Script-Focused Image Editor: Progrimage

A Senior Honors Thesis

Submitted in Partial Fulfillment of the Requirements
for Graduation in the Honors College

By
Jacob Brandt
Computer Science Major

SUNY Brockport, State University of New York
March 3, 2023

Thesis Director: Dr. Eric Owusu, Assistant Professor, Computing Sciences

Abstract

Image editors are powerful tools but can be more difficult to work with than code for some specific tasks. They are great for art, but not always as good for precise image manipulation. This is where code-based editors are more effective. There are some image editors that support scripting, however, some image editors are complicated to use, with no available documentation. Other image editors have scripting hidden within submenus making it difficult to find. Furthermore, some image editors implement scripting in a way where they only a basic "run" button and lack the use of tools and interactive filters. To achieve the goal of creating a scripting tool for image editing that will be user-friendly and accessible, this exploratory study examines: "How well can scripting be implemented in an image editing application when scripting is the main focus?"

The objective of this study is to create an easy-to-use image editor that allows users to create scriptable tools and image filters without having to search for the ability to do so. The proposed application, Progrimage, uses a scripting language called Lua which was designed to be beginner-friendly. Progrimage is designed to have a dedicated button to create a tool and to create a filter that can be used for drawing and manipulating objects, processing image pixels, creating layers, and more. With this image editor application, scripting has high priority. Applying a user centered design approach, Progrimage will allow users to create their own tools and filters and is intended to increase user satisfaction, and intensify learnability and user adoption.

Contents

1. Introduction

Image editors, like Adobe Photoshop, are tools used to manipulate images. Some of them have scripting support allowing users to write code to achieve tasks in the application. The benefits of scripting support is that it allows users to achieve tedious or high precision tasks. It could also be useful for users who are better at coding than editing images by hand. For example, it could be used to generate images from code. However, this means that users need to know how to code in the given scripting language. The problem addressed by this exploratory work examines the ease-of-use, beginner-friendliness, and applications of end-user scripting in image editors. Specifically, it will address the question of "How well can scripting be implemented in an image editing application when scripting is the main focus?" There are many existing image editors that support scripting, but most have drawbacks to their implementation.

2. Background Study

Many image editors have less than ideal implementations for scripting if they support it. For example, Krita, a free and open-source image editor, supports scripting in python. However, Krita's scripting can be very complex, can be very confusing to use, and is limited to "run-once" applications. To clarify what "run-once" means in this study, it refers to an action that only runs when the user performs some action such as clicking a button, and then does not run again without the user repeating this action. GIMP, another free and open-source image editor, also supports scripting. In the case of GIMP, the program has a custom-made scripting language called "Script-Fu" [1]. Using a proprietary language puts it at a disadvantage due to a lack of or significantly less existing online help outside of official documentation. Additionally, Script-Fu is unlike standard languages and can be very difficult to use as it is more similar to languages

like Lisp. Adobe Photoshop, an industry-standard for image editing, also supports scripting. However, it is complex to use with the official documentation being an 88-page pdf [2]. This discourages new users who might have an interest in exploring it. There are many other image editors that support scripting, but many of them have issues similar to these examples such as complexity, non-beginner-friendliness, “run-once” applications, hard to find documentation or documentation that does not readily present users with a list of code features and objects, and other functionality.

3. Methodologies

Online support and research were helpful in gathering knowledge about algorithms and concepts. The article by Erika Jansson titled “Brush Painting Algorithms” [3] was very helpful in learning how to write the brush drawing algorithm, as well as how to visualize it. C# was chosen for the program’s backend due to familiarity with the language. Lua was chosen for the scripting language due to its simplicity. The application requires at least Windows 10 to run. Windows is the only operating system supported due to using Windows Forms libraries for file dialogues and setting and getting clipboard data. The idea of a script-focused image editor originated due to the ease of which some image manipulation or image generation tasks are to achieve with code as opposed to manual editing. The project uses the applied research methodology of developing an innovative piece of software.

The programs used for creating the proposed image editor Progrimage were Visual Studio Community 2022, and Visual Studio Code. The project was primarily built with Visual Studio Community 2022 while some parts of the Lua scripting as well as all of the documentation was created with Visual Studio Code. The source code for Progrimage as well as

example scripts and documentation are available on GitHub. All test scripts were written with Visual Studio Code using the Lua extension to provide syntax highlighting and autocomplete, however, any text editor will work. The application also uses ImGui.NET and DesktopGL for the user interface, NLua, which itself uses KeraLua, to run Lua in C#, and ImageSharp for image handling and manipulation. Additional proprietary libraries were developed for use in the application such as a vector library, a calculator library, ImageSharp extensions, and a library to directly modify Windows Forms Bitmap objects. The calculator library will attempt to calculate a numerical value from a math expression given by a string of text. This library was adapted from a previous project that had been developed for use as a standalone GUI application. The ImageSharp extension library provides faster drawing methods as well as other features missing in ImageSharp such as positioned images, retrieving subimages, or pieces of existing images, and more. The last library directly modifies Windows Forms Bitmap images for better performance and is used when copying images from the editor to the user's clipboard.

Additionally, Krita was used to design the logo for Progrimage which is also used as a default texture when none has been assigned yet. This can be seen on the program's executable file, as well as being used for the default icon when creating a new scripted Lua tool. A screenshot of Krita showing how the logo was made is shown below in Figure 1.

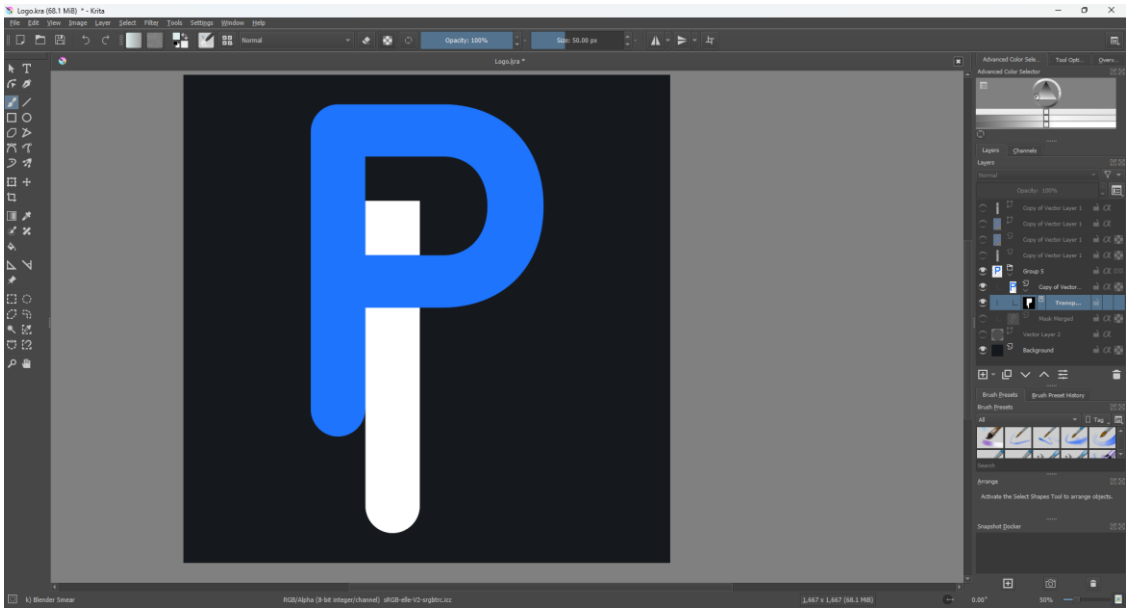

Figure 1: Krita Logo

This is the file containing Progrimage's logo along with the work put into it. This image is also used as the default texture when no texture is currently assigned.

4. Generated Content

Many icons were created with code. These include the icons for the marque selection, rectangle, oval, line, quadratic and cubic Bézier curves, text, crop, and create script tools, as well as the create, hide, unhide, and delete layer button icons which are also used for the hide, unhide, and delete buttons for composites. Currently, four tools use placeholder icons. These include the brush, eraser, fill, and move tools. A sample of code which generates the icon for the cubic Bézier curve tool is shown below in Figure 2.

```
public static void MakeCubicCurveIcon()
{
    using var img = new Image<La16>(size.x, size.y);
    img.Mutate(i =>
    {
        float thickness = size.min * 0.1f;
        i.Clear(new Color(new Argb32(0, 0, 0, 0)));
```

```
        double2 padding = thickness;
        double left = padding.x;
        double right = size.x - padding.x - 1;
        double2 a = padding;
        double2 b = new(right, (size.y - 1) / 3.0);
        double2 c = new(left, (size.y - 1) * 2 / 3.0);
        double2 d = new(right, size.y - 1 - padding.y);
        double dashLength = a.Distance(b) / 10;
        double gapLength = dashLength;

        i.Draw(new Color(new Rgb24(169, 169, 169)), thickness, new
PathBuilder().AddCubicBezier(a.ToPointF(), b.ToPointF(), c.ToPointF(),
d.ToPointF()).Build());

        float lineThickness = thickness * 0.5f;
        DrawDashedLine(i, Color.White, a, b, dashLength, gapLength,
lineThickness);
        DrawDashedLine(i, Color.White, b, c, dashLength, gapLength,
lineThickness);
        DrawDashedLine(i, Color.White, c, d, dashLength, gapLength,
lineThickness);

        float radius = thickness * 0.75f;
        i.Fill(Color.White, new EllipsePolygon(a.ToPointF(), radius));
        i.Fill(Color.White, new EllipsePolygon(b.ToPointF(), radius));
        i.Fill(Color.White, new EllipsePolygon(c.ToPointF(), radius));
        i.Fill(Color.White, new EllipsePolygon(d.ToPointF(), radius));
    });
    img.SaveAsPng(OutputDir + "cubic_curve.png", new PngEncoder()
    {
        ColorType = PngColorType.GrayscaleWithAlpha,
        CompressionLevel = PngCompressionLevel.BestCompression
    });
}
```

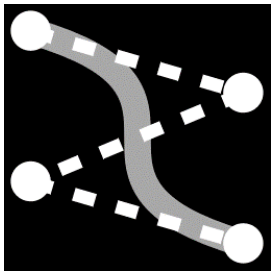

Figure 2: Cubic Bézier curve tool icon

The image shown in this figure is the icon used for the cubic Bézier curve tool.

Brushes in Progrimage use two files: an image file for the texture and a normalization file. The image file is used for what image to draw to the canvas. The normalization file comes

from compiling the image. First, it is important to understand how the brush draws. To draw, the program stamps the brush texture many times in a line using the brush step size for the interval. It also will not stamp distances smaller than the brush step size when trying to continue the stroke. In order to achieve high quality brush strokes, a brush step size of one pixel is used. Before the user begins a stroke, the brush texture must be rescaled to fit the selected brush size, meaning the image dimensions should fit within the bounds of a box whose width and height are the same as the brush size. Next, the data from the normalization file must be resized to match these new dimensions exactly. This is done using a proprietary image rescaling algorithm which upscales bilinearly but downscales using a modified version of the box sampling algorithm where edges with fractional coverage are not treated as solid pixels. This means that the edges are not treated as full pixels if the target pixel does not fully cover them. In addition, the algorithm can both downscale one axis while upscaling the other if needed. When stamping the brush texture, its pixels are sampled and each one is multiplied by its corresponding pixel in the normalization data. This is then cumulatively added to a buffer which represents the strength of the brush color at each pixel. When the stroke should be applied to the image, the values in the buffer are used to multiply the alpha of the current brush color which is then alpha blended with the image's pixels.

5. Processes and Algorithms

5.1 Undo and Redo

The undo system is comprised of 3 classes and an interface. There is an undo manager class, an undo action class, and an undo image patch class. The undo manager is what various parts of the program interact with. It handles undoing, redoing, and adding items to the undo

history. These items must inherit from the IUndoAction interface. This interface ensures that all classes inheriting from it have an undo method, redo method, and the rough size they take up in memory. The undo action and undo image class both inherit from this interface. While the undo image patch can accurately provide its size in memory, the undo action class cannot. This is because the undo image patch knows all of its data because it is comprised of a position and size, as well as two positioned images, while the undo action class only contains references to two delegates. Delegates in this case are like functions that can be stored in a variable and called from that variable.

When the undo manager is instructed to undo, it takes the next item in its history list and tells it to undo. In the undo image patch class, this is done by first saving the affected region of the layer as a positioned image and then replacing it with an already provided replacement image, as well as potentially changing the size and position of the layer. This replacement image is given when the undo image patch is created, typically done just before applying a change to the layer. In the undo action class, being told to undo simply calls its undo delegate. When the undo manager is instructed to redo, it takes the previous item in its history and tells it to redo. In the undo image patch class, this results in the positioned image resulting from the previous undo being used to replace the same region in the layer, then that image is disposed of to free up memory. In the undo action class, being instructed to redo simply calls the redo delegate. If the undo manager is told to undo several times and then a new item is pushed to its history, it will first clear all history items that came after its current position because they are effectively in the future and the past has been changed. The undo manager also has a memory size limit it uses to remove history items undo the total size is back undo the limit, starting with the oldest items.

5.2 Brush Compilation

In order to compile a brush texture and create a normalization file, four things are needed: the brush texture, scale, brush step size, and a quality value. First, the brush texture can be any image. Second, the scale multiplies the resolution of the brush texture to give the resolution of the normalization file. This should be above zero and less than or equal to one. Using a value of one will result in the normalization file being the same resolution as the brush texture but the compilation time may be longer. The scaling mode used is the same one used for resizing the normalization data described above. Third, the brush step size should be the same value used when drawing. Progrimage uses a brush step size of one pixel so a value of one should be used in compiling. Finally, the quality value controls the number of directions the stroke should be simulated drawing in which affects the quality of the resulting normalization file. When compiling, the program draws the brush in straight lines without normalizing it and keeps track of the minimum values of each pixel as long as they are above zero. After this, the normalization file is constructed out of the resulting measurements where each pixel in the file is a single precision floating point number resulting from the strength of the brush texture's pixel divided by the minimum value of its corresponding drawn pixel. When drawing in Progrimage, these multiply the strength of the pixels accumulated in the buffer resulting in an opaque pixel being made when a stroke is drawn. The entire compilation process is multithreaded to improve efficiency.

5.3 Drawing Brush Strokes

There exists a special class for handling brush stroke related tasks. This is used by the eraser and brush tool. It handles beginning, continuing, and rendering strokes. Before drawing, settings for it must be configured. This is done by keeping a state of brush settings updated with

choices the user makes. There is a state object for the brush and one for the eraser. When a setting is changed or the tool is swapped, the tool's state object is sent to the stroke class which prepares everything necessary to begin a stroke. This includes resizing the brush texture and normalization data explained in section 5.2. The stroke class stores many components as arrays of single precision floating point numbers for fast manipulation where each pixel is a single float. This includes the mask, scaled and unscaled brush textures, and scaled and unscaled normalization data. The unscaled brush texture and normalization data must be stored for quick resizing if needed. The mask is what holds the strength of the drawn stroke and is what is drawn to the layers. When it is told to begin a stroke, all data is cleared and reinitialized so previous strokes do not interfere with the new stroke. When told to continue a stroke, it adds the provided coordinate to a list and begins stamping the brush texture in a lines iterating through this list using the brush step size discussed in section 5.2 as the interval. When the brush texture is sampled in order to stamp it, the value in the scaled brush texture array is multiplied by the corresponding value in the scaled normalization data array in order to add up to full opacity when a full stroke is drawn without drawing too much or too little. This is then accumulated in the mask array. When the stroke should be drawn to an image, it multiplies the values in the mask array by the brush color in its stored state object and alpha blends them to the image. In the case of erasing, the image's pixel's alpha is multiplied by the complement of this product instead.

6. Results and Analysis

6.1 Results

Firstly, the issue of “run-once” applications of scripting is addressed in Progrimage by introducing the idea of “composites”. Composites are like image filters that are non-destructive and procedural. They are non-destructive in that they do not edit the source image, but rather sample it and replace it when rendered. Additionally, when using multiple composites, each one uses the output of the previous composite. These composites will also rerun when the layer they are attached to is modified. If the user wishes to merge them into the layer in their current state, they can click a button in the layer list. Where these come into the scripting aspect is that users can create their own composites with the click of a button and then writing code for it. These scripts can do things like change the size and position of the incoming image, draw shapes with built-in functions, edit pixels directly, and more.

Secondly, the issue of language and usage complexity was solved by using a scripting language called Lua. Lua was designed to be beginner friendly, and as such is the perfect choice for this use case. The code users write is short with little excess complexity. For example, below is a script for Krita from Krita’s lessons [4] which creates a one pixel wide by one pixel tall blue image:

```
from krita import *

# set up new document and set background color
newDocument = Krita.instance().createDocument(1, 1, "Document name",
"RGBA", "U8", "", 300.0)
activeView = Krita.instance().activeWindow().activeView()
Krita.instance().activeWindow().addView(newDocument) # shows it in the
application

activeNode = newDocument.activeNode()
activeNode.setOpacity(255) # this does not change the pixel alpha data,
just so our layer is visible


# create a new color in Python
# setting the foreground color needs a "Managed Color", so let's use that
colorRed = ManagedColor("RGBA", "U8", "")
colorComponents = colorRed.components()
colorComponents[0] = 1.0 # Red???
```

```
colorComponents[1] = 0.0 # Green???
colorComponents[2] = 0.0 # Blue???
colorComponents[3] = 1.0 # Alpha???
print(colorComponents) # the final values set
colorRed.setComponents(colorComponents)

# fill the canvas with our custom set color
activeView.setForeGroundColor(colorRed)
Krita.instance().action('fill_selection_foreground_color').trigger()
newDocument.refreshProjection() #update canvas on screen
```

For context on why the variable names imply that it sets the pixel to red, the code is from a portion of the lesson before informing the reader that the color space works in BGR instead of RGB in which this code assumes to be working in. Now, here is a script that achieves the same result in Progrimage utilizing a composite:

```
function Run(image)
	image.size = vec2(1, 1)
	image:setPixel(0, 0, vec3(0, 0, 255))
end
```

As is evident by these examples, Progrimage's scripting is much more succinct and easier to understand and reproduce.

Thirdly, the issue of poor documentation is solved by having code examples as well as a list of all datatypes and their methods publicly available on GitHub. This allows users to quickly see examples if they are just starting out, or if they are more experienced, allows them to quickly find the exact function or method that they need. The list portion of the documentation shows what is returned by each function, what objects have the method in question, what the inputs and datatypes of the inputs are for each function, and a description of the function or method. Many documentation sources for the scripting aspect of image editors that do support it do not show the users what is available. Many instead show examples or lessons intended for beginners as well as descriptions of the work process when creating these examples, but then offer little support beyond them.

Furthermore, users can create their own tools by clicking a dedicated button in the tool panel. The user can then edit the code for their tool and have access to various events such as pressing or releasing the mouse, moving the mouse, layer selections, and more. The user also has access to all the same features as they did with composites, but now with the addition of events and optionally running every frame. This allows them to create tools such as brushes with custom movement, tools that draw random shapes within the bounds creating by clicking and dragging, and is performant enough to implement a simple interactive Verlet integration-based physics engine simulating around 200 objects. Scripts also have access to a simple function that can be called to check if they should yield due to processing time. In Lua, there exist coroutines which are like functions but they can be yielded which pauses them and can later be resumed at the same position. By using coroutines, scripts can yield to avoid degrading the performance of the application. Moreover, the application runs the events inside these scripts within a coroutine, allowing users to utilize yielding without writing additional code.

A full list of features offered by Progrimage includes the basic tools, composites, and features that come with the editor, as well as the hooks and features available to user-created Lua scripts. Progrimage contains the following tools: a brush that can optionally act as a pencil tool, an eraser that can optionally act as a pencil tool, a fill tool with the options of contiguous filling, or only spreading to adjacent pixels, sampling all layers, erase or draw mode, replacing pixels instead of blending them, and the tolerance the maximum difference in color that allows propagation. There is also a move tool that can move layers or selections, as well as resize selections, a marque selection tool to select a region of a layer, a line tool that can remain horizontal or vertical if the shift key is held down, quadratic and cubic Bézier curve tools which have draggable points, a text tool with a font selector, and a tool to crop the image which, on

selection, selects just the region made up of non-invisible pixels. There are also rectangle and oval tools which draws rectangles and ovals. Holding the shift key will instead draw squares or circles. Additionally, holding the control key will use the starting point as the center as opposed to a corner.

The composites that come with Progrimage are a glow effect based on bloom rendering, HSV and HSL editors, a contrast editor, inverting pixel colors, making the image grayscale, removing the image's alpha channel and making it fully opaque, and multiplying an image's alpha channel. The other features include rerunning Lua composites and tools when a file change in the script file is detected, allowing simple and convenient real-time testing of scripts, showing errors with scripts when hovering over them, being able to drag and drop files onto the program's executable file to launch with these images brought into the canvas, being able to drag and drop files onto the program's window to add them, a style editor, saving as png, jpeg, or tga files, buttons to create, hide, and delete layers, buttons to hide, delete, and apply composites, resizing the canvas, image, or individual layers, the ability to select a region and copy it to your clipboard without having to save it to a file, zooming with the scroll wheel without clearing tools, panning the canvas by holding the middle mouse button and moving the mouse, and a bottom bar that display information such as cursor position on the canvas, layer and selection position and size, and zoom level.

The Lua hooks available to user-created scripts are different depending on whether the script is for a tool or a composite. For composites, the only available event is "Run" which executes whenever the composite should reapply. Meanwhile, tools have many events including tool and layer selection or deselection, mouse press and release for left and right mouse buttons, canvas and screen movements, entering and exiting the canvas, and an "Update" event that runs

every frame. For some features available to Lua scripts, there is creating and deleting layers and images, getting or setting the position and size of layers and images, setting a layer as the active layer, clearing images either with or without a color or bounds, resizing with a specific image sampling mode, expanding to contain a region, getting and setting individual pixels as well as getting and setting a table of all pixels, getting the portion of an image contained by given bounds, tinting an image, and 3 different drawing modes for compositing 2 images together. There is an “over” mode for simple alpha blending, there is a “mask” mode where the alpha of the second image multiplies the alpha of the first, and a “replace” mode where the pixels in the first image are replaced by the pixels in the second. There are also functions for drawing an arc, lines, rectangles, ovals, quadratic and cubic Bézier curves, and polygons, with thickness. Additionally, there are functions for drawing filled rectangles, ovals, and polygons.

6.2 Analysis

This program was written primarily in C# with some libraries used by scripts written in Lua. These languages were primarily chosen because of familiarity with them and ease-of-use. However, the disadvantages are that C# is not ideal for GUI, or Graphical User Interface, development and as such the graphics library used has some issues. NLua also has performance loss when calling between C# and Lua. The project idea was chosen because writing code to modify or create and image can sometimes be faster and easier than doing it manually and many image editors were found to be less than ideal for scripting. In retrospect, the applicaton has already proven to be useful and the preferred method for simple tasks. However, there are areas that could be improved such as performance and GUI reliability. If the project were to develop

further, more tools, composites, and Lua features should be added, as well as finding a more reliable graphics and GUI library, and utilizing the GPU for image manipulation. More basic features include changeable brush textures, allowing multiple theme files, saving and loading projects, user-created Lua scripts to create layers without needing to be in the form of a tool or composite, resizing layers with the mouse, snapping, flipping layers, output file specifications like JPEG quality and PNG compression level, optimized and more thorough undo and redo functionality, project tabs, and creating custom icons to replace the placeholders.

7. Conclusion

In conclusion, this application project phase was a first time for many things. As such, there are bound to be flaws. Despite this, the project has yielded a useful tool and provided knowledge of designing a user interface with C# without the use of Windows Forms components, running Lua in C#, and image libraries which are much better suited for image processing compared to C#'s Bitmap object. It has also shown that better GUI and/or graphics libraries should be used in the future to avoid potential issues of textures being blank in certain circumstances, resulting in the inability to use the application.

The purpose of this study was to create an image editor focused on scripting that is easy to use, easy for beginners to pick up, and has comparatively better applications of scripting relative to existing alternatives. This purpose has been achieved by making documentation public and succinct, providing scripting examples, and with the use of composites. Overall, this study has yielded a tool which provides a much more convenient method of generating or modifying images with the use of code.

References

1. “Adobe Photoshop Scripting Guide.” *Adobe*, Adobe, https://community.adobe.com/havfw69955/attachments/havfw69955/photoshop/551569/1/photoshop-scripting-guide-2020_unlocked.pdf.
2. “Chapter 13. Scripting.” *GIMP Documentation*, GIMP, https://docs.gimp.org/2.10/en/gimp-scripting.html.
3. “Image Data.” *Krita Scripting School*, Krita, https://scripting.krita.org/lessons/image-data#:~:text=When%20you%20want%20to%20start,a%20raw%20format%20called%20QByteArray.
4. Jansson, Erika. *Brush painting algorithms*. Chalmers tekniska högskola, 2004.

Appendix

I. GitHub

Jacob Brandt, Progrimage, https://github.com/Jacbo1/Progrimage

Notice: Repository does not include any proprietary libraries. The vector and direct Bitmap libraries are not available on GitHub but they are included in the repository's releases in .dll form.