Visualisierungsinstitut der Universität Stuttgart
University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Studienarbeit Nr. 2369

Scale-Invariant Image Editing

Hendrik Siedelmann

Course of Study:	Computer Science
Examiner:	Jun.-Prof. Dr. Martin Fuchs
Supervisor:	Dipl.-Inf. Markus Kächele
Commenced:	April 1, 2012
Completed:	Oktober 1, 2012
CR-Classification:	I.3.6, I.4.0

Abstract

We introduce a system architecture for image editing which decouples image filtering from the image size, resulting in a system which allows interactive editing with constant response times, independent of image sizes. Scale invariance means filters are designed to allow scaled rendering from a pre-computed image pyramid, approximating the result of the filter when rendered at full resolution and scaled afterwards. Our implementation of the proposed architecture allows interactive editing on all image sizes with minimal hardware requirements, the least powerful device tested with a 300 megapixel image was based on a dual core ARM Cortex A9 clocked at 1Ghz. The architecture is based on a graph based image editing approach, extended by scaled rendering for all filters. The filter graph is exploited to allow automatic configuration of filter properties and conversion between color spaces, which simplifies filter implementation and increases performance. The handling of image data is based on tiles and a tile cache allows to manage memory requirements and increase interactive performance. The implementation is provided as a portable library written in c and can provides interactive editing on device as slow as last generation smartphones, while at the same time exploiting the performance available to current multi core processors, using effective multithreading. In this work we explore both the architectural details that make this possible as well as the properties of common image editing filters, regarding the required scale invariance. We also examine possible approaches that can be followed to implement practical filters for such as system. Finally, the implemented architecture and filters are extensively tested for performance and accuracy and the results are examined.

Contents

List of Figures

List of Tables

1 Motivation

Interactive image editing – the interactive manipulation of 2D raster images as possible with programs like Photoshop [Pho] or the Gimp [Gim], is typically implemented by a direct manipulation of 2D raster data. While typical hardware today has abundant processing power when working with the image sizes that an average digital camera produces, for this standard approach the time to process an image depends on the image size. Independent of image size, the size of the display, which allows the interaction in the first place, stays the same.

Digital consumer and smartphone cameras have resolutions from three to twenty megapixels, whereas a typical display has a resolution of around two megapixels. This mismatch between image size and display size is exploited by scale invariant image editing, which is a system architecture that can render filtered images at display resolution with a computational complexity that is constant with respect to the image size, and only depends on the display size. Scale invariant image editing takes advantage of the mismatch between screen sizes and the image size to speed up processing compared to other approaches that take place at the pixel level of the full resolution image. Previous approaches to speed up interactive image editing using scaled filtering either examined only single filters or applications ([PV95, BBS94, PV92, VP02, ZG08]) or relied on hardware acceleration and on-the-fly filtering [JJY^{+}11], without regard to the huge performance benefits that can be gained by using a cache to reuse previously rendered results. Our architecture not only implements the cache but also optimizes operation on the level of the filter graph which is the structure that stores the sequence of filters prior to the actual rendering.

While processing of small images is certainly fast enough on a typical computer, even without scale invariant image editing, typical image sizes are continuously increasing, while at the same time low power mobile devices become a greater focus for tasks commonly executed on computers or laptops. In addition to the mismatch between screen and camera resolutions, it is easily possible to create nearly arbitrarily large images by utilizing image stitching, seamlessly joining multiple images to create one large image, either by hand, or by using devices that automatically move a camera through a large field of view [Giga] and specialized stitching software [hug]. The goal of this procedure can be the capture of a larger field of view as possible with standard optics, the simulation of the shallow depth-of-field of a larger image sensor or simply the creation of a very high resolution image. Very large images are also produced by many other procedures, like large format photography using a scan back [Bet, WH04], sweeping panoramic scan systems [Pan] or by scanning an image obtained using a chemical film camera [Gigb].

The classic approach is not able to keep image editing interactive for such large file size and the mismatch between image and screen size is especially grave on mobile devices where processing power and memory is more limited, but which may nevertheless have to work with large images, like for example Nokia's pure-view 808 which features a 41 megapixel image sensor but has a display resolution of under 0.3 megapixels.

Because the task of rendering images to a display is a procedure that is completely independent of the actual image size, when using scale invariant image editing, this allows interactive image editing on relatively slow hardware like smartphones, and with any image size. Working with gigapixel sized image is not slower than working with only a few megapixel, a size difference of three orders of magnitude.

In addition to the architecture that is introduced, implemented and tested in this work, we examine the behavior of common generic filters with regard to their ability to be executed in a scaled manner, as well as generic approaches on how filters, which are not directly scalable, can be implemented using an approximation approach.

2 Problem Description

The problem which scale invariant image editing solves is that of interactive raster image editing. This means an image is manipulated with direct visual feedback of the result of this editing. Of special importance is the term interactive. As long as the computational complexity, of rendering the result of a filter to a display depends on the image size, it is impossible to guarantee interactivity, because it is always possible an image is used which is large enough to overwhelm any given processing hardware. To better outline what interactive image editing comprises, we will first break it down into a few definitions. Then we will define scenarios of possible user interactions and in the last section compare the different approaches to interactive image editing with each other, starting with direct and graph-based image editing and eventually introducing our scale invariant approach.

2.1 Definitions

Raster Image

For the scope of this paper we define a raster image as a two dimensional array of pixels, where each pixel is composed of a fixed number of scalar values. For simplicity we will from now on refer to this definition simply under the term image. We are less concerned with the actual interpretation of this array, what is important is that the image will be displayed and it will be edited. The means by which the image was acquired and other properties, like the sampling utilized or the spectrum represented by a color channel, are subject to assumptions by the user of the system and/or by the specific filters utilized.

Interactive Editing

Editing means that an image is manipulated using a filter, producing an output image where the value of each pixel depends on one or multiple pixels from the input image and on the filter utilized. Of great importance for us is the term interactive, which means that the result of applying a filter and changing settings on that filter should be visible to the user effectively immediately, by means of a graphical interface, as opposed to batch processing where images are processed with the actual raster data hidden from the user.

Scale Invariance

Scale invariance describes the ability of the system to render a filter at any given scale, independent of the full size of the image and with a computational complexity which is constant with respect to the size of the image. While filters may still use the size of the image to define the effect of the filter with respect to the filter parameters, a scale invariant filter must be able to approximate the result of the filter executed on the full scale and downscaled to the desired scale only by using a downscaled unprocessed input image.

Filter

A filter is an operation which creates a set of output images from a disjunct set of input images. The output depends on the input images, the filter used and the specific settings of the filter. A filter can have, but does not need to, a number of settings with which the behavior can be influenced. Either the input and the output set can be empty which represents filters that load raster images from disk or create them from something else that is not a raster image, and for filters that display the input image to the user or save it to disk, or do something else with the raster image that does not fit into the model of scale invariant image editing.

Filter Graph

Because the output of a filter is in most cases also an image, we can apply another filter, with the result that this sequence can be described as a directed graph of filters. Filters are the nodes in the graph and every node is connected to those nodes that represent the connected filters, where the output of a filter is the input of another filter. For now we will assume that this graph is acyclic. This might not be strictly necessary, a filter representing an iterated function system ([Hut81]) could effectively be calculated using a fixed number of iterations of the filter, to render a finite resolution image. Note that such a filter graph will in most cases be a chain if no composition filters are involved and it is not required to be a tree. Multiple “sink” filters filter are possible, which only consume image data and a source filter might be connected to multiple other filters.

2.2 Scenarios

The following scenarios are a rough description of sequences of system interactions that a user might execute to manipulate an image. We will use them as examples throughout this work, to show the properties of scale invariant image editing, as well as to demonstrate the differences between the scale invariant and other approaches. Our implementation also uses the scenarios as benchmarks, to examine the performance characteristics the actual implementation.

Scenario 1: Global Filtering

In this scenario the user is only interested in the global effects of the image. The image is viewed at a scale that allows the whole image to fit on the display, and a filter is applied with different settings.

Scenario 2: Pan and Zoom

In this scenario filters are applied at the beginning and after that the user moves the viewport around, viewing the image at multiple scales without changing any settings.

Scenario 3: Evaluate

This is basically a variation of scenario 2, multiple filters are applied and multiple positions in the image a examined. But in this case different settings are tested at every location before moving to the next.

Scenario 4: Redo

In this scenario a filter which is involved only with image detail (for example sharpen or denoise filters) is applied and evaluated at a single location at full scale. After that a global filter like brightness or contrast is added and examined at a scale so the image fits completely on the screen and after that at full resolution. As the effect of the global filter on the full resolution details becomes visible, the settings of the first filter are changed to accommodate the effects the global filter has on the image details. The idea of this scenario is to demonstrate that the result of image editing is a combination of multiple filters where full effect can not always be evaluated without all filters present. For example the result of the sharpen or denoise filter is not visible in the shadows of an image, but is only revealed by a filter which changes the image brightness.

2.3 Approaches to interactive image editing

In the following we will examine the two common approaches for image editing and look at their respective advantages and limitations. The first approach is direct image editing which basically means filters are simply applied to an image at pixel level, and because the result is another image, this procedure can be repeated. This direct approach does not actually handle any dependencies between multiple filters, it is just a direct application of the fact that a image filter results in a new image. The second approach, graph based image editing, understand image filtering as a sequence of filters and allows to apply filters only to a region of an image, automatically calculating which regions which filter requires for the desired output and therefore allows higher performance if only a small area of the image is requested.

At last we introduce scale invariant image editing and compare it to the previous approaches. Of great importance are the interactive aspects of the different approaches, especially regarding the computational complexity for scaled display, as that is something commonly happening in an interactive scenario.

2.3.1 Direct Image Editing

The standard method to manipulate images defines and implements a filter by the effect the filter has on the pixels of an image. The normal approach is to load the whole image in memory and then apply each filter one after the other to the whole image. Because this approach is normally not referenced by a specific term we will use direct image editing to describe this approach. The direct approach does not look at a succession of filters as a whole, but evaluates one filter after the other. This is simple to implement and possesses a very low overhead, if the job is simply to apply a specific set of filters and, in the end, to save the resulting image. This is also what is used by basically every free or commercially available image editing software, like for example the commercial Photoshop [Pho], or the open source program the Gimp [Gim].

An organization of the raster image in tiles or stripes and a mechanism similar to the page cache can be utilized to reduce memory usage and allow to work with image larger than main memory. But the important part here is that a filter is defined by the effect it has on the image on a pixel level, and because normally composition of image operations is not commutative, one filter has to process the whole image before the next filter can start its processing, which leads to a number of limitations associated with this approach.

Note that technically, direct image editing has been superseded by graph based image editing many years ago. For example the implementation of the graph based VIPS ([VIP]) was started in 1991. But common image editing software that is available today, still relies mostly on direct image editing, probably because it is simpler in the implementation. Also while graph based image editing provides some advantages, the one large problem for interactivity, the dependency on the image size for computational complexity is only partially removed by the graph based approach.

Limitations

Delays: Because the whole image needs to be processed, before the result can be displayed, direct image editing often results in long delays when using large images or slow filters. As a delay of even a few seconds can already be most hindering for interactive work, very fast hardware or the utilization of hardware acceleration is necessary to provide acceptable performance. To compensate for the delays, most applications facilitate a preview mode, where the results of a filter can be previewed before the filter is applied to the image. But this preview is often very limited, sometimes without the possibility to zoom out to view the whole image or simply by being very slow.

Memory Usage: The amount of main memory that currently available personal computers are equipped with, is more than decently proportioned to work with images that are obtained with recent cameras. Or so it seems at the first glance. As a base we assume 10 megapixel as the minimum that is provided even by some smartphone integrated cameras. With dedicated large format cameras the resolution goes up to around 100 megapixels. But high quality image editing often requires a bit depth of at least 16 bits per pixel and channel, so our baseline color image of 10 megapixels requires 60 megabytes of main memory. If we assume a factor of ten for the use of multiple layers, buffers for color space conversion, undo log and overhead we arrive at a value of 600 megabytes for a single image. A quick test shows this value as quite realistic, opening a 8 megapixel image in the Gimp shows a resident set size of 300 megabytes with 8 bit processing.

Using image stitching or scan cameras it is easily possible to create images in the range from 100 megapixels up to the gigapixel range, which would then result in memory consumption of between 1.6 to 16 gigabytes. In face of large images most image editing programs rely on mechanisms similar to the page cache to swap out image data from main memory to disk, but this will, in most cases, result in a much lower performance.

Undo/Redo: To allow for undo functionality old versions of the image need to be held in memory or on disk. To limit memory consumption only a few versions are normally kept, and while a change can be undone very quickly if the old version is still available, it is only possible to go backwards in time, not to change a setting on a single filter applied earlier. For an example, consider the redo scenario from section 2.2. To change the first of the two filters both filters have to be undone and re-applied with different parameters, wasting both the time of the user as well as computation time because the whole image was filtered two times and the result then discarded.

2.3.2 Graph Based Image Editing

Also sometimes found under the term out-of-core, streaming or on-demand rendering, *graph based* denotes all approaches which explicitly or implicitly manage filters as some kind of filter object without directly applying them to the image. Such systems are normally on-demand because it is not normally possible to know in advance which parts of an image will be needed, although it is possible to implement prefetching to render possibly required parts of the image before they are needed (see [PV02]). The term out-of-core usually references an approach where only a part of the whole image resides in main memory at any given point in time. While this is not an requirement of graph-based image editing, for example, it is conceivable to create a system where the image is rendered from a very large compressed image, with the compressed data being held in main memory, and such a system would have unique performance characteristics, such a system would not provide all the benefits that can be obtained by graph based rendering, as laid out in the following. Notable practical implementations of graph-based editing systems are GEGL [GEG] which is currently being integrated into the Gimp [Gim], and VIPS [VIP], which is more oriented towards batch processing, but also integrates a graphical user interface. See section 3.1 for more details on those systems.

Benefits

Memory Usage: Only temporarily buffers to hold the part of the image that needs to be processed are necessary. What area of an image is processed at once depends on the implementation, be it a row of the image or a block, possibly a bit more depending on the filter. The important part is that the memory consumption no longer depends on the image size, but on system and filter parameters, and arbitrarily large images can be processed with very moderate memory requirements. More memory can increase performance by using caching, but graph-based image editing does not need much memory to provide very high performance.

Undo/Redo: Because a graph based system has a more abstract view based on the filter graph, not on actual image data, to implement undo functionality it is only necessary to remember the changes made to the graph, not the changes to the image. As this is both very fast to do and also requires virtually no memory (all changes are done by the user, with a compact representation, the user will not be able to perform changes fast enough to actually fill up main memory), it is possible to implement a complete history, or to even allow for version control. Because the filter graph is explicitly defined, it is even possible to change any part of the filter graph, for example to change the first filter that was applied, without implementing an undo functionality, something that can be both computationally inefficient, and more work for the user, for direct image editing.

Delays: Because it is possible to process only a part of the image, graph based image editing is much faster to display the results. Performance only depends on the filter and the part of the image that is visible, so applying a slow filter to a gigapixel image and then viewing a small portion of that image, is as fast as with an image that is only the size of the screen. Also a preview functionality is not needed as the system only renders the portion that is needed for display.

Limitations

Processed export: While graph based editing can be faster compared to direct editing, in an interactive scenario, because only the visible part of an image has to be processed, this does not help if the whole image has to be processed. This is the case when a processed image should be exported from the system, for example to save it to disk. In this case it is necessary to process the whole image at full resolution. Memory consumption can still be much lower, but processing times do not benefit from the graph based approach in this case.

However, exporting does not have the same time constrains as compared to interactive editing as the user does not depend on direct visual feedback. Therefore the export functionality can be executed in background, allowing the user to continue editing of the same or another image.

Scaled display: When the complete image needs to be shown, and in interactive image editing this is often the case, for the user would not be able to visualize the effect of a filter to the whole image otherwise, the whole image needs to be rendered. While there is still the benefit of lower memory consumption, it takes the same amount of time to process the whole image, compared to direct image editing and this is not feasible with large images, because the time required for processing still depend on the image size.

2.3.3 Scale Invariant Image Editing

We have seen that the graph based approach possesses some nice properties which eliminate a few of the limitations of direct image editing. But for interactive editing there is still one big problem, the scaling. Displays have only a limited and fixed resolution, which is typically smaller than an average image, sometimes by some orders of magnitude. For interactive editing of such an image it is only necessary to show a fixed resolution, scaled window of the whole image. For a gigapixel image manipulated using a 2 megapixel screen, at any point in time we can only show a window with a size of 0.2% of the whole image at full resolution. If the whole image should be displayed, then it is shown at a reduced resolution.

To implement a system that exploits this resolution mismatch, we need a way to execute the filtering in a scaled manner which makes it possible to calculate the desired filter directly at the scaled resolution with the same result as if we applied the filter to the whole image and scaled down later. And this filter's time complexity must not depend on the full scale resolution of the window processed, but only on the actual pixel count of the processed window. The need to keep the time complexity independent from image sizes, requires the scaling to be precomputed, using an image pyramid.

The idea to exploit scale for image processing is ubiquitous in computer vision because the scale of objects and scenes are normally not known in advance, but the idea of exploiting multi-scale representations not only for computer vision tasks, but for image manipulation has also been proposed before, see section 3.2. However earlier works have mostly been limited to a single filter. But combining multiple scale invariant filters is not as simple as for the direct approach. For example many filters require, for the rendering of a small area, a much larger area of the input image and this means an infrastructure is needed that handles these dependencies, and schedules the rendering of parts of the image accordingly.

Benefits

Scaled display: While editing images it is often necessary to view the full image. Direct image editing or graph based image editing has to filter the image at full resolution, but scale invariant image editing can render any filter at the desired resolution, which results in render times which are completely independent of the actual image size.

Interactive for any image size: Because for very large images it takes too long to display filter results for approaches that do not utilize scaled rendering, scale invariant image editing is actually the only practical solution to work with such large images. At which point the other methods become irrelevant depends on the image size and the hardware and filters used. Nevertheless, while it is always possible to create an image that is large enough to become a problem for the other two methods, the render times of scale invariant image editing do not depend on the image size at all and thus it is not possible to find an image that is too large for the system.

All benefits of graph base image editing: Scale invariant image editing retains all the benefits of graph-based image editing because it is basically a graph-based system extended by scaled rendering. This means memory requirements stay low, and the availability of the filter graph allows non-destructive editing and changes to filters early in the filter graph.

Limitations

Approximation: For most filters it is not possible to find an exact way of rendering the filter at lower scales (see section 4.1.2), but it is possible to find an approximation. This can lead to problems, because the filtered image will then look different at different scales and care has to be taken that this difference does not become too large.

Filter implementation: All filters need to be able to work on multiple scales which makes the implementation of a filter for the scale invariant approach more time consuming and often very difficult, compared to the other approaches. But more grave is, that for many filters it is impossible to implement a scaled version, because the result of the filter depends only, or to a large part, on image detail that simply is not available any more, if the image is scaled. An example for this would be various edge filters, which cannot be implemented in the same manner for scale invariant image editing.

Processed export: Just as is the case for graph-based image editing, when a processed image should be exported from the system at full resolution, it is necessary to process the whole image at the full resolution and therefore processing time depends on image size. But this can also be executed in background as it is no longer an interactive task.

3 Related Work

3.1 Graph Based Image Editing

Graph based image editing is not a new concept, in “A model for efficient and flexible image computing” [Sha94], Michael A. Shantzis describes an efficient way to evaluate an imaging graph, which is a structure nearly identical to our filter graph as described in section 2.1. The main focus of Shantzis work is the way the required bounding-boxes are calculated for the graph, calculating the required areas before rendering, which is a bit more complex, compared to the the on-demand method of calculating the required input areas per tile when the tile is rendered, which is used in our architecture. Also tiling is deliberately avoided because of the perceived complexity for an implementation of tiled filters, instead a subdivision approach is chosen to limit memory usage. Looking at our implementation most efforts for filter implementations are directed at the scaling property of the filters, and tiling makes the use of a cache very simple which brings obvious advantages in an interactive setting. In his paper Shantzis considers the scale of images with regard to the composition of images with different scales, but not easily scalable filters are simply applied on the corresponding scale of the image and no effort is made to provide an approximation for a faster preview.

A notable implementation of graph-based image editing is the Generic Graphics Library (GEGL, [GEG]), designated to provide the new core of the Gimp’s [Gim] image processing, which is to date performed by direct image editing. GEGL is heavily influenced by Shantzis work, and while it provides high quality processing using floating point pixel handling, and performs automatic color space conversion if necessary, it does not provide the more generic automatic configuration capabilities (see section 5.3) that our approach incorporates. A tiled organization and a cache are utilized, but while GEGL allows for scaled retrieval of tiles, the actual rendering is done on the original scale, with the corresponding performance penalty.

A second graph based implementation, VIPS [VIP], is an image processing architecture, originally designed to allow processing of the very large images recorded in museums and art libraries for reproduction. VIPS utilizes two modes to process images but no cache. The first mode is a tiled access mode and the second a sliding window approach, where a window corresponding to the required scanlines is moved across the image. While VIPS provides very low memory consumption and extremely fast processing for arbitrarily large images, scaled filtering is not possible and the included GUI will allow scaled operation, but is very slow for large zoomed-out images because the corresponding full size window needs to be filtered by VIPS.

3.2 Scaled Image Editing

Due to the memory and performance constraints of direct image editing, scaled processing of images for interactive image editing has been proposed before, often in the context of interactive painting. While many works are based on a wavelet basis ([BBS94, PV92, VP02]), painting is also possible with a conventional image pyramid [PV95]. The scale of images can also be exploited to provide improved image processing compared to a filter which operates only on a single scale [ZG08]. But while those works show the possibilities which scaled image editing can provide, what is still missing, to integrate multiple scaled filters, is a common framework that allows the concatenation of multiple such filters.

3.3 Scale Invariant Image Editing

Display-aware image editing [JJY$^+$11] is a solution to scaled image editing with the same premise as that of our scale invariant image editing. The application is mainly panorama stitching and editing and the system provides tools for stitching, region cloning and to change contrast, saturation and brightness. However instead of the explicit filter graph used by graph based image editing and by our approach, all operations are kept in a list and applied on-the-fly when the viewpoint changes. Instead of the region calculation as used in Shantzis work or in our implementation, the system processes an region larger than the actually displayed window and displays only the center to hide the corruptions caused by missing neighboring image data at the borders. The image pyramid is pre-computed, using a Gaussian pyramid. To allow for interactive frame rates, hardware acceleration and operator reduction [LJP11] are utilized to combine multiple operations into equivalent single operations to speed up performance. While providing the same time complexity as our approach, the choice of on the fly calculation and hardware acceleration imposes much higher hardware requirements, utilizing a Intel Xeon 3.0 GHz CPU and a NVIDIA Quadro FX 5800 GPU for interactive frame-rates on 1 to 2 Megapixel screens. In contrast, thanks to the tile cache, our implementation can provide interactive pan and zoom on a dual-core ARM Cortex A9 running at 1.0GHz and only requires longer reaction times when the filter graph is changed, with an interactive scaled preview that makes effective use of the scaled infrastructure.

4 Scalable Image Filters

In order to implement scale invariant image editing it is necessary for all filters to be able to work on lower scales with an accordingly higher performance per full resolution image area. In this chapter we will first have a look at the properties a filter must provide to be usable in a scale invariant image editing system and then examine a few generic filter types for those properties. The last part is a short overview over approaches that can be followed, in order to implement a filter that is not directly scalable by virtue of the type of the filter.

4.1 Requirements

Normally when the result of image editing should be displayed, with direct or graph based image editing, the filter renders the required area at full resolution, and if it does not fit on the screen the result is scaled afterwards. Therefore scale invariance of a filter in the context of scale invariant image editing means the filter must be able to provide the same or a similar output when the two steps of filtering and scaling are reversed. The filter can be aware of the scaling that is taking place, but to calculate a scaled output, the input is taken from the image pyramid and the scaling is precomputed.

The scaled output does not need to be identical, but is has to be close enough, that for a user utilizing the system, the full scale result is not unexpectedly different from the scaled rendering.

Because the computational complexity must only depend on the number of pixels filtered, not on the corresponding full resolution window, it is not acceptable, even though possible, to calculate a down-scaled version by up-scaling the low resolution input, applying the filter and afterwards down-scaling again, as this would be too slow. We will define a filter as being scale invariant in the scope of scale invariant image editing if it satisfies the following two properties.

4.1.1 Computational complexity

We desire an implementation that is unaffected by image size, and only depends on the actual pixel count processed. Thus a scale invariant filter must have a scaled implementation which has a linear time complexity with respect to the actual number of pixels displayed, meaning the number of pixels of the window from the full scale image that is to be rendered, in display coordinates, not the number of pixels the displayed window contains in the full resolution image.

4.1.2 Scaled accuracy

The goal is interactive image editing. Because of this fact it is not necessary to require the result of the scaled filter to be identical to the full-resolution, down-scaled filter. For this reason we do not require the scaled result to stay within a fixed interval from the reference full resolution result, but we require that the accumulated difference over a small region must not exceed a fixed limit over multiple scales.

In practice this requirement results in the ability to utilize *supersampling* to avoid large deviation from scaled to full-resolution output. From the above requirement it follows that only details, areas with high frequency changes, may have a notable deviation, else a larger region would be affected. What passes as high frequency at one scale is of a lower frequency at a higher scale and thus supersampling can reduce the deviation. It is also unlikely for the user of an interactive image editing application to closely examine the details on a low-scale representation, more likely the user will zoom in to examine details and thus those details can be rendered with greater accuracy at the a higher scale. A user of our scale invariant system simply has to be aware of the accuracy problems that can arise on image details, but given the large amount of software that does not even take gamma correction into account when scaling ([gam]), but with which users still work without large problems, this should not be a huge issue.

For practical purposes we can measure accuracy as the maximum difference between the downscaled image filtered at full resolution and the direct renderings of the scaled down filters. This does not yet take into account the allowed difference over a small area, and therefore we additionally compare the reference with the scaled filter after applying an extra scale-down step.

4.2 Definition of Scale

While the proposed implementation uses a simple mipmap for performance and compatibility reasons, we will investigate scaled filters with a more generic definition. A down-scaled version of an image is itself an image where every pixel is the weighted average of a local neighborhood in the original image. The size of the neighborhood depends on the scale and therefore the number of pixels averaged depends on the scale. Every s_i pixel from the scaled image relates to the unscaled pixels p_j as follows:

$$s_i = \sum_j \alpha_{ij} p_j$$

with the premise that

$$\forall i : \sum_j \alpha_{ij} = 1$$

4.3 Generic filter types in the presence of scaling

4.3.1 Linear Point Operations

The most basic operations for image editing are *point operations*, where each output pixel depends on only one input pixel. A linear point operation is directly scale invariant simply by applying the same operation at any given scale as we will show in the following. For perfect scalability we want:

$$f(s_i) = \sum_j \alpha_{ij} f(p_j)$$

where scale is defined as in section section 4.2 and f denotes the linear operation. We can easily show this using the homogeneity and additivity properties of linear operations, starting at the right side of the equation above.

$$\sum_j \alpha_{ij} f(p_j) = \sum_j f(\alpha_{ij} p_j) = f(\sum_j \alpha_{ij} p_j) = f(s_i)$$

In theory *brightness* and *contrast* filters are examples of linear point operations. However in practice even those are not actually linear, because image data is normally represented by fixed length integers and therefore we have to apply clipping to the operators which results in effectively non-linear operations.

4.3.2 Linear Local Operations

The normal notation of a linear operation in the sense of image processing uses a whole image as element and thus addition of images and scalar multiplication (scalar with the image) are compatible with a linear operation. We denote those operations as linear local operation to distinguish from linear point operations. The problem for scaling is, that the composition of linear operators is not commutative and therefore the assumption $s \circ f = f \circ s$ is in general not true (where f is a filter and s denotes the scale operator). This means we cannot simply swap scaling and filtering, as we can do for linear point operations.

4.3.3 Nonlinear Point Operations

Nonlinear point operations can be represented by a look up table and are in general not scalable. The reason for this is that every element in the look up table has no relation to any other element. Lets take an unfiltered scaled pixel with a value of a. This value is the average of a number pixels from the full scale image, but as long as these pixels do not all contain the same value, the entry in the lookup-table can be completely unrelated to the entry for a and therefore the scaled approximation can be arbitrarily bad.

4.3.4 Local averaging filters

We will say a filter preserves the local average, if there are two fixed neighborhoods with corresponding positive weights so that for every pixel the average of the unfiltered image weighted by the first neighborhood is the same as a the average of the filtered image weighted by the second neighborhood. For example for a Gaussian convolution the neighborhood and weights for the unfiltered image are the same as the desired convolution and the second neighborhood consists only of the pixel itself with a weight of one. The respective weighted averages are the same, by the definition of the Gaussian convolution.

Now if a filter is average preserving as defined above then the filtered and the unfiltered result naturally converge as the scale is lowered. The reason is, that as the number of neighborhoods, which are completely encompassed by the pixel boundary, increases as the scale is lowered, the difference between the filtered and unfiltered image decreases as a larger fraction of the weights that comprise the filtered pixels are completely contained in the scaled pixel. In reality this can only be applied when the neighborhood is sufficiently small, so that the first or second scale can be calculated by using the full resolution image and scaling down afterwards. For the rest of the scales the unfiltered scaled image can then simply be copied by the filter. If the neighborhood is larger, the overhead for calculating the much higher resolution filter is simply too much for a practical implementation, even if the complexity of the implementation does not increase, because the number of scales used for super-sampling is limited.

4.4 Generic approaches to approximation

As we have seen in the last section, that for most filter types it is not possible to get the scaling property directly from the type of the filter. Not even local linear operations stay scalable in the face of clipping caused by integer arithmetic. But because our definition of accuracy (section 4.1.2 above) allows us to use an approximation, we will have a look at two generic approximation schemes that can be used for practical implementations of filters for scale invariant image editing.

4.4.1 Approximate full scale

Image filters as used when ignoring scaling can be described by the effect they have on the pixels of the full resolution image. Therefore this approach takes the “bottom” layer of the image pyramid, the full resolution image, as a reference, and tries to implement the scaled filter so that lower scales approximate the effect the filter has on the full resolution image. This approach leans itself well to average preserving filters because as the scale decreases, filtered and unfiltered results converge. For this reason this approach was chosen in the implementation of the Gaussian blur filter, see section 6.7.4 for more details on the actual implementation.

4.4.2 Coarse to fine

As shown above it is impossible to directly approximate for example a linear operation like a sharpen filter. However we can change the actual filter to better fit the model of scale invariant image editing. For this we basically start at the topmost, coarsest layer of the pyramid and define the effect of the filter starting from the top layer with the restriction that every scale may only change the value in a way that does not change the lower scales. This approach is utilized successfully for scaled filtering in [JJY$^+$11], and this approach is basically what happens when image editing is executed on a laplacian pyramid.

Note that it is not necessary to make the filter completely accurate, it is possible to change the filter only so far as to make the approximation adhere to the accuracy restriction as define above, and therefore still allow for small differences between the full resolution and the scaled filter.

5 System Architecture

In this chapter we will introduce the system architecture with the specific design characteristics and the reasoning behind those features. The underlying principle is the same as for graph based image editing, but always with the goal of interactive editing as opposed to batch processing in mind. Because we are dealing with a graph based representation we make extensive use of the fact that we can work on the filter graph itself, additionally to just working on the raster data. We call this optimization at graph-level *autoconfiguration* and to provide the system with the informations that this autoconfiguration requires we utilize *Filter Specification Trees* which are a possibility for filters, to exactly specify their behavior and requirements, in a way that can be used by the system to coordinate their behavior and for example automatically convert between different color spaces between different filters. To simplify access to the raster data and to allow for caching, the actual image data is organized in tiles.

This chapter first highlights the low-level design decisions, like the choice of scaling for the image pyramid and the format of raster data. After that the Filter Specification Trees are introduced in detail and the autoconfiguration is examined. At last we look at the way the actual processing takes place and how the cache is organized.

5.1 Basic Considerations

As already mentioned all image data is organized in tiles, which are non overlapping rectangular blocks of raster data which are individually cached and processed by the system. Tiles are addressed by position and scale and when filter actually process data, the unit with which processing is handled is one tile of output. In the following we examine the other properties our architecture imposes both on a pixel level and on a the level of the filter graph.

5.1.1 Image Pyramid

An image pyramid is a raster image sampled in both space and scale. There are many ways to obtain lower scales from the original high resolution input. To obtain fast interactive behavior of the whole system the lower resolutions cannot be created on the fly, as this would not be possible within the complexity constraints, but need to be available in the raster image file. While there are different approaches to scale space, mostly based on the Gaussian function but also for example, on wavelets, the requirement for pre-calculated input makes a simple mipmap a suitable approach. The common use of very large raster images in geographic information systems (GIS) lead to the common availability of tools and support for mipmapped TIFF files. Also the ubiquitous JPEG image file format allows to speed up decoding by scaled decoding. But this does not mean that other approaches can not be used. As we show with the Gaussian convolution filter in our implementation, it is very fast to get an approximately Gaussian convolution from the mipmap, and other transforms might be possible analogical. With the support of both GIS-style multi-scale TIFF and the JPEG file format the system is able to directly read without preprocessing and without performance penalties two commonly used image formats.

5.1.2 Raster Data

Organization

To simplify handling of image data and filter programming, the actual raster data is organized in tiles. The image is made up of square or possibly rectangular tiles of a fixed size. No tiles overlap and to render an area the system calculates the tiles that make up the area and renders it tile by tile. The tile size can differ per filter in the filter graph and per scale, and the system takes care to copy the required data into the format required by a filter. Caching is also done per tile, with information like the last access time recorded per tile to optimize operation.

Tile Size

The tile size determines the granularity with which raster data can be accessed, and there are several considerations to make, like ease of implementation, overhead, granularity and CPU cache misses. For ease of implementation the tile size should be as large as necessary for a simple filter design. It does not make sense for example for a block-wise Fourier transform with a window size of n to use a tile size smaller than n as this would complicate implementation immensely. Tiles should be large enough for the overhead of tile handling to not matter much. Smaller tile-sizes incur much more work per area of raster data because all bookkeeping and all calculations outside of the actual filtering are done per tile. The cache performance depends on the granularity of entries in the cache. Larger tiles will give less room for cache decisions, because for the same amount of memory less tiles can be cached, and cause a higher overhead for areas that are rendered but not displayed. CPU caches are of limited size, if tiles are larger than those caches they might cause more misses which could slow down the overall performance.

Format

For performance and memory reasons the actual raster data is processed per default as 8 bit integers with a planar organization. Planar as opposed to an interleaved organization signifies that the samples of one channel of a tile are kept together in memory, where interleaved would mean all channels of one pixel are kept together, effectively interleaving different channels in memory.

Modern CPUs feature extensive vector processing capabilities. The smaller a single color value the more values can be processed with a single vector operation. The planar organization has two benefits. It is easier to vectorize the processing of non-interleaved image data, and as a second benefit, the system is able to handle channels separately from one another, which decreases the use of the cache where a filter only processes a single color plane, but also allows to better generalize filters that work the same on different color channels. Those can be programmed for grayscale images and then applied separate for each channel without bothering on different alignments of different channels as would be the case with an interleaved format.

5.1.3 Local view

In order not to limit scalability especially in the face of very large images it is necessary to not load any image-size dependent data structures into system memory. Similar to distributed systems, the system maintains no global view of the whole image. This means we can not have a full list of all tiles in main memory even if most of those tiles are only place-holders for the actual raster data. Also it is not permissible to hold all tiles, which are necessary to render a specific region, in memory, as this region could be arbitrarily large, for example when saving the processed image to a file. All parts of the system, especially those concerned with the actual rendering take this into account and there are special filter modes, for filters that need to process arbitrarily large areas of raster data, for example to save an image to a file.

5.1.4 Autoconfiguration

With autoconfiguration we describe a feature of our system that automatically mediates between filters to get the best performance. In contrast to other architectures, filters implemented for our architecture allow some of their properties to be set by the system. For example some filters allow working in more than one color space and in this case, the choice is not made by the user but by the system to minimize the number of color space conversions required.

Consider the following processing chain:

> Load - denoise - saturation - contrast - crop

In a naive implementation there would be a common color space, be it RGB or LAB, and for example the filter to change saturation would have to convert into HSV color space internally, change saturation and then convert back. Lets assume we have an image in the JPEG file format which normally uses YUV color space. The actual processing chain might then look like this (the output color space of the filters is indicated in brackets):

> decompress (YUV) - cs-convert (RGB) - cs-convert(LAB) - denoise - cs-convert (HSV) - saturation - cs-convert(RGB) - contrast - crop

Our architecture avoids such redundant conversions, which is possible because we can operate on the filter graph and insert only the conversions that are necessary. This means only filters that are really required are specified by the user, the system inserts the filters that do the necessary conversion. We will demonstrate the procedure with the following example input chain:

load - denoise - saturation - contrast - crop

From this, the system will automatically produce the following chain:

load - decompress (YUV) - denoise - cs-convert (HSV) - saturation - contrast - crop

There are no limits on the combination of filters, the system executes the necessary conversions automatically, even if it is necessary to insert multiple filters. This makes the system more versatile and easier to use. The system only inserts conversions when necessary, improving performance compared to a system which has a single common default color space.

This concept does not only apply to color space conversions, other conversions are possible, for example wavelet basis could be used by multiple filters, or some of the image processing could take place on the DCT blocks of a halfway decompressed JPEG image, allowing additional performance gains.

5.2 Filter Specification Trees

To allow the system to automatically insert the necessary operations we use a abstract way to specify image operations. Any filter possesses input and output specifications as well as settings and tunables. Input and output define what the filter can process, and state the assumptions or restrictions the filter poses on input and output. Settings are a way to allow the user to change the filter parameters and tunables allow the system to influence the mode of the filter (for example color space), which does then itself change input and output specifications.

To allow for a common structure, all specifications are organized as a tree, where every node possesses a data type, value pointer and a list of values and can itself point to multiple children. The system knows of a few data types which are used directly to influence system behavior, but most are only there to configure the chain correctly and to forward metadata from one filter to the next. See figure 5.1 for a small example of a specification tree.

A node can be of only one type but it has a list of values of this type. The meaning of this list depends on where the node is used. For a node under the settings tree, this means the user can select one of the multiple choices. Under the tunable tree this specifies the multiple possibilities the system can select and under the input and output tree this specifies all possible values the node may assume depending on the selected settings and tunables. This allows the system to check whether the output of one filter can be passed on to the input of the next filter if configured correctly, or whether it is necessary to insert a conversion filter. Before any processing of image data takes place, the system configures the filters, which means it selects the tunables that make every node that is connected to another node assume the same value.

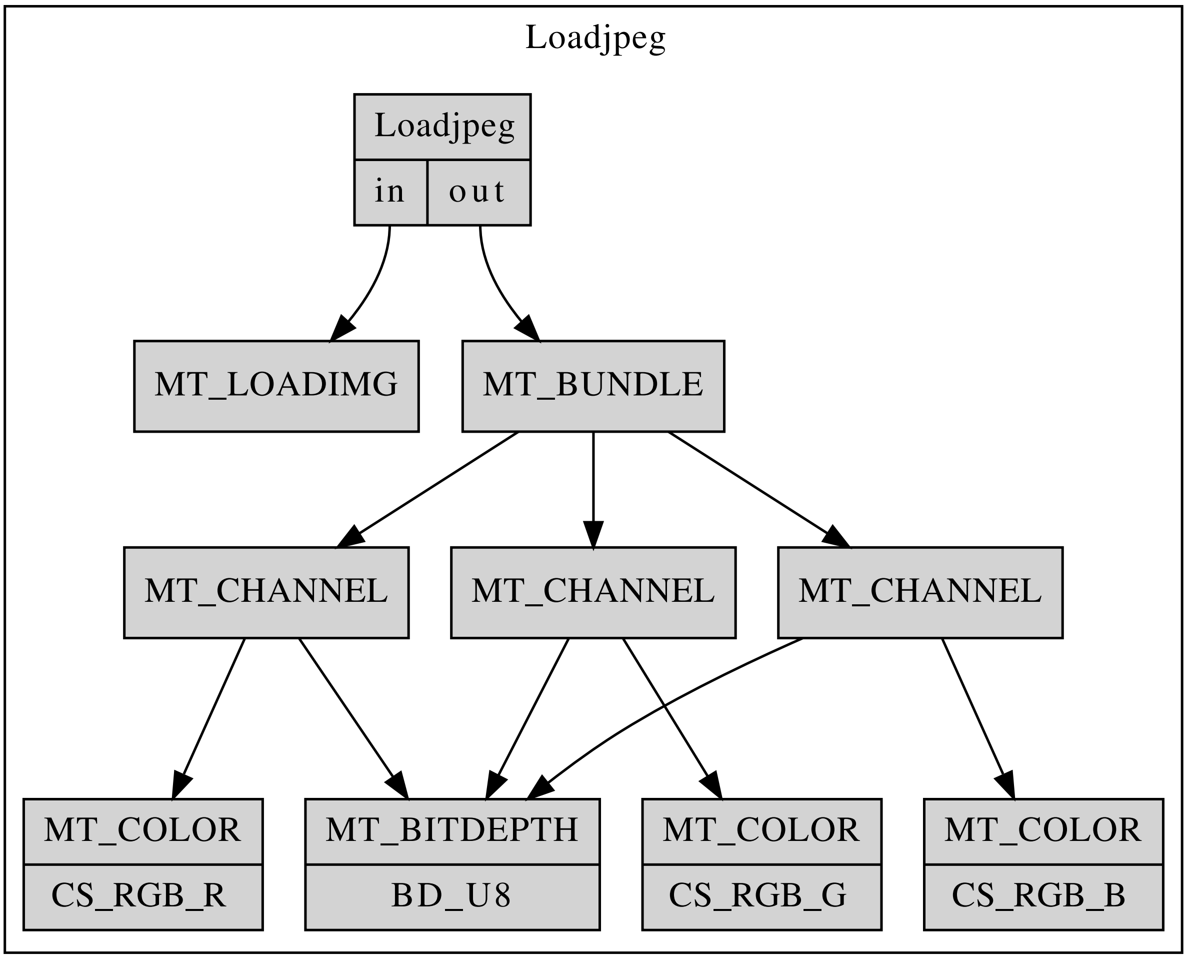


Figure 5.1: Filter specification trees as used by the Loadjpg filter. Input is of type MT_LOADIMG which actually uses a string type and simply specifies the image to load. The output tree contains 3 channels with different colors, but with a shared bitdepth node. This is a very simple example without tunables or settings.

The tree approach as opposed to a simpler scheme was chosen because it is very flexible and can allow both higher performance as well as simpler programming, as we will detail in the following.

Minimal Specification

Many operations are more or less indifferent to the color space they operate in. While the quantitative result is different, a blur operation can be applied to a grayscale image just the same as to an image in the LAB color space image. Because color space and bit depth are specified with separate nodes in our filter specification trees, such a blur filter would simply not specify the color space and the system can then apply it to any possible color space. Implementations of operations will only specify the assumptions made, restrictions that are left out give the system more possibilities to combine operations without extra overhead for useless conversions.

For example the contrast filter only requires its input to be a channel with a color space from one of the lightness channels of LAB, HSV or YUV. The system will automatically apply the filter only to one of the channels of the previous filter and forward the other channels to the next filter. While for most filters the result is different if the filter is applied in a different color-space, this difference is mostly numeric, the qualitative result is more or less the same. For example between LAB and YUV the L and Y channels are defined differently, but in both color spaces they represent some form of brightness as seen by human visual perception. So while changes to these channels will not have the same effect when translated to the respective other color space or a different third color space, the qualitative effect as the change in brightness for a human observer is similar.

From the point of view of a filter which should support all the color spaces mentioned above, which are sRGB, LAB, YUV and HSV and that is indifferent to the actual channel color space, like a blur filter, it is necessary to implement three different processing routines. One for linear channels, like the lightness channels or the color channels of LAB and YUV, one for gamma-corrected values for the sRGB color space and one for the angular values of the hue channel of HSV.

Optional Properties

Filters can provide any additional restriction or information that their output adheres to, for example an input filter could provide an exact color profile that was used to obtain the image if that was embedded in the image file and a color space conversion filter could use that. Or exif data from a file could be consumed by a denoise filter to optimize its operation. See section 5.3 for how the tree nodes are exactly handled to automatically handle the forwarding of such extra nodes. Filters that are not aware of those extra informations will neither see them nor need to handle them in any way. An example for this are the image dimensions. Point filters do not need to handle those, because they simply get provided with input and output buffers and do not need to worry about the fact that a part of those buffers can lie outside the image borders. Together with a fixed tile size this can make implementation of a simple filter even simpler compared to direct image editing, and at the same time faster, because with a fixed tile-size the compiler has more opportunities to utilize vectorization.

Pre/Post-filtering

The specification trees can also be used to force the system to do some pre- or post-processing on the raster data. This way a filter with multiple stages can easily be implemented, with the benefit that every in-between step is also subject to caching and the implementation can make use of the automatisms provided by the system.

5.3 Configuration

The details of handling specification trees are of course subject to the actual implementation, but we will outline the basic behavior. The input and output trees of each filter restrict the possibilities of which values the connected tree nodes can assume. The filter graph is specified by the user, but the user only specifies which input is connected with which output, not the connections of the nodes of those trees. When configuring the filter graph prior to rendering anything, the system goes form source node to the output/sink node and tries to match up all connected filters. The tunables of the filters are adjusted to those values that make the input and output trees compatible with each other. If this fails because, for example, one filter only works in RGB color-space and another only in LAB, then the required conversion filters are inserted between the filters and the system proceeds to the next pair.

5.3.1 Pairing

Pairing is the process of finding a match between the nodes of the output tree of the source filter to the nodes of the input tree of the sink filter so that the following restrictions can hold true.

1. All nodes of the sink filters input tree must have a connection.
2. If the parent of sink node B is connected to the parent of the source node A than any child b of B has to have a connection to a child a of A.
3. The values of the connected nodes must be identical.
4. Each node must be connected to only one other node.

The first rule ensures that all input nodes are provided with a connection. It is not necessary to connect all output nodes, because outputs nodes can be optional, but the input nodes state the restrictions and assumptions a filter makes for its input, and not having a connection for an input node therefore would mean, that a restriction does not hold true. The second and third rule are necessary, because children of nodes detail the specification of the parent node, a connection from a sink node to a source node with the same value means this specification is the same for source and sink. An example of this is the color space between source and sink. color space is a child node of a channel node and thus a connection between two color space nodes with the same value means the parent channel nodes utilize the same color space. The fourth rule is necessary because else “strange” pairings might be possible where for example one source channel is connected to multiple sink channels, while the other source channels are not connected at all.

Note that it is not necessary for all nodes to actually be connected, the specified rules allow a subtree, starting from the output tree root node of the source filter, not to be connected, as long as all sink nodes are connected. This is intentional as it works in concert to the forwarding of those unconnected nodes (see section 5.3.2 below), to allow filters to only specify the minimal specification trees on which they operate. For example the contrast filter’s root nodes for the input and output trees are single channels, with a color space child node specified as for example LAB_L, which refers to the lightness channel of the LAB color space. The filter that is connected as source to the contrast filter might have a tree that has type `MT_BUNDLE` as root with children that have color space of LAB_L, LAB_A and LAB_B. Only the one channel of the contrast filter and its children are then connected and the root and the other channels are forwarded to the output of the contrast filter.

5.3.2 Forwarding

Nodes of the output tree of a filter A that are not connected to any node of the input tree of the next filter B are forwarded and joined with the output tree of B. The unconnected nodes can be either the subtree starting at the root of the output tree of A or individual subtrees from the output tree of A whose root has a connection to the input tree of B but who themselves have no equivalent in the input tree of B. To coordinate the forwarding, for every node in its input tree a filter can specify a corresponding node in its output tree. The tree that is then actually used as the output of B is constructed using this replacement specifications together with the output tree of B and the unconnected nodes from the output tree of A.

5.3.3 Virtual Output Trees

As stated in the previous section the tree used to connect a filter B to the next filter is not actually the output tree of the previous filter but is constructed from the output of the previous filter A and the output of B. The resulting tree is the virtual output tree of B and will be used to pair B with the next filter. For every node that has a connection from A to B the specified replacement node of the output tree of B is used. A node might not have a replacement in which case that node has no corresponding node in the output tree and is left out. Nodes of the output tree of A might not have a connection, in which case they need to be forwarded to the new output tree so the next filter can work with them. To place them the parent/child relationship of the output tree of A is exploited and so every node that of the output tree of A has either a connection to an input node of B or is forwarded to the virtual output tree of B. Note that it is not necessary for the root node of the output tree of B to be the root node of the virtual output tree. If the root node of the output tree of A has no connection to B then this node is the root node of the virtual output tree, and the root node of the output tree of B is simply a child or grandchild of that node, depending on which node from A was connected to the input node in B that had the root node of B's output tree as replacement node.

5.3.4 Tunables

As nodes from two filters are connected so are implicitly the tunables that control those nodes. If only one of the nodes has a corresponding tunable than the remaining tunable is forced to one value and can be left at that. But if both nodes are controlled by a tunable then the two tunables effectively merge to form one new tunable that controls all nodes that were previously controlled by the independent tunables. In line with virtual output trees those tunables are virtual tunables because they are not actually part of any filter, but work the same as a conventional tunable when the virtual output tree is paired with the next filter.

5.3.5 Conversion filters

A color space conversion does not change the image contents like other filters would do, but only changes the representation. All filters that change the representation will be addressed as conversion filters, as they do not process the image content. Such filters will not be inserted by the user, normal filters specify the representation they need or assume and the system then inserts the appropriate conversion filters to convert between those representations.

Color-space conversion was already mentioned but another example are the filters to read image files. The user only inserts a “load” filter whose output is a specification tree with an output of type “loadfile”, but without any channel nodes. The load-tiff and the loadjpeg filter have an input of type loadfile and output channels and are inserted automatically depending on the file format. The user does not have to bother with the file format, just like the color space, can be ignored. Another example for a conversion filter would be a demosaicing filter for the Bayer pattern as used by most digital still cameras. Filters to load different raw camera formats could output the raw Bayer image data and the demosaicing filter would be inserted automatically.

Taking this idea even further the filter to load files with the JPEG file format could consist of several conversion filters for the lossless decompression, dequantization and inverse dct and other filters could then directly work on dct coefficients or even quantized dct blocks to further increase performance.

5.4 Filter Processing

The processing of a filter graph is started by the request of the application to render a specific tile of the graph, normally due to user interaction. The system will then check whether this tile is already in the cache, and if it is, directly return the tile. If a tile is not available the system will calculated which tiles are needed as input by the current filter for this output tile, and for every one of those tiles, will then recursively apply this procedure until the initially requested tile is rendered. The output of a filter is always a tile, but the input depends on the filter mode, see section 5.4.2.

5.4.1 Input/Output

The unit or granularity used to process images is that of a tile. This means the output of a filter is always a tile. Because tile size can be specified by the filter itself, the tile size can always be selected to suit the filter best. Processing is always out-of-place, the output tile is never the same as the input buffer, this allows the system to determine which steps are best cached and which can be discarded. This can also allow the compiler to perform a few additional optimizations if advised correctly on pointer aliasing. Input is also in tiles but size may vary depending on the area requested by the filter. As the channels are handled independent of each other, the filter gets as many pointers to input and output tiles, as there were nodes of the channel node type in the input and output specifications of that filter.

5.4.2 Modes

For a filter to actually render a tile, the area that the filter needs for the specified output area has to be provided. We use two distinct filtering modes, which are two completely different approaches to how filters can request and process raster data. With those two modes every type of filter can be implemented, as we will show in section 5.4.3.

Buffering

This is the simpler method, buffering means the filter gets the needed part of the input image in one big buffer allocated by the system. To allow the usage of this method, a filter has to provide a function which, from the output tile rectangle coordinates calculates the rectangle coordinates of the input image that the filter requires. The system then allocates a buffer large enough to hold that rectangle and iterates all tiles of the input image that intersect the rectangle. After every tile that is rendered the system copies the respective raster data in the allocated buffer and after all tiles are iterated the buffer contains the whole area the filter requested. After that the filter itself can be executed on the buffer.

This method is simple to implement, easy to use and leads to very straightforward implementations of filters. However, because the whole area, that a filter needs to render one output tile, has to be allocated, this can lead to arbitrarily large buffers for some filters. An example for a larger request would be the filter to save raster data to a file. If used together with buffering the required buffer would have the size of the whole image, defeating the purpose of this system.

Iterating

If a large area needs to be accessed, there is a second mode that can be utilized. Instead of the system iterating the required tiles and storing them in the buffer the filter does provide an iterator. The system iterates the tiles using this iterator and calls a partial processing function on every tile. At the end a final processing function is called that lets the filter finish its work. For example a filter to save the whole processed image to disk, can use this iterating approach to iterate the whole image and directly store every completed tile on disk.

5.4.3 Filter Types

Point

For a point filter every output pixel depends only on one single input pixel with the same coordinates. Such a filter can use the buffering processing mode and if the input tile size is the same as the filters tile size the system can even directly use the previous filters output tiles as input buffer.

Local

A local filter requires for every output pixel a specific area from the input. In most cases this area is not arbitrarily large and so the buffering processing mode can be used with the input area of a tile being the union of all the required areas of all pixels. The area required may be quite large, but as long as it does not depend on image size, the filter will, while being probably quite slow, still stay independent of image resolution.

Global

It is not possible to implement a global filter where every output pixel depends on the all input pixels, as this can not be made scalable. However, global filters can be approximated, either by being only rough approximations to the actual full scale filter, but close enough to fulfill our accuracy requirement, or by approximating the global filter with a fixed-size input. The filter must not depend on the full image size at full resolution, but it can depend on a fixed size scaled version of the image. For example histogram equalization can not be implemented directly, but it is possible to calculate the lookup-table for a fixed, low-scale version of the image, and then apply it to the whole image. This will be only an approximation but might just be good enough for most cases.

Input

Input filters do not themselves consume raster data. As such, it does not matter which processing mode they use as in either case their worker can be called directly. Because of the simpler interface all input filters would therefore use the buffering filter mode.

Output

Output filters export their input from the system, either to a memory buffer, or disk. For displaying it is convenient to use the buffering mode, as the display resolution is itself limited, and the application might only request fixed tile sizes. However for file output the iterating mode has to be used, to avoid the necessity to buffer the whole image before saving it.

5.5 Cache

As all access to image data makes use of tiles, efficient caching of tiles is crucial. We can obviously not cache every access, as this can be an arbitrarily large number of tiles, but we can also not throw away every tile after use, because the probability of reuse is high, especially when using local filters. There are a large number of replacement strategies that could be utilized with a relatively good baseline strategy being random cache replacement mainly for the reason that it is extremely simple to implement in software, and still offers consistent and good performance without any bookkeeping overhead, as we can see in figure 5.5 from our performance measurements. For this reason we chose random replacement as our baseline and compared it with a few other metrics in practical benchmarks, see figure 7.8 for the results. To enable different metrics and strategies to be combined at will, we split the problem in two parts, the replacement metric and the replacement strategy. The replacement metric gives a score for every tile, which is just a measure for the probability of reuse according to that metric. The strategy then decides on the basis of this number which tile to replace. As metrics change during the runtime of the program and we want to keep both bookkeeping overhead down and the implementation simple, we use an approximation scheme to select the tile to replace. Also two or more metrics can be combined to archive a combined evaluation of the reuse probability of a tile. We will define all metrics as higher metric means higher probability of reuse, but without a specific scale, a metric does not necessarily need to be in the range between 0 and 1, but is has to be positive.

5.5.1 Per tile metrics

The following metrics are calculated per tile and are therefore able to order tiles from the same filter.

Least Recently Used (LRU)

Probably the most commonly used metric, tiles are scored by the time they were last accessed, oldest will be discarded first. The reasoning is simple, what is not used for a long time has a higher probability of not being used in the near future and can be discarded.

Least Frequently Used (LFU)

Least frequently used metric. For every tile in the cache LFU counts the number of accesses and uses that number as metric. The argumentation being that frequent requests in the past will hint to more requests in the future. One possible disadvantage compared to other metrics is that a tile can get stuck if it was used frequently, and more recently used tiles might not get requested frequently enough to push the old tile out. This can be overcome by either statistics expiration or by a probabilistic replacement strategy, see below.

Distance

The inverse euclidean distance between the tiles coordinates and the last accessed position is used. As the user pans or zooms around an image a local filter requests the same tile multiple times for different output tiles. This metric should protect tiles nearby to the currently rendered tiles and therefore increase locality of cached tiles.

5.5.2 Per filter metrics

The following metrics use statistics that are obtained per filter and will always score tiles from the same filter exactly the same. This is done because either the statistics do not apply to a single tile alone, or cannot reliably be calculated for a single tile. Therefore it makes sense to combine per-filter metrics with per-tile metrics and obtain a combined metric.

Creation Time

This metric uses the average processing time that was required to render tiles of a filter. While the render time is in no way related to the probability of reuse, tiles that are fast to render do not gain much when cached compared to tiles that take longer, even if the probability of reuse is lower. But as we can approximately calculate this trade off between reuse probability and processing time (see time normalized hit-rate) it makes more sense to use that metric instead.

(Normalized) Hit-Rate

For this metric it is necessary to count the number of cache hits h and the number of cache misses m of requests to every filter. We define the the hit-rate as the fraction of hits per request: $\frac{h}{h+m}$. This gives a measure of the observed reuse of tiles in the past. But the higher the number of tiles cached the higher the probability of reuse, independent of the probability of reuse per filter. Because we want to compare filters independent of the number of tiles that are actually cached for that filter, we will use the normalized hit-rate as hit-rate divided by the number of tiles t cached for this filter: $\frac{h}{(h+m)\cdot t}$. If either $h + m = 0$ or $t = 0$ we will define the metric as infinite, which will result in tiles not being discarded and thus the cache will try to store at least one tile per filter so that statistics can be obtained.

Time normalized Hit-Rate

As mentioned above we can combine these two metrics and take into account both the hit-rate and the time to render a tile. With this we preferably keep tiles from filters which are slow to process and have a high hit-rate implying high reuse probability.

Depth

The Depth metric simply uses the distance from the start of the filter chain to the filter as metric. A tile cached at a later point in the graph will not cause requests from earlier filters, as the tiles inputs don't need to be calculated if it is cached. However this metric comes with a few pitfalls as local filters will cause multiple requests to the same tile earlier in the graph and this metric completely ignores this locality.

5.5.3 Replacement Strategies

Random

Random simply picks one random tile from the set of cached tiles. The metric does not influence the behavior of this strategy. This is the most simple strategy as it needs to collect zero informations over the use of tiles and is such the fastest and most simple to implement. As performance is in general pretty good compared to more complex strategies and performance is also very consistent over all usage patterns, this strategy is a good baseline.

Random Set Replacement (RSR)

When the system needs to find a tile which can be replaced, it is possible to sort all tiles according to the selected metric and then take the one with the lowest probability of reuse. But to avoid the high overhead of sorting or keeping track of which filter metrics might change when some statistics change, RSR describes an approximation scheme to select not the optimal but a good tile. Whenever a tile needs to be replaced, RSR randomly selects a set of N tiles and from this set selects the optimal tile according to the metric. This way the complexity of tile replacement is constant and arbitrary metrics are simple to implement.

Probabilistic Replacement

For some metrics, notably LFU it is possible for a tile to become stuck in the cache. In this case we want the replacement strategy to contain some form of randomization additionally to the random set utilized in RSR, to allow a tile even with high metric to be replaced sometimes. However we want the probability of ignoring the metric to be low, else random replacement could be used directly and the metric would then be completely ignored. Probabilistic replacement provides both requirements by randomly choosing a tile with the probability depending on the metric. E.g. tile A with the metric of n has a halved probability compared to tile B with a metric of $2n$ to be selected for replacement.

5.5.4 Inter-Dependencies

In difference to other caching problems the probabilities of tile requests can be directly related. In a chain of filters, if the tile at a specific position is not available it will cause tiles from the previous filter to be requested. This also means that for optimal caching it is necessary to take this dependency into account. This is a complex task in relation to the metrics presented above, because it is still necessary to allow caching on all layers, as local filters can request the same tile multiple times from different output tiles, but might allow even higher performance.

5.5.5 Variations in use

The cache has to cope with varying usage patterns, panning and zooming around in an image will lead to different request behavior as compared to, for example, fixed window of the image where the settings of a single filter in the middle of the filter chain is changed. As the cache does not currently take this into account, the best strategy at the moment is to chose the cache parameters in a way that work with many scenarios, without optimizing for a specific use case.

5.5.6 Statistics

The per-tile statistics are independent of a perceived global state and do not case any extra maintenance overhead. However logging hits and misses is subject to changes in the usage of the cache. As the metrics explained above can not specifically react to differences in cache usage, the current implementation does not expire cache statistics, but a more intelligent cache would need to observe changes in usage and use or expire the obtained statistics accordingly.

6 Implementation details

In the following we give an insight in our example implementation of the whole architecture. This implementation is intended as a demonstration for the advantages that scale-aware image editing can offer, as well as an example for such an architecture that can be further extended. The implementation is far from implementing every feature that scale-aware image editing can deliver, but it should demonstrate the core features.

6.1 Code

The core of the implementation is provided as a POSIX compatible library written in the c99 standard. The code should run on a broad range of hardware and software configurations and was tested with Linux kernel 2.6.38 to 3.5.3 on hardware starting with a dual-core ARM Cortex A9. The implementations requires only few dependencies, which are a some libraries from the Enlightenment project [Enl], for low level data-management and the user interface, as well as the LittleCMS library [lcm] for color space conversion, the libtiff library [tif] and the IJG JPEG library [jpe], all of which are available under an open-source license and are normally included in most Linux distributions. The implementation goes by the name of liblime for the library (lime for large image editing) and all library functions as well as the applications included are prefixed with `lime`

6.2 Features

The implementation provides a library that implements the core architecture, as well as a few programs to work with the filters and the system, and for benchmark and testing purposes. The implementation provides the following features:

Resolution independency

The performance characteristics of the implementation do not depend on image resolution, response times and memory consumption are completely decoupled from image resolution, and therefore working with a 10 megapixel image requires roughly the same resources as compared to working with gigapixel image sizes.

Fast response times

Interactive editing of extremely large images is possible on hardware as low-powered as an ARM Cortex A9, for all filters but the denoise filter. Interactive means reaction times are well below one second and in most cases reaction is more or less instant. The actual rendering is decoupled from the interface using threading and the interface stays reactive at all times even if individual render times are longer. In that case of slow filters, the application displays a lower resolution version of the filtered image until the higher resolution is available.

Low memory consumption

If provided in the pyramidal TIFF format, the system can work with very large images while maintaining a small memory footprint. Memory consumption depends on the filters, which may allocate small buffers, and on the cache size. Larger cache sizes can lead to higher performance, but are not required for low response times.

Low Complexity

All image operations have a complexity which is linear with respect to screen size. Render times do not depend on the full resolution equivalent of the window that is displayed but only on the actual pixel count to render. The only two exception are the JPEG loader as JPEG does not allow scaled random access, and the TIFF save filter as that always processes the whole image in full resolution.

Implemented filters

The implemented filters that can be applied by the user are contrast, exposure, Gaussian blur, denoise and sharpen. Filters that are not available to the user but are applied by the application are the load, save and “memsink” filter (exports tiles to a application provided buffer). Filters that are automatically inserted by the system consist of the specific file loading filters for JPEG and TIFF files, and the color space conversion filter.

Threading

The rendering interface is thread safe and allows to render multiple tiles in parallel. The `limeview` application uses this property to render with the same number of threads as the CPU has cores.

Cache

The cache addresses tiles by a hash value calculated from a serialization of the filter chain with the respective filter settings. Because tiles are not discarded when settings change but only replaced when the cache limits are reached, the cache does not distinguish between tiles that cannot be requested with the current settings and tiles that could be requested. Thus for example when switching from one image to another and then back, the tiles of the first image might still be cached and can be used, or when changing settings of a very slow filter, multiple settings can be compared very fast, because after changing back the tiles associated with the old setting are still available. This completely eliminates the necessity for an application to implement any caching by itself, as is often used in image viewers to speed up operation when switching between multiple files.

6.3 Limits

The following limits are implementation specific, not inherent limitations of scale invariant image editing itself.

Image size

While the size limitations of the implementation are very generous (maximum width and height: $2^{31} - 1$) the image file format used, tiled pyramidal TIFF, imposes a file size limit of 4 gigabytes. The BigTiff format extension strives to remove this limitation and should be compatible with this implementation.

Bit depth

Bit depth of raster data is limited to 8 bit by the implementation. While the architecture itself allows for different bit depths thanks to autoconfiguration, this would not be usable without filters that support those bit depths and no filter was implemented with that feature. So the foundations are there but a bit of filter implementation effort is still required.

Color space

Color space supported internally consist of sRGB, LAB, YUV and HSV as implemented by LittleCMS [lcm] which is used for color space conversion. This can easily be expanded to every color space that LittleCMS implements if necessary. All operations on RGB color space are properly gamma-corrected as ignoring gamma-correction can lead to bad accuracy (see figure 7.1).

Load and save

Only with the TIFF loader the system actually possesses all the properties detailed. This is not possible for the JPEG loader, but the JPEG loader was implemented anyway, as it gives convenient access to small JPEG files (dozens of Megapixels), larger images must utilize the TIFF loader.

6.4 Data structures

There are two important structures used throughout the implementation an which are also exposed through the API to describe the behavior of filters. The first one is the filter structure itself which provides various callbacks that configure and execute a filter, and the meta type, which is used for the filter specifications trees which define input, output, settings and tune characteristic of a filter.

6.4.1 Filter specification

Filter specifications are implemented using the generic "Meta" structure. This structure forms a tree node which has a specified type, like channel or color space, and a list of data pointers to elements of that type, and a data pointer. The meaning of the list differs a bit depending on where it is used, see section 5.3. Every meta node can have a number of child nodes and can point to a replacement node, for forwarding input child nodes. Nodes also can have a name which is there mostly for informational purposes as functionality only depends on the type and data of a node. If a node depends on a tunable before assuming a value it points to that tunable and can also contain a pointer to a function to calculate the value of the node when the tunable is fixed. The implementation uses this to test all possible tunables to find that value for the tunables that makes pairing of the node with its corresponding node at the connected filter possible.

6.4.2 Filter

The filter struct contains mostly callbacks, like functions for area calculation for a requested output and callbacks for tunable and configuring the filter. A filter does not need to create its worker functions on creation, but can wait until it is fully configured, and has all variables fixed on how the operation will take place, and can then select the appropriate worker function and store it in its filter structure. Every filter defines a structure of type *Filter_Core* which is registered with the system and contains filter name and a constructor to allocate and initialize a new filter structure, which then contains the necessary data for the new filter instance. There are five filter specification trees which specify the characteristics and assumptions of the filter to the system and other filters: The in and out tree specify input and output characteristics respectively, the tunable tree contains the tunables selectable by the system and the settings tree is actually rather a flat list containing the settings a user or application can influence.

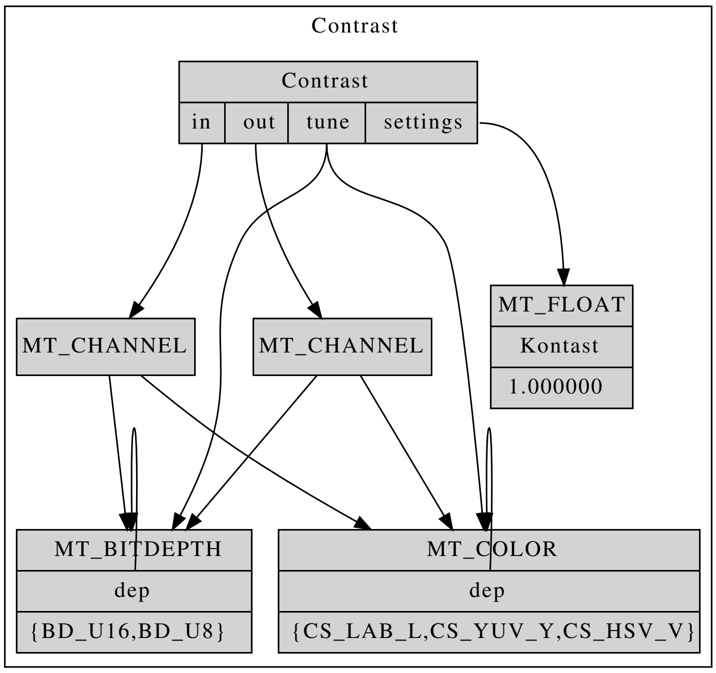


Figure 6.1: The specification trees of the contrast filter. This filter uses only one channel, with bitdepth and color space obviously shared between those channels. The `dep` field points to the tunable, and because the tunable and the nodes should contain the same value at all times they are simply the same node.

The child nodes of a settings node only state the limits of the respective setting. Lastly the core tree contains information that the system can directly used, and is at the moment only used to specify image dimensions so the system can check against them for render requests outside the image border and to provide the dimensions to an application. Note that the trees may share all nodes that have the same child and data at any time, for example input and output color space or bitdepth, which can also point to themselves as their own tunable. See figure 6.1 for an example of this sharing of nodes.

6.5 Image Files

For optimal operation, specifically to stay within the complexity constraints, the system needs access to a pre-scaled image pyramid. The file format of choice is tiled pyramidal TIFF which is commonly utilized in geographic information systems. The Tagged Image File Format (TIFF) is a very flexible file format and tiled pyramidal TIFF simply describes a standard TIFF file which contains, within the same file, multiple images where each consecutive image is the same image scaled by a factor of 0.5 in each dimension. Additionally the recently introduced BigTiff standard lifts the four gigabyte limitation that was part of the older TIFF standard and therefore can work with very large image files (up to 2^{64}Bytes). Because our scale invariant image editing implementation can utilize different tile sizes per filter the implementation of the TIFF loader simply uses the tile size of the TIFF file and the system converts those tiles if necessary. There exist other file formats that allow for scaled and random access, like JPEG2000 but no format enjoys the same spread and the excellent support as does TIFF, when using the LibTiff library ([tif]).

A very important image format is the JPEG image file format which is arguably the most commonly used image file format for general imaging. While JPEG files do not easily allow for random access the open source IJG JPEG library ([jpe]) includes a feature which allows to decompress in a scaled manner up to a scale-down factor of 1:8 while skipping decoding and inverse dct on coefficients that are not needed for the given scale. Because of this feature the implementation contains a JPEG file loader, which avoids random access by using a tile size at each scale equal to the image size at the respective scale. For large images, tile size therefore becomes very large for the highest scale and JPEG files only allow a scale-down of 1:8 because of the dct block size. Therefore performance becomes rather erratic when large JPEG files (above a few dozen megapixel) are used, because decoding a single tile of the JPEG file is very slow compared to the small tile sizes used in tiled pyramidal tiffs, and if one scale is evicted by the cache the system sometimes has to wait for a very long time (seconds) when that scale is needed again. Even so the JPEG loader allows to work with relatively small JPEG files with higher performance compared to for example the Gimp or VIPS even if this is not as fast as with tiled pyramidal tiffs.

6.6 Architecture Implementation

This section gives insight into how the architecture described in section 5 translates into an actual implementation. Specific details that have to be taken care of are the actual rendering, the implementation of threading, the cache and especially the filter specification trees. While our architecture only specifies some properties like the hashed addressing of tiles, this section will detail exactly how this was accomplished in our implementation.

6.6.1 Rendering

To actually render a tile the application requests the rendering of an area for the last filter in the graph, the sink filter. The implementation then uses a stack to keep track on which tile for which filter needs to be rendered or is already completed. Basically every element in the stack points to a filter and an iterator object, which is used to iterate the required input for the filter. The system starts at the sink filter and inserts the sink filter in the stack. The two render modes that a filter can utilize only differ in the iterator used. When using the iterating filter mode the implementation calls the iterator provided by the filter, which returns a single tile coordinate and a channel number, and the system then checks whether this tile is already available in the cache. If it is the tile is passed to the processing function of the iterator and this step is repeated until the iterator signals its completion, at which point the finish callback from the iterator is called, with which the filter can execute concluding work that might be necessary, and after that the stack is popped and the current filter removed.

If a tile is not cached the system pushes a new element to the stack, records the filter and the tile requested from this filter, and starts with the iterator of this new element. If a filter uses the buffering mode instead of the iterating mode, then it does not have to implement the iterator itself but the iterator is provided by the system. The filter only has to provide a callback which calculates the required input area from the tile coordinates, requested by the filter, and the system then allocates a buffer with the size of the input area. The buffering iterator does not call a filter callback when iterating but simply copies the required image data from the rendered tiles into the buffer allocated for the input. Only when this iterator is finished the single worker function of the filter is called which can then process the whole image in one call. The obvious simplification is, that the filter does not need to implement an iterator and can do the processing with one step, but this comes at the cost of additional `memcpy` operations and higher memory consumption.

6.6.2 Threading

Multithreading to increase performance on SMP hardware is implemented using the POSIX Threads API with only one single lock. Only the rendering function of the API is protected by this lock. For all other calls the application has to deal with threading by itself. The reason for this is that if changes are made to the filter graph, concurrent to a rendering call, this would make the current rendering work on an invalid graph and the operations would have to be aborted. Instead the application must wait until all rendering is finished and only after that may commit changes to the graph. Blocking the lock is the first thing that is done in the rendering function and releasing it the last thing before returning. The lock is also released before calling the worker function of a thread safe filter and blocked again after the worker returns. The result is that only the actual filtering is threaded, the work in between like calculating required areas, cache-lookups etc. can only be executed in parallel to the worker function of filters, but not to each other. This works because the tile size is large enough so that even very fast filters use much more processing time compared to the rest of the work performed in other parts of the code. The big advantage is, that the threading implementation is very simple. The basic threading code actually consists of only ten lines of codes. Because a thread may request a tile which is currently rendered by another thread there is additional code that waits in this case and also releases the lock until the other thread is finished. While this could in theory hinder scalability because of the dependencies of threads, in practice we were unable to determine any impact, probably because rendering a single tile is normally very fast and the number of collisions is very low.

Filters have to specify whether they are thread safe or not. A thread safe filter can provide a callback to allocate additional memory, so it can work in parallel using independent buffers, the normal worker functions are always called with a thread number and a filter must use only the buffers associated with this thread number.

6.6.3 Cache

The cache is a very straightforward implementation of the infrastructure as detailed in section 5.5. The cache stores tiles in a single hash-table and all requests address tiles by the hash value of the tile. Noteworthy is the way those hashes are calculated. Every filter has its own hash which is calculated from a serialization of all filters and the settings that come before this filter, up to the filter itself and its settings. This filter hash is recalculated whenever a filter setting or the filter graph is changed, starting from the changed filter in direction of the sink. To calculate the hash of a tile, the implementation uses the filter hash together with the tile coordinates, scale and size, and calculates the tile hash over those four values. The advantage is that the calculation of the tile is performed in a constant time, independent of the size of the filter graph. Because the cache replacement algorithm always examines a fixed number of tiles, the tile for replacement is also determined in constant time, independent of cache size.

To calculate the actual hash our implementation uses a temporary buffer where for every filter the previous hash is printed as a string, followed by all filter properties as strings. This buffer is then simply hashed by a fast string hashing algorithm from the support library Eina from the enlightenment project [Enl]. The choice of hashing should not have a great impact on performance as long as it is fast, because the number of requests relative to the CPU time spent in actual filter work is quite small.

6.6.4 Configuration

Due to the tree organization of the filter specifications, the automatic configuration of the filter graph is a relatively complex endeavor. To keep the implementation as simple as possible, the configuration does not strive to provide the optimal configuration that would give the best performance but satisfies itself with a working configuration. Basically the implementation start at the source filter and works it way from there to the sink. For each pair of filters, it tries to find a combination of tunables between the tunables that can still be changed in the already configured part, and the tunables from the unconfigured filter. If it is possible to find a compatible combination, a new virtual output tree is created from the configured filters as described in section 5.3.3 and the system goes on to the next filter. Only if it is not possible to find compatible combinations of tunables the system starts to insert conversion filters, trying first all single filter, then all two-filter combinations and so forth, until a fixed limit is reached. Because most filters can work in several color spaces this configuration strategy is in most cases not far off from the optimum, at least with respect to the number of filters additionally inserted. An optimal approach would need some kind of performance measure for all filters and their tunables and might use an A* search where a node in the graph represents a filter instance with a fixed selection tunables.

To determine compatibility the implementation first checks whether it can find a node from the respective source filter so that the subtree with that node as root could be compatible the input tree of the sink filter. If this fails the two filters are not compatible, and conversion filters need to be inserted. The “could” stands for the fact that in this first stage no tunables are checked, but for every node the implementation only checks whether the list of values of those nodes have at least one common value. After this first stage for every filter the tunables are checked and for every tunable any node depending on that tunables gets assigned a single value. The system can then restrict the possibilities for each tunable by checking each node and keep throwing out those possible tunable values, that cannot be selected in one of the filters nodes. Because tunables might not be connected the same in both filters it is possible that no compatible combination of tunables exists and then the whole process also fails and the system falls back to inserting conversion filters.

From the filter side there are multiple stages where the configuration interacts with the filter. The first stage is when the system is testing the tunable values. For every node the filter can provide a callback that calculates that nodes value depending on the tunable value. The system assumes that a node value depends only on one tunable, therefore a property that depends on multiple tunables needs to be split into multiple nodes. After the system successfully paired two filters the inputs_fixed() callback of the filter is invoked which allows to calculate values of the output tree which depend on the input tree of the filter. After the whole graph is configured all remaining tunables are assigned a value and the system calls the tunes_fixed() callback that allows the filter to execute whatever is necessary before the actual rendering can start, but what could not be done with some tunes not yet fixed.

6.7 Filter Implementation

In the following we will examine our implementation of the most important filters, which should serve as an example, both of how the specific filters might be implemented and as an example to the procedure of implementing filters for scale invariant image editing, following the approaches presented in section 4.4.

6.7.1 Contrast and Exposure

Those two filters both utilize the buffering interface for simplicity but could as well use the iterating interface. The filters are simple linear point filters, and while clipping can introduce large differences between the full scale and the scaled filter, these differences only affect scaled pixels that contain clipped and unclipped pixels which are mostly borders of completely clipped areas or single pixel detail. In both cases the approximation reacts positively to supersampling as no large areas are affected. Multiple color-spaces are selectable by the implementation via the tunables, all with some kind of “lightness” channels which are LAB, YUV and HSV. The code for each filter amounts to less than a hundred lines with most of the actual code accounting for the filter specification trees. To allow threading no extra work had to be done as the filters do not use a buffer and so no conflicts arise.

6.7.2 Color space conversion

The color space conversion filter uses the LittleCMS library [lcm] to calculate the actual conversion. The implementation only works with the sRGB, LAB, YUV and HSV color spaces, but can easily be expanded, to cover more of the color spaces that LittleCMS provides, if necessary. LittleCMS compiles the selected conversion, to speed up operations, and this is feature is integrated within the inputs_fixed() callback prior to the actual processing. The same conversion is used by multiple threads, but the filter has to use buffers because LittleCMS uses one large buffer for the planar configuration whereas our implementation provides three independent buffers. Therefore the filter has to copy around the image data using an intermediate buffer, and for threading this buffer is allocated extra for each thread. Because most color space conversions are non-linear the filter does not have perfect accuracy, but because color-space conversion is also obviously not a completely random look up table, the filter responds well to super-sampling.

6.7.3 Save TIFF and compare filter

The purpose of the save TIFF filter is to save the whole processed image of a filter chain into the file-system in the tiled pyramidal TIFF format, suitable for the use as an input image for scale invariant image editing. The comparison filter on the other hand is a filter which compares the result of a filter applied to the full resolution filter which was scaled after filtering, to the directly applied filter at that scale. Both have to process the whole image and calculate all scales themselves, for they need the non-approximated reference image to save it to the file and to compare it to the filtered result respectively.

It is obvious that those two filters need to utilize the iterating filter mode, because a temporary buffer with the size of the whole image is not feasible. What is less obvious and what sets our implementation apart from others is the order in which image tiles are processed. The obvious solution to limit memory usage is an approach with two passes, a first pass iterates all image tiles and saves the respective scaled versions to one or multiple temporary files, in the order they are created. This approach is utilized for example in the pyramidal TIFF writer used in the VIPS image processing system. Our solution utilizes a single pass over the image while maintaining a space complexity in $\mathcal{O}(\log(n))$ where n is the full size of the image. To achieve this we traverse the whole image in a pattern based on the Lebesgue curve. The first four tiles are iterated in a "Z" pattern. A buffer, the size of a single tile, but with higher bitdepth (32bit) accumulates the four tiles of the pattern, summing up four pixels to result in one pixel in the buffer. The original tiles are directly save in the TIFF file and from the accumulation buffer we calculate the first scaled-down tile, which is saved to a second directory in the same TIFF file. Images are made up of so called Image File Directories in the TIFF file format, each directories points to an additional image inside the same file. After the first four tiles for the original scale and the first scaled tile is saved to disk, the "Z" pattern is repeated for the next four tiles, where every "Z" forms the corner of a larger "Z". Whenever a tile of scale 1 (where 0 is original resolution and each increase by 1 denotes an additional scale factor of 0.5) is completed, it is accumulated in yet another buffer which accumulates the next higher scale. Because of the order in which tiles are iterated, the system never touches a second tile at any given scale, as long as the first tile is not completely accumulated. This results in a space complexity of $\mathcal{O}(\log(n))$ because we only need a single buffer for every scale, and each additional buffer increases the possible image size by four.

The TIFF file written can be directly used after this single pass through the whole image, because the libtiff library allows to rewrite directories, which only copies the required directory header, but not the actual image tiles. The resulting file is a bit larger than necessary because of dead directories headers, but if necessary the file can be rewritten with a single extra pass using the tiffcp utility.

The compare filter uses the same strategy but instead of saving a completed tile to a file, requests the corresponding scaled tile from the system, so the scaled and the full resolution result can be compared.

Both filters demonstrate the possibility of using different worker functions depending on configuration, to provide a linear and a gamma-corrected accumulation depending on color-space. Also these two filters allow to specify the color space, because for compare and file saving the color space critically influences the result, where other filters can automatically use the color-space of the input filter.

6.7.4 Gaussian Blur

Iterating a box filter is a well known approximation for a Gaussian convolution, which is very simple to implement and possesses a runtime independent of standard deviation and linear with respect to the image size [Wel86]. The problem is that the usage of a simple box filters introduces a quantization to the range of standard deviations because the width of the box filter can only assume integer values. The extended box filter builds upon the normal box filter, by introducing a non-integer width calculated with an additional fractional weight of one pixel at every side of the box filter's support window, and can therefore approximate arbitrary standard deviations [GGBW12].

With the iterated extended box filter we can approximate a Gaussian convolution, but what still has to be solved is the scaling of such a filter. In the following we will illustrate the steps and the reasoning undertaken to implement the scaled Gaussian filter, as an example for a possible approach to the implementation of scaled filters. The basic approach was to approximate the full scale result at lower scales, as described in section 4.4.1.

First, the full scale filter was implemented using the iterated extended box filter with the formula of Gwosdek et al. from [GGBW12] to calculate the width of the box filter from the standard deviation. The filter is controlled only over the standard deviation and avoids the use of division by using a pre-calculated integer multiplication together with a power of two division. To implement scale invariance we started by simply scaling the width of the box filter by the scale required. For example at half the full resolution we would simply use the half width for the extended box filter.

When testing this implementation with the comparison filter it was immediately obvious that our approach to the scaling resulted in a image that was too blurry compared to the correctly scaled filter. The reason for this becomes obvious when we examine what the scaling means for the a single pixel. Pixel size effectively increases when we scale down, and we have additional blur compared to the full resolution because one pixel can assume only a single value, which

Scale	width times 16, difference in LAB						
1	width	72	73	74	75	76	77
	difference	5	4	2	2	2	3
2	width	32	33	34	35	36	37
	difference	8	4	2	4	8	16
3	width	12	13	14	15	16	17
	difference	9	9	9	8	8	11
4	width	0	1	2	3	4	5
	difference	73	23	12	23	32	39

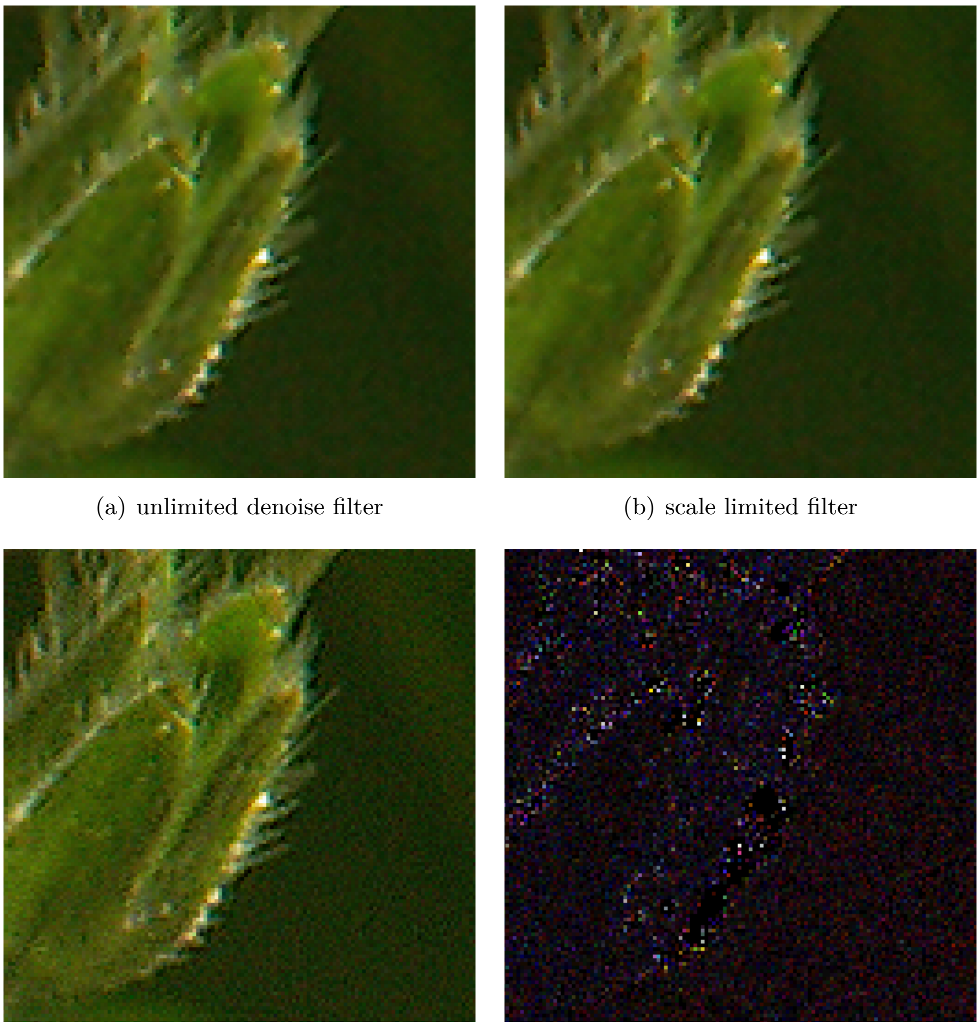
Table 6.1: Difference between a scaled blur ($\sigma = 5$) with the specified width, and the full scale filter scaled down accordingly. Note that the actual width is multiplied by 16 because the fractional part of the extended box filter is realized using fixed point arithmetic, for performance reasons.

results in an additional blur because, in full resolution terms, an additional averaging step is executed over all pixels that make up a scaled pixel. The further we scale down the larger a pixel becomes, compared to the highest resolution, and the larger is the additional blur because of the larger pixels. To test our theory we tested a number of widths of the scaled filter, and after every pass examined the result, which resulted in table 6.1.

For reference, straight down scaling would have given us widths of 80, 40, 20 and 10. From this observation we assume the formula for the scaled width of $w_s = \max(\frac{w_0}{2^s} - 2 \cdot s, 0)$. This formula is just a guess and it might be possible do derive a better approximation by a more detailed investigation of the effect the scaling has on the Gaussian convolution. But this gives sufficient accuracy also for other standard deviations, as can be seen in the accuracy analysis of our Gaussian blur implementation in appendix A.1.

6.7.5 Denoise

The denoise filter is a very basic implementation of a Non-Local-Means filter [BCM05], based on the PatchMatch algorithm to find corresponding patches. Non-local-means denoises images by averaging for every pixel a number of center pixels from patches in the image whose neighborhood is similar. PatchMatch is a randomized correspondence algorithm that detects corresponding patches by sampling at random offsets and propagating good matches to neighboring pixels [BSGF10]. Our implementation is very limited in that it uses a fixed patch size of 3x3, a fixed upper bound on the number of patches to average and a fixed maximum distance from where patches might be matched. The implementation utilizes three steps: *Random sampling* which tries a few random offsets, *propagation* which tries samples from neighboring pixels and *enrichment*, which propagates not the offset from a neighboring match but propagates one match from the matches of the neighboring pixels. See [BSGF10] for more details on this approach.



(a) unlimited denoise filter

(b) scale limited filter

(c) original noisy image

(d) difference between limited and unlimited filter (amplified by a factor of 10)

Figure 6.2: The images above show the result of the noise filter, without any restrictions in figure 6.2(a) and limited to a maximum difference between the scaled denoised and the original image of 20 (absolute per channel) in 6.2(b). While there certainly is a difference it is not very large, as can be seen in the difference image in 6.2(d).

The two things that can be tuned for this filter are the number of passes which are executed, where more passes giving better results due to the discovery of more matches, and the maximum difference a block may have from the respective reference block before it is excluded from the averaging process. The difference is calculated as the sum of absolute differences between two patches and summed up over all channels. The maximum difference can be selected independent for color and lightness channels and the filter works only in the LAB color space.

Achieving scaling is simple for this implementation. Most denoising filters should be local average preserving, as described in section 4.3.4, for the reason that most noise types result from probabilistic processes and the rule of large numbers leads to reduced variance if we average over more pixels. Our implementation utilizes a fixed and very small block size and is therefore only usable for relatively light denoising. Therefore we apply the most simple version of a coarse-to-fine approach (see section 4.4.2) and simply define our filter to produce an output that is, when scaled down one time, equal to the input. Of course the implementation above does not ensure this in any way and therefore we simply add a final pass after the actual filtering which limits the changes the filter has made to be within a fixed range of the input when scaled down. This somewhat decreases the efficiency of the denoising, but still leads to acceptable results and obviously good accuracy. For an example of the results and the impact of limiting the filter effect see figure 6.2.

6.8 Library API

The API that can be utilized by an application to make use of the library is relatively simple and straightforward. The following sections will give a brief overview of the usage and the reasoning to this API. See listing 1 for a very small code example which mainly showcases the API for handling the filter graph. The shown c source code can be compiled on supported systems where our implementation was installed, using the gcc compiler:

```
gcc tiffblur.c -o tiffblur ‘pkg-config --cflags --libs lime‘
```

For a more extensive example which also shows scaled rendering, see appendix A.3.

6.8.1 General Notes

Aside from the rendering calls, the API is not thread safe, applications have to take care to call functions only from one thread at the same time. Also, to benefit from threaded rendering, the application needs to render multiple tiles in parallel via the thread-safe rendering function. The following sections lay down the parts that make up the application API, starting with the initialization.

6.8.2 Initialization

The first thing an application has to do is to initialize the library with with `lime_init()`. Before exiting the application `lime_shutdown()` hat to be called to clean up the cache and other working data.

Listing 1 This small example program takes a input TIFF file, applies a Gaussian blur with a user defined σ and stores the result in a second TIFF file. This example only shows how filter settings can be handled and how the filter graph can be constructed.

```
#include "Lime.h"

int main(int argc, char **argv)
{
  Filter *load, *blur, *save;

  if (argc != 4) {
    printf("usage: infile sigma outfile");
    return EXIT_FAILURE;
  }

  lime_init();

  load = lime_filter_new("load");
  lime_setting_string_set(load, "filename", argv[1]);

  blur = lime_filter_new("gauss");
  lime_setting_float_set(blur, "sigma", atof(argv[2]));
  lime_filter_connect(load, blur);

  save = lime_filter_new("savetiff");
  lime_setting_string_set(save, "filename", argv[3]);
  lime_filter_connect(blur, save);

  lime_render(save);

  lime_shutdown();

  return EXIT_SUCCESS;
}
```

6.8.3 Cache

If desired cache parameters can be changed with `lime_cache_set()` this must happen after initialization can also left out to use default cache parameters or changed at a later point in the program.

6.8.4 Graph Handling

The two important functions for actually handling the filter graph are the `lime_filter_new()` function which return a pointer to a new filter structure, that can then be used by `lime_filter_connect()` to connect two filters. To remove a filter from the filtergraph the filter before and behind it can simply be connected directly via `lime_filter_connect()` which will override the previous connections.

6.8.5 Settings

Applications can use the settings specification tree of a filter to see available settings. Settings nodes might have sub nodes with a name of `PARENT_SETTING_MIN` or `PARENT_SETTING_MAX` to specify the limits to which the settings value must adhere. Settings must be changed only with the `lime_setting_`*`TYPE`*`_set()` functions depending on the type of the setting node, as the implementation has to recalculate the hash value identifying tiles from a filter, and mark the filter chain for reconfiguration.

6.8.6 Rendering

There is no direct way to request image data from a filter graph because the graph is closed in the sense, that there is no connection remaining where channels are not connected to the next filter. Input filters do not have channels as input and output filters do not have any output in the sense used by the filter graph, but from the view of the system only consume image data. In listing 1 we have an example where the application does not actually need the image data and just starts the rendering, the filter then save the raster data to a tiff file. When raster data is actually needed by the application for example for display, the memory sink filter has to be used, which takes an application supplied buffer and copies its input image data to this buffer.

6.8.7 Configuration

The library keeps track of the filter graph and of the settings that are changed and automatically configures the filter graph on rendering if this is necessary. But configuration can fail completely when specific settings are changed, for example a non-image file is selected for the load file filter. For those cases the application can check with `test_filter_config()` whether the graph can be configured at all.

6.9 Graphical user interface

The graphical user interface is implemented using the Elementary GUI toolkit from the Enlightenment project [Enl]. This toolkit has very low hardware requirements while at the same time being simple to customize. The toolkit transparently utilizes hardware acceleration if available and is highly portable. The toolkit itself uses on-demand rendering, which is well suited to the task at hand, because this vastly simplifies the display of the image pyramid.

The GUI, as depicted in figure 6.3, is a straightforward interface to the library and allows to manipulate the filter graph and to change the settings of filters. When starting the application, a file or directory has to be passed, else the working directory is used. In the case of a directory it is possible to switch from one image to the next, keeping the same filter graph and without invalidating the cache.

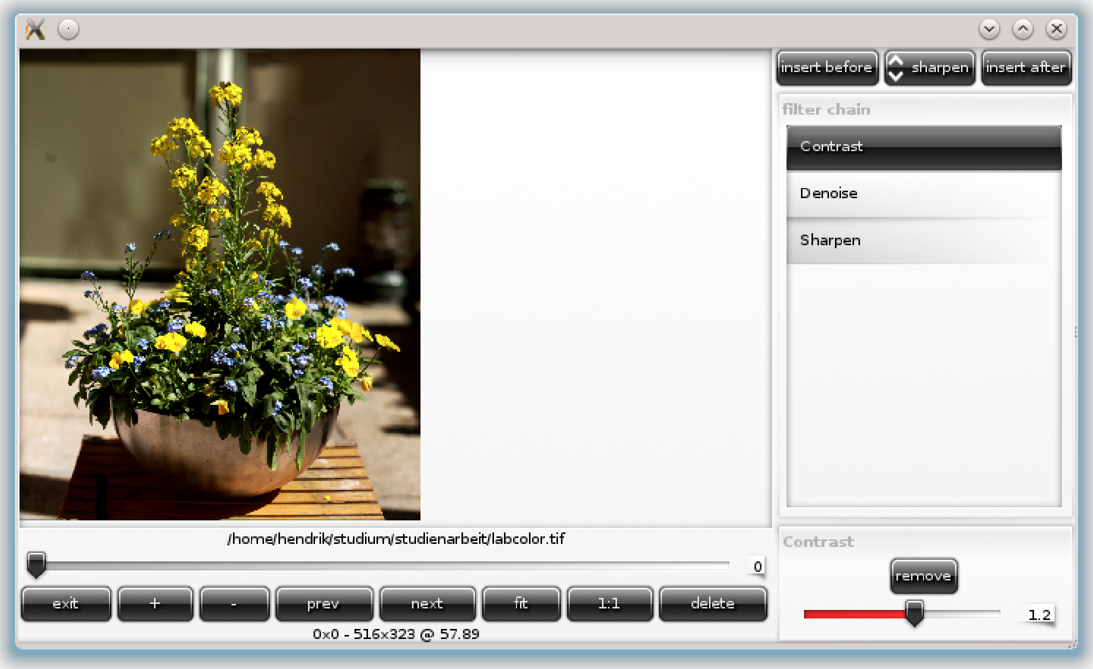


Figure 6.3: Example of the user interface with the sharpen, denoise and contrast filter. The image visible in the application is our example image with 300 megapixels, this resolution is apparent from the small text label at the bottom part of the application. the number right to the @ gives the scale of the visible viewport.

The most noteworthy part of this implementation is the complete decoupling from rendering and GUI and the utilization of the image pyramid available through the scale invariant image editing system to provide fast response times. The application maintains itself a pyramid from the rendered tiles. As the user moves around parts of the pyramid become visible that were not rendered before. The application then starts to render a very low scale of the pyramid, independent of the scale that is actually necessary for display at that moment. All rendered tiles are displayed immediately, scaled up to the required scale for display. Whenever a scale is completed, the next higher resolution scale is rendered. Because rendering lower resolutions is much faster as compared to the full resolution currently required, the application is able to provide a low scale preview very fast and can then render the full resolution, resulting in excellent response times. All rendering takes place in extra threads which leads to a completely non-blocking interface.

7 Results

Below we will examine various measurements which expose the performance characteristics of the implementation, as well as the trade-offs that were made concerning accuracy and performance. Where not stated otherwise, the tests were executed on a computer equipped with a dual core AMD Athlon64 X2 5200+ running at 2.60GHz and with 2 gigabytes of RAM. The benchmark mode of our implementation uses the buffer engine of the elementary toolkit which is a pure software implementation. This allows to use the same window size on any system without influences to the benchmark and also allows the execution on headless systems for benchmark purposes. As most of the execution time is spend in the actual execution of filters, headless operation does not interfere with the benchmark result in any way. For all test a 298 megapixel (15943 x 18700) TIFF image was used, stored in the tiled pyramidal TIFF format with a tile size of 256x256 and deflate compression, which weights in at around 560 megabytes. Depending on the test either a sRGB or a LAB converted version of the same image was used. The screen size utilized for the tests was 1024x1024 (1 megapixel). The scripts to generate the data and the figures in this section are also part of the software distribution.

7.1 Accuracy

7.1.1 Accuracy measure

Accuracy was measured by utilizing the comparison filter which calculates the reference image by filtering at full resolution and compares this result with the result achieved by directly filtering at every scale. As stated in section 4.1.2 the goal of our definition of accuracy was to enable supersampling to diminish approximation errors. Therefore to measure the effective accuracy in terms of effectivity of supersampling, we executed the same filter directly at the scale that was compared and with one or two scale down steps between rendering and comparison.

The difference between reference was measured using a simple maximum difference metric, which is simply the maximum difference over the rendered test image and over each color channel between the reference full scale rendering and the scaled rendering. The value this gives us is the worst case experienced over the whole test image, actual differences are much smaller on average. We do not use a regular quality metric like PSNR (Peak Signal to Noise Ratio) as this would give us an estimate of the overall quality of the approximation, but we want to specifically examine the worst case.

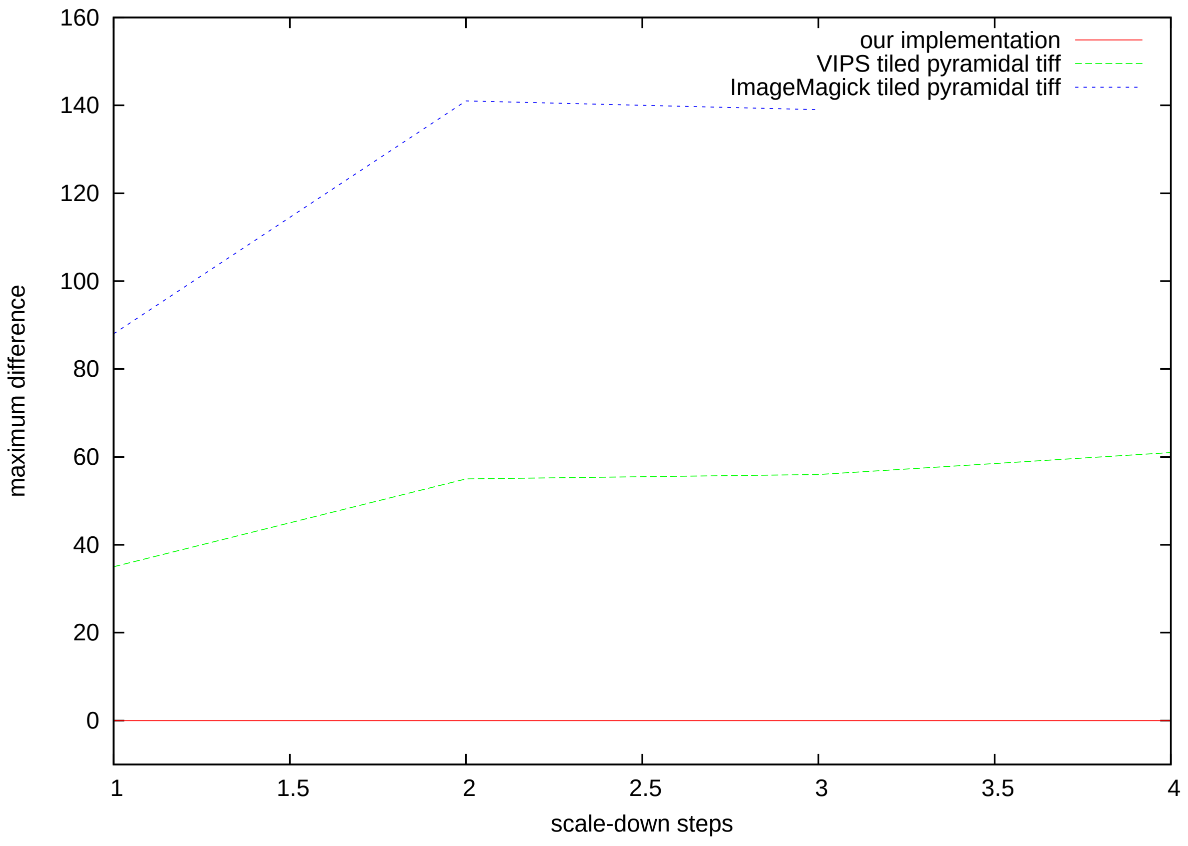


Figure 7.1: Effects of gamma correction on accuracy. Both ImageMagick's convert and the TIFF file saver in VIPS produce pyramids that ignore gamma-correction for the sRGB color space. Note that the image tested with ImageMagick is a lower resolution version as `convert` failed to process the original 300 megapixel image within a day.

All filters were tested with the source image and the compare filter in the color space of the filter, else the required color space conversion would conceal the actual filter performance, as color space conversion introduces errors by itself.

7.1.2 Image pyramids

A surprising result is that of the test performed in figure 7.1. We used ImageMagick's convert and VIPS to produce a tiled pyramidal TIFF file in sRGB color space. Both applications completely ignore gamma correction when calculating the scaled layers, which results in bad accuracy even before any image processing taking place. But the implemented filters assume correctly scaled input, as it is not possible to correct this problem after scaling, and the difference this causes is easily larger than that caused by the implemented filters. While the problem is not as large as to be unusable, it is avoidable and the fact that our implementation requires around 5 minutes to convert our 300 megapixel test image, while VIPS needs 12

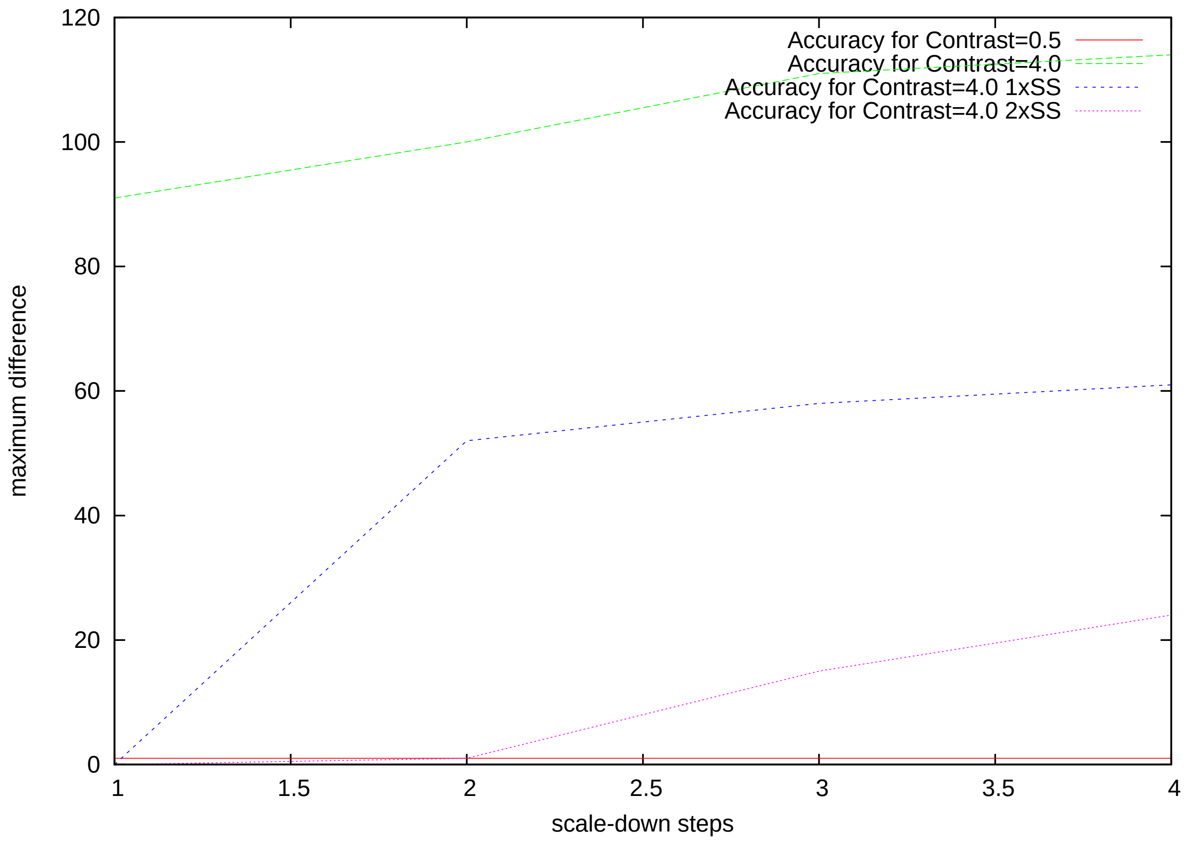


Figure 7.2: Due to clipping the contrast filter obtains bad accuracy if contrast is stretched. A value of 0.5 for the filter does not result in clipping and therefore obtains near perfect accuracy. As we can see, the filter reacts well to supersampling.

minutes and convert needs at least 30 hours (the conversion was aborted after 30 hours, tested was only a smaller image) leads to the conclusion that it is better to avoid those tools to create tiled pyramidal TIFF files, if that is possible.

7.1.3 Filters

While all filters that actually process the image were tested for accuracy, we will only examine the color space conversion and the contrast filter in detail. For the results of the remaining filters see appendix A.1. Note that the result for the contrast filter was the worst of all tested filters, which is the reason why the result for the contrast filter is presented as an example for the worst case of all evaluated filters.

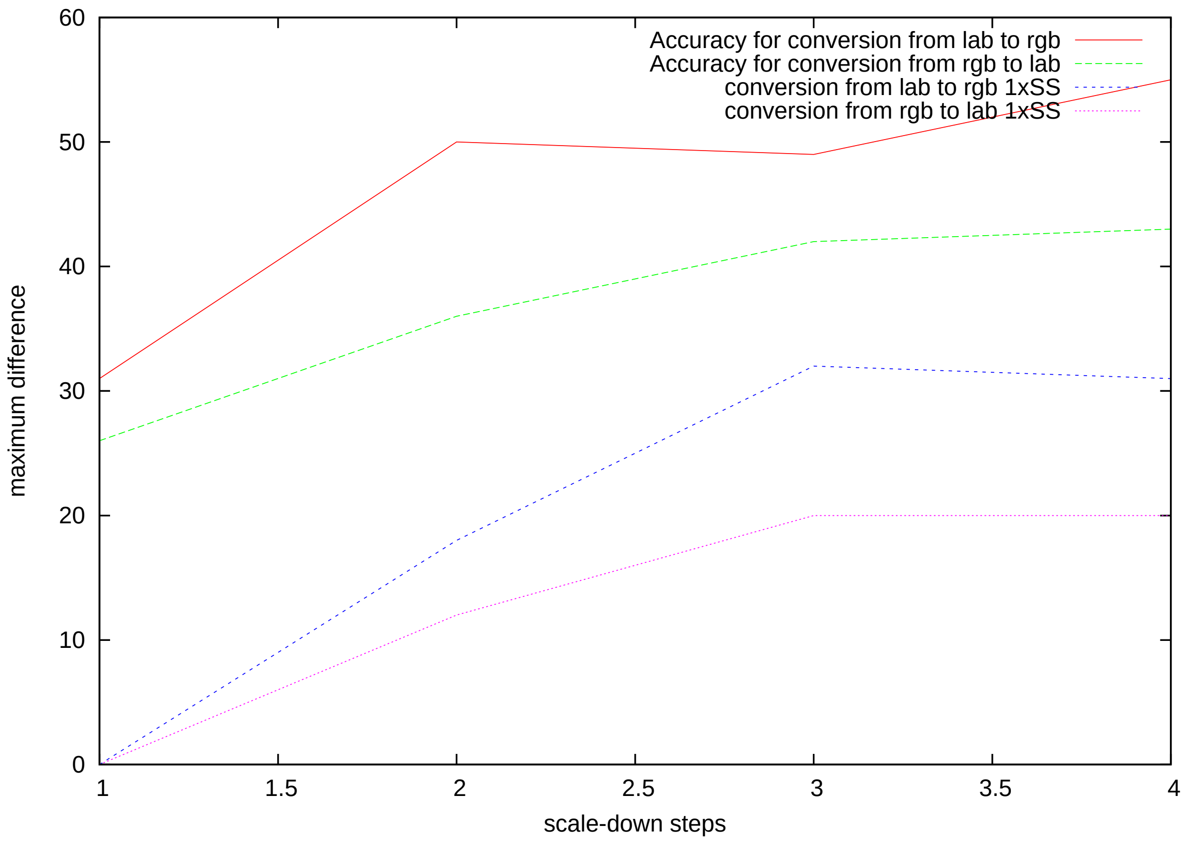


Figure 7.3: Simple color space conversion already leads to differences between scaled and unscaled filtering, but reacts well to supersampling.

Contrast

The approximation errors of the contrast filter, shown in figure 7.2 are caused solely by clipping. This is obvious when looking at the result for a contrast value of 0.5 which results in near perfect accuracy. Supersampling works well to remove those problems, because the problem exists only on scaled pixels that cover both clipped and unclipped pixels at the full scale. Supersampling can calculate exact values for those sub pixels that themselves cover only clipped or only unclipped pixels at the full scale and therefore allows for a better approximation.

Color space conversion

Color space conversion is automatically used by the system if required, without user intervention, therefore it is important that this conversion does not introduce large errors. As we can see in figure 7.3, the results are better than for the contrast filter and also respond well to supersampling.

7.1.4 Impact

The results for the accuracy tests show how our implementation trades accuracy for the possibility of scaled filtering. Note, that all filters respond well to supersampling, which means higher quality results can be achieved by trading performance if that is necessary. While the difference is well notable for many filters when looking at the filter results magnified after rendering, the differences are limited to the pixel level. In reality single pixels of the output are not magnified, because when zooming in, a higher resolution is rendered and the difference stays always at pixel level for the given display. Increasing display resolution, as seems to be the trend with recent hardware, therefore also helps to hide the differences as pixels become so small they are not visible individually.

7.2 Performance

Different performance measurements were conducted, testing the absolute performance of the system using the scenarios featured in section 2.2, as well as the relative performance of filters at different scales. Tested was also the impact of different cache configurations on performance and the maximum delays from (simulated) user interaction was measured for all scenarios. The following sections will be sorted by what parameter was tested for its influence on the performance.

7.2.1 Scale

The single most important benchmark tests whether the promise of scale invariant image editing, the independence of filter rendering for display, from the scale holds true. Therefore different filters were tested at multiple scales, with the result shown in figure 7.4. It is obvious that the promise holds true and no filter is slowed down by scaled rendering. Actually all filters gain a bit of performance, in part because fixed size neighborhoods become smaller as scale down is increased, which is very visible for the Gaussian blur filter. Smaller performance gains can also be attributed to the fact that a lower number of low resolution steps is rendered prior to the requested resolution, because the input image has a limited number of pyramid layers. This effect is probably responsible for the characteristics of the contrast and exposure filter. A special case is the denoise filter. This filter is very slow, but due to the coarse to fine approach used for its implementation the lower scales simply pass through the input image. Therefore this filter is the slowest at full resolution, but the fastest thereafter.

7.2.2 Scenarios

Tested were all four scenarios from section 2.2, using cache sizes between 10 and 1000 megabytes. The first three scenarios are implemented using a Gaussian filter, the *Redo* scenario uses the sharpen filter, followed by the exposure filter, varying the strength of the sharpen filter. Measured was not only the total CPU time of the scenarios but the actual delays experienced

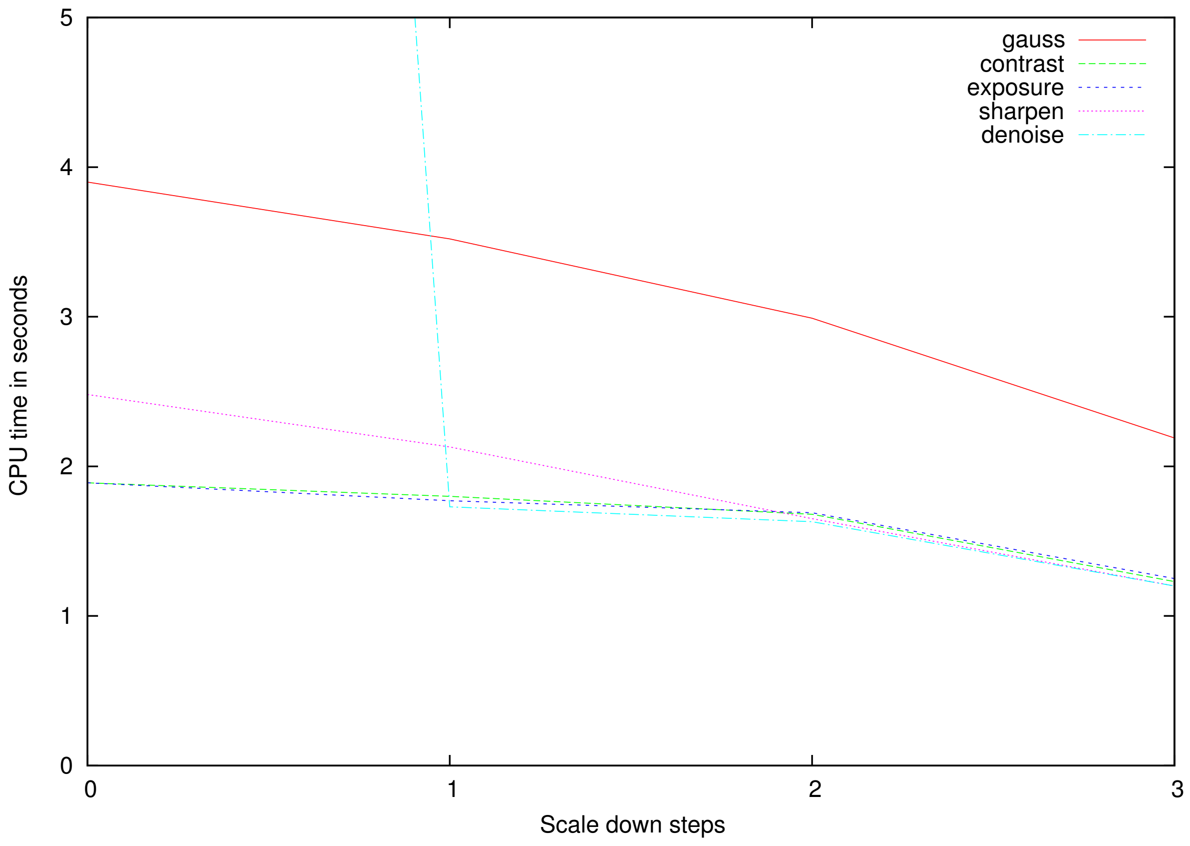


Figure 7.4: Multiple filters tested for their performance regarding scaled rendering. Filters were tested with their respective native color space to avoid the influence of an additional conversion filter. The cache size was set to 4 gigabytes for this test, but was never fully exploited.

from the simulated user interaction like the change of settings, the modification of the filter graph or the movement of the viewport. The delays are measured as scaled delay and as full scale delay. The scaled delay denotes the time until the implementation displayed a scaled preview, the full scale delay is the delay until the system rendered the whole viewport at the full resolution of the window.

Figure 7.5 to 7.7 show the measured results. We can see that the CPU time benefits from larger cache sizes, but for the first three scenarios a size of 50 megabytes already allows for nearly peak performance. Only the forth scenario benefits from a cache size over 100 megabytes which is not surprising as it involves the heaviest interaction case with two filters, where the first is changed at multiple locations in the image. The maximum delay does not depend on cache size as can be seen in figure 7.6 and 7.7. The reason is that the maximum delay probably takes place when a minimum of the required tiles are cached for the used graph configuration, which is the case when adding a new filter or changing filter settings to a value not set earlier. As no tiles are rendered anticipatory, larger cache sizes can not benefit the reaction times of the system.

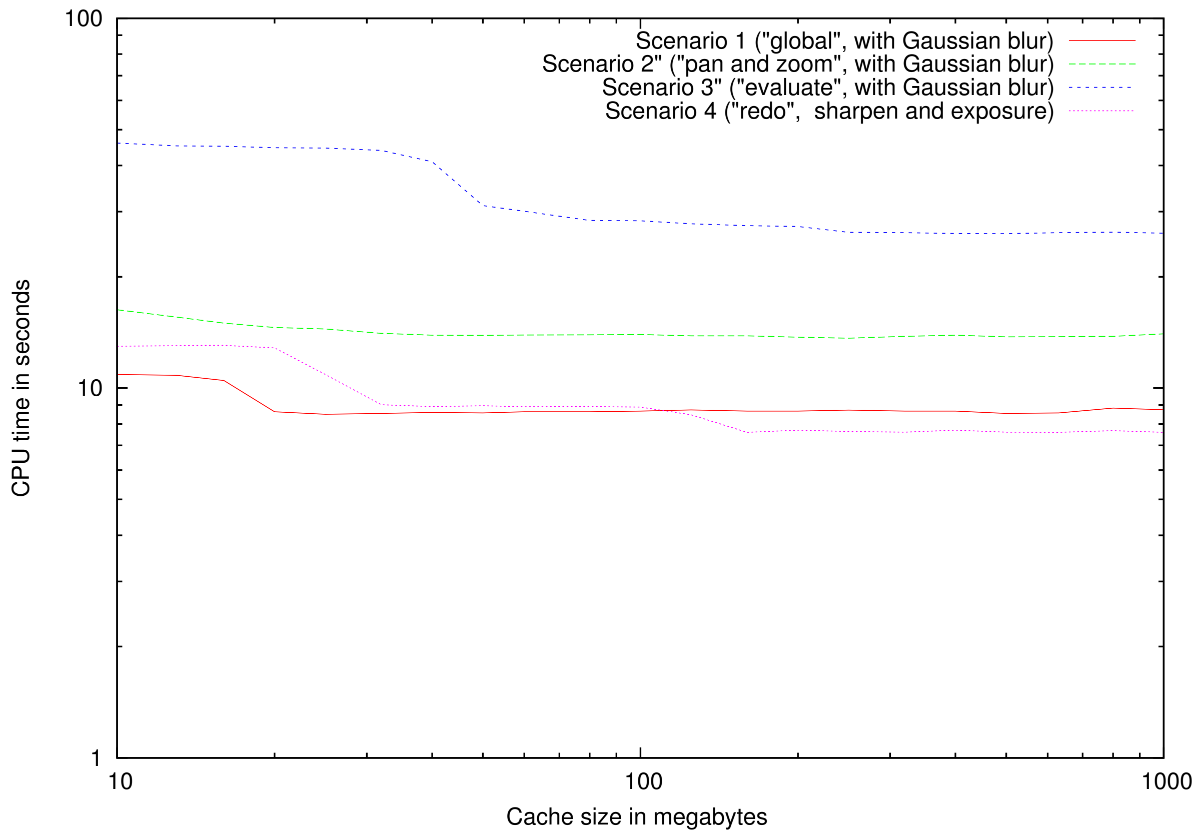


Figure 7.5: CPU times for different scenarios, plotted against cache size. Note the logarithmic scale on both axis.

Note that the maximum delays experienced can not be translated to a frame rate, as the frame rate possible for panning and zooming is much higher because moving the viewport does not directly require tiles to be rendered. The delays state the maximum time the system needs to react to user interaction, mainly when settings or the filter graph was changed. As we can see the maximum delay for the scaled response stays uniformly under 200 milliseconds, while the delay until the full resolution was rendered for the respective viewport goes up to over one second. Because the user can continue to work even without the full resolution rendering, this does not impact the perceived reactiveness of the system.

For a small comparison, the attempt was made to simulate the *global* benchmark with the Gimp. But just opening the 300 megapixel test image takes over 30 seconds, while our implementation is able to render the image with 8 distinct values for the Gaussian filter strength in around 10 seconds.

7.2.3 Cache metrics

The implementation contains four of the cache metrics from section 5.5, two tile metrics (the LRU (Least Recently Used) and the distance metric), and two filter metrics (the normalized

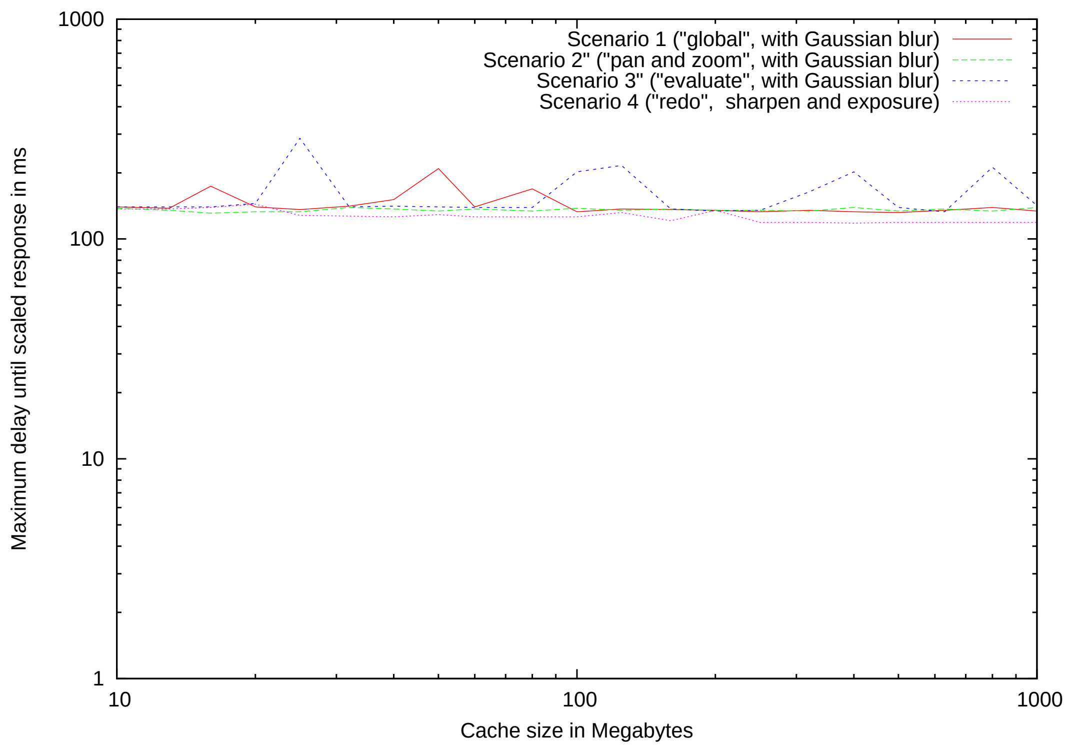


Figure 7.6: Scaled delay for different scenarios, plotted against cache size.

hit rate and the creation time metric). Also tested were all possible combinations from those metrics. Scenario 4 was selected for this test because from the comparison of different scenarios in figure 7.5, it is apparent that this scenario benefits the most from a larger cache, where the other scenarios have already reached their respective performance peak. For cache strategies, *random* was included as a baseline, the metrics were only tested with the random approximation strategy, the probabilistic strategy was left out because the LFU (Least Frequently Used) metric was not implemented and therefore there was no danger that tiles could get stuck in the cache indefinitely.

Figure 7.8 shows the result of these tests, but only contains a selection of all possible metric combinations. Left out were all combined metrics that were worse than any other metric for all data points, for the full data set see appendix A.2. The result is rather surprising, there is no metric or combination of metrics that is better than all other metrics. Rather all metrics seem to excel either at small or at large cache sizes, but not at both. The metric that is worst for the smallest tested cache size of 10 megabytes is actually the best between 50 and 100 megabytes and after that tied with the rest. On the other hand one of the best metrics for small cache sizes, a combination of LRU with the time and the normalized hit rate, trails

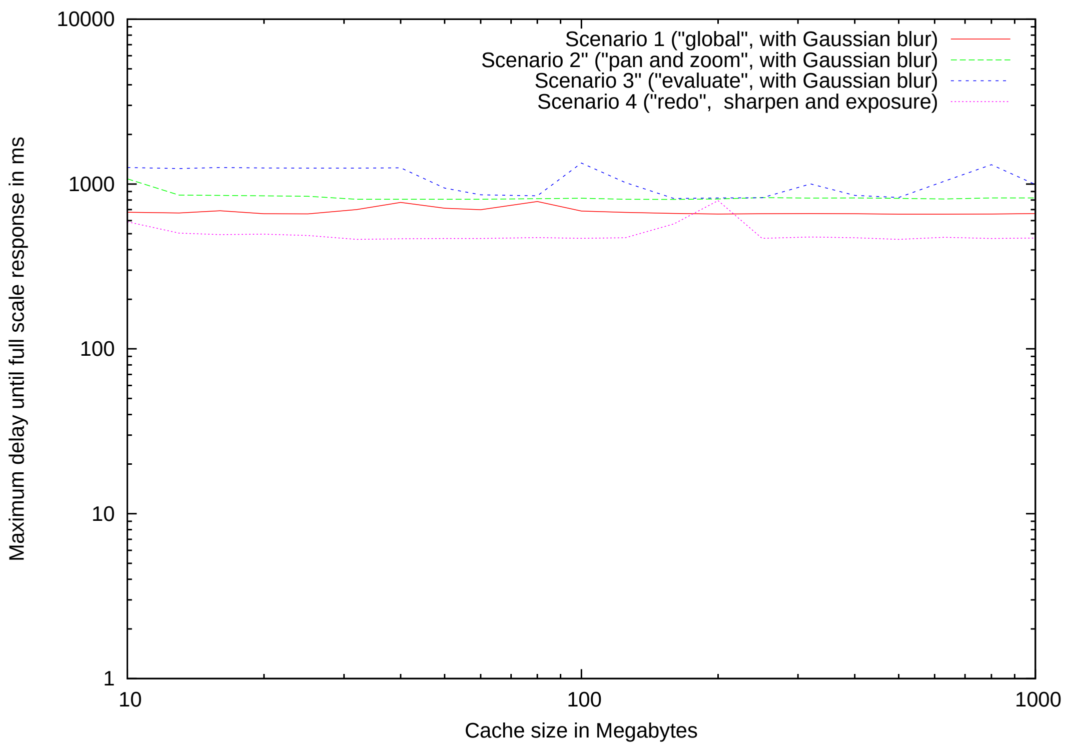


Figure 7.7: Full scale delay for different scenarios, plotted against cache size.

behind between 20 and 100 megabytes and is then also tied with the best metrics. Now this is only one scenario with a fixed filter selection, but more advanced cache metrics as proposed in 5.5 might further increase performance compared to what is visible in this test.

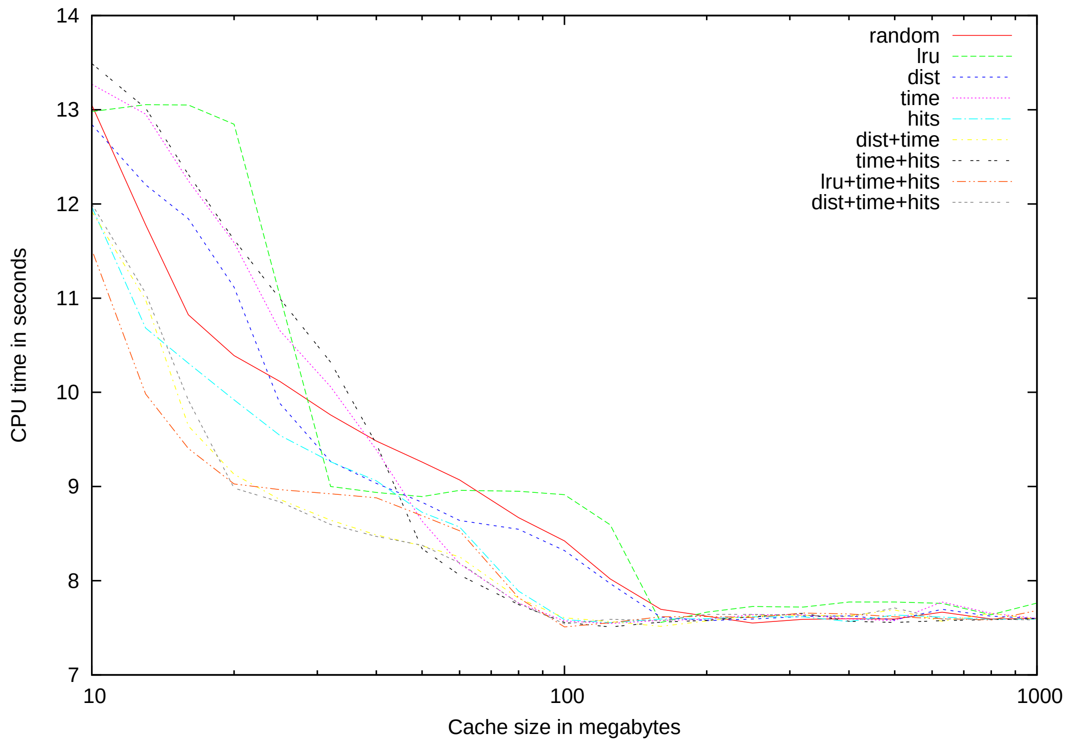


Figure 7.8: Multiple cache metrics for scenario 4: Redo, with sharpen and exposure filter, plotted against cache size.

8 Outlook

Scale invariant image editing shows many desirable properties for interactive image editing. But although this work tries to cover most aspects of an invariant image editing system there are many areas which should be further improved.

Cache

The cache plays a very important role in our architecture, but all metrics introduced so far are relatively ignorant in that they do not exploit the relation between the filters. A more intelligent cache should make decisions based on the fact that a tile that is cached at a later position in the graph can prevent earlier tiles to be requested at all. For example for a point operation it is not necessary to hold the input tile in cache if the output tile is cached, at least as long as settings are not changed. When settings are changed, the tiles of the filter directly before the filter which was changed, should be cached with a higher priority because they will be requested again and again as the filter's settings are changed. Also statistics could be collected about which regions of an image are accessed with what frequency at what scale, to allow patterns in the user interaction to be exploited.

Also the cache currently exists only in main memory, the larger reserves of space available on persistent storage could be utilized to allow a much larger cache that should still be fast enough for interactive display.

Anticipatory rendering

To further decrease delays when rendering, the system could utilize an anticipatory rendering scheme similar to [PV02], which tries to predict areas that might be requested in the future and render those in advance. Together with a disk cache this would result in a system that becomes completely lag-free, if given the time to render the whole image in background, similar to conventional image editing after a filter was completely applied.

Exploiting the image pyramid for filtering

An implementation for scale invariant image editing has to provide an infrastructure to access multiple scales. Filters could exploit this infrastructure to increase performance, for example the Gaussian blur filter could work on a lower scale for larger filter widths and then scale the result up for display. Or the denoise filter could start at a lower scale to collect candidate matches to find better matches with fewer iterations. The scale infrastructure is already there, it just takes some implementation effort for filters to utilize it, but could further increase performance.

Hardware acceleration

Image processing often relies on hardware acceleration to provide high performance. While our implementation shows that hardware acceleration is not required to achieve interactive performance, it is certainly possible to utilize hardware acceleration to increase performance additionally. Indeed, because of the possibility to use fixed tile sizes our architecture should make hardware acceleration easier, because special cases like border handling are simplified or could be handled on the CPU and the implementation of filters is simplified because of the possibility to use fixed tile sizes for the implementation, ignoring the actual image size.

File formats

Our implementation mainly uses the tiled pyramidal TIFF format, but other file formats are in use which provide scaled random access, like JPEG2000. Also image formats that benefit from performance gains when accessed in a scaled manner can be used to increase performance compared to normal decoding even if random access is not possible, for example raw Bayer pattern camera formats can avoid expensive demosaicing if accessed at a lower scale, because scaling is faster than demosaicing.

Explicit quality-speed trade off

Currently all filters deliver a fixed accuracy which only depends on the input image and on the settings used. A better solution would be to implement a tunable which allows to change the trade off between quality and speed and make the system automatically use this tunable depending on real-time requirements.

Optimal graph configuration

The graph configuration described in section 5.3 certainly works and provides good performance, but the presence of the filter graph allows for much more optimizations. Filters might provide data on their performance characteristics, dependent on their tunables, and with this data the system could utilize a shortest path algorithm to find the fastest way through the filter graph. This would result not in a good but in an optimal solution to the auto-configuration problem.

Fourier and Wavelet transformation

Many filters rely a on Fourier or Wavelet basis for their realization, like multi-scale painting ([BBS94, VP02, PV92]). The required transformations might be implemented similar to the color space conversions, where a filter only states the requirements for its input and the system takes care of the appropriate conversion automatically.

Distributed processing

Our implementation provides efficient threading to increase performance on SMT systems, but with the filter graph, processing could also be accomplished distributed over a network of rendering nodes, to increase performance for a single or for multiple displays connected to those nodes. The possible configurations and the need to distribute the work over multiple nodes either by distributing the graph itself or by dividing a requested area into multiple parts makes this a challenging problem.

9 Conclusion

As our implementation shows, Scale Invariant Image Editing can provide interactive performance on very larger images for both very simple and very complex filters. Example operations implemented ranged from simple contrast and exposure changes, over a Gaussian blur to the relatively complex Non-Local-Means denoising. The use of an abstract specification for image filters allows many optimizations, minimizing the use of color space conversions as well as providing simplified filter implementations. While a scaled implementation of many filters is not possible with perfect accuracy, it was possible for all implemented filters, to either approximate the scaled result sufficiently for interactive work, or to change the filter itself in a way, which made scaled approximation possible. Scale Invariant Image Editing builds on the advantages of graph based image editing and provides a system which can display the result of image operations at any scale and position effectively immediately, without the need to wait for a filter to complete the calculation on the full scale image. Our implementation allows interactive editing of large size images on hardware as slow as current smartphones. While this work gives an overview of the problem, practical solutions, as well as working examples, the architecture has the potential for further improvements, for example by increasing performance using hardware acceleration.

A Appendix

A.1 Accuracy measurements

The following figures show the result of the accuracy measurements for the remaining filters that were not discussed in the main body.

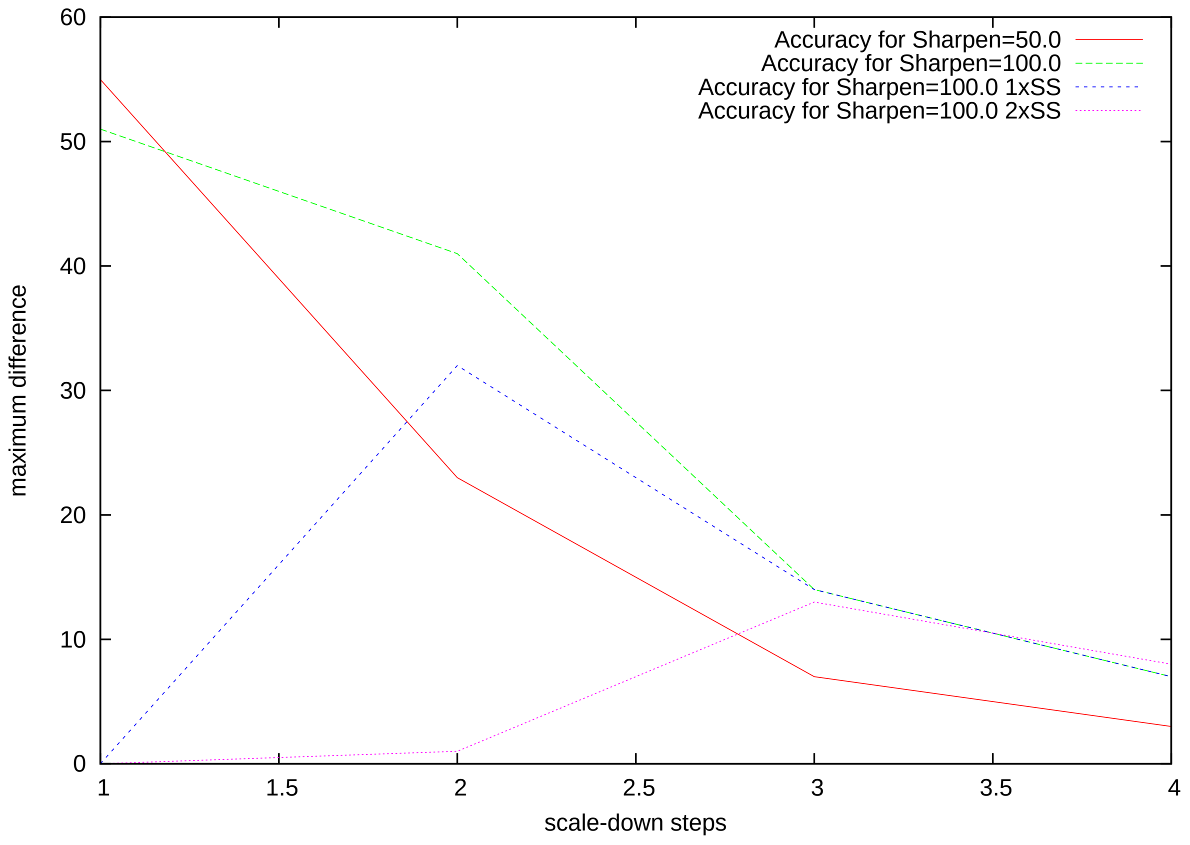


Figure A.1: Accuracy measurements for the sharpen filter

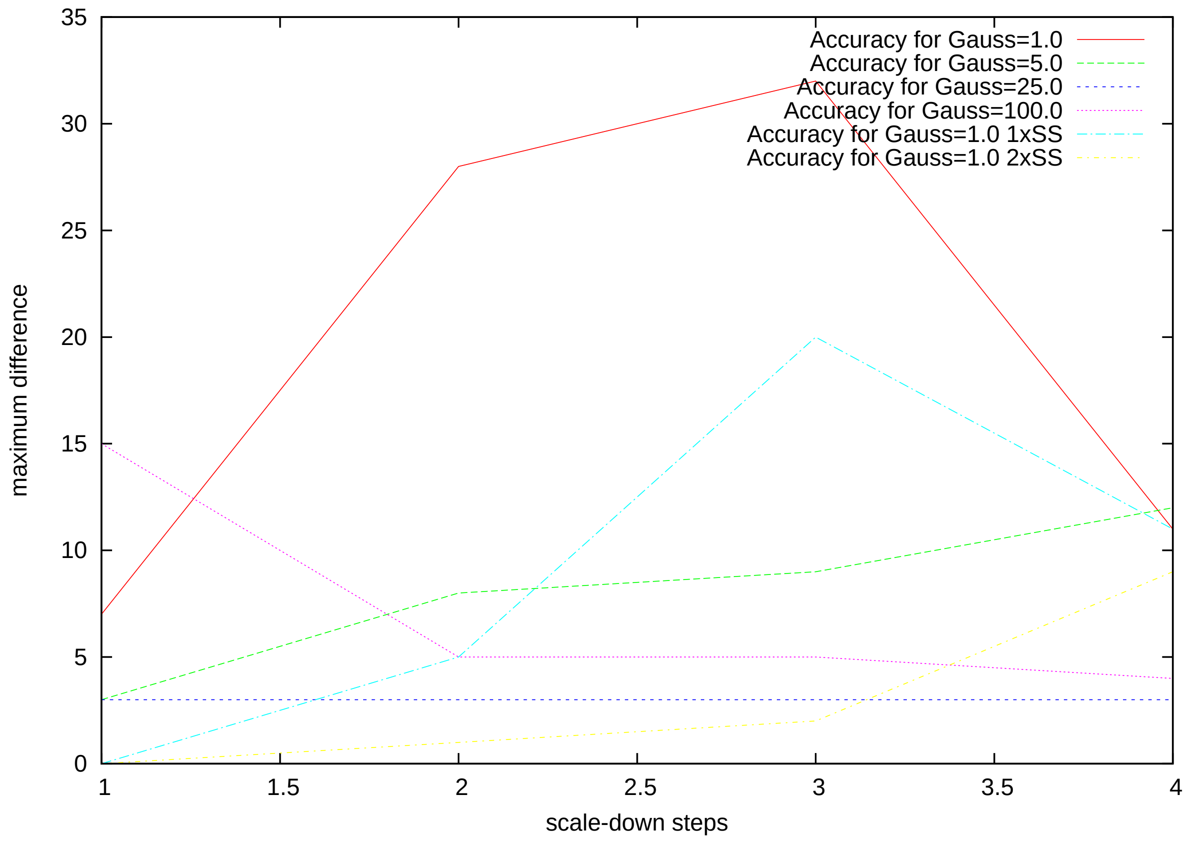


Figure A.2: Accuracy measurements for the Gussian blur filter

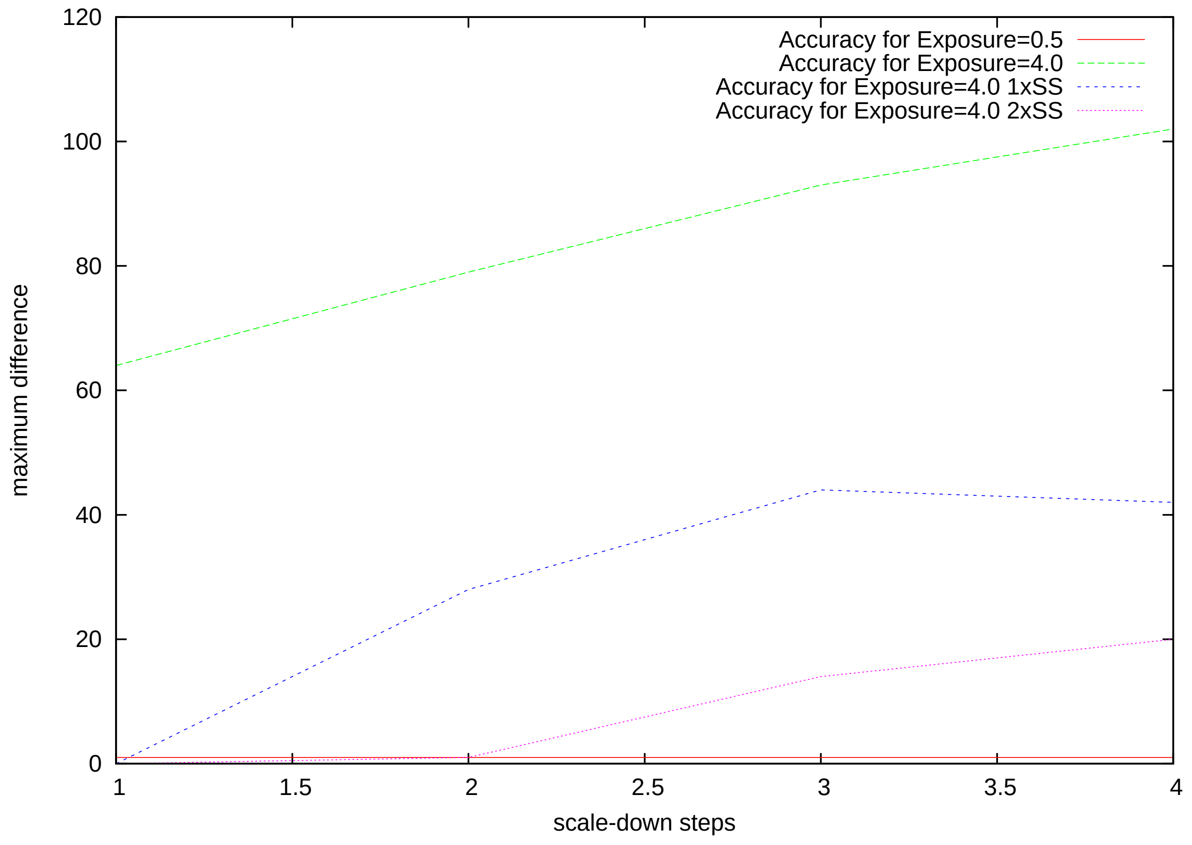


Figure A.3: Accuracy measurements for the exposure filter

A.2 Cache metrics

Below is the full result for all cache metrics and their combinations. This was ommited in our analysis due to the confusingly large number of combinations.

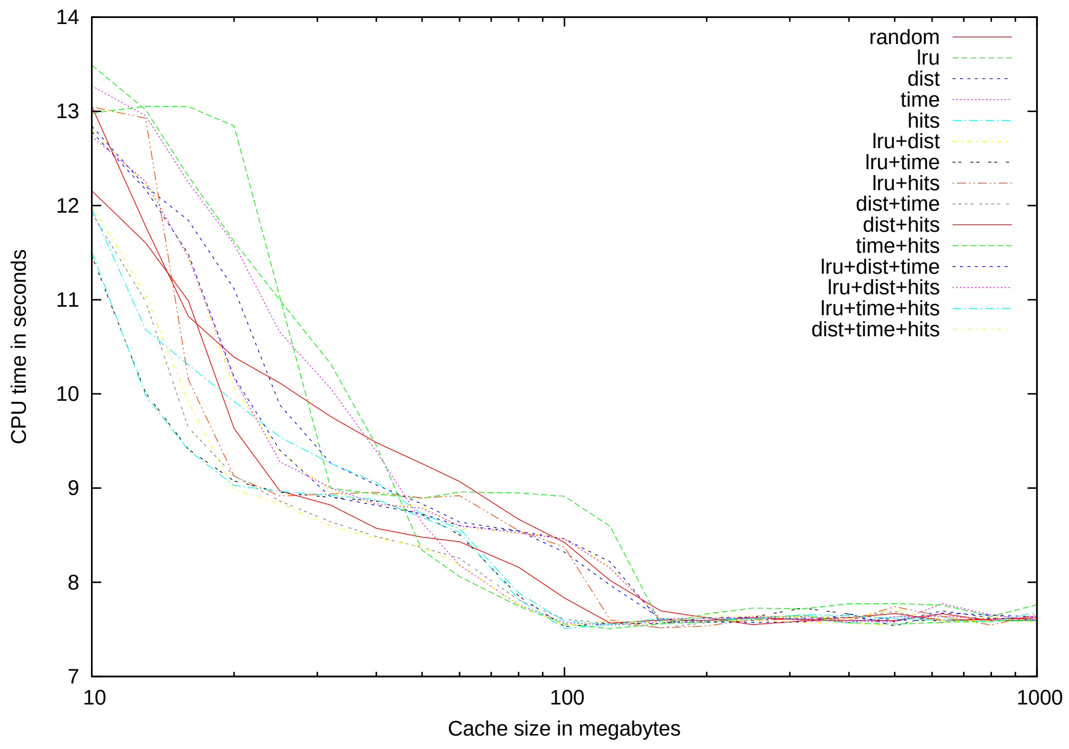


Figure A.4: Execution times for all cache metrics and their combinations, measured agains cache size.

A.3 Program example

Listing A.1 shows the API that can be used to perform scaled rendering. The example code loads an input image from a TIFF file and renders the result of on applied filter to a PPM file. The region and scale can be set as desired and rendering is performed using scaled rendering, which means performance only depends on the filter and the area requested, not on the full resolution are.

```
#include "Lime.h"

void exit_fail(const char *fmt, const char *str)
{
  printf(fmt, str);
  exit(EXIT_FAILURE);
}

int main(int argc, char **argv)
{
  Filter *source, *filter, *sink;
  Rect area;
  uint8_t *buf;
  FILE *ppmfile;

  if (argc != 11)
    exit_fail("usage:\n%s infile filter setting value outfile x y w h scale\n", argv[0]);

  //initialize lime
  lime_init();

  //add source filter and set filename
  source = lime_filter_new("load");
  lime_setting_string_set(source, "filename", argv[1]);

  //add the filter by using the name
  filter = lime_filter_new(argv[2]);
  if (!filter)
    exit_fail("unknown filter %s\n", argv[2]);

  //we assume the setting is of type float
  if (lime_setting_type_get(filter, argv[3]) != MT_FLOAT)
    exit_fail("setting %s must be of type MT_FLOAT\n", argv[3]);

  lime_setting_float_set(filter, argv[3], atof(argv[4]));

  //add the sink filter
  sink = lime_filter_new("memsink");

  //put everything together
  lime_filter_connect(source, filter);
  lime_filter_connect(filter, sink);

  //check configuration
  if (lime_config_test(sink))
    exit_fail("configuration failed, incompatible input?\n", NULL);

  area.corner.x = atoi(argv[6]);
  area.corner.y = atoi(argv[7]);
  area.corner.scale = atoi(argv[10]);
  area.width = atoi(argv[8]);
  area.height = atoi(argv[9]);

  //allocate memory buffer for sink
  buf = malloc(area.width*area.height*3);
  filter_memsink_buffer_set(sink, buf, 0);

  //actual rendering call
  lime_render_area(&area, sink, 0);

  //write rendered buffer to file
  ppmfile = fopen(argv[5], "w");
  fprintf(ppmfile, "P6\n%d %d\n255\n", area.width, area.height);
  fwrite(buf, area.width*area.height*3, 1, ppmfile);
  fclose(ppmfile);

  //cleanup
  lime_shutdown();

  return EXIT_SUCCESS;
}
```

Listing A.1: Example code showing the use of scaled rendering.

Bibliography

[BBS94] D. F. Berman, J. T. Bartell, D. H. Salesin. Multiresolution painting and compositing. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, SIGGRAPH '94, pp. 85–90. ACM, New York, NY, USA, 1994. doi:10.1145/192161.192181. URL http://doi.acm.org/10.1145/192161.192181. (Cited on pages 9, 20 and 73)

[BCM05] A. Buades, B. Coll, J.-M. Morel. A Non-Local Algorithm for Image Denoising. In *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume 2 - Volume 02*, CVPR '05, pp. 60–65. IEEE Computer Society, Washington, DC, USA, 2005. doi:10.1109/CVPR.2005.38. URL http://dx.doi.org/10.1109/CVPR.2005.38. (Cited on page 54)

[Bet] Better Light Large Format Digital Photography. Retrieved September 12, 2012, from http://www.betterlight.com/. (Cited on page 9)

[BSGF10] C. Barnes, E. Shechtman, D. B. Goldman, A. Finkelstein. The Generalized PatchMatch Correspondence Algorithm. In *European Conference on Computer Vision*. 2010. (Cited on page 54)

[Enl] Enlightenment - Beauty at your fingertips. Retrieved September 19, 2012, from http://www.enlightenment.org/. (Cited on pages 43, 50 and 58)

[gam] Gamma error in picture scaling. Retrieved September 19, 2012, from http://www.4p8.com/eric.brasseur/gamma.html. (Cited on page 22)

[GEG] GEGL-0.2.0. Retrieved September 18, 2012, from http://www.gegl.org/. (Cited on pages 15 and 19)

[GGBW12] P. Gwosdek, S. Grewenig, A. Bruhn, J. Weickert. Theoretical foundations of gaussian convolution by extended box filtering. In *Proceedings of the Third international conference on Scale Space and Variational Methods in Computer Vision*, SSVM'11, pp. 447–458. Springer-Verlag, Berlin, Heidelberg, 2012. doi:10.1007/978-3-642-24785-9_38. URL http://dx.doi.org/10.1007/978-3-642-24785-9_38. (Cited on page 53)

[Giga] Gigapan. Retrieved September 19, 2012, from http://gigapan.com/. (Cited on page 9)

[Gigb] Gigapxl Project. Retrieved September 12, 2012, from http://www.gigapxl.org/. (Cited on page 9)

[Gim] Gimp - GNU Image Manipulation Program. Retrieved September 17, 2012, from http://www.gimp.org/. (Cited on pages 9, 14, 15 and 19)

[hug] Hugin - Panorama photo stitcher. Retrieved September 19, 2012, from http://hugin.sourceforge.net/. (Cited on page 9)

[Hut81] J. Hutchinson. Fractals and Self-Similarity. *Indiana University Mathematics Journal*, 30(5):713–747, 1981. (Cited on page 12)

[JJY$^+$11] W.-K. Jeong, M. K. Johnson, I. Yu, J. Kautz, H. Pfister, S. Paris. Display-aware Image Editing. In *International Conference on Computational Photography*. 2011. (Cited on pages 9, 20 and 25)

[jpe] Independent JPEG Group. Retrieved September 19, 2012, from http://www.ijg.org/. (Cited on pages 43 and 48)

[lcm] Little CMS - Great color at small footprint. Retrieved September 19, 2012, from http://www.littlecms.com/. (Cited on pages 43, 45 and 51)

[LJP11] M. Lee, W.-K. Jeong, H. Pfister. Interactive large-scale image editing using operator reduction. In *SIGGRAPH Asia 2011 Posters*, SA '11, pp. 20:1–20:1. ACM, New York, NY, USA, 2011. doi:10.1145/2073304.2073325. URL http://doi.acm.org/10.1145/2073304.2073325. (Cited on page 20)

[Pan] Panoscan Panoramic Camera for High Speed Digital Capture. Retrieved September 12, 2012, from http://www.panoscan.com/. (Cited on page 9)

[Pho] Photoshop. Retrieved September 19, 2012, from http://www.adobe.com/de/products/photoshop.html. (Cited on pages 9 and 14)

[PV92] K. Perlin, L. Velho. A Wavelet Representation for Unbounded Resolution Painting. In *Technical Report*. New York University, 1992. (Cited on pages 9, 20 and 73)

[PV95] K. Perlin, L. Velho. Live paint: painting with procedural multiscale textures. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '95, pp. 153–160. ACM, New York, NY, USA, 1995. doi:10.1145/218380.218437. URL http://doi.acm.org/10.1145/218380.218437. (Cited on pages 9 and 20)

[PV02] S. Pinheiro, L. Velho. A Virtual Memory System for Real-Time Visualization of Multi-resolution 2D Objects. In *WSCG*, pp. 365–372. 2002. URL http://dblp.uni-trier.de/db/conf/wscg/wscg2002.html#PinheiroV02. (Cited on pages 15 and 71)

[Sha94] M. A. Shantzis. A model for efficient and flexible image computing. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, SIGGRAPH '94, pp. 147–154. ACM, New York, NY, USA, 1994. doi:10.1145/192161.192191. URL http://doi.acm.org/10.1145/192161.192191. (Cited on page 19)

[tif] LibTIFF - TIFF Library and Utilities. Retrieved September 19, 2012, from http://www.libtiff.org/. (Cited on pages 43 and 48)

[VIP] VIPS. Retrieved September 18, 2012, from http://www.vips.ecs.soton.ac.uk/. (Cited on pages 14, 15 and 19)

[VP02] L. Velho, K. Perlin. B-Spline Wavelet Paint. *Revista de Informatica Teorica e Aplicada (RITA)*, IX(2):100–119, 2002. (Cited on pages 9, 20 and 73)

[Wel86] W. Wells, III. Efficient Synthesis of Gaussian Filters by Cascaded Uniform Filters. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(2):234–239, 1986. doi:10.1109/TPAMI.1986.4767776. URL http://dx.doi.org/10.1109/TPAMI.1986.4767776. (Cited on page 53)

[WH04] S. Wang, W. Heidrich. The Design of an Inexpensive Very High Resolution Scan Camera System. *Computer Graphics Forum*, pp. 441–450, 2004. (Cited on page 9)

[ZG08] M. Z. M. Zhang, B. K. Gunturk. Multiresolution Bilateral Filtering for Image Denoising, 2008. URL http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=2614560&tool=pmcentrez&rendertype=abstract. (Cited on pages 9 and 20)

All links were last followed on Septermber 19, 2012.

Declaration

All the work contained within this thesis, except where otherwise acknowledged, was solely the effort of the author. At no stage was any collaboration entered into with any other party.

(Hendrik Siedelmann)