POLITECNICO DI TORINO

Computer, Cinema and Mechatronic commission

Department of Electronics and Telecommunications

Master of Science Degree in
Mechatronic Engineering

Master of Science Degree Thesis

Visual and Tactile Perception to play Jenga with a Robotic Arm

Supervisor	Candidate
Prof. Marcello CHIABERGE	Giulio PUGLIESE

December 2021

Summary

Visual perception and sensory feedback are key elements in any recent robotic system. Human vision and dexterity have been analyzed and adapted to teach the machines how to perform tasks the way people do it. The scope of this Master's Degree thesis is the study of computer vision and tactile perception methodologies to realize a multisensory robotic application able to understand its surroundings and to act with precision on the desired target. More in detail, the goal of this project is to enable the robotic arm e.DO, developed by Comau, to play Jenga tower through a control system based on vision and touch. The constraints given by the rules of the game and the accurate physical interaction with objects constitute a challenging framework for this robotic task.

The analysis of the task leads to the development of a 3D model of the tower and a dataset of images to train a segmentation neural network. From its detections a visual tracker estimates the pose of the specific objects in the camera. Finally, a force sensor is implemented for feedback. The tower is modelled in a synthetic environment with the goal of training an instance segmentation neural network to detect and discriminate each block in the camera image. Using Blender, 2D renderings of the tower's 3D model constitute an image dataset to train the Yolact neural network and allows it to understand the tower's configuration from its pictures. Here the game dictates how the tower is built from blocks and how it changes in time by removing them.

The employed vision system is the position-based visual servoing, composed by three steps adapted from the library of ViSP, an open source visual servoing platform. The first one is post-processing on the segmented image, producing an initial guess pose with P-n-P IPPE of the target block and its adjacent. It initializes a model-based tracker that, given the generic 3D model of an object, finds the real object in the camera frame, learns and detects its keypoints, then follows its movement in subsequent frames; furthermore, the output is the pose (translation and orientation) of the tracked object in each frame. Instead of a single Jenga block, a group of them is tracked for an increased number of visual features, exploiting the tower's staticity and known geometry. The final step is a visual control law computing a twist command (linear and angular velocities) to move the

camera and reach the desired pose for the extraction of the tracked object. This is achieved by mounting a RealSense D435 depth camera on the robotic arm in the so-called eye-in-hand configuration to link the movement of camera to the arm.

Physical interaction to extract a block by pushing it is studied from the properties of wood material, weights of blocks and their relative positions. After an estimate of friction values a force sensor is mounted on top of the robot's end effector to take the real measurements. The sensor's touch feedback allows to abort a potentially disruptive push on the tower's balance, because the extraction of the real blocks is made harder by their irregularities. These avert from the ideal friction behaviour as some blocks are not free to move but they cannot be detected by vision alone.

After developing and testing all software components separately, they are connected through ROS interfaces into a final system which ultimately allows to publish e.DO robot specific commands and actuate its joints. The work presents theory and experiments for each unit and, in conclusion, the obtained results achieved by the complete mechatronic system.

Acknowledgements

Il ringraziamento più sentito va alla mia compagna Benedetta che mi ha pazientemente ascoltato e consigliato superando qualunque tecnicismo.

Ringrazio Luca Marchionna, mio collega in questo stimolante progetto, con cui ho condiviso i successi e gli imprevisti dei trascorsi mesi; il mio relatore prof. Marcello Chiaberge e l'intera comunità del PIC4Ser per aver reso possibile questo lavoro mettendo a disposizione materiale, laboratori e il primato sul braccio robotico; i dottorandi Mauro, Vittorio e Simone per i preziosi consigli dalle loro competenze tecniche. Stefano Pesce per il celere supporto da Comau.

Un grazie a mio fratello Federico per essere il mio mentore e avermi fatto appassionare così tanto nella scienza. Ringrazio l'enorme supporto dei miei genitori dalle cose più semplici alle decisioni più complesse, sempre con grande fiducia.

Infine a tutti i miei amici che meriterebbero ognuno un ringraziamento speciale.

Table of Contents

List of Tables

List of Figures

List of listings

Chapter 1

Introduction

1.1 Jenga, the robot and the task

Jenga is a game where wooden blocks of known and determined solid rectangular shapes are stacked three per floor into a tower of many stories, with alternate orientations. The player must remove one block at a time and if the tower loses balance during the move and other blocks fall, the game ends.

Figure 1.1: Visual example of the tower, courtesy of Hasbro

The Intel Realsense D435 camera can stream both RGB color data and depth information and it is the eye of many robotics applications as it's compact weighting less than 200 g and can be easily carried on board.

Figure 1.2: Reference of Intel Realsense D435 camera

The arm e.Do is a six-axis robot that guarantees six degrees of freedom, enabling its end-effector to reach positions specified in 3-dimensional space with the required 3-dimensional orientation. It's a robot designed for research and learning.

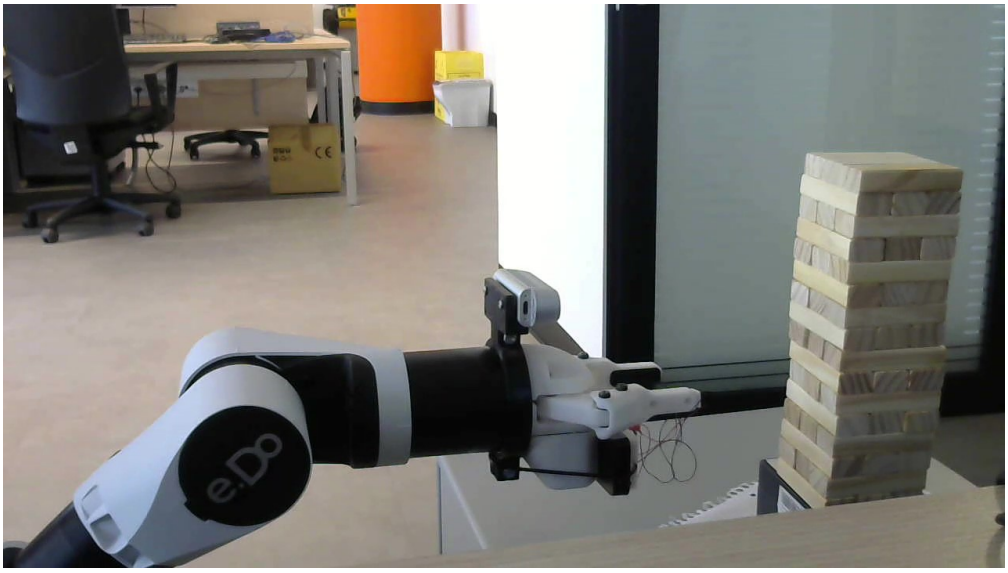

Figure 1.3: Robot e.DO and Jenga tower

1.2 Related work

The most recent and similar work is found in this article from MIT, Boston [1]. They tackled the same task where a 6-axis robot manipulator autonomously plays Jenga; their end goal was to design an artificial intelligence system able to learn the best way to play the game and integrate the physical interactions from mathematical models and high accuracy sensors. In particular a hierarchical architecture was employed to first gather information from physics-based simulation environment and then enforce them with real-world exploration thanks to a camera and a force sensor mounted on the robot wrist. The intent was to reproduce human-like behaviors when learning through stimuli and building beliefs and intuitions. For the computer vision part, an eye-to-hand configuration with Jenga fixed, neural network image segmentation and probabilistic based pose inference. The underlying technologies, instruments and knowledge were on a high level.

The main differences with this thesis are the complete 6-axes force sensor that can measure all directions of forces and torques; the static camera and fixed Jenga tower position; the beliefs probabilistic learning phase; the force-based control of the arm; the metric of numbers of consecutive extractions. In the following, a 1-axis force sensor is used taking inspiration by the main concept of the article, but as a corrective feedback tool rather than an explorative one; the camera and the Jenga are not constrained and in particular the first one is moved by the arm itself in hand-eye configuration; pose estimation and 3D modelling are employed as real-time systems.

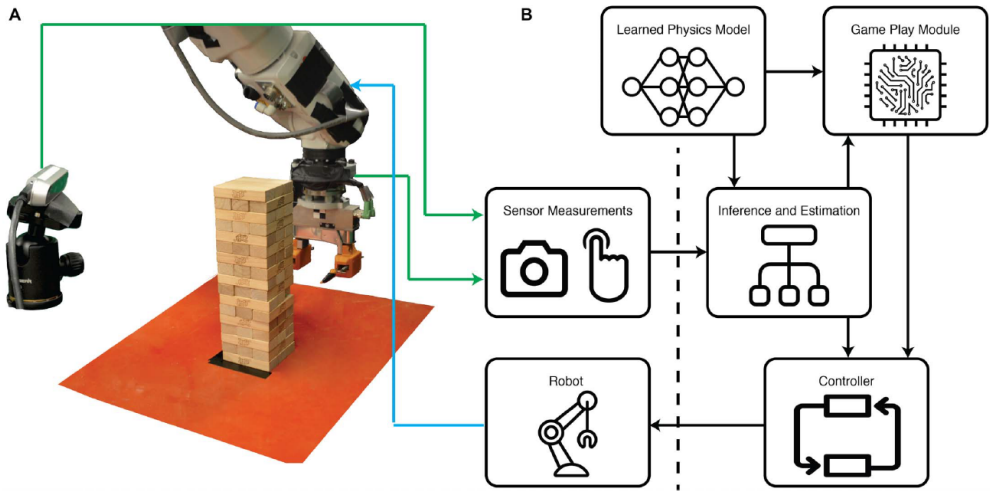

Fig. 1. Robot setup. (A) Physical setup consisting of the robot, Jenga tower, Intel RealSense D415 camera, and ATI Gamma force/torque sensor (mounted at the wrist). (B) Machine intelligence architecture with the learned physics model.

Figure 1.4: Robot, setup and system architecture of the great work of MIT

The other, older, related task is in this IROS conference paper, [2]. Low-cost components and simpler technologies were used, combining classical computer vision algorithm with designing strategic planning to evaluate which block to extract in the best way, from knowledge based on physics simulators. The main concept was to strategize and understand the physics of the game while addressing structural limitations of their devices, as 2 CMOS cameras were used, and a 5-axes manipulator on wheels. The main differences with the following are that the e.DO robot has all the 6 degrees of freedom and the camera is not fixed to a known distance from the tower; the game's physics is not used to strategize extraction; feedback on tower stability is from a force sensor rather than their approach on vision; pose estimation is not assumed from empiric knowledge.

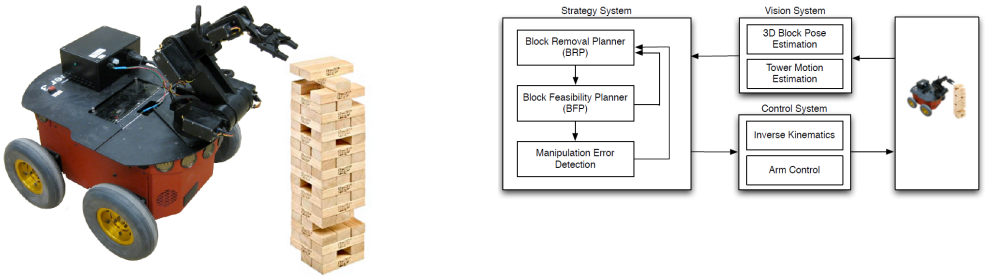

Figure 1.5: Their Pioneer 5-axis robot and control subsystems

1.3 Solution approach

This thesis is placed in the middle ground between the two previously cited works.
The analysis of the task leads to tackle the following challenges:

- From new and unknown tower's configuration, each wood block instance must be detected in the camera image with a high degree of generalization:
  - Any heights of the tower
  - Any missing blocks
  - Any textures of wood
  - Any positions of the tower
- Blocks are very small and require a precision in the millimeters range and within couple of degrees to be correctly approached and extracted:
  - They are well visible only if close to the camera
  - Estimate must be accurate, the more the robot is close to them
- Blocks have irregularities in dimensions that cannot be detected by vision, as they change the friction.
  - Push must be delicate to avoid losing the tower's balance
  - Exploration by touch is needed to understand constrained blocks

Designing the solution of the task, the creation of a CAD model of the tower and a custom dataset of images to train a segmentation neural network is performed. From its initial detections and postprocessing with OpenCV [3], a visual tracker estimates the pose of the specific objects in the camera reference frame, mounted on the manipulator and moving with it, using that information in real-time to approach the block. Finally, a force sensor is implemented for touch feedback.

Chapter 2

Training dataset

2.1 Dataset and COCO format

Neural networks need a large amount of data to learn the required specific information. The data must contain examples in the same data format the net will encounter when used. In the case of segmentation networks the single piece of data is composed of the following:

- An RGB image
- Identifier for wanted classes
- For each pixel, which class it belongs: its identifier
- Each object contained, the pixels that compose it: its mask

A supervised training dataset is then a group of images with detailed description of the pictured objects. COCO, short for Common Objects in COntext, is both the name of a large real photos dataset, 330k images which 200k annotated, figuring 80 ordinary objects and of a data annotation format. The dataset content description, creation methods and purpose is written here [4]; however, the specific object required in this work is not present. In the following, only the format is of interest and a custom dataset is written complying with it using the Object Detection annotation type; a JSON file with dictionary key-values pairs is populated alongside the images. Basic file structure is:
{ "info": {...},
"licenses": [...],
"images": [...],
"categories": [...],
"annotations": [...] }

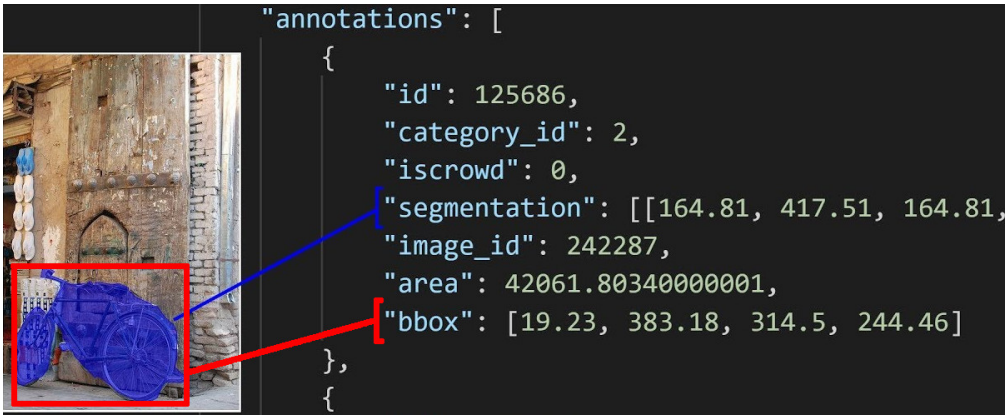

Figure 2.1: COCO annotation example, courtesy of [5]

- INFO: basics like description and date
- LICENSES: if images are under copyright
- IMAGES: list of file names with each a unique ID and pixel dimensions
  {"id": 242287
  "file_name": "0242287.jpg",
  "height": 640,"width": 360,
  "date_captured": "2013–11–14 11:18:45",}
- CATEGORIES: name and ID of each object type contained
  {"supercategory": "person", "id": 1, "name": "person"},
  {"supercategory": "vehicle", "id": 2, "name": "bicycle"}, ...
- ANNOTATIONS (for Object Detection): in each image the area, bounding box corners, segmentation polygon vertices, category label of all the contained objects
  { "id": 125686, "category_id": 58,
  "segmentation": [[239.97, 260.24, 222.04, . . .]],
  "image_id": 242287, "area": 2765.14,
  "bbox": [199.84, 200.46, 77.71, 70.88], }

Alongside the training, a validation dataset must be supplied on which the network evaluates its performance; from just the data point of view this other dataset has the same format.

For generic entities and objects like images of people, cars, animals, grass or sky manual datasets have been created employing thousands of hours of manpower with very good results. However datasets for real custom objects are very hard to find and the more they are specific the harder it becomes.

2.2 3D model and Blender synthetic scene

At the practical level a supervised dataset provides many different looks and possibilities of target object. The result is that hundreds of different images are required with a pixel-precise information; a single person, manually by hand, would take dozens of hours with an accuracy far below the 100% of pixels. Time would be wasted, and the quality of the dataset would be low and biased by the creator's skill and patience. For example, a single Jenga picture would take around 10-20 minutes depending on height and number of blocks. The problem of populating an automatic, realistic, and precise custom dataset can be solved by using computer graphics technology to create a synthetic environment where photorealistic 3D CAD models of objects emulate their real counterparts, and then taking pictures inside this fake world. Blender is used in this work since it offers all is needed:

- Open source: APIs to control all parameters via programming
- Modeling the single blocks as meshes, with their real dimensions
- Building the tower by duplicating and moving the blocks
- Choosing the textures on their faces
- Simulating light conditions, reflections and realistic shadow effects
- Configuring the rendering tool to take correct size pictures
- In the render, full knowledge of each pixels comes from the simulated world.

Blender version 2.92 is used in this work. Wood textures are taken from cgbookcase.com, a free site with high quality photorealistic materials.

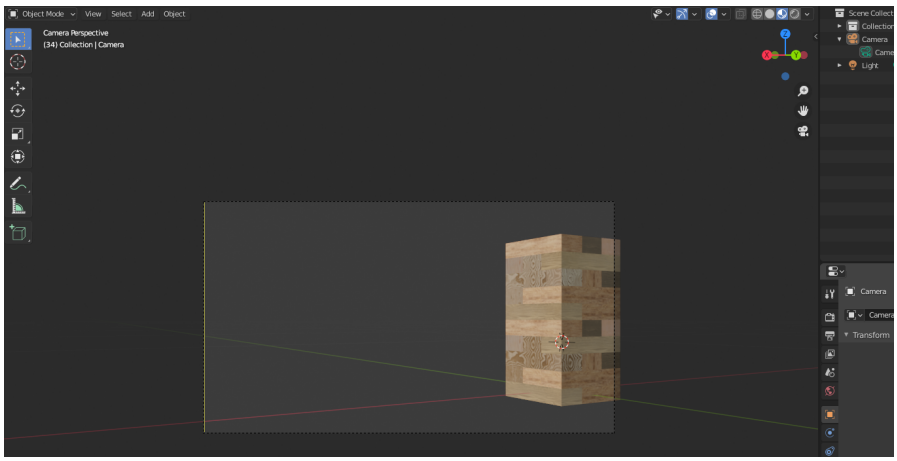

Figure 2.2: Blender scene and camera renderer

2.3 Renderings with BlenderProc

BlenderProc1, presented here [6], is a modular procedural pipeline that interfaces with Blender to use its Python API; it's specifically designed for providing datasets to train convolutional neural networks on synthetic environments.

Its goal is to offer the user all the customization possibilities without learning low level details of how Blender manages its entities, but instead they can just describe their needs and tune the parameters to achieve the desired dataset. The result is then the generation of photorealistic images and accurate annotations.

Figure 2.3: BlenderProc segmented scene from [6]

A feature worth noting is that the authors state generation speed was not a main objective during development and was not optimized, claiming that a dataset of great size is usually produced and changed a small number of times.

The pipeline is configured by choosing the modules and placing them in the right order, and then programming them with the required functions and data. The config file is a YAML file and the user writes specific key-values pairs; these settings will be parsed by a Python script to populate API calls to Blender.

At its core the general single run of the pipeline is shown in the figure:

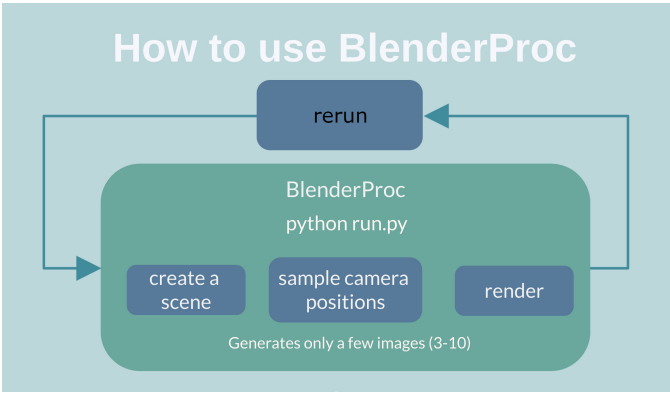

Figure 2.4: BlenderProc rerun scheme

The following module sequence will be explained in the specific use case:

1. Initializing Blender
2. Loading a Blender scene
3. Setting background
4. Randomizing textures
5. Setting lighting
6. Computing camera poses
7. Hiding some objects
8. Rendering a picture for each camera pose
9. Writing annotations for the renderings

The first module is the simplest and it prepares the empty Blender environment. It also sets a global parameter that is the directory path to store the final output, taken from command argument to easily change it from outside the file.

The second module is then the loader: The tower scene designed in the previous section 2.1 is loaded into the software. Yet in this case the path to the file is taken from the command argument: this is useful to process different scenes in different pipeline runs. This is needed to acquire:

- the blocks 3D meshes and their relative positions
- the materials and textures files
- all custom properties

The third module is just used to apply an identificatory value to the scene background. It must be set to 0 correctly write the annotations only for the pixels of the objects. Blender scene background is kept empty as it wouldn't be segmented anyway, there is no need to produce an heavy photorealistic model in computer graphics. A solution is shown in section 4.

The fourth module randomizes the blocks' textures.

```
33 { # Randomize materials, picked from .blend scene
34   "module": "manipulators.EntityManipulator",
35   "config":
36   { "selector":
37     { "provider": "getter.Entity",
```

```
        "conditions":
        { #all meshes objects
          "type": "MESH"
        }
      },
      "cf_randomize_materials":
      { #all blocks
        "randomization_level": 1,
        #to ensure all have material
        "add_to_objects_without_material": True
      }
    }
```

Listing 2.1: Texture randomizer

After loading all the Blender scene the look of the blocks is randomized by applying the textures to all the faces. Using a high enough number of textures the blocks will have different looks in different positions in the image. However it is also needed that blocks with same texture are close to each other to cover the cases of similar-looking blocks in real images, and this is done naturally by the random nature of the sampler. Worth noting is that the tower won't be modified anymore for all the renders of the single run of the pipeline, and this means that multiple runs will have each one randomization.

The fifth module configure the lights' position, type and energy.

```
{ # To sample light positions on a 3D space
  "module": "lighting.LightSampler",
  "config":
  { "lights":
    { "location":
        { #cuboid all around [-0.3,0.3,0.1]
          "provider": "sampler.Uniform3d",
          "max": [-0.2, 0.4, 0.15],
          "min": [-0.4, 0.2, 0.05]
        },
        "type": "POINT",
        "energy": 20
    }
  }
},
{ # To load a light in fixed position
  "module": "lighting.LightLoader",
  "config":
```

```
69    { "lights":
70      { "location": [0.3, 0.3, 0.07],
71          "type": "POINT",
72          "energy": 5
73      }
74    }
75 },
```

Listing 2.2: Light sampler

To ensure a good simulation of the light that could shine on the real tower, two points of light are employed:

- one is brighter and its source position is random inside a cuboid of dimensions [0.1,0.1,0.05]m around the point [-0.3,0.3,0.1]m
- the other is dimmer and it is fixed in position [0.3, 0.3, 0.07]m

The combination of the two have a satisfactory effect on the wood material of the blocks; this is critical to provide a dataset that is robust to different light conditions and that contains realistic colors on which the neural network can learn generalized information. The most practical effect of this is that when blocks are missing, shadows are cast inside the holes that remain in the tower and the network must see many of these configurations to keep the association between illuminated and shaded blocks.

The sixth module is the most important and it samples the positions of the camera. Five different configurations of this module are used to have many random sources of locations.

```
76 { # Camera position for each render
77   "module": "camera.CameraSampler",
78   "config":
79   { #random all around the tower
80     "cam_poses":
81     { #number of poses
82       "number_of_samples": 10,
83       "proximity_checks":
84       { # avoid camera too close to ojects
85         "min": 0.2
86       },
87       "check_pose_novelty_translation": True,
88       "location":
89       { #cuboid all around [0,0,0.125]
90         "provider": "sampler.Uniform3d",
```

```
            "max": [0.4, 0.4, 0.2],
            "min": [-0.4, -0.4, 0.05]
        },
        "rotation":
        { "format": "look_at",
          "value":
          { # rotates to mean position
            "provider": "getter.POI"
          },
        }
    }
  }
},
```

Listing 2.3: Camera type 1 sampler

Here showing the first type with the description of the values in common with the others.

- Number_of_samples: is the amount of locations to compute. Since one picture is rendered from each camera location, this is also the number of renderings produced by a single run of the pipeline
- Proximity_checks: the camera will never be placed closer than 0.2 meters from any of the blocks, to avoid partial images.
- Rotation, look_at, Point Of Interest: camera orientation will be toward the mean value of the positions of all the blocks, to keep them in sight regardless of the camera positions.
- This first type of camera positions is the least constrained as it can be any value between [0.4, 0.4, 0.2]m and [-0.4, -0.4, 0.05]m, forming a cuboid of shapes [0.4, 0.4, 0.125]m around the point [0,0,0.125]m. With the proximity check value set, every point around the tower except closer than 0.2 m is eligible.

The second, third and fourth type of camera locations are very similar as they all are placed looking at the tower's corner, changing only which one. This is important to ensure that a good portion of the dataset contains pictures that shows two faces of the tower, that is the most useful and informative condition as all the blocks can be seen, while looking at one face only of the tower the other blocks are not shown in the image. Constraining the randomness in this way is mandatory as the first type of camera positions cannot be trusted to produce enough specific images even for a very high number of random samples. The three corners differ

only for the previously chosen position of the light: the illumination and shade is unique for each of the tower's faces.

```
106             "proximity_checks":
107             { # avoid camera too close to ojects
108               "min": 0.15
109             },
110             "check_pose_novelty_translation": True,
111             "location":
112             { #points on a circular arc
113               "provider": "sampler.Disk",
114               "sample_from": "sector",
115               "center": [0,0,0.07],
116               "radius": 0.45,
117               "start_angle": 30,
118               "end_angle": 60
119             },
120             "rotation":
121             { "format": "look_at",
122               "inplane_rot":
123               { "provider": "sampler.Value",
124                 "type": "float",
125                 "min": -0.07,
126                 "max": 0.07
127               },
```

Listing 2.4: Camera type 2-3-4 samplers

Proximity_checks: is slightly less, 0.15 m here

Check_pose_novelty_translation: due to the small space of random locations, similar positions could be sampled. This ensures that samples are always unique among the 10, without useless repetitions

Location: points sampled inside a section of a disk with center in [0,0,0.07]m, radius = 0.45m and start and end angle for the specific corner.
Face left in light, face right in light: start=30° and end=60°
Face left in shade, face right in light: start=120° and end=150°
Face left in light, face right in shade: start=-150° and end=-120°

Inplane_rot: camera will be randomly rotated by a max of 0.07 radians on its z axis to simulate a tilted camera.

The seventh module is the one that deals with missing blocks.

Figure 2.5: Camera front upper render

Figure 2.6: Tilted view

```
{ # Randomly hide and duplicate render
  "module": "object.HideModule",
  "config":
  { "selector":
    { "provider": "getter.Entity",
      "random_samples":
      { #random number of objects, from 2 to 9
        "provider": "sampler.Value",
        "type": "int",
        "min": 2,
        "max": 9,
      },
      "conditions":
      { #all meshes objects
        "type": "MESH"
      }
    },
  }
},
```

Listing 2.5: Hiding module

To generate images without some blocks the hiding module is employed. This takes

a number of random blocks and disable them for the rendering; the number itself is a random value between 2 and 9 blocks. This module doubles the amount of renderings for each camera pose: two images are taken where one shows all the blocks and the other doesn't. This brings the number to 20 renderings per run. Like in the case of the texture randomizer module, the blocks to be hidden are the same for all the single run of the pipeline, requiring multiple runs to change number and entities.

Figure 2.7: Complete tower

Figure 2.8: Partial tower

The eighth modules are the renderers.

```
{ # Render in colors
  "module": "renderer.RgbRenderer",
  "config":
  { "output_key": "colors",
    "transparent_background": True
  }
},
{ # Maps the segmentation for instance and (single) class
  "module": "renderer.SegMapRenderer",
  "config":
  { "map_by": ["instance", "class"]
  }
},
```

Listing 2.6: Render modules

Images are taken in RGB color, without the background, in the shape 512x512. Blender have all the information to associate pixels in the image with their corresponding 3D object in the scene since all is simulated and the camera pose is known. Pixels belonging to the objects are treated in an instance segmentation way where each of them is discriminated from each other and associated with its class. Hidden here is the use of the custom property "class_id" to define the mapping. As explained earlier, all blocks have a class=1 property in the Blender scene.

The ninth and final module is the COCO annotation writer.

```
{ # Producer of annotations in COCO style
  "module": "writer.CocoAnnotationsWriter",
  "config":
  { #RLE is default instead
    "mask_encoding_format": "polygon",
    #appending new annotation to existing
    "append_to_existing_output": True
  }
}
```

Listing 2.7: Annotation module

Describing the renders with full annotation and also adding it to the end of the already existing one. Since the pipeline runs many times, images will add up and annotations must keep their COCO format with new data. The chosen format is COCO for compatibility with the neural network used. The two practical effects are that new images are named and numbered following the last iteration's count and that their content description is appended at the end of the lists in the annotation file with no repercussions on images already described.

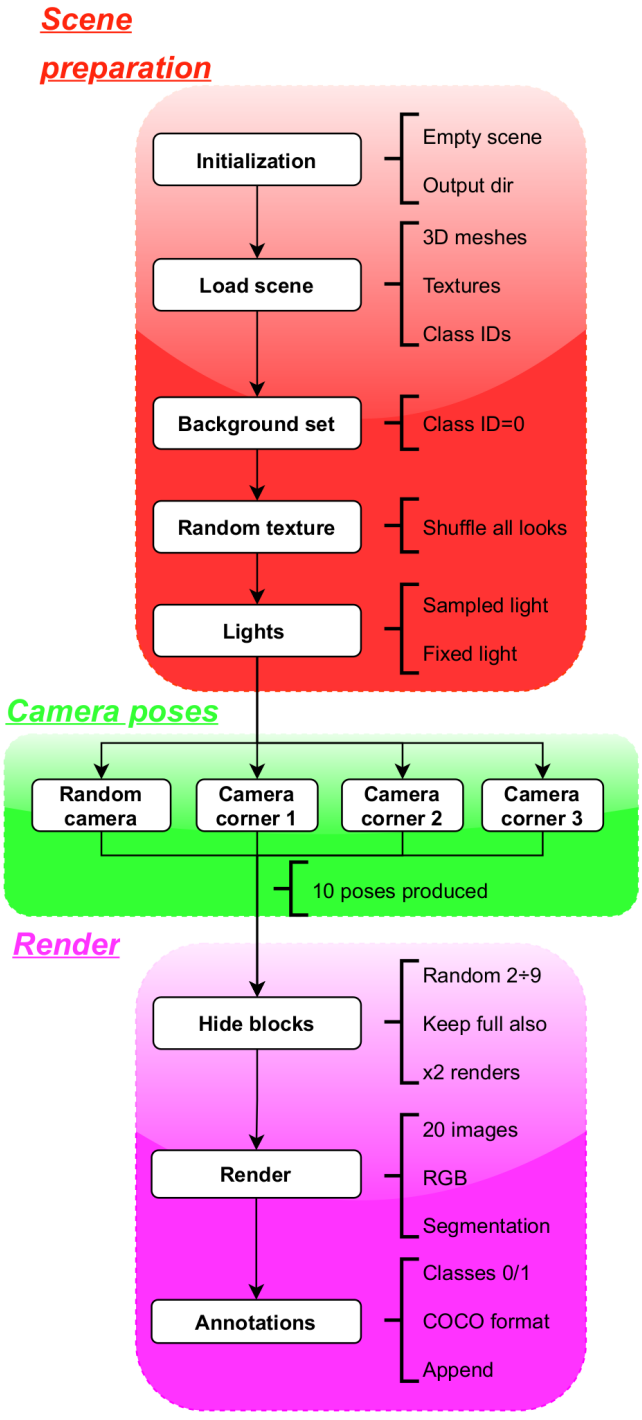

Figure 2.9: BlenderProc pipeline scheme

2.4 Dataset creation and COCO format viewer

As the previous section described, a single run of the pipeline produces 20 images and it appends the output to any previous run. In order to produce a dataset of 800 training images the pipeline is executed 40 times; in the same way a validation dataset of 80 images requires other 4 calls. In the terminal, a single run is called as python BlenderProc/run.py <path/config.yaml> <path/scene.blend> <path/outputDir> An interactive Python script is written to make all the calls to the BlenderProc command with the following benefits:

- To store and change output directory path and Blender scene path
- Command in a for loop with programmable length
- Computation of the execution time
- To show and save the verbose terminal output
- Apply additional commands needed

```
nTimes_around=10   #times to call command

time0=time.perf_counter()    #to count elapsed time

#20 images each call
#config.yaml: camera type 1 all around the tower
#blender scene path
#output folder of training data, appends annotations at each run
for i in range(nTimes_around):
  !python ~/BlenderProc/run.py \
  ~/BlenderProc/configs/config.yaml \
  ~/Blender/jenga_scenes/jenga_scene.blend \
  ~/BlenderProc/outputs/OutputTrain

end0=time.perf_counter()

print("\nElapsed time config0: ",end0-time0)
```

Listing 2.8: Loop command call

To increase the realism of the images in a simple way the background is added from 32 free stock real photos from categories like “indoor” and “office”, to generalize the data and avoid artificial bias that the network could produce during the training on images with only a tower and transparent background. The script used is an utility offered by BlenderProc by giving the correct parameters, resulting in 2.11.

Figure 2.10: Transparent background

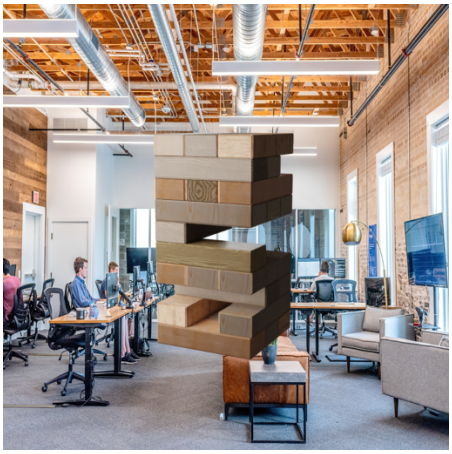

Figure 2.11: Filled background

Worth noting is that the network isn't interested in precisely understanding the background's content but due to its architecture is able to detect the tower's blocks and to put every other pixel in a "don't care" class with ID=0. This makes it more robust to the real environment where real pictures will be taken.

Camera type	Images	Time[s]	Memory[MB]
1	200	345	66
2	200	368	69
3	200	370	69
4	200	367	69
TOTAL:	800	1450	273

Table 2.1: Training dataset info

Camera type	Images	Time[s]	Memory[MB]
1	20	33	6
2	20	34	7
3	20	38	6
4	20	38	6
TOTAL:	80	143	25

Table 2.2: Validation dataset info

Finally using a script it is possible to overlap the annotations on top of the images to check if everything is working; since the description of each image is not human readable a qualitative way by showing and saving pictures is chosen. Courtesy of [7] in ImmersiveLimit.com.

Figure 2.12: Graphical view produced by the annotations of the image

Figure 2.12 produced by this source annotations file of thousands of text lines.

```
"images":
[    # ...other images...
     {"id": 16,
     "file_name": "rgb_0016.png",
     "width": 512,
     "height": 512,
     } # ...other images...
],
"annotations":
[    # ...other segmentations...
     {"id": 0,
     "image_id": 16,
     "category_id": 1,
     "area": 1126,
     "bbox": [152, 335, 41, 39],
     "segmentation": [[192.0, 373.5, 153.0, 362.5,
    151.5, 335.0, 191.0, 344.5, 192.0, 373.5]],
     "width": 512
     "height": 512
     },
     # ...
     {"id": 21,
     "image_id": 16,
     "category_id": 1,
     "area": 1273,
     "bbox": [192, 345, 44, 41],
     "segmentation": [[235.0, 385.5, 193.0, 373.5,
    191.5, 345.0, 233.0, 354.5, 235.5, 360.0, 235.0,
    385.5]],
     "width": 512,
     "height": 512
     } # ...other segmentations...
]
```

Listing 2.9: Annotation content source of the visual information

Chapter 3

Instance segmentation neural network

3.1 Yolact network

The problem of computing the 6D-pose of an object in the camera frame (its 3D position and its 3D rotation with respect to the camera reference frame) is of great importance in robotics applications and like in this case similar to manipulator's picking. It becomes much more challenging in cluttered scenes when the target objects are occluded by others, since the lost visual information can cause wrong pose estimations. Furthermore, rich textures on the surface as well as asymmetric geometries help finding correct values and reject wrong and impossible ones. Finally, the speed of the computation is usually very low as many iterations are needed to form a consensus on the pose value.

For all this reasons, neural networks with end-to-end pose estimation are not considered in this work. The use-case and the specific properties of the real objects match all the elements that would make it refrain from their training and inferences:

- Wood blocks have great symmetries
- Wood blocks are almost textureless, with few good visual features
- High number of blocks are in the same scene
- Inside the tower, blocks are heavily occluded one from the other and only few faces are visible
- Fast estimation is required

The design of the solution separates the pose estimation into a later stage, in 4 and making this part focus on the detection of many similar object instances

in one image, working at the single pixel level. The two main types of image segmentation networks are the semantic and the instance. In this work the second type is considered: a high number of objects belonging to the same class must be separately detected one from the other, ignoring the background; the first type instead would deal with different object classes.

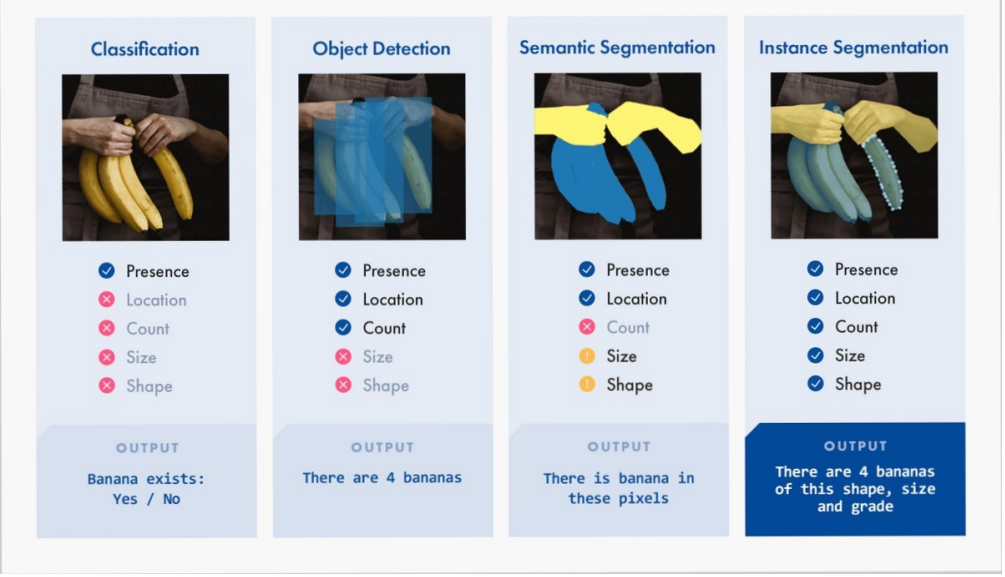

Figure 3.1: Types of segmentation, only the instance one can precisely detect close objects. Courtesy of [8]

In particular, a Yolact instance segmentation neural network from GitHub repository [9] is used here because of the following capabilities:

- Fully convolutional architecture
- Single shot detection
- Real-time computation (for inference on a GPU)
- Fast computation (for inference on a CPU)
- Precise object masks localization
- Ease of training and use for inference

3.2 Training and validation results

The network is an end-to-end system, with provided Python scripts to train the network using PyTorch that are leveraged to optimize the weights on the dataset built in 2. The supervised annotations are required to be in COCO format; some indications from this tutorial [7]. Training images are used to modify the weights based on the annotations that for each pixel provide the accurate information of its class ID. Validation, instead, is used to check if the new weights' values increase or decrease the quality of the inference on new images and their content is not memorized by the network. The weights do not start from default values but a backbone is used to perform transfer learning. These are pretrained weights that are generic enough to extract basic information from an image without giving a constrained meaning, and then it is used as a first filter whose outputs jumpstart the actual learning. Configuration for the training starts with the following:

- Folder with training images and folder with validation images
- Annotation of training and annotation of validation
- Class names and ids association, including the background
- Which backbone and which pretrained weights

GPU for training	Nvidia 2080
Parallel architecture	CUDA available
Image size	512x512
Backbone type	Resnet 101
Backbone weights	resnet101_reducedfc.pth
Training size	800
Validation size	80

Table 3.1: Base info for training

Important parameters of the configuration are the constant fixed hyperparameters such as maximum iterations to reach and the learning rate, momentum and decay as they set how the weight values change during the training as their gradient descend. Learning rate is the amplitude of the change in the gradient. The bigger, the more the weights can move from the previous value. Momentum dictate how important is the previous value, and it is a ration between 0 and 1, with 1 meaning that the weight should not change. Decay is used to force a change and avoid being constrained on local minima.

$$dw' = momentum * dw - lr * (grad + decay * w) \tag{3.1}$$

As the training progresses, the chance of overfitting increases. This happens because the network in the long run memorizes the training images and specialize the weights to be perfect for those precise pixel content, but with no guarantee that it remains generalized enough to work on new, unknown images with different content. This is avoided by reducing the learning rate by steps after a tuned number of iterations, so to slow the convergence of the weights to the real optimal value that would be perfect only for the training images; the reduction is done exponentially by dividing the learning rate at each step. Depending on the application, the user must decide the maximum number of detections to consider inside an image to be evaluated for the validation. Finally, the batch size is the group of images that are evaluated together, with the goal of speeding up and reducing the specialization to the single case, avoid biases. To note is that for 512x512 pixel images, each batch size increase takes 1.5 GB of video card memory, VRAM, so the maximum is constrained by the hardware.

Number of classes	1+1
Max iterations	280000
Starting learning rate	10^{-3}
Momentum	0.9
Decay	$5 * 10^{-4}$
γ	0.1
Learning rate steps	(150000, 200000, 250000)
Maximum detections	100
Batch size	6

Table 3.2: Hyperparameters configuration for training

The evaluation of the training quality is done comparing the bounding box and the pixel masks with the ground truth; this means that running the network on the validation images, the result is overlapped with the real precise annotations. The mean average precision is the metric for each crescent correct percentage overlap: it's the ratio between bbox and masks overlapped with that percentage accuracy over the total number, and it is also called IoU (Intersection over Union).

The training is stopped with the following final precision on the validation dataset. Running time: 14 hours up to 280000 iterations.

	all	.50	.55	.60	.65	.70	.75	.80	.85	.90	.95
box	86.38	96.86	96.75	96.71	96.70	95.54	95.27	94.01	92.33	77.23	22.44
mask	76.86	96.89	96.87	96.85	96.78	95.79	95.37	89.92	70.66	29.02	0.47

Figure 3.2: Mean average precision (mAP) values for each IoU

3.3 Evaluation of masks on real images

Evaluation on the training GPU is in real-time with 30 fps. Due to the lack of a dedicated GPU on the laptop that runs the inference, the network is then used through the CPU only, with the only difference being the computation speed as the parallel architecture cannot be used. Quality of the inference is the same regardless, as it only depends on the weights' values. The code to do it is taken from this forked repository [10] elaborated from the original.

Confidence threshold is always 0.5, below that percentage the mask is rejected.

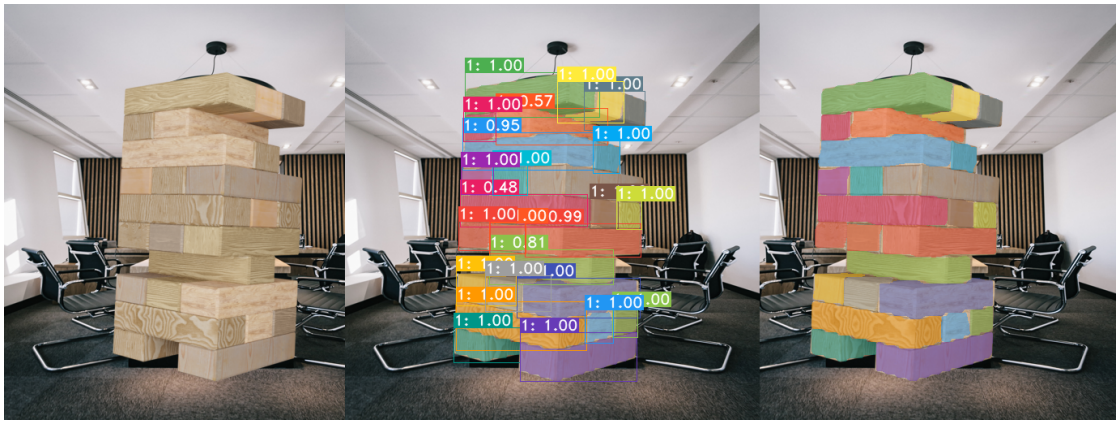

Figure 3.3: Inference on validation dataset
High precision since synthetic model like training images

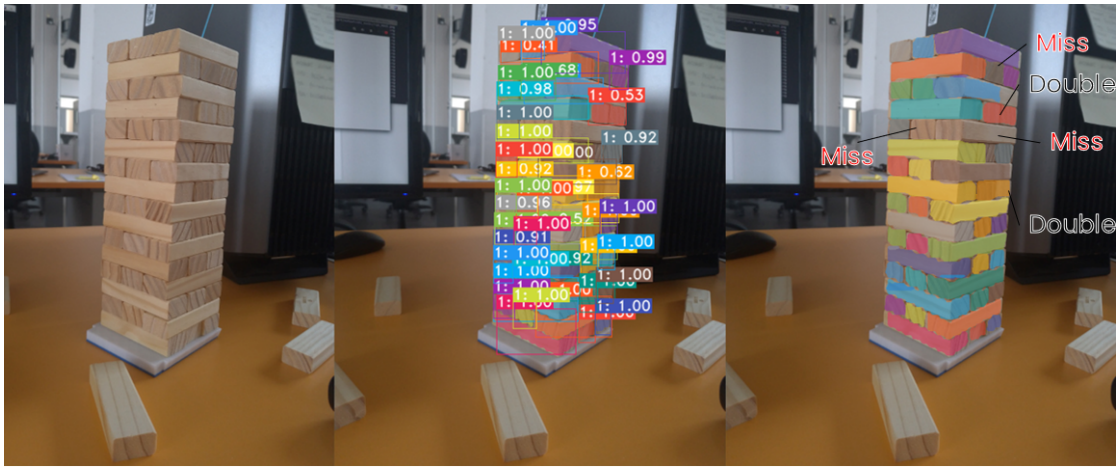

Figure 3.4: Inference on real picture
3 misses, 2 aggregations (50% confidence), 43 exact (>90% confidence)

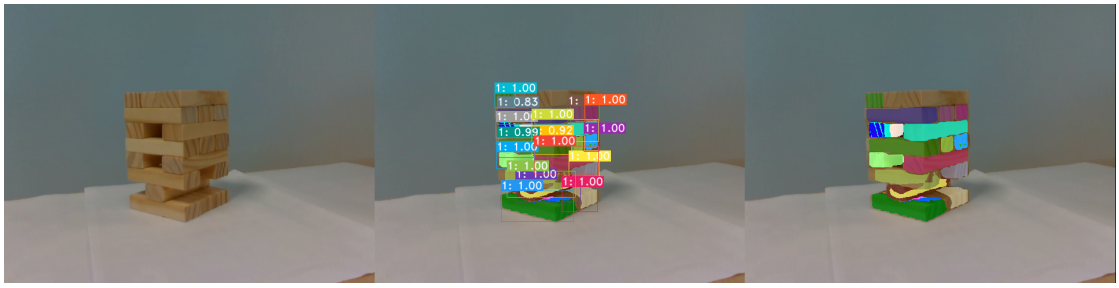

Figure 3.5: Missing blocks produce worse masks sometimes like in this case.

Most of the blocks are found and very few false positives are detected, but there is the presence of a high number of blocks that are ignored. Most of the pixels are correctly associated with their masks. Even missing blocks do not alter too much the quality and are mostly correct.

3.4 Tower structure from inference

Through the use of postprocessing utilities provided by the network's code base, each single mask of the detected blocks is computed with corresponding confidence value. Due to being only one class, the masks refer to the same object type. Then these images are elaborated with OpenCV and Python scripts.

The resulting masks have an undulated border caused by the segmentation effort to discriminate between one block and the others that are very similar-looking. A polygonal approximation is performed to reduce the number of vertices of the mask and to smooth the sides, using OpenCV implementation of the Douglas-Pecker algorithm, with a result shown in figure. Here the epsilon value to specify the distance between the original edge and the approximation is different for front faces and for side blocks: the second are much less approximated to avoid artifacts. Info at [11]. This has the added benefit to detect simple geometric shapes in the image, in particular the small front side of a block when it is fully occluded by the adjacent. A distinction between the small front face and the bigger/more complex shapes is made by comparing the number of vertices and the enclosing area of the smoothed mask.

Small front face: Exactly 4 vertices AND area less than average

Side block: More than 4 vertices

Corner detection is employed to find the vertices of the smooth shapes in the image from the OpenCV function of Harris corners, as to compute points with high spatial intensity derivative, from [12]. Knowing the distinction between the masks,

special meaning is given to the corners of the small front faces: they are exactly the vertices of the Jenga block and are associated to the 3D model. Instead, the detections fails if shapes are not approximated beforehand, as in figure 3.7.

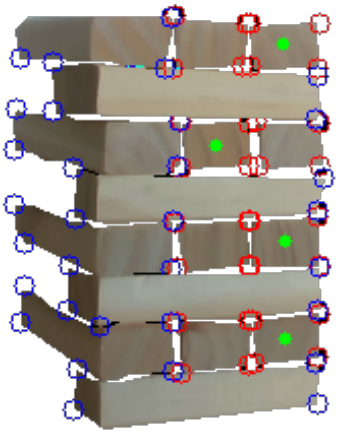

Figure 3.6: Corners of the small faces are highlighted in red, while those of the bigger side blocks are in blue and not guaranteed to be 4.
Shapes are approximated to smooth polygons.

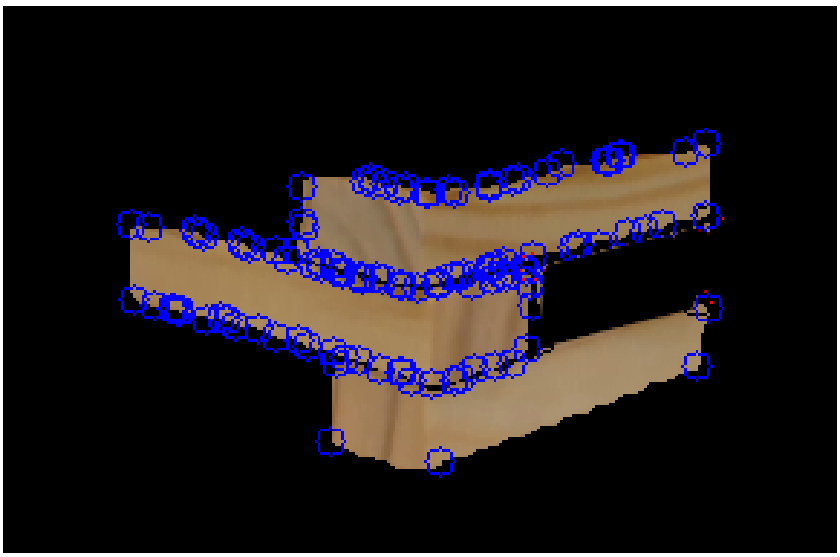

Figure 3.7: Applying the corner detection on non approximated masks, wrong values are found.

Starting from the small front faces, all their adjacent blocks can be found with

a search in the image. The four corners give the slope and the length of segments that, drawn with the origin on the mask centroid, find which mask is:

- Up from positive vertical slope, in white
- Down from negative vertical slope, in blue
- Right from positive horizontal slope, in red
- Left from negative horizontal slope, in green
- Center the target block

The lines will detect if they land on another block's mask and identify which is. Right and left are then extended to double length, to search possible third block inside the same floor. The result is to know which blocks compose a group with the target block.

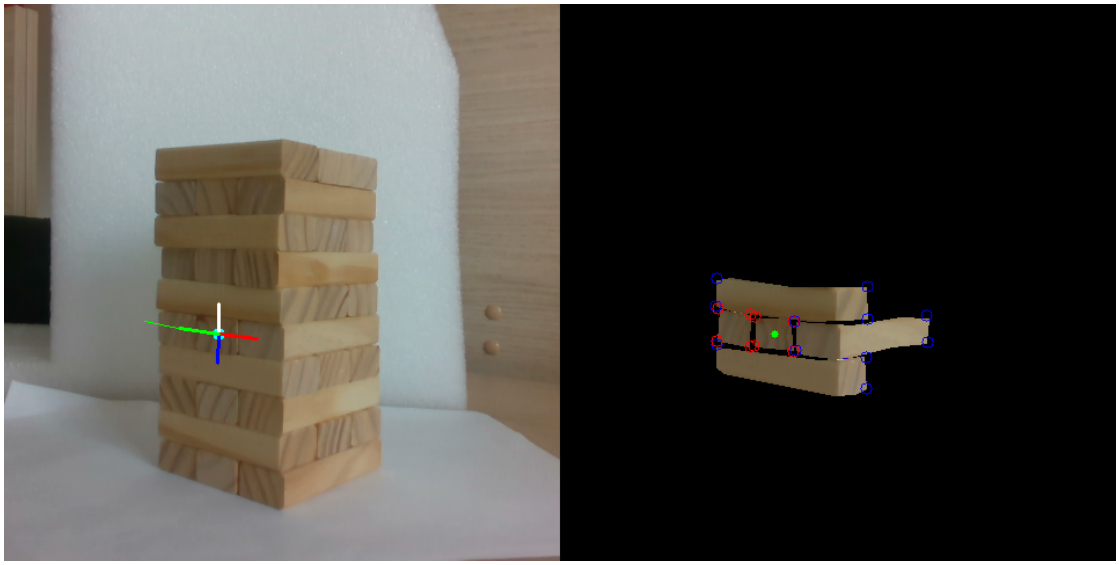

Figure 3.8: Search showed by drawing lines, with slope and length

Worth noting is that all this system cannot work with missing blocks as the segmentation outputs the entire mask of the block and not only its small front face, making it fall in either the side block category or in the front face category.

Chapter 4

ViSP Model-based tracking and pose estimation

4.1 Object pose estimation in camera frame

The task of finding the correspondence between 2D pixel points in an image and their 3D coordinates in a specific frame has been tackled in many ways originating specific solutions with trade-offs between accuracy, computation effort, and assumptions on input data. All of them stem from the perspective projection camera model on an ideal pinhole camera and the subsequent addition of lesser-order approximations to increase its similarity with real cameras, such as various degrees of distortion coefficients to rectify images taken by small of fish-eye lenses, or corrective actions to approximate errors in the projection. The goal is in any case to put in relation the pixels in the image, projected from the scene, with the real measurement unit, for example meters, given the knowledge of the distance from the camera.

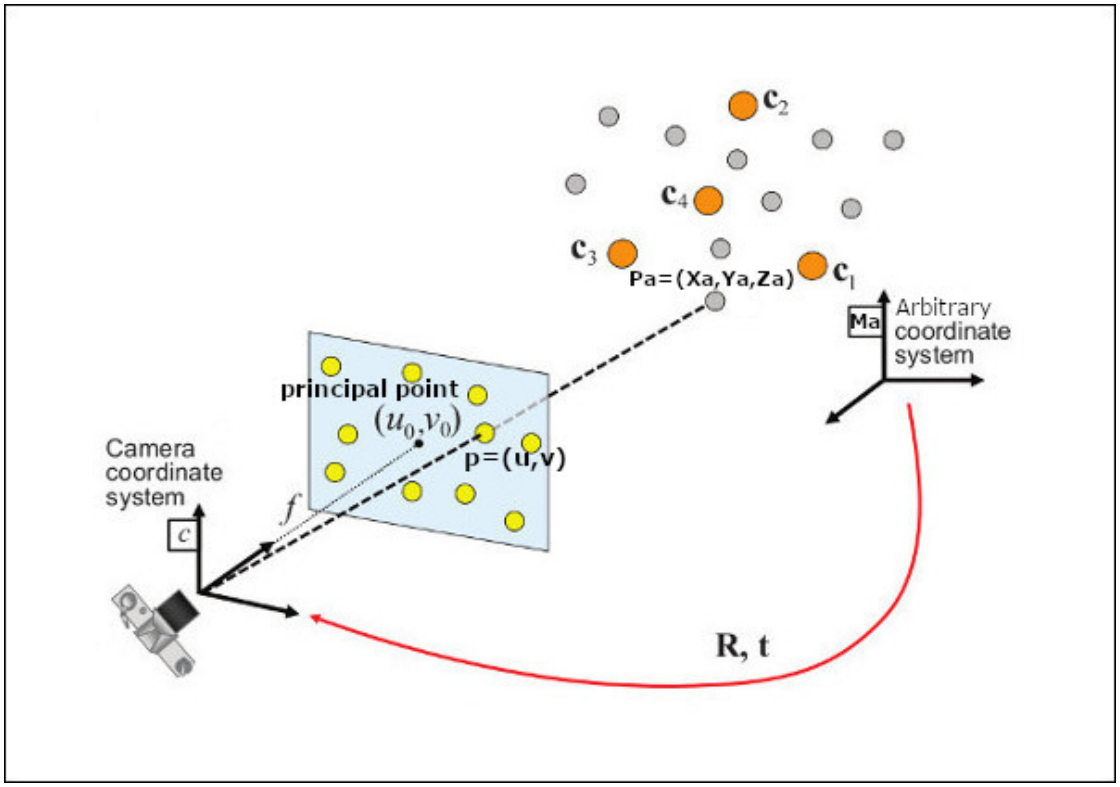

Figure 4.1: Perspective projection pinhole camera model

$$sp = KP_c \tag{4.1}$$

Matrix K contains the five intrinsic parameters that are specific for the camera and for its configuration (aspect ratio and video stream quality). Their meaning is to convert the measurement units from meters into image pixels in the corresponding image 2D plane. They can be estimated once since they are fixed and then they will be used as known values in the system.

$$K = \begin{bmatrix} F_x/L_x & \gamma & u_0 \\ F_y/L_y & 0 & v_0 \\ 0 & 0 & 1 \end{bmatrix} \approx \begin{bmatrix} f_x & 0 & u_0 \\ f_y & 0 & v_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.2}$$

$$\begin{bmatrix} X_c/Z_c \\ Y_c/Z_c \\ Z_c \end{bmatrix} = s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & u_0 \\ f_y & 0 & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix} \tag{4.3}$$

F_x **and** L_x**:** Focal length on the camera's x axis in meters and width size of one pixel in meters, respectively. Their ratio forms f_x = scaled focal x length as the independent parameter.

F_y **and** L_y**:** Focal length on the camera's y axis in meters and height size of one pixel in meters, respectively. Their ratio forms f_y = scaled focal y length as the independent parameter.

u_0**:** Principal point's coordinate on the width of the image, in pixels.

v_0**:** Principal point's coordinate on the height of the image, in pixels.

γ**:** Skew angle of the image's axes. Assumed to be zero, leads to only four unknown parameters.

Matrix M_a^c contains the six extrinsic parameters and represents the homogeneus transformation matrix to express 3D points from an arbitrary coordinate system in the camera coordinate system.

$$M_a^c = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.4}$$

$$P_c^{hom} = M_a^c P_a^{hom} \tag{4.5}$$

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_a \\ Y_a \\ Z_a \\ 1 \end{bmatrix} \tag{4.6}$$

R: Rotation of the arbitrary frame from the camera, defined with three independent parameters and 3x3 matrix.

t: Translation of the arbitrary frame from the camera, defined with other three values in meters and 3x1 vector.

The left hand side of the equation and the right-most element are the required inputs of the system.

$$sp = K M_a^c P_a \tag{4.7}$$

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & u_0 \\ f_y & 0 & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{bmatrix} \begin{bmatrix} X_a \\ Y_a \\ Z_a \\ 1 \end{bmatrix} \tag{4.8}$$

(u,v,1) Normalized image coordinates of the point in pixels on a camera plane at a distance of 1 as third coordinate. Can be seen as homogeneus vector.

(X_a, Y_a, Z_a): 3D point coordinates in meters expressed in the arbitrary reference frame

The calibration is the process of computing all the eleven unknown parameters in the intrinsic matrix K and the extrinsic matrix M_a from the most general solution of the equation called Direct Linear Transform. When the intrinsic parameters are already known (calibrated or given) different algorithms are employed to obtain only the extrinsic matrix, reducing the complexity of the estimation thus increasing accuracy and speed.

An analytic-based approach is chosen to perform the computation of the object's frame position and orientation with respect to the camera frame, starting from the association between an image 2D point in pixels and its 3D values in the arbitrary object's frame. The requirement is then to select specific keypoints (also called features) in the object and to detect their projections in the image with high confidence, while maximizing their number. Since one correspondence is not enough to solve the equation, at least four points are used to concur to the unique solution; RANSAC (Random Sample Consensus) is employed as an iterative approach to exploit an high number of points and to make the computation more robust to possible mismatches and wrong pairings. Pseudocode from [13], detailed description of the approach is in the algorithm 1 that applies for the general case, while the model and the data in the specific case are the camera projection and the couples 2D-3D.

Algorithm 1 Random Sample Consensus procedure to fit a model into a large quantity of data, from [13].

procedure RANSAC POSE(data, model, n, k, t, d)
    ▷ data – A set of observations.
    ▷ model – The camera projection pinhole.
    ▷ n – Minimum number of data points to estimate model parameters = 4.
    ▷ k – Maximum number of iterations allowed in the algorithm.
    ▷ t – Threshold value to determine data points that are fit well by model.
    ▷ d – Number of close data points to assert that a model fits well.
    ▷ Returns: bestFit – model parameters which best fit the data.
    $iterations \leftarrow 0$
    $bestFit = null$
    $bestErr = inf$
    while iterations < k **do**
        $maybeInliers \leftarrow nRandomData$ ▷ n randomly selected values from data
        $maybeModel \leftarrow modelFittedToRandom$ ▷ model parameters fitted to maybeInliers
        $alsoInliers \leftarrow emptySet$
        for every point in data not in maybeInliers **do**
            if point fits maybeModel with an error smaller than t **then**
                $alsoInliers \leftarrow alsoInliers + point$
            end if
        end for
        if the number of elements in alsoInliers is > d **then**
            ▷ This implies that we may have found a good model
            ▷ now test how good it is
            $betterModel \leftarrow modelFittedToInliers$ ▷ fit to all points in maybeInliers and alsoInliers
            $thisErr \leftarrow modelErrorInliers$ ▷ a measure of how well betterModel fits these points
            if thisErr < bestErr **then**
                $bestFit \leftarrow betterModel$
                $bestErr \leftarrow thisErr$
            end if
        end if
        $iterations \leftarrow iterations + 1$
    end while
    return $bestFit$
end procedure

A first application is then the calibration of the camera intrinsic parameters. It is a delicate process that needs real pictures taken from the specific device, depicting an already well-known calibration board from multiple points of view. These boards can be the following 2D patterns: an asymmetric-size chessboard with white and black tiles on a rigid cardboard; fiducial markers tag like Aruco or QRcodes; other specific high contrast white/black shapes. The choice is from the accuracy of the detection algorithm used to extract and classify the specific pattern's points. The desired result is to exactly find all the features in the image regardless of how skewed or tilted the board is with respect to the camera. Here the classical chessboard is considered where the corners are the characteristic points, and the object reference frame is placed at the origin of the grid leading to know all the corner points positions in meters with respect to it as multiples of one tile's size.

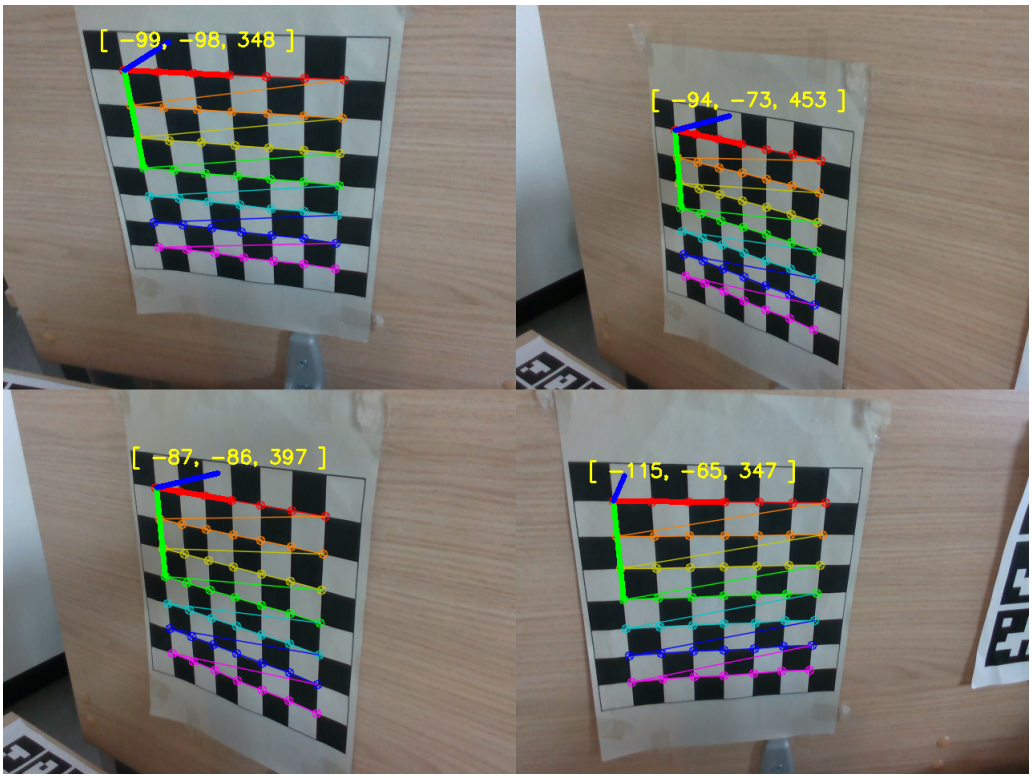

Figure 4.2: Calibration from detection of tile's corners. Origin of arbitrary frame shown in millimeters from camera reference frame

Two apparently concurrent concepts underly the required number of images: reducing the noise contribute, bias and external light conditions by shooting from many different points of view; each new camera pose adds 6 new extrinsic parameters to the equation. However each additional image brings 8 equations: in theory 2 images are enough to find a solution as they have 16 equations with 4+12 intrinsic and extrinsic parameters, and in practice the more the pictures, the better the result.

Algorithm 2 Calibration procedure. Stops after the chessboard pattern with all the corners is successfully found in 20 images.

```
1: procedure INTRINSIC CALIBRATION([] )
2:     while chessboards detected < 20 do
3:         New different camera position
4:         Take a picture
5:         Search all corners points
6:         Order corners into expected rows and columns
7:         Associate 2D image corner with 3D grid position
8:     end while
9:     Solve the system: 160 equations, 4+120 unknowns
10:    Keep the 4 intrinsic parameters
11:    Ignore the 120 extrinsic parameters
12:    return f_x, f_y, u0, v0                ▷ Camera parameters are returned
13: end procedure
```

Considering the delicacy of the calibration process (whose errors would affect all the measurements) and that the parameters are specific to the device, in the following the intrinsics are directly extracted from the Realsense D435 camera in the specific video stream configuration that is already calibrated by the producer. Moreover, the video streams are actually two: one from a RGB camera and the other as depth from a stereo camera, both at 30 FPS.

	RGB[px]	DEPTH[px]
Height	480	480
Width	640	640
f_x	609	610
f_y	327	248
u_0	384	384
v_0	320	240

Table 4.1: RGB and depth configuration

Finally to perform pose estimation with both cameras, since there are two points of view but all the measurements need to be expressed in one reference frame, a conventional choice is made to consider the RGB camera position. Knowing the relative positions of the two cameras' lenses and, through a geometric transformation matrix it is easy to refer all coordinates into one of them. From 4.9, depth frame origin is just a translation on the positive x axis.

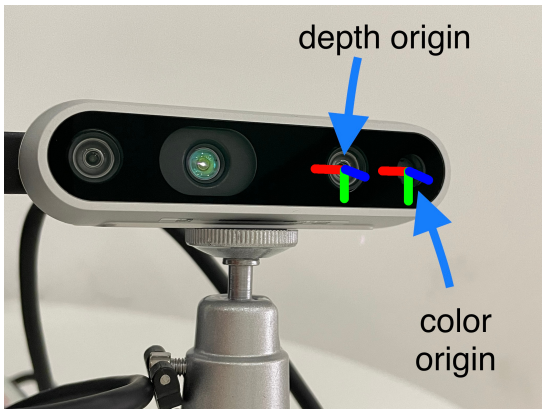

Figure 4.3: Reference frame positions of the two cameras

$$M_{depth}^{color} = \begin{bmatrix} 0.9998 & 0.0166 & -0.0057 & 0.0148 \\ -0.0166 & 0.9998 & -0.0022 & -0.0003 \\ 0.0056 & 0.0023 & 0.9999 & -0.0003 \\ 0 & 0 & 0 & 1 \end{bmatrix} \approx \begin{bmatrix} 1 & 0 & 0 & 0.015 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.9}$$

Then a point in depth coordinate frame is referred to color frame as:

$$P_{color} = M_{depth}^{color} P_{depth} \tag{4.10}$$

4.2 Features detection and tracking in the image

The goal of a computer vision tracking system is to follow a group of pixels of interest while they change position from one camera frame image to the consecutive. In general it enables applications that:

- Understand the direction of motion of an object
- Compute the speed of an object from fixed camera
- Provide a surrounding region to search the object in the following image

In classical computer vision there a plethora of algorithms to process one image and extract points of interest that are generic to different objects: these points are called visual features or keypoints. The visual features considered in the following work are the edges and the Kanade Lucas Tomasi (KLT) keypoints; even though they have been developed in the late 90' and lack robustness in light conditions they still maintain a good relevance in modern computer vision as they have a solid mathematical basis, fast computational speed, wide range of use cases, and in the years many applications have been built from them. A third visual information added is the depth from the stereo camera of the Realsense D435.

Edges: Contours points of the object projection in the image. Efficient for most objects as it does not consider texture inside the contour, only the external points.

KLT: Keypoints extracted from the image and associated to identifier information. It is powered by textured objects as they have many points of interest.

Depth map: map: Distance from the camera of a point in the image. Works for any kind of objects, as long as the points are inside an optimal distance and the stereo camera have lines of sight.

The resulting system is able to output specific keypoints in a single camera image and assign to them descriptions of their properties and position in the image. As the object changes position in time its points in the image move from their previous coordinates while keeping a visual similarity; by extension the system could detect most of the same features' properties in the following camera frame. Tracking is employed to reduce the complexity cost: from the assumption that the object is not moving too fast (that is, large change in its projected image coordinates) the search can be performed in small windows around the previous positions instead of on the full image, with far less points to consider. Computation speed is greatly increased and a better accuracy can be achieved at the cost of a dependency on the relative velocity between the camera and the object. Noteworthy is that tracking

considers relative movement so it is equal if either the camera is moving and object is static or the vice versa.

4.3 ViSP model-based tracker and 6D-pose estimation

ViSP is an open source Visual Servoing Platform providing a C++ library to build powerful computer vision applications, presented for the first time here [14]. Their model-based tracker is at the core of the system developed to solve the task of this work as it implements both the tracking of features and the estimation of the object's pose; those two elements work together and they enforce each other to increase accuracy and detect failures. As the camera moves together with the robotic arm a real-time 30 fps pose estimation must be achieved. The generic model-based tracker is described here [15] and the great tutorial is found here [16].

First of all, a CAD model of the target object is given. This is the mesh composed by the vertices as 3D coordinates in meters in an arbitrary local object's reference frame, then by how these points are connected to form the edges and the faces. The result is a collection of polygons all referred to the local frame, each convex and with the normal to the surface in the direction going outside of the object. For a simple model all this information can be manually written in a text file, while more complex meshes can require the graphical interface of a CAD software and then exporting the data into the text file format. An important additional information in the CAD model is the string name identifier of the defined faces.

Point	X	Y	Z
0	-W/2	H/2	0
1	-W/2	-H/2	0
2	W/2	-H/2	0
3	W/2	H/2	0
4	-W/2	H/2	L/2
5	W/2	H/2	L/2
6	W/2	-H/2	L/2
7	-W/2	-H/2	L/2

Table 4.2: CAD model definition

```
# 3D Points
8                     # Number of points
-0.0125     0.0075    0.0       # Point 0: X Y Z
-0.0125    -0.0075    0.0       # Point 1
 0.0125    -0.0075    0.0       # Point 2
 0.0125     0.0075    0.0       # Point 3
-0.0125     0.0075    0.075     # Point 4
 0.0125     0.0075    0.075     # Point 5
 0.0125    -0.0075    0.075     # Point 6
-0.0125    -0.0075    0.075     # Point 7
# 3D Lines
0                     # Number of lines
# Faces from 3D lines
0                     # Number of faces
# Faces from 3D points
6                     # Number of faces
4 6 2 3 5              name=right
4 0 4 5 3              name=bottom
4 0 1 7 4              name=left
4 1 2 6 7              name=top
4 4 7 6 5              name=rear
4 0 3 2 1              name=front
# 3D cylinders
0                     # Number of cylinders
# 3D circles
0                     # Number of circles
```

Listing 4.1: CAD model in .cao format. Faces are defined connecting the 3D vertices, each connection defines an edge.

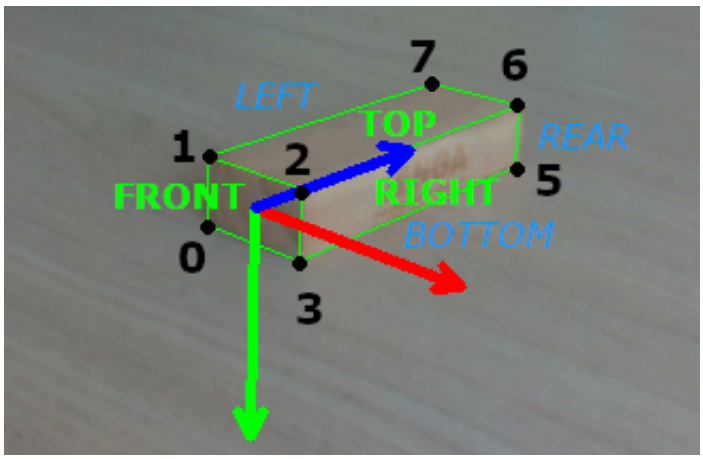

Figure 4.4: CAD model with object reference frame position and orientation, edges, faces and vertices

At this point the geometric structure of the object is known with all the appropriate real dimensions. All points on its surface will be assumed to lay either on the edges or on the faces. No information is given for the texture or the appearance as they are assumed to be unknown (they are different for each instance of the Jenga block). Two tracker systems are available that work on different geometric concepts and visual features but that are complementary and used together make the system much more robust.

Moving edges (ME) tracker. from the knowledge of the model's edges as contours of the faces, they are searched in the image as points on straight lines with high contrast change as the edges features described in the previous section. This is more robust when either the division between two faces is marked or if the two faces have different colours. In any case when the connected faces form a small angle the light will often help in providing a different contrast as one face could be slightly darker.

In the Jenga wood block case, the edge is not marked and the faces are very similar in colour but at least the angle between faces is always 90° and so the light will likely shine in different ways: from figure 4.5, many edges with the floor are found but very few in with background or between faces. Each edge will be tracked separately but the model's knowledge is used to search their points in specific locations in the image by projecting the relative 3D positions to maintain always the correct shape. In detail the moving edge tracker detects and computes a number of points and tries to find them again in the following camera image. The method is a convolution window that is oriented along the normal direction to the edge, centred on each of the points. The resulting value is compared to find the same point again and also the angle from the starting one, resulting in new set of edge points that must agree with the new orientation of the line or else are discarded. Parameters of this algorithm can be tuned to fit the use case:

Size: Convolution mask size

Range: Pixel distance in both directions of the normal to the edge to search

Step: Minimum distance in pixel between two consecutive points on the edge

Threshold: Likelihood value to compare the result of the convolution with

KLT tracker: from the knowledge of the model's faces, points on them are detected from the KLT feature extractor explained in the previous section and are associated with the known polygon. Assumed to be on a plane whose orientation and position is known, the 2D image position of the points has a correspondence with their position in the 3D plane of the object. Then finding those same points in the following camera image it is possible to compute the new position and orientation of the plane they are on; doing this for all the faces the polygons

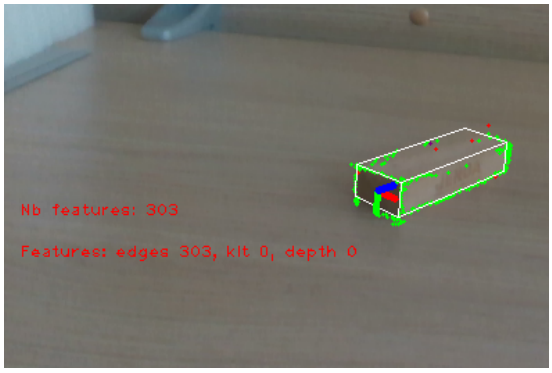

Figure 4.5: Edge features are the green points, easily found between the block and the background. Red points are wrong excluded.

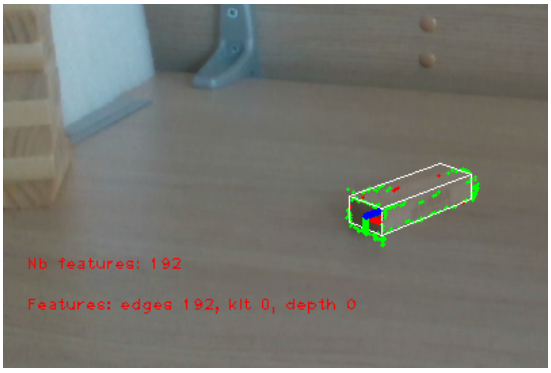

Figure 4.6: From a greater distance, less points are detected, 33% less.

composing the object's mesh will be placed in space accordingly. This method is good if the faces have notable textures whose distinct features can be easily tracked again in the images with high precision and low number of false positives. In detail the knowledge of the 3D model is used to perform the first detection of the keypoints only on the image location where the face is projected. During the movement these features will be searched but their number will decrease in time as the face changes orientation with respect to the camera, modifying the look of the features; in this case a new detection is performed and new fresh keypoints are added to the search. This tracker is highly dependent on the number of points. Parameters to be tuned are:

Mask border: Pixels to erode the perimeter of the face and not consider points too close to the edge

Max features: Maximum number of KLT to consider in the entire image. High number would slow down the computation

Quality: Percentage to consider good points to track on the face; features below this threshold are discarded

Min distance: Minimum forced distance in pixels from one feature to its closest. Low number increases the number of features on the face

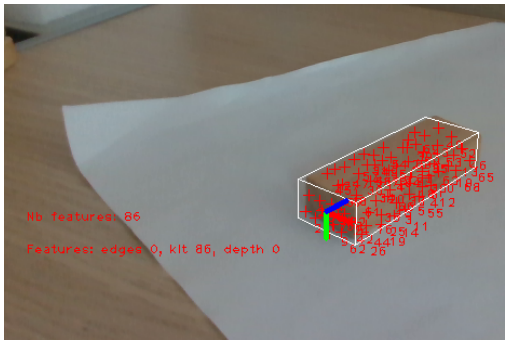

Figure 4.7: KLT=86 and leads to wrong orientation of the model

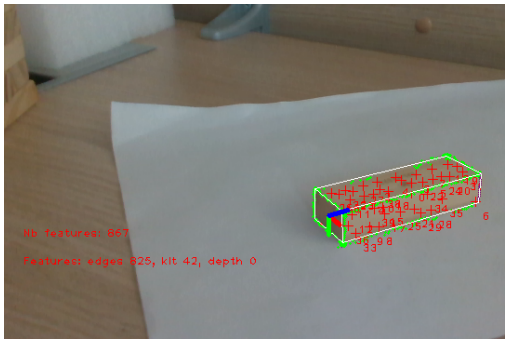

Figure 4.8: Combined Edges + KLT=367 yields better performance and completes the model

Pose estimation as explained in section 4.1 is computed from the association of 3D points in the object's reference frame with their corresponding projection in the 2D image. Both the Moving Edges and the KLT trackers constantly compare their results with the 6D-pose of the whole object projected in the camera to build a consensus and from there discard wrong points and enforce correct ones.

The depth camera is then used to provide a reference of third dimension to the 2D points in the image with two results: one is to ensure that the trackers' algorithms won't choose pixels of the background as features of the object since otherwise possible based only on edge's contrast; the other is to speed up the convergence of the pose estimation by restricting possible the values of Z coordinates in camera frame of the tracked 2D points to a range close to the measured one by the stereo camera. The advantage of this visual feature is that it does not depend on either the textures or the look of the object like the previous two. The disadvantages are that it depends on the optimal working range of the depth sensor and in particular the minimum distance from the target; also it produces holes when the stereo camera cannot see a point because occluded for one of the two lenses. Parameters to be tuned are:

Min range: Minimum distance in meters to consider a face for depth information

Max range: Maximum distance in meters to consider a face for depth information

Depth occupancy: Percentage of points whose depth is available to consider the face

Sample step: Pixel distance in x and y camera frame between points whose depth is computed

Figure 4.9: Adding the depth features the final number is way higher, EDGES + KLT + DEPTH = 1093

An important concept is the visibility of the edges and faces: not all of them must be considered during the tracking as the projection of the object in the camera image will show only the front facing ones while the others will be hidden behind. The tracked points explained above are only contained on edges and faces that can be seen in the camera, otherwise they would never be found in the image and would give wrong information to destabilize the system. This visibility can be checked by comparing how the normal's direction of one face is oriented with respect to the camera. If the angle between the projection of the center of mass of the face and its normal is above 90°, the entire face is discarded with all its edges expect the edges that are marked as visible by passing this test from another face.

Stronger conditions can be applied by tuning the angle's value, considering a face as disappearing if $\alpha > \alpha_{disappear}$ and appearing if $\alpha < \alpha_{appear}$. This helps to avoid bad features detected on a face too skewed. Noteworthy is that the condition either excludes or keeps the entire face, never partially. This means that the polygon must be a plane.

Angle appear: Below this angle a face is considered visible and added to tracking

Angle disappear: Above this angle a face is considered not visible and removed from tracking

Finally for the visibility, clipping planes can be set to exclude parts of faces that are too close, too far or outside of the Field Of View (the lateral sides). Parameters to be tuned are:

Near clipping: Minimum distance in meters to cut close faces

Far clipping: Maximum distance in meters to cut far faces

Field of view use: Use or not the lateral sides to cut faces that partially exit the camera frame

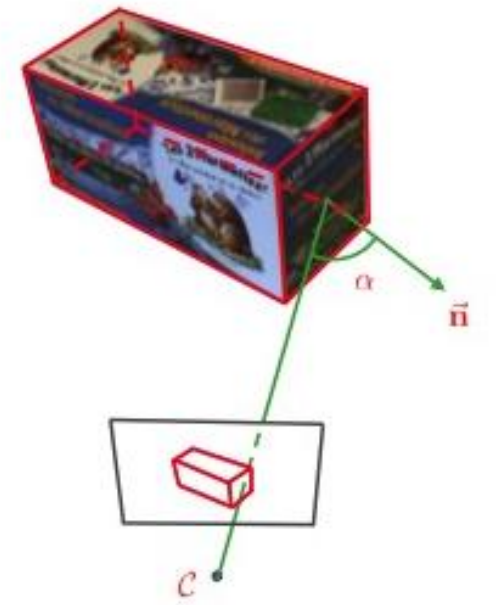

Figure 4.10: Angle comparison with the projection on the camera frame to consider the face as visible from that position

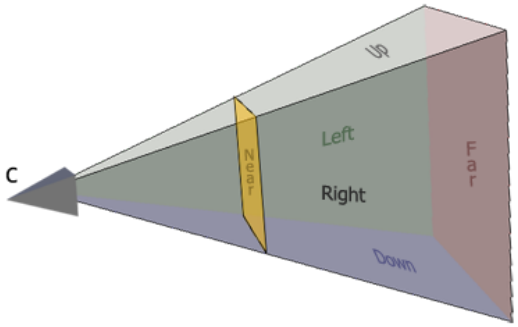

Figure 4.11: Diagram of the field of view to customize object visibility

4.4 Tracking blocks and handling occlusion

The vision system is applied to the task with real camera and the target object in a series of experiments. To be noted here is that the tracking must receive an initial guess pose to reproject the overlay of the model onto the image and consider the starting location of the keypoints as edges and faces. In this section it is done manually by the user associating 2D points in the image with 3D points of the considered object model, basically reproducing the pose estimation. In the following two sections this will be refined to produce an automated pipeline without the user's interaction.

A first round of experiments is performed with the following setup:

- Single Jenga block with dimensions: 25x15x75 mm
- Block located on a uniformly colored plane and no other blocks
- Block fixed in its position
- Realsense camera moved manually
- Camera distances are: 0.18 , 0.29, 0.4 m
- Camera always oriented towards block

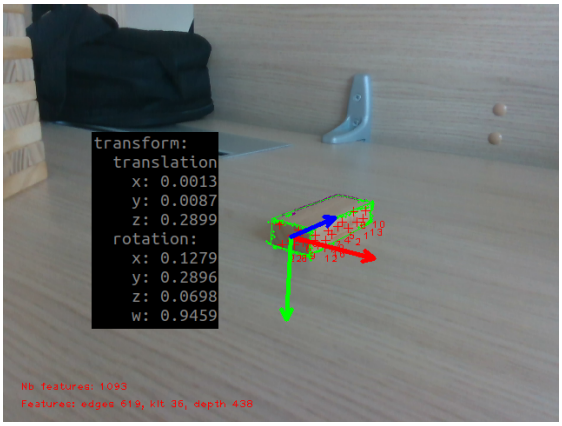

Figure 4.12: From a middle distance of 0.29 m on the z camera axis.

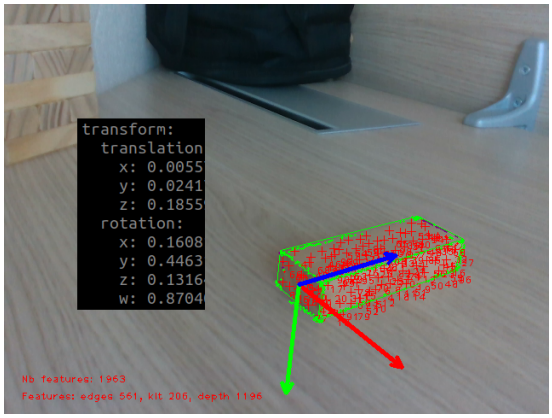

Figure 4.13: Distance of 0.18 m

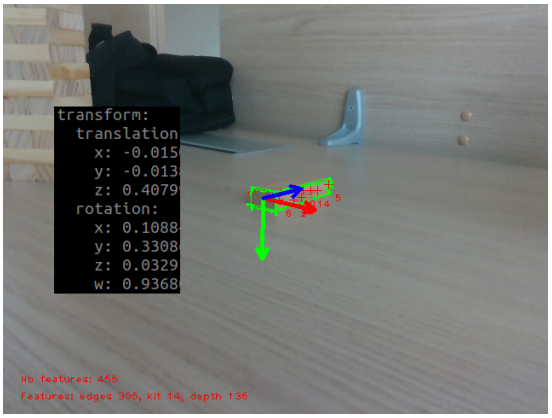

Figure 4.14: Distance of 0.40 m

Analysis of advantages and disadvantages of this configuration:

OK Strong edge contrast against the background: object's contour has no mismatches

OK Block is fully visible: three faces are completely seen at any time by the camera

OK Depth values of the block helps associating 3D info

! The block is small so it covers a little number of pixels in the image when far

! It is almost textureless producing very few KLT points on faces when far

! Edges between faces are not distinct enough from wood lines

! Background depth info not interesting: similar Z distance between block points and plane

Results and considerations are heavily affected by the distance to the camera as tracking from a far distance (>0.3 m) is impossible: too few keypoints are found. The pose estimation from a single still image is accurate only when the tracker can overlay the 3D model correctly on the block with sufficient keypoints. However tracking is easily lost: during an even slow movements of the camera some keypoints make wrong matchings. Depth information is used only partially

A second round of experiments on the use case:

- Single Jenga block with dimensions: 25x15x75 mm
- Target block in the tower among other blocks
- Target block in external position on the tower's layer
- Tower fixed in position
- Realsense camera moved manually
- Camera always oriented towards the tower

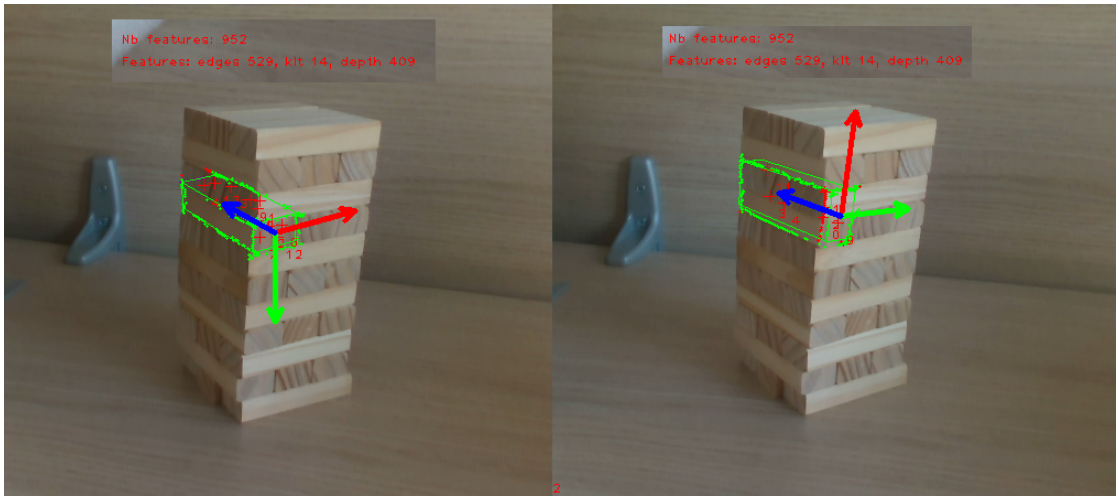

Figure 4.15: The top face is occluded by other blocks causing a failure of the tracking, mainly from the edges and the depth.

Analysis of advantages and disadvantages of this configuration:

OK Strong edge contrast with background ONLY IF it is uniformly colored

OK Strong edges between target and surrounding blocks, marked better than edges between faces

OK Depth information helps excluding background points as they have different Z coordinates

! Block is not fully visible: only two faces can be seen by the camera, the others are occluded by the rest of the tower

! Block is still textureless and now even less keypoints can be seen

! Both edges and KLT keypoints find mismatches with nearby blocks, having visually similar features

! Depth values of hidden faces are instead closer

Results and considerations are mainly referred to the occlusion of the block's faces, as they not only hide good features to be tracked but also provide completely wrong data to the system: • Edges: the tracker expects an edge but instead finds points of variable contrast in the other blocks and tries to align them. • KLT: the tracker finds keypoints on the region where it expects the hidden face but the plane they are on is oriented in a very different way. • Depth: the tracker finds Z coordinates that are not further than the front face depth.

The developed solution to the problem has the goal of greatly increasing the number of available visual features and involves the design of new CAD models that accounts for the specific situation of the tower. Based on empirical knowledge and from the Jenga rules the relative positions of the blocks are known and assumed to remain the same. They are either on the same floor and oriented in the same way or they are one floor above or one below and they are rotated. In particular the creation of three new CAD models that are the building blocks whose combinations produce any possible groups. The new .cao model file is a collection of these blocks that are sequentially loaded into the model-based tracker each adding 8 vertices, 12 edges and 6 faces.

To produce the correct geometric structure the arbitrary object reference frame is positioned on the first block with the already shown origin and orientation, the same as the single block. All the other additions have a translation and rotation with respect to that frame to correctly locate them; translation is in meters while the rotation is in the form of axis-angle in degrees. The first parameter is the source single block .cao file that loads the vertices and faces.

```
# Block 1
load("jenga_Center.cao", t=[0.0; 0.0; 0.0], tu=[0.0; 0.0; 0.0])

# Block 2
load("jenga_Right.cao", t=[-0.025; 0.0; 0.0], tu=[0.0; 0.0; 0.0])

# Block 3
load("jenga_Left.cao", t=[0.025; 0.0; 0.0], tu=[0.0; 0.0; 0.0])

# Block 4
load("jenga_Left.cao", t=[-0.0375; -0.015; 0.0125], tu=[0;90 deg;0])

# Block 5
load("jenga_Left.cao", t=[-0.0375; 0.015; 0.0125], tu=[0;90 deg;0])
```

Listing 4.2: CAD model of the group loading the single models

- The first block has a translation and rotation equal to 0 vectors as its reference frame is coincident with the group reference frame.
- The second and third blocks are located on the same floor of the first and their rotation with respect to it is 0 (aligned to it). They are translated by 25 mm on the X axis (red axis) in positive and in negative direction respectively.
- The fourth and fifth blocks are located in different floors and they are rotated by 90° around the Y axis (green axis). The translation to move them up and down is 15 mm in positive and negative direction of the Y axis (green axis); also values in X and Z are required to have the correct origin.

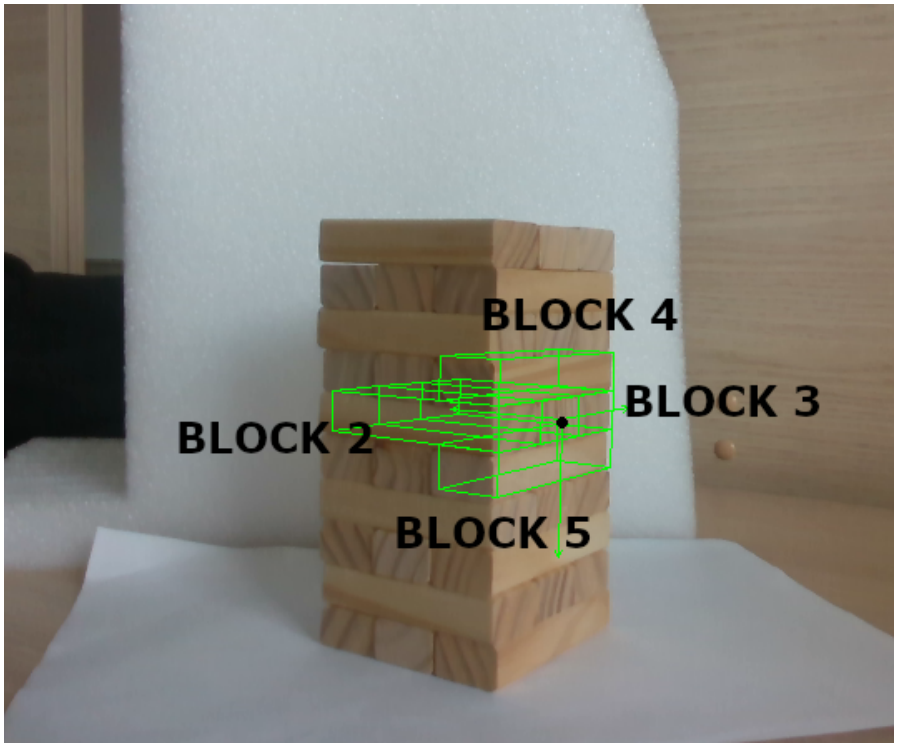

Figure 4.16: Overlay the group composed of target block plus four others, built by projecting from the initial pose

Vertices and object frames orientation are identical among the new single models and the only difference is the name of their faces where instead of assigning a unique string identifier, a binary value is used: either the face is called "visible" or "occluded". To account for the visibility problem, then, the faces called "occluded" are removed a priori from the tracking and none of their points will be searched in their image. The decision is taken at the moment of building the group where:

Right faces: depends on position in the group

Bottom faces: always "occluded"

Left faces: depends on position in the group

Top faces: always "occluded"

Rear faces: always "visible"

Front faces: always "visible"

The only variables are the faces on the side producing 4 combinations, where the last is not used.

RIGHT	LEFT	RESULT
"Occluded"	"Occluded"	Central block
"Visible"	"Occluded"	Right block
"Occluded"	"Visible"	Left block
"Visible"	"Visible"	-

Table 4.3: Combinations of faces visibility

```
16 # Faces from 3D points
17 6                    # Number of faces
18 4 6 2 3 5             name=occluded  #right
19 4 0 4 5 3             name=occluded  #bottom
20 4 0 1 7 4             name=occluded  #left
21 4 1 2 6 7             name=occluded  #top
22 4 4 7 6 5             name=visible   #rear
23 4 0 3 2 1             name=visible   #front
```

Listing 4.3: CAD model in .cao format, showing only the faces part.

On image 4.16, the first block is in the center and its only visible faces are the front and the rear. The second, fourth and fifth blocks also have the left face visible, while the third block has the right one instead. Different cases follow the same reasoning and even if a block is missing the group is built without it.

```
1  # Block 1
2  load("jenga_Center.cao", t=[0.0; 0.0; 0.0], tu=[0.0; 0.0; 0.0 deg])
3
4  # Block 2
5  load("jenga_Right.cao", t=[-0.025; 0.0; 0.0], tu=[0.0; 0.0; 0 deg])
6
7  # Block 3
8  load("jenga_Left.cao", t=[-0.0375; -0.015;  0.0125], tu=[0;90 deg;0])
9
10 # Block 4
11 load("jenga_Left.cao", t=[-0.0375;  0.015;  0.0125], tu=[0;90 deg;0])
```

Listing 4.4: CAD model of the group with a missing block, where there are only four blocks

Even if the internal face is partially visible, it is removed regardless as it is dependent on camera point of view and covered by other blocks that may or may not be in the group. In any case its shadows and light conditions are so unstable and different that very few good visual features could be found: they would not really help the tracker.

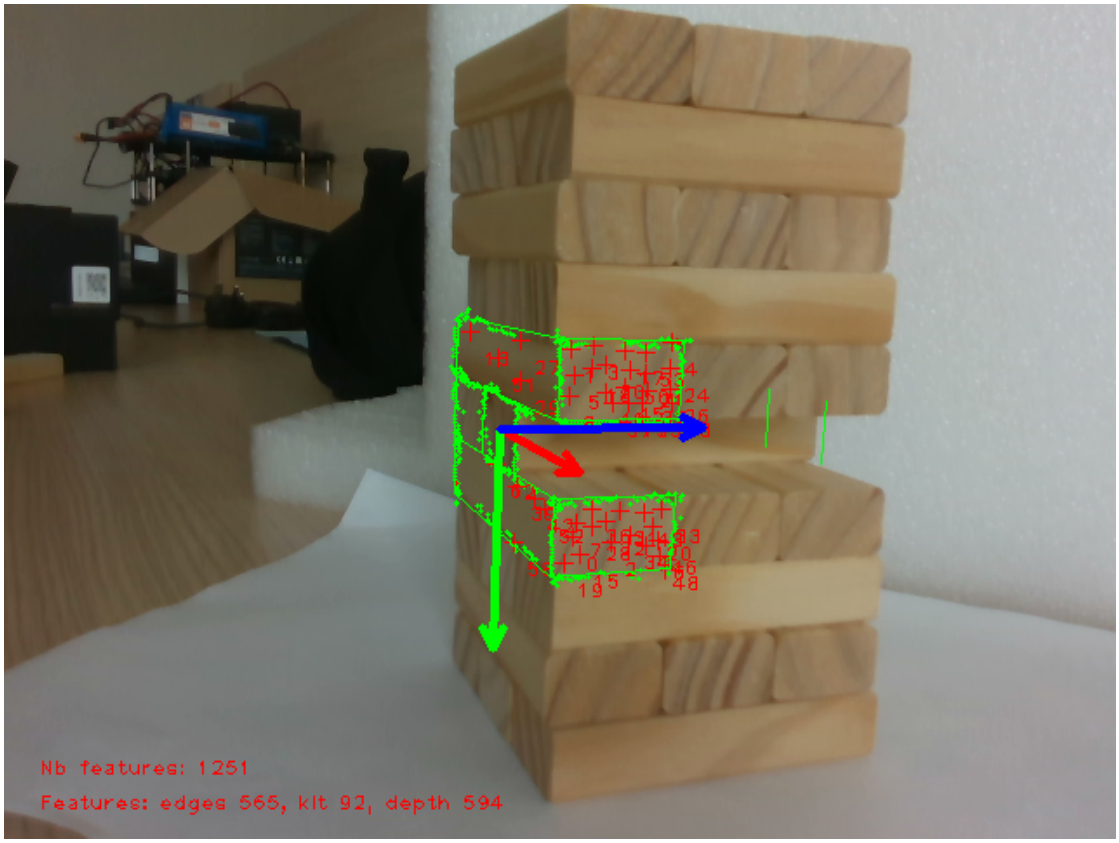

Figure 4.17: Example of tracking a group with a missing block

Disclosing here the tuned values of the parameters:

Size	5
Range	5
Step	2
Threshold	10000

Table 4.4: ME

Mask border	5
Max features	300
Quality	0.015
Min distance	8

Table 4.5: KLT

Min range	0.01 m
Max range	1.00 m
Depth occupancy	25%
Sample step	4

Table 4.6: Depth

Angle appear	75 °
Angle disappear	80 °
Near clipping	0.01 m
Far clipping	1.5 m
FOV clipping	True

Table 4.7: Visibility angles and clipping faces parameters

4.5 Learning and detecting online

The tracking of the group of objects is stable during slow movements and while the tower appears in the camera images; however, to implement the system on a camera mounted on the robot's arm a greater level of robustness is taken into account. In general the robot could perform the following operations during the movement: 1) Starting with high velocity in the first part of the approach to the tower 2) Tower partially leaving the field of view of the camera 3) Any abrupt change in direction 4) Changes in the orientation of the end-effector leading to rotations of the camera 5) Reaching joint limits and replanning the approach While points 2) and 4) can be dealt with by correctly designing the initial position and the control law, the others could lead to a new camera position too far from the previous one therefore making the tracked object move too fast in the image. The solution is then designing a computer vision system that is half tracking and half learning and detection.

TRACKING: Performed every camera frame and it follows the object in image.

LEARNING: First phase done only once, extracting keypoints from full images of the real tower.

DETECTION: Acting on the entire image based on learning data and it is started only in the case if: the tracking loses the position of the target; or the tracking finds a loss in the accuracy.

In detail, the learning begins with the extraction of features using ORB (Oriented-FAST and Rotated BRIEF). This algorithm converts points of interest that are visually distinct one from the other, into a database associating them with detailed descriptors of their visual features with properties. A short explanation can be found here [17]. The main drawback is the weakness to image scaling, which means that it's not robust to changes distance from the target.

The choice of the points is made with FAST (Features from Accelerated Segment Test), a corner detection method that with high computation speed from a full image selects keypoints in a grayscale image with the property of being darker or lighter than all its nearby pixels.

Detection phase is instead finding again the points learned from a first image in a new picture portraying the same scene from a slightly different distance or angle: using the same corner detection method most of the new keypoints refer to the same real physical point, then the extraction of features will lead to similar descriptions of the detected points. To finally associate learned points and new extracted, a BruteForce-Hamming matcher is employed. For each descriptor in the new set of keypoints it computes the Hamming distance from all the learned

descriptors, returning the closest in terms of number of equal features. Performing the operation on all the new keypoints, the percentage of matchings against not matched is also the criteria to decide if the new image contains enough of the same information as the reference learned one. In this case the ratio is 80%.

DETECTOR	FAST
EXTRACTOR	ORB
MATCHER	BruteForce-Hamming

Table 4.8: Configuration of the real-time computer vision system for learning and detecting

The implementation of this system using ViSP is done through the OpenCV library that provides easy access to all three of the mentioned algorithms in terms of handling of the image, the keypoints and the results; moreover, they are license-free. Building on these utilities the application is to restrict the learning to only the single target object to be tracked, which means that the featured keypoints to be saved in the image are those belonging to the group of Jenga blocks; naturally following from this is that the detection phase is only able to match those descriptors and doesn't consider the rest of the image. This can be performed only while the tracking is currently active because as mentioned it has the knowledge of where the 3D model currently is located in the image. The FAST corner detection and the ORB feature extraction still act on the entire image but the keypoints are filtered to keep only the pixels that in the current camera frame are tracked to be on the object. Extending the learning from 2D image to CAD object model an additional information is stored in the feature descriptor database: for each keypoint its corresponding 3D coordinates in the object frame. Exploiting the model-based tracker the points on edges and faces, assumed to be located on known planes, are associated with their projection in the image and then with the extracted keypoints.

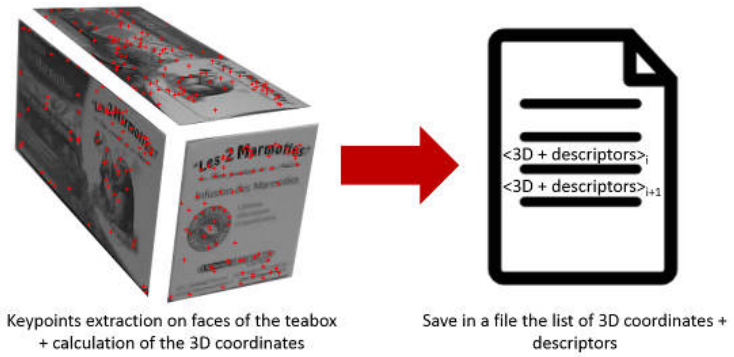

Figure 4.18: Learning step from [18]

Multiple points of view can build a reference database containing keypoints seen on different faces of the group of blocks. Each new learning image could contain new points or already saved points depending on how similar the camera positions are; regardless, same physical points will refer to same 3D coordinates with possibly slightly differences in features' descriptions resulting in a richer information more robust to viewpoint. The final purpose of the learning stage is then to combine the generic 3D CAD model of the object with the actual textures appearing on its surface, in the moment it's confronted with its real-world counterpart.

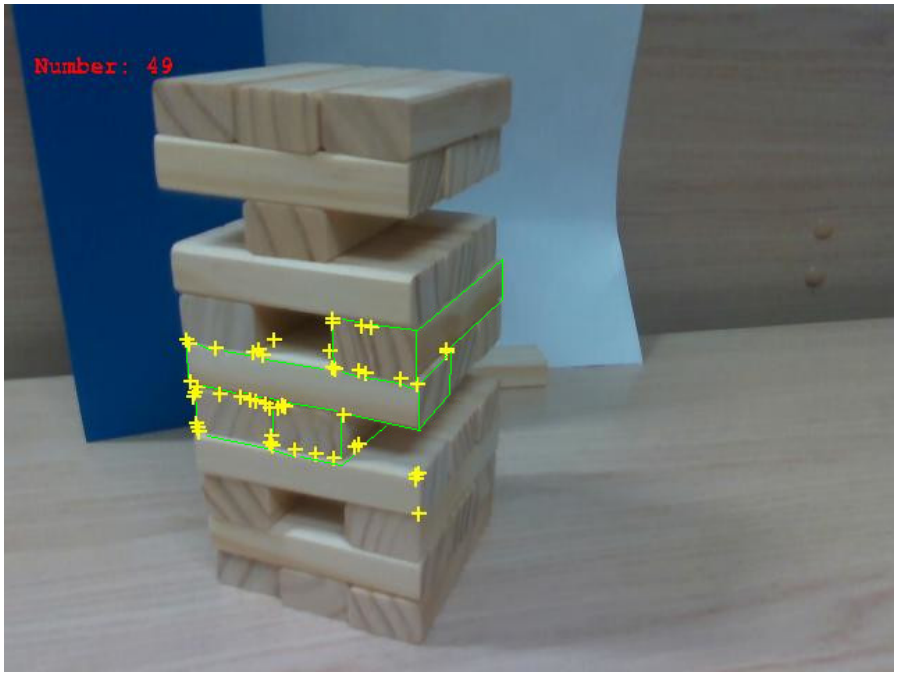

Figure 4.19: Learned keypoints on the model, they are corners in the grayscale image.

Moving on to the detection and matching stage, this is also extended to leverage the added 3D data of the model-based learning stage. A 6D-pose estimation of the object can be computed from the association between the keypoint and the corresponding 3D coordinate in the object frame, resulting in an automatic re-initialization of the model-based tracker from the new pose in the camera reference frame. Here the detection is from the entire image while the reference database contains only points of the object; the matcher finds many of those points but often also false positives in nearby Jenga blocks that are naturally very similar looking.

Here the details of this pose estimation algorithm where the VVS (Virtual Visual Servo) is the function at the core, from the work at [19]. The addition of the RANSAC method described in the procedure 1 in section 4.1 helps to increase the

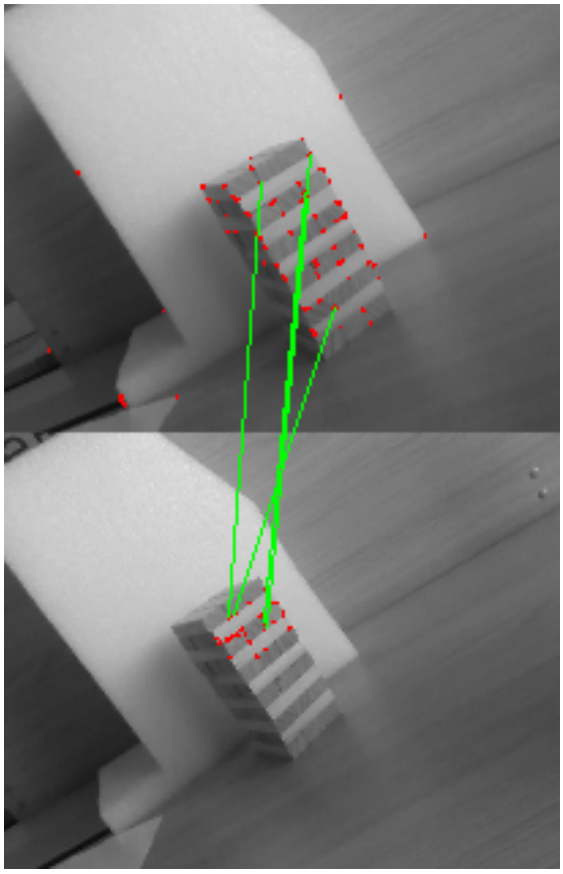

Figure 4.20: Matching from learned image, below, into current image above. Red points are the detected corners, green lines are the matchings, either correct or wrong

robustness to outliers that is critical when the 2D-3D pairings originate from long and complex computations prone to mismatches. In particular the parameters can be tuned to the application and the quality of the matcher; in this case since the output is the initialization pose whose accuracy is not required to be very high, a low percentage of the consensus is enough for refinement while the large complementary percentage of possible outliers avoids using too many wrong matchings in the computation. Number of maximum iterations is kept high but it's usually higher than the number of groups of 4 keypoints, since the matcher can only find points on the object. Values are in table 4.9.

Pose method	VVS
RANSAC	consensus
percentage	20%
max iterations	200
reprojection threshold	5°

Table 4.9: Pose estimation configuration for large number of pairings

The full detection pipeline for a new image is then the following:

1. FAST corner detection on full new image
2. ORB feature extraction and description of corners
3. BruteForce-Harris matcher each keypoint
4. VVS pose estimation
5. RANSAC iterations of VVS estimation
6. Resulting pose

The computation effort is very high, and it is not suitable for running at every single frame of a 30 fps camera stream as a run of it on a regular laptop averages more than 33 ms, which is the time before the new frame is provided. For this reason, the pipeline is activated only after failures of the real-time tracker and to recover the correct current pose to reinitialize.

Chapter 5

Force sensor implementation

5.1 Force sensor characteristics

There is an implicit rule in the Jenga tower game that makes playing it much more entertaining: the wooden blocks' dimensions are purposedly not equal as there are slight variations in width and height in the range of ± 1 mm. With this small difference when a player looks at the tower they cannot just see which blocks are smaller or larger and so neither they can understand how adjacent blocks would interact with each other. This means that even with perfect knowledge of the tower's physical properties and the ideal dynamic behaviour during one interaction, the effective movement of the blocks is different for every configuration and in particular the friction acting from the target block to all those around it. Since vision is not enough to acquire all the information, players are then required to touch before making their move. Through the feedback of their fingers they can discover if a block resists the motion or not: with a slight push on one of them a human can quickly understand if too much friction force is acting between the target and the adjacent. In theory the combination of sight and touch gives the full data to extract a block without perturbing the tower: players also look at the tower when they explore it with the slight pushes. Not only they feel the force but they can also see if the rest of the tower is rotating due to the interaction or if the floors above or below are sliding away. Moving from human player to a robotic arm some adjustments and data are needed to design the implementation. First the sensor is placed on top of the robot's end effector and it directly interacts with the tower during the push. As the sensor and the robot's finger are rigidly connected all the force acting on one is also fully transferred to the other, expecting a reaction normal to the physical touch interface. Second an estimate of static friction is

computed from the ideal block's weight. Reference values are taken from [1] and they have quite a large variance.

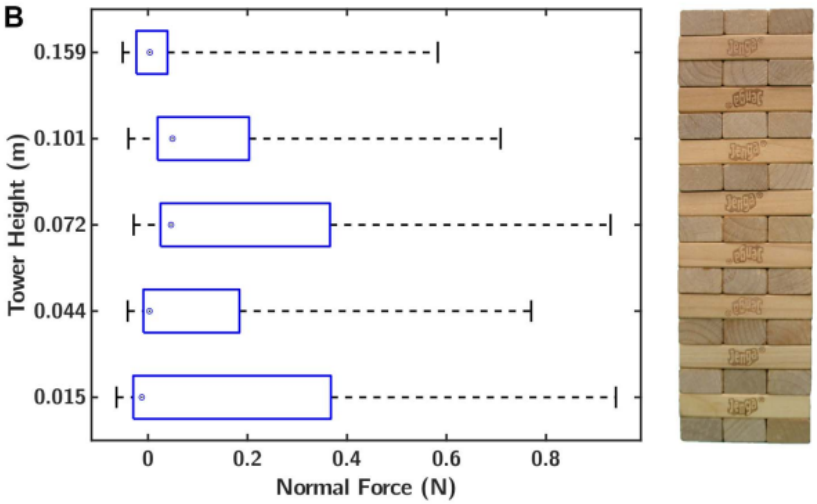

Figure 5.1: Friction force per floor. In blue when blocks are free while constrained blocks have up to 1 N

Finally the sensor's transduction technology is chosen: piezoresistive material that changes its electric resistance value when a strain is applied.

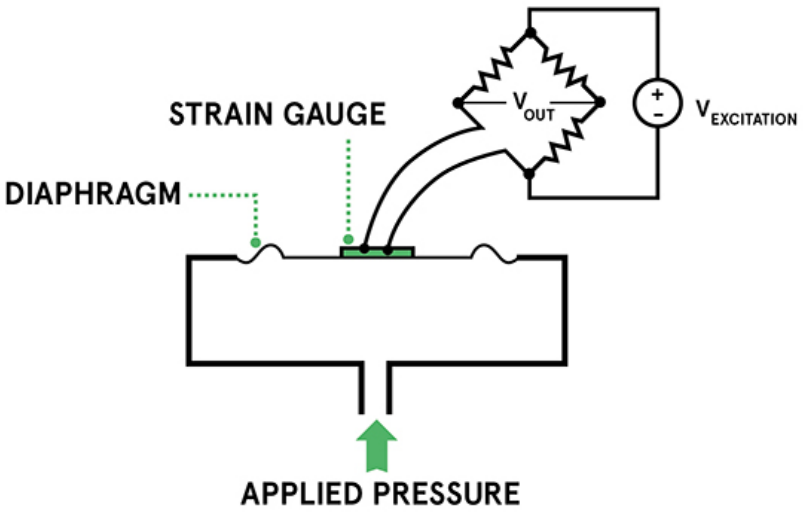

Figure 5.2: Schematic of a resistometric force sensor with voltage output

Sensor's size (constraint 1): Its width and height must be smaller than the single face of a Jenga block to avoid hitting the adjacent blocks.

Sensor's size (constraint 2): Its width and height must also be smaller than the available surface of the robot's finger to be able to firmly attach to it.

Expected force type: Compressive and normal to the surface in quasi-static condition.

Force low range: Highly sensible to small values to have a precise differentiation between no contact yet, very low block's reactions and excessive force.

Force high range: Saturation from a high value is ok. Must endure some overforce but the tower will fall instead of exerting a dangerous force reaction.

	System data	Sensor constraint
Jenga block size	Height= [15+-1]mm Width= [25+-1]mm	Side $<$ 14 mm
Fingertip size	Length1= 11mm Length2= 8mm	Side $<$ 8 mm
Ideal static friction	Full= [0.8÷1.2]N Small= [0.4÷0.6]N	Range: [0÷$>$1]N

Table 5.1: Quantitative requirements for choice of the sensor

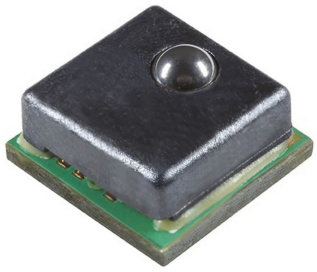

Figure 5.3: Sensor chosen

Name	MicroForce FMA
Producer	Honeywell
Size	5x5 mm
Force type	Normal compression
Sensor technology	Piezoresistive
Force range	[0 ÷ 5]N

Table 5.2: Sensor's main information

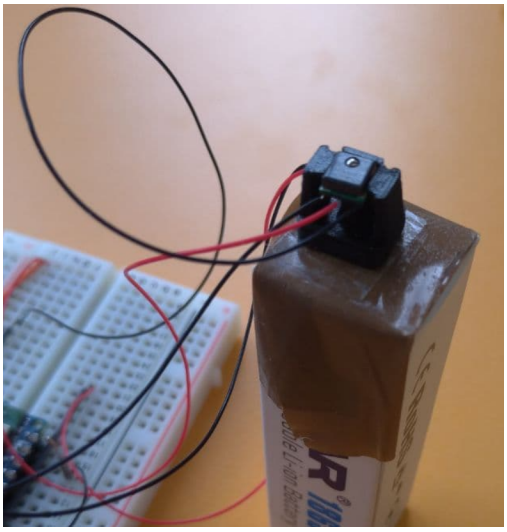

Figure 5.4: Sensor with 3D printed support

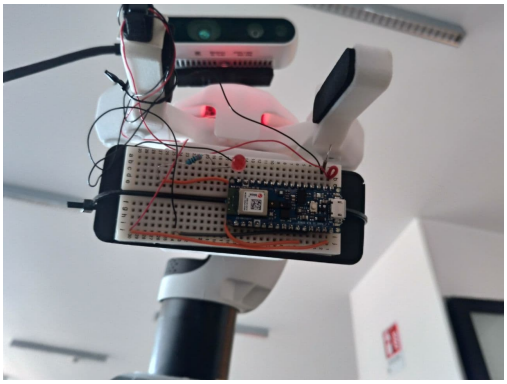

Figure 5.5: Sensor on robot end effector

5.2 Readings with Arduino Nano BLE

The chosen sensor combines the analogic property of the piezoresistive material with a digital circuit that takes the measurements and offers input and output interfaces. The core element is the piezoresistor, a material that placed under a contact force changes the value of its electric resistance. It's a transductor used to convert from mechanical quantities to electric signals.

- It has maximum resistance with no strain, ideally open circuit.
- It has minimum resistance with high strain, ideally short circuit.
- It's a passive component, needing a voltage supply to produce an electrical output.
- Signal is computed from a measurement of resistance

The FMA Microforce is an amplified sensor with a digital interface that is used to send a serial output to the controller. The protocol used is the Serial Protocol Interface (SPI) in a single Master single Slave configuration where the latter can only output data (half-duplex mode).

PINOUT	NAME	DESCRIPTION
1	VS	Voltage Supply
2	SS	Slave Select
3	GND	Ground reference
4	SCLK	Synchronous Clock
5	MISO	Master In Slave Out

Table 5.3: Sensor's pinout details

SS: Slave Selection that the master sets to activate the sensor, before this the measurements are not sent.

SCLK: Digital input synchronous clock from the connection Master to the Slave to dictate the frequency and the timing of all the communication.

MISO: Digital output synched with the input clock. Master In Slave Out is the line connection where the slave sends data and the master receive it. The bits composing the measurement are sent in a serial timing.

The FMA sensor needs a Master with compatible electrical and digital quantities. Technical details of the interface are described here.

QUANTITY	MIN	MAX
SPI input clock frequency	50 KHz	800 KHz
Voltage Supply	3 V	3.6 V
Current Supply	2.8 mA	3.9 mA
Digital LOW levels	0 V	20% of Voltage Supply
Digital HIGH levels	80% of Voltage Supply	3.6 V

Table 5.4: Digital and electrical operating specifications

The timing and the sequence of the SPI transfer protocol is the following as seen from the Slave point of view. The byte is sent in Most Significant Bit First.

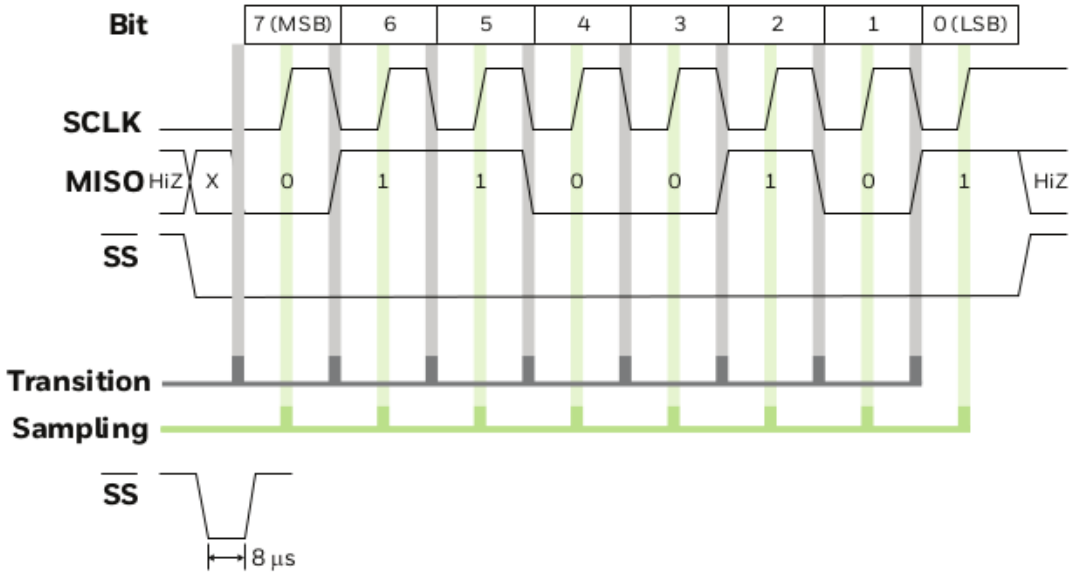

Figure 5.6: Detailed single byte SPI transfer with timing

The meaning of the data output at low level is from the single bits. Two bytes are sent by the sensor containing two different information. The first two bits are the 4 status and they can be used to have a continuous feedback about the correct functioning: if they are 00 everything is fine. 10 means master that is reading faster than the next measurement is taken, causing the data to be stale and already read before. 11 is the error status and the sensor can be broken. The remaining 14 bits are the numbers of interest that form a digital value between 0 and 2^{14}. This is converted later by the Master microcontroller into a force measurement in N units.

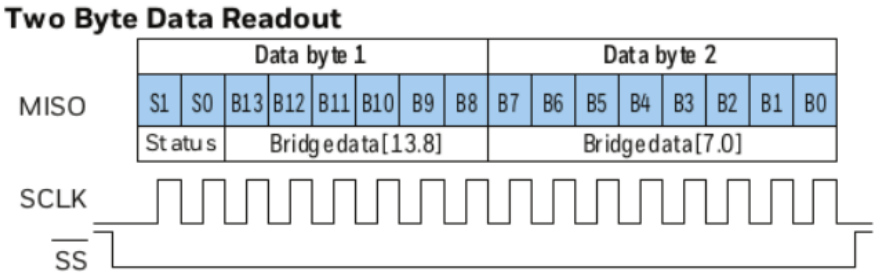

Figure 5.7: FMA sensor SPI two bytes transfer and bit content

Microcontroller Arduino Nano 33 BLE is chosen as compatible Master at first for the electrical reasons explained below. Moreover the implementation of the sensor in the overall system forced its position to be on top of the finger of the robot's end effector that is the most mobile element of the whole arm, the furthest

point from the base and finally a critical point to place some weight.

- Arduino Nano is very small in width, height, length
- Its weight is low and can't perturbate the dynamics when mounted
- Low consumption of power to last long even with a not very large power supply
- Provides 3.3V of voltage supply with signals up to 3.3V in input and in output.
- Bluetooth capabilities to avoid physical connection with central computer.

Setting Arduino up to be the Master of a SPI Slave is done through the dedicated software library that hides all the low level operations during the connection, after configuring with the same Slave parameters for the specific use case.

Voltage supply	3.3 V
SPI Clock speed	125 KHz
Data bit order	MSB First
Data mode	Mode0 (CPOL=0,CPHA=0,transitionFall,captureRise)
Dedicated clock pin SCLK	13 (default)
Dedicated MISO pin	12 (default)
Chosen SS pin	9

Table 5.5: Master configuration for the SPI

Presenting the code in the Arduino language format .ino, it has first the definition of the SPI connection and the choice of the pins that will be used in the circuit, while keeping the dedicated pins for their sole purpose. The SPI.h library is imported and started with the result of initializing the pins 12 (MISO) and 13 (SCLK) as output and idle. To avoid any possible miscommunications, before that the pin 9 (SS) is set as HIGH because the sensor is idle with that. Up to now the connection parameters are not yet used. Second the function that actually performs the reading is written with the goal of capturing the 16 sequential bits from the serial connection into a single variable to be parsed.

Here the master begins the transaction with the constant parameters defined previously, then providing the clock and enabling the slave by LOW value of the pin 9 (SS). Since in theory the SPI is a two ways communication, the library starts reading the values only after sending a series of bit through the hypothetic MOSI line (Master Out, Slave In); in this case the sensor has not even a pin to perform

this task and never reads anything the master is sending. After the 16 bits are transferred to the slave and the 16 bits are read on the pin 12 (MISO), the slave is deselected by HIGH value of the pin 9 (SS) and the clock stops, ending the transaction. The elapsed time for this is around TIME.

```
// the sensor communicates using SPI, so include the library:
#include <SPI.h>
// Slave select pin as master
const int SS_PIN = 9;
// Sensor SPI settings
const long SPEED = 125000;  //50k to 800k Hz of clock speed
// MSB first
// MODE0 behavior

void setup() {
  // initialize slave select pin as output and idle:
  pinMode(SS_PIN,OUTPUT);
  digitalWrite(SS_PIN,HIGH);  //FMA sensor idle when SS=high
  // start the SPI library:
  SPI.begin();
}

// SPI reading of the sensor value, returns 16bits of data

unsigned int readSPIsensor(){
  unsigned int result16 = OUTPUT_MIN;

  // Begin SPI transaction with custom parameters
  SPI.beginTransaction(SPISettings(SPEED, MSBFIRST, SPI_MODE0));

  // put slave select pin to low to notify the sensor:
  digitalWrite(SS_PIN, LOW);

  // Read 2 bytes from SPI
  // send a random value, read the first two bytes returned:
  result16 = SPI.transfer16(0);

  // put slave select pin to high to deselect the sensor:
  digitalWrite(SS_PIN, HIGH);

  // End SPI transaction
  SPI.endTransaction();

  return result16;
}
```

Listing 5.1: Arduino code to program the SPI Master and the data reading transfer

The sensor's datasheet explains how to use this binary number. As stated, the first two bits sent are the current status of the sensor that will be the 2 most significant bits in the number and are read by right-shifting 14 positions, then compared with their referencing values. The remaining 14 bits are used after placing to 00 the status bits; this wouldn't be necessary in the status=ok condition since those bits are 00 anyway. The sensor's datasheet provides the way to convert the number in a floating point value in the range of 0 to 5 N, from the already calibrated transfer function that relates the minimum force value to a minimum digital value, the maximum force value to a maximum digital value and the points in between with a linear approximation. The two limits are there to avoid the nonlinearities at the edges of the linear approximation.

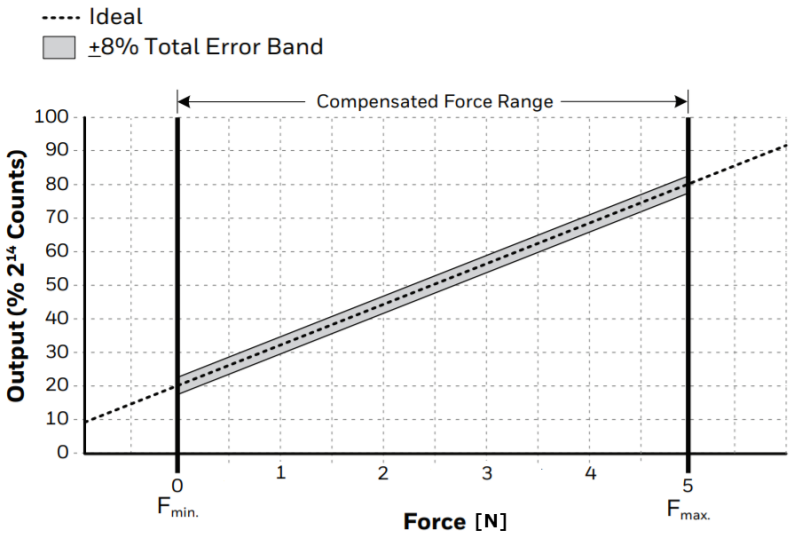

Figure 5.8: Compensated and calibrated transfer function of the sensor as linear relation

A recalibration is done from experimental results since the compensation at 20% and 80% was not working as expected. A lower value of 15% is needed to take the actual minimum force as 0 N when the sensor is not in contact with anything.

Name	Description	Value
Conversion equation	$force = RANGE * \frac{(Reading - MIN)}{(MAX - MIN)}$	Force [N]
Reading	Transferred from sensor	Variable
MIN	15% of 2^{14}	2470
MAX	80% of 2^{14}	13107
RANGE	Rated maximum force	5 N

Table 5.6: Definition of constants and implementation of the transfer function equation to compute the force value

The function in Arduino code also handles the bad status condition by returning a conventional value of -1 to notify it.

```
// Sensor's bits data format, minimum, maximum, rated
const int STATUS_REMOVE = 0x3FFF;      //2 most significant bits of data = 001..1
const int FORCE_LEN = 14;  //14 least significant bits of reading
const int OUTPUT_MAX = 0x3333;  //0.8*2^14 calibrated at max force
const int OUTPUT_MIN = 2470;  //0.15*2^14 calibrated at min force (experimental)
const int FORCE_RATED = 5;      //Newton max force

/*
 * compute force in N from 16bits of reading
 * Uses formula from FMA datasheet
 * Min and max are experimental, not rated from datasheet (min is not 0.2*2^14)
 * Also checks if status bits are ok
 */
float computeForce(unsigned int result16){

  unsigned int forceBits = OUTPUT_MIN;
  byte statusBits = 0;
  float forceMeasure = 0.0;

  //Separate the 2 most significant bits of status, right shift of 14
  statusBits = result16 >> FORCE_LEN;

  if (statusBits == 0) {  // status = 00: ok
```

```
24      // remove first 2 bits of data read putting them to 0
25      forceBits = result16 & STATUS_REMOVE;
26
27      // compute force value in Newton
28      if (forceBits>=OUTPUT_MIN)
29        forceMeasure = FORCE_RATED*(float)(forceBits - OUTPUT_MIN)/(
      OUTPUT_MAX - OUTPUT_MIN);
30      else
31        forceMeasure = 0.0;
32    }
33
34    if (statusBits == 2) {  // status = 10: stale data
35      forceMeasure = -1;
36    }
37
38    if (statusBits == 3) {  // status = 11: sensor broken
39      forceMeasure = -1;
40    }
41
42    return forceMeasure;
43  }
```

Listing 5.2: Arduino code implementation of the transfer function and the handling of the sensor's status

5.3 Bluetooth ROS node

Since the sensor and the microcontroller are placed on top of the robot's end-effector a physical wiring to the central computer would be not suitable during the movements, then leveraging the Bluetooth capabilities of the Arduino NANO 33 BLE it is possible to send the data via wireless. Bluetooth Low Energy is a communication technology that performs a short-range broadcast radio transmission while being optimized to minimum power consumption. The two active elements of the protocol are the peripheral device and the central device as they are respectively the server and the client. The first is the one that advertises the data keeping it ready to be sent and updating it periodically; the latter is instead the one that asks for the required information in an asynchronous way and receives the current contents. Describing the format of the advertised data, it is composed of a number of layers down to the raw information. First, the peripheral profile is unique to each physical device and it's identified by a MAC address; then a service from that device is identified by a service UUID and groups different coherent data; specifying with a characteristic UUID it is possible to access the single piece of data; finally the raw value is read knowing its data type. Here UUID is Universally Unique IDentifier while the MAC address is Media Access Control.

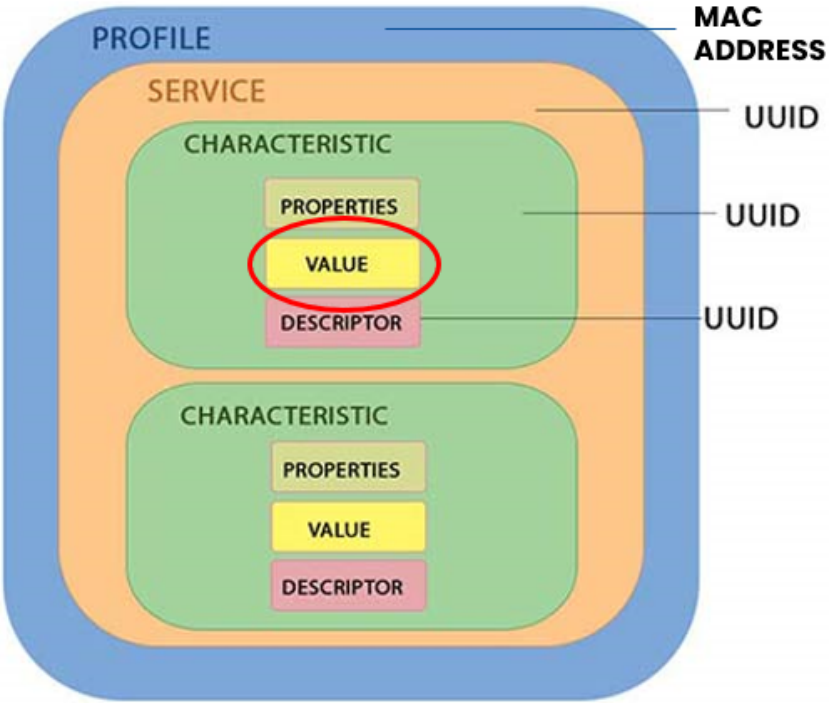

Figure 5.9: Scheme of layers in BLE advertisement

In this system the peripheral device is the Arduino that is programmed with chosen identifier to start the advertisement of a single float value, while the central device is the computer that accesses to the data by connecting with the known IDs. Using the specific ArduinoBLE.h library it is possible to set up and manage the connection. At first the library is imported and the parameters are set; then the structure of the advertisement is programmed by specifying the device name (can be used as a local non unique substitute to the MAC address), the service standard type as float and the single characteristic standard type as float number after choosing a UUID for both; finally it starts advertising from an initial value.

```
1  // library of Bluetooth connection
2  #include <ArduinoBLE.h>
3
4  // Define BLE IDs and type of communication
5  BLEService floatService("180C");  // User defined service
6  BLEFloatCharacteristic floatCharacteristic("2A56",  // standard
       16-bit characteristic UUID
7      BLERead); // remote clients will only be able to read this
8
9  void setup() {
10   // start the BLE library
11   if (!BLE.begin()) {   // initialize BLE
```

```
12      while (true);
13    }
14
15    // initialize BLE comunication
16    BLE.setLocalName("Nano33BLE");  // Set name for connection
17    BLE.setAdvertisedService(floatService); // Advertise service
18    floatService.addCharacteristic(floatCharacteristic); // Add
       characteristic to service
19    BLE.addService(floatService); // Add service
20    floatCharacteristic.setValue(0.0); // Set init value
21    BLE.advertise();  // Start advertising
22  }
```

Listing 5.3: Bluetooth constant parameters setup and advertisement info

The main loop of the program is then the following , where the sensor readings and consequent updates of characteristic are executed only if a central device is connected: there is no use of taking measurements when no one is listening.

```
24  void loop() {
25    unsigned int result16 = OUTPUT_MIN;
26    float forceMeasure = 0.0;
27
28    BLEDevice central = BLE.central();  // Wait for a BLE central to
       connect
29
30    // if a central is connected to the peripheral, send sensor
       values
31    if (central) {
32      // keep looping while connected
33      while (central.connected()){
34
35        result16 = readSPIsensor();
36
37        forceMeasure = computeForce(result16);
38
39        //Write on bluetooth the new value
40        floatCharacteristic.writeValue(forceMeasure);
41
42        //force wait for next iteration, 100ms -> 10 Hz
43        delay(100);
44
45      }
46    }
47  }
```

Listing 5.4: Main loop that connects to Bluetooth, reads sensor, computes force and advertises the data

Noteworthy is the frequency of the updates that is 10 Hz forced by the delay of 100 milliseconds, this is a tradeoff between a max value that must account for sensor reading speed and central Bluetooth reading speed and a minimum value that is experimentally found from pushing the tower and retracting in time to avoid its fall. Central device acts through a ROS node that connects in asynchron to the peripheral and reads the characteristic by knowing all the unique IDs; it is performed through Bleak Python module that can be used to program the device's Bluetooth resources in a script and hide the low level details. A property of the central reader is that it inquires the advertisement but never reads the same value twice no matter the frequency, but instead the it waits for an update of the measurement and collects it as soon as it changes. It then publishes the value on a dedicated ROS topic for the rest of the system.

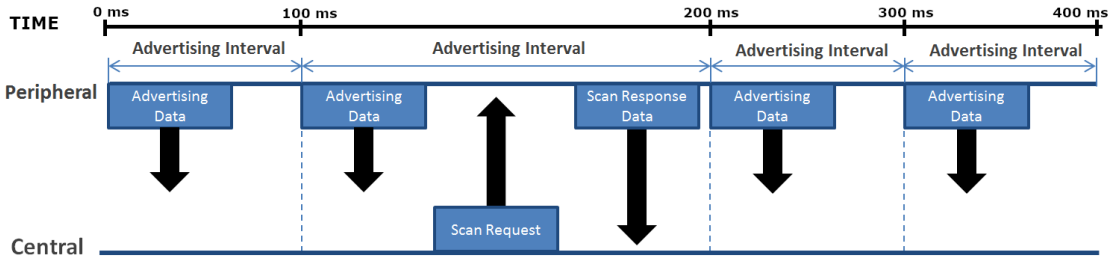

Figure 5.10: Advertisement asynchronous protocol with periodic updates and arbitrary request

```
import rospy
from std_msgs.msg import Float32
import asyncio
import struct
from bleak import BleakClient

#Address of device, always the same for same board
ADDRESS = "A9:88:08:2C:E9:C6"
#UUID chosen inside Arduino sketch,16bit standard
nanoUUID="2A56"
#Arduino standard full 128bit ID
charUUID="0000"+nanoUUID+"-0000-1000-8000-00805f9b34fb"
#Service ID chosen inside Arduino sketch,16bit standard
nanoService="180C"

async def publish_sensor(mac_addr: str):
    client = BleakClient(mac_addr)
    await client.connect()

    #If connected, initialize ROS node, message, topic
```

```
21      forceMsg=Float32()
22      pub = rospy.Publisher('sensor_topic', Float32, queue_size=10)
23      rospy.init_node('sensor_publisher', anonymous=False)
24      while not rospy.is_shutdown():
25          reading=bytes(await client.read_gatt_char(charUUID))
26          value = round(struct.unpack('f',reading)[0], 2)
27          forceMsg.data=value;
28          pub.publish(forceMsg)
29      else:
30          await client.disconnect()
31
32  if __name__ == '__main__':
33      try:
34          loop = asyncio.get_event_loop()
35          loop.run_until_complete(publish_sensor(ADDRESS))
36      except rospy.ROSInterruptException:
37          pass
```

Listing 5.5: Python code of the Bluetooth reader and ROS publisher

5.4 Experiment for force threshold value

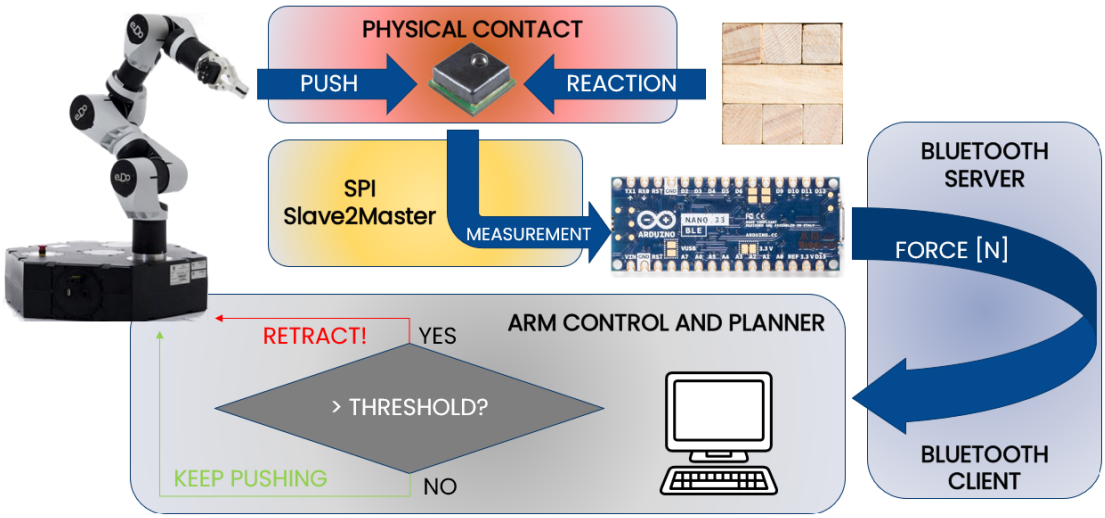

Figure 5.11: Complete scheme of the force feedback system, its different interfaces and final output

The complete force feedback requires a threshold value to perform a binary choice: when the push is encountering too much opposing force then the block is considered constrained and it would make the tower fall, while if the force is

instead very low the block is considered free to move. This is a static number that must be the maximum allowed force. All this translates in the need for practical experiments to take real measurements and analyze the tower's behaviour. As previously explained the ideal static and dynamic cannot be used and a theoretical study of possible values is excluded.

Experiment setup is the following:

- Sensor on the robot's finger
- Computer reading force values via Bluetooth from Arduino
- Tower with 16 floors and full 48 blocks
- Tower is destroyed and rebuilt after every set of measurements
- Robot's movement is not affected by measurements
- Manually moving the tower to ensure the right push position and orthogonality.
- Constrained block: pushing it moves the rest of the tower
- Free block: pushing it moves the target only

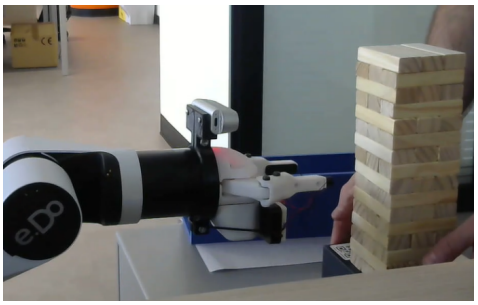

Figure 5.12: Start

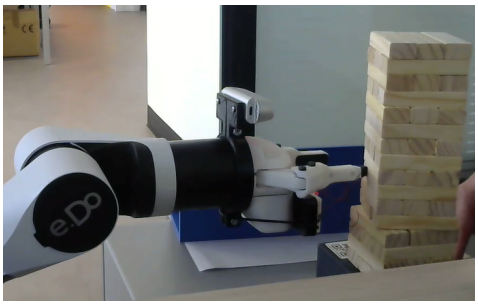

Figure 5.13: Perturbation of the tower

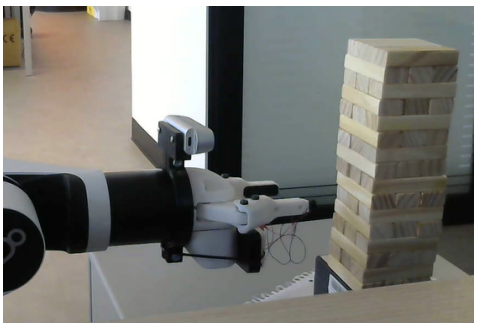

Figure 5.14: Start

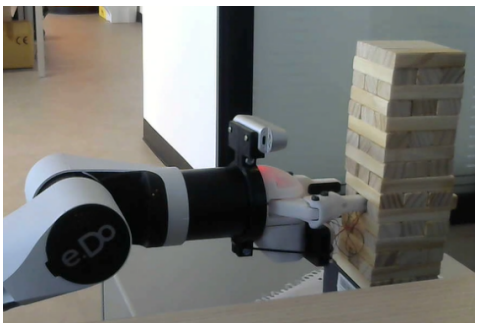

Figure 5.15: Free moving block

The experiment is then conducted by controlling the arm to do two separate movements: from starting position move directly straight ahead until a final position; move back to starting position. Both the positions are manually defined and the arm's movement is a purely offline control with no knowledge of the surrounding for a direct push. To obtain a threshold that accounts for many tower configurations and for a general target position, the pushes are performed on all three blocks forming a floor and from the fourth to the tenth floor. This ensure that both blocks that are in the center and external of the floor are considered since the first have two blocks in contact on their sides while the latter have a free side, thus possibly having different friction values. Furthermore different floors have different weight above them and the friction depends on it. Finally after measuring each three of the floor, the tower is rebuilt to change configuration of smaller and larger blocks.

Blocks per floor	3
Total floors	16
Measured floors	4 to 10
Times floors are measured	3
TOTAL MEASUREMENTS	63

Table 5.7: Experiment setup data

Central computer reads the measurements and writes them on a file in the sequence they are obtained. Supervising the experiment, for each file the actual block's condition is noted: constrained or free. After gathering all the data a plot is computed and from it a threshold value is decided by keeping all the peaks of the free blocks' under it and all the peaks of the constrained blocks above it. The peak values are considered as they correspond to the maximum friction force that is the static friction; the following smaller values are either dynamic frictions or due to the tower rotating or falling under the disruptive push.

In conclusion of the experiment another element is considered: a trade-off between arm response speed and minimization of false positives. A high threshold can always detect free blocks even if they offer a bit of friction, but it causes the arm to retract only when a large force is detected that could already be affecting the stability of the tower.

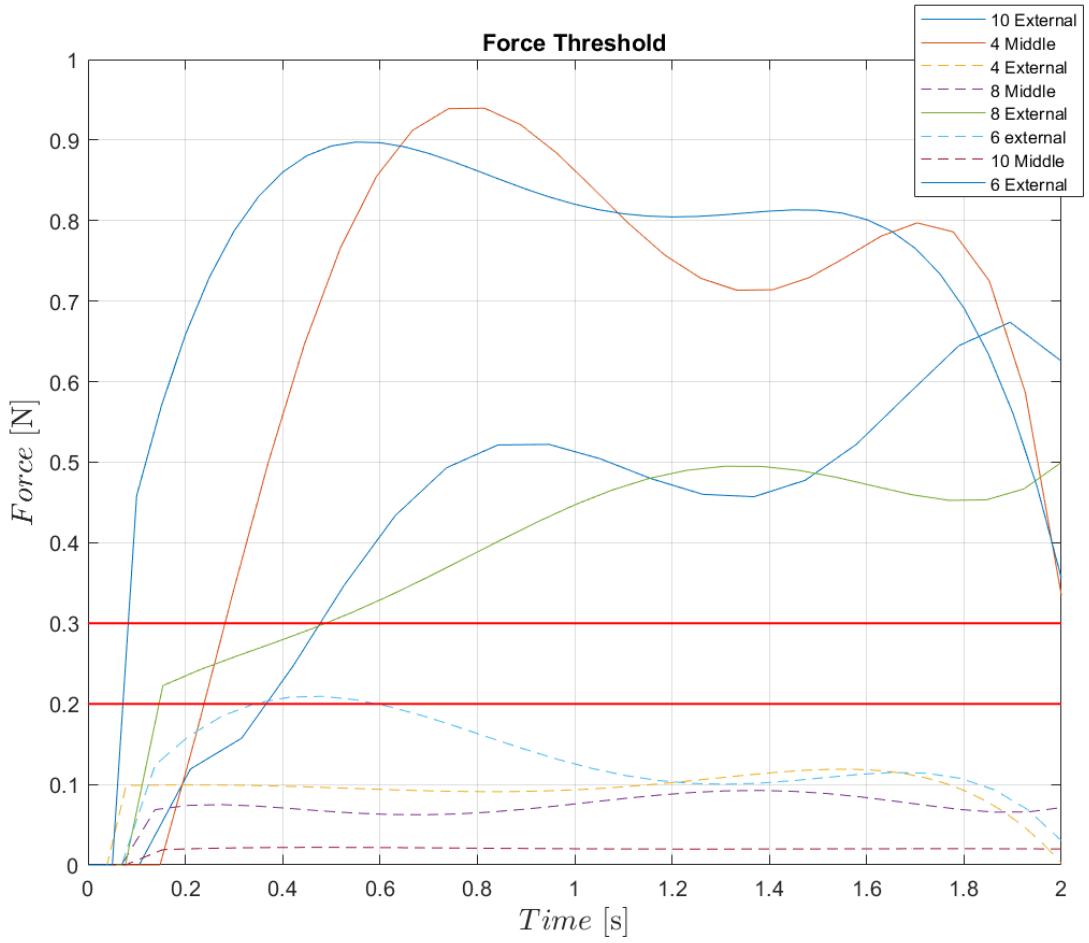

Figure 5.16: Constrained blocks and free blocks compared to the threshold

Chapter 6

Conclusions

6.1 Units integration and final results

The content of 4 assumed to know both the composition of the CAD model group of blocks and its initial position and orientation with respect to the camera reference frame as up to now they were manually provided by the user. Even the automatic detection and the learning phase require to first start the tracker with the correct initial information from which the keypoint features are built.

The already trained Yolact instance segmentation of chapter 3, being a more powerful and generalized system, is employed to perform the first detection for a starting guess of pose estimation. Thanks to the training on generalized data it can provide an inference on a real scene with completely new Jenga blocks with little external information. Noteworthy is that one image is processed in around 2 seconds (or 2000 ms to compare with previous timings); this computation cannot be done in real-time and neither every few frame when the tracking is lost, but really only once per target block.

As shown in section 3.4 the extraction of all the corners of the front faces can be performed with confidence, while the neighbours of a target block are searched and found. Adding here the specific information of the Jenga block's size and with the camera intrinsic parameters the initialization data is computed autonomously.

In particular at first the CAD model .cao format text file is written where each line of the group file as discussed in section 4.4 corresponds to a model of one block filling the three properties values, that are fully known. The block's visibility type is forced by the neighbour search group composition. Based on the content of the up, down, left, right indices:

- If the target has a block on its right and a block on its left, it is a central
- If the target has two blocks on its right, it is a left (external)
- If the target has two blocks on its left, it is a right (external)
- The above and below blocks are always left blocks by construction.

This results in the production of a string name pointing to the source model of the single block, for each component of the group. Second, the translation with respect to the target block is based on the block's real dimensions and it descends from the group composition, with their fixed values. Third and finally, the rotation is either null or 90° on the second component.

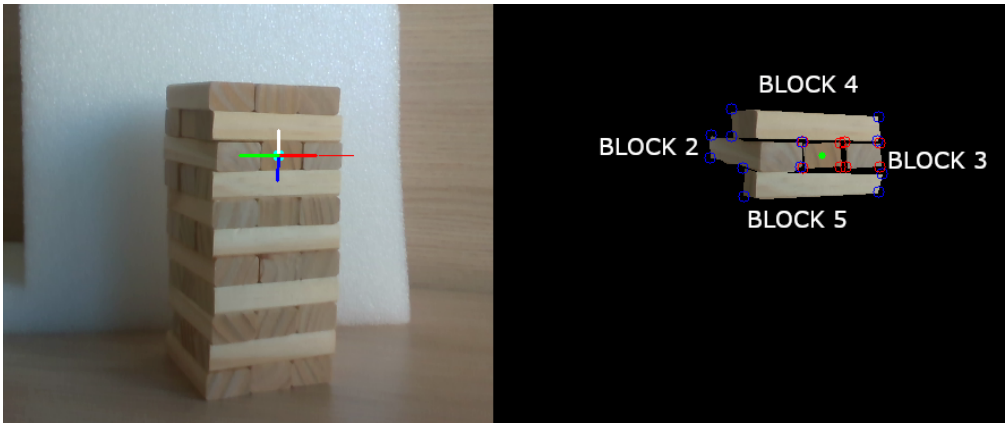

Figure 6.1: Group found from target block at the center of the floor

```
# Block 1
load("jenga_Center.cao", t=[0.0; 0.0; 0.0], tu=[0.0; 0.0; 0.0])

# Block 2
load("jenga_Right.cao", t=[-0.025; 0.0; 0.0], tu=[0.0; 0.0; 0.0])

# Block 3
load("jenga_Left.cao", t=[0.025; 0.0; 0.0], tu=[0.0; 0.0; 0.0])

# Block 4
load("jenga_Left.cao", t=[-0.0375; -0.015; 0.0125], tu=[0;90 deg;0])

# Block 5
load("jenga_Left.cao", t=[-0.0375; 0.015; 0.0125], tu=[0;90 deg;0])
```

Listing 6.1: Automatic generated CAD

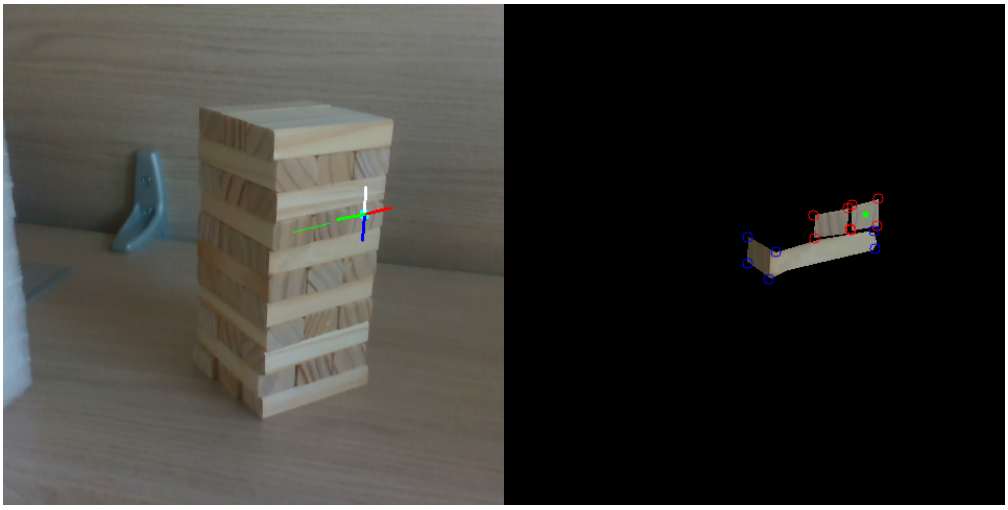

Figure 6.2: Group found from target block at the side, with wrong detections

```
# Block 1
load("jenga_Left.cao", t=[0.0; 0.0; 0.0], tu=[0.0; 0.0; 0.0 deg])

# Block 2
load("jenga_Center.cao", t=[-0.025; 0.0; 0.0], tu=[0.0; 0.0; 0 deg])

# Block 3
load("jenga_Left.cao", t=[-0.0625; 0.015;  0.0125], tu=[0.0;  90.0
    deg; 0.0])
```

Listing 6.2: Generated CAO starting from block on the side

Initial pose of the target is computed with an OpenCV Perspective-n-Points function containing an algorithm specifically designed for coplanar points, as it is in this case, called IPPE (Infinitesimal Plane-based Pose Estimation), based on the projection equations of 4.1. The function needs the points in 3D object frame and the camera intrinsic parameters, already known. The solution is the 6 unknown extrinsic values of the pose of the object with respect to the camera from 8 sets of equations. The requirement of having at least 4 points is guaranteed as the front face of the block is defined by the 4 corners, while in the case of two adjacent front faces the available points become the 8 corners (they are actually only 6 because 2 corners of the other face are already included in the first set and don't add new information).

The association between 2D in pixels and 3D in object frame is perfectly known and RANSAC iterations are not needed as the consensus is 100%. All other corners, colored in blue, do not have a direct correspondence known a priori and therefore

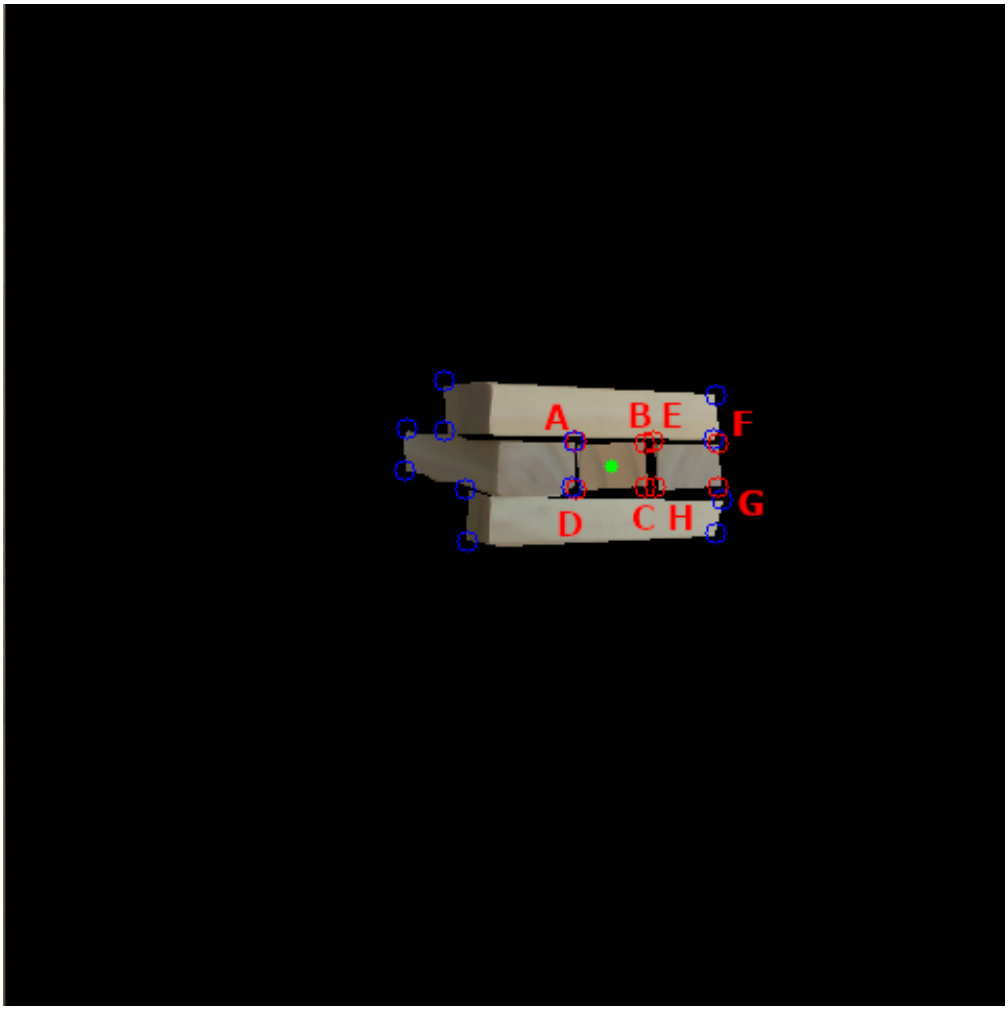

Figure 6.3: The 8 points inputs for the PnP IPPE

Point(px)	Coordinates(m)
A	(-0.0125,-0.0125)
...	...
H	(0.0375,-0.0125)

Table 6.1: Associating pixels to object coordinate frame, coplanar with z=0

are ignored; in any case PnP with 4 points is enough for a unique solution with a good quality to start the tracking and the stronger model-based system discussed previously.

The complete algorithm for each new scene is shown here.

Algorithm 3 From a new tower configuration, producing a real-time pose estimation and tracking during the camera approach to the tower and the target block.

```
1: procedure SEGMENT LEARN AND APPROACH(New-tower-scene)
2:     Unknown new tower configuration of blocks
3:     Camera facing towards the tower
4:     Take a single picture
5:     Full tower segmentations
6:     Finding all groups of blocks
7:     Choose one group
8:     Produce CAO model file and initial pose
9:     Initialize tracking
10:    for learnedPictures<6 do
11:        if TrackingLost then
12:            ExitWithError
13:        end if
14:        Reach new position
15:        Take picture
16:        Extract keypoints
17:        Keep only those on target
18:        Learn keypoints and 3D info
19:    end for
20:    while BlockNotReached do
21:        Move toward block
22:        if TrackingLost then
23:            Extract Keypoints
24:            Match with learned images
25:            Restart tracking
26:        end if
27:        Estimate pose of target
28:    end while
29: end procedure
```

In the following pages, some experiments are made with the tracking, always by moving manually the camera.

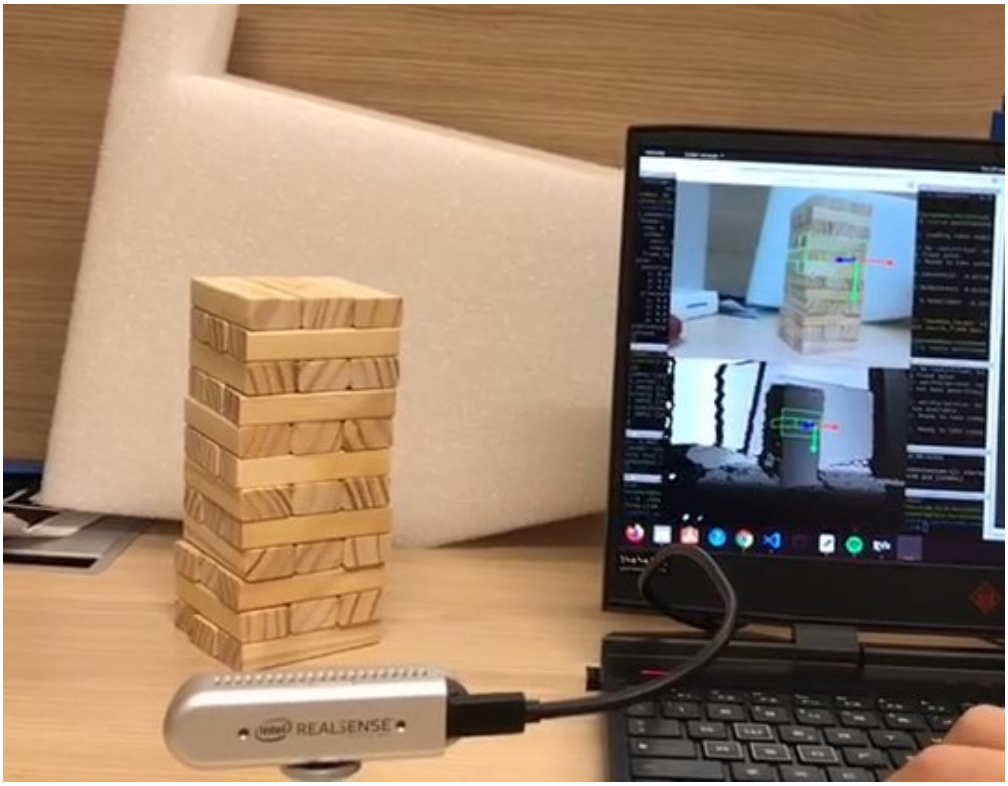

Figure 6.4: Experimental setup with camera, tower and tracking active

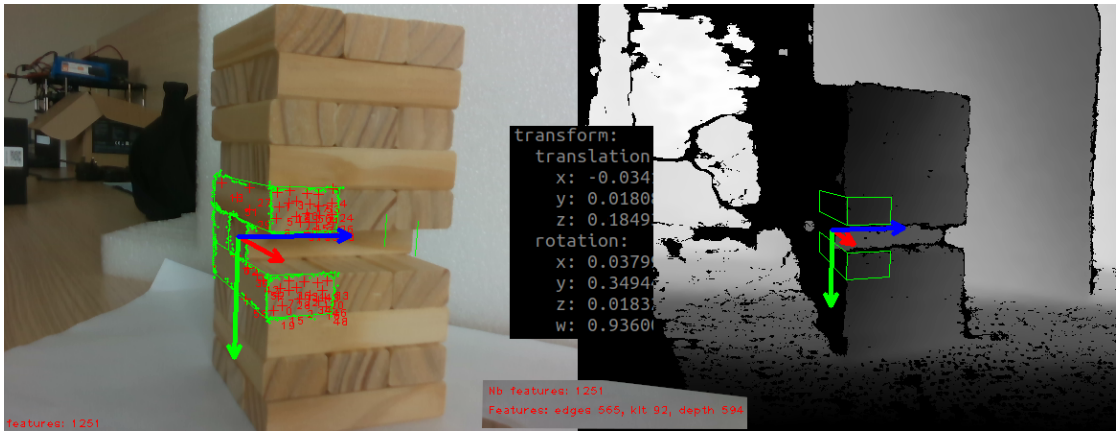

Figure 6.5: Only the tracking and pose estimation, with high number of features.

Mean time	54 ms
Std deviation	10 ms

Table 6.2: Processing time during tracking, without learning and detecting

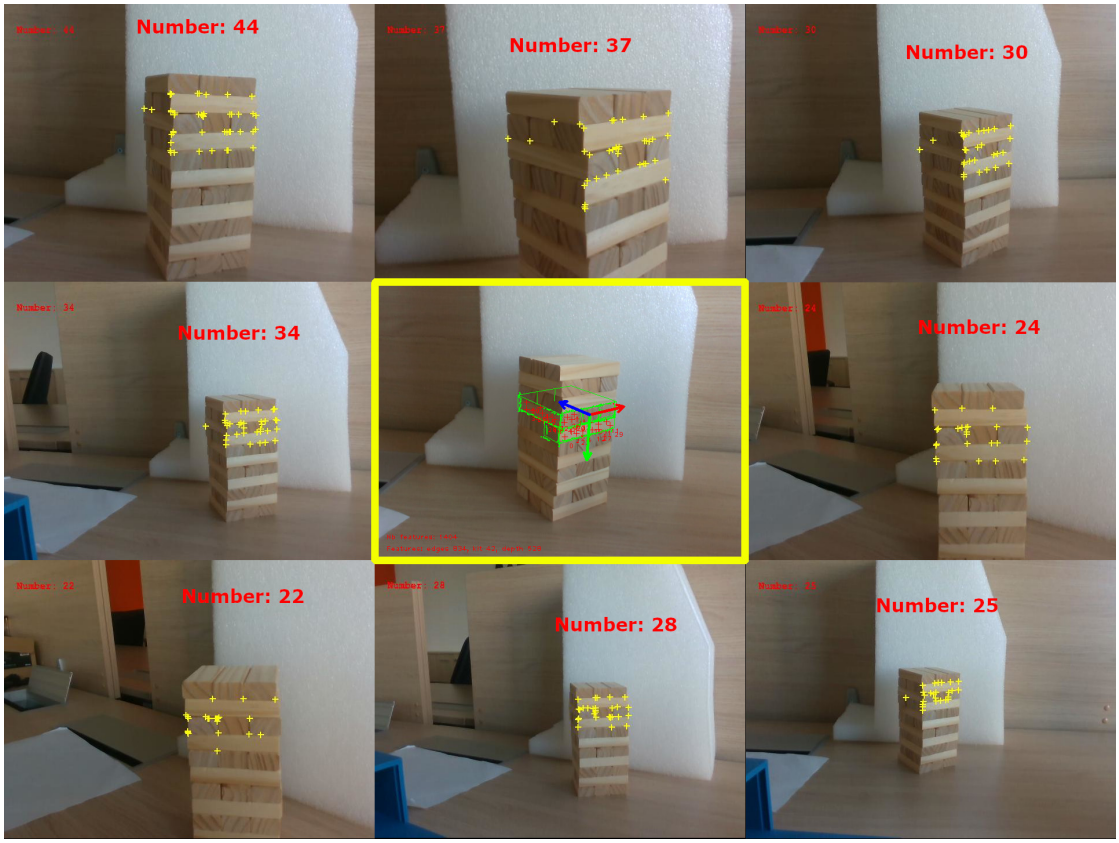

Figure 6.6: Learning images with saved keypoints shown, during active tracking

Processing time of tracking	Mean: 55 ms Std: 19 ms
Learning timing	Mean: 71 ms Std: 12 ms

Table 6.3: Processing time of all the frames compared with the 8 extraction and learning

Since the tracking operates on every single frame while the learning stage is only activated 6 or 8 times, the mean time of the first is not affected while the standard deviation is slightly increased by those events. Table 6.3 shows that the learning stage on a single image takes a considerable amount of time, more than single track.

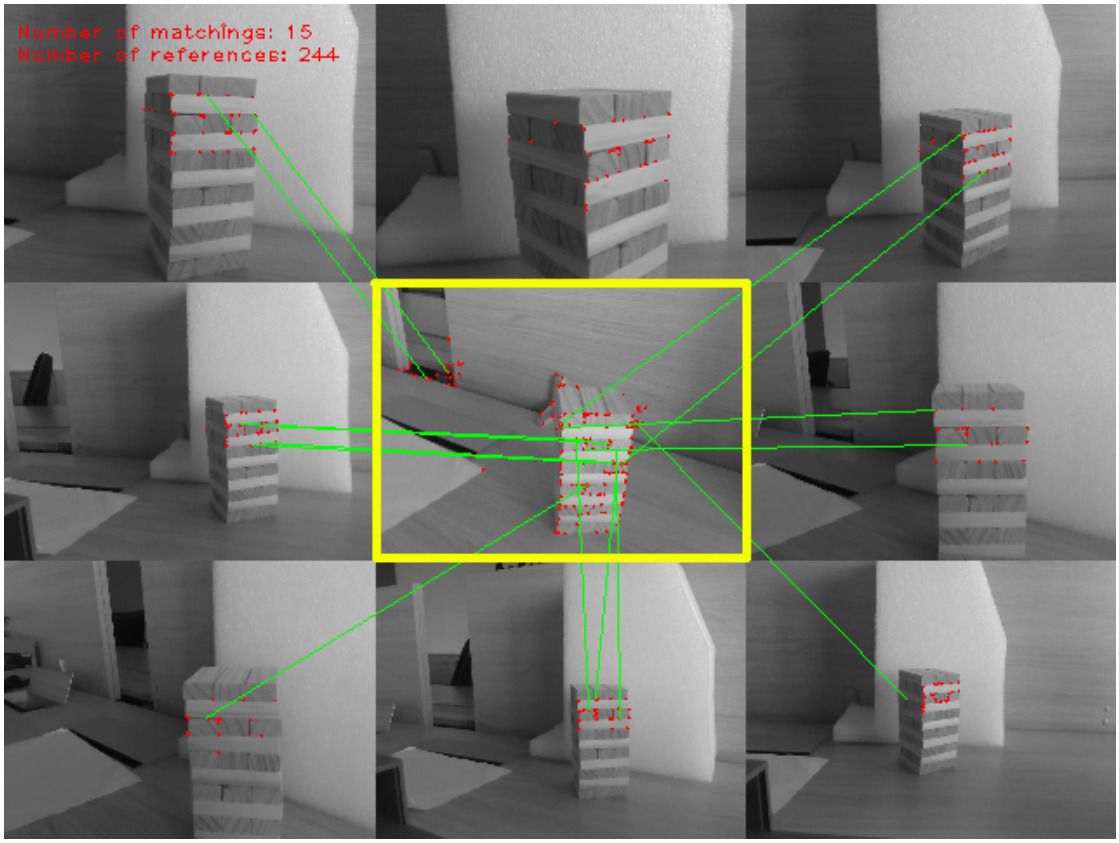

Figure 6.7: Matching from the previously learned images

Processing time with detections	Mean: 56 ms Std: 24 ms
Detection timing	Mean: 78 ms Std: 5 ms

Table 6.4: Processing time of detecting, extracting and matching when tracking is lost

The negative effect on the overall processing time depends on how stable is the tracking and how many times it loses the position. The detections and re-initializations are very heavy on computation.

To obtain a very robust system, even rotations of the camera should be considered.

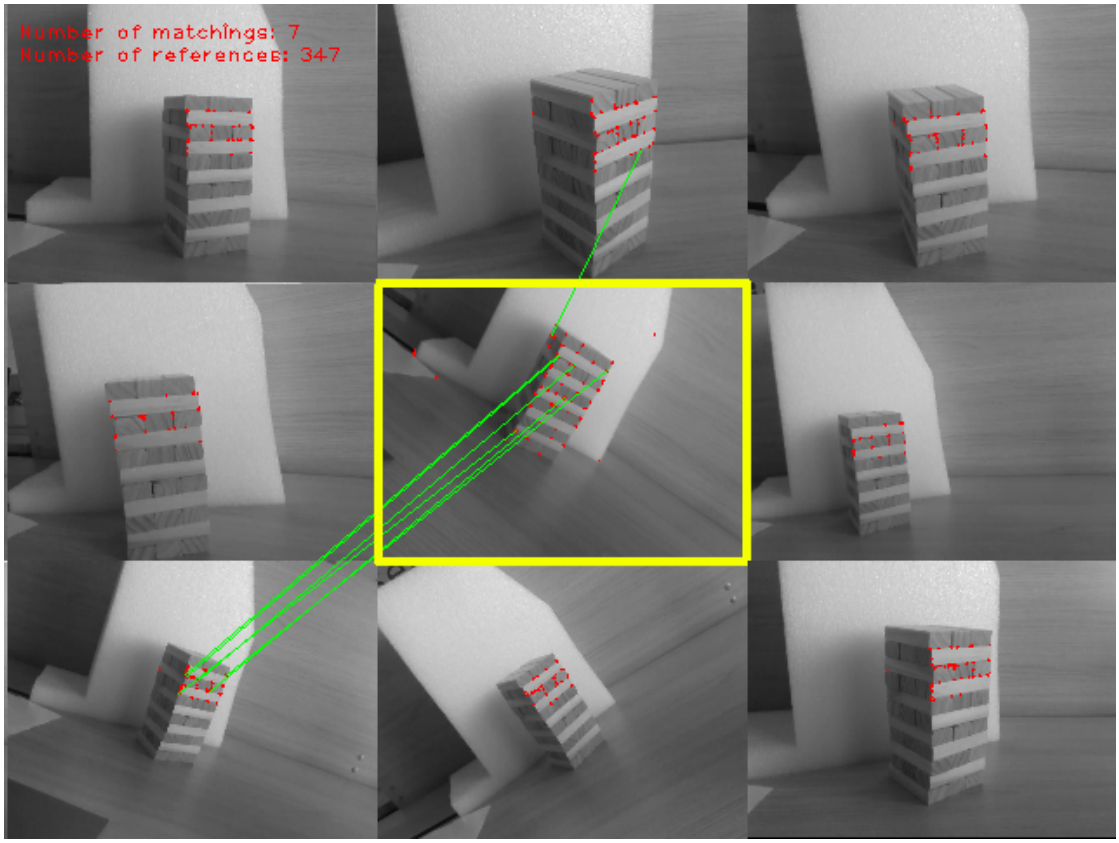

Figure 6.8: Matching from other 8 images with rotated camera learned images. Current camera image is at the center. Only 7/347 matched points

6.2 Future work

The final integrations to the manipulator's movement control require two additional steps. The first, calibrating the extrinsic parameters of the fixed pose of the camera with respect to the end-effector or the robot's base, is done through a camera calibration algorithm specific for the configuration chosen. If eye-to-hand, camera fixed and pose to the robot's base. If eye-in-hand like in this case, camera moving together with the end-effector, mounted on the robot's wrist, and pose with respect to the finger. The second is to adopt a Position-Based Visual Servoing with eye-in-hand approach that, from the estimated pose of the object in the camera frame, computes a velocity the camera should obtain to finally reach the target. Since the camera is on top of the robot, it translated into velocity commands of the robot's hand.

Bibliography

[1] N. Fazeli, M. Oller, J. Wu, Z. Wu, J. B. Tenenbaum, and A. Rodriguez. «See, feel, act: Hierarchical learning for complex manipulation skills with multisensory fusion». In: *Science Robotics* 4.26 (2019), eaav3123. DOI: 10.1126/scirobotics.aav3123. URL: https://www.science.org/doi/abs/10.1126/scirobotics.aav3123 (cit. on pp. 3, 60).

[2] Philip Rogers, Lonnie Parker, Douglas Brooks, and Mike Stilman. «Robot Jenga: Autonomous and Strategic Block Extraction». In: Nov. 2009, pp. 5248–5253. DOI: 10.1109/IROS.2009.5354303 (cit. on p. 4).

[3] OpenCV. *Open Source Computer Vision Library*. 2015 (cit. on p. 5).

[4] Tsung-Yi Lin et al. *Microsoft COCO: Common Objects in Context*. 2015. arXiv: 1405.0312 [cs.CV] (cit. on p. 6).

[5] Adam Kelly. *Create coco annotations from scratch*. Apr. 2020. URL: https://www.immersivelimit.com/tutorials/create-coco-annotations-from-scratch (cit. on p. 7).

[6] Maximilian Denninger, Martin Sundermeyer, Dominik Winkelbauer, Youssef Zidan, Dmitry Olefir, Mohamad Elbadrawy, Ahsan Lodhi, and Harinandan Katam. «BlenderProc». In: *arXiv preprint arXiv:1911.01911* (2019) (cit. on p. 9).

[7] Adam Kelly. *Train yolact with a custom Coco Dataset*. Mar. 2020. URL: https://www.immersivelimit.com/tutorials/train-yolact-with-a-custom-coco-dataset (cit. on pp. 21, 25).

[8] CloudFactory. *Image annotation for computer vision*. URL: https://www.cloudfactory.com/image-annotation-guide (cit. on p. 24).

[9] Daniel Bolya, Chong Zhou, Fanyi Xiao, and Yong Jae Lee. «YOLACT: Real-time Instance Segmentation». In: *ICCV*. 2019 (cit. on p. 24).

[10] SpaceView. *SpaceView/yolactcpu*. June 2021. URL: https://github.com/SpaceView/yolact_cpu (cit. on p. 27).

[11] *Contour features*. URL: https://docs.opencv.org/3.4.16/dd/d49/tutorial_py_contour_features.html (cit. on p. 28).

[12] *Harris corner detection*. URL: https://docs.opencv.org/3.4/dc/d0d/tutorial_py_features_harris.html (cit. on p. 28).

[13] *Random sample consensus*. Nov. 2021. URL: https://en.wikipedia.org/wiki/Random_sample_consensus (cit. on pp. 34, 35).

[14] E. Marchand, F. Spindler, and F. Chaumette. «ViSP for visual servoing: a generic software platform with a wide class of robot control skills». In: *IEEE Robotics and Automation Magazine* 12.4 (Dec. 2005), pp. 40–52 (cit. on p. 40).

[15] S. Trinh, F. Spindler, E. Marchand, and F. Chaumette. «A modular framework for model-based visual tracking using edge, texture and depth features». In: *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems, IROS 18*. Madrid, Spain, Oct. 2018 (cit. on p. 40).

[16] Fabien Spindler. *Model-based generic tracker tutorial*. Jan. 2019. URL: https://visp-doc.inria.fr/doxygen/visp-daily/tutorial-tracking-mb-generic.html (cit. on p. 40).

[17] Satya Mallick. *Feature based image alignment using opencv cpp-python*. Mar. 2018. URL: https://learnopencv.com/image-alignment-feature-based-using-opencv-c-python (cit. on p. 54).

[18] Fabien Spindler. *Object detection and localization*. Feb. 2015. URL: https://visp-doc.inria.fr/doxygen/visp-daily/tutorial-detection-object.html (cit. on p. 55).

[19] E. Marchand and F. Chaumette. «Virtual visual servoing: A framework for real-time augmented reality». In: *EUROGRAPHICS 2002 Conference Proceeding*. Ed. by Drettakis, G., Seidel, and H.-P. Vol. 21(3). Saarebrün, Germany, Germany, 2002, pp. 289–298. URL: https://hal.inria.fr/inria-00352096 (cit. on p. 56).