Image-Based Situation Awareness Audit 8.5.2018

Sakari Lampola

Previous Audit 28.2.2018

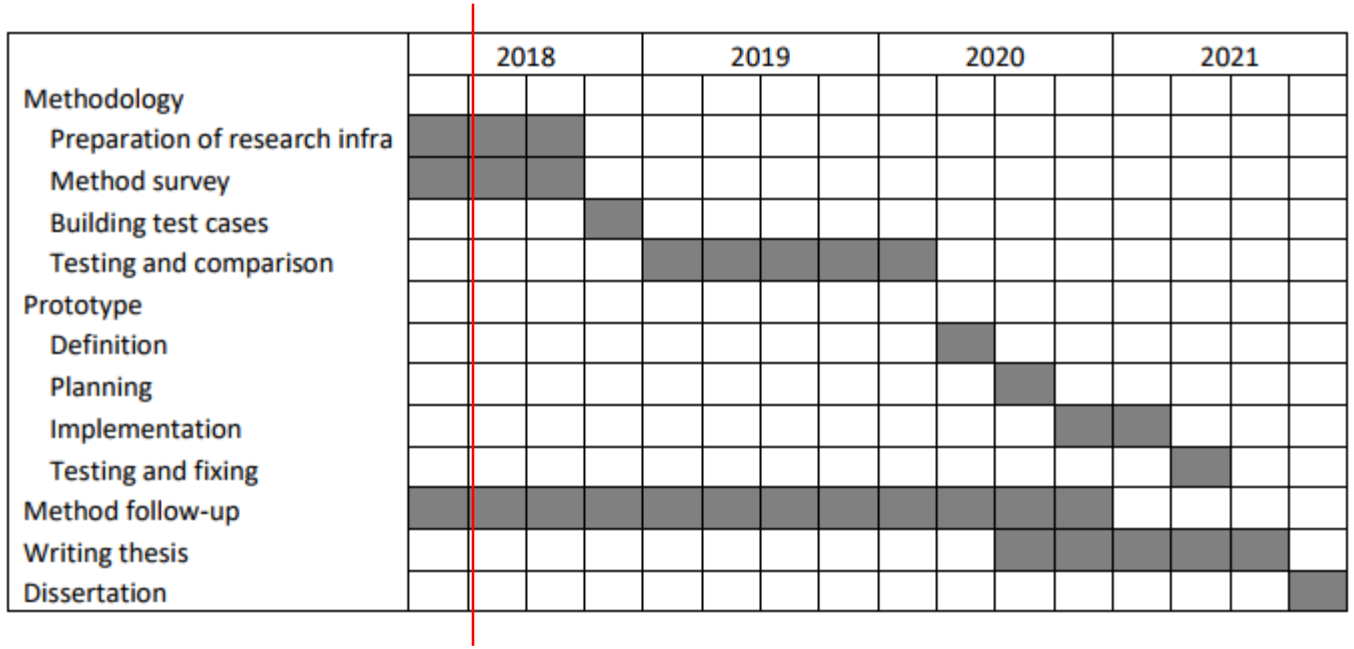
Next steps

- Kalman filter parameter adjustments (Q1)
- Dataset selection (Q1)
- Stereo vision (Q2)
- Camera yaw, pitch, roll estimation (Q2)
- Speech recognition (Q2)
- Semantic segmentation (Q2)
- ~~Experiments in the wild (Q2)~~
- Paper (Q3)
- Speech analysis (Q3)
- Speech generation (Q3)
- Use cases (Q4)

Other

- Body forecast
  - kinetic
  - based on class history
  - based on swarm history
- R matrix estimation
- Monograph or papers?

Project Plan



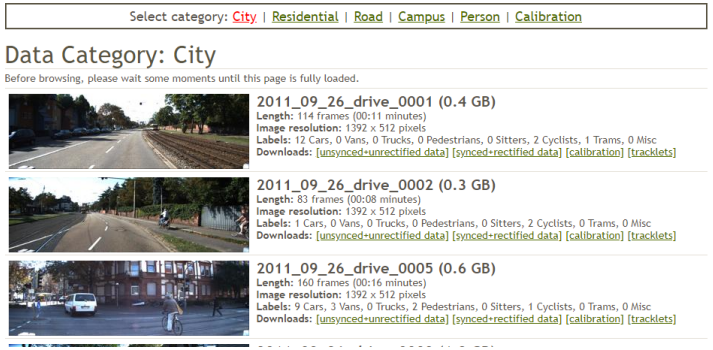
1. Methodology / Preparation of research infra
    a. Software platforms are constructed and tested
    b. Off-the-shelf models are acquired and tested
    c. Necessary skills on platforms are learned
2. Methodology / Method survey
    a. Current state-of-art methods are studied
    b. Methods are constructed and tested on the software platforms
3. Method follow-up
    a. Screening of conference papers related to the subject
    b. Possibly integrating new methods to the project

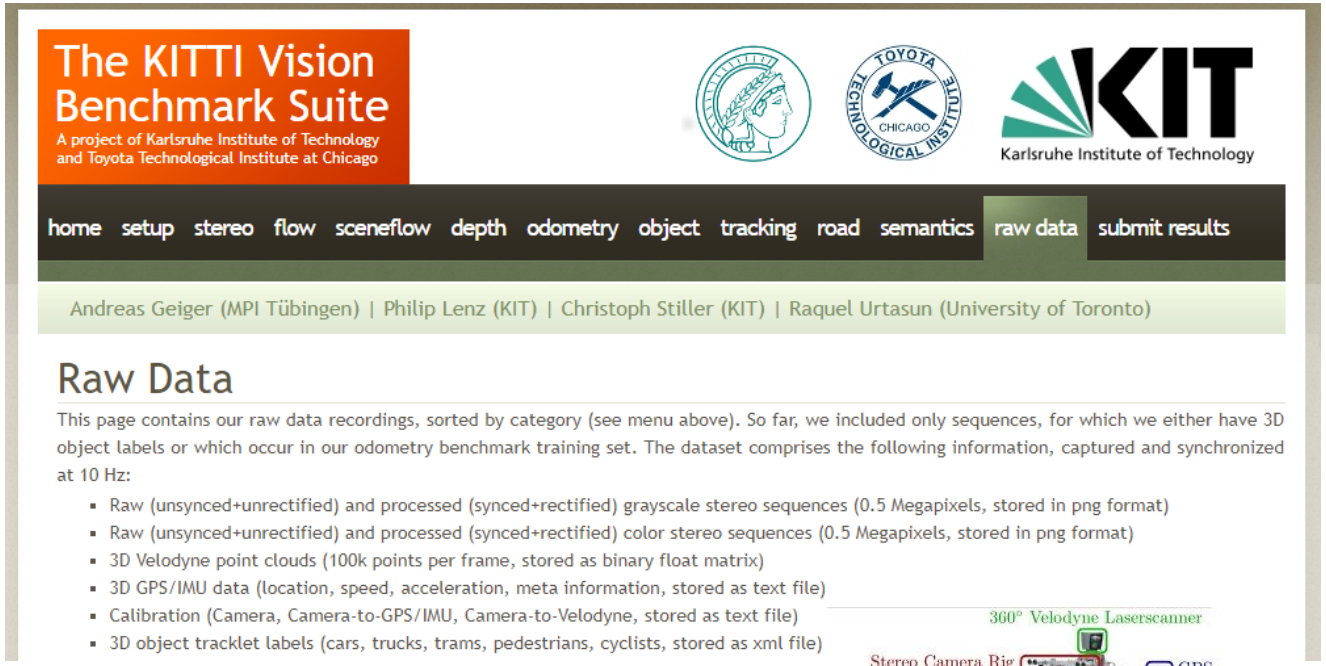
Work Done

Dataset Selection

Specification:

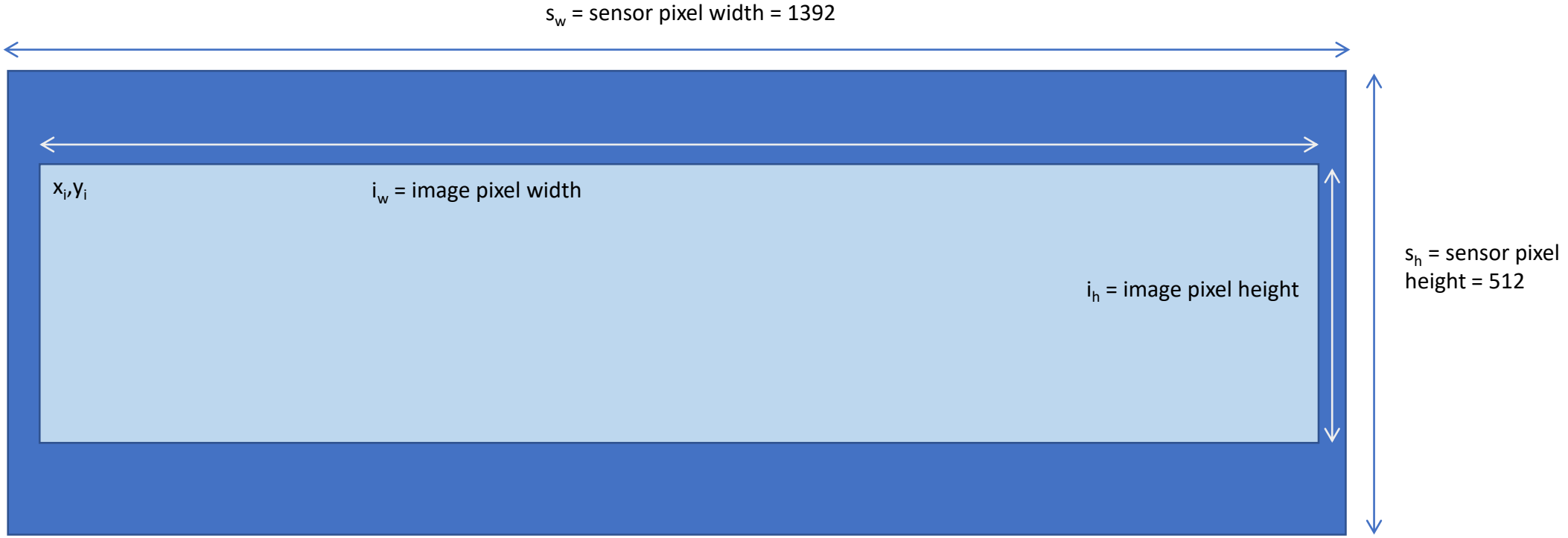
- Video
- Stereo
- Distance information
- Outdoor + indoor
- Odometry





Open question: Indoor? Self generated?

Image format / cropping



Assumptions:

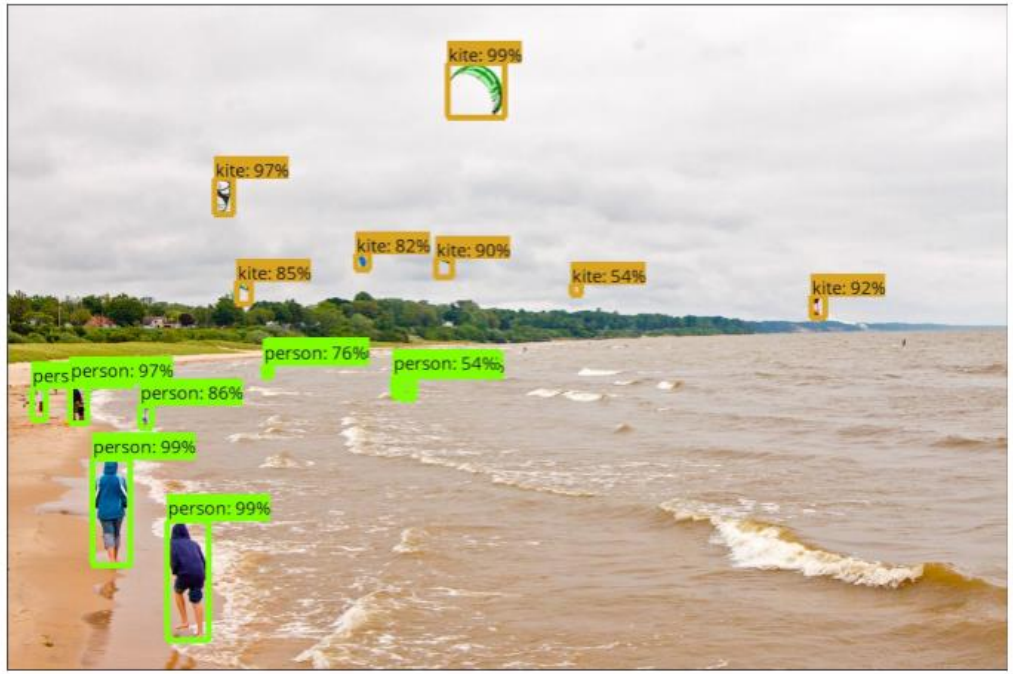
- image is symmetrically cropped (optical center fixed)
- rectification ignored

$$x_i = (s_w - i_w)/2$$

$$y_i = (s_h - i_h)/2$$

Tensorflow Object Detection API

Creating accurate machine learning models capable of localizing and identifying multiple objects in a single image remains a core challenge in computer vision. The TensorFlow Object Detection API is an open source framework built on top of TensorFlow that makes it easy to construct, train and deploy object detection models. At Google we've certainly found this codebase to be useful for our computer vision needs, and we hope that you will as well.



https://github.com/tensorflow/models/tree/master/research/object_detection

COCO-trained models {#coco-models}

Model name	Speed (ms)	COCO mAP[^1]	Outputs
ssd_mobilenet_v1_coco	30	21	Boxes
ssd_inception_v2_coco	42	24	Boxes
faster_rcnn_inception_v2_coco	58	28	Boxes
faster_rcnn_resnet50_coco	89	30	Boxes
faster_rcnn_resnet50_lowproposals_coco	64		Boxes
rfcn_resnet101_coco	92	30	Boxes
faster_rcnn_resnet101_coco	106	32	Boxes
faster_rcnn_resnet101_lowproposals_coco	82		Boxes
faster_rcnn_inception_resnet_v2_atrous_coco	620	37	Boxes
faster_rcnn_inception_resnet_v2_atrous_lowproposals_coco	241		Boxes
faster_rcnn_nas	1833	43	Boxes
faster_rcnn_nas_lowproposals_coco	540		Boxes
mask_rcnn_inception_resnet_v2_atrous_coco	771	36	Masks
mask_rcnn_inception_v2_coco	79	25	Masks
mask_rcnn_resnet101_atrous_coco	470	33	Masks
mask_rcnn_resnet50_atrous_coco	343	29	Masks

One of these will be the final model.

Kitti-trained models {#kitti-models}

Model name	Speed (ms)	Pascal mAP@0.5 (ms)	Outputs
faster_rcnn_resnet101_kitti	79	87	Boxes

Lottery prize!!!! Will be used to implement localization and velocity estimation

Open Images-trained models {#open-images-models}

Model name	Speed (ms)	Open Images mAP@0.5[^2]	Outputs
faster_rcnn_inception_resnet_v2_atrous_oid	727	37	Boxes
faster_rcnn_inception_resnet_v2_atrous_lowproposals_oid	347		Boxes

COCO
Common Objects in Context

info@cocodataset.org

Home | People | Dataset | Tasks | Evaluate

News

- 2017 Challenge Winners for Detection, Keypoint, & Stuff tasks have been announced! Please visit the Joint COCO and Places Recognition ICCV workshop page for details.
- This website is now hosted on Github, which provides page source and history.
- Keypoint analysis tools are now available, see keypoints evaluation, Section 4.

What is COCO?

COCO is a large-scale object detection, segmentation, and captioning dataset. COCO has several features:

- Object segmentation
- Recognition in context
- Superpixel stuff segmentation
- 330K images (>200K labeled)
- 1.5 million object instances
- 80 object categories
- 91 stuff categories
- 5 captions per image
- 250,000 people with keypoints

Collaborators

Tsung-Yi Lin Google Brain
Genevieve Patterson MSR
Matteo R. Ronchi Caltech
Yin Cui Cornell Tech
Michael Maire TTI-Chicago
Serge Belongie Cornell Tech
Lubomir Bourdev WaveOne, Inc.
Ross Girshick FAIR
James Hays Georgia Tech
Pietro Perona Caltech
Deva Ramanan CMU
Larry Zitnick FAIR
Piotr Dollár FAIR

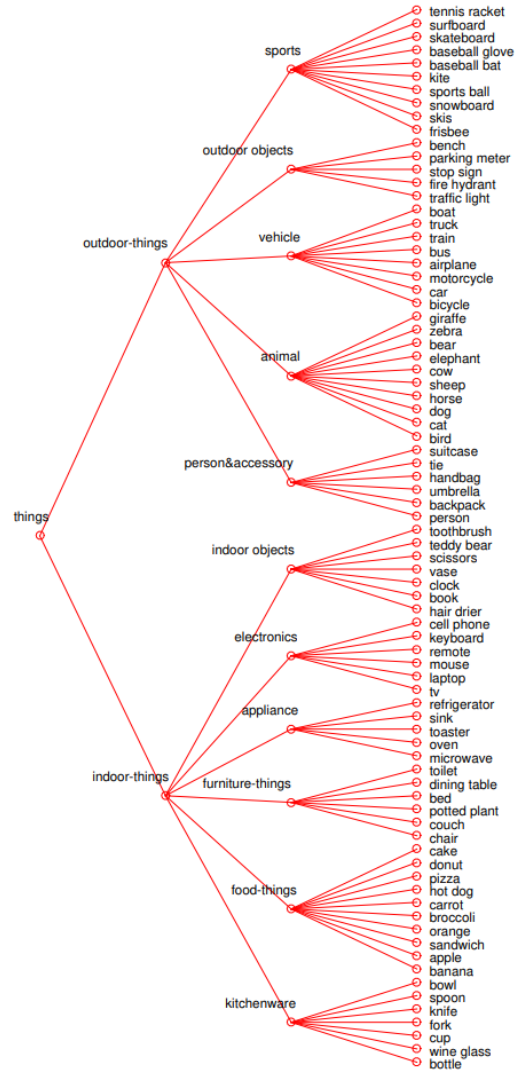
Sponsors

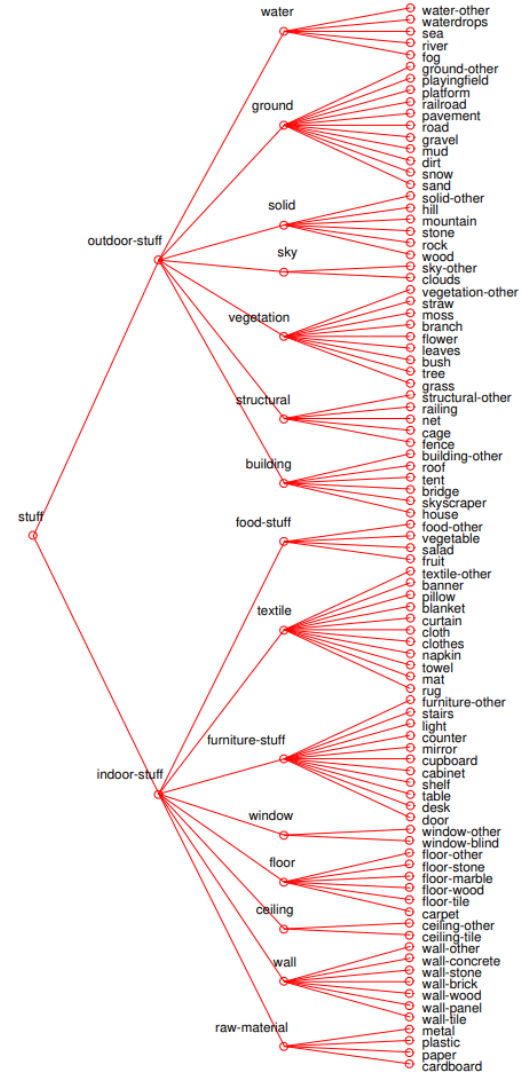
CVDF
Microsoft
facebook
Mighty AI

Research Paper

Download the paper that describes the Microsoft COCO dataset.

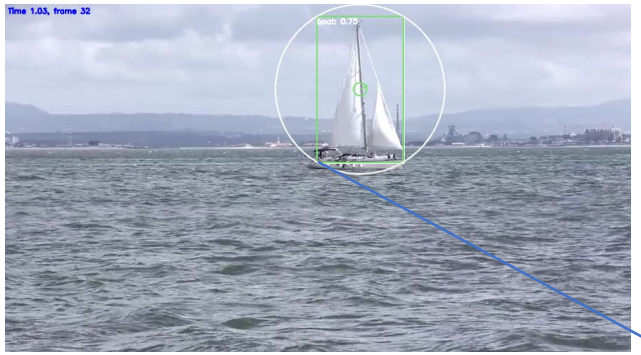
Download paper here



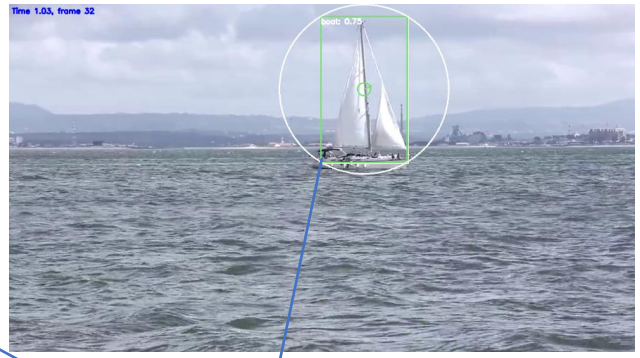


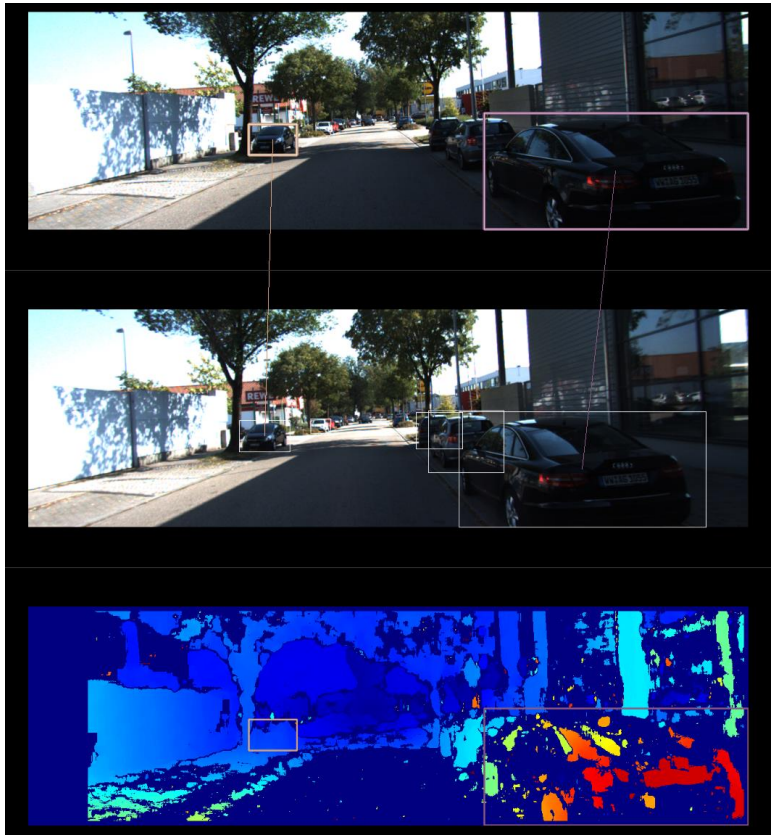
Stereo Vision

Left



Right





disparity d

STEREOLABS

PRODUCTS DEVELOPER SUPPORT BLOG GET STARTED Order Now

ZED™

2K Stereo Camera

The World's First
3D Camera for Depth Sensing
and Motion Tracking

ORDER FOR $449

Shipping Worldwide

Compatible OS

Windows 7, 8, 10

Linux

Third-party Support

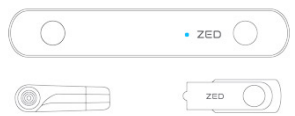
ROS unity UNREAL ENGINE OpenCV MATLAB

SDK System Requirements

- Dual-core 2,3GHz or faster processor
- 4 GB RAM or more
- Nvidia GPU with compute capability > 3.0

In The Box

- ZED Stereo camera
- Mini Tripod stand
- USB Drive with Drivers and SDK
- Documentation



Dimensions

Features

- High-Resolution and High Frame-rate 3D Video Capture
- Depth Perception indoors and outdoors at up to 20m
- 6-DoF Positional Tracking
- Spatial Mapping

Video

Video Mode	Frames per second	Output Resolution (side by side)
2.2K	15	4416x1242
1080p	30	3840x1080
720p	60	2560x720
WVGA	100	1344x376

Depth

Depth Resolution
Same as selected video resolution

Depth Format
32-bits

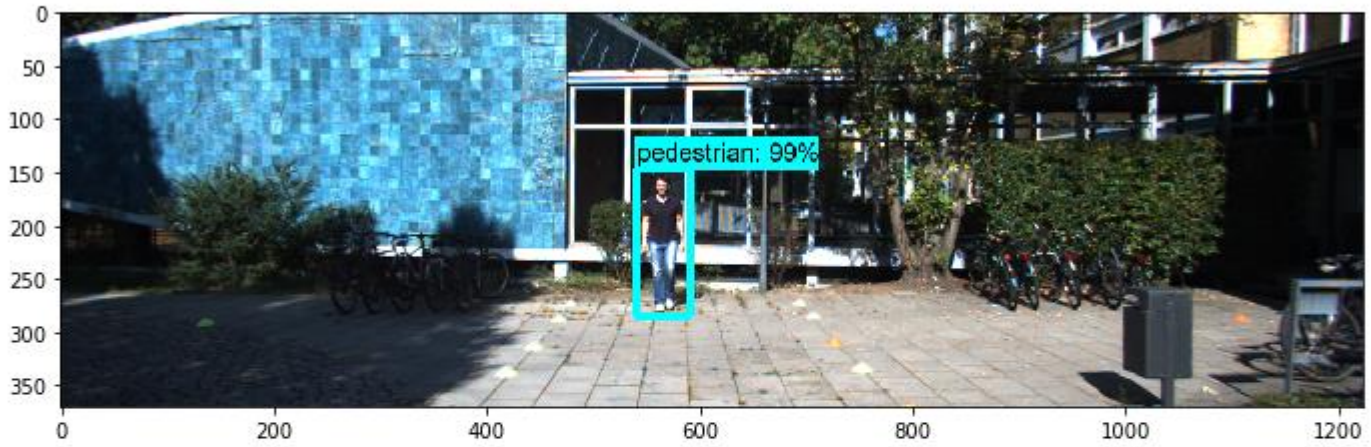
Depth Range
0.5 - 20 m (2.3 to 65 ft)

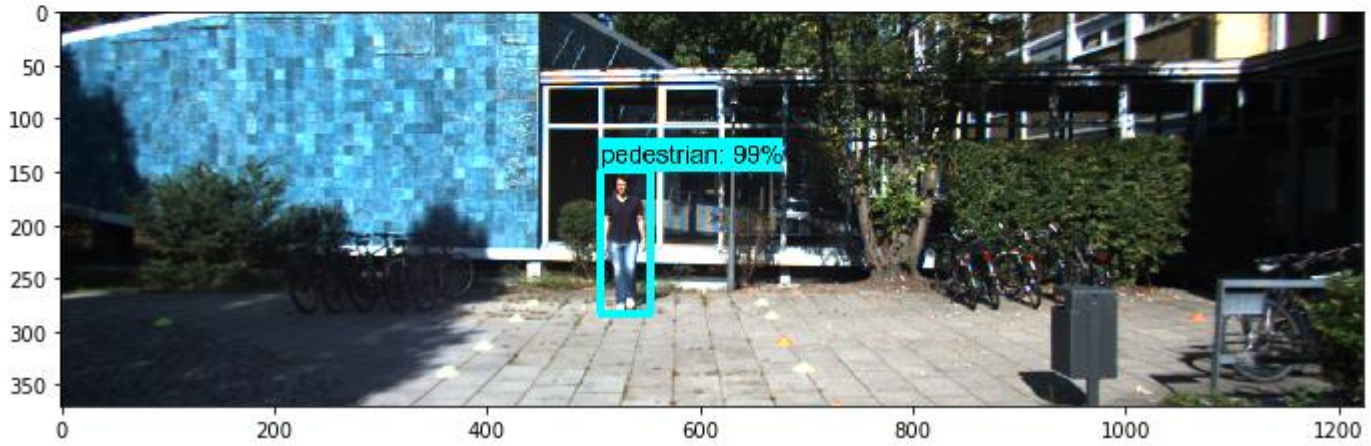
Stereo Baseline
120 mm (4.7")

Mapping left and right image patterns

```
class,confidence,x,y,width,height,hue0,hue1,hue2,saturation,value
-------------------------------------------------------------
2,1.00,566.00,215.50,52.00,137.00,0.315,0.437,0.247,71.996,124.843
-------------------------------------------------------------
2,1.00,530.50,216.00,49.00,134.00,0.291,0.468,0.242,72.229,121.822
```

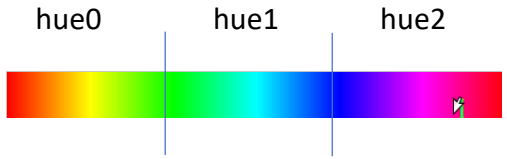
Pattern features to match

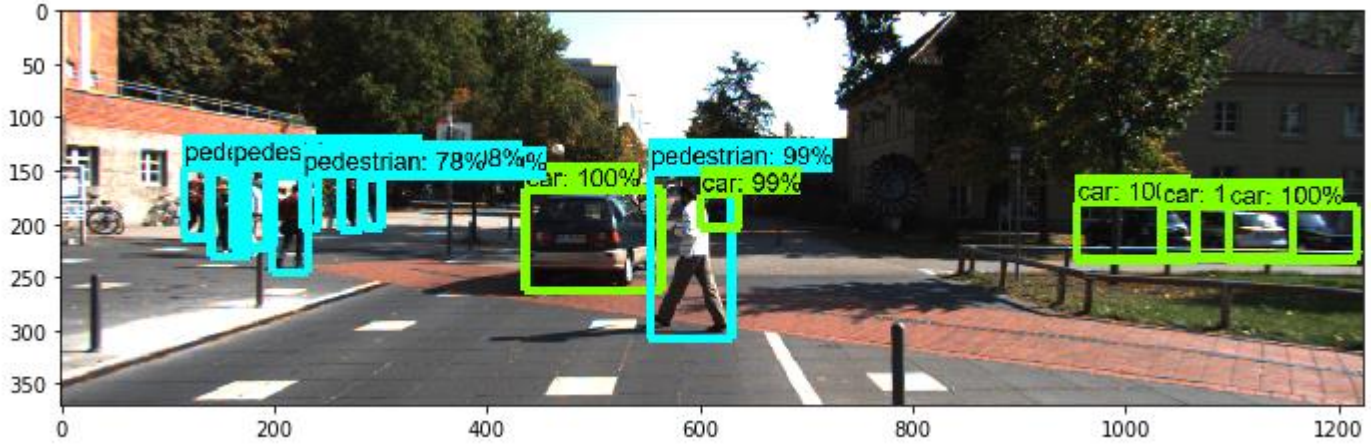


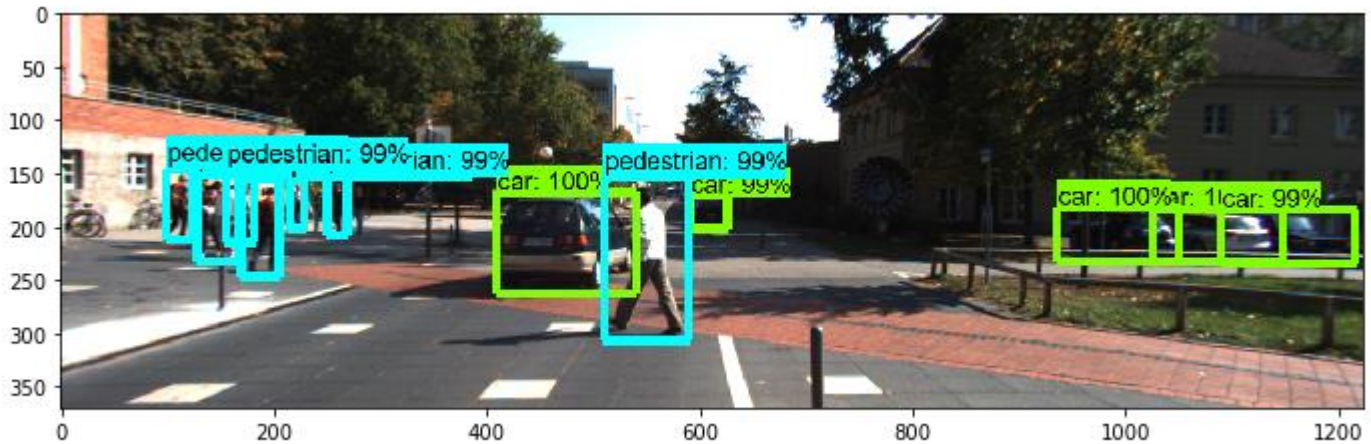


x,y = bounding box center location
saturation,value = mean values

3-bin hue histogram:








```
class,confidence,x,y,width,height,hue0,hue1,hue2,saturation,value
-------------------------------------------------------------
1,1.00,1010.50,206.50,113.00,53.00,0.354,0.438,0.208,74.793,48.389
1,1.00,1096.00,207.00,124.00,48.00,0.313,0.514,0.173,75.248,79.118
1,1.00,1157.50,208.50,121.00,49.00,0.274,0.551,0.175,79.360,81.186
1,1.00,500.50,215.00,129.00,94.00,0.367,0.389,0.244,75.836,70.045
2,1.00,592.00,229.50,78.00,155.00,0.550,0.254,0.196,59.824,104.288
1,1.00,619.50,187.50,37.00,31.00,0.286,0.323,0.391,58.958,58.623
2,1.00,159.00,190.50,36.00,79.00,0.434,0.220,0.346,77.767,116.533
2,1.00,293.50,180.50,17.00,45.00,0.354,0.299,0.346,78.097,70.915
2,1.00,127.50,180.00,25.00,64.00,0.493,0.174,0.333,97.595,144.548
2,1.00,216.00,200.00,34.00,90.00,0.422,0.256,0.322,82.717,112.057
2,1.00,184.50,184.00,33.00,72.00,0.413,0.229,0.358,68.688,159.507
2,1.00,177.50,185.50,33.00,75.00,0.395,0.244,0.361,70.105,148.906
2,0.98,272.50,179.50,19.00,51.00,0.397,0.261,0.342,80.279,69.196
2,0.79,235.00,180.50,16.00,47.00,0.483,0.209,0.309,94.992,127.202
-------------------------------------------------------------
1,1.00,1087.00,207.50,124.00,49.00,0.292,0.507,0.201,71.784,76.482
1,1.00,993.50,207.00,115.00,50.00,0.348,0.410,0.242,72.279,47.679
1,1.00,475.00,215.00,132.00,94.00,0.311,0.366,0.323,69.133,74.729
1,1.00,609.00,186.00,36.00,30.00,0.144,0.298,0.557,59.536,30.091
2,1.00,550.50,230.00,81.00,152.00,0.564,0.262,0.175,60.986,116.519
2,1.00,187.50,200.50,37.00,91.00,0.394,0.285,0.322,78.999,107.080
2,1.00,147.50,191.50,39.00,81.00,0.381,0.260,0.359,69.986,128.833
2,1.00,222.50,178.50,15.00,45.00,0.376,0.330,0.293,78.055,120.033
2,1.00,260.00,180.50,20.00,55.00,0.394,0.267,0.339,79.143,69.303
2,1.00,112.00,178.50,26.00,65.00,0.509,0.172,0.319,90.011,143.473
1,1.00,1153.50,209.00,127.00,50.00,0.209,0.593,0.198,77.370,78.260
2,0.99,169.00,180.50,28.00,67.00,0.432,0.317,0.251,75.630,159.326
```

Pattern matching based on feature difference is required!

Probabilistic model, answering the question: What is the probability two patterns represent the same object?

Feature vector F:

- confidence
- x
- y
- width
- heigth
- hue0
- hue1
- hue2
- saruration
- value

Note: Class is not included as it is **required** to be the same

Assumption:

$P(i \text{ and } j \text{ are same pattern}) \sim N(F_i - F_j \mid \mu_F , V_F)$

μ_F , V_F were estimated by matching 84 patterns in 28 KITTI stereo image pairs representing city, residential, campus and person categories, including both cars and pedestrians.

```
In [25]: mean = df.mean()

In [26]: mean
Out[26]: dConfidence    -0.002024
         dX             30.119048
         dY              0.142857
         dWidth          1.595238
         dHeight         1.071429
         dHue0           0.020762
         dHue1          -0.012524
         dHue2          -0.008333
         dSaturation     3.859619
         dValue         -1.029405
         dtype: float64
```

Note: Mean disparity (dX) is appr. 30 pixels

```
In [21]: covariance=df.cov()

In [22]: covariance
Out[22]:
```

	dConfidence	dX	dY	dWidth	dHeight	dHue0	dHue1	dHue2	dSaturation	dValue
dConfidence	0.002886	-0.069214	-0.022539	0.116641	-0.000818	-0.000207	0.000077	0.000131	0.046382	-0.003060
dX	-0.069214	411.545898	1.482788	53.121056	-1.707401	0.007306	-0.041094	0.034841	-31.935069	5.324796
dY	-0.022539	1.482788	6.991394	-20.158348	-2.624785	0.019775	-0.025219	0.005289	-2.481722	2.036318
dWidth	0.116641	53.121056	-20.158348	559.761905	75.860585	0.054818	-0.032215	-0.022245	4.598916	-36.784720
dHeight	-0.000818	-1.707401	-2.624785	75.860585	30.356282	0.030632	-0.030119	-0.000337	2.336329	-2.746706
dHue0	-0.000207	0.007306	0.019775	0.054818	0.030632	0.001820	-0.001190	-0.000622	-0.019008	0.125514
dHue1	0.000077	-0.041094	-0.025219	-0.032215	-0.030119	-0.001190	0.001925	-0.000741	0.040519	-0.041532
dHue2	0.000131	0.034841	0.005289	-0.022245	-0.000337	-0.000622	-0.000741	0.001361	-0.021563	-0.083944
dSaturation	0.046382	-31.935069	-2.481722	4.598916	2.336329	-0.019008	0.040519	-0.021563	24.449793	2.102903
dValue	-0.003060	5.324796	2.036318	-36.784720	-2.746706	0.125514	-0.041532	-0.083944	2.102903	78.502261

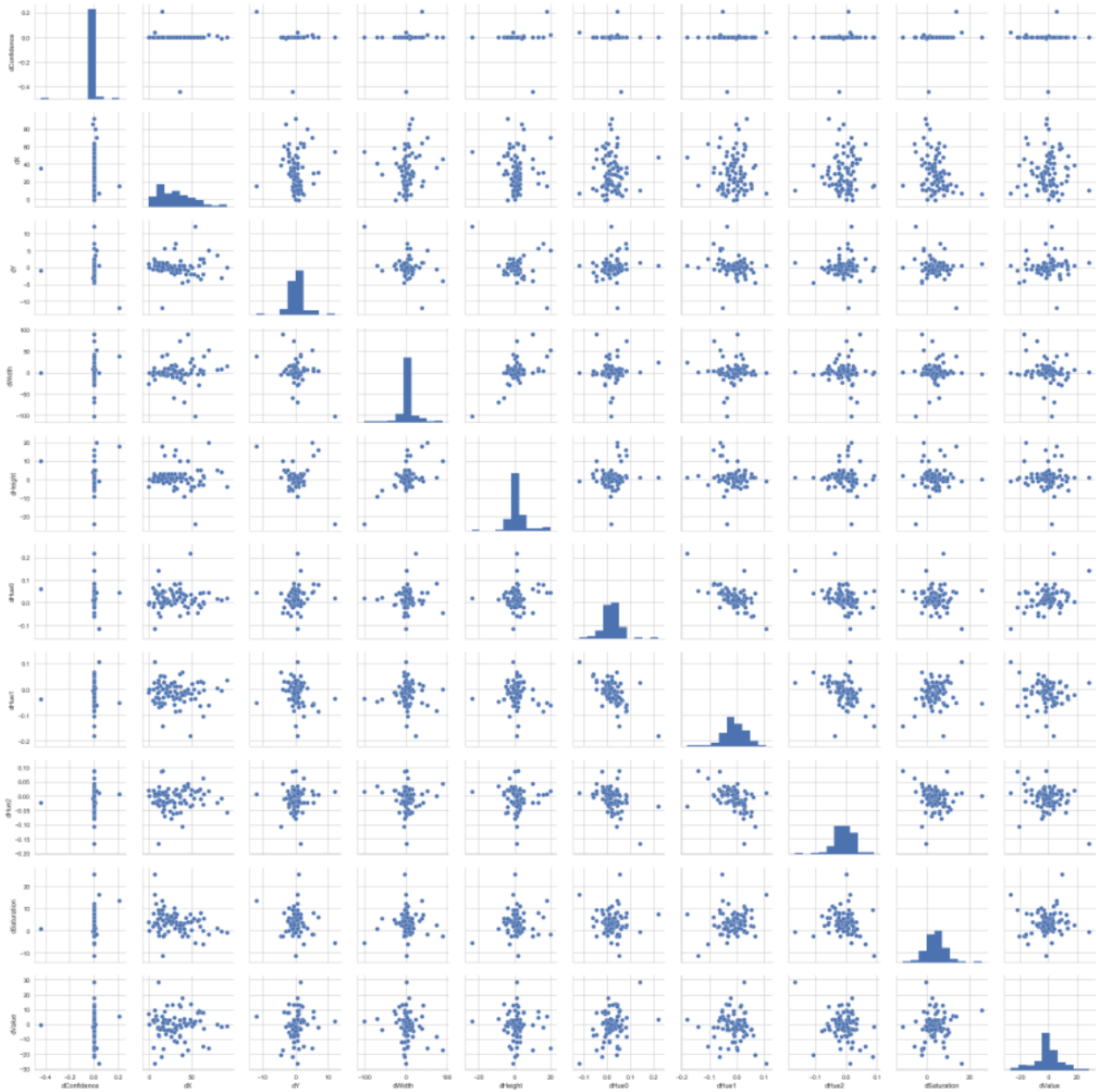
In [24]:
```
df.describe()
```

Out[24]:

	dConfidence	dX	dY	dWidth	dHeight	dHue0	dHue1	dHue2	dSaturation	dValue
count	84.000000	84.000000	84.000000	84.000000	84.000000	84.000000	84.000000	84.000000	84.000000	84.000000
mean	-0.002024	30.119048	0.142857	1.595238	1.071429	0.020762	-0.012524	-0.008333	3.859619	-1.029405
std	0.053724	20.286594	2.644124	23.659288	5.509654	0.042662	0.043879	0.036886	4.944673	8.860150
min	-0.440000	-1.000000	-12.000000	-102.000000	-24.000000	-0.117000	-0.181000	-0.166000	-11.327000	-26.690000
25%	0.000000	14.875000	-0.500000	-3.250000	-1.000000	-0.001250	-0.035250	-0.024500	1.041750	-3.944250
50%	0.000000	27.500000	0.000000	0.000000	1.000000	0.018500	-0.010000	-0.002000	3.741000	-1.281000
75%	0.000000	41.125000	0.625000	7.000000	2.250000	0.043000	0.014250	0.014250	5.875000	3.061000
max	0.210000	91.500000	12.000000	89.000000	20.000000	0.219000	0.106000	0.089000	25.595000	28.532000

In [20]:
```
sns.pairplot(df)
```

Out[20]:
```
<seaborn.axisgrid.PairGrid at 0x29b57b73588>
```

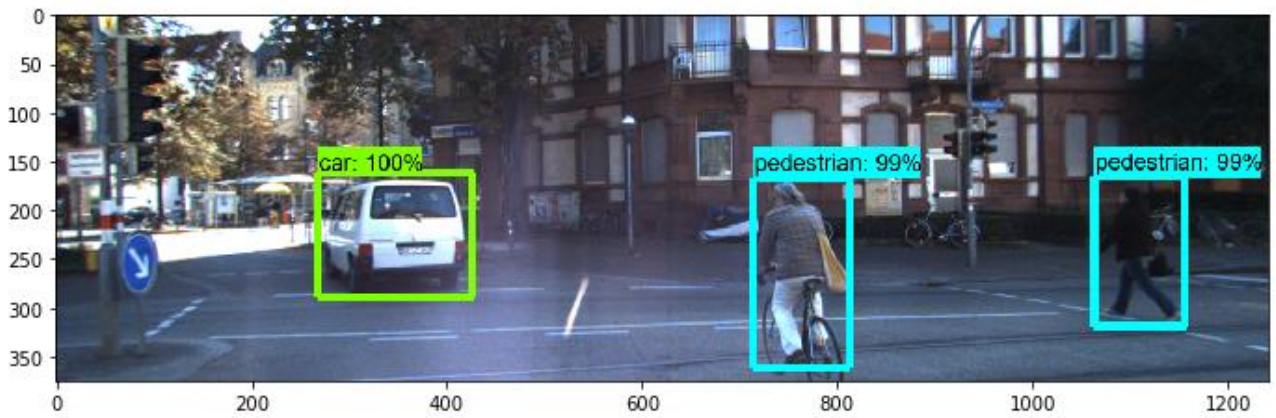


Pattern matching is done using Hungarian algorithm with the distance metrics:

d_{ij} = -log(P(i and j are same pattern)) = -log(N($F_i - F_j \mid \mu_F , V_F$))

If the probability that the patterns are same is near 1, the distance will be near zero. As the probability decreases, the distance increases. The log is required to compare small numbers without numerical issues.

Simple example




```
class,confidence,x,y,width,height,hue0,hue1,hue2,saturation,value
-------------------------------------------------------------
2,1.00,1155.50,241.00,89.00,150.00,0.074,0.669,0.256,85.880,58.390
1,1.00,376.50,226.00,163.00,126.00,0.114,0.626,0.259,70.091,137.364
-------------------------------------------------------------
1,1.00,347.50,225.50,159.00,127.00,0.104,0.614,0.282,66.123,148.577
2,1.00,1110.00,242.00,94.00,152.00,0.094,0.682,0.224,82.096,58.026
2,1.00,763.50,263.50,99.00,193.00,0.078,0.631,0.291,69.520,119.373
```

In [162]:
```
np.set_printoptions(precision=0)
print(distance_matrix)
```

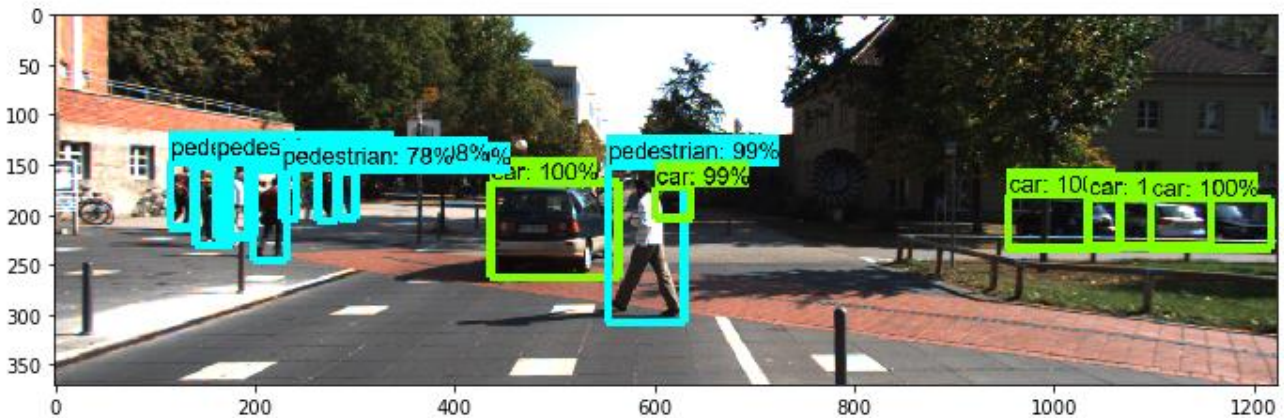
```
[[ 999.    6.  335.]
 [   6.  999.  571.]]
```

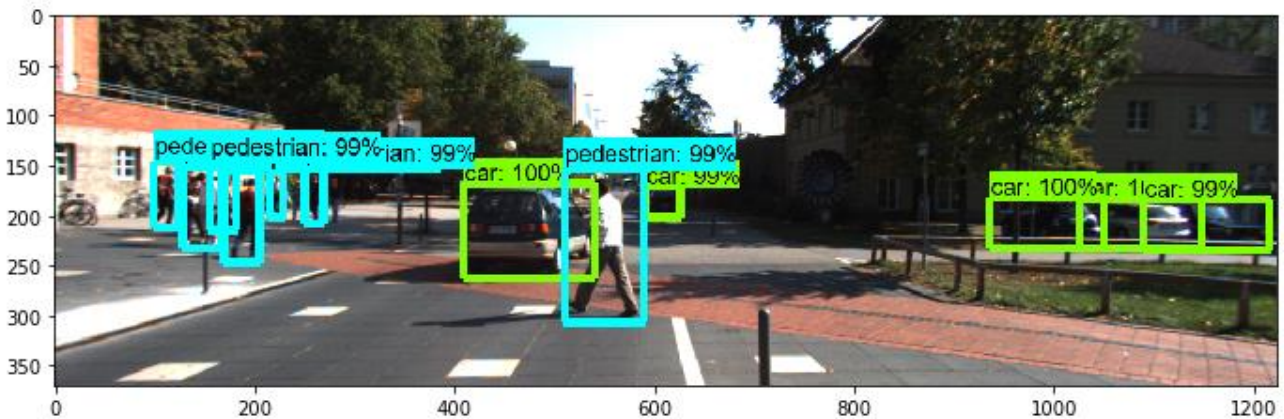
In [164]:
```
row_ind, col_ind = linear_sum_assignment(distance_matrix)
print(row_ind)
print(col_ind)
```

```
[0 1]
[1 0]
```


car: 99%
pedestrian:
car: 100%
pedestrian: 99%
pedestrian: 99%
0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200

Complicated example






```
class,confidence,x,y,width,height,hue0,hue1,hue2,saturation,value
-------------------------------------------------------------
1,1.00,1010.50,206.50,113.00,53.00,0.354,0.438,0.208,74.793,48.389
1,1.00,1096.00,207.00,124.00,48.00,0.313,0.514,0.173,75.248,79.118
1,1.00,1157.50,208.50,121.00,49.00,0.274,0.551,0.175,79.360,81.186
1,1.00,500.50,215.00,129.00,94.00,0.367,0.389,0.244,75.836,70.045
2,1.00,592.00,229.50,78.00,155.00,0.550,0.254,0.196,59.824,104.288
1,1.00,619.50,187.50,37.00,31.00,0.286,0.323,0.391,58.958,58.623
2,1.00,159.00,190.50,36.00,79.00,0.434,0.220,0.346,77.767,116.533
2,1.00,293.50,180.50,17.00,45.00,0.354,0.299,0.346,78.097,70.915
2,1.00,127.50,180.00,25.00,64.00,0.493,0.174,0.333,97.595,144.548
2,1.00,216.00,200.00,34.00,90.00,0.422,0.256,0.322,82.717,112.057
2,1.00,184.50,184.00,33.00,72.00,0.413,0.229,0.358,68.688,159.507
2,1.00,177.50,185.50,33.00,75.00,0.395,0.244,0.361,70.105,148.906
2,0.98,272.50,179.50,19.00,51.00,0.397,0.261,0.342,80.279,69.196
2,0.79,235.00,180.50,16.00,47.00,0.483,0.209,0.309,94.992,127.202
-------------------------------------------------------------
1,1.00,1087.00,207.50,124.00,49.00,0.292,0.507,0.201,71.784,76.482
1,1.00,993.50,207.00,115.00,50.00,0.348,0.410,0.242,72.279,47.679
1,1.00,475.00,215.00,132.00,94.00,0.311,0.366,0.323,69.133,74.729
1,1.00,609.00,186.00,36.00,30.00,0.144,0.298,0.557,59.536,30.091
2,1.00,550.50,230.00,81.00,152.00,0.564,0.262,0.175,60.986,116.519
2,1.00,187.50,200.50,37.00,91.00,0.394,0.285,0.322,78.999,107.080
2,1.00,147.50,191.50,39.00,81.00,0.381,0.260,0.359,69.986,128.833
2,1.00,222.50,178.50,15.00,45.00,0.376,0.330,0.293,78.055,120.033
2,1.00,260.00,180.50,20.00,55.00,0.394,0.267,0.339,79.143,69.303
2,1.00,112.00,178.50,26.00,65.00,0.509,0.172,0.319,90.011,143.473
1,1.00,1153.50,209.00,127.00,50.00,0.209,0.593,0.198,77.370,78.260
2,0.99,169.00,180.50,28.00,67.00,0.432,0.317,0.251,75.630,159.326
```

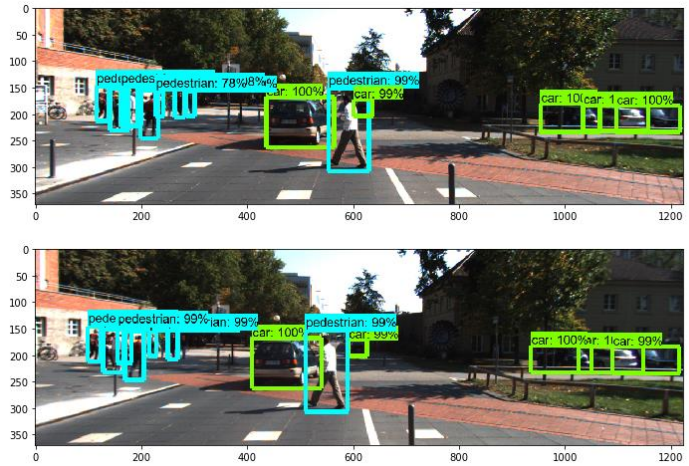
```
class,confidence,x,y,width,height,hue0,hue1,hue2,saturation,value
-------------------------------------------------------------
1,1.00,1010.50,206.50,113.00,53.00,0.354,0.438,0.208,74.793,48.389
1,1.00,1096.00,207.00,124.00,48.00,0.313,0.514,0.173,75.248,79.118
1,1.00,1157.50,208.50,121.00,49.00,0.274,0.551,0.175,79.360,81.186
1,1.00,500.50,215.00,129.00,94.00,0.367,0.389,0.244,75.836,70.045
2,1.00,592.00,229.50,78.00,155.00,0.550,0.254,0.196,59.824,104.288
1,1.00,619.50,187.50,37.00,31.00,0.286,0.323,0.391,58.958,58.623
2,1.00,159.00,190.50,36.00,79.00,0.434,0.220,0.346,77.767,116.533
2,1.00,293.50,180.50,17.00,45.00,0.354,0.299,0.346,78.097,70.915
2,1.00,127.50,180.00,25.00,64.00,0.493,0.174,0.333,97.595,144.548
2,1.00,216.00,200.00,34.00,90.00,0.422,0.256,0.322,82.717,112.057
2,1.00,184.50,184.00,33.00,72.00,0.413,0.229,0.358,68.688,159.507
2,1.00,177.50,185.50,33.00,75.00,0.395,0.244,0.361,70.105,148.906
2,0.98,272.50,179.50,19.00,51.00,0.397,0.261,0.342,80.279,69.196
2,0.79,235.00,180.50,16.00,47.00,0.483,0.209,0.309,94.992,127.202
-------------------------------------------------------------
1,1.00,1087.00,207.50,124.00,49.00,0.292,0.507,0.201,71.784,76.482
1,1.00,993.50,207.00,115.00,50.00,0.348,0.410,0.242,72.279,47.679
1,1.00,475.00,215.00,132.00,94.00,0.311,0.366,0.323,69.133,74.729
1,1.00,609.00,186.00,36.00,30.00,0.144,0.298,0.557,59.536,30.091
2,1.00,550.50,230.00,81.00,152.00,0.564,0.262,0.175,60.986,116.519
2,1.00,187.50,200.50,37.00,91.00,0.394,0.285,0.322,78.999,107.080
2,1.00,147.50,191.50,39.00,81.00,0.381,0.260,0.359,69.986,128.833
2,1.00,222.50,178.50,15.00,45.00,0.376,0.330,0.293,78.055,120.033
2,1.00,260.00,180.50,20.00,55.00,0.394,0.267,0.339,79.143,69.303
2,1.00,112.00,178.50,26.00,65.00,0.509,0.172,0.319,90.011,143.473
1,1.00,1153.50,209.00,127.00,50.00,0.209,0.593,0.198,77.370,78.260
2,0.99,169.00,180.50,28.00,67.00,0.432,0.317,0.251,75.630,159.326
```

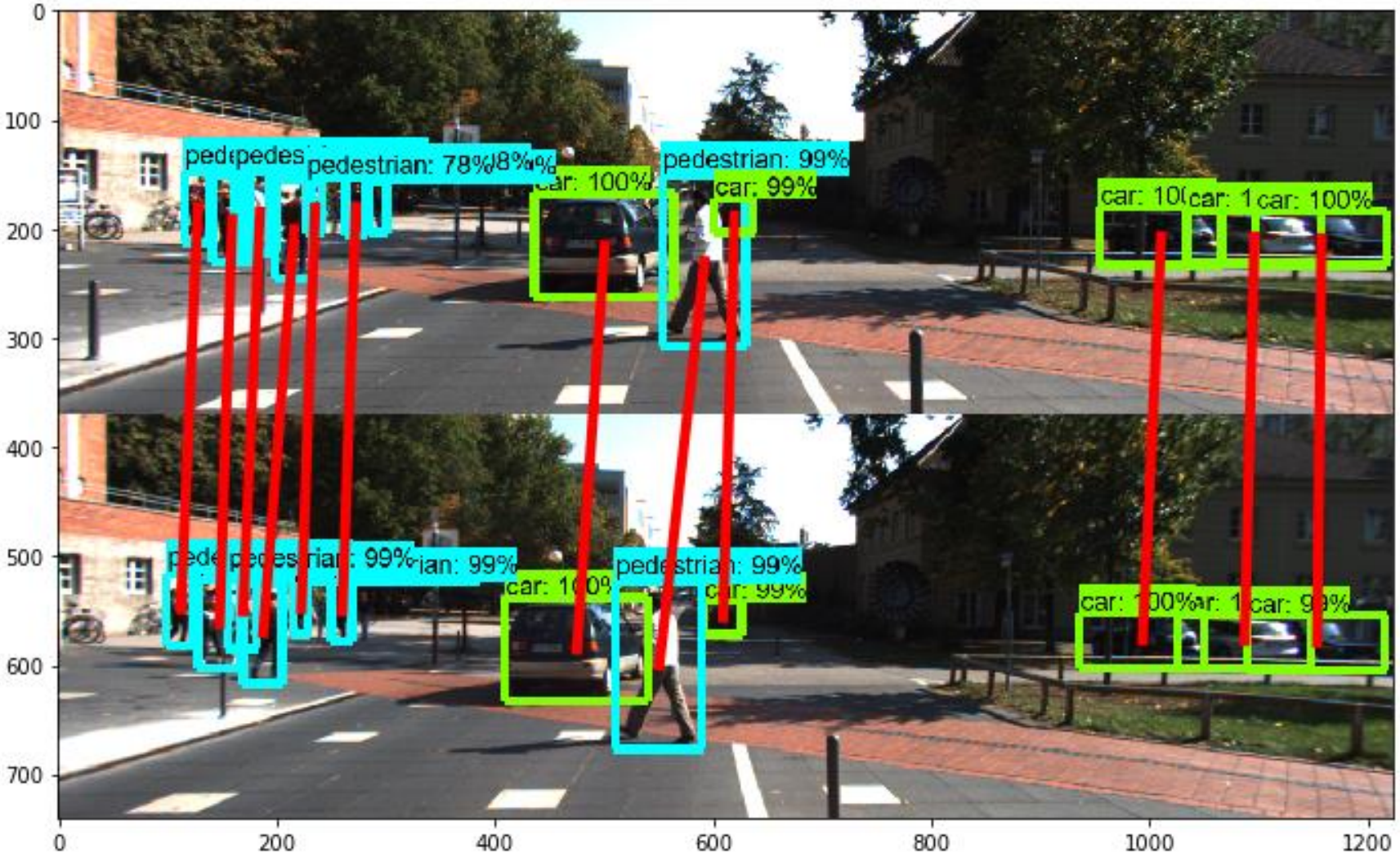
In [120]:
```
np.set_printoptions(precision=0)
print(distance_matrix)
```

```
[[  27.    6.  424.  315.  583.  999.  999.  999.  999.  999.   64.  999.]
 [   6.   24.  543.  427.  695.  999.  999.  999.  999.  999.   21.  999.]
 [   9.   46.  653.  532.  999.  999.  999.  999.  999.  999.    7.  999.]
 [ 566.  438.    7.  235.  154.  153.  257.  295.  232.  387.  693.  328.]
 [ 718.  616.  164.  600.    7.  372.  532.  690.  592.  999.  999.  651.]
 [ 453.  331.  191.   19.  542.  313.  345.  208.  155.  367.  588.  328.]
 [ 999.  999.  260.  406.  546.   29.    8.   54.   74.   35.  999.   31.]
 [ 999.  999.  246.  182.  631.  107.   93.   23.    9.   79.  999.   82.]
 [ 999.  999.  385.  449.  740.   96.   45.   40.   85.    6.  999.   25.]
 [ 999.  999.  157.  372.  372.    5.   22.   92.   92.   86.  999.   68.]
 [ 999.  999.  336.  443.  643.   79.   21.   53.  114.   35.  999.   14.]
 [ 999.  999.  314.  435.  613.   62.   14.   53.  106.   36.  999.   16.]
 [ 999.  999.  260.  206.  633.  102.   88.   25.    6.   68.  999.   79.]
 [ 999.  999.  327.  302.  717.  122.   86.   23.   49.   35.  999.   51.]]
```

In [122]:
```
row_ind, col_ind = linear_sum_assignment(distance_matrix)
print(row_ind)
print(col_ind)
```

```
[ 0  1  2  3  4  5  6  8  9 10 12 13]
[ 1  0 10  2  4  3  6  9  5 11  8  7]
```

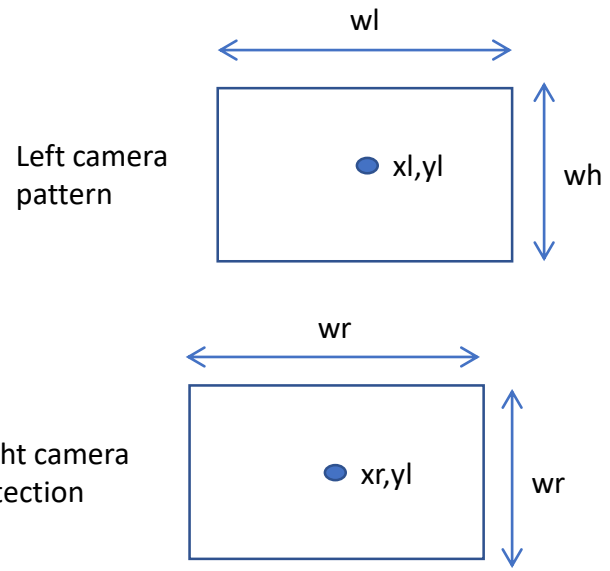



0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200
pedestrian: 78%
pedestrian: 99%
car: 99%
car: 100%
pedestrian: 99%
pedestrian: 99%
car: 99%
car: 100%

Too complicated!!! Curse of dimensionality makes probabilities (even with log) too small.

Simpler solution:

```
if (patterns_left[row].class_id == detections_right[col].class_id):
    c = abs(wl-wr)+abs(hl-hr)+abs(yl-yr) + STEREO_MATCHING_HORIZONTAL*abs(xl-xr)
else:
    c = 999999.0
if xl - xr < 0:
    c = 999999.0
cost[row][col] = c
```

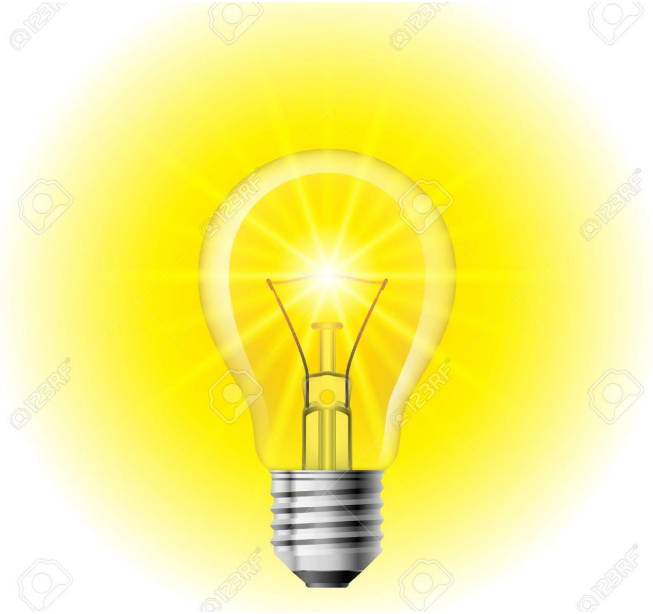


STEREO_MATCHING_HORIZONTAL = 0.4
Hungarian algorithm for matching.

Note: Patterns are not generated for the right camera. Patterns life spans in two cameras lead to complications.

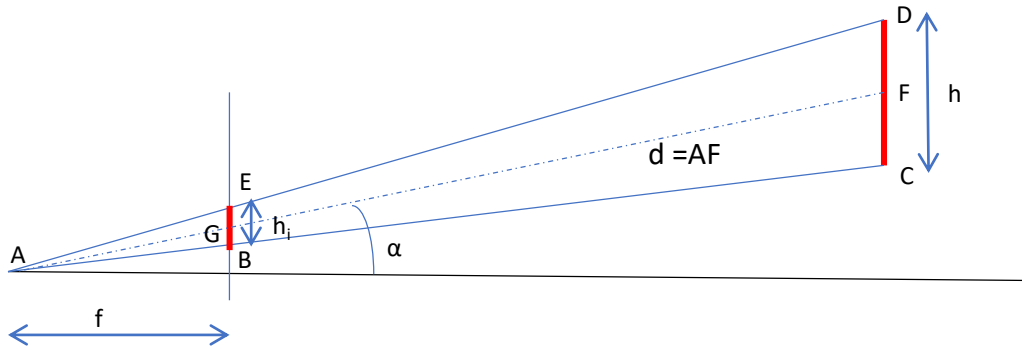
Not perfect, but good enough for the moment.

Research topic for 2019



After implementing stereo vision, we have two distance estimates:

1. Distance based on stereo vision. Accurate in short distances (for Kitti, 20 meters)
2. Distance based on size. Can be used in long distances where stereo vision is inaccurate.



$$d_{size} = \frac{f * r}{\cos(\alpha) * \cos(\beta) * r_i * s_h / p_h}$$

s_h= *sensor height* (m)
p_h= *image height (pixels)*
r_i = *pattern radius (pixels)*
r = body radius (m), mean from class specific distribution
f = focal length (m)
α = *altitude (rad)*
β = *azimuth (rad)*

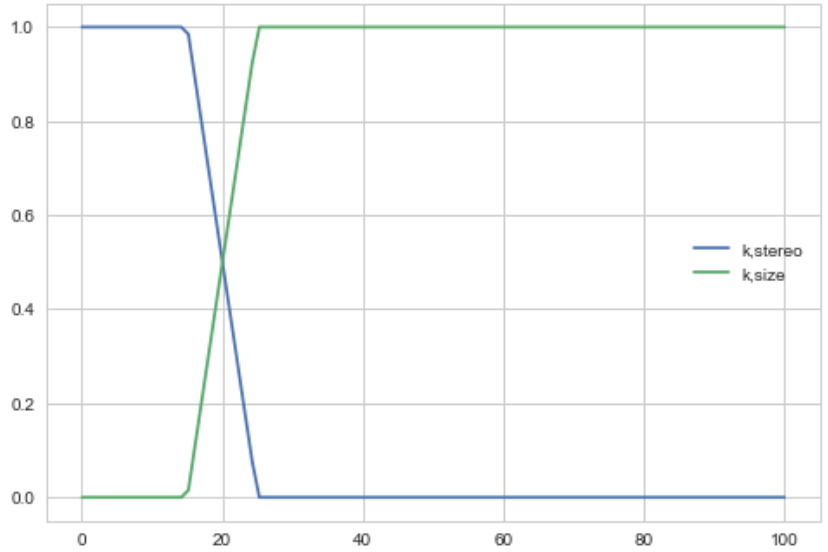
$$d_{stereo} = \frac{f * b}{\cos(\alpha) * \cos(\beta) * ds * s_w / p_w}$$

s_w= *sensor width* (m)
p_w= *image width (pixels)*
f = focal length (m)
b = base line (m)
ds = disparity (pixels)
α = *altitude (rad)*
β = *azimuth (rad)*

Combining distance estimates:

$$d = k_{stereo} * d_{stereo} + k_{size} * d_{size}$$

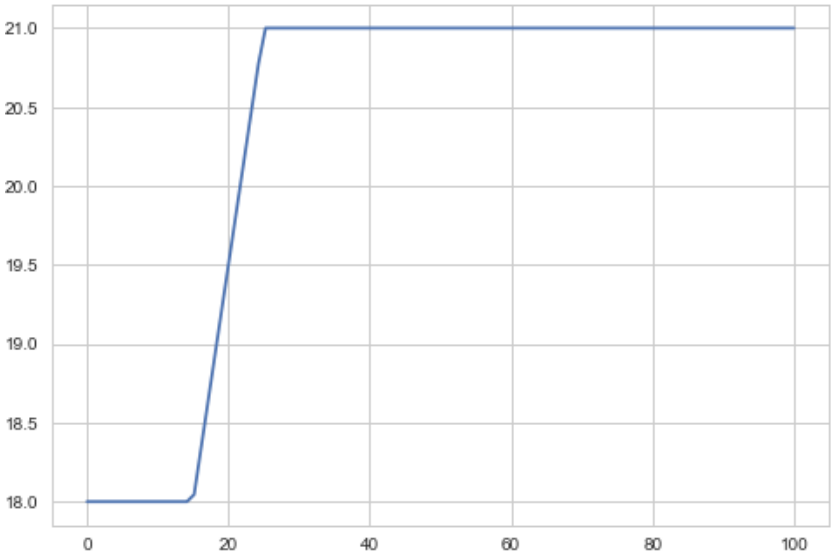
```
fraction = 0.25
def calculate_coefficients(estimated_distance, stereo_max_distance):
    if estimated_distance < (1-fraction)*stereo_max_distance:
        k_size = 0.0
        k_stereo = 1.0
    elif estimated_distance > (1+fraction)*stereo_max_distance:
        k_size = 1.0
        k_stereo = 0.0
    else:
        l1 = estimated_distance - (1-fraction)*stereo_max_distance
        l2 = (1+fraction)*stereo_max_distance - (1-fraction)*stereo_max_distance
        k_size = l1/l2
        k_stereo = 1 - k_size
    return k_stereo, k_size
```



stereo_max_distance = 20.0

Initialization:

$$estimated\_distance = 0.5 * d_{stereo} + 0.5 * d_{size}$$



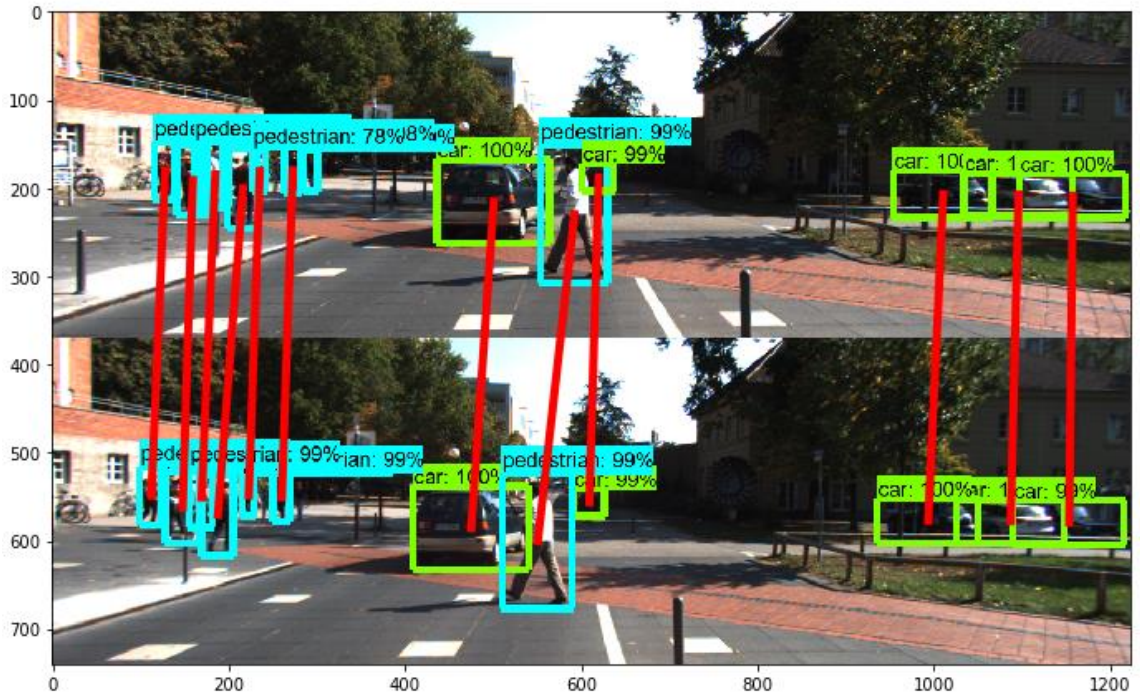

```
dstereo = 18.0
dsize = 21.0
```

Procedure is iterated until convergence or max_iter (or just used once?)

Estimating disparity using matched patterns

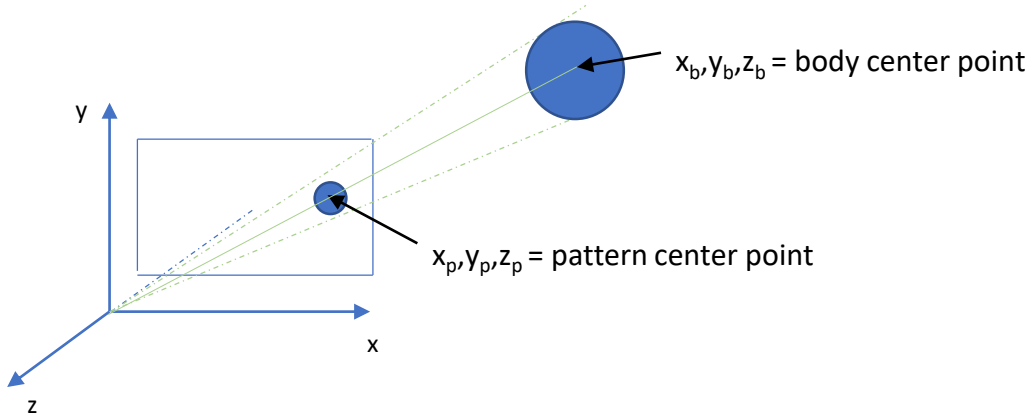
Left camera pattern:
x_min_left
x_max_left

Right camera pattern:
x_min_right
x_max_right



$Pattern\ disparity$ = 0.5 * (x_min_left+x_max_left)-0.5*(x_min_right+x_max_right)

3D projection



$$(x_b, y_b, z_b) = t^* (x_p, y_p, z_p)$$

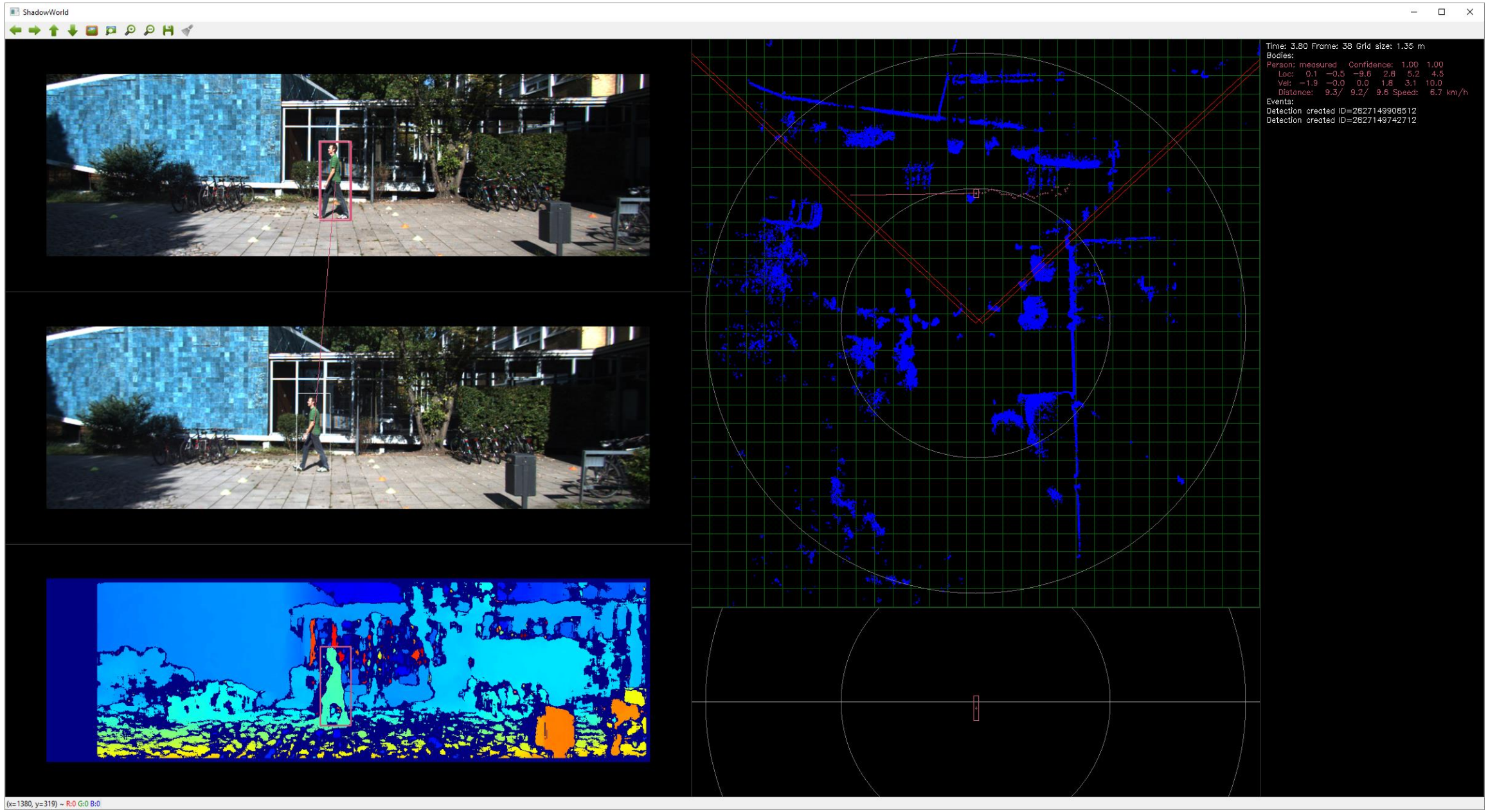
Where:

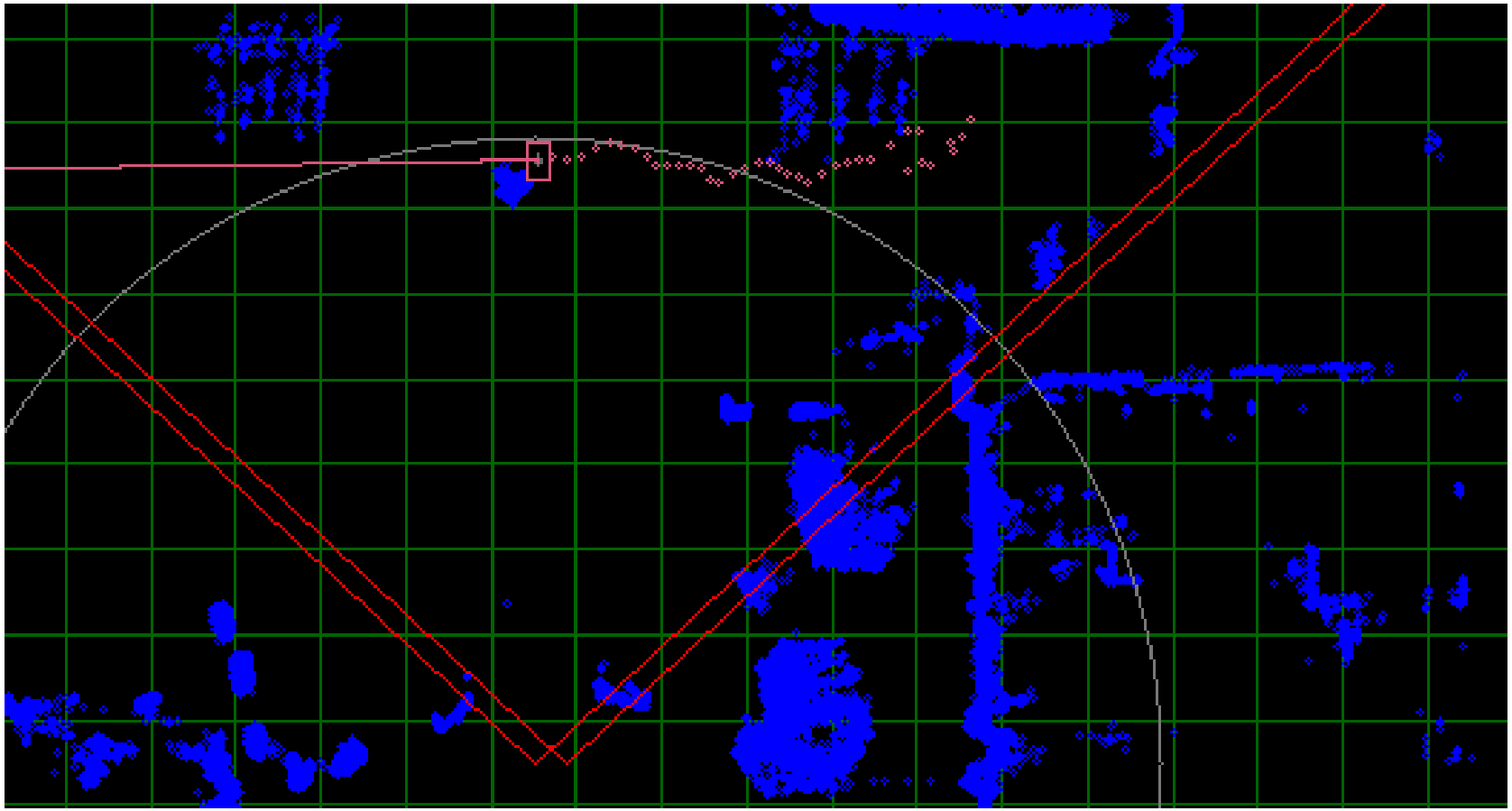
$$(x_p, y_p, z_p) = \left(-\frac{s_w}{2} + p_x * \frac{s_w}{p_w},\ \frac{s_h}{2} - p_y * \frac{s_h}{p_h},\ -f\right)$$

$$t = \frac{d}{\sqrt{x_p{}^2 + y_p{}^2 + z_p{}^2}}$$

s_w= *sensor width* (m)
s_h= *sensor height* (m)
p_w= *image width (pixels)*
p_h= *image height (pixels)*
f = *focal length (m)*
p_x = *pattern center point location (x, pixels)*
p_y = *pattern center point location (y, pixels)*

Note! Only left image used. Right image is used only for disparity calculation (in the context of distance estimation and 3D projection).


ShadowWorld
Time: 3.80 Frame: 38 Grid size: 1.35 m
Bodies:
Person: measured Confidence: 1.00 1.00
Loc: 0.1 −0.5 −9.6 2.8 5.2 4.5
Vel: −1.9 −0.0 0.0 1.8 3.1 10.0
Distance: 9.3/ 9.2/ 9.6 Speed: 6.7 km/h
Events:
Detection created ID=2627149908512
Detection created ID=2627149742712
(x=1380, y=319) ~ R:0 G:0 B:0



Localizations follows Velodyne cloud pretty well!

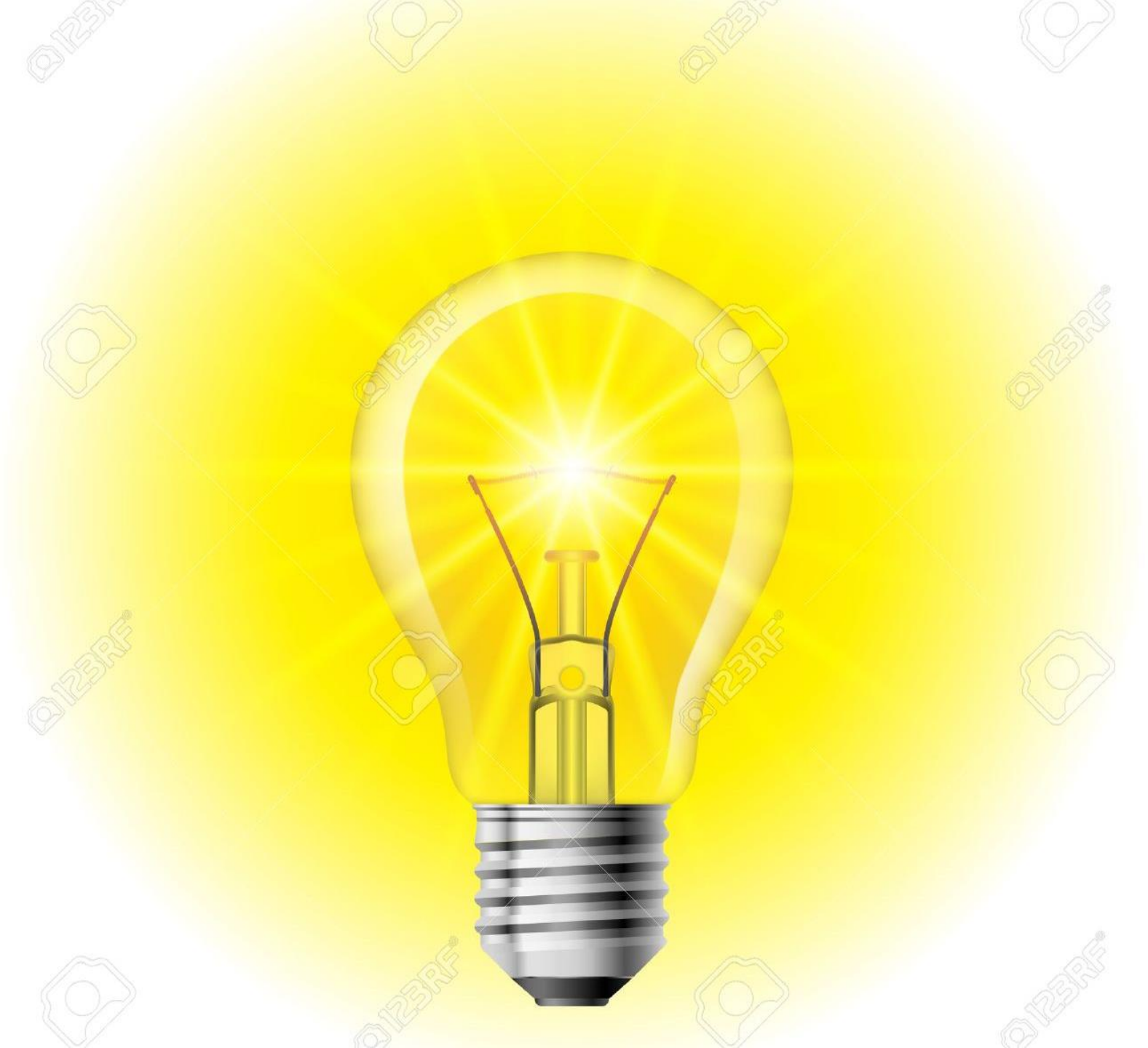
Current overall matching algorithm:

- For left camera:
  - Create detections
  - Predict pattern locations
  - Match detections to patterns
  - For each unmatched detection
    - Create pattern
    - Create body
- For right camera:
  - Create detections
- Match right camera detections to left camera patterns
- Predict body locations
- Create location measurements based on size and stereo
- Correct body locations

Note: Pattern and body removals are not described.

Research topic!

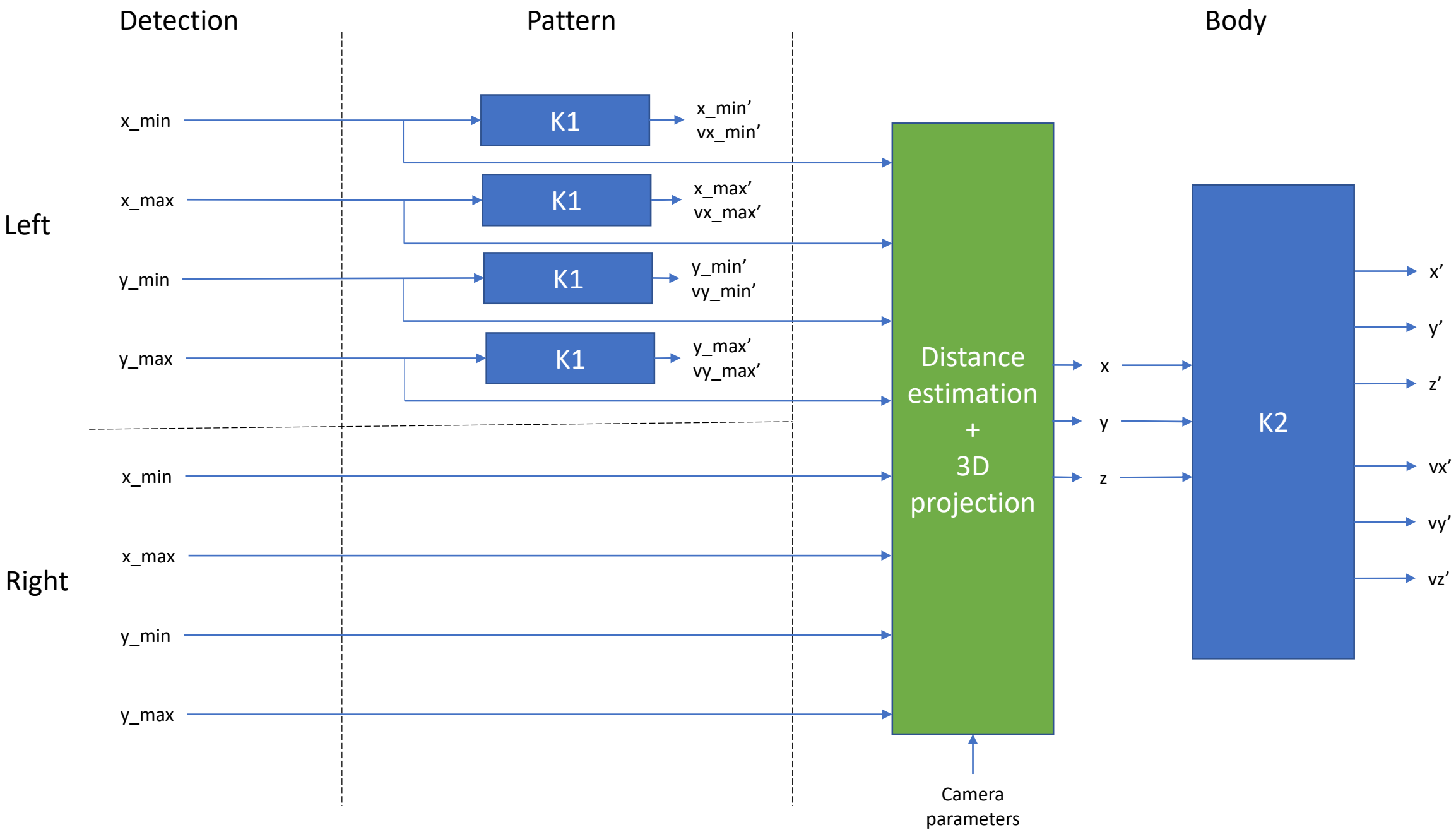
The joint estimation of pattern and body state would be a good area for further study. The current algorithm is by no means optimal.



Kalman Filter Parameter Adjustments

Two levels of filtering:

1. Pattern
   - Pattern movement estimation for missing detections
2. Body
   - Coordinate and velocity estimation


Detection
Pattern
Body
Left
Right
x_min
x_max
y_min
y_max
K1
K1
K1
K1
x_min'
vx_min'
x_max'
vx_max'
y_min'
vy_min'
y_max'
vy_max'
x_min
x_max
y_min
y_max
Distance estimation + 3D projection
x
y
z
K2
x'
y'
z'
vx'
vy'
vz'
Camera parameters

Before

```
PATTERN_ALFA = 200.0 # Pattern initial location error variance
PATTERN_BETA = 10000.0 # Pattern initial velocity error variance
PATTERN_C = np.array([[1.0, 0.0]]) # Pattern measurement matrix
PATTERN_Q = np.array([200.0]) # Pattern measurement variance
PATTERN_R = np.array([[0.1, 0.0],
                      [0.0, 1.0]]) # Pattern state equation covariance
```

```
BODY_ALFA = 100000.0 # Body initial location error variance
BODY_BETA = 100000.0 # Body initial velocity error variance
BODY_C = np.array([[1.0, 0.0, 0.0, 0.0, 0.0, 0.0],
                   [0.0, 1.0, 0.0, 0.0, 0.0, 0.0],
                   [0.0, 0.0, 1.0, 0.0, 0.0, 0.0]
                  ]) # Body measurement matrix
BODY_DATA_COLLECTION_COUNT = 30 # How many frames until notification
BODY_Q = np.array([[200.0, 0.0,   0.0],
                   [0.0, 200.0,   0.0],
                   [0.0,   0.0, 200.0]]) # Body measurement variance 200
BODY_R = np.array([[0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
                   [0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
                   [0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
                   [0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
                   [0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
                   [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
                  ]) # Body state equation covariance
```

After

```
PATTERN_ALFA = 100.0 # Pattern initial location error variance
PATTERN_BETA = 10000.0 # Pattern initial velocity error variance
PATTERN_C = np.array([[1.0, 0.0]]) # Pattern measurement matrix
PATTERN_Q = np.array([100.0]) # Pattern measurement variance
PATTERN_R = np.array([[0.0, 0.0],
                      [0.0, 100.0]]) # Pattern state equation covariance
```

```
BODY_ALFA = 10.0 # Body initial location error variance
BODY_BETA = 100.0 # Body initial velocity error variance
BODY_C = np.array([[1.0, 0.0, 0.0, 0.0, 0.0, 0.0],
                   [0.0, 1.0, 0.0, 0.0, 0.0, 0.0],
                   [0.0, 0.0, 1.0, 0.0, 0.0, 0.0]
                  ]) # Body measurement matrix
BODY_DATA_COLLECTION_COUNT = 30 # How many frames until notification
BODY_Q = np.array([[25.0, 0.0,  0.0],
                   [0.0, 25.0,  0.0],
                   [0.0,  0.0, 25.0]]) # Body measurement variance 200
BODY_R = np.array([[0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
                   [0.0, 1.0, 0.0, 0.0, 0.0, 0.0],
                   [0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
                   [0.0, 0.0, 0.0, 0.1, 0.0, 0.0],
                   [0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
                   [0.0, 0.0, 0.0, 0.0, 0.0, 1.0]
                  ]) # Body state equation covariance
```

```
PATTERN_ALFA = 100.0 # Pattern initial location error variance
PATTERN_BETA = 10000.0 # Pattern initial velocity error variance
PATTERN_C = np.array([[1.0, 0.0]]) # Pattern measurement matrix
PATTERN_Q = np.array([100.0]) # Pattern measurement variance
PATTERN_R = np.array([[0.0, 0.0],
                      [0.0, 100.0]]) # Pattern state equation covariance
```

PATTERN_ALFA
- 2*std = 20px, first detection bounding box accuracy

PATTERN_BETA
- 2*std = 200 px/s = 20 px / frame

PATTERN_Q
- 2*std = 20px, detection accuracy

PATTERN_R
- Location, no teleporting
- Velocity, 2*std = 20px/frame = 200 px/s

```
BODY_ALFA = 10.0 # Body initial location error variance
BODY_BETA = 100.0 # Body initial velocity error variance
BODY_C = np.array([[1.0, 0.0, 0.0, 0.0, 0.0, 0.0],
                   [0.0, 1.0, 0.0, 0.0, 0.0, 0.0],
                   [0.0, 0.0, 1.0, 0.0, 0.0, 0.0]
                  ]) # Body measurement matrix
BODY_DATA_COLLECTION_COUNT = 30 # How many frames until notification
BODY_Q = np.array([[25.0, 0.0,  0.0],
                   [0.0, 25.0,  0.0],
                   [0.0,  0.0, 25.0]]) # Body measurement variance 200
BODY_R = np.array([[0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
                   [0.0, 1.0, 0.0, 0.0, 0.0, 0.0],
                   [0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
                   [0.0, 0.0, 0.0, 0.1, 0.0, 0.0],
                   [0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
                   [0.0, 0.0, 0.0, 0.0, 0.0, 1.0]
                  ]) # Body state equation covariance
```

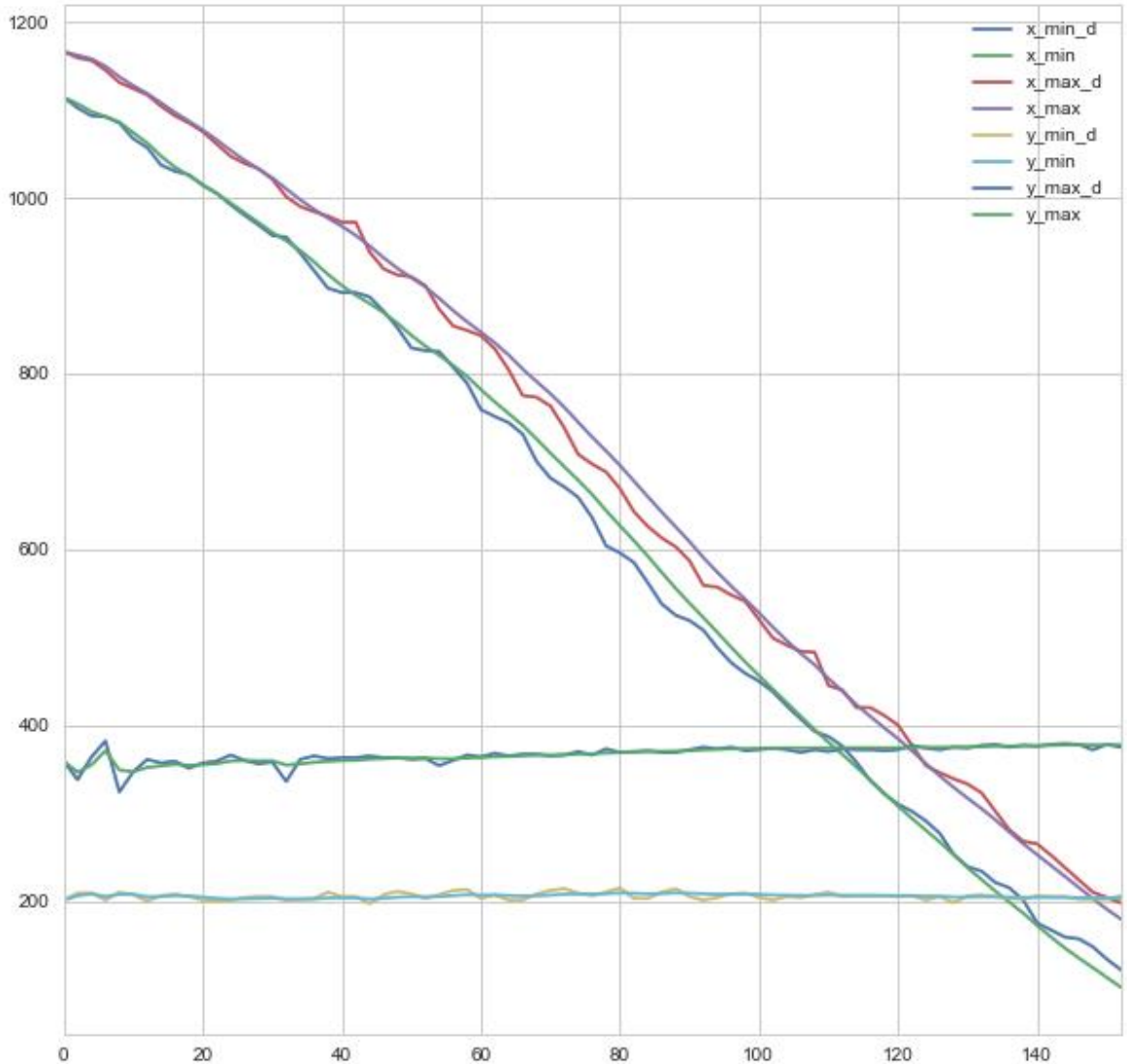
BODY_ALFA
- 2*std = 6 m, first location accuracy

BODY_BETA
- 2*std = 20 m/s

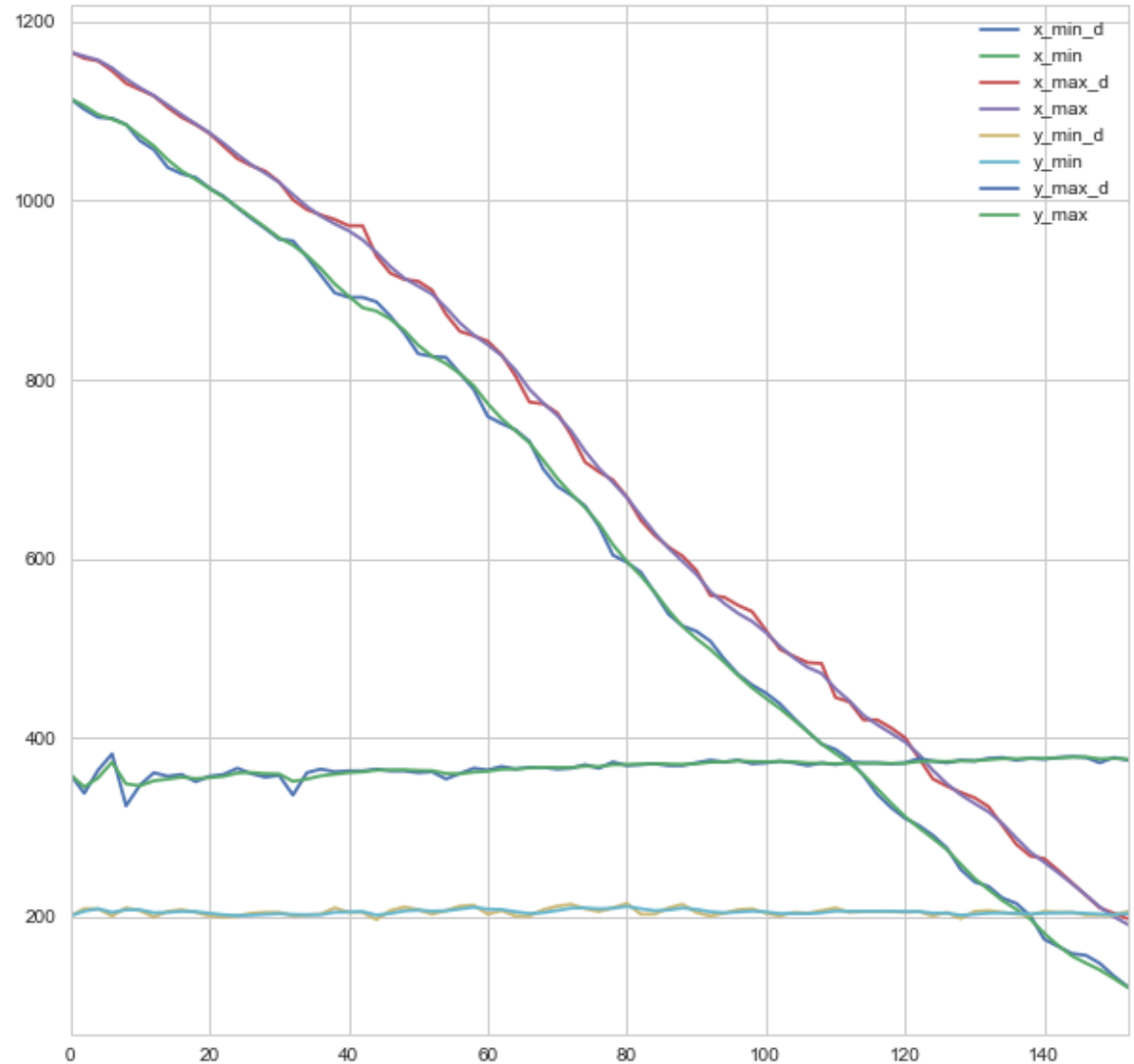
BODY_Q
- 2*std = 10m, measurement accuracy

BODY_R
- Location, no teleporting
- Velocity (x,z), 2*std = 2 m/s2
- Velocity (y), 2*std = 0,6 m/s2

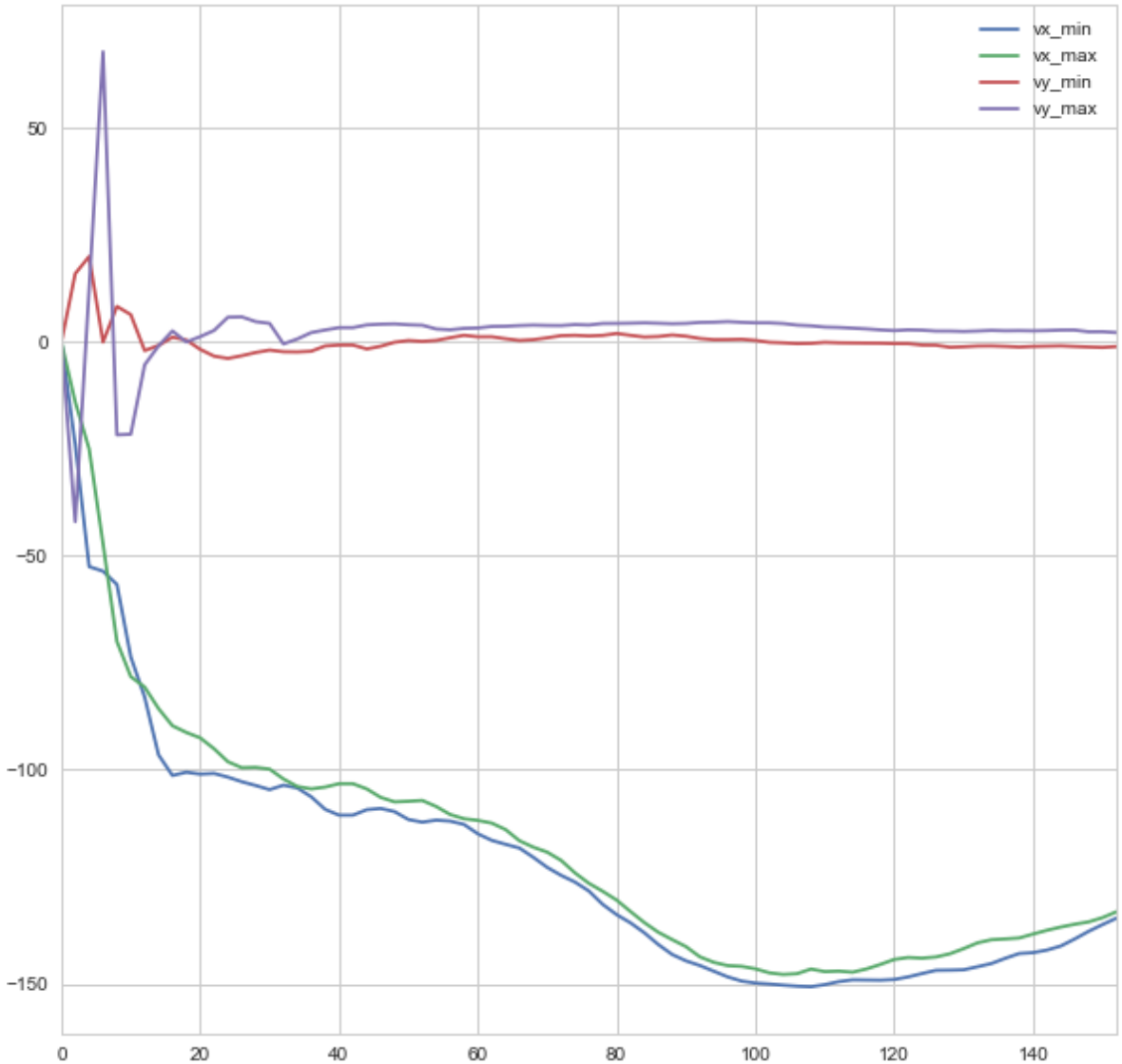
Pattern location / before



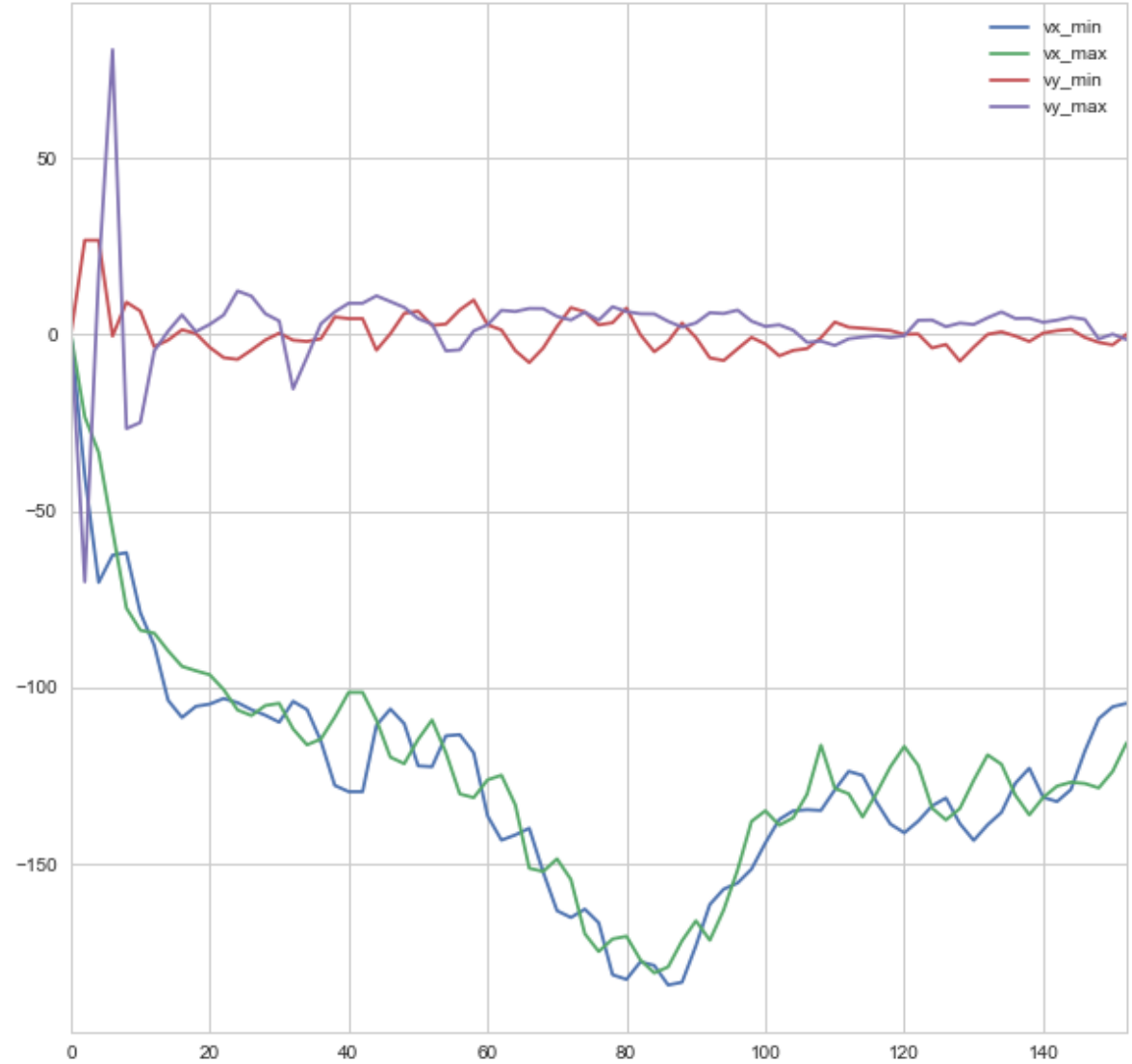
Pattern location / after



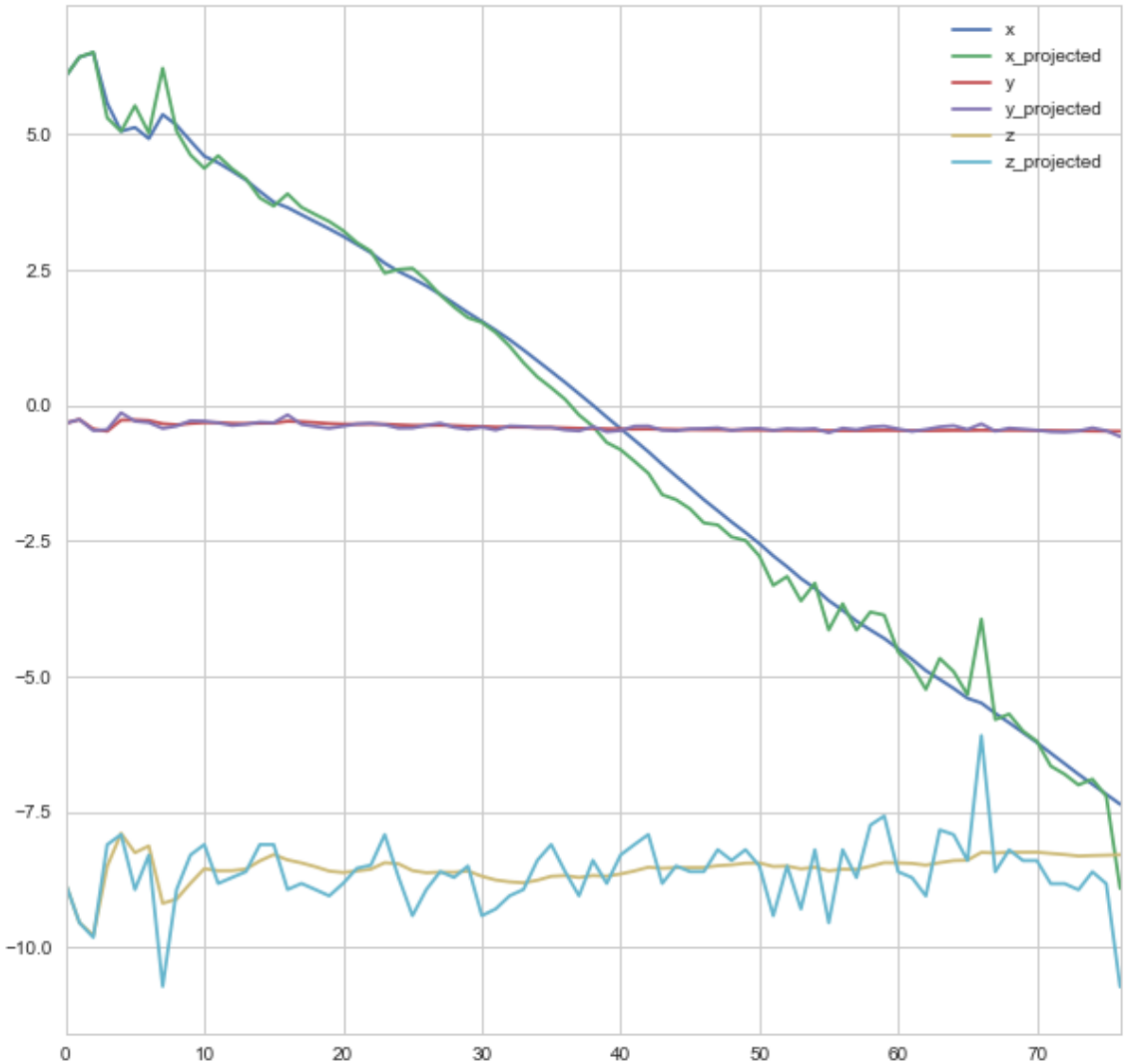
Pattern velocity / before



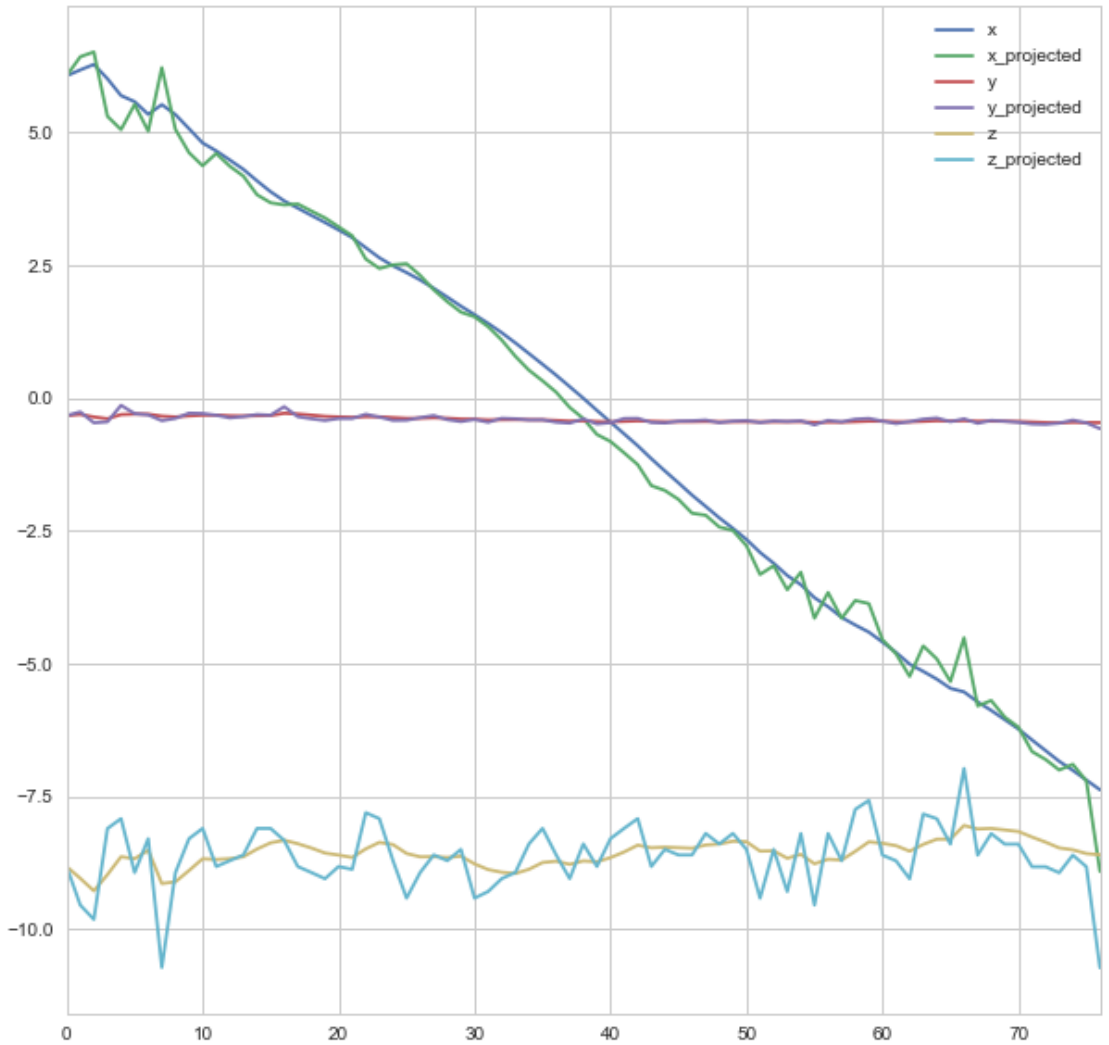
Pattern location / after



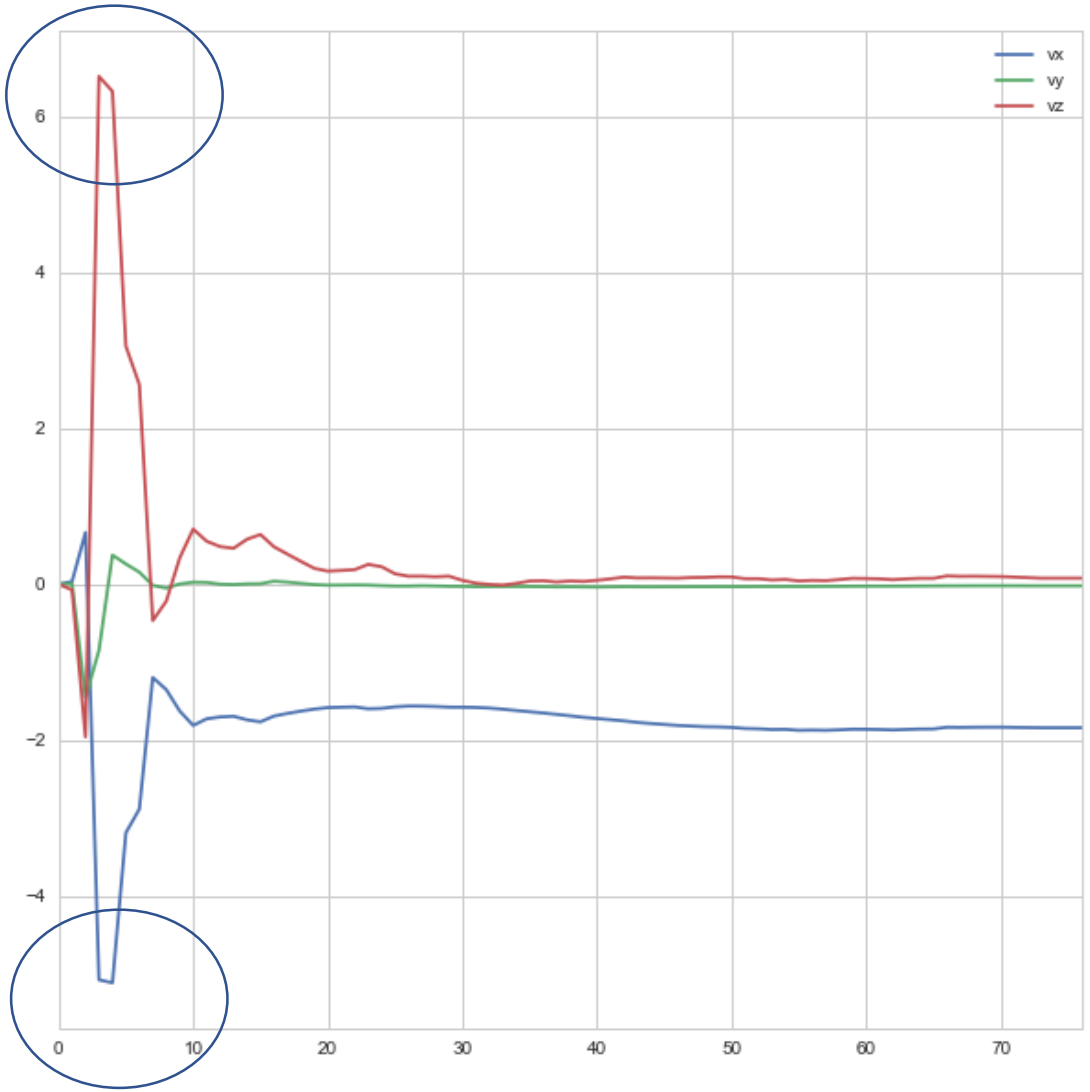
Body location / before



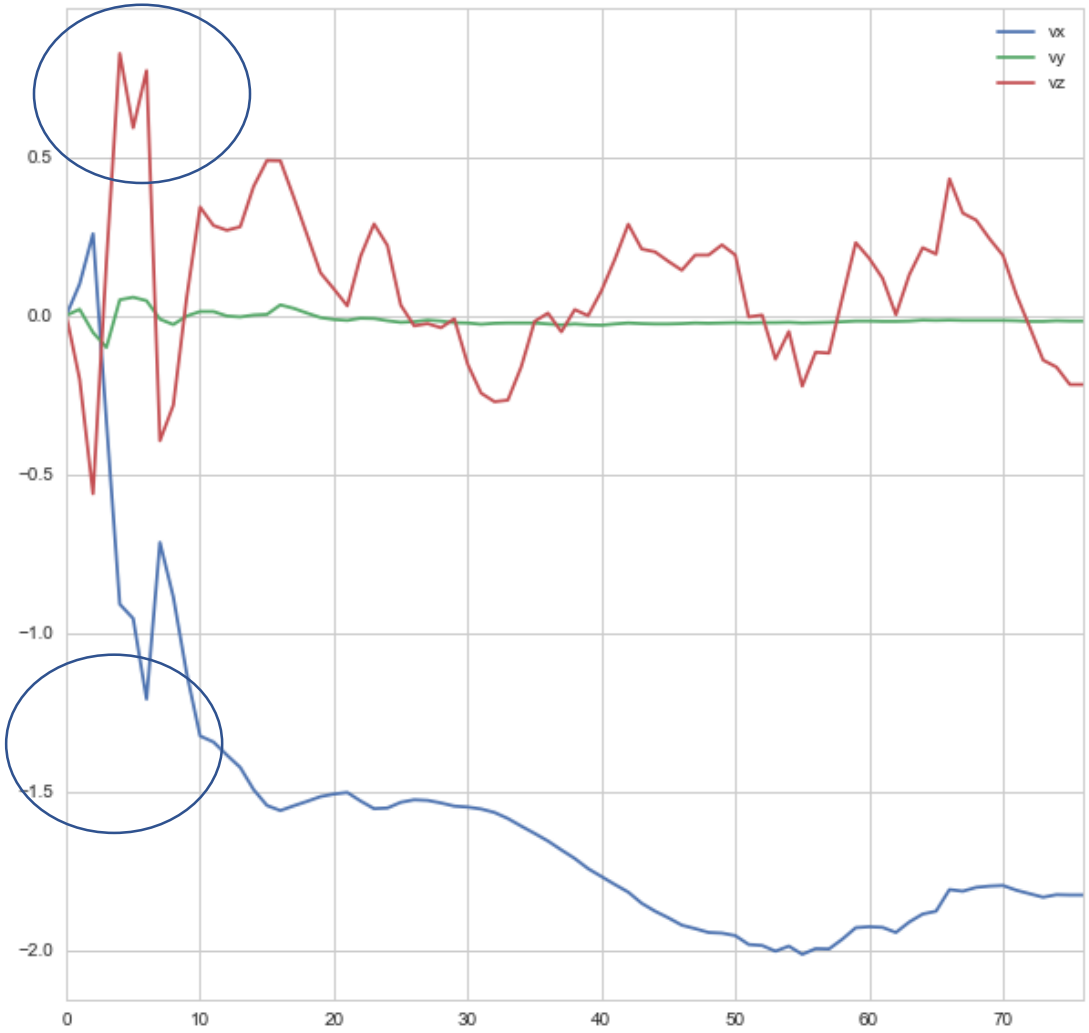
Body location / after



Body location / before



Body location / after



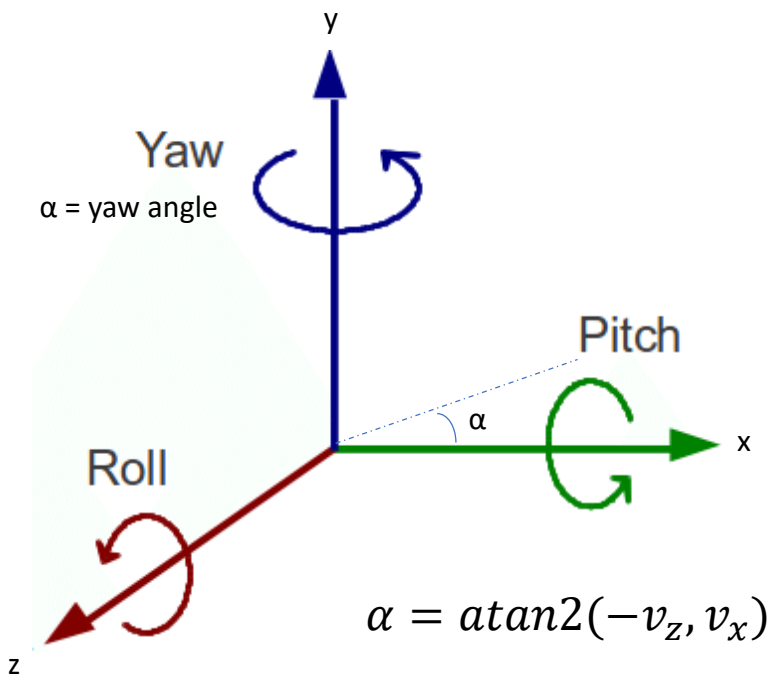
Research topic!

Note. The current filter architecture and parameters do their job, but certainly are not optimal. Field for further study!

Body Volume and Orientation

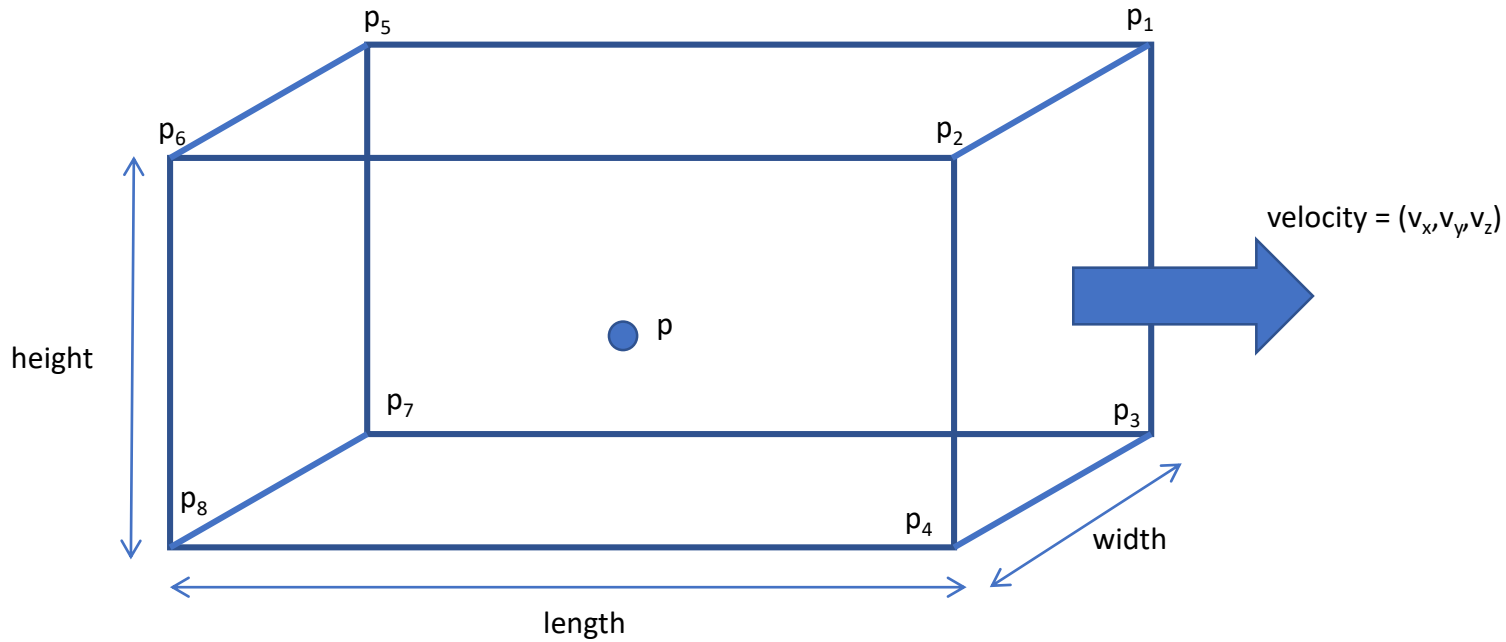
- Bounding sphere is not a good shape for volumes like car, human or train…
- KITTI uses bounded 3D box

Assumption: Only yaw allowed



$$\alpha = atan2(-v_z, v_x)$$

Each body is supposed to have “normal” movement direction, which defines length, height and width.

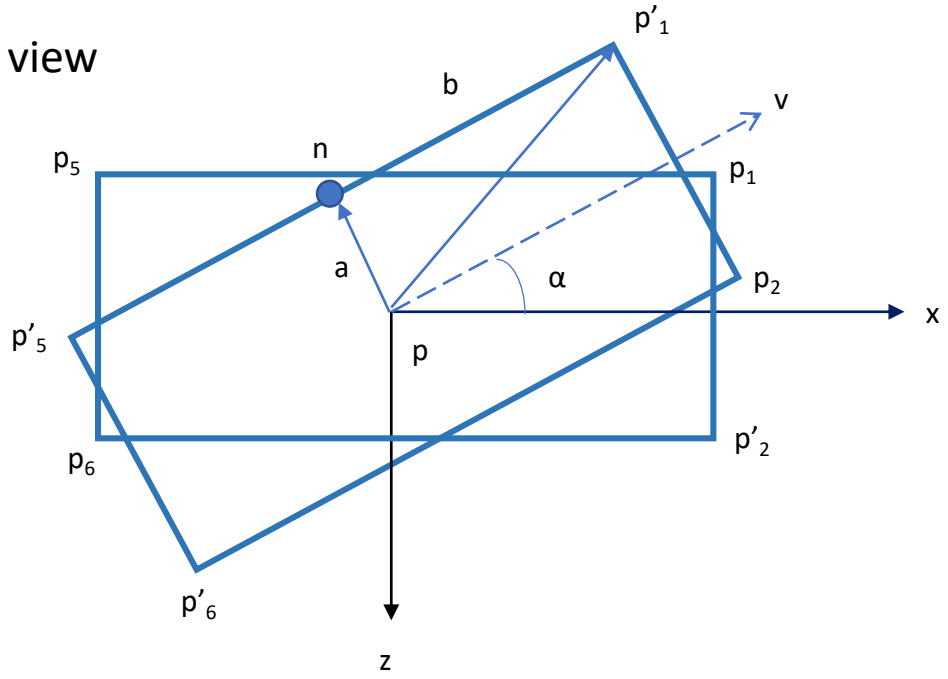


$$(p_{1,x}, p_{1,y}, p_{1,z}) = (p_x + 0.5 * length,\ p_y + 0.5 * heigth,\ p_z - 0.5 * width)$$

…

$$(p_{8,x}, p_{8,y}, p_{8,z}) = (p_x - 0.5 * length,\ p_y - 0.5 * heigth,\ p_z + 0.5 * width)$$

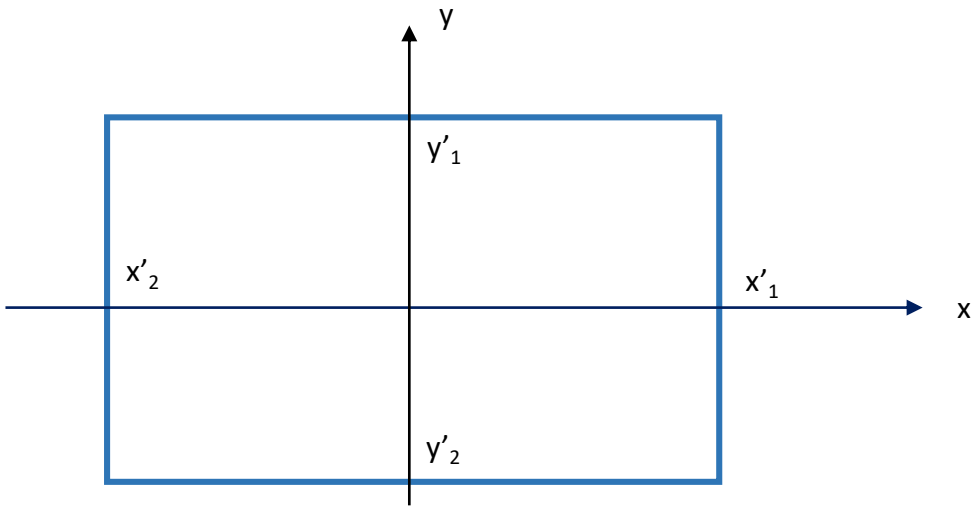
Top view



$$\alpha = atan2(-v_z, v_x)$$

$$\begin{bmatrix} p'_{x,i} \\ p'_{y,i} \\ p'_{z,i} \end{bmatrix} = \begin{bmatrix} cos(\alpha) & 0 & \sin(\alpha) \\ 0 & 1 & 0 \\ -\sin(\alpha) & 0 & \cos(\alpha) \end{bmatrix} * \begin{bmatrix} p_{x,i} \\ p_{y,i} \\ p_{z,i} \end{bmatrix}$$

Side view

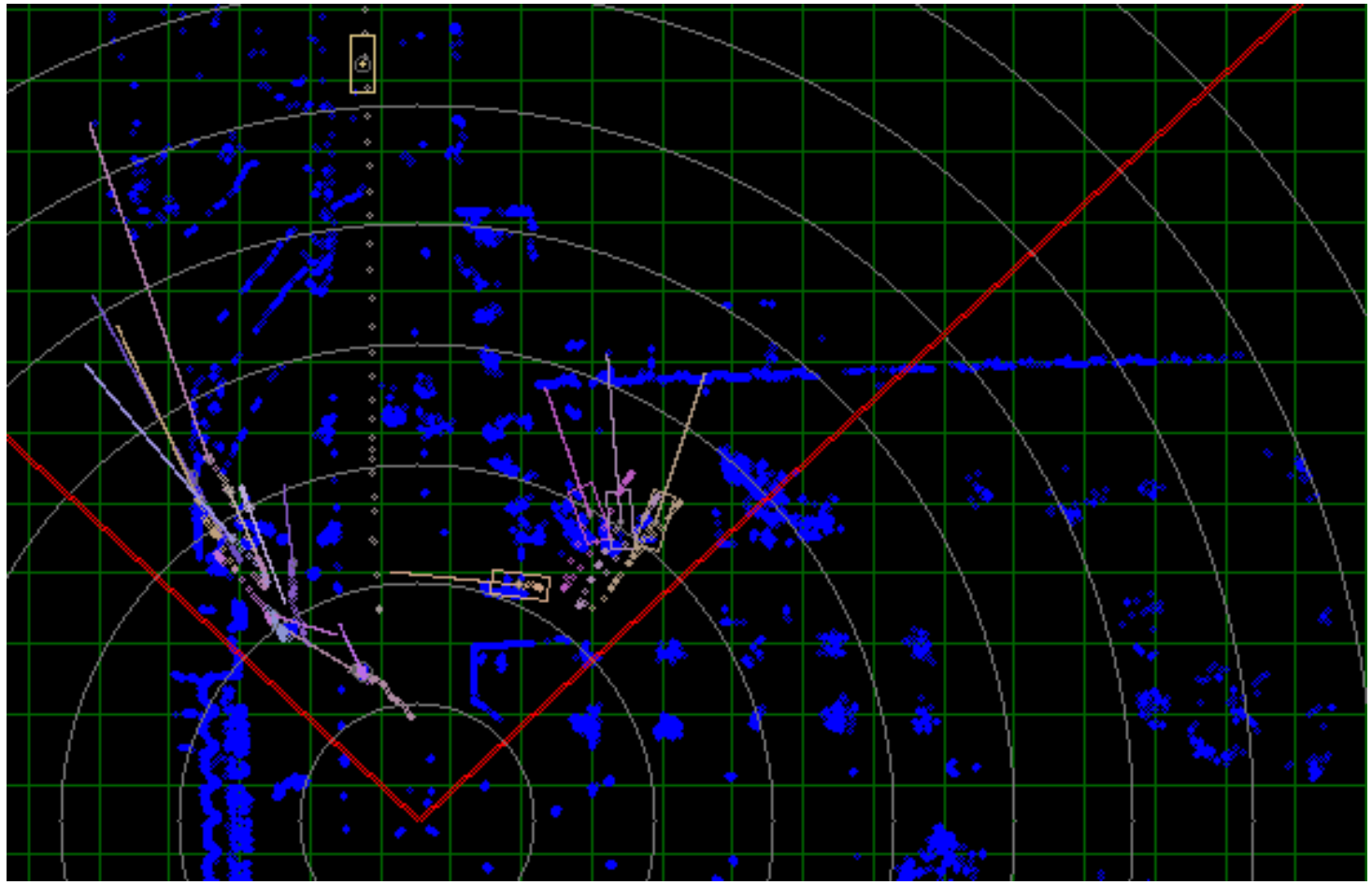


$$y'_1 = height/2$$

$$y'_2 = -height/2$$

$$x'_1 = \max(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$$

$$x'_2 = min(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$$

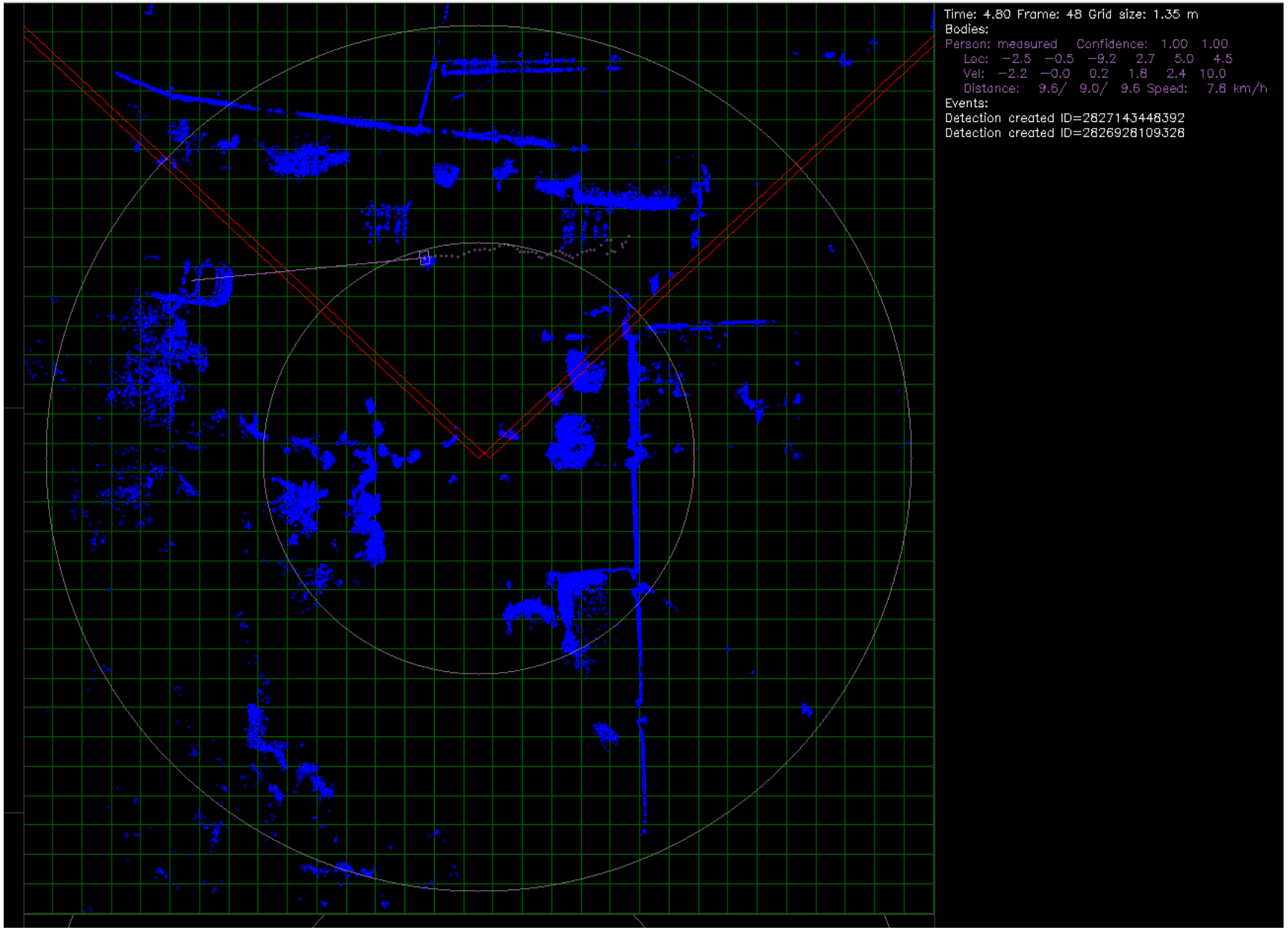


Currently used for display. Collision detection still uses spheres. TBD

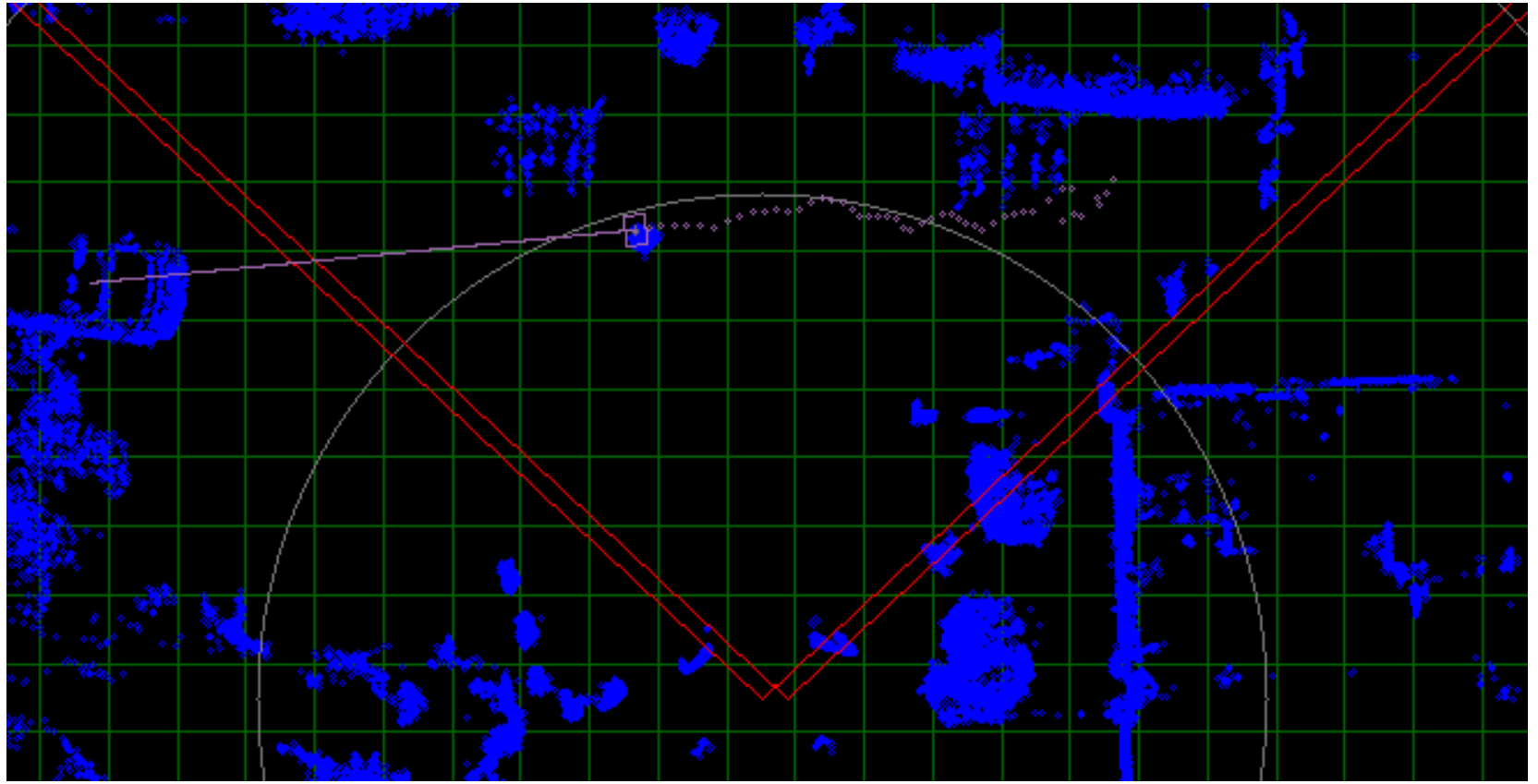
Grid Representation

Background:

Representing body movement forecast and probabilities in continuous world is presumably much more difficult than in discrete space. Grid is needed…



2D grid is used, with evenly spaced lines.
GRID_COUNT parameter (=31)



Representing conditional path statistics and probabilities is easier with grid.

Note. KITTI dataset includes ”Person” category wich is well suited to do research.

Data Category: Person

Before browsing, please wait some moments until this page is fully loaded.

2011_09_28_drive_0053 (0.3 GB)
Length: 74 frames (00:07 minutes)
Image resolution: 1392 x 512 pixels
Labels: 0 Cars, 0 Vans, 0 Trucks, 0 Pedestrians, 0 Sitters, 0 Cyclists, 0 Trams, 0 Misc
Downloads: [unsynced+unrectified data] [synced+rectified data] [calibration]

2011_09_28_drive_0054 (0.2 GB)
Length: 51 frames (00:05 minutes)
Image resolution: 1392 x 512 pixels
Labels: 0 Cars, 0 Vans, 0 Trucks, 0 Pedestrians, 0 Sitters, 0 Cyclists, 0 Trams, 0 Misc
Downloads: [unsynced+unrectified data] [synced+rectified data] [calibration]

2011_09_28_drive_0057 (0.3 GB)
Length: 80 frames (00:08 minutes)
Image resolution: 1392 x 512 pixels
Labels: 0 Cars, 0 Vans, 0 Trucks, 0 Pedestrians, 0 Sitters, 0 Cyclists, 0 Trams, 0 Misc
Downloads: [unsynced+unrectified data] [synced+rectified data] [calibration]

2011_09_28_drive_0065 (0.2 GB)
Length: 45 frames (00:04 minutes)
Image resolution: 1392 x 512 pixels
Labels: 0 Cars, 0 Vans, 0 Trucks, 0 Pedestrians, 0 Sitters, 0 Cyclists, 0 Trams, 0 Misc
Downloads: [unsynced+unrectified data] [synced+rectified data] [calibration]

2011_09_28_drive_0066 (0.1 GB)
Length: 35 frames (00:03 minutes)
Image resolution: 1392 x 512 pixels
Labels: 0 Cars, 0 Vans, 0 Trucks, 0 Pedestrians, 0 Sitters, 0 Cyclists, 0 Trams, 0 Misc
Downloads: [unsynced+unrectified data] [synced+rectified data] [calibration]

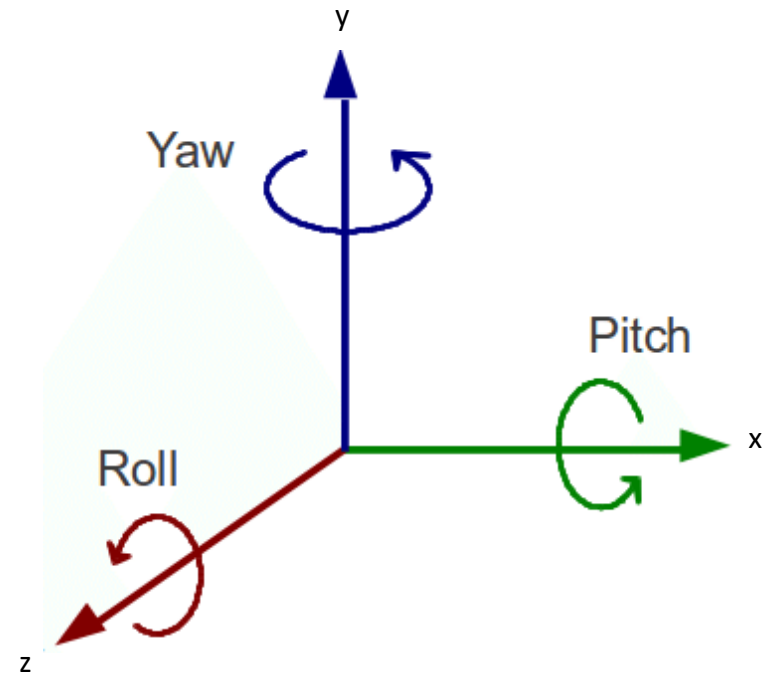
2011_09_28_drive_0068 (0.3 GB)
Length: 73 frames (00:07 minutes)
Image resolution: 1392 x 512 pixels
Labels: 0 Cars, 0 Vans, 0 Trucks, 0 Pedestrians, 0 Sitters, 0 Cyclists, 0 Trams, 0 Misc
Downloads: [unsynced+unrectified data] [synced+rectified data] [calibration]

2011_09_28_drive_0070 (0.2 GB)
Length: 45 frames (00:04 minutes)

Research topic!

Nearly 100 videos where one or more persons are walking using different paths across the same area.

World Coordinates, Yaw Estimation



Assumptions:

- initially world coordinates coincide with left camera
- y, roll and pitch are fixed
- x and z are estimated from odometry (GPS)
- yaw is estimated from optical flow, or given

Camera has three degrees of freedom:

- x
- z
- yaw

Transformation from (left) camera coordinates into world coordinates:

Rotation + translation. Also known as 3D *rigid body motion* or the 3D *Euclidean transformation*, it can be written as $\boldsymbol{x}' = \boldsymbol{R}\boldsymbol{x} + \boldsymbol{t}$ or

$$\boldsymbol{x}' = \begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \end{bmatrix} \bar{\boldsymbol{x}} \tag{2.24}$$

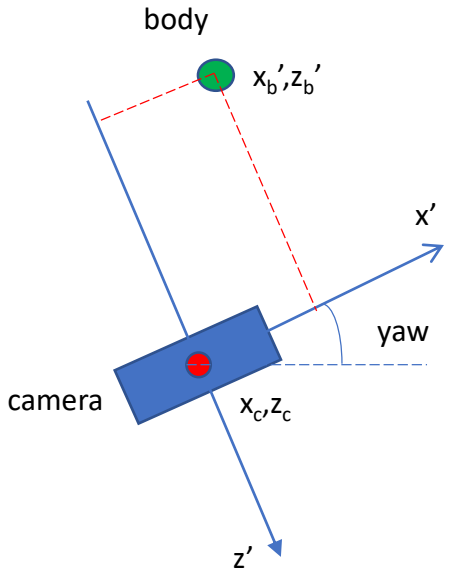
where $\boldsymbol{R}$ is a 3×3 orthonormal rotation matrix with $\boldsymbol{R}\boldsymbol{R}^T = \boldsymbol{I}$ and $|\boldsymbol{R}| = 1$. Note that sometimes it is more convenient to describe a rigid motion using

$$\boldsymbol{x}' = \boldsymbol{R}(\boldsymbol{x} - \boldsymbol{c}) = \boldsymbol{R}\boldsymbol{x} - \boldsymbol{R}\boldsymbol{c}, \tag{2.25}$$

where $\boldsymbol{c}$ is the center of rotation (often the camera center).

(Szeliski)

c from odometry
R from optical flow



From camera coordinates into world coordinates:

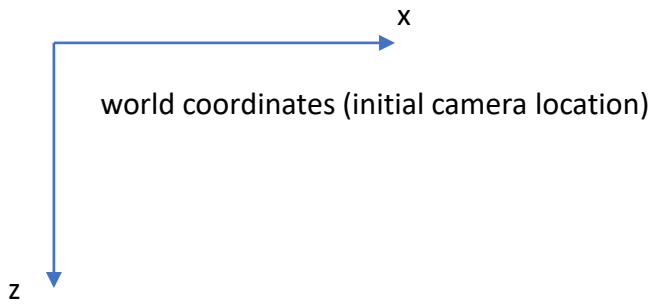
$$x_b = x_c + \cos(yaw) * x_b' + \sin(yaw) * z_b'$$

$$z_b = z_c - \sin(yaw) * x_b' + \cos(yaw) * z_b'$$

From world coordinates into camera coordinates:

$$x_b' = \cos(yaw) * (x_b - x_c) - \sin(yaw) * (z_b - z_c)$$

$$z_b' = \sin(yaw) * (x_b - x_c) + \cos(yaw) * (z_b - z_c)$$



If more degrees of freedom are wanted:

Which rotation representation is better?

The choice of representation for 3D rotations depends partly on the application.

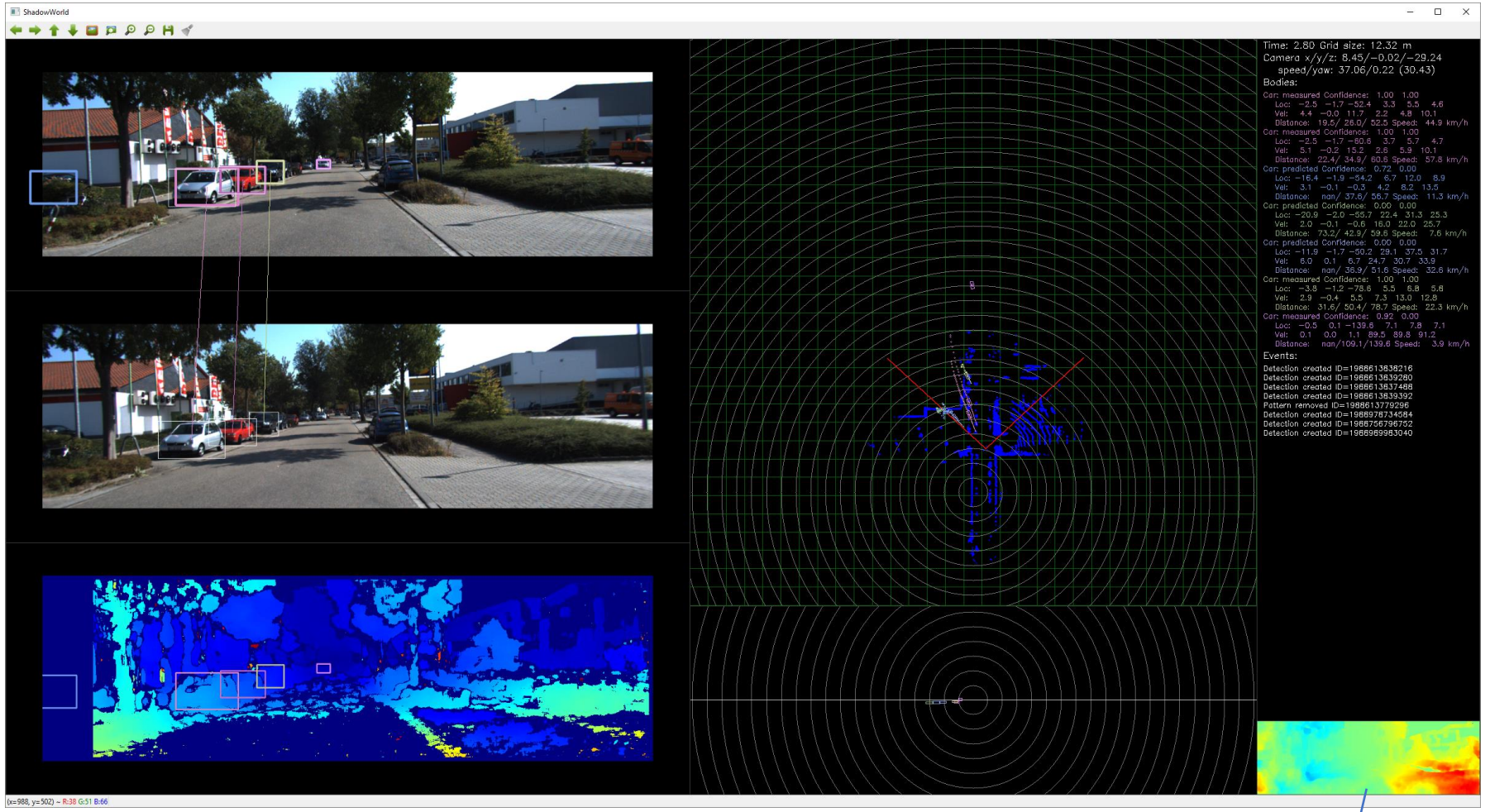
The axis/angle representation is minimal, and hence does not require any additional constraints on the parameters (no need to re-normalize after each update). If the angle is expressed in degrees, it is easier to understand the pose (say, 90° twist around x-axis), and also easier to express exact rotations. When the angle is in radians, the derivatives of $\boldsymbol{R}$ with respect to $\boldsymbol{\omega}$ can easily be computed (2.36).

Quaternions, on the other hand, are better if you want to keep track of a smoothly moving camera, since there are no discontinuities in the representation. It is also easier to interpolate between rotations and to chain rigid transformations (Murray, Li, and Sastry 1994; Bregler and Malik 1998).

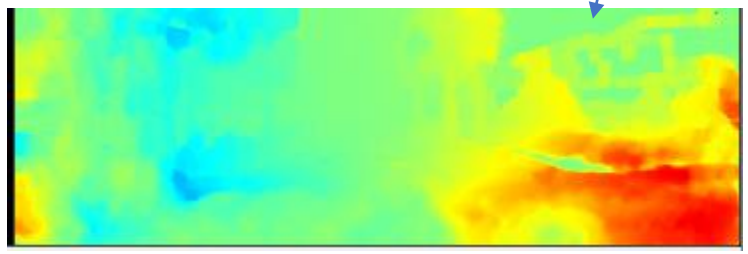
My usual preference is to use quaternions, but to update their estimates using an incremental rotation, as described in Section 6.2.2.

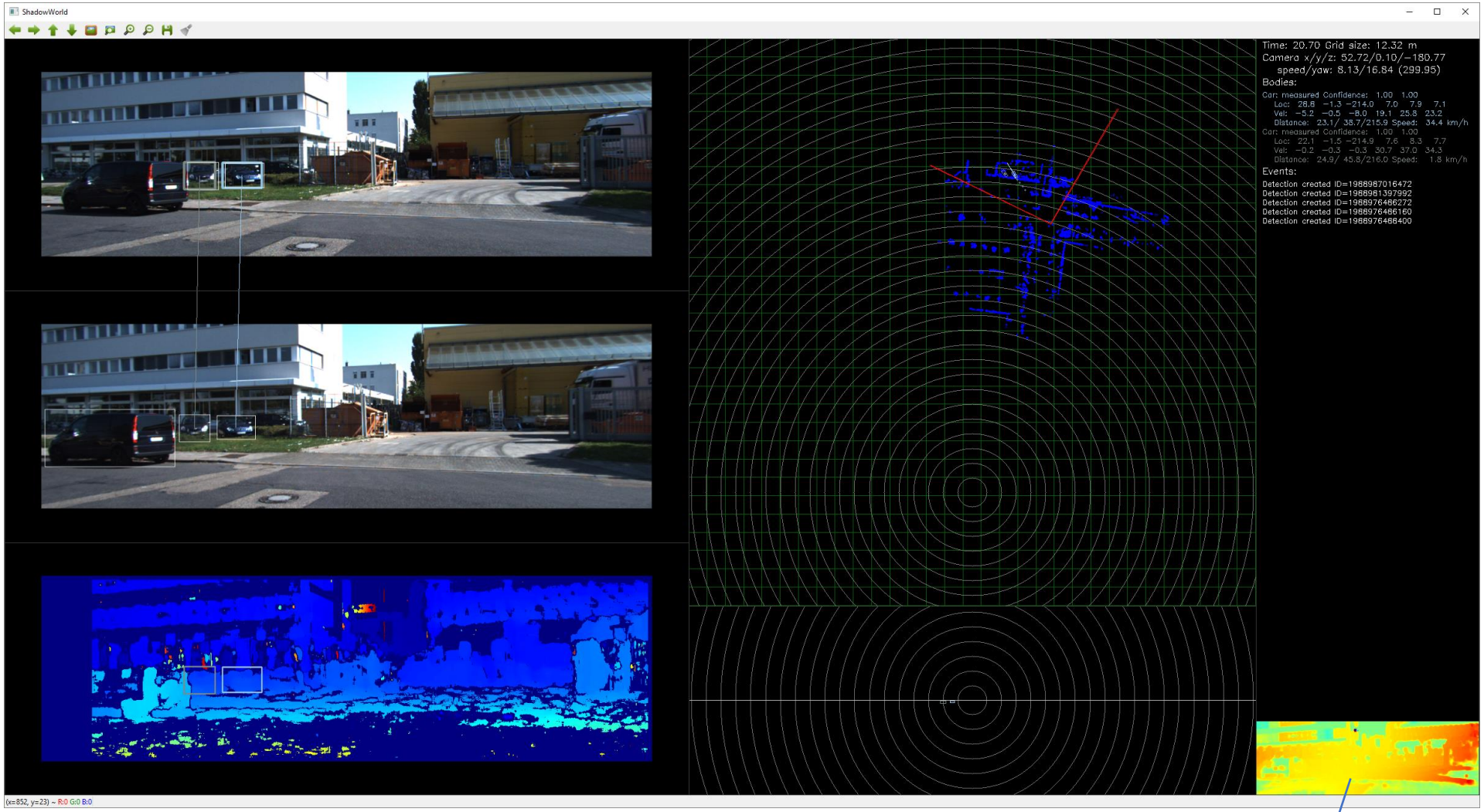
(Szeliski)

So, quaternions, but later when needed…

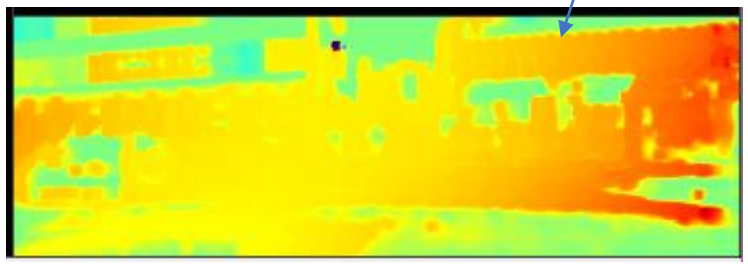


Car going straight ahead. Optical flow to left is coloured blue, and flow to right red. It is calculated using Farneback algorithm.

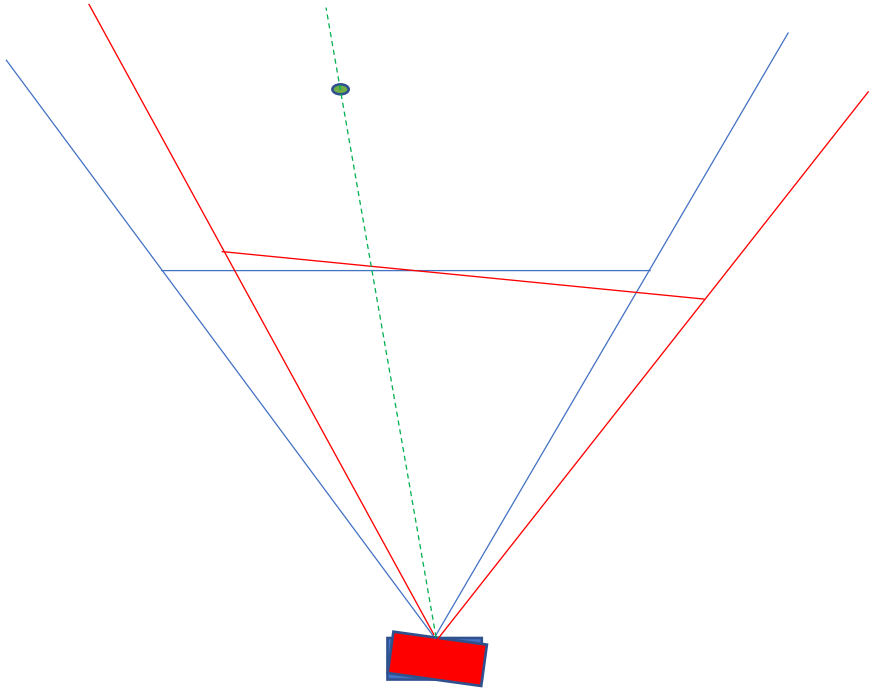




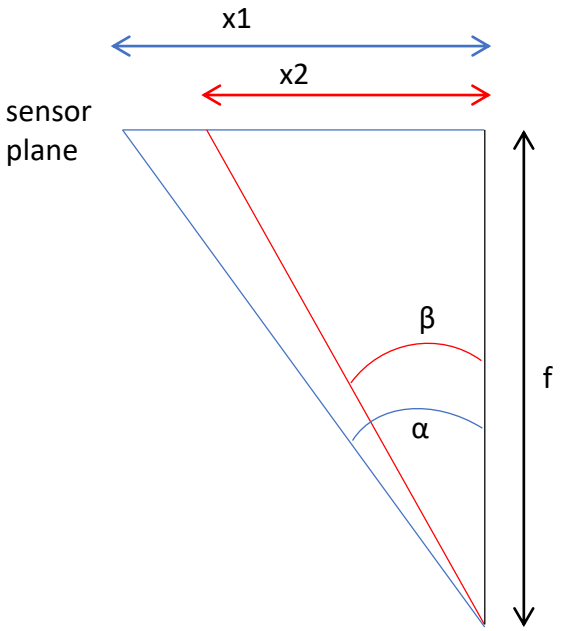
Car turning left. Optical flow is mainly to right and colored red/yellow.



From optical flow to yaw



Approximation for small turns:



$$x_1 = f * \tan(\alpha) \qquad x_2 = f * \tan(\beta)$$

$$x_2 - x_1 = f * \tan(\alpha) - f * \tan(\beta)$$

$$\beta = \arctan(\tan(\alpha) - \frac{x_1 - x_2}{f})$$

Substituting α=0 (reference):

$$\gamma = (amount\ of\ turning) = \beta - \alpha = \arctan(\frac{x_2 - x_1}{f})$$

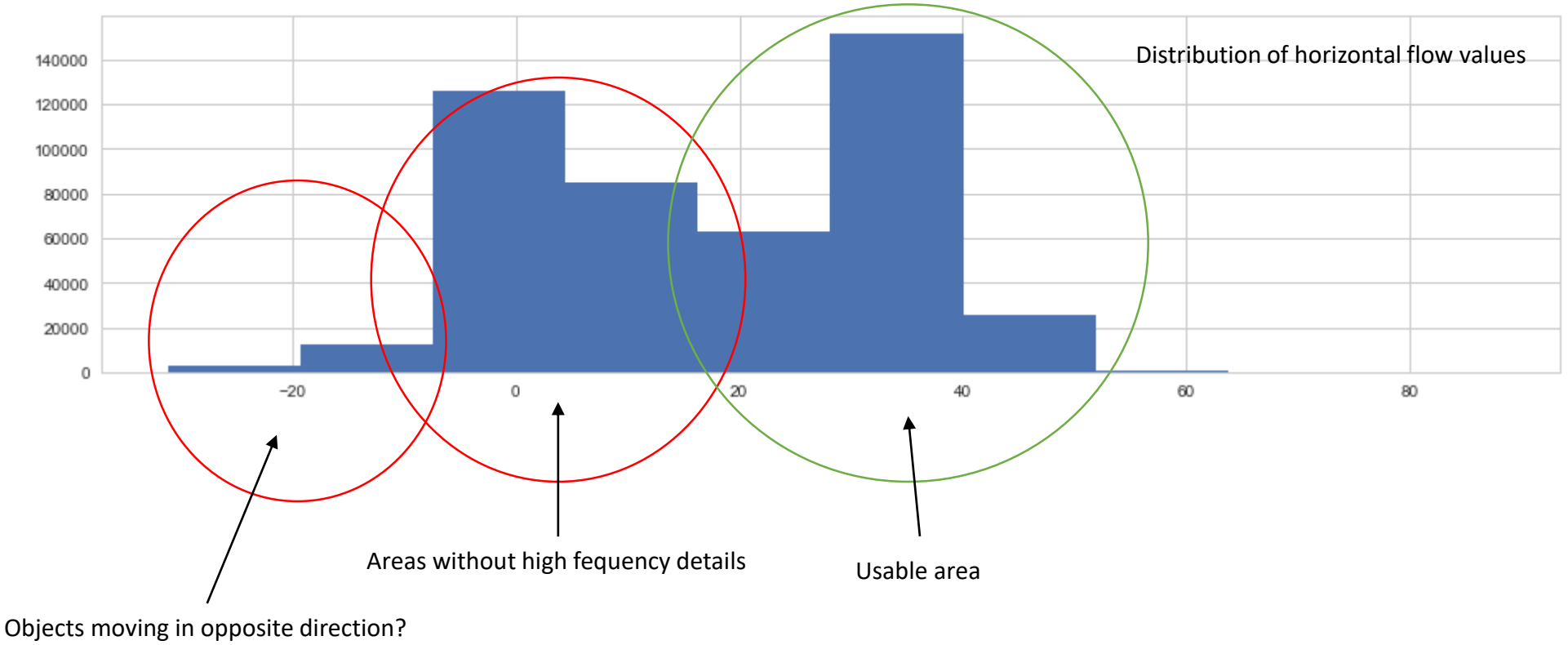
$$\gamma = (amount\ of\ turning) = \beta - \alpha = \arctan(\frac{x_2 - x_1}{f}) = \arctan(\frac{dx_p * sw}{f * pw})$$

$s_w = sensor\ width\ (m)$
p_w = *image width (pixels)*
f = *focal length (m)*
dx_p = *horizontal optical flow in pxels*

Cumulative yaw is maintained for the reference (left) camera:

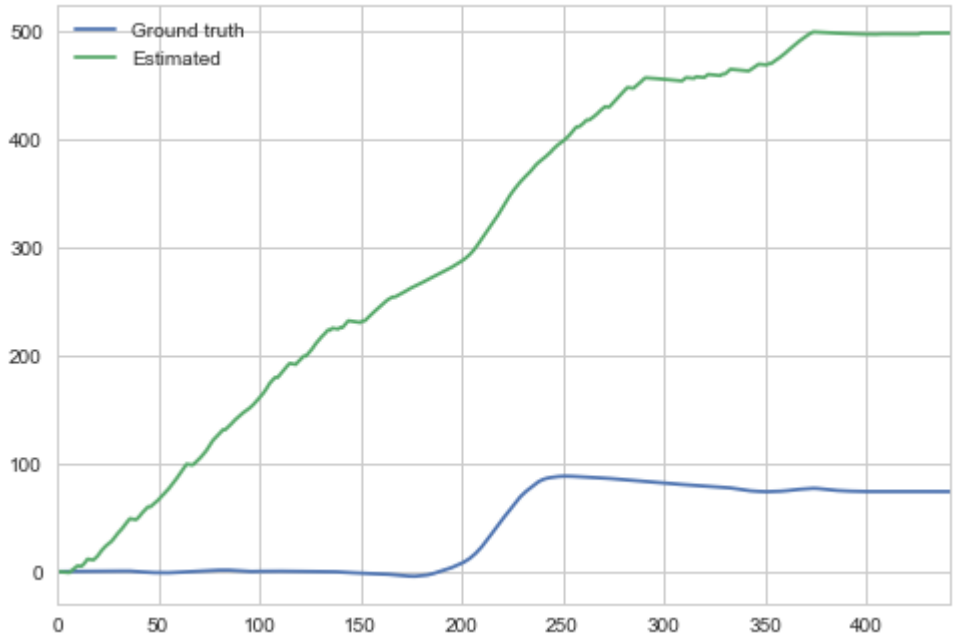
$$yaw_n = \sum_{i=1}^{n} \gamma_i$$

Note: Approximation and optical flow errors are cumulative over time. However, the major effect of error is that the initial reference coordinate system is turning in space around y-axis. As this is completely arbitrary, it doesn't matter much. If we would like to map environment, it would matter more.



Ad hoc algorithm to determine ’average’ background optical flow:

1. Determine main direction (+/-) by calculating mean of flows
2. Drop all flows in the opposite direction
3. Calculate mean value of flows in right direction
4. Drop all flows whose absolute value is below mean value
5. Determine the final value by calculating mean value of remaining flows



Naive ad hoc algorithm is not very good, but a starting point…

Note 1. Lucas-Kanade algorithm is another possibility. It concentrades on areas with texture and high frequency details. Anyway, areas corresponding to patterns should be eliminated.

Note 2. KITTI dataset includes 6 DOF data for moving camera. Lots of data to do research!!!

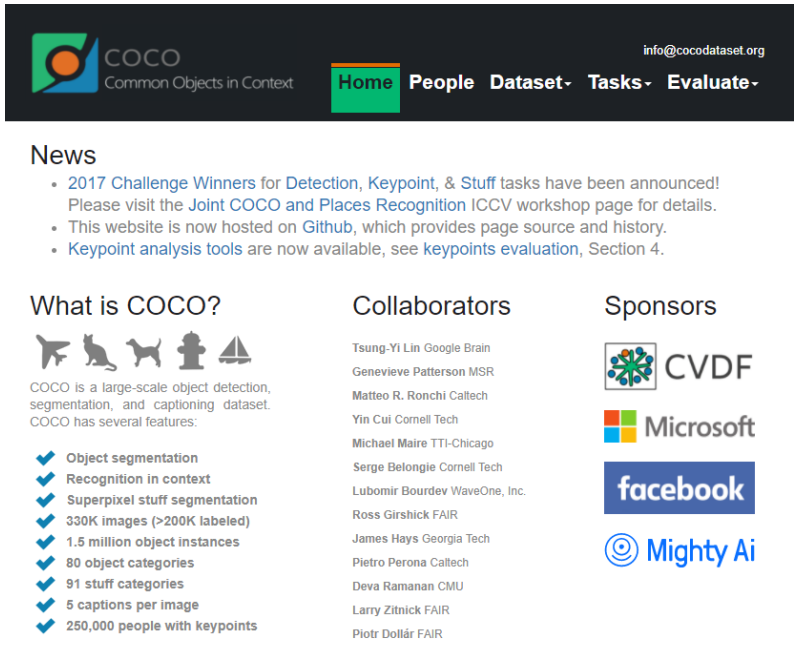
dataformat.txt

```
lat:   latitude of the oxts-unit (deg)
lon:   longitude of the oxts-unit (deg)
alt:   altitude of the oxts-unit (m)
roll:  roll angle (rad),    0 = level, positive = left side up,      range: -pi   .. +pi
pitch: pitch angle (rad),   0 = level, positive = front down,        range: -pi/2 .. +pi/2
yaw:   heading (rad),       0 = east,  positive = counter clockwise, range: -pi   .. +pi
vn:    velocity towards north (m/s)
ve:    velocity towards east (m/s)
vf:    forward velocity, i.e. parallel to earth-surface (m/s)
vl:    leftward velocity, i.e. parallel to earth-surface (m/s)
vu:    upward velocity, i.e. perpendicular to earth-surface (m/s)
ax:    acceleration in x, i.e. in direction of vehicle front (m/s^2)
ay:    acceleration in y, i.e. in direction of vehicle left (m/s^2)
ay:    acceleration in z, i.e. in direction of vehicle top (m/s^2)
af:    forward acceleration (m/s^2)
al:    leftward acceleration (m/s^2)
au:    upward acceleration (m/s^2)
wx:    angular rate around x (rad/s)
wy:    angular rate around y (rad/s)
wz:    angular rate around z (rad/s)
wf:    angular rate around forward axis (rad/s)
wl:    angular rate around leftward axis (rad/s)
wu:    angular rate around upward axis (rad/s)
pos_accuracy:  velocity accuracy (north/east in m)
vel_accuracy:  velocity accuracy (north/east in m/s)
navstat:       navigation status (see navstat_to_string)
numsats:       number of satellites tracked by primary GPS receiver
posmode:       position mode of primary GPS receiver (see gps_mode_to_string)
velmode:       velocity mode of primary GPS receiver (see gps_mode_to_string)
orimode:       orientation mode of primary GPS receiver (see gps_mode_to_string)
```

Research topic!

COCO Dataset

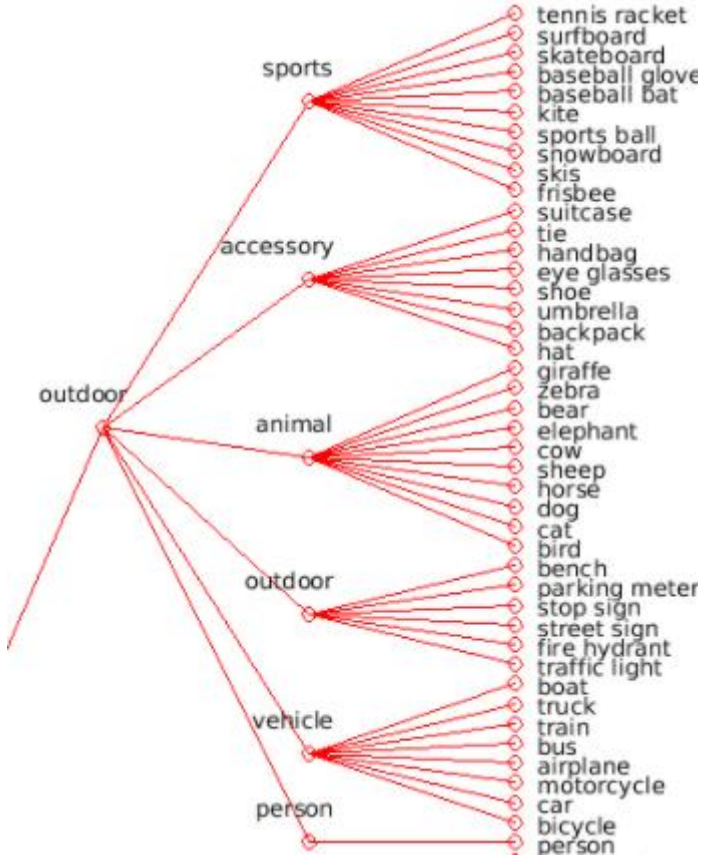
Common Objects in Context

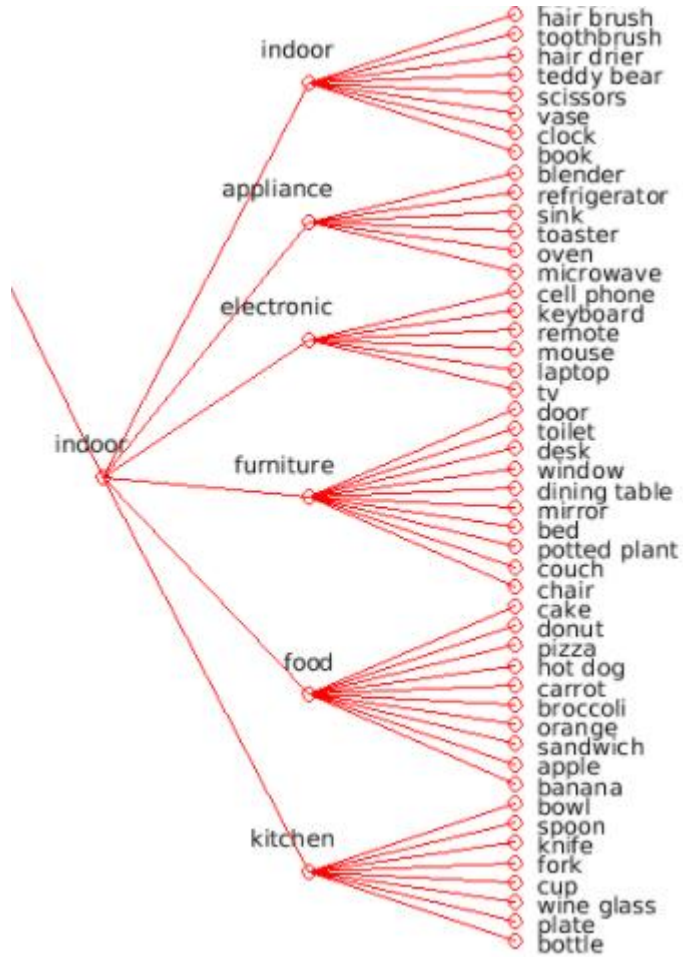


Why?

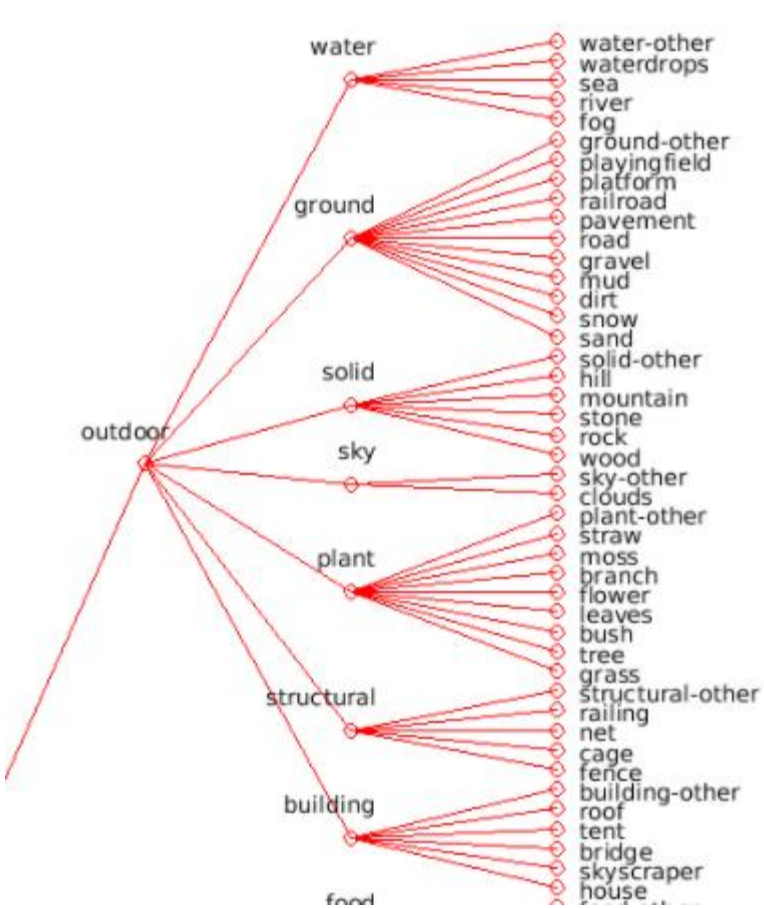
- 171 everyday common categories
- Large amount of images
- Good API
- Emerging support for implementations
- Captions

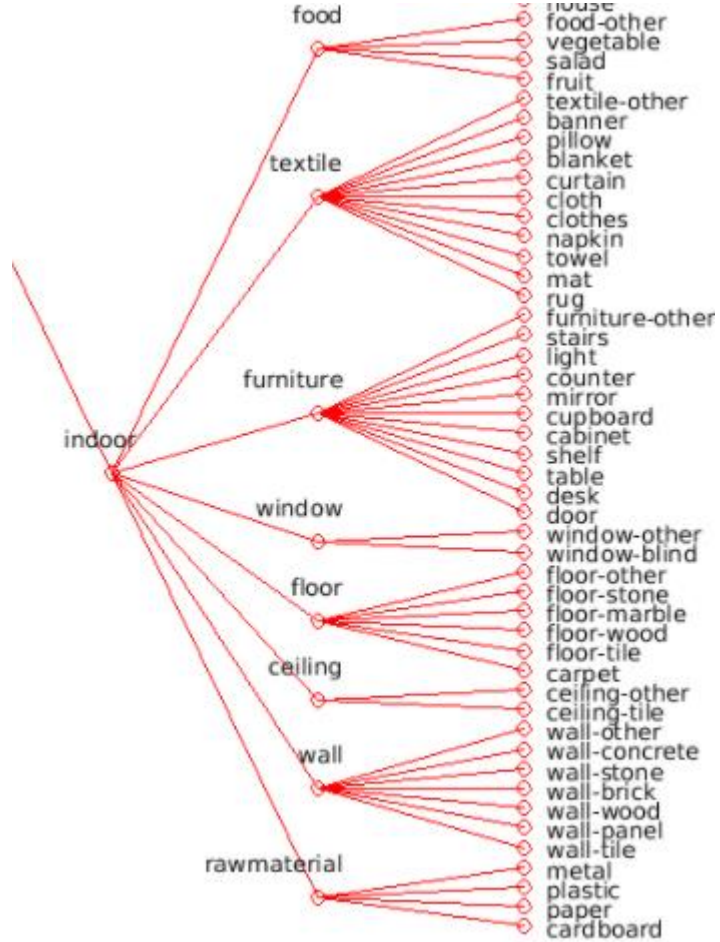
COCO things (=bodies)



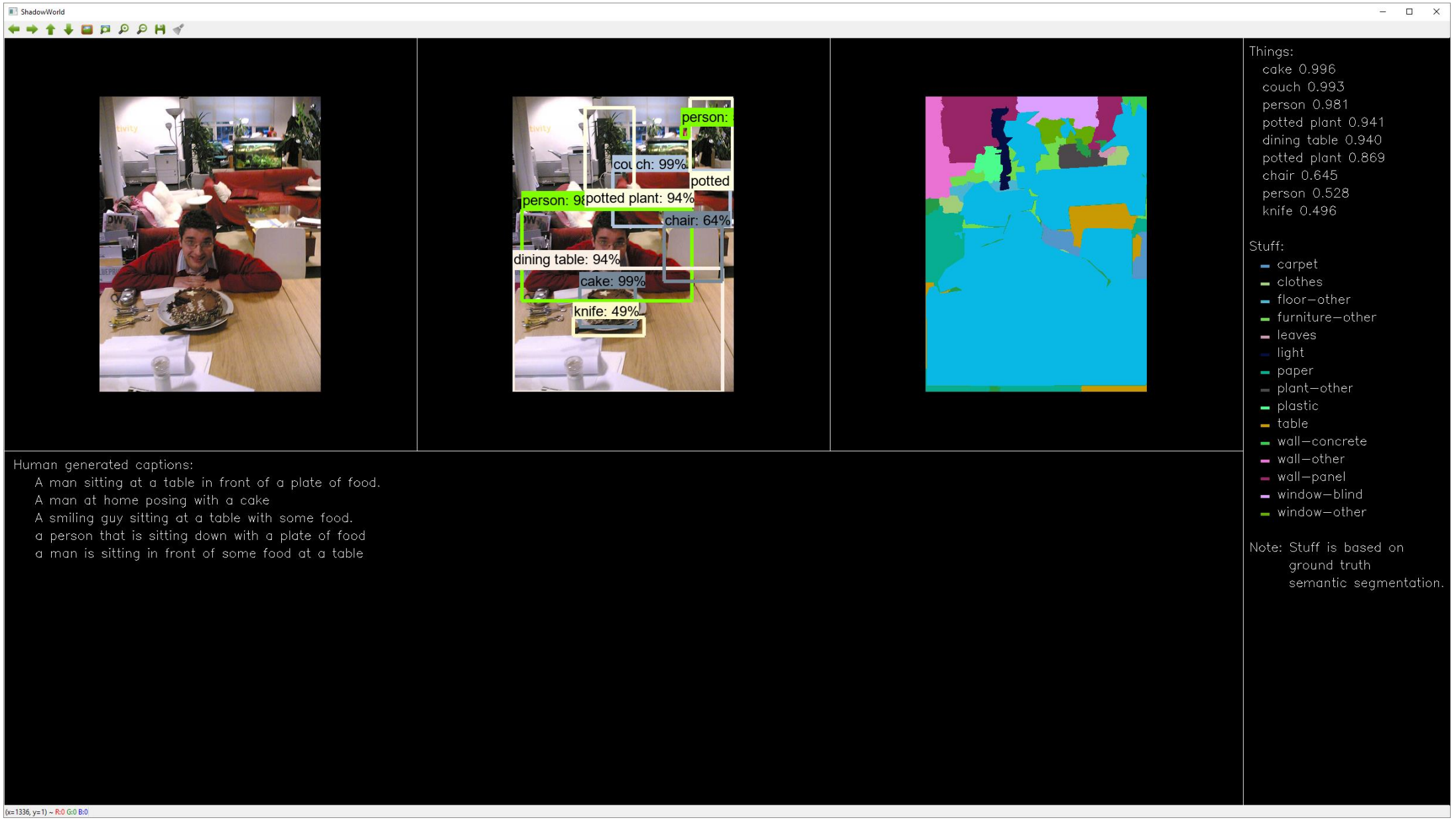


COCO stuff

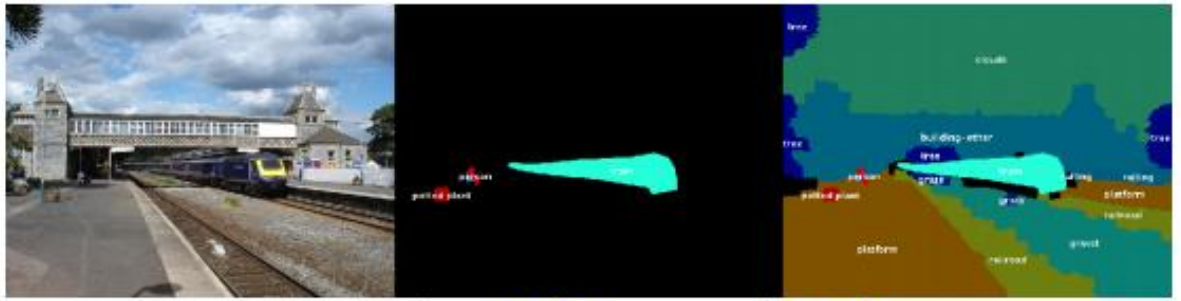

outdoor
water
water-other
waterdrops
sea
river
fog
ground
ground-other
playingfield
platform
railroad
pavement
road
gravel
mud
dirt
snow
sand
solid
solid-other
hill
mountain
stone
rock
wood
sky
sky-other
clouds
plant
plant-other
straw
moss
branch
flower
leaves
bush
tree
grass
structural
structural-other
railing
net
cage
fence
building
building-other
roof
tent
bridge
skyscraper
house
food


indoor
food
food-other
vegetable
salad
fruit
textile
textile-other
banner
pillow
blanket
curtain
cloth
clothes
napkin
towel
mat
rug
furniture
furniture-other
stairs
light
counter
mirror
cupboard
cabinet
shelf
table
desk
door
window
window-other
window-blind
floor
floor-other
floor-stone
floor-marble
floor-wood
floor-tile
carpet
ceiling
ceiling-other
ceiling-tile
wall
wall-other
wall-concrete
wall-stone
wall-brick
wall-wood
wall-panel
wall-tile
rawmaterial
metal
plastic
paper
cardboard

ShadowWorld 4.0 user interface for COCO (with faster rcnn/resnet101 object detection network):



Semantic Segmentation



A large long **train** on a steel **track**.
A blue and yellow transit **train** leaving the **station**.
A **train** crossing beneath a city **bridge** with brick **towers**.
A **train** passing by an over **bridge** with a railway **track** (..).
A **train** is getting ready to leave the train **station**.

Why?

The goal is to give rich view description. For that, we need the bodies (things) and context (stuff).

Unfortenately, there is no TensorFlow implementation for COCO stuff detection. However, there are Caffe implementations. For those implementations, necessary files for converting them into OpenCV DNN could not be found.

Options to continue:

1. Wait until TensorFlow implementation is available.
2. Learn Caffe and export an implementation into TensorFlow (or construct a bridge from Caffe to TensorFlow).
3. Take existing TensorFlow implementation (for example DeepLab/Pascal VOC 2012) and use transfer learning to adapt into COCO.
4. Teach a TensorFlow network from scratch.

I chose to wait until July 2018. If there is no TensorFlow implementation at that time, option 2 is followed. If that fails, option 3 is taken. Option 4 would be very tempting, with a lot to learn. However, there are so many moving parts and hyperparameter adjustments in the process, that this time option 4 should be avoided. If it were followed, it would be a great research topic. It would also require using cloud services, which are not free. Data is available.

Until having a semantic segmentation implementation, the COCO ground truth will be used to develop natural language models.

To summarize:

Things are created by object detection, stuff using semantic segmentation.

Speech Recognition

SpeechRecognition 3.8.1 Python package
+
PocketSphinx

Google API Client Library for Python +
Google Cloud Speech API much more better, but not free.
Will be used in the prototype. Requires network connection.

Next Steps

Discussion

Thank you!

lampola@student.tut.fi
https://github.com/SakariLampola/Thesis