Chapter No. 4
"Exploring Structure from Motion Using OpenCV"
**In this package, you will find:**

- A Biography of the authors of the book
- A preview chapter from the book, Chapter NO.4 "Exploring Structure from Motion Using OpenCV"
- A synopsis of the book’s content
- Information on where to buy this book

---

**About the Authors**

**Daniel Lélis Baggio** started his work in computer vision through medical image processing at InCor (Instituto do Coração – Heart Institute) in São Paulo, where he worked with intra-vascular ultrasound image segmentation. Since then, he has focused on GPGPU and ported the segmentation algorithm to work with NVIDIA's CUDA. He has also dived into six degrees of freedom head tracking with a natural user interface group through a project called ehci (http://code.google.com/p/ehci/). He now works for the Brazilian Air Force.

I'd like to thank God for the opportunity of working with computer vision. I try to understand the wonderful algorithms He has created for us to see. I also thank my family, and especially my wife, for all their support throughout the development of the book. I'd like to dedicate this book to my son Stefano.

---

**For More Information:**

Shervin Emami (born in Iran) taught himself electronics and hobby robotics during his early teens in Australia. While building his first robot at the age of 15, he learned how RAM and CPUs work. He was so amazed by the concept that he soon designed and built a whole Z80 motherboard to control his robot, and wrote all the software purely in binary machine code using two push buttons for 0s and 1s. After learning that computers can be programmed in much easier ways such as assembly language and even high-level compilers, Shervin became hooked to computer programming and has been programming desktops, robots, and smartphones nearly every day since then. During his late teens he created Draw3D (http://draw3d.shervinemami.info/), a 3D modeler with 30,000 lines of optimized C and assembly code that rendered 3D graphics faster than all the commercial alternatives of the time; but he lost interest in graphics programming when 3D hardware acceleration became available.

In University, Shervin took a subject on computer vision and became highly interested in it; so for his first thesis in 2003 he created a real-time face detection program based on Eigenfaces, using OpenCV (beta 3) for camera input. For his master’s thesis in 2005 he created a visual navigation system for several mobile robots using OpenCV (v0.96). From 2008, he worked as a freelance Computer Vision Developer in Abu Dhabi and Philippines, using OpenCV for a large number of short-term commercial projects that included:

- Detecting faces using Haar or Eigenfaces
- Recognizing faces using Neural Networks, EHMM, or Eigenfaces
- Detecting the 3D position and orientation of a face from a single photo using AAM and POSIT
- Rotating a face in 3D using only a single photo
- Face preprocessing and artificial lighting using any 3D direction from a single photo
- Gender recognition
- Facial expression recognition
- Skin detection
- Iris detection
- Pupil detection
- Eye-gaze tracking
- Visual-saliency tracking
- Histogram matching
- Body-size detection

For More Information:
OpenCV was putting food on the table for Shervin's family, so he began giving back to OpenCV through regular advice on the forums and by posting free OpenCV tutorials on his website (http://www.shervinemami.info/openCV.html). In 2011, he contacted the owners of other free OpenCV websites to write this book. He also began working on computer vision optimization for mobile devices at NVIDIA, working closely with the official OpenCV developers to produce an optimized version of OpenCV for Android. In 2012, he also joined the Khronos OpenVL committee for standardizing the hardware acceleration of computer vision for mobile devices, on which OpenCV will be based in the future.

I thank my wife Gay and my baby Luna for enduring the stress while I juggled my time between this book, working fulltime, and raising a family. I also thank the developers of OpenCV, who worked hard for many years to provide a high-quality product for free.

David Millán Escrivá was eight years old when he wrote his first program on an 8086 PC with Basic language, which enabled the 2D plotting of basic equations. In 2005, he finished his studies in IT through the Universitat Politècnica de Valencia with honors in human-computer interaction supported by computer vision with OpenCV (v0.96). He had a final project based on this subject and published it on HCI Spanish congress. He participated in Blender, an open source, 3D-software project, and worked in his first commercial movie Plumíferos - Aventuras voladoras as a Computer Graphics Software Developer.

David now has more than 10 years of experience in IT, with experience in computer vision, computer graphics, and pattern recognition, working on different projects and startups, applying his knowledge of computer vision, optical character recognition, and augmented reality. He is the author of the "DamilesBlog" (http://blog.damiles.com), where he publishes research articles and tutorials about OpenCV, computer vision in general, and Optical Character Recognition algorithms.
David has reviewed the book *gnuPlot Cookbook* by *Lee Phillips* and published by *Packt Publishing*.

Thanks Izaskun and my daughter Eider for their patience and support. Os quiero pequeñas.

I also thank Shervin for giving me this opportunity, the OpenCV team for their work, the support of Artres, and the useful help provided by Augmate.

**Khvedchenia Ievgen** is a computer vision expert from Ukraine. He started his career with research and development of a camera-based driver assistance system for Harman International. He then began working as a Computer Vision Consultant for ESG. Nowadays, he is a self-employed developer focusing on the development of augmented reality applications. Ievgen is the author of the *Computer Vision Talks* blog (http://computer-vision-talks.com), where he publishes research articles and tutorials pertaining to computer vision and augmented reality.

I would like to say thanks to my father who inspired me to learn programming when I was 14. His help can't be overstated. And thanks to my mom, who always supported me in all my undertakings. You always gave me a freedom to choose my own way in this life. Thanks, parents!

Thanks to Kate, a woman who totally changed my life and made it extremely full. I'm happy we're together. Love you.

**Naureen Mahmood** is a recent graduate from the Visualization department at Texas A&M University. She has experience working in various programming environments, animation software, and microcontroller electronics. Her work involves creating interactive applications using sensor-based electronics and software engineering. She has also worked on creating physics-based simulations and their use in special effects for animation.

I wanted to especially mention the efforts of another student from Texas A&M, whose name you will undoubtedly come across in the code included for this book. Fluid Wall was developed as part of a student project by Austin Hines and myself. Major credit for the project goes to Austin, as he was the creative mind behind it. He was also responsible for the arduous job of implementing the fluid simulation code into our application. However, he wasn't able to participate in writing this book due to a number of work- and study-related preoccupations.

**For More Information:**

Jason Saragih received his B.Eng degree in mechatronics (with honors) and Ph.D. in computer science from the Australian National University, Canberra, Australia, in 2004 and 2008, respectively. From 2008 to 2010 he was a Postdoctoral fellow at the Robotics Institute of Carnegie Mellon University, Pittsburgh, PA. From 2010 to 2012 he worked at the Commonwealth Scientific and Industrial Research Organization (CSIRO) as a Research Scientist. He is currently a Senior Research Scientist at Visual Features, an Australian tech startup company.

Dr. Saragih has made a number of contributions to the field of computer vision, specifically on the topic of deformable model registration and modeling. He is the author of two non-profit open source libraries that are widely used in the scientific community; DeMoLib and FaceTracker, both of which make use of generic computer vision libraries including OpenCV.

Roy Shilkrot is a researcher and professional in the area of computer vision and computer graphics. He obtained a B.Sc. in Computer Science from Tel-Aviv-Yaffo Academic College, and an M.Sc. from Tel-Aviv University. He is currently a PhD candidate in Media Laboratory of the Massachusetts Institute of Technology (MIT) in Cambridge.

Roy has over seven years of experience as a Software Engineer in start-up companies and enterprises. Before joining the MIT Media Lab as a Research Assistant he worked as a Technology Strategist in the Innovation Laboratory of Comverse, a telecom solutions provider. He also dabbled in consultancy, and worked as an intern for Microsoft research at Redmond.

Thanks go to my wife for her limitless support and patience, my past and present advisors in both academia and industry for their wisdom, and my friends and colleagues for their challenging thoughts.

For More Information:
Mastering OpenCV with Practical Computer Vision Projects

Mastering OpenCV with Practical Computer Vision Projects contains nine chapters, where each chapter is a tutorial for an entire project from start to finish, based on OpenCV’s C++ interface including full source code. The author of each chapter was chosen for their well-regarded online contributions to the OpenCV community on that topic, and the book was reviewed by one of the main OpenCV developers. Rather than explaining the basics of OpenCV functions, this is the first book that shows how to apply OpenCV to solve whole problems, including several 3D camera projects (augmented reality, 3D Structure from Motion, Kinect interaction) and several facial analysis projects (such as, skin detection, simple face and eye detection, complex facial feature tracking, 3D head orientation estimation, and face recognition), therefore it makes a great companion to existing OpenCV books.

What This Book Covers

Chapter 1, Cartoonifier and Skin Changer for Android, contains a complete tutorial and source code for both a desktop application and an Android app that automatically generates a cartoon or painting from a real camera image, with several possible types of cartoons including a skin color changer.

Chapter 2, Marker-based Augmented Reality on iPhone or iPad, contains a complete tutorial on how to build a marker-based augmented reality (AR) application for iPad and iPhone devices with an explanation of each step and source code.

Chapter 3, Marker-less Augmented Reality, contains a complete tutorial on how to develop a marker-less augmented reality desktop application with an explanation of what marker-less AR is and source code.

Chapter 4, Exploring Structure from Motion Using OpenCV, contains an introduction to Structure from Motion (SfM) via an implementation of SfM concepts in OpenCV. The reader will learn how to reconstruct 3D geometry from multiple 2D images and estimate camera positions.

Chapter 5, Number Plate Recognition Using SVM and Neural Networks, contains a complete tutorial and source code to build an automatic number plate recognition application using pattern recognition algorithms using a support vector machine and Artificial Neural Networks. The reader will learn how to train and predict pattern-recognition algorithms to decide if an image is a number plate or not. It will also help classify a set of features into a character.

For More Information:
Chapter 6, Non-rigid Face Tracking, contains a complete tutorial and source code to build a dynamic face tracking system that can model and track the many complex parts of a person's face.

Chapter 7, 3D Head Pose Estimation Using AAM and POSIT, contains all the background required to understand what Active Appearance Models (AAMs) are and how to create them with OpenCV using a set of face frames with different facial expressions. Besides, this chapter explains how to match a given frame through fitting capabilities offered by AAMs. Then, by applying the POSIT algorithm, one can find the 3D head pose.

Chapter 8, Face Recognition using Eigenfaces or Fisherfaces, contains a complete tutorial and source code for a real-time face-recognition application that includes basic face and eye detection to handle the rotation of faces and varying lighting conditions in the images.

Chapter 9, Developing Fluid Wall Using the Microsoft Kinect, covers the complete development of an interactive fluid simulation called the Fluid Wall, which uses the Kinect sensor. The chapter will explain how to use Kinect data with OpenCV's optical flow methods and integrating it into a fluid solver.

You can download this chapter from:

For More Information:
Exploring Structure from Motion Using OpenCV

In this chapter we will discuss the notion of Structure from Motion (SfM), or better put as extracting geometric structures from images taken through a camera's motion, using functions within OpenCV's API to help us. First, let us constrain the otherwise lengthy footpath of our approach to using a single camera, usually called a monocular approach, and a discrete and sparse set of frames rather than a continuous video stream. These two constrains will greatly simplify the system we will sketch in the coming pages, and help us understand the fundamentals of any SfM method. To implement our method we will follow in the footsteps of Hartley and Zisserman (hereafter referred to as H and Z), as documented in chapters 9 through 12 of their seminal book *Multiple View Geometry in Computer Vision*.

In this chapter we cover the following:

- Structure from Motion concepts
- Estimating the camera motion from a pair of images
- Reconstructing the scene
- Reconstruction from many views
- Refinement of the reconstruction
- Visualizing 3D point clouds

Throughout the chapter we assume the use of a calibrated camera—one that was calibrated beforehand. **Calibration** is a ubiquitous operation in computer vision, fully supported in OpenCV using command-line tools and was discussed in previous chapters. We therefore assume the existence of the camera's intrinsic parameters embodied in the K matrix, one of the outputs from the calibration process.

For More Information:

To make things clear in terms of language, from this point on we will refer to a camera as a single view of the scene rather than to the optics and hardware taking the image. A camera has a position in space, and a direction of view. Between two cameras, there is a translation element (movement through space) and a rotation of the direction of view.

We will also unify the terms for the point in the scene, world, real, or 3D to be the same thing, a point that exists in our real world. The same goes for points in the image or 2D, which are points in the image coordinates, of some real 3D point that was projected on the camera sensor at that location and time.

In the chapter’s code sections you will notice references to Multiple View Geometry in Computer Vision, for example // HZ 9.12. This refers to equation number 12 of chapter 9 of the book. Also, the text will include excerpts of code only, while the complete runnable code is included in the material accompanied with the book.

**Structure from Motion concepts**

The first discrimination we should make is the difference between stereo (or indeed any multiview), 3D reconstruction using calibrated rigs, and SfM. While a rig of two or more cameras assume we already know what the motion between the cameras is, in SfM we don’t actually know this motion and we wish to find it. Calibrated rigs, from a simplistic point of view, allow a much more accurate reconstruction of 3D geometry because there is no error in estimating the distance and rotation between the cameras—it is already known. The first step in implementing an SfM system is finding the motion between the cameras. OpenCV may help us in a number of ways to obtain this motion, specifically using the `findFundamentalMat` function.

Let us think for one moment of the goal behind choosing an SfM algorithm. In most cases we wish to obtain the geometry of the scene, for example, where objects are in relation to the camera and what their form is. Assuming we already know the motion between the cameras picturing the same scene, from a reasonably similar point of view, we would now like to reconstruct the geometry. In computer vision jargon this is known as **triangulation**, and there are plenty of ways to go about it. It may be done by way of ray intersection, where we construct two rays: one from each camera’s center of projection and a point on each of the image planes. The intersection of these rays in space will, ideally, intersect at one 3D point in the real world that was imaged in each camera, as shown in the following diagram:
In reality, ray intersection is highly unreliable; H and Z recommend against it. This is because the rays usually do not intersect, making us fall back to using the middle point on the shortest segment connecting the two rays. Instead, H and Z suggest a number of ways to triangulate 3D points, of which we will discuss a couple of them in the *Reconstructing the scene* section. The current version of OpenCV does not contain a simple API for triangulation, so this part we will code on our own.

After we have learned how to recover 3D geometry from two views, we will see how we can incorporate more views of the same scene to get an even richer reconstruction. At that point, most SfM methods try to optimize the bundle of estimated positions of our cameras and 3D points by means of Bundle Adjustment, in the *Refinement of the reconstruction* section. OpenCV contains means for Bundle Adjustment in its new Image Stitching Toolbox. However, the beauty of working with OpenCV and C++ is the abundance of external tools that can be easily integrated into the pipeline. We will therefore see how to integrate an external bundle adjuster, the neat SSBA library.

Now that we have sketched an outline of our approach to SfM using OpenCV, we will see how each element can be implemented.
Estimating the camera motion from a pair of images

Before we set out to actually find the motion between two cameras, let us examine the inputs and the tools we have at hand to perform this operation. First, we have two images of the same scene from (hopefully not extremely) different positions in space. This is a powerful asset, and we will make sure to use it. Now as far as tools go, we should take a look at mathematical objects that impose constraints over our images, cameras, and the scene.

Two very useful mathematical objects are the fundamental matrix (denoted by F) and the essential matrix (denoted by E). They are mostly similar, except that the essential matrix is assuming usage of calibrated cameras; this is the case for us, so we will choose it. OpenCV only allows us to find the fundamental matrix via the `findFundamentalMat` function; however, it is extremely simple to get the essential matrix from it using the calibration matrix \( K \) as follows:

\[
Mat_{<\text{double}>} E = K.t() \ast F \ast K; \quad //\text{according to HZ (9.12)}
\]

The essential matrix, a 3 x 3 sized matrix, imposes a constraint between a point in one image and a point in the other image with \( x'Ex = 0 \), where \( x \) is a point in image one and \( x' \) is the corresponding point in image two. This is extremely useful, as we are about to see. Another important fact we use is that the essential matrix is all we need in order to recover both cameras for our images, although only up to scale; but we will get to that later. So, if we obtain the essential matrix, we know where each camera is positioned in space, and where it is looking. We can easily calculate the matrix if we have enough of those constraint equations, simply because each equation can be used to solve for a small part of the matrix. In fact, OpenCV allows us to calculate it using just seven point-pairs, but hopefully we will have many more pairs and get a more robust solution.

Point matching using rich feature descriptors

Now we will make use of our constraint equations to calculate the essential matrix. To get our constraints, remember that for each point in image A we must find a corresponding point in image B. How can we achieve such a matching? Simply by using OpenCV's extensive feature-matching framework, which has greatly matured in the past few years.
Feature extraction and descriptor matching is an essential process in computer vision, and is used in many methods to perform all sorts of operations. For example, detecting the position and orientation of an object in the image or searching a big database of images for similar images through a given query. In essence, **extraction** means selecting points in the image that would make the features good, and computing a descriptor for them. A **descriptor** is a vector of numbers that describes the surrounding environment around a feature point in an image. Different methods have different length and data type for their descriptor vectors. **Matching** is the process of finding a corresponding feature from one set in another using its descriptor. OpenCV provides very easy and powerful methods to support feature extraction and matching. More information about feature matching may be found in Chapter 3, Marker-less Augmented Reality.

Let us examine a very simple feature extraction and matching scheme:

```cpp
// detecting keypoints
SurfFeatureDetector detector();
vector<KeyPoint> keypoints1, keypoints2;
detector.detect(img1, keypoints1);
detector.detect(img2, keypoints2);

// computing descriptors
SurfDescriptorExtractor extractor;
Mat descriptors1, descriptors2;
extractor.compute(img1, keypoints1, descriptors1);
extractor.compute(img2, keypoints2, descriptors2);

// matching descriptors
BruteForceMatcher<L2<float>> matcher;
vector<DMatch> matches;
matcher.match(descriptors1, descriptors2, matches);
```

You may have already seen similar OpenCV code, but let us review it quickly. Our goal is to obtain three elements: Feature points for two images, descriptors for them, and a matching between the two sets of features. OpenCV provides a range of feature detectors, descriptor extractors, and matchers. In this simple example we use the SurfFeatureDetector function to get the 2D location of the **Speeded-Up Robust Features (SURF)** features, and the SurfDescriptorExtractor function to get the SURF descriptors. We use a brute-force matcher to get the matching, which is the most straightforward way to match two feature sets implemented by comparing each feature in the first set to each feature in the second set (hence the phrasing brute-force) and getting the best match.

---

For More Information:
In the next image we will see a matching of feature points on two images from the Fountain-P11 sequence found at http://cvlab.epfl.ch/~strecha/multiview/denseMVS.html.

Practically, raw matching like we just performed is good only up to a certain level, and many matches are probably erroneous. For that reason, most SfM methods perform some form of filtering on the matches to ensure correctness and reduce errors. One form of filtering, which is the built-in OpenCV’s brute-force matcher, is cross-check filtering. That is, a match is considered true if a feature of the first image matched a feature of the second image, and the reverse check also matched the feature of the second image with the feature of the first image. Another common filtering mechanism, used in the provided code, is to filter based on the fact that the two images are of the same scene and have a certain stereo-view relationship between them. In practice, the filter tries to robustly calculate the fundamental matrix, of which we will learn in the Finding camera matrices section, and retain those feature pairs that correspond with this calculation with small errors.

**Point matching using optical flow**

An alternative to using rich features, such as SURF, is using optical flow (OF). The following information box provides a short overview of optical flow. OpenCV recently extended its API for getting the flow field from two images and now it is faster and more powerful. We will try to use it as an alternative to matching features.
Optical flow is the process of matching selected points from one image to another, assuming both images are part of a sequence and relatively close to one another. Most optical flow methods compare a small region, known as the search window or patch, around each point from image A to the same area in image B. Following a very common rule in computer vision, called the brightness constancy constraint (and other names), the small patches of the image will not change drastically from one image to the other, and therefore the magnitude of their subtraction should be close to zero. In addition to matching patches, newer methods of optical flow use a number of additional methods to get better results. One is using image pyramids, which are smaller and smaller resized versions of the image, which allow for working "from-coarse-to-fine"—a very well-used trick in computer vision. Another method is to define global constraints on the flow field, assuming that the points close to each other "move together" in the same direction. A more in-depth review of optical flow methods in OpenCV can be found in Chapter Developing Fluid Wall Using the Microsoft Kinect which is available on the Packt website.

Using optical flow in OpenCV is fairly easy by invoking the calcOpticalFlowPyrLK function. However, we would like to keep the result matching from OF similar to that using rich features, as in the future we would like the two approaches to be interchangeable. To that end, we must install a special matching method—one that is interchangeable with the previous feature-based method, which we are about to see in the code section that follows:

```cpp
Vector<KeyPoint> left_keypoints, right_keypoints;

// Detect keypoints in the left and right images
FastFeatureDetector ffd;
ffd.detect(img1, left_keypoints);
ffd.detect(img2, right_keypoints);

vector<Point2f> left_points;
KeyPointsToPoints(left_keypoints, left_points);

vector<Point2f> right_points(left_points.size());

// making sure images are grayscale
Mat prevgray, gray;
if (img1.channels() == 3) {
    cvtColor(img1, prevgray, CV_RGB2GRAY);
    cvtColor(img2, gray, CV_RGB2GRAY);
} else {
    prevgray = img1;
}
```

For More Information:
gray = img2;

// Calculate the optical flow field:
// how each left_point moved across the 2 images
vector<uchar>vstatus; vector<float>verror;
calcOpticalFlowPyrLK(prevgray, gray, left_points, right_points,
vstatus, verror);

// First, filter out the points with high error
vector<Point2f>right_points_to_find;
vector<int>right_points_to_find_back_index;
for (unsigned int i=0; i<vstatus.size(); i++) {
  if (vstatus[i] && verror[i] < 12.0) {
    // Keep the original index of the point in the
    // optical flow array, for future use
    right_points_to_find_back_index.push_back(i);
    // Keep the feature point itself
    right_points_to_find.push_back(j_pts[i]);
  } else {
    vstatus[i] = 0; // a bad flow
  }
}

// for each right_point see which detected feature it belongs to
Mat right_points_to_find_flat = Mat(right_points_to_find).reshape(1,to_find.size()); //flatten array

vector<Point2f>right_features; // detected features
KeyPointsToPoints(right_keypoints,right_features);

Mat right_features_flat = Mat(right_features).reshape(1,right_features.size());

// Look around each OF point in the right image
// for any features that were detected in its area
// and make a match.
BFMatcher matcher(CV_L2);
vector<vector<DMatch>>nearest_neighbors;
matcher.radiusMatch(
  right_points_to_find_flat,
  right_features_flat,
  nearest_neighbors,

For More Information:
// Check that the found neighbors are unique (throw away neighbors
// that are too close together, as they may be confusing)
std::set<int> found_in_right_points; // for duplicate prevention
for (int i = 0; i < nearest_neighbors.size(); i++) {
    DMatch _m;
    if (nearest_neighbors[i].size() == 1) {
        _m = nearest_neighbors[i][0]; // only one neighbor
    } else if (nearest_neighbors[i].size() > 1) {
        double ratio = nearest_neighbors[i][0].distance / nearest_neighbors[i][1].distance;
        if (ratio < 0.7) { // not too close
            _m = nearest_neighbors[i][0];
        } else { // too close - we cannot tell which is better
            continue; // did not pass ratio test - throw away
        }
    } else {
        continue; // no neighbors... :
    }

    // prevent duplicates
    if (found_in_right_points.find(_m.trainIdx) == found_in_right_points.end()) {
        // The found neighbor was not yet used:
        // We should match it with the original indexing
        // of the left point
        _m.queryIdx = right_points_to_find_back_index[_m.queryIdx];
        matches->push_back(_m); // add this match
        found_in_right_points.insert(_m.trainIdx);
    }
}

cout << "pruned " << matches->size() << " / " << nearest_neighbors.size() << " matches" << endl;

The functions KeyPointsToPoints and PointsToKeyPoints are simply convenience functions for conversion between the cv::Point2f and the cv::KeyPoint structs.
Exploring Structure from Motion Using OpenCV

In the previous segment of code we can see a number of interesting things. The first thing to note is that when we use optical flow, our result shows a feature moved from a position in the image on the left-hand side to another position in the image on the right-hand side. But we have a new set of features detected in the image to the right-hand side, not necessarily aligning with the features that flowed from the image to the left-hand side in optical flow. We must align them. To find those lost features we use a **k-nearest neighbor (kNN)** radius search, which gives us up to two features that fall within a 2-pixel radius to the points of interest.

One more thing that we can see is an implementation of the ratio test for kNN, which is a common practice in SfM to reduce errors. In essence, it is a filter that removes confusing matches when we have a match between one feature in the left-hand side image and two features in the right-hand side image. If the two features in the right-hand side image are too close together, or the ratio between them is too big (close to 1.0), we consider them confusing and do not use them. We also install a duplicate prevention filter to further prune the matches.

The following image shows the flow field from one image to another. Pink arrows in the left-hand side image show the movement of patches from the left-hand side image to the right-hand side image. In the second image to the left, we see a small area of the flow field zoomed in. The pink arrows again show the motion of patches, and we can see it makes sense by looking at the two original image segments on the right-hand side. Visual features in the left-hand side image are moving leftwards across the image, in the directions of the pink arrows as shown in the following image:

The advantage of using optical flow in place of rich features is that the process is usually faster and can accommodate the matching many more points, making the reconstruction denser. In many optical flow methods there is also a monolithic model of the overall movement of patches, where matching rich features are usually not taken into account. The caveat in working with optical flow is that it works best for consecutive images taken by the same hardware, whereas rich features are mostly agnostic to that. The differences result from the fact that optical flow methods usually use very rudimentary features, like image patches around a keypoint, whereas higher-order richer features (for example, SURF) take into account higher-level information for each keypoint. Using optical flow or rich features is a decision the designer of the application should make depending on the input.
Finding camera matrices

Now that we have obtained matches between keypoints, we can calculate the fundamental matrix and from that obtain the essential matrix. However, we must first align our matching points into two arrays, where an index in one array corresponds to the same index in the other. This is required by the `findFundamentalMat` function. We would also need to convert the KeyPoint structure to a Point2f structure. We must pay special attention to the queryIdx and trainIdx member variables of `DMatch`, the OpenCV struct that holds a match between two keypoints, as they must align with the way we used the matcher. `match()` function. The following code section shows how to align a matching into two corresponding sets of 2D points, and how these can be used to find the fundamental matrix:

```cpp
vector<Point2f> imgpts1, imgpts2;
for( unsigned int i = 0; i<matches.size(); i++ )
{
    // queryIdx is the "left" image
    imgpts1.push_back(keypoints1[matches[i].queryIdx].pt);
    // trainIdx is the "right" image
    imgpts2.push_back(keypoints2[matches[i].trainIdx].pt);
}

Mat F = findFundamentalMat(imgpts1, imgpts2, FM_RANSAC, 0.1, 0.99, status);
Mat_<double> E = K.t() * F * K; //according to HZ (9.12)
```

We may later use the status binary vector to prune those points that align with the recovered fundamental matrix. See the following image for an illustration of point matching after pruning with the fundamental matrix. The red arrows mark feature matches that were removed in the process of finding the F matrix, and the green arrows are feature matches that were kept.
Now we are ready to find the camera matrices. This process is described at length in chapter 9 of H and Z's book; however, we are going to use a very straightforward and simplistic implementation of it, and OpenCV makes things very easy for us. But first, we will briefly examine the structure of the camera matrix we are going to use.

\[
P = [R|t] = \begin{bmatrix} r_1 & r_2 & r_3 & t_1 \\ r_4 & r_5 & r_6 & t_2 \\ r_7 & r_8 & r_9 & t_3 \end{bmatrix}
\]

This is the model for our camera, it consists of two elements, rotation (denoted as \( R \)) and translation (denoted as \( t \)). The interesting thing about it is that it holds a very essential equation: \( x = PX \), where \( x \) is a 2D point on the image and \( X \) is a 3D point in space. There is more to it, but this matrix gives us a very important relationship between the image points and the scene points. So, now that we have a motivation for finding the camera matrices, we will see how it can be done. The following code section shows how to decompose the essential matrix into the rotation and translation elements:

```cpp
SVD svd(E);
Matx33d W(0,-1,0, //HZ 9.13
```

For More Information:
1,0,0,
0,0,1);
Mat_<double> R = svd.u * Mat(W) * svd.vt; //HZ 9.19
Mat_<double> t = svd.u.col(2); //u3
Matx34d P1(  R(0,0),R(0,1), R(0,2), t(0),
            R(1,0),R(1,1), R(1,2), t(1),
            R(2,0),R(2,1), R(2,2), t(2));

Very simple. All we had to do is take the Singular Value Decomposition (SVD) of the essential matrix we obtained from before, and multiply it by a special matrix W. Without going too deeply into the mathematical interpretation of what we did, we can say the SVD operation decomposed our matrix E into two parts, a rotation element and a translation element. In fact, the essential matrix was originally composed by the multiplication of these two elements. Strictly for satisfying our curiosity we can look at the following equation for the essential matrix, which appears in the literature: E=[t]xR. We see it is composed of (some form of) a translation element and a rotational element R.

We notice that what we just did only gives us one camera matrix, so where is the other camera matrix? Well, we perform this operation under the assumption that one camera matrix is fixed and canonical (no rotation and no translation). The next camera matrix is also canonical:

\[
P_0 = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0
\end{bmatrix}
\]

The other camera that we recovered from the essential matrix has moved and rotated in relation to the fixed one. This also means that any of the 3D points that we recover from these two camera matrices will have the first camera at the world origin point (0, 0, 0).

This, however, is not the complete solution. H and Z show in their book how and why this decomposition has in fact four possible camera matrices, but only one of them is the true one. The correct matrix is the one that will produce reconstructed points with a positive Z value (points that are in front of the camera). But we can only understand that after learning about triangulation and 3D reconstruction, which will be discussed in the next section.
One more thing we can think of adding to our method is error checking. Many a times the calculation of the fundamental matrix from the point matching is erroneous, and this affects the camera matrices. Continuing triangulation with faulty camera matrices is pointless. We can install a check to see if the rotation element is a valid rotation matrix. Keeping in mind that rotation matrices must have a determinant of 1 (or -1), we can simply do the following:

```cpp
bool CheckCoherentRotation(cv::Mat_<double>& R) {
    if(fabsf(determinant(R)) - 1.0 > 1e-07) {
        cerr << "det(R) != +-1.0, this is not a rotation matrix" << endl;
        return false;
    }
    return true;
}
```

We can now see how all these elements combine into a function that recovers the $P$ matrices, as follows:

```cpp
void FindCameraMatrices(const Mat& K, const Mat& Kinv,
const vector<KeyPoint>& imgpts1, const vector<KeyPoint>& imgpts2,
Matx34d& P, Matx34d& P1, vector<DMatch>& matches,
vector<CloudPoint>& outCloud )
{
    //Find camera matrices
    //Get Fundamental Matrix
    Mat F = GetFundamentalMat(imgpts1, imgpts2, matches);
    //Essential matrix: compute then extract cameras \([R|t]\)
    Mat_<double> E = K.t() * F * K; //according to HZ (9.12)
    //decompose E to \(P'\), HZ (9.19)
    SVD svd(E, SVD::MODIFY_A);
    Mat svd_u = svd.u;
    Mat svd_vt = svd.vt;
    Mat svd_w = svd.w;
    Matx33d W(0, -1, 0, //HZ 9.13
        1, 0, 0,
        0, 0, 1);
```
Chapter 4

Mat_<double> R = svd_u * Mat(W) * svd_vt; //HZ 9.19
Mat_<double> t = svd_u.col(2); //u3

if (!CheckCoherentRotation(R)) {
    cout<<"resulting rotation is not coherent\n";
    P1 = 0;
    return;
}

P1 = Matx34d(R(0,0),R(0,1),R(0,2),t(0),
               R(1,0),R(1,1),R(1,2),t(1),
               R(2,0),R(2,1),R(2,2),t(2));
}

At this point we have the two cameras that we need in order to reconstruct the scene. The canonical first camera, in the \( \mathbf{P} \) variable, and the second camera we calculated, form the fundamental matrix in the \( \mathbf{P}_1 \) variable. The next section will reveal how we use these cameras to obtain a 3D structure of the scene.

Reconstructing the scene

Next we look into the matter of recovering the 3D structure of the scene from the information we have acquired so far. As we had done before, we should look at the tools and information we have at hand to achieve this. In the preceding section we obtained two camera matrices from the essential and fundamental matrices; we already discussed how these tools will be useful for obtaining the 3D position of a point in space. Then, we can go back to our matched point pairs to fill in our equations with numerical data. The point pairs will also be useful in calculating the error we get from all our approximate calculations.

This is the time to see how we can perform triangulation using OpenCV. This time we will follow the steps Hartley and Sturm take in their article *Triangulation*, where they implement and compare a few triangulation methods. We will implement one of their linear methods, as it is very simple to code with OpenCV.

For More Information:
Remember we had two key equations arising from the 2D point matching and P matrices: x = PX and x' = P'X, where x and x' are matching 2D points and X is a real world 3D point imaged by the two cameras. If we rewrite the equations, we can formulate a system of linear equations that can be solved for the value of X, which is what we desire to find. Assuming X = (x, y, z, 1)^t (a reasonable assumption for points that are not too close or too far from the camera center) creates an inhomogeneous linear equation system of the form AX = B. We can code and solve this equation system as follows:

```cpp
Mat_<double> LinearLSTriangulation(
    Point3d u,  // homogenous image point (u,v,1)
    Matx34d P,  // camera 1 matrix
    Point3d u1, // homogenous image point in 2nd camera
    Matx34d P1  // camera 2 matrix
)
{
    // build A matrix
    Matx43d A(u.x*P(2,0)-P(0,0), u.x*P(2,1)-P(0,1), u.x*P(2,2)-P(0,2),
              u.y*P(2,0)-P(1,0), u.y*P(2,1)-P(1,1), u.y*P(2,2)-P(1,2),
              u1.x*P1(2,0)-P1(0,0), u1.x*P1(2,1)-P1(0,1), u1.x*P1(2,2)-P1(0,2),
              u1.y*P1(2,0)-P1(1,0), u1.y*P1(2,1)-P1(1,1), u1.y*P1(2,2)-P1(1,2)
    );
    // build B vector
    Matx41d B(-(u.x*P(2,3)-P(0,3)),
                -(u.y*P(2,3)-P(1,3)),
                -(u1.x*P1(2,3)-P1(0,3)),
                -(u1.y*P1(2,3)-P1(1,3)));

    // solve for X
    Mat_<double> X;
    solve(A, B, X, DECOMP_SVD);
    return X;
}
```
This will give us an approximation for the 3D points arising from the two 2D points. One more thing to note is that the 2D points are represented in homogenous coordinates, meaning the x and y values are appended with a 1. We should make sure these points are in normalized coordinates, meaning that they were multiplied by the calibration matrix $K$ beforehand. We may notice that instead of multiplying each point by the matrix $K$ we can simply make use of the $KP$ matrix (the $K$ matrix multiplied by the $P$ matrix), as $H$ and $Z$ do throughout chapter 9. We can now write a loop over the point matches to get a complete triangulation as follows:

```c++
double TriangulatePoints(
    const vector<KeyPoint>& pt_set1,
    const vector<KeyPoint>& pt_set2,
    const Mat&Kinv,
    const Matx34d& P,
    const Matx34d& P1,
    vector<Point3d>& pointcloud)
{
    vector<double> reproj_error;
    for (unsigned int i=0; i<pts_size; i++) {
        //convert to normalized homogeneous coordinates
        Point2f kp = pt_set1[i].pt;
        Point3d u(kp.x,kp.y,1.0);
        Mat_<double> um = Kinv * Mat_<double>(u);
        u = um.at<Point3d>(0);
        Point2f kp1 = pt_set2[i].pt;
        Point3d u1(kp1.x,kp1.y,1.0);
        Mat_<double> um1 = Kinv * Mat_<double>(u1);
        u1 = um1.at<Point3d>(0);

        //triangulate
        Mat_<double> X = LinearLSTriangulation(u,P,u1,P1);

        //calculate reprojection error
        Mat_<double> xPt_img = K * Mat(P1) * X;
        Point2f xPt_img_(xPt_img(0)/xPt_img(2),xPt_img(1)/xPt_img(2));
        reproj_error.push_back(norm(xPt_img_-kp1));

        //store 3D point
        pointcloud.push_back(Point3d(X(0),X(1),X(2)));
    }

    //return mean reprojection error
    Scalar me = mean(reproj_error);
    return me[0];
}
```

For More Information:
In the following image we will see a triangulation result of two images out of the Fountain P-11 sequence at http://cvlab.epfl.ch/~strecha/multiview/denseMVS.html. The two images at the top are the original two views of the scene, and the bottom pair is the view of the reconstructed point cloud from the two views, including the estimated cameras looking at the fountain. We can see how the right-hand side section of the red brick wall was reconstructed, and also the fountain that protrudes from the wall.

However, as we discussed earlier, we have an issue with the reconstruction being only up-to-scale. We should take a moment to understand what up-to-scale means. The motion we obtained between our two cameras is going to have an arbitrary unit of measurement, that is, it is not in centimeters or inches but simply a given unit of scale. Our reconstructed cameras will be one unit of scale distance apart. This has big implications should we decide to recover more cameras later, as each pair of cameras will have their own units of scale, rather than a common one.
We will now discuss how the error measure that we set up may help us in finding a more robust reconstruction. First we should note that reprojection means we simply take the triangulated 3D point and reimage it on a camera to get a reprojected 2D point, we then compare the distance between the original 2D point and the reprojected 2D point. If this distance is large this means we may have an error in triangulation, so we may not want to include this point in the final result. Our global measure is the average reprojection distance and may give us a hint to how our triangulation performed overall. High average reprojection rates may point to a problem with the \( P \) matrices, and therefore a possible problem with the calculation of the essential matrix or the matched feature points.

We should briefly go back to our discussion of camera matrices in the previous section. We mentioned that composing the camera matrix \( P_1 \) can be performed in four different ways, but only one composition is correct. Now that we know how to triangulate a point, we can add a check to see which one of the four camera matrices is valid. We shall skip the implementation details at this point, as they are featured in the sample code attached to the book.

Next we are going to take a look at recovering more cameras looking at the same scene, and combining the 3D reconstruction results.

### Reconstruction from many views

Now that we know how to recover the motion and scene geometry from two cameras, it would seem trivial to get the parameters of additional cameras and more scene points simply by applying the same process. This matter is in fact not so simple as we can only get a reconstruction that is up-to-scale, and each pair of pictures gives us a different scale.

There are a number of ways to correctly reconstruct the 3D scene data from multiple views. One way is of resection or camera pose estimation, also known as Perspective N-Point (PNP), where we try to solve for the position of a new camera using the scene points we have already found. Another way is to triangulate more points and see how they fit into our existing scene geometry; this will tell us the position of the new camera by means of the Iterative Closest Point (ICP) procedure. In this chapter we will discuss using OpenCV’s `solvePnP` functions to achieve the first method.

---

For More Information:
The first step we choose in this kind of reconstruction—incremental 3D reconstruction with camera resection—is to get a baseline scene structure. As we are going to look for the position of any new camera based on a known structure of the scene, we need to find an initial structure and a baseline to work with. We can use the method we previously discussed—for example, between the first and second frames—to get a baseline by finding the camera matrices (using the FindCameraMatrices function) and triangulate the geometry (using the TriangulatePoints function).

Having found an initial structure, we may continue; however, our method requires quite a bit of bookkeeping. First we should note that the solvePnP function needs two aligned vectors of 3D and 2D points. Aligned vectors mean that the ith position in one vector aligns with the ith position in the other. To obtain these vectors we need to find those points among the 3D points that we recovered earlier, which align with the 2D points in our new frame. A simple way to do this is to attach, for each 3D point in the cloud, a vector denoting the 2D points it came from. We can then use feature matching to get a matching pair.

Let us introduce a new structure for a 3D point as follows:

```cpp
struct CloudPoint {
    cv::Point3d pt;
    std::vector<int> index_of_2d_origin;
};
```

It holds, on top of the 3D point, an index to the 2D point inside the vector of 2D points that each frame has, which had contributed to this 3D point. The information for index_of_2d_origin must be initialized when triangulating a new 3D point, recording which cameras were involved in the triangulation. We can then use it to trace back from our 3D point cloud to the 2D point in each frame, as follows:

```cpp
std::vector<CloudPoint> pcloud; //our global 3D point cloud

//check for matches between i'th frame and 0'th frame (and thus the current cloud)
std::vector<cv::Point3f> ppcloud;
std::vector<cv::Point2f> imgPoints;
vector<int> pcloud_status(pcloud.size(),0);

//scan the views we already used (good_views)
for (set<int>::iterator done_view = good_views.begin(); done_view !=
good_views.end(); ++done_view)
{
    int old_view = *done_view; //a view we already used for reconstruction
```
/check for matches_from_old_to_working between <working_view>'th frame and <old_view>'th frame (and thus the current cloud)
std::vector<cv::DMatch> matches_from_old_to_working = matches_matrix[std::make_pair(old_view,working_view)];
// scan the 2D-2D matched-points
for (unsigned int match_from_old_view=0; match_from_old_view<matches_from_old_to_working.size(); match_from_old_view++) {
    // the index of the matching 2D point in <old_view>
    int idx_in_old_view = matches_from_old_to_working[match_from_old_view].queryIdx;
    // scan the existing cloud to see if this point from <old_view> exists for (unsigned int pcldp=0; pcldp<pcloud.size(); pcldp++) {
        // see if this 2D point from <old_view> contributed to this 3D point in the cloud
        if (idx_in_old_view == pcloud[pcldp].index_of_2d_origin[old_view] && pcloud_status[pcldp] == 0) // prevent duplicates
            {
                // 3D point in cloud
                ppcloud.push_back(pcloud[pcldp].pt);
                // 2D point in image <working_view>
                Point2d pt_ = imgpts[working_view][matches_from_old_to_working[match_from_old_view].trainIdx].pt;
                imgPoints.push_back(pt_);
                pcloud_status[pcldp] = 1;
                break;
            }
    }
}
cout << "found " << ppcloud.size() << " 3d-2d point correspondences" << endl;

Now we have an aligned pairing of 3D points in the scene to the 2D points in a new frame, and we can use them to recover the camera position as follows:

    cv::Mat_<double> t, rvec, R;
    cv::solvePnPRansac(ppcloud, imgPoints, K, distcoeff, rvec, t, false);

    // get rotation in 3x3 matrix form
    Rodrigues(rvec, R);

    P1 = cv::Matx34d(R(0,0),R(0,1),R(0,2),t(0),
                    R(1,0),R(1,1),R(1,2),t(1),
                    R(2,0),R(2,1),R(2,2),t(2));

For More Information:
Note that we are using the `solvePnPRansac` function rather than the `solvePnP` function as it is more robust to outliers. Now that we have a new $P_1$ matrix, we can simply use the `TriangulatePoints` function we defined earlier again and populate our point cloud with more 3D points.

In the following image we see an incremental reconstruction of the Fountain-P11 scene at http://cvlab.epfl.ch/~strecha/multiview/denseMVS.html, starting from the 4th image. The top-left image is the reconstruction after four images were used; the participating cameras are shown as red pyramids with a white line showing the direction. The other images show how more cameras add more points to the cloud.
Re
fi
nement of the reconstruction

One of the most important parts of an SfM method is refining and optimizing the reconstructed scene, also known as the process of Bundle Adjustment (BA). This is an optimizing step where all the data we gathered is fitted to a monolithic model. Both the position of the 3D points and the positions of cameras are optimized, so reprojection errors are minimized (that is, approximated 3D points are projected on the image close to the position of originating 2D points). This process usually entails the solving of very big linear equations of the order of tens of thousands of parameters. The process may be slightly laborious, but the steps we took earlier will allow for an easy integration with the bundle adjuster. Some things that seemed strange earlier may become clear; for example, the reason we retain the origin 2D points for each 3D point in the cloud.

One implementation of a bundle adjustment algorithm is the Simple Sparse Bundle Adjustment (SSBA) library; we will choose it as our BA optimizer as it has a simple API. It requires only a few input arguments that we can create rather easily from our data structures. The key object we will use from SSBA is the CommonInternalsMetricBundleOptimizer function, which performs the optimization. It needs the camera parameters, the 3D point cloud, the 2D image points that correspond to each point in the point cloud, and cameras looking at the scene. By now it should be straightforward to come up with these parameters. We should note that this method of BA assumes all images were taken by the same hardware, hence the common internals, other modes of operation may not assume this. We can perform Bundle Adjustment as follows:

```cpp
void BundleAdjuster::adjustBundle(
    vector<CloudPoint>& pointcloud,
    const Mat& cam_intrinsics,
    const std::vector<std::vector<cv::KeyPoint>>& imgpts,
    std::map<int, cv::Matx34d>& Pmats
    
) {
    int N = Pmats.size(), M = pointcloud.size(), K = -1;

    cout << "N (cams) = " << N << " M (points) = " << M << " K (measurements) = " << K << endl;

    StdDistortionFunction distortion;

    // intrinsic parameters matrix
    Matrix3x3d KMat;
    makeIdentityMatrix(KMat);
    KMat[0][0] = cam_intrinsics.at<double>(0, 0);
```

For More Information:
Exploring Structure from Motion Using OpenCV

KMat[0][1] = cam_intrinsics.at<double>(0,1);
KMat[0][2] = cam_intrinsics.at<double>(0,2);
KMat[1][1] = cam_intrinsics.at<double>(1,1);
KMat[1][2] = cam_intrinsics.at<double>(1,2);

...  

// 3D point cloud
vector<Vector3d >Xs(M);
for (int j = 0; j < M; ++j)
{
    Xs[j][0] = pointcloud[j].pt.x;
    Xs[j][1] = pointcloud[j].pt.y;
    Xs[j][2] = pointcloud[j].pt.z;
}

cout<<"Read the 3D points."<<endl;

// convert cameras to BA data structs
vector<CameraMatrix> cams(N);
for (int i = 0; i < N; ++i)
{
    int camId = i;
    Matrix3x3d R;
    Vector3d T;

    Matx34d & P = Pmats[i];
    R[0][0] = P(0,0); R[0][1] = P(0,1); R[0][2] = P(0,2); T[0] = P(0,3);
    R[1][0] = P(1,0); R[1][1] = P(1,1); R[1][2] = P(1,2); T[1] = P(1,3);
    R[2][0] = P(2,0); R[2][1] = P(2,1); R[2][2] = P(2,2); T[2] = P(2,3);

    cams[i].setIntrinsic(Knorm);
    cams[i].setRotation(R);
    cams[i].setTranslation(T);
}
cout<<"Read the cameras."<<endl;

vector<Vector2d > measurements;
vector<int> correspondingView;
vector<int> correspondingPoint;

// 2D corresponding points
for (unsigned int k = 0; k < pointcloud.size(); ++k)
{
for (unsigned int i=0; i<pointcloud[k].imgpt_for_img.size(); i++) {
  if (pointcloud[k].imgpt_for_img[i] >= 0) {
    int view = i, point = k;
    Vector3d p, np;

    Point cvp = imgpts[i][pointcloud[k].imgpt_for_img[i]].pt;
    p[0] = cvp.x;
    p[1] = cvp.y;
    p[2] = 1.0;

    // Normalize the measurements to match the unit focal length.
    scaleVectorIP(1.0/f0, p);
    measurements.push_back(Vector2d(p[0], p[1]));
    correspondingView.push_back(view);
    correspondingPoint.push_back(point);
  }
}
} // end for (k)

K = measurements.size();

cout<<"Read "<< K <<" valid 2D measurements."<<endl;

...
Exploring Structure from Motion Using OpenCV

```cpp
pointcloud[j].pt.y = Xs[j][1];
pointcloud[j].pt.z = Xs[j][2];
}
// extract adjusted cameras
for (int i = 0; i < N; ++i)
{
    Matrix3x3d R = cams[i].getRotation();
    Vector3d T = cams[i].getTranslation();
    Matx34d P;
    P(0,0) = R[0][0]; P(0,1) = R[0][1]; P(0,2) = R[0][2]; P(0,3) = T[0];
    P(1,0) = R[1][0]; P(1,1) = R[1][1]; P(1,2) = R[1][2]; P(1,3) = T[1];
    P(2,0) = R[2][0]; P(2,1) = R[2][1]; P(2,2) = R[2][2]; P(2,3) = T[2];
    Pmats[i] = P;
}
```

This code, albeit long, is primarily about converting our internal data structures to and from SSBA's data structures, and invoking the optimization process.

The following image shows the effects of BA. The two images on the left are the points of the point cloud before adjustment, from two perspectives, and the images on the right show the optimized cloud. The change is quite dramatic, and many misalignments between points triangulated from different views are now mostly consolidated. We can also notice how the adjustment created a far better reconstruction of flat surfaces.

For More Information:
Visualizing 3D point clouds with PCL

While working with 3D data, it is hard to quickly understand if a result is correct simply by looking at reprojection error measures or raw point information. On the other hand, if we look at the point cloud itself we can immediately verify whether it makes sense or there was an error. For visualization we will use an up-and-coming sister project for OpenCV, called the Point Cloud Library (PCL). It comes with many tools for visualizing and also analyzing point clouds, such as finding flat surfaces, matching point clouds, segmenting objects, and eliminating outliers. These tools are highly useful if our goal is not a point cloud but rather some higher-order information such as a 3D model.

First, we should represent our cloud (essentially a list of 3D points) in PCL's data structures. This can be done as follows:

```cpp
pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud;

void PopulatePCLPointCloud(const vector<Point3d>& pointcloud,
                            const std::vector<cv::Vec3b>& pointcloud_RGB

    //Populate point cloud
    {
        cout<<"Creating point cloud..."
        cloud.reset(new pcl::PointCloud<pcl::PointXYZRGB>);

        for (unsigned int i=0; i<pointcloud.size(); i++) {
            // get the RGB color value for the point
            Vec3b rgbv(255,255,255);
            if (pointcloud_RGB.size() >= i) {
                rgbv = pointcloud_RGB[i];
            }

            // check for erroneous coordinates (NaN, Inf, etc.)
            if (pointcloud[i].x != pointcloud[i].x || isnan(pointcloud[i].x) ||
                pointcloud[i].y != pointcloud[i].y || isnan(pointcloud[i].y) ||
                pointcloud[i].z != pointcloud[i].z || isnan(pointcloud[i].z) ||
                fabsf(pointcloud[i].x) > 10.0 ||
                fabsf(pointcloud[i].y) > 10.0 ||
                fabsf(pointcloud[i].z) > 10.0) {
                continue;
            }

            pcl::PointXYZRGB pclp;
            // 3D coordinates
```

For More Information:
Exploring Structure from Motion Using OpenCV

```c
pclp.x = pointcloud[i].x;
pclp.y = pointcloud[i].y;
pclp.z = pointcloud[i].z;

// RGB color, needs to be represented as an integer
uint32_t rgb = ((uint32_t)rgbv[2] << 16 | (uint32_t)rgbv[1] << 8 | (uint32_t)rgbv[0]);
pclp.rgb = *reinterpret_cast<float*>(&rgb);

cloud->push_back(pclp);
}

cloud->width = (uint32_t) cloud->points.size(); // number of points
cloud->height = 1; // a list of points, one row of data
}

To have a nice effect for the purpose of visualization, we can also supply color data as RGB values taken from the images. We can also apply a filter to the raw cloud that will eliminate points that are likely to be outliers, using the statistical outlier removal (SOR) tool as follows:

```c
Void SORFilter() {

pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud_filtered (new pcl::PointCloud<pcl::PointXYZRGB>);

std::cerr<<"Cloud before SOR filtering: "<< cloud->width * cloud->height <<" data points"<<std::endl;

// Create the filtering object
pcl::StatisticalOutlierRemoval<pcl::PointXYZRGB> sor;
sor.setInputCloud (cloud);
sor.setMeanK (50);
sor.setStddevMulThresh (1.0);
sor.filter (*cloud_filtered);

std::cerr<<"Cloud after SOR filtering: "<<cloud_filtered->width * cloud_filtered->height <<" data points"<<std::endl;

copyPointCloud(*cloud_filtered,*cloud);
}
```

For More Information:
Then we can use PCL's API for running a simple point cloud visualizer as follows:

```cpp
Void RunVisualization(const vector<cv::Point3d>& pointcloud,  
const std::vector<cv::Vec3b>& pointcloud_RGB) {
  PopulatePCLPointCloud(pointcloud,pointcloud_RGB);
  SORFilter();
  copyPointCloud(*cloud,*orig_cloud);

  pcl::visualization::CloudViewer viewer("Cloud Viewer");

  // run the cloud viewer
  viewer.showCloud(orig_cloud,"orig");

  while (!viewer.wasStopped ())
  {
    // NOP
  }
}
```

The following image shows the output after the statistical outlier removal tool has been used. The image on the left-hand side is the original resultant cloud of the SfM, with the cameras location and a zoomed-in view of a particular part of the cloud. The image on the right-hand side shows the filtered cloud after the SOR operation. We can notice some stray points were removed, leaving a cleaner point cloud:
Using the example code

We can find the example code for SfM with the supporting material of this book. We will now see how we can build, run, and make use of it. The code makes use of CMake, a cross-platform build environment similar to Maven or SCons. We should also make sure we have all the following prerequisites to build the application:

- OpenCV v2.3 or higher
- PCL v1.6 or higher
- SSBA v3.0 or higher

First we must set up the build environment. To that end, we may create a folder named build in which all build-related files will go; we will now assume all command-line operations are within the build/folder, although the process is similar (up to the locations of the files) even if not using the build folder.

We should make sure CMake can find SSBA and PCL. If PCL was installed properly, there should not be a problem; however, we must set the correct location to find SSBA's prebuilt binaries via the `-DSSBA_LIBRARY_DIR=...` build parameter. If we are using Windows as the operating system, we can use Microsoft Visual Studio to build; therefore, we should run the following command:

```
cmake -G "Visual Studio 10" -DSSBA_LIBRARY_DIR=../3rdparty/SSBA-3.0/build/ ..
```

If we are using Linux, Mac OS, or another Unix-like operating system, we execute the following command:

```
cmake -G "Unix Makefiles" -DSSBA_LIBRARY_DIR=../3rdparty/SSBA-3.0/build/ ..
```

If we prefer to use XCode on Mac OS, execute the following command:

```
cmake -G Xcode -DSSBA_LIBRARY_DIR=../3rdparty/SSBA-3.0/build/ ..
```

CMake also has the ability to build macros for Eclipse, Codeblocks, and more. After CMake is done creating the environment, we are ready to build. If we are using a Unix-like system we can simply execute the make utility, else we should use our development environment's building process.

After the build has finished, we should be left with an executable named ExploringSfMExec, which runs the SfM process. Running it with no arguments will result in the following: `USAGE: ./ExploringSfMExec <path_to_images>`

For More Information:
To execute the process over a set of images, we should supply a location on the drive to find image files. If a valid location is supplied, the process should start and we should see the progress and debug information on the screen. The process will end with a display of the point cloud that arises from the images. Pressing the 1 and 2 keys will switch between the adjusted and non-adjusted point cloud.

Summary

In this chapter we have seen how OpenCV can help us approach Structure from Motion in a manner that is both simple to code and to understand. OpenCV’s API contains a number of useful functions and data structures that make our lives easier and also assist in a cleaner implementation.

However, the state-of-the-art SfM methods are far more complex. There are many issues we choose to disregard in favor of simplicity, and plenty more error examinations that are usually in place. Our chosen methods for the different elements of SfM can also be revisited. For one, H and Z propose a highly accurate triangulation method that minimizes the reprojection error in the image domain. Some methods even use the N-view triangulation once they understand the relationship between the features in multiple images.

If we would like to extend and deepen our familiarity with SfM, we will certainly benefit from looking at other open-source SfM libraries. One particularly interesting project is libMV, which implements a vast array of SfM elements that may be interchanged to get the best results. There is a great body of work from University of Washington that provides tools for many flavors of SfM (Bundler and VisualSfM). This work inspired an online product from Microsoft called PhotoSynth. There are many more implementations of SfM readily available online, and one must only search to find quite a lot of them.

Another important relationship we have not discussed in depth is that of SfM and Visual Localization and Mapping, better known as Simultaneous Localization and Mapping (SLAM) methods. In this chapter we have dealt with a given dataset of images and a video sequence, and using SfM is practical in those cases; however, some applications have no prerecorded dataset and must bootstrap the reconstruction on the fly. This process is better known as Mapping, and it is done while we are creating a 3D map of the world, using feature matching and tracking in 2D, and after triangulation.

In the next chapter we will see how OpenCV can be used for extracting license plate numbers from images, using various techniques in machine learning.
References

- *Multiple View Geometry in Computer Vision*, Richard Hartley and Andrew Zisserman, Cambridge University Press
- [http://cvlab.epfl.ch/~strecha/multiview/denseMVS.html](http://cvlab.epfl.ch/~strecha/multiview/denseMVS.html)
- [http://www.inf.ethz.ch/personal/chzach/opensource.html](http://www.inf.ethz.ch/personal/chzach/opensource.html)
- [http://photosynth.net/](http://photosynth.net/)
- [http://pointclouds.org](http://pointclouds.org)
- [http://www.cmake.org](http://www.cmake.org)
Where to buy this book


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.