Terasense software. Python API reference

Terasense Group, Inc
21143 Hawthorne Blvd., #459, Torrance, CA 90503, USA

December 15, 2015

Contents

1 Installation

Terasense software is a http://www.python.org/ package. It requires Python and the following extension modules to be installed:

- Python 2.7 http://www.python.org/download/
- NumPy 1.7 http://www.numpy.org/
- OpenCV 2.4 http://opencv.org with Python bindings
- wxPython 2.9 http://www.wxpython.org
- Pyserial 2.5 http://sourceforge.net/projects/pyserial/files/

It also depends on http://www.microsoft.com/en-us/download/details.aspx?id=5555

Installers for all required software are provided in default installation package. Download it from the provided link, unzip it to a temporary folder, and execute *install.bat* script by double-clicking it in Windows Explorer, it will run all necessary installers for you. Install all packages with default settings, agreeing to EULA where required.[1]

Terasense software won't work with Python 3.x because it is not supported by NumPy. However, if you use Python 3.3 or newer, you can seamlessly install Python 2.7 alongside Python 3.x (see http://www.python.org/dev/peps/pep-0397/ for additional information).

[1] By default most of the stuff will be installed to the folder C:\Python27\ and take about 180 MB of the disk space. If you want to change the location, you should do that in the very first installer to run, the one for the Python itself. Destinations for all other packages will be changed accordingly.

2 Overview

Two main modules are supplied in the *terasense* package, *terasense.processor* and *terasense.worker*. The first provides *processor* class for data acquisition and basic processing (including background compensation and normalizaion); the second provides *worker* class, which can be used to convert data to a RGB image.

terasense.processor module can work in two modes — multi-threaded or single-threaded. In the single-threaded mode data acquisition and data processing are performed consequently. It is simple and cause no potential pitfalls associated with multi-threading. The multi-threaded mode performs data acquisition asynchronously, while data processing is running in parallel; it may be up to two times faster at short exposures. It should cause no problems in simple usage scenarios, but if you want to use it for something complex, you should get acquinted with Python's *threading* module.

Data processing consists of three main parts — background compensation, normalization, and stitching of the holes caused by non-performing pixels. It if performed in *terasense.processor* module.

Background data are read from a config file, there is separate set for each exposure, which is automatically selected when exposure is changed. They can be re-recorded and it should be done if external temperature is changed. Obviously, incoming radiation should be switched off. Normalizaion data are stored in a dictionary; there is a default set, supplied from the factory, but you can record your own (which will be stored under the label "recorded") to take into account distribution of the incoming radiation and effectively flatten the field. Pixels that produce too low signal-to-noise ratio during recording (either due to defect or to being in dark spot) are marked as non-performing (the treshold separating performing from non-performing pixels can be changed after the recording). Readouts from non-performing pixels are substituted by values extrapolated their neighbours. If the number of the pixels is large, the procedure becomes cpu-intensive and may limit performance of the camera; you may switch off greedy stitching or stitching completely.

3 Module *terasense.processor*

class *terasense.processor*.**processor**([**threaded, config, defaults, flags**])

This class provides main data acquisition and processing capabilities.

Parameters:

threaded whether to use separate threads (via *threading* module) for aquisition and processing or not (boolean, default: *True*)

config path to a configuration file containing background and normalization data (default:*None*)

defaults path to a file with auxilliary default settings (default:*None*)

flags data processing flags (default: *terasense.processor*.DEFAULT_FLAGS)

Data processing flags:

STITCH heal over isolated missing pixels

GREEDYSTITCH heal over large area of missing pixels

ACCUMULATE turn on accumulation (time-domain filtering)

DIFFERENCE turn on the difference mode

DEFAULT_FLAGS default value, equivalent to STITCH|GREEDYSTITCH

3.1 Methods

The module provides methods in several groups:

Main:

- start([callback, errorcallback, resume])
- stop([join])
- read()

Exposure:

- SetExposure(exp)
- GetExposure()
- GetExposureRange()
- GetIntTime([exp])

Background and normalization:

- SetBG(data)
- SelectBG(val)
- GetSelectedBG()

- SetNorm([data, mask])
- GetNormList()
- SelectNorm([val])
- GetSelectedNorm()
- LoadConfig([zipName])
- SaveConfig(filename)
- RecordCurrentBG([count, ticker])
- RecordBG([count, callback])
- RecordNorm([count, callback])

Processing

- SetStitch([on])
- SetGreedyStitch([on])
- SetThreshold(val)
- GetThreshold()
- SetAccumulation([on])
- SetAccuLength(val)
- GetAccuLength()
- ResetAccumulation()
- SetDifference([on])

3.1.1 Main

start([callback, errorcallback, resume])

If **processor** instance have been created in a multi-threaded mode, starts acquisition and processing threads. Otherwise has no effect.

Parameters:

callback a target to be called on each aquisition cycle as *callback(data)*, where *data* is a *numpy.ndarray* containing processed data within [0,1] interval (default: *None*);

errorcallback a target to be called in case of an error as *errorcallback(error, critical)*, where *error* is an instance of *Exception*, and *critical* is a boolean value indicating wheather operation should be stopped or may continue (default: *None*);

resume if *True* and *callback* is *None* the value of *callback* from previous call to this function will be used (default: *False*).

Returns:

None

stop([join])

If **processor** instance have been created in a multi-threaded mode, stops acquisition and processing threads. Otherwise has no effect.

Parameters:

join if *True*, it will try to join acquisition and processing threads (default: *True*).

Returns:

None

read()

Reads processed data. If **processor** instance have been created in a multi-threaded mode, the method will work only if acquisition have been started using *start*, otherwise it will return *None*. This method will not read the same frame twice and it will block if the new frame is not available yet.

Returns:

data *numpy.ndarray* containing processed data within [0,1] interval.

read_raw()

Reads raw unprocessed data. If **processor** instance have been created in a multi-threaded mode, the method will work only if acquisition was started using *start*, otherwise it will return *None*. This method is not intended for end-user.

Returns:

data *numpy.ndarray* containing unprocessed data.

3.1.2 Exposure-related

SetExposure(exp)

Sets current exposure. If running in a multi-threaded mode, the function will return immediately, but exposure would be actually changed only for the next frame. Background compensation is selected automatically.

Parameters:

exp new exposure number (integer, call **GetExposureRange** to get supported range; standard is from 0 to 10 inclusive).

Returns:

error *None* for success or *Exception* instance for failure if not running in multi-threaded mode; otherwise always returns *None* and in the case of failure *errorcallback* is called by processing thread instead.

GetExposure()
Gets current exposure.

Returns:

exp current exposure number

GetExposureRange()
Gets available range of exposure numbers.

Returns:

(min, max) a tuple with minimal and maximal available numbers.

GetIntTime([exp])
Gets integration time in microseconds for a given exposure.

Parameters:

exp exposure number; if none is porvided, current exposure is used.

Returns:

duration integration time in microseconds. It is proportional to amplification and is about 1/32 of time required for acquisition of a frame.

Warning: this function is not thread-aware; if you're calling it without parameter immediately after *SetExposure()* while running in multi-threaded mode, you'll probably get value for a previous exposure number. Use explicit parameter.

3.1.3 Background and normalization-related

SetBG(data)
Sets data as current backgound data to be used in processing.

Parameters:

data *numpy.ndaray* with data to be used as a background data. If it is *None*, empty array is used (i.e. no background compensation is performed).

Returns:

None

SelectBG(val)
Selects a background data from existing list. The data are set as current backgound data to be used in processing.

Parameters:

val index of the data in the list (corresponds to exposure); if it is out of range, empty array is used.

Returns:

None

GetSelectedBG()

Gets index of a currently selected background data within the list.

Returns:

idx index of a currently selected background; *None* if no background compensation is performed.

SetNorm([data, mask])

Sets current normalization data to be used in processing.

Parameters:

data *numpy.ndaray* with data to be used for normalization. If it is *None*, empty array is used (i.e. no normalization is performed).

mask *numpy.ndaray* with corresponding mask data to be used for normalization. If it is *None*, all pixels are assumed to be good.

Returns:

None

SelectNorm([val])

Selects normalization from the dictionary by the key (either "default", "recorded", or *None*)

Parameters:

val key value. If the key is *None* or does not exists normalization is switched off.

Returns:

key value of the key on success or *None* otherwise.

GetNormList()

Gets a list of normalizations as dictionary. It includes all possible keys and boolean values indicate whether the corresponding normalizaion is available at the moment.

Returns:

dict dictionary of a form {"default": *True*— *False*, "recorded": *True*— *False*}

GetSelectedNorm()

Gets key for currently selected normalization.

Returns:

key key for the currently selected normalization ("default" or "recorded") or *None* if none is selected.

LoadConfig([zipName])

Loads previously saved background and normalization data from a configuration file and puts them into the corresponding list and dictionary.

Parameters:

zipName filename for a configuration file. If it is not provided, default configuration file is loaded. If it is a relative path, it is relative to the module folder.

Returns:

None

SaveConfig(filename)

Saves complete list of backgrounds and normalization data dictionary to a configuration file to be loaded later.

Parameters:

filename filename for a configuration file. If it is a relative path, it is relative to the module folder.

Returns:

None

RecordCurrentBG([count, ticker])

Record background data for current exposure.

Parameters:

count number of repetitions used for averaging (default: 30).

Returns:

refdata an instance of *terasense.ref.RefData* object with the background information.

RecordBG([count, callback])

Record background data for all available exposures and places them into the background list.

Parameters:

count number of repetitions used for averaging. If 0 or not provided, then default value is used (30, may be changed in defaults file).

callback callback to indicate progress. It is called with completed percentage and it is expected to return tuple (continur, skip), where continue is True unless the process should be aborted (see wx.ProgressDialog from wxPython package).

Returns:

success *True* if completed successfully, *False* if canceled.

Warning: this function requires significant time to be completed — several minutes at default parameters.

RecordNorm([count, callback])

Record normalization data and put them into the normalization dictionary under "recorded" key.

Parameters:

count number of repetitions used for averaging. If 0 or not provided, then default value is used (30, may be changed in defaults file).

callback callback to indicate progress. It is called with completed percentage and it is expected to return tuple (continur, skip), where continue is True unless the process should be aborted (see wx.ProgressDialog from wxPython package).

Returns:

success *True* if completed successfully, *False* if canceled.

3.1.4 Processing

SetStitch([on])

Turns stitching (healing over isolated missing pixels) on or off.

Parameters:

on *True* to turn on, *False* to turn off (default: *True*).

Returns:

None

SetGreedyStitch([on])

Turns on greedy stitching (healing over large area of pixels) on or off. Has no effect if stitching is off.

Parameters:

on *True* to turn on, *False* to turn off (default: *True*).

Returns:

None

SetThreshold(val)

Sets threshold value, which separates performing from non-performing pixels in a mask. The value roughly corresponds to signal-to-noise ratio under normalization conditions.

Parameters:

val new value for threshold (default: 10.0, may be changed in defaults file).

Returns:

None

GetThreshold()

Returns current threshold value.

Returns:

val current threshold value.

SetAccumulation([on])

Turns accumulation on or off.

Parameters:

on *True* to turn on, *False* to turn off (default: *True*).

Returns:

None

SetAccuLength(val)
Sets accumulation length (window size for time-domain filtering).

Parameters:

val accumulation length (coerced to [1,100]).

Returns:

None

This method does not turn accumulation on! Use SetAccumulation().

GetAccuLength()
Returns current accumulation length (window size for time-domain filtering). Result does not depend on whether the accumulation is on or off.

Returns:

val current accumulation length.

ResetAccumulation()
Resets accumulated data (i.e. starts accumulation anew).

Returns:

None

SetDifference([on])
Turns the difference mode on or off.

Parameters:

on *True* to turn on, *False* to turn off (default: *True*).

Returns:

None

3.2 Properties

3.2.1 General

X_SIZE X dimension of the sensor array (integer, read-only).

Y_SIZE Y dimension of the sensor array (integer, read-only).

bgList background list containing background information for each available exposure. Each item is an instance of *terasense.ref.RefData*. See SelectBG, GetSelectedBG, LoadConfig, RecordCurrentBG, RecordBG methods.

normDict normalization dictionary containing normalization information. By default it contains "default" and "recorded" keys (with values possibly being *None*). Each item is an instance of *terasense.ref.RefData*. See GetNormList, SelectNorm, GetSelectedNorm, LoadConfig,

RecordNorm methods.

3.2.2 Multi-threading

Generally, it is recommended to use either callback of the start method or read method to get access to the data in multi-threaded mode. However, if you want to have a direct access, here are several properties to do that.

result *numpy.ndarray* with the shape (X_SIZE, Y_SIZE), which contains processed data during multi-threaded operation.

datalock instance of *threading.Lock()*. Acquire it if you access *result* property directly.

ready instance of *threading.Event()*. It is set when *result* property is renewed.

4 Module *terasense.worker*

class *terasense.worker.***Worker(size, [flags])**

This class provides means for converting data array to an image with some additional processing. It relies on *Numpy* and *OpenCV*.

Parameters:

size a tuple with dimensions of the imag/data array (width, height)

flags processing flags (default: *terasense.processor*.DEFAULT_FLAGS)

Data processing flags:

FALSECOLOR produce image in false colors (rainbow) instead of b/w (tinted)

SMOOTH smooth image (space-domain filtering)

NEGATIVE invert image

MEDIAN use median filtering instead of gaussian blurring for smoothing

MIRROR mirror the image

DEFAULT_FLAGS default value, equivalent to FALSECOLOR|SMOOTH

4.1 Methods

The module provides the following methods:

- makeImg(data)
- SetBrightness(black,white)
- SetContrast(black,white)
- SetGamma(val)
- SetSmoothness(val)
- GetSmoothness()
- data2RGB(data)
- statistics(data, [selection])

makeImg(data) Generates image from data according to the current settings.

Parameters:

data input data (one-channel [0,1] array of floats).

Returns:

img three-channel RGB image.

SetBrightness(black,white) Sets brightness using black and white points.

Parameters:

black black point value in [0, white)

white white point value in (black, 1]

Returns:

None

SetContrast(black,white) Sets contrast using black and white points.

Parameters:

black black point value in [0, white)

white white point value in (black, 1]

Returns:

None

SetGamma(val) Sets gamma value.

Parameters:

val gamma value (gamma ¿ 0).

Returns:

None

SetSmoothness(val) Sets smoothness parameter.

Parameters:

val smoothness parameter (0, 100]. For gaussian blur smoothing it is the standard deviation $\times 100$, for median smoothing it sets 3×3 kernel if $val \leq 50$ and 5×5 kernel otherwise.

Returns:

None

GetSmoothness() Gets smoothness value.

Returns:

val value of the smoothness parameter (0, 100]

data2RGB(data) Converts one-channel BW data to three-channel RGB data, output depends on the presense of FALSECOLOR flag in *processFlags* property. Parameters:

data input data (one-channel [0,1] array of floats).

Returns:

img three-channel RGB image.

4.2 Properties

processFlags mask, which defines processing options according to data processing flags (see description of the constructor). It may be changed at any time.

brightness brightness parameter, float, possible values in $[-1, 1]$ range.

contrast contrast parameter, float, possible values in $(0, \text{inf})$ range.

size tuple with 2D dimensions of the data (width, height) (read-only).

5 Examples

5.1 The simplest program

```
from terasense import processor as tp
source = tp.processor(False) # no multithreading
source.SetExposure(3)
for i in range(100):
    data = source.read()
    #do something with data
```

5.2 Multithreaded version

Data acquisition from the camera is performed in a separate thread. In this case you need to start acquisition explicitly, otherwise *read()* would return *None*.

```
from terasense import processor as tp
source = tp.processor() # multithreading is on by default
source.SetExposure(3)

#In the multithreaded case you need to start data acquisition explicitly
source.start()
for i in range(100):
    data = source.read()
    #do something with data

#Call stop() to stop acquisition and join the acquisition and processing thread
source.stop()
```

5.3 Multithreaded version using callbacks

You can provide callback function to be called each time new frame became available. Callback function is executed in the same thread as data processing.

```
from terasense import processor as tp
import time
count = 0
def callback(data):
    global count
    #do something with data
    count +=1
    print count,

source = tp.processor() # multithreading is on by default
source.SetExposure(3)

#In the multithreaded case you need to start data acquisition explicitly
```

```
source.start(callback)

# Sleep 10 seconds or do something while acquisition is going on
time.sleep(10)

#stop() function should be called from the main thread, not from the callback
source.stop()
```

5.4 Image generation

```
from terasense import processor as tp
from terasense import worker as tw

source = tp.processor()
convert = tw.Worker(size = (source.X_SIZE,source.Y_SIZE))
source.SetExposure(3)

#In the multithreaded case you need to start data acquisition explicitly
source.start()
for i in range(100):
    data = source.read()
    img = convert.makeImg(data)
    #do something with image

#Call stop() to stop acquisition and join the acquisition and processing thread
source.stop()
```

5.5 Use of background and normalization

```
from terasense import processor as tp

def ticker(progress):
    if progress < 100:
        print ".",
    else:
        print " "
    return (True, False)

source = tp.processor()

raw_input("Prepare to record background. Switch off incoming radiation and press Enter")

#Background is recorded for all exposures, it will take several minutes.
# ay be you'll want to use SaveConfig/LoadConfig to avoid repeating the procedure.
source.RecordBG(callback = ticker)
```

```
raw_input("Prepare to record normalization. Switch on incoming radiation and press Enter")

#Do not forget to select desired exposure.
source.SetExposure(5)

# Normalization recording may raise an Exception if illumination condition are detected to be unsuitable
try:
    source.RecordNorm(callback = ticker)
except Exception as e:
    print e

#You are free to change exposure - all backgrounds are stored.
source.SetExposure(3)
source.start()
for i in range(100):
    data = source.read()
    #do something with data
source.stop() #call stop() to stop acquisition and join the acquisition and processing thread
```