangles is allowed in the rectangular solution. This means that they can be optimized to fit a camera model, maximizing the effective camera cone usage, as shown in figure 3.3.7 where the area of the subrectangles are maximized considering the shape of the effective camera cone.
- The resulting triangles are independent of how the polygon is entered. The 3-Color triangulation algorithm produces different results depending on which vertex is used as starting point and the order polygon vertices are handled in.
- The camera facing is easy to control in the rectangular algorithm as each camera only need to cover one triangle. The 3-Color algorithm on the other hand is not able to handle a limited field of view and cameras will therefore monitor multiple triangles. This result in difficulties when aligning the camera cone to a specific triangle side and might also end in several cameras in one position.
- The rectangular algorithm is limited to orthogonal polygons, restricting the number of manageable room shapes.

Another early observation we made is that the rectangular algorithm finds the corner camera positions of a room, which are considered good positions for many reasons, such as minimizing the dead zones and maximizing use of the actual camera range.

The 3-Color and rectangular algorithm advantages and drawbacks will be more closely examined and discussed in the later sections of the evaluation chapter.

## 5.2 The quality of a solution and individual camera positions

Historically the optimal solution to art gallery problems has been defined as finding a solution with the minimum number of guards that satisfy the parameters of the problem. In reality this is an abstraction that produces results with little or no practical use. The placement of the cameras is often affected by other parameters of the problem to such an extent that the number of cameras is considered of less importance. It is for example optimal to guard a square room with two cameras as can be seen in figure 3.3.4 if only considering geometrical factors. The problem is that we have guidelines written by the Swedish police that dictate that one camera should be used for every surveillance task [8]; so if a square room had four defined surveillance tasks, four cameras would be needed in a practically sound solution, which would be considered suboptimal if only considering the number of cameras. The number of cameras also becomes less important as their position in relation to each other and various legal aspects begin to put demands on the solution. Also there will, in some applications, be a need for overlap because a surveillance task might need to be monitored from multiple angles at a time. For a solution to be usable in a practical context numerous parameters needs to be taken into account, making the problem increasingly complex and therefore an optimal solution hard to define. We have the problem that from a practical point of view, considering all these parameters, several solutions will be satisfactory while almost certainly none of them would be optimal if judged only by the number of cameras. We are in other words working with a somewhat ambiguous definition of the optimal solution.

All relevant camera limiters are described in section 2.3. In addition to the problem defining first four limiters, we have defined six other factors with an impact on the quality of an individual camera position. First of all, depending on application, there are more and less suitable heights to put a camera. When in need of facial recognition it is preferred to place a camera with its angle to the faces of the people monitored at a minimum while cameras monitoring train stations for example should be placed out of reach to discourage vandalism. Secondly positions with large portions of the field of view dominated by windows should be avoided because placing cameras against the light reduces image quality. Also there might be restrictions on what may be filmed outside the window due to legal constraints. Furthermore, the camera dead zone could cause coverage issues if not minimized by placing the cameras in corners or where another camera is able to cover that area. There are also a few issues that need to be considered when using a surveillance task coverage approach, since it for example would be problematic if a camera monitoring a bank entrance is placed where only the backs of people entering can be seen.

The camera placement limiters only cover the issues of a single isolated camera position. Finding an acceptable solution becomes increasingly more complex when trying to solve a problem where several cameras are needed since the mutual positioning of the cameras can be used to compensate for each others shortcomings. For example, a camera has dead zones both directly beneath it and at its sides. When two cameras are placed facing one another they cover each others dead zones. The dead zone of a camera can also be minimized by placing it in a corner, reducing the dead zones at its sides, and by lowering its position, reducing the dead zone beneath it. Overlap between the effective camera

cones can also affect the quality of the solution. It is most often very hard to avoid at least some overlap in a solution and sometimes it is even desired to allow overlap in a few selected areas. Overlap is best put to use in the center of areas where several objects or tasks are monitored and not in corners or in dead ends where there is low activity. Overlap should also, when possible, be generated by cameras facing each other to use the overlap to monitor an area from several different angles. Another factor that may be taken into account is that we are modeling cameras with a limited field of view and effective range, cutting the coverage to zero outside this cone. In reality this is not entirely accurate since cameras can see objects beyond their effective range, just not with the desired resolution. If this is considered when placing cameras further optimization of camera utilization may be possible, depending on the usage context.

## 5.3 Algorithm characteristics

### 5.3.1 3-Color algorithm

The main advantage of the 3-Color algorithm is its time efficiency. Triangulation can easily be done in  $O(n \log(n))$  and coloring in  $O(n)$  thus making it very fast. Unfortunately this is about the only advantage this algorithm has when solving the camera placement problem.

The list of drawbacks is extensive:

- The algorithm does not handle limited camera field of view and effective range.
- It is very difficult to adapt the algorithm to handle the environmental and legal limiters from section 2.3.
- The solution is not guaranteed to be optimal.
- The algorithm does only place cameras in corners.
- The algorithm has no support for polygons with holes.

It can be concluded that the unmodified 3-Color algorithm does not have any of the characteristics mentioned in section 3.1, making it a poor choice of algorithm for solving the camera placement problem.

We considered the possibility of adding camera field of view and effective range support by controlling the shape of the triangles found during the triangulation but it proved to be difficult. We could not find a method that allowed control over both the angle and the side lengths of the triangle while finding a suitable coordinate for the third triangle corner when splitting triangles. Another problem is that the 3-Color algorithm is designed to let guards be responsible for multiple triangles, which is a bad strategy for the camera placement problem since the camera field of view is limited. There was also consistency problems related to the triangulation because it generates different results depending on which vertex the algorithm starts with.

This being said, the result of the 3-Color algorithm might very well be good enough in some situations. When the cameras are placed in corners from where it is possible to see most of the room, in corners evenly spread throughout the polygon, using an effective

range larger than the longest side of a room and when the desired coverage is below 100% 3-Color may generate acceptable solutions. In other words luck is a big factor in generating good results with the 3-Color algorithm.

### 5.3.2 Rectangular algorithm

The rectangular algorithm has all the basic characteristics specified as the minimum requirement for algorithms solving the camera placement problem in section 3.1. During the implementation we realized that it would be fairly simple to add support for more than the first four, problem defining, camera limiters if needed at later stages of development, at least in the greedy strategy.

There are three major drawbacks with the rectangular algorithm:

- It is only able to handle orthogonal polygons. Most rooms have this shape but it should still be considered a drawback since more complex room polygons will not be possible to be used without approximating rounded surfaces with rectangles. As explained in section 5.7.1 it may be possible to improve the algorithm to handle any quadrilateral shape negating this drawback.
- The algorithm has a bad time complexity. Both the greedy and the basic strategy need to do coverage calculations, which is very time consuming. The time consumption of the greedy variant of the algorithm is far worse than for the basic variant since it has to calculate coverage both during the initial camera placement and in the final cleanup phase.
- The algorithm produces no mathematically optimal solution. The rectangular algorithm is optimal on a local scale, per rectangle, as shown in figure 3.3.4. The final solution is not guaranteed to be optimal though since it is combined together from the different rectangles. The risk of getting a suboptimal solution might be lowered by using different camera placement strategies, for instance dynamic programming or some kind of genetic algorithm, but at this point it is only speculation.

## 5.4 Algorithm solution comparisons

We have omitted the poor result of the 3-Color algorithm from this section since comparing it to the result of the rectangular algorithm would make no sense. We will instead focus on the differences between the basic and the greedy strategies of the rectangular algorithm.

**Number of cameras:** The greedy strategy is generally better when it comes to the number of cameras used in a solution since it has twice as many camera positions to choose from. In some cases though, the basic strategy will use a fewer number of cameras, most likely because the greedy strategy avoids placing cameras away from walls and on top of other cameras. In cases where such a position would result in the greatest coverage increase and the second best camera position has a substantially lower coverage increase the basic strategy will probably produce a solutions with fewer cameras, but at the cost of cameras on top of each other or in other undesired positions.

**Overlap:** The greedy strategy appears to be more successful in placing overlap at relevant locations than the basic one, at least in larger rooms. This is a positive side effect of placing cameras on walls since the bulk of the effective camera cone then is put in the center area of a room and therefore also the majority of the overlap. In rooms that are so small that the effective camera cone reach from wall to wall there is little difference between the strategies.

**Facing of cameras:** There is very little difference between the greedy and basic strategy concerning the facing of the cameras. The greedy algorithm might have a slight advantage since it has more camera positions to choose from but it is difficult to verify since these additional positions generate other advantages that might make the facing improvement negligible. Both strategies share the problem coming from letting the effective camera cone follow a side of the rectangle it guards to decide the facing of a camera. This may not always be a good strategy since people entering a room through a door in a wall with a camera aligned in this way only can be observed from the side and from behind. Both strategies also have problems in a few situations where the algorithm removes redundant cameras from only one side of a room. Most cameras then see the room from similar angles, which is something that should be avoided if possible.

**Time consumption:** The basic strategy seems to be about twice as fast as the greedy strategy at a 100% threshold. This is mostly because the greedy strategy does coverage calculations both during camera placement and during removal of redundant cameras while the basic strategy only does coverage calculations for the latter.

**Response to changes in coverage threshold:** With a coverage threshold of 100% the basic strategy is approximately twice as fast as the greedy one. This gap between them diminishes as the threshold shrinks. The running time of the basic strategy increases because the removal of redundant cameras takes more time. The running time of the greedy strategy shrinks instead since it is able to abort the camera placement earlier. The greedy strategy also has the advantage of finding very few redundant cameras, since the camera placement method aborts as soon as the coverage threshold is met. When observing the total number of cameras placed in a solution we have found the greedy strategy have even better results compared to the basic one when using a coverage threshold below 100%. This is mostly because the greedy strategy tends to end up with small areas without coverage throughout the room polygon as it nears the 100%. In a worst case scenario one additional camera is needed for every small area to reach 100% resulting in very inefficient usage of the effective camera cone. At coverage thresholds below 100% the algorithm aborts before this behavior has an effect on the solution.

**Camera placement limiters coverage:** Both strategies handle the first four problem defining camera placement limiters well and with the ability to add preset cameras it is possible to work around or simulate support for most of the ones not currently supported by the algorithm. The height limiter is difficult to simulate though since we have chosen to model the camera placement problem in 2D.

**Ease of adding support for additional limiters:** It is much easier to add support for additional limiters in the greedy strategy. The camera placement method already has a code structure supporting some sort of ranking of camera positions and it would be easy to extend this part of the method to take more factors than walls and already occupied positions into account. The basic strategy is harder to modify since there is no obvious place to add some kind of ranking of camera positions.

## 5.5 Boundary analysis of camera to vertex count

It is easy to realize that the rectangular algorithm will place more cameras than the 3-Color algorithm per solution since guards can not be responsible for multiple triangles in a solution when using the rectangular algorithm. This means that the upper bound for number of cameras placed in the 3-Color algorithm ( $\lfloor n/3 \rfloor$ ) will likely not apply to the rectangular algorithm.

The upper bound for the rectangular algorithm is a lot more complicated to calculate since it depends on the field of view and effective range of the cameras. To keep the following line of reasoning relatively simple the rectangle splits generated from limited field of view and effective range are ignored:

To determine the number of cameras needed to cover a room, the number of unique rectangles found by the rectangular algorithm needs to be estimated. In the trivial case, a square room, the room is represented by a polygon with four vertices and one unique rectangle is found by the algorithm. If two new vertices are introduced the room forms an 'L' and the algorithm finds two rectangles. It is easily realized that for each two new edges introduced, one extra rectangle is found by the algorithm. In the worst case there are  $n/2 - 1$  rectangles found in a  $n$ -sided polygon.

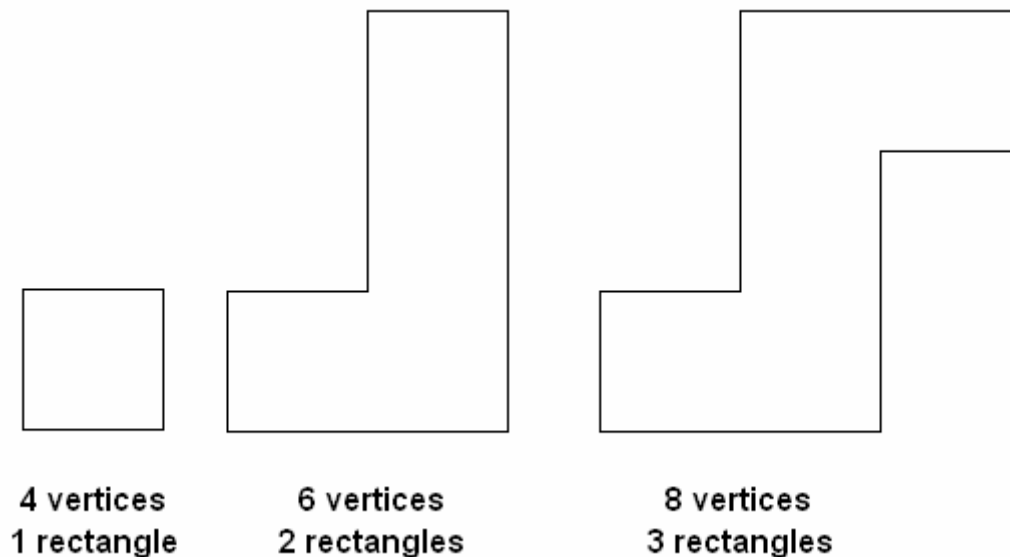


Figure 5.5.1: The number of rectangles found in relation to the number of vertices.

There are two distinct cases when guarding a rectangle with cameras: If the camera has over  $90^\circ$  field of view only one is needed for full coverage; otherwise two cameras are needed. So depending on the cameras field of view either  $n/2 - 1$  or  $n - 2$  cameras are needed in the worst case making this the upper bound of the algorithm. It is possible to estimate the number of cameras needed in orthogonal polygons with holes in a similar way. The result of that study is  $n/2$  and  $n$  number of cameras respectively depending on the camera's field of view, similar to the example without holes.

When considering limited field of view and effective range it becomes impossible to set an upper bound, since it in the absurd case requires an infinite number of splits of rectangles to gain full coverage. In most practical cases a camera model with suitable field of view and effective range is chosen to minimize the number of splits making the  $n - 2$  a good estimate of the upper bound even in this extended line of reasoning.

On a final note the actual number of cameras placed by the rectangular algorithm is very dependant on the shape of the room being monitored. It is often possible to omit cameras because of the overlap between rectangles or because of a coverage threshold of less than 100%. The above line of reasoning is an estimate of the worst case scenario setting the upper bound of the algorithm.

## 5.6 Impact of design choices

Our strategy have been to implement as many features as possible and worry about time complexities later and only if necessary. This has had an impact on the time consumption of the algorithms, with the exception of the 3-Color algorithm. The number of vertices in the room polygon is too small to make any increase in running time noticeable for this algorithm.

The inefficient implementation of the coverage calculation is another matter. The large number of sample points needed for accurate coverage estimations make the time consumption of the rectangular algorithms increase very fast with the area of the room polygon. The coverage calculation algorithm will probably need to be replaced if the prototype software is going to be further developed and later released as a product.

## 5.7 Possible improvements

There are numerous possible improvements both related to how the algorithm is modeled and to the prototype software. We have listed the most important issues divided into two separate lists with issues ranked by importance.

### 5.7.1 Algorithm modeling improvements

- **Investigate the floodlight problem:**  
At the end of this thesis a report was found, discussing how to illuminate a polygon with a minimal amount of  $45^\circ$  floodlights. Since this problem is very similar to our problem, and also seem to solve it in a similar fashion, it is likely to be of use when further developing the rectangular algorithm. The described algorithm works by finding arbitrary quadrilaterals instead of limiting itself to

rectangles. On the other hand the intensity of the floodlight is additive in contrary to camera resolution, which may make the algorithm complicated to use in a camera placement problem. For further information see reference [9].

- **Improve the time complexity of the coverage calculation algorithm:**  
This algorithm can either be improved by dynamically deciding on sampling distances depending on the room polygon area or camera cone area. Alternatively it could be improved by trying to find areas with homogeneous coverage with the help of algebraic set operations.
- **Add an alternative to calculate best possible coverage with limited number of cameras:**  
This is a very common way to place cameras in practice. A customer generally has a budget for a set amount of cameras and wants best possible coverage under this limitation. Supporting this kind of placement would therefore need to be added in a final product.
- **Add support for limiters 5-10 described in section 2.3:**  
Some of these limiters should be relatively easy to implement support for, such as camera facing and lighting, at least for the greedy rectangular algorithm, since only the camera placement strategy algorithm will need to be changed so that it prioritizes possible camera positions differently. Other limiters, such as height, need more work (See the next point).
- **Add support for the third dimension, the height limiter:**  
There are two ways to add support for height. Either the problem can be formulated as a 3D-problem, making our algorithms obsolete, or it can be formulated as a “semi-3D” or 2.5D problem. This means that a plane is defined at a certain height where the objects you want to monitor are located. The camera cones used by the algorithm are then calculated as the intersection of the three dimensional camera cones and this plane. When this is done the algorithms described in this report can be executed as usual with the new camera cones.
- **Try different camera placement strategies:**  
The greedy strategy only guarantees optimization locally, per rectangle. There are other approaches that might produce better result such as dynamic programming or some kind of genetic algorithm. We have not had time to investigate any of these approaches but they would probably be too time inefficient to be useful until the coverage calculation algorithm is improved.
- **Add support for non-orthogonal polygons to the rectangular algorithm:**  
This could probably be achieved by searching for trapezoids instead of rectangles.
- **Add support for more advanced camera models able to zoom and rotate:**  
Camera models such as dome cameras would probably need to be supported in a final product but we decided to omit them from our investigations because of the added complexity they would add to the problem.

### 5.7.2 Software improvements

- **Add a more advanced camera model database:**  
The current database only has a few models and probably does not scale very well if a large number of camera models are added. A more advanced database is probably desired in a final product.

- **Improvement of the input method for polygons:**  
It would be easier to input the room polygon if vertices could be added by clicking in the paint area of the input tab but this would require the implementation of more CAD-like tools. It would also be helpful if a map of the area could be imported and placed in the background, scaled to the paint area, to ease the input process.
- **Add support for known CAD-formats:**  
It would be beneficial if widely used file formats such as *.dwg*, used by for example AutoCAD, could be opened directly in the final software.
- **Add printing and export functionality:**  
A final product would need to have support for printing of the solution in addition to some sort of export to picture function.

## 6 Conclusions

At the beginning of this thesis much effort was put into finding an already existing algorithm that could solve the camera placement problem. The 3-Color algorithm was the most promising one but, as argued earlier in this report, solutions lacked several key characteristics, for example support for a limited field of view and effective camera range.

When this had been concluded we moved on to designing our own algorithm with support for the basic requirements listed in section 3.1. The result was the rectangular algorithm. The idea behind this algorithm is to divide the room into rectangles and then solve the coverage problem on a local scale, per rectangle. In the end sub-solutions are combined and a last search for redundant cameras, because of overlapping rectangles, is done.

The rectangular algorithm produces a solution in  $O(n^4s)$  time<sup>3</sup> which is a lot compared to the 3-Color algorithm time complexity of  $O(n\log(n))$ . The rectangular algorithm is also worse than the 3-Color algorithm when it comes to the upper bound for the number of cameras in the solution:  $n - 2$  compared to  $\lfloor n/3 \rfloor$ . These comparisons do not say a lot though since the rectangular algorithm is able to produce acceptable solutions for the camera placement problem while the 3-Color algorithm is not.

The rectangular algorithm has been tested with two different strategies for placing cameras. The first, most basic, strategy simply places two cameras per rectangle. The other one uses a greedy strategy where the four corners of each rectangle are considered possible camera positions. The camera positions used by the final solution are decided by a greedy iterative method, where the camera position with the greatest coverage increase is chosen in each cycle. The evaluation in chapter 5 shows that the greedy strategy, with a few exceptions, is better than the basic one. It generally produces solutions with fewer and better placed cameras in comparison to the basic strategy and it also has the greatest potential for improvement, since its structure makes it easy to add new features.

Finally we conclude that the prototype software is best used as a complementary tool, not as a replacement, for a camera installer. A camera installer usually works by identifying surveillance tasks as described in section 2.3 and then placing cameras considering the bottom half of the described limiters. These limiters handle the technical and by humans introduced limiters such as lighting and legal aspects. The first four limiters handle the strictly geometrical part of the problem and are much more difficult to estimate which often results in worse coverage than intended. This is where the prototype software is helpful since it excels at the geometrical part of the problem. A possible workflow would be: identify surveillance tasks, place cameras monitoring them and finally let the software fill the room with additional cameras until a desired coverage threshold is met.

---

<sup>3</sup> See Appendix B for more information.

## 7 References

- [1] Y. Amit, J. S. B. Mitchell and E. Packer, “Locating Guards for Visibility Coverage of Polygons”, *Society for Industrial and Applied Mathematics*, 2007
- [2] Code Project, “Art Gallery Problem: polygon triangulation & 3-coloring”, [http://www.codeproject.com/KB/recipes/Art\\_Gallery\\_Problem.aspx](http://www.codeproject.com/KB/recipes/Art_Gallery_Problem.aspx)  
2007-09-12
- [3] H. Ganzález-Banos and J-C. Latombe, “A Randomized Art-Gallery Algorithm for Sensor Placement”, *Stanford University*, 2001
- [4] A.A. Kooshesh and B. M. E. Moret, “Three-coloring the vertices of a triangulated simple polygon”, *Pattern Recognition*, vol 25, p. 443, 1992
- [5] D. Lee and A. Lin, “Computational Complexity of Art Gallery Problems”, *IEEE Trans. Inform. Theory*, 32:276-282, 1986
- [6] J. O’Rourke, “Art Gallery Theorems and Algorithms”, *Oxford University Press*, 1987, ISBN: 0-19-503965-3
- [7] The Swedish Police, The Swedish National Laboratory of Forensic Science, The Swedish Bankers’ Association and The Swedish Federation of Trade and Services, “Camera Surveillance – Test your system before a criminal does”, [http://www.polisen.se/mediaarchive/1688/32329/Riktlinjer\\_\\_\\_eng\\_ver\\_060119.pdf](http://www.polisen.se/mediaarchive/1688/32329/Riktlinjer___eng_ver_060119.pdf)  
2007-09-12
- [8] Swedish police requirements for camera surveillance systems, [http://www.polisen.se/mediaarchive/1688/32329/SKL\\_o\\_Polisens\\_rad\\_\\_\\_eng\\_ver\\_060119.pdf](http://www.polisen.se/mediaarchive/1688/32329/SKL_o_Polisens_rad___eng_ver_060119.pdf)  
2007-09-12
- [9] C. D. Tóth, “Illumination of polygons by 45°-floodlights”, Institute for Theoretical Computer Science, ETH, Zürich, IFW, CH-8092, Switzerland, 2002

## 8 Appendix A: Wordlist

- **Actual camera cone:** The cone calculated only from the camera field of view, not taking a limited range into account.
- **ATM:** Automated teller machine. An electronic machine that allows bank customers to perform simple bank transactions without interacting directly with the bank's staff.
- **Camera position:** The coordinates and the angle of a camera.
- **Coverage threshold:** The percentage of the room polygon that needs to have camera coverage in the solution.
- **Dead zone:** Area that is in the camera's effective range when modeling the problem in 2D, but not covered in reality due to the limitations of the camera's field of view.
- **Effective camera cone:** The cone calculated from the camera field of view, limited by the effective range.
- **Effective camera range:** The resolution of the image sensor and the size of the objects monitored effectively limit the range of the camera.
- **Facial recognition:** An application of cameras where the face of a person is captured and then digitally processed with the intent to identify the person. This application often requires footage with high resolution.
- **Image sensor:** This is the film of a digital camera. The resolution of the image sensor decides how close the real world is sampled into a digital image. The height and width of the image sensor affects the field of view of the camera.
- **Lens:** The lens of the camera focuses the rays of light onto the image sensor. The focal length of the lens affects the field of view of the camera.
- **NP-complete:** The hardest group of problems belonging in the NP complexity class. These problems are only solvable in polynomial time on a non-deterministic Turing machine. Since no such machine exists the only way to find the optimal solution to an NP-complete problem is to try all possible combinations. Each suggested solution can be tested in polynomial time on a normal deterministic Turing machine though.
- **Plan file:** The file containing a room polygon and a set of solutions produced by the prototype software. They have .cip as extension, an acronym for Camera Installation Plan.
- **Polynomial time:** It is possible to express the time complexity of a function as a polynomial.
- **Room polygon:** A polygon shape representing a room structure.
- **Unique rectangle:** A rectangle not enclosed by another rectangle in the polygon. Overlap with other unique rectangles is allowed.

## 9 Appendix B: Walkthrough of algorithms

This appendix will walk through the algorithms used by the rectangular algorithm to solve the camera placement problem also showing the time complexities of the different operations. The algorithms handle a polygon object, representing the room, structured in the following way:

All time complexity calculations are worst case. All lines without written complexity are done in constant time. The following variables are used:

- **n** is the number of vertices of the room polygon.
- **s** is the number of sample points in the coverage algorithm.

### **FindRectangles:** $O(n^3)$

Input: *polygon* //the room polygon

Variable: List - *verticalLines* //used for collision detection

Variable: List - *horizontalLines* //used during extension of lines

Variable: List - *intervals* //used to keep track of active intervals

Variable: List - *rectangles* //contains the found rectangles

- Fill *verticalLines* with all vertical edges from *polygon*.  $O(n)$
- Fill *horizontalLines* with all vertical edges from *polygon*.  $O(n)$
- Sort *verticalLines* and *horizontalLines*.  $O(n \log(n))$
- For each edge in *verticalLines*:  $O(n)$ 
  - For each interval in *intervals*:  $O(n)$ 
    - If – the edge encloses the interval:
      - Create a rectangle with the interval as base, limited by the edge, and add it to *rectangles* if it is unique.  $O(n)$
      - Mark the current interval for deletion from *intervals*.
    - Else if – the interval encloses the edge:
      - Create a rectangle with the interval as base, limited by the edge, and add it to *rectangles* if it is unique.  $O(n)$
      - Create a new interval from the edge's lowest point to the interval's lowest point, with the same x-coordinate as the interval, if possible and mark it added to *intervals*.
      - Create a new interval from the edge's highest point to the interval's highest point, with the same x-coordinate as the interval, if possible and mark it to be added to *intervals*.
      - Remove the interval from *intervals*.
    - Else if – the edge has its lowest point inside and its highest point above the interval:
      - Create a rectangle with the interval as base, limited by the edge, and add it to *rectangles* if it is unique.  $O(n)$

- Create a new interval from the edge's lowest point to the interval's lowest point, with the same x-coordinate as the interval, if possible and mark it to be added to *intervals*.
  - Remove the interval from *intervals*.
- Else if – the edge has its highest point inside and its lowest point below the interval:
  - Create a rectangle with the interval as base, limited by the edge and add it to *rectangles* if it is unique.  $O(n)$
  - Create a new interval from the edge's highest point to the interval's highest point, with the same x-coordinate as the interval, if possible and mark it to be added to *intervals*.
  - Mark the current interval for deletion from *intervals*.
- If - the edge did not cross any interval:
  - Extend the edge upwards and downwards until it intersects a line in both directions in *horizontalLines*.  $O(n)$
  - Mark the extended line to be added to *intervals*.
- Remove all intervals from *intervals* that are marked for deletion and add all intervals marked to be added to *intervals*.
- Return *rectangles*.

**CalcCoverage:**  $O(n^2s)$

Input: *polygon* //the room polygon  
 Input: List - *cameras* //list of cameras representing the solution

Const: *nbrSamples* //number of samples, usually very large  
 Variable: *totalSamplesInPolygon* //tracks how many samples was inside the polygon  
 Variable: *visiblePoints* //tracks how many samples inside the polygon was seen by cameras

- Set *totalSamplesInPolygon* = 0; *visiblePoints* = 0.
- Spread samples evenly across an area bounded by the min and max x- and y-values of *polygon*.
- For each sample point:  $O(s)$ 
  - If – the sample point is not in *polygon*:
    - Continue to next sample point.  $O(n)$
  - *totalSamplesInPolygon*++.
  - For each camera:  $O(n)$ 
    - If – the sample point is in the current camera's cone and is in line of sight  $O(n)$  :
      - *visiblePoints*++
      - Break the for each camera loop.
- Return *visiblePoints* divided by *totalSamplesInPolygon*.

**RemoveRedundantCameras:**  $O(n^4s)$

Input: *polygon* //the room polygon

Input: List – *cameras* //list of cameras representing the solution  
 Input: *coverageThreshold* //the required coverage  
 Variable: List - *coverage* //keeps track of the coverage contribution of each  
 //camera

- While true:  $O(n)$ 
  - For each camera:  $O(n)$ 
    - Calculate coverage without the current camera.  $O(n^2s)$
    - Add current camera + the calculated coverage to *coverage*.
  - Sort *coverage* so that first element is the camera with least impact on coverage.  $O(n \log(n))$
  - Remove the camera in *coverage* with least influence on coverage from *cameras* if the resulting coverage is above *coverageThreshold* or if the coverage remains the same without it (prioritize removal of cameras inside the room over wall cameras).  $O(n^2)$
  - If – there was no camera to remove:
    - Break the while loop.
  - Remove all elements from *coverage*.
- Return *cameras*.

**BasicRectangle:**  $O(n^4s)$

Input: *polygon* //the room polygon  
 Input: *coverageThreshold* //the required coverage

Variable: List - *cameras* //the list of cameras representing the solution

- Find rectangles.  $O(n^3)$
- For each rectangle found:  $O(n)$ 
  - Add a camera placed in the top right corner to *cameras*.
  - Add a camera placed in the bottom right corner to *cameras*.
- Remove redundant cameras.  $O(n^4s)$
- Return *cameras*.

**GreedyRectangle:**  $O(n^4s)$

Input: *polygon* //the room polygon  
 Input: *coverageThreshold* //the required coverage

Variable: List - *cameras* //the list of cameras representing the solution  
 Variable: List - *possibleCameraPositions* //the list with possible camera positions  
 Variable: List - *badCameraPositions* //the list containing cameras that should be  
 //avoided

- Find rectangles.  $O(n^3)$

- For each rectangle:  $O(n)$ 
  - Add top left corner to *possibleCameraPostions*.
  - Add top right corner to *possibleCameraPostions*.
  - Add bottom left corner to *possibleCameraPostions*.
  - Add bottom right corner to *possibleCameraPostions*.
- While true:  $O(n)$ 
  - For each camera in *possibleCameraPositions*:  $O(n)$ 
    - Calculate coverage with the current camera + all cameras in *cameras*.  $O(n^2s)$
    - Add the current camera + the calculated coverage to *coverage*.
  - Sort *coverage* so that the camera with most added contribution is first in the list.  $O(n\log(n))$
  - Add the best camera to *cameras* if the it increases the coverage (internal cameras need to be twice as good as wall cameras to be chosen; only add a camera if it improves the result, otherwise all cameras would be added if an impossible *coverageThreshold* was set).  $O(n^2)$
  - Move all cameras with the same coordinates as the best camera from the *possibleCameraPositions* list to the *badCameraPositions* list (try to avoid placing multiple cameras in the same place as long as possible).  $O(n)$
  - Remove all elements from *coverage*.
  - If – *coverageThreshold* is met or if no camera was added:
    - Break the while loop.
- Remove redundant cameras.  $O(n^4s)$
- Return *cameras*.

## 10 Appendix C: Program commands

Millimeters are used for all intervals and coordinates. Commands are written on the “enter command” line, brackets excluded.

- **add [x,y]:** Adds a vertex to the polygon, if the starting point of the room polygon is defined, creating a line from the previously added vertex to this one.
- **addCam:** Opens the add new camera window, allowing the user to predefine cameras before running an algorithm.
- **close:** Adds a line between current point and the start point, closing the polygon.
- **cones on | off:** Toggles the on/off status of the camera cones. Default is off.
- **end:** Adds a line between current point and the start point, closing the polygon.
- **grid on | off | [spacing]:** Toggles the on/off status of the gridding or changes the spacing of the grid. Default is off and 1000mm.
- **start [x,y]:** Defines the starting point of the room polygon to be at coordinate x,y.