Peakfinder

Elias Kristmann[*]

August 12, 2025

1 Problem formulation

We want to augment a mobile phone's camera view with relevant information about the real-world surroundings, such as labels of the closest peaks. To do so, we need to know the accurate pose of the camera so that we can render the correct information using Alpine Maps. However, while the GPS is quite accurate, the compass on current phones is not, making the exact orientation difficult. In addition, the projective parameters of the camera need to be known if we want the augmented information to match the corresponding real-world mountain silhouettes. As a result, this project explores the idea of matching an image to the existing Alpine Maps' 3D terrain model to estimate the orientation of the camera.

2 Camera math and pose estimation

This section summarises camera transformations that are relevant for pose estimation based on the relationship between one or multiple images. In addition, we want to render our image in Alpine Maps with the same projections as the real-world camera to minimise distortion between the image and the rendered scene and ensure objects have the same size and shape to achieve better-matching results. Therefore, we also explain the relation between real-world cameras with their physical properties, such as focal length and lens width, and the camera models used in rendering applications.

2.1 Camera matrix

The intrinsic camera matrix K maps points from 3D camera coordinates to 2D pixel coordinates and can be written as:

$$\mathbf{K} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{1}$$

[*]e-mail: elias.kristmann@tuwien.ac.at

The variables fx and fy are the focal lengths in pixels. Since a pixel does not have to be square, we can have different values for the x and y directions. Following, if the width and the height of a pixel are the same, we have $f_x = f_y$. The focal length is measured in pixels because it has the advantage that the values are invariant to the uniform scaling of the camera geometry (otherwise, we would have infinite solutions). To convert the focal length in millimetres to pixels, we need to know the camera's lens width or height [2].
c_x and c_y are the coordinates of the camera's principal point and determine the sensor's centre, which is usually half the width and height in pixels, respectively. In real-world sensors, this is prone to small deviations due to manufacturing errors and therefore varies slightly for each individual camera slightly.
If these parameters are not given, which they usually are not, or if the values in the specification are imprecise, they can be estimated by matching image points with real-world correspondences and using least square algorithms to minimise the reprojection error. This has the advantage that we get the actual values for the individual camera concerning these manufacturing errors or damages in the lens. This camera calibration is usually performed by taking multiple pictures of a chessboard pattern where we know the square size in real-world units, matching the corners of the pattern, and doing some trigonometric calculations.
In computer graphics, the camera matrix can be seen as the projection matrix. However, unlike real-world cameras that have a physical focal length affecting perspective, OpenGL's camera model simplifies this aspect to a perspective projection where all rays of light converge at a single point. Therefore, we need to modify the camera matrix so that the projection is the same for a rendered image. To convert the intrinsic camera matrix into an OpenGL projection matrix, we first express the focal lengths in pixels, f_x and f_y, and the principal point coordinates (c_x, c_y) relative to the image resolution (W, H). The OpenGL projection matrix defines a frustum in normalised device coordinates (NDC) and requires the specification of near and far clipping planes (n, f). The projection matrix is then defined as:

$$P = \begin{bmatrix} \frac{2f_x}{W} & 0 & \frac{W-2c_x}{W} & 0 \\ 0 & \frac{2f_y}{H} & \frac{2c_y-H}{H} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}. \tag{2}$$

A detailed instruction and explanation can be found at [14, 7].

2.2 Field of view

The field of view (FOV) describes the angular extent of the observable scene captured by the camera. From Figure 1, we can determine the relationship between the focal length, the sensor width, and the resulting field of view. The geometry of the camera model forms a right triangle between the centre of the lens and the edges of the image sensor. Since the angle of the field of view is symmetric on both sides of the optical axis, we can consider just half of it and apply basic trigonometry. Let w be the sensor

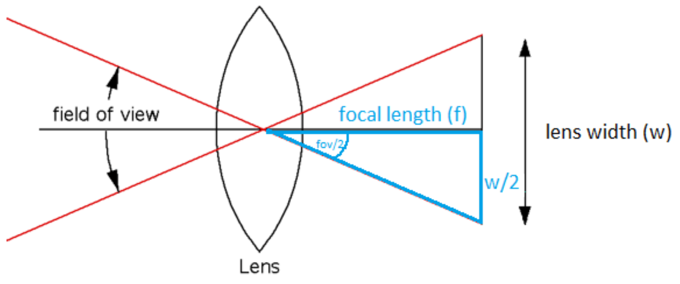

Figure 1: Illustration showing the relation between lens width, focal length and field of view.

width and f the focal length. The half-angle of the field of view, denoted as θ, satisfies the relationship:

$$\tan(\theta) = \frac{w}{2f} \tag{3}$$

Solving for θ, we get:

$$\theta = \arctan\left(\frac{w}{2f}\right) \tag{4}$$

Since the total field of view spans both sides of the optical axis, the full field of view is:

$$\text{FOV} = 2 \cdot \arctan\left(\frac{w}{2f}\right) \tag{5}$$

This formula gives the horizontal field of view if w is the width of the image sensor. A similar approach can be used to compute the vertical or diagonal FOV by replacing w with the corresponding dimension of the sensor.
The EXIF information of cameras usually provides only the focal length in millimetres. However, for the projection matrix, we need the focal length in pixels. To calculate the focal length in pixels, we need to know the sensor size of the camera, which is rarely given. One option would be to look it up in a database. Since we are more interested in images taken by mobile phones, we can also take an educated guess since the average size of the lenses is very similar, with most at 1/3" (8.46 mm), 1/3.2" (8.12 mm), or 1/1.2" (2.11 mm). The standard lens width for a non-phone camera is approximately 35.9 mm. Alternatively, we can rearrange Equation (5) to estimate the sensor width w if the FOV and focal length f are known:

$$w = 2f \cdot \tan\left(\frac{\text{FOV}}{2}\right) \tag{6}$$

Knowing the sensor size then allows us to convert the focal length from millimetres to pixels by:

$$f_x = \frac{f \cdot W}{w}, \quad f_y = \frac{f \cdot H}{h} \tag{7}$$

where W and H are the image resolution in pixels, and w and h are the sensor width and height in millimetres, respectively. There is also a second EXIF tag that states the equivalent focal length for a 35 mm lens. This is useful because we can then calculate the FOV using the standard full-frame sensor width of 35.9 mm. However, these values are only specified by the manufacturer and can vary for individual cameras and are not always present.

2.3 Essential Matrix

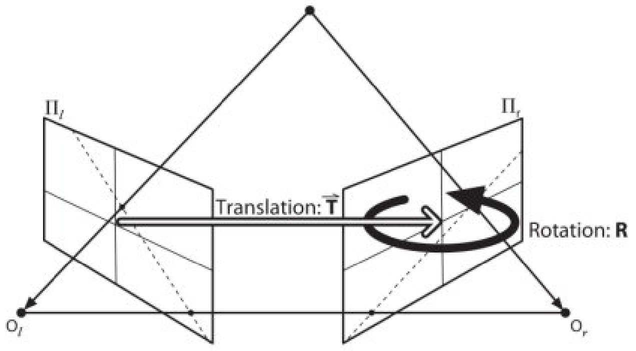

Figure 2: Relation between cameras encoded in the essential matrix.

The essential matrix is a 3×3 matrix that encapsulates the relative rotation and translation (up to scale) between two camera views in a stereo setup in camera coordinates. It plays a crucial role in understanding the epipolar geometry of two calibrated views. Figure 2 visualises the relation encoded by the essential matrix.
To estimate the essential matrix, we require at least five point correspondences between the two images (these should be reliable matches), and both cameras must be calibrated (i.e., their intrinsic parameters are known). The five-point algorithm used for this estimation is detailed by Li and Hartley [9].
Once the essential matrix E is obtained, we can recover the relative pose (rotation and translation) of the cameras. This is done by first calculating the Singular Value Decomposition (SVD) [5]:

$$E = U\Sigma V^\top \tag{8}$$

We then define a fixed matrix W as:

$$W = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{9}$$

and replace Σ from the SVD with W. Using this decomposition, the two possible rotation matrices are:

$$R_1 = UWV^\top \tag{10}$$

$$R_2 = UW^\top V^\top \tag{11}$$

The translation vector t, up to scale, is extracted from the third column of U, leading to two possible solutions:

$$t_1 = U[:,3] \tag{12}$$

$$t_2 = -U[:,3] \tag{13}$$

The translation obtained from the essential matrix does not include absolute scale—only the direction is recovered. As a result, the decomposition yields four possible relative poses (two possible rotations combined with two possible translations). To identify the correct one, a chirality check is applied. This involves triangulating a set of corresponding points and verifying that the reconstructed 3D points lie in front of both cameras (i.e., have positive depth in both views). The correct configuration is the one for which the majority of points satisfy this constraint, ensuring a physically valid and consistent reconstruction [12].

2.4 Fundamental Matrix

The 3×3 fundamental matrix F is closely related to the essential matrix E, but it operates in pixel coordinates and incorporates the intrinsic parameters of the cameras. As a result, it encodes the epipolar geometry of uncalibrated images, capturing both the relative pose (rotation and translation) and the effect of camera intrinsics. Equation 14 shows the relation between the fundamental matrix and the essential matrix. K1 and K2 are the camera matrices of the first and second cameras, respectively.

$$\mathbf{E} = \mathbf{K}_2^\top \mathbf{F} \mathbf{K}_1 \tag{14}$$

The fundamental matrix defines the epipolar geometry between two uncalibrated images. In essence, given a point in one image, it tells us the corresponding line in the second image on which the matching point must lie. This relationship can be used for image rectification, a process that warps both images so that all epipolar lines become horizontal and aligned. After rectification, stereo matching is simplified to a straightforward search along corresponding horizontal scanlines.

2.5 Homography

Homography defines a planar projective transformation that maps points from one image to another while preserving straight lines. This transformation assumes that all points lie on a single plane or that the camera undergoes pure rotation with no translation, as visible in Figure 3.
Homographies can be estimated using the direct linear transform algorithm, even when the correspondences are noisy or partially incorrect. Typically, RANSAC is applied to

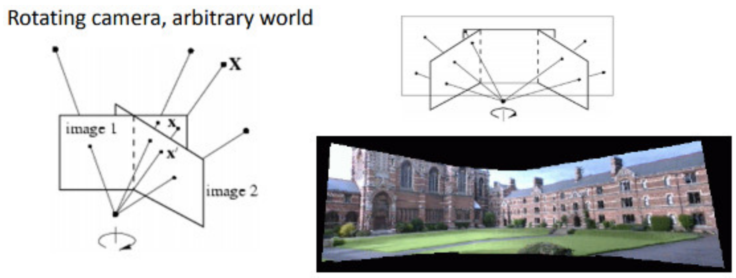

Figure 3: The camera rotates around its optical centre without translation. The points can be seen as if they were on a plane at infinity.

the points beforehand to robustly filter out outliers and compute a reliable homography.
If the two views are captured by calibrated cameras with different intrinsics, we can decompose the homography into a rotation matrix R and a translation vector t, given the camera calibration matrices K_1 and K_2. The relationship is expressed as:

$$H = K_2 \cdot \left(R + \frac{tn^\top}{d} \right) \cdot K_1^{-1} \tag{15}$$

This decomposition assumes a planar scene and allows for recovery of the relative motion between cameras under the planar constraint [10]. In the special case where the camera undergoes pure rotation, the observed transformation between the two images simplifies to

$$H = K_2 \cdot R \cdot K_1^{-1} \tag{16}$$

and the rotation can be retrieved by

$$R = K_2^{-1} \cdot H \cdot K_1 \tag{17}$$

2.6 Perspective-n-Point

The Perspective-n-Point (PnP) problem estimates the position and orientation of a calibrated camera by using correspondences between known 3D points in world space and their 2D projections in the image. At least three correspondences are needed (P3P), but having more points improves robustness. Given the intrinsic camera matrix K, PnP computes a rotation matrix R and translation vector t that relate the 3D points X_i to their 2D image points x_i via the projection equation:

$$s\mathbf{x}_i = \mathbf{K}[\mathbf{R} \mid \mathbf{t}]\mathbf{X}_i,$$

where s is a scale factor. The goal is to find R and t, minimising the reprojection error between observed and projected points. When more than the minimum number of points are available, iterative methods and robust estimators like RANSAC are used to handle noise and outliers, improving accuracy and reliability [8].

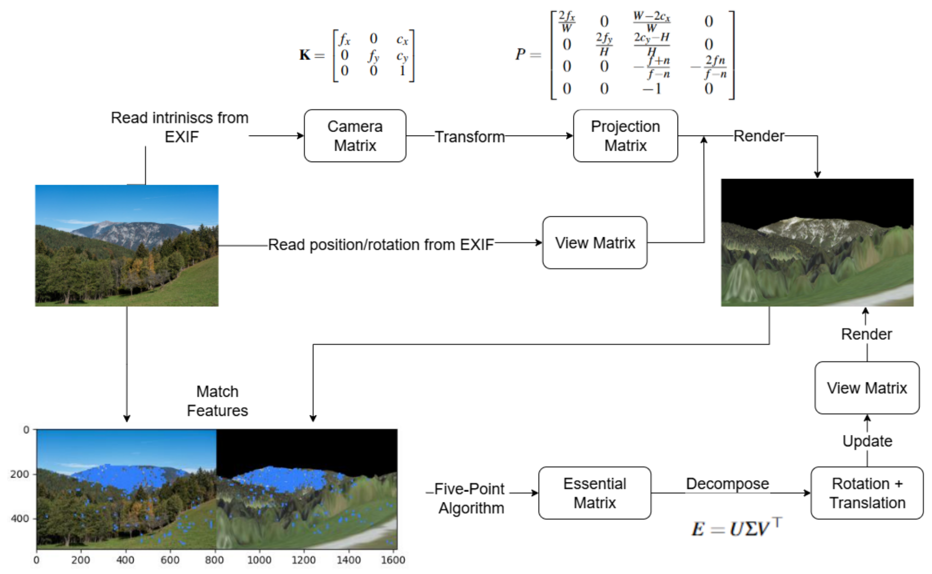
$\mathbf{K} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$
$P = \begin{bmatrix} \frac{2f_x}{W} & 0 & \frac{W-2c_x}{W} & 0 \\ 0 & \frac{2f_y}{H} & \frac{2c_y-H}{H} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$
Read intriniscs from EXIF
Camera Matrix
Transform
Projection Matrix
Render
Read position/rotation from EXIF
View Matrix
Render
View Matrix
Update
Match Features
Five-Point Algorithm
Essential Matrix
Decompose
Rotation + Translation
$E = U\Sigma V^{\top}$
0
200
400
0
200
400
600
800
1000
1200
1400
1600

Figure 4: Pipeline for camera pose estimation and refinement.

2.7 Example

Figure 4 illustrates a pipeline for estimating and refining camera pose using image EXIF metadata, feature matching, and the essential matrix. The process begins by extracting camera intrinsics from EXIF data to form the camera matrix, which maps 3D points from camera space to image space. Position and rotation from EXIF metadata are used to build the view matrix, which transforms coordinates from world space to camera space. Together with the camera matrix, these parameters define the projection matrix for rendering an initial terrain model. Features are then matched between the original photograph and the rendered model. The five-point algorithm computes the essential matrix from these matches, which is decomposed into updated rotation and translation parameters. These refined pose parameters update the view matrix, producing a more accurate rendering aligned with the scene

2.8 Summary

Method	Input Data	Assumptions	Output	Scenario	Pros	Cons
Homography	Matched points in two images (can be calibrated or not)	Scene is planar or camera only rotates	Homography matrix describing plane-induced transformation	Planar scenes or pure camera rotation; or when points lie on a plane	Robust for planar scenes; simpler estimation	Only valid for planar scenes or pure rotation; not general 3D motion
Essential Matrix	Matched points in two calibrated images	Calibrated cameras; at least 5 corresponding points	Relative rotation and translation (up to scale) between two cameras	General two-view geometry for calibrated cameras	Provides full relative pose	Requires calibration; translation up to scale; sensitive to noise
Fundamental Matrix	Matched points in two uncalibrated images	Uncalibrated cameras; at least 7 corresponding points	Epipolar geometry; constrains point correspondences	General two-view geometry with unknown intrinsics	Works without camera calibration; widely applicable	Does not directly give pose; must be decomposed with additional info; scale ambiguity
PnP	3D points in world + corresponding 2D points in one image + camera intrinsics	Known 3D points and calibrated camera	Camera pose (rotation and translation) relative to 3D points	Pose estimation for image with known 3D points	Accurate if 3D points are known	Requires known 3D points in world space

Table 1: Summary of pose estimation methods for stereo vision.

3 Implementation

The implementation can be found on GitHub at `https://github.com/elias1518693/peakfinder/`. The peak finder method is implemented in the `generate_panorama.py` Python script using OpenCV, as this facilitates prototyping and importing modern feature matching algorithms. On an NVIDIA device, the feature matching executes on the GPU. The plain renderer is run with command line parameters, and the results are read back as PNG files and from a buffer stream. The program flow is outlined in Figure 5. When starting the script, the user is asked to select an image for which the following steps will be executed.

3.1 Read EXIF and estimate field of view

The first step is to estimate the intrinsic parameters of the camera to render images with a similar field of view. If this estimation is inaccurate, the following matching will be less precise and, in the worst case, will fail to produce enough matches to estimate the pose. For all calculations and estimations, we assume that there is no optical zoom, and the input images are not cropped or altered in any way, since this would additionally complicate the process.
As mentioned in the previous section, the EXIF data does not contain the field of view or the exact sensor size and even the focal length can be noisy. A very good summary

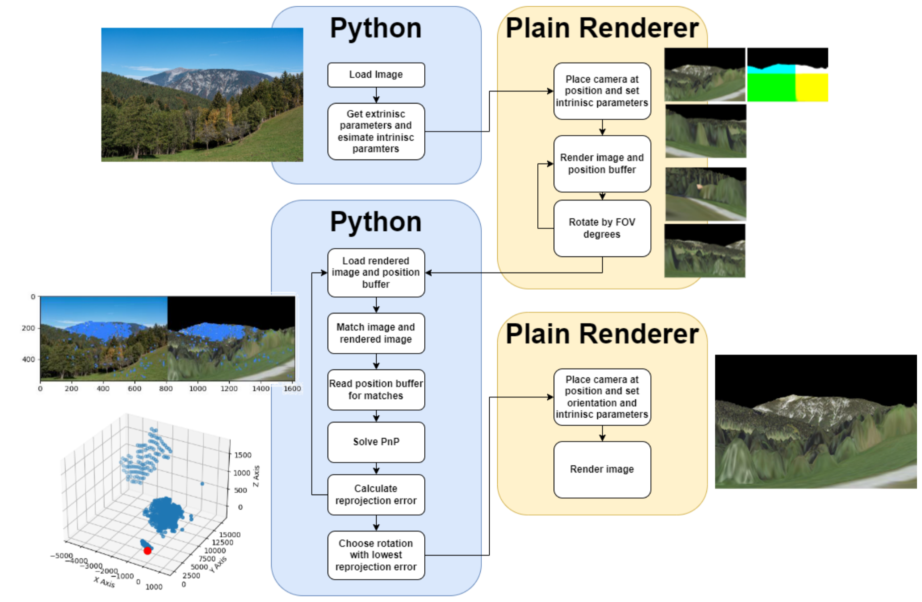
Python
Load Image
Get extrinisc parameters and esimate intrinisc paramters
Plain Renderer
Place camera at position and set intrinisc parameters
Render image and position buffer
Rotate by FOV degrees
Python
Load rendered image and position buffer
Match image and rendered image
Read position buffer for matches
Solve PnP
Calculate reprojection error
Choose rotation with lowest reprojection error
Plain Renderer
Place camera at position and set orientation and intrinisc parameters
Render image
X Axis
Y Axis
Z Axis

Figure 5: Flowchart of the application. Top left: Read EXIF and estimate the field of view. Top Right: Render images and depth map. Bottom left: Match images and pose estimation. Bottom right: Render result.

of the problem can be found at [4]. Therefore, the application tries to make the best guess using the information that is provided. In the first step, the EXIF information is analysed. If the 35mm equivalent focal length tag is present, we can use it and the 35mm sensor size to calculate a pretty good estimate. If not, the camera maker is read from the EXIF to check for keywords that could indicate the image was taken by a phone camera. If so, a typical lens width for phone cameras is chosen; otherwise, a typical lens width for standard cameras is chosen.
Optionally, we provide a script `https://github.com/elias1518693/peakfinder/blob/main/calibrate.py` to calibrate the camera and then import the JSON file containing the intrinsic parameters for more accurate results. To calibrate a camera multiple images of a 9×7 checkerboard with known square size are taken. Such a checkerboard can be downloaded from the OpenCV homepage or the pdf attached in the repository `https://github.com/opencv/opencv/blob/4.x/doc/pattern.png`. From these images, the intrinsic parameters can be calculated. If a calibrated camera is used, the Python script needs to be extended with a path to the calibration JSON file.
In addition to the intrinsic estimations, we read the latitude and longitude of the source image that needs to be provided. The altitude is not needed, as we adjust it according to the height map anyway. The image is also scaled down to be at most 960 pixels wide or 540 pixels tall while keeping the aspect ratio. This improves the later deployed image-matching algorithm immensely while still ensuring that it is producing enough matches.

3.2 Render images and depth map

The next step is to start the plain renderer application with the command line parameters calculated from the EXIF information. The location is then set accordingly, the altitude is read from the height map and the projection matrix is adjusted to mimic the input camera. The orientation is set to 0 degrees (looking north) and the width and height are set to the one of the scaled input image. Once all tiles are loaded in the application, a signal notifies the render loop and the next image that is rendered is also stored. In addition, the position buffer from the deferred rendering pipeline is written to an output stream since these are in 32-float format and can't be stored as easily from the current application setup. The position buffer contains the 3D coordinates of the point in camera space for every pixel. Once this is completed, the camera is rotated by the field of view in degrees around the up axis and the next image is rendered. When the camera has completed at least one entire 360-degree rotation the application exits.

3.3 Match images and pose estimation

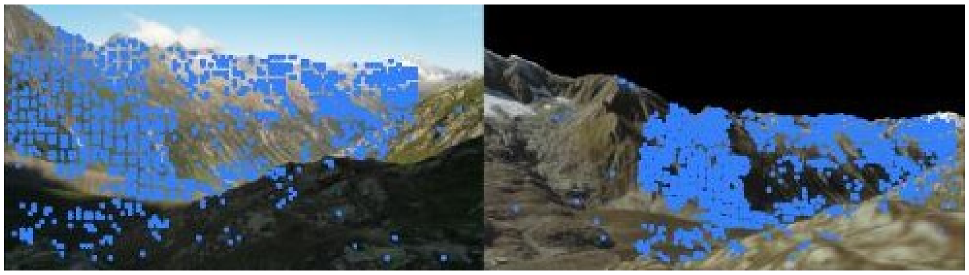

Figure 6: Keypoints found between original and rendered image.

After the plain renderer has finished, the rendered images and position buffers are read in the Python script. Following, we match every rendered image with the source image using Detector-Free Local Feature Matching with Transformers (LoFTR) [13]. The results are 2D correspondences between the two images with pairwise x and y coordinates of the feature points. If more than 6 matches are found, we read the rendered image's key point's x and y coordinates in the associated position buffer and write them to an array. We then solve the Perspective-n-Point (PnP) problem iteratively for these 2D to 3D correspondences using RANSAC to robustly handle outliers. This results in a translation and rotation vector of our camera compared to the 3D points. Since these 3D points are in the camera space of the rendered image, we get the relation of the original camera to the rendered origin. Finally, we calculate the reprojection error for each image, which is the average of the distance of each point when projected back with the calculated parameters. This reprojection error serves as a quality metric indicating how closely the estimated camera parameters align with the observed correspondences and will become important in the next step.
The PnP implementation of OpenCV sometimes gives a wrong rotation that is off by 90 degrees, for which I could not find a reason so far. Therefore, I also calculate the rotation and translation using PnP for the rendered image (which should be 0 because the

camera the image is rendered from is in the origin of this camera space) and calculate the relative rotation and translation between those and the input image's transformations to negate this effect.

3.4 Render result

We then select the rendered image that yields the lowest reprojection error from the previous section and execute the plain renderer again, but this time with the rotation of the best image as an additional parameter. We ignore the translation as the results are too inaccurate to improve the position. One reason for this could be precision errors when sampling the position buffer since the objects are so far away. After rendering the first image, we store it and quit the application. Figure 7 shows the rendered image overlayed on top of the original image.

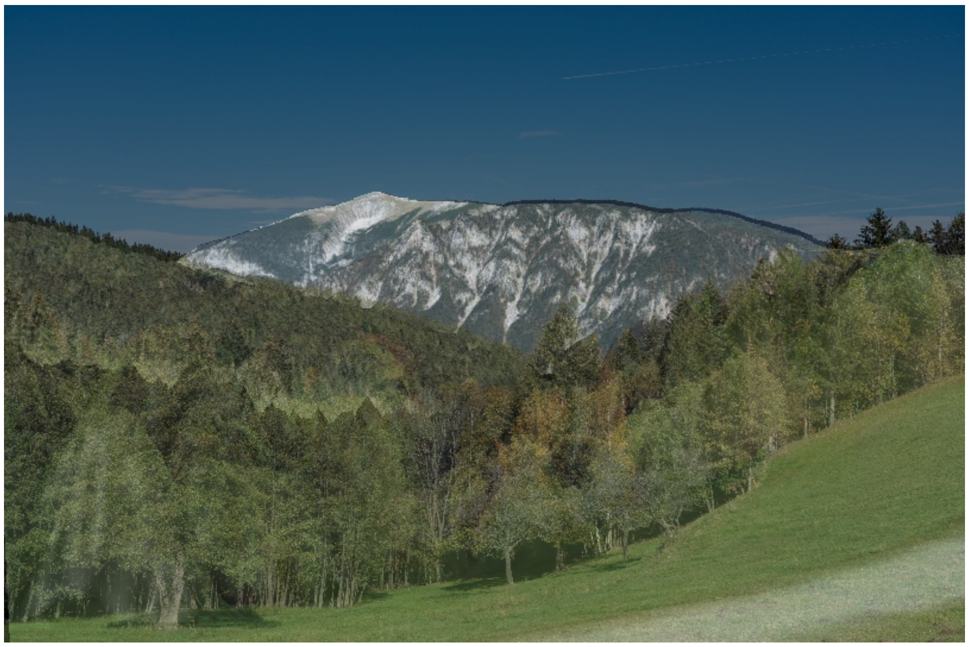

Figure 7: Original image and rendered image overlaid.

3.5 Refine with homography

Instead of solving the PnP problem, I also tried calculating the homography matrix. Since we place the camera at almost the same position, and the objects (mountains) should be decently far away, we can treat the scene as an almost planar scene. In addition, since we try to mimic the real-world camera with the rendered image, it allows us to see the homography as a rotation matrix between the two cameras. The results were decent, and the advantage of this approach is that calculating the homography is more lightweight, but PnP turned out to be more accurate and robust. However, using the homography in the last step to additionally warp the resulting rendered image can negate small inaccuracies in translation, rotation and projection estimations for a more accurate overlay. Figure 8, 9, 10 and 11 show the difference it can have on the results.

4 Failed attempts

I will briefly highlight the methods I experimented with that, unfortunately, did not yield significant improvements. This information is provided to streamline the process for future contributors to this project.
Recently, a method [6] was published that tries to estimate the intrinsic parameters from a single view using a deep network. Tests with our collected data show that the field of view is not accurate and can not outperform the manual estimates when the results are compared. However, I should mention that I was only able to test their provided online tool that is trained for cities, outdoor, and indoor scenes combined. As a consequence, a network that is specifically trained for mountainous scenes might perform better.
I tried matching with SIFT, KAZE, ORB, and other well-known handcrafted features, but in my experience, they do not work well when matching real images and rendered views and the results are too noisy and imprecise.
I also tried utilising the 3D reconstruction and structure from motion application Colmap [11], but it has difficulties finding correspondences between the rendered image and the real-world photograph. This is because it uses SIFT features that are not up to the task. I tried running the pipeline with custom LoFTR matching, but it still did not find a good image pair to start the reconstruction. This is probably due to Colmap being tuned for large-scale reconstruction and not two-view reconstruction. It is still a very useful tool if a sparse reconstruction should be planned in the future.
Approaches that use the horizon line have been the standard approach before the learned features started taking off [1, 3]. They have the advantage of being faster, but they need very specific input images to work well (the horizon line must be visible and distinct enough). In addition, many of the implementations found are not scale invariant, which makes the field of view estimation even more crucial and therefore also a more limiting factor. I tried following the implementation of Fedorov et al. [3] but did not manage to find correspondences in the horizon lines consistently. The cylindrical projection also significantly complicates the process of trying to mimic the real-world camera, in my experience.
One idea was to refine the field of view at the end by testing different FOV and minimising the reprojection error. I implemented a Nelder-Mead Simplex optimiser that solves the PnP problem and then reprojected the points for different FOV, but it always converged to a random FOV that was worse than the initial guess.

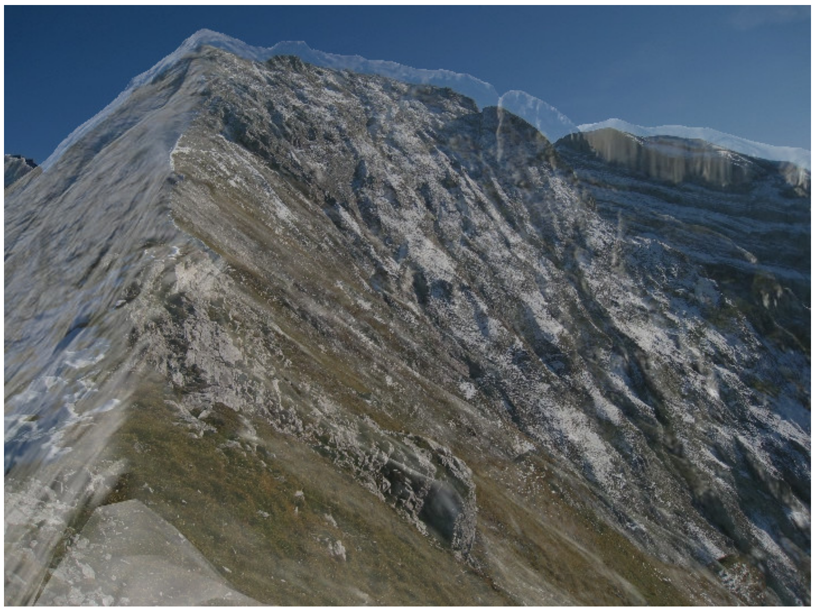
Figure 8: Without homography warping. While the general direction seems to be correct, the projection is off, making the horizon line not overlap.

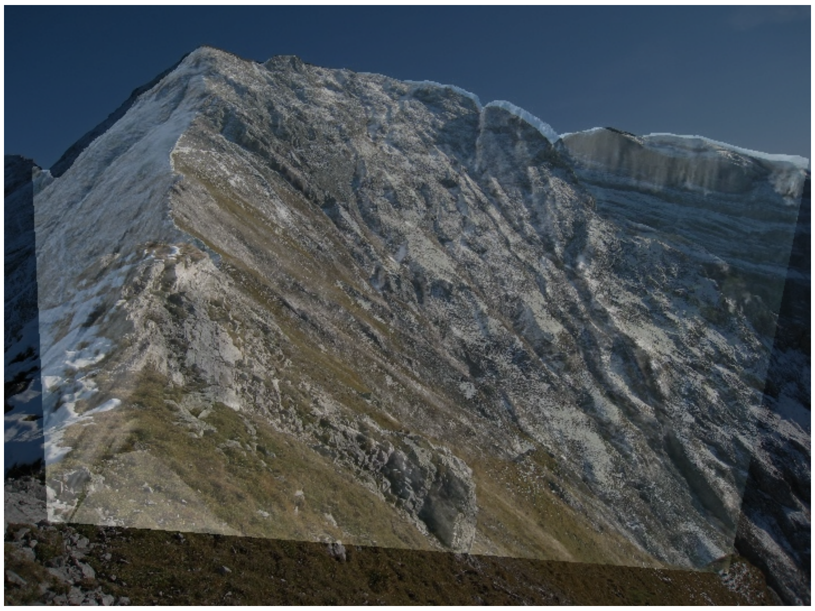
Figure 9: With homography warping, we can see the image within the image, showing that the FOV should probably be smaller.

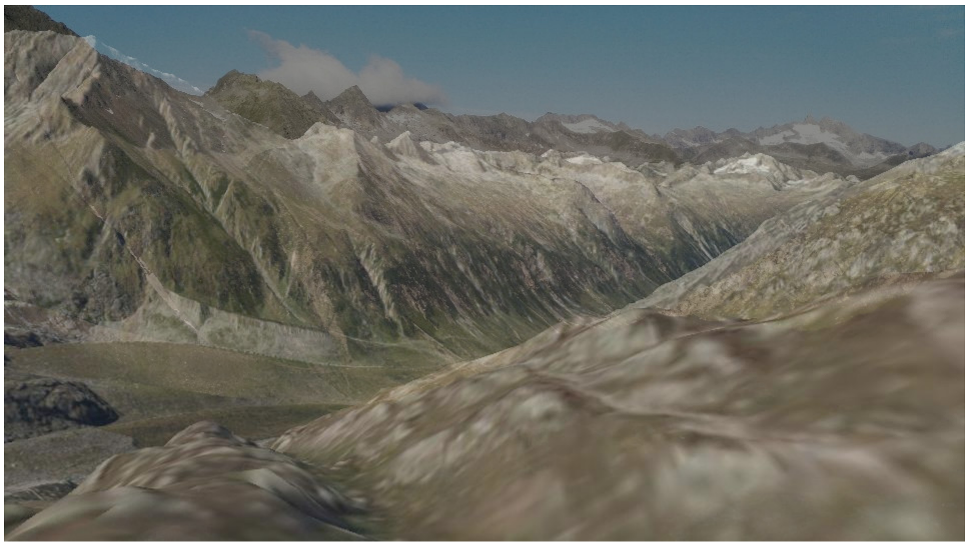

Figure 10: Without homography warping. It's decent for the ridges in the foreground, but fails completely for the horizon line.

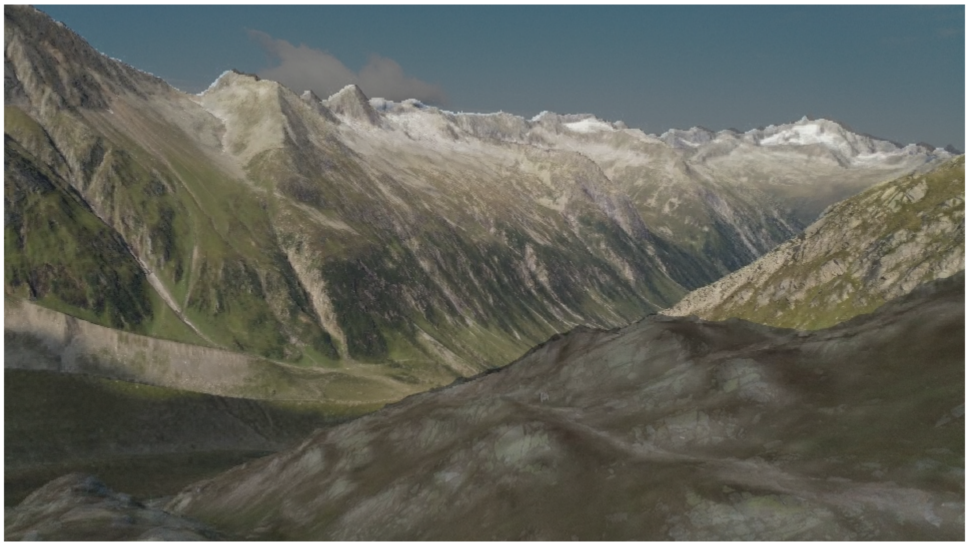

Figure 11: With homography warping, the horizon line is fixed.

References

[1] L. Baboud, M. Cadik, E. Eisemann, and H.-P Seidel. Automatic photo-to-terrain alignment for the annotation of mountain pictures. In *Proceedings of the 2011 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '11, page 41–48, USA, 2011. IEEE Computer Society.

[2] OpenCV Contributors. Opencv: Camera calibration and 3d reconstruction (calib3d module). `https://docs.opencv.org/4.x/d9/d0c/group__calib3d.html`, 2025. Accessed: 2025-08-12.

[3] Boris Fedorov, Arnaldo G O de Lacerda, Alex A J S G da Silva, Rodrigo A F de Lima, and Fabrício M V da Silva. A horizon-based method for single-image geo-localization from natural images. *arXiv preprint arXiv:1507.01185*, 2015.

[4] Wayne Fulton. Understanding sensor size in digital cameras. `https://www.scantips.com/lights/sensorsize.html`, 2024. Accessed on: January 31, 2024.

[5] Richard Hartley and Andrew Zisserman. *Multiple view geometry in computer vision*. Cambridge university press, 2003.

[6] Linyi Jin, Jianming Zhang, Yannick Hold-Geoffroy, Oliver Wang, Kevin Matzen, Matthew Sticha, and David F. Fouhey. Perspective fields for single image camera calibration, 2023.

[7] Paul Ksimek. Calibrated cameras and opengl. `https://ksimek.github.io/2013/08/13/intrinsic/`, 2013. Accessed: 2024-05-15.

[8] Vincent Lepetit, Francesc Moreno-Noguer, and Pascal Fua. Ep n p: An accurate o (n) solution to the p n p problem. *International journal of computer vision*, 81(2):155–166, 2009.

[9] Hongdong Li and Richard Hartley. Five-point motion estimation made easy. In *18th International Conference on Pattern Recognition (ICPR'06)*, volume 1, pages 630–633. IEEE, 2006.

[10] Ezio Malis and Manuel Vargas. *Deeper understanding of the homography decomposition for vision-based control*. PhD thesis, Inria, 2007.

[11] Johannes L Schönberger and Jan-Michael Frahm. Structure-from-motion revisited. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4104–4113, 2016.

[12] Chahat Deep Singh. Structure from motion tutorial. `https://cmsc426.github.io/sfm/`, 2017. Accessed: 2025-08-07.

[13] Jiaming Sun, Zehong Shen, Yuang Wang, Hujun Bao, and Xiaowei Zhou. LoFTR: Detector-free local feature matching with transformers. *CVPR*, 2021.

[14] Amy Tabb. Code for OpenCV cameras to OpenGL cameras. `https://amytabb.com/tips/tutorials/2019/06/28/OpenCV-to-OpenGL-tutorial-essentials/`, July 2 2019. Includes code and links to a supporting GitHub repository.