Desktop app for image to ASCII art conversion

Ivan Menshikov

ČVUT–FIT

menshiva@fit.cvut.cz

November 30, 2020

1 Introduction

The main task is to project, develop and build a desktop application for converting images of supported formats to ASCII art.

Program reads data from one or multiple images of common format (JPG/PNG/PPM/PGM) and validly converts them into sequences of ASCII symbols. In case of multiple loaded images, program will have simple options for playing the animation (start, stop, show specific frame, delete frame).

Application will also have some features:

1. Light/Dark theme
2. Personalized ASCII art style
   - Program suggests user to input symbols, from which all ASCII arts will be drawn.
3. ASCII art symbols size adjustment
   - Because loaded image can be very big (in case of its size) program is capable of resizing image for valid displaying its ASCII art on screen.
4. ASCII art exportation
   - User can copy resized ASCII art from application to clipboard (and share it with his friends for example). Or if user needs ASCII art of his image in its full resolution, program can export this ASCII art into external text file.
5. The **key feature** is that user can apply multiple effects from list below:
   - Contrast
   - Negative
   - Sharpen (convolution)
   - Emboss (convolution)

2 Input/output data

The only input data application can access and read, are image files of supported formats (.jpg/.png/ .ppm/.pgm).

When user decides to add an image, a file browser opens and shows only images of supported formats in current directory. File browser asks user to choose an image which user wants to add and sends path of chosen image to application.

The only output data application can access and write are text files.

Application is capable of drawing ASCII art in its own window. It is good for "quick take-a-look" at resulting ASCII art with all effects and symbol style that user applied to his loaded image. But it is bad idea for showing ASCII art in full image's resolution. For example if user loads an image with 3840 x 2160 pixels (4K resolution), we will have ASCII art with $3840 \cdot 2160$ symbols in it, which (probably) may not fit in app's window. In this case a new function was added: exporting ASCII art in full image resolution to text file.

When user decides to export an ASCII art, a file browser opens and suggests user to choose a path where new text file should be created (or rewrite if file already exists). File browser creates new file by itself in chosen directory and sends its path to application.

3 Program logic/methods/algorithms and results

For this project I decided to use module-based architecture with these modules: GUI (graphical user interface), Factory and Image.

3.1 GUI

The biggest module in project. This module provides user interface logic. It stores and controls all views and windows. Manages connection between

main program logic and user interface interaction.

I decided to use *Qt framework for Python* [1] because of its simplicity and ability to work with *Google's Material design* [2].

General part of this module is written not in Python, but in QML language. QML files provides the language and infrastructure for visual components (views), model-view support, animation framework and etc. In short, QML files allows us to define design and basic logic between views and windows for our GUI.

Application has 1 window and 2 popup-styled windows (dialogs).

1. Main window holds 2 other windows and can open/close them. Main window is used for displaying ASCII art, showing list of all loaded images, removing images from list, exporting ASCII art and displaying animation.

2. "Add image" dialog is used for adding images to application. In this window user can tag his ASCII art with name, apply effects on it and take a look at preview of resulted image with applied effects.

3. "Settings" dialog is used for switching between Light and Dark themes, suggesting user to input his own ASCII art symbol sequence and changing delay between animation frames.

I would say, that the most difficulty part of this module to work with is **Qt Slots and Signals**. They are used for communication between objects and Python logic.

In GUI programming, when we change one widget, we often want another widget to be notified. More generally, we want objects of any kind to be able to communicate with one another. For example, if a user clicks a Close button, we probably want the window's close() function to be called.

Other toolkits achieve this kind of communication using callbacks. A callback is a pointer to a function, so if you want a processing function to notify you about some event you pass a pointer to another function (the callback) to the processing function.

The other difficult problem, which I had to solve were **Threads**. Threads are used for animation and image to ASCII art conversion (second is explained in subsection 3.3 Image).

If user loads **at least** 2 images to application, the "Play animation" button in main window activates. Every time "Play animation" button is clicked, a new thread is created. We pass art factory (more at subsection 3.2 Factory) to this thread. Animation thread is used to iterate through all loaded images in factory with delay (specified by user in "Settings" window). Every iteration thread changes currently drawn ASCII art with the next one from factory and events GUI with Qt Signal to redraw it. When "Stop animation" button is clicked, main program kills animation thread and removes it from memory.

3.2 Factory

Image controller module. This is a general module for images factorization. It stores all loaded images and can control (add, update, remove, count etc.) them. Manages connection between main program logic, loaded images and GUI.

Factory class is derived from QAbstractListModel because we pass class' object to main window's QML's list in GUI, which displays all loaded images with its names and previews.

3.3 Image

The most interesting and complex (in my opinion) module of the project. Data module for all supported image formats. This module holds all data of image file, needed for raw format without any compression. Operates with image in common way, such as: read from path, convert to ASCII, add effect etc.

When we create Image object, we should pass image's given name, path to image file, flags indicating which effects are applied and grayscale level (symbol sequence from which ASCII art will be drawn).

On creation, object reads image file from path, defines its suffix (image's format) and uses specific algorithm for image file decompression to "raw" pixels data. For that purpose I decided to use Python library, called *imageio* [3] because of its simplicity and ability to work with *NumPy* [4]. Then program gets image's properties such as: height, width and color space. By color space program detects if conversion from RGB to Grayscale is needed or if alpha channel (which is used in PNG format) needs to be truncated.

Other application's modules are able to call Image module's functions listed below.

3.3.1 Get ASCII art

We pass the container's width and height in which we want to draw ASCII art. Function resizes / fits our ASCII art to this sizes. Here I used some useful algorithms and optimizations:

1. **Image scaling, preserving aspect ratio**. For this purpose I used one of the simpler ways of changing image size: **Nearest-neighbor interpolation** [9]. We replacing every pixel with the nearest pixel in the output. This can preserve sharp details in pixel art. 'Nearest' in nearest-neighbor doesn't have to be the mathematical nearest. My implementation is to always round towards zero. Rounding this way produces fewer artifacts and is faster to calculate.

2. **Using cache for resized art**. We check if cached ASCII art has the same size as the container. If it has, we simply return this cached data. Otherwise we resize "raw" data and save it in cache. This improvement helps to display ASCII art (which was previously displayed) much faster and smoothly (especially for animation).

3.3.2 Convert to ASCII art

General function for ASCII art conversion. Updates grayscale level (defined by user in "Settings" window), applies all effects on image and converts it from RGB to Grayscale if needed. Uses **image to ASCII art conversion algorithm** (more below).

Algorithms, used in this module:

Image to ASCII art conversion algorithm.

1. Apply *effect algorithms* (more below) on image's raw data, based on flags, indicating which effect should be applied. This is the most complex (in case of performance) function. Before this function was created, I decided to move its calling to new background thread. So, when user clicks on "Add image" button, a new thread is created. This thread performs all complex computations on background, so GUI will not "freeze". When the image is converted, main program logic kill this thread.

2. If image uses RGB color model, then apply *RGB to Grayscale algorithm* (more below).

3. Convert image's gray data to needed ASCII symbol, defined in grayscale level in "Settings" window. Grayscale level is a sequence of symbols from darkest to lightest. So, for each image's gray pixel we compute its brightness by dividing its (pixel's) value by 255. Then we multiply this value by grayscale level's length. These computations gives us needed ASCII symbol for each gray pixel, based on pixel's brightness.

RGB to grayscale algorithm.
In the first version of application I used simple and straightforward algorithm where we apply coefficient (0.30 for red channel, 0.59 for green channel and 0.11 for blue channel) on each channel of the image and summarize them all into one value:

$$Gray = 0.30 \cdot R + 0.59 \cdot G + 0.11 \cdot B$$

But then I realized that we have more complex algorithms that would work better in our case (conversion to ASCII art), so I implemented **Colorimetric (perceptual luminance-preserving) conversion to grayscale** [6], which is able to "distinguish" the most lightest and darkest parts in image:

1. Firstly, we compute gamma-compressed value for each component (red, green, blue):

$$C_{\text{linear}} = \begin{cases} \frac{C_{\text{srgb}}}{12.92}, & \text{if } C_{\text{srgb}} \leq 0.04045 \\ \left(\frac{C_{\text{srgb}}+0.055}{1.055}\right)^{2.4}, & \text{otherwise} \end{cases}$$

2. Secondly, linear luminance is calculated as a weighted sum of the three linear-intensity values:

$$Y_{\text{linear}} = 0.2126 R_{\text{linear}} + 0.7152 G_{\text{linear}} + 0.0722 B_{\text{linear}}$$

We can see the difference between these two algorithms on this example:

Figure 1: Channel coefficients algorithm (left)

Figure 2: Luminance-preserving conversion (right)

Negative effect algorithm.
The simplest algorithm of all effect's algorithms.

1. Iterating each pixel in image's "raw" data without conversion, we subtract pixel's value from 255.

Contrast effect algorithm [7].

1. Firstly, we define contrast level (from 0.0 to 255.0. I prefer using maximum contrast level for the best image's parts qualification.

2. Secondly, we count real factor for our contrast formula:

$$factor = \frac{259.0 \cdot (level + 255.0)}{255.0 \cdot (259.0 - level)}$$

3. Thirdly, we apply factor on each pixel of image data:

$$C_{contrasted} = (C - 128.0) \cdot factor + 128.0$$

Convolution effects (sharpen and emboss) algorithms.
The most complex algorithm of all effect's algorithms. Convolution is one of the most important operations in signal and image processing. It could operate in 1D (e.g. speech processing), 2D (e.g. image processing) or 3D (video processing). Each convolution operation has a kernel which could be a any matrix smaller than the original image in height and width. Each kernel is useful for a specific task, such as sharpening, blurring, edge detection, and more.

In the first version of application I used straightforward algorithm:

1. Put the middle element of the kernel at every pixel of the image (element of the image matrix).

2. Multiply each element of the kernel with its corresponding element of the image matrix.

3. Sum up all product outputs and put the result at the same position in the output matrix as the center of kernel in image matrix.

Victor Powell's kernel visualisation [5] helped me with understanding how image kernels works.

But then I found article about relationship between image convolution and Fourier transform, which is called **Convolution theorem** [8].

In mathematics, the *Convolution theorem* states that under suitable conditions the Fourier transform of a convolution of two signals is the pointwise product of their Fourier transforms. So, from Convolution theorem, if we want to apply kernel effect g on our image f, we need to apply Fourier transform on our image and kernel effect, and then apply inverse Fourier transform on this product. If $\mathcal{F}$ denotes the Fourier transform operator, then $\mathcal{F}\{f\}$ and $\mathcal{F}\{g\}$ are the Fourier transforms of f and g, respectively:

$$f \cdot g = \mathcal{F}^{-1}\{\mathcal{F}\{f\} * \mathcal{F}\{g\}\}$$

So, based on this information I have implemented new algorithm using FFT (Fast Fourier Transformation):

1. Because multiplying between two Fourier transforms requires the same array size, we should *pad* kernel to image's sizes with zeroes.

2. The Discrete Fourier transform (DFT) and, by extension, the FFT (which computes the DFT) have the origin in the first element (for an image, the top-left pixel) for both the input and the output. This is the reason we need to use *fftshift* function on the kernel.

3. Based on *Convolution theorem*, we compute inverse FFT on multiplication of FFT on image and FFT on **padded** kernel.

This algorithm is much harder to understand, but it works **a lot faster** than the straightforward one:

Image size	Straightforward	FFT
128 x 128	0.12308 s	0.00261 s
256 x 256	0.45205 s	0.00908 s
512 x 512	1.94967 s	0.05010 s
1280 x 720	6.78832 s	0.18625 s
3840 x 2160 (4K)	58.58174 s	2.51028 s

4 Conclusion

I think this is one of the best projects I've ever had. I really enjoyed working with images and effect algorithms. Performance of application can be improved more (especially GUI, because it was my first time to work with Qt), more effects can be added and working with threads could be improved.

References

[1] The Qt Company. Qt for Python. https://wiki.qt.io/Qt_for_Python.

[2] Google. Material design. https://material.io/design/.

[3] Almar Klein. imageio Python library. https://imageio.github.io/.

[4] Travis Oliphant. NumPy Python library. https://numpy.org/.

[5] Victor Powell. Kernel visualisation. https://setosa.io/ev/image-kernels/.

[6] Wikipedia. Colorimetric (perceptual luminance-preserving)conversion to grayscale. https://en.wikipedia.org/wiki/Grayscale#Colorimetric_(perceptual_luminance-preserving)_conversion_to_grayscale.

[7] Wikipedia. Contrast effect algorithm. https://en.wikipedia.org/wiki/Contrast_(vision).

[8] Wikipedia. Convolution theorem. https://en.wikipedia.org/wiki/Convolution_theorem.

[9] Wikipedia. Image scaling algorithm. https://en.wikipedia.org/wiki/Image_scaling#Nearest-neighbor_interpolation.