THE PENNSYLVANIA STATE UNIVERSITY

DEPARTMENT OF ELECTRICAL ENGINEERING

3D MEDICAL IMAGE PROCESSING TOOLBOX

Edward Wang

Spring 2013

A paper
submitted in partial fulfillment
of the requirements
for baccalaureate degrees
in Electrical Engineering and Computer Engineering
with honors in Electrical Engineering

Reviewed and approved* by the following:

William E. Higgins
Distinguished Professor of Electrical Engineering
Thesis Supervisor

John Mitchell
Professor of Electrical Engineering
Honors Adviser

* Signatures are on file in the Schreyer Honors College.

Abstract

The Multidimensional Image Processing Laboratory (MIPL) has been developing methods and systems for general 3D medical image analysis and visualization for over 20 years. During this period, several incarnations of a general 3D image processing software program have been created. The current program, VFX 2.1, was derived from these past programs and includes functions for processing both 8-bit and 16-bit multidimensional images. In light of modern developments in medical imaging, we worked towards upgrading VFX to a fully 16-bit standard. We did so by upgrading outdated functions or archiving unnecessary functions when progressing towards the next version of VFX. This paper describes the methods and testing performed when executing these changes. While this upgrade made significant strides towards this goal, there remains many functions left to be upgraded in future revisions of VFX.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Since the late 1980s, the Multidimensional Image Processing Lab (MIPL) at the Pennsylvania State University has been developing systems and techniques for processing 3D medical images. A critical aspect of this is the development of a system to process and manipulate these images. An ideal toolbox allows the user to load any image or images, choose from a wide variety of image processing functions, and display or save the result.

In this paper, we continue the development and modernization of VFX, a general image processing system. VFX 2.1, the most recent version, was developed in 2010 by Jue Li. While significantly improved from previous versions, VFX 2.1 still lacks full compatibility with modern 16-bit medical images. Our primarily goal in our upgrade to VFX 2.2 is to phase out the older 8-bit image processing functions, replacing them with 16-bit equivalents.

The current version of VFX is the result of decades of development and iteration that extends beyond the history of the MIPL. The following section provides a brief history of the origins and development of VFX. Afterwards, we will look at the initial state of VFX 2.1 and identify the problems to be addressed in the remainder of the paper.

1.1 History of VFX

The origins of VFX go back over 20 years. The system originated from the ANALYZE package, developed by R. A. Robb *et al.* This is an 8-bit image processing system developed in the late 1980s. The ROI (Region of Interest) system, developed by Dr. William Higgins while he was with the Mayo Clinic, was added to the ANALYZE system. ROI provided a graphical interface and toolbox of image processing functions.

In 1989, Dr. Higgins joined Penn State and continued the development of such tools in the MIPL. The next program, IMPROMPTU, was developed in the 1990s [6]. IMPROMPTU, which stands for IMage PROcessing Module for the Prototyping, Testing, and Utilization of image-analysis processes, was built on top of the ROI system. This program

also operated only on 8-bit images, but provided significantly more image processing functionality than ROI. Also, like ROI, IMPROMPTU was written in C and ran on Sun/Unix platforms.

In 1997, VFX 1.0 was developed by Greg Simon [5]. This upgrade moved the program to a Windows PC platform and was written in C++. It significantly improved the organization of the code, easily allowing for new functions to be added. VFX 1.0 was originally a command line program without a means of viewing images. In 2001, Anthony Sherbondy improved the system by providing a GUI using the Microsoft Foundation Class library [4].

In the mid 1990s, 16-bit images came into common usage as the output of 3D medical imaging devices. However, VFX 1.0 could only operate on 8-bit images. In 1997, Shu-Yen Wan began designing NV (NaVolgator or Volume Navigator), a system which could run 16-bit image processing functions [8]. This program also allowed for the loading and processing of object skeletons and data structures, which were a subject of research by the MIPL at the time. NV could originally only be run in a command line or with scripts specified in text files. In 2000, Kun-Chang Yu continued developing NV, importing it into an arterial tree analysis system called Tree Analyzer [9]. This system gave NV a limited graphical interface that allowed the user to view images and see the progress of operations.

In 2004, Andrew Weitz developed VFX 2.0, which incorporated the 16-bit image processing functionality of NV with the graphical interface and the 8-bit functionality of VFX 1.0 [10]. In 2010 Jue Li upgraded this program to VFX 2.1, which fixed many of the outstanding code issues from VFX 2.0 [3]. In the next section, we will look at VFX 2.1 and show its features and operation.

For a more detailed history of VFX and the programs that preceded it, refer to Jue Li's 2010 thesis on VFX 2.1.

1.2 Overview of VFX 2.1

VFX 2.1 is an image processing toolbox based on scripts. Scripts can be created interactively by adding a series of operations from VFX's function selection dialog box. The user must also specify the parameters for each function, which control options related to the function's operation. Once the script is created, the user can execute the script or save it for later use.

The functions in VFX vary widely in purpose and usage. The most commonly used are basic operations such as loading, saving, and displaying images. Additional functions are provided to manipulate workspaces, which allow working with more than one image at a time. The vast majority of the functions perform various image processing operations, such as filtering or morphology.

A run of a sample VFX script can be seen in Figure 1.1. The script can be seen on the top left. This is a four-step process. Step 1 of the process loads an image called *curve_branch_tube128*. Step 2 displays the image as is. Step 3 performs binary morphological dilation on the image using a 3×3×3 jack (6-connected) mask. Step 4 displays the resulting image. In the bottom left, the log window can be seen. This window displays information printed out by the functions. In this case, it shows information about the image printed by the DisplayImage function. The log window also gives the running time of each function. On the right side of the window, the two displayed images can be seen. Each image has an associated control dialog which changes the way the image appears. This allows the user to scroll through the slices, change the gray scale, and increase or decrease the size of the display window.

VFX 2.1 still needs modernization to improve its compatibility with modern medical images. Also, additional upgrades to the functions and interface are needed. In the next section, we will look at the outstanding issues with VFX 2.1 that we hope to address in our upgrade to VFX 2.2.

1.3 Problem Description

The goal of this thesis project is to make the following improvements to VFX:

1. Make overall improvements in the program's organization and interface.
2. Progress VFX towards a 16-bit standard.
3. Improve the robustness of VFX's functions.
4. Add additional functions to provide desired functionality.
5. Update and improve the program's documentation.

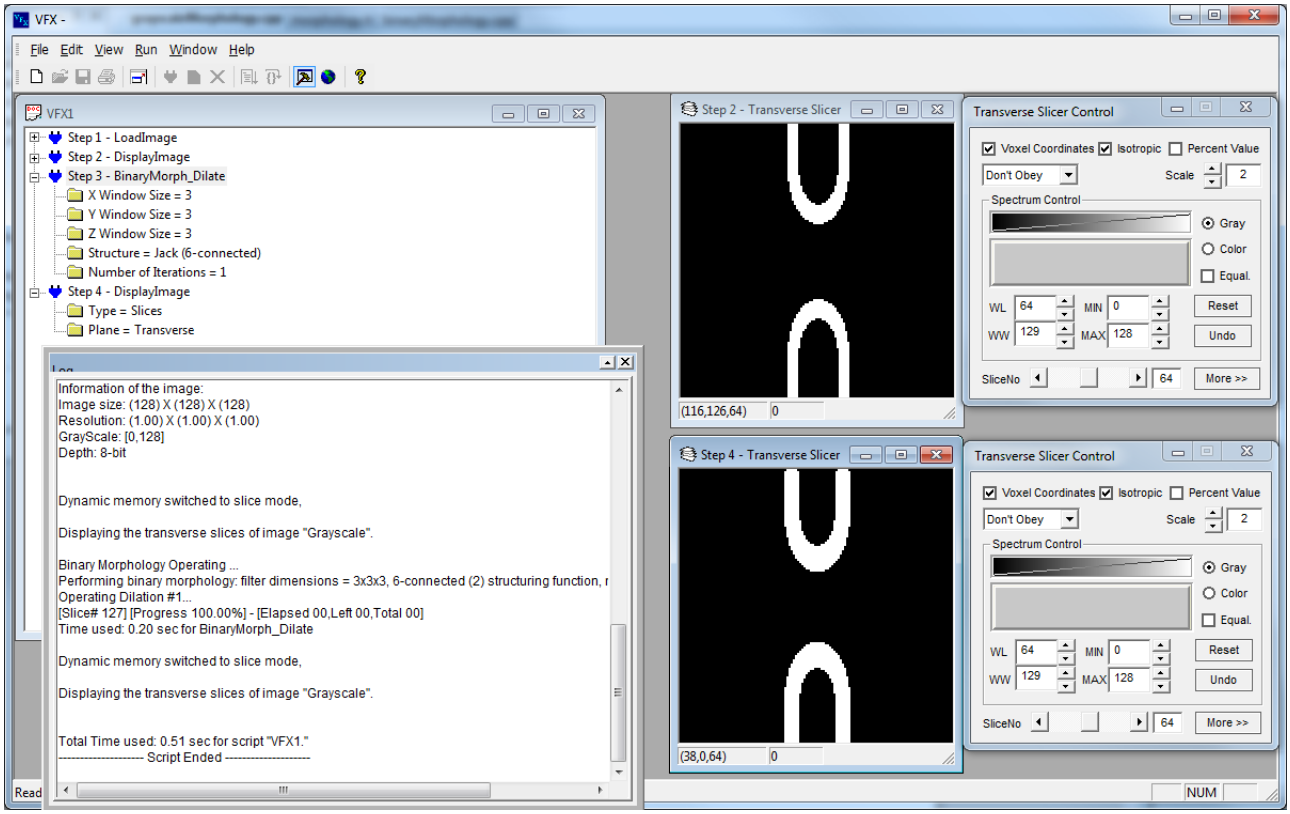


Fig. 1.1. The interface of VFX, showing a sample run of the *BinaryMorph_Dilate* function on the image *curve_branch_tube128*.

In this section, we will justify the need for these improvements and briefly overview the upgrades performed. Our first goal is to address several system level issues pertaining to VFX's interface and code structure.

One change to VFX 2.1's interface pertains to the function selection dialog box. Because every operation in VFX involves the selection of a function, it is very important to make this dialog as organized as possible. In this project, we address the following two issues:

- Outdated and obsolete filters are mixed in with current ones. It may not be clear which is the current version and what the differences between them are.
- All filters are sorted alphabetically; this ordering does not take into account the relative rates of use for certain functions, and often makes commonly used functions harder to find.

To address these, we create a function archive, providing a clear categorical division between current and outdated functions. We also allow for a custom ordering of functions within each category.

Another issue with VFX 2.1's interface is the progress bar window, which does not function correctly. The progress bar itself does not move. The current and total progress parameters are not correctly being passed from the current function to the progress bar window, and thus do not display properly.

Another system level upgrade considered in this project is the reorganization of code. This is particularly important for the file *nvsharedcommand.h*. This is one of the most commonly edited files in the program, as every time a developer adds, removes, or modifies a function, he or she must edit *nvsharedcommand.h*. However, this file is nearly 6,000 lines of code long, causing very long compile times and making it difficult to read and edit. We work to move and reorganize some of this code in this project.

The next major goals of our project involve improvements to the functions and categories of VFX. We will first look at the distinction between 8-bit and 16-bit functions.

VFX's functions fall into two classes. Those imported from VFX 1.0 can only operate directly on 8-bit images. While VFX allows these functions to run on 16-bit images, it automatically converts the grayscale values to 8 bit values during processing, resulting in

loss of information. Functions developed for VFX 2.x or NV, on the other hand, can operate on both 8-bit and 16-bit images.

8-bit images use 8 bits to store each voxel, allowing for graylevels between 0 and 255. 16 bit images, in contrast, use 16 bits to store the graylevel. These images generally allow for a 12-bit range, for values between -1024 and +1024. Modern medical imaging tends to exclusively use 16-bit images, as it allows scanned data to be saved without loss, so 8-bit images are very rarely used today. Our ultimate goal for VFX is to phase out all of the 8-bit functions and replace them with equivalent 16-bit functions, and this project makes significant progress towards this goal.

When phasing out the older functions, we must be careful not to lose any capabilities. Many of VFX's 8-bit functions include very diverse features and robust error detection. The equivalent 16-bit functions, however, tend to be more basic. This project includes significant upgrades to the functionality and error detection of 16-bit functions.

The final important aspect of this project pertains to the program's documentation. The *VFX 2.1 Function Manual* contains information about the purpose and usage of each function. Firstly, we update the manual to reflect the changes to functions mentioned above. Additionally, we work to improve the documentation of existing functions. Many of the older 8-bit VFX functions have very detailed documentation on the way they work and how to use them. However, most of the NV-based functions lack this detail, and some contain no more than a single line of information. We add additional relevant information to these functions' entries for the VFX 2.2 Function Manual.

1.4 Overview of the paper

The remaining chapters of this paper describe the upgrades made to address these issues. Chapter 2 describes the methodology used in making system level changes to VFX. Chapter 3 describes the methods used in upgrading individual functions and categories. Chapter 4 describes the procedure by which different types of functions were tested. Chapter 5 outlines the changes and current state of the functions in VFX. Finally, the Appendix gives the location of VFX and its function manual on the MIPL network.

Chapter 2

System-Level Upgrades

This chapter outlines the process and changes used to solve the system level issues from the Introduction. These are the changes related to the overall program and its interface. In Section 2.1, we look at how code organization issues were resolved. Next, in Section 2.2, we look at how the function selection dialog was improved. In Section 2.3, we describe the fixes to the progress bar window. In Section 2.4, we introduce the VFX 2.2 splash screen. In Section 2.5, we look at the process of adding a function and provide an example in Section 2.6.

The methods for making upgrades to specific functions and categories are given in the next chapter.

2.1 Code Reorganization

The first system-level upgrade to VFX pertained to the organization of code, particularly in the file *nvsharedcommand.h*. This file contains the initialization and operation information for every function in VFX, so it is by far the most commonly edited code file in the program. We worked to improve the organization of this file, primarily by removing code from this file and moving it to external files.

The first change was to move code pertaining to the initialization and implementation of functions to function-specific files. Most functions are already implemented in external code files; for instance, for the binaryMorpholgy function, the function is implemented in the separate file binaryMorphology.cpp but initialization and execution details are defined in *nvsharedcommand.h*. For these functions, this code was moved two new functions in the separate function file:

`firstTimeSetup( commandDecoder* oc, int cCode )`
This member function performs initialization in nvsharedcommand.h. It is run when VFX initializes to populate the function list.

`applyFilter( commandDecoder* oc, int cCode, const int id,`
`const char op[], NVImage<NVImage_t>& img )`
This member function performs the following steps as well as outputting appropriate info to the log:

1. Initialize filter instance
2. Apply parameters specified in GUI
3. Run filter on image 'img'

For a few functions in nvsharedcommand.h, no separate file exists for the function. The entire implementation, sometimes hundreds of lines long, is stored within nvsharedcommand.h. For instance, the *RenderSurfaceWithSkeleton* function was implemented without an external file. The code for this function can be seen in Figure 2.1. As can be seen, it contains nearly 150 lines of code, contributing to the excessive length and disorganization of the nvsharedcommand.h file.

For functions like this, a corresponding header (*.h) and source (*.cpp) file were created, and the code was moved into these new files. After moving the code over, the code in nvsharedcommand.h was replaced with references to these files.

In the next section, we will look at changes made to VFX's graphical user interface (GUI), starting with the function selection dialog.

2.2 Function Selection Dialog

The function selection dialog is the window used to select a function to add to a script. As such, it is one of the most commonly used windows in VFX. In this section, we show how the organization of this dialog was improved.

It was clear that outdated or obsolete functions needed to be archived in some way. It is often undesirable to completely remove the functions in question; one may wish at a future date to revive a formerly unneeded function, or there may be some behavior of the older 8-bit functions which should be preserved for future use or reference.

Our solution was to add a function archive at the end of the list of categories. The process by which a function is archived is by simply inserting an "Archive_" in front of the

nvsharedcommand.h

```
2211
2212    // Method: RenderSurfaceWithSkeleton
2213    // Usage: RenderSurfaceWithSkeleton <vtkSurfFileName> <skaFileName>
2214    // Description:
2215    //   Render the polygonal data stored in the file <vtkFileName>
2216    //   overlayed with the skeleton stored in <skaFileName>
2217    //
2218
2219    if (firstTimeFlag == 1) {
2220        if (COUT_DEBUG) {
2221            cout << "Adding : " << commandCode << endl;
2222        }
2223
2224        operationCommand->addACommand(commandCode ++, "VTK", "RenderSurfaceWithSkeleton");
2225        operationCommand->setDescription(commandCode-1, "Render an image surface and a skeleton representation.");
2226        operationCommand->addStringParameter(commandCode - 1, "Input Tree Surface File");
2227        operationCommand->addStringParameter(commandCode - 1, "Input Skeleton Representation");
2228    }
2229    else if (commandCode ++ == operationCode) {
2230        char surFile[1024];
2231        char skaFile[1024];
2232
2233        strcpy(surFile, operationCommand->getStringValue(commandCode - 1, "Input Tree Surface File", id));
2234        strcpy(skaFile, operationCommand->getStringValue(commandCode - 1, "Input Skeleton Representation", id));
2235
2236        cout << "Render the surface with skeleton ...\n" << flush;
2237
2238        aTime.init();
2239
2240        // Load the surface file
2241        vtkPolyDataReader *surf = vtkPolyDataReader::New();
2242        surf->SetFileName(surFile);
2243        vtkPolyDataMapper *surfMapper = vtkPolyDataMapper::New();
2244        surfMapper->SetInput(surf->GetOutput());
```

•
•
•

```
2329        // interact with data
2330        //renWin->SetSize(300, 300);
2331        renWin->LineSmoothingOn();
2332        renWin->Render();
2333
2334        // SAVEIMAGE( renWin );
2335        iren->Start();
2336
2337        skaActor->Delete();
2338        skaMapper->Delete();
2339        skaData->Delete();
2340        surf->Delete();
2341        surfMapper->Delete();
2342        surfActor->Delete();
2343        outlineData->Delete();
2344        mapOutline->Delete();
2345        outlineActor->Delete();
2346        aCamera->Delete();
2347        aRenderer->Delete();
2348        renWin->Delete();
2349        iren->Delete();
2350
2351        cout << aTime << " for " << operation << endl << flush;
2352        break;
2353    }
```

Fig. 2.1. The code for *RenderSurfaceWithSkeleton*, which is implemented entirely in *nvsharedcommand.h* with no external files.

name of the function's category. Each category was given its own archive. This way, the organization of the functions can be preserved even after they have been archived. For the current contents of the function archive, refer to Table 5.28 in Section 5.3.

Another means by which the function selection dialog's organization was improved is by reordering the functions therein. The existing Visual Studio library used to implement the function selection dialog provides two options for sorting; it either can be in alphabetical order, or in the order in which items are inserted. Before changing this setting, the order of functions in the file *nvsharedcommand.h*, which was previously arbitrary, needed to be defined in some way. Commonly used filters were moved to the top of the list, related filters were grouped together, and older 8-bit functions were moved to the bottom.

Another change made to the function archive was increasing the size of the window. Particularly, the portion of the window where the categories and functions appear was expanded. The new design allows 18 functions to appear at a time, while the old design only allowed for 12 to appear. This change reduces the amount of scrolling a user needs to do to find a desired function.

Additionally, we changed the names and categories of many individual functions to better reflect their usages. This makes it much easier to quickly find and identify a desired function. More details on these changes are given in Chapter 3.

The final implementation of the function selection dialog may be seen in Figure 2.2. Note the addition of the function archive and the changes to the window's size and organization.

In the next two sections, we will look at a few miscellaneous interface upgrades.

2.3 Progress Bar Window

The progress bar window appears when a script is executed and displays the progress of the current function being run. Figure 2.3 shows the progress bar window in VFX 2.1. As can be seen, the progress bar does not appear and the percentage progress reads 1900 percent.

To fix the issue of the progress bar not appearing, the bad pointer referring to the progress bar object needed to be fixed. The problem occurred because the progress bar pointer was initialized when VFX itself is initialized, and this pointer was not properly being passed to the progress bar window's code. To bypass this, the initialization of the progress bar was moved to the progress bar window code. The other problem with the

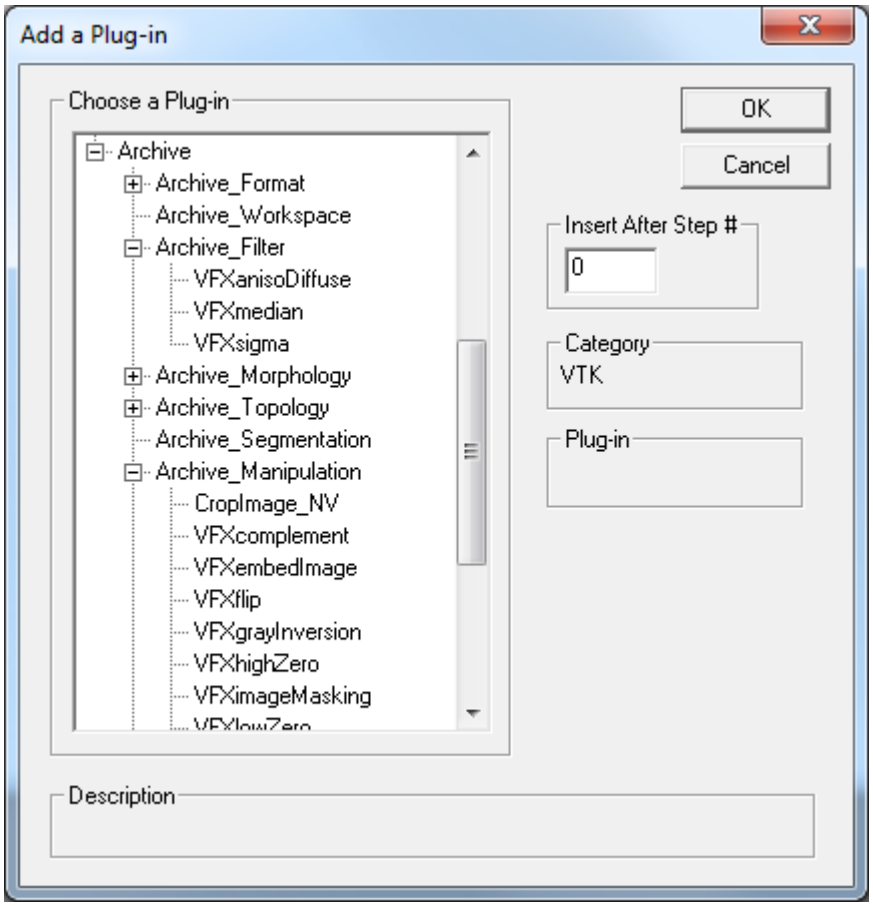


Fig. 2.2. Implementation of VFX 2.2 function selection dialog, showing the function archive and revised organization.

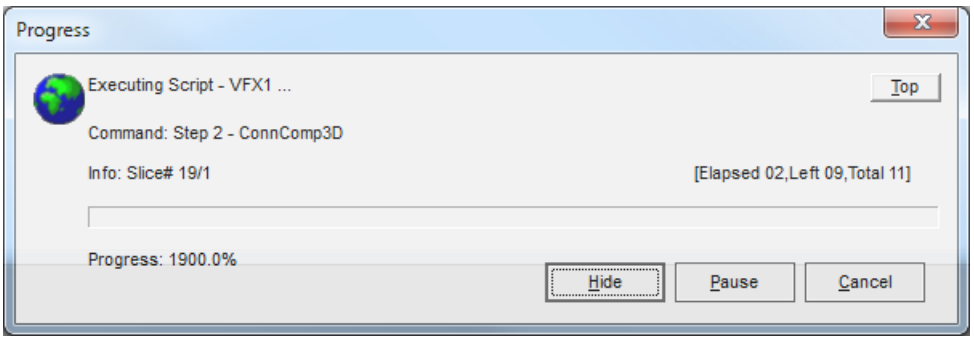


Fig. 2.3. VFX 2.1 Progress bar window, showing nonfunctioning progress bar and progress.

progress bar window was of the percent progress displaying incorrectly. This was solved by adding code to pass the necessary parameters from the calling script to this window. The new progress bar window is shown in Figure 2.4. As can be seen, the progress bar and percentage progress are now working correctly.

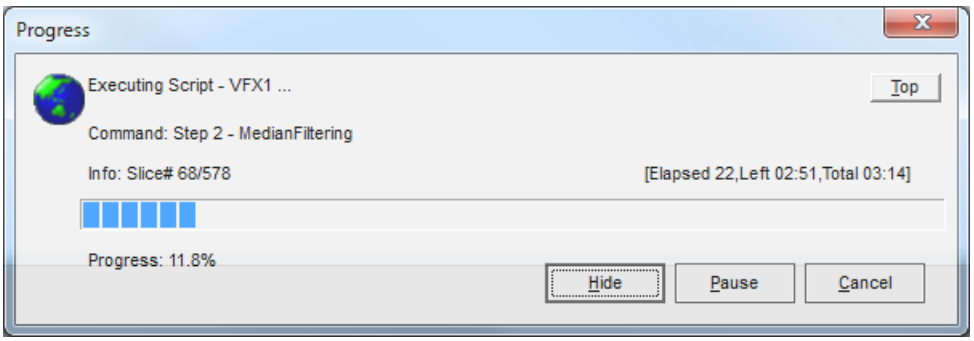


Fig. 2.4. Progress bar window with functioning progress bar and progress.

2.4 Splash Screen

The other miscellaneous interface upgrade was to update the splash screen. The VFX 2.1 splash screen had not been updated since VFX 2.0, so it displayed outdated version and copyright information. After upgrading to VFX 2.2, we created the splash screen in Figure 2.5 using Adobe Photoshop and replaced the VFX 2.0 splash screen with this one.

Next, we will look at the process to add a function to VFX.

2.5 Addition of Functions

In the course of this project, many new functions were added to VFX. The organization of the program makes it fairly easy to add new functions. The process described below was first introduced in version 2.0 of VFX.

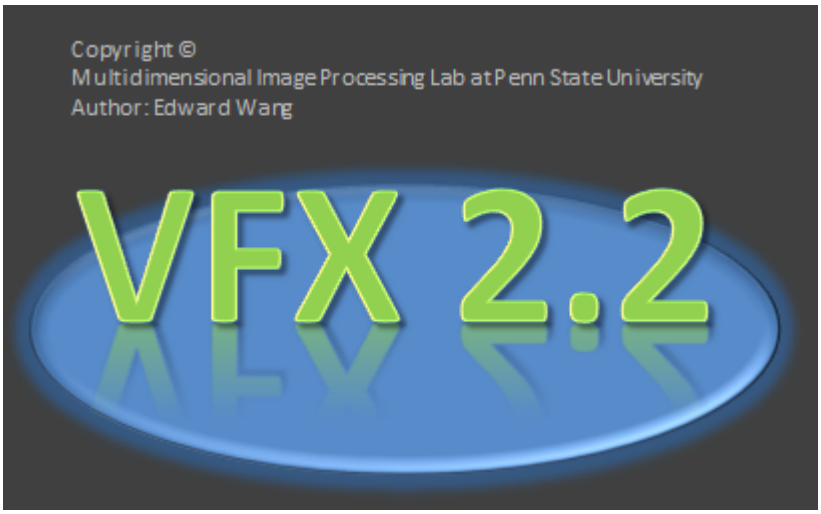


Fig. 2.5. VFX 2.2 splash screen image.

All code relating to the initialization and execution of every function in the program is stored in the file *nvsharedcommand.h*. The code in this file is executed first when the program starts, and also upon every subsequent script run.

2.5.1 Function information in nvsharedcommand.h

The minimum a programmer must do to add a function is to add an entry in the file *nvsharedcommand.h*. For very simple functions, the code can be implemented without any external files. The entry should contain the following:

- In the initialization section:
  - Add the function, specifying the category and name of the function.
  - Set the function's description.
  - Specify the input parameters of the function.
- In the execution section:
  - Read the parameters.
  - Execute the function.

This is the means by which many of the format and workspace operations are implemented, as the functions are fairly simple and use image libraries to perform their operations. However, for more complicated functions, such as filters, it is undesirable to implement the function entirely in the file *nvsharedcommand.h*. Because every function in VFX must be implemented in this file, doing so would cause the length of the file to become unmanageable. As such, separate files should be created to implement the function.

2.5.2 Format of Function Files

For each function, a header and a main source file should be created to store the relevant code. These files should contain the following information:

- Header
  - Required import statements.
  - Function prototypes.
  - Variable declarations.
- Main Code File
  - *getHelp()*, returning a descriptive line about the usage of the function.
  - *registerParameters()*, a function which specifies the names, types, and bounds of all the input parameters of the function.
  - *setParameters()*, a function which reads the values of the parameters at runtime and stores them to local variables.
  - *perform()*, which executes the function on an input image object.

Additional methods may be inserted in these files, but all functions should have at least these methods implemented. After these files have been created, an import statement should be added to the file *nvsharedcommandheader.h* to include the header of the function, and the revelant functions should be called in an entry in the file *nvsharedcommand.h*.

2.6 Example Process for Adding a Function

As an example, we will walk through how a programmer would add the *Thresholding* function to the *Manipulation* category. The following subsections will cover these steps:

1. Adding the Source Files
2. Writing the Comment Header
3. Coding the getName Function
4. Coding the getHelp Function
5. Coding the registerParameters Function
6. Coding the setParametersGUI Function
7. Coding the perform Function

2.6.1 Step 1: Adding the Source Files

First, we would add the two required files, the header and the source file. The *nv-alpha* directory stores all the information about the individual functions in VFX, including both the NV-based functions and the VFX-based functions. The header file will always be stored in the *INCLUDE* directory in the *nv-alpha* directory in the home directory of VFX; this is true of all header files which do not pertain directly to the interface of VFX. In Figure 2.6, the appropriate directory for the header file is highlighted in red.

When it comes to storing the source file, there are two pertinent folders, *NV* and *VFX* in the *nv-alpha* directory. The *VFX* directory contains the back-end source files for functions ported from the original 8-bit version of VFX. The *NV* directory, on the other hand, contains the source files for functions ported from NV, an older 16-bit image processing program. Because the thresholding function we are adding is designed to work with 16-bit images, we are adding our function to the *NV* directory. Within these two folders, there are also subfolders for each category and function. When adding a new function, one should identify the appropriate category for the function (Manipulation in this case), and create a new folder with the name of the function. In Figure 2.6, the appropriate directory for the source file is highlighted in green.

Once the programmer has added the two empty files, a *Thresholding.h* and a *Thresholding.cpp*, it is time to write the necessary code. First, the programmer should choose a separate function and the associated header and source files, then copy the code directly to the files just created. Doing so provides a template with the necessary comment headers, function declarations, and certain imports which provide a starting point to code the new function.

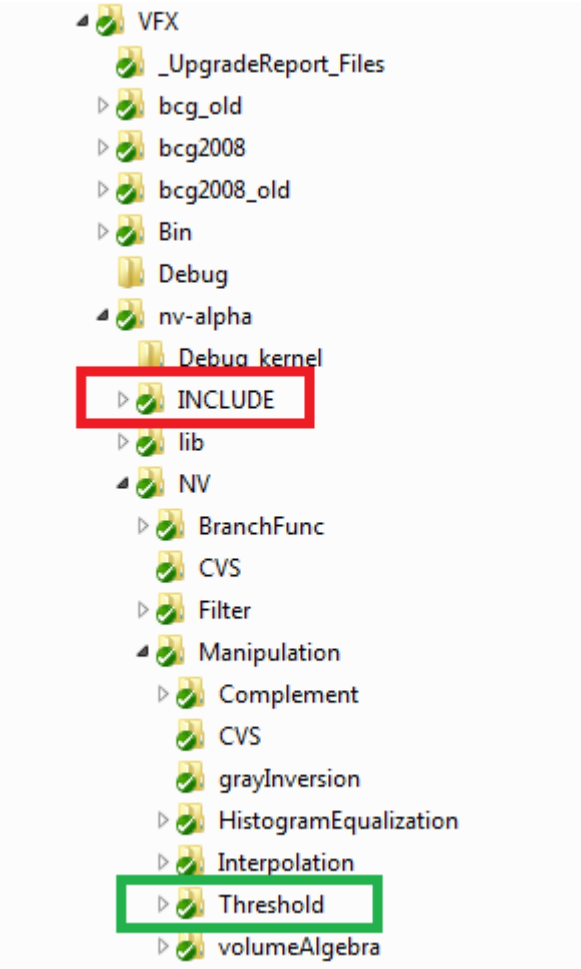


Fig. 2.6. The location to which the header (red) and source (green) files should be stored.

2.6.2 Step 2: Writing the Comment Header

The comment header appears at the top of both the header and source files. These two headers should be identical and contain the following information:

- *Module.* The name of the file; i.e., "thresholding.h".
- *Usage.* The purpose of the file itself; i.e., "Include file for Thresholding plug-in".
- *Purpose.* The purpose of the function implemented in the file.
- *Input Variables.* The parameters the user needs to enter as input for the function.
- *Returned Results.* The output of the function.
- *Processing Flow.* A basic overview of the process the function follows.
- *Restrictions/Notes.* Any restrictions; i.e., "Only operates on 8-bit image", or miscellaneous notes.
- *See Also.* Any related code file either referenced by, or that references, the current function.
- *References.* Any references for the algorithm or code.
- *Author.* The author of the function.
- *Date.* The date the function was originally created.
- *Revisions.* The date of last revision of the function.

In the remainder of the header file, any references to the template function must be changed to those of the new function, such as the name of the class. Also, at the bottom of the header file, the data structures and variables to be used in the processing of the image are declared. Any variables that are input parameters should be added here. If any additional functions need to be added for use in the processing flow of the function, the prototypes should be added to the header file as well.

Next, the functions in the source file should be changed to reflect the desired functionality of the new function.

2.6.3 Step 3: Coding the getName Function

The *getName* function requires a return value of the name of the function. For our example, we simply require:

```
return "Thresholding";
```

2.6.4 Step 4: Coding the getHelp Function

The *getHelp* function requires a descriptive string about the usage of the function. For our example, we will have:

```
return "16-bit thresholding to convert grayscale to binary";
```

2.6.5 Step 5: Coding the registerParameters Function

The *registerParameters* function registers all the required input parameters so that they can be entered by the user into the GUI. There is a variety of parameter types which can be used:

- *Integer Parameter*. This parameter can take on any integer value between the specified minimum and maximum. The parameter registration code takes the following form, where the last two arguments are optional:
  `params->addIntParameter(i, "`*parameter name*`",` *default value*`,` *minimum value*`,` *maximum value*`);`

- *Real Parameter*. This can take on any continuous real value between the specified minimum and maximum. The parameter registration code takes the following form, where the last two arguments are optional:
  `params->addRealParameter(i, "`*parameter name*`",` *default value*`,` *minimum value*`,` *maximum value*`);`

- *Choice Parameter*. This should be used when the parameter can only take a small number of discrete values, particularly with non-numeric choices or non-continuous values such as odd integers. Adding a choice parameter requires two discrete steps. The first is to register the parameter itself:
  `params->addChoiceParameter(i, "`*parameter name*`");`
  The next step is to register the individual choices:

`params->addChoices(i, "`*First choice name*`", `*Index of default value*`, "`*Second choice name*`", `*...additional choices...*`, "");`

For our example of a basic *Thresholding* function, the *registerParameters* function will contain the following:

```
params->addIntParameter(i, "Minimum foreground graylevel value");
params->addChoiceParameter(i, "Use max graylevel value?");
params->addChoices(i, "Use max graylevel value?", 0, "No", "Yes", "");
params->addIntParameter(i, "Maximum foreground graylevel value");
```

2.6.6 Step 6: Coding the setParametersGUI Function

The *setParametersGUI* function operates at runtime and assigns the variable values input by the user into the function itself. There should be a line corresponding to each parameter required by the filter. The code in this function takes on the following form:

- *Integer Parameter*:
  variable name `= params->getIntValue(i, "`*parameter name*`", id);`
- *Real Parameter*:
  variable name `= params->getIntValue(i, "`*parameter name*`", id);`
- *Choice Parameter*. There are two options for importing the value of a choice parameter. The first is to take the index of the selected option; i.e., 0 for the first option, 1 for the second option, etc:
  variable name `= params->getChoiceValueOffset(i, "`*parameter name*`", id);`
  The other option is to return the choice itself:
  variable name `= params->getChoiceValue(i, "`*parameter name*`", id);`

It is also a good idea to perform parameter checks on the input parameters at this point; more detail regarding the procedure and methods for doing so are given in the below section titled "Parameter Bounding and Checks".

For our example of a basic *Thresholding* function, the *setParametersGUI* function will contain the following. Our example will include a parameter check to ensure that the maximum foreground graylevel value is not less than the minimum foreground graylevel value:

```
m_opt = params->getChoiceValueOffset(i, "Use max graylevel value?", id);
```

```
m_par1 = params->getIntValue(i, "Minimum foreground graylevel value", id);
m_par2 = params->getIntValue(i, "Maximum foreground graylevel value", id);
if (m_opt && m_par1 > m_par2)
      throw commandDecoderError(0, 4, "Maximum < Minimum","");
```

2.6.7 Step 7: Coding the perform Function

Lastly, the *perform* function needs to be implemented. This function performs the necessary image processing operations on the input image. To begin, we use the following line to load the image pointer to the local image object:

```
Img = &img;
```

Once this is done, the processing of the image can begin. We will not run through all possible operations that can be performed on an image; see the NVImage class documentation for a more complete reference. The most commonly used functions are:

- `Img->getNumSlices()`. Returns the number of slices in the image.
- `Img->getNumRows()`. Returns the number of rows in the image.
- `Img->getNumColumns()`. Returns the number of columns in the image.
- `Img->kij(`*slice number*`,` *row number*`,` *column number*`)`. This function allows for both accessing and modifying the pixel values in an image. If used by itself, the function returns the value of the specified voxel. If a number is assigned to this function using the '=' character, the specified number is saved to the voxel.

A combination of these functions can implement almost any filtering, manipulation, or morphological operation.

In the next chapter, we will look at upgrades made at the function level, beginning with the methods used to add functions to VFX.

Chapter 3

Category and Function Upgrades

This chapter outlines the process and changes used in upgrading the individual categories and functions in VFX. Our ultimate goal is to modernize VFX by upgrading all functions to 16-bit, and these upgrades make significant progress towards this goal. In Section 3.1, we will look at general parameter bounding and error detection. In the remaining sections, we will look at the individual categories and the upgrades made to their functions. We will look at the format (Section 3.2), workspace (Section 3.3), morphology (Section 3.4), topology (Section 3.5), manipulation (Section 3.6), and other categories (Section 3.7), in that order. Finally, in Section 3.8, we will look at the VFX Function Manual and the process of documenting these function changes.

Chapter 4 then overviews the process used for verifying these changes.

3.1 Parameter Bounding and Error Detection

Parameter bounding and error detection are two distinct means of preventing invalid parameter values. Bounding works while the user is editing the script, while error detection occurs at runtime.

Bounding can prevent errors such as running 0 iterations of a filter or specifying a filter with an even-valued mask dimension. This is a feature already incorporated in many functions in VFX. It is implemented in the parameter registration process. There are two ways of adding parameter bounding. The first, as seen in Figure 3.1, is to specify all the valid values in a list. This is most appropriate in cases where only specific values of the parameter will work. In the example, the parameter is for a filtering window size, and only odd values are valid.

The second means of bounding, as seen in Figure 3.2, is to specify a maximum and minimum value of the parameter. In this case, the user may enter any value that falls within the specified limits. In the example, the user is only allowed to enter values between 0.00 and 0.25 for the lambda value of the anisotropic diffusion filter.

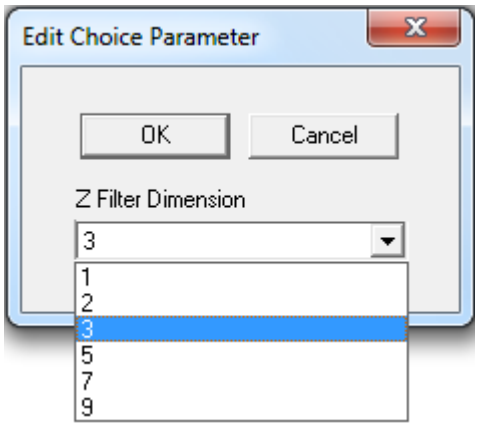


Fig. 3.1. Bounding the median filter's Z window dimension to odd values between 1 and 9 using a choice parameter. The user may select any of these choices.

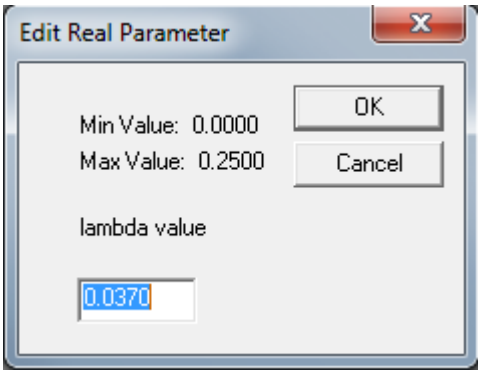


Fig. 3.2. Bounding the lambda value of the AnisotropicDiffusionFilter between 0.00 and 0.25 using parameter limits. The user cannot enter a value outside these bounds.

Parameter bounding only works to prevent invalid values that are *never* allowed. In some cases, the list of valid values may vary depending on other parameters or on the input image. Parameter error checking prevents invalid values in these situations.

Parameter error checking can prevent the entry of incompatible parameters, and provides additional verification over bounds. For instance, if the user specifies a 2D structure with a window depth greater than 1, these values satisfy the independent parameter bounds but not the parameter compatibility checks that occur at runtime. This feature is inserted after the function reads the parameter values, but before the function is executed. If the parameters are incompatible, it is desirable to cancel execution of the current process and require the user to correct the issue. To do so, a non-fatal NVException is thrown, which stops the script and allows a specific error message to be displayed. The following lines of code allow this to be done:

```
char errSubj[80];
throw(NVException(errSubj, "Error Message"));
```

In Figure 3.3, we see an example of the implementation of parameter error checking. This function, *Thresholding*, uses both a maximum and a minimum as an input. It is obviously not valid to enter a minimum value greater than the maximum value. If the user does so, it causes this error message to be thrown during runtime and stops the execution of the script.

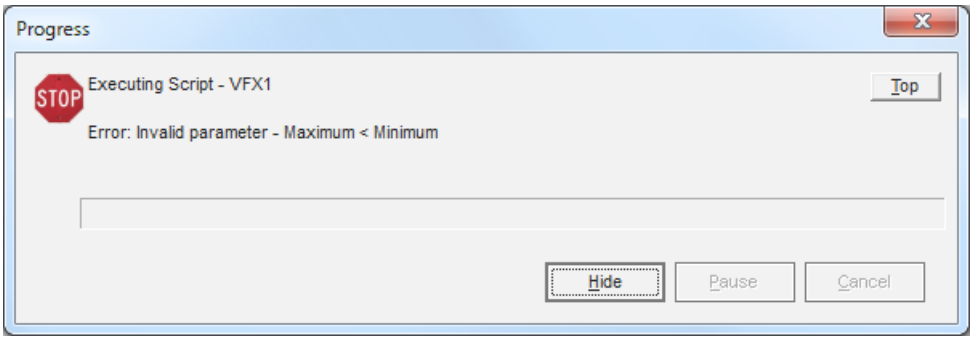


Fig. 3.3. An example of parameter error detection for the Thresholding function using exceptions at runtime.

In the preceding sections, we looked at general procedures applicable to all categories of functions. In the following sections, we will describe specific upgrades made to individual categories. This is a list of all categories in VFX that were upgraded in this project:

- Format
- Workspace
- Filter
- Morphology
- Topology
- Manipulation

These are the categories that were NOT considered for upgrades:

- Segmentation
- Measurement
- System
- VTK
- Turnkey
- BranchFunc
- TestingPurpose

In the following sections, we will provide details on the upgrades that were performed. Categories will be described in the order they appear in the function selection dialog.

3.2 Format Function Upgrades

Several minor upgrades were made to functions in the Format category. A bug in the CreateImage function was fixed, and the ability to load MetaImage (*.mhd) and bitmap (*.bmp) files was added.

The CreateImage function in the Format category of VFX crashed upon the second and subsequent runs of the function, as the function failed to clear the image workspace

memory before attempting to regenerate the image. To solve this, function calls were added to clear the previously generated image before the new image is generated.

Ronnarit Cheirsilp upgraded the LoadImage, LoadHeader, and SaveImage functions to support the loading and saving of MetaImage files. This is a file format commonly used in medical imaging consisting of a *.mhd header file and a *.img image file.

A function to load 2D bitmap files was added in this version of VFX. Loading a bitmap file required two distinct steps; loading the bitmap file into memory, and porting it to an ANALYZE format image.

To load the bitmap file, the bitmap file header is first read. This initial header includes information about the size of the DIB header, which includes additional details about the image. The DIB header is then read, storing the image size and depth. Using this information, the remainder of the file is then read and stored into memory.

To convert the BMP data to ANALYZE format, both the header information and the image data needed to be stored. Some of the header information, such as the image size and depth, was copied directly from the BMP header. Other information, such as the resolution, has no equivalent in the BMP header and is given a default value. After the header information is stored, the image is initialized and the voxels are individually copied.

Next, we will look at the changes made to the Workspace category.

3.3 Workspace Function Upgrades

In the Manipulation category, there were numerous functions operating directly on multiple workspaces. In VFX 2.2, these functions have been moved to the Workspace category. Parameter error detection for image size mismatch and other incompatibilities were added for most of the functions.

Here, we list the functions which were moved to the Workspace category. Most of the workspace functions are in sets of three; the “From” functions operate on the current workspace, the “To” functions operate on the specified target workspace, and the “FromTo” functions operate on two specified workspaces. The functions moved are shown below:

- *WorkspaceCrop.* Crops one workspace to a different workspace.
- *CopyFrom, CopyTo, CopyFromTo.* Copies an image from one workspace to another.
- *EmbedFrom, EmbedTo, EmbedFromTo.* Embeds an image from one workspace to a subset of another.

- *SubFrom, SubTo, SubFromTo.* Subtracts one image from another image.
- *AddFrom, AddTo, AddFromTo.* Adds one image to another image.
- *MultFrom, MultTo, MultFromTo.* Multiplies one image with another.
- *DiffFromImage, DiffImages.* Prints the different voxels between two images.

The next section covers upgrades to the Morphology category.

3.4 Morphology Function Upgrades

In this project, significant work was done to upgrade VFX's morphology category to a 16 bit standard. The existing 16 bit binary and grayscale morphology functions underwent significant upgrades. New 16 bit maximum and minimum morphology functions were introduced. Finally, the 8-bit equivalents of these functions were archived. We will begin by overviewing the Morphology category prior to these upgrades.

The original organization of the morphology functions consisted of the following functions:

1. *BinaryMorphology.* 16-bit binary morphology. Erode/Dilate/Open/Close is a selectable parameter.
2. *GrayscaleMorphology.* 16-bit grayscale morphology. Erode/Dilate/Open/Close is a selectable parameter.
3. *VFXbinErode, VFXbinDilate, VFXbinOpen, VFXbinClose.* 8-bit binary morphology
4. *VFXgrayErode, VFXgrayDilate, VFXgrayOpen, VFXgrayClose.* 8-bit grayscale morphology
5. *VFXmax, VFXmin.* 8-bit maximum/minimum morphology

Our goal in this upgrade was to archive the 8-bit VFX morphology functions. Before we could do so, functionality from these 8-bit functions that was missing in the 16-bit functions needed to be ported over. In addition, other new features were also desired in the 16-bit functions. Some of the changes include:

1. Parameter bounds and compatibility checks

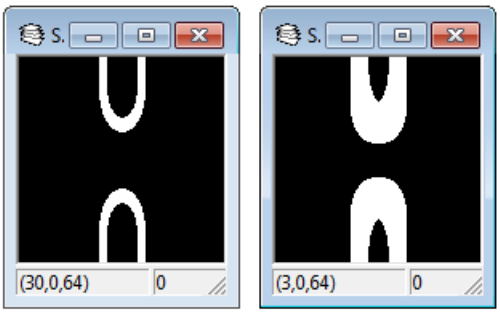


Fig. 3.4. A sample slice of *Curve_Branch_Tube* before (left) and after (right) binary morphological dilation using the *binaryMorph_Dilate* function.

2. Separation into separate Erode, Dilate, Open, Close functions
3. Introduce additional morphology structures
4. Introduction of Maximum and Minimum functions

First, we separated the BinaryMorphology and GrayscaleMorphology functions into individual functions for erosion, dilation, opening, and closing. New code files were created and the relevant code was divided amongst them. The "number of iterations" parameter was removed for opening and closing, as these are idempotent functions; they produce the same result regardless of how many times they are run.

We also upgraded the capabilities of these functions. A 2D 4-connected morphology structure was present in the older 8-bit morphology functions but not in the newer equivalents. In this project, we introduced this structure in the 16-bit morphology functions. The option for this structure was added to the structure selection parameter, and the corresponding code to perform 4-connected morphology was added.

Finally, we introduced maximum and minimum morphology functions for 16-bit images. Two new functions called *Maximum* and *Minimum* were introduced. These functions analyze all the voxels in a specified mask, find the maximum or minimum, and assign that value to the voxel of consideration. New files were created to implement these two functions. The available morphology structures for these functions were ported over from the BinaryMorphology function.

In the next section, we will look at changes made in the Topology category.

3.5 Topology Function Upgrades

The only functions considered in the Topology category were the 16-bit *ConnectedComponentLabeling* and *ConnComp2D* functions. In this project, we worked to upgrade these functions to match the functionality of their equivalent 8-bit functions. We also combined these two functions into a single function, *ConnCompLabeling*. We will first overview the capabilities of the 8-bit connected component labeling functions.

The 8-bit VFX connected component analysis functions, *VFXConnComp2D* and *VFXConnComp3D*, allowed for the following selectable parameters:

1. *Connectivity.* The options are 4/8 connectivity for 2D analysis, or 6/26 connectivity for 3D analysis. The higher connectivity options allow for edge/vertex connectivity, while the lower option only allows for face connectivity.

2. *Minimum Region Size.* This is the smallest contiguous region size allowed in the output image. Regions below this size are discarded.

3. *Maximum Number of Regions.* This is the maximum number of regions that is retained in the output image. If the number of regions exceeds this number, the largest regions are retained while the smaller ones are discarded.

4. *Save Labels.* If this option is enabled, the regions in the output image are given independent labels, or grayscale values. Labels are assigned between 1 and 255. If this is disabled, the background is assigned to 0 and the foreground is assigned to 1. In Figure 3.5, the effect of this option can be seen. In this example, image *r1a* was thresholded with a threshold of 128 and connected component labelling was performed with saving labels enabled. In the displayed slice, 81, six unique regions are visible. As can be seen, all regions have unique grayscale values assigned.

In the 16-bit VFX connected component analysis functions, *ConnComp2D* and *ConnectedComponentLabelling*, only the first two parameters are available. The remaining two features needed to be added to both of the 16-bit functions. While the 2D and 3D functions are distinct and have different implementations, the means via which these features were implemented remain fairly consistent between them.

To implement the *Maximum Number of Regions* parameter, the largest regions in the image needed to first be identified. In the original function from VFX 2.1, an array of region sizes is created; the index of the array corresponds to the label of the region. To

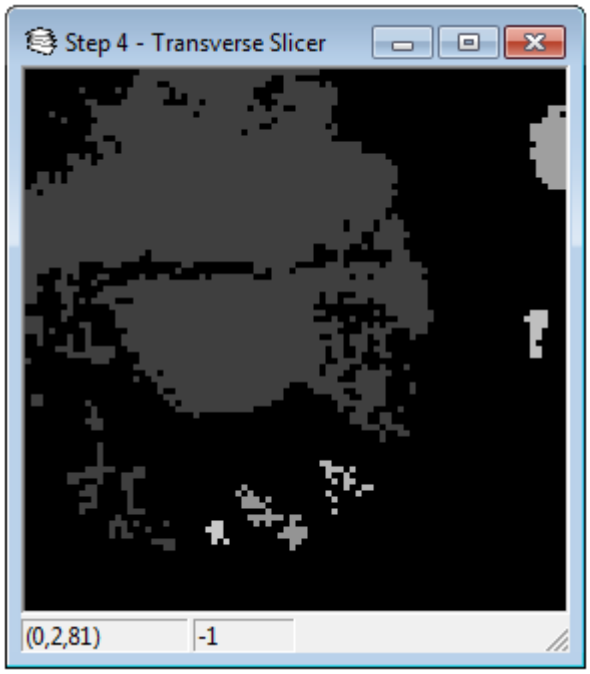


Fig. 3.5. Slice 81 of image *r1a* after *Thresholding* and *ConnCompLabeling* with labels saved. Six unique regions are visible.

allow for reordering of labels, a separate array was initialized with the labels of the regions. Then, the arrays were sorted in descending order using the region size as the key. Next, all regions beyond the specified maximum were removed by setting all voxels with their labels to -1. Finally, because the remaining labels are no longer consecutive, these were then reassigned to label numbers between 0 and the maximum number of regions.

To implement the *Save Labels* parameter, each independent region needed to be assigned a unique grayvalue in the output image. While the existing function assigned labels as grayvalues within the analysis process, the labels are not retained nor reordered after the region filtering process. To address this, the segmentation process to remove the labels was changed from default behavior to an option. Also, a loop was written to reassign the labels to grayvalues between 0 and the number of regions. Because we are working with 16-bit images, we are now able to assign tens of thousands of labels instead of just 254.

Finally, after the changes had been made, it we combined the 2D *ConnComp2D* and the 3D *ConnectedComponentLabelling* functions into a single function. To do this, an additional parameter was added to the *ConnectedComponentLabelling* function which allows the user to select between 2D and 3D analysis. Next, the code to perform 2D connected component labelling was copied over from the *ConnComp2D* function, and all variable names and other details were updated. The new 2D/3D parameter allows the user to select between the two operations within this filter.

In the next section, we will look at the upgrades made to the Manipulation category.

3.6 Manipulation Function Upgrades

Another category which underwent major changes is the Manipulation category. Several 16-bit versions of existing 8-bit functions were introduced. Additionally, more robust error detection was added to several functions. Finally, all functions operating between multiple workspaces was moved to the Workspace category. We will first look at the newly introduced 16-bit functions.

The Manipulation category of VFX 2.1 contains numerous 8-bit functions without 16-bit equivalents. To progress towards our goal of a 16-bit standard, we introduced several new equivalent functions and archived the 8-bit versions. We did so for the following functions:

- *AssignValue*. This assigns a specified value to a specified voxel.

- *ImageMasking.* This zeroes out voxels outside a defined window.
- *HighZero.* This zeroes out grayscale values above a certain threshold.
- *LowZero.* This zeroes out grayscale values below a certain threshold.
- *RangeLabel.* This reassigns a specified range of grayscale values to a single value.
- *CropImage.* This extracts a subset of the original image to be the new image. Two existing functions, CropImage_NV and CropImage_new, did not function as intended.
- *GrayInversion.* This inverts the grayscale values.

3.7 Other Categories

Only the above mentioned categories were considered in the course of this project. All other categories were left as is, and are in the same state as they were in VFX 2.1.

In the next section, we will look at the documentation of the above mentioned changes in the VFX Function Manual.

3.8 Function Manual

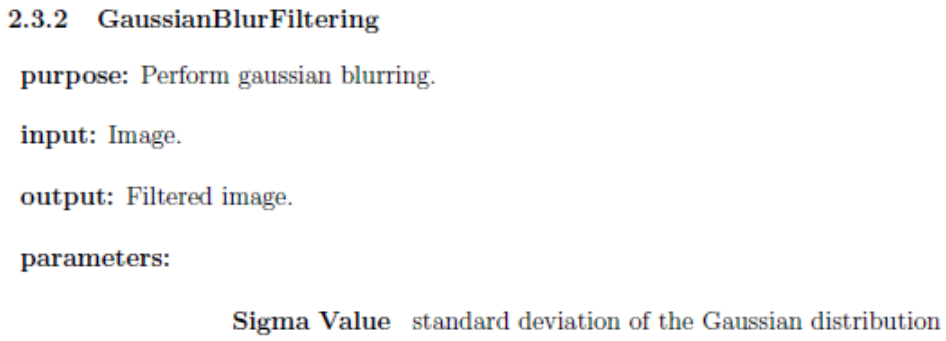
The VFX 2.1 Function Manual describes every function in VFX. It provides information about the purpose and methods of each function, as well as guidance about how the function should be used. The function manual for VFX 2.1 is written using LaTeX and the source files are stored alongside the pdf file for the function manual.

It is organized in the following manner:

1. *Table of contents.*
2. *Overview of categories.* This section summarizes all of VFX's functions and is provided as a quick reference. It lists all functions by category and gives a short, single-line description of each.
3. *Detailed Description.* Detailed descriptions of the purpose, parameters, and usage of each function are given. The functions are separated by category and each function is given a separate page.

During the development of VFX 2.2, significant changes were made in VFX's functions. These changes include the order in which functions appear, their parameters and operation, and their names. In addition, many older functions were archived and many new functions were added. These changes needed to be reflected in the updated VFX function manual.

Additionally, for many functions, there was a significant lack of detail in the detailed description of many functions. In Figure 3.6, for instance, the documentation for the *GaussianBlurFiltering* function is shown. As can be seen, very little useful information about the function's operation and usage is provided. During the course of updating the function manual, an effort was made to improve the description of functions which lack adequate documentation.



2.3.2 GaussianBlurFiltering

purpose: Perform gaussian blurring.

input: Image.

output: Filtered image.

parameters:

Sigma Value standard deviation of the Gaussian distribution

Fig. 3.6. VFX 2.1 Function Manual entry for the *GaussianBlurFiltering* function.

The location of the final VFX 2.2 Function Manual can be found in Appendix A.

In the next chapter, we will look at the procedure used in testing the functions changed above.

Chapter 4

Tests

In this chapter, the detailed testing procedure is outlined for the different elements of the program. In Section 3.1, a general introduction to the considerations and available sample data is given. In Section 3.2, the general process to test a function is described. Finally, we look at the process for testing Workspace (Section 3.3) and Format (Section 3.4) functions specifically. The results of this testing are given in the next chapter, Results.

4.1 General Procedure and Sample Data

When functions are added or modified, testing must be done to ensure that they work as intended. When doing so, the primary goal is to cover all possible test cases, to account for all possible images and parameter values. As a result, a wide sample of images and test cases are needed to fulfill these various possibilities.

The first factor to account for is the file format of the image. Many medical images currently used by the lab are in ANALYZE format, a 3D image format which consists of a *.img file and a *.hdr file. The header file includes information about the dimensions, resolution, and other information about the image, while the image file stores the voxel data. When testing the LoadBMP function, a sample 2D bitmap image was needed to verify its functionality. Bitmap header information and image data is stored within the same *.bmp file.

The second factor to consider is the distinction between 8-bit and 16-bit images. As noted earlier, many older functions only operate on 8-bit images, while newly coded functions handle both 8-bit and 16-bit images. 8-bit images allow for 256 distinct gray levels within an image, while 16-bit images allow for 65,536 distinct gray levels.

A third property of test images is whether it is binary or grayscale, or in the case of bitmap images, RGB. Binary images only have two distinct grayscale values, one being zero (black) and one being nonzero (white). Grayscale images, on the other hand, can cover a much wider range of voxel values. For bitmap RGB images, there are three separate values

stored for each pixel; one represents the brightness of the red channel, one for the green channel, and one for the blue channel.

The following images were used in testing:

- *curve_branch_tube128 (ANALYZE, 8-bit, binary, 128 × 128 × 128)* [7] This is a 3D image of a synthetic tube that branches from a single tube into two. It has two distinct voxel values of 0 and 128, where the brighter value of 128 represents voxels inside the tubes.

- *r1a (ANALYZE, 8-bit, grayscale, 97 × 97 × 102)* [2] This is a 3D image of a heart. The voxel values range from 0 to 255, where brighter regions represent the left ventricular chamber and the darker regions represent the myocardium heart muscle.

- *20349_3_3_B31 (ANALYZE, 16-bit, grayscale)* [1] This is a 3D image of a human chest. The voxel values represent Hounsfield units, ranging from around -1100 to +1000 HU. A sample slice of this image is shown in Figure 4.1.

- *gray (Bitmap, 8-bit, RGB)* This is a sample BMP image with three 8-bit color channels. Once converted to grayscale, it contains four distinct gray levels ranging from 0 to 255.

Next, we will describe the process for testing general functions using this test data.

4.2 General Function Testing

The test procedure described in this section is applicable to the majority of functions in VFX. We are considering functions which operate on the image in the current workspace and directly modify the image in place. All of the tested functions in the filter, manipulation, and morphology sections operate in this manner.

In this project, all functions which were modified in any way were tested. For functions meeting the above criteria, the following process was used:

1. Load the sample image data.
    - 8-bit binary image curve_branch_tube
    - 8-bit grayscale image r1a for grayscale morphology and max/min functions
    - 16-bit grayscale image 20349_3_3_B31 for grayscale and max/min functions

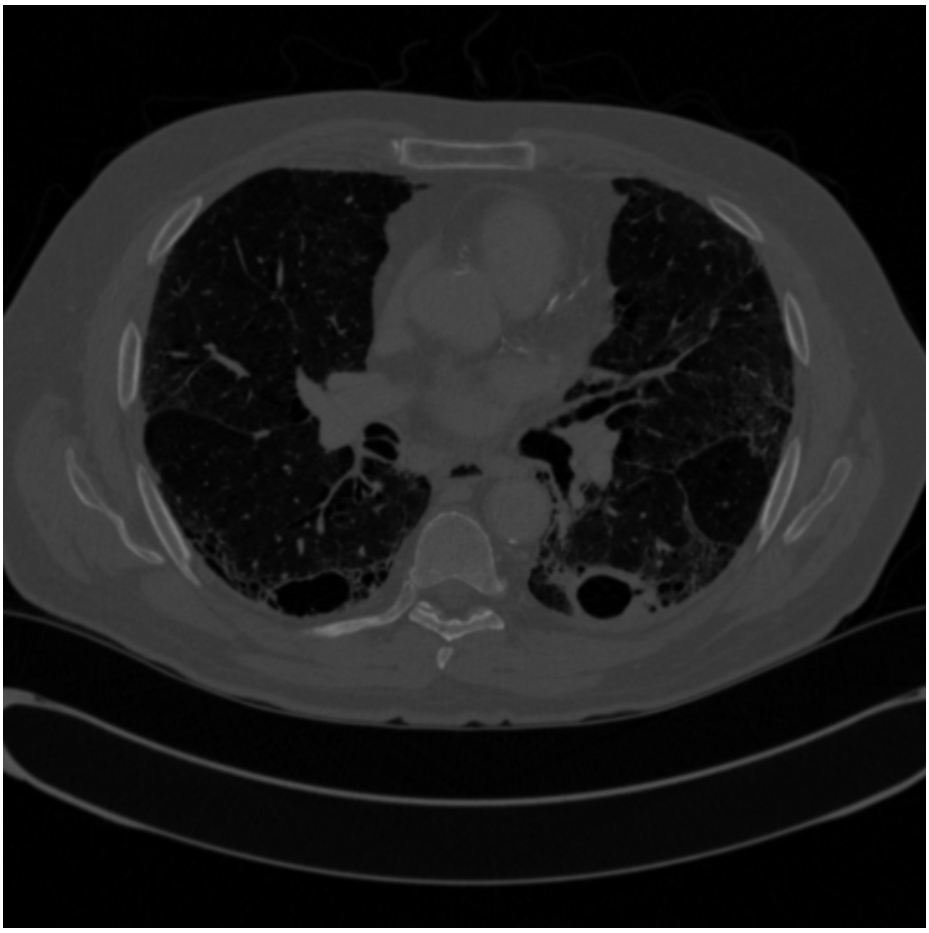


Fig. 4.1. Slice 175 of 16-bit Image *20349_3_3_B31*.

2. Display the unmodified image for later comparison.

3. Operate on the image.

    - Vary the window size (dependent on function):
        - Default size 3×3×3
        - Larger size 5×3×5
        - 2D mask 3×3×1
    - For morphology functions:
        - Test all possible morphology structures
        - For erosion, dilation, maximum, and minimum functions, try multiple iterations

4. Display the image after operation and compare with original

An example of a simple testing script used to test the *MedianFiltering* function can be seen in Figure 4.2. Because we are simply testing a function that operates on a single image, we simply need to load an image and operate on it. To test the parameters, we simply run the test script multiple times after changing the parameters of the function.

This is a breakdown of the purpose of the functions used in the testing script mentioned above:

1. *LoadImage* Loads the test image *r1a* into the current working image.

2. *DisplayImage* Displays the original unmodified image. By running this function at this point in the script, we avoid the need to create an additional workspace to store a copy of the unmodified image.

3. *MedianFiltering* Performs median filtering on the image. According to the function documentation, this function should at each point in the image take the median of all voxel values within the window specified by the three input dimensions, and assign that value to the current voxel. We are using a $5 \times 5 \times 5$ sized window for this run of the testing script, but must also run it for additional sizes to ensure that all parameter inputs work.

4. *DisplayImage* Displays a copy of the image after it has been median filtered using the above function. Now that both the original and modified images have been output,

we can directly compare the two images and visually determine whether the output is correct.

The output of both *DisplayImage* functions in the above testing script can be seen in Figure 4.3. It can clearly be seen that the modified image is significantly smoother than the original image, yet edges have still been preserved to some extent. Also, while not shown, the degree of smoothing is visibly greater than when the filter was run with a $3 \times 3 \times 3$ sized window. This is the desired output of the *MedianFiltering* operation.

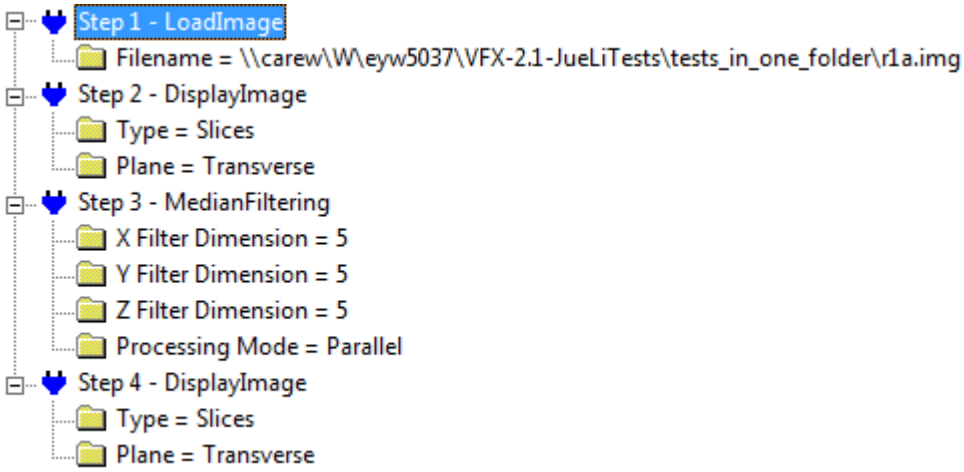


Fig. 4.2. A sample script to test the *MedianFiltering* filter.

When testing the Workspace and Format categories, we are considering functions which do not operate in place on a currently loaded image. Thus, we need specialized testing procedures to test these functions. We begin by looking at the tests done on the Workspace category.

4.3 Workspace Function Testing

By default, VFX scripts use a single workspace. The functions in the Workspace category allows users to create and manipulate multiple workspaces. This allows the user

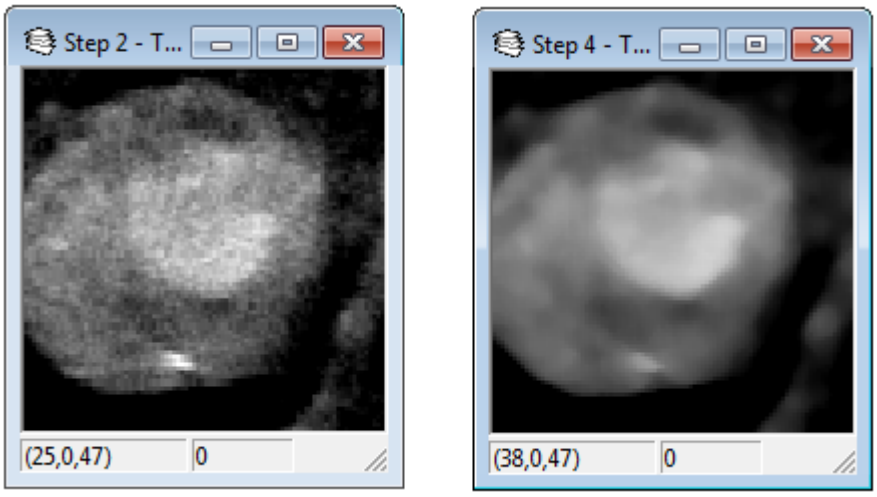


Fig. 4.3. A slice of image *r1a* before (left) and after (right) median filtering. This is the output of the test script in Figure 4.2.

to work with multiple images or multiple copies of images. In this section, we describe the procedure for testing Workspace functions.

Most of the workspace functions require two initialized workspaces to function. Thus, the first step in testing is to create and set up the two workspaces. The following procedure was used to do this:

1. Create the initial workspace, *Image1*.
2. Set *Image1* to be the working image.
3. Load the test image.
4. Add an additional workspace, *Image2*.
5. Set *Image2* to be the working image.
6. Copy or re-load the test image.
7. Modify the image using a filter or crop operation.

Once the two workspaces are initialized, we can now apply a workspace operation to the two workspaces and verify that the result of the operation is correct given the two inputs. The copy functions are tested through their use in the above test script. An example of an actual workspace testing script for the *EmbedTo* operation can be seen in Figure 4.4.

This is a more detailed rundown on the individual functions used in the above referenced testing script:

1. *CreateWorkSpace* Creates the initial working space and assigns it an image name of "Orig".
2. *SetWorkImage* Sets the current working image to the workspace we have just created, so that the subsequent operations operate on this workspace.
3. *LoadImage* Loads the test image *r1a* into the working image "Orig".
4. *AddWorkSpace* Adds an additional workspace and assigns it an image name of "Modified".
5. *SetWorkImage* Sets the current working image to the workspace we have added in the previous step.

6. *LoadImage* Loads the test image *r1a* again, this time into the "Modified" image workspace.

7. *CropImage* Crops the image in the "Modified" image workspace to a subset of the original image.

8. *EmbedTo* This is the function to be tested. According to the function manual, it embeds the current image, "Modified", into the target image, "Orig", at the specified x, y, and z values of (10, 10, 10).

9. *SetWorkImage* Sets the current working image back to "Orig" so we can check that the target image of the EmbedTo function has indeed been modified.

10. *DisplayImage* Displays the target image so we can verify that the result of the operation is correct.

The output of this testing script can be seen in Figure 4.5. As can be seen, a cropped version of the original, "Modified", has clearly been embedded into a subset of the original image, "Orig".

Finally, we will look at the means by which Format category functions were tested.

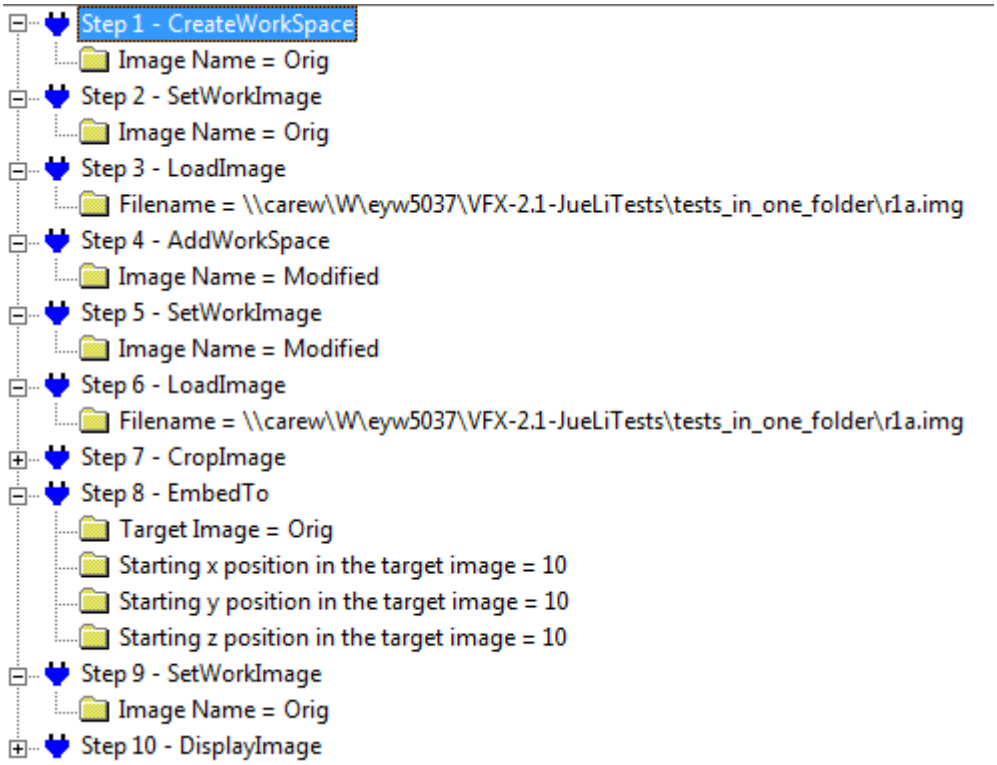


Fig. 4.4. A sample script to test the EmbedTo workspace function.

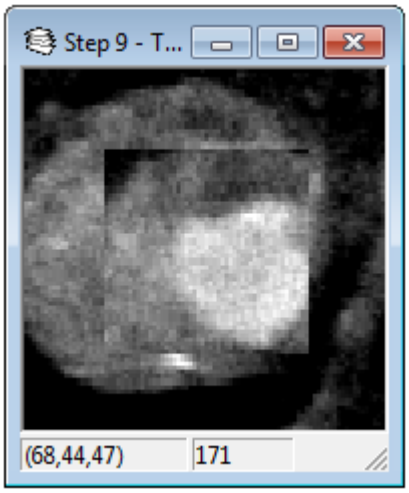


Fig. 4.5. A subset of image *r1a* embedded in another copy of *r1a*. This is the output of the test script in Figure 4.4.

4.4 Format Function Testing

Within the Format category, the LoadImage, LoadHeader, SaveImage, and CreateImage functions were modified and the LoadBMP function was added. While these functions operate only on the current workspace, similar to the general functions noted above, the SaveImage function outputs an image file to disk and the other functions import image files to VFX.

The load functions are used to import an image file at the beginning of a script. In the course of this project, we added the ability to load *.mhd and *.bmp files. The following procedure was used to test the Format category functions:

1. Load a file
    - Bitmap image
    - MetaImage
2. Apply other VFX functions and verify functionality
    - SaveImage function followed by subsequent LoadImage
    - binaryMorphology or grayscaleMorphology function

- medianFilter function

All test results are given in the next chapter.

Chapter 5

Results

This chapter summarizes the changes made to the categories and functions of VFX in the upgrade to version 2.2. First, we show the VFX 2.1 function tables from Jue Li's thesis in 2010, which provides a list of functions and test results from before this upgrade. Next, a series of tables illustrate the various changes made to these functions. Afterwards, we present tables showing the final state of VFX 2.2 at the conclusion of this upgrade. At the end of this section, sample runs for selected upgraded functions are given.

5.1 VFX 2.1 Function Tables and Test Results

In this section, we present the state of VFX 2.1's functions *prior to* the upgrades in this project. This provides a baseline reference that the next sets of tables will build off. The test results given below have been copied from Jue Li's thesis on VFX 2.1 in 2010.

The testing was performed on the binary-valued image "curve_branch_tube128", the grayscale (8-bit) image "r1a", and the 16-bit image "20349_3_3_B31." In the test results columns of the tables, the various entries mean the following:

- "Pass" means that the function worked for the specified data type.
- "Fail" means that the function did not work for the specified data type.
- "N/A" means that the function is not applicable to the specified data type. For example, "VFXgrayClose" is not applicable to a binary-valued image.

The phrase "This category was NOT upgraded" in the table caption means the category was not changed from VFX 2.1 to VFX 2.2. Thus, for these categories, the tables in this section also reflect the current state of these categories. The categories appear in the order in which they appear in VFX's function selection dialog.

Function	Binary Image Type	8-bit Image Type	16-bit Image Type
AddSkeleton	—	—	—
CreateImage	Pass	Pass	Pass
DisplayImage	Pass	Pass	Pass
ImportTr2	—	—	—
LoadHeader	Pass	Pass	Pass
LoadImage	Pass	Pass	Pass
LoadPath	—	—	—
LoadSkeleton	—	—	—
ResumeLoadImage	—	—	—
SampleLine	Pass	Pass	Pass
SaveImage	Pass	Pass	Pass
SaveImageNoCalGray	Pass	Pass	Pass
SavePath	—	—	—
SaveSkeleton	—	—	—
SaveThin	—	—	—
SaveTrc	—	—	—

Table 5.1. VFX 2.1 status of **Format** functions.

Function	Binary Image Type	8-bit Image Type	16-bit Image Type
AddWorkSpace	Pass	Pass	Pass
CreateWorkSpace	Pass	Pass	Pass
FreeWorkImage	Pass	Pass	Pass
RenameWorkImage	Pass	Pass	Pass
SetImageShared	Pass	Pass	Pass
SetWorkImage	Pass	Pass	Pass
WorkImage	Pass	Pass	Pass

Table 5.2. VFX 2.1 status of **Workspace** functions.

Function	Binary Image Type	8-bit Image Type.	16-bit Image Type
AnisotropicDiffusionFilter	Pass	Pass	Pass
GaussianBlurFiltering	Pass	Pass	Pass
LaplacianOfGaussian	Pass	Pass	Pass
MeanFiltering	Pass	Pass	Pass
MedianFiltering	Pass	Pass	Pass
SigmaFiltering	Pass	Pass	Pass
VFXanisoDiffuse	Pass	Pass	Pass
VFXaverageMax	Pass	Pass	Pass
VFXaverageMin	Pass	Pass	Pass
VFXdespike	Pass	Pass	Pass
VFXfilter4D	Need a 4D image input.		
VFXlowpass	Pass	Pass	Pass
VFXmaxHomog	Pass	Pass	Pass
VFXmedian	Pass	Pass	Pass
VFXsigma	Pass	Pass	Pass
VFXsymmetricNN	Pass	Pass	Pass
VFXwshedFilter	Need a 4D input image gradient.		

Table 5.3. VFX 2.1 status of **Filter** functions.

Function	Binary Image Type	8-bit Image Type	16-bit Image Type
BinaryMorphology	Pass	N/A	N/A
GrayscaleMorphology	N/A	Pass	Pass
VFX4Dmorphology	Need a 4D input image.		
VFXbinClose	Pass	N/A	N/A
VFXbinDilate	Pass	N/A	N/A
VFXbinErode	Pass	N/A	N/A
VFXbinOpen	Pass	N/A	N/A
VFXcondDilate	Need a 4D condition volume.		
VFXgrayClose	N/A	Pass	Pass
VFXgrayDilate	N/A	Pass	Pass
VFXgrayErode	N/A	Pass	Pass
VFXgrayOpen	N/A	Pass	Pass
VFXmax	Pass	Pass	Pass
VFXmin	Pass	Pass	Pass
VFXtopHat	Pass	Pass	Pass
VFXultimateErode	Fail	Fail	Fail

Table 5.4. VFX 2.1 status of **Morphology** functions.

Function	Binary Image Type	8-bit Image Type	16-bit Image Type
BorderBranchLabel	Pass	Pass	Pass
BranchLabeling	Pass	Pass	Pass
CavityDelete2D	Pass	Pass	Pass
CavityDeletion	Fail. *Fail when choosing recursive mode.*		
CenterlineTracking	Need a surface file.		
CheckBranchLabel	Fail	Fail	Fail
ComnnComp2D	Fail. *It can generate an output, but the output is wrong.*		
ConnectedComponentLabeling	Pass	Pass	Pass
FreeCheckBranchLabel	Pass	Pass	Pass
PrintBranchLabel	Pass	Pass	Pass
SetCenterlineTracking	Pass	Pass	Pass
SplineSkeleton	Pass	Pass	Pass
Thinning	Pass	Pass	Pass
Thinning2	Fail	Fail	Fail
ThinningSaha	Fail	Fail	Fail
VFXcavityDeletion_ajs	Fail	Fail	Fail
VFXconnComp2d	Fail	Fail	Fail
VFXconnComp3d	Fail.	Fail.	Fail.
VFXhomotopicThick	Pass	Pass	Pass
VFXhomotopicThin	Pass	Pass	Pass

Table 5.5. VFX 2.1 status of **Topology** functions.

Function	Binary Image Type	8-bit Image Type	16-bit Image Type
BrightRegionGrowing	Pass	Pass	Pass
FWHMRegionGrowing	Pass	Pass	Pass
HybridHalfMaximumRegionGrowing	Need the input, threshold and local-half-max images.		
HysteresisThresholding	Pass	Pass	Pass
LocalThresholdRegionGrowing	Pass	Pass	Pass
MaskingImage	Pass	Pass	Pass
RegionGrowing	Pass	Pass	Pass
RemoveOuterRegion	Fail. *It can generate an output, but the output is wrong.*		
SegmentGen	Need a input skeleton representation.		
VFXadaptiveThresh	Pass	Pass	Pass
VFXbinBordDetect	Pass	Pass	Pass
VFXbtRegionGrow	Fail	Fail	Fail
VFXcueRegionGrow	Fail for number of region above 2.		
VFXcueRelax	Need a cue filenames.		
VFXcueWatershed	Pass	Pass	Pass
VFXhystThreshold	Fail	Fail	Fail
VFXhystThreshSeed	Need a cue filenames.		
VFXkirschEdge	Pass	Pass	Pass
VFXmorphGradient	Pass	Pass	Pass
VFXnonmaxSuppress	Pass	Pass	Pass
VFXrelaxLabeling	environment variable ISEHOME not defined.		
VFXseedRegGrow	Fail	Fail	Fail
VFXsobel	Pass	Pass	Pass
VFXwshedCueRelax	Need cue filenames and gradient input volume.		
VFXwshedMarkerRelax	Need the gradient input volume.		

Table 5.6. VFX 2.1 status of **Segmentation** functions. This category was NOT upgraded.

Function	Binary Image Type	8-bit Image Type	16-bit Image Type
AddFrom	Pass	Pass	Pass
AddFromTo	Pass	Pass	Pass
AddTo	Pass	Pass	Pass
AddValue	Pass	Pass	Pass
Complement	Pass	Pass	Pass
ConvertImage	Pass	Pass	Pass
CopyFrom	Pass	Pass	Pass
CopyFromTo	Pass	Pass	Pass
CopyTo	Pass	Pass	Pass
CreateCrossSection	Need the segmented image name.		
CropImage_new	Pass	Pass	Pass
CropImage_NV	Fail	Fail	Fail
DiffFromImage	Pass	Pass	Pass
DiffImages	Pass	Pass	Pass
EmbedFrom	Pass	Pass	Pass
EmbedFromTo	Pass	Pass	Pass
EmbedTo	Pass	Pass	Pass
FixHeader	Pass	Pass	Pass
FlipImage	Pass	Pass	Pass
HistogramEqualization	N/A	Pass	Pass
Interpolation	Pass	Pass	Pass
MultFrom	Pass	Pass	Pass
MultFromTo	Pass	Pass	Pass
MultiplyValue	Pass	Pass	Pass
MultTo	Pass	Pass	Pass
ResizeImage	Pass	Pass	Pass
SetBinaryMinMax	N/A	Pass	Pass
SubFrom	Pass	Pass	Pass
SubFromTo	Pass	Pass	Pass
SubTo	Pass	Pass	Pass
Thresholding	Pass	Pass	Pass

Table 5.7. VFX 2.1 status of **Manipulation** functions (part 1).

Function	Binary Image Type	8-bit Image Type	16-bit Image Type
VFXassign	Pass	Pass	Pass
VFXcomplement	Pass	Pass	Pass
VFXdemaskRegion	Fail	Fail	Fail
VFXembedImage	Pass	Pass	Pass
VFXexpand	Pass	Pass	Pass
VFXflip	Pass	Pass	Pass
VFXgaussNoise	Pass	Pass	Pass
VFXgrayInversion	Pass	Pass	Pass
VFXhighZero	Pass	Pass	Pass
VFXimageMasking	Pass	Pass	Pass
VFXinterpOrlick	Fail	Fail	Fail
VFXinterpWEH	Pass	Pass	Pass
VFXlevelSlicing	Pass	Pass	Pass
VFXlinearInterp	Fail	Fail	Fail
VFXlowZero	Pass	Pass	Pass
VFXrangeLabel	Pass	Pass	Pass
VFXresize	Pass	Pass	Pass
VFXsaveLargest	Pass	Pass	Pass
VFXshade2D	Pass	Pass	Pass
VFXshrink	Pass	Pass	Pass
VFXthreshold	Pass	Pass	Pass
VFXumbra	Pass	Pass	Pass
VFXvolumeAlgebra	Need the “vol2” input parameter.		
VFXzeroPad	Pass	Pass	Pass
Volume Algebra	Need volume 1 and volume 2.		

Table 5.8. VFX 2.1 status of **Manipulation** functions (part 2).

Function	Binary Image Type	8-bit Image Type	16-bit Image Type
ComputeMedianGrayScale	Pass	Pass	Pass
Information	Pass	Pass	Pass
Profiling	Pass	Pass	Fail
SaveHistogram	Pass	Pass	Pass
SupportArea	Pass	Pass	Pass
VFX3Dhistogram	Fail	Fail	Fail
VFXerrorFunc	Need the second volume.		
VFXhistogram	Pass	Pass	Pass
VFXminBdCube	Pass	Pass	Pass
VFXregionProps	Need the grayscale volume.		
VFXvolProperties	Pass	Pass	Pass

Table 5.9. VFX 2.1 status of **Measurement** functions. This category was NOT upgraded.

Function	Binary Image Type	8-bit Image Type	16-bit Image Type
Time	Pass	Pass	Pass
Version	Pass	Pass	Pass

Table 5.10. VFX 2.1 status of **System** functions. This category was NOT upgraded.

Function	Binary Image Type	8-bit Image Type	16-bit Image Type
RenderSurface	Pass	Pass	Pass
RenderSurfaceWithSkeleton	Need input tree surface and input skeleton representation.		
SetSurfaceWriter	—	—	—
Skeleton2vtkFile	—	—	—
SurfaceGeneration	Pass	Pass	Pass
SurfaceWriter	—	—	—

Table 5.11. VFX 2.1 status of **VTK** functions. This category was NOT upgraded.

Function	Binary Image Type	8-bit Image Type	16-bit Image Type
LungMask	Pass	Pass	Pass
VFXangioAnalysis	Fail	Fail	Fail
VFXblobExtract	Fail	Fail	Fail
VFXconcaveGapFill	Fail	Fail	Fail
VFXlvExtract	Fail	Fail	Fail
VFXmicroCT	Fail	Fail	Fail
VFXregionIso	Fail	Fail	Fail
VFXregionSep	Pass	Pass	Pass
VFXrootRemoval	Fail	Fail	Fail

Table 5.12. VFX 2.1 status of **Turnkey** functions. This category was NOT upgraded.

Function	Binary Image Type	8-bit Image Type	16-bit Image Type
PruneGenHard	Invalid parameter: Generation ID		
PruneGenSoft	Invalid parameter: Generation ID		
Pruning	Need input skeleton.		
RootSearch	Need input skeleton representation.		
SkeletonRep	Fail	Fail	Fail
VFXbranchMeas	Fail	Fail	Fail

Table 5.13. VFX 2.1 status of **BranchFunc** functions. This category was NOT upgraded.

Function	Binary Image Type	8-bit Image Type	16-bit Image Type
FilterResponse	Pass	Pass	Pass
MaxResponse	Pass	Pass	Pass
TubularStructureDetection	Fail	Fail	Fail
VesselThresholder	Fail	Fail	Fail

Table 5.14. VFX 2.1 status of **TestingPurpose** functions. This category was NOT upgraded.

5.2 Overview of Category Changes

We have just looked at the list of functions in VFX 2.1 prior to any changes. The following set of tables present this same list of functions, but show the changes made to each of the functions during the course of the upgrade to VFX 2.2. Tables are only shown for categories which have changed; categories which were not considered in this project are left out.

The columns of these tables have the following meanings:

- "Examined" indicates that the function was considered for changes in this paper.
- "Upgraded" means that functionality, parameter bounding, or error checking was improved, or a bug was fixed.
- "Archived" means that the function was moved to the function archive.
- "Renamed" means that the function has been renamed or merged with a function of a different name. The new name of the function is provided.
- "Moved" indicates that the function has been moved to a different category. The new category name is provided.

The entries in the columns have the following meanings:

- "Y" means the the change applies to the current function.
- "Y*" is used to represent changes made by Ronnarit Cheirsilp.
- "-" means that the corresponding change does not apply.

The tables appear in the order in which the categories appear in VFX's function selection dialog.

VFX 2.1 Function	Examined	Upgraded	Archived	Renamed	Moved
AddSkeleton	Y	-	Y	-	-
CreateImage	Y	Y	-	-	-
DisplayImage	-	-	-	-	-
ImportTr2	Y	-	Y	-	-
LoadHeader	Y*	Y*	-	-	-
LoadImage	Y*	Y*	-	-	-
LoadPath	Y	-	Y	-	-
LoadSkeleton	Y	-	Y	-	-
ResumeLoadImage	-	-	-	-	-
SampleLine	Y	-	Y	-	-
SaveImage	Y*	Y*	-	-	-
SaveImageNoCalGray	-	-	-	-	-
SavePath	Y	-	Y	-	-
SaveSkeleton	Y	-	Y	-	-
SaveThin	Y	-	Y	-	-
SaveTrc	Y	-	Y	-	-

Table 5.15. Changes made to **Format** functions.

VFX 2.1 Function	Examined	Upgraded	Archived	Renamed	Moved
AddWorkSpace	Y	-	-	-	-
CreateWorkSpace	Y	-	-	-	-
FreeWorkImage	-	-	-	FreeWorkSpace	-
RenameWorkImage	-	-	-	RenameWorkSpace	-
SetImageShared	-	-	-	-	-
SetWorkImage	-	-	-	SetWorkSpace	-
WorkImage	-	-	-	WorkspaceName	-

Table 5.16. Changes made to **Workspace** functions.

VFX 2.1 Function	Examined	Upgraded	Archived	Renamed	Moved
AnisotropicDiffusionFilter	Y	-	-	-	-
GaussianBlurFiltering	-	-	-	-	-
LaplacianOfGaussian	-	-	-	-	-
MeanFiltering	Y	Y	-	-	-
MedianFiltering	Y	Y	-	-	-
SigmaFiltering	Y	Y	-	-	-
VFXanisoDiffuse	Y	-	Y	-	-
VFXaverageMax	-	-	-	-	-
VFXaverageMin	-	-	-	-	-
VFXdespike	-	-	-	-	-
VFXfilter4D	-	-	-	-	-
VFXlowpass	-	-	-	-	-
VFXmaxHomog	-	-	-	-	-
VFXmedian	Y	-	Y	-	-
VFXsigma	Y	-	Y	-	-
VFXsymmetricNN	-	-	-	-	-
VFXwshedFilter	-	-	-	-	-

Table 5.17. Changes made to **Filter** functions.

VFX 2.1 Function	Examined	Upgraded	Archived	Renamed	Moved
BinaryMorphology	Y	Y	-	BinaryMorph_Erode BinaryMorph_Dilate BinaryMorph_Open BinaryMorph_Close	-
GrayscaleMorphology	Y	Y	-	GrayscaleMorph_Erode GrayscaleMorph_Dilate GrayscaleMorph_Open GrayscaleMorph_Close	-
VFX4Dmorphology	-	-	-	-	-
VFXbinClose	Y	-	Y	-	-
VFXbinDilate	Y	-	Y	-	-
VFXbinErode	Y	-	Y	-	-
VFXbinOpen	Y	-	Y	-	-
VFXcondDilate	-	-	-	-	-
VFXgrayClose	Y	-	Y	-	-
VFXgrayDilate	Y	-	Y	-	-
VFXgrayErode	Y	-	Y	-	-
VFXgrayOpen	Y	-	Y	-	-
VFXmax	Y	-	Y	-	-
VFXmin	Y	-	Y	-	-
VFXtopHat	-	-	-	-	-
VFXultimateErode	Y	-	Y	-	-

Table 5.18. Changes made to **Morphology** functions.

VFX 2.1 Function	Examined	Upgraded	Archived	Renamed	Moved
BorderBranchLabel	-	-	-	-	-
BranchLabeling	-	-	-	-	-
CavityDelete2D	-	-	-	-	-
CavityDeletion	-	-	-	-	-
CenterlineTracking	-	-	-	-	-
CheckBranchLabel	-	-	-	-	-
ConnComp2D	Y	Y	-	ConnCompLabeling	-
ConnectedComponentLabeling	Y	Y	-	ConnCompLabeling	-
FreeCheckBranchLabel	-	-	-	-	-
PrintBranchLabel	-	-	-	-	-
SetCenterlineTracking	-	-	-	-	-
SplineSkeleton	-	-	-	-	-
Thinning	-	-	-	-	-
Thinning2	-	-	-	-	-
ThinningSaha	-	-	-	-	-
VFXcavityDeletion_ajs	-	-	-	-	-
VFXconnComp2d	-	-	-	-	-
VFXconnComp3d	-	-	-	-	-
VFXhomotopicThick	-	-	-	-	-
VFXhomotopicThin	-	-	-	-	-

Table 5.19. Changes made to **Topology** functions. Note: ConnComp2D and Connected-ComponentLabelling were merged to a single function, ConnCompLabeling.

VFX 2.1 Function	Examined	Upgraded	Archived	Renamed	Moved
AddFrom	Y	Y	-	-	Workspace
AddFromTo	Y	Y	-	-	Workspace
AddTo	Y	Y	-	-	Workspace
AddValue	Y	Y	-	-	-
Complement	-	-	-	-	-
ConvertImage	-	-	-	-	-
CopyFrom	Y	-	-	-	Workspace
CopyFromTo	Y	-	-	-	Workspace
CopyTo	Y	-	-	-	Workspace
CreateCrossSection	-	-	-	-	-
CropImage_new	Y	-	-	WorkspaceCrop	Workspace
CropImage_NV	Y	-	Y	-	-
DiffFromImage	Y	-	-	-	Workspace
DiffImages	Y	-	-	-	Workspace
EmbedFrom	Y	-	-	-	Workspace
EmbedFromTo	Y	-	-	-	Workspace
EmbedTo	Y	-	-	-	Workspace
FixHeader	-	-	-	-	-
FlipImage	-	-	-	-	-
HistogramEqualization	-	-	-	-	-
Interpolation	-	-	-	-	-
MultFrom	Y	Y	-	-	Workspace
MultFromTo	Y	Y	-	-	Workspace
MultiplyValue	Y	Y	-	-	-
MultTo	Y	Y	-	-	Workspace
ResizeImage	-	-	-	-	-
SetBinaryMinMax	-	-	-	-	-
SubFrom	Y	Y	-	-	Workspace
SubFromTo	Y	Y	-	-	Workspace
SubTo	Y	Y	-	-	Workspace
Thresholding	Y	Y	-	-	-

Table 5.20. Changes made to **Manipulation** functions (part 1).

VFX 2.1 Function	Examined	Upgraded	Archived	Renamed	Moved
VFXassign	Y	-	Y	-	-
VFXcomplement	Y	-	Y	-	-
VFXdemaskRegion	-	-	-	-	-
VFXembedImage	Y	-	Y	-	-
VFXexpand	-	-	-	-	-
VFXflip	Y	-	Y	-	-
VFXgaussNoise	-	-	-	-	-
VFXgrayInversion	Y	-	Y	-	-
VFXhighZero	Y	-	Y	-	-
VFXimageMasking	Y	-	Y	-	-
VFXinterpOrlick	-	-	-	-	-
VFXinterpWEH	-	-	-	-	-
VFXlevelSlicing	-	-	-	-	-
VFXlinearInterp	-	-	-	-	-
VFXlowZero	Y	-	Y	-	-
VFXrangeLabel	Y	-	Y	-	-
VFXresize	Y	-	Y	-	-
VFXsaveLargest	-	-	-	-	-
VFXshade2D	-	-	-	-	-
VFXshrink	-	-	-	-	-
VFXthreshold	Y	-	Y	-	-
VFXumbra	-	-	-	-	-
VFXvolumeAlgebra	-	-	-	-	-
VFXzeroPad	-	-	-	-	-
Volume Algebra	-	-	-	-	-

Table 5.21. Changes made to **Manipulation** functions (part 2).

5.3 VFX 2.2 Function Tables and Test Results

We have just looked at the original status of VFX 2.1's functions and the changes we have made to them. In this section, we overview the organization and functionality of VFX 2.2's functions *after* the conclusion of the project.

The first column, *Function*, lists the functions in the order they appear in the categories. The three *Tests* columns give the results of testing the functions using binary, grayscale, and 16-bit images as inputs. The test images used are the binary image "curve_branch_tube128", the grayscale image "r1a", and the 16-bit image "20349_3_3_B31." All test results in this section were performed as part of this upgrade *after* any changes were made. The entries in the Tests columns have the following meanings:

- "Pass" means that the function worked for the specified data type.
- "Fail" means that the function did not work for the specified data type.
- "N/A" means that the function is not applicable to the specified data type. For example, "VFXgrayClose" is not applicable to a binary-valued image.
- "-" means that the function was not tested. Refer to Jue Li's tests in the original function tables for the most recent test results.

The last four columns of these tables correspond to upgrades. The columns have the following meanings:

- "New" indicates that the function was newly introduced in VFX 2.2.
- "Upgraded" means that functionality, parameter bounding, or error checking was improved, or a bug was fixed.
- "Renamed" means that the function has been renamed or merged with a function of a different name. The former name of the function is not provided - see the previous section for details on the name change.
- "Moved" indicates that the function has been moved to a different category. The former category name is not provided - see the previous section for details on the name change.

The entries in the upgrade columns have the following meanings:

- "Y" means the the change applies to the current function.
- "-" means that the function was considered, but the change was not made.
- "Not considered" means that the function was not considered for changes in this project.

The tables will appear in the order in which the categories appear in VFX's function selection dialog. Categories which were not upgraded are not shown. For these categories, refer to their tables in Section 5.1 for their state. The last table shows the contents of the VFX 2.2 function archive.

	Tests			Changes From 2.1			
Function	Binary	Grayscale	16-bit	New	Upgrade	Rename	Move
LoadImage	Pass	Pass	Pass	-	Y	-	-
SaveImage	Pass	Pass	Pass	-	Y	-	-
DisplayImage	Pass	Pass	Pass	-	-	-	-
ResumeLoadImage	Pass	Pass	Pass	-	-	-	-
LoadBMP		– Pass –		Y	-	-	-
LoadHeader	Pass	Pass	Pass	-	Y	-	-
CreateImage		– Pass –		-	Y	-	-
SaveImageNoCalGray	Pass	Pass	Pass	-	-	-	-

Table 5.22. VFX 2.2 status of **Format** functions.

Function	Tests			Changes From 2.1			
	Binary	Grayscale	16-bit	New	Upgrade	Rename	Move
CreateWorkSpace	– Pass –			-	-	-	-
AddWorkSpace	– Pass –			-	-	-	-
FreeWorkImage	-	-	-	-	-	-	-
SetImageShared	-	-	-	-	-	-	-
SetWorkImage	Pass	Pass	Pass	-	-	-	-
RenameWorkImage	Pass	Pass	Pass	-	-	-	-
WorkImage	Pass	Pass	Pass	-	-	-	-
WorkspaceCrop	Fail	Fail	Fail	-	-	Y	Y
CopyFrom	Pass	Pass	Pass	-	-	-	Y
CopyTo	Pass	Pass	Pass	-	-	-	Y
CopyFromTo	Pass	Pass	Pass	-	-	-	Y
EmbedFrom	Pass	Pass	Pass	-	-	-	Y
EmbedTo	Pass	Pass	Pass	-	-	-	Y
EmbedFromTo	Pass	Pass	Pass	-	-	-	Y
SubFrom	Pass	Pass	Pass	-	Y	-	Y
SubTo	Pass	Pass	Pass	-	Y	-	Y
SubFromTo	Pass	Pass	Pass	-	Y	-	Y
AddFrom	Pass	Pass	Pass	-	Y	-	Y
AddTo	Pass	Pass	Pass	-	Y	-	Y
AddFromTo	Pass	Pass	Pass	-	Y	-	Y
MultFrom	Pass	Pass	Pass	-	Y	-	Y
MultTo	Pass	Pass	Pass	-	Y	-	Y
MultFromTo	Pass	Pass	Pass	-	Y	-	Y
DiffFromImage	Pass	Pass	Pass	-	Y	-	Y
DiffImages	Pass	Pass	Pass	-	Y	-	Y

Table 5.23. VFX 2.2 status of **Workspace** functions.

Function	Tests			Changes From 2.1			
	Binary	Grayscale	16-bit	New	Upgrade	Rename	Move
MedianFiltering	Pass	Pass	Pass	-	Y	-	-
MeanFiltering	Pass	Pass	Pass	-	Y	-	-
TentFiltering	Pass	Pass	Pass	Y	-	-	-
SigmaFiltering	Pass	Pass	Pass	-	Y	-	-
LaplacianOfGaussian	-	-	-	Not considered.			
GaussianBlurFiltering	-	-	-	Not considered.			
AnisotropicDiffusionFilter	Pass	Pass	Pass	-	Y	-	-
VFXaverageMax	-	-	-	Not considered.			
VFXaverageMin	-	-	-	Not considered.			
VFXdespike	-	-	-	Not considered.			
VFXfilter4D	-	-	-	Not considered.			
VFXlowpass	-	-	-	Not considered.			
VFXmaxHomog	-	-	-	Not considered.			
VFXsymmetricNN	-	-	-	Not considered.			
VFXwshedFilter	-	-	-	Not considered.			

Table 5.24. VFX 2.2 status of **Filter** functions.

Function	Tests			Changes From 2.1			
	Binary	Grayscale	16-bit	New	Upgrade	Rename	Move
binaryMorph_Erode	Pass	-	-	-	Y	Y	-
binaryMorph_Dilate	Pass	-	-	-	Y	Y	-
binaryMorph_Open	Pass	-	-	-	Y	Y	-
binaryMorph_Close	Pass	-	-	-	Y	Y	-
Maximum	Pass	Pass	Pass	Y	-	-	-
Minimum	Pass	Pass	Pass	Y	-	-	-
grayscaleMorph_Erode	Pass	Pass	Pass	-	Y	Y	-
grayscaleMorph_Dilate	Pass	Pass	Pass	-	Y	Y	-
grayscaleMorph_Open	Pass	Pass	Pass	-	Y	Y	-
grayscaleMorph_Close	Pass	Pass	Pass	-	Y	Y	-
VFX4Dmorphology	-	-	-	Not considered.			
VFXcondDilate	-	-	-	Not considered.			
VFXtopHat	-	-	-	Not considered.			

Table 5.25. VFX 2.2 status of **Morphology** functions.

Function	Tests			Changes From 2.1			
	Binary	Grayscale	16-bit	New	Upgrade	Rename	Move
ConnCompLabelling	Pass	N/A	N/A	-	Y	-	-
CavityDeletion	-	-	-	Not considered.			
CavityDelete2D	-	-	-	Not considered.			
Thinning2	-	-	-	Not considered.			
Thinning	-	-	-	Not considered.			
ThinningSaha	-	-	-	Not considered.			
branchLabeling	-	-	-	Not considered.			
checkBranchLabel	-	-	-	Not considered.			
printBranchLabel	-	-	-	Not considered.			
SplineSkeleton	-	-	-	Not considered.			
borderBranchLabel	-	-	-	Not considered.			
freeCheckBranchLabel	-	-	-	Not considered.			
setCenterlineTracking	-	-	-	Not considered.			
centerlineTracking	-	-	-	Not considered.			
VFXhomotopicThick	-	-	-	Not considered.			
VFXhomotopicThin	-	-	-	Not considered.			

Table 5.26. VFX 2.2 status of **Topology** functions.

Function	Tests			Changes From 2.1			
	Binary	Grayscale	16-bit	New	Upgrade	Rename	Move
AssignValue	Pass	Pass	Pass	Y	-	-	-
ImageMasking	Pass	Pass	Pass	Y	-	-	-
HighZero	Pass	Pass	Pass	Y	-	-	-
LowZero	Pass	Pass	Pass	Y	-	-	-
RangeLabel	Pass	Pass	Pass	Y	-	-	-
CropImage	Pass	Pass	Pass	Y	-	-	-
AddValue	Pass	Pass	Pass	-	-	-	-
MultiplyValue	Pass	Pass	Pass	-	-	-	-
ConvertImage	-	-	-	-	-	-	-
CreateCrossSection	-	-	-	-	-	-	-
FixHeader	-	-	-	-	-	-	-
FlipImage	Pass	Pass	Pass	-	-	-	-
HistogramEqualization	-	-	-	-	-	-	-
Interpolation	-	-	-	-	-	-	-
GrayInversion	Pass	Pass	Pass	Y	-	-	-
Thresholding	Pass	Pass	Pass	-	-	-	-
Complement	Pass	Pass	Pass	-	-	-	-
ResizeImage	Pass	Pass	Pass	-	-	-	-
SetBinaryMinMax	-	-	-	Not considered.			
VolumeAlgebra	-	-	-	Not considered.			
VFXdemaskRegion	-	-	-	Not considered.			
VFXexpand	-	-	-	Not considered.			
VFXgaussNoise	-	-	-	Not considered.			
VFXinterpOrlick	-	-	-	Not considered.			
VFXinterpWEH	-	-	-	Not considered.			
VFXlevelSlicing	-	-	-	Not considered.			
VFXlinearInterp	-	-	-	Not considered.			
VFXsaveLargest	-	-	-	Not considered.			
VFXshade2D	-	-	-	Not considered.			
VFXshrink	-	-	-	Not considered.			
VFXumbra	-	-	-	Not considered.			
VFXvolumeAlgebra	-	-	-	Not considered.			
VFXzeroPad	-	-	-	Not considered.			

Table 5.27. VFX 2.2 status of **Manipulation** functions.

Category	Function
Format	LoadPath
	LoadSkeleton
	AddSkeleton
	ImportTr2
	SaveTrc
	SavePath
	SaveSkeleton
	SampleLine
	SaveThin
Filter	VFXanisoDiffuse
	VFXmedian
	VFXsigma
Morphology	VFXbinClose
	VFXbinDilate
	VFXbinErode
	VFXbinOpen
	VFXgrayClose
	VFXgrayDilate
	VFXgrayErode
	VFXgrayOpen
	VFXmax
	VFXmin
	VFXultimateErode
Topology	ConnComp2D
	VFXcavityDeletion_ajs
	VFXconnComp2D
	VFXconnComp3D
Manipulation	CropImage_NV
	VFXcomplement
	VFXembedImage
	VFXflip
	VFXgrayInversion
	VFXhighZero
	VFXimageMasking
	VFXlowZero
	VFXrangeLabel
	VFXresize
	VFXthreshold

Table 5.28. List of archived functions.

In the following sections, we will provide sample runs to demonstrate some of the function upgrades made.

5.4 Sample Runs of Topology Functions

Major upgrades were made to the *ConnCompLabelling* function in VFX 2.2. Here, we will demonstrate many of the changes and upgraded features in the new *ConnCompLabelling* function. This function takes in a binary image and identifies distinct regions of connected (contiguous) foreground pixels according to a specified connectivity, and performs basic filtering and labelling on these regions. The first step is to set up the input data.

To use the function, we need a binary image as the input. This binary image *must* have values of only 0 and 1. All grayscale values not equal to one would be considered as part of the background, not the foreground. In other words, using a binary image with discrete values 0 and 255 as the input would not be correct; this image would also need to be thresholded prior to being run. In Figure 5.1, we see the process for converting a sample image, *r1a*, from grayscale to binary and displaying the result. The threshold used in thresholding, 128, was chosen to provide a good balance between the foreground and background regions of the image. In Figure 5.2, we see the thresholded image with no modifications.

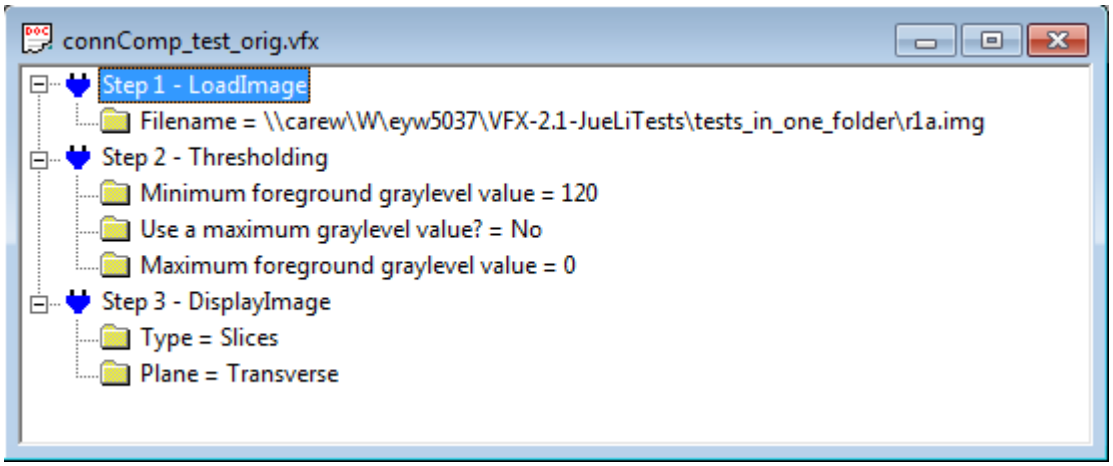


Fig. 5.1. The script to threshold grayscale image *r1a* to a binary image.

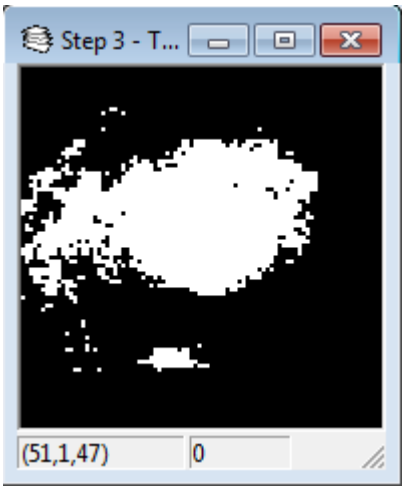


Fig. 5.2. A slice of image *r1a* after thresholding. This is the result of the script in Figure 5.1.

Next, we introduce the *ConnCompLabelling* function to the script. We are using the default values for Connectivity, Minimum size of a region, Max number of regions, and Save labels. However, we have set the "2D or 3D" parameter to 3D; we will test the 2D option further below.

From Figure 5.4, it can be seen that compared to the original thresholded image in Figure 5.2, many of the smaller regions have been removed; this is due to the fact that we have specified a minimum region size of 10 voxels. Thus, all regions that have fewer than ten voxels have been removed. Because we are performing 3D connected component labelling, there remain regions which have fewer than ten pixels on the current slice, as they may have voxels connected on neighboring slices. Because we are using the default value of 4/6 connectivity, this means that only sets of voxels that share an entire face with another voxel in the set are considered as a region. The 4 refers to the 4 edges of a 2D pixel, while the 6 refers to the 6 faces of a 3D voxel. If we had instead chosen 8/26 connectivity, connected regions of voxels could have included foreground voxels sharing only a single vertex with each other.

Next, we will illustrate the effect of changing the "Minimum size of a region" parameter. This parameter removes all connected components, or regions, that contain fewer

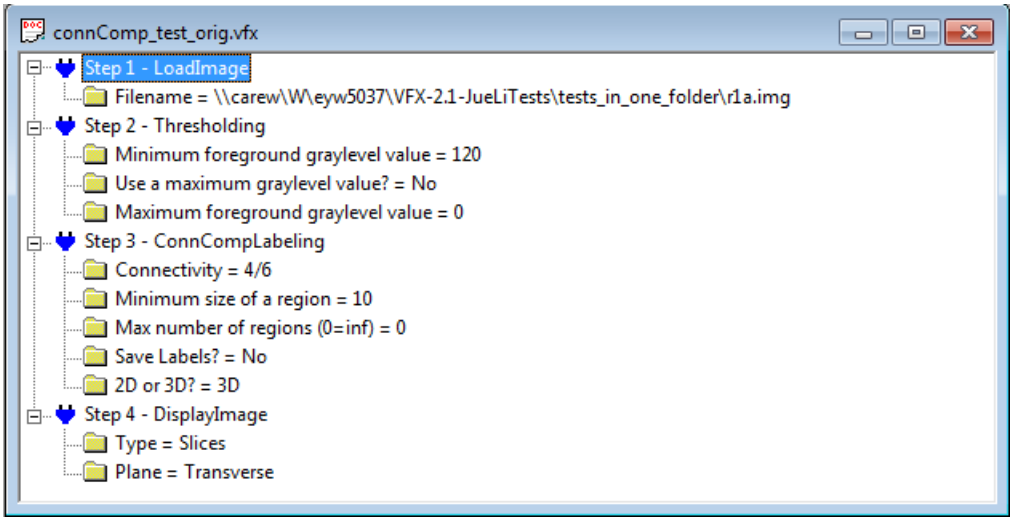


Fig. 5.3. The script to perform 3D connected component labelling with default parameters.

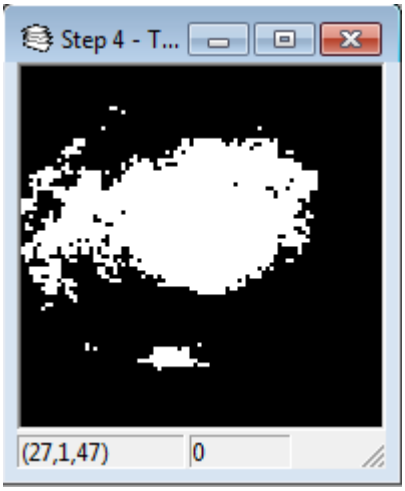


Fig. 5.4. A slice of image *r1a* after connected component labelling. This is the result of the script in Figure 5.3.

than the specified number of voxels. Thus, by decreasing this value, we identify and retain more regions, or by increasing this value, we filter out and remove more regions.

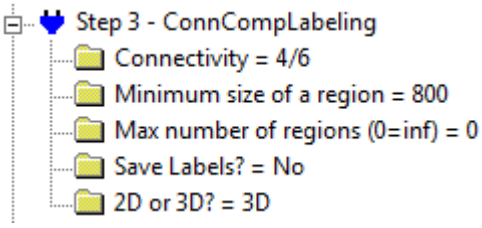


Fig. 5.5. The parameters of ConnCompLabelling with an increased minimum region size.

In Figure 5.5, we have changed the value of the "Minimum size of a region" parameter to 800. The result of this change can be seen in Figure 5.6. As expected, because very few regions are able to meet this criteria, all but the largest regions have been removed. While the entire 3D image is not visible in the included figure, by scrolling through the slices of the image, it can be seen that only about six to eight regions have been retained in the output image.

In VFX 2.1, the only way to filter out smaller regions in the *ConnectedComponent-Labelling* function was to use the above mentioned parameter to filter our regions below a specified size. However, this creates an arbitrary number of regions, and there was no way to directly specify the desired number of regions. In VFX 2.2, we have introduced the "Max number of regions" parameter. By default, the parameter takes the value of 0, which effectively disables the feature by allowing for an infinite number of regions. This was the case for the two prior sample runs. By specifying a finite nonzero value, the user can specify the maximum number of regions in the final output. If the number of regions meeting the other criteria is less than this maximum, this parameter does nothing; if the number of regions exceeds this maximum, this feature removes the smallest regions, and keeps the specified number of the largest regions.

In Figure 5.7, we have changed the value of the "Max number of regions" parameter to 3. We have also changed the value of the "Minimum size of a region" parameter back to 10. The result of this change can be seen in Figure 5.8. As can be seen, even though

Fig. 5.6. A slice of image *r1a* after connected component labelling with an increased minimum region size. This is the result of the script in Figure 5.5.

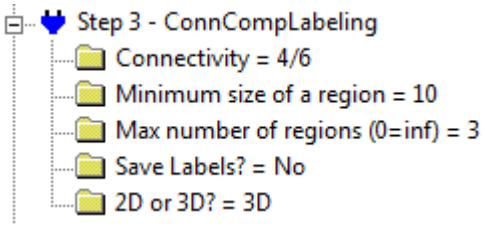


Fig. 5.7. The parameters for ConnCompLabeling with a maximum of three regions.

Fig. 5.8. A slice of image *r1a* after connected component labelling with a maximum of three regions. This is the result of the script in Figure 5.7.

the "Minimum size of a region" has been returned to its value in Figure 5.4, the number of regions is the same as in Figure 5.6. This is because we have only allowed for three regions to be displayed. Thus, even if there are more than three regions which satisfy the minimum size, only the three largest regions will appear.

Next, we will demonstrate the save labels functionality of the new *ConnCompLabelling* function. While this was a feature of the *VFXConnComp2D* and *VFXConnComp3D* functions, it was missing from the *ConnectedComponentLabelling* function in VFX 2.1. As a result, there was no way to use the graylevel to identify unique regions from each other and count the total number of regions. In VFX 2.2, if the user chooses to save labels, the output is grayscale instead of binary; each distinct region is assigned a unique grayscale value. The background is assigned to -1 and all foreground regions are numbered from 0 to the total number of regions displayed.

In Figure 5.9, we have enabled the saving of labels. The result of this change can be seen in Figure 5.10. Also, we have selected a different slice to view which contains a greater number of regions, in order to better demonstrate the result of our change. As can be seen, each region has been assigned a different graylevel, and thus appears as a different shade of gray in the output image. The background is the darkest color, having the value of -1. By looking at the maximum gray value of the image, we can find the total number of regions.

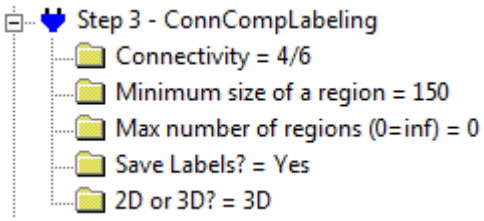


Fig. 5.9. The parameters of ConnCompLabelling set to save region labels.

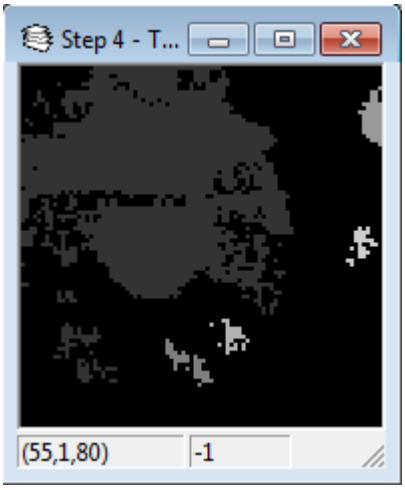


Fig. 5.10. A slice of image *r1a* after performing connected component labelling with region labels saved. This is the result of the script in Figure 5.9.

Lastly, we will demonstrate the 2D functionality of the *ConnCompLabelling* function. This function was merged with the *ConnComp2D* function that existed in VFX 2.1. It is important to note the differences from the 3D connected component labelling function.

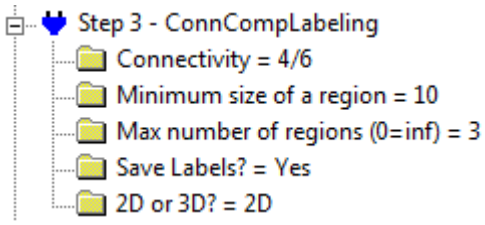


Fig. 5.11. The parameters of ConnCompLabeling set to perform 2D region analysis.

Fig. 5.12. A slice of image *r1a* after 2D connected component analysis. This is the result of the script in Figure 5.11.

In Figure 5.11, we have chosen to perform 2D labelling with a maximum of 3 regions. The result of this change can be seen in Figure 5.12. The important thing to note is that instead of applying to the entire image, the parameters now only apply to each slice. In other words, the maximum of 3 regions now means that three regions are allowed *per slice*; there could be up to 300 distinct regions in an image with 100 slices. Also, the specified minimum region size means that each region must have ten pixels in the *current slice*, whereas before, regions could span dozens of slices. Therefore, much lower minimum sizes are required to achieve a desired output. Also, the user should be aware that the labelled regions only have unique labels within a single slice. Distinct regions in separate slices could share the same label, as the algorithm only considers a single slice at a time.

We have now demonstrated some of the added features of the ConnCompLabeling function. Next, we will demonstrate some of the Morphology functions.

5.5 Sample Runs of Morphology Functions

Another set of functions to which significant modifications were made are the morphology functions. Upgrades were made to the binary morphology functions, and 16-bit maximum and minimum functions were introduced. Here, we will show some sample runs of the binary morphology dilation function. The erosion function is the inverse operation of the dilation function, and the opening and closing functions perform a combination of erosion and dilation. Opening performs erosion and then dilation, while closing performs dilation and then erosion. We will begin by showing the input test data.

Here, we will display the image without any modifications. The binary morphology images run on any images, including grayscale ones; it simply treats all values of 0 as the background, and all values 1 or above as the foreground. Therefore, unlike the connected component labelling function, any binary image with a background level of 0 can be used without prior thresholding.

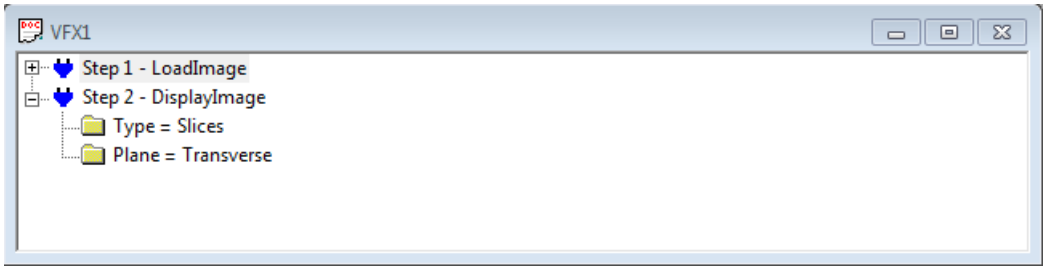


Fig. 5.13. The script to load and display the image curve_branch_tube128, to be built upon in the following sample runs.

In Figure 5.13, we have loaded and displayed the sample test image *curve_branch_tube128*. This is a binary image. The displayed image can be seen in Figure 5.14.

First, we will run the binary morphology dilation function using the default parameters. By default, the function operates with a window size of $3 \times 3 \times 3$ with a rectilinear structure, which is a rectangular prism shaped mask with the specified dimensions. The rectilinear structure of size $3 \times 3 \times 3$ occupies all of the 26-neighbors of the voxel of consideration.

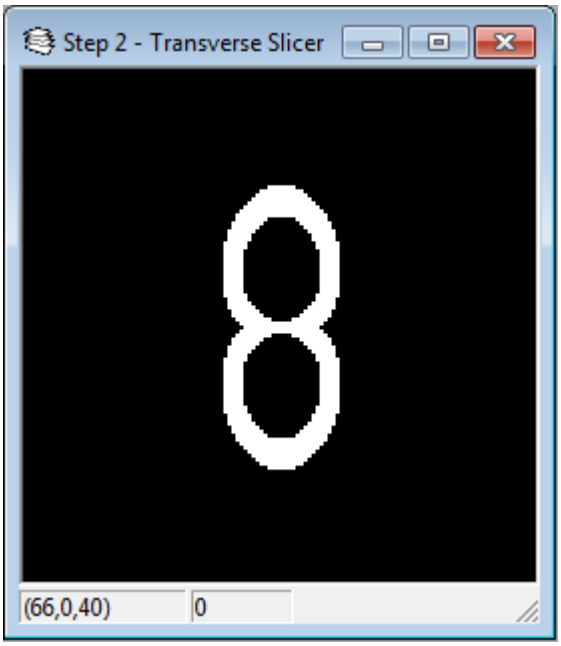


Fig. 5.14. A slice of image *curve_branch_tube128* with no modifications. This is the result of the script in Figure 5.13.

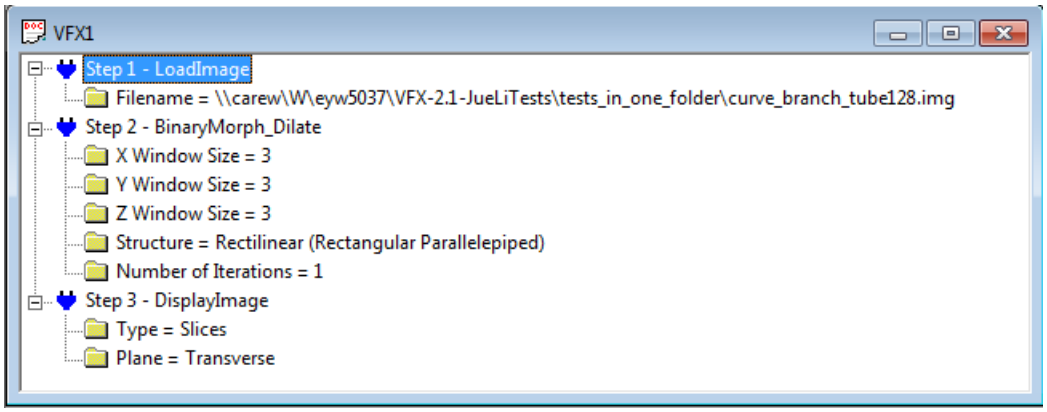


Fig. 5.15. The script to run binary morphology dilation on image *curve_branch_tube128* with the default parameters.

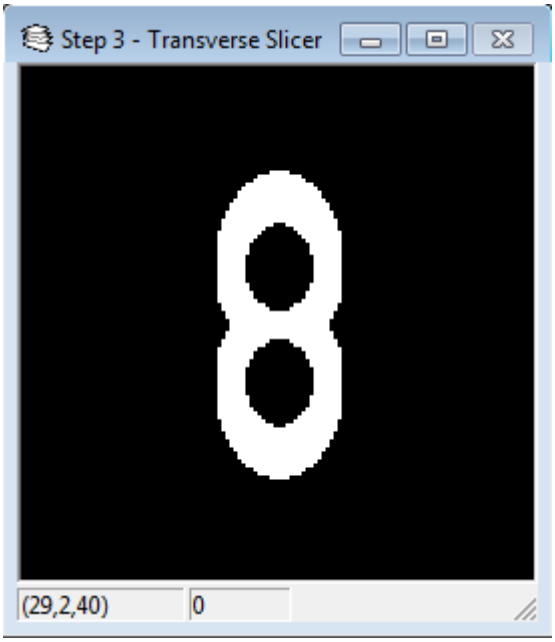


Fig. 5.16. A slice of image *curve_branch_tube128* after 3 × 3 × 3 binary morphological dilation. This is the result of the script in Figure 5.15.

In Figure 5.15, we have performed binary morphology dilation on the test image. The image after this processing has been done can be seen in Figure 5.16. As can be seen, the dilation has 'expanded' the white (foreground) portion of the image has been expanded. More specifically, all 26-neighbors of the original foreground in Figure 5.14 have now been added to the foreground. The erosion operation would have performed the same operation on the background of the image, expanding the black background region instead of the white foreground region.

Next, we will modify the window size parameter. By expanding the size of the morphology mask, we will produce a greater dilation effect in a single iteration of the process. In Figure 5.17, we have increased the window size to an $11 \times 11 \times 11$ rectilinear mask. This is a rectangular prism shaped mask that extends 5 voxels outwards from the voxel of consideration in every direction.

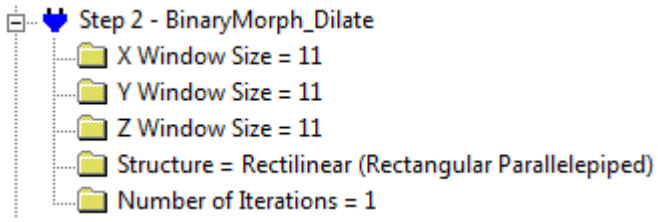


Fig. 5.17. The paramaters of BinaryMorph_Dilate with a larger $11 \times 11 \times 11$ morphology mask.

In Figure 5.18, we see the result of morpholoy using an $11 \times 11 \times 11$ rectilinear mask. This has the effect of adding all voxels within 5 voxels in all directions from any existing foreground voxel to the foreground. For instance, a single pixel of the initial foreground would add all pixels up to 5 slices, 5 rows, and 5 columns away to the final foreground. As can be seen, the white foreground region of the image has been significantly expanded relative to the original image and the image after the $3 \times 3 \times 3$ mask dilation. The extent of the dilation has also caused the kinks on either side of the foreground shape to completely removed.

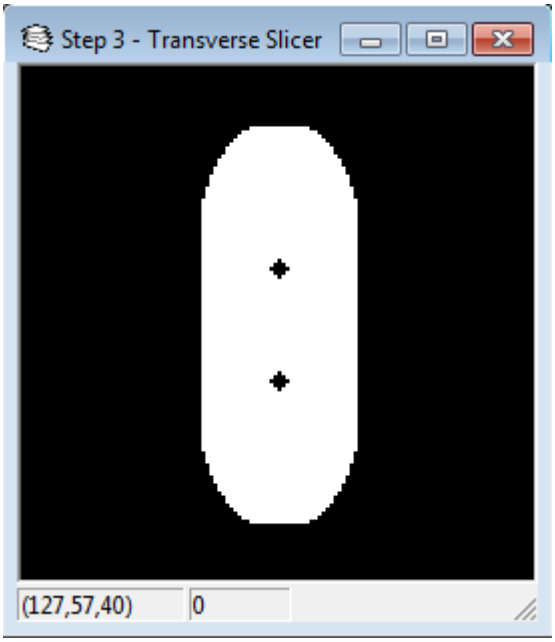


Fig. 5.18. A slice of image *curve_branch_tube128* after 11 × 11 × 11 binary morphological dilation. This is the result of the script in Figure 5.17.

Finally, we will explore the other available morphology masks. One mask available in VFX 2.1 was the 3D 6-connected mask; this is a mask that occupies a $3 \times 3 \times 3$ window, but instead of including all 26 neighbors, it only includes the 6 neighbors that share a face with the voxel of interest. One mask that was introduced in VFX 2.2 is the 2D 4-connected mask. This allows the user to specify a 2D mask that only consists of the 4 voxels which share a face with the voxel of interest on the same slice (z value).

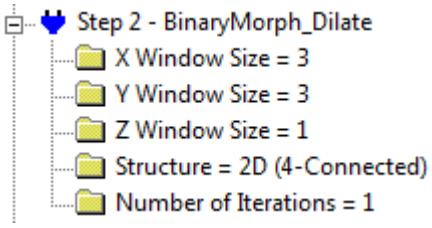


Fig. 5.19. The paramaters of BinaryMorph_Dilate to perform morphology with a 2D 4-connected mask.

In Figure 5.19, we have performed binary morphology dilation using a 2D 4-connected mask on the test image. Note that the dimensions of the morphology mask have been changed to $3 \times 3 \times 1$, the dimensions of the 4-connected window. The image after this processing has been done can be seen in Figure 5.20. When comparing this image with the result of the $3 \times 3 \times 3$ dilation in Figure 5.16, we can see a distinct difference. The result of this dilation has added exactly 1 pixel horizontally and 1 pixel vertically to the foreground when compared to the original image in Figure 5.14, while the $3 \times 3 \times 3$ dilation has added not only all the 8-connected pixels on the same slice, but also all diagonally and vertically connected voxels from neighboring slices. It is clear from the slice visible that the 2D 4-connected mask has resulted in significantly less dilation than the $3 \times 3 \times 3$ rectilinear mask, despite the fact that they technically have the same window size in the x and y directions.

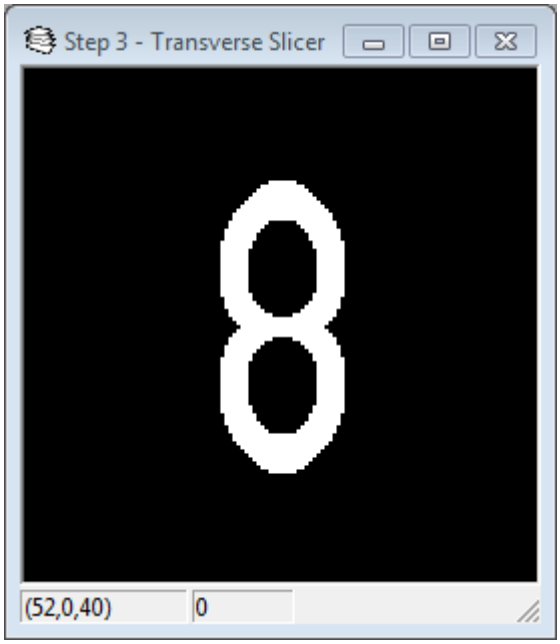


Fig. 5.20. A slice of image *curve_branch_tube128* after 2D 4-connected binary morphological dilation. This is the result of the script in Figure 5.19.

Chapter 6

Discussion and Future Work

6.1 Discussion

In this paper, we made significant progress in updating VFX to a 16-bit standard. A function archive has been introduced to separate out older functions, and nearly 40 functions have been archived. Many older 8-bit functions have been replaced by equivalent 16-bit image processing functions. Around 25 functions have been upgraded. These functions have been given more powerful functionality, more diverse parameter options, and greater robustness in error detection and tolerance. Finally, over ten completely new functions have been introduced to VFX, some providing entirely new functionality and some re-implementing functions which only operate on 8-bit images. Additional miscellaneous improvements have been made in the program's code organization and user interface.

6.2 Future Work

While much progress was made in this upgrade of VFX, there remains significant work to be done to upgrade VFX to a fully 16-bit image processing toolbox. Additionally, there are other issues that were not addressed due to time constraints.

The following are outstanding issues to be addressed in future upgrades of VFX:

- *Continue upgrading functions.* Within the categories worked on in the preceding sections, there still remains many 8-bit functions that have not been upgraded or archived. In the above results section, a listing of functions that were not considered in the current upgrade can be found; the vast majority of these are functions derived from VFX 1.0 for which 16-bit equivalents have not been implemented. The following categories received no attention in terms of upgrades in this paper:
  - Segmentation
  - Measurement
  - System

  - VTK
  - Turnkey
  - BranchFunc

- *Reorganize the code.* While some progress was made in reorganizing the code in the file *nvsharedcommand.h*, there remains many functions which should be moved to separate files. Other potential improvements are possible, including completely reorganizing the function initialization process to take place within the function files themselves; only a handful of functions have had this change implemented.

- *Fix the debug mode.* When run in debug mode, VFX is extremely unstable, tending to cause faults upon script runs. In the future, an effort should be made to identify the source of these crashes. Solving this problem will allow for much more effective debugging of future implemented functions by allowing for the use of Visual Studio's debug tools.

- *Improve the display image dialog.* When presented with small images with fewer than 200 columns, it is impossible to move the window without first resizing it because no portion of the top bar is visible. The current implementation of the image display dialog doesn't allow for an artificial border to be inserted, as the window automatically takes on the size of the image. In a future iteration of VFX, the display image dialog should be modified to prevent

References

[1] M. W. Graham, J. D. Gibbs, D. C. Cornish, and W. E. Higgins. Robust 3-D Airway Tree Segmentation for Image-Guided Peripheral Bronchoscopy. *IEEE Transactions on Medical Imaging*, 29(4):982–997, April 2010.

[2] W. E. Higgins, N. Chung, and E. L. Ritman. Extraction of left-ventricular chamber from 3-D CT images of the heart. *IEEE Transactions on Medical Imaging*, 9(4):384–395, December 1990.

[3] J. Li. *Graphical User Interface System for 3D Medical Image Processing.* Master's thesis, The Pennsylvania State University, Summer 2010.

[4] A. J. Sherbondy and W. E. Higgins. *PC VFX Manual.* The Pennsylvania State University, 2001.

[5] G. R. Simon. *Object-Oriented System for Multidimensional Image Processing.* Master's thesis, The Pennsylvania State University, May 1997.

[6] G. Sundaramoorthy, J. D. Hoford, E. A. Hoffman, and W. E. Higgins. IMPROMPTU: A System for Automatic 3D Medical Image-Analysis. *Computerized Medical Imaging and Graphics*, 19(1):131–143, January-February 1995.

[7] R. D. Swift, A. P. Kiraly, A. J. Sherbondy, A. L. Austin, E. A. Hoffman, G. McLennan, and W. E. Higgins. Automatic axes-generation for virtual bronchoscopic assessment of major airway obstructions. *Computerized Medical Imaging and Graphics*, 26(2):103–118, March-April 2002.

[8] S. Y. Wan. *Analysis and Visualization of Large Branching Networks in 3D Digital Images.* PhD thesis, Penn State University, May 2000.

[9] S. Y. Wan, K. C. Yu, and W. E. Higgins. *"NV" Manual for Users and Developers.* The Pennsylvania State University, September 2001.

[10] A. C. Weitz. *Computer-Based System for Multidimensional Medical Image Processing.* BS thesis, The Pennsylvania State University, Fall 2004.

Appendix A

Network Location of VFX

In this appendix, we provide the location of VFX 2.2 and the VFX 2.2 Function Manual on the MIPL network. We first provide the final location of VFX 2.2 and its source code. We also give the location of the SVN repository, which stores the entire version history of this upgrade.

In the next section, we give the final location of the VFX 2.2 Function Manual and the associated LaTeX source files. An SVN repository was also used in the development of the function manual, and the location of that is provided as well.

A.1 VFX 2.2

A final copy of the code and compiled release executable is located on RIPKEN at this location:

`S/VFX/VFX-2.2/`

During development, an SVN repository was used for version control. It can be found on CAREW at:

`W_1TB/eyw5037/SVN/VFX`

A.2 VFX 2.2 Function Manual

A final copy of the function manual has been stored on RIPKEN at this location:

`T/Manuals/VFX/VFX-2.2-functions`

During development, an SVN repository was used for version control. It can be found on CAREW at:

`W_1TB/eyw5037/SVN/VFX Function Manual`

Academic Vita of Edward Wang

Name: Edward Wang

Address: 58 Dutch Meadows Dr
Cohoes, NY 12047

Email Address: eyw5037@psu.edu

Education:
University: The Pennsylvania State University, University Park, PA
Expected Graduation: May 2013
Major: B.S., Electrical Engineering and Computer Engineering
Minor: Physics

International Education:
School: Royal Institute of Technology
Location: Stockholm, Sweden
Date: Spring 2012

Honors and Awards:
- Schreyer Travel Ambassador Grant
- Cisco Systems Scholarship
- 4-year Penn State Engineering Scholarships

Professional Experience:
Company: Intel Corporation
Date: May to August, 2012
Title: Validation Intern

Company: GE Transportation
Date: May to August, 2011
Title: Reliability Intern

Company: PPL Corporation

Date: May to August, 2010

Title: Information Technology Intern

Research Interests:

- Image Processing
- Integrated Circuits