FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Pano.NET

An interactive application for camera calibration, image stitching and projective operations

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Florian Michael Schaukowitsch
Matrikelnummer 0927289

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Dipl.-Ing. Michael Birsak
Dr.techn. Dipl.-Mediensys.wiss. Przemyslaw Musialski

Wien, 30.08.2013

(Unterschrift Verfasser) (Unterschrift Betreuer)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Pano.NET

An interactive application for camera calibration, image stitching and projective operations

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Florian Michael Schaukowitsch
Registration Number 0927289

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Dipl.-Ing. Michael Birsak
Dr.techn. Dipl.-Mediensys.wiss. Przemyslaw Musialski

Vienna, 30.08.2013

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Florian Michael Schaukowitsch
Obere Hochstraße 1/4, 7400 Oberwart

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Abstract

The automatic generation of panoramic image mosaics has evolved into a mainstream consumer application in recent years, thanks to modern algorithms and methods that can generate beautiful-looking, artifact-free panoramas with minimal user interaction. Simple panoramas are only one possible application, other examples such as the scene-aware Google Street View show what is possible using these technologies. Software libraries such as OpenCV allow integration of image stitching and related operations into a multitude of applications with relative ease.

In this thesis, we present an interactive application (*Pano.NET*) that allows the user to experiment with several commonly used methods for camera calibration and image stitching (as implemented in OpenCV), and export related data such as registration information. We also give an overview of the used algorithms.

In addition, we implemented a projective drawing system that allows the user to draw lines and polygons onto a panorama and on the views that it consists of. Finally, a method is presented that can generate six-sided cube maps out of a spherical panorama.

Contents

CHAPTER 1

Introduction

Capturing the real world in a single photographic image taken with a standard consumer camera provides insight only to a limited "window" of information. Compare the impression you get from such an image with the experience of standing directly at the same location the photograph was taken. You have full freedom to change your personal "viewport" to focus on any part of the surrounding area, gaining vast amounts of information about the composition of the scene.

To get a similar impression from a single basic camera, you could take multiple shots from the same location, rotating the camera between shots and afterwards combine them into a panoramic image that shows much more of the scene than a single image ever could. The problem remains on how to combine these images of the single camera views into a single composite image. This process is called *image stitching*.

Research into algorithmic image stitching respectively its subparts like image registration, alignment, calibration and blending is one of the oldest topics in computer vision. Computer-aided stitching of satellite imagery with minimal seams was achieved as early as 1975 [21]. Today, image stitching for purposes of panoramic mosaic creation can be used by amateur photographers and professionals alike without requiring detailed knowledge, using digital photography software like Photoshop that provide easy-to-use instructions and interfaces [1]. In recent years, the processing capabilities of digital cameras has increased rapidly, and some consumer-grade models are able to automatically generate panoramas on-the-fly. The same holds for the increasingly popular smartphones that often boast multi-megapixel cameras [12].

Care must be taken to avoid potential artifacts like ghosting, blurring or invalid alignments, caused for example by temporal or exposure differences between each single shot.

Not only does image stitching provide an excellent all-around view in a specific scene, it also forms the basis for further computer vision tasks. New applications use panoramic data to automatically extract detailed scene information using structure from motion, or use a large amount of data for virtual navigation systems, like the *Photo-Guide* [36] project.

1.1 Goal definition

The goal for this bachelor's thesis was to implement an interactive application that allows the user to experiment with various aspects of image stitching and related tasks. We called the program *Pano.NET*, and it will serve as basis for further work (see chapter 5). The application should provide a graphical front-end to various methods already implemented in the OpenCV computer vision library [22], but also implement additional custom functionality.

The user should be able to do a number of tasks. The explanations and theoretical backgrounds of these tasks are provided in the next chapters. The tasks are as follows:

- Perform a basic single *camera calibration* by taking images of a known pattern, acquiring intrinsic camera parameters and distortion coefficients
- *Undistort* images taken by the same camera based on the acquired information from the calibration step
- Perform feature-based *image stitching*, with automatic *image registration* and with various projection and composition methods
- Simple *projective drawing*, where the user can draw primitives like lines or polygons on the panoramic image or in a single view, the program automatically drawing in the other respective image, using the panoramic projection to distort the primitives correctly
- Generate six-sided *cube maps* of a panoramic mosaic
- Experimental cube map generation of Google Street View [13] data
- *Export* related data, such as camera parameters or panorama metadata

1.2 Structure

The thesis is structured into different chapters which are as follows:

Chapter 2 : Related Work
This chapter reviews the theoretical and algorithmic backgrounds of various techniques used for camera calibration and image stitching. The principles of cube mapping are also introduced.

Chapter 3 : Implementation
The *Pano.NET* application is introduced, and it is described how it integrates the algorithms with use of the OpenCV library. Our approaches for projective drawing and cube mapping are presented.

Chapter 4 : Results
This part of the thesis contains results achieved with our implementation. Example images, basic benchmarks and comparisons between algorithms are given.

Chapter 5 : Conclusion

The conclusion summarizes the thesis and includes a discussion of possible extensions to our application.

Appendix A : Export file format examples

At the end of the thesis a section of annotated examples of our JSON data exporter is given, describing the structure of the files to make writing an importer easier.

CHAPTER 2

Related Work

2.1 The pinhole camera model

Usually, the basic transformational model used when modeling a standard camera in computer graphics is that of a simple ideal *pinhole camera*. This is of course a major simplification because real cameras are more complicated, using different lenses and aperture sizes that can cause effects like blurring of out-of-focus objects and geometric distortions.

We will now look at the properties of this model, as explained by Szeliski [33, chapter 2]. Figure 2.1 shows the basic structure of this camera model. A 3D-point p_c, which in this figure is assumed to be in the camera coordinate system that has its origin at O_c (the optical center) with the axes x_c, y_c and z_c is transformed onto the image sensor plane, usually by a projective transformation which divides the (x, y) coordinates of the 3D-point through its z coordinate. The projected point p lies on the sensor plane coordinate system which is determined by the 3D origin c_s, as well as scaling factors s_x and s_y. z_c defines the so-called *optical axis*, and its intersection with the image plane is called the *principal point* (not visible in this figure).

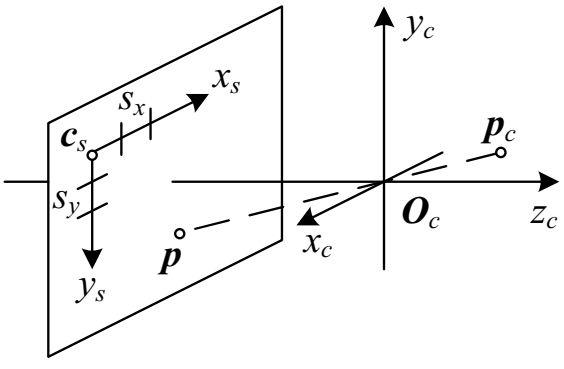


Figure 2.1: Basic pinhole camera model. Image courtesy of Szeliski [33]

To get the 3D-point from its own (world) coordinate system into the the camera coordinate system, we use a combination of rotation and translation, defined by the 3D rotation matrix R and the translation vector t. Commonly, this is written as a single combined 3×4 matrix $[R|t]$, and this matrix defines the *extrinsic camera parameters*, determining the orientation of the camera in the world space.

When we now examine the full relationship between a 3D-point p and its image projection x (using homogeneous coordinates), we arrive at the following formula [33, chapter 2]:

$$sx = K[R|t]p = s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & \gamma & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{2.1}$$

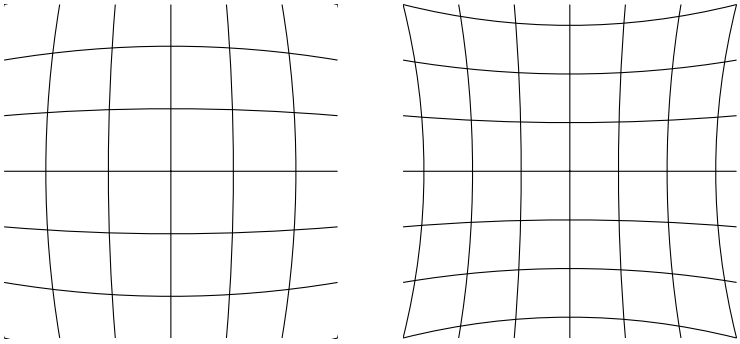
The pixel coordinates of the projection point (u, v) can be acquired when, in addition to camera rotation and translation, the *intrinsic camera parameters*, defined through the *intrinsic matrix* K are known. s is a scalar scaling factor. The combination of $K[R|t]$ is also known as the 3×4 *camera matrix*.

The definition of K combines the *focal lengths* f_x and f_y together with the *principal point* (c_x, c_y), which determines the intersection of the camera Z-axis with the viewing plane, and the skew γ between the sensor axes. The model can be simplified by assuming zero skew (usually negligible in real-world cameras). Further adaptations can be the definition of the second focal length as $f_y = \alpha f_x$, making the *aspect ratio* between the two axes explicit. Note that this ratio is not directly related to the aspect ratio of an image produced by the camera (width divided by height in pixels), but rather defines the pixel aspect ratio. Usually we can assume square pixels, so we can simplify $f_x = f_y = f$, meaning a single focal length is suitable in most cases. The principal point usually lies near the image center, so another assumption can be to divide the width and height of the image in pixels by 2 to get the principal point.

The intrinsic camera matrix does not depend on scene properties, only on internal camera properties. In practice, this means the matrix can be reused as long as parameters like output image size and focal length (changes when changing the lens, or the zoom level when using a zoom lens) stay the same.

2.2 Camera calibration

After the transformation definition in the previous section, the question remains on how to find the camera parameters of a real camera. In addition, even though the basic pinhole camera model does not model optical distortion, real cameras might have a significant amount because of complex lens configurations (especially wide-angle and zoom lenses), or for artistic effects. We might want to get information about the distortion parameters, enabling us to reduce or remove the distortions from images taken by the camera. These distortion coefficients are part of the camera intrinsics, and do not change when image size is changed. They cannot be modeled in a matrix, because the distortion transformation is nonlinear, destroying straight lines.



(a) Barrel (b) Pincushion

Figure 2.2: Lens distortion examples

Distortion

Figure 2.2 shows examples of possible lens distortion. Figure 2.2a shows a basic *Barrel* distortion: straight lines seem to bulge outwards at the center, like on a barrel. Compare this with the *Pincushion* distortion in Figure 2.2b, where points seem to be pressed inwards, the grid having some resemblance to a cushion. These two are examples of purely *radial* distortion, which means that the effect gets stronger the further a point is away from a center point (which is usually the principal point, near the center in pixel coordinates). The effect is mainly caused by the spherical shape of lenses.

Another type of distortion is *tangential* distortion. This can for example be caused by the lens not being completely parallel to the image sensor. Many lenses have some sort of radial distortion, but tangential distortion is usually much less pronounced unless the lens system has a defect, so it could be ignored in normal circumstances.

Radial and tangential distortions can be separated, and their effect can be formalized in Brown's distortion model [33, chapter 6] [5]:

$$\begin{aligned}
\hat{x} &= x + \bar{x}(1 + k_1 r^2 + k_2 r^4 + k_3 r^6 + \cdots) \\
&\quad + [p_1(r^2 + 2\bar{x}^2) + 2p_2\bar{x}\bar{y}][1 + p_3 r^2 + \cdots] \\
\hat{y} &= y + \bar{y}(1 + k_1 r^2 + k_2 r^4 + k_3 r^6 + \cdots) \\
&\quad + [2p_1\bar{x}\bar{y} + p_2(r^2 + 2\bar{y}^2)][1 + p_3 r^2 + \cdots]
\end{aligned} \tag{2.2}$$

with

$$\begin{aligned}
\bar{x} &= x - c_x \\
\bar{y} &= y - c_y \\
r^2 &= \bar{x}^2 + \bar{y}^2
\end{aligned}$$

The $(\hat{x}, \hat{y})$ represent the real observed image coordinates, and (x, y) are the ideal (distortion-free, nonobservable) coordinates.

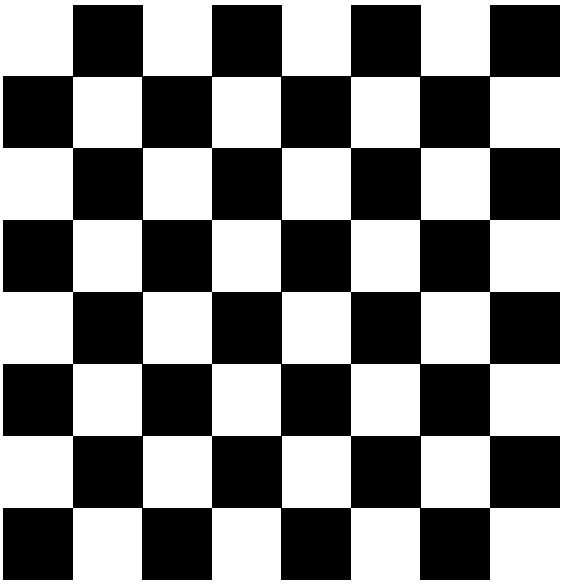


Figure 2.3: A chessboard pattern, commonly used for calibration.

Radial and tangential distortion are modeled as an infinite series, but for a suitable approximation only the first few terms are needed. Higher order coefficients do not change much in the result, OpenCV uses only 2 tangential coefficients, for example. The k_i represent the radial distortion coefficients, and the p_i are the coefficients of the tangential distortion, also called decentering distortion by Brown (because the distortion can move the apparent center point). When related to the examples in Figure 2.2, a Barrel distortion generally has a $k_1 > 0$, while a Pincushion distortion will have $k_1 < 0$.

Calibration

The calibration method used by OpenCV is based on a method described by Zhang in 2000 that requires no complicated calibration rack, only a planar pattern shown in a few different orientations [37]. This pattern can be as simple as a checkerboard (like Figure 2.3) printed on a piece of paper, and attached to a rigid cardboard to make it reasonably planar. The only requirement is that the coordinates of specific points on this pattern are known. These points can for example be corners of boxes, chessboard corners or centers of circles. For better performance, the pattern should have a reasonable amount of points.

In Zhang's method the camera is used to take several images of the calibration pattern in various orientations. This can be achieved by either fixating the camera or the calibration pattern, and then moving the other around. In each image the feature points of the pattern now have to be detected. Zhang does not propose any particular method for detection. Feature finding can be done manually, but is usually (at least for assistive purposes) done with automatic pattern recognition methods like corner detection algorithms. Methods specifically tweaked for chessboards also exist, one for example is described in a recent paper from Bennet and Lasenby [4].

We now take our camera model from the previous section. From the points detected on the image plane, and the known point locations on the pattern (assuming $Z = 0$), we can compute a *homography* H encoding their relation, where r_i denotes the i^{th} column of the rotation matrix:

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K \begin{bmatrix} r_1 & r_2 & t \end{bmatrix} \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} = H \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} \tag{2.3}$$

H is determined up to a scale factor, and can be estimated with a technique based on the maximum likelihood criterion, see the Zhang paper for details.

Given these homographies for each image, various constraints can be placed on the matrix $B = (K^{-1})^T K^{-1}$, which is actually called the *image of the absolute conic*, an important concept in projective geometry [16]. Once B is estimated, the intrinsic camera parameters can be recovered from it. Extrinsic parameters as pertaining to each view can be calculated from the inverse of the intrinsic parameters and the respective homographies from each image. Of course, if the pattern is moved instead of the camera, these parameters describe the movement of the pattern, not the orientation of the camera in space.

All this happens without considering distortion, but Zhang explains how to estimate the first two radial distortion coefficients afterwards using a linear least squares solution [37].

2.3 Panorama stitching

The problem of combining multiple images into a single mosaic can have a varying number of starting conditions. For example, in a fully calibrated environment, you may already know all your camera parameters and extrinsics, so the first steps in the following guide are irrelevant (because you already have the required information). But we will look especially at the problem of fully automatic image stitching, where the following information is not available and has to be estimated:

- Intrinsic parameters of the camera (or even multiple cameras) that has been used to take the single pictures
- Change of camera rotation and/or translation between shots
- Which images are overlapping, and how much
- Exposure differences between shots

To tackle these problems, image stitching is split into several subtasks that each solve a part of the process, allowing us to gain information about the scene without any previous knowledge other than the single image files. The following steps of a stitching pipeline are similar to the one proposed by Brown and Lowe [6], and contains the basic set pieces that are also modeled by OpenCV [25].

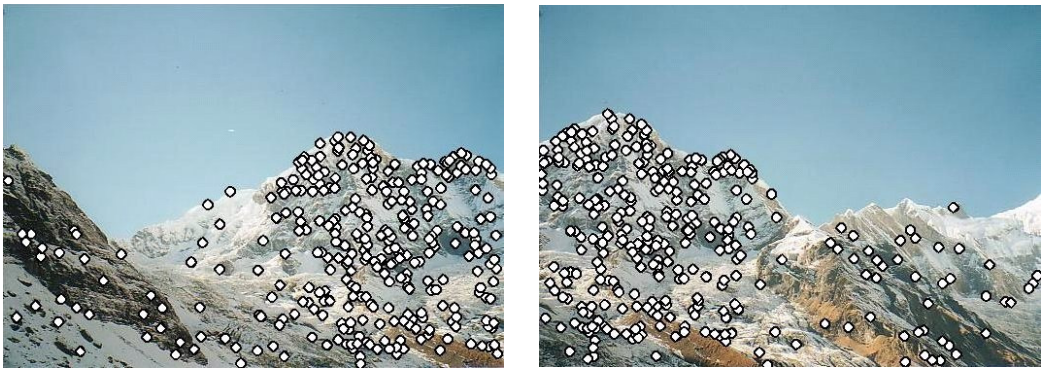


Figure 2.4: SIFT interest points in two images. Image courtesy of Brown and Lowe [6]

Image registration

Image registration aims to determine the geometric relations between the single views, by looking at correspondences of distinct *interest points*, also called *features*, and estimating the transformation matrices from them.

Feature detection

The first task aims to find distinctive features in the images that can then be compared with each other to determine the relationship between each image. Overlapping images will share an amount of features, and this information can be used to establish transformative relations.

The algorithm used for feature detection should be *invariant*, meaning that changes to the image such as transformations, rotations and scaling do not notably affect the feature descriptors. This allows us to find the same interest point in two images by comparing the descriptors, even when one image is more "zoomed" in and rotated by a specific angle, for example.

Brown and Lowe propose the *SIFT* algorithm (scale invariant feature transform) for this task [6] [18]. SIFT works by finding features that are located at the scale-space extrema of a Difference-of-Gaussian function. At each feature point, a 128-element descriptor vector is created: by computing gradient magnitude and orientation at sample points around the keypoint, combining them into a 4×4 array of orientation histograms with 8 bins each ($4 \times 4 \times 8 = 128$). This descriptor "frame" is robust against scale and rotation changes because the values are computed relative to the scaling (from the scale-space) and the primary orientation of the feature point (also computed with gradients). Figure 2.4 shows interest points found by SIFT in two images which represent part of a larger scene, partly overlapping.

Another scale- and rotation-invariant interest point detector is *SURF* (Speeded Up Robust Features) by Bay et al. [3]. This algorithm is based on the Hessian matrix, using integral images to speed up the computation. The descriptors are calculated using Haar wavelets, and can be a 64 or a 128 element vector (the larger is slower to compute, but more precise). The authors claim that SURF is not only faster to compute than SIFT (making it more usable for real-time applications, for example), but that the descriptors are more representative of the features, leading to more accuracy in further processing like object detection (or like in our case, image stitching).

A final, more recent example of a feature point detector is *ORB* (Oriented FAST and Rotated BRIEF) by Rublee et al. [28]. This method is based on the high-speed keypoint detector FAST [27], and the keypoint descriptor BRIEF [8], adding rotational invariance to these techniques. According to the authors, the ORB algorithm is an order of magnitude faster than SURF and over two orders faster than SIFT, making it especially suitable for real-time processing.

Image matching and homography estimation

After we have found interesting locations and their descriptors in each image, we can now try to find similar locations in the other pictures, calculating pairwise relationships. If we have information on which images overlap each other, we can use it in this step. If not, we have to consider each possible pair of images, leading to quickly increasing complexity with more images.

If a rotational camera model (without camera translation between the shots) is assumed, as is commonly the case, the homography H_{ij} mapping points from image j to image i can be explained as follows [6]:

$$H_{ij} = K_i R_i R_j^T K_j^{-1} \tag{2.4}$$

where K_l and R_l are the intrinsic and rotational matrices of image l, respectively. The transposed rotation matrix R_j^T does of course equal its inverse, R_j^{-1}. We now try to estimate these pairwise homographies.

First, the extracted features of each image must be matched. Brown and Lowe propose building a k-d tree to find approximate nearest neighbors [6].

The problem of quadratic complexity when having to consider each single pair of images can be made easier by only considering the m images that have the greatest number of matches with the current image, because in a realistic panorama there is only a limited number of images that overlap any given picture.

An homography between two images, when the feature matches between them are known, can be robustly estimated by the *RANSAC* (random sample consensus) method [11]. This method selects 4 random feature correspondences out of the whole correspondence set. From these 4 interest point pairs, the homography between them is found using the direct linear transformation algorithm as described by Hartley and Zisserman [16, chapter 4]. We afterwards find *inliers* by projecting the interest points of one image onto the other using the found homography, and discarding all points where the projections are more than a few pixels away. This whole process (random correspondence selection, homography calculation, inlier finding) is repeated a number of times (usual iteration counts are 500 or 1000), and the homography that produces the largest amount of inliers is used as our best estimation for the real image homography.

Figure 2.5 shows the result of RANSAC applied to the interest points in Figure 2.4. One can easily see that only the matching points remain in the inlier set. The so-found homography can then already be used to align one image to the other, as can be seen in Figure 2.6. Note how the general geometry seems to be correct, but the transitions between the images are still easily visible. These problems will be tackled in the compositing step, by using advanced seam estimation, exposure compensation and blending techniques.

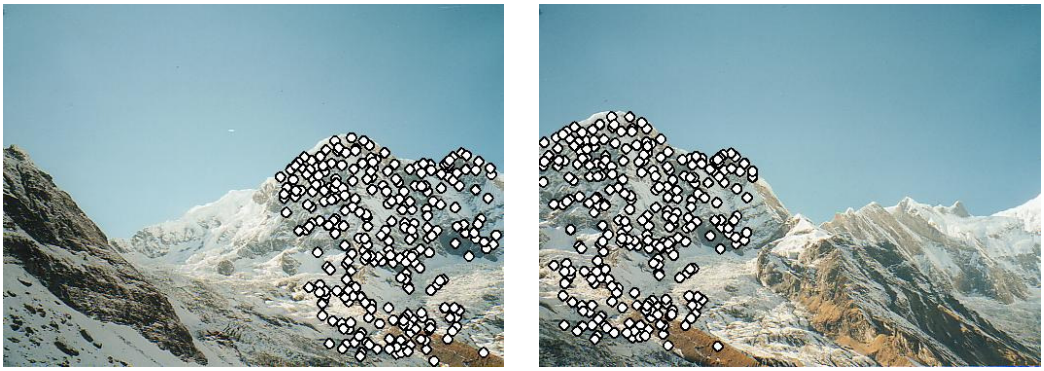


Figure 2.5: SIFT inliers after RANSAC between two images. Image courtesy of Brown and Lowe [6]

Figure 2.6: Projecting one image on the other using the found homography. Image courtesy of Brown and Lowe [6]

After RANSAC, Brown and Lowe propose a simple probabilistic model to verify the validity of the match (determining if the images really overlap each other) [6].

3D transformation estimation and bundle adjustment

OpenCV then estimates the rotation and intrinsic matrices from the homographies, by following a technique from Shum and Szeliski [29]. Because the model used is purely rotational, some simplifications can be used. The camera parameters are then refined by a *bundle adjuster*. Obtaining intrinsic estimations without any prior scene and camera information is called *autocalibration*.

Concatenation of pairwise homographies, as needed in a multi-image panorama, produces

cumulated errors and disregards constraints on images like connecting the ends of the panorama. It is thus required to solve the system considering all camera parameters jointly. The process to achieve this is called bundle adjustment. It works by adding one image at a time to a bundle adjuster system, and refining the known parameters. For the extensive details, please see the papers by Brown & Lowe and Triggs et al. [6] [34]. After this step, we have estimated the intrinsic matrix and the rotation parameters for each single view.

In addition, we may want to correct the up-vector: we expect the final panorama to be upright, and not twisted. The vertical axis in the real world should point straight up in our images. We can exploit the way the images are usually taken: the camera is seldom twisted, only tilt and horizontal rotation is done. Brown and Lowe propose a simple method that can straighten the resulting panorama [6], Szeliski also explains a similar approach [32]. This correction is sometimes called *wave correction*.

Compositing

After image registration is complete we now have information about the orientation of each camera when each view was taken. We can now proceed to combine the images to a single panorama, using various different methods.

Projection

Having the intrinsic matrix K and the rotation matrix R available, we can map the 2D coordinates (x, y) of a image to 3D coordinates (X, Y, Z).

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = R^{-1}K^{-1} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{2.5}$$

We can now map these points onto various surfaces. Any 2-dimensional surface parameterization (u, v) may be used for rendering. The most simple one is a plane, where we can simply divide by the depth Z:

$$\begin{aligned} u &= X/Z \\ v &= Y/Z \end{aligned} \tag{2.6}$$

Another common case, especially for panoramas which were created only by horizontally rotating the camera, is a cylinder mapping:

$$\begin{aligned} u &= atan2(X, Z) \\ v &= Y/\sqrt{X^2 + Z^2} \end{aligned} \tag{2.7}$$

where $atan2$ is the 2-argument $\arctan$ function, with appropriate sign handling.

When also considering vertical tilt of the camera, a better model would be a spherical projection, mapping points to latitudes and longitudes on a unit sphere:

$$
\begin{aligned}
u &= atan2(X, Z) \\
v &= \pi - \arccos(w) \\
\text{with } w &= Y/\sqrt{X^2 + Y^2 + Z^2}
\end{aligned}
\tag{2.8}
$$

The normalized coordinates are then multiplied by a scale factor s, $(u', v') = (s * u, s * v)$. This is exactly how the OpenCV stitching pipeline projects points from the single views [25, implementations of `ProjectorBase`], using the median focal length of the source views as the scaling factor. Other projections are also available in OpenCV, but omitted here for sake of brevity.

When combining the views, the minimal bounding box that includes all bounding boxes of the warped images (in (u', v')space) definies the final panorama. Because the center of the bounding box does not need to coincide with the center of the projective coordinate system, the location of the top left corner ($(0, 0)$ in panorama pixel-coordinates) in the (u', v')-space is stored for further processing.

Seam estimation

When blending multiple images, a boundary definition between them is usually required. Finding the optimal boundary that reduces visual artifacts due to registration errors or dynamic scenes (such as movement of observed objects) is non-trivial, and the task of the *seam estimation* step. This is a global optimization problem, usually minimizing a pixel-based energy function like the variation of colors or gradients across the seam.

Summa et al. explain the basic problem as follows [31]: having a collection of n single images $I_1, I_2 \dots I_n$ and the combined panorama P, the boundary problem can be defined as finding a labeling $L(p)$ for each panorama pixel p out of P ($L(p) = k$ meaning that the pixel value comes from image I_k) which minimizes an energy function based on the sum of the piecewise smoothness $E_s(p, q)$, where p, q are from the neighboring pixels.

$$
E(L) = \sum_{p,q \in Neighbors} E_s(p, q) \tag{2.9}
$$

E_s could for example describe the transition in pixel values:

$$
E_s(p, q) = \left\| I_{L(p)}(p) - I_{L(q)}(p) \right\| + \left\| I_{L(p)}(q) - I_{L(q)}(q) \right\| \tag{2.10}
$$

A common technique for solving the minimization, one that is also already implemented in OpenCV, employs *graph cuts* [17] [2] [25, `GraphCutSeamFinder`]. While the papers focus on texture synthesis and photomontage, respectively, the problem of optimal seam finding discussed is very similar to the boundary detection in panoramic mosaics.

Summa et al. observe that the graph cut algorithm has high computational cost, and that a minimal energy cut is not always visually pleasing. They propose a new technique that is based on combining results from independent pairwise boundary computations and allows interactive user input, so that the seams can be set according to the user expectation, with minimal visual

Figure 2.7: 3 different valid seam configurations in a 4-image panorama. Image courtesy of Summa et al. [31]

artifacts [31]. Figure 2.7 shows an example: three different seam configurations, but all are valid and without artifacts. Because of the temporal difference between the single shots, the persons walking around the scene are not visible in each picture. The best solution depends on what the user wants, showing or hiding the persons as needed. The labeling can be seen in the lower left corner of each image.

Exposure compensation

The next problem we look at is caused by varying exposure between the single images, as usually happens when using an automatic camera mode. This produces different brightness levels in each image, and the images do not match up correctly, as can easily be seen in Figure 2.8a. Different blending methods may help at the seams, but the overall brightness still will change too abruptly to appear natural.

Brown and Lowe formulate the problem as follows [6]: $\bar{I}_{ij}$ being the mean value of image I_i in the overlapping region of images I_i and I_j, the error function can be written as

$$e = \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} N_{ij}((g_i\bar{I}_{ij} - g_j\bar{I}_{ji})^2/\sigma_N^2 + (1-g_i)^2/\sigma_g^2) \tag{2.11}$$

(a) no exposure compensation

(b) with exposure compensation

(c) with exposure compensation and multiband blending

Figure 2.8: Multi-image panorama, various composition settings. Images courtesy of Brown and Lowe [6]

with N_{ij} being the number of pixels in image i that overlap image j. σ_N^2 represents the standard deviation of the intensity errors, and σ_g the gain standard deviation. These have been set by the authors to constant values obtained by empiric testing. The solution is now to minimize this error function in respect to the gains g_k.

Another solution by Uyttendaele et al., also available in OpenCV, is based on a block-wise division of each image. Each block is compared with the images overlaying it, producing a quadratic transfer function. Neighboring blocks are averaged, and the results are interpolated bilinearly [35].

After gain compensation, the differences between the relative brightness of each image are now much less pronounced as can be seen in Figure 2.8b. The seams are still very visible, but this can be corrected using blending methods as explained in the next section.

Blending

When creating the panoramic mosaic, the various images of the panorama will overlap each other — at least when using the automatic registration methods discussed in a previous section, where a degree of overlap is absolutely required. This causes the seams to be strongly visible. The main problem is now how to blend the overlapping images together, without introducing more artifacts like blurring or ghosting. Naive blending methods, like averaging all overlapping pixels or using the median value, provide unsatisfactory results with strong ghosting (for the average) and invalid seams (for the median), as can be seen in Figures 2.9a and 2.9b. This is caused by the scene in this picture being highly dynamic, with people moving around, and so the differences between each single image can be very large.

A simple method that provides a smoother solution, and may be suitable for some non-dynamic scenes, is *feathering*. Here, each pixel of each image gets a weight value assigned. Pixels in the center of each image are weighted heavily, at the edges low. In practice, this can be done by computing the distance transform to the edges of each image [32]. The color of pixel x can then be computed as follows:

$$C(x) = \frac{\sum_k w_k(x)\tilde{I}_k(x)}{\sum_k w_k(x)} \tag{2.12}$$

with $\tilde{I}_k(x)$ being the projection of image k, and w_k the weighting function of image k [32]. Feathering may sometimes be improved by raising the weights to some power p, meaning more focus on the larger values. This is sometimes called *p-norm blending* [32]. A result of basic feathering can be seen in Figure 2.9c.

A much more sophisticated method uses *multi-band blending*, originally by Burt and Adelson [7]. The main idea is that lower frequencies should be blended over a larger range (to achieve smoothness), and higher frequencies over short range (to prevent blurring and ghosting). It can be implemented using Laplacian and Gaussian pyramids, and is thus also sometimes called *pyramid blending* [32]. Brown and Lowe use a slight variation in methodology [6]. For ideal results, a seam estimation method should be applied before blending. Examples of this method can be seen in Figure 2.8c from the previous section, and in Figure 2.9d. The image seams have been eliminated, with only very slight blurring remaining.

(a) average

(b) median

(c) center-based feathering

(d) pyramid/multi-band (with prior seam estimation)

Figure 2.9: Multi-image panorama, various blending methods. Images courtesy of Szeliski [32]

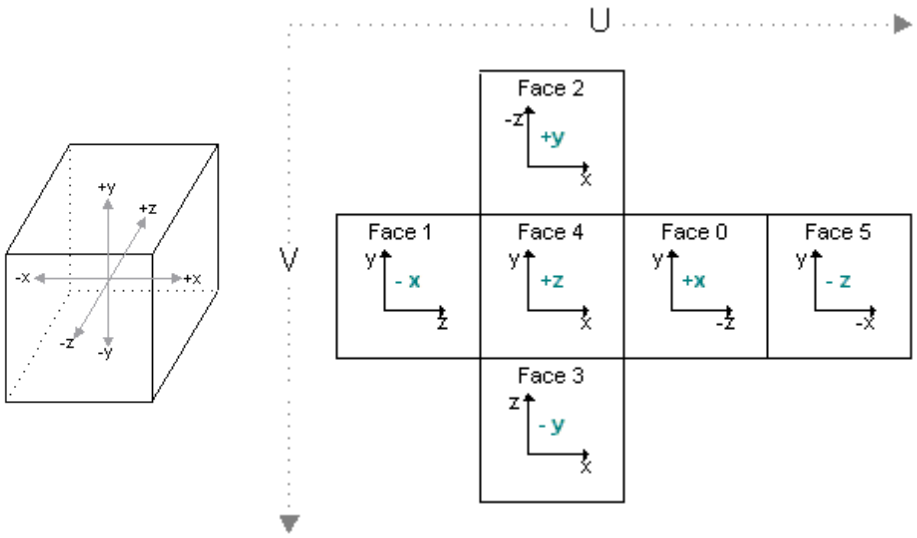


Figure 2.10: An axis-aligned cube and the unwrapped "cross" representation of its faces. Images courtesy of Microsoft [19]

2.4 Cube mapping

Cube mapping, first proposed by Greene in 1986 [15], is a common method to simulate reflections of an object, or to create so-called *skyboxes*: a way to simulate objects that are seemingly infinitely away from the viewer by using a simple texture-mapped axis-aligned cube that is always positioned at the viewpoint of the camera. Advantages of such a simple texture-based method are its simplicity and performance, especially for scenes which do not change very often, and so the cubemaps can be pre-generated. These techniques and others in context of the larger area of *environment mapping*, while only an approximation of physically correct reflections, have been useful in computer graphics (real-time and otherwise) for a long time and still continue to be, with graphics hardware supporting cube maps natively with a three-dimensional vector lookup method [19].

A cube map consists of the texture maps of its 6 faces that are usually aligned to world space axes. Assuming the up vector as Y and cameras looking in the positive Z direction (as in the Direct3D graphics API), as seen in the left image of Figure 2.10, one can denote the faces as "front" $(+Z)$, "back" $(-Z)$, "right" $(+X)$, "left" $(-X)$, "up" $(+Y)$ and "down" $(-Y)$ [19].

The texture maps on these faces may be unwrapped into a single texture often called the "cross" representation of cube maps, as seen in Figure 2.10, with a common (u, v) texture coordinate system (but it is not uncommon to address each face separately).

A simple way to generate cubemaps can be to place a camera on a point in the world space, rendering the scene using a perspective projection with $90°$ horizontal and vertical field of view (the image aspect ratio should be 1). The image plane of the camera corresponds now directly to a face of the cube map. To get the other faces, vary the horizontal and vertical rotation in increments of $90°$ to get each single face.

CHAPTER 3

Implementation

During the work on this thesis, we implemented an interactive application to achieve the goals as described in section 1.1. We called this application *Pano.NET*.

3.1 Program architecture

Most parts of the program were implemented in the C# programming language on top of the Microsoft .NET Framework in version 4.5.

The OpenCV Open Source Computer Vision Library [22] is the basis for our image processing code, and some parts of the program are not much more than easy-to-use wrappers on the massive functionality the library already provides. We used a custom-compiled development version based on a 2.4.9 snapshot at the time of the 28th Feb. 2013.

Because OpenCV is a C/C++ library which does not easily integrate into the .NET platform without extensive wrapper code, we additionally used the Emgu CV cross-platform .NET wrapper for OpenCV [10] in our C# code. Since some parts of the OpenCV API are not yet wrapped by Emgu CV (for example the stitching APIs), we implemented our own wrapper library for those parts in Microsoft C++/CLI code, which allows to mix .NET code with native code. The program was developed to run on Microsoft Windows, and can be compiled to run on the x86 as well as the x86-64 CPU architectures.

The interface of the application was implemented using the Windows Presentation Foundation interface system [20]. We used the Model-View-Viewmodel (MVVM) pattern [30] in our main C# assembly.

The project code is separated into the following libraries, which were built from scratch:

- Pano.NET — main part of the program, contains interface, viewmodels and service definitions as per the MVVM pattern
- Pano.NET.Data — contains data classes used throughout the project
- Pano.NET.Utils — various utilities related to WPF, C# and OpenCV.

- PanoNative — C++/CLI wrapper code for additional OpenCV functions not handled by Emgu CV

3.2 Camera calibration and image undistortion

The first part of the application allows the user to perform camera calibration using a chessboard pattern, to find the camera's unknown intrinsic camera parameters and radial and tangential distortion coefficients, as explained in section 2.2.

OpenCV uses a slightly modified form of Brown's distortion model, using a maximum of 6 radial coefficients (3 of which enable an optional rational model) and 2 tangential coefficients [23]. Starting from the basic pinhole projection of point $\left[\begin{smallmatrix} X \\ Y \\ Z \end{smallmatrix}\right]$, the model presents itself as follows:

$$
\begin{aligned}
&\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t, \quad x' = x/z, \quad y' = y/z && \text{projective transformation} \\
&x'' = x' \frac{1 + k_1 r^2 + k_2 r^4 + k_3 r^6}{1 + k_4 r^2 + k_5 r^4 + k_6 r^6} + 2p_1 x' y' + p_2(r^2 + 2x'^2) && \text{distortion of } x' \\
&y'' = y' \frac{1 + k_1 r^2 + k_2 r^4 + k_3 r^6}{1 + k_4 r^2 + k_5 r^4 + k_6 r^6} + p_1(r^2 + 2y'^2) + 2p_2 x' y' && \text{distortion of } y' \\
&\quad \text{where } r^2 = x'^2 + y'^2 \\
&u = f_x * x'' + c_x && \text{final image x coordinate} \\
&v = f_y * x'' + c_y && \text{final image y coordinate}
\end{aligned}
\tag{3.1}
$$

The k_i represent the 6 radial distortion coefficients supported by OpenCV, and p_1 and p_2 are the tangential distortion coefficients.

The OpenCV function `calibrateCamera` allows to estimate these distortion coefficients, in addition to the intrinsic camera parameters, using an algorithm based on Zhang's calibration as shown in section 2.2 [23].

For input, this function requires the points as found on a calibration pattern, in its own coordinate system as well as in each calibration image. In our application, we use a simple chess/checkerboard pattern of customizable size. This pattern can easily be printed out with a standard printer, and attached to a reasonably rigid surface like a larger cardboard section. The interest points are defined as the corners of the black squares of the checkerboard. We can easily determine the model coordinates of the points when applying a uniform scaling to a cartesian grid, as seen in Listing 3.1:

```
1 public MCvPoint3D32f[] Create3DPattern()
2 {
3     MCvPoint3D32f[] points = new MCvPoint3D32f[PatternSize.GetArea()
          ];
4
5     for (int i = 0; i < PatternSize.Height; ++i)
```

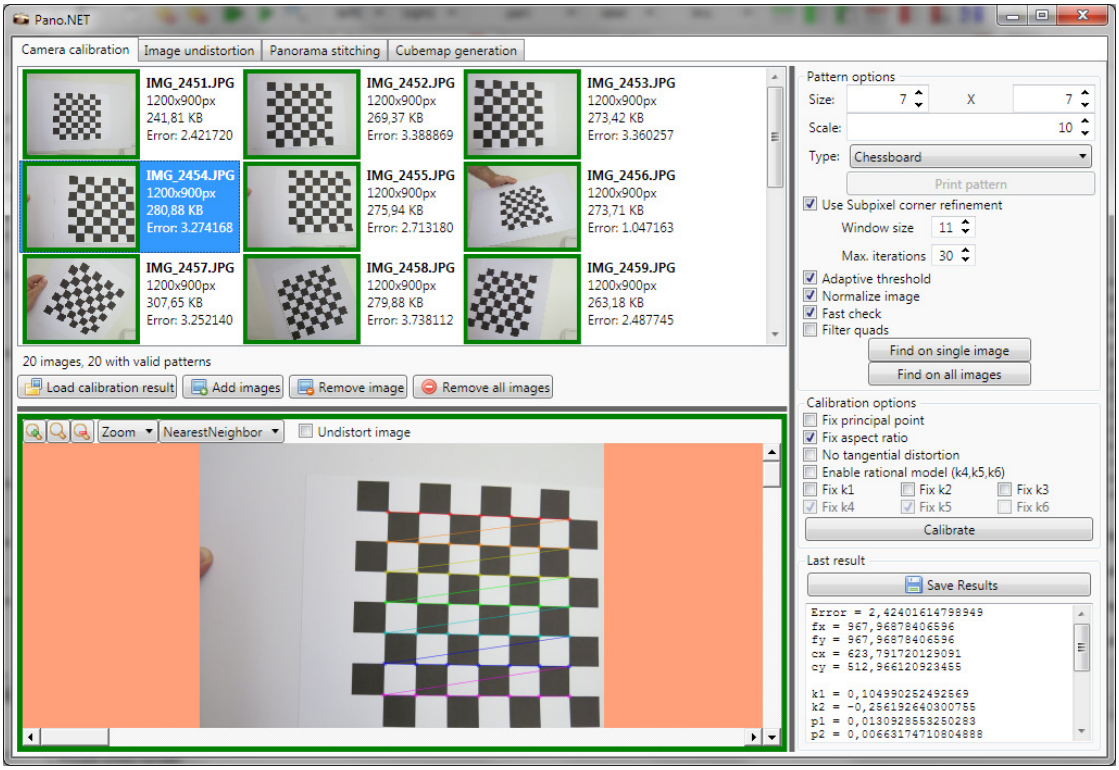


Figure 3.1: Screenshot of the calibration interface of Pano.NET, showing a successful chessboard detection and estimated camera parameters.

```
6       {
7           for (int j = 0; j < PatternSize.Width; j++)
8           {
9               int idx = i * PatternSize.Width + j;
10              points[idx] = new MCvPoint3D32f(j * PatternScaling, i *
                    PatternScaling, 0.0f);
11          }
12      }
13
14      return points;
15  }
```

Listing 3.1: Determining points of a checkerboard grid

OpenCV expects 3D-coordinates for the pattern, but for a planar pattern, the Z coordinate should be zero. The pattern size in this code listing is the number of corner points, not the number of chessboard squares (i.e. 7×7 pattern size for a 8×8 chessboard). This size, in addition to the scaling — which should be the distance between the corners in real units, like centimeters — can be configured by the user in our calibration interface.

To find the corresponding points in each calibration image, OpenCV provides a chessboard

detection algorithm with the function `findChessboardCorners` [23]. For optimal results, the chessboard should be surrounded by a white-space area. In addition, the user can further refine the found corner positions by enabling corner sub-pixel refinement in the interface, which additionally calls the `cornerSubPix` method of OpenCV, working on a sub-pixel basis to more accurately find corners.

Workflow

The calibration interface of Pano.NET, as seen in Figure 3.1, allows the user to load multiple calibration images at once. After configuring the pattern size and scaling, the user may select various settings exposed by OpenCV to aid the automatic corner detection:

- enable sub-pixel corner refinement, as described above, with variable window/iteration settings
- enable adaptive thresholding (use average brightness for binary separation of black/white areas, helpful in some cases, usually not needed and can slow down the processing tremendously in larger images)
- normalize the image by histogram equalization
- fast check for invalid images (determines if the image contains a checkerboard with a high probability with a simple algorithm, the real algorithm may take a long time to fail in invalid images). More useful for real-time applications.
- apply additional quad filtering, useful when wrong squares are detected

The application can then automatically find the corners in one image at a time, or all images simultaneously. Valid images, where all corners have been found, are indicated by a green frame. One can view the detected chessboard corners by clicking on such a green image. Images, in which the detection failed for any reason (most probable cause is that a corner is outside the image bounds), are marked with a red frame.

After some valid calibration images (where the corners have been found) are available, the calibration can be started. The user can choose from various options in this step, such as:

- fixing the principal point in the center of the image ($c_x = \text{Width in pixels}/2$, $c_y = \text{Height in pixels}/2$)
- fixing the aspect ratio ($f_x = f_y = f$)
- disable tangential distortion (setting $p_1 = p_2 = 0$)
- enable/disable the rational radial distortion model (k_4, k_5, k_6 available, as in Equation 3.1)
- ignoring certain radial distortion coefficients by fixing them at zero

After a successful calibration, the results are shown in the bottom right text box. It contains the intrinsic camera parameters (f_x, f_y, c_x, c_y) and the distortion coefficients. In addition, for each calibration image, the extrinsic parameters have been calculated. These are not shown directly in the interface, but can be exported, see the next section. This information can be used to calculate the reprojection error for each single image (shown near each calibration image after calibration), as well as the total average reprojection error (in the text box).

Export

The application can export the calculated camera parameters, together with the used calibration options, calibration image information with extrinsic matrices, reprojection errors and detected interest points. Export can be done in a custom JSON [9] or XML based format. A commented example of an exported file can be found in appendix section A.1. Exported JSON files can also be imported again into the application, restoring the full session with all calibration images and the calibration results.

Undistortion

After the intrinsic matrix and the distortion parameters have been estimated, one can undistort images taken with the camera to remove the radial and tangential distortion. This can be done with help from lookup maps, which encode the relation between pixels in a source image and in the destination image. Using two maps for x and y coordinates, respectively, one can write the lookup process as follows:

$$dst(x, y) = src(map_x(x, y), map_y(x, y)) \tag{3.2}$$

Pixels with non-integer coordinates can be interpolated. OpenCV provides this map-based transformation with its `remap` function [24]. The function `initUndistortRectifyMap` is used to generate these maps [24]. Apart from undistortion, this method can also apply a camera matrix change. This may be desired because undistortion usually introduces invalid areas in the image, or causes parts of the image to be out of bounds (for example the corners when correcting a strong barrel distortion). For this purpose, OpenCV provides the `getOptimalNewCameraMatrix` function [23]. It provides a scaling parameter α that allows to exclude all invalid pixels (when set to 0, possibly excluding some valid pixels as well), definitely retain all source pixels (when set to 1, may result in large amounts of invalid pixels) or anything in between.

The interface, seen in Figure 3.2, allows to load multiple images at once. Calibration information can either be copied from the active calibration in the calibration tab, or imported using the JSON file format. The user can optionally adjust the camera matrix as described before, the final output size can also be changed. A preview of the current undistortion settings is updated each time the user makes a change. All loaded images can be processed at once and stored into a destination folder.

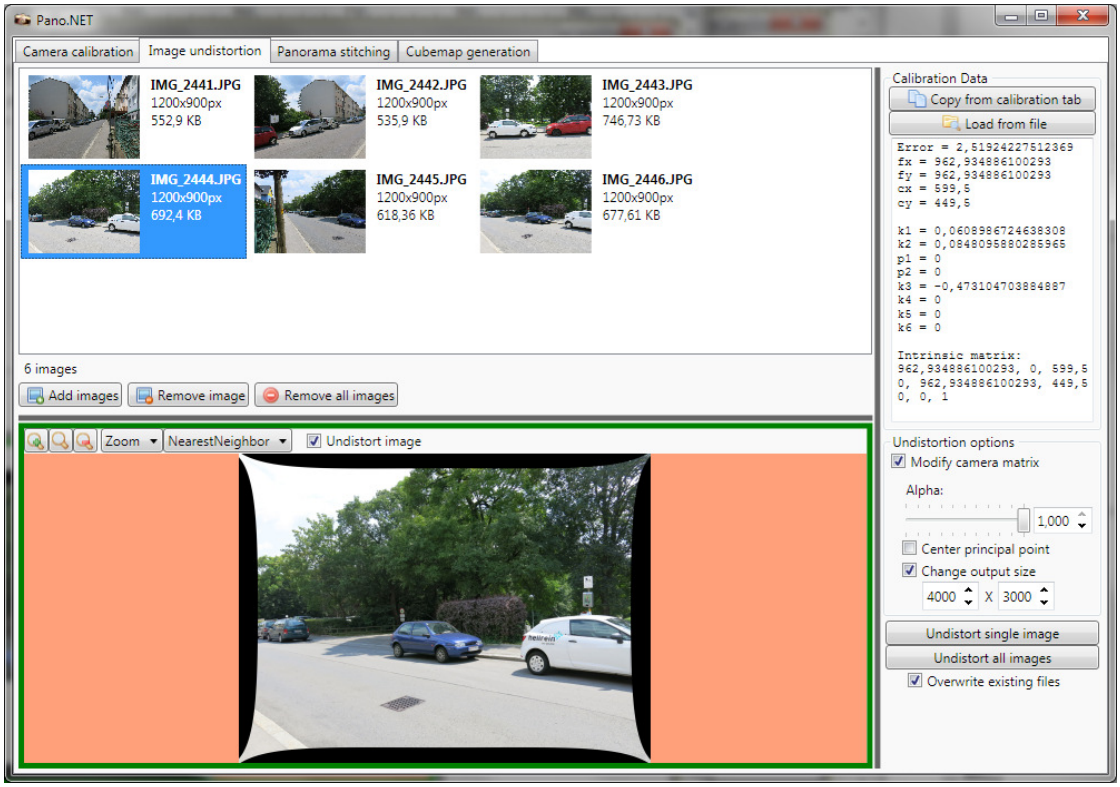


Figure 3.2: Screenshot of the undistortion interface of Pano.NET, ready for batch processing.

3.3 Image stitching

The stitching pipeline included in OpenCV is similar to the one proposed by Brown and Lowe, with some additions like seam estimation algorithms [25] [6]. The overall structure can be seen in Figure 3.3.

For some steps of the pipeline OpenCV includes more than one algorithm, from which the user in our application can freely choose. Some of these algorithms are described in section 2.3:

- Feature detection — SURF (`SurfFeaturesFinder`, [3]) and ORB (`OrbFeaturesFinder`, [28])
- Warping/Projection — plane, cylinder, sphere, fisheye, stereographic, (transverse) Mercator, custom Panini and rectilinear projections (all implementations of `RotationWarper`)
- Seam estimation — Voronoi diagram-based (`VoronoiSeamFinder`), graph-cut based (`GraphCutSeamFinder`, [17])
- Exposure compensation — blockwise (`BlocksGainCompensator`, [35]), average (like Brown and Lowe [6], `GainCompensator`)
- Blending — feathering (`FeatherBlender`), multiband (`MultiBandBlender`, [7])

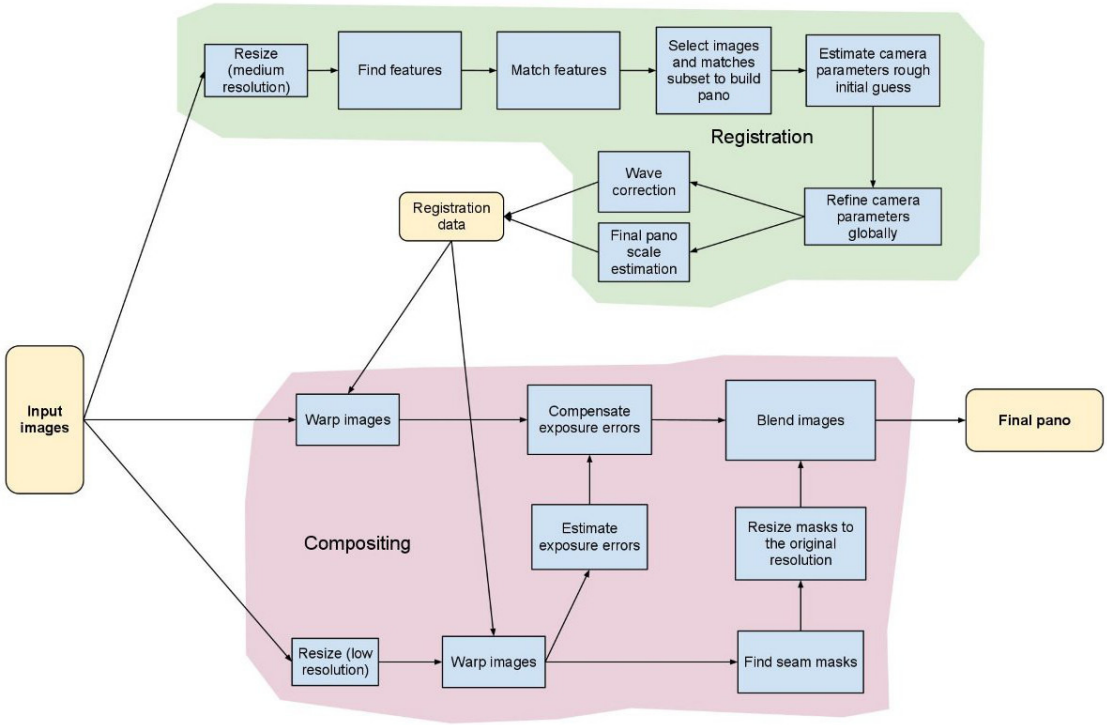


Figure 3.3: OpenCV stitching pipeline. Image courtesy OpenCV [25]

We recreated the pipeline as implemented in the high-level `Stitcher` class [25] using these building blocks, with some minor modifications to primarily improve support for the two-step approach of registration and compositing (with the user able to composite the same scene multiple times, without re-registration) and the access to registration and projection data, needed for our own extensions as described in the next sections.

The user interface, seen in Figure 3.4, is structured after this two-step model. We also support undistortion of source images as a pre-process, using data obtained as described in section 3.2. The user first has to perform image registration by clicking on "Match images". The given images are rescaled to a smaller registration resolution, and keypoints are detected using the detector selected. These keypoints are also shown in the interface when clicking on an image after registration. Images not deemed to be part of the panorama get a red border, the composition of the remaining (green) images is still possible.

After the registration step, the intrinsic and extrinsic parameters of each view are known, and the composition can be repeated multiple times. For seam estimation and exposure compensation, even smaller versions (resolution can be tweaked) of the images are used for better performance. The final composite is also created from scaled copies, but one may also use the full source resolution if required. This may use much more memory though, as described in chapter 4.

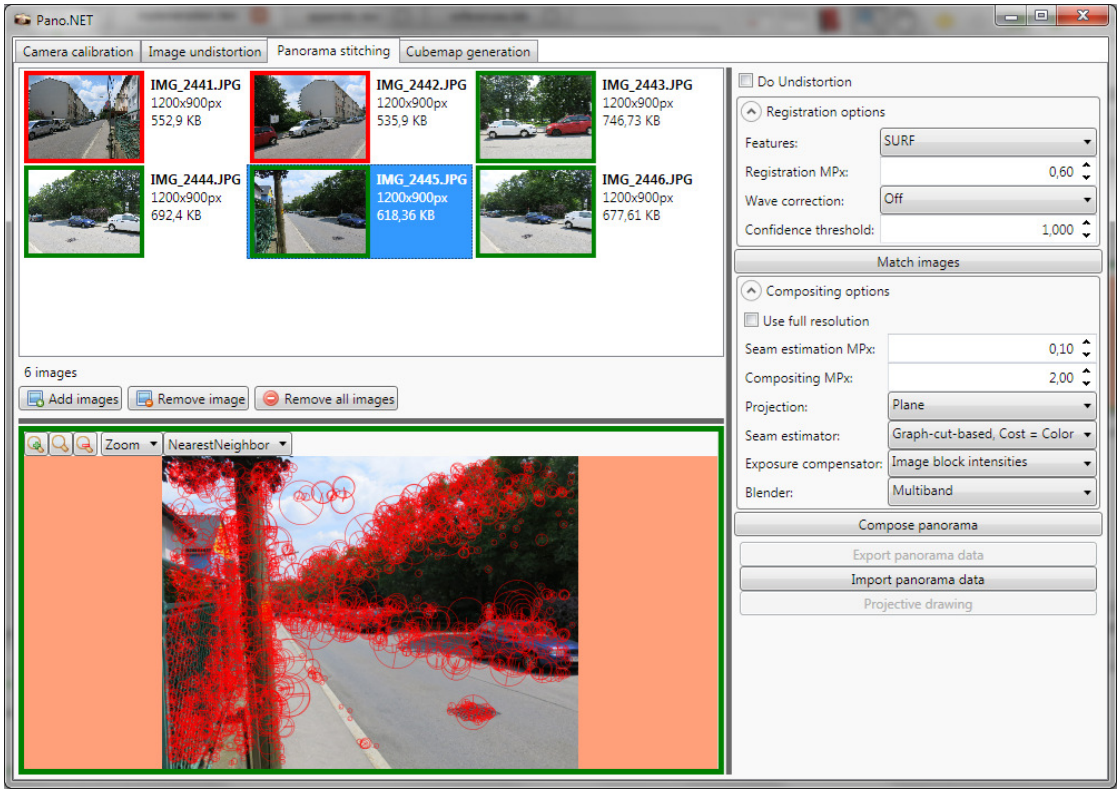


Figure 3.4: Screenshot of the stitching interface of Pano.NET, showing some registered images, two where the registration failed, and detected interest points of one image.

Export

The metadata associated with the last created panoramic mosaic can be exported after composition. It includes information on the projection used (type, scaling, coordinates of top-left corner in projective space). This information is needed for our cubemap generation method described in section 3.5. Furthermore, it includes the registration information of the source images (intrinsic and rotational matrices), which is additionally required for our projective drawing process from section 3.4. Interest points in the images are not exported for simplification reasons. The user can also re-import the panorama data to reuse the source images and their calibration in the stitching window. An annotated example of an exported metadata file can be found in appendix section A.2.

3.4 Projective drawing

After we have created the composite image, we can retain all information needed to transform points from a source view to the panoramic mosaic. This includes the type of the projection, the scaling of the source images, the scaling of the projective (u, v) coordinate system and the coordinates of the top-left corner of the final mosaic in this projective coordinate system. Using

the projection equations as described in section 2.3, one can project points from the coordinate system of a single view to the final panorama. Similarly, one can use an inverse transformation to project points from the panorama to a single view. As an example, an inverse to the spherical projection in section 2.3 is given below, transforming points (u, v) from the panorama to a source view (x, y) with given parameters K, R:

$$\begin{aligned} h &= \sin(\pi - v) \\ X &= h * \sin(u) \\ Y &= \cos(\pi - v) \\ Z &= h * \cos(u) \\ \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} &= KR \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \\ x &= x'/z', y = y'/z' \end{aligned} \tag{3.3}$$

Given these considerations, we implemented a basic drawing system that allows the user to draw points, lines and polygons into the source view or the panoramic view. The program automatically creates a copy of the primitive in the respective other view, correctly applying the projection to distort the drawing so that the same corresponding area is covered.

For point drawing, the transformation is trivial: its coordinates can be transformed as described above. To implement line and polygon drawing, the user can set control points on the images. In the default mode, it is assumed that a line between these control points is always straight in the currently selected single view image. We can now simply interpolate points between 2 control points, and transform these intermediate points to the panorama. If the user places the control points on the panoramic view, one can first transform them into the single view coordinate system, interpolate points in-between and transform them back to the panoramic system. In an alternative mode that can be enabled by checking "Draw straight lines in panorama", the reverse is assumed: lines between control points in the panorama are straight, we can interpolate in-between and then transform back to the single view. To get a line in each single view, we have to transform the panoramic interpolation points backwards with each respective view transformation (using their K and R matrices). We directly use the projection classes of OpenCV to achieve all this functionality, for each projection natively supported by their stitching pipeline [25, `ProjectorBase` implementations].

We use a simple way to determine the amount of interpolation points needed for a smooth representation: in addition to transforming the start/end control points of a line to the panorama, we transform the point that lies in the middle, enabling us to very roughly consider stronger distortion. The sum of the length in pixels of the start/middle and middle/end vectors is scaled by a constant factor like 0.05 to get the needed amount of interpolation points.

The user interface, shown in Figure 3.5, allows the user to select a source view, which is used for the calculations described above. He can choose from points, lines and polygons for drawing. In line and polygon drawing mode, each click on the image adds another control point, enabling the user to draw multiple line segments in one go. Right-clicking ends the drawing,

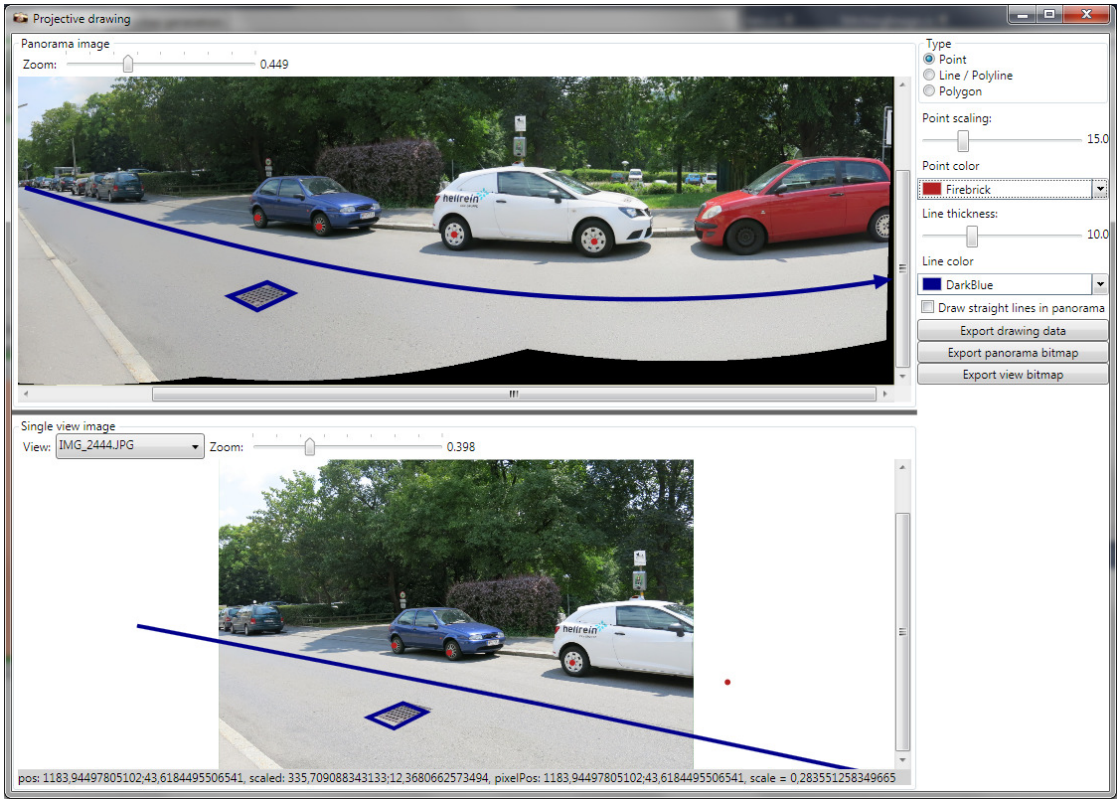


Figure 3.5: Screenshot of the projective drawing interface of Pano.NET, showing a 4-image panorama with a warped line, a basic polygon and points (on the car wheels) that correspond in a single view and the panorama.

closing the current polygon if in polygon mode. One can choose the scaling and the color of the drawn primitives. In line drawing mode, configurable arrow heads can be added to the end of the lines. A drawn object can be deleted again by right-clicking on it.

The user can export the panorama or the current single view as a bitmap file. In addition, the drawn primitives can be exported in a JSON format, that also contains all interpolation points calculated. An example can be found in appendix section A.3.

3.5 Cube map generation

When transforming the parametric (u, v) representation of a spherical panorama back into 3D-space, using the first steps of Equation 3.3 to obtain $(X, Y, Z)^T$, one gets an representation similar to Figure 3.6. We can now render the resulting sphere (or rather the parts of it that corresponds to the panorama) onto the faces of a cube map. We will consider a freely selectable azimuth angle θ that determines the horizontal rotation (on the X/Z plane) of the cube.

When assuming that the positive Z direction defines "front", Figure 3.7 shows three different cubemap faces: the front, the right and the bottom faces as they appear when projected onto the

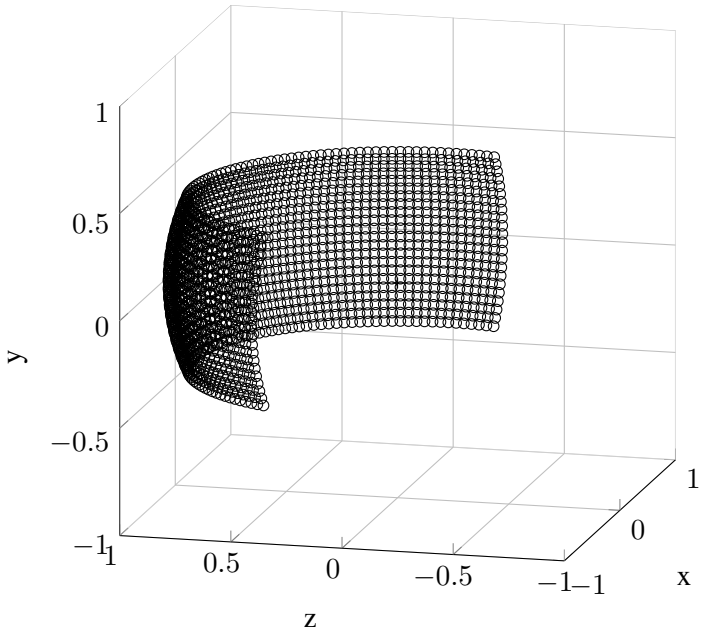
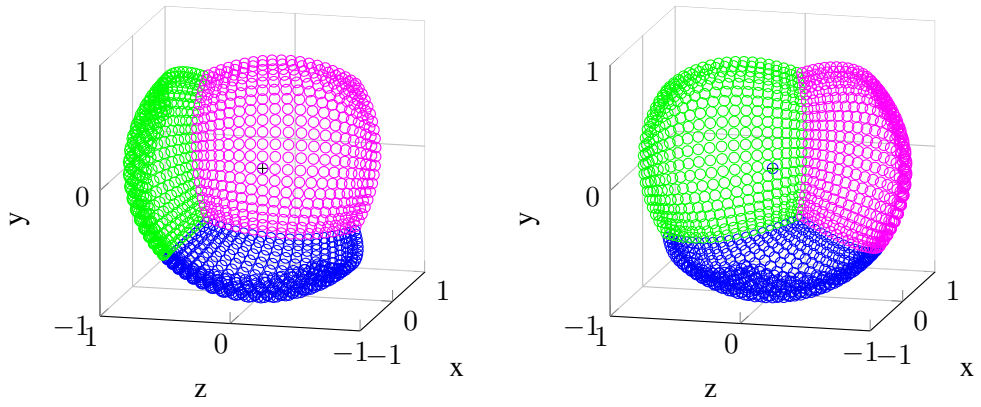


Figure 3.6: A spherical panorama, displaying about 180° horizontal FOV, but limited vertical range



(a) $\theta = 0°$ (b) $\theta = 45°$

Figure 3.7: Front (green), right (magenta) and bottom (blue) cubemap faces projected on the sphere

unit sphere of the panoramic coordinates. When using an angle $\theta \neq 0$, the whole cube/sphere gets rotated in the X/Z plane as seen in the right figure. We can look at the problem backwards: starting at a specific pixel in a face of the cube map, how do we find the corresponding pixel position in the panorama image? This way, we can build transformation maps to be used in `remap`, as described in section 3.2. Starting with the front face, we see that we can get all other

faces by simply rotating it in steps of 90 degrees in the azimuth and elevation angles, so we can begin by determining the coordinates of points on this sphere segment. Assume a square cube face of pixel size $S \times S$ with coordinates (u, v) starting with $(0, 0)$ in the top left corner. First, we convert to normalized coordinates, using the pixel centers:

$$\begin{aligned} u' &= ((u + 0.5)/S) * 2 - 1 \\ v' &= ((v + 0.5)/S) * 2 - 1 \end{aligned} \tag{3.4}$$

Next, we calculate the required angles:

$$\begin{aligned} \lambda &= atan2(u', 1) \\ \phi &= \arcsin(\frac{v'}{\sqrt{1 + u'^2 + v'^2}}) + \frac{\pi}{2} \end{aligned} \tag{3.5}$$

From this, we can get the 3D position by using the reverse mapping described in section 3.4:

$$\begin{aligned} h &= \sin(\pi - \phi) \\ X &= h * \sin(\lambda) \\ Y &= \cos(\pi - \phi) \\ Z &= h * \cos(\lambda) \end{aligned} \tag{3.6}$$

If $\theta = 0$, this is already a correct coordinate for a front face pixel. To get the other faces (and account for θ), we perform one or two 3D rotations. The top and bottom faces need a rotation around the X-axis first ($\alpha = \pm\frac{\pi}{2}$):

$$\begin{aligned} Y' &= \cos(\alpha) * Y - \sin(\alpha) * Z \\ Z' &= \sin(\alpha) * Y + \cos(\alpha) * Z \end{aligned} \tag{3.7}$$

We do the Y rotation afterwards, with β being zero for front, top and bottom, and $(90°, 180°, 270°)$ for the (right, back, left) faces.

$$\begin{aligned} \gamma &= \beta + \theta \\ X' &= \cos(\gamma) * X + \sin(\gamma) * Z' \\ Z'' &= -\sin(\gamma) * X + \cos(\gamma) * Z' \end{aligned} \tag{3.8}$$

The point (X', Y', Z'') can now be projected back onto the panorama pixel coordinates (from which scaling factor S and top-left corner coordinate p are known).

$$u_P = atan2(X', Z'') * S - p_x$$
$$v_P = (\pi - \arccos(\frac{Y'}{X'^2 + Y'^2 + Z''^2})) * S - p_y \tag{3.9}$$

We now have a mapping from each (u, v) position on each cube face to the corresponding pixel position (u_P, v_P) in the spherical panorama, and can now fill the transformation maps for `remap`.

Interface

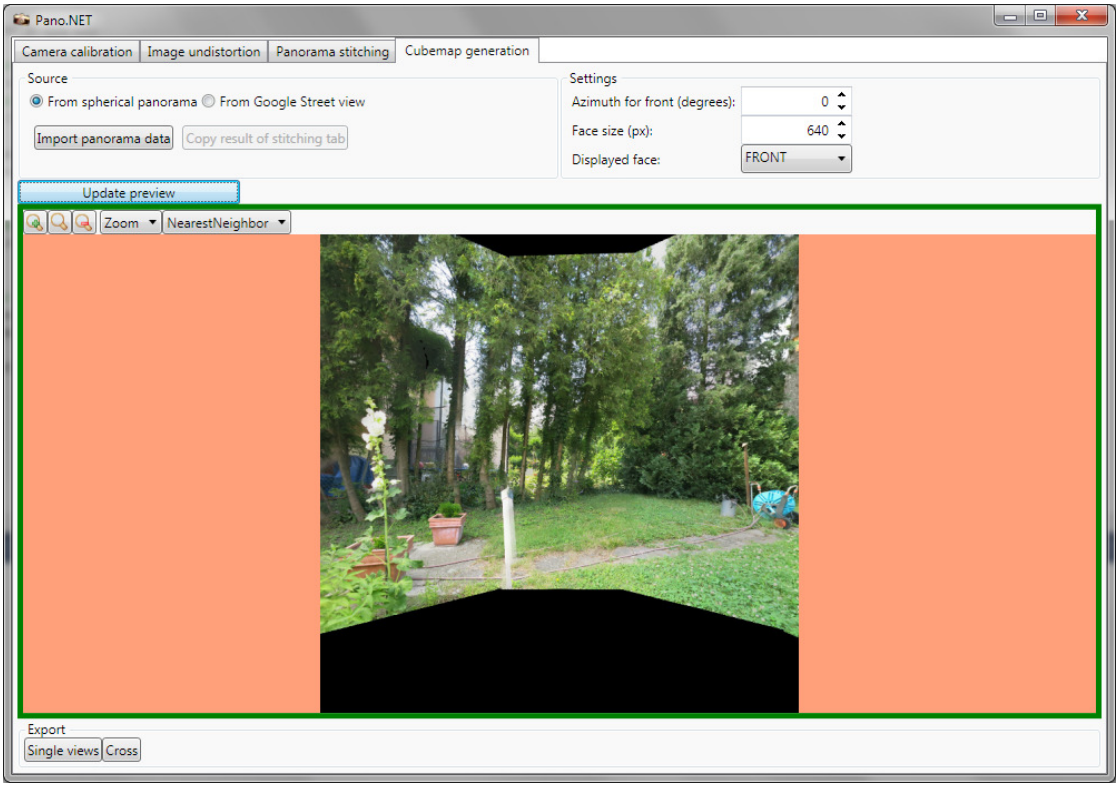


Figure 3.8: Screenshot of the cubemap generation interface of Pano.NET, showing the front face of the generated cube map.

The user interface for the cubemap generation, seen in Figure 3.8, allows the user to load a previously created spherical panorama from a metadata file, or use the panorama currently on the stitching tab. The face size in pixels and the azimuth angle θ is customizable, and the user can preview the generated faces before exporting them. They can either be exported as single faces (where each face is stored in a separate image file), or with a single-image cross configuration, using exactly the same layout as the cross shown in Figure 2.10.

Google Street View integration

Google Street View [13] is an online service from Google that provides 360° panoramas from streets and roads all around the world. Google also provides an API that allows applications to retrieve static images from any supported location, with specifiable heading and pitch [14]. For experimental purposes, we integrated this API into our cube map generator, allowing us to instantly create a cube map of panoramas of a multitude of locations.

The API consists of simple HTTP requests of the following form [14]:

```
http://maps.googleapis.com/maps/api/streetview?size=400x400&location
    =40.720032,%20-73.988354&fov=90&heading=235&pitch=10&sensor=false
```

Location can either be latitude/longitude pairs or a string describing a place (like "Times Square, New York City"). A standard PNG image is returned on success.

The heading/pitch system allows us to retrieve cube faces similar to our previously described method. The user can define the location and azimuth angle θ. The front face has then simply a `heading` parameter equal to θ and a `pitch` of 0. The right face has `heading` $= \theta + 90°$, and so on. Top and bottom have `heading` $= \theta$, and `pitch` $= \pm 90°$.

When using a `fov` (Field-of-View) parameter of $90°$, one could assume that the result is a perfect cubemap. Unfortunately, the projection method used by Google is not suitable for this naive type of cubemap generation, causing visible alignment problems as can be seen in section 4.5.

CHAPTER 4

Results

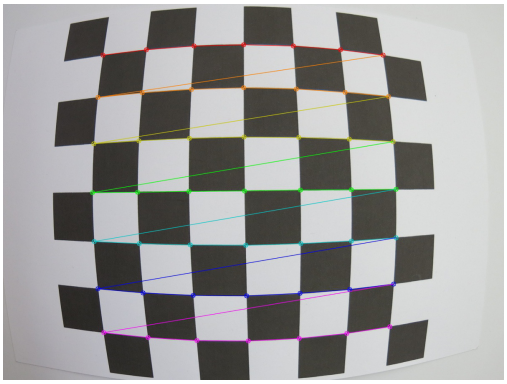
This chapter will present some examples we achieved with our solution, as well as some basic performance tests. The detailed evaluation of the effects each parameter has in the many methods we used would greatly exceed the scope of this thesis, but we will mention if we discovered remarkable quality or performance effects of a specific configuration.

4.1 Test setup

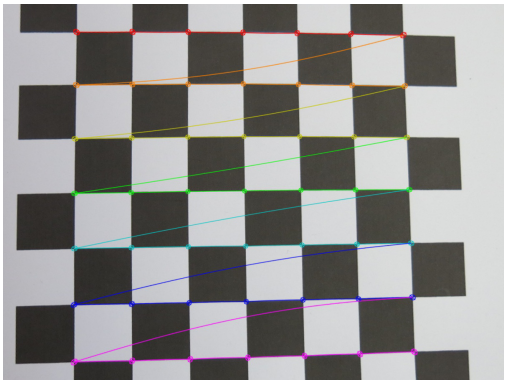
Our tests were made on a system with an Intel i7-860 quad-core processor and 16 GiB RAM. The GPGPU functionality of OpenCV could not be used because of missing access to supported hardware. Additionally, the library was not compiled with performance-enhancing multi-threading support, even though we used parallel processing in some parts of the application ourselves (pattern detection for camera calibration). Therefore, many mentioned performance indicators could probably be enhanced significantly if these features were implemented. Unless noted otherwise, we used the 64-bit version of our application.

4.2 Camera calibration and image undistortion

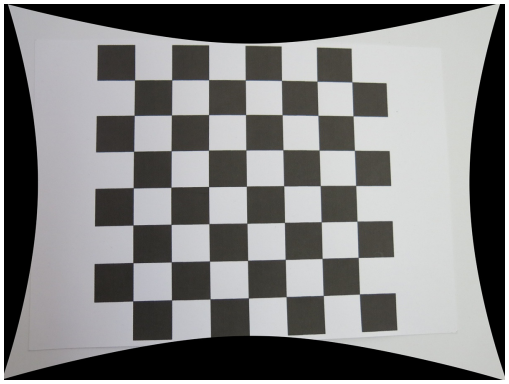
We used a set of 17 calibration images using a standard 8×8 chessboard, and additionally introduced some barrel distortion using image processing software to make the effect more apparent. The images were taken by locking the camera into a fixed position, and moving the calibration pattern printed on cardboard in front of it manually, using rotations and translations in all axes. We used a resolution of 1200×900 pixels. Fixing the aspect ratio, using the rational model of OpenCV (see section 3.2) and assuming zero tangential distortion, we achieved results like in Figure 4.1.



(a) Original image with large radial distortion, showing detected checkerboard corners



(b) Basic undistortion without camera matrix change



(c) Setting $\alpha = 1$, scaling to retain all source pixels

Figure 4.1: Image undistortion assuming square pixels, no tangential distortion and using the rational model

Option used	Results
None	6.9s
using subpixel corner refinement	7.1s
using adaptive thresholding	127.0s

Table 4.1: Performance of chessboard recognition

Performance

We compared the pattern recognition performance of some options using 20 higher-resolution versions of 4000×3000 pixels, with a maximum number of 4 images that are being processed concurrently. Results using different options can be seen in Table 4.1. With the lower resolution images, the differences are not that pronounced (around 1000ms plus maximal 300ms), but with the larger images, the adaptive threshold calculation introduces a huge increase in running time.

Normalization of the image leads to 10 out of the 20 not patterns being recognized. It is probably best to leave adaptive thresholding and normalization off by default, and only use it when some patterns are not detected.

4.3 Image stitching

The stitching component offers a wide variety of options that have an effect on performance and final panorama quality. The general stitching process is not very optimized, retaining all images in memory at all time, and does not allow for the user to add constraints on the layout of the images (for example by defining which images are overlapping). Furthermore, it is possible to crash the application by using extremely memory intensive operations like stitching of high-resolution images, using a large number of views (especially with basic plane projection) or by setting seam estimation or registration resolution too high. A way to mitigate mild cases is to use the 64-bit version of the application, which allows it to access more memory, but even then there are cases where memory usage is unacceptable, even leading to whole system crashes in very specific situations. Currently, no effort is made to predict these cases and warn the user. The defaults we chose should be suitable for panoramas a typical hobbyist might take.

The following examples all used 1200×900 pixel source images.

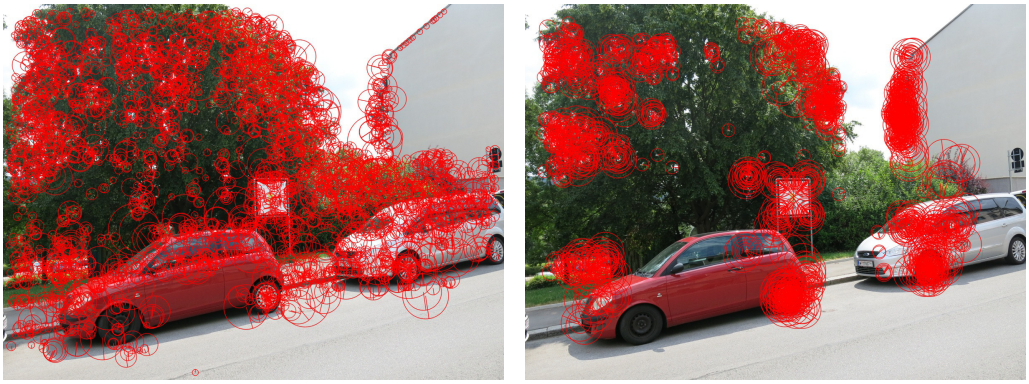
Differences between feature detectors

We found some significant difference between the results when using the OpenCV built-in SURF vs. the ORB feature detectors with their default parameters. In our test scene (seen in Figure 4.3) depicting a suburb road, mixing natural objects with man-made ones, the ORB detector seems to cluster features around a few points in each image, while the SURF detector is more spread out. This difference can be seen in Figure 4.2, and it leads to 3 out of 7 images not being added to the panorama (without changing the confidence threshold). The results of the ORB detector can probably be improved by tweaking its parameters, but this has yet to be investigated. Another probable cause is our usage of a development snapshot of OpenCV, that is not yet properly optimized in this regard.

We did a basic performance comparison on another test set that consists of 5 images of a garden scene. Here, all images are correctly added to the panorama in either detector type. The results can be seen in Table 4.2. Interestingly, even though the basic ORB detection itself is an order of magnitude faster than the SIFT detection, the overall time needed for image registration in our example was consistently higher than the time needed when using SIFT. There is also a limit on the number of ORB keypoints that can be found in each image: while the SURF detector found between 4550 and 5925 interest points per image, all but one image has 1530 ORB keypoints.

Effects of wave correction

The wave correction option can be used to straighten panoramas that are primarily horizontal or vertical. Its effects can be seen in Figure 4.3. The effect is usually more dramatic the more images are used.



(a) SURF (b) ORB

Figure 4.2: Detected features in one image

Feature detector	Detection time	Total registration time	Total feature count
SIFT	7s	40s	27777
ORB	0.2s	58s	7637

Table 4.2: Feature detector performance

(a) without wave correction

(b) with horizontal wave correction

Figure 4.3: “Road” panorama

Effects of different compositing options

The varying methods available in the compositing step can have dramatic effects on the performance of the stitching process as well as the final quality of the output panorama. For this test, we used a garden scene consisting of 26 single images, representing a 360° horizontal panorama with 2 "rows" of 13 images each, using a spherical projection. With high-frequency image details like grass and leaves, and some dramatic exposure differences, this test set presents some difficulties to the algorithms used.

Option used	Results
None / Feathering only	9.9s
Voronoi seam detection	10.4s
Graph cut seam detection (color weight)	36s
Graph cut seam detection (gradient weight)	86s
Average image exposure compensation	10.2s
Blockwise image exposure compensation (32x32 blocks)	52s
Multiband blending	11.2s
Combination for best quality	129s

Table 4.3: Performance effect of different compositing methods in the "garden" scene

The effect of some algorithms on the final panorama can be seen in Figure 4.4. Using only feathering, as seen in the top image, leads to visibly noticeable artifacts like ghosting and blurring: see the top right corner, or the wooden pole in the middle. In addition, there is a notable change in brightness in the left part of the image, owing to different camera exposure caused by strong sunlight intensity in that direction. A hard cut in brightness is seen on the walls of the house.

Seam estimation, seen in Figure 4.4b, helps remove the ghosting and blurring by defining concrete boundaries between the images. For better results, multiband blending has to be used, because the boundaries are now of course even more visible when only using feathering, as seen in this example. We used gradient-based graphcuts here, a color-based cut had more artifacts in this particular scene.

The results of the blockwise gain compensation (Figure 4.4c) are a bit conflicting: the sky gets a more uniform brightness distribution, but it also darkens the already underexposed area in the left of the image even more. Together with the seam estimation, which also left the underexposed image part, it leads to an unnatural darkening of the area in the final image in Figure 4.4e. We also tried the other compensation methods, but none led to a fully satisfactory result here. The averaging image compensation method (after Brown and Lowe [6]) brightens the area the most, but leads to an overexposure in all sky areas removing details like clouds.

The multiband blending method (Figure 4.4d) helps with removing hard cuts like seen on the house, but still retains many of the blurring and ghosting effects of feathering, unless seam estimation is used. In Figure 4.4e, it is combined with the graphcut seam estimation and blockwise gain compensation to create a good looking panorama. Blurring is not apparent, no hard seams are visible. The most obvious remaining problem is the hedge on the left side, it has a much too obvious change in brightness. The simplest way to prevent these problems is probably to already

avoid strong exposure changes when taking the pictures. Advanced methods like high dynamic range imaging [26] could also be considered, but are not supported in the application yet.

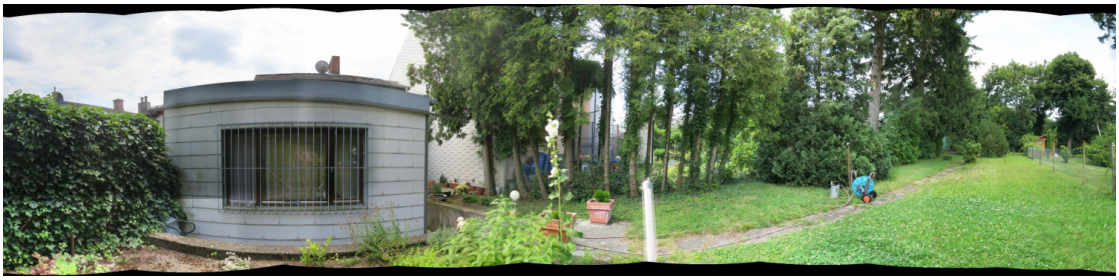
The differences in execution time of the various algorithms are sometimes pretty dramatic, as can be seen in Table 4.3. The "Combination for best quality" means the combination of graphcut seam estimation, blockwise gain compensation and multiband blending as seen in Figure 4.4e, which are incidentally the slowest methods in our application, leading to a long compositing time.

4.4 Projective drawing

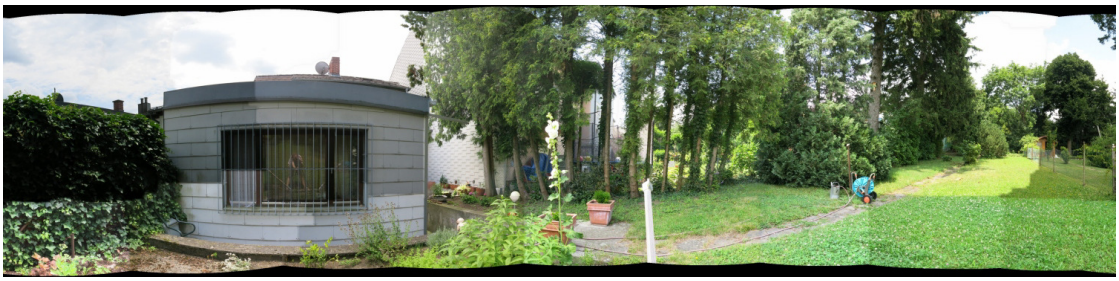
A typical result that can be achieved with our method can be seen in Figure 4.5. A series of lines follows the curvature of the road, and a polygon approximating a building facade can be seen in the panorama as well as the single views. Additionally, the wheels of the car have been marked on the single views where each is seen best. Note that this may not correspond perfectly to the location in the panorama: because of its composition of multiple overlapping images, the location may vary slightly, depending on the seams chosen.

A major problem with our approach is that lines drawn outside the frame of the currently selected view may become invalid. This is by caused our method of 2D-to-2D projection described in section 3.4, and by directly using the projection classes provided in OpenCV. The problem here is that in the backwards mapping of each projection (from panorama to view space), a check is done that when $z' \leq 0$ in Equation 3.3 (meaning that the point would be *behind* the camera), x and y get set to -1. This problem can be seen in Figure 4.6 and it is the reason why the line in Figure 4.5 has to be drawn segmented. To get around this a reimplemention of the OpenCV projection classes is required.

When doing this, a most likely better approach, one that would also allow to draw primitives on the panorama without necessarily having the registration information of each view, would be to go from 2D panorama space to the 3D space of the projection (for example the unit sphere), interpolate the lines there, and project it to each image.

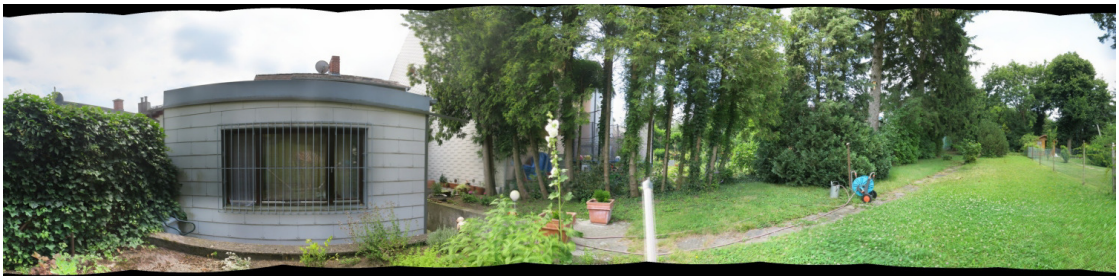


(a) Feathering only

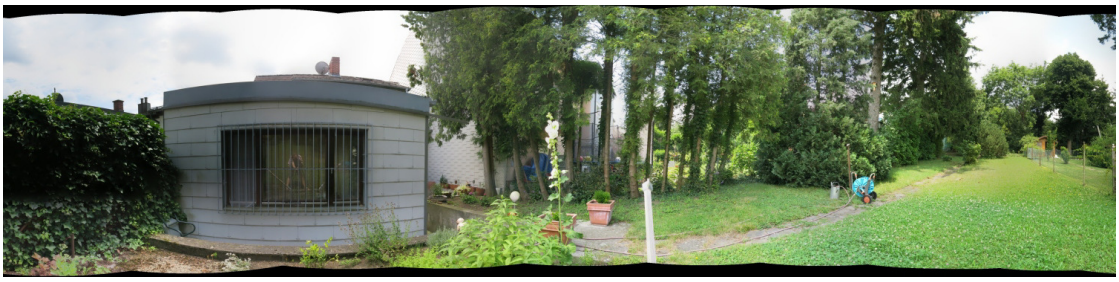


(b) Graph cut seam estimation (gradient weighting)

(c) Blockwise gain compensation (32x32 blocks)



(d) Multiband blending



(e) GC/gradient seam estimation + blockwise gain compensation + multiband blending

Figure 4.4: 26-image “garden” scene (cropped), various composition options

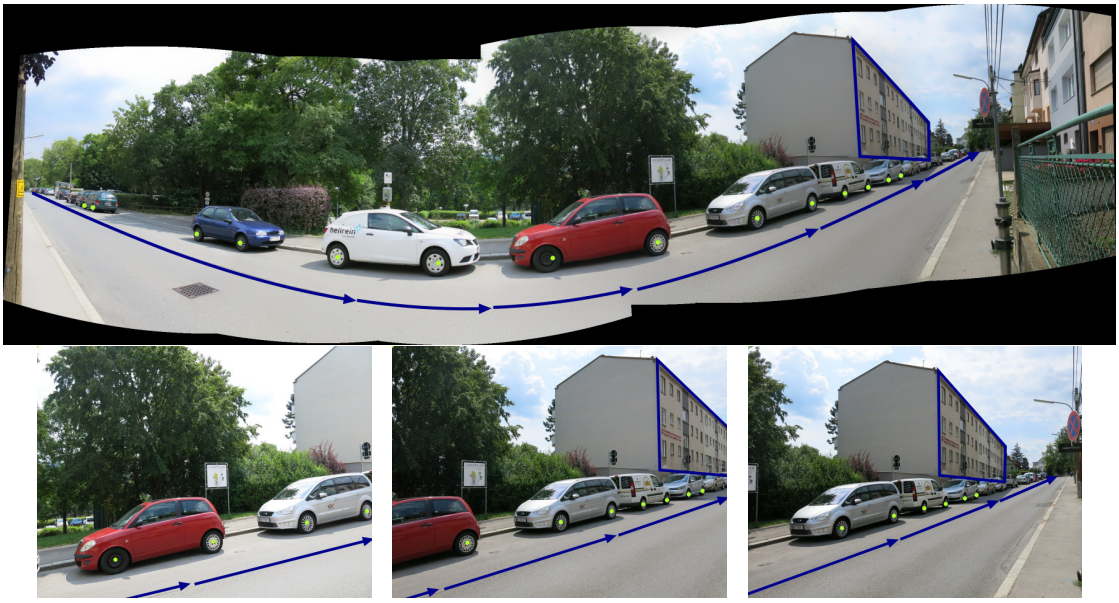


Figure 4.5: Panoramic drawing, and three select single views

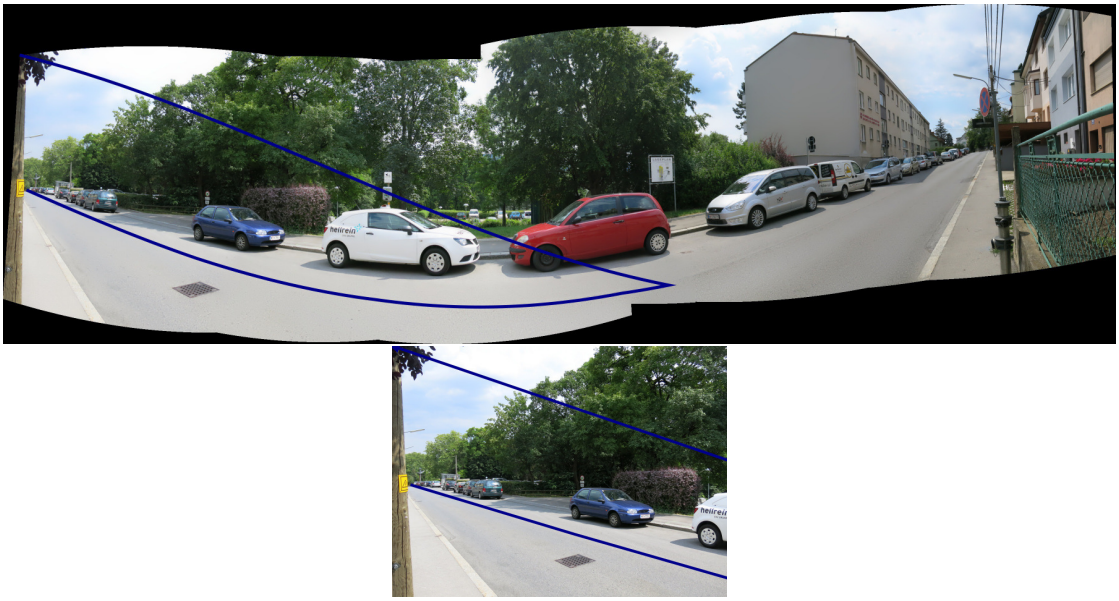


Figure 4.6: Panoramic drawing, trying to draw the same line in one stroke, but exceeding the frame of the view used for the projection too much

4.5 Cube map generation

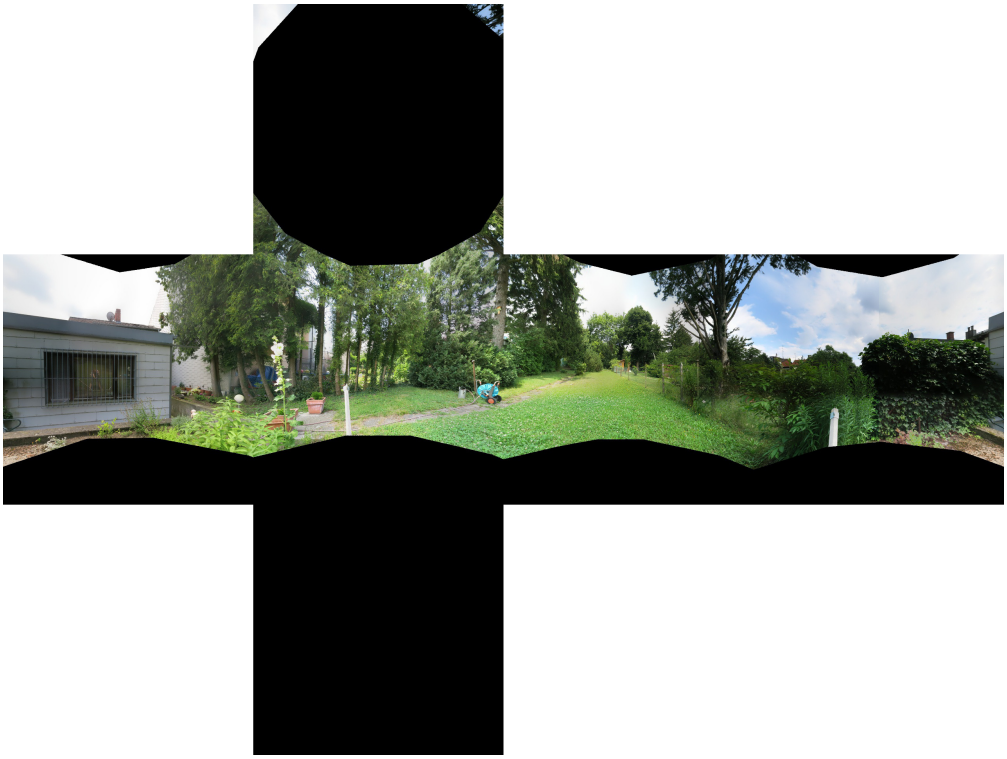
Our cube map generation method can create cubemaps out of any spherical panorama created with our application. An example, converting the garden scene panorama seen in Figure 4.4 into a cube map, can be seen in Figure 4.8. This uses the traditional cross representation, as described in section 2.4. When using a $\theta \neq 0$, a seam might be visible in $360°$ horizontal panoramas, as seen here in Figure 4.8b (central/front face), where the edges of the original panorama are. This is caused by the interpolation when remapping, and can be solved by clamping to edge values instead of setting to zero.

Figure 4.7: A high-FOV rendering of a cube map in a 3D application. Cube edges are visible.

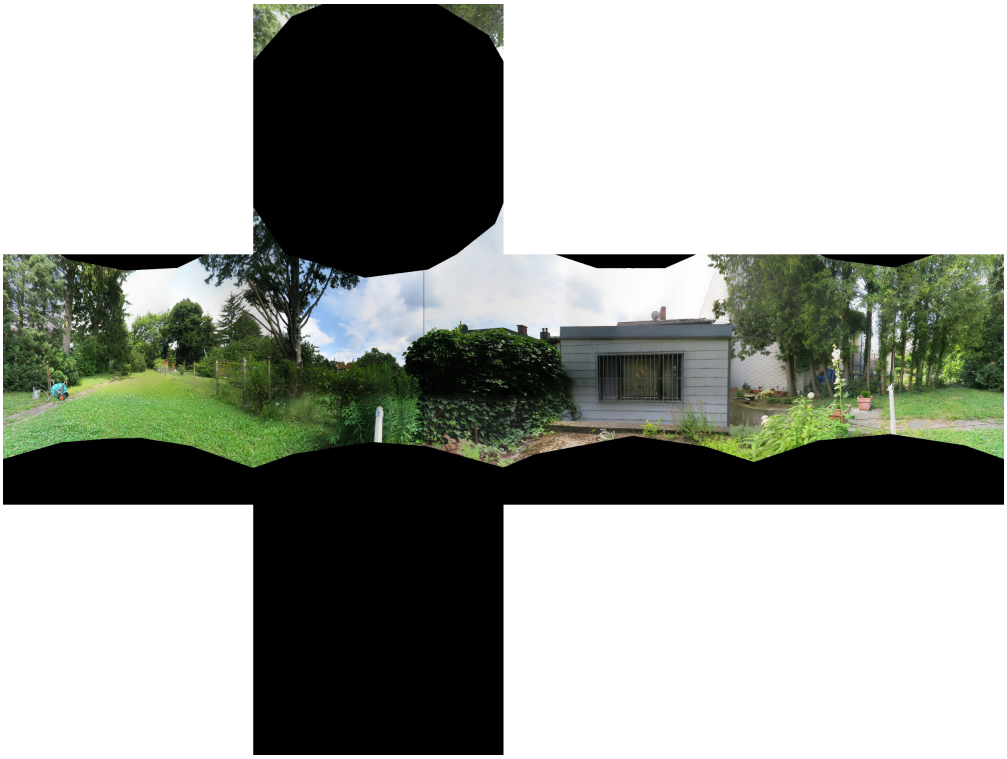
The generated cubemap can then be used for applications such as skyboxes, as seen in Figure 4.7. Here, a camera was placed inside the box, and used to take a high field-of-view shot of the cube. The edges of the cube are also visible.

Google Street View integration

The application allows the user to request panoramic images for each location supported by Google Street View. The face size is limited to a maximum of 640^2 pixels, because of API limitations. Unfortunately, cube maps generated by the method described in section 3.5 have misalignments at the edges, the severity also seemingly dependent on the requested location, with some having a stronger misalignment than others. This problem can be seen in Figure 4.9. The problem may be slightly reduced in some cases by varying the requested FOV to be a few degrees above or below $90°$, but overall it can't be removed with our basic cubemap generation method.



(a) $\theta = 0°$



(b) $\theta = 160°$

Figure 4.8: Cross cube map representation of the panorama in Figure 4.4

(a) Cube map, showing large amounts of misalignment

(b) 3D cube map rendering

Figure 4.9: Cube map with data from Google Street View (New York City location), showing alignment problem

CHAPTER 5

Conclusion

In this thesis we focused on topics related to camera calibration and image stitching techniques. We took a look at the basic algorithms required for calibration, image registration and panorama compositing in chapter 2, and how they are implemented using the OpenCV [22] computer vision library in chapter 3. We created an interactive .NET application using this library that allows the user to do camera calibration and image stitching tasks. Furthermore, we presented a method for drawing points and lines on a panoramic mosaic or its sub-images, considering also the projective warping of the panorama. Our last topic was the generation of cubemaps out of existing panoramas, including an experimental usage of Google Street View [13] data.

The results of our work and a comparison of different methods can be seen in chapter 4, showing that there is sometimes a large difference between various algorithms in terms of quality produced and calculation time required, and some limitations in our drawing and cubemap generation methods.

5.1 Future work

Image stitching, and its application in larger problems like the automated detection of scene structures, is an ongoing research topic. There are several areas where the application might be improved. Some possible enhancements which we considered during the work on this thesis are listed as follows:

- Improving general application performance in many areas by using GPU acceleration and multithreading (as described in section 4.1)
- Support for additional pattern types (not just checkerboards) in the camera calibration module, allowing the user to directly print the pattern from the application
- Adding additional algorithms to various steps of OpenCV's stitching pipeline, like the SIFT [18] feature detector or the "panorama weaving" [31] method for advanced interactive seam processing

- Better support for larger panoramas, consisting of many images and/or high resolution views, using methods to reduce memory usage such as on-demand loading of image files
- Support for preexisting scene information, such as a simple way for the user to specify which images are overlapping, saving processing time
- Fixing the problems with projective drawing described in section 4.4, considering also the possibility to interpolate lines directly in 3D projective space
- Support for more panorama types (not just spherical ones) in cubemap generation

It is planned to use the application written during this thesis in work done in the course of the *Photo-Guide* [36] project. The project aims to create tools for image-based city exploration, allowing for example virtual first-person navigation through a city and even building interiors. Applications can be found in tourism, city planning and other areas.

APPENDIX A

Export file format examples

Note that JSON does not support comments, these are only included for clarification purposes.

A.1 Camera calibration

```
{
  //calibration options used
  "Options": {
    "Pattern": {
      "PatternSize":[7,7],
      "PatternScaling": 10.0,
      "PatternType": "CHESSBOARD",
      "UseSubpixelRefinement": true,
      "SubpixelWindowSize": 11,
      "SubpixelIterations": 30,
      "ChessboardFlags": "ADAPTIVE_THRESH, NORMALIZE_IMAGE,
         FAST_CHECK"
    },
    "CalibrationFlags": "CV_CALIB_FIX_ASPECT_RATIO,
       CV_CALIB_FIX_PRINCIPAL_POINT, CV_CALIB_ZERO_TANGENT_DIST,
       CV_CALIB_FIX_K4, CV_CALIB_FIX_K5",
    "ImageSize":[1200,900]
  },
  //average reprojection error
  "Error": 3.1194570536782247,
  "IntrinsicCameraParameters": {
    //3x3 intrinsic matrix, in row-major order
    "IntrinsicMatrix": [
      3694.6987984298621,
      0.0,
      599.5,
```

```
      0.0,
      3694.6987984298621,
      449.5,
      0.0,
      0.0,
      1.0
    ],
    "DistortionCoeffs": [
      0.70678661285199573, //k1
      -186.74482384962428, //k2
      0.0, //p1
      0.0, //p2
      17911.937698921975, //k3
      0.0, //k4
      0.0, //k5
      0.0  //k6
    ]
  },
  //array of all calibration images
  "Images": [
    {
      "PatternFound": true,
      "ImageFile": "C:\\Users\\freddi\\Desktop\\BApanos\\smaller\\
         calib\\IMG_2452.JPG",
      "ImageSize":[1200,900],
      "ReprojectionError": 3.11945748,
      "CameraParameters": {
        "RotationVector": [
          -0.23971747238474295,
          -0.0510133431974203,
          -0.012786064630724013
        ],
        "TranslationVector": [
          -32.283580226348477,
          -29.193752665278968,
          420.21514957334716
        ]
      },
      //positions of detected pattern points, (x,y) pairs
      "PatternPoints":[311.803345,188.345825,403.0889,
         188.983414,493.432068,189.555222,...],
      "PatternSize":[7,7]
    },
    //...
    //repeat for all other calibration images
  ]
}
```

A.2 Image stitching

```
{
  //scaling of the projective space to get final image coordinates (=
     median focal length)
  "WarpedImageScale": 987.79461669921875,
  //scaling of the source images for final compositing
  "CompositingScale": 1.0,
  //the projection used for this panorama
  "WarperUsed": "CYLINDRICAL",
  //top left corner of the image in projective coordinates
  "TopLeft": {
    "IsEmpty": false,
    "X": -1010.0,
    "Y": -449.0
  },
  //array of all source views
  "SourceImages": [
    {
      "IsPartOfPano": true,
      "ImageFile": "C:\\Users\\freddi\\Desktop\\BApanos\\smaller\\
         street\\IMG_2443.JPG",
      "ImageSize":[1200,900],
      "Camera": {
        //3x3 intrinsic matrix, row-major order
        "K": [
          993.7617,
          0.0,
          599.71344,
          0.0,
          993.7617,
          450.120483,
          0.0,
          0.0,
          1.0
        ],
        //3x3 rotation matrix, row-major order
        "R": [
          0.7414955,
          -0.0292380154,
          0.6703205,
          0.0331419632,
          0.9994266,
          0.00693196058,
          -0.670138836,
          0.01707572,
          0.7420394
        ]
      }
```

```
    },
    //....
    //repeat for further views
  ],
  //panorama file
  "ImageFile": "C:\\Users\\freddi\\Desktop\\BApanos\\smaller\\street
     \\cylinder.jpg",
  "ImageSize":[2281,959]
}
```

A.3 Projective drawing

```
{
  //array for all drawn points
  "Points": [
    {
      //the location in the panorama (in pixels)
      "PanoramaPoint": "1295.58644462726,621.882689200162",
      //locations in each of the source views (in pixels)
      "ViewPoints": [
        "132.155532836914,651.332397460938",
        "893.895141601563,630.711791992188",
        "1501.16931152344,684.217590332031",
        "950.72119140625,597.698120117188"
      ],
      //the index of the source view where the user has place the
         point originally, or -1 for points placed in the panorama
      "OriginalIndex": -1
    },
    //...
    //repeated for each point placed
  ],
  //array for drawn lines AND polygons
  "Lines": [
    {
      //determines the indices of the PanoramaLine array which
         correspond to control points placed by the user
      "PanoCornerIdx": [
        0,
        39,
        89,
        151,
        195
      ],
      //the line control + interpolation points in panorama space
      "PanoramaLine": [
        "334.422192953115,516.459331619277",
```

```
        "342.121520996094,518.789093017578",
        "349.918395996094,521.144035339355",
        "357.813659667969,523.524116516113",
        "365.808166503906,525.929313659668",
        //... snip ...
      ],
      //an array of arrays, for the same points in each view space
      "ViewLines": [
        [
          "-5840.61279296875,1103.275390625",
          "-5521.65283203125,1085.81640625",
          "-5228.56103515625,1069.7734375",
          "-4958.31103515625,1054.98046875",
          //...snip...
        ],
        [
          //note that the view in which the user created the line
             only has the control points, because of the straight
             line assumption!
          "-206.505630493164,537.261169433594",
          "290.772613525391,632.538208007813",
          "796.212341308594,732.280639648438",
          "1669.77026367188,916.011901855469",
          "3785.140625,1376.80078125"
        ],
        [
          "381.793151855469,465.199737548828",
          "389.673858642578,467.818359375",
          "397.626007080078,470.460723876953",
          "405.650695800781,473.127197265625",
          "413.748779296875,475.818115234375",
          "421.921356201172,478.533721923828",
          //...snip...
        ],
        //...
        //continue for the other views
      ],
      //the arrow type, currently End or None
      "ArrowType": "End",
      "ArrowLength": 25.0,
      "ArrowAngle": 45.0,
      //if true, the last control point equals the first. this is not
          added to the arrays above, but the interpolation points are
         !
      "IsPolygon": false,
      //the index of the source view where the user has place the
         point originally, or -1 for points placed in the panorama
      "OriginalIndex": 1
    },
```

```
78      //...
79      //repeat for all other lines and polygons
80    ]
81  }
```

Bibliography

[1] Adobe Systems Inc. Photoshop - Create panoramic images with Photomerge. http://help.adobe.com/en_US/photoshop/cs/using/WSfd1234e1c4b69f30ea53e41001031ab64-75e8a.html. Accessed: 2013-07-06.

[2] Aseem Agarwala, Mira Dontcheva, Maneesh Agrawala, Steven Drucker, Alex Colburn, Brian Curless, David Salesin, and Michael Cohen. Interactive Digital Photomontage. *ACM Trans. Graph.*, 23(3):294–302, August 2004.

[3] Herbert Bay, Tinne Tuytelaars, and Luc Gool. SURF: Speeded Up Robust Features. In *Computer Vision - ECCV 2006*, volume 3951 of *Lecture Notes in Computer Science*, pages 404–417. Springer Berlin Heidelberg, 2006.

[4] Stuart Bennett and Joan Lasenby. ChESS-Quick and Robust Detection of Chess-board Features. *arXiv preprint arXiv:1301.5491*, 2013.

[5] Duane C. Brown. Close-range camera calibration. *Photogrammetric engineering*, 37(8):855–866, 1971.

[6] Matthew Brown and David G. Lowe. Automatic Panoramic Image Stitching using Invariant Features. *International Journal of Computer Vision*, 74(1):59–73, 2007.

[7] Peter J. Burt and Edward H. Adelson. A Multiresolution Spline with Application to Image Mosaics. *ACM Trans. Graph.*, 2(4):217–236, October 1983.

[8] Michael Calonder, Vincent Lepetit, Christoph Strecha, and Pascal Fua. BRIEF: Binary Robust Independent Elementary Features. In Kostas Daniilidis, Petros Maragos, and Nikos Paragios, editors, *Computer Vision - ECCV 2010*, volume 6314 of *Lecture Notes in Computer Science*, pages 778–792. Springer Berlin Heidelberg, 2010.

[9] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627 (Informational) http://www.ietf.org/rfc/rfc4627.txt, July 2006.

[10] EmguCV. Emgu CV - a cross platform .NET wrapper to the OpOpen image processing library. http://www.emgu.com. Accessed: 2013-07-15.

[11] Martin A. Fischler and Robert C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24(6):381–395, June 1981.

[12] Google Inc. Android documentation - Panoramic photos. https://support.google.com/android/answer/1753571. Accessed: 2013-07-06.

[13] Google Inc. Google street view. https://maps.google.com/intl/en-US/help/maps/streetview/. Accessed: 2013-08-06.

[14] Google Inc. Google street view image api. https://developers.google.com/maps/documentation/streetview/, June 2013. Accessed: 2013-08-06.

[15] N. Greene. Environment Mapping and Other Applications of World Projections. *Computer Graphics and Applications, IEEE*, 6(11):21–29, 1986.

[16] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, ISBN: 0521540518, second edition, 2004.

[17] Vivek Kwatra, Arno Schödl, Irfan Essa, Greg Turk, and Aaron Bobick. Graphcut Textures: Image and Video Synthesis Using Graph Cuts. *ACM Trans. Graph.*, 22(3):277–286, July 2003.

[18] David G. Lowe. Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[19] Microsoft Corporation. MSDN - Direct3D9 Cubic Environment Mapping. http://msdn.microsoft.com/en-us/library/windows/desktop/bb204881%28v=vs.85%29.aspx. Accessed: 2013-07-15.

[20] Microsoft Corporation. WPF - Windows Presentation Foundation. http://msdn.microsoft.com/en-us//library/ms754130.aspx. Accessed: 2013-07-15.

[21] David L. Milgram. Computer methods for creating photomosaics. *IEEE Transactions on Computers*, C-24(11):1113–1119, 1975.

[22] OpenCV. OpenCV - The Open Source Computer Vision Library. http://opencv.org/. Accessed: 2013-07-06.

[23] OpenCV. OpenCV Camera Calibration documentation. http://docs.opencv.org/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html. Accessed: 2013-07-15.

[24] OpenCV. OpenCV Geometric Transformation documentation. http://docs.opencv.org/modules/imgproc/doc/geometric_transformations.html. Accessed: 2013-07-15.

[25] OpenCV. OpenCV Image Stitching documentation. http://docs.opencv.org/modules/stitching/doc/stitching.html. Accessed: 2013-07-12.

[26] Erik Reinhard, Wolfgang Heidrich, Paul Debevec, Sumanta Pattanaik, Greg Ward, and Karol Myszkowski. *High dynamic range imaging: acquisition, display, and image-based lighting*. Morgan Kaufmann, 2010.

[27] Edward Rosten and Tom Drummond. Machine Learning for High-Speed Corner Detection. In *Computer Vision - ECCV 2006*, volume 3951 of *Lecture Notes in Computer Science*, pages 430–443. Springer Berlin Heidelberg, 2006.

[28] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski. ORB: An efficient alternative to SIFT or SURF. In *2011 IEEE International Conference on Computer Vision (ICCV)*, pages 2564–2571, 2011.

[29] Heung-Yeung Shum and Richard Szeliski. Systems and Experiment Paper: Construction of Panoramic Image Mosaics with Global and Local Alignment. *International Journal of Computer Vision*, 36(2):101–130, 2000.

[30] Josh Smith. WPF Apps With The Model-View-ViewModel Design Pattern. http://msdn.microsoft.com/en-us/magazine/dd419663.aspx, February 2009. Accessed: 2013-07-15.

[31] Brian Summa, Julien Tierny, and Valerio Pascucci. Panorama weaving: fast and flexible seam processing. *ACM Trans. Graph.*, 31(4):83:1–83:11, July 2012.

[32] Richard Szeliski. Image Alignment and Stitching: A Tutorial. *Found. Trends. Comput. Graph. Vis.*, 2(1):1–104, January 2006.

[33] Richard Szeliski. *Computer vision: algorithms and applications*. Springer, 2011.

[34] Bill Triggs, PhilipF. McLauchlan, RichardI. Hartley, and AndrewW. Fitzgibbon. Bundle Adjustment - A Modern Synthesis. In Bill Triggs, Andrew Zisserman, and Richard Szeliski, editors, *Vision Algorithms: Theory and Practice*, volume 1883 of *Lecture Notes in Computer Science*, pages 298–372. Springer Berlin Heidelberg, 2000.

[35] M. Uyttendaele, A. Eden, and R. Skeliski. Eliminating ghosting and exposure artifacts in image mosaics. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, volume 2, pages II–509–II–516 vol.2, 2001.

[36] Michael Wimmer. Photo-Guide: Image-Based City Exploration. http://www.cg.tuwien.ac.at/research/projects/Photo-Guide/, 2013. Accessed: 2013-08-16.

[37] Zhengyou Zhang. A flexible new technique for camera calibration. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 22(11):1330–1334, 2000.