Heidelberg University
Institute of Computer Science
Computer Vision and Learning Lab

Master's Thesis

A 2D-to-3D Projection Approach to 3D Mesh Segmentation

A Case Study on Indoor Climbing Gyms

Name:	Tomáš Sláma
Matriculation Number:	3768224
Supervisor:	Prof. Dr. Carsten Rother
Date of submission:	January 15th, 2025

Declaration of Indenpendence

Abstract

We introduce a fully-automatic system for generating 3D segmented models of indoor scenes from images, focused on artificial indoor climbing walls and utilizing a new projection-based 3D segmentation algorithm. Our segmentation algorithm outperforms state-of-the-art transformer-based methods on indoor climbing walls, being able to utilize information from the 3D reconstruction itself, such as camera positions and images.

First, COLMAP is employed to reconstruct the 3D model, aligning the new reconstruction with an existing one. Next, a new 3D segmentation approach uses an image segmentation network and projection onto the mesh to detect individual holds. Finally, a clustering algorithm groups segmented holds into distinct routes based on color, type and spatial arrangement.

A proof-of-concept online application that uses this system has been developed independently to the thesis, and is currently available at `https://climbuddy.com/`, showcasing the potential of this approach to online climbing platforms.

Contents

1 Introduction

Bouldering is a form of rock climbing that takes place on artificial indoor walls without the use of ropes or harnesses. Climbers ascend *boulders/routes* that consist of same-colored *holds* and are typically no more than 4 meters tall (Figure 1.1). The boulders are created by *route setters*, who arrange holds on the wall to create boulders of varying levels of difficulty. Each problem is assigned a *grade*, indicating the level of difficulty, with higher grades representing more challenging climbs.

Climbers want to be able to track their progress, which is usually measured by the difficulty of boulders they can climb over time. A convenient way to do this is via a platform that displays the current boulders in the gym, so that the users can mark them as climbed and keep track of their climbing level, habits, socialize with friends, and more. While a few such platforms exist, they achieve this by either showing a 2D top-down map, or as individual annotated images showing the route from one particular angle, as seen in Figure 1.2.

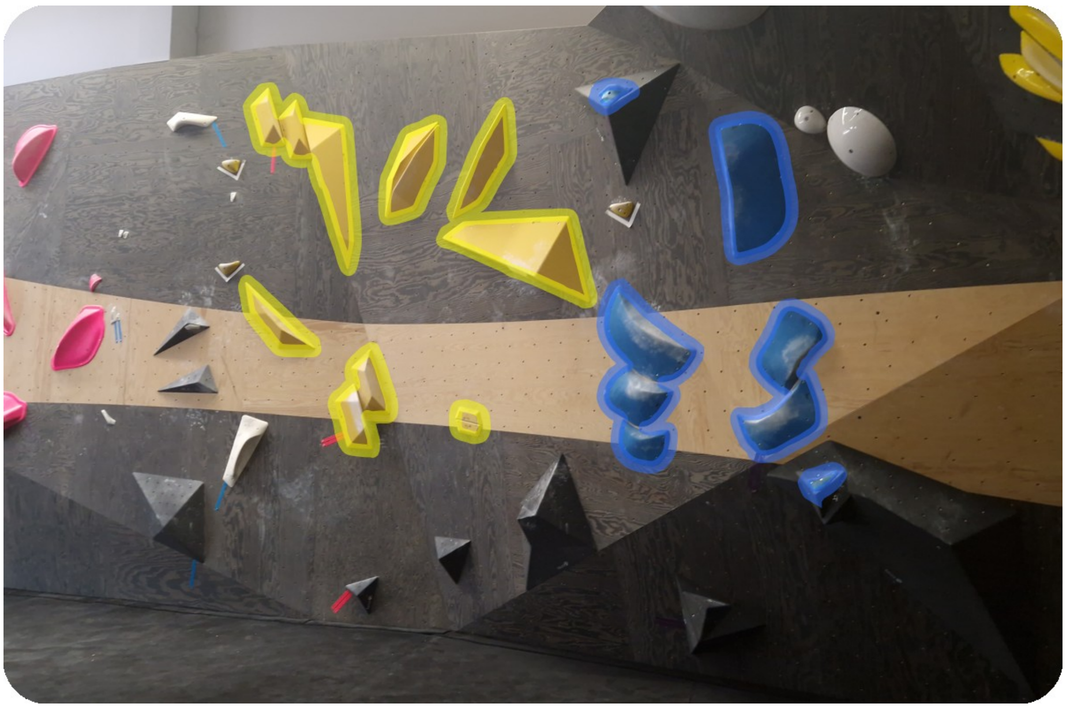
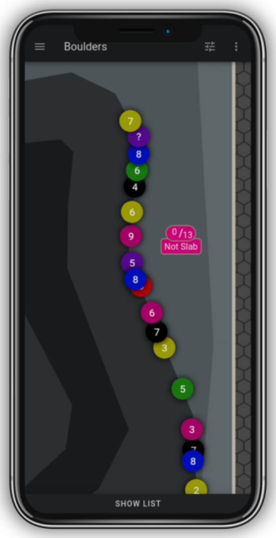
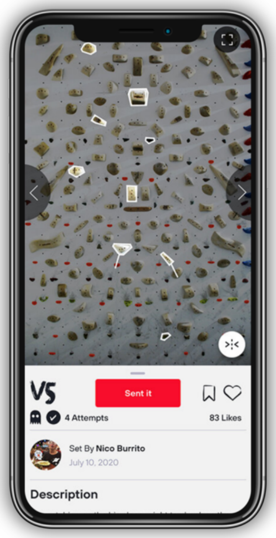


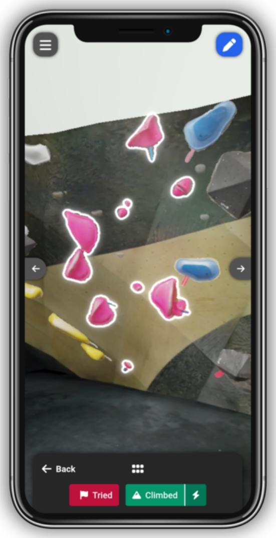
Figure 1.1: An overview of climbing terminology on an indoor climbing wall. Two boulders are highlighted: one consisting of yellow holds, the other of blue holds.



(a) **TopLogger** – routes are visible colored dots, determined by the hold colors.



(b) **Stökt** – routes are visible via annotated images with highlighted holds.



(c) **Ideal system** – routes are visible in an interactive way using their 3D model.

Figure 1.2: Current platforms for indoor bouldering (a), (b) and sample UI displaying a 3D model (c).

While platforms using both approaches have seen success, they suffer from a lack of interactivity that can only be solved by displaying a 3D model of the boulders, which allows for viewing a route from different angles, for the ability to study it ahead of climbing, etc. To obtain a segmented model while minimizing manual work, an ideal system would take in images of the newly created routes, and automatically produce a segmented textured model, separated into individual routes to be displayed online for users to interact with, without any additional input.

The goal of this thesis is the implementation of the core logic behind such system – given images from a commercial mobile phone, produce a segmented textured mesh of holds clustered into routes, to be usable as a backend for a potential online climbing platform. In order for this system to function, a number of problems have to be solved, as outlined in Table 1.1.

Table 1.1: Overview of the problems, tasks, solutions and backgrounds the thesis aims to solve.

Problem	**Task**	**Solution**	Background
Want to obtain a 3D model...	3D Reconstruction	COLMAP 3.1	(2.2)
... and segment into holds...	Image Segmentation	YOLO, SAM 3.2	(2.3, 2.4)
	Mesh Segmentation	New Algorithm . . . 3.3	(2.1, 2.3, 2.4)
... and combine into routes.	Instance Clustering	New Algorithm . . . 3.4	(2.1, 2.4)

The thesis is structured in the following way: Chapter 2 provides necessary theoretical background for Chapter 3, which covers the solution to the problems outlined in Table 1.1. Implementation details, experimental results for the new algorithms and a description of a proof-of-concept platform that uses the results are covered in Chapter 4, followed by conclusion in Chapter 5.

Further expanding on the contents of Chapter 3, Section 3.1 covers the use of a popular open-source 3D reconstruction software COLMAP [1, 2] to create the sparse reconstruction of the boulder scene. However, since a climbing gym contains many different sections, of which only a portion is changed at one time, the images have to be aligned to an existing reconstruction.

After obtaining the model, Sections 3.2 and 3.3 describe an approach to mesh instance segmentation using a new algorithm, which uses the camera positions and an image segmentation network to detect the objects in images, project them onto the mesh, cluster them based on their overlap and merge & threshold them to create the instance and semantic segmentation.

Finally, Section 3.4 covers clustering into routes based on their relative position and color/type by constructing a graph and performing graph clustering to obtain the individual routes.

2 Background

2.1 Graph Theory

Graphs are a fundamental and highly versatile abstraction for capturing many concepts discussed in this thesis, such as 3D meshes, overlaps of projections, distances and relations of objects in space and much more. They come in many different form, but the most common definition is the following [3].

Definition 2.1.1 (weighted graph)**.** A weighted graph $G = (V, E)$ consists of a set V of n vertices, a set $E \subseteq \binom{V}{2}$ of m edges, and an edge weight function $w : E \mapsto \mathbb{R}$.

Given a vertex u, we say v is its neighbour if $\{u, v\} \in E$. For our graph definition, this relation is symmetric – if u is a neighbour of v, then v is also a neighbour of u, making the graph *undirected*. We also define the function $N : V \mapsto 2^V$, which returns the neighbours of a given vertex, and $\deg(v) = |N(v)|$ as their number. A graph is *connected* if there is a *path* (sequence of non-repeating neighbouring vertices) between any two vertices.

2.1.1 Graph Clustering

The problem of graph clustering arises naturally in places such as parallel computing and social networks. In these cases, the graphs contain distinct groups/clusters of vertices that are more tightly connected and which we would like to identify, as seen in Figure 2.1.

Definition 2.1.2 (graph clustering)**.** A graph clustering of $G = (V, E)$ is a set of vertex subsets $\mathcal{C} = \{C_1, \ldots, C_k\}$ such that $C_i \subseteq V$ and $\forall v \in V$ exists exactly one C_i such that $v \in C_i$.

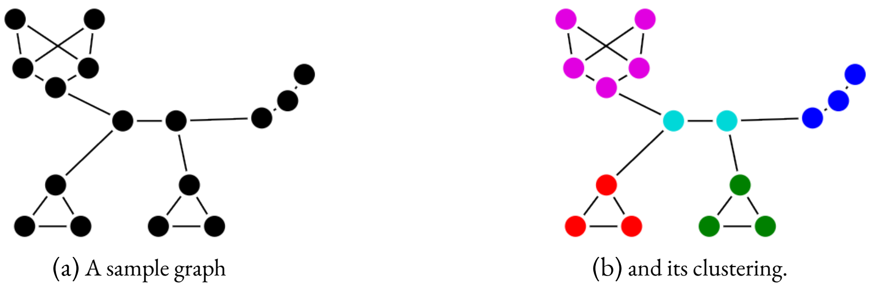

(a) A sample graph (b) and its clustering.

Figure 2.1: Graph (a) and one of its clusterings (b). Edge Weights are Euclidean distances between the vertices. Each cluster is represented by a single color.

The notion of graph clustering is useless without a way to measure its quality. We therefore define an *objective function* $f : \mathcal{C} \mapsto \mathbb{R}$, which, given a clustering, returns a value corresponding to how good it is. Higher values of the objective function correspond to better clusterings.

The choice of an objective function depends on the task at hand – for example, we could attempt to minimize the fraction of edges within clusters (*intra-cluster* edges). This is called the coverage $\operatorname{cov}(\mathcal{C})$ and is generally a poor choice, as the optimal solution for every connected graph yields only one cluster.

Although many objective functions exist, a good choice for common clustering problems is *modularity* [4]. The concept of modularity is rather straightforward – it is defined as coverage of the clustering minus the *expected* coverage of the clustering if edges were randomized. In other words, high modularity values correspond to clusterings with significantly more edges within clusters than expected if edges were randomized.

Let c_v denote the index of the cluster vertex v belongs to and a delta function $\delta(i,j)$ (being 1 if i equals j and 0 otherwise). Also, let the weight function extend to sets of edges $E' \subseteq E$ and vertices $v \subseteq V$ as

$$w(E') = \sum_{\{u,v\}\in E'} w(\{u,v\}) \qquad w(u) = \sum_{\substack{\{u',v\}\in E \\ u'=u}} w(\{u,v\}) \tag{2.1}$$

We can then define modularity for a connected weighted graph as [5]

$$\operatorname{mod}(\mathcal{C}) = \frac{1}{2w(E)} \sum_{u,v\in V} \left[w(\{u,v\}) - \frac{w(u)\,w(j)}{2w(E)} \right] \delta(c_i, c_j) \tag{2.2}$$

Unfortunately, finding a clustering that maximizes modularity is NP-hard [6], so we have to resort to using an approximation algorithm. One such algorithm is the Louvain method [7], which comes from the following two observations about Equation (2.2) [8]:

1. modularity will never increase by moving vertices between non-adjacent clusters, and
2. if maximum modularity gained by moving a vertex is negative, we found a local optimum.

The Louvain method uses these observations as follows: first, assume all vertices are clusters of size 1. In the first phase, go through each vertex, find an adjacent cluster that maximally increases modularity (observation 1) and move it if the increase is positive, repeating until no modularity increase can be made (observation 2). In the second phase, merge all clusters into single vertices, summing the weights of the merged edges, and go back to phase 1 until a single cluster remains.

This method is very fast, since the modularity change of moving a vertex can be calculated in constant time [7] (given pre-calculated values about the communities, which can be updated during the first phase), and yields a hierarchy of clusterings at the end of each phase 2, which can be used to select a clustering of a desired granularity.

We use the Louvain method in our 3D segmentation algorithm for clustering segmentation projections into a single instance (Section 3.3.2), with vertices corresponding to individual projections and edges their respective overlaps within the mesh. It is also used in the route clustering algorithm (Section 3.4), with vertices being the holds and edges the distances between them.

Besides a metric for measuring the quality of a clustering itself, we would also like to obtain one for measuring similarity of two clusterings, as we can use it to tune the hyperparameters of the clustering parts of our algorithms. In the most general formulation, we are given two clusterings of n items $\mathcal{C}_{\text{gt}}$ and $\mathcal{C}_{\text{pred}}$, and would like to measure their similarity $\in [-1, 1]$. Increasing values should correspond to more similar clusterings, obtaining 0 for randomly assigned clusters and 1 for a complete match.

Since the cluster ordering for $\mathcal{C}_{\text{gt}}$ and $\mathcal{C}_{\text{pred}}$ differs, we can't easily determine whether a particular item is in the "correct" cluster. Instead, we take a different approach – we count the number of item pairs with matching/differing clusters for $\mathcal{C}_{\text{gt}}$ and $\mathcal{C}_{\text{pred}}$ respectively, as these are order-independent. There are four options: items that are in the same clusters for both $\mathcal{C}_{\text{gt}}$ and $\mathcal{C}_{\text{pred}}$ (n_{11}), items that are in different clusters for both (n_{00}), and items that are in the same cluster for only one (n_{01} and n_{10}). We call n_{00} and n_{11} the positive occurrences (where the clusterings agree), while n_{01} and n_{10} the negative occurrences (where they disagree).

Measuring the positive occurrences over the total yields the Rand index [9]

$$\text{RI} = \frac{n_{00} + n_{11}}{n_{00} + n_{01} + n_{10} + n_{11}} \tag{2.3}$$

Formulated in this way, value 0 corresponds to clusterings with no positive occurrences instead of random assignment, which is not desirable. Furthermore, the value for random assignment is not fixed in this formulation, as it depends on the number of items and clusters. We address this issue by using the adjusted Rand index [10], which offsets the Rand index by its expected value, assuming the number and size of clusters stays the same

$$\text{ARI} = \frac{\text{RI} - \mathbb{E}[\text{RI}]}{\max\{\text{RI}\} - \mathbb{E}[\text{RI}]} = \frac{2(n_{00}n_{11} - n_{01}n_{10})}{(n_{00} + n_{01})(n_{01} + n_{11}) + (n_{00} + n_{10})(n_{10} + n_{11})} \tag{2.4}$$

We use ARI to measure the performance of the route clustering algorithm (Section 3.4), as well as the objective function for optimizing the route clustering hyperparameters (Section 4.5).

2.2 3D Reconstruction

The task of 3D reconstruction is, in the most general formulation, the process of obtaining information about an object's properties in 3D space. A common example is using 2D images to reconstruct a colored point-cloud in the shape of the object. While common, 3D reconstruction is not limited to just images and point-clouds – we can use other sources of information, such as GPS or LIDAR, and obtain more advanced properties, such as the recently introduced Neural Radiance Fields [11] and Gaussian Splats [12].

For the purposes of 3D segmentation, which is the main focus of this thesis, we are interested in the classic problem – given a set of 2D images, reconstruct a 3D mesh (which we can then segment). To do this, we use COLMAP [1, 2] for sparse reconstruction (camera positions, sparse point-cloud), and OpenMVS [13] for dense reconstruction (dense point-cloud, mesh).

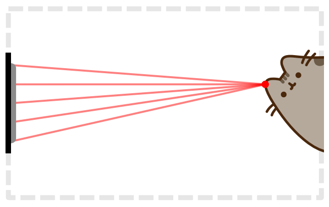
This section covers camera basics (2.2.1), which form the basis for 2D/3D projections used throughout the thesis, as well as sparse reconstruction (2.2.2), which is related to Section 3.1. Since understanding dense reconstruction is not relevant to the thesis, we treat it as a black box and skip it for brevity.

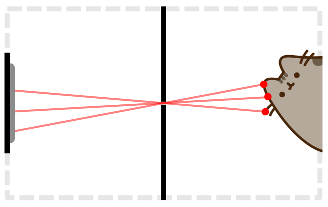
2.2.1 Camera basics

The human eye is an immensely complicated organ with a simple task – take in light and use it to form an image of the outside world. It does this by channeling the incoming light onto a light-sensitive layer at the back of the eye, which in turn transmits the information to the brain. Modern digital cameras solve this task in a similar manner – channel the incoming light via a lens onto a photo-sensitive sensor at the center of the camera, which processes the light per sensor pixel and produces a digital image [14].

To begin, it is useful to consider the simplest camera model – using only the sensor (2.2a). This is as simple as non-functional, since light rays from one point of an object will hit all parts of the sensor in view, producing a completely blurry image.

To solve this problem, we can filter rays emitted by a point of the object and only permit one to go through. We can do this by using a *pinhole* (2.2b) – an infinitesimal hole that only permits a single ray from each point to hit the sensor of the camera.


(a) Sensor camera


(b) Pinhole camera

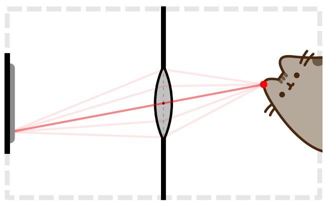

(c) Lens camera

Figure 2.2: Three camera models and their effect on rays reflected from points of an object.

While functional in theory, real-world sensors are discrete and finitely sensitive – detecting single rays of light through a tiny hole results in sub-par image quality. In practice, we will use a *lens* (2.2c) to channel multiple light rays from a single point on the object to a single part of the sensor. This improves the quality of the image, but results in a more complicated camera model.

We will first focus on the pinhole camera model, as seen in Figure 2.3. More specifically, we would like to formulate the relation between points in 3D space and their corresponding 2D projections in the image plane. Since $[x, y, f]^T$ and $[X, Y, Z]^T$ are multiples of each other, we obtain

$$\frac{x}{f} = \frac{X}{Z} \iff x = f\frac{X}{Z} \qquad \frac{y}{f} = \frac{Y}{Z} \iff y = f\frac{Y}{Z} \tag{2.5}$$

This makes p the origin of the image plane, which is unintuitive, since the pixel $(0, 0)$ should be in the corner. To account for this, we further offset the equations by $\mathbf{p}$, obtaining

$$x = f\frac{X}{Z} + \mathbf{p}_x \qquad y = f\frac{Y}{Z} + \mathbf{p}_y \tag{2.6}$$

Re-writing the equations using homogeneous coordinates, we obtain

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \cong \underbrace{\begin{bmatrix} f & 0 & \mathbf{p}_x \\ 0 & f & \mathbf{p}_y \\ 0 & 0 & 1 \end{bmatrix}}_{\substack{\text{intrinsic} \\ \text{matrix}}} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{2.7}$$

where $\cong$ is equality up to homogeneity, i.e. $\forall w \neq 0 : [x, y, 1] \cong [wx, wy, w]$.

We have formulated the *intrinsic matrix* $\mathbf{K}$, which encodes intrinsic information of the camera model. This includes the *focal length* f (distance from the pinhole to the image plane) the *principal point* $\mathbf{p}$ and can be further modified to include more complex intrinsic, such as sensor skew and magnification [14].

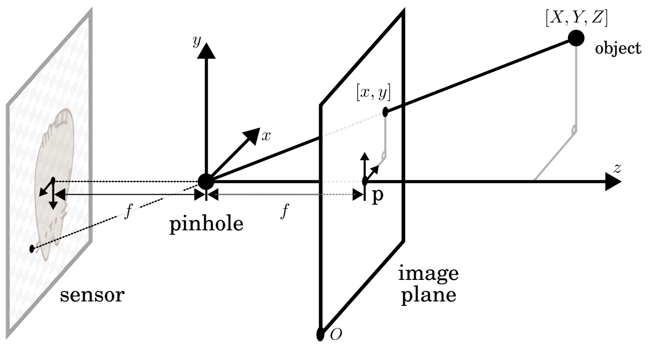


Figure 2.3: The pinhole camera model. f is the *focal distance* of the camera, p is the *principal point*, O is the origin of the *image plane*, with the sensor rotated 180° relative to the image plane.

The calculation above assumes that the position of the pinhole is located in the origin of the 3D space and is oriented in the axis directions, which usually isn't the case – in general, the camera can be in any place in the scene, which can be done by offsetting it by a vector $\mathbf{T}^{3\times 1}$ and rotating by a rotation matrix $\mathbf{R}^{3\times 3}$. Here we can again use homogeneous coordinates to simultaneously perform both translation and rotation, forming an *extrinsic matrix* and obtaining

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \cong \mathbf{K} \underbrace{\begin{bmatrix} \mathbf{R} & \mathbf{T} \end{bmatrix}}_{\substack{\text{extrinsic} \\ \text{matrix}}} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{2.8}$$

Getting back to the lens camera, the most important difference between the lens model and the pinhole model is lens distortion. Due to the imperfection of the lens, a variety of artifacts can be introduced to the image, with the most common one being radial distortion.

For radial distortion located at the center of the image, we can model its effects by changing pixels at radius r to radius r_{new} (assuming normalized units) in the following way [14]

$$r_{\text{new}} = r \cdot (Ar^3 + Br^2 + Cr + D) \tag{2.9}$$

where A, B, C, D are the distortion coefficients. Sample distortions can be seen in Figure 2.4.

Combining the described pinhole model with radial distortion, we have described the default camera model that COLMAP uses for 3D reconstruction. Although omitting more advanced distortion types and camera parameters might seem like we are wasting potential accuracy, their parameters have to be reconstructed[1] – a simpler model has fewer parameters to reconstruct and thus a smaller chance of incorrect/degenerate values, which is worth the accuracy trade-off.

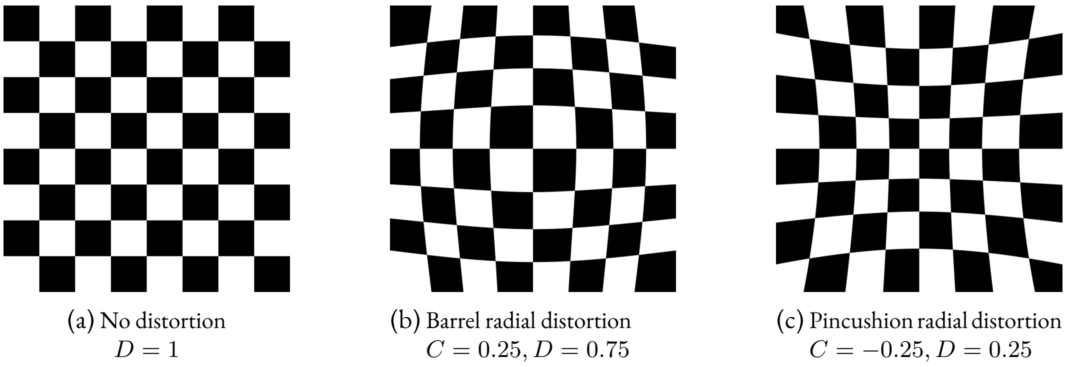


Figure 2.4: Radial distortion of a checkerboard pattern. C, D are values for Equation (2.9), with A, B being 0. Positive values correspond to barrel distortion, negative to pincushion distortion.

[1]If you have calibrated the camera in advance and know the distortion parameters in advance, you can opt for more advanced camera models – COLMAP supports OpenCV's [15] camera model, which also models tangential distortion and a few other parameters.

2.2.2 Sparse reconstruction

Variables used in the intrinsic matrix, such as focal length f and principal point p, are usually stored in image metadata and obtaining them is simple. This is, however, not the case for the variables of the extrinsic matrix (rotation, translation) and the distortion coefficients, which have to be inferred to form a good model of a scene.

This is the main goal of sparse reconstruction – infer the transformations of the cameras, distortion coefficients and other unknowns in the camera model, all of which are necessary for reconstructing the complete 3D model. This can be done in the following steps, which we will cover in the following sections.

1. **Feature Extraction** – extract features ("interesting points") from all images.
2. **Feature Matching & Verification** – find matching features across different images, making sure that the relative transformation of the cameras is geometrically sensible.
3. **Image Registration & Bundle Adjustment** – iteratively register images to the scene and perform joint adjustment of all scene parameters (triangulated points, extrinsics, etc...).

Feature Extraction

Given an arbitrary 2D image, not all of its pixels are equally "interesting" – a pixel from a white background is not recognizable across multiple images, while a pixel on the corner of an object likely is. A well-designed feature has to be recognizable from one image to another, which means that it must be invariant to *affine transformations* (rotation, translation, change in perspective), *color transformations* (brightness, contrast, saturation), and should be robust wrt. *image noise*.

For 3D reconstruction, the SIFT [16, 17] (Scale-Invariant Feature Transform) features have been used ubiquitously, since they require no machine learning, are invariant to the aforementioned transformations and are simple to implement and match. While alternatives such as ORB [18] and LIFT [19] exist, SIFT is well-suited to the task of 3D reconstruction.

To detect SIFT features, we start by calculating difference-of-Gaussians on multiple image scales (octaves). This is done by repeated convolution of the images with the Gaussian kernel with deviation σ and subtraction. Each scale layer is further multiplied by a factor of k.

Given the input image $I(x, y)$, Gaussian kernel $G(x, y, \sigma)$ and the convolution operator $*$, we can obtain the scale space image $L(x, y, \sigma)$ and the difference-of-Gaussians image $D(x, y, \sigma)$ as follows [16]

$$L(x,y,\sigma) = G(x,y,\sigma) * I(x,y) \qquad \text{with } G(x,y,\sigma) = \frac{1}{2\pi\sigma^2}e^{-(x^2+y^2)/2\sigma^2} \tag{2.10}$$

$$\begin{aligned} D(x,y,\sigma) &= (G(x,y,k\sigma) - G(x,y,\sigma)) * I(x,y) \\ &= L(x,y,k\sigma) - L(x,y,\sigma). \end{aligned} \tag{2.11}$$

The local local minima and maxima of D across its 3×3 neighbourhood (also including adjacent images in the same scale) images are taken as potential keypoints and their positions are further refined using the Taylor expansion and filtered based on their local contrast.

To prevent detected keypoints that lie on an edge, which are considered undesirable due to their edge position being arbitrary, the principal curvatures for each keypoint are calculated and those with low ratios of the minimum/maximum curvature are rejected.

With the positions of features identified, we have to calculate their descriptors, which we will use to match the same features across multiple images later on. To ensure correct matches, descriptors should be invariant to the transformations which we've previously mentioned – the value of the descriptor should stay approximately the same, such that it's occurrence in another image will be its nearest neighbour.

To calculate a keypoint's descriptor, we first calculate its rotation and magnitude based on the image gradient, calculated via Equation (2.12) and Equation (2.13) respectively. We then use a region around the image based on the gradient to calculate an orientation histogram, using its peaks to create one or more keypoints of different orientations but same positions.

$$m(x,y) = \sqrt{(L(x+1,y) - L(x-1,y))^2 + (L(x,y+1) - L(x,y-1))^2} \tag{2.12}$$

$$\theta(x,y) = \tan^{-1}(L(x,y+1) - L(x,y-1))/(L(x+1,y) - L(x-1,y)) \tag{2.13}$$

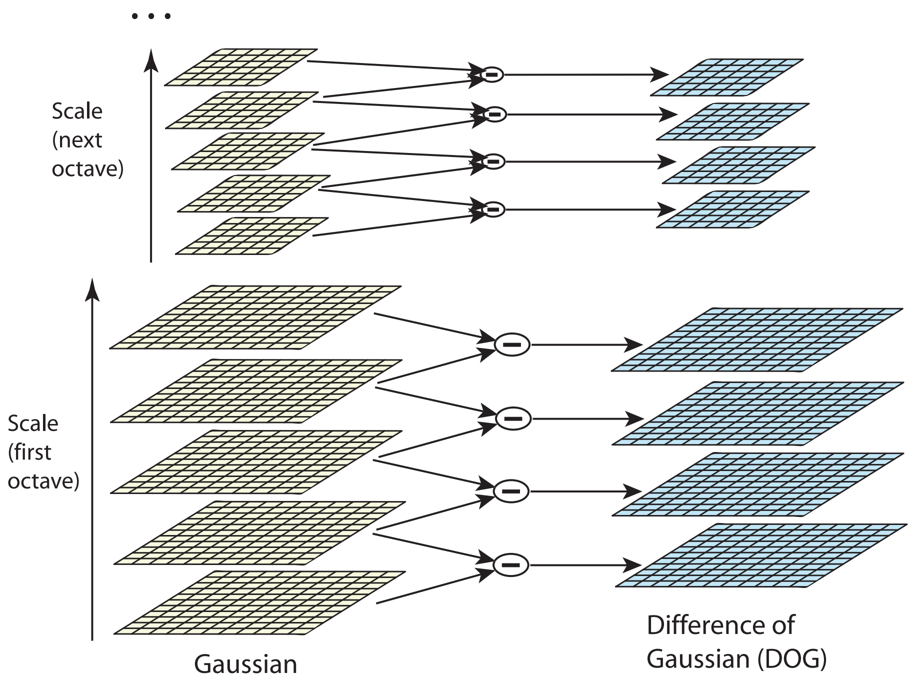


Figure 2.5: Difference of Gaussians across multiple scales (Lowe [16], page 6).

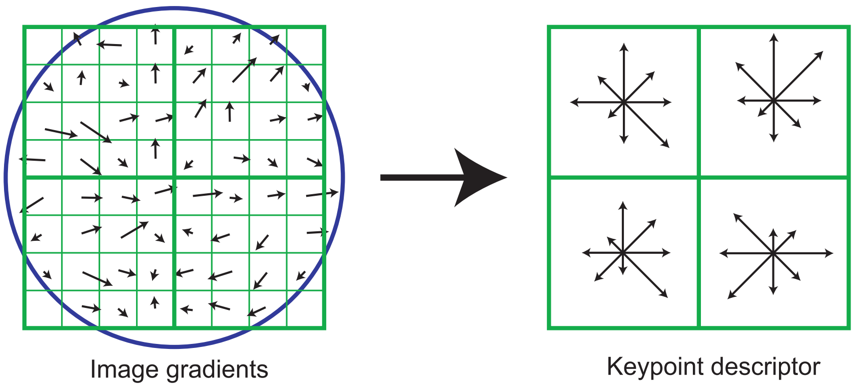


Figure 2.6: Local image gradients forming a keypoint descriptor, weighted by a Gaussian distribution centered around the keypoint. The values used for SIFT are 16×16 gradients and 4×4 histogram array (Lowe [16], page 15).

Finally, with the keypoints' rotation and scales identified, we define the descriptor as an array of histogram of image gradients, weighted using a Gaussian distribution around the keypoint position, as seen in Figure 2.6.

Feature Matching

With features identified across images, the next step is to find images with matching features. COLMAP implements multiple strategies to do this, the most common being *exhaustive* (all image pairs are compared) and *sequential* (all image pairs in a sliding window are compared). It also contains more advanced matching methods, such as a *vocabulary tree* [20], which we omit since they are mostly useful for extremely large datasets (1 000+ images). *Exhaustive* is generally a safe choice for smaller numbers of images (< 500), while *sequential* is a good choice when the images are ordered.

To match two sets of features A and B, a naïve brute-force algorithm would, for each feature of A, find the nearest neighbour in B and vice versa, matching if the two features are the other's nearest neighbour. Formally, two features $a \in A, b \in B$ match, if

$$\forall b' \in B : \mathrm{d}(a, b) \leq \mathrm{d}(a, b') \quad \wedge \quad \forall a' \in A : \mathrm{d}(a, b) \leq \mathrm{d}(a', b) \tag{2.14}$$

for a suitable distance function (e.g. Euclidean distance).

Despite seeming inefficient, a implementation GPU allows for massive parallelization and is very fast for a smaller number of features. However, due to the quadratic nature of this approach, the performance degrades significantly with increasing number of features, making it only usable for a limited number of images.

A better approach is to build a data structure to store keypoints of image B, and use the keypoints of image A as queries, and vice versa. Since in theory, finding a nearest descriptor is linear, our data structure will have to return an *approximate* nearest neighbour, only requiring that it is *likely* the nearest neighbor. Formally, for $\varepsilon \approx 0$, we require that

$$p(\exists b' \in B : d(a,b) > d(a,b')) < \varepsilon \quad \wedge \quad p(\exists a' \in A : d(a,b) > d(a',b)) < \varepsilon \tag{2.15}$$

COLMAP uses the FLANN library [21], which implements a number of approximate nearest neighbour approaches; specifically, it matches SIFT descriptors using a randomized K-d tree.

Constructing a randomized K-d tree is simple – recursively split the data, choosing the dimension with the largest variance, until the leafs contain less than the specified number of values. Since we're splitting the data in half at each step of the process, the depth of the tree will be logarithmic, allowing for quick traversal and value retrieval.

Querying is done by linearly comparing values in leaf nodes nearest to the queried value – larger number of explored leafs increases the probability that the nearest neighbour will be found, but also increases the query time. Due to the logarithmic depth of the tree, this operation also runs in logarithmic time.

In practice, two additional things are done to ensure better matching performance. First, the values of the split positions impact precision of the queries – a query close to a split has a higher change of not finding its nearest neighbor, since it might be in a leaf the query won't explore. To address this issue, COLMAP builds multiple trees (i.e. a forest) and determines the closest neighbour by querying all of them simultaneously, which significantly reduces the impact of split positions and can be done in parallel.

Second, each query also retrieves the second nearest neighbour and rejects the match if the ratio between the distances to the nearest neighbour and the second nearest is large (Lowe [16] uses 0.8). This seemingly small change leads to a drastic improvement in match accuracy, since we're filtering weaker queries that have multiple plausible matches.

Geometric Verification

Even with the match filtering techniques described in the previous sections, invalid matches will inevitably be introduced. To preserve only valid matches, we need to perform geometric verification by finding a relative 3D transformation between the cameras that has a sufficient number of matching keypoints – by checking that such transformation exists, we can safely eliminate incorrect matches and obtain relative transformations between cameras.

The relation between images observing the same 3D scene is referred to as *epipolar geometry* [14], with a sample scenario captured in Figure 2.7. For simplification, we omit camera distortion effects, and only discuss the pinhole model as outlined in Figure 2.3.

Without loss of generality, let the first camera $\mathbf{c}_1$ be located at origin with an identity rotation, and the second camera $\mathbf{c}_2$ have rotation $\mathbf{R}$ and translation $\mathbf{T}$ from origin. Assuming that the point is at a distance d_1 from $\mathbf{c}_1$ and d_2 from $\mathbf{c}_2$ and its image plane coordinates are $\mathbf{x}_1$ and $\mathbf{x}_2$ for both cameras respectively, using Equation (2.7) for 2D-to-3D projection, we obtain

$$\mathbf{p} = d_1 \underbrace{\mathbf{K}_1^{-1}\mathbf{x}_1}_{\hat{\mathbf{x}}_1} = \mathbf{R}(d_2 \underbrace{\mathbf{K}_2^{-1}\mathbf{x}_2}_{\hat{\mathbf{x}}_2}) + \mathbf{T} \tag{2.16}$$

which, using the $[\mathbf{T}]_\times$ notation for the matrix form of a cross product

$$[\mathbf{T}]_\times = \begin{bmatrix} 0 & -\mathbf{T}_z & \mathbf{T}_y \\ \mathbf{T}_z & 0 & -\mathbf{T}_x \\ -\mathbf{T}_y & \mathbf{T}_x & 0 \end{bmatrix} \tag{2.17}$$

can be further modified into the following constraint

$$\hat{\mathbf{x}}_2^T \ \underbrace{[\mathbf{T}]_\times \mathbf{R}}_{\substack{\text{essential}\\ \text{matrix}}} \ \hat{\mathbf{x}}_1 = 0 \tag{2.18}$$

The *essential matrix* $\mathbf{E}$ encodes the relation between two points of calibrated cameras. We can equally formulate the constraint using a *fundamental matrix* $\mathbf{F}$, which will include the intrinsic matrices and work on the image coordinates directly, as

$$\mathbf{x}_2^T \underbrace{\mathbf{K}_2^{-T}\ \mathbf{F}\ \mathbf{K}_1^{-1}}_{\substack{\text{fundamental}\\ \text{matrix}}} \mathbf{x}_1 = 0 \tag{2.19}$$

Satisfying this constraint obtains the relative transformation and rotation of the cameras, which can be done by expanding the constraint into homogeneous equations and using matched points to solve them. Although in general, 8 points are required to obtain a unique solution (assuming calibrated cameras), algorithms that require less points (i.e. the 5-point algorithm [22]) but obtain multiple solutions can also be used, trading uniqueness for speed.

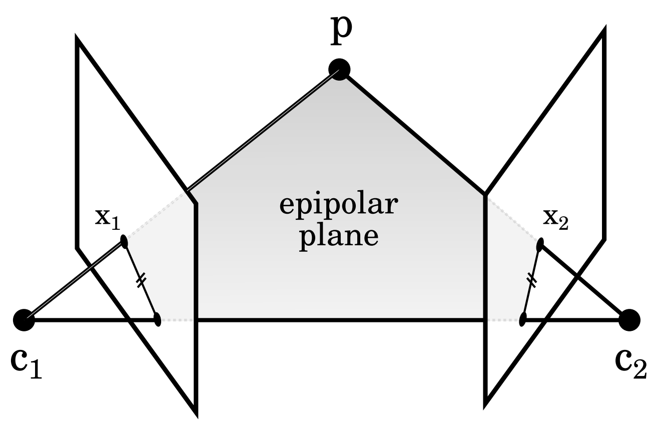


Figure 2.7: Epipolar geometry of two cameras $\mathbf{c}_1, \mathbf{c}_2$ observing a common point $\mathbf{p}$ as $\mathbf{x}_1, \mathbf{x}_2$ respectively.

Finally, with the relative transformation obtained, we can filter out outlier matches as those whose projections are sufficiently far apart after the relative transform and projections are applied to the cameras, as well as use the inlier matches as triangulated points.

Image Registration & Bundle Adjustment

Equipped with an algorithm for calculating relative transformations of cameras from matched keypoints, it might seem like we can repeatedly register images into the scene until all are added. This will, however, fail in practice – while the relative transformation between two cameras might only contain a slight error, successive transformations will magnify this error and lead to an incorrect reconstruction.

To solve this, we need to use a global operation called *bundle adjustment*, which will fix these relative errors in a joint manner. More specifically, given a set of registered cameras $\mathcal{C}$ and triangulated points $\mathcal{P}$, each of which is seen by a subset of the registered cameras, we would like to minimize the difference between the point's measured image plane position (i.e. the keypoint position in the image), and the position projected from its corresponding 3D point. This is referred to as *reprojection error*, and can be formulated as

$$\sum_{c \in \mathcal{C}} \sum_{\substack{\mathbf{p} \in \mathcal{P} \\ c \text{ sees } \mathbf{p} \text{ as } \mathbf{x}}} \left|\left|\mathbf{K}_c \begin{bmatrix} \mathbf{R}_c & \mathbf{T}_c \end{bmatrix} \mathbf{p} - \mathbf{x}\right|\right|_2^2 \tag{2.20}$$

with $\mathbf{x}$ being the measured position for the camera c in context.

Equation (2.20) can be modified to use a more complex camera model, as well as other scene parameters, but in either case we obtain a non-linear least squares minimization problem, which can be tackled by algorithms such as, in the case of COLMAP, the Levenberg–Marquardt algorithm [23, 24] using its Ceres solver implementation [25]. Furthermore, as each camera observes only a limited subset of points, bundle adjustment problems tends to be sparse, which can be exploited, in the case of large numbers of cameras and points, by more advanced sparse non-linear least squares optimizers.

COLMAP's reconstruction loop can thus be described as repeated *image registration*, followed by *triangulation* of points as direct products of satisfying Equation (2.19) for a given point $\mathbf{p}$, and subsequent *bundle adjustment*. In practice, the bundle adjustment is run in an amortized way only after growing the reconstruction by a certain percentage [1, 2]. It also utilizes more robust methods, such as RANSCAC [26] for outlier filtering, such as in the case of finding the relative camera transformations for two-view geometry.

2.3 Machine Learning

Machine learning has been an essential tool in addressing problems where classical approaches perform poorly. This is especially true for tasks whose optimal solutions can't be formulated precisely, but can be described using a large number of inputs (X) with known solutions (y).

A few examples where machine learning performs better than classical approaches are

- **image classification** – X is the image and y the image class,
- **chess engine** – X are chess positions, y are move predictions, and
- **weather forecast** – X is weather data, y is future forecast prediction.

2.3.1 Neural Network

With the rapid advancements in computer hardware, classical learning-based methods (SVMs, decision trees, LDAs), have been largely surpassed by neural networks, which have proven to be superior in a wide variety of tasks, including image & mesh segmentation.

Artificial Neural networks consist of *neurons*, which aim to mimic their biological counterparts. A neuron takes in values and, based on its weights, bias and activation function, outputs values to neurons it is connected to. More formally, a neuron's computation looks as follows

$$y = \varphi(\underbrace{zw + b}_{\tilde{z}}) \tag{2.21}$$

with z input vector, w weights vector, b bias and φ activation function such as ReLU or $\tanh$. A fully-connected neural network (Figure 2.8) is a collection of neurons, consisting of an input layer, a number of fully-connected hidden layers and an output layer.

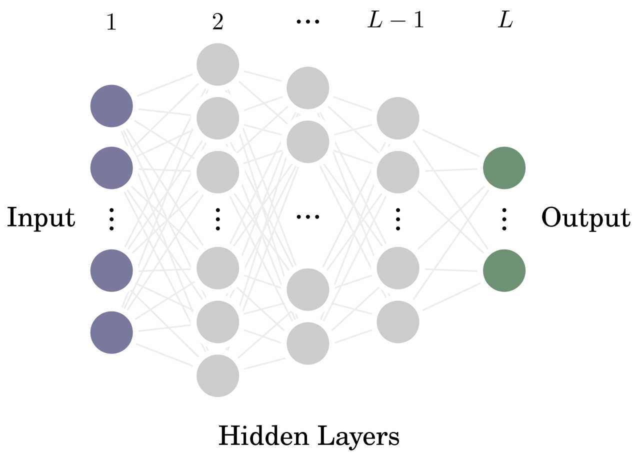


Figure 2.8: A fully-connected neural network with L layers, with each dot representing a neuron. Biases are omitted for simplicity, but can be implemented using a neuron of constant 1 in each layer.

The size of the input and output layers are determined by the size of input/output data of the task, while the number and size of the hidden layers are arbitrary, usually depending on the task complexity. The activation functions of the last network layer usually differ from the ones of the other layers and depend on the type of task – *classification problems* commonly use the softmax function and produce a probability distribution on the classification classes, while *regression problems* prefer the identity function.

Performing *inference* using a fully connected neural network is simple – feed data to the input layer, perform the layer computations and output the neuron values in the output layer. Labeling inputs and weights of each layer as $\mathbf{z}_i$, $\mathbf{w}_i$ respectively, and using the fact that the bias of each layer can be represented by a single neuron with a constant value of 1, we can describe the inference as

$$\begin{aligned}\mathbf{y} &= \varphi_L(\mathbf{z}_L\mathbf{w}_L) \\ &= \varphi_L(\varphi_{L-1}(\mathbf{z}_{L-1}\mathbf{w}_{L-1})\mathbf{w}_L) \\ &= \varphi_L(\varphi_{L-1}(\dots\varphi_1(\mathbf{z}_1\mathbf{w}_1)\dots\mathbf{w}_{L-1})\mathbf{w}_L)\end{aligned} \tag{2.22}$$

Training the network is more complicated – ideally, we would like to teach it to mimic the training data in a way that generalizes to inputs it hasn't seen during training. The idea is to perform inference on training data, calculate the *loss* $\mathcal{L}$ using a loss function suitable to the task, and adjust the network weights such that the loss decreases. The most common approach is *gradient descent*, which finds the local minimum of the loss function by moving in the direction of its derivative by adjusting the weights as follows

$$\boxed{\mathbf{w}_l^{\text{new}} = \mathbf{w}_l^{\text{old}} - \tau\,\Delta\mathbf{w}_l^{\text{old}}} \qquad \text{with } \Delta\mathbf{w}_l^{\text{old}} := \frac{\partial\mathcal{L}}{\partial\mathbf{w}_l^{\text{old}}} \tag{2.23}$$

where τ is the *learning rate* for the algorithm, since we can't move continuously, but instead by discrete *steps/iterations*.

Since the inference is just a successive application of functions, we can utilize the chain rule on Equation (2.22) to obtain the derivatives necessary to perform the gradient descent on our network. Assuming a ReLU activation function, we obtain

$$\frac{\partial\mathcal{L}}{\partial\mathbf{w}_l} = \frac{\partial\mathcal{L}}{\partial\tilde{\mathbf{z}}_l}\cdot\frac{\partial\tilde{\mathbf{z}}_l}{\partial\mathbf{w}_l} = \boxed{\tilde{\delta}_l\cdot\mathbf{z}_{l-1}} \qquad \text{with } \tilde{\delta}_l := \frac{\partial\mathcal{L}}{\partial\tilde{\mathbf{z}}_l} \tag{2.24}$$

$$\tilde{\delta}_l = \frac{\partial\mathcal{L}}{\partial\tilde{\mathbf{z}}_{l+1}}\cdot\frac{\partial\tilde{\mathbf{z}}_{l+1}}{\partial\mathbf{z}_l}\cdot\frac{\partial\mathbf{z}_l}{\partial\tilde{\mathbf{z}}_l} = \boxed{\tilde{\delta}_{l+1}\cdot\mathbf{w}_{l+1}^T\cdot\operatorname{diag}(\varphi_l'(\tilde{\mathbf{z}}_l))} \tag{2.25}$$

As seen in Equation (2.24) and Equation (2.25), we need both $\mathbf{z}_l$ and $\tilde{\mathbf{z}}_l$ to calculate the partial derivatives of $\mathbf{w}_l$. However, per Equation (2.25), we also need the partial derivatives of $\tilde{\mathbf{z}}_l$ to calculate the partial derivatives of $\tilde{\mathbf{z}}_{l-1}$. These constraints give rise to *backpropagation* (Algorithm 1) [27], which is the algorithm that we'll use for training the network in a way that reduces repetitive calculations.

Algorithm 1 Backpropagation Algorithm

```
Require: Training data $(X, y)$, learning rate $\tau$
 1: // This is important – we need to break symmetry!
 2: Initialize weights $\mathbf{w}$ randomly

 3: for iterations $t$ from 1 to $T$ do
 4:     // Forward pass
 5:     for each layer $l$ from 1 to $L$ do
 6:         Compute $\mathbf{z}_l$ and $\tilde{\mathbf{z}}_l$ using $X$
 7:     end for

 8:     // Backward pass
 9:     $\Delta \mathbf{w}_{\tilde{i}}^{(t)} = 0$
10:     Set $\tilde{\delta}_L$ based on the loss function and $y$
11:     for each layer $l$ from $L$ to 1 do
12:         $\Delta \mathbf{w}_l \mathrel{+}= \tilde{\mathbf{z}}_{l-1} \cdot \tilde{\delta}_l$
13:         $\tilde{\delta}_{l-1} = \tilde{\delta}_l \cdot \mathbf{w}_l^T \cdot \mathrm{diag}\big(\varphi'_{l-1}(\tilde{\mathbf{z}}_{l-1})\big)$
14:     end for

15:     // Gradient descent
16:     for each layer $l$ from 1 to $L$ do
17:         $\mathbf{w}_l \mathrel{-}= \tau \Delta \mathbf{w}_l$
18:     end for
19: end for
```

The algorithm first performs a forward pass (line 4), calculating $\mathbf{z}_l$ and $\tilde{\mathbf{z}}_l$. Next, a backward pass (line 8) alternates between calculating $\Delta \mathbf{w}_l$ and $\tilde{\mathbf{z}}_{l-1}$, since they depend on one another and calculating them backwards prevents repetitive calculations, hence the name. Finally, gradient descent is performed (line 15), repeating for T iterations.

As we've previously mentioned, both the loss function and the activation function depend on the task we're trying to solve, which influences what line 10 calculates. Taking regression as an example, we can set the loss function to $\mathcal{L}(a, b) = \frac{1}{2}(a - b)^2$, the activation function of the last layer to identity and obtain

$$\mathcal{L}(\mathbf{z}_L, y) = \frac{1}{2}(\mathbf{z}_L - y)^2 \qquad \mathbf{z}_L = \varphi_L(\tilde{\mathbf{z}}_L) = \tilde{\mathbf{z}}_L \tag{2.26}$$

$$\boxed{\tilde{\delta}_L = z_L - y} \tag{2.27}$$

For classification, all calculations we've done except Equation (2.26) and Equation (2.27) are valid; the last two need to be recalculated with the classification loss function (negative log likelihood) and activation function (softmax) of the last layer.

2.3.2 Convolutional Neural Networks

The fully connected neural network, as described in the previous section, is in theory a *universal approximator* – given enough neurons, it can approximate any function to arbitrary accuracy. In practice, however, it will perform very poorly on certain tasks, such as anything image-related, e.g. image classification. Since the dimensionality of the image (2D for black & white) doesn't match the dimensionality of network output (1D), the image either has to be flattened and discard the spatial information, or the network structure has to change to accommodate for this.

Convolutional Neural Networks (CNNs) introduce new sparsely connected 2D layers (so far we've only seen fully connected ones), which preserve the spatial information and thus significantly improve network performance. They are based on *convolutions*; for the purposes of CNNs, we define the convolution operation of an image I and a kernel K of size $[2a+1, 2b+1]$ as

$$K * I(x,y) = \sum_{i=-a}^{a} \sum_{j=-b}^{b} K(i,j) \cdot I(x-i, y-j) \tag{2.28}$$

Ordinarily, a kernel would be hand-picked based on the desired image operation, e.g. a convolution with a Gaussian kernel would blur the image. In the case of a CNN, the kernel is learned instead, which allows the network to extract relevant image features.

A sample architecture of a CNN classification network for 2 classes is described in Figure 2.9. It contains an *input*, corresponding to the input of the image. The input layer is connected to a *convolution* layer which performs convolution on the image with a learned kernel. Next, a *pooling* layer downsamples the image using the maximum of the input. Finally, two *fully connected* layers, with the last one using a softmax activation function conclude the network, producing a probability distribution on the classification classes.

In practice, the number of convolution layers, as well as the depth of the network would be significantly larger. One of the first implementations, LeNet-5 [28], used for digit recognition, used [input $\rightarrow$ 6 conv. $\rightarrow$ pool $\rightarrow$ 16 conv. $\rightarrow$ pool $\rightarrow$ fully connected], achieving a 0.35% error rate on the MNIST [29] dataset.

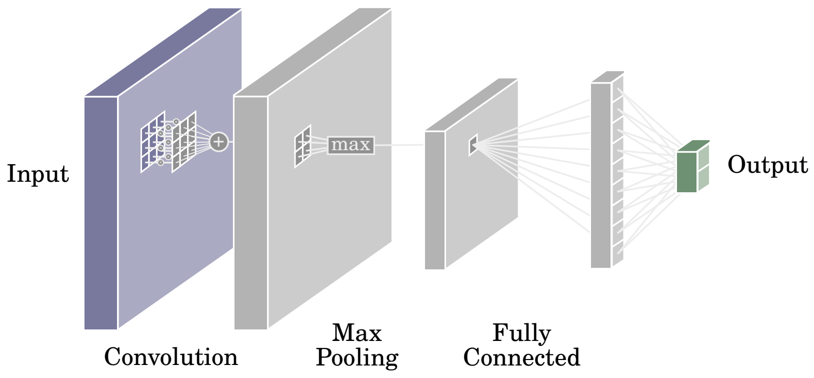


Figure 2.9: Sample architecture of a CNN. Operations of the layers are shown on select neurons.

2.3.3 Image Detection & Segmentation

In the field of computer vision, the task of *image detection* involves determining the bounding boxes and classes of objects in an image (Figure 2.11a). Although this is sufficient for some problems, others require finer detail in the form of a precise segmentation mask for the detected objects, in which case the task becomes *image segmentation* (Figure 2.11b).

Image detection and segmentation are closely connected – the task of creating a segmentation mask is strictly mode difficult than creating a bounding box around the object. The architecture of a modern image detector [30–33] consists of the following parts, usually as some forms of CNNs

- **backbone** – pre-trained on a large dataset, extracts key features,
- **neck** – bridge between backbone and head; gathers important features for the current task,
- **head** – predicts bounding boxes, classes and confidence scores from the gathered feature.

Due to the modularity of this architecture, the head can be designed to perform a number of image-related tasks, such as pose estimation, object tracking, and object detection/segmentation. This means that, besides the head, the network architecture can stay unchanged (Figure 2.10).

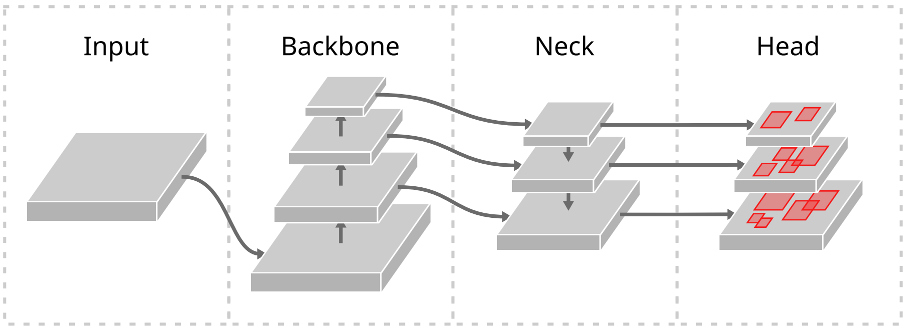
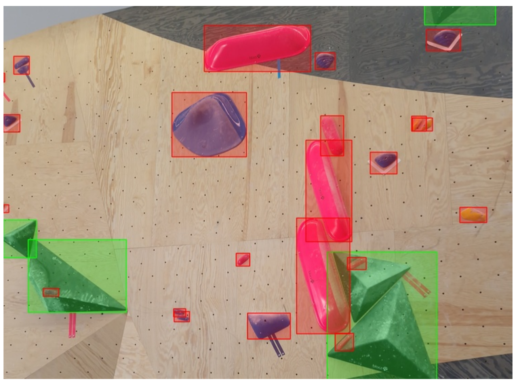
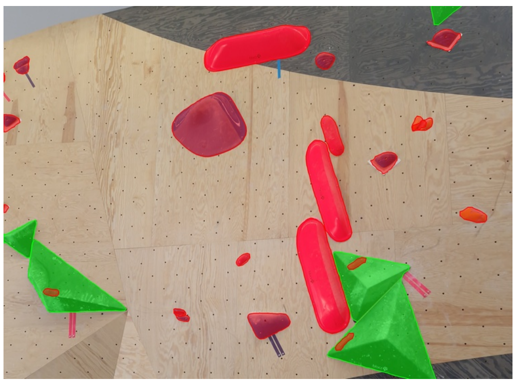


Figure 2.10: Common image detector network architecture.



(a) Image detection.



(b) Image segmentation.

Figure 2.11: Image detection (a) and segmentation (b) on a sample image from the `crimp-1` dataset.

Since the goal of image detection and image segmentation algorithms is to denote objects of interest, we would like to determine the network quality in terms of matches between predicted objects and their ground truth annotations. The predictions likely won't be a perfect match, so we can instead calculate intersection over union and count the match if the IoU value exceeds a threshold.

More formally, given two object masks I_1, I_2, we define their *intersection over union* (IoU) as

$$\mathrm{IoU}(I_1, I_2) = \frac{I_1 \cap I_2}{I_1 \cup I_2} \tag{2.29}$$

Given a fixed IoU threshold and a set of predictions, we match them with the ground truth predictions to obtain a set of true positives (TP), false positives (FP) and false negatives (FN). We then define their *precision* and *recall* as

$$\mathrm{precision} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}} \qquad \mathrm{recall} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}} \tag{2.30}$$

Both precision and recall are important metrics for measuring the quality of a network. *Precision* determines how precise the network is, in terms of only making detections that are correct – high values might not correspond to a well-trained network, as it only measures the fraction of correct guesses, ignoring the number of objects it missed. Conversely, *recall* measures how many of the objects it detected, ignoring the mistakes/incorrect guesses it made.

To combine the two into a usable metric for measuring the network quality, we can iterate over all detections in descending order of confidence, plotting a precision-recall curve. With each added detection, the recall will monotonically increase as more objects are detected, while the accuracy will form a zig-zag pattern, increasing with correct matches and decreasing with incorrect ones, as seen in Figure 2.12. Smoothening the precision-recall curve and integrating yields the *average precision* (AP), which balances both precision and recall. Finally, averaging AP over all IoU thresholds yields the *mean average precision* (mAP), which is the metric that we use to evaluate both our image segmentation (Section 4.3) and mesh segmentation (Section 4.4) results.

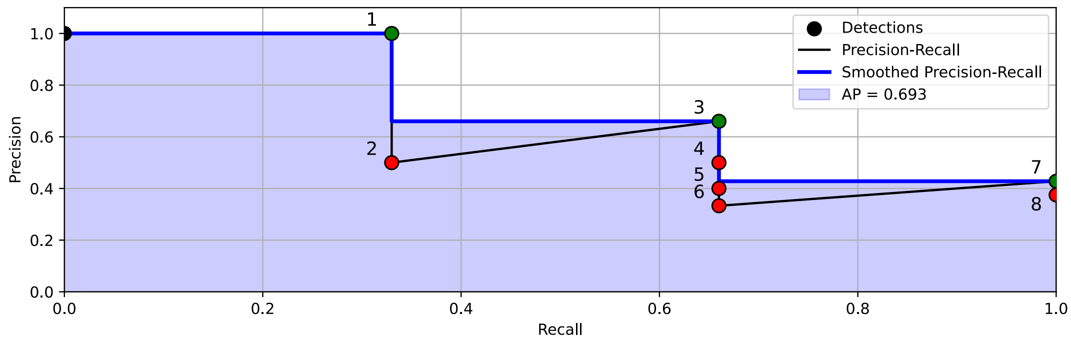


Figure 2.12: A precision-recall curve for a sample dataset of 3 objects and 8 detections. Green points indicate true positives, while red points indicate false positives.

2.4 Hyperparameter Optimization

In the previous section, we have covered the *backpropagation* algorithm, which used gradient descent to find a local minimum wrt. the loss of the network. This allowed us to optimize the parameters of the network such that the loss was minimized. But what if we wanted to also optimize the *hyperparameters* of the network, such as the size of the hidden layers of the network or the learning rate?

2.4.1 Bayesian optimization

Given a system with d real-valued hyperparameters, the task of hyperparameter optimization is equal to that of maximizing a suitable fitness function $f : \mathbb{R}^d \mapsto \mathbb{R}$. For a classification network, we could use the network's validation accuracy after training for 100 epochs.

To formulate the problem, we make the following general assumptions about f [34]:

- f is expensive to calculate, *The less evaluations the better.*
- f is continuous, but derivatives can't be calculated, *No derivatives available.*
- f's feasible region is bounded. *We constrain the hyperparameter values.*

The lack of derivatives means we can't use them to move towards the optimum. What we can do instead is to evaluate f at points which are most strategic for finding the maximum. *Bayesian optimization* uses the Bayesian rule to determine the best points to evaluate f; more specifically, it treats f as the *prior* and uses the evaluated points to calculate the *posterior*.

The algorithm is as follows:

1. pick a *surrogate function* f that models our fitness function
2. pick an *acquisition function* α that determines how advantageous evaluating each point is
3. evaluate $f(\text{argmax}\, \alpha)$, obtaining new data point x_{t+1}
4. update f based on x_{t+1}, returning to (3) until a stop condition is met

For f, *Gaussian Processes* are commonly used, since they are easy to work with and versatile due to the choice of the kernel [35]. For α, we can use the *Probability of Improvement*, which picks points that are most likely to improve on the current best [34], formulated as

$$x_{t+1} = \text{argmax}(\alpha_{PI}(x)) = \text{argmax}\big(P(f(x) \geq (f(x^+) + \varepsilon)\big) \tag{2.31}$$

with x^+ the current maximum and ε determining the ratio between exploration of more uncertain values, as opposed to maximizing in sections that were already explored.

An example of Bayesian optimization in action is captured in Figure 2.13. A one-dimensional fitness function is optimized over 5 iterations, showing the values of the fitness function, as well as those of the acquisition function.

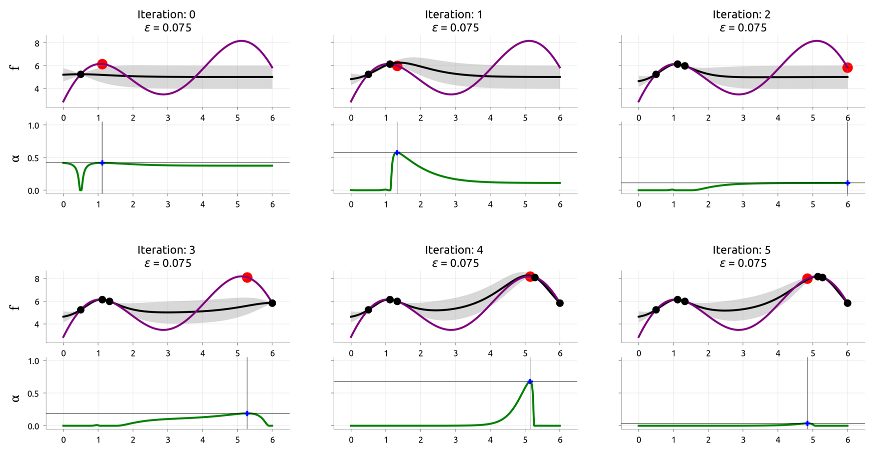


Figure 2.13: Bayesian optimization of a one-dimensional fitness function (Agnihotri and Batra [34]).

2.4.2 Successive Halving

While Bayesian optimization is an elegant solution to hyperparameter optimization under the assumptions we've made about f, problems such as network training can provide partial values of f (e.g. validation set accuracy) during training. Since Bayesian optimization requires a full evaluation of f, it can't easily determine whether a particular set of hyperparameters should be ignored early.

Successive Halving [36, 37], described as Algorithm 2, addresses this by only focusing on hyperparameters that seem promising, working as follows: given a certain budget B (computation time), we create n initial hyperparameter configurations, evaluate them, eliminate $1/\eta$ of the worst performers while increasing the budget for the remaining ones and repeat. Note that Algorithm 2 this is a simplified version as compared to [36, 37], omitting minimum resources and early stopping.

Algorithm 2 Successive Halving Algorithm

```
Require: Number of configurations $n$, total budget $B$, reduction factor $\eta$
1: Set $T$ = get_configurations(n)
2: Set $s = \lfloor \log_\eta(B) \rfloor$    // number of elimination rounds

3: for each halving $i$ from 1 to $s$ do
4:     $n_i = \left\lfloor \frac{n}{\eta^i} \right\rfloor$    // number of remaining configurations
5:     $r_i = \frac{B}{s \cdot n_i}$    // budget for each configuration

6:     $L = \text{evaluate}(T, r_i)$
7:     $T = \text{eliminate}(T, L, \eta)$
8: end for

9: return best-performing hyperparameter configuration
```

3 Method & Related Works

The goal, as outlined in the introduction, is to create a system which, given images from a commercial mobile phone, produces a segmented textured mesh of holds clustered into routes to be displayed online.

As this includes a number of subtasks (see Table 1.1), some of which can be solved using existing methods, the main contribution of this work and focus of this chapter is a new 3D segmentation algorithm, described in Sections 3.2 and 3.3. We show that this algorithm is conceptually simple, can be used on any 3D-reconstructed scene and outperforms state-of-the-art 3D segmentation algorithms on datasets with limited size (Section 4.4), such as in our case of indoor climbing gyms.

The remaining contents of this chapter are comparatively much shorter, and are included for the sake of completeness, as they are necessary in the image-to-segmented-mesh process. They include aligned 3D reconstruction (Section 3.1) and route detection via clustering (Section 3.4).

3.1 3D Reconstruction

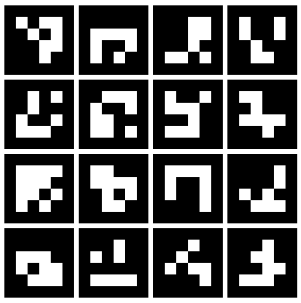
The use of COLMAP for the purposes of 3D reconstruction, given a set of images, has been covered extensively in Section 2.2. However, to obtain models that can be displayed online, we have to solve two additional challenges.

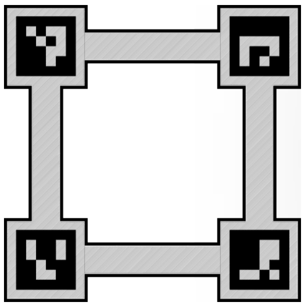
First, the reconstruction has to be rigidly transformed (rotated, translated and scaled) to real-world scale and orientation, in order to achieve sensible camera controls when viewing the reconstruction. Since this can't generally be done automatically given images with commonly available metadata, we opt for a marker-based solution to infer the rigid transformation of the scene.

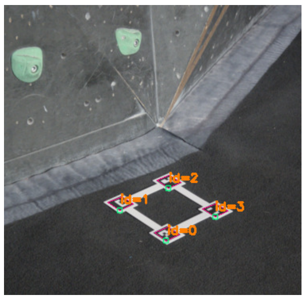
Second, subsequent updates to the model, caused by changes in the routes, have to be aligned to the original reconstruction in order for the new routes to be in the correct place wrt. the model. We achieve this by registering the images into the reconstruction and then performing constrained bundle adjustment to keep the original cameras in place.

3.1.1 Rigid Transformation Inference via ArUco Markers

To transform the reconstruction, we use a 3D-printed measure (Figure 3.1b) consisting of 4 ArUco markers (Figure 3.1a) [38, 39]. Detecting markers is done using OpenCV's ArUco implementation [15], obtaining their respective rigid transformations (Figure 3.1c). The final transformation is the average of the respective marker transformations over all images.


(a) ArUco markers.


(b) Measure model.



(c) Marker pose estimation.

Figure 3.1: Measure model and its usage for pose estimation and rigid scene transform.

Since the terrain might be uneven when only using one measure, multiple measures can be used to make the orientation inference more precise. If this is the case, linear regression over the marker points is used to infer the overall rotation, since individual measures might be locally inaccurate. This is especially useful for 3D reconstruction of larger spaces, where a small local error can cause a large global one.

3.1.2 Aligning Reconstructions

The process of sparse 3D reconstruction, as described in Section 2.2, repeatedly registers images and then uses bundle adjustment to jointly adjust their respective parameters. In our case, however, we would like to align the 3D reconstruction of the new routes to the reference one, so that the 3D models align when combined after mesh segmentation.

A naïve approach is to perform a separate reconstruction from the new images and then rigidly transform it to match the reference, either by iterative methods such as RANSAC, or by registering images from the new reconstruction into the reference and transforming according to the average difference in camera transforms. While it might seem that the latter will be prone to errors due to the underlying model changing (old routes are replaced by new routes), the registration algorithm is robust enough to find a set of valid matches.

While this works for smaller reconstructions and is directly supported by COLMAP, the local errors in larger reconstructions lead to significant global errors over longer distances and thus make it unusable for our use case. Instead, we take a different approach: register the new images into the existing reference reconstruction, removing the prior registration images after the process is complete. Although this might seem obvious, simply registering the images without bundle adjustment is inaccurate, as the new images are not jointly optimized. Running bundle adjustment after will, however, shift the new reconstruction relative to the original and thus can't be used.

To address this, we can use *constrained bundle adjustment* instead, which fixes a desired set of parameters – in our case of cameras from the original reconstruction – and only optimize the unconstrained parameters (i.e. the new images). This ensures that the newly registered images are jointly optimized, and that their transformation matches that of the reference reconstruction.

3.2 Image Detection & Segmentation

3.2.1 Related Work

In the past decade, the field of image detection & segmentation has been dominated by CNNs. While their use for classification and regression problems is rather intuitive, it is not obvious how to adopt their architecture for image detection, mainly due to the localization of objects.

Early attempts, such as OverFeat [40], use a multi-scale sliding window approach and subsequent merging of the predictions. This approach was shortly surpassed by R-CNN [41–43], which use a separate algorithm for region proposals, a CNN paired with per-class SVMs for classification and non-maximum suppression for final predictions.

Despite the success of the R-CNN architecture, the reliance on a proposal algorithm does not allow for end-to-end training and thus requires individual tuning of its components. This was addressed by YOLO [30–33], which incorporates bounding box predictions into the CNN network architecture and thus allow for faster and more accurate bounding box predictions.

Since the release of the "Attention Is All You Need" paper [44], transformer-based approaches [45–47] have achieved impressive results in many computer vision tasks, setting record scores in COCO-related benchmarks [48].

The following section describes how to obtain high-quality image segmentation datasets from 3D reconstructions in an efficient way, as well as a number of techniques for creating segmentation networks using YOLO [30–33], RT-DETR [49] and SAM [50] architectures.

3.2.2 Obtaining Datasets

The performance of every neural network is dependent on the data it is trained on. Although obtaining a quality image dataset given arbitrary images is usually time-consuming, we are working with a special category of images that are used as inputs for 3D reconstruction. Using the images to perform 3D reconstruction creates constraints between the images and the 3D model via camera projection, as described in Section 2.2.1 – an instance of an object in the 3D space projects back to its 2D segmentation with the image space, and vice versa. This means that, under ideal conditions, an image segmentation dataset can be created from a mesh segmentation using camera projection, and it is therefore sufficient to manually annotate the less time-intensive of the two.

The quality of the image segmentation dataset created by doing this is dependent on a number of factors, with the two main ones being the accuracy of camera positions and the quality of the reconstructed 3D model. Both of these introduce different kinds of errors – an inaccuracy in camera position shifts the segmentations within the image, while an error in the 3D model causes the shapes of the segmentation masks to be incorrect. Using SAM [50], with the existing segmentation masks as input prompts, improves the coarse segmentation mask, as seen in Figure 3.2.

In order to test the viability of this approach to creating image segmentations, 5 datasets were created from 2 climbing gyms. Images were annotated using LabelMe [51] with the help of SAM,

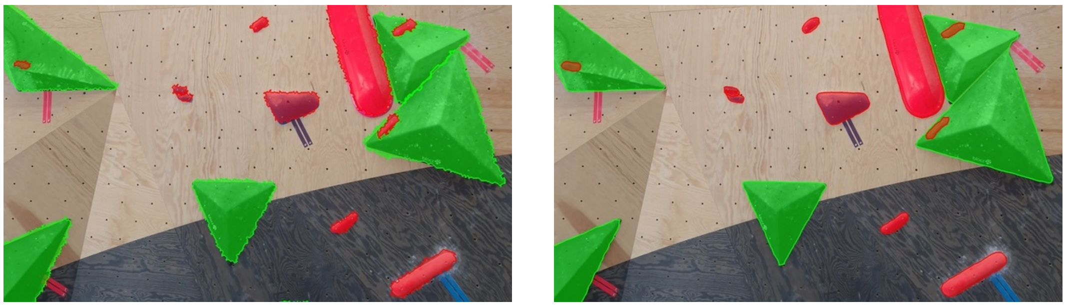

(a) Projected segmentation without SAM. (b) Projected segmentation with SAM.

Figure 3.2: Part of an image from the `crimp-1` dataset without SAM (a) and with SAM (b).

and the mesh was annotated using Blender [52]. The precision of the projected segmentations with and without SAM was compared against the annotated ground truth segmentations, with the results described in Section 4.3.

3.2.3 Segmentation Approaches

Image segmentation, as described in Section 2.3.3, is very similar to image detection, generally only differing in the ending parts of the model architecture. The recent introduction of SAM [50], which can achieve high-quality segmentation masks from various input prompts, such as those produced by a image detector, poses a question – can a detection network paired with SAM outperform a segmentation network?

As SAM accepts bounding boxes and points as inputs, we can directly use the detection results from a detection network to create a segmentation mask. While the performance might be inferior to a trained segmentation network for more complex objects and a specific set of classes, the small amount of training data available from our datasets could be insufficient to properly train a segmentation head.

Combining YOLO [30–33] and RT-DETR [49] with SAM [50], we obtain the following models:

- `yolov11-seg` – YOLOv11 segmentation model.
- `yolov11-sam` – YOLOv11 detection model with SAM segmentations.
- `detr-sam` – RT-DETER detection model with SAM segmentations.

Furthermore, we test the following model variants:

- `*-pt` – Pre-train the model on a larger dataset, such as COCO [53].
- `*-aug` – Use image augmentations to improve model generalization.
- `*-hyp` – Use hyperparameter tuning.

The performance on the `crimp` and `boulderhaus` datasets is described in Section 4.3.

3.3 Mesh Segmentation

3D segmentation is commonly understood to be the task of assigning semantic information to the points of a 3D pointcloud based on their attributes such as position, color, local density, local shape, etc. [54, 55]. The semantic information is most commonly either the class the point belongs to, referred to as *3D semantic segmentation*, or which instance of an object the point belongs to, referred to as *3D instance segmentation*[1]. Furthermore, we can distinguish between *closed-vocabulary* approaches, which define a finite set of classes to classify the points into, and an *open-vocabulary* approaches, which learn features that can be queried using open-vocabulary descriptions such as object texture, color, type or size.

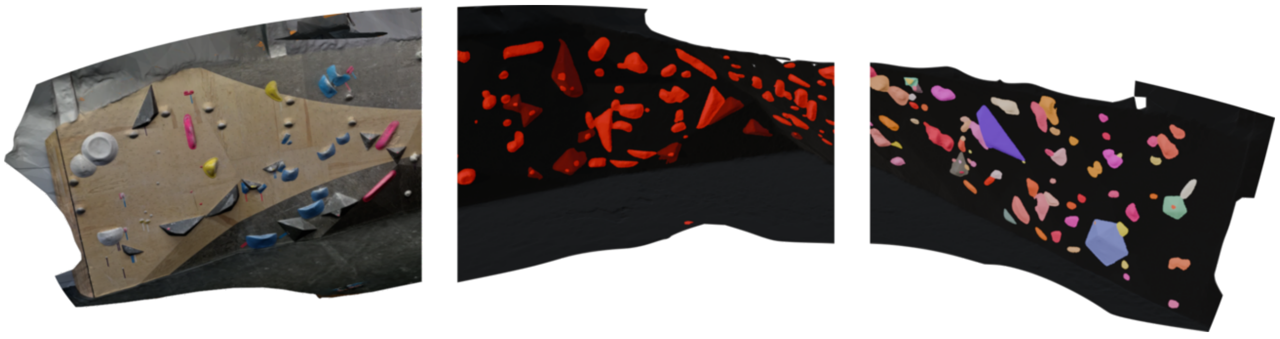


Figure 3.3: Example of a textured mesh (left), along with its 3D semantic segmentation (middle) and instance segmentation (right). The colors are chosen randomly per-instance and per-class.

3.3.1 Related Work

There have been many recent advancements in the field of closed-vocabulary 3D segmentation [54–65] using a variety of learning-based methods, with one thing in common: the learning is done on the level of the pointcloud and thus requires training data in the form of (at least) segmented pointclouds. This data is generally available for more common objects and indoor scenes [66, 67], but it is not easy to obtain for more specialized tasks, such as the subject of this thesis.

The solution proposed in this thesis is to use readily available state-of-the-art image segmentation infrastructure to obtain datasets and segmentation models for a desired subset of classes, and then use 3D projections of the 2D segmentations to create the resulting 3D instance segmentation.

The algorithm shares a number of similarities with [68], which also performs 2D segmentation and subsequent projection onto a mesh for the purpose of recognizing building rooftops. It differs in a number of ways: first, it uses RGB-D input for the segmentation network, while we use RGB. Since using RGB-D would preclude us from using pre-trained checkpoints (Section 4.3) and our data is limited, this is not a viable choice. Our algorithm, however, makes use of distance data during merging. Second, it merges projections by selecting strong candidates and then merging

[1] It is important to note that instance segmentation is not a special case of semantic segmentation – a instance-segmented mesh only carries information about distinct objects, not which classes the objects belong to.

iteratively based on overlap, whereas we use a graph clustering method on a graph made out of projections and their overlaps, allowing for a global approach instead of a local one.

Another similar work is SAM3D [69], which uses SAM [50] for image segmentation and then iteratively merges segmented depth-maps using *bi-directional merging* of the segmented point-clouds. The concept of projection is shared with our algorithm, but again differs in the merging approach – instead of local merges, we use graph clustering to make the approach global. Furthermore, they use an over-segmentation of the scene via distance-based approaches [70, 71], since the SAM-only segmentation generates a large amount of noise.

Unfortunately, as the source code for [68] is not openly available and [69] only performs instance segmentation without semantic segmentation, we will instead compare our algorithm with open-source state-of-the-art 3D segmentation methods: OneFormer3D [65] and PointTransformer [64]. The results on the obtained datasets (3.2.2) can be found in Section 4.4.

These two methods were selected since they are the top-performing methods for 3D instance and semantic segmentation on the S3DIS indoor segmentation dataset [72], which should make them suitable for segmenting 3D models of indoor climbing gyms.

OneFormer3D [65] is a transformer-based approach, which unifies the semantic and instance segmentation tasks into one, as opposed to standard practice of using different models for both tasks respectively. It starts with a U-net, which uses sparse 3D convolutions to create features for each point, creating superpoint features by either using voxel pooling or more computationally intensive point-wise feature pooling. The superpoints are then fed into the transformer decoder layers, obtaining trained instance kernels. Convolving the kernels with the superpoints produce instance and semantic masks.

PointTransformer [64] approaches the problem by decreasing the dimensionality of the problem, referred to as *serialization*, from 3D to 1D using space-filling curves. Sorting points based on the curve preserves locality, as nearby points on the curves correspond to nearby points in the original space, and can thus benefit from existing solutions to 1D problems. The transformer attention mechanism then works on patches of the ordered points, with successive attention layers using different space-filling curves for improved generalization. Similar to OneFormer3D, it uses a U-net architecture for pooling, and alternating encoder/decoder transformer layers.

Both OneFormer3D and PointTransformer use augmentations to enhance the model generalization. They are similar to image augmentations used for training our image segmentation networks as described in Section 4.3, and include rotation, scaling, flipping, position, color jitter and elastic distortions.

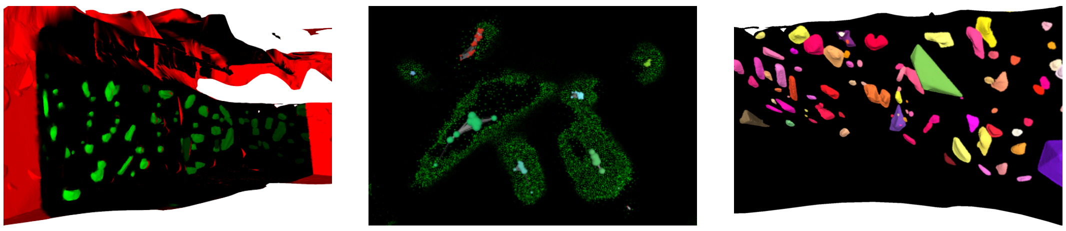
3.3.2 Algorithm

The general idea is to use a segmentation network to predict the objects in the images, project them on the mesh and merge them to create the object instances. This differs from the standard 3D instance segmentation approaches [54–63], which operate on the pointcloud level instead.

Our algorithm can be broadly split into the following parts

1. **projection** – 2D segmentations are projected onto the mesh, with their scores being the detection network scores multiplied by distance to the camera (Figure 3.4a)
2. **per-class clustering** – per-class graphs are constructed, with vertices being projections and edges their IoU multiplied by their scores, and clustered via Louvain (Figure 3.4b)
3. **merge & threshold** – clusters are merged, preserving triangles based on the ratio between positive and negative detections from all segmentations (Figure 3.4c)

The following sections cover each of these parts in turn. Since the algorithm contains a number of hyperparameters, Bayesian optimization [29] has been used to optimize them for their respective datasets, with the results, as well as comparisons against the aforementioned transformer-based 3D segmentation approaches, discussed in Section 4.4.


(a) **projection** (b) **per-class clustering** (c) **merge & threshold**

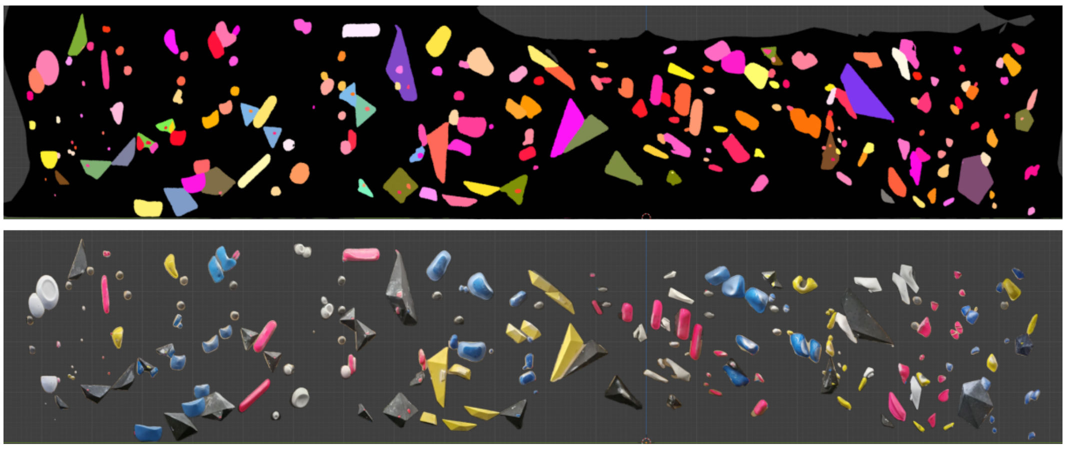

(d) **resulting segmentation**

Figure 3.4: Step-by-step visualization of the 3D instance segmentation algorithm.

Projection

The projection algorithm (Algorithm 3) uses the camera model developed in Section 2.2.1 to project the segmentations onto the mesh. It does this by first creating segmentation masks, ray-casting the mesh triangles (with some prior filtering for optimization) onto them and grouping all triangles from to the same projected segmentation.

Algorithm 3 Projection

```
Require: 2D segmentation $\mathcal{S}_i$, camera parameters $\mathcal{C}_i$, triangle data $\mathcal{T}$
 1: // Step 1: create a segmentation mask
 2: $\mathcal{M}_i = \text{mask\_from\_segmentation}(\mathcal{S}_i, \texttt{MASK\_RESOLUTION})$

 3: // Step 2: Filtering of distant/unseen triangles
 4: Filter $\mathbf{t} \in \mathcal{T}$ where $\text{dist}(\mathbf{t}, \mathcal{C}.\text{center}) > \texttt{VISIBILITY\_DISTANCE}$   // too far
 5: Filter $\mathbf{t} \in \mathcal{T}$ where $\text{angle}(\mathbf{t} - \mathcal{C}.\text{center}, \mathcal{C}.\text{direction}) > 90°$   // not seen by the camera

 6: // Step 3: Raycast from image to mesh triangles
 7: for each remaining triangle $\mathbf{t} \in \mathcal{T}$ do
 8:     Raycast from $\mathcal{C}$ to $\mathbf{t}$, obtaining $\mathbf{t}_{\text{hit}}$ and normalized image coordinates $\mathbf{p} = [x_{\text{im}}, y_{\text{im}}]^T$
 9:     $d_{\text{norm}} = \texttt{VISIBILITY\_EASING\_FUNCTION}(\text{distance}(t, \mathcal{C}.\text{center}) \ / \ \texttt{VISIBILITY\_DISTANCE})$

10:     if $\mathbf{t} == \mathbf{t}_{\text{hit}}$ then
11:         $\mathbf{t}_{\text{hit}}.\text{score} = \mathcal{M}_i[p].\text{score} * d_{\text{norm}}$
12:         $\mathbf{t}_{\text{hit}}.\text{class} = \mathcal{M}_i[p].\text{class}$
13:         $\mathbf{t}_{\text{hit}}.\text{object\_id} = \mathcal{M}[p].\text{object\_id}$
14:     else
15:         $\mathbf{t}_{\text{hit}}.\text{distance} = d_{\text{norm}}$   // incorrect hit – record distance (for thresholding)
16:     end if
17: end for

18: // Step 4: Create triangle groups based on projection object ID
19: Initialize empty list of projections $\mathcal{P}_i$
20: for each unique $\text{object\_id} \in \mathcal{T}$ do
21:     Initialize new mesh projection $\mathbf{p}$

22:     $\mathbf{p}.\text{triangles} = \{\mathbf{t} \mid \mathbf{t} \in \mathcal{T} \wedge \mathbf{t}.\text{object\_id} == \text{object\_id}\}$
23:     $\mathbf{p}.\text{class} = \mathbf{p}.\text{triangles}[0].\text{class}$   // they're all the same
24:     $\mathbf{p}.\text{score} = \text{average}(\mathbf{p}.\text{triangles.score})$
25:     $\mathbf{p}.\text{triangle\_scores} = \mathbf{p}.\text{triangles.score}$

26:     $\mathcal{P}_i.\text{append}(\mathbf{p})$
27: end for

28: return projections $\mathcal{P}_i$, non-object triangle distances $\mathcal{D}_i := \{\mathbf{t}.\text{distance} \mid \mathbf{t} \in \mathcal{T}\}$
```

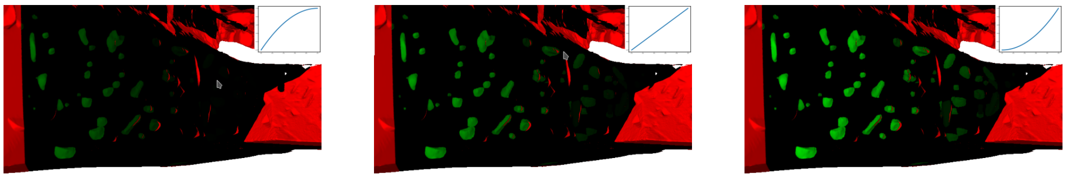

(a) Quadratic Ease Out (b) Linear (c) Quadratic Ease In

Figure 3.5: Influence of different easing functions on the same camera projection. Green areas are positive detections, black are negative detections and red parts are not seen by the camera.

To account for the fact that more distant objects tend to be recognized less precisely / incorrectly, a visibility distance multiplier is applied to the segmentation object scores (line 11). Since an object is seen from multiple views and distances in a 3D reconstruction, this increases the impact of better views on the 3D segmentation.

The multiplier is calculated using two hyperparameters (line 9): `VISIBILITY_DISTANCE`, which determines how far each camera sees, and `VISIBILITY_EASING_FUNCTION`, which is applied on the normalized distances for further control over how objects are scored. The influence of the easing functions on a sample reconstruction can be seen in Figure 3.5

Creating segmentation mask (line 2) is necessary for quickly checking the results of raycasting (line 8), since 2D segmentation objects are usually stored as polygons, making checking for many intersections very slow. The `MASK_RESOLUTION` parameter determines the mask resolution, trading projection speed and memory for accuracy.

The function notably doesn't just return groups of triangles of the i-th image $\mathcal{P}_i$, but also its non-object distances $\mathcal{D}_i$. These are distances to triangles which the camera saw but didn't contain an object. This is useful information for thresholding – if a triangle was a part of an object in a single camera but not in many others, there is a high chance that it was a mistake.

Per-class clustering

With the per-class triangles identified, we group projections of the same object based on the projection overlaps using Louvain clustering [7] (Algorithm 4). To do this, we build a graph for each class (line 4), with vertices being projected segmentations and edges being triangle IoUs between them, multiplied by the respective object scores (lines 10, 11). The idea behind this is that strong detections with high overlap seen from a closer distance should cluster more strongly than weaker ones with smaller overlap from afar.

Since the number of graph edges increases quadratically with the number of detected projections and weight calculation is not cheap, building the graph naïvely is infeasible. Fortunately, since the vertices correspond to objects in 3D space, we can use spatial hashing (line 5) to skip checks for objects that can't intersect and thus have edge weight 0, which provides a significant speedup over the naïve approach. The size of the spatial hashing cells is determined by `CELL_RESOLUTION` and can be calculated from the sizes of the projections, e.g. a multiple of the average.

Algorithm 4 Per-class Clustering

```
Require: Filtered projections $\mathcal{P}$
 1: Initialize empty clusters of projections $\mathcal{C}$

 2: for each class $c$ do
 3:   // Step 1: initialize graph with vertices and no edges, along with the spatial hash
 4:   $\mathcal{G} = (\{\mathbf{p} \mid \mathbf{p} \in \mathcal{P} \wedge \mathbf{p}.\text{class} == c\}, \{\})$
 5:   $\mathcal{SH} = \text{build\_spatial\_hash}(\mathcal{G}.\text{vertices}, \texttt{CELL\_RESOLUTION})$

 6:   // Step 2: create edges using the spatial hash only where necessary
 7:   for each projection $\mathbf{p} \in \mathcal{P}$ do
 8:     for each neighbor $\mathbf{q} \in \mathcal{SH}.\text{neighbours}(\mathbf{p})$ do
 9:       // Compute edge weight
10:       $\text{iou} = \frac{\text{area}(\mathbf{p}.\text{triangles} \cap \mathbf{q}.\text{triangles})}{\text{area}(\mathbf{p}.\text{triangles} \cup \mathbf{q}.\text{triangles})}$
11:       $\text{weight} = \text{iou} * \mathbf{p}.\text{score} * \mathbf{q}.\text{score}$

12:       if $\text{weight} > 0$ then
13:         $\mathcal{G}.\text{add\_edge}(\mathbf{p}, \mathbf{q}, \text{weight})$
14:       end if
15:     end for
16:   end for

17:   // Step 3: perform Louvain clustering
18:   $\mathcal{C}.\text{extend}(\text{louvain\_clustering}(\mathcal{G}, \alpha = \texttt{DETECTION\_RESOLUTION}))$
19: end for

20: return clusters of projections $\mathcal{C}$
```

The α parameter for the Louvain algorithm controls the size of detected clusters, with larger values corresponding to larger clusters, implemented as an extension to the original Louvain clustering algorithm [73]. Its effects on the clustering can be seen in figure Figure 3.6 – smaller values lead to undersegmentation (Figure 3.6a), while higher lead to oversegmentation (Figure 3.6c).

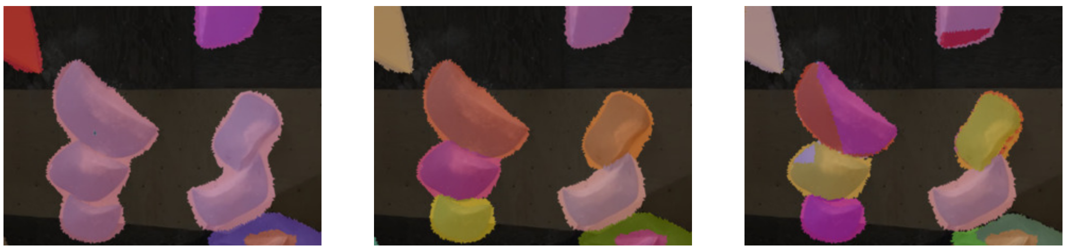

(a) $\alpha = 0.01$ (b) $\alpha = 1.00$ (c) $\alpha = 100.00$

Figure 3.6: Influence of different Louvain resolutions on the segmentation.

Merge & threshold

With clusterings available, the final step of our algorithm is to merge the projections to obtain the resulting object masks (Algorithm 7). To ensure the object boundaries are correct, we consider two values when deciding which triangles form an object. The *positive triangle score* (Algorithm 5) is the sum of the scores of its projections, since the more projections it appears in, the more likely it is to belong to the object. Conversely, the *negative triangle score* (Algorithm 6) is the sum of its non-object distances (i.e. distances to cameras where the triangle wasn't a part of any object), plus the scores of same-class projections from other clusters.

The balance between positive and negative scores ensure smooth object borders – triangles further away from the object centers decrease in positive score and increase in the negative, as they are seen in less projections, or in projections belonging to other clusters. The total score is then just the portion of positive score over total score, with higher values corresponding to a higher likelihood that a triangle is part of the segmented object, determined by a `THRESHOLD` parameter. The influence of the `THRESHOLD` parameter on the resulting segmentation can be seen in Figure 3.7 – smaller values permit more triangles and might over-estimate the object boundaries, while larger values do the opposite.

Algorithm 5 `compute_positive_score`$(\mathbf{t}, \mathbf{c})$ [subroutine of Algorithm 7]

```
1: pos = 0

2: for each projection $\mathbf{p} \in \mathbf{c}$ do
3:     if $\mathbf{t} \in \mathbf{p}$.triangles then
4:         pos += $\mathbf{p}$.score
5:     end if
6: end for

7: return pos
```

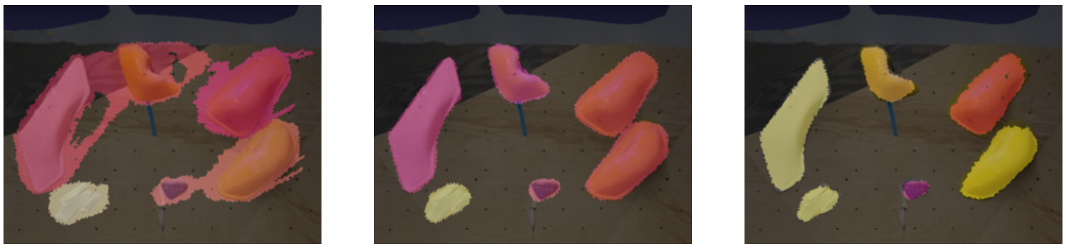
Algorithm 6 `compute_negative_score`$(\mathbf{t}, \mathbf{c}, \mathcal{D}, \mathcal{C})$ [subroutine of Algorithm 7]

```
1: neg = 0

2: for each non-object camera distance $\mathcal{D}_i \in \mathcal{D}$ do
3:     neg += $\mathcal{D}_i[\mathbf{t}]$
4: end for

5: for each other cluster $\mathbf{c}' \in \mathcal{C}, \mathbf{c}' \neq \mathbf{c}$ and its projections $\mathbf{p}' \in \mathbf{c}'$ do
6:     if $\mathbf{c}'$.class == $\mathbf{c}$.class $\wedge\ \mathbf{t} \in \mathbf{p}'$.triangles then
7:         neg += $\mathbf{p}$.triangle_scores$[\mathbf{t}]$
8:     end if
9: end for

10: return neg
```



(a) THRESHOLD $= 0.05$ (b) THRESHOLD $= 0.35$ (c) THRESHOLD $= 0.85$

Figure 3.7: Influence of different THRESHOLD values on the resulting segmentation.

Algorithm 7 Merge & threshold

```
Require: Clustered projections $\mathcal{C}$, non-object distances $\mathcal{D} = \{\mathcal{D}_1, \dots, \mathcal{D}_i\}$, triangle data $\mathcal{T}$
 1: Initialize empty list of merged object instances $\mathcal{O}$
 2: for each cluster $\mathbf{c} \in \mathcal{C}$ do
 3:     cluster_triangles $= \bigcup_{\mathbf{p} \in \mathbf{c}} \mathbf{p}$.triangles

 4:     for each triangle $\mathbf{t} \in$ cluster_triangles do
 5:         // Positive score: sum of scores of projections the triangle is a part of
 6:         $\mathbf{t}$.pos = compute_positive_score($\mathbf{t}, \mathbf{c}$)

 7:         // Negative score: non-object distances + scores of projections from different clusters
 8:         $\mathbf{t}$.neg = compute_negative_score($\mathbf{t}, \mathbf{c}, \mathcal{D}, \mathcal{C}$)

 9:         // Total score is just the fraction of the positive over total score
10:         $\mathbf{t}$.score $= \frac{\mathbf{t}.\text{pos}}{\mathbf{t}.\text{pos} + \mathbf{t}.\text{neg}}$

11:         if $\mathbf{t}$.score < THRESHOLD then
12:             Filter $\mathbf{t}$ from cluster_triangles
13:         end if
14:     end for

15:     if |cluster_triangles| > 0 then
16:         Initialize new merged instance $\mathbf{o}$

17:         $\mathbf{o}$.triangles = cluster_triangles
18:         $\mathbf{o}$.class = $\mathbf{o}$.triangles[0].class    // they're all the same

19:         $\mathcal{O}$.append($\mathbf{o}$)
20:     end if
21: end for

22: return object instances $\mathcal{O}$
```

3.3.3 Modifications for Unscaled Reconstructions

As described, the algorithm assumes an appropriately scaled reconstruction, such that the `VISIBILITY_DISTANCE` and `CELL_RESOLUTION` parameters can be based off of real-world length measurements. Although this is not an issue for our use case of hold segmentation as all models are rigidly transformed to real-world coordinates (Section 3.1.1), this makes the algorithm unusable for general 3D reconstructions.

The solution is to replace the parameters with their normalized variants, such that there is no scale dependence and that they can be tuned based on the particular task. We opt for the following transformation:

$$\texttt{PARAMETER} = \tan\left(\frac{\pi}{2} \cdot \texttt{NORMALIZED\_PARAMETER}\right) \cdot d_{\text{avg}},$$

where `NORMALIZED_PARAMETER` $\in [0, 1)$, and d_{avg} being the average over all camera-to-object distances. Values in the range $[0, 0.5]$ behave linearly, obtaining the average distance at 0.5, and going to infinity as we approach 1.

To accommodate for this in our algorithm, we first perform raycast to all object centers before running Algorithm 3. As this is not a costly operation, we can perform multiple randomly sampled raycasts to ensure that the calculated distance is representative of the given projection.

3.4 Instance Clustering

The final step, after obtaining the segmented mesh, is to group the holds into their respective routes. Since this is a relatively niche and not particularly difficult problem, we omit related works section and simply formulate our solution, for the sake of completeness.

Routes are made up of individual holds of the same color that are closely packed together. Since a particular gym has only a limited selection of holds, the set of colors is fixed for each gym, both technically and practically, as people need to be able to distinguish the colors of the individual routes they are climbing. This means that we can simply train a classifier to detect colors/types of holds, and use graph clustering (similar to the previous section) to form the individual routes.

As a byproduct of this approach, we can provide statistics about the number of holds of particular color/type currently on the wall, which is useful information for the setters in order to track holds on/off the wall. This also allows us to filter out holds that shouldn't be clustered – for example, one type of climbing holds is commonly reserved as "downclimb holds", which serve the purpose of safely climbing down after an ascent of a boulder and should be ignored.

First, we train an image classifier on the individual segmentations of the input images (Figure 3.8, Algorithm 8) using the YOLO [30–33] architecture. We use the masked cutouts of the holds created via the segmentations, helping the network to focus on the relevant part of the image.

Since we have a set of predictions for the same 3D object, we sum the per-class confidences, weighing them by the number of pixels in the classified image (line 6), as smaller segmentations are generally less precise, and set the object's class to the one with the highest score (line 8).

Next, per-class graphs are constructed, with vertices being the segmented 3D holds and edges their geodesic distances within the mesh, calculated using the FlipOut algorithm [74] (Algorithm 9). Using geodesic distance is crucial, since two holds could be close together using metrics such as Euclidean distance, but be on the opposite side of a wall, resulting in impossible routes. The z

Algorithm 8 Object Type Classification

```
Require: Object instances $\mathcal{O} = \{O_1, \ldots\}$
 1: for each object instance $\mathbf{o} \in \mathcal{O}$ do
 2:     // Classify the object using its corresponding segmentations, weighted by their area
 3:     Initialize class confidences list $\mathcal{C}$
 4:     for each segmentation $\mathbf{s} \in \mathbf{o}$.segmentations do
 5:         $\text{area} = \mathbf{s}.\text{width} \times \mathbf{s}.\text{height}$
 6:         $\mathcal{C}.\text{append}(\text{area} * \texttt{class\_confidences}(\mathbf{s}))$
 7:     end for

 8:     $\mathbf{o}.\text{class} = \text{argmax } \mathcal{C}$
 9: end for

10: return class groups $\mathcal{G} = \{\{\mathbf{o}.\text{class} = c_i \mid \mathbf{o} \in \mathcal{O}\} \mid c_i \in \text{classes}\}$
```

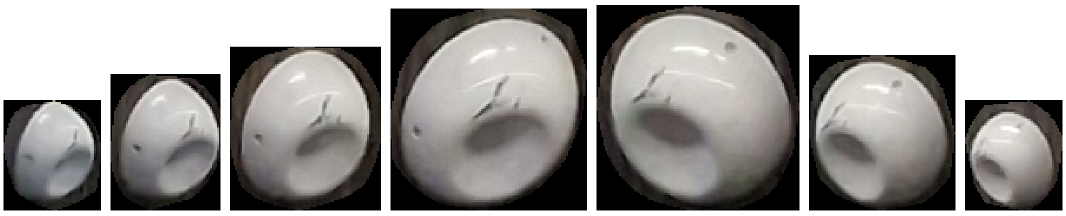


Figure 3.8: Segmentations belonging to one 3D segmented hold.

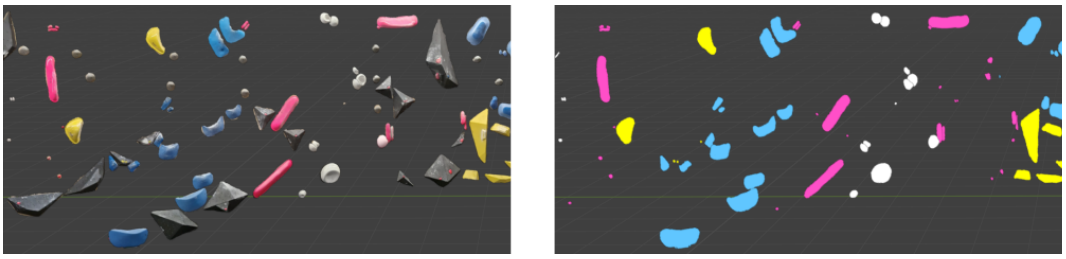


Figure 3.9: 3D hold instances (left) and their color assignments from the classifier (right). The missing holds in he right image have been classified as *volumes*, which belong to all routes, and have thus been filtered out from the route detection process.

(up) coordinate of the geodesic path is optionally multiplied by `FLATTEN_RATIO` $\in [0, 1]$ (line 5), with the idea being that since routes generally go up, flattening the model in the z coordinate will increase the accuracy.

Finally, we use Louvain clustering on each per-class graph to obtain the resulting routes. Similar to Section 3.3.2, we use an α parameter to adjust the size of clusters. Bayesian optimization [29] has been used to optimize the hyperparameters for their respective datasets, with the results discussed in Section 4.5.

Algorithm 9 Route Clustering

```
Require: Class groups $\mathcal{G}$
 1: Initialize empty list of routes $\mathcal{R}$

 2: for class group $G \in \mathcal{G}$ do
 3:     // Calculate geodesic paths between objects (possibly flattened)
 4:     paths = pairwise_geodesic_paths($G$, MIN_HOLD_DIST, MAX_HOLD_DIST)
 5:     paths.$z$ *= FLATTEN_RATIO

 6:     // Cluster group, with edge weights being geodesic path distances
 7:     graph = create_graph(vertices = $G$, edges = $\binom{G}{2}$, weights = paths)
 8:     $\mathcal{R}$.extend(louvain_clustering(graph, $\alpha$ = DETECTION_RESOLUTION))
 9: end for

10: return clustered routes $\mathcal{R}$
```

4 Experiments

To evaluate the performance of the algorithms described in Chapter 3, we first create datasets for 2 different climbing gyms (Section 4.2), comparing the use of semi-automatic annotation to the mesh annotation and projection approach. To ensure model generalization, each gym contains at least 2 separate datasets, the first being used for training and any additional one being used for testing/validation, respectively. We show that the projection approach is superior to semi-manual annotation, saving a significant amount of time in exchange for a minor loss in accuracy.

Next, we benchmark the proposed image segmentation approaches (Section 4.3), picking the best performing variant for our mesh segmentation algorithm. We conclude that, in our case, the detection + SAM-based segmentation approach is superior to a segmentation network.

Our mesh segmentation algorithm is evaluated on the obtained datasets and compared against two transformer-based state-of-the-art 3D segmentation algorithms (Section 4.4), outperforming them by a wide margin, as it is able to utilize additional information from the available 3D reconstruction. However, it is orders-of-magnitude slower, though this trade-off is acceptable for our use case, as the main bottleneck is 3D reconstruction.

4.1 Setup

The training and evaluation of all models has been performed on a single Nvidia 4090 (24 GB VRAM). Excluding training, which utilizes the full amount of available VRAM for an increase in batch size, all models used in our segmentation algorithm can fit within 10 GB of VRAM, making them usable on consumer-grade hardware. If training is also required, the model batch and input sizes can be decreased, resulting in accuracy losses (see non-hyp model variants in Table 4.3) and decreased inference speed due to limited batching.

The 3D segmentation algorithm is implemented in Python and uses a number of open-source libraries, with a heavy emphasis on NumPy [75] for general array operations (which most of the algorithm consists of), and Trimesh [76] + Open3D [77] for mesh operations. As mentioned, sparse and dense 3D reconstruction is handled by COLMAP [1, 2] and OpenMVS [13] respectively, using their Python bindings, and image segmentation is implemented as a wrapper over the Ultralytics Python package [78].

4.2 Obtaining Datasets

We have created 5 datasets across 2 different climbing gyms – Crimp (`crimp-*`) and Boulderhaus (`boulderhaus-*`). Crimp (Figure 4.1b) is a newly open climbing gym with a light wall and contains many modern hold sets, a lot of which are dual texture – a modern trend in climbing where a climbing hold contains both a glossy texture and a matte one. Boulderhaus (Figure 4.1a), on the other hand, is an old-school climbing gym with a dark wall and mostly contains older hold sets.

Images for all datasets were annotated using LabelMe [51] with the help of SAM, using point prompts as inputs and subsequent manual adjustments, which significantly reduced the time in annotating images, as opposed to an entirely manual process. The mesh was annotated using Blender [52], by painting objects' triangles with different colors depending on their class, obtaining both an instance and semantic segmentation based on the color values.

As seen in Table 4.1, using mesh annotation and projection to images, the time required to produce a segmentation dataset can be reduced by a factor of 13 while maintaining a high average mIoU of 88% to the ground truth. Furthermore, Using SAM with bounding box prompts leads to an average mIoU improvement of 5% and does not result in a decrease in accuracy.

Table 4.1: Comparison between datasets created using manual image annotation and mesh annotation with projection to images. [1]Image Time is an estimation of image annotation time, calculated by timing manual annotation on 15 randomly sampled images from each of the datasetes and extrapolating to the rest. Error assumes 95% confidence with standard deviation.

Dataset	**Size**	**Image**[1]	**Mesh**	**mIoU**	**mIoU** (SAM)
`crimp-1`	184	5h 57m ± 24.7%	31m	83.8%	92.7% (+7.9%)
`crimp-2`	115	4h 50m ± 32.0%	31m	84.9%	88.5% (+3.6%)
`crimp-3`	77	3h 13m ± 20.3%	29m	85.8%	89.1% (+3.3%)
`boulderhaus-1`	85	6h 7m ± 36.7%	26m	79.5%	84.5% (+5.0%)
`boulderhaus-2`	205	11h 38m ± 38.4%	31m	80.9%	85.0% (+4.1%)
`total`	666	30h 47m ± 41.3%	2h 28m	83.0%	88.0% (+5.0%)

(a) *Boulderhaus* climbing wall, Heidelberg.

(b) *Crimp* climbing wall, Heidelberg.

4.3 Image Detection & Segmentation

Next, we examine the performance on the datasets obtained in Section 3.2.2. The three types of architectures we analyze are YOLO-segmentation (`yolov11-seg`), YOLO-detection with SAM (`yolov11-sam`) and RT-DETR with SAM (`detr-sam`). For each, we test the benefits of using pre-trained weights (`-pt`, trained on COCO [53]), image augmentations (`-aug`) and hyperparameter optimization (`-hyp`).

The augmentations can be broadly separated into three categories [79] – *affine* transformations, which include affine operations such as translation, rotation and perspective, *color* transformations, which include all color operations such as changes in the HSV image space, and *selective*, which are more complex and include forming a mosaic and image cropping. Their values are captured in Table 4.2, with a sample batch shown as Figure 4.2.

Table 4.2: Default values for augmentation parameters, split by categories. The values are uniformly sampled from the specified value range / set of permitted values and are applied jointly.

	Name	**Values**
Color	`hsv` hue shift	$[0, 1.0]$
	`hsv` saturation shift	$[0, 0.7]$
	`hsv` volume shift	$[0, 0.4]$
Affine	rotation	$[0, 360°]$
	translation	$[0\%, 10\%]$
	scale	$[0, 0.5]$
	vertical flip	$\{\text{false}, \text{true}\}$
Sel.	mosaic (4 images)	$\{\text{true}\}$
	crop by fraction	$[0, 1.0]$

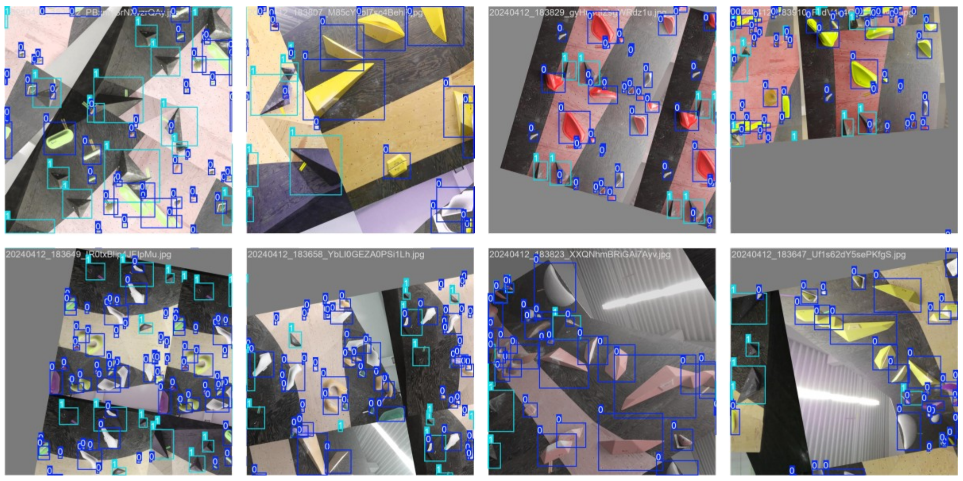


Figure 4.2: Sample image augmentations on `crimp-1` dataset images, using values from Table 4.2.

As for the hyperparameters, we are optimizing the *learning rate* (loguniform $\in [10^{-4}, 10^{-1}]$), *momentum* (uniform $\in [0.5, 0.99]$) and *batch size* ($\{2, 4, 8, 16, 32\}$) of the optimizer, the *image size* ($\{640, 960, 1024, 1280\}$) of the input model and the *model sizes* for both YOLO (`n`, `s`, `m`, `l`, `x`) and RT-DETR (`l`, `x`), respectively. The hyperparameter optimization is done via successive halving (Section 2.4.2), with the fitness function being mAP_{50}.

The results are captured in Table 4.3. To properly evaluate whether the models generalize correctly, they are trained on the first dataset for both walls respectively (`crimp-1` and `boulderhaus-1`), with the remaining datasets for each wall split evenly for validation and testing. As seen in Table 4.3, the model performance increases by using pre-trained weights, image augmentations described in Table 4.2 and hyperparameter optimization.

The top-performing model for both datasets is `yolov11-sam-pt-aug-hyp`, which will be the model used as the segmentation model in the following mesh segmentation section. Interestingly, there is a significant discrepancy in mAP between the purely segmentational `yolo-seg` models, and the `yolo/detr` models paired with SAM (23.9 for `crimp`, 12.3 for `boulderhaus`). Considering the limited amount of training image, the most likely explanation is that properly training a segmentation head requires more data to create stable segmentation masks.

The hyperparameter values of the top-performing models were consistently maximum possible *image size* (1280), paired with the largest possible *batch size* that fit in the memory (8), while other parameters didn't play a significant role.

Table 4.3: Performance of YOLO and RT-DETR image segmentation models on the respective datasets. Scores are calculated as an average of 5 trained models with the given parameters, with train/val/test splits being 0.8/0.1/0.1 for `crimp-1`, `boulderhaus-1`, and 0.0/0.5/0.5 for the remaining datasets for both walls respectively.

	Image Segmentation	`crimp`		`boulderhaus`	
		mAP_{50}	mAP	mAP_{50}	mAP
YOLO-seg	`yolov11-seg`	31.2	12.2	10.0	3.5
	`yolov11-seg-pt`	56.5	26.1	31.2	13.1
	`yolov11-seg-pt-aug`	63.9	30.3	33.3	14.1
	`yolov11-seg-pt-aug-hyp`	73.9	**45.1**	45.4	**23.5**
YOLO-SAM	`yolov11-sam`	39.6	32.0	14.7	11.3
	`yolov11-sam-pt`	67.2	59.9	39.7	31.9
	`yolov11-sam-pt-aug`	73.0	65.5	42.3	33.2
	`yolov11-sam-pt-aug-hyp`	77.7	$\boxed{69.0}$	51.3	$\boxed{39.4}$
RT-DETR	`detr-sam`	23.9	18.7	8.5	5.7
	`detr-sam-pt`	71.7	61.7	39.2	29.3
	`detr-sam-pt-aug`	70.4	60.4	40.0	30.8
	`detr-sam-pt-aug-hyp`	78.7	**68.9**	41.5	**32.2**

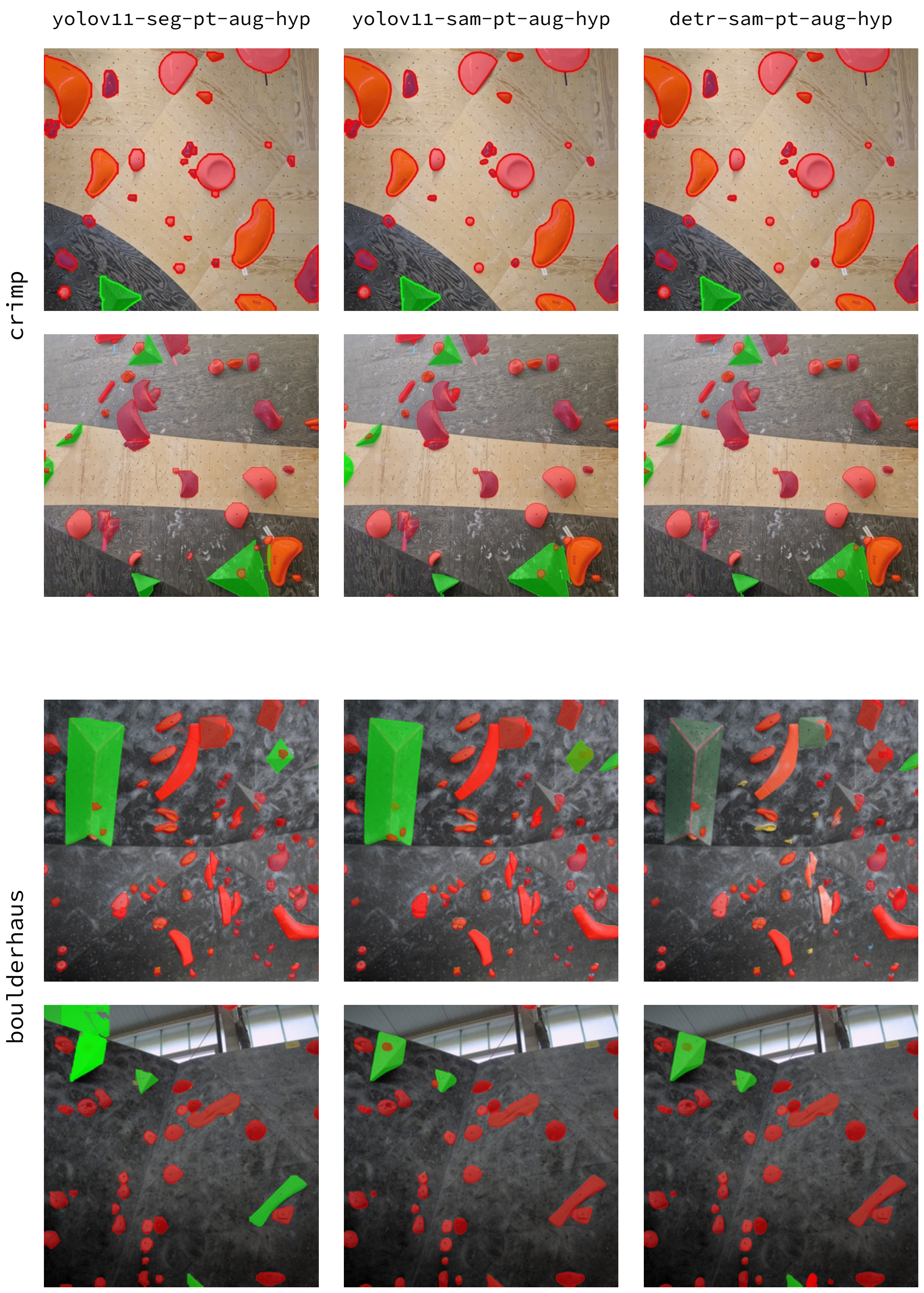


Figure 4.3: Image segmentations of the segmentation models on the `crimp` and `boulderhaus` datasets.

4.4 Mesh Segmentation

We examine the performance of our 3D segmentation algorithm against the state-of-the-art semantic and instance segmentation algorithms OneFormer3D [65] and PointTransformer [64]. To verify the validity of our setup, we first replicated the results of both methods on S3DIS [72], which is an indoor dataset with classes such as door, window, chair, sofa, etc., achieving similar results as those reported by the authors.

Next, we modified the S3DIS model configuration for both models to train on our indoor wall datasets, using the same pre-trained weights as S3DIS. The results for both semantic and instance segmentation are captured in Table 4.4. For our algorithm, we use the `yolov11-sam-pt-aug-hyp` segmentation models, as described in the previous section. The hyperparameter-optimized variant uses Bayesian optimization [80] on the following parameters: `VISIBILITY_DISTANCE` ($[2.0, 20.0]$), `VISIBILITY_EASING_FUNCTION_POWER` ($[-1.0, 1.0]$), `THRESHOLDING_CONSTANT` ($[0.0, 1.0]$) and `LOUVAIN_COMMUNITY_RESOLUTION` ($[0.0, 100.0]$).

Both methods perform poorly on our data for 3D instance and semantic segmentation, especially for the `boulderhaus` dataset. This can be explained by the significant difference in the amount of training data for our problem and S3DIS respectively, as S3DIS has 271 input pointclouds while we only have 1 for each dataset. We can get the methods to overfit when using all of the data for both training and validation, but couldn't find a way to make them generalize.

A major disadvantage of our algorithm is its speed, averaging 151s on the evaluated datasets, as opposed to <1s for both transformer-based approaches. This is due to the fact that the transformer-based approaches are efficiently implemented on the GPU, whereas our algorithm is mainly CPU-bound and not overly optimized. Fortunately, this is not a large issue for our use case, as the time is dominated by the duration of the 3D reconstruction.

Interestingly, the hyperparameter optimization slightly decreases the performance of semantic segmentation on the `crimp` dataset ($81.5 \rightarrow 81.4$) and doesn't yield much improvement for other metrics, suggesting that the hand-crafted hyperparameters were close to optimal. On the other hand, it leads to a decent improvement for the `boulderhaus` dataset for both task metrics ($23.5\,\text{mAP} \rightarrow 26.1\,\text{mAP}$, $62.7\,\text{mIoU} \rightarrow 71.8\,\text{mIoU}$).

Table 4.4: Combined metrics for 3D instance and semantic segmentation on `crimp` and `boulderhaus`.

	Instance				**Semantic**	
	`crimp`		`boulderhaus`		`crimp`	`boulderhaus`
	mAP$_{50}$	mAP	mAP$_{50}$	mAP	mIoU	mIoU
OneFormer3D	16.2	6.0	3.2	1.0	57.8	22.5
PointTransformer	19.7	7.3	3.2	1.4	78.1	41.8
Our method (`base`)	72.8	33.1	56.6	23.5	$\boxed{\textbf{81.5}}$	62.7
$\boxed{\textbf{Our method (hyp)}}$	75.9	$\boxed{\textbf{33.2}}$	66.0	$\boxed{\textbf{26.1}}$	81.4	$\boxed{\textbf{71.8}}$

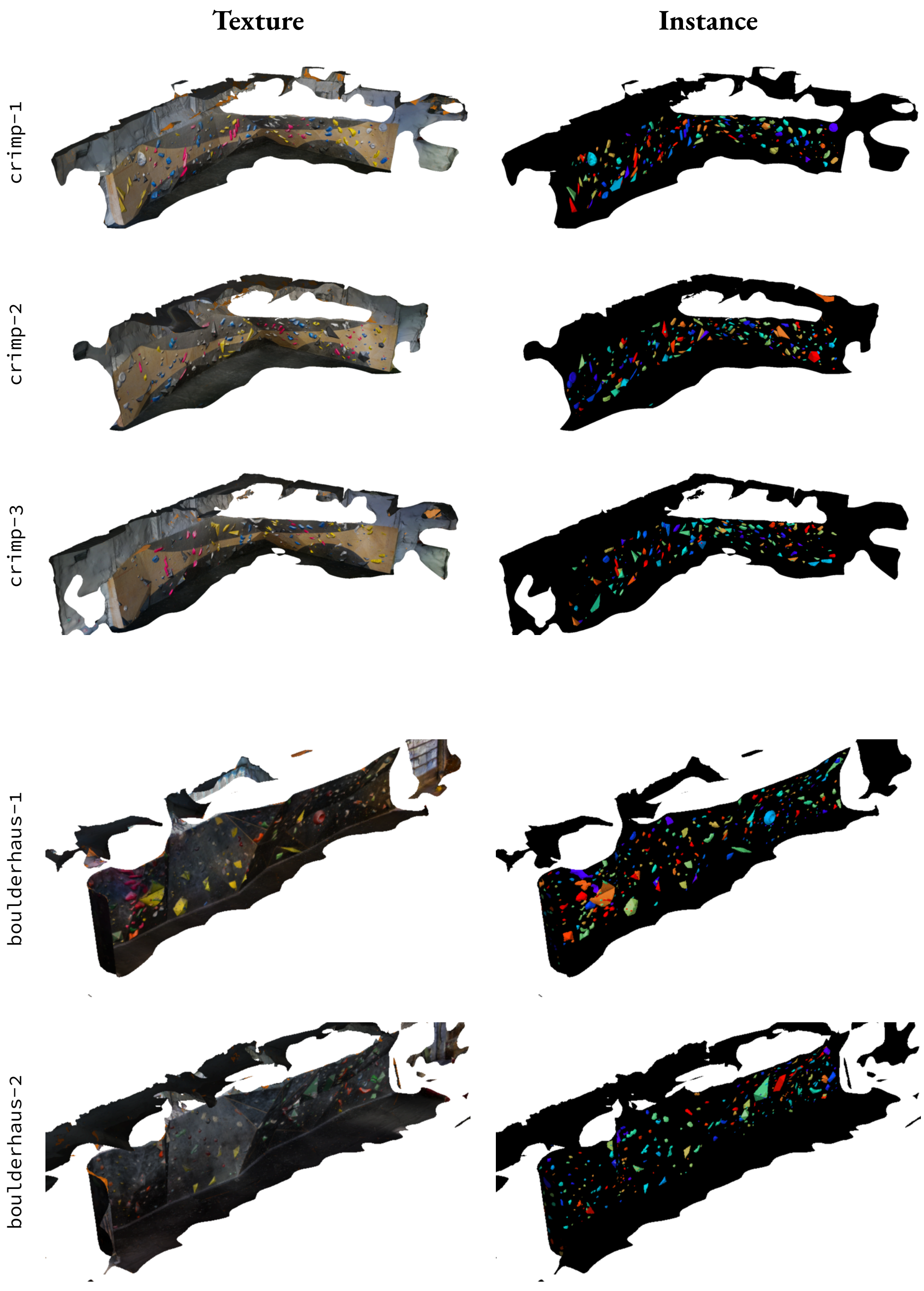


Figure 4.4: Visualization of instance and mesh segmentations produced by our algorithm.

4.5 Instance Clustering

With the 3D segmentations available, the remaining task is to cluster the 3D segmented object instances into individual routes. We again use Bayesian optimization as no partial results are available, and optimize for `FLATTEN_RATIO` ($[0.0, 1.0]$), `LOUVAIN_COMMUNITY_RESOLUTION` ($[0.0, 100.0]$), `MIN_HOLD_DIST` ($[0.0, 5.0]$) and `MAX_HOLD_DIST` ($[0.0, 10.0]$).

As seen in Table 4.5, the method performs well with default parameter values, achieving 0.93 ARI average over both datasets before hyperparameter optimization and 0.98 ARI after.

As for the values of the hyperparameters, the `FLATTEN_RATIO` played no role to the method performance, remaining arbitrary over all improvements in ARI. The community resolution and hold distance parameters deviated slightly from the default values for `boulderhaus`, stabilizing around `LOUVAIN_COMMUNITY_RESOLUTION` $= 0.76$, `MIN_HOLD_DIST` $= 1.1$ and `MAX_HOLD_DIST` $= 3.0$.

We have experimented with different hyperparameters for the classifier network, such as model size and input image size, as well as not using segmentation masks (as visualized in Figure 3.8) and have always achieved close to 100% accuracy for the hold type classifier (Algorithm 8). Although this might seem unrealistic, we are essentially training a classification network to recognize a limited set of known colors, which is a trivial task for a classifier, especially when averaging many classifications of the same hold of different sizes and angles.

Table 4.5: Results of our method compared to SOTA methods.

Route Clustering	`crimp` ARI	`boulderhaus` ARI
Our method (`base`)	0.978	0.883
$\boxed{\textbf{Our method (hyp)}}$	$\boxed{\mathbf{0.978}}$	$\boxed{\mathbf{0.982}}$

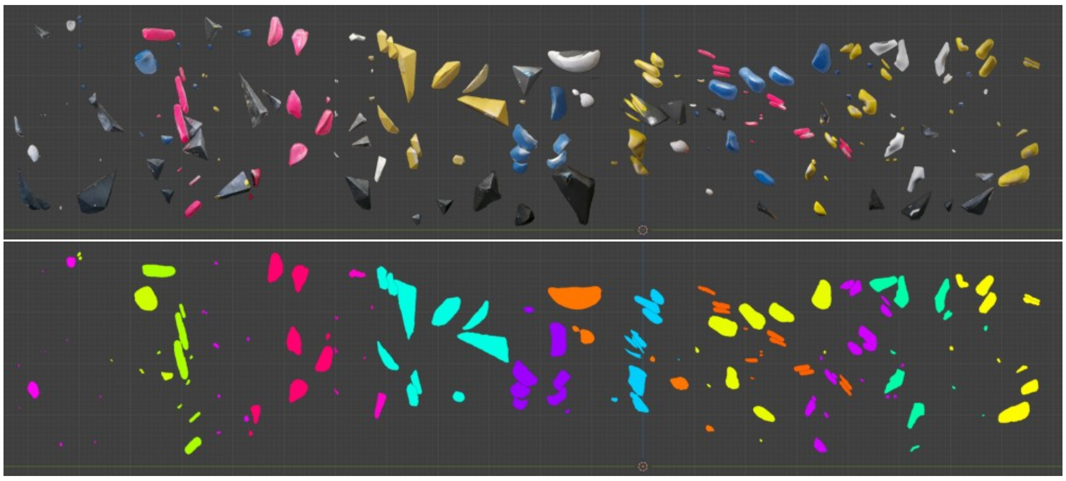
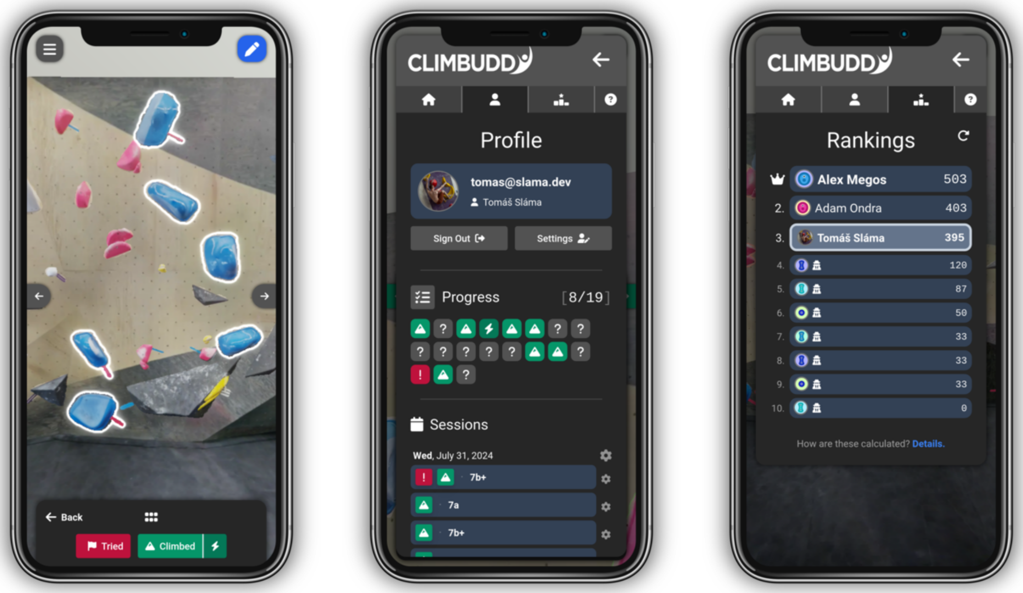


Figure 4.5: 3D hold instances (top) and the clustered instances (bottom) of the `crimp-1` dataset.

4.6 Usage

A proof-of-concept version of a climbing app called **CLIMBUDDY** that uses our 3D segmentation method, developed by Tomáš Sláma (thesis author) and Matěj Kripner, is available at `https://climbuddy.com/`, showcasing boulders for the Crimp climbing gym. The current functionality, besides displaying the current 3D models of the boulders online (Figure 4.6a), include various social features such as rating the boulders (Figure 4.6a), progress and session tracking (Figure 4.6b) and a global leaderboard (Figure 4.6c).

Future releases of the app include managing competitions and events, the ability to view older boulders, and more social features. Viewing old boulders is especially valuable, both for climbers, who can mark memorable ascents, and also for the gym, as the setters can gain insights about which types of boulders climbers prefer and set accordingly.



(a) Boulder overview & rating. (b) Progress & session tracking. (c) Global leaderboard.

Figure 4.6: Screenshots from the **CLIMBUDDY** web app.

5 Conclusion

We have implemented a fully automatic system for generation and subsequent 3D semantic, instance segmentation and clustering of a scene, focusing on indoor climbing gyms. In particular, we identified four subtasks: aligned 3D reconstruction (Section 3.1), image segmentation (Section 3.2), 3D mesh segmentation (Section 3.3) and instance clustering (Section 3.4).

We have used commonly available algorithms for well-established subproblems (i.e. COLMAP for 3D reconstruction), and developed a new algorithm for 3D semantic and instance segmentation, outperforming the state-of-the-art approaches due to the utilization of information from the 3D reconstruction, such as camera positions and images. The segmentation algorithm consists of three parts: first, images are segmented using an image segmentation network, with the segmentations projected onto the reconstructed mesh (Section 3.3.2). Next, graph clustering is used on the graph of projections, with edge weights being the projection overlaps (Section 3.3.2). Finally, a thresholding approach is used to create the final segmentations, using the ratio between positive and negative detections (Section 3.3.2).

To prove the viability of such system as a potential backend for an online climbing platform, a demo of such platform, referred to as **CLIMBUDDY**, was developed independently to the thesis by Tomáš Sláma (the thesis author) and Matěj Kripner, and is currently available at `https://climbuddy.com/`.

There is much room for improvement for our segmentation approach, namely with regards to speed – an efficient re-implementation of the computationally intensive parts of the algorithm would significantly speed up the inference. Furthermore, data from the 3D reconstruction, such as depth map estimates, could be used to improve the accuracy of the segmentation networks and thus the overall quality of the resulting 3D segmentations.

The availability of 3D models of the individual boulders gives rise to problems which couldn't be easily solved without them. *Automatic grade inference* can be used to suggest grade ranges for boulders without the input from the setter, reducing the work setters have to do after setting new boulders. *Learning-based route setting* can also be explored, using learning-based methods for setting indoor boulders. While there have been attempts to tackle these problems for standardized walls such as Moonboard [81, 82], there is little to no progress for regular climbing walls due to the prior lack of availability of 3D data.

Bibliography

1. J. L. Schönberger and J.-M. Frahm. "Structure-from-Motion Revisited". In: *Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016. 3, 8, 16, 41

2. J. L. Schönberger, E. Zheng, M. Pollefeys, and J.-M. Frahm. "Pixelwise View Selection for Unstructured Multi-View Stereo". In: *European Conference on Computer Vision (ECCV)*. 2016. 3, 8, 16, 41

3. M. Mareš and Tomáš. Valla. *Průvodce labyrintem algoritmů*. Druhé vydání. CZ.NIC, z.s.p.o., Praha, 2022. ISBN: 978-80-88168-63-8. 5

4. M. E. J. Newman and M. Girvan. "Finding and Evaluating Community Structure in Networks". *Physical Review E* 69:2, 26, 2004, p. 026113. ISSN: 1539-3755, 1550-2376. arXiv: cond-mat/0308217. URL: http://arxiv.org/abs/cond-mat/0308217. 6

5. M. E. J. Newman. "Modularity and Community Structure in Networks". *Proceedings of the National Academy of Sciences* 103:23, 6, 2006, pp. 8577–8582. ISSN: 0027-8424, 1091-6490. URL: https://pnas.org/doi/full/10.1073/pnas.0601602103. 6

6. U. Brandes, D. Delling, M. Gaertler, R. Goerke, M. Hoefer, Z. Nikoloski, and D. Wagner. *Maximizing Modularity Is Hard*. 30, 2006. arXiv: physics/0608255. URL: http://arxiv.org/abs/physics/0608255. 6

7. V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. "Fast Unfolding of Communities in Large Networks". *Journal of Statistical Mechanics: Theory and Experiment* 2008:10, 9, 2008, P10008. ISSN: 1742-5468. arXiv: 0803.0476 [physics]. URL: http://arxiv.org/abs/0803.0476. 6, 33

8. A. Clauset, M. E. J. Newman, and C. Moore. "Finding Community Structure in Very Large Networks". *Physical Review E* 70:6, 6, 2004, p. 066111. ISSN: 1539-3755, 1550-2376. URL: https://link.aps.org/doi/10.1103/PhysRevE.70.066111. 6

9. W. M. Rand. "Objective Criteria for the Evaluation of Clustering Methods". *Journal of the American Statistical Association* 66:336, 1971, pp. 846–850. ISSN: 0162-1459, 1537-274X. URL: http://www.tandfonline.com/doi/abs/10.1080/01621459.1971.10482356. 7

10. N. X. Vinh, J. Epps, and J. Bailey. "Information Theoretic Measures for Clustering Comparison: Is a Correction for Chance Necessary?" In: *Proceedings of the 26th Annual International Conference on Machine Learning (ICML '09)*. ACM, 2009, pp. 1073–1080. 7

11. B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng. *NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis*. 3, 2020. arXiv: 2003.08934 [cs]. URL: http://arxiv.org/abs/2003.08934. 8

12. B. Kerbl, G. Kopanas, T. Leimkühler, and G. Drettakis. *3D Gaussian Splatting for Real-Time Radiance Field Rendering*. 8, 2023. arXiv: 2308.04079 [cs]. URL: http://arxiv.org/abs/2308.04079. 8

13. D. Cernea. "OpenMVS: Multi-view Stereo Reconstruction Library". 2020. URL: https://cdcseacave.github.io/openMVS. 8, 41

14. R. Szeliski. *Computer Vision: Algorithms and Applications*. Second edition. Texts in Computer Science. Springer, Cham, 2022. 1 p. ISBN: 978-3-030-34371-2 978-3-030-34372-9. 8–10, 14

15. G. Bradski. "The OpenCV Library". *Dr. Dobb's Journal of Software Tools*, 2000. 10, 25

16. D. Lowe. "Object Recognition from Local Scale-Invariant Features". In: *Proceedings of the Seventh IEEE International Conference on Computer Vision*. Proceedings of the Seventh IEEE International Conference on Computer Vision. IEEE, Kerkyra, Greece, 1999, 1150–1157 vol.2. ISBN: 978-0-7695-0164-2. URL: http://ieeexplore.ieee.org/document/790410/. 11–14

17. D. G. Lowe. "Distinctive Image Features from Scale-Invariant Keypoints". *International Journal of Computer Vision* 60:2, 2004, pp. 91–110. ISSN: 0920-5691. URL: http://link.springer.com/10.1023/B:VISI.0000029664.99615.94. 11

18. E. Rublee, V. Rabaud, K. Konolige, and G. Bradski. "ORB: An Efficient Alternative to SIFT or SURF". In: *2011 International Conference on Computer Vision*. 2011 IEEE International Conference on Computer Vision (ICCV). IEEE, Barcelona, Spain, 2011, pp. 2564–2571. ISBN: 978-1-4577-1102-2 978-1-4577-1101-5 978-1-4577-1100-8. URL: http://ieeexplore.ieee.org/document/6126544/. 11

19. K. M. Yi, E. Trulls, V. Lepetit, and P. Fua. *LIFT: Learned Invariant Feature Transform*. 29, 2016. arXiv: 1603.09114 [cs]. URL: http://arxiv.org/abs/1603.09114. 11

20. J. L. Schönberger, T. Price, T. Sattler, J.-M. Frahm, and M. Pollefeys. "A Vote-and-Verify Strategy for Fast Spatial Verification in Image Retrieval". In: *Computer Vision – ACCV 2016*. Ed. by S.-H. Lai, V. Lepetit, K. Nishino, and Y. Sato. Vol. 10111. Springer International Publishing, Cham, 2017, pp. 321–337. ISBN: 978-3-319-54180-8 978-3-319-54181-5. URL: https://link.springer.com/10.1007/978-3-319-54181-5_21. 13

21. M. Muja and D. Lowe. "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration." In: vol. 1. VISAPP 2009 - Proceedings of the 4th International Conference on Computer Vision Theory and Applications. 2009, pp. 331–340. 14

22. D. Nistér. "An Efficient Solution to the Five-Point Relative Pose Problem". *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26:6, 2004, pp. 756–770. 15

23. K. Levenberg. “A Method for the Solution of Certain Non-Linear Problems in Least Squares”. *Quarterly of Applied Mathematics* 2:2, 1944, pp. 164–168. JSTOR: 43633451. URL: http://www.jstor.org/stable/43633451. 16

24. D. W. Marquardt. “An Algorithm for Least-Squares Estimation of Nonlinear Parameters”. *Journal of the Society for Industrial and Applied Mathematics* 11:2, 1963, pp. 431–441. JSTOR: 2098941. URL: http://www.jstor.org/stable/2098941. 16

25. S. Agarwal, K. Mierle, and T. C. S. Team. *Ceres Solver*. Version 2.2. 2023. URL: https://github.com/ceres-solver/ceres-solver. 16

26. M. A. Fischler and R. C. Bolles. “Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography”. *Communications of The Acm* 24:6, 1981, pp. 381–395. ISSN: 0001-0782. URL: https://doi.org/10.1145/358669.358692. 16

27. D. E. Rumelhart, G. E. Hinton, and R. J. Williams. “Learning Representations by Back-Propagating Errors”. *Nature* 323:6088, 1986, pp. 533–536. ISSN: 0028-0836, 1476-4687. URL: https://www.nature.com/articles/323533a0. 18

28. Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. “Gradient-Based Learning Applied to Document Recognition”. *Proceedings of the IEEE* 86:11, 1998, pp. 2278–2324. ISSN: 00189219. URL: http://ieeexplore.ieee.org/document/726791/. 20

29. L. Deng. “The Mnist Database of Handwritten Digit Images for Machine Learning Research”. *IEEE Signal Processing Magazine* 29:6, 2012, pp. 141–142. 20, 31, 39

30. J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. *You Only Look Once: Unified, Real-Time Object Detection*. 9, 2016. arXiv: 1506.02640 [cs]. URL: http://arxiv.org/abs/1506.02640. 21, 27, 28, 38

31. A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao. *YOLOv4: Optimal Speed and Accuracy of Object Detection*. 23, 2020. arXiv: 2004.10934 [cs]. URL: http://arxiv.org/abs/2004.10934. 21, 27, 28, 38

32. C. Li, L. Li, Y. Geng, H. Jiang, M. Cheng, B. Zhang, Z. Ke, X. Xu, and X. Chu. *YOLOv6 v3.0: A Full-Scale Reloading*. 13, 2023. arXiv: 2301.05586 [cs]. URL: http://arxiv.org/abs/2301.05586. 21, 27, 28, 38

33. A. Wang, H. Chen, L. Liu, K. Chen, Z. Lin, J. Han, and G. Ding. *YOLOv10: Real-Time End-to-End Object Detection*. 23, 2024. arXiv: 2405.14458 [cs]. URL: http://arxiv.org/abs/2405.14458. 21, 27, 28, 38

34. A. Agnihotri and N. Batra. “Exploring Bayesian Optimization”. *Distill* 5:5, 5, 2020, 10.23915/distill.00026. ISSN: 2476-0757. URL: https://distill.pub/2020/bayesian-optimization. 23, 24

35. J. Görtler, R. Kehlbeck, and O. Deussen. “A Visual Exploration of Gaussian Processes”. *Distill* 4:4, 2, 2019, 10.23915/distill.00017. ISSN: 2476-0757. URL: https://distill.pub/2019/visual-exploration-gaussian-processes. 23

36. K. Jamieson and A. Talwalkar. *Non-Stochastic Best Arm Identification and Hyperparameter Optimization*. 27, 2015. arXiv: 1502.07943 [cs]. URL: http://arxiv.org/abs/1502.07943. 24

37. L. Li, K. Jamieson, A. Rostamizadeh, E. Gonina, M. Hardt, B. Recht, and A. Talwalkar. *A System for Massively Parallel Hyperparameter Tuning*. 16, 2020. arXiv: 1810.05934 [cs]. URL: http://arxiv.org/abs/1810.05934. 24

38. S. Garrido-Jurado, R. Muñoz-Salinas, F. Madrid-Cuevas, and R. Medina-Carnicer. "Generation of Fiducial Marker Dictionaries Using Mixed Integer Linear Programming". *Pattern Recognition* 51, 2016, pp. 481–491. ISSN: 00313203. URL: https://linkinghub.elsevier.com/retrieve/pii/S0031320315003544. 25

39. F. J. Romero-Ramirez, R. Muñoz-Salinas, and R. Medina-Carnicer. "Speeded up Detection of Squared Fiducial Markers". *Image and Vision Computing* 76, 2018, pp. 38–47. ISSN: 02628856. URL: https://linkinghub.elsevier.com/retrieve/pii/S0262885618300799. 25

40. P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun. *OverFeat: Integrated Recognition, Localization and Detection Using Convolutional Networks*. 24, 2014. arXiv: 1312.6229 [cs]. URL: http://arxiv.org/abs/1312.6229. 27

41. R. Girshick, J. Donahue, T. Darrell, and J. Malik. *Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation*. 22, 2014. arXiv: 1311.2524 [cs]. URL: http://arxiv.org/abs/1311.2524. 27

42. S. Ren, K. He, R. Girshick, and J. Sun. *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*. 6, 2016. arXiv: 1506.01497 [cs]. URL: http://arxiv.org/abs/1506.01497. 27

43. K. He, G. Gkioxari, P. Dollár, and R. Girshick. *Mask R-CNN*. 24, 2018. arXiv: 1703.06870 [cs]. URL: http://arxiv.org/abs/1703.06870. 27

44. A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. *Attention Is All You Need*. 2, 2023. arXiv: 1706.03762 [cs]. URL: http://arxiv.org/abs/1706.03762. 27

45. N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov, and S. Zagoruyko. *End-to-End Object Detection with Transformers*. 28, 2020. arXiv: 2005.12872 [cs]. URL: http://arxiv.org/abs/2005.12872. 27

46. Z. Zong, G. Song, and Y. Liu. *DETRs with Collaborative Hybrid Assignments Training*. 10, 2023. arXiv: 2211.12860 [cs]. URL: http://arxiv.org/abs/2211.12860. 27

47. H. Zhang, F. Li, S. Liu, L. Zhang, H. Su, J. Zhu, L. M. Ni, and H.-Y. Shum. *DINO: DETR with Improved DeNoising Anchor Boxes for End-to-End Object Detection*. 11, 2022. arXiv: 2203.03605 [cs]. URL: http://arxiv.org/abs/2203.03605. 27

48. *Papers with Code - COCO Test-Dev Benchmark (Object Detection)*. URL: https://paperswithcode.com/sota/object-detection-on-coco. 27

49. Y. Zhao, W. Lv, S. Xu, J. Wei, G. Wang, Q. Dang, Y. Liu, and J. Chen. *DETRs Beat YOLOs on Real-time Object Detection*. 3, 2024. arXiv: 2304.08069 [cs]. URL: http://arxiv.org/abs/2304.08069. 27, 28

50. A. Kirillov, E. Mintun, N. Ravi, H. Mao, C. Rolland, L. Gustafson, T. Xiao, S. Whitehead, A. C. Berg, W.-Y. Lo, P. Dollár, and R. Girshick. *Segment Anything*. 5, 2023. arXiv: 2304.02643 [cs]. URL: http://arxiv.org/abs/2304.02643. 27, 28, 30

51. K. Wada. *Labelme: Image Polygonal Annotation with Python*. URL: https://github.com/wkentaro/labelme. 27, 42

52. Blender Online Community. *Blender - a 3D Modelling and Rendering Package*. manual. Blender Foundation, Blender Institute, Amsterdam. URL: http://www.blender.org. 28, 42

53. T.-Y. Lin, M. Maire, S. J. Belongie, L. D. Bourdev, R. B. Girshick, J. Hays, P. Perona, D. Ramanan, P. Doll'a r, and C. L. Zitnick. “Microsoft COCO: Common Objects in Context”. *CoRR* abs/1405.0312, 2014. arXiv: 1405.0312. URL: http://arxiv.org/abs/1405.0312. 28, 43

54. C. R. Qi, H. Su, K. Mo, and L. J. Guibas. *PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation*. 10, 2017. arXiv: 1612.00593 [cs]. URL: http://arxiv.org/abs/1612.00593. 29, 31

55. C. R. Qi, L. Yi, H. Su, and L. J. Guibas. *PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space*. 7, 2017. arXiv: 1706.02413 [cs]. URL: http://arxiv.org/abs/1706.02413. 29, 31

56. S. Chen, J. Fang, Q. Zhang, W. Liu, and X. Wang. *Hierarchical Aggregation for 3D Instance Segmentation*. 5, 2021. arXiv: 2108.02350 [cs]. URL: http://arxiv.org/abs/2108.02350. 29, 31

57. F. Engelmann, M. Bokeloh, A. Fathi, B. Leibe, and M. Nießner. *3D-MPA: Multi Proposal Aggregation for 3D Semantic Instance Segmentation*. 30, 2020. arXiv: 2003.13867 [cs]. URL: http://arxiv.org/abs/2003.13867. 29, 31

58. L. Han, T. Zheng, L. Xu, and L. Fang. *OccuSeg: Occupancy-aware 3D Instance Segmentation*. 28, 2020. arXiv: 2003.06537 [cs]. URL: http://arxiv.org/abs/2003.06537. 29, 31

59. L. Jiang, H. Zhao, S. Shi, S. Liu, C.-W. Fu, and J. Jia. *PointGroup: Dual-Set Point Grouping for 3D Instance Segmentation*. 3, 2020. arXiv: 2004.01658 [cs]. URL: http://arxiv.org/abs/2004.01658. 29, 31

60. J. Schult, F. Engelmann, A. Hermans, O. Litany, S. Tang, and B. Leibe. *Mask3D: Mask Transformer for 3D Semantic Instance Segmentation*. 12, 2023. arXiv: 2210.03105 [cs]. URL: http://arxiv.org/abs/2210.03105. 29, 31

61. T. Vu, K. Kim, T. M. Luu, X. T. Nguyen, and C. D. Yoo. *SoftGroup for 3D Instance Segmentation on Point Clouds*. 3, 2022. arXiv: 2203.01509 [cs]. URL: http://arxiv.org/abs/2203.01509. 29, 31

62. J. Lahoud, B. Ghanem, M. Pollefeys, and M. R. Oswald. *3D Instance Segmentation via Multi-Task Metric Learning*. 1, 2019. arXiv: 1906.08650 [cs]. URL: http://arxiv.org/abs/1906.08650. [29, 31]

63. J. Hou, A. Dai, and M. Nießner. *3D-SIS: 3D Semantic Instance Segmentation of RGB-D Scans*. 29, 2019. arXiv: 1812.07003 [cs]. URL: http://arxiv.org/abs/1812.07003. [29, 31]

64. X. Wu, L. Jiang, P.-S. Wang, Z. Liu, X. Liu, Y. Qiao, W. Ouyang, T. He, and H. Zhao. *Point Transformer V3: Simpler, Faster, Stronger*. 25, 2024. arXiv: 2312.10035 [cs]. URL: http://arxiv.org/abs/2312.10035. [29, 30, 46]

65. M. Kolodiazhnyi, A. Vorontsova, A. Konushin, and D. Rukhovich. *OneFormer3D: One Transformer for Unified Point Cloud Segmentation*. 24, 2023. arXiv: 2311.14405 [cs]. URL: http://arxiv.org/abs/2311.14405. [29, 30, 46]

66. C. Yeshwanth, Y.-C. Liu, M. Nießner, and A. Dai. "ScanNet++: A High-Fidelity Dataset of 3D Indoor Scenes". In: *Proceedings of the International Conference on Computer Vision (ICCV)*. 2023. URL: https://kaldir.vc.in.tum.de/scannetpp/. [29]

67. A. Dai, A. X. Chang, M. Savva, M. Halber, T. Funkhouser, and M. Nießner. "ScanNet: Richly-annotated 3D Reconstructions of Indoor Scenes". In: *Proc. Computer Vision and Pattern Recognition (CVPR), IEEE*. 2017. URL: https://www.scan-net.org/. [29]

68. W. X. Wang, G. X. Zhong, J. J. Huang, X. M. Li, and L. F. Xie. "INSTANCE SEGMENTATION OF 3D MESH MODEL BY INTEGRATING 2D AND 3D DATA". *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences* XLVIII-1/W2-2023, 14, 2023, pp. 1677–1684. ISSN: 2194-9034. URL: https://isprs-archives.copernicus.org/articles/XLVIII-1-W2-2023/1677/2023/. [29, 30]

69. Y. Yang, X. Wu, T. He, H. Zhao, and X. Liu. *SAM3D: Segment Anything in 3D Scenes*. 6, 2023. arXiv: 2306.03908 [cs]. URL: http://arxiv.org/abs/2306.03908. [30]

70. A. Karpathy, S. Miller, and L. Fei-Fei. "Object Discovery in 3D Scenes via Shape Analysis". In: *2013 IEEE International Conference on Robotics and Automation*. IEEE, 2013, pp. 2088–2095. [30]

71. P. F. Felzenszwalb and D. P. Huttenlocher. "Efficient Graph-Based Image Segmentation". *International Journal of Computer Vision* 59, 2004, pp. 167–181. [30]

72. I. Armeni, O. Sener, A. R. Zamir, H. Jiang, I. Brilakis, M. Fischer, and S. Savarese. "3D Semantic Parsing of Large-Scale Indoor Spaces". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2016, pp. 1534–1543. [30, 46]

73. R. Lambiotte, J.-C. Delvenne, and M. Barahona. "Laplacian Dynamics and Multiscale Modular Structure in Networks". *IEEE Transactions on Network Science and Engineering* 1:2, 1, 2014, pp. 76–90. ISSN: 2327-4697. arXiv: 0812.1770 [physics]. URL: http://arxiv.org/abs/0812.1770. [34]

74. N. Sharp and K. Crane. "You Can Find Geodesic Paths in Triangle Meshes by Just Flipping Edges". *ACM Trans. Graph.* 39:6, 2020. [38]

75. C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. "Array Programming with NumPy". *Nature* 585:7825, 2020, pp. 357–362. URL: `https://doi.org/10.1038/s41586-020-2649-2`. [41]

76. *Trimesh*. 2019. URL: `https://github.com/mikedh/trimesh`. [41]

77. Q.-Y. Zhou, J. Park, and V. Koltun. *Open3D: A Modern Library for 3D Data Processing*. 30, 2018. arXiv: `1801.09847 [cs]`. URL: `http://arxiv.org/abs/1801.09847`. [41]

78. G. Jocher, J. Qiu, and A. Chaurasia. *Ultralytics YOLO*. Version 8.0.0. 2023-01-10, 2023. URL: `https://ultralytics.com`. [41]

79. A. Buslaev, A. Parinov, E. Khvedchenya, V. I. Iglovikov, and A. A. Kalinin. "Albumentations: Fast and Flexible Image Augmentations". *Information* 11:2, 24, 2020, p. 125. ISSN: 2078-2489. arXiv: `1809.06839 [cs]`. URL: `http://arxiv.org/abs/1809.06839`. [43]

80. F. Nogueira. *Bayesian Optimization: Open Source Constrained Global Optimization Tool for Python*. 2014–. URL: `https://github.com/bayesian-optimization/BayesianOptimization`. [46]

81. Y.-S. Duh and J.-R. Chang. *Recurrent Neural Network for MoonBoard Climbing: Route Classification and Generation*. Stanford University, 2020. URL: `https://cs230.stanford.edu/projects_spring_2020/reports/38850664.pdf`. [51]

82. A. Dobles, J. C. Sarmiento, and P. Satterthwaite. *Machine Learning Methods for Climbing Route Classification*. Stanford University, 2017. URL: `https://cs229.stanford.edu/proj2017/final-reports/5232206.pdf`. [51]