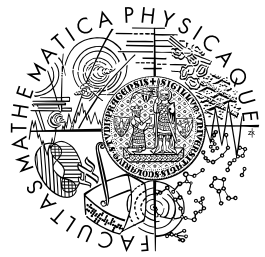


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Tomáš Sláma

Climbing route editor

Department of Applied Mathematics

Supervisor of the bachelor thesis: Martin Mareš
Study programme: Computer Science
Study branch: IPP1

Prague 2022

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

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

In ..... date ....

Author's signature

I'm grateful for the support from my friends and family, namely Kateřina Sulková, Matěj Kripner, Jakub Medek, Jan Černý, Benjamin Schiller and Jakub Pelc. I would also like to thank my supervisor Martin Mareš for patience and valuable feedback.

Title: Climbing route editor

Author: Tomáš Sláma

Department: Department of Applied Mathematics

Supervisor: Martin Mareš, Department of Applied Mathematics

Abstract: With the rising popularity of sport climbing and bouldering, there are increasing incentives to take it to the online space. This thesis does so in a way suitable for regular climbing gyms. A workflow for creating 3D hold and wall models and the accompanying editor has been developed, so that routes can be copied from the real world to the virtual one and be interacted with using software in a more immersive way.

Keywords: photogrammetry, routesetting, 3D modelling

Contents

Introduction **3**

1 Scanner **5**
1.1 Feature-based photogrammetry 5
1.1.1 Feature extraction 6
1.1.2 Feature matching 8
1.1.3 Bundle adjustment 8
1.1.4 Dense point cloud generation 9
1.1.5 Surface reconstruction 10
1.1.6 Texturing 11
1.2 Creating hold models 11
1.2.1 Turntable design 12
1.2.2 Scale and position inference 13
1.2.3 Environment setup 14
1.2.4 Camera settings 15
1.2.5 Image masking 16
1.2.6 Dual texture holds 16
1.2.7 Hold metadata 17
1.3 Creating the wall model 18

2 Editor **19**
2.1 Functionality 19
2.1.1 Movement 19
2.1.2 Wall and hold format 20
2.1.3 Editor modes 20
2.1.4 Hold selection 21
2.1.5 Import and Export 21
2.1.6 Building routes 24
2.1.7 Capturing images 25
2.1.8 Lighting 25

2.2 Developer Documentation . . . 25
2.2.1 Building from source . . . 26
2.2.2 Project structure . . . 26
2.2.3 Class interactions . . . 27

Conclusion **29**
Future developments . . . 29
Routesetter feedback . . . 29

Bibliography **31**

A Clis quickstart **35**
A.1 Setting up . . . 35
A.2 Contents . . . 36
A.3 Usage . . . 36
A.3.1 Using tasks . . . 36
A.3.2 Using scripts . . . 36

B Cled quickstart **39**
B.1 Setting up . . . 39
B.2 Key bindings . . . 39
B.2.1 Movement . . . 39
B.2.2 UI . . . 39
B.2.3 Editing . . . 40
B.2.4 Import/Export . . . 40
B.2.5 Capturing images . . . 40
B.2.6 Lighting . . . 40

C Clis source code **41**

D Cled source code **43**

E Cled release 2.3.1 **45**

Introduction

With the rising popularity of climbing and bouldering, in no small part due to the addition of the sport to the Tokyo 2020 Summer Olympics [1], climbing gyms are seeing a steady increase in new climbers. An obvious attraction to both climbers and route setters is being able to virtually view the current way the holds are set up (the current "setting"). This offers a great number of advantages, such as:

- archiving older settings for tracking trends like difficulty and style,
- saving an existing route to be rebuilt later (useful in competitions),
- being able to view the current setting online before visiting a gym and seeing if it is suitable (and possibly opting to go somewhere else if not),
- filtering boulders by difficulty and popularity,
- setting community-made boulders (if an editor exists) and
- adding a social aspect by (dis)liking and commenting on boulders, adding videos and beta[1] hints, connecting with other climbers, etc.

Models of boulders are gradually becoming used in competitive climbing and certain gyms, lead mainly by the OnlineObservation team [2]. Their approach is simple – take photos of the wall with the holds already on it and use them to generate a 3D model. This works really well for a one-time model generation, but becomes infeasible for a climbing gym that replaces boulders periodically, as the model would have to be regenerated each time, which takes a significant amount of time and specialized equipment for higher quality models. It is also difficult to individually highlight certain boulders and edit them if a change is made after the modelling.

This thesis attempts to solve the problem by focusing on what actually changes from setting to setting – the position of the holds on the wall. The repeated scanning of the holds and the wall adds redundancy, which could be removed by

[1] A term used for the intended sequence of moves for reaching the top of the route.

a program that edits the placements of the holds. This adds time initially, since models of the wall and the holds have to be created. However, it saves time after repeated settings and offers the aforementioned advantages, making it a viable option for commercial climbing gyms.

A system for semi-automatic scanning of climbing holds has been developed to efficiently create hold models, along with a workflow for modelling climbing wall interiors (Clis – the climber's scanner [3]). The generated data is then used in a virtual editor that can be used to efficiently model the routes (Cled – the climber's editor [4]).

The process and techniques behind the creation of the hold and wall models are described in section 1. The functionality and implementation of the editor are covered in section 2. Appendices A and B provide a quickstart for the scanning and editing, respectively.

Chapter 1

Scanner

When it comes to automatically generating textured 3D models from real-world objects, photogrammetry is the obvious choice. Broadly speaking, it is the process of obtaining information about the objects using their images (in our case, the model and the texture). It can be done cheaply, since it only requires high-quality images of the modelled object, which can be obtained using either a digital camera, or any recently released mobile phone (further discussed in section 1.2.4).

Besides photogrammetry, another option for creating 3D models is laser scanning. This was not a viable option due to the high price, because products that could be used (such as Revopoint POP, XYZ 1.0 Pro or the Creality 3D Scanner) cost upwards of 10 000 CZK (420 USD), which is non-trivial considering only a potential quality improvement. However, since a number of the aforementioned programs support laser scanning out of the box, the photogrammetry approach could be combined with laser scanning and thus enjoy the advantages of both.

1.1 Feature-based photogrammetry

Because the problem of creating 3D models from images is difficult and involves a lot of steps and specialized algorithms, many open-source and proprietary programs have been developed to simplify this task.

This thesis uses the Agisoft Metashape photogrammetry software [5]. While there are other suitable photogrammetry programs such as Meshroom (free), 3DF Zephyr (commercial) and COLMAP (free), Metashape was chosen for the following reasons:

- It created the highest-quality models out of the tested programs, given a representative sample of the hold images, which consisted of a foothold, two regular holds and a dual texture hold.

- It includes a well-documented Python API, along with a good GUI.

Since it is closed-source, there is no easy way to determine the exact algorithms used. However, a forum statement from the lead developer Dimitry Seymonov [6], with addition of license files for various open-source projects included with Metashape, give a good overview of the general methods used.

Starting with a set of images and ending with a textured model, the process can be split into the following parts:

1. **feature extraction:** for each image, find features that are stable under linear and affine transformations, viewpoint change and illumination
2. **feature matching:** find matching features across multiple images
3. **bundle adjustment:** find the approximate positions of the camera and the matched features (obtaining "structure from motion" data)
4. **dense point cloud generation:** obtain additional features (along with their normals) for higher model quality
5. **surface reconstruction:** reconstruct the surface of the model
6. **texturing:** map a texture from the images onto the model

The following sections aim to cover each of these in part.

1.1.1 Feature extraction

The image features are extracted using the Scale Invariant Feature Transform (SIFT) method described in Lowe [7, 8]. Each of the features is a vector describing some properties of a pixel of the image and should be invariant or partially invariant to linear and affine transformations, illuminance change and 3D transformations.

To find such vectors, a Gaussian scale-space is used. Formally, a scale-space for a given 2D image $I(x, y)$ is defined as a function

$$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y) \tag{1.1}$$

The $*$ symbol is a convolution of the two functions in x and y, and G is the Gaussian function

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} \, \mathrm{e}^{-(x^2+y^2)/2\sigma^2} \tag{1.2}$$

This operation is applied twice to image I with $\sigma = \sqrt{2}$, yielding images $I_{1,1}, I_{2,1}$, the difference of which yields the image G_1 (see figure 1.1). After this,

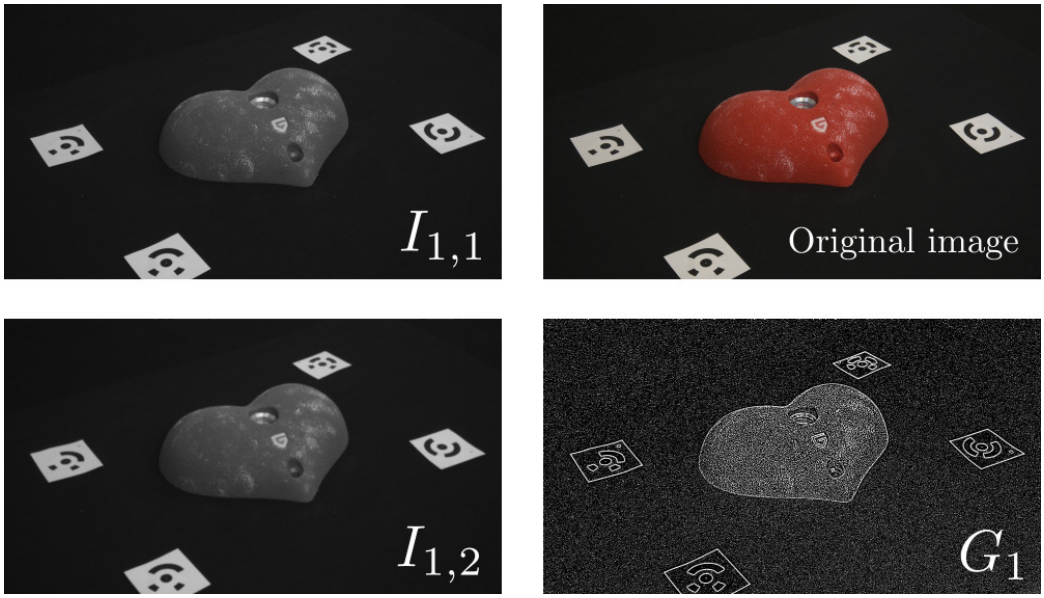


Figure 1.1: An example of the convolution and difference applied to a hold image. Generated using Python and the NumPy, SciPy and the Pillow libraries.

the image $I_{2,1}$ is downscaled by a certain factor (the original method uses 1.5 with bilinear interpolation) and the process is repeated, yielding $I_{1,2}$, $I_{2,2}$ and G_2, respectively.

To find the vectors, the local minima and maxima of the pixels of G_1 (compared to the 8 neighbouring pixels) are examined — if they are also minima/maxima in G_2 (accounting for the downscaling), they are selected as features. The intuitive idea behind this approach is to suppress the finer details of the image, since each of the main features of the original image should be present in the "coarse/blurred" version [9].

To calculate the vectors for the selected pixels, the vector's magnitude $M(x, y)$ and orientation $R(x, y)$ is determined using their neighbouring points via the following formulas:

$$M(x,y) = \sqrt{\left(I_{1,1}(x,y) - I_{1,1}(x+1,y)\right)^2 + \left(I_{1,1}(x,y) - I_{1,1}(x,y+1)\right)^2} \quad (1.3)$$

$$R(x,y) = \operatorname{atan2}\left(I_{1,1}(x,y) - I_{1,1}(x+1,y), I_{1,1}(x,y) - I_{1,1}(x,y+1)\right) \quad (1.4)$$

The $\mathrm{atan2(x, y)}$ function used returns the angle between the x axis and the point (x, y). It is a more general version of the $\mathrm{atan(y/x)}$ function, which doesn't distinguish points like $(-x, -y)$ and (x, y) (since only their ratio is used).

For the vectors to be invariant to orientation and contrast changes, a canonical representation is calculated from their local image gradients.

A note to be made is that SIFT can be utilized in other feature-matching problems such as stitching panoramas and finding objects in images, which could potentially be used to locate hold models.

1.1.2 Feature matching

Once the canonical representations of the vectors have been calculated, the images are examined pairwise. For each feature in one image, the nearest neighbour (in terms of the vector distance in a suitable norm) is found in the other. To filter out features that only appear in one of the images, a limit is imposed on the ratio of the distances between the closest and the second closest neighbour — intuitively, good features should have only one matching candidate.

Since the number of features to match can be quite large and no efficient exact algorithm exists for the problem of finding the nearest neighbour, an approximate algorithm called best-bin-first is used [10]. This algorithm returns the nearest neighbour with a high probability, otherwise returning a close one. It internally uses a k-d tree to partition the space into bins of the feature vectors, which it then searches in an efficient way.

1.1.3 Bundle adjustment

After the features have been determined and matched across the images, the next step is to simultaneously calculate their position in the 3D space and also the positions of the cameras. This is referred to as the bundle adjustment problem [11], the name referring to bundles of rays from the predicted positions of the points going into the cameras.

Formally, we can model the scene by n vectors of 3D points $\mathbf{X}_{i\in[n]}$ (sometimes called a sparse point cloud), taken from m cameras with parameters given by vectors $\mathbf{P}_{j\in[m]}$ (position, focal length, etc.). The features can then be seen as observations $\overline{\boldsymbol{x}}_{ij}$ of point i from camera j.

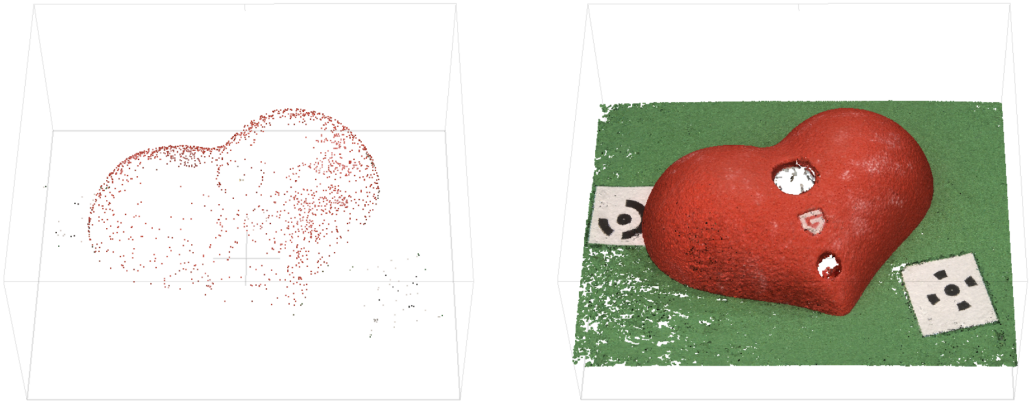
Let's now assume that we have a function $\boldsymbol{x}(\mathbf{X}_i, \mathbf{P}_j)$ that, given the feature position $\mathbf{X}_i$ and camera parameters $\mathbf{P}_j$, models the calculated observation position $\mathbf{x}_{i,j}$. Given this function, the bundle adjustment problem can be formulated as a minimization problem of the function

$$\Delta x_{i,j}(\mathbf{X}_i, \mathbf{M}_j) = \overline{\boldsymbol{x}}_{ij} - \boldsymbol{x}(\mathbf{X}_i, \mathbf{P}_j) \tag{1.5}$$

over the observed features, with an appropriate cost function. For estimation using nonlinear least squares, the cost function to minimize is the weighted sum of squared error, formulated as

$$\frac{1}{2}\sum_{i,j} \Delta x_{i,j}(\mathbf{X}_i, \mathbf{M}_j)^T W_{i,j} \Delta x_{i,j}(\mathbf{X}_i, \mathbf{M}_j) \tag{1.6}$$

where each W_i is a symmetric positive definite matrix indicating the weight of the vector, chosen to be approximately the inverse measured covariance of $\overline{\boldsymbol{x}}_{ij}$ (to account for the accuracy of measured feature points).



(a) Sparse point cloud.

(b) Dense point cloud.

Figure 1.2: Sparse and dense point cloud comparison on a sample hold, generated using Agisoft Metashape.

1.1.4 Dense point cloud generation

Since the feature matching step generates a small number of features compared to what should be used to generate a "high-quality" mesh, additional feature points have to be generated. The approach is based on generating depth-maps (grayscale images indicating the distance of each pixel from the camera) for each image and merging them together [12].

Stereo pair selection

First, each (target) image is paired with another (reference) image to generate its depth-map. The reference image should view the object from a similar direction but a different angle that is not too close (5°) and too far (60°). The camera centers should additionally be between $2\overline{d}$ and $0.05\overline{d}$, where $\overline{d}$ is the median of the distances between the images within the viewing angle.

The reference image is selected as the one minimizing the product of the viewing angle and the camera center distance, while the others (sorted and limited to 10) are considered the image neighbourhood N_i.

Depth-map computation

The core of the computation is based on the PatchMatch algorithm [13].

A random 3D point, along with a normal (forming a plane), is assigned to each of the pixels of all images, such that they are in the viewing ray of their camera. For each of the pixels, a square "patch" centered on the pixel is placed on the image. Then, for each pixel in it (across the reference image), a matching cost is calculated (how well the point corresponding to the pixel fits with the others). This cost is then iteratively improved either by moving the point to the plane of one of its neighbours, or by randomly changing it.

Depth-map refinement

To make the depth of the common areas consistent, each pixel of the depth-maps is considered — if its corresponding point is close to its position in the neighbouring images N_i (when projecting it onto the neighbouring image and then back using the neighbouring image depth-map), it is retained; otherwise it is removed from the depth-map.

Depth-map merging

The calculated depth-maps for each image are merged by taking their pixels' points and combining them to form the resulting dense point cloud. To prevent duplication, the depth-maps are filtered by comparing their overlap with their neighbouring images.

1.1.5 Surface reconstruction

For surface reconstruction, the Poisson surface reconstruction algorithm [14] is used. It takes points in 3D space and their normals (oriented towards the object), and constructs a function that represents the surface.

It does this by first computing a piecewise indicator function χ that is 0 outside of the model and 1 inside. The gradient of this indicator function (when convolved with a suitable smoothing filter) is a vector field, whose values near the surface are the point normals.

The problem can thus be formulated as a minimization problem in terms of the difference between observed normals as a vector field $\vec{V}$, and the indicator function gradient (denoted as ∇):

$$\min_{\chi} \|\nabla\chi - \vec{V}\| \tag{1.7}$$

Since we only have a set of points and their normals, the difference in vector fields is approximated by a discrete sum. Least squares are again used to find the solution (using the euclidean norm for the minimized vectors).

1.1.6 Texturing

All that is left is to create a texture for the modeled object. Since both the model and the camera position are known at this point, the images can be projected onto the faces in view (given that the face is pointed towards the camera). Additional steps can be taken to normalize certain properties of the images, such as their brightness, contrast, etc.

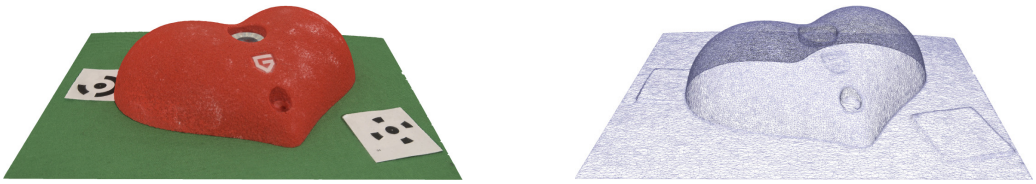
1.2 Creating hold models

Since a regular climbing wall contains hundreds or even thousands of holds of varying sizes, it would be infeasible to model each of them manually. It is therefore important to automate as many steps as possible so that the amount of manual work done is minimized.

It would be convenient if this could be done with the holds still on the wall, but this is difficult to accomplish for a few reasons:

- the scale and position of each hold would be difficult to infer,
- the hold boundary could not be clear (holds can be placed on top of each other, they can be touching side-by-side, etc.), and
- the quality of the models (texture and model-wise) might not be sufficient.

Having the holds off the wall, one way to automate the scanning is to construct a turntable that turns the hold, with a stationary camera taking the images from different angles. If both the turntable and the camera are programmable, the time



(a) Textured view. (b) Wireframe view.

Figure 1.3: Model of a sample hold, generated using Agisoft Metashape.

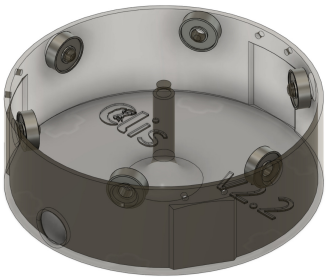
spent doing manual work would only include replacing one hold with another when the scanning is finished.

It would be unreasonable to expect the turntable to carry any hold, since larger structures can weigh tens of kilograms and measure more than a meter. However, both the carry weight and size should still be maximized. Additionally, faster turntable rotation corresponds to quicker scanning and should thus also be maximized.

1.2.1 Turntable design

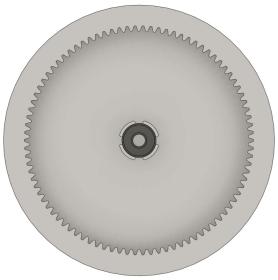
A three stepper-motor turntable (figure 1.4) has been developed and 3D printed using the Fusion 360 CAD software [15] and an Ender Pro 5 3D printer. An Arduino board and A4988 motor controllers (figure 1.5) are used for fine motor control. The models, the code and a short tutorial on the turntable construction are freely available at the Clis GitHub page [3].

Communication with the Arduino is implemented via the USB port. The protocol is simple — the Arduino listens on the serial port until it receives a string. After reading it, it interprets it as integer and turns the corresponding number of

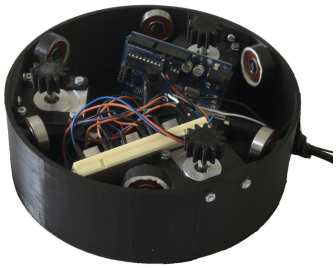


(a) Bottom part (side).

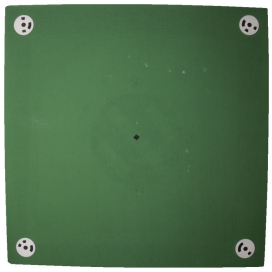
(b) Bottom part (top).



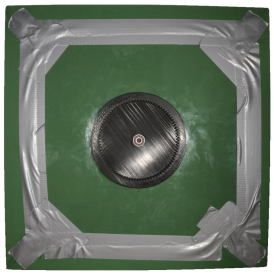
(c) Top part.



(d) Bottom part (side).



(e) Plexiglass (top).



(f) Plexiglass (bottom), connected to the top part.

Figure 1.4: Parts of the model of the turntable, generated using Fusion 360 (top row) and photographed using the Nikon z50 DLSR (bottom row).

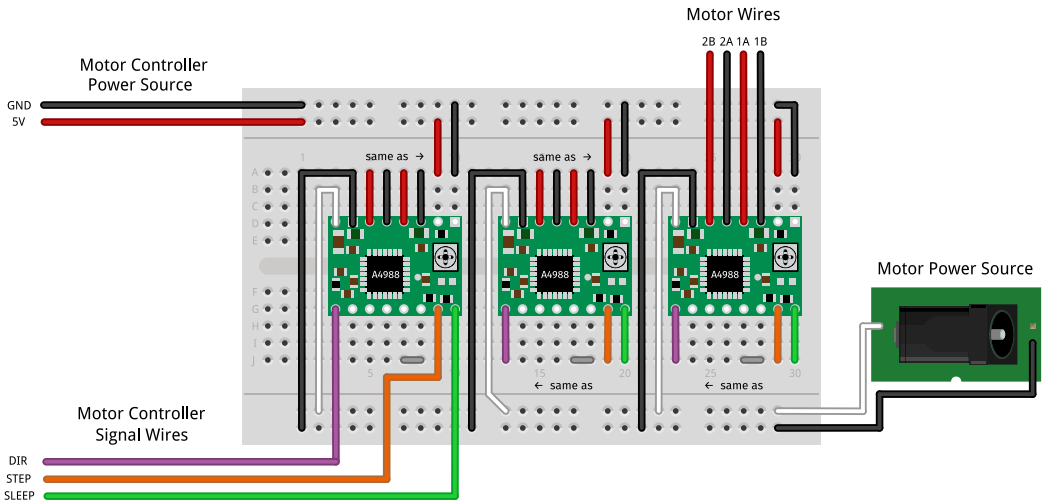


Figure 1.5: The wiring diagram of the turntable, created using Fritzing [17].

degrees. After the turn, it writes this number back to the serial port and again starts listening for another string.

The gphoto2 [16] library is used to control the camera. Its API can capture images, focus the camera and also interact with the camera file system. All of the images were taken using Nikon z50 with a Nikkor 18-55mm f/3.5–5.6 lens.

To minimize the amount of friction, the only points of contact between the top and bottom part of the turntable are 7 bearings (6 placed in a circular pattern and 1 placed in the middle). Additionally, gears mounted to the motors turn the top of the turntable, achieving a $90/12 = 7.5$ reduction. This allows the turntable to handle objects of up to 8 kg, with heavier objects requiring manual turning.

The top area is connected to a 50 cm $\times$ 50 cm plexiglass with a marked center and 4 markers (to infer scale and position, see section 1.2.2). This means that any hold that can fit on the plexiglass (and possibly extend beyond it, as long as it is not blocking the markers) can be scanned automatically.

The resulting workflow can model a single hold in 1 minute of scanning (varying depending on the lighting conditions and the number of photos, the default being 12) and additional 4 minutes of processing.[1]

1.2.2 Scale and position inference

The hold size and position (in local coordinates) needs to correspond to its real counterpart. To do this automatically, markers are placed around it — since their position in space is known (measured in advance), it can be used to infer the

[1]Computed using Metashape on Linux with AMD Ryzen 7 1700 and AMD Radeon RX 5500.

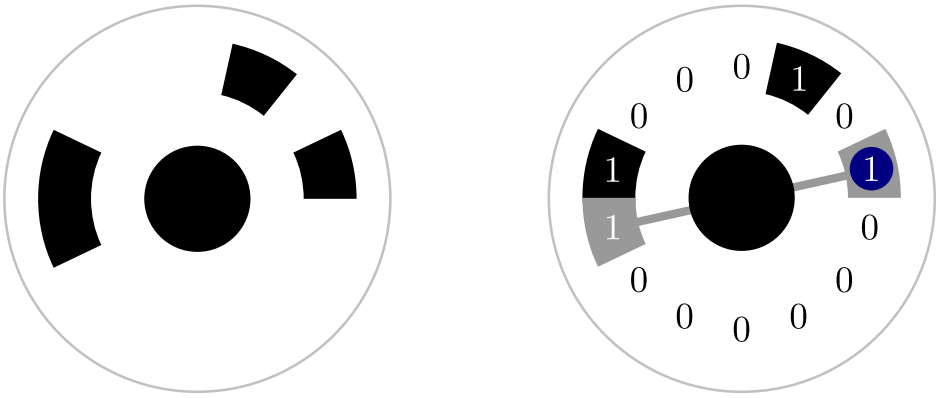


Figure 1.6: The 18th 14-bit marker (left) and its corresponding binary value (right). Grey denotes an opposing pair of 1 bits, blue denotes the parity bit. See [20] for a comprehensive list of all markers, along with code that generates all valid marker binary values.

correct scale and position of the entire model.

The type of the markers supported by Metashape is based on Schneider [18] and Thielbeer [19]. Each circular target contains an inner and outer circle, the inner being used for locating the marker in the image and the outer for encoding the marker data.

The exact number of stored bits depends on the size of the marker, most common being 12 and 14, respectively. The 0 bits correspond to white segments and 1 to black ones. For decoding correctness and robustness, the following constraints are additionally imposed on the targets:

- the targets must be rotationally invariant
- the first bit is a parity bit (0 for even, 1 for odd)
- there must exist an opposing pair of 1 bits

After the positions of the markers are determined in 3D space (they are a special type of feature described in section 1.1.1 that uses a different algorithm to be located across the images), it is just a matter of simple linear algebra to transform the model appropriately.

1.2.3 Environment setup

Two LED lights, mounted on adjustable stands and covered with baking paper (for light diffusion) were used to achieve good lighting conditions. A large black cloth was used as the background to reduce false image features.

Figure 1.7: An image of the setup during scanning, including the taken image.

The entire setup can be carried in a backpacking pack,[2] making it portable. Additionally, all of the setup components (excluding the camera) can be purchased for under 4000 CZK (180 USD), making it affordable.

1.2.4 Camera settings

Photogrammetry works best with sharp, high-resolution images with the least amount of noise possible. To achieve this, the following settings were used to take all of the images:

- resolution: 5600×3728
- ISO sensitivity: 320 — lower sensitivity generally means less image noise
- aperture: 22 — higher aperture widens the focal plane, which results in the entire scanned object being in focus and thus reducing blur
- exposure time: apx. 1.5 s — calculated from the sensitivity and the aperture for the image to have the correct brightness

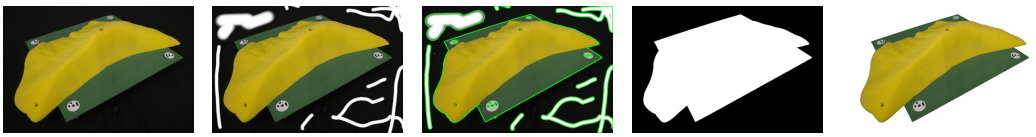
The settings could only be used since the camera is static and the hold doesn't move while taking the image, otherwise the exposure time would be too high.

[2] Excluding the plexiglass, which is inflexible and the camera, which is fragile.

1.2.5 Image masking

The photogrammetry software expects the images to be viewed from various angles. This poses a problem for the turntable setup, since the background might not be perfectly dark and thus create false image features. Fortunately, the contrast between the scanned area and the background is large enough to automatically construct a mask to filter the background out. This is done using OpenCV [21] (as seen in figure 1.8) by

1. converting the image to grayscale,
2. generating a binary image using a threshold on the pixel brightness,
3. applying a contour-generating algorithm [22] on the image and finally
4. using the largest (flood-filled) contour as the image mask.



(a) Original. (b) Smudges. (c) Hold contours. (d) Mask. (e) Result.

Figure 1.8: The process of masking a hold, generated using OpenCV. A number of smudges were artificially added to the image to show resiliency towards errors.

1.2.6 Dual texture holds

Glossy surfaces are a problematic surface type for photogrammetry, since the angle under which they are viewed changes their appearance due to light reflections. This problem is present for "dual-texture" holds which, as the name suggests, contain two textures — a matte texture that is meant for the climber to hold (or step) onto, and the glossy texture which is (usually) not.

A standard way of dealing with glossy surfaces in photogrammetry is to cover them in something that is matte. While this does solve the problem of model generation, it ruins the texture, because the matte solution is usually opaque, thus requiring another set of images for texturing.

In our case, however, the solution is pretty straightforward — cover them in climbing chalk. Since it is matte, it reduces the reflections of the glossy surface and provides additional feature points, making it easier for the photogrammetry software to reconstruct the model. Additionally, since climbing chalk will be

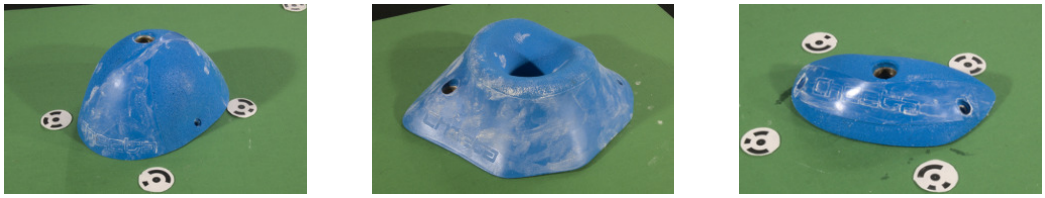


Figure 1.9: Example of holds with chalk applied. Some reflections are still visible, but they are not as pronounced as with no chalk applied.

applied to the holds by climbers during regular usage anyway, there is no need to obtain more images for texturing.

It is worth noting that this solution is not perfect — since the chalk doesn't cover the hold entirely, some reflections persist and can introduce model inaccuracies (see figure 1.9). In such cases, manual modelling using Blender [23] to reconstruct the missing hold parts is required.

1.2.7 Hold metadata

After all models have been generated, a `yaml` metadata file is created for storing their properties. It is a dictionary, with keys being the hold models and the attributes their properties. The specification of a single item is as follows:

```yaml
<the first 12 characters of the sha-256 hash of model data>:
  color: [<name>, <hex value>]                  # strings
  type: <the type of hold>                      # string
  manufacturer: <the name of the manufacturer>  # string
  labels: [<list>, <of>, <custom>, <labels>]    # strings
  volume: <a float volume of the hold>          # float
  date: <the date the hold model was created>   # datetime
```

All of the specified attributes are optional. Some (like volume and date) are automatically inferred, while others (type, manufacturer) have to be added manually. The first 12 characters of the hash were chosen for readability purposes and because collisions are very unlikely.[3]

[3]This is a generalization of the birthday paradox. Total pool of hashes has size $N = 16^{12}$. The number of items needed for the probability of collision being $\geq \frac{1}{2}$ can be calculated as the largest r for which $\prod_{i=1}^{r} \frac{N-r}{N} \geq \frac{1}{2}$ holds. An approximation for $r = \left\lceil \sqrt{2N \ln 2} + 1 \right\rceil$ [24] yields 1 177 411, which is likely more than enough holds.

1.3 Creating the wall model

Since the wall model only gets created once, there is no real reason to automate this process. Nevertheless, there is no reason to create it manually either, since photogrammetry can again be used.

First, a set of 188 images was taken and used by the Agisoft Metashape software to create a reference model. This model was then imported and edited in Blender to remove inaccuracies. After the changes, the resulting model was reimported to Metashape to be textured and exported as the final model.

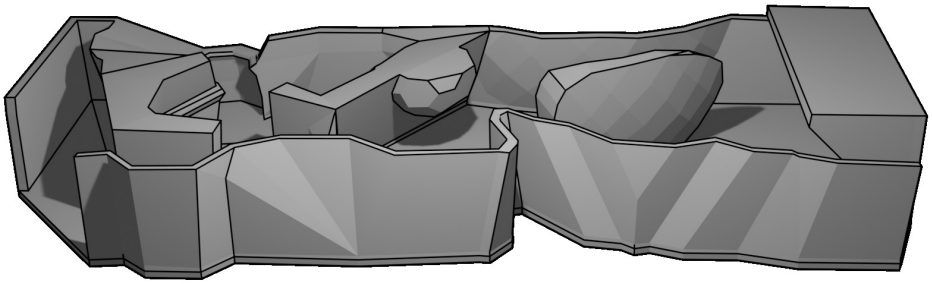


Figure 1.10: The model of the Smíchoff bouldering wall, created using Agisoft Metashape and Blender.

Chapter 2

Editor

There are a number of requirements on the editor functionality, namely that

- the setting should be in 3D, visual, intuitive and efficient,
- the import and export format should be simple and human-readable,
- the code should be open-source,
- it should be cross-platform for all major operating systems and
- it should contain image capture functionality.

There are many frameworks that support 3D, the two main groups being user interface software (Qt, GTK+, Electron, etc.) and game engines (Unreal Engine, Godot, Source, Three.js, etc.). The choice was made to use a game engine because the main functionality of the editor is moving around and interacting with objects in 3D, which game engines are arguably more suited for.

Unity was chosen as the framework to implement the editor in, mainly due to the author's experience with Unity and C# in general, along with a large ecosystem of packages that were used to simplify the implementation and allow for rapid prototyping (see section 2.2).

The editor code is open-source and cross-platform (Linux, Windows, Mac), with newest releases being freely available at the project releases page.

2.1 Functionality

2.1.1 Movement

Movement is implemented using keyboard in a way commonly used in 3D programs. The arrow or `WSAD` keys provide left, right, forward and backward movement, while the `Space` and `Shift` keys allow flying up and down.

Physics and gravity are also implemented, with the character being pulled to the ground when not flying, and colliding with both the wall and the holds.

2.1.2 Wall and hold format

For the editor to correctly load the scene, it expects the wall and holds to be in the Wavefront object (`.obj`) file format, possibly with a Material Template Library (`.mtl`) file and a corresponding bitmap texture. It has specific requirements on naming and location of the files, which differs slightly from Clis. This is because the editor uses only a subset of the files generated from Clis along with a few that Clis doesn't generate. To make things easier, the editor contains a Python script to perform the import automatically.

The wall can also contain a `yaml` metadata file for wall-specific things: the possible route grades, names of the setters and wall zones. This is to simplify editing route properties by selecting the right option from a dropdown, instead of having to manually type it in. Here is an example of one such file:

```yaml
Grades: ["black", "purple", "red", "salmon", "blue", "yellow"]
Setters: ["Danny", "Bert"]
Zones: ["pool", "overhang", "comp"]
```

2.1.3 Editor modes

The editor contains three modes in which it operates, with the current mode being always displayed in the top right. While this might seem like an implementation detail, it is helpful for general usage, because the user always knows which state the editor is in and can refer to the respective part of the documentation. An obvious inspiration is the Vim text editor [25], which uses the same concept.

The modes are the following:

- **NORMAL** — the mode the user is normally in; hovering highlights holds, which can be either picked up or selected
- **HOLDING** — the mode in which the user is holding a specific hold and can place it, or switch to the previous/next one from the selection
- **ROUTE** — the mode in which the user is modifying the selected route by adding/removing new holds or modifying the route parameters

2.1.4 Hold selection

Quickly selecting which holds to place on the wall is crucial for effective virtual route setting, because it largely dictates the time the route setting takes. The editor contains a menu (accessed via pressing either `Tab` or `Q`) from which a subset of holds can be filtered and used while editing.

Holds can be filtered by their color, type, manufacturer, or custom labels. By default, the currently filtered holds are also sorted first by their color and then by their volume. Additionally, hovering on each of the menu items rotates them around, which is useful when the static hold image isn't descriptive enough.

To make it further apparent which hold is currently held and which one is previous/next, all three are displayed at the bottom of the screen.

Figure 2.1: The hold picker UI (no filters applied) on a sample dataset.

2.1.5 Import and Export

The import and export format for the project files is a human-readable `yaml` file containing information about the state of the holds, routes, paths to the models and other metadata. This makes usage in other applications easier (both for viewing and modification), since it is well readable and parseable. Here is how the format looks like:

```yaml
Version: 2.3.1 # Editor version (for compatibility reasons)

Player: # player settings
  Light: false # whether the player's light is on
  Position: # the player position as a vector
    x: 4.66010237
    y: 1.93126345
    z: -8.63198185
  Orientation: # the player orientation (up/down, left/right)
    x: 35.8053398
    y: 123.996086
  Flying: false # whether the player is currently flying

WallModelPath: /home/tom/Wall/wall.obj
HoldModelsPath: /home/tom/Holds

Holds: # a dictionary of hold instances
  f21cde933ad9: # a unique ID for the hold instance
    BlueprintId: 4bdff2704462 # an ID of the hold model
    State:
      Position: # the position of the hold in space
        x: 1.20504105
        y: 0.378549755
        z: -7.67928696
      Normal: # the normal of the hold, away from the wall
        x: 0.994500279
        y: 0.0156783238
        z: 0.103554823
      Rotation: 0 # the hold rotation about the normal
      Flipped: false # whether the hold is horizontally flipped
  0436ff53a1d8:
    BlueprintId: 4bdff2704462
    State:
      Position:
        x: 1.46525836
        y: 0.472094655
        z: -8.19389153
      Normal:
        x: 0.752805889
        y: -0.0226488542
```

```
        z: 0.657852829
      Rotation: 1.21387374
      Flipped: false

Routes: # a list of routes
- Name: The Nose
  Grade: VI
  Zone: El Capitan
  Setter: Mother Nature
  HoldIDs: # which hold instances the route contains
  - 0436ff53a1d8
  - f21cde933ad9

StartingHoldIDs: # a list of starting holds
- f21cde933ad9

EndingHoldIDs: # a list of ending holds
- 0436ff53a1d8

SelectedHoldBlueprintIDs: # a list of currently selected holds
- f21cde933ad9
- 0436ff53a1d8

Lights: # a list of point lights around the wall
  Positions:
  - x: 3.71035504
    y: 3.53241396
    z: -7.11679459
  - x: 5.59678602
    y: 3.53241396
    z: -3.97351503
  Intensity: 0.2 # how strong they are
  ShadowStrength: 0.2 # how hard their shadows are

CaptureSettings: # image capture settings
  ImagePath: /home/tom/Pictures # captured images path
  ImageSupersize: 2 # captured images resolution scaling
```

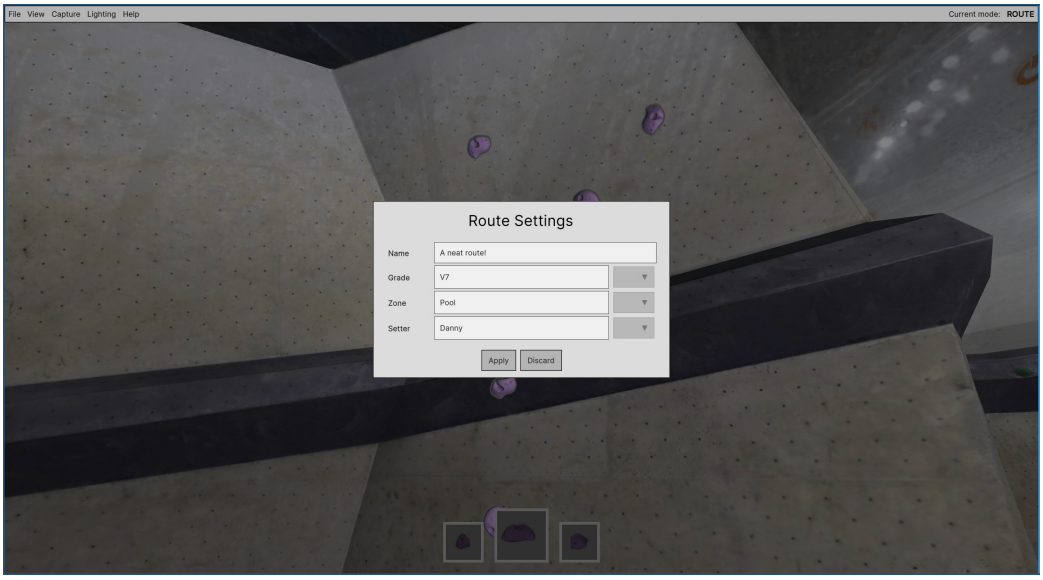


Figure 2.2: Route settings for a sample route.

2.1.6 Building routes

After holds have been placed, they can be combined to form routes by right-clicking to select a hold and then `Ctrl` + left/right clicking to add other holds to the route. Right-clicking an already selected route brings up the route settings menu (figure 2.2), from which various route attributes can be set.

In addition, a hold can be marked as the starting hold or the ending hold of a route by hovering it and pressing `t` (top) or `b` (bottom) respectively. The hold is then denoted by a marker, as seen in figure 2.3. This marker always moves to the bottom of the hold so it is visible, no matter the hold position and orientation.

Figure 2.3: Marker position on the same hold with different orientations.

2.1.7 Capturing images

The editor contains functionality for capturing the current image of the wall. This can be done anywhere in the editor by either selecting the option from the toolbar, or by pressing `Ctrl+P` (see figure 2.4). The image is then saved to the default system "Pictures" folder, unless configured otherwise.

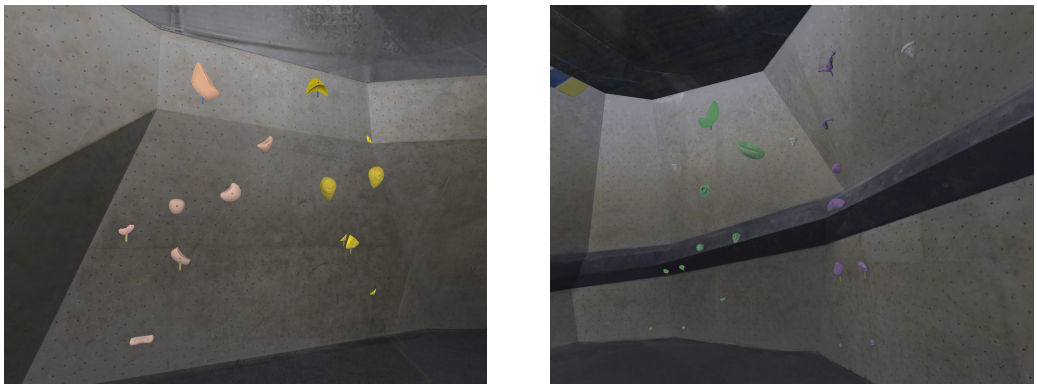


Figure 2.4: Images generated using the editor capture functionality.

2.1.8 Lighting

Since each wall has different lighting needs and specifying them in the wall metadata would be too impractical, the editor supports placing lights on the current player position and removing them. This can be done either from the `Lighting` section of the toolbar, or by using `F` to toggle the player lighting and `Ctrl+F` to place one at the player position.

2.2 Developer Documentation

The project has been developed on Linux in Unity Editor 2021.2.19f, using Rider as the code editor of choice. All of the code (besides the freely used packages, which have their own licensing) is licensed under GPL-3.0, chosen for the perpetual availability of the source code for any and all derived works.

In addition to the editor code, the following free packages were used:

- **Quick Outline** [26] — object outline creator,
- **Runtime OBJ Importer** [27] — an `.obj` file importer and
- **Unity Standalone File Browser** [28] — a cross-platform file browser.

The new Unity UI package (UIElements) was used for creating the user interface. It uses structured UXML (Unity XML) files to describe each respective UI part, which are styled using USS (Unity Style Sheets). It was chosen over the older, GameObject-oriented approach because it is more structured and scalable (no need to manually create game objects).

Git is used for version control, with the entire repository being hosted on GitHub [4]. Each release is automatically built for all supported platforms from the source code using GitHub Actions on tagged commits. The version numbering scheme follows Semantic Versioning 2.0 [29].

2.2.1 Building from source

To build the project from source, first clone the repository. Next, open the project either by adding it to Unity Hub, or by using the Unity Editor 2021.2.19f (other 2021.x versions will likely also work but weren't tested). Finally, build the project by pressing `Ctrl+B`.

2.2.2 Project structure

The codebase can be broadly split into the following parts:

- Controller classes:
  - `CameraController` — controls the camera movement
  - `MovementController` — controls the player movemen
  - `EditorController` — controls interaction with holds
  - `UIKeyboardController` — controls UI using the keyboard
- Manager classes:
  - `EditorModeManager` — manages the current editor mode
  - `HighlightManager` — manages hold highlighting
  - `HoldStateManager` — manages the state of the holds
  - `LightManager` — manages the wall lights
  - `RouteManager` — manages routes
  - `Preferences` — manages preferences
- User Interface classes:
  - `BottomBar` — controls bottom bar

  - `HoldPickerMenu` — controls hold picker menu
  - `LoadingScreenMenu` — controls loading screen
  - `PauseMenu` — controls the pausing
  - `PopupMenu` — controls the popups
  - `RouteSettingsMenu` — controls route settings menu
  - `RouteViewMenu` — controls route view menu
  - `SettingsMenu` — controls settings menu
  - `ToolbarMenu` — controls the main toolbar
- Import and Export classes:
  - `Importer` — facilitates importing state
  - `Exporter` — facilitates exporting state
  - `HoldLoader` — loads the holds
  - `WallLoader` — loads the wall
- `Utilities` — random utility functions and classes

2.2.3 Class interactions

The interactions of the user with the code are done either through one of the controller classes (movement, camera, hold interactions) or through the UI classes (toolbar, hold picker, popups, etc.).

The manager classes encapsulate functionality that needs to be managed, such as the holds, routes, lights and highlighted objects. They don't depend on any classes (the only exception being `HighlightManager`, which relies on an instance of `HoldStateManager` for making all holds transparent when a route is selected).

The UI classes themselves have more complex dependencies due to the fact that certain parts of the UI depend on others. Examples include

- `BottomBar` displays the previous/current/next selected hold, which are stored in `HoldPickerMenu`,
- `RouteViewMenu` relies on `SettingsMenu`, which it displays when one of its routes is double clicked and
- `ToolbarMenu` contains buttons to display various other parts of the UI, such as `SettingsMenu`, `RouteViewMenu` and also `HoldPickerMenu`.

Conclusion

Clis and Cled have been successfully tested on the Smíchoff bouldering wall, creating a dataset of 119 holds (in 4 hours), the textured wall model (in 5 hours) and virtually setting a number of routes from the wall.

Future developments

Future developments include implementing a web interface and a mobile application, so that the exported routes can be interacted with by regular users. Such interface should support the ideas outlined in the introduction, such as route filtering, liking/disliking, videos and beta hints.

Additional features that were beyond the scope of this project but will be implemented in the future are automatic hold placement from a photo of the wall using SIFT (described in section 1.1.1), "snapping to hole" for holds that can only be placed in specific ways and a large number of quality-of-life improvements and bugfixes. Laser scanning is also a potential improvement for model generation. Furthermore, the option of using VR to increase setting efficiency will be explored.

Another possible use for the data produced by Clis and Cled is for the purposes of simulating the movements of a climber on the wall.

The end goal is to provide Clis and Cled as a service for climbing gyms all over the world, so that climbers have an easier time picking where to go climbing and can better find people to go climbing with, and routesetters have an easier time of building higher-quality routes.

Routesetter feedback

To test the editor functionality, two of Smíchoff's routesetters were asked to provide feedback regarding the editor usage. The feedback was positive, with the main conclusion being that the editor would be suitable for climbing gyms with lower frequency of wall changes, since the editor adds a non-trivial overhead that smaller climbing gyms could not sustain.

Bibliography

[1] International Olympic Committee. *Tokyo 2020 Sport Climbing Results*. Available on-line at: https://olympics.com/en/olympic-games/tokyo-2020/results/sport-climbing (visited on 04/16/2022).

[2] Arque Inc. *OnlineObservation*. Available on-line at: https://onlineobservation.com/en/whatis (visited on 02/19/2022).

[3] Tomáš Sláma. *Clis – The climber's scanner*. Available on-line at: https://github.com/Climber-Tools/Clis (visited on 04/22/2022).

[4] Tomáš Sláma. *Cled – The climber's editor*. Available on-line at: https://github.com/Climber-Tools/Cled (visited on 04/22/2022).

[5] Agisoft. *Agisoft Metashape*. Available on-line at: https://www.agisoft.com/ (visited on 05/10/2022).

[6] Dmitry Semyonov. *Topic: Algorithms used in Photoscan*. Available on-line at: https://www.agisoft.com/forum/index.php?topic=89.0 (visited on 02/24/2022).

[7] David G. Lowe. "Object recognition from local scale-invariant features". In: *Proceedings of the seventh IEEE international conference on computer vision*. Vol. 2. IEEE. 1999, pp. 1150–1157.

[8] David G. Lowe. "Distinctive image features from scale-invariant keypoints". In: *International journal of computer vision* 60.2 (2004), pp. 91–110.

[9] Tony Lindeberg. *Scale-space: A framework for handling image structures at multiple scales*. 1997. Available on-line at: https://www.csc.kth.se/~tony/cern-review/cern-html/ (visited on 02/25/2022).

[10] Jeffrey S. Beis and David G. Lowe. "Shape indexing using approximate nearest-neighbour search in high-dimensional spaces". In: *Proceedings of IEEE computer society conference on computer vision and pattern recognition*. IEEE. 1997, pp. 1000–1006.

[11] Bill Triggs et al. "Bundle adjustment – a modern synthesis". In: *International workshop on vision algorithms*. Springer. 1999, pp. 298–372.

[12] Shuhan Shen. “Accurate Multiple View 3D Reconstruction Using Patch-Based Stereo for Large-Scale Scenes”. In: *IEEE transactions on image processing* 22.5 (2013), pp. 1901–1914.

[13] Connelly Barnes et al. “PatchMatch: A Randomized Correspondence Algorithm for Structural Image Editing”. In: *ACM Transactions on Graphics (Proc. SIGGRAPH)* 28.3 (Aug. 2009).

[14] Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe. “Poisson surface reconstruction”. In: *Proceedings of the fourth Eurographics symposium on Geometry processing*. Vol. 7. 2006.

[15] Autodesk. *Fusion 360*. Available on-line at: `https://www.autodesk.com/products/fusion-360/overview` (visited on 05/10/2022).

[16] *gphoto2*. Available on-line at: `http://gphoto.org/` (visited on 05/09/2022).

[17] *fritzing*. Available on-line at: `https://fritzing.org/` (visited on 05/09/2022).

[18] C. T. Schneider. “3-D vermessung von Oberflächen und Bauteilen durch Photogrammetrie und Bildverarbeitung”. In: *Proc. IDENT/VISION* 91 (1991), pp. 14–17.

[19] Boris Thielbeer. “Stabilization of latex and stabilized compositions”. Pat. DE19733466A1. Aicon 3D Systems GmbH. 1997.

[20] Mathew Petroff. *Photogrammetry Targets*. 2018. Available on-line at: `https://mpetroff.net/2018/05/photogrammetry-targets/` (visited on 03/01/2022).

[21] *OpenCV*. Available on-line at: `https://opencv.org/` (visited on 05/09/2022).

[22] Satoshi Suzuki et al. “Topological structural analysis of digitized binary images by border following”. In: *Computer vision, graphics, and image processing* 30.1 (1985), pp. 32–46.

[23] Blender Foundation. *Blender*. Available on-line at: `https://www.blender.org/` (visited on 05/09/2022).

[24] David Brink. “A (probably) exact solution to the Birthday Problem”. In: *The Ramanujan Journal* 28.2 (2012), pp. 223–238.

[25] *Vim*. Available on-line at: `https://www.vim.org/` (visited on 05/09/2022).

[26] Chris Nolet. *Quick Outline*. Available on-line at: `https://assetstore.unity.com/packages/tools/particles-effects/quick-outline-115488` (visited on 04/22/2022).

[27] Dummiesman. *Runtime OBJ Importer*. Available on-line at: https://assetstore.unity.com/packages/tools/modeling/runtime-obj-importer-49547 (visited on 04/22/2022).

[28] Gökhan Gökçe. *Unity Standalone File Browser*. Available on-line at: https://github.com/gkngkc/UnityStandaloneFileBrowser (visited on 04/22/2022).

[29] *Semantic Versioning 2.0.0*. Available on-line at: https://semver.org/ (visited on 05/08/2022).

Appendix A

Clis quickstart

A.1 Setting up

For the setup, you will need to first install `pyenv`:

```
# make sure to read what the script does before running it!
curl https://pyenv.run | bash
```

and configure your shell's environment: `https://github.com/pyenv/pyenv#basic-github-checkout`. You will then need to install Python `3.7.x` (due to Metashape and Blender working only with older Python versions):

```
pyenv install -v 3.7.12
```

To set up the virtual environment, do (in this directory):

```
pyenv virtualenv 3.7.12 clis
pyenv local clis
pyenv activate clis
pyenv exec pip install -r requirements.txt
pyenv exec bpy_post_install # locates Blender and copies over
                            # some of its scripts
```

A.2 Contents

Before running the scripts (using `pyenv exec`), it is advisable to check the configuration file `config.py`, since a lot of the values will most likely differ from their default values for your particular setup. Each of the respective folders contain a `README.md` that further explains the usage of each of the scripts.

- `01-scanning/` — tools for scanning the holds.
- `02-processing/` — tools for processing the hold images into models.
- `03-data/` — hold data format specification + tools for managing the models.

A.3 Usage

A.3.1 Using tasks

To simplify the usage of Clis, various pre-programmed scripts have been added. To create a single model using them, you can do the following:

- `./scan_single.sh` to scan a single hold,
- `./move_from_camera.sh` to move the hold photos from the camera,
- `./convert_from_raw.sh` to convert the photos to a usable format,
- `./generate_models.sh` to generate the model and finally
- `./add_models.sh` to add the model to the `holds.yaml` file.

A.3.2 Using scripts

Run

```
pyenv exec python 01-scanning/01-scan.py automatic 15 \
        && pyenv exec python 01-scanning/02-copy.py camera \
        && pyenv exec python 01-scanning/03-convert.py
```

to automatically take 15 pictures of the hold, copy them from the camera and convert them from raw. If you want to take them manually because you don't have the turntable, replace `automatic` with `manual`. Then run

```
pyenv exec python 02-processing/01-model.py
```

to generate the model. Finally, run

```
pyenv exec python 03-data/01-add_models.py
```

to add the information about the hold to the `holds.yaml` dictionary.

Appendix B

Cled quickstart

B.1 Setting up

First, download the latest version from the Releases page.

To use Cled, you need a wall and holds dataset. A dataset example, along with the specification, can be found in `Models/Example`. This dataset can then be imported to Cled either by `File > New`, or by pressing `Ctrl+N`.

To generate your own dataset, use Clis [3] to generate the hold and wall models and then use `Scripts/ClisImporter.py` to prepare them for usage. Note that you will likely need to install a number of Python packages — to do this, you can follow the Clis setup on the `Scripts/requirement.txt` file (see appendix A.1).

B.2 Key bindings

B.2.1 Movement

`WSAD`	move ↑/↓/←/→
`Space`	fly upward
`Shift`	fly downward

B.2.2 UI

`Esc`	pause/cancel
`Enter`	confirm
`Q` or `Tab`	open holds menu
right button on selected route (ROUTE)	open route settings menu

B.2.3 Editing

left button	pick up/place the hovered/held hold
right button	select the hovered route
`E`	toggle between NORMAL and EDITING
`R` or `Delete`	delete hovered hold
`Ctrl+R` or `Ctrl+Delete`	delete hovered route/route of held hold
middle button (EDITING) + mouse	rotate held hold
`T/B`	toggle hovered hold as ending/starting
`Ctrl` + click (ROUTE)	toggle hovered hold as being in the route
wheel up/down	cycle filtered holds

B.2.4 Import/Export

`Ctrl+N`	open new dataset
`Ctrl+O`	open existing Cled project
`Ctrl+S`	save project
`Ctrl+Shift+S`	save project as
`Ctrl+Q`	quit

B.2.5 Capturing images

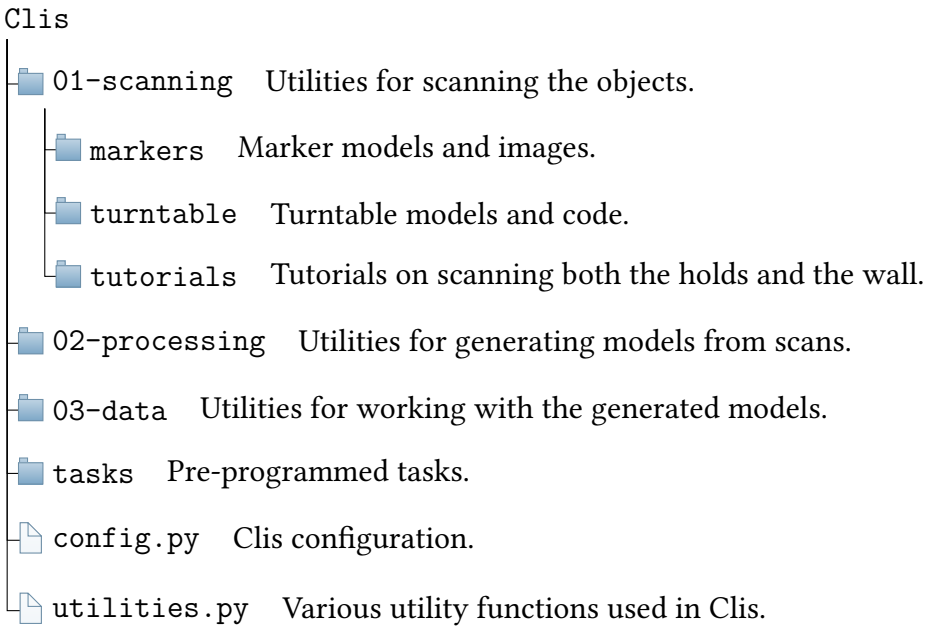
`Ctrl+P`	capture image
`Ctrl+Shift+P`	capture image as

B.2.6 Lighting

`F`	toggle user light
`Ctrl+F`	add new light at the user position

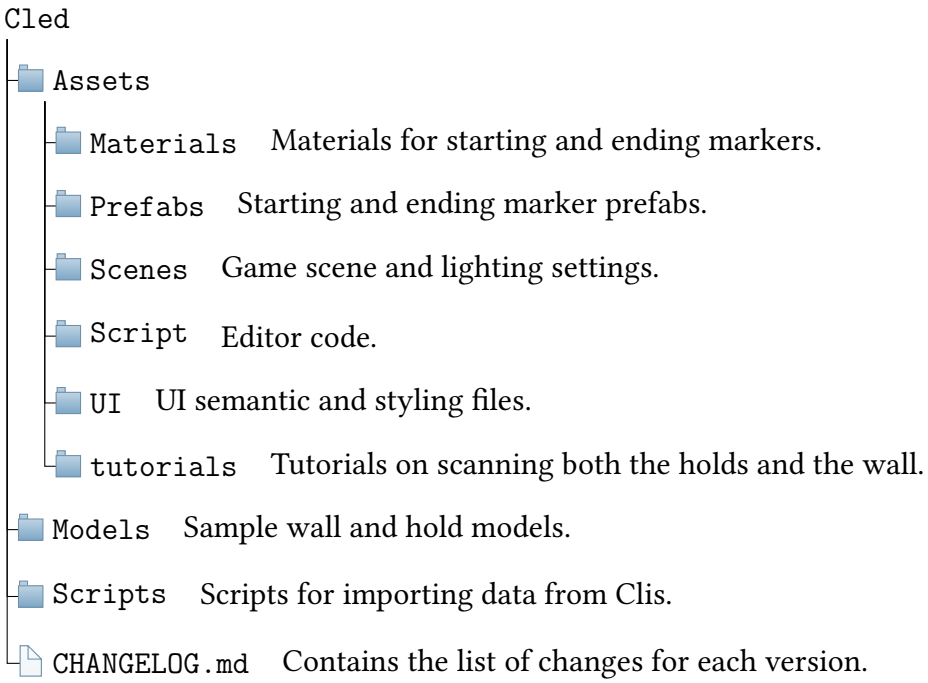
Appendix C

Clis source code



Appendix D

Cled source code



Appendix E

Cled release 2.3.1