Bachelor Project

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Scene reconstruction from multiple RGB-D cameras and detection of the best additional camera viewpoint

Jan Jirman

Supervisor: Mgr. Karla Štěpánová, Ph.D.
Co-supervisor: Mgr. Radoslav Škoviera, Ph.D.
Field of study: Open informatics
Subfield: Software
May 2020

BACHELOR'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Jirman Jan** Personal ID number: **466370**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Science**

Study program: **Open Informatics**

Branch of study: **Software**

II. Bachelor's thesis details

Bachelor's thesis title in English:

Scene reconstruction from multiple RGB-D cameras and detection of the best additional camera viewpoint

Bachelor's thesis title in Czech:

Rekonstrukce scény z více RGB-D kamer a určení vhodného přídavného pohledu kamery

Guidelines:

The goal of the thesis is to reconstruct a scene viewed by multiple RGB-D cameras, find occlusions in the scene, and compute ideal viewpoint that would minimize the occlusion. The objects in the scene are of known type and shape. However, their position and orientation is unknown. The computed information about occlusion in the scene should contain uncertainty of the object pose detection. Based on this information, the proposed system should find a view that would minimize the occlusion and the uncertainty. The proposed system is to be utilized as a part of a collaborative robotic workplace with multiple cameras, with one camera attached to a robotic manipulator.

1. Preparation of experiments in both virtual and real environments.
2. Detection of objects based on AruCo markers from RGB-D data.
3. Fusion of data from multiple cameras.
4. Visualization of the detected objects.
5. Computation and visualization of scene occlusion by the detected objects.
6. Evaluation of the quality of the object pose and shape detection.
7. Viewpoint optimization based on the collected and computed data. The optimal viewpoint should minimize the amount of occlusion in the scene.
8. Incorporating restrictions of the real robot (at least a size of the workspace)
9. (optional) Extension to objects without AruCo markers.

Bibliography / sources:

1. Hu, Yinlin, et al. "Segmentation-driven 6d object pose estimation." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2019.
2. Vidal, Joel, et al. "A Method for 6D Pose Estimation of Free-Form Rigid Objects Using Point Pair Features on Range Data." Sensors 18.8 (2018): 2678.
3. Lepetit, Vincent, Francesc Moreno-Noguer, and Pascal Fua. "Epnp: An accurate o (n) solution to the pnp problem."International journal of computer vision 81.2 (2009): 155.
4. Rad, Mahdi, and Vincent Lepetit. "BB8: A scalable, accurate, robust to partial occlusion method for predicting the 3D poses of challenging objects without using depth." Proceedings of the IEEE International Conference on Computer Vision. 2017.
5. Hodaň, Tomáš, et al. "Detection and fine 3D pose estimation of texture-less objects in RGB-D images." 2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). IEEE, 2015.

Name and workplace of bachelor's thesis supervisor:

Mgr. Karla Štěpánová, Ph.D., Robotic Perception, CIIRC

Name and workplace of second bachelor's thesis supervisor or consultant:

Date of bachelor's thesis assignment: **12.02.2020** Deadline for bachelor thesis submission: **22.05.2020**

Assignment valid until: **30.09.2021**

Mgr. Karla Štěpánová, Ph.D. Supervisor's signature	Head of department's signature	prof. Mgr. Petr Páta, Ph.D. Dean's signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Acknowledgements

I would like to thank my thesis supervisor Mgr. Karla Štěpánová, Ph.D. for all the help with this work, valuable advice, consultations and comments provided to help me with this work. I would also like to thank Mgr. Radoslav Škoviera, Ph.D. for valuable consultation and advice with this work.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, May 22, 2020

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uved lveškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 22. května 2020

Abstract

In a multi-camera environment, scene reconstruction and merging data from multiple cameras is an important task with applications in numerous areas. In this work, we develop multiple methods of scene reconstruction and camera pose suggestion for an additional viewpoint at the scene that maximizes visible area. We create a software that implements those methods, for a use case scenario of a human-robot shared workspace with multiple fixed cameras and one camera on a movable robot arm, for additional viewpoint.

Keywords: Multicamera environment, Scene reconstruction, Camera pose suggestion

Supervisor: Mgr. Karla Štěpánová, Ph.D.

Co-supervisor: Mgr. Radoslav Škoviera, Ph.D.

Abstrakt

V prostředí s více kamerami je důležitá úloha správně složit informaci z těchto kamer a zrekonstruovat scénu. Tato úloha najde uplatnění v mnoha odvětvích. V této práci se zabýváme vývojem několika metod pro rekonstrukci scény a návrhu pozice a rotace kamery pro přidaný pohled na scénu, který maximalizuje viditelnou oblast. Vytvořili jsme software, který tyto metody používá jako součást většího projektu sdíleného pracoviště mezi člověkem a robotickou rukou. Na tomto pracovišti je několik pevných kamer a jedna přimontovaná kamera na robotické ruce, pro přídavný pohled.

Klíčová slova: Vícekamerové prostředí, rekonstrukce scény, návrh polohy kamery

Překlad názvu: Rekonstrukce scény z několika RGB-D kamer a detekce nejlepšího dalšího pohledu kamery

Contents

Figures

Tables

Glossary

- `Pose`, `6DoF` – transformation information using both position and rotation. We have opted for using quaternion to describe rotation. 6DoF is represented as a quaternion and translation vector: *q[a, b, c, d] + [x, y, z]*.
- `Detection`, `Object detection` – process of transforming raw pixel data to information about object type and its pose. This serves as input for this project.
- `Software pipeline` – All software and connections used for entire robot arm system (described in more details in [Sec: 5]). This thesis serves as one of many parts of this pipeline.
- `Ray` – Half line, starting at its origin and continues in only one direction, as defined by its direction vector.
- `Object type` – Definition (3D model, ...) of an object, which might be seen at the workspace, as described by knowledgebase.
- `Object instance` – One observed occurrence of an object. There can be many observed occurrences of object of the same type, at any given time.
- `Feature` – Structure consisting of a value of measured property in a data source (e.g.: an ID of an ArUco marker detected in an image) and its reconstructed 6DoF pose.

Chapter 1

Introduction

In a multi-camera environment, scene reconstruction is an important task with applications in multiple areas. It is not sufficient to only detect objects in each camera alone, but a combination of these views is necessary to:

- Overcome occlusion problems from some cameras.
- Fully utilize range and coverage of entire setup, not just single camera.
- Increase precision and stability of reconstructed scene.

1.1 Use case scenario

This project is part of a bigger software pipeline controlling a robot arm in a human-robot shared workspace. The goal is to merge data and reconstruct a scene from multiple camera streams, gathered from cameras mounted around the workspace and one mounted on the robot arm. This software will also propose a new camera pose, to which the robot will move its arm maximizing visual information detected from the scene, if there isn t enough relevant information already.

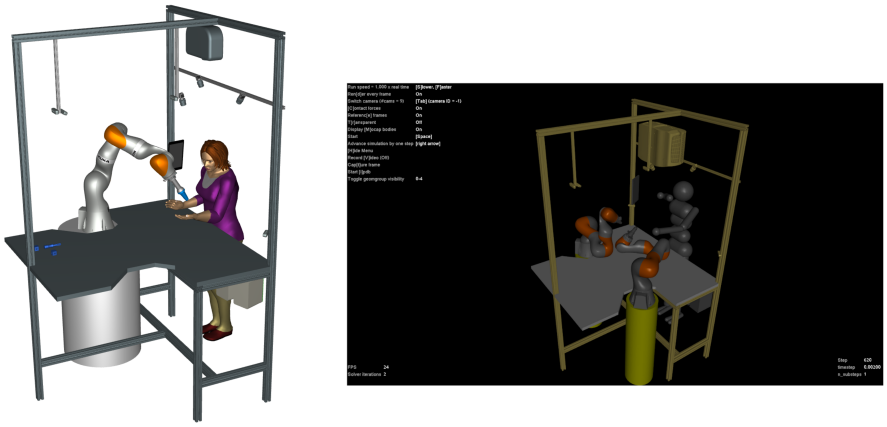


Figure 1.1: Schematic image of human-robot workstation.

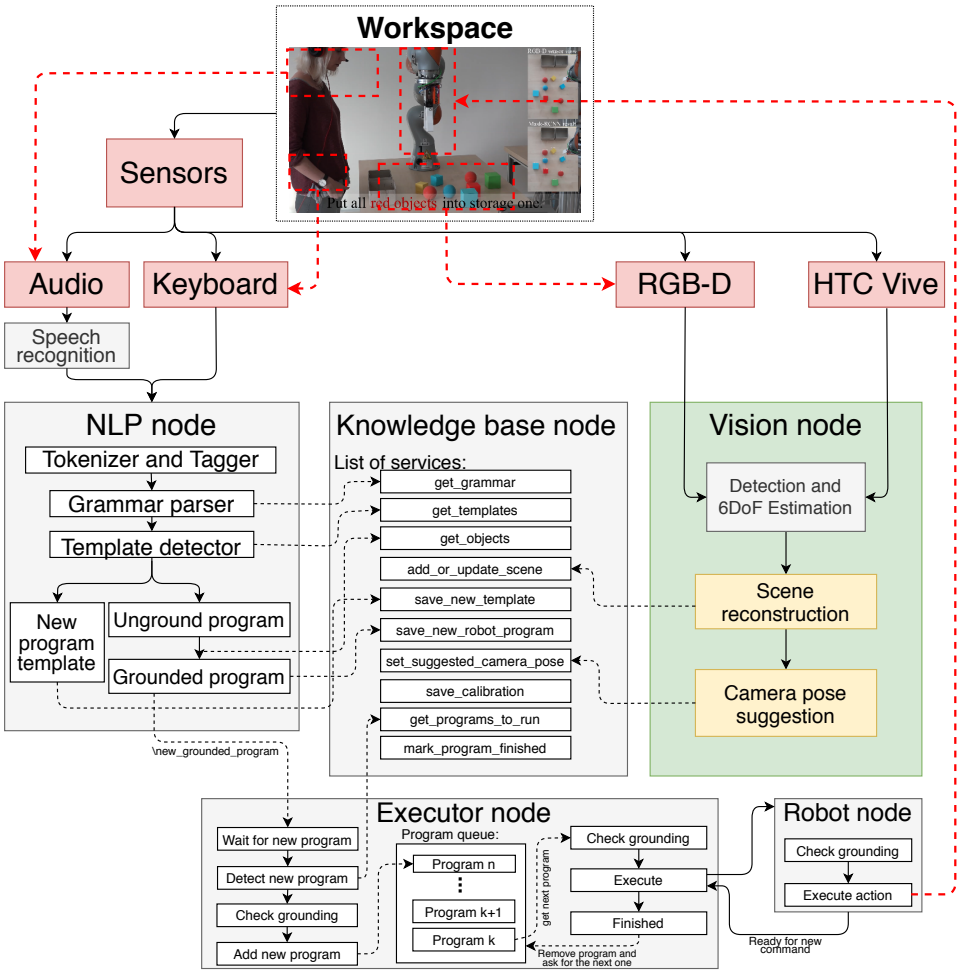


Figure 1.2: Schematic diagram of a bigger software pipeline. Vision node is shown in green, external modules are in gray. The focus of this thesis is implementing yellow modules (part of vision node). Red modules are sensor inputs to the entire system.

1.2 Description of the goal

The goal of this thesis is to develop software implementing VisionNode (as shown on [Fig: 1.2]), its modules and provide a framework for their communication. Modules which are not fully implemented yet, but are required for the operation of the vision node are replaced with a mockup version, until the time when the production modules are ready, at which point we will only swap module for module and all communication systems will already in place. Part of this software is also a visualization tool, screenshots from which are shown throughout this thesis.

Goal of scene reconstruction module [Sec: 6] and camera pose suggestion module [Sec: 7] is to merge multiple views of the scene to one reconstructed scene. Detect unknown areas (due to occlusion or other factors) and propose

new camera viewpoint such that would minimize information loss (unknown areas/ambiguous object detections or poses). The software takes already detected objects, or "features" (Described in [Sec: 5])) as its input from a detection module. Development of a module for detection in the color images and 6DoF pose estimation is outside of the scope of this work. At the time of writing the thesis, the detection module which would be used in production was not ready and therefore we could not connect to it and use it.

1.2.1 Evaluation and the results

This work proposes several different approaches (methods) for tackling both Scene reconstruction [Sec: 6] and Camera pose suggestion [Sec 7] tasks. These methods have been qualitatively evaluated and compared to each other, with some quantitative evaluation as well [Sec: 8, Sec: 9].

Chapter 2

Related work

Related work in this area can be split into multiple sections, none of which are trying to solve the same problem as this thesis does.

2.1 Object detection and pose estimation

There are multiple papers describing transformation using color images (and some in combination with depth images) to detect what type of object the camera is looking at, and its pose. These works use only one camera and are potential candidates for detection module (described in [Sec: 5.1.1]).

- **Segmentation-driven 6D Object Pose Estimation** [1].
  A segmentation-driven 6D pose estimation framework where each visible part of the objects contributes a local pose prediction in the form of 2D keypoint locations. From a set of 3D-to-2D correspondences, a reliable pose estimate can be obtained.

- **A Method for 6D Pose Estimation of Free-Form Rigid Objects Using Point Pair Features** [2].
  A feature-based method for 6D pose estimation of rigid objects based on the Point Pair Features voting approach. The presented solution combines a novel preprocessing step, which takes into consideration the discriminative value of surface information, with an improved matching method for Point Pair Features.

- **Deep Object Pose Estimation for Semantic Robotic Grasping of Household Objects** [3].
  One of the key challenges of synthetic data, to date, has been to bridge the so-called reality gap, so that networks trained on synthetic data operate correctly when exposed to real-world data. The authors of this paper explore the reality gap in the context of 6-DoF pose estimation of known objects from a single RGB image and show that for this problem the reality gap can be successfully spanned by a combination of domain randomized and photorealistic data.

- **Detection and Fine 3D Pose Estimation of Texture-less Objects in RGB-D Images** [4].

This paper proposes a practical method for the detection and accurate 3D localization of multiple texture-less and rigid objects depicted in RGB-D images.

2.2 Scene and object reconstruction

There are several papers, which propose techniques of scene (or object) reconstruction and representation. Most of these methods aim at reconstructing a scene or object which shape is not known. Whereas the problem solved is similar as multiple camera viewpoints have to be merged together to gather information, it i not the same as object detection and pose estimation, because in the latter case, the shape of the objects to detect is known. Our work explores techniques for reconstructing a scene from known 3D models and their estimated poses.

- **Scene reconstruction from multiple cameras** [5].
  The paper reviews some approaches to the problem of recovering a depth map from two or more images. It then describes a number of representations, including volumetric representations, layered plane-plus-parallax representations, and multiple depth maps.
- **Pixels, voxels, and views: A study of shape representations for single view 3D object shape prediction** [6].
  This study compares surface-based and volumetric 3D object shape representations, as well as viewer-centered and object-centered reference frames for single-view 3D shape prediction. The comparison is with respect to the neural net model prediction.
- **3D scene reconstruction from multiple uncalibrated views** [7].
  In this project, the authors explore different methods, even try some commercial applications to reconstruct a 3D scene from multiple uncalibrated views.
- **Real-Time 3D segmentation of cluttered scenes for Robot Grasping** [8].
  This paper presents a real-time algorithm that segments unstructured and highly cluttered scenes. Using depth information, the algorithm robustly separates objects of unknown shape in congested scenes of stacked and partially occluded objects.

These papers study different methods and techniques related to object *modeling* (prediction of the model shape), whereas this work focuses on merging information about known object models.

2.3 Reasoning

This group of papers talks about different reasoning or post-processing strategies with already obtained scene and reconstruction quality improvements.

- **3D-Based Reasoning with Blocks, Support, and Stability** [9].
  The authors propose a new approach for parsing RGB-D images using 3D block units for volumetric reasoning. The algorithm fits image segments with 3D blocks, and iteratively evaluates the scene based on block interaction properties.

  This has not been used in this work yet, but it could be integrated into the scene reconstruction module[6.4.3] in the future, for improved object detection and pose estimation quality, by reasoning about physical possibilities of reconstructed scenes. (E.g.: Are objects floating in the air? Are some on top of different ones, with enough support and stability to not tip over? If not, the detected scene is probably wrong, because in the real-life "source" objects are clearly not tipping over.)

- **Guidance of Robot Arms using Depth Data from RGB-D Camera** [10].
  Image Based Visual Servoing (IBVS) is a robotic control scheme based on vision. This scheme uses only the visual information obtained from a camera to guide a robot from any robot pose to a desired one.

  This paper has inspired the idea of Simulated Visual Servoing for camera pose suggestion method enhancement (see [Sec: 7.5]. One of the main differences is that we are not using a real camera, but rather a simulated view from a specific pose, which gets refined according to what is visible from that pose.

Chapter 3

Methods and algorithms

All the code used in this thesis is available at [Src: 1].

3.1 Voxelization

Voxelization is the process of taking general 3D meshes and turning them into an occupancy grid that covers the entire work area without overlap (see [Fig: 3.1], image from [11]).

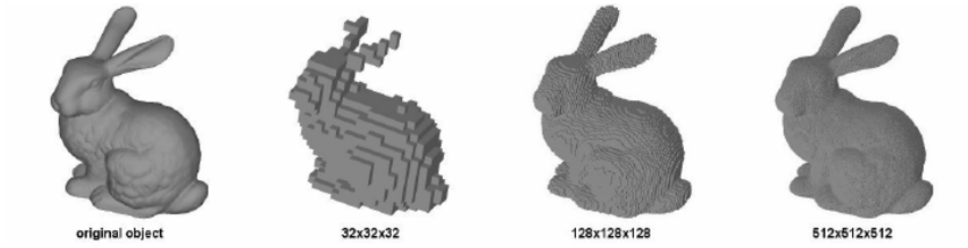


Figure 3.1: Stanford bunny voxelized at different resolutions. Left-most bunny is the original mesh. Grid resolution at the bottom. Image from [11]

For performance reasons, we have used octree data structure [Sec: 3.4] for storing and manipulating voxels.

3.2 PnP

PnP – "perspective-n-point problem" (PnP) is coined by Fischler and Bolles [12] for the problem of estimating the pose of a calibrated camera given a set of n $3D$ points in the world and their corresponding $2D$ projections in the image. The camera pose consists of 6 degrees-of-freedom (DoF).

The problem is to find rotation-translation matrix $[\mathbf{R}\,\vec{t}]$ in

$$\begin{bmatrix} x_c \\ y_c \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{R} & \vec{t} \end{bmatrix} \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix},$$

where x_c and y_c are coordinates of point projection into $2D$ camera reference frame, f_x, f_y, c_x, c_y are camera intrinsic parameters (focal length and principal point), and x_w, y_w and z_w are coordinates of a point in world reference frame. This formulation neglects lens distortion.

3.3 ArUco Markers

An ArUco marker [13, 14] is a type of fiducial square marker composed by a wide black border and an inner binary matrix that determines its identifier (ID). The black border facilitates its fast detection in the image and the binary codification allows its identification and the application of error detection and correction techniques.

Figure 3.2: An example of a 6x6 ArUco marker with ID=23.

3.4 Octree data structure

An octree [15] is a tree data structure in which each internal node has exactly eight children. Octrees are most often used to partition a three-dimensional space by recursively subdividing it into eight octants.

3.5 SVD

SVD – singular value decomposition is a factorization of matrix $\mathbf{M}$ such that $\mathbf{M} = \mathbf{USV}^T$, where $\mathbf{S}$ is a diagonal matrix with non-negative real numbers on diagonal. $\mathbf{U}$ and $\mathbf{V}^T$ are orthonormal matrices.

In this work, SVD is used for solving overdetermined homogeneous linear equations while minimizing the total least squares error. More detail is [Sec: 6.1].

Chapter 4

Experimental setup

The experimental environment consists of a board (roughly 50x35 cm) with ArUco markers on its edges and corners (shown in [Fig: 4.1]) and a handheld D435 Intel Realsense camera that is pointed at this board, such that enough markers are in its viewport. The camera was held in the hand for added flexibility when recording datasets for later use (multiple viewpoints and transitions between them. Easy setup of the desired viewpoint to test a specific phenomenon or edge case.) We can still detect the camera's extrinsic transformation relative to the board's coordinate system (Also referred to as "scene coordinate system" or "world coordinate system") using the ArUco markers. The only added benefit to having a fixed structure is the elimination of motion blur, but the amount of motion blur introduced by holding a camera was negligible and did not pose a problem for this work.

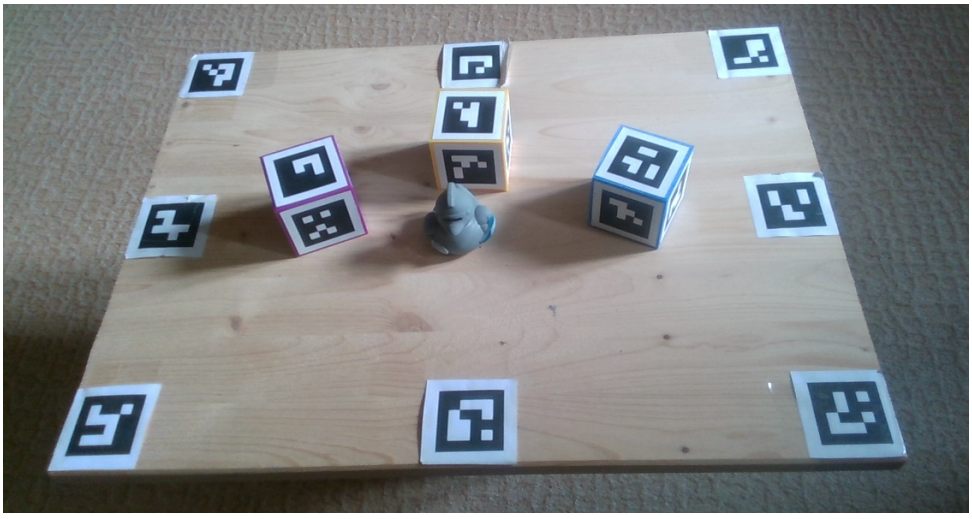


Figure 4.1: Photo of a board with ArUco markers and a simple scene.

Objects to be detected are placed on the board. We refer to the specific placement of these objects on the board as a "scene". One such scene consisting of three cubes and one rubber duck is shown in the [Fig: 4.1]. To simplify detection, objects are 5 cm cubes with a unique ArUco marker on each side. This is a simplified setup for the development. The board and handheld

camera will be replaced with actual robot-human workspace and multiple fixed cameras and one arm-mounted camera. In [Sec: 5.1], we will discuss that in spite of this simplification, we can still use the rest of this work without any major modification for real-world scenarios, without the need for any markers.

Robot arm-mounted camera creates several constraints for viewpoint suggestion, mainly the reach of the robot arm and reach with the desired orientation of the end effector. The other constraint is self-occlusion (the robot arm may hold the camera in such a way that it will partially occlude its view).

We will neglect this as the issue can be partially mitigated with camera placement on the end effector and selecting a good way of moving the robot arm to the desired pose. For the remainder of this work, let us assume that transformation between the end-effector pose and actual camera pose is small and therefore negligible, for purposes of suggesting optimal camera pose (and therefore viewpoint).

4.1 Datasets and collection

Dataset is a scene taken from one or multiple views, where color and depth information from the camera are recorded, as well as its intrinsic parameters (focal point, principal point, distortion coefficients)

Few selected datasets are shown in the following subsections. In the following subsections, there are images that show the color view from selected viewpoint of each dataset (on the left side) and a merged reconstructed scene, with cameras visualized at the original camera's extrinsic transformation.

4.1.1 Wall

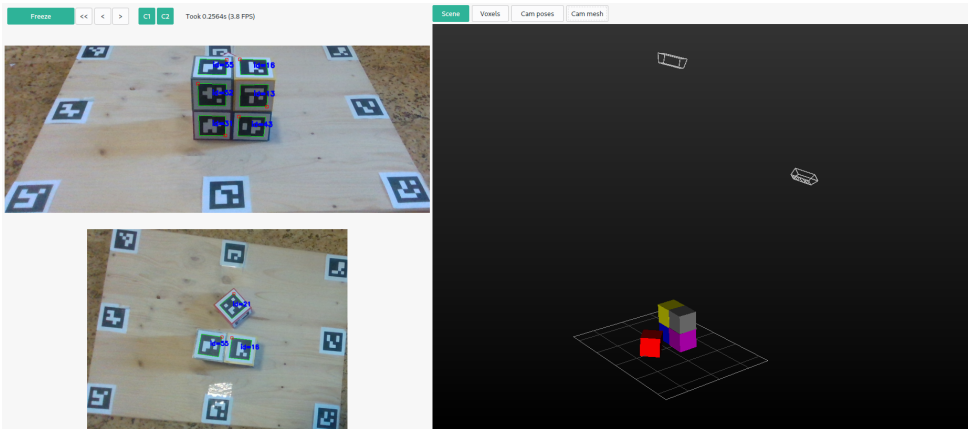


Figure 4.2: Wall dataset. Color camera images on the left, reconstructed scene including camera poses on the right.

The red cube is occluded from the view of the front camera and blue and magenta cubes are occluded from the top view. Both camera views are

therefore required for full reconstruction of this scene.

4.1.2 Corridor

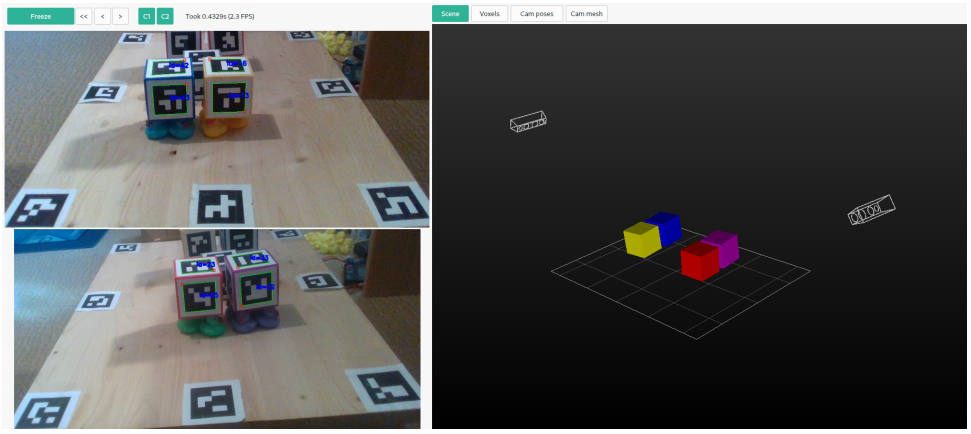


Figure 4.3: Corridor dataset, seen from the sides. Color camera images on the left, reconstructed scene including camera poses on the right.

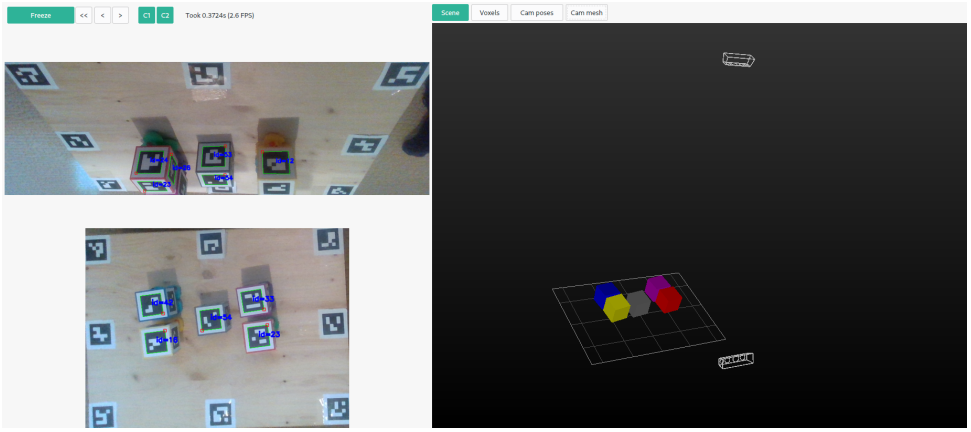


Figure 4.4: Corridor dataset, seen from above and front. Color camera images on the left, reconstructed scene including camera poses on the right.

A gray cube is hidden between two slightly elevated walls. Middle cube cannot be seen from either side view. Each side view can see only its respective wall. The front view does not show the entire wall. The only viewpoint from which everything can be seen at the same time is from the top.

4.1.3 ForeignObjects

View is obstructed by non-detectable objects. Non-detectable objects represent foreign occlusion, such as, but not limited to, human hands in the view, parts of the robotic arm, a random coffee mug and many others. Because of the occlusion, the reconstructed scene appears to be empty without any objects on the scene.

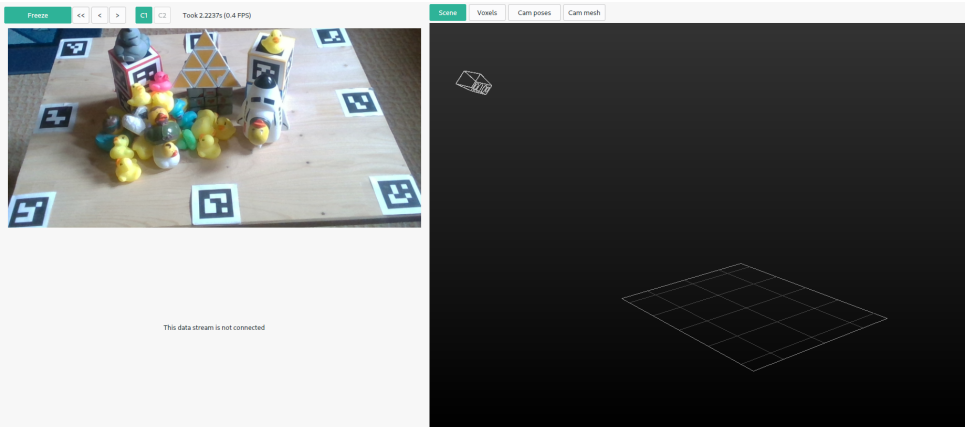


Figure 4.5: ForeignObjects dataset. Color camera images on the left, reconstructed scene including camera poses on the right.

4.1.4 TwoPillars

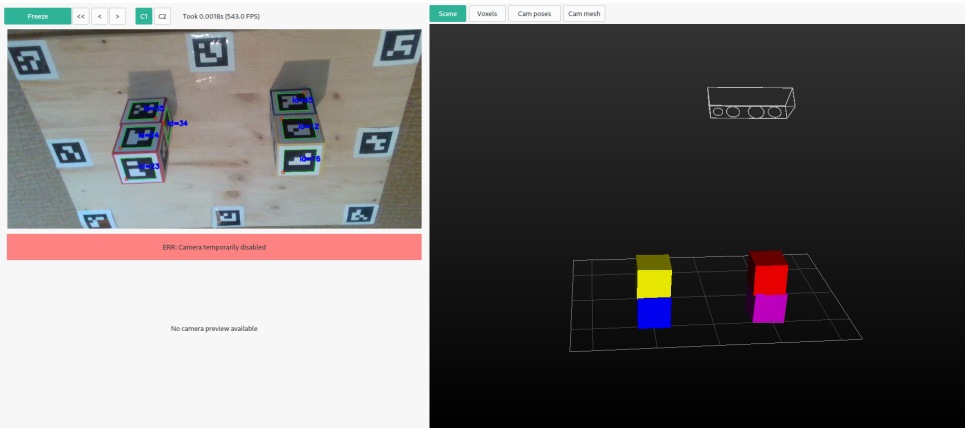


Figure 4.6: TwoPillars dataset. Color camera images on the left, reconstructed scene including camera poses on the right.

Scene casting two different non-intersecting shadows (again, from the front view).

4.1.5 Castle

Scene consisting of perfectly corner-to-corner aligned cubes taken from nine different camera viewpoints. This is later used for evaluation of scene reconstruction.

4.2 Implementation

The code is written in C++17 and some methods use GLSL to leverage the power of the GPU. It is implemented as a ROS node (Robot Operating

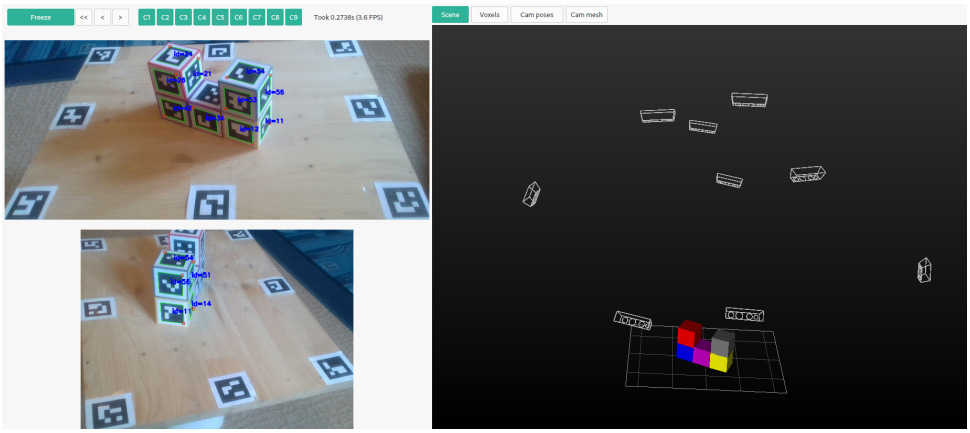


Figure 4.7: Castle dataset. Color camera images on the left, reconstructed scene including camera poses on the right.

System [16]). ROS is used for communication with other parts of the software pipeline, such as detection module or knowledge base.

Major used libraries are

- `librealsense` [17], library for communicating with D435 RGB-D camera and reading live image streams from it.
- `OpenCV` [18], library used for image processing of captured images and simple ArUco detection.
- `ROS` [16], used for communication with other nodes of the pipeline.
- `CGAL`, The Computational Geometry Algorithms Library [19], a library for mesh operations, such as boolean or decimation, (library used through `libigl` [20]).
- `VTK`, Visualization ToolKit [21], a library used by the visualization subsystem of the software.

Chapter 5

Architecture

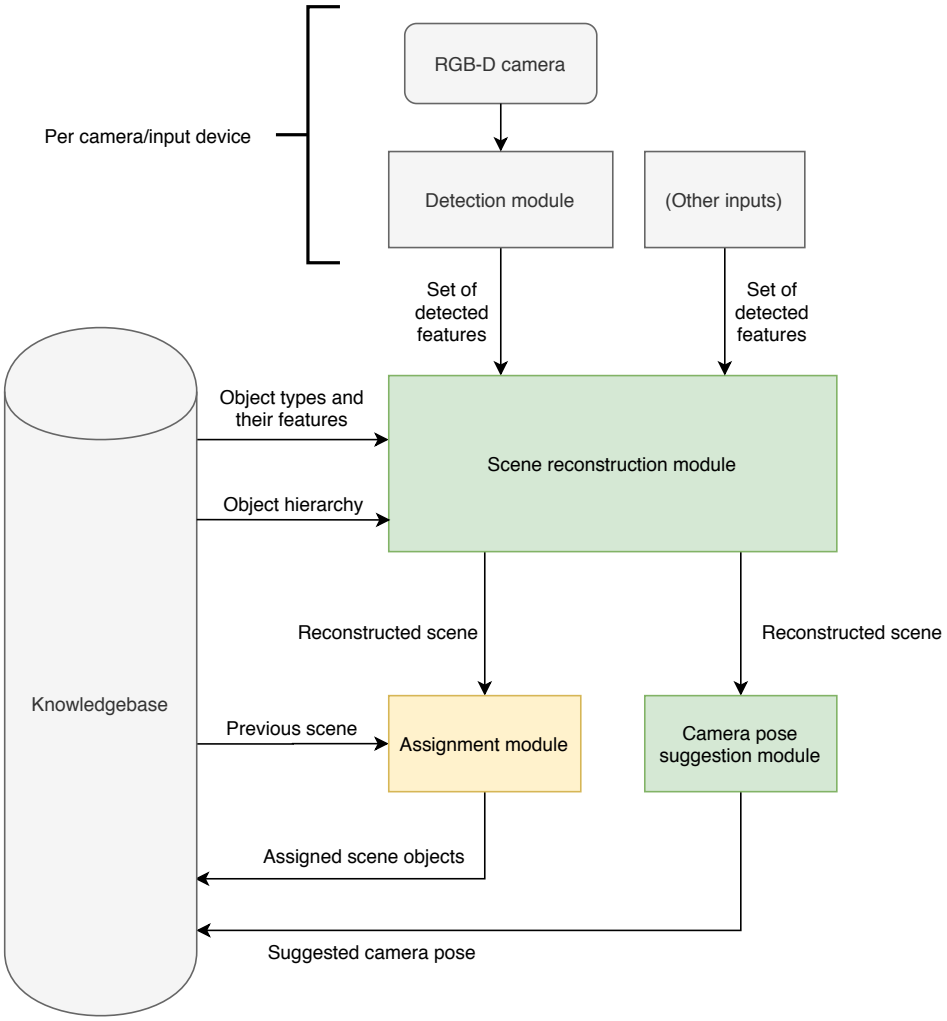


Figure 5.1: Diagram of modular architecture. Green modules are part of this work, Yellow module is future extension of this work and Gray modules are external, developed independently.

Vision node can be broken down into multiple modules (as shown in [Fig: 5.1]), which can interact with each other.

The modular design allows us to change modules between development and production environment, artificially manipulate information that goes in or out to find and isolate the strengths and weaknesses of the system or quickly test out multiple different methods to solve the same sub-task.

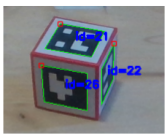
We intend to replace the detection module used in this work with a different one for production, which is currently being developed independently. This new detection module will be able to work on real-world objects, without relying on ArUco markers, either for pose estimation or camera extrinsic calibration. The module itself is outside of the scope of this project.

5.1 Modules

5.1.1 Detection module

The first action is to detect features and their 6DoF estimation, along with camera 6DoF. Currently, the detection module is a mockup that provides adequate data flow. The "feature" consists of an ID of the ArUco marker and marker's 6DoF. This feature is then paired with the camera extrinsic and is sent along the line to the next module (Scene reconstruction module). This module will be replaced by an independently developed one in the future, one which would not require ArUco markers.

This process happens for each camera (and potentially other pose tracking devices, such as HTC Vive) in each frame, effectively yielding multiple instances and potentially multiple types of detection modules.



Feature: [21]: 6DoF: q[0.72, -0.68, -0.13, -0.12] + [-0.02, 0.05, -0.01]
Feature: [26]: 6DoF: q[0.98, -0.03, -0.19, 0.01] + [-0.03, 0.03, 0.02]
Feature: [22]: 6DoF: q[0.61, -0.39, 0.39, -0.57] + [0.01, 0.03, 0.00]

Figure 5.2: Feature detection and pose estimation of ArUco cube by the mockup detection module. 6DoF represented by a quaternion and a translation vector.

5.1.2 Scene reconstruction module

The purpose of this module is to collect all the features from all the detection modules connected and output the final scene representation, which is a set of pairs [object type, 6DoF] with a confidence value. The module has to

combine pieces of information that might conflict with each other because of imperfect measurements of detection module and ambiguities in the scene. It has to then decide on the quality of each piece of information and how to use it. More information about the actual function and multiple methods used in [Sec: 6]

The scene reconstruction module communicates with a knowledge base, from where it downloads descriptions of the object types using all the feature types that the detection module might return. E.g., In human words, "There is an object type, let us call it `a lemon`, and it is yellow and round" or "There is a `red cube`, with an ArUco marker `13` on the `front side`. This red cube has other markers as well."

5.1.3 Assignment module

This module will keep track of instances of the same type of objects throughout the changing logical frames in the scene outputs from the scene reconstruction module. This information is essentially quantization of a statement "This particular instance of this type of an object has been moved from here to there, whereas all others stayed at their original locations." Implementation of the assignment module is subject to future work.

5.1.4 Camera pose suggestion module

The purpose of this module is to determine which parts of the reconstructed scene are unknown, and based on this information, suggest a new camera pose that will maximize collected information. We call the unknown areas "shadows". More details in [Sec: 7].

5.2 Knowledgebase

Knowledgebase (KB) provides information to different modules and expects a result (reconstructed scene, suggested camera pose, ...) to be stored in it. All software systems (e.g., robot arm motion optimization, NLP system, ...) share the same KB in a production environment. As such, every system queries it through a shared network connection, managed by ROS. KB itself is not part of this work as it is developed independently. However, because production KB is not fully implemented yet, we are using a mock KB [Src: 2], which implements common interface [Src: 3] that provides the rest of the modular architecture with adequate data flow and will be used for actual KB in the future.

Mock KB is initialized with data describing cube shapes, names an object hierarchy (e.g., "`Red cube` is also a `Cube`, but not every `Cube` is a `Red cube`") and what features (only of ArUco feature type) are associated with each object and in what relative 6DoF to each other (to a reference point of the object) [Src: 4].

The knowledgebase used in this project is an ontology (see [Fig: 5.3]).

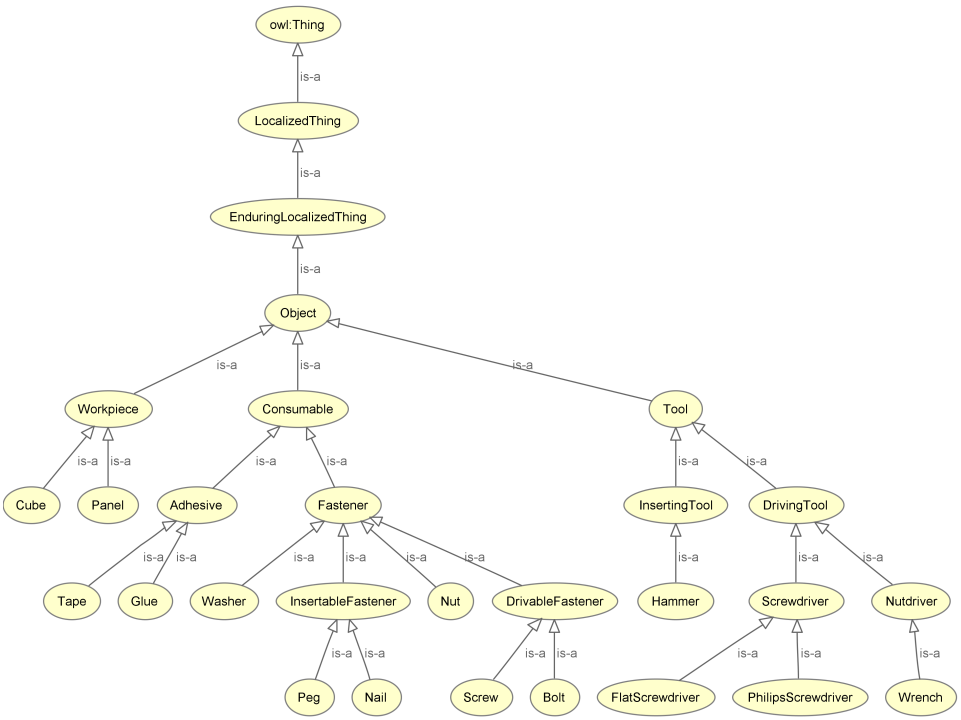


Figure 5.3: Diagram of selected part of the ontology KB structure.

Chapter 6

Scene reconstruction methods

Scene reconstruction outputs a hypothesis about objects and their poses. There can be multiple ambiguous "explanations of observations", for example a front view of a cube or a long box with the same color. In such a case, we output a "superposition" of both.

In the following sections, we will describe multiple scene reconstruction methods. The only implemented method is the first one SVD 6DoF estimation method [Sec: 6.1] which has serious limitations and is not suitable for production environment. The other two are at the conceptual level only and we would like to implement and explore the methods in the future.

There was not enough time to explore all of these methods, as the camera pose suggestion system is more important in the current context. Sub-optimal results of scene reconstruction module do not affect the quality of camera pose suggestion module, because it can just assume that its input is correct and when reconstruction module gets exchanged for a different one, the camera pose suggestion will only work on slightly different data (a cube is at a different position than it was before in the same dataset) which is essentially invisible change from the perspective of the camera pose suggestion module.

6.1 6DoF estimation method using SVD

We can use SVD method for estimating 6DoF of objects from feature's representative set of 3D points $\vec{x}_F$ (in the world coordinate system) from the detection module (in this case, four corners of each ArUco marker, in world space) and finds affine transformation T between corners of each marker belonging to given object and 3D positions of respective points $\vec{x}_S$ in the object's coordinate frame (equal to world coordinate frame, if the object is at origin, with zero rotation). See [Fig: 6.1]. Matrix $[\mathbf{A}\,\vec{t}]$ of this transformation T, describing the projection between points $\vec{x}_F = T(\vec{x}_S) = [\mathbf{A}\,\vec{t}]\vec{x}_S$ is found using SVD [Sec: 3.5] method:

Find centroid of each set of points:

$$\vec{c}_F = \frac{1}{\|\vec{x}_F\|} \sum_i \vec{x}_{F_i},$$

$$\vec{c}_S = \frac{1}{\|\vec{x}_S\|} \sum_i \vec{x}_{S_i},$$

then construct 3x3 covariance matrix H:

$$\mathbf{H} = \sum_i (\vec{x}_{S_i} - \vec{c}_S) \cdot (\vec{x}_{F_i} - \vec{c}_F)^T.$$

From SVD decomposition:

$$\mathbf{H} = \mathbf{USV}^T.$$

Matrix $\mathbf{S}$ is a diagonal matrix and matrix $\mathbf{A} = \mathbf{UV}^T$ is a rotation-only transformation matrix, representing optimal rotation of set of points $\vec{x}_S$ such that errors between transformed points and $\vec{x}_F$ are minimal.

As a last step, we have to construct adequate translation $\vec{t}$, because so far we have been treating all points as linear, not affine.

$$\vec{t} = -\mathbf{A} \cdot \vec{c}_S + \vec{c}_F.$$

We now know the matrix $[\mathbf{A}\,\vec{t}]$ which is our desired transformation T.

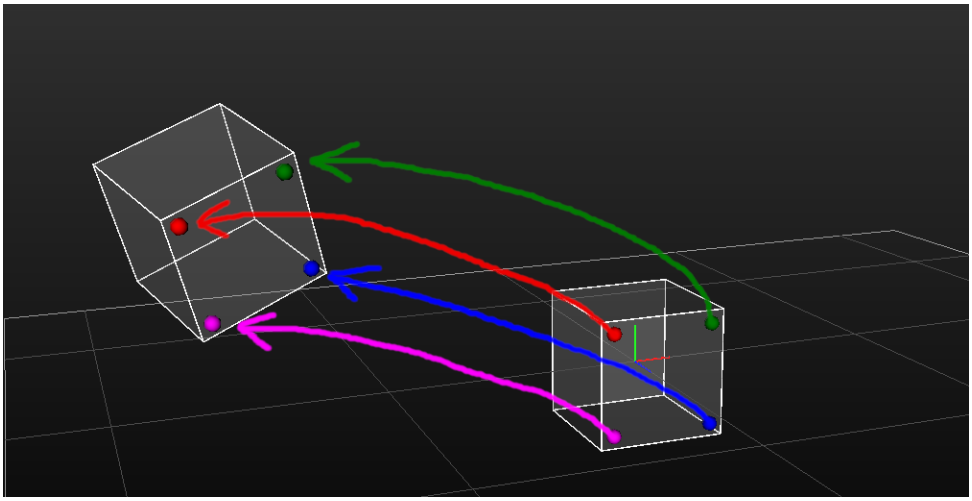


Figure 6.1: An image showing transformation from object (cube, on the right) at world origin (3D cross) to new object's pose (cube, on the left), using detected feature points $\vec{x}_F$ (colored dots near corners, on the left cube), matched with their source positions $\vec{x}_S$ (dots of the same color, on the right cube). Transformation of these points defines transformation T of the entire object.

This is a very simple reconstruction method, which works under two strong assumptions:

1. There is at most one instance of an object type visible in the scene at any given time. Current implementation of this method makes a list of all 3D points belonging to one object type and uses that to find its pose. Therefore it does not differentiate between different instances or occurences of objects (of the same type) and in the case of multiple

instances per object type, it tries to find a transformation that best fits all the points, yielding object pose appearing in the centroid of all instances of that object type, with "averaged" rotation.

2. Every feature value references only one object type. This is somewhat related to the previous assumption. Once feature value (let us say, a marker with the same ID) is assigned to an object type (its corner points are added to a list of vectors for SVD) it cannot be removed or ignored anymore as it is treated as a support evidence for that particular object type. If there were multiple objects (cubes) in the knowledgebase with same marker IDs (on at least one side), a detected marker would be assigned to each cube and therefore multiple cubes would get reconstructed at the same time.

Assumption `1` will not be satisfied in the production environment. There will be many instances of the same object type on the scene (for example a box full of screws, each screw in the box being an instance of the object type `screw`)

Assumption `2` may or may not be satisfied in the production environment, depending on the detection module used. There might be however multiple detection modules, outputting multiple feature types for the same scene (for example one detecting shape of an object, other detecting a color of an object). This assumption would have to be satisfied for all feature types, which will very likely not be the case.

6.2 Voting method

Voting method gets its name from the fact that it combines pieces of information that might conflict with each other (because of imperfect measurements of detection module and ambiguities in the scene) using a voting scheme. Every feature casts a "vote" in a 7-dimensional table (6 for 6DoF and 1 for object type) in a respective cell for quantized values of the object's 6DoF and object's type. Features also cast negative votes on nearby 6DoF for objects of a different type. The negative votes are the case only for object types which do not share features with same (or similar) values (In human words "If I see `an object with red color` at this pose, I will vote positively for each object type that is characterized by feature `an object with red color` around that pose. And negatively for each that is not.").

After each vote is counted, this table represents (if normalized such that it sums to one) a distribution of probabilities that at given 6DoF there is an instance of an object of a given type (see [Fig 6.2]). We can extract local maximal values from this table and save object instances to the reconstructed scene at their respective poses.

This method does not require either of the assumptions mentioned in the previous method to hold, making it a viable choice for the production environment. One of several advantages is that it can easily combine many

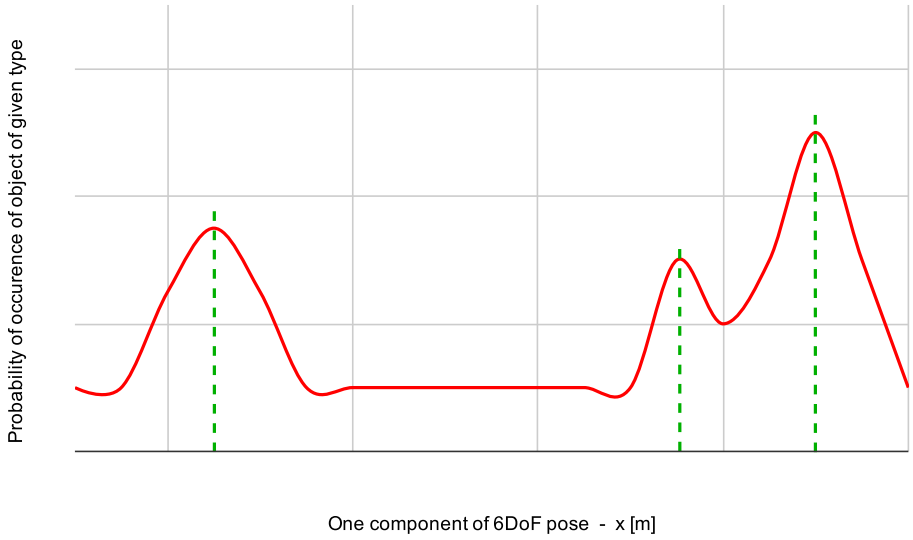


Figure 6.2: Graph showing probability of occurrence of an object instance of given type (vertical axis) based on different 6DoFs (one component of 6 6DoF axes projected onto a horizontal axis). There are three object instances present, marked with green dashed lines at their respective 6DoFs.

feature streams of different feature types from multiple sources at the same time (for example different detection methods from visual streams of cameras, different detection methods from a depth only camera stream, HTC Vive trackers, etc.).

A major disadvantage is the voting process, which requires discretization of all 6DoF values (6DoF are continuous variables, but table has only discrete cells) and therefore the table has to be very large (have many small cells), in all 6 dimensions of 6DoF, to achieve required accuracy. On the other hand, the smaller the cells are, the more votes get "dilluted" among the neighbouring cells, effectively hiding the local peaks. There are techniques to improve quantization artefacts to be explored, including casting votes to neighboring cells to mitigate the negative effect of values which get quantized to near border of a cell (as described in [2]).

This method has not been implemented yet. It is only a proposal which is subject to further work.

6.3 Multiview solvePnP

PnP (Perspective n points) [Sec: 3.2] is a technique normally used for reconstructing 3D points from their projection to 2D plane (e.g., a camera plane), in such a way that reprojection error is minimized. Reprojection error is an image distance between a point projected from the reconstructed scene and a point in the measured camera image. This leads to different error numbers when 3D points are moved in different directions relative to the

camera (moving points further away or closer to the camera will have less effect than moving it side to side).

We can extend this process to the multicamera environment and use this interesting property, where the directions with small sensitivity (depth direction in camera reference frame) will be very different for each camera (assuming very different positions or orientations between different camera views) in the world coordinate system. When we reconstruct 3D points in a way that minimizes global reprojection error (sum of reprojection errors from all cameras), we can achieve precise reconstruction, because the method intrinsically cherry picks parts of information from each camera that are relevant with respect to the camera pose (e.g., it is better to set a point's left-right position from the front facing camera, but not its depth. Use a different (for example top facing) camera for that). More views of the same object instance should improve its reconstructed pose precision dramatically.

This approach creates a challenge of what to do if you do not see corresponding points in all of the camera views. It is essential to not to reason about each point separately, but rather with constraints between them ("this is a corner of a cube, this is a different corner of the same cube, therefore they have to be placed at a given distance from each other"). If one of them moves in order to minimize reprojection error in first camera's view, the second one has to move with it, even if it increases reprojection error in the second camera's view).

Related problem to this, is a detection of points that belong to a different instance of the same object type, even if the objects are relatively close to each other. This could be addressed with multipass/iterative approach:

1. Say that all points of given type belong to the same object instance.
2. Run PnP.
3. Cut away points from this instance which introduce large reprojection error (split the object instance to two or more).
4. Repeat PnP on each.
5. If some reprojection errors are outside of threshold, split again and repeat from step `3`.
6. Otherwise stop.

This is similar to K-means problem, where we are also solving which data point belongs to which "center" and where that center is. The difference is that with K-means, there is a given number of centers and we assign data points to the one with closest distance, whereas in our method we do not know the number of centers (object instances) beforehand and have to "split" (or possibly "merge") said "centers". We do however have a way of deciding that the result is sufficient – based on threshold of global reprojection error.

The disadvantage of the multiview solvePnP is that it requires "features" with specific information and cannot deal with any arbitrary feature type.

The features have to contain information where specific known points of the objects are being projected to (in camera plane coordinates), and information about the camera itself (its extrinsic transformation, intrinsic calibration and distortion coefficients). All of this information should be available in the production environment though, so this does not pose a major problem.

This method has not been implemented yet. It is only a proposal which is subject to further work.

6.4 Possible extensions and future work

6.4.1 Temporal filtering

One possible improvement is using short-term memory of last several frames of scenes and average over them (or use more advanced statistical method) for filtering to remove small movements caused by instability or near-instability of current observation of a scene or confusion between two similar types of object for one object instance. This would make the scene more stable.

Possible ways of implementing this feature are either:

- Adding features from older frames to current frames with smaller and smaller weights, based on the age of the said features and evaluate with the rest of the data, without any modification to the method itself. A different implementation strategy might be native support for short term temporal filtering in the method itself.
- Native support for short term temporal filtering in the method itself.

6.4.2 Reasoning about empty spaces

Shadow information from the camera's depth sensor can be used not only for occlusion and visibility detection, but also to remove some of the "superposition candidates" that cannot be correct, based on the shadow information. At places where there is no `SOLID_or_SHADOW` space detected from cameras, we have knowledge that the space is empty and therefore cannot be occupied by any object or part of it. In human words, we are filtering areas where the proposed object candidate would not be able to fit. That is, using information about what we do not see, to refine knowledge acquired from information about what we do see.

6.4.3 Reasoning about support and stability

Paper [9] proposes using an information about support and stability interactions between different objects on the scene and refining reconstruction in such a way that it will output stable scenes (e.g., without floating objects or objects that would fall instantly, but they do not in the real world, therefore it means that the detection and pose estimation must have been wrong). Their approach could be adapted to our system.

6.4.4 Propose disambiguation camera views

Some reconstructions might be ambiguous or very hard for the system, from current camera views. An example would be 2 instances of thin (in the direction of camera view) object type, which may be accidentally merged into one, because of failing depth perception. Imagine a camera looking at the front side of a phone. It knows that it is a phone and where it is in the image (left-right, top-bottom). There is one more camera looking from the back side of the phone. It also knows where a phone is in the image (left-right, top-down). Neither camera has a very good depth estimate. It is reasonable to assume that both of them are looking at the same phone (one instance of a phone). If there were two phones, positioned such that the front side of one would be glued to (or just aligned with) the back side of the other, the second phone is virtually invisible, unless we have a side view available.

It would be nice, if a method used would be able to detect situation such as this and pass information to camera pose suggestion module that it wants a view from a particular side or angle to enhance reconstruction.

Chapter 7

Camera pose optimization methods

In this section, we will describe four different camera pose suggestion methods, each with different strengths and weaknesses and an enhancement which can be applied to each of mentioned methods. The first method is only a concept, all the other ones have been implemented and tested.

7.1 Light sources method

Light sources method is a first concept which never got implemented at the end.

The idea was to construct a 3D scene (from voxels or some other representation), add point lights to the scene, at the positions of the original cameras, render it and areas which are in shadow were not visible. Add a next light source (camera in real life) such that the new shadow area will be minimized. This method would not work, because a correctly detected object would still be in shadow, from a different side. E.g.: Top-down view of a single cube would yield 2 more necessary camera views, one centering north east edge of the cube and other one south west edge (or 90 deg rotated equivalent). It also suffers from an inability to handle scenes with foreign or not detected objects and would-be shadows cast by them. All such shadows and occlusions are treated as if the respective parts of the scene were known, but without any `SOLID` objects.

7.2 Sum of voxel volumes method

Sum of voxel volumes method (In the thesis referenced as "Voxel method" or just "VOX" for short) works by scoring each camera view by a "voxel score" or "vox score" and then selecting a view with the best score (highest number).

The process could be broken down into several steps:

1. Generate candidate camera poses to test.
2. Voxelize the scene. We are using Octree [Sec: 3.4] as a voxel structure, for performance reasons.

3. Find shadow areas (voxels) which are not visible from any camera.
4. AntiAlias shadow voxels, to mitigate some discrete world artefacts.
5. Score each of candidate camera pose. Score is volume of area that was marked as `SHADOW` in step 2, but is not marked as `SHADOW` anymore, with the new (tested) camera pose.

7.2.1 Generate candidate camera poses

Camera pose suggestion works by scoring candidate poses using some metric and selecting the best one. For that, we need to have good starting candidate poses.

Currently the software generates [Src: 5] two half-spheres each with different radius and uniform regular distribution of positions on these half-spheres [22] (see [Fig: 7.1]). Each position is coupled with a camera rotation (to create 6DoF pose) that looks to the center of the work area. The smallest sphere also generates poses where camera direction is different, to have more poses inside of the scene, to add the ability to look through or under structures in the scene.

Camera poses are generated with the reach of the robot arm in mind, such that they are not too far from the scene, where the robot arm could not reach and use the pose for camera viewpoint. We can generate poses at different distances from the center, in different configurations or amount, if we need to for that particular robot arm setup.

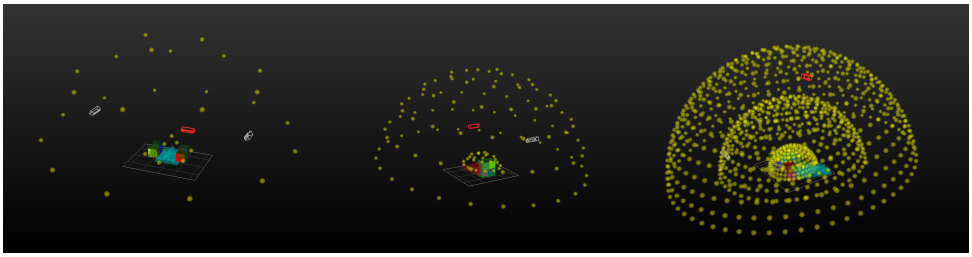


Figure 7.1: Example sets of initial camera poses. Two half spheres with low point density (on the left), Two half spheres with medium density (in the middle) and 3 half spheres with high point density (on the right).

7.2.2 Scene voxelization

Scene voxelization [Sec: 3.1] is a process of creating a grid of non-intersecting cubes (or "voxels"), the union of which covers the entire work area based on 3D models of objects in the reconstructed scene. Each voxel gets a type assigned in this process, either `EMPTY` or `SOLID`.

This grid is in fact Octree data structure [Sec: 3.4]. This is an implementation decision for performance and memory reasons. Base voxels (voxels in the first octree level) (in current implementation have the size (length of voxel's edge; voxels are cubes – all edges have the same length) set to

75 mm) can be split into eight smaller voxels, each of exactly half the size. In the current implementation there is a level limit of maximum 3 subdivisions (which means four sizes, one base and three splits – as shown in [Fig: 7.2]). This datastructure achieves high voxel detail where needed, while saving memory and processing power at areas which are the same and do not need that level of detail (for example: empty spaces far from the reconstructed objects)

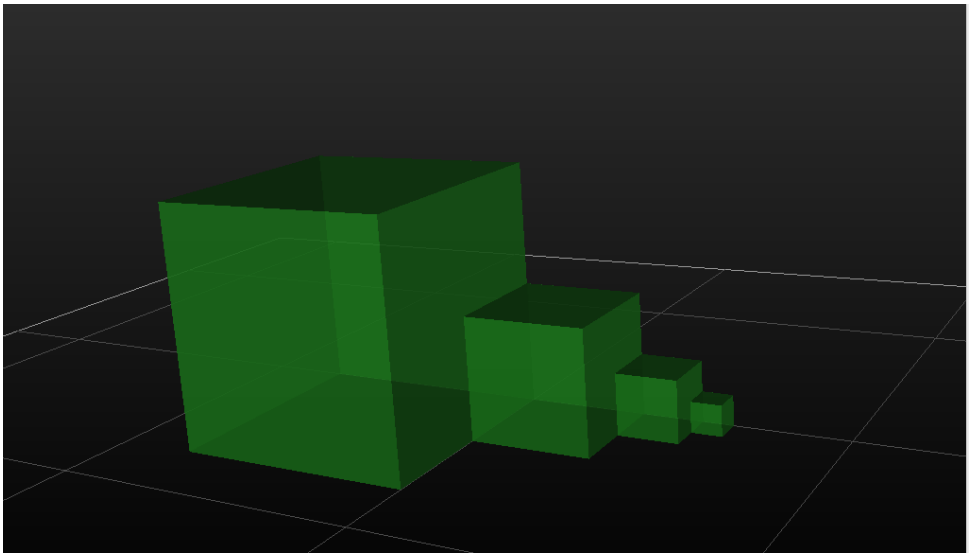


Figure 7.2: Image of voxels of different sizes, showing variability of level of detail.

7.2.3 Find shadow voxels

`SHADOW` is another voxel type that represents a volume in space which is not solid (no reconstructed object was found at that location in the scene), but is not directly visible by any of available cameras. Meaning that in the real world, there can be an object or empty space, we have no way of knowing just yet.

Process of finding shadow voxels (in codebase, function named `processVisibility` [Src: 6]) first sets every `EMPTY` voxel as `SHADOW` and then iterates over every reconstructed camera pose and over every `SHADOW` voxel and either sets it to `EMPTY` if this voxel is visible from the current camera or leaves it as a `SHADOW`. Voxel is considered visible if and only if there is no `SOLID` voxel in the ray from tested voxel's center to the camera position and at the same time, the angle at which the ray enters the camera is within its field of view. Voxels can also be subdivided into eight more voxels if it is required for more detail and maximum subdivision level has not been reached yet. Camera visibility is evaluated from the center position of voxel and positions near its corners. If these do not return the same result, voxel is subdivided and each part is processed individually. Empirical results showed that this subdivision check is sufficient, even though in theory there might be (rare) cases where

subdivision is required but not triggered.

Pseudocode:

```
// Actual code: VoxelConstruction.cpp:isPointVisible
function isPointVisible(Vec3 pos, CamPose cam) {
    Ray r = Ray(origin: pos, direction: cam.pos - pos);

    if (cam.isRayOutsideFov(r)) return false;

    foreach (Voxel in the octree : vox) {
        if (vox.type == SOLID && r.intersects(vox)) {
            return false;
        }
    }
    return true;
}

// Actual code: VoxelConstruction.cpp:processVisibility
function processVisibility(Voxel vox, CamPose cam) {
    bool isVisible = isPointVisible(vox.center, cam);

    if (!vox.levelOfSubdivisionsReached) {
        foreach (near-corner position : p) {
            if (isPointVisible(p, cam) != isVisible) {
                vox.split();
                foreach (vox.children : child)
                    processVisibility(child, cam);

                return;
            }
        }
    }

    if (isVisible)
        vox.type = EMPTY;

}

// Actual code: VoxelConstruction.cpp:calculateVisibility
function calculateVisibility() {
    foreach (Voxel in the octree : vox) {
        if (vox.type == EMPTY)
            vox.type = SHADOW;
    }
    foreach (Reconstructed camera pose : cam) {
        foreach (Voxel in the octree : vox) {
            processVisibility(vox, cam);
        }
    }
}
```

After running `calculateVisibility()` correct voxels will end up being marked as `SHADOW` in the octree, voxels will be split if needed.

7.2.4 AntiAlias shadow voxels

Voxelization and subsequent operations in voxelized world have inevitable discretization artefacts. One of them is occurence of "thin shadow layers" on objects not aligned with scene axes. (see [Fig: 7.3]). This behaviour is caused by a "stair effect" – each step (voxel) is casting a small shadow and the center of the next voxel is in it, resulting in being marked as `SHADOW` voxel type.

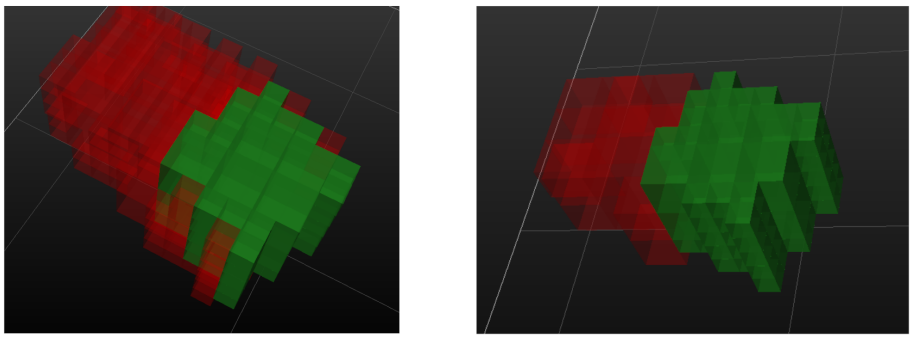


Figure 7.3: Image of axis misaligned cube and thin layer of `SHADOW` on its side (on the left) and the same scene, same camera positions with anti-aliasing applied.

This is very undesirable behaviour as it incentifies suggested camera pose to look at this thin layer even though it may be an area well known. This wrong incentive is even more manifested in the subsequent method [Sec: 7.3] which takes into account the visible surface area of shadow, not its full volume, as this method does.

To counteract this issue, we filter detected `SHADOW` voxels and remove some of them (mark them as `EMPTY`) if they do not satisfy our filter criterion.

The criterion is for each smallest allowed `SHADOW` voxel v to have a score s_v at least number thr or more of smallest (or equivalent number of larger) voxels touching it, summed across all cardinal directions. Number thr set to 5 in current implementation.

Score s for voxel v is determined as:

$$s_v = \sum_{\vec{dv} \in \{(\pm sz,0,0),(0,\pm sz,0),(0,0,\pm sz)\}} [\![\text{voxelAt}(\vec{v_{pos}} + \vec{dv}) == \text{SHADOW}]\!].$$

Function voxelAt(p) returns voxel v which contains given position p. $\vec{v_{pos}}$ is a position vector of voxel v and sz is a size of the smallest voxel, making expression voxelAt($\vec{v_{pos}} + \vec{dv}$) return a voxel which is just next to given voxel v, in every cardinal direction direction.

7.2.5 Score each camera pose

Score s (or quality) of each camera pose is calculated as

$$s = \sum_{i \in E} \text{vol}(v_i),$$

where $\text{vol}(v_i)$ is a volume of voxel v_i, E is a set of voxels which changed state from `SHADOW` to visible (`EMPTY`) for the new camera pose, when step `3 Find shadow voxels` was applied.

Current implementation does not actually mark these voxels as `EMPTY` and does not even subdivide them. It only calculates volume if they are already marked as `SHADOW`. This calculation is done recursively in place where step `3 Find shadow voxels` would split the voxels. Because we are avoiding splitting (and therefore modification of the grid in octree), we can run this operation in parallel, testing multiple camera poses at the same time, with shared voxel grid datastructure (no copies are required).

After all scores are computed, we select the pose with the best score (highest number) as the "new suggested camera pose" to instruct a robot arm with camera to look from that viewpoint.

For visualisation purposes, step `3 Find shadow voxels` is ran again on the selected pose and it marks all originally `SHADOW` voxel that would have been marked as `EMPTY` because of this suggested camera pose as a type `SHADOW_EMPTY`, which has a different color (cyan, instead of red) in the visualisation, as shown in [Fig: 7.4].

Discussion about the method

The Sum of voxels method described in this section has the following properties:

- **Speed:** Rather slow because the evaluation of shadows is asymptotically $O(|C| * |V| * |V|)$ where C is a set of cameras on the scene and V is set of all voxels in grid octree. V is rather large and slow to traverse because of random access through non-continuous memory block. It is still faster than one continuous memory block filled with many more smallest-sized voxels. Because of this random access and numerous branching in the evaluation algorithm, this method is not suitable to run on massively SIMD compute devices (such as GPUs). It can however be parallelized when scoring camera poses, because the algorithm does not need to write anything into a shared structure (or deep-copy it before each camera pose scoring).

- **Quality of estimate:** Good results (see [Sec: 9]) because it takes into account entire volume of shadows, not just visible surface area of shadow, which makes it more robust to voxelization artefacts (as described in [Sec: 7.2.4]) and helps with reasoning about "importance" of a shadow: "This big shadow mountain behind a wall of objects is probably more relevant than a long thin line of shadow near the edge of this wall or underneath a spherical object"

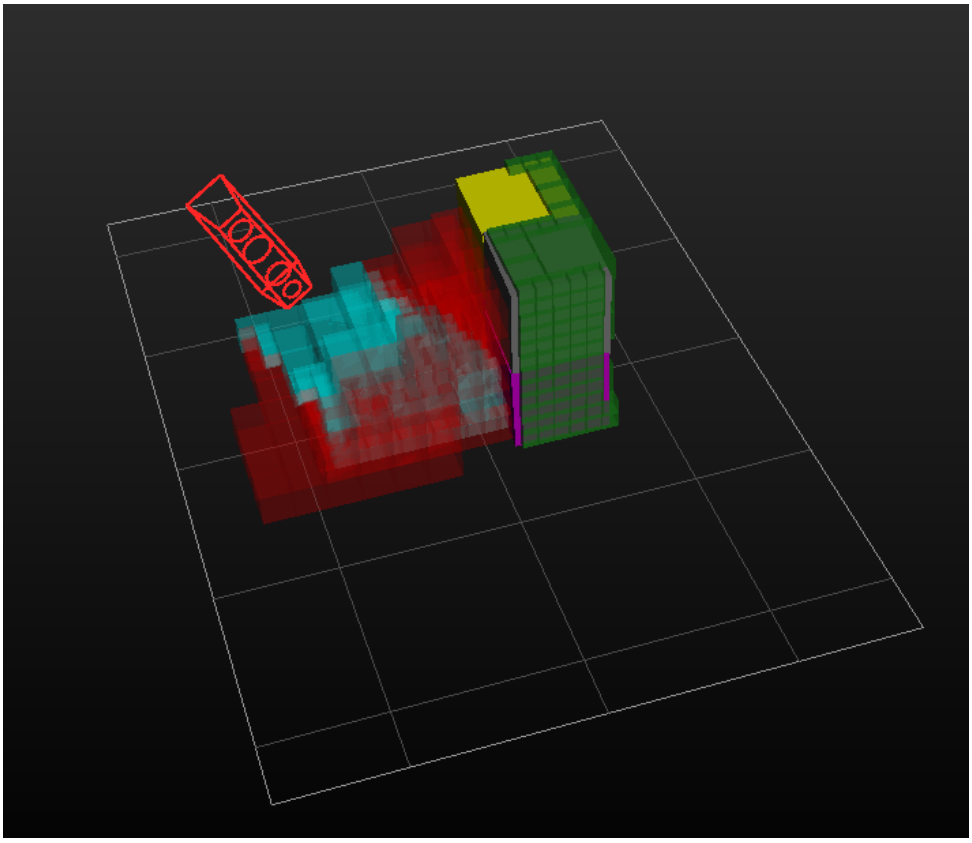


Figure 7.4: Visualisations of voxelized scene (green voxels), with cast shadow (red voxels) and "de-shadowing" by suggested camera pose (red camera, cyan voxels).

- **Artefacts:** Distance from the expected location of a possible object on the scene (inside a shadow) is not reflected in the score of the viewpoint in any way. Therefore camera parameters such as focal length (and therefore effective range) are ignored and the method does not optimize for good detail and coverage of detection of new hidden objects. Empirical results show that suggested camera pose is sometimes way too close (and therefore unusable).

- **Dealing with foreign occlusions:** Does not handle scenes with foreign (not detected) objects and would-be shadows cast by them. All such shadows and occlusions are treated as if the respective parts of the scene were known, but without any `SOLID` objects. For real-world use cases, this will very likely prove problematic and insufficient.

7.3 Render and count shadow pixels method

Render and count shadow pixels method also works by assigning a score value to each proposed camera pose and then selecting a pose/view with the best

score (highest number). Computation of score is described in [Sec: 7.3.6].

The process could be broken down into several steps. The first four are shared with the Voxel method (described in [Sec: 7.2]):

1. Generate candidate camera poses to test.
   Shared with Voxel method. For details see [Sec: 7.2.1].
2. Voxelize the scene. Using Octree [Sec: 3.4] like voxel structure, for performance reasons.
   Shared with Voxel method. For details see [Sec: 7.2.2].
3. Find shadow areas (voxels) which are not visible from any camera.
   Shared with Voxel method. For details see [Sec: 7.2.3].
4. AntiAlias shadow voxels, to mitigate some discrete world artefacts.
   Shared with Voxel method. For details see [Sec: 7.2.4].
5. Render voxelized scene with 100% ambient light, no shadows, no transparency. Shadow voxels are represent by red color.
6. Count shadow pixels we see in a given frame (render FOV matches real camera's FOV).

7.3.5 Render voxelized scene

Software renders voxelized scene from viewpoint of each candidate camera pose. The rendering is done in memory only, not on the screen. It does not need to create an extra window (there is still the ability to create such a window for debugging purposes).

Scene gets rendered with ambient light only (there are no shades or light "intensities" caused by directional or point light bouncing from source to camera at varying angles). `SHADOW` voxels are rendered as red cubes, all other objects have zero red channel (they are not red, or even mixed from anything that contains any amount of red color). Only other rendered objects are `SOLID` voxels and a box aligned with ground plane to serve as occlusion for any camera pose which might want to look up, underneath a scene (even partially) as this camera pose is not legal in a real-world scenario.

Render camera has focal length and FOV parameters set to copy real camera parameters.

Rendered scene consists of two parts: color buffer and depth buffer (visualized in [Fig: 7.5]. We will need both later.

7.3.6 Count shadow pixels

Score of given camera pose is determined by a number of shadow (red) pixels visible in the color buffer obtained in the previous step. This, however, leads to choosing a pose "inside" or very close to a shadow voxel, because the closer the camera is, the "bigger" red voxels are (the entire view is red)

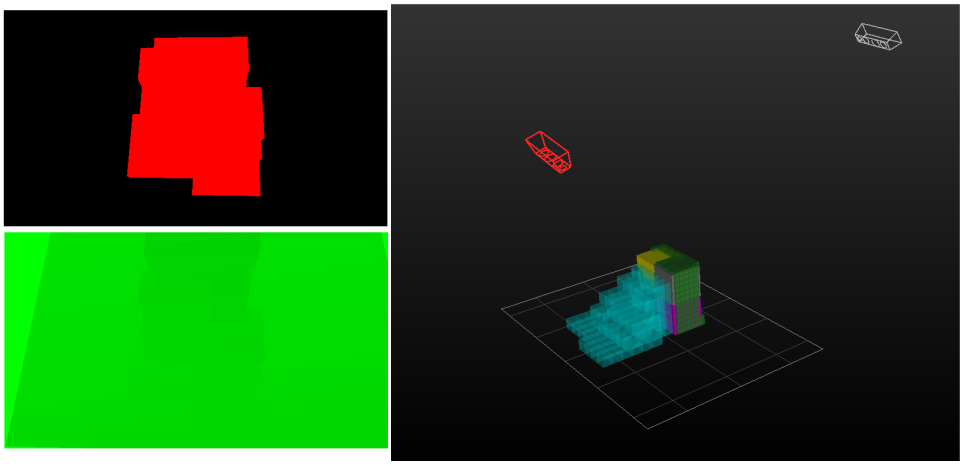


Figure 7.5: Color buffer showing shadow voxels (now pixels) (on the top left), depth buffer of the same scene, using green channel to visualize depth (on the bottom left) and the entire overview of the scene (on the right). Rendered from viewpoint of suggested (red) camera.

Because of its poor performance in its simplest form (more details in [Sec: 9], we have an enhanced version where instead of counting the number of red pixels, we sum each pixel's score values.

$$Score = \sum_{p \in Image} C_p \cdot {D_p}^{pow},$$

where C is 1 if pixel p has a red color, 0 otherwise, D is the depth distance from camera to pixel p in meters. pow is an enhancement parameter – when set to 0, the depth enhancement is mitigated and simple method of just counting amount of red pixels is used. For higher values of pow, the depth has a bigger effect on the overall score. We have experimentally found that value of $pow = 6$ gives the best results.

Polynomial degree of depth value (at given pixel) serves as a weight of depth information against shadowness (how much is given pixel red) valuing pixels that are further away more, even though there are less of them.

For simplicity we will be referring to this method as `Shadow method`, and `ShadowD0`, `ShadowD3`, `ShadowD6` for versions of this method with parameter pow set to `0`, `3` or `6` respectively.

Discussion about the method

The Shadow method described in this section has the following properties:

- **Speed:** Relatively fast method. All rendering is done on GPU, a hardware specialized for fast rendering. Scoring is processed on a GPU as well, via a fragment shader which uses `atomic_counter` to count accumulate score of the view from individual pixel score. Thanks to this shader, we do not have to download framebuffer from GPU to CPU for every

camera pose and count pixels on CPU, which would be quite costly. In the future we can experiment with multithreaded rendering techniques, which would evaluate (score) multiple camera poses at the same time, further reducing the time required for this operation.

- **Quality of estimate:** ShadowD0 tends to choose a pose "inside" or very close to a shadow voxel, because the closer the camera is, the "bigger" red voxels are (the entire view is red). "ShadowD3" and "ShadowD6" do not suffer from this issue anymore. In fact, unlike the voxel method, ShadowD3 and D6 prefer views which are further away from the scene, better accomodating for the focus and 3D precision distance sweetspot of the D435 camera.

- **Artefacts:** Looks at the "top" surface, not the entire volume of shadows. More prone to be mislead by discretization artefacts, described in AntiAliasing section [Sec: 7.2.4]. Voxelized "stairs" of depth 0-1 voxel have high score. This is mitigated using AntiAliasing technique [Sec: 7.2.4]

- **Dealing with foreign occlusions:** Does not handle scenes with foreign (not detected) objects and would-be shadows cast by them. All such shadows and occlusions are treated as if the respective parts of the scene were known, but without any `SOLID` objects. For real-world use cases, this will very likely prove problematic and insufficient.

7.4 Shadow mesh method

Same as previous methods, Shadow mesh method (or only "Mesh method" for short) also works by assigning a score value to each proposed camera pose and then selecting a pose/view with the best score (highest number):

1. Generate candidate camera poses to test.
   Shared with Voxel method. For details see [Sec: 7.2.1].

2. Construct a visibility mesh from the depth information of each camera.

3. Combine visibility meshes from multiple cameras.

4. Add reconstructed objects to work area mesh and color them.

5. Render mesh for each candidate camera pose and score the same way as described in method "Shadow method" (Only "ShadowD6" used and tested in practice).

6. Count shadow pixels in the given frame (render FOV matches real camera's FOV).

7.4.2 Construct visibility mesh

We create a 3D point from each pixel in the camera's depth image. All areas with missing data (due to occlusion, bad lighting conditions, or other reasons) are linearly extrapolated from the 3D positions of surrounding points. After this, the mesh is constructed by connecting neigbouring points in a way that creates a waterproof layer on top (similar to how heightmap works) after this, each edge of the mesh (rectangular from camera's point of view) gets connected to the 3D coordinate of camera position, creating a "pyramid-like" object, with its top vertex in the camera origin and bottom plane copies the shape of the scene and sides will represent borders of FOV. See [Fig: 7.6].

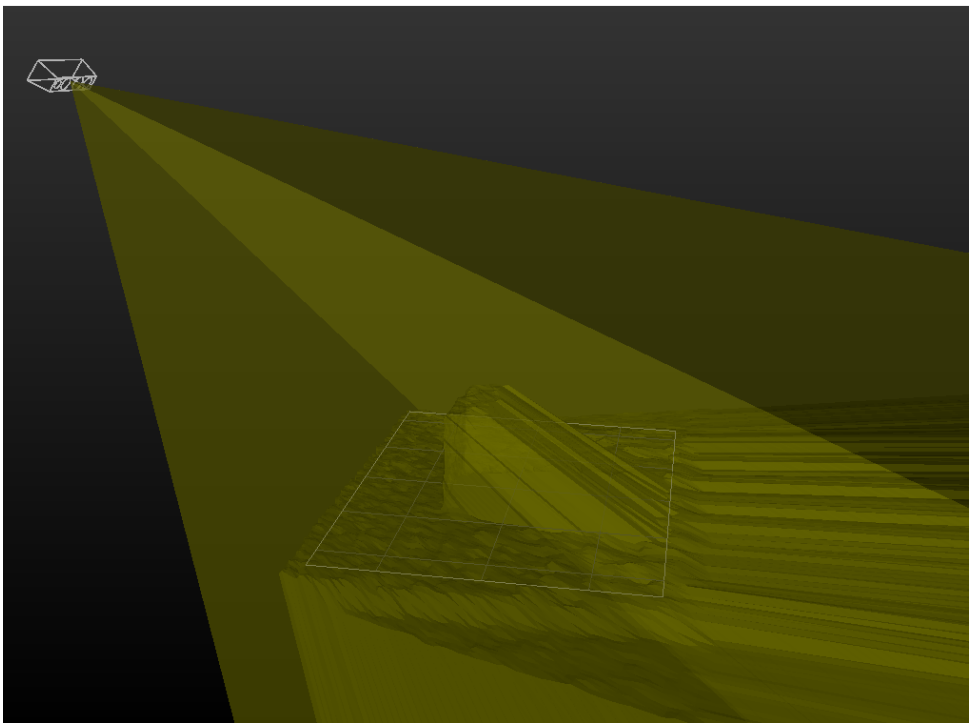


Figure 7.6: Image of camera visibility mesh on the Wall dataset (with only side camera view).

For performance reasons, in actual implementation [Src: 7] we do not construct the mesh from every neighbouring vertex, but every n-th (in current code $n = 8$). This seems to be a good balance between the level of detail and processing requirements (in later stages). We also do not use the entire depth frame returned from the camera, but only a rectangular cutout (ROI, region of interest) obtained from positions of board's ArUco markers in the camera's color frame (see [Fig: 4.1]) and construct the visibility mesh from the cutout with added border around camera's field of view, again saving quite a lot of unnecessary vertices for later processing.

Inner part of this camera mesh represents `EMPTY` space – a space that is seen through by a camera and does not contain any `SOLID` object (detectable or not).

7.4.3 Combine visibility meshes

We start with a "work area" mesh – a box spanning the size of the work area extending to the desired height above it (see left image in [Fig: 7.7]). We then do boolean subtract operation where from this mesh, we subtract visibility mesh of each camera, one after the other. We are left with an updated work area mesh. This mesh represents "`SOLID` or `SHADOW` volume". All of reconstructed objects from a scene lie inside of this mesh and everything that is not `SOLID` can be considered `SHADOW` or undetectable object (very likely casting shadow as well). See [Fig: 7.7] for visualisation of this substraction process.

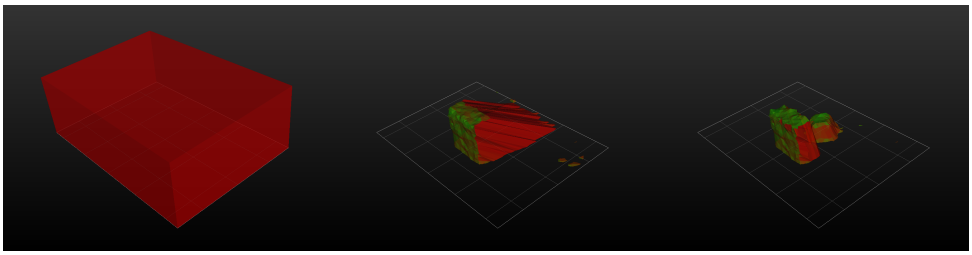


Figure 7.7: Image of work area mesh reconstruction. Initial box work area mesh (on the left), work area mesh with first camera visibility mesh substracted (in the middle) and work area mesh with second camera visibility mesh substracted (on the right).

We assign a color, or rather shadowness value v, to each face of this work area mesh. A face that is visible from its source camera (camera that was used to generate visibility mesh of which this face is part of) gets value v ranging from 0 to 1:

$$v = (\vec{F_C} - \vec{C_C})^T \cdot \vec{F_N}.$$

$\vec{F_C}$ is centroid vector of the face, $\vec{C_C}$ is center position of camera (or its origin) and $\vec{F_N}$ is face normal. Value v (0 when face is orthogonal to camera view and therefore unobservable, and 1 when face is dead front facing the camera with good visibility) is a visibility value. We have already seen and have a good idea about parts of view where this visibility value is near 1 and we do not need to see it again – either the corresponding part of an image was used to detect an object and we have it detected now, or it failed and is not useful for detecting an object and as such it would be a waste to look at the same place again. What we need to do now is to point a camera towards the mesh positions where shadowness v is near 0. We will call this parameter `shadowness` – 0 is very much a `SHADOW`, 1 is `SOLID_or_NOT_USEFUL`. Implementation of this entire process can be found at [Src: 8].

7.4.4 Add reconstructed objects

We need to add all reconstructed objects to our mesh and color them as a `SOLID_or_NOT_USEFUL` (which means with `shadowness` value $v = 1$). This step

ensures that the system does not suggest views at already known scene from a different side. E.g.: If our scene consisted of one cube and camera looking at it from top, we pretty much have all we need now. Sides of the cube would look red (= in need of another look) even though we actually do not need another look from different perspective, we have all the information we can get from the scene already. Adding a colored 3D model of a cube positioned over its detected location effectively overrides the red (shadow) color with the green (solid or not useful; in this case solid) color.

For scene reconstruction and depth information imprecisions and missalignments, we scale up the 3D model by about 20% around its origin before applying boolean add operation to the work area mesh. See [Fig: 7.8].

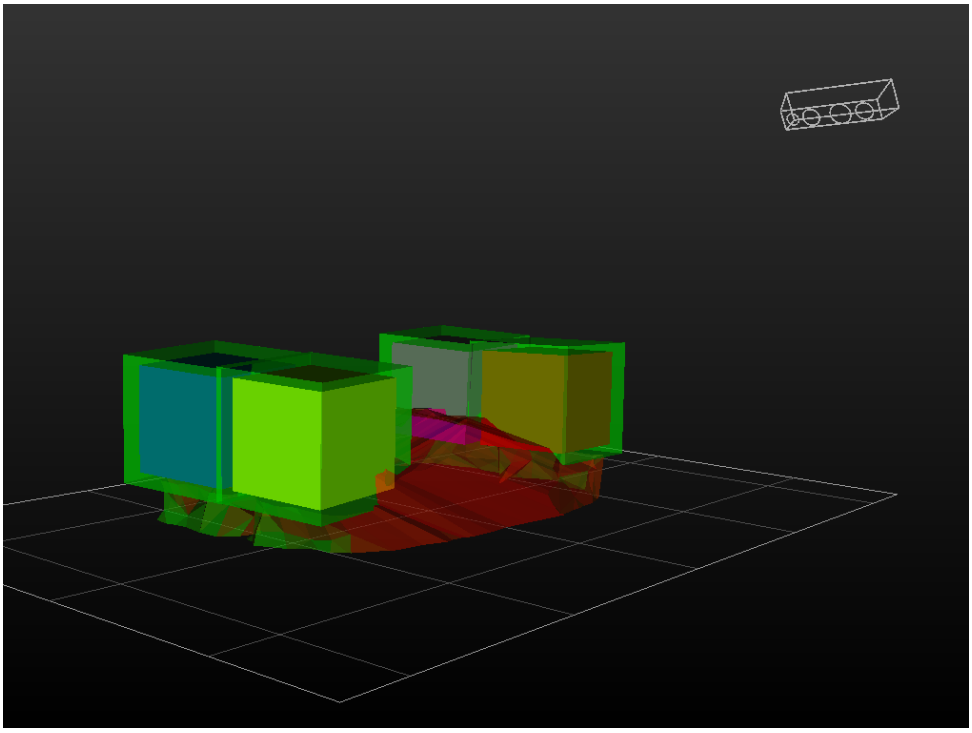


Figure 7.8: Partially transparent work area mesh with reconstructed and colored objects (green cubes) and reconstructed objects (cubes) visible through the mesh at their reconstructed poses.

7.4.5 Render colored work area mesh

This is very similar to rendering for Shadow method [Sec: 7.3.5]. The only difference is that instead of rendering `SHADOW` and `SOLID` voxels, we render entire work area mesh, with red (shadow) color according to `shadowness` value. See [Fig: 7.9].

7.4.6 Count shadow pixels

Shared with shadow method. For details see [Sec: 7.3.6]

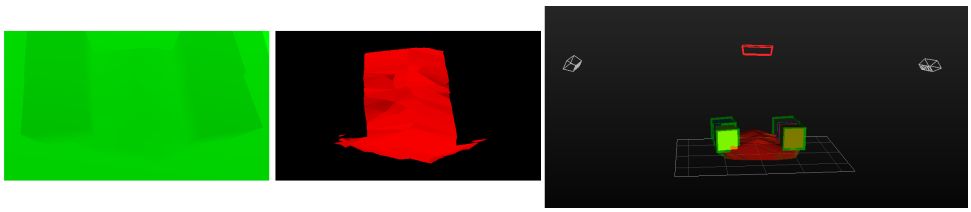


Figure 7.9: Color buffer showing shadowness of a mesh (now pixels of varying red value) (on the left), depth buffer of the same scene, using green channel to visualize depth (in the middle) and entire overview of the scene (on the right). Rendered from viewpoint of suggested (red) camera.

Discussion about the method

The Mesh method described in this section has the following properties:

- **Speed:** Method is quite slow in its current state, because the boolean operations take a lot of time (5-25 s, depending on the scene). The rendering and scoring, however, is very fast, because everything is done on GPU (just like "Shadow method" [Sec: 7.3]), but it is even faster, because we are rendering only one mesh per frame, not many (as is the case with voxels). Performance is subject to future research, namely I am interested in trying rendering method of boolean operations – the result is not a mesh, but rather an image, from give camera angle. This method has the potential to be much faster.

- **Quality of estimate:** Using ShadowdD6 (as described in [Sec: 7.3.6]) for scoring the rendered scene comes with an advantage of D3 and D6 preferring views which are further away from the scene, better accomodating for the focus and 3D precision distance sweetspot of the D435 camera.

- **Artefacts:** Looks at the "top" surface, not the entire volume of shadows. This makes reasoning about the level of importance of the shadows more challenging, especially considering noisy depth data from the sensor itself, which has a tendency to create numerous small areas of shadow on the scene. (See [Fig: 7.10])

- **Dealing with foreign occlusions:** Can handle foreign objects and occlusion caused by them (shadows cast by the foreign objects or their parts). Even partial occlusion by non-detected objects (which might even be known type of object, but occluded in such a way that is invisible for the object detection module. Furthermore, because of composing depth meshes directly, there are no artefacts caused by voxelization (non axis aligned object poses, not grid aligned poses, discretization in general)

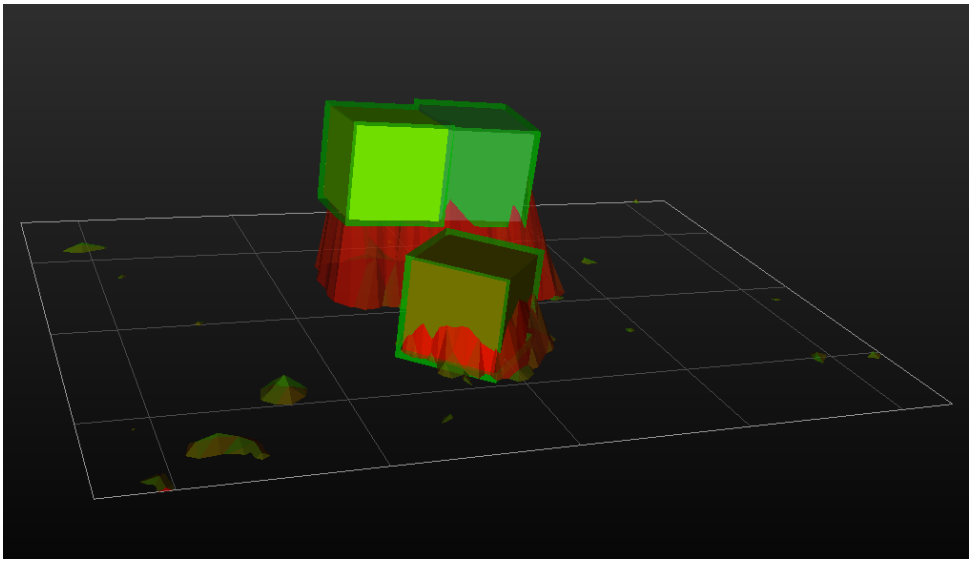


Figure 7.10: Image of small false shadow areas on the ground and walls of detected objects.

7.5 Simulated visual servoing

Simulataed visual servoing is an iterative enhancement method, that can be applied to all of the aforementioned methods. Its functionality and integration can be broken down into several steps:

1. Generate candidate camera poses to test.
2. Render scene from viewpoint of currently tested camera pose. Find bounds of shadow areas (leftmost pixel, rightmost pixel, ...) and update camera pose accordingly.
3. Repeat step `2` several times (limit in current implementation is set to 5) and record method score in each iteration.
4. Chose final candidate camera pose that had the best score in any of the iterations or even initial candidate pose itself.

7.5.1 Generate candidate camera poses

Shared with Voxel method. For details see [Sec: 7.2.1].

Fewer number of initial generated test poses is needed than in the case without any iterative enhancement.

7.5.2 Find the shadow bounds

Render scene from viewpoint of currently tested camera pose, using the same method as described in “Shadow method”. The rendered scene is actually

reused, both for scoring of Shadow method and Mesh method. For the voxel method, the shadow method's render is applied but the score value is not used.

From the rendered scene, find bounds of shadow areas (leftmost pixel, rightmost pixel, ...) and rotate the camera to center the view. If both the left and right most pixel are too close to image border, then move the camera back. Same for the top/bottom most pixels. Analogically for moving camera closer. Left/right/top/bottom most shadow pixels are calculated on GPU, using `atomic_min/max` operations. This way we still do not have to download the entire image buffer from GPU to CPU at every frame.

7.5.3 Repeat iteration

Repeat the previous step, each time with a new starting pose – the one from the previous iteration. We calculate the respective method score at each iteration of each camera pose and keep the pose with the best score. It is worth noting, that this pose will not be one of the initial poses, but a new one, which means we have to store a copy of the pose, not just its index.

7.5.4 Chose the best pose

We are interested in only one pose and that is a pose with the best method score, regardless of the way how we found it (Is it one from the initial set? Is it a new iterated one? Were some iteration useless, because the best pose was found sooner? It does not matter).

In the actual implementation [Src: 9], the steps `2`, `3` and `4` are merged and executed as one iterative step.

Discussion about the method

This simulated visual servoing can be applied to any and all methods discussed above and in all cases reduces runtime (as there are fewer poses to test) and increases quality of results (as search space of all possible poses is navigated more quickly, zeroing in on more useful poses).

Chapter 8

Experimental results – Scene reconstruction

We have been reconstructing the scene from real captured data, without ground truth, which poses a challenge on evaluation. Because of that our metric is of self-consistency of the reconstructed scene. More specifically error value e is a square distance (in mm) between corners of cubes which are supposed to touch (and therefore the desired distance is 0).

$$e = \sum_{[i,j] \in Corners} \|\vec{c_i} - \vec{c_j}\|^2,$$

where e is the error value, $Corners$ is a set of pairs of corners which are supposed to touch each other (in any direction, even red and purple cube, as shown in [Fig: 4.7 or Fig: 8.1] have two touching corners) and $\vec{c_i}, \vec{c_i}$ are positions of those corners in the scene reference frame.

For this purpose, we have created a new dataset "Castle" [Sec: 4.1.5] which has many (nine) different camera viewpoints. We have tested the reconstruction error while activating different cameras [Fig 8.1] to see, if more camera views yield better accuracy (smaller error values). We have tested all camera combinations which were able to reconstruct the full scene (have all five cubes present at the scene) and measured their error values. Graph [Fig: 8.2] shows error values clustered by camera counts.

It can be clearly seen that with the exception of the single camera-viewpoint, reconstruction is getting more accurate with the increasing number of active views (the means are getting lower and lower) and at the same time, reconstructions are getting more stable (the sample standard deviations are smaller and smaller). This demonstrates the need for multiple camera views even if the smaller number of views covers the entire scene. This importance will be more profound once we implement and use more elaborate scene reconstruction methods (described in [Sec: 6]).

The reason for the single-camera viewpoint having significantly better error values than reconstructions with few more cameras is that we are only considering local error – how much are the objects on the scene misaligned from each other. Objects reconstructed from a single view have the same absolute error (e.g., they might all be little further back than predicted) but it

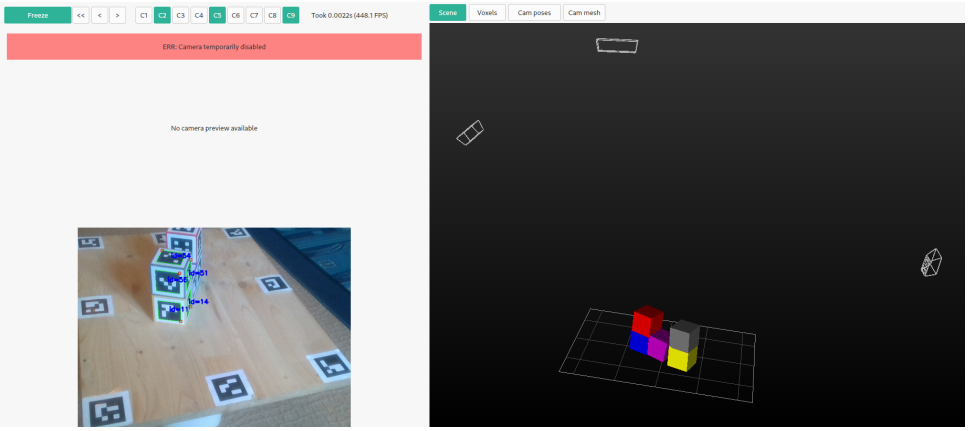


Figure 8.1: Castle dataset with most cameras disabled. Only three cameras are active at the moment. Color camera image on the left, reconstructed scene including camera poses on the right.

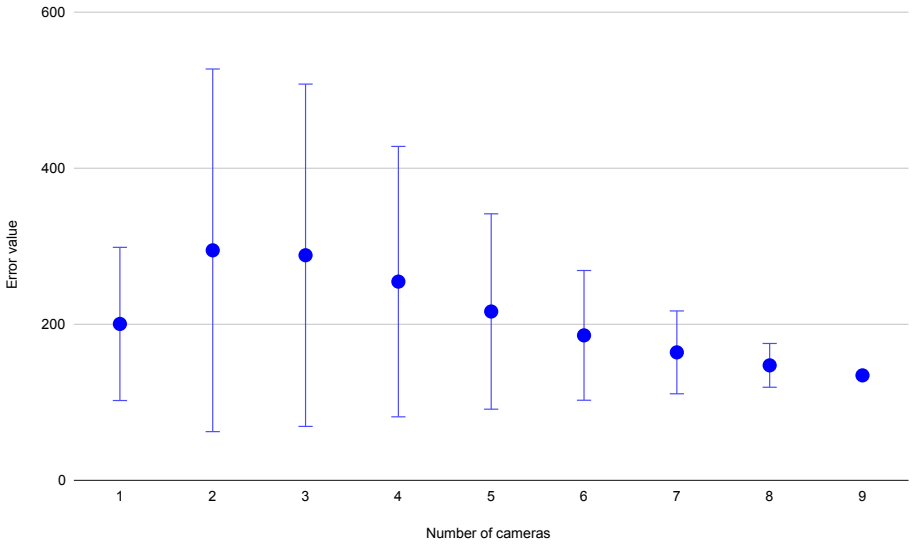


Figure 8.2: Plot showing mean error values and their respective sample standard deviation for different number of activated cameras.

is the same for all of the objects, effectively making this type of error invisible for our metric.

In the future when we will implement other scene reconstruction methods, we will also run them in a fully simulated environment, which will provide us with a ground truth for measuring this global error as well.

Chapter 9

Experimental results – Camera pose suggestion

We have tested multiple methods on multiple scenes, with a much larger number of camera poses to test than there will be in the production environment (34 225 poses, half spheres at 3 distances from the center, with several camera angles centered around "looking towards the center" viewpoint.

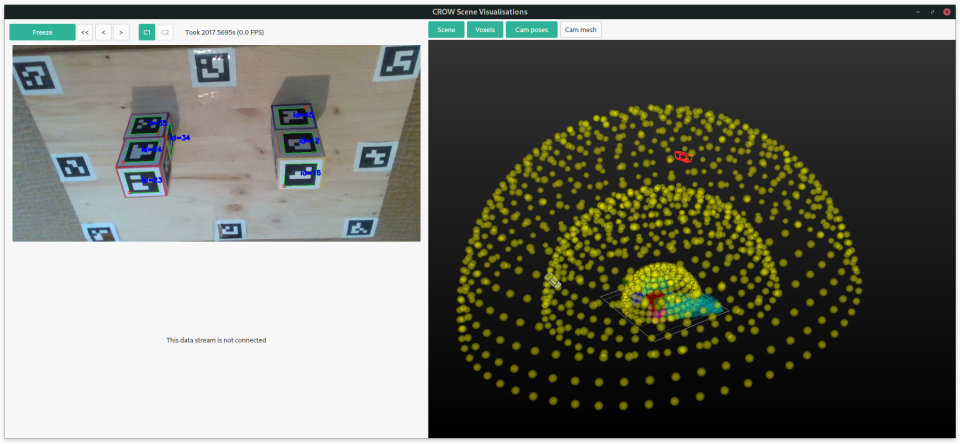


Figure 9.1: Image of camera poses relative to a scene.

Furthermore, simulated visual servoing is active, with a limit of 5 iterations per pose, meaning we are computing about 170k test views and choosing the best one. There may be multiple best views or "good enough" views which might be taken from very different poses (e.g., a view behind an object from one side or the other). We are however only interested in one pose and how good is it.

In this section we will explore different methods, their strengths and weaknesses and visual artefacts. All of the evaluated methods are fully deterministic and multiple runs will not change or increase the quality of the results.

9.1 Evaluation metric

Score value: As an evaluation metric we have chosen a score value of Vox method [Sec: 7.2], because it represents how much volume (not just surface area, as is the case with rendering) that has been previously unknown (each voxel in an unknown – `SHADOW`) area can, in reality, be `SOLID` or `EMPTY`) will become known, assuming all voxels are see-through, accommodating for the fact that there might be an unknown object somewhere "deeper" in the "shadow mountain" or even not at all (in which case, it is important to know that there is no object at all)

Vox ratio: The quality of suggested pose from a different method will be directly compared to the voxel one: $voxel\_ratio = best\_score_{diff} / best\_score_{vox}$, where $best\_score_{vox}$ is a score of the best pose, found by voxel method and $diff\_score_{vox}$ is a vox score of a pose proposed by a different method. In other words, what is the ratio of the current pose's voxel score to the best possible voxel score in this scene. This is referred to as the "vox ratio". From the design of this metric, the vox ratio of the voxel method will always be 1.

Voxel method consistency: We have tested different parameters of voxel method – lowering base grid size and increasing subdivision count – in an effort to increase voxel grid resolution and therefore precision. It turns out that poses suggested by the voxel method (on WallSide and Corridor datasets) score 97% and 89% of the maximum achievable score of the test with increased resolution. This does not necessarily mean that one set of parameters is better or worse, it just gives slightly different values. We have decided that 7 percentage points difference means that the method is consistent enough and can be used to compare other methods to itself.

Run time: Run time is a time that it took to run the method itself, on all tested camera poses. Does not involve ArUco marker detection (the detection module). Experiments have been run on i7-6700HQ laptop with 32 GB of RAM.

9.2 Visualization guide

All results are accompanied by visualisations from Visualisation guide. In these visualisations:

- New suggested camera pose is rendered in the scene using red color. Gray ones are original camera poses used to take the scene.
- In visualisations, green voxels represent solid objects, red voxels represent shadows and cyan voxels represent parts of shadows that are visible to the new suggested camera pose.
- Rendered image of the expected view from the suggested camera pose consists of 2 channels. Red represents shadow intensity and green represents depth from camera. Most of the shadow objects look yellow, because that is the combination color of green and red. See [Fig: [9.2]].

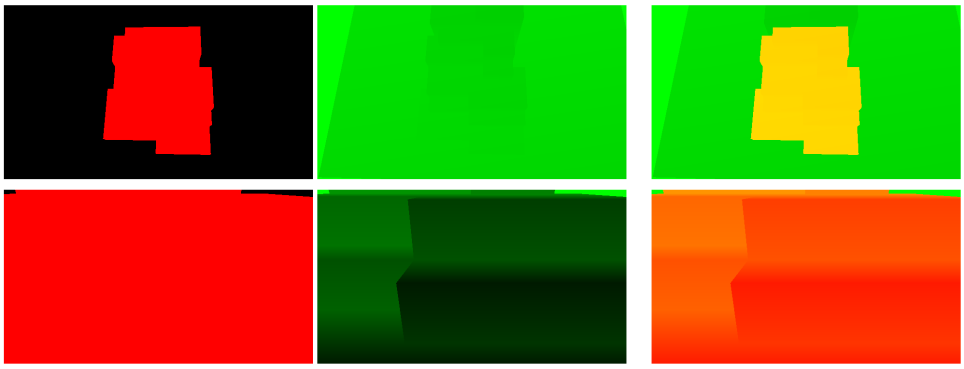


Figure 9.2: Suggested camera visualization guide: Each row is a picture of different scene. Red channel (isolated on the left side) shows where shadow from scene get projected to camera view. Green channel (isolated in the middle) shows distance from camera to an object (object in bottom row is very close, as the values are near zero (near black)). Image on the far right is combination of both red and green channel.

9.3 Different datasets

9.3.1 Wall dataset with only side camera enabled

Voxel method: Best camera pose using the voxel volumes method is shown in [Fig: 9.3], with score of 9.43 E-04 (cubic meters of shadow. The higher number the better).

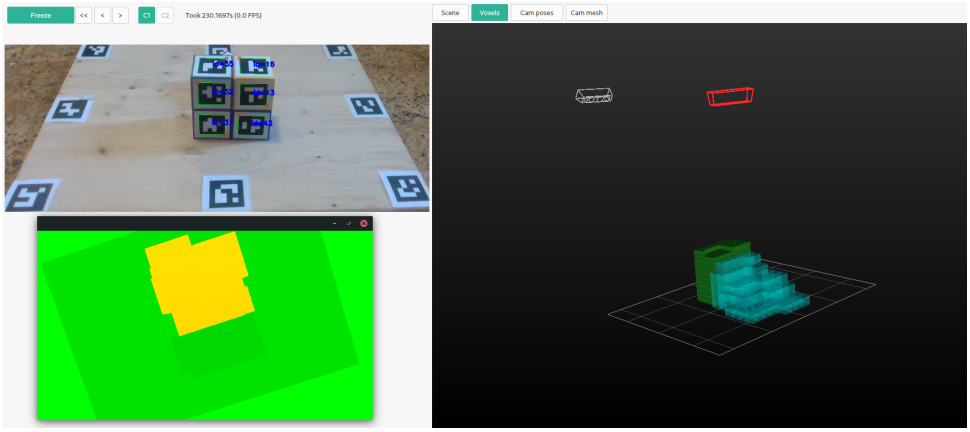


Figure 9.3: Best camera pose using the voxel volumes method on Wall dataset. Color camera stream on the top left, suggested camera view on the bottom left and reconstructed scene on the right, with suggested camera pose shown in red. Refer to visualization guide [Fig 9.2].

As we can see, the suggested camera is positioned above shadow mountain, looking straight down at it, having its entirety in view. It can be argued that this pose is similar to the one where a human would place the camera.

Shadow method: We have tested the same scene using the "Shadow method", where red (shadow) pixels are used as a stencil and the score is then sum of $depth^{pow}$ at those shadow pixels. pow is a parameter, and the experiment was run for values $pow = 0$, 3, and 6.

For 'pow = 0' – "ShadowD0" – (therefore ignoring the depth value, just counting red pixels) see [Fig: 9.4].

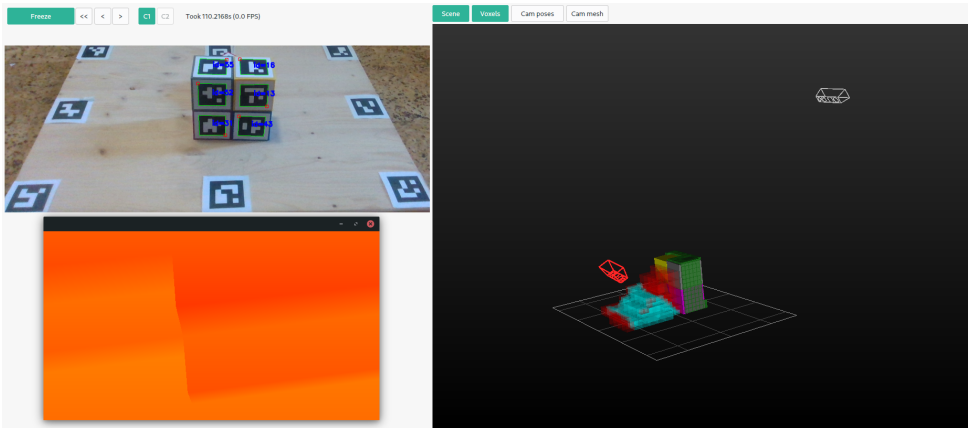


Figure 9.4: Best camera pose according to pixel count with ShadowD0 method. Color camera stream on the top left, suggested camera view on the bottom left and reconstructed scene on the right, with suggested camera pose shown in red. Refer to visualization guide [Fig 9.2].

As we can see, the camera is zoomed in as close as it can go towards the shadow, because the closer it gets, the more red pixels it will see, up to a point when the pixels fill its entire view, yielding maximum score. This artefact can be tackled by taking into account depth at which the camera sees the pixels, counteracting the effect of "the closer object is, the bigger it appears"

Visual servoing: You can see first 5 iterations of visual servoing, using ShadowD3 in [Fig: 9.5] for one initial camera pose.

The camera movement between iteration is caused by the "simulated visual servoing" algorithm, which rotates the camera to move visible shadow pixels to the center of an image and moves the camera backward if the red pixels are too close to the edges (it is zoomed in way too much) and moves it forward when all of the shadow pixels are positioned only in the middle (zoomed out way too much, resulting in lack of detail).

Same experiment has been done for 'pow = 6' – "ShadowD6" ([Fig: 9.6, Table: 9.1]).

The best view (according to this method) is at iteration number 4 (center image in bottom row) (and this initial camera pose) and all but the first iteration have VOX score 100% of achievable (at this scene).

9.3.2 Two Pillars

ShadowD0 vs ShadowD6: The extreme case of near-touching the object (using ShadowD0 method) could be easily filtered out using only further away

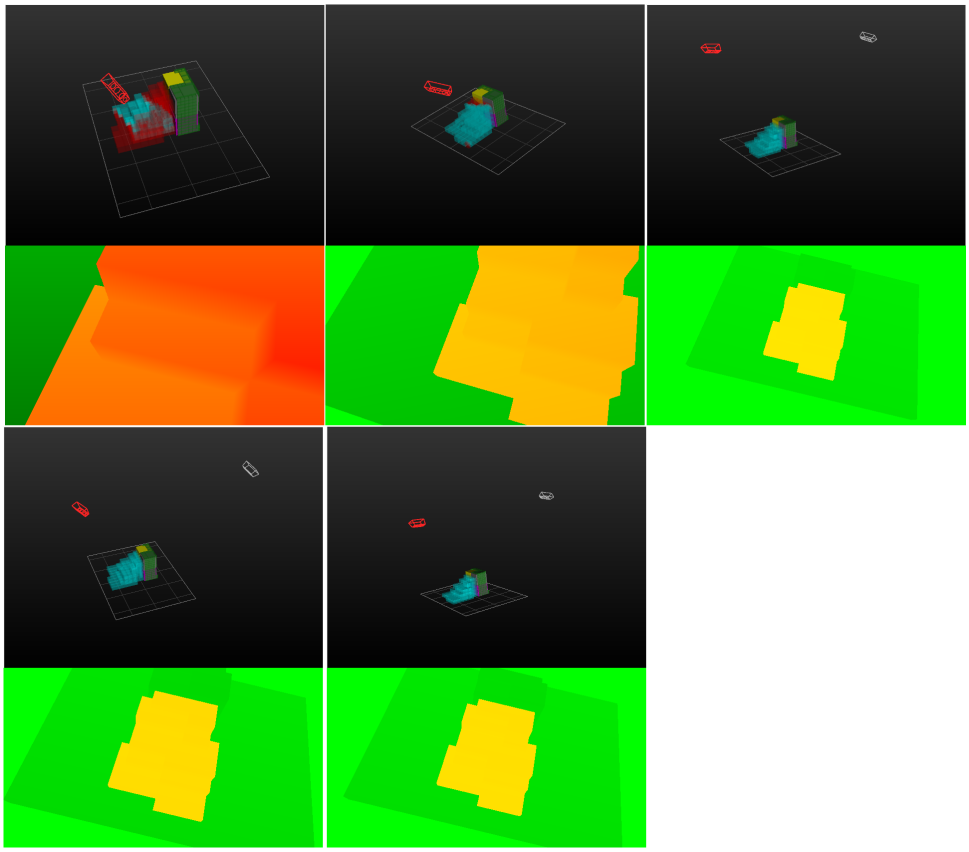


Figure 9.5: First 5 iterations of visual servoing algorithm using "ShadowD3" method. Iteration 0 is in top left corner. Suggested camera view on the bottom of each iteration and reconstructed scene on the top, with suggested camera pose shown in red. Refer to visualization guide [Fig 9.2].

Visual servoing iteration	Method score	Vox score	Vox ratio
0	0.011 E+06	5.67 E-04	60.1%
1	1.51 E+06	9.43 E-04	100%
2	1.66 E+06	9.43 E-04	100%
3	1.56 E+06	9.43 E-04	100%
4	**1.99 E+06**	**9.43 E-04**	**100%**
5	1.56 E+06	9.43 E-04	100%

Table 9.1: Score values for visual servoing iteration on Wall dataset with ShadowD6 method. Best iteration according to the method score is in bold, which does not imply the best vox ratio.

initial poses or introducing some area where the camera is not allowed to be in. However, that does not solve the problem of it favoring views of objects that are closer. This effect is most profoundly visible on the "TwoPillars" dataset, where two pillars are casting two distinct non-overlapping shadows. Without the depth value in the score, the camera is just trying to look at one, not both shadows at the same time, because it can be closer to it.

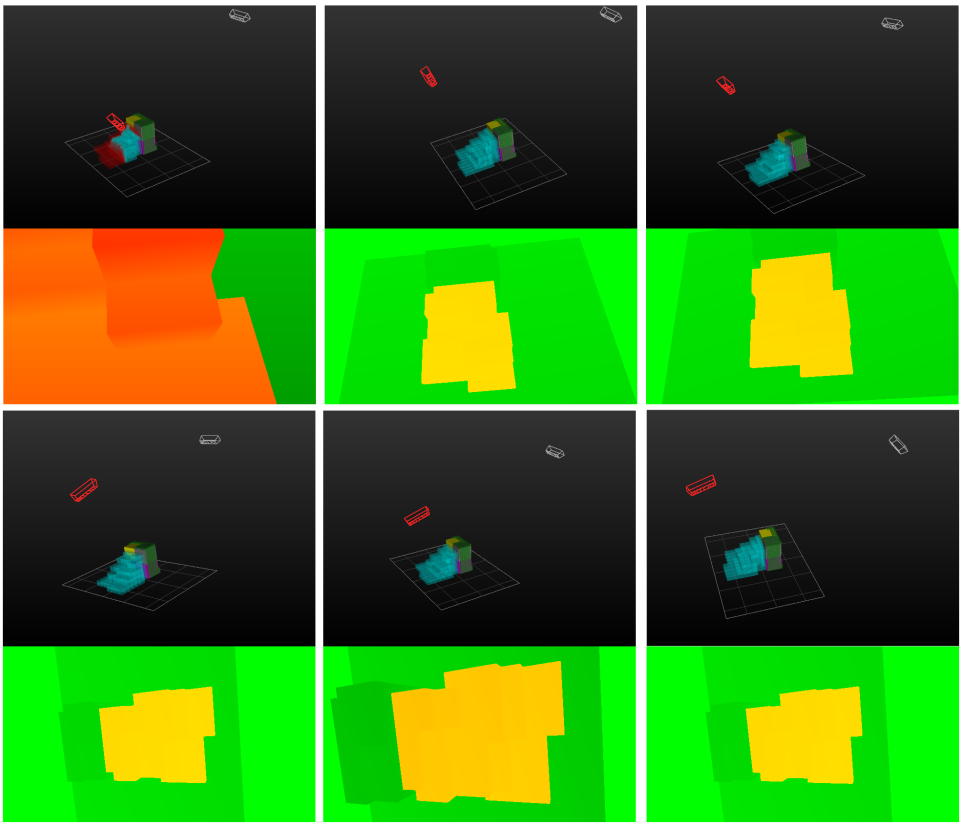


Figure 9.6: 6 iterations of visual servoing algorithm using "ShadowD6" method. Iteration 0 is in top left corner. Suggested camera view on the bottom of each iteration and reconstructed scene on the top, with suggested camera pose shown in red. Refer to visualization guide [Fig 9.2].

The voxel shape that appears to be floating in the air is actually shadow area caused by voxels being outside of FOV of the current camera on the scene and therefore are not visible. The methods are using this kind of shadows as well, for suggesting the camera pose. As you can see, viewpoint suggested by ShadowD6 contains the outside of FOV shape and both shadows cast by pillars.

In the [Fig: 9.8] shows the same task with camera FOV constraint artificially removed.

9.3.3 Corridor

Corridor scene yields sub-perfect vox ratio score for all ShadowD6 [Fig: 9.10, Table: 9.2], ShadowD3 [Fig: 9.11, Table: 9.3] and ShadowD0 [Fig: 9.12, Table: 9.4] methods:

```
Best vox score: 6.20 E-04
```

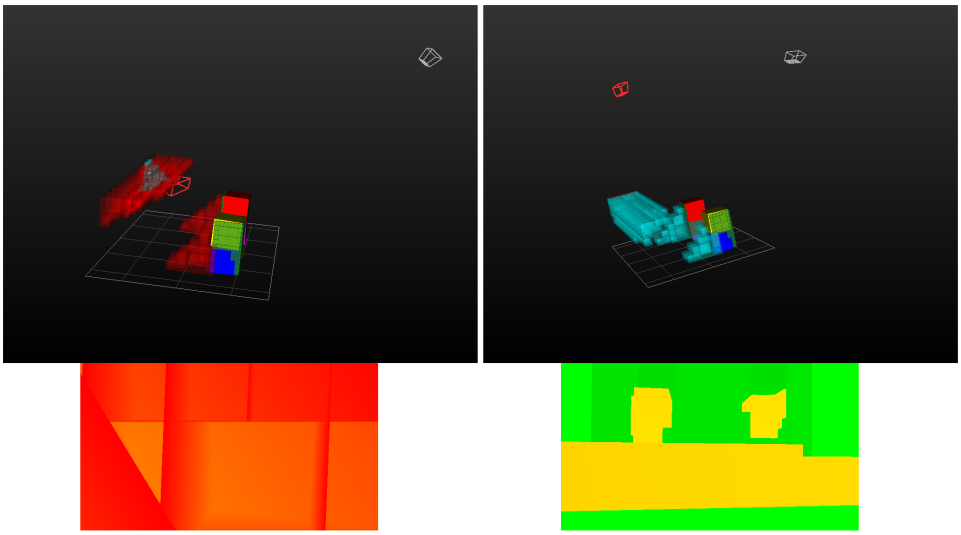


Figure 9.7: Comparison of best poses using ShadowD0 (on the left) and ShadowD6 (on the right). Suggested camera view on the bottom and reconstructed scene on the top, with suggested camera pose shown in red. Refer to visualization guide [Fig 9.2].

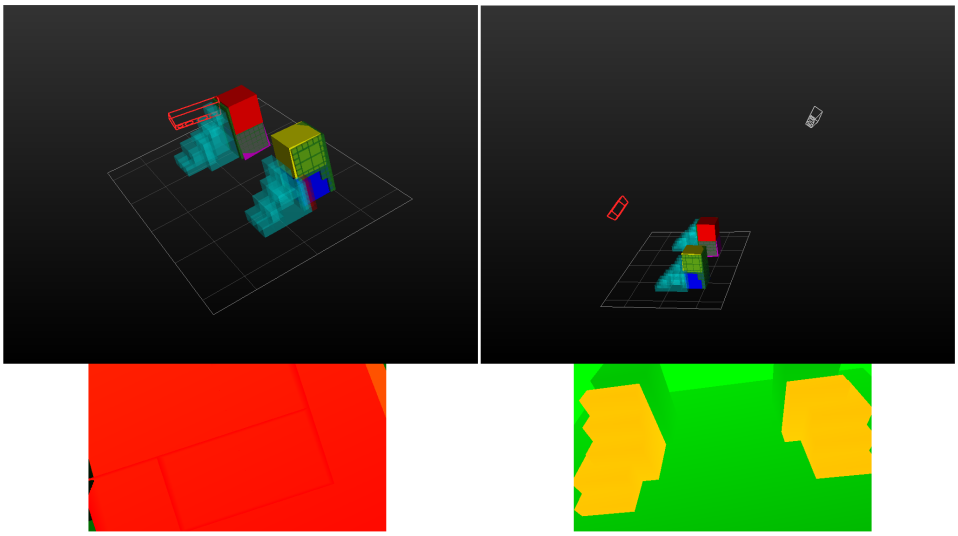


Figure 9.8: Comparison of best poses with artificially remove camera FOV constraint using ShadowD0 (on the left) and ShadowD6 (on the right). Suggested camera view on the bottom and reconstructed scene on the top, with suggested camera pose shown in red. Refer to visualization guide [Fig 9.2].

9.4 Summary and evaluation of experimental results

Methods have been evaluated in two different ways – qualitatively, using visualisation techniques, and quantitatively using score metrics introduced in

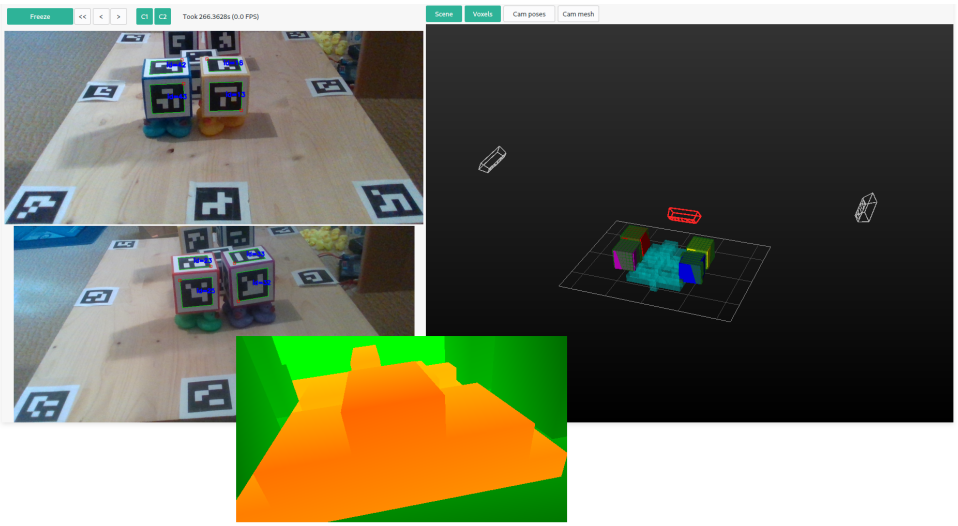


Figure 9.9: Corridor scene using VOX method. Color images of scene on the left, reconstructed scene on the right, with red suggested camera pose and camera view on the bottom.

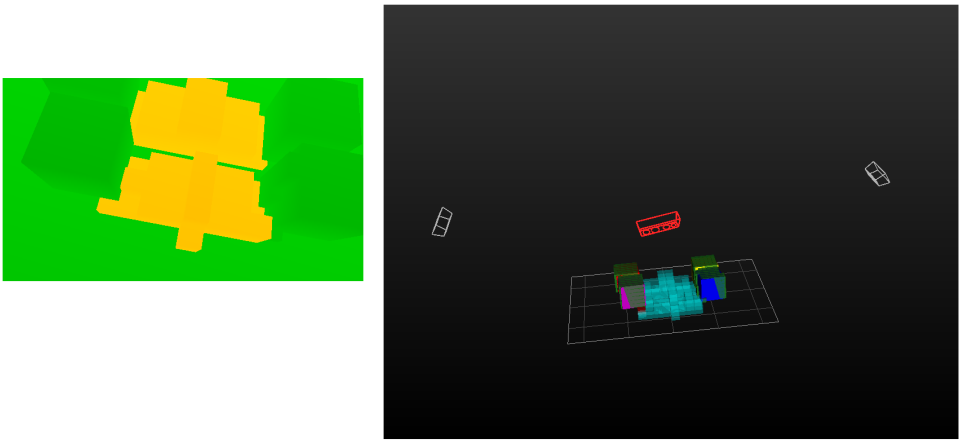


Figure 9.10: Corridor scene using ShadowD6 method. Suggested camera view on the left, reconstructed scene on the right, with red suggested cam pose.

Visual servoing iteration	Method score	Vox score	Vox ratio
0	0.019 E+06	1.92 E-04	31.1%
1	1.26 E+06	5.29 E-04	85.4%
2	**1.49 E+06**	**5.97 E-04**	**96.4%**
3	1.07 E+06	5.87 E-04	94.8%
4	1.43 E+06	5.91 E-04	95.3%
5	1.18 E+06	5.87 E-04	94.8%

Table 9.2: Score values for visual servoing iteration on Corridor dataset with ShadowD6 method. Best iteration according to the method score is in bold, which does not imply the best vox ratio.

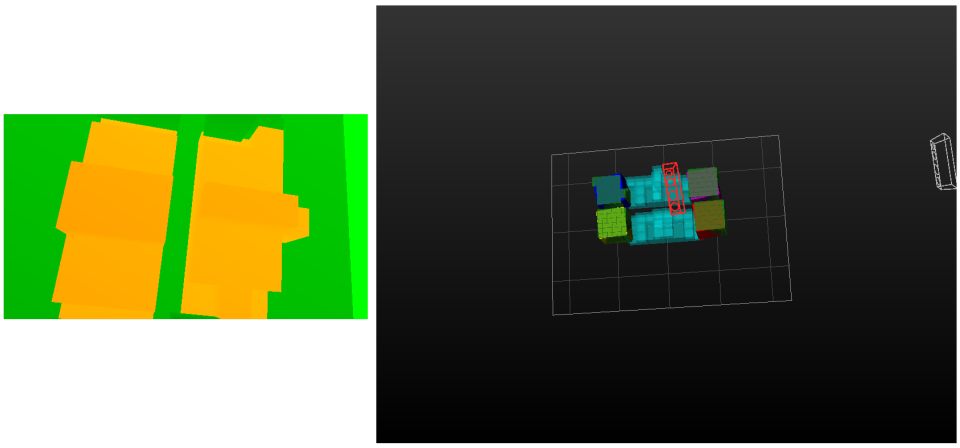


Figure 9.11: Corridor scene using ShadowD3 method. Suggested camera view on the left, reconstructed scene on the right, with red suggested cam pose.

Visual servoing iteration	Method score	Vox score	Vox ratio
0	0.082 E+06	1.17 E-04	18.8%
1	0.158 E+06	1.38 E-04	22.3%
2	1.85 E+06	5.72 E-04	92.3%
3	3.48 E+06	5.90 E-04	95.2%
4	1.41 E+06	5.67 E-04	91.4%
5	**3.72 E+06**	**5.90 E-04**	**95.2%**

Table 9.3: Score values for visual servoing iteration on Corridor dataset with ShadowD3 method. Best iteration according to the method score is in bold, which does not imply the best vox ratio.

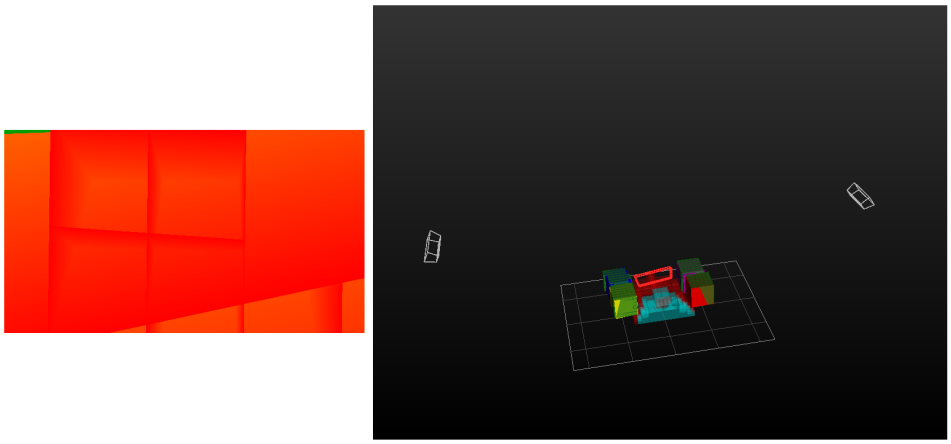


Figure 9.12: Corridor scene using ShadowD0 method. Suggested camera view on the left, reconstructed scene on the right, with red suggested cam pose.

[Sec: 9.1].

Visual servoing iteration	Method score	Vox score	Vox ratio
0	1.05 E+06	3.01 E-04	48.5%
1	**22.94 E+06**	**2.60 E-04**	**42.0%**
2	3.58 E+06	5.97 E-04	96.4%
3	4.42 E+06	5.97 E-04	96.4%
4	3.44 E+06	5.97 E-04	96.4%
5	4.39 E+06	5.97 E-04	96.4%

Table 9.4: Score values for visual servoing iteration on Corridor dataset with ShadowD0 method. Best iteration according to the method score is in bold, which does not imply the best vox ratio.

9.4.1 Iterative visual servoing

Results of every tested method improved with the addition of iterative visual servoing. The method score and the vox ratio is in every case (see [Table: 9.1, Table: 9.2, Table: 9.3, Table: 9.4]) better in some iteration other than 0. It is worth noting that neither the method score nor vox ratio is always improving monotonically with visual servoing iterations and it is not expected that increasing the limit of 5 iterations would have a dramatic impact on the quality of suggested pose.

This behaviour is caused by the simulated camera rotating and moving without prior knowledge of what will be seen after that movement – suppose there is a lot of shadow area on the left side of the camera's view, it then rotates to the left, because there might have been more shadow hidden behind the left border of the image. In fact, the next frame shows, that there is no additional shadow at all. Some of the shadow that was visible in the previous frame, towards the right side, is not visible anymore, because the camera rotation moved the shadow outside of the frame. This visual servoing iteration, therefore, has a lower method score and vox ratio than the previous one and will not get chosen at the end.

9.4.2 Special cases

All of the methods mentioned above (including the VOX) have a common issue: They cannot deal with shadows caused by not-detected/non-detectable objects. If there is an occlusion or a partial occlusion of non-scene nature (human hand, robot hand, dirt on one of the cameras, foreign objects on the scene...), there is no object on the scene that can cast shadows and therefore no shadows, making the entire workspace visible. "Mesh method" attempts to solve this issue, by using the depth data from the camera directly, bypassing the object detection and merging modules.

On a Corridor dataset with only one side view available, we can clearly see the difference between shadow reconstructed using the mesh method and voxel method. Cubes from the far side cast a shadow as well, even though they are only partially visible and not detected. See [Fig: 9.13].

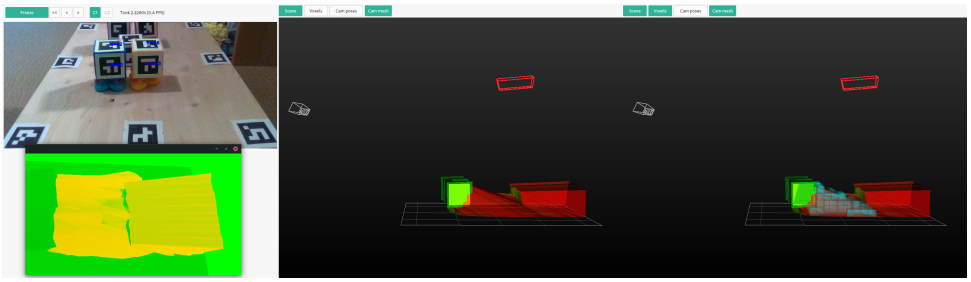


Figure 9.13: Camera pose suggestion using mesh method. Reconstructed scene with red shadow mesh and suggested camera (in the middle) and the same scene with voxel-based shadow overlaid (cyan voxels) (on the right). Camera source color image on the top left, suggested cam pose view on the bottom left. Refer Refer to visualization guide [Fig 9.2].

The advantage of the Shadow mesh method is even more visible on the ForeignObjects dataset, where there are no detected and reconstructed cubes on the scene, and yet this method suggests correct camera pose – a view from the other side. Unlike the vox method, which does not suggest any view at all because of no shadows present in the scene. See [Fig: 9.14, Fig: 9.15].

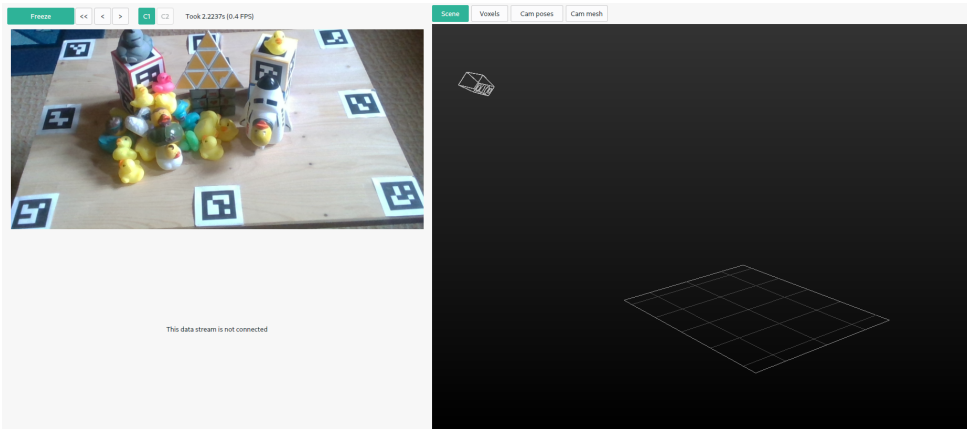


Figure 9.14: Camera pose suggestion (there is none) using voxel method on ForeignObjects dataset.

9.4.3 Comparison of methods

As has been already shown, “ShadowD6” is on all datasets superior to “ShadowD0” and even “ShadowD3”. Let us remove D0 and D3 from further comparisons and keep only D6 as a representative of the Shadow method.

Relative quality comparison of suggested camera pose in terms of vox score on different datasets, excluding ForeignObjects dataset to keep methods comparable (Mesh method is the only one that can handle it correctly) is shown in [Table: 9.5] and [Fig: 9.16].

VOX method gives the best result, but is significantly slower than all of the other ones. D6 gives results only slightly worse than VOX, but is 2-3 times

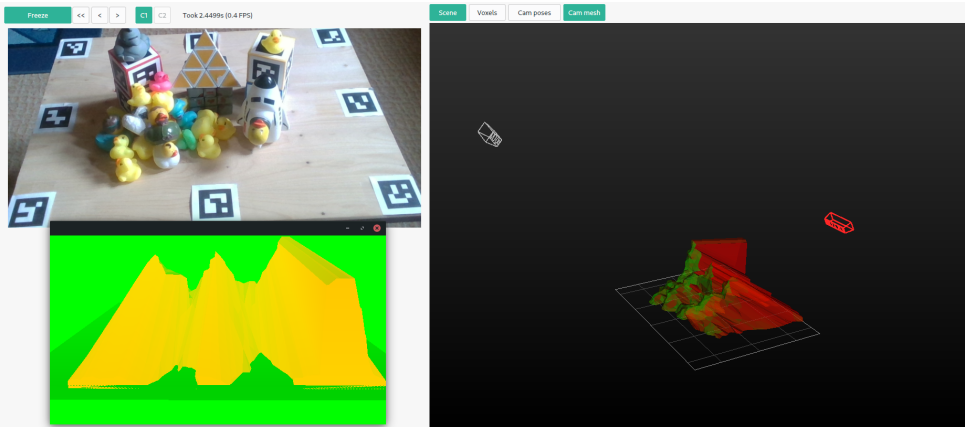


Figure 9.15: Camera pose suggestion (red color, other side of the scene) using mesh method on ForeignObjects dataset.

Method	Method score	Vox score	Vox ratio	Run time
Wall dataset with side camera view				
VOX	9.43 E-04	9.43 E-04		234.8 s
ShadowD6	1.99 E+06	9.43 E-04	100%	107.6 s
Mesh	2.23 E+06	9.43 E-04	100%	**74.4 s**
Wall dataset with top camera view				
VOX	3.23 E-04	3.23 E-04		174.9 s
ShadowD6	0.796 E+06	3.16 E-04	98.0%	100.9 s
Mesh	0.559 E+06	2.49 E-04	77.0%	**52.2 s**
TwoPillars dataset				
VOX	22.81 E-04	22.81 E-04		480.9 s
ShadowD6	3.93 E+06	22.71 E-04	99.6%	150.9 s
Mesh	4.62 E+06	20.87 E-04	91.5%	**86.5 s**
Corridor dataset with both side view cameras				
VOX	6.20 E-04	6.20 E-04		272.8 s
ShadowD6	1.49 E+06	5.97 E-04	96.4%	152.3 s
Mesh	1.43 E+06	5.95 E-04	96.0%	**72.5 s**
Averaged results				
VOX				290.9 s
ShadowD6			98.5%	127.9 s
Mesh			91.1%	**71.4 s**

Table 9.5: Comparison of best suggested camera poses according to different methods.

faster. Mesh method does not always give a very good score (namely Wall dataset with only the Top camera enabled) but can handle ForeignDataset well. This method is even faster than D6, but requires a one-time calculation of a mesh, which takes long time compared to other methods when evaluating only a few initial camera poses (see table below)

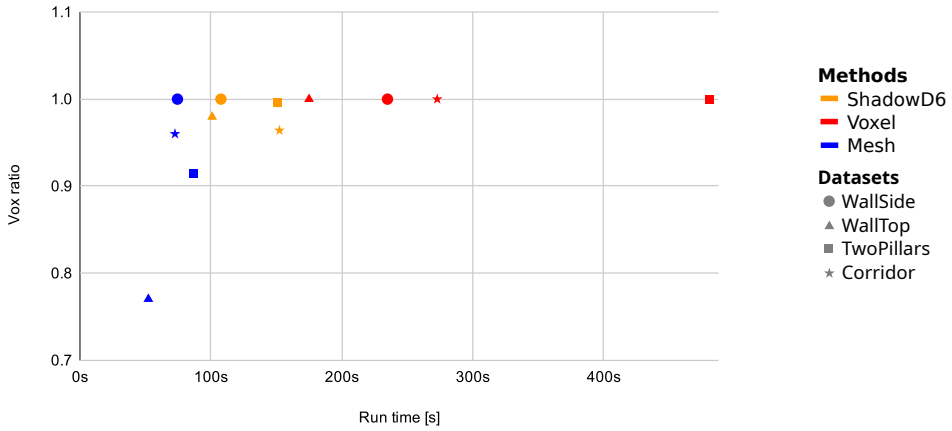


Figure 9.16: Graph showing run time (on horizontal axis) and voxel ratio (on vertical axis) of different methods and datasets, on 34 225 of initial camera poses. From the design of this metric, vox ratio of voxel method will always be 1.

Because the evaluation of the methods was comparing them against each other, the results are biased towards the one chosen "master" method, however still useful for understanding these methods in more detail.

9.4.4 Evaluation vs Production environment

For evaluation purposes we have used way more initial camera poses (34 225) than would be practical for real-time or near real-time calculations. In the production environment, we only use 40 base poses, 2 half spheres, each with a different radius, with each camera looking to the center.

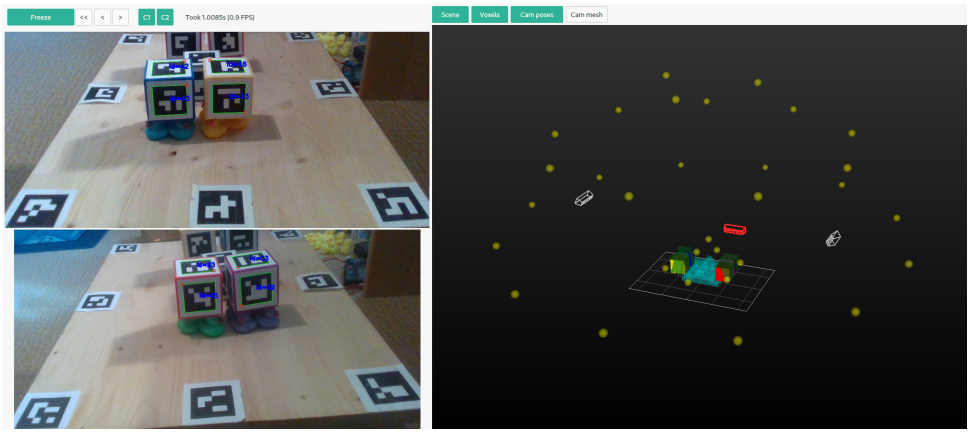


Figure 9.17: Image of fewer camera poses relative to a scene.

This combined with iterative visual servoing mechanism seems to be enough, as the table below shows. Vox quality is ratio $vox\_quality = vox\_score_{full}/vox\_score_{production}$, where vox_score_{full} is a voxel score of camera pose suggested by the evaluated method on the full set of initial camera

poses (34 225) and $vox\_score_{production}$ is a voxel score of camera pose suggested by the evaluated method on production set of initial camera poses (40).

Method	Vox quality	Speedup
Wall dataset with side camera view		
VOX	1	234.8 s -> 0.76 s
ShadowD6	1	107.6 s -> 0.60 s
Mesh	1	74.4 s -> 3.2 s
Wall dataset with top camera view		
VOX	1	174.9 s -> 0.62 s
ShadowD6	0.99	100.9 s -> 0.51 s
Mesh	0.98	52.2 s -> 2.6 s
TwoPillars dataset		
VOX	1	480.9 s -> 1.1 s
ShadowD6	0.96	150.9 s -> 0.62 s
Mesh	1.09	86.5 s -> 3.8 s
Corridor dataset with both side view cameras		
VOX	1	272.8 s -> 0.90 s
ShadowD6	0.99	152.3 s -> 0.72 s
Mesh	0.99	72.5 s -> 7.0 s
Averaged results		
VOX	1	290.9 s -> 0.85 s
ShadowD6	0.99	127.9 s -> 0.61 s
Mesh	1.02	71.4 s -> 4.2 s

Table 9.6: Method behaviour on smaller initial camera poses set (40 instead of 34 225). Vox quality is a ratio between vox score of the particular method on full 34 225 initial poses to vox score of the method on production 40 initial poses.

For comparison with the full camera poses (34 225), [Fig: 9.18] shows different methods and datasets with only 40 initial camera poses, in the same way as [Fig: 9.16] shows it for all initial camera poses. It is worth noticing that run times of Mesh method are slower than linear with the number of initial poses. The one-time mesh processing (boolean operations) takes a lot of time which is not negligible when evaluating only on few initial camera poses.

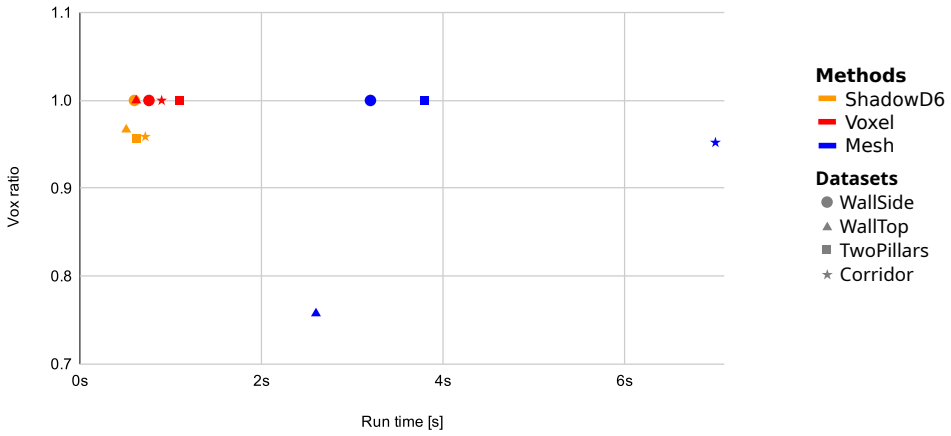


Figure 9.18: Graph showing run time (on horizontal axis) and voxel ratio (on vertical axis) of different methods and datasets, with only 40 initial camera poses. You can compare with [Fig: 9.16] which shows the same thing, only with 34 225 initial camera poses. From the design of this metric, vox ratio of voxel method will always be 1.

Chapter 10

Conclusion and discussion

We have developed and qualitatively compared multiple methods for both, scene reconstruction and camera pose suggestion modules in the vision node architecture [Sec: 5]. We have evaluated the importance of multiple camera views on scene reconstruction stability [Sec: 8] and we have also done a quantitative evaluation of camera pose suggestion methods [Sec: 9].

Each method has its strengths and weaknesses, or even types of scenarios which it cannot handle. Voxel method has good results and takes into account the entire volume of shadow, not just its projected cross-section, but on the other hand does not take into account distance between camera and observed objects. Shadow method does take this distance into account, but has no information about depth or "penetration" of shadows. As it turns out during the quantitative evaluation, this does not seem to be that big of a deal, when ShadowD6 is used (opposed to D0 or D3). Shadow method is more than two times faster than the voxel method at the cost of only 1.5% vox ratio loss, as [Table: 9.5] shows.

Neither voxel nor Shadow method can deal with foreign object scenarios – undetectable objects, such as human or robot hands, random coffee mug in front of the camera, causing occlusion and shadows, without reconstructing objects which are the causes of said shadows.

Mesh method solves this problem by directly utilizing depth image from the cameras and constructing a global visibility mesh, which it then analyzes the same way as the Shadow method. At its current state, the mesh method is too slow for realtime applications (several seconds per frame, see [Table: 9.6]) and does not have a score on par with Voxel or ShadowD6 (8.9% vox ratio loss, see [Table: 9.5], mainly because of artefacts caused by imprecise depth sensor data, resulting in small false shadow areas [Fig: 7.10].

Each of camera suggestion methods accommodate for the physical constraints of the environment, by generating only valid camera poses. Valid in this case means "within reach of the robot arm and not intersecting with a possible structure (camera fixtures, light fixtures, ...) of the environment."

10.1 Future work

During the development of this work we have identified several approaches to continue and improve this work:

- Improve mesh method speed. More specifically, explore different methods or libraries for mesh construction, one of which is `OpenCSG` [23], a library for rendering the result of boolean operations, from a specific camera viewpoint, rather than calculating the mesh itself. This has the potential to be much faster, when relatively low amount of renders (camposes to evaluate) is required.
- Develop a filter for mesh method, which would remove small false shadow areas on the ground and walls of detected objects [Fig: 7.10]. We think that this will improve the method's voxel score dramatically, to be on par with the ShadowD6 method.
- Experiment with multithreaded rendering techniques, to improve speed of Shadow and Mesh camera pose suggestion pose methods. Voxel method would benefit slightly as well, because it uses rendered scene for simulated visual servoing.
- Assignment module, as was mentioned in chapter on Architecture. This assignment module will be needed for the final production environment, where it is important to not only what the current scene looks like, but also what has changed since the previous one.
- Implement scene reconstruction methods, which are currently at a conceptual level only and possibly explore other methods (See [Sec: 6]). Generic improvements to scene reconstruction such as Temporal filtering [Sec: 6.4.1], Reasoning about empty spaces [Sec: 6.4.2], Reasoning about support and stability [Sec: 6.4.3] and Proposal of disambiguation camera views [Sec: 6.4.4].

Resources

Literature

[1] Yinlin Hu et al. *Segmentation-driven 6D Object Pose Estimation*. 2018. arXiv: 1812.02541 [cs.CV].

[2] Joel Vidal et al. "A Method for 6D Pose Estimation of Free-Form Rigid Objects Using Point Pair Features on Range Data". In: *Sensors* 18 (Aug. 2018), p. 2678. DOI: 10.3390/s18082678.

[3] Jonathan Tremblay et al. *Deep Object Pose Estimation for Semantic Robotic Grasping of Household Objects*. 2018. arXiv: 1809.10790 [cs.RO].

[4] Tomas Hodan et al. "Detection and fine 3D pose estimation of texture-less objects in RGB-D images". In: Sept. 2015, pp. 4421–4428. DOI: 10.1109/IROS.2015.7354005.

[5] R. Szeliski. "Scene reconstruction from multiple cameras". In: *Proceedings 2000 International Conference on Image Processing (Cat. No.00CH37101)*. Vol. 1. 2000, 13–16 vol.1.

[6] Daeyun Shin, Charless C. Fowlkes, and Derek Hoiem. *Pixels, voxels, and views: A study of shape representations for single view 3D object shape prediction*. 2018. arXiv: 1804.06032 [cs.CV].

[7] Li Tao and Xuerong Xiao. "3D Scene Reconstruction from Multiple Uncalibrated Views". In: (2016).

[8] Andre Uckermann, Robert Haschke, and Helge Ritter. "Real-time 3D segmentation of cluttered scenes for robot grasping". In: Nov. 2012, pp. 198–203. DOI: 10.1109/HUMANOIDS.2012.6651520.

[9] Zhaoyin Jia et al. "3D-Based Reasoning with Blocks, Support, and Stability". In: June 2013, pp. 1–8. DOI: 10.1109/CVPR.2013.8.

[10] G. García et al. "Guidance of Robot Arms using Depth Data from RGB-D Camera". In: vol. 2. July 2013. DOI: 10.5220/0004481903150321.

[11] Georgios Passalis et al. “General Voxelization Algorithm with Scalable GPU Implementation”. In: *Journal of Graphics, GPU, and Game Tools* 12 (Jan. 2007), pp. 61–71. DOI: 10.1080/2151237X.2007.10129233.

[12] R. Fischler and M. Bolles. “Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography”. In: *Commun ACM* 24 (Jan. 1981), pp. 619–638.

[13] Sergio Garrido-Jurado et al. “Generation of fiducial marker dictionaries using Mixed Integer Linear Programming”. In: *Pattern Recognition* 51 (Oct. 2015). DOI: 10.1016/j.patcog.2015.09.023.

[14] Francisco Romero-Ramirez, Rafael Muñoz-Salinas, and Rafael Medina-Carnicer. “Speeded Up Detection of Squared Fiducial Markers”. In: *Image and Vision Computing* 76 (June 2018). DOI: 10.1016/j.imavis.2018.05.004.

[15] Donald Meagher. *Octree Encoding: A New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer*. Oct. 1980.

[16] Stanford Artificial Intelligence Laboratory et al. *Robotic Operating System*. Version ROS Melodic Morenia. URL: https://www.ros.org.

[17] Intel Corporation. *librealsense – Intel RealSense SDK*. URL: https://github.com/IntelRealSense/librealsense.

[18] G. Bradski. “The OpenCV Library”. In: *Dr. Dobb’s Journal of Software Tools* (2000).

[19] Intel Corporation. *The Computational Geometry Algorithms Library*. URL: https://www.cgal.org/.

[20] Alec Jacobson, Daniele Panozzo, et al. *Simple C++ geometry processing library*. URL: https://github.com/libigl/libigl.

[21] Kittware. *The Visualization Toolkit (VTK)*. URL: https://vtk.org/.

[22] Markus Deserno. “How to generate equidistributed points on the surface of a sphere”. In: (Sept. 2004).

[23] Florian Kirsch and Hasso-Plattner-Institute. *The CSG rendering library*. URL: http://opencsg.org/.

Code references

[Src: 1] *Code repository.* URL: https://gitlab.ciirc.cvut.cz/imitrob/imitrob_scene_representation/-/tree/BP-Thesis/src.

[Src: 2] *crow_scene/src/modules/kb/LocalFakeKnowledgeBase.cpp.* URL: https://gitlab.ciirc.cvut.cz/imitrob/imitrob_scene_representation/-/blob/BP-Thesis/src/crow_scene/src/modules/kb/LocalFakeKnowledgeBase.cpp.

[Src: 3] *crow_scene/src/modules/kb/KnowledgeBase.h.* URL: https://gitlab.ciirc.cvut.cz/imitrob/imitrob_scene_representation/-/blob/BP-Thesis/src/crow_scene/src/modules/kb/KnowledgeBase.h.

[Src: 4] *crow_scene/src/modules/kb/LocalFakeKnowledgeBase.cpp:131.* URL: https://gitlab.ciirc.cvut.cz/imitrob/imitrob_scene_representation/-/blob/BP-Thesis/src/crow_scene/src/modules/kb/LocalFakeKnowledgeBase.cpp#L131.

[Src: 5] *crow_scene/src/modules/posesuggestion/shared/VisualServoing.cpp:186.* URL: https://gitlab.ciirc.cvut.cz/imitrob/imitrob_scene_representation/-/blob/BP-Thesis/src/crow_scene/src/modules/posesuggestion/shared/VisualServoing.cpp#L186.

[Src: 6] *crow_scene/src/modules/posesuggestion/shared/VoxelConstruction.cpp:175.* URL: https://gitlab.ciirc.cvut.cz/imitrob/imitrob_scene_representation/-/blob/BP-Thesis/src/crow_scene/src/modules/posesuggestion/shared/VoxelConstruction.cpp#L175.

[Src: 7] *crow_vision_aruco/src/camera/CameraPreparation.cpp:129.* URL: https://gitlab.ciirc.cvut.cz/imitrob/imitrob_scene_representation/-/blob/BP-Thesis/src/crow_vision_aruco/src/camera/CameraPreparation.cpp#L129.

[Src: 8] *crow_scene/src/modules/posesuggestion/mesh/MeshMethod.cpp:63.* URL: https://gitlab.ciirc.cvut.cz/imitrob/imitrob_scene_representation/-/blob/BP-Thesis/src/crow_scene/src/modules/posesuggestion/mesh/MeshMethod.cpp#L63.

[Src: 9] *crow_scene/src/modules/posesuggestion/shared/VisualServoing.cpp:334.* URL: https://gitlab.ciirc.cvut.cz/imitrob/imitrob_scene_representation/-/blob/BP-Thesis/src/crow_scene/src/modules/posesuggestion/shared/VisualServoing.cpp#L334.