Catena
Programmer's Guide

Adam J. Rossi

February 18, 2013

Contents

1 Introduction

Catena is a workflow framework implemented in the Python programming language that allows for the development of stages and chains. The stages encapsulate functionality that can be generalized and employed in various applications. The chain is a collection of stages connected together to perform a specific task. Catena ensures consistent stage connectivity by enforcing interface definitions. It also provides efficient chain rendering by employing caching and distributed computing techniques.

This document serves as a programmer's guide to implementing Catena stages and explains the usage of common components in the framework.

2 Stage Implementation

This section will illustrate how to implement a stage for use with the Catena framework. The 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 in that it will simply pass the images input to the output.

The code below 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)
```

2.1 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 inherit. The `Common` module contains data types that are typically used throughout the chain.

```
import Chain
import Common
```

2.2 Class Definition

Next, the class is defined and as mentioned previously, all Catena stages must inherit from `Chain.StageBase`, as this is where the power and underlying stage functionality exists.

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

2.3 Constructor

The convention in Catena is for the constructor to take a list of input stages as the first parameter. The following parameters are optional and specific to the stage being implemented. Another requirement is for each parameter to be given a default parameter; this is dual purpose. It provides a nominal parameter setting in the event the user does not wish to override the default value. Also, given that Python is a loosely typed language, it provides important datatype information to other software components.

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

2.4 Constructor Implementation

The base class' constructor must be called immediately. The constructor takes 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, that offer users online help.

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

2.5 Stage Properties

Lastly, the properties dictionary is initialized using the parameters of the stage, provided as parameters to the constructor. The user is free to carry out any other initialization procedure required for the stage.

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

2.6 Input Interface

The `GetInputInterface` method must be implemented (overloaded) as part of the `StageBase` contract. 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 error checks to minimize user errors, both in development and application.

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

2.7 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 datatype of the output parameter.

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

2.8 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):
```

2.9 Execution Implementation

Each `Execute` method typically follows the same pattern.

1. Get input parameter from input stages (GetInputStageValue)
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 (SetOutputValue)

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 upstream from the end stage, as this is a demand-pull render model. The stage indices and parameter names (and datatypes) 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 to maintain timing information for chain rendering.

```
self.StartProcess()
```

The main work of the stage is carried out next. The text file path specified as an input property (Resolution Path) is accessed using the properties dictionary, opening a text file for write. The images that were obtained from the input stage are iterated through and the file name and resolution information are written for each image. Finally, 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()
```

Last, we set the output parameter values using the `SetOutputValue` method. The parameter names (and datatypes) must be consistent with the interface defined in the `GetOutputInterface` method.

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

3 Chain Implementation

The following section will utilize the structure-from-motion (SfM) stages that have been developed in order to carry out a 3D modelling task. The script below represents the complete implementation of the SfM chain. However, it will be broken down and explained thoroughly in the following subsections.

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

3.1 Import Modules

First, the modules used throughout the script to build the 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 basically finds all occurrences of classes which inherit from the `Chain.StageBase` class. Then, the packages which contain the stage definitions used during the chain

building are imported.

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

3.2 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")
```

3.3 Chain Construction

The following steps create an instance of each stage in the chain. One should observe the pattern of creating a stage instance and then passing the object to the following stage instance as its input stage. Also, each stage defines different properties, provided as parameters to the constructor.

3.3.1 ImageSource

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.

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

3.3.2 SIFT

The scale-invariant feature transform (SIFT) [1] stage accepts a list of images and generates a keypoint descriptor files for each, which contain feature vectors of salient points in the image. The keypoint descriptors collection is represented as a class and pass 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")
```

3.3.3 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 which represent the keypoint matches. 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")
```

3.3.4 Bundle Adjustment

The bundle adjustment stage is an abstraction of the Bundler [2] program. It accepts the keypoint matches and images as input, generating a proprietary bundler output file, which is represented as a class within the programming environment.

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

3.3.5 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 in attempt to remove the distortion. A new bundler file is generated, along with a new collection that represents the undistorted images.

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

3.3.6 CMVS/PMVS Preparation

The cluster-based multi-view stereo software (CMVS) and patch-based multi-view stereo software (PMVS) programs expect their input to be in a particular form that is different from Bundler's output. Therefore, this preparation stage is used prepare 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 camera matrices that are both computed from the Bundler file.

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

3.3.7 CMVS

CMVS [3] requires a Bundler file and image collection as its input. It runs a clustering algorithm which reduces the input image set. It outputs a Bundler file, image collection, "vis" file, "cluster" file, and "camera centers" file.

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

3.3.8 PMVS

PMVS [4] 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)
```

3.4 Chain Rendering

Finally, the chain is rendered by calling the `Chain.Render` method and providing the last stage from which to render. The method also accepts a string parameter which specifies where the log file shall be written.

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

3.5 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. The following section illustrates how to restore the persist file and render the chain.

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

3.6 Persisted Chain Rendering

The following script 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 apriori knowledge of the chain structure. Since there exists a single tail stage from the chain that was built, the PMVS stage is located at index zero in the list.

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

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

4 Stage Development Patterns

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

4.1 Conditional Execution

A convenience method has been included in the `Utility` package that can be used to determine if the core stage functionality should be executed. 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 (`forceRun`) 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)
```

4.2 External Program Execution

A stage typically represents an external application that takes a set of command line arguments for execution. A method named `RunCommand` has been provided on the `StageBase` class for ease of implementation. 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, this will be explained further later. The next parameter is a string of the command line arguments. The `CommandArgs` and `Quoted` methods are provided for convenience and will be discussed later. The user is also free to 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])
```

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.

5 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). As illustrated below, the `TapPoint` stage takes a single stage as the input and an optional dictionary of print functions. The print function dictionary is keyed by type; 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 no print dictionary is provided, the overloaded string method on the object will be utilized, this is illustrated in the second tap point stage instance.

```
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")
```

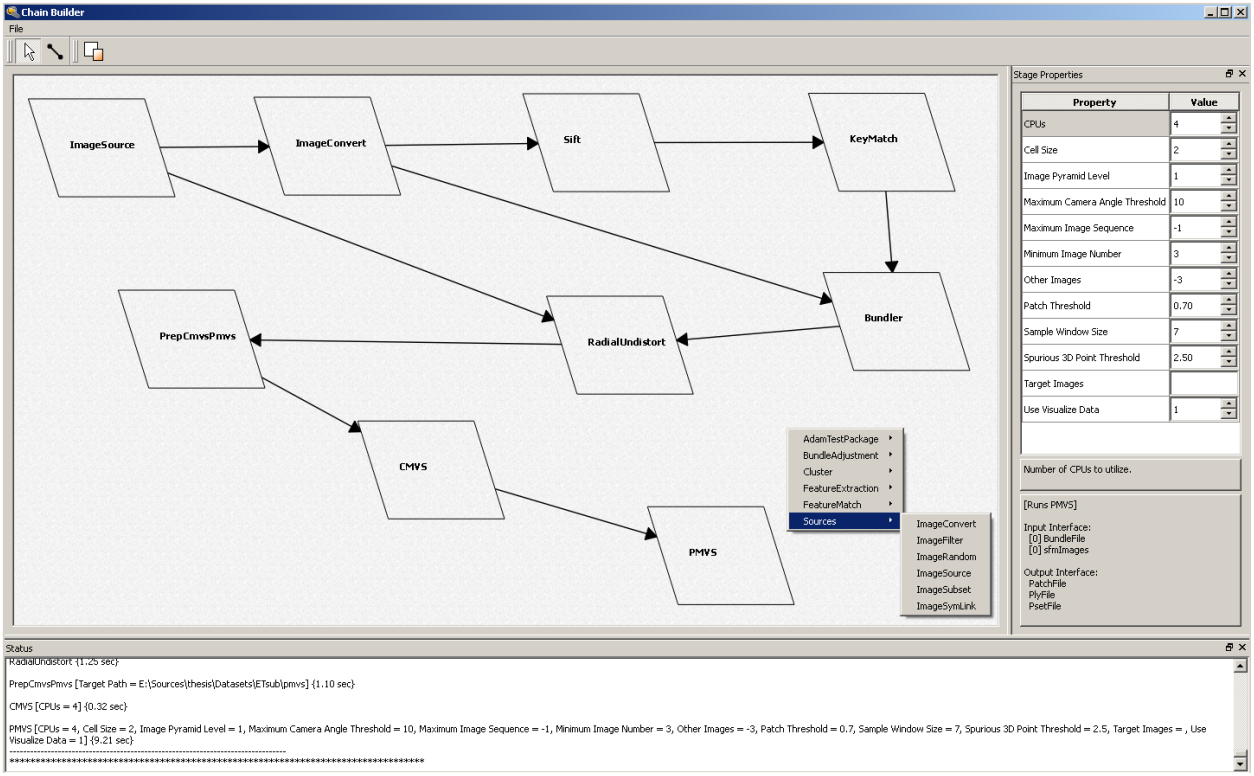
6 Unit Test

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

7 Chain Builder

The Chain Builder tool allows for graphical building of chains. The list of stage packages is accessed by right-clicking on the canvas. A context menu of stages that 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.



8 Dynamic 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.

1. BundleAdjustment

   Bundler Performs bundle adjustment on the given images

   Constrain Focal
   Add a soft constraint on focal lengths to stay near their estimated values (bool)

   Constrain Focal Weight
   Strength of the focal length constraints (float)

Estimate Radial Distortion
Whether to estimate radial distortion parameters (bool)

Fisheye Parameter File
Fisheye parameter file path (str)

Fixed Focal Length
Use fixed focal length (Initial Focal Length) (bool)

Force Run
Force run if outputs already exist (bool)

Initial Focal Length
Initial focal length (float)

Maximum Projection Error Threshold
The maximum value of the adaptive outlier threshold (int)

Minimum Projection Error Threshold
The minimum value of the adaptive outlier threshold (int)

Previous Bundler Results File
Previous bundle adjustment results file path (str)

Projection Estimation Threshold
RANSAC threshold when performing pose estimation to add in a new image (int)

Ray Angle Threshold
Triangulation ray angle threshold (int)

Run Slow Bundler
Run slow bundle adjustment (adds one image at a time) (bool)

Seed Image Index 1
First image index to seed bundle adjustment (int)

Seed Image Index 2
Second image index to seed bundle adjustment (int)

Trust Focal Estimate
Trust the provided focal length estimates (i.e., don't attempt to cross-check with self-calibration) (bool)

Use Focal Length Estimate
Initialize using focal length estimates specified in the list file (bool)

Variable Focal Length
Use variable focal length (bool)

2. Cluster

CMVS Runs CMVS

CPUs
Number of CPUs to utilize (int)

Force Run
Force run if outputs already exist (bool)

PMVS Runs PMVS

CPUs
Number of CPUs to utilize. (int)

Cell Size
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. (int)

Force Run
Force run if outputs already exist (bool)

Image Pyramid Level

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. (int)

Maximum Camera Angle Threshold

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. (int)

Maximum Image Sequence

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. (int)

Minimum Image Number

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. (int)

Other Images

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. (int)

Patch Threshold

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. (float)

Sample Window Size

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. (int)

Spurious 3D Point Threshold

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. (float)

Target Images

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. (str)

Use Visualize Data
Whether to use the visualize data. (int)

PrepCmvsPmvs Prepares directories for CMVS/PMVS

Force Run
Force run if outputs already exist (bool)

Target Path
Target path for CMVS/PMVS preparation (str)

RadialUndistort Radially undistorts images

Force Run
Force run if outputs already exist (bool)

3. Common

MuxStage A stage that 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

Print Functions
Dictionary of functions used to print parameters, keyed by type (dict)

4. FeatureExtraction

Daisy Generates Sift descriptors for images

Disable Interpolation
Whether to disable interpolation (bool)

Force Run
Force run if outputs already exist (bool)

Number Random Samples
If random samples is enabled, the number of samples to take (int)

Orientation Resolution
If computing rotation invariant features, number of bins to use (int)

Parse Descriptors
Whether to parse the keypoint descriptors after generation (bool)

ROI
Region of interest to process, in the form: x,y,w,h (str)

Random Samples
Whether to take random samples of keypoints (bool)

Rotation Invariant
Whether to computer rotation invariant features (bool)

Scale Invariant
Whether to compute scale invariant features (bool)

Sift Generates Sift descriptors for images

Force Run
Force run if outputs already exist (bool)

Parse Descriptors
Whether to parse the keypoint descriptors after generation (bool)

Sift Method
Sift implementation to use {SiftWin32, SiftHess, SiftGPU, VLFeat} (str)

5. FeatureMatch

KeyMatch Performs keypoint matching

Force Run
Force run if outputs already exist (bool)

Key Match Method
Key matching implementation to employ {KeyMatchFull, KeyMatchGPU} (str)

Parse Matches
Whether to parse the keypoint matches (bool)

6. Sources

ImageConvert Converts images to a desired format

Image Extension
Image extension (str)

Image Path
Output image path (str)

Mode
Conversion mode {PIL} (str)

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

Day Month
Day and month of filter (str)

End Time
Ending time of filter (str)

Randomize
Whether to randomize the image list (bool)

Skip Images
Number of images to skip (int)

Start Time
Starting time of filter (str)

ImageRandom Randomizes the source of images

ImageRename Renames the source of images

Base Name
Base name of images (str)

Move Files
Whether to move files, if not copy (bool)

Output Path
Output image path (str)

ImageSource Provides a source of images from disk

Focal Pixel Override
Focal pixel value when metadata not found (int)

Image Extension
Image extension (str)

Image Path
Path to images (str)

Recursive
Whether to perform a recursive search (bool)

ImageSubset Takes a subset of the source of images

Max Images
Maximum images in the output set (int)

ImageSymLink Creates symbolic links for a list of images

Delete Existing Links
Whether to delete existing symbolic links (bool)

Link Keys
Whether to attempt linking corresponding key files (bool)

Symbolic Link Path
Path to create symbolic links (str)

9 Distributed Extension

TODO

References

[1] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *Int. J. Comput. Vision*, vol. 60, pp. 91–110, Nov. 2004.

[2] N. Snavely, S. M. Seitz, and R. Szeliski, "Photo tourism: exploring photo collections in 3d," *ACM Trans. Graph.*, vol. 25, pp. 835–846, July 2006.

[3] Y. Furukawa, B. Curless, S. M. Seitz, and R. Szeliski, "Towards internet-scale multi-view stereo," in *CVPR*, 2010.

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