**Robot Vision Midterm Report**

Putzer Thomas (2024-81222, Computer Science and Engineering)

Ditha Anggraini (2022-15281, Civil and Environmental Engineering)

Chao Weihao (2024-81353, Mechanical Engineering)

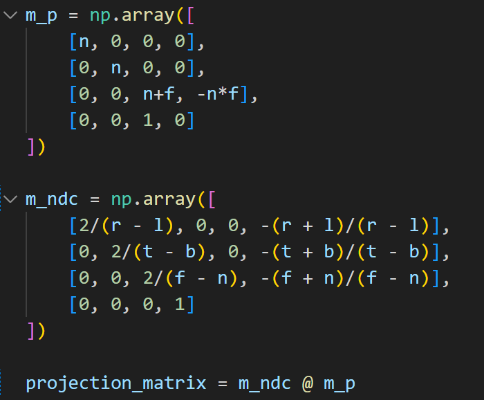
**Part 1: Detecting Keypoints**

<https://github.com/PuuTzzA/robot_vision_2024-2>

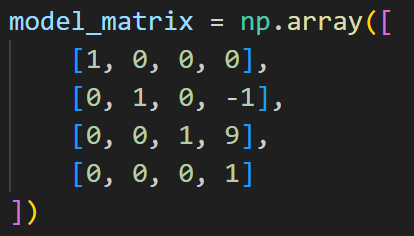
**Part 2: keypoints2pose**

<https://github.com/PuuTzzA/keypoint2pose>

We have started with the part that transforms the detected 2d keypoints into the 3d position of the car. The knowns are the detected 2d keypoints from Part 1, the projection Matrix that can be calculated with the focal length of the camera and its sensor size and the 3d model. The unknown is the model\_martix that describes the translation and scale of the 3d model in the scene. The 3d model is given as a .obj.



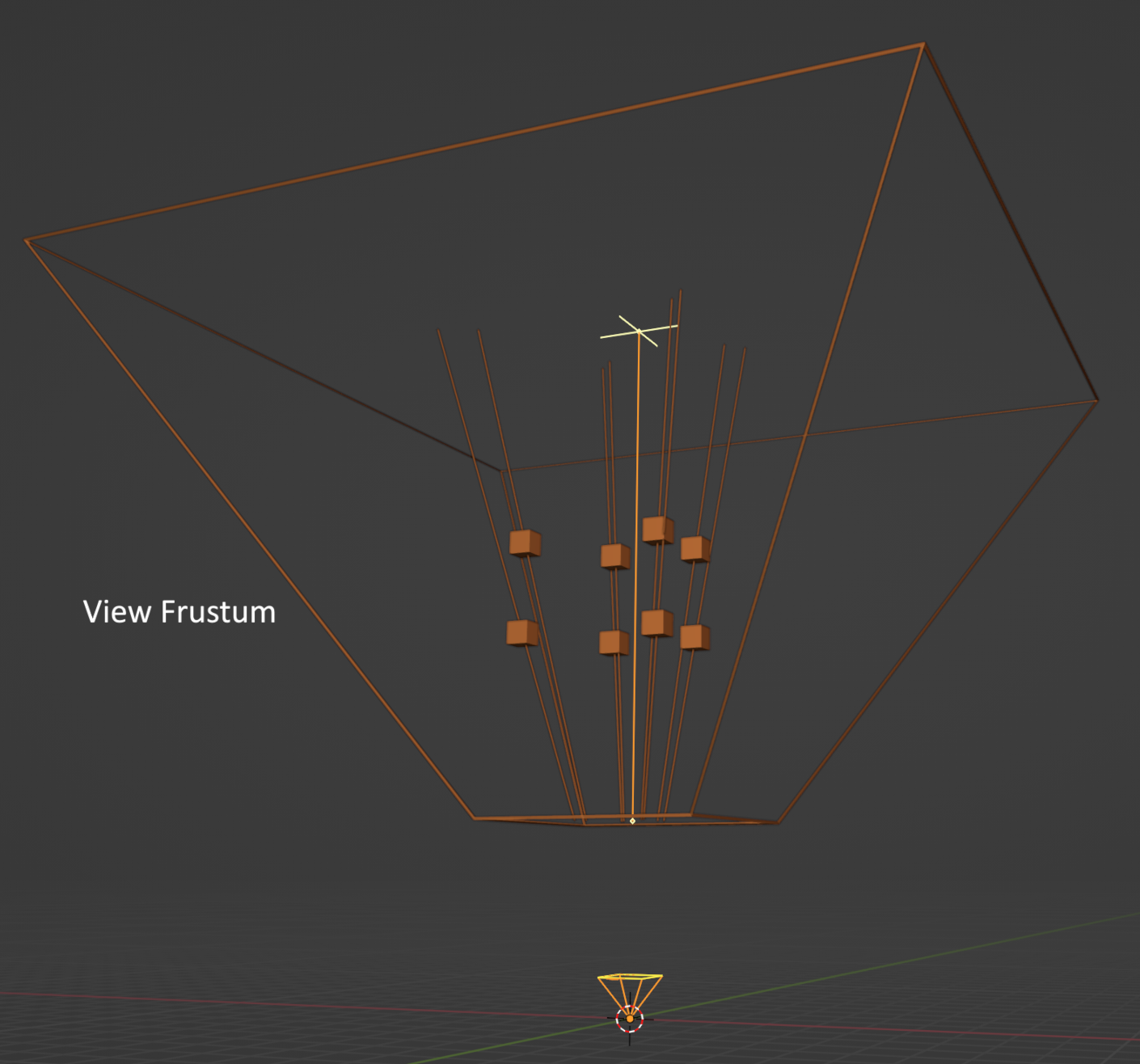
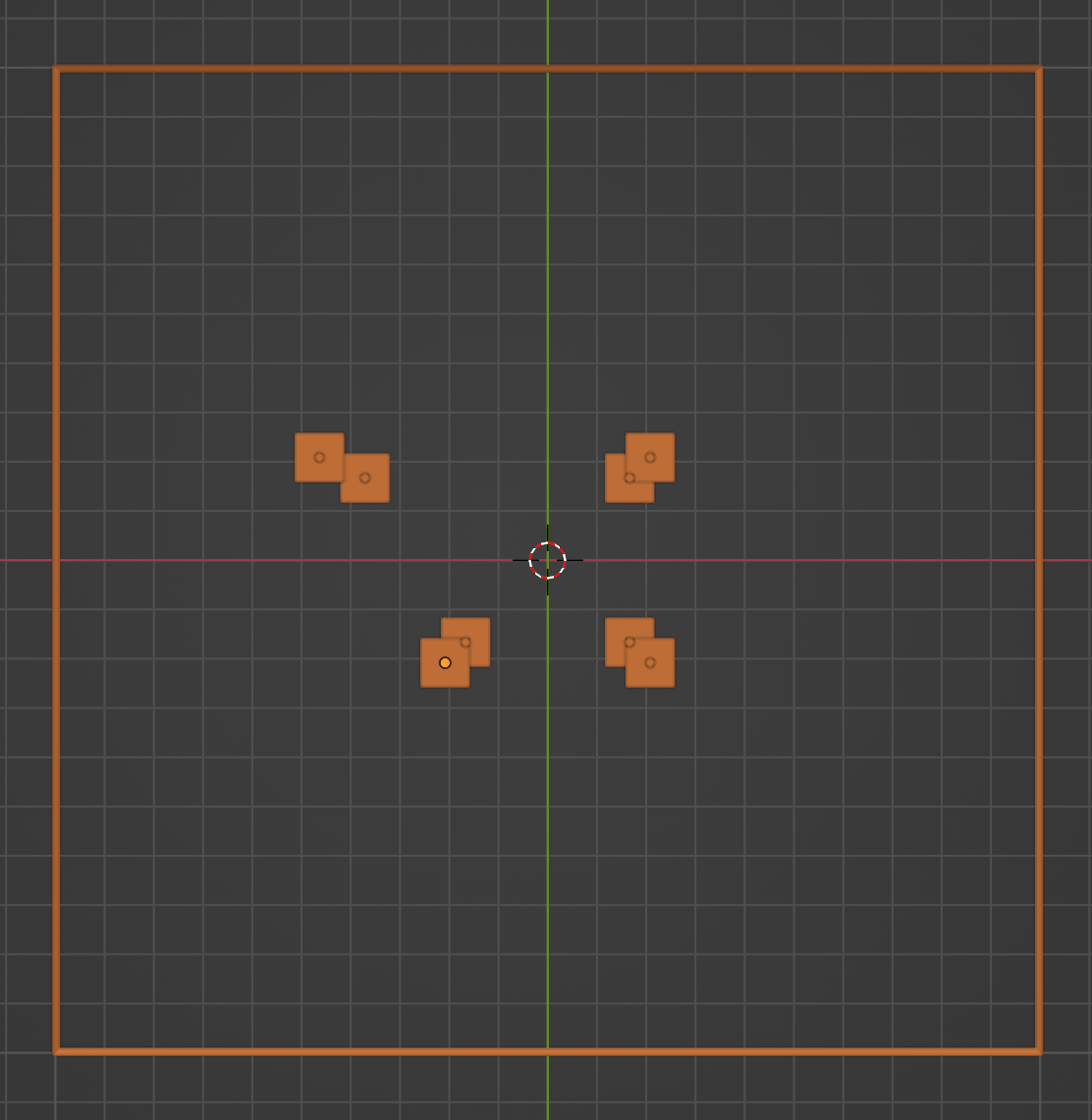
This is the projection Matrix: It is calculated from the camera parameters focal length and sensor size. It is calculated by multiplying the projection Matrix by the Matrix that transforms the points from the camera Frustum into the Normalized Device Coordinates.



This is an example Model Matrix that translates the model by -1 in the y direction and by 9 in the z direction.

The relation of the 2d points and the 3d points is as follows: The 2d points are the first two components of the vector x’ = M\_projection \* M\_model \* x after the perspective division of x’ by the 4th component of x’.

Until now we only implemented visualisation of the 3d to 2d perspective. This visualisation is done in Blender (<https://www.blender.org/>) a free and open source 3d program.  

  A screenshot of a computer generated cube

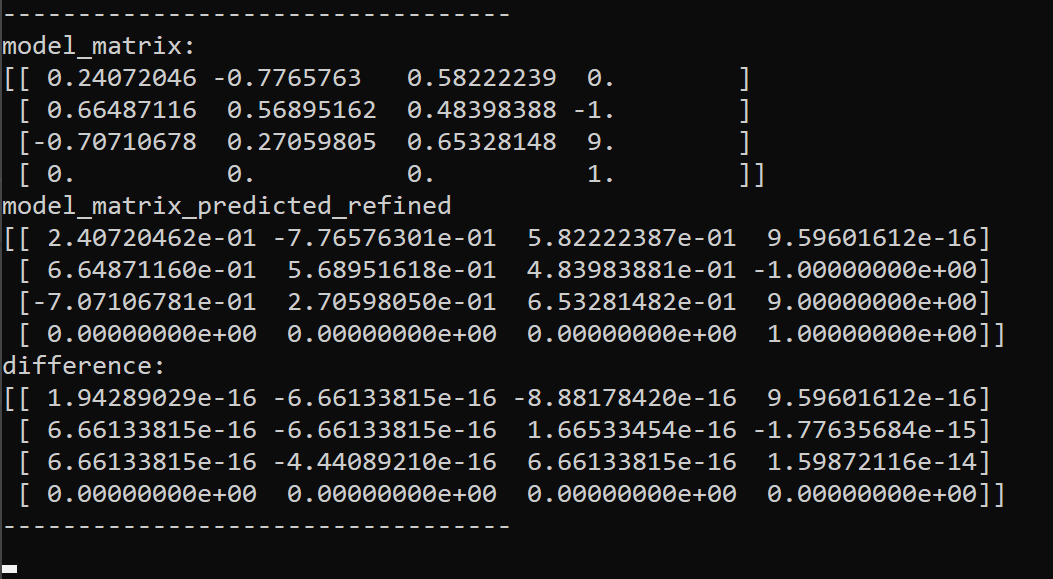
Description automatically generated

The 3d points inside the view frustum get transformed into the Normalised Device coordinates. But we only have the 2d positions that you see in the second image. Because we have this we cannot fully reconstruct the 4x4 model\_martix, since the scale could vary freely. But fortunately we know that the scale is one, so that is not a problem.

We have done literature review and found two promising ways to solve for the model\_matrix. The first one is with a Direct linear transformation. You can immagine our problem as  where everything except Mm is known. The matrices look like this: where c1...c4 are the camera constants and a...l are the unknowns. The reason why there are only two numbers in x2 is that we only know the 2d positions and nothing more. You can multiply this formula out and then you would get:

You can than transform this into a matrix vector multiplication of the form Ab = 0 with all the matrix A containing all the knowns and b containing all the unknowns. We can then solve this by doing a Singular Value Decomposition and taking the last column of V^T as our result vector (=the smallest eigenvector of A\*A^T). This does not give us the exact result right away, because the result scale invariant. Luckily we know that the scale is one. That means that the Rotation Part R of the Model Matrix, who looks like this , should be orthogonal. To achieve this we do another Singular Value Decomposition of only R and replace R with U \* V^T (we leave out ∑ since we do not scale). Now we only have to scale tx, ty and tz by the same amout we have scaled R right now and were done.

This gives us results like this:



To note is that the thest keypoints are very accurate (the only errors are floating point errors) so the result may be worse with worse keypoints.

The second way is choosing a good starting model\_matrix in the form of and parameters θx, θy, θz, tx, ty and tz and using an optimization method like Levenberg-Marquardt to find the optimal model\_matrix. This seems to be a better way, if the keypoints are really bad, but we still need to implement and test this.

We also consider using RANSAC to handle outliers and improve the estimate. Another way to handle it would be to give the keypoints different weights according to how confident the Deep-learing model is in them.