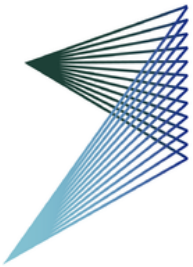

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Drone Obstacle Avoidance: the Software Engineering Strategy applied

Jore Booy
jorejoshua.booy@studio.unibo.it

Eetu Kahelin
eetu.kahelin@studio.unibo.it

Christoph Konheisner
christoph.konheisner@studio.unibo.it

Lorenzo Morelli
lorenzo.morelli9@studio.unibo.it

June 1, 2022

Contents

1 Introduction: the problem to be solved

Software engineering is hard. Engineering something intelligent is even harder. That is when the process of software development becomes even more important. In this report, we will share how we applied different principles from the Software Engineering for Intelligent Distributed Systems course at the University of Bologna to create the software solution for our project.

We took part in the ICRA 2022 DodgeDrone Challenge. The DodgeDrone Challenge is a competition where anyone can participate and hand-in submissions online. Then, the finalists will be announced and the grand finals will happen live at the ICRA 2022 conference in Philadelphia, USA.

The goal is to create a vision or state-based algorithm for a drone to reach a destination as fast as possible while avoiding obstacles. More specifically, we needed to write a python program that given the current state and vision of the drone (and possibly knowledge of the positions of the obstacles), returns a command that would lead the drone down the optimal path.

The environment can be static or dynamic,

1.1 The problem of intelligent systems (module 2)

The definition of the problem to be solved suggests the drone needs to be an agent. Agents are autonomous computational entities, as they are entities that compute and agents are autonomous, "in that they encapsulate control along with a criterion to govern it" [Omicini, 2022b]. Our drone is indeed a computational entity, and it is also truly autonomous in the sense that it is given a goal and boundaries to work in (for instance, obstacles to avoid). We do this with direct instructions.

The software used to run the simulator, give instructions to the drone and to test the different runs can become very complex. "When software systems grow too complex, decomposition makes it possible to articulate them as ensambles of distinct, interacting components" [Omicini, 2022a]. Decomposition can be done in multiple different types of abstractions. It could be done into functions, classes, procedures, or agents. The distinguishing factor of agent programming is that rather than follow direct instructions, it reacts to it's environment (see Figure 2). This environment can be real or simulated.

In the end, our goal is to discover and implement software solutions for the agent that the drone is. That's why the focus of the problem for this report is going to be on just the agent that encapsulates a functionality separate from the rest of the software systems, though we will discuss parts of the software system the agent interacts with. This we will do for instance in subsection 3.4 and subsection 1.2.

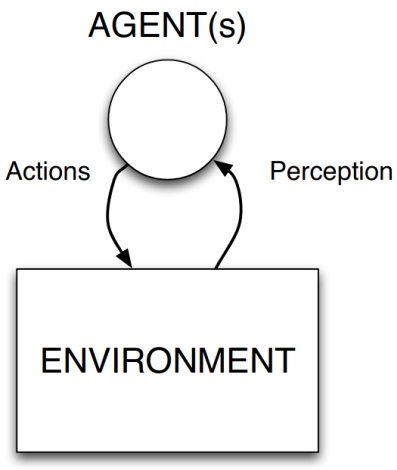

Figure 2: The simplest agent [Omicini, 2022a]

1.2 The simulator

Flightmare is the simulator we used which is implemented in ROS. In Figure 3 the simulator world is shown. There are 4 different worlds in which the drone can be tested, although the competition map is the one that is shown.

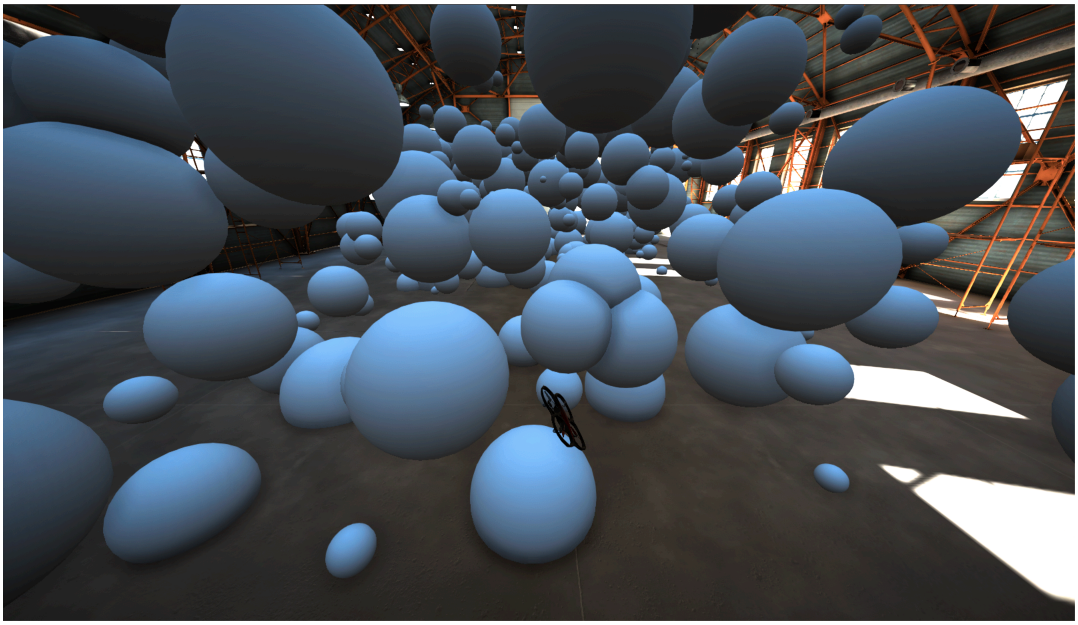

Figure 3: Environment

2 Development strategy

2.1 Domain-Driven Design

We applied the principles of Domain-Driven Design to go from problem to solution. DDD is an approach to software development . The central theme of DDD tools and practice is effective communication [Ricci, 2022a]. It consists of two parts: strategic design and tactical design.

Strategic design is most concerned with the ”what?” and ”why?” of software development. It is about the structure of the problem, the context in which the problem exists, what stakeholders are involved et cetera. For us, this means analyzing the ”problem space”, looking at why in the first place this problem needs to be solved.

Tactical design is concerned about the ”how?”, the implementation of the solution. For us, this means looking into how other researcher or companies have tried to solve this problem, gathering domain knowledge in the process.

The main way domain knowledge was discovered by looking at recent research papers on drone obstacle avoidance. Research aggregates include Google Scholar and IEEE Xplore. Most research seemed to be focused on machine learning approaches, specifically using reinforcement learning.

2.2 Ubiquitous language

Often the hard part of this process was the translation from the scientific jargon behind the mathematics involved in the technical solution to a software solution. To solve this problem, in DDD we use ubiquitous language. This is a language that is used to describe the business domain. The idea behind it is you avoid technical jargon but use business language to describe the domain the business is working in. In section 3 we will use a ubiquitous language to explain the solution, avoiding technical language when explaining the non-technical part of the approach. In the technical solutions however, technical details will be explained as if written to someone with technical knowledge.

2.3 Methodology

We used the 'Agile' methodology when developing our solutions to the problem. Specifically we strictly followed the 'incremental development' approach of programming. The main idea is that instead going through the four main software engineering activities sequentially (specification, development, validation, evaluation) and only then possibly returning to an earlier stage, these activities are happening concurrently. The idea is to create an

initial version quickly, iterate and test all intermediate version in order to deliver a final version that just works. This is visualized in Figure 4.

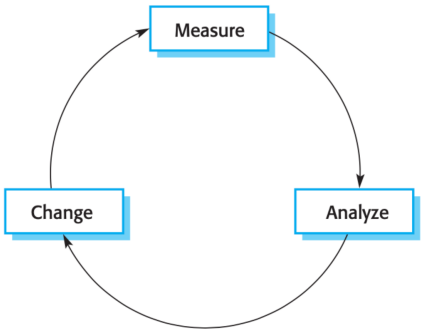

Figure 4: The general process improvement lifecycle [Ricci, 2022b]

After informally creating the specification (first don't touch obstacles, second get the fastest time) we worked on getting an initial version working on our computers. After this came many intermediate versions, during which the solution would be validated. At the end we submitted a final version that was validated to be the best solution we have. This process is visualized in Figure 5.

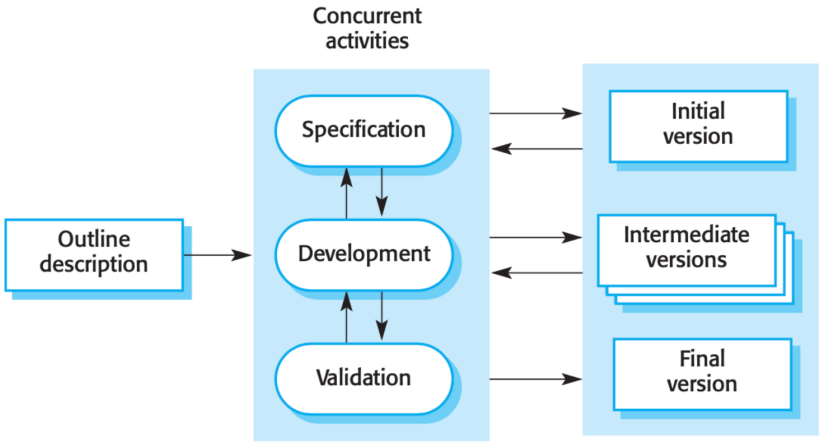

Figure 5: Incremental Development strategy [Ricci, 2022b]

2.4 Using Git

We used Git and GitLab as a version control system. While working on different solutions, a new branch would be created. Each commit would be a new iteration of the solution.

This process made our lives much easier when working on solutions. Instead of having to constantly send our 'latest' versions to each other, we had the latest version always updated in the repository on Gitlab. Instead of having a signel place where multiple people want to try different approaches in the same code, we used different branches with different approaches so that one person updating his or her approach won't interfere with the others. Just keeping track of different iterations was a critical feature for us.

Our progress over time can be seen in Figure 6.

Gitlab was used because we like that the system is open source (for security reasons), and because we already had a group in Gitlab from previous projects, so adding a project to the group was really easy.

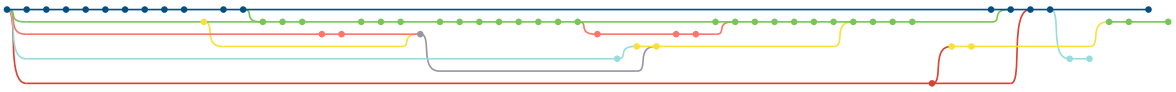

Figure 6: Our commit graph over time
Light blue is approach 1, dark blue approach 2, green is approach 3.

3 Development process

3.0 Getting started

For the initial version, our goal was to set up the coding environment on our computers and getting control of our drone.

3.1 Approach 1: if/else

3.1.1 Description

To get used to the simulator and more importantly to get a feeling of the drone's behaviour, we started with a relatively simple approach. While working on the code we learned a couple of important factors that were significant for later versions. We tried to code our own algorithm from scratch without relying on any research papers, to develop a good feeling on what is important. This should help us to figure out which research paper algorithms should work good and which one may not, because there are many different scenarios and environments that are quite different to our case.

We used the state based approach with linear velocity steering as it was suggested to start with that at the drone challenge wiki. With the state based programming we can get current data of the 10 nearest obstacles. This data includes the relative X, Y and Z distances of each obstacle and their radii (scale), which is structured in a list. In Figure 7 you can see an example data of one obstacle.

```
position:
    x: -4.440182860028466
    y: 2.636502577531738
    z: 2.050067668302656
  scale: 1.3034376754914525
```

Figure 7: Obstacle data

Unfortunately the order of all 10 obstacles in the list is random and we had to calculate the one that is nearest to the drone. With the X, Y and Z distances we can calculate the resulting distance with the calculation of Pythagoras which is visualized in Figure 8.

To make the drone look more forward and therefore more ignoring obstacles on its side, we implemented a little algorithm: we are multiplying the Y and Z distances (left/right and up/down) by an adjustable factor (in our case 4 works good, see Figure 9) to get a longer resulting distance for these obstacles. That's why these obstacles are then not anymore considered as closest obstacles whereas others, that may be further away but in front of the drone, are. Besides that we always have to subtract the radius of the obstacle (obstacles are spheres) to the calculated distances because the distance was previously calculated from the drone to the center of the obstacle.

Now that we have the most important obstacle we can control the drone related to it. First we check if this obstacle is directly in front of us (see Figure 10).

Furthermore we check if the drone is on the left or right side of an obstacle. According to that we steer the drone so it moves away from it. For example if the drone is in front of the upper right part of an obstacle, it moves up and right at the same time because that is the shortest path to avoid it. The same behaviour is programmed so that if the drone is in front of the lower part of an obstacle, the drone steers downwards. This behaviour is visualized in Figure 11. In case there is no obstacle in front of the drone, it only goes forward, also at a higher speed to save time.

To avoid hitting the ground while the drone is flying, we set a threshold value. Once the drone comes below it, it tries to increase height again. Find the code in Figure 12.

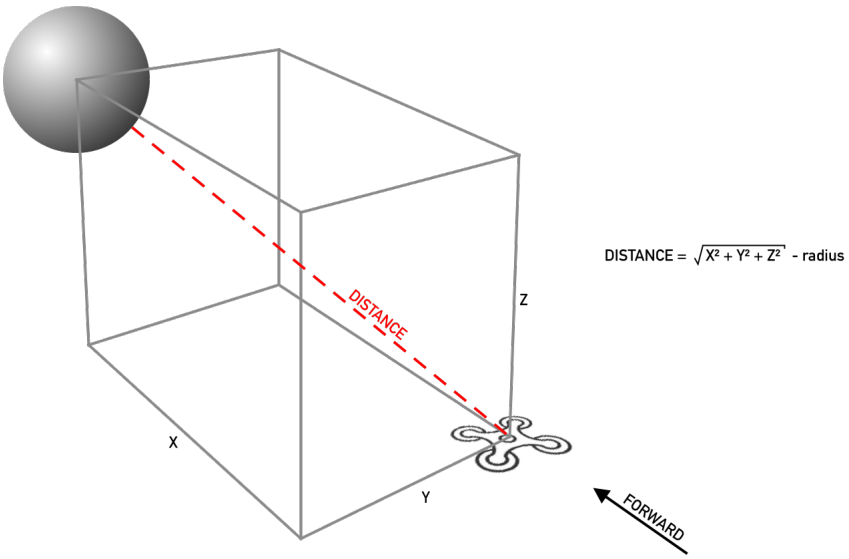

Figure 8: Calculating the distance

```
obstacle_x = obstacle.position.x
obstacle_y = obstacle.position.y * 4
obstacle_z = obstacle.position.z * 4
```

Figure 9: Distance algorithm

```
if abs(nearest_obstacle.position.z) < nearest_obstacle.scale and nearest_obstacle.position.z < 0:
    ...
    ...
elif abs(nearest_obstacle.position.z) < nearest_obstacle.scale and nearest_obstacle.position.z > 0:
    ...
    ...
```

Figure 10: The code to check if we are in front of an obstacle

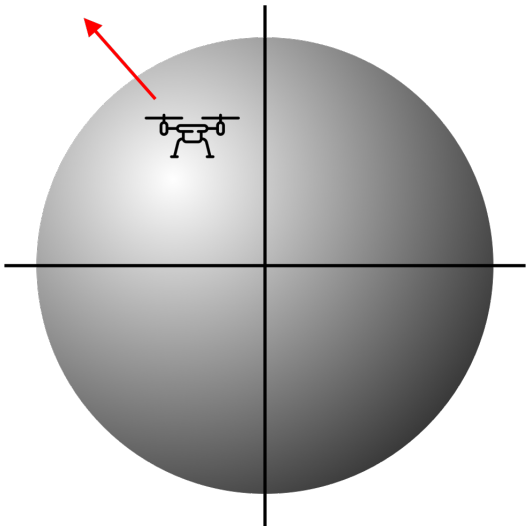

Figure 11: Avoidance behaviour

```
# check if we respect floor safe distance
if state.pos[ARRAY_Z_INDEX] < floor_safe_distance:
    command.velocity = [command.velocity[0], command.velocity[1], 1]
```

Figure 12: Check floor distance

3.1.2 Result

The first half of the world consists of fewer obstacles than the second half. In the beginning the drone flies quite confident, stable and fast without any crashes. The further it goes, the more obstacles appear which makes it harder for the drone to avoid crashes. We had to fight a bit with the problem that the world's random obstacle generator is not really random, which will be described in subsubsection 3.4.1. That is why the drone often takes the same path and we adjusted the parameters according to it. In the rare case that the drone takes a different path, it is a lot more unstable and it will likely crash into obstacles. In the second half of the world the drone still often makes it through in a way that is totally acceptable for this approach. Often it only has about 0 to 2 hits. The main problem is that we only consider just one obstacle at a time. This makes it impossible for the drone to avoid some spheres in specific situations. The algorithm is just not smart enough for situations where a lot of obstacles are surrounded closely by the drone. We found a speed that scarifies the consistency just a bit and so we achieve times at around 14 seconds.

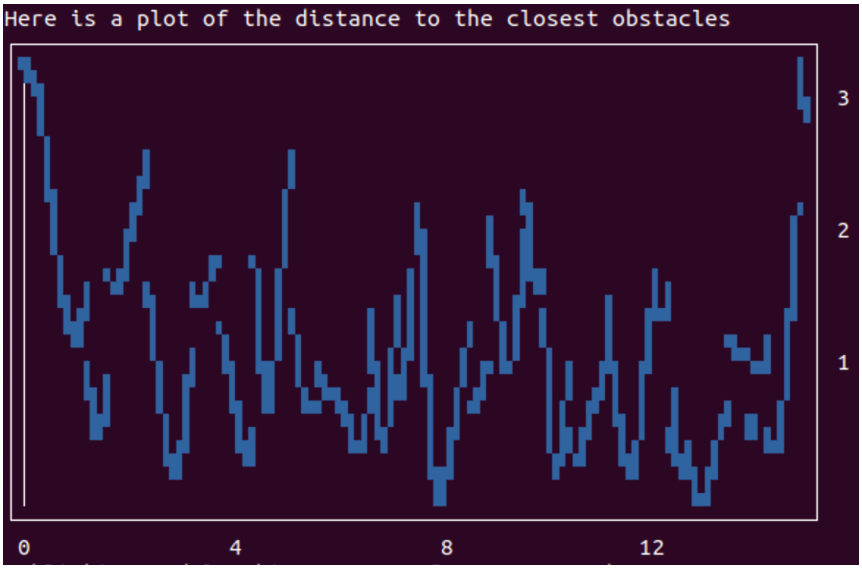

Figure 13: Typical result of approach 1

3.1.3 Possible improvements

All obstacles in the world are spheres, although currently we do not consider them as such. In this approach we are only calculating the outline borders of the spheres as cubes (see Figure 14). For smaller obstacles this may not make a huge difference but for bigger ones it would. In the end we did not implement it that way because we wanted to concentrate on overall more promising algorithms.

A second point that makes this approach quite inconsistent and unstable is that it only considers just one obstacle at a time. Especially when there are many obstacles around the drone, there often is not just one most important one but a couple of important ones. Imagine an obstacle in front of the drone which tells the drone to go left, but next to that sphere is another one that makes the drone to move even further. In that case it would be better to detect also the second obstacle for the drone to think more intelligent and move right. Flying right may take more time at first view but the drone could then have a free sight earlier (see Figure 15).

Also, let's consider an obstacle that makes the drone fly left. But, right next to that sphere is another one so that the drone has to move back right. This makes the drone to stay in the middle of these two obstacles and it would crash. The only way to get out of this situation is to fly up or down, but if we are unlucky and there are even more obstacles interlacing with the other ones then we would definitely crash (see Figure 16). To overcome this

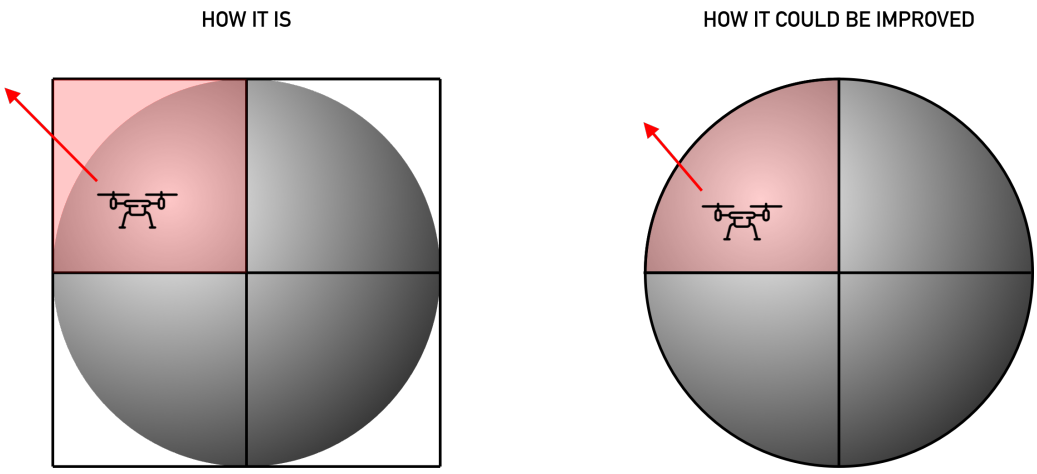

Figure 14: Behavior of avoiding obstacles

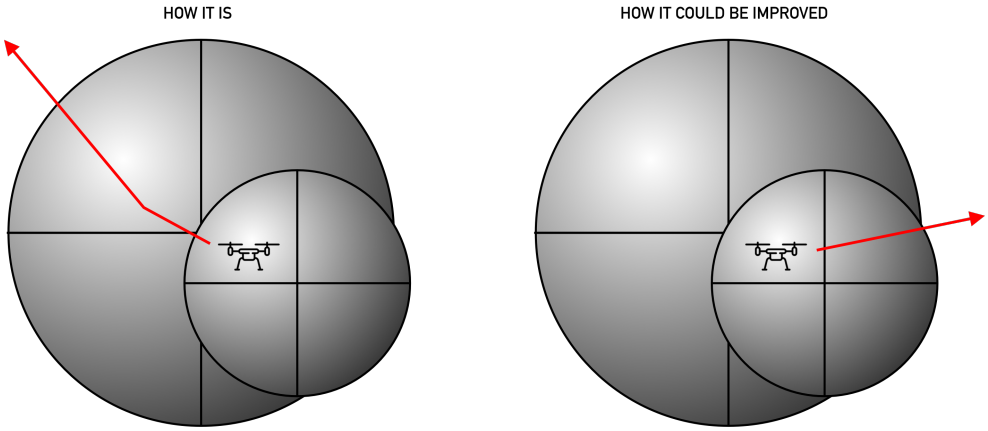

Figure 15: Behavior of avoiding multiple obstacles

problem a little bit, we are slowing down the drone if there are obstacles in front of it to give it more time to react. If there is a free view so that nothing is in front of the drone, it will fly twice as fast.

Our approach to detect the most important obstacle was greatly improved by the factor we added to the obstacles that are not really in the view of the drone, so they are considered as less important. Especially if the drone is flying at higher speeds, the more it should "look forward" and ignore obstacles on its side, although they may be closer. Although this worked good for most cases, there is still a lot of room to improve the algorithm to decide which is the most important obstacle. For example we could distinguish between moving and static obstacles because dynamically moving spheres are of greater importance than still standing ones. In some cases a fast moving obstacle crashed into the drone because the time to react for the drone was just not long enough. If we could have detected this obstacle a bit earlier because of the fact that it was moving, we could have avoided this situation. Considering this case, the drone has to move even before the obstacle which is in front of it. This could be implemented by giving moving obstacles a larger size than they actually have (see Figure 17).

Furthermore, the speed of the spheres could be related to the size multiplier. That means, faster obstacles are considered larger than slower ones (see Figure 18).

Detecting moving obstacles could be a bit tricky though, because their given X, Y and Z coordinates are relative to the drone and not absolute. That is why we also would need to look at the speed of the drone to compare it with the obstacle's velocity.

Another flying behaviour that could be improved is a variable speed of the drone. When there are not a lot of obstacles around the drone or in other words, if the distance from the drone to one or more obstacles is long, it should fly faster in order to save time. On the other hand, the drone should fly slow when it is in a tricky situation (see Figure 19).

Not only the forward speed of the drone should be variable but also the up/down and left/right movement

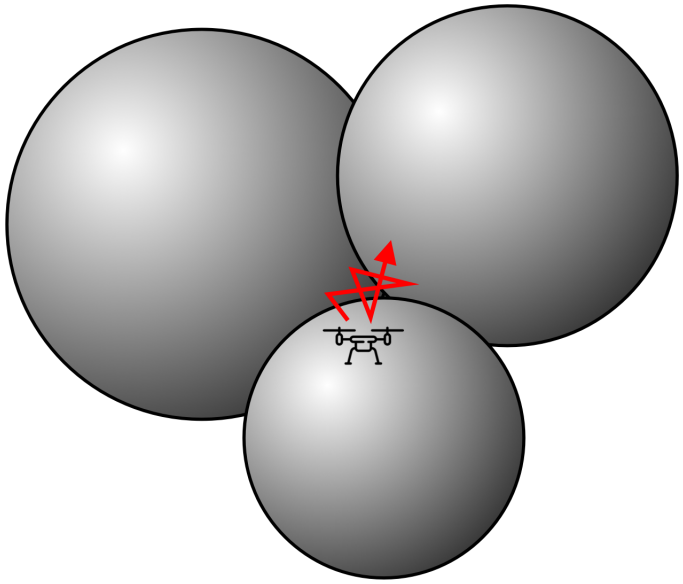

Figure 16: Unlucky situation

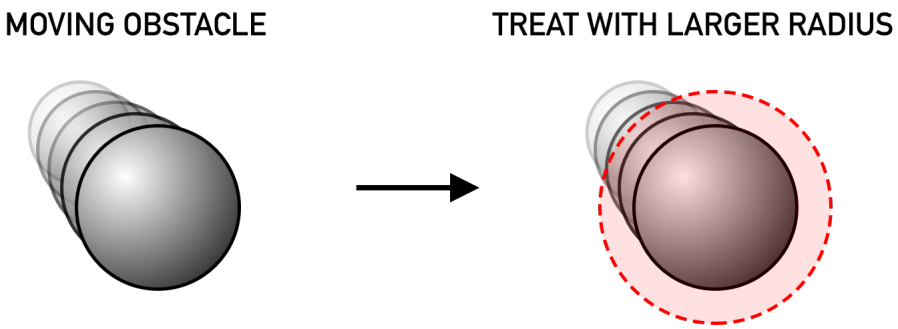

Figure 17: Treating moving obstacles differently

speed. In particular, sideways- and up/down-movements should always be connected to the forward speed. It does not make a lot of sense to go quite slow forward but fast in all other directions. This would result in an unstable behaviour of the drone.

Some of the mentioned improvements have been considered for further versions or different algorithms.

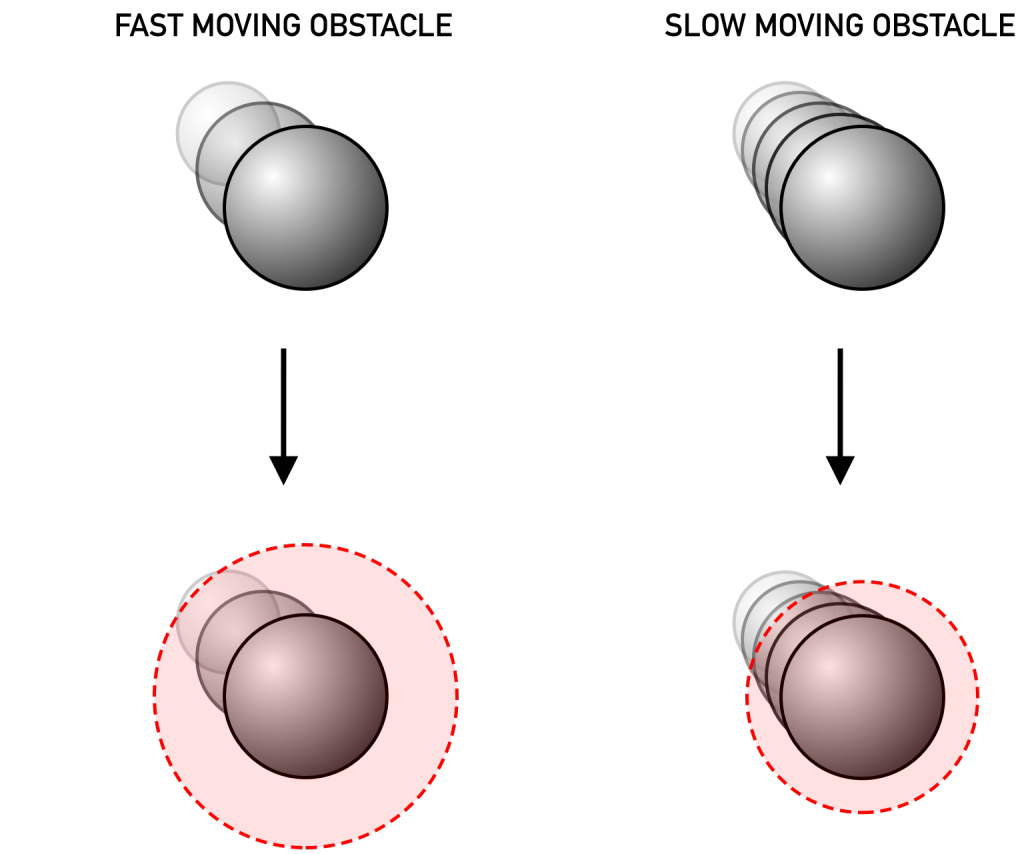

Figure 18: Radius related to obstacle speed

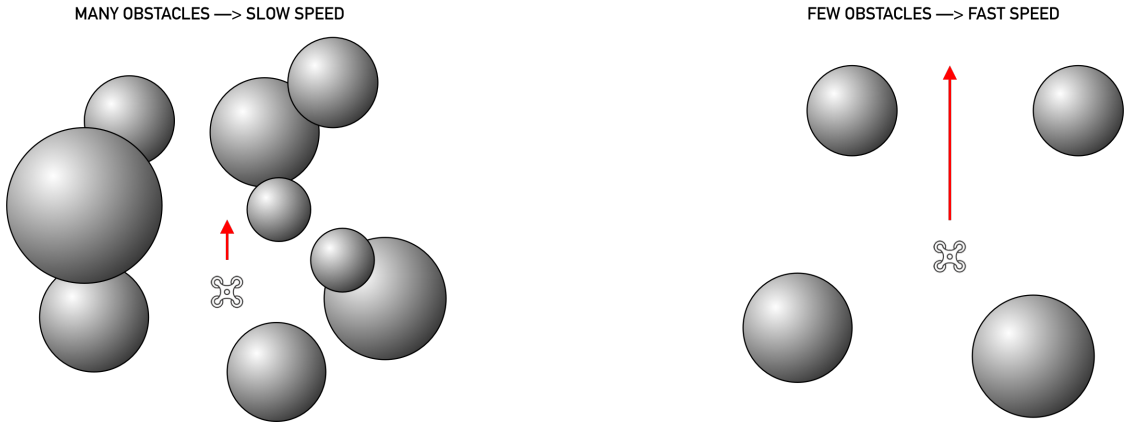

Figure 19: Variable speed

3.2 Approach 2: if/else with more obstacles

3.2.1 Description

This approach is a simple improvement on the first one mentioned in subsection 3.1. After working on the first approach for a while we got to the point that it was good, or at least not completely bad, so we started to work on it not only as a way to get confident with the drone environment but also as a possible candidate to win the challenge. Even though the first approach was good, it had some problems when there were many obstacles. We thought that considering only one obstacle at a time to figure out the direction to move is not enough for our goal. In this approach we used the same structure and logic of the previous one, but we considered the three nearest obstacles instead of just one. In terms of drone behaviour, the strategy of considering 3 obstacles

to figure out the movement direction consist of moving the drone in zones where there are less near obstacles and at the same time avoiding the nearest one. In particular the algorithm to figure out the movement direction consist of getting the direction coordinates (x, y, z) for avoiding each of the three obstacles. Then we adjust the left/right and up/down speed by a factor based on the distance of these obstacles. The smaller the distance to the obstacle is, the higher the speed will be. The resulting value is the sum of each of the three directions speed. In this way the drone will tend to avoid the first obstacle choosing a direction in which it will be placed far from the other two. In Figure 20 you find the visualization.

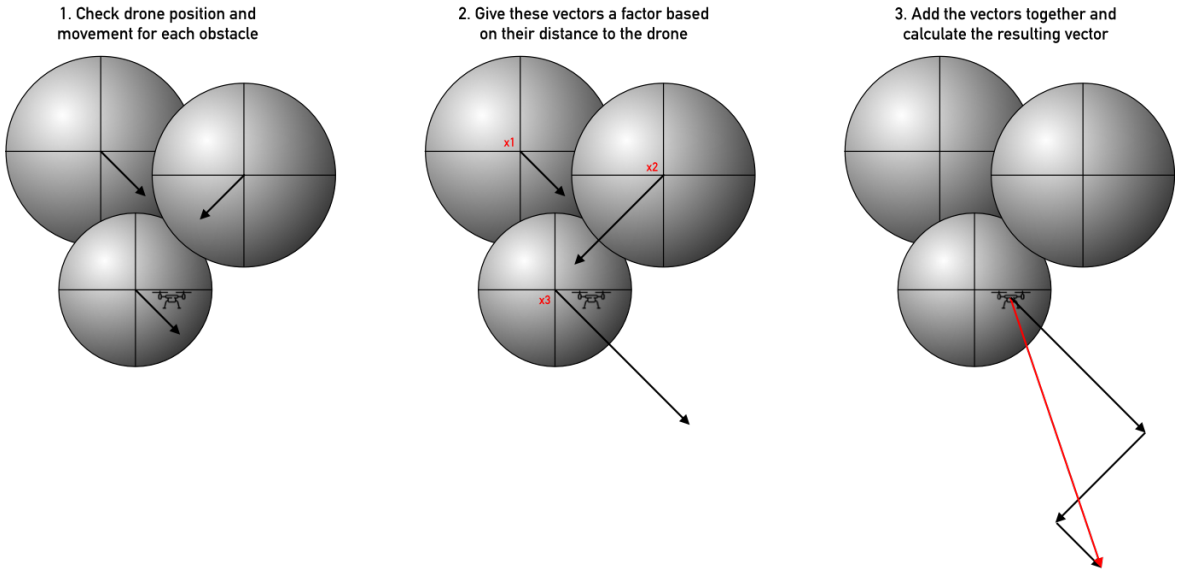

Figure 20: Drone behavior on the second approach

3.2.2 Result

Using this approach the drone was able to get to the end in less than 11 seconds without any collisions. Unfortunately this behaviour was not consistent because the drone still collided sometimes. Especially moving obstacles could not be fully avoided. From observing the behaviour, the problem seemed to be our algorithm which considered the obstacle as ”dangerous” only when it is close to the drone, but the moving obstacles are dangerous even if they are not so close because they might move towards the drone quickly. To consider moving obstacles in the same way as the static ones does not avoid a collision. Another problem was that we do not consider more than 3 obstacles in this algorithm. This means that in a place with a lot of obstacles (more than 3), avoiding the drone could collide with a not so near obstacle while it tries to avoid the nearest one.

3.2.3 Possible improvements

The more straightforward improvement is to increase the number of obstacles to be considered in the algorithm. In this case also the parameters that regulate the algorithm behavior should be updated in order to fit the new amount of obstacles considering number. Secondly we thought a good idea for assuring the avoiding of in movement obstacles was to split the algorithm in 2 different obstacle avoidance approaches. The first one focuses only on the static obstacles and the second one focuses only on moving obstacles.

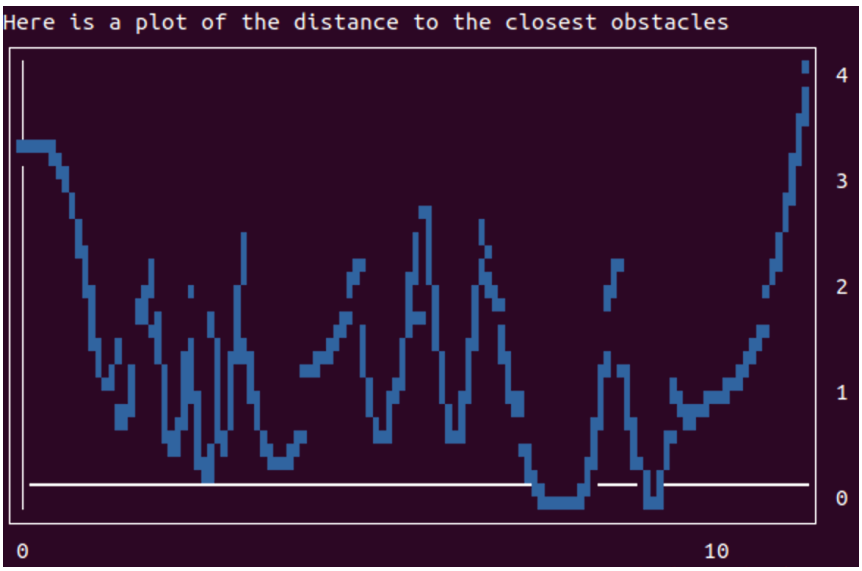

Figure 21: Typical result of approach 2

3.3 Approach 3: research paper

3.3.1 Description

This approach is based on the paper from Ngyuen et al. (2018) [H. Nguyen et al., 2018]. The paper describes a novel approach to define a path for the drone to fly (path generation), also considering any obstacles that might be in the way of the path. Then, based on the defined path, the direction and speed is determined. In our case obstacles might be moving (they are dynamic), so the path is constantly regenerated and followed.

On the technical side, it works as follows. Any 3D path can be defined as the intersection of functions $f_i(x, y, z)$ where $x, y, z \in \mathbb{R}$ and $i = 1, 2$, for example when $f_0(x, y, z) = f_1(x, y, z) = 0$. In our space, x is forward/backwards, y is right/left, z is up/down. The idea is that $f >= 0$ and it indicates an 'error' function.

For obstacles, we need to 'bend' the path in order to increase f whenever it is close to an obstacle. For a path without obstacle f is enough, but now let's define a function that modifies the path such that its starts avoiding obstacles. Define N as the number of obstacles. The obstacle function $O_j(x, y, z)$ where $j \in 1, 2, \ldots, N$ has as requirements that it should be much larger than 0 when our position (x, y, z) is very close to the center of the obstacle (x_j, y_j, z_j) and that it should be approximately be 0 when our position is far away from the obstacle.

Now, we can add all the obstacle function to the path to modify it.

$$f'(x, y, z) = f(x, y, z) + \sum_{j=1}^{N} O_j(x, y, z)$$

Then, based on the gradient function of f' we can calculate the direction 'down the gradient' and use that to direct the drone.

Using f', we calculate the speed and direction we want to move (the velocity vector). Depending on the iterations we do this calculation differently. In general, it looks like this.

$$V(x, y, z) = -k_{grad} f'(x, y, z) \nabla f'(x, y, z) + t(x, y, z)$$

Here f' is a factor of the speed, k_{grad} an independent scalar we can tweak, $-\nabla f'$ the gradient vector with its direction inverted, and t a tangent function along the path. A visualization of our calculation of V in 2D can be found in Figure 22.

3.3.2 First iteration

In the beginning we wanted the drone to in general go to the finish while staying in the middle and staying around 2 meters high. In terms of equations, this is

$$f_1(x, y, z) = y = 0$$
$$f_2(x, y, z) = z = 2 \implies z - 2 = 0$$

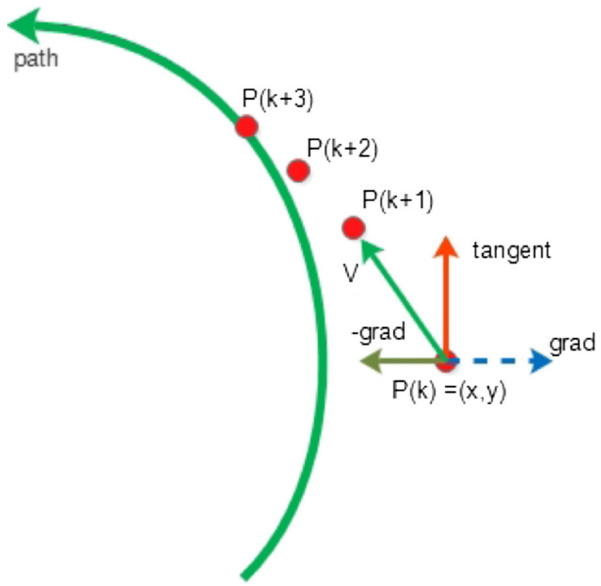

Figure 22: Path generation and following [H. Nguyen et al., 2018]

Combining them gives many possible solutions. Since f is ultimately used as an absolute error function, we settled on $f(x, y, z) = y^2 + (z - 2)^2$ to that it is always greater than 0 and the 'error' increases superlinearly.

For the obstacle function, we used the one suggested by the paper, namely a Gaussian function described as:

$$O_j(x, y, z) = A_j e^{-\frac{(x-x_j)^2+(y-y_j)^2+(z-z_j)^2}{\sigma^2}}$$

For σ we used the diameter of the obstacle (as they were spheres), and experimented with $A_j = 15$ which we left as the same constant for all obstacles.

The tangent function was $t = (-(\nabla f')_y, (\nabla f')_x, -(\nabla f')_z)^T$ along with a base speed forward in x-direction.

3.3.3 Second iteration

Next, we started to give the drone more room by allowing it to move anywhere in the range $-10 > y > 10$ and $0.5 > z > 5.5$ without increasing f. We started to not consider any obstacles that were behind us. In this iteration we've changed k_{grad} and A_j a lot, and adjusted the way the gradient function was calculated (as it was done numerically, changing step size for instance). The tangent function t we used was just vector $(5, 0, 0)^T$ directed straight forward as a 'base speed' and $k_{grad} = 0.5$ was used.

3.3.4 Result

Considering the 'simplicity' of the path generation and object avoidance functions, the drone manages to avoid both static and dynamic obstacles surprisingly well using the latest iteration. With just under 20 lines of functional code, hence 'simplicity', the drone is able to clear the course in around 12 seconds and usually with 0-2 crashes. With the latest k_{grad}, A_j and gradient step size adjustments the drone flew the course two times out of ten without any crashes, six times with one crash and twice with more than one crash. As we were using a gradient function from the numdifftools python package, the calculated gradient made the drone behave rather weird in specific cases. For example, usually when it would have been more beneficial for the drone to avoid the obstacle going under it, we got results from the used gradient function instructing the drone go higher up and thus resulting to a crash.

3.3.5 Possible improvements

Even though drone was clearing the course acceptably well using this approach based on the paper from Ngyuen et al. (2018) [H. Nguyen et al., 2018], there is some room for possible improvements. As already the initial iteration of the algorithm managed to guide the drone around the course finely, there are no significant improvements to be considered to the logical part of the avoidance algorithm. But what comes for the smaller adjustments, we

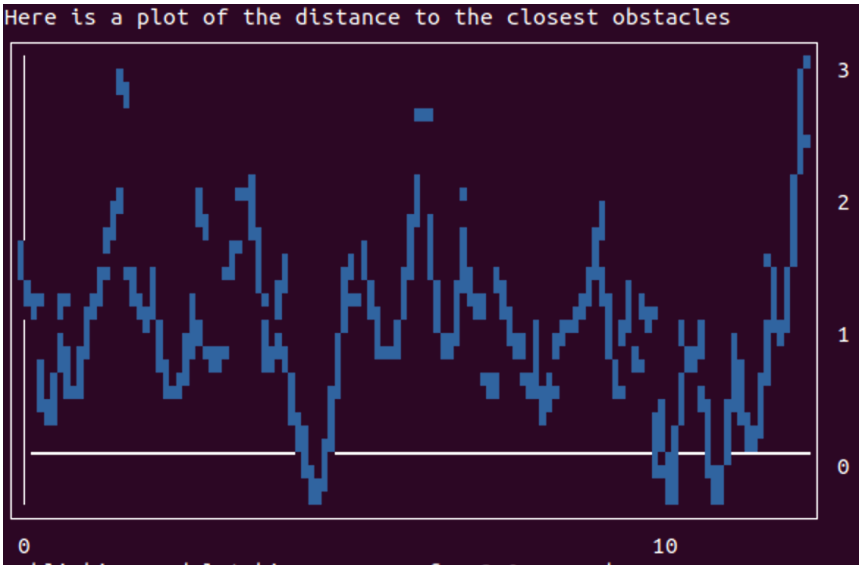

Figure 23: Typical result of approach 3

have three suggestions. First improvement would be creating an own gradient calculation function, that would give different 'weights' and calculation method for different axes, thus improving the accuracy and the speed of the object avoidance movement. Second improvement could include fine-tuning the variables (k_{grad}, A_j, speed, gradient step size and gradient order) used in the algorithm. This improvement should be automated by machine learning; artificial intelligence would be rewarded for changing the values of the variables to find most effective way to clear the course without any crashes. Last improvement is introducing the improved tangent function t to calculate the most successful path to follow after avoiding the obstacle.

3.4 Simulation difficulties

3.4.1 The world is not really random

There are static and dynamic obstacles in the world which should be placed at a random position every time. This did not seem to work properly: the static obstacles always were at the same positions and the dynamic ones only change their starting position a little. This results in an environment which is quite similar most of the time. That is why the drone often takes the same path over and over, which is not ideal for testing. Only in rare cases the drone took different paths and then it behaved completely different. We ended up adjusting some of the parameters in our algorithms so that it was ideal only for the given world. It even would have been possible to hard-code the drone for specific parts of the map. Furthermore, we didn't know if the world is completely different for the competition environment, which made it even harder to find the right parameters.

3.4.2 Lack of visualisation

While running the simulator, we often had problems understanding what was going on. We wished to have a couple of different visualisations tools to iterate faster. For example, it would have been useful to see the calculated path that the drone would follow. Also, seeing the calculated most important obstacles in a different colour would have facilitated our testing procedure.

References

[H. Nguyen et al., 2018] H. Nguyen, P. D., Recchiuto, C. T., and Sgorbissa, A. (2018). Real-Time Path Generation and Obstacle Avoidance for Multirotors: A Novel Approach. Journal of Intelligent & Robotic Systems, 89(1-2):27–49.

[Omicini, 2022a] Omicini, A. (2022a). Agents for intelligent systems engineering.

[Omicini, 2022b] Omicini, A. (2022b). On autonomy concepts & definitions.

[Ricci, 2022a] Ricci, A. (2022a). Domain-driven design (ddd) - strategic part.

[Ricci, 2022b] Ricci, A. (2022b). Software processes and engineering activities - bird's eye overview.