Interactive 3D reconstration for insects

Yiran Zhong

A thesis submitted for the degree of
Master of Engineering (honor)
The Australian National University

May 2014

– 30 May 2014

Except where otherwise indicated, this thesis is my own original work.

Yiran Zhong
30 May 2014

– 30 May 2014

to my supervisor, Dr. Chuong Nguyen and Dr. Hongdong Li

– 30 May 2014

– 30 May 2014

Acknowledgments

I would like to express my gratitude to my supervisor in CSIRO, Dr. Chuong Nguyen, whose expertise, understanding, and patience, added considerably to my graduate experience. I appreciate his vast knowledge and skill in many areas (e.g., programming, camera calibration, 3D reconstruction, manufacturing and interaction with colleagues), and his assistance in writing reports (i.e., research proposals, progress report and this thesis), which have on occasion made me "GREEN" with envy.
A very special thanks goes out to Dr. Hongdong Li, without whose motivation and encouragement I would not have considered a graduate career in 3D reconstruction research. Dr. Hongdong is the one professor who truly made a difference in my life. It was under his tutelage that I developed a focus and became interested in computer vision. He provided me with direction, technical support and became more of a mentor and friend, than a professor. It was though his, persistence, understanding and kindness that I'm about to complete my graduate degree and is encouraged to apply for phd training. I doubt that I will ever be able to convey my appreciation fully, but I owe him my eternal gratitude.
I would also like to thank my family for the support they provided me through my entire life, without whose love, encouragement, I would not have finished this thesis.

– 30 May 2014

Abstract

3D modeling of insects has important applications that require rich information sharing and visual recognition, such as taxonomy and quarantine control. The quality of 3D insect models is crucial to provide correct taxonomic information or to lead to correct decision within sort time constraint of quarantine control and inspection. To provide high-quality 3D insect models, a number of factors need to be considered, including sample preparation, image quality, scanning settings and 3D reconstruction algorithm. Currently there are several non-optimal steps that lead to poor reconstruction result. This thesis aims to address these steps. We improved the 3D insect reconstruction system described in [Nguyen et al., 2014].

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Thesis Statement

Interactive 3D reconstruction can improve current reconstructed model by capturing extra images at selected angles. Also generating camera poses based hardware information helps improve the quality of the reconstructed model as compared with general Bundle Adjustment.

1.2 Problem Statement

Nowadays, people are still using images of small species to identify them. However, for many small species, several images could not provide enough information to classify them or in other words if we have more images from different angles, we can accelerate this classification process. Now, imaging we are using an accurate 3D model instead of several images captured from some fixed angles to do the classification. And there will be no doubt that it will speed up the classification process dramatically.

Currently there are several method that can generate millimeter-scale objects. One of them is called Micro Computed Tomography (Micro CT) [Metscher]. It can create micron-accurate volumetric models of millimeter-scale specimens and their internal structure. However, the main drawback of this method is that it could not capture important information about the surface of the specimen: its natural color. Moreover,they are cost in hundred-thousand dollar range and most of them are not portable.

Passive 3D reconstruction allows user to maintain the natural color while creating the 3D model [Brusco et al. [2005]]. But for such small objects, it still a big challenge for us, because the main drawback of Passive 3D reconstruction is its accuracy.

[Nguyen et al. [2014]] created a method that can generate true color 3D model for insects with a certain level of accuracy. However, there are still remaining some non-optimal steps that lead to poor reconstruction result. This project aims to address these steps.

For example, as shown in Figure 1.1, A typical set of camera poses can not resolve the occlusion created by the wings of this insect, leading to inaccurate reconstruction between its wings. Additional images take from camera poses looking along the insect body and wing surfaces dramatically improves reconstruction accuracy. Based on this, an interactive 3D scanning scheme is needed for accurate 3D reconstruction.

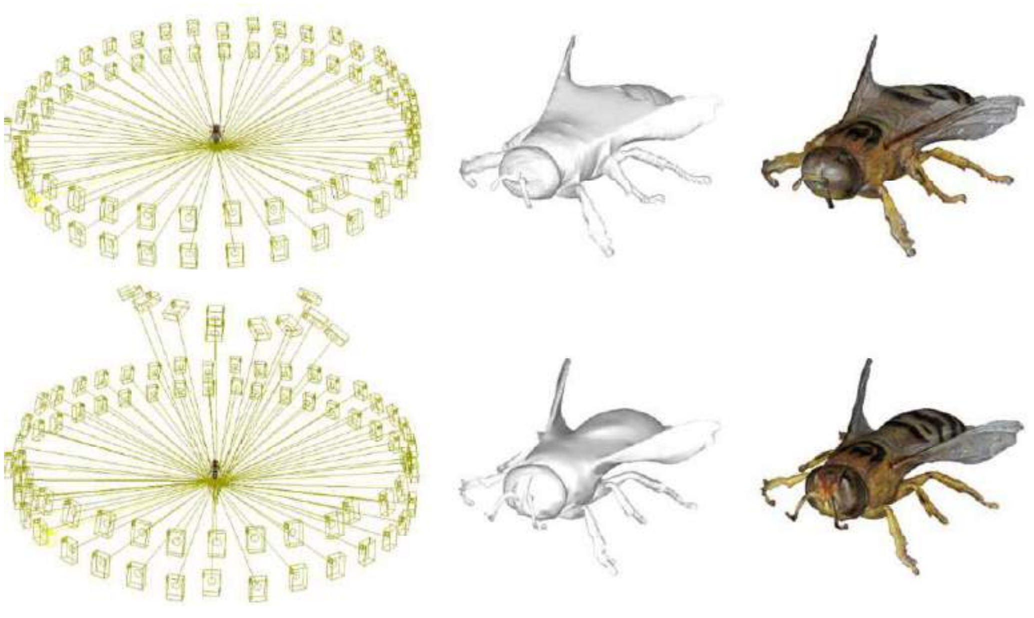


Figure 1.1: Default scanning may not lead to sub-optimal 3D shape[Nguyen et al. [2014]]

1.3 Thesis Outline

Chapter 2 describes the motivation of this project and some of the related work which has been done before this project. Then I present the system design and implementation in Chapter 3. Also in Chapter 4, the software and hardware platform has been described. Finally, the results of this project is presented in Chapter 5 and the conclusion of this project is in Chapter 6.

Chapter 2

Background and Related Work

2.1 Motivation

3D modeling of insects has important applications that require rich information sharing and visual recognition, such as taxonomy and quarantine control. The quality of 3D insect models is crucial to provide correct taxonomic information or to lead to correct decision within sort time constraint of quarantine control and inspection. To provide high-quality 3D insect models, a number of factors need to be considered, including sample preparation, image quality, scanning settings and 3D reconstruction algorithm. Currently there are several non-optimal steps that lead to poor reconstruction result. This project aims to address these steps.

2.2 Related work

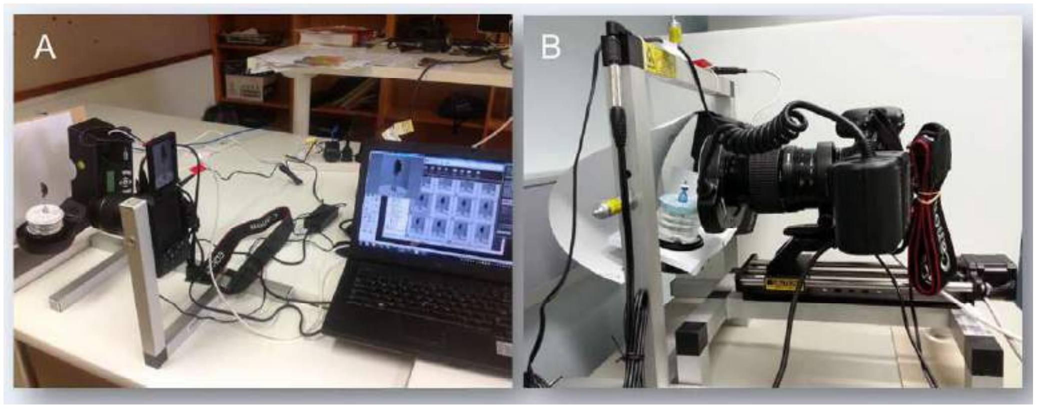
There are several existing methods that can create a 3D model. Micro Computed Tomography (Micro CT) is currently a key method, able to create micron-accurate volumetric models of millimeter-scale objects and their internal structure. However, like recent 3D reconstructions from scanning electron microscope (SEM) micrographs [Akkari et al.], Micro CT is unable to capture important information about the surface of the object: its natural colour. Exposure and reconstruction times can be long (tens of hours) and, as an X-ray imaging method, Micro CT generally demands special safety equipment. Current systems cost in the hundred-thousand dollar range and, while more compact desktop models are available, these are still not especially portable.

The inability of X-ray based methods for insect digitization to capture colour led us to consider image-based 3D reconstruction techniques. These methods have been successfully applied to the reconstruction of 3D cityscapes and other (generally fairly simple) objects. Some small biological specimens have been digitized but the methods used do not specifically cater for the complex structures and challenging surface optical properties of insects. Human-in-the-loop approaches have been proposed for insect modeling as have methods (limited to simple insect geometries) for inferring 3D insect shape from a single 2D image. Experiments with laser scanning systems like have suggested that this approach has difficulties with the fine structures and the small scale of many insects, as well as reflective, transparent or iridescent surfaces.

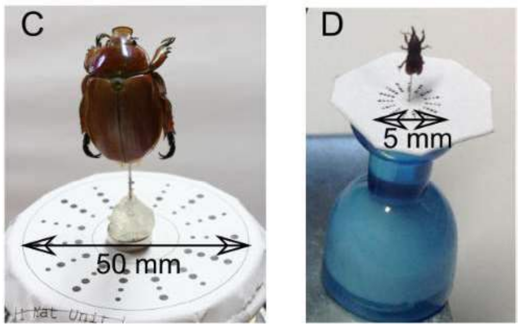
One way to avoid these difficulties is to steer clear of 3D reconstruction altogether and simply present 2D images obtained from different viewing angles. While this method of 3D visualization is popular for museum collections it does not provide the quantitative information (e.g., 3D morphology) needed to analyze and compare insect specimens. Furthermore large amounts of data are involved: many high-resolution images are needed to give a convincing illusion of looking at an actual 3D object. This makes smooth, realistic interaction difficult and precludes straightforward email exchange or embedding of the object data.

The method provided in [Nguyen et al. [2014]] can successfully create a true color 3D model for small insects. However, it still has some drawbacks:

- The turntable has no interaction with the computer.
- The camera needs to estimate the angle from the pattern on the base, which leads to large estimation error.
- Limitation on capturing angles. As shown in Figure 2.1. User couldn't use the images captured from the bottom view because the camera couldn't capture the calibration pattern.



(a) Scanning device



(b) Calibration pattern

Figure 2.1: Scanning device and calibration pattern [Nguyen et al. [2014]]

2.3 Summary

In this chapter, the motivation of this project has been presented as well as some of the related work. In summary, there is a lack of existing systems that could capture the 3D structure and surface optical properties of small, intricate insect specimens at sufficient resolution. And the current existing one still has some drawbacks that need to improve.

Chapter 3

Design and Implementation

3.1 System Introduction

This whole system can be divided by 6 modules: Motor control module, camera control module, 3D model display module, camera calibration module, image segmentation module and visual hull 3D reconstruction module.

All these 6 parts can be controlled through a GUI interface as shown in figure 3.6, figure 3.8, figure 3.18and figure 3.24. The system architecture is shown in figure 3.1, and the system flow chart is shown in figure 3.2.

First, we use motor and camera control module to capture a set of calibration target images. Then we perform two-axis turntable camera calibration using the captured images. Therefore we have the camera parameters that allow us to generate camera poses for any rotation angles. After that, we replace the calibration target by a insect specimen, and capture its images by using the default scanning scheme. Then using the image segmentation module to get the specimen's silhouette of each angle, and use them as an input for visual hull 3D reconstruction and it will produce a 3D model in ".ply" file that we can visualize in the 3D model displayer. If we are not happy with the result, we then move on to the interactive scanning. We can manually (or automatically) rotate the camera to a desired position and take another image, add its silhouette to the input of 3D reconstruction and see how it goes. We can do the interactive scanning again and again until we get desired result.

3.2 Motor Control

3.2.1 Stepper Motor

There are two motors in this turntable, one for x-axis turning, and the other for y-axis turning.The X-axis motor has a gear ratio 1:5.2 while the other one is 100:1. That's because the x-axis motor only need to turn the insects but the y-axis one need to rotate the whole x-axis rig therefore more torque will be needed.

The stepper motors which has been used in this project are *NEMA-17 Bipolar 5.18:1 Planetary Gearbox Stepper* for x-axis, and *NEMA-17 Bipolar 99.51:1 Planetary Gearbox Stepper* for y-axis.

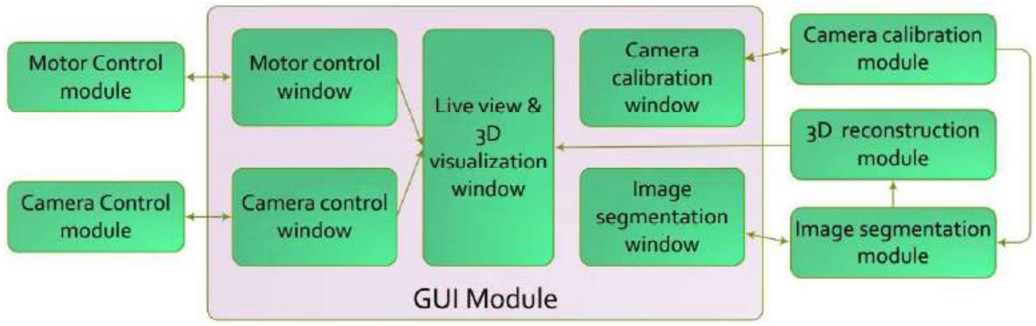

Motor Control module
Camera Control module
Motor control window
Camera control window
Live view & 3D visualization window
Camera calibration window
Image segmentation window
GUI Module
Camera calibration module
3D reconstruction module
Image segmentation module

Figure 3.1: System Architecture

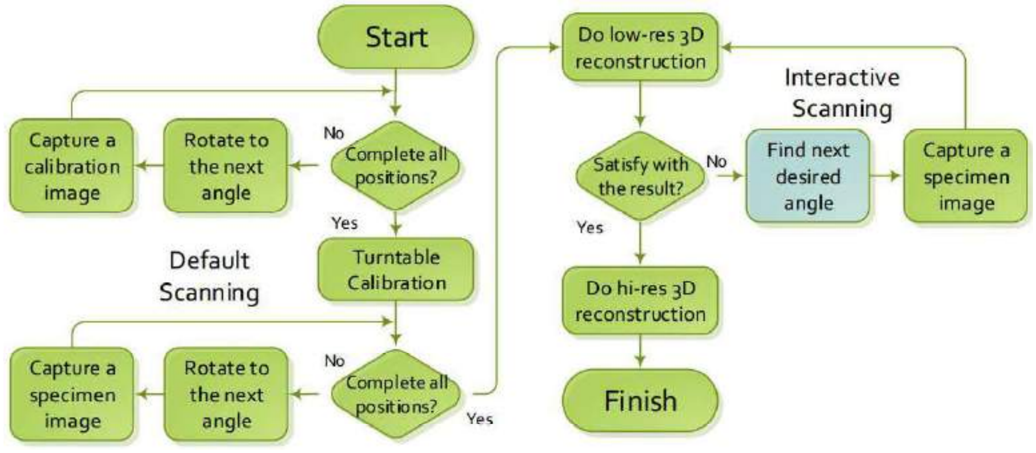


Figure 3.2: System Flow Chart

Using a stepper motor with a gearbox can be a good solution in applications that need very low rotation speeds and/or lots of torque. Selecting a gearbox to attach to the stepper will result in increasing the output torque and decreasing the speed. Simply, the Gearbox Output Speed = Motor Speed / Gearbox ratio, Gearbox Step angle = Motor Step Angle / Gearbox ratio.

In this project, it is important to know the exact gear ratio of a gearbox (we need the output shaft to make a several complete rotations and stop in the same position it started). Most motor datasheets will only list an approximate ratio, like "10:1" or "27:1". However, there are ways to figure out the exact ratio.

In a planetary gearbox where the sun gear is used as an input and the rotation of the spider linking the planet gears is the output, the reduction ratio can be expressed by the following equation:

$$Gearboxratio = (T_r + T_s)/T_s \tag{3.1}$$

(a) X-axis Stepper Motor

(b) Y-axis Stepper Motor

Figure 3.3: Stepper Motor [Phidget documentation]

Where T_r is the number of teeth on the ring gear, and T_s is the number of teeth on the sun gear.

The following table 3.4 lists exact ratios for gearboxes commonly sold at Phidgets Inc. :

3.2.2 Stepper Motor Controller

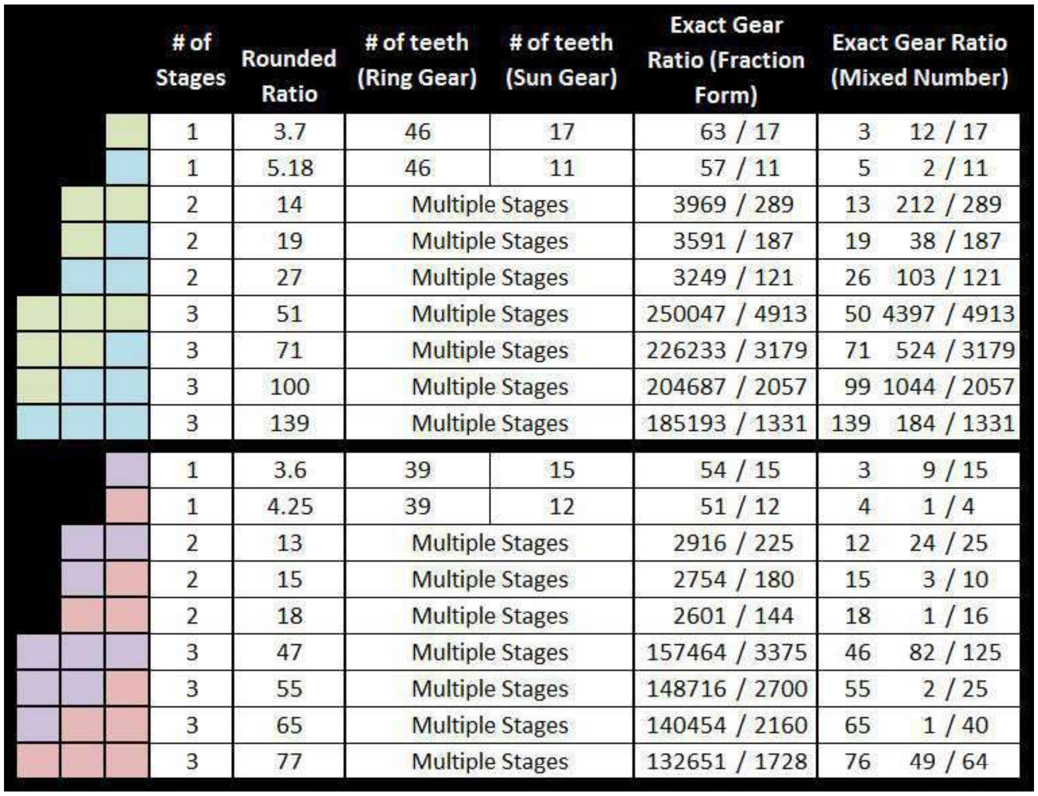
The stepper motor controller which has been used in this project is *1063_1 - Phidget-Stepper Bipolar 1-Motor* which allows us to control the position, velocity, and acceleration of one bipolar stepper motor.

The *1063* can be used to control 4, 6, or 8 wire bipolar stepper motors. When the steppers are first engaged from software, the stepper motor likely will not be at the same state as the default output state of the controller. This will cause the stepper to ’snap’ to the position asserted by the controller - potentially moving by 2 full steps.

At low speeds, less than 1024 1/16th steps per second, the 1063 uses a microstepping scheme to allow precise control of the current to each coil, and therefore the position of the stepper. It’s important to note that many steppers are not designed to be microstepped precisely, and will not accurately produce the expected angle. At higher speeds, the *1063* switches to a full stepping mode. The *1063* will automatically switch back to microstepping when approaching the target position.

To avoid the use of floating point in the APIs, the position, velocity and acceleration parameters are expressed in microsteps (1/16th steps), instead of full steps. So when I wants to move the motor by 1 full step, I change the position by 16.

Because stepper motors do not have the inherent ability to sense their actual shaft position, they are considered open loop systems. This means that the value contained in the current position property is merely a count of the number of steps that have occurred towards the target value; it can not be relied upon as a measure of the actual

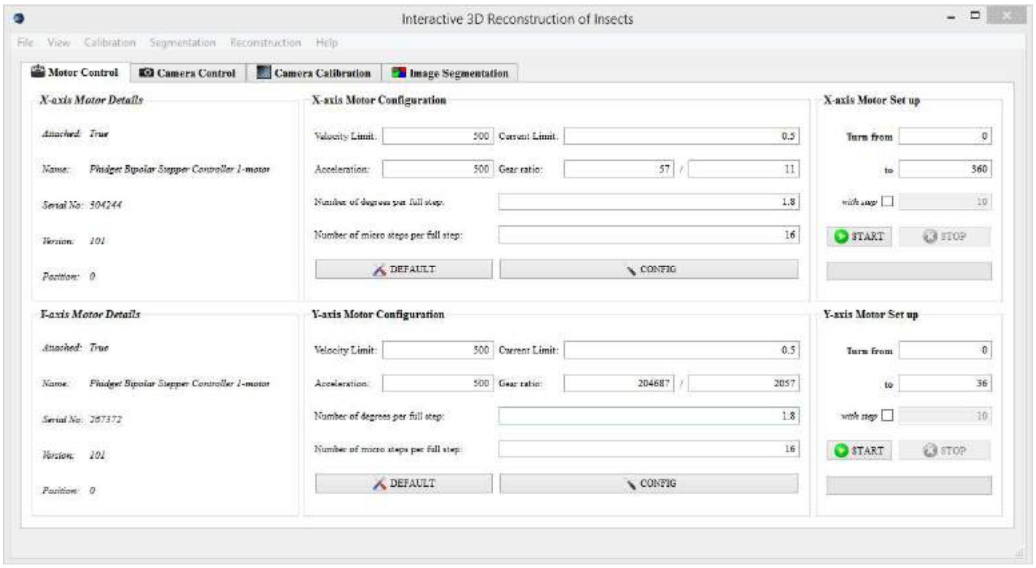


			# of Stages	Rounded Ratio	# of teeth (Ring Gear)	# of teeth (Sun Gear)	Exact Gear Ratio (Fraction Form)	Exact Gear Ratio (Mixed Number)
		🟩	1	3.7	46	17	63 / 17	3 12 / 17
		🟦	1	5.18	46	11	57 / 11	5 2 / 11
	🟩	🟩	2	14	Multiple Stages		3969 / 289	13 212 / 289
	🟩	🟦	2	19	Multiple Stages		3591 / 187	19 38 / 187
	🟦	🟦	2	27	Multiple Stages		3249 / 121	26 103 / 121
🟩	🟩	🟩	3	51	Multiple Stages		250047 / 4913	50 4397 / 4913
🟩	🟩	🟦	3	71	Multiple Stages		226233 / 3179	71 524 / 3179
🟩	🟦	🟦	3	100	Multiple Stages		204687 / 2057	99 1044 / 2057
🟦	🟦	🟦	3	139	Multiple Stages		185193 / 1331	139 184 / 1331
		🟪	1	3.6	39	15	54 / 15	3 9 / 15
		🟥	1	4.25	39	12	51 / 12	4 1 / 4
	🟪	🟪	2	13	Multiple Stages		2916 / 225	12 24 / 25
	🟪	🟥	2	15	Multiple Stages		2754 / 180	15 3 / 10
	🟥	🟥	2	18	Multiple Stages		2601 / 144	18 1 / 16
🟪	🟪	🟪	3	47	Multiple Stages		157464 / 3375	46 82 / 125
🟪	🟪	🟥	3	55	Multiple Stages		148716 / 2700	55 2 / 25
🟪	🟥	🟥	3	65	Multiple Stages		140454 / 2160	65 1 / 40
🟥	🟥	🟥	3	77	Multiple Stages		132651 / 1728	76 49 / 64

Figure 3.4: Stepper Motor Gearbox Table [Phidget documentation]

shaft angle, as external forces may also be affecting the motor. In our experiments, the open loop controller is precise enough for our purpose.

Figure 3.5: Stepper Motor Controller: *1063_1 - PhidgetStepper Bipolar 1-Motor*


Interactive 3D Reconstruction of Insects
File View Calibration Segmentation Reconstruction Help
Motor Control
Camera Control
Camera Calibration
Image Segmentation
DEFAULT
CONFIG
START
STOP

X-axis Motor Details

Attached: True
Name: Phidget Bipolar Stepper Controller 1-motor
Serial No: 504244
Version: 101
Position: 0

X-axis Motor Configuration

Velocity Limit: 500 Current Limit: 0.5
Acceleration: 500 Gear ratio: 57 / 11
Number of degrees per full step: 1.8
Number of micro steps per full step: 16

X-axis Motor Set up

Turn from 0
to 360
with step ☐ 10

Y-axis Motor Details

Attached: True
Name: Phidget Bipolar Stepper Controller 1-motor
Serial No: 267372
Version: 101
Position: 0

Y-axis Motor Configuration

Velocity Limit: 500 Current Limit: 0.5
Acceleration: 500 Gear ratio: 204687 / 2057
Number of degrees per full step: 1.8
Number of micro steps per full step: 16

Y-axis Motor Set up

Turn from 0
to 36
with step ☐ 10

Figure 3.6: Motor Control GUI interface

3.2.3 GUI Interface

The GUI interface of the motor control module is shown in figure 3.6 The interface should be able to accomplish five tasks which are list below:

- Connect two stepper motors with order: We have to make sure x/y-axis motor can be controlled by x/y-axis control panel.
- Configure the stepper motors: Several parameters need to be set before running the motor such as gear ratio, speed and position.
- Terminate the stepper motors when needed: For some situations we need to stop the motor in the middle of the operation.
- Add a demo mode: Automatically pause every configured angle(e.g. 10 degree) and resume running until rotate a whole cycle.
- Need to be cross-platformed: Sometime we need to run this application on Linux system.

For each stepper motor, there are 3 panels: Motor details, Motor configuration and Motor set up. The Motor details panel shows details of the motor including serial number and position. The motor configuration panel allows users to configure some basic parameters of the motor such as the velocity limit and gearbox ratio. Also there is a default button that can set default setting parameters. In Motor set up panel, users can set the angle they want to rotate. And there is a with step check box, if it checked, the motor will inter demo mode. At any time, if users want to stop the operation, they can click the stop button to terminate the motor.

In this application, controlling motors will occupy lots of CPU time and therefore blocking the main thread-the GUI thread. So in the vary early version of this application, the GUI interface will loose response when the stepper motor is running. In order to avoid this problem, I move all the motor control part into threads and let the main thread only to deal with the GUI display job. In this case, each motor has its own thread, which makes easier for further upgrades like adding more motors to control.

The reason why it cast so much CPU time is that I used "while" loop to detect the state of motor but didn't use "usleep()" function to release the CPU. In order to fix that, instead of using "usleep()" function, I added a timer to detect it. The reason is:

Blocking on a timer and blocking on sleep should be exactly the same in respect of CPU cycles and power consumption. Except that on at least one major operating system (Windows), blocking on a timer has a much better precision and reliability than sleeping. Sleeping will give away the remainder of the time slice and not schedule the thread again before some time, rounded up to the next 16 or so milliseconds. Blocking on a timer puts the thread on the ready list the moment the timer expires and dynamically boosts the thread priority, scheduling it before other and possibly interrupting running threads with the same (original) priority. That's a difference between "yeah, some time, eventually" and "as soon as we can handle it".

By replacing "while" loop with a timer, it seems even if the motor control runs in the main thread will no longer block the GUI interface and the multi-thread design looks unnecessary. It's true if this application only to use for motor control. However, this application needs to integrate many other functions such as rendering 3D model and camera calibration. These functions also need lots of CPU time. So if there is no multi-thread design, the GUI interface will takes more time to response the user event therefore we will feel unsmooth during manipulation.

If we want to execute a function (including timer) in a new thread, this function (timer) must be started in the virtual function run(). Because of that, I use flags to control the motors: after initialize the new thread, the thread will keep polling the value of the flags therefore control the motors. These flags and other states variables are global variables, so they can be read or modified by other threads. This is how the threads communicate with each other.

This application need to control two separate motors with an order. For example, the x-axis motor must be controlled in x-axis control panel. However, when computers connect to these motors, the order is completely random-it depends on the connection time. The first connected one will displayed in x-axis control panel. In order to keep them in an order, I compare their serial numbers after initialize. If the order is right, then move to the next stage. If not, stop the threads and restart them in an opposite order. That's because the x-axis control panel always controls motor in thread1 while the y-axis one controls motor in thread2.

3.2.4 Flow Chart

The flow chart of motor control is shown in figure 3.7.

When the application start, user can click the "connect to stepper" button to initialize the motors. Then configure them in the configuration panel. This is for telling the controller what kind of stepper motor we are using. Then in set up panel, user can decide the rotation angle they want to rotate or start a default scanning by click the "with step" check box. In default scanning model, it will rotate one step angle and wait 2.5 seconds for taking a image. If the camera is connected, it will automatically take a picture, otherwise it will do nothing. User is allowed to terminate the process at any time.

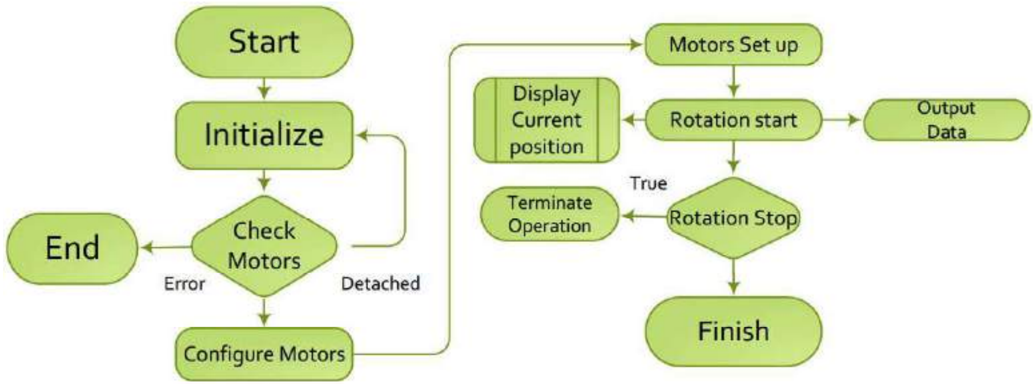


Figure 3.7: Motor Control Flow Chart

3.3 Camera Control

We use Canon EOS Digital Camera to capture images in this system. On windows platform, Canon Inc. has already provided a EOS Digital Camera Software Development Kit – *EDSDK*, while on linux platform, we can use *gPhoto2* library to control the camera.

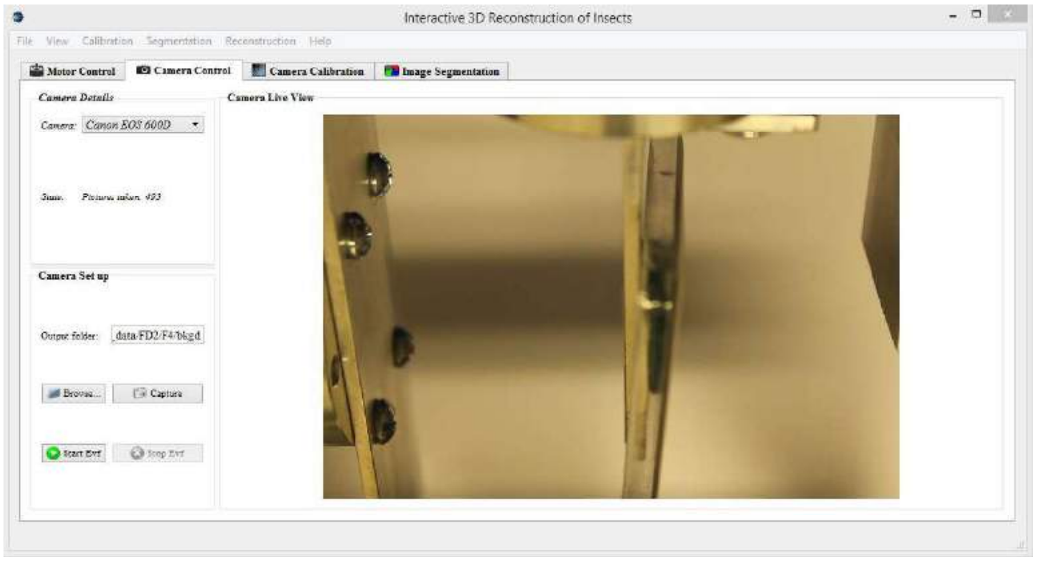
EDSDK EDSDK stands for EOS Digital Camera Software Development Kit. EDSDK provides the functions required to control cameras connected to a host PC, digital images created in digital cameras, and images downloaded to the PC.

EDSDK provides an interface for accessing image data shot using a Canon EOS digital camera. Using EDSDK allows users to implement the following types of representative functions in software.

- Allows transfer of images in a camera to storage media on a host PC.
- Allows RAW images to be processed and saved in JPEG format.
- Allows remotely connected cameras and the image being shot to be controlled from a host PC.

gPhoto2 gPhoto is a program and library framework that lets users download pictures from their digital cameras. The libgphoto2 library gives you access to hundreds of models of digital cameras on Linux, FreeBSD, NetBSD and other Unix-like operating systems. That means our application can control more models of commercial camera than just Canon under Linux platform.

3.3.1 GUI interface



Camera: Canon EOS 600D

State: Pictures taken: 493

Output folder: data/FD2/F4/bkgd

Figure 3.8: Camera control GUI interface

As shown in Figure 3.8, the control panel can be divided by 3 parts. One part for showing the camera details, one part for setting and controlling the camera (i.e. set up the output folder, start EVF) and a window to show the camera live view.

Actually the only necessary function for camera control is capture a image and sent it to the host computer. I added the live view function because it makes users more convenient when they manually adjust the focus of the camera.

3.3.2 Flow Chart

First, the user need to connect the camera. If the camera(s) is(are) available, this program will initialize all the connected cameras. Then select one on the list of cameras and set a valid folder as the output folder. If the path is not valid, the buttons of capture image and start EVF will not be enabled. After that, the user can click the capture button to capture an image or click the start EVF button to start a live view.

It's ok to capture an image during the live view, but the flash light will no enabled.

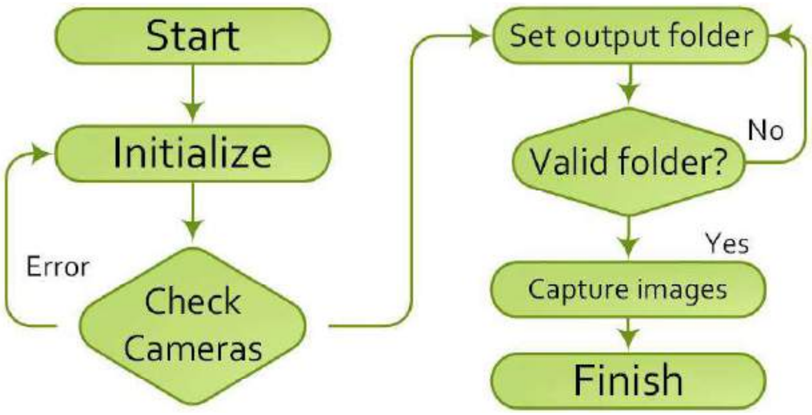


Figure 3.9: Camera control GUI interface

3.4 3D model Displayer

3.4.1 PLY File Format

The 3D model I created is in PLY file format, which is known as the Polygon File Format or the Stanford Triangle Format. The format was principally designed to store three-dimensional data from 3D scanners. It supports a relatively simple description of a single object as a list of nominally flat polygons. A variety of properties can be stored including: color and transparency, surface normals, texture coordinates and data confidence values. The format permits one to have different properties for the front and back of a polygon.

FLY Files are organized as a header, that specifies the elements of a mesh and their types, followed by the list of elements itself, usually vertices and faces - potentially other entities such as edges, samples of range maps, and triangle strips can be encountered.

An example header is shown in Figure 3.10.

The header always starts with a "magic number", the first line identifies the file as a PLY file. The second line indicates which variation of the PLY format this is (i.e. ASCII or binary). Comments may be placed in the header by using the word comment at the start of the line. Everything from there until the end of the line should then be ignored (i.e. line 3 & 4). The ’element’ keyword introduces a description of how some particular data element is stored and how many of them there are. Hence, in a file where there are 126626 vertices, each represented as a floating point (X,Y,Z) triple. Other ’property’ lines might indicate that colours or other data items are stored at each vertex and indicate the data type of that information (i.e. line 5-12). Line 13 indicates that this 3D model has 253248 polygonal faces. The word ’list’ indicates that the data is a list of values âĂŞ- the first of which is the number of entries in the list (represented as a ’uchar’ in this case) and each list entry is (in this case) represented as an ’int’. At the end of the header, there must always be the line 16.

```
ply
format binary_little_endian 1.0
comment VCGLIB generated
comment TextureFile Cricket2.jpg
element vertex 126626
property float x
property float y
property float z
property uchar red
property uchar green
property uchar blue
property uchar alpha
element face 253248
property list uchar int vertex_indices
property list uchar float texcoord
end_header
```

Figure 3.10: An example PLY Header

3.4.2 Libraries

OpenGL OpenGL (Open Graphics Library) is a cross-language, multi-platform application programming interface (API) for rendering 2D and 3D vector graphics. The API is typically used to interact with a graphics processing unit (GPU), to achieve hardware-accelerated rendering.

OpenGL was developed by Silicon Graphics Inc. (SGI) from 1991 and released in January 1992 and is widely used in CAD, virtual reality, scientific visualization, information visualization, flight simulation, and video games. OpenGL is managed by the non-profit technology consortium Khronos Group.

In this application, it has been used for 3D rendering, and as a basic component of VCG Library.

GLEW The OpenGL Extension Wrangler Library (GLEW) is a cross-platform open-source C/C++ extension loading library. GLEW provides efficient run-time mechanisms for determining which OpenGL extensions are supported on the target platform. OpenGL core and extension functionality is exposed in a single header file.

VCG The Visualization and Computer Graphics Library (VCG for short) is a open source portable C++ templated library for manipulation, processing and displaying with OpenGL of triangle and tetrahedral meshes.

The VCG library is tailored to mostly manage triangular meshes: The library is fairly large and offers many state of the art functionalities for processing meshes, like:

- high quality quadric-error edge-collapse based simplfication,
- efficient spatial query structures (uniform grids, hashed grids, kdtree, ...) ,
- advanced smoothing and fairing algorithms,
- computation of curvature,

- optimization of texture coordinates,
- Hausdorff distance computation,
- Geodesic paths,
- mesh repairing capabilities,
- isosurface extraction and advancing front meshing algorithms,
- Poisson Disk sampling and other tools to sample point distributions over meshes,
- subdivision surfaces.

The most famous software developed based on VCG library is *MeshLab*.

3.4.3 GUI interface

I use another *Main Window* to display the 3D model. And it has the ability to display a 3D model with texture.

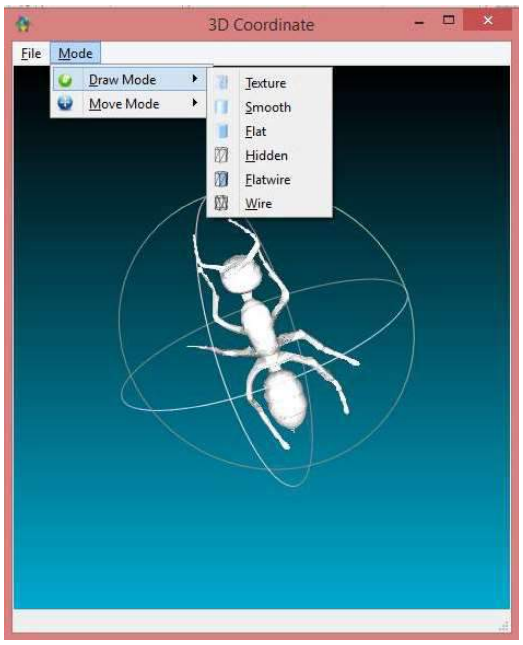


Figure 3.11: 3D Model Displayer

There are three main steps to apply the texture image to the 3D model.

1. Enable the *EnableWedgeTexCoord()*, therefore the loaded texture can be applied to the 3D model,
2. Load the texture image,
3. Scale the texture image up to the next highest power of 2.

The reason why it has to be a power of 2 is that: By ensuring the texture dimensions are a power of two, the graphics pipeline can take advantage of optimizations related to efficiencies in working with powers of two. For example, it can be (and absolutely was several years back before we had dedicated GPUs and extremely clever optimizing compilers) faster to divide and multiply by powers of two. Working in powers of two also simplified operations within the pipeline, such as computation and usage of mipmaps (a number that is a power of two will always divide evenly in half, which means you don't have to deal with scenarios where you must round your mipmap dimensions up or down).

3.5 Camera Calibration

Camera calibration is a necessary step in 3D computer vision in order to extract metric information from 2D images. Many calibration method has been proposed in very recent years. However, we can not say that one method is the best because it is depends on the situation we faced.

Generically, Self-calibration [Maybank and Faugeras [1992]] cannot usually achieve an accuracy comparable with that of pre-calibration because self-calibration needs to estimate a large number of parameters, resulting in a much harder mathematical problem.

In this project, we are using Zhang's camera calibration method [Zhang [2000]], which has been integrated in OpenCV library to get the camera intrinsic matrix.

A camera is modeled by the usual pinhole as shown in Figure 3.12, A 2D point is denoted by $\mathbf{m} = [u, v]^T$. A 3D point is denoted by $\mathbf{M} = [X, Y, Z]^T$. There are several advantages if we use homogeneous coordinates, which defined as $[x, y, z, w]$ for 3D coordinate:

- If $w = 1$, then the vector $[x, y, z, 1]$ is a position in space,
- If $w = 0$, then the vector $[x, y, z, 0]$ is a direction.

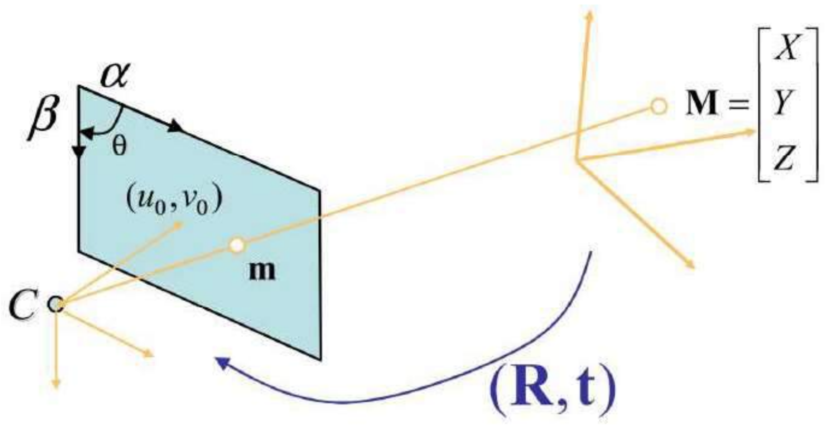
So we use $\tilde{x}$ to denote the augmented vector by adding 1 as the last element: $\tilde{\mathbf{m}} = [u, v, 1]^T$, and $\tilde{\mathbf{M}} = [X, Y, Z, 1]^T$. The image of a 3D point $\mathbf{M}$, denoted by m is formed by an optical ray from $\mathbf{M}$ passing through the optical center C and intersecting the image plane. The three points $\mathbf{M}$, $\mathbf{m}$, and C are collinear. In Figure 3.12, for illustration purpose, the image plane is positioned between the scene point and the optical center, which is mathematically equivalent to the physical setup under which the image plane is in the other side with respect to the optical center. The relationship between the 3D point $\mathbf{M}$ and its image projection $\mathbf{m}$ is given by

$$s\tilde{\mathbf{m}} = \mathbf{A}[\mathbf{R}\ \mathbf{t}]\tilde{\mathbf{M}} = \mathbf{P}\tilde{\mathbf{M}} \tag{3.2}$$

$$\mathbf{A} = \begin{bmatrix} f_x & \gamma & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.3}$$

$$\mathbf{P} = \mathbf{A}[\mathbf{R}\ \mathbf{t}] \tag{3.4}$$

where s is an arbitrary scale factor, $(\mathbf{R}, \mathbf{t})$, called the extrinsic parameters, is the rotation and translation which relates the world coordinate system to the camera coordinate system, and $\mathbf{A}$ is called the camera intrinsic matrix, with (u_0, v_0) the coordinates of the principal point, f_x and f_y the scale factors in image u and v axes, and γ the parameter describing the skew of the two image axes. The 3×4 matrix $\mathbf{P}$ is called the camera projection matrix, which mixes both intrinsic and extrinsic parameters. In Figure 3.12, the angle between the two image axes is denoted by θ, and we have $\gamma = f_x \cot\theta$. If the pixels are rectangular, then $\theta = 90°$ and $\gamma = 0$.



α
β
θ
(u_0, v_0)
m
C
$\mathbf{M} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$
$(\mathbf{R}, \mathbf{t})$

Figure 3.12: Pinhole camera model [Zhang [2000]]

The task of camera calibration is to determine the parameters of the transformation between an object in 3D space and the 2D image observed by the camera from images:

- Extrinsic parameters: orientation (rotation) and location (translation) of the camera, i.e. $(\mathbf{R}, \mathbf{t})$;
- Intrinsic parameters: characteristics of the camera, i.e., camera matrix $\mathbf{A}$.

The rotation matrix, although consisting of 9 elements, only has 3 degrees of freedom. The translation vector t obviously has 3 parameters. Therefore, there are 6 extrinsic parameters and 5 intrinsic parameters, leading to in total 11 parameters.

A popular technique for solving these parameters consists of four steps [Faugeras [1993]]:

1. Detect the corners of the checker pattern in each image;
2. Estimate the camera projection matrix $\mathbf{P}$ using linear least squares;
3. Recover intrinsic and extrinsic parameters $\mathbf{A}$, $\mathbf{R}$ and $\mathbf{t}$ from $\mathbf{P}$;
4. Refine $\mathbf{A}$, $\mathbf{R}$ and $\mathbf{t}$ through a nonlinear optimization.

3.5.1 Corner extraction

The function provided by Opencv –*findchessboardcorners()* can perform corner extraction. The function attempts to determine whether the input image is a view of the chessboard pattern and locate the internal chessboard corners. The function returns a non-zero value if all of the corners are found and they are placed in a certain order (row by row, left to right in every row). Otherwise, if the function fails to find all the corners or reorder them, it returns 0. Note that the detected coordinates are approximate, in order to determine their positions more accurately, we use the function calls *cornerSubPix()*.

However, the C++ Opencv always has some problems with high resolution image and the chessboard occupies larger proportion of the image. For example, when I tried to preform *findchessboardcorners()* on the image (1728 × 2592 pixels) with 100mm fixed lens, it can hardly extract corners. But when I use 50mm fixed lens, there will be no problem. Strangely, python Opencv didn't have such problem. Moreover, for images that it couldn't detect any corners, it took much long time for searching corners. One solution for these problem is to resize the image to a smaller one and scaler it back along with the detected corners then apply *cornerSubPix()* to refine the results.

Note that it is not a good idea to use general corner detection such as Harris corner detector, to detect the corners in the check pattern image, the result is usually not good because the detector corners have poor accuracy (about one pixel). A better solution is to leverage the known pattern structure by first estimating a line for each side of the square and then computing the corners by intersecting the fitted lines, which is exactly how *findchessboardcorners()* works.

Here is an example image (Figure 3.13) of corner extraction:

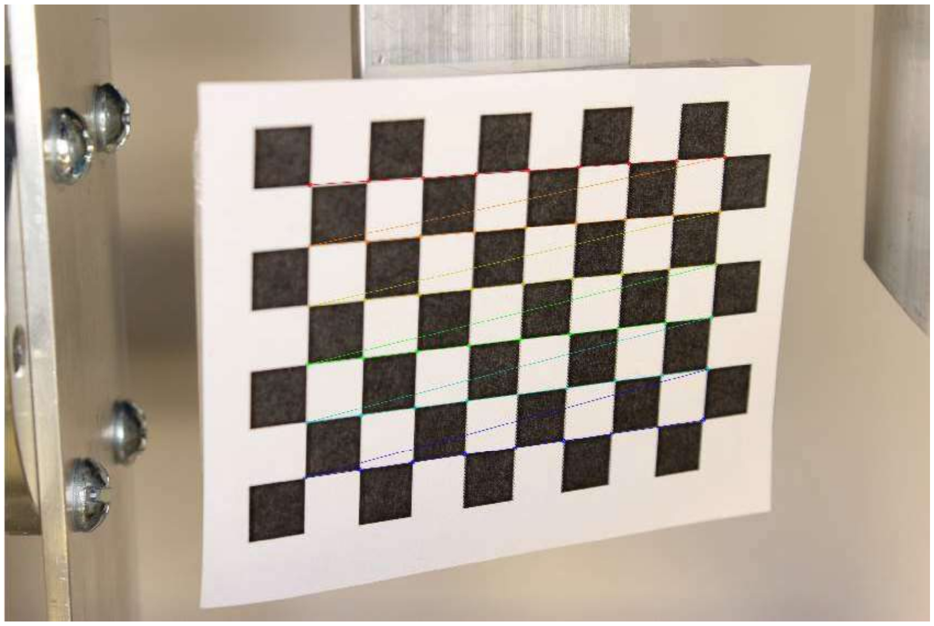


Figure 3.13: Corner extraction example

3.5.2 Turntable calibration

Once we extract the corner points in the image **m** and the world coordinate **M**, we can use the projection equation 3.2 to estimate the camera parameters. Opencv provide a function called *calibrateCamera()* to do this job for us.

This function estimates the intrinsic camera parameters and extrinsic parameters for each of the views with three steps:

1. Compute the initial intrinsic parameters or read them from the input parameters.
2. Estimate the initial camera pose as if the intrinsic parameters have been already known.
3. Run the global Levenberg-Marquardt optimization algorithm to minimize the reprojection error with constraint of turntable movement.

After this process, we can get an initial estimation of the intrinsic camera parameters as well as extrinsic parameters for each images. An example result is shown in Figure 3.14.

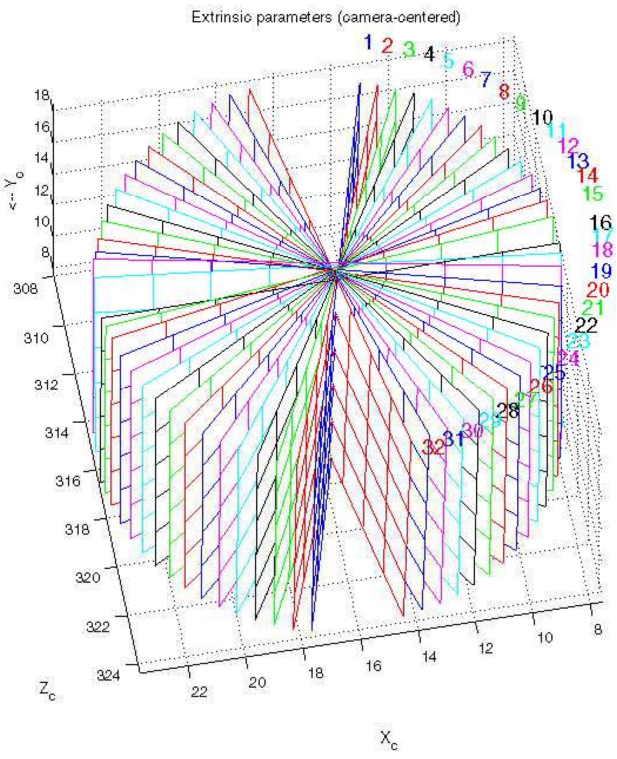


Figure 3.14: Initial Calibration results from Matlab

This method can provide the extrinsic parameters for each image, and only these images. Which means that taking pictures of the specimen at the same position is needed so that the same extrinsic parameters can be used for later work. In order to get more freedom when capture the specimen images, some more steps has been added to make use of the knowledge of the angle between each image. So that only one extrinsic parameters is necessary, and the angles can be used to generate others' extrinsic parameters, which means all the position of Y-axis for this circle (i.e. X-axis) has been calibrated. Then for all the positions of X-axis, the only thing need to do is to fix the position of Y-axis and rotate X-axis. This will calibrate all the position along X-axis. Then apply global Levenberg-Marquardt optimization algorithm to minimize the re-projection error, After that, the extrinsic parameters for all the possible position of this turntable can be generated. We call this calibration method as "Two-axes Turntable Calibration".

Here is the flow chart of the Two-axes Turntable Calibration:

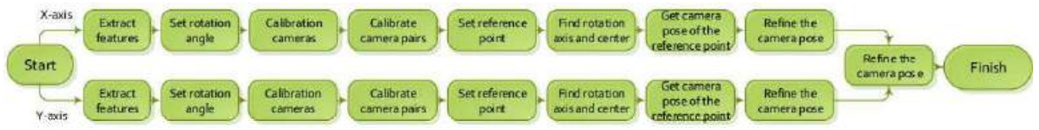


Figure 3.15: Two-axes turntable calibration flow chart

First, let's discuss the calibration of rotating X-axis.

1. Using *findchessboardcorners()* and *cornerSubPix()* to get the coordinate of corners **m** and corresponding 3D model coordinate **M**;
2. Generate angles for each images;
3. Using *calibrateCamera()* to get camera intrinsic parameters (i.e. camera matrix and distortion coefficients) as well as extrinsic parameters **r,t**;
4. Using *stereoCalibrate()* to get rotation-transformation between camera pairs $R_{12}[k]$, $T_{12}[k]$, where k stands for k^{th} camera pair;
5. Set a reference point t_{mid} at the center of the calibration pattern (in the model coordinate space, specified by object points);
6. Using $R_{12}[k],, T_{12}[k]$ to rotate-transform other cameras to the first camera – the reference camera;

$$R_1 = R_{12}^T R_2 \tag{3.5}$$
$$T_1 = R_{12}^T (T_2 - T_{12}) \tag{3.6}$$

   Then the reference point t_{mid} can be projected to the world coordinate t_{pos} by:

$$t_{pos} = T_1 + R_1 t_{mid} \tag{3.7}$$

   $t_{pos}[k]$ for the k^{th} camera;
7. Stack distance vectors between N points (based on t_{pos}) into a matrix A. The rotation axis n must be perpendicular to these vectors. i.e.

$$An = 0 \tag{3.8}$$

   Then we only need to solve the minimization problem:

$$\min ||A||, \ s.t.||n|| = 1 \tag{3.9}$$

8. Get rotation between rotation axis and X-axis (i,e,$(1.0, 0.0, 0.0)$) from the world coordinate. Therefore we can set the rotation axis align to X-axis;

9. Find the center of the rotation by using algorithm provide by [Coope [1993]];
10. Get camera poses (R, T) from rotation axis and stereo calibration;
11. Get target poses (R_t, T_t) from circle fitting;
12. Apply bundle adjustment to minimize the reprojection error with constraint of turntable movement and save R, T, R_t, T_t.

Then apply the same process for Y-axis, and combine them all to do another bundle adjustment. This method is assuming physically X and Y axis intersect at the origin, which is hard to achieve. So more improvement should be done in the future to deal with such problems.

Figure 3.16 shows the refined calibration result: the detected corners and re-projected points.

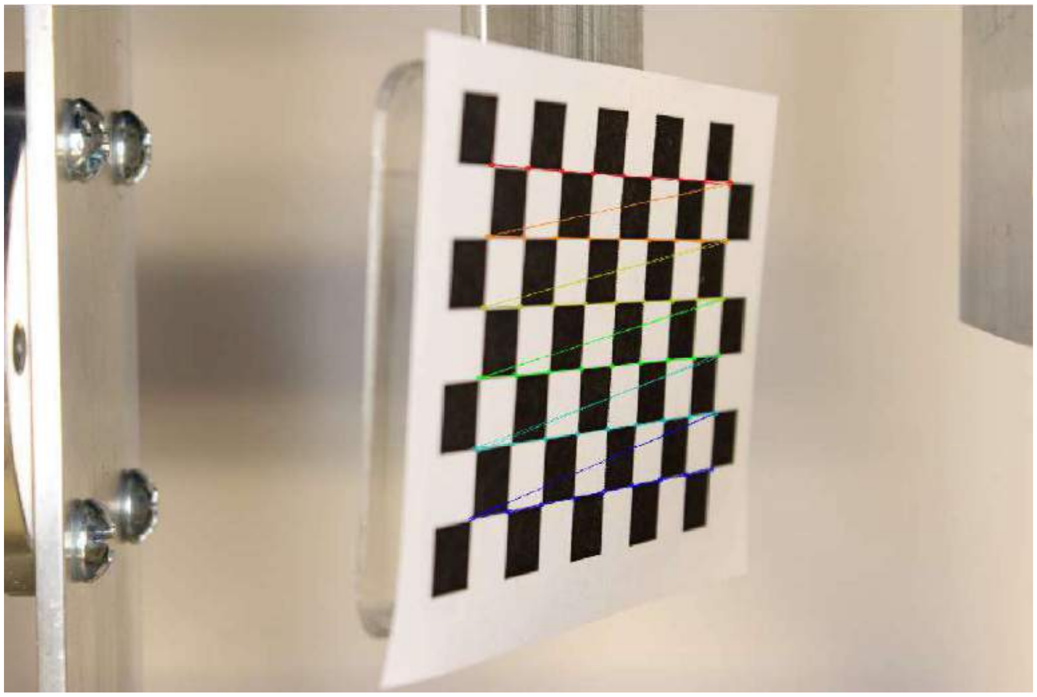


Figure 3.16: Calibration result

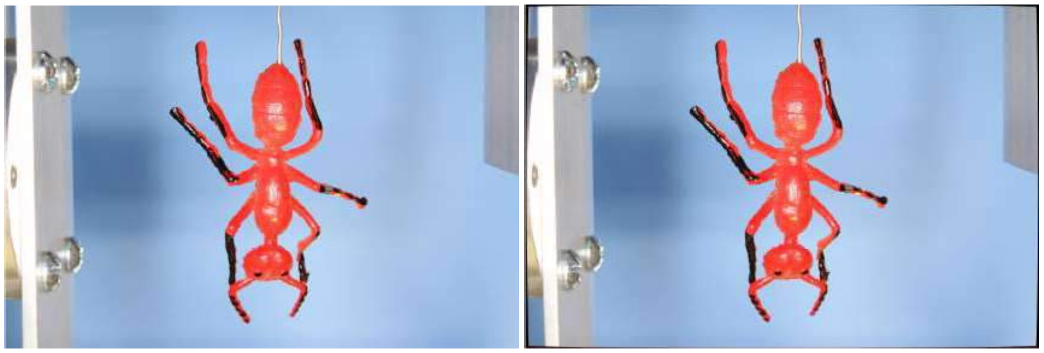
However, there are some errors in two-axes turntable calibration process which lead to large calibration errors. So currently I still considers cameras at different tilt as separate cameras and uses one axis turntable calibration as a temporary solution.

3.5.3 Correct Distortion

After calibrating camera, we get the camera matrix $\mathbf{A}$ and distortion coefficients $(k_1, k_2, p_1, p_2, [k_3])$ as the intrinsic parameters of camera.

Before doing image segmentation, one necessary step is to correct the distortion of the specimen image by using the distortion coefficients. Opencv function *initUndistortRectifyMap()* can be used to builds the maps for the inverse mapping algorithm that is used by *remap()*. That is, for each pixel (u, v) in the destination (corrected and rectified) image, the function computes the corresponding coordinates in the source image (that is, in the original image from camera).

Figure 3.17 compares the original image and the image after correct distortion.



(a) Orignal Image (b) After correct distortion

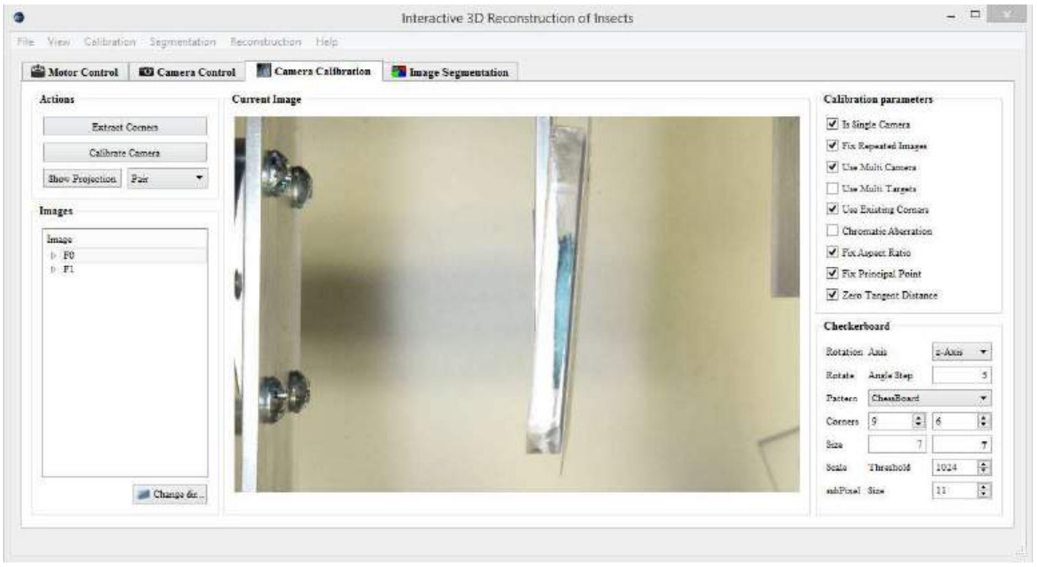
Figure 3.17: Correct distortion result

3.5.4 GUI interface

As shown in Figure 3.18, there are 5 main parts. The Action panel allows user to extract corners, calibrate the camera and show some projections. By showing these projections, user can figure out the performance of calibration and analyze the problems occurred during the calibration. It's much useful by only given the re-projection errors.

The current image panel can show the image which selected in the image panel. Also, after extracting corners, it will show the detected control points in the current image, which allows user to check if the extraction is right.

The two right side panels allows user to set some parameters both in corner extraction phase and camera calibration phase. During this thesis work, OpenCV is found sometimes not able to identify control points if the image size is too large as I mentioned in Subsection 3.5.1. The way to solve this problem is to scale the image down(i.e. 1/2 or 1/3) and allows the opencv to detect the corners on low resolution image, then scales the image as well as the detected points back to the original size and use *cornerSubPix()* function to get more accurate results.



Calibration parameters

- [✓] Is Single Camera
- [✓] Fix Repeated Images
- [✓] Use Multi Camera
- [ ] Use Multi Targets
- [✓] Use Existing Corners
- [ ] Chromatic Aberration
- [✓] Fix Aspect Ratio
- [✓] Fix Principal Point
- [✓] Zero Tangent Distance

Checkerboard

Rotation Axis		z-Axis
Rotate Angle Step		5
Pattern	ChessBoard	
Corners	9	6
Size	7	7
Scale Threshold		1024
subPixel Size		11

Figure 3.18: Camera calibration GUI interface

3.6 Image Segmentation

3.6.1 MeanShift

MeanShift algorithm is a Gradient-ascent-based algorithm. The basic idea of Mean-Shift algorithm [Comaniciu and Meer [2002]] is simple – clusters are places where data points tend to be close together. There is a simple image given below Figure 3.19 illustrate this algorithm:

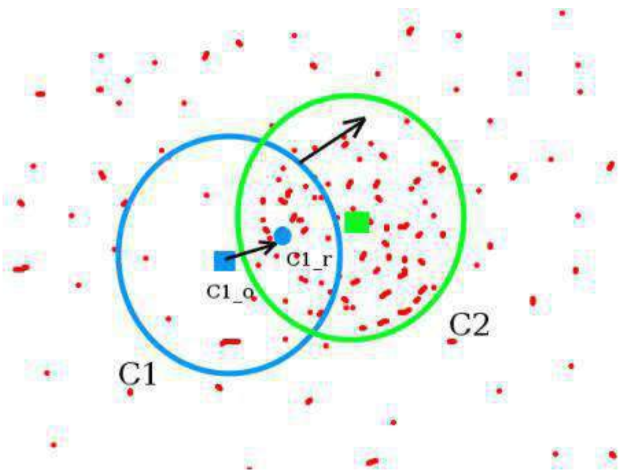


Figure 3.19: Illustrate meanshift algorithm. [provided by opencv document]

The initial window is shown in blue circle with the name "C1". Its original center is marked in blue rectangle, named "C1_o". But if you find the centroid of the points inside that window, you will get the point "C1_r" (marked in small blue circle) which is the real centroid of window. Surely they don't match. So move your window such that circle of the new window matches with previous centroid. Again find the new centroid. Most probably, it won't match. So move it again, and continue the iterations such that center of window and its centroid falls on the same location (or with a small desired error). So finally what you obtain is a window with maximum pixel distribution. It is marked with green circle, named "C2". As you can see in image, it has maximum number of points.

This algorithm has assumed that the data are IID (independent, identically distributed) samples from probability distribution. Therefore that aim is to find local maxima in this probability distribution. The problem is that we don't know the distribution. In this paper [Comaniciu and Meer [2002]], they build a multivariate kernel density distribution model to estimate the actual PDF.

Define a Gaussian bump (kernel):

$$K(x;h) = \frac{(2\pi)^{(-d/2)}}{h^d} exp\left(\frac{1}{2}\frac{||x||^2}{h}\right) \tag{3.10}$$

where x for data, h for a scale parameter, d is dimension.

Place this bump on top of each data point. The model of data density becomes:

$$f(x) = \left(\frac{1}{n}\right)\sum_{i=1}^{n} K(x_i - x; h) \tag{3.11}$$

For simplify the notation, define $k(u) = exp(-\frac{1}{2}u)$, which is called *kernel profile*, and $C = \frac{(2\pi)^{(-d/2)}}{nh^d}$. Therefore we have:

$$f(x) = C\sum_{i=1}^{n} k\left(||\frac{x - x_i}{h}||^2\right) \tag{3.12}$$

We want to find a maximum of $f(x)$, so the gradient of $f(x)$ should be 0:

$$\begin{aligned}
\nabla f(x)|_{x=y} &= 0 && (3.13)\\
&= C\sum_{i=1}^{n} \nabla k\left(||\frac{x_i - y}{h}||^2\right) && (3.14)\\
&= C\frac{2}{h}\sum_{i=1}^{n}[x_i - y]\left[g\left(||\frac{x_i - y}{h}||^2\right)\right] && (3.15)\\
&= C\frac{2}{h}\sum_{i=1}^{n}\left[\frac{\sum_i x_i g(||\frac{x_i - y}{h}||^2)}{\sum_i g(||\frac{x_i - y}{h}||^2)} - y\right] \times \left[\sum_i g\left(||\frac{x_i - y}{h}||^2\right)\right] && (3.16)
\end{aligned}$$

Because $\sum_i g\big(||\frac{x_i-y}{h}||^2\big)$ is nonzero, we have:

$$\left[\frac{\sum_i x_i g(||\frac{x_i-y}{h}||^2)}{\sum_i g(||\frac{x_i-y}{h}||^2)} - y\right] = 0 \tag{3.17}$$

Means that:

$$y = \frac{\sum_i x_i g(||\frac{x_i-y}{h}||^2)}{\sum_i g(||\frac{x_i-y}{h}||^2)} \tag{3.18}$$

Then we have a iteration equation:

$$y^{(j+1)} = \frac{\sum_i x_i g(||\frac{x_i-y^{(j)}}{h}||^2)}{\sum_i g(||\frac{x_i-y^{(j)}}{h}||^2)} \tag{3.19}$$

In every iteration, compare the value between $y^{(j+1)}$ and $y^{(j)}$, if it less than a manually chose threshold, break the loop. In Opencv, a maximum number of iteration should be set for more efficiency: if it didn't reach the threshold but reach the maximum No. of iteration, break the loop. The set of all locations that converge to the same mode defines the basin of attraction of that mode. The points which are in the same basin of attraction is associated with the same cluster.

This a robust algorithm and doesn't require prior knowledge of the number of clusters and doesn't constrain the shape of the clusters. For better performance, I always perform *connect and fill* process before doing MeanShift. This can be automatically performed by opencv cv2.meanShift() function.

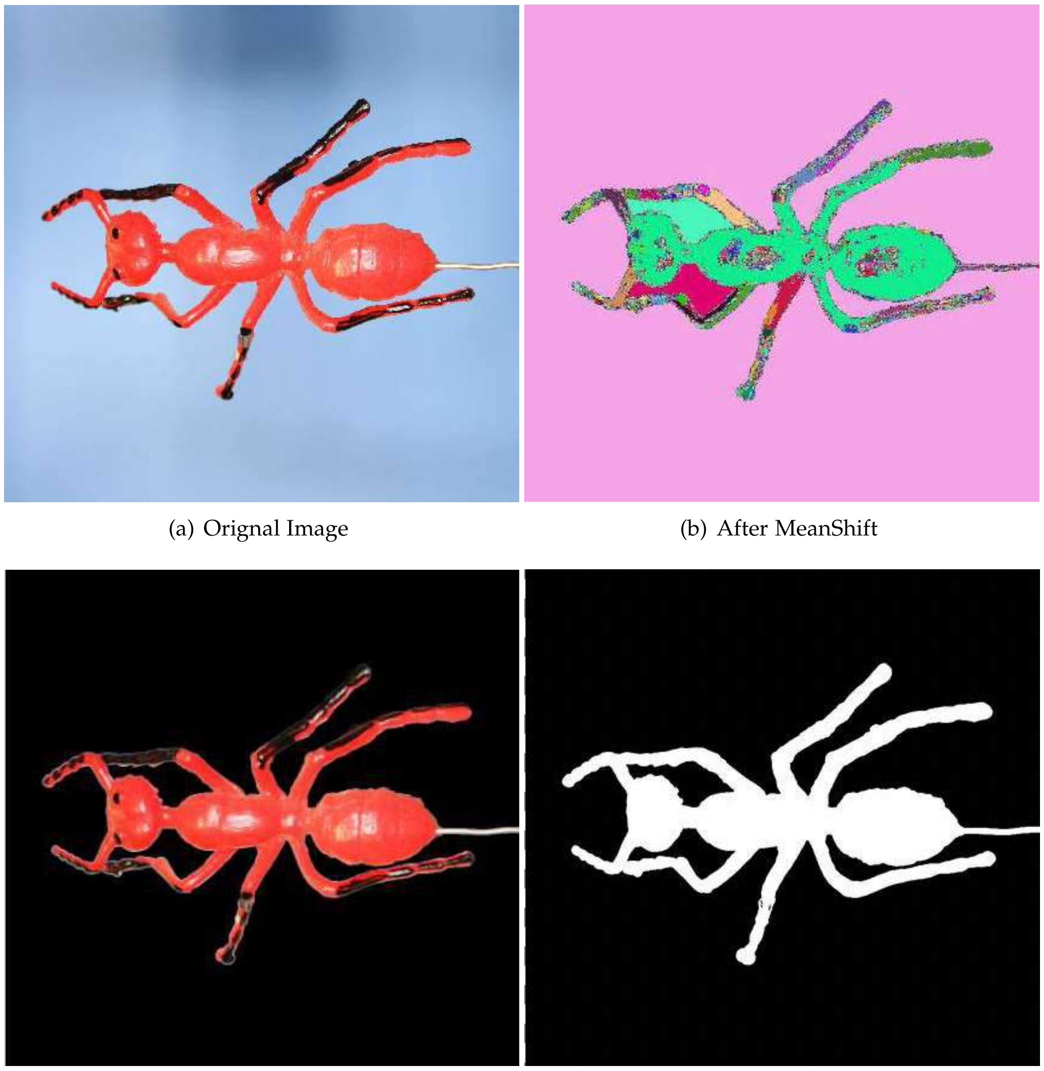
One example of MeanShift result is shown in Figure 3.20.

As we can see it has good performance if we have a clean background. However, once there are other objects on the background, it will segment other objects other than the target. In other words, it doesn't have the ability to remove background. Another drawback of this algorithm is that it cost 10 more seconds to segment a image at size (2592*1728) on my computer.

3.6.2 Grab Cut

Unlike MeanShift, Grab Cut Rother et al. [2004] is a Graph-based algorithm. Also, it's a interactive segmentation algorithm. It uses the color information and border information of the image and combined some user interaction to perform image segmentation. The process of iterative image segmentation in GrabCut is shown in Figure 3.21.

Unlike graph cut, which need the user to point out some seed points for the object and background, the set of background pixels provide enough information to perform GrabCut. That makes automatically segmenting a set of images become possible – the only needed information is a initial bounding box for these set of images. Because in my experiment, the target is always in the middle of the image.


(a) Orignal Image (b) After MeanShift

(c) Segmented Image (d) silhouette of the target

Figure 3.20: Meanshift Result

In opencv, there is a function that can perform GrabCut while there is no function to do the border matting. However, during the experiment, I found this algorithm is very slow (more than 1 min for the first iteration) for large bounding box (i.e. 800*600), which means the computational complexity of this algorithm may not grow linearly along with the size of the image. Also, if I only given the bounding box, even after more than 10 iteration, the result is not acceptable, which is shown in Figure 3.22.

However, as shown in Figure 3.22, this algorithm has capability to remove the background and as long as there are enough interactions, we can always get the

Initialisation

- User initialises trimap T by supplying only T_B. The foreground is set to $T_F = \emptyset; T_U = \overline{T}_B$, complement of the background.
- Initialise $\alpha_n = 0$ for $n \in T_B$. and $\alpha_n = 1$ for $n \in T_U$.
- Background and foreground GMMs initialised from sets $\alpha_n = 0$ and $\alpha_n = 1$ respectively.

Iterative minimisation

1. Assign GMM components to pixels: for each n in T_U,

$$k_n := \arg\min_{k_n} D_n(\alpha_n, k_n, \theta, z_n). \tag{3.20}$$

2. Learn GMM parameters from data $\mathbf{z}$:

$$\underline{\theta} := \arg\min_{\underline{\theta}} U(\underline{\alpha}, \mathbf{k}, \underline{\theta}, \mathbf{z}). \tag{3.21}$$

3. Estimate segmentation: use min cut to solve:

$$\min_{\{\alpha_n : n \in T_U\}} \min_{\mathbf{k}} \mathbf{E}(\underline{\alpha}, \mathbf{k}, \underline{\theta}, \mathbf{z}). \tag{3.22}$$

4. Repeat from step 1, until convergence.
5. Apply border matting.

User editing

- *Edit*: fix some pixels either to $\alpha_n = 0$ (background brush) or $\alpha_n = 1$ (foreground brush); update trimap T accordingly. Perform step 3 above, just once.
- *Refine operation*: [optional] perform entire iterative minimisation algorithm.

Figure 3.21: Iterative image segmentation in GrabCut

deserted result. So, this algorithm is more suitable for dealing with some images that other algorithm fail to segment.

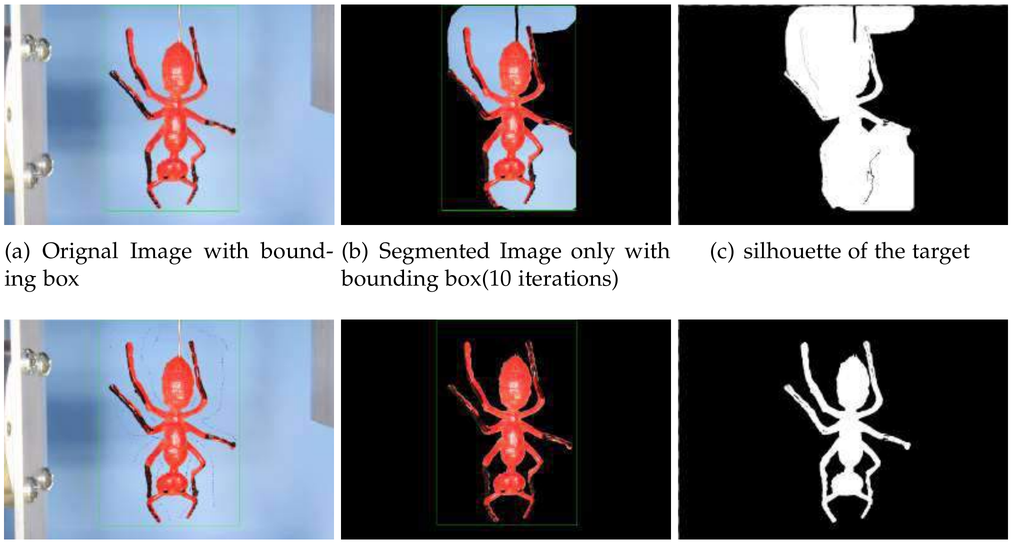
3.6.3 Background Removal

This background removal algorithm is provided by [Lo et al. [2006]], which can efficiently remove the background (less than 1 second for processing 8 images) and has a relatively good performance on segmentation if the background color is not similar with the target. The background images should be provided for this algorithm.

For static background this algorithm is based on two properties:

1. Between-Pixel Invariants.
2. Within-Pixel Invariants.

Suppose $I(x,y)$ is the intensity value of the pixel located at (x,y), $E(x,y)$is the



(a) Orignal Image with bounding box

(b) Segmented Image only with bounding box(10 iterations)

(c) silhouette of the target

(d) Orignal Image with bounding box and seed points

(e) Segmented Image with lots of interaction (8 iterations)

(f) silhouette of the target

Figure 3.22: GrabCut Result

irradiance of the 3D point projecting to (x, y) and $\rho(x, y)$is the diffuse reflectance of the same 3D point. Then build a luminance model based on the assumption of Lambertian reflectance can be expressed as:

$$I(x, y) = E(x, y)\rho(x, y) \tag{3.23}$$

The property between-pixel invariants assumes the 3D points projecting to neighboring pixels receive the same irradiance: $E(x, y) = E(x + 1, y)$. Therefore we have:

$$\frac{I(x, y)}{I(x + 1, y)} = \frac{E(x, y)\rho(x, y)}{E(x + 1, y)\rho(x + 1, y)} = \frac{\rho(x, y)}{\rho(x + 1, y)} \tag{3.24}$$

Which means the ratio holds roughly no matter it is covered by shadow or not.

The second property within-pixel invariants assumes the color of the illumination do not change by the effect of shadow. Therefore we have:

$$\frac{R(x, y)}{B(x, y)} = \frac{E_r(x, y)\rho(x, y)}{E_b(x, y)\rho(x, y)} = E_c(x, y)\frac{\rho_r(x, y)}{\rho_b(x, y)} \tag{3.25}$$

Where $R(x, y), B(x, y)$ stand for the red and blue channels' value. $E_c(x, y)$ is the radio between $E_r(x, y)$ and $E_b(x, y)$ which is a constant based on the assumption.

Let's define I to be the current image, I' to be the background image. So, if pixel

(x, y) is in shadow region, we have:

$$\frac{I(x,y)}{I(x+1,y)} = \frac{I'(x,y)}{I'(x+1,y)} \tag{3.26}$$

Then define two ratio maps, $d_h(x,y)$ and $d_v(x,y)$ as the horizontal and vertical first-order derivative mask. Also,$d'_h(x,y), d'_v(x,y)$ can be defined similarly.

$$d_h(x,y) = \frac{I(x,y)}{I(x+1,y)} \tag{3.27}$$

$$d_v(x,y) = \frac{I(x,y)}{I(x,y+1)} \tag{3.28}$$

A simple pixel-wise comparison between $d(x,y)$ and $d(x,y)$ can be used to determine whether a pixel belongs to shadow regions or not because a pixel is classified as shadow only if its value in the ratio map (texture information) is similar to those in the background. A error score $\Psi(x,y)$ is introduced for discriminating the pixel (x, y) as shadow.

$$\Psi(x,y) = |d_h(x,y) - d_h(x,y)| + |d_v(x,y) - d_v(x,y)| \tag{3.29}$$

Same as the first property, by using the second property we have:

$$\frac{R(x,y)}{B(x,y)} = \frac{R'(x,y)}{B'(x,y)} \tag{3.30}$$

if pixel (x, y) is in shadow region.

Also, define $r(x,y) = \frac{R(x,y)}{B(x,y)}, g(x,y) = \frac{G(x,y)}{B(x,y)}$, similarly we have another error score:

$$\Theta(x,y) = |r(x,y) - r'(x,y)| + |g(x,y) - g'(x,y)| \tag{3.31}$$

Then use a variable Ω to combine these two factors as a single one:

$$\Omega(x,y) = \frac{a\Psi(x,y) + b\Theta(x,y)}{a+b} \tag{3.32}$$

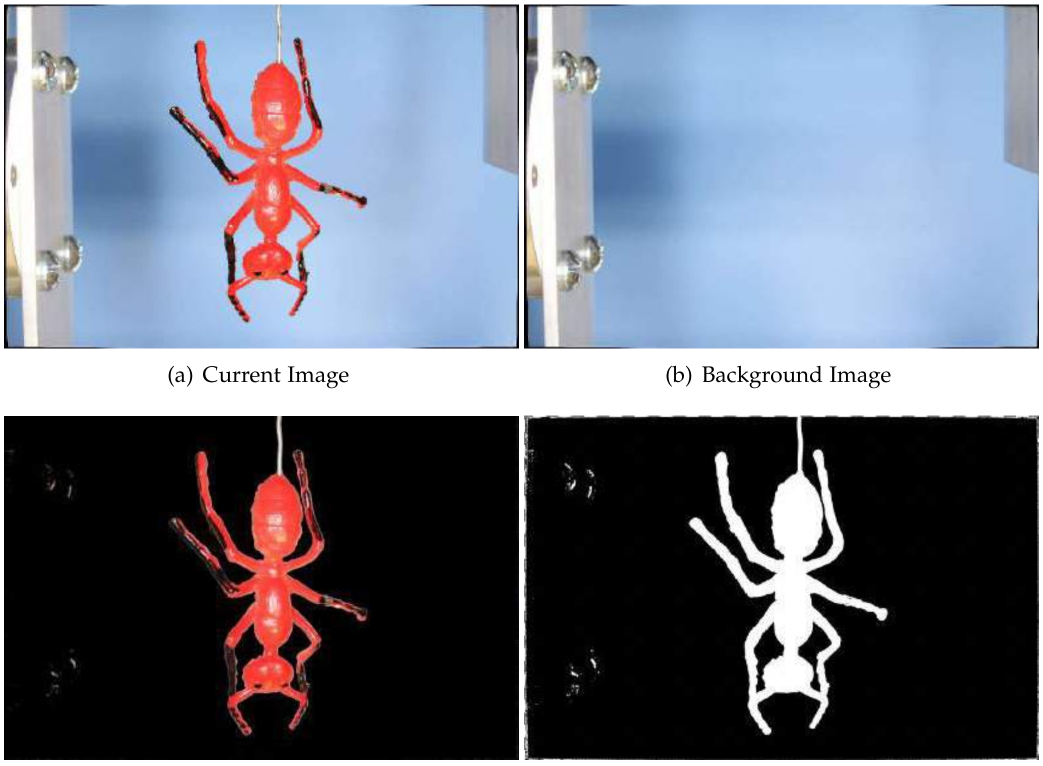
where a, b are the weight parameters of each factors.

Finally, a thresholding operation is applied on $\Omega(x,y)$ to determine whether the pixel (x, y) belongs to foreground object or cast shadow region.

The performance of this algorithm is shown in Figure 3.23.

3.6.4 GUI interface

Similar with the camera calibration interface, it allows user to set parameters for MeanShift and GrabCut segmentation, as well as output parameters.


(a) Current Image (b) Background Image

(c) Segmented Image ($\Omega(x,y) < 0.1$) (d) silhouette of the target

Figure 3.23: Background Removal Result

For background removal, there is an another setting panel as shown in Figure 3.25.

User should provide the target file which contains all the images' names as well as the background one. The *.txt* files should in the same folder along with the images.

3.7 3D Reconstruction

3.7.1 Methods

Patch-based Multiview Stereo This is a novel algorithm for calibrated multiview stereopsis that outputs a (quasi) dense set of rectangular patches covering the surfaces visible in the input images.

This algorithm does not require any initialization in the form of a bounding volume, and it detects and discards automatically outliers and obstacles. It does not perform any smoothing across nearby features, yet is currently the top performer in terms of both coverage and accuracy for four of the six benchmark datasets presented.

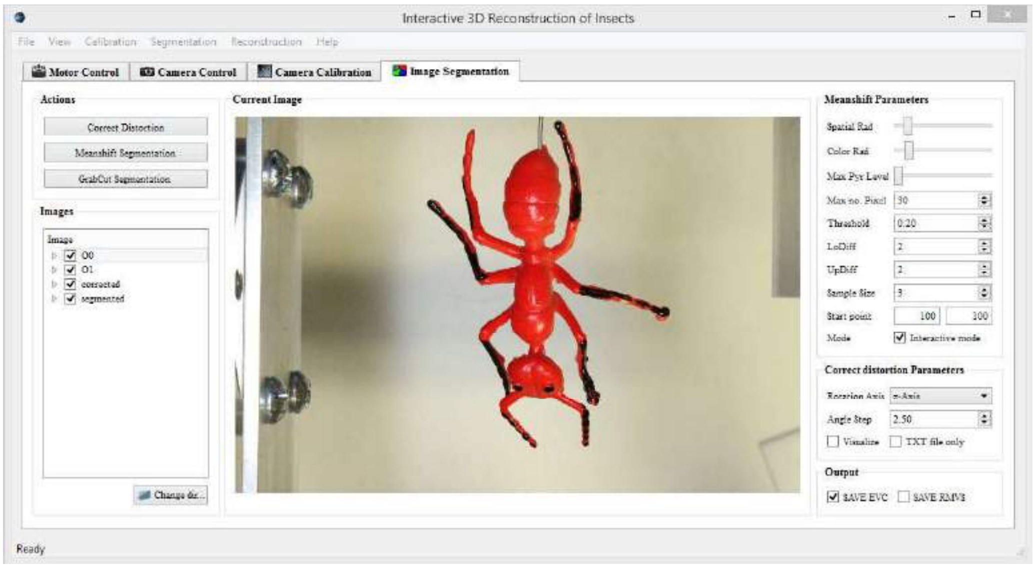


Image
- [✓] O0
- [✓] O1
- [✓] corrected
- [✓] segmented

Meanshift Parameters

Spatial Rad	
Color Rad	
Max Pyr Level	
Max no. Pixel	30
Threshold	0.20
LoDiff	2
UpDiff	2
Sample Size	3
Start point	100 100
Mode	[✓] Interactive mode

Correct distortion Parameters

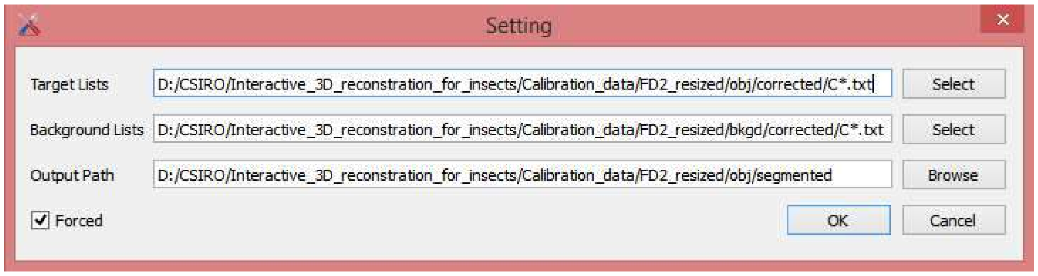
Rotation Axis	z-Axis
Angle Step	2.50

[ ] Visualize [ ] TXT file only

Output

[✓] SAVE EVC [ ] SAVE RMVB

Figure 3.24: Image segmentation GUI interface



Target Lists	D:/CSIRO/Interactive_3D_reconstration_for_insects/Calibration_data/FD2_resized/obj/corrected/C*.txt
Background Lists	D:/CSIRO/Interactive_3D_reconstration_for_insects/Calibration_data/FD2_resized/bkgd/corrected/C*.txt
Output Path	D:/CSIRO/Interactive_3D_reconstration_for_insects/Calibration_data/FD2_resized/obj/segmented

[✓] Forced

Figure 3.25: Background removal GUI interface

The keys to its performance are effective techniques for enforcing local photometric consistency and global visibility constraints. Stereopsis is implemented as a match, expand, and filter procedure, starting from a sparse set of matched key points, and repeatedly expanding these to nearby pixel correspondences before using visibility constraints to filter away false matches.

The bottleneck of this multi-view stereo matching algorithm is the patch expansion step, whose running time varies from about 20 minutes. The running times of polygonal surface extraction also range from 30 minutes to a few hours depending on the size of datasets. It will be a big challenging to apply this algorithm into real-time 3D reconstruction.

Visual Hull The Visual hull is a concept of a 3D reconstruction by a Shape-From-Silhouette (SFS) technique. This kind of 3D scene reconstruction first has been introduced by Baumgart in his PhD thesis in 1974. The basic principle is to create a 3D representation of an object by its silhouettes within several images from different viewpoints. By using this basic idea there are many advantages in using SFS techniques. First, the calculation for the silhouettes is easily to implement. Second, the implementations of the SFS-algorithm are straight forward. However, there are also disadvantages for these techniques. There are time consuming testing steps that are a bottleneck for real-time applications. This ends up in problems for the intersection of the visual cones and therefore bad results for the resulting 3D shapes. So the result of each SFS algorithm is just an approximation of the actual object's shape.

The main problem of the SFS-based algorithms are that they are not able to perform an accurate reconstruction of concave objects. The limitation is formal defined by [Laurentini, 1994] as following:

"The visual hull $VH(S, R)$ of an object S relative to a viewing region R is a region of E^3 such that, for each point $P \in VH(S, R)$ and each viewpoint $P \in R$, the half-line staring at V and passing through P contains at least a point of S."

"...the visual hull of an object S is the envelope of all the possible circumscribed cones of S. An equivalent intuition is that the visual hull is the maximal object that gives the same silhouette of S from any possible viewpoint."

Based on these definitions, we can find that $VH(S, R)$ is the closest approximation of S, and can be archived by using volume intersection techniques. So if we want to have a higher precision we need more different viewpoints and so more visual cones. The modern approaches use surface-based representations instead of the volumetric representation of the scene, which allows to use regularization in a energy minimization framework.

In this project, I address the visual hull method into insect 3D reconstruction because it produced less errors at thin parts such as wings, legs and antennae as compared to multiview stereo or laser scanner (based on previous work of [Nguyen et al. [2014]]). Although this method have some shortages to resolve concave surfaces, this problem can be partially corrected using photo consistency techniques. Therefore the current problem I am trying to solve is to develop a technique to capture optimal image views: enough information (for accurate reconstruction) but with less repeated information.

3.7.2 Implicit surface polygonizer

The idea of visual hull is quite simple: project the silhouette of the target into a virtual volume at each viewing angle and carving away the volume outside the silhouette to leave a 3D visual hull which approximates the shape of the actual target. That means the more angles of images provided, the smoother the produced 3D model's surface. However, based on the experiments, the number of images doesn't affect the processing time much. The resolution of reconstruction is the key factor for determine the processing time, which will be explained later.

Different from *Exact Polyhedral Visual Hulls* [Franco and Boyer [2003]], implicit surface polygonizer is used [Bloomenthal [1994]] to generate the triangles, vertices as well as its normals.

Here is a Table 3.1 that compares implicit and parametric expression.

Factors	Implicit	Parametric
Definition	$f(x,y,z)=0$	$(x,y,z)=F(u,v)$
characterizes	volume	surfaces
blends	easy	hard
inside/outside	easy	hard
point generation	hard	easy
precise control	hard	easy

Table 3.1: Comparison between implicit and parametric expression

For better illustration, suppose we would like to define a sphere centered at C with radius r. It can be described parametrically as {P}:

$$(P_x, P_y, P_z) = (C_x, C_y, C_z) + (r\cos\beta\cos\alpha, r\cos\beta\sin\alpha, r\sin\beta). \tag{3.33}$$

where $\alpha \in (0, 2\pi), \beta \in (-\pi/2, \pi/2)$.

Or using more compact definition by implicit:

$$(P_x - C_x)^2 + (P_y - C_y)^2 + (P_z - C_z)^2 - r^2 = 0 \tag{3.34}$$

Because an implicit representation does not produce points by substitution, root-finding must be employed to render its surface. Polygonization is a method whereby a polygonal (i.e., parametric) approximation to the implicit surface is created from the implicit surface function. It consists of two principal steps:

1. space is partitioned into adjacent cells at whose corners the implicit surface function is evaluated; negative values are considered inside the surface, positive values outside.
2. within each cell, the intersections of cell edges with the implicit surface are connected to form one or more polygons.

According to these steps. let's define the cell's size as the Polygonization resolution. Figure 3.26 shows how the resolution effect the generated result.

This algorithm uses *Exhaustive Search* rather than *Continuation*, although continuation methods only require $O(n^2)$ function evaluations (n is some measure of the size of the object, n^2 corresponds to the object's surface area) while exhaustive search needs $O(n^3)$. The benefit of choosing exhaustive search is its detection of all pieces of a set of disjoint surfaces, which is not guaranteed with continuation methods. So in our program, user should provide a starting point for reconstruction(i.e. in our project, normally should be (0,0,0), because the object is always in the middle of the image). All the disjoint parts (don't connect with the starting point) will be discard during the polygonization. So as shown in Figure 3.23(d), it's ok for some noise in the background as long as it doesn't connect with the target.

However, it does requires no error in the silhouette of the target (i.e. the white

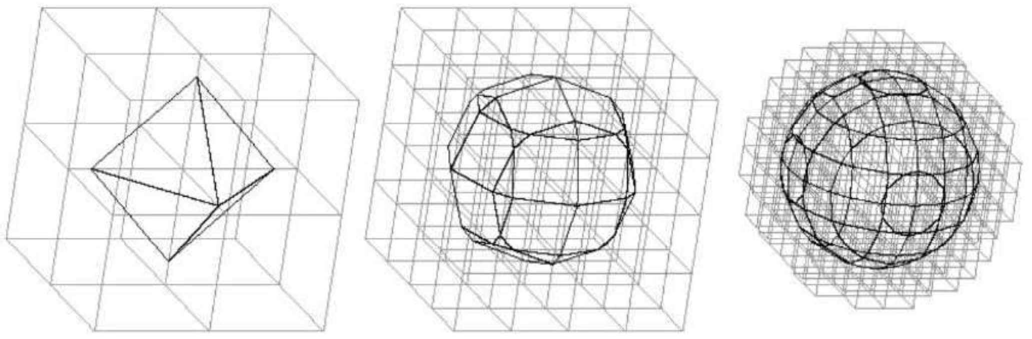


Figure 3.26: Polygonization resolution [SIGGRAPH 2003 Implicit Tutorial]

part). For example, if there is a black point in the middle of the target, it will carve a hole directly through the 3D model with the size of the point. Unfortunately there is no algorithm that can automatically correct these errors, we have to do it manually.

3.8 Summary

In this chapter, I present the details of each component of each system such as turntable controller, camera controller, turntable calibration module, image segmentation module as well as 3D reconstruction module. A new calibration method called turntable calibration method has been proposed during this chapter. In the next chapter, I will discuss the software and hardware platform that I used for creating this system.

Chapter 4

Experimental Methodology

This project is implemented in both software platform and hardware platform. In hardware platform, we built a two-axis turntable rig in order to scan the specimen and in software platform, we write a GUI interface to control the rig and generate the 3D reconstruction model.

4.1 Software platform

Considered this is a cross-platform GUI application, we decided to use Qt – a powerful development environment as our software development environment.

Qt is a cross-platform application framework that is widely used for developing application software with a graphical user interface (GUI) (in which cases Qt is classified as a widget toolkit), and also used for developing non-GUI programs such as command-line tools and consoles for servers.

It has several features:

- Intuitive C++ class library
- Portability across desktop and embedded operating systems
- Integrated development tools with cross-platform IDE
- High runtime performance and a small footprint on embedded

The architecture of Qt SDK is shown in Figure 4.1.

Qt Creator is a cross-platform IDE (integrated development environment) provided by Qt for Qt application.

As an advanced code editor and GUI designer, it integrated GUI layout and forms builder for C++ projects, which allows we to rapidly design and build widgets and dialogs using on-screen forms using the same widgets that will be used in our application. Forms are fully-functional, and they can be previewed immediately to ensure that they will look and feel exactly as we intended.

Moreover, Qt provides Qt Designer, which is a powerful, drag-and-drop GUI layout and forms builder. For writing a GUI interface, we can either use Qt layout system or Qt designer. They both provide the same results. Personally I will choose Qt designer for the list of benefits below:

- Greatly speeds the interface design process
- Enables native look and feel across all supported platforms

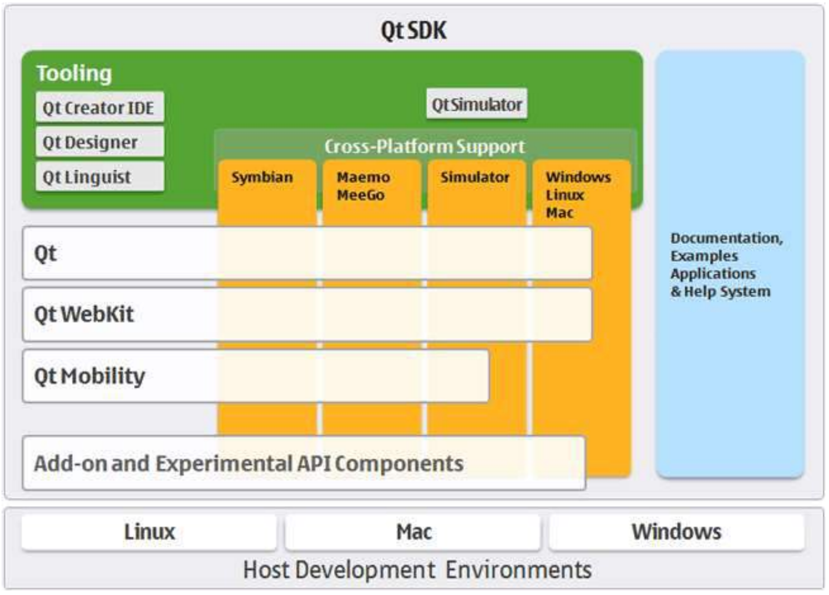
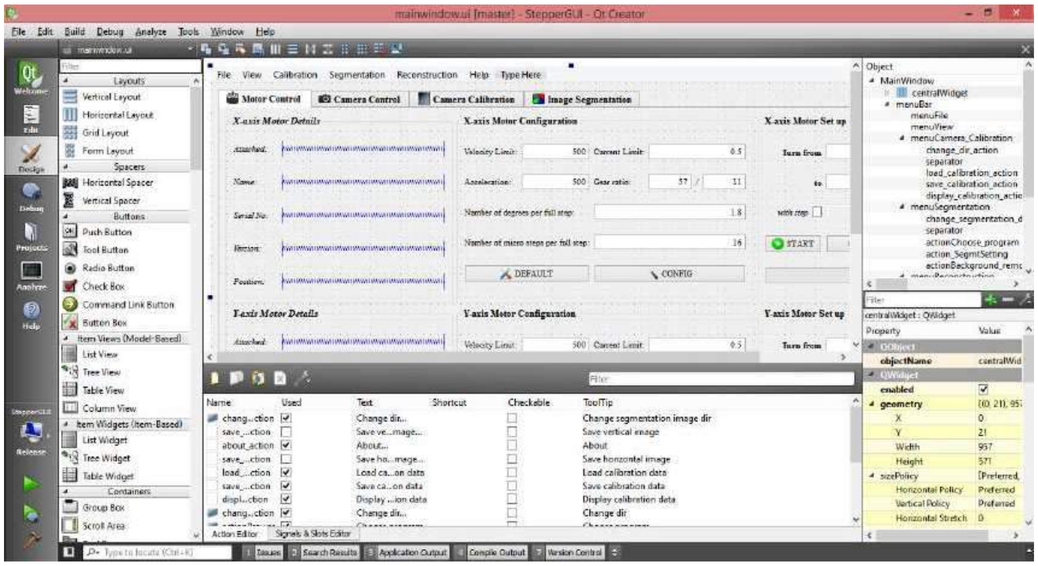


Figure 4.1: The architecture of Qt SDK

- Developers work within the environment of their choice, leveraging existing skills

The Qt Creator and Qt Designer may look like in Figure 4.2.



Name	Used	Text	Shortcut	Checkable	ToolTip
chang...ction	✓	Change dir...		☐	Change segmentation image dir
save_...ction	☐	Save ve...mage...		☐	Save vertical image
about_action	✓	About...		☐	About
save_...ction	☐	Save ho...mage...		☐	Save horizontal image
load_...ction	✓	Load ca...on data		☐	Load calibration data
save_...ction	✓	Save ca...on data		☐	Save calibration data
displ...ction	✓	Display ...ion data		☐	Display calibration data
chang...ction	✓	Change dir...		☐	Change dir

centralWidget : QWidget

Property	Value
QObject	
objectName	centralWid
QWidget	
enabled	✓
geometry	[(0, 21), 95
X	0
Y	21
Width	957
Height	571
sizePolicy	[Preferred,
Horizontal Policy	Preferred
Vertical Policy	Preferred
Horizontal Stretch	0

Figure 4.2: Qt Creator and Qt Designer interface

4.2 Hardware platform

The built rig for scanning specimen is shown in Figure 4.3. It has two stepper motors that can rotate the specimen along X-axis and Y-axis. And a rail to zoom in and out the camera, which controls the distance between the camera and specimen. The two stepper motors and the camera can be controlled by the software I built while the rail is controlled separately.

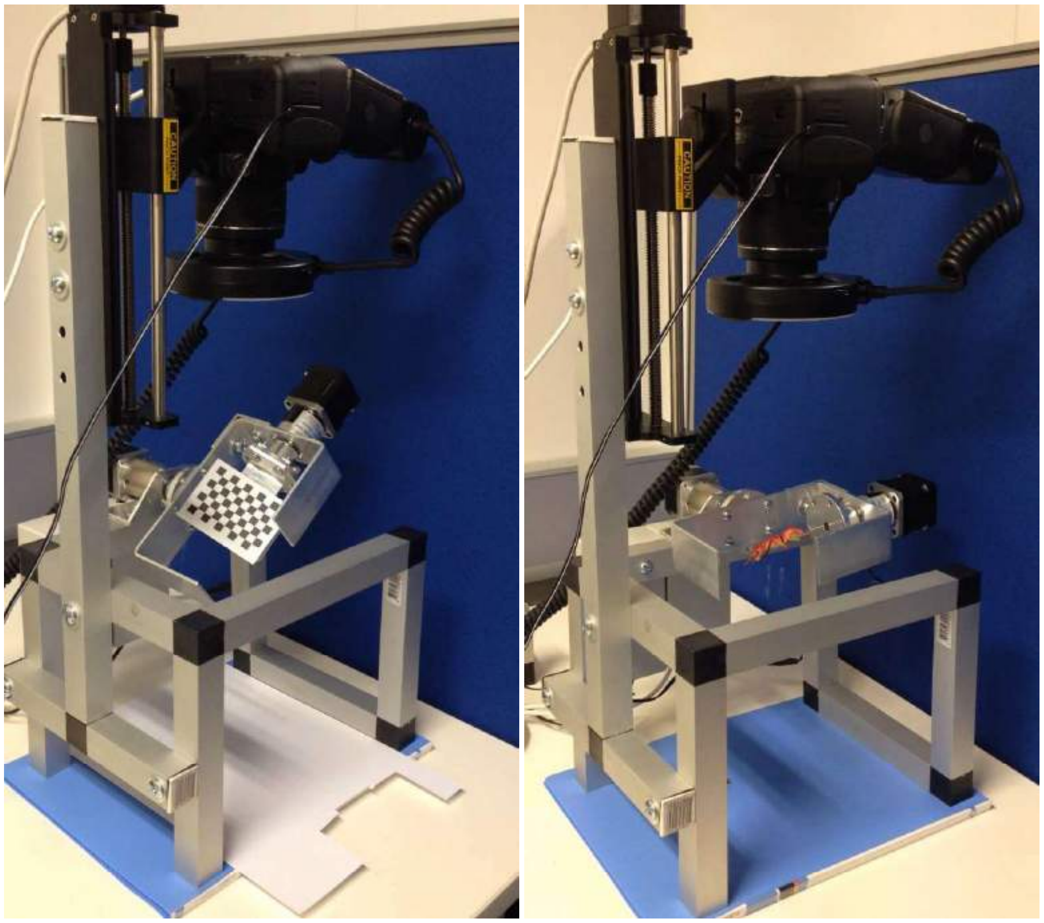

(a) Capturing calibration images (b) Capturing specimen images

Figure 4.3: Capturing images from this rig

Chapter 5

Results

This rig should have the capability for scanning specimen with size from 5mm (or less) to 80mm. For small specimen, the problem will be how to mount them. Currently, due to lack of resources, we only use a needle to mount the specimen.

Here is a demonstration result of an ant model with dimension of $65mm \times 55mm$.

The calibration pattern as well as the specimen we used are shown in Figure 5.1 (each small block in the pattern is about 7mm).

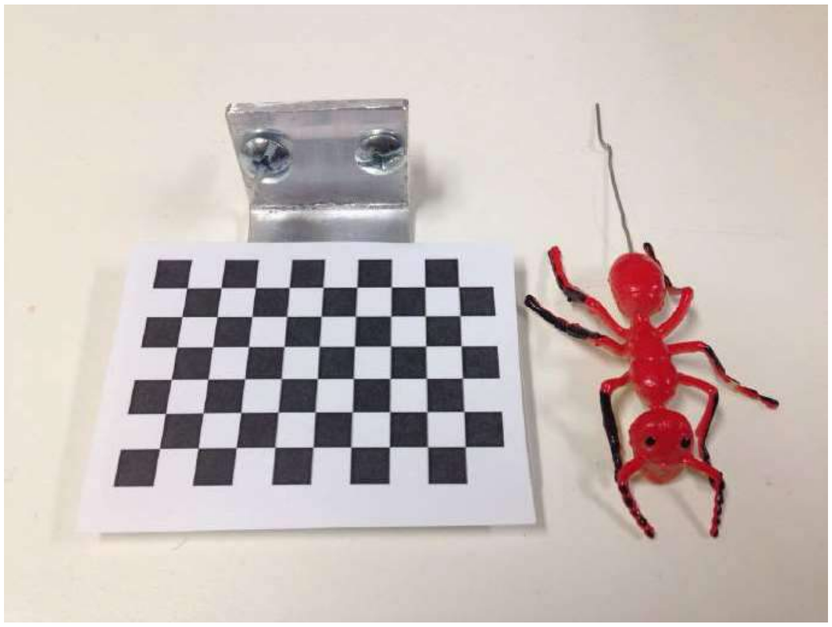


Figure 5.1: Calibration Pattern & Ant model

We define the axis which used for mounting the specimen as Y-axis, and the other one as X-axis. Initially, Y-axis is in the horizontal line. After capturing the calibration images for half a circle (because there is only one calibration pattern and it's on the first side), rotate X-axis to the next position (i.e. 10°). This will calibrate the camera around Y-axis. Then fix the Y-axis, repeat the same process above to calibrate the camera around X-axis.

The calibration error for each position is around 1.0 pixels to 1.3 pixels. Comparing with the size of the images (i.e.1728×2592), the calibration error is small enough for accurate 3D reconstruction. The calibration result data file (partly) is shown in Figure 5.3. The results of correct distortion and image segmentation has already shown in Chapter 3.

In 3D reconstruction phase, I use the reconstruction resolution of $0.08965mm$ and then generate a mesh file (.ply) of the 3D model in about 8 minutes. This file is too large (over $100MB$) for display, so we need to down sample the mesh points to reduce its size to about $10MB$.

Figure 5.2 shows the generated 3D model of the ant model.

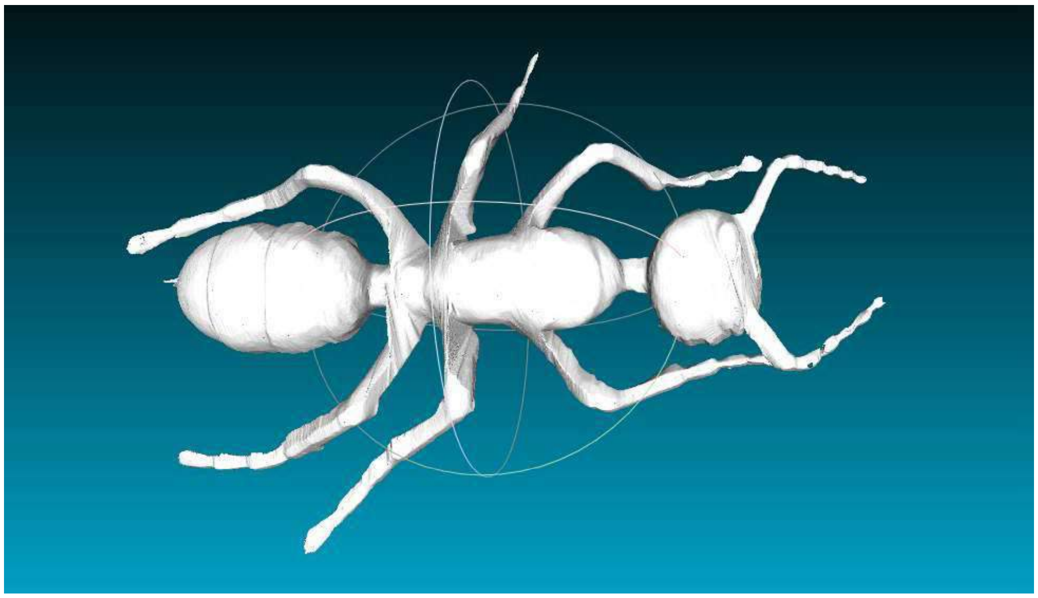


Figure 5.2: Generated 3D model

As we can see in this model, there are some errors at the end of its legs. It's because the base for mounting the specimen is too big. When the X-axis angle is greater than 50 degrees, it will block part of the camera view, which impeded us get enough information from it. Smaller base will fix this problem. Moreover, because we now manually choose additional camera poses, an automatic selection method should be deployed in the future.

```
# physical square size [mm] or distance between control points
square_size: 7.
rotation_axis: !!opencv-matrix
   rows: 3
   cols: 1
   dt: d
   data: [ 1.6051547685840294e-02, -9.9970915242263858e-01, 1.7998843832282379e-02 ]
num_of_cameras: 2
rotation_direction_0: -1
image_width_0: 2592
image_height_0: 1728
camera_matrix_0: !!opencv-matrix
   rows: 3
   cols: 3
   dt: d
   data: [ 5.8568355165064077e+03, 0., 1.2919713458123140e+03, 0.,
       5.8473143463626720e+03, 8.6077786408605834e+02, 0., 0., 1. ]
distortion_coefficients_0: !!opencv-matrix
   rows: 5
   cols: 1
   dt: d
   data: [ -2.1033770711986832e-01, 1.2515956813162539e+01, 0., 0., 0. ]
R_0: !!opencv-matrix
   rows: 3
   cols: 3
   dt: d
   data: [ 9.9969972224256376e-01, 1.7218110490123128e-02, 1.7435653681206927e-02,
           1.7218110490123128e-02, 1.270300777746483e-02, -9.9977105901420782e-01,
           -1.7435653681206927e-02, 9.9977105901420782e-01, 1.2402723020310140e-02 ]
T_0: !!opencv-matrix
   rows: 3
   cols: 1
   dt: d
   data: [ 3.6025428473933894e+00, -2.4529550573112417e+00, 2.524644460982525e+02 ]
# reprojection error [pixel] for the estimated cameras and targets
error_0: 1.2056361843317303e+00
... ...
num_of_targets: 1
# rotation matrix of calibration target when z-axis as rotation axis of turntable
Rt_0: !!opencv-matrix
   rows: 3
   cols: 3
   dt: d
   data: [ 6.0840202427582452e-02, 3.4450675199017122e-01,-9.3681031569990392e-01,
          -6.1131047498733104e-03,9.3865996350811665e-01, 3.447899401915044e-01,
           9.9812879916315822e-01, -1.5250270165609564e-02,5.9214268052455513e-02 ]
# translation vector of calibration target when z-axis as rotation axis of turntable
Tt_0: !!opencv-matrix
   rows: 3
   cols: 1
   dt: d
   data: [-2.5522381841891075e+00, -2.9731658320673830e+01,-1.7041089924653793e+01]
```

Figure 5.3: An example Calibration data file

Chapter 6

Conclusion

This is a novel 3D reconstruction system. In this system, I integrated all the necessary parts for creating a 3D model of small insects, such as turntable controller, camera controller, turntable calibration module, image segmentation module as well as 3D reconstruction module into one GUI software. Moreover, it also includes a 3D model displayer that allows user to display the created 3D model.

By using the knowledge of the actual rotation angle, I present a new turntable calibration method which dramatically reduce the estimation angle error from 1° to 0.011°, and therefore largely reduces the calibration error. Also, because of the turntable calibration method, for each tilt angle, users are allowed to capture specimen images from any arbitrary angles, while the calibration method provided by OpenCV can only use the angles that have been calibrated.

This system has several advantages:

- It allows user to capture high quality images for insects from a few cm to a few mm long, the range cannot be done by alternative methods to create true color 3d models
- Comparing with the commercial 3D reconstruction software shown in Figure, this system doesn't have the limitation of image capturing angle (i.e. not restrict on the upper hemisphere).
- Comparing with Micro CT, it's much cheaper and costs much less time in creating 3D model. Moreover, Micro CT may have difficulty with thin surfaces such as wing. People needs to manually remove the errors and draw the mesh. Also, Micro CT couldn't provide true color 3D model while this system can.

6.1 Future Work

For now I treat each tilt angle as a separate camera, which means we could not generate a camera pose for an arbitrary angle from the sphere, only for each calibrated tilt angle. The second step will be integrating the other axis and combine these two axes to do the two-axes turntable calibration. Therefore, we can generate any camera poses from any angles on the sphere. Also, I will add texture information into the 3D model to create the true color 3D model as shown in Figure 6.1.

Moreover, instead of manually choosing the extra camera poses, an algorithm

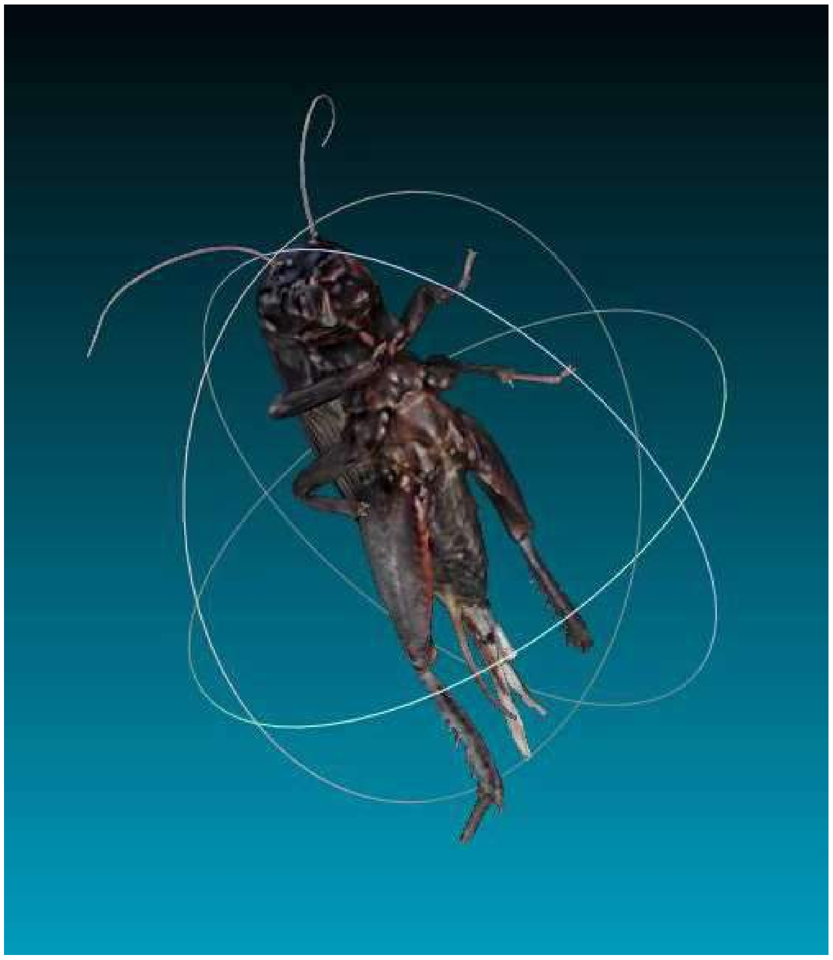


Figure 6.1: A 3D model with texture

called Next Best View can be integrated into the system that allows the software automatically select the best angle for creating more accurate 3D shape.

Bibliography

Akkari, N.; Cheung, D. K.-B.; Enghoff, H.; and Stoev, P. (cited on page 3)

Bloomenthal, J., 1994. An implicit surface polygonizer. In *Graphics Gems IV*, 324–349. Academic Press. (cited on page 34)

Brusco, N.; Ballan, L.; and Cortelazzo, G. M., 2005. Passive reconstruction of high quality textured 3d models of works of art. In *Proceedings of the 6th International Conference on Virtual Reality, Archaeology and Intelligent Cultural Heritage*, VAST'05 (Pisa, Italy, 2005), 21–28. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland. doi:10.2312/VAST/VAST05/021-028. http://dx.doi.org/10.2312/VAST/VAST05/021-028. (cited on page 1)

Comaniciu, D. and Meer, P., 2002. Mean shift: A robust approach toward feature space analysis. *Pattern Analysis and Machine Intelligence,*, 24, 5 (2002), 603–619. (cited on pages 25 and 26)

Coope, I. D., 1993. Circle fitting by linear and nonlinear least squares. *J. Optim. Theory Appl.*, 76, 2 (Feb. 1993), 381–388. doi:10.1007/BF00939613. http://dx.doi.org/10.1007/BF00939613. (cited on page 23)

Faugeras, O., 1993. *Three-dimensional Computer Vision: A Geometric Viewpoint*. MIT Press, Cambridge, MA, USA. ISBN 0-262-06158-9. (cited on page 19)

Franco, J.-S. and Boyer, E., 2003. Exact polyhedral visual hulls. In *IN BRITISH MACHINE VISION CONFERENCE*, 329–338. (cited on page 34)

Laurentini, A., 1994. The visual hull concept for silhouette-based image understanding. *IEEE Trans. Pattern Anal. Mach. Intell.*, 16, 2 (Feb. 1994), 150–162. doi:10.1109/34.273735. http://dx.doi.org/10.1109/34.273735. (cited on page 34)

Lo, K.-H.; Yang, M.-T.; and Lin, R.-Y., 2006. Shadow removal for foreground segmentation. *Advances in Image and Video Technology*, 4319 (2006), 342–352. doi:10.1007/11949534_34. (cited on page 29)

Maybank, S. J. and Faugeras, O. D., 1992. A theory of self-calibration of a moving camera. *The International Journal of Computer Vision*, 8, 2 (Aug. 1992), 123âĂŞ152. (cited on page 18)

Metscher, B. D. (cited on page 1)

Nguyen, C.; DR, L.; M, A.; and J, L. S., 2014. Capturing natural-colour 3d models of insects for species discovery and diagnostics. *PLoS ONE*, 9, 4 (Apr. 2014), e94346. doi:10.1371/journal.pone.0094346. (cited on pages ix, xiii, 1, 2, 4, and 34)

Rother, C.; Kolmogorov, V.; and Blake, A., 2004. Grabcut: Interactive foreground extraction using iterated graph cuts. *ACM Trans. Graph*, 23 (2004), 309–314. (cited on page 27)

Zhang, Z., 2000. A flexible new technique for camera calibration. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22, 11 (Nov. 2000), 1330–1334. doi:10.1109/34.888718. http://dx.doi.org/10.1109/34.888718. (cited on pages xiii, 18, and 19)