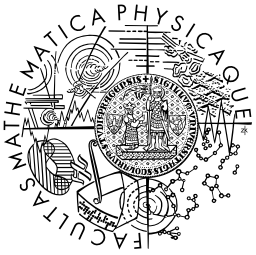

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

**MASTER THESIS**

Tomáš Kremel

# Evaluating Point Cloud Rendering Approaches for Camera Pose Verification

Department of Software and Computer Science Education

Supervised by: doc. Ing. Tomáš Pajdla, Ph.D.
Torsten Sattler, Dr. rer. nat.

Study programme: Computer Science

Study branch: Artificial Intelligence

Prague 2024

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date .....  .....

Author's signature

I would like to express my gratitude and appreciation to my supervisors, for their suggestions, great support, and inexhaustible patience while I was finding motivation to put this work into reality. I would also like to thank The Czech Institute of Informatics, Robotics, and Cybernetics (CIIRC) for providing me with all the necessary GPU resources required for computations. Lastly, I would like to recognize my friends Tomáš Souček and Vojtěch Kužel for their consultations.

Title: Evaluating Point Cloud Rendering Approaches for Camera Pose Verification

Author: Tomáš Kremel

Department: Department of Software and Computer Science Education

Supervisor: doc. Ing. Tomáš Pajdla, Ph.D., Department of Algebra, Torsten Sattler, Dr. rer. nat.

Abstract: Visual localization is the problem of estimating the 6 degrees of freedom camera pose from which a query image was taken relative to a known reference scene representation. It is the key for applications such as Augmented, Mixed, and Virtual Reality, as well as autonomous robotics such as drones or self-driving cars.

This thesis focuses on a visual localization pipeline, especially on its pose verification and reranking step. The pipeline uses 3D point clouds and 2D-3D correspondences between the query image and 3D scene points for candidate camera poses estimations. The thesis explores point cloud rendering approaches as they are utilized in the pipeline and the verification step—the render of the discretized scene from a given candidate position is compared to the actual query image to asses if the given couple depicts the same place.

One of the main challenges of such rendering is occlusion handling. Due to the sparsity of points employed for otherwise continuous real world representation, information about what lies in the front and what is hidden can be easily lost when projected to the 2D image. Rendering approaches explored in this thesis focus on the challenge directly or as a component of a novel view synthesis DNN-based renderer. Rendering influence on localization performance is investigated.

Keywords: point clouds, rendering, neural rendering, localization

Název práce: Použití metod zobrazení mračna bodů pro ověření polohy kamery

Autor: Tomáš Kremel

Katedra: Katedra softwaru a výuky informatiky

Vedoucí diplomové práce: doc. Ing. Tomáš Pajdla, Ph.D., Katedra algebry, Torsten Sattler, Dr. rer. nat.

Abstrakt: Vizuální lokalizace je problém odhadování parametrů šesti stupňů volnosti pozice kamery, z níž byla pořízena dotazovaná fotografie, přičemž pozice je vztažena ke známé reprezentaci referenčního prostředí. Řešení tohoto problému je klíčové v aplikacích jako jsou rozšířená, smíšená a virtuální realita, stejně tak v oblasti autonomní robotiky zahrnující drony a samořiditelné automobily.

Tato práce se soustředí na vizuální lokalizačního algoritmus, zejména na jeho verifikační a přeřazovací krok. Tento algoritmus interně využívá tří dimenzionální mračna bodů a hledání korespondencí mezi těmito body a dotazovanou fotografií pro nalezení odhadů kandidátních pozic kamery. Práce zkoumá přístupy k renderování mračen bodů a jejich využití v rámci algoritmu a jeho verifikačního kroku – render diskretizovaného prostředí z konkrétní kandidátní pozice se v něm porovnává s danou dotazovanou fotografií za účelem určení toho, zda oba pohledy zobrazují to samé místo.

Jedna z hlavních výzev renderingu diskretizovaného prostředí jsou okluze. Kvůli řídkosti bodů využitých jako reprezentace jinak spojitého reálného světa může být informace o tom co leží v popředí a co v pozadí lehce ztracena při promítnutí bodů na dvou dimenzionální obraz. Přístupy k renderování zkoumané v této práci se soustředí na renderování bodů přímo nebo jako komponentu rendereru "nových pohledů" využívající hlubokých neuronových sítí. Je zde prověřen vliv těchto renderovacích přístupů na přesnost lokalizace.

Klíčová slova: mračna bodů, rendering, neurální rendering, lokalizace

# Contents

# Introduction

In recent years, robots and other technological aids have been spreading into more and more areas of human endeavor. Nowadays, people encounter such machines in public spaces like museums, airports, hospitals, and others.[1] Instead of just moving inside of a stationary protective box in a manufacturing facility, public spaces place high demands on the accuracy of localizing where the robot is, as the cost of a mistake can be high.

The problem of visual localization can be described as a task of finding the position of a camera that took a query photo relative to a reference scene representation.[2] By the position in visual localization 6 parameters are meant (degrees of freedom—DoF), 3 of which are an absolute position in reference coordinate system and the rest are the orientation of the camera. A scene in general visual localization is a set of RGB images, possibly associated with per-pixel depth information (denoted as RGBD). In the settings of this thesis, an "explicit" scene representation is used, unless otherwise said. In other words, a 3D colored point cloud obtained from the "implicit" representation based wholly on images [Tewari et al., 2020] is used.

As indicated, a solution to the problem is of great importance in autonomous robotics, namely self-driving cars, terrestrial or aerial drones, and robots, also in augmented, mixed or virtual reality applications. All of these examples interact through various means with the surrounding environment suggesting that they are dependent upon location.

In our day-to-day lives, people typically meet so-called network-based positioning algorithms based on measuring radio signals from various sources. These sources include Wi-Fi, Bluetooth Low Energy (BLE), Global Navigation Satellite Systems (GNSS, such as GPS, Galileo), and cellular networks [Trogh et al., 2019]. All of these positioning data sources suffer from shared limitations—they can provide only 3 degrees of freedom position estimation, and the accuracy of such estimation can vary a lot. Degradation outdoors comes from signal blockage or reflection due to high obstacles near the query position, solar storms, and indoors from signal damping through walls. Under the best circumstances for a single query, GNSS can result in positioning within a few centimeters,[3] other methods can provide an estimate with uncertainty measured in even hundreds of meters. Network-based methods find usage where an exact position is not that crucial, such as in asset tracking, analyzing traffic patterns, transportation planning, security, surveillance, and population movements tracking during disasters as a more specific example [Trogh et al., 2019].

Network-based localization methods typically build upon the triangulation of a few measurements of time-of-flight or signal strength from several sources, such as satellites, cell towers, and access points, possibly combined with other techniques to reach better accuracy. On the other hand, visual localization uses the rich visual information encoded in images to estimate the source camera's pose.

[1] `https://spring-h2020.eu/about-spring`

[2] `https://www.visuallocalization.net`

[3] `https://www.gps.gov/systems/gps/performance/accuracy`

One family of localization approaches that can be easily compared to network-based methods relies on establishing an ideally high number of correspondences between features of a query image and those of the scene representation. These correspondences are then used to estimate the query image's camera pose with respect to the scene, typically more robustly due to the large amount of them. Arguably, besides the different numbers of DoF these methods output, visual localization may provide more reliable and accurate poses.

The applications mentioned require higher accuracy, 6 DoF positions, and fairly often they require good performance either directly indoors or at least working reliably under various troublesome conditions for network-based methods. The visual localization has the potential to fulfill these requirements accurately up to a few centimeters and degrees.[4]

Visual localization brings its own challenges stemming mainly from the scene representation. 1) Illumination changes caused by artificial lighting and changes of the night and day in the scene representation data, especially in query images, pose a significant problem for both indoor and outdoor localization. The former type is a more challenging problem than the latter, as indoors more complexities arise. 2) Furniture, equipment, and people add high dynamics as they move significantly more than objects in urban settings. Also, previously unseen objects may be added to the environment. All of these may cause occlusions of possibly essential parts of the scene from the localization perspective. 3) Moreover, most areas in buildings that stay the same (walls, floors) are largely textureless, which is problematic for many feature-extracting approaches. Combined with dynamically moving objects that are, on the contrary, more likely to be textured and thus resulting in better features to be worked with, feature matches are often clustered in small unreliable areas leading to unstable pose estimates. 4) Even in cases when features are more reliable, another issue arises as interiors are frequently highly symmetric with repetitive elements on large (corridors, rooms, ...) and small (chairs, tables, doors, ...) scales. 5) Finally, due to inherently smaller distances measured in interiors compared to outdoors, a slight change in viewport leads to a substantial change in resulting view [Taira et al., 2018].

Traditionally, camera position estimates can be computed during Structure-from-Motion (SfM) [Schönberger and Frahm, 2016] or (Visual) Simultaneous Localization and Mapping, (V)SLAM, see Durrant-Whyte and Bailey [2006]. The former is being used for creating a 3D point cloud model of a given scene from reference photos alongside pose estimations of reference cameras relative to the model built and nowadays may rather be used solely for 3D model creation, alongside modern 3D scanners, such as LiDAR (an acronym for "light detection and ranging"). The latter is typically used by mobile robots moving within a particular environment. When computed in real-time, the SfM and the VSLAM processes are equivalent, sharing the same technical aspects and functionalities [Yasuda et al., 2020]. Currently, state-of-the-art approaches use matches between a 2D image and a 3D environment model, producing several candidate poses for a query image. Computed poses are verified and ranked based on sparse evidence, e.g., on the number of inlier matches found by, e.g., RANSAC algorithm [Fischler and Bolles, 1981]. Due to the inherent candidate pose instability and imperfections mentioned in the previous paragraph caused by the presence of repeated

[4] `https://www.visuallocalization.net/benchmark`

Figure 1: A render of one of the datasets' point cloud. We can see an issue with using OpenGL's point primitive. Closer surfaces (the column, for instance) to the camera are not continuous but have holes and points that should be occluded by closer surfaces being visible. Fixed screen space point primitive is thus not a reliable way of rendering a point cloud without per-render pre-computations to set the correct pixel primitive.

structures and global ambiguities, a (weighted) inlier count has been shown as not a good decision criterion [Sattler et al., 2016]. Because of that, InLoc paper authors [Taira et al., 2018] proposed to compare the query photo against a render of the scene from the candidate pose. The paper showed the pixel-wise comparison to lead to a significantly better overall pose accuracy.

This thesis focuses on the verification step of the InLoc localization pipeline as realistic rendering of point clouds[5] is a problem of its own. Arguably, the most widespread solution for a point cloud rendering is the usage of the modern graphics APIs native primitive, such as `GL_POINTS` for OpenGL and WebGL. From a realistic rendering standpoint, this poses a problem because "points are rasterized as screen-aligned squares of a given window-space size"[6]. Since this primitive has a fixed size in the screen space, rendering of, for instance, a long hallway view results in nonuniform "surface" coverage. It does not matter whether the point cloud has a uniform spatial point density produced by a LiDAR sensor or a nonuniform one produced by SfM. Walls farther from the camera may seem like a solid surface, whereas closer ones can contain holes. The closer to the perspective camera, the bigger the viewing angle of a point pair separated by a fixed-length line segment perpendicular to the viewing direction is. With a specific distance-independent screen space point dimension, it happens that the two point primitives visually split with no overlap at a certain distance from the camera, see Fig. 1. Transversely, such holes may influence pixel-wise compari-

[5]Point clouds are being used here primarily because they are the direct output of SfM or triangulation of RGBD photo database and creation of a mesh from such a point cloud is an unnecessary computational burden with an uncertain result when used without human intervention.

[6]`https://www.khronos.org/opengl/wiki/Primitive`

son in the pose verification step in InLoc as per-pixel computation of descriptor distances between the render and the real query image would either be skipped because of missing 3D structure in the render or would result in the usage of a bad descriptor of a point that should not be visible as it should be occluded by closer points ("surfaces") with, in general, different colors and neighborhoods.

The thesis explores three approaches to tackle the problem of realistic point cloud rendering. 1) Neural rendering approach that uses a deep generative model to deal with occlusions, 2) a rendering approach called ray marching paired with modeling points as spheres having the source point's color and radius equal to the distance to the nearest neighbor to avoid holes in the resulting renders of flat surfaces, and 3) surface splatting algorithm [Zwicker et al., 2001] that models points as oriented colored disks with radii determined the same way as for ray marching-based renderer. Several evaluations and experiments are performed for all the renderers and applied to the InLoc dataset from the original InLoc article, a dataset from the ARTwin project,[7] and the Phototourism dataset.[8]

# Thesis Contribution

In the thesis, we generalize the original InLoc implementation to tackle a general dataset format as the original source codes are hardcoded towards the InLoc dataset. We also add generation of proxy data objects needed for InLoc runtime missing in the original implementation and provide generation scripts for the dataset format out of all three dataset types we study in the thesis and on which we demonstrate these capabilities.

We supply all the necessary transformations between various data formats used by the neural model and other renderers. We utilize GPU-based C++ & OpenGL surface splatting implementation and enhance it with the ability to render headlessly (without a window) both RGB and depth information based on views' camera parameters and a point cloud file. To our best knowledge, such a headless surface "splatter" did not exist before. We also provide GPU-based C++ & CUDA & OpenGL ray marching renderer with both headless and windowed FPS camera modes with the same dataset generation component shared with the previous splatter. As for the splatter, such implementation was not available before, considering the performance. Both C++ renderers are shown to process tens of millions of points in a competitive time.

The statistical, visual, and computational comparisons of renderers' performance are performed, showing the benefits of the splatter and marcher over the baseline, boosting localization as a whole, and the pose verification step solely. We also show that these renderers lead to better training of the neural model with notably better scores for all compared areas. Furthermore, we show the benefits of neural rendering in the pose verification step of the InLoc pipeline as it further pushes localization scores for a query image set.

Based on the results, the pros and cons of all approaches are outlined. On top of these considerations, we propose which approach to use for varying problem settings. See the Conclusion chapter for more details.

[7] `https://artwin-project.eu`

[8] `https://www.cs.ubc.ca/research/image-matching-challenge/2021/data`

## Thesis Structure

The thesis is divided into three chapters. The *first chapter*, Visual Localization, presents the visual localization methods and presents the InLoc localization pipeline and the reason why the method is chosen for further investigation. The *second chapter*, Point Cloud Rendering, presents the theory behind rendering points and how it compares to rendering triangles, it introduces various traditional methods for visualizing points as well as neural rendering techniques with their classification. Finally, the chapter presents in detail the Neural Rerendering In the Wild, Surface Splatting, and ray marching with signed distance fields methods that are further examined in the *third chapter*, Camera Pose Verification. This last chapter then gets to experiments conducted through the description of datasets examined and notes on the implementation of the localization pipeline, all renderers, and transformations of data to formats expected by all components. For the experiments, the comparison of renderers is carried out, and their influence on localization is examined.

Enclosed to the thesis, there are source codes for all renderers, evaluations, and datasets' transformations. Repositories for these can be found also at Github: `https://github.com/Auratons/inloc`, `https://github.com/Auratons/neural_rendering`, `https://github.com/Auratons/renderer_ray_marching`, and `https://github.com/Auratons/renderer_surface_splatting`.

# 1. Visual Localization

As stated in Introduction, visual localization is the task of finding the position of a camera that took a query photo relative to a reference scene representation, and it is one of the fundamental problems in computer vision.

Compared to network-based localization methods, such as GNSS, visual localization, even though being able to work in network-denied environments, comes with its own set of problems that any successful method must consider. For both outdoor and indoor localization, to which the field is typically separated due to different localization complexity, illumination changes throughout the day and artificial lighting influence present in the environment's representation data pose one class of such problems. Further, it must cope with transient dynamic objects that can be present in both query and database data, possibly occluding important feature-rich areas but having nothing to do with the long-term visual appearance of the given location. Outdoors, seasonal and weather-caused changes must be handled as well. For indoors, more problems stem from textureless areas such as walls, ceilings, and floors; from repetition and symmetry on both the global level with corridors, for example, and the local level, such as door handles. Also, compared to outside, with typically longer distances between objects, inside small change of viewing position leads to a vastly different view.

In this chapter, we present previous work on the matter and describe the InLoc method explored in the thesis, explaining why this very method is chosen.

## 1.1 Related work

There are three main method categories for visual localization, as of Torsten Sattler [2018], Sattler et al. [2019a], Sattler et al. [2021], Humenberger et al. [2021]: methods based on structure, image retrieval, and pose regression.

#### Structure-based methods

Structure-based methods are the traditional way of estimating poses where a 3D model (the *structure*) is pre-created in order to later find 2D-3D correspondences.

The 3D model is typically created by Structure-from-Motion [Schönberger and Frahm, 2016; Schönberger et al., 2016], by computing local sparse features (keypoints with descriptors, Se et al. [2002] used SIFT [Lowe, 2004] descriptor, Robertson and Cipolla [2004] was pre-SIFT, using image rectification) per database image with known focal length, match them against each other across images, and triangulate resulting 3D points from these matches. Since the model already contains pre-computed features, matching against a query image's features can then be performed. Since the 2D-3D matches are determined, the camera pose is computed using the perspective-n-point (PNP) solver [Kneip et al., 2011]. Because of the possible presence of outlier matches, a RANSAC loop [Fischler and Bolles, 1981] is utilized to increase robustness. Other examples of this approach are Sattler et al. [2011, 2017].

With the growing size of a 3D point cloud, the runtime gets prolonged. To mitigate matching speed deterioration, these methods also get paired with image

retrieval, described next, to find the most relevant images from the SFM model. Examples of these methods are Taira et al. [2018, 2019]; Peng et al. [2021].

### Image retrieval-based methods

Image retrieval can be used to speed up the structure-based method family and make mapping and localization more robust. That is because the restriction of matching to the parts of the scene visible in the given query photo helps to avoid global ambiguities in the scene, e.g., caused by similar structures found in unrelated parts of a scene [Taira et al., 2019]. It can also be used on its own for a closely related task to visual localization called place recognition, which strives to find the approximate location of a query photo within a database of geo-tagged images. Unlike visual localization, place recognition does not need an explicit model representation, so no depth values nor point clouds are necessary for these methods to work. Because of less input information, the location obtained by the retrieval and interpolating of several geo-tags or camera poses is, in general, less accurate [Torii et al., 2011; Sattler et al., 2019b].

In both cases, the goal is to gather a set of images that are the most similar according to a selected criterion—here, a retrieved image is considered relevant if it sees the same scene—followed by an optional re-ranking step. Historically, image retrieval methods have used variants of Bag of Visual Words [Sivic and Zisserman, 2003] and Vector of Locally Aggregated Descriptors (VLAD) [Jégou et al., 2010], newer approaches utilize features extracted by a Deep Neural Network (DNN) as such features encode high-level semantics better than sparse features such as SIFT [Gordo et al., 2016; Kendall and Cipolla, 2017; Hausler et al., 2021].

### Pose regression-based methods

This category of methods uses a DNN for regressing the query pose end-to-end from an RGB image directly to a 6 DoF pose. Based on the assumption that features obtained by a Neural Network (NN) trained for a general vision task also include some helpful information for pose estimation, transfer learning is leveraged for the pose regression.

PoseNet [Kendall et al., 2015] is an example of such an approach using an image classification CNN architecture, like VGGNet or ResNet, with fully connected layers to regress the pose at the end of the architecture. Regression-based methods are generally less accurate than structure-based localization (for PoseNet, by order of magnitude). However, their advantage lies in short, constant inference time and smaller memory and computation power requirements using just a single forward pass, even without requiring the camera intrinsics parameters, which may be inaccurate and unavailable [Shavit and Keller, 2022]. The accuracy problem is inherent here, as end-to-end learning imposes a tight coupling with the database coordinates. Thus, such a network can be seen as a compressed version of the database itself, which limits the generalization power of the network [Humenberger et al., 2021]. As the approach is still interesting for other use cases, many improvements were presented, such as Kendall and Cipolla [2017]; Brahmbhatt et al. [2017]; Valada et al. [2018] and Radwan et al. [2018].

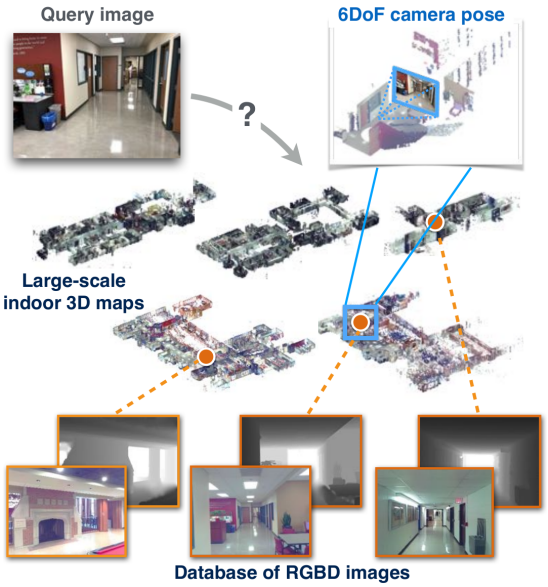


Figure 1.1: Given a database of geometrically-registered RGBD images, InLoc predicts the 6 DoF camera pose of a query RGB image by retrieving candidate images, estimating candidate camera poses, and selecting the best matching camera pose. Image taken from Taira et al. [2018].

## 1.2 InLoc

This visual localization method, so-called *Indoor Visual Localization with Dense Matching and View Synthesis*, falls amid two-staged structure-based approaches combined with image retrieval. The first stage finds correspondences between a query image features and model of a scene, and the second estimates the camera pose. The method's input is a database of RGBD images with known focal lengths (from EXIF data, for instance), and the method internally uses a point cloud 3D scene representation. The method focuses on indoor localization and addresses several issues presented in Introduction. Visual representation of the method with a short summary can be seen in Fig. 1.1.

All of illumination changes (1), textureless areas (3) leading to lack of sparse local features, such as SIFT [Lowe, 2004], repetitive elements in indoor settings (4) leading to similar repetitive features being produced, and even viewpoint changes (5) are overcome by utilizing multi-scale dense CNN features computed densely on a regular grid by NetVLAD [Arandjelović et al., 2015]. These features are used for database image retrieval, as $N = 100$ best matching images are chosen based on sorted normalized L2 distances of the extracted database feature vectors and the query feature vector.

In the next stage, candidate images are re-ranked by another feature matching in the geometric verification process and pose estimation. Firstly, features are extracted by VGG [Simonyan and Zisserman, 2014] model on conv5 and subsequently on conv3 layer restricted by previously found matches are used for finding geometrically consistent sets of correspondences with RANSAC [Fischler and Bolles, 1981]. Based on the number of RANSAC inliers, top $M = 10$ candidate database images are kept. It is to be noted that these features are obtained with no additional computation burden as VGG is used internally by NetVLAD.

As database images used as input to the method are RGBD and hence they have associated 3D points, the query camera pose is then estimated by finding pixel-to-pixel correspondences between the query and the top $M$ database images followed by P3P-LO-RANSAC [Lebeda et al., 2012].

To further cope with self-similarity found in indoor locations, counting the number of inliers as positive evidence to decide whether two views are taken from an exact location is not the only decisive criterion. Negative evidence is also used in the form of the portion of the view rendered from the candidate query pose that does not match the query photo. Authors of the paper refer to this as *explicit pose estimate verification based on view synthesis*. Verification is done pixel-wise to obtain consistent and inconsistent pixels between the render and the query photo. To keep invariance to illumination changes and small misalignments, pixel comparison operates with RootSIFT local patch descriptors [Arandjelović and Zisserman, 2012]. The final image-render similarity is the median of descriptor distance across the entire image while ignoring areas with missing 3D structure resulting in background-filled regions in renders.

This thesis further examines the view synthesis part of the verification process as it changes the rendered source imputed to the RootSIFT descriptor computation process. In further chapters, three rendering techniques alternative to the baseline `GL_POINTS` rendering approach are described in detail.

The decision to use the InLoc method in the thesis was driven by the fact that it is the first method of its kind, state-of-the-art of its time that puts basis or plays a role of a baseline for other subsequent state-of-the-art methods [Hyeon et al., 2021]. Further, its source codes are public[1], whereas a subsequent paper [Taira et al., 2019] presenting some improvements does not provide source codes.

[1] `http://www.ok.sc.e.titech.ac.jp/INLOC`

# 2. Point Cloud Rendering

As stated in Gross and Pfister [2007], using point primitives for rendering has been driven by two main reasons. Over the years, there was a dramatic increase in the polygonal complexity of models being rendered, leading to the overhead of managing and processing extensive mesh connectivity information. Further, modern 3D scanners (LiDAR, stereo camera setup) or photogrammetry methods (SfM) produce both geometry and appearance of complex, real-world objects in the form of a point cloud. Points in a point cloud play the role of (discrete) building blocks for 3D scenes, similar to how pixels are the digital ones for images.

Points are the simplest graphic primitive, generalizing pixels towards irregular samples of geometry and appearance. They differ from triangles typically used in computer graphics by carrying all attributes needed for processing and rendering with themselves the same way as pixels do. That results in transformation of rendering pipeline, the terms vertex and fragment coincide in one entity. Even though the presence of just one such entity may lead to simpler graphical pipelines, it is not without issues [Schütz et al., 2021].

1) Straightforward points projection leaves empty spaces in the image that need to be filled for close-up views as it may lead to problems with occlusions and visibility or depth perception—with less dense sampling, a render can end up with just many points scattered across the background with no notion of what is closer and farther. Thus, point clouds typically require a denser sampling compared to triangle meshes. 2) Points do not possess any topology or connectivity information. This fact is an advantage and disadvantage at the same time, compared to meshed that contain this type of information, but only as a result of 3D reconstruction algorithms with point clouds being an input that typically still require some prior assumptions on topology and sampling. It is, for instance, possible to stream and render point clouds progressively, and change of topology (e.g., by filtering) is more straightforward than for meshes where one needs to recompute connectivity information [Gross and Pfister, 2007]. On the other hand, effective point processing typically needs elaborate data structures, including KD-trees [Bentley, 1975] or spatial hashing [Gaede and Günther, 1998].

Over the years, many approaches have been devised for processing point primitives and tackling the issues presented in the preceding paragraphs. In the thesis, we take advantage of having point clouds-based datasets. We found out that for large indoor areas, it may be tricky to come up with sufficiently good mesh that can be further used for the localization verification step rendering, as it can be seen in Fig. 2.1. Proceeding with point cloud-based rendering techniques, we work with three of them together with the mentioned baseline `GL_POINTS` approach; see their descriptions in the following sections.

## 2.1 Related Work

Tewari et al. [2020] defined *rendering* as transforming a scene definition, including some of the cameras, lights, surface geometry, and material, into a simulated camera image. The process can be organized in two ways [Marschner and Shirley, 2021]. *Object-order* rendering considers each object; for such, all the pixels it in-

Figure 2.1: A render of one of the meshes created from the raw datasets' point cloud, camera pose is the same as for Fig. 1. The mesh was created semi-manually, meaning that boxes, for instance, were meshed separately and then placed at the right place of the resulting mesh. We can see some disadvantages of using meshes in complex environments. Despite long processing time and laborious manual work, the result is not compelling compared to renders using point primitives.

fluences are found and updated. In *image-order* rendering, the loop goes the other way round, each pixel is considered, and for such, all the objects that influence it are found, and the pixel value is computed. From these two approaches, image-order rendering is simpler to implement and more capable in the effects that can be incorporated and usually (though not always) takes more execution time compared to the second approach. Object-order rendering is also known as *rasterization*, whereas under *image-order* rendering, there are more possible approaches, such as ray-casting and ray-tracing.

Rasterization is typically hardware-accelerated because it has good memory coherence [Tewari et al., 2020], which is also one of the reasons for one of the previous claims about execution speed comparison. (Though modern GPU cards already have hardware support for ray-tracing as well.[1]) The rasterization, as a representative of the object-order methods, requires an explicit scene representation, such as mesh or point cloud, whereas the other methods work with both implicit and explicit representations.

Ray-casting and ray-tracing are, in some sense, orthogonal methods within image-order realm; see Fig. 2.2. Ray-casting computes a ray (coming from the camera center through a specific pixel of the screen) intersections with the representation of the scene to project the scene onto the screen. In ray-tracing, the primary ray is considered to be coming from the scene, through the screen to the camera center, conveying color information gathered from all physics-based interactions of light with objects in the scene. Reflections and refractions are

[1] `https://developer.nvidia.com/rtx/ray-tracing`

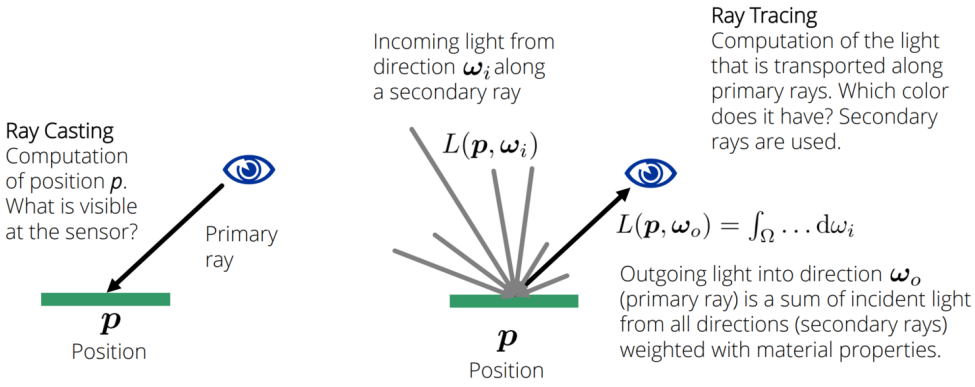


Figure 2.2: A demonstration of the difference between ray-casting and ray-tracing, together with an illustration of the recursive nature of the ray-casting algorithm. Taken from `https://cg.informatik.uni-freiburg.de/course_notes/graphics_01_raycasting.pdf`.

simulated by recursively casting new rays from the intersections with the geometry [Whitted, 1979]. The advantage of this rendering process is the realism of the simulation of real-world optical effects. While rasterization and ray-casting are a simple, one-way processes, ray-tracing is an inherently recursive problem. Hence it is a more complex task.

For rendering points with the classical approaches, surface splatting was proposed as a forward-projection approach that uses a z-buffer algorithm for visibility resolution of points that are exchanged for oriented ellipses. Splatting can process point clouds without additional acceleration data structures such as spatial hierarchies, which are often required in ray-tracing approaches [Gross and Pfister, 2007]. The initial article is mainly a mathematical model and a CPU demonstrator that was later revisited by several papers that enhanced some of its features and ported it to GPUs [Botsch et al., 2005, 2004; Zwicker et al., 2004; Sigg et al., 2006; Weyrich et al., 2007]. Splatting can also be enriched with ray-tracing again to simulate more complex visual effects [Adams et al., 2005]. Also, further GPU-related enhancements were proposed with better data structures suited for the usage [Dachsbacher et al., 2003].

Another family of methods takes a vastly different approach than the classical rendering described above—while traditional computer graphics methods focus on modeling scenes from a physics perspective, simulating light transport and other effects, machine learning can be used for modeling the distribution of real-world imagery. The models utilized for this task are called generative models, successors of the work on *Generative Adversarial Neural Networks* (GANs) [Goodfellow et al., 2014], and can generate high-resolution images [Radford et al., 2016; Brock et al., 2018] or videos [Vondrick et al., 2016; Clark et al., 2019]. More specifically, the field of so-called neural rendering combines generative machine learning techniques with knowledge from classical computer graphics. It is defined as "deep image or video generation approaches that enable explicit or implicit control of scene properties such as illumination, camera parameters, pose, geometry, appearance, and semantic structure" [Tewari et al., 2020]. GANs produce *random*,

realistically-looking images that resemble the training set [Ruthotto and Haber, 2021] statistically. As the definition of neural rendering states, user controllability is important—if used by an artist, outputs reflecting design ideas are preferred over some random imagery. For applications in the neural rendering field, GANs thus needed to be extended by the conditioning of output to enable guidance of the rendering process.

Further citing Tewari et al. [2020], neural rendering techniques can be classified along different axes:

- **Control**. This axis distinguishes neural rendering approaches based on what properties from the definition are controllable and how they condition the network's output. A general solution enabling to control everything is an open research problem. Typically only a subset of controllable properties is approached in subproblems like novel view synthesis, relighting, or face and bodies animation. The conditioning can be performed by passing the scene parameters as input to some network layer or concatenating them to activations of an inner one, by tiling scene parameters over all pixels of an input image resulting in packed input volume, it can also employ an image-to-image transformation DNN that fuses "guide image" into to the output one. Also, a more traditional approach uses scene parameters as an input to a graphical layer.

- **Computer Graphics Modules**. The separation along this axis is based on how much of the classical rendering pipeline is integrated into the specific method. The simplest way to achieve that is to use a non-differentiable computer graphics (CG) component in the network architecture, which would present the render as an input to subsequent differentiable layers of a given architecture. When the module is at the beginning of the architecture, the task transforms into well-researched image-to-image translation. Fully differentiable CG modules also exist.

- **Explicit vs. Implicit Control**. Here, the criterion is based on a type of control signal. Explicit control from a user perspective means manual editing capability of scene parameters in a semantically meaningful manner. By implicit control, a representative sample as input is meant. The difference also translates to training data as explicit control needs richer annotations, whereas implicit one performs well with less supervision.

- **Multi-modal Synthesis**. Not only from an art perspective, often it is beneficial to have multiple outputs from which a user can choose. Especially when only a subset of scene properties is controllable, within the rest, there lies an output space of possible results from which a given model can sample. This sampling capability adds complexity to the architecture, requiring some stochasticity or structured variance built-in, leading to GAN or variational auto-encoders (VAEs) variants.

- **Generality**. Does the rendering approach perform well over multiple scenes or objects without retraining the underlying model? Object-specific approaches still produce higher quality outputs at the cost of lengthy per-instance retraining. General models are still an open research area.

Methods spread across this classification landscape solve various subtasks of the neural rendering field. Given the model this thesis utilizes, the novel view synthesis task is described next.

*Novel view synthesis* generates a view of a scene, represented by a fixed set of input images, from previously unexplored camera poses. Challenges tied to this task are inferring the scene's 3D representation, given sparse observations in the form of images and deducing of occluded or unseen areas of the scene. For the scene reconstruction, the aforementioned SfM is being utilized, followed by MultiView Stereo (MVS) Furukawa and Ponce [2010]; Schönberger et al. [2016] or variational optimization Hiep et al. [2009].

The classical computer vision approach towards novel view synthesis utilizes so-called image-based rendering (IBR) methods [Debevec et al., 1998; Chaurasia et al., 2013; Debevec et al., 1996; Gortler et al., 1996] where views from new viewports are generated by warping input pixels into the outputs using proxy geometry. These methods are sensitive to the scene database size as IBR may fail with insufficient number of source photos, resulting in ghosting-like artifacts and holes [Tewari et al., 2020]. These approaches also do not handle multiple appearances well [Meshry et al., 2019]. Neural networks and rendering alternative approaches have been proposed to mitigate these issues, such as Hedman et al. [2018]; Eslami et al. [2018]; Meshry et al. [2019]; Pittaluga et al. [2019]; Riegler and Koltun [2020a,b]. These methods build on IBR and image-to-image translation using explicit scene models. Learned implicit scene representation can be leveraged as well, see the Neural Radiance Fields Martin-Brualla et al. [2020]; Mildenhall et al. [2020] and Sitzmann et al. [2019].

The Neural Radiance Fields, together with the Gaussian Splatting neural model [Kerbl et al., 2023] combining the neural field with the splatting idea explored in the thesis represent the recent progress in the field nicely, alleviating some of the issues of the previous models. However, they were not yet available when the work on the thesis was started, they are not considered.

## 2.2 Neural Rerendering In the Wild

According to the neural rendering techniques classification, the *Neural Rerendering In the Wild* (NRIW) method [Meshry et al., 2019] can be shortly described as a method explicitly controlling camera parameters, pose, and illumination, using non-differentiable CG module preprocessing an input, producing multiple modalities, and being scene-specific. More specifically, the authors tackle what they define as *total scene capture* with a deep generative model that can

1. perform novel view synthesis for a given scene,
2. can capture and render various appearances of the scene, e.g., all weather and illumination conditions,
3. and finally, it should understand the location and appearance of transient objects in the scene, such as people and vehicles, for reproducing or omitting them.

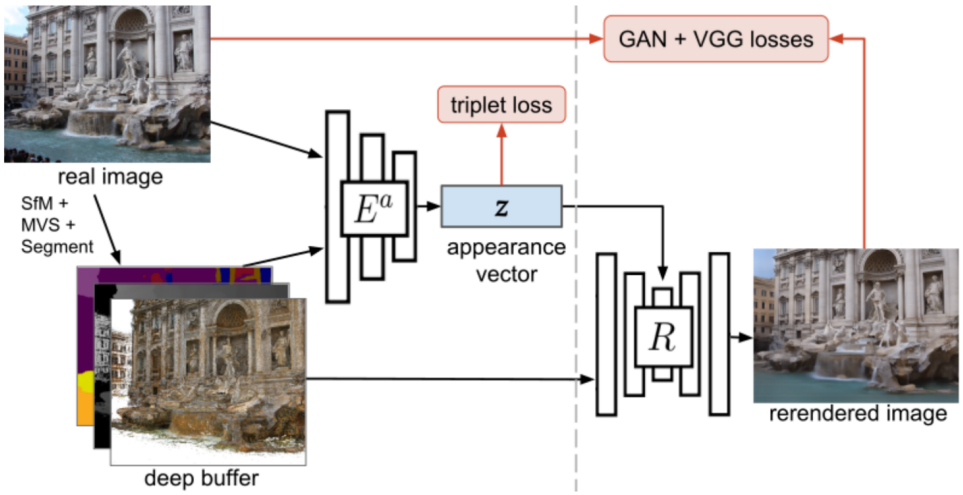


Figure 2.3: Both neural networks are trained in a staged approach that pre-trains the appearance encoder $E^a$ using a triplet loss, subsequently the rerenderer $R$ is trained with standard reconstruction and GAN losses (right), and finally, fine-tuned together with $E^a$. Taken from Meshry et al. [2019].

Following [Dechamps, 2020] in need of realistic point cloud renders, we utilize the model for both indoor and outdoor rendering.

"In the Wild" is related to unstructured photo collections from the internet NRIW can work with. The method starts with building a proxy explicit 3D colored point cloud representation from a collection of scene photos $\{I_i\}$ by utilizing Structure-from-Motion (SfM) and MultiView Stereo (MVS) implemented by COLMAP [Schönberger and Frahm, 2016; Schönberger et al., 2016]. Authors prefer point clouds over generating a mesh in a possible next step, even though meshes generate more complete renderings, as meshes "also tend to contain pieces of misregistered floating geometry which can occlude large regions of the scene".

In the next stage, an aligned dataset of deferred-shading deep buffers $B_i$ is generated. Such a buffer, in general, may contain per-pixel albedo, normal, depth, and any other derivative information. Authors use a combination of rendered and real images $\{I_i\}$, together with albedo and depth representations, all depicting the same view. By the rendered image, a point splatting with a z-buffer with a radius of 1 pixel render of the scene point cloud from a position $v_i$ recovered for the respective real image $I_i$ by SfM is meant. Even though this may resemble an image-to-image translation paradigm, it is not the case as such a model is uni-modal, not including appearance modeling. Image-to-image translation also fails to understand transient objects in the scene.

The aligned dataset is used to train a multimodal image translation model. Its goal is to learn a latent appearance vector $z_i^a$ that captures variations in the output domain $I_i$ that cannot be inferred from the input domain $B_i$. The method computes $z_i^a$ as $E^a(I_i, B_i)$, where $E^a$ is an appearance encoder of input $I_i$ and $B_i$ (the buffer is used for allowing the network to learn more complex appearance models by correlating the lighting in the real image with scene geometry in the corresponding buffer). Lastly, a rerendering network $R$ produces a scene rendering conditioned on both deep buffer $B_i$ and the latent appearance vector $z_i^a$. Fig. 2.3 presents a visual overview of the process.

The training process works as follows—to stabilize the joint training of $R$ and $E^a$, and improve the model expressiveness, pre-training the appearance encoder $E^a$ on a proxy task is first performed. In a staging manner, rendering network $R$ is then trained using fixed $E^a$ weights, allowing $R$ to find the correlations between output images and the embedding produced by the proxy task $E^a$ training. Finally, both networks are jointly fine-tuned.

The appearance pre-training works on a proxy task that optimizes embeddings of the input images in the appearance latent space based on a suitable distance metric. Similar images under the metric should also have similar embeddings. The metric itself should ignore viewport as appearance is independent of it. For that, authors use neural style-transfer triplet loss—for each image $I_i$, sets of $k$ closest and farthest neighboring images with respect to the metric below are found. From those, one positive $I_p$ and one negative $I_n$ image is sampled, respectively. The loss then is:

$$\mathcal{L}(I_i, I_p, I_n) = \sum_j \max\left(\|g_i^j - g_p^j\|^2 - \|g_i^j - g_n^j\|^2 + \alpha, 0\right) ,$$

where $g_i^j$ is the Gram matrix of activations at the $j$-th layer of a VGG network of image $I_i$, and $\alpha$ is a separation margin.

Lastly, semantic conditioning performed by concatenating a semantic labeling $S_i$ of image $I_i$ to the deep buffer $B_i$ is used to account for transient objects. The authors argue that it discourages the appearance encoder network from encoding variations caused by the location of transient objects in the appearance latent space or associating such objects with specific viewports.

## 2.3 Surface Splatting

Surface splatting, presented by Zwicker et al. [2001], is an efficient technique for rendering high-quality images of point clouds (point-sampled surfaces), supported by rigorous mathematical analysis around resampling. In contrast to ray-tracing, it is a forward-projection approach that uses z-buffer to resolve visibility. It can avoid aliasing artifacts brought alongside discretizing otherwise continuous space by a screen space formulation of the Elliptical Weighted Average (EWA) filter for irregularly spaced point samples without global texture parameterization.

It can be seen as a resampling process in signal processing [Gross and Pfister, 2007], effectively the method strives to reconstruct initially hole-free surfaces sampled in the form of a point cloud. To do so, the method uses a combination of an object-space reconstruction filter and a screen-space filter for each point primitive. The mathematical object-space reconstruction filter (*footprint function* $\rho_i(\mathbf{x})$ of a point $\mathbf{x}$) resembles typically an elliptical disk, a so-called splat whose position, orientation, and axes are usually chosen to provide a good approximation to the underlying source geometry. After a perspective projection of all splats to the screen space, the EWA filter mentioned above is used to avoid frequencies higher than the Nyquist frequency of the pixel sampling grid, and all contributions from the overlapping splats are combined.

The basic idea of splatting compared to a naive approach is shown in Fig. 2.4. The naive method does not work generally as it leads to holes in reconstructed

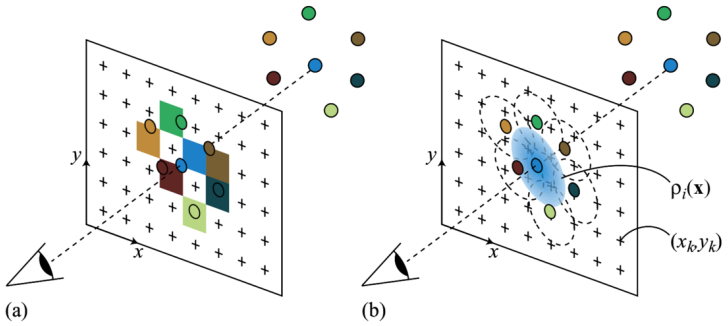


Figure 2.4: Point rendering by surface splatting compared to a naive approach that is used, for instance, by `GL_POINTS` OpenGL primitive. (a) Naive forward projection and rendering of point samples assigning the projected point's color to the closest pixel in the screen space. (b) By splatting footprint functions, each pixel gets color decided upon a combination of contributions scree-space neighboring points. Taken from Gross and Pfister [2007].

surfaces in the rendered image if the surface is not sampled with sufficient frequency. Also, another disadvantage happens when more than one point gets projected to the same closest pixel—then the rendering result depends on the order in which the points are processed. Surface splatting alleviates the problems by distributing the color of each projected point among more neighboring screen-space pixels with a suitable footprint function. The desirable footprint function is usually smooth, decays quickly with increasing distance from the projected center, and has local support as indicated by the ellipses in Fig. 2.4.

For a single channel of a possible multiple-channel (taken independently) image, image function $\phi(x, y)$ taking a pixel position and returning color could be defined according to previous thoughts as

$$\phi(x, y) = \sum_i c_i \rho_i(x, y)\,, \tag{2.1}$$

where the sum is carried over the indices of all points $\{\mathbf{p}_i\}$ of the surface, $\rho_i$ are individual footprint functions, and $c_i$ are channel color values of a given point.

The definition Eq. (2.1) has an issue with reproducing surfaces with constant color and thus can lead to visible artifacts. Also, footprint functions are truncated to finite support. Both leads to the below presented normalized image function used by surface splatting

$$\phi(x, y) = \sum_i c_i \frac{\rho_i(x, y)}{\sum_k \rho_k(x, y)}\,. \tag{2.2}$$

The image function defined by Eq. (2.2) leads to a two-pass algorithm for rendering Algorithm 1. In the first pass, all points are iterated over and their splat footprints $\rho_i$ and channel values $c_i$ are computed. The footprint functions are evaluated at each pixel, or rasterized, and their contributions are accumulated in a buffer. At each pixel $(x, y)$, the buffer stores the sum of the weighted contributions from the right side of Eq. (2.1), normalization factor sum from the denominator of Eq. (2.2) and the depth for z-buffering. In the second pass, all

**Algorithm 1** Pseudocode of the splatting algorithm.

```
1: procedure SPLAT_RENDERING(p[], c[], w[], z[])
2:     for all points i in p[] do
3:         rho_i ← footprint(p_i)
4:         c_i ← shade(p_i)
5:         rasterize(rho_i, c_i, c[], w[], z[])
6:     end for
7:     for all points [x, y] do
8:         c[x, y] /= w[x, y]
9:     end for
10: end procedure
```

pixels are processed by normalization of the accumulated contributions by the accumulated normalization factor.

How usable footprint functions are found and look is beyond the scope of this work as it requires signal-processing theory, Gaussian functions, and the Nyquist theorem. We utilize a GPU implementation by Sebastian Lipponer.[2]

## 2.4 Ray Marching with Signed Distance Fields

This method is an example of a ray-casting approach, in which a finite series of steps along a ray cast from a camera through a pixel is undertaken, until the ray hits an object or the maximum number of permitted steps is exceeded. This very simple idea is fundamental in computer graphics and dates back to works like Tuy and Tuy [1984]; Perlin and Hoffert [1989]. Building on the idea, many effects, such as lights, shadows, and transparency, can be incorporated, to name a few [Akenine-Möller et al., 2018]. This thesis implements a variant of the method using *signed distance functions* (SDF).

In a given scene consisting of solid bodies, a *signed distance function* is a scalar function $S(P)$ defined at every point $P$ in a (2D or 3D) space, such that

$$\begin{aligned} S(P) &= 0 \quad \text{when it is on the surface of a body,} \\ S(P) &> 0 \quad \text{when it is inside any body,} \\ S(P) &< 0 \quad \text{when it is outside all bodies [Evans, 2006].} \end{aligned} \tag{2.3}$$

A *scene SDF* defines the scene implicitly. An *object SDF* is the SDF of a scene containing only that one object. Object SDFs can be computed analytically for simple shapes; see work of Inigo Quilez,[3] or tabulated in grids, octrees, or other spatial data structures. A scene SDF can then be constructed by combining SDFs of objects in the scene. For simple analytically-describable shapes, an insight into how such a scene can be built may be given through Constructive Solid Geometry (CSG), a method of creating complex geometric shapes from simple ones via boolean operations, see the left of Fig. 2.5. This process has its mirror in combining SDFs; see code example in Listing 2.1. As an example, an SDF for the simplest 3D object, a sphere positioned at the origin and with a defined radius, is

[2] `https://github.com/sebastianlipponer/surface_splatting`
[3] `https://iquilezles.org/articles/distfunctions`

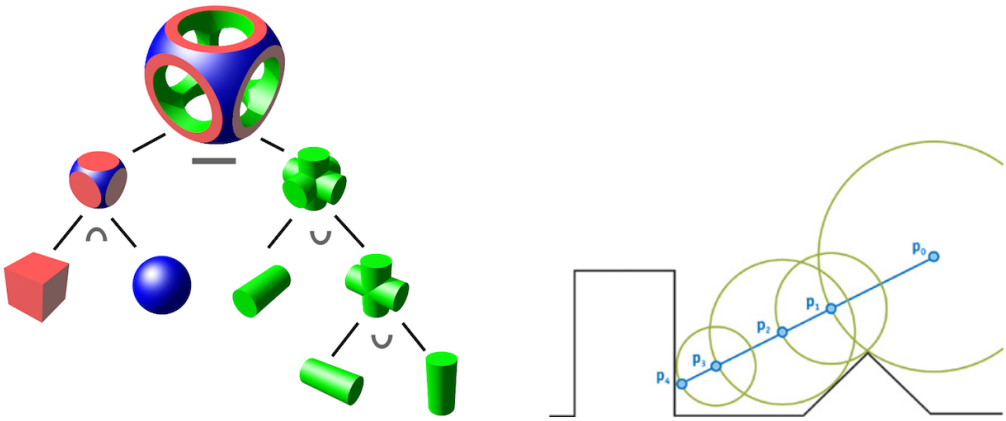


Figure 2.5: Left: CSG is built upon three primitive operations: intersection (∩), union (∪), and difference (−). Taken from `https://en.wikipedia.org/wiki/Constructive_solid_geometry`. Right: Demonstration of ray marching where at each step algorithm proceeds along given ray by a distance to the closest surface as it is a safe way how to find a hit, based on some threshold distance. Taken from `https://jamie-wong.com/2016/07/15/ray-marching-signed-distance-functions/`.

`SDF_sphere(vec3 pos) -> length(pos) - RADIUS`. In general, SDF does not need to be based on Euclidean distance and may be exact or approximate. The only theoretical requirement is Eq. (2.3) and from a practicality perspective, evaluation of such a function should be reasonably quick. The algorithm utilizing a scene SDF is outlined in Algorithm 2. The visual representation of the simplest form of the algorithm proceeding along a ray is to be found in the right of Fig. 2.5.

Listing 2.1: Code example showing how SDFs of simpler object can be combined together to gradually build a scene SDF.

```
def SDFintersect(obj1_SDF, obj2_SDF):
    return max(obj1_SDF, obj2_SDF)

def SDFunion(obj1_SDF, obj2_SDF):
    return min(obj1_SDF, obj2_SDF)

def SDFdifference(obj1_SDF, obj2_SDF):
    return max(obj1_SDF, -obj2_SDF)
```

In this thesis, we utilize scene representation relying on translated sphere SDFs with radii precomputed beforehand to be dependent on the distance to the closest neighbor for a given point primitive. Scene SDF for such a scene would be then a minimum of all point SDFs in the scene (following again Listing 2.1). This naive declaration is not scalable to millions of points a scene produced by a LiDAR or SfM may contain, even though only the simplest point primitives are used for the rendering process. To increase significantly rendering performance with sufficient reality reproduction capabilities, we take advantage of a spatial 3D KD-tree [Bentley, 1975] implemented in NVIDIA CUDA[4] toolkit that can quickly

[4] `https://developer.nvidia.com/cuda-toolkit`

**Algorithm 2** Pseudocode of the ray marching with SDF.

```
1: procedure RAY_MARCH(ray_origin, ray_direction)
2:     dist ← 0
3:     for i in range(MAX_STEPS) do        ▷ Hyperparameter to stop traversal
4:         current_pos ← ray_origin + dist ∗ ray_direction
5:         closest ← SDFscene(current_pos)
6:         if closest.dist < MIN_HIT_DIST then              ▷ Float comparison
7:             return closest.color
8:         end if
9:         if dist > MAX_DIST then                          ▷ No hit along the ray
10:            return BACKGROUND_COLOR
11:        end if
12:        dist ← dist + closest.dist
13:    end for
14: end procedure
```

return the closest point for a given location. Since the exact sphere SDF contains its radius and KD-tree built on top of the source point cloud returns the distance to the sphere center (point) itself, not the distance to the sphere's surface, we take $N$ closest points, instead of just one, compute exact SDF for those with their respective radii, and then take the point at the minimal distance determined. The same KD-tree is pre-build once at the start of the rendering process and is also used for radii computation instead of computing those exhaustively.

# 3. Camera Pose Verification

In the chapter, datasets, their transformations, and experiments performed upon them with the introduced methods' implementations are all presented. We use the generalized InLoc pipeline with a modified pose verification step. The synthesized image leveraged for pixel-wise computation of similarity with the given query image is swapped with views generated by renderers presented in the preceding chapter, depicting the scene's point cloud from estimated query positions. Apart from how the synthesized image is generated, the rest of the verification process is then performed as described in the original InLoc article, using namely the RootSIFT descriptors.

While discussing concrete details of datasets' definitions and algorithms' inputs, more technical aspects are taken into account—among them, of utmost importance are conventions used by coordinate systems in which points of explicit scene representations are expressed/expected to be and by matrices related to cameras taking database images. These pose a crucial difference between what a dataset provides, or localization pipeline expects and must be addressed by implementation to obtain valid localization results.

Coordinate system conventions address the decision of assigning positive directions, labels, and meanings in the human sense (up, right, forward) to the orthogonal frame of a 3D space, because without that an oriented triplet representing a point is meaningless. These conventions can be arbitrary depending on whether they come from computer vision, rendering, or another field. Examples of such conventions, linked to standard computer graphics libraries/tools that use/expect them, can be found in Fig. 3.1. In this thesis, computer vision and rendering conventions are used. In the figure Fig. 3.1, these are found alongside OpenCV and OpenGL labels, respectively. Even though both are right-handed, they understand x, y, and z point components differently. In rendering, the positive x-axis points to the right, the positive y-axis up, and the positive z-axis towards a viewer looking at the coordinate system frame. In computer vision, the positive x-axis points to the right, the positive y-axis to the bottom, and the z-axis away from the same viewer as before. Transformation matrices operating over both notations are thus related by inverting the y and z axes columns. As an example, since a camera in rendering is typically placed along z-xis, failing to take this relation into account when displaying a 3D model defined in computer vision notation in a visualization tool that uses rendering notation results in rendering half-space "away" from the model. For instance, in the case of other notations used by a produced model, a rendered view can be unexpectedly rotated.

Matrix conventions in the context of the thesis are related to terms coming from the graphics pipeline—*world space* and *view space*. In the case of datasets described below, world space is a space of the whole scene representation with the origin and orientation of the coordinate frame chosen arbitrarily in relation to the scene. The randomness in the coordinate frame placement is especially true in the case of SfM-generated scene models, where the algorithm decides these parameters. When preparing a model manually, e.g., in the game industry, the frame is typically artificially placed meaningfully concerning the model produced,

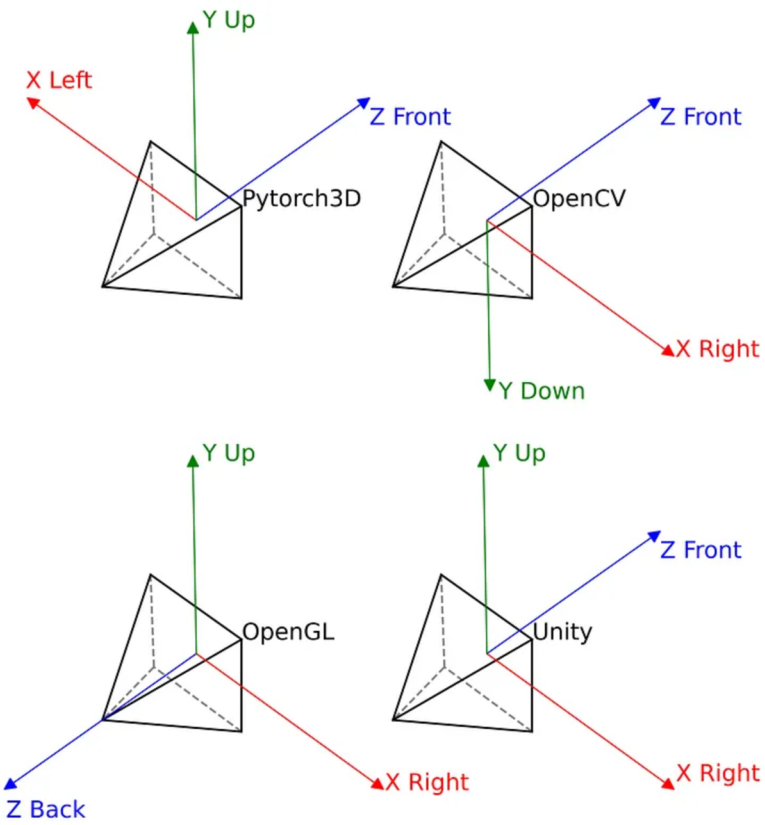


Figure 3.1: Examples of various camera / coordinate frame conventions used by common programmatic tools in fields dealing with computer graphics. Taken from `https://medium.com/check-visit-computer-vision/converting-camera-poses-from-opencv-to-opengl-can-be-easy-27ff6c413bdb`.

e.g., along the outer edges of a cube model. View space is a space of a camera looking at a portion of the scene—origin is the center of the camera with the coordinate frame oriented in a specific way alongside the optical axis of the camera depending on the exact graphics pipeline/tool used. Visualisation Fig. 3.1 can be used here as well—the square pyramids depict view frustums of virtual cameras, with the z-axes being their optical axes.

Provided both spaces are same-handed, the matrix inverse relates transformations between them. In homogeneous coordinates, both transformations are represented by $4 \times 4$ matrices, and the implementation must correctly distinguish between the actual meaning of these 16 real numbers, including how the matrix is stored on the disk. We refer to them as the *view matrix* transforming from world to view space and *camera pose* representing the opposite, inverse transformation.

## 3.1 Datasets

Several image collections were used for measuring localization performance, both indoors and outdoors. To ensure continuity and comparability with previous works of Taira et al. [2018]; Dechamps [2020], we utilize the open-source InLoc Dataset presented in the original Inloc paper [Taira et al., 2018]. Another closed-source, indoor dataset is a 3D scanner-generated digital twin of a SIEMENS manufacturing facility that is targeted by several use cases of the Industry & Construction 4.0 Solutions project called ARTwin, financially supported by the European Union's Horizon 2020 research and innovation program. Finally, an outdoor dataset is covered by the inclusion of the open-source Phototourism dataset from the Image Matching Challenge 2021.[1]

### 3.1.1 InLoc Dataset

The dataset consists of a database of Faro 3D scanner-generated RGBD scans that are geometrically registered to the floor plan of two buildings of Washington University in St. Louis. The test set is a composition of RGB photos taken by a hand-held device (an iPhone).

277 RGBD panoramic images have ground truth poses in the global coordinate system spanning across the floor plan. Each RGBD panoramic scan is a point cloud (*scan*) having roughly 40 million colored points. The final dataset is generated by obtaining 36 perspective RGBD images from each panorama by extracting standard perspective views (60° FoV) with a sampling stride of 30° in yaw and ±30° in pitch directions, resulting in cca 10 thousand perspective images in total, examples are in Fig. 3.2. This dataset contains all troublesome elements for indoor localization, namely repetitive patterns (such as stairs and pillars), global and local similarities (doors, windows), furniture changing positions in the test set, people moving across the scene, and textureless, highly symmetric areas (walls, floors, corridors, classrooms, open spaces).

The original query set consists of 356 photos taken by an iPhone 7 at various lighting conditions within a day, capturing a variety of occluders and layouts (people, furniture), also covering only a subset of the floor plan data, with the rest playing the role of confuser at search time. Ground truth poses for the test set are not publicly accessible, and evaluation can be done only indirectly via submission to the Visual Localization[2] page.

The structure of the dataset's database folder is as follows—`scans/<FLOOR>` folders, where `FLOOR` is one of DUC1, DUC2, CSE3, CSE4, and CSE5, representing five floors of the two mentioned buildings (CSE, DUC), contain files named `<NAME_WITH_SCAN_NUMBER>.ptx.mat` storing RGB and XYZ information of scanned points in Matlab file format. Every floor has its specific number of scans, uniquely numbered within a building. Final dataset's perspective views are stored in folders `cutouts/<FLOOR>/<SCAN_NUMBER>` containing JPG perspective RGB images of size 1600 × 1200 pixels and MAT files containing bundled RGB perspective image (RGBcut) and the respective scan points (XYZcut). Files `alignments/<FLOOR>/transformations/<NAME_WITH_SCAN_NUMBER>.txt`

[1] `https://www.cs.ubc.ca/research/image-matching-challenge/2021`

[2] `https://www.visuallocalization.net`

Figure 3.2: Samples from InLoc dataset database images.

contain $4 \times 4$ transformation matrices that convert 3D homogeneous points in original .ptx.mat files to the global coordinate system of the floor plan.

The dataset's query folder contains one subfolder named `iphone7` with the query set of photos taken by the iPhone camera. Photos are stored as JPG files of size $4032 \times 3024$ or $3024 \times 4032$ pixels, so both landscape and vertical acquisition modes were used. As the database is landscape, for InLoc algorithm processing, all images are made landscape, and the ones where the view was changed by rotation are remembered. Notably, even though sharing the same aspect ratio with the database after such operation, which InLoc localization pipeline can handle, resizing to the matching dimensions is also used to speed up the localization performance.

### 3.1.2 ARTwin Dataset

The dataset consists of registered 360° RGB panoramic images across two halls of a SIEMENS manufacturing facility together with point clouds for both produced by merging 3D data from a NavVis 3D scanner.

Over the two halls, 29 and 53 panoramic images were obtained. The final dataset used in this thesis contains roughly 4 thousand processed images and it is generated in accordance with InLoc Dataset except for a difference in the necessity to remap 360° spherical panorama to 2D surface again, examples are in Fig. 3.3. Hall point clouds are not matched to a common coordinate system as they overlap when displayed together, so for localization disambiguation one hall is lifted along the z-axis.

The raw dataset contains all the intermediate files, photos and logs from the acquiring process together with processed and merged results mentioned above. The structure of the relevant processed data is `proc/<HALL_ID>`, IDs of the halls are `2019-09-28_08.31.29` and `2019-09-28_16.11.53`. Within each of these folders, there is processed point cloud `<HALL_ID>.ply` and `pano` folder with JPG panoramic scans alongside `pano-poses.csv`. Poses are in the form of 3D scanner position and orientation quaternion per panoramic scan.

### 3.1.3 Phototourism Dataset

The smallest datasets taken from the Image Matching Challenge (IMC) data, photo-tourism image collections depicts several popular landmarks, collected from

Figure 3.3: Sample flattened images from ARTwin dataset, on the left hall `2019-09-28_16.11.53` is presented, on the right `2019-09-28_08.31.29`.

the Yahoo Flickr Creative Commons 100M (YFCC) dataset. Namely, Hagia Sophia Interior, Pantheon Exterior, and Grand Place Brussels collections were used. These datasets have around 1 000 photos each coming, using the terminology from Meshry et al. [2019], "from the wild" as they were taken by many authors, at various distances and with sensor sizes varying considerably, see Fig. 3.4.

The dataset per given collection used in the thesis is built on top of raw images by running COLMAP software. Structure of the COLMAP produced dataset is described in its documentation[3]. Shortly, there is `dense/sparse/cameras.bin` file with parameters of cameras capturing wild images retrieved by SfM method implemented in COLMAP, `dense/sparse/images.bin` file with retrieved 3D positions and orientation quaternions of each image in a common coordinate system of `dense/fused.ply` point cloud. This point cloud is generated by SfM from implicit scene representation contrary to the previously mentioned datasets that represent a 3D scanner-generated approach.

Putting everything together—all scene point clouds for all datasets explored in the thesis are placed in the right-handed coordinate system, though conventions of the coordinate frames vary, described side-by-side in Table 3.1. In order to properly render a virtual view, related camera poses must be preprocessed accordingly. Another side-by-side comparison of the datasets can be found in Table 3.2 presenting their basic statistical features.

[3] `https://colmap.github.io/format.html`

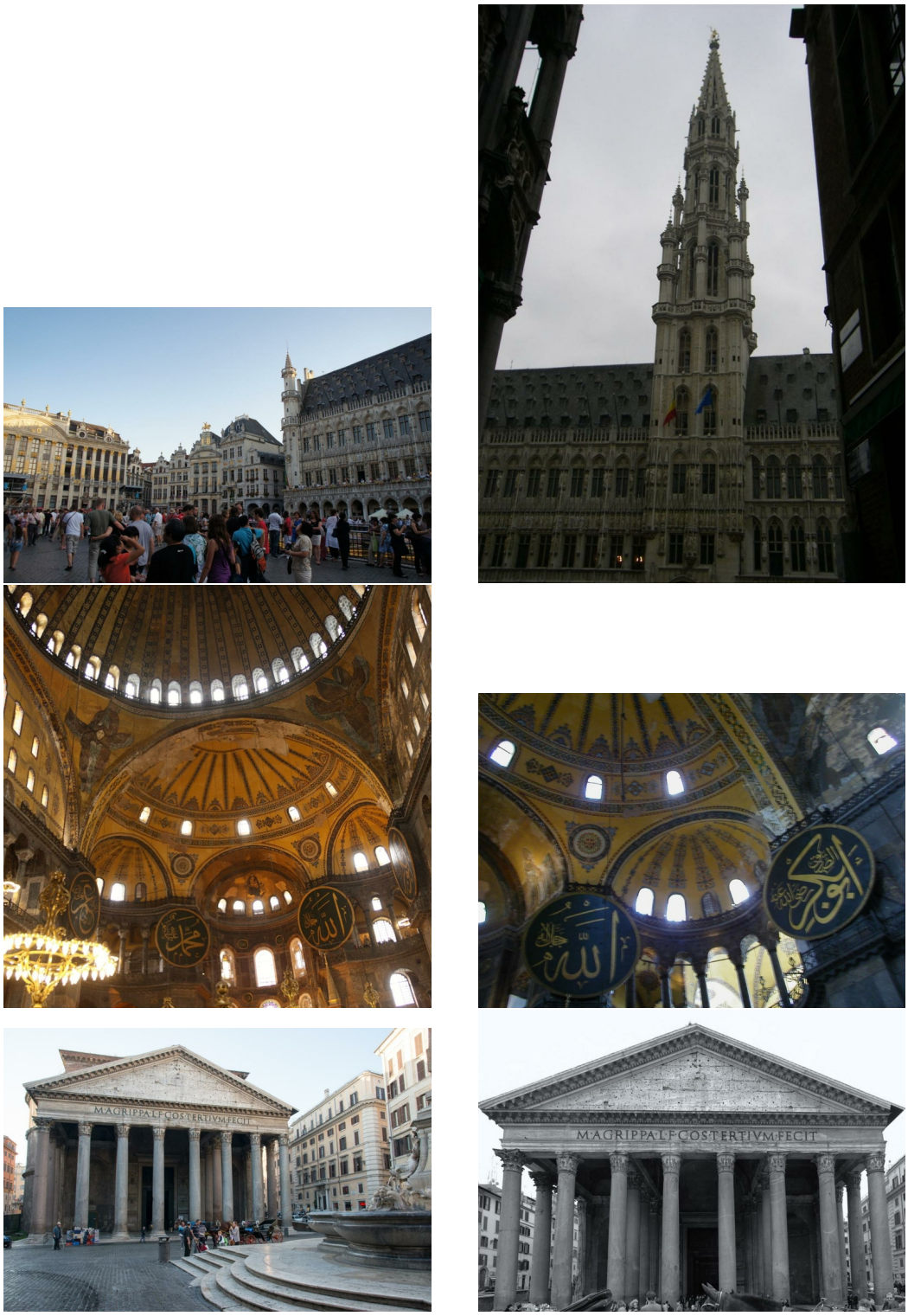


Figure 3.4: Sample images from Phototourism dataset taken "from the wild", stretching various aspect ratios, sizes, time of the day of the aquisition, and varying lighting conditions present in the data. In the top row there are images of Grand Place in Brussels, below of interior of Hagia Sophia Grand Mosque in Istanbul, and in the bottom of Pantheon in Rome.

Table 3.1: Comparison of conventions and notations found in scene representations of all datasets explored in the thesis.

| ARTwin | InLoc Dataset | IMC |
|---|---|---|
| Right-handed coordinate system, scans use convention where in order to be rendered by an OpenGL camera to match database images, in sequence, x-y and y-z axes must be switched. There is no notion of global CS where both hall point clouds can be placed, so an artificial translation along z-axis is performed on the hall labeled 53 for localization disambiguation. | Right-handed coordinate system, scans use convention where in order to be rendered by an OpenGL camera to match database images, in sequence, x-y and y-z axes must be switched. For each scan, a transformation from local to the defined global CS is known. | Right-handed coordinate system, model in computer vision (CV) notation. To render properly by an OpenGL camera to match database images, y and z axes must be inverted. COLMAP-generated per-view matrices are view matrices. |

Table 3.2: Comparison of various features of all datasets used in the thesis. InLoc test set specified here is generated from the dataset so that we have the ground truth poses, otherwise the online evaluation tool would need to be used. Number of points in a scan refers to the mean of points count for iInLoc and ARTwin datasets, and to the number of points in the whole scene model for the rest. Dimensions are specified in thousands of pixels and for Phototourism datasets it is not applicable as source photos have various dimensions.

| | InLoc | ARTwin | Hagia Sophia | Pantheon | Grand Place |
|---|---|---|---|---|---|
| Train Size | 7 977 | 2 423 | 670 | 1 078 | 821 |
| Val Size | 1 995 | 379 | 167 | 269 | 205 |
| Test Size | 356 | 150 | 50 | 50 | 50 |
| Scan Points | 40M | 27M | 5M | 5M | 4M |
| Dims [k pix] | 1.6x1.2 | 1.6x1.2 | - | - | - |

## 3.2 Implementation

For purposes of the thesis, several code projects are leveraged, either built from the ground up by the author or based on top of the previous work of others. Localization InLoc framework is based on the work of the article author Hajime Taira and further enhancements done by Pavel Lucivnak and Bastien Dechamps spread across several code repositories. From renderers, for Neural Rerendering in the Wild, the authors' implementation with surrounding scripts written by Bastien Dechamps is used as the base of further work. For surface splatting, the great work of Sebastian Lipponer, with some tweaks, is leveraged. Finally, the ray marching renderer based on OpenGL is entirely the author's work.

Aside from the localization algorithm and renderers, scripts transforming dataset formats, described in Table 3.1, into notations and conventions used by the InLoc pipeline and renderers themselves as described in Table 3.3 are also added; for more information, see below.

To be runnable in CIIRC computational cluster environment, which distributes jobs submitted by users by Slurm[4], batch job shell scripts are written. Slurm is an open-source, fault-tolerant, and highly scalable cluster management and job scheduling system for clusters of Linux-running machines. Slurm requires the batch job scripts to specify memory, CPU, and GPU requirements for encompassed computation. Specifying these in the code results in less time for experiment reproduction; it can also be immediately seen whether one has enough resources to run it in the first place. These bounds vary greatly for algorithms and models utilized in this thesis, from a few GB of RAM to almost 400 GB for InLoc processing the InLoc dataset, zero to eight GPUs for NRIW training, and typically a few CPU cores.

Alongside these shell scripts, an attempt to have a reproducible experimental pipeline was made using *Docker*/*Singularity* and *DVC*.

Docker[5] is an industry-grade platform allowing to build, test, and deploy applications quickly and robustly. Docker is an example of container-based virtualization, where a container is a running "image" that packs everything the software needs for running, including libraries, system tools, code, and runtime. This approach is suitable for computational cluster environments as the code can be executed without relying on cluster administrators to install necessary packages globally for all users, which often leads to software version collisions. Containerization is a more lightweight virtualization technique compared to classical virtual machines resulting in quick startup times, lower memory requirements overhead, and a more user-friendly working experience suitable for both development and productionalisation. Technically, this is enabled by sharing underlying OS kernel by all running containers, contrary to virtual machines that are ran on bare metal with the so-called hypervisor, emulating their own OS kernels separated from other virtual machines running on the same hardware. To be more specific, the thesis relies on GPU-based computations; to be able to run GPU workloads in a container, there is an exception to the mentioned advantage of container-based

[4] `https://www.schedmd.com`

[5] `https://www.docker.com`

virtualization to avoiding cluster administrators globally changing the cluster—suitable GPU drivers must be installed. Especially for all the renderers described to be able to use off-screen headless rendering (rendering to textures and saving them without displaying them on a monitor) on NVIDIA GPU cards used by the CIIRC computational cluster, a driver with bug-less EGL[6] support must be used, which is something not every driver version satisfies. CUDA and OpenGL libraries are then owned by each container, communicating with the shared driver on the operating system level.

Singularity[7] is a containerization platform similar to Docker, with one notable exception leading to the adoption of the tool by computational cluster administrators (including CIIRC's) instead of the otherwise industry-leading and widely used Docker—it does not require administrative privileges from its users. For this thesis, descriptions of Docker images to be built are written where applicable. Once built, images are transformed into Singularity variants runnable on the cluster. This functionality is supported natively by `singularity` binary as it is a common use-case for the tool.

Data Version Control (DVC[8]) should help traceable and reproducible science by leveraging the Git version system to also version data, intermediate results, tie them with the exact code that produced them and thus track all ideas and experiments. It also can manage workflows which is valuable for defining reproducible data pipelines. The advantages of this tool are simplicity as it uses Git—which is a standard tool in code development—and workflow management is done through simple shell scripting that is well suited for the Slurm environment with running various Singularity containers as scheduled jobs. Though the idea is promising in the recent growth of Machine Learning Operations (MLOps), the tool proved unsuitable when used for datasets consisting of an enormous number of smaller files which is often the case in computer vision. DVC uses hashes to check consistency and the necessity to recompute some steps in a workflow, so it may take many hours to run even elementary transformations. The overhead of these hash computations is considerable, leading to the decision not to use DVC after all.

### 3.2.1 InLoc localization pipeline

The implementation of the pipeline[9] is based on the Matlab sources written by the article's author.[10] The source code is unified for better readability and verifiability; it is also generalized, as the original implementation targets specifically the InLoc dataset;[11]. Furthermore, as the original code lacks computation of scores and evaluation for a general dataset, the proposed approach of Pavel Lucivnak for the former is leveraged and further developed;[12] [13]

The outline of the implementation is depicted by Fig. 3.5. Cutouts for a

[6] EGL is an interface between Khronos rendering APIs such as OpenGL ES or OpenVG and the underlying native platform window system, `https://www.khronos.org/egl`.

[7] `https://sylabs.io`

[8] `https://dvc.org`

[9] `https://github.com/Auratons/inlocciirc_demo`

[10] `https://github.com/HajimeTaira/InLoc_demo`

[11] `https://github.com/HajimeTaira/InLoc_dataset`

[12] `https://github.com/lucivpav/InLocCIIRC_demo`

[13] `https://github.com/lucivpav/InLocCIIRC_dataset`

dataset is a folder structure containing 3 files per database (DB) image—pose file, the image itself, and a so-called "XYZcut". The cut is a $M \times N \times 3$ array with XYZ coordinates of a surface that would be hit first by a ray cast from the center of the camera that took the DB image of size $M \times N$ (ignoring color dimension) through given pixel. Computation of the cuts is not part of the source InLoc pipeline implementation, so one method of getting the cut from a particular renderer-generated depth map and known camera parameters is implemented.[14] The InLoc dataset contains default XYZCuts. However, as mentioned, there is no generation script enclosed. The method implemented in the thesis uses depth maps to reproject from 2D to 3D space. Since these depend on the renderer, all XYZCut are recomputed per rendering approach. The default cuts were checked for the depiction of the background—when the ray does not hit anything in the given view frustum, the respective coordinate is a triplet of NaNs. Since OpenGL-based renderers typically output zero as the depth value of these "not hit" cases, reprojected points close to the origin of the cut are filtered.

A database and query image similarity score used later for image retrieval is computed as cosine similarity of normalized feature vectors. The original implementation using the matrix multiplication of query and database features stacked onto each other has immense memory requirements depleting all resources when executed on the considerably extensive InLoc dataset, thus some allowed linear algebra adjustments are made, lowering the requirements to reasonable numbers.

For the image retrieval step, we use 100 closest database images to every given query photo based on the similarity score, which is the same number of candidates as the origin article. For all these candidate poses, after transforming them into formats expected by the renderers explored in the thesis, we produce candidate renders and use those in the standard pose verification process described in Section 1.2 based on the RootSIFT descriptors. After reranking the candidate positions based on the image-render similarity, 10 best sorted candidates are outputted as in the original article.

Evaluation is done by comparing angular and spatial L2 distances between the candidate and the query's true pose, if known. Specifically, for the InLoc dataset, the ground truth poses for the query set are not publicly disclosed. Only an online evaluation tool `https://www.visuallocalization.net/submission/` returning the fraction of correctly localized queries within the distance and angular threshold can be used.

### 3.2.2 Neural Rerendering in the Wild

For the Neural Rerendering in the Wild, the original implementation is also used[15] without any substantial changes, just minor technical enhancements, such as support for alpha channel processing, etc.[16]

All scripts needed on the path from a raw dataset to "Aligned Dataset" and "Cutouts" in Fig. 3.6, Fig. 3.8 and Fig. 3.7 are also implemented in the repository. Aligned dataset is expected as input to the NRIW training process after packing into a TFRecord, similar to Cutouts being expected by the InLoc pipeline.

[14] `https://github.com/Auratons/inlocciirc_dataset`

[15] `https://github.com/google/neural_rerendering_in_the_wild`

[16] `https://github.com/Auratons/neural_rendering`

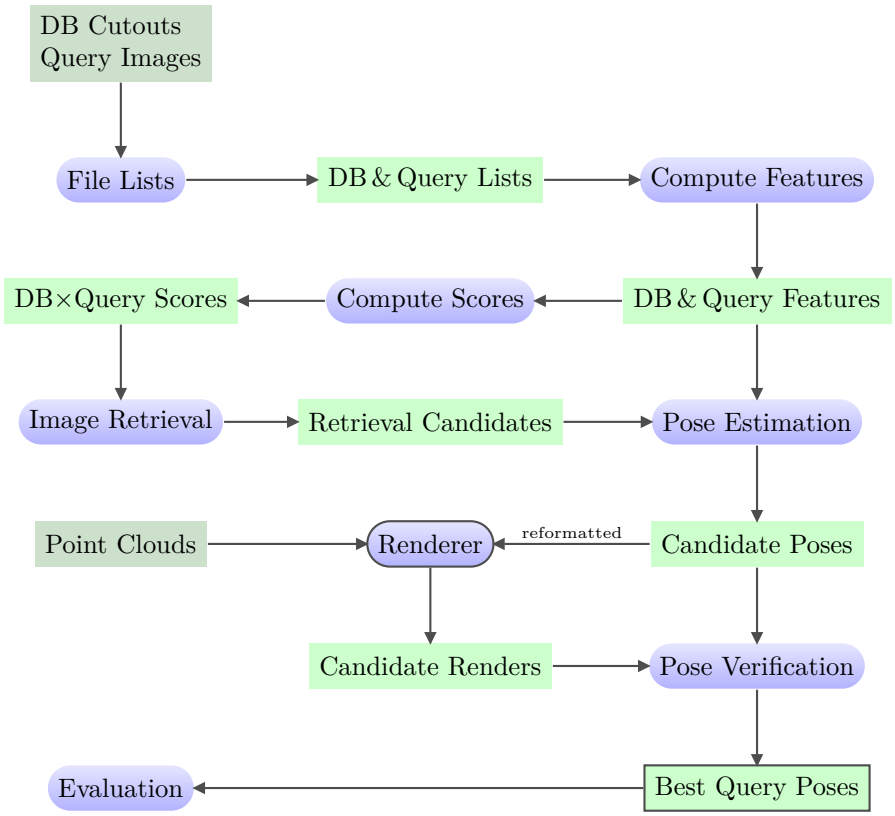


Figure 3.5: InLoc algorithm. The implementation outline uses terminology from the article. Rectangles represent file(s) on the disk, dark green ones denote algorithm inputs, and the rest are intermediate outputs except algorithm outputs with a border drawn. Blue nodes of the outline denote processing steps. "DB cutouts" are a database (DB) format the implementation expects. "File Lists" step scans the database and query, storing valid found examples and query images into a file for further reference. The "Renderer" step highlighted with border is the main concern of the thesis.

For ARTwin, spherical photos need to be unrolled to 2D with the exact sampling approach as for the InLoc dataset, resulting in the set of reference images. To be able to reuse scripting written initially for the IMC raw dataset, the creation of COLMAP-like camera and image information structure is implemented alongside unrolling in preprocess script. For depth information used within the aligned dataset, the point cloud is rendered via the load data script for ARTwin and IMC data and the render InLoc DB script for the remaining dataset. These scripts utilize the Pyrender python package internally, transversely `GL_POINTS` OpenGL primitive for rendering. The approach is a common baseline with previous works on the topic. To generate an aligned dataset with point cloud renders provided by other renderers, the "Generate matrices" step is used, for both splatting and marching, as they share technically the same headless rendering component mentioned below.

The aligned dataset is a structure containing, in the simplest case, a triplet of an image, a color render of the underlying scene representation by a renderer, and the respective depth map. The triplet forms a deep buffer mentioned in the article. In the original article, the authors also use semantic masking in their deep buffers. However, when rendering a novel view not seen in the training data, the semantic mask cannot be obtained from an actual photo. Authors thus train a separate segmentation network between the partial deep frame buffers and the semantic masks Si to tackle this issue. However, this makes the network more complex and lowers the prediction time performance. Semantically segmenting the point cloud might be used, but following Dechamps [2020], the additional complexity is avoided.

The model is then trained with the same staged approach, where the appearance encoder is first pretrained on a proxy task with the triplet loss. As in the original article, $256 \times 256$ central crops of the deep buffers are used. For all the datasets, the whole train/val sets are used as the model is scene-dependent, which is especially true for the InLoc dataset, where the test set covers only a portion of the database.

Finally, the training parameters are also used following the article, only with scaled batch sizes according to GPU memory available in the CIIRC cluster DGX nodes. That means training on 8 GPU for around four days for the complete staged pipeline, with the Adam optimizer set with with parameters $\beta_1$, $\beta_2$ set to 0, 0.99, respectively, and the learning rate equal to 0.001.

### 3.2.3 Spherical Ray Marcher

The spherical ray marcher is, on the high level, an OpenGL[17] application with both interactive and headless rendering capabilities. Interactive rendering includes FPS camera moved by keyboard, displaying real-time point cloud on user's monitor. Headless mode generates specific view renders on the fly, without a monitor, directly to files on the disk, and it serves for dataset generation.

The final texture depicting the requested view is computed from the main raycasting loop implemented in CUDA[18] due to usage of a KD-tree implementation

[17] `https://www.opengl.org`

[18] `https://developer.nvidia.com/cuda-toolkit`

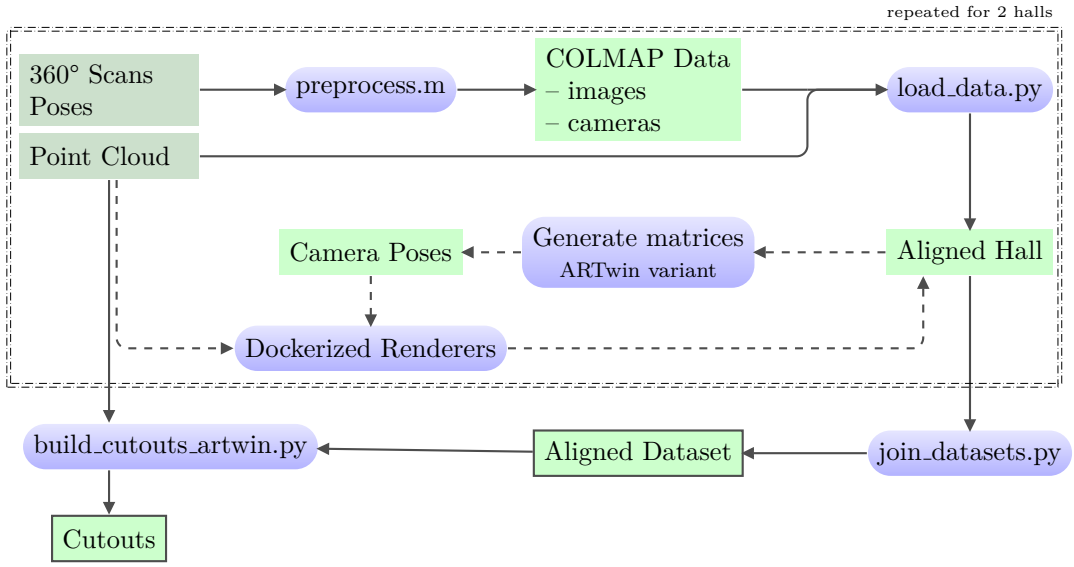


Figure 3.6: ARTwin dataset pathway. The schema displays transformations the ARTwin dataset undergoes in order to get either "Aligned Dataset" expected by the Neural Rerendering in the Wild DNN training or "Cutouts" for the localization pipeline. Rectangles represent file(s) on the disk, dark green ones denote algorithm inputs, and the rest are intermediate outputs except algorithm outputs with a border drawn. Blue nodes of the outline denote processing steps. The dashed paths are used to incorporate additional steps needed for non-default renderers. The default rendering with Pyrender is implemented in the load data script.

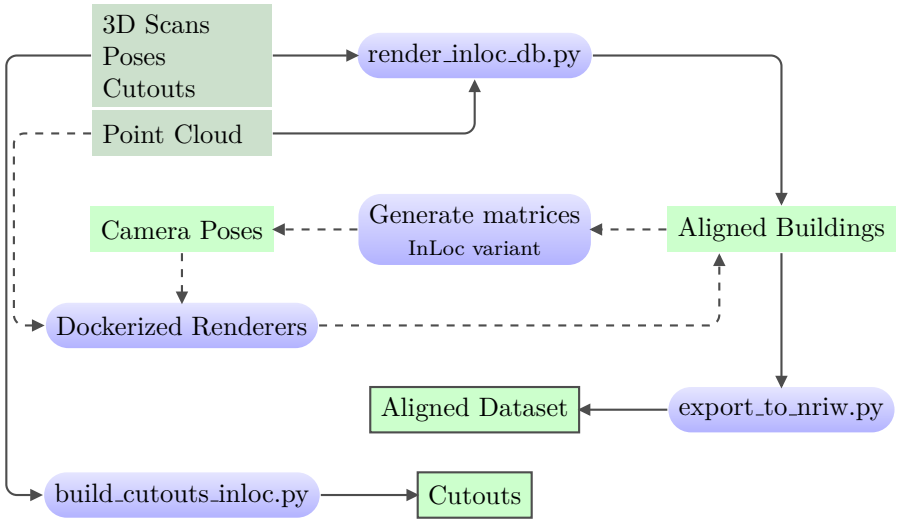


Figure 3.7: InLoc Dataset pathway. The schema displays transformations the InLoc dataset undergoes in order to get either "Aligned Dataset" expected by the Neural Rerendering in the Wild DNN training or "Cutouts" for the localization pipeline. Rectangles represent file(s) on the disk, dark green ones denote algorithm inputs, and the rest are intermediate outputs except algorithm outputs with a border drawn. Blue nodes of the outline denote processing steps. The dashed paths are used to incorporate additional steps needed for non-default renderers. The default rendering with Pyrender is implemented in the render InLoc db script.

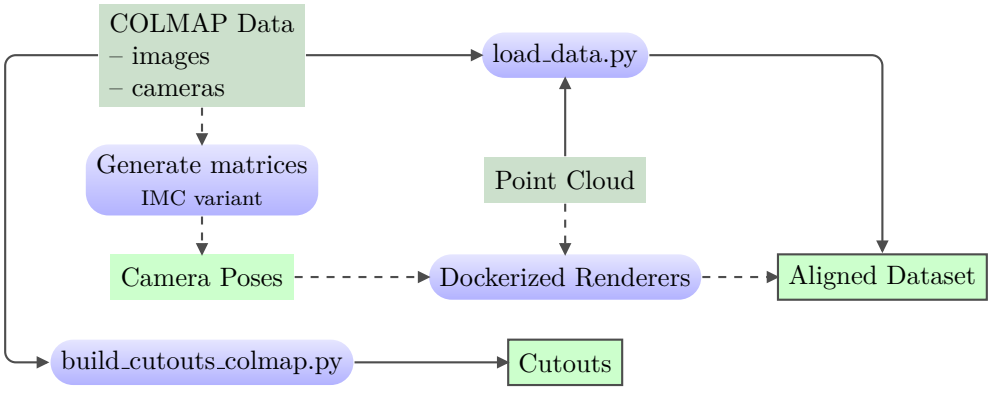


Figure 3.8: IMC Dataset pathway. The schema displays transformations the IMC dataset undergoes in order to get either "Aligned Dataset" expected by the Neural Rerendering in the Wild DNN training or "Cutouts" for the localization pipeline. Rectangles represent file(s) on the disk, dark green ones denote algorithm inputs, and the rest are intermediate outputs except algorithm outputs with a border drawn. Blue nodes of the outline denote processing steps. The dashed paths are used to incorporate additional steps needed for non-default renderers. The default rendering with Pyrender is implemented in the load data script.

based on the FLANN[19] project. As the NRIW training and cutout computations require depth maps, the implementation also provides outputting depth texture alongside any RGB render.

The algorithm needs radii for points to be rendered; the implementation[20] can compute those based on a distance to the nearest neighbor and cache them alongside the input point cloud. From this process, one hyperparameter stems out as the maximal displayable diameter. For outliers, radii may be too large, causing vast portions of a resulting render to be hidden behind giant spheres. The maximum can be determined as a percentile of cached radii for a given point cloud. Requested renders are expected to be specified by the respective camera poses and camera calibration matrices.

### 3.2.4 Surface Splatting

For surface splatting, Sebastian Lipponer's implementation was used[21] as a base, the project[22] was enhanced with the same headless rendering capability as in the case of the ray marching, expecting the same per-render format of camera poses and camera calibration matrices. Also, a mechanism for loading the radii of the point cloud being rendered was added, further reusing the component from the spherical ray marcher. Furthermore, for the same NRIW training process reason, the depth buffer content is made accessible as another output from the renderer. The original implementation produced only RGB outputs. Finally, a bug in camera handling of the underlying interactive rendering library GLviz of the same author was identified and resolved[23] to have rendered views for the same camera poses unified across all renderers used. The bug is not apparent when using the FPS camera to explore the displayed scene model. However, when comparing generated views to the outputs of a computer-vision grade renderer, it becomes obvious.

The algorithm needs not only per-point radii but also normal vectors in order to orient splats properly. For computing those, Meshlab[24] and, for automation, Pymeshlab[25] tools were used. The maximal diameter hyperparameter is used in the same sense as for the Marcher.

[19] `https://github.com/flann-lib/flann`
[20] `https://github.com/Auratons/renderer_ray_marching`
[21] `https://github.com/sebastianlipponer/surface_splatting`
[22] `https://github.com/Auratons/renderer_surface_splatting`
[23] `https://github.com/Auratons/glviz`
[24] `https://www.meshlab.net`
[25] `https://pymeshlab.readthedocs.io`

Table 3.3: Comparison of input format expectations of algorithms used in the thesis from the transformations perspective. The upper row presents the algorithms, and the lower one contains abbreviations of conventions, where *RC* means rendering convention, *CP* means camera pose matrix. InLoc pipeline is agnostic to matrix convention as far as it is consistent with data generation. NRIW itself is trained only with images, noteworthy those are generated by the preceding rendering approaches with their conventions.

| Marching | Splatting | Pyrender | InLoc Pipeline | NRIW |
|---|---|---|---|---|
| CP, RC | CP, RC | CP, RC | Agnostic | – |

## 3.3 Experiments

We explore the rendering performance of various renderers used in the thesis, both in terms of the rendering quality compared to the respective real image and the actual time it takes to render an image of a given size. Underlying dependency on point cloud density is also touched. Finally, the influence of the renderers used for pose verification in the InLoc localization pipeline is examined.

### 3.3.1 Comparison of renderers

For **statistical comparison** of an RGB rendering produced by given renderer to the respective real-world image captured by a camera, we utilize two metrics—pixel-wise $L_1$ loss and Peak Signal to Noise Ratio (PSNR). The final metric value across a dataset is computed as the mean value of the loss per photo.

Since many models needed to be trained in a considerably costly process, only Image Matching Challenge data is used. The results can be seen in tables Table 3.4, Table 3.5, and Table 3.6. We compare not only three non-neural renderers plus three neural models trained on those renderers' data but also point cloud density. The density generally affects rendering times for non-neural renderers and transversely affects complete render times for a neural model inference.

Table 3.4: Comparison of $L_1$ (the smaller, the better) and PSNR (the bigger the better) metrics over the IMC Hagia Sophia collection. For the neural models, the point cloud of a given density is used for training and inference. Column Renderer uses notation $P$ (Pyrender), $S$ (Splatter), $M$ (Marcher), and three $N$ variants standing for NRIW trained on the respective renderer.

| Dataset | Density [%] | Points [M] | Renderer | $L_1$ | PSNR |
|---|---|---|---|---|---|
| Hagia Sophia | 25 | 1.25 | P | 39.93 | 13.94 |
| | | | S | 43.01 | 13.00 |
| | | | M | 35.83 | 14.69 |
| | | | N-P | 22.66 | 18.62 |
| | | | N-S | **21.80** | **18.73** |
| | | | N-M | 21.99 | 18.67 |
| | 50 | 2.49 | P | 36.19 | 14.66 |
| | | | S | 36.56 | 14.51 |
| | | | M | 34.25 | 15.12 |
| | | | N-P | 21.85 | 18.88 |
| | | | N-S | 21.20 | 19.01 |
| | | | N-M | **20.78** | **19.32** |
| | 100 | 4.98 | P | 36.04 | 14.70 |
| | | | S | 35.11 | 14.91 |
| | | | M | 37.19 | 14.37 |
| | | | N-P | **22.37** | 18.78 |
| | | | N-S | 22.85 | **19.29** |
| | | | N-M | 22.66 | 18.89 |

Table 3.5: Comparison of $L_1$ (the smaller, the better) and PSNR (the bigger, the better) metrics over IMC Grand Place collection. For the neural models, the point cloud of a given density is used for both training and inference. Column Renderer uses notation $P$ (Pyrender), $S$ (Splatter), $M$ (Marcher), and three $N$ variants standing for NRIW trained on the respective renderer.

| Dataset | Density [%] | Points [M] | Renderer | $L_1$ | PSNR |
|---|---|---|---|---|---|
| Grand Place | 25 | 1.09 | P | 60.81 | 10.59 |
| | | | S | 38.61 | 14.25 |
| | | | M | 39.74 | 14.04 |
| | | | N-P | 25.33 | 18.23 |
| | | | N-S | 23.75 | **20.01** |
| | | | N-M | **23.82** | 19.94 |
| | 50 | 2.19 | P | 57.32 | 11.20 |
| | | | S | 39.85 | 14.05 |
| | | | M | 40.39 | 13.88 |
| | | | N-P | 26.44 | 17.92 |
| | | | N-S | 23.21 | 20.12 |
| | | | N-M | **23.05** | **20.20** |
| | 100 | 4.37 | P | 57.33 | 11.23 |
| | | | S | 37.85 | 14.52 |
| | | | M | 39.90 | 13.74 |
| | | | N-P | 25.63 | 18.25 |
| | | | N-S | **22.98** | 19.86 |
| | | | N-M | 23.11 | **20.03** |

The metric values for an image pair in the tables are, analogously to how similarity in the InLoc verification step is computed, determined over positions of pixels of the rendered image that are not of the background color.

We can see performance gains from using neural models across the tables. Further, ray marching and point splatting methods are better than basic Pyrender, as discussed below. The relative difference also translates to neural models. It suggests that for methods presented in the thesis, the difference in the quality of training data used for training notably positively impacts the NRIW model. It also suggests that it can positively influence the pose verification step explored further in this chapter. Splatter and Marcher are close in performance, and it cannot be decided which is better, notably because they work similarly.

There are no such considerable differences between point cloud densities for the other data dimensions. The absence of substantial metric difference means that for a given scene/environment it may make sense to explore and use less than the full number of points if the time needed for producing one virtual view is essential. For subsequent experiments, in order to reduce the size of the space explored, we use only full-size point clouds at our disposal.

The biggest difference in $L_1$ metric between Pyrender and other non-neural renderers visible in data for Grand Place and Pantheon can be again explained by the screen space point size by which the renderer is parametrized. The size

Table 3.6: Comparison of $L_1$ (the smaller, the better) and PSNR (the bigger, the better) metrics over IMC Pantheon Exterior collection. For the neural models, the point cloud of a given density is used for both training and inference. Column Renderer uses notation $P$ (Pyrender), $S$ (Splatter), $M$ (Marcher), and three $N$ variants standing for NRIW trained on the respective renderer.

| Dataset | Density [%] | Points [M] | Renderer | $L_1$ | PSNR |
|---|---|---|---|---|---|
| Pantheon | 25 | 1.18 | P | 50.68 | 11.31 |
| | | | S | 38.20 | 14.24 |
| | | | M | 39.32 | 13.94 |
| | | | N-P | 22.87 | 18.91 |
| | | | N-S | **20.35** | 20.65 |
| | | | N-M | 21.28 | **21.00** |
| | 50 | 2.35 | P | 49.22 | 11.50 |
| | | | S | 42.39 | 13.19 |
| | | | M | 40.10 | 13.76 |
| | | | N-P | 23.10 | 18.75 |
| | | | N-S | 21.04 | 19.94 |
| | | | N-M | **20.99** | **20.18** |
| | 100 | 4.70 | P | 49.16 | 11.53 |
| | | | S | 39.06 | 14.07 |
| | | | M | 40.67 | 13.64 |
| | | | N-P | 18.86 | 22.84 |
| | | | N-S | **17.70** | **22.92** |
| | | | N-M | 18.24 | 21.72 |

is almost view-dependent as for different views, there may be different screen space `GL_POINTS` dimensions needed for getting flat-like surfaces, whereas for Splatter and Marcher, fixed per-point diameter can be determined, making this parametrization whole scene-dependent. It is also not that easy to determine the size in the case of Pyrender programmatically and in the aforementioned cases, the size was less suitable in total when combining views over the data as a whole, resulting in occlusion problem with visible points from normally non-visible parts of the scene that transversely taints metric computation; this effect is described in Introduction. For Splatter and Marcher, per-point diameters based on nearest neighbors within a given (simplified) point cloud are used with the 90-th percentile used for maximal point diameter rendered to filter out outliers that cause huge splats and spheres, respectively. These outlier points are more prevalent for point clouds generated by SfM and MVS, compared to scanner-generated ones.

For **visual comparison**, several dimensions are explored. In Fig. 3.9, point cloud density visual qualities are visualized using Pyrender across IMC collections and point cloud densities explored, with one image example per given image collection. The point cloud simplification method used for obtaining smaller density point clouds is the so-called voxel downsampling that should preserve point cloud structure. The method relies on decimating neighboring points into one point in

the resulting smaller point cloud based on fixed-size volumes called voxels. The impact on visual representation is noticeable, especially on the Grand Place render of the lowest density point cloud, as the background intentionally displayed in contrasting colors shines through the building uniformly across facade.

For more complex point clouds with hidden structures behind a camera facing simple surfaces such as a facade or a boundary of one vast internal space, such background visibility could easily be substituted by points of those otherwise non-visible scene portions. The problem should be resolved by using Splatter and Marcher renderers—it is, as can be seen on the per-collection example image comparison matrices Fig. 3.10, Fig. 3.11, and Fig. 3.12. Again, for the Pyrender image produced by the sparsest point cloud, "background" is more visible, especially in the case of the Grand Place example and in the case of the Pantheon example, a hidden structure example in the form of a column in front of the temple. As far as the Hagia Sophia image is concerned, the view depicts the surface sufficiently far away from the camera, so the background is not visible in this case. For the two non-neural renderers, the background visibility problem is, as expected, not present in any visibly excessive amount. The smaller sparsity influences those as well, though—using fewer points leads to more blurry renders, as the diameters of the remaining points are bigger. On the other hand, the rendering is faster and, as visible, still more continuous.

The neural rendering approach is represented by just one unspecified model in Fig. 3.10, Fig. 3.11, and Fig. 3.12 as visually the results of differently trained models' outputs are relatively similar; instead, the density dimension is displayed. The relative differences are illustrated on InLoc dataset in Fig. 3.13. The notable feature of the neural rendering method is the *ability to fill* (to some extent) portions of resulting render image without any respective point information, namely sky or gaps between screen space points, masking Pyrender's flaws. Splats and spheres are far better, but not perfect, rendering primitive from this perspective. However, there is always some space between neighboring elements stemming from the inability to cover a surface using circular objects with no overlaps or gaps. A neural renderer can mask those as well. This has an influence on the pose verification step explored in the following subsections.

Influence of the training data on outputs of the NRIW model is explored in Fig. 3.13).[26] Two views from the database are displayed with all non-neural and neural renderers. In the first one with chairs, one point with a comparatively larger diameter than most others is near the left-down corner of the Splatter and Marcher image. These bigger splats and spheres result in artifacts in neurally-generated images, as depicted in the second image row. These patches are something that the screen space renderer does not suffer from, as all points are of the same screen size. The advantage of variable diameters is visible in the second view. With the same number of points, the text on the board is sharper. Also, the floor gets some coverage compared to the Pyrender-generated image. The background-filled portions of Splatter-generated images are caused by the concrete actual point-splatting implementation, both Splatter and Marcher use the same underlying diameters.

[26] As the ARTwin dataset is not publicly available, it is excluded from visualizations here as its display is minimized in the thesis.


Figure 3.9: Visual comparison of point cloud density influence on rendering. The renders are produced by Pyrender (`GL_POINTS`) with contrastive background color to present the differences better. Point cloud simplification is done by Open3D's voxel downsampling to preserve the original structure with fewer points. The downsampling method proceeds in two steps; firstly, points are bucketed into 3D volumes called *voxels*, each resulting in exactly one point of the simplified output by averaging all points inside the source voxel in the second step. Since Pyrender has a fixed point size in screen space, its disadvantage can be seen—closest portions of the scene look sparser than the farther portions.

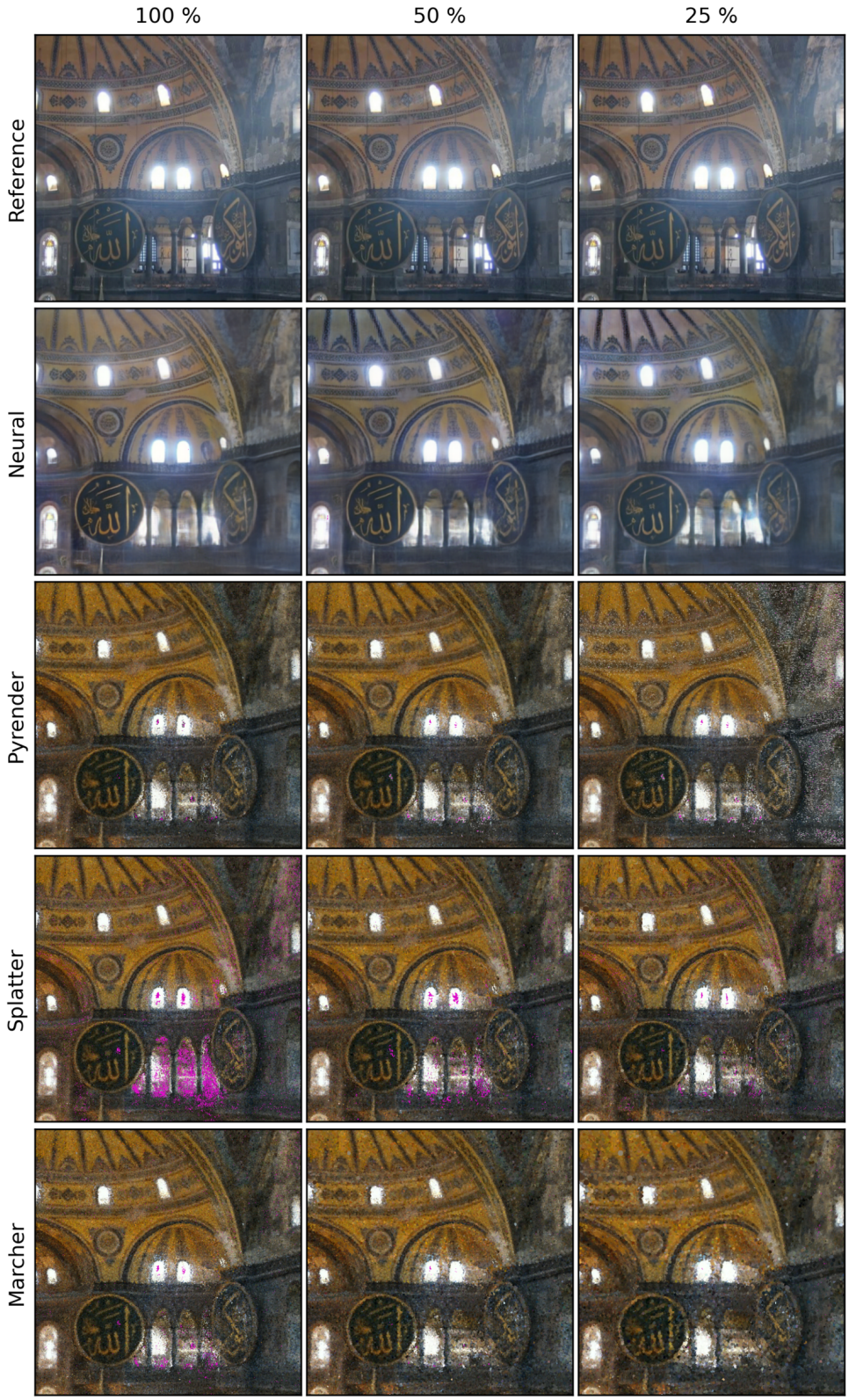


Figure 3.10: Visual comparison of various renderers and point cloud densities for the Hagia Sophia Collection. Contrastive background color is displayed, Open3D's voxel downsampling is used for point cloud simplification.


Figure 3.11: Visual comparison of various renderers and point cloud densities for Grand Place Brussels Collection. Contrastive background color is displayed, Open3D's voxel downsampling is used for point cloud simplification.


Figure 3.12: Visual comparison of various renderers and point cloud densities for Pantheon Collection. Contrastive background color is displayed, Open3D's voxel downsampling is used for point cloud simplification.

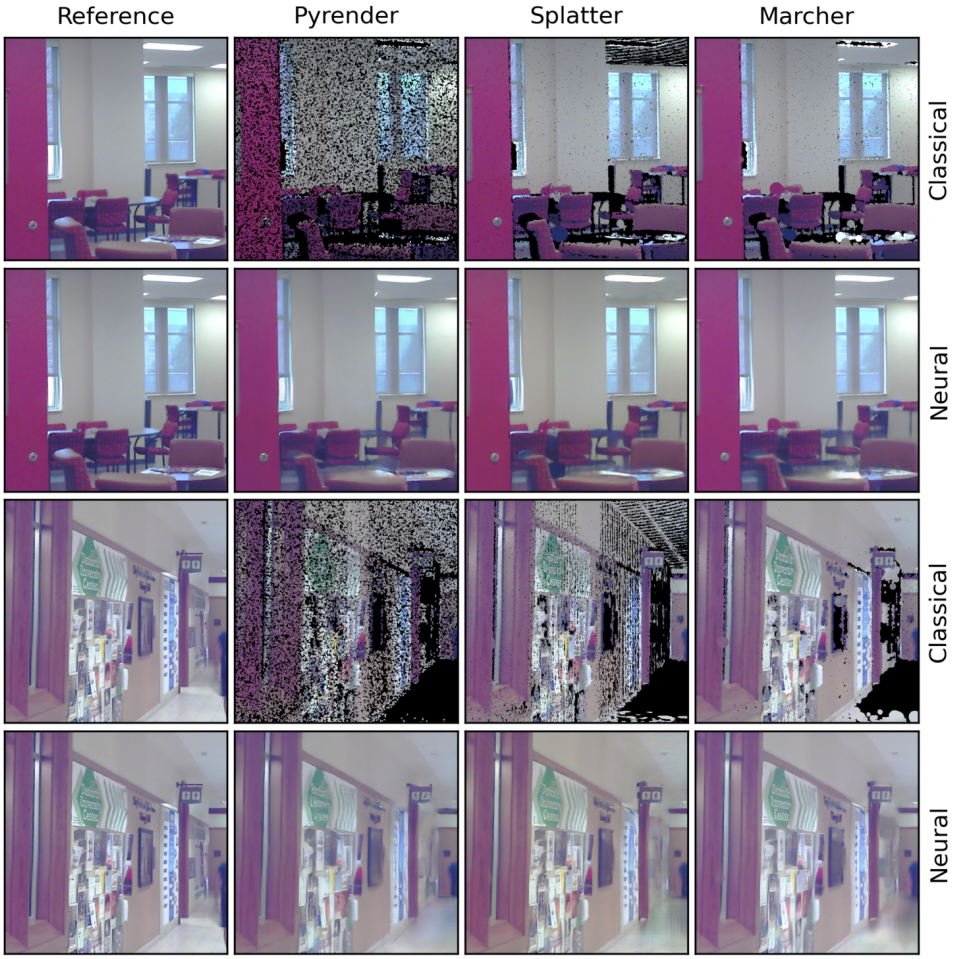


Figure 3.13: Visual comparison of various renderers for the InLoc dataset. For the indoor color profile, the default black background color is contrastive enough. The full point cloud was used for the visualizations. In the left-most column, repeated reference images for two different views are displayed. In the remaining columns, for a given view, up there always is an image generated by a non-neural type of renderer below which there is a render generated by NRIW trained on the data prepared by the respective non-neural renderer above.

The **computation rendering performance** relative comparison is explored in Table 3.7. Relative because computational time measurements are complex, and in a compute cluster environment that multiple users actively use, they are also inherently affected by many external entities. The influence can be alleviated to some extent by using proper OS time measuring clock type, but it is always better to ideally use the machine alone for the measuring task, which is hard to ensure on the cluster. This complexity is to be seen in the table with times varying across comparable output render dimensions and renderer types.

Across the datasets, the most consistent results are for C++ *Splatter* implementation. The smallest standard deviations show implementation consistency. The point cloud size influence is also visible, as for bigger renders and tens of millions of points, the rendering times get considerably higher.

The second most consistent results are for *Pyrender* rendering that internally also uses the standard OpenGL rendering pipeline and the standard and basic `GL_POINTS` rendering primitive. Since the implementation wrapper is in Python, we can see much higher standard deviations.

From the *neural* models, only one is picked for the measurements. The model size and inference implementation are the same for a dataset, not depending on the actual rendered data with which the model was trained. The times are comparable, though the second biggest standard deviations can be seen. The reason for shorter rendering times for the ARTwin dataset is unclear—the implementation is in Python, using many software layers underneath, including NVIDIA driver, so there may be the source of the difference, alongside demonstrating the complexity of time measurements. The times are considerably higher compared to those mentioned in the original paper, but that is to be linked to much higher render dimensions. Moreover, the entire neural rendering process includes rendering a given point cloud by some non-neural renderer before inference occurs, which may even double the rendering time in actual use cases.

C++ implementation of the Marcher does not use the standard OpenGL rendering pipeline but CUDA parallelization over pixels. Most notably, it iterates over points in the view frustum, so its rendering computational performance depends on the 3D structure of a given view of the scene. The view dependency is visible in the table, as the standard deviations are the biggest among renderer types. The implementation may hit some memory caching or another problem around the InLoc database scan size, as the mean rendering time and the standard deviation are orders of magnitude higher than everything else.

### 3.3.2 Comparison of localization approaches

In the localization pipeline, XYZcuts are a vital part of the database representation based on which a query image pose is calculated. Although there are XYZcuts present in the InLoc raw dataset, for other datasets explored in the thesis, they are not. Even for the InLoc dataset, some misalignments are hidden in verified scan poses, also noted in the InLoc algorithm Github repository issue. Thus, a way of computing this 3D data representation is devised and applied to all datasets, including the InLoc one. The process poses dependence of the localization pipeline on a non-neural renderer—as specified in InLoc localization pipeline, the XYZcut is computed utilizing the depth map produced by a renderer

Table 3.7: Measured mean rendering times for the renderers, collected with the thread CPU time clock, translating to the sum of the system and user CPU time of the main thread from which rendering is initiated. It does not include time elapsed during sleep, so it tries to avoid measuring disturbance from other processes running on the CPU. The clock is thus a rough approximation of what could be achieved when maximal performance is sought, and a real-time OS is used. The neural model is not distinguished here as the model size is the same even though tough weights vary based on the training data used. The neural rendering time represents solely the network inference, though preceding it, there must be an aligned triplet generated, which includes invocation of some non-neural renderer that takes also some time, as can be seen in the table. The dimension is the biggest one present in the data being rendered. The measurements were taken on an Intel Xeon E5-2698 and NVIDIA Tesla V100 GPU.

| Dataset | Points [M] | Dimension | Renderer | Time [ms] | $\sigma$ [ms] |
|---|---|---|---|---|---|
| Hagia Sophia | 5 | 1248 | Neural | 1 361 | 139 |
| | | | Pyrender | 783 | 111 |
| | | | Splatter | 156 | 13 |
| | | | Marcher | 1 031 | 397 |
| Grand Place | 4 | 1168 | Neural | 1 280 | 147 |
| | | | Pyrender | 679 | 82 |
| | | | Splatter | 123 | 7 |
| | | | Marcher | 1 448 | 570 |
| Pantheon | 5 | 1248 | Neural | 1 439 | 156 |
| | | | Pyrender | 857 | 108 |
| | | | Splatter | 173 | 4 |
| | | | Marcher | 446 | 127 |
| ARTwin | 27 | 1600 | Neural | 871 | 140 |
| | | | Pyrender | 1 380 | 145 |
| | | | Splatter | 1 072 | 21 |
| | | | Marcher | 679 | 382 |
| InLoc | 40 | 1600 | Neural | 1 749 | 328 |
| | | | Pyrender | 1 447 | 93 |
| | | | Splatter | 1 340 | 51 |
| | | | Marcher | $2\,054 \times 10^3$ | $1\,684 \times 10^3$ |

per localization database image. The depth map is used to transform per-pixel 3D points placed on a default plane in the distance of one from a camera center perpendicular to the given database image camera's optical axis.

In this section, three concepts are thus examined. Firstly, the influence of a non-neural renderer-based database representation on the whole localization process is explored on IMC image collections, end-to-end. Secondly, the influence of given database representation with fixed pose verification step renderer, including neural ones, on the localization performance is inspected on ARTwin data. Finally, the influence of solely pose verification renderer choice for one fixed database representation is considered for the InLoc dataset.

In Table 3.8, we can explore the localization on IMC image collection from the smallest datasets on top to the biggest at the bottom of the table. The overall rates of correctly localized queries are much smaller compared to ARTwin and InLoc data. That may be caused by the combination of three factors—several folds smaller dataset sizes; varying database image dimensions, some of which are as small as 100 pixels in each dimension; varying sensor types, sizes, and all adversarial effects of manual acquisition, such as extensive blur. The absolute error sizes, when compared to the size of the scene models, are relatively still small because, contrary to the other two datasets, IMC data cover enormous external or internal spaces over much bigger scale than mostly close looks at either manufacturing equipment or corridors and other, in nature, office spaces.

The table shows that the splatting InLoc variant is predominantly the most precise one with the most outliers on smaller precision thresholds among the smallest Hagia Sophia collection. The second most precise localization variant seems to be the Pyrender one. However, we will see that comparison to the marching variant may be affected by a small dataset size. The medians and means of Euclidean and angular distances support the thesis of database size influence on localization performance (aside from the simple idea that more database images mean a higher chance of retrieving one captured more closely, thus triangulating a more precise pose). The bigger the dataset size is, the smaller these statistics are. Also to be noted is the fact that having fewer database images to perform the image retrieval has a bigger influence on angular precision than Euclidean one.

Considering fixing the pose verification step and observing localization performance with varying database representations, Table 3.9 presents the results on the ARTwin dataset. The general positive impact of using neural rendering approach for pose verification is shown. Across the pose verification variants, the radii-based renderers perform better than the Pyrender ones, no matter whether neural or non-neural variant is considered. This further support the claim that better training data generation process positively affects neural model performance, though the percentual margin may not be that significant as these models can compensate for many imperfections in the deep buffer.

Further, for the candidate pose generation InLoc localization pipeline part, with a bigger dataset the marching-based InLoc Base variant shows its better performance rather closing to the splatting variant, both outperforming the Pyrender variant. From a median error point of view, having a bigger dataset is also ben-

Table 3.8: Evaluation of localization performance on IMC collections using the InLoc pipeline fully based on a given type of non-neural renderer, including the pose verification step. The performance is constituted by a percentual rate of correctly localized queries at a given precision threshold. General statistics of calculated poses in the form of mean and standard deviation for distance and angular distance are also displayed.

| Collection | Precision + Statistics | Pyrender | Splatter | Marcher |
|---|---|---|---|---|
| Hagia Sophia | 2.50 m, 7.5° | **2** | 0 | 0 |
| | 5.00 m, 10.0° | **4** | **4** | 2 |
| | 7.50 m, 15.0° | 18 | 16 | **20** |
| | 10.00 m, 20.0° | **52** | 44 | 50 |
| | 15.00 m, 30.0° | 86 | **90** | **90** |
| | 20.00 m, 30.0° | 86 | **90** | **90** |
| | Median [m] | 2.44 | **2.10** | 2.37 |
| | $\sigma$ [m] | 2.07 | **1.80** | 2.21 |
| | Median [°] | **19.88** | 20.28 | 20.07 |
| | $\sigma$ [°] | 21.09 | **19.50** | 21.42 |
| Grand Place | 2.50 m, 7.5° | 2 | **4** | 2 |
| | 5.00 m, 10.0° | 8 | **8** | 6 |
| | 7.50 m, 15.0° | 38 | **46** | **46** |
| | 10.00 m, 20.0° | 72 | **76** | 74 |
| | 15.00 m, 30.0° | **90** | 88 | 88 |
| | 20.00 m, 30.0° | **90** | 88 | 88 |
| | Median [m] | 1.63 | **1.50** | 1.75 |
| | $\sigma$ [m] | 2.17 | **1.45** | 2.29 |
| | Median [°] | 16.29 | **15.65** | 15.84 |
| | $\sigma$ [°] | 23.71 | **19.18** | 27.89 |
| Pantheon | 2.50 m, 7.5° | 14 | **26** | 22 |
| | 5.00 m, 10.0° | 34 | **44** | **44** |
| | 7.50 m, 15.0° | 72 | **82** | **82** |
| | 10.00 m, 20.0° | 88 | **92** | 90 |
| | 15.00 m, 30.0° | **100** | 98 | 96 |
| | 20.00 m, 30.0° | **100** | 98 | 96 |
| | Median [m] | 1.50 | **1.09** | 1.48 |
| | $\sigma$ [m] | 3.24 | **1.79** | 3.02 |
| | Median [°] | **10.04** | 11.99 | 10.52 |
| | $\sigma$ [°] | 21.80 | **5.44** | 19.08 |

Table 3.9: Evaluation of localization performance on the ARTwin dataset. The performance is constituted by a percentual rate of correctly localized queries at a given precision threshold. InLoc Base refers to a renderer type on which the localization database is constructed, pose verification renderer is denoted as $P$ (Pyrender), $S$ (Splatter), $M$ (Marcher), and three $N$ variants standing for NRIW trained on training data generated by the respective renderer.

| Pose Verification | InLoc Base | 0.25 m, 2° | 0.50 m, 5° | 5.00 m, 10° |
|---|---|---|---|---|
| P | Pyrender | 40.0 | 48.7 | 62.7 |
| | Splatter | **40.1** | **49.9** | 62.6 |
| | Marcher | 39.9 | 49.2 | **63.1** |
| S | Pyrender | **41.2** | 51.4 | 62.8 |
| | Splatter | 40.0 | 53.3 | **64.7** |
| | Marcher | 40.9 | **53.5** | 63.9 |
| M | Pyrender | 41.0 | 50.7 | 65.3 |
| | Splatter | **42.1** | **53.1** | **65.8** |
| | Marcher | 41.3 | 52.7 | 65.5 |
| N-P | Pyrender | 43.8 | 51.9 | 68.3 |
| | Splatter | 44.0 | 51.8 | **70.1** |
| | Marcher | **44.2** | **52.8** | 69.2 |
| N-S | Pyrender | 45.6 | 55.1 | 72.7 |
| | Splatter | **45.9** | 59.3 | 73.4 |
| | Marcher | 45.7 | **59.5** | **74.6** |
| N-M | Pyrender | 45.9 | 55.0 | 71.9 |
| | Splatter | 46.2 | **60.5** | 73.8 |
| | Marcher | **47.2** | 58.9 | **74.7** |

eficial. The mean of median Euclidean distance errors across the table is only 0.26 m, and the mean of median angular distance errors is as low as only 0.64°. These results are much better compared to those obtained on IMC collections, but as described in the paragraphs devoted to Table 3.8, ARTwin dataset consists of more database images with the same dimensions, consistent quality, and sensor parameters ensured by the image acquisition process. The scanner used for dataset generation uses more "normal" 2D cameras to create 360° RGB scans, so when the InLoc database is created, there are sometimes visible minor artifacts of this two-way process. An example of such an artifact together with localization visualization is in Fig. 3.14. In the light of such minor defects, the absolute errors are even more impressive.

Finally, for the InLoc dataset, the results are presented in Table 3.10. The percentual rates are comparatively worse than the one from previous research; the reason is that I faced a lot of issues around the InLoc dataset, including the mentioned skewed database positions, and I was not able to resolve it properly. Though lower in absolute numbers, the distortion is the same across all results; thus, the relative comparisons still pose valuable insights. Based on the outcomes

Figure 3.14: The first column shows the query photo, the second render of the best candidate position, the third a blend of the two preceding columns and the last column shows the database photo obtained from the image retrieval step. The upper row shows an error of 0.01 m and 0°, the second an error of 0.18 m and 1°. On the query image of the second row, there is also visible the relic of the reprojection from panorama photo to 2D photo on the tubes in the foreground.

Table 3.10: Percentual rate of correctly localized query images within the displayed distance and angular distance for the DUC1 floor. The table explores the behavior of fixed candidate positions localization part and multiple pose verification rendering methods denoted as $P$ (Pyrender), $S$ (Splatter), $M$ (Marcher), and three $N$ variants standing for NRIW trained on training data generated by the respective renderer.

| Rate [%] | P | S | M | N-P | N-S | N-M |
|---|---|---|---|---|---|---|
| 0.25 m, 2.0° | 22.2 | 19.2 | **22.7** | 26.8 | 21.2 | **29.8** |
| 0.50 m, 5.0° | 28.3 | 26.3 | **30.3** | 31.8 | 24.2 | **38.9** |
| 5.00 m, 10.0° | 37.4 | 36.4 | **41.9** | 43.4 | 31.8 | **51.0** |

of preceding experiments, with fixed localization dataset generation via splatting, we compare the performance of various pose verification approaches on the page `https://www.visuallocalization.net`.

For the InLoc dataset, the Marcher renderer performs best, outperforming both remaining non-neural renderers. The dominance over this dataset gets translated over to neural models as well. Even more than in the case of the ARTwin dataset, we can see the benefit of neural rendering in this case: ray marching varies between 7.1 % to 9.1 % for three precision thresholds. All three neural models show a positive impact over non-neural variants.

The splatting-based variants are the least performing among others on this dataset. The reason may be hidden in Lipponer's implementation, as some artifacts are visible in Fig. 3.13. That may arise from the number of points in the scan, as it is the highest among all the datasets in the thesis, or the format of the scans themselves, generated by Matlab in the raw dataset data and later transformed into PLY file format.

Despite all the complications with the InLoc dataset, radii-based renderers shown their value and benefits over all datasets, as well as neural rendering that further pushes localization performance rates.

# Conclusion

This thesis examines the usage of various rendering techniques in the InLoc algorithm solving the visual localization problem and its verification step. The point cloud rendering approaches, and later localization performance are evaluated on three different dataset types covering both exteriors and interiors; the InLoc implementation is generalized so that a general dataset, not only the one that InLoc was released with, can be input for localization.

Four different rendering approaches are utilized—as a baseline approach, the default OpenGL point rendering primitive `GL_POINTS` is used, further, point splatting and ray marching with signed distance fields are explored. Finally, aside from the three classical rendering approaches, a neural rendering deep neural network model is compared with the previous ones. As not all mentioned renderers were in existence prior to the thesis with sufficient performance to be able to render point clouds of sizes present in the datasets, third-party point splatting C++ implementation within a graphical interface is enhanced with headless rendering capabilities, the capability to read external point clouds and camera parameters, and output depth information for renders. The ray marching renderer is implemented in C++ and CUDA from scratch, eventually reusing the same components from the splatter enhancements. To the best of our knowledge, previously, there were no such implementations with these capabilities being able to render tens of millions of points in a reasonable time.

We considered the renderers from various angles—rendering performance from visual and statistical perspective, from computational performance, and finally, from influence on the localization performance. We show that aside from computational perspective, the four renderers split roughly into three groups: the predominantly least performant algorithm is Pyrender, followed by the Splatter and Marcher implementations with the NRIW model on the top.

*Pyrender* suffers from its slower implementation in Python and from the primitive it uses as the points have fixed size in the screen space, which defies perspective drawing principles, leading to visual artifacts in the form of occlusion problems.

*Splatter* and *Marcher* are less easy to separate. Their principle is similar as it uses diameters assigned to points and renders them as splats or spheres. In practical use cases, there are differences, however. The Splatter requires a normal vector per point on top of diameters to properly put a face to the splats. In the case of all datasets explored in the thesis, we did not encounter a situation where a dataset would simultaneously explore one space from the inside and outside. However, if this happens, Splatter would require additional functionality to dynamically compute normals per view or switch directions based on some condition fulfillment. On the other hand, Splatter shows less dependency on the view frustum contents, whereas Marcher performs much less consistently. For some views, the rendering time may be considerably longer than for others. There is room for improvements in the implementation that may mitigate this issue, including the possible memory caching boundary hit, causing extremely prolonged rendering times in some cases. Visually, Splatter can represent corners

and edges with less blur. Finally, both renderers increase the performance of the InLoc pipeline when used for the dataset's transformation into the InLoc format and for training a neural model used in the verification step.

The *NRIW* model further pushes localization performance due to its more realistically-looking rendering capabilities that get exploited in the verification step of the localization pipeline. The disadvantages and the price for the localization precision gains are speed, as the model needs, on top of its own slower runtime, a proxy render of a point cloud from a candidate position generated by another non-neural renderer, and also its scene dependency that requires training for every dataset explored. There has been considerable progress in the neural rendering field since work on the thesis started, so future work may explore these advancements. For instance, the Neural Radiance Fields (NeRF) models [Martin-Brualla et al., 2020; Mildenhall et al., 2020] and the Gaussian Splatting model [Kerbl et al., 2023] push neural rendering performance further. The latter model is highly related to the point splatting rendering approach explored here in the thesis.

To summarize, when maximal localization performance is sought, Splatter and Marcher help the localization pipeline's frontend, together with neural rendering based on the same. For concrete use cases, other differentiating factors can help to choose a specific renderer. When the time of answering a localization query is to be minimized, it may be worth sacrificing some precision by either using the non-neural renderers for the whole pipeline or, as we show, by lowering the number of points in the scene model that is from both statistical and visual point of view comparable with the advantage of faster rendering times. Future work may analyze the effect of lowering the resolution of the database and query images to inspect the computational performance further.

# Bibliography


Bart Adams, Richard Keiser, Mark Pauly, Leonidas Guibas, Markus Gross, and Philip Dutre. Efficient raytracing of deforming point-sampled surfaces. *Computer Graphics Forum*, 24, 09 2005. doi: 10.1111/j.1467-8659.2005.00892.x.

Tomas Akenine-Möller, Eric Haines, Naty Hoffman, Angelo Pesce, Michał Iwanicki, and Sébastien Hillaire. *Real-Time Rendering*. Taylor & Francis, United Kingdom, 4th edition, August 2018. ISBN 9781138627000. doi: 10.1201/b22086.

Relja Arandjelović and Andrew Zisserman. Three things everyone should know to improve object retrieval. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 2911–2918, 2012. doi: 10.1109/CVPR.2012.6248018.

Relja Arandjelović, Petr Gronat, Akihiko Torii, Tomas Pajdla, and Josef Sivic. Netvlad: Cnn architecture for weakly supervised place recognition. 2015. doi: 10.48550/ARXIV.1511.07247. URL `https://arxiv.org/abs/1511.07247`.

Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, sep 1975. ISSN 0001-0782. doi: 10.1145/361002.361007. URL `https://doi.org/10.1145/361002.361007`.

M. Botsch, A. Hornung, M. Zwicker, and L. Kobbelt. High-quality surface splatting on today's gpus. In *Proceedings Eurographics/IEEE VGTC Symposium Point-Based Graphics, 2005.*, pages 17–141, 2005. doi: 10.1109/PBG.2005.194059.

Mario Botsch, Michael Spernat, and Leif Kobbelt. Phong splatting. In Markus Gross, Hanspeter Pfister, Marc Alexa, and Szymon Rusinkiewicz, editors, *SPBG'04 Symposium on Point - Based Graphics 2004*, pages 25–32. The Eurographics Association, 06 2004. ISBN 3-905673-09-6. doi: 10.2312/SPBG/SPBG04/025-032.

Samarth Brahmbhatt, Jinwei Gu, Kihwan Kim, James Hays, and Jan Kautz. Geometry-aware learning of maps for camera localization, 2017. URL `https://arxiv.org/abs/1712.03342`.

Andrew Brock, Jeff Donahue, and Karen Simonyan. Large scale GAN training for high fidelity natural image synthesis. *CoRR*, abs/1809.11096, 2018. URL `http://arxiv.org/abs/1809.11096`.

Gaurav Chaurasia, Sylvain Duchêne, Olga Sorkine-Hornung, and George Drettakis. Depth synthesis and local warps for plausible image-based navigation. *ACM Transactions on Graphics*, 32, 2013. URL `http://www-sop.inria.fr/reves/Basilic/2013/CDSD13`. to be presented at SIGGRAPH 2013.

Aidan Clark, Jeff Donahue, and Karen Simonyan. Efficient video generation on complex datasets. *CoRR*, abs/1907.06571, 2019. URL `http://arxiv.org/abs/1907.06571`.

Carsten Dachsbacher, Christian Vogelgsang, and Marc Stamminger. Sequential point trees. *ACM Trans. Graph.*, 22(3):657–662, jul 2003. ISSN 0730-0301. doi: 10.1145/882262.882321. URL https://doi.org/10.1145/882262.882321.

Paul Debevec, Yizhou Yu, and George Borshukov. Efficient view-dependent image-based rendering with projective texture-mapping. In George Drettakis and Nelson Max, editors, *Rendering Techniques '98*, pages 105–116, Vienna, 1998. Springer Vienna. ISBN 978-3-7091-6453-2.

Paul E. Debevec, Camillo J. Taylor, and Jitendra Malik. Modeling and rendering architecture from photographs: A hybrid geometry- and image-based approach. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, page 11–20, New York, NY, USA, 1996. Association for Computing Machinery. ISBN 0897917464. doi: 10.1145/237170.237191. URL https://doi.org/10.1145/237170.237191.

Bastien Dechamps. Efficient camera pose hypothesis verification via neural rendering. Master's thesis, Ecole des Ponts ParisTech, 2020. URL https://drive.google.com/file/d/1LFOGpjomyUxY2b5f1Ir1XvoMOCEH7gzW/view.

H. Durrant-Whyte and T. Bailey. Simultaneous localization and mapping: part i. *IEEE Robotics & Automation Magazine*, 13(2):99–110, 2006. doi: 10.1109/MRA.2006.1638022.

S. M. Ali Eslami, Danilo Jimenez Rezende, Frederic Besse, Fabio Viola, Ari S. Morcos, Marta Garnelo, Avraham Ruderman, Andrei A. Rusu, Ivo Danihelka, Karol Gregor, David P. Reichert, Lars Buesing, Theophane Weber, Oriol Vinyals, Dan Rosenbaum, Neil Rabinowitz, Helen King, Chloe Hillier, Matt Botvinick, Daan Wierstra, Koray Kavukcuoglu, and Demis Hassabis. Neural scene representation and rendering. *Science*, 360(6394):1204–1210, 2018. doi: 10.1126/science.aar6170. URL https://www.science.org/doi/abs/10.1126/science.aar6170.

Alex Evans. Fast approximations for global illumination on dynamic scenes. In *ACM SIGGRAPH 2006 Courses*, SIGGRAPH '06, page 153–171, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933646. doi: 10.1145/1185657.1185834. URL https://doi.org/10.1145/1185657.1185834.

M. Fischler and R. Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981.

Yasutaka Furukawa and Jean Ponce. Accurate, dense, and robust multiview stereopsis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(8):1362–1376, 2010. doi: 10.1109/TPAMI.2009.161.

Volker Gaede and Oliver Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30(2):170–231, jun 1998. ISSN 0360-0300. doi: 10.1145/280277.280279. URL https://doi.org/10.1145/280277.280279.

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27, pages 1–9. Curran Associates, Inc., 2014. URL https://proceedings.neurips.cc/paper/2014/file/5ca3e9b122f61f8f06494c97b1afccf3-Paper.pdf.

Albert Gordo, Jon Almazan, Jerome Revaud, and Diane Larlus. End-to-end learning of deep visual representations for image retrieval, 2016. URL https://arxiv.org/abs/1610.07940.

Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen. The lumigraph. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, page 43–54, New York, NY, USA, 1996. Association for Computing Machinery. ISBN 0897917464. doi: 10.1145/237170.237200. URL https://doi.org/10.1145/237170.237200.

M. Gross and H. Pfister. *Point-based graphics*. Elsevier, Morgan Kaufmann Publishers, 2007. ISBN 0123706041.

Stephen Hausler, Sourav Garg, Ming Xu, Michael Milford, and Tobias Fischer. Patch-netvlad: Multi-scale fusion of locally-global descriptors for place recognition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14141–14152, 2021.

Peter Hedman, Julien Philip, True Price, Jan-Michael Frahm, George Drettakis, and Gabriel Brostow. Deep blending for free-viewpoint image-based rendering. 37(6), dec 2018. ISSN 0730-0301. doi: 10.1145/3272127.3275084. URL https://doi.org/10.1145/3272127.3275084.

Vu Hoang Hiep, Renaud Keriven, Patrick Labatut, and Jean-Philippe Pons. Towards high-resolution large-scale multi-view stereo. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1430–1437, 2009. doi: 10.1109/CVPR.2009.5206617.

Martin Humenberger, Gabriela Csurka Khedari, Nicolas Guerin, and Boris Chidlovskii. Methods for visual localization, 2021. URL https://europe.naverlabs.com/blog/methods-for-visual-localization/.

Janghun Hyeon, Joohyung Kim, and Nakju Doh. Pose correction for highly accurate visual localization in large-scale indoor spaces. In *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 15954–15963, 2021. doi: 10.1109/ICCV48922.2021.01567.

Hervé Jégou, Matthijs Douze, Cordelia Schmid, and Patrick Pérez. Aggregating local descriptors into a compact image representation. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 3304–3311, 2010. doi: 10.1109/CVPR.2010.5540039.

Alex Kendall and Roberto Cipolla. Geometric loss functions for camera pose regression with deep learning, 2017. URL https://arxiv.org/abs/1704.00390.

Alex Kendall, Matthew Grimes, and Roberto Cipolla. Posenet: A convolutional network for real-time 6-dof camera relocalization, 2015. URL https://arxiv.org/abs/1505.07427.

Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 3d gaussian splatting for real-time radiance field rendering. *ACM Transactions on Graphics*, 42(4), July 2023. URL https://repo-sam.inria.fr/fungraph/3d-gaussian-splatting/.

Laurent Kneip, Davide Scaramuzza, and Roland Siegwart. A novel parametrization of the perspective-three-point problem for a direct computation of absolute camera position and orientation. In *CVPR 2011*, pages 2969–2976, 2011. doi: 10.1109/CVPR.2011.5995464.

Karel Lebeda, Jiri Matas, and Ondrej Chum. Fixing the locally optimized ransac. In *British Machine Vision Conference*, 09 2012. doi: 10.5244/C.26.95.

David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, Nov 2004. ISSN 1573-1405. doi: 10.1023/B:VISI.0000029664.99615.94. URL https://doi.org/10.1023/B:VISI.0000029664.99615.94.

S. Marschner and P. Shirley. *Fundamentals of Computer Graphics*. CRC Press, 2021. ISBN 9781000426281. URL https://books.google.nl/books?id=QUQ3EAAAQBAJ.

Ricardo Martin-Brualla, Noha Radwan, Mehdi S. M. Sajjadi, Jonathan T. Barron, Alexey Dosovitskiy, and Daniel Duckworth. Nerf in the wild: Neural radiance fields for unconstrained photo collections, 2020. URL https://arxiv.org/abs/2008.02268.

Moustafa Meshry, Dan B. Goldman, Sameh Khamis, Hugues Hoppe, Rohit Pandey, Noah Snavely, and Ricardo Martin-Brualla. Neural rerendering in the wild. *CoRR*, abs/1904.04290, 2019. URL http://arxiv.org/abs/1904.04290.

Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis, 2020. URL https://arxiv.org/abs/2003.08934.

Shuxue Peng, Zihang He, Haotian Zhang, Ran Yan, Chuting Wang, Qingtian Zhu, and Xiao Liu. Megloc: A robust and accurate visual localization pipeline. *CoRR*, abs/2111.13063, 2021. URL https://arxiv.org/abs/2111.13063.

K. Perlin and E. M. Hoffert. Hypertexture. In *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '89, page 253–262, New York, NY, USA, 1989. Association for Computing Machinery. ISBN 0897913124. doi: 10.1145/74333.74359. URL https://doi.org/10.1145/74333.74359.

Francesco Pittaluga, Sanjeev J. Koppal, Sing Bing Kang, and Sudipta N. Sinha. Revealing scenes by inverting structure from motion reconstructions, 2019. URL https://arxiv.org/abs/1904.03303.

Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. ICLR, 2016. URL http://arxiv.org/abs/1511.06434.

Noha Radwan, Abhinav Valada, and Wolfram Burgard. VLocNet++: Deep multitask learning for semantic visual localization and odometry. *IEEE Robotics and Automation Letters*, 3(4):4407–4414, oct 2018. doi: 10.1109/lra.2018.2869640. URL https://doi.org/10.1109%2Flra.2018.2869640.

Gernot Riegler and Vladlen Koltun. Free view synthesis, 2020a. URL https://arxiv.org/abs/2008.05511.

Gernot Riegler and Vladlen Koltun. Stable view synthesis, 2020b. URL https://arxiv.org/abs/2011.07233.

Duncan Robertson and Roberto Cipolla. An image-based system for urban navigation. 01 2004. doi: 10.5244/C.18.84.

Lars Ruthotto and Eldad Haber. An introduction to deep generative modeling, 2021. URL https://arxiv.org/abs/2103.05180.

Torsten Sattler, Bastian Leibe, and Leif Kobbelt. Fast image-based localization using direct 2d-to-3d matching. In *2011 International Conference on Computer Vision*, pages 667–674, 2011. doi: 10.1109/ICCV.2011.6126302.

Torsten Sattler, Michal Havlena, Konrad Schindler, and Marc Pollefeys. Large-scale location recognition and the geometric burstiness problem. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1582–1590, 2016. doi: 10.1109/CVPR.2016.175.

Torsten Sattler, Bastian Leibe, and Leif Kobbelt. Efficient & effective prioritized matching for large-scale image-based localization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(9):1744–1756, 2017. doi: 10.1109/TPAMI.2016.2611662.

Torsten Sattler, Eric Brachmann, Giorgos Tolias, and Marc Pollefeys. Large-scale visual place recognition and image-based localization, 2019a. URL https://sites.google.com/view/lsvpr2019/home. publisher: ICCV 2019 Tutorial.

Torsten Sattler, Qunjie Zhou, Marc Pollefeys, and Laura Leal-Taixe. Understanding the limitations of cnn-based absolute camera pose regression, 2019b. URL https://arxiv.org/abs/1903.07504.

Torsten Sattler, Eric Brachmann, Giorgos Tolias, Yannis Avrithis, Marc Pollefeys, Zuzana Kukelova, and Sudipta N. Sinha. Large-scale visual localization, 2021.

URL https://sites.google.com/view/lsvpr2021. publisher: ICCV 2021 Tutorial.

Johannes L. Schönberger, Enliang Zheng, Jan-Michael Frahm, and Marc Pollefeys. Pixelwise view selection for unstructured multi-view stereo. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *Computer Vision – ECCV 2016*, pages 501–518, Cham, 2016. Springer International Publishing.

Johannes Lutz Schönberger and Jan-Michael Frahm. Structure-from-motion revisited. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

Johannes Lutz Schönberger, Enliang Zheng, Marc Pollefeys, and Jan-Michael Frahm. Pixelwise View Selection for Unstructured Multi-View Stereo. In *European Conference on Computer Vision (ECCV)*, 2016.

Johannes L. Schönberger and Jan-Michael Frahm. Structure-from-motion revisited. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4104–4113, 2016. doi: 10.1109/CVPR.2016.445.

Markus Schütz, Bernhard Kerbl, and Michael Wimmer. Rendering point clouds with compute shaders and vertex order optimization, 2021. URL https://arxiv.org/abs/2104.07526.

Stephen Se, D. Lowe, and J. Little. Global localization using distinctive visual features. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 1, pages 226–231 vol.1, 2002. doi: 10.1109/IRDS.2002.1041393.

Yoli Shavit and Yosi Keller. Camera pose auto-encoders for improving pose regression, 2022. URL https://arxiv.org/abs/2207.05530.

Christian Sigg, Tim Weyrich, Mario Botsch, and Markus Gross. GPU-Based Ray-Casting of Quadratic Surfaces. In Mario Botsch, Baoquan Chen, Mark Pauly, and Matthias Zwicker, editors, *Symposium on Point-Based Graphics*. The Eurographics Association, 2006. ISBN 3-905673-32-0. doi: 10.2312/SPBG/SPBG06/059-065.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. 2014. doi: 10.48550/ARXIV.1409.1556. URL https://arxiv.org/abs/1409.1556.

Vincent Sitzmann, Michael Zollhöfer, and Gordon Wetzstein. Scene representation networks: Continuous 3d-structure-aware neural scene representations, 2019. URL https://arxiv.org/abs/1906.01618.

Sivic and Zisserman. Video google: a text retrieval approach to object matching in videos. In *Proceedings Ninth IEEE International Conference on Computer Vision*, pages 1470–1477 vol.2, 2003. doi: 10.1109/ICCV.2003.1238663.

Hajime Taira, Masatoshi Okutomi, Torsten Sattler, Mircea Cimpoi, Marc Pollefeys, Josef Sivic, Tomás Pajdla, and Akihiko Torii. Inloc: Indoor visual localization with dense matching and view synthesis. *CoRR*, abs/1803.10368, 2018. URL `http://arxiv.org/abs/1803.10368`.

Hajime Taira, Ignacio Rocco, Jirí Sedlár, Masatoshi Okutomi, Josef Sivic, Tomás Pajdla, Torsten Sattler, and Akihiko Torii. Is this the right place? geometric-semantic pose verification for indoor visual localization. *CoRR*, abs/1908.04598, 2019. URL `http://arxiv.org/abs/1908.04598`.

Ayush Tewari, Ohad Fried, Justus Thies, Vincent Sitzmann, Stephen Lombardi, Kalyan Sunkavalli, Ricardo Martin-Brualla, Tomas Simon, Jason M. Saragih, Matthias Nießner, Rohit Pandey, Sean Ryan Fanello, Gordon Wetzstein, Jun-Yan Zhu, Christian Theobalt, Maneesh Agrawala, Eli Shechtman, Dan B. Goldman, and Michael Zollhöfer. State of the art on neural rendering. *CoRR*, abs/2004.03805, 2020. URL `https://arxiv.org/abs/2004.03805`.

Akihiko Torii, Josef Sivic, and Tomas Pajdla. Visual localization by linear combination of image descriptors. In *2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*, pages 102–109, 2011. doi: 10.1109/ICCVW.2011.6130230.

Eric Brachmann Torsten Sattler. Visual localization, 2018. URL `https://sites.google.com/view/visual-localization-eccv-2018/home`. publisher: ECCV 2018 Tutorial.

Jens Trogh, David Plets, Erik Surewaard, Mathias Spiessens, Mathias Versichele, Luc Martens, and Wout Joseph. Outdoor location tracking of mobile devices in cellular networks. *EURASIP Journal on Wireless Communications and Networking*, 2019(1):115, May 2019.

Heang K. Tuy and Lee Tan Tuy. Direct 2-d display of 3-d objects. *IEEE Computer Graphics and Applications*, 4(10):29–34, 1984. doi: 10.1109/MCG.1984.6429333.

Abhinav Valada, Noha Radwan, and Wolfram Burgard. Deep auxiliary learning for visual localization and odometry, 2018. URL `https://arxiv.org/abs/1803.03642`.

Carl Vondrick, Hamed Pirsiavash, and Antonio Torralba. Generating videos with scene dynamics, 2016. URL `https://arxiv.org/abs/1609.02612`.

Tim Weyrich, Simon Heinzle, Timo Aila, Daniel B. Fasnacht, Stephan Oetiker, Mario Botsch, Cyril Flaig, Simon Mall, Kaspar Rohrer, Norbert Felber, Hubert Kaeslin, and Markus Gross. A hardware architecture for surface splatting. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 26(3), August 2007.

Turner Whitted. An improved illumination model for shaded display. In *SIGGRAPH '79*, 1979.

Yuri D. V. Yasuda, Luiz Eduardo G. Martins, and Fabio A. M. Cappabianco. Autonomous visual navigation for mobile robots: A systematic literature review. *ACM Comput. Surv.*, 53(1), feb 2020. ISSN 0360-0300. doi: 10.1145/3368961. URL `https://doi.org/10.1145/3368961`.

Matthias Zwicker, Hanspeter Pfister, Jeroen Baar, and Markus Gross. Surface splatting. *Proceedings of the ACM SIGGRAPH Conference on Computer Graphics*, 2001, 08 2001. doi: 10.1145/383259.383300. URL `https://vcg.seas.harvard.edu/publications/surface-splatting/`.

Matthias Zwicker, Jussi Räsänen, Mario Botsch, Carsten Dachsbacher, and Mark Pauly. Perspective accurate splatting. In *Proceedings of the Graphics Interface 2004 Conference*, GI '04, page 247–254, Waterloo, CAN, 2004. Canadian Human-Computer Communications Society. ISBN 1568812272.

# List of Figures

# List of Tables

# List of Abbreviations

**CG** Computer Graphics

**CNN** Convolutional Neural Network

**CSG** Constructive Solid Geometry

**CV** Computer Vision

**DB** Database

**DNN** Deep Neural Network

**EWA** Elliptical Weighted Average

**FoV** Field of View

**GAN** Generative Adversarial Networks

**GPU** Graphics Processing Unit

**IMC** Image Matching Challenge

**MVS** MultiView Stereo

**NN** Neural Network

**NRIW** Neural Rerendering in the Wild

**OS** Operating System

**RGB(D)** Red Green Blue (Depth)

**SDF** Signed Distance Functions

**SfM** Structure from Motion

**VAE** Variational Autoencoder

**VGG** Visual Geometry Group