Scene Reconstruction and Visualization from Internet Photo Collections

Keith N. Snavely

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

University of Washington

2008

Program Authorized to Offer Degree: Computer Science & Engineering

University of Washington
Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Keith N. Snavely

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Chair of the Supervisory Committee:

Steven M. Seitz

Reading Committee:

Steven M. Seitz

Richard Szeliski

Brian Curless

Date:

Signature______________________________

Date__________________________________

University of Washington

Abstract

Scene Reconstruction and Visualization from Internet Photo Collections

Keith N. Snavely

Chair of the Supervisory Committee:
Professor Steven M. Seitz
Computer Science & Engineering

The Internet is becoming an unprecedented source of visual information, with billions of images instantly accessible through image search engines such as Google Images and Flickr. These include thousands of photographs of virtually every famous place, taken from a multitude of viewpoints, at many different times of day, and under a variety of weather conditions. This thesis addresses the problem of leveraging such photos to create new 3D interfaces for virtually exploring our world.

One key challenge is that recreating 3D scenes from photo collections requires knowing where each photo was taken. This thesis introduces new computer vision techniques that robustly recover such information from photo collections without requiring GPS or other instrumentation. These methods are the first to be demonstrated on Internet imagery, and show that 3D reconstruction techniques can be successfully applied to this rich, largely untapped resource. For this problem *scale* is a particular concern, as Internet collections can be extremely large. I introduce an efficient reconstruction algorithm that selects a small *skeletal set* of images as a preprocess. This approach can reduce reconstruction time by an order of magnitude with little or no loss in completeness or accuracy.

A second challenge is to build interfaces that take these reconstructions and provide effective scene visualizations. Towards this end, I describe two new 3D user interfaces. *Photo Tourism* is a 3D photo browser with new geometric controls for moving between photos. These include zooming in to find details, zooming out for more context, and selecting an image region to find photos of an object. The second interface, *Pathfinder*, takes advantage of the fact that people tend to take photos

of interesting views and along interesting paths. Pathfinder creates navigation controls tailored to each location by analyzing the distribution of photos to discover such characteristic views and paths. These controls make it easy to find and explore the important parts of each scene.

Together these techniques enable the automatic creation of 3D experiences for famous sites. A user simply enters relevant keywords and the system automatically downloads images, reconstructs the site, derives navigation controls, and provides an immersive interface.

TABLE OF CONTENTS

LIST OF FIGURES

Figure Number Page

LIST OF TABLES

Table Number Page

ACKNOWLEDGMENTS

I wish to thank Steve Seitz and Rick Szeliski for their guidance, inspiration, and support during my graduate studies. I have learned a great deal from them about research, computer vision, and otherwise, and I deeply cherish the time I have spent working with them. I have also benefited greatly from knowing and working with Brian Curless, Sing Bing Kang, Michael Cohen, and Larry Zitnick while at the University of Washington, and with Greg Andrews and Saumya Debray while an undergraduate at the University of Arizona.

I would also like to thank the numerous current and former UW students and postdoctoral fellows who have helped make my time at the UW such a pleasure. Thanks especially to my collaborators Li Zhang, Pravin Bhat, Ian Simon, Michael Goesele, Colin Zheng, Rahul Garg, Ryan Kaminsky, and Sameer Agarwal, and to Kevin Chiu and Andy Hou for their help with the Pathfinder project. Very special thanks as well to fellow UW students Craig Prince, Alan Liu, and Tyler Robison for their camaraderie, and for helping me to move on so many occasions.

Finally, I wish to thank my parents for their love and support.

DEDICATION

to Yihui

Chapter 1

INTRODUCTION

A photograph is a window into the world. A good photo can depict a place or an event in vivid richness and detail, and can give the viewer a taste of being there. However, a single photo has a fixed boundary in space and time, and, unlike with a real window, we cannot simply adjust our perspective to see what is beyond the frame. Part of the art of photography is working within these limitations to create works of beauty, by capturing just the right moment with just the right composition. But photographs are taken for all sorts of reasons in addition to purely artistic ones. Photos can document and communicate information about people, places, and things in our world, and are widely used in commercial advertising, classified advertisements, and tourism. At a more personal level, photos are used to capture moments in our lives, so that we can later relive memories and share stories with others. For these kinds of applications, where the aim is to capture and convey information, the limitations of a photograph are more significant. For instance, it can be difficult or impossible to give a complete understanding of a large, complex space—a house, Trafalgar Square, the Louvre, the Grand Canyon, the entire city of Rome—or to document your own visit to one of these places, with a single photo.

Of course, there is no reason to limit oneself to a single photo. This may have been the case in the early days of photography, when taking a photo involved unwieldy equipment and a painstaking development process, but the invention of film, pioneered by George Eastman in the last decades of the 19th century, led to cheap, easy-to-use cameras and the advent of the snapshot. The past fifteen years have seen another revolution in photography, as digital cameras have steadily replaced film, making it easy and inexpensive for a single person to take thousands of photos at a time. The digital age has also radically changed the way in which we store, view, and share photos. Where we once stuffed shoeboxes and photo albums with hundreds of photos, we can now store tens of thousands of photos on a hard drive, and organize them on a computer with tools like Picasa [111] and iPhoto [71]. During the same time, the growth of the Internet has taken this trend even further. We can now store

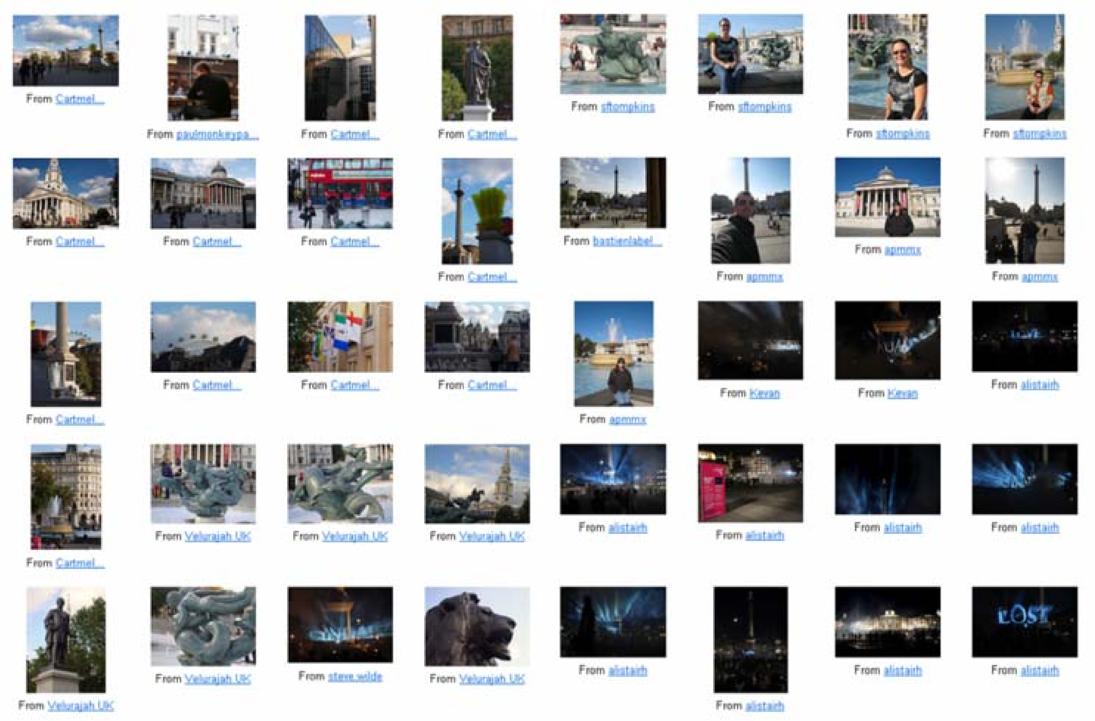


Figure 1.1: The first two pages of results of a Flickr search for “Trafalgar Square.”

virtually unlimited numbers of photos online on sites such as Flickr [47], SmugMug [135], Picasa Web Albums [112], and Facebook [43]. Not only do these sites give us access to our photos from anywhere, but they also give us access to *everyone’s* photos, compounding the number of photos we have at our fingertips many times over.

Thus, if we are selling our house and decide to advertise it online, we need not use just a single photo—why not take a hundred? Similarly, if we want to understand what it is like to experience a famous place like Trafalgar Square, we can go to Flickr, search for “Trafalgar Square,” and find tens of thousands of photos captured from every imaginable viewpoint, by thousands of different people, under many different weather conditions, at many times of day and night, and during all times of the year. We can also find photos of the Square during different events: protests (search for “Trafalgar Square protest”), a visit by the Sultan’s Elephant (“Trafalgar Square elephant”), or Christmas celebrations (“Trafalgar Square Christmas”). We can find similarly rich collections of photos for almost every famous world site. The Internet, with its billions and billions of photos, represents an unprecedented visual record of our world.

How can we use these vast, rich photo collections to effectively communicate the experience of being at a place—to give someone the ability to virtually move around and explore a famous landmark, to see what it looks like at sunrise and sunset or at night, to revisit different events; in short, to convey a real understanding of the scene? Unfortunately, the availability of vast image collections and the ability to capture them are alone not enough to create these kinds of experiences. In fact, with current tools, the more photos there are, the *harder* it often is to make sense of them. Most photo browsing tools treat photos as independent views of events or scenes, perhaps grouped together or labeled in meaningful ways, but otherwise visually disconnected. Such tools do not attempt to exploit or convey the rich common structure that exists *among* photos. For instance, the Trafalgar Square photos from Flickr shown in Figure 1.1 are arranged in a grid of thumbnails, the default way of presenting search results in Flickr, and a nearly ubiquitous display idiom in photo browsing software. This kind of presentation does not facilitate the understanding of the space as a whole. Nor does it really even scale well as a way of simply displaying photos. Only about twenty-five thumbnails are shown on a page, so to see all the photos we would have to view hundreds of pages of results. Finally, the fact that these photos are submitted by many different people is a source of richness and variety, but also a source of disorder. The photos are largely unstructured, and not presented in a particularly meaningful order. Certain tasks, such as finding a specific viewpoint or a detail of a particular object, can be quite difficult.

In contrast, several commercial software applications have started to present large photo collections of urban cityscapes organized in a much more structured way, making it much easier to explore the underlying scene [58, 159, 41]. For instance, Google Street View simulates the experience of walking down the streets of major cities by displaying omnidirectional photos taken at intervals along every city street, as shown in Figure 1.2. Such web applications, combining the high visual fidelity of photos with simplified 3D navigation controls, are useful for recreating the experience of, say, walking around the Haight Ashbury district of San Francisco. However, capturing these experiences typically requires special camera hardware, careful attention to the photo capture process, and time-consuming post-processing and quality control [154], hence they are not easy for casual photographers to create (in fact, Everyscape, a company with a related application, recommends special training courses and equipment to people who want to create their own content [42]). Part

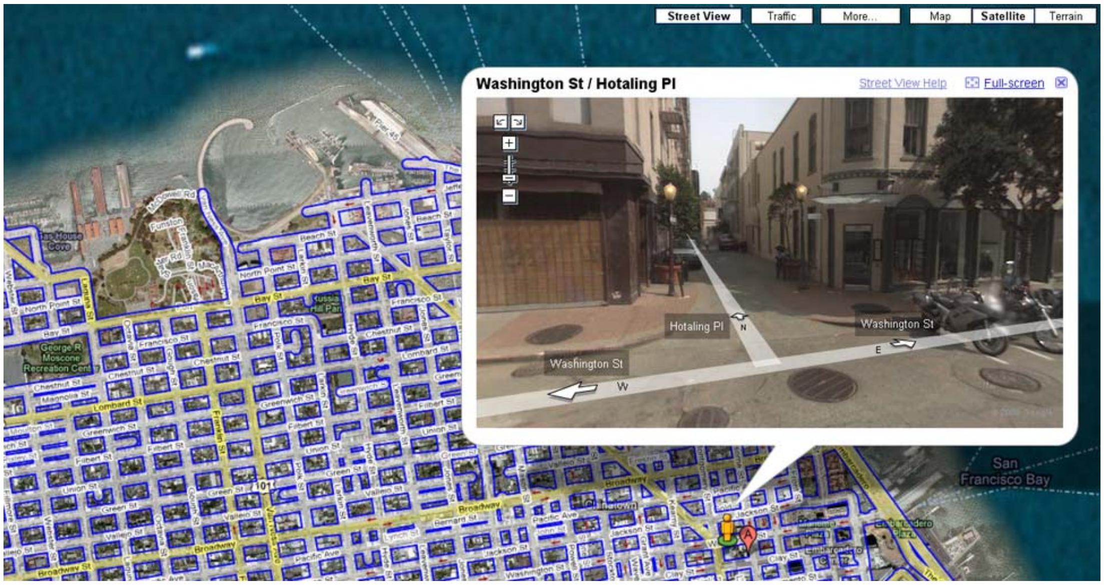


Figure 1.2: Screenshot from Google's Street View, showing an intersection in downtown San Francisco. By clicking on the arrows, a user can virtually move through the city streets.

of the difficulty arises from the need for accurate *location* information for every photo to put each in its proper context.

Is it possible to apply these kinds of 3D scene visualization tools to personal or Internet photo collections? If so, we could not only create 3D experiences for all the world's sites from existing imagery, but we could also potentially create much more powerful tools for browsing photos of places. Unfortunately, the vast majority of these casually acquired collections lack location information, a prerequisite for assembling the photos into a coherent whole.

My solution to this problem is to turn to automatic computer vision techniques. Researchers in the field of multi-view geometry have made great progress towards *automatic* recovery of fine-grained 3D location information for photo collections through the analysis of the photos themselves [68]. However, most previous systems have focused on reconstructing controlled image collections: sequences that are well-ordered in time (such as video) and which consist of images taken under similar conditions and with the same camera. Applying computer vision techniques to the large, fundamentally unordered collections typical of the Internet—taken by many different people, under diverse conditions, and with myriad cameras—raises many new challenges.

Furthermore, adding location information to photo collections is by itself insufficient for scene visualization: we also need intuitive, interactive interfaces for exploring these scenes. There are several challenges faced in the design of such interfaces. First, unlike with Google Street View, where photos are taken at regular intervals, personal or Internet collections are typically an unstructured soup of photos. Nevertheless, the navigation controls should still be intuitive and exhibit regularity. Second, such controls should make it easy to find and explore the *interesting* parts of a scene, particularly for tourist sites.

1.1 Contributions

There are two fundamental research challenges identified above: (1) the computer vision challenge of reconstructing 3D information from Internet photo collections, and (2) the computer graphics challenge of using this 3D information to enable new ways to visualize and explore famous sites. In this thesis, I present a range of new algorithms and techniques in both vision and graphics that address each of these challenges.

Reconstruction of Internet photo collections. To address the computer vision challenge, I have developed new 3D reconstruction algorithms that operate on large, diverse image collections. These algorithms recover both camera pose and scene geometry and demonstrate, for the first time, that 3D geometry can be reliably recovered from photos downloaded from the Internet using keyword search. Hence, they enable reconstruction of the world's famous sites from existing imagery on the web. A few example reconstructions are shown in Figure 1.3, and many more appear in Chapters 3 and 4.

Two major issues faced in handling Internet photo collections are *robustness* and *scale*. We desire computer vision techniques that work reliably across a wide variety of scenes and photo collections, and on collections with large numbers of photos. Robustness is a concern because there are often ambiguities and degeneracies that crop up in multi-view geometry, and because there is no perfect algorithm for finding correspondences between images. These problems are heightened when handling unstructured collections, which lack prior information on the structure of the image sequence. Chapter 3 describes a robust reconstruction pipeline that recovers scene and camera geometry incrementally, carefully building a reconstruction a few images at a time. The

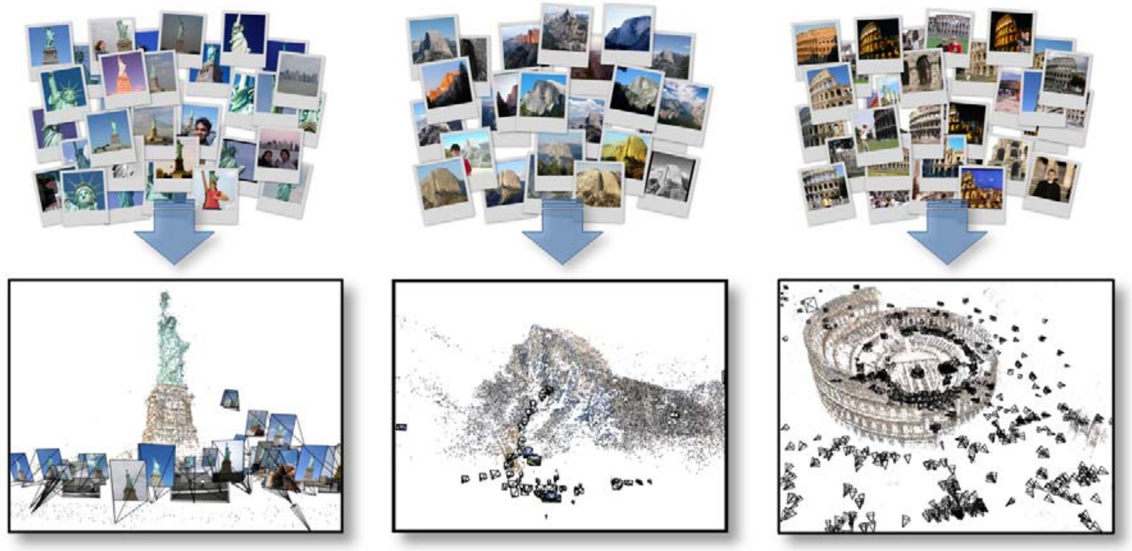
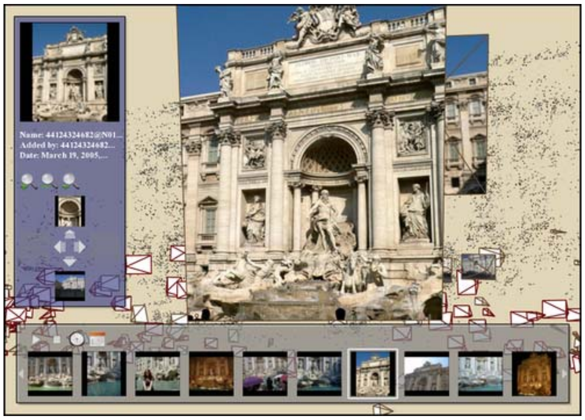


Figure 1.3: *3D reconstructions from Internet photo collections.* My computer vision system takes large collections of photos from the Internet (sample images shown at top) and automatically reconstructs 3D geometry (bottom). The geometry includes camera information—where each photo was taken and which direction it was looking—as well as a *point cloud* of the scene. In these images of reconstructions the recovered cameras are shown as black wireframe pyramids, and the scene is rendered as a point cloud. From left to right: the Statue of Liberty (reconstructed from 388 photos from Flickr), Half Dome in Yosemite National Park (reconstructed from 678 photos), and the Colosseum in Rome (reconstructed from 1,135 photos).

pipeline starts with a pair of images that exhibit strong geometry, giving the reconstruction a solid foundation, then conservatively adds new information until the entire scene model is reconstructed. I demonstrate this pipeline on a number of different photo collections, including urban environments, mountains, and indoor scenes. This system has been used to reconstruct well over one hundred photo collections, by myself and other researchers. In addition, *Photosynth* [109], a commercial photo reconstruction system released by Microsoft and based largely on the work described here, has been applied to thousands of different photo collections by many users.

Chapter 4 addresses the issue of scalability by introducing *skeletal sets*, an approach for improving the efficiency of reconstruction by reducing the amount of redundant information. This technique represents the information present in a photo collection as a graph, and computes a much sparser *skeletal* subgraph that preserves the overall structure of the graph and bounds the amount



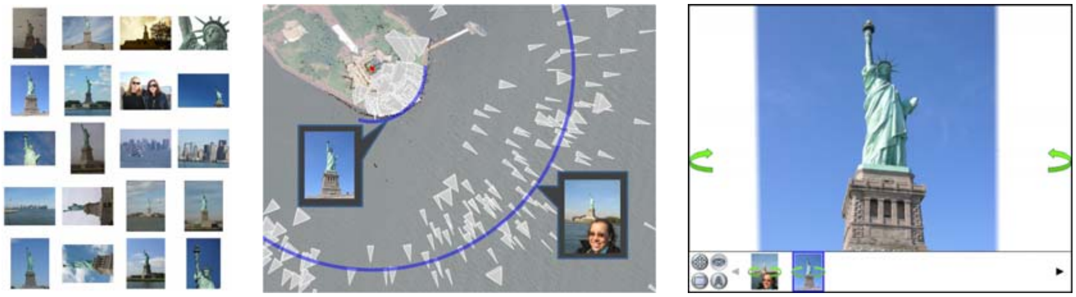
Name: 44124324682@N01...
Added by: 44124324682...
Date: March 19, 2005,...

Figure 1.4: *Photo Tourism.* A screenshot from the Photo Tourism interface, showing a reconstruction of the Trevi Fountain. Information about the currently displayed photo, as well as geometric search tools for finding related photos, appear in panes on the left and bottom of the screen.

of information lost. The hard computational effort of scene reconstruction is then applied only to the skeletal subgraph; the remaining images can be added in later using simple pose estimation techniques. The skeletal sets algorithm reconstructs scenes using many fewer images and reduces reconstruction time by an order of magnitude for large image collections, with little or no loss in reconstruction quality.

Visualization of 3D photo collections and scenes. This thesis also describes computer graphics and interaction techniques for taking these reconstructed scenes and creating new ways to browse photo collections and to visualize the world. For this problem, two primary concerns are (1) how to provide the user with effective navigation controls for exploring scenes and (2) how to display or render scenes using the imagery available in large photo collections. I have designed two interfaces that address these challenges in different ways.

Chapter 5 describes *Photo Tourism*, an immersive 3D photo browser. Photo Tourism situates the user in the 3D scene among the photo collection, as shown in Figure 1.4. The interface provides new geometric controls for moving through the collection: users can zoom in and find details of a given photo, zoom out to see more context, select a region of an image to find a good photo of a specific object, and move in different directions (e.g., to the left or right) to view more images.



(a) (b) (c)

Figure 1.5: *Finding paths through a photo collection*. (a) Several Flickr images of the Statue of Liberty from a collection of 388 input photos. (b) Reconstructed camera viewpoints for this collection, revealing two clear orbits, shown superimposed on a satellite view. (c) A screenshot from the Pathfinder user interface. The two orbits are shown to the user as thumbnails at the bottom of the screen. Clicking on a thumbnail moves the user on an automatic path to the selected control.

The interface also provides powerful annotation features. A user can label an object in one photo and have that label propagate to all other images in the collection. On the rendering side, I introduce new techniques for creating attractive 3D transitions between photos and generating stabilized slideshows that make it easier to see how a scene changes over time.

Chapter 6 describes the second system, *Pathfinder*. While Photo Tourism considers the problem of photo browsing, Pathfinder takes advantage of large Internet photo collections to create an enhanced interface for exploring the scenes themselves. These photo collections, captured by many people, are an extremely valuable source of information for determining good controls for navigating through a scene, as they represent samples of how people actually experienced the scene, where they stood, and what views they found interesting. For instance, Figure 1.5(b) shows a reconstructed collection of Flickr photos of the Statue of Liberty. Even if we knew nothing about the structure or content of this scene, the photos are a powerful indicator of the presence of an interesting object, as they are almost all trained on a single focal point (the center of the statue), and are organized into *orbits* around that point. These orbits are natural controls for exploring the scene. Pathfinder analyzes the distribution of reconstructed photos to derive such controls, and presents them to the user in a fluid, game-like navigation interface, shown in Figure 1.5(c). Pathfinder also introduces a new rendering technique that continuously selects and warps input photos as the user moves through a scene.

Taken as a whole system, my work allows a user to simply specify a search term (“Trevi Fountain,” “Notre Dame Cathedral”) and get back an immersive 3D experience of that place. The system automatically downloads related photos from sites like Flickr, creates a 3D reconstruction, computes a set of custom navigation controls, and provides these controls to the user in an interactive scene browser.

This thesis is divided into two main parts. Part I describes the new computer vision algorithms required to reconstruct 3D geometry from large Internet photo collections. Part II describes the computer graphics, 3D navigation, and user interface innovations that enable new ways to explore photo collections and scenes in 3D. Chapter 7 discusses several directions for future work. Before describing the new algorithms and techniques, I first review related work in computer vision and computer graphics in Chapter 2.

Chapter 2

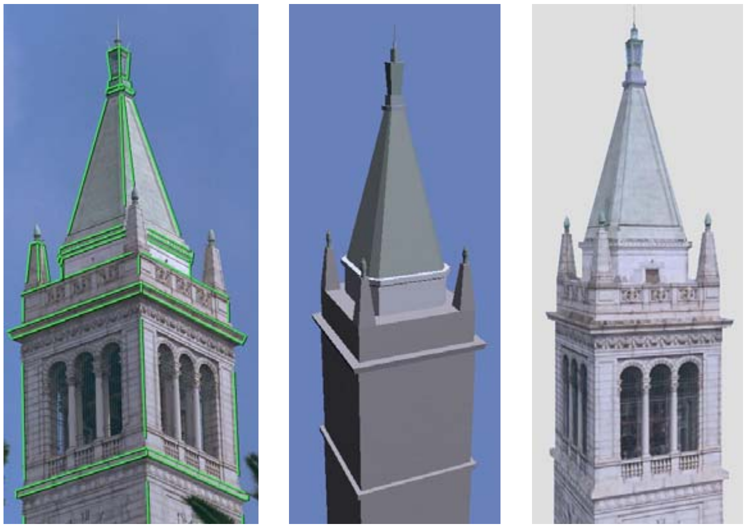
RELATED WORK

The work described in this thesis ultimately has two intertwined goals: (1) to make it easier to browse and explore large collections of photos related by place, and (2) to create new ways of virtually exploring real-world scenes through image collections, a version of the *image-based rendering* problem. Both of these goals rely on the ability to reconstruct geometry automatically from large, unstructured image collections. In this chapter, I review related work on each of these problems: 3D reconstruction from multiple images (Section 2.1), photo browsing (Section 2.2), and image-based rendering (Section 2.3).

2.1 Reconstructing 3D geometry from multiple images

Multi-view geometry, or the recovery of 3D information from multiple images of a scene, has long been an active research area in computer vision and graphics, and has inspired a wide variety of different approaches and algorithms. In computer graphics, the aim is often to acquire geometry suitable for high-quality rendering in a traditional graphics pipeline, i.e. closed, polygonal 3D models, such as those shown in Figure 2.1, with detailed appearance information (texture maps, bi-directional reflectance distribution functions, etc.). This problem is known as *image-based modeling*. Image-based modeling techniques tend to be manual or semi-automatic, as graphics applications often demand a level of photorealism that requires fine control over the quality of the 3D models.

In contrast, I show that *sparse* scene geometry, recovered completely automatically from a photo collection, is often sufficient to produce attractive and comprehensible scene renderings and transitions between photographs. In my approach, the key problem is to robustly and efficiently—and completely automatically—recover camera and sparse scene geometry from large, unorganized photo collections.



(a) (b) (c)

Figure 2.1: *Traditional image-based modeling.* Images from Façade, a semi-automatic image-based modeling system [32]. (a) Images are annotated with lines indicating geometric structures, and (b) these annotations are used to recover a polygonal 3D model. (c) The model (and associated appearance information) can then be used to render new views of the scene.

2.1.1 Structure from motion

The precursor to modern multi-view geometry techniques is the field of photogrammetry—the science of taking 3D measurements of the world through photographs—which began to develop soon after the advent of photography.[1] The key problem in photogrammetry is to determine the 3D location of a point in a scene from multiple photographs taken from different vantage points. This problem can be solved using triangulation if the pose (the position and orientation in the world) of each photo is known. Conversely, if we have a set of points with known 3D coordinates, the pose of photos that observe those points can be determined using a process known as *resectioning*[2] or *camera pose estimation*. But what if we know neither the camera poses nor the point coordinates? Estimating both at once seems like a chicken-and-egg problem. The traditional photogrammetric

[1] Although an important technique in photogrammetry, triangulation, has been known since antiquity.

[2] The word "resection" is by analogy with computing the "intersection" of rays. To triangulate a 3D point, we can observe it from two or more known locations, and *intersect* the appropriate rays. Conversely, another way to find the coordinates of a 3D point is to stand at that point and observe two or more points with known coordinates. We then *resect* the rays to the known points to find our location. This is a simplified form of camera pose estimation.

solution to this problem is to place a set of reference fiducial markers at known 3D positions in the scene, manually identify them in each image, and use these to recover the poses of the cameras through resectioning. Once the camera poses are known, additional unknown points can be identified in each image and triangulated.

Over time, the requirements of this traditional approach have steadily been relaxed, due to advances in both computer vision and photogrammetry. Correspondences between image points can often be computed completely automatically without the need for fiducials, and algorithms have been developed for computing camera pose and scene structure *simultaneously*, without requiring either to be known *a priori*. This is known in computer vision as the structure from motion (SfM) problem.

The simplest version of the SfM problem, involving just two images, has been extensively studied. Nearly one hundred years ago, Kruppa proved that for two views with five point correspondences, the camera poses and 3D point locations can be determined (up to a similarity transform) [81], and based on this result several *five-point* algorithms for estimating two-view geometry have recently been developed [102, 86]. The mathematical and algorithmic aspects of the three-view problem have also received a great deal of attention [68]. For larger numbers of images, however, the problem becomes more difficult. For some specific scenarios the multi-image SfM problem can be solved exactly, but for the general case no such closed-form solutions exist, and a wide variety of different algorithms have been proposed.

One common multi-image scenario is that the images are given in a particular sequence, such as the frames of a video or images captured by a robot moving through a building. Assuming that one can match corresponding points between consecutive images, a simple way to reconstruct such a sequence is to use the five-point algorithm to recover the poses of the first two frames and triangulate their common points, then resection the next frame, triangulate any new points, and repeat: a form of visual dead reckoning. With well-calibrated cameras, this technique can perform well, particularly on short image sequences [102]. However, the main problem with this approach is that errors in pose estimates will accumulate over time. Better results can be obtained by propagating information forwards and backwards within a window of frames, e.g., through Kalman filtering and smoothing [94].

These local techniques are very efficient, and work well for camera trajectories that are simple, i.e., that do not double-back or form loops. For complex paths, much more accurate geometry can be obtained by using global optimization to solve for all the camera poses and point positions at once. In certain specific cases, this can be done in closed form using *factorization*. Originally proposed by Tomasi and Kanade [148], factorization methods are based on the observation that for orthographic cameras, the $2n \times m$ observation matrix W, formed by stacking rows of zero-mean observations for each camera, can be factored into the product of a $2n \times 3$ matrix R whose rows represent camera rotations, and a $3 \times m$ matrix S whose columns are the 3D positions of the points. The factorization $W = RS$ can be computed using singular-value decomposition. Factorization methods have been extended to more general camera models, such as paraperspective [114] and perspective cameras [141, 23], but have a number of drawbacks. First, for perspective camera models, factorization methods minimize an algebraic error function with no direct geometric interpretation. Second, in order to directly factor the observation matrix W, all of its entries must be known, i.e., all points must be visible in all images. In most real-world situations, some—indeed, most—of the observations are unknown, due to occlusions or missing correspondences. When there is missing data, no known closed-form solution to factorization exists, although several iterative schemes have been proposed [148, 67]. Finally, it is difficult to incorporate priors and robust error functions into factorization methods [16], extensions which can be critical for handling noisy correspondences contaminated with outliers.

Bundle adjustment [150] is an alternative to factorization that can easily handle missing data and accommodate priors and robust objective functions. Bundle adjustment[3] seeks to minimize a geometric cost function (e.g., the sum of squared reprojection errors) by jointly optimizing both the camera and point parameters using non-linear least squares [143]. Given a known measurement noise distribution, bundle adjustment is the statistically correct way to find the parameters, and is regarded as the *gold standard* for performing optimal 3D reconstruction [68]. However, bundle adjustment has no direct solution and involves minimizing a non-linear cost function with potentially many local minima. Therefore, bundle adjustment requires careful initialization, which can be difficult to obtain.

Another disadvantage to bundle adjustment is that it can be quite slow for scenes with large num-

[3] The name “bundle adjustment” stems from the idea of adjusting “bundles” of light rays from scene points to camera centers to align them with the observations.

bers of cameras and points. While bundle adjustment problems have a particular form that makes them amenable to sparse matrix techniques [143], even approaches that take advantage of sparsity (e.g., [15] and [89]) can become very slow when the number of cameras is large. Triggs *et al.*found empirically that sparse bundle adjustment methods can have approximately cubic complexity in the number of cameras [150].

Faster SfM methods based on bundle adjustment have been designed for ordered image sequences, such as video, where several simplifying assumptions can be made. For instance, a common assumption is that the information gained by adding a new video frame only affects a small number of frames immediately preceding the new frame. As a result, only a constant number of camera parameters, and the points they observe, must be adjusted whenever a new frame is added. This assumption is the basis for several real-time systems for SfM from video [116, 103, 25, 115]. Engels *et al.* [39] analyzed the inherent tradeoff between speed and accuracy for this type of approach when considering how many frames to optimize, and Steedly and Essa proposed a more principled algorithm that considers the flow of new information in a system when determining the set of parameters to update [140].

Other *bottom-up* systems improve performance using a divide-and-conquer approach. In the work of Fitzgibbon and Zisserman [46], a video is divided into small, three frame subsequences, which are reconstructed independently, then merged into progressively larger reconstructions. Nistér [101] generalized this approach to partition the video into contiguous subsequences of irregular length, resulting in better-conditioned reconstructions. Steedly *et al.* [139] proposed a technique for refining an initial reconstruction by segmenting it into clusters using spectral partitioning, then treating each cluster as a single rigid body during optimization, dramatically decreasing the number of parameters to estimate. Ni *et al.*, also use segmentation to enable a fast, out-of-core bundle adjustment technique [100].

These bottom-up techniques are effective because of the spatial continuity of video; dividing a video into contiguous subsequences is a reasonable way to partition the frames. Unfortunately, photo collections, especially those found through Internet search, often have much less structure than video sequences. Typically, search results are presented in order of relevance, or other criteria, such as date, which have little to do with spatial continuity. It can therefore be difficult to apply fast SfM methods designed for video to these kinds of collections. Recently, a few researchers, including

Schaffalizky and Zisserman [124], Brown and Lowe [15], and Vergauwen and Van Gool [153] have considered the problem of applying SfM to such unordered photo collections. Martinec *et al.* [91], presented a way to use estimates of relative pose between pairs of cameras to robustly estimate a set of globally consistent camera poses. However, these techniques have largely been demonstrated on relatively small and controlled sequences of photos captured by a single person. Internet photo collections can be orders of magnitude larger than collections that have been considered previously, and have fundamentally different properties due to the distributed nature of their capture. In my work, I demonstrate new algorithms that can robustly reconstruct Internet photo collections of up to several thousand images.

2.2 Photo browsing and visualization

Over the past few years, many researchers have sought better computational tools for organizing, visualizing, and searching through large collections of digital photos. Most have focused on large personal photo collections, but others have considered the problem of Internet collections. In this section, I survey previous work in the area of photo browsing and visualization.

Large-scale collections of media can be very challenging to interact with and to visualize. A person can easy accumulate thousands of photos, far more than can be viewed in a comprehensible way all at once. As the volume of media increases, tasks such as getting an overview of a collection, organizing or structuring a collection, finding a specific photo, or finding a set of photos with certain characteristics become more difficult and require more sophisticated tools. Many proposed tools in the photo browsing literature can be decomposed into two components: (1) a method to filter or sort the images and (2) a method to display and interact with the results of the sorting and filtering operations. Many of these sorting and display techniques have found their way into commercial photo browsers, such as Google’s Picasa [111] and Apple’s iPhoto [71].

Sorting and filtering photos. Many approaches to sorting and filtering focus on either (a) making it easy to organize the photos into meaningful structures or (b) creating better tools for photo search. For many people, the default method for organizing photos is to use the file system to arrange the photos in a directory tree, where photos in each node can be sorted by attributes such as time. Some tools, such as the PhotoTOC system [113], attempt to automatically create a meaningful hierarchy by clustering photos into *events* based on time stamps. Girgensohn *et al.* [53] also cluster photos into events, but provide multiple, parallel hierarchies based on the people and places featured in the images.

Other work addresses the problem of organization through search. The PhotoFinder system [75] represents the photo set as a database and presents an intuitive interface for formulating complex database queries (involving attributes such as time, people, location, and rating). Tagging, or annotating photos with text, has also emerged as a useful tool for managing personal photo collections [82]. Tags are a flexible way to organize photos in a way that makes sense to a particular user, without enforcing a strict hierarchical structure; tags are also easily searchable. Several researchers have sought easy ways to apply tags to large numbers of personal photos, for instance through drag-and-drop interfaces [129], automatic tagging based on contextual information [29], or efficient selection mechanisms [37]. Others have addressed the problem of tagging Internet photos, through collaborative tools such as LabelMe [123] or games such as the ESP Game [155]. Tagging is also an important capability of many Internet photo-sharing sites, including Flickr [47] and Picasa Web Albums [112].

Displaying photos. Other research has focused on the display of large photo collections. Most photo browsing software displays photos using grids of thumbnails or slideshows, which do not always scale well. Several other 2D display techniques have been proposed, such as calendar and scatter plot views [75], alternative thumbnail layouts [9], and zoomable browsers including PhotoMesa [9] and Seadragon [127]. The MediaBrowser [37] also has a timeline mode where photos are displayed in 3D stacks.

Using image content. Another line of research has looked at using the *content* of photos to improve the search and display of image collections. While a complete survey of content-based search tools is outside the scope of this thesis, one particularly relevant project is the Video Google system [133], which enables a user to find all the frames of a video in which a certain object appears by selecting it in a single frame.

Using location information. More relevant to my research is work that uses *location* information to assist in organizing, querying, and displaying photo collections. In most previous work, absolute location, obtained through GPS or manual tagging, is assumed to be provided in advance for each photo; such photos are referred to as being *geo-referenced* or *geo-tagged*. Some approaches have used location to aid in clustering photos in a personal collection. For instance, Naaman *et al.* [98] recognize that an event (such as a birthday party) has an extent in both space and time, and present a system, Photocompas, that uses time and location to compute better event clusterings.

In the LOCALE project [97], Naaman *et al.* explore how location can be used to enable automatic tagging of photos. LOCALE uses a database of tagged, geo-referenced photos to transfer tags to new geo-referenced photos based on proximity. Large Internet photo collections have also been used to infer semantic information about scenes. For instance, several researchers have used clustering to segment collections of tagged photos into separate places (e.g., “Coit Tower” and “Transamerica Building”), and then infer tags describing each place, using location information alone [72], visual content [130, 73], or both [77, 118]. These systems also select a set of summary images useful for getting a quick overview of a collection.

Others have integrated maps into tools for interacting with geo-tagged photos. One such project is GTWeb [138], which produces a webpage summary of a trip by merging time-stamped photos with information from a GPS track, a digital map, and a landmark database. In the domain of Internet photo collections, the World-Wide Media eXchange (WWMX) [149] allows users to share geo-referenced photos and search for and view photos on a map. Large-scale commercial applications, such as Flickr [47] and Google Earth [57], have recently adopted similar interfaces for visualizing geo-referenced photos. Screenshots from WWMX and Flickr Maps are shown in Figure 2.2.

Relatively little work has been done on *immersive* photo browsing tools, where photos are situated directly in a 3D environment. This is perhaps partly because in order for such tools to be most effective, each photo should be localized well enough to be visually aligned with its subject to within a few pixels (i.e., *pixel-accurate* camera pose is needed), which requires more positional accuracy than can be reliably obtained from GPS or manual methods. (In addition, such applications require accurate orientation information, requiring an electronic compass or similar device.) Nevertheless, a few projects have taken steps towards immersion. Realityflythrough [93] is a method for browsing a live scene using a set of location-aware video cameras. A user can move around the scene by

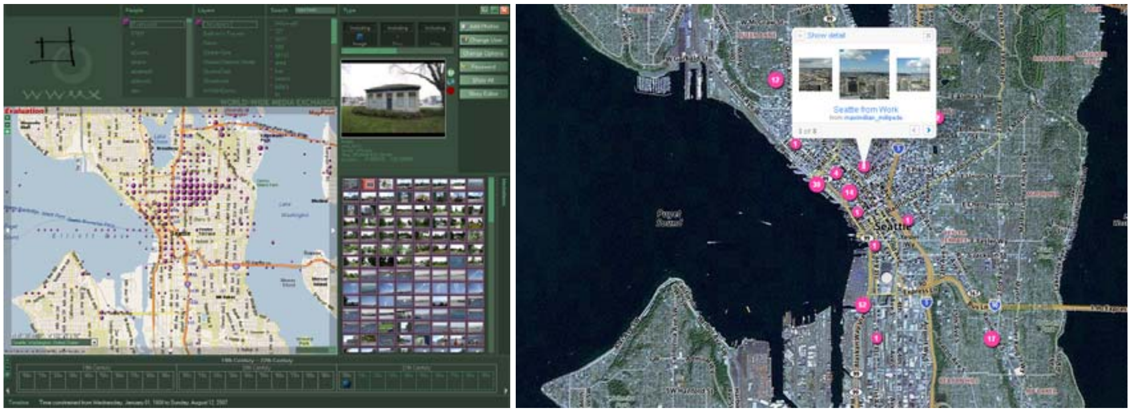


Figure 2.2: Two 2D interfaces for browsing photos with location. Left: the World-Wide Media eXchange (WWMX); right: Flickr Maps

selecting different videos, and smooth transitions provide context when moving from one video to another. While Realityflythrough allows a user to explore the scene by selecting among video feeds, the work of Kadobayashi and Tanaka [74] enables the opposite functionality: a user moves a virtual camera through a 3D model of a real scene (such as a archaelogical site), then queries an image database to find photos taken near the current virtual camera position.

Another tool, PhotoWalker [146], allows a user to create a walkthrough of a scene from a collection of photos by manually linking and specifying morphs between pairs of photos. When a user is viewing the collection, links are displayed as highlighted regions superimposed on an image; clicking on one of these regions activates a morph to the next photo. This tool can give the illusion of a 3D world, although only 2D morphs are used. Similarly, Sivic *et al.* use 2D morphs between images unrelated by a specific place (for instance, generic city images) to create the illusion of a large virtual space [132]. Other projects have made more explicit use of 3D information, including the 4D Cities project [125], which has the goal of creating an interactive 4D model of the city of Atlanta from a collection of historical and modern photographs; and Photosynth [109], a web-based tool for creating and exploring 3D photo collections, which is based on the Photo Tourism project described in Chapter 5.

These immersive 3D (or pseudo-3D) photo browsers use aspects of 3D navigation, but arguably do so with the aim of making it easier to browse a photo or media collection. There is also a large

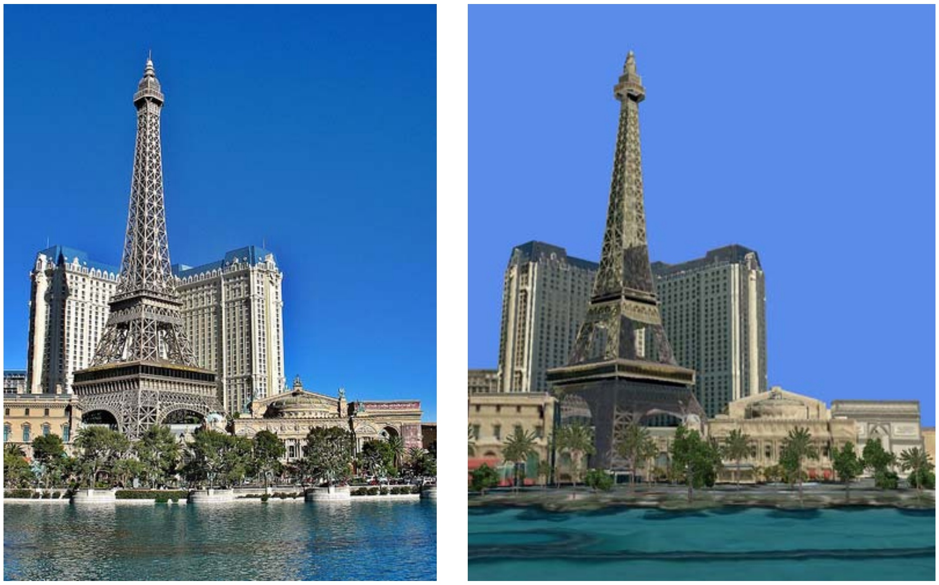


Figure 2.3: *Comparison of a computer-generated rendering with a photograph.* Right: a photograph of the Paris Las Vegas casino. Left: a screenshot from Microsoft's Virtual Earth 3D from roughly the same viewpoint. While these views show (mostly) the same objects, the photograph is more vivid and lifelike, capturing the ripples and reflections in the water, sharper detail in the buildings, and realistic shadows and other lighting effects. There are other, more subtle differences as well. For instance, in the photograph, the color of the sky at the horizon is different from the color closer to the zenith.

body of work that starts from the opposite direction, where the goal is to recreate virtual 3D versions of real-world scenes using collections of images or video. This approach to capturing and displaying the world is known as *image-based rendering*.

2.3 Image-based rendering

A primary goal of computer graphics is to recreate the experience of *being there*, giving a user a sense of being in another place, whether it be a real-world location (such as the Taj Mahal), or a fictional world (Middle Earth). Two critical aspects of this problem are (a) rendering, or visually depicting the scene, and (b) navigation, or the mechanisms by which the user moves around in and explores the scene.

Approaches to rendering and navigation depend fundamentally on the way in which the virtual scene is created and represented. The traditional computer graphics approach to scene representation, dominant in 3D computer animation and games, is to explicitly model the 3D geometry and appearance of the scene in full detail. However, creating good 3D models often requires difficult, time-consuming manual effort, despite the progress in automatic and semi-automatic modeling described in Section 2.1. Furthermore, there can be a lack of verisimilitude in renderings produced using 3D models, especially in real-time applications. Figure 2.3 shows a Virtual Earth [88] rendering of the Paris Las Vegas casino juxtaposed with a photograph taken from the same viewpoint. While the rendering is quite good, and similar in many respects to the photo, it is still possible to tell the two apart.

Another very promising approach to creating the experience of "being there" involves capturing the appearance of the desired scene through photos or video, then replaying or resampling the captured images in an interactive viewer. This approach is known as image-based rendering (IBR) and has inspired a large body of research over the past fifteen years. This section reviews prior work in IBR and related fields.

Moviemaps. A forerunner to the field of image-based rendering was Andrew Lippman's seminal Aspen Moviemap project from the late 1970's [87]. The Aspen Moviemap used a set of images taken throughout the town of Aspen, Colorado and stored on laserdisc to create an interactive "surrogate travel" application. The goal of the project was, in the words of Lippman, "to create so immersive and realistic a 'first visit' that newcomers would literally feel at home, or that they had been there before." [99]

To capture the town on laserdisc, a crew drove a car mounted with cameras through every street in town, taking photos every ten feet. The cameras were arranged so as to capture views in all directions. The resulting data was then processed by hand to turn it into an immersive, interactive travel experience, where a user could virtually drive down the streets of Aspen. The interface was relatively simple yet intuitive: controls for moving forward and turning were shown on a touchscreen display (see Figure 2.4). The moviemap was not only an effective new medium in itself but was also "conceived of as a backbone on which a comprehensive audio-visual survey of Aspen could be spatially organized." [99] Various other media, including still images of every building in town (shot in both summer and winter), historical photos, documentary video, and sound, were accessible through hyperlinks from relevant parts of the moviemap.

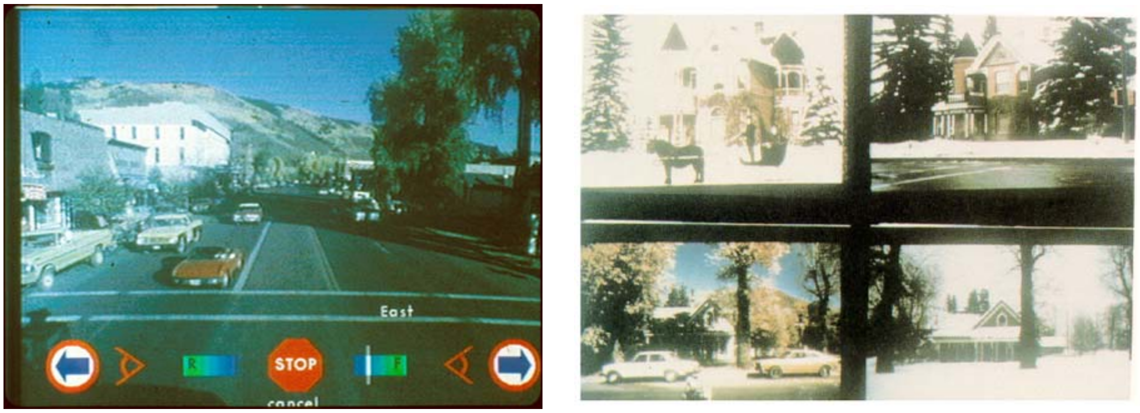


Figure 2.4: Screenshots from the Aspen Moviemap. Left: the interface for virtually driving through the city. Right: a few still photographs from different seasons hyperlinked to the moviemap.

Although the Aspen Moviemap was an effective means of virtual travel, it required a substantial amount of time to capture and process—more than a year of effort by a “small squadron” of at least a dozen people [99]. Only recently have companies, such as Microsoft and Google, begun to create such surrogate travel applications on a large scale. Though some advances, such as GPS, have made the process easier, the preparation, time, and resources required to create such experiences is still significant.

The Aspen Moviemap simply plays back existing video frames based on a user’s actions, resulting in an experience where the user moves discretely between views. Much of the more recent work in image-based rendering has focused on generating a continuum of new views from a discrete database of existing images. These techniques have been applied to different scenarios of varying complexity. The QuickTime VR system [21] renders new views of a scene from a fixed position, but where the camera can rotate and zoom freely; recent work has extended this idea to video panoramas [3], multi-viewpoint panoramas [2], and gigapixel images [79]. Approaches based on explicit sampling of a 4D light field [84, 59], allow for free motion of a camera within a volume of viewpoints. When per-pixel scene geometry or correspondence is known, methods based on warping and combining images can be used to generate in-between views [20, 95]. An alternative approach, which also uses scene geometry, is view-dependent texture mapping [32], in which polygons in the scene are texture mapped with several weighted textures derived from a set of captured images. The weights are based on how close the current viewpoint is to each captured view. Finally, unstructured lumigraph rendering [17] generalizes many of these previous methods with a framework that can handle a variety of arrangements of image samples, with or without known geometry.

Researchers have also developed techniques for creating walkthroughs of large, complex spaces. Arbeeny and Silver [6] created a system for registering video of a scene walkthrough with a 3D model, allowing for spatial navigation of the video stream. Plenoptic stitching [4] is a method for generating novel views of a large-scale scene by resampling frames of omnidirectional video captured on a rough 2D grid throughout the space, allowing a user to walk anywhere within the captured scene. Other approaches, including Taylor's VideoPlus project [147] and Google's Street View [58], have revisited the moviemap's "nearest neighbor" approach for resampling images, but use omnidirectional images to allow for continuous panning and tilting of the camera. Uyttendaele, *et al.* [152] propose a similar system using omnidirectional video, and augment their walkthroughs with additional information, including overhead maps, explicit branching points where the video stream crosses itself in space, and other media such as video textures.

These recent systems bring us around nearly full circle to the end of Section 2.2, which described photo browsers that incorporate elements of 3D navigation. Here we have a number of systems for 3D walkthroughs that are constructed from photos or video. While researchers have explored a wide variety of different design points in image-based rendering (and 3D photo browsing), most IBR systems are created using carefully captured imagery taken along planned paths through a space. In this thesis, I focus on large Internet photo collections, which are captured in a highly distributed manner by uncoordinated individuals, and are thus fundamentally unplanned and unstructured. In Part II of this thesis, I address the open question of creating effective renderings and navigation controls—and tying these together into a understandable experience of a scene—for such unstructured photo collections.

Part I

RECONSTRUCTION OF UNORDERED PHOTO COLLECTIONS

The ultimate goal of my work is to take a collection of photos related by place, and create an immersive 3D experience where a user explores the photo collection and the underlying 3D scene. As a prerequisite to creating this kind of experience, we first need to recreate a 3D version of the scene and localize the cameras within that scene. In particular, we need to know where each photograph was taken and in which direction the camera was pointed, as well as internal camera settings, such as zoom, which affect how incoming light is projected onto the film or sensor plane. I will refer to photos for which such information has been determined as being *registered*.

How can we register our photos? One solution is to use special hardware. We could equip our cameras with a Global Positioning System (GPS) device and electronic compass, and use them to stamp each photo with a position and orientation. Unfortunately, the vast majority of existing photographs were taken without this kind of specialized hardware, and most digital cameras still lack such intrumentation.[4] Moreover, the visualization techniques I have developed require *pixel-accurate* registration, meaning that if the photograph were retaken from the recorded position and orientation, the original and new photo should be in alignment within a few pixels. The accuracy of GPS depends on a number of factors (atmospheric effects, the presence of nearby walls, etc.), but many manufacturers of GPS receivers report a typical accuracy of around 3-5 meters [1]. Localization errors on the order of several meters are much too large to guarantee pixel-accurate registration. Furthermore, GPS works poorly indoors.

What about internal camera parameters such as zoom? These can be determined by calibrating a camera, which usually involves taking several photos of a known pattern such as a checkerboard, as with the calibration systems of Zhang [160] and Bouguet [13]. Again, however, most existing photographs have been taken with cameras that have not been calibrated, and by people who may

[4] A few camera models with built-in GPS, such as the Ricoh 500SE, the Apple iPhone 3G, and the Nikon P6000, are beginning to appear.

not be willing to invest time in calibration.

Because most existing photos lack the necessary metadata for registration, in my work I do not rely on the camera or any other piece of equipment to provide location information, nor do I assume that calibration information is available for each image. Instead, I develop a set of algorithms that derive camera geometry directly from the images themselves using computer vision techniques. In addition to camera geometry, my system also recovers scene geometry in the form of a sparse set of 3D points, or “point cloud.” In this thesis, I refer to the camera and scene geometry recovered from a photo collection as a *reconstruction*. Example reconstructions are shown throughout the next two chapters. The computer vision techniques I use only recover *relative* camera positions and scene geometry which are not grounded in an absolute coordinate system (e.g., latitude and longitude)—the system can register all the cameras with respect to each other, but cannot determine where in the world the reconstruction sits, unlike with GPS.[5] However, this relative pose information is sufficient for most of the visualization techniques described in Part II. Chapter 3 also describes several methods for obtaining absolute coordinates.

In the next two chapters, I describe the computer vision algorithms I have developed for reconstructing camera and scene geometry from unordered photo collections. Chapter 3 gives an overview of my basic reconstruction pipeline, and presents results on eleven Internet and personal photo collections. The basic algorithm can operate on collections of up to about a thousand images, but does not scale well beyond that. To address the scalability problem, Chapter 4 introduces the idea of the *skeletal graph*, a concept which can be used to dramatically prune the number of images required for reconstruction while maintaining guarantees on completeness and accuracy. This technique allows for much more efficient reconstruction of large photo collections.

[5] These reconstructions satisfy a slightly different definition of “pixel-accurate” than the one described above: reconstructed 3D points, when projected into a reconstructed camera, lie close to where that 3D point actually appears in the image.

Chapter 3

A SCENE RECONSTRUCTION PIPELINE

My reconstruction pipeline takes an unordered collection of images (acquired, for instance, from Internet search or a personal photo collection), and produces 3D camera and scene geometry. In particular, for each input photo, the pipeline determines the location from which the photo was taken and direction in which the camera was pointed, and recovers the 3D coordinates of a sparse set of points in the scene (a "point cloud"). While the problem of 3D reconstruction from images has been well-studied, previous work has mainly focused on ordered sequences, such as video. Reconstruction of Internet photo collections poses a particular challenge, as the images are given in more or less random order, are taken with many different cameras, exhibit significant variation in lighting and weather, and can be extremely large in number. Thus, my pipeline is carefully designed to robustly handle these kinds of uncontrolled input collections.

The basic principles behind recovering geometry from a set of images are fairly simple. Humans and other animals implicitly use multi-view geometry to sense depth with binocular vision. If we see the same point in the world (the corner of a window, say) in both eyes, we can implicitly "triangulate" that point to determine its rough distance.[1] This form of depth perception depends on two key faculties: (1) identifying parts of the images seen in the left and right eyes that correspond to the same point in the world, and (2) knowing where the eyes are roughly located relative to each other (to enable triangulation of corresponding points).[2] The human visual system

Similarly, given two photographs of the same (motionless) scene, a list of pixels in image A and their corresponding pixels in image B, and the relative poses (i.e., the position and orientation) of the cameras used to capture the images, one can calculate the 3D position of the point in the world associated with each pair of matching pixels. To do so, we shoot a 3D ray from each camera

[1] Binocular vision is only one of a number of cues we can use to estimate depth. Others include focus, parallax induced by motion of the head, and the apparent size of objects.

[2] Hence, by optically increasing this interocular distance, e.g., through a carefully engineered system of mirrors [26], our perception of distance can be altered.

location through each respective matched pixel, and find where the two rays intersect. If there is any noise in the correspondences or the camera poses, the two rays may not intersect exactly, but we can compute the point with the smallest distance to the two rays. The same procedure can be extended to any number of images, as long as we know the pose for each camera, and as long as each point we wish to triangulate is identified in at least two images.

Now suppose we want to automate this process. We first take our digital camera and walk around an object (a priceless Peruvian idol, say) shooting photos from different angles. Our goal is to create a rough 3D model of the object, and so we upload the photos to a computer to go through the calculations described above. However, we run into a problem: where do we get the list of matching points, and how do we know the vantage point from which each image was taken? Unlike the human brain, the computer does not have an innate ability to match points between different images, nor does it know the poses of the cameras used to take the photos. These are two fundamental problems in computer vision.

The first problem, that of matching 2D points between images, is known as the *correspondence problem*. There are many automatic techniques for finding correspondences between two images, but most work on the principle that the same 3D point in the world (the left eye of the idol, for instance) will have a similar appearance in different images, particularly if those images are taken close together.

Considering the case of two photos for a moment, suppose we can solve the correspondence problem and identify correct pixel matches between two photos. How can we determine the poses of the cameras? As it turns out, the correspondences place constraints on the physical configuration of the two camera.[3] For instance, the two cameras must have been situated in such a way that rays through corresponding pixels actually intersect (or nearly intersect, given noise in the system). This is a powerful constraint, as two 3D rays chosen at random are very unlikely to pass close to one another. Thus, given enough point matches between two images, the geometry of the system becomes constrained enough that we can determine the two camera poses (up to a similarity transformation) [81], after which we can estimate the 3D point positions using triangulation. The problem of using pixel correspondences to determine camera and point geometry in this manner is known

[3] I refer to each image as being taken by a different "camera," meaning a specific 3D pose and set of camera parameters, even if the same physical device is used from photo to photo.

as *structure from motion* (SfM). In general, SfM methods take an arbitrary number of images and an arbitrary number of correspondences, and estimate camera and point geometry simultaneously. Such methods typically work by finding the configuration of cameras and 3D points which, when related through the equations of perspective projection, best agree with the correspondences.

This chapter describes an SfM pipeline specially designed to handle diverse collections of photos resulting from Internet search, and consists of two main stages, corresponding to the correspondence and SfM problems. First, a set of pixel correspondences among all images is determined through feature detection and matching. Second, an incremental SfM algorithm uses the correspondences to estimate where the images were taken and the positions of a sparse set of 3D scene points. The pipeline is shown as a block diagram in Figure 3.1.

The remainder of this chapter is organized as follows. Section 3.1 describes the correspondence estimation stage, and 3.2 describes the incremental SfM algorithm. Section 3.3 discusses the time and space complexity of these stages. Section 3.4 describes how reconstructions can be aligned to a geocentric coordinate system, and Section 3.5 describes how recovered geometry is processed to prepare it for viewing. Finally, Section 3.6 shows results for eleven data sets, Section 3.7 discusses the limitations and breaking points of the system, and Section 3.8 offers concluding remarks.

3.1 Finding correspondence

The input to the correspondence estimation stage is the collection of raw images. The goal of this stage is to find sets of matching 2D pixel positions among the input images. Each set of matching pixels across multiple images should correspond to a single point in 3D, i.e., each individual pixel in a matched set should be the projection of the same 3D point. I will refer to such a corresponding set of pixels as a *point track*. The name “track” is in analogy to tracking features across a video sequence. Unlike with video tracking, however, where tracks are computed across sequences of consecutive frames, tracks in the unordered setting may contain points in images widely distributed throughout the input set.

The basic approach for finding correspondences is to identify features in different images which have a similar appearance; two image patches that look very similar likely correspond to the same 3D point in the world. The correspondence estimation stage has three main steps: (1) find distinctive

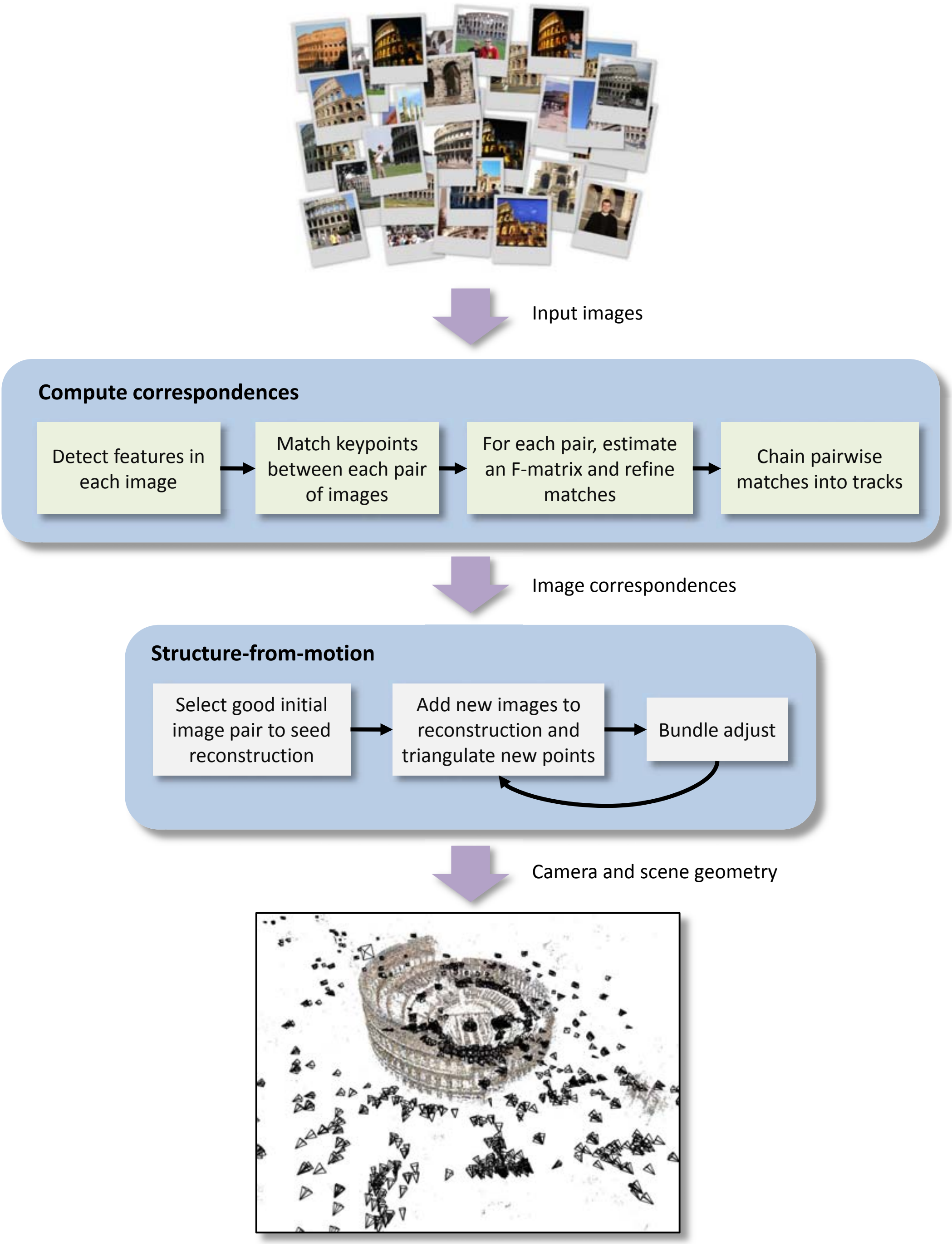

Input images
Compute correspondences
Detect features in each image
Match keypoints between each pair of images
For each pair, estimate an F-matrix and refine matches
Chain pairwise matches into tracks
Image correspondences
Structure-from-motion
Select good initial image pair to seed reconstruction
Add new images to reconstruction and triangulate new points
Bundle adjust
Camera and scene geometry

Figure 3.1: *Block diagram of the structure from motion pipeline.*

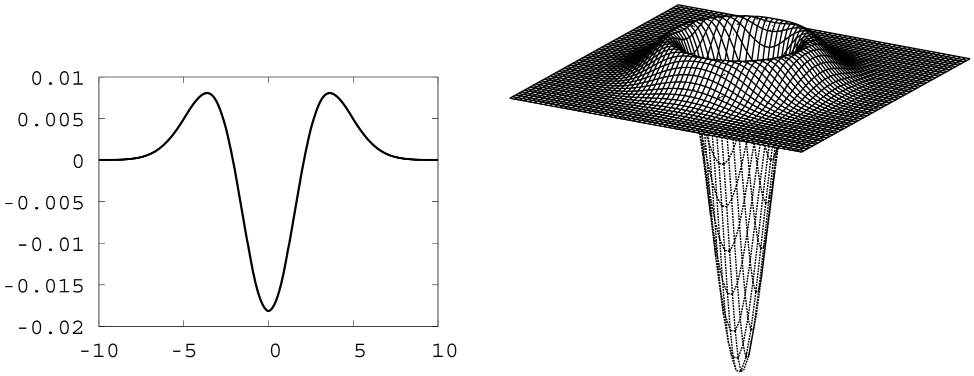


Figure 3.2: *Difference of Gaussians image filter.* Left: a 1D profile of the DoG filter, which is defined as the difference between a Gaussian function and another Gaussian with somewhat smaller width. Left: a 2D version of the filter applicable to images.

feature points in each image, (2) match features between pairs of images, and (3) link up pairwise matches to form point tracks across multiple images.

3.1.1 Feature detection

For the feature detection step, I use the Scale-Invariant Feature Transform (SIFT) feature detector [90]. SIFT is well-suited to matching images in Internet collections due to its repeatability and invariance to certain geometric and photometric image transformations. The SIFT detector works by applying a differences of Gaussian (DoG) filter to the input image, then finding all local maxima and minima of this filtered image; each (x, y) position of an extremum is kept as the location of a feature. The DoG filter, shown in Figure 3.2, attains a maximum or minimum (or is said to “fire”) at the center of “blob”-shaped regions (e.g., the center of a dark circle on a light background, or a light circle on a dark background). The DoG filter also fires on other types of features as well, such as corners where two thick edges meet.

However, the DoG filter only fires at the center of image features whose size is roughly the width of the filter. One of the innovations of SIFT is that it detects features at multiple scales in the image, that is, it detects blobs of varying sizes. The result is that even if an object appears at different scales in two images—e.g., if the Colosseum is 200 pixels wide in one image, and 400 pixels wide in another—SIFT has the ability to match features between the two images (hence, the

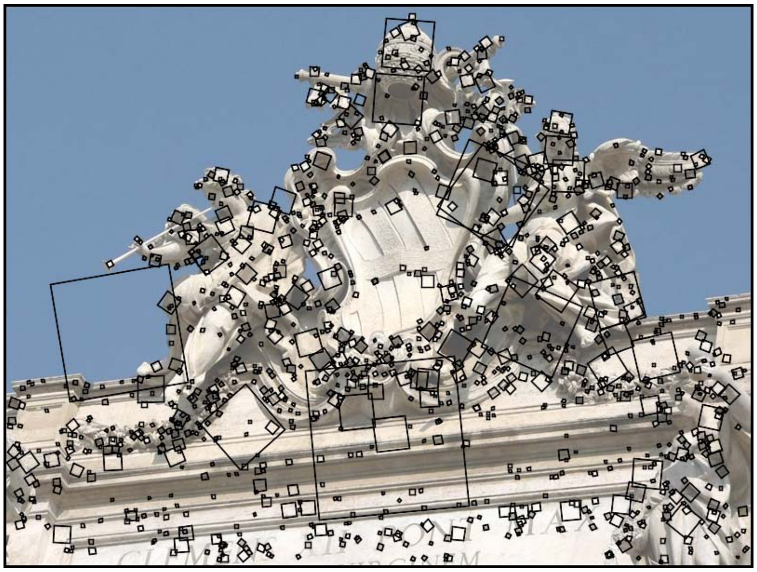


Figure 3.3: *Example set of detected SIFT features.* Each detected SIFT feature is displayed as a black box centered on the detect feature location. SIFT detects a canonical scale and orientation for each feature, depicted by scaling and rotating each box.

"scale-invariant" in the name). SIFT achieves such invariance by, in essence, applying DoG filters of varying widths to the input image, thus creating a stack of filtered images, then finding points in this image stack that are extrema in both image space and scale space.[4] The scale (DoG radius) at which the feature was detected is that feature's canonical scale. Figure 3.3 shows detected SIFT features in a sample image, showing each feature's location and canonical scale. SIFT also detects a canonical orientation for each feature by estimating a dominant local gradient direction.

In addition to detecting a set of feature positions, SIFT also computes a *descriptor* for each feature, or a vector describing the local image appearance around the location of that feature (computed at the feature's canonical scale). One simple example of a descriptor is a window of color (or grayscale) values around the detected point. The descriptor used by SIFT considers image *gradients*, rather than intensity values, as image derivatives are invariant to adding a constant value to the intensity of each pixel. In fact, SIFT looks at the *directions* of these gradients, rather than their

[4] The implementation of SIFT generates the scale space stack somewhat differently, applying a Gaussian filter to an input image multiple times, downsampling when possible, then subtracting consecutive filtered images to create a scale-space pyramid.

raw magnitude, as gradient directions are even more invariant to variations in brightness and contrast across images. In particular, SIFT computes histograms of local image gradient directions. It creates a 4×4 grid of histograms around a feature point, where each histogram contains eight bins for gradient directions, resulting in a $4 \times 4 \times 8 = 128$-dimensional descriptor. Thus, each feature f consists of a 2D location (f_x, f_y), and a descriptor vector $\mathbf{f}_d$. The canonical scale and orientation of a feature are not used in the remainder of the pipeline. The number of SIFT features detected in an image depends on the resolution and content of the image, but a typical 2-megapixel (e.g., 1600×1200) image contains several thousand SIFT features.

Many other feature detectors have been developed, and some, such as MSER [92] or SURF [8] are also designed to be invariant to common image transformations. These could also be used in an SfM system designed for unordered collections.

3.1.2 Feature matching.

Once features have been detected in each image, the system matches features between each pair of images. Let $F(I)$ denote the set of features found in image I. For every pair of images I and J, the system considers each feature $f \in F(I)$ and finds its nearest neighbor (in descriptor space) $f_{nn} \in F(J)$:

$$f_{nn} = \operatorname*{arg\,min}_{f' \in F(J)} ||\mathbf{f}_d - \mathbf{f}'_d||_2.$$

In practice, for efficiency I find an *approximate* nearest neighbor using the approximate nearest neighbors library of Arya and Mount [7], which uses a kd-tree data structure to efficiently compute nearest neighbors. We now have a candidate pair of matching features (f, f_{nn}). Let d_1 be the distance between their descriptors. This candidate match is classified as a true or false match using the ratio test described by Lowe [90]: the matching algorithm finds the second nearest neighbor, $f_{nn2} \in F(J)$, to f (with distance d_2), and accepts the match if $\frac{d_1}{d_2}$, the ratio of the distances to the two nearest neighbors, is less than a threshold (I use 0.6 in practice).

After matching features in I to J, each feature $f \in F(I)$ will be paired with at most one feature in $F(J)$. However, each feature in $F(J)$ may be paired with many features in $F(I)$, as a single feature in $F(J)$ may be closest to multiple features in $F(I)$. The true correspondence must be one-to-one, however, so some of these matches must be spurious. Rather than trying to reason about which are correct, all such multiple matches are removed. If, after this pruning step, an image pair has fewer than a minimum number of matches (I use sixteen), the images are deemed not to match, and all of their feature matches are removed.

We now have a set of putative matching image pairs (I, J), and, for each matching image pair, a set of individual feature matches. Because the matching procedure is imperfect many of these matches—both image matches and individual feature matches—will often be spurious. Fortunately, it is possible to eliminate many spurious matches using a geometric consistency test. This test is based on the fact that, assuming a stationary scene, not all sets of matching features between two images are physically realizable, no matter what the actual shape of the scene is. There is a fundamental constraint between two perspective images of a static scene defined by the possible configurations of the two cameras and their corresponding *epipolar geometry*.[5] The epipolar geometry of a given image pair can be expressed with a 3×3, rank-2 matrix **F**, called the *fundamental matrix* (or *F-matrix*), defined by the relative positions and orientations of the two cameras, as well as internal camera settings such as zoom. Each pair of corresponding points $(x, y) \rightarrow (x', y')$ between two images must satisfy the *epipolar constraint*:

$$\begin{bmatrix} x' & y' & 1 \end{bmatrix} \mathbf{F} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = 0 \tag{3.1}$$

Thus, for a given image pair, only sets of matching features which all satisfy the epipolar constraint for some (initially unknown) fundamental matrix **F** are admissible. We can use this fact to try to separate the true features matches from the spurious ones: the true matches will all be explained by the same F-matrix, while spurious matches are likely to disagree with this F-matrix (and with each other as to what the correct **F** is, assuming all the false matches are independent). Finding such an **F** (and a corresponding set of inliers to this **F**) is essentially a model-fitting problem with noisy data, a problem that can be solved using an algorithm called RANSAC (or RANdom SAmple Consensus) [45]. RANSAC finds an F-matrix consistent with the largest number of matches by generating many different hypothesis F-matrices from random subsets of the data, counting the number

[5] This constraint is very much related to the constraint between image pairs described earlier, that rays through corresponding pixels must intersect.

of matches in agreement with each hypothesis, and keeping the best hypothesis, i.e., the one with the most matches in agreement.[6]

I run RANSAC on each potential matching image pair (I, J), using the eight-point algorithm [68] to generate each RANSAC hypothesis, and normalizing the problem to improve robustness to noise [66]. For each hypothesized F-matrix, I use an outlier threshold of 0.6% of the maximum of the image width and height (roughly six pixels in a 1024×768 image) to decide if each individual feature match is consistent with that F-matrix. I then remove all outliers to the best hypothesis F-matrix from the list of matches. If the number of remaining feature matches is less than sixteen, I remove all of the feature matches between I and J from consideration, and deem (I, J) to be a non-matching image pair.

Once all $\binom{n}{2}$ image pairs have been matched, I organize the matches into point tracks by finding connected sets of matching features across multiple images. For instance, if feature $f_1 \in F(I_1)$ matches feature $f_2 \in F(I_2)$, and f_2 matches feature $f_3 \in F(I_3)$, these features will be grouped into a track $\{f_1, f_2, f_3\}$. Tracks are found by examining each feature f in each image and performing a breadth-first search of the set of features in other images that match f until an entire connected component of features has been explored. These features are then grouped together into a track, and the next unvisited feature is considered, until all features have been visited. Because of spurious matches, inconsistencies can arise in tracks; in particular, a track can contain multiple features from the same image, which violates the assumption that a track corresponds to a single 3D point. For instance, if feature $f_3 \in F(I_3)$ in the example above matches a different feature $f'_1 \neq f_1 \in F(I_1)$, then image I_1 will observe the track in two different locations (corresponding to features f_1 and f'_1). Such tracks are identified as inconsistent, and any image that observes a track multiple times has all of their features removed from that track.

Once correspondences have been found, I construct an *image connectivity graph*, which contains a node for each image, and an edge between any pair of images with shared tracks. A visualization of the connectivity graph for the **Trevi** data set (a collection of Internet photos of the Trevi Fountain) is shown in Figure 3.4. To create this visualization, the graph was embedded in the plane using the *neato* tool in the Graphviz graph visualization toolkit [60]. *Neato* works by modeling the graph as a

[6] RANSAC is not specific to F-matrices, but can be used to fit many other types of models to noisy data as well. Later in this chapter RANSAC is used to fit homographies and projection matrices.

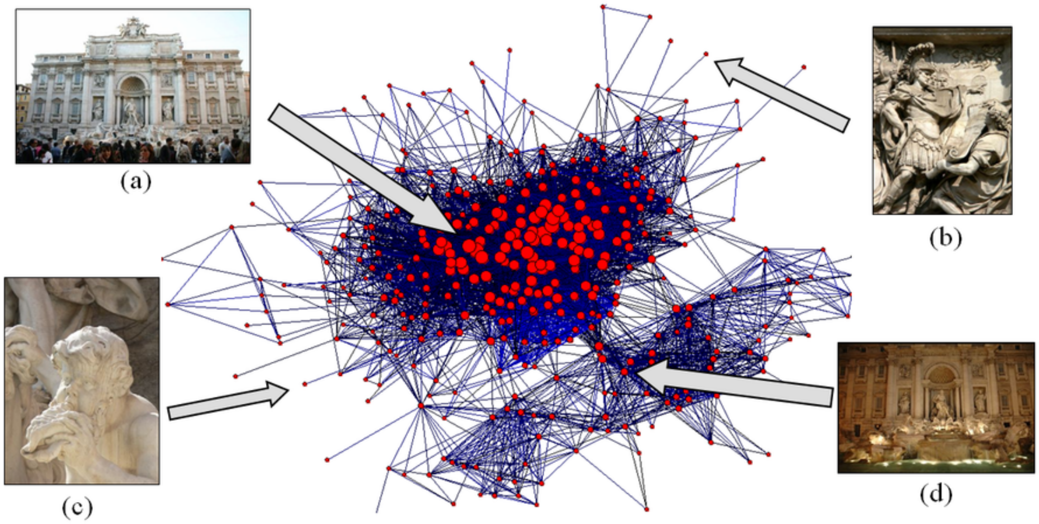


Figure 3.4: *Image connectivity graph for the Trevi Fountain.* This graph contains a node (red dot) for each image in a set of photos of the Trevi Fountain, and an edge between each pair of photos with shared tracks. The size of a node is proportional to its degree. There are two dominant clusters corresponding to daytime photos (e.g. image (a)) and nighttime photos (image (d)). Similar views of the facade are clustered together in the center of the graph, while nodes in the periphery, e.g., (b) and (c), are more unusual (often close-up) views. An interactive version of this graph can be found at `http://phototour.cs.washington.edu/imagegraphs/Trevi/`.

mass-spring system and solving for an embedding whose energy is a local minimum.

The image connectivity graph for this particular collection has several distinct features. There is a large, dense cluster in the center of the graph which consists of photos that are all fairly wide-angle, frontal, well-lit shots of the fountain (such as image (a)). Other images, including the "leaf" nodes (such as (b) and (c)) corresponding to tightly-cropped details, and nighttime images (such as (d)), are more loosely connected to this core set. Other connectivity graphs are shown in Figures 3.10, 3.11, and 3.12, and their structure is discussed in more detail in Section 3.6.1.

3.2 Structure from motion

The second stage of the pipeline takes the set of point tracks found in the first stage and estimates the 3D camera and scene geometry, by finding the configuration of cameras and 3D points which, when related through the equations of perspective projection, best agrees with the detected tracks. This the *structure from motion* (SfM) problem.

First, let us establish some notation. The scene geometry consists of a 3D point $\mathbf{X}_j$ for each point track j. A camera can be described by its pose (position and orientation, also known as the *extrinsic parameters*), and parameters modeling the internal image formation process (the *intrinsic camera parameters*). I represent the extrinsic parameters of camera i with a 3-vector, $\mathbf{c}_i$, describing the camera center, and a 3×3 rotation matrix $\mathbf{R}_i$ describing the camera orientation (I will also sometimes refer to the camera *translation*, $\mathbf{t}_i$, where $\mathbf{t}_i = -\mathbf{R}_i\mathbf{c}_i$). Camera intrinsics are often represented with an upper-diagonal 3×3 matrix $\mathbf{K}_i$ mapping 3D rays to homogeneous 2D image positions. In this thesis, however, I assume that this linear mapping can be modeled with a single parameter f_i, the focal length, and that $\mathbf{K}_i = \mathrm{diag}(f_i, f_i, 1)$. In general, the off-diagonal terms of the $\mathbf{K}$ matrix model *skew* (the amount by which the angle between the image x-axis and y-axis deviates from $90°$) and the *principal point* (the point where the camera's optical axis intersects the image plane), and the first two diagonal entries need not be equal (the *aspect ratio* between the width and height of each pixel need not be one). However, for most digital cameras, the skew is close to zero, the principal point is near the image center, and the aspect ratio is close to one, thus the assumption that $\mathbf{K}_i = \mathrm{diag}(f_i, f_i, 1)$ is reasonable [122]. While this is not an essential assumption, it reduces the number of parameters that need to be estimated, and increases the stability of the system, as certain parameters, especially the principal point, tend to be severely ill-conditioned. However, certain images, such as images created by cropping a non-centered region of a large image, do not fit this model.

On the other hand, while the $\mathbf{K}$ matrix models a linear mapping, most consumer cameras have noticeable non-linear lens distortion. I model this distortion with a polynomial in ρ, the distance from the center of the image, using a quartic polynomial

$$\kappa_1\rho^2 + \kappa_2\rho^4 \tag{3.2}$$

with two distortion parameters, κ_1 and κ_2, for each image. I refer to the collection of camera parameters for a single camera as $C_i = \{\mathbf{c}_i, \mathbf{R}_i, f_i, \kappa_{1i}, \kappa_{2i}\}$. This set of parameters can be described by nine numbers: three for the camera center $\mathbf{c}_i$, three for the rotation, as explained in Section 3.2.3, and the three intrinsic parameters f_i, κ_{1i} and κ_{2i}. Together, the camera parameters C_i specify how a 3D point $\mathbf{X}_j = (\mathbf{X}_{jx}, \mathbf{X}_{jy}, \mathbf{X}_{jz})$ is projected to a 2D point $\mathbf{x}$ in image i (via the projection equation

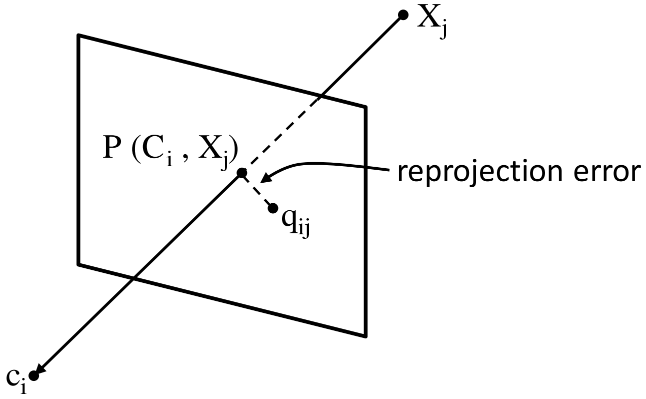


Figure 3.5: *Reprojection error.* A 3D point $\mathbf{X}_j$ is projected into camera C_i. The reprojection error is the distance between the projected image point, $P(C_i, \mathbf{X}_j)$ and the observed image point $\mathbf{q}_{ij}$.

$P(C_i, \mathbf{X}_j)$ defined below).

The point tracks found in the correspondence estimation stage can be thought of as noisy measurements of the scene. In particular, each track (ideally) contains measurements of the projected positions of a single 3D point in certain viewpoints. I denote the measured position of track j in image i as $\mathbf{q}_{ij}$; note that many $\mathbf{q}_{ij}$ are typically unknown, as not all images see all tracks. Structure from motion is the problem of taking these measurements and jointly solving for the camera and scene parameters that predict these measurements as well as possible. This problem is usually posed as an optimization over the collective set of camera and scene parameters $C = \{C_1, C_2, \ldots, C_n\}$ and $X = \{\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_m\}$, where an objective function measures the discrepancy between the measured 2D point positions and those predicted by the projection equation P. For n views and m tracks, the objective function g can be written as:

$$g(C, X) = \sum_{i=1}^{n} \sum_{j=1}^{m} w_{ij} ||\mathbf{q}_{ij} - P(C_i, \mathbf{X}_j)||^2 \tag{3.3}$$

where w_{ij} is an indicator variable: $w_{ij} = 1$ if camera i observes track j, and $w_{ij} = 0$ otherwise. The expression $||\mathbf{q}_{ij} - P(C_i, \mathbf{X}_j)||$ in the interior of this summation is called the *reprojection error* of track j in camera i. Reprojection error is illustrated in Figure 3.5. Thus, the objective function g is the sum of squared reprojection errors, weighted by the indicator variable. The goal of SfM is to find the camera and scene parameters that minimize this objective function. In my pipeline,

g is minimized using non-linear least squares optimization, a process known as *bundle adjustment* [150].

The projection equation. The projection equation $P(C_i, \mathbf{X}_j)$ is defined as follows. First, the point $\mathbf{X}_j$ is converted to the camera's coordinate system through a rigid transformation:

$$\mathbf{X}' = \begin{bmatrix} \mathbf{X}'_x \\ \mathbf{X}'_y \\ \mathbf{X}'_z \end{bmatrix} = \mathbf{R}_i(\mathbf{X}_j - \mathbf{c}_i).$$

Next, the perspective division is performed, and the result is scaled by the focal length:

$$\mathbf{x}' = \begin{bmatrix} f_i \mathbf{X}'_x / \mathbf{X}'_z \\ f_i \mathbf{X}'_y / \mathbf{X}'_z \end{bmatrix}.$$

$\mathbf{x}' = \left[\mathbf{x}'_x, \mathbf{x}'_y\right]^T$ is now a 2D point in the image. Finally, radial distortion is applied to this point:

$$\begin{aligned} \rho^2 &= \left(\frac{\mathbf{x}'_x}{f_i}\right)^2 + \left(\frac{\mathbf{x}'_y}{f_i}\right)^2 \\ \alpha &= \kappa_{1i}\rho^2 + \kappa_{2i}\rho^4 \\ \mathbf{x} &= \alpha \mathbf{x}', \end{aligned}$$

$\mathbf{x}$ is then the computed projection.

Another common way to write the projection equation is to group the camera parameters C_i (excluding the distortion parameters) into a 3×4 projection matrix $\mathbf{\Pi}_i$:

$$\mathbf{\Pi}_i = \mathbf{K}_i[\mathbf{R}_i|\mathbf{t}_i]$$

which maps homogeneous 3D points to homogeneous 2D image positions. The distortion and perspective division can then by applied after the projection matrix:

$$P(C_i, \mathbf{X}_j) = r(z_{\text{div}}(\mathbf{\Pi}\mathbf{X}_j))$$

where z_{div} performs the perspective division and r applies the radial distortion.

Because of the rotation, perspective division, and radial distortion, P is a non-linear function, and therefore bundle adjustment is a non-linear least squares problem. Algorithms for non-linear least squares, such as Levenberg-Marquardt, are only guaranteed to find local minima, and large-scale SfM problems are particularly prone to getting stuck in bad local minima. Therefore it is important to provide good initial estimates of the parameters. Rather than trying to initialize the parameters for all cameras and points at once, I take an incremental approach. I start with two cameras, find their optimal parameters, and those of the points they observe, then iteratively add one camera at a time to the optimization.

Reconstructing the initial pair. I begin by estimating the parameters for a single pair of cameras. This first step is critical: if the reconstruction of the initial pair gets stuck in the wrong local minimum, the optimization is unlikely to ever recover. The initial pair must therefore be chosen carefully. The images should have a large number of matches, but also have a large baseline (distance between camera centers), so that the initial two-frame reconstruction can be robustly estimated. I therefore choose the pair of images that has the largest number of matches, subject to the condition that their matches cannot be well-modeled by a homography. A homography models the transformation between two images of a single plane, or two images taken at the same location (but possibly with cameras looking in different directions). Thus, if a homography cannot be fit to the correspondences between two images, it indicates that the cameras have some distance between them, and that there is interesting 3D structure visible. These criteria are important for robust estimation of camera pose.

In particular, I estimate a homography between each pair of matching images using RANSAC with an outlier threshold of 0.4% of $\max(\text{image width}, \text{image height})$, and store the percentage of feature matches that are inliers to the estimated homography. I select the initial two images to be the pair with the lowest percentage of inliers, but which have at least 100 matches.

The system estimates the extrinsic parameters for this initial pair using Nistér's implementation of the five point algorithm [102],[7] and the tracks visible in the two images are triangulated, giving an initial set of 3D points. I then perform a two-frame bundle adjustment starting from this initializa-

[7] I only choose the initial pair among pairs for which a focal length estimate is available for both cameras, and therefore a calibrated relative pose algorithm can be used.

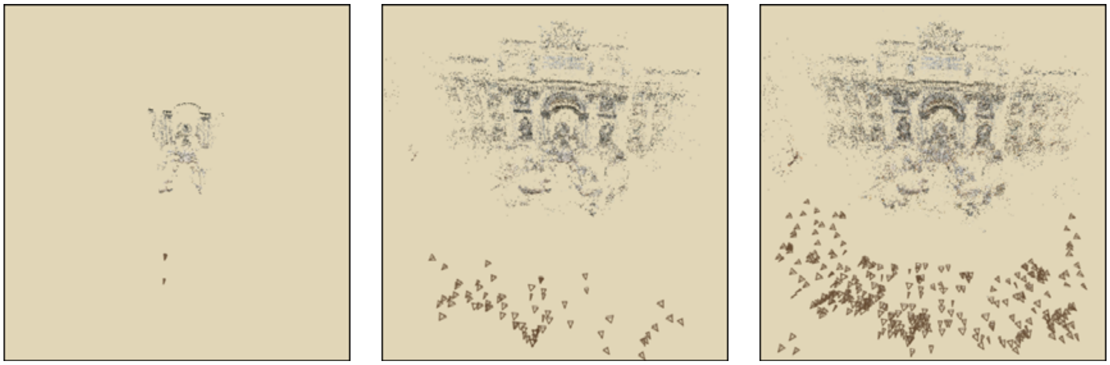


Figure 3.6: *Incremental structure from motion.* My incremental structure from motion approach reconstructs the scene a few cameras at a time. This sequence of images shows the **Trevi** data set at three different stages during incremental reconstruction. Left: the initial two-frame reconstruction. Middle: an intermediate stage, after fifteen images have been added. Right: the final reconstruction with 360 photos.

tion. To perform bundle adjustment, I use the sparse bundle adjustment (SBA) package of Lourakis and Argyros [89]. SBA is described in Section 3.2.4.

Adding new cameras and points. Next, I add another camera to the optimization. I select the camera that observes the largest number of tracks whose 3D locations have already been estimated. To initialize the pose of the new camera, I first estimate its projection matrix $\mathbf{\Pi}$ using the direct linear transform (DLT) technique [68] inside a RANSAC procedure. For this RANSAC step, I use an outlier threshold of 0.4% of $\max(\text{image width}, \text{image height})$.

Recall that the projection matrix has the form

$$\mathbf{\Pi} = \mathbf{K}\left[\mathbf{R}|\mathbf{t}\right] = \left[\mathbf{K}\mathbf{R}|\mathbf{K}\mathbf{t}\right],$$

therefore the left 3×3 submatrix of $\mathbf{\Pi}$ (denoted $\mathbf{\Pi}_3$) is the product of an upper-triangular matrix $\mathbf{K}$ and a rotation matrix $\mathbf{R}$. $\mathbf{K}$ and $\mathbf{R}$ can therefore be computed as the RQ decomposition of $\mathbf{\Pi}_3$.[8]

[8] Note that, in order for it to be a valid intrinsic matrix, the upper triangular matrix $\mathbf{K}$ should have positive entries along the diagonal. A unique RQ decomposition with this property is guaranteed as long as $\mathbf{\Pi}_3$ is non-singular. $\mathbf{\Pi}_3$ will be non-singular if the 3D points used to solve for $\mathbf{\Pi}$ are non-coplanar, and the corresponding 2D points are non-colinear, which almost always is the case. Note, however, that the 3D points are often nearly co-planar, which can cause instability in the estimation of $\mathbf{K}$. In my experience, most scenes have sufficient non-planar 3D structure to avoid this problem.

The translation $\mathbf{t}$ is then the last column of $\mathbf{K}^{-1}\mathbf{\Pi}$.

I use the rotation $\mathbf{R}$ and translation $\mathbf{t}$ estimated using the above procedure as the initial pose for the new camera. In addition, I use the estimated $\mathbf{K}$ to either initialize the focal length of the new camera, or verify an existing focal length estimate (see "Recovering focal lengths" later in this section). Starting from this set of initial parameters, I do a round of bundle adjustment, allowing only the new camera to change; the rest of the model is held fixed.

Next, I add points observed by the new camera into the optimization. A point is added if it is observed by at least two cameras, and if triangulating the point gives a well-conditioned estimate of its location. I estimate the conditioning by considering all pairs of rays that could be used to triangulate the new point, and finding the pair with the maximum angle of separation $\theta_{\max}$. If $\theta_{\max}$ is larger than a threshold of 2°, then the point is triangulated and added to the optimization.[9] Once the new points have been added, bundle adjustment is performed on the entire model.

This procedure of initializing a camera, triangulating points, and running bundle adjustment is repeated, one camera at a time, until no remaining camera observes a sufficient number of points (at least twenty in my implementation). Figure 3.6 illustrates this process for the **Trevi** data set. In general only a subset of the images will be reconstructed. This subset is not selected beforehand, but is determined by the algorithm as it adds images until no more can reliably be added.

3.2.1 Improvements to the algorithm

For increased robustness and speed, I make a few modifications to the basic SfM procedure outlined above. The first modification deals with robustness to mismatches in the set of point tracks. Spurious matches can have a dramatic effect on the reconstruction, pulling the recovered structure far away from the correct solution. It is therefore critical to handle such outliers in a robust way. After each run of bundle adjustment I detect outliers by identifying tracks that contain at least one feature with a high reprojection error. These tracks are then removed from the optimization. The outlier threshold for a given image adapts to the current distribution of reprojection errors for that image. For an image I, I compute d_{80}, the 80th percentile of the reprojection errors for that image, and use

[9] This test tends to reject points at infinity; while points at infinity can be very useful for estimating accurate camera rotations, I have observed that they can sometimes result in problems, as using noisy camera parameters to triangulate points at infinity can result in points at erroneous, finite 3D locations. See Section 3.7, "Cascading errors."

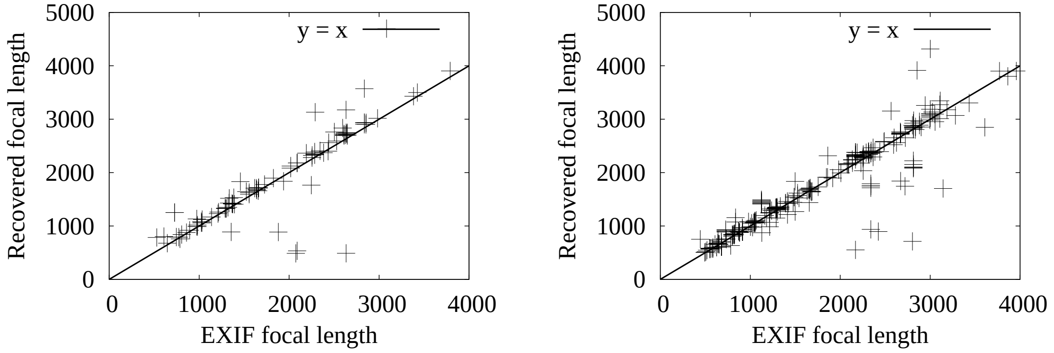
$\mathrm{clamp}(2.4d_{80}, 4, 16)$ as the outlier threshold (where $\mathrm{clamp}(x, a, b)$ clamps x to the range $[a, b]$). The effect of this clamping function is that all points with a reprojection error of more than 16 pixels in any image will be rejected as outliers, and all points with a reprojection error of less than 4 pixels will be kept as inliers; the exact threshold is adaptive, and lies between these two values. After rejecting outliers, I rerun bundle adjustment and repeat until no more outliers are detected.

The second modification is that, rather than adding a single camera at a time into the optimization, I add multiple cameras. To select which cameras to add, I first find the camera with the greatest number of matches, M, to existing 3D points, then add any camera with at least $0.75M$ matches to existing 3D points. Adding multiple cameras at once results in many fewer overall iterations of bundle adjustment and thus improved efficiency.

3.2.2 Recovering focal lengths

Most digital cameras embed metadata into JPEG images in the form of Exchangeable image file format (Exif) tags. These tags often include information on the camera make and model, the date and time a photo was taken, and camera settings such as aperture, shutter speed, and focal length. When an image has a focal length estimate available, it can be very helpful as initialization for the focal length parameter of that camera in the optimization, as it can be difficult to reliably estimate focal lengths without any calibration, particularly for planar or near-planar scenes. Exif tags invariably provide focal length estimates in millimeters. To be useful, these estimates must be converted to pixel units, as described in Appendix A.

I have found that most focal length estimates obtained from Exif tags are accurate enough to use for initialization, as demonstrated in Figure 3.7; the median absolute difference between the focal lengths calculated from Exif tags and those derived from SfM for the **Trevi** data set is just 3.5%. However, the occasional image will be tagged with a wildly inaccurate focal length. Therefore, when estimating the pose of a new camera, I check its focal length estimate against the independent estimate obtained through the DLT procedure. Recall that the DLT procedure results in an estimate



Trevi data set

Hagia Sofia data set

Figure 3.7: *Accuracy of Exif focal lengths.* These scatter plots compare focal length estimates obtained from the Exif tags of the image files (x-axis) with those recovered using the structure from motion pipeline (y-axis) for two data sets (**Trevi** and **Hagia Sophia**); the line $y = x$ is plotted for comparison. The Exif tags were completely ignored for this experiment; initial values for focal lengths were set using f_{dlt}, and no soft constraints were used during bundle adjustment. For Trevi, the recovered focal lengths tend to be slightly larger than those obtained from the Exif tags (by about 2.4% on average), while the average absolute difference is about 12%. However, this difference is skewed by the presence of outliers; the median absolute difference is just 3.5%. For Hagia Sophia, the average difference is about 0.9%, with an average absolute difference of 7.8% and a median absolute difference of 3.1%. While in general Exif tags are accurate enough to obtain a useful prior, the two focal length estimates occasionally vary significantly, hence the outliers in the plot. The reason for the estimated focal length being slightly longer may be due to a systematic source of error. For instance, perhaps cameras use a slightly smaller region of the image sensor than the dimensions listed in camera specifications, or perhaps cameras usually round the focal length down to the nearest discrete setting when preparing Exif tags.

of the upper-triangular intrinsic matrix $\mathbf{K}$:

$$\mathbf{K} = \begin{bmatrix} k_{11} & k_{12} & k_{13} \\ 0 & k_{22} & k_{23} \\ 0 & 0 & 1 \end{bmatrix}$$

Let $f_{\mathrm{dlt}} = \frac{1}{2}(k_{11} + k_{22})$ be the DLT estimate of the focal length, and let f_{Exif} be the focal length calculated from the Exif tags of an image. I prefer to use f_{Exif} or initialization when it exists, but I first test whether f_{Exif} disagrees with the DLT estimate by a large factor; a large discrepancy can indicate that f_{Exif} is erroneous. In particular, I check that $0.7 f_{\mathrm{dlt}} < f_{\mathrm{Exif}} < 1.4 f_{\mathrm{dlt}}$. If this test fails, I use f_{dlt} as the initial value for the focal length. Otherwise, I use f_{Exif}. In addition, when f_{Exif} exists and is in agreement with f_{dlt}, I add a soft constraint to the optimization to encourage the focal length parameter f for the camera to stay close to this initial value, by adding the term $\gamma(f - f_{\text{Exif}})^2$ to the objective function g (using a value $\gamma = 1 \times 10^{-4}$ in all experiments). No such constraint is used if f_{dlt} is used for initialization.

3.2.3 Parameterization of rotations

During bundle adjustment, I parameterize the camera rotation with a 3-vector ω representing an incremental rotation. ω is equal to a rotation axis (represented as a unit 3-vector $\hat{n}$) times the angle of rotation θ:

$$\omega = \theta\hat{n}$$

and the incremental rotation matrix $\mathbf{R}(\theta, \hat{n})$ is defined as:

$$\mathbf{R}(\theta, \hat{n}) = \mathbf{I} + \sin\theta\,[\hat{n}]_\times + (1 - \cos\theta)\,[\hat{n}]_\times^2\,,$$

where

$$[\hat{n}]_\times = \begin{bmatrix} 0 & -\hat{n}_z & \hat{n}_y \\ \hat{n}_z & 0 & -\hat{n}_x \\ -\hat{n}_y & \hat{n}_x & 0 \end{bmatrix}.$$

The incremental rotation matrix $\mathbf{R}(\theta, \hat{n})$ is pre-multiplied by the initial rotation matrix to compute the current rotation inside the global optimization. For small incremental rotations, $\mathbf{R}(\theta, \hat{n})$ is nearly linear in ω.

3.2.4 Sparse bundle adjustment

To optimize the objective function g, my system uses the sparse bundle adjustment (SBA) package [89]. SBA is a non-linear optimization package that takes advantage of the special sparse structure of SfM problems to provide an algorithm with reduced time and memory complexity. At the heart of SBA is an implementation of the Levenberg-Marquart algorithm [105], which, like many non-linear solvers, takes as input an initial guess at the solution and finds a nearby local minimum by solving a series of linear systems $\mathbf{H}\mathbf{x} = \mathbf{b}$ (called *normal equations*) that locally approximate the objective function. The matrix $\mathbf{H}$ is an approximation to the Hessian of the objective function at a given point, where the Hessian is a square, symmetric matrix describing the shape of a quadratic function.

$\mathbf{H}$ has a row and column for each variable in the optimization. For n cameras and m point tracks, $\mathbf{H}$ is a $(9n + 3m) \times (9n + 3m)$ matrix; thus, $\mathbf{H}$ can be quite large. For 1,000 cameras and 100,000 points (a typical problem size), $\mathbf{H}$ has 300,900 rows and columns, and nearly a billion entries, requiring a hefty 3.6GB of memory to store as an upper-diagonal matrix of double precision floating point numbers. The time complexity of directly solving a dense linear system is cubic in the number of variables, also quite high.

Fortunately, however, most of the entries of $\mathbf{H}$ are zeroes, and the matrix has a particular sparsity pattern. Each entry h_{ij} is non-zero if (and only if) the two variables with index i and j directly interact (i.e., they appear together in any single reprojection error computation in the interior of g). Hence, all nine parameters of any single camera interact, as do the three parameters of any single point. If a given camera sees a given point, the parameters of the camera and the point interact. However, any two distinct cameras do not directly interact, nor do any pair of distinct points; in addition, if a camera does not observe a point, the camera and the points do not interact.

If we order the variables so that the camera parameters (grouped by camera) come first, followed by the point parameters (grouped by point), the sparsity pattern of $\mathbf{H}$ takes a form similar to that shown in Figure 3.8. This matrix can be broken into four parts:

$$\mathbf{H} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{B}^T & \mathbf{C} \end{bmatrix} \tag{3.4}$$

where $\mathbf{A}$ is the camera sub-matrix, $\mathbf{C}$ is the point sub-matrix, and $\mathbf{B}$ represents the interactions between the cameras and points.

We can take advantage of this sparsity pattern to save memory by representing the matrix in a sparse storage format. SBA does so, but also uses a technique called the *Schur complement* [150] to improve efficiency. This technique factors out the point parameters to produce a *reduced camera system* whose size is the the number of camera parameters; the particular sparsity pattern of SfM

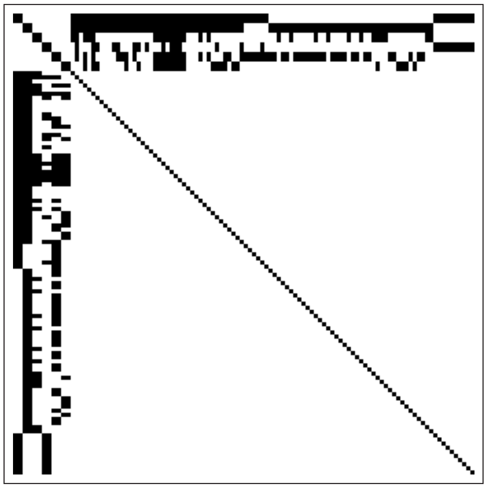


Figure 3.8: *Sparsity pattern of the Hessian matrix for a structure from motion problem.* In this image, each row and column corresponds to a variable in a structure from motion problem containing six cameras and about one hundred points. The camera parameters appear first, followed by the point parameters. Non-zero entries in this matrix are colored black, while zeros are colored white.

makes this reduction particularly efficient. SBA applies a direct linear solver to the reduced camera system to obtain the new camera parameters, then back-substitutes for the new point parameters. For large numbers of cameras and points, the bottleneck in this procedure becomes the solution of the reduced camera system, which has cubic complexity in the number of cameras.

3.3 Algorithmic complexity

There are several parts of the SfM pipeline that require significant computational resources: SIFT feature detection, pairwise feature matching and F-matrix estimation, linking matches into tracks, and incremental SfM . This section analyzes the time and space complexity of each of these steps. I will mainly analyze time and space complexity as a function of the number of input images n. In practice, this analysis also depends on the number of features detected in each image, which in turns depends on factors such as image resolution and how textured each image is. However, image resolution, and hence number of features, is bounded (each image containing a few thousand features on average), so I treat the number of features as roughly constant per image, thereby factoring them out of the analysis.

SIFT. The feature detection step is linear in the number of input images ($O(n)$). Running SIFT on an individual image, however, can take a significant amount of time and use a large amount of memory, especially for high-resolution images with complex textures. For instance, in the **Hagia Sofia** data set (described in Section 3.6) the maximum number of SIFT features detected in an image was over 120,000. SIFT ran for 1.25 minutes on that image, and used 2.7 GB of memory, on a test machine with a 3.80 GHz Intel Xeon processor. (The minimum number of features detected was 31; SIFT ran for 4 seconds on that image.) SIFT spent an average time of 12 seconds processing each image in this collection (and about 5.2 hours of CPU time in total). Because SIFT runs on each image independently, it is very easy to parallelize by assigning different sets of images to different processors. Thus, despite being relatively time-consuming for some images, the low overall complexity of the feature matching step, combined with its easy parallelization, means that it takes a small percentage of the overall computation time.

Feature matching and F-matrix estimation. The feature matching step, on the other hand, does take a significant percentage of the total processing time. This is mostly due to its relatively high complexity. Since each pair of images is considered, it has quadratic time complexity in the number of input images ($O(n^2)$). For the Hagia Sofia data set, each individual image match took an average of 0.8 seconds; in total, it took 11.7 days of CPU time to match the entire collection. Fortunately, matching is also very easy to parallelize, as each pair of images can be processed independently. Parallelization can compensate for the high complexity to a point, and for several example scenes in Section 3.6, the matching stage took much more CPU time, but somewhat less wall clock time, than the SfM stage.

Because F-matrix estimation is only run on the pairs of images that successfully match, it tends to take a much smaller amount of time than the matching itself. Though this step is $O(n^2)$ in the worst case (if all images match), for Internet collections, the percentage of image pairs that match is usually fairly small. For instance, for the Hagia Sofia collection, only 33,637 out of a possible 1,226,961 image pairs (2.7%) made it to the F-matrix estimation stage. Estimating an F-matrix for each pair took about one hour, and 11,784 of the 33,637 pairs survived this step. This step is also easily parallelized.

Linking up matches to form tracks. To implement this step, I perform a breadth-first search on the graph of feature matches, marking each feature as it is visited. Each feature in each image is visited once (i.e., $O(n)$ features are visited), and each individual feature match is also visited once (in the worst case, $O(n^2)$ matches are visited). However, the graph of feature matches also tends to be quite sparse. For the Hagia Sofia collection, there were 1,149,369 individual feature matches left after the F-matrix estimation step, about 100 per matching image. Grouping matches into tracks took a total of just six minutes for this collection.

Structure from motion. The main computational bottleneck in the incremental structure from motion algorithm is the bundle adjustment stage computed using SBA.[10] As described in Section 3.2.4, SBA uses the Levenberg-Marquardt (LM) algorithm to find a local minimum of the objective function. Each iteration of LM involves solving a linear system; in SBA's implementation this is done using a dense, direct method such as LU factorization. Solving the reduced camera system using such methods takes time cubic ($\Theta(t^3)$) in the number of cameras t in the current reconstruction [56]. Each call to SBA is capped at a maximum number of iterations, thus the total run time of each call to SBA is still $\Theta(t^3)$. During the course of the SfM stage, SBA is called repeatedly with more and more images as the reconstruction grows. The total running time of SfM thus depends on how many times SBA is called, and on how many images are added to the reconstruction during each iteration of SfM.

If all the images are added immediately after the initial two-frame reconstruction, there will be a single call to SBA with all n images, and the total running time for SfM will be $\Theta(n^3)$. This $\Theta(n^3)$ complexity will also hold if SBA is called a constant k number of times, e.g., if on average some proportion $\frac{n}{k}$ of the input images are added during each iteration of SfM (see Appendix B for the details of this analysis). If, on the other hand, a roughly constant number of images ℓ are added per iteration, independent of n, then the total running time will be $\Theta(n^4)$. Roughly speaking, each call to SBA with t images does work in proportion to t^3, so when ℓ images are added per iteration, the

[10] Time is also spent in SBA triangulating new points and estimating the pose of new cameras, but the running times of these steps are dominated by bundle adjustment.

total amount of work done by SBA over the course of SfM grows roughly as:

$$\sum_{i=1}^{n/\ell}(i \cdot \ell)^3 = \ell^3 \sum_{i=1}^{n/\ell} i^3 = \ell^3 \left[\frac{\left(\frac{n}{\ell}\right)\left(\frac{n}{\ell}+1\right)}{2}\right]^2 = O(n^4)$$

Appendix B presents a more detailed analysis showing that the running time of this scenario is $\Theta(n^4)$.

The total running time depends on the properties of the collection. If all the photos are very similar, they may be added in large batches, and the running time may be close to $\Theta(n^3)$. If, on the other hand, the images are taken with a roughly uniform sampling along some path, it is likely that a constant number of images will be added at each iteration (on average), resulting in a total running time of $\Theta(n^4)$. In my experience, Internet photo collections behave more like the latter case, resulting in run times closer to $\Theta(n^4)$.[11] In the future, analyzing the behavior of the algorithm on a larger collection of image sets will allow for better characterization of the (empirical) run time.

In addition to the high complexity of the SfM stage, out of all the stages described in this section, SfM is the most difficult to parallelize. Doing so would require either a parallel implementation of a linear system solver, or a way to subdivide the images into independent groups. Despite these difficulties, in practice the time required for SfM is often less than the time spent matching images. For instance, SfM took only 5.5 hours for the Hagia Sofia data set, compared to more than one day (of wall clock time) for matching. There may be several reasons for this. First, typically only a subset of the images in a collection are ultimately registered. For instance, in the Hagia Sofia collection, 446 of the 1567 input images (28%) were registered. Second, I use the highly optimized Intel Math Kernel Library (MKL) linear solver [70]. For an example 600 image data set, MKL solves the reduced camera system in about 27 seconds. This is a significant amount of time, but for collections of 1,000 photos or less this typically leads to running times on the order of days (rather than weeks). Nevertheless, for truly massive, well-connected collections, SfM would likely dwarf the amount of time spent in the rest of the pipeline.

SBA uses $O(nm)$ amount of memory (for n cameras and m points) to store the matrix of binary indicator variables w_{ij} defining which points are visible in which views, and $O(n^2)$ memory to store

[11] This despite the fact that only a subset of images are ultimately registered, and SfM is only run on that subset. The size of this subset tends to grow in proportion to the total number of images, however.

the reduced camera matrix. In practice, the matrix of indicator variables is quite sparse, so a more memory-efficient sparse matrix representation could be used. The reduced camera matrix is also often sparse (an entry corresponding to two cameras is only non-zero if the cameras have shared point tracks), and thus could also be potentially more storage-efficient in practice. In addition, reordering techniques [150] could be used to reduce fill-in while factoring the reduced camera matrix.

3.4 Geo-registration

The SfM procedure above estimates *relative* camera locations, reconstructing geometry up to an unknown similarity transformation. These relative camera locations are sufficient for many tasks, including most of the scene visualization applications described in Part II, which do not absolutely require knowing where in the world a scene exists, or how large the scene is. However, other applications require more information. For instance, if we want to measure distances in the reconstructed scene in meaningful units, we need to know the scale of the reconstruction. If we want to display the reconstruction on a site such as Google Maps, we need to know the latitude and longitude of each camera. For these tasks, I describe techniques for aligning the recovered model with a geo-referenced image or map (such as a satellite image, floor plan, or digital elevation map) so as to determine the absolute geocentric coordinates of each camera.

Given a perfectly accurate reconstruction, the estimated camera locations are related to the absolute locations by a similarity transform (global translation, 3D rotation, and uniform scale). My system provides a simple manual interface for determining the alignment of a model to a geolocated map or overhead image; the user interactively rotates, translates, and scales the model until it is in agreement with a provided map. To assist the user, I first estimate the "up" or gravity vector, as described in Section 3.5. The 3D points and camera locations are then rendered superimposed on the alignment image, using an orthographic projection with the camera positioned above the scene, pointed downward. If the up vector was estimated correctly, the user needs only to rotate the model in 2D, rather than 3D. In my experience, it is usually fairly easy, especially in urban scenes, to perform this alignment by matching the recovered points to features, such as building façades, visible in the image. Figure 3.9 shows a screenshot of such an alignment.

In some cases the recovered scene cannot be well-aligned to a geo-referenced coordinate system

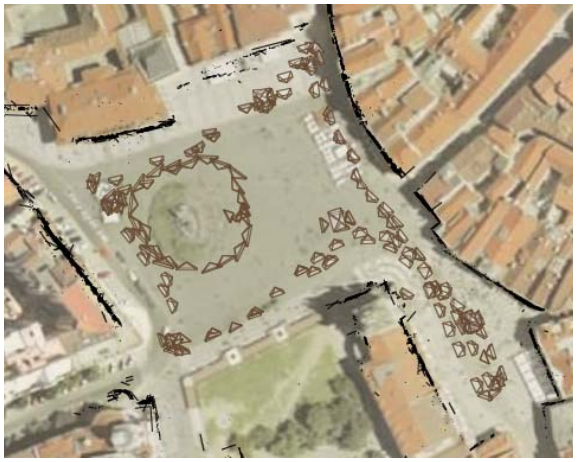


Figure 3.9: *Example registration of cameras to an overhead map.* Here, the cameras and recovered line segments from the Prague data set are shown superimposed on an aerial image. (Aerial image shown here and in Figure 5.1 courtesy of Gefos, a.s. [51] and Atlas.cz.)

using a similarity transform. This can happen because errors accumulated during reconstruction can cause low-frequency drift in the recovered point and camera locations. Drift does not have a significant effect on the visualization tools described in Part II, as the error is not usually locally noticeable, but is problematic when an accurate model is desired.

One way to "straighten out" the recovered scene is to pin down a sparse set of ground control points or cameras to known 3D locations (acquired, for instance, from GPS tags attached to a few images) by adding constraints to the SfM optimization. Alternatively, a user can manually specify correspondences between points or cameras and locations in an image or map, as in the work of Robertson and Cipolla [120].

3.4.1 Aligning to Digital Elevation Maps

For landscapes and other very large scale scenes, we can take advantage of Digital Elevation Maps (DEMs), used for example in Google Earth [57] and with coverage of most of the United States available through the U.S. Geological Survey [151]. To align point cloud reconstructions to DEMs, I manually specify a few correspondences between the point cloud and the DEM, and estimate a 3D similarity transform to determine an initial alignment. I then re-run the SfM optimization with an additional objective term to fit the specified DEM points. In the future, as more geo-referenced ground-based imagery becomes available (e.g., through websites like WWMX [149], Photosynth [109], or Flickr), this manual step will no longer be necessary.

3.5 Processing the recovered scene

The output of the reconstruction pipeline is organized into a scene description with the following elements:

- A set of points $P = \{p_1, p_2, \ldots, p_n\}$. Each point p_j has a 3D location, $\mathbf{X}_j$, and a color, $\mathrm{Color}(p_j)$, obtained by projecting the point into one of the images that observes it and sampling the color at that image point.
- A set of cameras parameters, $C = \{C_1, C_2, \ldots, C_k\}$.
- A *visibility mapping*, $\mathrm{Points}()$, between cameras and the points they observe. $\mathrm{Points}(C)$ is the subset of P containing the points observed by camera C. $\mathrm{Points}(C)$ is derived from the set of point tracks created in the correspondence estimation stage, and not from an analysis of occlusions (which would require dense geometry). Therefore, the visibility mapping is approximate. A point $p \in \mathrm{Points}(C)$ is very likely to be visible to camera C, whereas a point $q \notin \mathrm{Points}(C)$ is possibly visible to C. q may be outside the field of view or occluded (and thus correctly excluded from $\mathrm{Points}(C)$), or it may have not been correctly detected and matched (and incorrectly excluded).

After a scene is reconstructed, a few additional processing steps are required to clean the geometry and prepare the reconstructed scene description for viewing. These steps are:

1. Remove low-confidence points and cameras.
2. Remove radial distortion from the photos.
3. Estimate the up vector for the reconstruction.
4. Compute a set of proxy planes for use in scene rendering.
5. Optionally, detect 3D line segments.

Removing low-confidence points and cameras. Occasionally, due to bad matches or weak geometry, spurious points and misregistered cameras can appear in a reconstruction. The first processing step attempts to automatically detect and remove these undesirable points and cameras by identifying low-confidence geometry. A point p is deemed low-confidence if:

1. p is seen by fewer than three cameras; a point seen by only two views is much more likely to be spurious than one seen in three or more views, as the likelihood of finding a spurious point that is geometrically consistent with k views dramatically decreases as k grows.
2. The maximum angle over all pairs of rays that see p is less than a threshold $\theta_{\max}$ (points which are seen by views which are too close together can have very high uncertainty). I use $\theta_{\max} = 1.5^{\circ}$.

Points which meet either of these criteria are pruned from the point set P. After points have been pruned, images which now see fewer than sixteen points are removed as low-confidence views. This processing step often cleans up the geometry considerably.

Estimating the up vector. As described earlier, SfM can only recover geometry up to an unknown similarity transform, and thus the recovered scene can be in an arbitrary orientation; in particular, there is no natural up direction (often called the *up vector* in computer graphics). Thus, to make sure the scene is properly oriented, I estimate the up vector using the method of Szeliski [144]. This method uses the observation that most photos are taken without significant twist (rotation about the camera's viewing direction), and computes the up vector which minimizes the amount of residual twist over all cameras. This observation is largely correct, with the exception that cameras are sometimes rotated by $\pm 90^{\circ}$ (for portrait shots); such photos can appear sideways in image viewing software if the camera fails to properly orient them at the time they are taken. While such images can skew the up vector computation, they are also usually easy to identify, because in most collections the majority of photos are property oriented, and the remainder appear as outliers.

Removing radial distortion. For each registered photo I_j, a new, undistorted, image is created by applying the inverse distortion transform to I_j. It is not straightforward to compute the inverse transform in closed form from the distortion parameters κ_{1j} and κ_{2j}, so I compute an approximate inverse transform by fitting a degree-4 polynomial to sampled pairs of distorted and undistorted points.

Computing proxy planes for rendering. The visualization tools described in Part II set up virtual projection planes in the scene onto which images are projected for rendering. These planes should match the actual scene geometry (e.g., coincide with the sides of buildings) as well as possible. The tools in Part II require a projection plane for each camera (for scene rendering), as well as a projection plane for each *pair* of cameras (for creating transitions between two images):

- For each camera C_i, I compute a 3D plane, $\mathrm{Plane}(C_i)$, by using RANSAC to robustly fit a plane to $\mathrm{Points}(C_i)$.

- For each pair of neighboring cameras C_i and C_j (cameras which view at least three points in common), I compute a 3D plane, $\mathrm{CommonPlane}(C_i, C_j)$ by using RANSAC to fit a plane to $\mathrm{Points}(C_i) \cap \mathrm{Points}(C_j)$.

Many planar surfaces which occur in the world are vertical (walls) or horizontal (ground and ceiling planes). However, few images are taken looking directly at the ground or other horizontal surfaces, so I normally constrain $\mathrm{Plane}(C_i)$ and $\mathrm{CommonPlane}(C_i, C_j)$ to be vertical. This constraint often helps produce better renderings in the visualization tools. To enforce this constraint, I project all the points onto the ground plane, and fit a 2D line, rather than a 3D plane, to the projected points.

Detecting line segments. Finally, I detect 3D line segments using a method similar to that of Schmid and Zisserman [126]. This technique is described in Appendix C.

The scene representation, originally consisting of the point set P, the camera set C, and the visibility mapping, Points, is now augmented with:

- A set of 3D line segments $L = \{l_1, l_2, \ldots, l_m\}$ and a mapping, Lines, between cameras and sets of visible lines.
- The set of computed planes, $\mathrm{Plane}(C_i)$ and $\mathrm{CommonPlane}(C_i, C_j)$.

3.6 Results

I have applied my system to many different input photo collections, including uncontrolled Internet sets consisting of images downloaded from Flickr, and collections captured by a single person. In each case, my system detected and matched features on the entire set of photos and automatically identified and registered a subset corresponding to one connected component of the scene. This section gives results for several different scenes, including eight Flickr collections:

1. **Trevi Fountain**, a set of photos of the Trevi Fountain in Rome.
2. **St. Basil's**, photos of Saint Basil's Cathedral in Moscow.
3. **Mount Rushmore**, photos of Mount Rushmore National Monument, South Dakota.
4. **Sphinx**, photos of the Great Sphinx of Giza, Egypt.
5. **Hagia Sophia**, photos of the Hagia Sophia in Istanbul.
6. **Yosemite**, photos of Half Dome in Yosemite National Park.
7. **Colosseum**, photos of the Colosseum in Rome.
8. **Notre Dame**, photos of the Notre Dame Cathedral in Paris.

Three other sets were taken in more controlled settings (i.e., a single person with a single camera and lens):

1. **Great Wall**, a set of photos taken along the Great Wall of China.
2. **Prague**, photos of the Old Town Square in Prague.
3. **Annecy**, photos of a street in Annecy, France.

More information about these data sets (including the number of input photos, number of registered photos, CPU time, and average reprojection error), is shown in Table 3.1. The running times reported in this table were generated by running the complete pipeline on a 3.80GHz Intel Xeon machine with 4GB of core memory. While total CPU time is reported in Table 3.1, in practice, the keypoint detection and matching phases were run in parallel on ten such machines. Table 3.2 contains information on how much time was spent in each stage of the reconstruction pipeline for each data set. While the majority of CPU cycles were spent on image matching, this phase is much

Table 3.1: *Data sets and running times.* Each row lists information about each data set used: **Collection**, the name of the set; **Search term**, the Flickr search term used to gather the images; **# photos**, the number of photos in the input set; **# reg.** the number of photos registered; **# points**, the number of points in the final reconstruction; **CPU time**, the total CPU time for reconstruction; **error**, the mean reprojection error, in pixels, after optimization. The first eight data sets were gathered from the Internet, and the last three were each captured by a single person. In each subset, the rows are sorted by number of input views.

Collection	**Search term**	**# images**	**# reg.**	**# points**	**CPU time**	**error**
Trevi Fountain	trevi ∧ rome	466	370	114742	2.8 days	0.698
St. Basil’s	basil ∧ red square	627	197	25782	3.8 days	0.816
Mt. Rushmore	mount rushmore	1000	437	131908	10.2 days	0.444
Sphinx	sphinx ∧ egypt	1000	511	130182	10.2 days	0.418
Hagia Sophia	hagia sophia	1567	446	82880	12.2 days	0.395
Yosemite	halfdome ∧ yosemite	1882	678	264743	23.8 days	0.757
Colosseum	colosseum ∧ (rome ∨ roma)	1994	964	425828	23.3 days	1.360
Notre Dame	notredame ∧ paris	2635	598	305535	36.7 days	0.616
Great Wall	N/A	120	81	24225	0.3 days	0.707
Prague	N/A	197	171	38921	0.5 days	0.731
Annecy	N/A	462	420	196443	3.5 days	0.810

Table 3.2: *Running times for each stage of the structure from motion pipeline.* This table breaks out the total CPU time into time spent (a) detecting features in each image (**Feat. extract**), (b) matching features between each pair of images (**Matching**), and (c) structure from motion (**SfM**). The SfM stage was run on a single machine with a 3.8GHz Intel Xeon processor, and the feature detection and matching stages were run in parallel on ten such machines. The overall *wall clock time* spent during reconstruction is shown in the last column.

Collection	**# images**	**# reg.**	**Feat. extract**	**Matching**	**SfM**	**Total CPU**	**Total wall**
Trevi Fountain	466	370	1.6 hrs	2.0 days	15.6 hrs	2.8 days	20.7 hrs
St. Basil’s	627	197	2.1 hrs	3.7 days	1.2 hrs	3.8 days	10.3 hrs
Mt. Rushmore	1000	437	3.4 hrs	9.4 days	15.4 hrs	10.2 days	1.6 days
Sphinx	1000	511	3.4 hrs	9.4 days	15.7 hrs	10.2 days	1.6 days
Hagia Sophia	1567	446	5.2 hrs	11.7 days	6.6 hrs	12.2 days	1.5 days
Yosemite	1882	678	6.4 hrs	19.4 days	4.1 days	23.8 days	6.3 days
Colosseum	1994	964	6.8 hrs	18.9 days	4.1 days	23.3 days	6.3 days
Notre Dame	2635	598	9.0 hrs	33.1 days	3.2 days	36.7 days	12.7 days
Great Wall	120	81	0.3 hrs	3.2 hrs	2.5 hrs	0.3 days	2.8 hrs
Prague	197	171	0.4 hrs	8.7 hrs	2.2 hrs	0.5 days	3.1 hrs
Annecy	462	420	1.6 hrs	1.7 days	1.8 days	3.5 days	1.9 days

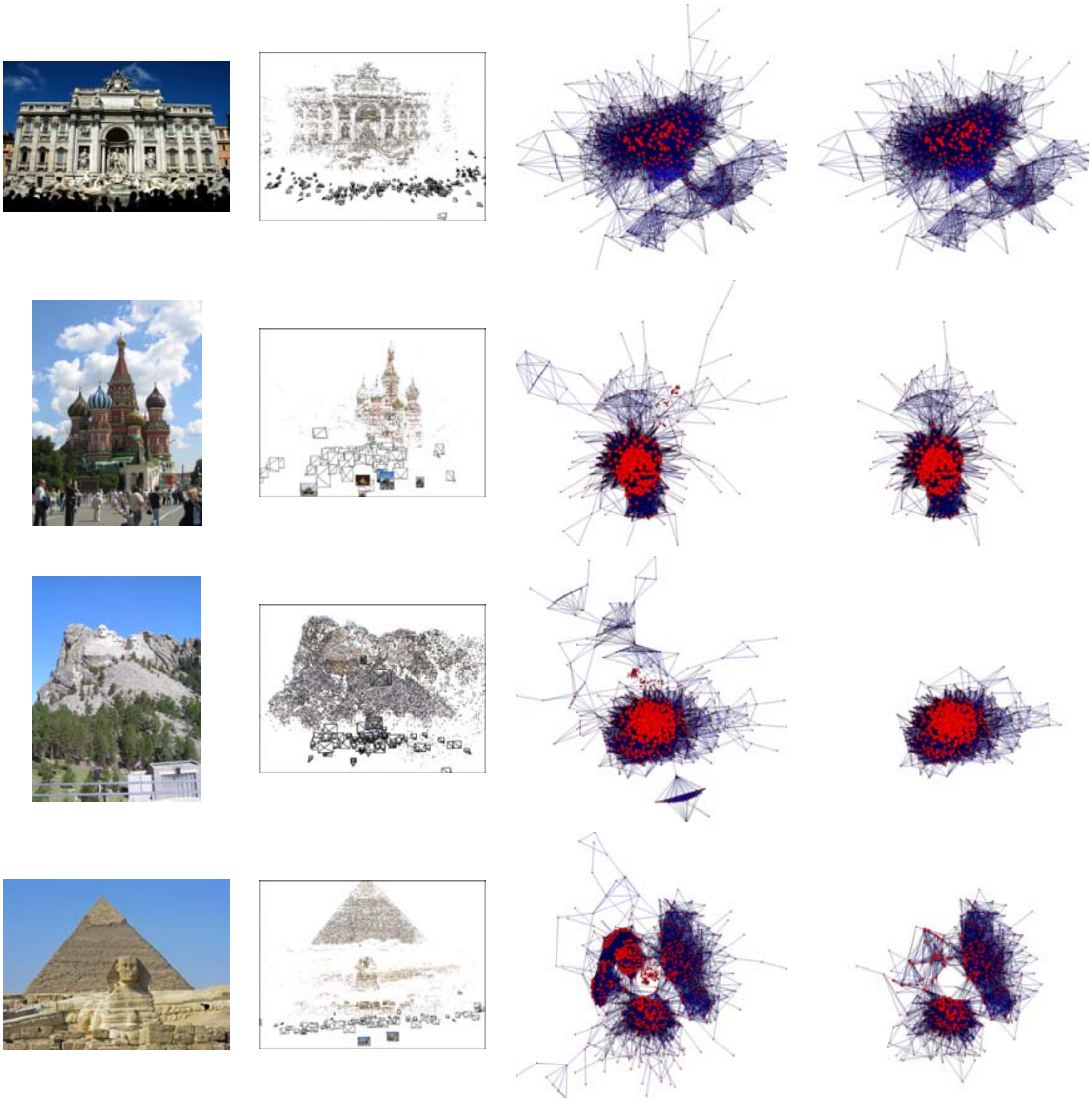
easier to parallelize than SfM, and in practice matching (on 10 machines) often took less wall clock time than SfM. Visualizations of these data sets are shown in Figures 3.10, 3.11, and 3.12.

These results demonstrate that the reconstruction system can robustly handle a variety of scenes: indoor scenes such as the Hagia Sofia, natural scenes such as Half Dome and Mount Rushmore, and urban scenes such as Annecy. However, these are just a small sample of the more than one hundred collections that have been successfully reconstructed with my system. My system has also been used by other researchers. Examples of reconstructions used in other systems, and the applications they have helped enable, can be found in [2], [10], and [55]. The system has also been used to aid in reconstruction of over twenty different scenes in Florence, Italy, for an installation by the artist Marnix de Nijs [31], and is the original basis for Microsoft's Photosynth [109], a online system for reconstructing and viewing scenes that has been applied to thousands of photo collections.

3.6.1 Discussion

The image connectivity graphs shown in the third column of Figures 3.10-3.12 suggest that there is a common pattern among Internet photo collections: they consist of several large clusters of photos, a small number of connections spanning clusters, and a sparse set of leaves loosely connected to the main clusters. The large clusters usually correspond to sets of photos from similar viewpoints. For instance, the large cluster that dominates the **Mount Rushmore** connectivity graph are all images taken from the observation terrace or the trails around it, and the two large clusters on the right side of the **Colosseum** connectivity graph correspond to the inside and the outside of the Colosseum. Sometimes clusters correspond not to viewpoint but to different lighting conditions, as in the case of the **Trevi Fountain** collection (see Figure 3.4), where there is a “daytime” cluster, a “nighttime” cluster, and a sparser set of links connecting the two.

The *neato* mass-spring system, used to embed the graph into the plane, acts in a way similar to multidimensional scaling: similar images will tend to be pulled together, and dissimilar images are pushed apart. This behavior can result in photos being laid out along intuitive dimensions. For instance, in the large daytime cluster at the top of the connectivity graph for the **Trevi** dataset, as well as the sparser cluster on the bottom, the x-axis roughly corresponds to the angle from which the fountain is viewed.



sample image reconstruction graph (matches) graph (reconstructed)

Figure 3.10: *Sample reconstructed Internet photo collections.* From top to bottom: **Trevi**, **St. Basil's**, **Rushmore**, and **Sphinx**. The first column shows a sample image, and the second column shows a view of the reconstruction. The third and fourth columns show photo connectivity graphs, in which each image in the set is a node and an edge links each pair of images with feature matches. The third column shows the photo connectivity graph for the full image set, and the fourth for the subset of photos that were ultimately reconstructed.

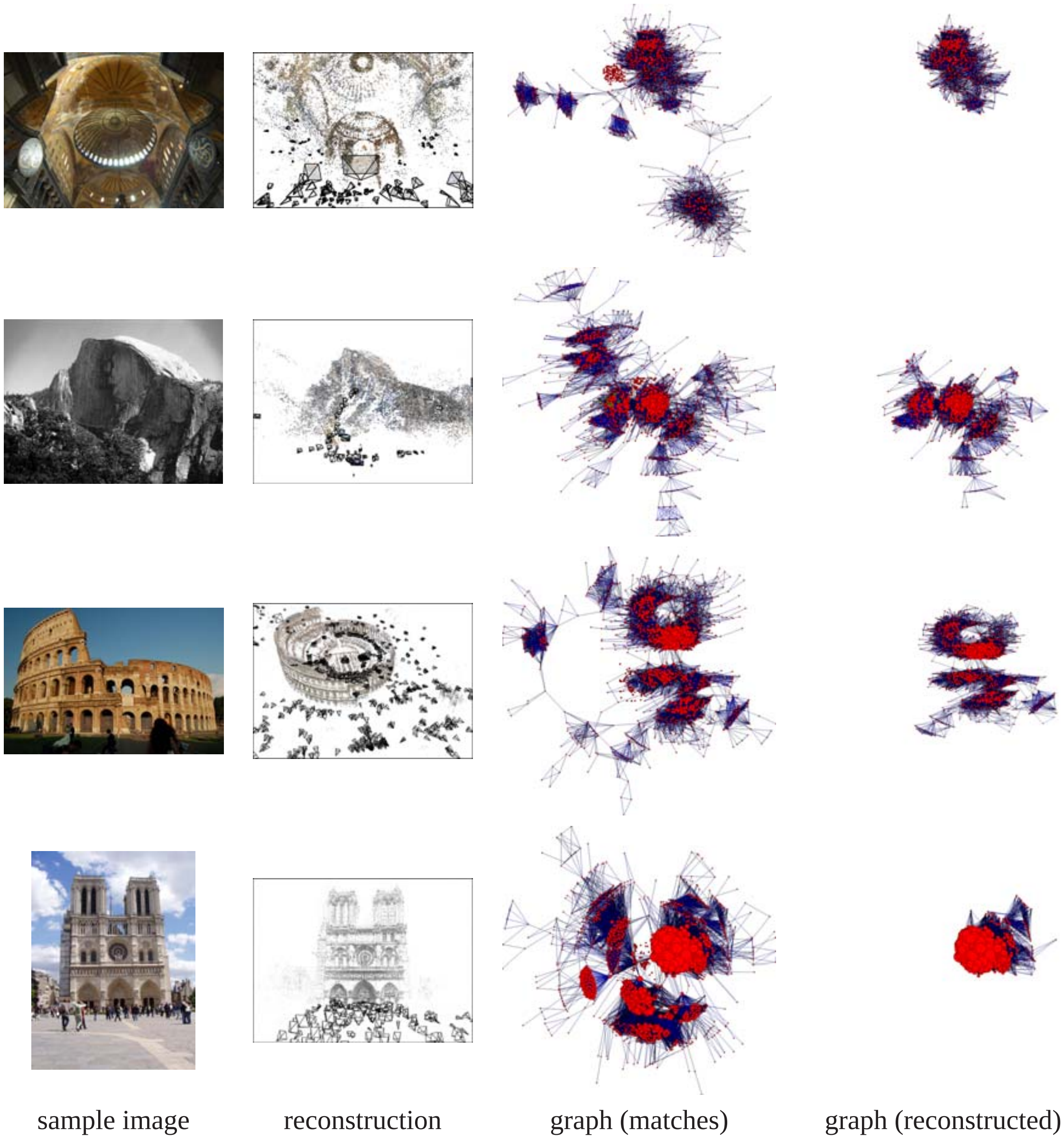


Figure 3.11: *More sample reconstructed Internet photo collections.* From top to bottom: **Hagia Sofia**, **Yosemite**, **Colosseum**, and **Notre Dame**.

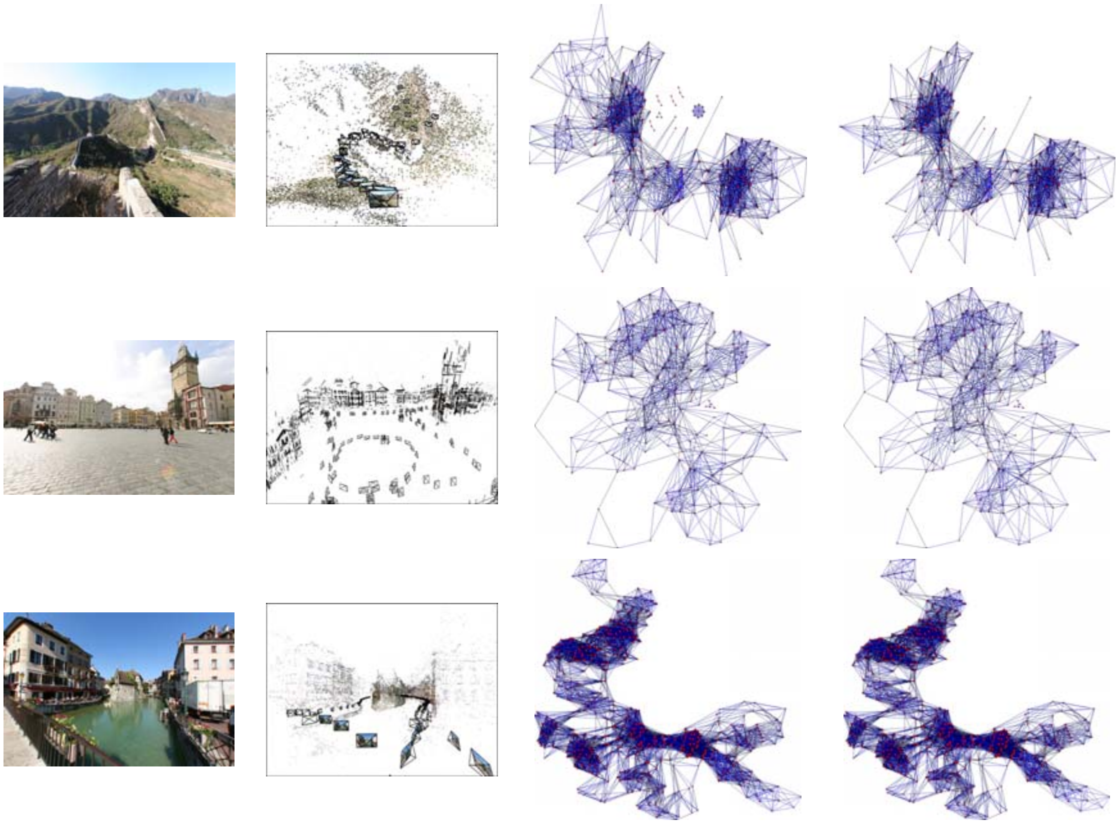


Figure 3.12: *Sample reconstructed personal photo collections.* From top to bottom: **Great Wall**, **Prague**, and **Annecy**. Each of these three photo collections were taken by a single person.

While the graphs in the third column of Figures 3.10-3.12 represent the connectivity of entire photo sets, the fourth column shows the subgraph the SfM algorithm was able to reconstruct. As described in Section 3.2, the algorithm does not, in general, reconstruct all input photos, because the input set may form separate connected components, or clusters that are too weakly connected to be reliable reconstructed together. These subgraphs suggest that for unstructured datasets, the reconstruction algorithm tends to register most of one of the main clusters, and can sometimes bridge gaps between clusters with enough connections between them. For instance, in the **Sphinx** collection, the SfM algorithm reconstructed two prominent clusters, one on the right side of the graph, and one on the bottom. These clusters correspond to two sides of the Sphinx (the front and the right side) which are commonly photographed; a few photos were taken from intermediate angles, allowing the the two clusters to be connected. Similarly, in the **Colosseum** collection, two weakly connected clusters corresponding to the inside and outside of the Colosseum were reconstructed. In general, the more connected the image graph, the greater the number of images that can successfully be registered.

For the controlled datasets (**Annecy**, **Prague**, and **Great Wall**), the photos were captured with the intention of generating a reconstruction from them, and thus the images are more uniformly sampled; as a result, the connectivity graphs are less clustered. In the **Prague** photo set, for instance, most of the photos were taken all around the Old Town Square, looking outward at the buildings. A few were taken looking across the square, so a few longer range connections between parts of the graph are evident. The SfM algorithm was able to register most of the photos in these datasets.

3.7 Failure modes

While a large percentage of the time the SfM algorithm produces qualitatively good reconstructions, it sometimes fails, resulting in bad, or no, geometry. I have observed four main failure modes:

1. **Insufficient overlap or texture**. The input images are simply too sparse, or too textureless, for the system to find sufficient correspondences between the images.
2. **Ambiguous, repeating textures**. Many scenes, especially urban environments, exhibit repetitive textures or symmetry. While my algorithm does well on such scenes surprisingly often, there are times when two parts of a scene are simply too similar, and are confused as identical

objects.

3. **Bad initialization**. As described in Section 3.2, the reconstruction of the initial image pair is critical; a bad start is almost impossible to recover from.
4. **Cascading errors**. Occasionally, the algorithm will make a mistake in the placement of a camera. Such errors can propagate and be compounded further along the pipeline as points and other images that depend on that view are added, leading to misestimation of large sections of a reconstruction.

I now discuss each of these failure modes, in turn.

Insufficient overlap or texture. To ensure enough overlap for reconstruction, a general rule of thumb is that every point in the scene must be visible, and must be matched, in at least three images (the "rule of three" [110]). Consider, for instance, a sequence of photos $I_1, I_2, \ldots, I_n$ of a row of flat building facades, taken by a person walking down a city street. The rule of three implies that pairs of images taken two apart, such as I_1 and I_3, must have some overlap, otherwise no point would be visible in more than two images. Thus, each pair of *neighboring* images, such as I_1 and I_2 must have more than 50% overlap.

Even when points are visible in enough images, they may not be linked up correctly due to limitations in feature detection and matching. For instance, my algorithm cannot reconstruct a complete scene from the images in the **nskulla** multi-view stereo data set, shown in Figure 3.13.[12] Three representative neighboring images (labeled A, B, and C) are shown in the figure; feature matches were found between A and B, as well as between B and C, but none were found between A and C. These two images are taken too far apart; their rotation angle about the skull is too large. Hence, there is no feature that was seen by A, B, and C at once—the rule of three is not satisfied. This is true of many triplets in the collection, and this collection could not be reconstructed. Interestingly, some multi-view stereo algorithms do well on this collection [54, 49].

Ambiguous, repeating textures. Many buildings exhibit a regularity that can confuse SfM. For instance, the Florence Duomo, shown in Figure 3.14, has an overall shape which is largely symmet-

[12] The **nskulla** set is available at Yasutaka Furukawa's collection of data sets at `http://www-cvr.ai.uiuc.edu/ponce_grp/data/mview/`.

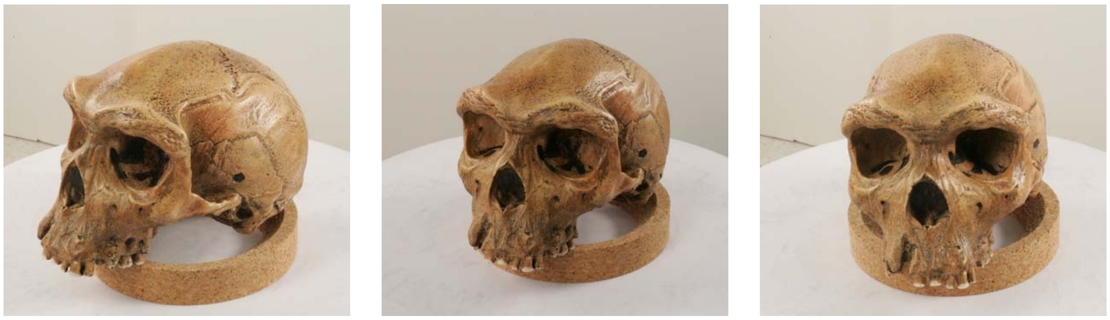


Image A Image B Image C

Figure 3.13: *Three neighboring images from the* **nskulla** *data set.* The structure from motion algorithm fails on this data set due to too little overlap between views. Images A and B have sufficient matches, as do B and C, but A and C have too few. Having matches between *triplets* of views is a requirement for the SfM algorithm.

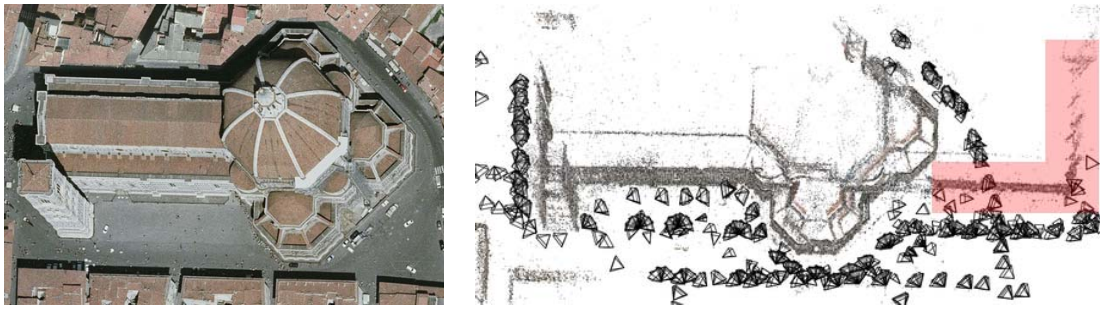


Figure 3.14: *Incorrect interpretation of the Florence Duomo.* Left: an overhead view of the Florence Duomo. Right: a reconstruction of the Florence Duomo from a collection of 460 images. The Duomo is mistakenly “unwrapped,” with the north and south sides of the cathedral joined together on the south side. That is, the north wall is erroneously reconstructed as an extension of the south wall, and the west wall appears again on the east side of the reconstruction; these problem areas are highlighted in red. This problem occurs because both sides of the dome area look very similar, and are confused as the same side by the SfM algorithm.

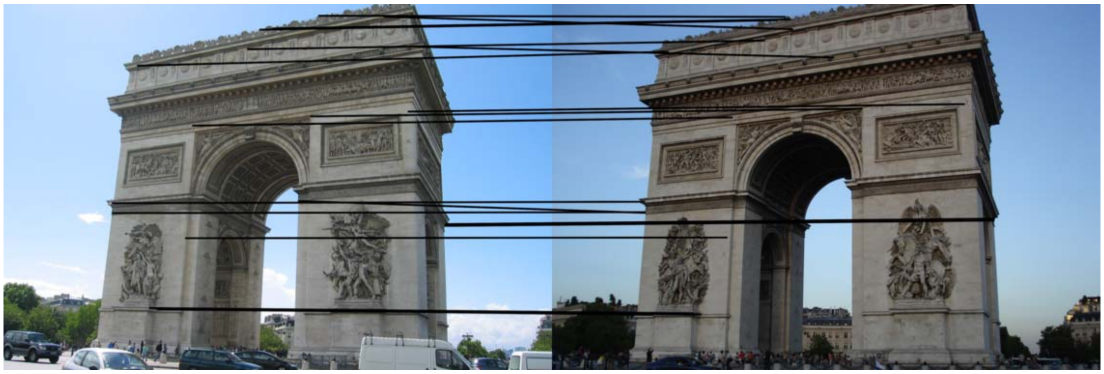


Figure 3.15: *Erroneous matches between images of the Arc de Triomphe.* The two images shown are of opposite sides of the monument (note the differences in the large sculptures on the right side of the images). Yet many features are extremely similar; lines are drawn between points mistakenly identified as matches during correspondence estimation. The SfM algorithm merges the two sides into one.

ric about its east-west axis, a dome which is symmetric about multiple axes, and a uniform striped texture on all sides. The reconstructed Duomo appears to be nearly twice as long as the real Duomo. What happened is that the matching algorithm matched images of the north side of the dome with images of the south side of the dome; thus both sides of the dome become the *same* side—it is as if the north side of the Duomo were picked up, rotated 180 degrees about the center of the dome, and set back down. The north wall then extends out towards the right, and the west wall is mirrored on the *east* side of the building.

Another example of erroneous matching is shown in Figure 3.15. As the figure shows, the two sides of the Arc de Triomphe are quite similar, though not identical, and images of different sides can occasionally have enough feature matches to be deemed matching images. My SfM algorithm thus merges the two sides into one.

Bad initialization. Another source of error is erroneous initialization, due to the sensitivity of the optimization to the reconstruction of the initial pair. There are two common ways the initialization can fail:

1. **Necker reversal**. This problem is a generalized version of the well-known Necker cube illusion in which a line drawing of a cube has two possible 3D interpretations. The same ambi-

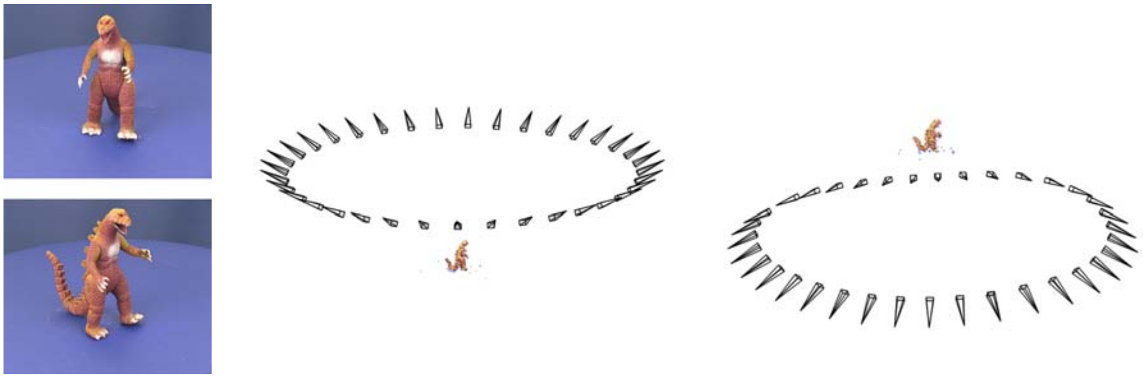


Figure 3.16: *Example of Necker reversal on the* **dinosaur** *sequence.* Left: two images from the data set. Middle: the correct interpretation of the scene. The images are taken above the dinosaur, looking down. Right: Necker-reversed solution. The images are reconstructed below the dinosaur, looking up. The Necker-reversed dinosaur is (approximately) a reflection of the correct solution. [Images used with thanks to Wolfgang Niem, University of Hanover, and the Oxford Visual Geometry Group.]

guity arises in multiple orthographic views of a 3D scene; two different, stable interpretations of the shape of the scene and positions of the cameras exist, hence there is an ambiguity in the reconstruction. For perspective images, the symmetry of the two solutions is broken; the correct interpretation will generally be the global minimum of the SfM objection function, but the Necker-reversed solution may still be a strong, stable local minimum. If the initial image pair is reconstructed with the incorrect, Necker-reversed solution, it is exceedingly likely that the final solution will also be reversed. An example of a Necker-reversed reconstruction (of the Oxford dinosaur sequence) is shown in Figure 3.16. When Necker reversal occurs, choosing a different starting pair generally solves the problem; alternatively, the user can explicitly tell the system to reverse the initial solution. Brown presents a different possible solution: trying out both interpretations for the initial pair and choosing the two-frame reconstruction with the lower reprojection error [15] to seed the rest of the reconstruction procedure.

2. **Insufficient parallax**. On occasion, the algorithm can pick a pair of images that have insufficient baseline to admit a well-conditioned reconstruction. Recall that to estimate the amount of parallax between an image pair, I fit a homography to the feature matches, and compute the percentage of points that are outliers. This estimate of parallax can sometimes be inaccurate;

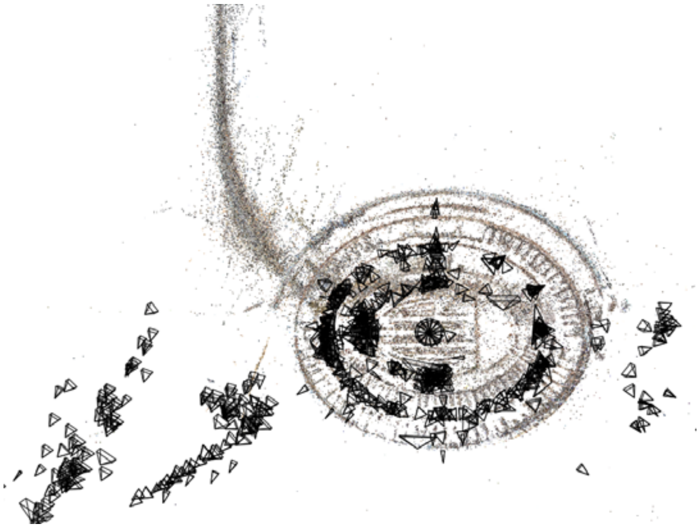


Figure 3.17: *Cascading errors in the Colosseum.* This overhead view of a reconstruction of the Colosseum shows evidence of a large problem. The interior is well-reconstructed, but the outer wall (the spray of points curving towards the top of the image) is jutting out from the interior at nearly right angles. This problem occurred because the inside and outside of the Colosseum are very weakly connected. The reconstruction began from the inside, and proceeded correctly until images of the outside began being added; at that point a few mistakes occurred and were compounded as the reconstruction of the outside grew. Interestingly, when the reconstruction is started from images of the outside of the Colosseum, this problem does not occur (see the reconstruction in Figure 3.11), suggesting that the order in which the images are added can make a difference.

for example, if a pair of images are taken close together (and therefore should be explained by a homography), but there are a large number of spurious matches, many of these spurious matches may be counted as outliers. Again, manually selecting a different starting pair can often solve this problem.

Cascading errors. When the algorithm makes a mistake in placing a camera, that error can propagate to later stages of the algorithm; points observed by that view can be triangulated wrongly, views which are in turn initialized by erroneous points will be misestimated, and so on. An example of this problem is shown in Figure 3.17, in which part of the Colosseum has been reconstructed quite poorly (part of the outside wall is jutting out from the rest of the Colosseum at nearly right angles). The most common reasons for this type of problem are twofold:

1. **Bad camera initialization.** If a new camera views a relatively small number of 3D points and enough of them are outliers, then the RANSAC algorithm for finding the initial pose of the camera can fail, resulting in an initial estimate with significant error.
2. **Large uncertainty.** Related to the initialization problem is the problem of large uncertainty in parameter estimates. Under certain circumstances, a camera can be initialized more or less correctly, but will have large uncertainty. For instance, a new camera which views a set of existing 3D points all clustered together in a small region of the image may not be estimated with high certainty. As another example, consider a view whose pose is estimated using only relatively distant points (on a building 100 meters away, say). Relative to the distance to that building, the uncertainty in the position of this camera might be low—perhaps one or two meters—but relative to much closer objects the uncertainty can be quite significant. If the view is used to triangulate points on an object three meters away, the points may have significant error (despite having low reprojection error). These problems can potentially be avoided if the system were extended to model uncertainty in parameter estimates.

Once a problem of this sort occurs, it can result in the reconstruction breaking apart into two irreconcilable pieces, one built before the mistake, and one after, as in the case of Figure 3.17, where the inner and outer parts of the Colosseum have been reconstructed in seemingly unrelated coordinate systems.

3.8 Conclusion

This chapter presented a structure from motion pipeline adapted to diverse collections of unordered images, demonstrated results on several data sets, and discussed common failure modes. The main contribution is the pipeline itself, the first to be demonstrated on large collections of photos obtained from Internet search.

Scale is still a primary challenge with structure from motion. Several reconstructions described in Section 3.6 took several days to compute; the Notre Dame reconstruction took nearly two weeks. Most of the CPU cycles are concentrated in two parts of the pipeline: the pairwise matching stage and the incremental SfM stage. Several researchers have investigated linear time methods for image matching using bag-of-words models [104, 24] to help address the matching problem. To improve the efficiency of SfM , several existing techniques could be applied, such as the work of Steedly, *et al.*, on applying spectral partitioning to SfM to reduce the number of parameters.

However, one significant aspect of the scale problem for Internet photo collections is that such collections are often highly redundant, with similar images appearing over and over again. Redundant views contribute little additional information to the reconstruction, but weigh the optimization down with extra parameters. In addition, such views lead to clusters of point tracks that are visible in many images, increasing the density of the reduced camera system. This increased density tends to slow down the non-linear optimization engine even further. The next chapter introduces a technique for reducing the amount of redundancy in a collection by intelligently selecting a subset of images to reconstruct. This technique can significantly improve the efficiency of SfM on large photo collections.

Chapter 4

SKELETAL GRAPHS FOR EFFICIENT STRUCTURE FROM MOTION

The basic structure from motion pipeline presented in the previous chapter demonstrates the possibility of adapting structure from motion (SfM) methods, originally developed for video, to operate successfully on unstructured Internet photo collections. The logical final step is to run this algorithm on the ultimate unstructured collection—all the images on the Internet. Unfortunately, the reconstruction algorithm of the last chapter—and others in the current generation of unstructured SfM methods [124, 15, 153]—simply do not scale to the thousands or tens of thousands of images typical of image search results, let alone the billions of images on the entire Internet. Taking just one example from Chapter 3, a single building, the Notre Dame cathedral, took nearly two weeks to reconstruct from 2,600 photos, in an optimization involving over 300,000 points and nearly a million parameters; a challenging task, and for just one building! While the pairwise matching stage took a significant fraction of this time, linear-time image matching algorithms, based on ideas from information retrieval, are beginning to appear [104, 24]. Thus, this chapter will focus on the SfM stage. While there exist techniques for scalable SfM for video, such as sub-sampling and hierarchical decomposition [46, 101], these are not directly applicable to Internet collections. Internet collections have very different properties from video: not only they unordered, but they also tend to be highly oversampled in some regions of popular viewpoints and undersampled in others, as can be observed in some of the image connectivity graphs shown in the previous chapter.

Intuitively, however, the difficulty of reconstructing a scene should depend on the complexity of the scene itself, not the number of images. For large Internet photo collections a much smaller subset of images may be sufficient to represent most of the information about the scene. If we could identify such a subset of views, we could focus our reconstruction effort on these images and produce truly scalable algorithms.

The key technical problem is to identify a subset of views that maximizes the accuracy and completeness of a reconstruction while minimizing the computation time. This is of course a tradeoff:

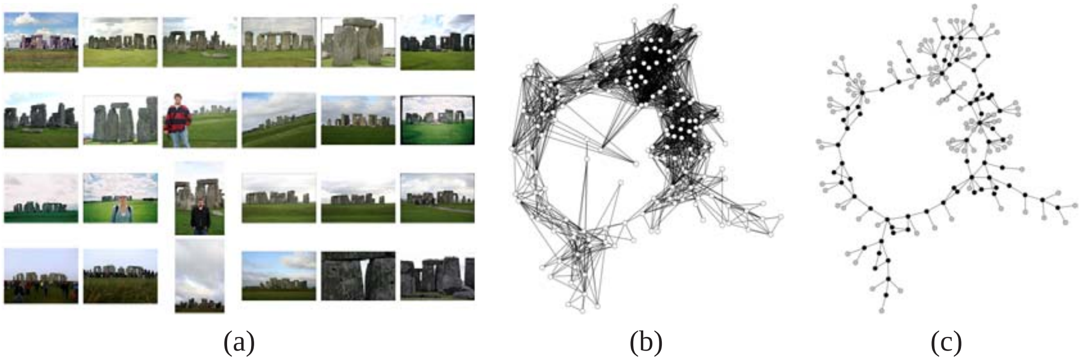


Figure 4.1: *Image graph and skeletal graph for the Stonehenge data set*. (a) A few images from the Stonehenge data set. (b) The image graph G for the full set of images from this data set. The graph forms a large loop due to the circular shape of Stonehenge, but is more densely sampled in some regions than others. (c) A *skeletal graph* for G. This graph retains the same overall topology, but is much sparser.

the more views we leave out, the faster we can reconstruct the scene, but the less complete and accurate the reconstruction is likely to be.

Even expressing this tradeoff by translating high-level concepts like *accuracy*, *completeness* and *run time* into mathematical objectives that can be optimized in the context of SfM is a challenging problem. However, we can get an intuitive sense of this tradeoff by examining the image connectivity graphs in Figure 4.1. This figure shows a connectivity graph for a set of images of Stonehenge; the full graph, not surprisingly, forms a ring, since people photograph Stonehenge from all sides. This graph is quite dense in some regions (e.g., near the top of the graph), but much less dense in others (such as the region near five o'clock). Now consider the graph on the right (which I refer to as a *skeletal graph*). This graph is much sparser than the full graph, but still retains the overall circular topology of the full graph and (nearly) spans the entire image collection; in some sense, it captures most of the information in the full graph, with many fewer edges.

To turn this intuition into a concrete optimization problem, I make two approximations. First, I optimize *uncertainty* instead of accuracy, since the former can be computed without knowledge of ground truth geometry. Second, I use the number of images as a proxy for run time, which enables an algorithm-independent measure of efficiency. Completeness is ensured via the constraint that the skeletal set must "span" the full set and enable reconstruction of all the images and 3D points in the full set using pose estimation and triangulation.

I formulate this problem by representing the joint uncertainty or covariance of a collection of images using a weighted version of the image connectivity graph, where each edge encodes the relative uncertainty in the positions of a pair of images. The distance between two images in this graph represents an approximate measure of the amount of uncertainty or information content connecting the images. The problem is then to determine a sparse skeletal subgraph with as many leaf nodes as possible (i.e., a small set of core interior nodes) that spans the full graph and *preserves distances* between images as much as possible. This formulation allows us to achieve a provable bound on the full covariance of the reduced skeletal reconstruction.

To achieve the desired bound on covariance, for every pair of images, I compare their estimated relative uncertainty in the original graph to the uncertainty in the skeletal graph, and require that the latter be no more than a fixed constant t times the former. While this problem is NP-complete, I develop a fast approach that guarantees this constraint on the covariance and in practice results in a dramatic reduction in the size of the problem.

My experiments show that this approach increases efficiency for large problems by more than an order of magnitude, still reconstructs almost all of the recoverable parts of an input scene, and results in little or no loss of accuracy in the reconstruction.

Related work My approach is closely related to research on intelligent subsampling of video sequences for SfM to improve robustness and efficiency [46, 101, 119]. The main difference in my work is that I operate on unordered collections with more complex topology than typical linear video sequences. Thus, my algorithm reasons about the structure of an image collection as a whole (represented as a graph), rather than considering subsequences of an input video.

My work is also related to selecting a set of representative *canonical views* for an image collection for applications such as robot localization [11] or summarizing large image collections [130]. Also closely related is the work of Krause and Guestrin [80] on using the property of *submodularity* to efficiently select a near-optimal subset from a large set of possible observations, where the goal is to maximize the coverage of the observations for applications such as environmental monitoring. In contrast to these previous applications, I am optimizing over a more complex set of criteria. I not only want to maximize coverage, but also the *accuracy* of the reconstruction. This leads to a different set of requirements on the selected observations, described in Section 4.1.

Uncertainty analysis is of critical importance in robotics and other fields where an autonomous agent is actively taking measurements to learn about its state and the state of its environment (called *active vision* when the agent uses cameras). In mapping and exploration applications, a robot must often keep track of not only its own state (position, orientation, etc.), but also its uncertainty in this state, and make observations which help to minimize this uncertainty as much as possible. Thus, researchers in active vision have studied the problem of selecting informative observations in an online setting, in order to make most efficient use of available information. For instance, in [30], Davison uses information value as a basis for selecting new measurements in a real-time tracking application. My overall goal is similar, but I am given a large number of possible measurements upfront, and select an optimal subset as a pre-process.

This chapter is organized as follows. In Section 4.1, I describe my approach for using *uncertainty* as a basis for computing a skeletal graph. In Section 4.2, I discuss how I compute covariances for image pairs, and in Section 4.3, I describe my algorithm for computing skeletal graphs. Finally, in Section 4.4 I present results and analysis for several large image sets, and in Section 4.5 I describe limitations and opportunities for future work.

4.1 Approximate uncertainty analysis using the image graph

Recall that SfM is the problem of building a reconstruction from a set of scene measurements (in the form of feature correspondences) obtained from a collection of images. The algorithm described in Chapter 3 uses all available measurements during scene reconstruction; the goal in this chapter is to reduce the number of measurements as much as possible, while still recovering a high-quality reconstruction.

How can we measure the quality of a reconstruction? Two desirable properties are *completeness* and *accuracy*; a reconstruction should span all parts of the scene visible in the images and should reflect the ground-truth scene and camera positions as closely as possible.

If completeness and accuracy were the only considerations, SfM should use all available measurements; the more information, the better.[1] However, *efficiency* is also important. For Internet

[1] Provided that the information fits our assumptions, e.g., independent measurements corrupted by Gaussian noise.

image sets, this tradeoff is particularly relevant. Such collections typically contain large numbers of popular, and therefore redundant, views, along with some rare, but important (for reconstruction) views. If this small set of important images could be identified, the scene could potentially be reconstructed much more quickly.

The main question is how to choose the subset of measurements to use. For simplicity, rather than considering individual measurements, I make decisions at the level of images. For each image I either include or exclude its measurements as a group. I call the set of images selected for reconstruction the *skeletal set* and roughly define the problem of selecting a skeletal set as follows: given an unordered set of images $\mathcal{I} = \{I_1, \ldots, I_n\}$, find a small subset $\mathcal{S} \subset \mathcal{I}$ that yields a reconstruction with bounded loss of quality compared to the full image set. Such a reconstruction will be an approximation to the full solution. Moreover, it is likely a good initialization for a final bundle adjustment, which, when run with all the measurements, will typically restore any lost quality.

To make this problem concrete, I must first define *quality*. Let us first consider the *completeness* of a reconstruction, the property that it spans the entire image collection. A complete reconstruction contains a camera pose for each image and a 3D point for each observed point track. However, it is sufficient to consider completeness in terms of cameras only; if all cameras are recovered, it must be possible to recover any observed point through triangulation. A reconstruction from the skeletal set cannot by itself be complete, as any camera not in the skeletal set will be missing. However, if a missing camera has significant overlap with an image in the skeletal set, it can likely be easily added after reconstructing the skeletal set using simple pose estimation. The graph theoretic concept of a *dominating set* (DS) captures this intuition. A dominating set of a graph $G(V, E)$ is a subset of nodes $S \subseteq V$ such that every node in V is either in S or adjacent to a node in S. An example of a dominating set is shown in Figure 4.2.

However, it is also important that the skeletal set be connected (assuming the input set is connected), so that it yields a single reconstruction rather than several disconnected reconstructions. A *connected dominating set* (CDS), or a dominating set that forms a single connected component, fits this requirement. Finally, a *minimum connected dominating set* is a CDS of minimum size. In this paper, I will use an equivalent definition of a minimum CDS called a *maximum leaf spanning tree* (MLST). An MLST of a graph G is the spanning tree T with the largest number of leaves; the interior nodes of T form a minimum CDS. The concepts of a dominating set, connected dominating

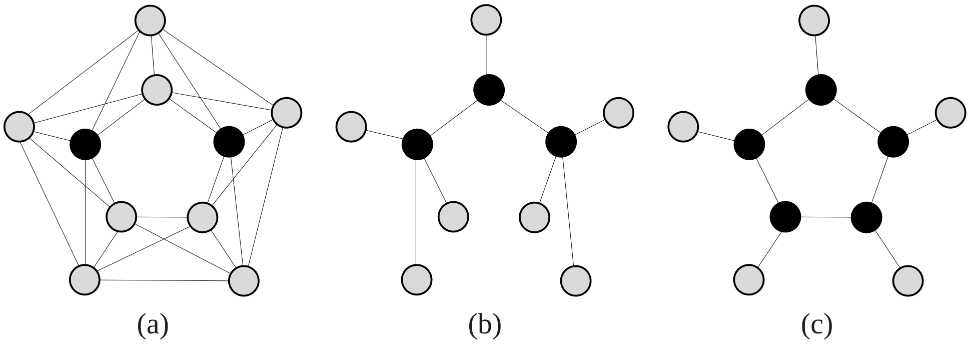


Figure 4.2: *Dominating sets and t-spanners*. (a) An unweighted graph G. The nodes colored black form a minimum dominating set. (b) A maximum leaf spanning tree (MLST) of G. The interior nodes (colored black) form a minimum connected dominating set. Note that this graph is also a 4-spanner of G. (c) A 3-spanner of G. In this case, the *skeletal set* (the set of interior nodes, colored black) has cardinality 5. This graph preserves distances in G more closely than the 4-spanner in (b).

set, and maximum leaf spanning tree are illustrated in Figure 4.2.

One possible way of choosing a skeletal set would thus be to find a MLST of the image connectivity graph, and take the interior nodes as the skeletal set. While this would satisfy the completeness property, such a skeletal set would not give any guarantees about the *accuracy* of the reconstruction. Thus, let us next consider accuracy, i.e., the property that the recovered cameras and points should be as faithful to the actual scene as possible. Without ground truth, it is impossible to measure accuracy directly. However, it is possible to estimate the *uncertainty* of a reconstruction, which is a statistical estimate of the accuracy.

Uncertainty describes the “looseness” of a system, or how easy it is to perturb the recovered parameters. A stiff reconstruction, where small perturbations result in a sharp increase in reprojection error, has relatively low uncertainty. Many reconstructions, however, suffer from *drift*. For instance, when running SfM on a set of images taken along a linear path (e.g., a set of images of building façades taken while walking down a long city street), the recovered camera path may be slightly curved rather than perfectly straight. This occurs because information from one end of the camera path to the other must travel a large distance, and errors in the recovered camera positions slowly accumulate along the length of the path. Another way to put the problem is that the system is relatively “loose” along a certain mode of variation—bending the path slightly will not result in a large increase in reprojection error—and therefore the uncertainty in the reconstruction is relatively high. If, on the other hand, the path of images forms a closed loop, rather than a linear chain, there will tend to be less uncertainty in the reconstruction, as information must travel at most halfway around the loop to connect any two images.

Uncertainty in a system is often modeled with a covariance matrix for the parameters being estimated (in this chapter, I will use the terms "uncertainty" and "covariance" interchangeably). For SfM, the covariance is rank deficient because the scene can only be reconstructed up to an unknown similarity transform (translation, rotation, and scale). This freedom in choosing the coordinate system is known as the *gauge freedom* [150]. Covariance can only be measured in a particular gauge, which can be fixed by anchoring reference features, e.g., by fixing the location and orientation of one camera, and constraining the distance to a second camera to be of unit length. Covariance is highly dependent on the choice of gauge. In the example of the linear chain of images, if the first camera is fixed, there is no uncertainty in its parameters, but there may be large uncertainty in the last camera in the chain; conversely, if the last camera were fixed, the uncertainty in the first camera's parameters could be quite large. When fixing a given camera I and computing the uncertainty in another camera J, we are really computing the uncertainty in J's parameters *relative* to image I; or, equivalently, how informative knowledge of the parameters of I is when estimating the parameters of J.

For this reason, I do not measure uncertainty in a single gauge, but rather fix each camera in turn and estimate the resulting uncertainty in the rest of the reconstruction; I want the relative uncertainty between *all* pairs of cameras to be as small as possible, to preserve the information flow in the camera network as much as possible. Since reconstructing the scene and measuring the actual covariance would defeat the purpose of speeding up SfM, I approximate the full covariance by computing the covariance in reconstructions of *pairs* of images and encoding this information in a graph. The global connectivity and "stiffness" of this graph captures information about the global uncertainty of the SfM system. I also only consider covariance in the cameras, and not in the points, as the number of cameras is typically much smaller than the number of points, and the accuracy of the cameras is a good predictor for accuracy in the points.

In particular, I define the *image graph* $G_{\mathcal{I}}$ on a set of images $\mathcal{I}$ as the graph with a node for

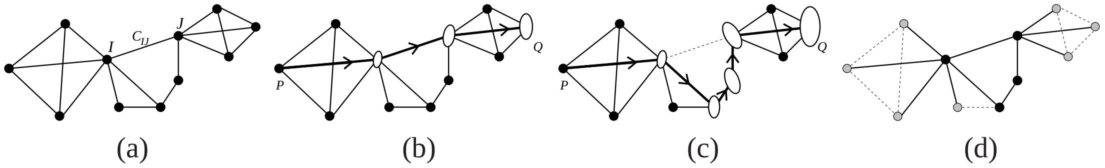


Figure 4.3: *Modeling covariance with an image graph.* (a) Each node represents an image, and each edge a two-frame reconstruction. Edge (I, J) is weighted with a covariance matrix C_{IJ} representing the uncertainty in image J relative to I. (b) To estimate the relative uncertainty between two nodes P and Q, we compute the shortest path between them by chaining up covariances (and taking the trace at the end). In this graph, the shortest path is shown with arrows, and ellipses represent the accumulated covariance along the chain. (c) If an edge is removed (in this case, the dashed edge), the shortest path from P to Q becomes longer, and therefore the estimated covariance grows. (d) A possible skeletal graph. The solid edges make up the skeletal graph, while the dotted edges have been removed. The black (interior) nodes form the skeletal set $\mathcal{S}$, and would be reconstructed first, while the gray (leaf) nodes would be added using pose estimation after $\mathcal{S}$ is reconstructed. In computing the skeletal graph, we try to minimize the number of interior nodes, while bounding the maximum increase in estimated uncertainty between all pairs of nodes P and Q in the original graph.

every image and two weighted, directed edges between any pair of images with common features.[2] Without loss of generality, I assume that $G_{\mathcal{I}}$ is connected (in practice, I operate on its largest connected component). Each edge (I, J) has a matrix weight C_{IJ}, where C_{IJ} is the covariance in the two-frame reconstruction (I, J) of the parameters of camera J when camera I is held fixed. In general, C_{IJ} could be a full covariance matrix with entries for both position and orientation. In my implementation, I model only the positional uncertainty of a camera, so C_{IJ} is a 3×3 matrix. Figure 4.3(a) shows an example image graph with covariance weights.

For any pair of cameras (P, Q), $G_{\mathcal{I}}$ can be used to estimate the uncertainty in Q if P is held fixed, by chaining together covariance matrices along a path between P and Q. To compute the exact covariance, information would need to be integrated along all paths from P to Q, and accumulated according to the equations given by Smith and Cheeseman [134]. However, the *shortest* path from P to Q gives an upper bound on the true covariance. Figure 4.3(b) illustrates this idea.

For the concept of "shortest path" to be well-defined, path lengths in $G_{\mathcal{I}}$ must be scalar-valued, rather than matrix-valued. I use the trace, $\mathrm{tr}(C)$, of the final covariance matrix as the scalar length of a path. The trace of a matrix is equal to the sum of its eigenvalues, so it expresses the magnitude

[2] $G_{\mathcal{I}}$ is a directed, weighted version of the image connectivity graphs introduced Chapter 3.

of the uncertainty. It is also a linear operator, i.e. $\mathrm{tr}(C_1 + C_2) = \mathrm{tr}(C_1) + \mathrm{tr}(C_2)$, and is invariant to rotation. Thus, adding up covariance matrices and taking the trace at the end is equivalent to adding up the traces of the individual covariances. I can therefore convert the covariance edge weights to scalar values, $w_{IJ} = \mathrm{tr}(C_{IJ})$.

The goal of the skeletal sets algorithm is to sparsify $G_{\mathcal{I}}$ as much as possible. What happens if we start removing edges from $G_{\mathcal{I}}$? As we remove edges, the lengths of some shortest paths will increase, as illustrated in Figure 4.3(c). On the other hand, removing edges from $G_{\mathcal{I}}$ yields a *skeletal graph* $G_{\mathcal{S}}$ that is more efficient to reconstruct. I estimate this efficiency by simply counting the number of *interior* (i.e., non-leaf) nodes in $G_{\mathcal{S}}$, since once we reconstruct the interior nodes, the leaves can easily be added in using pose estimation, and the leaves do not affect the overall connectivity of the graph. The objective is therefore to compute a skeletal graph with as few interior nodes as possible, but so that the length of any shortest paths (i.e., the estimated uncertainty) does not grow by too much.

There is an inherent trade-off in this formulation: the greater the number of edges removed from $G_{\mathcal{I}}$ (and the greater the number of leaves created), the faster the reconstruction task, but the more the estimated uncertainty will grow. I express this trade-off with a parameter, t, called the *stretch factor*. For a given value of t, the skeletal graph problem is to find the subgraph $G_{\mathcal{S}}$ with the maximum number of leaf nodes, subject to the constraint that the length of the shortest path between any pair of cameras (P, Q) in $G_{\mathcal{S}}$ is at most t times longer than the length of the shortest path between P and Q in $G_{\mathcal{I}}$. A subgraph $G_{\mathcal{S}}$ with this property is known as a *t-spanner* [5]; thus, the problem is to find a *maximum leaf t-spanner*. My algorithm for solving this problem is described in Section 4.3. Examples of t-spanners (for an unweighted graph) are shown in Figure 4.2.

A t-spanner subsumes the property of completeness, since if a node in $G_{\mathcal{S}}$ were to become disconnected from the rest of the graph, some shortest path in $G_{\mathcal{S}}$ would have infinite length. Furthermore, the skeletal graph will tend to preserve important topological features in $G_{\mathcal{I}}$, such as large loops, as breaking such structures will dramatically increase the distance between one or more pairs of nodes.

My approach is based on a simplified probability model. In particular, I consider only positional uncertainty, and use shortest path covariance as a bound on the full pairwise covariance. I make these simplifications so that the cost of making decisions is significantly smaller than the time saved during reconstruction [30]. These approximations produce skeletal sets that yield remarkably good reconstructions with dramatic reductions in computation time, as I demonstrate in the experimental results section.

Modeling higher-level connectivity. One issue with the above problem formulation is that the image graph $G_{\mathcal{I}}$ is not a sufficiently expressive model of image connectivity for SfM. To see why, consider three images A, B, and C, where A and B overlap, as do B and C, but A and C do not (violating the "rule of three" described in Section 3.7). These nodes form a connected subgraph in $G_{\mathcal{I}}$. However, these three images do not form a consistent reconstruction, because the scale factor between the two-frame reconstructions (A, B) and (B, C) cannot be determined (recall the problem with the **nskulla** data set in Section 3.7). In order to determine the scale factor, (A, B) and (B, C) must see at least one point in common. Therefore, any path passing through nodes A, B, C in sequence is not a realizable chain of reconstructions; I call such a path *infeasible*.

To address this problem, I define another graph, the *image pair* graph $G_{\mathcal{P}}$. $G_{\mathcal{P}}$ has nodes for every reconstructed *pair* of images, and edges between any two reconstructions that share common features.[3] $G_{\mathcal{P}}$ is also augmented with a node for every image, and is constructed so that a path between two images P and Q has the same weight as the analogous path in $G_{\mathcal{I}}$; the only difference is that only feasible paths can be traversed in $G_{\mathcal{P}}$.

More concretely, the vertex set of $G_{\mathcal{P}}$ is $\{(I, J) \,|\, (I, J) \text{ is an edge in } G_{\mathcal{I}}\} \cup \mathcal{I}$. Since $G_{\mathcal{I}}$ is directed, and each connected pair of nodes (I, J) in $G_{\mathcal{I}}$ has edges in both directions, $G_{\mathcal{P}}$ contains a node for (I, J) and a node for (J, I). The edge set of $G_{\mathcal{P}}$ contains two directed edges between each pair of reconstructions (I, J) and (J, K) that have common features. The weight on edge $[(I, J), (J, K)]$ is w_{IJ}. The sequence $(I, J), (J, K)$ represents a path from I to J to K; however, only w_{IJ} is represented in the edge weight. Thus, I add an extra edge from (J, K) to the special node K representing a terminal image in a path. Edge $[(J, K), K]$ has weight w_{JK}. There are also edges from each terminal node I to each reconstruction (I, J); these edges have weight zero. An example of an image graph and an image pair graph showing this construction is shown in Figure 4.4. To compute a shortest path between two nodes P and Q using $G_{\mathcal{P}}$, I find the shortest path between the image nodes P and Q. Additionally, I add a constraint that such terminal image nodes

[3] Note that $G_{\mathcal{P}}$ is closely related to the *line graph* of $G_{\mathcal{I}}$ [65].

can only appear at the ends of the path (i.e., another image node I cannot appear in the path from P to Q), as otherwise it is possible to find infeasible paths through $G_{\mathcal{P}}$.

This higher-level connectivity imposes an addition constraint on the skeletal graph: it must yield a single, feasible reconstruction. One way to express this is to define the *embedding* of a subgraph $G_{\mathcal{S}}$ of $G_{\mathcal{I}}$ into $G_{\mathcal{P}}$ as the subgraph of $G_{\mathcal{P}}$ containing the nodes corresponding to the edges of $G_{\mathcal{S}}$, and any edges between these nodes. The embedding of $G_{\mathcal{S}}$ into $G_{\mathcal{P}}$ must be connected for the skeletal graph to be feasible.

Overview of the reconstruction algorithm. The basic pipeline of the skeletal sets reconstruction algorithm is as follows:

1. Compute feature correspondence for the images, as in Section 3.1.
2. Compute a pairwise reconstruction and covariance matrices for each matching image pair, and create the graphs $G_{\mathcal{I}}$ and $G_{\mathcal{P}}$ (Section 4.2).
3. Construct a feasible maximum leaf t-spanner for $G_{\mathcal{I}}$ (Section 4.3).
4. Identify the skeletal set and reconstruct it using the reconstruction algorithm of Chapter 3.
5. Add in the remaining images using pose estimation.
6. Optionally, run a final bundle adjustment on the full reconstruction.

4.2 Building $G_{\mathcal{I}}$ and $G_{\mathcal{P}}$

As in the previous chapter, the first step is to finding correspondences by extracting SIFT features from each image [90], matching features between each pair of images, and forming connected components of matches to produce tracks. The matching step is extremely time-consuming on large data sets, but researchers are making significant progress on matching [104, 24], and I anticipate that much faster matching techniques will soon be available.

Once correspondences have been computed for an image collection, the next step of the skeletal sets approach is to create the image graph $G_{\mathcal{I}}$ and the pair graph $G_{\mathcal{P}}$. These graphs become the input to the skeletal graph algorithm described in the next section. The algorithm computes $G_{\mathcal{I}}$ and $G_{\mathcal{P}}$ in three stages: (1) it creates a two-frame reconstruction for every pair (I, J) of matching images, removing duplicate images as it goes, (2) it computes the relative covariance matrices C_{IJ}

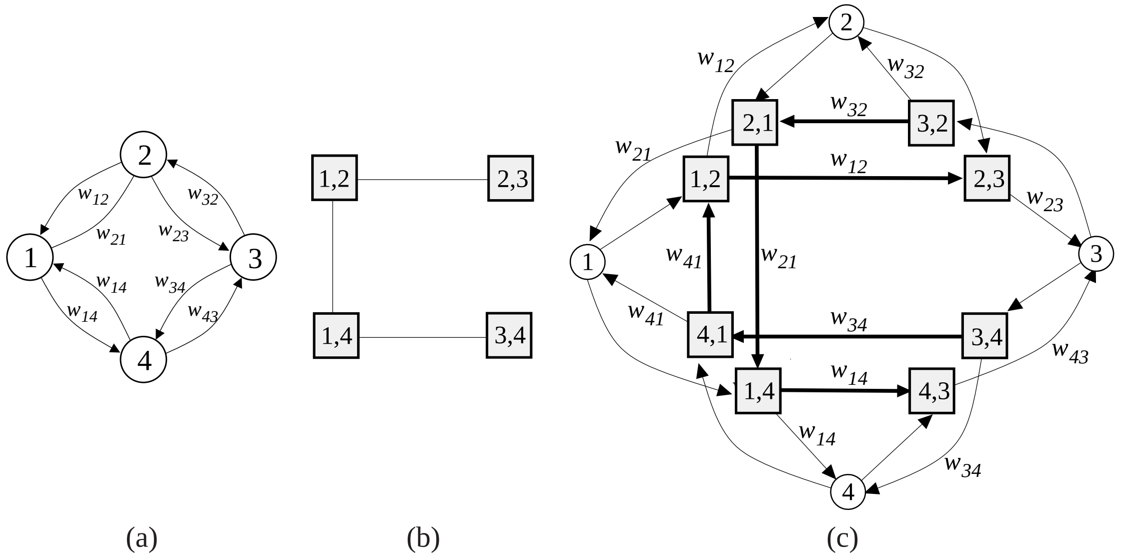


Figure 4.4: *Pair graph construction*. (a) An example image graph, with four images, showing the overlaps between images (1,2), (2,3), (3,4), and (1,4). (b) A possible (simplified) pair graph for (a), with a node for each pair of images. All pairs of reconstructions overlap, i.e., share some points in common, except for pairs (2,3) and (3,4). (c) An augmented pair graph with edge weights shown; the bold edges correspond to the edges in graph (b). This graph is augmented with a node for each image, and allows for computation of lengths of feasible paths between images, with the constraint that an image node (shown as a circle) can only appear at the ends of a path—only the bold edges of this graph can be used in a path, except at the very beginning and very end of the path. For instance, the only allowable path from image 2 to image 4 is a path through nodes 2,1 and 1,4. A path that contains image node 1 or 3 is disallowed.

and C_{JI} for each pair, and (3) it checks which pairs of two-frame reconstructions overlap (and are therefore edges in $G_{\mathcal{P}}$). These steps are now described in detail.

Two-frame reconstruction and duplicate detection. I first compute a reconstruction for each matching image pair using the five-point relative pose algorithm of Nistér [102] inside of a RANSAC loop. The five-point algorithm requires both cameras to be calibrated. Therefore, I only consider images that have a focal length estimate encoded in their Exif tags (and assume that each camera has unit aspect ratio, zero skew, and a principal point at the image center, as before). As described in Section 3.2.2 of the previous chapter, the focal length estimates in Exif tags are typically off by several percent (and are occasionally completely wrong); however, I have found that they are usually close enough to give reasonable pairwise reconstructions. After estimating the relative pose for a pair, I triangulate matches that are inliers to the recovered two-frame reconstruction and run bundle adjustment using the Sparse Bundle Adjustment package (SBA) [89]. If the average reprojection error of a reconstruction is too high (I use a threshold of 0.6 pixels), I discard the reconstruction, as these have usually been misestimated (due, for instance, to an erroneous focal length, or to other factors described in Section 3.7, "Bad initialization").

Once a pair has been reconstructed, I check whether the images are near-duplicates, i.e., whether they have very similar image content, or if one is subsumed by other. The reason for this check is that many images in Internet collections are very similar (and occasionally exactly identical), and much work can be saved by identifying such near-duplicates early on. I consider image J to duplicate image I if (a) they represent views taken very close together,[4] and (b) all the images connected to J in $G_{\mathcal{I}}$ are also connected to I (so that node J can be collapsed into node I without affecting the connectivity of the graph). If both of these criteria hold, it is likely that nearly all of the geometric information in image J is also present in image I, and the algorithm removes J from consideration (it will be added back in at the very end of the reconstruction process).

Without duplicate detection, the total number of pairs processed would be equal to the number of matching images, which could be as high as $O(n^2)$ for n images. With duplicate detection, the algorithm can often avoid processing many pairs. For instance, in the extreme case where all n images are the same, I will only process n pairs, rather than n^2. In practice, the total number of pairs processed depends on the order in which they are considered; if many duplicates are removed early on, fewer pairs will be processed. Therefore, observing that images that are more similar also tend to have more matching features, I sort the pairs by number of matches, and consider those with the most matches first. For the Internet collections I tested, typically about a third of the images are removed as duplicates.

Covariance estimation. Once I have reconstructed a (non-duplicate) pair (I, J), I estimate the relative covariances C_{IJ} and C_{JI} of the two camera positions. During bundle adjustment, SBA uses the Schur complement to factor out the 3D point parameters and compute the Hessian $\mathbf{H}_C$ of the reduced camera system [150]. To estimate C_{IJ} and C_{JI}, I invert $\mathbf{H}_C$ and select the blocks

[4] In particular, if the distance, d_c, between the camera centers is small compared to the median distance, d_p, between the cameras and the reconstructed points (I use $d_c < 0.025 d_p$).

corresponding to the camera positions of I and J. However, because of the gauge freedom, $\mathbf{H}_C$ is a singular matrix, so I first add constraints to fix the gauge and make $\mathbf{H}_C$ invertible. In particular, to estimate C_{IJ}, I fix the position and orientation of the reconstruction by constraining camera I to be at the origin with an identity rotation. I also fix the scale of the reconstruction by adding a soft constraint on the mean of the 3D points encouraging the mean point to stay at its current depth with respect to the first camera (the mean is computed after removing very distant points; in my implementation, I remove all points beyond 1.2 times the 90^{th} percentile distance). An additional penalty term equal to the difference between the target depth and current depth (weighted by a constant; I use 2) is added to the objective function to enforce this constraint.[5]

With these constraints, $\mathbf{H}_C$ is invertible, and can be used to compute C_{IJ}. C_{JI} is computed analogously by fixing camera J (in general, the two covariances are not identical, i.e., $C_{IJ} \neq C_{JI}$).

Constructing the pair graph $G_{\mathcal{P}}$. After computing covariances for every image pair, I construct the pair graph $G_{\mathcal{P}}$. Recall that every node of $G_{\mathcal{P}}$ represents a pairwise reconstruction, and that an edge connects every pair of overlapping reconstructions (I, J) and (J, K). The main remaining task is to determine this set of edges, i.e., to compute which pairs of reconstructions are connected by common points. To do so, I consider each triple of images (I, J, K) where (I, J) and (J, K) are reconstructions. I find the intersection of the point sets of (I, J) and (J, K), then use absolute orientation [69], inside a RANSAC loop, to find a similarity transform T between the two pairwise reconstructions. If there are at least a minimum number of inliers to T (I use 16), the two directed edges $((I, J), (J, K))$, and $((K, J), (J, I))$ are added to $G_{\mathcal{P}}$. The scale factors s_{ijk} and s_{kji} (where $s_{ijk}s_{kji} = 1$) between the two reconstructions are also stored with each edge, so that the algorithm can properly align the scales of adjacent reconstructions when computing shortest paths in $G_{\mathcal{P}}$, as described in Section 4.3.4.

Finally, I augment the graph $G_{\mathcal{P}}$ with nodes and edges for each image, as described in Section 4.1. Now, with $G_{\mathcal{I}}$ and $G_{\mathcal{P}}$ in hand, we are ready for the skeletal set algorithm.

[5] Another approach to fixing the scale is to constrain the distance between the cameras to be of unit length; however, this will remove any uncertainty in the translation direction, and can therefore bias the covariance. In particular, for image pairs where the translation is primarily “forward,” i.e. one camera is roughly in front of another, the uncertainty is typically largest in the translation direction, therefore zeroing out this uncertainty will make such pairs look more certain than they really are.

4.3 Computing the skeletal set

As mentioned in Section 4.1, I formulate the problem of computing a skeletal set as that of finding a (feasible) maximum leaf t-spanner of $G_{\mathcal{I}}$, called the *skeletal graph*, $G_{\mathcal{S}}$. The feasibility criterion requires that the embedding of $G_{\mathcal{S}}$ in $G_{\mathcal{P}}$ must be connected. To ensure that this constraint is satisfied, my algorithm maintains data structures for both $G_{\mathcal{S}}$ and its embedding in $G_{\mathcal{P}}$ when computing $G_{\mathcal{S}}$. Once $G_{\mathcal{S}}$ is found, the skeletal set $\mathcal{S}$ is found as the set of interior nodes of $G_{\mathcal{S}}$.

Unfortunately, the problem of computing a minimum t-spanner for general graphs is NP-complete [18], so it is unlikely that an exact solution to the maximum leaf t-spanner problem can be found efficiently. To get around this, I have developed an approximation algorithm for computing $G_{\mathcal{S}}$ which consists of two steps. First, a spanning tree $T_{\mathcal{S}}$ of $G_{\mathcal{I}}$ is constructed. The construction of $T_{\mathcal{S}}$ balances computing a tree with a large number of leaves (a maximum leaf spanning tree) with computing a tree with a small stretch factor (a t-spanner). Because no tree may have the desired stretch factor of t, the second step is to add additional edges to $T_{\mathcal{S}}$ to satisfy the t-spanner property. I now describe these two steps in detail.

4.3.1 Constructing the spanning tree

I begin by describing a simple, greedy approximation algorithm for computing a maximum leaf spanning tree (MLST) proposed by Guha and Khuller [61]. The idea behind the algorithm is to grow a tree $T_{\mathcal{S}}$ one vertex at a time, starting with the vertex of maximum degree.

Basic MLST algorithm. The algorithm maintains a color for each node. Initially, all nodes are unmarked (white), and the algorithms proceeds as follows:

1. Let $T_{\mathcal{S}}$ be initialized to the empty graph. Select the node v of maximum degree. Add v to $T_{\mathcal{S}}$, and color v black.
2. Add every unmarked neighbor of v, and the edge connecting it to v, to $T_{\mathcal{S}}$ and color these neighbors gray.
3. Select the gray node v with the most unmarked neighbors, color it black, and go to step 2, until all nodes are black or gray.

I first modify this algorithm to ensure that the constructed tree is feasible. To do so, I maintain a parent for each node (except the first). The parent $P(v)$ of a node v is the node that caused v to be colored gray. In step 2 of the algorithm, I only color a neighbor u of v gray if the path $(P(v), v, u)$ is feasible. Similarly, in step 3, when counting unmarked neighbors of a node v I only consider those for which $(P(v), v, u)$ is feasible. These constraints guarantee that the reconstructed tree is a feasible reconstruction.

4.3.2 Considering edge weights.

The basic MLST algorithm finds a spanning tree with a large number of leaves, but the stretch factor of this tree could be arbitrarily high, and many extra edges may need to be added to reduce the stretch factor to t. Thus, the spanning tree algorithm should ideally produce a tree with a stretch factor as close to the target value of t as possible, so as to minimize the number of extra edges required. It should therefore select nodes and edges that not only have high degree, but which are also critical for keeping distances between nodes as short as possible. Some images and edges are naturally more important than others. For instance, some nodes and edges might be on a large number of shortest paths, and removing them may result in large growth in distances between many pairs of nodes. On the other hand, some edges in a graph may not be along *any* shortest path, and are therefore relatively unimportant. Therefore, I integrate a notion of node and edge *importance* into the algorithm.

There are many possible ways of measuring the importance of a node to the global connectivity of a graph. Many measures of the *centrality* of a node have been proposed, including *betweenness* centrality, which measures the number of shortest paths that pass through a node, *eigenvector* centrality (a variant of which is used in Google's PageRank algorithm [106]), and *closeness*, which measures the average distance from a node to every other node. These measures can be expensive to compute, so in my algorithm, I take a very simple, local approach. I first measure the importance of an edge (I, J) by computing the length of the shortest feasible path between I and J in $G_{\mathcal{I}}$ (I denote this length $d_f(I, J; G_{\mathcal{I}})$), and dividing by the weight of (I, J) itself:

$$\mathrm{imp}(I, J) = \frac{d_f(I, J; G_{\mathcal{I}})}{w_{IJ}}.$$

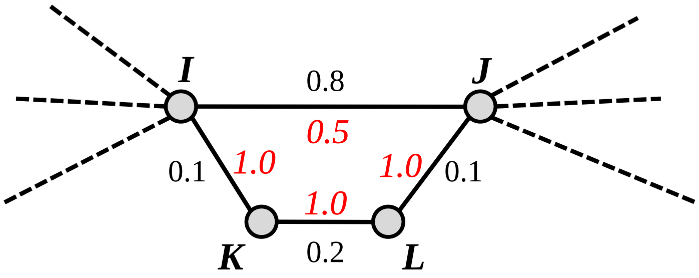


Figure 4.5: *Edge importance.* A four-node region of an undirected graph is shown. The edge weights are shown in normal typeface, and the importance of an edge is shown in red italics. Three of the edges ((I, K), (J, K), and (K, L)) have an importance of 1, because each of these edges is the shortest path between its endpoints. On the other hand, the shortest path between I and J is the sequence (I, K, L, J), which has length 0.4, while the length of edge (I, J) itself is 0.8. Because there is a path 0.5 times the length of (I, J), $\mathrm{imp}(I, J) = 0.5$. Edge (I, J) would be removed from the graph if the importance threshold $\tau > 0.5$.

If the edge (I, J) is itself a shortest path between I and J, $\mathrm{imp}(I, J) = 1$. Otherwise, $\mathrm{imp}(I, J) < 1$, and the longer the edge is compared to the shortest path, the smaller $\mathrm{imp}(I, J)$ will be. Some pairwise reconstructions (I, J) are naturally ill-conditioned, and a much higher certainty can be achieved via a detour through one or more other images. Such edges receive a low importance score. This definition of importance is illustrated in Figure 4.5.

While this is a much more local definition of importance than the others mentioned above, it has several advantages. First, it is relatively efficient to compute. Second, it gracefully handles edges that are *almost* shortest paths (unlike, e.g., the betweenness centrality measure, which would assign an importance of zero to an edge that is almost, but not quite, a shortest path). Finally, there is a connection between this measure of importance and the stretch factor of a graph, described below ("Setting the importance threshold").

Before running the basic MLST algorithm, I remove edges from $G_{\mathcal{I}}$ that have an importance score lower than a threshold τ, forming a pruned graph $G_{\mathcal{I}}{}'$. The degree of a node in $G_{\mathcal{I}}{}'$ is then the number of incident "important" edges, and is a better predictor for how important the node is than its "raw" degree in $G_{\mathcal{I}}$.

Setting the importance threshold. How should the importance threshold τ be set? The tradeoff is that with a very small threshold, very few edges will be pruned and the MLST algorithm will try to maximize the number of leaves in $T_{\mathcal{S}}$, without considering the stretch factor, as before. With a larger threshold, more edges will be pruned, and it may be more difficult to create a tree with a large number of leaves, but the stretch factor of the tree will likely be smaller. Is there a connection between t and τ?

Yes. Consider the case when $t = \infty$. Any spanning tree of $G_{\mathcal{I}}$ will satisfy this stretch factor, so we need not prune any edges ($\tau = 0$). Now consider a finite value of t, and suppose an edge (I, J) has an importance score $\mathrm{imp}(I, J) < \frac{1}{t}$. I claim that this edge will not be included in a minimum t-spanner of $G_{\mathcal{I}}$. To see why, first notice that, by the definition of importance, there must be a path between I and J in $G_{\mathcal{I}}$ that has length $\ell < \frac{w_{IJ}}{t}$. Thus, a t-spanner $G_{\mathcal{S}}$ of $G_{\mathcal{I}}$ must contain a path of length at most $t \cdot \ell$ between I and J. Edge (I, J) does not satisfy this requirement, because $w(I, J) > t \cdot \ell$. Therefore there must exist another, shorter path π between I and J in $G_{\mathcal{S}}$. As a result, *no* shortest path in $G_{\mathcal{S}}$ will contain edge (I, J), because a shorter path can be obtained with a detour through π. Any t-spanner containing edge (I, J) will therefore not be minimal, as (I, J) can be removed without changing the stretch factor.

Given this, we can safely remove from $G_{\mathcal{I}}$ all edges (I, J) such that $\mathrm{imp}(I, J) < \frac{1}{t}$ before constructing the spanning tree $T_{\mathcal{S}}$; none of these edges can appear in $G_{\mathcal{S}}$. In my implementation, I use a larger importance threshold of $\tau = \frac{4}{t}$ (clamped to a maximum of $\tau = 1$), thereby removing a larger number of weak edges.

4.3.3 From MLST to t-spanner

The tree $T_{\mathcal{S}}$ computed above spans the entire graph, but may not be a t-spanner. To guarantee that the stretch factor of the skeletal graph is at most t, additional edges may need to be added, forming a graph $G_{\mathcal{S}}$ with cycles. In order to determine which, if any, edges to add, the algorithm must first test whether the target stretch factor has been met. How can we verify that a graph is a t-spanner? One way would be to take every pair of nodes I and J, compute the shortest paths between I and J in both $G_{\mathcal{I}}$ and $G_{\mathcal{S}}$, and compare the lengths of the two shortest paths. However, it is not necessary to check shortest paths between *all* pairs of nodes to verify that a subgraph is a t-spanner; it suffices to only check shortest paths between all neighboring nodes (I, J) in $G_{\mathcal{I}}$. The reasoning behind this is that if the distance between all *neighboring* nodes in a graph is dilated by at most a constant factor

t, the distance between *any* two nodes must also be dilated by at most a factor t, because each edge on the original shortest path can be replaced by a new path at most t times longer.

I therefore enumerate all edges of $G_{\mathcal{I}}$ not included in the tree $T_{\mathcal{S}}$. For each edge (I, J), I compute the feasible distance between I and J in both graphs, denoting these distances $d_f(I, J; G_{\mathcal{I}})$ and $d_f(I, J; G_{\mathcal{S}})$. If $t \cdot d_f(I, J; G_{\mathcal{I}}) < d_f(I, J; G_{\mathcal{S}})$, I add edge (I, J) to $G_{\mathcal{S}}$; otherwise, I omit the edge. Once this procedure is complete, all neighboring nodes (I, J) of $G_{\mathcal{I}}$ must satisfy the t-spanner property, and therefore $G_{\mathcal{S}}$ must be a t-spanner of $G_{\mathcal{I}}$.

The set of edges added to $G_{\mathcal{S}}$ depends on the order in which the edges are processed, since adding a single edge can affect many shortest paths, obviating the need to add other edges later on. Therefore, I first consider edges between nodes that are already on the interior of $G_{\mathcal{S}}$ (i.e., the black nodes of the MLST algorithm). I then follow the heuristic of Älthofer, *et al.*, [5] and consider the remaining edges in order of increasing covariance weight.

Putting it all together. Once $G_{\mathcal{S}}$ has been augmented with the necessary edges, the skeletal set $\mathcal{S}$ is selected as the set of non-leaf nodes of $G_{\mathcal{S}}$. The skeletal set is then reconstructed with the incremental structure from motion technique described in the previous chapter. Next, the remaining (leaf) images are added using pose estimation [68], forming a full reconstruction. Bundle adjustment is then optionally run on the full reconstruction.

In summary, the skeletal sets algorithm has the following steps:

1. Compute feature correspondences for the images.
2. Compute a reconstruction and covariances for each matching image pair, and remove duplicate images (Section 4.2). Create the graphs $G_{\mathcal{I}}$ and $G_{\mathcal{P}}$.
3. Compute an importance score for each edge of $G_{\mathcal{I}}$, and prune edges with low importance (Section 4.3.2)
4. Construct a MLST from $G_{\mathcal{I}}$ (Section 4.3.1).
5. Add edges to guarantee the stretch factor (Section 4.3.3).
6. Identify and reconstruct the skeletal set.
7. Add in the remaining images using pose estimation.
8. Optionally, run bundle adjustment on the full reconstruction.

4.3.4 Implementation of shortest path computation in $G_{\mathcal{P}}$.

Shortest paths are computed in two steps of the skeletal set algorithm: (1) step 3, computing the importance of each edge in the graph, and (2) step 5, determining which edges must be added to $T_{\mathcal{S}}$ to achieve the stretch factor. These paths are computed in the graph $G_{\mathcal{P}}$ so that only feasible paths are considered.

The shortest path between two nodes in a graph $G = (V, E)$ can be computed using Dijkstra's algorithm [35] in $O(|E| + |V| \log |V|)$ amortized time by using a Fibonacci heap [48] to find the minimum-distance node in each iteration of the algorithm. I compute shortest paths in $G_{\mathcal{P}}$ with a modified version of this algorithm. The main difference comes from the fact that the covariance weights on the edges of $G_{\mathcal{P}}$ are derived from reconstructions in different coordinate systems, so the covariances are not directly comparable. Thus, I use the scale factors computed when constructing the pair graph to scale the edge weights during the shortest path computation (the edge weights are scaled by the square of the scale factors, as the trace of a covariance matrix grows with the square of the scene scale). In addition, for each image I, care must be taken to make sure that all outgoing edges have weights measured in the same coordinate system. Therefore, as a preprocess to the skeletal sets algorithm, for each image I I select a reconstruction (I, J) to be the canonical coordinate system for I, and align all other reconstructions (I, K) to (I, J). Not all reconstructions (I, K) may overlap with (I, J), but through transitivity most can be aligned (I remove any remaining unaligned reconstructions).

Note that the shortest path algorithm can often be terminated early (i.e., it is not always necessary to find the exact shortest path between two nodes). In the importance pruning stage, if at any time a path between I and J in $G_{\mathcal{I}}$ shorter than $\tau \cdot w_{IJ}$ is found, (I, J) can immediately be pruned. Similarly, in the stretch factor stage, if any path between I and J in $T_{\mathcal{S}}$ shorter than $t \cdot w_{IJ}$ is found, we know that (I, J) can be omitted from $T_{\mathcal{S}}$.

4.4 Results and discussion

I have evaluated my algorithm on several large Internet photo collections of famous world sites (St. Peter's Basilica, Stonehenge, the Pantheon, the Pisa Duomo, Trafalgar Square, and the Statue of Liberty). I obtained these data sets by doing keyword searches on Flickr and downloading the results.

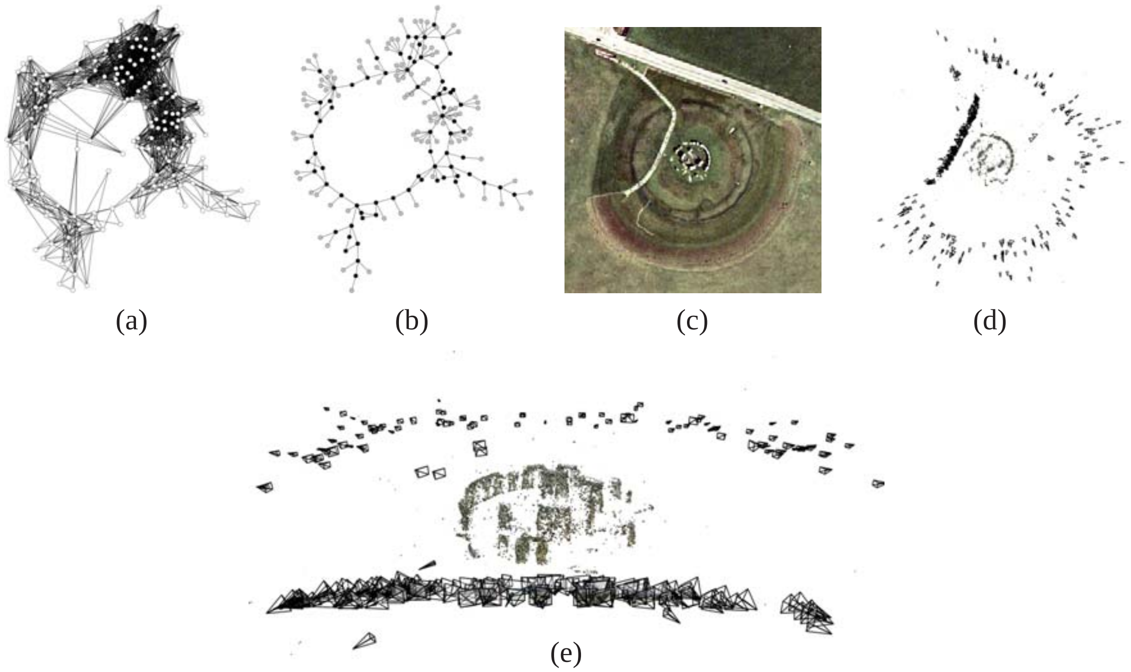


Figure 4.6: *Reconstruction of Stonehenge*. (a) The full image graph for Stonehenge and (b) the skeletal graph (for $t = 16$). The black (interior) nodes of (b) comprise the skeletal set, and the gray (leaf) nodes are added in later. Note that the large loop in the graph is preserved. (The graph layouts are not based on the physical position of the cameras, but are created with the *neato* tool in the Graphviz package). (c) Aerial photo. (d-e) Two views of the Stonehenge reconstruction. The reconstructions show the recovered cameras, rendered as small frusta, in addition to the point cloud.

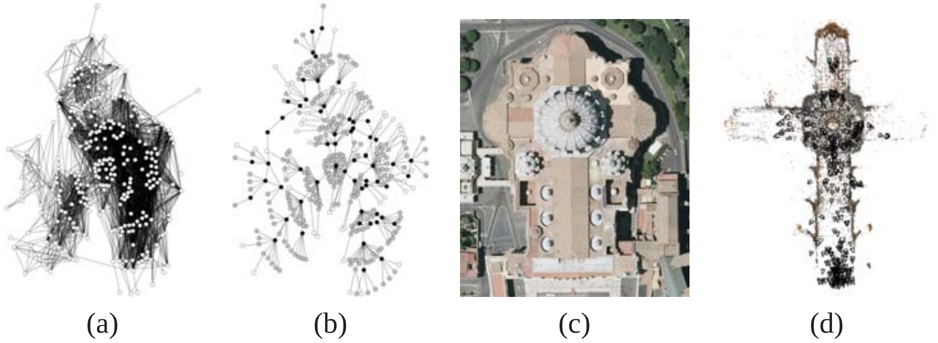


Figure 4.7: *Reconstruction of the interior of St. Peter's*. (a) The full image graph for St. Peter's and (b) our skeletal graph (for $t = 16$). In this graph, white nodes represent images found to be duplicates. These nodes are removed before computing the skeletal graph. (c) Overhead photo of St. Peter's. (d) The final reconstruction.

Table 4.1: *Data sets and running times.* Each row lists: **name**, the name of the scene; **# images**, the number of input images; **largest cc**, the size of the largest connected component of the image graph; $|\mathcal{S}|$, the size of the computed skeletal set; **#reg full**, the number of images registered in the full reconstruction; **#reg** $\mathcal{S}$, the number of images registered in the skeletal set reconstruction; **rt full**, the running time of the full reconstruction; **rt** $\mathcal{S}$, the running time of the skeletal set reconstruction, including computing the pairwise reconstructions and the skeletal graph; **rt** $\mathcal{S}$**+BA**, the running time of the skeletal set reconstruction plus a final bundle adjustment. The columns in bold represent the running times for the full reconstruction with both the baseline method and the skeletal sets algorithm.

Name	# images	largest cc	$\lvert\mathcal{S}\rvert$	#reg full	#reg $\mathcal{S}$	**rt full**	rt $\mathcal{S}$	**rt** $\mathcal{S}$**+BA**
Temple	312	312	54	312	312	**46 min**	58.5 min	**63.5 min**
Stonehenge	614	490	72	408	403	**276 min**	14 min	**26 min**
St. Peter's	927	471	59	390	370	**11.6 hrs**	3.54 hrs	**4.85 hrs**
Pantheon	1123	784	101	598	579	**108.4 hrs**	7.58 hrs	**11.58 hrs**
Pisa1	2616	1671	298	1186	1130	**17.8 days**	14.68 hrs	**22.21 hrs**
Pisa2	1112	1110	352	1101	1093	**18.5 days**	32.9 hrs	**37.4 hrs**
Trafalgar	8000	3892	277	-	2973	$>$ **50 days**	17.78 hrs	**30.12 hrs**
Liberty	20931	13602	322	-	7834	$>$ **50 days**	15.59 days	N/A[a]

[a] Running bundle adjustment on the full reconstruction was not possible due to memory constraints.

I also tested on a second collection of images of the Pisa Duomo taken by a single photographer with the express purpose of scene reconstruction (I refer to the Internet collection as Pisa1, and the single-photographer collection as Pisa2). Finally, I evaluated the algorithm on the **Temple** data set. This 312-image collection is taken from the multi-view stereo evaluation data of Seitz, *et al.* [128], and was captured in the lab using the Stanford spherical gantry, so the camera poses are known (the camera used was also calibrated, so the intrinsics are known as well). The images are regularly sampled on a hemisphere surrounding a plaster reproduction of the Temple of the Dioskouroi.

I reconstructed each data set using the skeletal graph algorithm with a stretch factor $t = 16$. Visualizations of the full and skeletal image graphs for the Stonehenge data set are shown in Figure 4.6, for St. Peter's in Figure 4.7 and for the Pantheon in Figure 4.8. Note that the skeletal graphs are much sparser than the original graphs, yet preserve the overall topology. For instance, the large loop in the full image graph for Stonehenge is retained in the skeletal graph.

Figure 4.8 shows overhead views of the Pantheon during several stages of our algorithm; note that both the inside and outside are reconstructed (connected by images that see through the door-

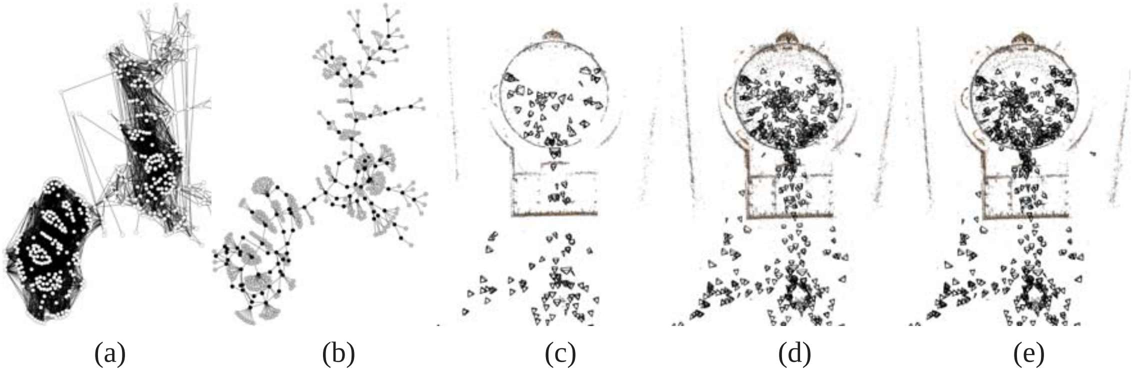


Figure 4.8: *Reconstructions of the Pantheon.* (a) The full image graph for the Pantheon and (b) our skeletal graph. The black (interior) nodes of (b) comprise the skeletal set, and the gray (leaf) nodes are added in later. The Pantheon consists of two dense sets of views (corresponding to the inside and outside), with a thin connection between them (views taken outside that see through the door). Note how the skeletal set preserves this important connection, but sparsifies the dense parts of the graph. (c) Reconstruction from the skeletal set only. (d) After using pose estimation to register the remaining images. (e) After running bundle adjustment on (d).

way). Views of the reconstruction of Stonehenge are shown in Figure 4.6, St. Peter's in Figure 4.7, Pisa1 in 4.9, Pisa2 in Figure 4.10, and Trafalgar Square in Figure 4.11.

Table 4.1 summarizes the running time for each data set (excluding the correspondence estimation stage). The running times reported for my algorithm are for the entire skeletal sets pipeline (computing pairwise reconstructions, building the skeletal graph, and reconstructing the scene). For Trafalgar and the Statue of Liberty (the largest sets), the baseline method was still running after 50 days.

The results show that the skeletal sets method (with the exception of the Temple dataset, to be discussed shortly) reconstructs scenes significantly faster than the baseline method, and the performance gain generally increases dramatically with the size of the data set. The speedup ranges from a factor of 2 for St. Peter's, to a factor of at least 40 for Trafalgar Square. At the same time, the algorithm recovers most of the images reconstructed by the baseline method. A few images are lost; most of these are very tenuously connected to the rest, and can mistakenly be pruned as infeasible while building the skeletal graph. My method also works well on the set taken by a single person (Pisa2), although the fraction of images in the skeletal set for this collection is higher than for the

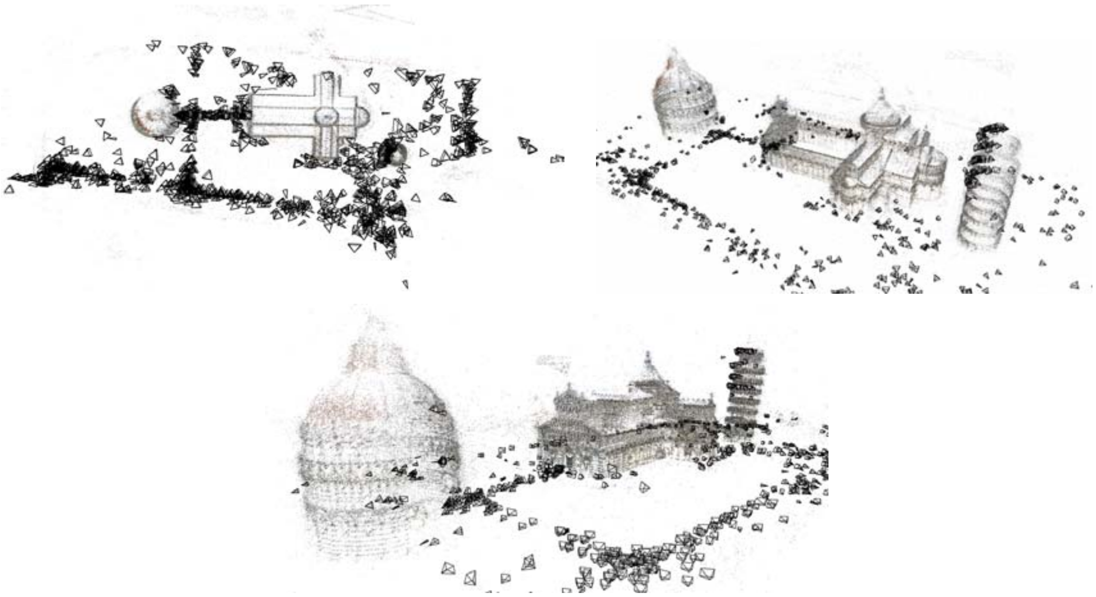


Figure 4.9: Several views of the Pisa1 reconstruction.

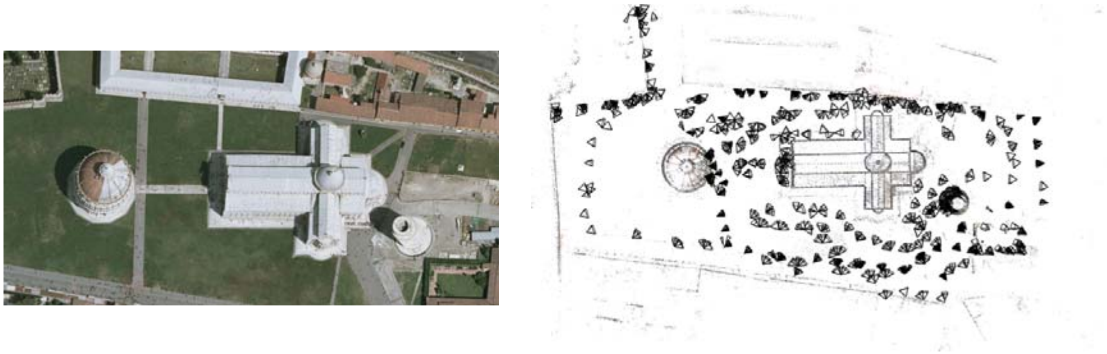


Figure 4.10: Overhead photo and view of the Pisa2 reconstruction.

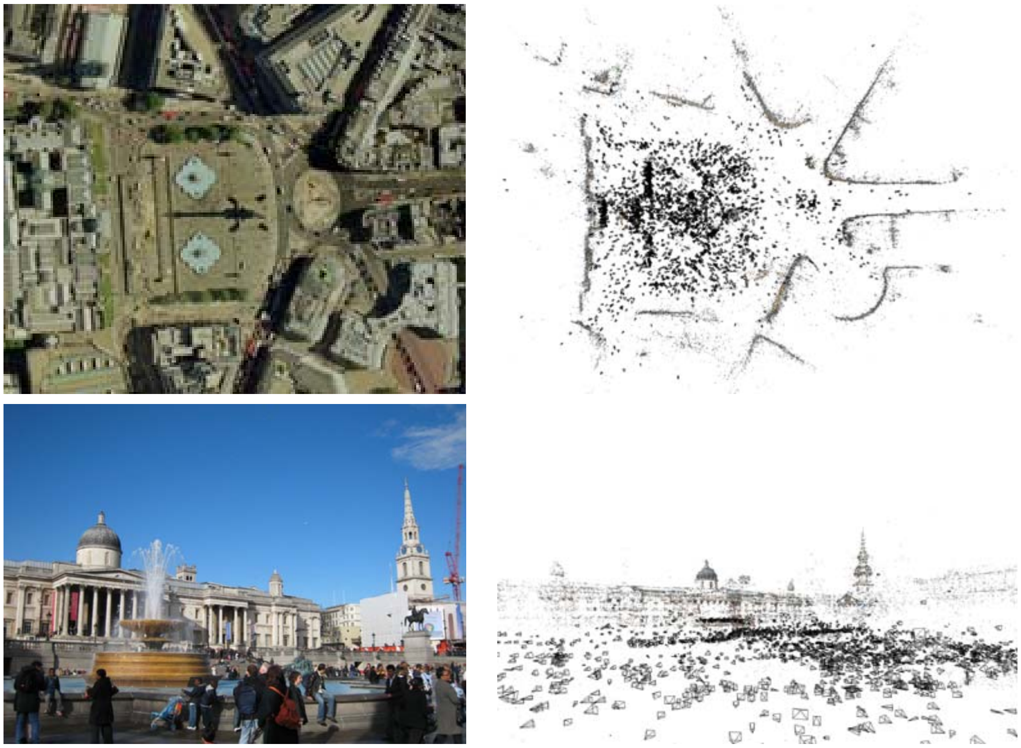
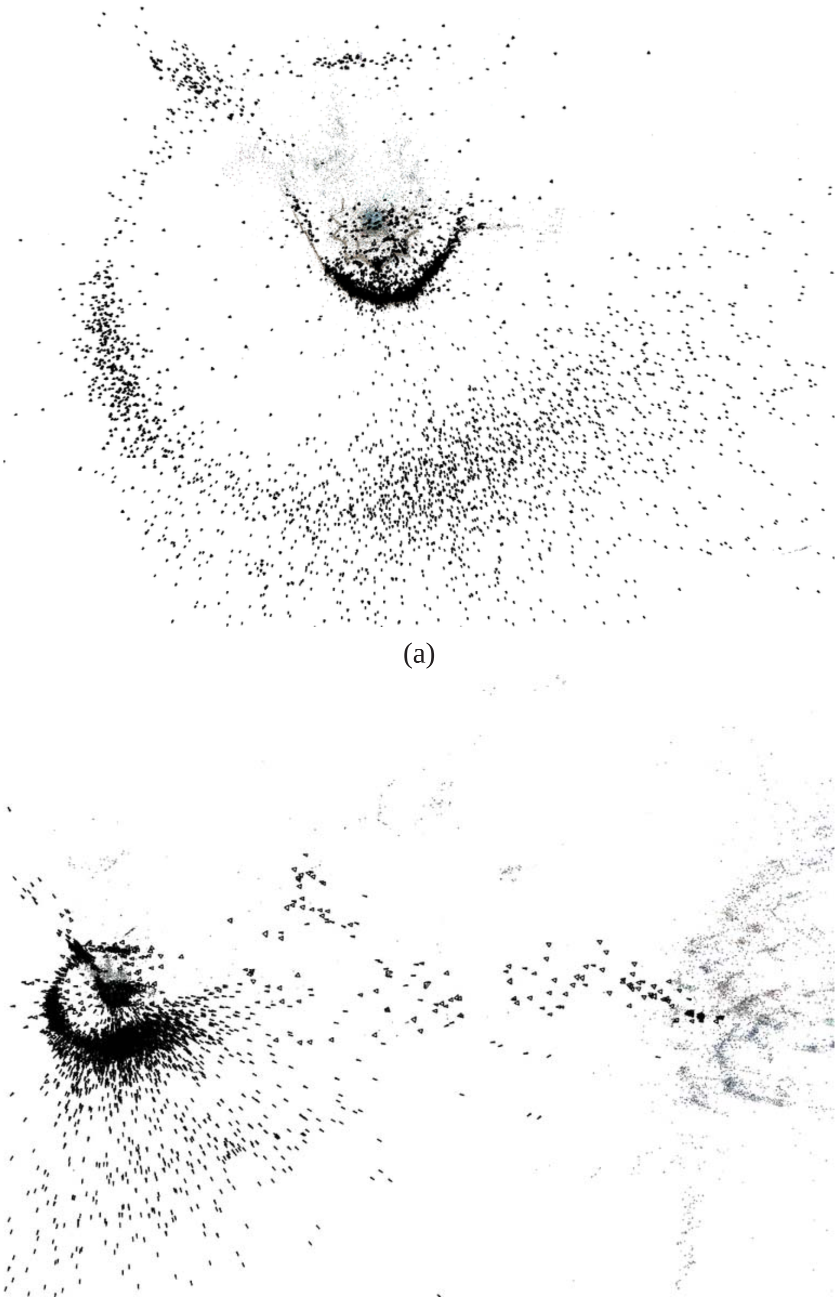


Figure 4.11: Views of the Trafalgar Square reconstruction, with images for comparison.



(a)

(b)

Figure 4.12: *Views of the Statue of Liberty reconstruction.* (a) An overhead view of the reconstruction centered on the Statue. The point cloud itself is in the very center of the image, and the small black triangles around the point cloud represent recovered cameras. Many photos were taken either on Liberty Island itself or from boats out in the water. (b) Overhead view of the full reconstruction, including parts of Manhattan on the right.

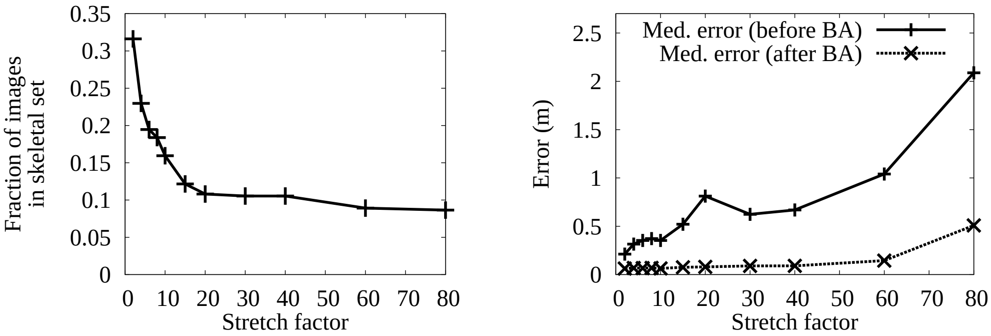


Figure 4.13: *Stretch factor analysis for St. Peter's.* Left: stretch factor vs. number of nodes in the skeletal set. Right: median error (in meters) in camera position for reconstructions before and after final bundle adjustment. As the stretch factor increases, the error before bundling increases, but applying bundle adjustment results in a low error level (around 6-8cm; note that the nave of the cathedral is about 24m across), even for stretch factors as large as 30.

Internet sets, due to the more regular distribution of photos.

For most of the data sets, the algorithm spent more time in reconstruction than in building the skeletal graph; for a few particularly dense sets (e.g., the Pantheon), the preprocessing took more time (for the Statue dataset, the pairwise reconstructions took over 90% of the reconstruction time). This was also true for the **Temple** data set, the only set for which the baseline method outperformed the skeletal set algorithm. Because of the high density of images, there are a large number of overlapping pairs, yet few pairs that match my definition of *duplicate*. Hence, a large amount of time (approximately 55 minutes) was spent creating pairwise reconstructions. In contrast, once the skeletal set was computed, the full model was reconstructed in just 9 minutes, compared to 64 minutes for the baseline reconstruction. The baseline reconstruction still took less time, however, due to the length of time taken in computing pairwise reconstructions. There is significant opportunity to speed up this step, for instance, through parallelization (each pairwise reconstruction can be computed independently), or through a more efficient implementation of RANSAC.

Next, I analyze the tradeoff between the stretch factor t and the accuracy of the reconstruction. To evaluate this tradeoff, I took one of the data sets (St. Peter's) and reconstructed it with multiple values of t. There is no ground truth reconstruction for this data set, so I used the reconstruction obtained from running the baseline method on the full image set as a standard by to measure ac-

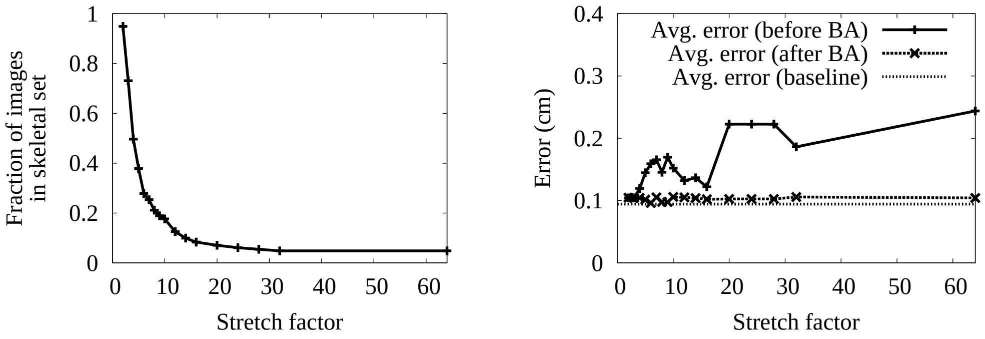


Figure 4.14: *Stretch factor analysis for the* **Temple** *data set.* The graph on the left plots the number of images in the skeletal set for several values of the stretch factor t. The graph on the right plots the average distance between the reconstructed and ground truth camera centers for these values of t, showing average error (in cm) both before and after a final bundle adjustment. Note that the error before bundle adjustment, while noisy, tends to increase with the stretch factor, but the error after bundle adjustment stays roughly constant. I also ran SfM with the full set of input images, and plot the error as a baseline. This data set is very controlled and regularly sampled compared to the Internet collections, and the behavior as t changes is somewhat different than the St. Peter's collection. For instance, even for the largest values of t we tried, the initialization provided by the skeletal reconstruction was close enough to the correct solution for the final bundle to pull it to the right place.

curacy. (This baseline reconstruction, which uses all available information, should in theory have the highest accuracy.) For each value of t, I first aligned the resulting reconstruction to the baseline reconstruction by finding a similarity transform between corresponding points. I then computed the distances between corresponding cameras, both before and after a final bundle adjustment. Figure 4.13 shows the results, plotting the size of the skeletal set, and the median error in camera position, for several values of t. As t increases, the size of the skeletal set decreases, and the error before the final bundle adjustment increases. However, applying the final bundle results in a low, relatively constant error level (in this case, a median error between 6-8cm for a building about 24m in width), even for stretch factors as large as 30, at which point only 10% of the images are used in the skeletal set. For even larger stretch factors, however, the bundled solution begins to degrade, because the initialization from the skeletal set is no longer good enough to converge to the correct solution.

I then ran the same experiment on the **Temple** dataset (this time comparing to the known ground truth, rather than the results of the baseline method). Figure 4.14 shows the results. The error before the final bundle adjustment is noisy, but tends to generally increase with the stretch factor; the error after bundle adjustment stays roughly constant, even for the largest stretch factors I tried. This data set is very controlled and regularly sampled compared to the Internet collections, and the behavior as t changes is somewhat different than the St. Peter's collection. For instance, even for the largest values of t I tried, the initialization provided by the skeletal reconstruction was close enough to the correct solution for the final bundle to pull it to the right place.

4.5 Conclusion

I have presented an algorithm for reconstructing Internet photo collections by computing a skeletal graph, and shown that this method can improve efficiency by up to an order of magnitude, with little or no loss in accuracy. There are also many interesting avenues for future work. First, my experiments indicate that a bottleneck in the algorithm is the computation of the two-image reconstructions and covariances for each pair of matching images. This step could easily be made much faster through paralellization, but are also other potential ways to improve efficiency. For instance, if a lower bound on, or an approximation to, the covariance for a pair of images could be computed quickly, it might be possible to compute the skeletal set and the pairwise reconstructions at the same time, only computing exact edge weights when required. An online approach to creating the skeletal set would also be interesting in cases where new images are coming in all the time (as is the case with the Internet), and should be incorporated into existing reconstructions. Many of these new images would likely contain redundant information, and need not be added to the skeletal set, but others may be desirable to strengthen or extend the current skeletal set.

It would also be interesting to create a more sophisticated model of uncertainty, e.g, taking uncertainty in camera orientation, and perhaps in scene structure, into account, or by considering multiple paths between image pairs. It could also be fruitful to work with triples, rather than pairs, for convenience in representing connectivity and improved robustness.

Part II

NAVIGATION AND VISUALIZATION OF 3D PHOTO COLLECTIONS

In Part I of this thesis, I described my approach to recovering the geometry of large, unstructured photo collections. This capability can enable a host of applications, e.g., taking 3D measurements of objects, studying the behavior of photographers, and analyzing images for use in forensics. However, one of the most exciting applications, and the focus of Part II, is to automatically recreate the experience of the world's famous places from photos on the Internet and to build new 3D interfaces for immersively exploring and interacting with our world.

What should such interfaces look like? The answer is not immediately clear, because these reconstructions really combine two intertwined artifacts—photos and scenes—each of which are interesting to explore in isolation. In a photo collection, each individual photo was carefully composed and captured to tell a story or record a moment of some sentimental or aesthetic importance to the photographer. The photos themselves are therefore worthy of examination, not only to the photographer, but potentially to their larger circle of friends and family or to the world as a whole. Photo-sharing sites like Flickr and SmugMug have built up communities of people who share photos with one another and view, comment on, and tag each others photos, validating the idea that photographs can be artifacts of value to many people, even complete strangers. The second artifact is the underlying scene that is reconstructed. Scenes—examples of which include the Taj Mahal, the Pantheon in Rome, Yosemite National Park, and one's own home—are also interesting and can be meaningful for aesthetic, historical, and personal reasons. Thus, within these reconstructed collections, there are two interrelated tasks to consider:

1. **Photo browsing**: getting an overview of a photo collection, finding specific images, and viewing and sharing progressions of photos (telling stories).
2. **Scene exploration and understanding**: building a mental map of a place, finding specific objects, rooms, or views, and learning the names of and facts about particular parts of a scene.

In my work, I have created interfaces that address both of these tasks together. For each of these tasks, I argue that there are elements of the other that can be beneficial. For photo browsing, placing the user inside of a 3D scene to immersively interact with a photo collection can aid in understanding the physical relationships between different photos and allow for the creation of new types of *geometric* search controls that help find photos of specific objects or viewpoints. I also believe that putting photos together in a single, consistent 3D world can help strengthen photo communities like Flickr, by placing and relating the community's photos together in a way that makes it easier see how they fit together. On the other hand, for the task of scene exploration, the very fact that photos are meant to be *interesting* means that the statistics of large photo collections, taken by many different people, can reveal a great deal of information about the underlying scene; for instance, what parts of the scene seem important, how people tend to move around in the scene, and the locations of individual objects.

I explore these issues in the next two chapters, which describe two new systems for exploring 3D photo collections. In Chapter 5, I describe the Photo Tourism system, an immersive photo browser that uses 3D information to provide new geometric controls for exploring and annotating large photo collections. In Chapter 6, I describe Pathfinder, a scene exploration tool that uses large photo collections for rendering, and which automatically discovers interesting, scene-specific navigation controls by analyzing the distribution of views in the input collections. While these two systems look at two sides of a coin, they are very much related—they both take the same input, and neither one is purely about photo browsing, or purely about scene understanding. Rather, both examine the ways in which these two tasks complement each other.

Two critical components of both systems are *rendering* (how to effectively display the scene and create good transitions between images) and *navigation* (how to browse the photos and explore the scene). For the problem of rendering, the sparse points produced by SfM methods are by themselves very limited and not always effective for rendering a scene. Nonetheless, I demonstrate that camera information and sparse geometry, along with new morphing, image stabilization, and non-photorealistic proxy rendering techniques, are sufficient to provide compelling scene renderings and view interpolations. Likewise, the unstructured nature of the photo collections can make it difficult to provide the free-viewpoint navigation controls found in games and 3D viewers, or even the simplified 3D controls in applications like Google Street View. For both systems, I describe new techniques for creating simple, easily-understood controls for unstructured collections.

Chapter 5

PHOTO TOURISM: AN IMMERSIVE 3D PHOTO BROWSER

How can we build a better photo browser for collections of photographs for which we know exactly where each photo was taken, and in which direction it was looking, and have a sparse 3D model of the scene? The reconstruction algorithms of Part I of this thesis present us with this opportunity, and this chapter presents a 3D photo browser called *Photo Tourism* that addresses this question.

In many ways Photo Tourism is primarily a photo browser. It has tools for searching for images, playing slideshows, and tagging images, and the emphasis of the system on moving the user from one photo to another. However, it combines these common photo browsing features with several elements of 3D navigation. First, the primary view is a window into a virtual world where the user is moving around in 3D among the reconstructed photos. Second, the search controls include new *geometric* and *object-based* controls, which make it easier to find photos of particular views or objects. Finally, the reconstructed 3D information is used to produce attractive 3D transitions between images, which highlight the spatial relationships between photos. Such transitions can be much more powerful and evocative than the cross-dissolves used in traditional photo browsers.

In addition to navigation and rendering, *semantics* plays an important role in photo browsing. It is often useful or interesting to learn more about the content of an image, e.g., “which statue is this?” or “when was this building constructed?”—things someone familiar with the content of the photos might be able to tell us. A great deal of annotated imagery of this form already exists in guidebooks, maps, and Internet resources such as Wikipedia [158] and Flickr. However, the images in your own personal collection, a random photo on the web, or the image you may be viewing at any particular time (e.g., on your cell phone camera) may not have such annotations, and a tour guide may not be readily available. A very powerful feature of Photo Tourism is the ability to transfer annotations automatically between images, so that information about an object in one image can be propagated to all other images that contain that same object.

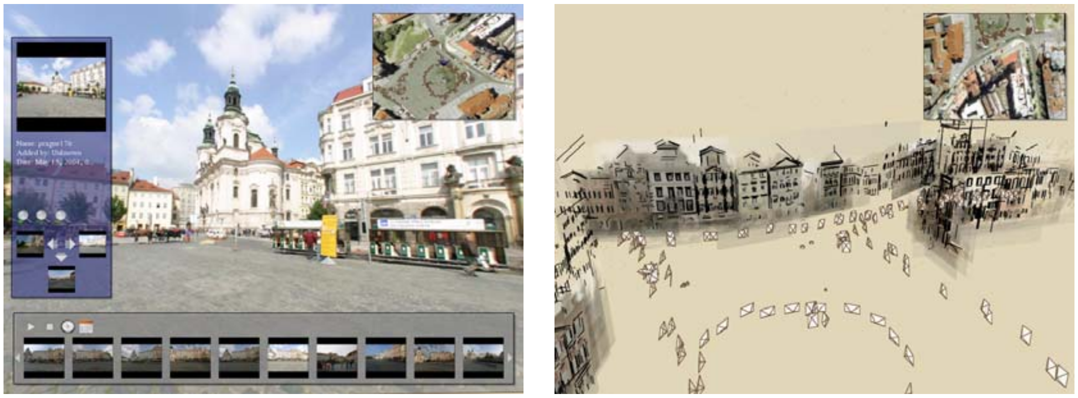
System overview. The input to Photo Tourism is the scene and camera geometry recovered from the structure from motion (SfM) algorithms described in Chapters 3 and 4. Recall from Chapter 3 that this geometry consists of:

1. A set of cameras, $C = \{C_1, C_2, \ldots, C_k\}$, each with a corresponding image image I_j and a set of viewing parameters $\mathbf{C}_j$.
2. A set of 3D points $P = \{p_1, p_2, \ldots, p_n\}$, each with a color and a position $\mathbf{X}_i$.
3. A set of line segments $L = \{l_1, l_2, \ldots, l_m\}$.
4. A 3D plane, $\mathrm{Plane}(C_i)$, for each camera C_i, fit to the 3D points seen by that camera.
5. A 3D plane, $\mathrm{CommonPlane}(C_i, C_j)$, for each pair of overlapping cameras C_i and C_j.

Photo Tourism consists of three main components. The first is the rendering engine. While the user is often looking at a single photo, as in any other photo browsing tool, the user really exists in a 3D world and can view the scene from any angle. In addition, when the system transitions from one image to another, it physically moves users between the two, as if they were flying between the images in 3D. In addition to camera motion, these transitions use a specialized rendering algorithm for morphing between pairs of photographs. Users are not strictly confined to the photos, however, and can also choose to move freely through the scene, which is rendered using the recovered points and line segments. These rendering techniques are described in Section 5.1.

The second component of Photo Tourism consists of the navigation interface for exploring and searching for photos. There are four kinds of controls: *zooming* controls for finding details, similar views, or broader perspectives of a given photo; *directional* controls for finding photos that show more in a particular direction; *object-based* controls for finding views of a specific object; and *registered slideshows* for viewing a sequence of related images. These navigation controls are described in Section 5.2.

The final component of Photo Tourism consists of tools for scene annotation and enhancement. A user can label an object in one image and have that label propagate to all other relevant images. In addition, a user can also augment a scene with additional photos, which are automatically registered to the reconstructed scene. These capabilities are described in Section 5.3. Finally, Section 5.4 (and especially the companion video on the project webpage [108]) presents results, and Section 5.5 concludes with a discussion of the system.



Name: prague176
Added by: Unknown
Date: May 15, 2004, 0

(a)

(b)

Figure 5.1: *Screenshots from the Photo Tourism viewer.* Left: when the user visits a photo, that photo appears at full-resolution, and information about it appears in a pane on the left. Results of search operations appear in the thumbnail pane on the bottom of the screen, and an overhead map is shown on the upper-right side of the screen. Right: a view looking down on the **Prague** dataset, rendered in a non-photorealistic style.

5.1 Rendering scenes and transitions in Photo Tourism

This chapter describes the visual user interface and rendering components of the Photo Tourism system. Figure 5.1 (left-hand side) shows a screenshot from the user interface. The elements of this window are the main view into the scene, which fills the window, and three overlay panes: an information and search pane on the left, a thumbnail pane along the bottom, and a map pane in the upper-right corner.

The main view shows the world as seen from a virtual camera controlled by the user. This view is not meant to show a photo-realistic rendering of the scene (unless the user is situated directly at a photo), but rather to display photographs in spatial context and give a sense of the geometry of the true scene.

The information pane appears when the user visits a photograph. This pane displays information about that photo, including its name, the name of the photographer, and the date and time when it was taken. In addition, this pane provides access to geometric controls for searching for other photographs with certain relations to the current photo.

The thumbnail pane shows the results of search operations as a filmstrip of thumbnails. When the user mouses over a thumbnail, the corresponding image I_j is projected onto $\mathrm{Plane}(C_j)$ to show the content of that image and how it is situated in space. The thumbnail panel also has controls for sorting the current thumbnails by date and time and viewing them as a slideshow.

Finally, when a reconstruction has been aligned to a map, the map pane displays an overhead view of the scene. The map tracks users' position and heading as they move through the scene.

5.1.1 Rendering the scene

While the main emphasis in Photo Tourism is directing users towards photos, the system also can render the scene using the point, line, and plane geometry recovered from SfM, as shown in the right-hand side of Figure 5.1. This scene is rendered by drawing the reconstructed points and lines. In addition, the interface supports a non-photorealistic rendering mode that provides more attractive visualizations. This mode creates a washed-out rendering designed to give an impression of scene appearance and geometry, but is abstract enough to be forgiving of the lack of detailed geometry. To generate the rendering, the system projects a blurred, semi-transparent version of each image I_j onto $\mathrm{Plane}(C_j)$, using alpha blending to combine the projected images together to create a watercolor effect.

The cameras themselves are rendered as small wire-frame pyramids with a solid back face. If the user is visiting a camera, the back face of that camera pyramid is texture-mapped with an opaque, full-resolution version of the photograph, so that the user can see it in detail, as shown in the left-hand image of Figure 5.1. The back faces of other cameras are texture-mapped with a low-resolution, semi-transparent thumbnail of the corresponding photo.

5.1.2 Transitions between photographs

Another important component of Photo Tourism is the ability to render transitions between photos. While traditional 2D photo browsers typically use simple cuts or cross-fades to transition between two photos, the accurate geometric information provided by SfM allows Photo Tourism to use 3D camera motion and view interpolation techniques to make transitions more visually compelling and to emphasize the spatial relationships between photographs.

Camera motion. During a transitions between two cameras C_{start} and C_{end}, the system moves the virtual camera along a path between the endpoints (linearly intepolating the locations and rotations, represented using quaternions). The field of view of the virtual camera is also interpolated so that when the camera reaches its destination, the destination image will fill as much of the screen as possible, while still remaining wholly within the field of view. The timing of the camera motion is non-uniform, easing in and out of the transition to maintain second-order continuity [83].

If the camera moves as the result of an object selection (as described in Section 5.2.3), the transition is slightly different, and tries to keep the object of interest centered in the field of view. Before it begins moving, the virtual camera is first oriented to point at the mean of the selected points. As the position of the camera is then interpolated towards the destination image, the camera orientation is updated so as to keep the selected object fixed in the center of the view. The final orientation and focal length are computed so that the selected object is centered and fills the screen.

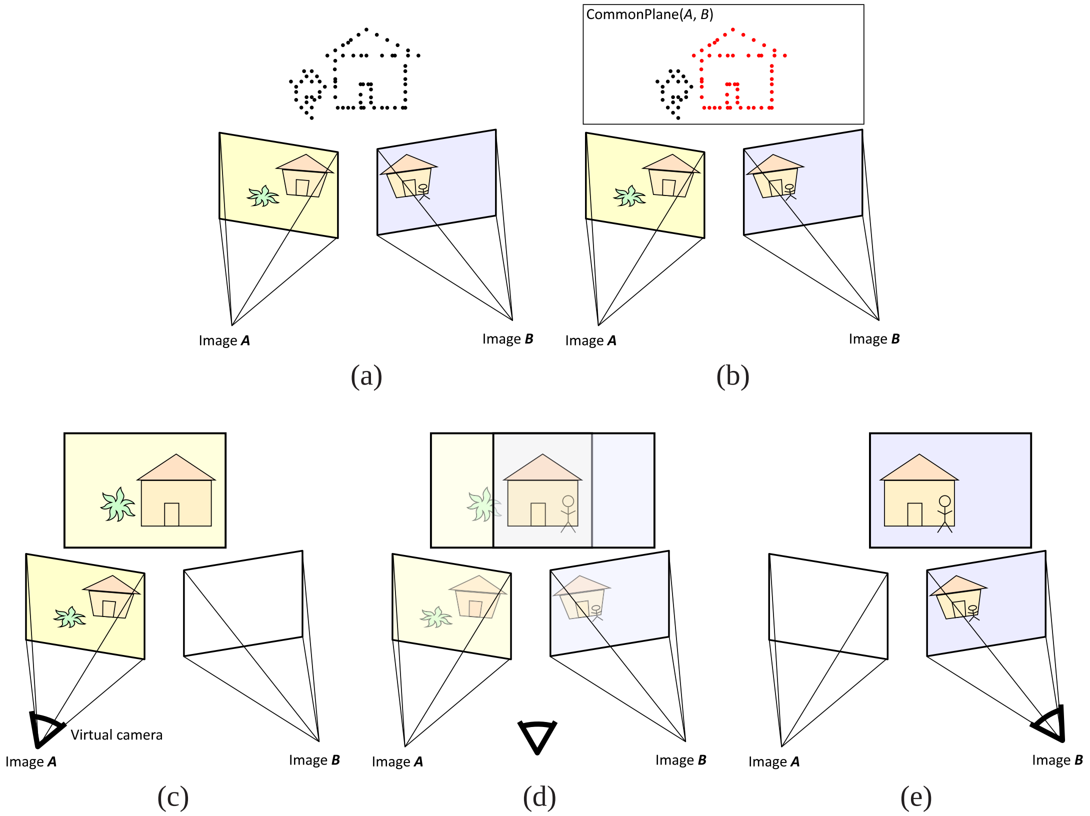
View interpolation. While the camera is moving during a transition, the system also interpolates or morphs between the two photos to create in-between views; this interpolation smooths the visual transition between the two images. To do the interpolation, I first compute *projection surfaces* onto which the images will be projected during the transition. Then, as the camera moves from C_{start} to C_{end}, image I_{start} is slowly faded out as I_{end} is faded in, both projected onto their respective projection surface. One way to imagine the process is to think of the cameras C_{start} and C_{end} as two slide projectors, each projecting onto a screen placed in the scene. During the camera motion, the first projector is slowly turned off, as the second is turned on. If the projection surfaces are close to the actual scene geometry, the effect of this transition is to visually align the two images as the cross-fade occurs. Combined with the camera motion, this alignment can create a strong sense of both the physical and visual relationships between the two views. The effect can also be quite visually striking, especially when morphing between photos with very different appearances, such as daytime and nighttime.

Photo Tourism supports two types of projection surfaces: (1) planes and (2) polygonal meshes created from triangulating the point cloud. When planar projection surfaces are used, a single common plane is computed for the two images used in the transition. When polygonal meshes, two meshes are used for each transition, one for each image. Figure 5.2 illustrates how transitions work when planar projection surfaces are used.

Planar projection surfaces. In this mode, $\mathrm{CommonPlane}(C_{\mathrm{start}}, C_{\mathrm{end}})$ is used as a common projection surface for the two images. Though a plane is a very simple geometric object and this method tends to stabilize only a dominant plane in the scene, this technique can still produce compelling transitions, particularly for urban scenes, which are often composed of planar surfaces.

Polygonal meshes. In this mode, a triangulated mesh is created for each of the two images involved in the transition. To create the meshes, I first compute a 2D Delaunay triangulation for image I_{start} from the set of 2D projections of $\mathrm{Points}(C_{\mathrm{start}})$ into I_{start}. The projections of $\mathrm{Lines}(C_{\mathrm{start}})$ into I_{start} are also imposed as edge constraints on the triangulation [22]. The resulting constrained Delaunay triangulation may not cover the entire image, so I overlay a grid onto the image and add to the triangulation each grid point not contained inside the original triangulation. Each added grid point is associated with a 3D point on $\mathrm{Plane}(C_{\mathrm{start}})$. Thus, we now have a set of triangulated 2D points that covers the image plane, and a corresponding 3D point for each 2D point. An example triangulation is shown in Figure 5.4. The 2D triangulation is then used to create a 3D mesh, by triangulating the 3D points using the same connectivity as their 2D counterparts. This mesh is used as the projection surface for I_{start}. I compute a second projection surface for I_{end} in an analogous way.

During the transition, each of the two images is projected onto its own projection surface. The depth buffer is turned off to avoid popping due to intersections between the two projection surfaces. While the meshes are fairly coarse and are not a completely accurate representation of the scene geometry, they are often suitable as projection surfaces and sufficient to give a sense of the 3D geometry of the scene. For instance, this approach works well for many transitions in the **Great Wall** data set. However, missing geometry, spurious points, and the projection of people and other foreground elements onto the mesh can often cause distracting artifacts. In my experience, the artifacts resulting from planar projection surfaces are usually less objectionable than those resulting from meshes, perhaps because we are accustomed to the distortions caused by viewing planes from different angles from our experiences in everyday life. Thus, planar impostors are used as the default projection surfaces for transitions. Example morphs using both techniques are shown in the video



(a) (b)

(c) (d) (e)

Figure 5.2: *Transitions between images.* This figure illustrates how Photo Tourism renders transitions between images (using a planar projection surface). (a) Given two registered images A and B and the recovered point cloud, a common plane is computed for the image pair by (b) finding the points visible to both images (colored red), and fitting a plane, denoted $\mathrm{CommonPlane}(A, B)$ to these points. For architectural scenes, such planes tends to coincide with walls of buildings, which are often nearly planar. Parts (c)-(e) show three moments in time during the transition from A to B. The system starts with the virtual camera at the location of image A (c), and treats A and B as slide projectors projecting onto $\mathrm{CommonPlane}(A, B)$. Initially, projector A is turned completely on, and projector B is completely off. The system then moves the virtual camera from A to B, while turning off projector A and turning off projector B. (d) Midway through the transition, the virtual camera is halfway between A and B, and both projectors are projecting at 50% opacity, hence the projected image is a blend of A and B. (e) The end of the transition, with the virtual camera at B and projector B turned completely on. If the plane is close to the plane of the front wall of the house, that part of the scene will be stabilized during the transition.

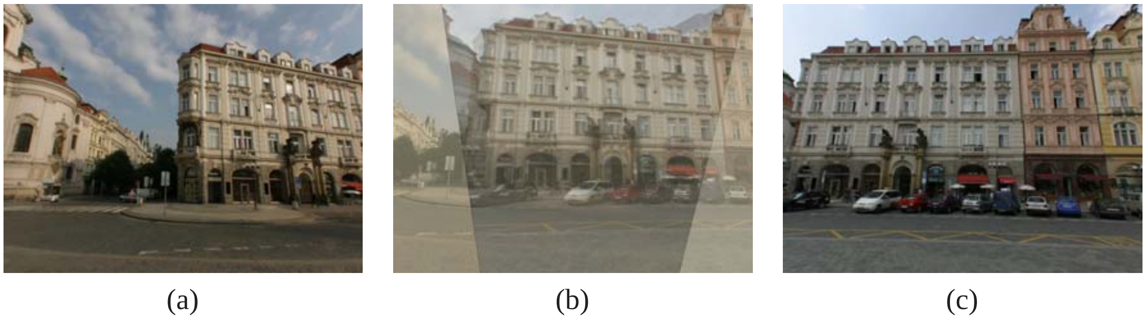


Figure 5.3: *Frames from an example image transition.* Images (a) and (c) are two photos from the **Prague** data set. Image (b) is the midpoint of a transition between (a) and (c), rendered using a planar projection surface. In this case, the building on the right in image (a) is stabilized.

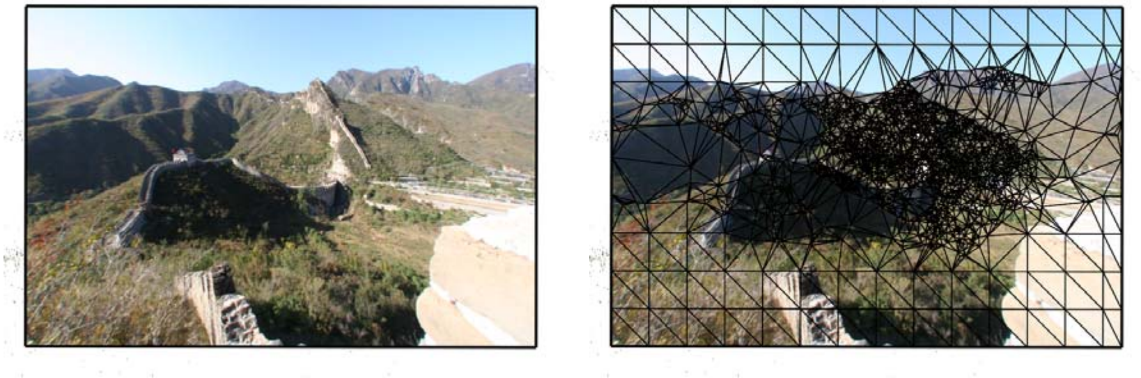


Figure 5.4: *Delaunay triangulation of an image.* Left: an image from the **Great Wall** data set. Right: Delaunay triangulation of the points seen by the image, augmented with points placed on a coarse grid over the image.

on the project website [108].

Special cases for planar projection surfaces. When planar projection surfaces are used, there are a few situations that must be handled as special cases. First, if C_{start} and C_{end} observe no common points, $\mathrm{CommonPlane}(C_{\mathrm{start}}, C_{\mathrm{end}})$ is undefined, and the system has no basis for creating a projection surface for the images. In this situation, the system reverts to rendering the transition using points, lines, and textured planes, as in Figure 5.1(b), rather than using the images.[1]

Second, if the normal to $\mathrm{CommonPlane}(C_{\mathrm{start}}, C_{\mathrm{end}})$ is nearly perpendicular to the viewing directions of C_{start} or C_{end}, the projected images undergo significant distortion during the morph. In this case, I instead use a plane which passes through the mean of $\mathrm{Points}(C_{\mathrm{start}}) \cap \mathrm{Points}(C_{\mathrm{end}})$, whose normal is the average of the viewing directions of the two cameras. Finally, if the vanishing line of $\mathrm{CommonPlane}(C_{\mathrm{start}}, C_{\mathrm{end}})$ is visible in either I_{start} or I_{end}, it is impossible to project the entirety of I_{start} or I_{end} onto the plane; imagine turning on a slide projector in an infinite room, and observing how much of the slide projects onto the floor—there is a problem because the horizon of the room (the vanishing line of the floor) is visible to the slide projector. In this case, I project as much as possible of I_{start} and I_{end} onto the plane, and project the rest onto the plane at infinity.

5.2 Navigation in Photo Tourism

Photo Tourism supports several different types of controls for navigating through the scene and finding interesting photographs. These include free-flight navigation, geometric and object-based search tools, and a stabilized slideshow viewer.

5.2.1 Free-flight navigation

The free-flight navigation controls include standard controls for 3D motion found in games and 3D viewers. The user can move the virtual camera forward, back, left, right, up, and down, and can control pan, tilt, and zoom. This allows the user to freely move around the scene and provides a simple way to find interesting viewpoints and nearby photographs.

At any time, the user can click on a camera pyramid in the main view, and the virtual camera

[1] Chapter 6 introduces a *path planning* algorithm that gets around this problem by using multi-image transitions.

will smoothly move until it is coincident with the selected camera. The virtual camera pans and zooms so that the selected image fills as much of the main view as possible.

5.2.2 Selecting related views

When visiting a photograph, the user has a snapshot of the world from a single point of view and a single instant in time. The user can pan and zoom to explore the photo, but may also want to see aspects of the scene beyond those captured in a single picture. He or she might wonder, for instance, what lies just outside the field of view, or to the left of the objects in the photo, or what the view looks like at a different time of day.

To make it easier to find such related views, the interface provides the user with a set of *geometric* browsing controls. These controls find photos with certain spatial relationships, and fall into two categories: zooming controls for selecting the scale at which to view the scene, and directional controls for seeing more in a particular direction (e.g., left or right). Icons associated with these controls appear in two rows in the information pane that appears when the user is visiting a photo.

Zooming controls. Photo Tourism supports three zooming controls: (1) finding *details*, or higher-resolution close-ups, of the current photo, (2) finding *similar* photos, and (3) finding *zoom-outs*, or photos that show more surrounding context.

Let C_{curr} be the photo the user is currently visiting. How can we identify photos that depict the current view at different scales? One approach would be to find photos taken from a similar location, but with different zoom settings. However, the intent of these controls are to show the *content* of the image at different levels of detail, rather than to keep the view itself absolutely fixed. For instance, an image taken twenty meters backwards from C_{curr} with the same zoom setting may be just as good a zoom-out as an image taken at the same location as C_{curr} with a wider field of view. Similarly, a detail of part of the scene may have been taken at a slightly different angle from C_{curr}, in order to capture a more representative or visually pleasing close-up of an object. Therefore, the zooming relations are based on the content of the photos rather than their absolute positions and orientations.

To compute these relations, I first define the notion of *apparent size*. Roughly speaking, the apparent size of an object in an image is the proportion of the image occupied by that object. For instance, a given object might take up much more of an image I_j than an image I_k, and thus have a larger apparent size in I_j. Such an image could reasonably be called a *detail* of I_k.

As the scene is not really divided up into specific objects, I simply use the visible point set $\mathrm{Points}(C)$ as the "object" visible to camera C and estimate the apparent size of the point set in an image by projecting it into that image and computing the area occupied by the projections. Specifically, to estimate the apparent size of a point set P in a camera C, I project the points into C, compute the bounding box of the projections that land inside the image, and calculate the ratio of the area of the bounding box (in pixels) to the area of the image. I refer to this quantity as $\mathrm{ApparentSize}(P, C)$.

Clicking on one of the scaling controls activates a search for images with the selected relation (detail, similar, or zoom-out). The images returned from this search are selected from among images with sufficient overlap with C_{curr} (I use images which have at least three points in common with C_{curr}). I classify each such camera C_j as:

- a *detail* of C_{curr} if

$$\mathrm{ApparentSize}(\mathrm{Points}(C_j), C_{\mathrm{curr}}) < 0.75 \cdot \mathrm{ApparentSize}(\mathrm{Points}(C_j), C_j), \qquad (5.1)$$

  i.e., $\mathrm{Points}(C_j)$ appears at least 25% larger in C_j than in C_{curr}. In addition, 95% of the points in $\mathrm{Points}(C_j)$ must project inside the field of view of C_{curr}.

- *similar* to C_{curr} if

$$0.75 < \frac{\mathrm{ApparentSize}(\mathrm{Points}(C_{\mathrm{curr}}), C_j)}{\mathrm{ApparentSize}(\mathrm{Points}(C_{\mathrm{curr}}), C_{\mathrm{curr}})} < 1.3 \qquad (5.2)$$

  and the angle between the viewing directions of C_{curr} and C_j is less than a threshold of 10°

- a *zoom-out* of C_{curr} if C_{curr} is a detail of C_j.

The results of these searches are displayed in the thumbnail strip (sorted by decreasing apparent size, in the case of details and zoom-outs, i.e., the most zoomed-in view is first). These controls are useful for viewing the scene in more detail, comparing similar views of an object which differ

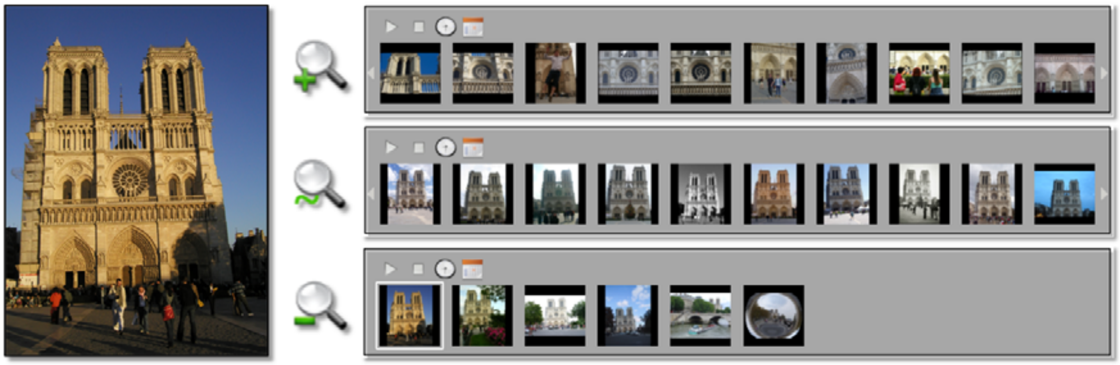


Figure 5.5: *Zooming controls.* Left: an image from the **Notre Dame** data set. Right: the results of searching for details (top), similar images (middle), and zoom-outs (bottom), starting from this input image.

in other respects, such as time of day, season, and year, and for "stepping back" to see more of the scene. Results returned from these zooming controls are shown in Figure 5.2.2.

Directional controls. The directional tools give the user a simple way to "step" left or right, i.e., to see more of the scene in a particular direction. For each camera, I compute a left and right neighbor and link these neighbors to arrows displayed in the information pane. To find the left and right neighbors of a camera C_j, I look for images in which the objects visible in C_j appear to have moved right or left, respectively.[2] In particular, I compute the average 2D motion m_{jk} of the projections of $\mathrm{Points}(C_j)$ from image I_j to each neighboring image I_k:

$$m_{jk} = \frac{1}{|\mathrm{Points}(C_j)|} \sum_{\mathbf{X} \in \mathrm{Points}(C_j)} \left(P(C_k, \mathbf{X}) - P(C_j, \mathbf{X})\right) \tag{5.3}$$

where $P(C, \mathbf{x})$ is the projection equation defined in Chapter 3. If the angle between m_{jk} and the desired direction (i.e., left $= (-1, 0)^T$ or right $= (1, 0)^T$) is less than $15°$, and the apparent size of $\mathrm{Points}(C_j)$ in both images is similar, C_k is a candidate left (or right) neighbor to C_j. Out of all the candidates (if there are any), the algorithm selects the left or right neighbor to be the camera C_k whose motion magnitude $||m_{jk}||$ is closest to 20% of the width of image I_j.

[2] Facing an object and stepping to the right induces a *leftward* motion of the object in the field of view.

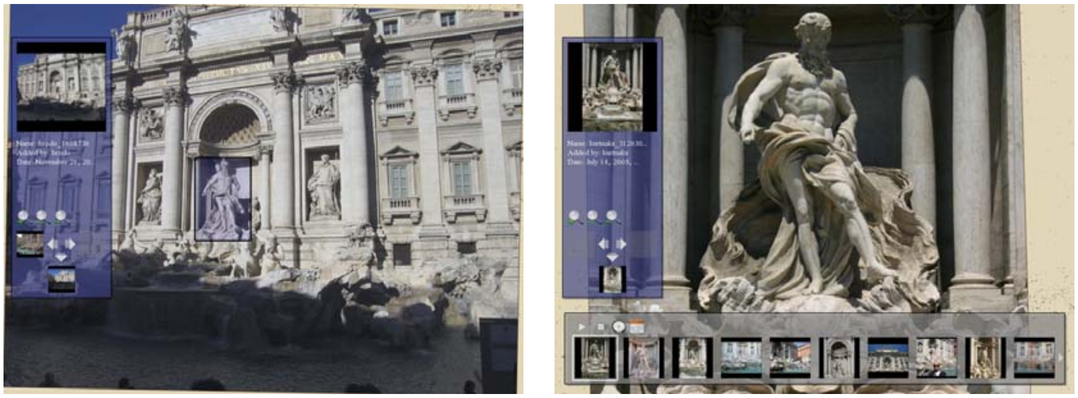


Figure 5.6: *Object-based navigation.* Left: the user drags a rectangle around the statue in the current photo. Right: the system finds a new, high-resolution photograph of the selected object, and moves to that view. Additional images of the object are shown in the thumbnail pane at the bottom of the screen.

5.2.3 Object-based navigation

Another type of search query supported by the viewer is "*show me photos of this object*," where the object in question can be directly selected in a photograph or the point cloud. This type of search, similar to that applied to video in the Video Google system [133], is complementary to, and has certain advantages over, other types of queries such as keyword search for finding images of a specific object. The ability to select an object is especially useful when exploring a scene—when the user comes across an interesting object, direct selection is an intuitive way to find a better picture of that object, and does not require knowing the name of that object.

The user selects an object by dragging a 2D box around a region of the current photo or the point cloud. All points whose projections lie inside the box form the set of selected points, S. The system then searches for the "best" photo of S by scoring each image in the database based on how well it is estimated to depict S. The top scoring photo is chosen as the representative view, and the virtual camera is moved to that image. Other images with scores above a threshold are displayed in the thumbnail strip, sorted by descending score. An example object selection interaction is shown in Figure 5.6.

My function for scoring images is based on three criteria: (1) whether most of the points in S are visible in an image, (2) the angle from which the points in S are viewed (frontal views are preferred), and (3) the image resolution. For each image I_j, I compute the score as a weighted sum of three terms, E_{visible}, E_{angle}, and E_{detail}.

To compute the visibility term E_{visible}, I first check whether $S \cap \text{Points}(C_j)$ is empty. If so, the object is deemed not to be visible to C_j at all, and $E_{\text{visible}} = -\infty$. Otherwise, $E_{\text{visible}} = \frac{n_{\text{inside}}}{|S|}$, where n_{inside} denotes the number of points in S that project inside the boundary of image I_j.

The term E_{angle} is used to favor head-on views of a set of points over oblique views. To compute E_{angle}, I first fit a plane to the points in S using a RANSAC procedure. If the percentage of points in S which are inliers to the recovered plane is above a threshold of 50% (i.e., there appears to be a dominant plane in the selection), I favor cameras that view the object head-on by setting $E_{\text{angle}} = V(C_j) \cdot \hat{n}$, where V indicates viewing direction, and $\hat{n}$ the normal to the recovered plane. If fewer than 50% of the points fit the plane, then $E_{\text{angle}} = 0$.

Finally, E_{detail} favors high-resolution views of the object. E_{detail} is defined to be the area, in pixels, of the bounding box of the projections of S into image I_j (considering only points that project inside the boundary of I_j). E_{detail} is normalized by the area of the largest such bounding box, so the highest resolution available view will have a score of 1.0.

Once the three scores E_{visible}, E_{angle}, and E_{detail} have been computed for an image, the final score is computed as a weighted combination of the three: $E = E_{\text{visible}} + \alpha E_{\text{angle}} + \beta E_{\text{detail}}$. In my system, I set α to 1 and β to 3. The image that achieves the highest score is selected as the next image, and a transition to the image is played.

The set S can sometimes contain points that the user did not intend to select, especially occluded points that happen to project inside the selection rectangle. These extra points can belong to completely unintended objects, and can therefore cause unexpected images to be returned from an object selection search. If the system had complete knowledge of visibility, it could cull such hidden points. However, the fact that the system uses only sparse geometry means that occlusions cannot be computed without additional assumptions, as infinitesimal points will occlude each other with zero probability. Therefore, I use a set of heuristics to prune the selection. If the selection was made while visiting an image I_j, I can use the points that are known to be visible from that viewpoint (i.e., $\text{Points}(C_j)$) to refine the selection. In particular, I compute the 3×3 covariance matrix for the points in $S' = S \cap \text{Points}(C_j)$, and remove from S all points with a Mahalanobis distance greater than 1.2 from the centroid of S'. If the selection was made while not visiting an image, I instead

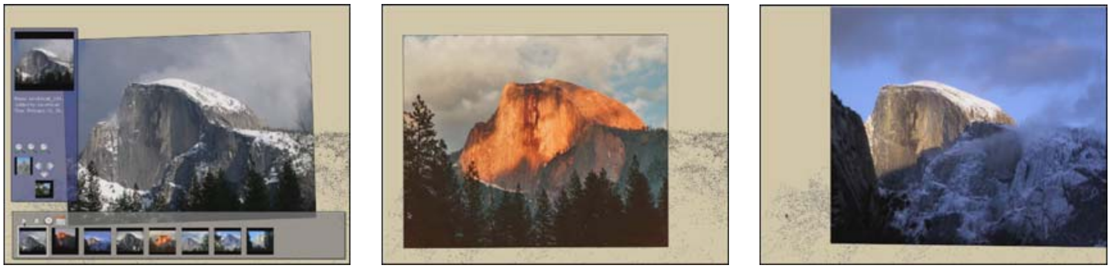


Figure 5.7: *A stabilized slideshow.* The user first finds a photo they like (left-most image), then finds similar views using the toolbar. The next two images are part of a stabilized slideshow, where Half Dome is in the same place in each view.

compute a weighted centroid and covariance matrix for the entire set S. The weighting favors points that are closer to the virtual camera, since these are more likely to be unoccluded than points that are far away. Thus, the weight for each point is computed as the inverse of its distance from the virtual camera.

5.2.4 Creating stabilized slideshows

Whenever a search is activated, the resulting images appear in the thumbnail pane; when the thumbnail pane contains more than one image, its contents can be viewed as a slideshow by pressing a play button in the pane. The default slideshow behavior is that the virtual camera will move in 3D from camera to camera, pausing at each image for a few seconds before proceeding to the next. However, the user can also choose to “lock” the virtual camera, fixing it to its current position, orientation, and field of view. When the images in the thumbnail pane are all taken from approximately the same location, this locked camera mode stabilizes the images, making it easier to compare one image to the next. This mode is useful for studying changes in the appearance of a scene over time of day, seasons, years, weather patterns, etc. An example stabilized slideshow from the Yosemite data set is shown in Figure 5.7, and in the companion video [108].

5.3 Augmentation and enhancement of scenes

Photo Tourism also allows users to add content to a scene in two ways: a user can (1) register new photographs to an existing scene and (2) annotate regions of an image and have these annotations automatically propagate to other images.

5.3.1 Registering new photographs

New photographs can be registered on the fly to an existing reconstruction using two different techniques, one interactive, the other automatic. In the interactive technique, the user switches to a mode where an overhead map fills the view, opens a set of images, which are displayed in the thumbnail panel, and drags and drops each image onto its approximate location on the map. After each image has been placed on the map, the system estimates the location, orientation, and focal length of each new photo by running an abbreviated version of the SfM pipeline described in Chapter 3. First, SIFT keypoints are extracted from the image and matched to the keypoints of the twenty registered images closest to the specified initial location. The matches to each of the other images are then pruned to contain geometrically consistent matches, and the 3D points corresponding to the matched features in the existing registered images are identified. Finally, these matches are used to refine the pose of the new camera using bundle adjustment. Once a set of photos has been placed on the map, it generally takes around ten seconds to compute the final registration for each new camera on my test machine, a 3.80GHz Intel Pentium 4.

The second way to register a new image requires no initialization. The SIFT features in the new image are matched directly to the point cloud, and these matches are used for registration. As a pre-process, a representative SIFT feature descriptor is computed for each 3D point in the scene by averaging the feature descriptors associated with that point. When a new image is to be registered, its SIFT keypoints are extracted and matched to the representative feature descriptors. The resulting matches to 3D points are used to initialize the pose of the new camera by running a direct linear transform (DLT) procedure [68], and the pose is refined with a local bundle adjustment. Although this method has the advantage of not requiring initialization, it is slower for very complex scenes and tends to identify fewer matches than the first method, and can therefore be somewhat less reliable. An example registration of a new image to an existing scene (using this second method) is shown in Figure 5.8.

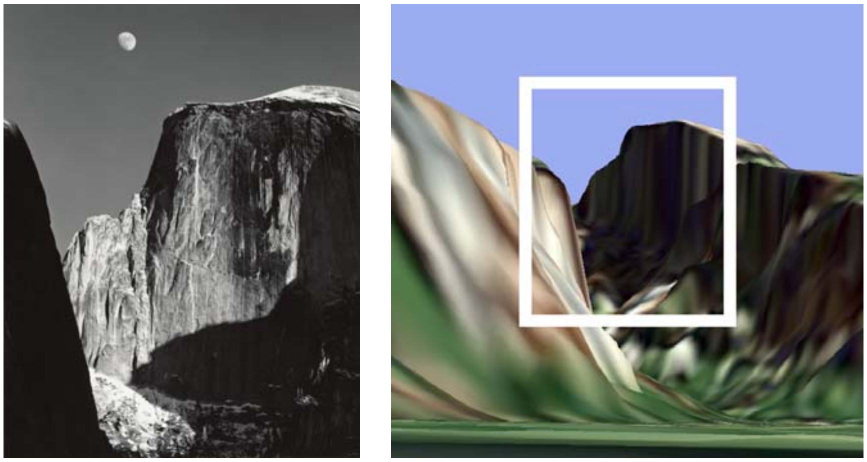


Figure 5.8: *A registered historical photo.* Left: *Moon and Half Dome*, 1960. Photograph by Ansel Adams. This historical photo taken by Ansel Adams was automatically registered to our Half Dome reconstruction. Right: rendering of DEM data for Half Dome from where Ansel Adams was standing, as estimated by the system. The white border was drawn manually for clarity. (DEM and color texture courtesy of the U.S. Geological Survey.)

5.3.2 Creating and transferring annotations

Annotations on image regions are an increasingly popular feature in photo organizing tools such as Flickr. A unique capability of Photo Tourism is that annotations can be automatically *transferred* from one image to all other images that contain the same scene region.

In the Photo Tourism interface, the user can select a region of an image and enter a label for that region. This annotation is then stored, along with the 3D points S_{ann} that lie in the selected area. When the user visits the photo, the annotation appears as a semi-transparent box around the selected points. Once annotated, an object can also be linked to other sources of information, such as web sites, guidebooks, and video and audio clips.

When an annotation is created, it is automatically transferred to all other photographs that see the annotated object. To transfer an annotation to another image I_j, the system first checks whether the annotation is visible in I_j, and whether it is at an appropriate scale for the image—that it neither fills too much of the image nor labels a very small region. To determine visibility, I simply test that at least one of the annotated points S_{ann} is in $\text{Points}(C_j)$. To check whether the annotation is at an appropriate scale, I compute the apparent size, $\text{ApparentSize}(S_{\text{ann}}, C_j)$, of the annotation in image

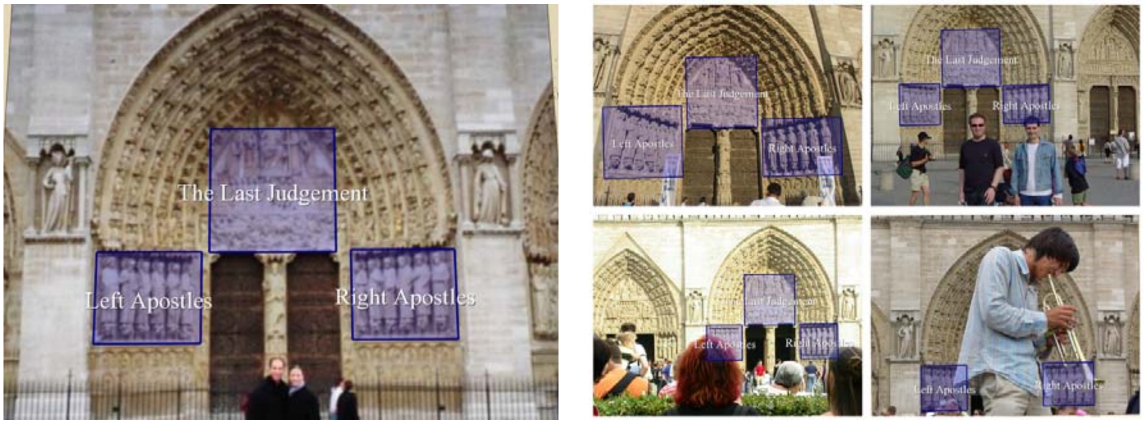


Figure 5.9: *Example of annotation transfer.* Three regions were annotated in the photograph on the left; the annotations were automatically transferred to the other photographs, a few of which are shown on the right. Photo Tourism can handle partial and full occlusions.

I_j; if the annotation is visible and

$$0.05 < \mathrm{ApparentSize}(S_{\mathrm{ann}}, C_j) < 0.8, \tag{5.4}$$

i.e., the annotation covers between 5 and 80% of the image, the system transfers the annotation to C_j. When the user visits C_j, the annotation is displayed as a box around the annotated points, as shown in Figure 5.9.

Besides quickly enhancing a scene with semantic information, the ability to transfer annotations has several applications. First, it enables a system in which a tourist can take a photo (e.g., from a camera phone that runs my software) and instantly see information about objects in the scene superimposed on the image. In combination with a head-mounted display, such a capability could offer a highly portable, computer-vision-based augmented reality system [44]. Second, it makes labeling photographs in preparation for keyword search much more efficient; many images can be labeled with a single annotation.

There are several existing sources of annotated imagery that can be leveraged by my system. Flickr, for instance, allows users to attach notes to rectangular regions of photos. Tools such as the ESP Game [155] and LabelMe [123] encourage users to label images on the web and have accumulated large databases of annotations. By registering such labeled images with an existing collection of photos using our system, Photo Tourism could transfer the existing labels to every other relevant photo in the system. Other images on the web are implicitly annotated: for instance, an image on a Wikipedia page is "annotated" with the URL and topic of that page. By registering such images to a reconstruction, I could automatically link other photos in that collection to the same page.

5.4 Results

I have evaluated the features of Photo Tourism on several photo collections. Because Photo Tourism is an interactive system, the results are best demonstrated through the video captures of interactive sessions found on the Photo Tourism project website [108]. Screenshots from these sessions are shown in figures throughout this chapter.

The first two collections were taken under relatively controlled settings (i.e., a single person with a single camera and lens): **Prague**, a set of 197 photographs of the Old Town Square in Prague, Czech Republic, taken over the course two days, and **Great Wall**, a set of 120 photographs taken along the Great Wall of China (82 of which were ultimately registered). The images in the **Prague** data set were taken by a photographer walking along the sides of the square, facing the buildings. A few more distant views of the buildings were also captured. The directional and zooming controls work particularly well on this collection, as demonstrated in the companion video. In addition, the reconstruction was registered to an aerial image, and therefore a map appears as part of the interface. For the **Great Wall** collection, one interesting way to view the images is to play them in order as a slideshow, simulating the effect of walking along the wall in the photographer's footsteps. For this collection, the mesh-based projection surfaces are particularly effective.

I also demonstrate the system on several "uncontrolled" sets consisting of images downloaded from Flickr. **Trevi Fountain** is a set of 360 photos of the Trevi Fountain in Rome, registered from 466 photos matching "trevi AND rome." **Notre Dame** is a set of 597 photos of the front façade of the Notre Dame Cathedral in Paris. These photos were registered starting from 2,635 photos matching the search term "notredame AND paris." **Yosemite** is a set of 325 photos of Half Dome in Yosemite National Park, registered from 1,882 photos matching "halfdome AND yosemite."

The **Trevi Fountain** collection contains several interesting, detailed sculptures. The video segment for this collection demonstrates the object selection feature, in which the system finds and transitions to a good photo of an object selected by the user, as shown in Figure 5.6. **Notre Dame** contains photos at many different levels of detail, from images taken from over a hundred meters away from the cathedral, to extreme close-ups showing intricate details of the sculptures above the central portal. The total difference in scale between the extreme views is a factor of more than 1,000. Thus, the zooming features of Photo Tourism are well-suited to this collection. The video also demonstrates the annotation transfer feature on this collection, as well as the selection of annotations for display based on the scale of the current image. Finally, the **Half Dome** collection contains many spectacular images of the mountain during different seasons and times of day. The stabilized slideshow feature, demonstrated in the video and in Figure 5.7, is an effective way to view the resultant variation in the appearance of the mountain.

After the initial reconstruction of the **Half Dome** collection, I aligned it to a digital elevation map using the approach described in Chapter 3, Section 3.4.1. I then registered a historical photo, Ansel Adam's "Moon and Half Dome," to the data set, using the automatic method described in Section 5.3.1. The result is an estimate of where Adams was standing when he took the photo. Figure 5.8 shows a synthetic rendering of Half Dome from this estimated position.

5.5 Discussion

This chapter described Photo Tourism, a new interface for exploring photo collections related by place. Photo Tourism situates the user in a 3D world among the reconstructed photo collection, and provides new tools for searching for images, including zooming controls, directional controls, and an object selection tool. Photo Tourism also enables new ways to visualize and render scenes, through 3D transitions between images and stabilized slideshows. Finally, it provides a powerful annotation mechanism that can be used to label an entire collection of images at once.

Photo Tourism also has several limitations. The 3D transitions work well for images of planar or near-planar parts of a scene, such as walls. The results can be less attractive for images that view more complex 3D geometry, or for images that view multiple dominant planes. The mesh-based projection surfaces can handle non-planar geometry, but produce their own artifacts, especially where the geometry is inaccurate or unmodeled, or when people or other foreground objects are projected onto the mesh. More accurate geometry produced, e.g., from a laser scan or with multi-view stereo, would solve some, but not all, of these problems. Hence, better methods for rendering transitions between photos containing complex surfaces and foreground objects is an important open research challenge.

The navigation controls provided by Photo Tourism are primarily photo-centric, designed to move the user from one photo to another. While these can be effective tools for exploring the photo collection, they are not necessarily the best interface for exploring the *scene* itself. A more common approach, used in most 3D games, is to give users continuous control over their movement, rather than constraining them to a particular fixed set of views. Of course, in Photo Tourism, users can choose to use the free-viewpoint navigation controls, but they then lose the realistic views afforded by the images. In any case, it is not clear that even free-viewpoint navigation controls are the ideal interface for exploring scenes, as they can provide *too* many degrees of freedom, leaving a user with too many possible directions to explore. In the next chapter, I present a second user interface designed for exploring scenes. The interface analyzes the distribution of camera viewpoints in large photo collections to create sets of constrained 3D navigation controls tailored to specific locations.

Chapter 6

FINDING PATHS THROUGH THE WORLD’S PHOTOS

The previous chapter described Photo Tourism, a 3D photo browser that provides new geometric controls for exploring photo collections. Let us now consider a slightly different task: exploring *scenes*. How well does Photo Tourism work as a tool for exploring a large space such as the Prague Old Town Square? As demonstrated in the previous chapter and accompanying video, Photo Tourism allows us to move around the square in several ways. We can zoom in or out of a photo to see more detail or more context. We can move to the left or right to see a photo of the next building over in a row of shops. We can also drag a box around an interesting object or building, and transition to a good photo of that selected object.

All of these controls are potentially useful, but what are the *best* possible controls? To attempt to answer that question, let’s step back for a moment, and consider what we want to accomplish in exploring the scene. Suppose we have never been to the square. We might know that it is a significant place, with interesting sights, but we may not know exactly where we should go or what we should see. Indeed, the square contains several notable landmarks, including the Old Town Hall with its astronomical clock, the Tỳn Cathedral, the St. Nicholas Church, and the statue of Jan Hus. An ideal 3D navigation interface might make it easy to find all of these landmarks, and offer additional controls, such as a way to orbit the Jan Hus statue and see it from different angles, when suitable. Another scene—the Vatican, say—will have a completely different set of interesting views (the Sistine Chapel, Michelangelo’s Pietà, St. Peter’s Square) and might require a different set of controls. Scenes also come in a wide variety of shapes and sizes. The Old Town Square consists of rows of buildings, while other scenes, such as the Statue of Liberty, consist of a single, dominant object.

This variability is at the heart of the problem I consider in this chapter: how can we discover or devise the best navigation controls for any given scene? The solution I present in this chapter is to derive controls from large Internet photo collections of the scene. These collections of photos, captured by many people, provide an extremely valuable source of information for creating controls for exploration tasks, as they represent samples of how people actually experienced the scene, where they stood, and what views they found interesting.

The navigation controls provided by Photo Tourism also adapt to photo collections, but only at a very local level. For each photo, considered in isolation, a set of possible details, similar images, left and right neighbors, etc., are computed. Analyzing the structure of the photo collection at this level might miss important features of the scene. Of course, Photo Tourism also provides standard continuous, free-viewpoint navigation controls, as do the vast majority of games and simulators, that let users move wherever they want. However, while such unconstrained controls might make it *possible* to explore the scene in any particular way, they still might not make it *easy*; they may provide too many degrees of freedom when only a few are required (e.g., the one-dimensional case of rotating around an object), and do not directly solve the problem of getting users where they really want to, or ought to, go. Thus, devising a good interfaces for exploring scenes is a challenging problem. As image-based rendering (IBR) methods scale up to handle larger and larger scenes, the problem of devising good viewpoint controls becomes even more critical.

The overarching issue here is that to quickly and efficiently explore understand a scene, a user may want a variety of controls—Photo Tourism-style controls (for instance, for finding a photo of a selected object), continuous, object-based navigation controls (to orbit around a statue), automatic controls (to take the user to an interesting landmark), and perhaps even controls that suggest where to go next. These various controls depend on the task the user wants to perform, the contents of the scene (e.g., does it contain objects?), and the part of the scene the user is currently visiting.

One solution to this problem is to manually plan a set of desired paths through a scene and capture those views, then design a user interface that only exposes controls for following these paths. This approach is used for many IBR experiences such as panoramas, object movies [21], and moviemaps [87]. While effective, this kind of approach cannot leverage the vast majority of existing photos, including the millions of images of important landmarks available through the Internet. While deriving such controls is a challenging research problem, using Internet photo collections to generate controls also has major advantages. Internet photo collections represent samples of views from places people actually stood and thought were worth photographing. Therefore, through consensus, they tend to capture the “interesting” views and paths through a scene. I leverage this

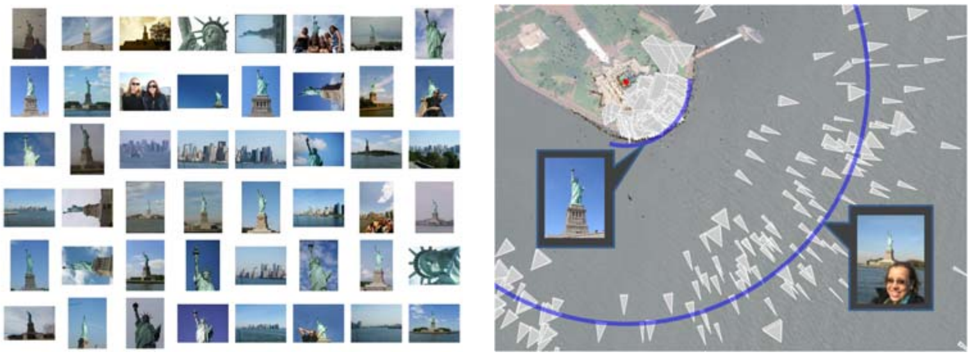


Figure 6.1: *Paths through the Statue of Liberty photo collection.* Left: several Flickr images of the Statue of Liberty, from a collection of 388 input photos. Right: reconstructed camera viewpoints for this collection, revealing two clear orbits, shown here superimposed on a satellite view. The goal is to automatically discover such orbits and other paths through view space to create scene-specific controls for browsing photo collections.

observation to generate controls that lead users to interesting views and along interesting paths.

For example, consider the overhead view of a reconstruction of the Statue of Liberty from 388 photos downloaded from Flickr, shown in Figure 6.1. Most of the photos in this collection were captured from the island or from boats out in the water, and are distributed roughly along two circular arcs. Even if we knew absolutely nothing about the content of the scene, this distribution of views would tell us that (a) there is some very important thing in the scene, since almost all images are trained on the same point, and (b) the object of interest is typically viewed from a range of angles forming arcs around the object. The distribution thus suggests two natural orbit controls for browsing this scene. While the viewpoints in this scene have a particularly simple structure, I have observed that many Internet collections can be modeled by a combination of simple paths through the space of captured views.

In this chapter, I describe the Pathfinder system,[1] which extends Photo Tourism to provide a fluid, continuous IBR experience with effective *scene-specific* controls derived from the distribution of views in a photo collection. As with Photo Tourism, there are both rendering and navigation aspects to this problem. On the rendering side, I introduce new techniques for selecting and warping images for display as the user moves around the scene, and for maintaining a consistent scene

[1] The work described in this chapter originally appeared at SIGGRAPH 2008 [136]. Rahul Garg contributed significantly to the ideas and results described in this chapter.

appearance. On the navigation side, I describe my approach for automatically generating good controls from a photo collection and integrating these controls into the user interface. These two problems, rendering and navigation, are not separate, but are intimately linked together, for two reasons. First, my approach to rendering is based on a new view scoring function that predicts the quality of reprojecting an input photo to a new viewpoint. This function is used to select good views for rendering, but also to find trajectories through viewpoint space with high visual quality, suitable for use as navigation controls. Second, the projection surfaces used for rendering are selected based on how the user is currently moving through the scene.

The remainder of this chapter describes the various components of the Pathfinder system. First, however, I review related work in 3D navigation interfaces to motivate the design of the system.

6.1 Related work on 3D navigation

A primary goal of the Pathfinder system is to create controls that make 3D navigation through a scene easy and intuitive. While computer graphics researchers have concentrated mainly on *rendering* of scenes, there has also been a wealth of research on interfaces for 3D navigation, primarily in the field of human-computer interaction. This section surveys previous work on 3D navigation and distills a set of key guiding principles for the design of navigation interfaces.

6.1.1 Principles of 3D scene navigation

3D scene navigation refers to manipulating a first-person viewpoint in a virtual 3D scene. It involves translating some input—from a mouse, keyboard, head-mounted display, or other device—into a 6D set of camera parameters which specify the 3D position and 3D orientation of the camera (sometimes with an additional degree of freedom for zoom). 3D navigation is a surprisingly difficult problem; without good exploration and visualization tools it can be easy to get lost in a scene [131]. The development of interfaces for navigating virtual 3D environments dates back at least to the work of Sutherland in the 1960's [142]. Since then such interfaces have appeared in numerous settings: games, simulations, 3D modeling software, mapping applications such as Google Earth, virtual reality systems, and others.

Before considering the design of navigation interfaces, however, it is useful to consider the goals of 3D navigation, i.e., what tasks a user may be interested in performing in a 3D environment. Tan, *et al.*, [145], present a task-based taxonomy of navigation controls which enumerates three categories of tasks:

- **Exploration**: gaining survey knowledge, e.g., building a mental map of the scene.
- **Search:** finding and moving to a particular view, region, or object.
- **Object inspection:** exploring a range of desired views of a particular object.

While this list is not necessarily comprehensive, these tasks are relevant to the kinds of experiences I seek to enable in Pathfinder. The first of these, exploration, is particularly relevant; my work focuses on exploration of popular real world scenes, scenes which many users may be interested in visiting or learning more about, but with which they may not initially be intimately familiar. For such users, an ideal interface might be a set of simplified controls that make it easy to find the interesting parts of the scene.

What kind of controls best aid a user in accomplishing these tasks? While there is no simple answer, certain principles for 3D interface design can be found in the literature on 3D navigation.

The right navigation controls are scene- and task-dependent. Researchers have found that navigation controls that are entirely natural in one setting can be completely ineffective in another. In one study, Ware and Osbourne [157] evaluated several types of navigation controls, including "flying vehicle," where a user moves around the scene by directly moving and turning the virtual camera, and "scene-in-hand," where the user instead moves and rotates the scene itself. They found that in scenes consisting of a single object, test subjects overwhelmingly found the scene-in-hand controls to be most natural and the flying vehicle controls to be the least natural, while the exact opposite was true for complex scenes such as a maze. Similarly, different types of controls have been shown to be more useful depending on the task a user is trying to perform, such as getting an overview of a scene or inspecting a particular object [145].

For these reasons, many 3D interfaces, including those used in the Maya 3D modeling software and several VRML browsers, provide different sets of controls which users can toggle between, such as camera-centric and object-centric controls. However, manually toggling between modes may be more cumbersome than necessary. Another approach is to *learn* the right set of controls for a given scene, and a few researchers have explored this idea. Singh and Balakrishnan [131] propose a system that attempts to learn good visualizations of a scene based on a set of previous interactions with the same scene. In the Pathfinder system, such controls are learned from the distribution of views in a large photo collection.

Constrained navigation can be more efficient. A common problem with 3D viewers that allow completely unrestricted motion is that it can be easy to get “lost in space,” getting in a state where the user doesn’t know where they are or how to get back. Even if a user stays grounded, unconstrained controls may provide more degrees of freedom than are strictly necessary, making them more difficult to use. Furthermore, common 2D input devices, such as a mouse, cannot always be naturally mapped to the 6 or 7D space of camera views. Constraining the navigation controls to a smaller number of degrees of freedom is one way to solve these problems. Indeed, some approaches, such as Galyean’s River Analogy [50], simply move users automatically along a pre-specified path, but give the user freedom to control certain parameters, such as viewing direction and speed. Hanson, *et al.* [64] suggest several design techniques for adding such constraints to a controller, such as making the orientation of the camera dependent on the position. They also present a preliminary evaluation showing that adding constraints can help improve a user’s survey knowledge about an environment. Several specific constrained navigation techniques have been developed, such as Hovercam [78], which converts 2D input to a generalized orbit around an arbitrary object for tasks that require close object inspection. The main drawback of constrained navigation is that it can give a user less of a sense of freedom [64].

Use high-level goals and automatic navigation when possible. Often, *automatic* navigation is the easiest way for a user to satisfy a certain goal [63, 36, 38]. For instance, suppose a user is virtually visiting Prague and has the goal of finding the Jan Hus Memorial. It would probably be much easier for the user to express that goal to the system, and have the system automatically move the user to the Memorial, than for the user to try and hunt for it himself. A good example of a system that implements this behavior is Google Earth, which supports automatic navigation through its search features. In Google Earth, a user can search for a place by typing in a name or address, and a smooth camera motion automatically brings the user to the destination. The path along which the user is transported is a factor in how comprehensible the transition is. The quickest path from point A to point B is teleportation—instantaneously cutting from one view to the next; however, teleportation can be quite disorienting [14]. Another approach, explored by Santos, *et al.* [36], moves the user along shortest paths in 3D space between two views. Drucker and Zeltzer [38] describe a general system where multiple tasks and constraints can be combined in a path planning algorithm to generate automatic, natural camera motions.

The user's spatial position and motion should always be clear. Unless users can easily tell where they are and how they are moving, they can easily become disoriented [63]. If the scene has a variety of visual content, then landmarks assist users in keeping track of their position [52]. Other, more explicit cues have also been investigated, such as augmenting a scene with gridlines or additional landmarks [28]. One particular cue—an overhead map which displays or tracks the user's position—is very commonly used [28, 152], and has been shown to substantially aid in navigation tasks [27].

These interface considerations are all relevant to the design of the Pathfinder system. However, Pathfinder takes these ideas a step further by trying to *automatically* infer good sets of navigation controls by analyzing how many different people have photographed a given scene. In particular, Pathfinder finds *scene-specific*, constrained navigation controls, such as orbits around objects, and suggests these controls to the user. In addition to these controls, Pathfinder attempts to find important views of the scene, and provides automatic controls for moving to these important views. Paths between these views are generated with a new path planning algorithm that attempts to create understandable paths by mimicking how people move through the scene.

6.2 System overview

The Pathfinder system takes as input a set of photos from a variety of viewpoints, directions, and conditions, taken with different cameras, and with potentially many different foreground people and objects. From this input, the systems creates an interactive 3D browsing experience in which the scene is depicted through photographs that are registered and displayed as a function of the current viewpoint. The system guides the user through the scene by means of a set of automatically computed controls that expose detected orbits, panoramas, interesting views, and optimal trajectories specific to the scene and distribution of input views.

The system consists of the following components:

A set of input images and camera viewpoints. The input is an unstructured collection of photographs taken by one or more photographers. We register the images using the SfM algorithms described in Chapters 3 and 4.

Image reprojection and viewpoint scoring functions that evaluate the expected quality of rendering each input image at any possible camera viewpoint. The reprojection process takes into account such factors as viewpoint, field of view, resolution, and image appearance to synthesize high quality rendered views. The viewpoint scoring function can assess the quality of any possible rendered view, providing a basis for planning optimal paths and controls through viewpoint space.

Navigation controls for a scene. Given the distribution of viewpoints in the input camera database and the viewpoint scoring function, the system automatically discovers scene-specific navigation controls such as orbits, panoramas, and representative images, and plans optimal paths between images.

A rendering engine for displaying input photos. As the user moves through the scene, the rendering engine computes the best scoring input image and reprojects it into the new viewpoint, transformed geometrically and photometrically to correct for variations between images. To this end, I introduce an *orbit stabilization* technique for geometrically registering images to synthesize motion on a sphere, and an appearance stabilization technique for reducing appearance variation between views.

A user interface for exploring the scene. A 3D viewer exposes the derived controls to users, allowing them to explore the scene using these controls, move between different parts of the scene, or simply fly around using traditional free-viewpoint navigation. These controls combine in an intuitive way.

These components are used in the three main stages of the Pathfinder system. First, the offline structure from motion process recovers the 3D location of each photograph. Next, the scene is automatically processed to derive controls and optimal paths between images. Finally, this augmented scene can be browsed in an interactive viewer.

The remainder of the chapter is organized as follows. In the next section, I describe the view scoring function at the heart of the system. Section 6.4 describes how scene-specific navigation controls are derived from a photo collection and integrated with other modes of navigation. Section 6.5 describes the scene rendering engine. Section 6.6 describes the path planning algorithm used to compute transitions between views. Section 6.7 describes techniques used to stabilize the appearance of images as the user moves through the scene. Section 6.8 demonstrates the approach on a variety of scenes and for a range of visualization tasks including free-form 3D scene browsing, object movie creation from Internet photos or video, and enhanced browsing of personal photo collections. Finally, Section 6.9 discusses limitations of the system and ideas for future work.

6.3 Viewpoint scoring

My approach is based on (1) the ability to reproject input images to synthesize new viewpoints, and (2) to evaluate the expected quality of such reprojections. The former capability enables rendering, and the latter is needed for computing controls that move the viewer along high quality paths in viewpoint space. In this section I describe my approach for evaluating reprojection quality.

As in Photo Tourism, the Pathfinder system is given a database of input images $\mathcal{I}$ whose camera parameters (intrinsics and extrinsics) have been computed. The term *camera* denotes the viewing parameters of an input image I_j. Each camera C_j observes a set of points $\mathrm{Points}(C_j)$. The term *image* denotes an *input* photo I_j from the database. The term *viewpoint* denotes a set of viewing parameters v in the virtual scene, and the term *view* denotes an *output* photo that we seek to render from a given viewpoint. A view is produced by reprojecting an input photo, through a rendering process, to the desired new viewpoint v.

The first step is to define a *reprojection score* $S_{\mathrm{proj}}(I_j, v)$ that rates how well a database image I_j can be used to render a new view at v. The best reprojection is obtained by maximizing $S_{\mathrm{proj}}(I_j, v)$ over the image database, yielding a *viewpoint score* $S_{\mathrm{view}}(v)$:

$$S_{\mathrm{view}}(v) = \max_{I_j \in \mathcal{I}} S_{\mathrm{proj}}(I_j, v) \tag{6.1}$$

Ideally, $S_{\mathrm{proj}}(I_j, v)$ would measure the difference between the synthesized view and a real photo of the same scene captured at v. Because the system does not have access to the real photo (unless v is

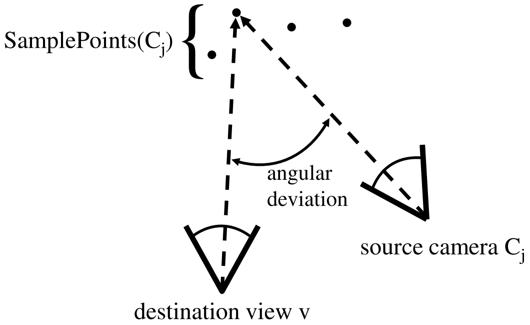


Figure 6.2: *Angular deviation.* The angular deviation factor of the viewpoint score penalizes the average angular deviation between rays from the current viewpoint and the image under consideration through a set of points in the scene. Here, the angle $\mathrm{penalty}_{\mathrm{ang}}$ is shown for one of the points.

exactly coincident with an input camera), I instead use the following three criteria to estimate how well image I_j represents v:

1. Angular deviation: the relative change in viewpoint between C_j and v should be small.
2. Field of view: the projected image should cover as much of the field of view of v as possible.
3. Resolution: image I_j should be of sufficient resolution to avoid blur when projected into v.

For a given image and viewpoint, each of these criteria is scored on a scale from 0 to 1. To compute these scores, we require a geometric proxy for each image to use as a projection surface during reprojection into v; the proxy geometry is discussed in Section 6.5.2.

Angular deviation. The angular deviation score $S_{\mathrm{ang}}(I_j, v)$ is proportional to the angle between rays from viewpoint v through a set of points in the scene and rays from camera C_j through the same points. This is akin to the *minimum angular deviation* measure used in Unstructured Lumigraph Rendering [17]. Rather than scoring individual rays in the database, however, the system scores entire images by averaging the angular deviation over a set of 3D points observed by I_j These points, denoted $\mathrm{SamplePoints}(C_j)$, are selected from $\mathrm{Points}(C_j)$ for each image in a pre-processing step. To compute $\mathrm{SamplePoints}(C_j)$, I project $\mathrm{Points}(C_j)$ into I_j, distribute them into a 10×10 grid of bins defined on the image plane, then select one point from each non-empty bin. This ensures that the points are relatively evenly distributed over the image.

The average angular deviation is computed as:

$$S'_{\text{ang}}(I_j, v) = \frac{1}{|\,\text{SamplePoints}(I_j)|} \sum_{\mathbf{X} \in \text{SamplePoints}(I_j)} \text{angle}(\mathbf{X} - \mathbf{c}_j, \mathbf{X} - \mathbf{c}(v)). \tag{6.2}$$

where $\mathbf{c}_j$ is the 3D position of camera C_j, $\mathbf{c}(v)$ is the 3D position of viewpoint v, and $\text{angle}(\mathbf{a}, \mathbf{b})$ gives the angle between two rays $\mathbf{a}$ and $\mathbf{b}$. The average deviation is clamped to a maximum value of $\alpha_{\max}$ (for all my examples, I set $\alpha_{\max} = 12°$), and mapped to the interval $[0, 1]$:

$$S_{\text{ang}}(I_j, v) = 1 - \frac{\min(S'_{\text{ang}}(I_j, v), \alpha_{\max})}{\alpha_{\max}}. \tag{6.3}$$

A score of 1 indicates that C_j and v are coincident viewpoints, and a score of 0 means that the average angle between corresponding rays is greater than $\alpha_{\max}$.

Field-of-view score. The field-of-view score $S_{\text{fov}}(I_j, v)$ is computed by reprojecting I_j into v and computing the area of the view at v that is covered by the reprojected image. I compute a weighted area, with higher weight in the center of the view, as I find that it is generally more important to cover the center of the view than the boundaries. The weighted area is computed by dividing the view into a grid of cells, $\mathcal{G}$, and accumulating weighted contributions from each cell:

$$S_{\text{fov}}(I_j, v) = \sum_{G_i \in \mathcal{G}} w_i \frac{\text{Area}(\text{Project}(I_j, v) \cap G_i)}{\text{Area}(G_i)}, \tag{6.4}$$

where $\text{Project}(I_j, v)$ is the 2D polygon resulting from reprojecting image I_j into viewpoint v (if any point of the projected image is behind viewpoint v, Project returns the empty set); Area is the area, in pixels, of a 2D polygon. The weights w_i approximate a 2D Gaussian centered at the image center.

Resolution score. Finally, the resolution score $S_{\text{res}}(I_j, v)$ is computed by projecting I_j into v and finding the average number of input pixels of I_j used per output screen pixel. An image of sufficient resolution should have at least one input pixel per output pixel. This score is thus computed as the ratio of the number of pixels in I_j to the area, in screen pixels, of the reprojected image

$\mathrm{Project}(I_j, v)$:

$$S'_{\mathrm{res}}(I_j, v) = \frac{\mathrm{Area}(I_j)}{\mathrm{Area}(\mathrm{Project}(I_j, v))}. \tag{6.5}$$

If this ratio is greater than one, then, on average, the resolution of I_j is sufficient to avoid blur when I_j is projected into the view (the system uses mip-mapping to avoid aliasing, so a ratio greater than one is acceptable). I transform S'_{res} to map the interval $[\mathrm{ratio}_{\mathrm{min}}, \mathrm{ratio}_{\mathrm{max}}]$ to $[0, 1]$:

$$S_{\mathrm{res}}(I_j, v) = \mathrm{clamp}\left(\frac{S'_{\mathrm{res}}(I_j, v) - \mathrm{ratio}_{\mathrm{min}}}{\mathrm{ratio}_{\mathrm{max}} - \mathrm{ratio}_{\mathrm{min}}}, \epsilon, 1\right), \tag{6.6}$$

where $\mathrm{clamp}(x, a, b)$ clamps x to the range $[a, b]$. I use values of 0.2 and 1.0 for $\mathrm{ratio}_{\mathrm{min}}$ and $\mathrm{ratio}_{\mathrm{max}}$, and enforce a non-zero minimum resolution score ϵ because I favor viewing a low-resolution image rather than no image at all.

The three scores described above are multiplied to give the final reprojection score S_{proj}:

$$S_{\mathrm{proj}}(I_j, v) = S_{\mathrm{ang}}(I_j, v) \cdot S_{\mathrm{fov}}(I_j, v) \cdot S_{\mathrm{res}}(I_j, v). \tag{6.7}$$

Again, to compute the viewpoint score S_{view}, the reprojection score is computed for each input image; S_{view} is the maximum over these scores.

6.4 Scene-specific navigation controls

In Section 6.1 I laid out several guidelines for the design of 3D navigation control. Three primary guidelines are that *the right navigation controls are scene- and task-dependent*, *constrained navigation can be more efficient*, and one should *use high-level goals and automatic navigation when possible*. Based on these observations, the Pathfinder system attempts to (a) derive constrained, *scene-specific* controls for different parts of the input scene, (b) find interesting views that users might want to visit, and (c) provide automated, optimized paths from getting between any pair of views or controls.

To these ends, the large Internet photo collections used as input to the system are extremely useful, in that the tend to cluster along interesting paths through a scene, and around interesting objects, and thus provide important cues as to what controls should be provided. Of course, the regions near the input samples will also be the areas where we can likely render good views of the scene (i.e., views for which $S_{\text{view}}(v)$ is high). I take advantage of this information through a set of automatic techniques for deriving controls from a reconstructed scene. The result of this analysis is a set of *scene-specific controls*. For instance, the Statue of Liberty scene shown in Figure 6.1 has two scene-specific controls: an orbit control for the cluster of views taken on the island, and a second orbit for views taken out on the water.

In the rest of this section, I outline the navigation modes of the Pathfinder system and describe how scene-specific controls are computed.

6.4.1 Navigation modes

Pathfinder supports three basic navigation modes:

1. Free-viewpoint navigation.
2. Constrained navigation using scene-specific controls.
3. Optimized transitions from one part of the scene to another.

Free-viewpoint navigation. The free-viewpoint navigation mode allows a user to move around the scene using standard 6-DOF (3D translation, pan, tilt, and zoom) “flying vehicle” navigation controls. I also provide an *orbit* control which allows for scene-in-hand-style rotation about an object.

While free-viewpoint controls give users the freedom to move wherever they choose, they are not always the easiest controls for move around complex scenes, as the user has to continually manipulate many degrees of freedom while (at least in IBR) ideally staying near the available photos.

Scene-specific controls. Pathfinder supports two types of scene-specific controls: orbits and panoramas. Each such control is defined by its type (e.g., orbit), a set of viewpoints, and a set of images associated with that control. For an orbit control, the set of viewpoints is a circular arc of a given radius centered at and focused on a 3D point; for a panorama the set of viewpoints is a range of viewing directions from a single 3D nodal point. When a control is active, the user can navigate through the corresponding set of viewpoints using the mouse or keyboard. In addition to scene-specific orbits and panoramas, I also compute a set of representative *canonical views* for a scene.

Transitions between controls. The final type of control is a transition between scene-specific controls or canonical views. Pathfinder allows users to select a control or image they are interested in exploring next; once a destination is selected, the virtual viewpoint is then moved on an automated path to the new control or image. The transition is computed using a new path planning algorithm that adapts to the database images, as described in Section 6.6. This method of directly selecting and moving to different parts of the scene is designed to make it easy to find all the interesting views, following the guideline that automatic navigation is often the most efficient.

6.4.2 Discovering controls

Once a scene is reconstructed, the Pathfinder system automatically analyzes the recovered geometry to discover interesting orbits, panoramas, and canonical views. Orbits typically correspond to interesting objects which can be viewed from many different angles, panoramas are viewpoints from which the scene is photographed from many different viewing directions (e.g., the top of a tall building), and canonical views correspond to important viewpoints represented many times in the database. This section describes how each of these are discovered in a scene.

Orbit detection. I define an orbit to be a distribution of views positioned on a circle and converging on (looking at) a single point, denoted the convergence point $\mathbf{p}_{\text{focus}}$. I only consider circles parallel to the ground plane as valid orbits, as most orbits that appear in the world are around objects photographed from different directions by people standing on a floor or other plane. I further constrain $\mathbf{p}_{\text{focus}}$ to lie on a vertical axis o passing through the center of the circle and perpendicular to the ground plane. The height of $\mathbf{p}_{\text{focus}}$ determines the tilt at which the object of interest is viewed. Because full 360° view distributions are uncommon, I allow an orbit to occupy a circular arc. I define a good orbit to be on that satisfies the following objectives, as illustrated in Figure 6.3:

- **quality**: maximize the quality of rendered views everywhere along the arc.
- **length**: prefer arcs that span large angles.
- **convergence**: prefer views oriented towards the center of the orbit.
- **object-centered**: prefer orbits around solid objects (as opposed to empty space).

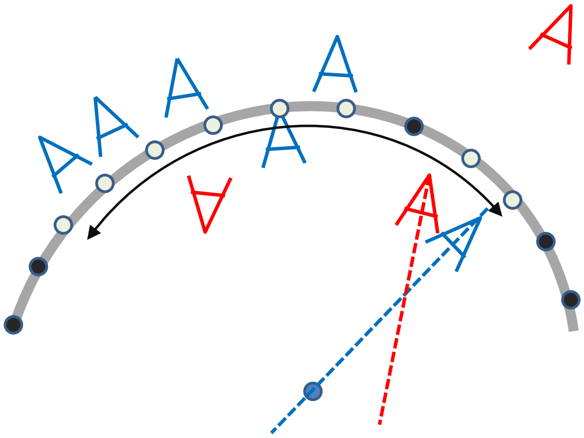


Figure 6.3: *Scoring a candidate orbit.* An orbit is evaluated by regularly sampling viewpoints along the candidate arc (here, the arc is shown in gray, and the samples are shown as small circles drawn on top of the arc). For each sample position, we want to find a nearby image with a high reprojection score that is oriented towards the orbit center (thus eliminating the red cameras). The light samples score well on these objectives while the black samples do not. We search for large arcs where the sum of the sample scores is high, and that do not contain large low-scoring gaps. Here, the optimal subset of the arc is shown with the curved black arrow.

Given these objectives, the problem of detecting orbits involves (1) defining a suitable objective function, (2) enumerating and scoring candidate orbits, and (3) choosing zero or more best-scoring candidates. One could imagine many possible techniques for each of these steps; in what follows, I describe one approach that has worked quite well in practice.

I first define my objective function for evaluating orbits. An orbit is fully specified by a center orbit axis o and an image I_j; the image defines the radius of the circle (the distance of the camera center from o), and the convergence point $\mathbf{p}_{\text{focus}}$ on the orbit axis ($\mathbf{p}_{\text{focus}}$ is the closest point on the axis o to the optical axis of camera C_j). Assume further that C_j is the point on the arc midway between the endpoints of the arc. The geometry of such an orbit is illustrated in Figure 6.4.

I define the orbit scoring function, $S_{\text{orbit}}(o, I_j)$, to be the sum of individual view scores, $S_{\text{orbit}}(o, I, \theta)$, sampled at positions θ along the arc. To compute $S_{\text{orbit}}(o, I_j, \theta)$ at a sample viewpoint $v(\theta)$ (the viewpoint on the arc at angle θ from I_j), I look for support for that view in the set of database images $\mathcal{I}$. In particular, I score each image $I_k \in \mathcal{I}$ based on (a) how well I_k can be used to synthesize a view at $v(\theta)$ (estimated using the reprojection score $S_{\text{proj}}(I_k, v(\theta))$), and (b) whether I_k is looking

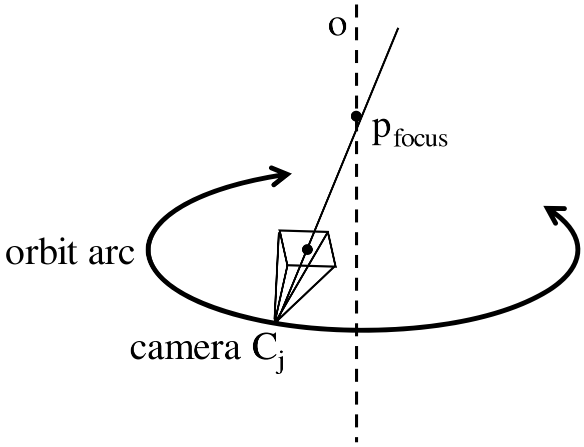


Figure 6.4: *An orbit can be defined by an orbit axis and a camera.* A vertical orbit axis orbit o, combined with a camera C_j (which defines the orbit radius), defines a family of orbit arcs around the axis. This family consists of all arcs of a circle centered on the axis which passes through the camera center (and for which the camera center is at the middle of the arc). The focal point $\mathbf{p}_{\text{focus}}$ is the point on the axis o closest to the ray through the center of C_j.

at the orbit axis (the *convergence score*). $S_{\text{orbit}}(o, I_j, \theta)$ is then the score of the best image I_k at $v(\theta)$:

$$S_{\text{orbit}}(o, I_j, \theta) = \max_{I_k \in \mathcal{I}} \{S(I_k, v(\theta)) \cdot f_o(I_k)\}. \tag{6.8}$$

The convergence score $f_o(I_k)$ is defined as:

$$f_o(I_k) = \max\left(0, 1 - \frac{\psi}{\psi_{max}}\right) \tag{6.9}$$

where $\psi = \text{angle}(\mathbf{v}(C_k), \mathbf{p}_{\text{focus}} - \mathbf{p}(C_k))$, i.e., the angle between the viewing direction $\mathbf{v}(C_k)$ of image I_k and the ray from the optical center $\mathbf{p}(C_k)$ of I_k to $\mathbf{p}_{\text{focus}}$; I use a value of $\psi_{max} = 20°$. This term downweights images for which $\mathbf{p}_{\text{focus}}$ is not near the center of the field of view.

I place a few additional constraints on the images I_k considered when computing $S_{\text{orbit}}(o, I_j, \theta)$:

- $\mathbf{p}_{\text{focus}}$ must be inside the field of view and in front of I_k.
- The tilt of I_k above the ground plane must be less than $45°$ (orbits with large tilt angles do not produce attractive results).
- There must be a sufficient number (I use $k = 100$) of 3D points visible to I_k whose distance

from I_k is less than the orbit radius. I enforce this condition to ensure that we find orbits around an object (as opposed to empty space).

I compute $S_{\text{orbit}}(o, I_j, \theta)$ at every degree along the circle $-180 < \theta \leq 180$. For simplicity, I refer to these sample scores as $s(\theta)$. A good orbit will have a long arc of relatively high values of $s(\theta)$. Simply summing the values $s(\theta)$, however, could favor orbits with a few sparsely scattered good scores. Instead, I explicitly find a long chain of uninterrupted good scores centered around I_j, then add up the scores on this chain. I define this chain as the longest consecutive interval $[-\theta_L, \theta_L]$ such that the maximum $s(\theta)$ in each subinterval of width 15° is at least $\epsilon = 0.01$. This definition allows for small "gaps," or intervals with low scores, in the arc. If this longest chain subtends an angle less than 60°, the score of the orbit is zero. Otherwise, the score is the sum of the individual scores in the chain:

$$S_{\text{orbit}}(o, I_j) = \sum_{k=-L}^{L} s(\theta_k). \tag{6.10}$$

Now that we have an objective function, we need a way to enumerate candidate orbits (mainly for sake of efficiency; we could score *all* possible orbits). My strategy for finding such orbits operates in two stages. First, I compute a set of good candidate orbit axes, by finding axes in 3D space that are the convergence points of many database images. Second, I create and score candidate orbits by pairing each candidate axis with each database image I_j.

To identify a set of candidate orbit axes, I take an approach similar to that of Epshtein *et al.*[40] and use the idea that an interesting object will often occupy the center of the field of view of many images. I first project all cameras onto the ground plane[2] and compute the 2D intersections of the optical axes of all pairs of cameras (discarding intersection points which lie in back of either camera, or which are not approximately equidistant from both cameras). I then compute the density D of these intersection points at each point x:

$$D(x) = \text{number of intersection points within distance } w \text{ of } x.$$

(I use $w = 0.02$, although this value should ideally depend on the scale of the scene). The local maxima in this density function identify vertical axes in the scene which are at the center of many

[2] This projection, reducing the problem to 2D, is possible because all orbit axes are assumed to be vertical.

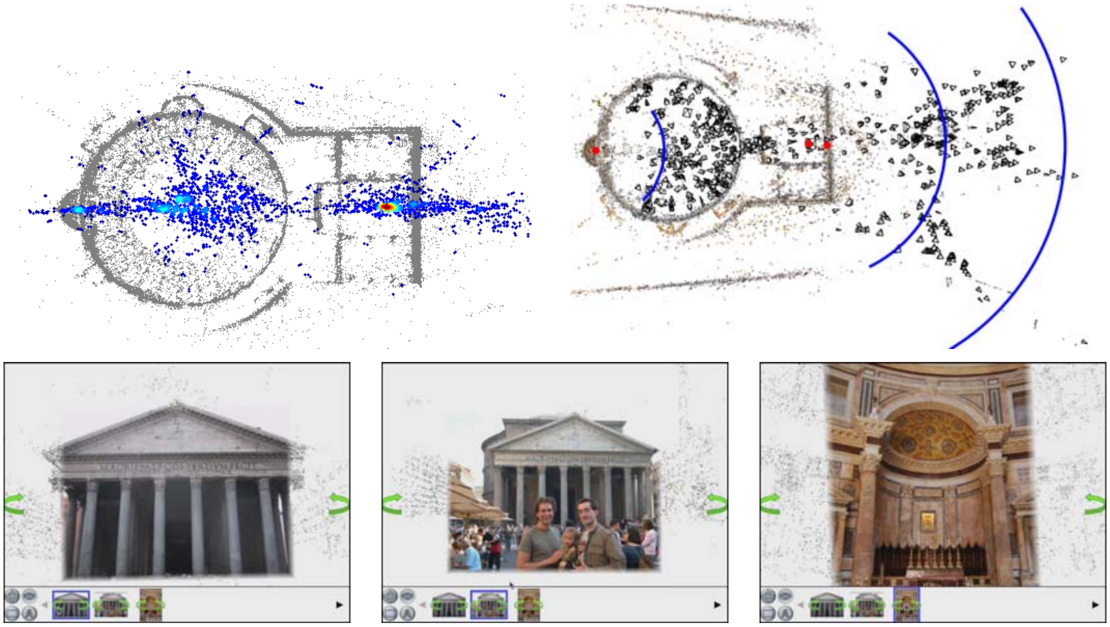


Figure 6.5: *Density function and detected orbits for the Pantheon data set.* Top row, left: the set of intersection points of viewing axes is superimposed on the point cloud of the Pantheon. The color of each point corresponds to the density of points in its neighborhood (a red point has the highest density, a dark blue point the lowest). There are several clear maxima of this function, including a point just behind the front facade and a point at the altar (at the extreme left of the figure).[4] Top row, right: the three final detected orbits shown as blue arcs centered at the red orbit points. Bottom row: images from each of the three detected orbits.

different photos, and are thus potentially interesting. A plot of the density function for the Pantheon dataset is shown in Figure 6.5, along with the three detected orbits. These orbits are also demonstrated in the video accompanying this chapter [107].

Next, I find the point with the highest density $D_{\max}$, then select all local maxima (points that have the highest density in a circle of radius w) that have a density at least $0.3D_{\max}$. These points form the set of candidate orbit axes.

The next step is to find arcs centered at these axes. I form a set of candidate orbits by considering all pairings of orbit axes o and input images I_j. I only accept candidate orbits that satisfy the three constraints enumerated above, i.e., that the point of convergence is in the field of view, the tilt is less than $45°$, and a sufficient number of points are visible in front of the orbit radius. I then evaluate $S_{\text{orbit}}(o, I_j)$ for each such suitable combination of orbit axis and image.

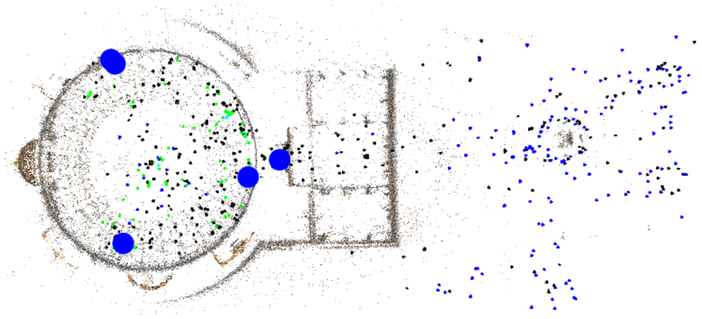


Figure 6.6: *Panoramas detected in the Pantheon.* The panorama detector found five panoramas in the Pantheon collection, shown in this image as blue circles (the two panoramas near the top of the image are quite close to each other). For this scene, it is common for people to stand near the circumference of the interior and take photos looking across the building in different directions.

I now have a set of orbits and a score for each orbit. To form a final set of orbits, I select the orbit with the highest score, remove it and all similar orbits from the set of candidates, then repeat, until no more orbits with a score of at least 0.5 times the maximum score remain. Two orbits are deemed similar if the area of intersection of the two circles defined by the orbits is at least 50% of their average area. An example of detected orbits for the Pantheon collection is shown in Figure 6.5 and in the companion video. In this case, three orbits were detected, two around the outer façade at different distances, and one around the altar in the interior. Note that the furthest detected orbit arc is actually some distance *behind* the reconstructed cameras. The reason why this orbit does not pass closer to the camera centers is the constraint that it should not be too similar to an existing orbit (in this case, the inner orbit around the façade, which is the highest-scoring orbit). Furthermore, this orbits still has a high camera score, as translating a viewpoint backwards from an image along the viewing direction does not affect the angular deviation of rays nearly as much as translating the viewpoint sideways.

When computing viewpoint scores for orbit samples, I use a default vertical field of view of 50 degrees and a default screen resolution of 1024×768 (for the purposes of computing the field of view and resolution scores).

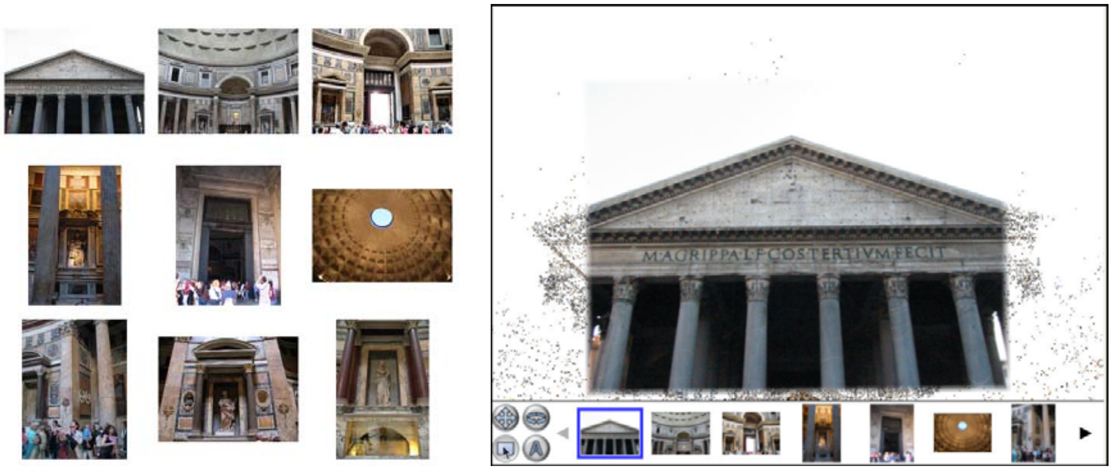


Figure 6.7: *Canonical views detected in the Pantheon data set.* Left: the first nine detected canonical views, using the algorithm of Simon *et al.*[130]. These views include the front façade, altar, oculus, and several sculptures. Right: the canonical views shown in the interface as a row of thumbnails at the bottom of the screen. Clicking on a thumbnail moves the user along an optimal path to the corresponding view.

Panorama detection. A panorama consists of a set of images all taken close to a nodal point, but possibly pointing in different directions. Similar to orbits, a good panorama has good views available from a wide range of viewing directions. To find panoramas in a scene, I first consider each image I_j to be the center of a candidate panorama and compute a panoramas score $S_{\text{pano}}(I_j)$ for each candidate. $S_{\text{pano}}(I_j)$ is computed as the sum of view scores S_{view} for a range of viewing directions around I_j:

$$S_{\text{pano}}(I_j) = \sum_{\phi=-5^\circ}^{25^\circ} \sum_{\theta=0^\circ}^{360^\circ} S_{\text{view}}(v(I_j, \theta, \phi)) \tag{6.11}$$

where $v(I_j, \theta, \phi)$ is the viewpoint located at the optical center of image I_j with viewing direction given by angles θ (pan) and ϕ (tilt). Once each candidate panorama has been scored, I select a set of final panoramas from among this set. To do so, I select the top scoring candidate I^*, remove all images that have a non-zero reprojection score for some view $v(I^*, \theta, \phi)$, then repeat this selection process, until no remaining candidate's score is above a threshold. The panoramas detected in the Pantheon collection are shown in Figure 6.6, and as images in Figure 6.8.

Selecting canonical views. To compute the set of representative, canonical views, I use the scene summarization algorithm of Simon *et al.*[130]. This algorithm seeks to capture the essence of a scene through a small set of representative images that cover the most popular viewpoints. To select a set of canonical views, the algorithm first represents each image I_j as a feature vector $\mathbf{f}_j$, where $\mathbf{f}_j$ has an entry for every 3D point p_k in the scene, and where the kth entry of the vector, $\mathbf{f}_j(k) = 1$ if p_k is visible in image I_j, and 0 otherwise; $\hat{\mathbf{f}}_j$ denotes a normalized version of $\mathbf{f}_j$. The scene summarization algorithm then clusters the normalized feature incidence vectors $\hat{\mathbf{f}}_j$, then selects a representative image for each cluster. The representative images are ordered by the size of their respective clusters; the first represents the most popular view, the second represents the second-most popular view, and so on.

In addition to canonical views computed for the entire scene, I also choose a representative image for each detected orbit and panorama. For a given set of images $\mathcal{S}$ associated with a given control, the representative image is chosen as:

$$\arg\min_{I_j \in \mathcal{S}} \sum_{I_k \in \mathcal{S}, I_j \neq I_k} \hat{\mathbf{f}}_j \cdot \hat{\mathbf{f}}_k, \tag{6.12}$$

i.e., the image whose normalized feature vector is most similar to those of all other images in $\mathcal{S}$.[5]

6.5 Scene viewer and renderer

Once a scene has been reconstructed and augmented with scene-specific controls and canonical views, it can be explored in the interactive scene viewer. The viewer situates the user in the scene, exposes the derived controls to the user, and depicts the scene by continually selecting and warping appropriate images as the user moves. This section describes the navigation interface and rendering components of the viewer.

6.5.1 Navigation interface

As described in Section 6.4, the Pathfinder viewer supports standard 3D translation, panning, and zooming controls, as well as an orbit control, which rotates the camera about a fixed 3D point or

[5] For orbits, the definition of $\hat{\mathbf{f}}$ is slightly different; it only contains an entry for every point inside the orbit circle, as only these points could possibly be salient to the object of interest.

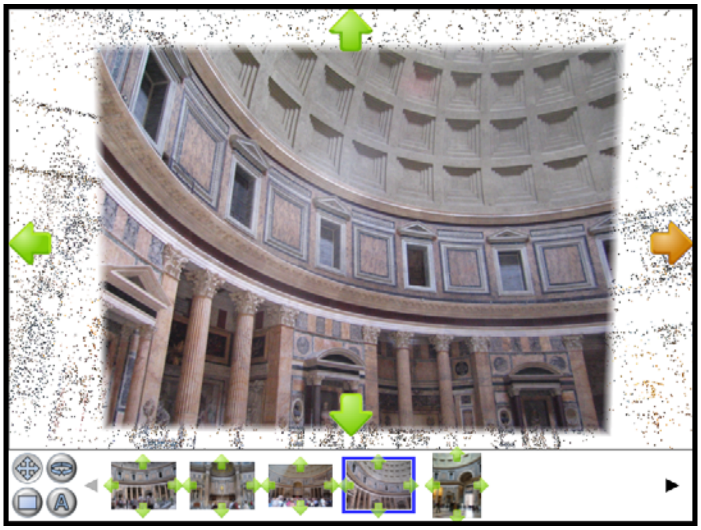


Figure 6.8: *Pathfinder user interface.* The scene viewer displays the currently photo in the main view, and shows suggested controls in the thumbnail pane at the bottom of the screen. The pane is currently showing detected panoramas. Arrows on the sides of the screen indicate which directions the user can pan to find more views.

axis. Typically, the orbital motion is constrained to a ring around an orbit axis, as many objects are viewed from a single elevation, but our viewer also supports orbital motion on a sphere. When the user is moving on a discovered scene-specific orbit, the orbit axis is automatically set to be the axis discovered for that control. Alternatively, the user can manually specify an orbit point by clicking on a 3D point in the scene. Once an orbit point is defined, the user can drag the mouse left and right to orbit around a vertical axis, or up and down to move the viewpoint vertically (when 2D orbital motion is enabled). The process is rapid and seamless: the user simply shift-clicks on a point in the image (the closest 3D feature defines the orbit point), and the orbit begins as soon as the mouse is moved. This orbit procedure is shown in the companion video for this chapter[6] for several scenes, including the Statue of Liberty and the Venus de Milo. Figure 6.9 demonstrates how a user can orbit around an object (in this case, a figurine of Mark Twain) to see it from different angles.

The user interface displays the derived scene-specific controls in a thumbnail pane at the bottom of the screen. The user can choose between displaying pre-defined orbits, panoramas, canonical

[6] http://phototour.cs.washington.edu/findingpaths/

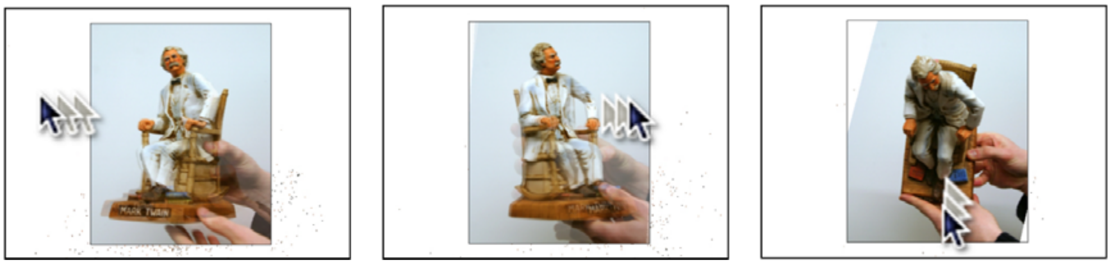


Figure 6.9: *Orbiting a home-made object movie of Mark Twain.* This sequence shows screenshots from a interaction session with a figurine of Mark Twain. The input images are from a video taken of the figurine while rotating it around in front of the camera. The images show a user clicking and dragging the mouse to rotate the object to the right (right image), left (middle image), and down (left image).

images, and all controls, as shown in Figures 6.10, 6.7, and 6.8. Each control is depicted in the pane with a thumbnail created from that control's representative image. The orbit and panorama thumbnails are annotated with small arrow icons to show the type of control.

When the user clicks on a thumbnail, the system computes a path from the user's current location to the selected control (as described in the next section) and animates the virtual camera along that path. If the user arrives at a panorama, left, right, up, and down arrows are drawn on the sides of the screen indicating directions in which more images can be found, as shown in Figure 6.8. Similarly, if the user arrives at an orbit control, left and right orbit arrows appear on the sides of the screen, as shown in Figure 6.10. To determine if a particular arrow cue should be shown, the system computes the viewpoint score for the view the user would see by moving in that direction a given amount. If the score is above a threshold, the arrow is displayed.

6.5.2 Rendering

As the user moves through the scene, the system continually chooses an image to display based on the reprojection score, $S_{\text{proj}}(I, v)$, which rates how well each database image I can be used to render the current viewpoint v (Section 6.3). The image with the top score is selected as the next image to be displayed. If no image has a large enough viewpoint score (i.e., $S_{\text{view}}(v) < \epsilon$), no image is displayed; only the point cloud is rendered.

If the input images are carefully captured and densely sample the space of all viewpoints (as

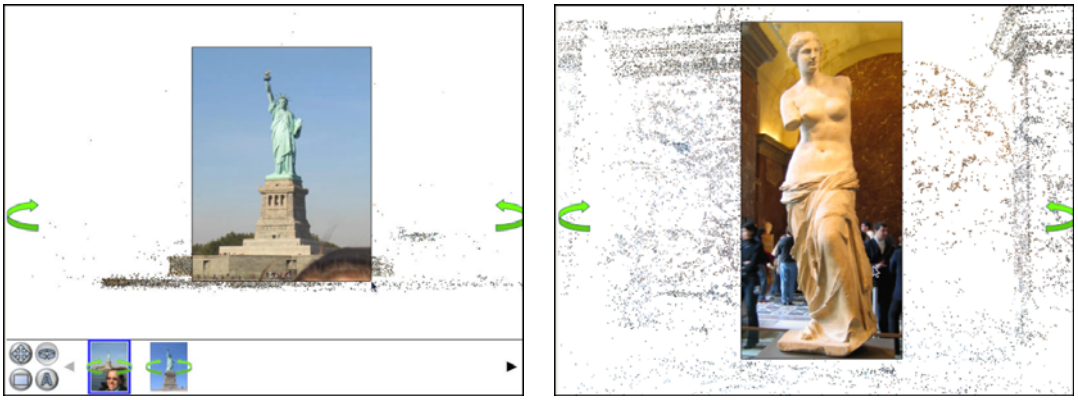


Figure 6.10: *Detected orbits in the Statue of Liberty and Venus data sets.* Left: two orbits were detected for the Statue of Liberty data set, an inner orbit and an outer orbit. These orbits are displayed to the user in the control panel at the bottom of the screen. Right: a single orbit was detected for the Venus data set.

in Quicktime VR object movies and moviemaps), the rendering process is straightforward—simply display the photo corresponding to the desired viewpoint. In practice, however, casually acquired image collections tend to be incomplete and irregularly sampled, with objects centered and oriented differently in each image. Hence, it is necessary to warp each image to better match the desired viewpoint. This is the function of the rendering engine.

To warp an image to match a desired viewpoint v, the image is projected onto a 3D projection surface in the scene, then into v. My system normally renders the scene using planar proxy geometry, but can also render using a dense 3D model, if one is available.

Warping with proxy planes. When the system uses planar proxy geometry, it associates a plane with each image and renders the image by projecting it onto the plane and back into the virtual viewpoint. Planar proxies are also used in the Photo Tourism system, which fits planes to image points for use in transitions. While these best-fit planes work well for some scenes and navigation modes, they can produce jerky motion in situations where the user is moving rapidly through a wide range of views. This is especially true when orbiting; while the viewer is fixated on a particular object, the per-image best-fit planes can stabilize different parts of the scene (including the background) in different images, causing the object to jump around from frame to frame. A demonstration of this

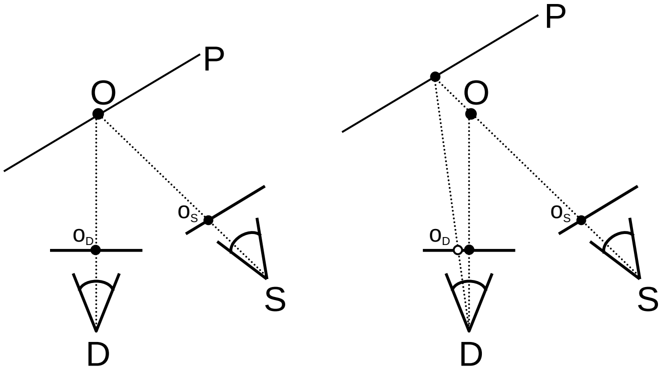


Figure 6.11: *Proxy planes should intersect the orbit point.* Left: to warp an image from view S to D, image S is projected onto the *proxy* plane P, which is then rendered into D. P passes through the orbit point o, ensuring that o is rendered to the correct position in D. Right: P does not pass though the orbit plane, causing o to be rendered to the wrong position.

effect can be found in the video on the project webpage [107].

My solution to this problem is simple but effective. Suppose the user is orbiting around an orbit point o, indicating an object of interest, and suppose we wish to render the scene captured by a "source" photo S into the "destination" viewpoint D. Consider a proxy-plane P in the scene. I compute the warp by perspectively projecting S onto P, then back into D. As shown in Figure 6.11, making P intersect the orbit point o ensures that o projects to the correct location in D. Hence, we can *stabilize* o in the rendered images (make o project to the same pixel (x, y) in all views) by (1) choosing P to intersect o for each input view, and (2) orienting the rendered views so that the viewing ray through (x, y) for each view passes through o. While any choice of P that passes through o will suffice, choosing P to be parallel to the image plane of S results in well-behaved warps (Figure 6.12). When an orbit axis, rather than a single point, is to be stabilized, we define the normal to P to be the projection of the viewing direction of S onto the plane orthogonal to the axis. I call this form of image stabilization *orbit stabilization*.

Orbit stabilization performs a similar function to software anti-shake methods that reduce jitter in video. However, it has the advantage of performing a *globally-consistent* stabilization, producing the effect of rotation about a single center, and avoiding the drift problems that can occur with frame-to-frame video stabilization methods. Note also that orbit stabilization does not require any

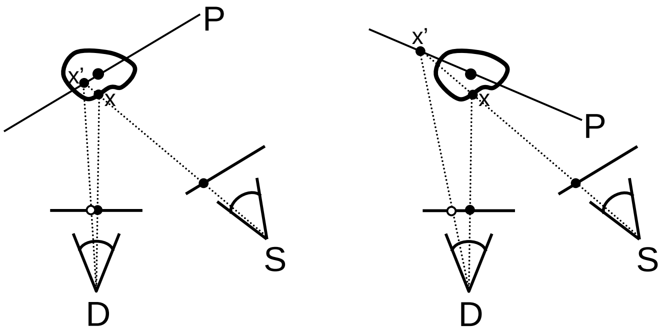


Figure 6.12: *Orientation of proxy planes.* Left: if P is parallel to S's image plane, a point x near the orbit point will get mapped to a nearby point x' on P, causing a small error in D. Right: an oblique choice of P will generally result in larger errors.

knowledge of scene geometry, although it does require known camera viewpoints and a reasonable orbit point.

My system defaults to using best-fit planar proxies, until an orbit point is selected, at which point it switches to orbit stabilization. The user can also opt to use best-fit planes even when an orbit point is selected, which can produce better results if the scene is truly planar or nearly planar.

Warping with a 3D model. When a 3D scene model is available, it can be used in place of the planar proxy to further improve rendering quality. Such models can be obtained, e.g., through laser scanning [85], multi-view stereo [55], or manual modeling. To render an image with a 3D proxy, I project the source image onto the proxy and then into the destination image, and place an additional plane in back of the model to account for unmodeled geometry.

Using a 3D model for rendering usually results in a more realistic experience, but, as in the Photo Tourism system, can also result in artifacts due to holes in the model or from projecting foreground objects onto the geometry. In my experience, planar proxies tend to produce less objectionable artifacts in these situations. The companion video shows a comparison of orbiting around the central portal of the Notre Dame Cathedral using planar and 3D proxies. 3D proxies work well in this example, and cause the 3D shape of the portal to be more pronounced. The artifacts resulting from 3D proxies are less noticeable here, as the façade of the Cathedral is roughly planar, hence projecting foreground objects onto the proxy is less objectionable. On the other hand, for the Statue of Liberty

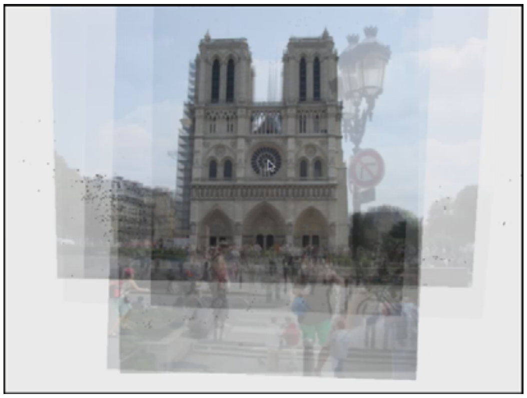


Figure 6.13: *Screenshot from the* **Notre Dame** *collection.* This screenshot from an interactive Pathfinder session exploring the **Notre Dame** collection shows multiple images blended together as the user moves around the scene.

collection, these artifacts are more pronounced, and hence the 3D proxies work less well for that data set.

Compositing the scene. The output image shown on the screen is rendered by first drawing a background layer consisting of the reconstructed point cloud drawn on top of a solid color, then rendering the currently selected image, projected onto its proxy geometry. Rather than instantaneously switching between images as new ones are selected for display, images are faded in and out over time. The system maintains an alpha value for each image; whenever a new image is selected for display, the alpha of the previous image decays to zero, and the alpha of the new image rises to one, over a user-specified interval of time. When the user is moving on a planned path, the system can look ahead on the path and fade images in early, so that each image reaches full opacity when it becomes the optimal image. When the user moves on a free-form path, this prediction is more difficult, and the system instead starts fading in an image at the moment it becomes optimal.

If the user is moving fairly quickly, multiple images can simultaneously have non-zero alphas. I blend the images by first normalizing all alphas to sum to one, then compositing the rendered images in an off-screen buffer. This image layer is then composited onto the background layer. An example screenshot showing multiple blended images is shown in Figure 6.13.

Once the photos are cached in memory, the Pathfinder viewer runs at over 30 frames per second with up to 1700 photos on a machine with a 3.4GHz Intel Xeon processor and an nVidia Quadro FX 4000 graphics card. A companion video showing the viewer being used to explore several scenes can be found on the project website [107].

6.6 Path planning

As described in the previous section, the Pathfinder viewer allows a user to select among the set of detected controls and representative views using the thumbnail pane, and takes the user along an automated path to a selected control. Unlike in Photo Tourism [137], which performs two-image morphs, the Pathfinder system can create transitions involving multiple images, making it much more effective for moving along long, complex paths. These multi-image transitions are created using a new path planning approach. The system attempts to find smooth paths between two images that are always close to good views, so that at any point on the path the user can be presented with a high-quality rendering of the scene. An additional benefit of constraining the path to pass near photos in the database is that it will more closely emulate how a human would move through the scene, as it will stay close to places where people actually stood and took photos. Such paths are more likely to be physically plausible, e.g., to not pass through walls or other obstacles.

Path planning, often used in robotics, has also found application in computer graphics for computing camera paths through 3D environments. For instance, Drucker and Zeltzer [38] use planning to help create paths through a 3D scene which satisfy task-based objectives (such as focusing on a specific object) and geometric constraints. In the realm of IBR, Kang *et al.*[76] analyze a sequence of images to predict which views or portions of views can be synthesized. In my work, I extend these ideas to use our view quality prediction score to plan good camera paths.

In order to find the best path between two views given our database of existing images $\mathcal{I}$, suppose we have a viewpoint cost function $\mathrm{Cost}_{\mathcal{I}}(v)$ (to be defined shortly) over the space of possible viewpoints, where $\mathrm{Cost}_{\mathcal{I}}$ is low for viewpoints close to existing image samples, and large for more distant views. The optimal path between two viewpoints can then be defined as the lowest-cost path (geodesic) connecting them.

The dimension of the viewpoint space, however, is relatively high (five degrees of freedom for camera pose, and six if zoom is included), and therefore this continuous approach is computationally expensive. I instead find a discrete solution by first computing an optimal piecewise linear path through the set of existing cameras, then smoothing this path. This discrete problem can be posed as finding a shortest path in a transition graph G_T whose vertices are the camera samples $\mathcal{I}$ (G_T is a weighted version of the image connectivity graph presented in Chapter 3).

G_T contains a weighted edge between every pair of images I_j and I_k that see common 3D points. The weight w_{jk} on an edge (I_j, I_k) is the predicted cost of a transition, computed based on how well the transition between the two images can be rendered. We denote this transition cost as τ_{jk}, and define τ_{jk} to be the integral of the viewpoint cost function over a straight-line path $\gamma_{jk}(t)$ between I_j and I_k.[7] Because edge (I_j, I_k) represents a two-image transition, when computing τ_{jk}, I restrict the rendering process to consider only I_j and I_k when generating in-between views along the path $\gamma_{jk}(t)$; this restricted viewpoint cost function is denoted $\mathrm{Cost}_{jk}(v)$. Thus,

$$\tau_{jk} = \int_0^1 \mathrm{Cost}_{jk}(\gamma_{jk}(t))\, dt. \tag{6.13}$$

I define Cost_{jk} by first considering the cost of rendering a new viewpoint with one of the images samples, I_j. In Section 6.3 I defined a reprojection score $S_{\mathrm{proj}}(I_j, v)$ for estimating how well an image I_j can be used to synthesize a new view v. I now turn this scoring function into a cost function:

$$\mathrm{Cost}_j(v) = e^{\alpha(1 - S_{\mathrm{proj}}(I_j, v))} - 1. \tag{6.14}$$

This function evaluates to 0 when $S_{\mathrm{proj}}(I_j, v) = 1$, and to $e^{\alpha} - 1$ when $S_{\mathrm{proj}} = 0$ (I use a value $\alpha = 8$). I now define the two-view cost function Cost_{jk} over the path $\gamma_{jk}(t)$ as the weighted sum of the single viewpoint cost function:

$$\mathrm{Cost}_{jk}(\gamma_{jk}(t)) = (1 - t)\,\mathrm{Cost}_j(\gamma_{jk}(t)) + t\,\mathrm{Cost}_k(\gamma_{jk}(t)). \tag{6.15}$$

I approximate the integral in Eq. 6.13 by computing the sum of Cost_{jk} at 30 samples along $\gamma_{jk}(t)$.

If edges are weighted using the transition cost alone, shortest paths in the graph may not correspond to smooth paths through viewpoint space; indeed, I have observed that such paths can zigzag

[7] A straight-line path is obtained by linearly interpolating the camera position and quaternions representing each camera orientation.

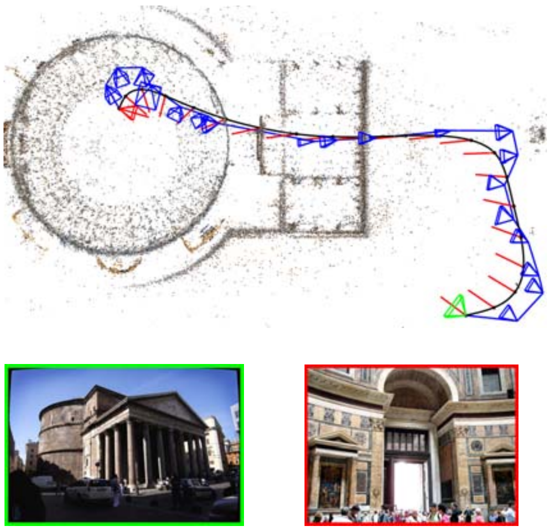


Figure 6.14: *Using path planning to compute a camera transition.* A transition from an image outside the Pantheon (green) to an image inside (red) computed using my path planning algorithm. The blue cameras are the intermediate nodes visited on the transition graph, and the blue line is the linearly interpolated path. The black curve shows the path resulting from smoothing this initial discrete path, and the red lines indicate the viewing directions at samples along this path.

in both position and orientation in a disorienting way. To penalize such paths, I add a smoothness cost σ_{jk} to each edge weight w_{jk}. This cost is simply the length of the edge in viewpoint space, which I compute as a weighted combination of the difference in position and orientation between cameras C_j and C_k:

$$\sigma_{jk} = \|\mathbf{c}_j - \mathbf{c}_k\| + \beta \operatorname{angle}(\mathbf{v}(C_j), \mathbf{v}(C_k)), \tag{6.16}$$

where $\mathbf{c}$ is the 3D position of camera C and $\mathbf{v}(C)$ is its viewing direction. I have found a value $\beta = 3.0$ to work well in practice.

The final weight of an edge (I_j, I_k) is the weighted sum of the transition cost τ_{jk} and the smoothness cost σ_{jk}:

$$w_{jk} = \tau_{jk} + \lambda \sigma_{jk}. \tag{6.17}$$

For my experiments, I use a value of $\lambda = 400$.

Generating smooth paths. To generate a path between two images I_{start} and I_{end}, I first use Djikstra's algorithm [35] to compute a shortest path π^* between I_{start} and I_{end} in G_T. π^* can be interpreted as a piecewise linear physical path through viewpoint space; this path is next smoothed to produce a more continuous path for animating the camera.

To smooth π^*, I first uniformly sample π^* to produce a sequence of viewpoint samples $v_i^0, i = 1$ to n (I use $n = 100$ samples in my implementation). I then repeatedly average each sample with its two neighbors, and with its original position v_i^0 (in order to keep the sample close to the initial path π^*):

$$v_i^{t+1} = \frac{1}{1+\mu}\left(0.5\left(v_{i-1}^t + v_{i+1}^t\right) + \mu v_i^0\right). \tag{6.18}$$

I apply 50 iterations of smoothing.[8] The parameter μ controls how closely the final path matches π^*, versus how smooth the path is. In my implementation I set $\mu = 0.02$, which produces nice, smooth paths which still generally stay close enough to the images along π^* to produce good views. An example of a path computed between two images in the Pantheon collection is shown in Figure 6.14.

The final path π_{final} between I_{start} and I_{end} is formed by linearly interpolating the final viewpoint samples v_i. To animate the camera, I sample π_{final} non-uniformly, easing in and out of the transition. The user need not be situated exactly at the position of I_{start} when the transition is started; I adjust the initial path π^* to begin at the user's current position before smoothing. If the user is not visiting any image, however (i.e., the user is at a viewpoint v where $S_{\text{view}}(v) < \epsilon$), then my system currently uses a simple linear path to move to I_{end}. This limitation could be addressed by adding a virtual node to G_T corresponding to the user's current viewpoint, adding edges between this node and nearby images, then weighting these edges with an appropriate cost function.

6.7 Appearance stabilization

Unstructured photo sets can exhibit a wide range of lighting and appearance variation, including night and day, sunny and cloudy days, and photos taken with different exposures. By default, Pathfinder displays photos based only on the user's current viewpoint, which can result in large changes in scene appearance as the user moves. These large, random variations can be useful in

[8] This smoothing operation is equivalent to applying a Gaussian filter to the path.

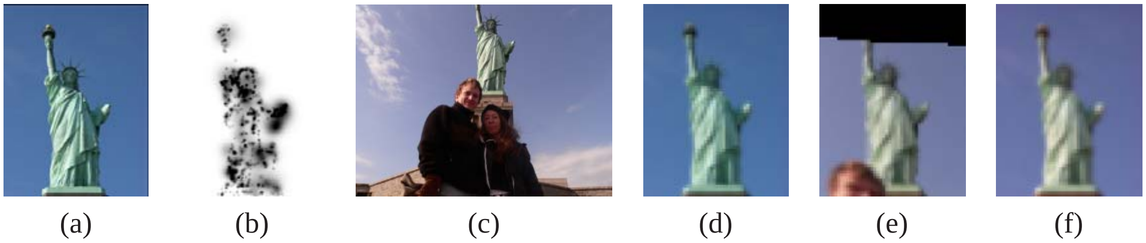


Figure 6.15: *Computing image similarity.* Image (a) with weight map (b); higher weights are darker. To compare (a) to another image (c), I first downsample (a) to a height of 64 pixels (d) and resize the second image to match the scale of the first. I then geometrically warp the second image to be in alignment with the first (e), and apply color compensation to the first image (f). The distance between the images is then computed as the weighted L_2 distance between the RGB pixel values of (e) and (f).

getting a sense of the variation in appearance space of a scene, but they can also be visually distracting. To reduce this appearance variation, the user can enable a *visual similarity* mode, which limits transitions to visually similar photos, and a color compensation feature, which hides appearance changes by modifying the color balance of new images to better match that of previous images. In addition, our system allows the photo collection to be separated into classes, such as day and night, and allows the user the option of restricting selected photos to a particular class.

This section describes these features in more detail. I first describe a metric used to compute photo similarity, then describe how this metric is incorporated into the viewer and how color compensation is done. Finally, I describe how an object can be browsed in different appearance states.

6.7.1 Computing image distances

To reduce the amount of appearance variation that occurs while viewing a scene, I first need a way to measure the visual distance between two images. To compute this distance, I register the images geometrically and photometrically, then compute the L_2 distance between corresponding pixel values, weighted by confidence in the registration. These steps are summarized in Figure 6.15.

Geometric alignment. To compute the distance between images I_j and I_k, I first downsample I_j to a resolution of 64×64 pixels, and downsample I_k to approximately the same sampling rate with respect to the scene, resulting in low-resolution images I'_j and I'_k.

Next, I warp I'_k into geometric alignment with I'_j. If the complete scene geometry is known, it can be used produce the warped version of I'_k, but since I do not assume geometry is available, I instead use the sparse data interpolation method of thin-plate splines (TPS) [12] to model a non-rigid 2D warp between images.[9] Given a set of 2D image correspondences $(x_i, y_i) \rightarrow (x'_i, y'_i), i = 1$ to k, a thin plate spline models the image warp $D(x, y) = (D_x(x, y), D_y(x, y))$ separately for each dimension using a combination of an affine transformation and radial basis functions:

$$D_x(x, y) = a_1 + a_x x + a_y y + \sum_{i=1}^{k} c_i \phi(||(x_i, y_i) - (x, y)||) \tag{6.19}$$

$$D_y(x, y) = b_1 + b_x x + b_y y + \sum_{i=1}^{k} d_i \phi(||(x_i, y_i) - (x, y)||) \tag{6.20}$$

where the kernel function $\phi(r) = r^2 \log(r)$ and the coefficients a, b, c, and d are the parameters of the model. These parameters can be solved for in closed form.

To compute the warp, I project all 3D points visible to I_j into both I'_j and I'_k to form a set of 2D basis points, and compute the corresponding TPS deformation D mapping I'_j onto I'_k. Computing the TPS parameters involves solving a large, dense linear system $\mathbf{A}x = \mathbf{b}$. In my implementation, I use an LU decomposition to solve the system; because $\mathbf{A}$ only depends on the projections in I'_j, I pre-factorize $\mathbf{A}$ into $\mathbf{L}$ and $\mathbf{U}$ in order to quickly solve for the TPS parameters each time an image is registered with I_j.

Given the deformation D, for each pixel location $\mathbf{x} = (x, y)$ of I'_j, I compute the corresponding pixel location $D(\mathbf{x})$ of I'_k. If $D(\mathbf{x})$ lies inside I'_k, I'_k is sampled at $D(\mathbf{x})$ using bilinear interpolation. This results in a sequence of pairs of RGB samples:

$$[I'_j(\mathbf{x}_1), I'_k(D(\mathbf{x}_1))], [I'_j(\mathbf{x}_2), I'_k(D(\mathbf{x}_2))], \ldots, [I'_j(\mathbf{x}_n), I'_k(D(\mathbf{x}_n))]$$

Figure 6.15 (e) shows an example of one image warped into another using this technique.

The TPS deformation D will not necessarily extrapolate well far away from the basis points; there is more confidence in the deformation near known 3D point projections. To make the image

[9] Planar proxies could also be used to warp the images into alignment, but I have found that, while planar proxies often create reasonable visual results, they are not always accurate enough for the purpose of registering images for computing image differences.

comparison more robust to misregistration, I precompute a spatial confidence map W_j for each image I_j. W_j is created by centering a 2D Gaussian at the projection of each 3D point observed by I_j, with standard deviation proportional to the scale of the SIFT feature corresponding to that point, and with height $\frac{1}{2}$. The Gaussians are then summed, sampled at each pixel, and clamped to the range $[0, 1]$. Figure 6.15 (b) shows an example confidence map.

When registering image I'_k to image I'_j, I also create a confidence map for the registered result. To do so, I downsample the weight maps W_j and W_k in the same way as the images, producing low-resolution weight maps W'_j and W'_k, then create a weight for each of the RGB sample pairs $[I'_j(\mathbf{x}), I'_k(D(\mathbf{x}))]$ by taking minimum of the two confidence values corresponding to these samples:

$$w_i = \min\{W'_j(\mathbf{p}_i), W'_k(D(\mathbf{p}_i))\} \tag{6.21}$$

giving a sequence of weights, $w_1, w_2, \ldots, w_n$.

Photometric alignment. After applying a spatial warp to I'_k, I next align the color spaces of the two images. I use a simple gain and offset model to warp each RGB color channel of I_k to match that of I_j:

$$\mathbf{M}_{kj} = \begin{bmatrix} s_r & 0 & 0 & t_r \\ 0 & s_g & 0 & t_g \\ 0 & 0 & s_b & t_b \end{bmatrix}$$

This affine transform can compensate linear changes in brightness, due, for instance, to changes in exposure time. To compute the gain and offset parameters s and t for each color channel, I fit a line to the pairs of color samples; I use RANSAC [45] during the fitting to achieve robustness to bad samples due to misaligned or saturated pixels [45]. Figures 6.15 (e) and 6.15 (f) show a pair of geometrically and photometrically warped images.

Finally, I compute the distance measure $\mathrm{d}(I_j, I_k)$ as the weighted average of color-shifted RGB samples:

$$\mathrm{d}(I_j, I_k) = \frac{1}{\sum w_k} \sum_{i=1}^{n} w_i \left\| I'_j(\mathbf{x}_i) - \mathbf{M}_{kj} I'_k(D(\mathbf{x}_i)) \right\| \tag{6.22}$$

using RGB values in the range $[0, 255]$.

Because it is difficult to reliably register photos with wide baselines, I only compute image distances between pairs of photos that are relatively close to each other. In the Pathfinder viewer, it is still possible to move a large distance when similarity mode is enabled, via a sequence of transitions between nearby, similar images.

6.7.2 Browsing with similarity

The Pathfinder system reads the pre-computed similarity scores at startup. When the visual similarity mode is enabled, these scores are used to prune the set of possible image transitions and to favor transitions between images that are more similar. This is implemented by multiplying the reprojection score $S_{\text{proj}}(I, v)$ with a similarity factor $S_{\text{sim}}(I, I_{\text{curr}})$, where I_{curr} is the currently displayed image. To compute $S_{\text{sim}}(I, I_{\text{curr}})$, I remap the interval $[\mathrm{d}_{\min}, \mathrm{d}_{\max}]$ to $[0, 1]$ and clamp:

$$S_{\text{sim}}(I, I_{\text{curr}}) = 1 - \text{clamp}\left(\frac{\mathrm{d}(I, I_{\text{curr}}) - \mathrm{d}_{\min}}{\mathrm{d}_{\max} - \mathrm{d}_{\min}}, 0, 1\right). \tag{6.23}$$

I use values of $\mathrm{d}_{\min} = 12$ and $\mathrm{d}_{\max} = 30$. Enabling similarity mode results in the selection of a sparser set of photos, so though their visual appearance is much more stable, the motion can be less continuous.

6.7.3 Color compensation

The viewer can also use the pairwise RGB gain and offset parameters, estimated while computing similarity scores, to disguise changes in appearance by adjusting the color balance of a new image to match that of the previous image. When color compensation is enabled, the viewer maintains a 3x4 color compensation matrix $\mathbf{M}_j$ for each image I_j, which is applied when an image is rendered. During a transition from an image I_j to an image I_k, $\mathbf{M}_k$ is set to $\mathbf{M}_{kj}$, pre-multipled by the color compensation already in effect for I_j,

$$\mathbf{M}_k = \mathbf{M}_j \cdot \mathbf{M}_{kj}.$$

Examples of color corrected images are shown in Figure 6.16. To reduce problems with accumulated drift over time and eventually return images to their true color balance, the matrices $\mathbf{M}_j$ fade back

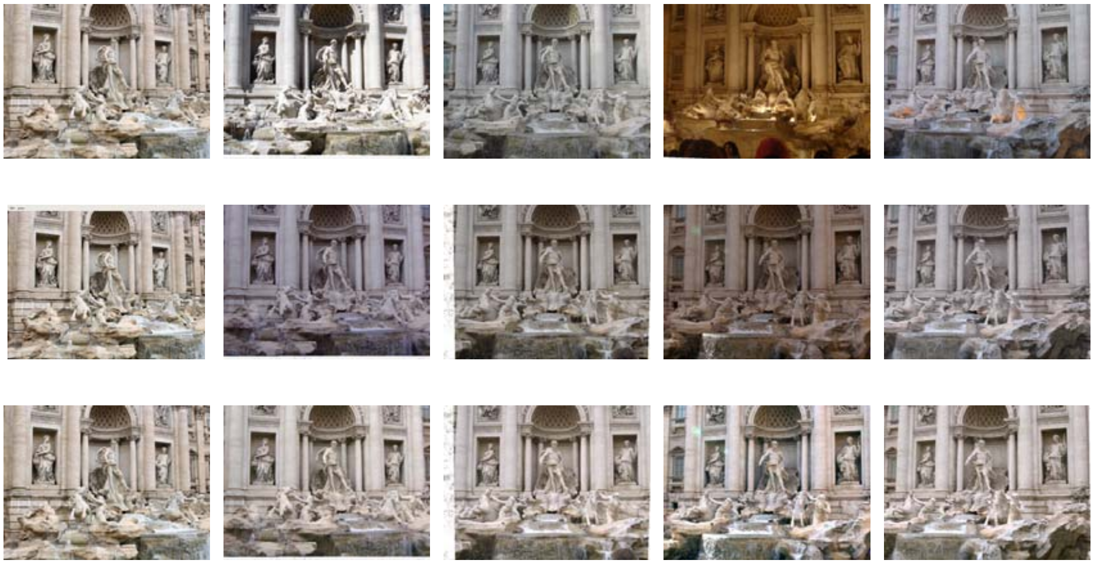


Figure 6.16: *Similarity mode and color compensation.* The first row shows a sequence of images, going from left to right, from an object movie of the Trevi Fountain resulting from orbiting the site. The second row shows the result of orbiting through the same path with similarity mode turned on. Note that a different set of images with more similar lighting is selected. The third row shows the same images as in the second row, but with color compensation turned on. The color balance of each image now better matches that of the first.

to the identity transform $[I|0]$ over time.

6.7.4 Viewing different appearance states

As with QuickTime VR object movies, my system allows an object to be viewed in different appearance states, such as day and night. This feature requires the photos to be classified into sets corresponding to each state; once the photos are classified and a user selects a certain state, the system will only display photos from that state. At a given viewpoint v, the user can toggle to any state for which a photo I_j with non-zero reprojection score $S_{\text{proj}}(I, v)$ exists.

I found that for two particular classes of photos, night and day, could be semi-automatically classified using the observation that many 3D points (corresponding to SIFT features in the original images) are highly correlated with either daytime or nighttime images; some features appear only in daytime images, and some only in nighttime images. By hand-labeling a small number daytime and seven nighttime photos, labels can be automatically propagated to all other photos.

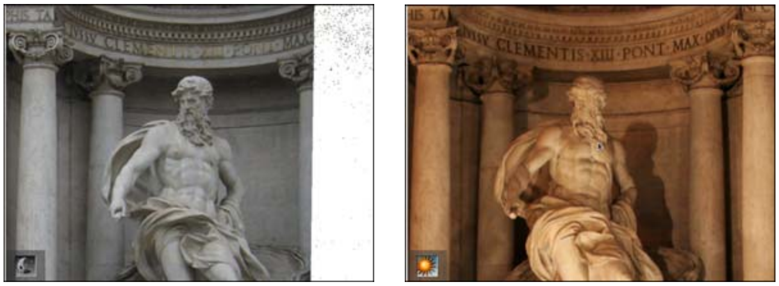


Figure 6.17: *Switching between day and night in the Trevi Fountain.* Left: a daytime view of the Trevi Fountain. Right: the result of switching to a nighttime view; the same viewpoint is shown at a different time of day.

I In particular, the labeling algorithm iteratively updates a set of (continuous) image labels $U(I)$ and point labels $V(I) \in [-1, 1]$, where -1 corresponds to a nighttime image and 1 to a daytime image. The image labels are first initialized to $U(I) = 1$ for the manually labelled daytime images, $U(I) = -1$ for the nighttime images, and $U(I) = 0$ for other images, and the point labels were initialized to $V(p) = 0$. Next, the point and image labels are updated using the following update equations:

$$V(p) = \frac{\sum_{I \in \mathrm{Imgs}(p)} U(I)}{\sum_{I \in \mathrm{Imgs}(p)} |U(I)|}, U(I) = \frac{1}{|\mathrm{Points}(I)|} \sum_{p \in \mathrm{Points}(I)} V(I),$$

where $\mathrm{Points}(I)$ is the set of points visible in image I, and $\mathrm{Imgs}(p)$ is the set of images in which point p is visible. In other words, points that are seen in mostly "night" (resp. "day") images are labeled as "night" (resp. "day") points, and vice versa. For the cases I tried (the Trevi and Notre Dame sets described in the next section), the update steps converged after about five iterations, and cleanly separated the images into daytime ($U(I) > 0$) and nighttime ($U(I) < 0$) sets.

6.8 Results

I have applied the Pathfinder system to several large collections of images downloaded from Flickr, as well as a collection of images taken from a hand-held camera. Screenshots of the interactive viewer are shown in figures throughout this chapter (Figures 6.5, 6.10, 6.8, 6.9, and 6.13 and the figures in this section); however, the results are best viewed in the video on the project web site [107]. Two of these scenes consist of dominant objects and provide an object movie experience: the Statue of Liberty, created from 388 photos, and the Venus de Milo, created from 461 images. The system detected two orbits for the Statue of Liberty, and one orbit for the Venus de Milo; these orbits are shown in Figure 6.10. The reconstruction of the Notre Dame Cathedral (created from 597 photos) has a wide distribution of camera viewpoints on the square in front of the Cathedral, and is therefore well suited for free-form 6-DOF navigation; a screenshot of the viewer with this data set loaded is shown in Figure 6.13. This is a case where automatic orbit detection is less useful, as a good orbit can be produced from almost anywhere on the square, as shown in the video. The reconstruction of the Trevi Fountain (1771 photos) contains large numbers of both day and nighttime images, making this a good candidate for evaluating both appearance stabilization and state-based modes. Figure 6.17 shows an example of switching between day and night mode.

I also demonstrate how the system makes it easy to create an object movie experience by manually rotating a hand-held object (a Mark Twain figurine) in front of a camera. In this case, the user manually specified a sphere of orbits, as the current implementation does not support spherical orbit detection; screenshots are shown in Figure 6.9.

Finally, I demonstrate the system with a collection of photos of the Pantheon (602 images), a complex scene consisting of both interior and exterior views. For this scene, Pathfinder detected three orbits (Figure 6.5), several panoramas (Figures 6.6 and 6.8), and a number of canonical images (Figure 6.7), including photos of the front facade, the altar, the oculus, and several sculptures inside the building. The accompanying video shows sample interactions with each of these types of controls, and demonstrates the results of my path planning approach.

I also use this reconstruction to demonstrate another application of the Pathfinder system: creating a 3D slideshow of a personal photo collection. Suppose that you visit the Pantheon and take your own set of photos, showing friends and family in a few discrete locations (e.g., the front façade, the altar, looking up at the oculus). Structure from motion techniques are not likely to be able to register the photos together due to insufficient overlap, unless they are captured much more frequently than is customary with personal collections. However, if we combine them with all the other photos of the Pantheon on the Internet, we can register our personal photos with that collection, and plan paths between our own photos to create a 3D slideshow (in a sense, retracing our steps through the Pantheon). The companion video shows such a slideshow created from a collection of four personal photos added to the 602-image reconstruction. Figure 6.18 shows several photos in this personal

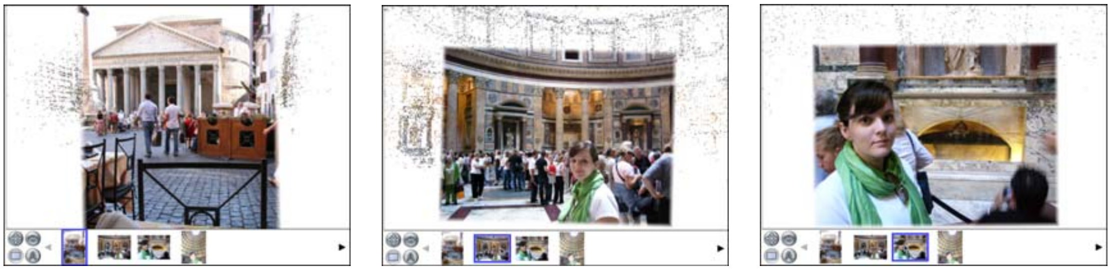


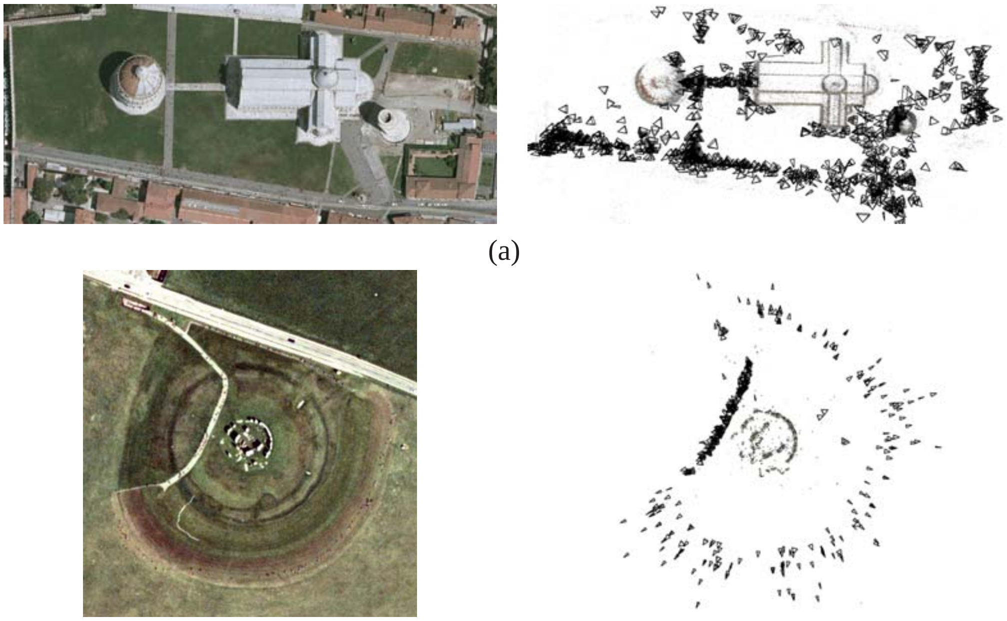
Figure 6.18: *Personal photo tour of the Pantheon.* In this sequence, a user has added their own personal collection to the Pantheon data set. This allows the user to create a 3D slideshow through their own photos, using the community's photos to fill in the gaps (please see the video on the project webpage for an animated version of this tour).

collection in context in the Pathfinder viewer.

6.9 Discussion

I have successfully used my approach to create fluid IBR experiences with scene-specific controls from unstructured community photo collections. I believe that these techniques represent an important step towards leveraging the massive amounts of imagery available both online and in personal photo collections in order to create compelling 3D experiences of our world. Part of the power of this approach is the ability to learn controls from and create renderings of the world from large photo collections. As these techniques are applied to larger and larger collections, I believe that the experiences generated will continue to improve.

However, my approach also has several limitations in both the navigation and rendering components. My geometric model for orbits is a circle, which fits many real-world scenes. On the other hand, many paths around objects are ellipses, lines and polygons, or more free-form shapes. For instance, consider the reconstructions of the Pisa Duomo and Stonehenge discussed in Chapter 4 and reproduced above in Figure 6.19. The paths people take around these objects follow walkways or (in the case of Stonehenge) fences, resulting in more complex, non-circular paths. In the future, it would be interesting to explore the detection of more general types of paths in a scene, perhaps by unifying the path planning algorithm with the orbit and panorama detection algorithms. An additional challenge is to devise better rendering algorithms for these more general paths, as orbit stabilization will not always be applicable.



(a)

(b)

Figure 6.19: *Free-form paths around the Pisa Duomo and Stonehenge.* Not all paths through scenes are circular. (a) Photos taken of the Pisa Duomo and its surroundings tend to follow walkways around the building. (b) Photos of Stonehenge follow an arc going about two-thirds of the way around the prehistoric monument, but then curve inwards to follow a straight section of path for the remainder of the circle.

Another interesting question brought up by this point is whether the paths a user *wants* to take through a scene are the same as the ones people *actually* take when physically at the scene. The paths discovered by my system reflect the latter, which results in controls that are arguably useful in two senses: (1) they show you what you would presumably see if you were there and (2) inasmuch as people take photos from “optimal” viewpoints and of interesting views, the derived controls are likely to be interesting as well. However, people are also constrained by physics, walls, and other physical barriers, which preclude viewpoints that might, in some sense, be better. For instance, the fences around Stonehenge prevent people from getting close to and climbing around the stones, but it would certainly be useful to have photos taken from such viewpoints (in fact, these views might be much more useful in a virtual setting, where there is no possibility of damage to the stones). One possible way around this problem might be to combine the controls learned from the masses with more carefully created paths captured by experts with special access or equipment, or to have the masses vote on interesting paths through the scene with a distributed 3D drawing interface.

An expert creating a specialized path through a scene is an example of the more general concept of authoring. Currently, the system discovers sets of controls and computes paths between views completely automatically and without user input (the user can specify a set of endpoints to paths, however, as in the 3D slideshow shown in Section 6.8). In some cases, a user might want to manually specify paths, either to correct a mistake made by the automatic algorithm, highlight additional interesting paths, or create a personalized tour through the scene. For instance, one could envision a kind of “magnetic lasso” tool [96], where a user draws a path that automatically snaps to nearby photos so as to improve the viewpoint score of the specified path.

Appearance compensation is also still a significant open problem. My color compensation method works well for images that are fairly similar in overall appearance, so it goes hand in hand with the similarity-based selection mode. However, because the method only models an affine transformation per color channel, compensating two very different images (e.g., sunny and cloudy) is not possible, limiting the number of possible transitions between images. Developing a more flexible appearance compensation model would help avoid these problems. It also would be interesting to explore more sophisticated image models that detect foreground objects, such as people, and background elements such as the sky. These elements could then be treated specially during rendering: foreground objects could be removed during transitions, or popped up onto their own proxy planes, and the sky could transition in a realistic way, with clouds rolling in or out.

For larger and larger scenes, memory will become a significant issue. Currently, the system caches all images when the program is loaded: this not only introduces a significant startup time, but will not scale to collections of tens or hundreds of thousands of images. Handling truly massive collections will require more efficient caching schemes, such as those used in Seadragon [127] and Photosynth [109].

How will the controls themselves scale to much larger scenes? For a city-sized scene, will the user want the same set of controls as for a small-scale scene? I imagine that the toolset for moving through the scene would probably consist of controls at different levels of detail, depending on whether the user wants to get an overview of the entire city or a particular momument. Paths between different parts of a city could be computed in a variety of ways as well, depending on the user’s intent. A Google-Earth style transition (zooming out to city-level, then zooming back in to the destination) would provide a broad context of the scene, while a “taxi”-style transition wending through the city streets would show how a person might actually move between two parts of the city.

Evaluation. Finally, evaluation is a very important area for further study. How effective are the derived scene-specific controls, and the Pathfinder user interface as a whole, at enabling users to navigate through and understand a scene? Part of the challenge in evaluating 3D navigation techniques is deciding what questions to ask—what is it that we want to evaluate? Gauging the effectiveness of an interface presupposes a certain task that users are trying to accomplish, as well as a certain metric for success, such as speed or accuracy. For instance, in Ware and Osbourne’s study comparing different navigation metaphors [157], the task is to locate several details distributed throughout the scene (and afterwards make a movie showing where each detail is), and the evaluation metric was how satisfied the user was with each metaphor. Other work has considered the efficiency of completing tasks such as moving the viewpoint to a specific object [145] (somewhat analogous to the 2D desktop task of moving a mouse cursor to a destination).

In practice, however, it can be hard to predict what task a user will wants to accomplish in a scene, or the task itself may be ill-defined. In the context of this chapter, which has focused mainly on famous, real-world scenes, the most natural task is perhaps “virtual tourism”—walking around a scene doing what one might do if physically visiting a famous place for the first time, such as looking at interesting objects and learning interesting facts about the location. These kind of tasks seem less straightforward to evaluate, as they are more about entertainment and education, or, perhaps, simply about the experience of being at an impressive or important place. For these kinds of tasks, a more informal type of study evaluating how engaged users are in the interface, and how long they remained interested, might be more appropriate. It would also be interesting to study how people move through and interact with real tourist environments in order to create improved interfaces and experiences. The work presented here uses an artifact of those interactions—people’s photographs—to reverse-engineer patterns of movement and find interesting objects, but studying behavior more directly could also be fruitful.

In addition to virtual tourism, there are also other scenarios that would be interesting to evaluate. One example is real estate, where an owner might want to create a virtual tour of their home for potential buyers to explore. From a buyer’s point of view, the goals of exploring a home might be more specific than with tourist sites, and could include seeing what individual rooms look like, how big they are, and how they are spatially arranged. One way to evaluate a virtual home interface would be to measure how well (i.e., how accurately, and how quickly) a user gains spatial knowledge of a home when using the interface. This could be tested in a study in which users explore the home virtually, then try to sketch out a floorplan. This approach was used by Darken and Seibert to evaluate how well different navigational aids (e.g., an overhead map or a grid) improve wayfinding ability in large virtual environments [27]. Other measures of spatial knowledge have also been explored. For instance, Ruddle *et al.*[121] use three different metrics in the context of navigating virtual buildings: route-finding ability, relative distance between landmarks, and directional estimates, i.e., how accurately users could "point" from one location to another after exploring a scene.

Individual components of the Pathfinder system could also be evaluated. For instance, the effectiveness of the scene-specific controls could be evaluated in comparison to other types of controls (e.g., different styles of free-viewpoint controls). The trajectories resulting from the path planning algorithm could also be compared to other types of paths, such as linear interpolation, in terms of scene and motion comprehension or visual appeal.

Chapter 7
CONCLUSION

This thesis has presented my work on taking large, unstructured photo collections and creating immersive 3D experiences of places, using combinations of new computer vision, graphics, and interaction techniques. This work makes the following specific contributions to computer vision and computer graphics:

- **Computer vision:**
  - **A structure from motion (SfM) pipeline**, described in Chapter 3, that combines many existing techniques (SIFT [90], RANSAC [45], sparse bundle adjustment [89]), into a robust system. With this system, I have demonstrated, for the first time, that SfM can be successfully applied to the large, unstructured, highly diverse collections of images found with Internet search.
  - **An approach for handling the scale of Internet collections.** In Chapter 4, I described *skeletal sets*, an approach for sifting through a large image collection to find the subset that is critical for reconstruction. This technique can significantly reduce the number of images that need to be processed with the heavy machinery of SfM, reducing the SfM processing time by an order of magnitude for large collections, while attaining provable bounds on the loss in accuracy of reconstruction.
- **Computer graphics and interactive techniques:**
  - **A new 3D photo browser.** In Chapter 5, I presented Photo Tourism, a photo browser that takes photo collections reconstructed with SfM and places the user in the 3D scene among the reconstructed photos. Photo Tourism provides new geometric photo browsing controls that enable actions such as zooming in and out of a photo, moving to photos taken to the left or right of a given image, finding good photos of a selected object, and

viewing stabilized slideshows. Photo Tourism also contains a powerful annotation tool, which can be used to label large numbers of photos at a time.

– **A new 3D navigation system** that automatically derives good navigation controls from an input photo collection. Chapter 6 described Pathfinder, a scene browser that analyzes the distribution of photos in a collection, detects interesting orbits, panoramas, and canonical views, and synthesizes optimal paths between different controls using a new path planning algorithm. Pathfinder tailors the navigation controls to different scenes in order to make exploring a particular scene more intuitive and automatic.

– **New rendering techniques for depicting scenes from unstructured photo collections**, including new ways to render transitions between images and to render scenes in a non-photorealistic style (Chapter 5), and an approach for selecting and displaying images automatically based on a user's current viewpoint and navigation controls (Chapter 6).

7.1 Future work

In this thesis, I have presented work that demonstrates the tremendous potential of using massive photo collections to create new ways to visualize the world. However, this work is just a first step towards a goal of creating a virtual version of our world that is as visually rich and as extensive as the real thing, putting a vivid recreation of the world at our fingertips. I envision a system that stores all of the images of places ever taken and can display or synthesize a photo-realistic view from nearly anywhere in the world, at any time, and under any possible weather condition. In this section, I describe several specific projects that reach toward this ambitious goal.

Reconstructing Rome. Scale is still a significant problem in structure from motion problems. To date, the largest collections I have reconstructed consist of about 10,000 photos. However, for many large-scale scenes, we can easily find collections of photos that are two or three orders of magnitude larger. For instance, searching for "Rome" on Flickr returns over 1.5 million photos. If we add the Italian name for Rome, *Roma*, we get an addition half million photos. The same search in Picasa Web Albums results in nearly 18 millions photos. I have downloaded just over a million of these, and matched a subset of about 20,000 of the downloaded images. The image connectivity graph for

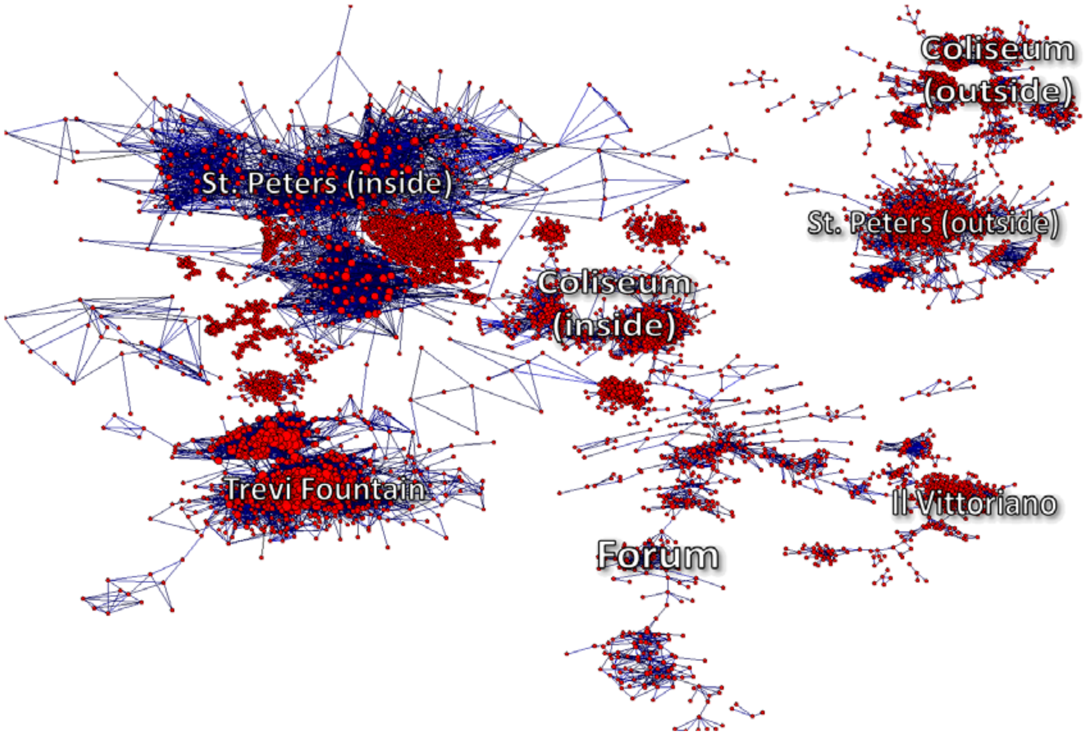


Figure 7.1: Image connectivity graph for 20,000 images of Rome from Flickr. The graph is made up of several large (and many smaller) connected components corresponding to individual sites, some of which are labelled.

this subset is shown in Figure 7.1. The goal is to reconstruct as much of the city of Rome as possible from these photos—to create a modern-day complement to the model of the ancient city created by the Rome Reborn project [62].

How can we reconstruct one million photos? Further, can we reconstruct all of them *in a single day*? This is a very challenging problem that will require new, more efficient algorithms, but also intelligent ways of parallelizing the reconstruction effort.

Generally speaking, there are two parts of the reconstruction pipeline that are particularly time-consuming: image matching and structure from motion. In the current image matching algorithm, every pair of images is compared; for one million photos, we need to consider just under half a trillion image pairs. At 0.1 second per pair, this process would take about a millenium and a half to complete on a single machine. The matching is easily parallelizable, but even with 1,000 machines, it would take more than 1.5 years (about 578 days) to complete. Linear-time algorithms based on a bag of words approach [133, 104, 24] are much more scalable, but may be less accurate than brute force matching. These approaches might be fruitfully combined together in stages. For instance, one could first run a linear-time algorithm to find likely pairs of matching images, followed by a brute force verifier on those pairs, followed by a stage to compress the matching information to remove redundancy, and finally another brute force step to find any missed matches.

For the structure from motion stage, the skeletal sets algorithm presented in Chapter 4 (combined with parallelization of the pairwise reconstructions) may be efficient enough to reconstruct each connected component of the match graph. If not, however, one way to make SfM even faster would be to parallelize the reconstruction of the skeletal graph by recursively breaking it apart, reconstructing individual pieces in parallel, and merging.

Robustness is another challenge with this large reconstruction effort. As noted in Chapter 3, the SfM pipeline has several failure modes that can result in incorrect results. For individual models, these failures can sometimes be fixed by manually tweaking parameters or by choosing a different initial image pair. However, hundreds of reconstructions may be present in the million images of Rome, and it would be tedious to check each one. Thus, improving the success rate of SfM is another key problem.

Capturing appearance and dynamic phenomena. In this thesis I have focused mainly on recovering camera and sparse scene geometry from photo collections and providing corresponding *geometric* controls. However, the appearance of a scene can change dramatically over time, from day to night, over the course of the year, and due to changes in the scene itself. In the Pathfinder system, I demonstrated techniques for normalizing appearance in order to maintain consistency across views, and for partitioning photos into broad categories, such as day and night. Even better would be controls for directly exploring the space of the appearance of a scene, in the same way that geometric controls let the user explore the space of viewpoints. Imagine, for instance, having a handle on the sun, being able to drag it across the sky, and having the appearance of the scene automatically adjust to reflect the selected time of day and year.

One approach to this problem would be to index each photo by viewpoint and time of capture, and display the photo that best matches the user's current viewpoint and selected time. However, the photo collections I have reconstructed are not yet dense enough—in both viewpoint and time—for this simple lookup scheme to work well. Moreover, the time and date information that is stamped on most digital photos can be unreliable, as photographers can easily forget to set their camera's clock or update the time zone when they travel.

If the time and date information could be reliably determined, another approach would be to learn an entire appearance model from scratch given the images. This would involve computing a full 3D model for the scene, learning material parameters (or bi-directional reflectance distribution functions (BRDFs)) for every surface, determining how the scene is lit, and using these recovered parameters to render the scene from new views and at new times. This is a challenging data-fitting problem, however, and would involve solving a complex, highly non-linear objective function; nevertheless, this would be a very interesting approach to try. A somewhat less ambitious approach would be to recover parameters of a simplified model from the data, using, e.g., PCA to learn a linear basis for the scene appearance.

Of course, the appearance of a scene changes for other reasons besides variations in illumination. At longer time scales, buildings are built and demolished[1] and at shorter time scales, people move, water flows, and trees sway. Can we also capture and display these kinds of dynamic effects? For shorter time scales, it would be interesting to incorporate video of a scene into a reconstruction, and attempt to resynthesize dynamic elements from the video in other views.

Video. The dense temporal sampling provided by video is not only helpful in capturing dynamic effects, but could also be useful in many other ways as well. For instance, video can record the paths people take through a scene much more densely, and thus it would be interesting to apply the analyses in Chapter 6 to image collections with intermingled videos. At an even more basic level, it would be interesting to characterize how (and why) people tend to capture video of a scene or event, and how this differs from how people capture photos. Audio often comes along for free with video; adding audio to scenes would make for richer experiences, and could provide an authoring interface for creating video tours. An interesting interface for browsing such video is presented in [117].

[1] Usually in that order; this constraint is being used by the 4D Cities project to find a temporal ordering of historical photos of Atlanta [125].

Interiors. I have mainly explored reconstructing and exploring outdoor (usually architectural) scenes. While indoor scenes are not necessarily more challenging to reconstruct (see, for instance, the St. Peters and Pantheon reconstructions in Chapter 4), they do seem to be more challenging to visualize through photo collections. One reason for this is that, while the exterior front of a building can often be reasonably approximated with a plane, the inside of, say, a home, is more complex. Typically, several walls (as well as floor and ceiling) are visible at any given time, and there tend to be more objects (e.g., furniture) inside than outside. Hence, planar projection surfaces do not work as well inside (the Pantheon is an exception, as the interior is approximately a simple cylinder topped by a dome). To create good renderings of interiors, it may be necessary to use more complex projection surfaces consisting of multiple planes or triangulated meshes.

Building a new photo-sharing community. The work presented in this thesis makes use of the photos of many different photographers taking photos independently and without the intention of collectively reconstructing 3D geometry or creating immersive experiences. What if people knew that they were contributing photos towards a larger reconstruction effort, and could see the results of that effort as they unfolded? It would be interesting to create a website where people could upload their photos, as in Flickr, but where uploaded photos are automatically registered with an existing set of reconstructions. The current state of this world reconstruction could be viewed at any time, and the boundaries—the parts where new photos are needed—could be highlighted. Users could also see their own photos, as well as the parts of the world to which they have contributed. This distributed reconstruction effort could also be turned into a game in which points or other incentives are offered for adding onto the reconstruction, or incorporated into other fun activities, such as virtual treasure hunts. This type of online reconstruction procedure poses certain algorithmic challenges. For instance, given an image collection and a corresponding skeletal graph, is it possible to update the skeletal graph in an online fashion as new images come in? On the other hand, having users more involved may simplify other problems. Image matching, for instance, may be much easier if users interactively provide hints as to their location, e.g., by clicking or tapping on a map, thus constraining candidate matching images to nearby photos.

Creating a website that collects and registers the world's photos could be beneficial for a number of reasons. First, it could help infuse the site with a greater sense of community; not only are people contributing to a large project (which no individual could possibly accomplish alone), but users can also see how their photos fit in with those of other users, which could help forge links between users who may otherwise not notice a connection ("oh, he's been to that cafe in Istanbul as well!"). Second, it would be very helpful in identifying—and encouraging people to fill in—gaps where people normally shoot few photos, i.e., the places in-between the famous sites. This could become a scalable way of capturing a rich visual record of the entire world. Finally, the resulting reconstruction could serve as a foundation onto which other types of information could be added: labels on statues, buildings, paintings, and other objects; links to Wikipedia pages and other sources of information; and comments and reviews of restaurants and other businesses. Creating such a gathering place for the world's images would not only require new algorithms for managing massive amounts of data and predicting where new images should be taken, but also research into how to best give people incentives to contribute. In spite of these challenges, I believe that this project could have significant impact, as a repository of data, a model of the world useful for education, research, and virtual tourism, and a compelling social networking site.

BIBLIOGRAPHY

[1] Garmin GPS76 Owner's Manual and Reference Guide. `http://www8.garmin.com/manuals/GPS76_OwnersManual.pdf`.

[2] Aseem Agarwala, Maneesh Agrawala, Michael Cohen, David Salesin, and Richard Szeliski. Photographing long scenes with multi-viewpoint panoramas. In *SIGGRAPH Conf. Proc.*, pages 853–861, 2006.

[3] Aseem Agarwala, Ke Colin Zheng, Chris Pal, Maneesh Agrawala, Michael Cohen, Brian Curless, David Salesin, and Richard Szeliski. Panoramic video textures. In *SIGGRAPH Conf. Proc.*, pages 821–827, 2005.

[4] Daniel G. Aliaga and Ingrid Carlbom. Plenoptic stitching: A scalable method for reconstructing 3D interactive walkthroughs. In *SIGGRAPH Conf. Proc.*, pages 443–450, 2001.

[5] Ingo Althöfer, Gautam Das, David Dobkin, Deborah Joseph, and José Soares. On sparse spanners of weighted graphs. *Discrete Comput. Geom.*, 9(1):81–100, 1993.

[6] Steele Arbeeny and Deborah Silver. Spatial navigation of media streams. In *Proc. Int. Conf. on Multimedia*, pages 467–470, 2001.

[7] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *J. of the ACM*, 45(6):891–923, 1998.

[8] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. Speeded-up robust features (SURF). *Computer Vision and Image Understanding*, 110(3):346–359, 2008.

[9] Benjamin B. Bederson. Photomesa: A zoomable image browser using quantum treemaps and bubblemaps. In *Proc. Symposium on User Interface Software and Technology*, pages 71–80, 2001.

[10] Pravin Bhat, C. Lawrence Zitnick, Noah Snavely, Aseem Agarwala, Maneesh Agrawala, Brian Curless, Michael Cohen, and Sing Bing Kang. Using photographs to enhance videos of a static scene. In *Proc. Eurographics Symposium on Rendering*, pages 327–338, 2007.

[11] Olaf Booij, Zoran Zivkovic, and Ben Kröse. Sparse appearance based modeling for robot localization. In *Proc. Int. Conf. on Intelligent Robots and Systems*, pages 1510–1515, 2006.

[12] Fred L. Bookstein. Principal warps: Thin-plate splines and the decomposition of deformations. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 11(6):567–585, 1989.

[13] Jean-Yves Bouguet. Camera calibration toolbox for Matlab. `http://www.vision.caltech.edu/bouguetj/calib_doc/index.html`, 2001.

[14] Doug A. Bowman, David Koller, and Larry F. Hodges. Travel in immersive virtual environments: An evaluation of viewpoint motion control techniques. In *Proc. Virtual Reality Annual Int. Symposium*, pages 45–52, 1997.

[15] Matthew Brown and David Lowe. Unsupervised 3D object recognition and reconstruction in unordered datasets. In *Proc. Int. Conf. on 3D Digital Imaging and Modeling*, pages 56–63, 2005.

[16] Aeron M. Buchanan and Andrew W. Fitzgibbon. Damped Newton algorithms for matrix factorization with missing data. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, pages 316–322, 2005.

[17] Chris Buehler, Michael Bosse, Leonard McMillan, Steven Gortler, and Michael Cohen. Unstructured lumigraph rendering. In *SIGGRAPH Conf. Proc.*, pages 425–432, 2001.

[18] Leizhen Cai. NP-completeness of minimum spanner problems. *Discrete Appl. Math.*, 48(2):187–194, 1994.

[19] John Canny. A computational approach to edge detection. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 8(6):679–698, 1986.

[20] Shenchang Chen and Lance Williams. View interpolation for image synthesis. In *SIGGRAPH Conf. Proc.*, pages 279–288, 1993.

[21] Shenchang Eric Chen. QuickTime VR – An image-based approach to virtual environment navigation. In *SIGGRAPH Conf. Proc.*, pages 29–38, 1995.

[22] L. Paul Chew. Constrained Delaunay triangulations. In *Proc. Symposium on Computational Geometry*, pages 215–222, 1987.

[23] Stephane Christy and Radu Horaud. Euclidean reconstruction: From paraperspective to perspective. In *Proc. European Conf. on Computer Vision*, pages 129–140, 1996.

[24] Ondrej Chum, James Philbin, Josef Sivic, Michael Isard, and Andrew Zisserman. Total recall: Automatic query expansion with a generative feature model for object retrieval. In *Proc. Int. Conf. on Computer Vision*, 2007.

[25] Nico Cornelis, Kurt Cornelis, and Luc Van Gool. Fast compact city modeling for navigation pre-visualization. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, pages 1339–1344, 2006.

[26] Cassidy Curtis and Chris Whitney. eyestilts: home of the telestereoscope. `http://eyestilts.com/`.

[27] Rudolph P. Darken and John L. Sibert. Navigating large virtual spaces. *Int. J. of Human-Computer Interaction*, 8(1):49–71, 1996.

[28] Rudy P. Darken and John L. Sibert. A toolset for navigation in virtual environments. In *Proc. Symposium on User Interface Software and Technology*, pages 157–165, 1993.

[29] Marc Davis, Simon King, Nathan Good, and Risto Sarvas. From context to content: Leveraging context to infer media metadata. In *Proc. Int. Conf. on Multimedia*, pages 188–195, 2004.

[30] Andrew J. Davison. Active search for real-time vision. In *Proc. Int. Conf. on Computer Vision*, pages 66–73, 2005.

[31] Marnix de Nijs. explodedviews. `http://www.marnixdenijs.nl/exploded.htm`.

[32] Paul E. Debevec, Camillo J. Taylor, and Jitendra Malik. Modeling and rendering architecture from photographs: A hybrid geometry- and image-based approach. In *SIGGRAPH Conf. Proc.*, pages 11–20, 1996.

[33] Digital photography review. `http://www.dpreview.com/`.

[34] Digital photography review glossary: Sensor sizes. `http://www.dpreview.com/learn/?/Glossary/Camera_System/sensor_sizes_01.%htm`.

[35] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, December 1959.

[36] Cristina Russo dos Santos, Pascal Gros, Pierre Abel, Didier Loisel, Nicolas Trichaud, and Jean-Pierre Paris. Metaphor-aware 3D navigation. In *Proc. of the IEEE Symposium on Information Vizualization*, pages 155–165. IEEE Computer Society, 2000.

[37] Steven M. Drucker, Curtis Wong, Asta Roseway, Steven Glenner, and Steven De Mar. Mediabrowser: Reclaiming the shoebox. In *Proc. Working Conf. on Advanced Visual Interfaces*, pages 433–436, 2004.

[38] Steven M. Drucker and David Zeltzer. Intelligent camera control in a virtual environment. In *Proc. of Graphics Interface*, pages 190–199, 1994.

[39] Chris Engels, Henrik Stewenius, and David Nistér. Bundle adjustment rules. In *Proc. Symposium on Photogrammetric Computer Vision*, pages 266–271, 2006.

[40] Boris Epshtein, Eyal Ofek, Yonathan Wexler, and Pusheng Zhang. Hierarchical photo organization using geometric relevance. In *ACM Int. Symposium on Advances in Geographic Information Systems*, 2007.

[41] Everyscape. `http://www.everyscape.com`.

[42] "How can you become a scape artist?". `http://www.winsper.com/EveryScape/scape_artist.php`.

[43] Facebook. `http://www.facebook.com`.

[44] Steven Feiner, Blair MacIntyre, Tobias Hollerer, and Anthony Webster. A touring machine: Prototyping 3D mobile augmented reality systems for exploring the urban environment. In *Proc. IEEE Int. Symposium on Wearable Computers*, pages 74–81, 1997.

[45] Martin Fischler and Robert Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Readings in Computer Vision: Issues, Problems, Principles, and Paradigms*, pages 726–740, 1987.

[46] Andrew W. Fitzgibbon and Andrew Zisserman. Automatic camera recovery for closed or open image sequences. In *Proc. European Conf. on Computer Vision*, pages 311–326, 1998.

[47] Flickr. `http://www.flickr.com`.

[48] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. of the ACM*, 34(3):596–615, 1987.

[49] Yasutaka Furukawa and Jean Ponce. Accurate, dense, and robust multi-view stereopsis. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, 2007.

[50] Tinsley A. Galyean. Guided navigation of virtual environments. In *Proc. Symposium on Interactive 3D Graphics*, pages 103–105, 1995.

[51] Gefos Geosystems. `http://www.gefos.cz`.

[52] James J. Gibson. *The Ecological Approach to Visual Perception*. Lawrence Erlbaum Associates, 1979.

[53] Andreas Girgensohn, John Adcock, Matthew Cooper, Jonathon Foote, and Lynn Wilcox. Simplifying the management of large photo collections. In *Proc. Human-Computer Interaction*, pages 196–203, 2003.

[54] Michael Goesele, Brian Curless, and Steven M. Seitz. Multi-view stereo revisited. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, pages 2402–2409, 2006.

[55] Michael Goesele, Noah Snavely, Steven M. Seitz, Brian Curless, and Hugues Hoppe. Multi-view stereo for community photo collections. In *Proc. Int. Conf. on Computer Vision*, 2007.

[56] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1996.

[57] Google Earth. `http://earth.google.com`.

[58] Google Maps. `http://maps.google.com`.

[59] Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen. The lumigraph. In *SIGGRAPH Conf. Proc.*, pages 43–54, 1996.

[60] Graphviz - graph visualization software. `http://www.graphviz.org/`.

[61] Sudipto Guha and Samir Khuller. Approximation algorithms for connected dominating sets. *Algorithmica*, 20(4):374–387, 1998.

[62] Gabriele Guidi, Bernard Frischer, and Ignazio Lucenti. Rome Reborn - Virtualizing the ancient imperial Rome. In *Workshop on 3D Virtual Reconstruction and Visualization of Complex Architectures*, 2007.

[63] Chris Hand. A survey of 3D interaction techniques. *Computer Graphics Forum*, 16(5):269–281, 1997.

[64] Andrew J. Hanson and Eric A. Wernert. Constrained 3D navigation with 2D controllers. In *Proc. Conf. on Visualization*, pages 175–183, 1997.

[65] Frank Harary. *Graph theory*. Addison-Wesley, 1969.

[66] Richard I. Hartley. In defense of the eight-point algorithm. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 19(6):580–593, 1997.

[67] Richard I. Hartley and Frederik Schaffalitzky. PowerFactorization: 3D reconstruction with missing or uncertain data. In *Australia-Japan Advanced Workshop on Computer Vision*, 2003.

[68] Richard I. Hartley and Andrew Zisserman. *Multiple View Geometry*. Cambridge University Press, Cambridge, UK, 2004.

[69] Berthold K.P. Horn, Hugh M. Hilden, and Shahriar Negahdaripour. Closed-form solution of absolute orientation using orthonormal matrices. *J. Opt. Soc. of America A*, 5:1127–1135, July 1988.

[70] Intel math kernel library. http://www.intel.com/software/products/mkl.

[71] iPhoto. http://www.apple.com/ilife/iphoto.

[72] Alexander Jaffe, Mor Naaman, Tamir Tassa, and Marc Davis. Generating summaries and visualization for large collections of geo-referenced photographs. In *Proc. ACM SIGMM Int. Workshop on Multimedia Information Retrieval*, pages 89–98, 2006.

[73] Yushi Jing and Shumeet Baluja. VisualRank: Applying PageRank to large-scale image search. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 30(11):1877–1890, 2008.

[74] Rieko Kadobayashi and Katsumi Tanaka. 3D viewpoint-based photo search and information browsing. In *Proc. ACM Int. Conf. on Research and Development in Information Retrieval*, pages 621–622, 2005.

[75] Hyunmo Kang and Ben Shneiderman. Visualization methods for personal photo collections: Browsing and searching in the PhotoFinder. In *Int. Conf. on Multimedia*, pages 1539–1542, 2000.

[76] Sing Bing Kang, Peter-Pike Sloan, and Steven M. Seitz. Visual tunnel analysis for visibility prediction and camera planning. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, pages 2195–2202, 2000.

[77] Lyndon Kennedy, Mor Naaman, Shane Ahern, Rahul Nair, and Tye Rattenbury. How Flickr helps us make sense of the world: Context and content in community-contributed media collections. In *Proc. Int. Conf. on Multimedia*, 2007.

[78] Azam Khan, Ben Komalo, Jos Stam, George Fitzmaurice, and Gordon Kurtenbach. HoverCam: Interactive 3D navigation for proximal object inspection. In *Proc. Symposium on Interactive 3D Graphics and Games*, pages 73–80, 2005.

[79] Johannes Kopf, Matthew Uyttendaele, Oliver Deussen, and Michael F. Cohen. Capturing and viewing gigapixel images. In *SIGGRAPH Conf. Proc.*, 2007.

[80] Andreas Krause and Carlos Guestrin. Near-optimal observation selection using submodular functions. In *AAAI Conf. on Artifical Intelligence*, pages 1650–1654, 2007.

[81] E. Kruppa. Zur ermittlung eines objectes aus zwei perspektiven mit innerer orientierung. *Sitz.-Ber. Akad. Wiss., Wien, Math. Naturw. Kl., Abt. Ila.*, 122:1939–1948, 1913.

[82] Allan Kuchinsky, Celine Pering, Michael L. Creech, Dennis Freeze, Bill Serra, and Jacek Gwizdka. FotoFile: A consumer multimedia organization and retrieval system. In *Proc. Conf. on Human Factors in Computing Systems*, pages 496–503, 1999.

[83] John Lasseter. Principles of traditional animation applied to 3D computer animation. In *SIGGRAPH Conf. Proc.*, pages 35–44, 1987.

[84] Marc Levoy and Pat Hanrahan. Light field rendering. In *SIGGRAPH Conf. Proc.*, pages 31–42, 1996.

[85] Marc Levoy, Kari Pulli, Brian Curless, Szymon Rusinkiewicz, David Koller, Lucas Pereira, Matt Ginzton, Sean Anderson, James Davis, Jeremy Ginsberg, Jonathan Shade, and Duane Fulk. The Digital Michelangelo Project: 3D scanning of large statues. In *SIGGRAPH Conf. Proc.*, pages 131–144, 2000.

[86] Hongdong Li and Richard I. Hartley. Five-point motion estimation made easy. In *Proc. Int. Conf. on Pattern Recognition*, pages 630–633, 2006.

[87] Andrew Lippman. Movie maps: An application of the optical videodisc to computer graphics. In *SIGGRAPH Conf. Proc.*, pages 32–43, 1980.

[88] Live Search Maps. `http://maps.live.com`.

[89] Manolis Lourakis and Antonis Argyros. The design and implementation of a generic sparse bundle adjustment software package based on the Levenberg-Marquardt algorithm. Technical Report 340, Inst. of Computer Science-FORTH, Heraklion, Greece, 2004.

[90] David Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. of Computer Vision*, 60(2):91–110, 2004.

[91] Daniel Martinec and Tomas Pajdla. Robust rotation and translation estimation in multiview reconstruction. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, 2007.

[92] Jiri Matas, Ondrej Chum, Urban Martin, and Tomás Pajdla. Robust wide baseline stereo from maximally stable extremal regions. In *Proc. of the British Machine Vision Conf.*, volume 1, pages 384–393, 2002.

[93] Neil McCurdy and William Griswold. A systems architecture for ubiquitous video. In *Proc. Int. Conf. on Mobile Systems, Applications, and Services*, pages 1–14, 2005.

[94] Philip F. Mclauchlan. A batch/recursive algorithm for 3D scene reconstruction. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, pages 738–743, 2000.

[95] Leonard McMillan and Gary Bishop. Plenoptic modeling: An image-based rendering system. In *SIGGRAPH Conf. Proc.*, pages 39–46, 1995.

[96] Eric N. Mortensen and William A. Barrett. Intelligent scissors for image composition. In *SIGGRAPH Conf. Proc.*, pages 191–198, 1995.

[97] Mor Naaman, Andreas Paepcke, and Hector Garcia-Molina. From where to what: Metadata sharing for digital photographs with geogr aphic coordinates. In *Proc. Int. Conf. on Cooperative Information Systems*, pages 196–217, 2003.

[98] Mor Naaman, Yee Jiun Song, Andreas Paepcke, and Hector Garcia-Molina. Automatic organization for digital photographs with geographic coordinates. In *Proc. ACM/IEEE-CS Joint Conf. on Digital Libraries*, pages 53–62, 2004.

[99] Michael Naimark. Aspen the verb: Musings on heritage and virtuality. *Presence Journal*, 15(3), June 2006.

[100] Kai Ni, Drew Steedly, and Frank Dellaert. Out-of-core bundle adjustment for large-scale 3D reconstruction. In *Proc. Int. Conf. on Computer Vision*, 2007.

[101] David Nistér. Reconstruction from uncalibrated sequences with a hierarchy of trifocal tensors. In *Proc. European Conf. on Computer Vision*, pages 649–663, 2000.

[102] David Nistér. An efficient solution to the five-point relative pose problem. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(6):756–777, 2004.

[103] David Nistér. Preemptive RANSAC for live structure and motion estimation. *Mach. Vis. Appl.*, 16(5):321–329, 2005.

[104] David Nistér and Henrik Stewénius. Scalable recognition with a vocabulary tree. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, pages 2118–2125, 2006.

[105] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer Series in Operations Research. Springer-Verlag, New York, NY, 1999.

[106] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the Web. In *Proc. Int. World Wide Web Conf.*, pages 161–172, 1998.

[107] Finding paths through the world's photos website. `http://phototour.cs.washington.edu/findingpaths/`.

[108] Photo tourism website. `http://phototour.cs.washington.edu/`.

[109] Photosynth. `http://photosynth.net/`.

[110] The Photosynth photography guide. `http://http://mslabs-777.vo.llnwd.net/e1/documentation/Photosynth\%20Gu%ide\%20v7.pdf`.

[111] Picasa. `http://www.picasa.com`.

[112] Picasa web albums. http://picasaweb.google.com.

[113] John C. Platt, Mary Czerwinski, and Brent A. Field. PhotoTOC: Automatic clustering for browsing personal photographs. In *IEEE Pacific Rim Conf, on Multimedia*, 2003.

[114] Conrad J. Poelman and Takeo Kanade. A paraperspective factorization method for shape and motion recovery. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 19(3):206–218, 1997.

[115] Marc Pollefeys, David Nistér, Jan-Michael Frahm, Amir Akbarzadeh, Phillipos Mordohai, Brian Clipp, Christopher Engels, David Gallup, S.-J. Kim, Paul Merrell, Christina Salmi, Sudipta Sinha, Brad Talton, Liang Wang, Qing-Xiong Yang, Henrik Stewénius, Ruigang Yang, Greg Welch, and Herman Towles. Detailed real-time urban 3D reconstruction from video. *Int. J. of Computer Vision*, 78(2-3):143–167, 2008.

[116] Marc Pollefeys, Luc van Gool, Maarten Vergauwen, Frank Verbiest, Kurt Cornelis, Jan Tops, and Reinhard Koch. Visual modeling with a hand-held camera. *Int. J. of Computer Vision*, 59(3):207–232, 2004.

[117] Suporn Pongnumkul, Jue Wang, and Michael Cohen. Creating map-based storyboards for browsing tour videos. In *Proc. Symposium on User Interface Software and Technology*, pages 13–22, 2008.

[118] Till Quack, Bastian Leibe, and Luc Van Gool. World-scale mining of objects and events from community photo collections. In *Proc. Int. Conf. on Content-based Image and Video Retrieval*, pages 47–56, 2008.

[119] Jason Repko and Marc Pollefeys. 3D models from extended uncalibrated video sequences: Addressing key-frame selection and projective drift. In *Proc. Int. Conf. on 3-D Digital Imaging and Modeling*, pages 150–157, 2005.

[120] D. P. Robertson and Roberto Cipolla. Building architectural models from many views using map constraints. In *Proc. European Conf. on Computer Vision*, volume II, pages 155–169, 2002.

[121] Roy A. Ruddle, Stephen J. Payne, and Dylan M. Jones. Navigating buildings in “desk-top” virtual environments: Experimental investigations using extended navigational experience. *J. of Experimental Psychology: Applied*, 3(2):143–159, 1997.

[122] Alberto Ruiz, Pedro E. López de teruel, and Ginés García-mateos. A note on principal point estimability. In *Proc. Int. Conf. on Pattern Recognition*, pages 11–15, 2002.

[123] Bryan C. Russell, Antonio Torralba, Kevin P. Murphy, and William T. Freeman. LabelMe: A database and web-based tool for image annotation. *Int. J. of Computer Vision*, 77(1-3):157–173, 2008.

[124] Frederik Schaffalitzky and Andrew Zisserman. Multi-view matching for unordered image sets, or “How do I organize my holiday snaps?”. In *Proc. European Conf. on Computer Vision*, volume 1, pages 414–431, 2002.

[125] Grant Schindler, Frank Dellaert, and Sing Bing Kang. Inferring temporal order of images from 3D structure. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, 2007.

[126] Cordelia Schmid and Andrew Zisserman. Automatic line matching across views. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, pages 666–671, 1997.

[127] Seadragon. `http://labs.live.com/Seadragon.aspx`.

[128] Steven Seitz, Brian Curless, James Diebel, Daniel Scharstein, and Richard Szeliski. A comparison and evaluation of multi-view stereo reconstruction algorithms. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, pages 519–528, 2006.

[129] Ben Shneiderman and Hyunmo Kang. Direct annotation: A drag-and-drop strategy for labeling photos. In *Proc. Int. Conf. on Information Visualisation*, pages 88–95, 2000.

[130] Ian Simon, Noah Snavely, , and Steven M. Seitz. Scene summarization for online image collections. In *Proc. Int. Conf. on Computer Vision*, 2007.

[131] Karan Singh and Ravin Balakrishnan. Visualizing 3D scenes using non-linear projections and data mining of previous camera movements. In *Proc. Int. Conf. on Computer graphics, virtual reality, visualisation and interaction in Africa*, pages 41–48, 2004.

[132] Josef Sivic, Biliana Kaneva, Antonio Torralba, Shai Avidan, and William T. Freeman. Creating and exploring a large photorealistic virtual space. In *Proc. of the First IEEE Workshop on Internet Vision*, 2008.

[133] Josef Sivic and Andrew Zisserman. Video Google: A text retrieval approach to object matching in video s. In *Proc. Int. Conf. on Computer Vision*, pages 1470–1477, 2003.

[134] Randall C. Smith and Peter Cheeseman. On the representation and estimation of spatial uncertainly. *Int. J. Robotics Research*, 5(4):56–68, 1987.

[135] Smugmug photo sharing. `http://www.smugmug.com`.

[136] Noah Snavely, Rahul Garg, Steven M. Seitz, and Richard Szeliski. Finding paths through the world’s photos. In *SIGGRAPH Conf. Proc.*, 2008.

[137] Noah Snavely, Steven M. Seitz, and Richard Szeliski. Photo tourism: Exploring photo collections in 3D. In *SIGGRAPH Conf. Proc.*, pages 835–846, 2006.

[138] Diomidis D. Spinellis. Position-annotated photographs: A geotemporal web. *IEEE Pervasive Computing*, 2(2):72–79, 2003.

[139] Drew Steedly, Irfan Essa, and Frank Delleart. Spectral partitioning for structure from motion. In *Proc. Int. Conf. on Computer Vision*, pages 996–103, 2003.

[140] Drew Steedly and Irfan A. Essa. Propagation of innovative information in non-linear least-squares structure from motion. In *Proc. Int. Conf. on Computer Vision*, pages 223–229, 2001.

[141] Peter Sturm and Bill Triggs. A factorization based algorithm for multi-image projective structure and motion. In *Proc. European Conf. on Computer Vision*, pages 709–720, 1996.

[142] Ivan E. Sutherland. A head-mounted three dimensional display. In *Proc. Fall Joint Computer Conf.*, pages 757–764, 1968.

[143] R. Szeliski and S. B. Kang. Recovering 3D shape and motion from image streams using nonlinear least squares. *J. of Visual Communication and Image Representation*, 5(1):10–28, March 1994.

[144] Richard Szeliski. Image alignment and stitching: A tutorial. *Foundations and Trends in Computer Graphics and Computer Vision*, 2(1), December 2006.

[145] Desney S. Tan, George G. Robertson, and Mary Czerwinski. Exploring 3D navigation: Combining speed-coupled flying with orbiting. In *Proc. Conf. on Human Factors in Computing Systems*, pages 418–425. ACM Press, 2001.

[146] Hiroya Tanaka, Masatoshi Arikawa, and Ryosuke Shibasaki. A 3-D photo collage system for spatial navigations. In *Revised Papers from the Second Kyoto Workshop on Digital Cities II, Computational and Sociological Approaches*, pages 305–316, 2002.

[147] Camillo J. Taylor. VideoPlus: A method for capturing the structure and appearance of immersive environments. *IEEE Transactions on Visualization and Computer Graphics*, 8(2):171–182, April-June 2002.

[148] Carlo Tomasi and Takeo Kanade. Shape and motion from image streams under orthography: A factorization method. *Int. J. of Computer Vision*, 9(2):137–154, 1992.

[149] Kentaro Toyama, Ron Logan, and Asta Roseway. Geographic location tags on digital images. In *Proc. Int. Conf. on Multimedia*, pages 156–166, 2003.

[150] Bill Triggs, Phil McLauchlan, Richard I. Hartley, and Andrew Fitzgibbon. Bundle adjustment – a modern synthesis. In *Vision Algorithms: Theory and Practice*, volume 1883 of *Lecture Notes in Computer Science*, pages 298–372, 2000.

[151] U.S. Geological Survey. `http://www.usgs.com`.

[152] Matthew Uyttendaele, Antonio Criminisi, Sing Bing Kang, Simon Winder, Richard Szeliski, and Richard I. Hartley. Image-based interactive exploration of real-world environments. *IEEE Computer Graphics and Applications*, 24(3):52–63, 2004.

[153] Maarten Vergauwen and Luc Van Gool. Web-based 3D reconstruction service. *Mach. Vis. Appl.*, 17(2):321–329, 2006.

[154] Luc Vincent. Taking online maps down to street level. *IEEE Computer*, 40(12):118–120, December 2007.

[155] Luis von Ahn and Laura Dabbish. Labeling images with a computer game. In *Proc. Conf. on Human Factors in Computing Systems*, pages 319–326, 2004.

[156] Matthias Wandel. Exif jpeg header manipulation tool. `http://www.sentex.net/~mwandel/jhead/`.

[157] Colin Ware and Steven Osborne. Exploration and virtual camera control in virtual three dimensional environments. In *Proc. Symposium on Interactive 3D Graphics*, pages 175–183. ACM Press, 1990.

[158] Wikipedia. `http://www.wikipedia.org`.

[159] Windows Live Local - Virtual Earth Technology Preview. `http://preview.local.live.com`.

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

Appendix A

ESTIMATING THE FOCAL LENGTH OF A DIGITAL PHOTO FROM EXIF TAGS

Almost every digital camera manufactured in the last few years embeds useful data about each captured photo into the JPEG files that get stored to the camera's memory. This information, encoded in Exif (exchangeable image file format) tags, often includes exposure time, focus, aperture, whether the flash was activated, and focal length. The last of these, focal length, is especially useful for structure from motion. This appendix describes how to extract the focal length from the Exif tags of a digital photo and to convert it to the pixel units that can be directly used in structure from motion. The following pieces of information are required for this conversion:

1. The focal length estimate (in mm), f_{mm}.
2. The width of the camera CCD (in mm), CCD_{mm}.
3. The dimensions of the image (in pixels), $w_{\text{pixels}}, h_{\text{pixels}}$.

Once all of these numbers are known, computing the focal length in pixel, f_{pixels}, can be done via a simple unit conversion:

$$f_{\text{pixels}} = \frac{f_{\text{mm}}}{CCD_{\text{mm}}}(\max\{w_{\text{pixels}}, h_{\text{pixels}}\})$$

This computation assumes that the image has not been cropped or warped in any way (aside from simple resizing of the image). For instance, if the focal length of a photo is listed in the Exif tags as 5.4mm, we used, say, a Canon PowerShot S100 (with a CCD width of 5.27mm) to take the photo, and the image resolution is 1600×1200, then

$$f_{\text{pixels}} = \frac{5.4}{5.27} \times 1600 = 1639.47.$$

To extract f_{mm} from the Exif tags, any Exif tag reader will do. JHead [156] is a good free one which works from the command line. The resolution of an image is also embedded in the Exif tags.

The camera manufacturer and model name are also usually embedded as well, and for many camera models CCD_{mm} can be found online on camera review sites such as Digital Photography Review [33]. For instance, the specifications page for the Canon PowerShot S100 on Digital Photography Review lists the dimensions of the sensor as 1/2.7” (5.27 x 3.96 mm). Some cameras embed the CCD width directly into the Exif tags, but in my experience, this number is often unreliable.

As with the example above, sometimes the CCD size is given as a ratio of inches (e.g., 1/2.7”). These sizes stem from standard diameters of television camera tubes, and the conversion from these units to millimeters is not straightforward; Digital Photo Review describes the conversion in its glossary [34].

Appendix B
COMPLEXITY ANALYSIS OF INCREMENTAL STRUCTURE FROM MOTION

The running time of the incremental structure from motion (SfM) algorithm of Chapter 3 depends on the exact schedule in which the images are added to the reconstruction, and is dominated by the calls to SBA [89] after every batch of new images is reconstructed. Let $S(t)$ be the running time of a call to SBA when there are t images currently in the reconstruction. Let $T(n, r)$ be the total amount of time spent inside SBA during SfM given n input images, r of which are added during each iteration of SfM.[1] Recall that $S(t) = \Theta(t^3)$, so there exists N such that, for all $t > N$,

$$c_1 t^3 \leq S(t) \leq c_2 t^3 \tag{B.1}$$

for some constants c_1 and c_2.

Case 1: $r = \frac{n}{k}$**.** First, suppose that $\frac{n}{k}$ images are added per iteration of SfM, i.e., we are considering $T\left(n, \frac{n}{k}\right)$. We have that

$$T\left(n, \frac{n}{k}\right) = \sum_{i=1}^{k} S\left(i\frac{n}{k}\right).$$

For $\frac{n}{k} > N$, the bounds in Equation B.1 apply. Considering just the upper bound, we have that:

$$\begin{aligned} T\left(n, \frac{n}{k}\right) &\leq \sum_{i=1}^{k} c_2 \left(i\frac{n}{k}\right)^3 \\ &\leq \sum_{i=1}^{k} c_2 \left(k\frac{n}{k}\right)^3 \\ &= c_2 k n^3. \end{aligned}$$

[1] In practice, the number of images added will vary between iterations. For simplicity of analysis, I only consider the case where the same number of images are added each time.

It follows that $T\left(n, \frac{n}{k}\right) = O(n^3)$. A lower bound proportional to n^3 can also be shown, thus $T\left(n, \frac{n}{k}\right) = \Theta(n^3)$.

Case 2: $r = \ell$**.** Now let us consider the case when a constant number ℓ of images are added during each iteration, for a total of $\frac{n}{\ell}$ iterations (assuming ℓ divides n). We have that

$$T(n, \ell) = \sum_{i=1}^{n/\ell} S(i \cdot \ell). \tag{B.2}$$

Let us group the first $\frac{N}{\ell}$ terms in the summation in Equation B.2 into a single constant, C (assuming, without loss of generality, that ℓ divides N); the remaining terms (where $i \cdot \ell > N$) can then be bounded using Equation B.1. Considering just the upper bound gives:

$$\begin{aligned}
T(n, \ell) &= \sum_{i=1}^{n/\ell} S(i \cdot \ell) && \text{(B.3)}\\
&= \sum_{i=1}^{N/\ell} S(i \cdot \ell) + \sum_{i=N/\ell+1}^{n/\ell} S(i \cdot \ell) && \text{(B.4)}\\
&= C + \sum_{i=N/\ell+1}^{n/\ell} S(i \cdot \ell) && \text{(B.5)}\\
&\leq C + \sum_{i=N/\ell+1}^{n/\ell} c_2 (i \cdot \ell)^3 && \text{(B.6)}\\
&= C + c_1 \ell^3 \sum_{i=N/\ell+1}^{n/\ell} i^3 && \text{(B.7)}\\
&= C + c_1 \ell^3 \left(\sum_{i=1}^{n/\ell} i^3 - \sum_{i=1}^{N/\ell} i^3 \right). && \text{(B.8)}
\end{aligned}$$

The sum over i^3 for $1 \leq i \leq N/\ell$ is another constant, which we denote by D:

$$\begin{aligned}
T(n,\ell) &\leq C + c_1\ell^3 \left(\sum_{i=1}^{n/\ell} i^3 - D \right) && \text{(B.9)} \\
&= \big(C - c_1\ell^3 D\big) + c_1\ell^3 \sum_{i=1}^{n/\ell} i^3 && \text{(B.10)} \\
&= \big(C - c_1\ell^3 D\big) + c_1\ell^3 \left[\frac{\left(\frac{n}{\ell}\right)\left(\frac{n}{\ell}+1\right)}{2} \right]^2 && \text{(B.11)} \\
&= O(n^4). && \text{(B.12)}
\end{aligned}$$

Therefore $T(n,\ell) = O(n^4)$. Using the lower bound in Eq. B.1, we can also show that $T(n,\ell) = \Omega(n^4)$ through a very similar analysis, replacing

$$T(n,\ell) \leq C + \sum_{i=N/\ell+1}^{n/\ell} c_2(i \cdot \ell)^3$$

in Eq. B.6 with

$$T(n,\ell) \geq C + \sum_{i=N/\ell+1}^{n/\ell} c_1(i \cdot \ell)^3$$

Hence, $T(n,\ell) = \Theta(n^4)$.

Appendix C
LINE SEGMENT RECONSTRUCTION

Once the point cloud has been reconstructed and the camera positions recovered using structure from motion (SfM), the SfM pipeline reconstructs 3D line segments in the scene. Line segments can be useful for generating better triangulated morphs (Chapter 5, Section 5.1.2) and for enhancing the renderings of architectural scenes. Line segment reconstruction from images has been investigated in work such as [126]. My 3D line segment reconstruction algorithm is similar, and has the following steps:

1. Detect 2D line segments in each image.
2. Derive a set of candidate 2D line segment matches by comparing line segments between pairs of nearby images.
3. Find sets of mutually consistent matches across multiple images (line tracks) above a certain size and triangulate the matching 2D line segments to obtain a 3D line segment.

These steps are now described in more detail.

To detect 2D line segments, I use Canny edge detection [19], followed by an edge-linking step to form edge chains. These chains may have any shape, and may be closed; the next step is to break the chains into sub-chains that approximate line segments. To do so, for each edge chain, I fit a line to the chain using orthogonal regression. If the furthest point on the chain from the fitted line has a distance to the line greater than a threshold, the chain is broken in two at that extremal point, then this procedure is recursively applied two the two new chains. Finally, I remove chains smaller than a threshold, and store the two endpoints of the line segment approximating of each of the remaining chains. I use $S(I)$ to denote the set of 2D line segments found in image I.

I then match line segments between images. For each image I_j, I first compute a set of other images with which to match. This set should contain images whose cameras are close to camera C_j and looking in roughly the same direction. To compute this set, I find the 32 cameras closest to C_j

and remove those whose viewing directions are at an angle greater than a threshold to the viewing direction of C_j.

Next, for each camera C_k in the set, I consider each line segment $s \in S(I_j)$. Each line segment $t \in S(I_k)$ is labeled as a candidate match of s if t meets the following two conditions:

1. The endpoints of t are not too far away from the epipolar lines of the endpoints of s in image I_k.

2. The L_2 distance between a strip of intensity values around s and a strip of intensity values around t is not too large. The intensity values are sampled along epipolar lines, and each strip is normalized for bias and gain before their L_2 distance is computed. After normalization, I use a threshold of 0.3 to reject candidate matches.

After computing candidate matches between pairs of neighbors, I compute connected components of candidate matches, as was done with SIFT feature matches in the SfM pipeline (Chapter 3, Section 3.1). Unlike with keypoint matching, however, I now know where each photograph was taken, so we can immediately check connected components for *geometric* consistency. For each connected component of matches with a subset of consistent line segments of size at least four, I create a 3D line segment. The 3D line segment is created by triangulating corresponding endpoints of the 2D line segments; the resulting 3D points form the endpoints of the 3D line segment.

Appendix D

PHOTO CREDITS

I would like to thank the following people for generously allowing me to use and reproduce their photos in my work. Photos from these Flickr users were used in the Photo Tourism project, and appear in Chapter 5 or the associated video.

Holly Ables of Nashville, TN	http://www.flickr.com/photos/tmlens/
Rakesh Agrawal	http://www.flickr.com/photos/asmythie/
Pedro Alcocer	http://www.flickr.com/photos/pealco/
Julien Avarre	http://www.flickr.com/photos/eole/
Rael Bennett	http://www.flickr.com/photos/spooky05/
Loïc Bernard	http://www.flickr.com/photos/loic1967/
Nicole Bratt	http://www.flickr.com/photos/nicolebratt/
Nicholas Brown	http://www.flickr.com/photos/nsgbrown/
Domenico Calojero[1]	http://www.flickr.com/photos/mikuzz/
DeGanta Choudhury	http://www.flickr.com/photos/deganta/
dan clegg	http://www.flickr.com/photos/mathmandan/
Claude Covo-Farchi	http://www.flickr.com/photos/bip/
Alper Çuğun	http://www.flickr.com/photos/alper/
W. Garth Davis	http://www.flickr.com/photos/garth/
Stamatia Eliakis	http://www.flickr.com/photos/12537899@N00/
Dawn Endico[2]	http://www.flickr.com/photos/candiedwomanire/
Silvana M. Felix	http://www.flickr.com/photos/jadoreparis/
Jeroen Hamers	http://www.flickr.com/photos/jhamers/
Caroline Härdter	http://www.flickr.com/photos/dotpolka/
Mary Harrsch	http://www.flickr.com/photos/44124324682@N01/
Molly Hazelton	http://www.flickr.com/photos/mollx/
Bill Jennings[3]	http://www.flickr.com/photos/mrjennings/
Michelle Joo	http://www.flickr.com/photos/maz/
Tommy Keswick	http://www.flickr.com/photos/mrwilloby/
Kirsten Gilbert Krenicky	http://www.flickr.com/photos/kirsten_gilbert_krenicky/
Giampaolo Macorig	http://www.flickr.com/photos/60405541@N00/
Erin K Malone[4]	http://www.flickr.com/photos/erinmalone/
Daryoush Mansouri	http://www.flickr.com/photos/daryoush/

[1] mikuzz@gmail.com

[2] endico@gmail.com

[3] Supported by grants from the National Endowment for the Humanities and the Fund for Teachers.

[4] photographs copyright 2005

Paul Meidinger	http://www.flickr.com/photos/pmeidinger/
Laurete de Albuquerque Mouazan	http://www.flickr.com/photos/21guilherme/
Callie Neylan	http://www.flickr.com/photos/williamcatherine/
Robert Norman	http://www.flickr.com/photos/bobtribe/
Dirk Olbertz	http://www.flickr.com/photos/dirkolbertz/
Dave Ortman	http://www.flickr.com/photos/dortman/
George Owens	http://www.flickr.com/photos/georgeowens/
Claire Elizabeth Poulin	http://www.flickr.com/photos/claireep/
David R. Preston	http://www.flickr.com/photos/dosbears/
Jim Sellers and **Laura Kluver**	http://www.flickr.com/photos/jim_and_laura/
Peter Snowling	http://www.flickr.com/photos/19654387@N00/
Rom Srinivasan	http://www.flickr.com/photos/romsrini/
Jeff Allen Wallen Photography	http://www.flickr.com/photos/goondockjeff/
Daniel West	http://www.flickr.com/photos/curious/
Todd A. Van Zandt	http://www.flickr.com/photos/vanzandt/
Dario Zappalà	http://www.flickr.com/photos/zappa/
Susan Elnadi	http://www.flickr.com/photos/30596986@N00/

I also would like to acknowledge the following people whose photographs were reproduced for Chapter 5 under Creative Commons licenses:

Shoshanah	http://www.flickr.com/photos/shoshanah[1]
Dan Kamminga	http://www.flickr.com/photos/dankamminga[1]
Tjeerd Wiersma	http://www.flickr.com/photos/tjeerd[1]
Manogamo	http://www.flickr.com/photos/se-a-vida-e[2]
Ted Wang	http://www.flickr.com/photos/mtwang [3]
Arnet	http://www.flickr.com/photos/gurvan [3]
Rebekah Martin	http://www.flickr.com/photos/rebekah [3]
Jean Ruaud	http://www.flickr.com/photos/jrparis [3]
Imran Ali	http://www.flickr.com/photos/imran [3]
Scott Goldblatt	http://www.flickr.com/photos/goldblatt [3]
Todd Martin	http://www.flickr.com/photos/tmartin [4]
Steven	http://www.flickr.com/photos/graye [4]
ceriess	http://www.flickr.com/photos/ceriess [1]
Cory Piña	http://www.flickr.com/photos/corypina [3]
mark gallagher	http://www.flickr.com/photos/markgallagher [1]
Celia	http://www.flickr.com/photos/100n30th [3]
Carlo B.	http://www.flickr.com/photos/brodo [3]
Kurt Naks	http://www.flickr.com/photos/kurtnaks [4]
Anthony M.	http://www.flickr.com/photos/antmoose [1]
Virginia G	http://www.flickr.com/photos/vgasull [3]

[1] http://creativecommons.org/licenses/by/2.0/

[2] http://creativecommons.org/licenses/by-nd/2.0/

[3] http://creativecommons.org/licenses/by-nc-nd/2.0/

[4] http://creativecommons.org/licenses/by-nc/2.0/

Collection credit and copyright notice for *Moon and Half Dome*, 1960, by Ansel Adams: Collection Center for Creative Photography, University of Arizona, © Trustees of The Ansel Adams Publishing Rights Trust.

The following people allowed me to use photos for the Pathfinder project. Their photos appear in Chapter 6 and in the accompanying video.

Storm Crypt	`http://www.flickr.com/photos/storm-crypt`
James McPherson	`http://www.flickr.com/photos/jamesontheweb`
Wolfgang Wedenig	`http://www.flickr.com/photos/wuschl2202`
AJP79	`http://www.flickr.com/photos/90523335@N00`
Tony Thompson	`http://www.flickr.com/photos/14489588@N00`
Warren Buckley	`http://www.flickr.com/photos/studio85`
Keith Barlow	`http://www.flickr.com/photos/keithbarlow`
beautifulcataya	`http://www.flickr.com/photos/beautifulcataya`
Smiley Apple	`http://www.flickr.com/photos/smileyapple`
crewealexandra	`http://www.flickr.com/photos/28062159@N00`
Ian Turk	`http://www.flickr.com/photos/ianturk`
Randy Fish	`http://www.flickr.com/photos/randyfish`
Justin Kauk	`http://www.flickr.com/photos/justinkauk`
Airplane Lane	`http://www.flickr.com/photos/photons`
Katie Holmes	`http://www.flickr.com/photos/katieholmes`
Cher Kian Tan	`http://www.flickr.com/photos/70573485@N00`
Erin Longdo	`http://www.flickr.com/photos/eel`
James McKenzie	`http://www.flickr.com/photos/jmckenzie`
Eli Garrett	`http://www.flickr.com/photos/portenaeli`
Francesco Gasparetti	`http://www.flickr.com/photos/gaspa`
Emily Galopin	`http://www.flickr.com/photos/theshrtone`
Sandro Mancuso	`http://www.flickr.com/photos/worldwalker`
Ian Monroe	`http://www.flickr.com/photos/eean`
Noam Freedman	`http://www.flickr.com/photos/noamf`
morbin	`http://www.flickr.com/photos/morbin`
Margrethe Store	`http://www.flickr.com/photos/margrethe`
Eugenia and Julian	`http://www.flickr.com/photos/eugeniayjulian`
Allyson Boggess	`http://www.flickr.com/photos/allysonkalea`
Ed Costello	`http://www.flickr.com/photos/epc`
Paul Kim	`http://www.flickr.com/photos/fmg2001`
Susan Elnadi	`http://www.flickr.com/people/30596986@N00`
Mathieu Pinet	`http://www.flickr.com/photos/altermativ`
© Ariane Gaudefroy	`http://www.flickr.com/photos/kicouette`
Briana Baldwin	`http://www.flickr.com/photos/breezy421`
Andrew Nguyen	`http://www.flickr.com/photos/nguy0833`
Curtis Townson	`http://www.flickr.com/photos/fifty50`
Rob Thatcher (rob@hypereal.co.uk)	`http://www.flickr.com/photos/pondskater`
Greg Scher	`http://www.flickr.com/photos/gregscher`

VITA

Keith (Noah) Snavely grew up in Tucson, Arizona, and received his B.S. in Computer Science and Math from the University of Arizona in 2003. He then joined the Ph.D. program of the Computer Science and Engineering department at the University of Washington, where he worked with Steven M. Seitz and Richard Szeliski. In 2008, he received the Doctor of Philosophy degree.