WARSAW UNIVERSITY OF TECHNOLOGY

FACULTY OF MATHEMATICS
AND INFORMATION SCIENCE

BACHELOR THESIS
COMPUTER SCIENCE

Advanced Image Editor

Author: Marek Foss
Supervisor: Krzysztof Mossakowski, MSc

WARSAW APRIL 2007

Supervisor's Signature

Author's Signature

Summary

Advanced Image Editor named Fedit is written in C++ using Windows API and GDI+ graphic libraries. It follows a set of five rules to benefit the end user: no installers, no archives, no registry keys, no additional runtimes and a single executable file. The result is a program that is ready to work without the need of installation, can be run on systems with limited privileges and consumes small amounts of resources.

It is capable of basic image editing, as well as has implemented advanced tools and workspace enhancements. These include the Magic Wand selection based on color similarity; image filtering using built-in matrix filters or created by the user; multilayer documents with various layer attributes, like transparency, visibility and flexible ordering; complete workflow history capable of returning to any state of work.

Fedit Image Editor is designed to read and write the most popular image formats like JPG, PNG, TIFF, BMP, GIF and ICO. It can also browse through compressed ZIP and RAR archives. An internal file format FED was implemented to save workspace state, including layers, their properties and workspace settings. Minimal system required is Microsoft® Windows® with GDI+ support.

The following paper describes the project of Fedit. Firstly it focuses on presenting the application tools and features. Secondly, analyses the problems connected to the implementation of such programs, and covers with details the subject of working speed and memory consumption. Finally, explains the framework, which was created for the application, including a careful look at the objects and their usage, class hierarchy, object grouping and relationships between each framework component.

Contents

Contents i

1 Introduction 1

2 Fedit Overview 3
- 2.1 Fapplication Principles . . . 3
- 2.2 Fedit Features . . . 4
  - 2.2.1 File Management . . . 4
  - 2.2.2 User Interface . . . 5
  - 2.2.3 Tools . . . 7
  - 2.2.4 Main Menu . . . 8
  - 2.2.5 Matrix Filters . . . 10

3 Analysis of Implementation Problems 11
- 3.1 Speed of Editing Tools . . . 12
- 3.2 Memory Consumption . . . 16

4 Framework and Implementation Details 19
- 4.1 Application Environment . . . 19
- 4.2 Framework Description . . . 21
  - 4.2.1 Objects Overview . . . 21
  - 4.2.2 Internal Common Controls . . . 24
  - 4.2.3 Fedit Document (.FED) . . . 26

5 Conclusion 29

List of Figures 31

Chapter 1

Introduction

The purpose of this project is to create an advanced image editing application. The keyword "advanced" does not only mean the ability to perform various image manipulation, but also the way the general user interface works and enhances the user's experience with the program. Final product should enable image editing on intermediate level, covering basic features like drawing, erasing, copying, and saving to standard image file formats, as well as implementing advanced tools of selecting, color choosing, text editing, matrix filtering, multilayer management etc.

The project aims at delivering a quality software that merges the best features of other similar solutions. Although image processing market seems to be filled with powerful applications, there is an empty space in the area of freeware, well designed and user friendly editors. On one side there are the industry standard Adobe® Photoshop® and Corel® Paint Shop Pro® that rise the bar with each new version, on the other are the free solutions like Paint.NET, which despite being a very good software, has serious problems with resource consumption, and GIMP, which Linux heritage does show in the user interface.

Ideal application would aim at functionality of Photoshop® and its interface while being light, simple and freeware. This is the essence of the problem, and this is the aim of this project. The name of the application is - **Fedit Image Editor**.

Chapter 2

Fedit Overview

2.1 Fapplication Principles

Fedit Image Editor is the second application under the domain of Fapplication.org. The first was the Fiew Image Viewer, a sequential image file browser. Fedit inherits several parts from its predecessor, however the most important ones are the principles that act as laws in the design of this application:

- No installers
  The application should be provided to the user without the need of installing it
- No archives
  The application should not be compressed into a file archive
- No registry keys
  The application should not set nor leave any keys in the system registry
- No additional runtimes (ie. .NET, Java Runtime)
  The application should be able to run on a basic installation of the lowest compatible system (any Windows® with GDI+ support, in the case of this project)
- Single .exe file
  The application should be distributed as a single, working-ready .exe file

The above principles where firstly inspired by the idea of the application management on Apple® Mac OS X® systems, where installation or deleting of any

program is an act of dragging its (single!) icon onto the Programs folder, or Trash, if deleting. Moreover, there are other benefits of applying such principles. Firstly, the application does not fill the operating system with unnecessary data, especially registry keys. Secondly, the application field of work is enhanced greatly because of the ability to work on a read-only CD-ROM or USB stick, on a computer where the user has no privileges for installing programs or any other environment with limited access. Thirdly, because of creating single file applications, the data overhead from distributing into several files or using installers etc. is shrunk which effectively produces very small executables. Finally, the program is easily available for the most inexperienced computer software users and does not require anything else than double-clicking to start working.

However, these principles that greatly benefit the end user, bring similarly great difficulties for the developer. No framework-dependant solutions, like Microsoft® .NET or Java Runtime can be used, which in the Microsoft® Windows® environment implies the use of Windows API. No registry keys are allowed to be set in the system, thus to provide application settings (if any are required) configuration files have to be used, however the program has to work also without them. Similarly, if any additional files are required for full functionality, they should be contained in the executable, but upon an inability to extract them the application should still work with most of its functionality.

Despite these facts, the specified approach for creating applications, although difficult, is the best for the benefit of the end user.

2.2 Fedit Features

2.2.1 File Management

Fedit is an application that uses the Multiple Document Interface, therefore the user can open and simultaneously edit many images. Acceptable file formats are:

- The six most popular image formats: JPEG, PNG, GIF, BMP, TIFF, ICO
- The two most popular compression formats: RAR (CBR), ZIP (CBZ)
- The Fedit Document: FED

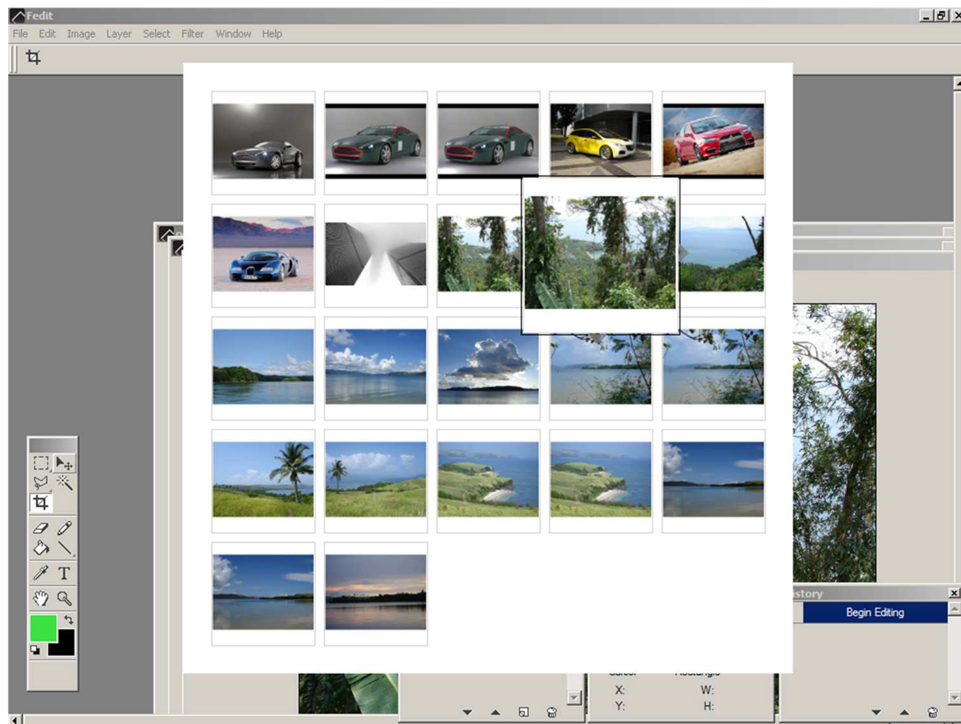


Figure 2.1: Browsing folder contents with thumbnail viewer.

The user can open multiple files at once with the common open file dialog box. The user can also open a compressed archive or a whole folder for browsing, and in this case a thumbnail browser dialog is displayed with the contents of the selected location. Using the mouse wheel or keyboard arrows the user can select the desired image and open it, and if necessary can browse further and open other images. Escape key or right mouse button closes the dialog and the user can begin the editing of the just opened images.

User can save the image in every of the mentioned popular image formats, as well as save the whole workspace along with layer information as a FED file, i.e. to continue editing later on.

2.2.2 User Interface

When at least one image is opened, the tool windows displaying Layers, History and general Information are activated. Also, the main menu items are enabled, and choosing a Tool from the Control Center affects the contents of the Dock Window.

Layers Tool Window Using the Layers tool window, the user can add new layers or delete them. Using the arrow buttons the layers can be ordered. The user can toggle the visibility of any layer, as well as its name (by double-clicking on the text displaying that name). Locking the layer prevents from performing actions using certain Tools, which is indicated by a special cursor icon when it is hovering above the workspace window. Changing opacity alters the level of transparency of the selected layer. The edit field of the opacity value is one of the many fields in Fedit which value can be changed also using keyboard arrow keys - Up and Down.

Info Tool Window The general Information tool window shows several workspace information basing on the position of the cursor - the color of the currently hovered pixel, both in RGB and CMYK color space, the XY position of the pixel, in workspace coordinates, and if selection exists - the size of the bounding rectangle of that selection.

History Tool Window The History tool window displays all the actions performed by the user, since the beginning of the editing. Text descriptions are accompanied by icons (if any) that represent the applied tool. By clicking on the arrow buttons or directly on the History labels, the user can move along the actions and eventually undo or redo the editing process.

Dock Window The contents of the Dock Window are connected to the current choice of an editing Tool. Some Tools may not have any configurable contents, then the Dock Window will only display the icon of the currently selected Tool. The Dock Window can be grabbed and moved which eventually may toggle the docking state of that window. The Dock Window at any given moment can be docked either on top or on the bottom or can float freely.

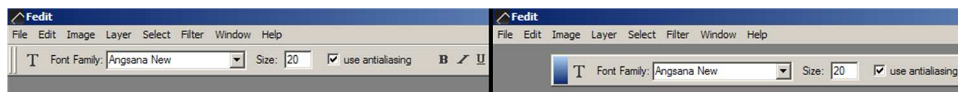


Figure 2.2: Dock Window in different docking states.

Tool Control Center In the Control Center the user can choose any editing Tool and use it to edit the image. As mentioned earlier, this choice may also affect

the Dock Window contents, if the selected Tool is configurable. The buttons that contain a small arrow at the bottom right corner contain also other, similar tools - to see these other tools the user has to either click the right mouse button or hold down the left mouse button. The bottom of the Control Center contains the foreground and background color choosers that bring a color dialog box when clicked. The upper right button swaps the current colors, the lower left - resets colors to default.

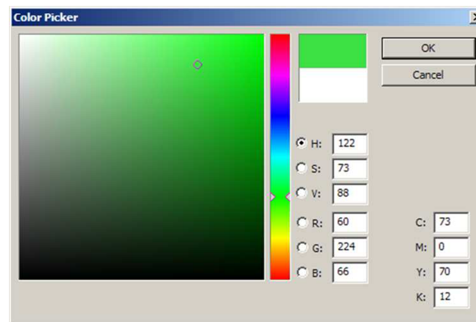
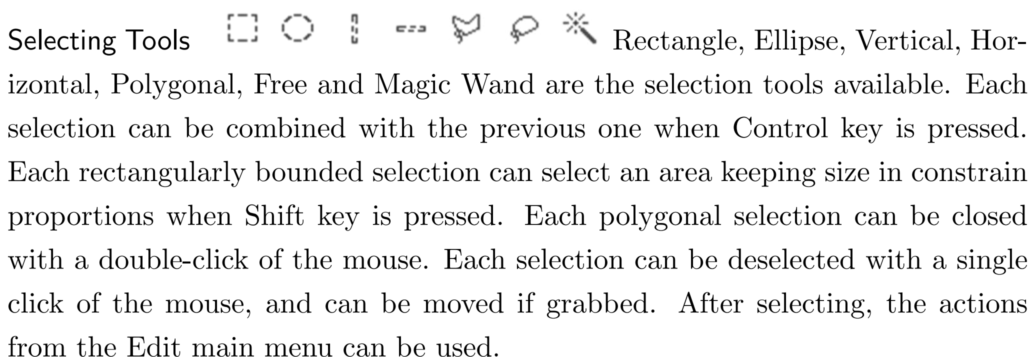


Figure 2.3: Rich color chooser dialog window in Hue color mode.

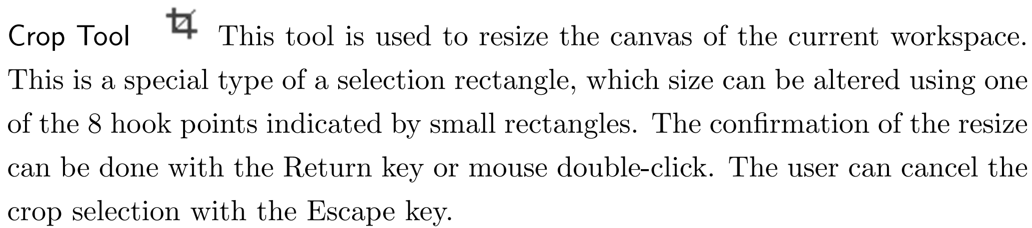
2.2.3 Tools

Tools are the main features of Fedit. Using them the user can perform various operations on the edited image and the outcome of such actions is strictly limited by the user's imagination. The Tools are divided into groups of similarity to improve the experience with the interface.

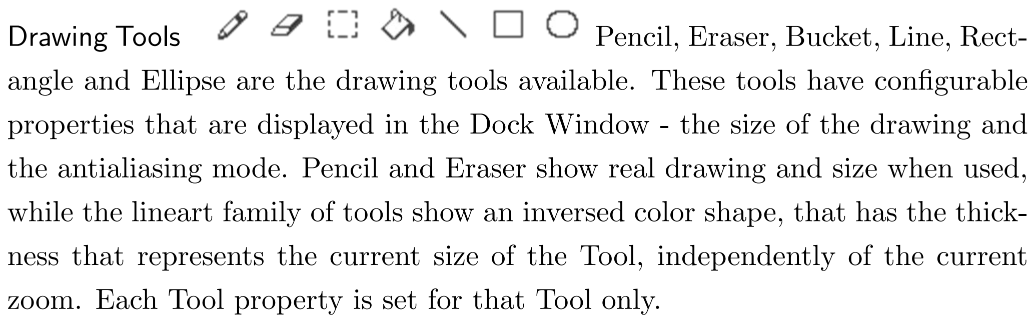
Move Tool Moves the selected layer along the workspace.



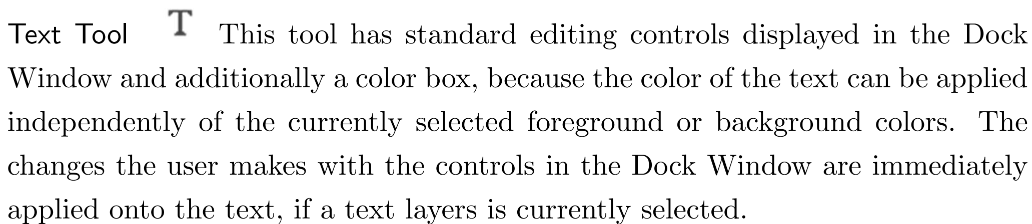
Selecting Tools Rectangle, Ellipse, Vertical, Horizontal, Polygonal, Free and Magic Wand are the selection tools available. Each selection can be combined with the previous one when Control key is pressed. Each rectangularly bounded selection can select an area keeping size in constrain proportions when Shift key is pressed. Each polygonal selection can be closed with a double-click of the mouse. Each selection can be deselected with a single click of the mouse, and can be moved if grabbed. After selecting, the actions from the Edit main menu can be used.



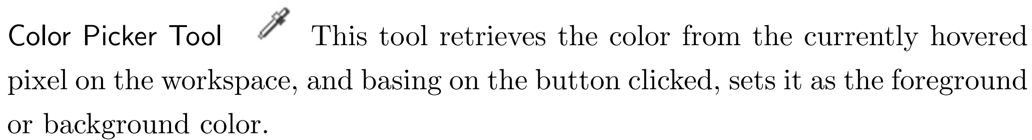
Crop Tool This tool is used to resize the canvas of the current workspace. This is a special type of a selection rectangle, which size can be altered using one of the 8 hook points indicated by small rectangles. The confirmation of the resize can be done with the Return key or mouse double-click. The user can cancel the crop selection with the Escape key.



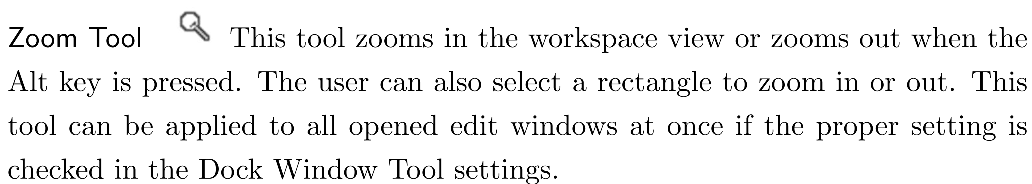
Drawing Tools Pencil, Eraser, Bucket, Line, Rectangle and Ellipse are the drawing tools available. These tools have configurable properties that are displayed in the Dock Window - the size of the drawing and the antialiasing mode. Pencil and Eraser show real drawing and size when used, while the lineart family of tools show an inversed color shape, that has the thickness that represents the current size of the Tool, independently of the current zoom. Each Tool property is set for that Tool only.



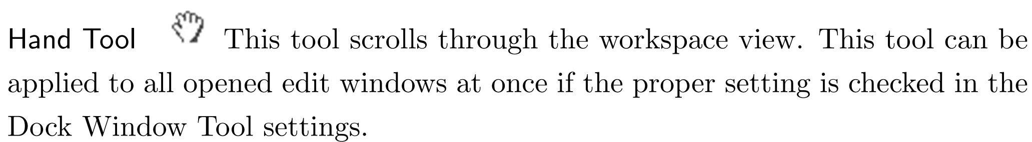
Text Tool This tool has standard editing controls displayed in the Dock Window and additionally a color box, because the color of the text can be applied independently of the currently selected foreground or background colors. The changes the user makes with the controls in the Dock Window are immediately applied onto the text, if a text layers is currently selected.



Color Picker Tool This tool retrieves the color from the currently hovered pixel on the workspace, and basing on the button clicked, sets it as the foreground or background color.



Zoom Tool This tool zooms in the workspace view or zooms out when the Alt key is pressed. The user can also select a rectangle to zoom in or out. This tool can be applied to all opened edit windows at once if the proper setting is checked in the Dock Window Tool settings.



Hand Tool This tool scrolls through the workspace view. This tool can be applied to all opened edit windows at once if the proper setting is checked in the Dock Window Tool settings.

2.2.4 Main Menu

Aside from the already mentioned File and Edit menu, Fedit Image Editor has other menus. Some of them are standard GUI menus, like Window menu that controls the Multiple Document Interface, others, like Layer or Select, are the

extension of the already existing shortcut keys or tool windows. Below is the description of the most important non-trivial menus.

Image > Image and Canvas Size This menu brings the dialog of resizing either image size or workspace canvas. In that dialog several options can be set including constrain proportions resize and the quality of image resampling.

Layer > Merge Down, Rasterize These 2 menus perform actions on layers. Merge Down applies the rendered projection of the currently selected layer onto the layer below it, unless it is the most bottom layer. It also rasterizes text layer on the fly when performing the merge operation on a text layer. On the other hand Rasterize can act only on text layers, and it converts the selected text layer into a normal raster layer for which the user can apply i.e. a filter.

Filter This menu contains one of the most powerful features of Fedit Image Editor, namely matrix filters. Each of these tools can drastically change the look of the edited image, and with the ability to give the user defined filter matrices, the possibilities of manipulation are almost infinite.

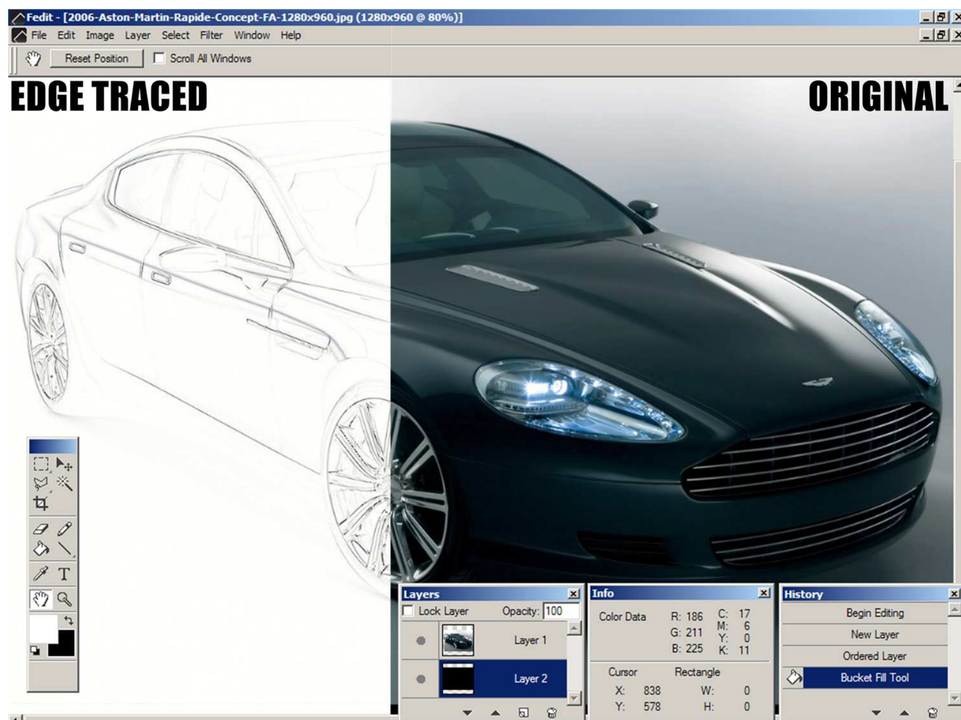


Figure 2.4: Example of Edge Tracing filter.

2.2.5 Matrix Filters

Matrix filters rely on 2-dimensional arrays of variable size. The centre of such matrix represents the filtered pixel on the source image, while the surrounding cells of that matrix define the amount of pixel data from the surrounding pixels that are to be added or subtracted (depending on the sign of the value in that cell) to the filtered pixel.

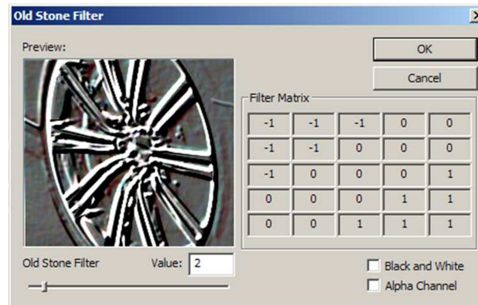


Figure 2.5: Matrix Filtering dialog window.

The user can choose from several predefined filtering matrices, that can be used in image processing such as Blurring, Defocusing, Sharpening, Edge Tracing, Embossing or Highlighting, even an Old Stone artistic filter. Moreover, if the the user finds these predefined filters not sufficient for the work, a custom filtering matrix of size 5x5 can be defined, with values ranging from -50 to 50. In the Matrix Filter dialog the user can preview the manipulation at any given moment, and apply quick changes in the values of either the predefined filters or the custom matrix the user is creating.

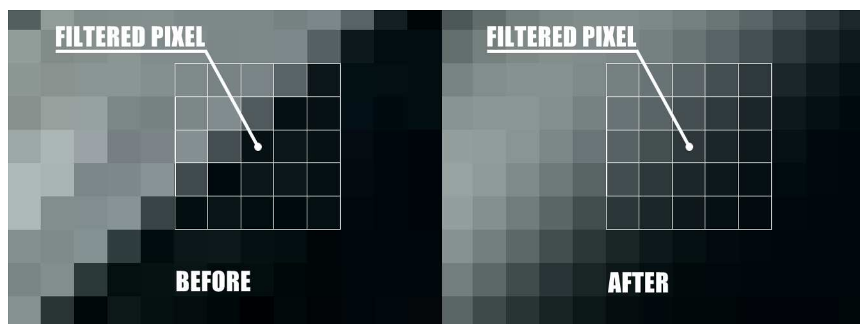


Figure 2.6: Closer look at the effect of Blurring filter.

Chapter 3

Analysis of Implementation Problems

Creating graphics editing software is a challenge in many fields of application design and implementation. With a rich featured program of this type, not only the actual effect of the applied tools is important, but what is even more crucial is the seemless workflow and satisfactory user experience.

The interface should be intuitive, not overwhelming the user with huge amounts of controls and information. On the other hand it should provide sufficient details of the current work and any hidden features should be easily accessible. On top of it the interface should be configurable, so that the user can arrange it to suit the needs.

The speed of work is an essential matter and the graphical tools should respond in a sensible amount of time to the actions made by the user. Where they have to consume more resources and delay the workflow for a longer moment, a clear indication of that process has to be passed to the user, either through mouse cursor or other visible effects.

Generally speaking, the main problem is to create reasonably fast working application, not an application that works properly, but slowly. Fedit Image Editor partially succeeds in solving this problem.

3.1 Speed of Editing Tools

Among the several tools Fedit Image Editor is equipped with, some require to work simultaneously with the user input (particularly mouse input), like the Drawing Tools, or the Selecting Tools. Others, which include Zoom Tool and Text Tool, do not rely on constant input, but perform the task just after the input has been finished - thus their speed is not so crucial.

Pencil and Erase Tool The best example of fast working tools are the Pencil and Eraser Tools. These tools have been fully optimized to achieve this performance. Both of these tools, when the user starts using them on an image (pushes the mouse button), do not act on the actual image, but on an invisible layer above the workspace, that firstly - is of size not greater than the current MDI child window's client area, and secondly - is repainted only in the area that covers the last 3 drawn points. The painted area is applied onto the actual image (or subtracted in case of the eraser) after the user finishes drawing - lifts up the mouse button.

Lineart Tool Such approach proved to give the best results and the Pencil and Eraser Tools are the fastest working tools that receive continuous input. Less optimized tools are the Lineart Drawing Tools. They also have an invisible layer as a negotiator between the drawing process and the actually edited image, however they do not draw directly to that layer. Instead, they pass a graphics path to the object responsible for the rendering of the workspace scene. This solution is not as fast as the previous one, however, because the path is a vector graphics, its coordinates modifications in response to the user input are fast enough to provide smooth workflow - no pixel manipulation of the shape that is currently being drawn occurs until the end of the user input.

Move Tool The slowest tool is the layer Move Tool. This tool moves a workspace layer from it's original position to another, effectively forcing the constant redraw of the current layer and the contents it covers below and above during the movement. Therefore a lot of redrawing occurs, which proved to be the most time consuming action, relatively to the effect of that work. Possible solutions of that problem include:

- Redrawing the appropriate scene area only after the mouse cursor has ceased to move. This solution should be the fastest from the application point of view, but the user would find it unacceptable.

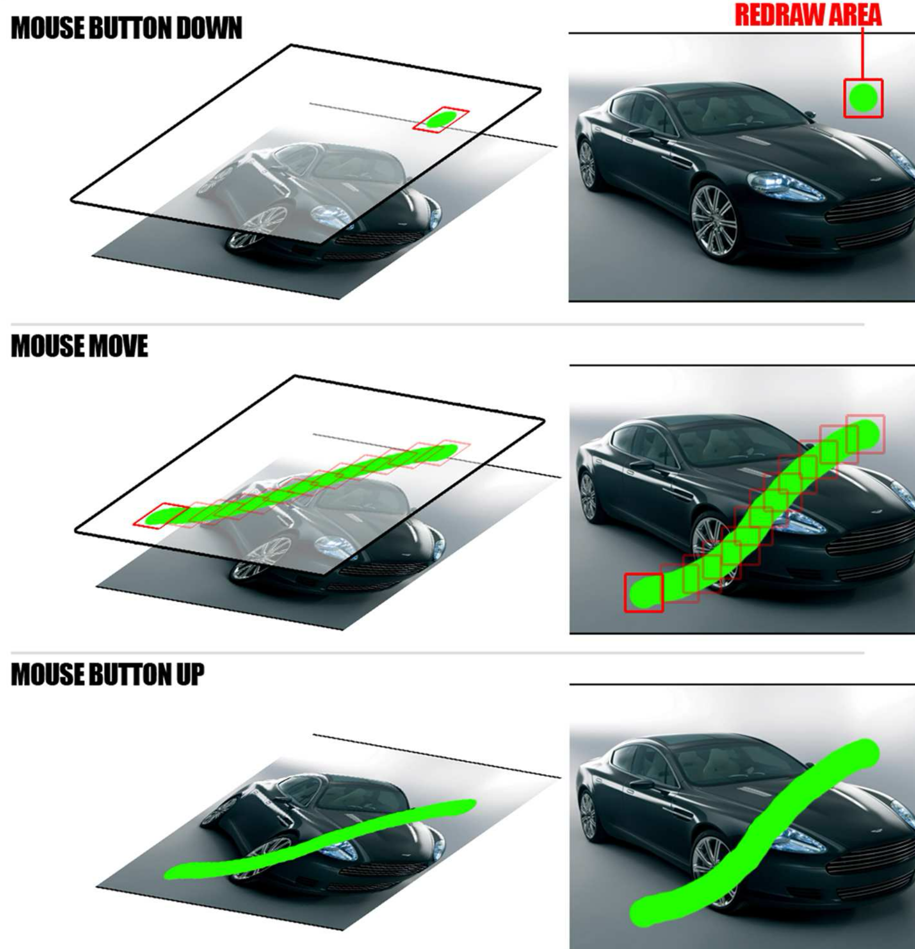


Figure 3.1: Pencil drawing algorithm.

- Redrawing the appropriate scene area sequentially, namely by splitting the area into several blocks and applying redraw to each of them in some order. This solution is not faster from the application point of view, but the user would be able to see the change of the area sooner.

- Converting the layer into a child window with appropriate clipping flags and using Windows API functions to move the position of the layer. Usage of the internal system interface functions may be the fastest solution.

Unlike previous tools, Move Tool's speed of work is strictly connected to the size of the current layer and the size of the current child window. Thus, when the user exceeds certain limit of pixel data that is in the process of editing of that tool, the performance may be poor despite any optimization implemented.

Other tools that are affected by this are the Magic Wand Selection and Filtering Tool. The following paragraphs describes the algorithms behind these tools.

Magic Wand The Magic Wand is a tool that selects an area basing on the similarity of colour. It essentially is a Flood Fill algorithm with shape outline recognition of the filled area. In Fedit Image Editor this tool is implemented in the following way:

1. Take the input pixel's position and colour
2. Flood fill an offscreen bitmap with a key colour using the bitmap data of the source bitmap
3. Create a graphics path
4. For every 1-pixel-width column of the offscreen bitmap, seek for key colour pixels. When found, begin tracking the height of that key colour strip until the null color or the bottom end of the bitmap is reached. Add the strip rectangle to the graphics path
5. Outline the graphics path

Because the amount of time needed to convert the shape into an outline is related to the size of that shape, the bigger the selection, the greater is the time of the algorithm computation. The solution to this problem is to further optimise the outlines recognition - however it is a task in the area of vectorisation, not raster graphics processing.

Filter Tool As mentioned earlier, Filter Tool uses matrices to change the value of pixels, thus resulting in filtering. The speed of this tool is acceptable, mainly due to a fast bitmap processing scheme of accessing color bits directly in the object's data buffer. All the computation is made using simple data types such as integer or double precision numbers, and simple mathematical operations.

However, filtering still has to be applied to every pixel on the source bitmap, therefore resulting in a strict relation between the time of processing and the bitmap size. Moreover, Fedit Image Editor implements filter matrix growth. For example, the initial matrix of Blur filter is a 3x3 array of 1s. With the

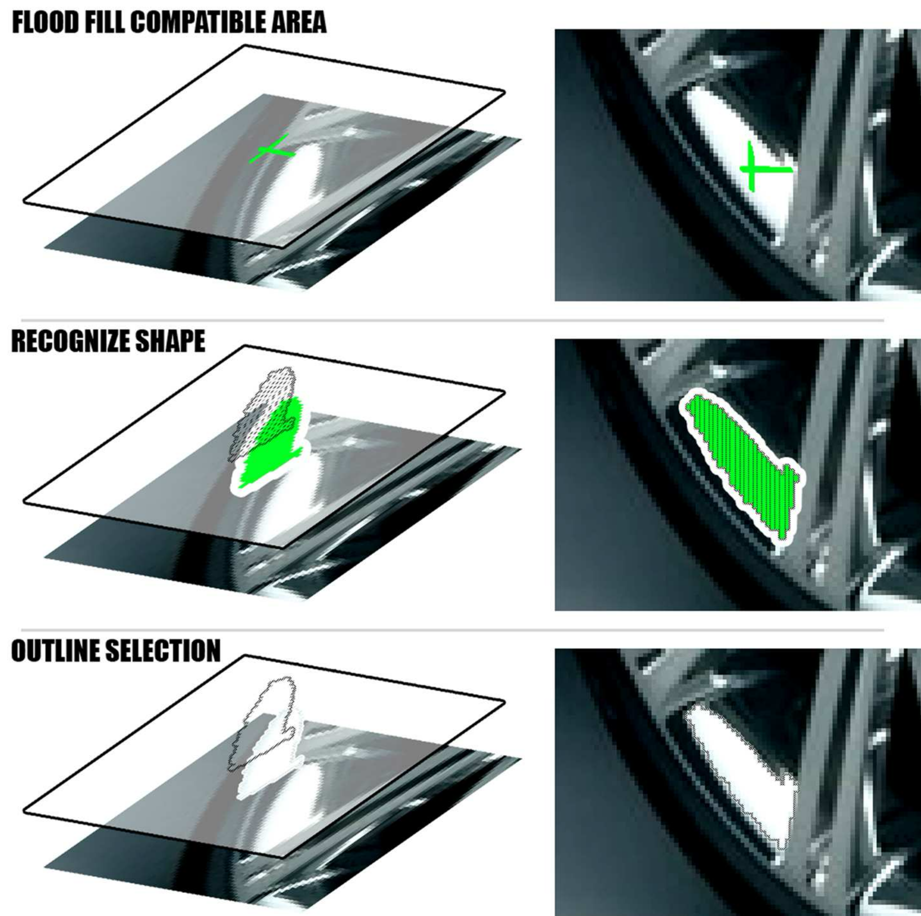


Figure 3.2: Magic Wand selection algorithm.

user increasing the value of the filtering, the matrix inflates to represent greater blurring. Thus, for every pixel that is filtered, the algorithm has to include more and more surrounding pixels, increasing the number of iterations, and resulting in a slight slowdown of the whole filtering process. There are several solutions to that problem:

- Preparing offscreen bitmaps with pre-filtering using values different than the user specified, but probable for the next choice. In the filtering dialog preview, the amount of overhead bitmap data would be acceptable, however upon implementing a feature of live preview directly onto the workspace layer, the results would consume too much memory
- If the user is increasing the value of the filter, instead of inflating the matrix, refilter the last bitmap with the current matrix. The drawbacks of this

method are that decreasing the filter values is not optimised at all, and that it is only applicable for filters that do not make radical color changes (i.e. for Blur filter, Sharpen filter, but not for Edge Tracing)

- Knowing that the user tends to operate only in a limited area of values, upon value change - remember the filtered bitmaps. Use them when the user is making the final decision of the filtering value and is switching between a small set of numbers

None of the above solutions is perfect, however implementing each of them to fit specific filters could result in smoother workflow. On the other hand, for such kind of problems the best result would also be to inform the user that for processing huge bitmap data, a sufficient amount of processing power and memory is required.

3.2 Memory Consumption

Although currently the price of random access memories have dropped dramatically, compared to the previous years, they are still not as cheap per megabyte as the hard disk memories. Moreover, the new applications and operating systems require more memory than before to provide the user with a rich graphical experience. Also, from the psychological point of view, the memory consumption is an important factor for most of the users and, along with the working speed and functionality, define their opinions on the software. Therefore it is crucial to minimise the resource needs of the application and ensure every data is freed from the memory before their handles are erased.

Fedit Image Editor takes great care to delete every object it creates and prevent any memory leaks, especially of Gdiplus objects. It has been tested to maintain the low resource consumption level and return to it upon closing of the image editing windows. Moreover, it's implemented in such a way that no unnecessary data is stored by the application. The best example of such approach is the History of the user actions.

History This component stores the changes the user has made to the workspace during the editing of an image. Each History Element is associated with a specific editing tool and because of that associations, each works differently, depending on the tool. However, a common thing among them is that they store the least amount of information they need to revert the user changes. For example the

Move Tool Element only holds the difference of the movement, as an integer number. The Crop Tool Element also holds integers representing the change of the workspace canvas. Selection Element remembers the points of the current and previous path, and the Layer Order Element stores the handles to the swapped layers.

The more memory consuming tools are obviously Drawing Tools. However, even here the History Element stores only the changed area of the painted bitmap, not the whole image. It is the same with any action of editing of limited areas - such as filtering of a bitmap selection or pasting bitmap data from the clipboard.

The most demanding is the process of resizing the image. Because this process affects every pixel of the source bitmap, and there is no guarantee that any of the pixels will remain unchanged, Resize Element stores the whole bitmap data. The increase in the resource usage can be clearly noticed when the user edits an image and uses many layers with rich bitmap info. Unfortunately there is no perfect solution for that problem. One of the ideas is to compress the saved bitmap data either to a lossless format like PNG, or to an archive file like ZIP or RAR. However, a decision has to be made whether the user will more likely benefit from the memory compression or the smoothness of the workflow, as the compression process would affect the general working speed of the application.

It has to be mentioned that the user is left with an option of deleting the resource consuming element, thus freeing the occupied memory. With that in mind, the memory management in Fedit Image Editor can be considered quite efficient and flexible.

Chapter 4

Framework and Implementation Details

This chapter focuses on implementation aspects of Fedit Image Editor. It describes the used environments and libraries, as well as presents the complete framework that was constructed to create Fedit.

4.1 Application Environment

Fedit is a Win32 application, written in C++ using Object Oriented Programming. It utilises classes, template classes, multilevel inheritance. Almost every component that is visible by the user is either a class or is enclosed by some class.

The development environment is Microsoft® Visual Studio® 2005. The application uses Windows API to create the user interface, GDI+ graphics library to manipulate and store the graphical data objects used for image editing, and IStream and IStorage objects of COM Structured Storage interface for writing and reading the Fedit Document (.FED).

The application construction is divided into "Core" objects, where the Core is the main application object, and "'name' Core" indicate specific group:

. . . Source Files: code that represents the Core of the application
. . . . . . Child Core: code that represents MDI child windows
. . . . . . En Core: code that represents accessing compressed archives
. . . . . . Hard Core: code that represents the very basic data structures
. . . . . . Mini Core: code that represents file browsing and additional display functionality
. . . . . . Soft Core: code that represents the whole interface
. . . . DialogHelpers: code that represents dialog helper objects
. . . . . . Toolw Core: code that represents Tool Windows functionality
. . . . Tools: code that represents all Control Center Tools
. . . . ToolwBox: code that represents particular Tool Window boxes

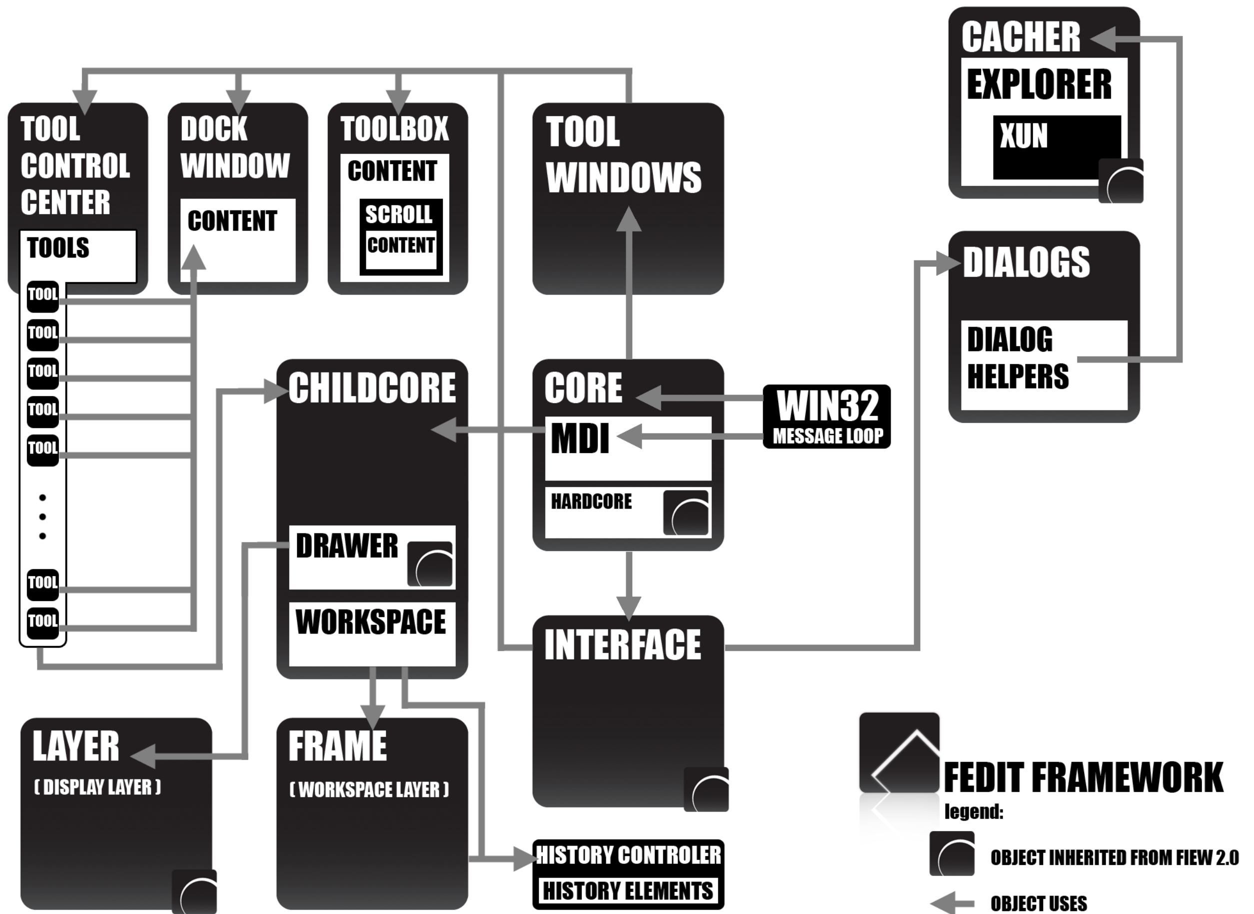


Figure 4.1: Fedit Image Editor Framework diagram.

4.2 Framework Description

4.2.1 Objects Overview

Core The core of the application that manages all other components, connects them and distributes messages to and among them. It also contains global usage variables and methods.

HardCore Partially inherited from Fiew 2.0. Custom Fedit WCHAR object for managing strings and custom typed list with various methods enhancing it's usage. These objects were implemented because of the need to fully control the processing of such basic types. The FwCHAR class is used throughout the application to process text strings. It's enhanced with easy string concatenation, string comparison, extracting of specific data, like filenames, folders, extensions, as well as is equipped with various status methods that calculate length, number of lines (in case of multiline strings) and enable conversion to other formats, like character arrays or wide characters.

The typed list is a double linked list with Nodes that encapsulate arbitrary T objects. The list has 3 cursors, main cursor 'that', and 2 helper cursors 'left' and 'right'. This class is enhanced as well, with specific cursor operations of rewinding, moving forward, jumping to selected object, counting elements in respect of the current object, swapping the element etc. Most of these operations can be performed on all 3 cursors, however the main cursor - 'that' - is equipped with the biggest number of helper methods.

Interface Partially inherited from Fiew 2.0. Interface object manages interactions between the main menu and the application Core, and controls the state of that menu, also it controls the common dialog boxes for file management.

Tool Windows This object represents the controller of Tool Windows and is the connection between the Tool Windows and the Core of the application or MDI children. This object is not a strict connection, that is - some messages can be passed from Tool Windows to Core or children without the controller, however not the other way around.

Tool Box This object represents a Tool Window that contains content like Layer, History, Info. It is a container and is prepared for the future option of moving inside contents between other Tool Box windows.

Tool Box Scroll This object represents a scrollable content pane. It contains another content pane inside itself that can be larger than the Tool Box Scroll and then the Tool Box Scroll displays scroll bars and enables scrolling through the contents of the insides. This content pane is autonomous and can be used anywhere.

Tool Box Content This object represents a container that is placed inside Tool Box. This class is an interface class, it is inherited by the contextual content like Layers, History, Info.

Dock Window This object represents the docking window. It contains a content pane that is filled by Tool objects independently of the Dock Window itself. This object contains usual docking properties.

Tool Control Center This object represents the Tool Control Center that enables the user to choose from a variety of tools, also contains the tools that are activated through main menu items.

Tools One of the major objects in Fedit. All Tools inherit from the base class Tool. They process the user input received by ChildCore to perform their tasks.

Child Core The ChildCore object is the MDI Child window, it contains all necessary elements to control the presentation of current work.

Drawer Partially inherited from Fiew 2.0. One of the major object in Fedit, along with Layer it coordinates the rendering of the scene of a MDI child window basing on the Workspace info

Layer Partially inherited from Fiew 2.0. One of the major Fedit components, responsible for rendering the scene of MDI child window. It performs scrolling, zooming, and several other tasks connected with image display.

Gridlay Inherits from Layer. This object is responsible for drawing the transparency grid in the scene presented on the MDI child window

Thumblay Partially inherited from Fiew 2.0. Object responsible for displaying thumbnail view, ie. for opening a folder dialog. It relies on a Cacher object, and is refreshed to represent current Cacher state.

Cacher Inherited from Fiew 2.0. Cacher browses through Explorer file path data and loads raw image data (NOT as Gdiplus objects) into memory. It uses multithreading and can manage strictly defined cache size i.e. only 10 images before and after the current image. Cell object that is used for Cacher data list is responsible for retrieving Gdiplus images and thumbnails from raw image data.

Explorer Fedit file browser that enables browsing of folders and archive files. Browsing is based on file headers and file extensions. Browsing is different depending of opened file:

1. Normal file - browses through all files in that file folder, including archives
2. Folder - browses through all files in that folder, including archives
3. Archive - browses through all files in that archive, excluding files in the folder that archive exists, and excluding any archive files inside that archive

XUn This object is a wrapper class for extracting contents from compressed archives. This class is inherited by XUnzip and XUnrar, that extract ZIP and RAR archives respectively. ZIP archive extraction uses built-in code basing on the Mark Adlers original UnZIP code that was modified by Lucian Wischik and Hans Dietrich. RAR archive extraction uses RARLabs unrar.dll which is extracted from the Fedit executable if necessary.

Workspace Workspace object represents the workflow contents of MDI child window. It contains the layers, controls them and applies actions on them, also is the content of the Fedit Document object

Frame Frame is the object that represents the layer in the editing process. It contains various information about the state of the layer and the Bitmap or Path data, if it is a derived object of FrameText type. FrameText object represents text layers. It contains info essential for displaying the text, the data string, and the Graphics Path representing the text.

Hisotry Controler This object controls the undo and redo processing for a particular Workspace. It stores, adds and removes History Elements.

History Elements Each element is responsible for undo and redo actions on a specified process of editing. All elements inherit from History Element and follow

the same interface, using set() methods for supplying data and undoAct() and redoAct() to perform going backword and forward with actions.

Dialogs One of the major objects in Fedit, consisting of mostly static methods that create or modify interface controls. Its main task is to process dialog boxes. This object can also convert or create controls that are Fedit Common Controls.

Dialog Helpers This objects coordinate dialogs box for which they were designed, namely:

1. DialogHelper_ Clr: an object that controls the custom color chooser dialog

2. DialogHelper_ Fil: an object that controls the matrix filters dialog

4.2.2 Internal Common Controls

Fedit Image Editor, to enhance the general user interface, modifies some standard Windows API controls. The following paragraphs describe which controls are modified and how.

Font Substitution All interface controls that contain text are created using a static method of Core object that takes the same parameters as CreateWindowEx, but also substitutes the default font with the new Windows dialogs font - MS Shell Dlg.

Edit Box Integer Limit All interface edit boxes that are made to accept only numbers, are converted on creation to a subclassed edit box that has a limit of maximal and minimal integer input. The edit box shows a warning message box on the try of exceeding the limit. Also, the edit box enables performing changes of the value of the edit box with Up and Down arrow keys.

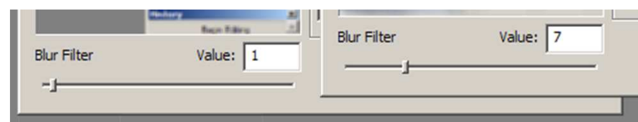


Figure 4.2: Slider control associated with a limited edit box.

Edit Box with Slider Control The above limited edit box can be associated with a slider control. After the linking of these 2 controls, the change of the value in

one of them, affects the other, in real time. The association can be made after the creation of the slider control or during its creation.

Editable Static Text A static control displaying text can be converted to an in-place editable static text control. Upon the users double-click on the static control, an edit box is created on top of the control, with the contents of that static control selected and ready for editing. The subclassed control processes confirmation and cancelation messages in the form of Return and Escape keys.

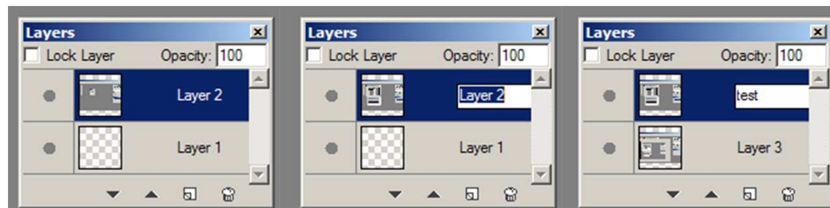


Figure 4.3: In-place editable static text control.

Custom Ownerdraw Button Each existing button can be converted into an ownerdraw button, that is displayed in the way that emulates the Windows® XP themed buttons. Moreover, these buttons can be autocheck-buttons or simple buttons, depending on the flag during the creation.

Figure 4.4: Owner draw autocheck buttons and a color box.

Color Box A static control can be converted into a color box that holds information about the recently chosen color. It displays the color by painting itself with it. Upon the user click on that control, it brings the Fedit color dialog, where the user can change the current color of that static control.

Button Popup Switcher A button from the Tools Control Center can be assigned with a list of buttons that it can morph into, changing its ID and thus the triggered action, but remaining in the same place. Such button toggles the popup switcher on the user's right button click or a longer left button hold, and the

popup accepts a direct movement from the left button hold, without the need of releasing the mouse button.

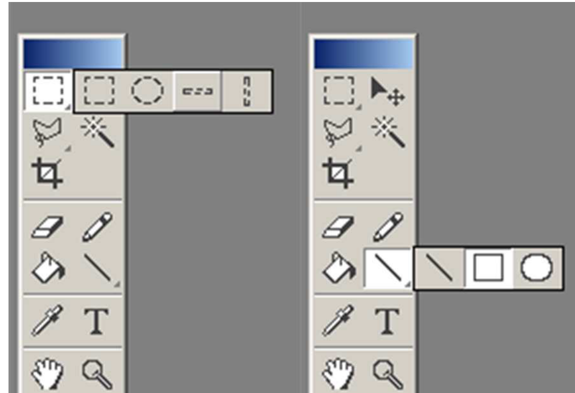


Figure 4.5: Popup switchers triggered by a right mouse button click and a left mouse button hold.

4.2.3 Fedit Document (.FED)

Because image editing can be very time consuming, it is essential for an editing application to be able to save the processed file at any moment, keeping the current state of the image, as well as the data created by the editor, like layers, effects, texts or layer properties flags. Therefore, a Fedit defined file format was introduced for saving and restoring workspace information. The following paragraphs describes that format.

The Fedit Document save and load functionalities are implemented using Compound Files of the COM Structured Storage. Windows® Platform SDK on structured storage delivers the following information:

“ Structured Storage provides file and data persistence in COM by handling a single file as a structured collection of objects known as storages and streams.

The purpose of Structured Storage is to reduce the performance penalties and overhead associated with storing separate objects in a single file. Structured Storage provides a solution by defining how to handle a single file entity as a structured collection of two types of objectsstorages and streamsthrough a standard implementation called Compound Files. This enables the user to interact with, and manage, a compound file as if it were a single file rather than a nested hierarchy of separate objects.”

Because of the decision to use Compound Files, the saved workspace had to be divided into sections that would represent single files of a Compound File. Thus the following sequence of structures was introduced:

- first item is the structure that reflects the information about the workspace
- followed by any number of pairs of structures:
  - structure that reflects the information about the layer
  - structure that reflects the image data of the layer

Workspace Information Structure This structure is an array of floating point numbers of length 10. Each of the array cells holds a specific information about the workspace object, namely:

- WSPC_ - unused, always 0
- WSPC_ W - width of the workspace canvas
- WSPC_ H - height of the workspace canvas
- WSPC_ RES - resolution of the workspace
- WSPC_ PXW - pixel width of the workspace canvas
- WSPC_ PXH - pixel height of the workspace canvas
- WSPC_ SUNIT - size unit of the canvas
- WSPC_ RUNIT - resolution unit
- WSPC_ DMODE - color depth mode
- WSPC_ DUNIT - color depth unit

Layer Information Structure This structure is an array of integers of length 9. Each of the array cells holds a specific information about the layer object, namely:

- FRMD_ TYPE - frame type (text or normal)
- FRMD_ X - frame horizontal coordinate
- FRMD_ Y - frame vertical coordinate
- FRMD_ OPCT - frame opacity value

- FRMD_ LOCK - frame lock flag
- FRMD_ VIS - frame visibility flag
- FRMD_ EMPT - frame empty flag
- FRMD_ NAMELEN - frame name length
- FRMD_ NAMEPTR - unused, always 0

Layer Image Data The layer bitmap image data is compressed using a lossless PNG format, using GDI+ built in saving functions.

Summarising, the Fedit Document .FED contains workspace information as a file header, and an unknown number of layers that are pairs of layer header information and layer image data. Because of the implementation uses Structural Storage, the file does not need to contain the info about the size or number of stored layers, as the IStorage object is able of enumerating all Compound File contents. This may rise a question of handling the corrupted files, however the Workspace loading method is designed in such a way that it tries to restore as much data from the file as possible, but returns safely upon failing with a proper error message to the user.

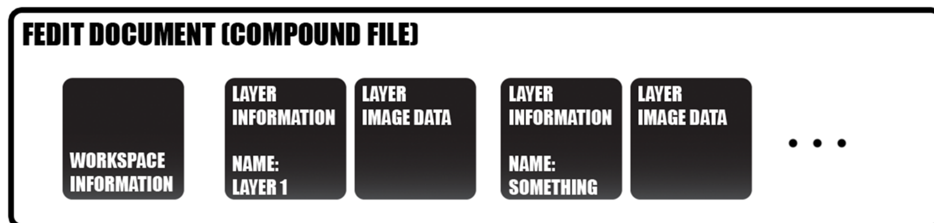


Figure 4.6: Fedit Document structure .

Chapter 5

Conclusion

The task to create an advanced image editor proved to be difficult. Especially with the Fapplication principles that were assumed for this project, the implementation was not as easy as it would be when using application designers such as .NET Framework, Java or Borland Builder. However, despite the difficulties, the principles were satisfied, and the resulting software is an editor with most of the planned capabilities. Moreover, it receives positive comments from the test group of users. On top of it, the Fapplication principles proved to be very appropriate and useful, when there occurred a situation of a corporate computer administrated periodically by the company headquarters, which had a limited user privileges and blocked installation of any applications. In such environment, Fedit Image Editor was the only option for that user to perform advanced image editing.

It is a certain fact that there is a lot of features that can be added to this application. On the other hand, there are already a good number of interesting features. Looking back at the imaging industry history, and the already 13th version of Adobe® Photoshop® that is hitting the market today, Fedit Image Editor may be a very good beginning for a popular, freeware and user friendly raster graphics tool.

List of Figures

2.1 Browsing folder contents with thumbnail viewer. 5
2.2 Dock Window in different docking states. 6
2.3 Rich color chooser dialog window in Hue color mode. 7
2.4 Example of Edge Tracing filter. 9
2.5 Matrix Filtering dialog window. 10
2.6 Closer look at the effect of Blurring filter. 10

3.1 Pencil drawing algorithm. 13
3.2 Magic Wand selection algorithm. 15

4.1 Fedit Image Editor Framework diagram. 20
4.2 Slider control associated with a limited edit box. 24
4.3 In-place editable static text control. 25
4.4 Owner draw autocheck buttons and a color box. 25
4.5 Popup switchers triggered by a right mouse button click and a left mouse button hold. 26
4.6 Fedit Document structure 28