

Alpha Matting in Gimp

Jan Rüegg, Johann Wolf

June 24, 2011

1 Introduction

Gimp is a freely available open source software for image manipulation. Though with much functionality the software still misses a decent foreground selection tool. The alpha matting algorithm described by Eduardo S. L. Gastal and Manuel M. Oliveira [2] promised to be the right choice for such a tool, having a good performance and accuracy. We show that it has clear advantages over the current foreground-extraction tool in gimp, that uses an older method called *Siox* [1].

The algorithm is characterized by being very fast with real time calculation capabilities. This is done by restricting most of the computations on a localized area and furthermore being able to do the computations in parallel on a GPU. Since Gimp only supports computations on a GPU in a special development branch, we decided to leave out these optimizations for the sake of providing the tool to a broader audience.

2 Algorithm

2.1 Working with Gimp

2.1.1 Data Layout

Before extending Gimp with a new tool there are many details that have to be considered. For performance reasons Gimp has a characteristic way of storing images, namely in tiles. These tiles are of the dimensions 64x64. Access to pixels through these tiles is much more efficient than through direct access. However, this layout has big implications on the ease of implementing an algorithm:

- If we want to access pixels in the neighborhood of a pixel, we have to either include a special case for the pixels on the border (since we access pixels that are outside of the tile) or we have to introduce a different data layout that is larger than the tiles themselves. This is necessary because only a limited amount of tiles can be loaded at one time, and there is overhead in loading and releasing tiles.
- Another issue is that not all the tiles are of dimension 64x64, because naturally the dimensions of images don't have to be multiples of 64. This imparts that we have to take into account special cases when using tiles.

2.1.2 Debugging

Since Gimp is implemented in C, the code runs very efficiently and there are many ways on how to optimize the code. However, working on such low level code imparts problems like hard to find bugs and more complicated ways of solving problems. Additionally we had to familiarize ourselves with many different libraries that are used, like *Gtk*, *Glib*, and the *Gimp API* itself, that has little to no documentation.

To be able to properly test and debug the code, we added many different debug features, mostly in the form of *defines* that can be disabled. For example, we wrote routines to save the content of the tile cache to a *.ppm* file, or to load and save the user selection from/to a file. That means, during testing, one doesn't have to draw the selection again and again. It is also possible to not only get the final image as a result, but also the estimated foreground, background or alpha value alone. Finally, we are able to stop the algorithm after each of the four steps to see the in-between results.

Additionally, we used various tools (like memory leak checkers and profilers) to improve speed and correctness of our implementation.

2.2 Alpha Matting

As a first step we generate a trimap from the input the user has provided, specifying foreground and background pixels. We take these pixels as being truthful, so it is important that these markings are made cautiously. From this basis all unmarked pixels look for a known nearest neighbor that has a specific minimal color distance and we assign that pixel to this known foreground/background neighbor. If no such neighbor can be found within range we mark this pixel as unknown, ending up with a trimap of foreground, background and unknown pixels.

In the next step we go through the unknown pixels of the first step. We look into four directions for a pixel, searching for each direction the closest foreground and background pixel. These directions are different for the pixels in the neighborhood, so that when we compare the retrieved values later on we will have implicitly searched in many directions for each pixel.

Once we've got the values - possibly four values each for foreground and background - we will find the best foreground/background pair to model our pixel color. This is done with an objective function, taking into account the distance of the found pixels, the fluctuations of the color gradient on the way of finding these pixels, and the chromatic distortion around the pixel in question.

After we've found a good combination of foreground/background colors for each unknown pixel we go through the entries again. This time we search for the three best color combinations for a pixel in its neighbourhood. Depending on the variance we take an average of these three values for the foreground and/or background or stay with the original value as an estimate.

Following up on this we retrieve the final background/foreground colors by performing a gaussian, weighted relative to the alpha value and the confidence of the colors. We then calculate the final confidence and use this result to calculate the final alpha value.

A few specific details of the algorithm as described in the paper were not entirely clear to us. Therefore, we contacted the authors of the paper. Eduardo Gastal was kind enough to provide the clarifications we needed, so that we were able to implement the algorithm in all details.

2.3 Implementation in Gimp

With a few minor differences mentioned below, we fully implemented all three steps (**Expansion of Known Regions**, **Sample Selection** and **Matte Computation** and **Local Smoothing**) explained in the paper. The implementation of the core algorithm is about about 1600 lines of code entirely written by us. Additionally, various *GUI* changes had to be made in many different files, that probably also add up to a few hundred lines of code.

As previously stated, Gimp uses tiles of size 64x64 to store the images. This complicates working in an area around a pixel (as it is required in this step), since in many cases we need to access pixels outside of the current tile. To simplify this and reduce frequent loading/releasing of tiles, we introduced a cache bigger than the actual tiles. Depending on the current step, we load either 1x1 or 3x3 tiles into the cache, work on the cache and save the results back.

To keep track of the unknown pixels, which are the only ones we have to process after the first step, we introduced a hash table with the x/y coordinates of the

images as a hash key and chained the entries together in a linked list. This speeds up the algorithm greatly by avoiding unnecessary access to already known pixels and thus prevents many tiles from being loaded into the cache. It also reduces the memory needed to process the pixels, compared to having whole image-layers for storing the intermediate results. However, frequently retrieving hash entries proved to be inefficient. Using a profiler we discovered that this amounted for up to 80% of the time for the computations!

To solve this problem we introduced a second, much smaller cache for the hash entries that is used during processing. After the cache has been processed, the data is written back to the hash table.

An optimisation we made as opposed to the original algorithm is when accessing neighbors of a pixel. The method described in the paper was always to search and compare other pixels in a certain radius. Instead we often did the comparisons in a square around the original pixel, making the access pattern easier and using the caches in a better way. The only downside is that a neighbor in the corner of the square might be selected instead of a spatially closer neighbor along the edge in certain cases.

Another difference to the paper is the range of the search for foreground/background pixels in the four directions. In the paper the step size is 6 pixels with a maximum number of 300 steps, amounting to a maximal distance of 1800 pixels for a search in one direction. Since this would be very computationally expensive and impart a large memory demand with our caching scheme, we restricted this search to a maximal distance of 64*3 pixel. In the future, this could be improved with a more intelligent access scheme for pixels outside of the cache, and therefore more accurate results for big images.



Figure 1: Input and result from a simple extraction

3 Results

3.1 Images

In Figure 1, one can see that the method gives quite good results, although there are still some artefacts left from bright background spots. The extraction was performed on a laptop with a touchpad, and took about 2 minutes for the image of size 800x533. This is including all user interaction and multiple runs of the algorithm. Running the algorithm once took about 5 seconds, with the image having 25% unknown pixels.

The comparison in Figure 2 clearly shows the advantages over the shipped foreground selection tool in Gimp. The algorithm called Siox [1] doesn't support semitransparency in extractions: The final alpha mask is only binary. This way, background color can be leaked into border pixels. Also, details like fine hair cannot be modeled well. This greatly limits the use of such a tool since such an extraction can also be done manually quite well.

Figure 3 shows how every step of the algorithm improves the result. We show, for every sub-step, the extracted foreground in front of a black background, the alpha mask and the estimated background and

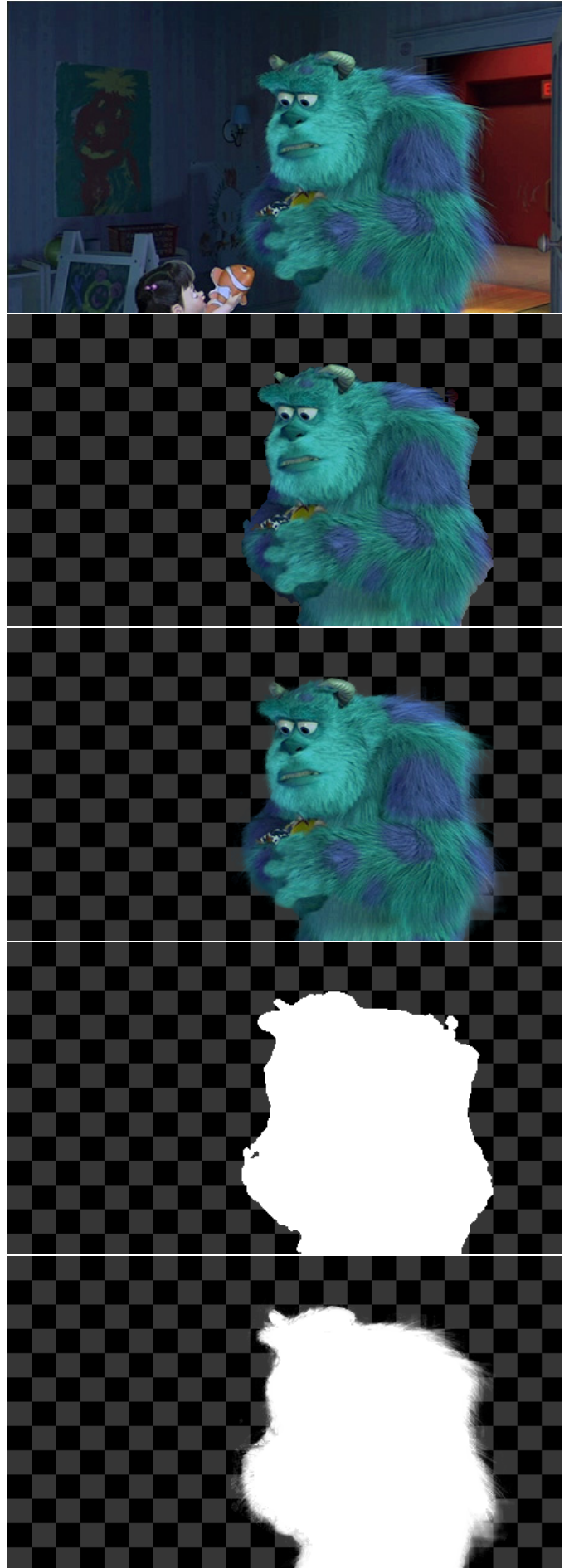


Figure 2: input image, extracted foreground from original tool and of our implementation and alpha mask of original tool and of our implementation

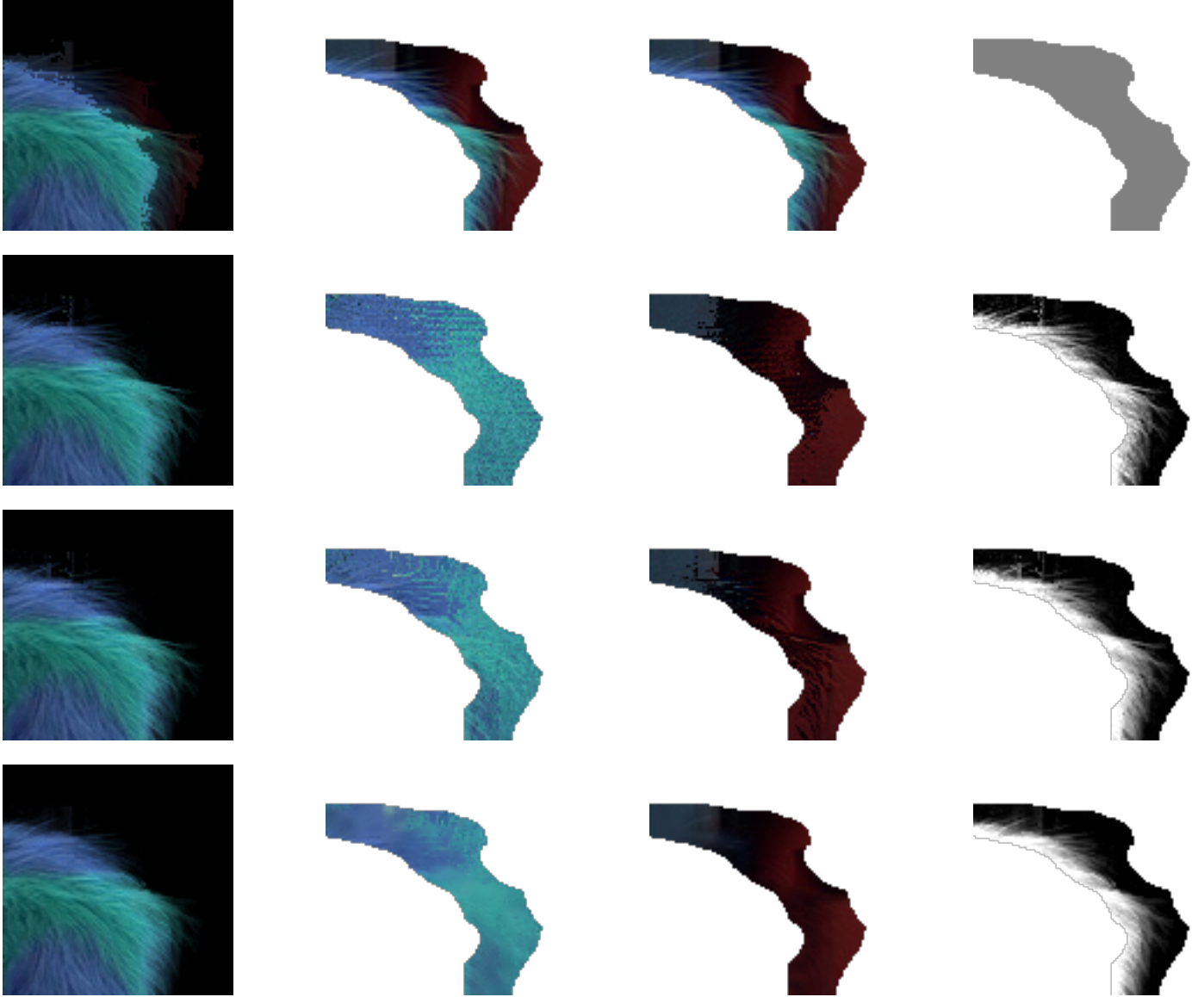


Figure 3: Parts of extracted foreground, alpha value and estimated foreground and background colors after each step of the algorithm (expansion, search, comparison, smoothing)

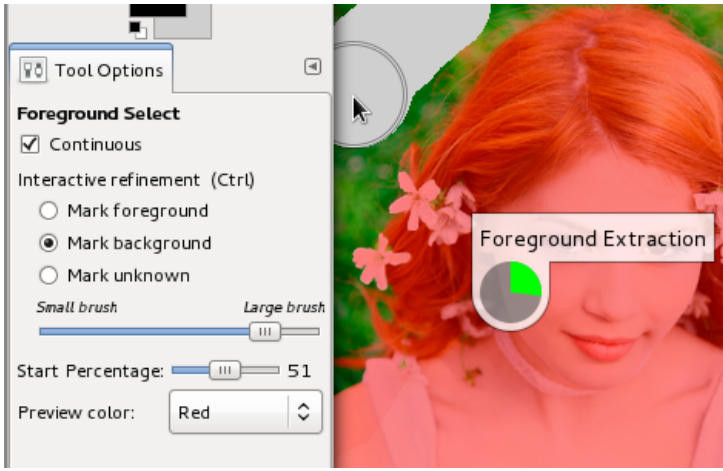


Figure 4: Implemented GUI elements

foreground colors. For the later three images the white regions to the left represent the foreground marked by the user and on the right the background. Combining the estimated foreground and background images according to the alpha mask should give the corresponding section of the original image.

3.2 GUI

Besides the algorithm itself, we implemented some GUI elements that give the user more interactive control over the tool, as you can see in Figure 4. Besides the brushing interface and the mask preview, that we partly reused from the old implementation, we added a percentage bar defining when to start the algorithm. As there should not be too many unknown pixels in a trimap-based approach, we do not waste time with computing initial rough strokes. We also added the possibility to entirely disable the calculations for refining strokes and to set regions to *unknown* again after accidentally marking regions wrong.

4 Conclusion & Outlook

The results are very promising and seem of good quality to us. The time for the computations is reasonable and the ability to tweak the results after a run are very convenient. Compared to the built in foreground selection tool in Gimp the difference is immense. However, there are still a few points on how we can improve on our tool:

First of all a more sophisticated (incremental) way of altering the trimap after a run of the algorithm could be provided. Currently after refining the trimap through user input, the whole algorithm will be run again on all pixels. In this case only pixels in the region of added strokes could be processed again.

More improvement could also be done to our painting interface. In contrast to our simple approach, a more sophisticated way could be implemented. For example filling holes, or even a flooding approach using detection of edges in the image, like mentioned in the paper. This would simplify the user input and might also improve the quality of the result.

For the means of speed there should be several ways to improve our code by using more efficient approaches. Furthermore we don't use parallelisation at all. However, multiple threads are supported in *Gimp* and it should be quite straightforward to make use of this for the algorithm.

Furthermore, we restricted our algorithm to certain types of images. *Gimp* supports many different colorspace, indexed images and images with and without alpha channel. Since our algorithm works on the raw image data, it relies heavily on the color layout, and currently assumes regular r/g/b data. However, it should be easy to extend it to other types of images, doing a simple conversion to r/g/b in between.

We're planning on including these changes into the official *Gimp* codebase, and will contact the *Gimp* developers about it in the near future. During our work we frequently pulled the upstream changes to ensure compatibility with the original code. We're hoping that our tool will provoke many positive reactions and that it will be included in a future Gimp release.

Code

The original code repository can be found under <http://git.gnome.org/browse/gimp/>

Our fork, including the changes, is hosted on GitHub <http://github.com/rggjan/Gimp-Matting>

References

- [1] L. Knipping R. Rojas G. Friedland, K. Jantz. Image segmentation by uniform color clustering. 2005.
- [2] Eduardo S. L. Gastal & Manuel M. Oliveira. Shared sampling for real-time alpha matting. 2010.