Marvin Image Processing Framework

Development of the Emboss Filter Plugin

prepared by

Chris Mack

prepared for

Dr. Michael Verdicchio

CSIS-602 – Software Engineering

December 7, 2011

Table of Contents

Acknowledgement: The author wishes to acknowledge and sincerely thank one of the primary Marvin Developers, Mr. Gabriel Ambrósio Archanjo for his generous support of this project and the plugin development. Gabriel provided several late-night consultations over different time zones (via Skype) to support the development of the emboss filter including: setting up the proper Eclipse IDE framework with Marvin, providing test case code examples, and assisting in the evaluation/coding of emboss algorithms applied to develop the plugin. The author looks forward to future contributions including topics explored with Mr. Archanjo including neural networks for image processing (e.g., see: http://www.cs.waikato.ac.nz/ml/weka/).

I. Project Plan

A. Project Description

This project involves the development of an image analysis plugin for the Marvin Image Processing Framework (MIPF) software package, which is currently in the beta stage of development. The MIPF provides a foundation for an image and video analysis tool framework. It is written using OO standards and documentation suitable for improving, developing new plugins, and expanding features. The video analysis tools are useful for scientific analysis of video streams to automate the analysis and extraction of changing video objects and are extensions of the static image analysis plugins. So, if an image processing filter is developed for static images, the MIPF allows for this same plugin to be applied to streaming video images in real time. The MIPF can be viewed at: http://www.marvinproject.org.

The MIPF is in the beta stage, which provides an opportunity to test, develop documentation, develop fixes, and provide new plugins or expanded capabilities of existing ones. Anyone can add new features to Marvin via a plugin interface. This project proposes to support the MIPF in this regard and apply an approach consistent with Software Engineering development practices. In consultation with one of the Marvin developers, a new plugin will be developed for the MIPF.

B. Selection Criteria

The following summarizes the selection criteria of the MIPF for this research project.

- Operating System/platform: Windows, Linux, & MacOS (w/JVM).
- Developers: Danilo Muñoz,
  Fabio Andrijauskas (http://andrijauskas.com.br/)
  Gabriel Ambrósio Archanjo (http://garchanjo.com).
- Development State: Beta.
- Programming Language: Java.
- Collaboration in lists/forums: Facebook, Twitter, and Youtube Channel (has video demonstrations of image analysis tools), and Google Groups: http://groups.google.com/group/marvin-project
- Project Plan: Proposed addition or Bug Fix.
- References:
- Archanjo, G.A., Andrijauskas, F. and Muñoz, D.R. "Marvin - A Tool for Image Processing Algorithm Development". Technical Posters of the XXI Brazilian Symposium on Computer Graphics and Image Processing, October 12- 15, Brazil, 2008. link: http://www.gpec.ucdb.br/sibgrapi2008/posters/47853.pdf

C. Selection Criteria

The proposed schedule and milestones for the project plan are shown in Table I.1 and Figure I.1 (Gantt Chart).

Table I.1 – Proposed schedule for MIPF development and testing.

ID	Task Name	Duration	Start	Finish	Pred
1	**Marvin Image Processing Framewo**	**64 days**	**Wed 9/7/11**	**Mon 12/5/11**	
2	Project Plan	6 days	Wed 9/7/11	Wed 9/14/11	
3	Concept of Operations Documen	7 days	Thu 9/15/11	Fri 9/23/11	2
4	Software Requirements Documer	7 days	Mon 9/26/11	Tue 10/4/11	3
5	Test Plan	7 days	Wed 10/5/11	Thu 10/13/11	4
6	Test Report	10 days	Fri 10/14/11	Thu 10/27/11	5
7	Software Architecture Document	10 days	Fri 10/28/11	Thu 11/10/11	6
8	Design Document	10 days	Fri 11/11/11	Thu 11/24/11	7
9	A submitted code patch that fixes	7 days	Fri 11/25/11	Mon 12/5/11	8

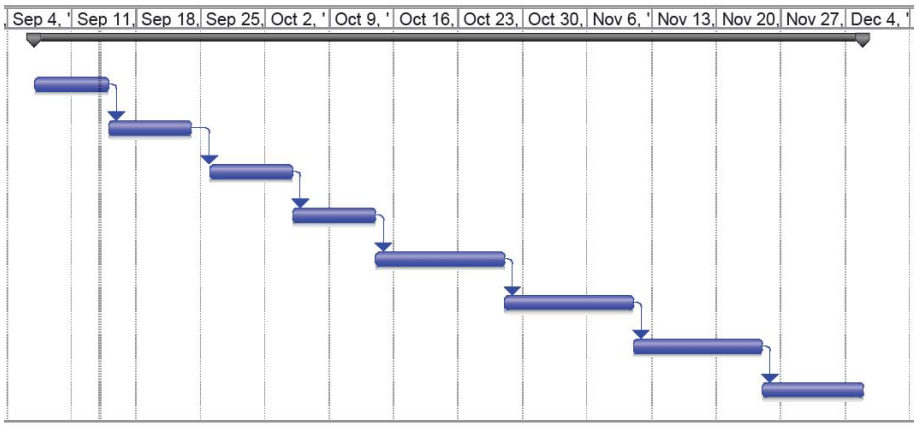

Figure I.1 – Proposed GANTT chart for MIPF development and testing (corresponds with Table 1).

II. Concept of Operations

A. Introduction and Background

Marvin Image Processing Framework (MIPF) is an open source image processing framework developed in Java. It provides various image analysis and processing plugins that run in the common framework. MIPF is currently in the Beta state of development and provides the following features:

- Manipulate images
- Manipulate captured video frames
- Multithreading image processing
- Integrate plugins with Graphical User Interface(GUI)
- Analyze plugin performance
- Extend features via plugins
- Unit test automation

Currently, there are 44 plugins available for a wide range of different purposes. In addition to the image analysis and processing plugins, MIPF provides an image editor called MarvinEditor. The intent of MIPF is to provide an extensible framework whereby newly developed plugins can be integrated without modifications or refactoring of the base MIPF code.

B. Problem Overview

MIPF is in the Beta stage of development. Although multiple plugins have been developed, the MIPF development team has identified several new plugins that need to be developed and added to the MIPF plugin library, which include:

Image processing:

- Emboss filter
- Glow filter
- Color histogram equalization
- Color restoration
- Haar-like features extraction
- Scale-invariant feature transform
- Structural similarity (SSIM)

Machine learning:

- K-nearest neighbors
- K-induction of decision trees
- Neural networks
- Genetic classifier

C. Goals and Outcomes

Marvin is an open source project and the main goal as noted by its developers is "to help society to bring image processing benefits to real world applications". In order to justify the efforts on a new image processing framework, MIPF developers are proposing the easiest and most extendable one. Marvin plugins are created to be used in third-party applications, MarvinEditor, video editors, web applications and so forth. Therefore, plugins are the core of our MIPF idea. The goal of this effort will focus on the development of a plugin for image analysis. The plugin proposed for development is the Emboss Filter.

D. Expected Difficulties

The development challenges are twofold. First, for the new plugin, an emboss algorithm needs to be developed. Several sources are available that can be used as either pseudocode or a "go-by" for developing the emboss filter. The second challenge will be developing a plugin that meets the integration requirements of the MIPF. Although the intent of MIPF is to create a seamless integration platform, for development purposes, some slight modifications of the MIPF framework will be required to implement new plugins.

Currently, as has been observed, the addition of plugins is not fully abstracted nor does it subscribe to a Design Pattern neatly. There are some "hard fixes" to code that have to be made in other classes in order to get a plugin to run properly in the MIPF development environment. This includes a need to update all ".jar" libraries (i.e., folders) with updated plugins. Opportunities to make the MIPF more loosely connected will be identified during the course of the plugin development.

III. Software Requirements Specification (SRS)

A. Introduction

1. Product Overview

Marvin Image Processing Framework (MIPF) is an open source image processing framework developed in Java. It provides various image analysis and processing plugins that run in the common framework. MIPF provides the following features:

- Manipulate images
- Manipulate captured video frames
- Multithreading image processing
- Integrate plugins with Graphical User Interface(GUI)
- Analyze plugin performance
- Extend features via plugins
- Unit test automation

Currently, there are 44 plugins available for a wide range of different purposes. MIPF is currently in the Beta state of development. Although multiple plugins have been developed, the MIPF development team has identified several new plugins that need to be developed. The following requirements focus on the development of one plugin to be developed, the Emboss Filter (EF).

2. Purpose of the Requirements Document

The purpose of the requirements document is to specify the user requirements and the system requirements for the EF plugin for MIPF. The user requirements will specify how the EF will be used inside MIPF and what options will be available to adjust the elements or characteristics of embossing digital images. The system requirements will specify MIPF system requirements to integrate the EF plugin within the MIPF.

3. Scope of the Project

The scope of this project will involve developing an EF plugin for MIPF. It will involve several steps that will conclude with the addition of the EF plugin to the MIPF plugin library. These steps will include:

- Research of EF algorithms.
- Selection of EF algorithm for the plugin.
- Development of Java code for the EF plugin.
- Testing of the EF plugin in MIPF.
- Final integration of EF with MIPF.

4. References

Friesen, J. 2005. Java Tech: Image Embossing
http://today.java.net/pub/a/today/2005/12/08/image-embossing.html

Java World's Daily Brew. 2009. Introducing an emboss effect to JavaFX
http://www.javaworld.com/community/node/2854

BufferedImage Emboss. 2009
http://www.java2s.com/Code/JavaAPI/java.awt.image/BufferImageemboss.htm

5. Definitions and Abbreviations

Emboss - to raise or represent (surface designs) in relief.
MIPF – Marvin Image Processing Framework.
ME – Marvin Editor.
Plugin - An accessory software or hardware package that is used in conjunction with an existing application or device to extend its capabilities or provide additional functions.

B. Overall Description

1. Product Perspective

The EF plugin will operate in the MIPF much like the other plugins. Images will be imported into the MIPF viewer (either the Editor or TestPlugin) and "embossed" with the EF plugin. Users will have an option to reset the to return to the original image. Or, users can keep the embossed image and save it.

2. Product Functions

The EF plugin will have several options to vary the degree of embossing; however, for this development, they will be functionless due to developer preferences for the moment. These include: Azimuth, Elevation, Depth. The Azimuth controls the angle of the light source to the direction of embossing (for example: from the N, NE, E, SE, S SW, W, NW) by moving the light source and its orientation to an embossed pixel. The elevation will control the height of light source to the image surface. The Depth will control the exaggeration of pixel values. Figure III.1 shows an example of the EF tool on an image (projected behavior).

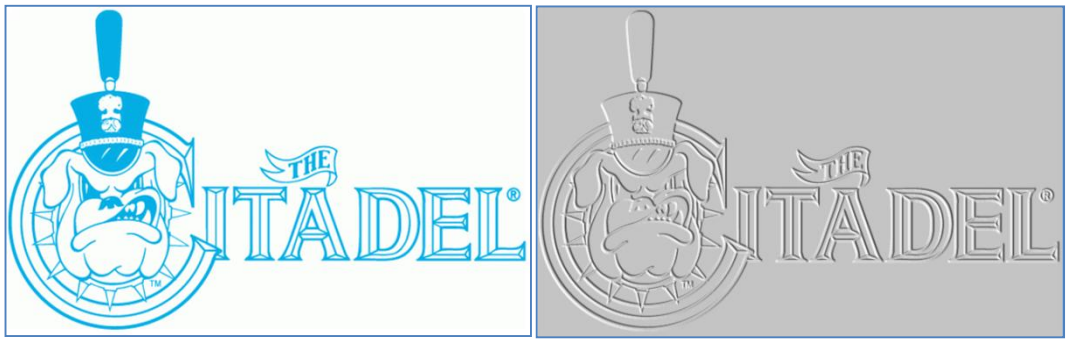

Figure III.1 – Original and "embossed" image.

3. User Characteristics

   Users of MIPF are interested in image processing, editing, and refinishing. They prefer a wide range of image filtering tools also for creating custom "looks" of original images. The EF plugin is another "tool in the toolbox" for users. Users prefer options to see image filters in draft mode before final filters are applied to the original or have an option to reset to the original image.

4. General Constraints

   The EF will be used to process static images.

5. Assumptions and Dependencies

   Original images will be common file types including JPG, TIF, and PNG.

C. Specific Requirements

1. Functional Requirements

   The user will be able to select the EF function from the main menu, under the Filter menu item. When selected, the image shall be embossed (a temporary copy). The user will have an option to save the embossed image or reset to the original.

   a. General Requirements

      Within the Marvin framework, users have several options to apply the Emboss filter. Developers can use the TestPlugin for testing plugins, which will assist with iterative developments of the EF plugin. The MarvinFramework functions more like COTS whereby users can select any of the plugins from drop-down menus. Requirements include:

      i. Developers will use the TestPlugin to develop and test the EF plugin.
      ii. General users will run the EF plugin using the MarvinEditor, which represents a COTS implementation of the Marvin software.

2. Non-Functional Requirements

   The MIPF is can be developed in the Eclipse IDE if installation instructions are followed (see: http://www.marvinproject.org)

   a. General Requirements

      i. REQ-1 - Code for the EF shall be written in Java.
      ii. REQ-2 – The MIPF framework shall run on any JVM, version 1.6 and higher.

D. Other Requirements

If running inside the Eclipse IDE, specific library references must be set up correctly in order to function properly. Developers should follow Marvin setup tutorial to configure Eclipse for development purposes.

IV. Test Plan

A. Test Plan

The test plan for the Marvin Framework Emboss filter will consist of testing an image embedded with the Marvin framework called "toucano". This image represents good variation in shapes and colors to test the emboss filter. It will also include a visual inspection of the code and bug tracing. In addition, three idealized images will be tested to evaluate the EF plugin with variations in colors, geometric shapes, and lines.

1. Image Test - Check/Comparison
   The test images will be embossed using the EF filter. They will be reviewed for consistency of the expected "embossed" look. The visual inspection will evaluate if the light position (i.e., default position of top left corner of image) was appropriately applied on the embossed image edges and surfaces.
   .
2. EF Code Testing Process
   The code will be inspected visually for consistency with Marvin Framework code interface requirements. Also, a debugging trace will be used within Ecclipse to trace threads, states, and the flow of control as it is stepped into/from the Marvin Framework.

B. Requirement List

1. Image Processing
   A typical image like "toucano" can be processed by the EF plugin without crashing the system. The same requirement applies for the idealized cases.
2. Expected Results
   The test "toucano" image should be embossed correctly. The idealized cases should be embossed correctly, but the emphasis is also on variation of colors, shapes, and line weights within images.
3. Pass/Fail Criteria
   Pass – Images embossed correctly with correct edge lighting and shadowing.
   Fail – Images embossed incorrectly, did not emboss, or MIPF failed.

C. Tested Items

1. Tested Items
   The "toucano" image with variable edge features and colors will be tested with the EF plugin. In addition, three idealized cases will be tested.

2. Testing Schedule
   The "toucano" and idealized images will be tested with a mask to compare the original with the embossed version. The testing will be completed within 1 day.
3. Test Recording Procedures
   Original images will be embossed with a mask that is smaller than the embossed image. In addition to being a useful future feature of the EF plugin, this will be useful for a check/comparison of results along edges and surfaces to compare embossed versus original areas of the image.
4. Hardware and Software Requirements for implementing tests (be specific)
   There are no known hardware limitations; however, a machine with a video card would be useful if large volume batch image processing were to be done using the Marvin EF plugin.
5. Constraints to the Testing Process
   Images are going to be tested manually. For more thorough testing, Marvin would have to be modified to run in batch mode and several thousand images would have to be tested. Results would have to be analyzed using automated pixilation comparison techniques to verify that the EF function performed the bitwise RGB shift required to generate the emboss look.

V. Test Report

A test was done to evaluate the Emboss filter, which consisted of evaluating the function on a representative image and three idealized images. The default Marvin "toucano" image was used for the test and provided a broad range of color spectrums and contrast (i.e., black versus white) to test the embossing functionality. An image mask was included to provide an ability to visually inspect along image boundaries and compare if image features were embossed correctly (a "quasi-" before and after comparison). As shown in Figures IV.1 through Figure IV.6, the tests were completed and the EF passed based on expected emboss effects. However, some edge effects were observed, which may be attributed to the original image resolutions. Table IV.1 summarizes the test procedure.

A bug trace was used at the inception of the Emboss Filter. Results indicate that the entry into and exit from the EF was correct and no errors were reported. Further bug tracing in Eclipse while stepping into the EF showed it behaved as expected and processed the "toucan" image correctly.

Table IV.1 – Test form of EF in the Marvin framework.

Test Case					
Requirement Number	C.1		Status:		Complete
Description	Emboss filter should emboss digital images (real and idealized cases).				
Conditions	Test conducted on Windows XP, JVM, version 1.6, on Ecclipse IDE.				
Pass Criteria	Images were embossed and compared with other embossing tools for verification.				
Test Information					
Name of Tester:	Chris Mack		Date:		26-Nov-11
Version:	Beta		Time:		2100
Test Results					
Action	Expected Results	Pass	Fail	N/A	Comments
Executed EF on (5) images	Images embossed.	X			Image comparison verified correctness.

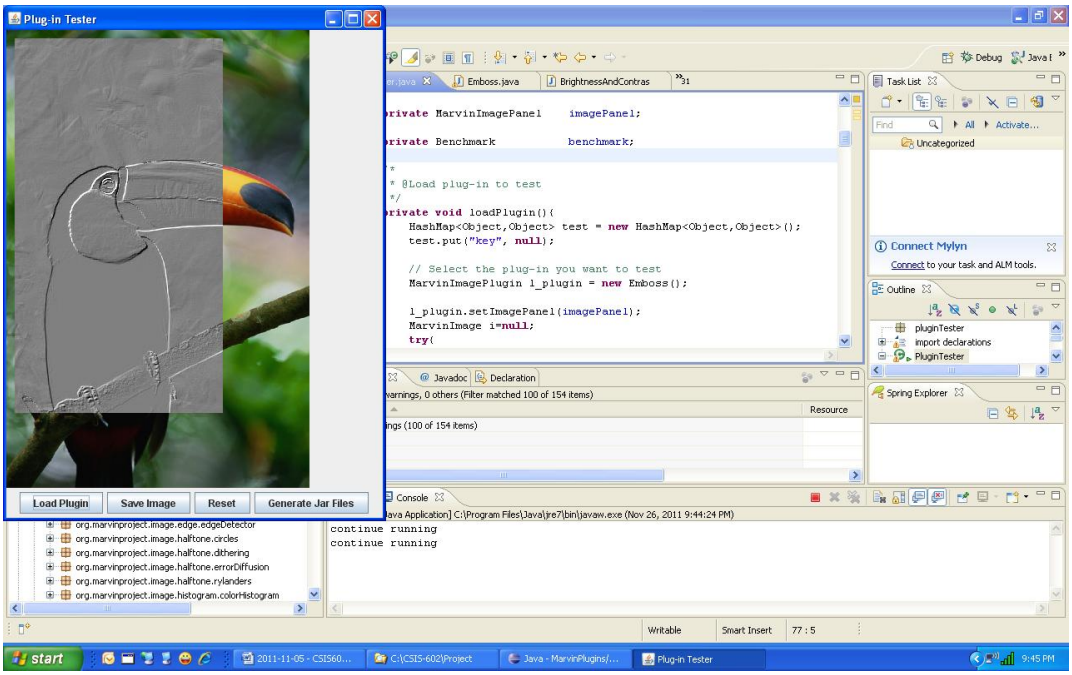

Figure IV.1 – Runtime test results within Ecclipse with the Marvin EF and the "toucan" image.

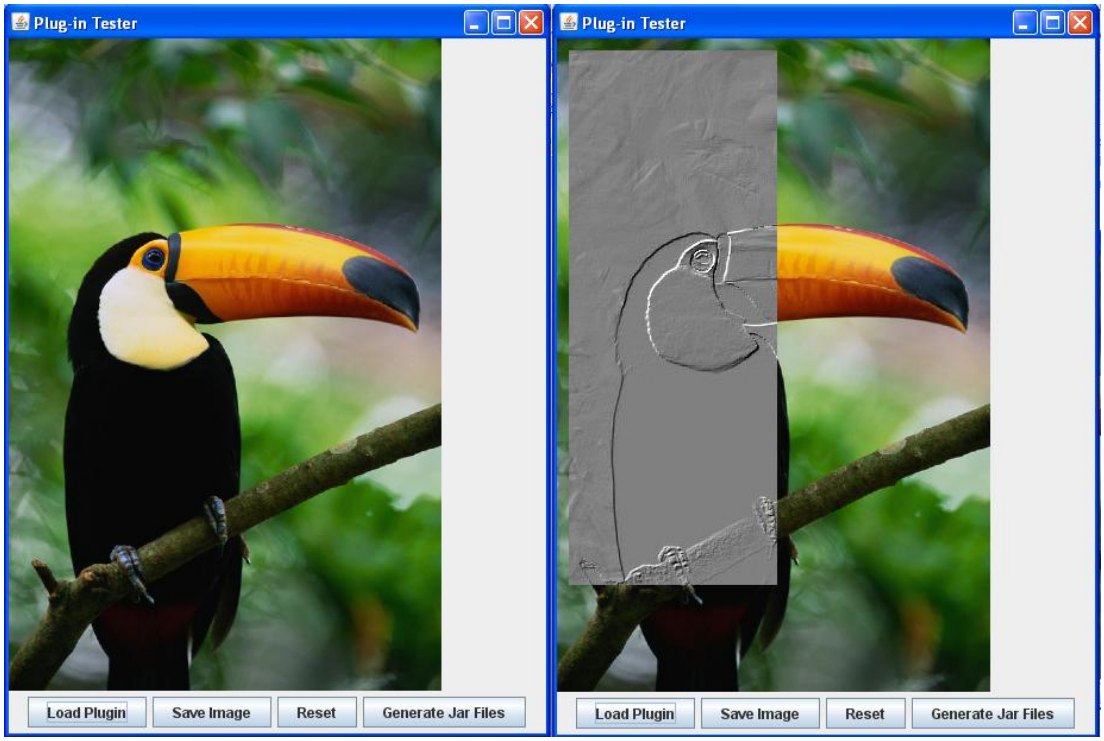

Figure IV.2 – Before and after embossing (using EF plugin) test results on the "toucano" image. Note, the "illumination" is from the top left edge and edges appear to mimic the appropriate brightening and shadowing edge effects. Also, the greyscaling of colors appears correct.

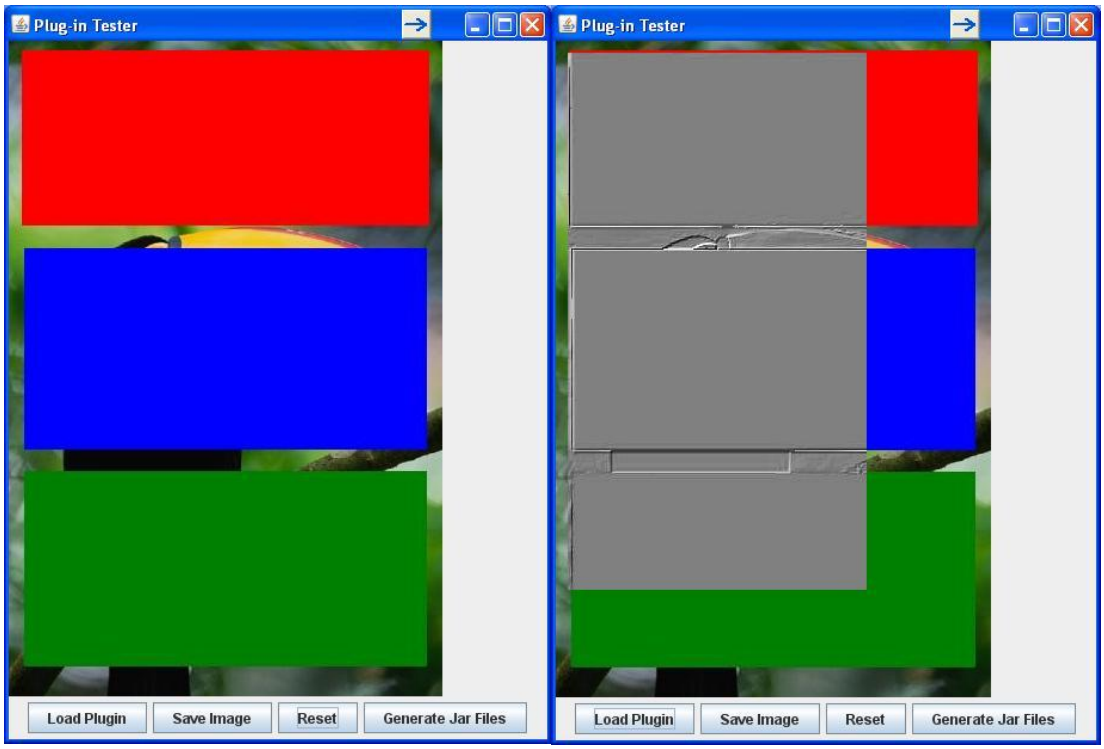

Figure IV.3 – Before and after embossing (using EF plugin) test results on the idealized case No. 1 image (i.e., testing color variations). Note the edge effects on red and green colors (possibly caused by image resolution).

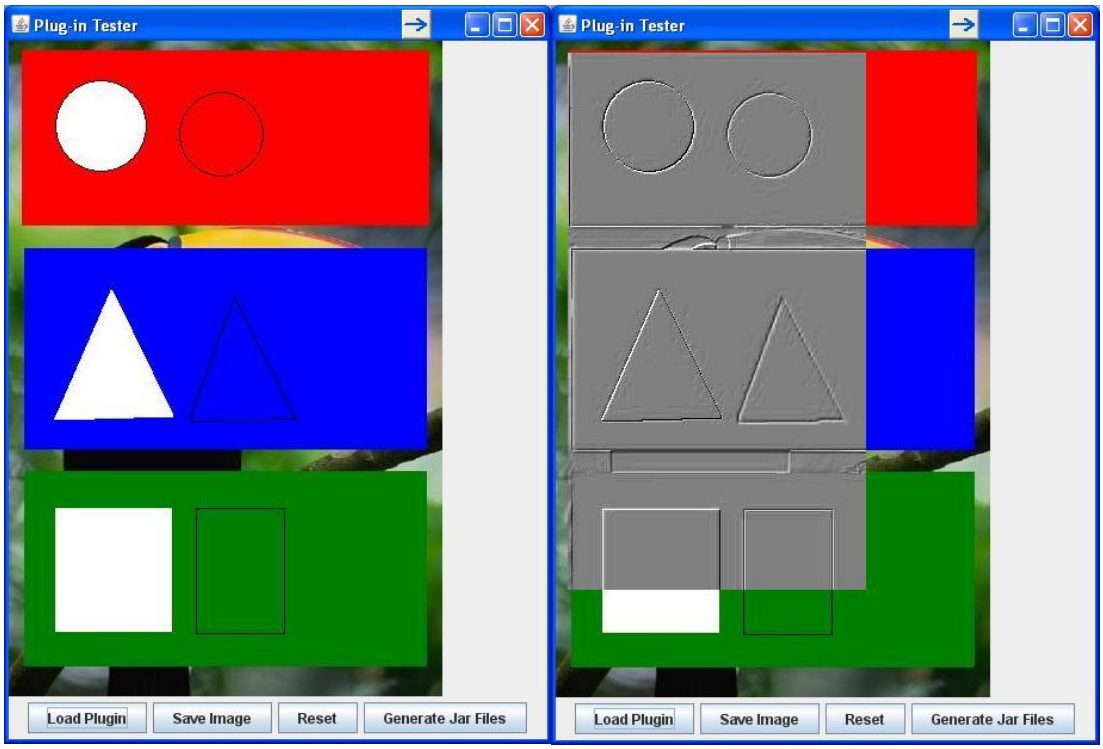

Figure IV.4 – Before and after embossing (using EF plugin) test results on the idealized case No. 2 image (i.e., testing geometric shapes and line weights). Note edge effects on green (w/black lines).

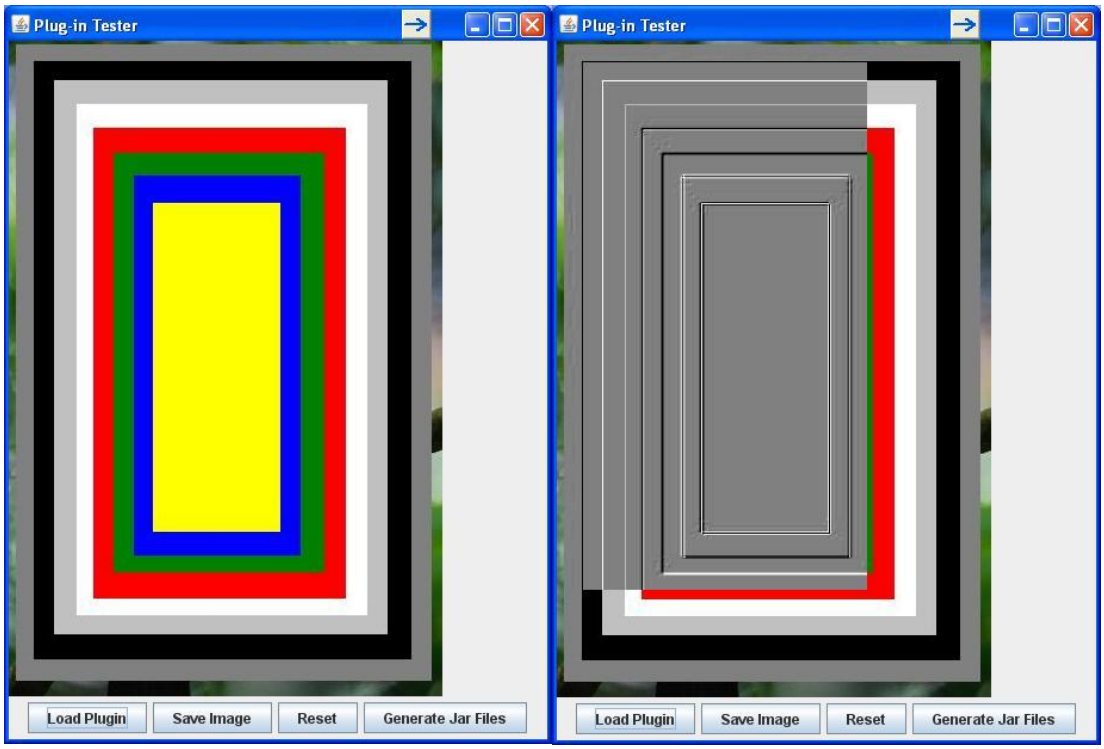

Figure IV.5 – Before and after embossing (using EF plugin) test results on the idealized case No. 3 image (i.e., testing graduated color variations). Note the variability in depth and some edge effects.

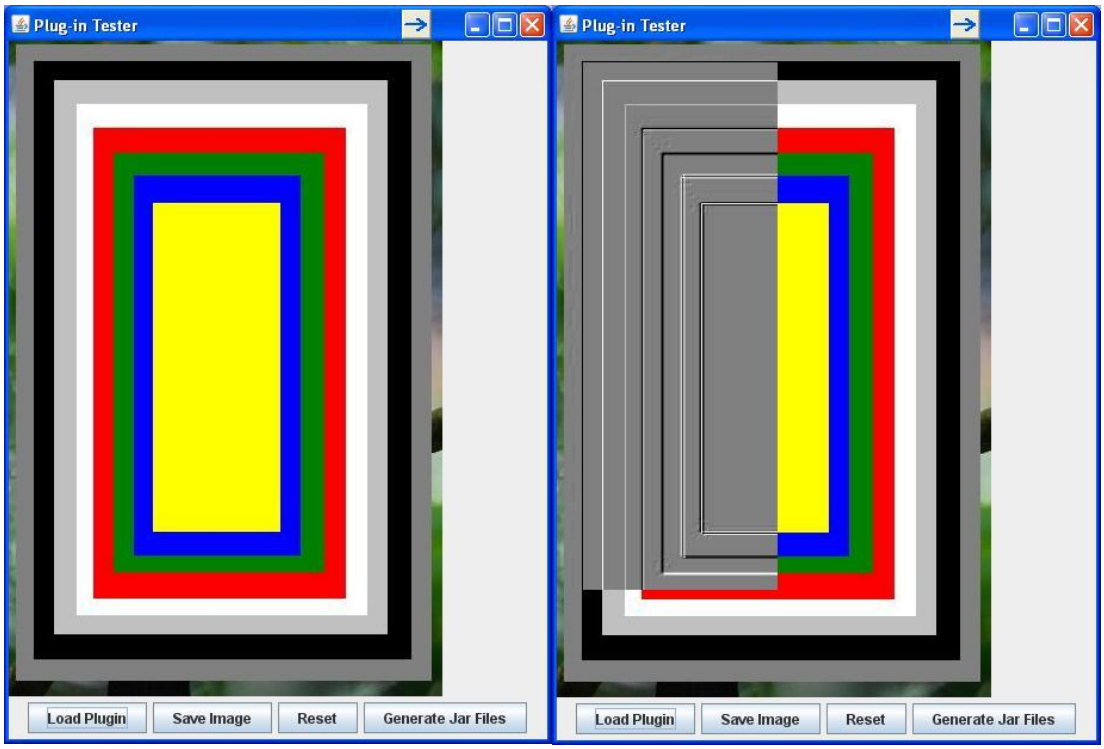

Figure IV.6 – Before and after embossing (using EF plugin) test results on the idealized case No. 4 image (same image as No. 3, but mask splits near center). Note the variation of edges and depths of embossment.

VI. Software Architecture

A. Marvin EF Framework – Class Diagram

Marvin uses several Java SWING classes to create the Marvin Framework upon which new plugins can be developed and added. Figure VI.1 depicts the general class diagram of the EF plugin and how it is implemented within the MIPF.

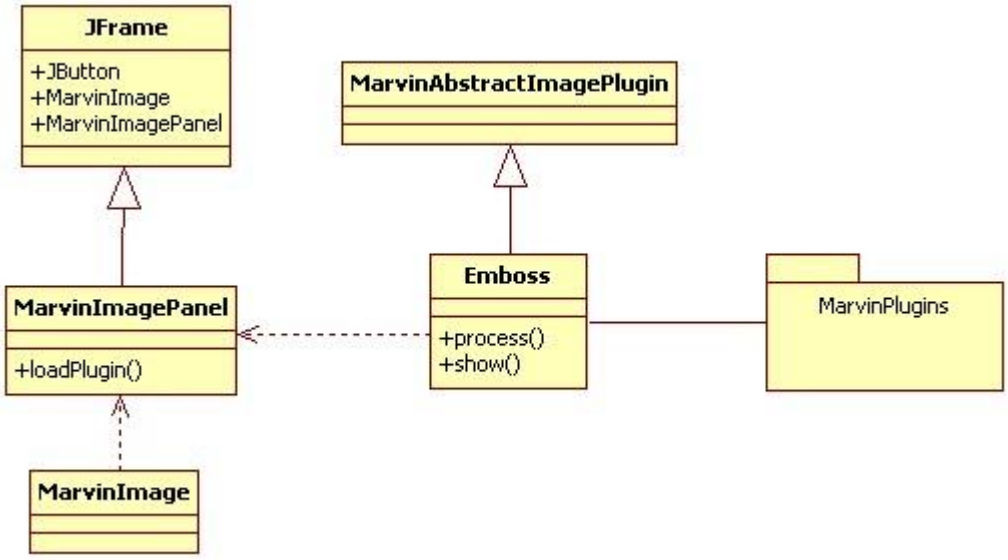

Figure 1 - UML Diagram of the Marvin Plugin Framework and Emboss filter

Marvin extends the JFrame class and adds components to a JPanel similar to the Decorator Design Pattern. Components such as the menu items (to execute plugins), buttons, and button listeners are also added. When an event is triggered (i.e., selection of a plugin tool), an EventListener passes this on to the appropriate plugin for processing an image.

The Emboss filter extends the MarvinAbstractImagePlugin, which implements the MarvinPlugin interface. The EF processes the image by applying the emboss algorithm, then calls a "show" method to display it in the Marvin image panel.

B. Developer's and User's Framework – Use Case Diagram

To add a new plugin to Marvin, developers develop the plugin as a separate class (in this case, Emboss.java), which will be exported to a ".jar" library of all Marvin plugins. The developer will be required to edit the client program (in this case the TestPlugin) to provide access to images to be processed. The developer will develop the plugin code, which handles all processing of the image explicitly and creates a new image that is rendered according to the plugin algorithm that is implemented.

Users who want to use the Marvin plugins, can access them through a client program that is modified to run all active plugins (active in the sense that they have been updated and added to the ".jar" library folder). Or in the example below used to develop the EF, users can run the plugin from a test class provided within the MarvinPlugin package called TestPlugin.

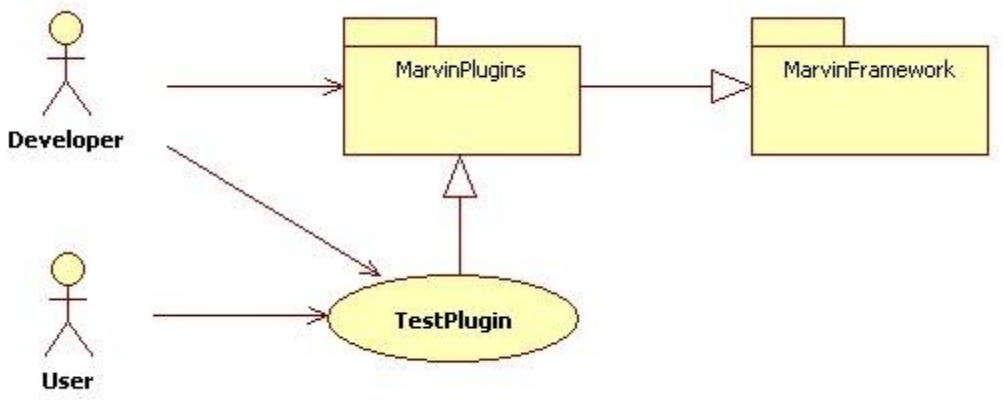

Figure 6.2 – Use case diagram of development and use environment.

C. MIPF Framework – Layer Architecture

Figure 6.3 depicts a layer architecture of the MIPF. From this perspective, there are essentially three layers of focus: Applications, Plugins, and the Framework. The framework provides classes for GUI, I/O, multithreading, and video stream capture. The Applications are specialized features that include pattern recognition, filtering applications, the MarvinEditor, and video game prototyping. The Emboss Filter will reside in the Plug-ins layer, which provides a broad spectrum of image processing functions to MIPF.

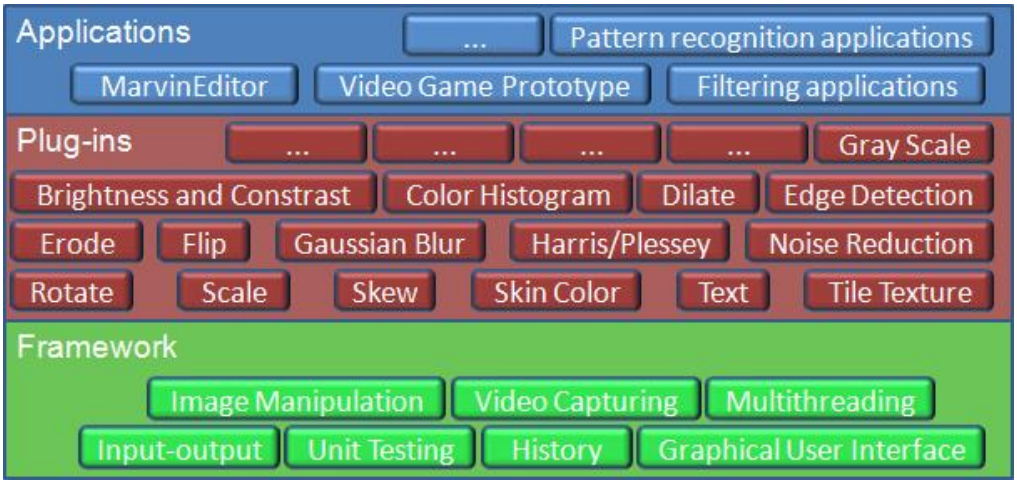

Figure 6.3 – Layer architecture of the MIPF.

VII. Software Design

A. Introduction

1. Purpose

This software design document describes the architecture and system design of the Emboss Filter (EF) plugin for the Marvin Image Processing Framework (MIPF).

2. Scope

The EF plugin allows users of the MIPF to apply an embossing filter on images. There is additional capability to apply the emboss filter to a selected region of the image if needed via a mask that is set by the left corner, width, and height of the rectangular mask.

3. Overview

This document provides the details of the Marvin EF system architecture, design algorithm, and design elements.

4. Reference Material

Friesen, J. 2005. Java Tech: Image Embossing.
http://today.java.net/pub/a/today/2005/12/08/image-embossing.html

Java World's Daily Brew. 2009. Introducing an emboss effect to JavaFX
http://www.javaworld.com/community/node/2854

BufferedImage Emboss. 2009
http://www.java2s.com/Code/JavaAPI/java.awt.image/BufferImageemboss.htm

5. Definitions and Acronyms

Emboss - to raise or represent (surface designs) in relief.
MIPF – Marvin Image Processing Framework. This is the software package upon which the Emboss Filter plugin was developed.
ME – Marvin Editor.
Plugin - An accessory software or hardware package that is used in conjunction with an existing application or device to extend its capabilities or provide additional functions.
EF – Emboss Filter. This is the plugin for MIPF developed to apply an emboss effect to images.

B. System Overview

The MIPF is designed to be a flexible image processing framework that is scalable to new plugins for image processing. The intent of MIPF is for developers to be able to focus on developing plugins that can be added almost seamlessly without editing a lot of code in the MIPF.

C. System Architecure

1. Architectural Design

The basic premise behind the architecture of the MIPF is to provide an ability to add plugins easily without having to modify the MIPF. Plugins, as long as they follow the same pattern as other plugins, should function properly. Figure VII.1 depicts the architecture design of MIPF. It consists of a repository of plugins (e.g., Emboss Filter), which are referenced by their ".jar" string name signature. The MIPF applies them to the image framework and takes care of the framework overhead like a layout container.

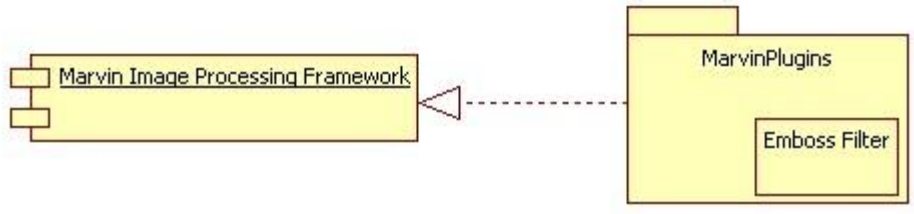

Figure VII.1 – General architecture of the MIPF and library of plugins (e.g., Emboss Filter).

2. Decomposition Description

Figure VII.2 depicts the class structure of the Emboss Filter. The emboss filter produces an image that is greyscale and appears to have raised elements, much like is done when physically embossing an image on a plate of metal. To create this effect digitally, uniform adjustments (or pixel calculations) of RGB color bits (or shifts in value) are made. Friesen (2005) describes the algorithm applied as follows:

> *Think of an image as mountainous terrain. Each pixel represents an elevation: brighter pixels represent higher elevations. When an imaginary light source shines down on this terrain, the uphills that face the light source are lit, whereas the downhills that face away from the light source are shaded. An embossment algorithm captures this information.*
>
> *The algorithm scans an image in the direction that a light ray is moving. For example, if the light source is located to the image's left, its light rays move from left to right, so the scan proceeds from left to right [and top-to-bottom]. During the scan, adjacent pixels (in the scan direction) are compared. The difference in intensities is represented by a specific level of gray (from black, fully shaded, to white, fully lit) in the destination image.*

...

After obtaining the red, green, and blue intensities for the current pixel and its upper-left neighbor, ...calculate the difference in each intensity. The upper-left neighbor's intensities are subtracted from the current pixel's intensities because the light ray is moving in an upper-left to lower-right direction. Identify the greatest difference (which might be negative) between the current pixel and its upper-left neighbor's three intensity differences. That's done to obtain the best-possible chiseled look. The difference is then converted to a level of gray between 0 and 255, and that gray level is stored in the destination image at the same location as the current pixel in the source image.

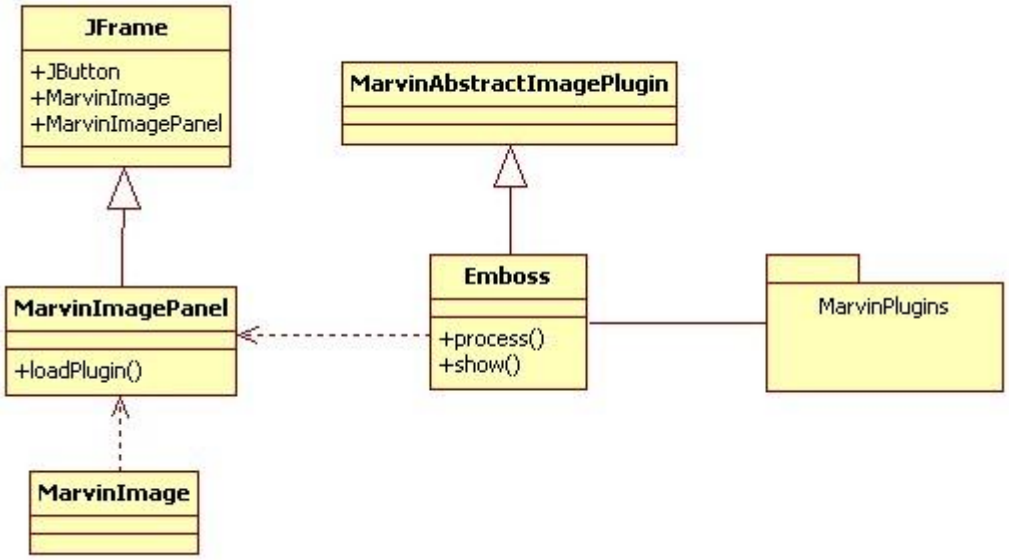

Figure VII.2 – The Emboss Filter is a java class that performs the emboss filter and "shows" it in the Marvin JPanel.

3. Design Rationale

Additional functionality is capable to add to the Emboss filter. This could include parameters such as the elevation, depth, and azimuth with respect to the light sources as depicted in Figure VII.3. The EF plugin currently does not have this functionality and was thought by the developers of MIPF to exceed their needs. However, for future potential, slide mechanisms have been embedded, but are functionless at the moment as shown in Figure VII.4.

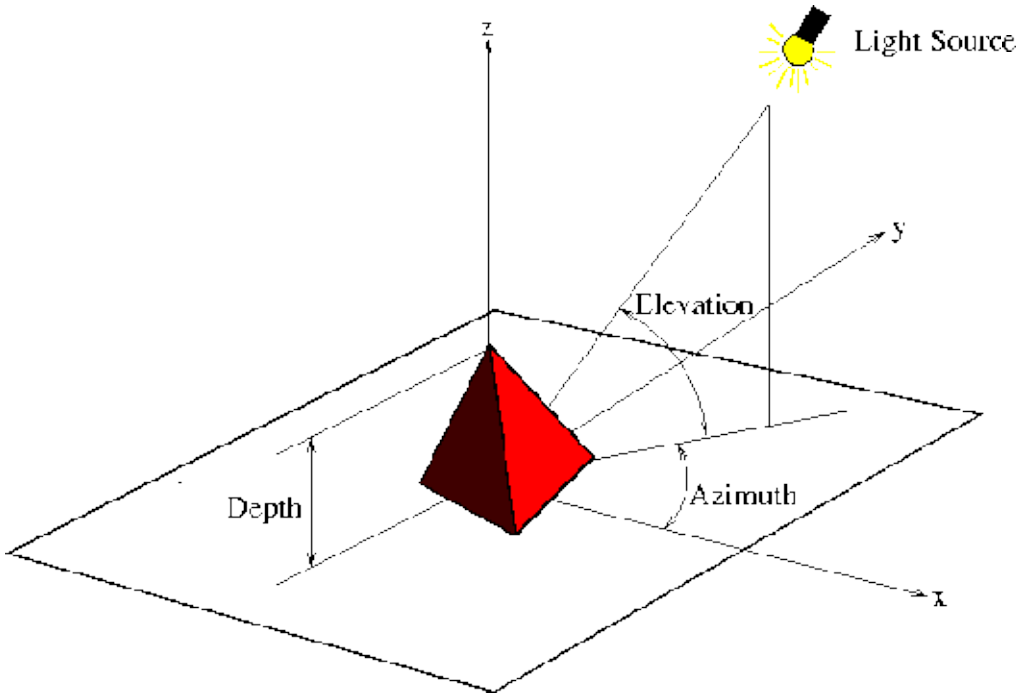

Figure VII.3 – Additional variable parameters that could be included with an emboss filter function (e.g., Depth, Elevation, and Azimuth).

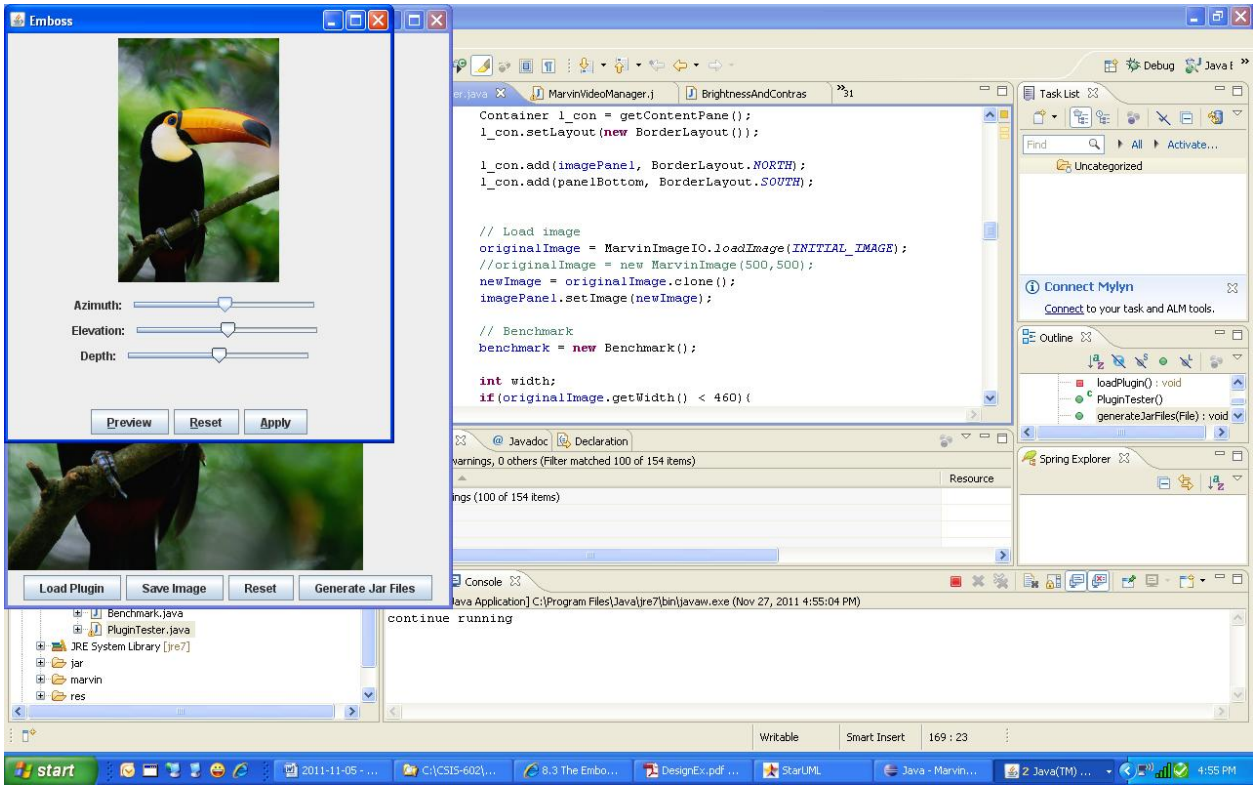

Figure VII.4 – Additional variable parameters Azimuth, Elevation, and Depth within the Emboss Filter plugin (currently functionless in the MIPF).

D. Data Design

1. Data Description

Images in MIPF can be selected from any source such as folders accessible by the OS. Images processed by MIPF can be stored in the same locations or other locations accessible by the OS. Current image formats supported include JPG, TIF, and PNG.

2. Data Dictionary

Emboss Class – This is the class that implements the emboss algorithm. It provides a process method that creates the new image from the pixel calculations. It has a show method that displays the new image that has been embossed.

E. Component Design

Friesen (2005) provides a pseudo code for an embossing algorithm. It involves a pixel value shift for each of the pixel values to replicate a fixed top-left light location for the image. Additional functionality to adjust the location of the light source (i.e., elevation, depth, and azimuth) is not yet functional in the Emboss Filter coded for Marvin in this document. The pseudo code (Friesen, 2005) is as follows:

```
FOR row = 0 TO height-1
    FOR column = 0 TO width-1
        SET current TO src.rgb [row][column]

        SET upperLeft TO 0
        IF row > 0 AND column > 0
           SET upperLeft TO
               src.rgb [row-1][column-1]

        SET redIntensityDiff TO
            red (current)-red (upperLeft)
        SET greenIntensityDiff TO
            green (current)-green (upperLeft)
        SET blueIntensityDiff TO
            blue (current)-blue (upperLeft)

        SET diff TO redIntensityDiff
        IF ABS (greenIntensitydiff) > ABS (diff)
           SET diff TO greenIntensityDiff
        IF ABS (blueIntensityDiff) > ABS (diff)
           SET diff TO blueIntensityDiff

        SET grayLevel TO
            MAX (MIN (128 + diff, 255), 0)

        SET dst.rgb [row][column] TO grayLevel
    NEXT column
NEXT row
```

F. Human Interface Design

1. Overview of User Interface

The Emboss plugin is contained at the top-level in the Filters menu and is then available via the Color submenu within the Marvin framework display tool. The menu names correspond also to the development environment with respect to plugin package and path names and provide for a systematic code logic.

2. Screen Images

As shown in Figure VII.5, the EF is available from a drop down submenu within the Marvin framework display tool.

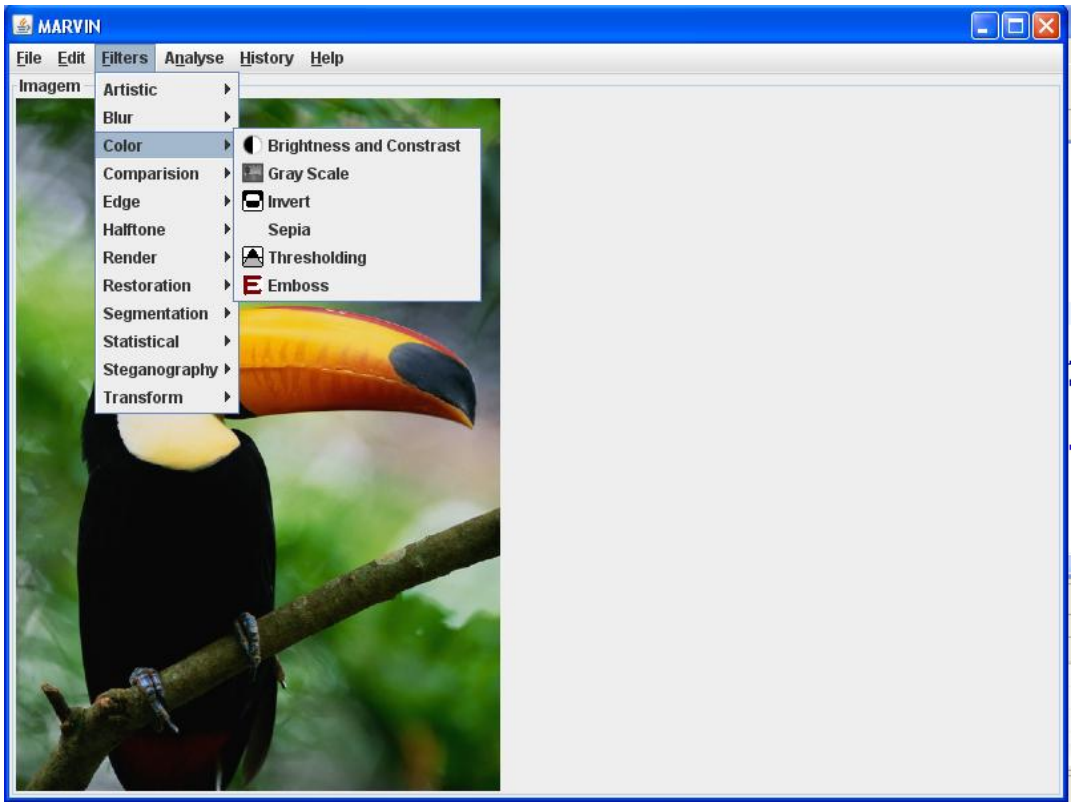

Figure VII.5 – User menu for accessing the Emboss Filter plugin in Marvin.

3. Screen Objects and Actions

Like many COTS image processing tools, the Marvin framework arranges plugins based on top-level functionality. In this case, the EF is a filter and doesn't alter the physical shape of pictured elements, but does change the pixel values to create the emboss effect. Based on experience with other image processing programs such as Adobe Photoshop and GIMP, the arrangement of menus and submenu items is consistent and reasonable for the user to navigate.

4. Requirements Matrix

Requirement Number	Description
REQ-1	Code for the EF will be written in Java.
REQ-2	The MIPF framework shall run on any JVM, version 1.6 and higher.

VIII. Submitted Code Patch

The code for the Emboss Filter is as follows and is highlighted in yellow indicating where new code was developed. This was based on the Invert Filter plugin and was modified to provide the emboss effect. Note, a video of the execution of the EF in Marvin is provided with an accompanying disc with this report along with all source code for the MIPF.

---------- Emboss.java ----------

```
/**
Marvin Project <2007-2009>

Initial version by:

Danilo Rosetto Munoz
Fabio Andrijauskas
Gabriel Ambrosio Archanjo

site: http://www.marvinproject.org

GPL
Copyright (C) <2007>

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License along
with this program; if not, write to the Free Software Foundation, Inc.,
51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
*/

package org.marvinproject.image.color.emboss;

import marvin.gui.MarvinFilterWindow;
import marvin.image.MarvinImage;
import marvin.image.MarvinImageMask;
import marvin.plugin.MarvinAbstractImagePlugin;
import marvin.util.MarvinAttributes;

/**
 * Invert the pixels color to create an emboss effect.
 * @author Chris Mack / Gabriel Ambrosio Archanjo
 * @version 1.0 02/28/2008
 */
public class Emboss extends MarvinAbstractImagePlugin
{

	MarvinAttributes attributes;

	public void load(){
		attributes = getAttributes();
		attributes.set("azimuth", 0);
		attributes.set("elevation", 0);
		attributes.set("depth", 0);
	}
```

```
        public void show(){
                MarvinFilterWindow  l_filterWindow  =  new  MarvinFilterWindow("Emboss",  400,450,
getImagePanel(), this);
                l_filterWindow.addLabel("lblAzimuth", "Azimuth:");
                l_filterWindow.addHorizontalSlider("sliderBrightness",  "azimuth",  -127,  127,  0,
attributes);
                l_filterWindow.newComponentRow();
                l_filterWindow.addLabel("lblElevation", "Elevation:");
                l_filterWindow.addHorizontalSlider("sliderBrightness",  "elevation",  -127,  127,  0,
attributes);
                l_filterWindow.newComponentRow();
                l_filterWindow.addLabel("lblDepth", "Depth:");
                l_filterWindow.addHorizontalSlider("sliderBrightness",  "depth",  -127,  127,  0,
attributes);
                l_filterWindow.setVisible(true);
        }

        public void process
        (
                MarvinImage a_imageIn,
                MarvinImage a_imageOut,
                MarvinAttributes a_attributesOut,
                MarvinImageMask a_mask,
                boolean a_previewMode
        )
        {

                int depth = (Integer)attributes.get("depth");


                // TODO: remove the two lines below. It's just for testing masks
                a_mask = new MarvinImageMask(a_imageIn.getWidth(), a_imageIn.getHeight());
                a_mask.addRectRegion(10, 10, 250, 450);

                boolean[][] l_arrMask = a_mask.getMaskArray();

                int r, g, b;
                for (int x = 0; x < a_imageIn.getWidth(); x++) {
                        for (int y = 0; y < a_imageIn.getHeight(); y++) {
                                if(l_arrMask != null && !l_arrMask[x][y]){
                                        a_imageOut.setIntColor(x, y, a_imageIn.getIntColor(x, y));
                                        continue;
                                }

                                int rDiff=0;
                                int gDiff=0;
                                int bDiff=0;

                        if (y > 0 && x > 0){

                                // Red component difference between the current and the upperleft
pixels
                                rDiff = a_imageIn.getIntComponent0(x, y) - a_imageIn.getIntComponent0(x-
1, y-1);

                                // Green component difference between the current and the upperleft
pixels
                                gDiff = a_imageIn.getIntComponent1(x, y) - a_imageIn.getIntComponent1(x-
1, y-1);

                                // Blue component difference between the current and the upperleft
pixels
                                bDiff = a_imageIn.getIntComponent2(x, y) - a_imageIn.getIntComponent2(x-
1, y-1);

                        }
                        else{
                                rDiff = 0;
                                gDiff = 0;
                                bDiff = 0;
                        }
```

```
                    int diff = rDiff;
                    if (Math.abs (gDiff) > Math.abs (diff))
                         diff = gDiff;
                    if (Math.abs (bDiff) > Math.abs (diff))
                         diff = bDiff;

                    int grayLevel = Math.max (Math.min (128 + diff, 255),0);

                   a_imageOut.setIntColor(x, y, grayLevel, grayLevel, grayLevel);

                  }
              }
        }
}
```