Camera Placement Optimizer for Optitrack System Documentation

Pázmány Péter Katolikus Egyetem

Kispál Benjámin

2023

Contents

1 Introduction

At the SZTAKI AIMotion Lab, there is a branch of research that focuses on creating autopilot functionality for small indoor drones. Currently the positioning of the drones is achieved using the Optitrack motion capture system. IR cameras surround

the space where the drones are expected to fly, with the criterion that the markers on the drones must be visible by at least 3 cameras for the position to be determined.

My goal is to create a tool to optimize the positioning of the cameras. To achieve this, I created a model for a camera, with the assumption that a camera can see everything inside a pyramid, without any regard for focus distance or lens distortions. While these assumptions may not be entirely accurate, optimizing within these circumstances should give a reasonable baseline for how the cameras should be set up.

Figure 1.1: The Crazyradio PA dongle, Crazyflie 2.1 with motion capture deck and Optitrack markers, and an Optitrack Prime 13 camera.

2 The model

The model was created with the help of GeoGebra geometry calculator and visualizer suite. It is easier and less error-prone to figure out a functional model restricted to 2 dimensions, and later expand it to 3D. The 2 dimensional slice of a pyramid is a triangle. Figure 2.1 highlights all the vertices and edges that are necessary to calculate the edges of a general triangle, given a height, an FOV value for the camera, and an arbitrary tilt.

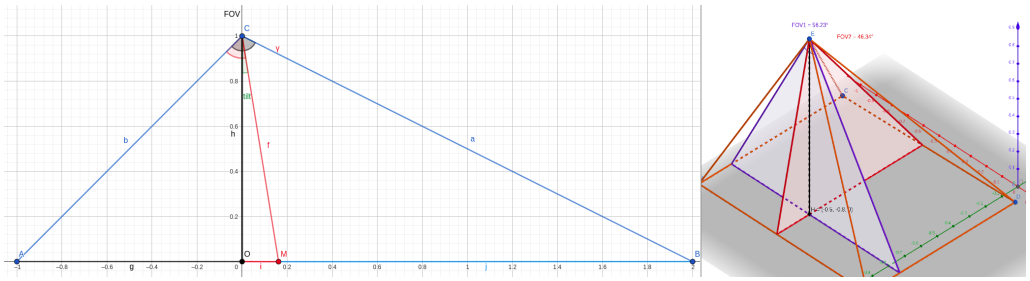

Figure 2.1: Left: sketch of a triangle, decomposed in a way that its properties can be calculated from the initial parameters (height, FOV and tilt). Right: a pyramid made in GeoGebra decomposed in a helpful way.

With the known/measurable factors being the FOV, tilt and height of the camera, the side lengths can be calculated as follows:

AOC triangle:

- $\alpha = \frac{FOV}{2} - tilt$
- $\mathrm{b} = \frac{\mathrm{h}}{cos(\alpha)}$
- $\mathrm{g} = \mathrm{h} * tan(\alpha)$

OMC triangle:

- $\mathrm{i} = \mathrm{h} * tan(tilt)$
- $\mathrm{f} = \frac{\mathrm{h}}{cos(tilt)}$

COB triangle:

- $\beta = \frac{FOV}{2} + tilt$
- $\mathrm{a} = \frac{\mathrm{i+j}}{sin(\beta)}$
- $\mathrm{i} + \mathrm{j} = \mathrm{h} * tan(\beta)$

Coordinate of vertices, with $\mathrm{C} = (x_0, \mathrm{h})$:
$\mathrm{A} = (x_0 - \mathrm{g}, 0); \mathrm{O} = (x_0, 0); \mathrm{M} = (x_0 + \mathrm{i}, 0); \mathrm{B} = (x_0 + \mathrm{i} + \mathrm{j}, 0)$

Extending this to three dimensions, we have tilt angles for two distinct axes: pitch and yaw and two separate FOV values. As figures 2.1 and 2.2 show, the positions of the vertices of the pyramid can be calculated by taking two perpendicular slices at the top most vertex (which is the position of the camera). Assuming $\mathrm{E} = (0, 0, \mathrm{h})$, the vertices of the pyramid are at $A = (\mathrm{A}''.x, \mathrm{A}'.y, 0)$; $B = (\mathrm{B}''.x, \mathrm{A}'.y, 0)$; $C = (\mathrm{A}''.x, \mathrm{B}'.y, 0)$; $D = (\mathrm{B}''.x, \mathrm{B}'.y, 0)$
After substituting, the vertices of the pyramid when the top is NOT at the origin ($\mathrm{E} = (\mathrm{E}.x, \mathrm{E}.y, \mathrm{h})$) are at:

- $\mathrm{A} = (\mathrm{E}.x - \mathrm{h} * tan(\frac{FOV_h}{2} - pitch), \mathrm{E}.y - \mathrm{h} * tan(\frac{FOV_v}{2} - yaw), 0)$
- $\mathrm{B} = (\mathrm{E}.x + \mathrm{h} * tan(\frac{FOV_h}{2} + pitch), \mathrm{E}.y - \mathrm{h} * tan(\frac{FOV_v}{2} - yaw), 0)$
- $\mathrm{C} = (\mathrm{E}.x - \mathrm{h} * tan(\frac{FOV_h}{2} - pitch), \mathrm{E}.y + \mathrm{h} * tan(\frac{FOV_v}{2} + yaw), 0)$
- $\mathrm{D} = (\mathrm{E}.x + \mathrm{h} * tan(\frac{FOV_h}{2} + pitch), \mathrm{E}.y + \mathrm{h} * tan(\frac{FOV_v}{2} + yaw), 0)$

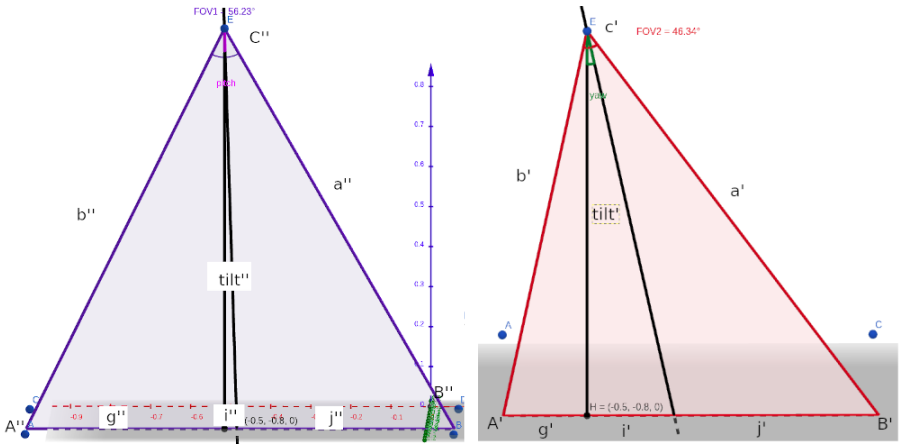

Figure 2.2: The two perpendicular slices of the pyramid, made to be analogous to the triangle in figure 2.1

3 Implementation

The Camera object creates a model of a camera, described above 2. Its main feature is its ability to determine whether it can see a point, or a set of points. When initialized, the Camera object is given the most upper vertex of the pyramid and the pitch and yaw angles. From these, the object determines all other vertices of the pyramid, as well as the coefficients for the planes that fit the faces of the pyramid. The function that determines whether a point is visible checks the inequality

$$\sum_{i=1}^{3} \theta_i * \mathrm{P}_i < \theta_4$$

where θ_i and P_i are the coefficients of the plane and the position of the point in question, respectively . If the inequality holds true for all the planes, the camera can see the point. [1]

The fitness function within the vanilla *CameraOptimizer* class creates a homogeneous distribution of points in space and - given a set of cameras - calculates the ratio of points that can be seen by at least 3. There is a *WeightedOptimizer*, the fitness function of which also takes into consideration a set of weights that it considers for the fitness value of a set of cameras. Depending on the weights there is a maximum value that can be achieved. The function calculates and normalizes by this value, so that the range of the fitness function remain $(0, 1)$ [2]. The *WeightedOptimizer* inherits from the *CameraOptimizer*.

[1] Depending on the ordering of the vertices, the relation sign may be flipped: this was taken into consideration within the code

[2] There is a *spread* option that tries to spread out the cameras as much as possible. I couldn't think of a way to implement this functionality without ruining the range and so unless this option is turned off, the $0 < loss < 1$ range cannot be expected

The CameraOptimizer object sets the global parameters (the dimensions of the physical space) for the optimization when initialized. There is a function for generating a random set of cameras withing the boundaries set by the global parameters, which works by scaling a uniform distribution. The fitness function is implemented within the optimizer class. The optimization is done using this implementation of CMA-ES, which is a non-linear optimization technique, the main benefits of which are that there is no need for gradient, the loss/fitness function doesn't have to be continuous, and the easy handling of constraints.
There is also a symmetric version of both of these optimizers, that reduce the number of free parameters and along them the dimensionality of the problem. This saves on runtime (in the case of 14 cameras 11 minutes instead of 40), while generally achieving the same quality. The symmetric optimizers have some caveats however, which is described in more detail at the end of section 5.

4 Usage

The optimizer package provides 4 classes for optimizing the positioning of the cameras. [1] The most basic is called *CameraOptimizer*. Every other optimizer class is just an extension of *CameraOptimizer*, and so that is what I will use in the example code.

```
1  import optitrackoptim as opt
2
3  # Define global variables for ease of use
4  FOV_H, FOV_V = np.deg2rad(56), np.deg2rad(46) #from the
       datasheet of the cameras
5  HEIGHT = 3. #meters
6  X_LEN, Y_LEN = 5., 7.5 #meters; length and with of the space
7  N_CAMERAS = 7 #number of cameras we plan to use
8
9  # Create optimizer object; get a good setup and save it
10 optim = opt.CameraOptimizer((FOV_H, FOV_V), (X_LEN, Y_LEN,
       HEIGHT), N_CAMERAS)
11 optim.set_random_cameras()
12 optim.train()
13 if optim.fitness()>0.80:
14     print("More than 80% of volume is covered, saving camera
           setup")
15     optim.save('cam_setup.npz')
```

[1] Of the 4, currently only 2 works properly, anything involving symmetry produces unpredictable results (although most of the time it just runs forever.)

The *save()* function also has a *load()* pair, which can be used to load in previously trained results. The setup of the cameras currently stored in the *CameraOptimizer* can be visualized.

```
...
# Load an Optimizer from file
optim2 = opt.CameraOptimizer(2*(0), 3*(0), 0)
optim2.load('cam_setup.npz')

# Plot the orientation of the cameras and the visible points
import numpy as np, matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.add_subplot(projection='3d')

optim.plot_cameras(ax) # vizualize the orientation of the
   cameras
optim.plot_seen_points(ax) # show points from an even
```

The *plot_cameras()* plots the cameras in space and gives a visual feedback for their orientation, while *plot_seen_points()* generates a 3D array of evenly distributed points and plots only those seen by at least 3 cameras. There is an option for manually tuning the placement of cameras. In this case one could use the *Camera* object and its *plot_vertices()* function. There is also a *plot_axes()* function for drawing a colorful *(x, y, z)* axes, which is currently implemented in a way that it needs an *optimizer* object to get the data for the space from.

```
import optitrackoptim as opt, matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.add_subplot(projection='3d')

cam = opt.Camera((1.2, -0.5, 3), np.deg2rad(-25),
   np.deg2rad(15))
cam.plot_vertices(ax)

opt.plot_axes(ax, opt.CameraOptimizer(2*(0), (5, 7.5, 3), 0))
```

There are 5 possible weights implemented for the *WeightedOptimizer*: 'distance_from_origin'; 'stay_within_range'; 'spread'; 'soft_convexity' and 'hard_convexity'. There are soft and hard weights. The latter rejects solutions that don't meet its criterion. The former places importance to certain attributes of a solution. The soft weights compete with one another and their magnitude determines their importance. **All attributes can be turned off by setting their value to a negative number**.

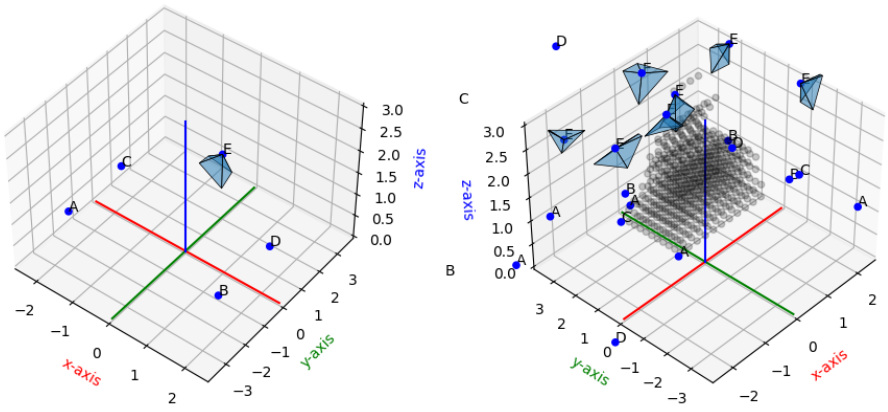

Figure 4.1: Left: plot of the single camera from the example code. Right: the points at least 3 cameras can see after calling the *set_random_cameras()* on an optimizer.

- *distance_from_origin* is a soft weighing, which assigns less weight to those points that are further from the origin, enticing the optimizer to prioritize the middle of the space. Its value can be a tuple $(magnitude, order)$, or a float $magnitude$, in which case $order = 1$. The magnitude determines the importance of the centering compared to other weights, while the order determines how strongly centered points are favored compared to peripheral points.

- *stay_within_range* is a hard weighing which rejects the solutions where the cameras look too much outside the defined space. If set to 0, all cameras' vision must stay within the specified space and in general, if set to a value of $0 < s$, the visible space must remain within the $(1+s)*length \times (1+s)*width$ plane.

- *spread* is a soft weighing, which prefers solutions where the cameras are more spread out. [1] **If this weight is used, the fitness function is not guaranteed to be within** $(0, 1)$**!**

- *soft_convexity* is a soft weighing, which estimates the ratio for the volume of the smallest convex cover where the visible points fit and the points that cannot be seen, yet are in the cover. From this, it calculates a value that could be called something like a 'convexity ratio'. This parameter entices the optimizer to maximize this convexity ratio.

- *hard_convexity* is a hard weighing, which rejects every solution that doesn't meet the convexity ratio specified. The range of this parameter is $(0, 1)$, where

[1] This was a solution to a problem encountered early-on in development, where the trained setups would group cameras very close together, which since has been resolved. Still, it's an extra choice.

Number of cameras:	4	5	7	9	11	14
Expected deviation fitness (%)	5	2.5	5.6	4.9	2.4	0.4
Expected deviation runtime (%)	25.5	19.2	24	10.1	12.1	28.2

Table 5.1: Table showing that there is a lot of noise/unpredictability to the runtime of an optimization.

0 accepts all solutions and 1 accepts only those that are convex. Of course a negative value can also be used, which turns off this weighing.

It is worth noting, that the parameters regarding convexity measurably slow down the evaluation of the fitness funciton.

Listing 1: Example for using the WeightedOptimizer

```
import optitrackoptim as opt
...
weights = {'distance_from_origin': (0.5, 1.0),
   'stay_within_range': 1.0, 'spread': -1.0,
   'soft_convexity': 2.0, 'hard_convexity': -0.4}
optim = opt.WeightedOptimizer(weights, (FOV_H, FOV_V),
   (X_LEN, Y_LEN, HEIGHT), N_CAMERAS, CAM_SIZE)
optim.set_random_cameras()
optim.train()
optim.save('weighted_setup.npz')
```

5 Results

Initially there was a bug in the code that restricted the pitch and yaw of the cameras, which yielded in relatively poor results: the cameras would see about 30-40 % of the space with the visible space being concave and very narrow at the top - which is where the drones fly. After this was resolved, most runs yielded much better fitness functions: 60%+ with just 6-7 cameras and a better overall shape (figure 5.1)

A notable result was that using symmetrical placement, 92% of the space could be seen, with only 4 cameras: figure **??** [1] . This is all according to the model at the moment and haven't yet been tried in reality just yet.

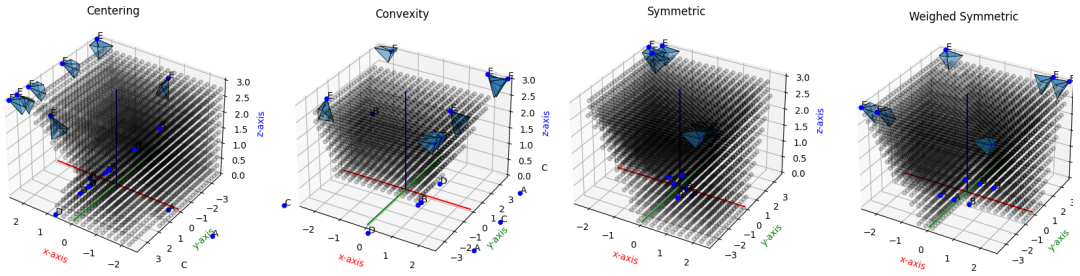

Figure 5.1: Comparing the optimizers. The **first** plot is of a *WeightedOptimizer* that was supposed to center the visible space - which it more-or-less did, but said space is very non-complex. The **second** considered only the convexity and indeed it achieved a seemingly perfectly convex shape, but the size of the visible space is only a quarter of the desired size. The **third** figure is from a *SymmetricOptimizer*, which did better than expected, achieving a larger visible volume than that of the first plot, and - funnily enough it seems more centered too. For this reason I chose *WeightedSymmetricOptimizer* to optimize for convexity, with partial success. The **fourth** plot shows a volume that is somewhat convex near the top, but has weird edges at the bottom. Still, this figure should illustrate how different optimizers can get different results.

5.1 Runtime

To get an estimate of expected runtimes, I fited a *CameraOptimizer* using $4 - 14$ cameras. The results can be seen in figure 5.2.
A few remarks: 1.) I was expecting the fitness function to resemble a logistic growth function, which it does: rapidly growing initially and slowing down as adding more cameras yields diminishing returns. 2.) I was expecting the runtime to keep on growing evermore rapidly as the dimensionality of the problem increases, but the growth of the runtime also seems to slow down alongside the fitness funciton. I would say that this is due to the optimizer ending the optimization process as the fitness values plateau (which happens even if the dimensionality is increased). 3.) The shape of the two graphs seems to match, except that there is a lot more noise in the runtime - as seen in table 5.1. This can be explained by the random nature of the CMA-ES: the initial solution is generated randomly and so we can get 'lucky' and start off close to a good solution, which reduces runtime. 4.) The deviations in table 5.1 are over 6 different runs and the data in figure 5.2 is the average of those runs. 5.) The fiting was done in a $5 \times 7.5 \times 3$ simulated space. For a larger space, expect larger runtimes for a given number of cameras and expect that more cameras are needed to achieve the same fitness value. 6.) The runtime was measured using a Raspberry Pi 4. 7.) A single optimization task is single-threaded.

[1]There is a messy, not-user-friendly version where the training of symmetrical cameras work.

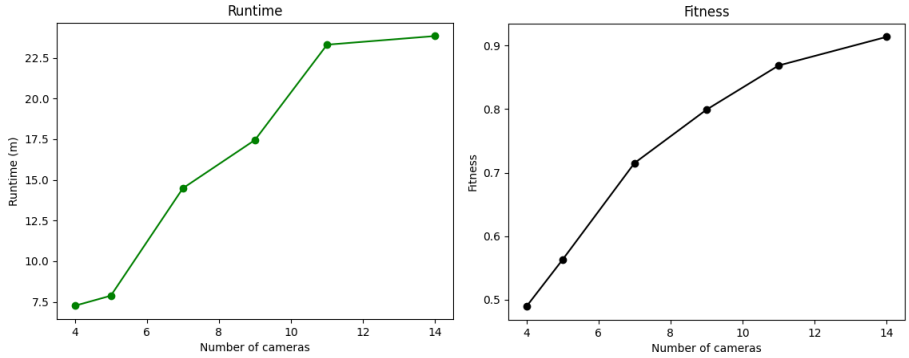

Figure 5.2: Runtime on the left and corresponding fitness values on the right as a function of the number of cameras.

One final word on some shortcomings. Reducing the dimensionality of the problem too much can result in the optimization process detecting a lack of improvement too early and terminating with terrible fitness values. This problem is most notable when using optimizers with **symmetry**. On the other hand, when using symmetrical optimizers with too many cameras, there is an increased chance that the result will feature cameras too close to one another. Such a solution is rejected by the fitness function and the *cma* library does consider this, however for some reason in the case of the symmetrical optimizers it can return with a setup it considers to be the best that is actually rejected. Since there is a level of randomness to the algorithm, the optimizers will eventually come up with a non-rejected solution, but there is a possibility that may take a while.

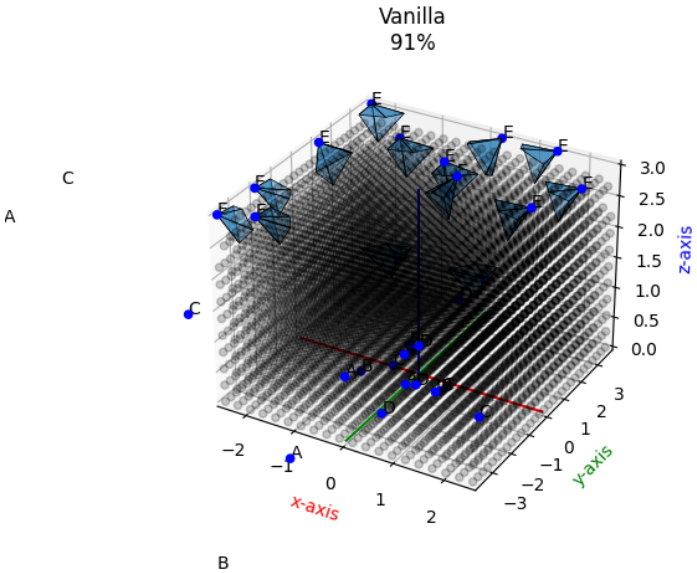

Figure 5.3: What I would consider a final best result for the problem this project was created for. With the dimensions being $5 \times 7.5 \times 3$ meters and the number of cameras being 14.