University of Burgundy

Computer Science

# Imago 2.0

*Author:*
Devesh Adlakha

*Supervisors:*
Dr. Yohan Fougerolle
Cansen JIANG

May 15, 2015

**Abstract**

Imago 2.0 is an image processing application developed in C++ using Qt 5.1.1 and OpenCV 2.4.8. As part of the Computer Science course module in the second semester of study in the Bachelor of Computer Vision program at the university of Burgundy, France, the project assignment was a graphical user interface program in C++ using Qt Widgets. The requirement was to develop a user-friendly interface following standard object oriented programming practices. Imago is designed to process images, videos and webcam input in a process flow, and features the addition and removal of the pipeline processes, as well as the dynamic variation of their parameters. The available image processing functionalities are: flip, salt and pepper noise, histogram equalization, filtering (low pass, high pass and morphology), Canny edge detection, and the Hough line and circle transforms. The design of both the software and the user interface is presented along with some results from the application. The implementation is subsequently evaluated for its merits and shortcomings, followed by a discussion of ideas for future work. A summary of the project management can be found in appendix [A]. The application is open-source, available on the following github repository: https://github.com/Devesh99/Imago.

# Preface

Imago is in its second release, as the first was developed as part of the Visual Perception course module in the Masters program. That project required using OpenCV to integrate a given list of functionalities in a graphical user interface program. The same development tools, i.e. Qt and C++ were used.

Though functional and completed satisfactorily, the project lacked a sound design strategy, with little object oriented programming employed. The application constituted an assembly of code accomplishing the required functionalities. Process flow was also a lacking feature, and the algorithms could only be tested independently.

Imago 2.0 is completely redesigned to address these shortcoming, and to be of greater utility in image processing. The continuation of this project as part of the Computer Science course module arises from the alignment of their respective objectives. An additional motivation was to review OpenCV as its use is anticipated in a forthcoming internship.

This implementation is inspired from the work of a few colleagues and highly skilled programmers, Marwan Osman, Emre Ozan Alkan and Abinash Pant. The software design is highly influenced by Marwans project, specifically in the structure of the abstract controller class and derived functionality classes. The use of or inspiration from their projects is cited in the code, with a complete list of references maintained in an included text file.

I am grateful to professor Abd El Rahman Shabayek for assigning the Visual Perception projects. The skills and insights gained during their development are much more appreciable in retrospect.

# Contents

# List of Figures

# Chapter 1

# Introduction

Image processing libraries are ubiquitous, with the image processing toolbox in MATLAB[1], and Numpy[2]-based libraries with Matplotlib[3] in Python, being prominent in academia. OpenCV[4] is a free, vast, and well-documented library, usable in C++ to allow fast and powerful processing. Imago is a graphical user interface to OpenCV , developed in C++ using Qt 5.1.1 and OpenCV 2.8.4. The motivating belief is that such a graphical user interface application offers a convenient and even powerful means of accomplishing image processing tasks.

The primary goals of this application are:

- Process flow: cascading processes, with their dynamic addition, removal, and variation of parameters.
- Capable of processing image, video, and webcam input.
- Structured to accomodate functions with varying number and types of parameters.

Imago 2.0 was developed and tested on a MacbookPro with the following specifications:

| | |
|---|---|
| Processor | 2.53 GHz Intel Core 2 Duo |
| Memory | 4 GB |
| OS | 10.8.5 |
| Display | 13" ($1280 \times 800$) |

# Chapter 2

# Design

## 2.1 Software

The development of the application was carried out with the following standard software design objectives:

- Object oriented: following standard and accepted object oriented programming practices.
- Extensibility: structured and simple method for the addition of other image processing functions.
- Reliability: error handling mechanisms, with feedback, if necessary.
- Maintainability: open-source, platform-independent, commented and documented program.

### 2.1.1 Design patterns

Design patterns are templates for dealing with generic, recurring problems in software engineering. The primary motivation was an introduction to a few patterns as a learning experience, and potentially using suitable elements in the development. The following design patterns were found to be particularly relevant, though only a superficial understanding is claimed, and their use is based largely on interpretation:

- Strategy: to encapsulate each algorithm in a class, with its parameters and functioning as member attributes.
- Model-View-Controller: to separate the data, processing, and display.
- Singleton: to ensure a single instance of a class in the execution of the program.

The encapsulation of algorithms in classes follows standard practices in object oriented programming. The distinguished layers of the model, view and controller structures the program for relatively simple design replacement. In this scheme, a unique instance of the controller class ensures expected and controlled behavior.

### 2.1.2 Survey of design strategies

An analysis of previous implementations and design strategies was essential in determining an apt design for the established development objectives and application goals. Based on the design patterns mentioned above, two conventional design strategies were analyzed:

- Controller class containing instance of each algorithm class, with a single process function[5].
- Interface class to all algorithm classes, with a virtual process function[6, 7]. The controller class maintains a pointer to the interface base class.

The viability of the first strategy was assessed through a dummy graphical user interface program. As the controller contained an instance of each algorithm class, the process flow was limited to a single process of each algorithm. Furthermore, the order of processing was determined by the implementation, rather

than user selection. These were also the major drawbacks of [8], which follows a slightly different design. However, dynamic updates to the algorithm parameters were conveniently handled through public mutators of each algorithm class.

The second strategy provided an elegant process flow through a vector of base class pointers, whereby processing reduced to successively calling the process function of each instance. Multiple isntances of an algorithm, and a user-defined ouorder could be maintinaed. However, dynamically updating the algorithm class parameters was not trivial using base class pointers. The use of a pure virtual mutator function constrained each algorithm to have the same signature, thereby the same number and type of parameters. [6] suffers from this limitation, whereas [7] has no provision for dynamic updates.

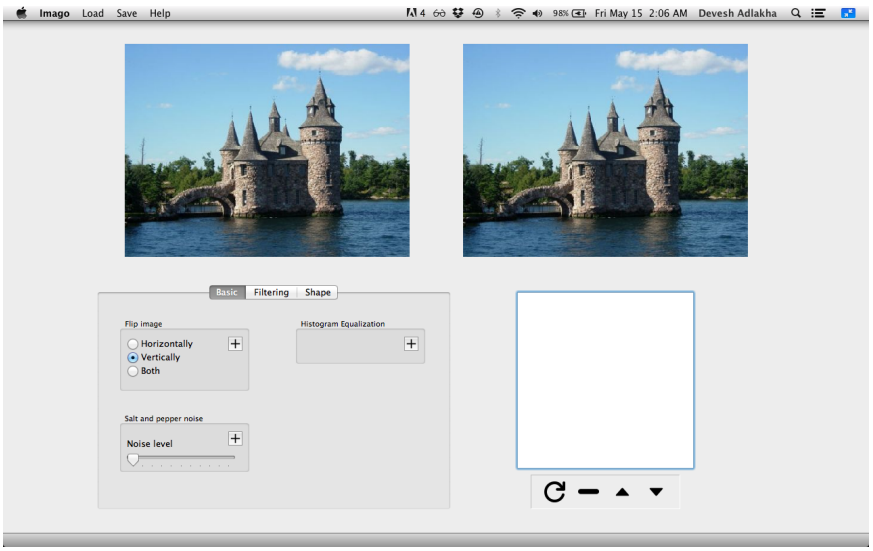



Figure 2.1: UML class diagram of

### 2.1.3 Design strategy

The latter strategy of an abstract interface class suited the application goals, and thus formed the foundation of the design strategy. A means to update the algorithm parameters was lacking, and two solutions supporting mutators with a variable number and type of arguments were conceived::

1. Dynamic casting: a brute force solution with a distinguished mutator member function in each algorithm class. A bse class pointer is dynamically cast to an dalgorithm class in order to access its mutator function, and thereby update its member parameters.
2. Variadic funtions: since these functions accept a variable number of arguments, the base interface class could have a virtual variadic mutator function, defined in each algorithm class according its respective algorithm parametrers.

Dynamic casting was in contradiction to the design objectives, and defeated the purpose of using an interface base class. Furthermore, in keeping with the Model-View-Controller design, the updates were performed in the controller class through defined signals and slots for each algorithm. This process was extremely cumbersome, error-prone and detrimental to exntensibility.

Variadic functions achieved the desired functionality within the design structure. These functions, however, are prone implementation errors as safety measures such as type checking and conversion cannot be carried out by the compiler. Since there was complete knowledge of the algorithms and their parameters in the software, the implementation was meticulous in this regard.

The potential need for accessors also was studied, and they were required to retireive and display the parameters of a selected process in its corresponding user interface widegets. Mutators were given priority in design though, as they were essential to process flow. Whereas, accessors primarily aided in visualization and user experience.

### 2.1.4 Implementation

Figure shows the UML class diagram of the program, comprised by the following major abstractions:

- Interface class: `Iprocesstechnique` is an abstract base class, with pure virtual process and mutator (variadic) functions. This class is derived by each algorithm class.
- Algorithm class: each algorithm is derived from `Iprocesstechnique`, and contains its parameters as member data, along with any helper functions.
- Controller class: `processmanagercontroller`.
- View class: `Imago`, the user interface generated by Qt.

In the model-view-controller scheme, the abstract interface and algorithm classes form the model. The processing is managed in the controller class, with communication with both view and model, as depicted in the UML class diagram.

Processmanager: the controller class encapsulates the processing in the application. The process flow is contained in a vector of base class pointers. The input, output images, timer and video objects are encapsulated within the class.

Imago:

## 2.2 User interface

As Imago 2.0 is essentially a user interface to OpenCV, a user-friendly design was vital. The following objectives were established for the interface design:

- Intuitive: short user learning curve.
- Minimal: simple and clutter-free.
- Uniform: among the widgets, as well as globally.
- Focus user attention: avoiding distractions such as pop-ups.
- Include support: accessible help.

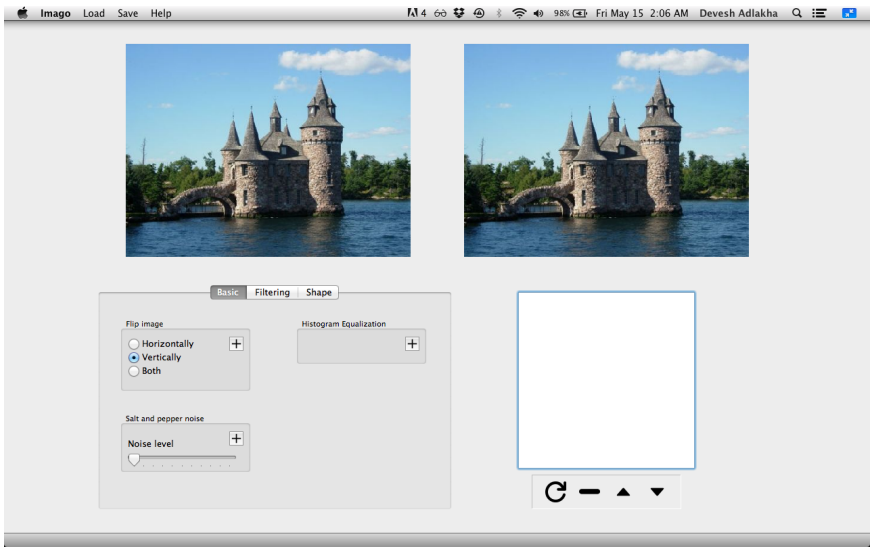



Figure 2.2: User interface

Four elements comprise the interface, shown in figure 2.2.

1. Menu: supplementary functionality contained in a menu bar.
2. View: the input and output displays, which form the focus of attention.

3. Functionality: the image processing functions, organized in a tab container.
4. Process flow: the pipeline display, with its toolbar in a sunken panel.

The menu bar allows loading an input, saving the output, and access to the help search and user manual. The help search, shown in figure 2.3, extends only to the menu bar. Although the application is offline, the user manual is opened in the default web browser of the system to be platform-independent and direct to the latest version. The 'About Imago' option provides application information shown in figure 2.4, and is the only instance of a pop up.

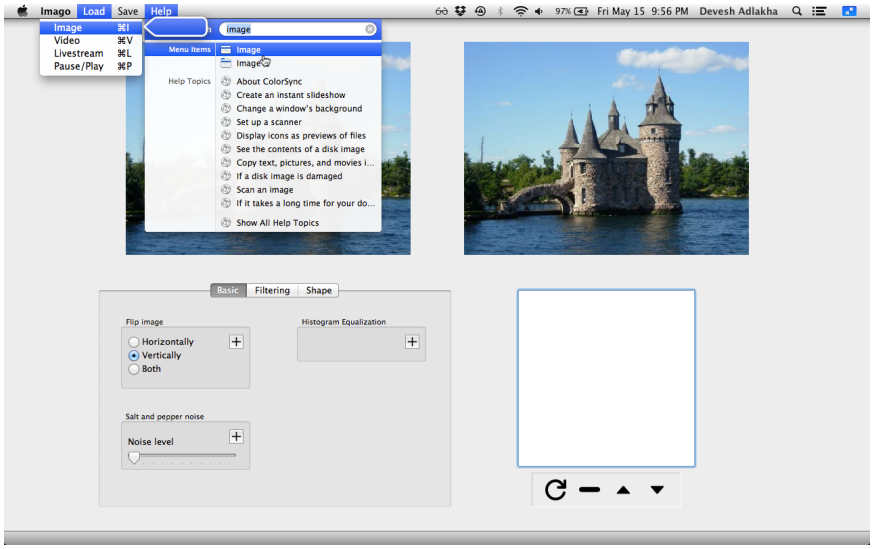



Figure 2.3: Help search

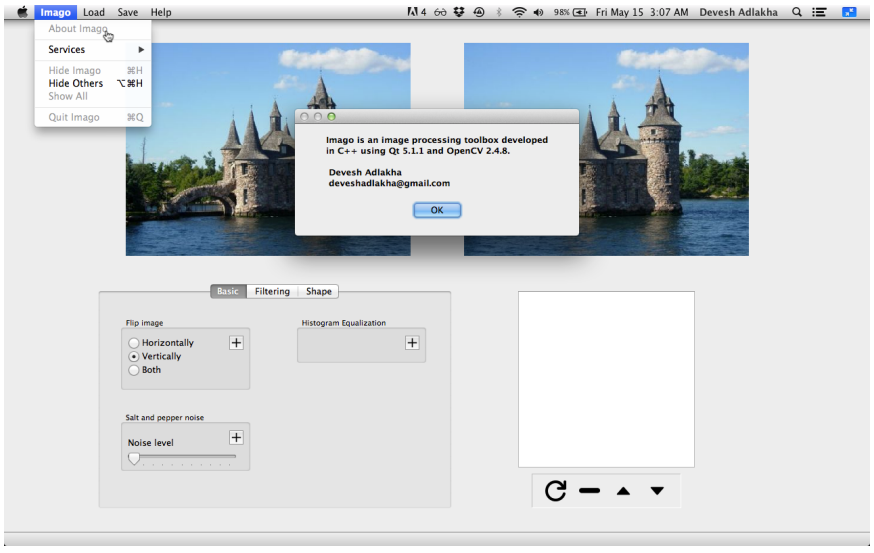



Figure 2.4: About Imago: only instance of a pop-up

The input and output fit to the configured display view size, while maintaining their aspect ratio.

The tab structure divides the image processing functions into three tabs: basic, filtering, and shape. Each function is contained in a group box, with its parameters and an add technique button. Where necessary, the parameters are labeled with descriptions or short forms following OpenCV conventions. The add technique buttons occupy the same location in each group box, and efforts were made to align the group boxes in providing a smooth transition among tabs.

The process flow list displays functions in the pipeline along with their parameter values. As a result, the parameter values of widgets such as sliders are not repeated in the tab structure. The process flow panel provides operations to move, remove, and refresh the processes in the pipeline.

The functions of the menu bar and the process flow toolbar are accessible through keyboard shortcuts. The complete list of the short cut combinations is provided in the user manual. Currently, the tab structure is not accessible through shortcuts, though the standard means of selecting using the keyboard may be employed. Error-handling does not provide feedback, and instead the processing maintains its state for continuity. For instance, an error in loading an input (due to an empty file or canceling the selection dialog box) provides no feedback of the error, and the processing is unaffected and continuous.

# Chapter 3

# Results

Some key results are presented to highlight the implementation details and working of the application. A complete description of all functionalities can be found in the user manual.

- Welcome screen: a default image is loaded for processing, as shown in figure 3.1.

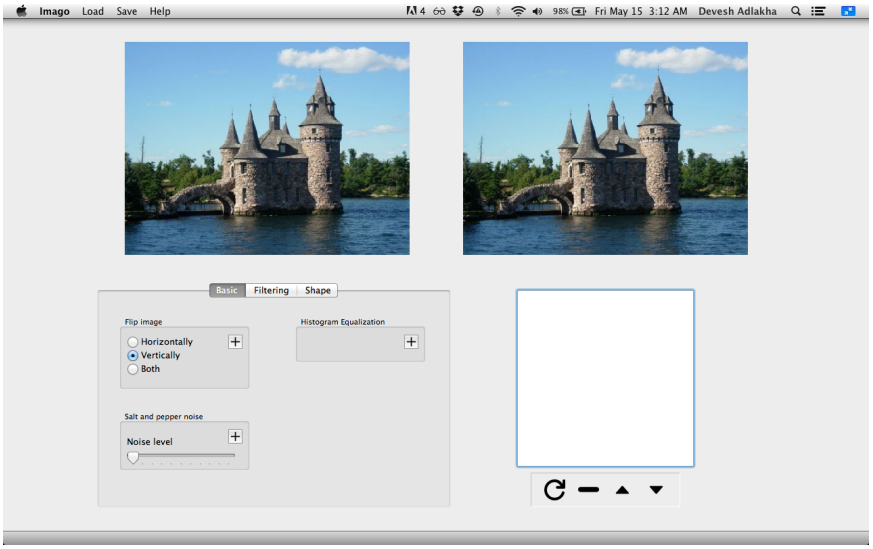



Figure 3.1: Welcome screen

- Add technique: a selected technique is added to the process flow list, and the result displayed in the view, as shown in figure 3.2.

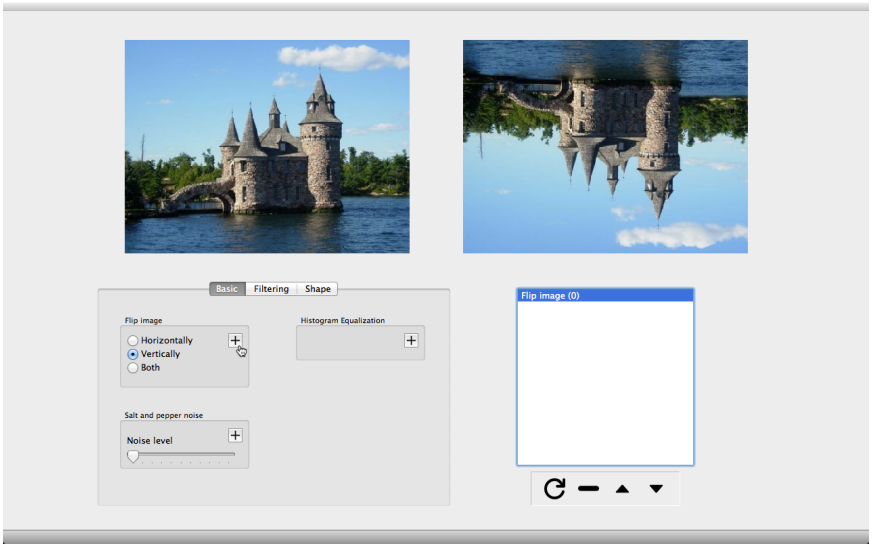



Figure 3.2: Add technique

- Process flow: figure 3.3 shows the result of median filtering a flipped image with added salt and pepper noise. The techniques and their corresponding parameters are displayed in the process flow list.

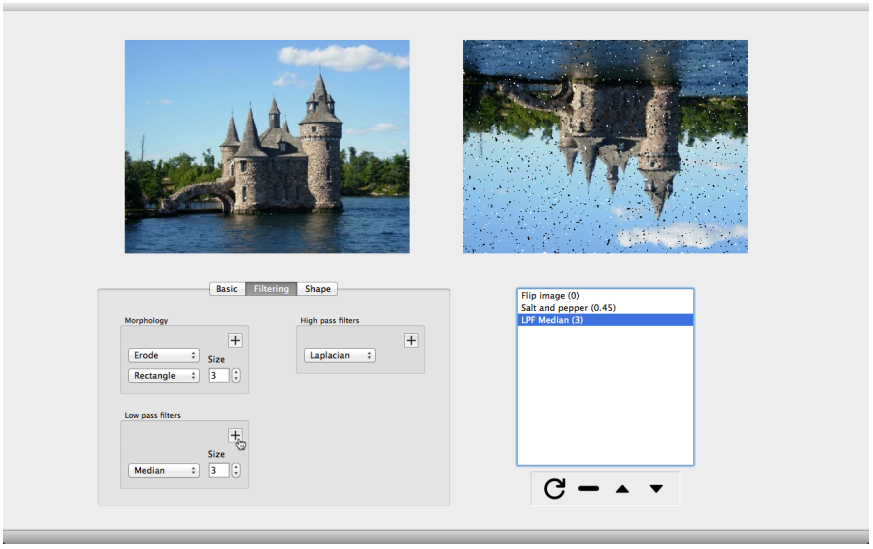



Figure 3.3: Process flow

- Dynamic updates: the parameters of a selected process in the list may be varied. In figure 3.4, the noise level is decreased from 0.45 to 0.2, consequently, the median filter successfully removes most of the added noise.

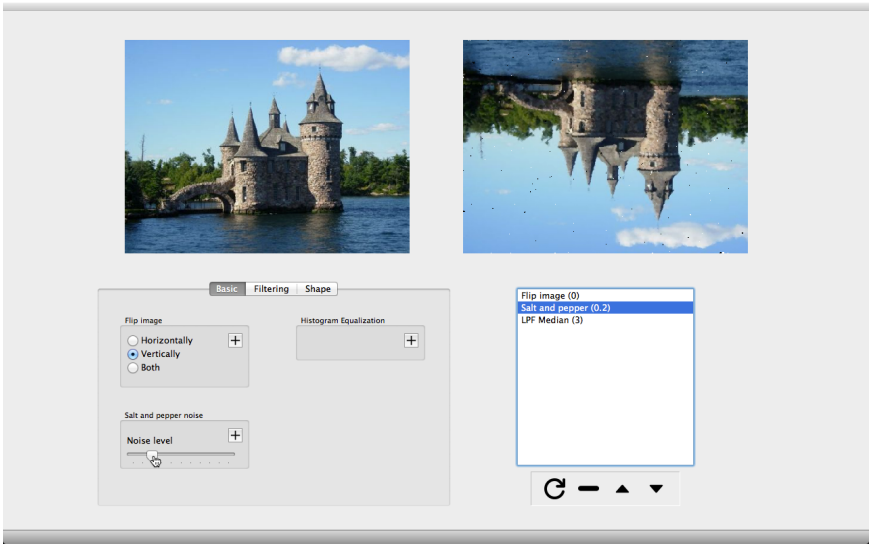



Figure 3.4: Dynamic updates

- Multiple instances of a process: while the noise in figure 3.4 may be reduced further by increasing the size of the median filter, in practice, cascading two filters with smaller kernels may be more efficient. This is shown in figure 3.5 as a demonstration of repeating processes in the flow (their parameters may be different).

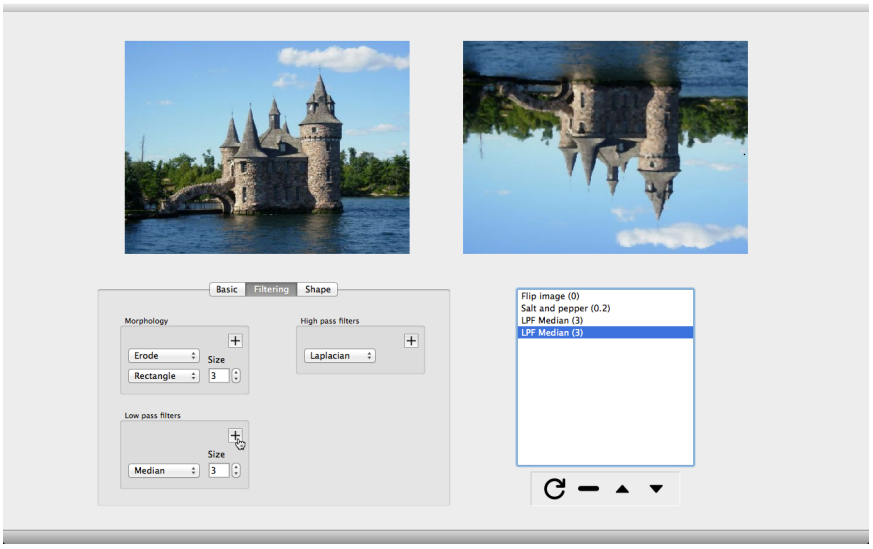
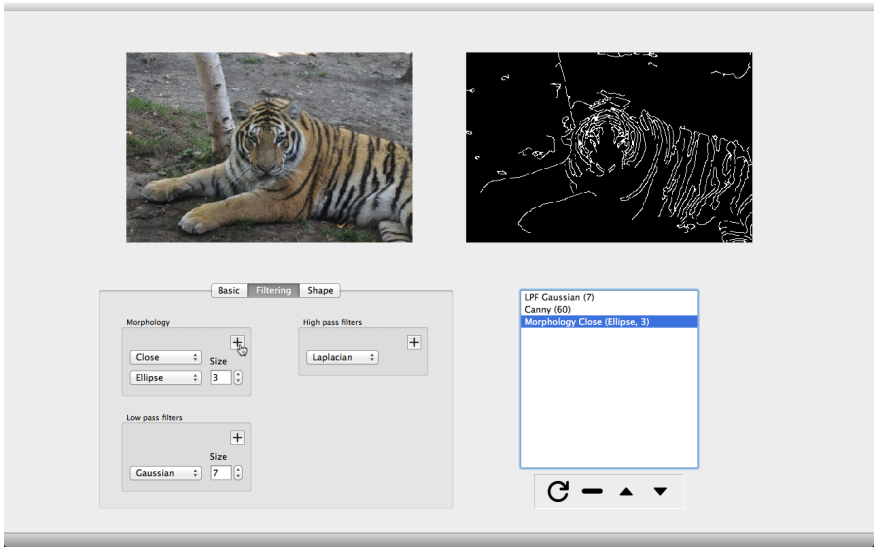
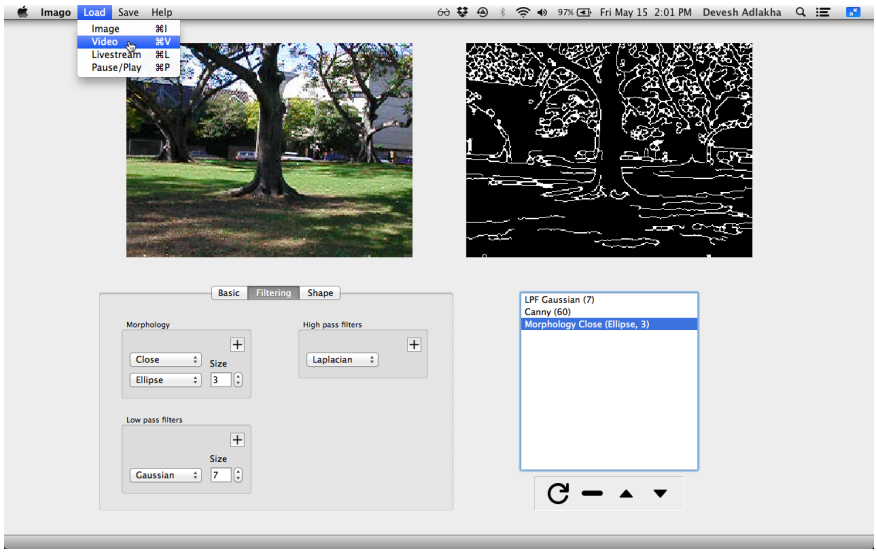



Figure 3.5: Multiple instances of a process

- Processing maintains state: the configured process flow is seamlessly applied to a new input. In figure 3.6, opening a video following the processing of the image in 3.6a maintains and applies the process flow, as in 3.6b.




(a) Image




(b) Video

Figure 3.6: Processing maintains state

- Move process: moving the Gaussian low pass filter to the bottom of the process flow list produces the result in figure 3.7.

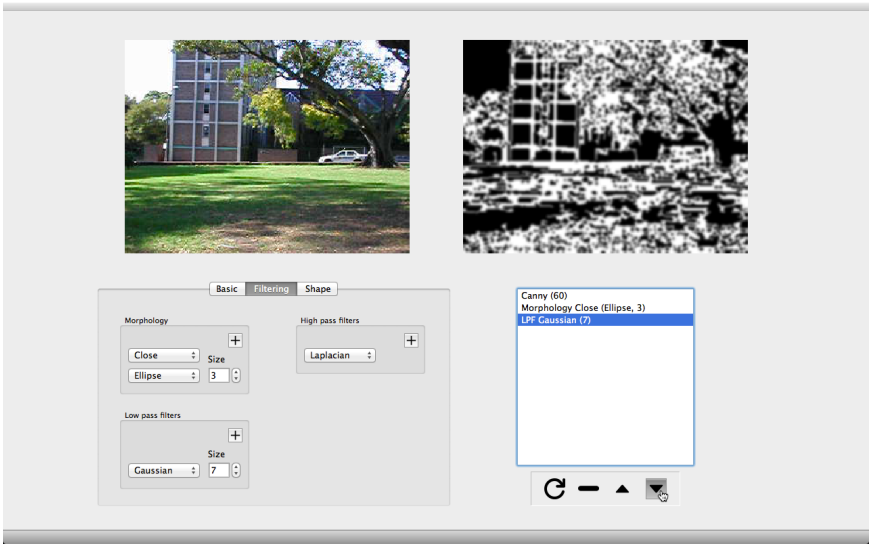



Figure 3.7: Move process down

- Remove process: the removal of the Canny edge detection process is apparent in figure 3.8, as the color of the image is restored from the binary map.

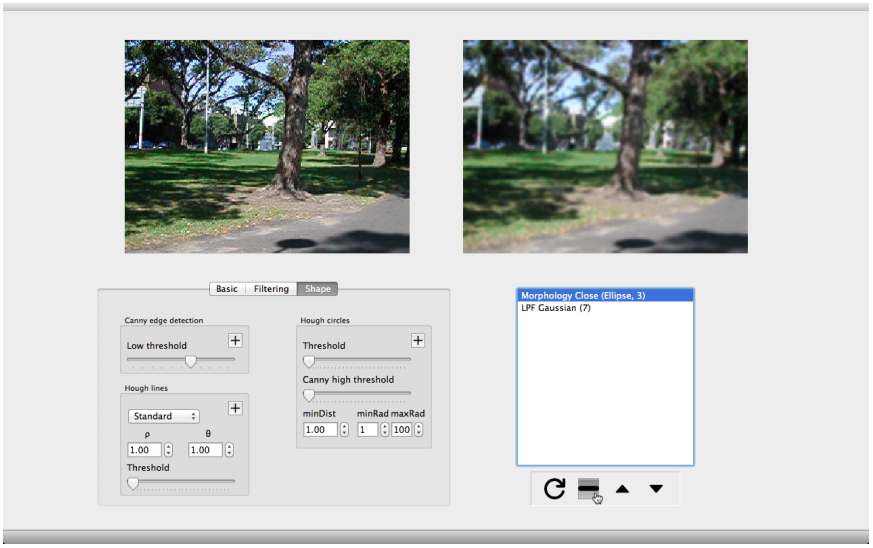



Figure 3.8: Remove process

- Refresh process: the process flow is cleared in figure 3.9, thus the output mirrors the input.

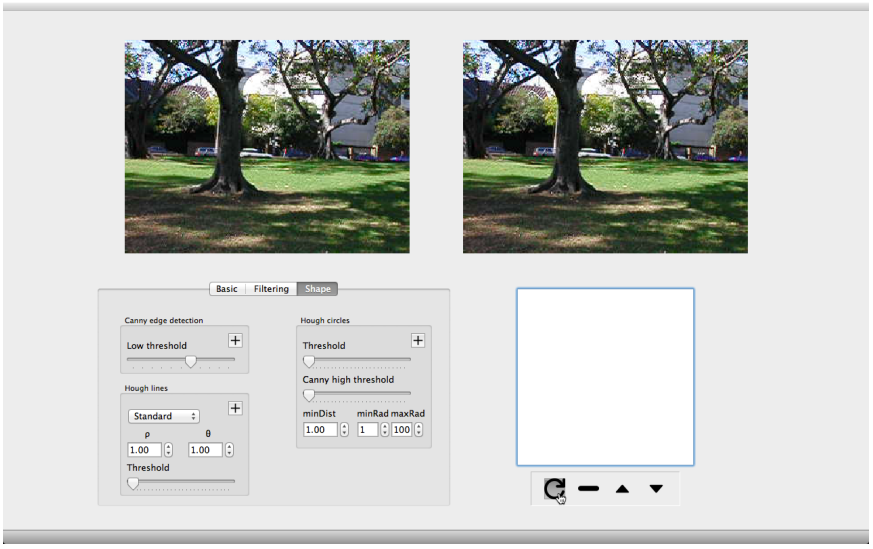



Figure 3.9: Refresh process flow

Some of the other notable functionalities of the application are:

- Pause/play: the video or webcam input may be paused to process the single stopped frame.
- Save: the result of processing may be saved as an image or video, for all inputs. This implies a single frame of a processed video or webcam may be saved as an image, and a sequence of processed images may be saved in a video. The frame rate of saving as a video is 1 for images, 15 for webcam, and the same as the input for videos. Additionally, the save dialog box displays the file name of the input ('livestream' for webcam) for user convenience in saving. A single format is currently supported, .png for images, and .avi for videos.

# Chapter 4

# Discussion

## 4.1 Evaluation

The implementation meets the application goals of process flow, processing image, video, and webcam input, and establishing a structure for extension. In this section, the software, user interface, and functionality are treated individually in evaluating the work and discussing features not implemented or incorporated in the interest of time.

### 4.1.1 Software structure

- String comparisons: form the basis of process addition, dynamic parameter updates, and error-handling. The rationale is that a name is inherent and unique to an algorithm, whereas generic approaches such as integer comparisons are error-prone, and do not scale as conveniently. However, the complexity and cost of the comparisons must be analyzed, relative to the processing.

### 4.1.2 User interface

- Adaptive view: the interface should adapt to the resolution of the system, and maximize proportionately to the window size. On a 15" computer running Linux, the lower section, including the process flow toolbar was truncated, as shown in figure 4.1.

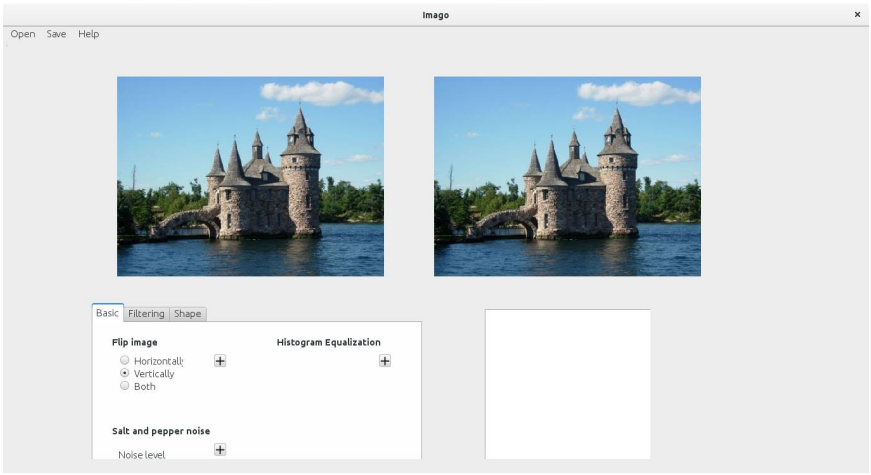
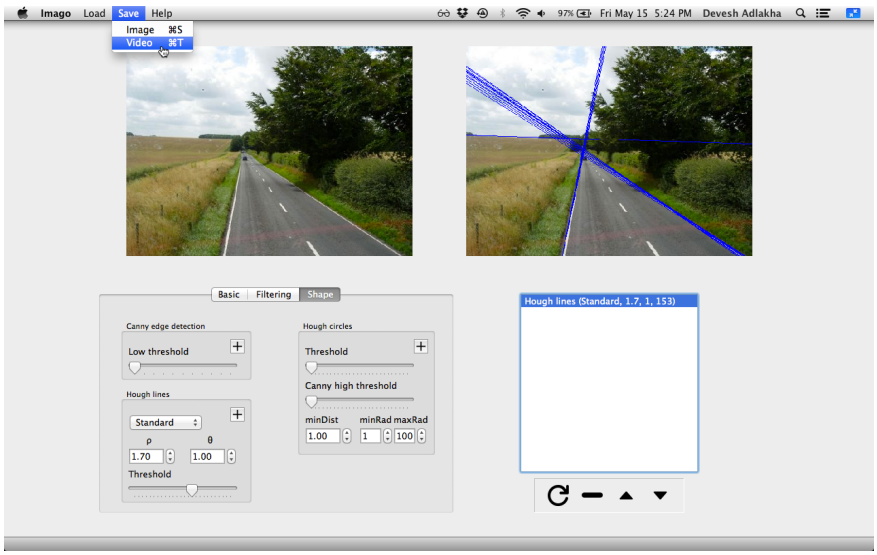
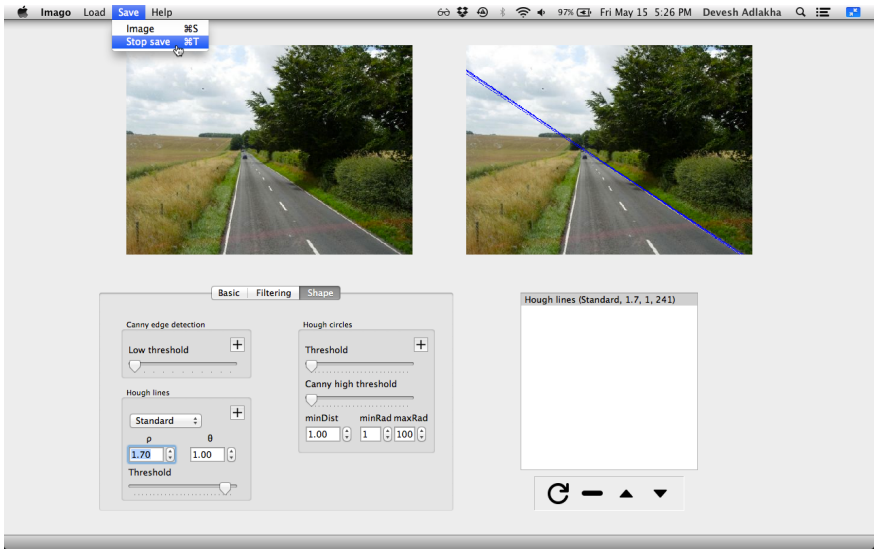



Figure 4.1: Test on 15" Linux machine: lower section missing

- Save: the need to stop the saving of a video from the menu bar is fairly non-intuitive, figure 4.2. An indication that the video is being saved could be a useful aid. Furthermore, the feature of saving an output as image or video may not be obvious.




(a) Start saving video




(b) Stop saving video

Figure 4.2: Save video: non-intuitive menu selections

- Input/output view: standard tools such as zoom and pan remain to be incorporated. The display labels could be superimposed on scroll areas[9] for panning through sliders, whereas zoom could be activated through mouse events.
- Help: The help search in the menu bar should be extended to the image processing functions in the tab structure. Additional information, such as parameter ranges of widgets could be provided through context menus or hover events.
- Accessibility: keyboard shortcuts for the tabs, and group boxes could allow complete processing from the keyboard.
- Process flow container: to hold the process flow list and toolbar, and provide additional details such as the image modality, size, processing time.

### 4.1.3 Functionality

- Additional image processing algorithms: a comprehensive toolbox of functions was not an objective for this release, and nine algorithms are provided as a working application. The following algorithms have already been tested, and ought to be integrated:

  - Segmentation: watershed, ROI.
  - Shape characteristics: bounding box, minimum enclosing circle.
  - Features: SIFT, SURF, FAST.
  - Add logo.
- Image histograms are notably missing due to two reasons:
  - Histograms do not fit in the process flow, and are meant for visualization in the display. A potential solution is to switch views among the image and its histogram, through a docking mechanism.
  - OpenCV is inferior to other image processing libraries in its visualization of a histogram, as depicted in figure 4.3. A modification to include transparency, axes values could be considered.

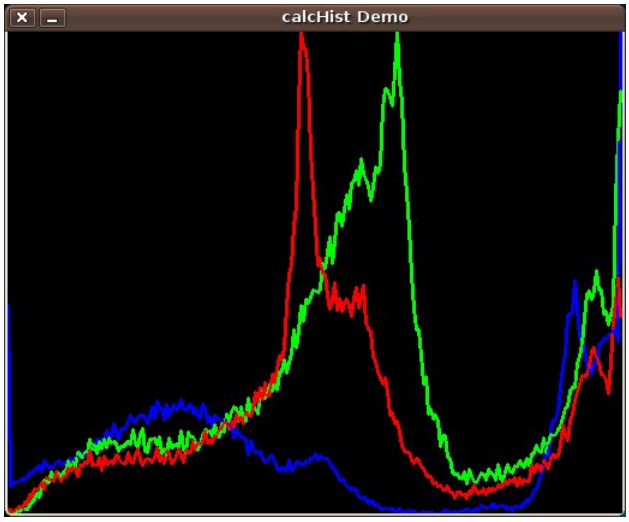



Figure 4.3: Histogram in OpenCV[10]

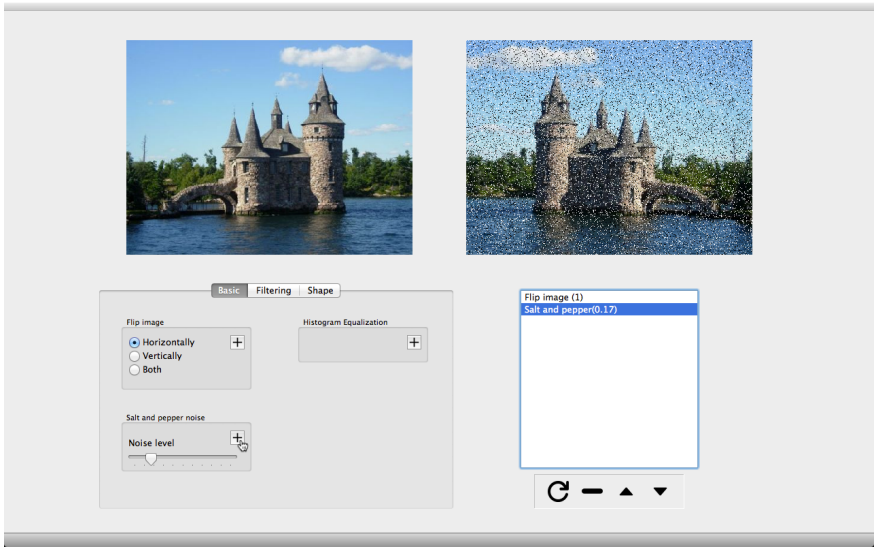
- Save video: the image modality must remain consistent in saving a video, thereby the binary output of the Canny edge detection is a potential source of error. As a quick fix, only color frames are recorded, and a more general solution is necessary.

## 4.2 Future work

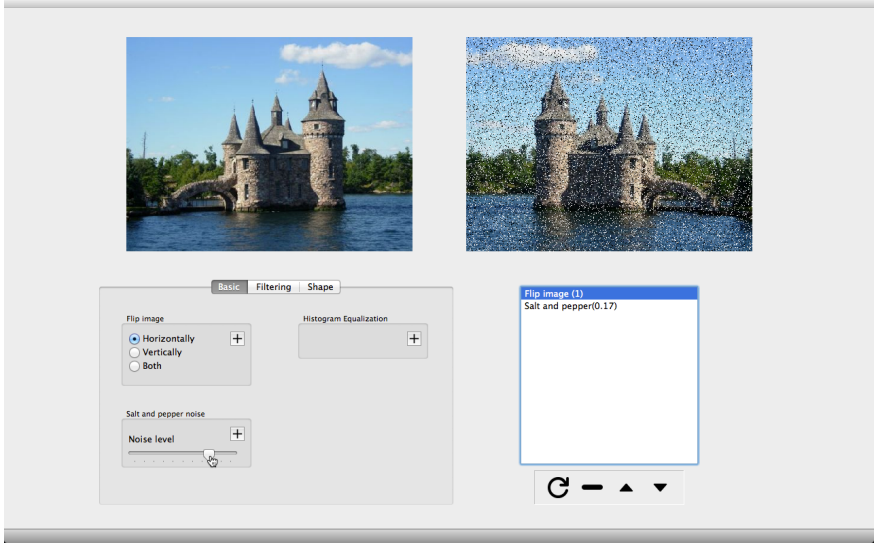
The following ideas, again distinguished as software, user interface, and functionality, require greater consideration to incorporate in the application.

### 4.2.1 Software structure

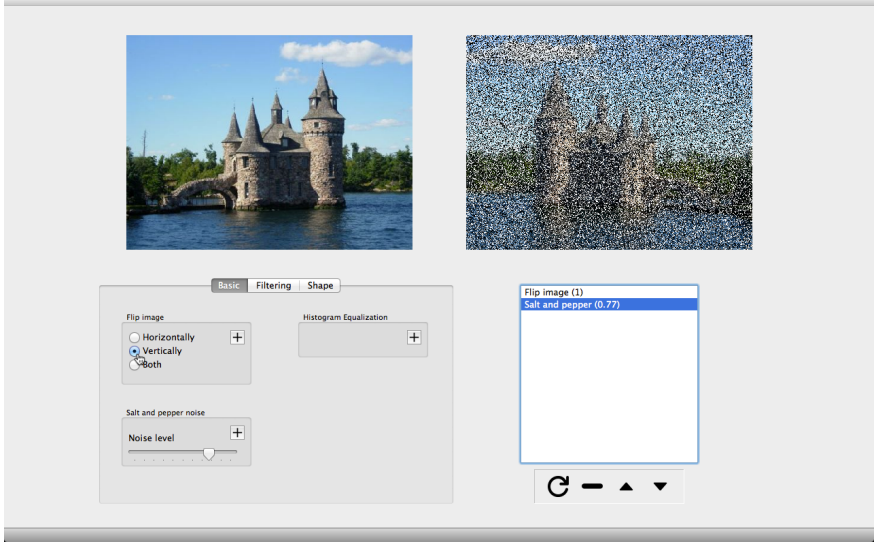
- Acessors: are required to display the parameters of a selected process in its corresponding widgets. As with the mutators, an approach other than dynamic casting is desirable. A known issue due to the lack of accessors is unexpected parameter updates, depicted in figure 4.4. In 4.4b, the noise level of the salt and pepper function is changed while its instance is not selected. In 4.4c, as the salt and pepper instance is selected, changing any parameter widget modifies its parameter to the widget value. Accessors would prevent this situation, as the selection of a process would reset the parameter widgets to reflect its parameters.




(a) Instance added




(b) Modification of process parameters while not selected




(c) Unexpected parameter update when any parameter is changed

Figure 4.4: Known issue: unexpected parameter update

- Algorithm class inheritance: algorithms such as low pass filtering, morphology appear suitable for a further inheritance scheme. The viability and benefits of such a class structure must be analyzed.
- Algorithm member data: the default values and ranges of a function are set in the user interface, whereas they should innately be encapsulated as member data for each algorithm class.

### 4.2.2 User interface

- Dynamic view: the standard view in such applications is a static input and output display. However, viewing the results from each operation is of particular interest in their analysis. A dynamic view could allow the navigation through the results of the process flow.

### 4.2.3 Functionality

- Save/load process flow: reproducing operations and results is essential in Science, thus the requirement of such a feature. A strategy could involve storing the instance names and their parameter values in a table, and parsing them while loading.
- Move/remove group of process in process flow: scaling the current feature of a single process being moved.
- Efficient processing: processing video and webcam input with complex, demanding algorithms could lead to a lag, depending on the system. Parallel processing and multi-threading are to be investigated to this end.

# Chapter 5

# Conclusion

Imago 2.0 accomplishes the established goals as a process flow-based image processing application. Image, video and webcam input may be processed, and the results saved as an image or video, independenlty of the input. The application is in its nascent stages, with a subsequent update required to address the identified issues. The structure is desinged to foster the addition of other image processing algorithms, which fit the process flow. Ideas for further work have been outlined, which should contribute to its utility. The aspiration is to be a useful aid in image processing, finding wide usage and application. The usefulness or potential usefulness of the application in practice is to be determined through feedback. As an open source project, feedback from the community would be valuable in assessing and maintaining the work.

The development of this project has been highly enriching. Many concepts from class were applicable in the program, and others were apprehended as different problems were encountered in teh development. Working individually provided an oppportunity to judge my skills and progress in C++, and the support throughout the course as well as from colleasgues has been instrumental in the completion of this project.

# References


[1] *MATLAB Image Processing Toolbox* http://in.mathworks.com/products/image/

[2] *Python Numpy* http://www.numpy.org

[3] *Python Matplotlib* http://matplotlib.org/index.html

[4] *OpenCV* http://opencv.org

[5] Laganiére, Robert *OpenCV 2 Computer Vision Application Programming Cookbook*, May 2011.

[6] Osman, Marwan *VPProject* https://sites.google.com/site/mscvfive/marwanosman

[7] Alkan, Ozan *VPOpenCV Project* https://sites.google.com/site/mscvfive/emreozanalkan

[8] Pant, Abinash *Tasbir* https://sites.google.com/site/mscvfive/abinashpant

[9] *Qt Image Viewer Example* http://doc.qt.digia.com/4.6/widgets-imageviewer.html

[10] *OpenCV documentation* http://docs.opencv.org/doc/tutorials/imgproc/histograms/histogram_calculation/histogram_calculation.html

[11] *Face recognition* http://wiki.ros.org/face_recognition

[12] *Zbar* http://wiki.ros.org/zbar_ros

[13] *Kinect aux* http://wiki.ros.org/kinect_aux

[14] *Hector mapping* http://wiki.ros.org/hector_mapping

# Appendix A

# Project Management

## A.1 Tools

The following tools aided in project management:

- GitHub: for version control and to maintain the application as open-source.
- Smartsheet: organize work schedule through Gantt charts.
- Trello: lists to organize and monitor progress, and to note ideas. Figure shows the interface with the lists maintained in the early stages of development.
- Paper and pen: although Trello was used extensively in a concurrent Robotics project, I prefer using paper and pen in programming. Figure shows the sketches of the GUI design, lists of desired functionality, and the steps to add a new algorithm class, among others used throughout the project.

The project requirements included a bi-weekly progress report, which provided a means of reflection and an objective measure of progress. Three progress reports were submitted, summarized below:

| | | |
|---|---|---|
| 02/03 | 16/03 | presented three potential project ideas. |
| 17/03 | 02/04 | selected Imago, unsuccessful attempts to integrate the Point Cloud Library (PCL). |
| 03/04 | 16/04 | tested OpenCV functionality and design strategy. |

## A.2 Development

As the primary motivation for this project, the design objectives and desired application capabilities were clear and established promptly. The structure of the software was considered critical, thus a majority of the time was spent in its design.

The development of the project could be divided in three stages, detailed below and summarized in the Gantt chart of figure:

1. Pre-implementation
   - Exploring other libraries (PCL).
   - Design strategy
   - Testing OpenCV functionality
2. Implementation
   - User interface integration.
   - Iteratively improving the strategy and user interface, and integrating additional algorithms.
   - Documenation
3. Review

– Finishing touches to the program and user interface.

– Completing this report and the supporting documents of the program (user manual, readme instructions, references).

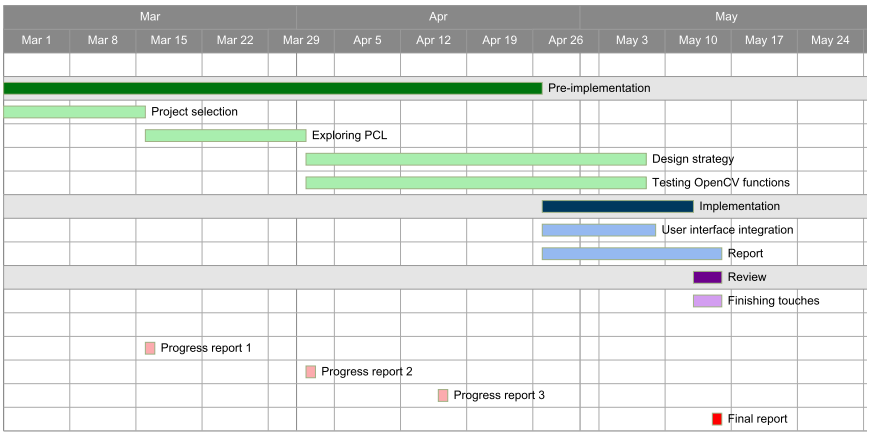



Figure A.1: GanttChart

Initially, a considerable amount of time was dedicated to integrate the Point Cloud Library (PCL) as an additional toolbox in the application. Though unsuccessful, the process of installing an external library was a useful experience. In the pre-implementation stage, a console application was maintained to test OpenCV functions, along with a graphical user interface application to assess design strategies. Prior experience in developing such a project prompted this approach of first validating the core design strategy and functionality, thereby reducing the implementation stage to managing and integrating to the user interface. As expected, however, both the design strategy and user interface required iterative modifications and improvements. For this reason, most of the documentation was completed subsequent to the implementation stage, whereby the implementation details were definite.