3D Scanner with Photometric Stereo

Binbin Xiong
Visual Intel Studio
Carnegie Mellon University
bxiong@andrew.cmu.edu

Yang Cai
Visual Intel Studio
Carnegie Mellon University
ycai@cmu.edu

Created May 12, 2015

1 Overview

This software is used to generate 3D models based on photometric stereo technologies. It supports ball detection based auto calibration, distant model photometric stereo and also near range photometric stereo[1]. The software takes multiple images (each taken with only one light on) as input, then generate 3D model and output to OBJ or ASC format. Both Linux and Windows versions are provided.

2 Use the program

The executable takes one input parameter, that is the path of the configuration file. All inputs are set through this configuration file.

2.1 Configuration File

The entries in the configuration file are *key = value* pairs, while the key means the name of the parameter. Strings will be trimmed so all leading and tailing spaces will be ignored. There are two *comment symbol* includes # and ;, all contents appear on the right of the comment symbol within a line will be ignored.

A typical entry might look like this:

```
# locate the data folder:
DataFolder = D:/data/ # this value tells the program where to find the data
```

Note that the trailing slash is needed. Also, DONNOT change the key names.
To be specific, all the parameters in the configuration file are:

- **DataFolder**: means the folder that contains both the image data and the calibration data. Note that the trailing slash shall never be ignored.
- **ResultFolder**: this folder saves the intermediate output of the program, especially for the calibration process. Note that when you do the calibration, you should always check this folder to see the ball detection results. If it is not accurate, you might need to change some parameters. Refer to 3.1 for more information.
- **CalibrationPrefix**: this indicates the filename prefix of the calibration data. Before running the program, you should modify all the calibration images names to make them be of the form *Prefix1_#* (and *Prefix2_#* if you want to run the near range model calibration), here # means the index number of the corresponding light source, and it should always starts with 1 continuously. For example, if you have four lights, then you might name the calibration files as:

[1] Currently the calibration algorithm for near range PS is very unreliable.

Calib-1_1.JPG Calib-1_2.JPG Calib-1_3.JPG Calib-1_4.JPG

In this case, you should set

CalibrationPrefix = Calib-

Note that the reason why there is a number 1 or 2 before the light index is to make it compatible with the calibration for near range lighting algorithm. See 3.3 for more information.

- **DataImagePrefix**: this is the file name prefix for the object image. Like the calibration images, you also need to rename the filenames and make them be of the form *Prefix#*, where # means the index of the light source, which means the data image and the calibration image that has the same index number must be taken under the same light.
- **ImageSuffix**: the suffix of both the calibration images and the data images. They should be the same. For the example above, you should set ImageSuffix = .JPG
- **NumLights**: the number of lights. For the example above, you should set NumLights = 4
- **Mode**: this parameter set the running mode of the program. The program mainly has three running modes now, they are:

Mode	Value
Calibration	1
Photometric Stereo	2
Depth Subtration	3

For example, if you want to run the calibration, then you should set:

Mode = 1

- **PSmode**: this parameter tells the program to use distant PS mode or the near range PS mode. 1 means the distant PS, and 2 means the near range mode. **You are encouraged to use distant model now unless you know how to calibrate the exact position of the lights yourself.**
- **RescaleImg**: tell the program whether to rescale the input images. The function is activated only when you set this value to yes, that is RescaleImg = yes. Note that in most cases you have to rescale the image first, otherwise you may encounter Out-of-Memory exception.
- **RecaleImgWidth**: this parameter works only when you activated RescalImg. The program will fix the ratio when it rescales the images, and it will choose between this width value and the height value to achieve the smallest scale.(The scaled image will have a width no more than RecaleImgWidth and a height no more than RecaleImgHeight).
- **RecaleImgHeight**: refer to RecaleImgWidth.
- **OutputFormat**: this parameter tells the program which kind of output format to choose. Currently the program supports OBJ file format (can be opened by Meshlab)and ASC file format (can be opened by Matlab).
- **MaxIteration**: this parameter is for near range PS. It tells the maximum number of iterations when solving the depth.
- **ReferenceModel**: this parameter is only valid when you run the program under mode 3, that is the *Depth Subtraction* mode. This is the path to the *reference model file*. The file can be either OBJ format or ASC format.

- **SubtractModel**: this parameter is only valid when you run the program under mode 3, that is the *Depth Subtraction* mode. This is the path to the *generated model*. The file can either be OBJ or ASC format. However, you must ensure that the ReferenceModel and the SubtractModel are of the same format. The resulting subtracted model file will be output to the same folder as this model, and the file name will be the original model file name appends *_REVISED*.
- **FitErrorThresh**: this parameter is for calibration. It tells when the fit error between the current image and the best fitted image is above what value should whether to re-run the ball detection for the current images. This number should be small. See 3.1 for more information.
- **HighlightRadius**: this is a smoothing parameter for calibration. See 3.1 for more information.
- **DarkLightThresh**: this is used in calibration mode. The threshold for detecting the rough position of the black ball. See 3.1 for more information.
- **FitCandidateNum**: this is used in calibration mode. This parameter tells the program how many points to choose for the least square error circle fitting. This number should be set according to the resolution of the image, and you should always *try* to set it bigger (unless error occurs). See 3.1 for more information.
- **HighLightThresh**: this is used in calibration mode. It is used when detecting the highlight position on the chrome ball. The highlight position is calculated as the *median* position of all points within the ball area that has a light intensity bigger than this value. See 3.1 for more information.
- **CenterParam**: this is used in calibration mode. When detecting the ball, the program will assume that the ball is placed within this share of the center of the image. If you set this value to x, then the program will assume the ball is placed within the rectangle that has vertex of $(\frac{1}{x}width, \frac{1}{x}height)$ and $(\frac{x-1}{x}width, \frac{x-1}{x}height)$. Set this number reasonably small may help better detect the ball. However, note that the ball should never appear outside this region. See 3.1 for more information.

3 Implemented Algorithms

3.1 Calibration

The calibration is done by ball detection. The core algorithm is WaterShed segmentation. The process of the calibration is:

1. Pick pixels within the area set by CenterParam that have a intensity lower than DarkLightThresh. This tells us the rough position of the ball and its shadow.
2. Find contours of the above pixels, choose the contour that has the biggest area.[2], then set this as the ball marker.
3. Use the rectangle set by CenterParam as the outlier marker.
4. Run watershed with the above two markers.
5. Find pixels within the segmented region that has a intensify higher (or with all its HighlightRadius*HighlightRadius neighbor pixels) than HighLightThresh, choose the median as the guess of the highlight point. The HighlightRadius is used to eliminate noise. Usually set it to 3 is fine.
6. Pick FitCandidateNum points on the edge of the segmented ball region that are most closer to the highlight point as the fitting candidate, then run least square error to fit a circle. We can imagine that the more circle points we choose to fit, the more accurate the result will be. However, if you set this number too big, points not on the circle may also be included, this will determine the fitting.

[2]In a recent test, I found that choosing the contour with biggest area might be problematic. But we can fix this problem with the help of other input images.

7. With the fitted circle, we will calculate the highlight again but this time only consider points inside the fitted circle. As we can see, by setting

8. Do the above for all the lights, then choose the one with the smallest fit error as the true ball position. After, we will go through all the images again, if the difference between the fit error of the current image and the best result is greater than FitErrorThresh, then we will use the best data to process the current image again. Usually set this value

After the calibration process, the program will output a file named "lights.txt". For distant PS, this file will store the incident direction of each light row by row. For near range PS, this file will store the position of each light in image space row by row.

3.2 Distant PS

The Distant PS is based on the classic model. We will first use *one* chrome ball to do the calibration, then the program will generate a file containing the incident direction of each light. Then based on this equation we will run a least square to resolve the normal vector for each pixel. Afterwards, we will use a surface from gradient technique to solve the depth for each pixel. The algorithm currently using is Frankot Chellappa Algorithm with the power of four terms. This can be found in [1]. Note that the current program implements a trick before using DFT. We extend the data matrix to four times of its size, that is flip up down, and left and right. By introducing *periodic* we can have better performance especially on the object edges.

Note that the current program doesn't handle shadows. Because the target objects are usually rock surfaces, not many shadows will appear on that.

When the program runs under this mode, it needs in total $2 * \#oflights$ images, of which $\#oflights$ are the calibration image for each light, then $\#oflights$ are the data image for the object.

To Do: We can use a non-linear least square optimization to optimize the lighting strength or even positions; for the surface from gradient algorithm, we can refer to [2] for a better solution.

3.3 Near Range PS

For the near range PS model, please refer to [3][4](and even more under the paper folder). This program implements the algorithm introduced in [3]. However, in the paper the author doesn't prove the convergence of the heuristic method. Note that you have to calibrate the 3D position of the lights under image space before can run the program under this mode. Currently the calibration method we implement is from [5], which is very unreliable.

To Do: Please refer to [4] for a better solution. This may be the first near range PS paper published in CVPR. In addition, we may also refer to their hardware design.

4 Compile the program

This program can be compiled for both Linux and Windows. I use eclipse to develop the program, and the whole eclipse project are available on a private repo in BitBucket. However, you need to configure the compilation environment first.

4.1 Third party dependencies

This program dependents on OpenCV[3] (I use 2.4.10, but later version should be fine too), Boost[4] (I use 1.58.0, but later version should be fine too), and Best-Fit[5]. These are all open source projects. And for both Windows and Linux OS, I use the GNU GCC compiler to build the whole project (NOT Visual Studio in Windows). For convenience, I have already compiled these dependencies into libraries (The BestFit which depends on Boost is compiled into static library, and OpenCV is compiled into shared library, which means

[3] http://opencv.org/
[4] http://www.boost.org/
[5] http://www.codeproject.com/Articles/692787/Best-fitting-line-circle-and-ellipse

that under Windows platform, you would also need the DLL compiled under Visual Studio. However, good news is, the official website offers pre compiled binaries thus you don't need to install the big VS on your PC), and in the Eclipse project, you can simply include these libraries as well as the headers to compile them.

4.2 Compile Under Windows

Since we are using GCC, the first thing we need to do is to install MinGW under Windows. Remember, don't forget to add the bin to your system PATH. Note that you don't need CygWin to compile them. (CygWin is also very BIG!)

For windows, the most common compiler is Visual Studio, and for both OpenCV and Boost, they only provide VS pre-built binaries, so we will need to compile these two projects our own.

There are two very nice tutorials, for OpenCV, you can refer to:

http://kevinhughes.ca/tutorials/opencv-install-on-windows-with-codeblocks-and-mingw/

for Boost, you can refer to:

https://theseekersquill.wordpress.com/2010/08/24/howto-boost-mingw/

In case the links may be broken, I saved them as PDF.

Next step is to compile the best-fit static library. You can new a c++ project for static library in Eclipse and import all the sources and header files downloaded from best fit website. Next, add the header dependencies on the project, like shown in figure 1.

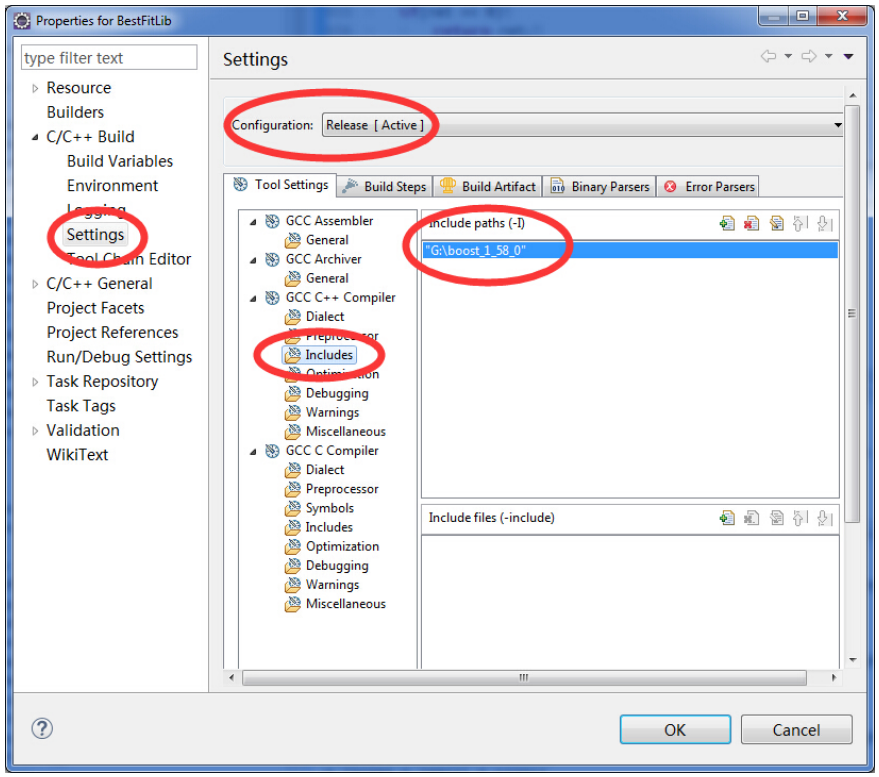

Figure 1: Include headers for project.

After you get all the libraries, you can then configure the 3DScanner project to include headers also like figure 1 and then configure the libraries like figure 2.

After the compilation, when you want to release the software to users, remember to copy all the DLLs for OpenCV and put them to the same folder as the executables. **DONNOT change the name of them.**

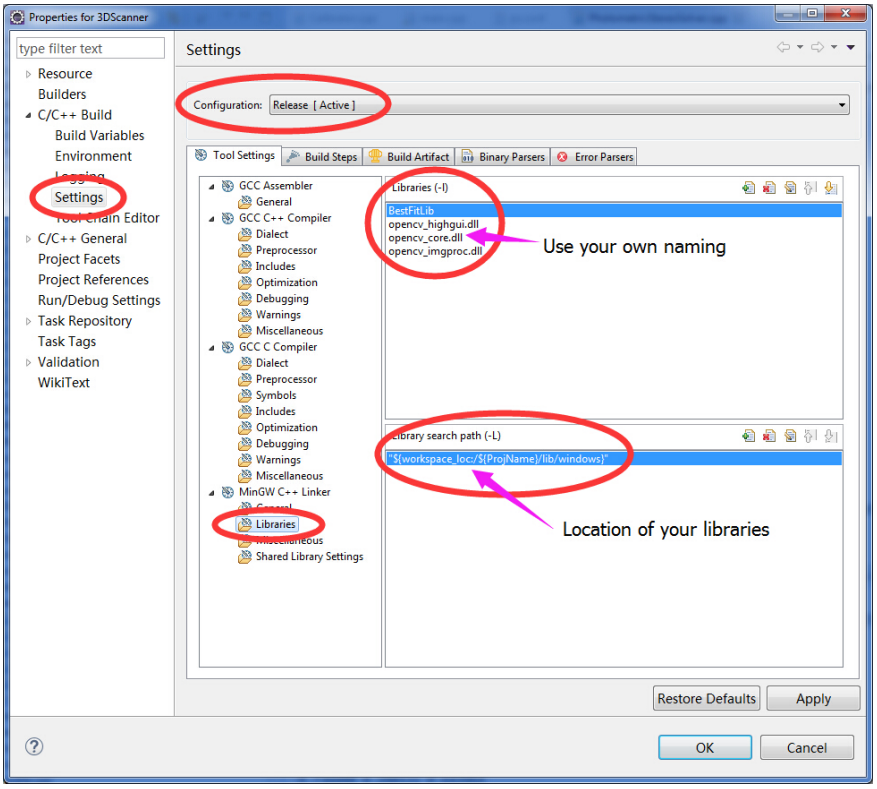

Figure 2: Include libraries for project.

4.3 Compile Under Linux

In Linux, the compilation process is much easier. Take Ubuntu for an example, you can install the Boost and OpenCV by just typing some commands. For OpenCV, refer to

https://help.ubuntu.com/community/OpenCV

For Boost, refer to

http://www.boost.org/doc/libs/1_57_0/doc/html/quickbook/install.html

After that, for BestFit library, you can simply run the **make** command from the original project. Then, you can do the same thing as showed in previous part to include headers and libraries to compile the program.

Reference

[1] Wei, Tiangong, and Reinhard Klette. A new algorithm for gradient field integration. CITR, The University of Auckland, New Zealand, 2001.
[2] Xie, Wuyuan, et al. "Surface-from-Gradients: An Approach Based on Discrete Geometry Processing." Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on. IEEE, 2014.
[3] Papadhimitri, Thoma, and Paolo Favaro. "Uncalibrated Near-Light Photometric Stereo."
[4] Xie, Wuyuan, Chengkai Dai, and Charlie CL Wang. "Photometric Stereo with Near Point Lighting: A Solution by Mesh Deformation." IEEE Transactions on Pattern Analysis and Machine Intelligence 25.10 (2003): 1239-1252.
[5] Ahmad, Jahanzeb, et al. "An improved photometric stereo through distance estimation and light vector optimization from diffused maxima region." Pattern Recognition Letters 50 (2014): 15-22.