Abstracted Workflow Framework with a Structure from Motion Application

by

Adam J. Rossi

A.S. Monroe Community College, 2001
B.S. Rochester Institute of Technology, 2004

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science
in the Chester F. Carlson Center for Imaging Science
College of Science
Rochester Institute of Technology

May 2014

Signature of the Author _____

Accepted by _____
Coordinator, M.S. Degree Program Date

CHESTER F. CARLSON CENTER FOR IMAGING SCIENCE
COLLEGE OF SCIENCE
ROCHESTER INSTITUTE OF TECHNOLOGY
ROCHESTER, NEW YORK

CERTIFICATE OF APPROVAL

M.S. DEGREE THESIS

The M.S. Degree Thesis of Adam J. Rossi
has been examined and approved by the
thesis committee as satisfactory for the
thesis required for the
M.S. degree in Imaging Science

Dr. Harvey Rhody, Thesis Advisor

Dr. Carl Salvaggio

Dr. Derek Walvoord

Date

Abstracted Workflow Framework with a Structure from Motion Application

by

Adam J. Rossi

Submitted to the
Chester F. Carlson Center for Imaging Science
in partial fulfillment of the requirements
for the Master of Science Degree
at the Rochester Institute of Technology

Abstract

In scientific and engineering disciplines, from academia to industry, there is an increasing need for the development of custom software to perform experiments, construct systems, and develop products. The natural mindset initially is to shortcut and bypass all overhead and process rigor in order to obtain an immediate result for the problem at hand, with the misconception that the software will simply be thrown away at the end. In a majority of the cases, it turns out the software persists for many years, and likely ends up in production systems for which it was not initially intended. In the current study, a framework that can be used in both industry and academic applications mitigates underlying problems associated with developing scientific and engineering software. This results in software that is much more maintainable, documented, and usable by others, specifically allowing new users to extend capabilities of components already implemented in the framework.

There is a multi-disciplinary need in the fields of imaging science, computer science, and software engineering for a unified implementation model, which motivates the development of an abstracted software framework. Structure from motion (SfM) has been identified as one use case where the abstracted workflow framework can improve research efficiencies and eliminate implementation redundancies in scientific fields. The SfM process begins by obtaining 2D images of a scene from different perspectives. Features from the images are extracted and correspondences are established. This provides a sufficient amount of information to initialize the problem for fully automated processing. Transformations are established between views, and 3D points are established via triangulation algorithms. The parameters for the camera models for all views / images are solved through bundle adjustment, establishing a highly consistent point cloud. The initial sparse point cloud and camera matrices are used to generate a dense point cloud through patch based techniques or densification algorithms such as Semi-Global Matching (SGM). The point cloud can be

visualized or exploited by both humans and automated techniques. In some cases the point cloud is "draped" with original imagery in order to enhance the 3D model for a human viewer. The SfM workflow can be implemented in the abstracted framework, making it easily leverageable and extensible by multiple users.

Like many processes in scientific and engineering domains, the workflow described for SfM is complex and requires many disparate components to form a functional system, often utilizing algorithms implemented by many users in different languages / environments and without knowledge of how the component fits into the larger system. In practice, this generally leads to issues interfacing the components, building the software for desired platforms, understanding its concept of operations, and how it can be manipulated in order to fit the desired function for a particular application. In addition, other scientists and engineers instinctively wish to analyze the performance of the system, establish new algorithms, optimize existing processes, and establish new functionality based on current research. This requires a framework whereby new components can be easily plugged in without affecting the current implemented functionality.

The need for a universal programming environment establishes the motivation for the development of the abstracted workflow framework. This software implementation, named Catena, provides base classes from which new components must derive in order to operate within the framework. The derivation mandates requirements be satisfied in order to provide a complete implementation. Additionally, the developer must provide documentation of the component in terms of its overall function and inputs. The interface input and output values corresponding to the component must be defined in terms of their respective data types, and the implementation uses mechanisms within the framework to retrieve and send the values. This process requires the developer to componentize their algorithm rather than implement it monolithically. Although the requirements of the developer are slightly greater, the benefits realized from using Catena far outweigh the overhead, and results in extensible software. This thesis provides a basis for the abstracted workflow framework concept and the Catena software implementation. The benefits are also illustrated using a detailed examination of the SfM process as an example application.

Acknowledgements

I would like to thank my advisor, Dr. Harvey Rhody, for his willingness to take me on as a research student and develop a thesis topic that was well-suited for my interests and background. I was fortunate to have taken many courses with Dr. Rhody, and it was a pleasure getting to know him as my thesis advisor throughout the course of our research. His personal and professional advice have been invaluable to me. I'm always amazed at how often I discover the applicability and foresight of his abstractions in practice.

Thanks to my committee members, Dr. Carl Salvaggio and Dr. Derek Walvoord. After taking my first class with Carl, it became obvious that he is one of the exceptional teachers at RIT. He truly cares about his students and puts a great deal of effort into lectures and class material. He is student-focused and goes out of his way to make sure they succeed. Additionally, I am very grateful not only to have Derek on my committee, but to have him as a teaching assistant for many courses, and for the privilege of working with him at Exelis. Derek solidified Image Science coursework theory through application and educated me in new subject areas. I am very grateful for the opportunity to work with Derek and I greatly value our friendship.

I would like to thank all the faculty, staff, and students at the Center for Imaging Science at RIT. Although I was a part-time student and infrequently on campus, they were very welcoming and helpful. I am also appreciative of their willingness to experiment with the Catena software, and I hope they were able to gain from their investment.

I would like to give special thanks to my family and friends for their support. You kept me motivated and helped me accomplish a significant milestone in my academic career.

Also, thank you to my former co-workers at Exelis (in alphabetical order): Bernie Brower, Brad Paul, Brian Staab, Brian Terwilliger, Frank Tantalo, Jason Wynne, Jon Antal, Ken Brodeur, Kevin Pietrzak, Tim Burt, and Wendy LeFebvre. You played a key role in the development of my career. Special thanks to those who adapted the Catena framework. Your input was invaluable for increasing software quality and flexibility: Brandon May, Jordyn Stoddard, Kyle Ausfeld, and Sue Munn. I would also like to thank my GPS III teammates. We spent a lot of time away from home supporting a crucial project that is going to improve national security: Bill Taft, Chris Bower, Greg Kirchoff, Howard Brayman, Paul Gilmour, Rich Lourette, Steph Panicali, and Tim Flynn.

Thank you to the following musicians for providing a soundtrack to writing my thesis and coding: Underworld, Com Truise, Tycho, Junior Boys, Hot Chip, Toro Y Moi, Lusine, Bibio, Cut/Copy, Gold Panda, Aphex Twin, Boards of Canada, Washed Out, Tangerine Dream, Philip Glass, and The Bad Plus.

Lastly, thank you to ITT / Exelis Geospatial Systems for the financial support of my education.

I would like to dedicate this thesis to my wife, Jamie. My success and motivation to complete the coursework and research would not have been possible without her support. She made numerous sacrifices to ensure my success, and I am forever indebted to her. Thank you for your loving support, Jamie.

Contents

List of Figures

List of Source Code

Chapter 1

Introduction

1.1 Problem Definition

In many scientific communities, including imaging science, it is common for a process to be defined, which requires various algorithms and tools in order to carry out a specific task. It is typically advantageous for the scientist to leverage previous work developed by others in order to advance a particular aspect of the field. This ultimately involves combining components implemented by others in order to establish a baseline system for the scientist's research and development. However, it is likely that the components were not developed together nor envisioned as being part of a greater overarching system, and they are likely inflexible for accomplishing a different goal. There is usually little to no consideration given to reuse or documentation, especially in the context of developing prototype software. This effectively results in monolithic, platform-specific, and disposable software that has limited use in specialized domains by a small group of users. This problem becomes amplified when multiple tools, developed by multiple users, are required to carry out a task. Since tools are developed without considering the overall scope, they are difficult to integrate. The need for a workflow framework that integrates applications and allows for flexible extensions and replacement components becomes apparent.

1.2 Motivation

The structure from motion (SfM) task requires the integration of many proprietary and open-source components in order to effectively carry out the processing. The common steps are enumerated below:

1. Image source specification
2. Image list filtering / creating subsets
3. Creating symbolic links
4. Image conversion
5. Feature extraction
6. Feature matching
7. Bundle adjustment
8. Radial distortion compensation
9. Output conversion
10. View reduction
11. Point cloud generation
12. Surface reconstruction / image draping
13. Visualization / exploitation

Historically, components for the SfM process have been implemented in various languages, had inconsistent interfaces, required various input data formats, and were generally incompatible with each other. If one wishes to establish a working SfM system, there is a steep learning curve and a considerable amount of time required to establish a base system. The integration of the components is typically very application and platform-specific, making it difficult for others to leverage previous work. Additionally, this solution does not result in a robust processing system and makes it difficult for users to utilize it as a test bed for the development of alternative algorithms and components. Therefore, an abstract solution to this problem will be presented, which can be utilized for the SfM process, and more generally applied to other domains.

In general, there are varying levels of operating environments for SfM, or any scientific application:

1. **Manual Method**: This includes manual execution of the tools and interfacing of components. This solution is very laborious, time consuming, and error-prone.

2. **Semi-Automated Method**: This provides a static workflow definition that cannot be changed to accommodate new components or utilized for alternative applications. Much of the implementation is "hard-coded" or otherwise inflexible.

3. **Fully Automated Method**: A flexible, automated workflow framework integrates disparate tools for general applications. The interface between components is addressed to provide inter-operability of different component implementations through strong contract. The components and workflow are self-documenting to promote reuse by other users.

Typically, the semi-automated method is used to conduct research in the SfM domain. Previous approaches, including scripts, Open Street Map (OSM) [3], and Visual SfM [4] have allowed for automation of the workflow; however, these are static definitions that lack extensibility and applicability to other domains. Therefore, the ideal solution for scientists is a flexible workflow framework, which is not only needed by the SfM community, but for scientific and engineering fields in general. By creating an abstract solution to this problem, the fundamental components become applicable to other domains and the components can be reused within the same general workflow specification. This thesis describes the development of the abstracted workflow framework, Catena, and its applicability to improving/optimizing the environment for scientific research in the SfM field.

1.3 Outline of Thesis

This thesis is organized in the following manner:

- **Chapter 2**: The thesis research is motivated by establishing an objective that addresses a common need for an abstracted workflow framework in many scientific and engineering domains.

- **Chapter 3**: Implementation and design details of the abstracted workflow framework and the Catena software implementation are presented.

- **Chapter 4**: The image science theory of the SfM application is presented, and implementation of the SfM components in the Catena framework is described.

- **Chapter 5**: Using the SfM workflow, 3D models from multi-view imagery were generated. Components from the SfM workflow were leveraged for other applications to demonstrate the extensibility of Catena.

- **Chapter 6**: Conclusions from the research are discussed and anticipated future work is outlined.

Chapter 2

Objectives

In many applications, the manual or semi-automated solution to a scientific problem is used, but lacks the flexibility and extensibility for use in future implementations. Using the SfM implementation as an example, it is clear that monolithic software development yields disposable software, which cannot be leveraged by future researchers. The solution to this issue is multi-disciplinary, drawing on imaging science theory and software engineering architecture, design, and implementation. Often there is disparity between these two disciplines; image scientists tend to minimize the software component because they analyze the problem in its native, entire form, while software engineers lack the theory knowledge to allow for componentization and algorithm substitution. Analysis of the independent requirements (i.e., without consideration for the other field) are described in Section 2.1 from an image scientist's perspective and Section 2.2 from a software engineer's perspective as it relates to the SfM implementation. Ultimately, a fully automated solution is required, which draws on strengths from both complementary fields of study to leverage design and implementation of an abstracted workflow framework. The benefits of the abstracted workflow framework will be exemplified in the SfM application.

2.1 Structure from Motion (Image Science)

Currently, the SfM implementation is run manually or semi-automatically with disparate components, which tends to be error-prone and time-consuming. Therefore, the need exists for a tool that easily integrates these individual components for fully automated execution of the process. Ideally, the resulting framework would provide a method of implementation that is extensible, and which minimizes the overhead to the researcher. A constrained environment for research and development would allow for consistency between users, and allow them to express the workflow in terms of components. This fully automated

solution must inherently allow for optimization of algorithms, and provide a method for documentation of algorithms, parameters, and interfaces.

2.2 Workflow Framework (Software Engineering)

The software engineering approach to the SfM application includes componentization with defined interfaces and documentation according to a contract set forth by the framework, such that the framework automatically executes the necessary steps that enable the components to be reused. High-level implementation in a cross-platform language would be required such that the framework may be run on any operating system with little-to-no consideration to maintain platform independence. The framework must be executed efficiently, requiring minimal computational resources, and allowing for parallel execution using all available cores and compute nodes without additional burden on the developer (i.e., feature of the framework). The framework should be able to persist / restore its state, and provide programmatic and graphical interfaces. Component interfaces should be defined / specified, and as such, the framework would validate legal combinations of constructed components.

2.3 Benefits of the Multi-Disciplinary Abstracted Workflow Framework

The potential benefits of combining the individual image science and software engineering requirements far outweigh current level of support from any manual or semi-automated method. The implementation capability will be easily accessible to the scientist / engineer developer, principal investigator, manager, student, etc., and will be easily understood through comprehensive documentation (requirement built into framework). In its simplest form, a novice user can be given a baseline component implementation and can establish predefined workflows with basic functionality that can be shared between users. Even at this basic level, the user can manipulate nominal baseline parameters. Advanced users can modify the baseline workflow or start constructing a custom workflow from predefined component packages. This fully automated solution can quickly and quantitatively identify deficiencies in components, which would expose areas needing additional development, and can aid in guiding research efforts. The abstracted workflow framework is not specific to the SfM application, but can be implemented for any workflow. This benefit minimizes the learning curve to new users and eliminates duplication of efforts, allowing researchers to easily contribute knowledge back to the field of study.

Chapter 3

Implementation

3.1 Workflow Framework

In many domains, the concept of a workflow, or *chain*, is applied to carry out a sequence of tasks that can be represented by individual *stages*. This concept is often utilized in the area of image processing. The output from a given stage is fed into subsequent stages and becomes the input required to carry out a specific process. The generalization of this concept is shown in Figure 3.1. This process continues sequentially until the final stage is complete. In the current study, the abstracted workflow framework, named Catena [5], was implemented in the Python programming language due to its object-oriented, platform-agnostic, and ubiquitous nature. In addition, Python has gained a large following throughout the scientific community and most of the anticipated target users are familiar with this language. Chains can be built programmatically or through the usage of a graphical user interface (GUI). Both methods support persistence for interoperability between environments and future usage of the constructed chain. The stages included as part of the core framework, and custom stages implemented by users, are dynamically discovered and loaded for ease of deployment, integration, operation, and extensibility.

3.1.1 Stages

The fundamental component of the workflow is the stage. The stage encapsulates a function and should be developed in such a manner that provides general capability across chain instances (i.e., it should be designed generically so that it can be leveraged by other applications). A stage defines three classes of information within it: the properties and self-documentation; input interface; and output interface (Figure 3.2). The properties required in the class constructor are used to control behavior and modes of the stage. Ad-

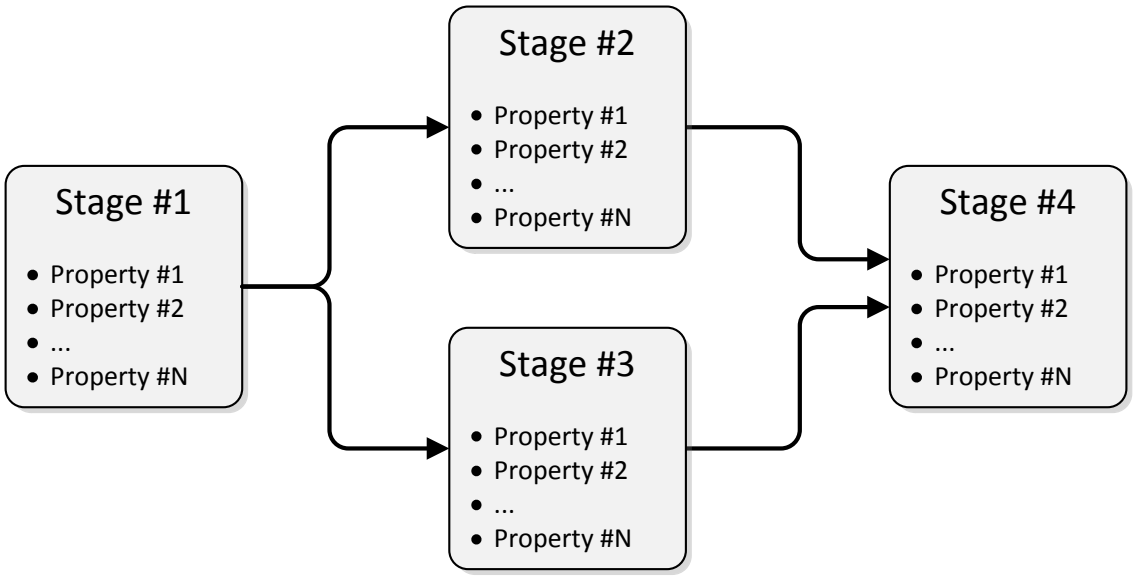


Figure 3.1: Generalized chain representation showing the connection of stages to compose a workflow.

ditionally, information contained in the constructor facilitates self-documentation of the stage, whereby it can be utilized to provide online help for users. Next, the input interface is declared with type information. This is required prior to rendering the chain to validate interface consistency among connected stages. Finally, the output interface is defined in the same manner as the input. The advantage of defining stage interfaces is that it allows the user to conveniently exchange one stage for another.

At the core of the stage lies the execution method, where the main function of the stage is invoked. Due to the demand-pull nature of the chain rendering (explained in further detail later), stages request input parameters from the associated up-stream stage(s) outputs. This affects a recursive request up-stream. The output parameters from the up-stream stage(s) and the properties provided upon construction by the instantiator comprise all of the required information needed to carry out the function of a given stage. The stage executes its specific function and sets the output parameters as defined on the output interface. Finally, the framework ensures that all of the outputs are set and of the correct type upon the completion of stage execution. Figure 3.3 provides the class diagram of the StageBase class and two examples of classes (stages) that derive from it. This guarantees consistency with the interface contracts specified between connected stages.

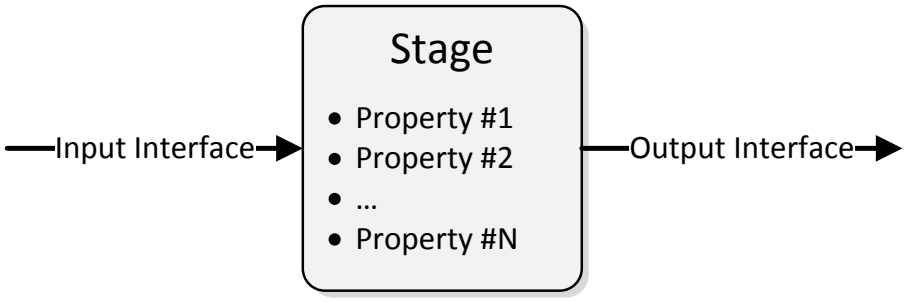


Figure 3.2: Representative stage structure showing properties and input / output interfaces. The input data comes from previous stages, while the output data is fed into the subsequent stage(s). The individual stage interfaces are well-defined such that equivalent stages can be swapped for other implementations in a chain.

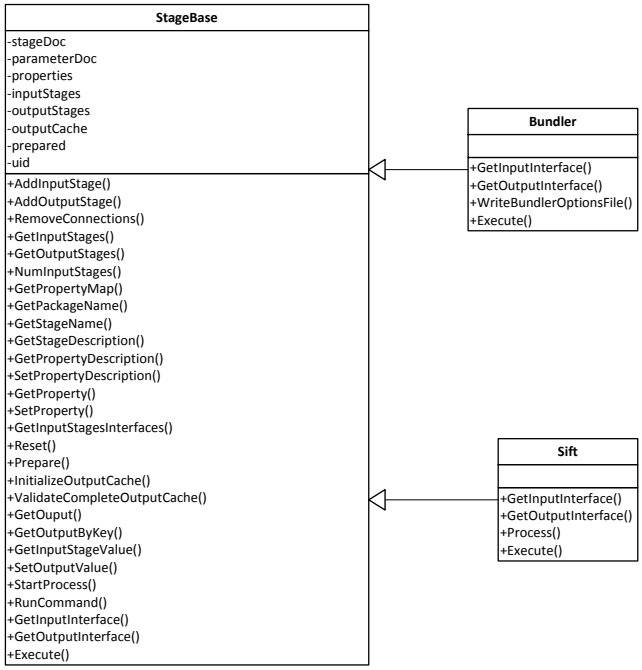


Figure 3.3: StageBase class diagram and two derived classes / stages. The StageBase class is the fundamental component in the workflow framework, from which every stage must inherit.

3.1.2 Chain

A chain is composed of an arbitrarily complex sequence of stages representing a desired workflow that is required to carry out a task. The user has the ability to employ a stage as input into one or many down-stream stages, or even include it in different chain segments. In addition, the user may construct the chain in such a manner that results in more than one output stage. In this case, the output stages and the respective chains can be rendered in parallel or sequentially.

3.1.3 Rendering

Once a chain is constructed from a collection of stages, it is ready for rendering. The verb *render* is borrowed from the image processing concept, describing the method by which the chain is executed to produce the desired output. The chain is rendered in a *demand-pull* fashion. This means that the chain is rendered from the perspective of the output stage. The request is made from stage to stage up-stream, and each stage executes its function in order to produce an output for the stage(s) in front of it.

In addition, caching is employed as the chain is rendered so that a stage and its associated outputs are only calculated once. This is essential in minimizing the computation time of the overall chain. The caching scheme described is beneficial when a chain is constructed with one stage connected to multiple up- or down-stream stages. It is important to note that the developer is isolated from the rendering and caching details, as the development methodology described in Section 3.1.1 ensures that both are employed as part of the framework.

Using the generic components of the abstracted workflow framework, the user defines customized stages by inheriting from a base stage class. This provides the self-documentation feature, interface validation, caching, and rendering capability. A benefit to this abstract approach is that an enhanced chain capability can be easily developed to operate in multi-threaded, distributed, or high-performance computing environments without modification to the concrete stage classes. The powerful benefit of abstraction easily allows for scalability. Additionally, this architecture provides the flexibility to develop an entirely different rendering or caching scheme without modification to the stage implementations.

3.2 Catena

Catena [5] is an abstracted workflow framework implemented in the Python programming language that allows for the development of stages and chains. The stage and chain properties are defined at a high level and subsequently demonstrated with the SfM application. This fully automated method was originally designed to solve SfM problems, however benefits of the abstracted workflow framework have been realized in other scientific domains and will be discussed. Catena ensures that stage implementation is intuitive, stage connectivity is consistent in the development of chains, and re-use is promoted throughout many fields of study.

3.2.1 Stage Implementation

This section illustrates the implementation of a stage for use with the Catena framework. The trivial example will write a text file with the resolution information of the images that are passed to the stage. The stage will act as a pass-through; it will simply set the input images as the output. The code in Listing 1 is the entire implementation of the stage. It will be explained thoroughly in the following subsections.

```
import Chain
import Common

class ResolutionInfo(Chain.StageBase):

  def __init__(self, inputStages=None, resolutionFilePath=""):

    Chain.StageBase.__init__(self,
                             inputStages,
                             "Resolution Information",
                             {"Resolution Path":
                              "Path to resolution information file"})

    self._properties["Resolution Path"] = resolutionFilePath

  def GetInputInterface(self):
    return {"images":(0,Common.sfmImages)}

  def GetOutputInterface(self):
    return {"images":Common.sfmImages}

  def Execute(self):

    images = self.GetInputStageValue(0, "images")

    self.StartProcess()

    f = open(self._properties["Resolution Path"], "w")
    for im in images.GetImages():
      f.write("%s: xres=%d, yres=%d\n" % (im.GetFileName(),
                                          im.GetXResolution(),
                                          im.GetYResolution()))
    f.close()

    self.SetOutputValue("images", images)
```

Listing 1: The code provides an example of a complete implementation of a stage that outputs resolution information of given images to a file.

Module Imports

First, the `Chain` and `Common` modules are imported. The `Chain` module contains the base Catena functionality and the `StageBase` class, from which all Catena stages must inherit. The `Common` module contains data types that are typically used throughout the chain.

```
import Chain
import Common
```

Class Definition

Next, the class is defined. As previously mentioned, all Catena stages must inherit from `Chain.StageBase`. This base class contains the fundamental power and stage functionality.

```
class ResolutionInfo(Chain.StageBase):
```

Constructor

The convention in Catena is for the constructor to take a list of input stages as the first parameter. The subsequent parameters are optional and specific to the stage being implemented. Another requirement is that each parameter be given a default value. This serves a dual purpose as it provides a nominal parameter setting in the event the user does not wish to override the default value, and the default parameter value conveys important data type information to the framework (as Python is a loosely typed language).

```
def __init__(self, inputStages=None, resolutionFilePath=""):
```

Constructor Implementation

The base class' constructor must be called immediately. The constructor takes three parameters, including a list of input stages, a string that describes the stage, and a dictionary. Each item of the dictionary describes the parameters of the stage. The key is the parameter name, and the value is a description of the parameter. This information serves as self-documentation in other applications, such as the Chain Builder and Chain GUI, that offers users online help.

```
Chain.StageBase.__init__(self,
                         inputStages,
                         "Resolution Information",
                         {"Resolution Path":
                          "Path to resolution information file"})
```

Stage Properties

Lastly, the properties dictionary is initialized using the parameters of the stage, provided as parameters to the constructor. At this point, the user is free to carry out other initialization procedures required for the stage.

```
self._properties["Resolution Path"] = resolutionFilePath
```

Input Interface

The `GetInputInterface` method must be implemented (overloaded) as part of the `StageBase` contract. This is effectively a "pure virtual" method. A dictionary must be returned, which represents the input interface. The item's key is the input parameter name and the value is a 2-tuple, where the first value is the index of the input stage from which the parameter originates. The second value is the type of the parameter. This information is used to perform interface consistency as chains are constructed, and to enforce type compatibility. This allows the framework to perform run-time checking to minimize user errors both in development and application.

```
def GetInputInterface(self):
    return {"images":(0,Common.sfmImages)}
```

Output Interface

The `GetOutputInterface` method is similar to the input method in that it defines the output parameters produced by the stage. The only difference is that the value of the dictionary items is not a tuple, it is simply the data type of the output parameter.

```
def GetOutputInterface(self):
    return {"images":Common.sfmImages}
```

Execution Method

The final method required by the framework is the `Execute` method. This is where the core functionality of the stage is carried out.

```
def Execute(self):
```

Execution Implementation

Each `Execute` method typically follows the same pattern:

1. Get input parameters from input stages using the `GetInputStageValue` method
2. Signal to the framework that processing is starting (`StartProcess`)
3. Carry out the work of the stage
4. Set the output parameters of the stage using the `SetOutputValue` method

The parameters from the input stages are acquired by calling the `GetInputStageValue` method, providing the index of the input stage and the name of the parameter. This effectively causes a recursive request up-stream from the end stage, as Catena implements a demand-pull render model. The stage indices, parameter names, and data types must be consistent with the interface defined in the `GetInputInterface` method.

```
images = self.GetInputStageValue(0, "images")
```

The framework is informed that processing is starting. This is used for analysis and logging (e.g., to calculate timing information for chain and individual stage rendering).

```
self.StartProcess()
```

The main work of the stage is carried out next. In this example, the “Resolution Path” string is accessed via the properties dictionary and a corresponding text file is opened for writing. The images that were obtained from the input stage are iterated through, and

the file name and resolution information are written for each image. The text file is closed and the process is complete.

```
f = open(self._properties["Resolution Path"], "w")
for im in images.GetImages():
    f.write("%s: xres=%d, yres=%d\n" % (im.GetFileName(),
                                        im.GetXResolution(),
                                        im.GetYResolution()))
f.close()
```

Finally, the output parameter values are set using the `SetOutputValue` method. The parameter names and data types must be consistent with the interface defined in the `GetOutputInterface` method.

```
self.SetOutputValue("images", images)
```

3.2.2 Chain Implementation

The following section will utilize SfM stages that have been developed in Catena order to carry out a 3D modelling task. Figure 3.4 provides a high-level view of the SfM chain and its component stages. The script in Listing 2 represents the complete implementation of the SfM chain, however, it will be decomposed and explained thoroughly in the following subsections.

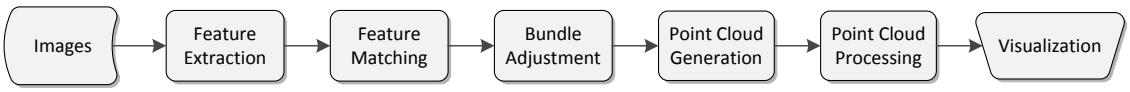


Figure 3.4: High-level SfM chain defining all the steps of the algorithm.

```
import sys, os
sys.path.append(os.path.abspath("."))
import Chain # Chain must be imported first, requirement of registry
import Sources, FeatureExtraction, FeatureMatch
import BundleAdjustment, Cluster

# path to images
imagePath = "/images"

# PMVS path
pmvsPath = os.path.join(imagePath,"pmvs")

# build chain
imageSource = Sources.ImageSource(imagePath, "jpg")
sift = FeatureExtraction.Sift(imageSource, False, "SiftHess")
keyMatch = FeatureMatch.KeyMatch(sift, False, "KeyMatchFull")
bundler = BundleAdjustment.Bundler([keyMatch, imageSource])
radialUndistort = Cluster.RadialUndistort([bundler, imageSource])
prepCmvsPmvs = Cluster.PrepCmvsPmvs(radialUndistort, pmvsPath)
cmvs = Cluster.CMVS(prepCmvsPmvs)
pmvs = Cluster.PMVS(cmvs)

# render chain
print Chain.Render(pmvs, "sfmLog.txt")

# persist chain
Chain.StageRegistry.Save("sfmChain.dat")
```

Listing 2: A script that programmatically builds a SfM Catena chain.

Import Modules

The packages that contain stages to build the desired chain must be imported. The order is very important in this case. First, the `sys` and `os` modules are imported in order to append the absolute path of the current working directory to the path environment variable. Catena requires scripts to be launched from the root directory, this is necessary in order to locate stage packages. The auto-discovery feature requires the `Chain` module to be imported first. This finds all occurrences of classes that inherit from the `Chain.StageBase` class. The packages, which contain the stage definitions used during the chain building are then imported.

```
import sys, os
sys.path.append(os.path.abspath("."))
import Chain # Chain must be imported first, requirement of registry
import Sources, FeatureExtraction, FeatureMatch
import BundleAdjustment, Cluster
```

Path Definitions

The `imagePath` variable is set to the location of the images that will be processed for creation of the 3D models. The `pmvsPath` variable is derived from the `imagePath`.

```
# path to images
imagePath = "/images"

# PMVS path
pmvsPath = os.path.join(imagePath,"pmvs")
```

Chain Construction

The remaining steps create an instance of each stage in the chain. The pattern of creating a stage instance requires that the previous stage be given as the first parameter (input stage) to the constructor. Each stage also defines its own properties, provided as subsequent parameters to the constructor. Please refer to Appendix A.2 for detailed information of each stage.

Image Source

The `ImageSource` stage is used to generate a list of images that exist on disk. In this case, "jpg" is the file extension of interest. Therefore, all files whose extension is "jpg" will be added to the image list and output to the following stage.

```
imageSource = Sources.ImageSource(imagePath, "jpg")
```

SIFT

The scale-invariant feature transform (SIFT) [1] stage accepts a list of images and generates keypoint descriptor files for each respective image. The descriptor files contain feature vectors of salient points in the image. The keypoint descriptors collection is represented as a class and the instantiated object is passed to the output stage. The SIFT stage takes

two parameters. The first controls whether the descriptor files are parsed and maintained in memory, and the second selects the SIFT implementation.

```
sift = FeatureExtraction.Sift(imageSource, False, "SiftGPU")
```

Feature Matching

The key matching stage takes a keypoint descriptor collection object and matches the descriptors based on their properties. The output is a class that represents the keypoint match table. The feature matching stage takes two parameters; the first controls whether the keypoint matches are parsed and maintained in memory, and the second selects the feature matching implementation to employ.

```
keyMatch = FeatureMatch.KeyMatch(sift, False, "KeyMatchGPU")
```

Bundle Adjustment

The bundle adjustment stage is an abstraction of the Bundler [6] program. It accepts the keypoint matches and images as input, generating a proprietary Bundler output file, which is represented as a class within the framework.

```
bundler = BundleAdjustment.Bundler([keyMatch, imageSource])
```

Radial Undistort

The radial undistort stage takes the Bundler output and the list of images as input. It uses the radial distortion coefficients computed by Bundler to warp the images so as to remove the radial lens distortion. A new Bundler file is generated, along with a new collection that represents the undistorted images.

```
radialUndistort = Cluster.RadialUndistort([bundler, imageSource])
```

CMVS / PMVS Preparation

The cluster-based multi-view stereo software (CMVS) and patch-based multi-view stereo software (PMVS) [7] programs expect their input to be in a particular form that is different from Bundler's output. Therefore, this preparation stage is used to pre-process the inputs. This stage takes the Bundler file and image collection as input. It moves the Bundler file and images to an acceptable directory for CMVS and PMVS. In addition, it generates a "vis" file and collection of camera matrices that are computed from the contents of the Bundler file.

```
prepCmvsPmvs = Cluster.PrepCmvsPmvs(radialUndistort, pmvsPath)
```

CMVS

CMVS [7] requires a Bundler file and image collection as its input. It runs a clustering algorithm, which reduces the overall input image set by identifying redundant views of the scene, and also breaks up the image set into smaller independent image sets for parallel processing. The stage outputs a Bundler file, image collection, "vis" file, "cluster" file, and "camera centers" file.

```
cmvs = Cluster.CMVS(prepCmvsPmvs)
```

PMVS

PMVS [8] requires a Bundler file and image collection as its input. It runs a patched-based multi-view stereo algorithm to generate a dense point cloud. The outputs are the dense 3D point cloud, a "patch" file, and a "pset" file.

```
pmvs = Cluster.PMVS(cmvs)
```

Chain Rendering

Finally, the chain is rendered by calling the `Chain.Render` method, which accepts the last stage object as its input. The method also requires a string parameter, which specifies where the log file shall be written.

```
print Chain.Render(pmvs, "sfmLog.txt")
```

Chain Persistence

The chain can be saved to a file for later usage by calling the `Chain.StageRegistry.Save` method and providing a path to the data file to write. This allows for restoration of the persist file, from which the chain can be rendered.

```
Chain.StageRegistry.Save("sfmChain.dat")
```

Persisted Chain Rendering

The script in Listing 3 illustrates how to restore a chain from a persist file and render a selected stage. First, the `Chain.StageRegistry.Load` method is called with a path to the persist file. The `Load` method returns a list of head and tail stages. This is useful for programmatic traversal. Next, the `Chain.Render` is called by providing the first tail stage as its parameter and a log file string. This requires *a priori* knowledge of the chain structure. Since a single tail stage exists from the chain that was built, the PMVS stage is located at index 0 in the `tailStages` list.

```
# load the sfm chain
headStages, tailStages = Chain.StageRegistry.Load("sfmChain.dat")

# render the tail stage (pmvs)
print Chain.Render(tailStages[0], "sfmLog.txt")
```

Listing 3: A script that loads a persisted Catena chain and renders the tail stage.

3.2.3 Stage Development Patterns

The following sections explain some of the common stage implementation patterns. The methods that aid in the implementation are explained.

Conditional Execution

A convenience method has been included in the `Common.Utility` package that can be used to determine if the core stage functionality should be executed (example provided in Listing 4). The first parameter is a boolean value that will typically be provided from the user, which indicates whether the stage should be executed, even if other conditions are satisfied that indicate execution is not needed. For example, if the outputs that would result from executing the stage already exist, the execution of the stage could be bypassed. The remaining parameters to the `ShouldRun` method are directories or files that will be checked for existence. If the first parameter (`Force Run`) is true or any of the directories or files given do not exist, the method will return true. The code below illustrates the usage of the `ShouldRun` method.

```
Common.Utility.ShouldRun(self._properties["Force Run"],
                         bundlerOptionsFilePath,
                         bundlerOutputPath,
                         bundlerOutputFilePath)
```

Listing 4: Example of the `ShouldRun` utility method for executing external applications.

External Program Execution

A stage can represent an external application that accepts a set of command line arguments for execution. A method named `RunCommand` has been provided on the `StageBase` class for ease of implementation (see example in Listing 5). The first parameter is the name of the executable. The location of the executable is determined at run-time and is dependent on the platform (explained below). The next parameter is a string of the command line arguments. The `CommandArgs` and `Quoted` methods are provided for convenience (discussed below). The user may also specify the execution working directory (`cwd`) and whether a shell should be used for invocation. Example usage of the `RunCommand` method with the `Bundle2PMVS` program is provided below.

```
self.RunCommand("Bundle2PMVS",
                Common.Utility.CommandArgs(
                Common.Utility.Quoted(imagelist),
                Common.Utility.Quoted(bundleFile),
                Common.Utility.Quoted(outputPath)),
                cwd = os.path.split(imagelist)[0])
```

Listing 5: An example of the `RunCommand` method provided on the `StageBase` class, which is used to invoke external applications.

The `RunCommand` method utilizes the `GetExePath` method in the `Utility` module. This method serves two purposes. First, it locates the executable according to the platform. For example, if executing on a 64-bit Linux environment, the executable will be searched for under the stage's directory: `Linux64bit/bin`. Secondly, in Linux environments, the `LD_LIBRARY_PATH` environment variable is used to include paths to dependent libraries. As such, in this example, the `Linux64bit/lib` directory will be added to the `LD_LIBRARY_PATH` environment variable in order for the executable to resolve dynamic libraries.

The `Quoted` method simply formats the given string in quotes, as required when specifying strings that contain spaces on the command line. The `CommandArgs` method accepts a collection of argument strings and formats them into a single string that can be passed to the `RunCommand` method.

Image Processing Base Stage Class

A base class named `ImageProcessStageBase` is provided in the `Common` package of Catena. A common interface pattern and sequence of operations were discovered throughout many stage implementations dealing with images, thus the common functionality was factored out into this base class for ease of implementation. The input and output interfaces include an image or images. The execution processes images depending on the "should run" pattern, writing the images to a new directory or constructing a new name based on the input name, and passing the processed images as the output.

There is one pure-virtual method that must be implemented, `ProcessImage`. This method has two parameters, the input image file name and output image file name. This method will be called by the base class on every image to process. Optionally, the `GetOutputImagePath` can be overridden to specify the output image file name and path. The default implementation assumes the output path is different from the input and a unique file extension, relative to the input images, is specified.

As a simple complete example provided in Listing 6, a stage that copies images to an "output path" will be implemented. One can envision the usage of this base class to implement a stage that processes image(s) using an algorithm implementation or other process. The constructor parameters include `StageBase` parameters, stage-specific parameters, and `ImageProcessStageBase` parameters (see code comments in Listing 6). However, many

derived classes are much simpler than this example in that they only include the implementation of the `ProcessImage` method.

Please see the full example in the Catena repository:
`Testing/exampleImageProcessStageBase.py`

```
class CopyImages(Common.ImageProcessStageBase):

    def __init__(self,
      inputStages=None,       # input stages (StageBase)
      prefixName="",          # file name prefix (CopyImages)
      outputPath="",          # output path (ImageProcessStageBase)
      imageExtension="tif",   # image extension (ImageProcessStageBase)
      forceRun=False,         # force run (ImageProcessStageBase)
      enableStage=True):      # enable stage (ImageProcessStageBase)

            Common.ImageProcessStageBase.__init__(self,
                inputStages,
                outputPath,
                imageExtension,
                forceRun,
                enableStage,
                "Copies images",
                {"Prefix Name":"Output file name prefix"})

            self._properties["Prefix Name"] = prefixName


    def GetOutputImagePath(self, inputImagePath):

        # construct the output image path, including the prefix name
        return os.path.join(self._properties["Output Image Path"],
            self._properties["Prefix Name"] +
            os.path.splitext(os.path.basename(inputImagePath))[0] +
            "."+self._properties["Image Extension"])

    def ProcessImage(self, inputImagePath, outputImagePath):

        # copy the input to output
        shutil.copy(inputImagePath, outputImagePath)
```

Listing 6: An example class derived from the `Common.ImageProcessStageBase` class. The base class provides common stage functionality for stages that define `sfmImage(s)` on both their input and output interface.

3.2.4 Tap Point Stage

A generalized tap point stage was implemented in the framework. It is essentially a pass-through stage in terms of the parameters, but it allows for default or specific printing to the log file (and `stdout`). Listing 7 illustrates the `TapPoint` stage, which takes a single stage as the input and an optional dictionary of print functions. The print function dictionary is keyed by type, where each value is a function that will print the input values (from the input stage) of the specific type. The example shows an inline lambda function declaration for the printing of `sfmImages` objects. If a print dictionary is not provided, the overloaded string method on the object will be utilized. This is illustrated in the second tap point stage instance in Listing 7.

```
import sys, os
sys.path.append(os.path.abspath("."))
import Chain # Chain must be imported first, requirement of registry
import Sources, FeatureExtraction, Common

# build chain
imageSource = Sources.ImageSource("/images", "jpg")

# insert tap point stage with print function
tap = Common.TapPoint(
  imageSource,{Common.sfmImages:lambda x: "Image Path: " + x.GetPath()})

# insert tap point stage without print function
tap = Common.TapPoint(tap)

sift = FeatureExtraction.Sift(tap, False, "SiftHess")

# render chain
print Chain.Render(sift,"log.txt")
```

Listing 7: An example Catena chain that uses the `TapPoint` stage to inspect intermediate stage output values.

3.2.5 Unit Test

A unit test script (`unitTest.py`) has been provided in the `Testing` directory. This script exercises all of the stages included with the framework, including all of the optional modes

(e.g., SIFT variants) and strives to exercise all the underlying support classes. The user is encouraged to execute this script upon checking out or exporting the repository to baseline the functionality of Catena on their system, as there may be subtle differences in the environment that affect execution of the components.

3.2.6 Chain Builder

The Chain Builder tool allows for graphical building of chains. A screenshot of the interface is provided in Figure 3.5. The list of stage packages is accessed by right-clicking on the canvas. A context menu of stages, which were dynamically discovered at start-up time, are displayed. By selecting the stage, an instance is placed on the canvas. When clicking on the stage, a list of properties, their current value, a description of the stage and properties, and interface definition are provided. The stages of the chain are connected by using the tool found in the top menu bar. Once the chain is complete, it can be rendered by right-clicking on the tail stage and selecting "render." The bottom status section will display progress of the render. In addition, the chain can be saved and loaded, similar to the programmatic method explained previously.

3.2.7 Chain GUI

The Chain GUI tool is similar to the Chain Builder, but it assumes the chain has been programmatically constructed. This tool is used to present stage properties to the user and mainly to visualize outputs of stages. Refer to the full example in the Catena repository: `sfmChainGUI.py`. The chain definition is given in Listing 8.

```
imageSource = Sources.ImageSource(imagePath, "jpg")
sift = FeatureExtraction.Sift(imageSource, False, "SiftHess")
keyMatch = FeatureMatch.KeyMatch(sift, False, "KeyMatchFull")
bundler = BundleAdjustment.Bundler([keyMatch, imageSource])
radialUndistort = Cluster.RadialUndistort([bundler, imageSource])
prepCmvsPmvs = Cluster.PrepCmvsPmvs(radialUndistort, pmvsPath)
cmvs = Cluster.CMVS(prepCmvsPmvs)
pmvs = Cluster.PMVS(cmvs)
```

Listing 8: SfM chain used as an example for the Chain GUI.

The Chain GUI is invoked as shown in Listing 9. A screenshot of the GUI is provided in Figure 3.6, which allows the user to manipulate stage properties, render chains, and visualize output.

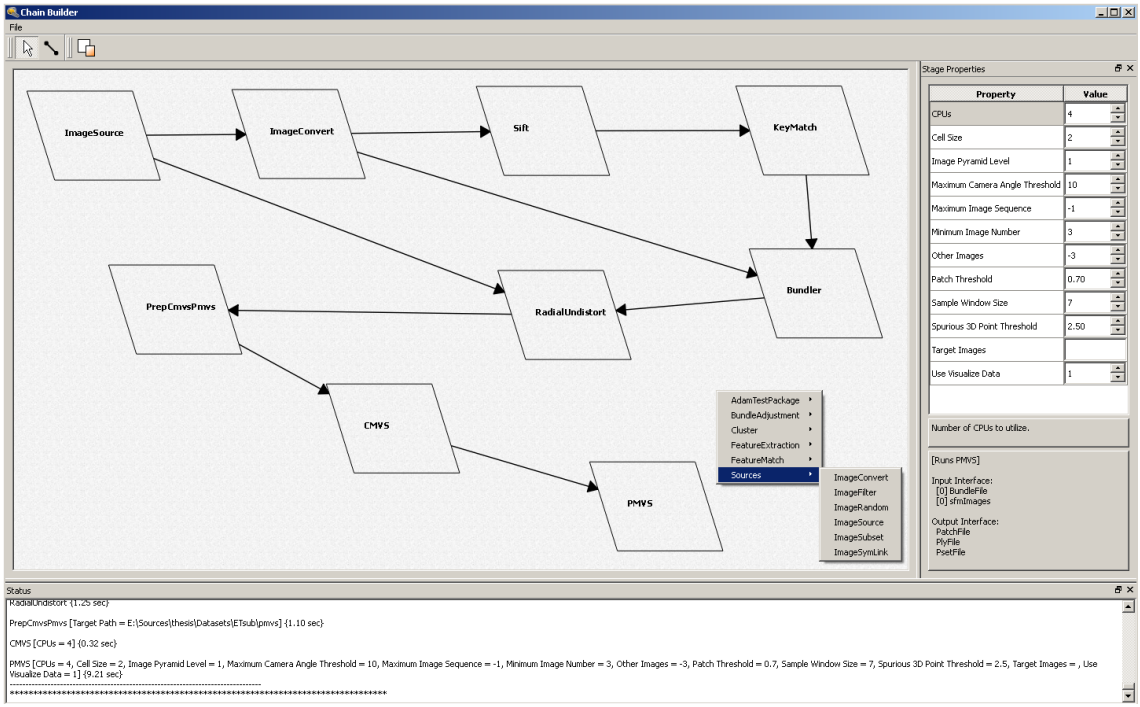


Figure 3.5: The Chain Builder GUI provides a graphical interface to construct workflows, browse stage packages, modify stage properties, and render chains.

```
Visualization.ChainGUI.display(stagesVisualizations,
                               stagesDisplayProperty,
                               stagesPropertyRanges)
```

Listing 9: Example invocation of the Chain GUI visualization tool.

The `stagesVisualizations` list contains 3-tuples. The first tuple element is the Catena stage object(s), followed by a label to be placed on the respective visualization widget, and the class to be used for visualization, as illustrated in Listing 10.

```
stagesVisualizations =
[(imageSource,"Images",Visualization.ChainGUI.ImageWidget),
((imageSource,sift),"Features",Visualization.ChainGUI.FeatureWidget),
((imageSource,keyMatch),"Correspondences",
    Visualization.ChainGUI.CorrespondenceWidget)]
```

Listing 10: The data structure required for stage visualizations using the Chain GUI.

The `stagesDisplayProperty` list contains 2 or 3-tuples. The first tuple element is the Catena stage object(s), followed by a label to be placed on the respective property sheet, and optionally a collection of properties to include on the sheet. This is illustrated in Listing 11. If the third element is not included, all of the properties of the stage will be included in the property sheet.

```
stagesDisplayProperty =
[(imageSource,"Source"),
 (sift,"Features"),
 (keyMatch,"Keymatch"),
 (bundler,"Bundler"),
 (radialUndistort,"Radial Undistort"),
 (prepCmvsPmvs,"Prep CMVS/PMVS"),
 (cmvs,"CMVS"),
 (pmvs,"PMVS")]
```

Listing 11: The data structure required for stage property sheets using the Chain GUI.

The `stagesPropertyRanges` dictionary contains dictionaries that define the property name and range, expressed in a 2-tuple, as shown in Listing 12. If the property has an integer or floating point data type, a slider will be used in conjunction with a spinbox to enforce the defined range.

```
stagesPropertyRanges=
{pmvs:{"Cell Size":(1,40),
       "Maximum Camera Angle Threshold":(1,45),
       "Patch Threshold":(0.0,10.0),
       "Sample Window Size":(1,20)}}
```

Listing 12: The data structure required for stage property ranges using the Chain GUI.

3.2.8 Design Approaches

When a scientist or engineer is presented with a problem, there are three general types of scenarios that should be considered before beginning componentization of the problem and the design and implementation of stages and chains, as listed below. The user should be cognizant of the interfaces between stages in order to develop generalized components that potentially can be leveraged in other applications.

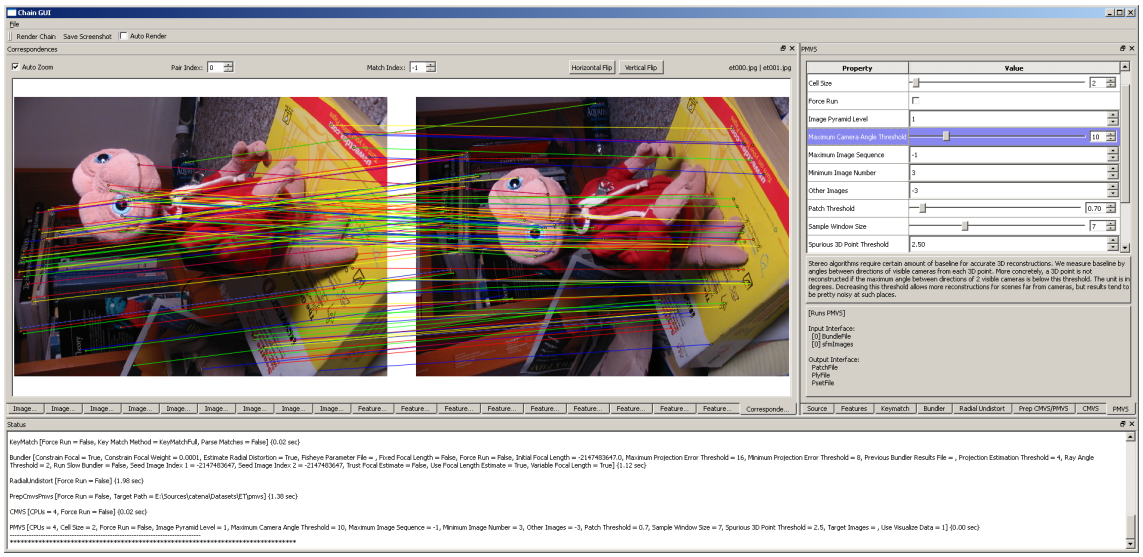


Figure 3.6: The Chain GUI accepts a programmatically constructed chain and provides a graphical interface to modify stage properties, render chains, and most importantly, visualize the outputs of stages. This provides feedback to tune stage parameters.

1. Leverage existing work, no stages need to be developed.

2. A subset of stages are available and/or existing third-party components wish to be leveraged, the overall task flow should be considered and broken up so that existing stages can be leveraged.

3. No stages are available for use.

The first case describes a scenario in which a complete set of stages are available to carry out a task. As such, the chain is constructed either programmatically or by using the Chain Builder GUI.

The second scenario is most probable in practice. Existing stages and/or components are available, but a subset of the functionality needs to be implemented. In this scenario, the existing stage functionality impacts the design of the chain / stage structure.

In the third case, there is complete freedom in defining the workflow breakup. However, it is usually advantageous to construct stages / chains such that they can be reused. This requires forethought and experience in componentization of functionality. It is also important to consider how the individual stages / components will be tested during development.

Mosaicing Example

The following example was taken from a scenario encountered in industry at Exelis, where the existing stages built for the SfM task were leveraged for image mosaicing. An image mosaic is a virtual image that is constructed of many images with some overlapping scene content. The common scene content is exploited to establish correspondences such that the images can be warped into a common coordinate system, resulting in a composite "mosaiced" image. The following mature stages have been implemented and tested for the SfM application, and will directly or indirectly be leveraged for the mosaicing application:

1. Image Source
2. Feature Extraction
3. Feature Matching
4. Bundle Adjustment
5. Point Cloud Generation

In order to facilitate the task of image mosaicing, new data types need to be defined, which relate pairs of images, (`MosaicImagePair`) and a collection of these objects (`MosaicImagePairs`). In order to utilize the existing stages, a new image source that provides `MosaicImagePairs` must be compatible with the next stage down-stream (i.e., Feature Extraction), which requires `sfmImages` as input. This is accomplished by inheriting from `sfmImages` and implementing the accessor methods such that the `MosaicImagePairs` class behaves identically as `sfmImages`. At this point, the Feature Matching stage can be leveraged, therefore no new functionality needs to be implemented. The output from the Feature Matching stage does not provide the exact information required for image mosaicing (with respect to the image pairs). This is the trade-off one needs to make when implementing a new stage from scratch. In this case, it was desirable to leverage existing stage capability at the cost of re-interpreting the output of the Feature Matching stage. This stage provides a match table for every combination of image in the input. In the mosiacing case, there are a limited number of pre-defined image pairs. Analysis was required to determine the benefit between the cost of computing matches for every image combination versus a new feature matching implementation. In this case, it was simple to implement a new stage (`MosaicImageMatching`) to leverage existing functionality of the pre-developed Feature Matching stage. Lastly, a new stage was developed for the chain that performs image mosaicing. This stage accepts the mosaic image pairs collection and feature matches (tie points). It computes homographies (perspective transforms) between pairs, and relates all pairs back to a reference image. This information is used to warp

the images into a common coordinate system resulting in an image mosaic. The complete chain is illustrated in Figure 3.7, the new stages are denoted by a cross-hatch pattern.

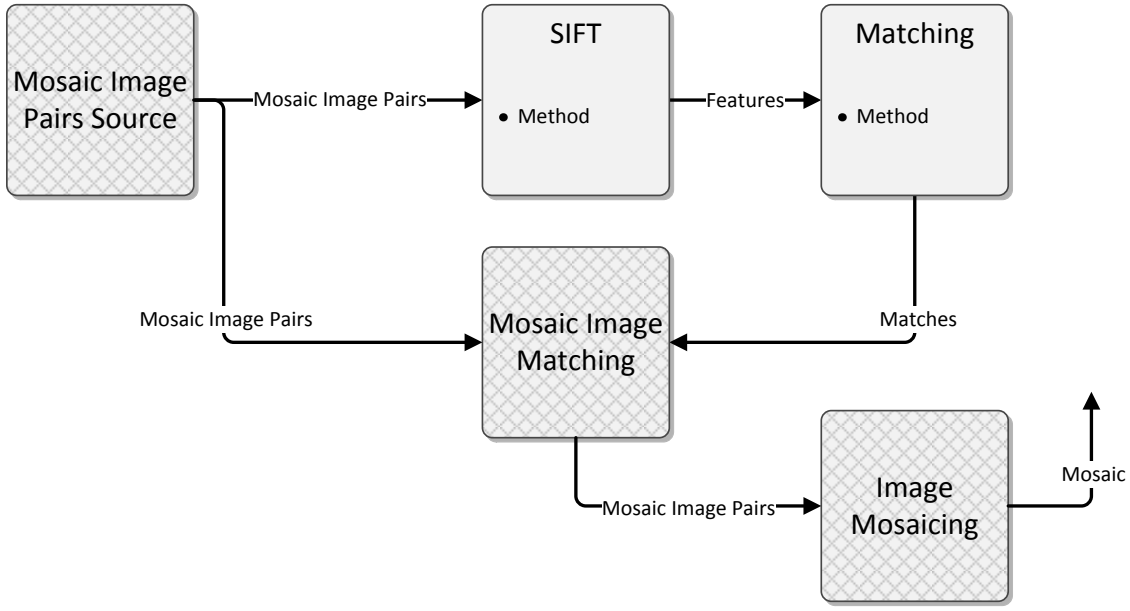


Figure 3.7: The image mosaicing chain uses feature extraction and matching stages from the SfM application to warp images with overlap into a composite virtual image mosaic.

To summarize, the feature extraction and matching stages originally developed for SfM were leveraged and a new mosaic image pairs source was created to facilitate image mosaicing. This is the essence of Catena. The SfM stages were successfully re-used even though the original functionality was not intended for different applications (i.e., mosiacing). Generalization and abstraction are very powerful concepts that enable software reuse.

3.2.9 Interface Definition Advantages

There are many benefits to breaking down workflows in terms of stages / components that define interfaces. Some of these advantages include: the ability to swap stages with those that have equivalent interfaces; ability to benchmark / analyze performance; encapsulation of functionality; componentization of processes; and leveraging components across application specific workflows, among others. The interface definition effectively

enforces structure and clean separation of components. This has the added benefit of being portable, as there are no external dependencies outside of a stage. This can be contrasted to functionality implemented in other environments and frameworks where there are interdependencies between all the components, which makes it extremely difficult to pick up a component and reuse it in another application. Often, functionality is not componentized and there are interdependencies that make it nearly impossible to leverage in other applications. Thus, Catena addresses these problems by enforcing a strict policy that enables scientists / engineers to componentize the desired workflow and encourages development of independent stages with well-defined interfaces that can be easily interchanged for new applications.

3.2.10 Optimization Process

Stages must define an input and output interface, making it very straight-forward to isolate stages from the original chain and import them into a "test bed" chain for profiling and analysis. Catena provides timing information at the stage level, but finer-grained details of the overall timing can be accomplished using native Python timing tools. If the stage is not a pure Python implementation, there are timing and analysis tools for C/C++ and other languages including Intel's VTune [9], Quantify [10], Very Sleepy [11], HPCToolkit [12], and Visual Studio Profiler [13]. These tools will break down the overall program run-time such that time consuming / CPU intensive operations can be isolated and subsequently optimized. This process deserves a much deeper treatment and is beyond the scope of this thesis. However, there are CPU vector extensions, graphics processing units (GPUs), field programmable gate arrays (FPGAs), digital signal processors (DSPs), distributed computing facilities, and other techniques using cooperative processing in order to optimize algorithm implementations.

3.2.11 Platform / Implementation Issues

Catena was implemented using Python (developed against v2.7.3). The framework and base stages have been tested on multiple platforms including many Linux distributions, Mac OS X, and 32/64-bit Windows. Python is inherently cross-platform, but the programmer must be cognizant to ensure platform independence. If native external libraries or applications are used as part of the stage implementation, they will have to be built for each desired platform. For example, in the SfM application, many native open-source components exist. This is generally the case when leveraging existing work. Additionally, native implementations are more efficient, and Python implementations have been optimized for speed in a language such as C++. The build tool CMake [14] attempts to generalize platform-specific build issues and allows for easy cross-platform compilation of

native libraries and applications. This build tool is strongly recommended.

Often there are many algorithms of the same basic class that are subtly different. For example, the feature extractors that are used for the SfM application can all be viewed as the same at a certain level of abstraction. Object-oriented software design can be utilized to account for these differences while maintaining a consistent interface and single component stage. First, all specific components of interest should be factored into a common class representation (e.g., `KeypointDescriptorFile`). The class should contain general properties and methods applicable to all components. In addition, any underlying data types should be defined and used throughout the classes. Once this is established, concrete classes that derive from the base class (e.g., `KeypointDescriptorFileVLFeat`) should be implemented to carry out the specific component's task. A Catena stage can be wrapped around this functionality, exposing general properties and a mode to select the desired implementation. Since the component was abstracted, the interface and the datatypes declared therein are by default generalized. Therefore, the stage is compatible with any stages that match the defined input/output interface.

3.2.12 Deployment Considerations

It is commonly desirable to distribute chains to target heterogeneous systems. Therefore, considerations must be made for successful deployment. First, it is important to test the execution of the chain on the target platform to ensure all stages operate as intended and that the underlying native components function properly. Ideally, unit tests are written to automate the testing of the stages developed. Experience has shown that it is common for library dependencies to be missing, especially on Unix operating systems, as shared objects are often added via package managers after the base installation. The developer might have to rebuild applications and libraries, or install missing dependencies via package managers or installers. There are Python tools such as Freeze [15] and py2exe [16] that turn Python modules, scripts, and dependencies into either a single executable or portable package. The developer should consider using these tools to aid in workflow deployment, as this simplifies the deployment process.

Chapter 4

SfM Theory

The SfM algorithm processes multi-view imagery of a scene and generates 3D point clouds and models. This chapter outlines the SfM theory by addressing the individual components required in the processing chain. A description of each component and justification for its utilization is presented, along with the underlying mathematics. Additionally, the generalization of each component will illustrate how it can be wrapped into a Catena stage for integration into the framework.

4.1 Image Source

The first step in the SfM process is to define the input images that will be used for the 3D reconstruction. While it is possible to implement a stage that includes images from various sources, different directories, etc., the assumption is made that all the images will reside in a given directory and have the same image file format / file extension. This allows for the implementation of a generalized image source that requires a path, file extension, and an optional parameter that defines the *focal pixel* value of the image set, as illustrated in Figure 4.1.

The focal pixel value is a requirement of the bundle adjustment stage further down the chain, but it is appropriate to associate it with the image source. It is optional because in some cases, the pixel pitch and focal length can be found in the exchangable image file format (exif) metadata, which are sufficient for the calculation of the focal pixel value (see Equation 4.1).

$$\text{focalPixels} = \frac{\text{sensorResolution[pixels]} * \text{focalLength[mm]}}{\text{sensorWidth[mm]}} \tag{4.1}$$

It is common for manufacturers to specify the "pixel pitch" of a sensor, which is the physical size of a single pixel / detector. From the focal pixels equation, the *sensor resolution* and *width* terms can be represented by pixel pitch, as shown in Equation 4.2.

$$\text{pixelPitch} = \frac{\text{sensorWidth[mm]}}{\text{sensorResolution[pixels]}} \tag{4.2}$$

By substituting Equation 4.2 into Equation 4.1, a more intuitive definition of focal pixels is obtained (see Equation 4.3).

$$\text{focalPixels} = \frac{\text{focalLength[mm]}}{\text{pixelPitch[mm/pixel]}} \tag{4.3}$$

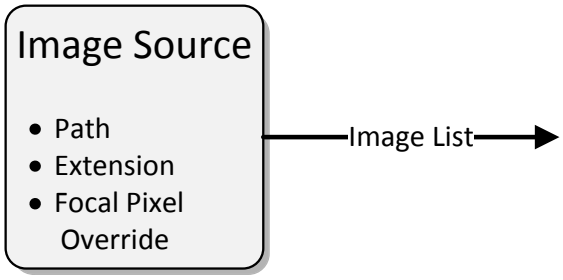


Figure 4.1: The Image Source stage generates a list of images from a given path and extension. This is typically the first stage in a chain.

4.2 Image Filtering / Subsets

Due to the fact that the entire directory of images was input by the image source, it is convenient to develop stages that filter or generate subsets of the complete image set. There are two stages that perform these functions in the framework, *ImageFilter* and *ImageSubset*. Additional stages can be easily implemented to down-select from the initial, complete set of images. For example, the *ImageSubset* stage is shown in Figure 4.2, which will be used to select a subset of images from the complete set.

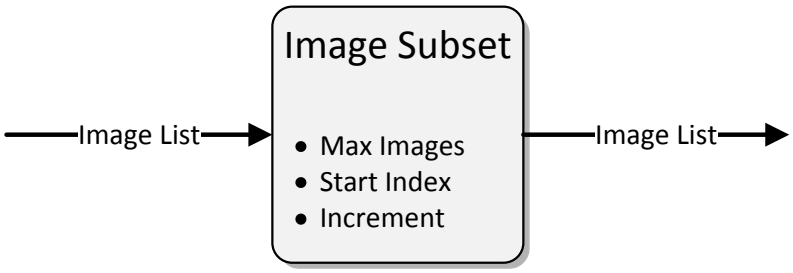


Figure 4.2: The Image Subset stage takes a list of images and down-selects using the supplied criteria.

4.3 Symbolic Links

Further down the chain, stages generate additional files in the directory where the image files exist. Therefore, it is desirable to create a directory, which contains symbolic links to images. However, this is currently only possible in Unix environments, due to limitations of other operating systems. This leaves the original directory of images pristine while establishing a "working directory" for the SfM process. The *ImageSymLink* stage provides this function, as shown in Figure 4.3.

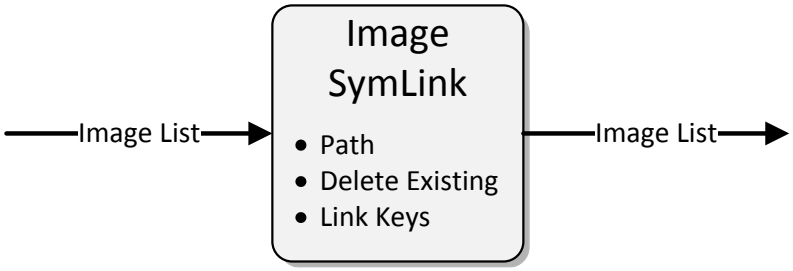


Figure 4.3: The Image Symbolic Link stage creates symbolic links to images on Unix operating systems to establish a working directory.

4.4 Image Conversion

Sometimes it is necessary to convert images into different image file formats due to input requirements of applications. Additionally, there are cases when the bit-depth, dynamic range, or data type of the imagery is incompatible or sub-optimal for processing by subsequent stages. For example, feature detectors typically operate well on 8-bit imagery with high dynamic range. Therefore, a conversion process needs to be performed in order to convert the imagery into a proper form. The *ImageConvert* stage (see Figure 4.4) in

the Catena framework utilizes the Python imaging library (PIL) and performs image file format conversion. This stage can also perform colorspace transformation from RGB to grayscale, if desired.

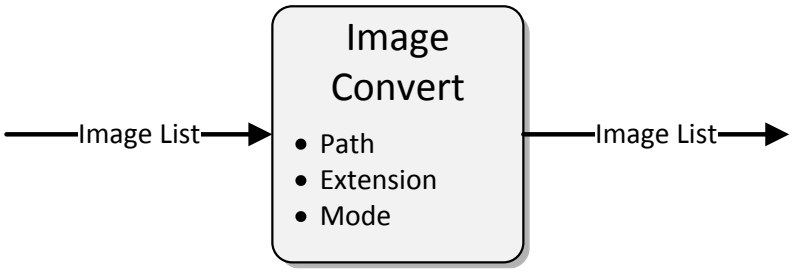


Figure 4.4: The Image Convert stage converts images to a desired file format.

4.5 Feature Extraction

The goal of the feature extraction step is to identify salient points or regions in an image that can be uniquely identified across imaging conditions (e.g., in varying illumination conditions, views / perspectives, and imaging modalities). A strong feature descriptor should be robust such that it can be easily differentiated among a large set of features. Characteristics of the points need to be computed in order to successfully establish correspondence in the following stage. Typically, there are attributes associated with each point, including scale, orientation, response, and a feature vector.

There are a variety of feature extraction algorithms available that attempt to establish an invariant feature space, including SIFT [1], ASIFT [17], SURF [18], STAR [19], BRISK [20], MSER [21], Daisy [22], FAST [23], ORB [24], and Harris Corners [25], among others. The algorithms define a multi-dimensional (typically 128 dimensions) feature space in which keypoints are established. The goal of the feature space is to be invariant to imaging conditions / distortions and provide an optimal space to establish correspondence of the next step in the SfM workflow.

4.5.1 SIFT

The SIFT [1] algorithm is the most commonly used feature extractor in the SfM application and many of the other feature extraction algorithms are heavily rooted in SIFT. First, the number of *octaves* and *intervals per octave* (or simply *intervals*) are selected. Each octave is a scale level of the image established by down-sampling or up-sampling by a factor of 2x.

This contributes to the scale-invariance of SIFT. Typically, the input image is up-sampled by 2x to establish the first octave. In most SIFT implementations the number of octaves is calculated using Equation 4.4.

$$\text{octaves} = floor(log(min(\text{imageWidth}, \text{imageHeight}))/log(2.0) - 2) \tag{4.4}$$

For example, if the sensor resolution is 1280x1024:

$$\begin{aligned}
\text{octaves} &= floor(log(min(1280, 1024))/log(2.0) - 2) \\
\text{octaves} &= floor(log(1024)/log(2.0) - 2) \\
\text{octaves} &= floor(3.01/0.30 - 2) \\
\text{octaves} &= floor(8.03) = 8
\end{aligned}$$

Therefore, the image pyramid would contain the following eight scales (octaves): 2560x2048, 1280x1024, 640x512, 320x256, 160x128, 80x64, 40x32, 20x16.

The number of intervals is typically set to three. The algorithm accepts a grayscale image and is up-sampled by 2x and filtered with a Gaussian kernel, typically where $\sigma = 1.6$ is used to generate the kernel. Next, the Gaussian intervals are computed for each octave, illustrated in Figure 4.5. Each interval is generated by convolving with the next Gaussian kernel in the filter bank, and is generated using Equation 4.5 (from the Hess implementation [26]):

$$\sigma^2_{total} = \sigma^2_i + \sigma^2_{i-1} \tag{4.5}$$

Next, the Difference of Gaussian (DoG) images are calculated by simply subtracting interval pairs of images in the Gaussian octave stack, as illustrated in Figure 4.5. The "scale-space extrema" are identified by iterating over all pixels in the DoG images, for each octave and interval. First, the pixel value is tested against a contrast threshold (e.g., 0.04), checking for a minimum signal level. If it passes this criterion, it is checked to see whether it is a maximum in the current interval's 8-point neighborhood and the 9-point neighborhood of the two adjacent intervals, as illustrated in Figure 4.6.

Finally, the "ratio of principal curvatures" is computed and checked against a threshold. This eliminates points that have a large response in only one of the vertical and horizontal directions. The "principal curvature" is computed at the location and octave of the detected feature by first calculating the 2x2 Hessian using the DoG image already computed using Equation 4.6.

$$\mathbf{H} = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{bmatrix} \tag{4.6}$$

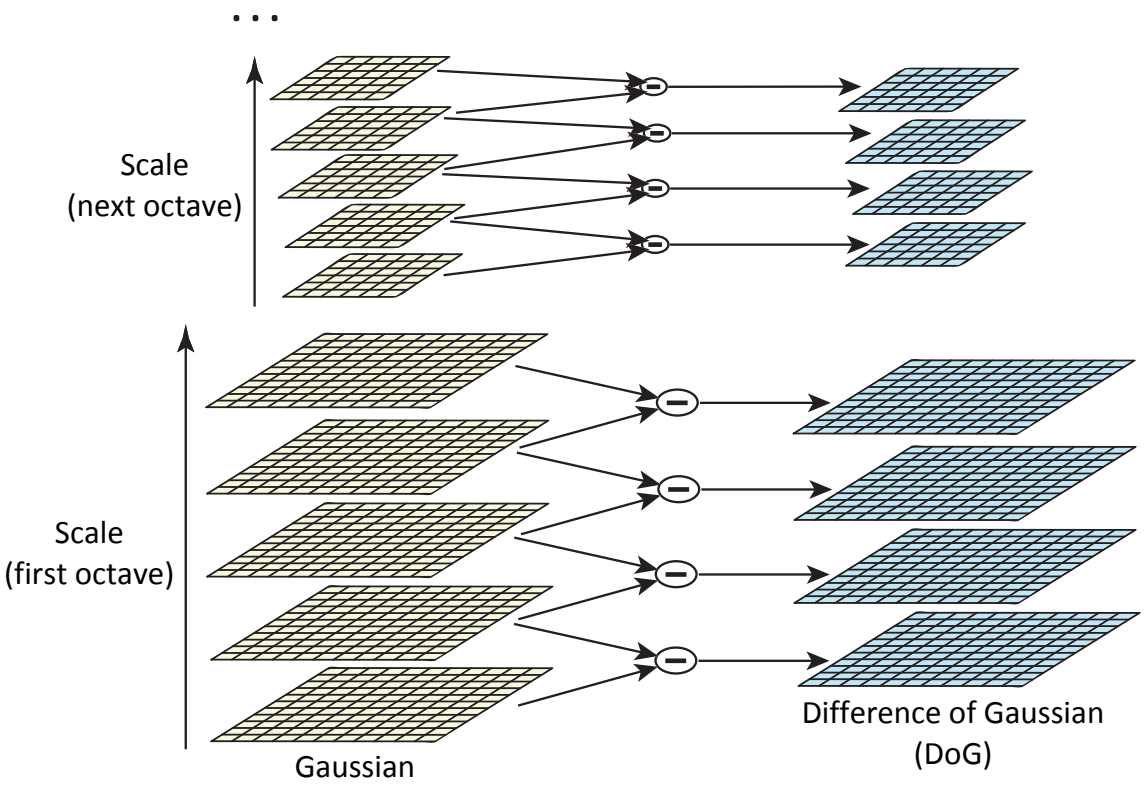


Figure 4.5: The SIFT algorithm establishes a difference of Gaussian scale space by convolving images at each octave to yield intervals. Reproduced from [1].

The trace and determinant (calculated in Equations 4.7 and 4.8) are then used to compare against a threshold r, which is typically set to 10. If the feature passes the condition in Equation 4.9, it is selected as a valid SIFT feature.

$$\mathrm{Tr}(\mathbf{H}) = D_{xx} + D_{yy} \tag{4.7}$$

$$\mathrm{Det}(\mathbf{H}) = D_{xx}D_{yy} - (D_{xy})^2 \tag{4.8}$$

$$\frac{\mathrm{Tr}(\mathbf{H})^2}{\mathrm{Det}(\mathbf{H})} < \frac{(r+1)^2}{r} \tag{4.9}$$

At this point, the salient points in the image are identified. The feature is located to sub-pixel accuracy by interpolating in the DoG scale space. The scale and orientation

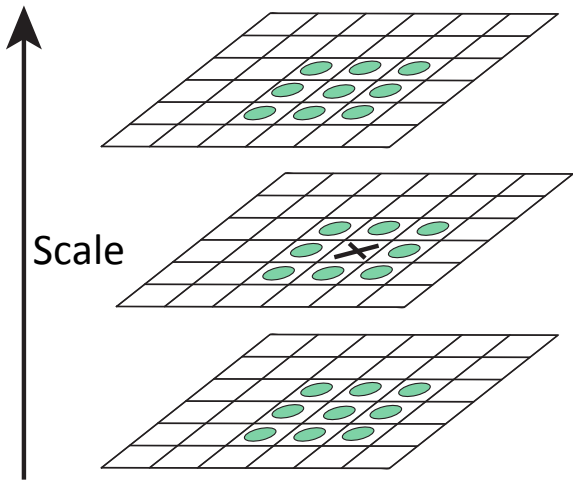


Figure 4.6: The SIFT algorithm finds scale space extrema by comparing the 8-point neighborhood at the current scale and 9-point neighborhood of adjacent scales. Reproduced from [1].

values are assigned to the feature, establishing scale and orientation invariance. The scale value is used to select the Gaussian image, for which the orientation will be computed. Orientation histograms are computed typically using a 36 bin histogram representing all the orientations around a feature. The magnitude and orientation are computed using Equations 4.10 and 4.11, respectively, where L represents the Gaussian image.

$$m(x,y) = \sqrt{(L(x+1,y) - L(x-1,y))^2 + (L(x,y+1) - L(x,y-1))^2} \tag{4.10}$$

$$\theta(x,y) = \tan^{-1}((L(x,y+1) - L(x,y-1))/(L(x+1,y) - L(x-1,y)) \tag{4.11}$$

The magnitude is scaled by a Gaussian-weighted circular window ($\sigma = 1.5$ times the scale of the keypoint) and added to the orientation histogram. This effectively provides a measure of the response at different angles around the feature. The histogram peak is detected and the corresponding orientation is assigned to the feature. If there are peaks within 80% of the global peak, they are also used to instantiate new features. This means that multiple features could be generated for a single coordinate, but with different orientations.

Finally, the descriptor vector is calculated by first locating the feature by rotating it according to the computed orientation. The magnitudes and orientations are computed around the keypoint, using Equations 4.10 and 4.11. The magnitude is modulated by a

Gaussian weighting function using $\sigma = d/2$, where d is the width of the descriptor window. This gives gradients farther from the feature center less contribution. The neighborhood size is variable, but typically a 16x16 neighborhood is used to compute the descriptor. The 16x16 neighborhood is broken into 4x4 windows, where 16 separate magnitude / orientation histograms are computed for each window. Next, the histogram is quantized into a desired number of bins, typically 8. Each of the 8 bins of the 16 histograms comprises one element of the feature vector. In the typical case, the vector has 128 dimensions, but this varies depending on the choice of the neighborhood and histogram bin size. The descriptor vector provides illumination and perspective invariance to the feature (see Figure 4.7).

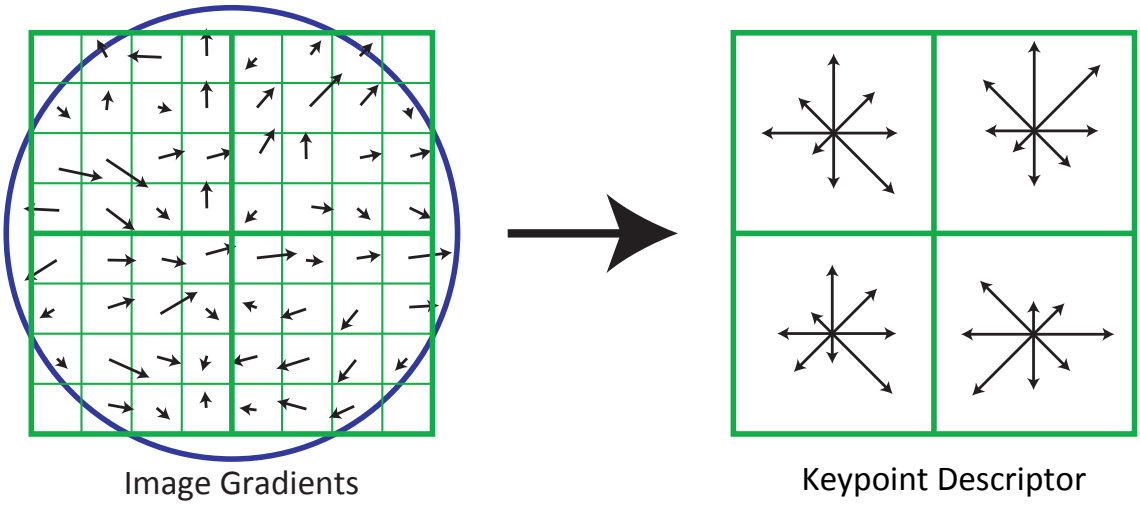


Figure 4.7: The SIFT descriptor is computed from localized gradients combined into localized histograms of magnitudes and orientations. Reproduced from [1].

There are a variety of SIFT implementations available including: SiftWin32 (base implementation from Lowe) [1], VLFeat [27], OpenCV [28], OpenSIFT [26], and SIFT GPU [29]. In order to provide a generalized SIFT component (shown in Figure 4.8), the intricacies of each implementation need to be addressed. There are two main issues that require attention: 1) depending on the implementation, there are subtle differences in how the application is executed, 2) the output from the implementation (i.e., the keypoint descriptors), need to be in different file formats. Therefore, the classes shown in Figure 4.9 were developed to allow for specific parsing of the file formats, while providing methods to output a standardized format. The choice was made to standardize on Lowe's file format.

Therefore, it is possible to invoke any of the SIFT implementations and convert their output to the Lowe format.

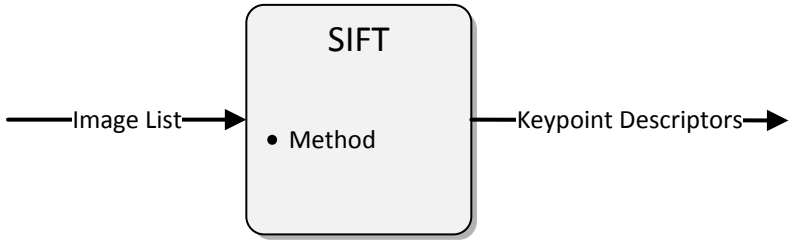


Figure 4.8: The SIFT Stage implements the SIFT algorithm, generating keypoint descriptors for each of the images provided.

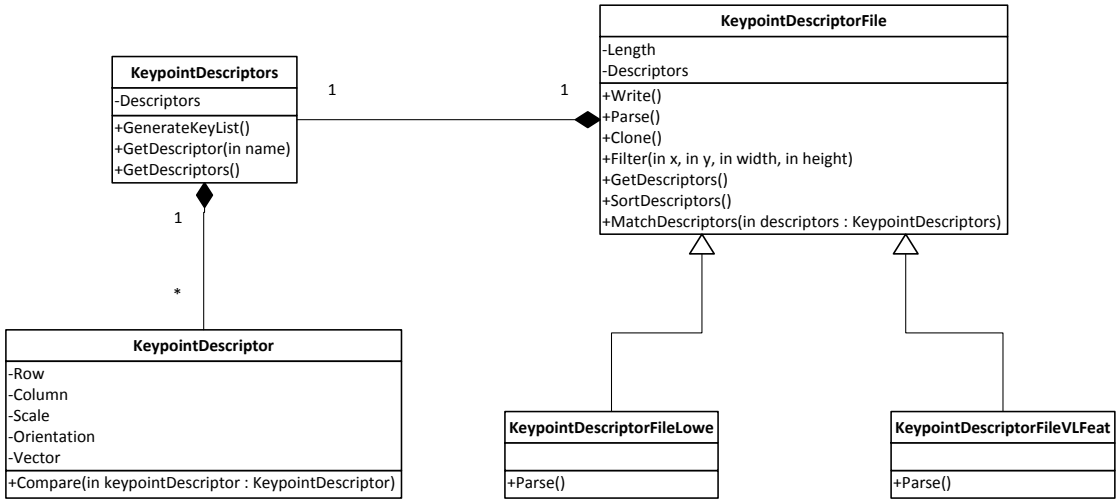


Figure 4.9: Keypoint descriptor class diagram illustrating the power of object-oriented design and polymorphism to handle multiple keypoint descriptor file formats.

4.6 Feature Matching / Filtering

After feature points are generated for a set of images, it is necessary to construct a table of matches. In the SfM domain, it is desirable to identify matches between every pair

of images in the set, as two-view geometry is used to relate the views / images. The feature vector, or signature of the keypoint, is used in conjunction with other metadata (scale, orientation, response) to establish correspondences. If the feature space is perfectly invariant, a feature vector for a keypoint from one view should have the same feature vector from another view. However, in practice this is typically not the case, therefore, thresholds must be employed when comparing feature vectors and metadata. For example, a feature matching algorithm could compare the keypoint orientation from one image and identify keypoint candidates from another image by employing a threshold of $\pm 15^{\circ}$ on the orientation. Next, the dot product between the two feature vectors from the respective keypoints are computed and compared against a threshold. In the case of SIFT, a 128-dimensional feature vector is typically used for keypoints, so the dot product represents the Euclidean distance between the two keypoints in a 128-dimensional vector space.

At this point, a table of putative matches has been established using only the feature vector and metadata. It is often desirable to fit the data to a model and filter outliers (e.g., using a technique such as random sample consensus (RANSAC) [30]). This algorithm assumes a model (e.g., perspective transform or homography) can be used to represent the transformation between views. A random subset of the matches are used to compute a transform using the direct linear transformation (DLT) algorithm (explained in Section 4.7.5). A fitness score is assigned by comparing the difference of a correspondence taken from the match table versus the transformed point through the model. After many iterations of this process, outliers are identified and filtered.

There are two feature matching implementations provided by the Catena framework: KeyMatchFull [6] and key matching provided in the SiftGPU [29] library. Since the SiftGPU library requires the implementation of an application layer, it was intentionally implemented to mimic the KeyMatchFull interface. Therefore, there are no considerations made at the workflow layer to account for differences, other than the invocation of the desired implementation via a "method" property (see Figure 4.10).

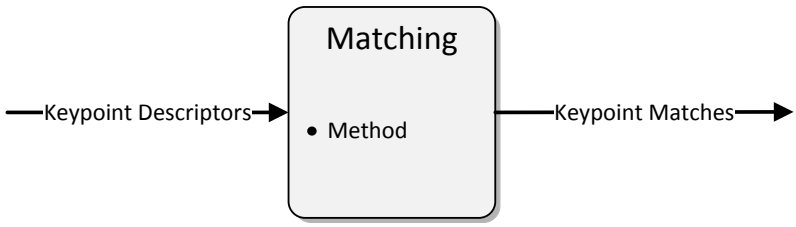


Figure 4.10: The Feature Matching stage generates a match table for every image combination using the independently generated keypoint descriptors.

4.7 Image-Based Geometry Estimation

It is possible to derive the camera matrices by exploiting epipolar geometry to estimate the 3D scene using the previously computed correspondences through a bundle adjustment process. A survey of multi-view geometry provides sufficient background information necessary for the component theory used for the image-based geometry estimation step.

4.7.1 Camera Model

A *pinhole camera model* will be derived and used throughout this section. The interior or intrinsic characteristics of a camera can be expressed in terms of the following parameters:

f:	focal length
s_θ:	skew
s_x, s_y:	2D scaling (number of pixels per unit distance)
c_x, c_y:	2D camera center

The intrinsic parameters can be composed in matrix form, as shown in Equation 4.12.

$$\mathbf{K} = \begin{bmatrix} fs_x & s_\theta & c_x \\ 0 & fs_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{4.12}$$

The exterior or extrinsic parameters describe the camera pose (i.e., how the camera is positioned in 3D space). The pose can be represented by three parameters: θ, t_x, and t_y, expressing the camera rotation and translation. These parameters can be composed in matrix form using Equations 4.13 and 4.14.

$$\mathbf{R} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \tag{4.13}$$

$$\mathbf{t} = \begin{bmatrix} t_x \\ t_y \end{bmatrix} \tag{4.14}$$

It is important to note that the pinhole camera model is unable to represent the radial lens distortion normally present in real-world imaging systems. The radial distortion effect needs to be removed, especially in close-range imaging scenarios. Hartley and Zisserman [2] introduce a lens distortion factor $L(r)$, where r is the radial distance ($\sqrt{x^2 + y^2}$) from the camera center. The corrections for each dimension are expressed in Equation 4.15.

$$\hat{x} = x_c + L(r)(x - x_c) \qquad \hat{y} = y_c + L(r)(y - y_c) \tag{4.15}$$

where,

x, y:	2D measured coordinate
$\hat{x}$, $\hat{y}$:	2D corrected coordinate
x_c, y_c:	2D radial distortion center

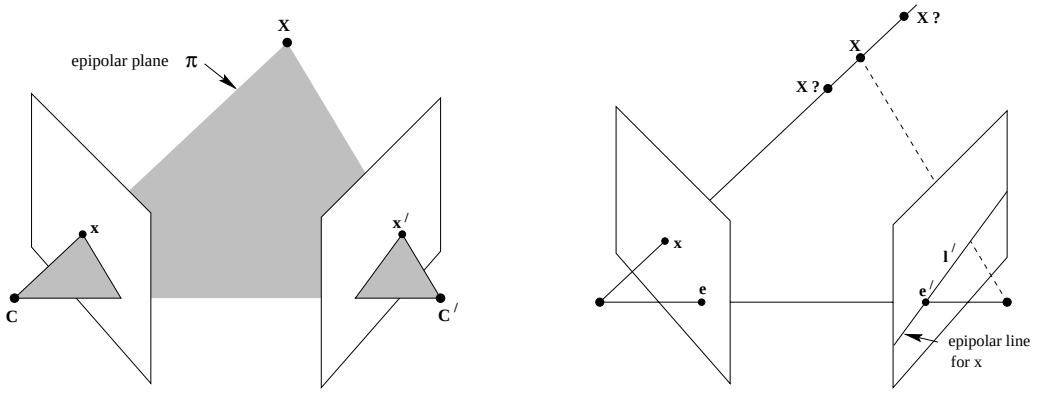
It is shown that $L(r)$ is only defined for positive r values and $L(0) = 1$. This allows for the Taylor expansion given in Equation 4.16.

$$L(r) = 1 + k_1 r + k_2 r^2 + k_3 r^3 + \dots \tag{4.16}$$

The intrinsic camera model can be extended with the radial distortion parameters: $k_1, k_2, k_3, \dots, x_c, y_c$. The compensation of radial lens distortion will be addressed in Section 4.8 of the SfM workflow to warp the imagery so that it is rectified for subsequent stages.

4.7.2 Epipolar Geometry

The epipolar geometry between two views states that for a given 3D point ($\mathbf{X}$), the corresponding image points ($\mathbf{x}$ and $\mathbf{x}'$), camera centers ($\mathbf{C}$ and $\mathbf{C}'$), and $\mathbf{X}$ lie on a common plane (π). Figure 4.11a illustrates the geometry and relationships. Figure 4.11b shows that for the image point ($\mathbf{x}$), its projection along the ray from $\mathbf{C}$ can take on any $\mathbf{X}$, which are imaged as a line in the second view, $\mathbf{l}'$. The 3D point imaged in the first view as $\mathbf{x}$, must lie on the epipolar line (for $\mathbf{x}$) in the second view, $\mathbf{l}'$. This is defined as the *epipolar line for* $\mathbf{x}$. Also, the epipole represents the image of the second view's camera center. This is advantageous since correspondences have already been established between views. The epipolar line constrains the possible image of $\mathbf{X}$ in the second view to a 2D line. This property will be exploited in the bundle adjustment process.



(a) The 3D point $\mathbf{X}$, the two imaged points, $\mathbf{x}$ and $\mathbf{x}'$ through the camera centers, $\mathbf{C}$ and $\mathbf{C}'$, lie in the epipolar plane π.

(b) The epipolar line ($\mathbf{l}'$) of $\mathbf{x}$, from the first view is defined as the 2D line in the second view from which all possible projections of the 2D point $\mathbf{x}$ along the ray that trace the 3D point $\mathbf{X}$ in the projection on the second view.

Figure 4.11: Epipolar geometry illustrations for point correspondences. Reproduced from Hartley and Zisserman [2].

4.7.3 Fundamental Matrix

The *fundamental matrix* can be derived using relationships established from epipolar geometry. Figure 4.12 illustrates the homography ($\mathbf{H}_\pi$) that transforms an image point from the first view ($\mathbf{x}$) to the corresponding image point in the second view ($\mathbf{x}'$). This is carried out by mapping through the epipolar plane π. The epipolar line of $\mathbf{x}$ can be expressed as the cross-product of the second view's epipole ($\mathbf{e}'$) with the image of $\mathbf{X}$ in the second view, $\mathbf{x}'$ (see Equation 4.17). Section A.1 illustrates the derivation of a cross-product using matrix multiplication.

$$\mathbf{l}' = \left[\mathbf{e}'\right]_{\mathrm{x}} \mathbf{x}' \tag{4.17}$$

However, $\mathbf{x}'$ can be expressed as the projection of $\mathbf{x}$ through H_π, i.e $\mathbf{x}' = \mathrm{H}_\pi\mathbf{x}$, as shown in Equation 4.18.

$$\mathbf{l}' = \left[\mathbf{e}'\right]_{\mathrm{x}} \mathbf{H}_\pi\mathbf{x} \tag{4.18}$$

This establishes the definition of the fundamental matrix in terms of the epipole and projection from the epipolar plane (see Equation 4.19).

$$\mathbf{F} = \left[\mathbf{e}'\right]_{\mathrm{x}} \mathbf{H}_\pi \tag{4.19}$$

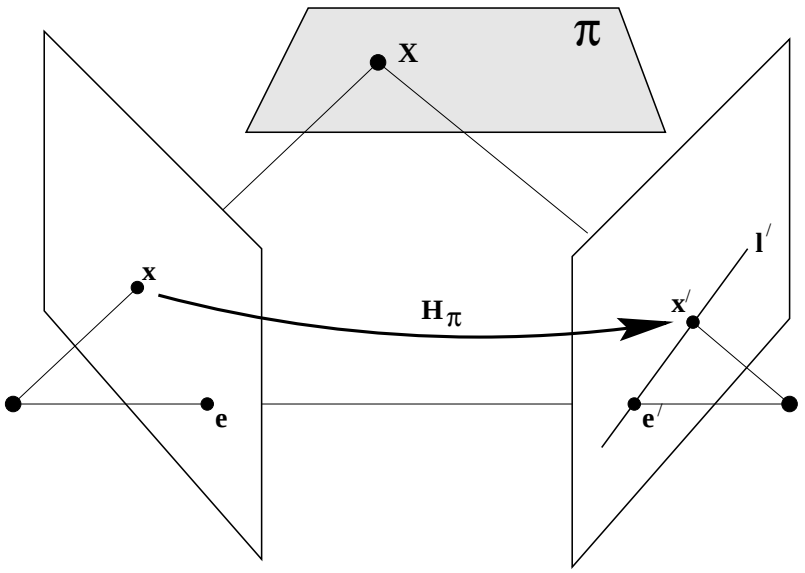


Figure 4.12: Image coordinates can be projected using the epipolar homography: $\mathbf{x}' = \mathbf{H}_\pi \mathbf{x}$. Reproduced from Hartley and Zisserman [2].

4.7.4 Fundamental Matrix Extensions Using Projective Transforms

Using the pre-defined pinhole camera model from Section 4.7.1, the projective camera model is defined as the product of the intrinsic and extrinsic camera parameters (see Equation 4.20).

$$\mathbf{P} = \mathbf{K}\,[\mathbf{R}|\mathbf{t}] \tag{4.20}$$

$\mathbf{P}$ can be used to project 3D points into the 2D coordinate system of the camera. The pixel coordinate on the focal plane, $\mathbf{x}$ is the projection, or image of $\mathbf{X}$ (see Equation 4.21).

$$\mathbf{x} = \mathbf{P}\mathbf{X} \tag{4.21}$$

Furthermore, a homography can be computed from two camera matrices by employing the pseudo-inverse of $\mathbf{P}$ (see Equation 4.22).

$$\mathbf{H} = \mathbf{P}'\mathbf{P}^{\dagger} \tag{4.22}$$

This effectively transforms coordinates from one camera's coordinate system to the other, as given in Equation 4.23. Namely, the first camera's point ($\mathbf{x}$) is inversely projected to the 3D coordinate system through $\mathbf{P}^{\dagger}$, resulting in $\mathbf{X}$, then projected into the second camera's coordinate system using $\mathbf{P}'$.

$$\mathbf{x}' = \mathbf{H}\mathbf{x} \tag{4.23}$$

Combining the homography formulated as the cross-product of the epipole and the product of two camera projection matrices (previously defined in Equation 4.19), the epipolar line can be expressed according to Equation 4.24.

$$\mathbf{l}' = \left[\mathbf{e}'\right]_{\mathrm{x}} (\mathbf{P}'\mathbf{P}^{\dagger})\mathbf{x} \tag{4.24}$$

This allows for the expression of the fundamental matrix in terms of the epipole and camera projection matrices (see Equation 4.25).

$$\mathbf{F} = \left[\mathbf{e}'\right]_{\mathrm{x}} \mathbf{P}'\mathbf{P}^{\dagger} \tag{4.25}$$

It is also worth noting that the dot-product of $\mathbf{x}'$ with the projection of $\mathbf{x}$ through the fundamental matrix yields zero, as shown in Equation 4.26. However, this assumes perfect correspondence, which is never found in practice. As explained in Section 4.6, thresholds need to be employed when exploiting this property for feature matching.

$$\mathbf{x}'^{T}\mathbf{F}\mathbf{x} = 0 \tag{4.26}$$

Essential Matrix

A useful extension to the fundamental matrix is the *essential matrix*. Placing one of the cameras at the origin (i.e., $\mathbf{P} = [\mathrm{I}|\mathbf{0}]$), allows for a pair of normalized cameras, where the first is *identity* and the second is expressed using Equation 4.27.

$$\mathbf{P}' = [\mathbf{R}|\mathbf{t}] \tag{4.27}$$

The extension from Equation 4.26 yields: $\hat{\mathbf{x}}'^{T}\mathbf{E}\hat{\mathbf{x}} = 0$, and the relationship with the fundamental matrix is shown in Equation 4.28.

$$\mathbf{E} = \mathbf{K}'^{T}\mathbf{F}\mathbf{K} \tag{4.28}$$

4.7.5 Fundamental Matrix From Correspondences

The *normalized 8-point algorithm* can be used to calculate the fundamental matrix from correspondences, according to Equation 4.26, using the DLT. The first step is to normalize all the coordinates such that the centroid of the points is shifted to the origin and the standard deviation from the origin is $\sqrt{2}$, expressed by the two transforms in Equation 4.29.

This process conditions the problem, making finite-precision computation of the singular value decomposition (SVD) numerically stable.

$$\hat{\mathbf{x}}_i = \mathbf{T}\mathbf{x}_i \qquad \hat{\mathbf{x}}'_i = \mathbf{T}'\mathbf{x}'_i \tag{4.29}$$

Next, the point correspondences are used to solve for the fundamental matrix. The column vectors of homogeneous point correspondences can be expressed as: $\mathbf{x} = [x, y, 1]^T$ and $\mathbf{x}' = [x', y', 1]^T$. Using the definition of the fundamental matrix in Equation 4.26, the product of the correspondences with the fundamental matrix should equal 0, as shown in Equation 4.30.

$$x'xf_{11} + x'yf_{12} + x'f_{13} + y'xf_{21} + y'yf_{22} + y'f_{23} + xf_{31} + yf_{32} + f_{33} = 0 \tag{4.30}$$

This can also be expressed as a vector inner-product, as shown in Equation 4.31.

$$[x'x, x'y, x', y'x, y'y, y', x, y, 1]\mathbf{f} = 0 \tag{4.31}$$

With a set of N correspondences, a set of linear equations can be formed (see Equation 4.32).

$$\mathbf{A}\mathbf{f} = \begin{bmatrix} x'_1x_1 & x'_1y_1 & x'_1 & y'_1x_1 & y'_1y_1 & y'_1 & x_1 & y_1 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x'_nx_n & x'_ny_n & x'_n & y'_nx_n & y'_ny_n & y'_n & x_n & y_n & 1 \end{bmatrix} \mathbf{f} = \mathbf{0} \tag{4.32}$$

The $\mathbf{A}$ matrix must have a rank 8 for an exact solution (up to a scale, given homogeneous equations), otherwise the least-squares solution is obtained when the rank is greater than 8 (i.e., more than 8 correspondences that are not co-linear).

Next, the SVD of $\mathbf{A}$ is calculated using Equation 4.33.

$$\mathrm{SVD}(\mathbf{A}) = \mathbf{U}\Sigma\mathbf{V}^{\mathrm{T}} \tag{4.33}$$

In the right null-space of $\mathbf{A}$, the last column of $\mathbf{V}$ contains the solution for $\mathbf{f}$. The 9-element vector is reformed into a 3x3 matrix that yields the fundamental matrix in the normalized space, $\hat{\mathbf{F}}$. The matrix can be denormalized by applying the inverse transforms from Equation 4.29, resulting in Equation 4.34.

$$\mathbf{F} = \mathbf{T}'^{\mathrm{T}}\hat{\mathbf{F}}\mathbf{T} \tag{4.34}$$

4.7.6 Iterative Fundamental Matrix Computation

The tools developed thus far can be used to robustly estimate the fundamental matrix between image pairs by employing the following general algorithm:

1. Perform feature extraction on images
2. Perform feature matching on all image pairs, establish putative correspondences
3. Compute the fundamental matrix, employing the RANSAC algorithm to filter inconsistent correspondences
4. Perform non-linear optimization on all Fundamental Matrices, resulting in a globally consistent set of Fundamental Matrices

The Bundler [6] software employs this type of algorithm to calculate the camera matrices for a set of views / images. From a high level, Bundler accepts correspondences from image pairs, performs a bundle adjustment, and generates a "Bundle" file, which contains camera projection matrices and 2D to 3D point correspondences, along with a sparse point cloud in PLY format (see Figure 4.13). Bundler uses the sparse bundle adjustment (sba) [31] library to perform bundle adjustment, which implements the Levenberg-Marquardt (L-M) non-linear optimization algorithm [32]. It is worth noting that a modern bundle adjustment library has been developed by Google called Ceres Solver [33]. This provides a generalized non-linear optimization solution with a much cleaner and intuitive application programming interface (API).

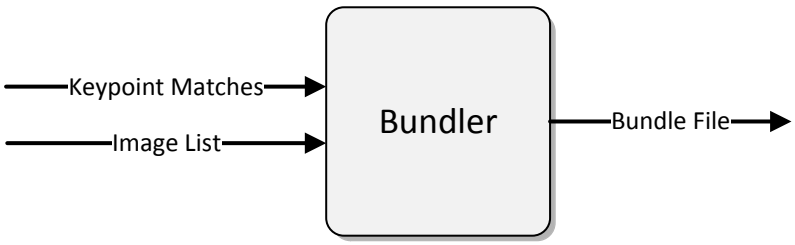


Figure 4.13: The Bundler Stage performs a bundle adjustment process using keypoint matches and images in order to generate consistent camera matrices, sparse point cloud, and 2D to 3D point correspondences.

4.7.7 Triangulation & Point Cloud Generation

Given a set of camera projection matrices and 2D point correspondences, it is desirable to compute 3D points using a *triangulation* algorithm. The triangulation takes on the general functional form shown in Equation 4.35.

$$\mathbf{X} = \tau(\mathbf{x}, \mathbf{x}', \mathbf{P}, \mathbf{P}') \tag{4.35}$$

A simple linear triangulation method employs the DLT, as used to compute the fundamental matrix in Section 4.7.5. The projection matrices map 3D points onto 2D points of the focal planes of the sensors: $\mathbf{x} = \mathbf{P}\mathbf{X}$ and $\mathbf{x}' = \mathbf{P}'\mathbf{X}$. The equations can be combined into a linear form by recalling the fundamental matrix given in Equation 4.26, which states that the cross-product of the two image points shall equal zero. In practice this will never be the case, as illustrated in Figure 4.14, as there will likely be triangulation error (d and d') due to errors in the correspondences and/or camera models. Therefore, the DLT solves for the linear least-squares solution for the 3D point.

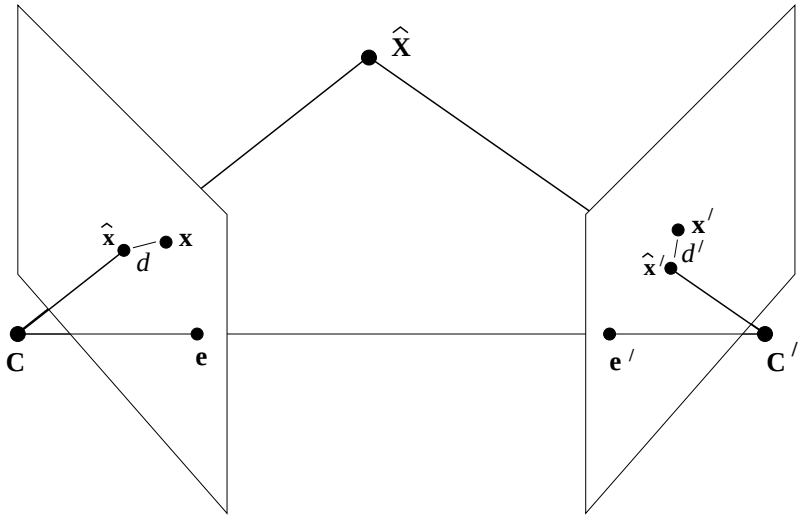


Figure 4.14: Triangulation methods are used to solve for the 3D point ($\hat{\mathbf{X}}$) given the 2D image correspondence coordinates ($\mathbf{x}$ and $\mathbf{x}'$). The triangulation error is highlighted in the figure as d and d', which is the difference in the correspondence points from the projection of $\hat{\mathbf{X}}$ ($\hat{\mathbf{x}}$ and $\hat{\mathbf{x}}'$). Reproduced from Hartley and Zisserman [2].

The cross product yields a set of linear equations as a function of 2D points, as shown in Equations 4.36, 4.37, 4.38.

$$x(\mathbf{p}^{3\mathrm{T}}\mathbf{X}) - (\mathbf{p}^{1\mathrm{T}}\mathbf{X}) = 0 \tag{4.36}$$

$$y(\mathbf{p}^{3\mathrm{T}}\mathbf{X}) - (\mathbf{p}^{2\mathrm{T}}\mathbf{X}) = 0 \tag{4.37}$$

$$x(\mathbf{p}^{2\mathrm{T}}\mathbf{X}) - y(\mathbf{p}^{1\mathrm{T}}\mathbf{X}) = 0 \tag{4.38}$$

These expressions can be placed into an $\mathbf{AX} = \mathbf{0}$ form (see Equation 4.39).

$$\mathbf{A} = \begin{bmatrix} x\mathbf{p}^{3\mathrm{T}} - \mathbf{p}^{1\mathrm{T}} \\ y\mathbf{p}^{3\mathrm{T}} - \mathbf{p}^{2\mathrm{T}} \\ x\mathbf{p}'^{3\mathrm{T}} - \mathbf{p}'^{1\mathrm{T}} \\ y\mathbf{p}'^{3\mathrm{T}} - \mathbf{p}'^{2\mathrm{T}} \end{bmatrix} \tag{4.39}$$

The DLT method will yield the 3D point that represents the linear least-squares solution to the equation. Applying this method for all the correspondences across views allows for the initial 3D point cloud to be generated.

4.8 Radial Distortion Compensation

As stated in Section 4.7.1, the original input images likely exhibit some amount of radial distortion that should be removed before proceeding through the workflow. The radial distortion coefficients from Equation 4.16, solved via the bundle adjustment method in Section 4.7.6, can be used to warp the images. This results in spatially rectified images. The Radial Undistort stage provides this function in the Catena framework (see Figure 4.15).

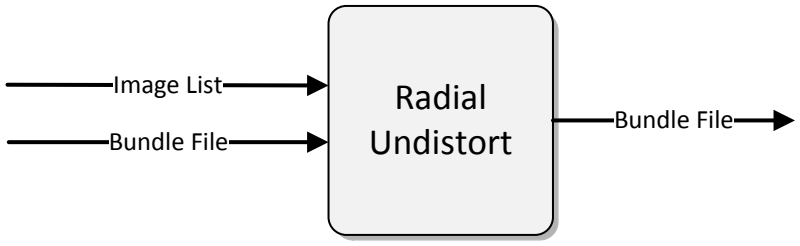


Figure 4.15: The Radial Undistort stage compensates for radial distortion detected in the images to improve down-stream processes.

4.9 Output Conversion

The outputs from the bundle adjustment stage need to be modified in order to be compatible with the dense point cloud generation steps. This is purely a "book-keeping" step to convert Bundler output to a form that is required by CMVS / PMVS (see Figure 4.16).

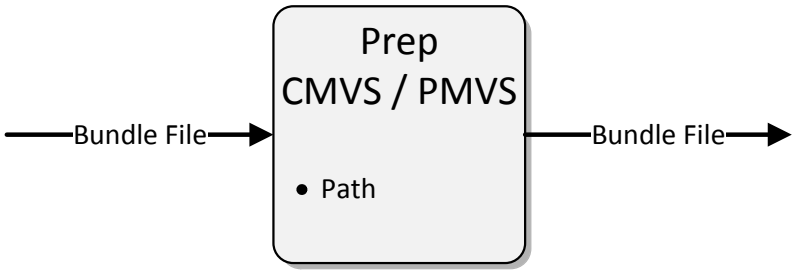


Figure 4.16: The Prep CMVS / PMVS stage converts the Bundler output to an acceptable form for CMVS and PMVS.

4.10 View Clustering / Reduction

Given a set of camera matrices and sparse 3D point cloud, it is desirable to cluster the views into sub-groups to minimize the size of image sets for a dense reconstruction in the following multi-view stereo (MVS) step. It is advantageous for the dense reconstructor to operate on a minimal set of images, as the required memory and computation time increases exponentially with the number of images, making some sets intractable to process. Likewise, if the images are broken into independent clusters, they can be processed in parallel. The algorithm implemented in the CMVS [8] has three main goals:

1. **Compactness**: remove redundant images from the collection
2. **Size**: minimize the size of the image clusters
3. **Coverage**: reconstruct the clusters to result in an identical point cloud as if it was computed from the original / complete set

CMVS accomplishes these goals through the constraint equations with parameters outlined in Table 4.1. There are default values that the user is able to tune for optimal performance with their dataset. CMVS strives to establish a minimum set of clusters,

Table 4.1: CMVS algorithm parameters.

Parameter	Default	Supporting Goal	Description
α	150	Size / Compactness	maximum cluster size, i.e., number of images per cluster
λ	0.7	Coverage	view coverage parameter as a function of 3D points and clusters (explained below)
δ	0.7	Coverage	ratio of covered points in a single view

where the individual cluster sizes are less than α. The coverage must meet the criteria, λ and δ, which are parameters of constraint functions.

CMVS expresses *reconstruction accuracy* as a function of 3D points ($\mathbf{X}$) and the cluster ($\mathbf{K}$), given as $f(\mathbf{X}, \mathbf{K})$. A detailed derivation of the CMVS function can be found in [8]. A 3D point $\mathbf{X}_j$ is considered "covered" if its reconstruction accuracy in the cluster, $\mathbf{K}_k$, is at least λ times the theoretical reconstruction accuracy. Individual coverage is enforced by the ratio parameter δ, whereby the number of points covered by the view must be at least δ times the cluster coverage. This minimization process removes redundant views, as the constraint criteria inherently discover views that meet constraints with less cost. It subsequently excludes poor-quality images, as they contain less reconstructed points, which makes them more costly to include in a cluster, especially when another high-quality view is available that contains the same 3D points.

The algorithm steps described below are carried out as part of CMVS, which is exposed in the Catena framework as a stage, and its interface is illustrated in Figure 4.17.

1. **3D Point Filter**: The neighborhood of every point in the input sparse 3D point cloud is analyzed. If there is more than one point contained in the neighborhood, it is consolidated, and the output 3D point is computed as the average of all the neighbors. This process yields a sparser 3D point cloud with coverage information that is required in the remaining steps of the algorithm.

2. **Redundancy Removal**: Using the constraint functions developed previously, each image is removed from the respective cluster. If the constraints are still satisfied, it is removed entirely from the image set. The algorithm processes images in increasing order of resolution such that low resolution images are discarded first.

3. **Cluster Establishment**: Clusters are divided until the size constraint is met. CMVS uses a *Normalized Cuts* algorithm [34] to facilitate the process, whereby the

reconstruction accuracy function is used to establish contribution of an image to the final reconstruction.

4. **Image Addition**: The previous step enforced the size constraint without consideration of coverage. This process adds images to clusters to maximize the coverage metric, while honoring the size constraint.

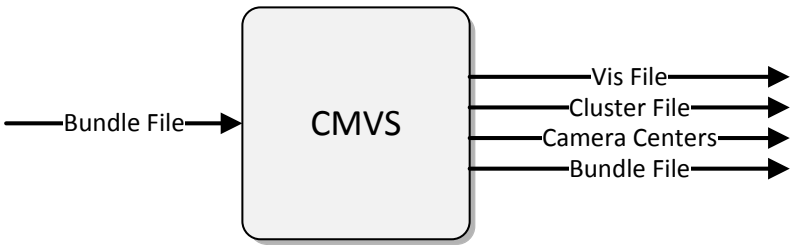


Figure 4.17: The CMVS stage detects and filters redundant views in preparation for PMVS.

4.11 Dense Point Cloud Generation

The camera matrices and sparse point cloud computed from the image-based geometry estimation process in Section 4.7 can be used to generate a denser point cloud by employing a multi-view stereo (MVS) technique. The PMVS [8] algorithm is a commonly used implementation for the SfM workflow. Note, this step can operate on either the output from Bundler or CMVS. By preparing the data with the CMVS implementation, the MVS algorithm can be run on the individual clusters. The sectioning is advantageous since the clusters are independent and can be run in parallel. Additionally, the smaller subset of clusters can be executed faster and requires less memory.

There are three basic high-level steps in the PMVS algorithm:

1. **Match**: The Harris corner detection algorithm [35] is run and the DoG images are computed. The image is sub-divided into 32x32 pixel regions (patches). Using the four local maxima within the patch, the strongest "Harris corner" and "DoG" responses identify correspondences between image pairs. The respective camera models are leveraged and epipolar geometry is utilized to verify consistent locations in the image. The corresponding 3D point is triangulated and photometric consistency is verified by calculating the normalized cross correlation (NCC) value of the patches.

2. **Expand**: Patch neighbors are identified. The patch definition is expanded to include nearby pixels, yielding dense patches.

3. **Filter**: The patches that have been expanded incorrectly due to occlusion (i.e., one patch lies in front of another in 3D space), are broken up. The number of patches adjacent to a given patch, relative to all views, is compared against a ratio that is initialized to 0.7 and decreased by 0.2 after each "expand / filter" iteration.

The *expand* and *filter* steps are run three times after *matching* to consolidate clusters. PMVS is wrapped in a Catena stage, as illustrated in Figure 4.18, and is available for use in workflow chains.

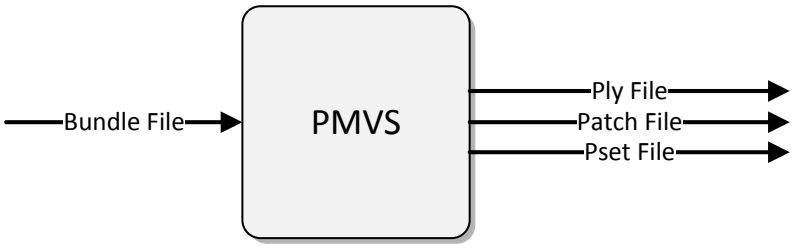


Figure 4.18: The PMVS stage uses the consistent camera matrices found by Bundler to generate a dense point cloud using a patch-based technique.

4.12 Geographic Considerations

The point cloud generated from Bundler and PMVS are in an arbitrary coordinate system. It is often desirable in geographic information systems (GIS) applications to transform the point cloud into a geographic coordinate system where the 3D points take on a physical meaning of latitude, longitude, and elevation. The method developed by Walvoord, et al. [36] [37] computes a similarity transform using the full physical sensor model of the collection system and the inertial navigation system (INS) and global positioning system (GPS) metadata. The transform is applied to the entire point cloud to map it into a geographic coordinate system for intuitive exploitation and display.

4.13 Surface Reconstruction

While a dense (or even sparse) point cloud is sufficient for automated exploitation methods, it is often desirable to generate a visually pleasing 3D model of the point cloud. Poisson

surface reconstruction [38] is a commonly used technique for interpolating point clouds and generating a mesh. By performing surface reconstruction, facets can be generated from the point cloud and the colors interpolated to produce a more visually pleasing result. Depending on the density of the point cloud, this technique can generate a sufficient result for human visualization. The Poisson surface reconstruction algorithm is exposed as a Catena stage, as illustrated in Figure 4.19. Alternative surface reconstruction methods include a denser point cloud generation, such as the Semi-Global Matching (SGM) [39] algorithm, which provides a denser input for the surface reconstruction, or a texture mapping technique that utilizes the original imagery in order to "drape" over the point cloud.

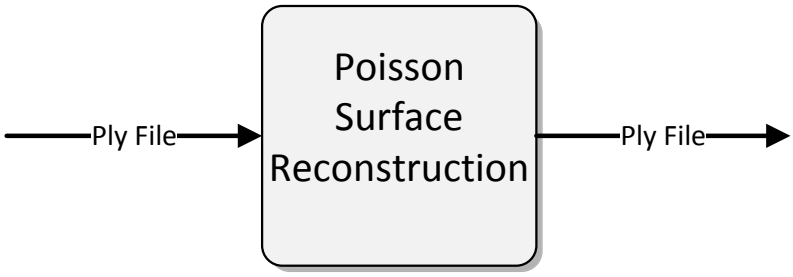


Figure 4.19: The Poisson Surface Reconstruction stage performs interpolation of vertices to generate faces between 3D points, yielding a meshed 3D model.

4.14 Visualization / Exploitation

There are a variety of tools that provide visualization capability, including Meshlab [40], Blender [41], and CloudCompare [42]. The point cloud can also be exploited in an automated fashion for mensuration purposes (e.g., using tools such as point cloud library (PCL) [43]). Figure 4.20 illustrates the typical termination of the SfM chain, where the point cloud or surface reconstructed result is visualized.

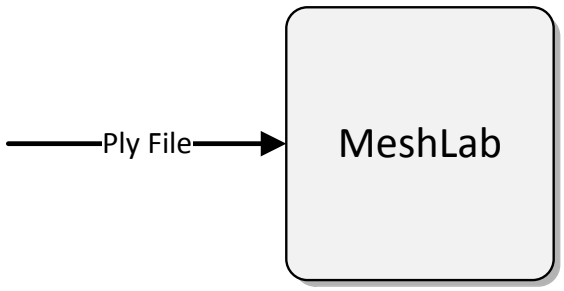


Figure 4.20: The MeshLab stage provides visualization of the 3D point cloud and model.

4.15 SfM Chain

Using the componentized stages described in this chapter, the SfM workflow can be constructed using Catena (see Figure 4.21). With this, multi-view imagery is processed to produce a 3D model of a scene, which can be mapped back to physical geographic coordinates.

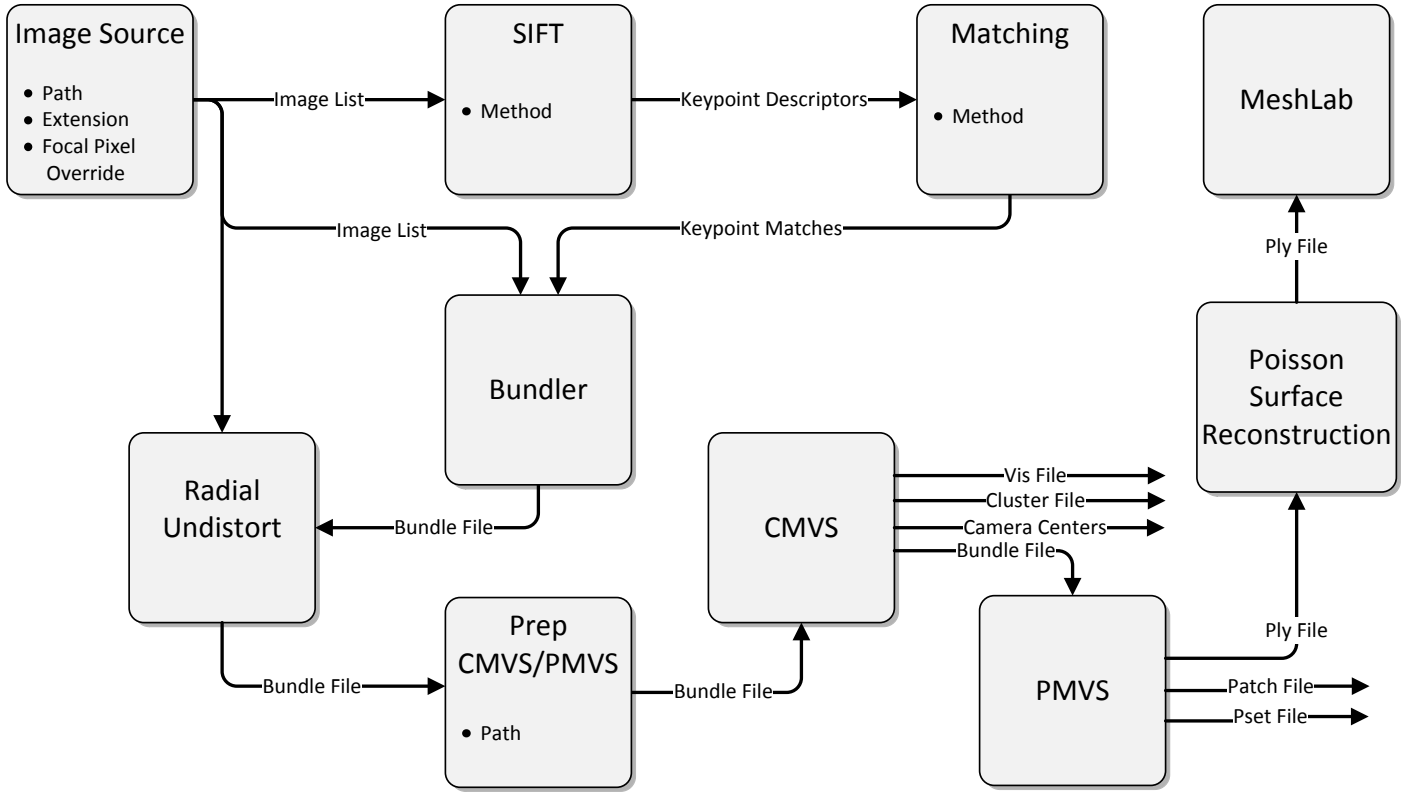


Figure 4.21: The complete SfM chain used to generate 3D models from multi-view 2D imagery.

Chapter 5

Results

5.1 SfM Application

The SfM chain was used on a variety of datasets to generate point clouds. Many multi-view image sets were used to compute a 3D model of the given scene. This section presents some of the results.

5.1.1 ET

The "ET" dataset provided with the CMVS / PMVS software consists of nine views that have a resolution of 640x480x3. This dataset is commonly used to demonstrate semi-automated SfM software and point cloud generation. The input images are given in Figure 5.1. Using the SfM chain described in Chapter 4, the point cloud in Figure 5.2 was generated. At the magnification shown in Figure 5.2, the point cloud is relatively sparse and gaps in the model can be observed. This is due to occlusion and a limited number of perspectives of the scene in the original input imagery. However, the point cloud is still easily recognizable as ET.

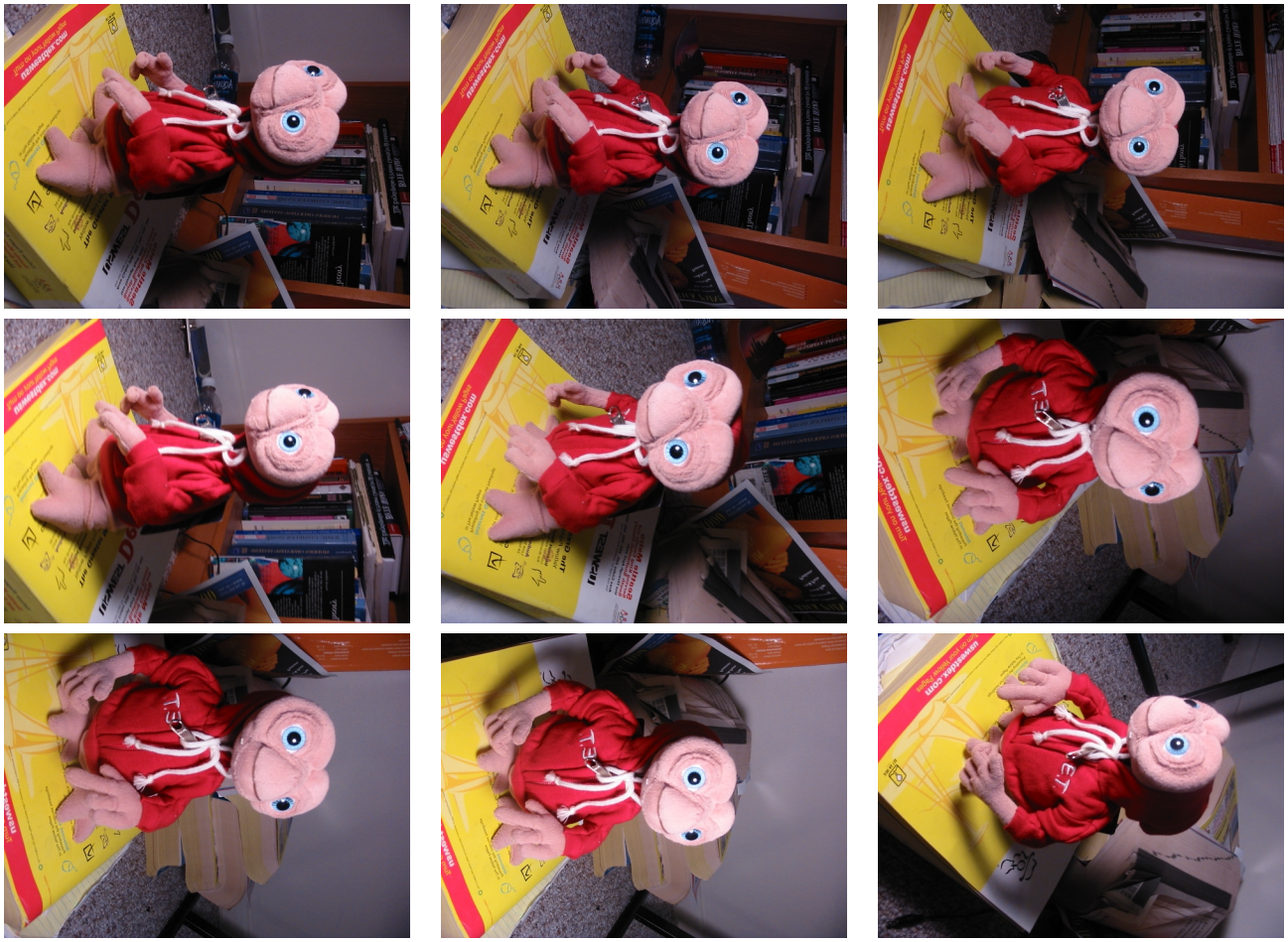


Figure 5.1: ET images provided by the CMVS / PMVS software distribution.

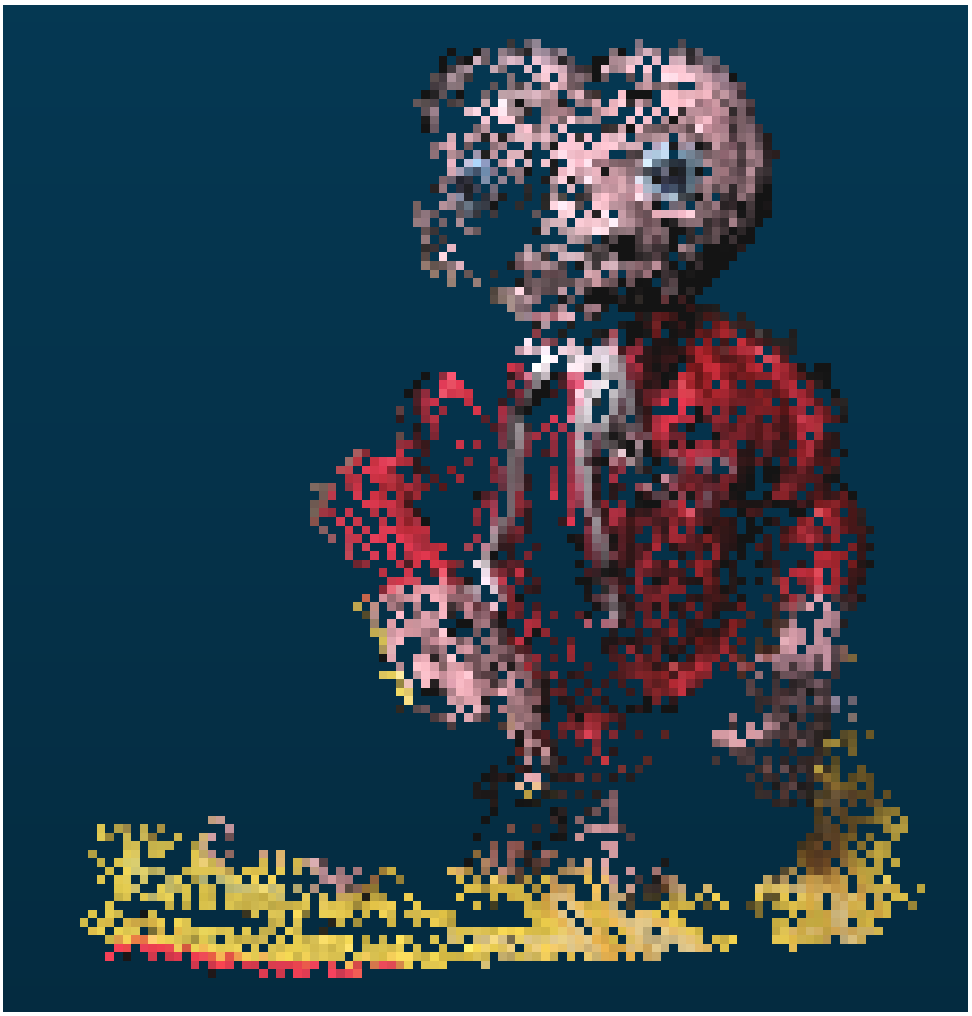


Figure 5.2: ET point cloud generated from nine multi-view images of the scene.

5.1.2 Hall

The "Hall" dataset provided with the CMVS / PMVS software is often used as an example of a building reconstruction. The entire dataset consists of 61 views (3008x2000x3), for which a subset is included in Figure 5.3. The Catena SfM workflow was utilized to generate the point clouds, which represents a 3D model of the scene (see Figure 5.4). Although Figure 5.4 shows a 2D representation of a 3D scene, this can be visualized with a 3D viewer and manipulated to observe different perspectives. If a particular dimension of a feature in the scene is known, this information can be used to transform the point cloud into a metric coordinate system. Therefore, the points have physical dimensionality. This offers the viewer the ability to perform mensuration exploitation of the scene (i.e., measure the size of a feature, distance between two features, etc.).

Figure 5.3: A subset of "Hall" images provided by the CMVS / PMVS software distribution.

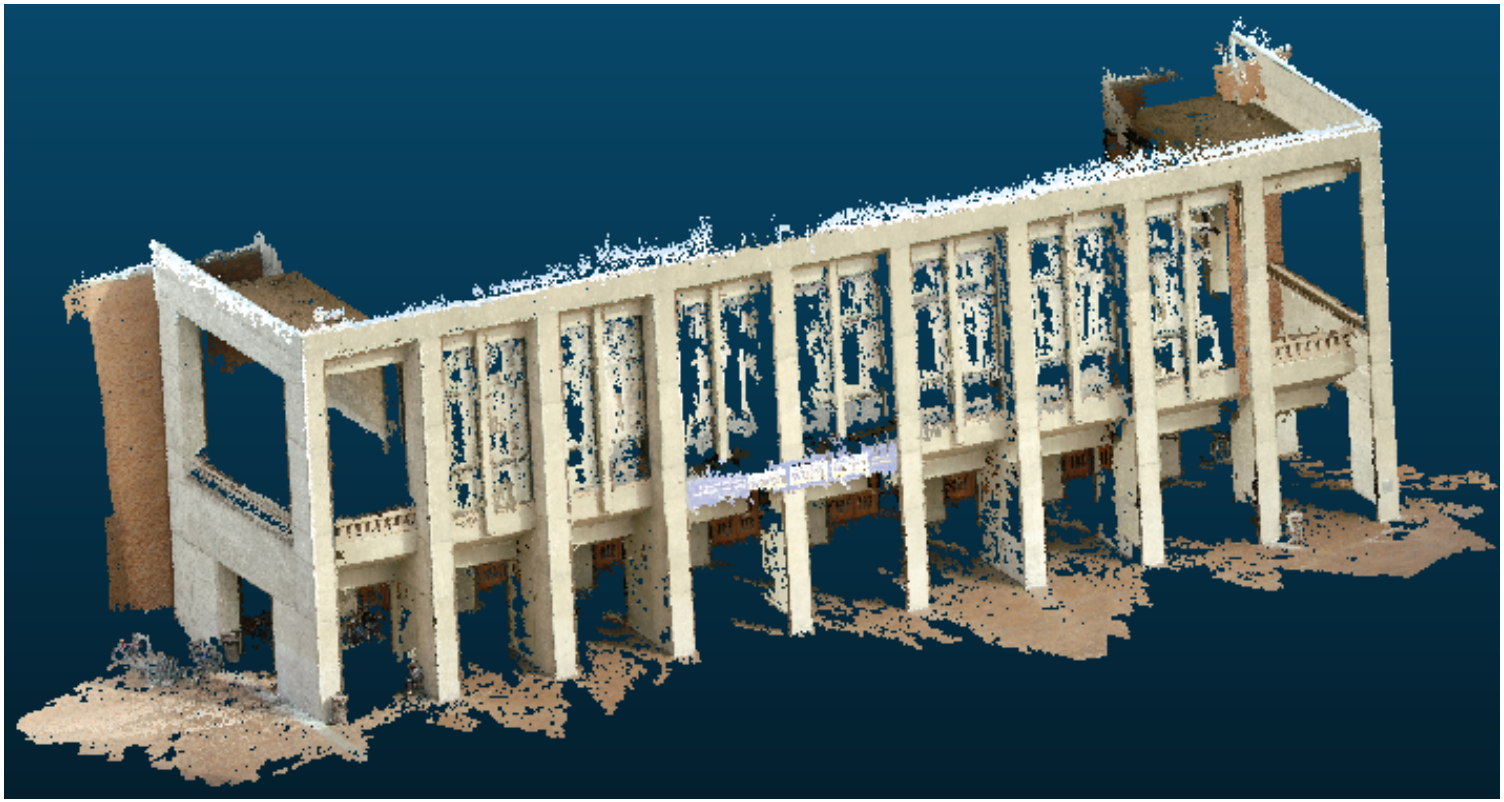


Figure 5.4: Point cloud of the “Hall” scene generated using the Catena SfM workflow and 61 multi-view input images.

5.1.3 RIT WASP

The Rochester Institute of Technology (RIT) Wildfire Airborne Sensor Program (WASP) sensor was originally developed for remote sensing of wildfires. However, this technology was used to obtain airborne imagery of downtown Rochester, NY at a resolution of 4000x2672x3. The entire dataset consists of 247 nadir views, and a subset is included in Figure 5.5. The point cloud generated in Figure 5.6 exhibits some gaps on the sides of buildings due to the nadir perspective of the collection geometry. There are also gaps in the water regions, as it is very difficult to perform feature extraction / matching in these areas. The content is inconsistent between views (moving water) and has no texture for the algorithm to detect.

Figure 5.5: A subset of images taken of Rochester, NY were obtained from the RIT WASP sensor.

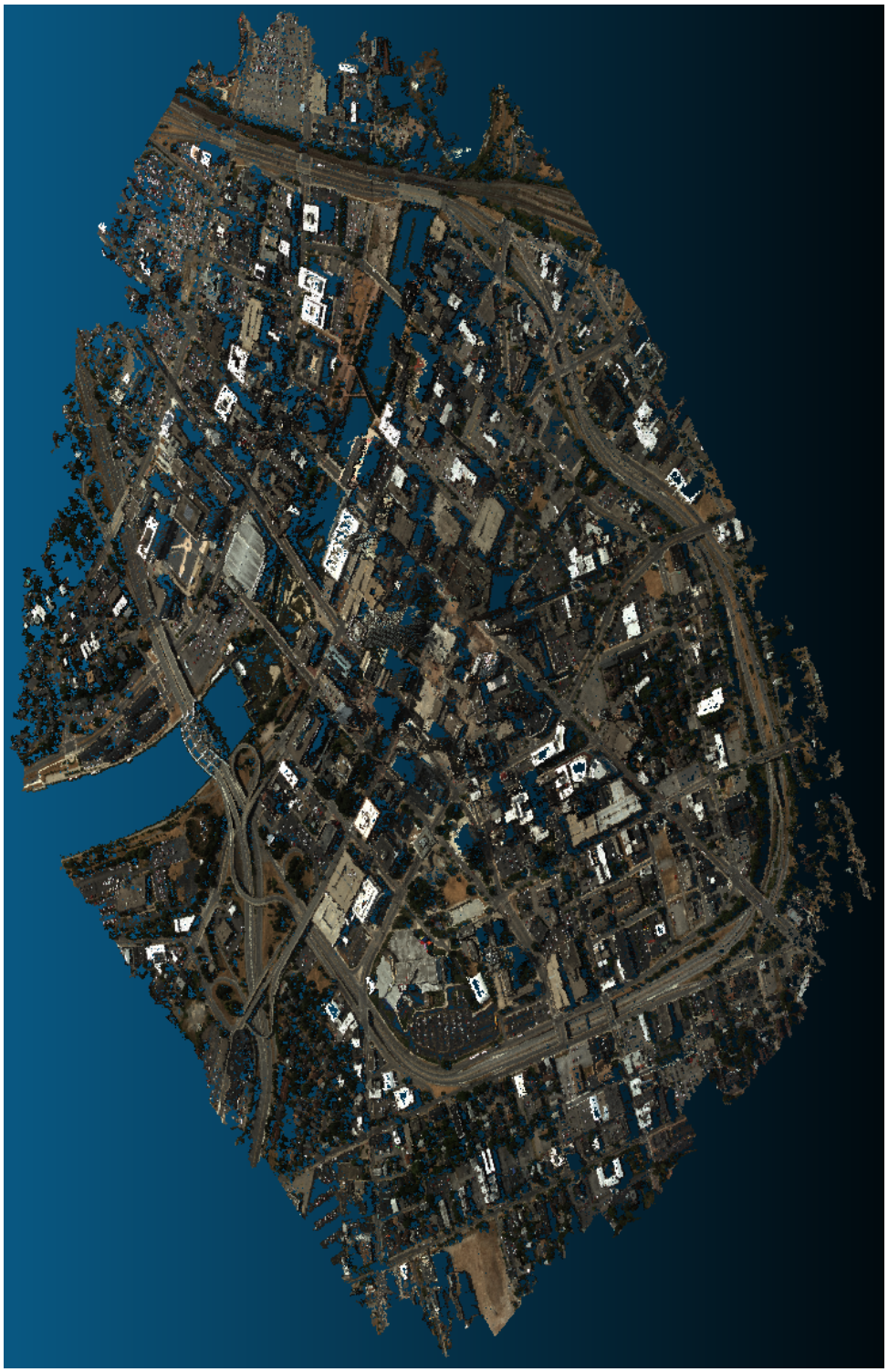


Figure 5.6: A point cloud was generated using the RIT WASP images presented in Figure 5.5.

5.1.4 ITT Exelis WAMI

The ITT Exelis Wide Area Motion Imagery (WAMI) system is a commercial sensor used to provide remotely sensed (nadir or oblique) persistent surveillance imagery at a resolution of 4872x3248x1. This system was used to capture 96 views of the city of Rochester, NY, and a subset of these images are provided in Figure 5.7. Using the Catena SfM workflow, a point cloud was generated (see Figure 5.8). The gaps previously present RIT WASP model are filled with content, due to the oblique collection geometry. There are also differences in resolution between the systems, which is a function of many variables, including sensor / vehicle altitude, optics, and sensor resolution, among others. In remote sensing, it is useful to introduce the concept of ground sample distance (GSD). GSD can be used to compare the resolutions of systems in terms of physical units (i.e., meters per pixel). It combines many of the factors previously listed that contribute to the effective resolution.

In this particular example, the point cloud was transformed into a geographic coordinate system using the method described in Section 4.12. As such, specific points in the model represent known latitude / longitude / elevation positions in the geographic coordinate system. This model was used to generate 3D fly-through animations.

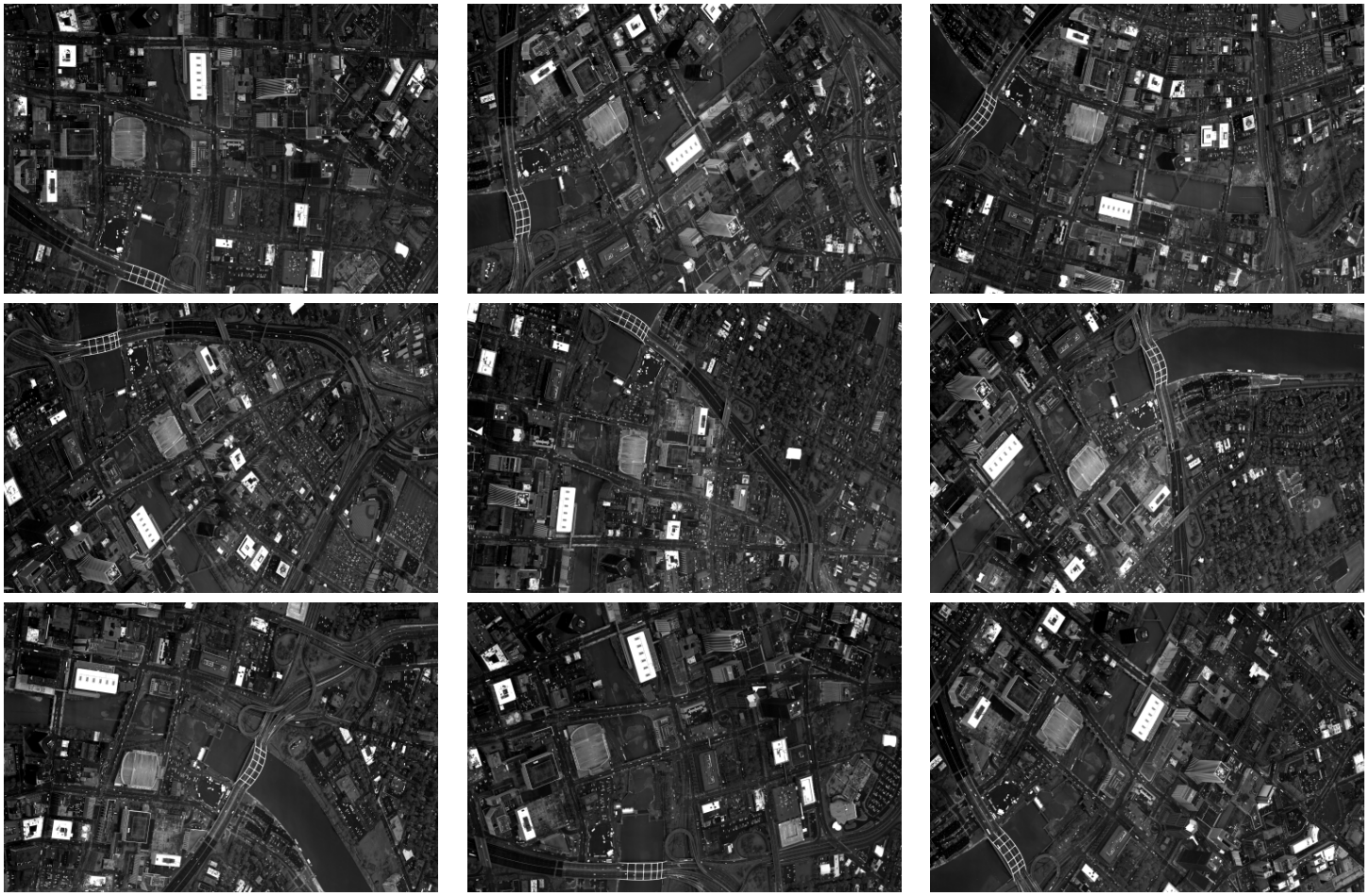


Figure 5.7: A subset of images over downtown Rochester, NY were obtained from the ITT Exelis WAMI sensor.

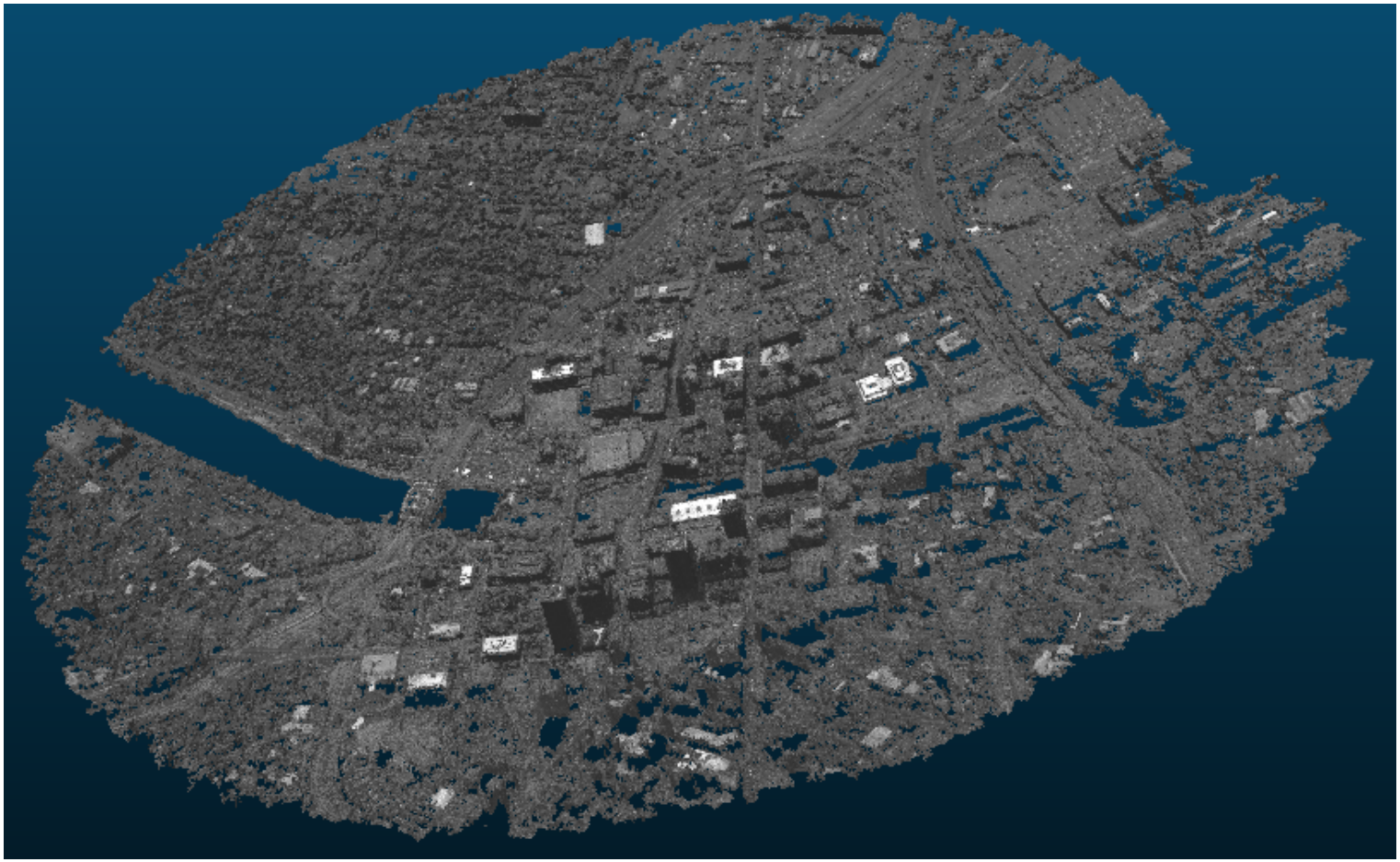


Figure 5.8: 3D model of downtown Rochester, NY generated from multi-view imagery obtained from the ITT Exelis WAMI sensor.

5.2 Extensibility

A strong feature of the Catena abstracted workflow framework is the aspect of extensibility. The stages developed for the SfM application were leveraged to carry out registration, mosaicing, and many other tasks with minimal new implementation of stages / components. Given this success, it becomes easy to accurately hypothesize how Catena could be used in new applications with additional stage development.

5.2.1 Registration

Many registration chains have been built in industry using pre-existing Catena stages. Figure 5.9 illustrates an example chain with new stages (denoted by a cross-hatch pattern) to facilitate image registration using image features via the SIFT algorithm. Two images sources are used to input "reference" and "test" images. Every combination of reference and test image is formed into an "image pair." Feature extraction is performed on the images and keypoint matches are established in the matching stage. The RANSAC algorithm is used to filter correspondences that are inconsistent with the selected transformation (e.g., homography). The correspondences are used to warp the test image to the reference image's coordinate space.

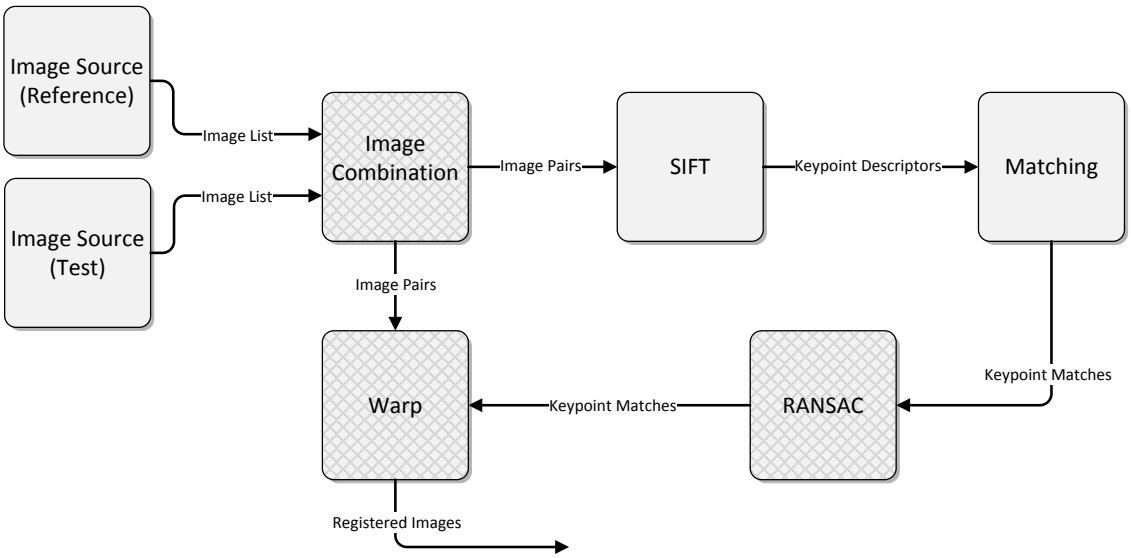


Figure 5.9: An example registration chain built in industry, leveraging stages from the SfM application.

5.2.2 Health Imaging

A case in the health imaging domain is presented to illustrate the flexibility and applicability of stages developed for the SfM application. A pair of "dual-energy" X-ray images are acquired by exposing a patient to a low and high dose of radiation. Since images are not acquired simultaneously, this results in misregistration. However, in order to exploit the imagery from this modality, the images must be accurately registered in order to compare their relative intensities. Stages from the SfM were leveraged, requiring only two new stages to be implemented. The image chain is shown in Figure 5.10, and the new stages are denoted with a cross-hatch pattern.

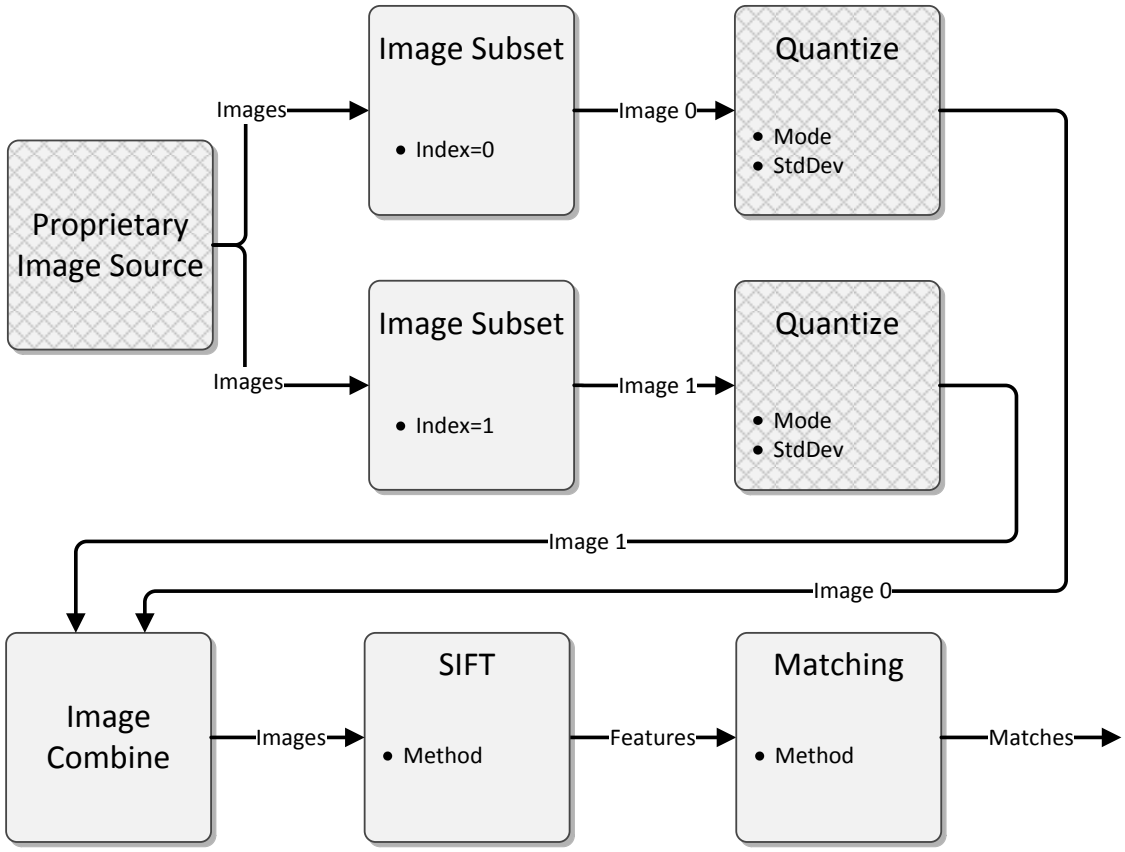


Figure 5.10: An example chain built in industry to facilitate dual-energy image registration, leveraging stages from the SfM application.

A propriety image source is required to input the X-ray images. The existing image subset stage is used to select each image from the pair in order to appropriately quantize the image, maximizing the dynamic range of the imagery. The images are recombined and the SIFT feature extraction algorithm is executed. Correspondences are established between the pair of images (shown in Figure 5.11), which illustrates the effectiveness of this algorithm. At this point, the correspondences can be input into a registration stage to warp the images.

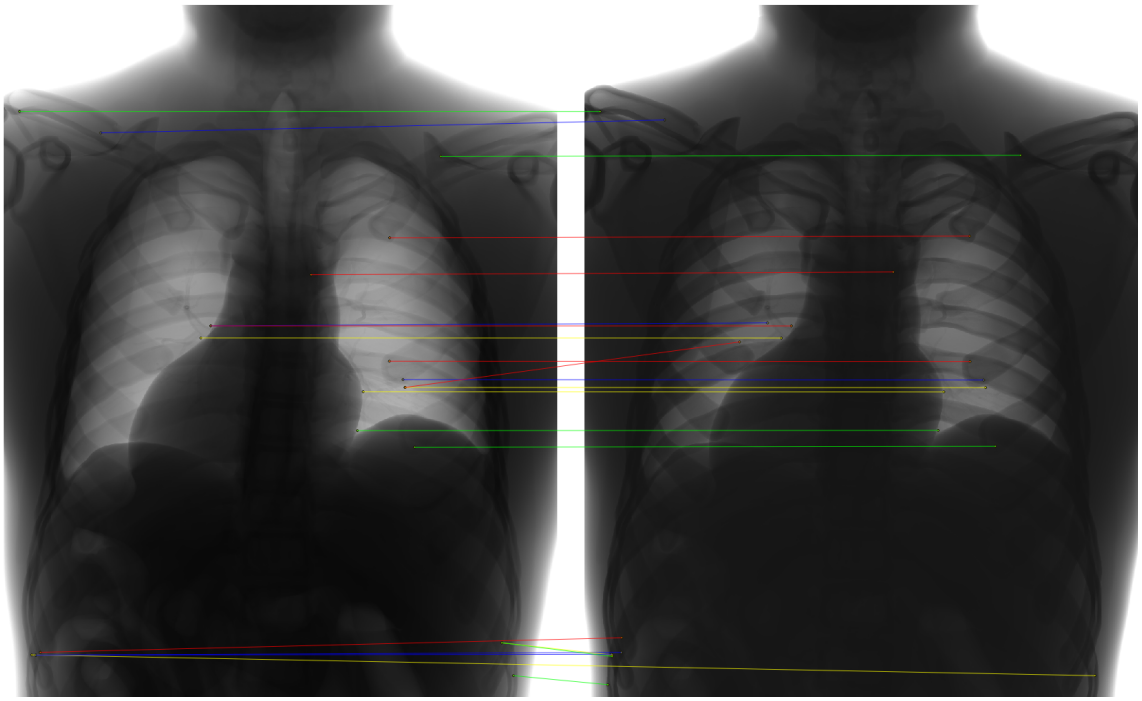


Figure 5.11: Correspondences between a pair of dual-energy X-ray images.

5.2.3 Hypothetical

In this hypothetical scenario, images could be sourced from Google using a search string, geolocation information, and a minimum image resolution criterion. It is very straightforward to contemplate the implementation of such a stage, enabling processing of existing imagery. In addition, a different feature extraction algorithm could be implemented (e.g., SURF [18]). This feature extractor is already provided via the generic FeatureDetector stage in the OpenCV package. The chain in Figure 5.12 illustrates the modification to

the basic SfM chain with only two new stages (cross-hatch pattern) and utilizing existing components.

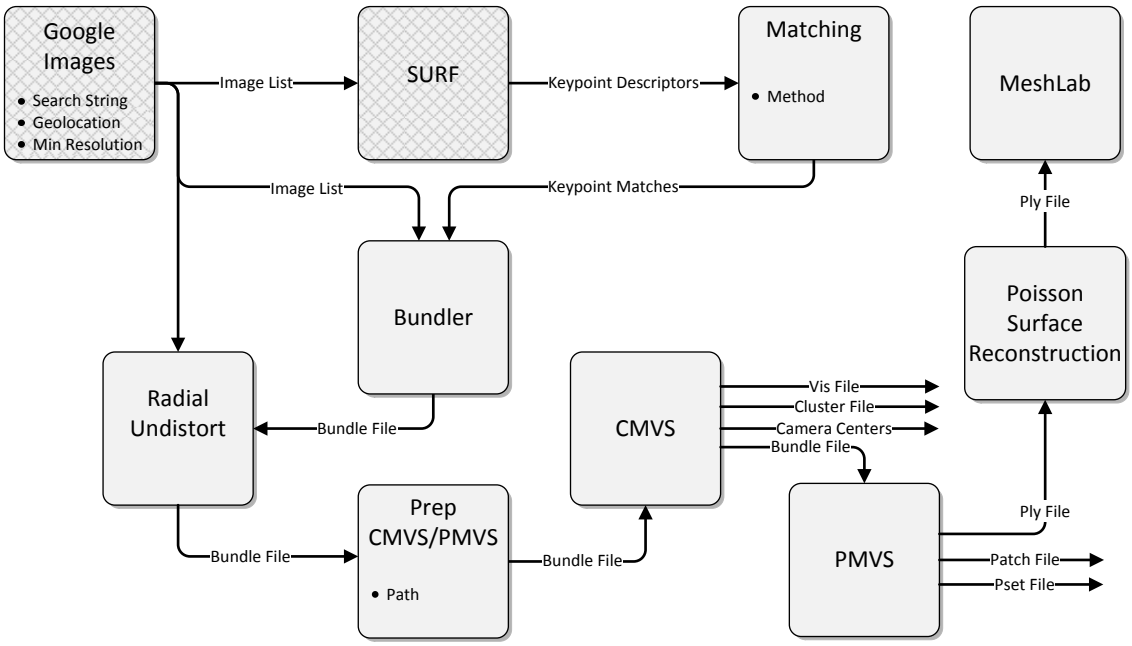


Figure 5.12: A hypothetical chain could be constructed using new "Google Images" and "SURF" stages.

Chapter 6

Conclusions

6.1 Overview

Throughout academia and industry, a common pattern of monolithic software development with no consideration for reuse has been observed. Generally there was no consideration given to architecture, resulting in many inflexible application-specific solutions. This resulted in software that had a very short lifespan after scientific development, studies, and analysis.

This motivated the development of a flexible workflow framework with specific design and architectural choices to mitigate problems in the areas of componentization, reuse, documentation, and maintainability. A software implementation named Catena [44] was created in the Python programming language and open sourced. Catena is currently used in many environments, including: Center for Imaging Science (CIS)/RIT, ITT Exelis, Carestream Health, and MIT Computer Science and Artificial Intelligence Laboratory, among others.

The components that make up the SfM chain, and the underlying data structures, were developed, implemented, and included in the base Catena distribution. Many alternative implementations of algorithms were tested by utilizing the interchangability feature of stages given an equal interface. Likewise, collections of stages were replaced with a single stage, showing that an entire process can be represented by multiple stages, implemented in a "super-stage," and easily replaced in the chain. The base packages were extended in many environments to facilitate new applications, including mosaicing, registration, general image processing, and other computer vision tasks.

Catena is an open-source project hosted on Google Code (`http://catena.googlecode.com`). It will be developed and maintained with the goal that it will continue to be used in many environments. An architectural foundation with

packages of stages related to computer vision, image processing, and SfM provide a basis for new work in different domains.

6.2 Abstraction Benefits

The benefits of abstraction have become obvious throughout practical usage of the Catena implementation. First, the ability to construct chains from packages of stages, either programmatically or graphically, is very powerful. This provides a high-level view of the overall workflow that is normally obscured in other environments, offering access to users at all levels.

The definition of stage interfaces and the ability to swap stages with equivalent interfaces is extremely powerful. There have been many instances when stages have been evaluated for performance (e.g., feature extractor) given an input dataset. The ability to seamlessly and effortlessly swap stages becomes invaluable.

The visualization tools provide a generic utility to build chains, debug stage outputs, and optimize parameters. This is normally a time-consuming process that is often overlooked during algorithm development and implementation.

Lastly, the ability to develop packages of stages that represent baseline functionality, with documentation of the interface and properties of the algorithm, is invaluable in enabling growth of the field. Without a framework that enforces a certain level of rigor and a communication mechanism to the next scientist / engineer, algorithms and implementations are lost. This results in duplication of effort and slow research and development.

6.3 Future Development

A host of enhancements and features became apparent throughout the development and usage of Catena, which are outlined and explained below.

6.3.1 Binary Overlays

The choice was made to include platform specific binaries with the Catena distribution. The directory structure of the binaries was changed many times before settling on the current structure where the executable resides underneath the package, in a directory named according to the platform (e.g., Linux64bit). Helper methods were implemented to make it easy for the developer to invoke an executable, in a platform-agnostic fashion, within a stage. Over time, it was discovered that there are subtle differences, even within the current breakdown of platforms, which creates the need for very specific platform binaries. Therefore, the binaries should be removed from the package directories and

placed in a separate directory, mirroring the package directory, but containing builds of software that are very specific to a platform (e.g., openSUSE v11.4). The Subversion (SVN) checkout / export process would involve checking out the main Catena software and then choosing the platform(s) of interest. The platform directory would be placed over top of the Catena software, effectively resulting in the current structure, but with binaries built for the specific desired platform(s).

6.3.2 Composite Stages

In some cases, it would be advantageous to represent a sequence of stages as a single, higher-level stage. The concept of a composite stage should be implemented, which abstracts multiple stages while exposing all the properties of the underlying stages. This will make it possible to abstract an entire chain, where the inputs are images and the output is a point cloud or 3D model. Another example is wrapping the feature extraction and matching stages into a single stage. There would be utility in this abstraction where a composition of stages needs to be provided to a high-level user (e.g., via the Chain Builder GUI).

6.3.3 Generalized Property Optimization

In many cases a chain is constructed from a collection of stages and a manual process is required to determine an optimal, or sometimes satisfactory, set of property values for stages given a dataset or input. The general problem of property optimization became apparent after creating chains that require feature extraction and matching stages. If a metric can be implemented as a stage, which provides an assessment for the amount of error given some property vector, an automated method could utilize the cost function and non-linear estimation technique (such as Genetic Algorithms, Simulated Annealing, or L-M) and implemented in a generic fashion to automatically "tune" the properties of the stages within a chain.

6.3.4 Distributed / Multi-threaded Execution

The abstracted workflow framework was implemented with distributed / multi-threaded execution capabilities in mind. Given a chain, it is very straightforward to analyze the dependencies to determine which can be executed in parallel. It is also possible with a slight modification to the base stage class to indicate independent looping that could be unrolled and executed across many threads and/or machines. A distributed computing facility could be leveraged to carry out concurrent execution of stages or work within a stage to drastically speed up rendering.

Appendix A

Supporting Information

A.1 Cross-product Notation

A cross product can be implemented as a matrix multiplication for 3-dimensional vectors by creating a skew-symmetric matrix, e.g., given $\mathbf{a} = [a_1, a_2, a_3]$, the skew-symmetric matrix is constructed as:

$$[\mathbf{a}]_\times = \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix} \tag{A.1}$$

The cross product of $\mathbf{a}$ with $\mathbf{b}$ can be expressed as:

$$\mathbf{a} \times \mathbf{b} = [\mathbf{a}]_\times \mathbf{b} \tag{A.2}$$

A.2 Auto-generated Stage Documentation

This section contains dynamically generated documentation of the stages found in the current set of packages. The self-documentation contract established by Catena is utilized to facilitate this feature. The tool used to generate the LaTeXdocumentation is included in the source code. This allows users to automatically generate documentation for their stages.

Table A.1: Auto-documentation for **BundleAdjustment** package

BundleAdjustment

Bundler

Bundler takes a set of images, image features, and image matches as input, and produces a 3D reconstruction of camera and (sparse) scene geometry as output. The system reconstructs the scene incrementally, a few images at a time, using a modified version of the Sparse Bundle Adjustment package of Lourakis and Argyros as the underlying optimization engine. Bundler has been successfully run on many Internet photo collections, as well as more structured collections. (https://www.cs.cornell.edu/ snavely/bundler/)

Property Name	**Data Type**	**Default Value**	**Description**
Constrain Focal	bool	True	Add a soft constraint on focal lengths to stay near their estimated values
Constrain Focal Weight	float	0.0001	Strength of the focal length constraints
Estimate Radial Distortion	bool	True	Whether to estimate radial distortion parameters
Fisheye Parameter File	str	[empty]	Fisheye parameter file path
Fixed Focal Length	bool	False	Use fixed focal length (Initial Focal Length)
Force Run	bool	False	Force run if outputs already exist
Initial Focal Length	float	-2147483647.0	Initial focal length
Maximum Projection Error Threshold	int	16	The maximum value of the adaptive outlier threshold
Minimum Projection Error Threshold	int	8	The minimum value of the adaptive outlier threshold
Previous Bundler Results File	str	[empty]	Previous bundle adjustment results file path

Projection Estimation Threshold	int	4	RANSAC threshold when performing pose estimation to add in a new image
Ray Angle Threshold	int	2	Triangulation ray angle threshold
Run Slow Bundler	bool	False	Run slow bundle adjustment (adds one image at a time)
Seed Image Index 1	int	-2147483647	First image index to seed bundle adjustment
Seed Image Index 2	int	-2147483647	Second image index to seed bundle adjustment
Trust Focal Estimate	bool	False	Trust the provided focal length estimates (i.e., don't attempt to cross-check with self-calibration)
Use Focal Length Estimate	bool	True	Initialize using focal length estimates specified in the list file
Variable Focal Length	bool	True	Use variable focal length

Table A.2: Auto-documentation for **Cluster** package

Cluster

CMVS

CMVS takes the output of a structure-from-motion (SfM) software as input, then decomposes the input images into a set of image clusters of managable size. An MVS software can be used to process each cluster independently and in parallel, where the union of reconstructions from all the clusters should not miss any details that can be otherwise obtained from the whole image set. (http://www.di.ens.fr/cmvs/)

Property Name	Data Type	Default Value	Description
CPUs	int	4	Number of CPUs to utilize
Force Run	bool	False	Force run if outputs already exist

PMVS

PMVS is a multi-view stereo software that takes a set of images and camera parameters, then reconstructs 3D structure of an object or a scene visible in the images. Only rigid structure is reconstructed, in other words, the software automatically ignores non-rigid objects such as pedestrians in front of a building. The software outputs a set of oriented points instead of a polygonal (or a mesh) model, where both the 3D coordinate and the surface normal are estimated at each oriented point. (http://www.di.ens.fr/pmvs/)

Property Name	Data Type	Default Value	Description
CPUs	int	4	Number of CPUs to utilize.

Cell Size	int	2	Controls the density of reconstructions. The software tries to reconstruct at least one patch in every csize x csize pixel square region in all the target images specified by timages. Therefore, increasing the value of csize leads to sparser reconstructions. Note that if a segmentation mask is specified for a target image, the software tries to reconstruct only foreground pixels in that image instead of the whole.
Force Run	bool	False	Force run if outputs already exist
Image Pyramid Level	int	1	The software internally builds an image pyramid, and this parameter specifies the level in the image pyramid that is used for the computation. When level is 0, original (full) resolution images are used. When level is 1, images are halved (or 4 times less pixels). When level is 2, images are 4 times smaller (or 16 times less pixels). In general, level = 1 is suggested, because cameras typically do not have r,g,b sensors for each pixel (bayer pattern). Note that increasing the value of level significantly speeds-up the whole computation, while reconstructions become significantly sparse.

Maximum Camera Angle Threshold	int	10	Stereo algorithms require certain amount of baseline for accurate 3D reconstructions. We measure baseline by angles between directions of visible cameras from each 3D point. More concretely, a 3D point is not reconstructed if the maximum angle between directions of 2 visible cameras is below this threshold. The unit is in degrees. Decreasing this threshold allows more reconstructions for scenes far from cameras, but results tend to be pretty noisy at such places.
Maximum Image Sequence	int	-1	Sometimes, images are given in a sequence, in which case, you can enforce the software to use only images with similar indexes to reconstruct a point. sequence gives an upper bound on the difference of images indexes that are used in the reconstruction. More concretely, if sequence=3, image 5 can be used with images 2, 3, 4, 6, 7 and 8 to reconstruct points.
Minimum Image Number	int	3	Each 3D point must be visible in at least minImageNum images for being reconstructed. 3 is suggested in general. The software works fairly well with minImageNum=2, but you may get false 3D points where there are only weak texture information. On the other hand, if your images do not have good textures, you may want to increase this value to 4 or 5.

Other Images	int	-3	Specifies image indexes that are used for reconstruction. However, the difference from timages is that the software keeps reconstructing points until they cover all timages, but not oimages. In other words, oimages are simply used to improve accuracy of reconstructions, but not to check the completeness of reconstructions. There are two ways to specify oimages, which are the same as timages.
Patch Threshold	float	0.7	A patch reconstruction is accepted as a success and kept, if its associated photometric consistency measure is above this threshold. Normalized cross correlation is used as a photometric consistency measure, whose value ranges from -1 (bad) to 1 (good). The software repeats three iterations of the reconstruction pipeline, and this threshold is relaxed (decreased) by 0.05 at the end of each iteration. For example, if you specify threshold=0.7, the values of the threshold are 0.7, 0.65, and 0.6 for the three iterations of the pipeline, respectively.
Sample Window Size	int	7	The software samples wsize x wsize pixel colors from each image to compute photometric consistency score. For example, when wsize=7, 7x7=49 pixel colors are sampled in each image. Increasing the value leads to more stable reconstructions, but the program becomes slower.

Spurious 3D Point Threshold	float	2.5	The software removes spurious 3D points by looking at its spatial consistency. In other words, if 3D oriented points agree with many of its neighboring 3D points, the point is less likely to be filtered out. You can control the threshold for this filtering step with quad. Increasing the threshold is equivalent with loosing the threshold and allows more noisy reconstructions. Typically, there is no need to tune this parameter.
Target Images	str	[empty]	The software tries to reconstruct 3D points until image projections of these points cover all the target images (only foreground pixels if segmentation masks are given) specified in this field (also see an explanation for the parameter csize). There are 2 ways to specify such images. Enumeration: a positive integer representing the number of target images, followed by actual image indexes. Note that an image index starts from 0. For example, '5 1 3 5 7 9' means that there are 5 target images, and their indexes are '1 3 5 7 9'. Range specification: there should be three numbers. The first number must be '-1' to distinguish itself from enumeration, and the remaining 2 numbers (a, b) specify the range of image indexes [a, b). For example, '-1 0 6' means that target images are '0, 1, 2, 3, 4 and 5'. Note that '6' is not included.
Use Visualize Data	int	1	Whether to use the visualize data.

PrepCmvsPmvs

Prepares directories for CMVS/PMVS.

Property Name	Data Type	Default Value	Description

Force Run	bool	False	Force run if outputs already exist
Target Path	str	[empty]	Target path for CMVS/PMVS preparation

RadialUndistort

Uses the radial distortion coefficients from the camera projection matrix solution to remove radial distortion from images.

Property Name	**Data Type**	**Default Value**	**Description**
Force Run	bool	False	Force run if outputs already exist

Table A.3: Auto-documentation for **Common** package

Common

MuxStage

Takes a collection of stages and allows for the selection of one of the given stages to mimic.

TapPoint

Generic tap point stage for inspecting output values of a stage.

Property Name	Data Type	Default Value	Description
Print Functions	dict	[empty]	Dictionary of functions used to print parameters, keyed by type

Table A.4: Auto-documentation for **FeatureExtraction** package

FeatureExtraction

ASIFT

A fully affine invariant image comparison method, Affine-SIFT (ASIFT). While SIFT is fully invariant with respect to only four parameters namely zoom, rotation and translation, the new method treats the two left over parameters : the angles defining the camera axis orientation. Against any prognosis, simulating all views depending on these two parameters is feasible. The method permits to reliably identify features that have undergone very large affine distortions measured by a new parameter, the transition tilt. State-of-the-art methods hardly exceed transition tilts of 2 (SIFT), 2.5 (Harris-Affine and Hessian-Affine) and 10 (MSER). ASIFT can handle transition tilts up 36 and higher. (http://www.cmap.polytechnique.fr/ yu/research/ASIFT)

Property Name	Data Type	Default Value	Description
Downsample	bool	False	Whether to downsample the image before feature extraction
Force Run	bool	False	Force run if outputs already exist
Number of Tilts	int	7	Number of tilts to use in algorithm

Daisy

DAISY is very fast and efficient to compute. It depends on histograms of gradients like SIFT and GLOH but uses a Gaussian weighting and circularly symmetrical kernel. This gives us our speed and efficiency for dense computations. We compute 200-length descriptors for every pixel in an 800x600 image in less than 5 seconds. (http://www.engintola.com/daisy.html)

Property Name	Data Type	Default Value	Description
Disable Interpolation	bool	False	Whether to disable interpolation
Force Run	bool	False	Force run if outputs already exist
Number Random Samples	int	100	If random samples is enabled, the number of samples to take
Orientation Resolution	int	36	If computing rotation invariant features, number of bins to use
Parse Descriptors	bool	False	Whether to parse the keypoint descriptors after generation
ROI	str	[empty]	Region of interest to process, in the form: x,y,w,h
Random Samples	bool	False	Whether to take random samples of keypoints
Rotation Invariant	bool	True	Whether to computer rotation invariant features
Scale Invariant	bool	True	Whether to compute scale invariant features

Sift

Generates SIFT descriptors for the input images. SIFT generates features that can be identified across varying imaging conditions, including scale, rotation, and illumination. The SIFT algorithm was developed by David Lowe (http://www.cs.ubc.ca/ lowe/keypoints/).

Property Name	Data Type	Default Value	Description
CPUs	int	4	Number of CPUs to utilize
Force Run	bool	False	Force run if outputs already exist
Parse Descriptors	bool	False	Whether to parse the keypoint descriptors after generation
Sift Method	enum	SiftWin32	Sift implementation to use {SiftWin32, SiftHess, SiftGPU, VLFeat}

Surf

SURF (Speeded Up Robust Features) is a robust local feature detector, first presented by Herbert Bay et al. in 2006, that can be used in computer vision tasks like object recognition or 3D reconstruction. It is partly inspired by the SIFT descriptor. The standard version of SURF is several times faster than SIFT and claimed by its authors to be more robust against different image transformations than SIFT. SURF is based on sums of 2D Haar wavelet responses and makes an efficient use of integral images. (http://en.wikipedia.org/wiki/SURF)

Property Name	**Data Type**	**Default Value**	**Description**
Force Run	bool	False	Force run if outputs already exist
Hessian Threshold	int	500	Only features with hessian larger than that are extracted. good default value is 300-500 (can depend on the average local contrast and sharpness of the image)
Number Octave Layers	int	4	The number of layers within each octave
Number Octaves	int	3	The number of octaves to be used for extraction. With each next octave the feature size is doubled

Table A.5: Auto-documentation for **FeatureMatch** package

FeatureMatch

ASIFTMatch

Performs matching of ASIFT descriptors.

Property Name	Data Type	Default Value	Description
Force Run	bool	False	Force run if outputs already exist
Parse Matches	bool	False	Whether to parse the keypoint matches

KeyMatch

Performs keypoint matching of SIFT descriptors.

Property Name	Data Type	Default Value	Description
Force Run	bool	False	Force run if outputs already exist
Key Match Method	enum	KeyMatchFull	Key matching implementation to employ {KeyMatchFull, KeyMatchGPU}
Parse Matches	bool	False	Whether to parse the keypoint matches

Table A.6: Auto-documentation for **OpenCV** package

OpenCV

FeatureDetector

Generates features for images using the methods implemented in the OpenCV library. Note: the stage properties are dynamically loaded based on the selected feature detection algorithm. Therefore, it is not possible to include a complete set of properties for each algorithm in the table.

Property Name	Data Type	Default Value	Description
Descriptor	enum	SIFT	Descriptor type{SIFT, SURF, ORB, BRISK, BRIEF}
Descriptor:contrastThreshold	float	0.04	Double
Descriptor:edgeThreshold	float	10.0	Double
Descriptor:nFeatures	int	0	Int
Descriptor:nOctaveLayers	int	3	Int
Descriptor:sigma	float	1.6	Double
Detector	enum	SIFT	Detector type {FAST, STAR, SIFT, SURF, ORB, BRISK, MSER, GFTT, HARRIS, Dense, SimpleBlob, GridFAST, GridSTAR, GridSIFT, GridSURF, GridORB, GridBRISK, GridMSER, GridGFTT, GridHARRIS, GridDense, GridSimpleBlob}
Detector:contrastThreshold	float	0.04	Double
Detector:edgeThreshold	float	10.0	Double
Detector:nFeatures	int	0	Int
Detector:nOctaveLayers	int	3	Int
Detector:sigma	float	1.6	Double
Force Run	bool	False	Force run if outputs already exist

FeatureMatcher *Performs feature matching using the methods implemented in the OpenCV library.*

Property Name	Data Type	Default Value	Description
Distance Threshold	float	0.5	Threshold as a percentage of mean distance
Force Run	bool	False	Force run if outputs already exist
Matcher	enum	BruteForce	Matcher type {BruteForce, BruteForce-L1, FlannBased}
Matches Path	str	[empty]	Path to matches output file

Table A.7: Auto-documentation for **Sources** package

Sources

ImageConvert

Converts images to a desired file format.

Property Name	Data Type	Default Value	Description
Image Extension	str	[empty]	Image extension
Image Path	str	[empty]	Output image path
Mode	enum	PIL	Conversion mode {PIL}

ImageFilter

Filters a source of images according to date/time and optionally randomizes the list.

Property Name	Data Type	Default Value	Description
Day Month	str	[empty]	Day and month of filter
End Time	str	[empty]	Ending time of filter
Randomize	bool	False	Whether to randomize the image list
Skip Images	int	0	Number of images to skip
Start Time	str	[empty]	Starting time of filter

ImageRandom

Randomizes the list of images.

ImageRename

Renames input images.

Property Name	Data Type	Default Value	Description
Base Name	str	[empty]	Base name of images

Move Files	bool	True	Whether to move files, if not copy
Output Path	str	[empty]	Output image path

ImageSource

Provides a source of images from disk.

Property Name	Data Type	Default Value	Description
Focal Pixel Override	int	0	Focal pixel value when metadata not found
Image Extension	str	[empty]	Image extension
Image Path	str	[empty]	Path to images
Recursive	bool	False	Whether to perform a recursive search

ImageSubset

Creates a subset of the input images according to the properties provided.

Property Name	Data Type	Default Value	Description
Increment	int	1	Index increment
Max Images	int	0	Maximum images in the output set
Start Index	int	0	Starting index of subset

ImageSymLink

Creates symbolic links for a list of images.

Property Name	Data Type	Default Value	Description
Delete Existing Links	bool	False	Whether to delete existing symbolic links
Link Keys	bool	False	Whether to attempt linking corresponding key files
Symbolic Link Path	str	[empty]	Path to create symbolic links

Bibliography

[1] David G. Lowe, "Distinctive image features from scale-invariant keypoints," *Int. J. Comput. Vision*, vol. 60, no. 2, pp. 91–110, Nov. 2004.

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

[3] M. Haklay and P. Weber, "Openstreetmap: User-generated street maps," *Pervasive Computing*, vol. 7, no. 4, pp. 12–18, Oct. 2008.

[4] Changchang Wu, "Visualsfm: A visual structure from motion system," Oct. 2011, `http://homes.cs.washington.edu/~ccwu/vsfm/`.

[5] A.J. Rossi, H. Rhody, C. Salvaggio, and D.J. Walvoord, "Abstracted workflow framework with a structure from motion application," in *Image Processing Workshop (WNYIPW), 2012 Western New York*, 2012, pp. 9–12.

[6] Noah Snavely, Steven M. Seitz, and Richard Szeliski, "Photo tourism: exploring photo collections in 3d," *ACM Trans. Graph.*, vol. 25, no. 3, pp. 835–846, July 2006.

[7] Yasutaka Furukawa, Brian Curless, Steven M. Seitz, Richard Szeliski, and Google Inc, "Towards internet-scale multi-view stereo," in *Proceedings of IEEE CVPR*, 2010.

[8] Yasutaka Furukawa and Jean Ponce, "Accurate, dense, and robust multi-view stereopsis," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 32, no. 8, pp. 1362–1376, 2010.

[9] Intel, "Intel vtune amplifier xe 2013," Feb. 2014, `http://software.intel.com/en-us/intel-vtune-amplifier-xe`.

[10] IBM, "Rational purifyplus," Feb. 2014, `http://www.ibm.com/developerworks/rational/library/957.html`.

[11] Coders Notes (Kayamon), "Very sleepy," Feb. 2014, `http://www.codersnotes.com/sleepy/`.

[12] Rice University, "Hpctoolkit," Feb. 2014, `http://hpctoolkit.org/`.

[13] Microsoft, "Beginners guide to performance profiling," Mar. 2014, `http://msdn.microsoft.com/en-us/library/ms182372.aspx`.

[14] Kitware, "Cmake, cross-platform, open-source build system," `http://www.cmake.org/`.

[15] "You can use freeze to compile executables for unix systems.," Feb. 2014, `https://wiki.python.org/moin/Freeze`.

[16] Mark Hammond Thomas Heller, Jimmy Retzlaff, "py2exe.org," Feb. 2014, `http://www.py2exe.org/`.

[17] Guoshen Yu and Jean-Michel Morel, "ASIFT: An Algorithm for Fully Affine Invariant Comparison," *Image Processing On Line*, vol. 2011, 2011.

[18] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool, "Speeded-up robust features (surf)," *Comput. Vis. Image Underst.*, vol. 110, no. 3, pp. 346–359, June 2008.

[19] Motilal Agrawal, Kurt Konolige, and MortenRufus Blas, "Censure: Center surround extremas for realtime feature detection and matching," in *Computer Vision ECCV 2008*, David Forsyth, Philip Torr, and Andrew Zisserman, Eds., vol. 5305 of *Lecture Notes in Computer Science*, pp. 102–115. Springer Berlin Heidelberg, 2008.

[20] Stefan Leutenegger, Margarita Chli, and Roland Siegwart, "BRISK: Binary robust invariant scalable keypoints," in *Proceedings of the IEEE International Conference on Computer Vision*, 2011.

[21] Per-Erik Forssén and David Lowe, "Shape descriptors for maximally stable extremal regions," in *IEEE International Conference on Computer Vision*, Rio de Janeiro, Brazil, October 2007, vol. CFP07198-CDR, IEEE Computer Society.

[22] Engin Tola, Vincent Lepetit, and Pascal Fua, "A fast local descriptor for dense matching," in *Conference on Computer Vision and Pattern Recognition*, Alaska, USA, 2008.

[23] Edward Rosten and Tom Drummond, "Machine learning for high-speed corner detection," in *In European Conference on Computer Vision*, 2006, pp. 430–443.

[24] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary R. Bradski, "Orb: An efficient alternative to sift or surf.," in *ICCV*, Dimitris N. Metaxas, Long Quan, Alberto Sanfeliu, and Luc J. Van Gool, Eds. 2011, pp. 2564–2571, IEEE.

[25] Konstantinos G. Derpanis, "The Harris Corner Detector," 2004.

[26] Rob Hess, "An open-source siftlibrary," in *Proceedings of the International Conference on Multimedia*, New York, NY, USA, 2010, MM '10, pp. 1493–1496, ACM.

[27] A. Vedaldi and B. Fulkerson, "VLFeat: An open and portable library of computer vision algorithms," `http://www.vlfeat.org/`, 2008.

[28] G. Bradski, "Opencv library," *Dr. Dobb's Journal of Software Tools*, 2000.

[29] Changchang Wu, "SiftGPU: A GPU implementation of scale invariant feature transform (SIFT)," `http://cs.unc.edu/~ccwu/siftgpu`, 2007.

[30] Martin A. Fischler and Robert C. Bolles, "Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography," *Communications of the ACM*, vol. 24, no. 6, pp. 381–395, 1981.

[31] M.I. A. Lourakis and A.A. Argyros, "SBA: A Software Package for Generic Sparse Bundle Adjustment," *ACM Trans. Math. Software*, vol. 36, no. 1, pp. 1–30, 2009.

[32] D. Marquardt, "An algorithm for least-squares estimation of nonlinear parameters," *Journal of the Society for Industrial and Applied Mathematics*, vol. 11, no. 2, pp. 431–441, 1963.

[33] Sameer Agarwal, Keir Mierle, and Others, "Ceres solver," `https://code.google.com/p/ceres-solver/`.

[34] Jianbo Shi and Jitendra Malik, "Normalized cuts and image segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, pp. 888–905, 1997.

[35] Chris Harris and Mike Stephens, "A combined corner and edge detector," in *In Proc. of Fourth Alvey Vision Conference*, 1988, pp. 147–151.

[36] Derek J. Walvoord, Adam J. Rossi, Bradley D. Paul, Bernie Brower, and Matthew F. Pellechia, "Geoaccurate three-dimensional reconstruction via image-based geometry," 2013.

[37] Derek J. Walvoord, Adam J. Rossi, Bradley D. Paul, and Bernie Brower, "Geoaccurate three-dimensional reconstruction via image-based geometry," 2013-05-03.

[38] Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe, "Poisson surface reconstruction," in *Proceedings of the fourth Eurographics symposium on Geometry processing*, Aire-la-Ville, Switzerland, Switzerland, 2006, SGP '06, pp. 61–70, Eurographics Association.

[39] Heiko Hirschmller, "Accurate and efficient stereo processing by semi-global matching and mutual information," *2012 IEEE Conference on Computer Vision and Pattern Recognition*, vol. 2, pp. 807–814, 2005.

[40] Paolo Cignoni, Massimiliano Corsini, and Guido Ranzuglia, "Meshlab: an open-source 3d mesh processing system," *ERCIM News*, , no. 73, pp. 45–46, April 2008.

[41] Ton Roosendaal, "Blender 3d computer graphics software," Oct. 2012, `http://http://www.blender.org/`.

[42] Daniel Girardeau-Montaut, "3d point cloud and mesh processing software," Feb. 2014, `http://www.danielgm.net/cc/`.

[43] Radu Bogdan Rusu and Steve Cousins, "3d is here: Point cloud library (pcl).," in *ICRA*. 2011, IEEE.

[44] Adam J. Rossi, "Catena: Python abstract workflow framework," Mar. 2013, `http://catena.googlecode.com/`.

Acronyms

API
application programming interface

CIS
Center for Imaging Science

CMVS
cluster-based multi-view stereo software

DLT
direct linear transformation

DoG
Difference of Gaussian

DSP
digital signal processor

exif
exchangable image file format

FPGA
field programmable gate array

GIS
geographic information systems

GPS
global positioning system

GPU
graphics processing unit

GSD
ground sample distance

GUI
graphical user interface

INS
inertial navigation system

L-M
Levenberg-Marquardt

MVS
multi-view stereo

NCC
normalized cross correlation

OSM
Open Street Map

PCL
point cloud library

PIL
Python imaging library

PMVS
patch-based multi-view stereo software

RANSAC
random sample consensus

RIT
Rochester Institute of Technology

sba
sparse bundle adjustment

SfM
structure from motion

SGM
Semi-Global Matching

SIFT
scale-invariant feature transform

SVD
singular value decomposition

SVN
Subversion

WAMI
Wide Area Motion Imagery

WASP
Wildfire Airborne Sensor Program

Index