T.R.

GEBZE TECHNICAL UNIVERSITY

GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

EFFICIENT CAMERA PLANNING FOR SURVEILLANCE APPLICATIONS

MEHMET ARİF ŞEKERCİOĞLU

A THESIS SUBMITTED FOR THE DEGREE OF

MASTER OF SCIENCE

DEPARTMENT OF COMPUTER ENGINEERING

GEBZE

2016

T.R.

GEBZE TECHNICAL UNIVERSITY

GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

EFFICIENT CAMERA PLANNING FOR SURVEILLANCE APPLICATIONS

MEHMET ARİF ŞEKERCİOĞLU

A THESIS SUBMITTED FOR THE DEGREE OF

MASTER OF SCIENCE

DEPARTMENT OF COMPUTER ENGINEERING

THESIS SUPERVISOR

ASSIST. PROF. DR. YAKUP GENÇ

GEBZE

2016

T.C.
GEBZE TEKNİK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

GÜVENLİK KAMERALARININ YERLERİNİN ETKİN PLANLANMASI

MEHMET ARİF ŞEKERCİOĞLU

YÜKSEK LİSANS TEZİ
BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

DANIŞMANI
YRD. DOÇ. DR. YAKUP GENÇ

GEBZE
2016

GEBZE TEKNİK ÜNİVERSİTESİ	**YÜKSEK LİSANS JÜRİ ONAY FORMU**

GTÜ Fen Bilimleri Enstitüsü Yönetim Kurulu'nun ...27.../...06.../2016 tarih ve ...2016/...43..... sayılı kararıyla oluşturulan jüri tarafından 26/8/2016 tarihinde tez savunma sınavı yapılan Mehmet Arif Şekercioğlu'nun tez çalışması Bilgisayar Mühendisliği Anabilim Dalında YÜKSEK LİSANS tezi olarak kabul edilmiştir.

JÜRİ

ÜYE (TEZ DANIŞMANI)	:Yrd. Doç. Dr. Yakup Genç
ÜYE	:Doç. Dr. Fatih Erdoğan Sevilgen
ÜYE	:Yrd. Doç. Dr. Tarkan Aydın

ONAY

Gebze Teknik Üniversitesi Fen Bilimleri Enstitüsü Yönetim Kurulu'nun ...../...../2016 tarih ve ...../..... sayılı kararı.

İMZA/MÜHÜR

SUMMARY

Any multi-camera system requires placement decisions for its cameras to achieve the best system performance. Automated planning algorithms attempt to find the best placement and orientation of the cameras in such systems. This thesis proposes an optimization-based approach maximizing the combined visible volume in the scene. Calculating the visible volume is a computationally intensive process. GPU's are used to make this calculation practical. Some of the well-known optimization methods (stochastic gradient descent, particle swarm optimization, artificial bee colony algorithm and their variations) were implemented and tested on real world scenarios. Experimental results show that the proposed approach can be used in practical applications.

Key Words: Camera Planning; Camera Placement; Particle Swarm Optimization; Artificial Bee Colony Optimization; Gradient Descent.

ÖZET

Çok sayıda kamera içeren her sistem kamera yerlerinin doğru bir şekilde belirlenmesine ihtiyaç duyar. Otomatik kamera yeri planlama algoritmaları kameraların yerlerini ve konumlarını doğru şekilde tespit etmeye çalışır. Bu çalışmada kamera planlaması optimizasyon tabanlı bir yaklaşımla ele alınarak sahnedeki toplam görünür hacim maksimize edilmiştir. Görünür hacmin hesaplanması oldukça maliyetli bir işlemdir. Bu işlemi hızlandırmak ve kullanılabilir hale getirmek için algoritmalar önemli bir oranda GPU üzerinde koşturulmuştur. Sıkça kullanılan optimizasyon metodlarının (gradient descent, parçacık sürü optimizasyonu, yapay arı kolonisi algoritması ve bunların varyasyonları) bazıları bu problemi çözmek için gerçeklenmiş ve gerçekçi senaryolarla test edilmiştir. Deneysel sonuçlar bu metodun pratik olarak kullanılabileceğini göstermektedir.

Anahtar Kelimeler: Kamera Planlaması; Parçacık Sürü Optimizasyonu; Yapay Arı Kolonisi Algoritması; Gradyan İniş Algoritması.

ACKNOWLEDGEMENTS

I want to thank my friends Abdullah Akay, Nurmuhammed Çimen and Mustafa Tunalı and many others for their help and friendship during my study. I also like to thank my family for their support and love.

TABLE of CONTENTS

LIST of ABBREVIATIONS and ACRONYMS

Abbreviations and Acronyms		**Explanations**
2D	:	2 dimensional
3D	:	3 dimensional
ABC	:	Artificial Bee Colony Algorithm
CPU	:	Central processing unit
FOV	:	Field of view
GD	:	Gradient Descent
GPU	:	Graphics processing unit
GTU	:	Gebze Technical University
ms	:	millisecond
PSO	:	Particle Swarm Optimization Algorithm

LIST of FIGURES

LIST of TABLES

1. INTRODUCTION

Camera planning is important for any multi-camera system as the placement of cameras can be a major cause of bad performance. Traditional approaches require an expert to make the placement decisions. As the multi-camera systems become more widespread finding experts capable of planning with multiple constraints might be difficult.

Automating the planning process can be beneficial for better system performance. In very complex systems better results can be obtained when compared to the traditional planning methods. Or when some extra constraints are needed, automating the camera planning process can yield better results than the traditional planning methods.

1.1. Camera Planning

This thesis addresses the problem of finding the optimal placement of cameras in a three dimensional (3D) volume with a complex shape and in which there are obstacles. Given n cameras with limited angles of views, the system tries to place those n cameras optimally so that the coverage of the cameras is maximum in terms of volume.

Camera planning can be posed as an optimization problem and its solution can be used for camera placement. Given the characteristics of the three dimensional (3D) workspace and the parameters of the cameras, the visible volume can be maximized. This thesis presents an optimization-based method camera planning method that is fast and practical. We first define a cost function defining the goodness of the placement of the cameras where 1 represents total visibility of the 3D volume we want to place the cameras, and 0 means no visibility. This function takes the 3D scene, a list of obstacles and a list of cameras as input. Each camera has 5 parameters (three for the location and two for the pan and tilt angles of the camera). The cost function optimized by using well-known optimization algorithms. Gradient descent, particle swarm optimization and artificial bee colony optimization algorithms and some variations implemented and tested. The tests have shown that the cost function can be optimized.

We have calculated the visibility of the volume by dividing the workspace into cubic voxels and by counting the visible voxels. After finding the number of visible volumes we divided that number to the number of total voxels. Which we get is a number between 0 and 1 which shows the visibility ratio of the overall 3D volume. This is an approximate solution but when the number of voxels increase the result converges to exact solution.

CUDA implementation of this calculation made it possible calculate the cost function for real world scenarios much faster than the CPU based implementation. Extensive experimental validation has shown that the proposed method is efficient and yields good results.

2. BACKGROUND

For a multi-camera system, camera planning places the cameras in a best configuration based on a metric. Metric is the maximum coverage. Camera planning was studied before both theoretically and practically in computational geometry and sensor planning.

There is a family of problems called art gallery problems (AGP). Studies about AGP forms the theoretical part of the studies about camera planning. The simplest version of this problem tries to find the minimum number of guards sufficient to see every point of the interior of an n-vertex simple polygon. In art gallery problem the goal is to place guards that is capable of seeing everywhere around them whereas in camera placement problem the goal is to place cameras that has a limited angle of view. So camera placement problem can be considered as a specific version of the art-gallery problem.

The method we use is a costly method. We figured out that we can use GPU at our system to make it parallel and faster. Many algorithms have been implemented on GPU's in the recent years and these worked several times faster than the CPU based versions [1]. Hence, implemented the cost function in GPU in order to make the system run faster.

2.1. Art Gallery Problem

In 1973 a mathematician called Victor Klee asked the following question [2], [3]: What is the minimum number of guards who together can watch an entire art-gallery?

Starting with the previous question a family of problems and theorems were proposed. Those family of problems are called art-gallery problems [4].

Camera placement problem can be regarded as a special version of the art-gallery problem. Because in art-gallery problem guards assumed to see everywhere around themselves but in our case surveillance cameras have a limited angle of view.
In 1975 Vaclav Chvatal proposed a theorem for the art-gallery problem called the art-gallery theorem. That theorem gives an upper bound for the minimum number of guards [5]. $\lfloor \frac{n}{3} \rfloor$ guards are sufficient and sometimes necessary for watching an entire art-gallery which is a 2D polygon with n vertices.

In 1975 Steve Fisk gave a shorter and easier proof of the Chvatal's art-gallery theorem [6]:

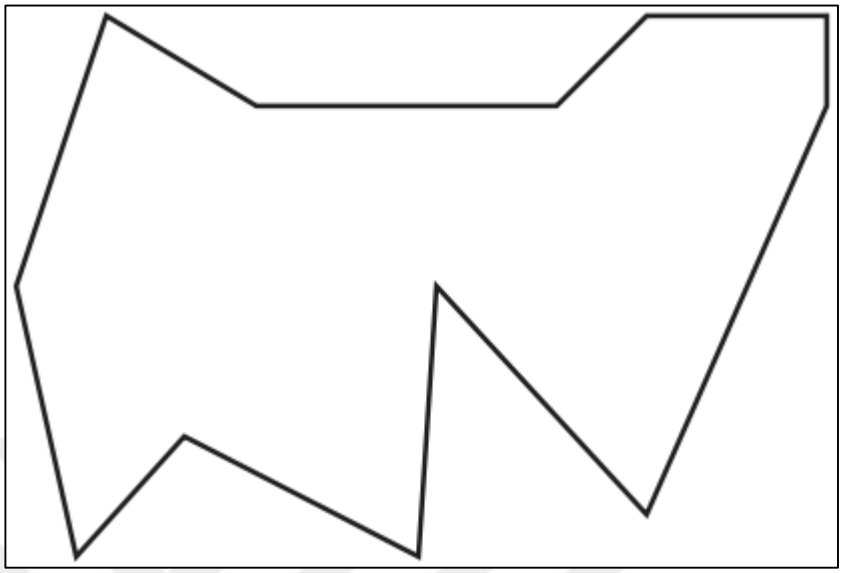

Figure 2.1: 2D polygon for the proof of 2D art-gallery problem.

- Step 1: Divide the polygon (art-gallery) into triangles by adding non-crossing diagonals. The main point of triangulation is that every triangle will be observed by a guard [7].

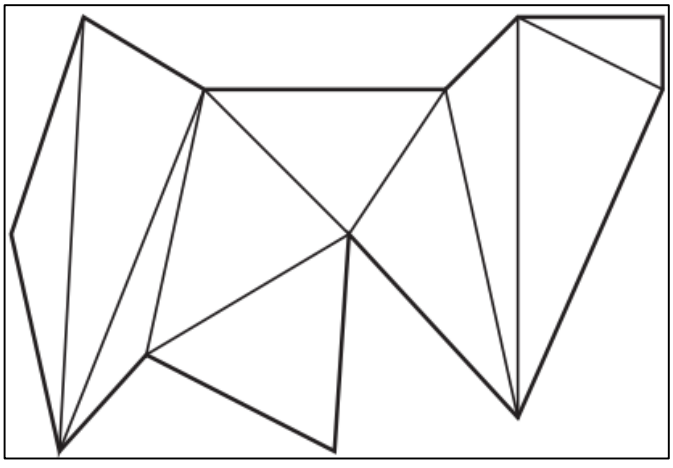

Figure 2.2: Triangulation of the polygon given in Figure 2.1 into rectangles.

- Step 2: Start with any of the triangles and label a, b and c to its vertices.

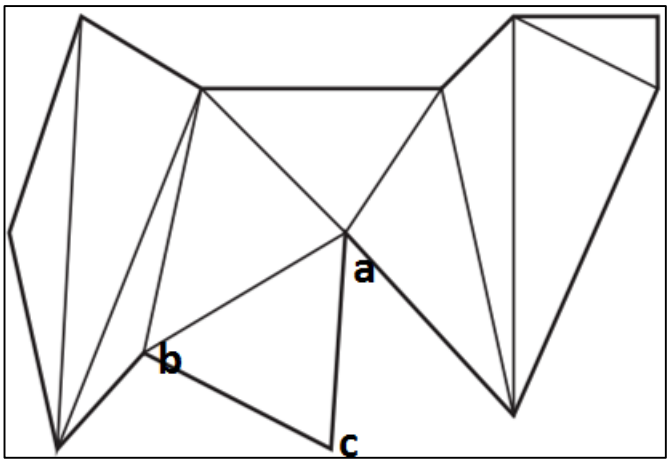

Figure 2.3: Starting with a triangle and labeling its vertices.

- Step 3: Continue labeling a, b or c to the vertices of other triangles such that each triangle has a, b and c type (or color) of vertices.

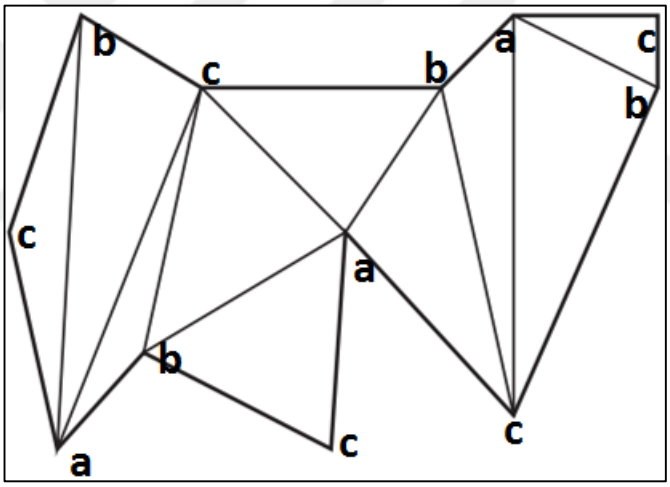

Figure 2.4: Complete the labeling of vertices according to Figure 2.3.

- Step 4: Let number of a's is n_a, number of b's is n_b and number of c's is n_c at the labeled triangle. Let $n_a \leq n_b \leq n_c$ (always there will be an ordering). Put the guards to the vertices of type a, that is to the one that has the smallest number. Then n_a guards are sufficient to protect the gallery. Because each triangle has a vertex of type a. So we can say that $n_a \leq \lfloor \frac{n}{3} \rfloor$. This completes the proof.

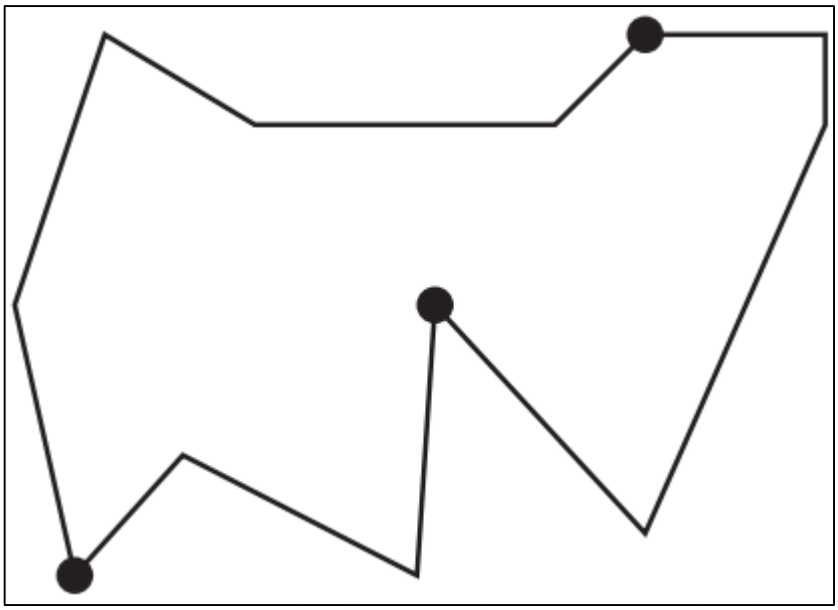

Figure 2.5: Positions of the guards for observing the entire space.

In 1983 it was proved that [8] if the shape of the 2D polygon is orthogonal at most $\lfloor \frac{n}{4} \rfloor$ guards are needed to observe each point.

2.1.1. Art-Gallery Problem in Three Dimensions

For the 2D version of the problem, it was proved that at most $\lfloor \frac{n}{3} \rfloor$ guards would always be more than enough for watching the entire gallery. However, for a 3D art-gallery problem which is represented as a polyhedron, putting a guard at each vertex will not guarantee that each point of the gallery is observed. An example for this is the octoplex [3]. Octoplex is constructed as follows. Start with a 20×20×20 cube. Remove 12×6×20 rectangular prisms from front and back faces. Remove 20×6×6 rectangular prisms from top and bottom faces. Remove 3×20×6 prisms from left and right faces. What is left is the octoplex (see Figure 2.6).

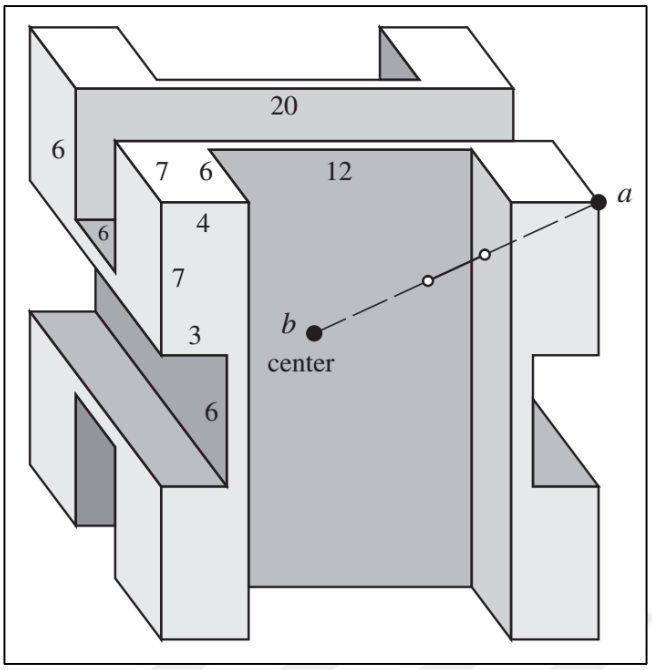

Figure 2.6: At Octoplex point b cannot be observed by any of the guards even if there is a guard at each vertex.

Octoplex [3] has 56 corners and 30 walls. Some points in the octoplex cannot be observed even if a guard is placed at each vertex. An example of such a point is point *b* at Figure 2.6. Point *b*, the center of octoplex, is invisible from all vertices. So there is no such solution for the 3D version of the problem related to the number of vertices like the 2D version of the problem.

There is a study that proposed a method for finding an upper bound on the number of guards for orthogonal polyhedra [9]. The method divides the orthogonal polyhedron into rectangular prisms. Each corner of a prism can be a candidate place for a guard. Next the number of candidate corners are minimized so that entire volume is observed by minimum number of guards.

Note that although art-gallery problem has many similarities with the camera placement problem, techniques for solving art-gallery problem is not enough to solve camera placement problem. Actually camera placement problem can be seen as a special version of the art-gallery problem. Because at art-gallery problem guards assumed to see everywhere around them whereas at camera placement problem guards have a limited angle of view.

2.2. 2D Version of the Camera Placement Problem

There is a study that deals with the 2D version of the camera placement problem on an orthogonal polygon [10]. The paper proposes a method called rectangular algorithm. This method divides the 2D region into a minimum number of rectangles each of which in turn is divided into two triangles. Corners of these rectangles obtained at Figure 2.7 will be possible camera locations.

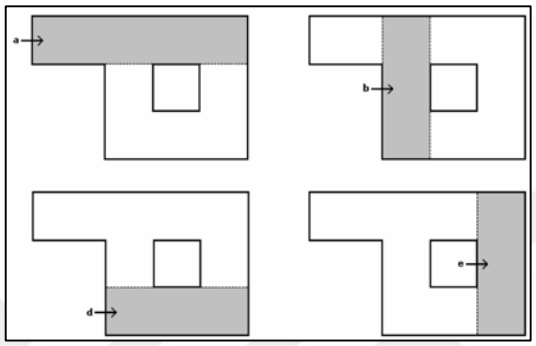

Figure 2.7: Dividing the region into minimum number of rectangular regions according to Pålsson's algorithm.

Figure 2.8 below shows the next step after obtaining the rectangles. Each rectangle will be divided into triangles and each triangle will be observed by a camera.

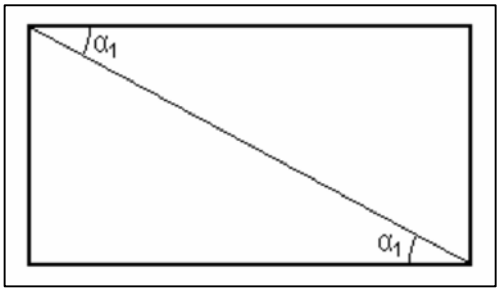

Figure 2.8: Corners of each rectangle are possible camera locations. If the diagonal length is greater than the effective range of the camera, a further division is required. Effective range is the distance that the camera can observe. It is a constraint of the algorithm.

Figure 2.9 shows the division of a rectangle into 2 so that the new diagonals of the new rectangles are within the effective range of the camera. If the diagonal angle

α_1 of the triangle is greater than the field of view (FOV) of the camera, a further division is required.

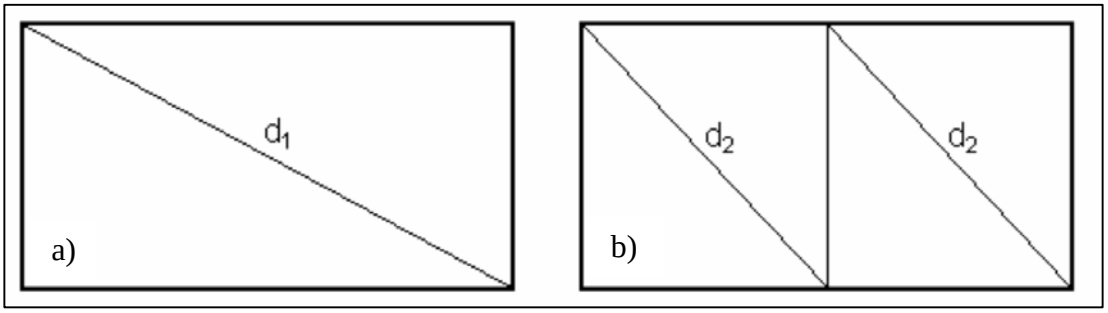

Figure 2.9 Further division may be required for effective camera range a) At the left rectangle diagonal d_1 is greater than the effective range of the camera, b) At the right rectangle, after the division, diagonal d_2 is within the effective range of the camera.

Figure 2.10 below shows the division of a rectangle into so that the diagonal angles α_2 of the new rectangles is less than the FOV of the camera.

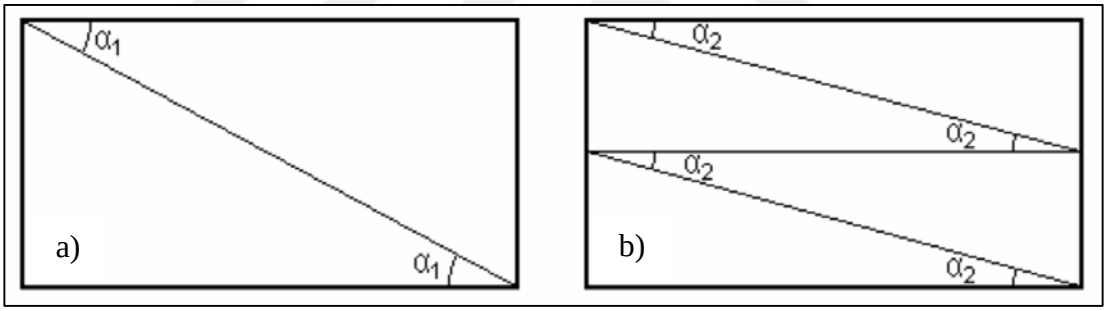

Figure 2.10: Further division may be required for convenient camera angle a) At the left rectangle diagonal angle α_1 is greater than the FOV of the camera, b) At the right rectangle after the division α_2 is less than the FOV of the camera.

After dividing the region into rectangles a greedy strategy is used. At each step best rectangle corner for improving the coverage is found and added to the system. That is at each step a camera is placed at a corner of a rectangle such that coverage increase is maximum.

At the last part of the algorithm redundant cameras are removed by starting to check with the camera that has the least coverage.

Note that this study deals with the 2D version of the camera placement problem where we deal with the 3D case. Triangulation or similar methods does not work for the 3D case. Because at when a rectangular prism is divided into 2 triangular prisms

we cannot observe those triangular prisms with one camera as in the 2D case. We need some other methods to observe them.

2.3. Sensor Planning

Camera placement problem is also studied as a branch of sensor planning. Studies about sensor planning forms the practical part of the studies about camera planning.

There is a study that divides the 2D space into grids and uses linear programming to optimize the cost function. Their 2D spaces are simple rectangular regions [11]. Their solution tries to ensure that each grid is observed by a camera and the minimum number of cameras are used. They deal with simple rectangular spaces.

There is a study that deals with the 2D camera placement problem for achieving maximum coverage of the 2D space [12]. They use greedy search, dual sampling and randomized approaches for optimizing their cost function. Study of [12] is a simplified and 2D version of our study.

There is a study that deals with 3D camera placement problem. But their cost function does not try to find the maximum coverage but they place 3D cylindrical objects randomly and next try to place the cameras so that maximum coverage for the cylindrical objects is found [13]. Their cost function is an exact cost function that returns the exact visibility of the 3D space. They put some 3D objects to the 3D space and their cost functions returns the visibility of those randomly put 3D objects. They use simulated annealing for optimizing the cost function. Their system works for low dimensional spaces and they tested their system with 2 and 3 cameras.

There is a study that solves the 2D version of the problem as an optimization problem. They divide the 2D area into grids [14]. Next they create camera locations and orientations. They then chose n cameras from the previously generated ones using a branch and bound strategy. This is a semi-automatic method for camera planning. In our study the system finds the locations and orientations automatically. Also this method solves the 2D version of the problem whereas we deal with the 3D case.

There is a study that divides the 2D workspace into grids and uses binary integer programming for optimizing the cost function [15]. Their workspaces are 2D and much simpler than the ones we used.

There is a study that deals with the 3D camera placement problem. Their cost function is a probabilistic function that is does not find the coverage of the space exactly [16]. They use binary integer programming to optimize the cost probabilistic cost function.

There is a study that deals with the 2D camera placement problem. They use particle swarm (PSO) optimization to optimize their cost function [17]. Their cost function gives more importance to some critical areas. We also used PSO as one of our optimization algorithms. However, our space is 3D.

In summary there are many theoretical work about the art-gallery problem. These do not solve the camera placement problem fully. There are also studies about sensor planning. These are more practical solutions when compared with the studies on AGP. These studies mostly deal with the 2D camera planning problem. There are also some sensor planning studies about the 3D camera planning. Their cost functions are mostly probabilistic cost functions whereas our cost function is exact.

3. METHOD

A cost function which determines the visibility of a given camera configuration for a given 3D space with obstacles is defined. Total visible volume that is seen by the cameras is $\bigcup_{n}^{i=1} V_{c_i}$. Here V_{c_i} is the volume that is observed by camera i. This cost function will be optimized so that the union of the volumes seen by the cameras is visible. Which can be expressed mathematically with the equation below:

$$maximize\ f(\theta_1, \theta_2, \dots, \theta_n) = \bigcup_{n}^{i=1} V_{c_i}(\theta_i) \tag{3.1}$$

Here θ_i is the parameters of camera i.

When computing the visible volume first the volume is divided into small cubes called voxels. A voxel is visible if it is in the field of view of any of the cameras and if there is no obstacle between the camera and voxel (see Figure 3.1).

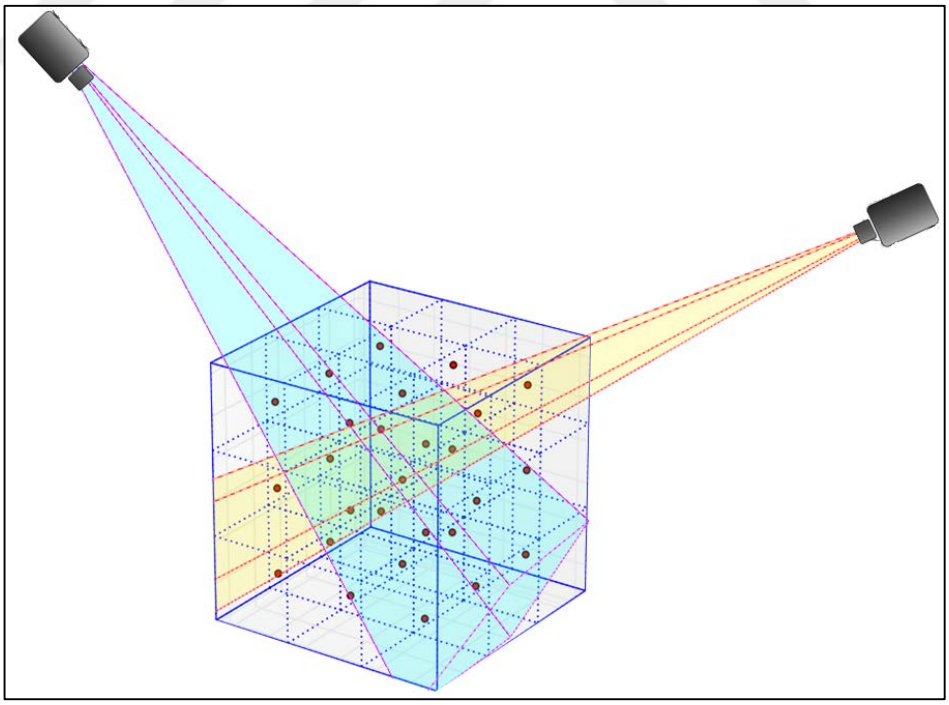

Figure 3.1: A voxel is visible if its center is in FOV of a camera and if there is no obstacle between that camera and the voxel.

Obstacles are defined as triangular fields. In order to define an obstacle in the system 3 points were defined. A wall for example is made up of two triangular obstacles. Pseudo code at Table 3.1 summarizes the cost function.

Table 3.1: Calculation of cost function based on the total visible volume.

```
Function isVisible(voxel, cameraList, obstacleList):
    for camera in cameraList :
        if voxel is in field of view of camera:
            occlusion=False
            for obstacle in obstacleList:
                if obstacle is between camera and voxel:
                    occlusion=True
                    break
            if not occlusion:
                return True
    return False


Function costFunction(voxelList, cameraList, obstacleList):
    numberOfVisibleVoxels=0
    for voxel in voxelList :
        if isVisible(voxel, cameraList, obstacleList):
            numberOfVisibleVoxels++
    return numberOfVisibleVoxels/length(voxelList)
```

Cost function returns a number between 0 and 1. 0 means no voxel is visible and 1 means all voxels are visible by at least one camera. Cost function takes the list of voxels, list of obstacles and list of cameras as parameter. Each camera has 5 parameters which are x, y, z locations of the camera center and spherical angles of the camera orientation. Spherical angles are horizontal (θ) and vertical (φ) [18] (see Figure 3.2).

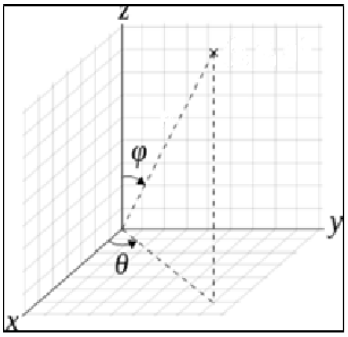

Figure 3.2: Spherical coordinate system used to formulate the optimization problem.

3.1. Details of the Computation

When computing the visibility of a voxel from a camera point, firstly the axes are rotated and translated so that the camera point becomes the origin of the axes and the direction of the camera becomes the new x axis. The rotation and translation process is as follows [19]:

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 3.3: Rotation matrices per each axis.

- Translate the voxel point so that camera point becomes the new origin. That is $Pvoxel = Pvoxel - Pcamera$.
- Rotate the voxel point around the z axis. Rotation angle is the horizontal angle of the camera. That is $Pvoxel = R_z * Pvoxel$.
- Rotate the voxel point around the y axis. Rotation angle is the vertical angle of the camera. That is $Pvoxel = R_y * Pvoxel$.

After rotation and translation, it is needed to be checked if the point is in the field of view of the camera. While computing the visibility of voxels horizontal and vertical field of view (FOV) angles assumed to be constant for all the cameras. We used 60˚ and 45˚ for horizontal and vertical FOV angles respectively in most of our experiments.

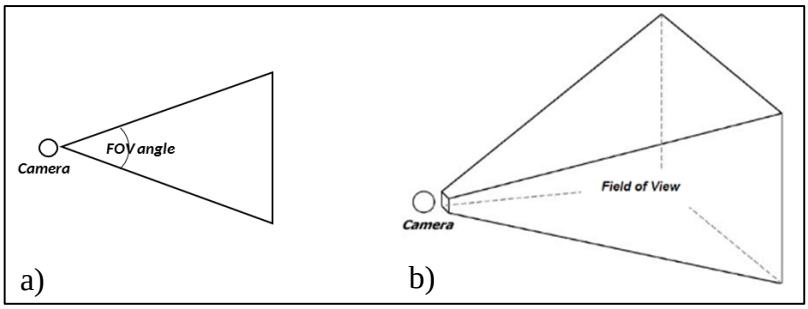

Figure 3.4: Field of view of a camera a) 2D version, b) 3D version.

In order to decide if a point is in FOV or not we will check the horizontal and vertical angles between the voxel and camera. After translation and rotation if voxel x is less than 0 it means the voxel is at the back of the camera which means voxel is not in the FOV of the camera. Next we computed the horizontal and the vertical angles between the voxel and camera. If the angles are less than the half of horizontal and vertical FOV angles-that is 30 and 22.5 respectively - the point is in the FOV of the camera else, it is not in the FOV. The pseudo code of the above process is seen in Table 3.2.

Table 3.2: The code for checking if a voxel is in the camera field of view.

```
Function isInFOV(voxel,camera):
      voxel=voxel-camera  # translation
      voxel= Ry*(Rz*voxel) #rotation
      if voxel.x < 0:
            return False
      x=voxel.y/voxel.x
      y=voxel.z/voxel.x
      if abs(x)<=tan30 and abs(y)<= tan22.5 :
     #30 and 22.5 are half of the FOV angles
            return True
      return False
```

Now the process of finding the occlusions will be explained. We have an obstacle which consists of 3 points V_1, V_2 and V_3 and a line segment representing the camera visibility which starts with camera point P_0 and ends at voxel P_1. We want to know if $[P_0, P_1]$ intersects with the triangle V_1, V_2, V_3. Let's start with determining if there is an intersection between the triangle and line segment. Let n be the normal vector of the plane determined by the triangle V_1, V_2, V_3. Let

$$r_1 = \frac{n \cdot (V_0 - P_0)}{n \cdot (P_1 - P_0)} \tag{3.2}$$

. If $r_1 \geq 0$ that means line segment and the plane determined by the triangle intersects somewhere. Else they don't intersect and we are done. So now let's deal with the case that there is an intersection between the plane determined by the triangle and the line segment. The equation of the plane is given by equation (3.3).

$$V(s,t) = V_0 + s(V_1 - V_0) + t(V_2 - V_0) = V_0 + su + tv \tag{3.3}$$

If $s + t \leq 1$ that means intersection point is in the triangle else intersection point is outside of the triangle. s and t are given by the equations (3.4) and (3.5).

$$\mathrm{s} = \frac{(\mathrm{u} \cdot \mathrm{v})(\mathrm{w} \cdot \mathrm{v}) - (v \cdot v)(w \cdot u)}{(\mathrm{u} \cdot \mathrm{v})^2 - (u \cdot u)(v \cdot v)} \tag{3.4}$$

$$\mathrm{t} = \frac{(\mathrm{u} \cdot \mathrm{v})(\mathrm{w} \cdot \mathrm{u}) - (u \cdot u)(w \cdot v)}{(\mathrm{u} \cdot \mathrm{v})^2 - (u \cdot u)(v \cdot v)} \tag{3.5}$$

So if $s + t \leq 1$ that means there is an occlusion and the voxel is not visible by that camera [20].

3.1.1. Parallelization via CUDA

Counting the number of visible voxels is a process that can be fully parallelized. We used CUDA for making the calculations faster. Visibility of each voxel was determined by a CUDA thread. CUDA usage greatly improved the performance of the system as will be seen later. CUDA implementation worked nearly 20 times faster than the CPU implementation.

3.2. Algorithms Used for Optimizing the Cost Function

We use Artificial Bee Colony Algorithm (ABC), Particle Swarm Optimization Algorithm (PSO), Gradient Descent (GD) and some variations of those algorithms to optimize the cost function. We compared those algorithms in terms of accuracy of the results and time in the experiments section.

Our first attempt to optimize the cost function is using the Gradient Descent Algorithm. We placed the cameras randomly and gave them random initial angles. Next we applied Gradient Descent [21], [22] for reaching to local optima.

ABC and PSO require a number of initial solutions. At each iteration some of the solutions are improved and at the end best solution among the set of solutions is the result of the algorithm. GD requires one solution to work on but for more comparable results we created a set of initial solutions and chose the best one among them and apply GD to that solution.

Optimization algorithms were run in two ways: First a number of steps and the second is a fix time period like 300 seconds. We chose the more convenient one for each different test. At some tests we use time as the stop criteria and at some other we use number of iterations as the stop criteria.

3.2.1. Gradient Descent

Gradient Descent also known as steepest descent is a first-order optimization algorithm. It is called a first-order algorithm because the first derivative of the function is used. GD takes steps proportional to the negative direction of the gradient and tries to reach the local minima.

Gradient Descent defines the step from point a to b as follows:

$$b = a - \gamma \nabla F(a) \tag{3.6}$$

Where γ is a positive number and $\nabla F(a)$ is the gradient of the function F at point a. γ is allowed to change at every step. GD algorithm stops when a maximum CPU time exceeded or when a number of steps exceeded or when the improvement between two steps is less than a number.

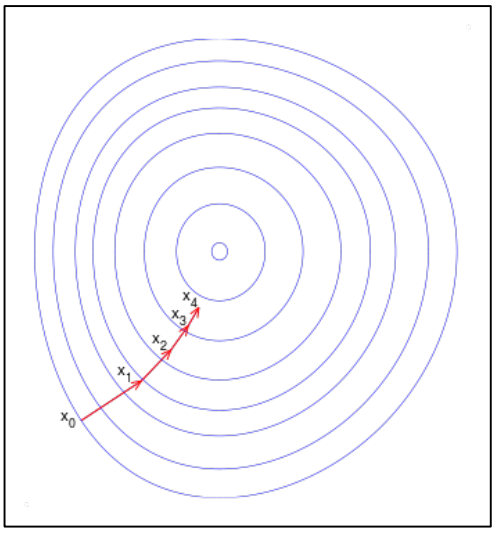

Figure 3.5: Gradient Descent steps are shown. At each step solution gets closer to the minimum in the center.

Note that in order to apply Gradient Descent we need F to be defined at a and again it needs to be differentiable in a neighborhood of a. In order for a function be differentiable at a point a it must be continuous at that point. In our case the function which we will apply GD is our cost function which returns the visibility of the workspace. Our cost function is not a mathematical function. It does not have a mathematical expression. Therefore, we used discrete differentiation technique [23]. That is for differentiating the F with respect to x we used equation () below.

$$\frac{\partial}{\partial \mathrm{x}} \mathrm{F}(\mathrm{x},\mathrm{y},\mathrm{z}) = \frac{\mathrm{F}(\mathrm{x}+\mathrm{h},\mathrm{y},\mathrm{z}) - \mathrm{F}(\mathrm{x},\mathrm{y},\mathrm{z})}{\mathrm{h}} \tag{3.7}$$

Where h is a small real number. We chose different h values for parameters related to angles and distances, like $\pi/50$ and 0.2 (Angles are expressed in terms of radian and distances are expressed in terms of meters.) respectively.

The cost function we use is not continuous at every point. Because there are obstacles in the system. Figure 3.6 shows an example of the discontinuity of the cost function. At the left part there is a camera at a side of a room which can observe most of the room. At the right part of the figure, the camera location changed slightly and the camera sees nothing.

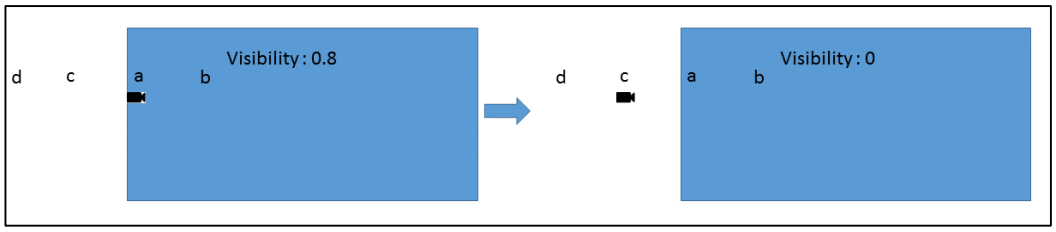

Figure 3.6: Discontinuity of cost function. When moving from point *a* to point *c* visibility suddenly becomes zero.

Figure 3.7 below shows the change of the value of the cost function in a room similar to the one in Figure 3.6. As the location of the camera changes in one direction, first the value increases constantly next it decreases to zero suddenly. In order to draw this function, we changed the x direction of the camera 0.001 at each iteration. As can be seen from the graph cost function is not continuous.

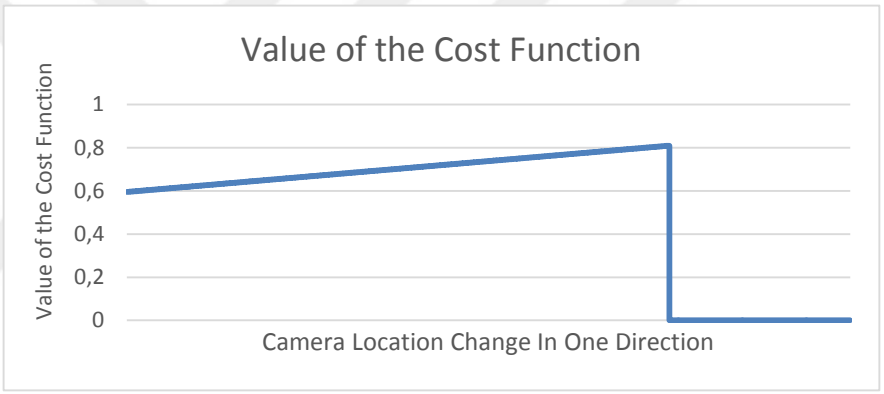

Figure 3.7: Shows value of the cost function as the location of the camera changes in one direction like in Figure 3.6 from point *b* to point *c*.

Because the cost function is discontinuous, it is possible to step to a point where the visibility of the cost function decreases. For example, at Figure 3.6 suppose camera started with the position and GD computed $\frac{F(a)-F(b)}{|a-b|}$ and saw that the value of the cost function decreases as it goes through the point b. So it steps at the opposite direction and came to the point c, where $F(c) = 0$. At the next step it will check $\frac{F(c)-F(d)}{|c-d|}$= 0. Therefore, it will not see a need for making a step and stop.

Because of this problem of ending up with an unwanted point, we needed to check both directions when having a new step. That is when we are at point *a* GD computes both $\frac{F(a)-F(b)}{|a-b|}$ and $\frac{F(a)-F(c)}{|a-c|}$ and choses the best direction or stops.

We run GD in two ways. First one is running the algorithm for a fix amount of time and the second one is running it for a certain number of steps. Table 3.3 shows the pseudo-code of the GD.

Table 3.3: Pseudo-code of the Gradient Descent algorithm used in the experiments.

```
#Assume F is the cost function
Function step(x_old):
    x_temp=x_old.copy()
    for i in range(size(x_old)):
        x_temp2=x_old.copy()
        x_temp3=x_old.copy()
        x_temp2[i]+=h
        x_temp3[i]-=h
        d1,d2,d3=F(x_old),F(x_temp2),F(x_temp3)
        if d2<=d1 and d2<=d3 :
            x_temp[i]+=h
        else if d3<=d1 and d3<=d2 :
            x_temp[i]-=h
    return x_temp


Function GD(x, stepLimit):
    # Assume x has 2 dimensions
    x_old=x
    x_new=step(x_old)
    numberOfSteps=0
    while  numberOfSteps< stepLimit:
        x_old=x_new
        x_new=step(x_old)
        numberOfSteps++
    return x_new
```

3.2.1.1. A Modified GD (GD1)

Consider there are two cameras looking at the same direction from same place as in figure below.

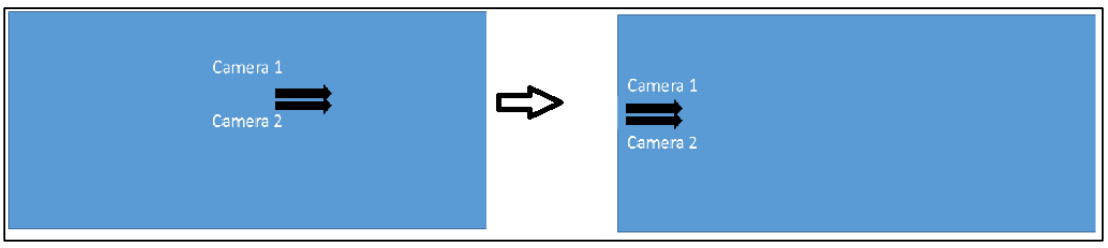

Figure 3.8: Cameras looking at the same direction still look at the same direction at the after applying GD.

In that case, after applying GD cameras will continue looking at the same direction. In order to overcome such situations, we modified the GD algorithm slightly such that when making a new step it changes the parameters gradually. The pseudo-code of this algorithm is shown at Table 3.4. This algorithm gave slightly better results than GD (see experiments section).

Table 3.4: Pseudo-code of the GD1 algorithm used in the experiments.

```
#Assume F is the cost function
Function step2(x_old):
    for i in range(size(x_old)):
        x_temp2=x_old.copy()
        x_temp3=x_old.copy()
        x_temp2[i]+=h
        x_temp3[i]-=h
        d1,d2,d3=F(x_old),F(x_temp2),F(x_temp3)
        if d2<=d1 and d2<=d3:
            x_old[i]+=h
        else if d3<=d1 and d3<=d2:
            x_old[i]-=h
    return x_old


Function GD1(x, stepLimit):
    # Assume x has 2 dimensions
    x_old=x
    x_new=step2(x_old)
    numberOfSteps=0
    while  numberOfSteps< stepLimit:
        x_old=x_new
        x_new=step2(x_old)
        numberOfSteps++
    return x_new
```

3.2.2. Particle Swarm Optimization

Particle swarm optimization [24] (PSO) mimics the food searching behavior of a group of animals. Each member of the group corresponds to a solution of the problem – in our case it is a list of parameters for the cost function which returns the visibility of the system.

Each particle in the system takes steps and tries to improve the quality of the current solution. When taking a new step each particle gets somewhat a composition of the current velocity, a vector from the current position to the particle's best position achieved that far and another vector from current position to the system's best solution achieved so far. Where these components multiplied with random

numbers between 0 and 1. Below is the pseudo-code of the PSO algorithm [25], [26], [27].

$$v[i] = wv[i] + c1 * rand * (pBest[i] - present[i]) + c2 * rand * (gBest[i] - present[i]) \quad (3.8)$$

$$present = present + v \quad (3.9)$$

Table 3.5 Pseudo-code of the PSO algorithm used in the experiments.

```
Initialize particles
while not maximumIteration exceeded:
    for each particle:
        calculate fitness value
        if fitness value better than fitness of pBest: #personal best value
            pBest=particle
    gBest=best of pBests #global best
    for each particle :
        calculate particle velocity according to equation (3.8)
        update the position of the particle according to equation (3.9)
```

PSO starts by initializing the particles. Algorithm runs until the maximum iteration number is reached. At each iteration, PSO computes the cost values of the particles and if the new cost value of a particle is better than the personal best of a particle, PSO updates the personal best of the particle. After updating the pBest values of the particles, PSO computes gBest. That is the best of the pBest values of each particle. Next PSO computes the velocity vector for each particle according to Equation (3.8). $rand$ is a random number between 0 and 1. $c1$ & $c2$ are constant numbers. These can be considered as the parameters of PSO. *i* parameter at Equation (3.8) is the *i*th parameter of the velocity vector *v*.

3.2.2.1. PSO Parameter Selection

Using nested loops, we checked many values for w, $c1$ and $c2$. We experimentally found that $w = 0.7$, $c1 = 0.95$ and $c2 = 0.95$ gives better results than the other values.

We also tried dynamic numbers for the *w* parameter. For example, we started with $w = 1$ and at each iteration of the algorithm, we multiplied w with a number. That is $w = w * k$. k is a number like 0.95. Many different k values were tried and

we compared the results with constant. The resulting graphs are in the experiments section.

3.2.2.2. A Variation of PSO (PSO1)

In order to explore the search space more we used global second best and global best interchangeably. This modification gave slightly better results than the original algorithm. Results are in the experiments section. In this variation of the PSO we used global second best and global best randomly at Equation (3.3) above. New form of equation is below.

$$v[i] = wv[i] + c1 * rand * (pBest[i] - present[i]) + c2 * rand * (gBestOrgSecondBest[i] - present[i]) \quad (3.10)$$

3.2.3. Artificial Bee Colony

Artificial bee colony algorithm [28] mimics a bee colony that searches food in the nature. Similar to PSO, ABC starts with a set of solutions and tries to improve this set of solutions iteratively. At the end, the best solution achieved that far is the result of the algorithm.

Components of the ABC algorithm are as follows:

- Food Sources: ABC starts with a set of solutions to the problem - in our case set of camera configurations - and tries to improve those solutions at each iteration. Food sources correspond to the set of the solutions which ABC tries to improve.
- Employed Bees: These are responsible for bringing food sources to the beehive and sharing the information about the richness or the quality of the source. In our case, a source is considered rich, if it returns a number close to 1 when evaluated at the cost function. Number of employed bees are equal to the number of food sources.
- Unemployed Bees: There are two types of unemployed bees. The first one is

the onlooker bees. These wait in the hive, use the information shared by employed bees, and try to find new food sources. Second type of unemployed bee is the scout bees. These try to find new food sources randomly without using any information from employed bees.

Below is a scheme about the elements of the ABC algorithm (see Figure 3.9). In our implementation of ABC number of food sources, number of employed bees and number of onlooker bees are equal. Outline of ABC is shown in Table 3.6 [29].

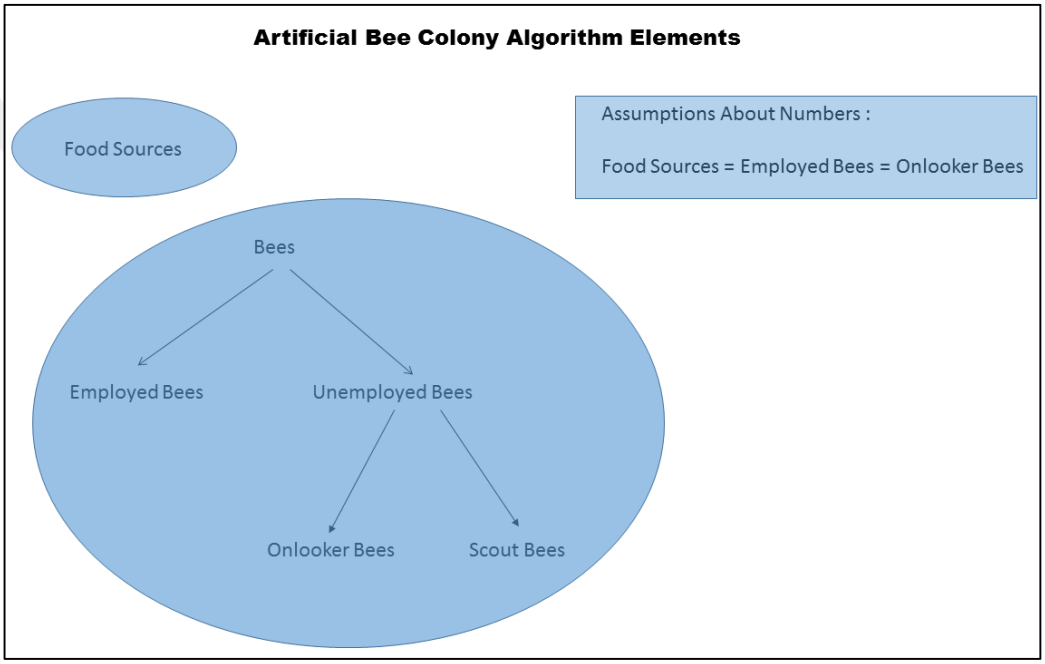

Figure 3.9: Elements of the Artificial Bee Colony Algorithm.

Table 3.6: Outline of the ABC used in the experiments.

```
Initialization Phase
REPEAT
  Employed Bees Phase
  Onlooker Bees Phase
  Scout Bees Phase
  Memorize the best solution achieved so far
UNTIL(Cycle=Maximum Cycle Number or a Maximum CPU time)
```

At initialization phase random solutions are created. At employed bees phase fitness values of the sources are evaluated. At onlooker bees phase quality of the sources are tried to be improved according to Equation (3.11):

$$v_{ij} = x_{ij} + \varphi_{ij}(x_{ij} - x_{kj}) \tag{3.11}$$

Where only one parameter namely j of the solution x_i is changed and the rest are remains unchanged. φ_{ij} is a random number in $[-1,1]$. x_{kj} is the jth parameter of solution x_k which is a randomly chosen solution from the solution set. v_i is the newly created solution by changing only the *j*th parameter of solution x_i.

At scout bees phase source that cannot be improved after a number of tries are abandoned and new sources are created instead of those. At the end of each iteration, the best solution is saved. Pseudo-code of ABC is shown in Table 3.7 [28].

Table 3.7: Pseudo-code of the ABC algorithm used in the experiments.

```
A set of random solutions is created
SN=size of solution set
Failure counters are initialized to 0
while stopping condition not satisfied:
        for i=1 to SN:
                create a new solution v_i for solution x_i according to equation (3.11)
                calculate fitness value of v_i
                if fitness(v_i) better than fitness(x_i):
                        x_i=v_i
                        failure_counter_of_x_i=0
                else:
                        failure_counter_of_x_i+=1
        Compute the probabilistic values of the sources p_i which will
        be used by onlooker bees when making a choice of source
        t,i=0,0
        while t<SN:
                if rand()<p_i:
                        create a new solution v_i for solution x_i according to equation
(3.11)
                        calculate fitness value of v_i
                        if fitness(v_i) better than fitness(x_i):
                                x_i=v_i
                                failure_counter_of_x_i=0
                        else:
                                failure_counter_of_x_i+=1
                        t+=1
        if max{failure_i}>limit:
             x_i=create a random solution for x_i
        Memorize the best solution
```

3.2.3.1. A Variation of ABC (ABC1)

In this variation a different improvement function, Equation (3.12), is used at the employed bees stage instead of Equation (3.11) above.

$$v_{ij} = x_{ij} + \varphi_{ij} \tag{3.12}$$

Here x_i is the *i*th solution of the solution set. x_{ij} is the *j*th parameter of solution *i*. For parameters that are related to distances φ_{ij} is a is a random number in [-1,1] and for parameters that are related to angles φ_{ij} is a random number in [-0.1,0.1]. Here we have changed all parameters of the solution x_i according to Equation (3.12) not only one parameter.

3.2.4. Run Time Analysis of the Method

Let *a=numberOfVoxels*, *m=numberOfCameras* and *n=numberOfObstacles*. Running time of isInFOV function is constant so isInFOV = $\Theta(c_1)$. Checking if a voxel is occluded by an obstacle takes $O(c_2)$ time. So running time of the cost function is Equation (3.13).

$$\Theta(mnk)\Theta(c_1)O(c_2) = \Theta(mnk) \tag{3.13}$$

Running time of GD is Equation (3.14).

$$O(m)\Theta(mnk) = O(m^2nk) \tag{3.14}$$

Running time of PSO will increase relative to the increase in the run time of the cost function but same number of iterations will not be enough if the number of cameras increases. The same condition is valid for ABC.

4. EXPERIMENTS

The algorithms described in the previous section have been implemented and tested on four different scenarios. These scenarios represent difficulty levels ranging from a simple rectangular room to more complex environments with obstacles. The visible volume of a given configuration is calculated by counting the visible voxels and normalizing by the number of voxels such that '1' means a total visibility. The run-time performance of this volume calculation depends on the number of voxels, or the grid size. We use GPU to make this calculation faster. Figure 4.6 shows the performance of a CUDA implementation (running on ZOTAC GTX 780 with 2304 cores and 3GBmemory) compared against a C++ implementation (running on a 3.50 GHz CPU with 16GB memory). As expected the GPU implementation clearly outperforms the GPU implementation. The cameras used in these experiments have 60˚ horizontal and 45˚ vertical field-of-view (FOV). We have implemented and tested 6 different algorithms. Using the same set of random initializations, we let each algorithm run and converge to a state and record the calculated volume. All of the graphs and values below are the average values of many tests.

4.1. Test Scenarios

Tests were conducted using four different test scenarios as seen below. At the left, there is the 2D shape of the test scenario and at the right, there is the 3D shape of the test scenario. First scenario, scenario A, is a simple shape with no obstacles in the room but the surrounding walls. The dimensions of the room in scenario A is 14m×4m×3m.

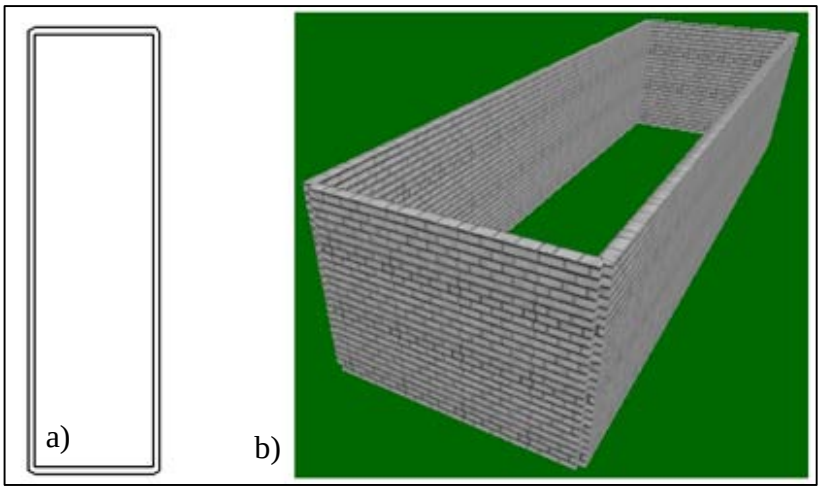

Figure 4.1: Test Scenario A for the optimization algorithms, a) 2D version, b) 3D version.

The second one, scenario B, is a more complex shape which is obtained from a real situation where up to four clerks are serving several hundred people a day. The camera planning result for this case is used in an application where customer satisfaction is measured using a multi-camera system. The dimensions of the room in scenario B is 14m×7m×3m.

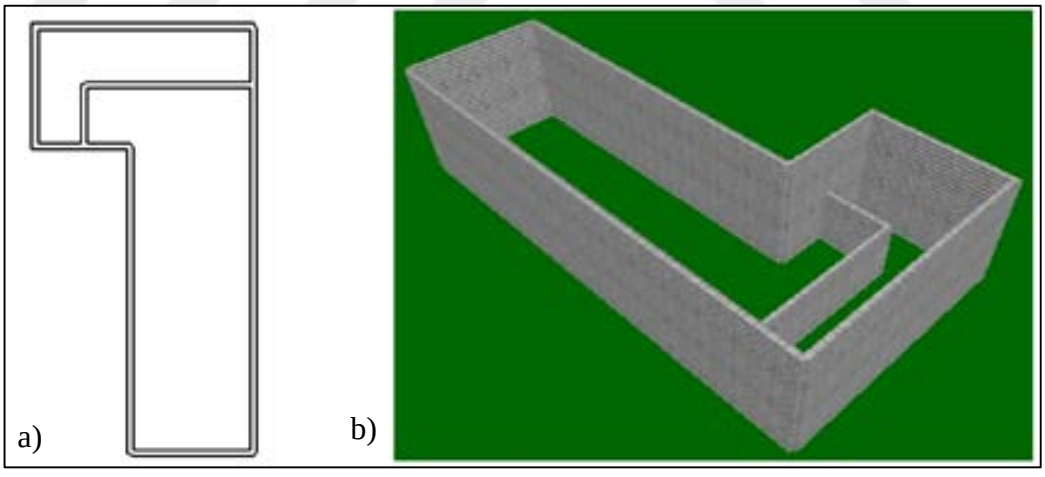

Figure 4.2: Test Scenario B for the optimization algorithms, a) 2D version, b) 3D version.

The third scenario, scenario C, is a more complex one than the previous ones. The dimensions of the room in scenario C is 20m×16m×3m. This scenario has a complex shape when compared to the others. Cameras need to be carefully placed to observe some parts of this scenario.

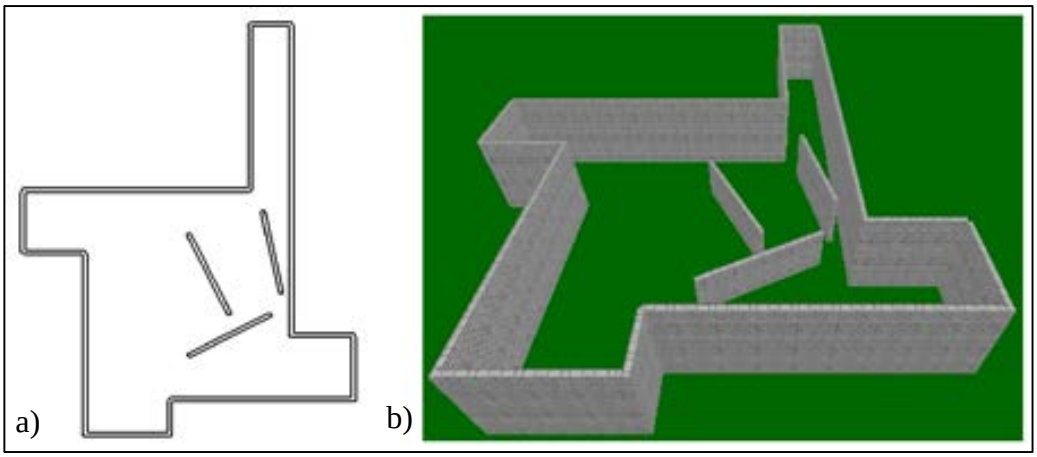

Figure 4.3: Test Scenario C for the optimization algorithms, a) 2D version, b) 3D version.

The last test scenario, scenario D is a square shaped room with many obstacles in it. The dimensions of the room in scenario D is 20m×20m×3m. This scenario is created to see the effect of many obstacles in the scene when running our algorithms.

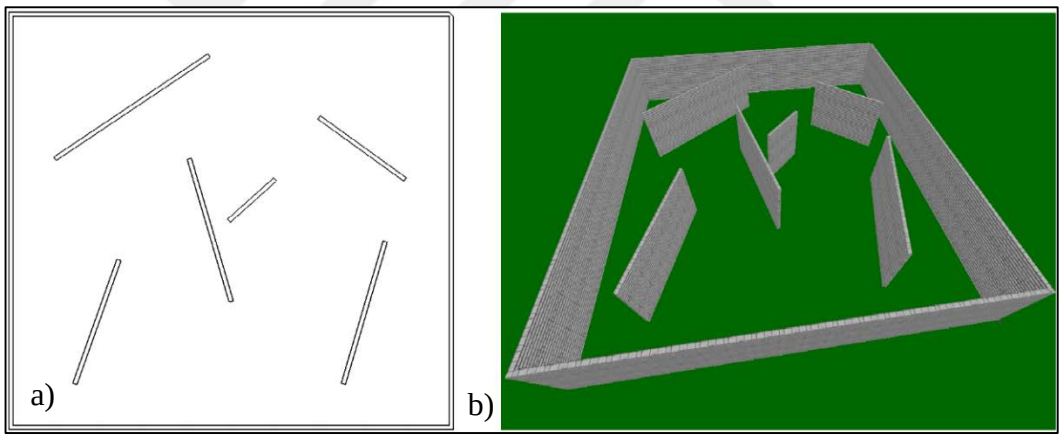

Figure 4.4: Test Scenario D for the optimization algorithms, a) 2D version, b) 3D version.

4.2. CUDA and CPU Speed Comparison

In Figure 4.5 running times of CPU and GPU are compared for different number of voxels for 58 obstacles in total. Y axis shows the running time in terms of milliseconds. At each case CPU and GPU based cost functions computed the coverage for 100 different camera configurations. The results at the Y-axis are the average of those 100 test results. The numbers at the axes are logarithmic numbers. For example, 15 at the x axis means $2^{15} \approx 30000$ voxels and 8 at the y axis means

$2^8 \approx 128$ ms. As clearly can be seen from the graph as the number of voxels increase the ratio of CPU and GPU running times increases. That is for small number of voxels CPU and GPU based cost functions run in closer times where as the number of voxels increases GPU based one runs much faster than CPU based one.

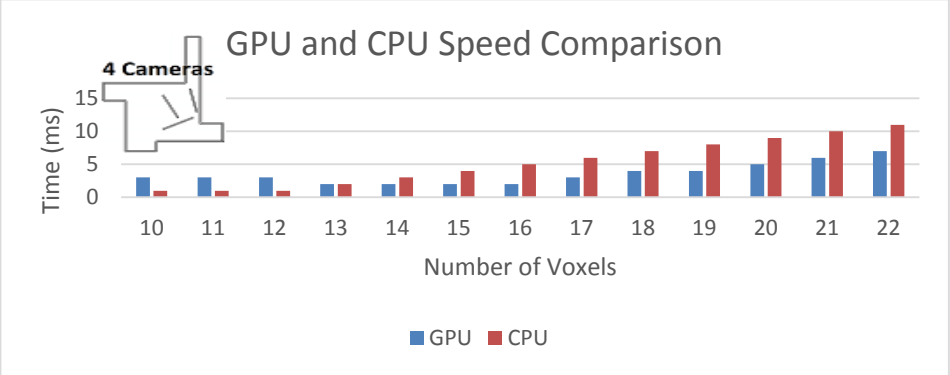

Figure 4.5: Speed comparison of CPU and GPU versions of the cost function we use at the experiments at scenario C for four cameras. Numbers at both axes logarithmic.

Figure 4.6 is the running time graphics of the CPU and GPU based cost functions for the case we mostly used at the experiments. That is for roughly 2000000 voxels. For 1000 different camera configurations –where each camera configuration includes 4 camera locations and angles- CPU and GPU based cost functions were run and average of the test results are used at the graphics. Our experiments show that GPU based cost function runs 17.3 times faster than the CPU based one.

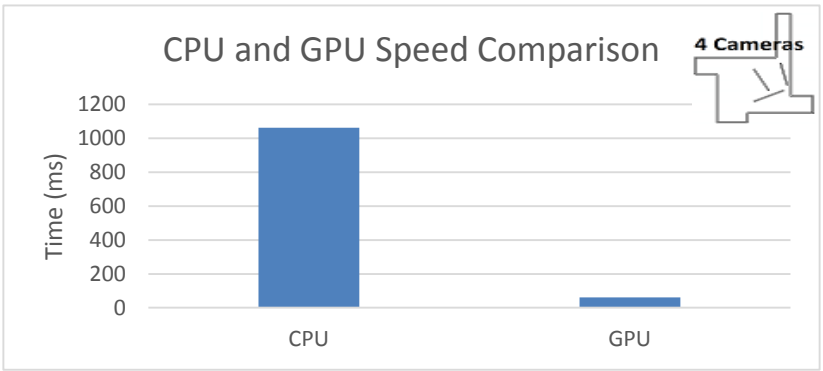

Figure 4.6: CPU and GPU speed comparison of the cost function for 2000000 voxels at scenario C for four cameras.

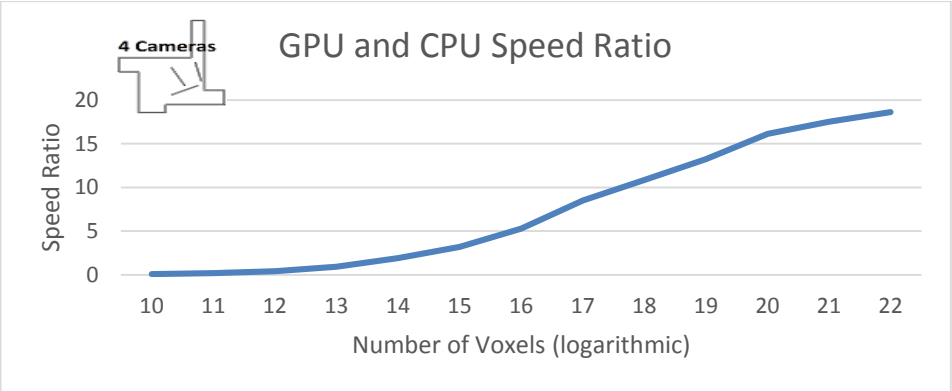

Figure 4.7: Speed ratios of GPU and CPU as the number of voxels increases for Figure 4.5.

As the number of voxels increases, the ratio between the GPU and CPU based versions increases. We can expect that this ratio will eventually converge to a limit.

4.3. Random Initialization and a Better Initialization

In this section random and a better then random camera initialization techniques are compared. When initializing a camera randomly a random point inside the test room is chosen. Next the angles of the camera are initialized randomly where horizontal angle is a random number in $[0,2\pi]$ and vertical angle is random number in $[-\pi/2, \pi/2]$. At the second Not-Random initialization technique we chose the x, y coordinates randomly, set z coordinate equal to 3 and directed the cameras to the center of the 3D scene. Second technique produced better initial values for the algorithms. Because that decreases the probability of a newly initialized camera looking at a wall. At scenario B some walls have the height of 1.5m. In such a case, a randomly created camera location at the bottom of the room will have less good coverage than a camera location at the top of the room. Table 4.1 shows the comparison of the results of the two initialization techniques for different scenarios and different number of cameras. The second technique, Not-Random one yields much better results than the random initialization technique.

Table 4.1: Comparison of random initialization and a Not-Random initialization technique for different scenarios and different number of cameras.

Scenario	Number of Cameras	Random	Not Random
A	1	0.05	0.40
A	2	0.08	0.61
B	2	0.05	0.42
B	3	0.09	0.54
C	3	0.07	0.29
C	4	0.09	0.35
C	5	0.12	0.41

4.4. Tests with Different Scenarios

In this section, there are visibility graphics for different scenarios and for different number of cameras. We have implemented six algorithms and run those algorithms equal amounts of times at different test scenarios. We made convergence graphics of those algorithms. We used the same set of initial solutions for each of the algorithm. PSO and ABC needs a set of initial solutions for working but GD needs one solution. Figure 4.8 shows the results of running six algorithms at scenario A for one camera. We run each test equal amount of times. In this test, time period is 30 seconds. We see that algorithms start from a quite good initial point and converge to a limit in 10 seconds. We see that ABC and PSO1 are best and the second best respectively.

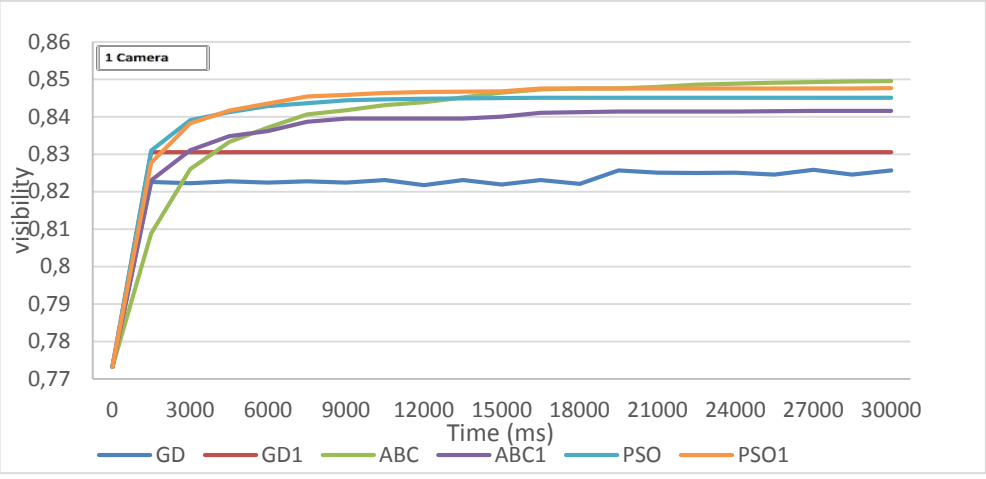

Figure 4.8: Results of running six different algorithms for scenario A with one camera for 30 seconds. Figure 4.9 shows results of running six algorithms at scenario

A for 2 cameras for 90 seconds. Although four algorithms converged to visibility ratio 1 in very short amount of times we see that PSO1 converged first. The tests at Figure 4.9 and Figure 4.10 are trivial.

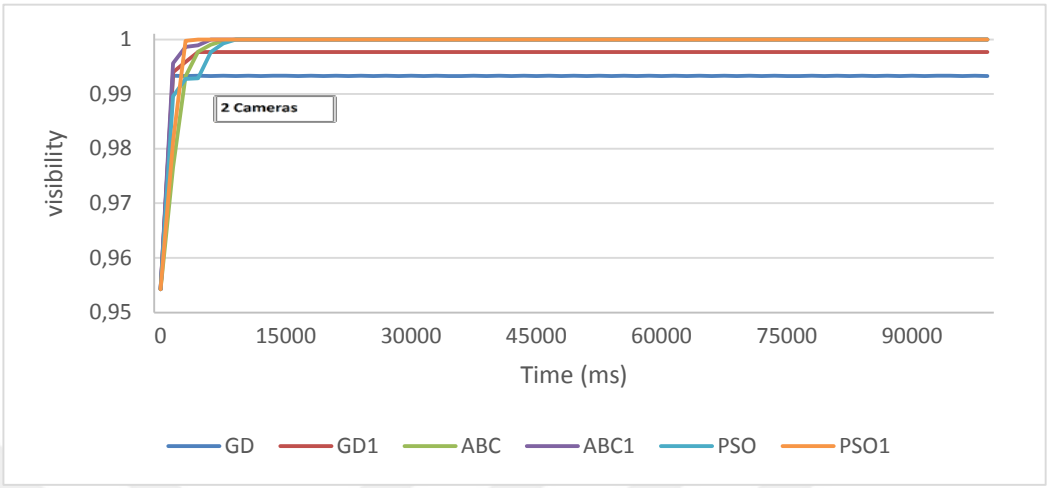

Figure 4.9: Results of running 6 different algorithms for scenario A with 2 cameras for 90 seconds. Figure 4.10 shows the results of running six algorithms at scenario B for 2 cameras for 150 seconds. We see that ABC and PSO1 are the best and second best algorithms for this test respectively.

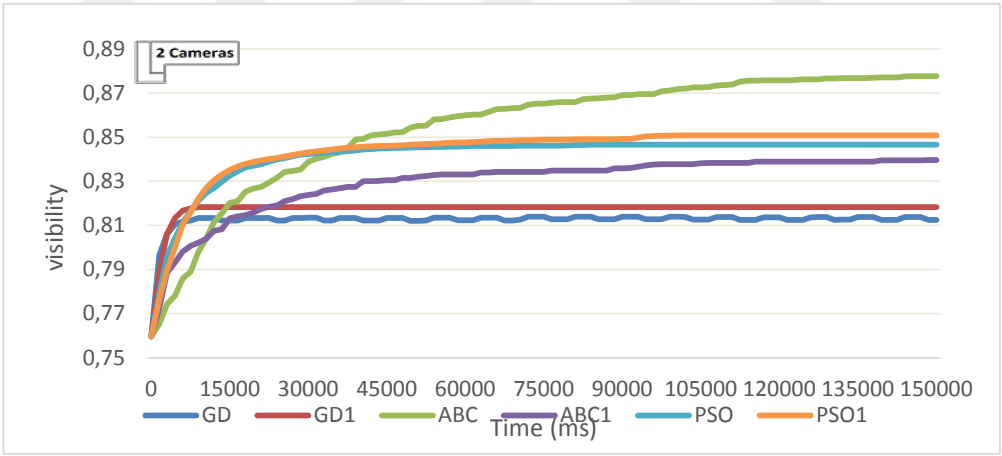

Figure 4.10: Results of running 6 different algorithms for scenario B with 2 cameras for 150 seconds. Figure 4.11 shows the results of running six algorithms at scenario B for 3 cameras for 240 seconds. We see that ABC and PSO1 are the best and second best algorithms for this test respectively.

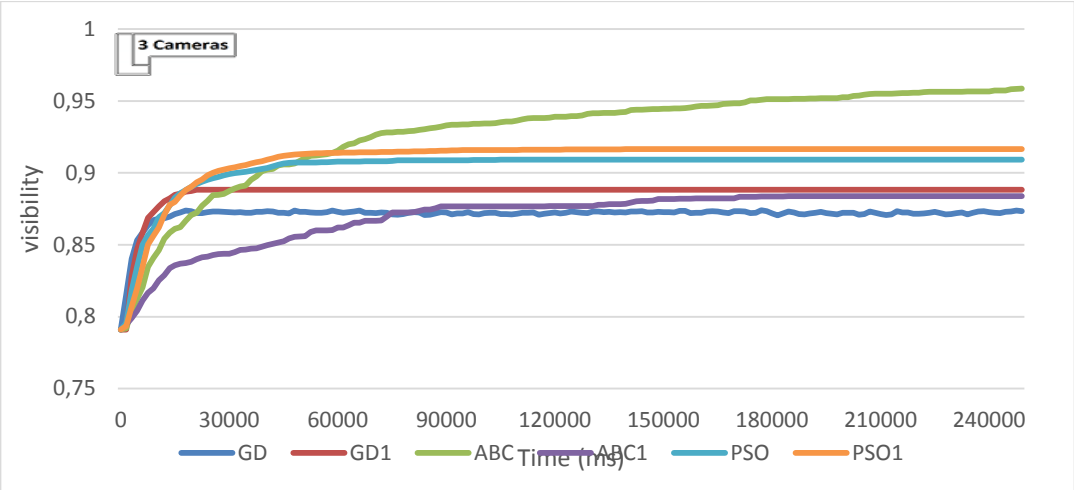

Figure 4.11: Results of running six different algorithms for scenario B with 3 cameras for 240 seconds. Figure 4.12 shows the results of running six algorithms at scenario C for 3 cameras for 600 seconds. We see that ABC and PSO1 are the best and second best algorithms for this test respectively.

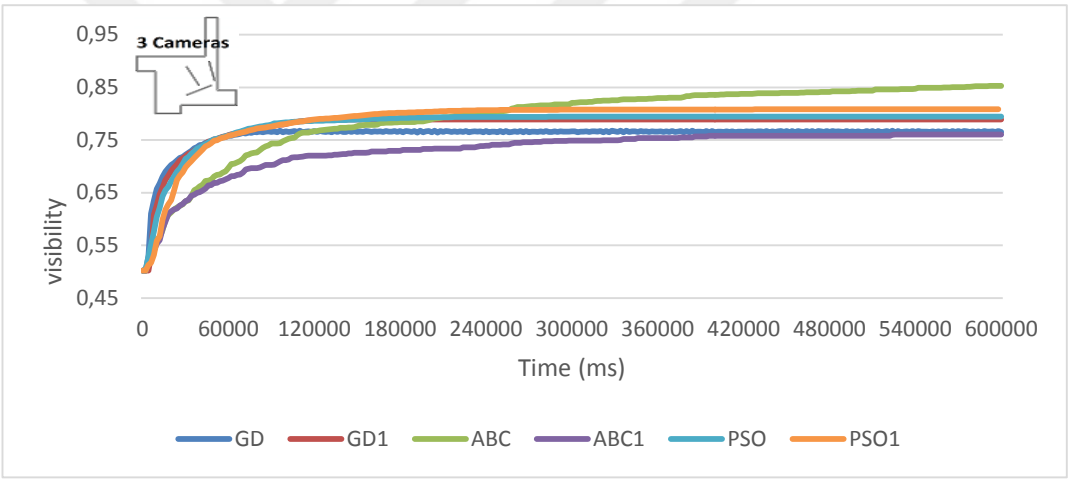

Figure 4.12: Results of running six different algorithms for scenario C with 3 cameras for 6000 seconds. Figure 4.13 Shows the results of running six algorithms at scenario C for 4 cameras for 900 seconds. We see that ABC and PSO1 are the best and second best algorithms for this test respectively.

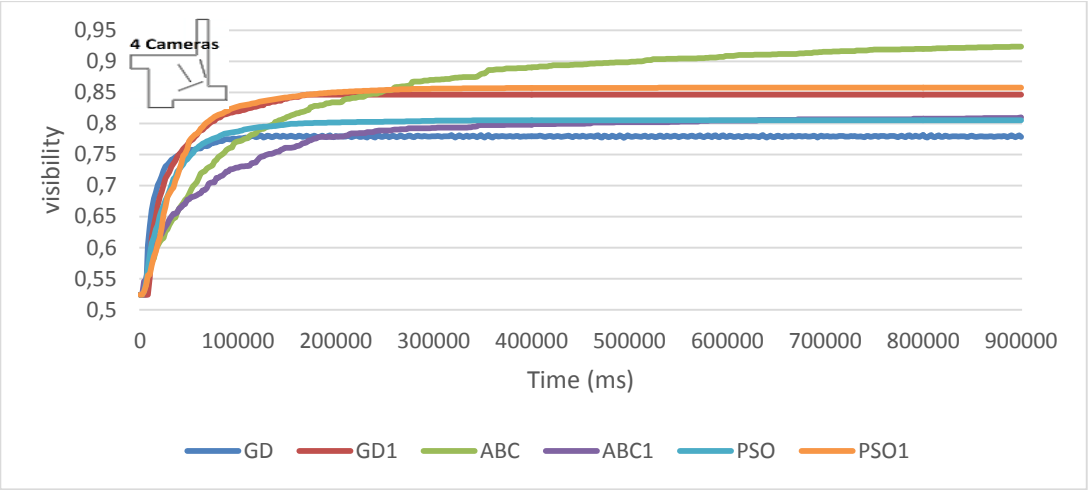

Figure 4.13: Results of running six different algorithms for scenario C with 4 cameras for 900 seconds. Figure 4.14 shows the results of running six algorithms at scenario C for 5 cameras for 900 seconds. We see that ABC and GD1 are the best and second best algorithms for this test respectively.

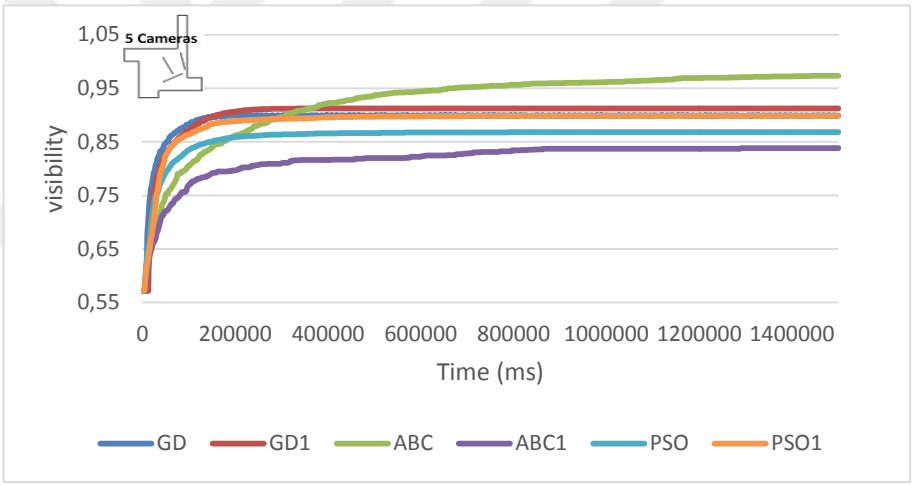

Figure 4.14: Results of running six different algorithms for scenario C with 5 cameras for 1400 seconds.

Table 4.2 shows the standard deviations of the results of the above algorithms for different scenarios and for different number of cameras. We see that the standard deviation of ABC algorithm is nearly always the smallest of the six algorithms.

Table 4.2: Standard Deviation of the Results of Algorithms for different scenarios and for different number of cameras.

Scenario	Number of Cameras	GD	GD1	ABC	ABC1	PSO	PSO1
A	1	0.012	0.006	0.001	0.003	0.005	0.004
A	2	0.024	0.008	0.0	0.0	0.0	0.0
B	2	0.012	0.011	0.012	0.006	0.014	0.016
B	3	0.027	0.024	0.010	0.012	0.040	0.030
C	3	0.076	0.067	0.012	0.027	0.051	0.069
C	4	0.071	0.051	0.015	0.037	0.054	0.073
C	5	0.042	0.047	0.009	0.024	0.046	0.056

4.4.1. Different Voxel Sizes

Figure 4.15 shows the result of testing the same algorithms with different voxel sizes. We used the same set of random numbers for more comparable results. Here also we used the same number of algorithm steps not same amount of times because when the number of voxels increases algorithm requires more time to complete a step. Results are expected they are mostly similar. However precision increases as the number of voxels increases.

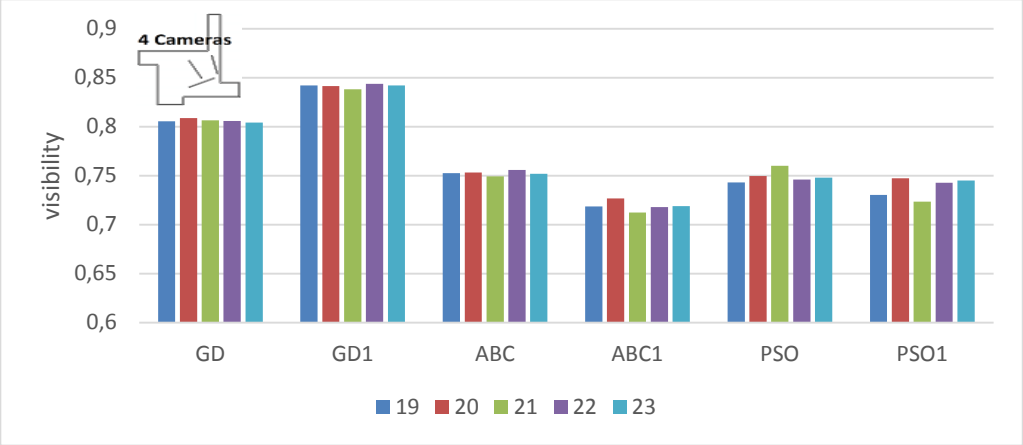

Figure 4.15: Same tests with different voxel sizes. Number of voxels are logarithmic.

4.4.2. w Parameter of PSO Algorithm

We tested two different approaches for the value of the w parameter at PSO. First approach is a constant w, where we chose w=0.7. Second approach is gradually

decreasing the value of w. We started with w=1 and at each iteration updated the value of w before using it by multiplying it with a number k. That is at each iteration w=w*k. Figure 4.16 compares constant w and 7 different k values ranging from 0.65 to 0.95. We figured out that w=0.7 gives better results than dynamic w parameter. However, as the value of k increases algorithms gave better results. Therefore, we made one more test where the value of k ranges from 0.96 to 0.99. Figure 4.17 shows the result of the second test. Constant w=0.7 gives still better results than the dynamic w.

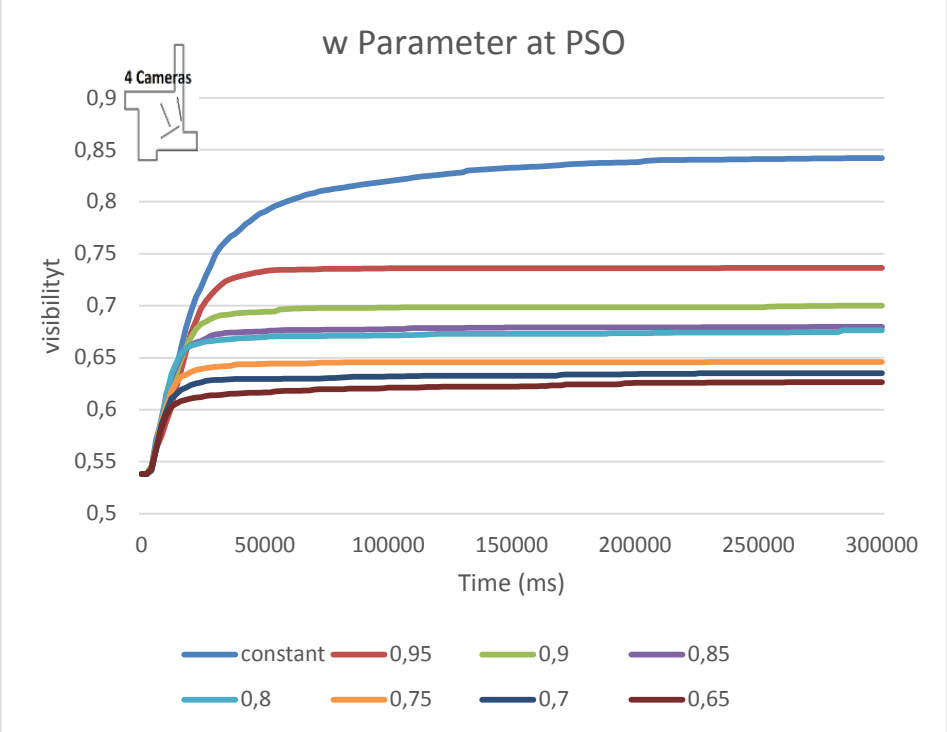

Figure 4.16: Constant and dynamic w parameter at PSO.

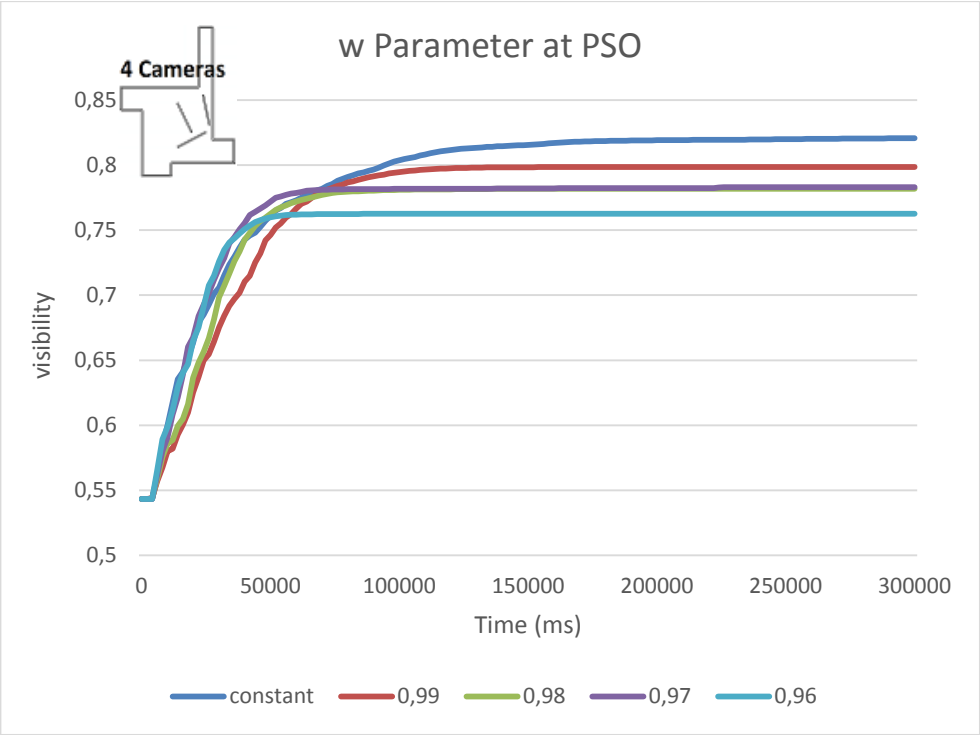

Figure 4.17: Constant and dynamic w parameter at PSO.

Figure 4.18 shows the change of the w parameter during the tests at Figure 4.17. w starts with 1 and at each iteration it is multiplied with a k value except the constant one.

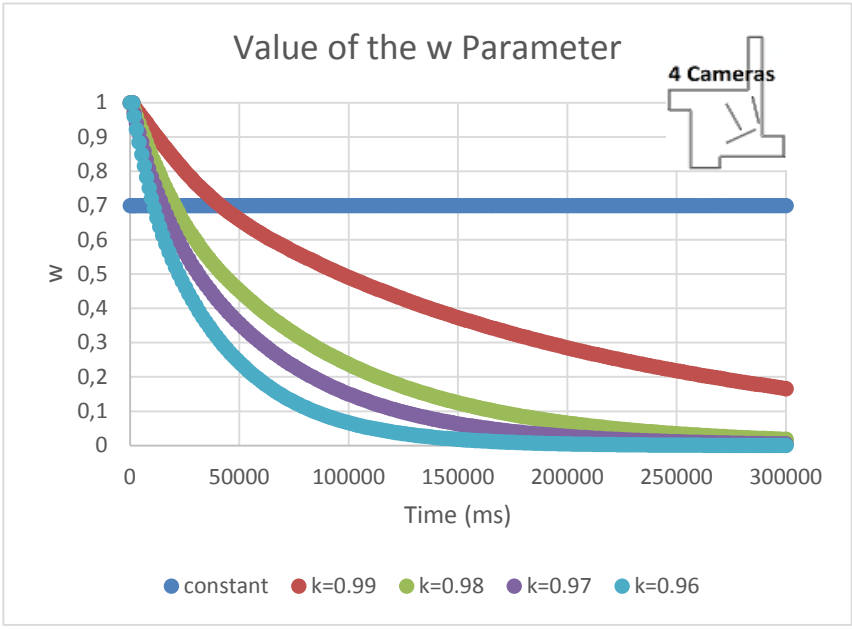

Figure 4.18: Value of the w parameter during the tests at Figure 4.17.

4.4.3. Mixture of Algorithms

Although some algorithms give better results than some other in the long run, some poor resulting algorithms gives better results in short amount of times. Therefore, we decided to combine those algorithms to get better results. We applied GD1 to ABC and PSO after running them for some time and compare the result with a pure ABC algorithm. We did the same thing for PSO as well.

Below are the graphs of ABC and ABC+GD1 comparison and PSO and PSO+GD1 comparisons for two different test scenarios and for different number of cameras. We applied ABC and PSO 3x amount times and at the hybrid algorithm applied ABC and PSO 2x amount of time and applied GD1 x amount of time.

Figure 4.19 shows the results of running ABC and ABC+GD1 and PSO and PSO+GD1 for 300 seconds at scenario C for 3 cameras. Hybrid algorithms performs better than their pure counterparts in both cases.

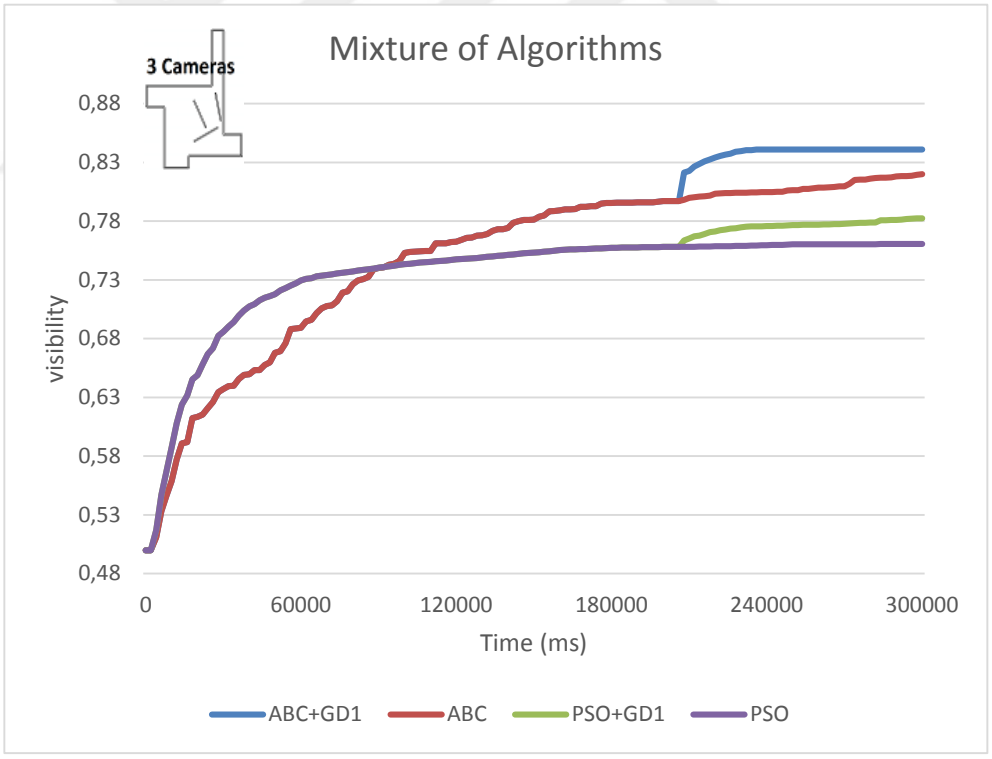

Figure 4.19: Shows 2 hybrid algorithms compared to their pure counterparts at scenario C for 3 cameras for 300 seconds. Figure 4.20 shows the results of running ABC and ABC+GD1 and PSO and PSO+GD1 for 500 seconds at scenario C. Hybrid algorithms performs better than the pure counterparts.

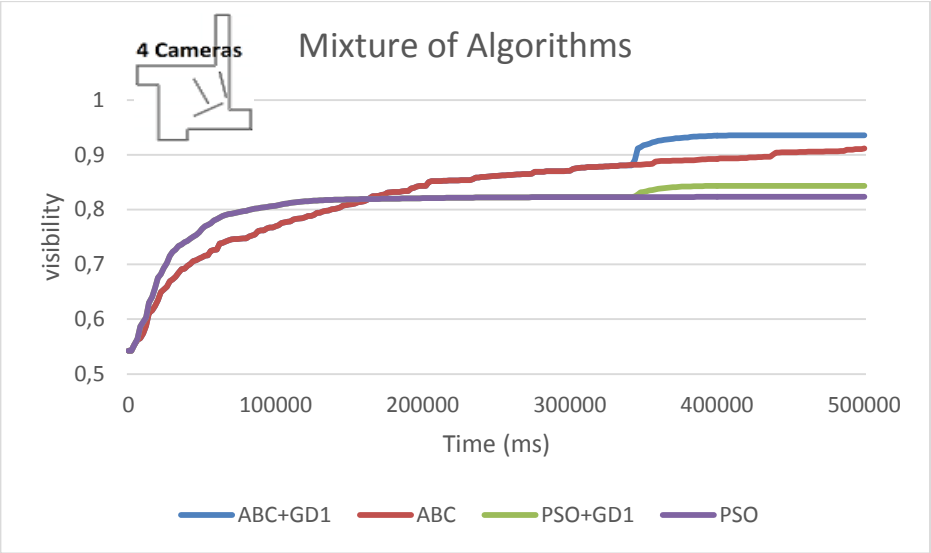

Figure 4.20: Shows 2 hybrid algorithms compared to their pure counterparts at scenario C for 4 cameras for 500 seconds. Figure 4.21 shows the results of running ABC and ABC+GD1 and PSO and PSO+GD1 for 300 seconds at scenario D for 3 cameras. Hybrid algorithms performs better than the pure counterparts.

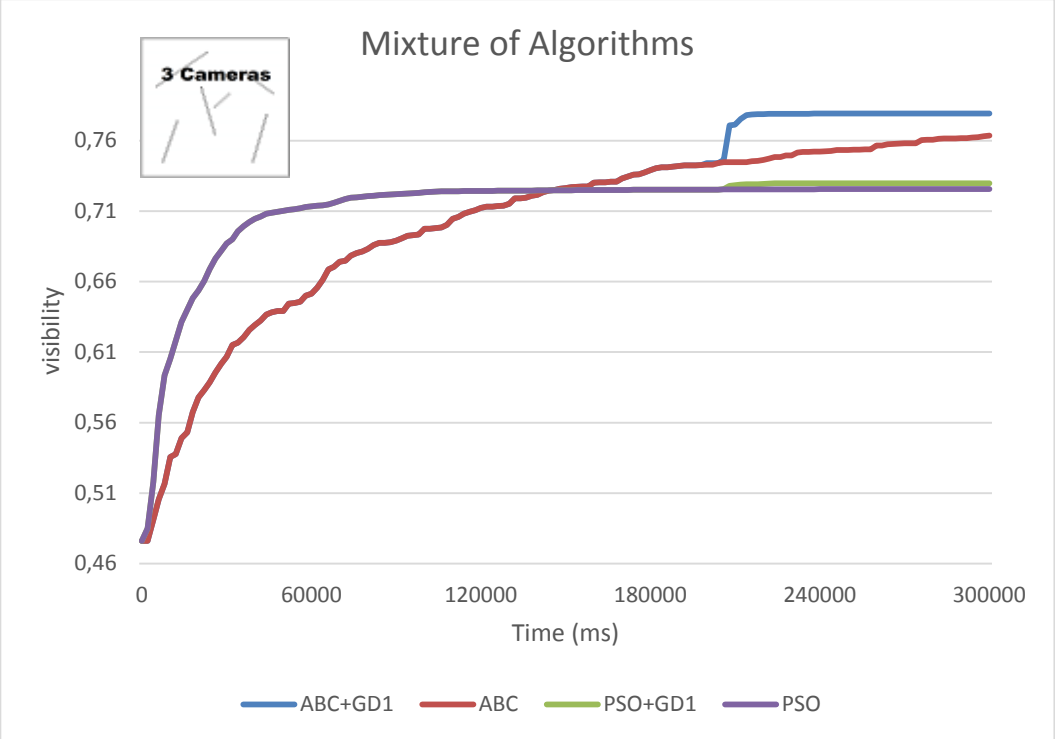

Figure 4.21: Shows 2 hybrid algorithms compared to their pure counterparts at scenario D for 3 cameras for 300 seconds. Figure 4.22 shows the results of running ABC and ABC+GD1 and PSO and PSO+GD1 for 500 seconds at scenario D for 4 cameras. Hybrid algorithms performs better than the pure counterparts.

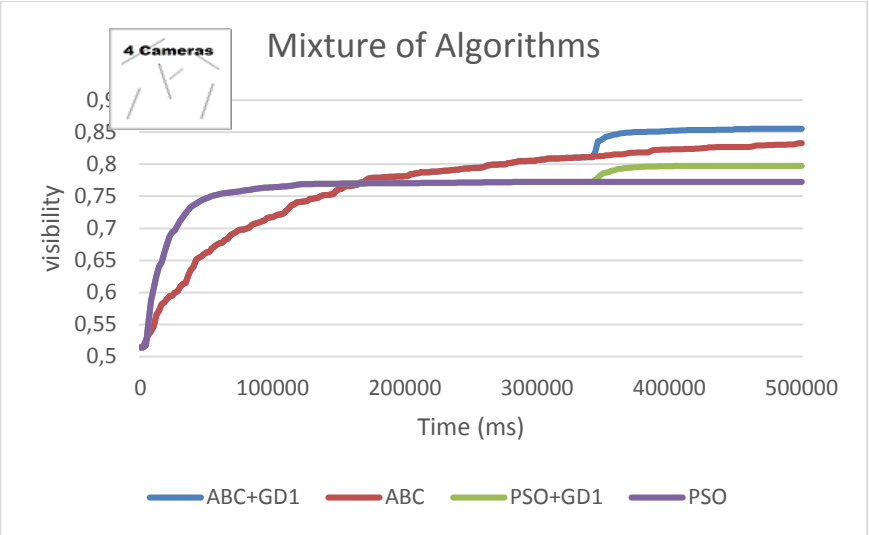

Figure 4.22: Shows 2 hybrid algorithms compared to their pure counterparts at scenario D for 4 cameras for 500 seconds.

At all of the tests hybrid algorithms performed better than their pure counterparts. The reason of this is some algorithms perform well in short times and perform bad in long times and vice versa. Combining the true algorithms give better results than all the other algorithms.

4.4.4. Different Field of Views

Figure 4.23 shows the results of applying same algorithms at different FOVs to the same solution set. FOVs are 45-60, 60-90 and 90-120 degrees. As expected as FOV increases algorithms give better results.

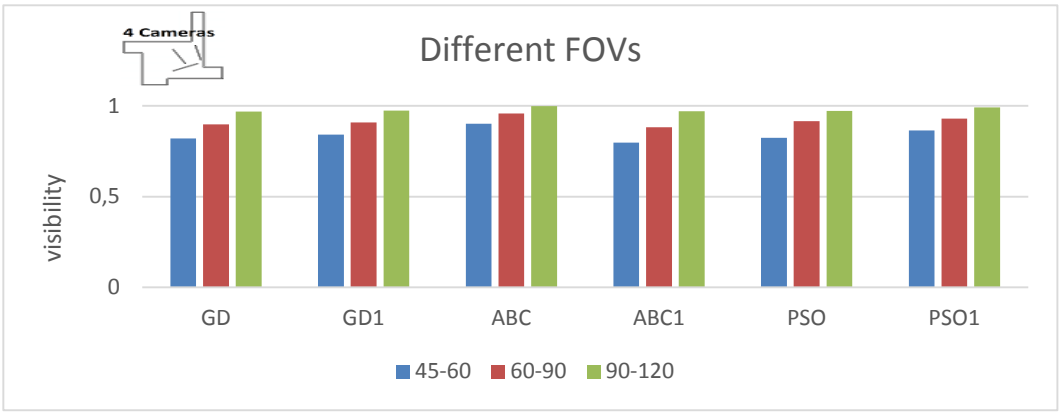

Figure 4.23: Results of running same algorithms at different FOVs.

4.5. 2D Visualization of the Results

Figure 4.24, Figure 4.25 and Figure 4.26 show the results of camera planning for the first scenarios A, B and C respectively. The left drawing shows the visibility of the volume (cross section at height 1m) before applying GD. The camera configurations are of the randomly chosen configurations. The visible areas are shown in dark gray. The right drawing illustrates the visible volume after the GD algorithm is run. As it can be seen, the visibility is increased after the optimization.

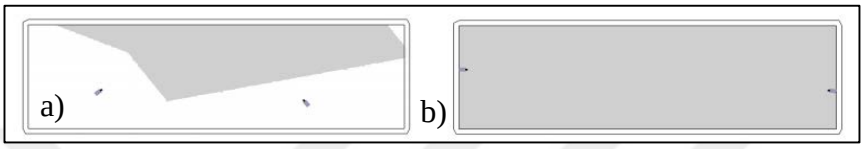

Figure 4.24: 2D visualization of applying GD to the scenario A for 2 cameras a) At the left, there is the initial coverage of the cameras, b) At the right there is the coverage of the cameras after applying GD.

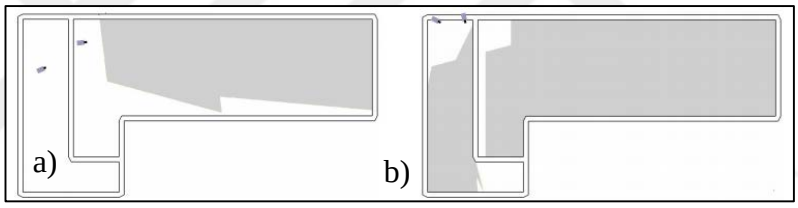

Figure 4.25: 2D visualization of applying GD to the scenario B for 2 cameras a) At the left, there is the initial coverage of the cameras, b) At the right there is the coverage of the cameras after applying GD.

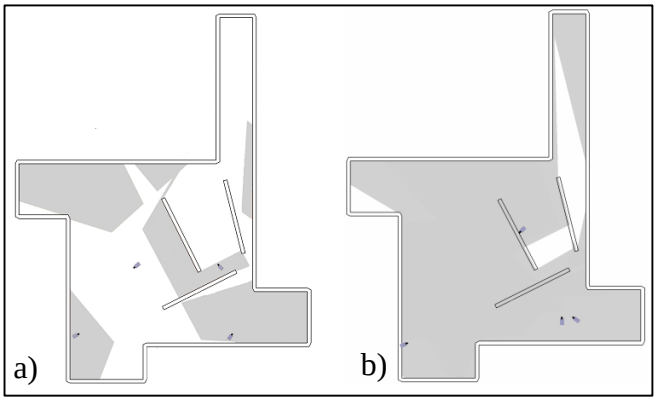

Figure 4.26: 2D visualization of applying GD to the scenario C for four cameras a) At the left, there is the initial coverage of the cameras, b) At the right there is the coverage of the cameras after applying GD.

4.6. Discussion about Experiments

We tested 6 algorithms on 4 different test scenarios. For more comparable results we gave equal amounts of times to the algorithms and the same initial solution sets. We tested the effect of GPU usage on the cost function and saw that the GPU version of the same cost functions works 20 times faster than the CPU version of the function in our test cases. Same GPU and CPU comparison tests were done with different voxel sizes as well. As the voxel size decreases and as the number of voxels inreases the ratio between the GPU and CPU running times increases. We made some tests about the ideal voxel size and ideal number of voxels. We checked the precison of the results. It is figured out that 6cm voxel size is both fast and precise enough and for our hardware and problem type. Recall that dimensions of the rooms are 14m×4m×3m, 14m×7m×3m, 20m×16m×3m and 20m×20m×3m. We used 6cm voxel size at the rest of the tests.

It is figured out that in the long run ABC gives better results than the rest of the algorithms. PSO1 is the second best algorithm in the long run. PSO1 is better than PSO. PSO keeps global best and personal best values but PSO1 keeps global best and global second best values as well as personal best values. So keeping glabal second best gave better results.

PSO gets somewhat an average of the personal best, global best and velocity. But this does not always guarantee a good solution for our problem type. However ABC tries to imrove more promising solutions most and less promising least and gives better results. ABC seems a better choice if there are more than one good points or good solutions in the problem domain.

ABC constantly tries to improve the current solutions it has. It randomly changes one of the parameters of the solution and keeps the new solution if it is better. Instead of changing only one parameter, ABC1 makes a perturbation around the solution. That changes all of the parameters slightly. But ABC still gave better results than ABC1. Maybe some further studies can be made to improve the improvement function of ABC.

We used to different approaches at the w parameter of the PSO algorithm. First approach is to use a constant w value. We chose w=0.7 after triying many possible values in a loop. Other approach is to decrease w gradually by multiyplying it with a number k. We tried many k values but the first approach gave better results.

Some algorithms run good in the short and and some run good in the long run. This lead us to mix some algorithms which gave better results than their pure counterparts. ABC+GD1 performed better than ABC and PSO+GD1 performed better than PSO. Because GD1 performs better in short run. Therefore, before ABC and PSO finish running we applied GD1 to the best solution in the solution set of ABC and PSO. Hybrid approach gave better results than their pure counterparts..

5. CONCLUSION

In this study camera placement problem is converted to an optimization problem. A cost function for evaluating the visibility of a 3D space for the given camera configuration is defined. The cost function returns a value between 0 and 1 which symbolizes the visibility ratio for the given camera configuration. Next by using several optimization algorithms, the cost function is optimized.

Parallel computing using CUDA helped us to run our tests much faster. Without CUDA programming it would be really hard to make this study because of too long running times of the algorithms. CUDA made our calculations roughly 20 times faster. A fast and practical solution to the camera placement problem is proposed and a system for evaluating the visibility of a given configuration and some methods for automatically proposing camera location is implemented.

We tried a bunch of algorithms for optimizing the cost function. These algorithms are ABC and a variant of it, PSO and a variant of it and Gradient Descent and a variant of it. Gradient Descent performs well in the short run but not long. PSO variant gives better result than PSO. ABC algorithm yields better results than the other algorithms. We also tried some hybrid algorithms. ABC+GD1 algorithm performed better than a pure ABC.

Some addition constraints to the system can be added at the future studies. These could be:

- Some points in the surveillance area can be more important than other points. An importance factor can be added to the points in the surveillance area.
- Effective range of a camera constraint can be added so that a point will not be considered observed by a camera if it is in the field of view of the camera and if it is not in the effective range of the camera.
- For some applications user of the system may want any point is visible by at least k cameras. Such a constraint can easily be added to the system if needed.

Results obtained during the study seem promising. Because in reasonable amounts of time (like 5, 10 or 15 minutes) we obtained quite good practical solutions.

REFERENCES

[1] Aggarwal A., (1984), “The art gallery theorem: Its variations, applications and algorithmic aspects”, Unpublished Doctoral Dissertation, The Johns Hopkins University.

[2] Web 1, (2016), https://en.wikipedia.org/wiki/Art_gallery_problem, (Erişim Tarihi: 17/06/2016).

[3] Avriel M., (2003), “Nonlinear programming: analysis and methods.”, 1 st Edition, Courier Corporation.

[4] Bárány I., (1987), “Computing the volume is difficult.”, Discrete & Computational Geometry, 2(4), 319-326.

[5] Chvatal V., (1975), “A combinatorial theorem in plane geometry.”, Journal of Combinatorial Theory, Series B, 18(1), 39-41.

[6] Conci N., (2009), “Camera placement using particle swarm optimization in visual surveillance applications”, 16th IEEE International Conference on Image Processing, 3485-3488, Cairo, Egypt, October.

[7] Eberhart R. C., (2001), Swarm intelligence, 1 st. Edition, Elsevier.

[8] Web 2, (2016), https://en.wikipedia.org/wiki/Finite_difference, (Erişim Tarihi: 21/06/2016).

[9] Fisk S., (1978), “A short proof of Chvátal's watchman theorem”, Journal of Combinatorial Theory, Series B, 24(3), 374.

[10] González-Baños H., (2001), “A randomized art-gallery algorithm for sensor placement”, 232-240, New York, USA, June.

[11] Gonzalez-Barbosa J. J., (2009), “Optimal camera placement for total coverage”, IEEE international conference on Robotics and Automation, 3672-3676, Kobe/Japan, May.

[12] Web 3, (2016), https://en.wikipedia.org/wiki/Gradient_descent, (Erişim Tarihi: 21/06/2016).

[13] Horster E., (2006), “Approximating optimal visual sensor placement”, IEEE International Conference on Multimedia and Expo, 1257-1260, Toronto, Canada, July.

[14] Hörster E., (2006), “On the optimal placement of multiple visual sensors”, ACM international workshop on Video surveillance and sensor networks, 111-120, New York, USA, November.

[15] Web 4, (2016), http://www.swarmintelligence.org/tutorials.php, (Erişim Tarihi: 23/06/2016).

[16] Kahn J. K., (1983), “Traditional galleries require fewer watchmen”, SIAM Journal on Algebraic Discrete Methods, 4 (2), 194-206.

[17] Web 5, (2016), http://www.scholarpedia.org/article/Artificial_bee_colony_ algorithm, (Erişim Tarihi: 2016/6/23).

[18] Karaboga D., (2007), “A Powerful and Efficient Algorithm for Numerical Function Optimization: Artificial Bee Colony (ABC) Algorithm”, J. of Global Optimization, 39(3), 459-471.

[19] Karaboğa D., (2011), “Yapay Zeka Optimizasyon Algoritmaları”, 2. Baskı, Nobel Yayın.

[20] Kennedy J., (1995), “Particle swarm optimization”, 1942-1948, New York,USA, 1995(11).

[21] Marzal J., (2012), ”The three-dimensional art gallery problem and its solutions”, Doctoral Dissertation, Murdoch University.

[22] Michael T. S., (2009), “How to guard an art gallery and other discrete mathematical adventures”, 1 st Edition, JHU Press.

[23] Michael, T. S., (2011), “Guards, Galleries, Fortresses, and the Octoplex”, The College Mathematics Journal, 42(3), 191-200.

[24] Mittal A., (2004), ”Visibility analysis and sensor planning in dynamic environments”, Prague ,Czech Republic, 2004(5).

[25] Pålsson J. M., (2008), “The Camera Placement Problem--An art gallery problem variation”, Master Thesis, Lund University.

[26] Web 6, (2016), https://en.wikipedia.org/wiki/Rotation_matrix, (Erişim Tarihi: 2016/6/15).

[27] Sclaroff U. M., (2004), “Optimal placement of cameras in floorplans to satisfy task requirements and cost constraints”, Prague/ Czech Republic, May.

[28] Shi Y., (1998), “A modified particle swarm optimizer”, IEEE World Congress on Computational Intelligence, Alaska/USA, May.

[29] Sinha S., (2006), “GPU-based video feature tracking and matching”, Workshop on Edge Computing Using New Commodity Architectures, Chapel Hill/North Carolina/USA, May.

[30] Snyman J. A., (2005), “Practical Mathematical Optimization”, 1 st Edition, Springer.

[31] Web 7, (2016), https://en.wikipedia.org/wiki/Spherical_coordinate_system, (Erişim Tarihi: 15/06/2016).

[32] Web 8, (2016), http://geomalgorithms.com/a06-_intersect-2.html, (Erişim Tarihi: 15/06/2016).

[33] Zhao J., (2009), ”Optimal visual sensor planning”, IEEE International Symposium on Circuits and Systems, 165-168, Taipei, Taiwan, May.

[34] Zhu G., (2010), ”Gbest-guided artificial bee colony algorithm for numerical function optimization”, Applied Mathematics and Computation, 217(7), 3166-3173.

BIOGRAPHY

Mehmet Arif Şekercioğlu was born in 1986 in Afyon, Turkey. He graduated from Middle East Technical University Mathematics Department at 2010. He has been a M.S. student and a research assistant at Computer Engineering Department of Gebze Technical University Graduate School of Natural and Applied Sciences since 2013. His research interests include computer vision and optimization.