

Master's Thesis

Masters in Automatic Control and Robotics (MUAR)

Perception and Reasoning for Automatic Configuration of Task and Motion Planning Problems

Report

September 23, 2021

Autor: Fatma Nur Arabaci

Directors: Prof. Dr. Joan Rosell Gratacos

Convocatòria: 10/2021



Escola Tècnica Superior
d'Enginyeria Industrial de Barcelona



Summary

This thesis proposes a framework for configuring Task Planning Problems flexibly in an automatic manner using two main modules which are the Perception Module and the Reasoning Module.

In order to automatize the overall process, initially, a knowledge layer is generated manually, in which the information regarding the environment is stored using ontologies, whereas the environmental state where the task is taking place is observed with the help of the Perception Module. The knowledge layer is then reasoned within the Reasoner Module in order to automatically configure task planning problems by filling Planning Domain Definition Language (PDDL) [1] files. During this reasoning process, the information retrieved from the Perception Module is used.

In this paper, both of these modules mentioned above are explained in detail before providing the results separately for each module. Then, in addition to individual results, a scenario is created within a lab environment to test the overall system including both modules. Furthermore, alternative areas where the Reasoning Module implementation can be benefited from is also discussed.

Contents

1	Preface	7
1.1	Origin of the Project	7
1.2	Prerequisites	7
2	Introduction	9
2.1	Objective of Thesis	9
2.2	State-of-the-Art	10
2.3	Scope of Thesis	11
3	Perception Module	13
3.1	Deep Object Pose Estimation	13
3.1.1	Unreal Engine 4	13
3.1.2	NVIDIA Deep learning Dataset Synthesizer (NDDS) Plug-in	14
3.1.3	Synthetic Training Data Generation	16
3.1.4	Training	22
3.1.5	ROS Interface	27
4	Reasoning Module	35
4.1	Perception	35
4.2	Reasoning Framework	36
4.2.1	Knowledge Layer	36
4.2.2	Ontology-based reasoning for robot manipulation	38
4.3	PDDL Parser	41
4.4	Automatic Generation of Domain PDDL File	41
4.5	Automatic Generation of Problem PDDL File	42
4.6	Test Scenarios	42
4.6.1	Tiago Kitchen Scenario	42
4.6.2	Tiago & Yumi Counter Scenario	44
5	Results	47
5.1	Deep Object Pose Estimation	47
5.2	Reasoning Module	48
6	Cost Analysis	53
7	Environmental & Social Impact	55
7.1	Environmental Impact	55
7.2	Social Impact	55
	Conclusions	59
	Acknowledgments	61
	Appendices	62

List of Figures

3.1	An Example Image Generated using NDDS, with Ground Truth, Segmentation, Depth, and Object poses	14
3.2	Example Annotation File	15
3.3	NVCapturableActorTag for the Object of Interest	16
3.4	NDDS Build Error	17
3.5	NDDS Build Error	18
3.6	NDDS Plug-in	19
3.7	Material Instance Script	20
3.8	Robotics Lab of the Institute of Industrial and Control Engineering (IOC-UPC) Environment Setup in Unreal Engine	21
3.9	UnderBody Model Visualized in Blender	22
3.10	UnderBody Model with Color Included	23
3.11	The <i>object.json</i> File	23
3.12	Scene Capturer Settings	24
3.13	Feature Extraction Options Selected for the Scene Capturer	24
3.14	Toy Plane Model Parts used as Distractors	25
3.15	Simple Collision Visualization for UnderBody Model	26
3.16	<i>Open Level Blueprint</i> Menu Access	27
3.17	Blueprint Structure to Spawn the UnderBody Model in the Environment	28
3.18	Blueprint Structure to Spawn the TopWing Model in the Environment	29
3.19	Example Photo-realistic Images	30
3.20	Example <i>nvdv_viz</i> for Photo-realistic Images	31
3.21	Lighting Randomization Parameters	32
3.22	Created Setup to Obtain Domain Randomized Images	32
3.23	Example Images Obtained with Domain Randomization	33
3.24	<i>camera_info</i> Topic for Kinect XBOX Camera	33
3.25	The <i>config_pose.yaml</i> File	34
4.1	SWI-Prolog Interface	38
4.2	Flowchart of the overall symbolic reasoning operation	40
4.3	Knowledge Management Schema	41
4.4	Tiago Kitchen Test Environment	43
4.5	Yumi Manipulation Task	45
5.1	Loss Graph for Training	47
5.2	Example Target Images Saved in Training for Epoch 1,3 and 60	48
5.3	Target & Output Belief Map Comparisons after 1 Epoch of Training	49
5.4	Target & Output Belief Map Comparisons after 3 Epoch of Training	49
5.5	Target & Output Belief Map Comparisons after 60 Epochs of Training	50
5.6	Validation Code Trial 1	50
5.7	Validation Code Trial 2	51
A.1	Model Hierarchy Graph for Tiago & Yumi Ontology	64
A.2	Instantiation Knowledge for Tiago & Yumi Ontology	65

List of Tables

6.1	<i>Cost table (Variable cost of workstation PC has been computed dividing their fixed cost by their life expectancy in hours. Variable cost of electric consumption of each system have been computed by multiplying the power consumption by the average price of electricity as described in the Cost section)</i>	53
-----	--	----

1 Preface

1.1 Origin of the Project

The origin of this thesis is to enhance the existing robotics applications that are currently being used in the Robotics Lab of the Institute of Industrial and Control Engineering (IOC-UPC). The motivation of this study can be divided into two main parts, one of which is the object detection process within the lab and the other is regarding the configuration of the Task Planning problem files.

Currently, in Robotics Lab of the Institute of Industrial and Control Engineering (IOC-UPC), the objects are detected using different variety of Fiducial Markers. Even though the usage of Fiducial markers to detect objects within the task environment gives us high precision results with high speed, one issue faced during the detection is, the struggle to detect the markers when there is an occlusion between the objects and the camera. Considering the tasks that are planned to be executed in IOC Robotics Lab, possible occlusions between the task related objects can occur. In addition to the occlusions, the Fiducial markers may not be fully visible by the cameras due to the placement of the marker on the object and the camera poses with respect to the object itself. One possible enhancement in this case is introducing an object pose estimation module which can detect the bounding boxes of the objects, even with partial occlusions, within the task environment.

Furthermore, currently in the lab, Task Planning Problems domain and problem files are created manually considering the task description. This task description includes assumptions regarding the state of the environment such as the available objects and agents. However, this implementation requires an adaptation of the PDDL files for different tasks which consist of divergent elements and goals to be achieved within the environment. Moreover, this current application is not feasible for the cases where the state of the environment is dynamic during the task execution, since the PDDL files remain the same throughout the implementation and the planner is called once in the beginning of the execution. Possible robotics applications in which re-generation of PDDL files, through the inclusion of the reasoning concept, can be benefited from will be discussed further in this work.

1.2 Prerequisites

In this thesis, the perception module studied is the adaptation to IOC Robotics Lab environment of the Deep Object Pose Estimation, DOPE, project explained in the paper [2] and the available source code in the repository [3]. For this purpose, Unreal Engine 4 [4] platform is adopted for the generation of synthetic training data with the help of NVIDIA Deep Learning Dataset Synthesizer, NDDS, plug-in [5]. Training and Validation codes are also provided within the same GitHub repository mentioned above. One important aspect is that this thesis is not the exact replicate of the study explained in the original paper, but an adaptation of it considering the resources available in the lab environment. The same project also provides a Robotic Operation System, ROS [6], Inference by which the perception module can easily be integrated with the current applications within the lab.

In addition to the Perception Module, within the Reasoning Module, Protege [7] ontology editor interface is used to create the knowledge layer which consists of the records of the environmental entities. Ontologies are used to describe the knowledge in the form of concepts considering the relationships among these concepts. The reasoning process is implemented with the help of the

SWI-Prolog [8] compiler and ROS [6] services created specifically for this task. These services retrieve information from the knowledge layer by querying over the ontologies and then with the help of a PDDL-Parser package [9], task specific PDDL files are generated considering the output of the reasoner.

2 Introduction

Some of the required capabilities for mobile manipulators to make them able to autonomously move and work in semi structured human environments are:

- Smart perception capabilities that are able to, not only perceive, but to understand the current state of the environment (including the robot itself).
- Adaptive planning capabilities that are able to: a) plan at task levels according to the current state of the environment and to the goal to be achieved, updating the initial state and choosing the actions the robot can do to solve the problem; b) plan at motion level according to the current poses of the objects, modifying grasping configurations if necessary, object placement poses or the robot base location, and choosing the most appropriate motion planner.
- Robust execution capabilities that are able to integrate both the smart perception capabilities and the adaptive planning capabilities to avoid failures or, if failures occur, be able to reason on the failed state in order to plan the best recovery strategy.

In this thesis, first two of these capabilities are aimed to be integrated in a single framework to flexibly configure task specific problem files.

In order to execute a specific task, a sequence of actions should be determined, considering the given initial and goal conditions of the entities within the environment, assuming that all the executable actions are deterministic but not stochastic. In this concept, classical planning approaches can be adopted in which the problem files are represented using the Planning Domain Definition Language (PDDL). This language consists of the definition of pre and post conditions of each possible action in addition to their effects. PDDL can be grouped into two main files, one for the domain description and one for the problem description for objects initial states and goal specifications.

Compliance with the dynamic state of the environment enhances the robot capabilities. For this purpose, generation of a knowledge layer is essential, in such a manner, a reasoning framework can be utilized on this layer to adapt the PDDL file contents regarding the perceived environment. Therefore, knowledge needs to be structured such that it is usable for reasoning tasks and, in this line, ontologies arise as hierarchical structures expressing the universe of discourse based on relations, such as *is-a* and *has-a*, between concepts and instances of classes, being these concepts, instances, and relations expressed in formal languages.

2.1 Objective of Thesis

The main objective of this thesis can be grouped into two main parts as follows:

- Enhancing the current perception structure based on detecting Fiducial markers placed on the objects of interest within the IOC Robotics Lab in ETSEIB by introducing a more robust and efficient object detection application.
- Configuring task planning problems with the help of a reasoning framework considering the current state of the environment where the task will be executed. This configuration includes the selection of feasible actions from a given global domain. Information regard-

ing the current state of the environment is aimed to be retrieved from the perception module solution suggested above. With this implementation, the closed-world assumption is discarded and more flexibility is expected to be achieved.

2.2 State-of-the-Art

In this section, previous research regarding both perception for pose estimation and reasoning using ontologies is discussed.

The state-of-the-art methods for object pose estimation consist of either the classical Fiducial marker detection or inclusion of Deep Learning with the help of computer vision. There are different types of Fiducial Markers available in robotics field such as ArUco [10], ARToolkit [11], ARTag [12], and AprilTag [13] markers. In IOC Lab, ArUco tags are currently being used and AprilTag & ARTag are also being studied for further enhancements. Each of these markers have different hand-crafted features that are designed to be robust to be used in detection and classification tasks. Even though classical Fiducial markers provide high precision results, they do not always perform well in natural environments. These natural environments define the circumstances where the pose estimation algorithm is expected to operate. Possible interference that can be present in detection and classification applications are the scaling, possible occlusions between the objects, motion-blur due to a non-fixed camera or moving objects, and off-axis viewing which is described as looking at a display from an angle that is at least one degree away from center. There are studies concerning these drawbacks of using Fiducial markers such as Fractal Markers [14] that are designed to tackle the occlusion issue. Another solution suggested is about the motion blur challenge which is especially a concern in quad-copters, since the markers on them are subject to sudden and unstable motion throughout the operation [15].

Due to all these downsides of the classical marker detection applications, introduction of deep learning is being studied recently, since with the appropriate training, they can be more robust to the above mentioned difficulties. There are many research regarding this topic such as [16], [17], [18]. They mainly differ in the selection of the neural network architecture or the loss function. One of the advantages of deep learning in pose estimation is that they benefit from having a huge training dataset. However, generation of this dataset is expensive to acquire since it requires the labeling of the objects of interest. Therefore, the concept of synthetic data is introduced into the field to deal with this problem. With the help of synthetic data, the labeling cost can be reduced since the labels come automatically during the generation process. However, using only synthetic data for training process causes a gap called 'Domain Gap' between the real world and simulation environments considering that after the training process, the network is expected to perform within real world applications. A network trained with fully synthetic data will not perform well in real world due to this gap. In order to solve this problem, Domain Adaptation and Domain Randomization [19] concepts are introduced to close this Domain Gap. For this purpose photo-realistic rendering platforms are being used to generate realistic training data to close the 'Appearance Gap' which is a reason why Domain Gap exists. Domain Gap also occurs due to a concept called 'Content Gap' which is because synthetic content imitates a limited set of scenes, not necessarily reflecting the diversity and distribution of objects of those captured in the real world [20]. One related work to close these mentioned gaps is Deep Object Pose Estimation for Semantic Robotic Grasping of Household Objects [2] which is the starting point for the perception module implementation in this thesis. It is adapted to the Robotics Lab of the Institute of Industrial and Control Engineering (IOC-UPC) and the steps

followed are explained in details in Section 3.

The state-of-the-art methods for using knowledge in planning with ontologies involve many domains such as manipulation domain [21], [22] which aims to find motion plans that are desired to be adopted regarding the obstacles in the environment satisfying certain constraints (i.e. which obstacles can be collided, from where and with which interacting force), the navigation domain [23], [24] in which, knowledge is used to build a representation of the environment combining the metric information needed for navigation tasks with the symbolic information that stores the context of the aspects within the environment, or perception domain for manipulation [25] in which the observed environment via sensors is stored in the form of ontologies to improve manipulation tasks planning. A review of the use of ontologies to give robot autonomy can be found in [26].

2.3 Scope of Thesis

This work studies the use of ontologies, through a suggested Reasoning Module, in solving manipulation problems, given the information regarding the objects within the environment which is detected through the Perception Module, predefined manipulation constraints, in addition to the available agents and their capabilities. Therefore, creation of task specific PDDL files is required so that the manipulation problem can be solved with any classical task planner.

With the purpose of detecting the aspects within the environment, so that the suggested Reasoning Module interprets the current state of the environment, a smart perception module implementation based on deep neural networks is adopted. By doing so, current object detection algorithms based on marker detection is planned to be enhanced.

In order to automatically generate task specific PDDL files, an initial global PDDL file, which is provided manually, including all possible actions the agents can execute and the predicates, is forwarded into the PDDL parser tool, which is described in Section 4.3, to store PDDL content. From this global PDDL content, only the task specific actions and agents, which are provided as the output of the reasoner through the framework explained in Section 4.2, are written into the task specific PDDL file by the same PDDL parser tool. The final PDDL file is used in higher level tasks such as Task and Motion Planning (TAMP).

The structure of the manuscript is as follows. In Section 3, the Perception Module which is implemented following the study explained in Section 3.1 is described. Then, Reasoning Module is explained in Section 4 including the background about Ontologies in Robot Manipulation and Knowledge Layer. The test scenarios generated for the Reasoning Module are specified in Section 4.6 and the results for both of the modules are provided in Section 5.

3 Perception Module

In robotics manipulation tasks, detection of objects of interest in the environment with high accuracy is essential to perform the specified task which includes variety of interactions with the detected objects. The detection includes the position and the orientation which is also called as the 6 DoF pose of these objects. Different methods are being used for 3D object detection and pose estimation such as Fiducial Markers or Deep Neural Networks which is taken as reference for the Perception Module implementation in this thesis. Therefore, Sections between 3.1.1 - 3.1.5 are dedicated to the implementation of a deep neural network in the Robotics Lab of the Institute of Industrial and Control Engineering (IOC-UPC) for object detection.

3.1 Deep Object Pose Estimation

Inclusion of deep neural networks into 6 DoF pose detection algorithms used to suffer from the lack of enough training data, since generation of ground-truth for the objects used to be costly before the introduction of synthetic data concept in which data already comes with the labels. In addition to facilitating the training of deep neural networks, establishment of synthetic data concept permit carrying out the training with a dataset that is not correlated with the test data which is usually captured with real cameras in real world setup.

On the other hand, one of the main concerns in using synthetic training data is that the neural networks that are trained with synthetic data only, does not perform well in real data due to the content gap between real and synthetic images. In this work, a concept called Domain Randomization is being implemented to try to close this Sim-to-Real gap, which is the difference between the real domain and virtual domain, and enable the network perform well also in real world without fine-tuning. In addition to domain randomized synthetic data, realistic training data is also benefited from in training the network to introduce the relevant complexity of real world scenes. In order to generate synthetic data, Unreal Engine and NVIDIA Deep learning Dataset Synthesizer (NDDS) plug-in are chosen which are discussed in Section 3.1.1 & 3.1.2. Environment created in Unreal Engine in which photo-realistic synthetic data is captured is described in Section 3.1.3.1 and the generation process of both of photo-realistic and domain randomized datasets is explained in Section 3.1.3.

3.1.1 Unreal Engine 4

Unreal Engine is a game engine developed by Epic Games [27]. Its newest version is Unreal Engine 4. In addition to its wide usage within gaming industry, it is also used in film making and virtual reality applications due to its wide range of features and plug-ins. Information regarding available features and the releases are published regularly on their website [4].

Unreal Engine is a feasible solution to synthetic image generation since photo-realistic images can be synthesized with the help of its high-performance rendering capabilities [28]. It also allows implementing other concepts such as Domain Randomization which is widely used within the synthetic data generation process in order to close the Sim-to-Real gap with the help of available plug-ins. For this purpose, a specific Unreal Engine plug-in named NVIDIA Deep learning Dataset Synthesizer (NDDS) [5] which will be discussed in the next section is used.

In order to install Unreal Engine 4, steps in [29] are followed, initially, to gain access to the Unreal Engine repository on GitHub. After these steps, desired version of Unreal Engine 4 source code can be cloned and built following the steps in the README file in Unreal Engine

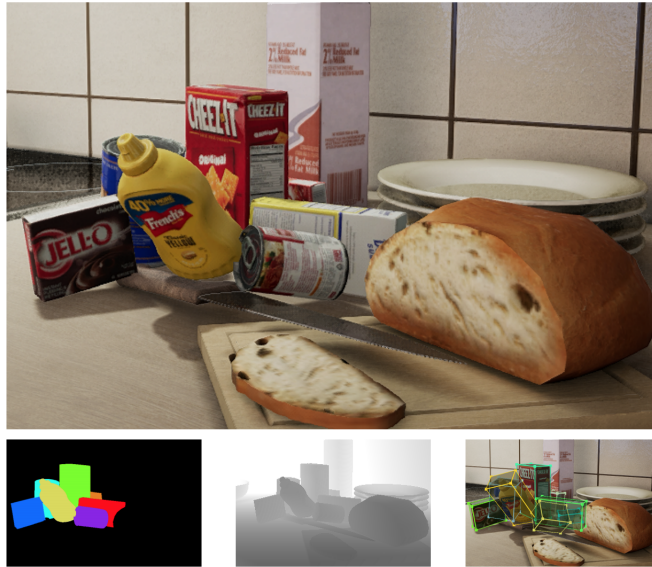


Figure 3.1: An Example Image Generated using NDDS, with Ground Truth, Segmentation, Depth, and Object poses

GitHub repository. In this thesis, Unreal Engine version 4.25.4 is installed and built.

3.1.2 NVIDIA Deep learning Dataset Synthesizer (NDDS) Plug-in

NDDS is a plug-in for Unreal Engine 4, developed by NVIDIA, to export synthetic images with ground truth labels. In addition to the RGB images, the following data can also be generated within the plug-in:

- Depth Images
- Class Segmentation Images
- Instance Segmentation Images
- Bounding Box of the objects of interest
- Poses of objects of interest

Examples of the above mentioned data can be seen in Figure 3.1. The plug-in also includes tools to implement Domain Randomization within the environment by randomizing lighting, camera pose, object poses, objects textures and also by adding distractors into the scene. With all these included features, synthetic training data can easily be generated combined with metadata. The metadata is also called as *Annotation File* in NDDS which provides the pose, class ID and the visibility (0 meaning fully occluded and 1 meaning fully visible) of the objects in an RGB image generated using the plug-in. An example of this annotation file can be seen in Figure 3.2.

In order to create this ground truth annotation files, NDDS provides a tool named *NVCapturable-ActorTag* which is added to objects of interest as in Figure 3.3. After adding the tag to the object and naming it from the *Tag* entry as shown Figure 3.3, *Include Me* option should be selected so that the tag is detected during the image capturing.

```

"camera_data":
{
  "location_worldframe": [ -731.73681640625, 279.64370727539062, 525.9033203125 ],
  "quaternion_xyzw_worldframe": [ -0.38019999861717224, 0.012099999934434891, 0.9243999719619751, 0.02930000051856041 ]
},
"objects": [
{
  "class": "Under",
  "instance_id": 0,
  "visibility": 0.7466999888420105,
  "location": [ 50.004398345947266, 5.7588000297546387, 58.060901641845703 ],
  "quaternion_xyzw": [ -0.34900000691413879, -0.28580000996589661, -0.52609997987747192, 0.72100001573562622 ],
  "pose_transform": [
    [ -0.045000001788139343, 0.80390000343322754, 0.59299999475479126, 0 ],
    [ 0.28310000896453857, -0.55900001525878906, 0.77929997444152832, 0 ],
    [ -0.95800000429153442, -0.20290000736713409, 0.20250000059604645, 0 ],
    [ 50.004398345947266, 5.7588000297546387, 58.060901641845703, 1 ]
  ],
  "cuboid_centroid": [ 49.430000305175781, 5.7353000640869141, 58.275100708007812 ],
  "projected_cuboid_centroid": [ 473.14401245117188, 281.19601440429688 ],
  "bounding_box":
  {
    "top_left": [ 259.2034912109375, 424.37289428710938 ],
    "bottom_right": [ 308.04110717773438, 533.84991455078125 ]
  },
  "cuboid": [
    [ 40.915500640869141, 5.7321000099182129, 64.140800476074219 ],
    [ 39.69110107421875, 8.1497001647949219, 60.770401000976562 ],
    [ 57.597400665283203, 11.942999839782715, 56.986099243164062 ],
    [ 58.821800231933594, 9.5253000259399414, 60.356498718261719 ],
    [ 41.262599945068359, -0.4724000096321106, 59.564098358154297 ],
    [ 40.038200378417969, 1.9452999830245972, 56.193801879882812 ],
    [ 57.944499969482422, 5.738599772216797, 52.409500122070312 ],
    [ 59.168998718261719, 3.3208999633789062, 55.779800415039062 ]
  ],
  "projected_cuboid": [
    [ 419.30300903320312, 278.87869262695312 ],
    [ 423.20150756835938, 290.33209228515625 ],
    [ 514.7462158203125, 309.652587890625 ],
    [ 505.4906005859375, 296.40188598632812 ],
    [ 433.34219360351562, 253.97039794921875 ],
    [ 438.40078735351562, 264.86300659179688 ],
    [ 539.0369873046875, 284.03170776367188 ],
    [ 527.55438232421875, 271.24191284179688 ]
  ]
}
]
]

```

Figure 3.2: Example Annotation File

There are two other alternatives to NDDS in labeled synthetic data exportation. One of which is the default Unreal Engine 4 screenshot function and the other alternative is the publicly available UnrealCV tool [30]. However, since NDDS leverages asynchronous, multithreaded sequential frame grabbing, the plug-in generates data at a rate of 50-100 Hz, which is significantly faster than the mentioned alternatives.

In this thesis, NDDS plug-in is built from [31], so that it is compatible with the installed Unreal Engine which is version 4.25.4. Even though, version 4.26 for NDDS plug-in is also available which is compatible with the newest Unreal Engine version (4.26), this version of NDDS plug-in is said to be not compatible with Linux. Since our computer setup is a Debian system, NDDS plug-in version 4.25.4 is selected for this thesis.

After cloning the project from the provided repository, steps below are followed to build the plug-in. Because when NDDS.uproject is opened via Unreal Engine the error in Figure 3.4 & Figure 3.5 occurs. Therefore, Visual Code Studio is used for building the project from the source. The steps below are followed within Visual Code Studio to do so:

1. Create the workspace by running `GenerateProjectFiles.sh /path_to_uproject_file -game -engine` in a terminal. This .sh file is located under Unreal Engine project folder.

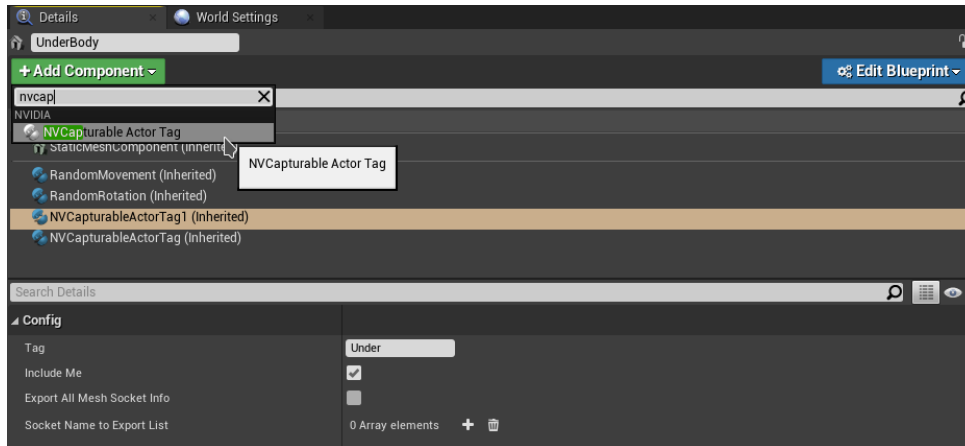


Figure 3.3: NVCapturableActorTag for the Object of Interest

2. Open created .workspace file in Visual Code Studio.
3. Ctrl + Shift + B and type *NDDSEditor Linux Development Build* to build the code from source

Then, *NDDS.uproject* file can be opened via Unreal Engine 4 and the NDDS plug-in with its sub-packages (Domain Randomization for Deep Neural Networks, Fast Scene Capturer, NVIDIA Data Object, and NVIDIA Utilities) is visible under the available plug-ins as shown in Figure 3.6.

3.1.3 Synthetic Training Data Generation

Synthetic training dataset consists of two main parts which are the photo-realistic images and domain randomized images as mentioned earlier. In the Deep Object Pose Estimation paper followed, using equal amounts of Domain Randomized and Photo-realistic images is suggested, considering other trials with different proportions. Therefore, the same strategy is applied in this thesis. Even though both of these datasets are generated within Unreal Engine 4 and NDDS plug-in, different environmental preparation is required for each case.

3.1.3.1 Environmental Setup

In order to create photo-realistic synthetic images, a realistic environment in Unreal Engine should be created. One way to do so is to use already existing environments in Unreal Engine Marketplace [32] which offers both free and paid realistic environments. The assets of these environments can be sent to Unreal Engine through Epic Games Launcher. However, Epic Games Launcher is not offered for a Debian platform. Therefore, an intermediate interface is required to install Epic Games Launcher into a Debian operating system. This can be achieved by using an open source gaming platform called Lutris [33] for Linux. Yet, in this thesis, Robotics Lab of the Institute of Industrial and Control Engineering (IOC-UPC) environment is decided to be created within Unreal Engine and different locations within the lab is selected for realistic data generation.

Meshes of the objects within the lab were already created to generate scenes to be used in Kautham [34], therefore, the same meshes are converted into wavefront object [35] files and im-

```

fatma.nur@sputnik: /srv/robotica/UnrealEngine.git/Engine/Binaries/Linux
fatma.nur@sputnik: /srv/robotica/UnrealEngine.git/Engine/Binaries/Linux 225x51
LogInit: Number of monitors: 1
LogInit: Monitor 0
LogInit: Name: 03279W658 33"
LogInit: ID: display0
LogInit: NativeWidth: 2560
LogInit: NativeHeight: 1440
LogInit: IsPrimary: true
[2021.02.11-13.26.14:00][453]LogExit: Exiting.
[2021.02.11-13.26.14:00][453]LogInit: Tearing down SDL.
fatma.nur@sputnik: /srv/robotica/UnrealEngine.git/Engine/Binaries/Linux$ STUBBED: FDesktopPlatformLinux::GetNativeFeedbackContext at /srv/robotica/UnrealEngine.git/Engine/Source/Developer/DesktopPlatform/Private/Linux/DesktopPlatformLinux.cpp:450 (GetNativeFeedbackContext)
Running /srv/robotica/UnrealEngine.git/Engine/Binaries/DotNET/UnrealBuildTool.exe Development Linux -Project="/srv/robotica/Dataset_Synthesizer/Source/NDDS.uproject" -TargetType=Editor -Progress -NoEngineChanges -NoHotReload
romIDE
chmod: changing permissions of 'bin/mcs': Operation not permitted
chmod: changing permissions of 'bin/xbuild': Operation not permitted
Running Mono...
Fixing inconsistent case in filenames.
Setting up Mono
/srv/robotica/UnrealEngine.git/Engine /srv/robotica/UnrealEngine.git/Engine/Binaries/Linux
Using 'git status' to determine working set for adaptive non-unity build (/srv/robotica/UnrealEngine.git).
Using 'git status' to determine working set for adaptive non-unity build (/srv/robotica/Dataset_Synthesizer).
Creating makefile for NDDSEditor (no existing makefile)
$progress push 5%
$progress push 5%
$progress pop
Parsing headers for NDDSEditor
Running UnrealHeaderTool "/srv/robotica/Dataset_Synthesizer/Source/NDDS.uproject" "/srv/robotica/Dataset_Synthesizer/Source/Intermediate/Build/Linux/B40820EA/NDDSEditor/Development/NDDSEditor.uhtmanifest" -LogCmds="loginit
warning, logexit warning, logdatabase error" -Unattended -WarningsAsErrors -abslog="/srv/robotica/UnrealEngine.git/Engine/Programs/UnrealBuildTool/Log_UHT.txt"
LogUnixPlatformFile: Warning: open("/home/users/leopold.palomo/.config/Epic/Epic Games/KeyValueStore.ini", 0_RDONLY | 0_CLOEXEC) failed: errno=13 (Permission denied)
Reflection code generated for NDDSEditor in 3.4570721 seconds
ERROR: Couldn't touch header directories: Access to the path "/srv/robotica/UnrealEngine.git/Engine/Intermediate/Build/Linux/B40820EA/UE4Editor/Inc/FieldSystemCore/TimeStamp" is denied.
LogInit: Warning: Still incompatible or missing module: NDDS
LogInit: Warning: Still incompatible or missing module: NVDataObject
LogInit: Warning: Still incompatible or missing module: NVDataObjectEditor
LogInit: Warning: Still incompatible or missing module: NVUtilities
LogInit: Warning: Still incompatible or missing module: DomainRandomizationDNN
LogInit: Warning: Still incompatible or missing module: NVSceneCapturer
LogInit: Warning: Still incompatible or missing module: NVSceneCapturerGame
LogInit: Warning: Still incompatible or missing module: NVSceneCapturerEditor
LogCore: Engine exit requested (reason: EngineExit() was called)
LogExit: Preparing to exit.
LogModuleManager: Shutting down and abandoning module DesktopPlatform (44)
LogModuleManager: Shutting down and abandoning module PlatformCryptoOpenSSL (42)
LogModuleManager: Shutting down and abandoning module PlatformCryptoTypes (40)
LogModuleManager: Shutting down and abandoning module PlatformCrypto (38)
LogModuleManager: Shutting down and abandoning module AnimationModifiers (36)
LogModuleManager: Shutting down and abandoning module PropertyEditor (35)
LogModuleManager: Shutting down and abandoning module AudioEditor (32)
LogModuleManager: Shutting down and abandoning module TextureCompressor (30)
LogModuleManager: Shutting down and abandoning module ShaderComp (29)

```

Figure 3.4: NDDS Build Error

ported to Unreal Engine. However, in order to create realistic images, the textures of these objects should also be taken into account. Since the original textures of the objects within the environment were not available, realistic texture packs are found online [36]. Chosen textures are added to the objects through creating a new *Layered Material Instance* [37] in UE4.

In order to integrate the textures into the meshes, the structure in Figure 3.7 is being created. The Texture Object blocks in the same figure are the Roughness, Normal and Occlusion from top to bottom. A Scaler, which is 256 in Figure 3.7, is also provided in order to set the resolution of the material. By adjusting this value, the same Material Instance can be used for objects in different scale.

Another important asset in realistic image generation is introducing realistic lighting within the environment. In this setup, no natural day light is taken into consideration since no window structure is created, therefore, only realistic fluorescent lighting is mimicked by using Rect Light asset [38] in UE4 with the color codes R = 253, G = 242, B = 198. These values are selected to mimic a natural lighting in the setup. Images of the created lab environment can be seen in Figure 3.8.

3.1.3.2 Object of Interest

Even though it is possible to train a single set of weights for detecting multiple objects at the same time, in this thesis, only one object is selected which is available in the Robotics Lab of the Institute of Industrial and Control Engineering (IOC-UPC). This object is a part of a toy plane which is already being used in manipulation tasks in the lab. The wavefront object of the model was also available. The model can be seen in Figure 3.9 which does not include color data initially. This feature is added within Unreal Engine 4 later.


```

fatma.nur@sputnik: /srv/robotica/UnrealEngine.git/Engine/Binaries/Linux
LogInit: NativeHeight: 1440
LogInit: bIsPrimary: true
fatma.nur@sputnik:/srv/robotica/UnrealEngine.git/Engine/Binaries/Linux$ STUBBED: FDesktopPlatformLinux::GetNativeFeedbackContext at /srv/robotica/UnrealEngine.git/Engine/Source/Developer/DesktopPlatform/Private/Linux/DesktopPlatformLinu
x.cpp:450 (GetNativeFeedbackContext)
Running /srv/robotica/UnrealEngine.git/Engine/Binaries/DotNET/UnrealBuildTool.exe Development Linux -Project="/srv/robotica/Dataset_Synthesizer/Source/NDDS.uproject" -TargetType=Editor -Progress -NoEngineChanges -NoHotReloadFromIDE
Running Mono...

chmod: changing permissions of 'bin/mcs': Operation not permitted
chmod: changing permissions of 'bin/xbuild': Operation not permitted
Fixing inconsistent case in filenames.
Setting up Mono
/srv/robotica/UnrealEngine.git/Engine /srv/robotica/UnrealEngine.git/Engine/Binaries/Linux
Using 'git status' to determine working set for adaptive non-unity build (/srv/robotica/UnrealEngine.git).
Using 'git status' to determine working set for adaptive non-unity build (/srv/robotica/Dataset_Synthesizer).
Creating makefile for NDDSEditor (no existing makefile).
@progress push 5%
Waiting for 'git status' command to complete
fatma.nur@sputnik:/srv/robotica/UnrealEngine.git/Engine/Binaries/Linux$ @progress push 5%
@progress pop
Parsing headers for NDDSEditor
Running UnrealHeaderTool "/srv/robotica/Dataset_Synthesizer/Source/NDDS.uproject" "/srv/robotica/Dataset_Synthesizer/Source/Intermediate/Build/Linux/B4D828EA/NDDSEditor/Development/NDDSEditor.uhtmanifest" -LogCmds=loginit warning, lo
gexit warning, logdatabase error -Unattended -WarningsAsErrors -abslog="/srv/robotica/UnrealEngine.git/Engine/Programs/UnrealBuildTool/Log_UHT.txt"
LogLinuxPlatformFile: Warning: open("/home/users/topool/paltonor.config/Epic/Epic Games/KeyValueStore.ini", 0_RDONLY | 0_CLOEXEC) failed: errno=13 (Permission denied)
Reflection code generated for NDDSEditor in 3,7286994 seconds
@progress pop
----- Build details -----
Using rootchain located at /srv/robotica/UnrealEngine.git/Engine/Extras/ThirdPartyNotUE/SDKs/HostLinux/Linux_x64/v16_clang-9.0.1-centos7/x86_64-unknown-linux-gnu'.
Using clang (/srv/robotica/UnrealEngine.git/Engine/Extras/ThirdPartyNotUE/SDKs/HostLinux/Linux_x64/v16_clang-9.0.1-centos7/x86_64-unknown-linux-gnu/bin/clang++) version '9.0.1' (string), 9 (major), 0 (minor), 1 (patch)
Using bundled libc++ standard C++ library.
Using lld linker
Using llvm-ar /srv/robotica/UnrealEngine.git/Engine/Extras/ThirdPartyNotUE/SDKs/HostLinux/Linux_x64/v16_clang-9.0.1-centos7/x86_64-unknown-linux-gnu/bin/llvm-ar
Using fast way to relink circularly dependent libraries (no FixDeps).
-----
ERROR: Building would modify the following engine files:
/srv/robotica/UnrealEngine.git/Engine/Intermediate/Build/Linux/B4D828EA/UE4Editor/Development/Engine/SharedPCH_Engine.h.d
/srv/robotica/UnrealEngine.git/Engine/Intermediate/Build/Linux/B4D828EA/UE4Editor/Development/Engine/SharedPCH_Engine.h.gch
/srv/robotica/UnrealEngine.git/Engine/Intermediate/Build/Linux/B4D828EA/UE4Editor/Development/UnrealEd/SharedPCH_UnrealEd.h.d
/srv/robotica/UnrealEngine.git/Engine/Intermediate/Build/Linux/B4D828EA/UE4Editor/Development/UnrealEd/SharedPCH_UnrealEd.h.gch

Please rebuild from an IDE instead.
LogCore: Engine exit requested (reason: EngineExit()) was called)
LogExit: Preparing to exit.
LogModuleManager: Shutting down and abandoning module DesktopPlatform (44)
LogModuleManager: Shutting down and abandoning module PlatformCryptoOpenSSL (42)
LogModuleManager: Shutting down and abandoning module PlatformCryptoTypes (58)
LogModuleManager: Shutting down and abandoning module PlatformCrypto (38)
LogModuleManager: Shutting down and abandoning module AnimationModifiers (36)
LogModuleManager: Shutting down and abandoning module PropertyEditor (35)
LogModuleManager: Shutting down and abandoning module AudioEditor (32)
LogModuleManager: Shutting down and abandoning module TextureCompressor (30)
LogModuleManager: Shutting down and abandoning module RenderCore (28)
LogModuleManager: Shutting down and abandoning module Landscape (26)
LogModuleManager: Shutting down and abandoning module SlateRHIRenderer (24)
LogModuleManager: Shutting down and abandoning module OpenGLDrv (22)
LogModuleManager: Shutting down and abandoning module AnimGraphRuntime (20)
LogModuleManager: Shutting down and abandoning module Renderer (18)
LogModuleManager: Shutting down and abandoning module Engine (16)
LogModuleManager: Shutting down and abandoning module CoreObject (14)
LogModuleManager: Shutting down and abandoning module NetworkFile (12)
LogModuleManager: Shutting down and abandoning module CookedIterativeFile (10)
LogModuleManager: Shutting down and abandoning module StreamingFile (8)
LogModuleManager: Shutting down and abandoning module SandboxFile (6)
LogModuleManager: Shutting down and abandoning module PakFile (4)
LogModuleManager: Shutting down and abandoning module RSA (3)
LogExit: Exiting.
LogInit: Tearing down SDL.
Exiting abnormally (error code: 1)
fatma.nur@sputnik:/srv/robotica/UnrealEngine.git/Engine/Binaries/Linux$

```

Figure 3.5: NDDS Build Error

One way to import the chosen model into UE4 environment is directly through *Import* button right next to *Add New* button. After importing the model into Unreal Engine and including the color codes which are obtained approximately from the images of the real object, the model looks as in Figure 3.10.

One problem encountered when importing the model with this method is the wrong visibility value inside the annotation files. In order to solve this problem, an alternative importing method is followed. For this purpose, a 3D computer graphics software tool Blender [39] is used. In Blender, there is an available add-on called *Send To Unreal* [40] which enables sending models to Unreal Engine 4 directly. Before sending the model to UE4, after importing the model to Blender, the model should be added into the *Mesh* category. In order to do so, after selecting the imported model by clicking on it, button *M* should be pressed on the keyboard and *Mesh* option should be selected from the appearing menu. Later, through the *Pipeline* -> *Export* -> *Send to Unreal* drop-down menu in Blender, the model can be sent to Unreal Engine. The visibility issue is solved when the objects are imported following this method.

The *object.json* file in Figure 3.11 is generated automatically when the scene is started being captured to represent the objects of interest which have the *NVCapturableActorTag* in the created environment. This file can be found in */Dataset_Synthesizer/Source 4.25/NVCapturedData/Test-Capturer* directory. The settings for the selected capturer, an asset of NDDS plug-in, to capture the RGB images with the desired features is explained in the next section.

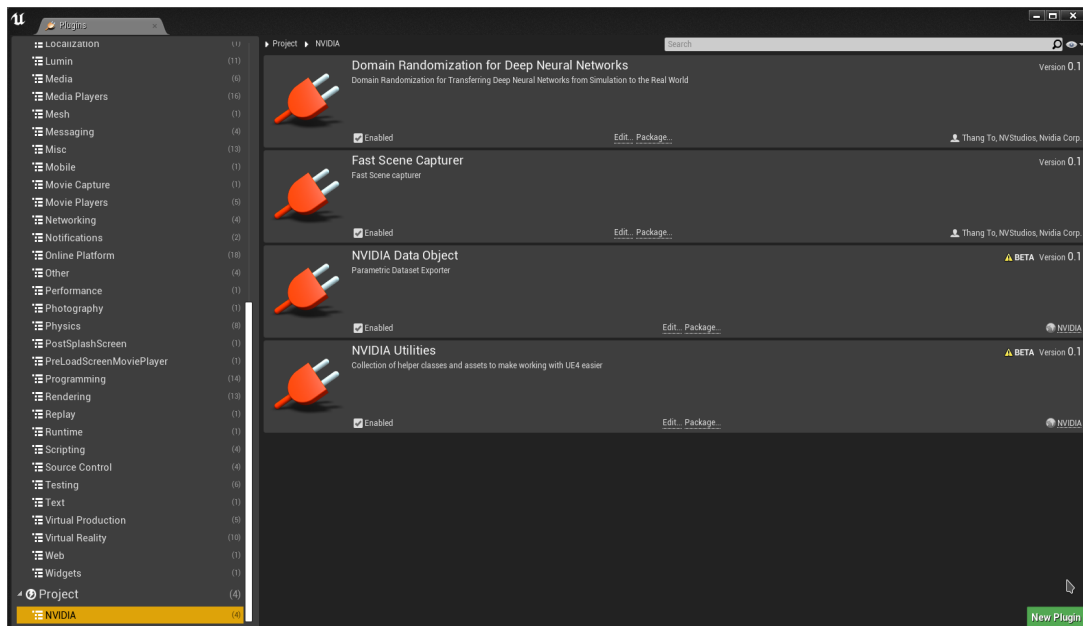


Figure 3.6: NDDS Plug-in

3.1.3.3 Scene Capturer Settings

In order to capture RGB images and desired features, NDDS offers an asset named *NVSceneCapturer*. The settings in Figure 3.12 & 3.13 are chosen to capture the RGB images and the annotation files.

The capturer settings are saved to *camera.json* in */Dataset_Synthesizer/Source 4.25/NVCaptured-Data/TestCapturer* directory automatically when the scene is started being captured.

3.1.3.4 Photo-realistic Image Generation

The Robotics Lab of the Institute of Industrial and Control Engineering (IOC-UPC) setup created in Unreal Engine 4 is used as the environment in this case. Four different locations within the lab are chosen which are:

1. The White Table with the Chessboard
2. The Lab Floor
3. The Wooden Table
4. The White Table without the Chessboard

Locations where the images are captured can be seen in Figure 3.19.

After the selection of places within the lab, the data generation strategy defined in Falling Things Dataset [41] is followed, in which the objects are rendered randomly around the selected locations and they are allowed to fall under gravity. In addition to the object of interest, UnderBody, other parts of the same toy plane, as in Figure 3.14, are also imported into the lab environment

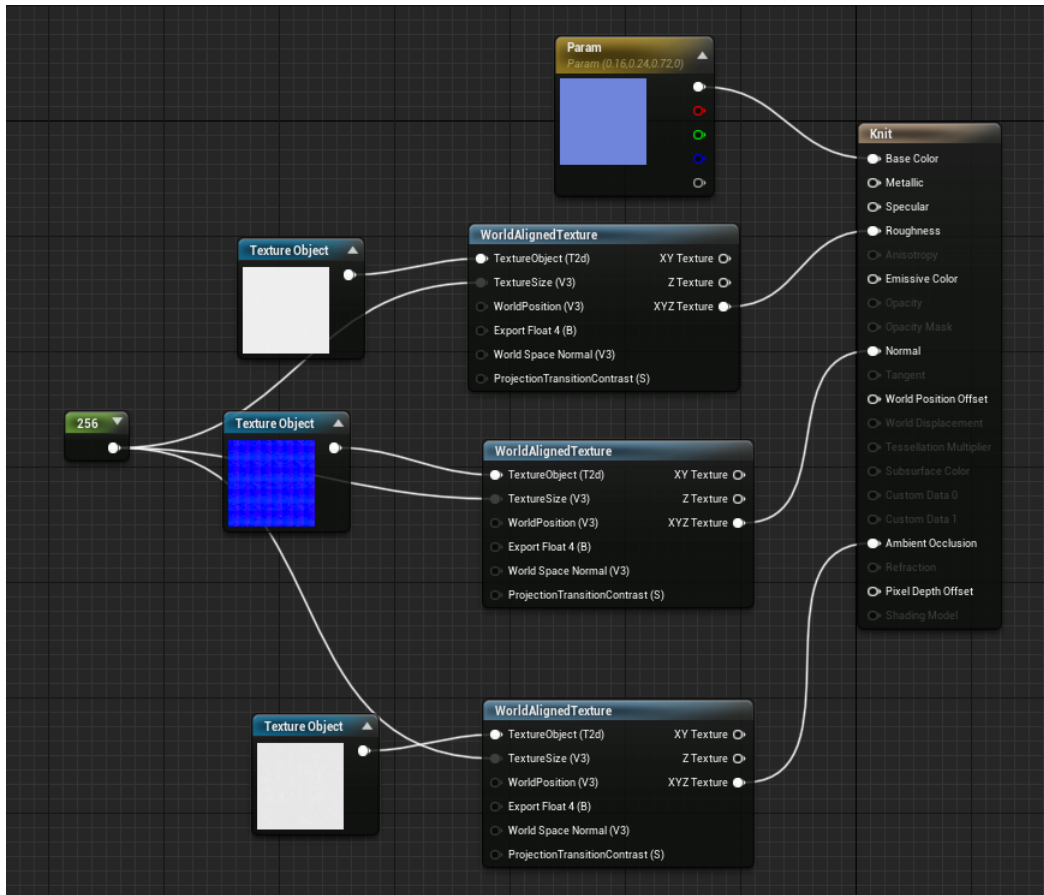


Figure 3.7: Material Instance Script

to create occlusion between the objects in the captured images. While these objects are falling, they can also collide with each other or with other elements in the environment which increases the randomness in the images generated. Meanwhile, the capturer is sent to random positions and orientations within a set of range while the scene is being captured.

In order to implement the collisions correctly, collision meshes should be added to the objects involved. These objects are not only the ones that are falling but also other assets that are involved in the collision such as the tables or the floor of the lab. There are two types of collisions which are called *Simple Collision* and *Complex Collision* which can be added to the objects. The one that is used for the meshes in the environment is the *Simple Collision* which can be added following [42]. It is visualized through the *Collision* dropdown menu as it can be seen in Figure 3.15. More details regarding collision meshes in Unreal Engine can be found in [42].

As a means to render the objects at random locations and let them fall under gravity, Blueprint Visual Scripting [43], which is the visual scripting system inside Unreal Engine 4 is used. It can be reached from the menu shown in Figure 3.16, in Unreal Engine 4.

The interface built for this purpose can be seen in Figures 3.17 & 3.18.

The following nodes are used in this blueprint structure with the described purposes:



Figure 3.8: Robotics Lab of the Institute of Industrial and Control Engineering (IOC-UPC) Environment Setup in Unreal Engine

- Random Float in Range - Generates a random number between Min and Max
- Make Vector - Makes a vector from $\{X, Y, Z\}$
- Random Rotator - Generates a random rotation, with optional random roll
- SpawnEvent - Custom Events provide a way for creating events that can be called at any point in the Blueprint's sequence [44]
- Event BeginPlay - Event when play begins for the actor
- Get Actor of Class - Find the first Actor in the world of the specified class
- Set Timer by Event - Set a timer to execute delegate
- Set Actor Location and Rotation - Move the actor instantly to the specified location and rotation

Example images in Figure 3.19 (512 x 512) are obtained with the described environmental setup. 50K images are generated using the same structure. For photo-realistic training dataset, only ground truth RGB images and their annotation files are generated within this setup.

The accuracy of the generated training data can be checked through an interface called NVIDIA Dataset Utilities (NVDU) [45] which is a Python based project that enables visualizing the annotations delivered by NDDS plug-in through the command `nvdv_viz`. An example image from the training dataset and its visualized 3D bounding box and axes can be seen in Figure 3.20.

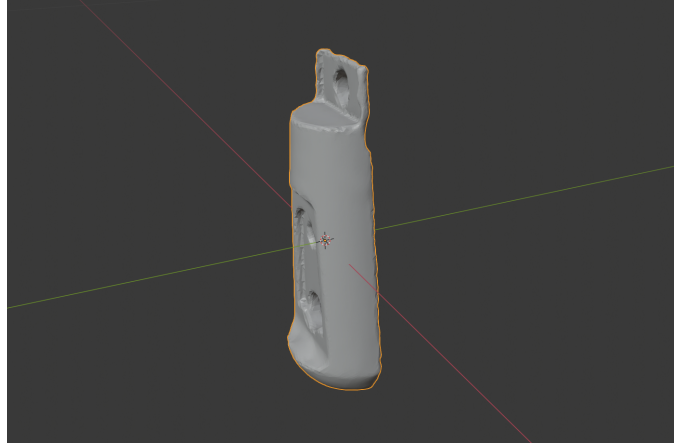


Figure 3.9: UnderBody Model Visualized in Blender

3.1.3.5 Domain Randomized Image Generation

Unlike the photo-realistic case, no environment setup is used in this case. The setup includes the scene capturer, the object of interest, *UnderBody*, the distractors in the scene to create occlusions. As a background, images from COCO dataset [46] are used as suggested in the original paper. 1923 images are selected randomly from the COCO dataset and placed as the background with a 100 Hz rate of change.

All the placed objects, including the object of interest, which is *UnderBody*, and distractors in different shapes, sizes and textures are located inside a 3D volume in front of the background. In this volume, they are spawned at random locations and orientations every 0.01 second. The texture of the distractors is also changed every 0.01 second given a manually created texture subset. Other parameters that are being randomized are the lighting intensity and HSV values within a range. The lighting randomization setting can be seen in Figure 3.21 and the created setup is as in Figure 3.22. In the same Figure, the image on the right bottom corner is what the scene capturer sees from the current location.

Scene capturer is static in this scenario unlike the case in photo-realistic image generation environment so the camera is not sent to random locations and orientations during the image capturing. The imported objects are randomly spawned inside the created cube which is a *TriggerVolume* asset [47], while the capturer is taking the images.

Again, 50K images (512 x 512) with annotation files are generated within this setup and examples of the data created can be seen in Figure 3.23. For domain randomized training dataset, only ground truth RGB images and their annotation files are generated within this setup.

3.1.4 Training

Training process in Deep Object Pose Estimation is based on detecting the keypoints in an RGB image and retrieving pose information from these keypoints. For this purpose, input to the selected deep neural network, which is mentioned in *Network Architecture* section, is an RGB image and the output has two main parts. One of which is the *Belief Maps* and the other output is the *Vector Fields*. There are nine Belief Maps at the output, eight for the vertices and one for the centroid of the bounding box of the object of interest. Likely, there are eight Vector Fields each

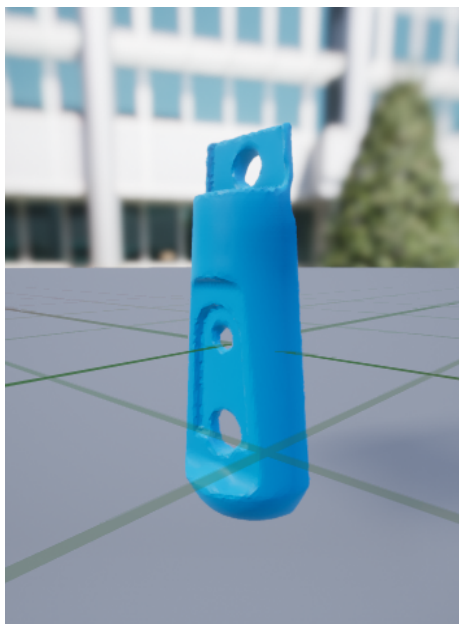


Figure 3.10: UnderBody Model with Color Included

```

>> "exported_object_classes": [
>>   "Under"
>> ],
>> "exported_objects": [
>>   {
>>     "class": "Under",
>>     "segmentation_class_id": 255,
>>     "segmentation_instance_id": 0,
>>     "fixed_model_transform": [
>>       [ 0.99989998340606689, -0.0078999996185302734, 0.015399999916553497, 0 ],
>>       [ -0.014899999834597111, 0.0586999999004602432, 0.99819999933242798, 0 ],
>>       [ -0.0087999999523162842, -0.99819999933242798, 0.058600001037120819, 0 ],
>>       [ 309.88970947265625, -487.64358520507812, -779.31500244140625, 1 ]
>>     ],
>>     "cuboid_dimensions": [ 4.3248000144958496, 18.690799713134766, 7.7175998687744141 ]
>>   }
>> ]

```

1

Figure 3.11: The *object.json* File

of which corresponds the direction from the vertex towards the centroid of the same object.

Belief Maps can be thought as a 2D heat-map of where a keypoint is located, high value is equivalent to high likely-hood of finding a point there. The maximum locations on the heat-map are where the keypoints are located. In addition to belief maps, DOPE can deal with multiple instances of a single object, therefore, different objects should be dissociated and thus each keypoint map is also associated to an Affinity Field, which can be considered as a vector field. Each pixel outputs a vector direction in x and in y . That vector is normalized and it is pointing towards a direction which is then benefited to detect multiple instances of the same object class. PnP algorithm [48] is used to estimate the 6 DoF pose of the objects from the belief maps and vector fields. More details regarding the creation of both belief maps and vector fields can be found in the original paper in *Section 3.2*

Training process which is implemented in the original Deep Object Pose Estimation project, the deep neural network architecture suggested in the original paper, hardware setup used for the

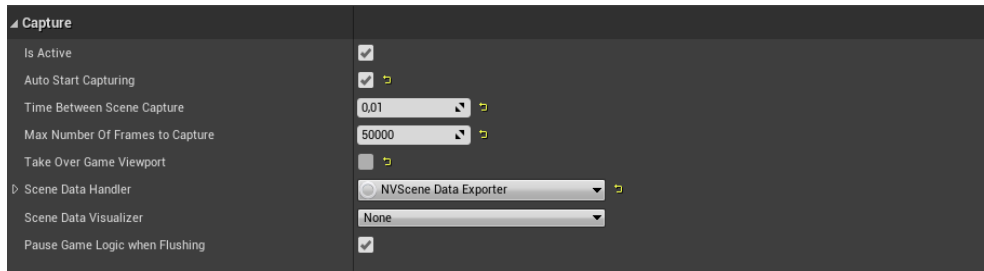


Figure 3.12: Scene Capturer Settings

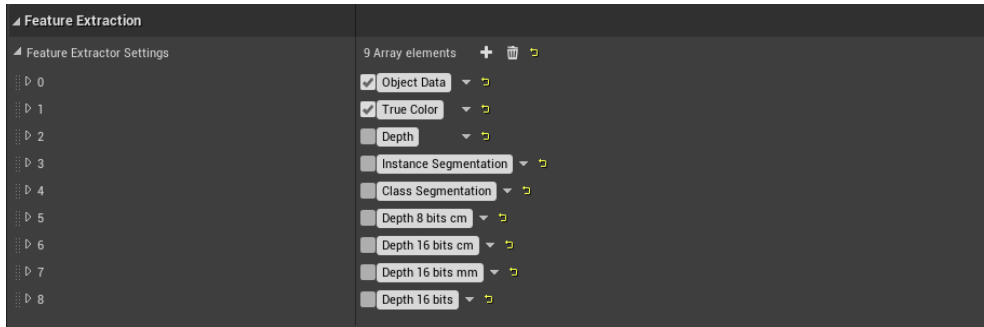


Figure 3.13: Feature Extraction Options Selected for the Scene Capturer

training in the thesis, the training algorithm which is adapted to the used hardware specifications, and the ROS interface provided in the original project are described in the following sections.

3.1.4.1 Original Training

Original source code for training is implemented using PyTorch [49] and the selected optimization algorithm for network weights update is the Adam Optimizer [50].

The following algorithm hyper-parameters are suggested in the original training process. The complete list of the used parameters can be seen in the original source code [51].

```
1 $ python train.py --data /path_to_training_dataset --datatest /
  path_to_test_dataset --batchsize 128 --epochs 60 --outf UnderBody --gpuids 0
  1 2 3 4 5 6 7
```

One important point is that in the original case, 8 GPUs are being used for the training. However, the hardware setup in this thesis has only a single GPU which has the specifications described in Section 3.1.4.3. Therefore, the original training code is adapted to the available hardware setup. The new training implementation is explained in Section 3.1.4.4.

3.1.4.2 Network Architecture

A fully convolutional deep neural network is being used in the original work, therefore, the same architecture is adopted in this thesis. Further details regarding Network Architecture can be accessed through the original paper [2], Section 2.1.

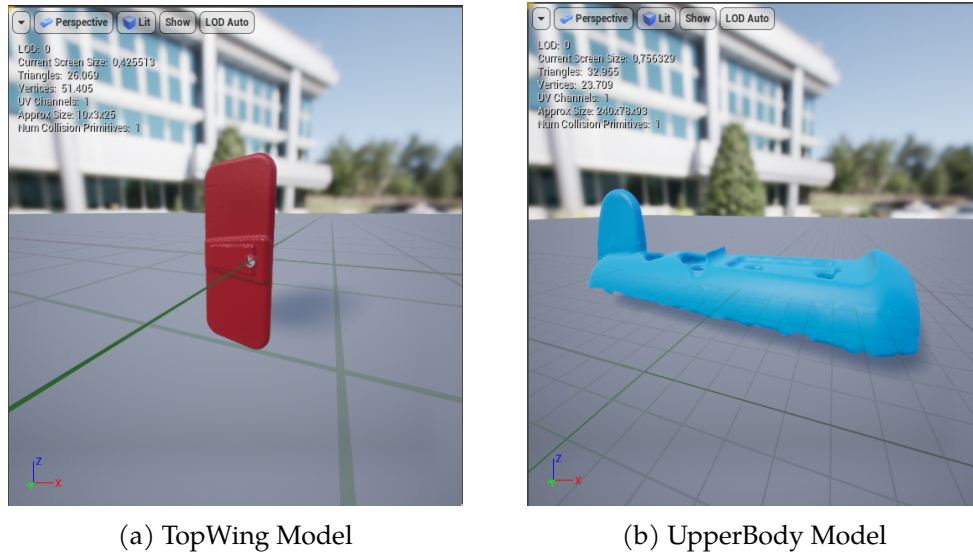


Figure 3.14: Toy Plane Model Parts used as Distractors

3.1.4.3 Hardware Setup

Hardware setup used in this thesis for training has the following specifications:

- Processor - AMD Ryzen 7 3700X 8-Core Processor
- GPU - NVIDIA Corporation TU106 (Geforce RTX 2060 Rev. A [52])
- RAM - 16410093 KB
- OS - Linux AMD64 Buster

3.1.4.4 Revised Training

Considering the hardware setup available, the maximum batchsize that can be selected is 10. However, 8 is suggested to be chosen in this case, since it is a number of power of 2. The reason behind this is the way computer memory works in GPU. Since mini-batches are going to be vectorized and parallelly processed in GPU, choosing a non binary (not power of 2) mini-batch size may result in inefficient hence poorer performance. When large volume of data is being dealt with, small inefficiencies can have a large impact on performance.

Even though training can be implemented using a really small batch size, using a smaller batch-size than the suggested one highly affects the training process and the performance of the trained network due to reasons discussed below:

- In general, choosing a really small batch size makes the training process too slow due to the significantly lower computational speed, because of not exploiting vectorization to the full extent.
- The smaller the batch size is the more noisy the gradients get. Therefore, the oscillations around a minimum are greater since the estimate of the gradient will be less accurate.

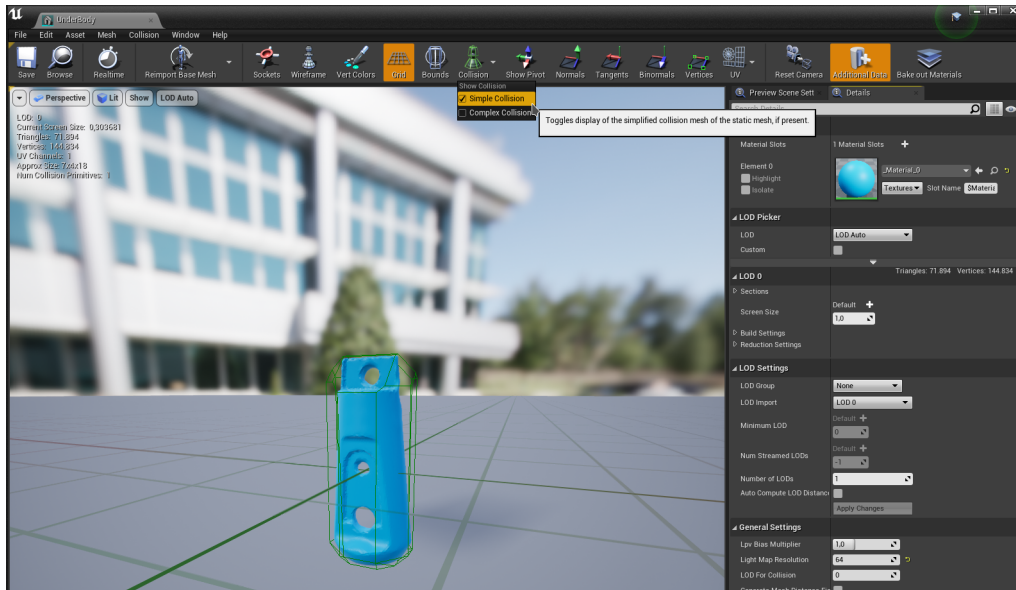


Figure 3.15: Simple Collision Visualization for UnderBody Model

Even though, greater oscillations can help getting out of a bad local minimum, choosing a really small batch size can also make the convergence to a minimum harder.

Due to memory insufficiency of NVIDIA Corporation TU106 (Geforce RTX 2060 Rev. A) to achieve the suggested batch size for training, which is 128, a training code adapted to a single GPU system is used. By using the *Sub-batching* concept, the proposed batch size for training is reached. The idea behind this concept is, the training data loader is called as if the batches are in size of the given *subbatchsize* instead of the *batchsize*. The new data loader for training data is created as below and the revised training source code can be found in [53].

```

1  trainingdata = torch.utils.data.DataLoader(train_dataset ,
2      batch_size = opt.subbatchsize ,
3      shuffle = True ,
4      num_workers = opt.workers ,
5      pin_memory = True
6  )

```

The training algorithm for the network can be seen in the Appendix Listing 2.

In addition to the sub-batch concept added into the original source code, another update is running the forward pass with auto casting. The reason why *amp.autocast()*, which is from the Automatic Mixed Package, is chosen for forward passing is that the instances of autocast serve as context managers or decorators that allow parts of the script to run in mixed precision. In these parts, CUDA ops run in an op-specific dtype chosen by autocast. This is due to the reason where some operations use the torch.float32 (float) datatype and other operations use torch.float16 (half). Some ops, like linear layers and convolutions, are much faster in float16. Other ops, like reductions, often require the dynamic range of float32. Therefore, Automatic Mixed Precision package is used to improve the performance while maintaining accuracy.

Following command is called for the revised training:

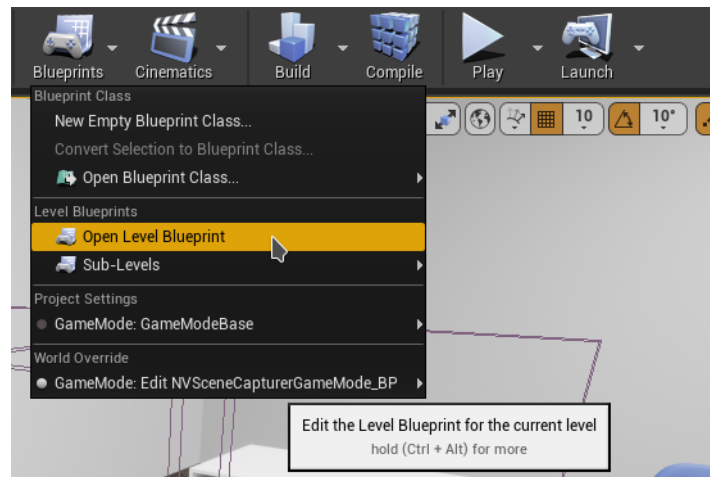


Figure 3.16: *Open Level Blueprint* Menu Access

```
1 $ python train.py --data /path_to_training_dataset --batchsize 128 --
  subbatchsize 8 --epochs 60 --outf UnderBody
```

Rest of the input arguments are kept at their default values which can be found in the source code [53]. Results of the training process are discussed in Section 5.1.

3.1.5 ROS Interface

A ROS Interface is provided in Deep Object Pose Estimation project to publish the information regarding the detected objects in the environment through ROS. This interface consists of two main ROS nodes which are:

1. **Camera Node:** This node publishes the RGB images obtained through the camera within the system. One important requirement is that the camera should publish the correct camera info topic so that the trained network can get the poses of the objects correctly. The content of the camera info includes:
 - Camera Matrix - Also known as the Intrinsic Matrix which allows to transform 3D coordinates to 2D coordinates on an image plane using the pinhole camera model.
 - Distortion Model - The model which describes the mathematical deviation of a camera from the pinhole model.
 - Distortion Coefficients - The coefficients used to represent Tangential & Radial distortion [54].
 - Rectification Matrix - A rotation matrix aligning the camera coordinate system to the ideal stereo image plane so that epipolar lines in both stereo images are parallel.
 - Projection Matrix - The matrix that describes the transformation between an image point and a ray in Euclidean 3-space [55].

The camera being used in Robotics Lab of the Institute of Industrial and Control Engineering (IOC-UPC) is a Kinect XBOX Camera with the camera info aspects in Figure 3.24.

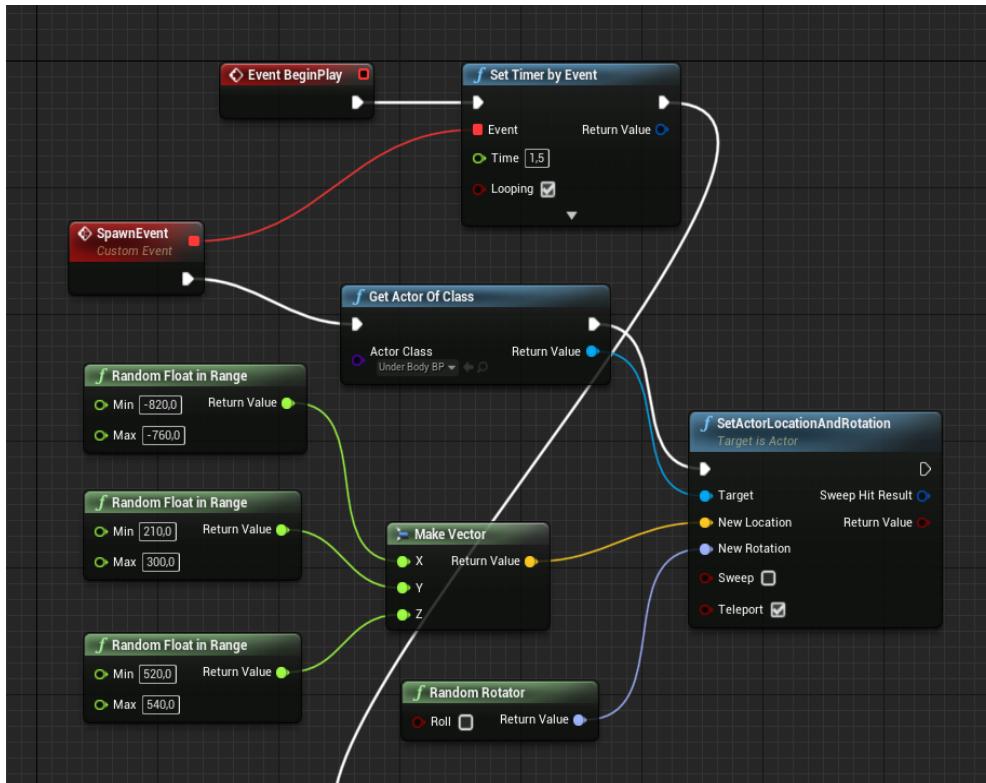


Figure 3.17: Blueprint Structure to Spawn the UnderBody Model in the Environment

This Kinect camera is connected to the Geminis computer in lab. Therefore, Geminis should be set as the ROS_MASTER in order to reach the published images from another computer in the lab. In order to create the bridge between two computers in the lab, the following commands are required to be called to start retrieving the rectified images from Geminis:

- (a) `roslaunch kinect2_bridge kinect2_bridge_hq.launch` (in Geminis)
- (b) `export ROS_MASTER_URI:https://geminis:11311` (in local)

From the topics Kinect XBOX camera publishes, the `/kinect2/hd/image_color_rect` is used.

2. **DOPE Node:** The node that reads the images coming from the Camera node and publishes information regarding the detected object. To do so, it reads the content of `config_pose.yaml`, an example of this file can be seen in Figure 3.25, provides the following concepts:

- Weights - The path for the trained network weights for every object wished to be detected.
- Dimensions - Object cuboid dimensions in cm.
- Class IDs - Different class ID assignment for every object.
- Draw Colors - Color codes for the cuboid to be drawn around the detected objects in

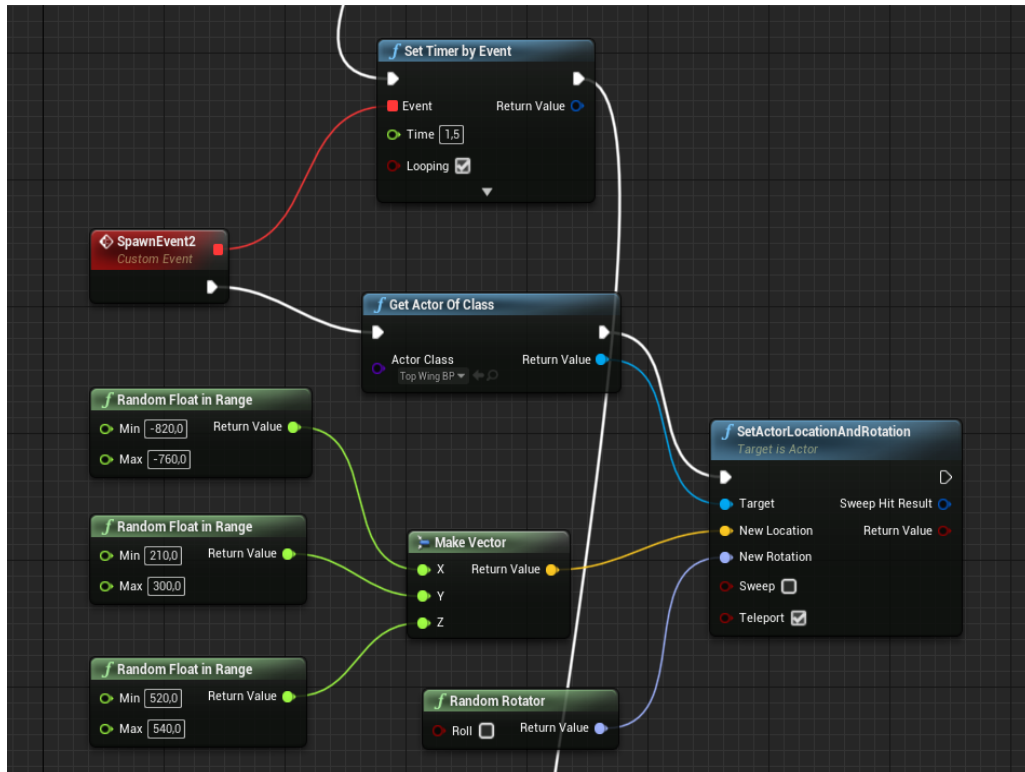


Figure 3.18: Blueprint Structure to Spawn the TopWing Model in the Environment

RGB images.

- Model transforms - An optional transform that can be applied to the cuboids returned by DOPE node.
- Meshes - Optional object mesh path for RViZ visualization purposes.
- Mesh Scales - An optional scale that will be applied to the dimension values in case the values provided are in meters.
- Threshold Map - A threshold map applied to the belief maps to retrieve the binary peaks for vertex detection.
- Sigma - The standard deviation of Gaussian filter applied to the belief maps.
- Threshold Points - Confidence threshold for objects corner detection.

Then, the following topics are being published:

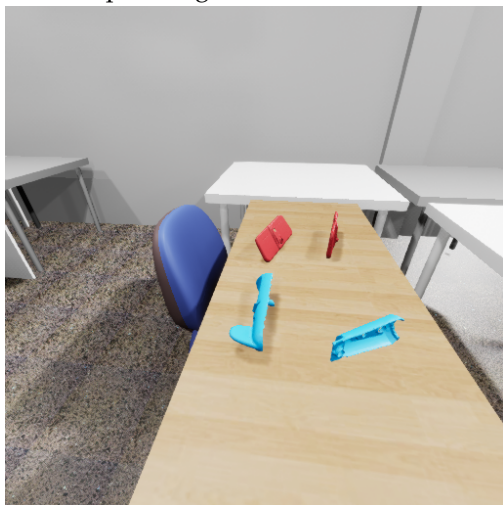
- `/dope/webcam_rgb_raw` - RGB images from camera
- `/dope/dimension_[obj_name]` - Dimensions of object
- `/dope/pose_[obj_name]` - Timestamped pose of object
- `/dope/rgb_points` - RGB images with detected cuboids overlaid



(a) An Example Image on the Chess Table Location



(b) An Example Image on the Floor Location

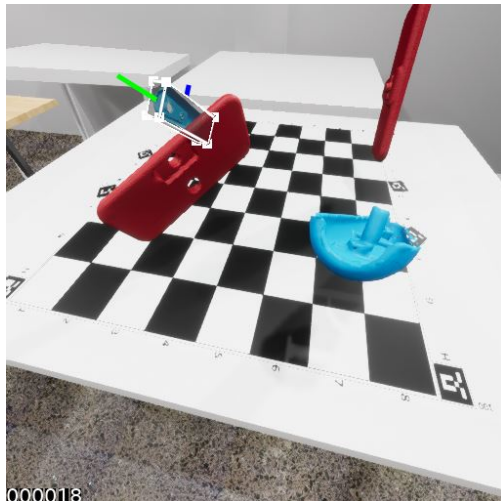


(c) An Example Image on the Wooden Table Location

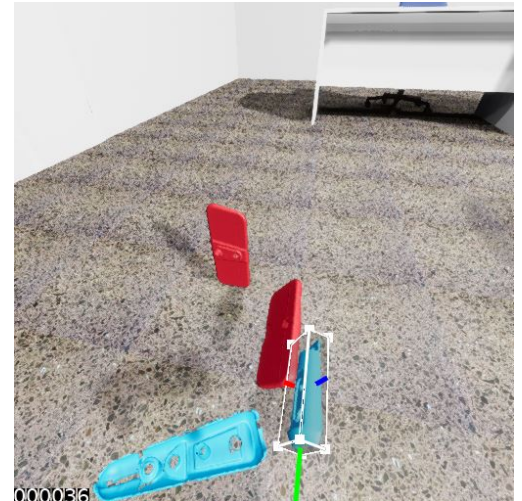


(d) An Example Image on the White Table Location

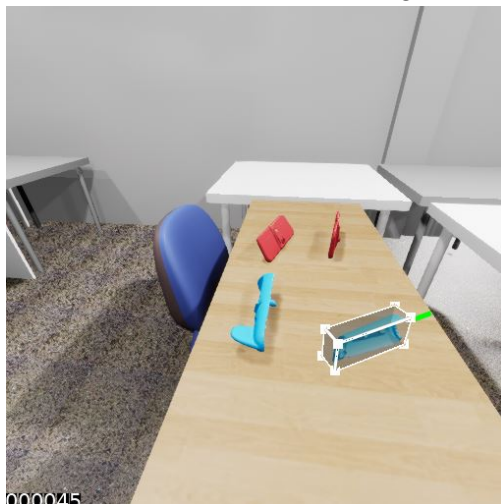
Figure 3.19: Example Photo-realistic Images



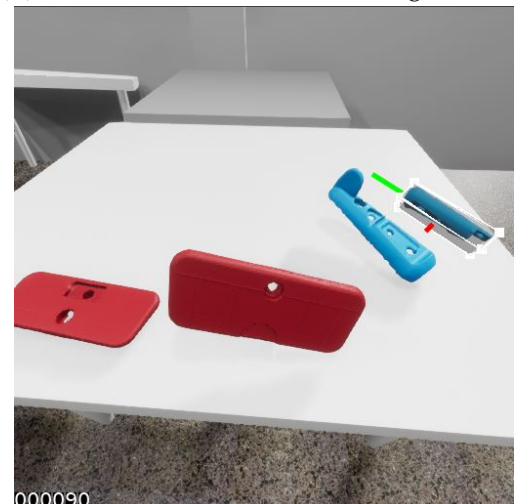
(a) Annotation Visualization for Figure 3.19a



(b) Annotation Visualization for Figure 3.19b



(c) Annotation Visualization for Figure 3.19c



(d) Annotation Visualization for Figure 3.19d

Figure 3.20: Example *node_viz* for Photo-realistic Images

- `/dope/detected_objects` - `vision_msgs/Detection3DArray` of all detected objects
- `/dope/markers` - RViz visualization markers for all objects

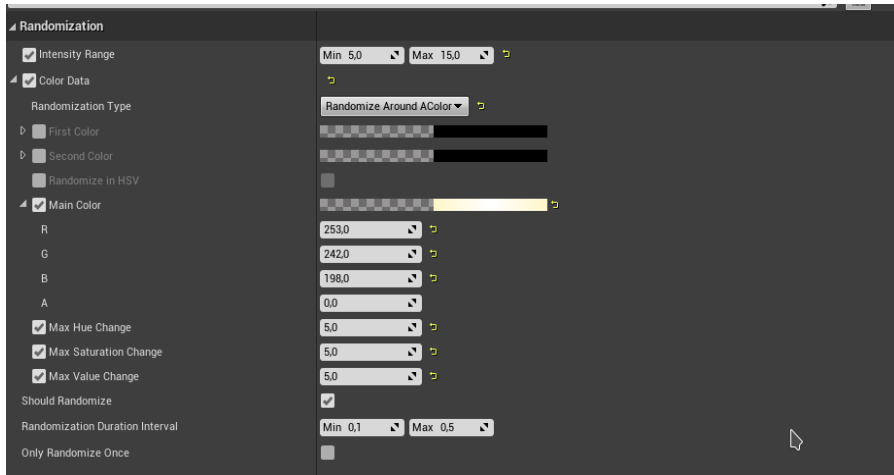


Figure 3.21: Lighting Randomization Parameters

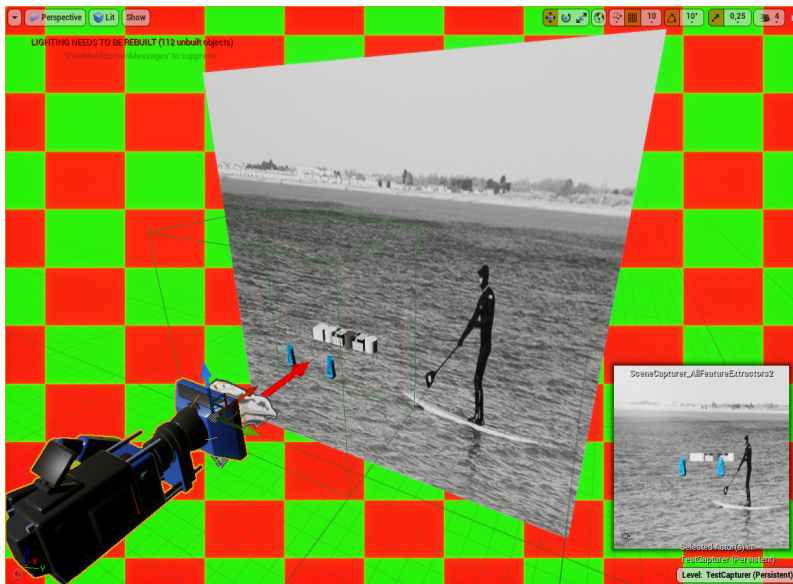
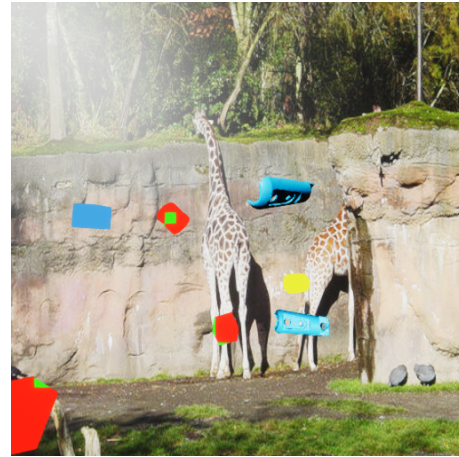


Figure 3.22: Created Setup to Obtain Domain Randomized Images



(a) Example Image 1



(b) Example Image 2

Figure 3.23: Example Images Obtained with Domain Randomization

```
fatma.nur@geminis:~$ rostopic echo -n 1 /kinect2/hd/camera_info
header:
  seq: 0
  stamp:
    secs: 1628162221
    nsecs: 954548046
  frame_id: "kinect2_rgb_optical_frame"
height: 1080
width: 1920
distortion_model: "plumb_bob"
D: [0.056863275373698105, -0.08184940257699182, -0.00120525069339969, 0.0005690407529780594, 0.030489123664987936]
K: [1056.9611222345209, 0.0, 960.5749657806307, 0.0, 1056.718100522538, 534.0560224877325, 0.0, 0.0, 1.0]
R: [1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0]
P: [1056.9611222345209, 0.0, 960.5749657806307, 0.0, 0.0, 1056.718100522538, 534.0560224877325, 0.0, 0.0, 0.0, 1.0, 0.0]
binning_x: 0
binning_y: 0
roi:
  x_offset: 0
  y_offset: 0
  height: 0
  width: 0
do_rectify: False
--
```

Figure 3.24: *camera_info* Topic for Kinect XBOX Camera

```

topic_camera: "/kinect2/hd/image_color_rect"
topic_camera_info: "/kinect2/hd/camera_info"
topic_publishing: "dope"
input_is_rectified: True # Whether the input image is rectified (strongly suggested!)
downscale_height: 400 # if the input image is larger than this, scale it down to this pixel height

▼ weights: {
  "Under": "file:///srv/robotica/catkin_ws/src/dope/scripts/train_UnderBody/net_final_60.pth",
}

# Cuboid dimension in cm x,y,z
▼ dimensions: {
  "Under": [ 4.3248000144958496, 18.690799713134766, 7.7175998687744141 ],
}

▼ class_ids: {
  "Under": 1
}

▼ draw_colors: {
  "Under": [255, 101, 0],
}

# optional: provide a transform that is applied to the pose returned by DOPE
▼ model_transforms: {
# "cracker": [[ 0, 0, 1, 0],
#             [ 0, -1, 0, 0],
#             [ 1, 0, 0, 0],
#             [ 0, 0, 0, 1]]
}

# optional: if you provide a mesh of the object here, a mesh marker will be
▼ meshes: {
}

# optional: If the specified meshes are not in meters, provide a scale here (e.g. if the mesh is in centimeters, scale should be 0.01). default scale: 1.0.
▼ mesh_scales: {
}

# Config params for DOPE
thresh_angle: 0.5
thresh_map: 0.01
sigma: 2
thresh_points: 0.1

▼ camera_settings: {
  "fx": 1056.9611222345209,
  "fy": 1056.718100522538,
  "cx": 960.5749657806307,
  "cy": 534.0560224877325,
  "dist_coeffs": [0.056863275373698105, -0.08184940257699182, -0.00120525069339969, 0.0005690407529780594, 0.030489123664987936]
}

```

Figure 3.25: The *config_pose.yaml* File

4 Reasoning Module

In order to model the task with appropriate PDDL files in an autonomous manner, which are specific to the given task, following challenges should be addressed:

1. Detecting the objects and agents within the environment using a proper perception module,
2. Interpreting the perceived environment through a reasoning framework to understand the current scene, and
3. Generating the task specific PDDL files, given a global domain PDDL file including all the possible actions that can be executed within the environment, based on the current state of the environment.

This paper suggests an overall schema shown in Figure 4.2 to implement a structure that resolves the above mentioned challenges.

4.1 Perception

In order to provide the necessary information to the reasoning process to generate PDDL files for a given task, perception module should transfer the observed environment where the task will take place. Various methodologies can be followed for this purpose, in this Reasoning Module section, detection is achieved through fiducial markers that are placed on the objects to be detected since Deep Object Pose Estimation implementation studied in Section 3 is not providing accurate detection results at this stage. The raw information regarding the detected aspects within the perceived environment is shared with the main perception module which organizes the structure of the data and then transfers this metadata in an XML, as in Listing 1, format to the reasoning algorithm. This metadata includes the following information for every object that has been detected:

- The name of the object
- The pose
- The fiducial marker ID
- The camera which detects the object

```

1 <?xml version="1.0"?>
2 <Object >
3   <Index>1</Index>
4   <ArucoID>101</ArucoID>
5   <FrameID>Kinect_Camera</FrameID>
6   <ObjectName>OBJECTA</ObjectName>
7   <NodeName>Kinect</NodeName>
8   <Pose>x=1.032 y=0.123 z=0.245 wx=0.9 wy=0.2 wz=0.3 w=0.6</Pose>
9 </Object>
10 <Object>
11   <Index>2</Index>
12   <ArucoID>102</ArucoID>
13   <FrameID>Kinect_Camera</FrameID>
14   <ObjectName>OBJECTB</ObjectName>
15   <NodeName>Kinect</NodeName>

```

```

16 <Pose>x=1.032 y=0.123 z=0.245 wx=0.9 wy=0.2 wz=0.3 w=0.6</Pose>
17 </Object>
18 <Object>
19 <Index>3</Index>
20 <ArucoID>103</ArucoID>
21 <FrameID>Kinect_Camera</FrameID>
22 <ObjectName>OBJECTC</ObjectName>
23 <NodeName>Kinect</NodeName>
24 <Pose>x=1.032 y=0.123 z=0.245 wx=0.9 wy=0.2 wz=0.3 w=0.6</Pose>
25 </Object>

```

Listing 1: Example obj_list.xml File

4.2 Reasoning Framework

4.2.1 Knowledge Layer

Prior to implementing the reasoning process which is explained in Section 4.2.2, the knowledge layer should be generated. For this purpose, Perception and Manipulation Knowledge (PMK) ontology is used as a template. More information regarding PMK knowledge structure can be found in Section *PMK Knowledge Structure* in [25].

Depending on the scenario, new instances and relations should be added to the existing template. In order to manipulate the knowledge layer which consists of the ontology files, an ontology editor interface Protege [7] and ontology web language (OWL) [56] are used in this thesis. New classes, individuals, object properties and data properties can be implemented within this editor in order to represent the scenario in which the robotics task will take place.

After generating the ontologies in OWL format, Prolog [8] and created predicates are used in this thesis to reason over the OWL file. Prolog is a logic programming language widely used in semantic web applications and Prolog engine's main job can be defined as reasoning if a statement is true or false. When a statement is true (in case of success) it outputs the instantiations which make it true. This happens through unification [57]. In this framework, Prolog is being used to retrieve the outputs of the defined predicates which can be considered as the questions that are used to query over the ontologies in the knowledge layer. In order to reason the predicates, the information coming from the perceived environment is used as inputs to the predicates.

For this purpose, a Prolog file is generated, which is connected to the ontology by loading the OWL file into the Prolog environment through the following code piece.

```

1 :- rdf_load('/home/fato/pmk_python_interface/pyswip/ontologies/twoRoom.owl').
2 :- rdf_db:rdf_register_ns(sir_pmk, 'http://www.semanticweb.org/fato/ontologies
   /2021/1/untitled-ontology-5#', [keep(true)]).

```

The first line loads the ontology and the second line registers the given prefix, which is *sir_pmk* in this case, to ontologies Uniform Resource Locator (URI), which is specific to the generated ontology and used to identify the ontology and the elements within the ontology file. Therefore, the instances of the ontology can be called through this generated prefix. An example to this can be seen in Figure 4.1 where *sir_pmk:'Can_1'* is an Artifact individual created in an example OWL file.

This created Prolog file also includes the initialization of all the predicates, specifying the num-

ber of required inputs. Each of the initialized predicate is defined later in the same Prolog file. The initialization step can be seen as below where also the number of required inputs are specified.

```

1 :- rdf_meta
2   find_subclass(r,r),
3   find_robot_cap(r,r),
4   find_spatial_rel(r,r,r),
5   find_obj_prop(r,r,r,r,r,r,r),
6   find_obj_cons(r,r),
7   find_grasp_pose(r,r),
8   find_robot(r,r).

```

An example predicate definition for *find_obj_prop(Artifact, Diameter, Length, Height, Width, Color, Type)* is as seen below, which returns the properties (i.e. Diameter, Length, Height, Width, Color, and Type) of the provided Artifact.

```

1 find_obj_prop(Artifact, Diameter, Length, Height, Width, Color, Type):-
2 (rdf_has(Artifact, sir_pmk:'diameter', D) -> literal_type_conv(D, Diameter);
3   true),
4 (rdf_has(Artifact, sir_pmk:'height', H) -> literal_type_conv(H, Height); true),
5 (rdf_has(Artifact, sir_pmk:'length', L) -> literal_type_conv(L, Length); true),
6 (rdf_has(Artifact, sir_pmk:'width', W) -> literal_type_conv(W, Width); true),
7 (rdf_has(Artifact, sir_pmk:'color', C) -> literal_type_conv(C, Color); true),
8 rdf_has(Artifact, rdf:type, T),
9 literal_type_conv(T, Type).

```

In order to compile the created Prolog files, SWI-Prolog [8] is being used, which is a compiler for the Prolog language.

SWI-Prolog can be built through the package manager (*sudo apt-get install swi-prolog*), however, in this thesis, it is built from its source code following the steps below which are also provided in [58]:

1. Install SWI-Prolog from the source.

```

1 git clone https://github.com/SWI-Prolog/swipl-devel.git
2 cd swipl-devel
3 git submodule update --init

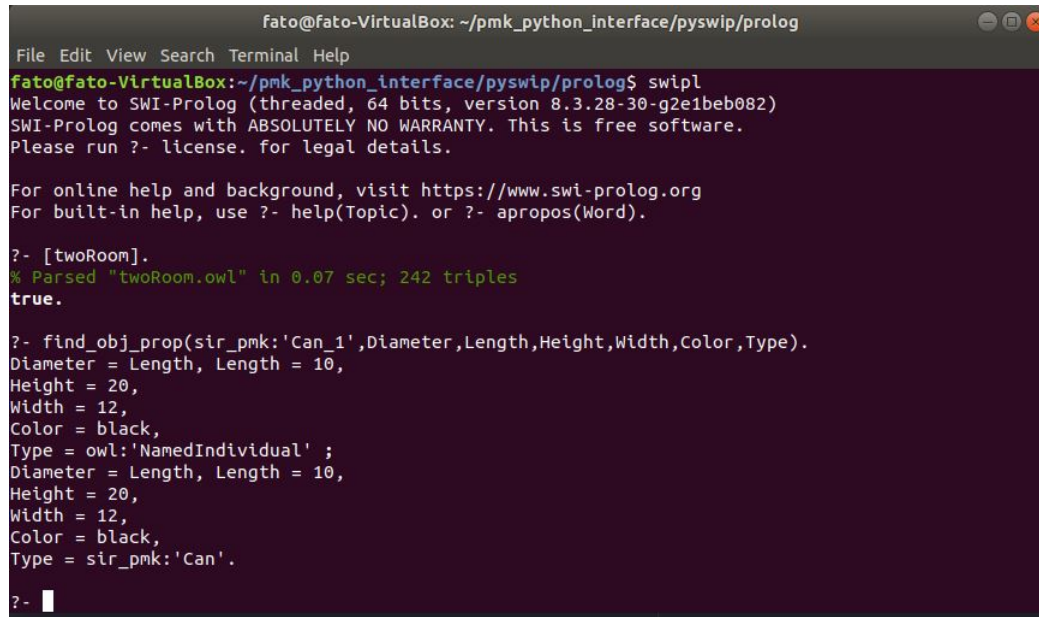
```

2. Get the prerequisites for SWIPL (The following code piece is for Debian based systems, for other systems check [58], Section *Getting the Prerequisites*):

```

1 sudo apt-get install \
2   build-essential cmake pkg-config \
3   ncurses-dev libreadline-dev libedit-dev \
4   libgoogle-perftools-dev \
5   libunwind-dev \
6   libgmp-dev \
7   libssl-dev \
8   unixodbc-dev \
9   zlib1g-dev libarchive-dev \
10  libssp-uid-dev \
11  libxext-dev libice-dev libjpeg-dev libxinerama-dev libxft-dev \
12  libxpm-dev libxt-dev \
13  libdb-dev \

```



```
fato@fato-VirtualBox: ~/pmk_python_interface/pyswip/prolog
File Edit View Search Terminal Help
fato@fato-VirtualBox:~/pmk_python_interface/pyswip/prolog$ swipl
Welcome to SWI-Prolog (threaded, 64 bits, version 8.3.28-30-g2e1beb082)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?- [twoRoom].
% Parsed "twoRoom.owl" in 0.07 sec; 242 triples
true.

?- find_obj_prop(sir_pmk:'Can_1',Diameter,Length,Height,Width,Color,Type).
Diameter = Length, Length = 10,
Height = 20,
Width = 12,
Color = black,
Type = owl:'NamedIndividual' ;
Diameter = Length, Length = 10,
Height = 20,
Width = 12,
Color = black,
Type = sir_pmk:'Can'.
?-
```

Figure 4.1: SWI-Prolog Interface

```
14 libpcre3-dev \
15 libyaml-dev \
16 default-jdk junit4
```

3. After installing the dependencies build SWI-Prolog.

```
1 cd swipl-devel
2 mkdir build
3 cd build
4 cmake -DCMAKE_INSTALL_PREFIX=/usr/local ..
5 make
6 sudo make install
```

By using SWI-Prolog, the predicates defined in the Prolog file can be reasoned through *swipl* interface in a terminal in the same directory where the Prolog file is located. The created Prolog file and the linked ontology should be parsed in the interface before reasoning the predicates. It is done by calling `[twoRoom].`, which is an example Prolog file, command in the same terminal where *swipl* is called initially. An example return of `find_obj_prop(Artifact, Diameter, Length, Height, Width, Color, Type)` predicate, within `twoRoom.pl` Prolog file, for the object `Can_1` can be seen in Figure 4.1. There are two returns to this predicate since `sir_pmk:'Can_1'` has two possible *Types*. It is either an `owl:'NamedIndividual'` which is by default for OWL individuals created and also a `sir_pmk:'Can'`, which is the Class `sir_pmk:'Can_1'` belongs to.

With the help of *swipl*, the knowledge layer which consists of OWL & Prolog files, can be accessed.

4.2.2 Ontology-based reasoning for robot manipulation

Robotic manipulation involves the planning at task level (determining which is the sequence of actions to be done to perform a given task) and at motion level (finding the sequence of collision-free motions that allow to safely move the robot from one configuration to another).

In this scope, a standardized ontological-based reasoning framework called Perception and Manipulation Knowledge (PMK) was introduced in [25] as a tool to help task and motion planning systems (TAMP) in terms of reasoning, by providing:

- Reasoning for perception related to sensors and algorithms, e.g. to determine which is the sensor to be used in a given situation.
- Reasoning about the objects features, e.g. to determine if an object is pickable or not.
- Reasoning for situation analysis to spatially evaluate the objects relations between each other, e.g. to determine if an object is behind another.
- Reasoning for planning to reason about the preconditions of actions, action constraints and geometric reasoning related to the robot and to the environment, e.g. to determine if a grasping pose is reachable or to select a feasible placement region.

PMK is enhanced here by including knowledge about the actions the available robots can perform into the ontologies, and by extending the object features related to those actions. Also, reasoning predicates are provided to help in the selection of the actions required to solve a given task and to automatically set the PDDL domain and problem files.

4.2.2.1 Robot-centered reasoning

Robot-centered reasoning predicates include:

1. `find_robot(Region, Robot)` to return the available robots within the environment and the regions they are located at;
2. `find_robot_capability(Robot, Capability)` to return the capabilities of a given robot;
3. `find_robot_reach(Robot, Region)` to check if the given robot can reach the given region using its capabilities.

4.2.2.2 State-centered reasoning

State-centered reasoning is required to reason on the initial and goal state of the world. Even though the perception module provides the poses of the detected objects and agents, it does not provide the symbolic regions where they are located, nor the spatial relation information between the detected objects (i.e. in, on, left, right). For this purpose, spatial evaluation predicates from PMK, which convert raw perceived environment information into spatial locations and relations through specified calculations, are used to reason on the state of the environment based on the perceived objects. These calculations are based on comparing perceived coordinates of the object with specific values depending on the spatial relation definition. For instance, to infer the *on* relation between two objects, the X & Y coordinates should be similar for the two objects. Meanwhile, the Z coordinate should be contiguous within a certain threshold. More details regarding the spatial relation descriptions can be found in [25], Table 1. In addition to finding the spatial relations between the objects, the initial and goal states can be reasoned with the following predicates that are inherited from PMK:

1. `retrieve_symb_init(ObjPose, SymbRgn)` to return the symbolic region where the given ob-

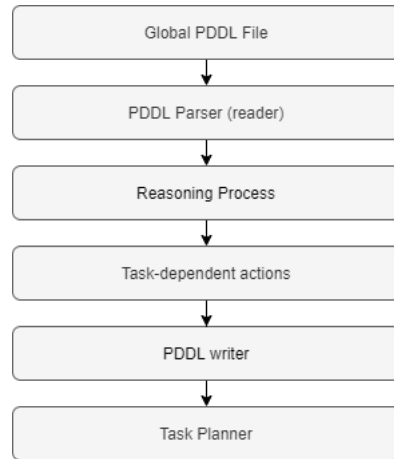


Figure 4.2: Flowchart of the overall symbolic reasoning operation

ject pose is located at;

2. `retrieve_symb_goal(Task, SymbGoal)` to return the symbolic goal region for a given task.

The overall reasoning process through the above mentioned predicates being queried over knowledge is presented in Section 4.4.

In order to reason over the knowledge, an intermediate layer between the knowledge and the user program is required to organize the sequence of the predicates to be queried with the relevant information coming from the perception module. For this purpose, in this thesis, the overall system schema in Figure 4.3 is implemented through a ROS interface.

One necessary step to implement this architecture is to establish a connection between SWI-Prolog and Python which is chosen as the programming language for this implementation. For this purpose, in this thesis, a package called *PySwip* [59] is used. With the help of this package, Prolog files can be queried within Python. In the same Python environment, a ROS interface is also implemented to access the query results through a ROS service which can be called within any ROS application when necessary.

In the original source of *PySwip*, it is suggested to work with the package in a virtual environment and an installation through *pip* [60], however, when the ROS service is implemented within this setup, a *Segmentation Fault (core dumped)* error occurs when the created ROS service is called to query the predicates. Therefore, instead of installing *PySwip* through *pip*, the source code is cloned and added inside the Python project. It is located inside the */tests* directory in the project. (Provided repository can be checked [61].) By doing so, the *Segmentation Fault* is fixed.

After creating the link between SWI-Prolog and Python, a service is created which takes the predicate to be reasoned and its inputs as a *Request* from the Client and returns the reasoned statement as a *Response*. Created ROS service, *Predicate.srv*, has the following structure.

```

1  pyswip_msgs/Question predicate
2  pyswip_msgs/Inputs [] inputs
3  ---
4  pyswip_msgs/Responses [] responses
  
```

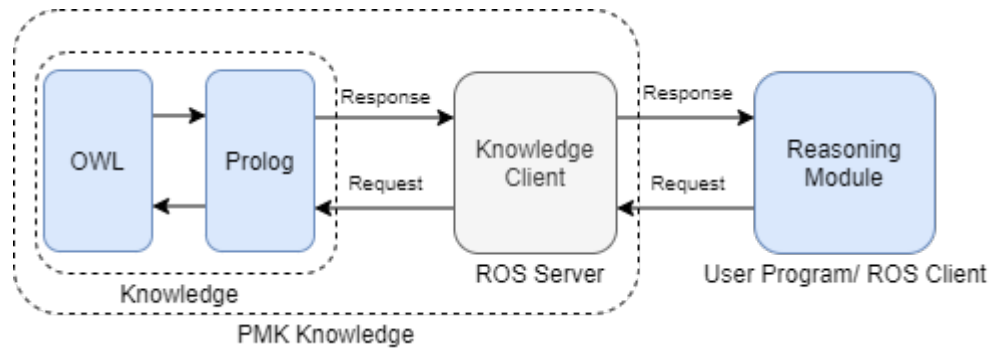


Figure 4.3: Knowledge Management Schema

pyswip_msgs package can be found in the provided source [61].

4.3 PDDL Parser

In order to deal with the last challenge which is the generation process of task specific PDDL files, a PDDL parser and writer tool is required. With the help of this package, the global PDDL file provided should be parsed into its elements (e.g. actions and predicates that are defined to set preconditions and effects of the given actions) and also the new PDDL files should be generated given the set of feasible actions. For this purpose a parser package called *Universal-PDDL-parser* [9] is being used. By using the features of this package, the initial global domain file with all the possible actions is parsed into its elements and the content is stored using the classes within the package. With the help of this hierarchical class structure, the links between the elements in the global file are also preserved.

After reading and storing the global domain file, the task specific PDDL files for the given task can be automatically generated by using the same parser tool to write into a new PDDL file including only the actions that are relevant for the task. The decision of this relevance is made by the reasoner. In order to rewrite the task specific domain file, the feasible actions list determined by the reasoning process is forwarded into `print(std::ostream& os, std::vector<std::string> actionList)` function in the parser package where `os` input is the new task specific PDDL file to be generated and the `actionList` input is the feasible action set coming from the reasoner. The new PDDL file output is saved inside the current directory.

4.4 Automatic Generation of Domain PDDL File

PDDL files are generated autonomously with the help of the aforementioned perception module, reasoning process, and the PDDL parser. Subsequent to perceiving the environment where the task will take place through perception module, elements within the environment are interpreted with the reasoning process.

In order to select an action or another, a set of core reasoning checks has to be done. These checks rely on the robot & state-centered reasoning in which they are combined to answer the query: *What are the actions to be included in the PDDL files to solve this problem?* The answer is the new PDDL files based on the current situation of the environment. In order to reason the specified query, the following core checks are reasoned for a given task and situation:

1. Spatial relations between the objects are retrieved as described in Section 4.2.2.2, such as

Obj1 on Obj2;

2. Initial and goal states are obtained through `retrieve_symb_init()` and `retrieve_symb_goal()` in symbolic form as defined in Section 4.2.2.2;
3. Available robots through `find_robot()` and their capabilities through `find_robot_capability()` are reasoned as defined in Section 4.2.2.1.

Finally, based on the answers of the previous core checks: a) The robot assigned to the task is selected; b) The PDDL file is automatically generated including feasible actions for a given task through the PDDL parser. These core checks can be extended based on the complexity of the task. An example code base of this implementation can be seen in Listing 3 with all the predicates queried in order. Depending on the actions available in the provided global domain PDDL, the actions selection can be implemented.

One example for the reasoning-based action selection process explained above can be an environment in which an object is placed on top of a counter which is out of reach for the given robot. This robot has to grasp this object in order to comply with its task description. In this case, navigation capability is required to bring the object from its current state to a region where it is reachable. After reasoning all the available robots and their capabilities, MOVE action can be added to the robot which has navigation capability and that robot can be assigned to this specific task to bring the object.

Another example can be, an environment where the objects should be placed on top of each other with a given order. If the retrieved spatial relations and the initial states of the objects are not correct in terms of ordering, UNSTACK action with the robot which is available within the environment and can implement this action through its capabilities can be included into the task.

4.5 Automatic Generation of Problem PDDL File

In addition to selection of task specific actions, the reasoning framework provides which available agent in the environment executes which task in order to solve the task problem. Therefore, depending on the robot and task pairing, appropriate predicates and goal states are written into the task specific PDDL problem file. An example case of task specific PDDL problem file is discussed in Section 5.2.

4.6 Test Scenarios

In order to test the explained reasoning framework, two test scenarios are generated and described in this section. The results regarding the scenarios are provided in Section 5.2.

4.6.1 Tiago Kitchen Scenario

In this scenario, Tiago is inside a kitchen environment, as in Figure 4.4, in which it has to transfer objects between counters & prepare beverages (glass of wine, cup of coffee etc.). Since there is only a single agent in the environment, Tiago, task specific domain PDDL is generated only for Tiago. This task specific domain PDDL file should contain the following relevant actions:

- Pick-up

- Put-down (place)
- Move
- Pour

The following actions should be eliminated from the global domain PDDL:

- Open Drawer
- Ask a Human
- Push & Pull

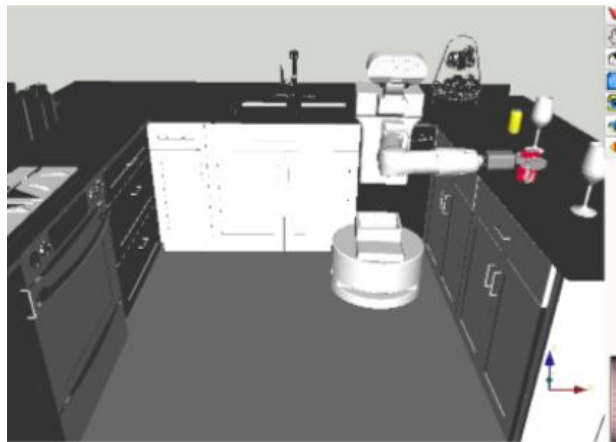


Figure 4.4: Tiago Kitchen Test Environment

In order to decide which actions are kept and which actions are eliminated from the global domain PDDL, the following queries need to be answered for this scenario:

1. Which robot is present in the environment?
2. Is there a human present in the environment?
3. Does the robot have navigation capabilities?
4. Does the robot have grasping capabilities?
5. Are there any objects stored in the drawers?
6. Are the objects within the environment pickable? (considering weight limitations)

The answers to these questions are:

1. Tiago. (Then, the queries regarding robot capabilities are for Tiago in this case.)
2. No. (We will ask client to eliminate 'Ask a human' action)
3. Yes. (We will ask client to keep 'Move' action)

4. Yes. (We will ask client to keep 'Pick', 'Place' & 'Pour' actions)
5. No. (We will ask client to eliminate 'Open Drawer' action)
6. Yes. (We will ask client to eliminate 'Push & Pull' actions)

4.6.2 Tiago & Yumi Counter Scenario

In this scenario, there are two counters (Counter1 & Counter2) available in the environment. Each of these counters has three regions (Left, Right & Middle). Yumi is placed in front of Counter2 and Counter1 is out of reach for Yumi since it does not have navigation capabilities. In the same environment there are three objects (BlockA, BlockB & BlockC) available. In addition to Yumi, Tiago is also present in the environment.

In the initial problem PDDL file, all of the objects are on top of Counter2 and Yumi has to implement a Stack & Unstack task using these objects. The task description can be seen in Figure 4.5a. However, an unexpected situation occurs when the perception module detects BlockB on Counter1 instead of Counter2 as in Figure 4.5b. Therefore, reasoning should be implemented in order to regenerate the domain and problem PDDL files to solve the specified task. The specified task requires BlockB to be brought on Counter2 so that Yumi can pick it. In this task, the reasoner should include Tiago into the task so that Tiago brings the BlockB from Counter1 to Counter2 for Yumi to complete the Stack & Unstack task. The environment can be visualized as in Figure 4.5.

This task specific domain PDDL file for Tiago should contain the following relevant actions:

- Pick
- Place
- Move

This task specific domain PDDL file for Yumi should contain the following relevant actions:

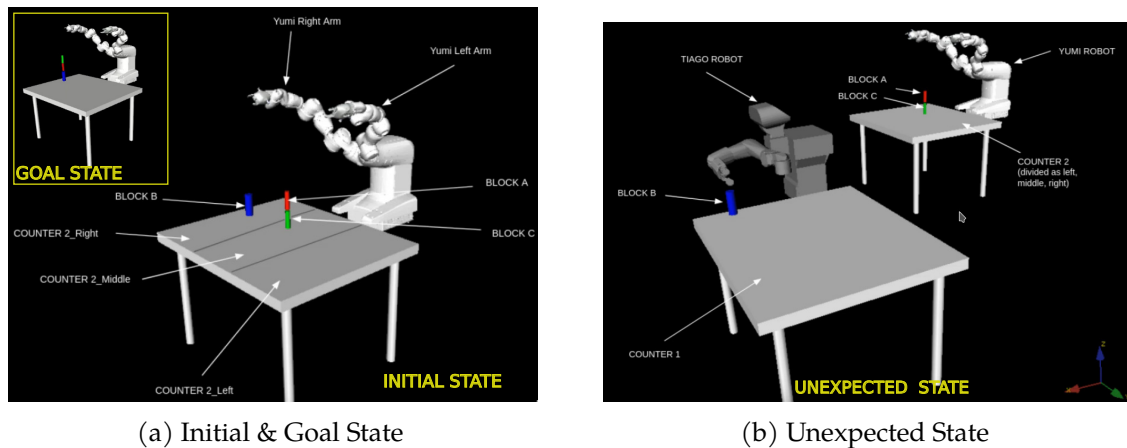
- Pick
- Stack
- Unstack

In addition to the task specific domain PDDL, the problem PDDL file should include *tiago at Counter2* as a goal state.

It can be seen that the objects also have *LocationName* info. This specifies the region where the object is detected at. The Regions are detected through Aruco Markers as well.

In order to fill both the domain and problem PDDL files, following sequence of queries are reasoned:

1. What are the available robots in the environment?
2. What are the capabilities of these robots?



(a) Initial & Goal State

(b) Unexpected State

Figure 4.5: Yumi Manipulation Task

3. Which symbolic regions can be reached by these robots using their capabilities?
4. Where is the current location of the Object of Interest? (ObjectC in this case)

5 Results

5.1 Deep Object Pose Estimation

This section presents the results obtained after training the network with the revised training scenario described in Section 3.1.4.4. Training loss convergence graph can be seen in Figure 5.1. In order to analyze the evolution of the output belief maps the target images in Figure 5.2 are forwarded through the network in the training process in epochs number 1, 3, and 60, the output belief maps generated by the network are saved, and the target belief maps for the given target images are compared for the same epochs in Figure 5.3, 5.4, and 5.5.

One important note is, the target images are not forwarded through the network in their original form but with Gaussian noise, random rotation and translation added as in Figure 5.2 in order to reduce over-fitting.

Another critical note is, in this thesis, there is no labeled real data available for test and validation purposes. The only possible test and validation dataset can be created using synthetic data which is not used for training. However, since these images are again captured within the same setup, it would be highly correlated with the training data. Therefore, in this work, the trained network is not analyzed with a test dataset and test loss graph is not presented. As a future work, real images can be captured and labeled for test purposes.

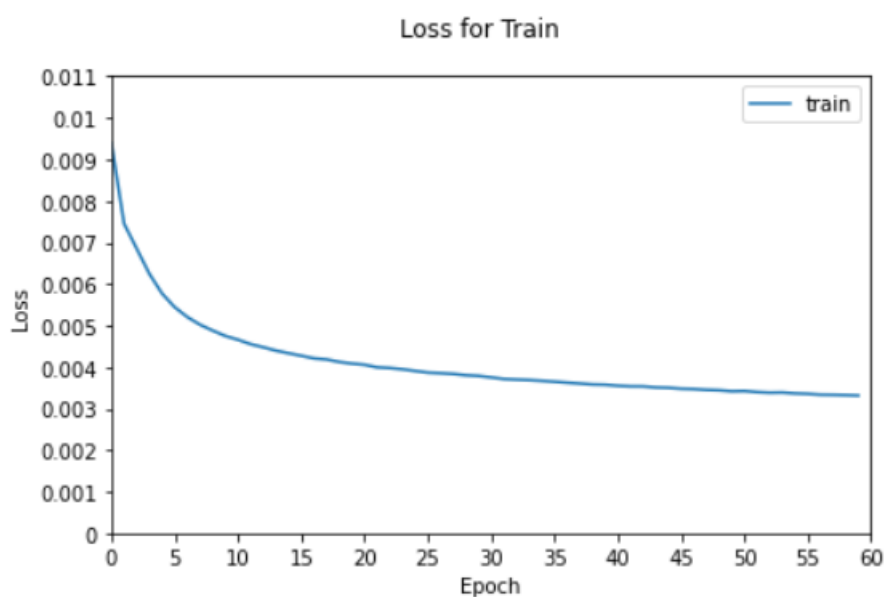
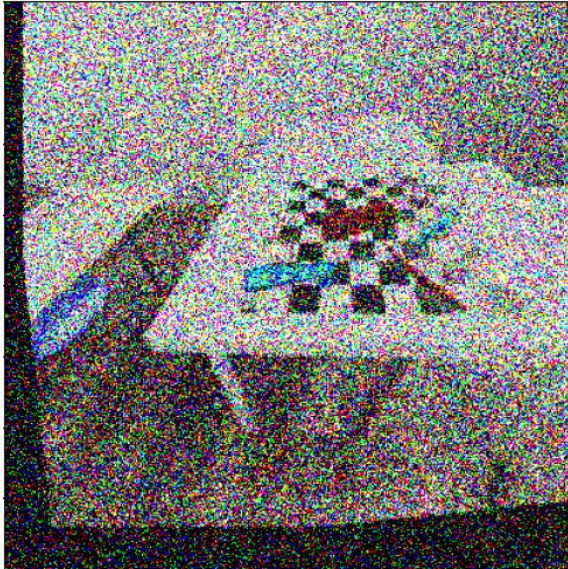


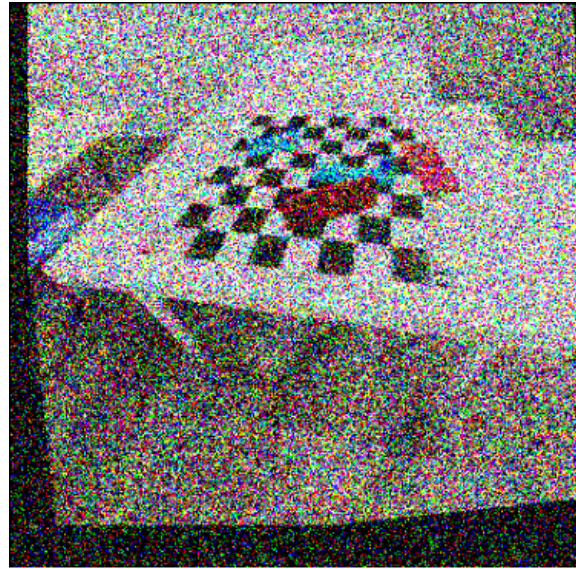
Figure 5.1: Loss Graph for Training

However, when the trained network is tested with the exact same target image in Figure 5.2c after loading the network weights, the obtained belief map is as in Figure 5.6b. From this result, it can be seen that the objects features are not learned correctly.

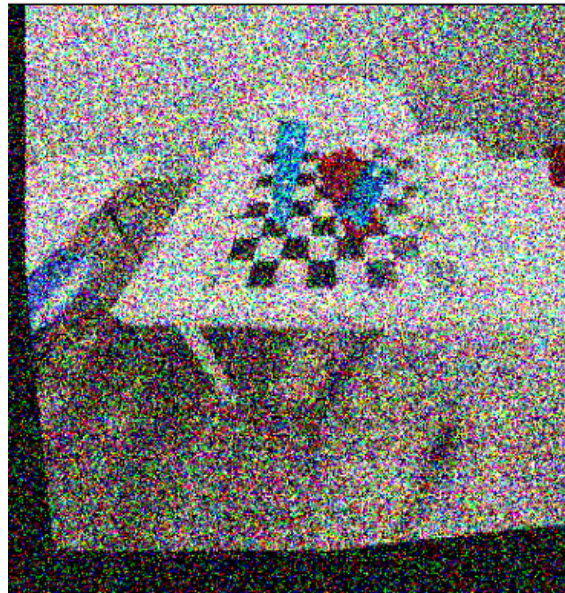
Another test case is with the input in Figure 5.7a and the output belief map is as in Figure 5.7b. The possible reasons behind this outcome is explained in *Conclusions* Section.



(a) Target Image Example for Epoch 1



(b) Target Image Example for Epoch 3



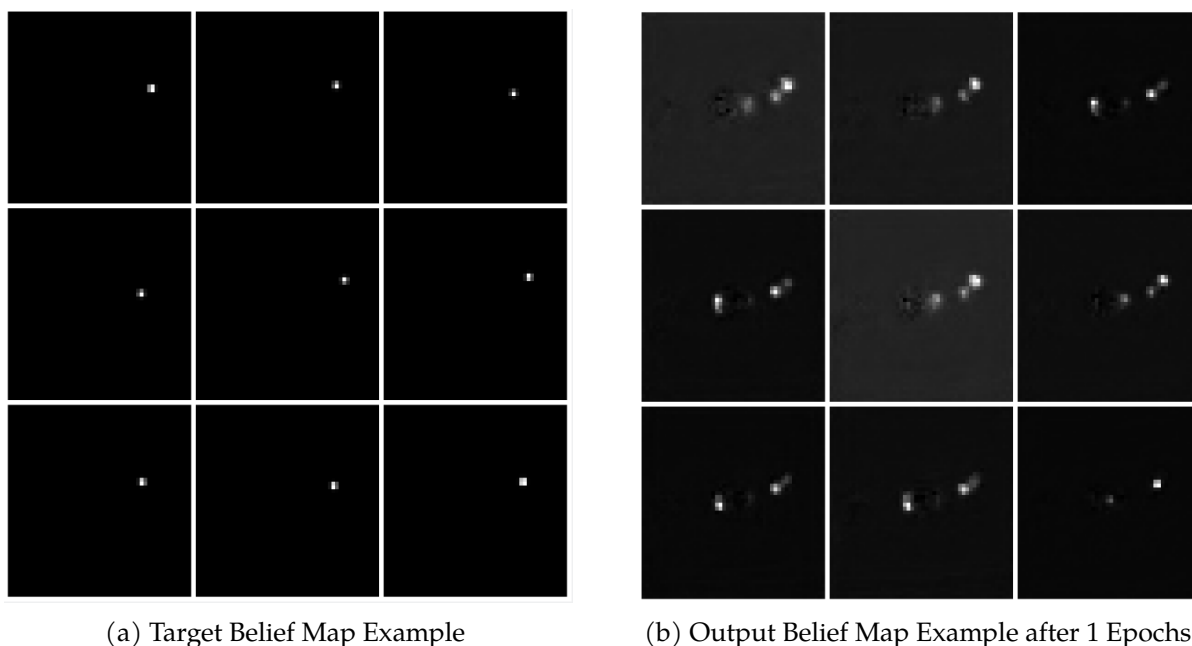
(c) Target Image Example for Epoch 60

Figure 5.2: Example Target Images Saved in Training for Epoch 1,3 and 60

5.2 Reasoning Module

This section presents the results for Tiago & Yumi Scenario as defined in Section 4.6 since the complete procedure described in Section 4.4 is implemented for this specific scenario. The same implementation can also be extended to Tiago Kitchen scenario in the future, which is also described in Section 4.6, which requires generating new global PDDL files and new actions.

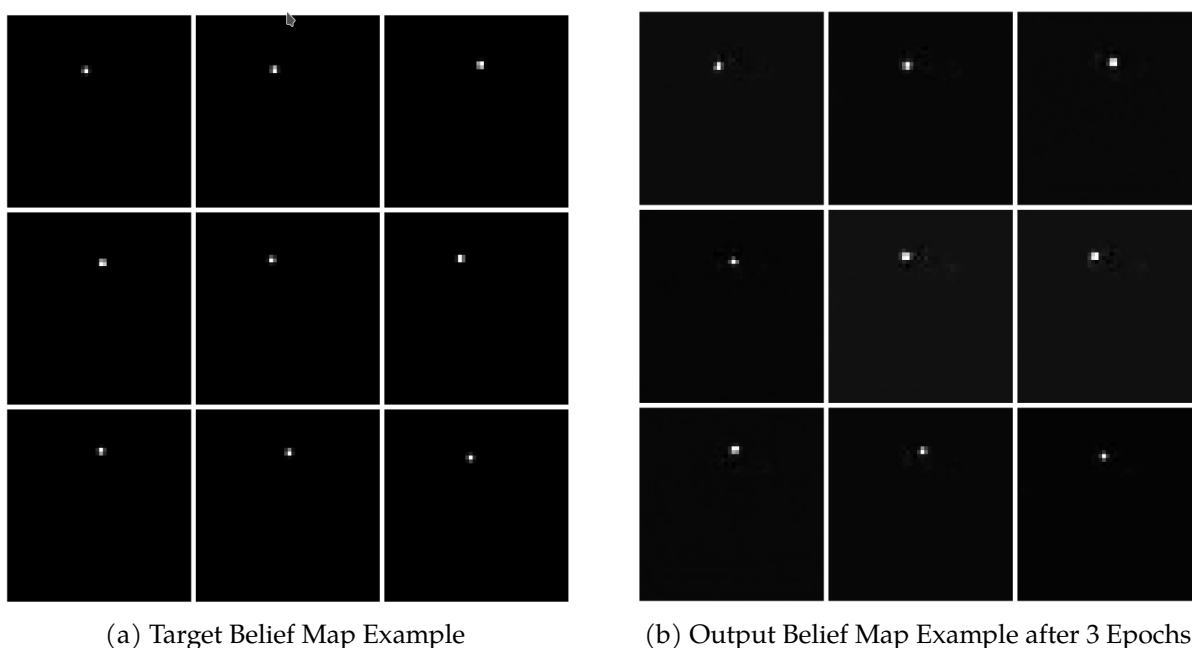
Before automatically generating PDDL files, the knowledge layer is created for Tiago & Yumi scenario. Ontology file created for Tiago & Yumi Scenario, which can be found in [61] under */pyswip/ontologies* directory, can be examined through the modeling hierarchy graph, which shows the hierarchical relation between the classes generated, as in Figure A.1. Moreover, the



(a) Target Belief Map Example

(b) Output Belief Map Example after 1 Epochs

Figure 5.3: Target & Output Belief Map Comparisons after 1 Epoch of Training



(a) Target Belief Map Example

(b) Output Belief Map Example after 3 Epochs

Figure 5.4: Target & Output Belief Map Comparisons after 3 Epoch of Training

instantiation knowledge, which shows the object properties between the instances, can be seen in Figure A.2. The different colored arrows in instantiation knowledge show different object properties created. In this specific figure, the instance *tiago* is visualized, however, other instances can also be visualized through this *OntoGraph* feature within Protege editor. Both of these visuals are generated with Protege ontology editor. Same figures can be obtained for Tiago in Kitchen scenario using the same ontology tools.

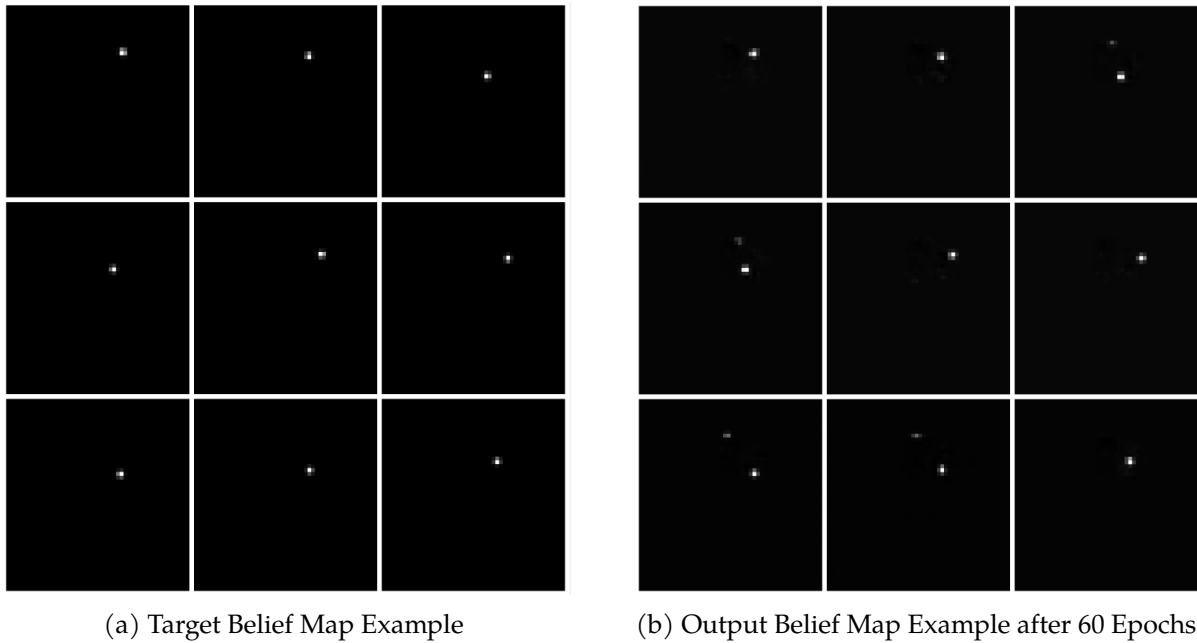


Figure 5.5: Target & Output Belief Map Comparisons after 60 Epochs of Training

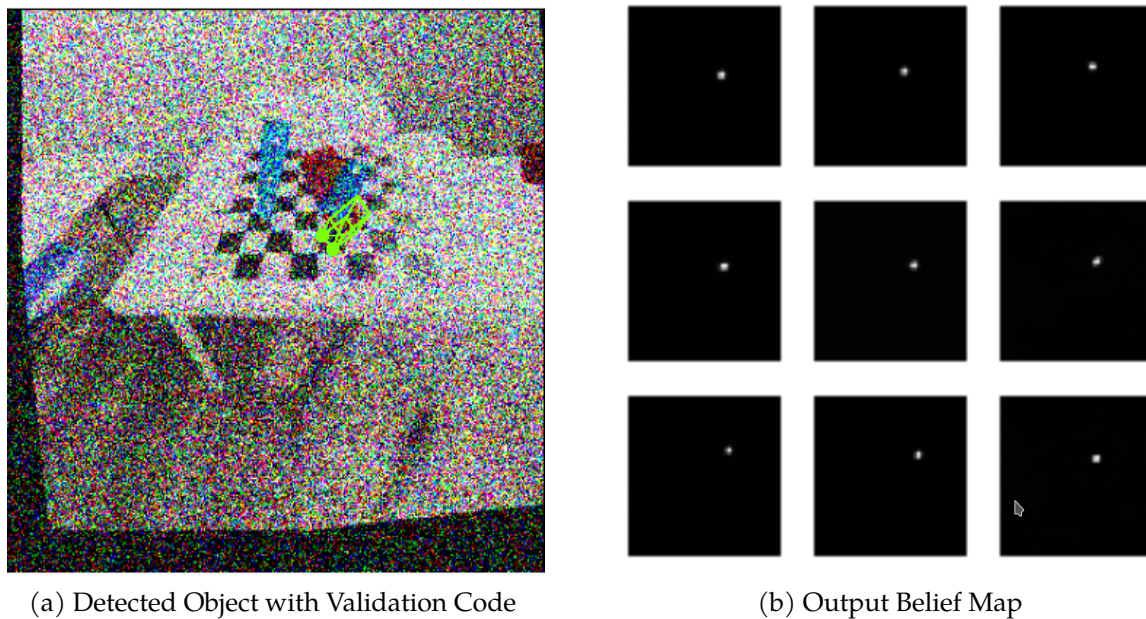
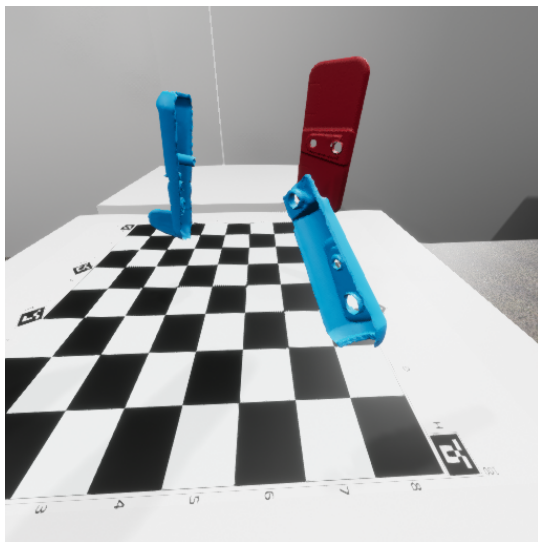


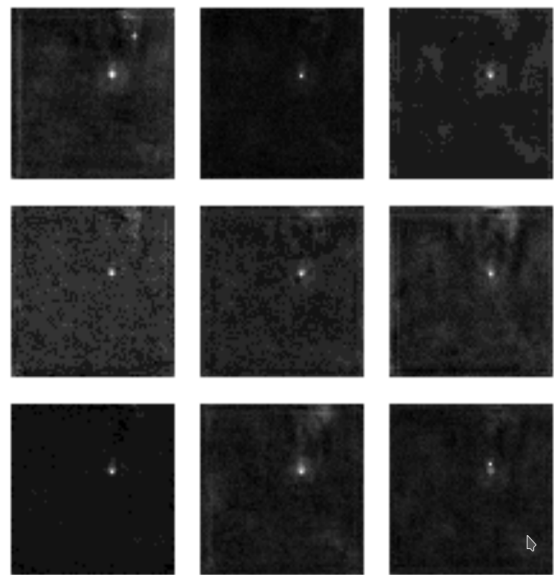
Figure 5.6: Validation Code Trial 1

The Prolog file generated for the same scenario can be found in the project repository [61] under */pyswip/prolog* directory.

Global PDDL domain file provided Tiago & Yumi scenario is as in Listing 4. Task specific domain PDDL file for the same scenario after the reasoning process are as in Listing 5 & 6 for Tiago and Yumi. The video of the simulation for Tiago & Yumi Scenario is available at https://youtu.be/MI7N0s1C_S0.



(a) An Example Test Input Image



(b) Output Belief Map

Figure 5.7: Validation Code Trial 2

6 Cost Analysis

The project has been completed from February to September (32 weeks) with an average of 4 hours a day. Four of these weeks are from August, therefore, in August only the student working cost and the personal laptop usage are included into the calculations since the lab was closed during August. Considering 5 working days a week, the total project hours are 640 hours, 80 of which includes only the personal laptop setup and the student working hours.

The hardware equipment involves a PC workstation and the personal laptop. Since the development of the project was done at Robotics Lab of the Institute of Industrial and Control Engineering (IOC-UPC), the depreciated cost of the workstation used has been added. The useful life of the workstation and the personal laptop is considered as 5 years when operated at 10 hours a day 7 days a week. This gives a useful life of 18,250 hours. The workstation PC is employed for 60% of the project hours that are spent in the lab and for the 40% of the overall time spent in the lab, personal laptop is used. The working hours of the student is considered the same as the total project hours. ETSEIB recommends the salary for the students will be considered of 8 €/h. Supervision and meeting hours with the director of the project and other members of the staff of the laboratory will be considered of 60 hours in total with an average cost of 30 €/h.

The electricity consumption is considered for the workstation PC, personal laptop and Kinect Camera. The average electrical cost in 2021 is taken as 0.21 €/kWh. The energy consumption for the personal laptop is estimated as 68.5 W, for 304 hours it makes 20.82 kWh. The energy consumption for the workstation PC is estimated as 200 W, for 336 hours it makes 67.2 kWh. On the other hand, Kinect Camera has a consumption of 12 W, for 20 hours which makes 0.24 kWh.

Table 6.1 presents the costs described. The total cost of the project is 6972.85 €.

Table 6.1: Cost table (*Variable cost of workstation PC has been computed dividing their fixed cost by their life expectancy in hours. Variable cost of electric consumption of each system have been computed by multiplying the power consumption by the average price of electricity as described in the Cost section*)

COST HEAD	Description	Fixed Cost €	Life Expectancy h	Variable cost €/h	Utilization Time h	Cost to Project €
Hardware Equipment	Workstation PC	1400	18,250	0.077	336	25.78
	Personal Laptop	500	18,250	0.027	304	8.33
	Kinect Camera	300	26,280	0.011	20	0.22
Electric Consumption	Workstation PC	-	-	0.042	336	14.11
	Kinect Camera	-	-	0.0024	20	0.048
	Personal Laptop	-	-	0.014	304	4.37
Student Working Hours	-	-	-	8	640	5120
Supervisor Working Hours	-	-	-	30	60	1800
Total Cost						6972.85 €

7 Environmental & Social Impact

This section discusses how the suggested reasoning-based robotics manipulation framework can impact both the environment and the society.

7.1 Environmental Impact

Flexibly configuring the task planning problems autonomously in robotics applications is expected to increase the integration of service robots into the daily and industrial routines. Increasing the demand towards service robots brings up concerns regarding powering these robots through sustainable and clean resources. Taking the enhancements within the powering industry into account, this may no longer be a matter with the elongated cycle life, efficient manufacturing and recycling strategies. On the other hand, if carefully managed, changes in automation levels especially among service robots could create environmental improvements compared with the processes used today.

7.2 Social Impact

Many efforts in industrial and service robotics pursue making mobile manipulators able to act autonomously in semi-structured human environments. The final aim is to actually make them able to be robot co-workers at the factory floor or robot helpers at home. The framework proposal presented in this thesis eases the adaptation of service robots into these semi-structured human environments. This advancement in service robots would shape the employment structure, especially for the manufacturing industry including non-complex tasks. Moreover, more complex tasks can be facilitated with the help of these improvements. Furthermore, these advances increase the utilization of robots for wide range of personal tasks in the daily routine of humans. Therefore, owning robot helpers in households can be the new normal.

Conclusions

In this work, regarding the Perception Module, Deep Object Pose Estimation implementation is adapted into Robotics Lab of the Institute of Industrial and Control Engineering (IOC-UPC). Following conclusions are drawn which are also the possible reasons behind the overall performance of the trained network, as presented in Section 5.1, of the module:

1. Since the synthetic data generation environment in Unreal Engine 4 is created from scratch, the following difficulties are faced during the implementation and these may also have an impact on the overall performance of the trained network especially for the real world usage:
 - (a) The created environment for photo-realistic image generation is not realistic enough. Therefore, in the future, the available environments can be imported to Unreal Engine through Epic Games Launcher. This requires building the Epic Games Launcher to the Linux environment in the lab through the software Lutris as mentioned earlier. With the help of high variety of available realistic environments in Unreal Engine Marketplace, adequate diversity of scenes can also be introduced into the photo-realistic training dataset.
 - (b) Selecting randomization parameter (i.e. lighting, camera pose, object poses) ranges can be hard to tune and they have a big impact on the content of the training dataset generated. The dataset content should be diverse enough to represent different scenes.
2. In addition to the difficulties mentioned above, the following aspects regarding the training dataset can have an impact on the trained network performance:
 - (a) The toy plane models that are used as distractors for the photo-realistic images are similar to each other in color and shape. This may cause difficulties within feature extraction. Therefore, different models can be imported into the environment and used as distractors during training data generation.
 - (b) The actual Falling Things Dataset to generate photo-realistic images capture data through two cameras placed in the environment. However, in this thesis, only a single camera is used to capture the scene.
 - (c) Training dataset folder consists of sub-folders for different environment setups and different locations in the same environment. Depending on how the data loader treats this folder structure, shuffling may not handled correctly since the images are in different folders. It may be shuffling only the content in the same directory but not all the training content. One proof of this is seeing the same location for target images saved during the training as it can be seen in Figure 5.2.
3. Finally, following training aspects can also have an effect on the performance of the network:
 - (a) Gradient scaling is being used in order to prevent the gradients from flushing to zero. It multiplies the networks losses by a scale factor. This functionality is used with its default values as suggested in the original project. However, the parameters of the function (init_scale, growth_factor, backoff_factor, growth_interval [62]) may need

to be tuned for this specific training implementation

- (b) The final training loss is recorded as 0.0003-0.0004 for the example training scenarios discussed in the original project repository. However, it is ~ 0.003 for the training in this thesis. This may be due to being stuck at a local minima since the convergence loss graph seems to be stable but not oscillating around a value.
- (c) Hyper-parameter tuning for the training also affects the trained network. However, regarding the learning rate selection, Deep Object Pose Estimation uses Adam adaptive learning rate optimization algorithm for training, so it automatically adjusts the learning rate over time. It is expected to see the loss going down immediately after starting the training, therefore, the learning rate can be left untouched so that Adam optimizer can modify it properly. By looking at the loss convergence graph in Figure 5.1, the drop in the beginning of the training can be seen clearly.

As a future work:

- Another alternative to using Unreal Engine to generate synthetic data can be the renderer called NVISII [63] which is a python-enabled ray tracing based renderer built on top of NVIDIA OptiX (C++/CUDA backend). This may be easier to work with since Unreal Engine is a complex interface including lots of different aspects and requires more experience to manipulate these elements.
- The original training code can be adopted with a more powerful setup and the original batchsize. Available GPU cloud servers can be checked for this purpose.
- As mentioned in the Results for Deep Object Pose Estimation in Section 5.1, no real images are acquired from the Kinect camera and labeled. Therefore, test and validation datasets are not created, since generating them in the same synthetic image generation setup would create a high correlation between them and the training dataset. Therefore, as a next step, non-correlated dataset can be created from real images. Even though there are plenty of labeling tools available for 2D bounding box of the objects of interest, labeling the 3D bounding box of the objects is required for this work. Since labeling 3D bounding boxes for each image is an expensive process, an alternative approach can be annotating 3D objects in a video clip and populating them to all frames in the clip as studied in [64]. Nevertheless, this involves an additional interface implementation for labeling process.

In this work, regarding the Reasoning Module, the main focus was the development of tools to provide robots with manipulation capabilities to make them autonomous enough to automatically plan and execute the tasks, continually adapting to possible changes in the environment. In order to achieve this, an already existing framework called Perception and Manipulation Knowledge (PMK) is taken as a template. PMK ontology is extended by including new instances, object properties, and data properties under the same hierarchical class structure. Its Prolog file is also extended by adding new predicates to query over the ontology. With assistance of this, the new knowledge layer consists of information about the actions that can be implemented by the robots available in the environment. This extension requires broadening the object features related to these actions.

The proposal has first dealt with the development of reasoning capabilities to reason on the environment detected by the perception module (objects and available robots) and on the task

goal to be achieved in order to find out the required actions to solve the task, and to assist the task planner with the automatic generation of the PDDL files. For this purpose, a PDDL parser package called Universal PDDL Parser is included into the ROS interface created which queries over the knowledge and generates the PDDL files respectively in an autonomous manner. The adaptive task and motion planning capabilities of the proposed framework is a step towards making robots more aware, smarter and reactive. As a future work:

- This work can be extended by adding more actions that the robots can execute and incorporate the human operator as an agent so as to allow robots to play the co-worker role.
- The overall system can be made even more robust and smart when a functioning Deep Object Pose Estimation module is integrated into the system. Since the detection can not be achieved through the Perception Module explained in Section 3, Aruco Marker detection was adopted into the overall system as the perception module which was already being used for detection purposes in Robotics Lab of the Institute of Industrial and Control Engineering (IOC-UPC).

Acknowledgements

I would like to express my special thanks to my advisor Prof. Dr. Jan Rosell Gratacos for giving me the opportunity to work in his lab on this project. His immense knowledge enabled me to gain insight into robotics field.

I am especially thankful to Mohammed Diab Elsayed Sharafeldeen for his continuous support, patience and enthusiasm during this whole process. Thanks to his guidance, I learned so many things and extended my vision as a robotics engineer.

My sincere thanks also goes to Leopold Palomo Avellaneda for his extensive expertise in solving my issues I came across with regarding the hardware setup.

I am grateful to my lab friend Parikshit Verma for his creative ideas to deal with the problems I faced in the lab.

I would not have finished this journey without the love and support of my family even from 3000 km away.

Appendices

Appendix A

```

1  for batch_idx, targets in enumerate(loader):
2
3      data = Variable(targets['image'].cuda())
4
5      # Runs the forward pass with autocasting.
6      with amp.autocast():
7          output_belief, output_affinities = net(data)
8
9          target_belief = Variable(targets['beliefs'].cuda())
10         target_affinity = Variable(targets['affinities'].cuda())
11
12         loss = None
13
14         # Belief maps loss
15         for l in output_belief: #output, each belief map layers.
16             if loss is None:
17                 loss = ((1 - target_belief) * (1-target_belief)).mean()
18             else:
19                 loss_tmp = ((1 - target_belief) * (1-target_belief)).mean()
20                 loss += loss_tmp
21
22         # Affinities loss
23         for l in output_affinities: #output, each belief map layers.
24             loss_tmp = ((1 - target_affinity) * (1-target_affinity)).mean()
25             loss += loss_tmp
26
27         if train:
28             # Scales loss. Calls backward() on scaled loss to create scaled
29             # gradients
30             # Backward passes under autocast are not recommended.
31             # Backward ops run in the same dtype autocast chosen for
32             # corresponding forward ops.
33             scaler.scale(loss).backward()
34
35             if batch_idx % (opt.batchsize // opt.subbatchsize) == 0:
36                 if train:
37                     # scaler.step() first unscales the gradients of the
38                     # optimizer's assigned params.
39                     # If these gradients do not contain infs or NaNs, optimizer.
40                     # step() is then called,
41                     # otherwise, optimizer.step() is skipped.
42                     scaler.step(optimizer)
43
44                     # Updates the scale for next iteration.
45                     scaler.update()
46                     nb_update_network+=1
47                     optimizer.zero_grad()

```

Listing 2: Revised Training Loop

```

1 ...
2 #Read the detected objects from object_list.xml
3 objs = initialize_objects()
4
5 #Find spatial relations between the detected objects
6 spatial = []
7 for obj1 in objs:
8     for obj2 in objs:
9         if obj1.index == obj2.index:
10            continue
11            response_spatial = call_reasoner(["",obj2.Pose,obj1.Pose],["
find_spatial_relation",("Obj1","Obj2","Relation")])
12            spatial.append(response_spatial.responses[0].responses[2])
13
14 #Retrieve the initial and goal state in symbolic form
15 for obj in objs:
16     response_init = call_reasoner(["",obj.Pose],["retrieve_sym_init",("ObjPose",
"SymbRgn")])
17     obj.init = response_init.responses[0].responses[1]
18
19 response_goal = call_reasoner(["",""],["retrieve_sym_goal",("Task","SymbGoal")])
20
21 response_robot = call_reasoner(["",""],["find_robot",("Room","Robot")])
22
23 #Append all the found robots into a robot array. It will be ['yumi','tiago'] for
icra.pl
24 robots = []
25 for i in range(len(response_robot.responses[0].responses)/2):
26     robots.append(response_robot.responses[0].responses[2*i])
27
28 print("Available robots are ", robots)
29
30 #Check the capabilities for every robot in the environment
31 for rob in range(len(robots)):
32     response_cap = call_reasoner(["",robots[rob]],["find_robot_capability",("
Robot","Capability")])
33
34 for rob in range(len(robots)):
35     response_reach = call_reasoner([objs[2].init,robots[rob]],["find_robot_reach
",("Robot","Region")])
36     #Choose the robot which returns true to this predicate, which is tiago in
this case since yumi can't reach Region_C but tiago can since it has
37     #navigation capabilities
38     if response_reach.responses[0].responses[0] == "True":
39         chosen_robot = robots[rob]
40 ...

```

Listing 3: Reasoning Predicates in Order

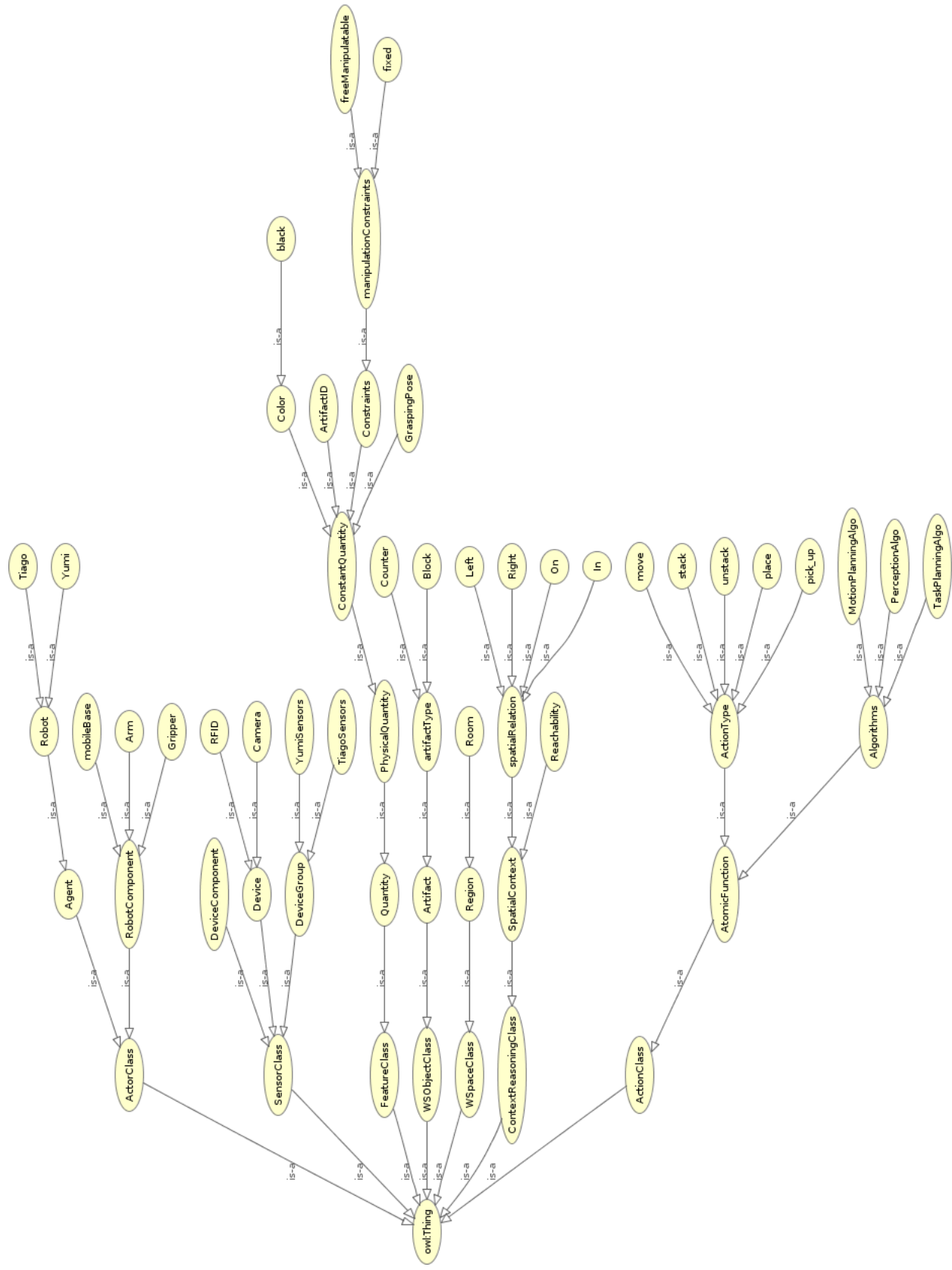


Figure A.1: Model Hierarchy Graph for Tiago & Yumi Ontology

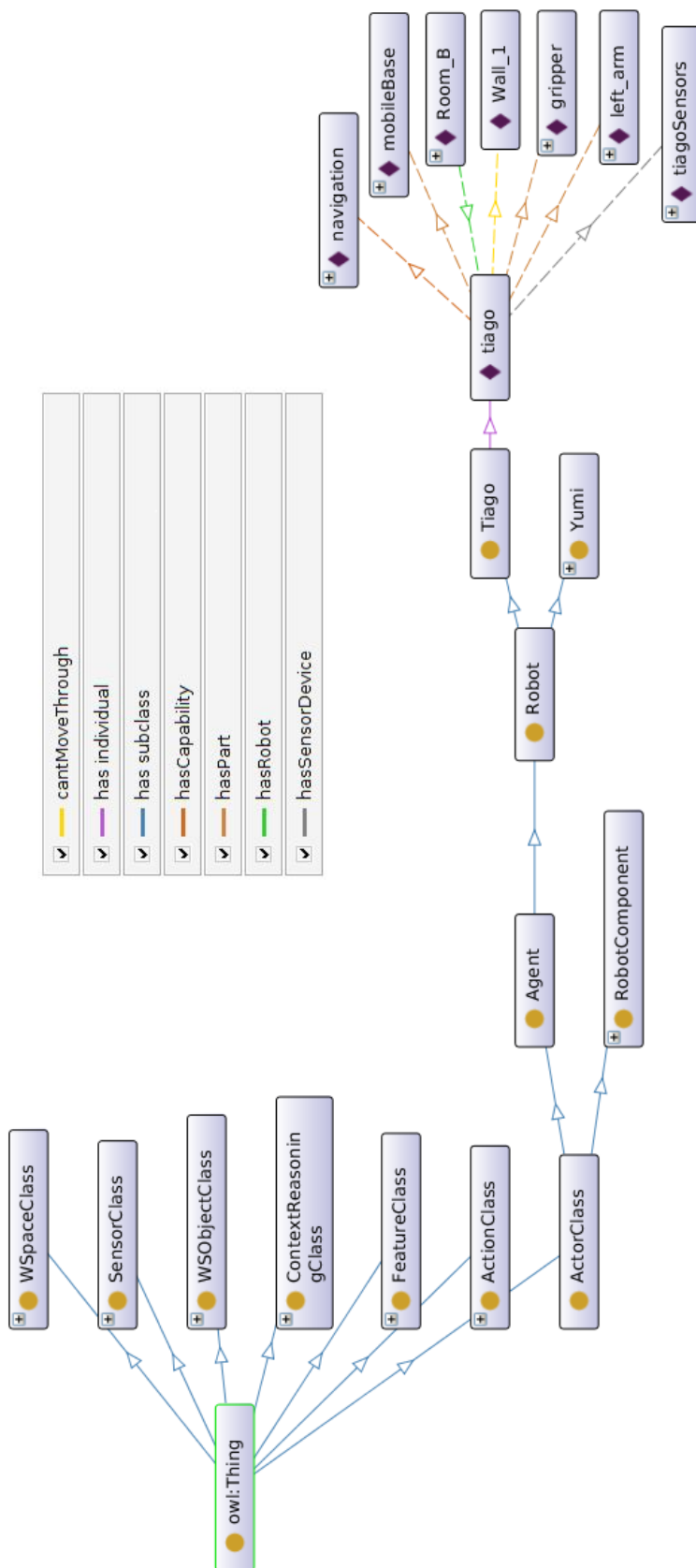


Figure A.2: Instantiation Knowledge for Tiago & Yumi Ontology

Appendix B

```

1 (define (domain blocksworld)
2
3 (:types obstacle robot location)
4
5   (:predicates
6     (clear ?obs)
7     (on-table ?obs)
8     (handEmpty ?rob)
9     (holding ?rob ?obs)
10    (on ?obs ?underObs)
11    (at ?rob ?from)
12    (in ?obs ?from)
13    (mobile ?rob))
14
15
16 (:action move
17 :parameters (?rob - robot ?from - location ?to - location)
18 :precondition (and (at ?rob ?from) (mobile ?rob))
19 :effect (and (at ?rob ?to)
20            (not (at ?rob ?from))))
21
22 (:action pick
23 :parameters (?rob - robot ?obs - obstacle ?from - location)
24 :precondition (and (handEmpty ?rob) (in ?obs ?from)
25                  (at ?rob ?from) (clear ?obs))
26 :effect (and (holding ?rob ?obs)
27            (not (handEmpty ?rob)) ))
28
29 (:action place
30 :parameters (?rob - robot ?obs - obstacle ?from - location)
31 :precondition (and (holding ?rob ?obs)
32                  (at ?rob ?from))
33 :effect (and (handEmpty ?rob) (in ?obs ?from)
34            (not (holding ?rob ?obs)) ))
35
36 (:action stack
37 :parameters (?rob -robot ?obs - obstacle ?underObs - obstacle ?from -
38            location)
39 :precondition (and (clear ?underObs) (holding ?rob ?obs) (not (mobile ?rob)
40            )
41                (in ?underObs ?from) (at ?rob ?from))
42 :effect (and (clear ?obs) (handEmpty ?rob) (on ?obs ?underObs) (in ?obs ?
43            from)
44            (not (clear ?underObs)) (not (holding ?rob ?obs))))
45
46 (:action unstack
47 :parameters (?rob -robot ?obs - obstacle ?underObs - obstacle ?from -
48            location)
49 :precondition (and (on ?obs ?underObs) (clear ?obs) (not (clear ?underObs))
50            (not (mobile ?rob))
51                (handEmpty ?rob) (in ?underObs ?from) (at ?rob ?from) )
52 :effect (and (holding ?rob ?obs) (clear ?underObs)
53            (not (on ?obs ?underObs )) (not (in ?obs ?from )) (not (clear ?
54            obs)) (not (handEmpty ?rob))))
55 )

```

Listing 4: Global Domain PDDL File

```
1 (define (domain blocksworld)
2
3 (:types obstacle robot location)
4
5 (:predicates
6   (clear ?obs)
7   (on-table ?obs)
8   (handEmpty ?rob)
9   (holding ?rob ?obs)
10  (on ?obs ?underObs)
11  (at ?rob ?from)
12  (in ?obs ?from)
13  (mobile ?rob))
14
15
16 (:action move
17 :parameters (?rob - robot ?from - location ?to - location)
18 :precondition (and (at ?rob ?from) (mobile ?rob))
19 :effect (and (at ?rob ?to)
20   (not (at ?rob ?from))))
21
22 (:action pick
23 :parameters (?rob - robot ?obs - obstacle ?from - location)
24 :precondition (and (handEmpty ?rob) (in ?obs ?from)
25   (at ?rob ?from) (clear ?obs))
26 :effect (and (holding ?rob ?obs)
27   (not (handEmpty ?rob)) ))
28
29 (:action place
30 :parameters (?rob - robot ?obs - obstacle ?from - location)
31 :precondition (and (holding ?rob ?obs)
32   (at ?rob ?from))
33 :effect (and (handEmpty ?rob) (in ?obs ?from)
34   (not (holding ?rob ?obs)) ))
35 )
```

Listing 5: Task Specific Domain PDDL File for Tiago

```

1 (define (domain blocksworld)
2
3 (:types obstacle robot location)
4
5   (:predicates
6     (clear ?obs)
7     (on-table ?obs)
8     (handEmpty ?rob)
9     (holding ?rob ?obs)
10    (on ?obs ?underObs)
11    (at ?rob ?from)
12    (in ?obs ?from)
13    (mobile ?rob))
14
15 (:action pick
16 :parameters (?rob - robot ?obs - obstacle ?from - location)
17 :precondition (and (handEmpty ?rob) (in ?obs ?from)
18                  (at ?rob ?from) (clear ?obs))
19 :effect (and (holding ?rob ?obs)
20             (not (handEmpty ?rob))) )
21
22 (:action stack
23 :parameters (?rob -robot ?obs - obstacle ?underObs - obstacle ?from -
24             location)
25 :precondition (and (clear ?underObs) (holding ?rob ?obs) (not (mobile ?rob)
26                  (in ?underObs ?from) (at ?rob ?from))
27 :effect (and (clear ?obs) (handEmpty ?rob) (on ?obs ?underObs) (in ?obs ?
28             from)
29             (not (clear ?underObs)) (not (holding ?rob ?obs))))
30
31 (:action unstack
32 :parameters (?rob -robot ?obs - obstacle ?underObs - obstacle ?from -
33             location)
34 :precondition (and (on ?obs ?underObs) (clear ?obs) (not (clear ?underObs))
35             (not (mobile ?rob))
36             (handEmpty ?rob) (in ?underObs ?from) (at ?rob ?from) )
37 :effect (and (holding ?rob ?obs) (clear ?underObs)
38             (not (on ?obs ?underObs )) (not (in ?obs ?from )) (not (clear ?
39             obs)) (not (handEmpty ?rob))))
40 )

```

Listing 6: Task Specific Domain PDDL File for Yumi

Bibliography

- [1] M. Ghallab et al. *PDDL—The Planning Domain Definition Language*. 1998.
- [2] Jonathan Tremblay et al. “Deep Object Pose Estimation for Semantic Robotic Grasping of Household Objects”. In: *Conference on Robot Learning (CoRL)*. 2018. URL: <https://arxiv.org/abs/1809.10790>.
- [3] *Deep Object Pose Estimation - ROS Inference*. URL: https://github.com/NVlabs/Deep_Object_Pose.
- [4] *Unreal Engine*. URL: <https://www.unrealengine.com/en-US/>.
- [5] Thang To et al. *NDDS: NVIDIA Deep Learning Dataset Synthesizer*. https://github.com/NVIDIA/Dataset_Synthesizer. 2018.
- [6] Morgan Quigley et al. “ROS: an open-source Robot Operating System”. In: *ICRA Workshop on Open Source Software*. Vol. 3. 2009, p. 5.
- [7] Stanford. *Protege*. 2007. URL: <http://protege.stanford.edu/>.
- [8] Jan Wielemaker et al. *SWI-Prolog*. 2010. arXiv: 1011.5332 [cs.PL].
- [9] *Universal PDDL Parser Multiagent*. URL: <https://github.com/aig-upf/universal-pddl-parser-multiagent>.
- [10] OpenCV. *Detection of ArUco Markers*. URL: https://docs.opencv.org/4.5.2/d5/dae/tutorial_aruco_detection.html.
- [11] H. Kato and M. Billinghurst. “Marker tracking and HMD calibration for a video-based augmented reality conferencing system”. In: *Proceedings 2nd IEEE and ACM International Workshop on Augmented Reality (IWAR'99)* (1999), pp. 85–94.
- [12] M. Fiala. “ARTag, a fiducial marker system using digital techniques”. In: *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*. Vol. 2. 2005, 590–596 vol. 2. DOI: 10.1109/CVPR.2005.74.
- [13] Edwin Olson. “AprilTag: A robust and flexible visual fiducial system”. In: *2011 IEEE International Conference on Robotics and Automation*. 2011, pp. 3400–3407. DOI: 10.1109/ICRA.2011.5979561.
- [14] Francisco J. Romero-Ramire, Rafael Muñoz-Salinas, and R. Medina-Carnicer. “Fractal Markers: A New Approach for Long-Range Marker Pose Estimation Under Occlusion”. In: *IEEE Access* 7 (2019), pp. 169908–169919. DOI: 10.1109/ACCESS.2019.2951204.
- [15] Meghshyam G. Prasad, Sharat Chandran, and Michael S. Brown. “A Motion Blur Resilient Fiducial for Quadcopter Imaging”. In: *2015 IEEE Winter Conference on Applications of Computer Vision*. 2015, pp. 254–261. DOI: 10.1109/WACV.2015.41.
- [16] Mahdi Rad and Vincent Lepetit. *BB8: A Scalable, Accurate, Robust to Partial Occlusion Method for Predicting the 3D Poses of Challenging Objects without Using Depth*. 2018. arXiv: 1703.10896 [cs.CV].
- [17] Yu Xiang et al. *PoseCNN: A Convolutional Neural Network for 6D Object Pose Estimation in Cluttered Scenes*. 2018. arXiv: 1711.00199 [cs.CV].
- [18] Bugra Tekin, Sudipta N. Sinha, and Pascal Fua. *Real-Time Seamless Single Shot 6D Object Pose Prediction*. 2018. arXiv: 1711.08848 [cs.CV].
- [19] Lilian Weng. “Domain Randomization for Sim2Real Transfer”. In: *lilianweng.github.io/lil-log* (2019). URL: <http://lilianweng.github.io/lil-log/2019/05/04/domain-randomization.html>.
- [20] Amlan Kar et al. *Meta-Sim: Learning to Generate Synthetic Datasets*. 2019. arXiv: 1904.11621 [cs.CV].
- [21] M. Diab et al. “An Ontology Framework for Physics-Based Manipulation Planning”. In: *ROBOT 2017: Third Iberian Robotics Conference, vol. 1*. Springer, 2017, pp. 452–464. ISBN:

- 978-3-319-70833-1. DOI: [10.1007/978-3-319-70833-1_37](https://doi.org/10.1007/978-3-319-70833-1_37). URL: <http://hdl.handle.net/2117/113055>.
- [22] M. Ud Din, J. Rosell, and A. Akbari. “k-PMP: enhancing physics-based motion planners with knowledge-based reasoning”. In: *Journal of intelligent and robotic systems* (Sept. 2017), pp. 1–19. DOI: [10.1007/s10846-017-0698-z](https://doi.org/10.1007/s10846-017-0698-z). URL: <http://hdl.handle.net/2117/108313>.
- [23] Gi Hyun Lim et al. “Ontology-based unified robot knowledge for service robots in indoor environments”. In: *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans* (2011).
- [24] Guglielmo Gemignani et al. “Living with robots: Interactive environmental knowledge acquisition”. In: *Robotics and Autonomous Systems* 78 (2016), pp. 1–16. ISSN: 0921-8890. DOI: <https://doi.org/10.1016/j.robot.2015.11.001>. URL: <https://www.sciencedirect.com/science/article/pii/S0921889015002468>.
- [25] M. Diab et al. “PMK –A knowledge processing framework for autonomous robotics perception and manipulation”. In: *Knowledge-Based Systems* 19 (5 2019), p. 1166.
- [26] A. Olivares-Alarcos et al. “A Review and Comparison of Ontology-based Approaches to Robot Autonomy”. In: *The Knowledge Engineering Review* 34 (2019).
- [27] Epic Games. *Epic Games Launcher*. URL: <https://www.epicgames.com/store/en-US/>.
- [28] Unreal Engine. *Rendering Overview*. URL: <https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/Overview/>.
- [29] Unreal Engine. *Installing Unreal Engine*. URL: <https://docs.unrealengine.com/4.27/en-US/Basics/InstallingUnrealEngine/>.
- [30] Weichao Qiu, Fangwei Zhong, Yi Zhang, Siyuan Qiao, Zihao Xiao, Tae Soo Kim, Yizhou Wang, Alan Yuille. “UnrealCV: Virtual Worlds for Computer Vision”. In: (2017).
- [31] SanggunLee. *Dataset Synthesizer*. URL: https://github.com/SanggunLee/Dataset_Synthesizer/tree/UE4.25.4.
- [32] Unreal Engine. *Unreal Engine Marketplace*. URL: <https://www.unrealengine.com/marketplace/en-US/store>.
- [33] Lutris. *Lutris - Open Gaming Platform*. URL: <https://lutris.net/>.
- [34] Jan Rosell et al. “The Kautham Project: A teaching and research tool for robot motion planning”. In: *IEEE Int. Conf. on Emerging Technologies and Factory Automation*. 2014.
- [35] *Wavefront OBJ File Format*. URL: <https://www.loc.gov/preservation/digital/formats/fdd/fdd000507.shtml>.
- [36] *3D Texture Packs*. URL: <https://freepbr.com/>.
- [37] Unreal Engine. *Layered Materials*. URL: <https://docs.unrealengine.com/4.26/en-US/RenderingAndGraphics/Materials/LayeredMaterials/>.
- [38] Unreal Engine. *Rect Light*. <https://docs.unrealengine.com/4.27/en-US/BuildingWorlds/LightingAndShadows/LightTypes/RectLights/>. 2020.
- [39] *Blender*. URL: <https://www.blender.org/>.
- [40] *Send-To-Unreal Add-on*. <https://www.unrealengine.com/en-US/blog/download-our-new-blender-addons>. 2020.
- [41] Jonathan Tremblay, Thang To, and Stan Birchfield. “Falling Things: A Synthetic Dataset for 3D Object Detection and Pose Estimation”. In: *CVPR Workshop on Real World Challenges and New Benchmarks for Deep Learning in Robotic Vision*. June 2018.
- [42] Unreal Engine. *Setting Up Collisions With Static Meshes*. URL: <https://docs.unrealengine.com/4.27/en-US/WorkingWithContent/Types/StaticMeshes/HowTo/SettingCollision/>.
- [43] Unreal Engine. *Blueprint Visual Scripting*. URL: <https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/>.
- [44] Unreal Engine. *Custom Events*. URL: <https://docs.unrealengine.com/4.26/en-US/ProgrammingAndScripting/Blueprints/UserGuide/Events/Custom/>.

-
- [45] *Nvidia Dataset Utilities (NVDU)*. URL: https://github.com/NVIDIA/Dataset_Uutilities.
 - [46] *COCO Dataset*. URL: <https://cocodataset.org/#home>.
 - [47] Unreal Engine. *Volumes Reference*. URL: <https://docs.unrealengine.com/4.27/en-US/Basics/Actors/Volumes/>.
 - [48] Vincent Lepetit, Francesc Moreno-Noguer, and Pascal Fua. "EPnP: An accurate $O(n)$ solution to the PnP problem". In: *International Journal of Computer Vision* 81 (Feb. 2009). doi: 10.1007/s11263-008-0152-6.
 - [49] *PyTorch*. URL: <https://pytorch.org/>.
 - [50] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].
 - [51] *Deep Object Pose Estimation - ROS Inference - Default Training Hyperparameters*. URL: https://github.com/NVlabs/Deep_Object_Pose/blob/dfcb7103402ba251a6d747341db7ca8819df25e4/scripts/train.py#L1012.
 - [52] *GeForce RTX™ 2060 OC 6G (rev. 1.0)*. URL: <https://www.gigabyte.com/Graphics-Card/GV-N20600C-6GD-rev-10/sp#sp>.
 - [53] *Deep Object Pose Estimation for Single GPU System*. URL: https://github.com/osu-uwrt/Deep_Object_Pose/blob/b5c623d8dc274ea3c551be13f813c42f6e6c434f/scripts/train.py.
 - [54] MathWorks. *What Is Camera Calibration?* URL: <https://www.mathworks.com/help/vision/ug/camera-calibration.html>.
 - [55] *The Perspective and Orthographic Projection Matrix*. URL: <https://www.scratchapixel.com/lessons/3d-basic-rendering/perspective-and-orthographic-projection-matrix>.
 - [56] Semantic Web. *Web Ontology Language (OWL)*. URL: <https://www.w3.org/OWL/>.
 - [57] JavaPoint. *Unification in Prolog*. URL: <https://www.javatpoint.com/unification-in-prolog>.
 - [58] *SWI-Prolog – Installation on Linux*. URL: <https://www.swi-prolog.org/build/unix.html>.
 - [59] Yuce Tekol and Rodrigo Starr. *pyswip - PySWIP is a bridge between Python and SWI-Prolog. - Google Project Hosting*. URL: <https://code.google.com/p/pyswip/>.
 - [60] *pip, Python Package Installer*. URL: <https://pypi.org/project/pip/>.
 - [61] *PMK Python Interface*. https://gitioc.upc.edu/fatma.nur/pmk_python_interface/-/tree/ROS_INTERFACE/.
 - [62] PyTorch. *Gradient Scaling*. <https://pytorch.org/docs/stable/amp.html#gradient-scaling>.
 - [63] Nathan Morrical et al. *NVISII: NVIDIA Scene Imaging Interface*. <https://github.com/owl-project/NVISII/>. 2020.
 - [64] Adel Ahmadyan et al. *Objectron: A Large Scale Dataset of Object-Centric Videos in the Wild with Pose Annotations*. 2020. arXiv: 2012.09988 [cs.CV].