Master Thesis

Detection Sensor Placement Algorithm for Protection Against Attacks Using Drones

Bc. Dominik Hoftych
Supervisor: Ing. Milan Rollo, PhD.

August 2020

Department of Computer Science
Faculty of Electrical Engineering
Czech Technical University in Prague

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení:	**Hoftych**	Jméno: **Dominik**	Osobní číslo: **456994**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Algoritmus pro rozmísťování detekčních senzorů pro ochranu před útoky s využitím bezpilotních prostředků

Název diplomové práce anglicky:

Detection Sensor Placement Algorithm for Protection Against Attacks Using Drones

Pokyny pro vypracování:

1. Seznamte se s problematikou detekce bezpilotních prostředků různými typy senzorů
2. Prostudujte existující metody rozmísťování senzorů pro pokrytí 3D prostředí
3. Formalizujte úlohu detekce bezpilotních prostředků a definujte optimalizační kritéria v závislosti na prostředí a typu útoku
4. Navrhněte a implementujte algoritmus pro volbu typu a rozmístění senzorů pro zadané prostředí
5. Experimentálně ověřte vlastnosti algoritmu prostřednictvím softwarové simulace

Seznam doporučené literatury:

[1] Á. D. De Quevedo, F. I. Urzaiz, J. G. Menoyo and A. A. López: Drone Detection and RCS Measurements with Ubiquitous Radar. 2018 International Conference on Radar (RADAR), Brisbane, QLD, 2018, pp. 1-6.
[2] Gabriel C. Birch, John C. Griffin, and Matthew K. Erdman: UAS Detection, Classification, and
Neutralization: Market Survey 2015, Sandia National Laboratories, 2015.
[3] Kúdelka J.: Critical Infrastructure Protection Against Attacks Using Drones. Bakalářská práce, ČVUT v Praze, 2018.
[4] I. Guvenc, F. Koohifar, S. Singh, M. L. Sichitiu and D. Matolak: Detection, Tracking, and Interdiction for Amateur Drones. In IEEE Communications Magazine, vol. 56, no. 4, pp. 75-81, April 2018.

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Milan Rollo, Ph.D., centrum umělé inteligence FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **11.02.2020** Termín odevzdání diplomové práce: **14.08.2020**

Platnost zadání diplomové práce: **30.09.2021**

Ing. Milan Rollo, Ph.D.	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D.
podpis vedoucí(ho) práce		podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.	
Datum převzetí zadání	Podpis studenta

Abstract

Nowadays, the use of drones is common for both recreational and commercial purposes. Drones can assist people in many activities including remote sensing, aerial photography and filming as well as medical supply delivery. However, their availability and potential for use also pose various threats that need to be protected against. The goal of this thesis is to propose an algorithm to solve the problem of optimal sensor placement around the monitored sector, for the purpose of detection of possibly dangerous drones. The algorithm assumes a realistic 3D environment and deals with camera occlusion as well. It also offers a high level of user parameterization that involves the priority areas, maximal cost of the sensor network, sensor prices, or multiple coverage. A part of the application is also a graphical user interface (GUI) displaying the monitored sector, priority areas, and the computed sensor network. The first part of the thesis describes today's commercial use of drones, the threats posed by the drones, existing systems for drone detection and neutralization, and the recorded drone incidents. It further discusses possible approaches and solutions to the problem of optimal sensor placement. The second part of the thesis devotes to the description of the application, the introduction of individual components and configuration parameters, and a detailed description of the proposed algorithm. The end of the thesis focuses on evaluating the proposed algorithm using experiments that simulate scenarios in which the protection against drones would be necessary.

Keywords: drone detection, drone neutralization, optimal sensor placement problem, sensor network

Abstrakt

V dnešní době je použití dronů oblíbené nejen pro rekreační, ale i komerční účely. Drony mohou asistovat lidem při mnoha činnostech, zahrnující například dálkový sběr informací, letecké snímání a filmování, ale i dodávku zdravotního materiálu. Nicméně, jejich dostupnost a potenciál využití představuje i různá nebezpečí, proti kterým je potřeba se umět ochránit. Tato práce se zabývá návrhem algoritmu pro optimální rozmísťování kamer a akustických senzorů v oblasti monitorovaného sektoru, za účelem detekce potenciálně nebezpečných dronů, pracujícího v reálném 3D prostředí včetně okluze kamery. Algoritmus umožňuje vysokou míru uživatelské parametrizace, zahrnující prioritní oblasti, maximální cenu senzorové sítě, ceny senzorů nebo vícenásobné pokrytí. Součástí aplikace je také grafické rozhraní, které zobrazuje monitorovaný sektor, prioritní oblasti a senzorovou síť. První část práce popisuje komerční využití dronů v dnešní době a nebezpečí, které představují, existující systémy pro detekci a neutralizaci dronů a incidenty zaznamenané v minulých letech. Dále popisuje možné řešení a přístupy k problému optimálního rozmísťování senzorů. Druhá část práce se zabývá popisem aplikace, představením jednotlivých komponent a konfiguračních parametrů a detailním popisem navrhovaného algoritmu. Závěr práce se soustředí na vyhodnocení navrhovaného algoritmu pomocí experimentů simulujících scénáře, ve kterých by ochrana proti dronům byla nutná.

Klíčová slova: detekce dronů, neutralizace dronů, problém optimálního rozmísťování senzorů, senzorová síť

Author statement for graduate thesis:

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, date ....

signature

Acknowledgements

At first, I would like to express my gratitude to my supervisor Milan Rollo for being helpful and keeping me on the right course during this thesis. I would also like to thank Vojtěch Kaiser for providing useful advice and explanations on the area of computer graphics and the simulation framework itself. In the last place, my sincere thanks go to my family and my girlfriend Kristýna for their patience and never-ending support during the whole study.

Contents

List of Figures

Chapter 1

Introduction

In today's world, unmanned aerial vehicles (UAV) or unmanned aerial systems (UAS), also referred to as drones, are becoming more and more common. People are able to buy drones on the internet and have them delivered by two days, just like any other product. That arouses a strong interest not only in recreational use but also in commercial use where drones may assist people with various tasks, including aerial photography and filming, monitoring objects, or even humanitarian aid.

Although drones offer a range of beneficial features, with such availability they can easily be misused for illegal actions, causing various types of threats such as unauthorized intrusion, spying (privacy violations), or a terrorist attack in the worst case. With the technology evolving, it is expected that the use of drones will only increase worldwide, forcing people and organizations to develop detection systems that will help with protecting against drones, maintaining security, and being prepared to introduce appropriate defensive countermeasures in time.

In this thesis, we propose an approach to the problem of optimal sensor placement, used for protection against drone attacks and threats generally. We focus on implementing a sensor placement algorithm, whose result is a sensor network of cameras and acoustic sensors around the monitored sector that is supposed to detect intruder drones in the monitored area. The algorithm works in real-world space, in three dimensions, and can be parameterized by several criteria such as the priority areas, the maximum price of the sensor network or multiple coverage. Besides of the algorithm, a simulation offering graphical user interface (GUI) showing the monitored sector and deployed sensor network is implemented using the AgentFly simulation framework. Finally, the sensor network computed by the sensor placement algorithm is thoroughly evaluated using an experimental evaluation, in which different types of flights are simulated for various purposes, such as smuggling, spying, or a terrorist attack.

1.1 Thesis structure

In Chapter 2, we familiarize the reader with today's usage of drones, the benefits they offer, and the threats they pose. We include a brief list of recorded drone incidents to emphasize why protection against drones is needed. Finally, we present a few state-of-the-art counter-drone systems, their features, and vendors.

Chapter 3 provides a survey of approaches to solving the problem of optimal sensor placement.

In Chapter 4, we present the design of the whole simulation. We describe the workflow and individual components of the simulation. We also focus on a detailed description of configuration properties that can be used to control the behavior of the sensor placement algorithm and the subsequent experimental evaluation.

In Chapter 5, we introduce our approach to the problem of optimal sensor placement and formalize the proposed algorithm. Prior to that, we describe important supportive features and how we dealt with the precise computation of the sensor's coverage.

In Chapter 6, we introduce the experimental evaluation that we used to evaluate the proposed sensor placement algorithm. We present the scenarios and discuss the results and consequences of the experimental evaluation.

Finally, Chapter 7 concludes the thesis and discusses possible extensions and improvements.

Chapter 2

Background

In this chapter, we present today's use of drones and the threats they pose with a brief overview of drone incidents recorded to date. Then, we present technologies that are used to protect against drones and a shortlist of state-of-the-art counter-drone systems.

2.1 Drone usage

As drone technology advances, drones offer many beneficial features that help commercial organizations efficiently deal with various problems and challenges that wouldn't be easy to accomplish otherwise. Besides recreational use of drones that was already mentioned in the introduction, typical examples of nowadays commercial use of drones are the following [1]:

- **Remote sensing.** "Remote sensing is the acquisition of information about an object or phenomenon without making physical contact with the object" [2]. A drone can be equipped with sensors of almost any type, which basically makes it a sensor that can be remotely operated. Such possibility finds use in the following areas:
  - **Biological sensors** - air-quality readings, detection of organic or micro-organics presence;
  - **Heat sensors** - livestock temperature, presence, and temperature of the water;
  - **Visual sensors** - inspection of agricultural fields, power lines, towers, roofs as well as monitoring infrastructures or buildings.
- **Humanitarian aid and disaster relief.** Drones can be used to assess the damage, locate victims, deliver aid after earthquakes or hurricanes. Drones are also able to monitor forest fires and detect abnormal forest temperatures.
- **Shipping and delivery**
- **Aerial photography and filming.** Drones equipped with high-quality cameras are used to take air pictures or videos from high places where people can't get.

2.2 Drone threats

The area where drones are making our everyday lives easier is really large. However, when their availability is combined with the number of possibilities they offer, it is rather hard to prevent them from being used for unwanted or illegal activities. Such unwanted or illegal activities range from less serious such as flying too high to serious threats such as terrorism, and might involve the following:

- **Violation of safety or airspace restrictions.** In order to fly the drone safely, the pilot must follow safety and airspace restrictions specified by the country in which the pilot operates. These restrictions might differ from country to country, however typically include the following [3, 4]:
  - **Flying too high.** In the case of recreational use, the maximum allowed height is usually 400 feet.
  - **Flying near other aircraft.** If the drone was sucked into an aircraft's engine or collision occurred, it could very easily end in tragedy.
  - **Flying beyond visual line of sight**. The pilot must maintain eye contact with the drone throughout the flight.
  - **Flying over specific areas.** Such areas include airports, stadiums, or sporting events.
- **Illegal surveillance.** Spying on people or objects without permission.
- **Illegal smuggling.** Drones can be used to smuggle illegal items such as weapons or drugs, for example across the country borders or into prisons. Speaking about prisons, other items such as cell phones or pornography materials are being smuggled as well.
- **Terrorist attack.** Dropping a bomb into a crowded place would be a common example of a terrorist attack using drones. Drones are still relatively small, the weight of load they can carry is limited and so is the damage potential. However, the damage potential of spraying a city from above with toxic or radioactive material is much bigger and could cause bigger problems. Another use of drones that is worth mentioning is the localization of enemies - using a drone to discover their positions.

2.2.1 Recorded drone incidents

There have been many drone incidents recorded worldwide to date, spreading out over the whole range of severity. Worth mentioning databases include regular UAS Sightings Report by Federal Aviation Administration [5] or database with 11340 drone incidents by Aviation Safety Network [6]. For drone incidents related to military areas, one must not forget the Drone Crash Database [7].

As the databases above provide only a brief report of the incident, we drew the examples of incidents from another publicly accessible database by Dedrone [8], from which we will now present a subset of noteworthy incidents that happened in Europe, organized by the industry in which they happened.

Airports and mid-air collisions

- On September 2, 2019, potentially lethal mid-air collision occurred at an altitude of 1700 feet when an illegal drone came too close to an Aer Lingus flight carrying 186 passengers. It is believed that this incident is the first case where the pilot was forced into action to avoid the drone [9].

- On May 17, 2019, a British Airways jet carrying up to 300 passengers was within 20 feet away from smashing into an illegally flown drone at an altitude of 6000 feet. This incident falls within the Category A - the most serious type of drone incident [10].
- On October 16, 2017, a rather big drone with a diameter of about 1 m nearly hit a jet that was just landing at the Gatwick airport, putting 130 lives at risk. The drone nearly made the plane's first officer disconnect the cockpit's autopilot system and take avoiding action [11].

Prisons

- On October 26, 2018, seven members of a drug-smuggling gang were jailed. In total, the gang performed 55 drone deliveries into prisons during a period of approximately 14 months. Deliveries mostly consisted of cannabis or synthetic cannabis, but included amphetamines, crack cocaine, and heroin as well and their combined price was estimated to 550 thousand pounds [12].
- On October 3, 2018, an incident that may almost be considered ridiculous happened, where two smugglers accidentally filmed themselves on the drone's camera when packing cannabis and pills into a drone. They were planning to fly the packaged drone into Perth Prison, but the drone crashed and was seized by the police. When police studied the footage from the drone's camera, they have found images of both the smugglers which led to successful identification and arrest of both [13].

Private/Non-Corporate

- On August 7, 2018, drone was flown over summer residence of the French President Emmanuel Macron's, where he and his wife were spending their holidays. The fact that this incident happened just one day after a drone explosion attack on the Venezuelan President Nicolás Maduro (who fortunately escaped safely) only aroused suspicion and fear of connection [14].

Stadiums

- On January 4, 2018, a drone caused an 11-minute delay during the football match between Crawley Town and Yeovil Town, teams of the lowest division of English Football League. When the drone was firstly spotted, the match was tied 1-1 and the referee had to take both teams off the field for safety reasons. The unexpected pause might have played its role in the outcome of the match, as soon after the match was continued, Crawley Town scored a goal and ended up winning the match 2-1 [15].

2.3 Drone detection

The danger that the inappropriate use of drones presents is large, and setting a proper defense is a hard task. With the speed drones can fly nowadays, there might be only several seconds since the drone is firstly seen until the attack. This short period of time must be efficiently used to select and perform appropriate actions to avoid the attack.

The first such action is to detect and categorize the drone. Is it even a drone? Is it a terrorist drone carrying a bomb? Is it a spying drone collecting some sensitive data? Or is it just an intruder drone that was flown the wrong direction?

In order to answer these questions, various types of sensors can be used, each having different pros and cons. In practice, more types of sensors at usually combined, forming a counter-UAV (C-UAV) system with better and more precise coverage as well as a lower false alarm rate than if the sensors were standalone.

The main types of sensors include radio frequency (RF) analyzers, radars, acoustic sensors, and optical sensors (cameras).

2.3.1 Radio frequency analyzers

The majority of commercial drones are not autonomous and thus require communication with their remote controller. During the communication, a lot of data is transmitted over specific frequencies of the radio frequency spectrum, which is done in a two-way fashion, i.e. drone sends telemetry data such as the battery life, position or altitude to the remote controller, and remote controller answers back with for example navigation commands.

By setting up a passive radio frequency sensor on selected frequencies, drone detection systems based on RF analyzers are able to capture the communication and relay it to a computer where specialized algorithms take over and compare it to a database of drone protocols. As different drones have different protocols, good RF-based detection systems are even able to identify the drone's model sometimes.

The advantages of RF-based drone detection systems are that they can detect multiple drones at once and the range of detection extends beyond 2 km, depending on the antennas. They can also be very cost-effective, although it obviously heavily depends on the specific manufacturer or model.

Disadvantages include mainly the fact that fully autonomous drones are undetectable as they don't need to communicate with their remote controller. Also, RF pollution might cause reduced effectiveness of RF-based detection systems [16].

An example of an RF sensor is shown in Figure 2.1.

Figure 2.1: RF-160 radio frequency sensor, Dedrone[1]

2.3.2 Radars

A radar system has a transmitter that emits radio waves which are called radio signals. When the radio signals encounter objects along their path, they are scattered, reflected, and bounce back to the radar receiver. Using specific algorithms, received waves are converted to visual and provide estimations of the shape, size, and density of the encountered objects.

Worth mentioning are also Pulse-Doppler radars, which are based on the Doppler effect[2] - distortion or bend in the wave. By emitting periodic bursts of radio waves and measuring the bends in the returned signal, estimations of distance and velocity of the encountered object can be calculated.

One of the main advantages of radars is that they can provide a long detection range. They can also provide constant coverage, drone tracking, and can detect multiple drones at once.

On the other hand, radars struggle with detecting drones of very small sizes. Moreover, drones usually tend to fly close to the ground which doesn't help either. In addition to that, radars often fail to distinguish between drones and flying birds, which is the main reason why radars are often supported by other types of sensors [16].

In Figure 2.2 we show a drone detection radar developed by Skylock anti-drone systems.

Figure 2.2: 360° drone detection radar, Skylock[3]

[1]image taken from https://www.dedrone.com/press/dedrone-introduces-next-generation-radio-frequency-sensor-rf-160-for-suas-detection-and-threat-mitigation

[2]https://en.wikipedia.org/wiki/Doppler_effect

[3]image taken from https://www.skylock1.com/drone-detection/

2.3.3 Acoustic sensors

Acoustic sensors usually consist of an array of microphones that detect the sound emitted by propellers and motors of the drone and use it to calculate the incoming direction [17].

Advantages include the ability to detect fully autonomous drones, easy installation, and low energy consumption.

Disadvantages include the need to constantly update the sound database and short detection range, usually about 300 to 500 m. Also, acoustic sensors don't work well in noisy environments.

Acoustic sensor Discovair G2 developed by Squarehead Technology is shown in Figure 2.3.

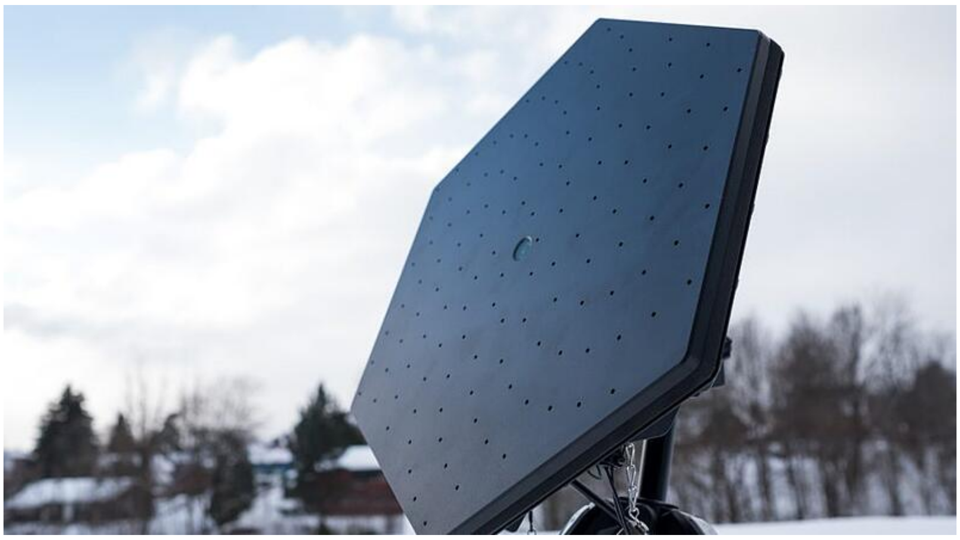

Figure 2.3: Discovair G2, Squarehead Technology[4]

2.3.4 Optical sensors

Optical sensors are essentially video cameras. Besides standard daylight cameras, optical sensors can be infrared or thermal imaging.

The main advantage is that unlike other sensor types, optical cameras can provide visual images of the drone and its payload (if there is any) that can later come useful as forensic evidence in eventual prosecution [17].

In terms of disadvantages, optical sensors perform poorly in bad weather conditions. Furthermore, they have high false alarm rates and struggle with detecting small drones.

Example of optical sensor is shown in Figure 2.4.

[4]image taken from `https://www.sqhead.com/squarehead-unveils-discovair-g2/`

Figure 2.4: PTZ camera, Flir[5]

2.4 Drone neutralization

Once the intruder drone is detected and identified as a threat, there are several options which countermeasures to perform, divided into two categories: *passive countermeasures* and *active countermeasures*.

2.4.1 Passive countermeasures

Passive countermeasures aim to protect against an intruder drone without physically interacting with it. Passive countermeasures are obviously limited, but to some extent extremely effective, reliable, and practical. Unlike active countermeasures, passive countermeasures are also completely legal.

A great example of an area where passive countermeasures have high efficiency is illegal surveillance. If there is a spying drone flying around a building, the simplest solution would be to for example closing window shades so that there is nothing to be seen for the drone. If the drone is spying over open space area, appropriate action would consist of taking people away from the open space, best to a building where the drone can't see them. Except for simply hiding from the drone, passive countermeasures also include deploying smoke clouds or shining a strong light towards the drone so that its camera is overpowered [18].

Another example of an attack that passive countermeasures are effective against is cybersecurity. Cybersecurity attacks might for example involve installing a device on the drone, such as an on-board computer (e.g. Raspberry Pi[6]) capable of hacking, or radio transceivers. Protection against such attack includes active monitoring of the network for intrusion or spoofing, seeking for anomalies in access logs, or for example turning off the guest wireless network [18].

[5]image taken from https://www.dedrone.com/products/hardware/extensions/ptz-cameras
[6]https://en.wikipedia.org/wiki/Raspberry_Pi

2.4.2 Active countermeasures

Active countermeasures simply mean to physically stop the drone. Unlike passive countermeasures, active countermeasures might not be perfectly legal in all cases, are often more expensive but also much more effective.

According to [19], active countermeasures are divided into two groups: *soft measures* and *hard measures*, which we will describe in detail in the following sections.

Soft measures. Soft measures mean to stop the drone without physical contact with the drone. A great example of soft measure is so-called "jamming", where radio frequency jammers are used. RF jammer is a device that transmits a large amount of RF energy towards the drone, masking the drone's signal which results in lost communication with the remote controller [17]. Based on a specific drone model, the lost communication causes the drone to either make a controlled landing in its current position or to return to its user-set home location, which often is its take-off location.

An example of an RF jammer called DroneDefender Counter-UAS Device is shown in Figure 2.5.

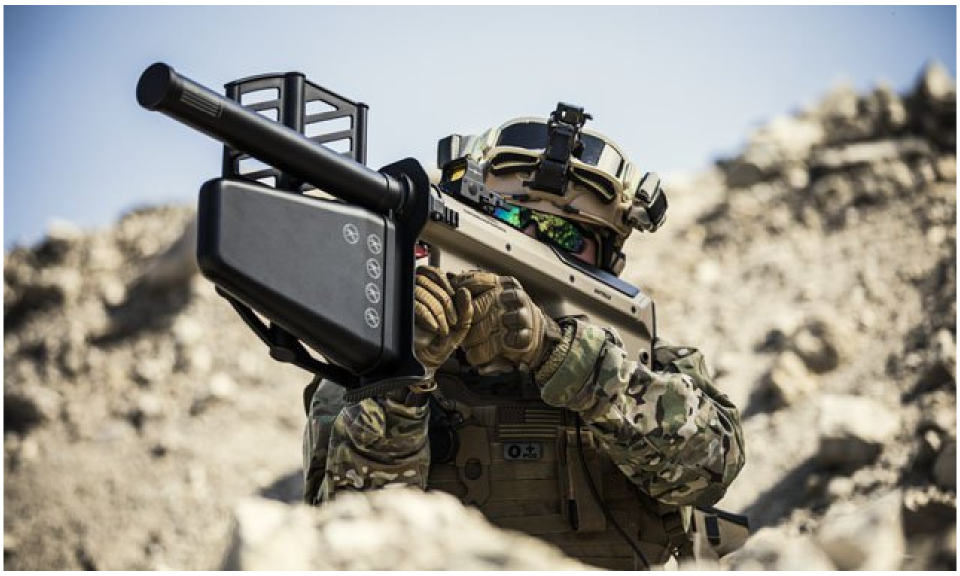

Figure 2.5: DroneDefender Counter-UAS Device, Battelle[7]

Another example of a soft measure is GPS spoofing (or Global Navigation Satellite System (GNSS) spoofing). GPS spoofing aims to deceive a GPS receiver, such as the drone's one, by broadcasting fake GPS signals from the ground. The signals might be fake but are technically sound and correct, so the receiver is not able to recognize that they are fake. Moreover, receivers are usually not able to detect the incoming direction of the signals, so they can't tell whether the signals come from the ground and not from the satellites, either. As a result, the GPS receivers start to unconsciously compute wrong coordinates, leading the drone to different courses and locations. Also, worth mentioning is the fact that GPS spoofing equipment is not expensive at all and as opposed to RF jammers, for example, it's not such a complicated technology, accessible only by military or special services [20].

[7]image taken from https://inside.battelle.org/blog-details/revolutionary-not-evolutionary

Hard measures. Hard measures include physically taking down the drone. One way of accomplishing that is using firearms, electromagnetic pulses (EMP), or lasers [21]. In such a case, the drone is destroyed and crashes. It will serve its purpose, but it also has its costs - those are military technologies that are not cheap and often require approval by local authorities as well. Notwithstanding that firearms are only effective at low range so their use is not common.

Another way of taking down an intruder drone is by using one or more counter drones. Such counter drones are usually autonomous, meaning they have built-in sensors using which they locate and track the moving target automatically [22]. Once the counter-drone gets close to the intruder drone, it is supposed to take down the intruder drone by either shooting a net over it, making the drone unable to fly as its propellers entangle in the net or by collision.

As a cheaper alternative to counter drones, it is also possible to use drone capture systems, i.e. net cannons, handheld guns that fire capture nets. In this case, however, the firing range is obviously limited. The SkyWall Patrol, handheld drone capture system is shown in Figure 2.6.

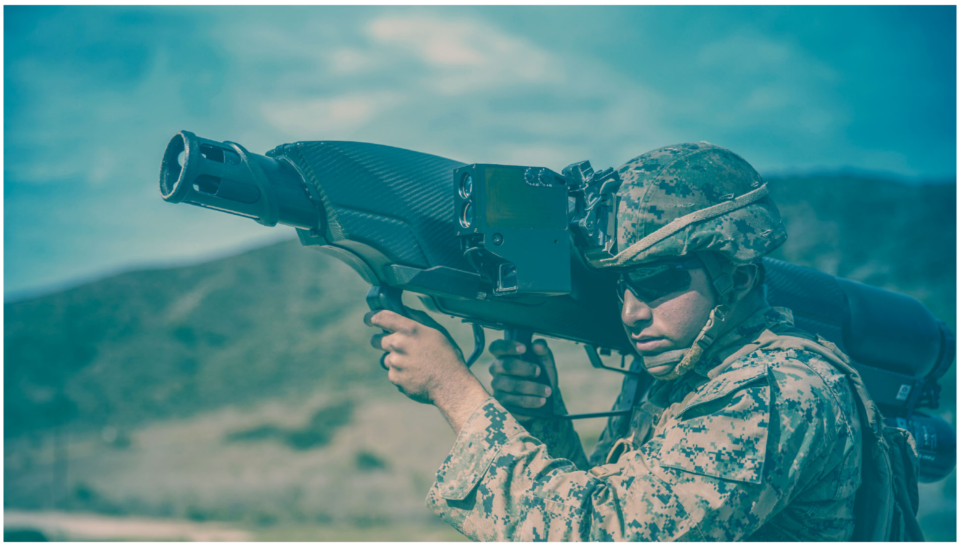

Figure 2.6: SkyWall Patrol, OpenWorks Engineering[8]

[8]image taken from `https://openworksengineering.com/skywall-patrol/`

2.5 Integrated C-UAV solutions

C-UAV systems are capable of performing all tasks required to protect against an intruder drone, including detection, tracking, and finally neutralization of the intruder drone.

In this section, we will briefly present a few state-of-the-art C-UAV solutions and their features.

Skylock anti drone systems. Skylock offers various services to prevent from drone attacks, ranging from drone detection to drone neutralization. Services include jamming technologies, where stationary and hand-held RF jammer guns are used [23] as well as non-jamming technologies where for example unique DRONELOCK system to intercept hostile drones was presented. The DRONELOCK system is integrated into a small fast-flying drone that has built-in sensors to lock onto the moving target and disable it by collision [22].

Another technology developed by Skylock anti drone systems is the Counter drone net catcher. In this case, the drone also locks into the target but instead of colliding with the intruder drone, it fires a capture net over it, swiftly demobilizing it [22].

In Figure 2.7 we show Skylocks's hand-held anti drone jammer gun with jamming range up to 1 km.

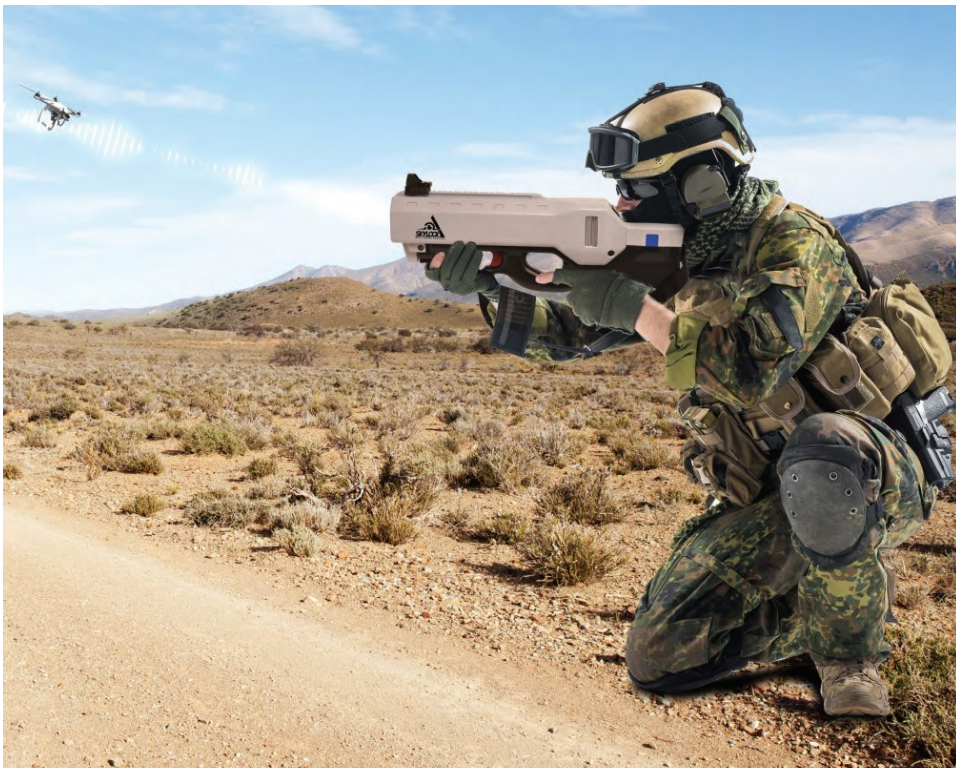

Figure 2.7: Anti drone jammer gun, Skylock[9]

[9]image taken from https://www.skylock1.com/anti-drone-jammers/?gclid=CjwKCAjw4871BRAjEiwAbxXi2wxkKOY9iR3HoDotdOfQSBfVEaCMPO7sVUS43-oBO5qOXexTKRhpsxoC6G8QAvD_BwE&utm_term=drone%20guard&utm_campaign=comp

DroneShield. DroneShield offers a complete system of products and services to counter threats possibly caused by drone technology. First of all, the system includes detection, where either stand-alone radars or body-worn RF detection devices are provided. Analysis and identification are based on multi-sensor approach, where radars are used to track the moving target and to separate it from background clutter, RF sensors are used to provide direction of bearing to the target and high definition zoom cameras and thermal sensors are used to provide visual confirmation to the user. With such a combination of sensors, the system provides high accuracy and good resistance to false alarms [24].

For a visual representation of the detection activities, a browser-based visual interface is provided.

Finally, the counter-drone response itself is done with the assistance of one of the products[10] offered by DroneShield, such as DroneGun Tactical, which is a portable, hand-held gun able to disrupt multiple RF frequency bands in the range up to 2 km.

DroneGun Tactical is presented in Figure 2.8.

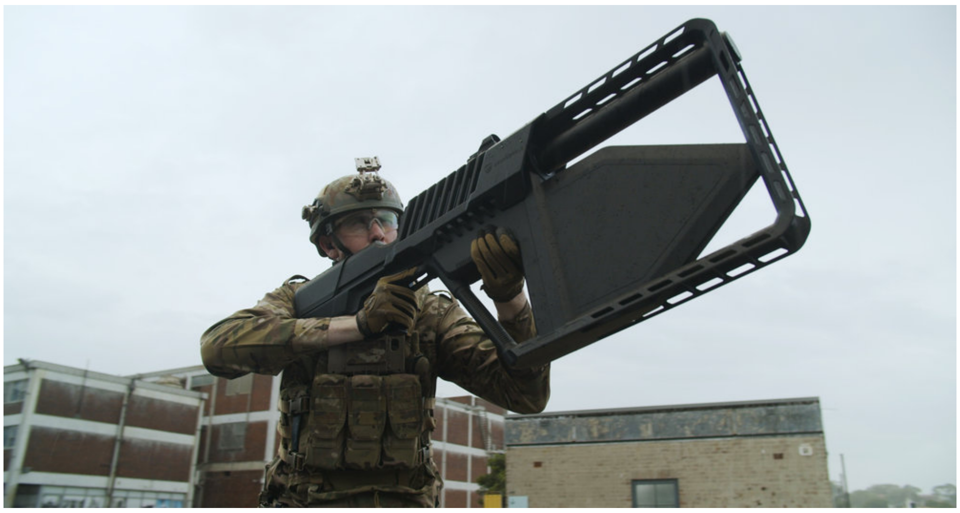
Figure 2.8: DroneGun Tactical, DroneShield[11]

Ctrl+Sky counter-drone system. Ctrl+Sky is a multi-sensor counter-drone system with the ability to detect, track, and neutralize intruder drones [25], developed by Advanced Protection Systems company. Because of the multi-sensor approach, Ctrl+Sky enables detection, identification, and neutralization of drones in both day and night as well as in all weather conditions.

The counter-drone systems exist in stationary, portable, and mobile versions and for detection usually include a 3D radar sensor with a detection range up to 3 km, RF sensor, an acoustic sensor with an array of eight microphones, and a visual sensor (camera) with optical zoom up to 30x. For neutralization, it provides a fully integrated RF jammer with the option to be activated either manually or automatically upon detection. It also comes with compatible software that can be installed on a computer or mobile device.

Figure 2.9 shows the Ctrl+Sky counter-drone system in its stationary version.

[10] `https://www.droneshield.com/view-all-products`
[11] image taken from `https://www.droneshield.com/dronegun-tactical`

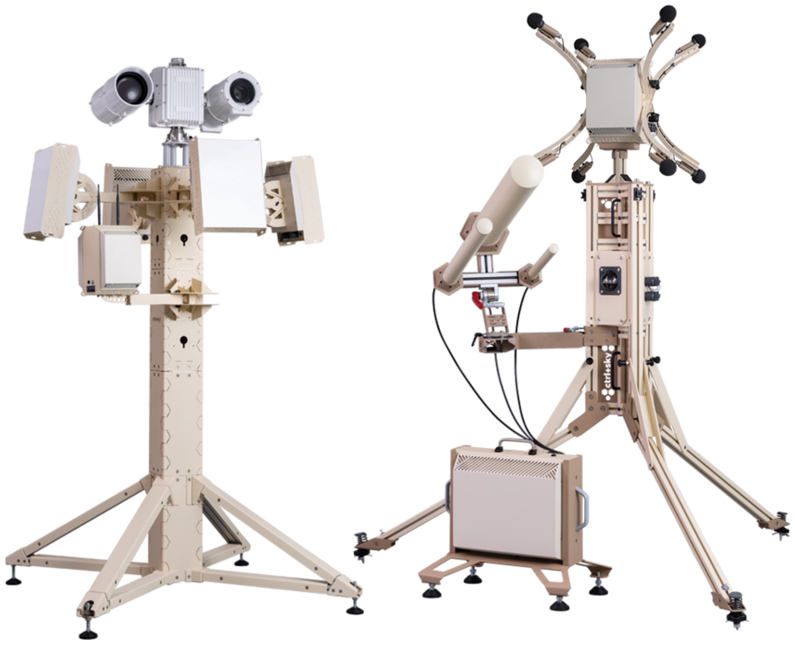

Figure 2.9: Ctrl+Sky Stationary counter-drone system[12]

[12]image taken from `https://apsystems.tech/en/products/ctrlsky-stationary/`

Chapter 3

Towards the optimal sensor placement problem

In previous years, technology has advanced enormously, which opens up a lot of new opportunities in various technology industries, including, for example, healthcare, where technology helps save lives, wireless communication allowing us to communicate with people who are on the other side of the Earth in real-time, or robotics, where machines are developed that assist people during various activities. On the other hand, with great power comes great responsibility as the technologies are becoming more and more accessible, even for ordinary people, which can pose a danger against which we must be ready to protect ourselves.

With such a level of digitization as in the world we live in now, the use of different types of sensors to monitor various properties in an area is perfectly normal. While having a smart home or cameras monitoring the private property is more of a nice thing to have rather than an indispensable thing, having proper surveillance and monitoring of, for example, traffic, or some important object, where security breaches can have more serious consequences, can often be more than appropriate, if not necessary.

When there is a need for monitoring and sensor coverage of an object, it is essential that the sensor network is efficient, as it impacts both the performance and the total cost of the sensor network. The problem of finding an effective placement of the sensors is commonly referred to as the *optimal sensor placement problem* or the *optimal camera placement problem* if only cameras are used in the network. The problem of optimal camera placement can also be understood as a restricted version of the famous art gallery problem (AGP) which has been studied in the field of computational geometry for decades. AGP concerns itself with placing one or more security guards such that the whole area of the gallery is covered by the guards, where the gallery is represented by a polygon, in a simple, two-dimensional version of the problem. For a point to be covered by a guard, the line segment connecting the guard and the point must lie inside the polygon as well, which means that the guards have full vision around themselves, i.e. the field of view (FOV) is $360°$. If we consider cameras with limited FOV instead of the guards, the problem would become very similar to the optimal camera placement problem [26].

The problem of finding the optimal sensor placement is not trivial to solve at all, as many constraints have to be taken into account, such as the modeling of the camera visibility. Following that, it is also nearly impossible to solve the optimal sensor placement in continuous space. For that reason, most of the state-of-the-art solutions aim to solve the problem in an entirely discrete domain by transitioning from the continuous space to a finite discrete space.

To solve the problem in the discrete domain, we first need to discretize the continuous space into a set of grid points, whether the continuous space is two-dimensional or three-dimensional, of which each represents a discrete position in space. Given the discrete space, we can formulate various properties about the grid points as integer or binary variables. For example, to be able to tell whether there is a sensor placed on some grid point, we would define a binary variable for each grid point, which would equal either to 0 or 1, based on whether the property holds for a particular grid point.

If the problem is modeled in such a way, it opens up several possibilities of how to solve it, one of which being the popular integer linear programming (ILP), where the decision variables are restricted to be integers and the constraints and the objective function must be linear.

Although approaching the problem of optimal sensor placement as an ILP model can lead to global optimum, it is rather impractical in a real-life scenario because ILP is NP-hard [27]. Even for small-sized problems, it is often very difficult and time-consuming to find the optimal solution. For that reason, a myriad of approximation algorithms and heuristics are being developed, to find the approximation of the optimal solution in a fraction of time compared to the ILP solutions.

As far as the problem of optimal sensor placement is concerned, most of the state-of-the-art solutions recognize two variants of the sensor placement problem, the *MIN* problem and the *FIX* problem. In the MIN problem, the task is to minimize the number of sensors such that given minimal space coverage is achieved, considering the other constraints in the problem as well. On the other hand, in the FIX problem, the task is to maximize the space coverage given a fixed number of sensors [28].

In the following sections, we will discuss various approaches to both the MIN variant and the FIX variant of the sensor placement problem, their advantages, drawbacks, and limitations.

3.1 Greedy algorithms

When an algorithm is greedy, it generally means that it tends to select the best option, i.e. the local optimum in each of its iterations, instead of seeking the global optimum by checking all possible combinations. Although greedy algorithms do not produce global optimums very often, the main advantage is the speed at which the greedy algorithms perform, compared to other algorithms or heuristics. Considering the fact that it is actually possible to achieve a local optimum that is a reasonable approximation of the global optimum with linear asymptotic complexity $\mathcal{O}(N)$ [28], the results produced by the greedy algorithms can be often sufficient. Furthermore, the greedy algorithm is usually very trivial to implement, so it might be a good candidate for the baseline solution.

One of the well-known problems where greedy algorithm performs well is the *set cover problem*. In the set cover problem, we consider a set of elements F and a collection S of m sets whose union equals F. Then, the set cover problem is to find the smallest sub-collection of S whose union equals F [29]. It was proven that the approximation ratio of greedy algorithm for the set cover problem is exactly $\ln(n) - \ln(\ln(n)) + \Theta(1)$ [30].

The idea of greedy approach in the sensor placement algorithm is to select the sensor configuration that performs the best in each iteration, i.e. covers the most grid points, until some ending criteria are encountered, such as all sensors are placed, there are no available grid points left or the maximum cost of the sensor network is reached.

Hörster's and Lienhart's version of the greedy algorithm for the optimal camera placement that they propose in [31] is based on the fact that each possible camera configuration is assigned a rank that simply states how good the particular configuration is. The rank of a camera is computed as the number of covered points that the camera would add to the set of points that are already covered by other cameras if selected, denoted as $NbCov_R$. It follows that the camera with the least overlap with other cameras will be selected in each iteration. If there are more types of cameras involved in the problem, the rank is defined as

$$r = \frac{K_k}{NbCov_R},$$

where K_k represents the price of respective camera types considered in the problem. Thus, the ratio r states the price per one point that the camera would add to the set of points covered by other cameras if it was selected in the iteration.

The main drawback of the greedy approach is obvious, as once the local optimum is reached, the solution can not be improved anymore due to the deterministic nature of the approach. To deal with it, randomness has to be introduced in the solution, bringing us a set of random sampling methods, where randomness plays a crucial role.

3.2 Sampling algorithms

Although the greedy algorithm can produce reasonable results for some problems, it fails to do so in many cases because it is not possible to improve the solution once it achieves the local optimum. Unlike that, sampling algorithms do not suffer from getting stuck in local optimum, because the solution can always be improved, simply by sampling more points of the initial dataset.

In a sampling algorithm, the idea is to sample, i.e. select either randomly or based on some probabilities, a subset S of n points from the initial set of points U referred to as the *population* [32]. The simplest type is random sampling, where all points in the population have the same probability of being selected. By repeatedly sampling a subset S from the population, different results can be achieved, thus a chance to improve the current solution always exists, if the current solution is no longer a global optimum. Such an approach is obviously very naive, as the chances of finding the global optimum are very small, if not almost zero.

A little improvement of the random sampling can be gained by setting the sampling probability of a point i, which represents a sensor position in the sensor placement problem, directly proportional to the coverage of the sensor at the position i [28]. With such an approach, sensors covering more space than the others would have a higher probability of being selected, which can lead to better results as in the case of the uniform probability distribution.

To further improve the result of sampling the subset S from the population and to avoid getting stuck in local optimum, Zhao et al. [28] adopted the Metropolis algorithm proposed by Jun S. Liu [33] and used it to solve the FIX problem. The idea is to choose a random candidate to proceed rather than exchanging with the camera maximizing the objective function, where the probability of selecting the particular candidate is proportional to the amount by which the random candidate's objective value exceeds the current choice.

One of the remaining approximation methods worth mentioning is simulated annealing (SA). SA is a probabilistic technique that can be used to approximate the global optimum of a function and its usage prevails in discrete search spaces such as in the traveling salesman problem (TSP) [34]. In [35], Junbin Liu et al. proposed a Trans-Dimensional Simulated Annealing (TDSA) algorithm to effectively solve the problem of optimal camera placement, generalized for large scale networks as well. They compare the proposed TSDA algorithm with a state-of-the-art BIP solution and gain significant results in the solution of the MIN problem in large search spaces.

3.3 Integer linear programming

In linear programming (LP), we are concerned with optimization of a linear objective function, assuming a set of constraints. The objective function consists of one or more optimization variables and the allowed values for the respective optimization variables are determined by a set of constraints that have to be linear as well. A simple example of a linear program might look as follows [36]:

$$\begin{array}{rlllll} \min & c_1x_1 & + c_2x_2 & + \cdots + & c_nx_n & \\ \text{s.t.} & a_{11}x_1 & + a_{12}x_2 & + \cdots + & a_{1n}x_n & \geq b_1 \\ & \vdots & \vdots & \vdots & \vdots & \\ & a_{m1}x_1 & + a_{m2}x_2 & + \cdots + & a_{mn}x_n & \geq b_m \end{array}$$

where c and b are coefficient vectors, A is a coefficient matrix, and X is the vector of the optimization variables, whose values are to be determined. From the point of view of geometry, the set of feasible solutions to the problem is a convex n-dimensional polytope, where n is the number of variables [27]. The optimal value of the objective function is a point in the polytope, in which the value of the objective function is the smallest or the largest, depending on whether we minimize of maximize the objective function.

So far, it is a problem of linear programming, which has shown to be solvable in polynomial time by Leonid Khachiyan in 1979 [37]. If the variables were further restricted to be integers, we would rather talk about ILP, which was shown to be NP-hard [27]. If there was another restriction that the variables are binary, i.e. can only take values 0 and 1, it would refer to 0-1 ILP, or more commonly, binary integer linear programming (BIP or BILP) [38]. If one or more variables in the problem would not be integer variables, then it would refer to mixed-integer linear programming (MIP or MILP).

A BIP formulation for both the MIN and FIX problem was proposed by Zhao et al. in [28], where two sets of binary variables were introduced, $\{x_j : j = 1,\ldots,N_p\}$ and $\{b_j : j = 1,\ldots,N_c\}$. Whether a camera is placed at position i is indicated by the fact that $b_i = 1$. A particular selection of these b_i variables defines a camera placement plan. If a position j in the space can be observed by the camera placement plan, the x variable equals to 1 for the given position, i.e. $x_j = 1$. Using these variables, the formulation of the MIN problem proposed by them is as follows:

$$\text{minimize} \sum_{i=1}^{N_c} b_i$$
$$\text{s.t. } f(x_1,\ldots,x_{N_p}) \geq p,$$

where $f(x_1,\dots,x_n)$ is a function measuring the coverage of the camera placement plan. and p is the minimal space coverage.

Similarly, they formulate the FIX problem using the same variables as follows:

$$\begin{gathered}\text{maximize } f(x_1,\dots,x_{N_p}) \\ \text{s.t. } \sum_{i=1}^{N_c} b_i \leq m,\end{gathered}$$

where m is the fixed number of cameras, and the rest is the same as in the formulation of MIN problem.

In [26], Jun-Woo et al. introduce a two-phase algorithm as a combination of both the MIN and FIX problems. In the first phase, the MIN problem is solved using BIP, obtaining the minimum of cameras that are required to satisfy the specified conditions. Then, the FIX problem is solved in the second phase, determining the best placement of the cameras for maximum coverage using the hill climbing method [39], where the number of cameras to be placed is the result of the first phase.

Horster and Lienhart also attempted to formulate a BIP solution to the optimal camera placement in [31], but they introduced their own four problems rather than solving the MIN or FIX problem.

3.3.1 ILP Solvers

There are many popular ILP solvers in the world from which to choose for the project. In terms of the free ILP solvers, the most popular is most likely the GNU Linear Programming Kit (GLPK)[1] and the lp_solve[2]. As efficiency is crucial when solving ILP programs, most of the solvers are written in C or C++. Therefore, if programming in C or C++ is not a problem or the solver offers application interfaces for other languages as well, then a free solver might be a good choice.

However, someone might want to grab a paid solver that is actively supported and maintained and more efficient. From the paid solvers, one of the solvers on the top is certainly the Gurobi optimizer[3]. Besides of that, authors claim it to be "the fastest and most powerful mathematical programming solver available for your LP, QP and MIP (MILP, MIQP, and MIQCP) problems" [40], it also offers both object-oriented interfaces for Java, .NET, Python, C++ and matrix-oriented interfaces for C, Matlab and R as well.

[1] https://www.gnu.org/software/glpk/

[2] http://lpsolve.sourceforge.net/5.5/

[3] https://www.gurobi.com/products/gurobi-optimizer/

Chapter 4

Simulation design

In this chapter, we present the architecture and design of the simulation. We mainly focus on describing the simulation as a whole rather than describing its implementation details or the sensor placement algorithm itself, for which Chapter 5 is devoted.

Firstly, we briefly present the AgentFly simulation framework which is used through the whole simulation and which makes our work easier on the visualization side of the application. Next, we show the simulation flowchart that represents the workflow of the simulation, with a brief description of the process as a whole. Then we move on to present the configuration in detail, describing mandatory input files required to run the program as well as optional input files and parameters. Finally, we describe how input files are processed, the discretization process, i.e. how a discrete environment is created for the sensor placement and properties of the graphical user interface (GUI).

4.1 AgentFly simulation framework

This thesis is built on the AgentFly[1] simulation framework which is being developed at the Department of Computer Science, Faculty of Electrical Engineering, Czech Technical University in Prague. The framework is used for various modeling and simulation, such as air traffic management (ATM), integration of UAS into shared airspace, detect and avoid (DAA), 3D trajectory planning, or tactical operations in military and security areas.

A part of the AgentFly simulation framework is a visualization engine referred as *visio*, which is build on JavaFX[2] and uses Java OpenGL (JOGL[3]) wrapper library to allow the usage of OpenGL[4] in Java, since it is originally written in C.

Visio allows us to easily develop the GUI by providing us with an application programming interface (API) that already contains the implementation of the rendering process, renderable structures, and their transformations, and provides support for widely used file formats such as OBJ[5] and MTL and user input mapping. Owing to that, we do not have to bother implementing the renderer which would cost great effort and we only have to use the structures and methods exposed by the API appropriately.

[1]https://www.agentfly.com/

[2]https://openjfx.io/

[3]https://jogamp.org/jogl/www/

[4]https://www.opengl.org/

[5]https://en.wikipedia.org/wiki/Wavefront_.obj_file

In the following sections, we will cover recent extensions of *visio* that are widely used throughout the program. However, for further and more detailed description of *visio*, see [41].

4.1.1 Sensor modelling

One of the features provided by *visio* that came in very handy through this thesis is the modeling of optical and acoustic sensors. The development of the sensors modeling originated in a bachelor's thesis by Jan Kúdelka (see [42]) and was further developed and finally integrated into *visio*.

In the following sections, we will cover the basics of techniques that needed to be implemented to allow the efficient modeling of sensors.

4.1.1.1 Ray tracing

Firstly, it was needed to implement object detection by which the sensor's visibility is modeled. For such, rendering technique called ray tracing[6] was selected. The basic idea of ray tracing is to follow rays of light from light sources as they reflect, transmit, and hit objects in the scene. Ray tracing algorithm then works by taking an image (i.e. matrix of pixels) as input, casting a ray through each pixel of the image, and recording intersections with objects in the scene. Many ray tracing algorithms also implement advanced optical effects such as reflection or refraction. In our case, however, we do not need these to be implemented, as the detection of the first intersection with an object in the scene is sufficient for our purposes.

In *visio*, an object in the scene can be represented by an array of triangles that together form a triangle mesh. Many objects, such as the environment model, can count thousands of triangles in total, needless to say, that more detailed 3D models, such as those obtained using photogrammetry[7], can count hundreds or even millions of triangles. If we were to check every triangle of every object in the scene for intersection with ray, we would end up with linear complexity, which would be very inefficient for large or high-quality scenes. To address this in-efficiency, ray tracing algorithms usually use various data structures that allow different structuring of the objects, one of them being *k-d tree*, which seems to be the most used nowadays [43].

4.1.1.2 K-d tree

K-d tree is a special case of binary-space partitioning (BSP) trees that recursively partitions the space with planes, with the constraint, that planes must be perpendicular to the axes of the coordinate system. Every inner node of the tree defines its splitting plane that divides the plane into two half-spaces that are included in the left and right child of the inner node. Leaf nodes then contain references to geometric primitives, such as the individual triangles. The building process is relatively expensive but since the k-d tree is usually built only once (i.e. at the beginning), it is not a big issue in our case, although various heuristics were invented to address the complexity of the building process, reducing it to theoretical lower bound of $\mathcal{O}(N \cdot \log(N))$ [44]. Once the k-d tree is built, it can be traversed using various algorithms, usually gaining average time complexity of $\mathcal{O}(log(N))$ and worst-case time complexity of $\mathcal{O}(N)$, since it is a binary tree [45]. From $\mathcal{O}(N)$ complexity to $\mathcal{O}(log(N))$ complexity is quite an improvement that allowed us to

[6] `https://en.wikipedia.org/wiki/Ray_tracing_(graphics)`
[7] `https://en.wikipedia.org/wiki/Photogrammetry`

perform ray tracing operations efficiently, which was more than necessary not only during modeling of sensors but also throughout the whole program.

4.1.1.3 Sensor scanning

With the efficient implementation of ray tracing, we were finally able to model sensor detection. Currently, there is support for two types of sensors in *visio*, the camera sensor, and the acoustic sensor. Detection is obviously modeled differently for each of them but both use ray tracing in their implementation.

Camera sensor In *visio*, the camera sensor is defined by several parameters, including basic parameters such as resolution, direction (i.e. orientation), field of view (FOV), and far and near clipping planes. The camera sensor can also be set with *environment trees*, which are k-d trees built from triangles of static structures in the scene such as the environment model.

When computing visibility of camera sensor, *visio* provides us with a method called *fullscan*, parameterized by a lambda expression, in which we can define what should happen with the result, and optionally by *target tree* that represents k-d tree built from triangles of an object in the scene whose visibility we want to determine. During *fullscan*, a ray is cast through each pixel of the sensor's screen and its intersection with either environment trees or the target tree is computed, whichever it hits first. It is important to note that the intersection is firstly computed for environment trees if there are any, where the result is cached, as environment trees are supposed not to change, and then for the target tree. Internally, the *fullscan* invokes a similar method called *boxscan* which does the same in principle but only casts ray through pixels within the axis-aligned bounding box (AABB) of the *target tree*, which improves the efficiency as it omits pixels that are guaranteed not to result in an intersection.

In practice, we usually perform a *fullscan* with no target tree first, which computes intersections with the environment trees for all pixels of the camera sensor's resolution and caches the result. Then we follow with another *fullscan* where the target tree is already specified, which internally performs a *box scan* and computes intersections only for relevant pixels. Finally, we store the intersection distances for each pixel in a matrix and provided with the lambda expression, we can retrieve additional information about the result, i.e. which particular k-d tree (some of the environment trees or target tree) or even which particular triangle it intersects for a particular pixel.

In Figure 4.1 we show the graphical representation of the camera sensor with Full HD resolution (1920x1080), 30° FOV, 120 m range and a drone in its view. In Figure 4.2 we show the scene from the view of the same camera sensor, where blue pixels represent the intersection with the environment model (bluer means closer), red pixels represent the intersection with the drone and green pixels represent no intersection. The drone is scaled up six times to be better seen.

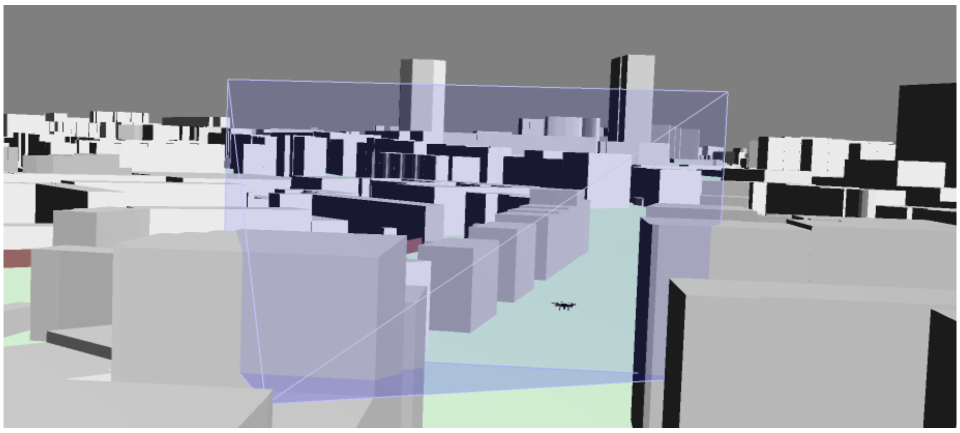
Figure 4.1: Drone detection using the camera sensor

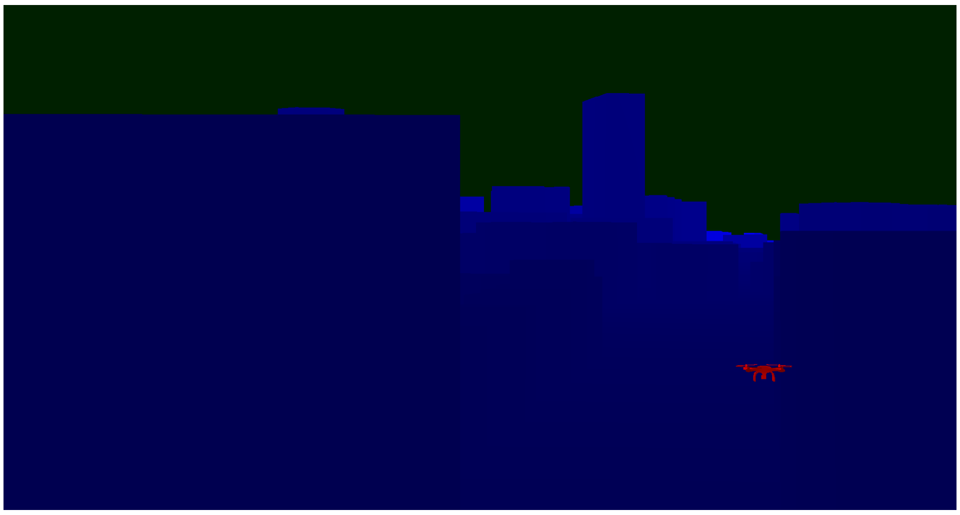
Figure 4.2: The image of the camera sensor

Acoustic sensor In contrast with the camera sensor where direct visibility of a given object is examined, the acoustic sensor rather tries to determine whether the sound of certain power level and frequency emitted from a given sound source will reach the acoustic sensor, considering reflections from the surface and sound absorption in the atmosphere.

Parameters of the acoustic sensor include air properties such as pressure, temperature, and humidity, and sensitivity, which determines the minimum required power level of sound for the sound source to be considered as detected by the sensor. Furthermore, environment trees can be set to the acoustic sensor as well. Finally, the acoustic sensor can be parameterized by trace properties that define the initial number of generated rays, the number of rays generated upon reflection, and maximum reflection depth, where rays represent the emitted sound.

Similarly, as in the case of the camera sensor, *visio* provides us with *scan* method, parameterized by lambda expression to define actions upon completion of the computation and by power level, frequency and position of the sound source. *Scan* firstly calculates atmospheric absorption of sound (in decibels per 1 m) and checks whether the sound is even able to reach the acoustic sensor with the desired power level. Then it randomly casts a given number of initial rays, checking intersections with environment trees and decreasing the power level as the sound travels through the air. Note that the rays are cast from the position of the sound source and not from the sensor, as it is in the case of the camera sensor. If the ray hits any environment tree, a given number of reflection rays is generated and the process repeats until maximum reflection depth is reached. If the current power level of a ray is less than the minimum required power level at any given moment during its path, the process is terminated for the particular ray. If the ray reaches the position of the sensor (with some tolerance radius in the order of units of meters) with the desired power level, the sound source is evaluated as detected.

In Figure 4.3 we illustrate the detection of a drone using the acoustic sensor, where the purple cube represents the acoustic sensor on the corner of a building and the red-black lines represent the respective rays emitted from the sound source, which is the drone. The transition of ray color from red to black speak for the gradual decrease of the power level of the sound. The drone is scaled up six times to be better seen as well.

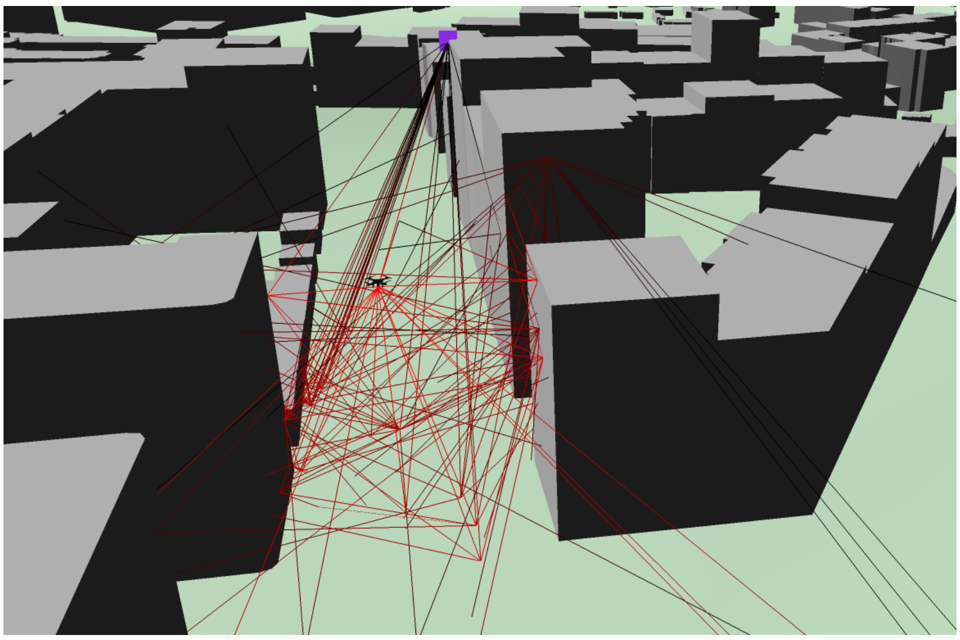
Figure 4.3: Drone detection using the acoustic sensor

4.2 Simulation workflow

The proposed design of the program consists of several steps that form the whole simulation process from the very beginning to its very end.

Firstly, the simulation reads the configuration file provided by the user, where paths to all necessary input files are specified. It can also contain paths to optional input files and values for optional parameters with the help of which a suitable setting of the environment for sensor placement can be achieved.

After that, the simulation loads the input files on given paths from the file system. Those input files represent the scene, i.e. the geographical environment model and the monitored sector, in the surroundings of which the sensor placement and following experiment will take place. Based on the loaded scene and specifically the monitored sector, it builds the so-called *world data structure* (hereinafter referred to as *the world*), by selecting an area of interest around the sector and discretizing it into a three-dimensional grid, to not operate in continuous space.

At this point, the simulation has processed all the input files and prepared a suitable and discrete environment in which the sensor placement can be solved. The simulation begins to solve the sensor placement in the background and during that, the GUI is launched. When the calculation of the sensor placement finishes, the simulation collects the results and deploys the sensors on their specific positions and orientations that were computed by the sensor placement algorithm and finally, visualizes them in the GUI.

Although we evaluate the sensor placement using three criteria that we described in Subsection 4.3.8, we want to know its performance and overall quality in practice, so an experimental evaluation follows. The experiment consists of a few detection scenarios, where a number of drones are used to simulate various types of both attack or non-attack flights, including spying flights, smuggling flights, flights with the purpose of an attack, and ordinary flights by ordinary (or inexperienced or both) persons.

Finally, the results of the experiment are collected, evaluated, and summarized. Brief feedback is given, providing information to what extent the sensor placement was sufficient and what were its shortcomings.

Individual steps of the simulation workflow are presented in Figure 4.4.

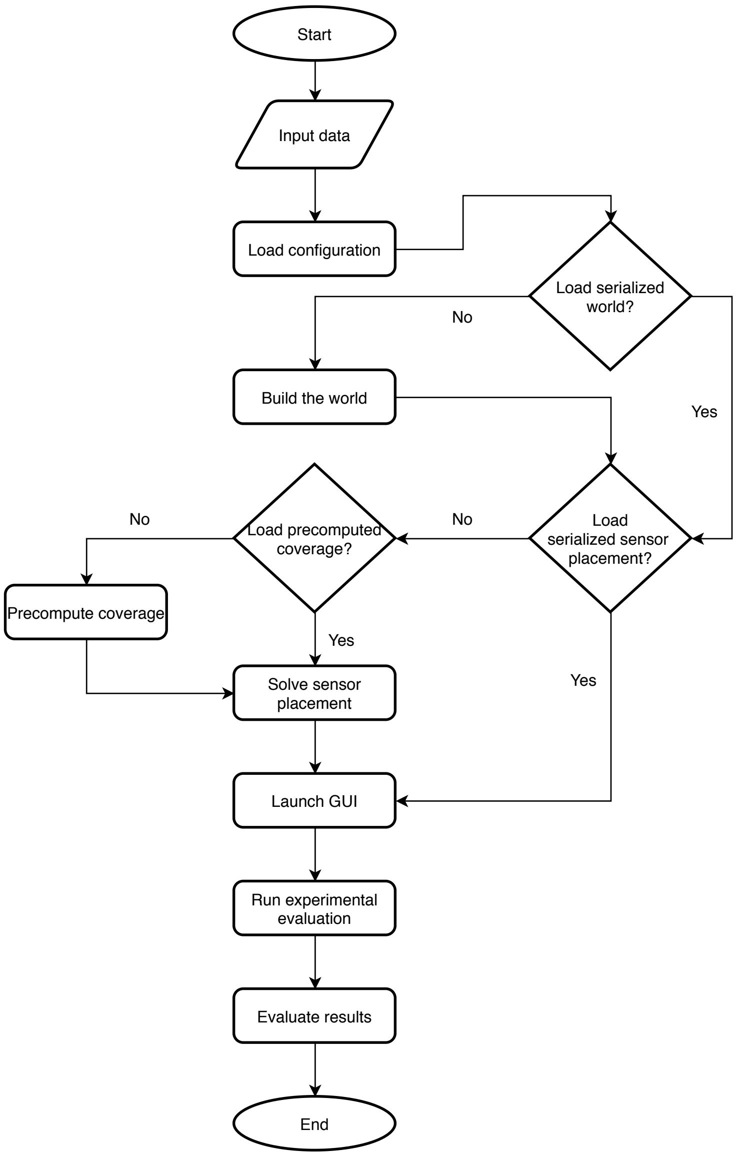

Figure 4.4: The simulation flowchart

4.3 Configuration overview

In order to run the simulation, we must provide it with the configuration file in which paths to input files and values for parameters are stored.

The configuration properties might be of three types:

1. **Mandatory input files**. Input files, i.e. paths to them, without which the simulation can't be started. These input files include the scene that will be rendered and the monitored sector where the sensor placement will take place.

2. **Optional input files**. Input files, i.e. paths to them, for which, if not provided, the simulation can automatically generate values or can run without any. These input files include priority areas, sensor spots, and serialized data, containing preprocessed data such as precomputed coverage for a particular set of sensor spots or already solved sensor placements.

3. **Input parameters**. Parameters can be used to configure the process of the world construction to the desired level and to influence the output of the sensor placement. None of the parameters is mandatory, meaning that there are default values hard-coded for each parameter. Parameterized can be for instance the size and weight of cells into which the monitored space is discretized or the maximum cost of the sensor placement.

We will now introduce a few of the main configuration properties in separate sections, describing them in more detail.

4.3.1 Geographical environment model

The main input file the simulation requires is a 3D object file in OBJ format that contains the three-dimensional object representing the geographical environment model that is to be used in the simulation, i.e. the surface terrain, buildings, and other real-world objects.

Once the simulation loads and parses the environment model, it is represented as triangle mesh[8] with individual triangles stored in an array so that it can be easily worked with.

In Figure 4.5 we show the wire-frame[9] model of the Šibenik Cathedral[10] .

[8]https://en.wikipedia.org/wiki/Triangle_mesh
[9]https://en.wikipedia.org/wiki/Wire-frame_model
[10]https://en.wikipedia.org/wiki/%C5%A0ibenik_Cathedral

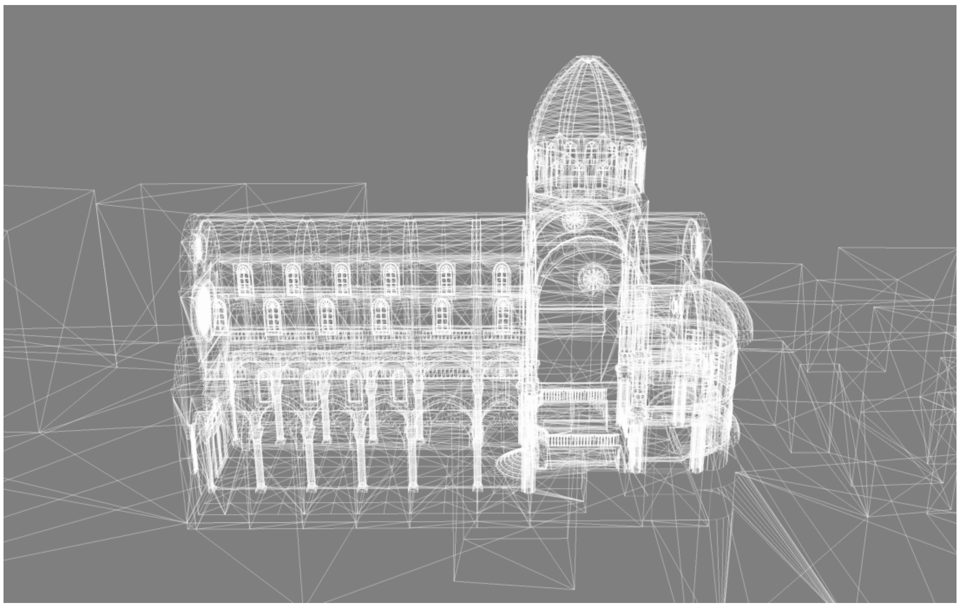
Figure 4.5: The wire-frame model of the Šibenik Cathedral

In order to make the environment model look more realistic, it can be supported by a complementary material file in .MTL format that can contain surface shading material for the environment model.

The effect of the material file is presented in Figure 4.6, showing the DJI Matrice 600 Pro[11] in wire-frame representation and Figure 4.7, showing the same drone with materials applied.

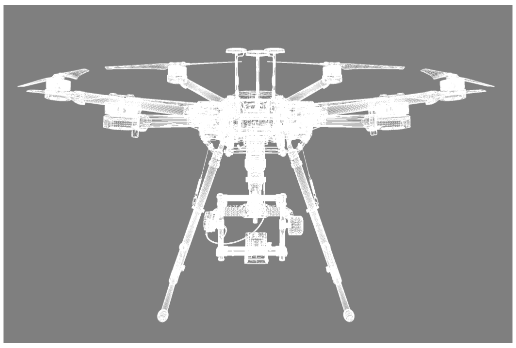
Figure 4.6: DJI Matrice 600 Pro in wire-frame representation

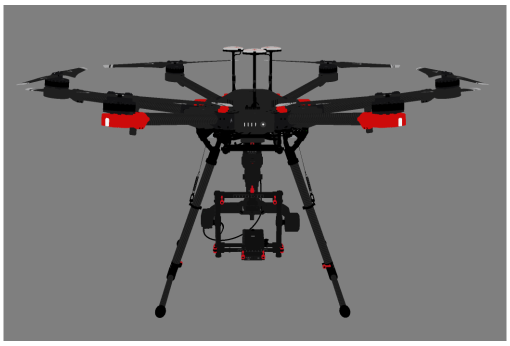
Figure 4.7: DJI Matrice 600 Pro with materials

In case that we don't have the material file and we don't want to bother generating it, we can assign materials to the environment model directly in the program. In such a case, the only difficulty occurs when we want to assign different materials to different parts of the environment model, i.e. terrain and buildings. In such a case, we have to separate the parts from each other and load them as separate environment models.

In Figure 4.8 we show a part of the environment model of Prague with different materials assigned to terrain and buildings.

[11] `https://www.dji.com/cz/matrice600-pro`

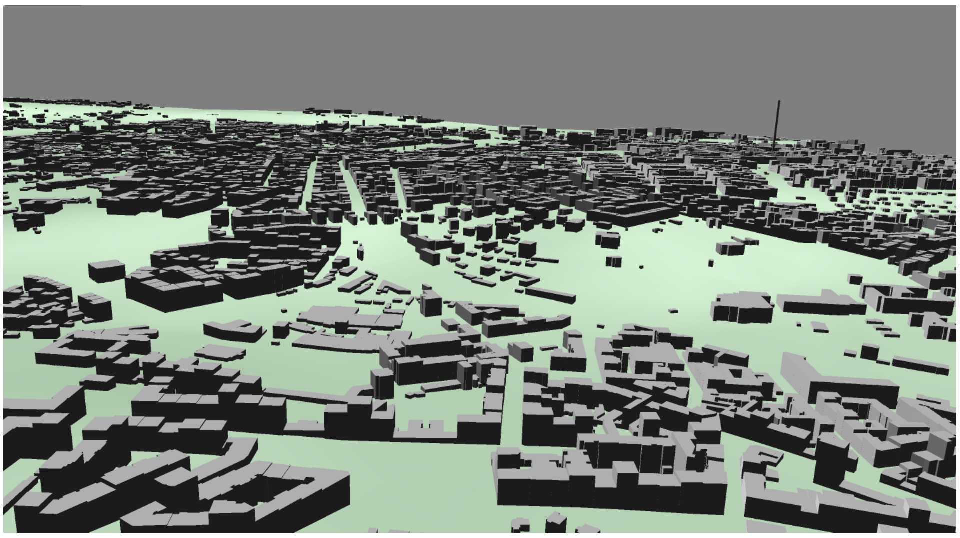

Figure 4.8: The model of Prague with different materials assigned to terrain and buildings

4.3.2 Sector

Another input file that must be provided to the simulation is the sector that is the actual subject of the sensor placement, i.e. the sector that we want to have monitored by the sensors. We can also understand it as a private or restricted area that must not be occupied by any intruder drones and thus must be monitored.

The sector is represented by a polygon defined by a set of positions that are loaded from a text file, where each position represents a single vertex of the polygon.

The simulation also offers a possibility to easily generate custom sectors within the simulation. By clicking the mouse left button, we can generate positions that can be later saved into a text file using a specific key bind.

We show the boundary of the sector of the Václav Havel Airport Prague in Figure 4.9, visualized as purple lines that connect consecutive positions of the sector.

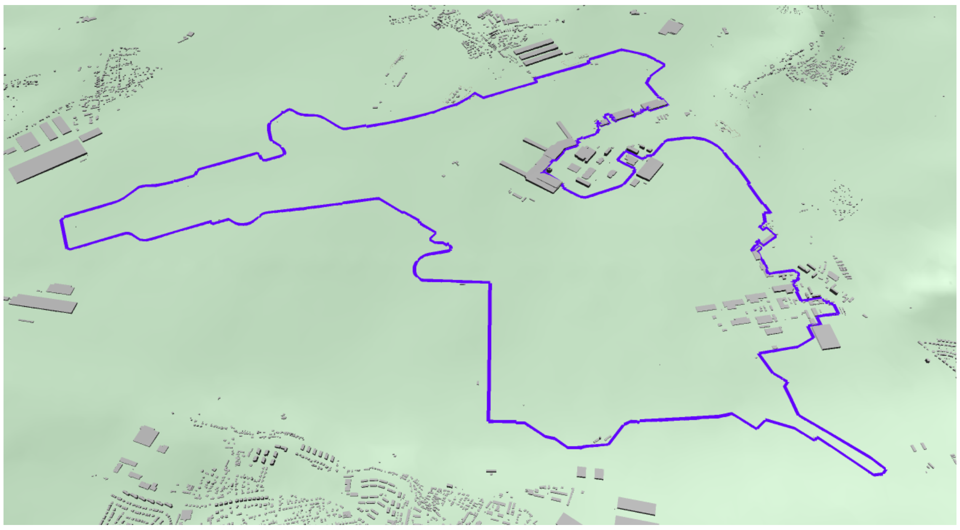

Figure 4.9: The sector of the Václav Havel Airport Prague

4.3.3 Priority areas

In terms of optional input files, the first one to mention is the priority areas file. Priority areas represent polygon shape areas with higher importance requiring a higher degree of monitoring, which is taken into account in the sensor placement later. Priority areas are loaded from a text file similarly as in the case of the sector, with the only difference that a single priority areas file can contain multiple priority areas, in such case they are separated by *priority_area* string separator.

Similarly, as in the case of the sector, simulation offers generation of priority areas by clicking the mouse left button and saving them into files using specific key binds.

We show the visualization of priority areas in Figure 4.10.

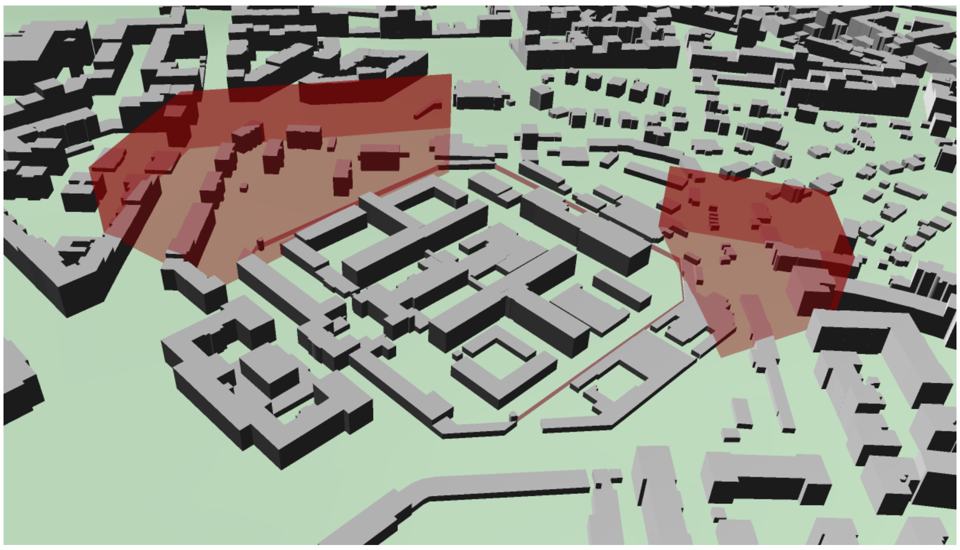

Figure 4.10: Two priority areas near the sector of the Pankrác Prison

4.3.4 Sensor spots

Another optional input file, that can be provided to the simulation, is the sensor spots file. Its format is the same as the format of sector file, thus a text file with a set of positions separated by a newline.

Sensor spots are discrete positions where it is possible to place a sensor, no matter what type of sensor it is. By default, when no sensor spots file is provided, sensor spot positions are sampled from the set of positions defining the sector (i.e. its perimeter). To be able to control the number and density of sensor spots that will be generated, sampling considers a constraint parameter *maximal spacing* that states the maximum allowed distance between each pair of consecutive positions.

When the sensor spots file is provided, which is the preferred option, the sampling is not done at all and the positions provided in the file are used instead, enabling us to set fixed positions as sensor spots that must not be located on the sector's perimeter.

Similarly, as in the case of sector and priority areas, simulation offers generation of sensor spots by mouse clicking and saving them using a specific key bind, too.

4.3.5 Serialized data

As we mentioned earlier in this section, simulation also offers the possibility to save the current state of its parts and load it in the future. This option was implemented for the sake of simplicity and time saving during the development and sensor placement tuning, where expensive or long-lasting processes would be repeated over and over again, although they would be identical every time, such as the world building process or precomputation phase of the sensor placement, which we describe in more detail in Subsection 5.3.2.

Unlike in the case of sector, priority areas, or sensor handlers mentioned in the previous sections, it can't be invoked by the user and is done automatically by the program instead. The serialization is invoked at the following points of execution:

1. **When the world is created.** After the world is built, the simulation automatically serializes it into a file. When the world is large or the world cells are very small (e.g. 10 m x 10 m x 10 m), it might count a large number of world cells (speaking about hundreds of thousands), in which case its building process might take some time. Hence, it might come to use not to construct the world from scratch in each run of the simulation but to load it from the serialized file instead.

2. **When the precomputation phase finishes.** The first phase of the sensor placement is the precomputation phase, where coverage is computed for each possible combination of sensor type, sensor spot, and sensor configuration (such as the camera pose (i.e. orientation) in case of the camera sensor). As the number of such combinations can count thousands, computing the coverages several thousand times with respect to a possibly large number of world cells often also takes a long time. With the precomputed coverage serialized, we can avoid repeating the computation and rather move on to the sensor placement instead.

3. **When sensor placement finishes.** The next point of execution when serialization is invoked is as soon as the sensor placement finishes. The sensor placement output with all necessary data, mainly the deployed sensors, is serialized, allowing us to repeatedly use the same sensor placement for the experiment that follows, without the need to compute it from scratch each time.

4. **When the experiment flights are generated.** The last point of execution when serialization is invoked is when the experiment flights are generated. In this case, it is not about the process of generating the experiment flights being expensive but rather about having the ability to use the same set of flights repeatedly in different executions of the experiment.

4.3.6 World configuration

Since the simulation happens to operate is continuous space consisting of an infinite number of arbitrary points, we need to discretize the space firstly. The discretization is done by splitting the space into a number of three-dimensional axis-aligned cells, referred to as *world cells*, which together form a three-dimensional grid.

In order not to discretize the whole space out of which only a fraction is used in the sensor placement, we define an area of interest around the monitored sector that we call the *world*, as we already mentioned earlier. The world is shaped like a three-dimensional waist around the sector's perimeter and its dimension is given by two parameters - *horizontal monitor distance* states the width of the waist and *vertical monitor distance* states its height, both in meters.

We show the visual representation of the world in Figure 4.11, with horizontal monitor distance being 100 m, vertical monitor distance being 50 m and the dimension of world cells being 25 m x 25 m.

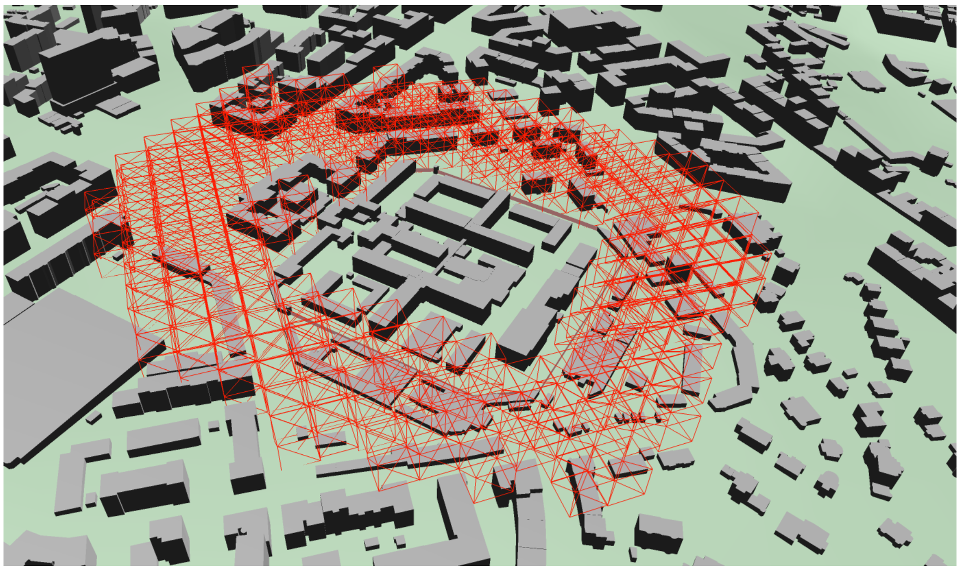

Figure 4.11: Discretized world around the Pankrác Prison

4.3.7 World cell configuration

Other important parameters that can heavily influence the final shape of the world, as well as the sensor placement, relate to world cells, i.e. the discrete cells the world was discretized into. The dimension of a single world cell is given by two parameters - *cell height* representing the height of the cell and *cell width* representing its width, both in meters. If its height and width are equal, then the cell is represented as a cube. Otherwise, it is a square prism (cuboid with a square base but a different height).

With the ability to change the dimensions of the world or world cells, we can control the final number of world cells that are generated by discretization. That allows us to find a suitable environment for the sensor placement, offering a good compromise between complexity (the more world cells we have, the longer the computation takes) and accuracy (with world cells of smaller size, computations might be more accurate in terms of coverage).

Furthermore, we recognize two categories of world cells - priority cells or normal cells. By default, all world cells are considered normal cells, unless they lie within any of the priority areas. In such a case, they are determined to be priority cells.

In relation to world cell categories, each world cell has its weight based on which category it falls in, which can be later used to influence the sensor placement output and possibly achieve better results.

4.3.8 Sensor placement configuration

Apart from the world configuration, the sensor placement itself can be parameterized as well.

First of all, the sensor placement algorithm is actually indirectly parameterized by the dimension of the world and world cells, as they strongly influence the computational time and the overall precision of the resultant sensor placement.

Another world parameter that plays an important role in the sensor placement is the weights of individual world cells which the sensor placement algorithm uses to recognize the priority cells from normal ones and also to distinguish between covered and non-covered cells.

Last but not least, sensor spots can be considered to be sensor placement parameters as well, as they specify the maximum number of sensors that can be deployed.

Speaking about direct parameters, worth mentioning is definitely the parameter *budget* which states the maximum allowed combined price of the sensors deployed within the sensor placement. The budget is most often the first encountered ending criteria during the optimization of the sensor placement.

The next important parameter is the *time limit* which specifies the maximum time until the computation terminates. Time limit is not mandatory and can be completely left out, in such a case the sensor placement doesn't terminate until fully computed. That can, however, take a really long time and can be rather unworthy, as improvements between iterations in the final phases of computation are often small.

Finally, the last parameter is the *coverage redundancy* which states the optimal coverage level, i.e. optimal number of sensors that cover each world cell. To account for the current coverage level of individual cells, their weights decrease with each additional sensor coverage by a factor of two. With this approach, the sensor placement is able to determine whether a cell is already covered by one or more sensors and eventually prefer other cells, i.e. those which are not yet covered at all.

4.4 Setting up the environment

In this section, we describe the necessary steps that must precede the sensor placement in order to set up a suitable environment for it.

4.4.1 Input files processing

After the program reads the configuration, input files must be loaded, processed, and stored in appropriate structures so that they can be easily operated with.

Firstly, the program loads the environment model file and processes it using *OBJLoader* that is provided by the AgentFly simulation framework and is specifically designed for purposes of loading and parsing files in .OBJ format. The produced result is an array of triangles that is essential for the functionality of the program as it is used to build a k-d tree structure that allows us to perform ray tracing operations later.

After the environment model is processed, the program proceeds to load the sector from the provided text file. The file is simply read line by line and values are added to the resulting set, using which the sector is initialized. Right after that, the program moves on to load the priority areas and sensor spots from their respective text file the same way as the sector, if paths to them are provided in the configuration.

When processing text files, such as those with sector, priority areas, or sensor spots, the expected format of positions is one of the following:

1. **Spheric coordinates**. Coordinates in spherical coordinate system[12], containing values for latitude and longitude in degrees and optionally altitude in meters.

[12]https://en.wikipedia.org/wiki/Spherical_coordinate_system

2. **Model coordinates**. Coordinates directly in the model space used in the program, i.e. triples of values along the x, y, and z axes.

The former option is implemented to enable the possibility to generate positions by an external service (such as Google maps for example) that would most likely provide them as spheric coordinates, and load them within the program without the need to manually convert them to the model space. In this case, the program automatically converts the positions to model space. Moreover, if the value of altitude is equal to zero or missing, the program automatically computes exact altitude with respect to the loaded environment model.

The latter option is implemented for a similar reason, namely to allow loading of positions that were generated within the program without the need to transform them to spheric coordinates beforehand.

4.4.1.1 Coordinate systems

As mentioned in the previous section, implementation uses two different formats of coordinates: *spheric coordinates* and *model coordinates*. In order to allow transitions between these two formats, the implementation provides methods that contain appropriate conversions.

In order to convert a position from spheric coordinates to model space, we must first convert it to Cartesian coordinate system[13], which can be achieved by applying the formula

$$(x, y, z) = (r \cdot \cos(\theta) \cdot \cos(\varphi),\ r \cdot \cos(\theta) \cdot \sin(\varphi),\ r \cdot \sin(\theta)), \tag{4.1}$$

where r is the mean Earth radius (6371009 m) + altitude in meters, θ is latitude in radians, φ is longitude in radians and (x, y, z) is the resulting position in Cartesian coordinate system. Then, we obtain the resulting position in the model space by converting the position in Cartesian coordinate system to model space using specific transformation matrix that is dependent on current environment model. In order to compute exact altitude with respect to current environment model, we can use ray tracing, where a ray perpendicular to the surface is cast and its intersection with environment surface is recorded.

When converting in the opposite direction, that is from model space to spheric coordinates, the order of conversions is reversed. Again, the position is firstly converted to Cartesian coordinate system the same way as before, just with the transformation matrix inversed. To convert it further to spheric coordinates, we apply the formula

$$(lat, long) = (\frac{\Pi}{2} - \arccos(\frac{z}{r}),\ \text{atan2}(y, x)), \tag{4.2}$$

where x, y, z are values along respective axes of the position in the Cartesian coordinate system, r is the length of (x, y, z) vector - mean Earth radius and (lat, long) are resulting values of latitude and longitude in radians.

4.4.2 Space discretization

After we processed the input files, we are finally able to construct the world, based on its parameter values specified in the configuration file.

The world construction process can be divided into three gradual steps:

[13] https://en.wikipedia.org/wiki/Cartesian_coordinate_system

1. **Select relevant area.** The available part of the whole space is only the part where the loaded environment is defined. However, we are only interested in its part, namely the part where the sector is located, plus some additional space around the sector to cover the waist around the sector. The computation of bounds of the relevant area is simple: we construct an AABB, whose bounds are given by the top-most, right-most, bottom-most and left-most positions of the sector and the value of *horizontal monitor distance* that is specified in the configuration file. Resulting AABB defines the maximum area that is relevant for the sensor placement.

2. **Discretize the area.** Since operating in continuous space would be complex and unnecessarily expensive, we discretize the continuous space of the relevant area into a number of axis-aligned world cells. Discretization starts in the top-left corner of the relevant area and continues firstly by columns and then by rows, similarly as when iterating a matrix. For each cell, its closest distance to the sector is computed, based on which it is determined whether it falls within the waist around the sector or not, in such case it is discarded. When the process finishes, we have not only discretized the whole relevant area (and thus got rid of continuous space) but we have also reduced it to even smaller area, having much fewer world cells in total, which improves efficiency during the sensor placement later.

3. **Build graph.** Since the world cells themselves do not contain any relational information about each other (such as neighboring cells), we build a graph on top of the world in which the relational information about world cells can be stored. The graph is built in a way such that each world cell is also a node of the graph and the neighboring world cells are its neighbors. Having the world cells connected like this allows us for easier execution of search operations that we often need to perform, including for example the selection of subareas within a radius from origin or solutions of shortest paths problems if it ever was necessary.

After the above-mentioned steps are completed, the world is fully initialized and its state is automatically serialized in a file, allowing us to load the same world in the future, without the need to repeat the process from scratch.

4.4.3 Graphical User Interface

As we have fully processed all input files, we are ready to run the GUI and fill it with a graphical representation of the environment model, priority areas, and the world (represented by individual world cells).

GUI is based on *visio*, a part of the AgentFly simulation framework, which handles all necessary initialization and rendering of graphics. The only task is to transform the objects that we want to display in the GUI to renderable structures provided by the framework and give it a signal to invoke the rendering tasks. GUI is also separated from the program's logic and the communication between these two sides is implemented using the observer pattern[14]. As it is not the main subject of this thesis, we will not further describe the implementation details related to GUI.

GUI offers user-interaction using the keyboard or mouse buttons, such as controlling the position of camera using arrow keys, controlling its direction by dragging the mouse, and zooming in and out using the mouse wheel. Furthermore,

[14] https://www.tutorialspoint.com/design_pattern/observer_pattern.htm

it allows the user to display or hide particular graphical objects, such as priority areas, sector's perimeter, sensors deployed in the sensor placement, or drones that are used in the experiments.

Figure showing GUI is omitted in this section because it is already shown in Figure 4.8, Figure 4.10 and Figure 4.11.

Chapter 5

Implementation

In the previous chapter, we presented the application as a whole, focusing on describing the configuration properties and possible inputs, individual components, important data structures, and graphic design.

This chapter is devoted to the description of implementation details of various parts of the application, where we consider it appropriate to elaborate in more detail. Firstly, we briefly describe the supportive features that helped us to optimize the performance during the computation of the sensor placement. Then, we introduce our approach to the precise calculation of coverage of the individual sensors in detail. Finally, we present our approach to the proposed sensor placement algorithm. We start with a description of the input parameters and the precomputation phase, in which we utilize some of the supportive features. Lastly, we present our solution to the sensor placement problem using mixed-integer linear programming (MILP). We formalize the MILP model and describe each variable, constraint, and parameter with a detailed explanation of their purpose.

5.1 World graph

As we already mentioned in Subsection 4.4.2, we build a graph on top of the *world data structure* (hereinafter referred to as *the world*) in order to store relational information between world cells (further only as 'cells'). If the relational information was not present, we would have to traverse all world cells every time we would like to find neighboring world cells for a given cell. When we structure the world as a graph, we allow for efficient traversal using standard graph traversal algorithms such as depth-first search[1] or breadth-first search[2].

Using the graph, we implemented two important methods which we refer to as *surface scanning* and *expansion*, that we further utilized during the development of the sensor placement algorithm. The purpose of the *surface scanning* method is to determine particular sensor spots where it could potentially be appropriate to place an acoustic sensor because placing acoustic sensors in an open space would not be effective due to their higher price. In terms of the *expansion* method, it is used solely for optimization purposes during the precomputation phase of the sensor placement algorithm.

[1]https://en.wikipedia.org/wiki/Depth-first_search

[2]https://en.wikipedia.org/wiki/Breadth-first_search

5.1.1 Expansion

Taking advantage of having the world structured as a graph, using the *expansion* method we can efficiently collect world cells around a given origin position, that satisfy given constraint. The procedure is simple: starting in the origin position, we perform a breadth-first search and expand the graph nodes as long as the given constraint is satisfied.

Throughout the program, we mainly use a specific type of expansion which we call the *circlic expansion*. During a *circlic expansion*, we consider a simple constraint which is the radius from the origin position. Thus, *circlic expansion* collects all world cells whose centers are within a given radius from the origin position. Using this method, we gained significant performance improvements during the precomputation phase of the sensor placement, which we will discuss later in Section 5.2.

5.1.2 Surface scanning

Surface scanning is a primitive method that allows us to roughly determine the evenness of an area on the environment model. Luckily, in our case, we are only interested in knowing whether there are tall objects, such as buildings, for example, located in the area that we are scanning. For that reason, this method is sufficient, although it is certainly not the most accurate and precise method.

When we scan the surface of a world cell, we scan the part of the surface that would be occupied by the vertical projection of the lower base of the cell onto the surface, i.e. ignoring the altitude in which the world cell is located. The surface scan is always performed in the area of a particular world cell. First, we generate a matrix of evenly distributed points within the area that we want to scan. The size of the matrix depends on the size of the area, as the points are generated with a constraint that states the minimum distance in meters between each pair of neighboring points.

Then, using ray tracing, we scan the surface in each of the generated points by casting a ray perpendicular to the surface, which gives us a matrix of elevation values for each of the generated points. Using this matrix, we can roughly analyze the surface of the scanned area in two different ways.

The first way requires, in addition to the environment model, an additional terrain model, i.e. a mere terrain model without any buildings or other objects. In such a case, we can perform the same surface scan (i.e. at the same points) on both the environment model and terrain model, gaining two matrices of elevation values. By pair-comparison of elevation values on the same indexes of each matrix, we can identify particular points in which the elevations differ by at least X m, where X is a predefined threshold (currently hard-coded to 5 m). Then, if the number of differing points is at least Y percent of the total number of generated points, where Y is a predefined threshold, the particular world cell is determined to have an uneven surface.

When the terrain model is not available for some reason, the other way can be used. In that case, we compare neighboring points of the matrix of elevation values, recording the number of points in which the elevation differs by at least X m, where X is a predefined threshold. Then, as in the first method, if the number makes at least Y percent of all generated points, the particular world cell is determined to have an uneven surface.

In Figure 5.1 we show the sector of the Václav Havel Airport Prague with the horizontal monitor distance of 600 m where the world cells with the uneven surface are displayed as green points.

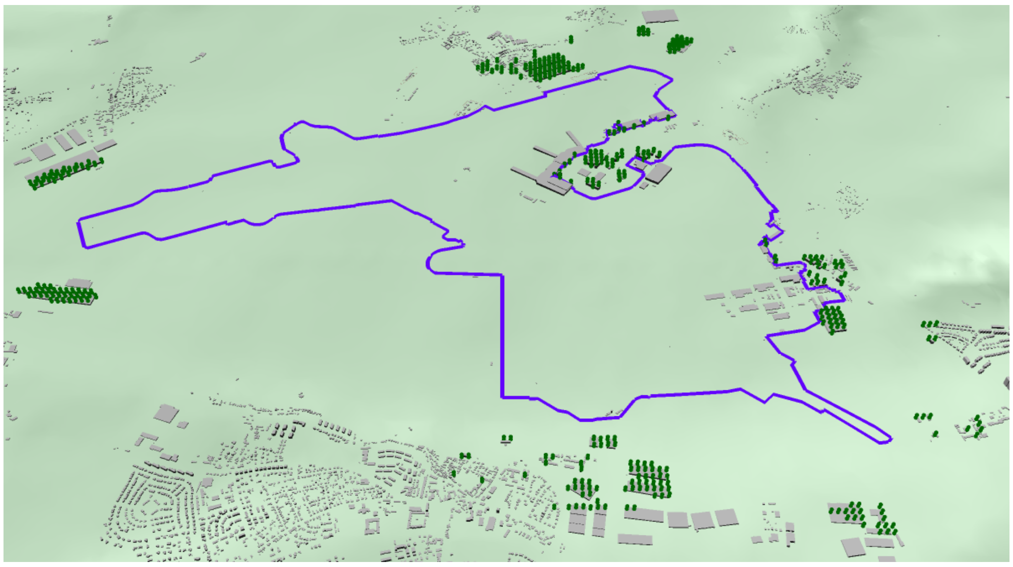

Figure 5.1: The sector of the Václav Havel Airport Prague showing the world cells with uneven surface as green points

The information we obtained by this method, i.e. which of the cells have an uneven surface, we further utilize in the determination which sensor spots are appropriate candidates for acoustic sensors. The idea is as follows: from each cell with an uneven surface, perform the *expansion*, collect all cells within the radius of 200 m and identify all sensor spots that are located in the collected cells as appropriate candidates for acoustic sensors. We chose a radius of 200 m because it was shown to be roughly the maximum distance at which is an acoustic sensor with reasonable configuration able to detect a drone, as discussed in Subsection 5.2.3. The cells that are collected during the expansion from one of the sensor spots are excluded from following expansions from the remaining sensor spots, i.e. are skipped as it is unnecessary and ineffective to expand them more than once.

In Figure 5.2 we show the sector of the Václav Havel Airport Prague and the world cells that contain the specific sensor spots that were identified as appropriate candidates for acoustic sensors, displayed as green points. As can be seen, only sensor spots near the areas with uneven surface were selected as suitable candidates for acoustic sensors.

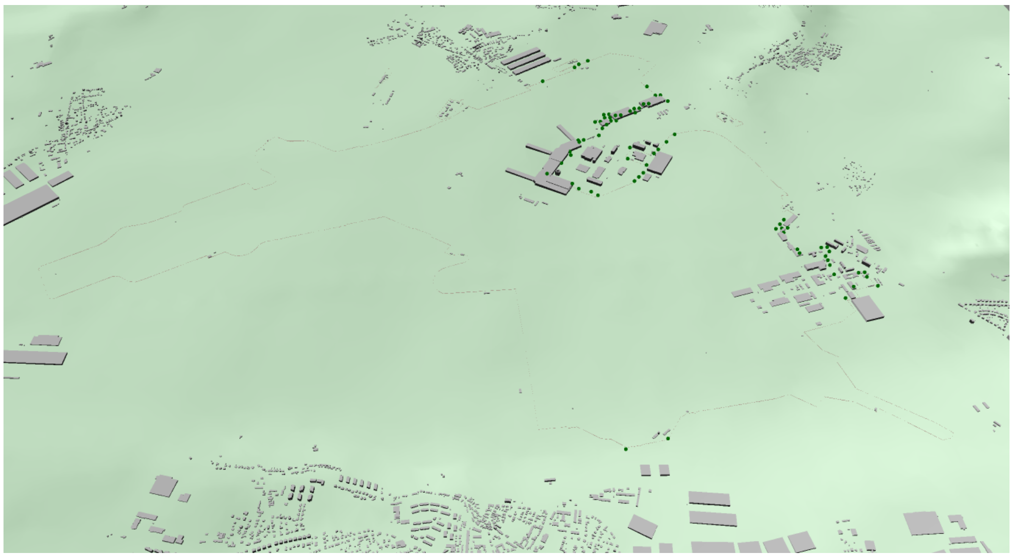

Figure 5.2: The sector of the Václav Havel Airport Prague showing the world cells in which sensor spots suitable for acoustic sensors are located as green points.

5.2 Sensor coverage

Back in subsubsection 4.1.1.3, we described how sensors are modeled in *visio* and explained how the scanning methods work, namely the *fullscan* method for camera sensor and *scan* method for acoustic sensor. Now, we will introduce the implementation of the sensor's coverage, how it is computed, and how it deals with a real-world environment. We will start off with the camera sensor (further only as 'camera') where the calculation of coverage is more complex, followed by the acoustic sensor where it is relatively simple.

5.2.1 Camera

To get as close to reality as possible, we had to account for occlusions, i.e. real-world obstacles standing in the camera's view and limiting its visibility, which made it a little bit more challenging task.

We came up with a relatively computationally demanding but very precise solution. A brief outline of the idea is, since the *fullscan* method can be used to compute intersection distance with the environment model for each pixel of camera's resolution (i.e. to compute so-called *depth profile*), to decompose the surface of each world cell into a set of 2D triangles, project the respective triangles to camera's screen space (often referred to as 'window space' too) obtaining pixel coordinates of their vertices, rasterize them and then compare the intersection distances of the depth profile and rasterized triangles at the pixel level.

We will now describe individual steps in more detail in the subsequent sections.

5.2.1.1 Reducing the space

As the projection and rasterization are relatively costly operations in the context of a large number of iterations, we try to reduce the number of cells on which the operations are performed to a minimum beforehand. First of all, we collect relevant cells that could possibly be covered by the camera by performing a *circlic expansion* from the camera's position and with the radius of the camera's range. The range of the camera is determined based on the detection thresholds that we describe in Subsection 5.2.3. This operation reduces the number of cells to a fraction very efficiently, which resulted in huge performance gains as otherwise there was no way to identify which cells are relevant and we had to work with the whole world.

By computing the camera's viewing frustum[3], we further filtered out cells whose centers do not lie within the area defined by the camera's viewing frustum. Given the camera's viewing frustum f defined by a set of planes $\{p_i \mid i \in 1 \dots 6\}$ and their normals, and a center point of a cell c, we determine whether the point c lies within frustum f using the following formula:

$$liesWithin = \mathbb{I}\left\{\sum_{i=1}^{6} \mathbb{I}\{p_i.normal \times c + p_i.d \leq 0\} = 6\right\} \tag{5.1}$$

where $p_i.normal$ is the normal of i-th plane and $p_i.d$ is the offset of i-th plane.

In the end, we check whether any cells remained at all, and in that case, we compute the *depth profile* of the environment using the *fullscan* method.

5.2.1.2 Projecting cells to screen space

Now when we have reduced the number of cells to as little as possible, we can move on to projecting those cells to the camera's screen space. Since we will need to rasterize those cells in the next step, we apply the projection rather on their individual triangles to be able to use triangle rasterization algorithms.

In computer graphics, when projecting a 3D object on a flat 2D surface such as the monitor, the object's coordinates need to go through a series of transformations which is known as *model view projection*. In model view projection, the object's coordinates are firstly transformed using the *model matrix* that defines the transformation from its local coordinates, i.e. a coordinate system with origin in object's position, to *world space*, which is a coordinate system relative to a global origin, i.e. is the same for all objects in the scene. Then, transformation using *view matrix* is applied, transforming the coordinates from *world space* to *view space* (also known as *eye space* or *camera space*), which can be understood as the space as seen from the camera's point of view. Next, the coordinates are transformed using the *projection matrix*, after which they are in *clip space*. Since we want to gain perspective, i.e. we want the more distant objects to appear smaller than close objects, we have to apply perspective division, i.e. divide the x, y and z components of the coordinates by their homogeneous w component, after which the coordinates result in normalized device coordinates (NDC). In NDC, all coordinates visible by the camera must fall between range -1 and 1 for every axis. The last transformation in the chain is the *viewport transformation*, which finally transforms the NDC coordinates to *screen space* (also known as *window space*) [46, 47]. By applying this chain of transformations to cell's triangles, we obtain exact pixel coordinates of their vertices that we use during rasterization in the next step.

Figure 5.3 shows how the chain of transformations is executed in the OpenGL library.

[3] `https://en.wikipedia.org/wiki/Viewing_frustum`

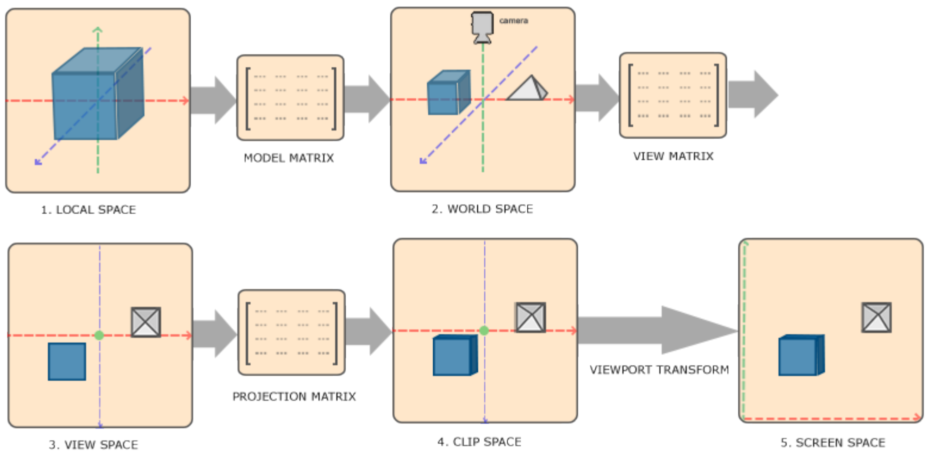

Figure 5.3: Chain of transformations in OpenGL[4]

5.2.1.3 Rasterizing cells

In the previous step, we obtained pixel coordinates and depth for each triangle vertex of each cell. Now we want to compute coordinates and depth of all pixels that are within the triangles, which is referred to as *triangle rasterization*. For such, we have used the Bresenham's algorithm [48].

The general idea of the algorithm is to firstly rasterize the line segments between pairs of triangle vertices using the Bresenham's line algorithm[5]and then rasterize the rows between pixels on different line segments but with the same y coordinate. For better understanding, we illustrate the idea in Figure 5.4.

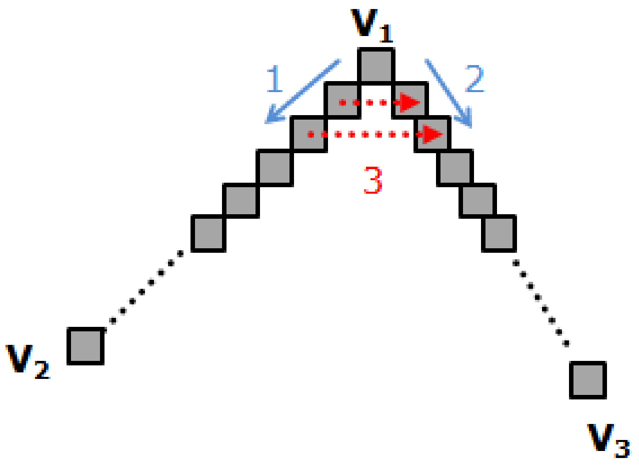

Figure 5.4: The idea of the Bresenham's algorithm[6]

To make this approach work, we firstly have to split the triangle in two, a flat-bottom triangle t_1 and a flat-top triangle t_2, which we achieve by sorting the vertices v_1, v_2, v_3 so that v_1 is the top-most vertex and and v_3 is the bottom-most vertex (i.e. descending according to their y coordinate) and computing additional vertex

[4]image taken from https://learnopengl.com/Getting-started/Coordinate-Systems

[5]https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm

[6]image taken from http://www.sunshine2k.de/coding/java/TriangleRasterization/TriangleRasterization.html

v_4 as

$$v4 = (v_1.x + \frac{v_2.y - v_1.y}{v_3.y - v_1.y} \cdot (v_3.x - v_1.x),\ v_2.y)$$

After that, the flat-bottom triangle is defined as $\triangle v_1 v_2 v_4$ and the flat-top triangle t_2 is defined as $\triangle v_2 v_4 v_3$, as is shown in Figure 5.5.

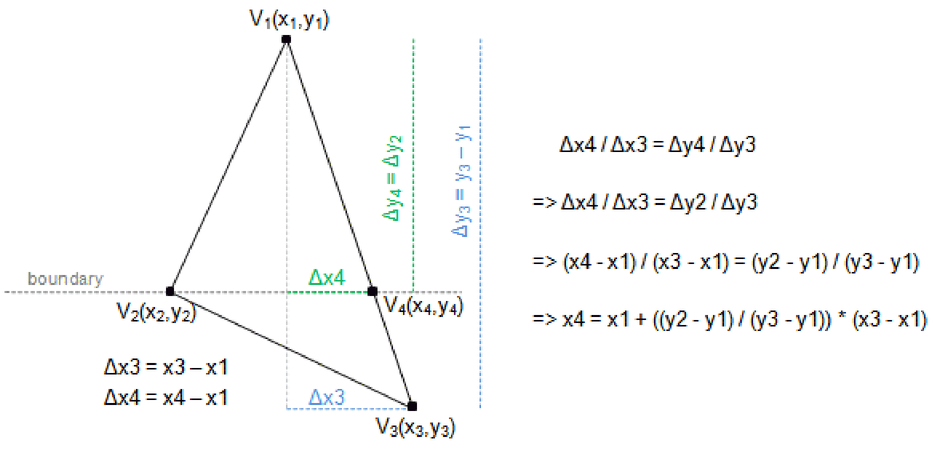

Figure 5.5: Division of the triangle to flat-bottom and flat-top triangle[7]

Thereafter, in the flat-bottom triangle t_1 for example, we rasterize the line segments v_1v_2 and v_1v_4 using the Bresenham's line algorithm, and for each pixel on v_1v_2, we find the corresponding pixel on v_1v_4 with the same y coordinate and rasterize the row between them. In case there are multiple pixels with the same y coordinate on either of the line segments, which can happen if the slope is too shallow, we only rasterize the row between the 'inner-most' pixels (i.e. the right-most pixel of the line segment to the left and the left-most pixel of the line segment to the right), otherwise, we would rasterize some pixels twice.

We must also not forget the depth, which we interpolate both during the rasterization of the line segments v_1v_2 and v_1v_4, and the following rasterization of rows between pixels with same y coordinate.

We repeat the same process for the flat-top triangle t_2 and only exclude the line segment v_2v_4 from the resulting set of pixels since it is common for both triangles and was rasterized twice.

In the end, we have rasterized the whole triangle and computed depth for each pixel, which we do for every triangle of each particular cell.

5.2.1.4 Evaluating the coverage

Through the rasterization of the cells, we obtained a set of pixel coordinates and their depths for each triangle of each cell, which we will use to evaluate their coverage, i.e. compute the percentage of pixels that are visible by a particular camera.

For each cell, we will record hits and misses for each rasterized pixel, where a hit means the pixel is visible by the camera and a miss means the pixel is not visible. Firstly, we 'clip' the set of pixels to the camera's resolution, i.e. we exclude those pixels whose coordinates are either negative, go beyond the camera's resolution or their depth is negative. Those pixels are either behind the camera or simply out of its field of view and thus are obviously counted as misses.

[7]image taken from `http://www.sunshine2k.de/coding/java/TriangleRasterization/TriangleRasterization.html`

Now, when we ensured that each of the remaining pixels is within the camera's resolution as well as in front of the camera, we compare each pixel's depth to the value of the corresponding pixel in the depth profile that we computed earlier. If the pixel's depth in the depth profile is less than the depth of the rasterized pixel, it means that there must be some obstacle standing in the way. We could prove it by casting a ray through that pixel and observing that it would hit the environment model first. In such a case, we count the pixel as a miss. In the other case, the ray ends up hitting the cell first and thus is counted as a hit.

At the end of the process, we have two numbers recorded, the number of misses and the number of hits. From those numbers, we simply calculate the percentage of hits, which, if it is greater than some predefined threshold, we evaluate the specific cell as covered. The threshold value is set to 50% within the program so the cell must be at least half-visible in order to be considered covered.

5.2.2 Acoustic sensor

In the case of the acoustic sensor, the process of coverage calculation is much more simple. In contrast with the camera sensor, we do not have to account for obstacles because it is already done by the acoustic sensor itself, using the reflections from the surface that we described in Section 4.1.1.3. Notwithstanding that the approach with cell projection and rasterization would not work very well either. Since the rays are cast from the sound source and not from the position of the sensor as it is in case of the camera, we would need to run the *scan* for each rasterized pixel, which would be very inefficient and time-consuming. For that reason, we perform the *scan* only from the center of each cell and each of its corners. As we aim to keep the cells as small as possible so that the discretized environment copies the reality as accurately as possible, it is also sufficient for us.

As the computations during the detection by the acoustic sensor are more expensive than those during the detection by the camera, we first reduce the number of cells to a minimum too, using the *circlic expansion* from the sensor's position and with the radius of sensor's range. As no orientation or FOV is defined for the acoustic sensor, we are unfortunately not able to reduce the space anymore.

Then, for each cell that was collected by the expansion, we perform a *scan* from the cell's center and each of its corners, with the initial power level of 73 dB and frequency of 8 kHz, which we use as default values for the detection as we can not determine in advance the exact power level and frequency of the drone we are supposed to detect. We selected these values based on the article by Haye Kesteloo [49], in which he compares the noise levels of various propellers commonly installed on popular DJI drones. Then, if at least half of the scans (that is 5 out of 9 scans) were successful, i.e. the rays emitted from the sound source reached the sensor with enough power level, we evaluate the particular cell as covered.

5.2.3 Detection thresholds

Before we start computing the coverage of a sensor, we need to reduce the number of cells to a minimum, as the determination of whether a cell is covered by the sensor is a computationally demanding process, for both cameras and acoustic sensors. For such, we use the *circlic expansion* that collects all cells within the range of the sensor, i.e. all cells that can possibly be covered by the sensor, regardless of the sensor configuration.

If the range of the sensor was overestimated, we would need to compute the coverage even for the cells farthest cells for which it is not needed actually, as a drone could not be detected at that distance anyway. For that reason, the real range of the sensor needs to be thoroughly estimated, i.e. the range at which is the sensor still able to detect the drone, and in the case of the camera, to recognize it as well.

Camera sensor. To determine the range of the camera, we consider a scenario in which a drone is put in the camera's view. By casting a ray through each pixel of the camera's resolution using the *fullscan* method, we obtain the depth profile of the camera's view, in which we are able to tell at which pixels the drone was detected and what was the number of such pixels. Then we process the depth profile into an image, in which we color separate the pixels that hit the environment and the pixels that hit the drone. Based on the image, we can determine whether it is recognizable by a human eye that the detected object on the image is a drone. By positioning the drone at different distances and examining the images, we estimate the real range of the camera for various resolutions.

We illustrate the scenario in the following figures. In Figure 5.6, we show a camera with full HD resolution (1920 x 1080 pixels) and 30° FOV. The blue transparent cone visualizes the view of the camera, in the middle of which is a drone that is 90 m away from the camera.

In Figure 5.7, we show the detection of a very small drone, whose size is approximately 44 cm x 40 cm x 11 cm, in the distance of 90 m from the camera. The drone was detected with 35 rays and as can be seen on the image, it is still relatively recognizable that it is a drone.

Obviously, the larger the drone, the easier it is to detect it. To account for various drone sizes, we further show the detection of a medium-sized drone, whose size is approximately 69 cm x 69 cm x 40 cm, in Figure 5.8. The drone is 140 m away from the same camera as in the previous case.

As for the large drones, we show the detection of a very large drone, the DJI Matrice 600 Pro, whose size is approximately 166 cm x 151 cm x 76 cm, in Figure 5.9. In this case, despite the drone being 400 m away from the camera, it is still recognizable as it is detected by 34 rays. The resolution and FOV of the camera is the same as in the previous cases.

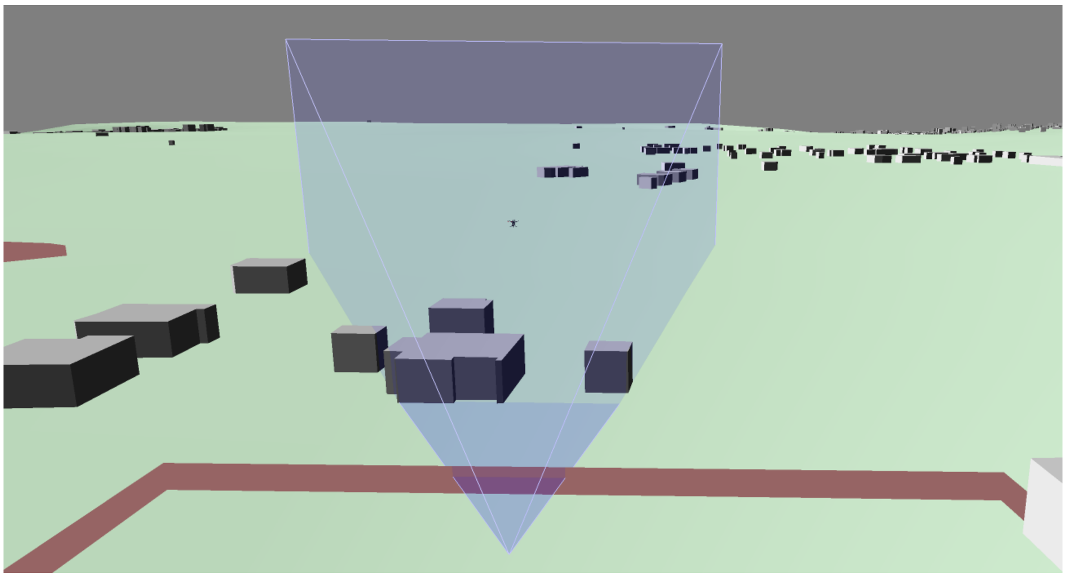

Figure 5.6: A drone in the view of a camera

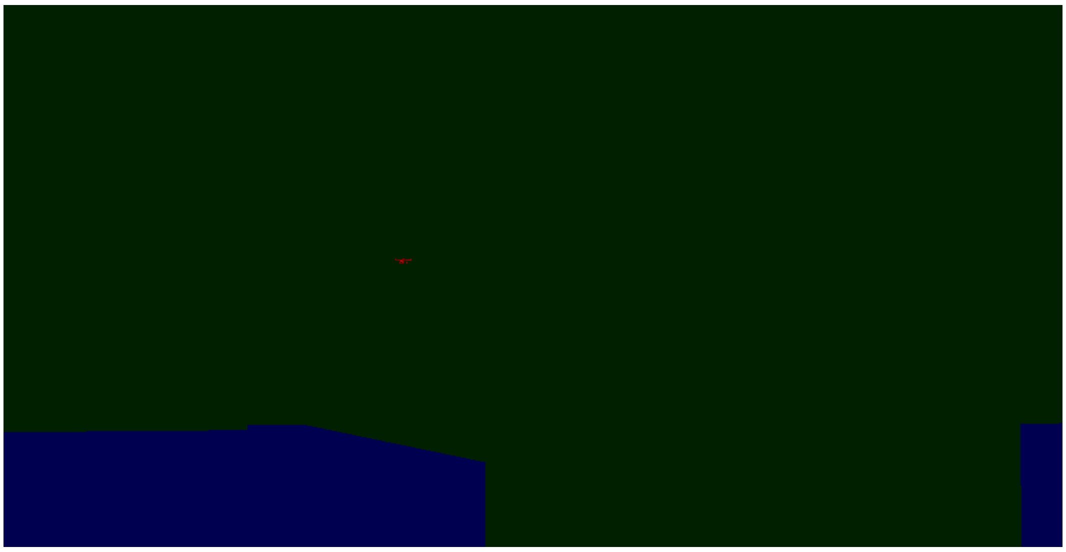

Figure 5.7: Detection of a small-sized drone 85 m away from the camera with full HD resolution and 30° FOV. The number of detected pixels is 35.

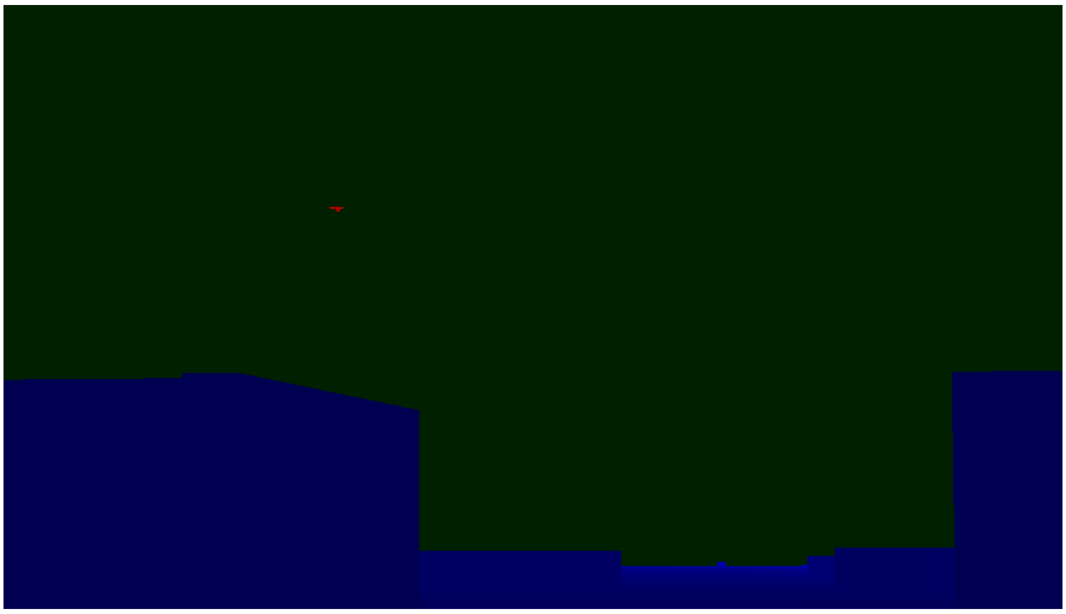

Figure 5.8: Detection of a medium-sized drone 140 m away from the camera with full HD resolution and 30° FOV. The number of detected pixels is 34.

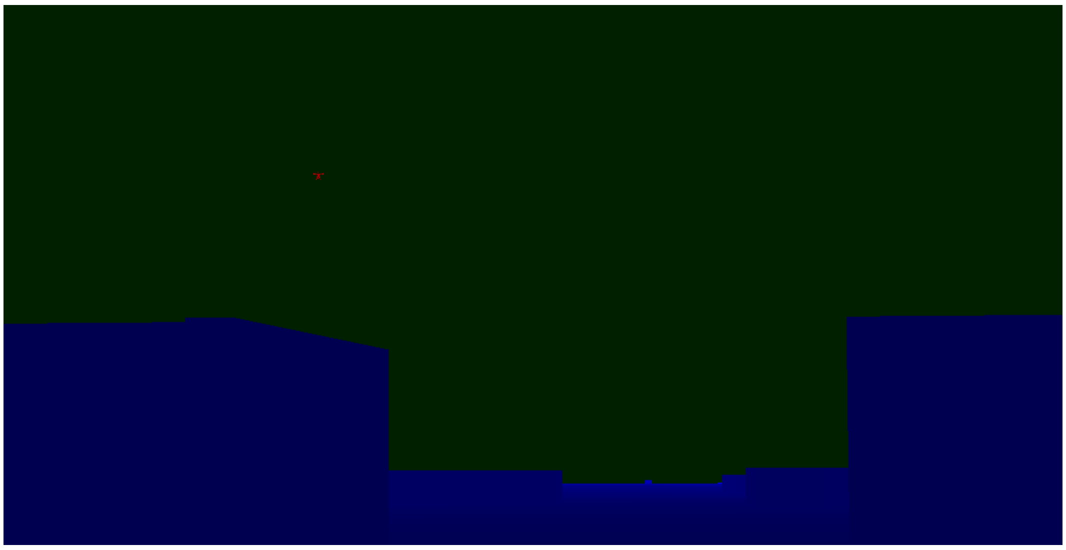
Figure 5.9: Detection of a large drone 400 m away from the camera with full HD resolution and 30° FOV. The number of detected pixels is 34.

Based on the detections presented in the previous figures, and the other detections presented in Appendix B, we determined that for the drone to be recognizable by the camera, it must be detected with approximately 30 to 40 rays, which results in the maximum detection range of around 85 m for small-sized drones, approximately 140 to 150 m for medium-size drones and around 250-400 m for drones of large size.

In reality, it is obvious that we can not know in advance which types or rather which sizes of drones will try to violate the safety zone around the monitored sector. In order not to overestimate the coverage of the sensors and also because we expect the majority of drones would not be exactly professional drones of large sizes such as the DJI Matrice 600 pro, but rather drones of small or medium size, we will use the range of 130 m for the calculation of coverage of the camera sensor, assuming that the resolution is full HD.

Acoustic sensor. In the case of the acoustic sensor, the detection range is rather based on the power level and frequency of the drone than on its size. Again, as we can not predict the power level and frequency of the intruder drones in advance, we need to consider default values, which are 73 dB for the power level and 8 kHz for the frequency, as already mentioned in Subsection 5.2.2.

To estimate the detection range of an acoustic sensor, we simply put a sensor on a position, calculate its coverage based on its configuration and determine its detection distance as the distance to the farthest detected cell. In Section 4.1.1.3 we described the parameters of the acoustic sensor, such as the air properties, sensitivity, and trace properties. As for the air properties, we use the value of 101325 Pa (often considered the standard pressure[8]) for the air pressure, 22 °C for the air temperature, and 20 % for the air humidity. The trace properties include the number of initial rays, number of reflection rays, and reflection depth, for which the default values we use are 100, 20, and 3 respectively. Increasing these values was not worth it, as it did not increase the detection range by much and made the computation of coverage rather slow, especially when the maximum reflection depth was increased.

[8]https://en.wikipedia.org/wiki/Standard_atmosphere_(unit)

The parameter that has the most significant impact on the acoustic sensor performance is the sensitivity, i.e. the minimum power level of the sound that must reach the sensor in order to be detected. We have chosen the possible values of power levels to be 40 dB, 30 dB, and 20 dB, as they represent ordinary daily noises rather in the lower volume spectrum[9] that could roughly correspond to a flying drone in the distance.

In Figure 5.10, Figure 5.11 and Figure 5.12, we show the detection ranges measured for the acoustic sensor with the power level of 40 dB, 30 dB, and 20 dB respectively. The purple point represents the acoustic sensor and the green points represent the world cells covered by the sensor. Based on these values, we decided to use the power level of 35 dB for the calculation of coverage of the acoustic sensor, resulting in the detection range of 145 m.

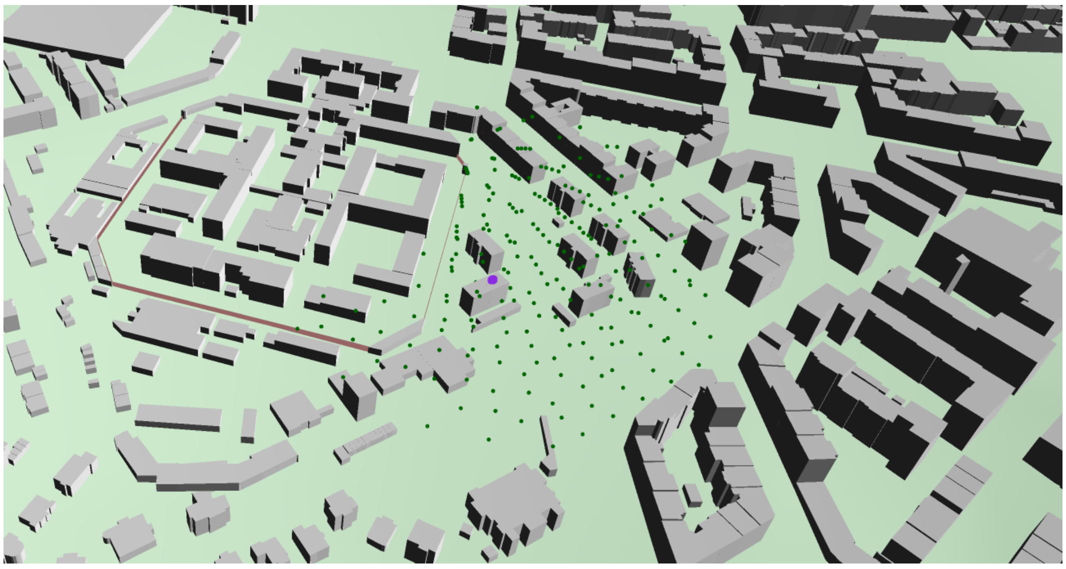
Figure 5.10: Coverage of acoustic sensor with a minimum power level of 40 dB. The farthest detected cell is 139 m away from the sensor.

[9]https://www.cdc.gov/nceh/hearing_loss/what_noises_cause_hearing_loss.html

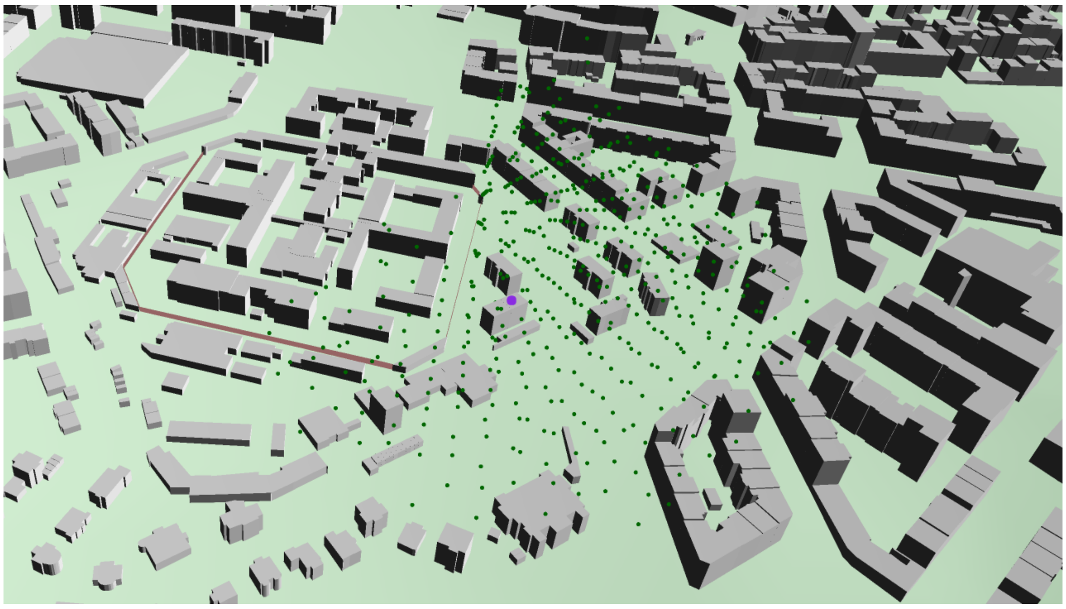
Figure 5.11: Coverage of acoustic sensor with a minimum power level of 30 dB. The farthest detected cell is 190 m away from the sensor.

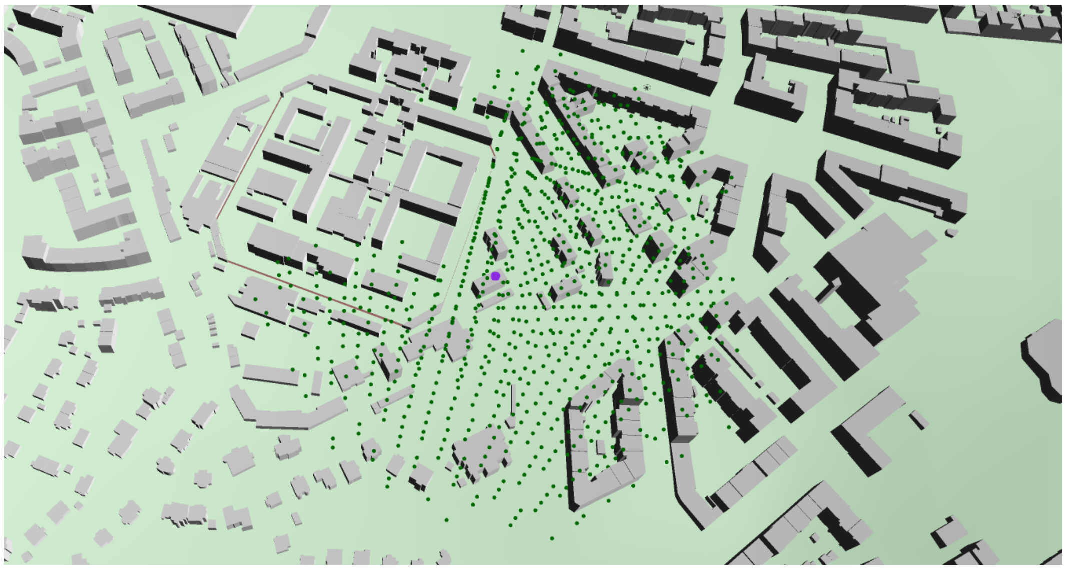
Figure 5.12: Coverage of acoustic sensor with a minimum power level of 20 dB. The farthest detected cell is 236 m away from the sensor.

5.3 Sensor placement

Having covered all the important features and components of the simulation in the previous sections, we can advance to the implementation of the proposed sensor placement algorithm, which is based on MILP with dynamic updating of cell weights based on the current coverage level of the given cells.

5.3.1 Input

In order to be able to formalize the algorithm as a MILP model, we needed to provide it with a discrete environment, which included the transition from continuous space to discrete space, as we already described in Subsection 4.4.2. As the result, we have the so-called *world* represented by a set of cells $\{C_i \mid i \in 0 \dots N-1\}$.

Furthermore, we defined a set of sensor spots $\{S_j \mid j \in 0 \dots M-1\}$, that is either loaded from a text file provided in the configuration, which is the preferred option or computed automatically from the perimeter of the sector. Every sensor spot can be equipped with a camera but only specific sensor spots can be equipped with acoustic sensors.

Sensors have their configuration as well which we have to recognize in the sensor placement. In the case of the camera sensor, its main property we are interested in is the camera pose, combining two properties, panning, and tilting. Currently, the possible values for panning and tilting are hard-coded and include $\{0, 15, \dots, 330, 345\}$ for panning and $\{-10, 0, 10\}$ for tilting in degrees, which gives us a set of 72 camera poses $\{P_k \mid k \in 0 \dots K-1\}$, where $K = 72$.

Last but not least, we define a weight function $\{f_i^r \mid i \in 0 \dots N-1,\ r \in 0 \dots R\}$ where r represents the current coverage level of the particular cell and R is the value of the *redundancy* parameter as described in Subsection 4.3.8. The weight function f stores the values of the weights of the cells based on their coverage level in decreasing order, i.e. the lower the coverage level, the greater the weight. However, as the weight is zero when the coverage level of R is reached, it only stores values for coverage levels less or equal to R. Also, as the stored values are not supposed to change during the execution of the program, they are computed in advance and stored in an array.

Finally, the remaining parameters to the sensor placement such as the budget, time limit, or minimal required coverage, are omitted in this section, as they were already presented in Subsection 4.3.8.

5.3.2 Precomputation phase

Before we start computing the sensor placement, we have to go through the so-called *precomputation phase*. In this phase, we precompute coverages for all possible sensor configurations in advance, so that the algorithm only focuses on the optimization of their placement. In the case of a camera, we have to consider every sensor spot $s \in S$ and every camera pose $p \in P$, giving us $|S| \cdot |K|$ configurations in total. In the case of an acoustic sensor, we only consider the sensor spots that were identified as appropriate candidates during the *uneven surface expansion* that we described in Subsection 5.1.2.

Although the precomputation of coverages for such a large number of sensors is a time-consuming operation, we can not limit or skip any variables in this phase, as even a single omitted sensor configuration could affect the resulting sensor placement in a negative manner. Hence, we decided to parallelize the process in order to use the maximum of available computing power and more importantly, to make the process less time-consuming.

At first, we determine the number of worker threads *nc* that will run simultaneously based on the number of available CPU threads which we get from the Runtime API[10]. Based on that, we split the set of sensor spots S into equal batches, except for the last batch which can have different sizes. After that, the main thread distributes the batches of sensor spots to respective worker threads, of which each asynchronously computes the coverages for sensors on each sensor spot in its batch and with each camera pose, returning an instance of the Future[11] (further referred

[10]`https://docs.oracle.com/javase/8/docs/api/java/lang/Runtime.html`
[11]`https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html`

to as *future*) which will eventually contain the computed coverages in some point of time in the future, once the computation is finished. After all worker threads finish their computations, the main thread retrieves the computed coverages from the futures and the precomputation phase is finished.

We present the pseudocode of the precomputation phase for cameras in Algorithm 1 and Algorithm 2. The pseudocode for the precomputation phase for acoustic sensors is omitted as it is basically the same.

Algorithm 1 precomputeParallel

```
Input: world W, sensor spots $S_0 \dots S_{M-1}$, camera orientations (poses) $P_0 \dots P_{K-1}$
Output: set of precomputed coverages for all possible cameras $C_0 \dots C_{M \cdot K-1}$
 1: $nc \leftarrow$ Runtime.availableCores()        ▷ Get number of available CPU threads
 2: $batch \leftarrow \lceil \frac{M}{nc} \rceil$        ▷ Batch size, i.e. number of sensor spots per thread
 3: $tasks \leftarrow \emptyset$        ▷ Set of tasks to be invoked asynchronously
 4: for $i \leftarrow 0$ to $nc$ do
 5:     $sf \leftarrow i \cdot batch$
 6:     $st \leftarrow sf + batch$
 7:     $tasks \leftarrow tasks \cup$ asyncTask($W$, $S$, $P$, $sf$, $st$)        ▷ Initialize tasks
 8: end for
 9:
10: $futures \leftarrow$ invoke($tasks$)        ▷ Invoke tasks and wait until all finished
11: $C \leftarrow \emptyset$
12: for each $f \in futures$ do        ▷ Retrieve results from the futures
13:     $C \leftarrow C \cup f.get()$
14: end for
15: return $C$
```

Algorithm 2 asyncTask

```
Input: world W, sensor spots $S_0 \dots S_{M-1}$, camera poses $P_0 \dots P_{K-1}$, range of sensor
    spots $[sf, st)$
Output: set of precomputed coverages for cameras on sensor spots in range $[sf, st)$
    $C_0 \dots C_{(st-sf) \cdot K}$        ▷ Set of covered cells
 1: $C \leftarrow \emptyset$
 2: for $j \leftarrow sf$ to $st - 1$ do
 3:     for $k \leftarrow 0$ to $M - 1$ do
 4:         $cam \leftarrow$ getCamera($S_j$, $P_k$)        ▷ Initialize camera sensor
 5:         $cells \leftarrow$ W.circlicExpansion($S_j$, $cam.range$)        ▷ Collect relevant cells
 6:         $C \leftarrow C \cup cam$.getCoverage($cells$)        ▷ Compute and get coverage
 7:     end for
 8: end for
 9: return $C$
```

5.3.3 MILP model

A simple approach to the sensor placement using ILP would be to maximize the space coverage (i.e. number of covered cells) under some constraints such as the maximum price for example. If we knew in advance that we can achieve full coverage, i.e. cover each of the cells, we could also set the objective to minimize the price required to ensure full coverage.

In our case, however, none of the above-mentioned approaches is applicable. In the latter case, we can not guarantee that the whole space can be covered unless firstly computing the sensor placement with no constraints (i.e. with infinity price limit), which would require another model to be optimized. Furthermore, our approach extends the maximization of coverage by considering possibly different and non-integer weights for each cell, so if we only optimized the number of covered cells, we would completely discard the idea of priority areas. As some of the cell weights might be non-integer, the variable representing the cell weights must be non-integer as well, from which it follows that our model is not ILP, but MILP.

Similarly, as the precomputation phase, our MILP model works with a set of discrete cells C, a set of sensor spots S, and a set of camera poses P. Additionally, we must provide the model with the weight function f, to be able to determine the weights of the cells during the optimization. Last, we need to transform the precomputed sensor coverages to a binary indicator, using which we can easily find out which cells are covered by which sensors.

Given N cells, M sensor spots and K camera poses, we define two binary indicators, for acoustic sensor and camera sensor respectively:

$$g_{j,i} = \begin{cases} 1, & \text{if the acoustic sensor placed on sensor spot } j \\ & \text{covers cell } i \\ 0, & \text{otherwise} \end{cases} \tag{5.2}$$

$$h_{j,k,i} = \begin{cases} 1, & \text{if the camera sensor placed on sensor spot } j \\ & \text{with camera pose } k \text{ covers cell } i \\ 0, & \text{otherwise} \end{cases} \tag{5.3}$$

Regarding the optimization variables, at first, we define binary variables X and Y for the camera sensor and acoustic sensor respectively as follows:

$$X_{j,k} = \begin{cases} 1, & \text{if a camera sensor with camera pose } k \text{ is} \\ & \text{placed on sensor spot } j \\ 0, & \text{otherwise} \end{cases} \tag{5.4}$$

$$Y_j = \begin{cases} 1, & \text{if a acoustic sensor is placed on sensor spot } j \\ 0, & \text{otherwise} \end{cases} \tag{5.5}$$

Further, let C be an integer variable that states the coverage level of each cell:

$$C_i \in \mathbb{Z}^* \, , \; i \in \{0, \ldots, N-1\} \tag{5.6}$$

In relation to the previous variable C, we define an additional continuous variable W that holds the current weights of cells (based on their current coverage level):

$$W_i \in \mathbb{R}_0^+ \, , \; i \in \{0, \ldots, N-1\} \tag{5.7}$$

As we already mentioned, the weight of cells depends on their current coverage levels. However, since the coverage level is represented by an optimization variable, we can not use it as an index to the weight function f directly. Instead, we need to express the index as another binary variable T, which we constraint so that the sum of its values equals one for each cell. Since it will contain exactly one value equal to 1 and the rest will be zeros, the index at which the only value equal to 1 appears, is the index to the weight function f.

$$T_{i,r} = \begin{cases} 1, & \text{if the current weight of cell } i \text{ should equal to } f_i^r \\ 0, & \text{otherwise} \end{cases} \tag{5.8}$$

Moving on to model constraints, we start off with expressing the coverage variable C in terms of the sensor variables X and Y and the indicator functions g and h, which is done as follows:

$$C_i = \sum_{j=0}^{M-1} Y_j \cdot g(j,i) + \sum_{j=0}^{M-1} \sum_{k=0}^{K-1} X_{j,k} \cdot h(j,k,i), \quad \forall i \in \{0,\ldots,N-1\} \tag{5.9}$$

We also need to ensure that no more than one sensor is placed on each sensor spot:

$$\sum_{k=0}^{K-1} X_{j,k} + Y_j \leq 1, \quad \forall j \in \{0,\ldots,M-1\} \tag{5.10}$$

As we already said, the variable T must sum to one for each cell, which we ensure with the following constraint:

$$\sum_{r=0}^{R} T_{i,r} = 1, \quad \forall i \in \{0,\ldots,N-1\} \tag{5.11}$$

Now, as the variable T sums to one, we can select the correct weight from the weight function f by multiplying the binary variable T with the weight function f for each cell:

$$W_i = \sum_{r=0}^{R} f_i^r \cdot T_{i,r}, \quad \forall i \in \{0,\ldots,N-1\} \tag{5.12}$$

The last thing we have to deal with in order for the model to work is to specify a constraint that determines at which index of the variable T the value 1 should be put for each cell. It is obvious that the model will put 1 on R-th, the highest index, because f stores the values in descending order and we are minimizing. In such a case, we would end up with all cells having zero weight instantly and there would be nothing to optimize. Because of that, we need to limit it so that C, i.e. the current coverage level, is the highest index where 1 may be put. As we minimize, it will ensure that the model will put 1 on the C-th index and that is exactly what we want:

$$r \cdot T_{i,r} \leq C_i, \quad \forall i \in \{0,\ldots,N-1\},\ r \in \{0,\ldots,R\} \tag{5.13}$$

In the end, we only need to ensure that the sum of prices of the sensors included in the sensor placement does not exceed the budget using the following constraint, where $pc(j,k)$ and $pa(j)$ are functions that return the price of the camera placed on sensor spot j with camera pose k and the price of the acoustic sensor placed on sensor spot j, respectively.

$$\sum_{j=0}^{M-1} \sum_{k=0}^{K-1} X_{j,k} \cdot pc(j,k) + \sum_{j=0}^{M-1} Y_j \cdot pa(j) \leq \text{budget} \tag{5.14}$$

The objective is then to minimize the sum of cell weights:

$$\min \sum_{i=0}^{N-1} W_i \tag{5.15}$$

If we put those variables and constraints together, the resulting ILP model looks as follows:

$$
\begin{aligned}
\text{min} \quad & \sum_{i=0}^{N-1} W_i \\
\text{s.t.} \quad & C_i = \sum_{j=0}^{M-1} Y_j \cdot g(j,i) + \sum_{j=0}^{M-1}\sum_{k=0}^{K-1} X_{j,k} \cdot h(j,k,i) && \forall i \in \{0,\dots,N-1\} \\
& \sum_{k=0}^{K-1} X_{j,k} + Y_j \leq 1 && \forall j \in \{0,\dots,M-1\} \\
& \sum_{r=0}^{R} T_{i,r} = 1 && \forall i \in \{0,\dots,N-1\} \\
& W_i = \sum_{r=0}^{R} f_i^r \cdot T_{i,r} && \forall i \in \{0,\dots,N-1\} \\
& r \cdot T_{i,r} \leq C_i && \forall i \in \{0,\dots,N-1\},\ r \in \{0,\dots,R\} \\
& \sum_{j=0}^{M-1}\sum_{k=0}^{K-1} X_{j,k} \cdot pc(j,k) + \sum_{j=0}^{M-1} Y_j \cdot pa(j) \leq \text{budget} \\
& X_{j,k} \in \{0,1\} && j \in \{0,\dots,M-1\},\ k \in \{0,\dots,K-1\} \\
& Y_j \in \{0,1\} && j \in \{0,\dots,M-1\} \\
& T_{i,r} \in \{0,1\} && i \in \{0,\dots,N-1\},\ r \in \{0,\dots,R\} \\
& W_i \in \mathbb{R}_0^+ && i \in \{0,\dots,N-1\} \\
& C_i \in \mathbb{Z}^* && i \in \{0,\dots,N-1\} \\
& M,\ N,\ K,\ R \in \mathbb{Z}^*
\end{aligned}
$$

Chapter 6

Experimental evaluation

Although we can evaluate the sensor placement based on the dimension of space covered by the sensors, we further want to evaluate it in a dynamic and more realistic environment from which we can obtain additional and more detailed data about the quality of the sensor placement.

In this chapter, we present an experimental evaluation, with the help of which we obtain detailed data that we utilize in the analysis of the quality of the sensor placement and from which we can draw statistical conclusions. Firstly, we present the overall design of the experiment and the specific drone flight profiles included in the experiment. Then, we briefly introduce the flight path generator using which we were able to generate realistic trajectories for the respective flight profiles. Last, we present the scenarios of which the experiment consists, measured data, and the conclusions that follow from it.

6.1 Experiment design

To get a rough overview of how good the sensor placement is, we have to specify the metrics by which we can compare the sensor placements with each other. The simplest metric is to express the quality of the sensor placement based on the coverage level, i.e. the number of world cells covered by the sensor placement.

We introduce three different criteria using which the level of coverage can be expressed:

- **Absolute coverage.** The absolute value of coverage defined as $\frac{N_c}{N}$, where N_c is the number of covered cells and N is the total number of cells.

- **Absolute weighted coverage.** The value of coverage with respect to the priority areas defined as $\frac{\sum_{i \in C} W_i}{W}$, where C is a set of covered cells, W_i denotes the initial weight of the i-th cell and W is the sum of weights of all cells.

- **Relative coverage.** The value of coverage with respect to the actual coverage level of each world cell, defined as $1\frac{\sum_{i=0}^{N-1}(W_i - w_i)}{W}$, where W_i denotes the initial weight of the i-th cell, w_i denotes the actual weight of the i-th cell (i.e the weight after the deployment of the sensor placement) and W is the sum of weights of all cells. As opposed to the previous two cases, only this approach takes into account the actual coverage level of each cell.

Using the values produced by those three criteria, we can tell to some extent whether one sensor placement covers more space than the other and thus whether it performs better. However, we can not draw any further conclusions from it, such as how good it will perform in a real drone detection task. For that reason, we need to investigate deeper into the evaluation process of the sensor placement.

To make the evaluation thorough and proper, we tried to design the experiment so that it reflects the reality as much as possible, by simulating a real-world event in which the protection against drones is required. We assume a scenario, in which one or more drones with various purposes violate the safe zone around the monitored sector, which would typically be an airport or a prison. The drone might just be an ordinary drone flown by an ordinary person that accidentally happens to get too close to the sector, or also a dangerous drone with the purpose of spying, smuggling, or causing damage.

Since the information about what kind of drone it is and what is its purpose is something that can not be known in advance, we have to ensure protection in all cases. In order to cover all types of flights in the experiment, we introduce four flight profiles in total, namely the flight profiles of **attacking**, **smuggling**, **spying** and finally the **ordinary** flight profile, which is a flight profile without any harmful purpose. To not overlook the priority areas in the experiment, in the flight profile of attacking and smuggling, there is a 60% chance that the drone will fly from the direction of one of the priority areas.

Flight profile of attacking. This flight profile represents an attack such as dropping a bomb into a critical area, e.g. a crowded place for example. The attacking drone is supposed to fly straight to the destination while keeping low above the ground level (AGL) so that it is hidden behind buildings or other obstacles such as terrain or vegetation when possible.

Flight profile of smuggling. This flight profile represents a smuggling flight, in which the drone is supposed to smuggle drugs or weapons into a critical area, such as the prison. The characteristics of this flight are the same as in the flight profile of attacking, with the only difference that after reaching the destination, the drone flies out of the critical area using the shortest path.

Flight profile of spying. This flight profile simulates the flight in which the drone spies on some object, such as private property or a restricted military object. As opposed to the previous two flight profiles, the drone is characterized by flying back and forth around the sector while keeping generally greater AGL.

Ordinary flight profile. Eventually, we wanted to include ordinary flights in the experiment as well, because such flights might actually be the most common. By an ordinary flight, we simply mean a drone flown by an ordinary or inexperienced person, without any harmful purpose. The path of the flight is completely random, i.e. consists of sudden movement changes, both horizontal and vertical.

We show the generated flight paths of the flight profile of attacking and smuggling in Figure 6.1, on which the sector of Pankrác Prison can be seen. The purple lines represent the flight profile of attacking, whereas the red lines represent the flight profile of smuggling. Given that there are two priority areas near the sector, there was a 60 percent chance that each of the flights would start from the direction of one of the priority areas. The AGL was chosen randomly from the interval from 10 to 20 m.

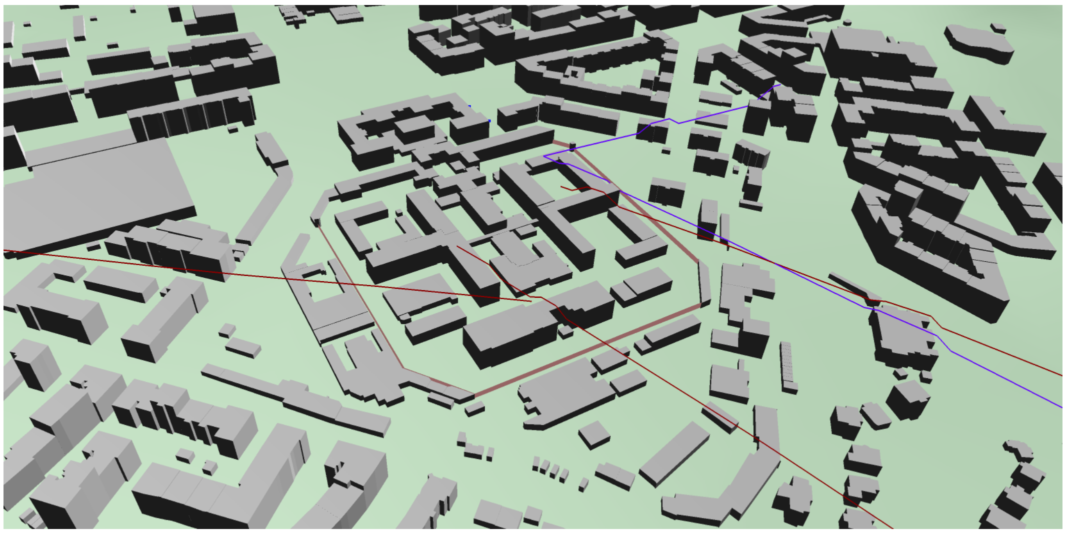
Figure 6.1: Flight profiles of attacking and smuggling

In Figure 6.2, we show the remaining two flight profiles, the flight profile of spying represented by the yellow lines, and the ordinary flight profile represented by the lines of light blue. In the case of the flight profile of spying, the AGL was chosen randomly from the interval from 35 to 80 m. The flight path of the ordinary flight profile is generated completely randomly.

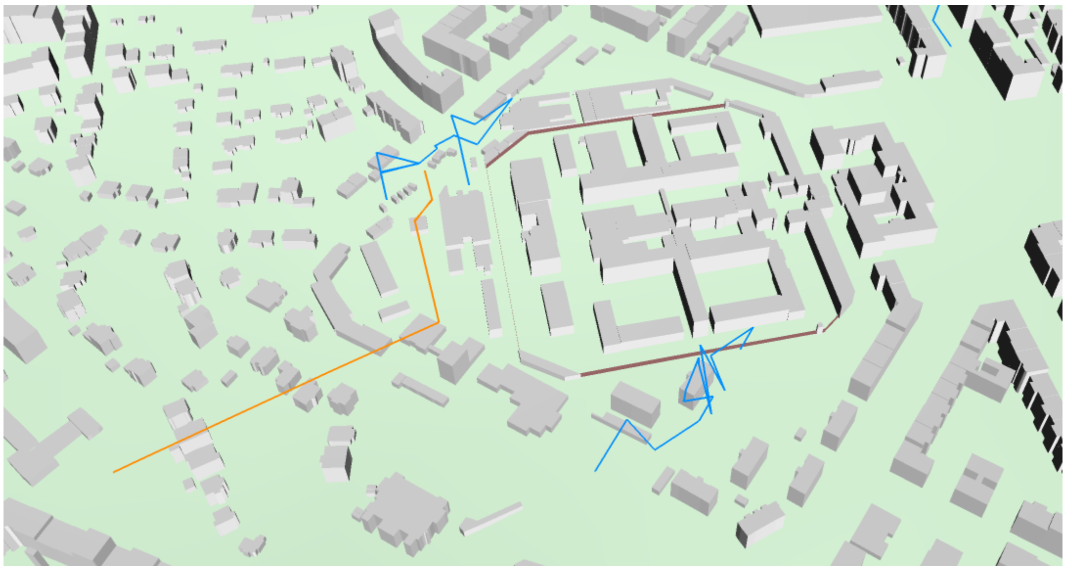
Figure 6.2: Flight profile of spying and the ordinary flight profile

6.2 Flight path generator

To avoid the need to generate the flight paths manually, even though such an option is also implemented in the program, we are using automatically generated flight paths in the experiment. Because some of the flight profiles require to keep certain AGL throughout the flight, it could easily happen that the trajectory of the generated flight path would lead through obstacles such as buildings, which will make it unrealistic and also invalid.

To carry off this unwanted possibility, we implemented a flight path generator, which, provided an initial set of waypoints and the value of desired AGL, generates a valid flight path such that it does not interfere with any obstacles while still connecting the original waypoints and keeping the desired AGL wherever possible. The flight path generator also considers another variable which is the so-called *above surface threshold* that states the minimum height above the surface in which a drone may fly.

Provided the initial set of waypoints, the flight path generator firstly validates that the initial waypoints are valid i.e. are not under the surface or inside an obstacle. If one of the waypoints is invalid, it sets its altitude to the surface elevation plus the *above surface threshold*, where the surface elevation is obtained using the ray tracing operations. Then, it continues with linearly interpolating the straight lines between each pair of consecutive waypoints by 1 m segments, where each segment has its altitude validated and eventually corrected similarly as with the initial waypoints.

When all waypoints are processed in this way, we end up with $N-1$ sets of 1 m segments that together form the whole trajectory of the flight path, where N is the number of initial waypoints. Such flight path is valid in most cases, but in case that the desired AGL is low, it might contain very steep height changes, such as those near the edges of buildings where the difference in height might be more than 10 m on 1 m horizontal segment. In addition to that, those steep height changes often cause the flight path to interfere with the edges of the buildings.

An example of such a flight path with low AGL and lots of steep height changes is shown in Figure 6.3.

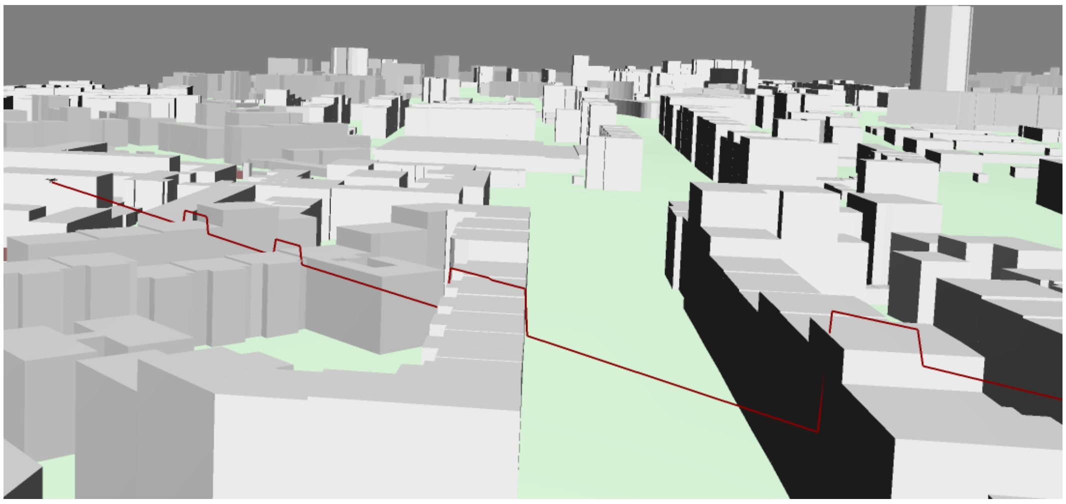
Figure 6.3: Flight path with steep height changes

Such a flight path would obviously be very unrealistic, not to mention it is possible that the drone would not even be able to fly safely along such trajectory. For that reason, we need to process the flight path once more, this time to smooth the trajectory, i.e. to get rid of the steep height changes and to create smooth and continuous transitions between the two positions with different heights.

The trajectory smoothing process is done by traversing the flight path by its segments until we come across a steep height change, in which case a smoothing maneuver is performed. Whether a height change is steep or not is determined by a threshold T that states the minimum difference in altitude for the height change to be considered steep. The maneuver can either be an *ascending maneuver* or a *descending maneuver*, however, in both cases, we try to keep its steepness angle at about 30° and its length at least 10 m when possible, to keep the maneuver smoothness. If it is not possible to ensure such conditions, for example when two buildings are too close to each other and the maneuver would not even fit between them, we rather increase the AGL on the given part of the trajectory so that the need for the maneuver is eliminated.

When an *ascending maneuver* needs to be performed at some point of the flight path traversal, we calculate the length of the maneuver as $max(10, \frac{\Delta_z}{\tan 30°})$, where Δ_z is the height difference of the maneuver, and traverse the trajectory backward by this length, to find the position where the maneuver should start. If we come across a position where the height difference is not greater than T anymore, we stop the backward traversal and set the maneuver start to the last position where the height difference was still greater than T. Unfortunately, this approach might not always meet the criteria of the desired steepness angle or maneuver length but is sufficient for our purposes.

The approach for the *descending maneuver* is very similar. The only difference is that as opposed to *ascending maneuver*, we know where the maneuver starts but not where it should end, so instead of performing a backward traversal, we perform a forward traversal.

The same flight path as in Figure 6.3 but with the smoothing process applied to it is shown in Figure 6.4.

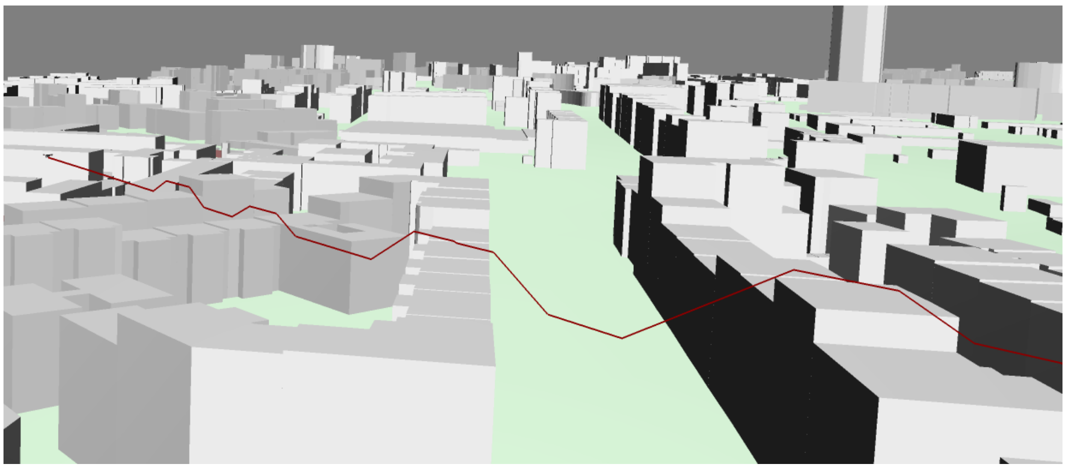

Figure 6.4: Flight path after the smoothing process

6.3 Evaluation

The quality of the sensor placement algorithm is evaluated on two scenarios in total. The first scenario takes place around a small sector of the Pankrác Prison in Prague, and the second scenario takes place around a large sector of the Václav Havel Airport Prague.

There were 200 flights generated for both scenarios in the experiment, however, the percentage distribution of the flight profiles was different for each scenario. For different sensor placement configurations, we measured the average detection distance, average detection percentage for each flight profile, and in the case of the flight profile of attacking and smuggling, in which the drones reach the critical area eventually, we also measured the average time to react, i.e. the time from the first detection of the drone before the drone enters the critical area.

The prices of the sensors that were used through the experimental evaluation were 2500 for the camera sensor and 55 000 for the acoustic sensors. Unfortunately, these prices are fictitious and do not correspond to reality.

In the following sections, we will provide a more detailed description of each scenario and the data measured from the experimental flights.

6.3.1 The Pankrác Prison scenario

In this scenario, the monitored sector is relatively small, owing to that the world cells could be small as well, i.e. 20 m x 20 m x 20 m. The horizontal monitor distance was set to 250 m and the vertical monitor distance was set to 170 m, which we considered as sufficient as only large drones can be detected at that distances, and thus the flights would be unnecessarily long for most of the drones. The total number of generated world cells was 9272.

Two priority areas were used in this scenario and the weight of the priority cells was 2, whereas the weight of non-priority cells outside the priority areas was 1. The coverage redundancy was set to 2, meaning that in the optimal case, each cell is covered by two sensors. Therefore, the values of the weight function f were (2, 1, 0) for the priority cells and (1, 0.5, 0) for the non-priority cells, for the coverage levels of 0, 1, and > 1 respectively. The possible values for camera panning and tilting were ($0°$, $15°$, $30°$, ..., $345°$) and ($0°$, $10°$, $20°$, $30°$, $40°$), respectively, resulting in total of 120 camera poses.

As the monitored sector is a prison, we assumed that the most common illegal activity with the use of drones is smuggling, so we chose a 40 % chance that the flight would have the flight profile of attacking. The ordinary flight profile had also 40 % chance, as a big part of drone incidents happens accidentally by ordinary people. As the flight profile of attacking and spying is not so common in the case of prison, we only gave them a chance of 10 %, thus the final distribution of the flight profile was 10%, 40%, 10%, 40% for the flight profile of attacking, smuggling, spying, and the ordinary flight profile, respectively.

The Pankrác Prison scenario is shown in Figure 6.5. There were 153 sensor spots in total, which are visualized as green points. We created those sensor spots manually because, in this scenario, we want to have sensors spots also on buildings that are inside the sector, which the method described in Subsection 4.3.4 would not be able to generate. Out of 200 generated flights, there was a total of 28 flight profiles of attacking, 82 flight profiles of smuggling, 12 flight profiles of spying, and 78 ordinary flight profiles. We show the generated flights in Figure 6.6.

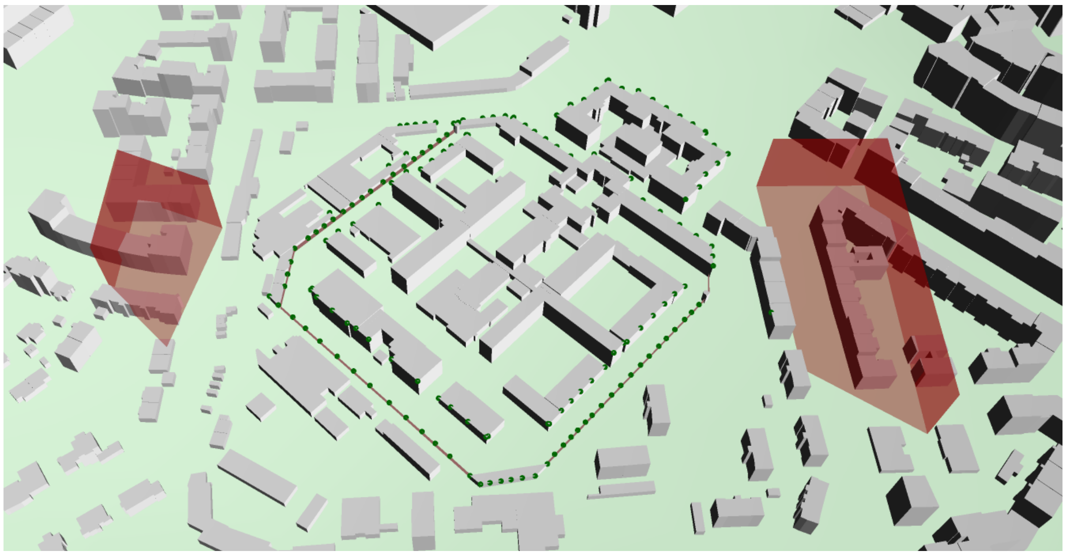
Figure 6.5: The Pankrác Prison scenario

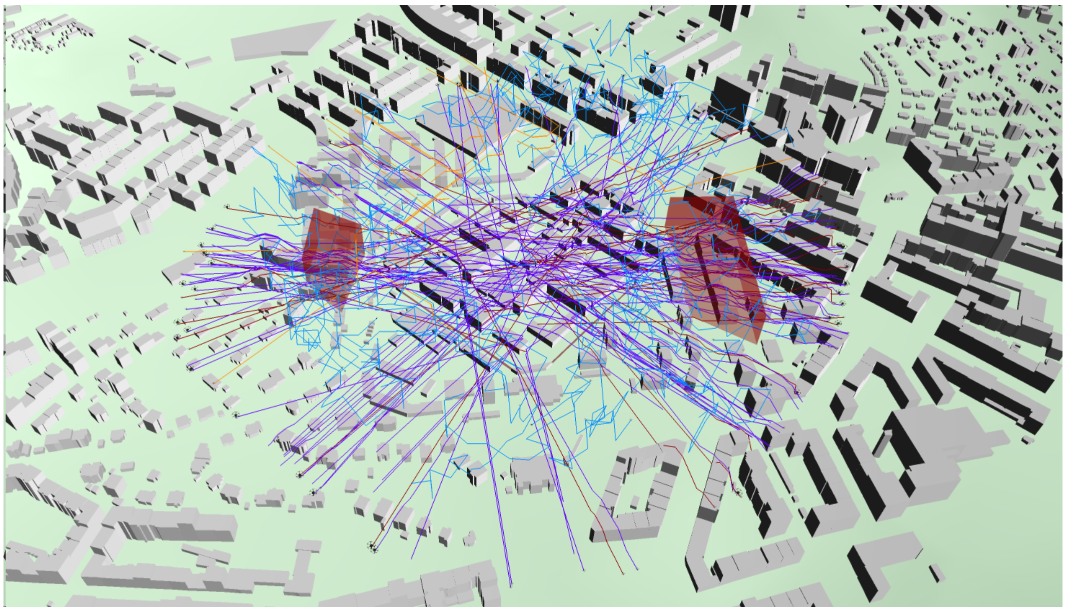
Figure 6.6: The Pankrác Prison scenario flights

In Figure 6.7 we illustrate a sensor placement in the sector of the Pankrác Prison, with the maximum price of 75 000. The covered cells are represented as green points. There is 1 acoustic sensor placed on the very corner of the building, covering a significant amount of both priority and non-priority cells. As the coverage redundancy is 2, there are further 2 cameras covering the same priority cells as does the acoustic sensor. Another three cameras focus on the second priority area. The remaining three cameras are placed so that they do not overlap and thus cover most of the cells not yet covered. In Figure 6.8 we show a much more dense sensor network with a maximum price of 300 000, consisting of 3 acoustic sensors and 54 cameras.

We show further examples of different sensor placements in this scenario in Appendix C.

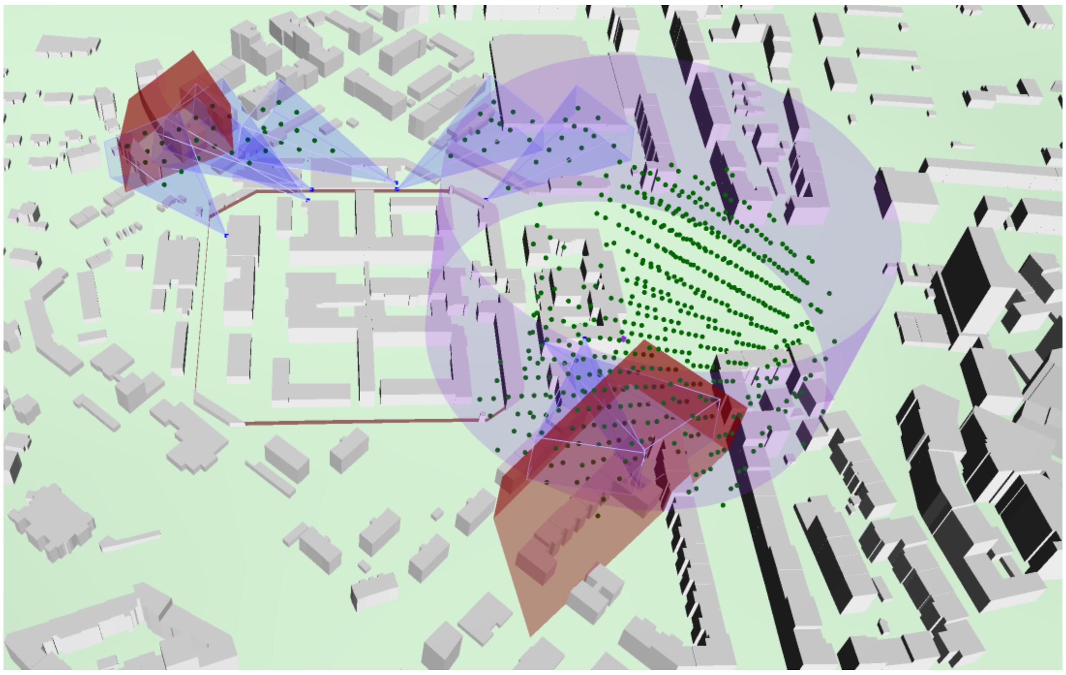

Figure 6.7: Sensor placement in the sector of the Pankrác Prison with the maximum price of 75 000

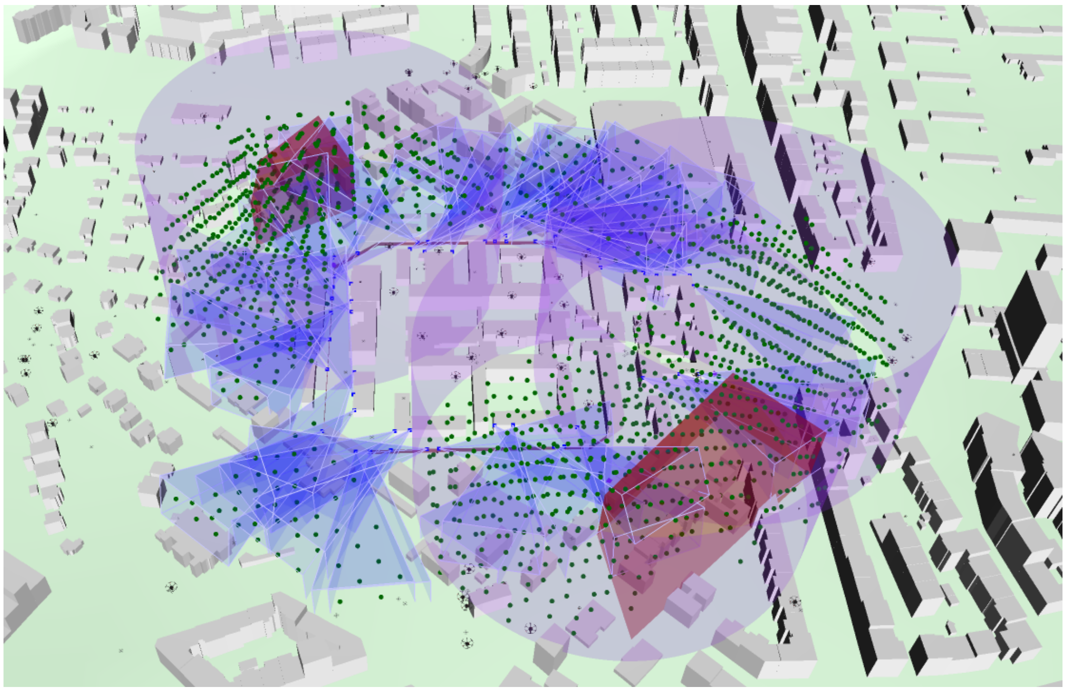

Figure 6.8: Sensor placement in the sector of the Pankrác Prison with the maximum price of 300 000

In Figure 6.9 we show the graph of average detection percentage of individual flight profiles in the Pankrác Prison scenario, depending on the maximum cost of the sensor placement. It can be clearly seen that the percentage of detection of the flight profiles of attacking and smuggling is high compared to the flight profile of spying and the ordinary flight profile. This is due to the fact that we were able to prioritize certain areas of the space around the sector, from which direction the drones were more likely to fly. From the maximum price of 200 000, the coverage of the sector was so big that the detection percentages of flight profiles of attacking and smuggling reached 100 %. The detection percentage of the ordinary flight profile is the lowest because the flights for this flight profile are generated completely randomly, so it is possible that some of them were not even detectable at some point. When the maximum cost of the sensor placement was higher than 300 000, the overall detection percentage did not increase anymore.

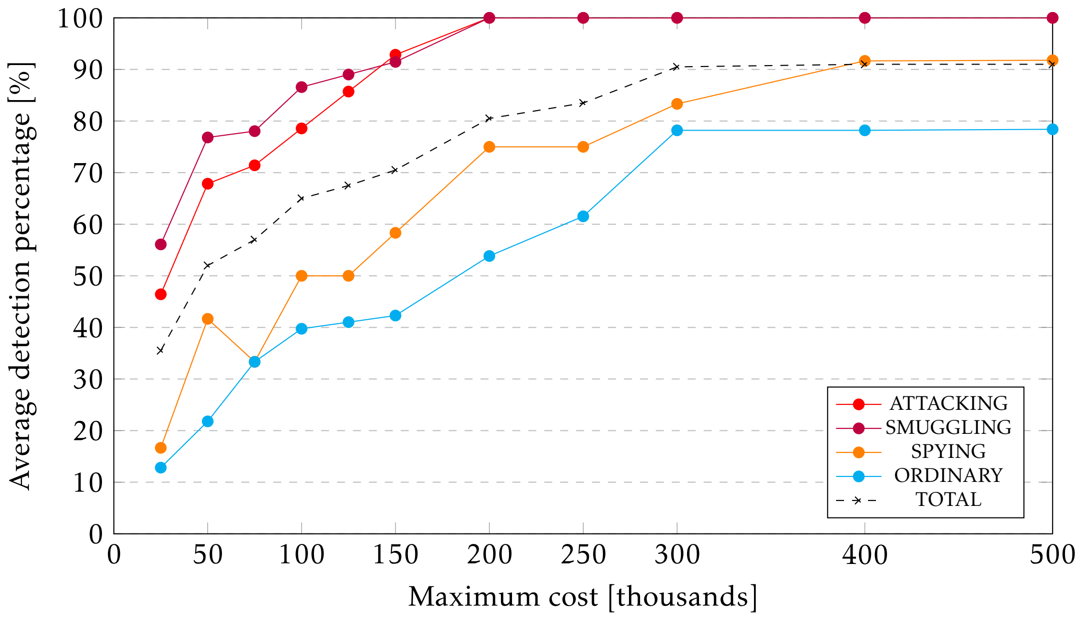

Figure 6.9: Average detection percentage of each flight profile per maximum cost of the sensor placement in the Pankrác Prison scenario

In Figure 6.10 we the graph of average detection distances for camera sensors and acoustic sensors in the Pankrác Prison scenario, depending on the maximum cost of the sensor placement. The average detection range of cameras is approximately two times larger than the average detection distance of acoustic sensors because, in the case of a large drone, the cameras can detect a drone at a much greater distance than the acoustic sensor. The average detection range of the acoustic sensors was nearly constant because the acoustic sensors were usually placed on the same spots that were the most suitable.

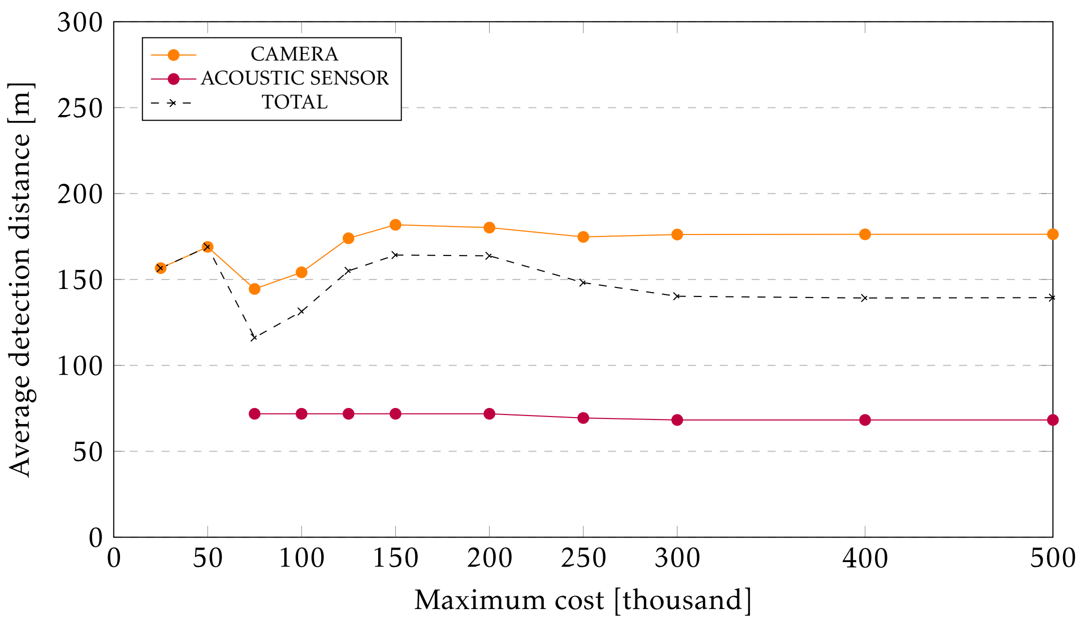

Figure 6.10: Average detection distances of cameras and acoustic sensors per maximum cost of the sensor placement in the Pankrác Prison scenario

In Figure 6.11 we show the average time to react (TTR) to the flight profiles of attacking and smuggling in the Pankrác Prison scenario, as only those two flight profiles reach the critical area eventually. In the cases of lower values of the maximum cost of the sensor placements, where there were fewer cameras and the drones were often detected only by the acoustic sensors, the average TTR is around 8 to 9 s. When the maximum cost of the sensor placement was higher and the sensor network consisted of tens of cameras, the average TTR reached nearly 12 s, which corresponds to the average detection distance shown in Figure 6.10.

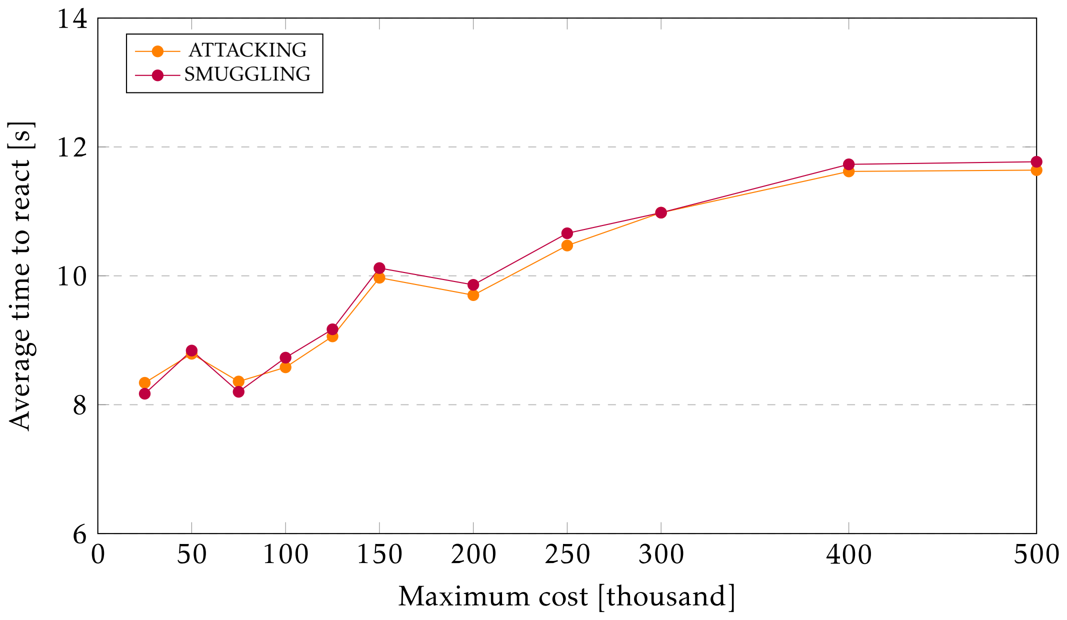

Figure 6.11: Average time to react to the flight profiles of attacking and smuggling per maximum cost of the sensor placement in the Pankrác Prison scenario

6.3.2 The Václav Havel Airport Prague scenario

Unlike the previous scenario, the monitored sector in the Václav Havel Airport Prague scenario is very large. Due to that, we, unfortunately, had to limit the accuracy of the discretized environment and set the size of the world cells to 40 m x 40 m x 40 m, for performance purposes. The horizontal and vertical monitor distance was the same as in the case of the Pankrác Prison, i.e. 250 m and 170 m, respectively. Despite the size of world cells being that big, the discretized space still counted 31803 world cells.

There were also two priority areas in this scenario, both near Terminal 3 of the Václav Havel Airport Prague. The weights of the priority and non-priority cells, coverage redundancy, and thus the weight function as well were the same as in the previous scenario. Due to the number of world cells being so large in the case of the Václav Havel Airport Prague scenario, we had to limit the number of possible values for camera panning and tilting to (0°, 20°, 40°, ..., 340°) and (0°, 10°, 20°, 30°) respectively, which results in 72 camera poses. Considering the fact there are 31803 cells, 331 sensor spots, and 72 camera poses, there had to be 757 millions of binary variables in the MILP model just to express the sensor coverage, which is a lot.

In terms of the distribution of flight profiles, we decided that there will be no flight profiles of spying as it does not give much sense to spy on an airport. Instead, we assumed a higher percentage of the flight profiles of attacking, namely 30 %. We further considered a 10 % chance of the flight profile of smuggling, to account for rare cases of smuggling items that would not pass the security checks. As the area around the airport is open space, it is an ideal place to fly with a drone, so we set the chance of the flight having the ordinary flight profile to 60 %.

The Václav Havel Airport Prague scenario is shown in Figure 6.12. There were 331 sensor spots in total, visualized as green points. In this scenario, the sensor spots were generated automatically with the method described in Subsection 4.3.4. Out of 200 generated flights, there was a total of 62 flight profiles of attacking, 24 flight profiles of smuggling, and 114 ordinary flight profiles. We show the generated flights in Figure 6.13.

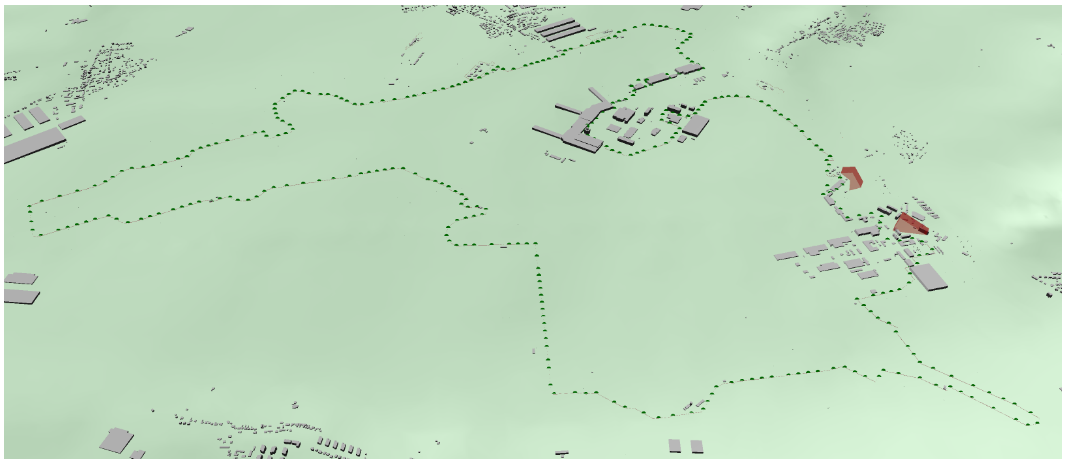

Figure 6.12: The Václav Havel Airport Prague scenario

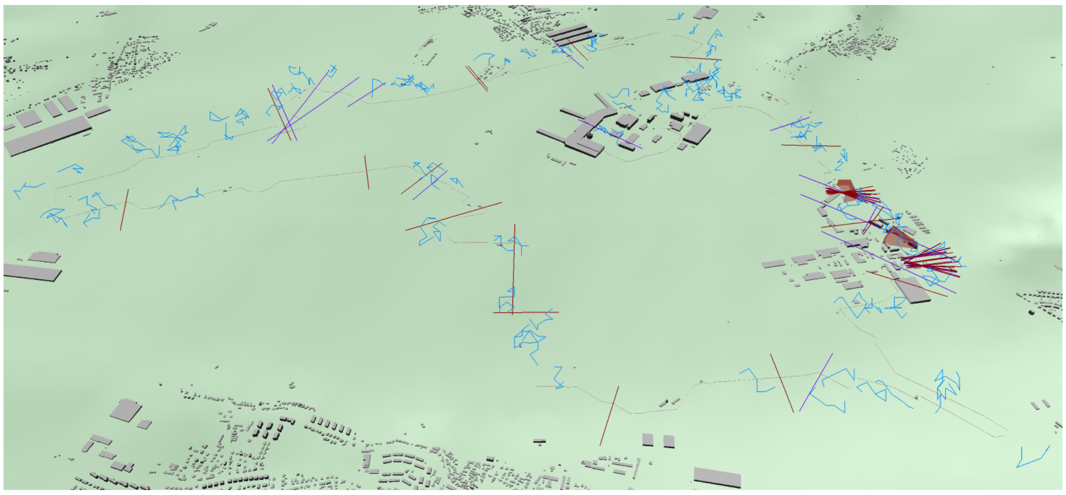
Figure 6.13: The Václav Havel Airport Prague scenario flights

In Figure 6.14 we illustrate a sensor placement in the sector of the Václav Havel Airport Prague with a maximum price of 700 000. The covered cells are represented as green points. Unlike the sector of the Pankrác Prison, acoustic sensor spots can be placed only on selected sensor spots near the areas with buildings. As the size of cell is 40 m x 40 m x 40 m and the assumed detection range of the cameras is 130 m, only 1 cell managed to fit within the frustum of the camera.

We show further examples of different sensor placements in this scenario in Appendix C.

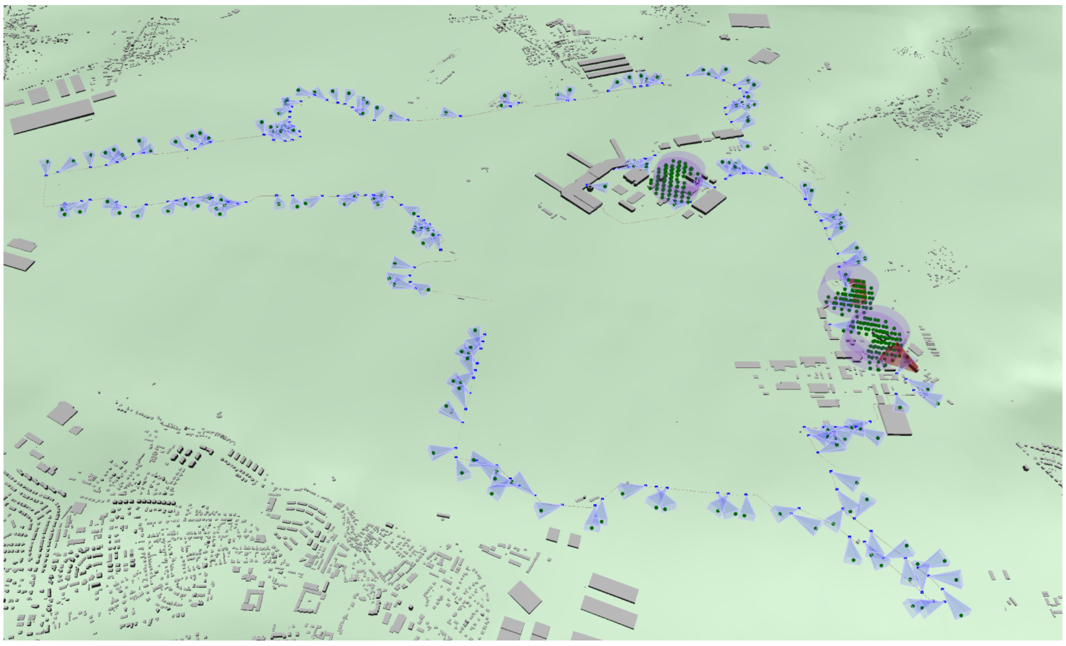
Figure 6.14: Sensor placement in the sector of the Václav Havel Airport Prague with the maximum price of 700 000

In Figure 6.15 we show the graph of average detection percentage of individual flight profiles in the Václav Havel Airport Prague, depending on the maximum cost of the sensor placement. Similarly, as in the other scenario, the highest percentage of detection was in the flight profiles as attacking and smuggling, as there was a 60 % change that the flight will lead through one of the priority areas, which we prioritized with the higher weight of cells. As in the other scenario, the detection percentage of the ordinary flight profile is the lowest because the flights for this flight profile are generated completely randomly, so it is possible that some of them were not even detectable at some point.

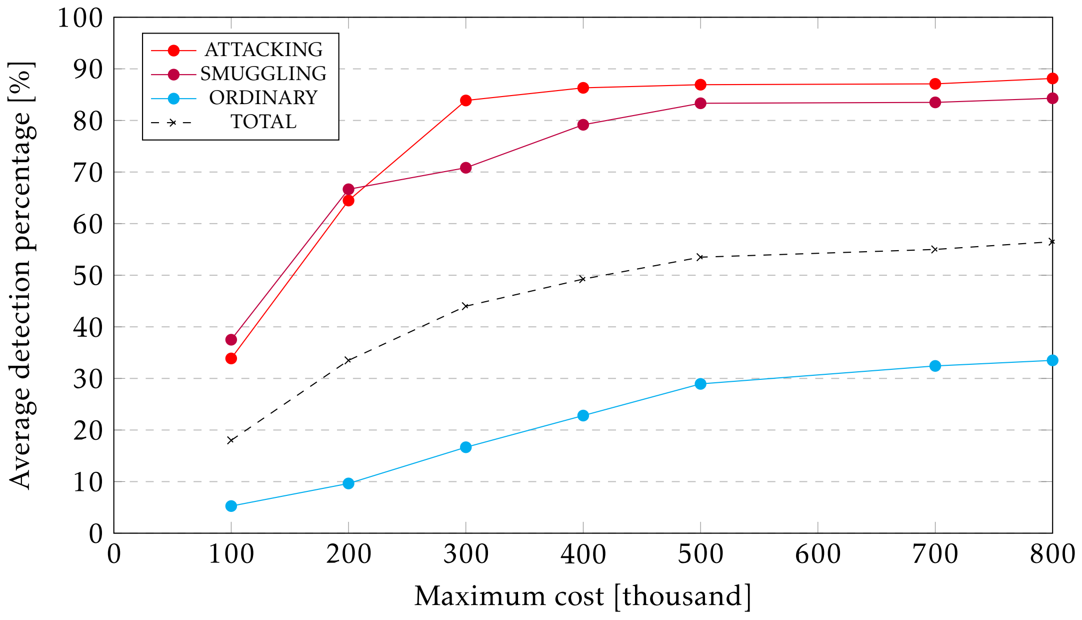

Figure 6.15: Average detection percentage of each flight profile per maximum cost of the sensor placement in the Václav Havel Airport Prague scenario

In Figure 6.16 we show the graph of average detection distances for camera sensors and acoustic sensors in the Václav Havel Airport Prague scenario, depending on the maximum cost of the sensor placement. Similarly, as in the case of the other scenario, the detection distance of cameras is more than two times greater than the detection distance of acoustic sensors, because a big part of the flights had the flight profile of attacking, which is easily detectable by the cameras as it keeps low AGL. The detection range of acoustic sensors was nearly constant for most of the sensor placements because they were often placed on the same sensor spots, as there were only a few sensor spots where it was appropriate to place an acoustic sensor.

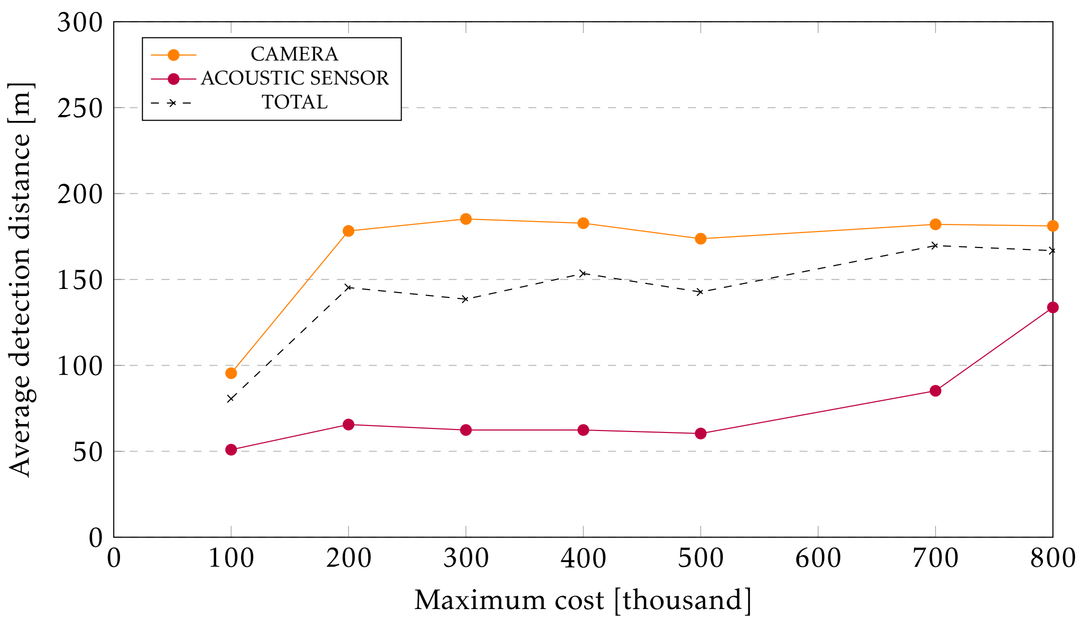

Figure 6.16: Average detection distances of cameras and acoustic sensors per maximum cost of the sensor placement in the Václav Havel Airport Prague scenario

In Figure 6.17 we show the TTR to the flight profiles of attacking and smuggling in the Václav Havel Airport Prague scenario. For the sensor placements with higher maximum prices, the TTR was stably around 10 s for the flight profiles of attacking and around 8 s for the flight profiles of smuggling. The fact that the TTRs of the flight profiles are not as similar as in the other scenario might be due to the flight profile of attacking having three times greater chance to be generated, so the sample size from which the average was computed was larger.

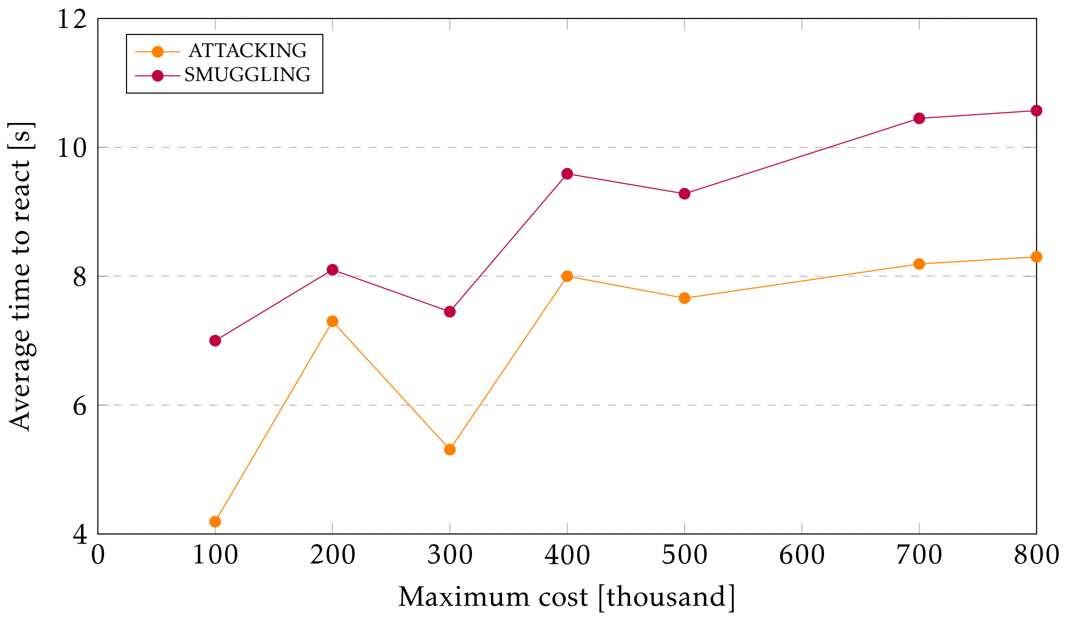

Figure 6.17: Average time to react to the flight profiles of attacking and smuggling per maximum cost of the sensor placement in the Václav Havel Airport Prague scenario

Chapter 7

Conclusion and future work

In this thesis, we proposed an approach to the problem of optimal sensor placement used for protection against drones. In the problem of optimal sensor placement, the goal is to find the optimal placement of the sensors, i.e. a sensor network, such that either the space coverage of the sensor network is maximized, or the cost of the sensor network is minimized, while a given level of coverage is achieved. We focused on making the sensor placement algorithm possible to be highly parameterized by the user, such as by the priority areas or coverage redundancy. During the implementation of the coverage computation of the camera, we also effectively dealt with the camera occlusion.

In the theoretical part of this thesis, we took a look at the state of drones in today's world, i.e. what are they used for, what options and possibilities they offer and where is their assistance useful. As drones are a powerful technology with the ability to fly fast, carry objects, and at the same time being very small, we presented various threats that they pose and illegal activities they can be misused for. To emphasize why protection against drones is needed, we included a brief list of drone incidents showing the threats are not just theory but are actually happening in the world. We also presented a short-list of state-of-the-art systems for drone detection and neutralization. Then we explored the area of possible approaches to the problem of optimal sensor placement. We started with the approximation algorithms such as the greedy algorithm or various sampling algorithms that excel at speed but usually produce only an approximation of the optimal solution in the best case. Then we presented the exact algorithms such as integer linear programming that finds the optimal solution but at great costs in terms of time, making it rather unpractical when the search space is large.

Moving on to the practical part of the thesis, we first introduced the program as a whole, describing the configuration properties and individual components that the program consists of. We presented the AgentFly simulation framework and the implementation of the camera sensor and acoustic sensor, and ray tracing provided by the framework. We explained the process of space discretization with the help of which we have turned the continuous space to a discrete environment that greatly reduces the complexity of modeling the sensor coverage and in which it is generally easier to operate from the point of view of algorithms.

After we familiarized the reader with our program, we advanced to the description of the implementation details. First, we focused on explaining how we effectively dealt with the camera occlusion and the sensor coverage in general. Then we introduced our approach to the problem of optimal sensor placement using the mixed-integer linear programming with the ability to dynamically update weights of the world cells based on their current coverage level.

At the end of this thesis, we presented the experimental evaluation, with the help of which we evaluate the proposed sensor placement algorithm. We introduced a similar scenario for two different sectors, a small sector of the Pankrác Prison, and a large sector of the Václav Havel Airport Prague. In both scenarios, we launched two hundred drones from the area around the sector with the flight profiles of attacking, smuggling, spying, and the ordinary flight profile that does not pose any danger. We measured the detection percentages and distances for the individual flight profiles and the average time to react to the approaching drone.

We have shown that a detection system consisting only of acoustic sensors and cameras would probably not be enough effective in the reality, as less than 20 s of reaction time is not enough to take appropriate defensive countermeasures, assuming a large drone where the detection distance is approximately 350 m.

7.1 Future work

In reality, a drone detection system consisting of only acoustic sensors and cameras would probably not be effective enough. As the data have shown, when the detection distance is about 140 m and the speed of the drone is approximately 60 km/h, the average time to react is less than 9 s, which is not much at all, considering the fact that appropriate countermeasures have to be introduced during that time as well. However, the way how we model the acoustic sensor probably does not fully reflect the reality, as we are left out with parameters for which we either do not know the correct values, or we have to limit them to some extent for optimization purposes, such as the number of initial and reflection rays or the sensitivity of the sensor. Because of that, the measured detection distances would most likely increase, because a quality acoustic sensor usually has, or should have, an effective detection range of more than 140 m.

Furthermore, the implementation of the camera sensor in *Visio* does not support zooming, which is a standard in today's cameras and can provide much greater detection distance once the direction of the drone is estimated. When the loaded drone model, such as the model of DJI Matrice Pro 600 shown in Figure 4.7, was very good quality and consisted of lots of individual triangles, the ray tracing operations were very computationally demanding, due to which could not run a smooth simulation of the drone movements while detecting multiple drones at once.

Moreover, to be applicable in real life effectively, the system would have to be extended with different types of sensors, such as the radio frequency (RF) analyzers or radars. RF analyzers and radars are included in most of the state-of-the-art drone detection systems and can provide a detection range in the order of kilometers. Also, they can provide additional data such as estimations of the drone model, or rough localization of the controller using which is the drone being controlled in some cases. However, the implementation of radars and RF analyzer is not an easy task at all, as various physics would need to be considered. In the current state of the detection systems, we assume fictitious data about the cameras and acoustic sensors as well, such as the price in the case of cameras, or sensitivity in the case of acoustic sensors.

Last, the drone database is not sufficient at the moment, as we only use three different drone models in the experimental evaluation, with fixed properties such as the speed that remains unchanged during the whole flight. The environment used throughout the program also does not reflect real-life adequately, as it only assumes fixed, unchanging air properties such as air temperature, air pressure, and air humidity. It also does not recognize day or night time, nor any kind of weather conditions, which is often crucial in the real-life.

Appendix A

Measured data

In Table A.1 and Table A.2 we provide a more detailed view of the data measured in the Pankrác Prison scenario and the Václav Havel Airport Prague scenario, respectively.

	25k	50k	75k	100k	125k	150k	200k	250k	300k	400k	500k
Num. of cameras	10	20	8	18	28	38	57	56	54	90	92
Num. of acoustic sensors	0	0	1	1	1	1	1	2	3	3	3
Avg. DP - all [%]	35.5	52	57	65	67.5	70.5	80.5	83.5	90.5	91	91
Avg. DP - attacking [%]	46.428	67.85	71.42	78.57	85.71	92.85	100	100	100	100	100
Avg. DP - smuggling [%]	56.097	76.82	78.04	86.58	89.02	91.46	100	100	100	100	100
Avg. DP - spying [%]	16.666	41.66	33.33	50	50	58.33	75	75	83.33	91.66	91.78
Avg. DP - ordinary [%]	12.82	21.79	33.33	39.74	41.02	42.3	53.84	61.53	78.2	78.2	78.4
Avg. DD - all [m]	156.61	168.95	116.19	131.38	155.01	164.21	163.77	148.14	140.23	139.22	139.44
Min. DD - camera [m]	13.22	7.75	30.41	30.41	58.86	58.83	58.86	14.96	14.96	14.96	14.96
Avg. DD - camera [m]	156.61	168.95	144.48	154.17	173.99	181.82	180.19	174.77	176.16	176.27	176.31
Max. DD - camera [m]	298.96	311.46	289.74	309.35	428.14	436.98	436.98	436.98	436.98	428.14	430.55
Min. DD - acoustic [m]			1.46	1.46	1.46	1.46	1.46	2.13	1.86	1.86	1.86
Avg. DD - acoustic [m]			71.84	71.84	71.84	71.84	71.84	69.39	68.24	68.24	68.24
Max. DD - acoustic [m]			129.47	129.47	129.47	129.47	129.47	139.35	139.35	139.35	139.35
TTR - attacking [s]	8.34	8.79	8.36	8.58	9.06	9.97	9.7	10.47	10.77	11.62	11.64
TTR - smuggling [s]	8.17	8.84	8.2	8.73	9.71	10.12	9.86	10.66	10.98	11.73	11.77

Table A.1: Detection data measured in the Pankrác Prison scenario

	100k	200k	300k	400k	500k	700k	800k
Num. of cameras	18	38	54	94	112	170	188
Num. of acoustic sensors	1	2	3	3	4	5	6
Avg. DP - all [%]	18	33.48	43.99	49.25	53.44	55	56.5
Avg. DP - attacking [%]	33.87	64.48	83.87	86.32	86.93	87.09	88.14
Avg. DP - smuggling [%]	37.5	66.66	70.83	79.16	83.33	83.5	84.3
Avg. DP - ordinary [%]	5.26	9.64	16.66	22.8	28.94	32.42	33.5
Avg. DD - all [m]	80.61	145.28	138.54	153.47	142.7	169.71	166.78
Min. DD - camera [m]	64.52	40.48	80.7	40.48	40.48	40.48	40.91
Avg. DD - camera [m]	95.44	178.24	185.21	182.74	173.72	182.07	181.17
Max. DD - camera [m]	129.66	371	381.14	461.75	432.16	423.62	426.15
Min. DD - acoustic [m]	19.55	0.7	0.7	0.7	0.7	3.64	1,81
Avg. DD - acoustic [m]	50.94	65.58	62.4	62.4	60.36	85.22	81,03
Max. DD - acoustic [m]	110.41	132.38	132.38	132.38	132.38	132.38	133,75
TTR - attacking [s]	4.19	7.3	5.31	8	7.66	8.19	8.3
TTR - smuggling [s]	7	8.1	7.45	9.59	9.28	10.45	10.57

Table A.2: Detection data measured in the Václav Havel Airport Prague scenario

Appendix B

Drone detections

In the following figures, we present more examples of detections of drones of various sizes and at different distances.

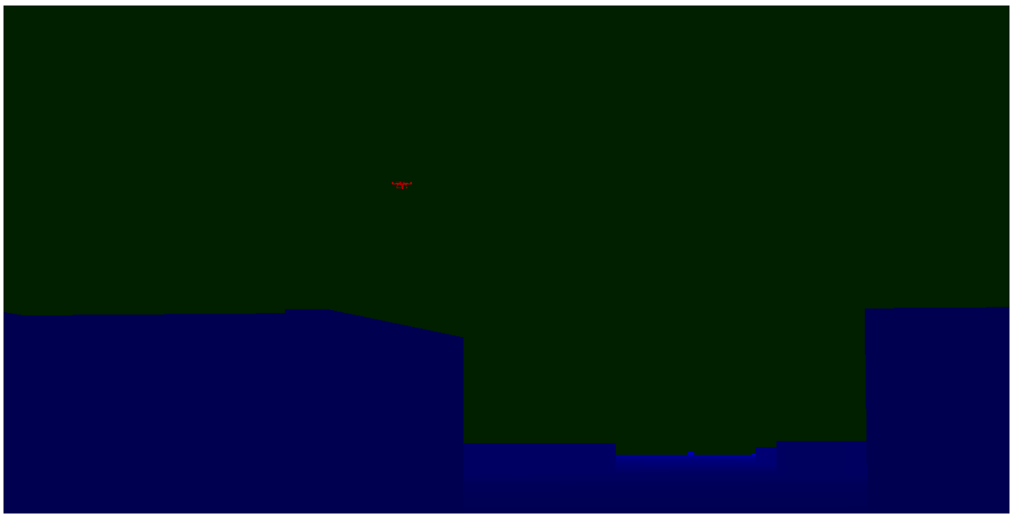

Figure B.1: Detection of a small-sized drone 50 m away from the camera with full HD resolution and 30° FOV. The number of detected pixels is 98.

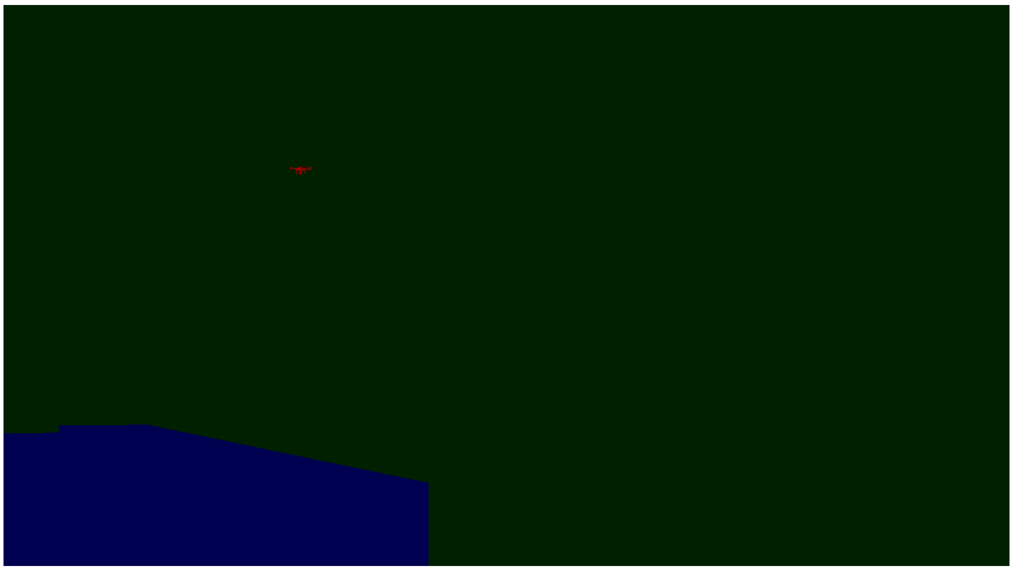

Figure B.2: Detection of a small-sized drone 100 m away from the camera with 4K resolution and 30° FOV. The number of detected pixels is 105.

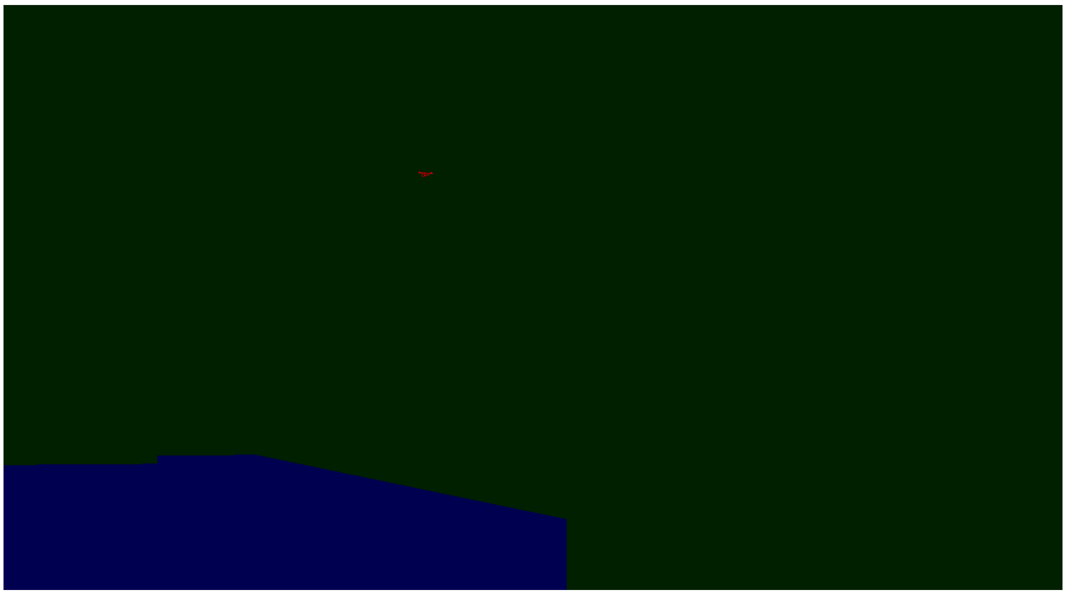

Figure B.3: Detection of a small-sized drone 150 m away from the camera with 4K HD resolution and 30° FOV. The number of detected pixels is 42.

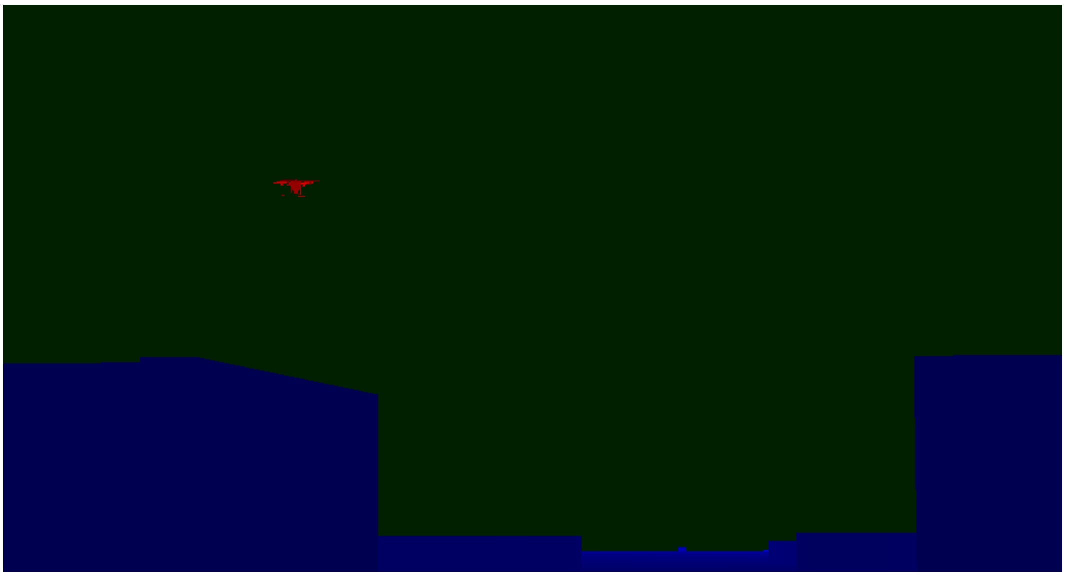

Figure B.4: Detection of a medium-sized drone 50 m away from the camera with full HD resolution and 30° FOV. The number of detected pixels is 246.

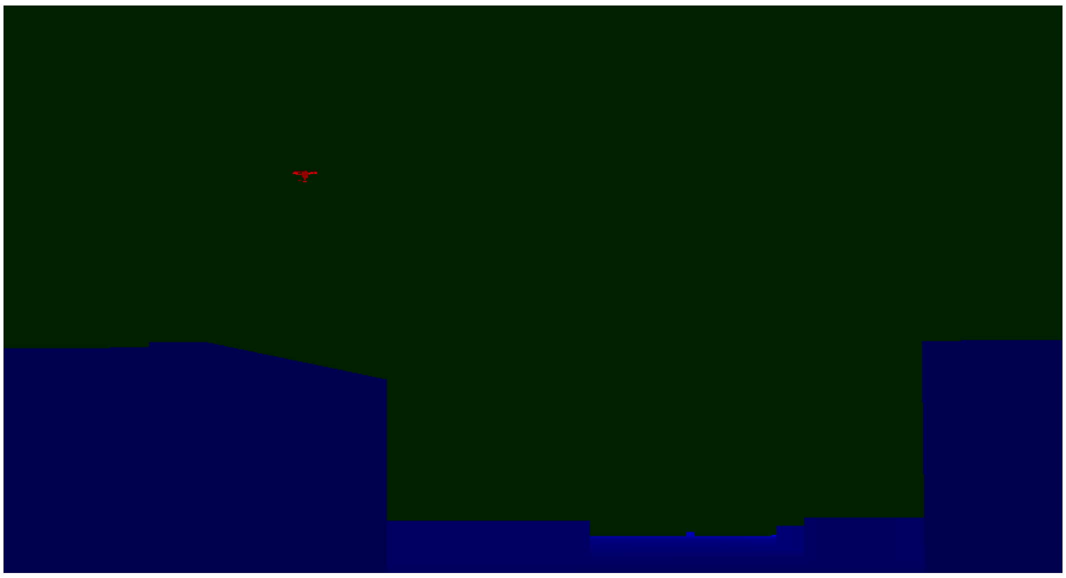

Figure B.5: Detection of a medium-sized drone 80 m away from the camera with full HD resolution and 30° FOV. The number of detected pixels is 98.

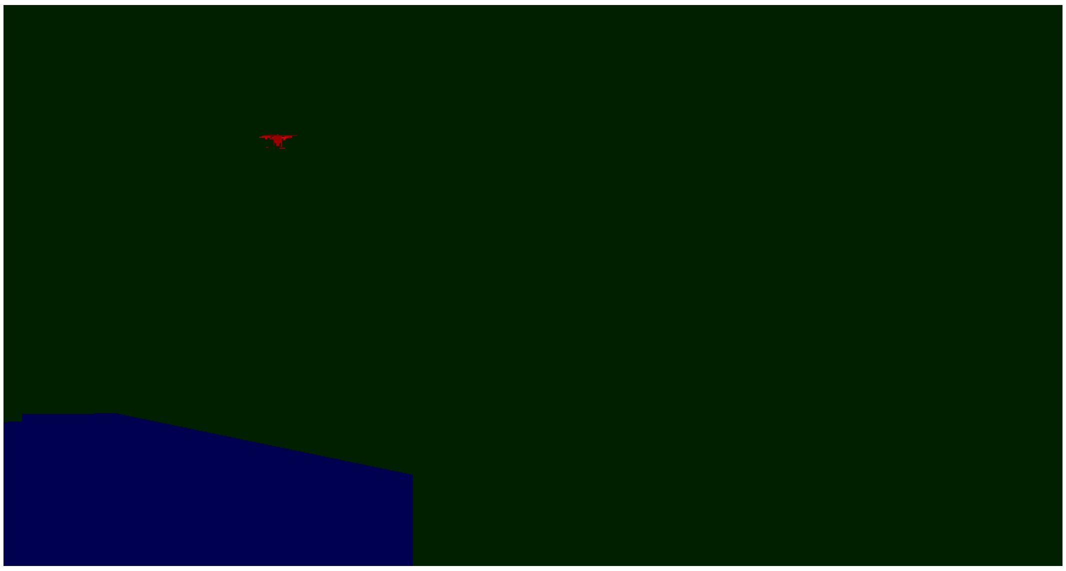

Figure B.6: Detection of a medium-sized drone 100 m away from the camera with 4K resolution and 30° FOV. The number of detected pixels is 248.

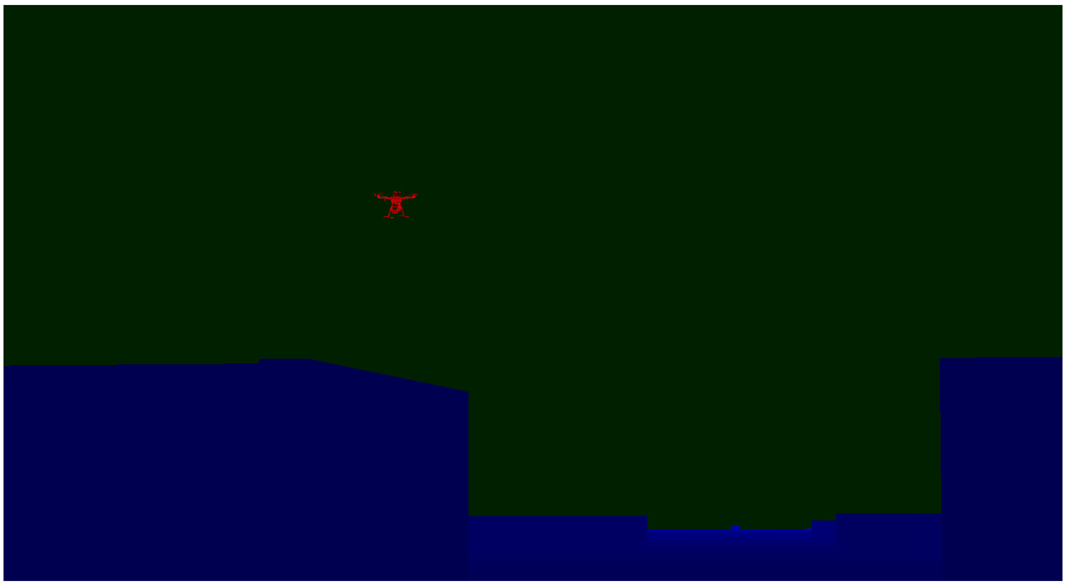

Figure B.7: Detection of a large drone 120 m away from the camera with full HD resolution and 30° FOV. The number of detected pixels is 323.

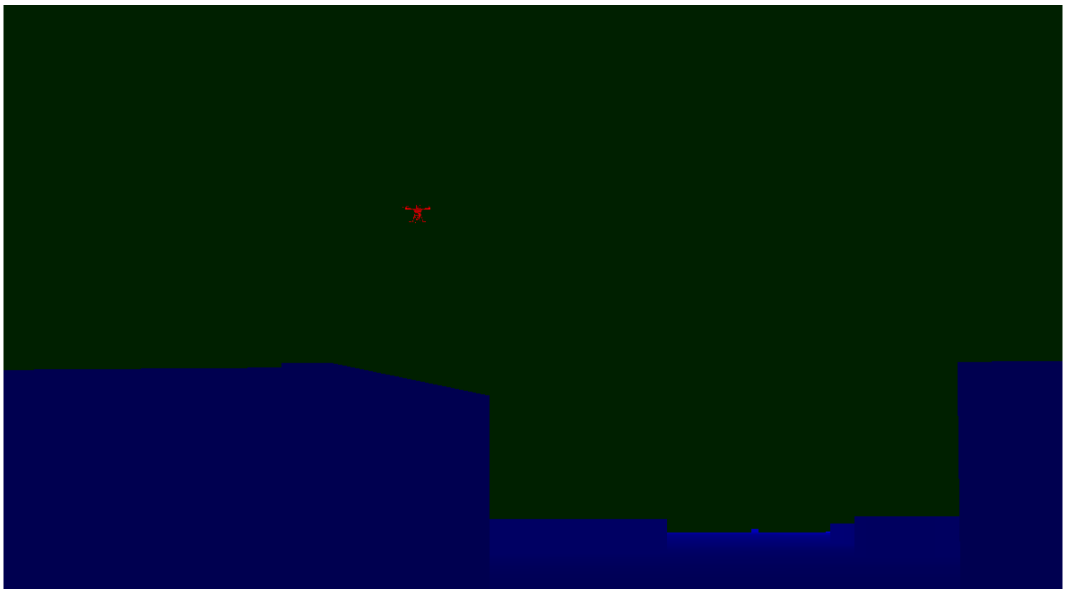

Figure B.8: Detection of a large drone 180 m away from the camera with full HD resolution and 30° FOV. The number of detected pixels is 141.

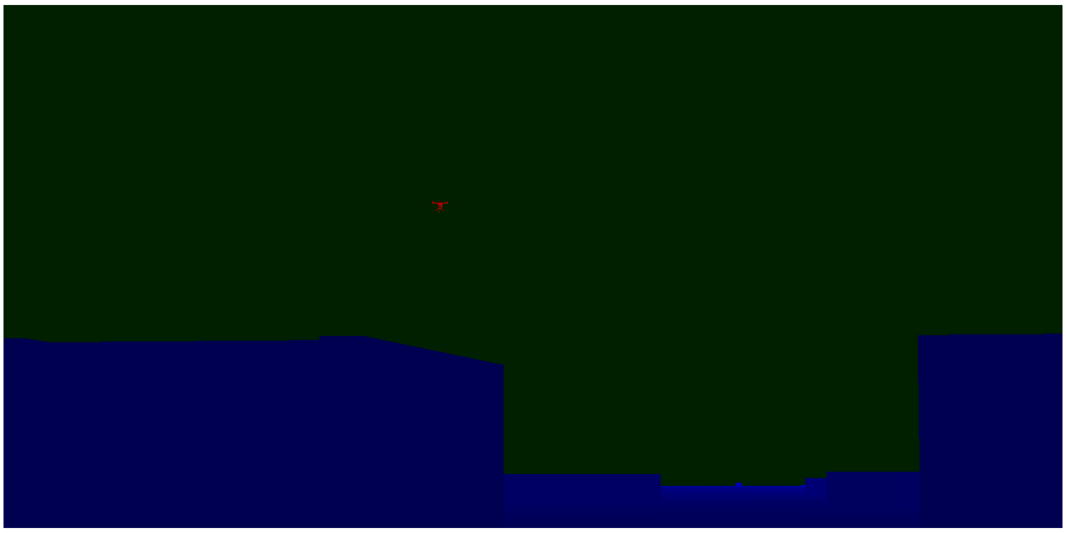

Figure B.9: Detection of a large drone 250 m away from the camera with full HD resolution and 30° FOV. The number of detected pixels is 71.

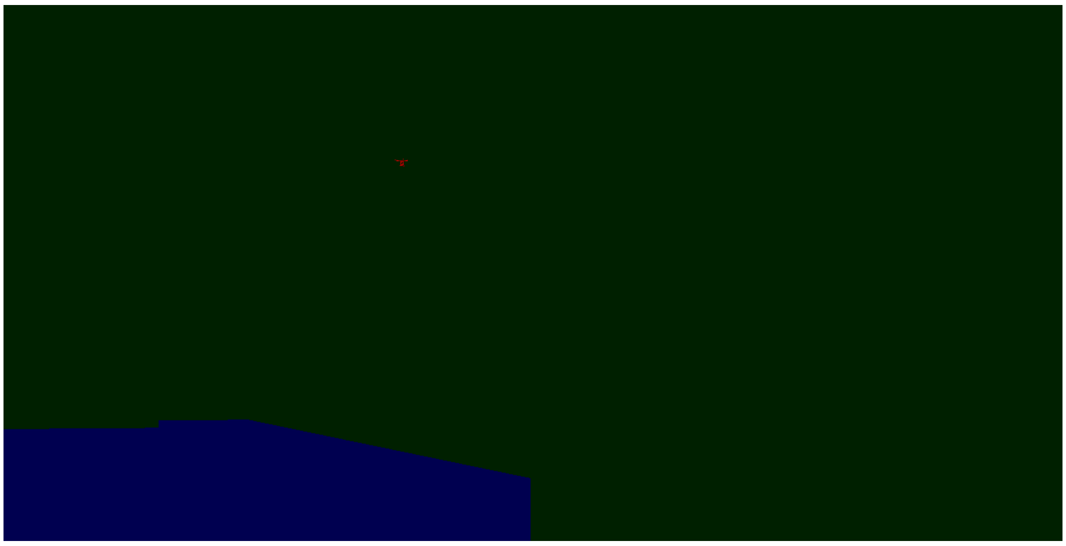

Figure B.10: Detection of a large drone 700 m away from the camera with 4K resolution and 30° FOV. The number of detected pixels is 41.

Appendix C

Sensor placements

In the following figures, we show more examples of sensor placements and how they are influenced by different weights of priority cells or different values of the coverage redundancy.

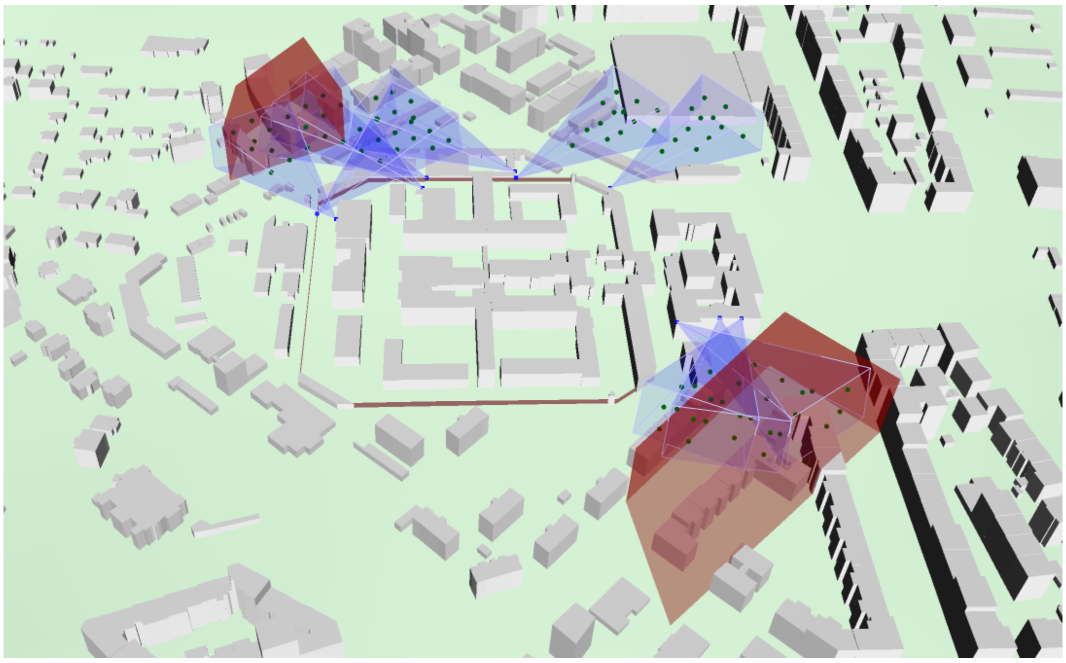

Figure C.1: Sensor placement in the sector of the Pankrác Prison with the maximum price of 25 000. The weight of the priority cells is 2. The value of redundancy is 2.

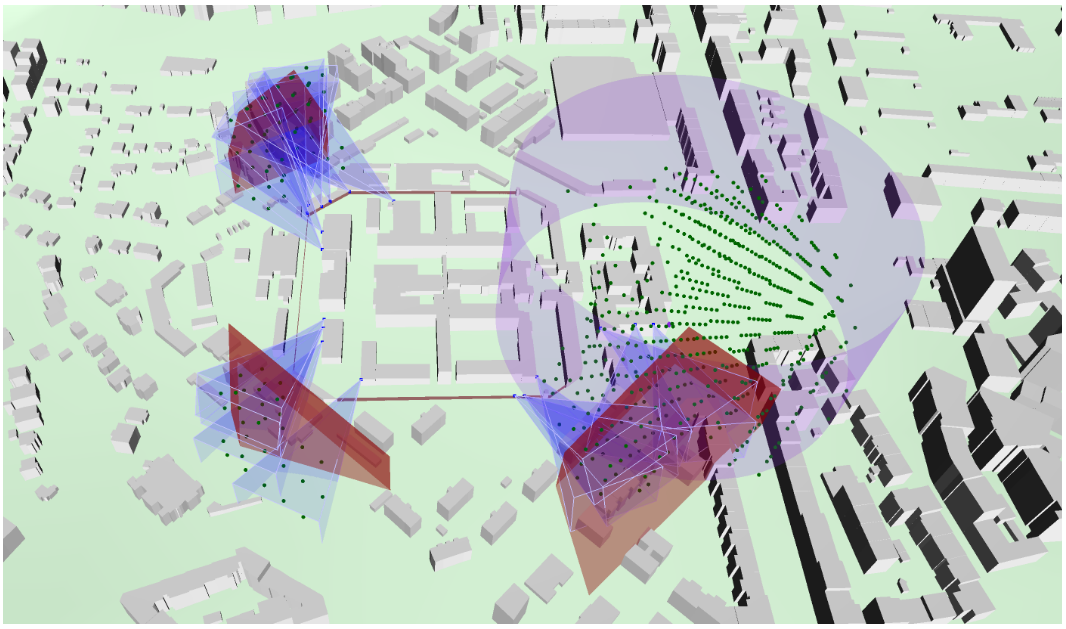

Figure C.2: Sensor placement in the sector of the Pankrác Prison with the maximum price of 100 000. The weight of the priority cells is 4. The value of redundancy is 3.

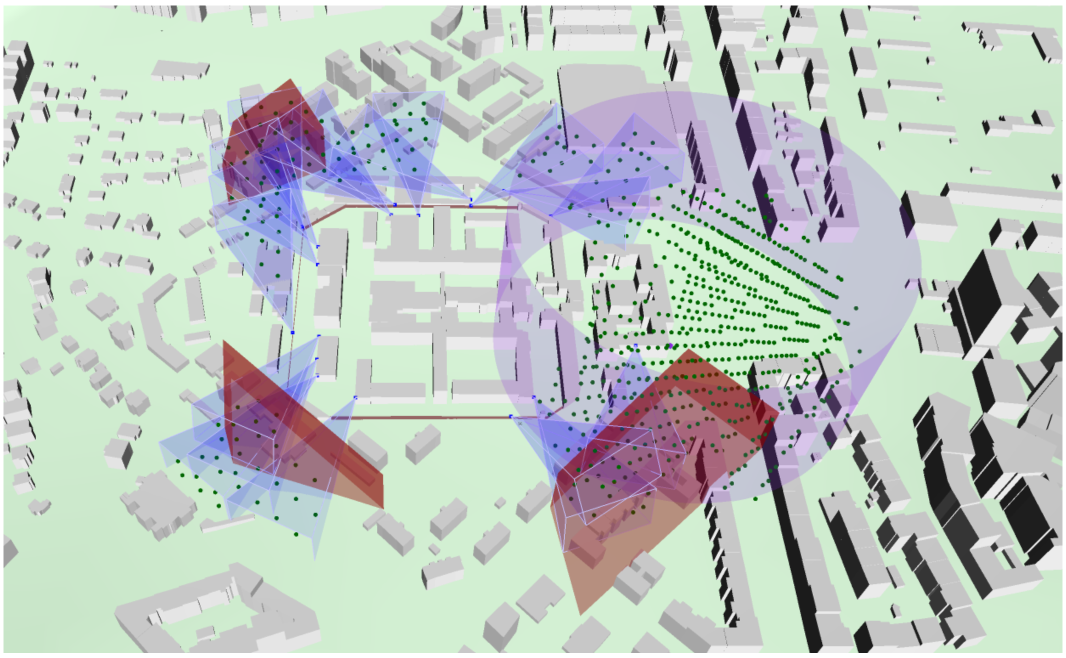

Figure C.3: Sensor placement in the sector of the Pankrác Prison with the maximum price of 100 000. The weight of the priority cells is 2. The value of redundancy is 3.

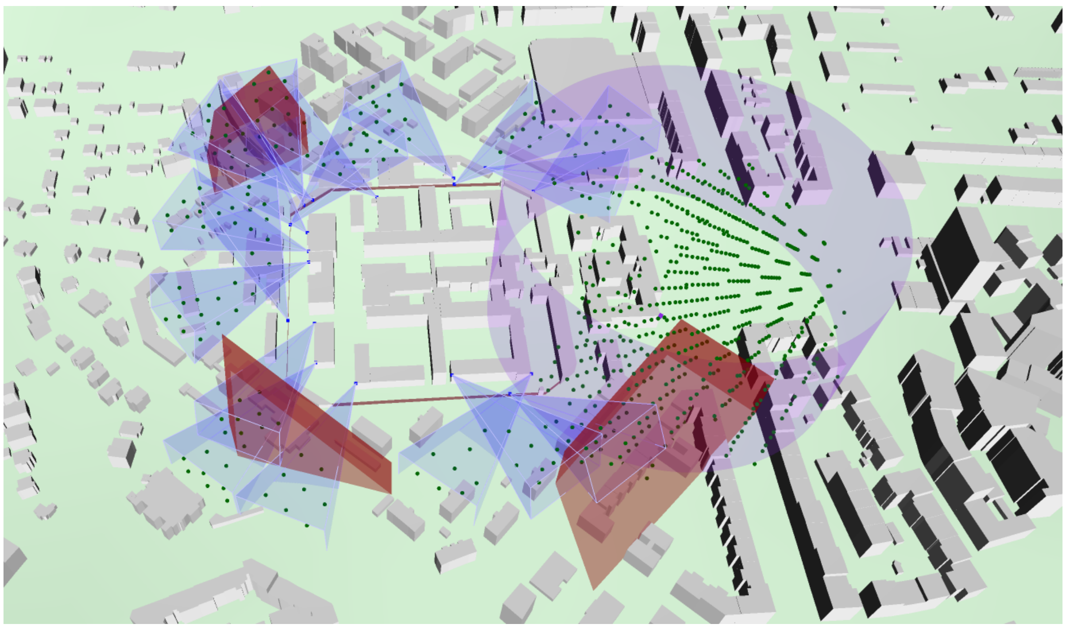

Figure C.4: Sensor placement in the sector of the Pankrác Prison with the maximum price of 100 000. The weight of the priority cells is 2. The value of redundancy is 1.

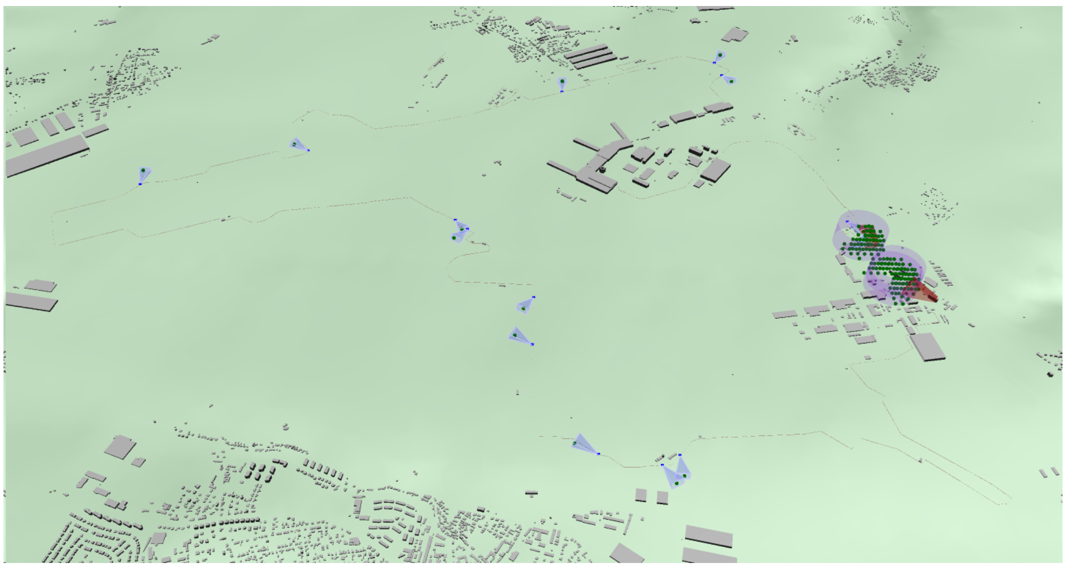

Figure C.5: Sensor placement in the sector of the Václav Havel Airport Prague with the maximum price of 200 000. The weight of the priority cells is 2. The value of redundancy is 2. As the most part of the sector remains uncovered, the maximum price of 200 000 is definitely not sufficient.

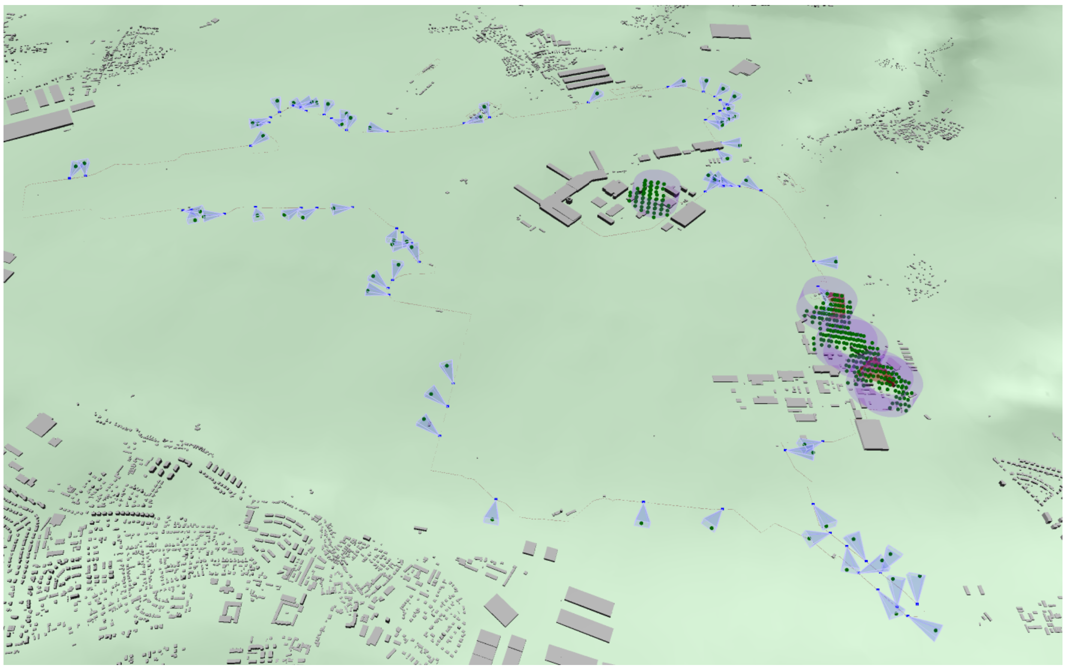

Figure C.6: Sensor placement in the sector of the Václav Havel Airport Prague with the maximum price of 500 000. The weight of the priority cells is 4. The value of redundancy is 2.

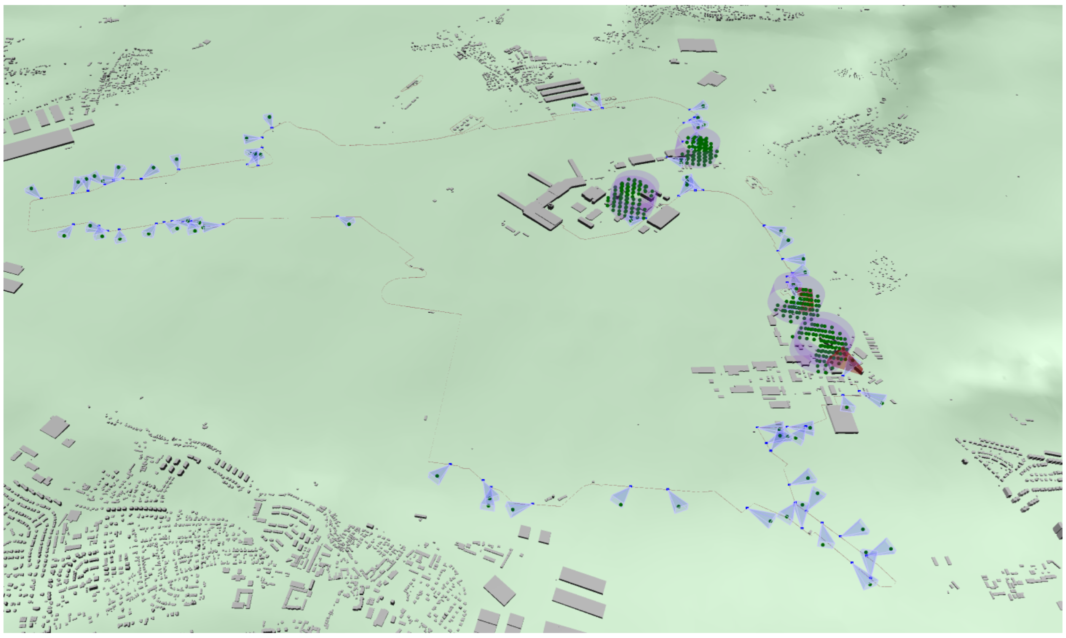

Figure C.7: Sensor placement in the sector of the Václav Havel Airport Prague with the maximum price of 500 000. The weight of the priority cells is 2. The value of redundancy is 2. In this case, the difference between the two sensor placements with the same maximum price but the different weight of priority cells is not as significant as in the case of the much smaller sector of Pankrác Prison.

Appendix D

Attached files

In Table D.1 we show the list of attached files and their structure. The resource files for drones and environments were left out due to the size limitations of the attached files.

File	**Description**
/src	source files
Master_Thesis_Dominik_Hoftych.pdf	PDF of this thesis

Table D.1: The list of attached files.

Bibliography

[1] Mark LaFay. *Popular Uses for Drones*. URL: https://www.dummies.com/consumer-electronics/drones/popular-uses-for-drones/?fbclid=IwAR1Xf7WDCC74YDSQnEDPBviEYizTuahNDzIqoMgKJAdK4M2K68ilzn4pVGs (page 3).

[2] Wikipedia contributors. *Remote sensing — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Remote_sensing&oldid=972005266. [Online; accessed 10-August-2020]. 2020 (page 3).

[3] *Summary of Safety rules in EU*. URL: https://dronerules.eu/en/recreational/obligations/summary-of-safety-rules-in-eu-1 (page 4).

[4] *Drone Laws in the U.S.A.: UAV Coach (2020)*. URL: https://uavcoach.com/drone-laws-in-united-states-of-america/ (page 4).

[5] *UAS Sightings Report*. Apr. 2020. URL: https://www.faa.gov/uas/resources/public_records/uas_sightings_report/ (page 4).

[6] Harro Ranter. *Aviation Safety Network; ASN Aviation Safety Database; ASN Drone Database*. URL: https://aviation-safety.net/database/issue/dronedb.php (page 4).

[7] Armygirl. *Drone Crash Database*. Apr. 2020. URL: https://dronewars.net/drone-crash-database/ (page 4).

[8] *Map of World Wide Drone Incidents*. URL: https://www.dedrone.com/resources/incidents/all (page 4).

[9] Jack Beresford. *Aer Lingus flight from Dublin to London came scarily close to colliding with illegal drone*. Sept. 2019. URL: https://www.irishpost.com/news/aer-lingus-drone-near-miss-170793?utm_source=twitter&utm_campaign=article&utm_medium=web (page 4).

[10] Darren Boyle. *BA jet in near miss with a drone 6,000 feet above Potters Bar*. May 2019. URL: https://www.dailymail.co.uk/news/article-7042177/BA-jet-near-miss-drone-6-000-feet-Potters-Bar.html (page 5).

[11] Jen Mills. *Drone 'put 130 lives at risk' as plane was trying to land at Gatwick*. Dec. 2019. URL: https://metro.co.uk/2017/10/16/drone-put-130-lives-at-risk-as-plane-was-trying-to-land-at-gatwick-7003871/ (page 5).

[12] *Gang who flew drones carrying drugs into prisons jailed*. Oct. 2018. URL: https://www.bbc.com/news/uk-england-45980560 (page 5).

[13] *Perth Prison drug smugglers filmed themselves on drone camera*. Oct. 2018. URL: https://www.bbc.com/news/uk-scotland-tayside-central-45731642 (page 5).

[14] *Drone flies over Macron's holiday home in wake of Maduro 'attack'*. URL: https://www.thelocal.fr/20180807/drone-flies-over-macrons-holiday-home-in-wake-of-maduro-attack (page 5).

[15] Diamond.leung@sporttechie.com. *Drone Delays Soccer Match, Prompting Safety Concerns Before Key Goal*. Jan. 2018. URL: https://www.sporttechie.com/drone-delays-soccer-match-safety-concerns-goal/ (page 5).

[16] *Part I – Drone Detection Technologies*. URL: https://www.cerbair.com/drone-detection-and-neutralization-technologies-parti-blog/ (pages 6, 7).

[17] *9 Counter-Drone Technologies To Detect And Stop Drones Today*. URL: https://www.robinradar.com/press/blog/9-counter-drone-technologies-to-detect-and-stop-drones-today (pages 8, 10).

[18] Pablo Estrada. *Protecting Against Drones: A Brief Review of Passive Countermeasures*. 2018. URL: https://www.dedrone.com/blog/protecting-against-drones-a-brief-review-of-passive-countermeasures (page 9).

[19] *Defense against drones – the danger on the radar screen*. Dec. 2019. URL: http://fraunhofer.de/en/research/current-research/defense-against-drones.html?fbclid=IwAR0jh4KQpu59dGHdkP_kxKAwpkbSVI7zdJbJJbISm6uyAHIev--Z-ZbguvM (page 10).

[20] Serge Malenkovich et al. *Is it possible to guard against GPS attacks?* URL: https://www.kaspersky.com/blog/gps-spoofing-protection/26837/ (page 10).

[21] Jörg Lamprecht and Aman Johal. *The Pros and Cons of Active and Passive Drone Countermeasures*. Mar. 2016. URL: https://www.informationsecuritybuzz.com/articles/pros-cons-active-passive-drone-countermeasures/ (page 11).

[22] *Counter Drone Systems - SKYLOCK - Anti Drone Solutions*. Dec. 2019. URL: https://www.skylock1.com/counter-drone-systems/?utm_term=+counter%20+drone&utm_campaign=eu&gclid=CjwKCAjw4871BRAjEiwAbxXi22JzrkXLf3NmIvjqi3b603QllwowfrrzGWBnfjZgHYGwrgPaShifDxoCb5AQAvD_BwE (pages 11, 12).

[23] *Anti Drone Jammers - SKYLOCK Anti Drone Solutions*. Dec. 2019. URL: https://www.skylock1.com/anti-drone-jammers/?gclid=CjwKCAjw4871BRAjEiwAbxXi2wxkKOY9iR3HoDotd0fQSBfVEaCMP07sVUS43-oB05q0XexTKRhpsxoC6G8QAvD_BwE&utm_term=drone%20guard&utm_campaign=comp (page 12).

[24] *How DroneShield Works*. URL: https://www.droneshield.com/how-droneshield-works1 (page 13).

[25] *how Ctrl Sky works? - aps - advanced protection systems*. URL: https://apsystems.tech/en/how-ctrlsky-works-2/ (page 13).

[26] Jun-Woo Ahn et al. "Two-Phase Algorithm for Optimal Camera Placement". In: *Scientific Programming* 2016 (Jan. 2016), pp. 1–16. DOI: 10.1155/2016/4801784 (pages 15, 19).

[27] Wikipedia contributors. *Linear programming — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Linear_programming&oldid=969478172. [Online; accessed 6-August-2020]. 2020 (pages 16, 18).

[28] Jian Zhao et al. "Approximate Techniques in Solving Optimal Camera Placement Problems". In: *International Journal of Distributed Sensor Networks* 9.11 (2013), p. 241913. DOI: 10.1155/2013/241913. eprint: https://doi.org/10.1155/2013/241913. URL: https://doi.org/10.1155/2013/241913 (pages 16–18).

[29] Wikipedia contributors. *Set cover problem — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Set_cover_problem&oldid=944382146. [Online; accessed 6-August-2020]. 2020 (page 16).

[30] Petr Slavík. "A Tight Analysis of the Greedy Algorithm for Set Cover". In: *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*. STOC '96. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, pp. 435–441. ISBN: 0897917855. DOI: 10.1145/237814.237991. URL: https://doi.org/10.1145/237814.237991 (page 16).

[31] E. Hörster and R. Lienhart. "On the Optimal Placement of Multiple Visual Sensors". In: *Proceedings of the 4th ACM International Workshop on Video Surveillance and Sensor Networks*. VSSN '06. Santa Barbara, California, USA: Association for Computing Machinery, 2006, pp. 111–120. ISBN: 1595934960. DOI: 10.1145/1178782.1178800. URL: https://doi.org/10.1145/1178782.1178800 (pages 17, 19).

[32] Yves Tillé. "Sampling Algorithms". In: *International Encyclopedia of Statistical Science*. Ed. by Miodrag Lovric. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 1273–1274. ISBN: 978-3-642-04898-2. DOI: 10.1007/978-3-642-04898-2_501. URL: https://doi.org/10.1007/978-3-642-04898-2_501 (page 17).

[33] Jun S Liu. *Monte Carlo strategies in scientific computing*. Springer Science & Business Media, 2008 (page 17).

[34] Wikipedia contributors. *Simulated annealing — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Simulated_annealing&oldid=966943761. [Online; accessed 8-August-2020]. 2020 (page 18).

[35] Junbin Liu et al. "On the Statistical Determination of Optimal Camera Configurations in Large Scale Surveillance Networks". In: Oct. 2012. DOI: 10.1007/978-3-642-33718-5_4 (page 18).

[36] Daryn Ramsden. "OPTIMIZATION APPROACHES TO SENSOR PLACEMENT PROBLEMS". In: 2009 (page 18).

[37] Leonid Genrikhovich Khachiyan. "A polynomial algorithm in linear programming". In: *Doklady Akademii Nauk*. Vol. 244. 5. Russian Academy of Sciences. 1979, pp. 1093–1096 (page 18).

[38] Wikipedia contributors. *Integer programming — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Integer_programming&oldid=950261457. [Online; accessed 6-August-2020]. 2020 (page 18).

[39] Hans-Paul Schwefel. *Evolution and Optimum Seeking*. Jan. 1995. ISBN: 978-0-471-57148-3 (page 19).

[40] Aug. 2020. URL: https://www.gurobi.com/products/gurobi-optimizer/ (page 19).

[41] Vojtěch Kaiser. "Efficient Rendering of Earth Surface for Air Traffic Visualization". MA thesis. Czech Technical University in Prague, 2018 (page 22).

[42] Jan Kúdelka. "Critical Infrastructure Protection Against Attacks Using Drones". Bachelor's Thesis. Czech Technical University in Prague, 2018 (page 22).

[43] M. Hapala and V. Havran. “Review: Kd-tree Traversal Algorithms for Ray Tracing”. In: *Computer Graphics Forum* 30.1 (2011), pp. 199–213. DOI: 10.1111/j.1467-8659.2010.01844.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2010.01844.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2010.01844.x (page 22).

[44] Ingo Wald and Vlastimil Havran. “On Building Fast kd-trees for Ray Tracing, and on Doing that in O(N log N)”. In: *Symposium on Interactive Ray Tracing* 0 (Sept. 2006), pp. 61–69. DOI: 10.1109/RT.2006.280216 (page 22).

[45] Wikipedia contributors. *K-d tree — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=K-d_tree&oldid=961252065. [Online; accessed 13-June-2020]. 2020 (page 22).

[46] *Model View Projection*. [Online; accessed 21-June-2020]. URL: https://jsantell.com/model-view-projection/ (page 43).

[47] *Coordinate Systems*. [Online; accessed 21-June-2020]. URL: https://learnopengl.com/Getting-started/Coordinate-Systems (page 43).

[48] Bastian Molkenthin. *Software Rasterization Algorithms for Filling Triangles*. May 2012. URL: http://www.sunshine2k.de/coding/java/TriangleRasterization/TriangleRasterization.html (page 44).

[49] Haye Kesteloo. *Video: More detail on the noise levels of the DJI Mavic Pro Platinum vs DJI Mavic Pro*. Sept. 2017. URL: https://dronedj.com/2017/09/28/video-detail-on-the-noise-levels-of-the-dji-mavic-pro-platinum-vs-dji-mavic-pro/ (page 46).