GPU-Based Occlusion Minimisation for Optimal Placement of Multiple 3D Cameras

Joacim Dybedal* and Geir Hovland*

*University of Agder
Faculty of Engineering and Science
Jon Lilletunsvei 9, 4879 Grimstad, Norway

Abstract **This paper presents a fast GPU-based solution to the 3D occlusion detection problem and the 3D camera placement optimisation problem. Occlusion detection is incorporated into the optimisation problem to return near-optimal positions for 3D cameras in environments containing occluding objects, which maximises the volume that is visible to the cameras. In addition, the authors' previous work on 3D sensor placement optimisation is extended to include a model for a pyramid-shaped viewing frustum and to take the camera's pose into account when computing the optimal position.**

C.1 Introduction

Depth cameras and other point cloud-generating 3D sensors are becoming increasingly important in many industries, particularly in intelligent autonomous systems. The automotive vehicle industry have historically been leading the development, see for example [C1, C2, C3]. Examples of sensors used are lidars, radars and 2D cameras, and the use of such sensors are currently expanding into other industries as well. The papers [C4] and [C5] present examples where the intended application domain was in offshore drilling.

Occlusion detection in three dimensions is important in multiple research fields. Most work on this topic is related to the tracking and classification of partially of fully occluded objects, or removing occluded areas when rendering 3D images. For example, in [C6], occlusion detection techniques for face recognition are discussed. A KD-tree based occlusion detection algorithm for hologram displays was developed in [C7]. For the 3D optimal sensor placement optimisation problem, occlusion detection is needed in order to place the sensors in such a way that as much as possible of the volume is visible to the cameras, despite the presence of occluding objects.

Previous work by the authors, [C4] and [C5], has aimed to solve the 3D optimal sensor placement problem. The problem has proven to be challenging, and most of the previous work in the open literature is limited to solving the 2D problem.

Mixed-integer programming is one of the approaches successfully used in solving the 2D problem, and this method was used to solve the 3D problem in [C4]. However, it was shown that this approach did not scale well when high accuracy is wanted, due to the fact that nonlinear equations were linearised by introducing many new variables and constraints in the optimisation problem.

In [C5], the 3D problem was solved using a massively parallelised CUDA program and random sampling. The solution was limited to a generic cone-shaped viewing frustum, and only included optimisation of sensor position, not pose. This approach reduced the computation time required to find good solutions from hours to minutes compared to [C4]. Other previous work has attempted solving the 3D problem by using heuristic approaches. However, such approaches tend to end up in local minima. The authors' previous work on this topic contained several references of different methods used for both 2D and 3D sensor placement optimisation, see [C8], [C9], [C10], [C11] and [C12].

In [C13] an approach for sensor placement optimisation is presented which is termed minimax. Instead of a typical probabilistic framework for dealing with dynamic occlusion, a robust (minimax) approach was presented to optimise the worst-case scenario. The objects in the workspace is represented as multiple polyhedra, and the occlusion detection involves Boolean operation such as the set-difference ($\mathcal{B} \setminus \mathcal{A}$), resulting in an exponential time complexity when the number of polyhedra increases.

In this paper, the previous work by the authors is extended by considering occlusions in the workspace, a more realistic (pyramid shaped) viewing frustum of the sensors as well as pan, tilt and roll angles of the cameras.

C.2 Problem Definition

Building upon the authors' previous work in [C5], where a GPU-based optimisation approach was considered for a generic cone-shaped field of view, a new method for considering a camera's pyramid-shaped viewing frustum is developed. The CUDA-based optimisation solver is further evolved to not only optimise the sensor position, but also include camera pan, tilt and roll angles. In addition, obstacles which occlude the camera's view are also included, and a method for detecting occluded space in a camera's viewing frustum is developed.

A series of test-cases are evaluated, where a limited volume in 3D space is considered. Six camera sensors are used for surveillance of this volume. Previous work by the authors considered a conical field of view. However, cameras typically do not have a conical field of view, but are instead often characterised by the pyramid-shaped viewing frustum, see for example [C14]. The considered volume in this paper has a size of $10\,\mathrm{m} \times 10\,\mathrm{m} \times 4.5\,\mathrm{m}$ and it is divided into a grid of smaller cubes of size $0.25\,\mathrm{m} \times 0.25\,\mathrm{m} \times 0.25\,\mathrm{m}$ resulting in a total of 28 800 cubes.

The optimisation problem is defined as follows: Maximise the number of cubes seen by the cameras subject to the following constraints:

- The camera field of view is specified by a viewing frustum including a minimum and maximum distance.
- The Z-position (height) of the cameras is fixed at a certain height, e.g. 4.5 m.
- The camera location is along a wall, meaning that either the X- or Y- direction is a free variable.
- The camera pan angle is a free variable (tilt and roll are fixed in these tests).
- Occluding obstacles represented by cubes are present in the considered volume.

To solve this expanded problem, a method to detect if an object is inside a pyramid shaped viewing frustum is needed and a check for occlusion must be implemented. The optimisation solver must then be expanded to include more free variables.

C.3 Methodology

As mentioned, previous work by the authors include a GPU-based optimisation solver for 3D sensor placement [C5]. This approach divided a given volume into a grid of cubes at a specified resolution. The optimisation solver was developed in CUDA, where a corresponding grid of threads was implemented. As seen in Fig. C.1, each cube was assigned a CUDA thread which determined if the cube was inside or outside the visible area of one or more sensors. This was done by calculating the angle formed between the sensor's direction vector and the vector pointing from the sensor to the cube, and comparing this to the angle of the cone-shaped field of view. In addition, the range of the sensor was considered. The process was then repeated thousands of times with different (random) values for the sensor positions, limited by a set of constraints. The positions that yielded the most visible cubes were then saved as the optimal solution. When multiple sensors were considered, one CUDA grid for each sensor was stacked on top of each other and the number of sensors viewing a single cube was counted. This allowed for an additional redundancy constraint, assuring that cubes were viewed by at least n sensors, where n was specified for each cube.

In this paper, the method described above was extended with multiple new features, which will be described in the following sections.

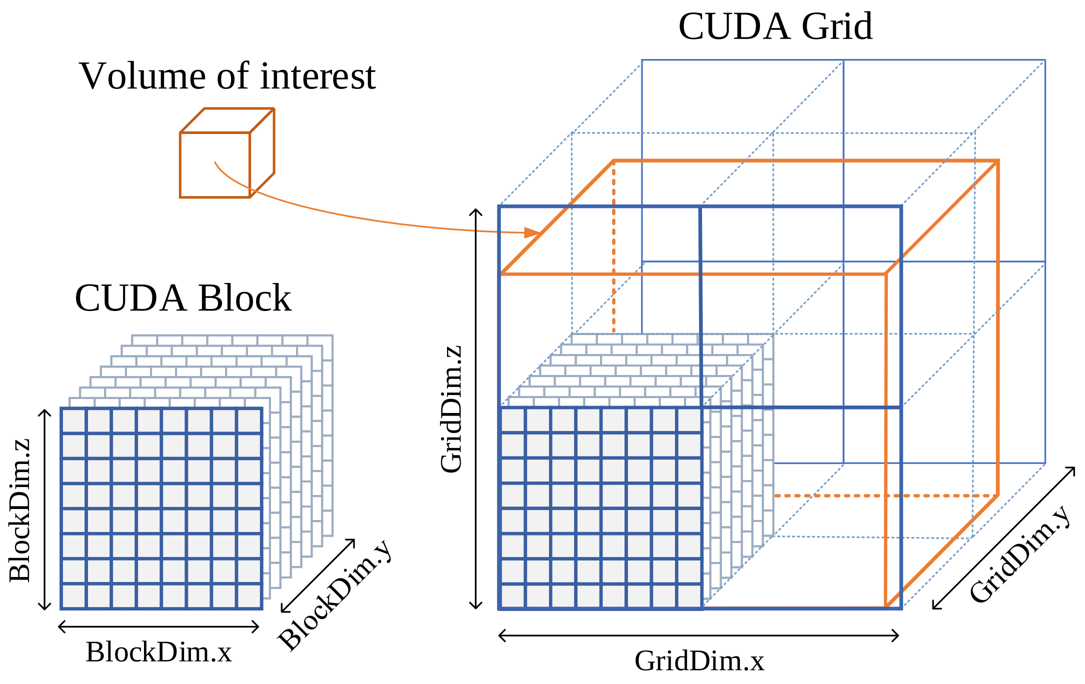


Figure C.1: Layout of CUDA threads compared to the volume of interest.

C.3.1 Occlusion Detection

First, a method for detecting occluded space was developed. Initially, before the optimisation procedure starts, a set of cubes in the volume is labelled as "possibly occluding", meaning these cubes will create a shadow inhibiting cubes behind them to be viewed by a camera. In a real scenario, these could be structures such as walls, machinery, pipes, etc. When running the optimisation solver and a cube is determined to be inside the camera's viewing frustum, a new set of CUDA threads are forked from the original thread. These new threads now correspond to all the cubes marked as "possibly occluding", and each of them will determine if they are occluding (i.e. casting a shadow over) the cube inside the viewing frustum. If one or more of the threads report that they are occluding, the original thread will mark the cube in question as not viewed by the camera.

The variables used in the occlusion detection are illustrated in Fig. C.2 and the algorithm for occlusion detection is described in Algorithm 1, and is computed by each possibly occluding cube for each cube determined to be inside the camera's viewing frustum.

The check $\alpha < 0$ determines if the point O is behind the point P relative to the sensor position S. If this is true, the cube at O can not be occluding. Next, for the cube at O to be occluding, the distance d must be less than half the diagonal of the

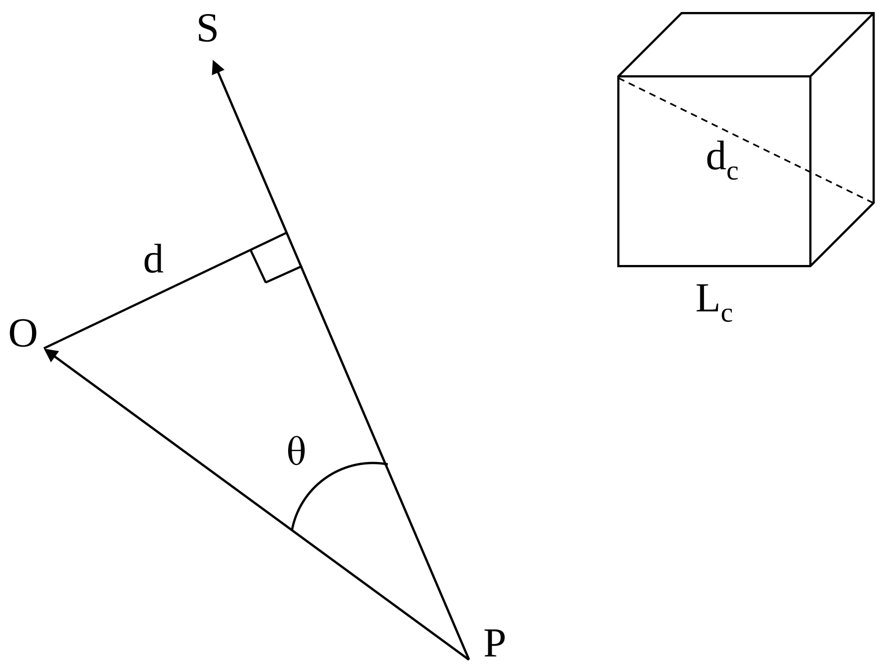


Figure C.2: Definition of variables used for occlusion detection. S is the camera location, O is the obstacle (possibly occluding cube) and P is the point to check for occlusion (a cube inside the camera's viewing frustum). d is the distance from the point O to the vector PS. L_c is the side length of a cube, and d_c is the diagonal of a cube.

cube, d_c, where d is defined as

$$\begin{aligned} d &= |PO| \sin(\arccos(\alpha)) \\ &= |PO|\sqrt{1-\alpha^2}. \end{aligned} \tag{C.1}$$

By testing against the diagonal of the cube, the worst case scenario is always considered, as the required distance will vary between $L_c/2$ and $d_c/2$ depending on the orientation of the cube relative to the vector PS. Hence, for this check, the cube is considered as a sphere with diameter d_c.

Algorithm 1 Occlusion Check

```
1: Find vectors $PS$ and $PO$
2: Calculate $\alpha = \cos\theta$
3: if $\alpha < 0$ then
4:     $P \leftarrow$ not occluded by $O$
5: end if
6: Calculate the distance $d$
7: if $d < \frac{1}{2}d_c$ then
8:     $P \leftarrow$ occluded by $O$
9: end if
```

C.3.2 The Pyramid-shaped Viewing Frustum

The pyramid-shaped viewing frustum of a typical camera differs from the generic cone-shaped frustum by being defined by two angles instead of one, i.e. vertical and horizontal field of view. Fig. C.3 shows an example where a point P is located outside the viewing frustum of sensor S. Four unit vectors $v_1, \cdots, v_4$ are defined, pointing from point S to the four corners describing the sensor's viewing frustum. In addition, another unit vector v is defined pointing from S to P. A sufficient condition to check if the point P is located inside the viewing frustum is as follows:

$$c_1 = v_1 \times v_2 \tag{C.2}$$

$$c_2 = v_2 \times v_3 \tag{C.3}$$

$$c_3 = v_3 \times v_4 \tag{C.4}$$

$$c_4 = v_4 \times v_1 \tag{C.5}$$

$$d_i = c_i^T v > 0 \qquad \forall i \in \{1, \cdots, 4\} \tag{C.6}$$

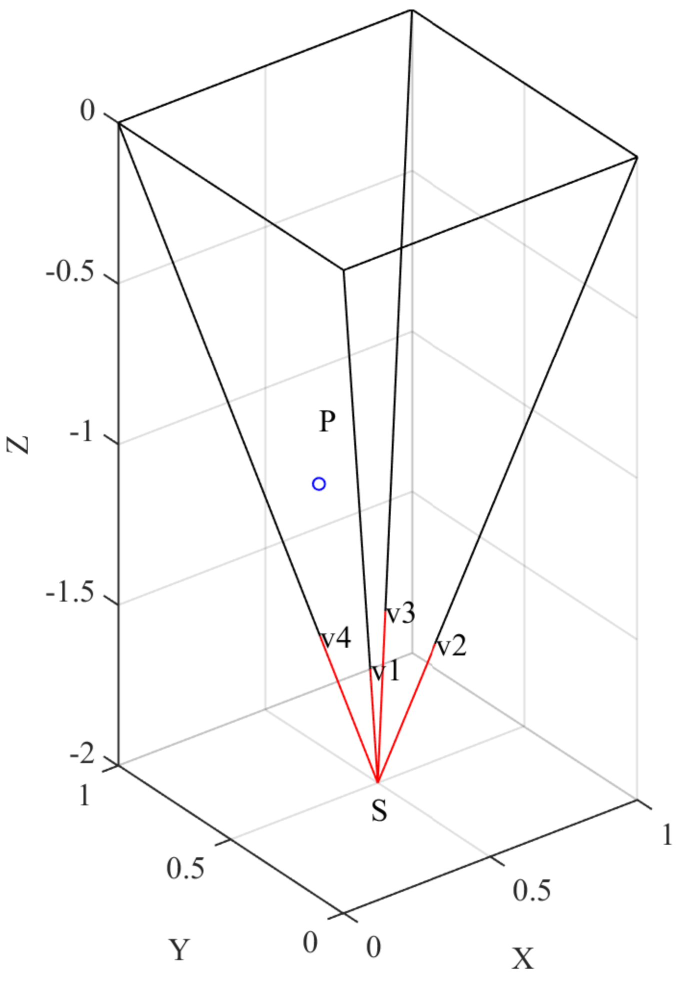


Figure C.3: Four vectors v_1 to v_4 used to determine if a point P is located inside a sensor S pyramid-shaped field-of-view.

In other words, if all the four inner products between the vectors c_i and v are positive, then the point P is inside the viewing frustum. A positive inner product means that the point P lies on the inner side of the plane spanned by the two vectors v_i and v_j. Finally, the length of the vector SP is checked against the minimum and maximum range of the sensor.

To construct the vectors v_i, five different rotation matrices $\mathbf{R}_s$ and $\mathbf{R}_i$ are created, where $\mathbf{R}_s$ describes the transformation from the global coordinate system to the sensor's coordinate system, and $\mathbf{R}_i$ describe the transformations from the sensor's direction vector to the four vectors v_i. The local coordinate system of the sensor is defined as a right-hand coordinate system where the sensor direction is a unit vector along the Z axis.

The sensor configuration related to the global coordinate system is described by the three Euler angles pan, tilt and roll, corresponding to rotating around the sensor's Z, Y and X axes, in that order. Using this Z-Y-X Euler rotation convention, the matrix $\mathbf{R}_s$ is generated based on these angles.

The sensor's pyramid-shaped viewing frustum is described by the vertical and horizontal field of view angles (FOV_v and FOV_h), corresponding to a rotation around the sensor's Y and X axes, respectively. The four matrices $\mathbf{R}_i$ can then be generated by the four possible combinations of a rotation of $\pm\mathrm{FOV}_v$ around the Y axis and $\pm\mathrm{FOV}_h$ around the X axis.

With the sensor's direction vector defined as $v_d = [0\ 0\ 1]^T$, then the vectors describing the sensor direction and field of view are

$$v = \mathbf{R}_s v_d \tag{C.7}$$

$$v_i = \mathbf{R}_s \mathbf{R}_i v_d \quad \forall i \in \{1, \cdots, 4\}. \tag{C.8}$$

C.3.3 Extension of the Sensor Placement Optimisation Solver

The optimisation solver outlined at the start of Section C.3 has been extended in this paper. While only the sensor positions were considered in the previous algorithm, the extended version includes sensor pan, tilt and roll angles, as well as occlusion detection. A simplified algorithm of the process for a single sensor is shown in Algorithm 2. Note that the **for** loops in line 4 and 7 are not executed sequentially, but in parallel on the GPU processor.

All extrinsic sensor configuration values can be selected as free variables in the optimisation problem, e.g. the X, Y, Z location in the global coordinate system and the pan, tilt and roll angles of each sensor. The variables must be constrained by a lower and upper bound, and in each iteration, the values are randomly assigned a value within these limits. This kind of random sampling ensures that the solver

Algorithm 2 Extended Optimization Solver

```
for $N$ number of iterations do
    Generate new values for free variables
    Generate $v$ and $v_i$ using Eq. C.7 and C.8.
    for each cube do
        Launch CUDA thread
        Check if cube is inside sensor's viewing
        frustum using Eq. C.6
        for all possibly occluding cubes do
            Check if the cube is occluding its parent
            cube according to Alg. 1
        end for
    end for
    Count the number of visible cubes
end for
Save the values of the set of free variables which yielded the maximum number
of visible cubes inside the sensor's viewing frustum.
```

does not end up in a local minimum, but it requires that enough samples are tested to reach a satisfying accuracy. However, as will be shown in the case studies, it is found that the solver quickly converges towards a near-optimal solution.

C.4 Case Studies

To verify the developed solution, three case studies were conducted.

1. Both the pyramid-shaped viewing frustum and the occlusion detection was tested on a simple case with two sensors looking at a beam in the middle of the volume. In this test, there is no sensor placement optimisation, thus only a single iteration of the program with no free variables was executed.

2. The extended sensor optimisation solver was tested using the new viewing frustum and including the sensor pan angle as a free variable in addition to sensor position, but without occlusion detection.

3. Last, Case 2 was repeated, but with occlusion detection active. This test was performed to gauge the increased computational complexity compared to Case 2.

The computation time measured in the results is the run-time of the entire program including pre- and post data reads and writes. To ensure ease of use, the CUDA solver was implemented as a library which is called by a Python script. Using Python, all required setup variables, occluding objects and constraints can easily be changed to test different scenarios. After the program launch, the results are saved in a single file using the JSON format for easy inclusion in other programs, e.g. Matlab.

C.4.1 Case Study 1

In the first case, the developed occlusion test and pyramid-shaped viewing frustum was evaluated. The sensors were specified with both vertical and horizontal field of view of 22.5°, a minimum range of 0.5 m and a maximum range of 10 m. The setup used for this case is shown in Table C.1.

Table C.1: Setup for Case Study 1

Number of sensors	2
Free variables	None
Room dimension	10 m × 10 m × 10 m
Cuboid size	0.25 m × 0.25 m × 0.25 m
Occlusion detection	On
Number of iterations	1

In addition, a "beam" of possibly occluding cubes were inserted in the middle of the room, spanning the length along the X axis. The result of the test can be seen in Fig. C.4 and Table C.2. As seen from the figure both the occlusion detection and the pyramid-shaped viewing frustum worked as expected.

Table C.2: Results for Case Study 1

Sensor 1 (X, Y, Z, pan, tilt, roll)	2.5, 10, 10, −90°, 135°, 0°
Sensor 2 (X, Y, Z, pan, tilt, roll)	7.5, 0, 10, 90°, 135°, 0°
Computation time	0.536 s
GPU Utilisation	N/A (Not measurable)
No. of cubes covered	4104

C.4.2 Case Study 2

In the second case, the extended optimisation solver was evaluated. Here, a simple model of the Industrial Robotics Lab at the University of Agder was used (see for example Figure 8 in [C15] for more information). The sensors were specified as Microsoft Kinect V2s, with a vertical field of view of 60° and a horizontal field of view of 70.2°. The minimal and maximal range was set to 1.0 m and 8.0 m respectively. Occlusion detection was not activated. The setup used for this case is shown in Table C.3.

The result of the test can be seen in Table C.4. The best result was found in iteration no. 3666. As seen from Fig. C.5 the sensors were placed to cover as many

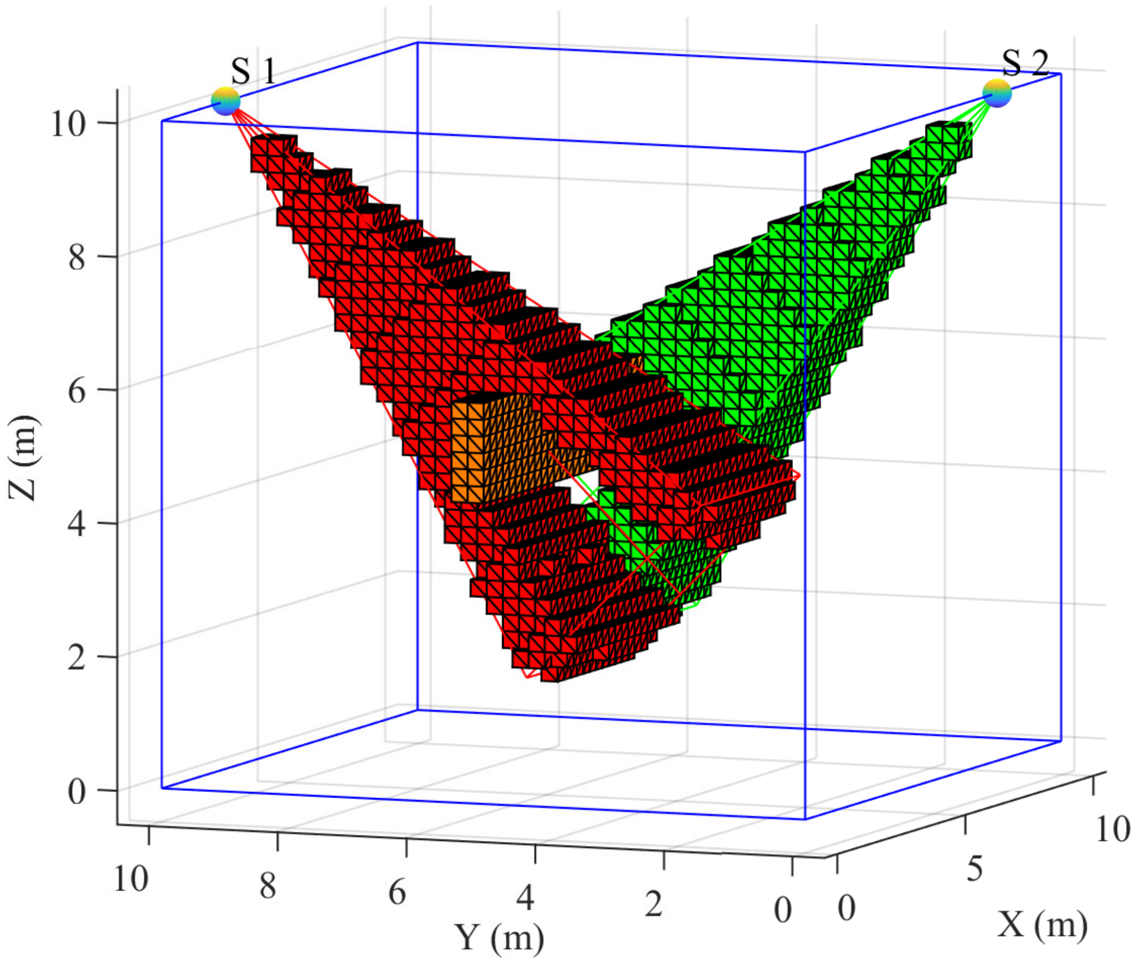


Figure C.4: Result of Case Study 1, with two sensors, one on each side of the room, where a beam is inserted in the middle of the room.

cubes as possible. The optimised pan angles are close to the middle of the constraint ranges.

C.4.3 Case Study 3

In the third case, Case 2 was repeated, but with occlusion detection active. The setup used for this case is shown in Table C.5.

In addition, four occluding objects were inserted. Two representing two robots and their track motion, and two representing persons. These can be seen in Fig. C.6a. The best result was found in iteration no. 1764. The result of the test can be seen in Fig. C.6b and Table C.6. As can be seen by the figure, sensors 3 and 4 are now placed lower on the Y axis, as well as panned to view as many cubes as possible behind the obstacles.

Table C.3: Setup for Case Study 2

Number of sensors	6
Free variables	Y coordinate and pan for all sensors
Position Constraints S1 and S2	$0 < Y < 4$m
Position Constraints S3 and S4	$2 < Y < 8$m
Position Constraints S5 and S6	$6 < Y < 10$m
Pan Constraints	S1: 135°±45°, S2: 45°±45°
	S3: 180°±45°, S4: 0°±45°
	S5: −135°±45°, S6: −45°±45°
Room dimension	10 m × 10 m × 4.5 m
Cuboid size	0.25 m × 0.25 m × 0.25 m
Occlusion detection	Off
Number of iterations	5000

Table C.4: Results for Case Study 2

Sensor 1 (X, Y^*, Z, pan*, tilt, roll)	10, 0.34, 4.5, 161.5°, 150°, 0°
Sensor 2 (X, Y^*, Z, pan*, tilt, roll)	0, 1.7, 4.5, 16.5°, 150°, 0°
Sensor 3 (X, Y^*, Z, pan*, tilt, roll)	10, 5.39, 4.5, −179.8°, 150°, 0°
Sensor 4 (X, Y^*, Z, pan*, tilt, roll)	0, 4.75, 4.5, 1.4°, 150°, 0°
Sensor 5 (X, Y^*, Z, pan*, tilt, roll)	10, 9.56, 4.5, −136.2°, 150°, 0°
Sensor 6 (X, Y^*, Z, pan*, tilt, roll)	0, 8.5, 4.5, −21.9°, 150°, 0°
Computation time	246.1 s
GPU Utilisation	100 %
No. of cubes covered	18152

* Optimised

Table C.5: Setup for Case Study 3

Number of sensors	6
Free variables	Y position and pan for all sensors
Position Constraints S1 and S2	$0 < Y < 4$m
Position Constraints S3 and S4	$2 < Y < 8$m
Position Constraints S5 and S6	$6 < Y < 10$m
Pan Constraints	S1: 135°±45°, S2: 45°±45°
	S3: 180°±45°, S4: 0°±45°
	S5: −135°±45°, S6: −45°±45°
Room dimension	10 m × 10 m × 4.5 m
Cuboid size	0.25 m × 0.25 m × 0.25 m
Occlusion detection	On
Number of iterations	5000

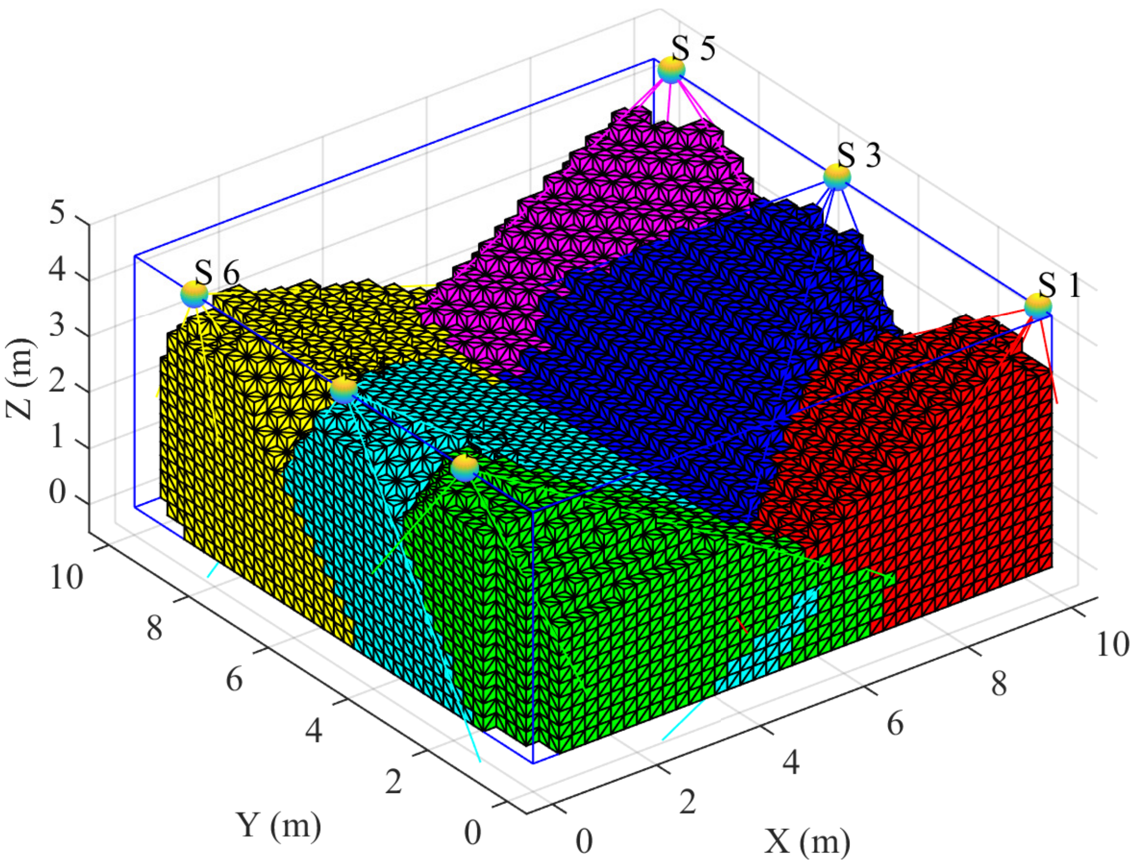
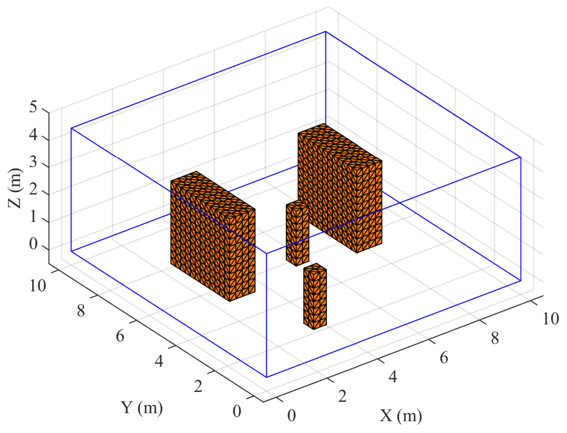
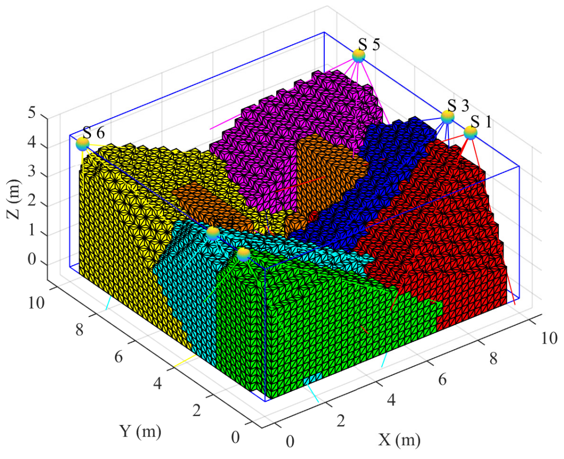


Figure C.5: Result of Case Study 2, with 6 sensors and no occluding objects.





(a) **(b)**

Figure C.6: (a) Obstacles created by inserting possibly occluding cubes. (b) Result of Case Study 3, with 6 sensors and occluding objects.

Table C.6: Results for Case Study 3

Sensor 1 (X, Y^*, Z, pan*, tilt, roll)	10, 2.5, 4.5, 169.6°, 150°, 0°
Sensor 2 (X, Y^*, Z, pan*, tilt, roll)	0, 1.1, 4.5, 19.8°, 150°, 0°
Sensor 3 (X, Y^*, Z, pan*, tilt, roll)	10, 3.67, 4.5, 147.1°, 150°, 0°
Sensor 4 (X, Y^*, Z, pan*, tilt, roll)	0, 2.67, 4.5, 31.1°, 150°, 0°
Sensor 5 (X, Y^*, Z, pan*, tilt, roll)	10, 8.2, 4.5, −169.8°, 150°, 0°
Sensor 6 (X, Y^*, Z, pan*, tilt, roll)	0, 9.3, 4.5, −35.6°, 150°, 0°
Computation time	1688.3 s
GPU Utilisation	100 %
No. of cubes covered	15862

* Optimised

C.5 Discussion and Conclusions

As can be seen by the results of the three case studies, the developed solutions for both the pyramid-shaped viewing frustum and the occlusion detection work as expected. When extending the sensor placement optimiser with these new features in addition to more free variables, the calculated optimal sensor positions and poses quickly converge to a good result. In Case Study 2, the best result was found in iteration 3666 out of 5000, and in Case Study 3, the best result was found after only 1764 iterations. Including the occlusion detection in Case Study 3, the sensor positions and pan angles are changed as expected to cover as many cubes as possible.

When comparing Case Study 2 and 3, the computation time was only increased by a factor of 6.86, even though the problem is exponentially more complex. When running n CUDA threads in Case Study 2, n^2 threads are launched in a worst case scenario when the occlusion detection is active (although only the threads corresponding to possibly occluding cubes will do actual work). This result implies that the GPU scheduler is able to utilise the GPU multiprocessors efficiently and that the number of cubes used in the tests (28 800) are not enough to completely exhaust the GPU resources. However, performing a full analysis and optimisation of the GPU utilisation is left for future work.

In [C13], the minimax solver was limited by a slow occlusion detector. Using the approach found in this paper could be a good alternative to the time-consuming set-difference operation ($\mathcal{B} \setminus \mathcal{A}$) on multiple polyhedra. The occlusion check requires only a single iteration of the presented solution.

Even though the optimiser seems to quickly converge to a good result, there is no guarantee that the optimal result will be found without running an infinite number of iterations. This is due to the fact that the solver is based on random sampling. Enabling more free variables such as pan, tilt, and roll, also decreases the chances of quickly reaching a good result. On the other hand, many algorithms in machine learning applications are based on random sampling.

In future work, the CUDA-based optimiser could be used to quickly find a set of good results, which could then be further refined using e.g. a gradient search based approach. Additional future work includes limiting the solver to only optimise coverage in confined areas inside the overall volume, e.g. minimise occlusion in areas which are more important. The developed solution in [C5] also contains a redundancy constraint which was not active during the test cases in this paper. In addition, a method for using the optimisation solver in rooms with arbitrary (not square) layouts should be developed.

C.6 Acknowledgement

The research presented in this paper has received funding from the Norwegian Research Council, SFI Offshore Mechatronics, project number 237896, https://sfi.mechatronics.no

References – Paper C

[C1] D. Göhring, M. Wang, M. Schnürmacher, and T. Ganjineh. Radar/lidar sensor fusion for car-following on highways. In *Proc. Intl. Conf. Automation, Robotics and Applications (ICARA)*, 2011.

[C2] Y. Alkhorshid, K. Aryafar, S. Bauer, and G. Wanielik. Road detection through supervised classification. In *Proc. Intl. Conf. Machine Learning and Applications (ICMLA)*, 2016.

[C3] E. Ward and J. Folkesson. Vehicle localization with low cost radar sensors. In *Proc. IEEE Intelligent Vehicles Symposium*, 2016.

[C4] Joacim Dybedal and Geir Hovland. Optimal placement of 3d sensors considering range and field of view. In *Proc. IEEE Intl. Conf. on Advanced Intelligent Mechatronics (AIM)*, 2017.

[C5] Joacim Dybedal and Geir Hovland. Gpu-based optimisation of 3d sensor placement considering redundancy, range and field of view. In *Proc. 15th IEEE Conf. on Industrial Electronics and Applications (ICIEA)*, 2020. in press.

[C6] Nicole Dagnes, Enrico Vezzetti, Federica Marcolin, and Stefano Tornincasa. Occlusion detection and restoration techniques for 3d face recognition: a literature review. *Machine Vision and Applications*, 29(5):789–813, jul 2018.

[C7] Shuangting Liu, Hui Wei, Ni Li, Zihan Liu, and Jiaqi Zhang. Occlusion calculation algorithm for computer generated hologram based on ray tracing. *Optics Communications*, 443:76 – 85, 2019.

[C8] Shenyu Mou, Yan Chang, Wenshuo Wang, and Ding Zhao. An optimal lidar configuration approach for self-driving cars. *CoRR*, abs/1805.07843, 2018.

[C9] Zuxin Liu, Mansur Arief, and Ding Zhao. Where should we place lidars on the autonomous vehicle? - an optimal design approach. In *Proc. 2019 Intl. Conf. on Robotics and Automation (ICRA)*, 2019.

[C10] Nicolaj Kirchhof. Optimal placement of multiple sensors for localization applications. In *Proc. IEEE Intl. Conf. on Indoor Positioning and Indoor Navigation*, 2013.

[C11] H. Topcuoglu, M. Ermis, I. Bekmezci, and M. Sifyan. A new three-dimensional wireless multimedia sensor network simulation environment for connected coverage problems. *Simulation: Trans. Society for Modeling and Sim. Intl.*., 88(1):110–122, 2006.

[C12] P. Rahimian and J.K. Kearney. Optimal camera placement for motion capture systems. *IEEE Trans. on Visualization and Computer Graphics*, 23(3):1209–1221, March 2017.

[C13] Rishi Mohan and Bram De Jager. Robust optimal sensor planning for occlusion handling in dynamic robotic environments. *IEEE Sensors Journal*, 19(11):4259–4270, jun 2019.

[C14] Kelvin Sung, Peter Shirley, and Steven Baer. Essentials of interactive computer graphics: Concepts and implementation. *CRC Press*, page 390, 2008.

[C15] Joacim Dybedal, Atle Aalerud, and Geir Hovland. Embedded processing and compression of 3d sensor data for large scale industrial environments. *Sensors*, 19(3), 2019.