POLITECNICO DI MILANO

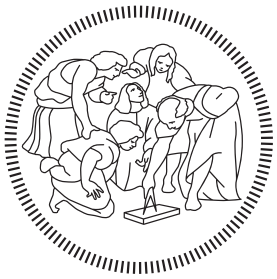
Facoltà di Ingegneria

Scuola di Ingegneria Industriale e dell'Informazione

Dipartimento di Elettronica, Informazione e Bioingegneria

Master of Science in

Computer Science and Engineering



6 Degrees of Freedom Pose Estimation with Differentiable Rendering

Advisor: PROF. MATTEO MATTEUCCI

Co-advisor: ENG. MARCO CANNICI

Master Graduation Thesis by:

MARCO ROGGERINI
Student Id n. 905651

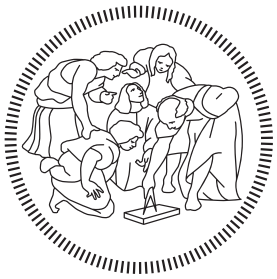
ANDREA SIMPSI
Student Id n. 905526

Academic Year 2020-2021

POLITECNICO DI MILANO

Facoltà di Ingegneria
Scuola di Ingegneria Industriale e dell'Informazione
Dipartimento di Elettronica, Informazione e Bioingegneria

Corso di Laurea Magistrale in
Computer Science and Engineering



Regressione di Posizione e Orientamento Tramite un Renderer Differenziabile

Relatore: PROF. MATTEO MATTEUCCI

Correlatore: ENG. MARCO CANNICI

Tesi di Laurea Magistrale di:

MARCO ROGGERINI
Matricola n. 905651

ANDREA SIMPSI
Matricola n. 905526

Anno Accademico 2020-2021

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

http://code.google.com/p/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/

This template has been adapted by Emanuele Mason, Andrea Cominola and Daniela Anghileri: *A template for master thesis at DEIB*, June 2015. This version of the paper has been later adapted by Marco Cannici, March 2018.

A Stefano Simpsi,
grazie di tutto

ACKNOWLEDGMENTS

The development of this thesis would not have been possible without the help and support of the kind people around us.

We would like to express our appreciation to Prof. Matteucci, both for introducing us to the fascinating world of machine learning and for its guidance in this thesis project. Many thanks also to Eng. Cannici for all the constant support and guidance in this journey. We learned a lot from your teachings.

We also would like to thank all the professors of Computer Science and Engineering for their preparation and passion. We grew a lot thanks to your teaching, both on the professional and personal level.

MARCO'S ACKNOWLEDGMENTS

My personal thanks go to all the people around me that always supported me with love. Many thanks to my family and Marta, for their constant love and support even when the path was rough. These few rows cannot show how much grateful I am to you all.

A special thank you goes to my colleague and friend Andrea: this work couldn't be possible without your constant work, dedication and all your bursting joy. I look forward to work with you again.

A big thank you goes to all my friends from Gorno: Andrea B., Andrea C., Andrea C., Francesco C., Francesco C., Lorenzo, Paolo. You always mange to put a smile on my face even in the most stressful times.

Thanks to all my friends of Politecnico di Milano: Alessandro P., Alessandro R., Alessandro V., Andrea, Fabrizio and Riccardo. You managed to make even the most boring lesson interesting, and I'm really happy of the strong bond we managed to create.

ANDREA'S ACKNOWLEDGMENTS

All this work would not have been possible without the help, love and joy received by all the people close to me.

My first thanks goes to my colleague and friend Marco: you have been one of the best people I had the honor of working with, thanks both to your patience and your incredible skill and perseverance during all this journey.

A thanks goes to my friends known at Politecnico di Milano: Andrea, Riccardo N., Marco, Alessandro Vannoni, Francesco I., Fabrizio. Thanks to all of you this experience has been one of the greatest of my entire life and I hope that our bond will continue in the future.

Another special thanks goes to my friend Alessandro Rizzo, for backing me up, helping me during my worst times and helping me always to put a smile on my face.

A big thanks goes to all my other friend from my town: Francesco R., Simona, Giulia, Alessio, Riccardo C., Gabriele, Domenico and all the other people who were close to me in all these years.

And my last, but not least, thanks goes to my whole family, your constant love and support during all these years had been measureless and is especially thanks to all of you if I'm here right now. I need a lot more lines to express all my gratitude to you.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Six Degrees of Freedom pose estimation is a crucial task in computer vision. It consists in obtaining the parameters that identify the translation and rotation of an object with respect to a system of coordinates. This task is prominent in several fields such as: robot manipulation, autonomous driving, scene reconstruction, augmented reality as well as aerospace.

Usually, the two prevailing methods used to tackle this task are: a direct regression of the object's pose from the input image, and regression of the keypoints of an object using an input image followed by a Perspective-n-Point algorithm to obtain the correct pose of the object.

These methods have shown great results in different areas, but both present some drawbacks. Usually the direct regression of a pose is done through a deep neural network which requires a lot of data to be correctly trained. Instead, the regression of key points requires costly annotation, and not many publicly available datasets provide them.

In this work we propose a new method to address this task using differentiable rendering: first, we reconstruct the 3D model of an object with a differentiable rendering technique. Then, we use this information to enrich our dataset with new images and useful annotations, and regress a first estimation of the six Degrees of Freedom. Finally, we refine this coarse pose with a render-and-compare approach using differentiable rendering.

We tested our method on ESA's Pose Estimation Challenge using the SPEED dataset. Our approach achieves competitive results both on the benchmark challenge and in enhancing the performance of existing state of the art algorithms.

SOMMARIO

La stima della posizione a sei gradi di libertà da immagini monoculari è un compito cruciale per la visione artificiale: esso consiste nell'ottenimento della traslazione e rotazione di un oggetto rispetto a un sistema di coordinate fissato, partendo da una sola immagine. Questo compito è molto impegnativo, dato che una singola immagine 2D non contiene alcuna informazione sulla profondità. Nell'ultimo anno, questo tema ha ottenuto sempre più interesse in diversi campi, come viene riportato nell'articolo di Sahin et al. [46].

Fra questi abbiamo: manipolazione robotica, dove un dispositivo automatico deve prendere degli oggetti e piazzarli in una precisa locazione, guida automatica, la quale beneficia molto dai software di stima della posizione, usati principalmente per evitare collisioni durante la guida. Un altro campo che si sta interessando molto a questo compito è quello spaziale, come mostrato nell'articolo di Opromolla et al. [20].

La stima della posizione dei veicoli spaziali è un problema che ha implicazioni in diversi compiti, come: navigazione in formazione, esplorazione di comete ed asteroidi, manutenzione in orbita e rimozione di detriti. Una dimostrazione di questo interesse è data anche dalla creazione della "Pose Estimation Challenge" della ESA [44], dove appassionati di machine learning da tutto il mondo si sono riuniti per affrontare questo problema di stima della posizione a sei gradi di libertà.

Solitamente il compito di ottenere la posizione da immagini monoculari viene espletato usando due note tipologie di approccio: o la posizione e l'orientamento dell'oggetto vengono regrediti direttamente da un'immagine data in input ad una deep neural network (da adesso in poi *regressione diretta*), oppure vengono identificati alcuni punti chiave dell'oggetto in analisi, con i quali si ricava la posizione sfruttando un algoritmo Pespective-n-Point (da adesso in poi *regressione di punti chiave*).

Questi approcci hanno ottenuto importanti risultati in diversi campi. Infatti la regressione diretta ha dimostrato di essere un metodo robusto e facile da applicare a diversi oggetti, ma solitamente risulta essere meno accurato della regressione di punti chiave. Inoltre la regressione diretta richiede un enorme quantitativo di dati. Invece, la regressione di punti chiave ottiene eccellenti risultati se consideriamo l'accuratezza della posizione estimata, ma richiede l'uso di annotazioni difficili da ottenere, come la vera posizione dei punti chiave dell'oggetto e un modello wireframe dell'oggetto da riconoscere.

Con questa tesi presentiamo un nuovo metodo per ottenere la posizione a sei gradi di liberà di un oggetto usando un renderer differenziabile, una nuova tecnologia per la manipolazione di dati 3D. Con il nostro approccio,

otteniamo un'accurata ricostruzione del modello 3D di un oggetto partendo da un piccolo insieme di immagini RGBA. Questo modello può essere usato per generare una grossa quantità di nuovi dati e annotazioni. Dopodiché regrediamo una stima grossolana della posizione dell'oggetto, che viene successivamente rifinita con una tecnica render-and-compare usando di nuovo il renderer differenziabile.

In questo lavoro, abbiamo dimostrato che questo metodo può sia generare risultati validi nella competizione indetta dalla ESA [44], sia aumentare la precisione di altri metodi che rappresentano l'attuale stato dell'arte in questo campo. Inoltre, abbiamo ricostruito un modello estremamente accurato del satellite Tango usato nella competizione [44], il quale può essere usato per generare un numero virtualmente illimitato di immagini. Una delle più grandi funzioni presentate in questo lavoro è il fatto che il metodo che presentiamo può essere utilizzato anche con altri oggetti. Infatti, mostriamo come è possibile ricostruire il modello 3D di altri oggetti, e da questa ricostruzione è possibile applicare l'intero sistema presentato, visto che essa non dipende in nessun modo dalla modello 3D in input. Il nostro lavoro è strutturato nel seguente modo:

- Nel Capitolo 2, forniamo una descrizione approfondita del compito di stima della posa a sei gradi di libertà, con annessa la descrizione degli attuali metodi dello stato dell'arte;
- Nel Capitolo 3, descriviamo alcune delle attuali librerie di renderer differenziabile;
- Nel Capitolo 4, descriviamo il nostro sistema per l'ottenimento del modello 3D, mostrandone la sua efficienza e la sua affidabilità;
- Nel Capitolo 5, analizziamo la stima della posizione a sei gradi di libertà e proponiamo un possibile metodo per risolvere il compito;
- Nel Capitolo 6, descriviamo e discutiamo il processo di perfezionamento della posizione stimata usando un renderer differenziabile e mostrandone i suoi punti di forza.
- Nel Capitolo 7, riassumiamo i nostri risultati.

1 INTRODUCTION

Six Degrees of Freedom (6DoF) pose estimation from monocular images is a crucial task in computer vision: it consists in obtaining the translation and rotation of an object with respect to a fixed system of coordinates from only one image. This task is very challenging since a single 2D image does not directly carry the depth information. In the last years, this topic has obtained a growing interest in different fields, as investigated by Sahin et al. in their review [46]. Among them, we have robotics manipulation, where an autonomous device can pick up different objects and place them in the desired target location, and autonomous driving, which benefits a lot from pose estimation to avoid collision with other objects during navigation. The space domain is another field interested in this information, as highlighted in the survey of Opromolla et al. [20]. Spacecraft pose estimation is a relevant problem in different scenarios, such as formation flying, comet and asteroid exploration, on-orbit servicing, and active debris removal. This interest is further shown by ESA's Pose Estimation Challenge [44], where machine learning enthusiasts from around the world joined together to tackle the 6DoF pose estimation problem.

Usually, pose retrieval from monocular images is carried out using two main types of approach: the pose of the object is directly regressed with a deep neural network from the image (from now on *direct regression*), or some key points of the object are detected, followed by a Perspective-n-Point algorithm on these keypoints (from now on *keypoints regression*).

These approaches achieved important results in different fields. Indeed, direct regression has shown to be robust and easy to apply to different objects, but it is usually less accurate than keypoints regression. Moreover, direct regression requires a huge amount of data. Instead, the keypoints regression obtained excellent results in the accuracy of the estimated pose but requires costly annotation, such as the ground truth position of key points of the object and the wireframe model of the object to be recognized.

With this thesis, we present a new method to obtain the 6 degrees of freedom pose of an object using differentiable rendering, a novel technology for 3D data manipulation. With our approach, we obtain an accurate 3D model of an object from a small set of RGBA images, this model then can be used to generate a huge amount of new data and annotations. Then, we regress a coarse estimation of the object's pose, which is finally refined with a render-and-compare technique using again differentiable rendering.

In this work, we prove that this method is competitive in ESA's challenge [44] as well as its ability to improve the estimation made by some of the state-of-the-art methods. Furthermore, we have reconstructed an extremely accurate model of the Tango satellite used in the challenge [44], that can be used to generate a virtually unlimited number of new images. One of the greatest features presented in this work is that this method can work also with other objects. In fact, we show that is possible to reconstruct the 3D model of other objects, and from that reconstruction is possible to apply the entire presented pipeline, since this pipeline does not depend in any way on the type of 3D model presented.

This work is structured as follows:

- In Chapter 2, an in-depth description of the 6 Degrees of Freedom pose estimation task is provided, with a description of the current state-of-the-art methods;
- In Chapter 3, some of the current Differentiable Rendering libraries are described;
- In Chapter 4 our pipeline for model retrieval is described, showing its efficiency and reliability;
- In Chapter 5, we analyzed the 6 Degrees of Freedom pose estimation problem and provided a possible approach to the task;
- In Chapter 6, the pose refinement step using Differentiable Rendering is discussed and described, showing its strengths;
- In Chapter 7, we summarize our findings.

2

STATE OF THE ART

In this chapter, the current and known methods used in this field are presented, highlighting their advantages and disadvantages. This information constitutes the basics of the method presented in this thesis.

2.1 3D COMPUTER VISION

Before talking about the state-of-the-art approaches in the field of 6DoF Pose Regression and Differentiable Rendering, we need to introduce some basic concepts of 3D Computer Vision. These concepts will allow us to better understand the task we will approach in the rest of this work.

2.1.1 *3D Coordinates*

In Computer Vision space can be represented in three ways, shown in Figure 2.1:

- Meshes: the world is composed by points connected together to form faces. Usually the faces are triangular and have an associated texture. It is a sparse representation: we only represent the position of each point and how they are connected, while we do not describe the void.
- Voxels: the world is divided in unitary portions of space (like for instance cubes). To each of these portions are associated some features, like for instance, color, material and transparency. It is a dense representation: the world is a tensor and we say what is the content of each subsection of volume (even the empty ones).
- Point Cloud: the world is composed only by points. Each point has some associated features, like for instance the position and the color. It is a sparse representation since we only describe the position of a point and its characteristics while not describing the void.

In our work we will only focus on Meshes representation, since it is the one mostly used in Computer Graphics and 3D animation. Its main advantages are that it allows to obtain photo-realistic images and the 3D models are easy to manipulate and animate.

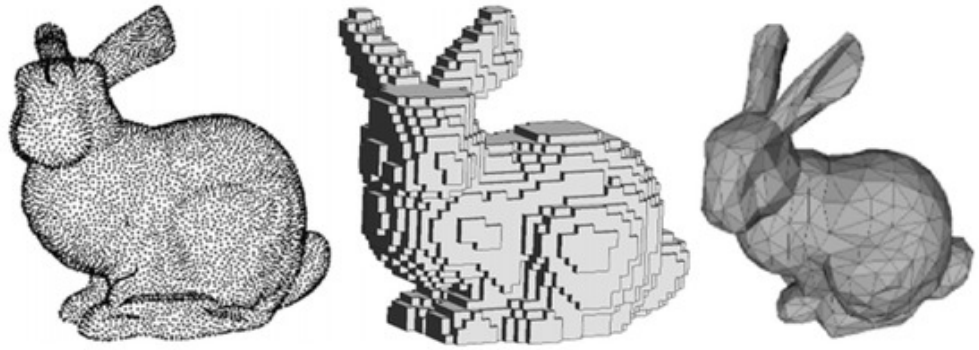


Figure 2.1: The three possible representations of an object in 3D space: Point Cloud [left], Voxel [center], triangle meshes [right]. Image from [31]

A mesh can be moved around the world with simple linear algebra operations on the vertices. We can move it rigidly around the world by summing to each point a movement vector:

$$\begin{bmatrix} x_f \\ y_f \\ z_f \end{bmatrix} = \begin{bmatrix} x_s \\ y_s \\ z_s \end{bmatrix} + \begin{bmatrix} x_m \\ y_m \\ z_m \end{bmatrix},$$

where x_f, y_f, z_f represent the final position of a vertex, x_m, y_m, z_m represent a movement and x_s, y_s, z_s represent the starting position of that vertex. A rigid rotation can be generated with a matrix multiplication:

$$\begin{bmatrix} x_f \\ y_f \\ z_f \end{bmatrix} = R_{3\times3} \cdot \begin{bmatrix} x_m \\ y_m \\ z_m \end{bmatrix},$$

where $R_{3\times3}$ is an orthonormal 3×3 matrix. A matrix is defined orthonormal when its column are perpendicular one another and they have unit norm:

$$R = \begin{bmatrix} R_1 & R_2 & R_3 \end{bmatrix}$$

$$\begin{array}{lll} \|R_1\| = 1 & \|R_2\| = 1 & \|R_3\| = 1 \\ R_1 \perp R_2 & R_2 \perp R_3 & R_1 \perp R_3 \end{array}$$

This rotation is around the point $[0, 0, 0]$.

We define the Standard View (or Object View) of a mesh as the position of a mesh before any deformation and movement. It is the position of each point as imported from a file or generated from a function. In this representation, we define the Z axis as the forward-pointing axis, the X axis as the leftward-pointing one and the Y axis as the upward-pointing one.

We define as World View the position of all the meshes with respect to the origin. In this representation, we define the Z axis as the northbound axis, the X axis as the westbound one, and Y as the zenith-pointing one.

2.1.2 *Camera Definition*

In Computer Vision, a linear camera is defined as an object that projects the 3D world on a 2D image. This is done in two steps:

1. Transform the World Space to Camera Space: in this representation the center of the coordinate system coincides with the one of the camera. This can be done with a rototranslation;
2. Transform the Camera Space in Screen Space: in this representation the points of the world (viewed from the camera perspective) are projected on a 2D space.

The camera is defined by 3 parameters. Two of these parameters are extrinsic parameters that represent the position and the orientation of the camera in the world coordinates, while the third one is an intrinsic parameter that represent how the camera projects the 3D points to a 2D image plane. These parameters can be described with 3 different matrices:

- Camera position: the extrinsic parameter of the camera that represent the position of the camera in the world space. This parameter is defined as a 3D vector, where each element represent the position of the camera on the three axes:

$$C = \begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix}.$$

- Camera rotation: the extrinsic parameter of the camera that represents how the camera is rotated with respect to the world coordinates. It is defined as:

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}, \tag{2.1}$$

  such that R is orthonormal. This matrix is called Rotation Matrix.

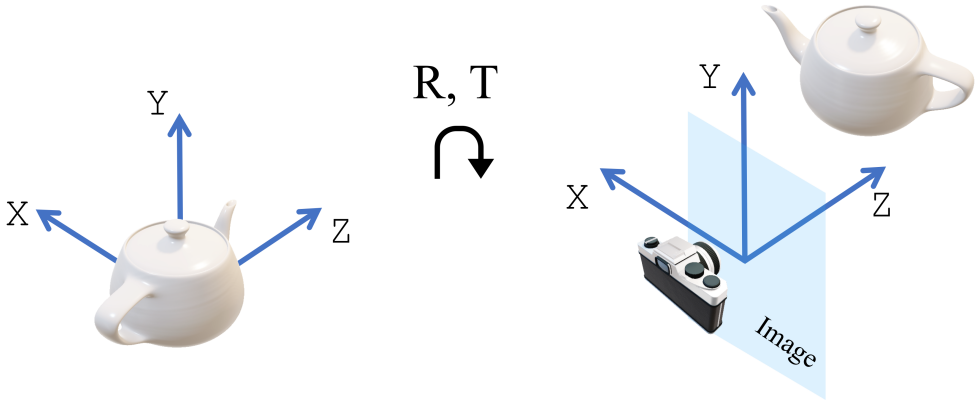


Figure 2.2: The teapot in world coordinates is transformed in camera coordinates via a rototranslation. In this example from PyTorch3D [45] the axes follow the convention +X: left, +Y: up, +Z: forward. Other systems may use different conventions

- Calibration Matrix: the intrinsic parameter of the camera that shows how the camera projects the 3D points of the world to the 2D image space. It is defined as

$$K = \begin{bmatrix} \alpha_x & \gamma & u_x & 0 \\ 0 & \alpha_y & u_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

  where $\alpha_x = f_x/p_x$ and $\alpha_y = f_y/p_y$ are the focal lengths of the camera f_x and f_y scaled by the pixel dimensions p_x, p_y, γ is the skew factor (usually 0), and u_x, u_y are the principal point coordinates.

Given these matrices, we can project a point of the world to the image with the following formula:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K \cdot \begin{bmatrix} R_{3\times3} & T_{3\times1} \\ 0_{1\times3} & 1_{1\times1} \end{bmatrix}_{4\times4} \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix} = K \cdot \begin{bmatrix} r_{11} & r_{12} & r_{13} & x_c \\ r_{21} & r_{22} & r_{23} & y_c \\ r_{31} & r_{32} & r_{33} & z_c \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix},$$

where u, v represent the points in the 2D image space and x_w, y_w, z_w represent the points in the world space.

The 4×4 matrix is composed by the extrinsic matrices of the camera, and represents a rototranslation from the world coordinates to the camera coordinates. In this representation, the camera is moved to the center of the system of reference and it is forward-facing, while the rest of the world is moved accordingly. This transformation is shown in Figure 2.2. Then,

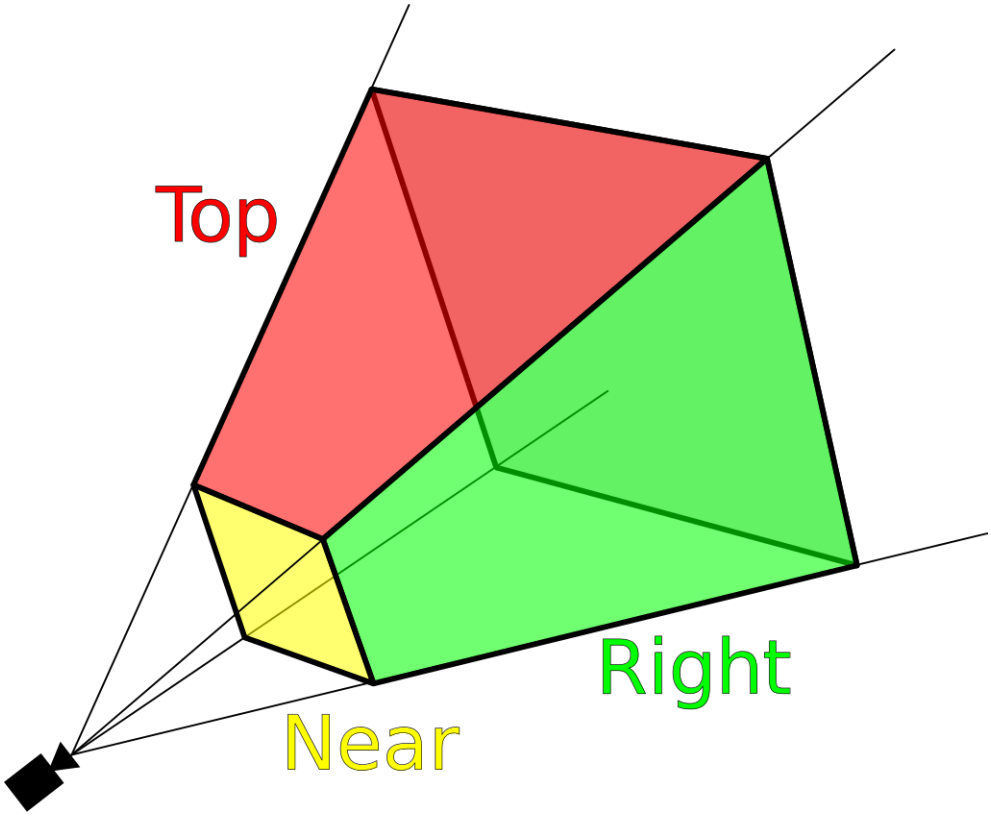


Figure 2.3: To speed up computation, only the part of the points that actually fall in the field of view of the camera is selected. Image from [48]

the calibration matrix K will transform these points in camera view to a perspective view, adjusting the position of the points in 2D given the distance. Then, the Z-coordinate is dropped: this projects the 3D space into 2D.

Usually, before projecting the 3D points on the 2D image plane, another step is taken to speed-up computation: only the portion of the camera space that will actually fall into the camera's field of view is taken. This means that the points outside the field of view of the camera and the ones too close or too far from the camera are removed, selecting only a volume of space to keep. Since the points are now much less than before, the computation is much easier and faster. This process is shown in Figure 2.3.

2.2 SIX DEGREES OF FREEDOM POSE REGRESSION

Six Degrees of Freedom Pose regression is a Computer Vision task. Its aim is to obtain the position and the orientation of an object whit respect to the camera shooting the image. This usually means that we need to find a translation T and a rotation R.

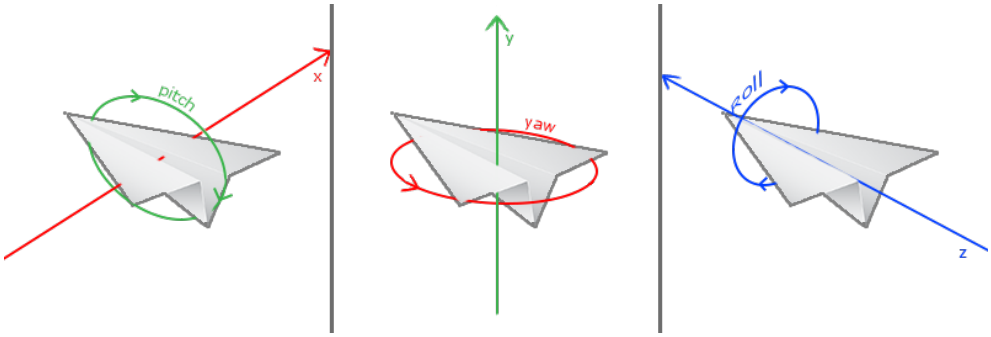


Figure 2.4: Visual representation of the three Euler angles as rotations around the three axes.

The position T is described as the distance vector between the camera and the object, ant is usually expressed as:

$$T = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

This vector represents the distance between them on the three axes.

We can represent the rotation R in many formats:

- Euler angles: three separate and independent values ψ, ϕ, θ representing the roll, pitch and yaw of an object with respect to the camera's coordinate system or the roll, pitch and yaw of the camera with respect to the object's coordinate system. These two representations are strictly related. We define the roll ψ as the rotation around the Z axis, the pitch ϕ as the rotation around the X axis and the yaw θ as the rotation around the Y axis. The main advantages of this representation are that it is very intuitive and easy to understand, and all these variables are independent one from the other. On the other hand, this representation is numerically unstable (can lead to gimbal lock, a situation where two axis of rotation are aligned and cause both an ambiguity on the representation and a loss of a degree of freedom, as shown in Figure 2.5) and has a different representation for each different order of rotations given. In fact, if we fix the magnitude of the rotations and we swap the order of the rotation, the final orientation may differ. This problem is represented in Figure 2.6, and can be solved by fixing the conventional order of rotations.

- Quaternion: a number $q \in \mathbb{R}^4$, such that:

$$q = w + ix + jy + kz, \;\; w, x, y, z \in \mathbb{R}$$
$$i^2 = j^2 = k^2 = i \cdot j \cdot k = -1 \tag{2.2}$$

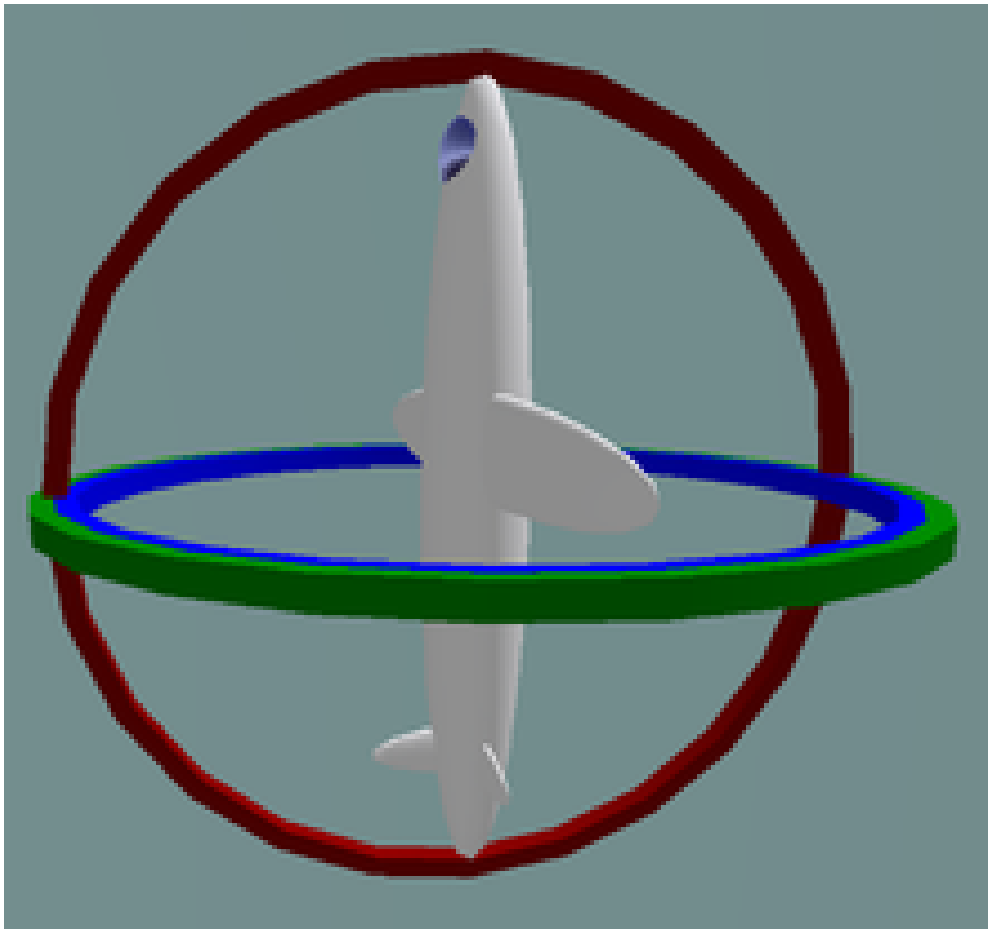


Figure 2.5: Visual representation of the gimbal lock on the Euler angles representation. In this image, the order of application of the Euler angles is $\theta \rightarrow \phi \rightarrow \psi$. As we can see, once the airplane reaches the vertical position ($\phi = 90°$), the rotation axes around Y and Z are aligned, the rotation becomes ambiguous and we suffer from a loss of degrees of freedom on the rotation. Image from [49]

$$\sqrt{w^2 + x^2 + y^2 + z^2} = 1 \tag{2.3}$$

Equation 2.2 tells us how the 4 axis of the system interact with each other. Equation 2.3 constraints $\mathbb{R}^4$ to the vectors having unitary norm. These values can be mapped to represent a rotation in any other form shown in this subsection. The quaternion representation has many advantages: it is numerically stable (no gimbal lock) and the operations between quaternions are easy to implement. Moreover, we can apply a subsequent rotation very easily using the quaternion product (which is non-commutative, as the subsequent rotations). On the other hand, the quaternion is a very unintuitive representation and its values are strictly tied together by Expression 2.3, making direct optimization harder.

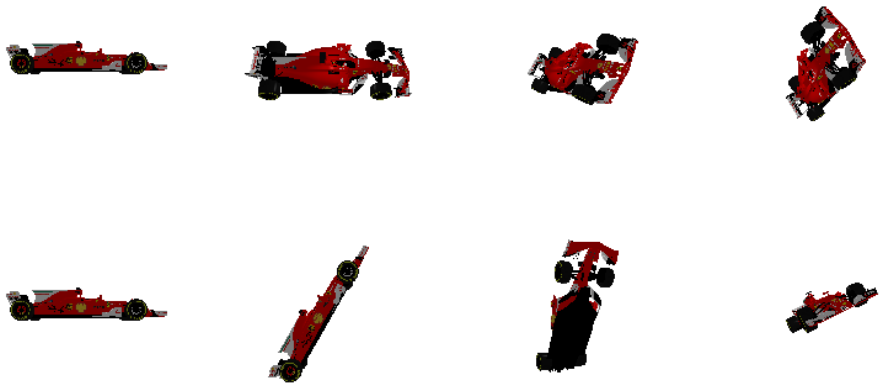


Figure 2.6: This image shows how a different order of the same rotations leads to different final orientations. In the two rows two different orders of rotation are shown: the first row the rotations are applied in the order $x \rightarrow y \rightarrow z$, while in the second row the rotations are applied in the order $z \rightarrow y \rightarrow x$. The magnitude of all the rotations is of 60°

- Rotation matrix: a rotation matrix as described in Subsection 2.1.1, representing the rotation of the object from standard view to camera view. This matrix is an orthonormal matrix. It takes into account two steps:
  1. The rotation of the object from standard view to world view;
  2. The rotation of the object from world view to camera view.

  These three rotation matrices are tied together by the following equation:

  $$R = R_{wtv} \cdot R_{stw},$$

  where R_{wtv} is the rotation matrix of the camera mapping world coordinates to view coordinates, while R_{stw} is the rotation mapping the standard coordinates to the world coordinates. Its main advantage is that it already encodes the transformation used to represent the points in view space (all other representations need to be transformed in a rotation matrix to be used for the same purpose), but since it is required to be orthonormal all its values are strictly tied together and thus very hard to optimize while preserving orthonormality. Moreover, in this optimization problem we need to optimize 9 parameters, while the other representation have 2-3 times less parameters and encode the same concept.

- Axis-Angle: the rotation of the object is represented as a unitary vector v describing the rotation axis, and a rotation magnitude θ around this axis:

$$v = \begin{bmatrix} x_v \\ y_v \\ z_v \end{bmatrix} \qquad \|v\| = 1 \qquad \theta \in (-\pi,\ \pi]$$

  This representation is numerically stable, but it is hard to optimize since the vector values are strictly tied together.

All these representations are equivalent since they all encode the same information. Moreover, you can switch from one representation to another easily.

Usually, to have a simpler process, we suppose either that the camera or the object is placed in the world in standard position, and everything else is placed accordingly. This way the unknowns are easier to determine: if the camera is in standard position, we need to find the position and the rotation of the object in the world; if the object is in standard position, we need to find the position and the rotation of the camera.

2.2.1 *Previous works on 3D pose regression*

A typical pipeline for 3D pose regression with deep neural networks is composed by 3 steps:

1. 2D object localization or segmentation;
2. Regression of a first approximation of the pose;
3. Pose refining.

Moreover, the various approaches can be categorized on the data representation used:

- Classification: instead of regressing the continuous space rotation or position, a discrete approach is taken. This has the advantage to allow easy-to-implement neural networks that exploit the strength of the deep learning approach. On the other hand, a classification leads to two different problems: firstly, the classes are not totally independent one another, but are tied together by their proximity. Moreover, a classification approach will lead to a loss of precision, since the value estimated is not a precise value, but a set of values.
- Regression: the neural network regresses directly position and rotation of the object in a continuous way;

- Keypoints: instead of directly obtain the 6DoF pose estimation from the image, some keypoints of the object are regressed, and then the position and rotation of the object is obtained from them.

Classification:
The first approach in this field is the one of Massa, Aubry, and Marlet (2014) [4]. In their approach the authors propose to recover the azimuth on the Pascal3D+ dataset [7]. It is a standard CNN pre-trained on ImageNet [11], followed by a decoder. This decoder is dependent on the experiment: in fact they proposed three different types of experiments:

1. Full classification: they classify both the object type and the predicted azimuth of the camera. To do so, they used a softmax layer that simultaneously classify the object type and the azimuth.

2. Full Regression: they regress both the continuous rotation azimuth and the object class. To do so, a linear layer is used.

3. Azimuth regression, Object type classification: this neural network has two different branches: one that classifies the object type with a softmax layer and one that regress the azimuth with a linear layer.

The work of Tulsiani and Malik (2014) [6] expands the first experiment of [4] by obtaining also the elevation and the in-plane rotation of the camera. The loss function used is the Cross Entropy function. Moreover, they use a Fully Convolutional Neural Network to identify the keypoints of the object using also the information on the pose.

One turning point for viewpoint estimation was provided by Su et al. (2015) [12]: in their paper they proved that a synthetic dataset could be used in this field to provide a more robust learning. They outperformed [6] on the benchmark of Pascal3D+ with a similar net trained on ShapeNet [8], showing the strength of this approach. Moreover, they improved the cross-entropy loss function adding a weight that penalizes predicted bins that are further from the correct one.

Regression:
The first work in 6 Degrees of Freedom pose estimation with a regression approach is the one of Kendall, Grimes, and Cipolla (2015) [10]. The main task of this neural network is to solve a relocation problem. This neural network is a simple CNN that directly and simultaneously regresses both a position vector v and a rotation quaternion q of the camera from a zero position. They used Mean Squared Error as loss function, with no normalization on the quaternion.

The approach of Mahendran, Ali, and Vidal (2017) [19] is fairly similar to the one of [6] [12], but instead of handling the problem as a classification task, they approached it used regression: their neural network outputs the

axis-angle representation of the viewpoint for each object type, joint with a classifier to determine the object depicted. Then the rotation selected is the one corresponding to the predicted object. They used both a Mean Squared Error loss and a Geodesic loss to train their architecture for rotation regression. Their result is comparable with [6] and [12].

In the work of Xiang et al. (2017) [24], the network is composed by a VGG16 [5] backbone, followed by 3 branches:

1. Semantic Labeling: each object's silhouette is recognised.

2. Translation Regression: instead of directly regressing the position of the object, the center 2D direction of each pixel and its depth is found. Then, each pixel votes for the direction in which the center is and the position with more votes is used. The pixels that voted for the actual center are considered to be the inliers of that object, and the Z center coordinate is the mean of their predicted depth. The inliers also define the 2D bounding box of the object.

3. Rotation regression: The output of the CNN backbone is pooled via RoI-pooling thanks to the 2D bounding box found in Translation branch. Than, the pooling layer is fed into a Fully Connected Neural Network that regress the 4 parameters of the rotation quaternion for each class. The loss can be calculated in two different ways:

   - The 3D model is roto-translated both by the predicted amount and the ground-truth amount. The loss is the Mean Squared Error of the distance between the corresponding points of the 3D model;
   - The 3D model is roto-translated by the predicted amount and by all the possible correct roto-translations (in the case of symmetrical objects). Then, only the closest ground-truth mesh is used, and the Mean Squared Error between the position of the vertices is back-propagated.

A scheme of this neural network can be seen in Figure 2.7

In their work, Zhou et al. (2018) [28] introduce a 6D vector representation of the rotation matrix: instead of directly regressing a 3×3 rotation matrix with the risk of regressing an invalid rotation matrix (i.e. non-orthonormal), they directly regress 6 values. The first three values are normalized against each other and are used to form the first column of the rotation matrix, while the last three values are made perpendicular with respect to the first column, and then normalized, forming the second column of the rotation matrix. The third column is the cross-product of the first two columns. This allows to regress a continuous function instead of a discontinuous one: as the authors show in their work, the Euler angles representation has three different discontinuities at $180°$, while the quaternion/axis-angle representation has one discontinuity

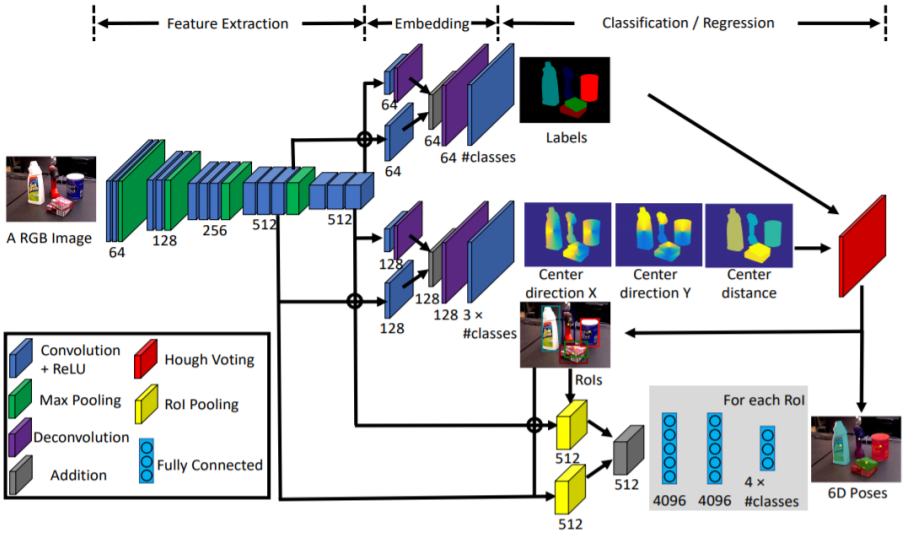


Figure 2.7: The scheme of PoseCNN [24]

at 180°. This representation on the other hand is continuous, and thus easier to regress for a neural network.

Keypoints:

The work of Pavlakos et al. (2017) [21] is different: they built a Fully Convolutional Neural Network to predict only the keypoints of the object and used the EPnP algorithm [1] on these points, joined with a deformable part CAD model. They minimize the L2 loss of the heatmap of the predicted keypoints, whose ground truth is a 2D Gaussian centered in the keypoint with variance 1. There is no direct prediction of viewpoint, which is only inferred trough the EPnP algorithm.

The approach of Rad and Lepetit (2017) [23] consists in a first step of object localization using a coarse-to-fine segmentation. Then, the 2D position of the vertices of the 3D bounding box is regressed. Finally, the camera pose is obtained from these point using EPnP algorithm [1]. The error function used is the mean squared error on the position of the projection in 2D of the 3D bounding box points. Rad and Lepetit also handle the case of the symmetric objects: they restrict the prediction in symmetric objects to a subset of values, then they use a classifier to tell if the object is outside this range. In the latter case, they mirror the image and adjust the output angle accordingly. Moreover, they introduce a rendering approach to refine the pose: given the predicted position, they render a silhouette or RGB image of the object in the predicted position. This rendered image is fed with the original image to another neural network, that predicts an update of the predicted pose.

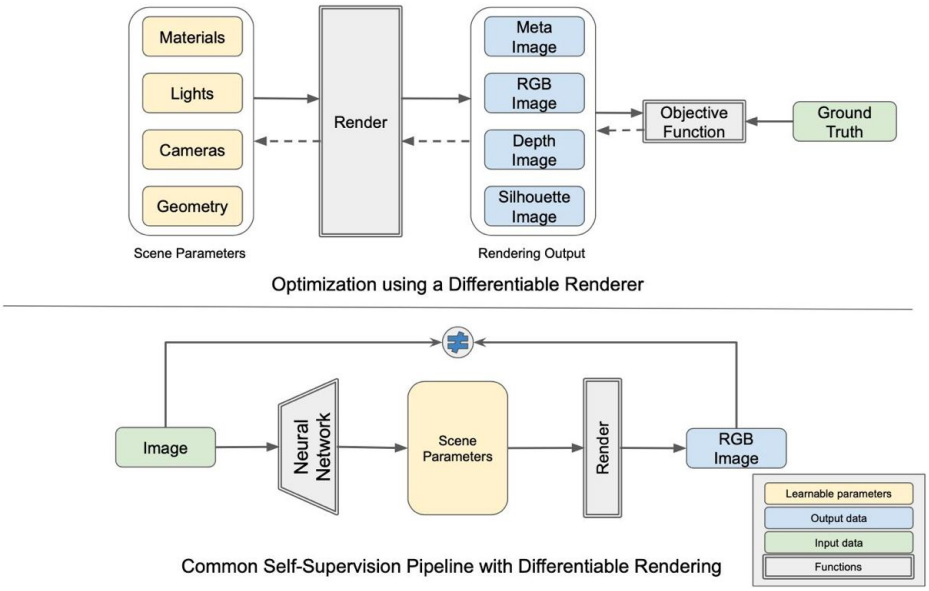


Figure 2.8: An example of two common pipeline involving differentiable renderer; a simple optimization and one self-supervised neural network. This scheme come from Kato et al. [42]

2.3 DIFFERENTIABLE RENDERING

2.3.1 *Introduction*

In the last years, the effectiveness of 3D estimation methods that rely on supervised training using deep learning has been proven by different works. All of these methods have proved to require costly annotations, hence Differentiable Rendering had seen a growing usage and interest since it can be used to obtain a great number of data with different annotations. Nevertheless Differentiable rendering methods are not easy to apply, reason why there is a plethora of them. Therefore to use the right method it is required a strong knowledge of the various existing methods, which proprieties of the differentiable renderer are required to the method, which type of data can be usable, etc. Moreover different libraries that support Differentiable Rendering methods have been developed and each one support different type of functionalities, or have different computational requirement constraints.

2.3.2 *Definitions*

Before dealing with the topic let's define some of the basic concepts that we will use through this thesis:

- Rendering: is the process of generating images of 3D scenes defined by geometry, materials, scene lights, and camera properties.
- Differentiable rendering: it represents an ensemble of techniques that tackle the integration of a rendering for end-to-end optimization, by obtaining useful gradients of the rendering process.
- Rendering function: Is a function $r(\cdot)$ that takes as input the parameters that identifies shape S, camera C, material$M$ and lighting L to give as an output an image I:

$$I = r(S, C, M, L)$$

2.3.3 *Method of approximation*

One of the main discriminators among methods is the type of data representation (as seen in Figure 2.1). The work presented in this thesis mainly focuses on Meshes representation, so here are defined some of the most famous methods using mesh as input representation. Nowadays there are different methods that allow to obtain a differentiable rendering pipeline using mesh as input. Among them, there are two categories that obtained great results. These two categories can be distinguished from which part of the rendering pipeline they approximate: the first family approximates the backward pass of the rendering process (approximated gradients), while the second family approximates the forward pass (approximated rendering).

2.3.4 *Approximated gradients*

Loper and Black [3] (2014) developed the first general-purpose differentiable renderer, called OpenDR. Indeed, until then, no one developed a DR since the rendering process was not differentiable. Loper and Black state that all the renderings do an approximation of some kind, so doing an approximation to make a renderer differentiable could be really useful and this approximation does not imply a loss in the accuracy of the renderer. However, since OpenDR was created to be a differentiable renderer of general-purpose, it created different problems for the use of differentiable renderer with neural networks.

To solve these problems Kato, Ushiku, and Harada [18] (2017) propose a differentiable renderer where the gradient that was computed by the renderer was specifically designed for neural networks. The differentiable renderer create by them was called Neural 3D Mesh Renderer, or shortly N3MR. Kato, Ushiku, and Harada illustrate the problem given by the rasterization process during the standard rendering pipeline, how we can see from Figure 2.9. The color I_j of the pixel P_j with respect to the movement of x_i is a step function

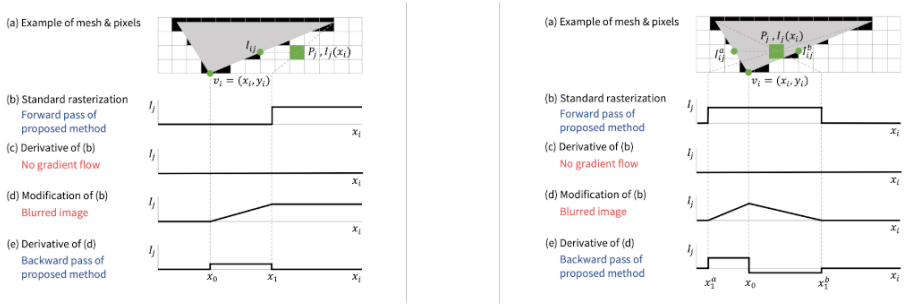


Figure 2.9: Effect of rasterizzation on a standard image or on a blurred image. This image comes from [18]

because the operation to compute the color is discrete. So, in N3MR this discrete process was replaced with a gradual change, as we can see in point d of the Figure 2.9. This is obtained replacing

$$\frac{\partial I_j}{\partial x_i} \tag{2.4}$$

with

$$\frac{\delta_j^I}{\delta_i^x} \tag{2.5}$$

Where δ_j^I is the difference in color between the color post-movement and pre-movement, like this:

$$\delta_j^I = I(x_1) - I(x_0) \tag{2.6}$$

Similarly δ_i^x is defined in this way:

$$\delta_i^x = x_1 - x_0 \tag{2.7}$$

2.3.5 *Approximated rendering*

Rhodin et al. [16] (2016) tried a different solution: approximate the rasterizzation process. Their work follows this idea: represent opaque objects through a translucent effect that gives a blurred effect to the image. Specifically, each object has a density parameter where the maximum value is at the center of the object and, follow a smooth Gaussian distribution, loses density moving away from the center. The effect of this approach can be seen in the Figure 2.10. Thanks to this approximation, we can remove the sharp edges and corners from the figure ensuring differentiability.

Liu et al. [34] (2019) expanded the previous idea creating the framework called Soft Rasterizer. This framework's differentiable renderer applies a blur to the image in the rasterization process, like Rhodin et al. In addition, they create a probabilistic system where each triangle projected into the pixel can

Figure 2.10: An example of the blurred effect on a image, This image comes from [16]

contribute to the pixel's color. In this way, the color of the pixel is obtained by a weighted sum of the color of each relevant triangle for the pixel.

Chen et al. [30] (2019) proposed instead a totally different approach to approximate the rasterization process with their DIB-R. Their idea is to deal independently with foreground pixel and background pixel. The foreground pixels are determined only by one face with a weighted interpolation of his local proprieties, while the background pixels are a distance-related function of global geometry. For the value of the foreground pixel I_i we compute a barycentric interpolation using the value of the face's vertex attributes (the face f_i is the one that contains the pixel) as follow:

$$I_i = w_0 u_0 + w_1 u_1 + w_2 u_2 \tag{2.8}$$

For the value of the background pixel $p_{i'}$, the ones that aren't covered by any faces, they compute a distance-related probability $A_{i'}^j$ that assign each face f_j to $p_{i'}$. Then they combine the probabilistic influence of each face in the following way:

$$A_{i'} = 1 - \prod_{j=1}^{n} (1 - A_{i'}^j) \tag{2.9}$$

In Figure 2.11 we can see this concept of foreground and background pixels.

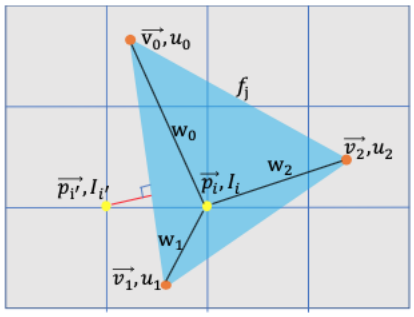


Figure 2.11: In this figure we can see a foreground pixel and a background pixel and the various attribute used to compute their value. This image comes from Chen et al. [30]

3 DIFFERENTIABLE RENDERING LIBRARIES

The first part of this thesis involved the evaluation and selection of the differentiable rendering library that better suits the requirements of this work. We recall that here, with a differentiable rendering library, we mean a library that implements a renderer (i.e., a function producing an output image given the environment definition), and it also provides the derivative of the image with reference to the end parameters (i.e., the gradients). Among the few libraries publicly available, we tested the two of them based on PyTorch's deep learning framework: Kaolin and PyTorch3D.

3.1 KAOLIN

Kaolin is a library developed by the research team of NVIDIA [32], and released at the end of 2019 that is currently in beta version. Kaolin falls within the category of differentiable rendering that approximated the rendering process. Specifically, this is obtained through the approximation of the rasterization process. The library is powerful and versatile and it has been developed to support the following specification:

- Support for different data representation of a 3D model, including:
  - triangle mesh,
  - quadratic mesh,
  - point cloud,
  - Voxel grid,
  - Signed Distance Function (SDF)
- Use of the original implementation for some famous differentiable rendering algorithm with approximated rasterization, namely:
  - Neural 3D Mesh Renderer [18]
  - Soft Rasterizer [34]
  - DIB-Renderer [30]
- Support for a huge collection of different state of the art architectures that exploit differentiable rendering, including:
  - Dib-r [30]
  - PointNet [22]

- PointNet++ [22]
- GResNet [41]
- 3D-GAN [17]
- Pixel2Mesh [27]
- GEOMetrics [39]
- Occupancy Networks [36]

3.2 PYTORCH3D

PyTorch3D [45] is a differentiable rendering library developed by Facebook Research. It provides support to operations both on 3D triangle meshes, point cloud, and voxels. In this work, only 3D triangle mesh representation will be discussed, since it is the representation used in all the experiments depicted here.

PyTorch3D's mesh differentiable renderer is based on the one described in [35], and it is a rendering process that approximates the rendering function. In this library, the renderer has been re-implemented, allowing a two-step rendering:

1. The first step is the rasterization step: the meshes are transformed from world view to camera view and projected onto the image plane. Each face is expanded via a sigmoid blur: this allows the rendering process to be differentiable, removing the step introduced on each edge by standard rasterization. Moreover, each pixel of the final image is assigned with more than one face, allowing the blending with more faces;

2. The second step is the shading step: the information of the renderization step is blended together and lighting is applied. This allows the generation of the final 2D image.

Splitting the rendering process has the main advantage that any user can rewrite one of the two steps (if not both), thus the rendering process is easier to modify than with a monolithic architecture.

PyTorch3D's built-in rasterizer is written both in CUDA, C++ and Python: this allows to use the renderer also on CPU-only, even if the rendering process is slow. On the other hand the many built-in shaders use standard PyTorch operations: this allows the use of the full renderer both on GPU and on CPU. The built-in PyTorch3D's shaders are:

- Soft Phong Shader, that applies Phong shading in a differentiable way (blending all the faces on the proposed Z-buffer together, instead of the first one only);

- Hard Phong Shader, that applies Phong shading in a non differentiable way;
- Soft Gouraud Shader, that applies a differentiable version of the Gouraud shading process;
- Hard Gouraud Shader, that applies a non-differentiable version of the Gouraud shading process;
- Soft Silhouette Shader, that only renders the silhouette of the 3D model, without applying lightning.

The non-differentiable shaders shown here allows the renderer to be used as a classic renderer, and to generate true rendered images.

In PyTorch3D all the main 3D supervision losses are implemented:

- Chamfer distance;
- Mesh Normal Consistency;
- Mesh Laplacian Loss;
- Mesh Edge Loss.

Moreover, it supports both the use of perspective cameras and orthographic cameras, that are described with a right-handed convention with the X-axis pointing to the left of the direction the camera is facing, the Y-axis pointing above the camera and the Z-axis pointing the same direction the camera is facing. The library supports both point lights and directional lights.

PyTorch3D allows to have an heterogeneous batching of meshes: in fact it support the parallel renderization of many different 3D models in the same forward step. This is shown in Figure 3.1, where four different 3D models are rendered at the same time. This is made possible by the joint work of many different representations of vertices, faces and textures inside the wrapper Meshes.

Finally, this library comes with a built-in dataloader for two classical 3D datasets (ShapeNet [8] and R2N2 [13]), functions to load and store 3D objects, many tutorials to learn how to use it and a well documented GitHub. This library is now in Beta version 0.4.0.

3.3 TEST OF DIFFERENTIABLE RENDERING LIBRARIES

First, we focused on replicating, within the Kaolin and PyTorch3D frameworks, one key experiment presented in the DIB-R paper [30], which involves learning to extract 3D models from 2D images of objects. The ability of performing this task reliably is of particular importance in the work presented in

Figure 3.1: Example of heterogeneous rendering. The four images are from 4 different models with a different number of vertices and faces, and represented with different translations and rotations. The number of vertices and faces are, from left to right: 2057 vertices, 8076 faces; 45099 vertices, 169222 faces; 12543 vertices, 46434 faces; 3914 vertices, 13964 faces. These models have been rendered with PyTorch3D's renderer using Hard Phong Shader. 3D models and textures from ShapeNet [8]

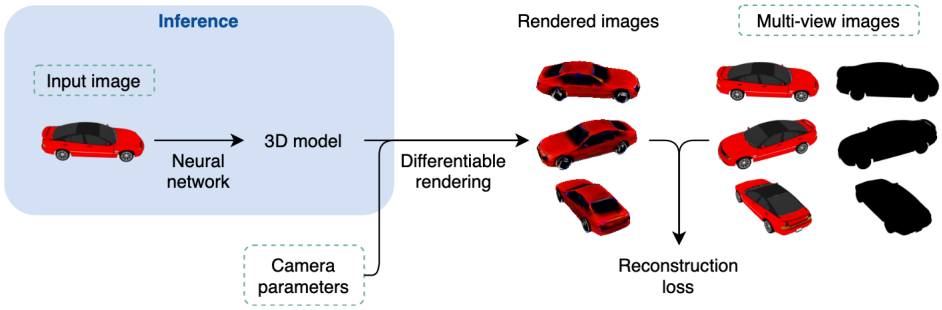


Figure 3.2: The pipeline proposed by [30] was used to obtain the 3D model from a single 2D image. The image was taken by Kato et al. [42]

this thesis and it is thus a requirement for the differentiable rendering library. Kaolin already comes with a set of pre-trained neural networks, also called model zoo, which makes prototyping easy.

This experiment, already documented in the paper of Chen et al., takes as input a dataset composed of 24 images generated from various 3D models. All of these models come from the ShapeNet dataset [8], specifically from 13 categories: Airplane, Bench, Dresser, Car, Chair, Display, Lamp, Speaker, Rifle, Sofa, Table, Phone, Vessel. For each object that belongs to one of these categories, we rendered 24 images using Blender [26], where each image was taken from 24 different azimuth angles (one every 15 degrees) but with the same elevation angle, camera, and illumination parameters; each image has a size of 64×64 pixels. We split the dataset following the same procedure used by Yan et al. [25]: the training set take always the first 5 images of each model as input image. This image is paired with one other view out of the 19 left, picked randomly, which is not input in the neural network but only used in loss calculation. This allows the training procedure to have two different views of the same object to learn better the 3D shape of it.

Category	Train	Validation
Airplane	15371	809
Bench	6900	364
Dresser	5973	315
Car	28484	1500
Chair	25765	1356
Display	4161	219
Lamp	8808	464
Speaker	6148	324
Rifle	9013	475
Sofa	12057	635
Table	32334	1702
Phone	3997	211
Vessel	7368	388

Table 3.1: Number of images for each split of the dataset.

The architecture presented by Chen et al. which is the same presented by Kato, Ushiku, and Harada [18] Liu et al. [34], consists of an encoder-decoder structure that takes as input a 2D image, and outputs the position and color of each vertex of a sphere. The network learns to deform the sphere into the target 3D object that is contained in the image given as input. For the encoder, we use 3 convolutional blocks followed by 3 linear blocks. A convolutional block is composed by a convolutional layer with kernel size of 5, stride of 2 and respectively 64/128/256 channels, followed by a Batch Normalization layer and a max pooling layer. Instead, each linear block is composed by a fully-connected layer with 1024 neurons, a Batch Normalization layer (aside from the last layer), and a ReLU activation function.

The decoder part of the network consists of two independent branches, the spatial decoder and the color decoder, each composed of three fully connected layers with 1024/2048/1920 neurons. The output of the spatial decoder represents the 3 spatial coordinates of each vertex of the 3d sphere, while the output of the color decoder provides the RGB color of each vertex.

From the outputs of the decoder, we generate a 3D model, which is given in input to the differentiable renderer with camera parameters. This step will generate the 2D image to compare to the input one. The differentiable renderers used in the two experiments are:

- Kaolin: the renderer is configured in "Vertex Color", one of the modes available for the renderer. The output obtained is visible in Figure 3.3. The renderer outputs a 64×64 RGB image.
- PyTorch3D: The renderer used is configured as follows:
  - Camera: perspective camera with FoV of 60°;
  - Output image size: 64×64;
  - Blur radius: 1×10^{-4};
  - Z-buffer (faces per pixel): 50;
  - cull back side of the faces;
  - no perspective correction;
  - shading: Soft Phong Shading, as described in [45].

We render two 2D images from the 3D model predicted, one using the same position of the input image of our network and one using the annotations on the pose of the second image described before. In this way, we can compute the loss function using both the 3D model and 2D rendering. Using a second image that was not seen by the network we force the neural network to try to obtain a more complex 3D model: if this second image is not used, the neural network will reduce the predicted shape to a flat model.

Following 3.5, we use a loss composed of 4 terms to train the network, which are:

- Color loss (Equation 4.2)
- IoU loss (Equation 4.1)
- Smoothness loss
- Laplacian loss

The results obtained are not at the same level as the paper, but proved themselves to be able to obtain a good 3D model from the input images. In fact, as we can see in the Figure 3.3 the result obtained was visually good and similar to the real object.

3.4 REFLECTIONS

After testing both the libraries we decided to adopt PyTorch3D in our experiments. This decision is not driven by the result of our experiment, since both libraries had provided optimal results, and neither for the functionalities of the renderer, since both work flawlessly. This choice was dictated mainly by the fact that the code presented by Kaolin for the replica of the model present

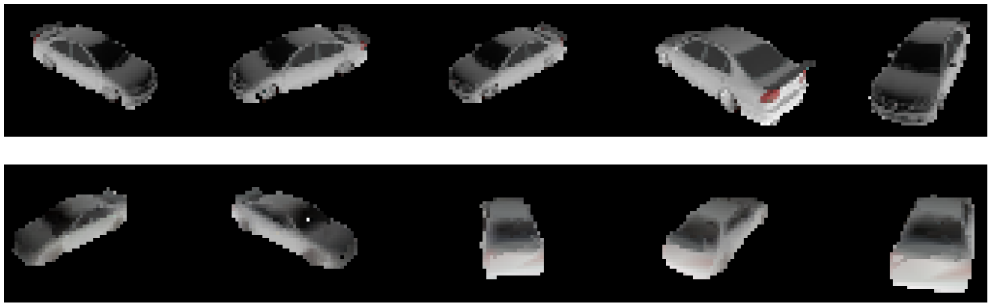


Figure 3.3: First row: input image of the network describe in Chapter 3.2. Second row: rendered image from the 3D model obtained by the input image in the first row, but with a different pose

in their model zoo was not functioning correctly. Indeed, with the latest version of the dependencies (version 0.9), they have completely deleted several of the different features shown in section 3.1, including the one provided by the different model zoo. So, since the main strength of Kaolin was not available we have chosen PyTorch3D, because:

- it was easier to use, thanks to their accurate and complete documentation
- it's core functionalities, like for instance their class use to handle the data representations or their differentiable renderer was flexible, easy to use and modify

3.5 LOSSES

3.5.1 *Smoothness Loss*

The smoothness loss used in this test is the same used also by [30] [18] [34]. As the name suggests, the goal of this is to encourage the network to create a smooth surface, reducing the number of predicted corners and giving a more natural appearance.

Let E be the set of all the edges in the 3D model and θ_i be the angle that exists between two adjacent faces sharing the edge e_i, then the smoothness loss is defined as:

$$L_{sm} = \sum_{e_i \in E} (\cos(\theta_i) + 1)^2 \tag{3.1}$$

As we can see in Figure 3.4, this loss helps the whole 3D model to remove the edges in excess. But if we give too much weight to this loss or we do not counterbalance it with another loss, we may end with a model that loses its main feature and tend to become too smooth.

Figure 3.4: In this figure from the paper Kato, Ushiku, and Harada [18] is shown the effect of the smoothness loss on the reconstruction of a CTR monitor. Left: target image, middle: model predicted without smoothness loss, right: model predicted with smoothness loss

3.5.2 *Laplacian Loss*

This loss follows the same design of [30] [27]. It promotes neighbour vertex to move consistently with each other. In this way, we constraint the vertices to not move potentially inside the mesh itself.

For each vertex v that belongs to the 3D model, let $\mathcal{N}(v)$ be the set of the neighbour vertices of v and δ_v the predicted translation of the vertex v, then the Laplacian loss is defined as follows:

$$L_{lap} = \left(\delta_v - \frac{1}{|\mathcal{N}(v)|} \sum_{v \in \mathcal{N}(v)} \delta_{v'} \right)^2 \tag{3.2}$$

3.5.3 *Edge Loss*

This regularization loss has been defined in [27]. Its goal is to minimize the average length of the edges. This way, the final mesh is smoother and has fewer imperfections. This also allows for less interpenetration of the meshes and fewer outlier vertices.

Let V be the set of all vertices of a 3D model and $\mathcal{N}(v)$ the set of all adjacent vertices of v. Then, the edge loss is defined as follows::

$$L_{edge} = \sum_{v \in V} \sum_{k \in \mathcal{N}(v)} \|v - k\|_2^2 \tag{3.3}$$

4

3D MODEL RECONSTRUCTION VIA DIFFERENTIABLE RENDERING

The 3D model is the starting point for all the pipelines of rendering: without the information on the 3D model we cannot render any image. Until few years ago, the gap between the 2D image and the 3D world was wider: a 2D image could be obtained from a 3D object, but the opposite transformation was very hard to achieve. On the other hand, the development of differentiable rendering techniques allowed to bridge the gap: the transformation from 3D object to 2D image now can be done in a differentiable way. This allows us to use the 2D information (joint with the knowledge of the position and rotation of the object) to obtain a 3D model that can be then employed in many different techniques. This pipeline is very simple and efficient, and allows us to obtain a good approximation of the true shape of the 3D object shown in the images.

4.1 MODEL STRUCTURE

The goal of the neural network described in this section is to regress the position and color of the vertices of a 3D object. The starting model is an icosphere with 10242 vertices and 20480 faces. We choose to optimize on an icosphere since it has central symmetry and the vertices are equally distributed on the surface. Moreover, PyTorch3D has a built-in function that automatically creates it. This icosphere can be pre-processed so that its initialization is more similar to the object to obtain, thus speeding up the process. We represent both the vertices offset from this sphere and their color as a tensor of size `(#vertices, 3)`. The offset tensor is initialized to `0`, while the color tensor is initialized to `0.9`.

The camera is set up to be a Perspective Camera. The Field of View of this camera must be calculated from the calibration values of the camera used to take the images that will be used in the process. The FoV must be regulated correctly, since a different value will put the the object in a different position with respect to the one in the ground truth image, making the rendered image different from the source one.

The light used for rendering is a directional light. The direction of the light is image-dependent: we process the image via a neural network to get an approximation of the light direction. This neural network is trained jointly to the 3D model prediction, and it is composed by a ResNet50 [9] backbone pre-trained on ImageNet [11], followed by one fully connected layer with

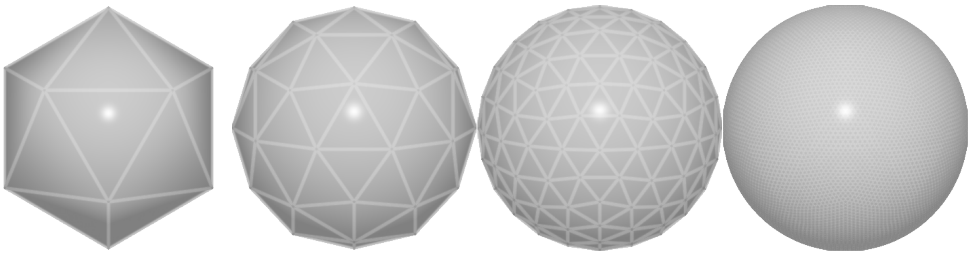


Figure 4.1: Icospheres with different numbers of vertices and faces. The leftmost is the base one, while the others are generated from that one by subdividing each face in 4 other faces. From left to right: 12 vertices and 20 faces; 42 vertices and 80 faces; 162 vertices and 320 faces; 10242 vertices and 20480 faces.

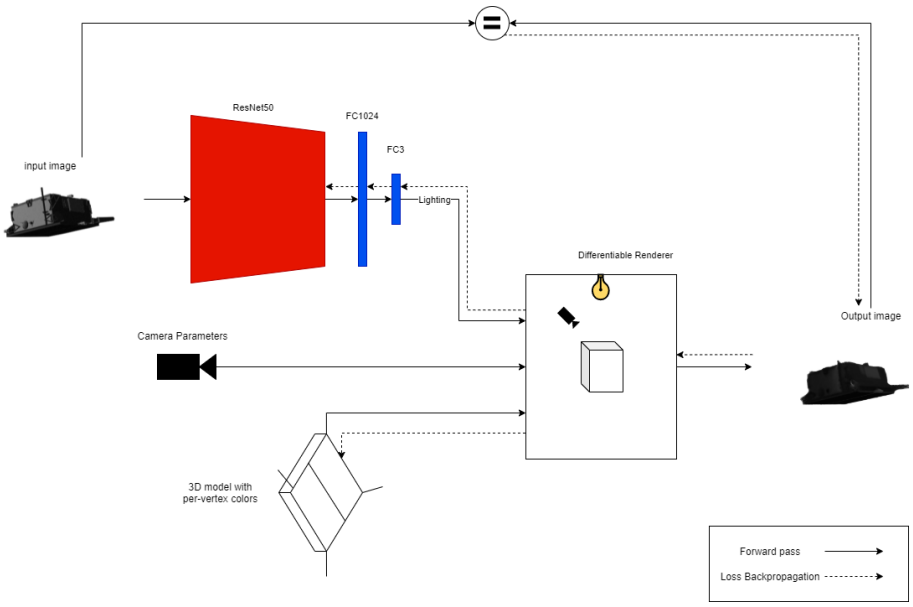


Figure 4.2: The scheme of the neural network used.

1024 neurons and ReLU activation, and an output fully connected layer with 3 neurons and linear activation. The l are fixed to bight's colors are fixed to be:

- Ambient color: `(0.4, 0.4, 0.4)`
- Diffuse color: `(0.6, 0.6, 0.6)`
- Specular color: `(0.0, 0.0, 0.0)`

We then process all this information via our Differentiable Renderer. The renderer used is configured as follows:

- Output image size: 384×384, then cropped to 384×240;
- Blur radius: 1×10^{-4};

- Z-buffer (faces per pixel): 100;
- Cull back side of the faces;
- No perspective correction;
- Shading: Soft Phong Shading, as described in [45].

This branch of the net takes as input:

- The current 3D model;
- The position of the camera T in the form of a 3-element vector;
- The rotation of the camera R in the form of a 3×3 rotation matrix;
- The lighting direction predicted from the neural net.

This branch outputs an RGBA image that can be confronted with the input image.

The loss used to regress the object's shape and color is:

$$L_{tot} = L_{IoU} + \lambda_{col} L_{col} + \lambda_{edge} L_{edge},$$

where L_{IoU} is the Intersection Over Union (IoU) loss, L_{col} the Color loss and L_{edge} the Edge Length Regularization loss (described in Subsection 3.5.3).

The Intersection Over Union loss L_{IoU} was proposed in [15] and is defined as:

$$L_{IoU} = 1 - \frac{\sum_{v \in V} S_1(v) \cdot S_2(v)}{\sum_{v \in V} S_1(v) + S_2(v) - S_1(v) \cdot S_2(v)}, \tag{4.1}$$

where $V = \{1, ..., N\}$ is the set of all pixels in the silhouettes, and S_1 and S_2 are the two silhouettes we want to compare. It is a differentiable version of the Intersection over Union metric used in segmentation tasks. This metric tells us, given two silhouettes, how much of the first one overlaps the second one (intersection of the two silhouettes), with respect to the total image covered by at least one of the two silhouettes (union of the two). Since intersection and union cannot be calculated in a differentiable way, we calculate an approximation of these values:

- The Intersection is calculated as the product pixel-wise between the two silhouettes. These values are then summed together.
- The Union is calculated as the sum pixel-wise between the two silhouettes. Then, the intersection is subtracted to eliminate the overlapping of the two silhouettes, that otherwise will be counted twice.

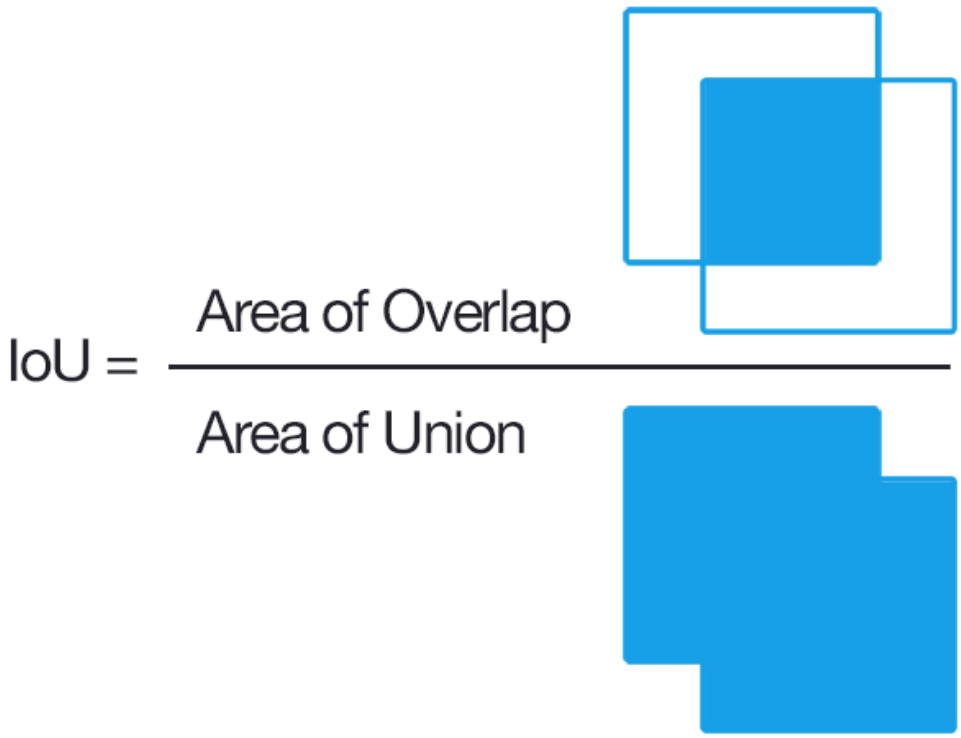


Figure 4.3: Visual representation of how the Intersection over Union loss is calculated. Image from [14]

To get the IoU score, we then divide the intersection by the union. Since we want to maximize this value, while we want to minimize the loss, we subtract this value to 1, to then minimize the output value. This loss provides a very strong learning signal, it is very robust and also is very easy to understand.

The color loss $\mathcal{L}_{col}$ is defined as the Mean Absolute Error between each channel of the two images:

$$L_{col} = \frac{1}{|C|} \sum_{c \in C} \frac{1}{|P|} \sum_{p \in P} |I_1(c,p) - I_2(c,p)| \tag{4.2}$$

where C is the set of the color channels of the image, P is the set of all pixels of the image and I_1, I_2 are the two images to compare. The main advantage of Mean Absolute Error is that it provides a good learning signal also on errors smaller than 1, thus accelerating the learning in this setting, since each channel's pixel's value is scaled to be in $[0, 1]$.

In these experiments, the values used are $\lambda_{col} = 1$ and $\lambda_{edge} = 100$. The Laplacian regularization loss (Subsection 3.5.2) and smoothness regularization loss (Subsection 3.5.1) are not used in this experiment since they tend to round the edges of the predicted mesh a lot. Moreover, the edge loss proved itself to be enough to smooth the mesh's surface.

Figure 4.4: Image showing the two spacecrafts of the PRISMA mission: Tango (left) and Mango (right). Image from [44].

4.2 SPEED DATASET DESCRIPTION

To perform this task, we used the SPEED dataset [38]. This dataset is provided by ESA for its Pose Estimation Challenge [44], a competition held on their Kelvin website that aims to obtain a 6 Degrees of Freedom pose estimation of the Tango satellite. This Tango satellite is one module of ESA's PRISMA mission [51], launched in 2010. The goal of the PRISMA mission was to demonstrate some sensing and navigation techniques in low Earth orbit: to do so, the mission was launched as a two-satellite mission: one satellite (Main or Mango) is highly maneuverable and has a lot of sensors, while the other (Target or Tango) is a very simple satellite that can only follow its orbit and at most stabilize it (to avoid spinning out of control). Mango's goal was to perform many maneuver around Tango, both in close and in long range.

This dataset is based on this mission's concept: since a camera is a very simple and lightweight sensor, it can be integrated easily in a satellite without impacting too much on the payload of the launch. In this dataset, on the other hand, no image from the actual mission is used. This dataset is composed by in 4 different subsets:

- `train` dataset, composed of 12000 rendered images with annotation on the pose of the satellite;
- `test` dataset, composed of 2998 rendered images without annotations on the pose;

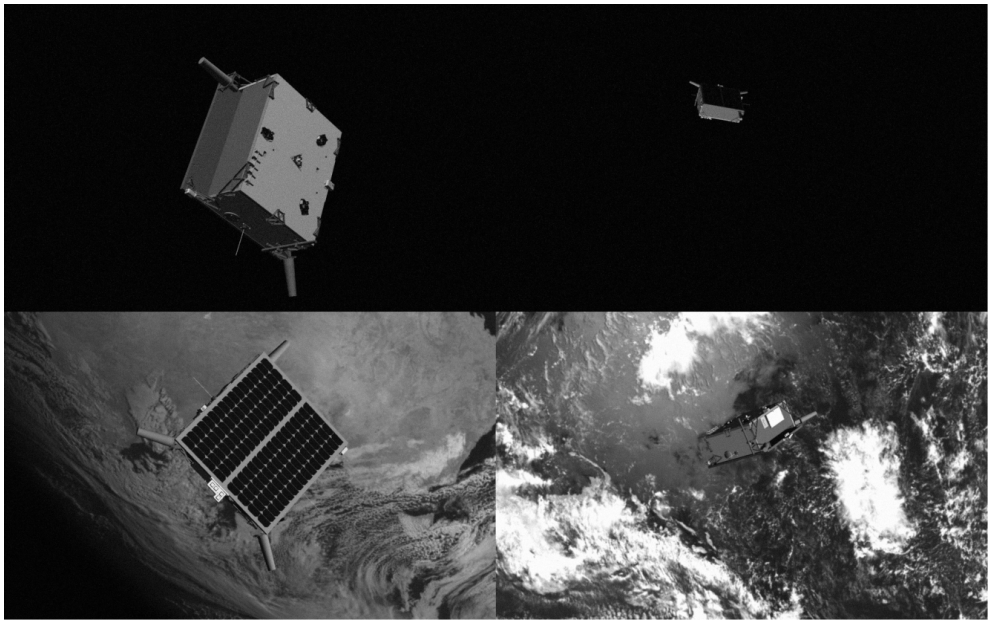


Figure 4.5: Some samples of the images of SPEED's rendered datasets.

- `real` dataset, composed of 5 real images of the satellite with the annotations on the pose;
- `real_test` dataset, composed of 300 real images of the satellite without annotations on the pose.

All the images provided are in black and white (B/W). The annotations on the pose are composed of a translation vector representing the position of the satellite with respect to the camera and a quaternion representing the rotation of the satellite with respect to the camera. The real dataset was created by snapping photos to a replica of the satellite in a studio, while the rendered dataset was generated from a 3D model of the satellite. While the real datasets have only images with a black background, the rendered dataset has both images with a black background and images with an earth background. This background was taken from the real images shot by Himawari 8 mission [50].

The camera used both for the real images and the rendered images has the parameters shown in Table 4.1. From these values we can calculate the Field of View (FOV) of the camera (in radians [**rad**]) as:

$$\mathrm{FOV}_u = 2 \arctan \frac{N_u \cdot d_u}{2 f_x} \tag{4.3}$$

$$\mathrm{FOV}_v = 2 \arctan \frac{N_v \cdot d_v}{2 f_y}. \tag{4.4}$$

All the elements of this formula are described in Table 4.1

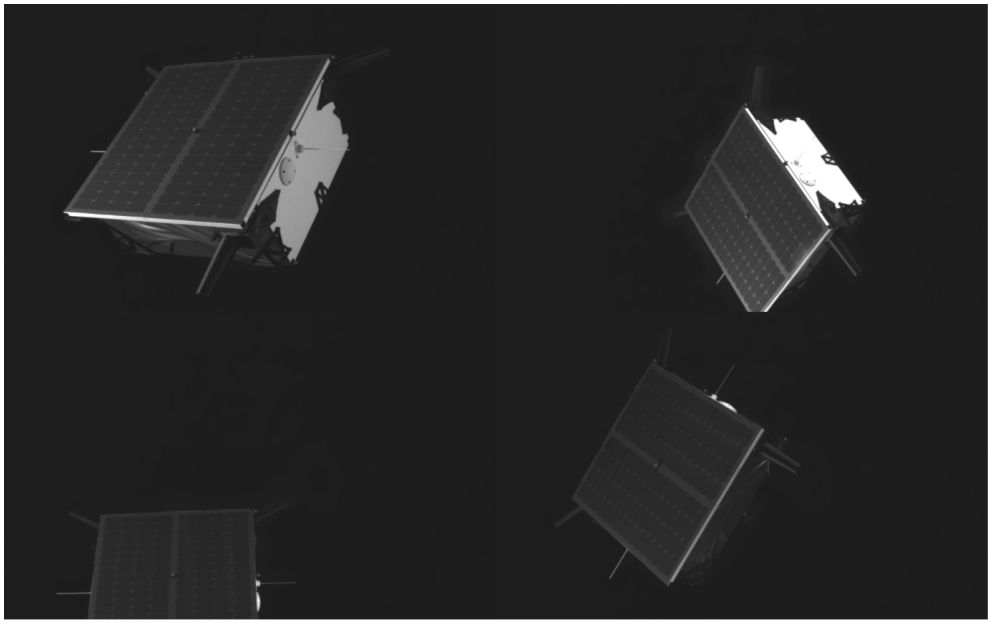


Figure 4.6: Some examples of the images of SPEED's real datasets.

Parameter	Description	Value
N_u	Number of Horizontal pixels	1920 [**px**]
N_v	Number of Vertical pixels	1200 [**px**]
d_u	Horizontal pixel dimension	5.86×10^{-6} [**m**]
d_v	Vertical pixel dimension	5.86×10^{-6} [**m**]
f_x	Horizontal focal length	0.0176 [**m**]
f_y	Vertical focal length	0.0176 [**m**]

Table 4.1: Parameters of the camera used in the SPEED dataset.

4.3 DATA PREPARATION

Since the SPEED dataset is composed only by B/W images, we need to add the silhouette annotation to the images. This will provide our net the needed information to understand the shape of the satellite via the Intersection over Union loss (described in Equation 4.1). We added this annotation manually using GIMP [40] to a subset of 215 images from the rendered train set. These images are chosen among the ones whose Z-distance was less than 10**m**. Moreover, we colored the background with white color, so that the output image of the renderer is as close as possible to the one provided in input: in fact, PyTorch3D colors the background of the image with a white color

Figure 4.7: Some samples of the images used to retrieve the 3D model before (left) and after (right) the pre-processing described in Subsection 4.3

by default. Each image is then reduced from 1920×1200 to 384×240. The output of this pre-processing step can be seen in Figure 4.7.

We also need to convert the annotations of the dataset, since the conventions used by PyTorch3D is different by the ones used to generate the images. These are the steps needed to convert from the dataset convention to the one used by PyTorch3D:

1. We multiply the x and y components of the translation vector by -1;
2. The provided quaternion is transformed into the corresponding rotation its Euler angles. Then, we multiply the Roll component ψ by -1.

This procedures are needed since:

- The provided representation is in camera view: the translation is thus the position of the object with respect to the camera (which is the center of reference), while in our experiment we need the position of

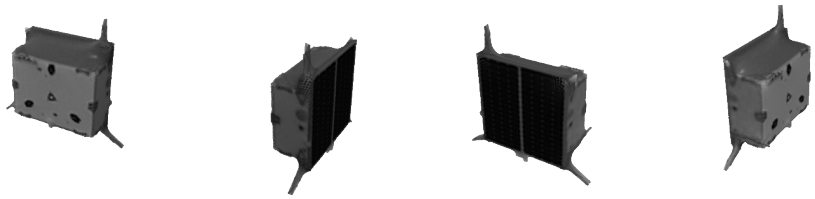


Figure 4.8: The final model.

the camera assuming that the object is in the center of the reference (standard view). So the translation vector we need is the opposite of the provided one.

- The Z-axis in the two representations goes in parallel but opposite direction. This is due to the fact that the classic definition of a camera is in a left-handed coordinate system, while PyTorch3D camera is represented in a right-handed coordinate system. This means that the translation Z component must be the opposite of the provided one, and that the rotation around the Z axis is in the opposite direction.

The dataset of 215 images is split in two subsets:

- a train dataset, composed of 194 images;
- a test dataset, composed of 21 images.

The train dataset is the one actually used to regress the 3D shape of the satellite. The test dataset is used to evaluate this prediction and to compare all the experiments run.

4.4 RESULTS AND COMPARATIVE ANALYSIS

We obtained the final result by training the neural network on the train set of 194 images until convergence. We used a batch size of 2 images and learning rate of 1×10^{-4} both on the mesh features and on the lighting predictor. Table 4.2 shows the resulting losses on the test set in its first row, along with the other results from the first comparative experiment. Figure 4.8 depicts the final predicted model, rendered with our Differentiable Renderer from 4 different angles.

To test the strength and the limits of this approach, we run four different sets of experiments:

1. Number of images: in this experiment we compare how the output changes with respect to the number of provided images;

#Images	IoU Loss	Color Loss	Edge Loss	Total Loss
194	**0.040881**	0.021851	0.024299	0.08703
108	0.04211	0.020173	0.024297	**0.086581**
44	0.042657	**0.02015**	0.024439	0.087247
22	0.047644	0.021036	0.024402	0.093084
10	0.068686	0.025496	0.024343	0.118527
4	0.166164	0.029678	**0.022009**	0.217852

Table 4.2: Comparison of the losses obtained in the experiment in Subsection 4.4.1. Best scores are represented in **bold**.

2. Number of faces: in this experiment we test how the learning is affected by the number of faces the starting sphere has;
3. Ablative analysis: we check how much the regularization loss affects the experiment;
4. Object represented: we test the portability of this experiment on 4 different objects.

4.4.1 *Number of images*

In this experiment the robustness of this approach is tested with a varying number of images. Since the number of images in this experiment varies so much, we need to balance the number of samples seen from the neural network: this is achieved by increasing the number of epochs proportionally to the number of images. This tweak is needed due to the fact that the model needs roughly the same number of optimization steps to converge. If less steps are taken, the final 3D model will both have an incorrect shape and worse colors. A comparison of a 44-images train and a 194-images train on the same number of epochs (360) is shown in Figure 4.10.

The outputs of this experiment are shown in Figure 4.9. It is clear from the images that the results are comparable for a number of images greater or equal to 22, while the process starts to fall off with a smaller number of images. In the experiment with only 4 images the regularization losses take over the 2D losses and try to collapse the model in the sections not depicted by the four images (shown in Figure 4.11). In the experiment with 10 images we can also see this phenomenon on a smaller scale on the bottom of the body on the left of the solar panel and in the upper part of the rear of the satellite (considering the solar panel the front of the satellite).

From Table 4.2 we can see the final loss obtained from each experiment. As we can see, the total loss is comparable for the models with more than 44

#Faces	#Images	IoU Loss	Color Loss	Total Loss
81920	194	**0.03728**	**0.02009**	**0.05737**
	22	0.05405	0.02316	0.07721
20480	194	0.04088	0.02185	0.06273
	22	*0.04764*	0.02104	*0.06867*
5120	194	0.05541	0.02134	0.07675
	22	0.06734	*0.01857*	0.08591
1280	194	0.18525	0.02369	0.20894
	22	0.20035	0.02454	0.22489

Table 4.3: Comparison of the losses obtained in the experiment in Subsection 4.4.2. Best scores with 194 images are represented in **bold**, best scores with 22 images in *italic*.

images, while it starts to fall off with 22 images and reaches very high values with less images. Moreover, it is shown how the Intersection Over Union loss increases with less images. The Color Loss is comparable for the models with more than 22 images, and the fluctuation is caused by the predicted light direction of the neural net. The edge loss has its minimum on the experiment with less images (due to the deformation of the final model); if we leave this particular case off, it reaches its best value with a lot of images: This is due to the fact that the vertices are more evenly spread on the surface, in particular in the three antennas.

In this comparative analysis it is shown that this model retrieval technique works with at least 22 images, while it gets better with an increasing number of images. A bigger number of images means that this technique can obtain more fine-grained details on the 3D model.

4.4.2 *Number of vertices*

In this experiment we tested how the output changes with the number of faces of the starting icosphere. The 3D models are obtained both with 194 and 22 images: this further tests how good is each regression with different images. To account for the longer edges present in the icospheres with less faces, the edge loss is halved in these cases. We tested this experiment on the same set of test images described before.

In Table 4.3 the results of the experiments are shown. Moreover, in Figure 4.12 we can see a visual comparison between the results of the experiment. As we can see from the table, the best result is obtained with 194 images and with the highest number of faces, while being comparable with the result with 20k faces. Then, the scores decrease with the number of faces. The model

Training type	IoU Loss	Color Loss	Total Loss
Full loss	0.04088	0.02185	0.06273
No Edge Loss	**0.03394**	**0.02143**	**0.05537**

Table 4.4: Comparison of the losses obtained in the experiment in Subsection 4.4.3. Best scores are represented in **bold**.

with only 1280 faces cannot capture fully the shape of the object, leaving the antennas out. On the other hand, we can see from the rendered images that the one with more faces shows some strange reflections: this is due to a wavy surface, and can be better seen analysing the 3D model in output. The IoU loss cannot capture this detail since the body is bent inward.

The result with 22 images has a different output: The experiment with 82k faces was worse than the one with only 20k faces, which was the best of the batch. Then, the resulting model gets worse with a smaller number of faces.

Finally, in this experiment we can see how the color gets worse with a smaller number of images: this is due to the fact that the models with less faces have a bigger number of pixels with too few faces to have a solid color. This mainly happens nearer the edges of the faces.

In this comparative analysis we can understand that with a small number of faces it is harder to capture the true shape of the model due to a lack of complexity of the base model. Moreover, due to the limitations of this particular differentiable renderer, the space in-between faces is harder to color. On the other hand, a model with many faces is harder to optimize (it takes more time) and it can sometime present a wavy surface (that can cause strange reflections).

4.4.3 *Ablative analysis*

This experiment aim to show the importance of the edge regularization loss. We train the model on 194 images and optimizing an icosphere with 20480 faces, both with and without the regularization loss.

As we can see from Table 4.4, The losses are much smaller without the regularization: this happens since the vertices are more free to move around, being no more tied one another. On the other hand, if we look at Figure 4.13 we can see that this lower loss comes at a price: the faces do not form a smooth surface, and this lead to some strange reflections that can be seen all over the satellite.

4.4.4 *Application on other objects*

In this subsection we test the portability of this approach on other objects. To do so, we took two 3D models from ShapeNet [8] (a guitar and an airplane) and two models from the internet (a cat and a Formula 1 car) and we rendered 100 images with PyTorch3D's built-in renderer set up as a non-differentiable rendered (this setting allows us to have more photo-realistic images). The rotation of the rendered object is a sample from a uniform distribution $X \sim \mathcal{U}(-\pi, \pi)$ for each Euler angle. The Z-axis translation is the absolute value of a sample from a Gaussian distribution $X \sim \mathcal{N}(3, 1)$, while the X-axis and Y-axis translation are samples from a Normal distribution $X \sim \mathcal{N}(0, 1)$, scaled by the Z-axis translation. The light direction components are samples from a Normal distribution $X \sim \mathcal{N}(0, 1)$, while its color components are fixed to be

- ambient color: `(0.4, 0.4, 0.4)`;
- diffuse color: `(0.6, 0.6, 0.6)`;
- specular color: `(0.0, 0.0, 0.0)`.

For each model we rendered 100 images and we used 90 images for training and 10 images for testing.

In Figure 4.14 we can see some samples of this rendering while in Figure 4.15 the output of the train process is shown. As we can see from these images, this process works well with other models and also with color images.

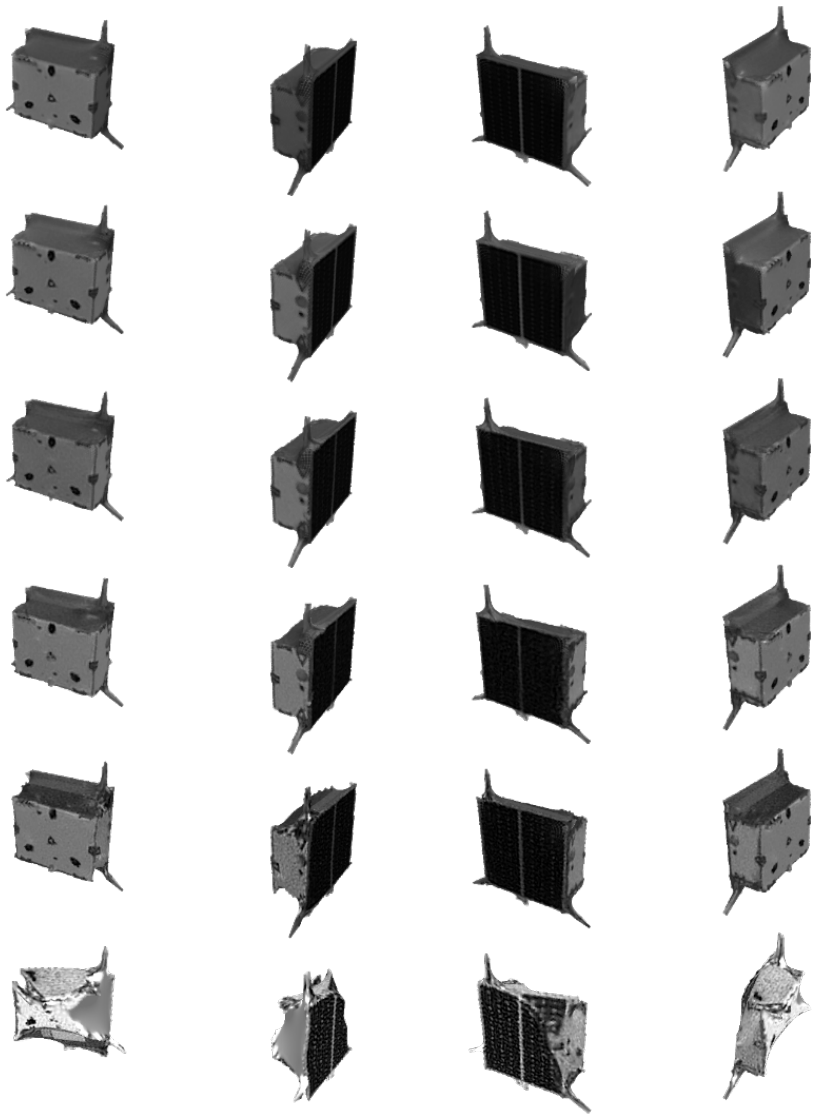


Figure 4.9: Comparison of the outputs of the first experiment. From top to bottom: model retrieved with 194 images, 108 images, 44 images, 22 images, 10 images and 4 images. All these models have 10242 vertices and 20480 faces.

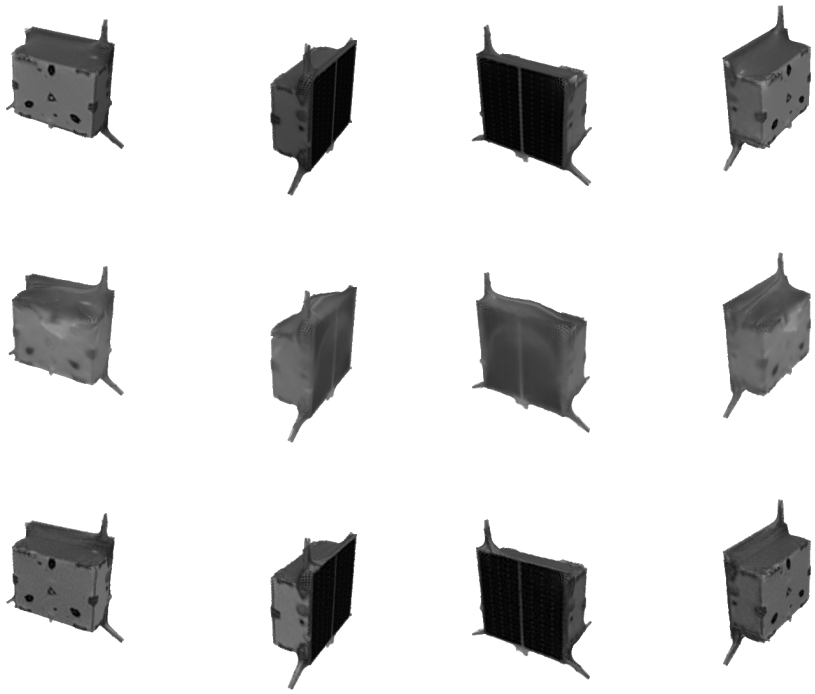


Figure 4.10: The effects that a uniform number of training epochs has on experiments with different number of images. From top to bottom: The model rendered after 360 epochs on 194 images; the model after 360 epochs on 44 images; the model after 1590 epochs on 44 images.

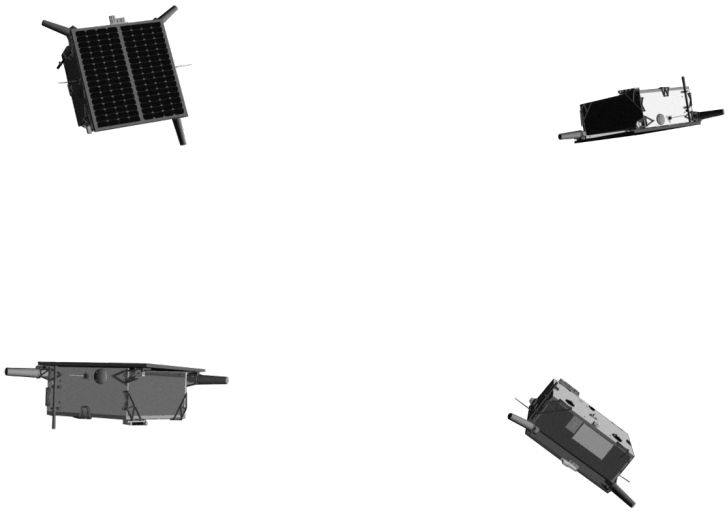


Figure 4.11: The dataset used to regress the 3D model with only 4 images.

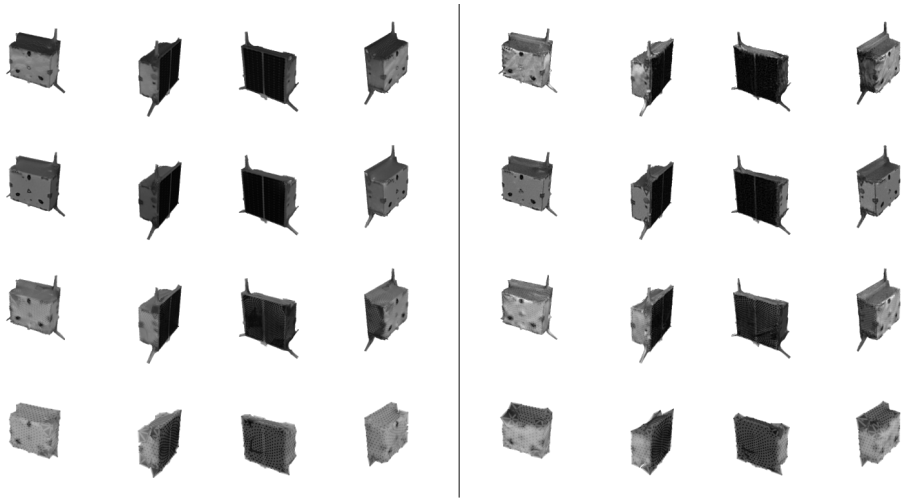


Figure 4.12: A comparison of the output images of the experiment shown in Subsection 4.4.2. The first column renders the model obtained with 194 images, while the second the one obtained with 22 images. Images are arranged from top to bottom in decreasing number of faces: The first image has 81920 faces, the second one 20480, the third one 5120, while the last one has 1280 faces.

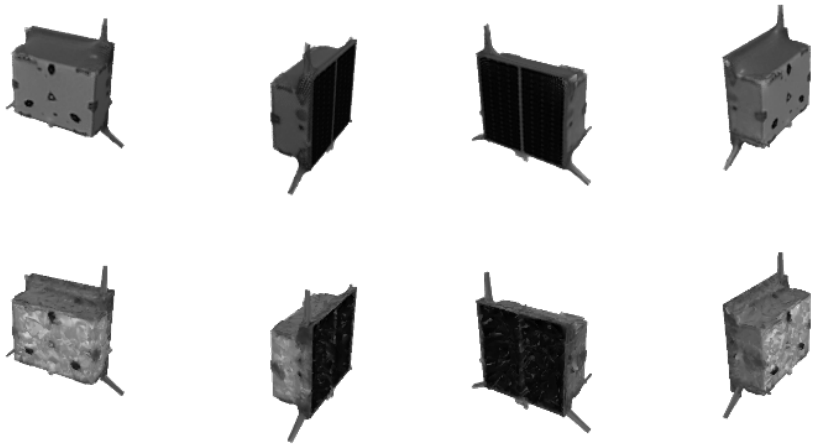


Figure 4.13: Rendering of the 3D model retrieved with (first row) and without (second row) the Edge Regularization Loss.

Figure 4.14: Some samples of the input images used for the 3D model retrieval with different objects.

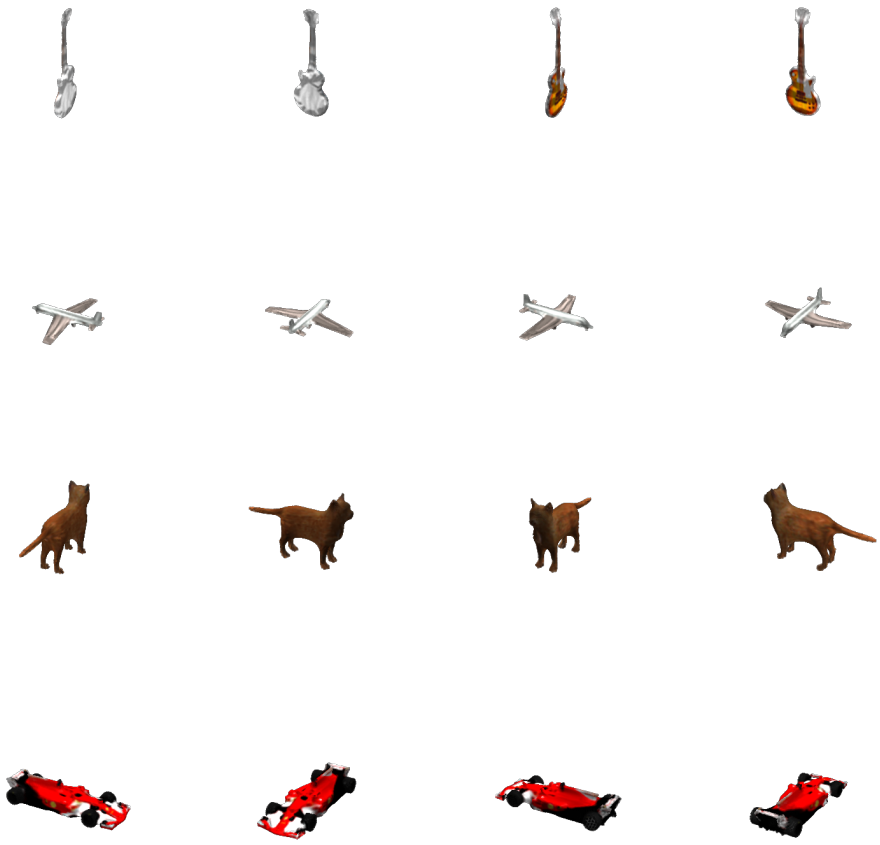


Figure 4.15: Rendering of the four models retrieved by our method.

5 SIX DEGREES OF FREEDOM POSE REGRESSION

In this chapter we will have a closer look to the Six Degrees of Freedom (6DoF) pose estimation problem starting from a monocular image. We propose a deep neural network architecture to tackle this problem, and we will explore how differentiable rendering can be employed in this task. Moreover, the main results obtained with this neural network will be used as a starting point for the pose refinement process that we will describe in Chapter 6.

5.1 MODEL ARCHITECTURE

The neural network that we built to tackle the 6DoF pose estimation problem is composed by two separate branches:

1. Rotation Branch: the branch of our neural network that regresses the rotation of the camera;
2. Localization Branch: the branch of our neural network that regresses the position of the camera with respect to the object.

This task will be addressed with a direct regression both on the translation and on the rotation of the object. We can see a visual description of this net in Figure 5.1.

Both branches share the same feature extractor: for this task, we used ResNet101 [9] pre-trained on ImageNet [11]. This feature extractor is slightly modified: we removed both the fully connected layer on top of it and the average pooling layer. This modification will allow for better results, since the fully connected layer's goal was to classify the elements of ImageNet, while the average pooling layer encodes an invariance to the pose and the rotation of the object. Since our goal is to estimate these values, we need to remove this layer.

The translation branch takes in input the full image (before any cropping) of size 384×240 (from now on called Localization Image). This image is processed in the feature extractor and then in two fully connected layers with 1024 neurons, followed by a fully connected layer with 3 neurons. This branch will output the predicted position of the object, encoded as a 3D vector.

The rotation branch takes in input only the region of interest representing the object, re-scaled to the dimension of 240×240 (from now on Rotation Image) and processes it with the feature extractor. Then, these features are passed trough 2 fully-connected layers with 1024 neurons and finally trough

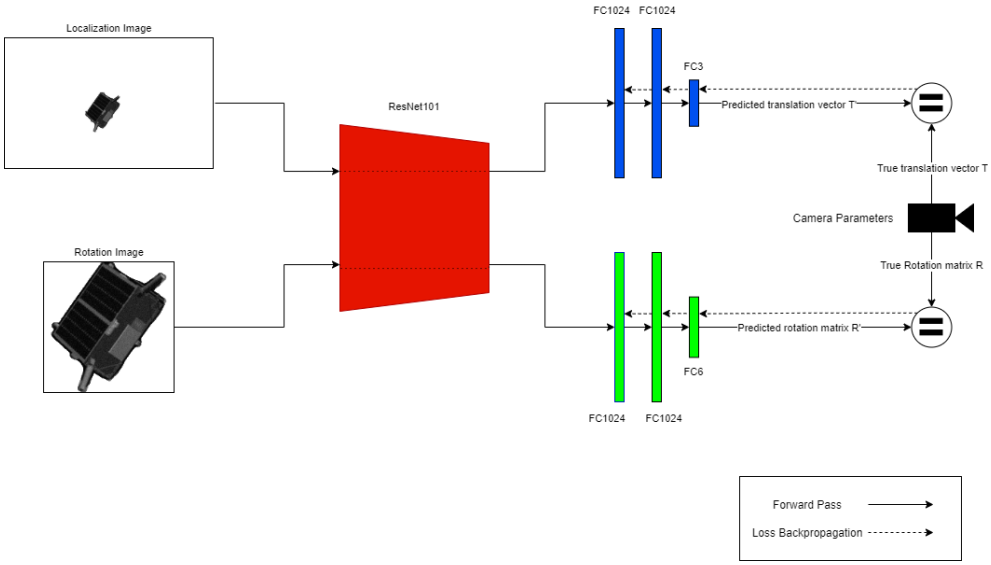


Figure 5.1: A scheme of the neural network used in this experiment.

a fully-connected layer with 6 neurons. ResNet101's output changes between the two branches: this is due to the fact that the dimension of the input image differs between the Translation Branch and the Rotation Branch (square 240×240 image for the Rotation Branch vs rectangular 384×240 image for the Translation Branch).

The 6D vector in output will be converted to a rotation matrix R as follows:

$$\vec{c}_1 = \frac{\vec{v}_1}{\|\vec{v}_1\|} \tag{5.1}$$

$$\vec{c}_2 = \frac{\vec{u}_2}{\|\vec{u}_2\|}, \quad \vec{u}_2 = \vec{v}_2 - \langle \vec{c}_1, \vec{v}_2 \rangle \vec{c}_1 \tag{5.2}$$

$$R = [\vec{c}_1 \quad \vec{c}_2 \quad \vec{c}_1 \times \vec{c}_2], \tag{5.3}$$

where $\vec{v}_1$ is the vector composed by the first 3 values of the output 6D vector, while $\vec{v}_2$ is the vector composed by the last 3 values. This matrix is orthonormal by construction, and thus a valid 3D rotation matrix.

The loss used in this experiment is:

$$L = L_T + \lambda L_R,$$

where L_T is a Translation loss and L_R is a Rotation loss. In this experiment we set $\lambda = 1$

The Translation loss used in this approach is defined as the Mean Absolute Error between the elements of the translation vectors:

$$L_T\left(\widehat{T}, T\right) = \frac{1}{3}\left\|\widehat{T} - T\right\|_1, \tag{5.4}$$

where T is the ground-truth translation vector and $\widehat{T}$ the predicted one.

The Rotation loss used in this approach is defined as:

$$L_R\left(\widehat{R}, R\right) = \sum_{e \in E} \left|\widehat{R}(e) - R(e)\right|, \tag{5.5}$$

where E is the set of all elements of a 3×3 matrix, $\widehat{R}$ is the predicted rotation matrix and R is the ground-truth one.

We trained this net over 100 epochs with an Adam optimizer with learning rate 1×10^{-4}.

5.2 DATA PRE-PROCESSING

In this subsection we will highlight how the input data are transformed before being used for training and testing. The dataset used in the experiment is the SPEED dataset [38], described in Section 4.2. This dataset provides many B/W images of the Tango spacecraft with annotations on the translation of the spacecraft and its rotation. In the experiment with this neural network, the annotations on the 3D translation and the rotation are converted to the PyTorch3D conventions as described in Section 4.3.

Image segmentation:

The first step of data preparation is to add the silhouette annotations to the dataset. We need the silhouette annotation for 2 different tasks:

- Background whitening: to remove the noise provided by the background, we color it in white in each image. This moreover will make the dataset images more similar to the added rendered ones (described in Subsection 5.2.1);
- RoI pooling: we use the silhouette of the satellite to define the Region of Interest (RoI) to input in the Rotation Branch.

We can add this silhouette with a Differentiable Rendering pipeline on the train images: We input the model obtained in Chapter 4 joint with the annotations on the 6DoF pose in a differentiable renderer, that will output the silhouette of the 3D model. The renderer is configured with the following settings:

- Camera: Perspective camera with FoV of 35.45°, calculated with Equation 4.3 from dataset parameters;
- Output image size: 384×384, then cropped to 384×240;
- Blur radius: 1×10^{-4};
- Z-buffer (faces per pixel): 100;

- Cull back side of the faces;
- No perspective correction;
- Shading: Soft Silhouette Shading, as described in [45]. This shader only outputs the silhouette of the object, skipping the textures interpolation.

To obtain the final image, we take the original B/W image and convert it to RGB, then we add the silhouette obtained from the renderer as fourth channel, obtaining an RGBA image. Unfortunately, we cannot employ this method to get the silhouettes of the test images: this is due to the fact that we don't have the annotations on the pose of the satellite in these images. This silhouette will be calculated by Detectron2 [47], an off-the-shelf neural network for segmentation. The details on its implementation and training are provided in Section 5.3.

Localization Images:

The second step is composed by a whitening of the background. This can only be done thanks to the silhouette annotation provided by the previous step. This allows the images to be standardized without any background, making the training more robust. To color the background, we take the silhouette annotation of the image and we color of white all the pixels with silhouette value of 0. This step is needed since the silhouette is just a mask on the image that does not change the values of the RGB channels. These normalized images can be seen in the second column of Figure 5.2. Unfortunately, since PyTorch3D rendering's blur does not scale with the distance, far images will have a little bit of background included in the silhouette. This can be seen in the first and last example of the same figure. The images created are the Localization Images, and will be the input of the Localization Branch.

Rotation Images:

A second set of all the images is then obtained from the Localization Images: this set aims to remove the invariance given by the different location that the satellite can have in the image. To do so, a square bounding box is drawn around the satellite starting from the silhouette. Then, the image is cropped following this square and this crop is resized to 240×240. Some examples of these images can be found in the third column of Figure 5.2 These images are the Rotation Images, and they will be the input of the Rotation Branch.

5.2.1 *Synthetic data enhancement*

In this subsection, we show how we generate synthetic data to input in our pipeline. This will provide us many more data to have a more robust learning. This procedure follows the findings of [12], that show how the rendered images can enhance the learning process.

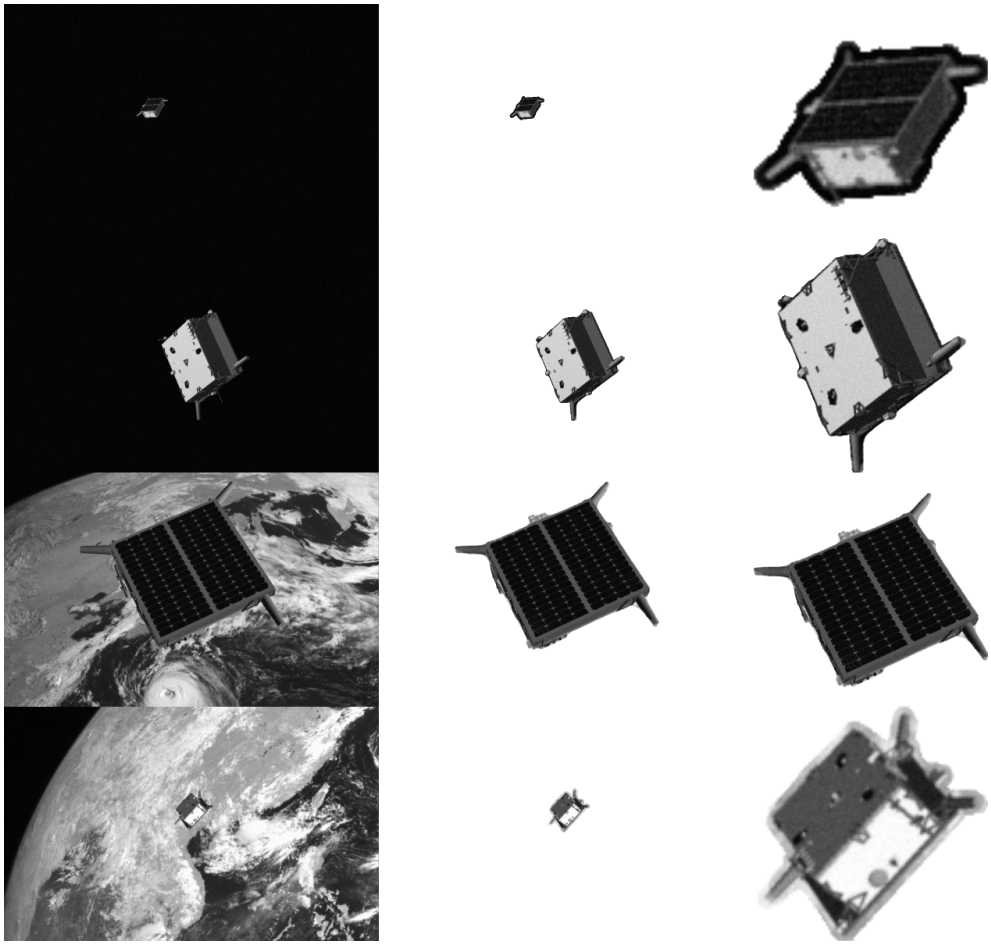


Figure 5.2: The images before (leftmost column) and after (center and right column) the data preparation. The second column shows some examples of the Localization Images, while the third column some examples of the Rotation Images.

To perform this data augmentation step using differentiable rendering, we used the 3D model of the Tango spacecraft that was obtained in Chapter 4 to render 100000 images. The output of this rendering will be used as Localization image, while the Rotation image will be obtained from the alpha channel (silhouette) of the image as described above. This allows the model to have many more views and thus to better generalize its pose regression process.

These images are rendered with a differentiable renderer with this configuration:

- Output image size: 960×960, then cropped to 960×600;
- Blur radius: 1×10^{-4};
- Z-buffer (faces per pixel): 100;
- Cull back side of the faces;

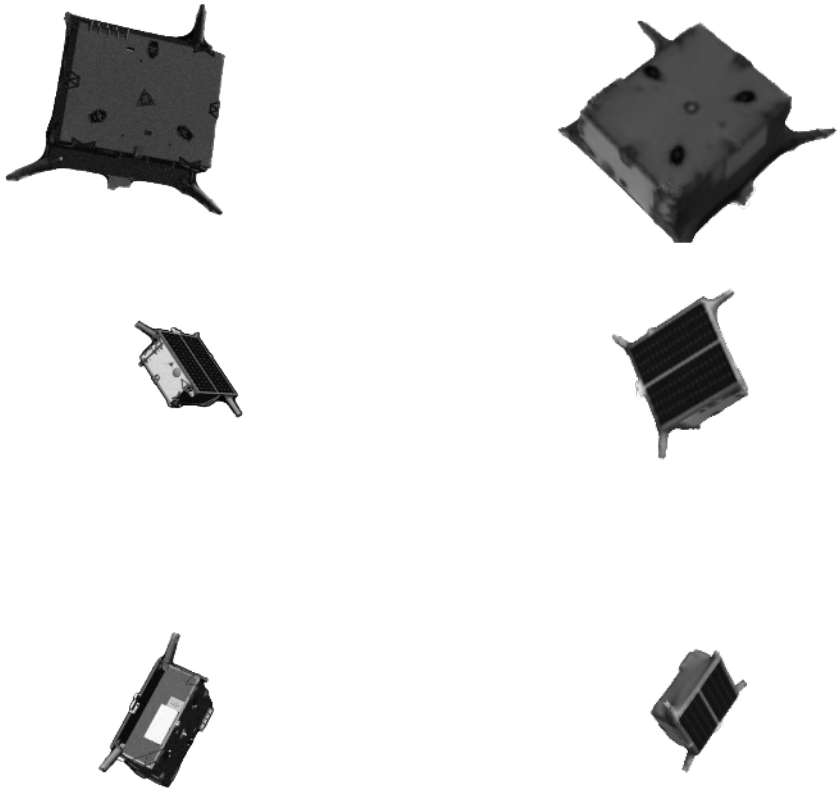


Figure 5.3: A montage of pre-processed images from SPEED's train dataset (left) and of images rendered with differentiable rendering (right).

- No perspective correction;
- Shading applied with a Soft Phong Shader as described in [45];
- Perspective camera with FoV of 35.45° (calculated with 4.3);
- Directional Light with ambient color `(0.3, 0.3, 0.3)`, diffuse color `(1.2, 1.2, 1.2)` and specular color `(0.0, 0.0, 0.0)`.

The lighting direction is generated randomly using a normal distribution $X \sim \mathcal{N}(0, 1)$ for all three directional components.

The position and the rotation of the satellite in these images are generated randomly. This will allow the neural network to see a lot of different views of the satellite, and thus to better regress the 6DoF pose. To generate a random rotation on one image, 3 different values are randomly sampled from a uniform distribution $X \sim \mathcal{U}(-\pi, +\pi)$: these values represent the three Euler angles of the rotation of the camera.

The translation vector is instead generated in two steps: firstly, a random value is sampled from a Gaussian distribution $X \sim \mathcal{N}(0, 100)$. The absolute value of this number is then summed to 3 to have the final distance value. The values of X, Y translations depend on the distance value: to generate these, two random numbers are sampled from a normal distribution $X \sim \mathcal{N}(0, 1)$ and are multiplied by the distance value and then by 0.025. These operations allow us to recreate the distribution of the provided dataset, as shown in Figure 5.4.

5.2.2 *Dataset split and composition*

Our final dataset is split as follows:

- Train dataset: the dataset that will provide a training signal to the neural network. It is composed of the full set of rendered images (100000 images) and of 10800 images of SPEED `train` dataset, for a total of 110800 images;
- Validation dataset: the dataset used to check how our neural network is generalizing. Since the final goal is to predict the 6DoF pose estimation of the satellite in ESA's rendered set, the validation set is composed only of the 1200 images SPEED's `train` set not used in our Train dataset;
- Real Validation set: with this dataset, we check how the net generalizes on a set of true images. It is composed by the 5 real images of SPEED's `real` dataset;
- Test set: the dataset on which the score is calculated. Since we do not have the annotations on the 6DoF pose for these images, the scoring can only be calculated by ESA's Pose Estimation Challenge. It is composed of the 2998 images of SPEED's `test` set;
- Real test set: the test dataset for the real images. This dataset is not provided by the annotations on the 6DoF pose of the satellite, too. It is composed by the 300 images of SPEED's `real_test` dataset

Each sample of the dataset is composed by

- the name of the source image;
- A Localization image;
- A Rotation image;
- A translation vector T (only for Train, Validation and Real Validation datasets);
- A rotation matrix R (only for Train, Validation and Real Validation datasets).

5.3 SEGMENTATION

To obtain the segmentation of the satellite in each image of the test sets where we don't have any annotation for the Tango's pose, we used an off-the-shelf object detection network. This choice has been motivated because several efficient networks have demonstrated their usefulness in different and challenging environments. In addition, some of them have published their weights trained on some of the most famous datasets, like COCO [2].

Among them, we chose to use Detectron2 [47], the last version of the object detection network developed by the Facebook Research Team.

5.3.1 *Training of Detectron2*

In order to obtain the segmentation of the full SPEED dataset, we used one of the models present in the model zoo of Detectron2, and then fine tuned it. In particular, we used the weights of Mask R-CNN-101-FPN trained on the COCO dataset of 2017.

For this fine-tuning, we tested different configurations of the training set, with annotations taken with different strategies, as well as their combinations. In particular, the following are the different training sets we developed:

1. **Hand Segmented Set**: We hand annotated a small set of 215 images with precise segmentation masks;

2. **Augmented 2D Data Set**: We then used the **Hand Segmented Set** set, and a set created by various images obtained through data augmentation operations. Among these operations, we used: random rotation of Tango, the addition of different backgrounds, resize of Tango, and darkening of the image. These last two operations had been implemented because some of the hardest images to recognize are those where Tango is really small and in shadow, and thus very dark;

3. **Rendered**: This set is composed by all the images of the SPEED's train set, where the segmentation is obtain using the 3D model generated in Chapter 4 and the annotation of the Tango's pose, as explained in Section 5.2;

4. **Augmented 3D Set**: We rendered a large set of images using the 3D model obtained in Chapter 4. Further details on data preparation are given in Subsection 5.2.1. We added a background to these images to make them similar to the ones of the provided dataset.

As we can see from the Table 5.1 (where the different settings, i.e., dataset combinations, are also detailed), the best result is obtained by the combination of Setting 5. The two images not recognized can be seen in the Figure 5.5.

Setting	Dataset	Image not recognized	Recognition Accuracy
Setting 1	Hand Made	16	99,51%
Setting 2	Hand Made + Data Augmentation 2D	3	99,90%
Setting 3	Rendered	4	99,88%
Setting 4	Rendered + Data Augmentation 3D	6	99,82%
Setting 5	Rendered + Data Augmentation 2D + Data Augmentation 3D	2	99,94%

Table 5.1: Image not recognized using each dataset describe earlier. For "Image not recognized" we mean all the images where the network does not find a segmentation or find a segmentation that does not resemble the real object, see Figure 5.5

Even if the recognition accuracy from Setting 4 is higher than Setting 2, the overall quality of the segmentations obtained by the first is greater. In the first row of Figure 5.6, we can see the quality of the segmentation increasing from left to right, succeeding to recognize even the upper right sensor of Tango, which is only partially segmented.

5.4 RESULTS AND COMPARATIVE ANALYSIS

In this section we report the results of this experiment and some comparative analysis to check which is the best configuration of the network and how it performs.

To compare our results, we used the score proposed in the Pose Estimation Challenge [44], composed by:

- A Translation score, that measures the error on the predicted translation vector $\widehat{T}$ with respect to the true translation vector T:

$$S_T\left(\widehat{T}, T\right) = \frac{\left\|\widehat{T} - T\right\|_2}{\|T\|_2};$$

- A Rotation score, that measure the rotation error as a geodesic loss between the predicted quaternion $\widehat{q}$ with respect to the true quaternion q:

$$S_R\left(\widehat{q}, q\right) = 2 \cdot \arccos\left(\left|\langle \widehat{q}, q \rangle\right|\right).$$

Approach	Renderer	Validation			Test
		T score	R score	tot score	Kelvins score
Regression	✗	**0.14857**	**0.10902**	**0.25759**	**0.38221**
Hybrid	✓	0.44624	0.11591	0.56214	0.61202
Renderer only	✓	1.0	1.46688	2.46688	3.18849

Table 5.2: Comparison on the different types of pipeline used. T score, R score and tot score are calculated on the validation set, while Kelvins score is calculated by Kelvins, ESA's Advanced Concepts Competition Website, where the Pose Estimation Challenge is hosted. Best scores in **bold**.

We report both the scores we calculated by us in validation and the score on the test set. This score is calculated by Kelvins - ESA's Advanced Concepts Competition Website[43], where ESA's Pose Estimation Challenge is hosted. We need to report both scores since the Kelvins scoring only gives us the total score $S = S_T + S_R$, thus providing no insight on the decoupled score on translation and rotation. Since we couldn't get a segmentation of image `img009630.jpg`, the prediction for that image is the same of the one on image `img009627.jpg`, which is the one just before that in the list of test images. For image `img011775.jpg`, on the other hand, we used the segmentation provided by Detectron2, even if the segmentation was very poor.

Our neural network reached a score of 0.25759 on our validation set, while the score given by Kelvins on the test set is of 0.38221. In Figure 5.7 some samples of the output predictions are shown with respect to the source image.

In the following subsections we will analyze how this model works in different scenarios:

1. Differentiable rendering: in this subsection we will analyse how viable is the employment of a differentiable renderer in the a 6 Degrees of Freedom pose estimation scenario;

2. Data representation: in this subsection we will compare four different rotation representations (6D-vector, quaternion, Euler angles regression and Euler angles classification) to check which one performs better in this application;

3. Added rendered images: In this subsection we will analyze how the added rendered images impact the training of the neural network, and if these images can be an asset in data augmentation.

5.4.1 *Differentiable rendering*

In this comparative analysis we check the advantages of adding a differentiable renderer in this approach with respect to a more classic regression approach.

The renderer used in this experiment is configured as follows:

- Camera: perspective camera with FoV of 35.45°, calculated with Equation 4.3 from dataset parameters;
- Lighting: fixed directional light with ambient color `(0.5, 0.5, 0.5)`, diffuse color `(0.3, 0.3, 0.3)`, specular color `(0.2, 0.2, 0.2)` and direction `(0, 1, 0)`. These are the settings of a default PyTorch3D directional light;
- Output image size: 384×384, then cropped to 384×240;
- Blur radius: 1×10^{-4};
- Z-buffer (faces per pixel): 100;
- Cull back side of the faces;
- No perspective correction;
- Shading: Soft Phong Shading, as described in [45].

This renderer takes in input the predicted rotation matrix and translation vector, joint with the 3D model of the spacecraft obtained in Chapter 4.

The total loss for this experiment can be rewritten as:

$$L = \lambda_{IoU} L_{IoU} + \lambda_{col} L_{col} + \lambda_T L_T + \lambda_R L_R,$$

Where L_{IoU} is the Intersection over Union loss described in Equation 4.1, L_{col} is the Color loss described in Equation 4.2, L_T is the Translation loss described in Equation 5.4 and L_R is the Rotation loss described in Equation 5.5. The various λ_i depend on the experiment.

We tested three different scenarios:

1. Direct Regression: the pipeline is composed only by the neural network described in Section 5.1. The loss is calculated only on the output rotation and translation of the pipeline:

$$\lambda_{IoU} = 0 \qquad \lambda_{col} = 0$$
$$\lambda_T = 1 \qquad \lambda_R = 1;$$

2. Hybrid approach: we stack the differentiable renderer on top of the pipeline of Direct Regression. The loss is calculated both on the output translation and rotation, and on the produced 2D image:

$$\lambda_{IoU} = 1 \qquad \lambda_{col} = 1$$
$$\lambda_T = 1 \qquad \lambda_R = 1;$$

3. Renderer only: the pipeline is composed of the neural network with the differentiable renderer on top. The loss is only calculated on the 2D image, without any loss on the translation vector or the rotation matrix:

$$\lambda_{IoU} = 1 \qquad \lambda_{col} = 1$$
$$\lambda_T = 0 \qquad \lambda_R = 0.$$

In Table 5.2 the results of this experiment are reported: the first row corresponds to the Direct Regression approach, the second to the Hybrid approach, while the third row to the Renderer only approach. As we can see, the worse performance is obtained with 2D supervision only: this is due to the fact that the neural network needs at least a bit of overlapping of the predicted silhouette and the true one to have a decent learning signal, and the random initialization almost never allows this. The Hybrid approach has a comparable result on the rotation score, while it falls behind in the translation score. This translation score difference is very impacting, and the final result on the test set is much worse in the Hybrid approach than in the Direct Regression approach. The best score was obtained with the Direct Regression method, which is also the faster approach to train, since the forward and backward pass trough the renderer is very computational-intensive.

In Figure 5.8, moreover, we have a visual comparison of the outputs of the neural network on the validation set. These images are rendered with the differentiable renderer described in Subsection 5.2.1 and are paired with the ground-truth image.

From these results we can say that the impact of the differentiable rendering in this task is not only useless, but damaging: it leads to worse overall results and it is slower to train. This may be due to the fact that the network before the differentiable rendering part is the same, and the 2D loss from the renderer conflicts with the translation and rotation loss to some extent. Moreover, the 2D supervision losses have lots of local minima: this may lead out neural network to stop its learning process in a sub-optimal state.

5.4.2 *Data representation*

In this analysis we compare how different rotation representations perform in 6DoF pose regression. The different representations used are

Representation	Validation T score	Validation R score	Validation tot score	Test Kelvins score
6D vector	**0.14857**	**0.10902**	**0.25759**	**0.38221**
Quaternion	0.16575	0.17057	0.33632	0.47944
Euler angles	0.16066	0.31071	0.47137	0.83889
Classification	0.15701	0.73306	0.89006	1.20431

Table 5.3: Comparison of the score obtained by each tested rotation representation on the validation and rendered test set. The first column shows the type of the representation, the second column the translation score obtained, the third column the rotation score, the fourth the sum of the previous values and the last column the scoring on testing images given by Kelvins. Best results in **bold**.

1. 6D vector: this representation is described in Section 5.1. The output of the Rotation branch is a 6D vector that is then converted to a 3×3 rotation matrix via Equation 5.1. The rotation loss used is described in Equation 5.5.

2. Quaternion: this representation is described in Section 2.2. The output of the rotation branch is a 4D vector that is then normalized trough the formula:
$$\hat{q} = \frac{\vec{v}}{\|\vec{v}\|_2}, \tag{5.6}$$
where $\vec{v}$ is the 4D vector in output from the neural network, while $\hat{q}$ is the final predicted quaternion. The loss used with this representation is:
$$L_{rot}(\hat{q}, q) = 2 \cdot \arccos(|\langle \hat{q}, q \rangle|) + \|\hat{q} - q\|_1,$$
where q is the ground-truth quaternion and $\hat{q}$ is the predicted one.

3. Euler angles: this representation is described in Section 2.2. The output of the rotation branch is a 3D vector $\hat{e} = (\phi, \theta, \psi)$. The loss function used with this representation is:
$$L_{rot}(\hat{e}, e) = 2 \cdot \arccos(|\langle \hat{q}, q \rangle|) + \|\hat{e} - e\|_1,$$
where $\hat{e}$ is the predicted vector of Euler angles, e is the ground-truth vector of Euler angles, $\hat{q}$ is the quaternion representation of the predicted Euler angles while q is the quaternion representation of the ground-truth Euler angles.

4. Classification on the Euler Angles: This representation consists in transforming the continuous problem of rotation estimation to a discrete one: the continuous space $(-\pi, \pi]$ is divided in 18 sections of the same

dimension, and each angle falls in one of these sections. The task then becomes predicting the correct section of the continuous space. The outputs of the neural network are 3 probability distributions over 18 values obtained with a softmax function, one for each Euler angle. The loss function is then a Cross Entropy loss function defined as:

$$L_{rot}(x, c) = -\log\left(\frac{\exp(x_c)}{\sum_{i \in C} \exp(x_i)}\right),$$

where x is the probability distribution in output, c is the correct class and C is the set of all classes. To calculate the score on this representation, the mean value of the predicted section is used as the predicted Euler angle.

As we can see from Table 5.3, the best scoring representation is the 6D vector one: as stated in [28], the strength of this representation lies in its continuity: thanks to this, the neural network has an easier job to regress the correct values. The quaternion representation falls a bit behind, but it is much better than the Euler angle representation due to its better continuity and the lack of gimbal lock. The classification representation is the worst performing: this may be due to the dimension of the bins, which introduces a lot of variance. In Figure 5.10 we can see the evolution of the Rotation score over the 100 epochs of train.

In Figure 5.9, moreover, we have a visual comparison of the outputs of the neural networks on the validation set. These images are rendered with the differentiable renderer described in Subsection 5.2.1 and are paired with the ground-truth image.

5.4.3 *Added Rendered Images*

In this subsection we analyze how the new rendered images impact the training. To test this, we run three different experiments with the same net and loss function as described in Section 5.1, and only changing the train set:

1. Full train set: in this experiment we used the full train set described in Subsection 5.2.2.
2. SPEED train set alone: in this dataset only the images provided by the SPEED dataset are used
3. Rendered set alone: in this experiment we will use only the images that we rendered as described in Subsection 5.2.1

As we can see from Table 5.4, the provided train dataset alone is not enough to have a good score: while the translation predicted is decent, the rotation is very bad. This is due to the fact that 10800 images may be

Set used	Validation			Test
	T score	R score	tot score	Kelvins score
Full set	**0.14857**	**0.10902**	**0.25759**	**0.38221**
SPEED Train set	0.21667	1.46912	1.68579	36.68126
Rendered set	0.76507	0.45268	1.21775	1.29453

Table 5.4: Comparison of the score obtained with different train sets. In the first row we can see the results of the main pipeline with both the SPEED train set and our rendered set; in the second row the results with only the SPEED train set are reported; in the last row, the results with only our rendered set is shown. Best results in **bold**.

very few to have a robust estimation of the rotation. Moreover, the learned pattern is not solid enough to be transferred to the test set images, whose segmentation is different (since it is provided by Detectron2 and not applied by the renderization of the 3D model).

The Rendered dataset compensate this problem: in fact its score on the predicted rotation is more robust. On the other hand, the estimation on this dataset is lacking in the translation vector estimation. This may happen since the predicted images are not similar enough to the provided ones, in particular in the far-away images where the Tango satellite is no more than a handful of pixels. This issue may be addressed by rendering images more similar to the provided ones. The close-up images, on the other hand, are similar enough to transfer this knowledge to the SPEED images.

The experiment with the full set of images outscores them both: this means that these types of images can be used together and that the images rendered with a differentiable renderer can be an asset in training this types of neural networks.

In Figure 5.11 we have a visual comparison of the outputs of the neural networks on the validation set. These images are rendered with the differentiable renderer described in Subsection 5.2.1 and are paired with the ground-truth image.

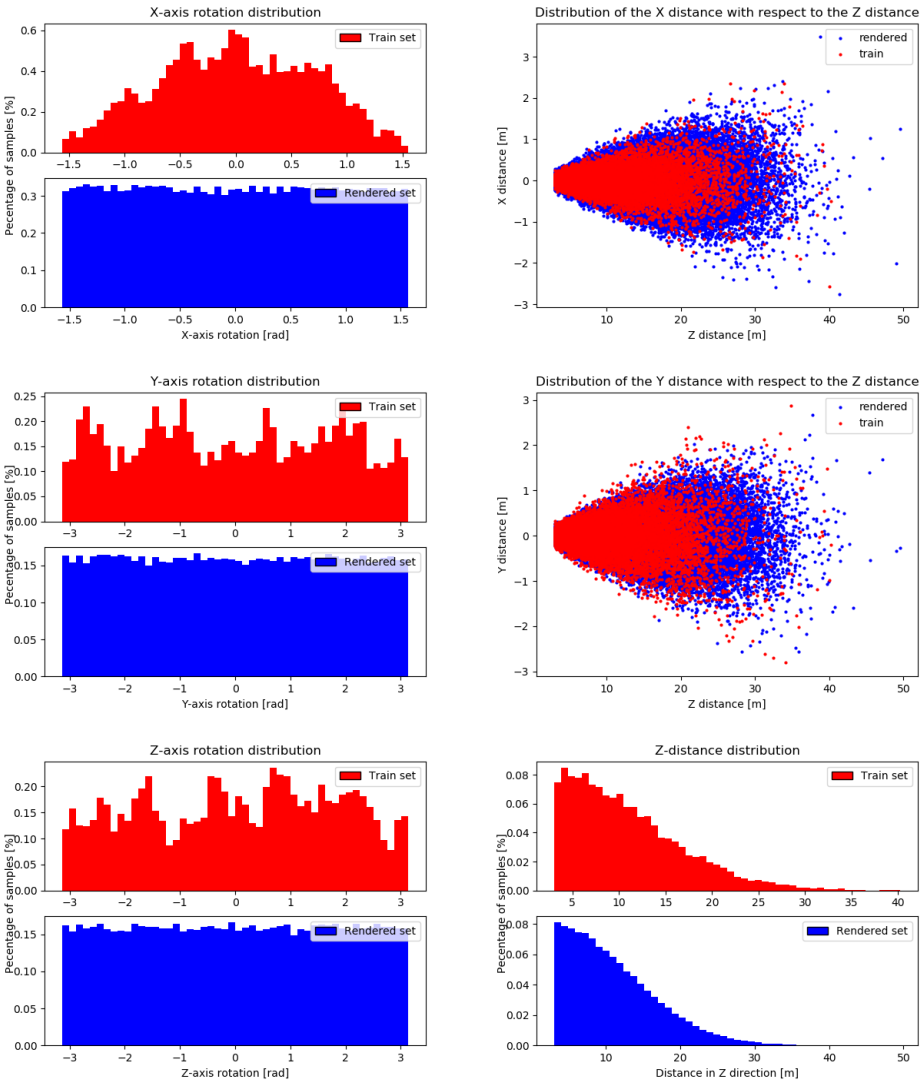


Figure 5.4: These graphs represent the distribution of the elements of the datasets both in rotation and in translation. For rotations, the continuous rotation around one axis is discretized in 50 bins of dimension 0.0628**rad** for X-axis rotation and 0.1257**rad** for Y-axis and Z-axis rotation. For translation, X and Y are represented as a scatter plot that represent the distribution with respect to Z, while the depth is represented in a discretized form dividing the continuous space in 50 bins of dimension 1**m** each. In red we see the train SPEED dataset and in blue our rendered dataset.

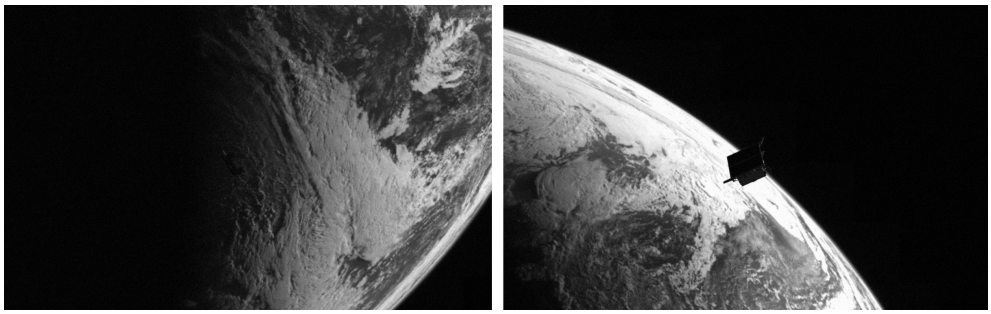


Figure 5.5: Images `img009630.jpg` (left) and `img011775.jpg` (right) from SPEED's `test` dataset. These two images are the only ones for which our network could not find a right segmentation. For image `img011775.jpg`, only a bad segmentation was found, while for `img009630.jpg` the object was never spotted.

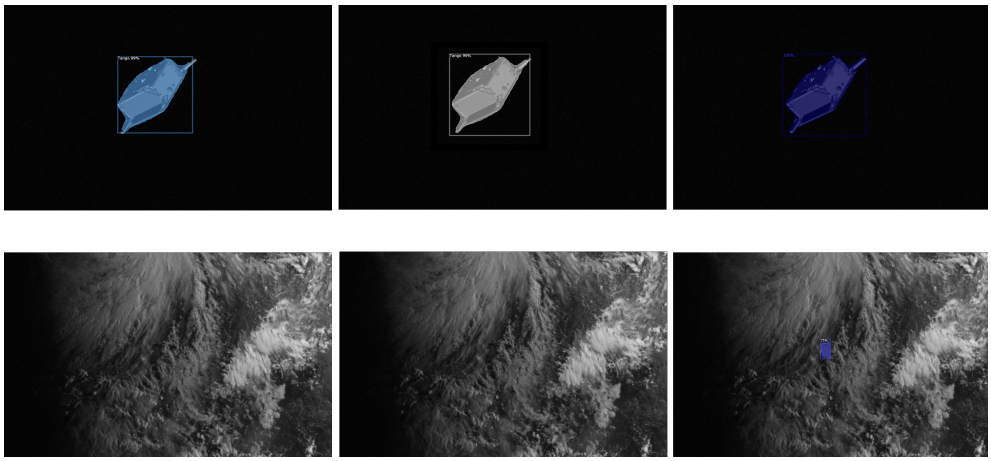


Figure 5.6: Result of the training of Detectron2. The datasets used are, from left to right, Handmade, Rendered + Data Augmentation 3D, Rendered + Data Augmentation 2D + Data Augmentation 3D. The first row shows image `img000022.jpg`, while the second row shows image `img008398.jpg`

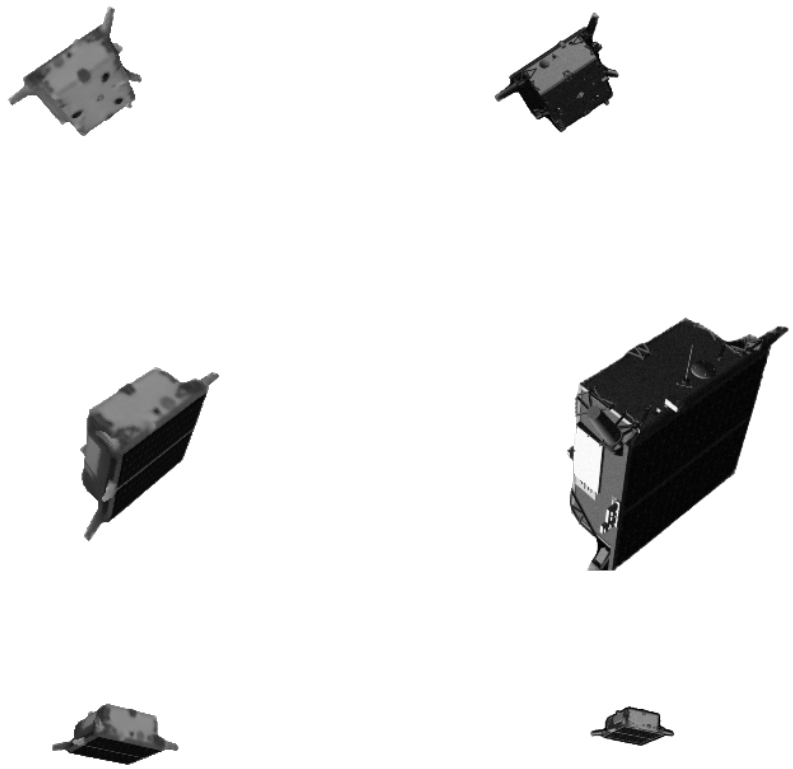


Figure 5.7: Some examples of the prediction of our neural network on the validation set. Our prediction is shown on the left, while the input Localization Image is shown on the right.

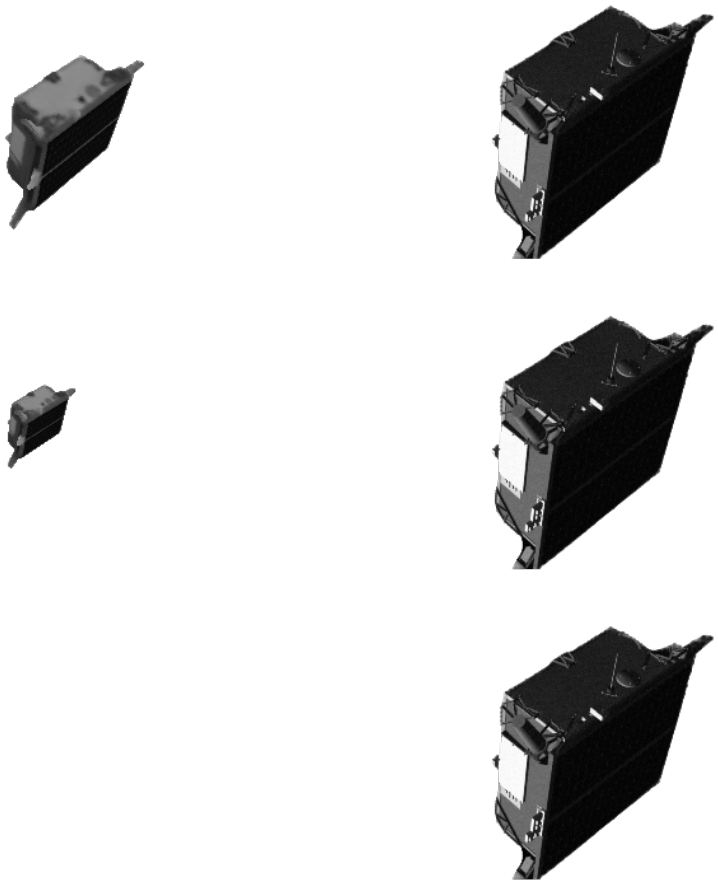


Figure 5.8: Visual comparison of the outputs of the experiment described in Subsection 5.4.1. On the left is shown a rendering of the prediction, while on the right is shown the input image. From top to bottom: Direct Regression, Hybrid Approach, Renderer only. In the last image the satellite is not present since its predicted location is out of the field of view of the camera.

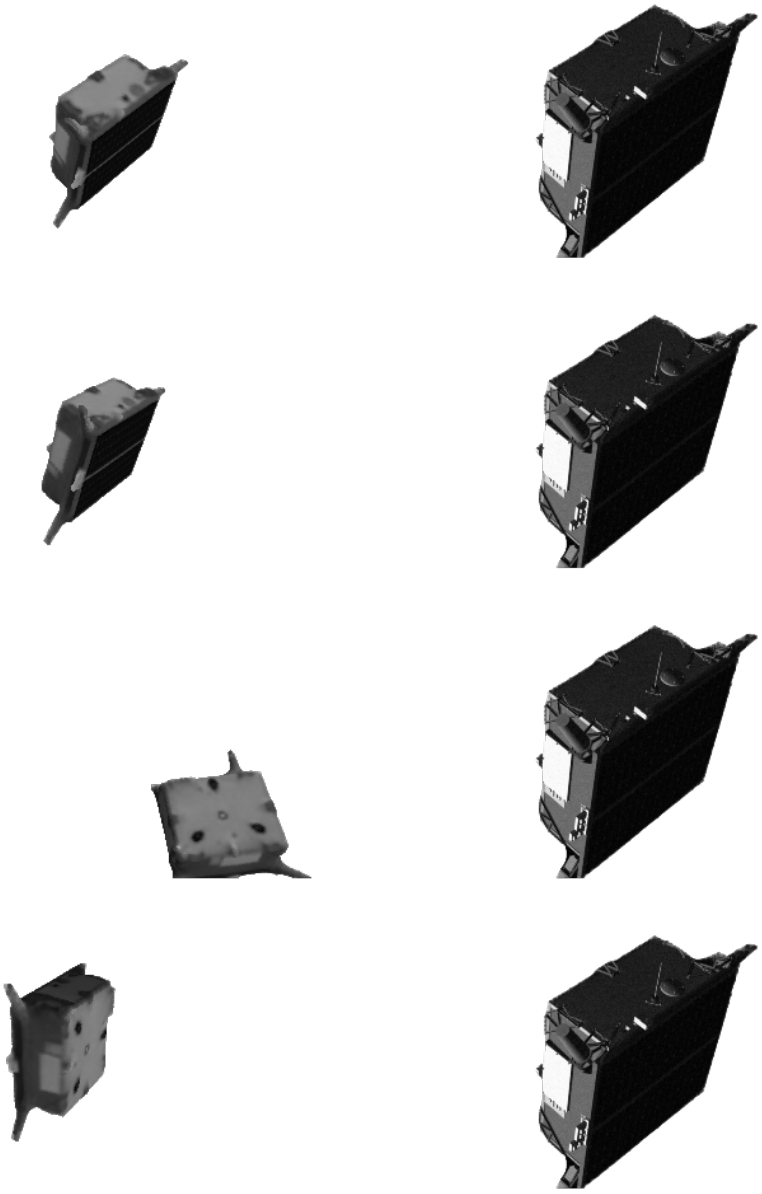


Figure 5.9: Visual comparison of the outputs of the experiment described in Subsection 5.4.2. On the left is shown a rendering of the prediction, while on the right is shown the input image. From top to bottom: 6D vector regression, quaternion regression, Euler angles regression, Euler angles classification.

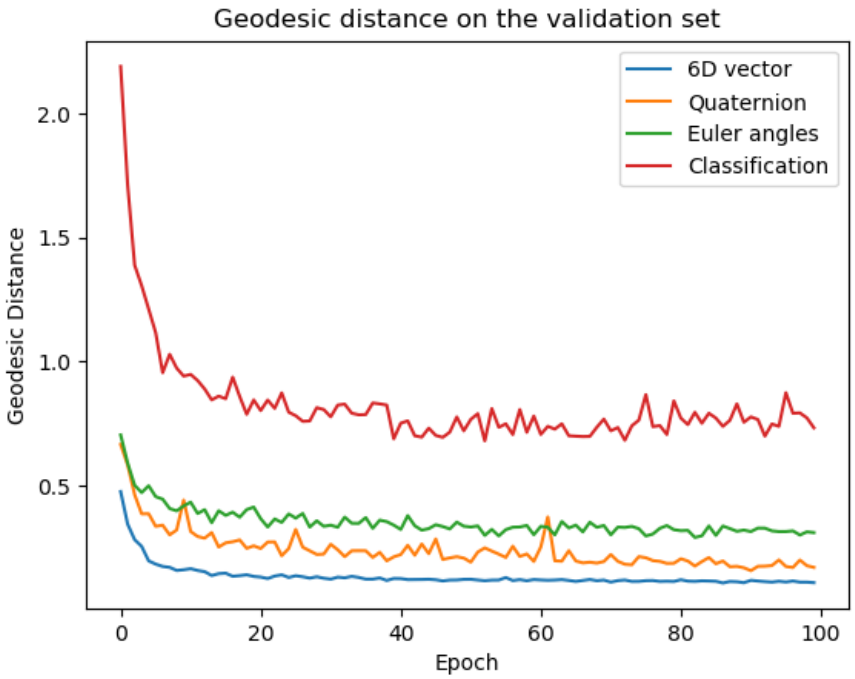


Figure 5.10: Comparison of the geodesic distances of the various methods tested in Subsection 5.4.2.

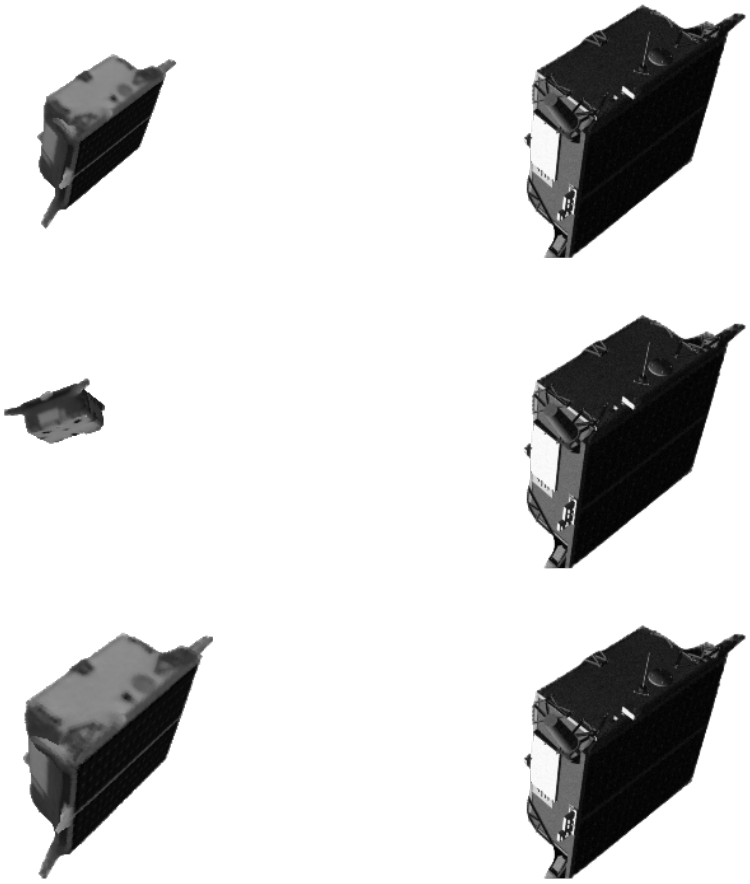


Figure 5.11: Visual comparison of the outputs of the experiment described in Subsection 5.4.3. On the left is shown a rendering of the prediction, while on the right the input image is shown. From top to bottom: Full dataset, SPEED dataset only, Rendered dataset only.

6 POSE REFINEMENT VIA DIFFERENTIABLE RENDERING

Another possible use of Differentiable Rendering is a pose refinement step with a render-and-compare approach. This technique consists in rendering with a differentiable renderer the image corresponding to a starting guess (usually an esteem from another 6DoF pose estimation process, like a neural network), compare this generated image to the source RGBA image, and then improve the predicted position based on this comparison. This procedure can be applied with an iterative approach, to have many optimization steps. In this chapter we describe the pipeline we used on the SPEED dataset [33], showing its ability in improving the accuracy of the predictions.

6.1 REFINEMENT PIPELINE

The goal of the neural network described in this chapter is, taken in input an esteem of the translation and the rotation of the camera, optimize it to have a better and stronger approximation. The values to be optimized are:

- The translation T, represented by a 3D vector `[Xt, Yt, Zt]`;
- The rotation R, that can be represented in many formats (3 Euler angles in case of the Euler regression, 4 un-normalized values in case of the quaternion regression, and 6 un-normalized values in case of the 6-DoF rotation regression). The chosen representation is then converted to a rotation matrix R_m. We need to optimize these representations instead of a rotation matrix or a normalized quaternion since they both have some sort of inter-dependency of the values that cannot be preserved easily via direct gradient descent. In fact the normalized quaternion must have norm 1 to encode a correct rotation, while the rotation matrix must me orthonormal not to deform the 3d model;
- The light direction, represented as a 3D vector `[Xl, Yl, Zl]`. This vector is initialized to `[0.0, 1.0, 0.0]` in our implementation for any input;
- The light color of the three components. Since the images have black and white colors, we only use one value for each light component: this way our light will also be black and white. The ambient light color is initialized to `0.1`, the diffuse color is initialized to `0.8`, while the specular color is initialized to `0.1`.

This neural network is fairly simple: it is composed only by a Differentiable Renderer. This renderer used is configured as follows:

- Camera: perspective camera with FoV of 35.45°;
- Output image size: 384×384, then cropped to 384×240;
- Blur radius: 1×10^{-4};
- Z-buffer (faces per pixel): 100;
- Cull back side of the faces;
- No perspective correction;
- Shading: Soft Phong Shading, as described in [45].

The renderer takes in input several elements:

- A 3D model of the object in the image;
- A rotation matrix R_m, representing the rotation of the camera with respect to the the object. This rotation matrix is obtained from the respective representation;
- A translation vector T, representing the translation of the camera from the object;
- A directional light object, comprehensive of the direction of the light and the color of the light.

We can see a scheme of this pipeline in Figure 6.1

The output image is then compared to the input RGBA image using the following loss function:

$$L_{tot} = L_{IoU} + \lambda L_{col},$$

where L_{IoU} is the Intersection over Union loss and L_{col} is the color loss, both described in Subsection 4.1. For this experiment the value of λ is set to be 1. The main advantage of the Intersection over Union loss is that, via a comparison of the two silhouettes, it allows us to match the shape of the true image providing a strong learning signal. On the other hand, it does not capture the intrinsic difference of the rotations with the same silhouette, but with different internal color, like in the case of a small tilt in one direction, where the edges of the model will not match. The Color loss address partially this issue: it can enhance the position in cases of small rotation error. The main disadvantage of this total loss is that it has a lot of local minima on the rotation side, so the starting pose must be already pretty accurate. These local minima arise since the Intersection over Union loss only tries to match a silhouette and has no actual knowledge on the rotation of the object, while the color loss can identify that some colors are more predominant in a very

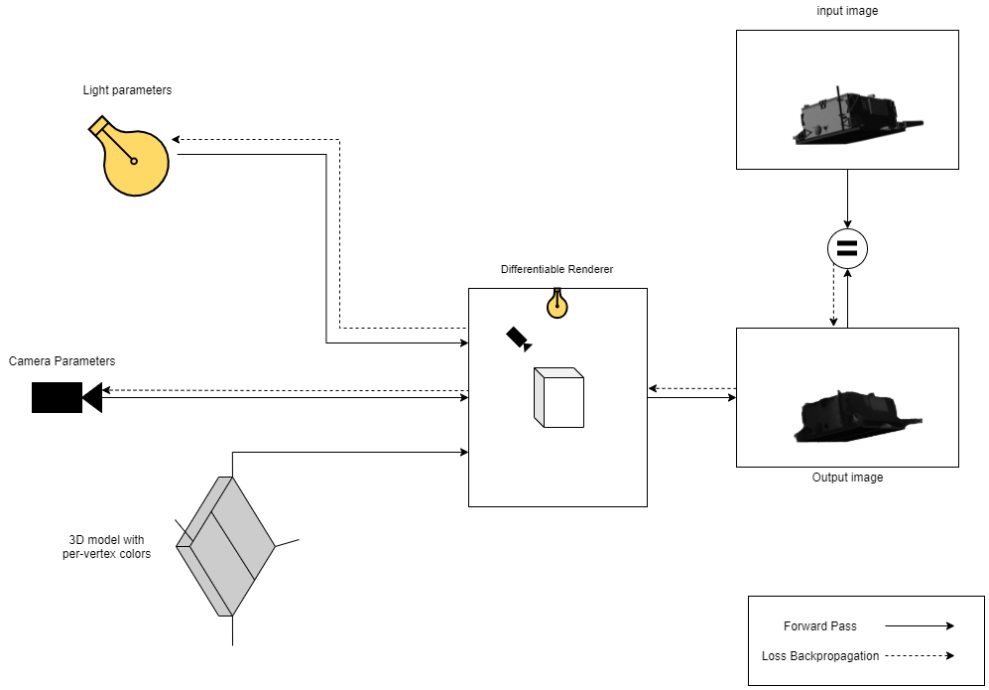


Figure 6.1: A scheme of the neural network used in this experiment.

	Translation	Rotation	Light Direction	Light Color
Start value	1×10^{-1}	1×10^{-2}	1×10^{-2}	1×10^{-2}
Decreased value	1×10^{-2}	1×10^{-3}	1×10^{-3}	1×10^{-3}

Table 6.1: The losses used in this experiment. The first row shows the learning rate at the beginning of the process, while the second row the learning rate used after the first convergence. The columns identify the value optimized with that learning rate

close area, but cannot match colors too far apart. Moreover, this loss needs to have at least a bit of starting Intersection over Union to converge, but given this it will provide a very good learning signal to optimize the translation vector.

The optimizer used to perform this optimization is an Adam optimizer. The learning rate vary between the different elements to be optimized: this is needed since a step of the same magnitude on translation is much smaller than a step of the same magnitude on the rotation. The learning rates are shown in Table 6.1

The model optimizes until convergence with the starting learning rates; then, the learning rates are multiplied by 1×10^{-1} to have a more fine-grained optimization, and are again trained until convergence. This allows to speed up the process without loosing on accuracy. To check for convergence, we used a patience of 30 optimization steps on the total loss. At the end of the

optimization procedure, the input values that generated the best image are saved to a file.

The optimization procedure is a render-and-compare loop operating on a single image. The procedure is represented in the following pseudo-code, that does not take into account patience and learning rate decrease to give a better picture of the core of the algorithm:

```
model = load_3d_model()  # the 3D model with color
patience = 30
for image_name in IMAGE_NAMES:
    # load the image and the predicted starting rotation and
    # translation
    in_img = IMAGES(image_name)
    rot = PREDICTED_ROTATIONS(image_name)
    T = PREDICTED_TRANSLATIONS(image_name)
    # setup starting light
    light_direction = tensor([0.0, 1.0, 0.0])  # [Xl, Yl, Zl]
    light_colors = tensor([0.1, 0.8, 0.1])  # [ambient, diffuse, specular]

    # is_best() checks for convergence
    while not is_best():
        optimizer.zero_grad()
        #transform in rotation matrix
        R = prediction_to_R_matrix(rot)
        # render...
        out_img = render_image(
            model, R, T, light_direction, light_color
        )
        # ... and compare
        loss = calculate_loss(out_img, in_img)
        #loss is back-propagated
        loss.backward()
        optimizer.step()
# the best values are saved
save_values()
```

6.2 RESULTS

In this section the results of this experiment are reported. We tested this process on SPEED's [38] `test` and `real_test` datasets. This dataset is described in Section 4.2. The ground-truth images for this process are the Localization Images described in Section 5.2, whose silhouette was obtained via Detectron2, as described in Section 5.3. We cannot generate the silhouettes of the images

Figure 6.2: Rendering of the model before (left) and after (center) the pose adjustment step. The image on the right is the ground truth image. Result on the **6D vector** representation. Starting IoU loss: 0.2453; final IoU loss: 0.0458; image name: `img 000014.jpg`.

Figure 6.3: Rendering of the model before (left) and after (center) the pose adjustment step. The image on the right is the ground truth image. Result on the **quaternion** representation. Starting IoU loss: 0.3655; final IoU loss: 0.044; image name: `img 000014.jpg`.

via differentiable rendering as described in Section 5.2 since we do not have the annotations of the ground-truth 6DoF pose of the two test sets that are used in this experiment. The input 3D model for this experiment is the model obtained with the procedure described in Chapter 4. This model's texture is described in a per-vertex notation: each vertex as an associated color, and the final texture of the model is an interpolation of these per-vertex colors in a shader-dependent way.

We tested this approach on 5 different scenarios:

1. 6D vector: we used as initialization the predictions of the experiment in Section 5.1. The optimization on the rotation is done optimizing the elements of a 6D vector, then transformed in a rotation matrix as described in Equation 5.1. A sample image before and after the pose adjustment step in this scenario can be seen in Figure 6.2

2. Quaternion: we used as initialization the quaternions predicted in the experiment in Subsection 5.4.2. The optimization is performed on a 4D vector, from which we obtain a quaternion trough a normalization (Equation 5.6). Then, we obtain the rotation matrix R as:

$$R = \begin{bmatrix} 1-2q_y^2-2q_z^2 & 2q_xq_y-2q_wq_z & 2q_xq_z+2q_wq_y \\ 2q_xq_y+2q_wq_z & 1-2q_x^2-2q_z^2 & 2q_yq_z-2q_wq-x \\ 2q_xq_z-2q_wq_y & 2q_yq_z+2q_wq_x & 1-2q_x^2-2q_y^2 \end{bmatrix},$$

Figure 6.4: Rendering of the model before (left) and after (center) the pose adjustment step. The image on the right is the ground truth image. Result on the **Euler angles** representation. Starting IoU loss: 0.998; final IoU loss: 0.0443; image name: `img 000014.jpg`. This particular image shows us how strong it can be this pose refinement technique: it managed to obtain a very good prediction given a little bit of IoU.

Figure 6.5: Rendering of the model before (left) and after (center) the pose adjustment step. The image on the right is the ground truth image. Result on the **classification** representation, then transformed to Euler angles. Starting IoU loss: 0.5961; final IoU loss: 0.1675; image name: `img000014.jpg`. In this case the predicted rotation was very far away from the correct one, so the pose refinement step can only better the translation while getting stuck in a local minimum in the rotation.

where $q = q_w + q_x \cdot i + q_y \cdot j + q_z \cdot k$. A sample image before and after the pose adjustment step in this scenario can be seen in Figure 6.3.

3. Euler Angles: we used as initialization the Euler angles regressed in the experiment in Subsection 5.4.2. The optimization is performed directly on the 3 Euler angles, that are applied in the order $Z \rightarrow X \rightarrow Y$ to obtain the final rotation matrix R. A sample image before and after the pose adjustment step in this scenario can be seen in Figure 6.4.

4. Classification: we used as initialization the best scoring class for each Euler angle. Each class is converted to its corresponding Euler angle, which is the mean value of all the Euler angles that correspond to that class. Then, we optimize directly these three Euler angles, that are applied in the order $Z \rightarrow X \rightarrow Y$ to obtain the final rotation matrix R. A sample image before and after the pose adjustment step in this scenario can be seen in Figure 6.5.

5. UrsoNet [37]: UrsoNet is a neural network that competed for the first tranche of the Pose Estimation Challenge, achieving third position. They made their neural network and the weights used public, so that other

Figure 6.6: Rendering of the model before (left) and after (center) the pose adjustment step. The image on the right is the ground truth image. Result on the **UrsoNet** prediction on the `test` dataset. Starting IoU loss: 0.0998; final IoU loss: 0.0444; image name: `img000014.jpg`.

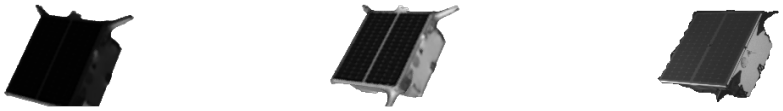


Figure 6.7: Rendering of the model before (left) and after (center) the pose adjustment step. The image on the right is the ground truth image. Result on the **UrsoNet** prediction on the `real` dataset. Starting IoU loss: 0.2772; final IoU loss: 0.0577; image name: `img000007real.jpg`.

people could experiment with their net. Their neural network is composed by a pre-trained feature extractor followed by two branches: the localization branch that directly regresses the translation vector T, and a rotation branch that obtains the quaternion q via probabilistic fitting. We used this neural network with the pre-trained weights provided and scored them to have a starting score of this model to use it as a comparison. Then applied the pose refinement step to these predictions. Since the output of this neural network uses the same conventions of the SPEED dataset, we converted both the predicted translation vector and the predicted rotation quaternion to PyTorch3D's convention using the procedure shown in Section 4.3. We optimize the value of the Euler angles obtained from this conversion. The order of application of the three Euler angles follows the convention $Z \rightarrow X \rightarrow Y$ to obtain the final rotation matrix R. A sample image before and after the pose adjustment step in this scenario can be seen in Figure 6.6. The results on UrsoNet are reported both for the test set of rendered images and for the test set of real images. This is possible only for this scenario: the scoring on Kelvins only shows the best result ever achieved by the submissions for this test set. On the other hand, it provides the scoring on the test set of rendered images for all submissions. In Figure 6.7 we can see the

Method	Competition score	Before P. R.	After P. R.
6D vector	✗	0.38221	0.23191
Quaternion	✗	0.47944	0.29602
Euler angles	✗	0.83889	0.74775
Classification	✗	1.20432	1.00247
UrsoNet [37]	0.05546	0.07311	0.05279
UrsoNet [37] real	0.14763	✗	0.14263

Table 6.2: The improvement obtained with the pose refining step on the score calculated by Kelvins - ESA's Advanced Concepts Competition Website [43], which host the Pose Estimation Challenge [44]. The first four rows are the results obtained in Subsection 5.4.2 on the `test` dataset before and after the pose refinement step. The last two rows show the score achieved by UrsoNet [37], respectively on the `test` and the `real_test` dataset. For this neural network, the score obtained by the authors in the challenge is also reported. P.R.: Pose Refining

application of the pose refinement technique on one image of the test set.

As we can see from Table 6.2, the process of pose refinement using differentiable rendering improves the score in all situations. In the figures showing the results of the experiments we have a visual representation of this positive impact: the final prediction is much more close to the correct position of the satellite than the starting one. Moreover, we can see how this impact works well on all types of rotation representation: this means that this approach is not limited by the representation and can be applied with every one of them. We just need to be sure that the representation is in the same convention of the renderer and that it is a valid rotation: this is no problem with the Euler angles representation, since each combination of numbers is a valid rotation representation. On the other hand, we must be sure to have a normalized quaternion and an orthonormal rotation matrix to avoid distortions on the final 3D model. An example of this distortion can be seen in Figure 6.8, where the optimization was performed directly on the values of the rotation matrix.

As we can see from the last rows of the table, this method managed to improve also the result of UrsoNet, both with respect to our prediction with its pre-trained neural network and the best result obtained by UrsoNet on the Pose Estimation Challenge, proving that this method can also improve very competitive scores. Moreover, we also managed to improve UrsoNet score on the real dataset with this technique, showing that both the 3D model obtained and this Pose Refinement technique can also be used on a dataset of real images to improve the score.

Figure 6.8: An example of optimization preformed directly on the rotation matrix R. The image on the right shows the initialization (translation from 6D vector esteem, random rotation), the central image shows the output of the pose refinement technique, while on the right the ground-truth image is shown. As we can see, the satellite was distorted to fit as much as possible to the silhouette of the input image. This behaviour can be avoided by optimizing on a set of values that are not linked together, like Euler angles or 4 distinct and uncorrelated values that will then be normalized against each other to form a quaternion representing a valid rotation.

While this technique is pretty simple to implement and its performance is noticeable, it has its drawbacks:

- To apply this method we need the 3D model of the object. The better is this model, the better the optimization is. In single-model datasets (like the SPEED dataset) this model is easy to obtain with the technique shown in Chapter 4, while in multi-model datasets (like ImageNet [11]) this task is harder, since it needs to obtain the specific model for the specific object in the image.
- This technique scales with the precision of the silhouette: the loss represented uses mainly this annotation to obtain the 3D pose of the object, so the more precise it is, the better will be the application of this technique. An example on how a bad silhouette impacts on this technique can be seen in Figure 6.9. In our case, it is worth considering that even if there were few bad silhouettes, the impact is visible and positive on the full test set.
- The 2D supervision has a lot of local minima, thus the advantages brought by this technique are limited by how good is the initialization. This method cannot handle the problem of 6 Degrees of Freedom pose regression alone. This can be seen from the results reported in Table 6.2: each result after the pose adjustment step depends a lot on how good was the result obtained by the initialization values.

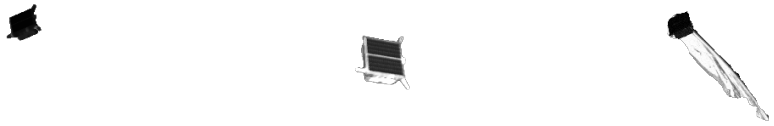


Figure 6.9: The effect of pose refinement on a badly cropped image. The image on the left is the prediction of UrsoNet, the central image is the output of the pose refinement technique, while the image on the right is the ground-truth image. As we can see, the final prediction may be worse than the starting prediction. On the other hand this happens only for images with a really poor segmentation.

7

CONCLUSIONS AND FUTURE WORKS

In this work we presented a new method to address the 6 degrees of freedom pose estimation using differentiable rendering. With this method, first we reconstruct the 3D model of an object with a differentiable rendering technique, then we use this information to enrich our dataset with new images and useful annotations, and regress a first estimation of the six degrees of freedom. Finally, we refine this coarse pose with a render-and-compare approach using differentiable rendering. From the results shown in this work, we can state that differentiable rendering can be an asset in 3D pose regression as well as in 3D model retrieval.

In our analysis, this novel technique's main employment was to reconstruct a 3D model starting with a small set of 2D images. As shown in Chapter 4, it is possible to recreate 3D models of different objects, given the camera parameters and the annotations on the pose of the object in these images. This can be used not only in the retrieval of the 3D pose of the objects, but also to generate a lot more data and annotations. For example, a 3D model can be rendered in the same position of the input data using his pose annotations to get the segmentation of that object.

Moreover, we show that this technique can be used as an effective strategy to generate a large number of new images. This could be really useful for all the approaches that desperately need a big set of data. Normally, deep learning methods applied to computer vision require huge amounts of data. With this technique is possible to recreate an unlimited amount of new images to train a robust prediction of the model.

Another application for this technique is 3D pose regression. We tested the use of differentiable rendering in two different scenarios: direct regression of the 3D pose and optimization of an estimated pose. In the first case, we found that this approach is not comparable with the actual state-of-the-art approach. In fact, the results that we have obtained, as showed in Chapter 5, are worse than the ones obtained with the other approach, as well as considerably slower.

On the other hand, the use of differentiable rendering in a pose refinement scenario proved itself to be very solid and consistent, enhancing the prediction in each different scenario we tested it in, even enhancing the score of one of the best competitors of the first tranche of the challenge (as shown in Chapter 6). Unfortunately, we could not test this approach on the best scoring result of the challenge: while the neural network used was described in [29],

the pre-trained weights were not provided and an hardware gap did not allow us to train it.

The final result obtained by our method is promising, and even if it did not reach the best absolute result in the ESA Competition it is a very competitive score. Moreover, it could be interesting to test our method using the pipeline used by the winner of the challenge, to verify if we can enhance the result that they have obtained.

A future work in the field of pose regression using differentiable rendering can be testing this approach with other types of differentiable renderers, e.g. DIB-R [30]. Moreover, this method could be tested in other domains, like pose estimation in domotic or urban environment: in these settings the laws of physics limit the degrees of freedom of an object (i.e. the object must be put on the ground or fixed to a support) and thus a differentiable renderer could be useful to encode real world dynamics. Finally, the pose refinement approach can be slightly modified to regress not the 6 Degrees of Freedom pose, but the lighting of a scene: given a 3D scene with the material and textures annotation, it is possible to obtain the light direction (in case of a directional light) or position (in case of a point light) and its color characteristics (ambient color, diffuse color and specular color).

BIBLIOGRAPHY

[1] Vincent Lepetit, Francesc Moreno-Noguer, and Pascal Fua. "EPnP: An Accurate O(n) Solution to the PnP Problem." In: *Int. J. Comput. Vis.* 81.2 (2009), pp. 155–166 (cit. on p. 14).

[2] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. "Microsoft COCO: Common Objects in Context." In: *CoRR* abs/1405.0312 (2014). arXiv: 1405.0312 (cit. on p. 54).

[3] Matthew M. Loper and Michael J. Black. "OpenDR: An Approximate Differentiable Renderer." In: (2014). Ed. by David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars, pp. 154–169 (cit. on p. 16).

[4] Francisco Massa, Mathieu Aubry, and Renaud Marlet. "Convolutional Neural Networks for joint object detection and pose estimation: A comparative study." In: *CoRR* abs/1412.7190 (2014). arXiv: 1412.7190 (cit. on p. 12).

[5] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition." In: *arXiv e-prints*, arXiv:1409.1556 (Sept. 2014), arXiv:1409.1556. arXiv: 1409.1556 [cs.CV] (cit. on p. 13).

[6] Shubham Tulsiani and Jitendra Malik. "Viewpoints and Keypoints." In: *CoRR* abs/1411.6067 (2014). arXiv: 1411.6067 (cit. on pp. 12, 13).

[7] Yu Xiang, Roozbeh Mottaghi, and Silvio Savarese. "Beyond PASCAL: A Benchmark for 3D Object Detection in the Wild." In: (2014) (cit. on p. 12).

[8] Angel X. Chang, Thomas Funkhouser, Leonidas Guibas, Pat Hanrahan, Qixing Huang, Zimo Li, Silvio Savarese, Manolis Savva, Shuran Song, Hao Su, Jianxiong Xiao, Li Yi, and Fisher Yu. *ShapeNet: An Information-Rich 3D Model Repository*. Tech. rep. arXiv:1512.03012 [cs.GR]. Stanford University — Princeton University — Toyota Technological Institute at Chicago, 2015 (cit. on pp. 12, 23, 24, 41).

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep Residual Learning for Image Recognition." In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385 (cit. on pp. 29, 47).

[10] Alex Kendall, Matthew Grimes, and Roberto Cipolla. "Convolutional networks for real-time 6-DOF camera relocalization." In: *CoRR* abs/1505.07427 (2015). arXiv: 1505.07427 (cit. on p. 12).

[11] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. "ImageNet Large Scale Visual Recognition Challenge." In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252 (cit. on pp. 12, 29, 47, 77).

[12] Hao Su, Charles Ruizhongtai Qi, Yangyan Li, and Leonidas J. Guibas. "Render for CNN: Viewpoint Estimation in Images Using CNNs Trained with Rendered 3D Model Views." In: *CoRR* abs/1505.05641 (2015). arXiv: 1505.05641 (cit. on pp. 12, 13, 50).

[13] Christopher B Choy, Danfei Xu, JunYoung Gwak, Kevin Chen, and Silvio Savarese. "3D-R2N2: A Unified Approach for Single and Multi-view 3D Object Reconstruction." In: (2016) (cit. on p. 23).

[14] *Intersection over Union (IoU) for object detection*. 2016. URL: https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/ (visited on 06/27/2021) (cit. on p. 32).

[15] M. A. Rahman and Yang Wang. "Optimizing Intersection-Over-Union in Deep Neural Networks for Image Segmentation." In: *ISVC*. 2016 (cit. on p. 31).

[16] Helge Rhodin, Nadia Robertini, Christian Richardt, Hans-Peter Seidel, and Christian Theobalt. "A Versatile Scene Model with Differentiable Visibility Applied to Generative Pose Estimation." In: (2016). arXiv: 1602.03725 [cs.CV] (cit. on pp. 17, 18).

[17] Jiajun Wu, Chengkai Zhang, Tianfan Xue, William T Freeman, and Joshua B Tenenbaum. "Learning a probabilistic latent space of object shapes via 3d generative-adversarial modeling." In: (2016), pp. 82–90 (cit. on p. 22).

[18] Hiroharu Kato, Yoshitaka Ushiku, and Tatsuya Harada. "Neural 3D Mesh Renderer." In: (2017). arXiv: 1711.07566 [cs.CV] (cit. on pp. 16, 17, 21, 25, 27, 28).

[19] Siddharth Mahendran, Haider Ali, and René Vidal. "3D Pose Regression using Convolutional Neural Networks." In: *CoRR* abs/1708.05628 (2017). arXiv: 1708.05628 (cit. on p. 12).

[20] Roberto Opromolla, Giancarmine Fasano, Giancarlo Rufino, and Michele Grassi. "A review of cooperative and uncooperative spacecraft pose determination techniques for close-proximity operations." In: *Progress in Aerospace Sciences* 93 (2017), pp. 53–72. ISSN: 0376-0421 (cit. on pp. xxi, 1).

[21] Georgios Pavlakos, Xiaowei Zhou, Aaron Chan, Konstantinos G. Derpanis, and Kostas Daniilidis. "6-DoF Object Pose from Semantic Keypoints." In: *CoRR* abs/1703.04670 (2017). arXiv: 1703.04670 (cit. on p. 14).

[22] Charles R. Qi, Hao Su, Kaichun Mo, and Leonidas J. Guibas. "PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation." In: (2017). arXiv: 1612.00593 [cs.CV] (cit. on pp. 21, 22).

[23] Mahdi Rad and Vincent Lepetit. "BB8: A Scalable, Accurate, Robust to Partial Occlusion Method for Predicting the 3D Poses of Challenging Objects without Using Depth." In: *CoRR* abs/1703.10896 (2017). arXiv: 1703.10896 (cit. on p. 14).

[24] Yu Xiang, Tanner Schmidt, Venkatraman Narayanan, and Dieter Fox. "PoseCNN: A Convolutional Neural Network for 6D Object Pose Estimation in Cluttered Scenes." In: *CoRR* abs/1711.00199 (2017). arXiv: 1711.00199 (cit. on pp. 13, 14).

[25] Xinchen Yan, Jimei Yang, Ersin Yumer, Yijie Guo, and Honglak Lee. "Perspective Transformer Nets: Learning Single-View 3D Object Reconstruction without 3D Supervision." In: (2017). arXiv: 1612.00814 [cs.CV] (cit. on p. 24).

[26] Blender Online Community. *Blender - a 3D modelling and rendering package*. Blender Foundation. Stichting Blender Foundation, Amsterdam, 2018 (cit. on p. 24).

[27] Nanyang Wang, Yinda Zhang, Zhuwen Li, Yanwei Fu, Wei Liu, and Yu-Gang Jiang. "Pixel2Mesh: Generating 3D Mesh Models from Single RGB Images." In: (2018). arXiv: 1804.01654 [cs.CV] (cit. on pp. 22, 28).

[28] Yi Zhou, Connelly Barnes, Jingwan Lu, Jimei Yang, and Hao Li. "On the Continuity of Rotation Representations in Neural Networks." In: *CoRR* abs/1812.07035 (2018). arXiv: 1812.07035 (cit. on pp. 13, 60).

[29] Bo Chen, Jiewei Cao, Alvaro Parra, and Tat-Jun Chin. *Satellite Pose Estimation with Deep Landmark Regression and Nonlinear Pose Refinement*. 2019. arXiv: 1908.11542 [cs.CV] (cit. on p. 79).

[30] Wenzheng Chen, Jun Gao, Huan Ling, Edward J. Smith, Jaakko Lehtinen, Alec Jacobson, and Sanja Fidler. "Learning to Predict 3D Objects with an Interpolation-based Differentiable Renderer." In: (2019). arXiv: 1908.01210 [cs.CV] (cit. on pp. 18, 19, 21, 23–25, 27, 28, 80).

[31] Long Hoang, Suk-Hwan Lee, Oh-Heum Kwon, and Ki-Ryong Kwon. "A Deep Learning Method for 3D Object Classification Using the Wave Kernel Signature and A Center Point of the 3D-Triangle Mesh." In: *Electronics* 8.10 (2019). ISSN: 2079-9292 (cit. on p. 4).

[32] Krishna Murthy Jatavallabhula, Edward Smith, Jean-Francois Lafleche, Clement Fuji Tsang, Artem Rozantsev, Wenzheng Chen, Tommy Xiang, Rev Lebaredian, and Sanja Fidler. "Kaolin: A PyTorch Library for Accelerating 3D Deep Learning Research." In: (2019). arXiv: 1911.05063 [cs.CV] (cit. on p. 21).

[33] Mate Kisantal, Sumant Sharma, Tae Ha Park, Dario Izzo, Marcus Märtens, and Simone D'Amico. "Satellite Pose Estimation Challenge: Dataset, Competition Design and Results." In: *CoRR* abs/1911.02050 (2019). arXiv: 1911.02050 (cit. on p. 69).

[34] Shichen Liu, Tianye Li, Weikai Chen, and Hao Li. "Soft Rasterizer: A Differentiable Renderer for Image-based 3D Reasoning." In: (2019). arXiv: 1904.01786 [cs.CV] (cit. on pp. 17, 21, 25, 27).

[35] Shichen Liu, Tianye Li, Weikai Chen, and Hao Li. "Soft Rasterizer: A Differentiable Renderer for Image-based 3D Reasoning." In: *CoRR* abs/1904.01786 (2019). arXiv: 1904.01786 (cit. on p. 22).

[36] Lars Mescheder, Michael Oechsle, Michael Niemeyer, Sebastian Nowozin, and Andreas Geiger. "Occupancy Networks: Learning 3D Reconstruction in Function Space." In: (2019). arXiv: 1812.03828 [cs.CV] (cit. on p. 22).

[37] Pedro F. Proenca and Yang Gao. "Deep Learning for Spacecraft Pose Estimation from Photorealistic Rendering." In: (2019). arXiv: 1907.04298 [cs.CV] (cit. on pp. 74, 76).

[38] Sumant Sharma and Simone D'Amico. "Pose Estimation For Non-Cooperative Spacecraft Rendezvous Using Neural Networks." In: (2019) (cit. on pp. 33, 49, 72).

[39] Edward J. Smith, Scott Fujimoto, Adriana Romero, and David Meger. "GEOMetrics: Exploiting Geometric Structure for Graph-Encoded Objects." In: (2019). arXiv: 1901.11461 [cs.CV] (cit. on p. 22).

[40] The GIMP Development Team. *GIMP*. Version 2.10.12. June 12, 2019 (cit. on p. 35).

[41] Jiawei Zhang and Lin Meng. "GResNet: Graph Residual Network for Reviving Deep GNNs from Suspended Animation." In: (2019). arXiv: 1909.05729 [cs.LG] (cit. on p. 22).

[42] Hiroharu Kato, Deniz Beker, Mihai Morariu, Takahiro Ando, Toru Matsuoka, Wadim Kehl, and Adrien Gaidon. "Differentiable Rendering: A Survey." In: (2020). arXiv: 2006.12057 [cs.CV] (cit. on pp. 15, 24).

[43] *Kelvins - ESA's Advanced Concepts Competition Website*. 2020. URL: https://kelvins.esa.int/ (visited on 06/27/2021) (cit. on pp. 56, 76).

[44] *Pose Estimation Challenge post mortem*. 2020. URL: https://kelvins.esa.int/pose-estimation-challenge-post-mortem/ (visited on 06/27/2021) (cit. on pp. xxi, xxii, 1, 2, 33, 55, 76).

[45] Nikhila Ravi, Jeremy Reizenstein, David Novotny, Taylor Gordon, Wan-Yen Lo, Justin Johnson, and Georgia Gkioxari. "Accelerating 3D Deep Learning with PyTorch3D." In: *arXiv:2007.08501* (2020) (cit. on pp. 6, 22, 26, 31, 50, 52, 57, 70).

[46] Caner Sahin, Guillermo Garcia-Hernando, Juil Sock, and Tae-Kyun Kim. "A Review on Object Pose Recovery: from 3D Bounding Box Detectors to Full 6D Pose Estimators." In: (2020). arXiv: 2001.10609 [cs.CV] (cit. on pp. xxi, 1).

[47] *Detectron2: A PyTorch-based modular object detection library*. 2021. URL: https://github.com/facebookresearch/detectron2 (cit. on pp. 50, 54).

[48] *Clipping (computer graphics) - Wikipedia*. URL: https://en.wikipedia.org/wiki/Clipping_(computer_graphics) (visited on 06/27/2021) (cit. on p. 7).

[49] *Gimbal lock - Wikipedia*. URL: https://en.wikipedia.org/wiki/Gimbal_lock (visited on 06/27/2021) (cit. on p. 9).

[50] *Himawari-8 and 9*. URL: https://directory.eoportal.org/web/eoportal/satellite-missions/h/himawari-8-9 (visited on 06/27/2021) (cit. on p. 34).

[51] *PRISMA (Prototype Research Instruments and Space Mission technology Advancement)*. URL: https://directory.eoportal.org/web/eoportal/satellite-missions/p/prisma-prototype (visited on 06/27/2021) (cit. on p. 33).