5th SC@RUG 2008 proceedings

Smedinga, Rein; Isenberg, Tobias

SC@RUG 2008 proceedings

Rein Smedinga
Tobias Isenberg
editors

2008
Groningen

ISBN 978-90-367-3522-3
Publisher: Bibliotheek der R.U.
Title: Proceedings 5th Student Colloquium 2007-2008
Computing Science, University of Groningen
NUR-code: 980

Contents

1 Generating Artistic Effects Using Edge and Corner Preserving Smoothers – Sander Land, Bob Dröge 6

2 Recent improvements in on-the-fly garbage collection methods – Wieger Hofstra, Bram Noordzij 12

3 Art-based Rendering with Graftals – Joël van Neerbos, Watze Winsemius 18

4 High Quality Printing Of Pen-and-Ink Rendering Methods – Hedde Bosman, Imco Veenstra 25

5 Non-Photorealistic Expressive Modeling and Animation – Jaap Bresser, Nico de Poel 31

6 The User-friendliness of NPR Interfaces – Tim Havinga, Jasper Hafkenscheid 37

7 Architecture documentation on design rationale and decision –Bart van Teeseling, Arnaud van Gelder 43

8 Creation and utilization of Pattern Languages – Dyon Keupink, Martijn de Groote 49

9 Using Model Checking to Prevent Data Loss by File System Errors – Simon P. Takens, Dennis Kanon 62

About SC@RUG

Introduction SC@RUG (or student colloquium in full) is a course that master students in computing science follow in the first year of their master study at the University of Groningen.

In the academic year 2007-2008 SC@RUG was organized for the fifth time as a conference. Students wrote a paper, participated in the review process, gave a presentation and were session chair during the conference.

The organizers Tobias Isenberg and Rein Smedinga would like to thank all colleagues, who cooperated in this SC@RUG by collecting sets of papers to be used by the students and by being an expert reviewer during the review process. They also would like to thank Femke Kramer from the Faculty of Arts for her help in organizing this course and Janneke Geertsema for her workshops on presentation techniques and speech therapy.

In these proceedings all accepted papers are published.

Organizational matters SC@RUG 2008 was organized as follows. Students were expected to work in teams, consisting of two persons. The student teams could choose between different sets of papers, that were made available through *Nestor*, the digital learning environment of the university. Each set of papers consisted of about three papers about the same subject (within Computing Science). Some sets of papers contained conflicting opinions. Students were instructed to write a survey paper about this subject including the different approaches in the given papers. The paper should compare the theory in each of the papers in the set and include own conclusions about the subject. Some teams proposed their own subject.

After submission of the papers individual students were assigned one paper to review using a standard review form. The staff member who had provided the set of papers was also asked to fill in such a form. Thus, each paper was reviewed three times (twice by peer reviewers and once by the expert reviewer). Each review form was made available to the authors of the paper through *Nestor*.

All papers could be rewritten and resubmitted, independent of the conclusions from the review. After resubmission each reviewer was asked to re-review the same paper and to conclude whether the paper had improved. Re-reviewers could accept or reject a paper. All accepted papers can be found in these proceedings.

All students were asked to present their paper at the conference and act as a chair and discussion leader during one of the other presentations. Half of the participants were asked to organize the of the conference day (i.e., to make the time tables, invite people etc.) The audience graded both the presentation and the chairing and leading the discussion.

Femke Kramer of the Faculty of Arts gave an introductory lecture about general aspects of presentation techniques to help the students with their presentation. She also taught a workshop on writing scientific papers. Janneke Geertsema gave workshops on presentation techniques and speech therapy that was very well appreciated by the participants.

Students were graded both on all three aspects: the writing process, the review process and the presentation. Writing and rewriting counted for 50% (here we used the grades given by the reviewers and the re-reviewers), the review process itself for 15% and the presentation for 35% (including 5% for the grading of being a chair or discussion leader during the conference). For the grading of the presentations we used the judgements from the audience and calculated the average of these.

In this edition of SC@RUG students were videotaped during their presentation. Jan Apotheker of the Institute of Didactics and Education provided equipment so that students could copy this recording on DVD.

On January 24th, the actual conference took place. Each paper was presented by both authors. That day, we had twelve presentations, each consisting of a total of 20 minutes for the presentation and 10 minutes for discussion. As mentioned before, each presenter also had to act as a chair and discussion leader for another presentation during that day. The audience was asked to fill in a questionnaire and grade the presentations, the chairing and leading the discussion. Participants not selected as chair were asked to organize the day. They did an excellent job and even provided coffee and tea and a lunch.

Thanks We could not have achieved the ambitious goal of this course without the invaluable help of the following expert reviewers: Marco Aiello, Paris Avgeriou, Henk Bekker, Wim Hesselink, Tobias Isenberg, Jan Jongejan, M Ikram Lali, Peng Liang, Nikolay Petkov, Gerard Renardel, Jos Roerdink, Alex Telea, Michael Wilkinson

Also, the organizers would like to thank the *School of Computing and Cognition* for making it possible to publish these proceedings and EyeToEye Informatica for sponsoring the conference.

Rein Smedinga

Generating Artistic Effects Using Edge and Corner Preserving Smoothers

Sander Land and Bob Dröge

University of Groningen

Abstract. In image processing, a special class of filters exists that can remove noise while preserving or enhancing edges and corners. Some of these edge and corner preserving smoothers can be used to generate artistic effects, effectively turning an image into a painting. Papari et al. proposed a new filter of this class which was designed specifically for this purpose. In this paper, we compare this new filter with several others. We not only show which of these smoothers are capable of generating artistic effects, but also how well they perform in removing noise.

1 Introduction

A major facet in image processing is the removal of noise. Several smoothing filters exists which are capable of removing noise, but often these filters blur sharp edges as well. A special class of smoothing filters, the so-called edge and corner preserving smoothers (ECPS), solves this problem by smoothing out texture details and removing noise while preserving or even enhancing edges.

Usually, paintings differ from photographic images in having sharp edges and not having texture details. Some ECPSs, when applied on a photographic image, will replace the small details by more homogeneous patches, while preserving or sometimes enhancing the edges and corners. Therefore, the resulting image will have a painting-like effect.

In this paper we discuss several existing ECPSs and review their ability to create artistic images. In the following section we introduce the ECPSs to be reviewed and the way they work. In section 3 we introduce a set of test images which we use to compare the ECPSs. In section 4 we compare the results of these ECPSs for this test set, both on their ability to generate artistic effects and on their ability to remove noise. Finally in section 5, we give a conclusion about the various ECPSs.

2 ECPS Filters

In this section, we give a detailed description of the various ECPSs which we implemented or used in our implemetation and hence which will be compared. This set of ECPSs consists of the median filter [5], the Kuwahara filter [1,2], the Papari-Petkov-Campisi filter [3], the weighted Kuwahara filter and finally the Rudin-Osher-Fatemi denoising model [4].

2.1 Median Filter

The median filter [5] is one of the most basic edge preserving smoothers. It simply replaces the color of the pixel with the median of the values in a $k \times k$ area centered on the pixel.

This filter is well-known for its ability to remove noise, especially salt-and-pepper noise, while preserving edges. It is

also extremely simple and, for small sizes, the fastest among the filters we will discuss. To increase the amount of noise reduction, the filter size k can be increased, or the filter can be iterated. We implemented both of these options as parameters.

2.2 Kuwahara Filter

The Kuwahara filter [1,2] is another well-known ECPS. Given a grayscale image, the Kuwahara filter looks at four square neighbourhoods of size $k \times k$, overlapping only on a central pixel, and replaces the value of the central pixel with the average value of the most homogeneous neighbourhood. More formally, the means μ_i and variances σ_i^2 of all the neighbourhoods are calculated, and the pixel is replaced with the μ_i of the region with the lowest σ_i^2. Because all the neighbourhoods are of the same size, only two convolutions are needed (one for the means and one for the variances), making the filter almost as fast as a single iteration of a median filter of size k.

Generalizing this filter for use with full-color images by simply applying the filter to each of the color channels does not work very well , because different color channels may have different regions with minimal σ_i^2, and choosing the μ_i from different regions could introduce colors very different from those in the original image. Instead, the minimal total variation $\|(s_r, s_g, s_b)\| = \sqrt{s_r^2 + s_g^2 + s_b^2}$ is used to pick a region, and the means of all the three color channels of that region are used. The filter is also well-known for its capability to generate artistic effects, even though it was not specifically created to do this.

There are many variations of the Kuwahara filter. The most common variations involve varying the number of regions, the shape of the regions, or function used to choose the pixel color given the mean and standard deviation values.

2.3 Papari-Petkov-Campisi Filter

Papari et al. [3] developed a variation of the Kuwahara filter, which involves a variable number of regions (N). These regions are circle sectors of a gaussian function with standard deviation σ.

Also, instead of simply choosing the region with the lowest variance, a weighted sum is used. The output of the pixel is given by $\frac{\sum \mu_i s_i^{-q}}{\sum s_i^{-q}}$, where μ_i, s_i are the means and variances and q is a parameter which influences the edge and corner preservation. For higher q the filter will be more like the standard choice function, and for lower q it approaches Gaussian smoothing.

The Papari-Petkov-Campisi filter counters a weakness in the Kuwahara filter that occurs when two regions have almost identical variance: the Kuwahara filter will pick one, almost at random, and this gives very discontinuous results. On the other hand, the Papari filter is two or more times slower than the Kuwahara filter.

Weighted Kuwahara Filter We also introduce a new edge preserving smoother, the weighted Kuwahara filter, by applying the weighting function proposed by Papari et al. to the standard Kuwahara filter. This filter is nearly as fast as the standard Kuwahara filter, but does not have the weaknesses inherent in the discontinuous choice function.

2.4 Rudin-Osher-Fatemi Denoising Model

The Rudin-Osher-Fatemi denoising model [4] is a more mathematical approach to edge preserving smoothing. It assumes that the given image $u_0(x, y)$ is the sum of the original image $u(x, y)$ and additive white noise $n(x, y)$ with mean zero and standard deviation σ. The noise needs to be removed, restoring the original image. This is done by finding an image u such that

$\int \sqrt{u_{xx}^2 + u_{yy}^2}$ is minimized under the constraints $\int u = \int u_0$ and $\int (u - u_0)^2 = 2\sigma^2$.

Because additive noise adds edges on all sides of the noisy pixel, while being easy to remove without changing the image too much, this method excels at removing such noise. It also preserves the rest of the image because of the strong constraints.

This implementation involves a parameter λ, which is related to the denoising strength. Higher λ will result in stronger denoising. There is also a parameter which limits the number of iterations for the numerical method which solves the resulting set of partial differential equations.

Fig. 1. The test images used. The top row has images with less detail (in the rest of the paper referred to as the 'bird' and 'grass' images), while the bottom row has images with all levels of detail (referred to as the 'field' and 'tracks' images).

3 Test methods

We will use four test images to compare these edge preserving smoothers. These images are shown in figure 1. The images were chosen for being suitable for artistic effects, and differing in their amount of detail, so the edge preserving effects of these filters can be compared.

The Kuwahara filter will be tested for sizes between 3 and 11 and the weighted Kuwahara filter also for $q \in \{2, 4, 8\}$. The Papari-Petkov-Campisi filter will be tested for 4, 6, 8 or 12 regions, $\sigma \in \{1, 2, 3, 5, 7, 9\}$ and $q \in \{3, 6, 12\}$. The median filter is tested for sizes between 3 and 9 and up to 16 iterations. Finally, the Rudin-Osher-Fatemi filter is tested for λ between 64 and 1024 and an iteration limit between 64 and 1024.

The test method consists of two parts: first we will compare the ECPSs based on generating artistic effects. In order to rank the ECPSs, we will pick the best painting-like image for every method by looking at the preservation of edges and corners and the removal of texture details. The second part of the test consists of a comparison based on noise removal, for which we use copies of the 'field' image with Gaussian noise (with zero mean and $\sigma = 0.01$) added. For this part of the test, we will rate the results on the amount of noise in the resulting image and the quality of the resulting image (preservation of edges, corners and details).

4 Results

In this section we show some results of the filters applied to images of the test set. We only show some of the, in our opinion, best results. However, the full set of results (containing nearly 800 images) is available on the internet. [1]

4.1 Artistic effects

The median filter hardly produces artistic effects for any of the test images: for small values of r, the result is still more photo-realistic than artistic. For higher values of r, the result is a very blurred image.

More or less the same holds for the Rudin-Osher-Fatemi filter: a lower number of iterations gives too realistic results, while using more iterations the image gets blurred too much. The value of λ hardly changes anything.

[1] http://d0td0td0t.com/images.tar.gz

The (weighted) Kuwahara filter does a better job at producing artistic effects. A size of about 5 replaces the too detailed parts by homogeneous patches. In case of the weighted Kuwahara filter, higher values of q, e.g. $q = 8$, result in stronger edges. However, there are hardly any differences in the best results of the Kuwahara and weighted Kuwahara filter.

The Papari-Petkov-Campisi filter usually gives the best artistic effects using $3 \leq \sigma \leq 5$. For lower values, the result is too photorealistic, while higher values result in very sharp edges and loss of too many details. However, for the 'field' and 'tracks' test images, using these values of σ already result in too much loss of details. For example, due to the close distance of the various cross ties, they are blurred to a brown-gray patch. Therefore, $\sigma = 2$ works better for these images. Furthermore, a high value of q, about $8 - 12$, preserves the edges and corners better and hence gives better results. As Papari et al. claim, the size of N does not significantly influence the edge preservation and usually $N = 8$ or $N = 12$ gives satisfactory results.

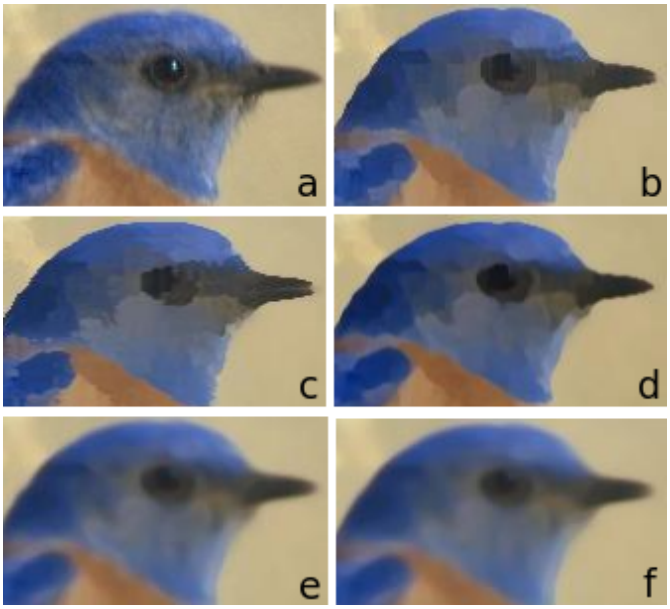


Fig. 2. Some of the best artistic effect results: a) Original, b) Kuwahara weighted ($s = 5$, $q = 8$), c) Kuwahara ($s = 7$), d) Papari-Petkov-Campisi ($\sigma = 3$, $N = 8$, $q = 6$), e) Median ($r = 4$, $N = 5$), f) Rudin-Osher-Fatemi ($\lambda = 64$, $N = 32$)

Some of the best results are shown in figure 2 and 3. As already indicated and as can be seen in these figures, the median filter and Rudin-Osher-Fatemi filter are the least suitable filters for generating artistic effects. Some additional results of the other filters are shown in figure 4. The Kuwahara and weighted Kuwahara filter perform about equally well, but the Papari-Petkov-Campisi filter just performs better in preserving edges and corners.

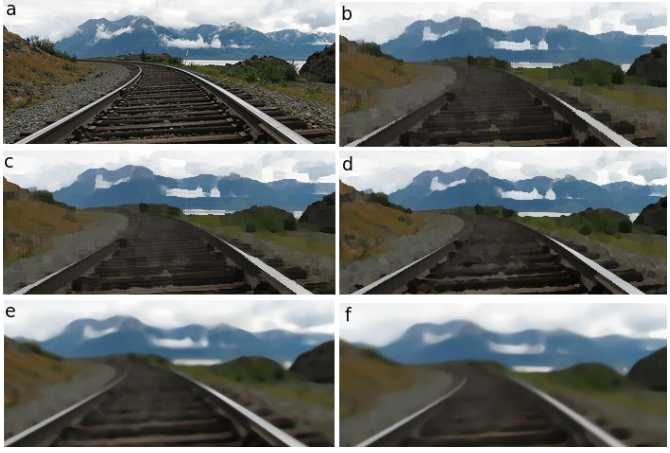


Fig. 3. Some of the best artistic effect results: a) Original, b) Kuwahara weighted ($s = 5$, $q = 8$), c) Kuwahara ($s = 5$), d) Papari-Petkov-Campisi ($\sigma = 2$, $N = 12$, $q = 12$), e) Median ($r = 4$, $N = 7$), f) Rudin-Osher-Fatemi ($\lambda = 64$, $N = 64$)

4.2 Noise removal

The median filter removes a decent amount of this noise for a size of $3 - 4$ and $2 - 3$ iterations, but higher sizes or more iterations degrade the image too much.

The Kuwahara and weighted Kuwahara filters for sizes larger than 5 the filter

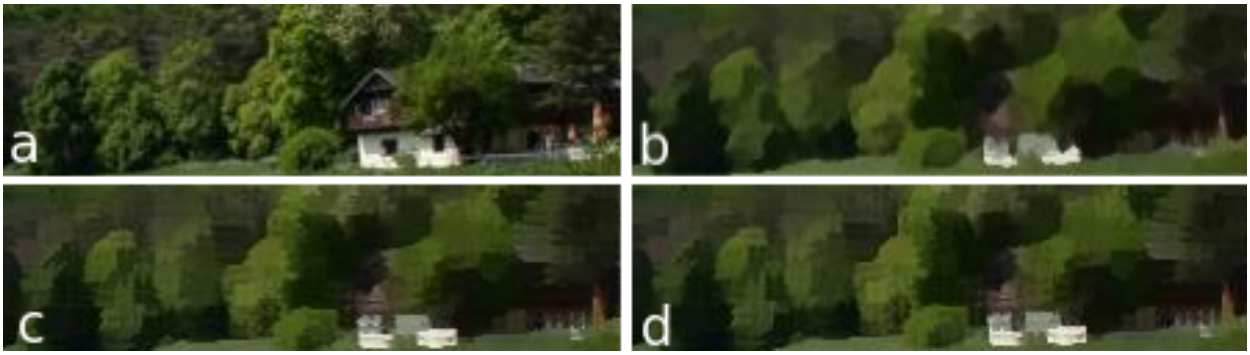


Fig. 4. Some of the best artistic effect results: a) Original, b) Papari-Petkov-Campisi ($\sigma = 3$, $N = 8$, $q = 12$), c) Kuwahara weighted ($s = 5$, $q = 8$), d) Kuwahara ($s = 5$)

are too strong and removes most important details. For the lower sizes there is a clear tradeoff between removing noise (greater size, lower q) and preserving edges (smaller size, greater q).

When using the Papari-Petkov-Campisi filter with $\sigma \geq 5$ many details are lost, although using a high number of regions can compensate a bit. Even at $\sigma = 3$ some important details are lost, so the best effects are obtained when using $\sigma = 1, 2$. At this level there is once again a tradeoff between losing some detail and removing noise.

The Rudin-Osher-Fatemi filter has good noise removal for $\lambda = 64$, iterations $= 16$, but for a higher number of iterations the image is blurred too much and changing λ does not visibly affect the result.

Five of the best results are shown in Figure 5. Even among these there are tradeoffs and no result is clearly the best. The Rudin-Osher-Fatemi filter has the best noise removal, but at a cost of some blurring of edges. The Papari-Petkov-Campisi filter with $\sigma = 2$ has similar performance, but losing slightly more details while enhancing some larger edges.

The Kuwahara filter and the Papari-Petkov-Campisi filter with $\sigma = 1$ perform about the same as eachother, although the Papari-Petkov-Campisi filter preserves

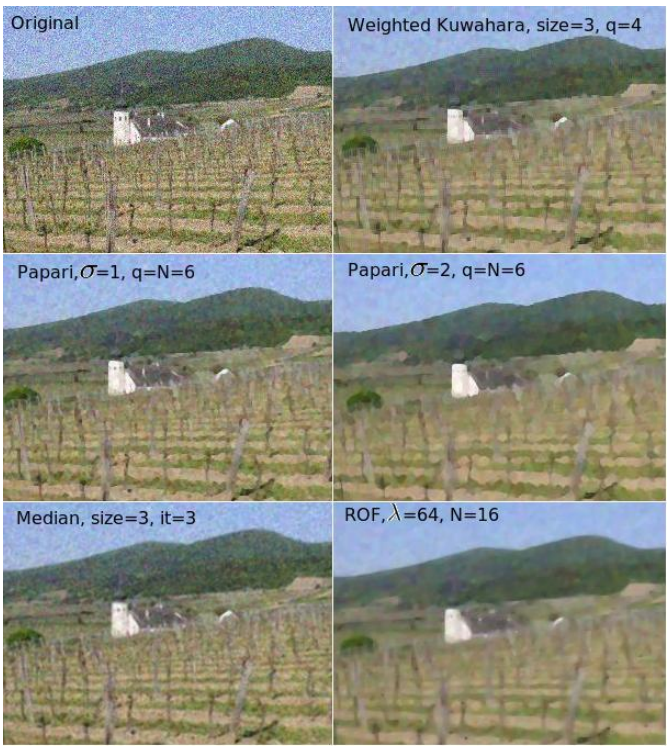


Fig. 5. Some of the best test results for denoising an image.

some more details. However, the Kuwahara filter is much faster.

The median filter is the worst of them all, although it has decent results considering its speed and simplicity.

For higher amounts of noise the results are nearly the same in terms of the maximum settings that are usable before the image is blurred too much, and the best settings for the filters, so we will not repeat them here. We did however discover a nice effect that appears when using the Papari-Petkov-Campisi filter for a low sigma and a rather high N and q on images with a lot of noise. As can be seen in figure 6, the high amount of noise causes patches to appear in the sky, even though this region was originally homogeneous.

Fig. 6. Part of the results of the Papari-Petkov-Campisi filter with $\sigma = 2, q = 12, N = 12$ applied to the 'field' image with Gaussian noise with zero mean and $\sigma = 0.05$.

5 Conclusion

We gave a comparison, based on generating artistic effects and removing noise, of the following edge and corner preserving smoothers: a median filter, a Kuwahara filter, the Papari-Petkov-Campisi filter, the weighted Kuwahara filter and the Rudin-Osher-Fatemi denoising model.

For generating artistic effects, the Papari-Petkov-Campisi filter indeed gave the best results, as was claimed in the paper by Papari et al. The Kuwahara and weighted Kuwahara filter performed well too, but the median filter and Rudin-Osher-Fatemi filter gave poor results.

The other part of the comparison, based on removing noise, did not really show a clear winner, though the median filter performed worst.

We conclude that even among filters that are designed specifically to preserve or enhance edges there is a tradeoff between the denoising strength and the edge, corner and detail preservation: better denoising filters blur edges and corners or lose details, while better edge and corner preserving filters are worse at removing noise.

References

1. Kuwahara, M.; Hachimura, K.; Ehiu, S.; Kinoshita, M. *Processing of riangiocardiographic images* Digital Processing of Biomedical Images (1976) 187–203
2. Nagao, M; Matsuyama, T. *Edge Preserving Smoothing* Computer Graphics and Image Processing 9 (1979) 394–407
3. Papari, G.; Petkov, N.; Campisi, P. *Artistic Edge and Corner Enhancing Smoothing* IEEE Transactions on image processing, Vol. 16, Issue 10 (2007) 2449–2462
4. Rudin, L.I.; Osher, S.; Fatemi, E. *Nonlinear total variation based noise removal algorithms* Physica D, Vol. 60 (1992) 259–268
5. Tukey, J.W. *Exploratory Data Analysis* Addison-Wesley (1971)

Recent improvements in on-the-fly garbage collection methods

Wieger Hofstra, Bram Noordzij

February 18, 2008

Abstract

With multiprocessor systems becoming the standard as well as the high demand for automatic memory management, the need for garbage collection methods which utilize all available processors is increasing. Most of the garbage collection methods currently in use incorporate synchronization algorithms requiring big locks, or even worse, many have the necessity to temporarily stop the application. These algorithms introduce uncontrollable delays while executing an application. These delays are a big obstacle towards high performance and they make garbage collection unusable for time critical applications. This paper contains an overview of recent developments in the field of on-the-fly garbage collectors, which promise to improve performance and possibly overcome the aforementioned problems.

Keywords: Memory management, Garbage collection, Reference-counting, Cycle collection

1 Introduction

Since the original invention of Garbage Collection (GC) by John McCarthy in 1959, this field of research has seen numerous developments, of which the most recent will be discussed. Automatic memory management, which fully depends on GC, is an important tool, aiding in the fast development of large and reliable software systems. It is a must in popular rapid development languages like Python and Ruby. In addition, popular programming environments like Java and .NET heavily rely on GC. While making the life of a developer easier and programs more stable with the elimination of most types of memory leaks, GC has a major drawback: the garbage collection process has a significant impact on the overall runtime performance of a system. The allocation and reclamation of memory space takes a noticeable percentage of the overall execution time. Therefore, developing smarter and more efficient garbage collection methods is an important field of research.

This paper surveys a selection of some of the most recent and promising techniques, which greatly improve garbage collection performance. First, in section 2 an overview of the basics of garbage collection algorithms is given. Secondly, in section 3 a novel reference counting algorithm created by Levanoni and Petrank[7] is presented, followed by recent improvements for cycle collectors by Paz et. al.[2] in section 4. Finally we present a conclusion.

2 Garbage Collectors

For simple memory allocation methods such as static and stack allocation, there is no need for garbage collection. However, these allocation methods are limited: the size of datastructures must be known at compile time. This is a severe limitation on data structures like lists and trees. This making it impossible to use dynamically-sized objects as return values of functions. Heap allocation resolves these issues, but comes at a prize: the administration of allocated and free memory is much more complex and since it dynamically allocates and deallocates in arbitrary order, it leaves 'holes' of free memory between blocks of allocated memory. When the application needs to allocate a data structure larger in size than the largest consecutive free space (or hole), the heap needs to be *compacted*, which is the process of reallocating all the allocated memory sequentially. The deallocation is the responsibility of the programmer in languages like C, C++ and Pascal. While this can be very

efficient, it is the cause of many programming errors such as memory leaks (forgetting to deallocate storage no longer in user) and double free errors (deallocating the same storage twice). Many modern programming languages offer automatic storage management. The task of this management is to collect *garbage*. An object in the heap is *live* when its address is held in a *root*, which can be: values in registers; the stack and global variables. Moreover, if there is a pointer to it held by another live heap object, it is also a live object. Objects that are not live, but not free either, are called *garbage*.

To determine the liveliness of objects, there can be two methods distinguished: direct and indirect. A direct method to determine liveliness, is to keep a *Reference Count* for every object which stores the number of pointers to the object. The alternative, indirect, method is to use a *tracing* collector. These collectors take all the roots as a starting point, follow all the pointers and thereby visit all the reachable objects. There are many direct and indirect algorithms and optimizations on these algorithms. In this paper we discuss a reference counter, complemented by a so called indirect mark-and-sweep collector as well as an innovative cycle collector.

3 Reference Counting

3.1 Introduction

As previously explained, conventional tracing collectors must traverse all live objects. The more live objects in the heap, the more work for the collector. However, when using reference counting, the asymptotic complexity changes from being proportional to the space in use by all the live objects, to just being proportional to the amount of work done by the *mutator* (the user program) per collection interval, in addition to the amount of space reclaimed. Reference-counting is consequently very promising for future garbage-collected systems, since the memory size is growing faster than processor speed; the increase in usage of very large heaps and the spread of the 64-bit architectures.

Reference-counting is an intuitive method for automatic storage management and follows the following simple steps:

1. Attach a counter to each object in memory
2. When a new reference is connected to that object, increment the counter
3. When a reference is removed, decrement the counter
4. Any object with counter value 0 is garbage, it can immediately be reused

However, there are two major disadvantages to reference counting. When two or more objects refer to each other they can create a cycle whereby neither will be collected as their mutual references never let their reference counts become zero. Also in naive implementations, each assignment of a reference and each reference falling out of scope, often requires modifications of one or more reference counters.

Many GC algorithms run on a single thread and require all mutators to be suspended. This is not suitable for a multiprocessor environment. So far, there is little research about reference-counting on multiprocessors, in comparison to the study of concurrent and parallel tracing collectors. The reason for this is that traditional reference-counting methods have problems with concurrency: the update of a reference-counter must be atomic, since they can be updated by concurrent mutators. A second problem updating a reference, is that a mutator must be aware of the previous value of the reference slot being updated. A single error in this process will corrupt memory and will result in a memory leak, data corruption or a total crash. A simple solution to this problem is the use of a locking mechanism on an update operation.

3.2 Recent improvements

Recently completed research done by Levanoni and Petrank[7] introduces a reference counting method that eliminates the need for this unwanted locking mechanism. The approach does not require any synchronized operation in its *write-barrier*[1], which is obviously a big improvement since modification of datastructures is very common in almost all applications. In addition, the algorithm eliminates most of the reference-count updates, thereby greatly improving its efficiency. By eliminating all

[1] Memory management code which is executed whenever a reference slot is modified

synchronization operations, an important step towards *on-the-fly* garbage collection[2] is made.

Updating a reference-count is an expensive operation: the operation of a write barrier; synchronization and possible the paging in of a paged-out object[3] are costly. This algorithm, which is partially based on work of Deutsch and Bobrow [5], does not keep track of the changes to the local references. The method only keeps track of references in the heap. When GC is needed, the collector inspects all objects in the heap with a reference-count of zero. By the roots unreferenced objects may be reclaimed. Levanoni and Petrank greatly improve on the work of Deutsch and Bobrow by removing many redundant and avoidable updates of the reference-counts. The following example is given:

Consider a reference slot p which is, between two garbage collection iterations, sequentially assigned with the values o_0, o_1, o_2,...,o_n, for objects o_0,...,o_n in the heap. All previous reference-counting collectors performed $2n$ updates on the reference counter:

$$RC(o_0)--, RC(o_1)++, RC(o_1)--,$$
$$RC(o_2)++, RC(o_2)--, ..., RC(o_n)++$$

However, only two are really required:

$$RC(o_0)--, RC(o_n)++$$

Building on this observation, it follows that in order to update all reference counts of all objects before a garbage collection, it is enough to know which reference slots have been modified between the collections. For each such slot, the value in the previous garbage collection and its current value must be determined. The difficulty in this approach is to keep a record of all changed reference slots *and* the o_0 value that was in the slot before it was first modified. It can be problematic to obtain the o_0 value in a concurrent setting. Levanoni and Petrank describe how this can be done without the need for synchronization. Since the reference-counts are only updated at the start of a garbage collector run, it is said that the garbage collector operates on a *snapshot*.

Taking a snapshot requires to stop all mutators. This is undesirable, since the goal is an on-the-fly collector. A smart approach is to halt only one thread at the time instead of all of them. The problem is however that we then obtain a sliding view instead of a snapshot. With a snapshot we have the content of the heap at time exactly t, with a sliding view we know the content associated with a time t somewhere between the time the first thread is suspended, t_1 and the time the last one is suspended, t_2. The solution described in Levanoni and Petrank[7] works with a snooping mechanism, which is implemented in the write barrier, and is in use during the time between t_1 and t_2. The snooping mechanism stores the addresses of all objects which have acquired a new reference, which are then treated as roots. Therefore they are not reclaimed, this makes the sliding view safe.

The reference-counting algorithm designed by Levanoni and Petrank still does not solve the problem of circular referencing cycles. To overcome this problem, the implementation of the algorithm uses an on-the-fly mark-and-sweep collector[4] at a very low interval to collect and restore stuck reference-counts. References can get stuck because the implementation uses only 2 bits for the reference-count like in the implementation of Roth and Wise[9], Wise[12], Stoye et al.[10], and Chikayama and Kimura[11]. In the event this happens it can be repaired by the mark-and-sweep algorithm. In benchmarking the complete algorithm it turns out it performs really well and has an extremely low latency. These findings are confirmed by Domani et al.[6].

4 Cycle collection

4.1 Introduction

Tracing collectors are known by this name because they trace through the working set of memory and perform collections in cycles. The original tracing collector employs a naive mark-and-sweep method,

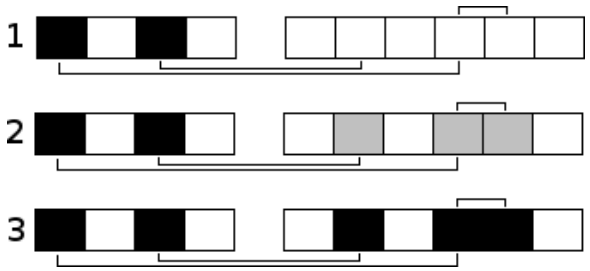
[2] On-the-fly Garbage Collectors do their work in a non-intrusive way, by removing the big delays which are present in most other GC algorithms.

[3] A paged-out object is a part of memory temporarily stored on the hard disk, instead of the main memory. Moving this object back, or paging in, is expensive since hard disk are much slower than main memory.

[4] A garbage collection method that recursively traces and marks objects starting from the roots, then frees all non-marked objects.

in which the entire memory set is touched several times. In this method, every object in memory has an extra bit reserved for garbage collection, initialized as cleared. During a collection cycle the tracer sweeps the entire root set, marking each accessible object. All transitively accessible objects from the root set are marked as well. After the sweep is completed the memory is examined again, all objects with the extra bit still cleared will be reclaimed. In the final step of the algorithm all the flags of the objects still in use will be set as cleared. This method has several disadvantages, most notable: the need for a stop-the-world event and the need for an examination of the complete working memory, most of it twice.

Most modern tracing garbage collectors implement some variant of the tri-colour marking abstraction. Tri-colour marking works as follows:



1. Init stage: Mark objects than cheaply can be proven to have no references to objects in the white set to black
2. Mark stage: trace reachable objects, subtract self references, mark all traversed nodes *grey*. When completed, all unreachable elements will have reference-count zero
3. Scan stage: trace through all grey elements, mark those with positive reference-count and their descendants *black*, mark all other elements *white*
4. Collect stage: reclaim all *white* elements

The tri-colour marking algorithm preserves an important invariant: no black object points directly to a white object. This property ensures all white objects can be safely reclaimed once the grey set is empty. A great advantage of the tri-colour method is that by marking objects during allocation and mutation, the algorithm can be performed on-the-fly.

The research of Bacon and Rajan[4] proposes two cycle collectors. The first one is a simple synchronous collector that requires a stop-the-world event. This collector is the most efficient cycle collector known today. The second one is the only on-the-fly asynchronous cycle collector currently available. However the asynchronous collector requires a lot of work to make it safe: in order to discover unreachable cycles, the live objects are traced multiple times. A problem arises when a mutator changes the object graph while this process is running. This results in two drawbacks: in order to achieve safety, all objects have to be scanned multiple times and the completeness can not be guaranteed. A rare race condition may prevent an unreachable cyclic structure from being ever reclaimed.

4.2 Recent improvements

Recent research done by Paz et al.[2] proposes an algorithm for on-the-fly cycle collection which solves these drawbacks. The new algorithm uses the sliding views techniques from research by Levanoni and Petrank[7] and applies these methods to the cycle collector. The idea is to 'fix' the changing object graph problem. It is using sliding views as in [7] and changes this model so it can scan for object graphs like in [1]. To improve speed of this algorithm an improved implementation of Bacon and Rajan[4] is applied to detect cyclic structures faster and more efficiently.

The use of sliding views for cycle collection is not trivial. Most cycle collectors need a list of all decrements of reference-counts to work on. This is a problem when introducing sliding views. The sliding views reference-quoting collectors basically improve the performance because they do not keep track of all decrements. Paz et al. find a solution in carefully analyzing the sliding view collector. They discovered that the collector can actually do its work by tracing only the recorded decrements. In this case it also needs to record newly created objects. The sliding view saves the addresses of all objects which have received a new reference, so it is easy to adapt this to create a separate list of the new objects. The implementation uses an Update and a YoungObjects buffer. The latter does not need to be checked by the cycle collector.

The research of [2] also detects, and later proves

by measurement, that newly created objects substantially add to the burden of the cycle collector. To exploit this characteristic, an age-oriented collector [8] was implemented. The age-oriented collector keeps track of generations, but it does not run on frequent young generation collections. The reason is that short pauses are obtained by concurrency already and do not need to be obtained by short young collections. The heap is collected only when it is full. When that happens, the age-oriented collector uses a reference counting collector to reclaim objects in the old generation and mark and sweep collector to reclaim objects in the young generation. Since these collections always happen together, there is no need to record inter-generational pointers. Using the age-oriented approach it is possible to eliminate a significant fraction of cycles and also eliminate a large fraction of the cycle collectors work, because it is no longer necessary to consider young objects, increasing the efficiency of the cycle collector.

Implementing all these optimizations results in an on-the-fly cycle collector with the same short pauses of recent on-the-fly collectors like in [7], [1] and [3]. In the benchmarks the cycle collector is outperformed by reference-counting with backup tracing collectors implementations. However they also measured the cycle collector when the reference counting was run only on objects in the old generation, in that case the cycle collector performed equally, and even better on tight heaps. In the future when heaps and live data become much larger, the implementation of Paz et al. may become a more and more effective GC method.

5 Conclusion

Vast amount of progress has being made on all aspects of automatic memory management. Levanoni and Petrank introduce a new on-the-fly reference-counting garbage collector, significantly reducing the overhead of reference slot updates. Using extremely fine synchronization it can avoid any synchronization in the write barrier. These improvements not only create a very efficient reference-counter, they also allow the algorithm to be used for multithreaded programs on multiprocessor systems.

Paz et al. use and adapt advancements made by Levanoni and Petrank to create an efficient on-the-fly cycle collector. Their implementation combines the use of sliding views and aging-collection methods. Both implementations are a huge step foreward, and allow others to create better, more efficient applications.

References

[1] Hezi Azatchi, Yossi Levanoni, Harel Paz, and Erez Petrank. An on-the-fly mark and sweep garbage collector based on sliding views. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 269–281, New York, NY, USA, 2003. ACM.

[2] D. F. Bacon, V. T. Rajan, E. Petrank, H. Paz, and E. K. Kolodner. An efficient on-the-fly cycle collection. *ACM Transactions on Programming Languages and Systems*, 29(4):1–43, 2007.

[3] David F. Bacon, Clement R. Attanasio, Han B. Lee, V. T. Rajan, and Stephen Smith. Java without the coffee breaks: a nonintrusive multiprocessor garbage collector. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 92–103, New York, NY, USA, 2001. ACM.

[4] David F. Bacon and V. T. Rajan. Concurrent cycle collection in reference counted systems. *Lecture Notes in Computer Science*, 2072:207–220, 2001.

[5] L. Peter Deutsch and Daniel G. Bobrow. An efficient, incremental, automatic garbage collector. *j-CACM*, 19(9):522–526, September 1976.

[6] Tamar Domani, Elliot K. Kolodner, Ethan Lewis, Eliot E. Salant, Katherine Barabash, Itai Lahan, Yossi Levanoni, Erez Petrank, and Igor Yanover. Implementing an on-the-fly garbage collector for java. In *ISMM*, pages 155–166, 2000.

[7] Yossi Levanoni and Erez Petrank. An on-the-fly reference-counting garbage collector for

java. *ACM Trans. Program. Lang. Syst.*, 28(1):1–69, 2006.

[8] H. Paz, D. Bacon, E. Kolodner, E. Petrank, and V. Rajan. fly cycle collection revisited. 2003.

[9] David J. Roth and David S. Wise. One-bit counts between unique and sticky. *ACM SIGPLAN Notices*, 34(3):49–56, 1999.

[10] W. R. Stoye, T. J. W. Clarke, and A. C. Norman. Some practical methods for rapid combinator reduction. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 159–166, New York, NY, USA, 1984. ACM.

[11] K. Ueda and M. Morita. Message-oriented parallel implementation of moded flat GHC. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 799–808, ICOT, Japan, 1992. Association for Computing Machinery.

[12] David S. Wise. Stop-and-copy and one-bit reference counting. *Information Processing Letters*, 46(5):243–249, 1993.

Art-based Rendering with Graftals

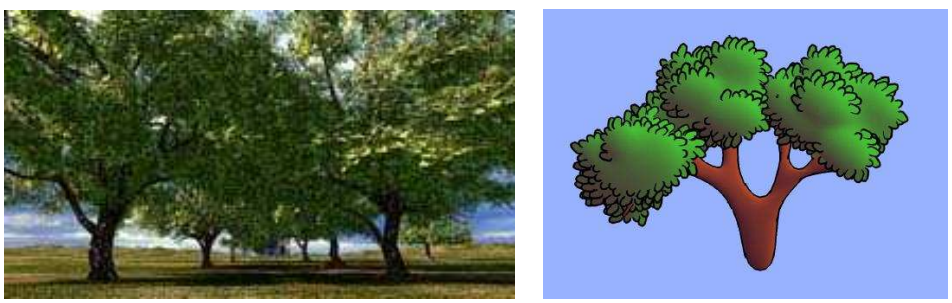
Joël van Neerbos and Watze Winsemius

University of Groningen

Abstract. An area of computer graphics is art-based rendering. The goal of art-based rendering is to create scenes like an artist would do on paper. Graftals can be very helpful for this task, especially for objects like trees, grass, and fur. In the first part of this paper we explain the concept of graftals, how this concept has been modified over time and how graftals can be used in rendering art-based scenes. In the second part of this paper we go into the challenges of rendering graftals. Several papers have come up with methods of using graftals in art-based scenes. We compare these methods with each other and draw some conclusions. Also, we give some suggestions on how these methods can be combined or extended.

1 Art-based Rendering

Photorealistic rendering is an area of computer graphics which focuses on photorealism. This in contrast with non-photorealistic rendering or, specifically, art-based rendering, which is inspired by artistic styles such as painting, drawing, and cartoons. Art-based rendering has application areas in, for example, movies and video games, but also in children's picture books. Judging by the fact that those books use hand-drawn images, it seems that art-based rendered images appeal better to children than photorealistic images. One reason for this could be that art-based rendered images are more abstracted and therefore leave more to the imagination than photorealistic rendered images, which show even the smallest details. This is probably the biggest advantage of the art-based rendering approach. An other advantage of art-based rendering is that it can reduce the cost of rendering under the right circumstances.



(a) (b)

Fig. 1. Photorealistic rendered tree (a)[1], art-based rendered tree(b)[2]

Figure 1(a) shows a photorealistic image of a tree, Figure 1(b) is art-based rendered. If both images could be rotated one can imagine that Figure 1(a) needs much more rendering time than Figure 1(b) and why use Figure 1(a) in children's books if they like Figure 1(b) better?

[1] Image from www.marlinstudios.com
[2] Image from [1]

This introduction shows a few reasons why art-based rendered images are valuable in computer graphics. In this paper we focus on rendering virtual scenes using art-based styles. For this we use graftals, which are most suitable for the rendering of fur, grass and trees. In section two we introduce graftals, first in the original form and then in the form it is used nowadays. After that, we show how graftals can be used in art-based rendering. We focus on the challenges which arise in art-based rendering with graftals and the different solutions to those challenges. We end this paper with a conclusion and some recommendations for future work.

2 Graftals

Graftals, as they are introduced by Alvy Ray Smith [4], are originally defined as parallel graph grammar languages. They are no different than L-systems [5], but specifically used to describe tree and plant models. Graftals are similar to fractals, in the sense that they are self-similar (the whole has the same shape as one or more of the parts). However, graftals are less restrictive than fractals. The L-systems, introduced by Lindenmayer [5], are similar to conventional grammars, but they differ in two aspects. Firstly, all grammar rules are applied simultaneously. Secondly, there is no distinction between terminal and non-terminal symbols, so the production rules can be applied infinitely often. An example is the L-system with the alphabet $\{0, 1, [,]\}$, the production rules $\{0 \rightarrow 1[0]1[0]0, 1 \rightarrow 11, [\rightarrow [,] \rightarrow]\}$ and the axiom (starting symbol) 0. For this grammar, the first three steps are:

1. 0
2. 1[0]1[0]0
3. 11[1[0]1[0]0]11[1[0]1[0]0]1[0]1[0]0

If such a grammar is translated to a graphical presentation, rather complex structures can emerge, as shown in Figure 2.

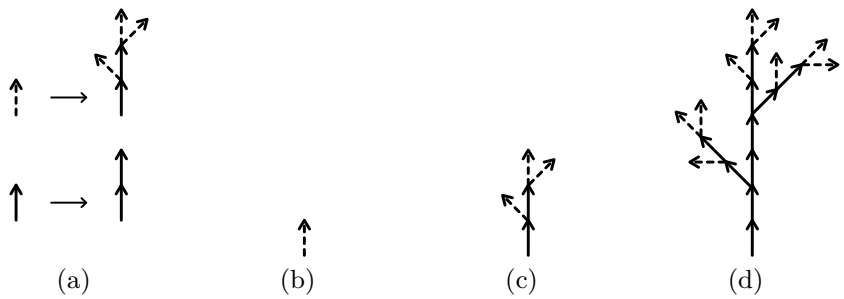


Fig. 2. Production rules (a), axiom (b) and the results after one (c) and two steps (d). Example taken from [4].

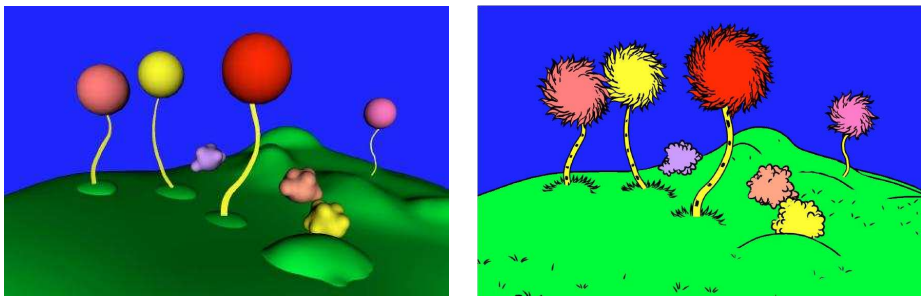
Later, graftals have come to be more generally described, for example by Badler and Glassner [6]: "Fractals and graftals create surfaces via an implicit model that produces data when requested". More specifically, the word 'graftals' is used for particles (graphical detail elements) in a three-dimensional scene that are only generated when requested. They do not necessarily need to be specifically placed on the geometry of the scene. For example, Markosian et al. [2] use graftals for giving leaves and blades of grass a cartoon-like style. When a surface should be entirely covered in grass, it may not look nice to just draw blades of grass over the entire surface. Instead, for just a few specific patches the graftals can be 'requested', covering those parts with blades of grass while the rest of the surface remains empty.

Kaplan et al. [3] introduced the concept of geograftals, which further generalizes the definition of graftals to include procedural geometric entities. The geograftals allow more precomputation of information about

the graftals, so less computations have to be performed at run-time. Both the graftals as described in the previous paragraph and the geograftals are discussed in this paper.

3 Art-based Rendering with Graftals

In this section we look into art-based rendering with graftals. The Figures 3(a) and 3(b) show the same scene, once without graftals and once with graftals.



(a) (b)

Fig. 3. Scene rendered without graftals (a)[3], the same scene rendered with graftals (b)[3].

Figure 3(b) demonstrates what the result can be of a scene rendered with graftals. In this section we focus on the challenges when rendering a scene like the one in Figure 3(b). We compare the papers [1], [2], and [3] to see what the challenges are in working with graftals and how these challenges can be dealt with. It appears that the challenges arise when navigating through the scene. The papers use different solutions for those challenges which we describe in this paper.

3.1 Graftal placement algorithm

To place the graftals in the scene, Kowalski et al. [1] used a modified version of the "difference image" algorithm by Salisbury et al. [7]. An important aspect of this algorithm is that it uses reference images: off-screen renderings of the scene. Two kinds of reference images are used: a color reference image and an ID reference image.

The color reference image is used to determine where graftals need to be placed. Every patch that is to be drawn in a specific style, should render its texture in some appropriate way to the color reference image. Which ways are appropriate depends on the style and the way that style is implemented. Kowalski et al. [1] let the "desire" (the desired density of graftals) depend on the amount of shade, so the color reference image of the scene in Figure 3 could be similar to Figure 3(a), because for most patches the amount of shade in Figure 3(a) gives a rough estimation of the graftal density in Figure 3(b).

To actually draw a scene like 3(b) from an image similar to 3(a), first a location with enough desire is determined. Then the graftal is placed and the color reference image is updated. Placing a graftal should decrease the desire in its location, so in this example the shade in that location should be decreased. This is done by subtracting a blurred version of the graftal, or usually just a Gaussian dot of roughly the same size, from the color reference image. Then a new location with high enough desire is determined and the procedure is repeated.

[3] Images from [1]

Because one of the requirements of believable rendering with graftals is that the graftals need to appear to stick to the surfaces in the scene, a 3D position of the graftal is needed. The 3D position can also be used to calculate the distance to the surface on which the graftal resides, to determine the level of detail in which the graftal needs to be rendered. To determine this position, the ID reference image is used. This reference image contains the same triangles that are rendered in the color reference image, but every triangle has a unique color. For every color the corresponding triangle is stored. So if the position of the graftal is determined, the color of that same position in the ID reference image is retrieved and from that color the corresponding triangle is identified. Because this triangle is defined in 3D space, it can be used to determine the 3D position of the graftal.

In every frame, the graftals are first attempted to be placed in the 3D positions of the preceding frame. After all graftals from the preceding frame have been attempted, new graftals will be rendered in areas with sufficient desire.

3.2 Challenges

Although the graftals seem to stick with the surface, a lot of appearance and disappearance of graftals still occurs between frames. This causes a flickering that is distracting and not aesthetically pleasing. Markosian et al. [2] discuss the challenges and propose some solutions that we discuss here.

One problem that was encountered was caused by the fact that only graftals on visible surfaces are rendered. Graftals on invisible surfaces just behind silhouettes should be partially visible in most cases. Their absence is most notable when their surface becomes visible and the graftal suddenly pops into view. An other problem was that a small number of discrete levels of detail were used with a sudden and visually distracting transition between them.

A major improvement to this was the addition of gradual transitions. Transitions between different levels of detail, but also new transitions for the appearance and disappearance of graftals. By varying the length, color and thickness of the lines over time, the transitions are not as sudden anymore and are less distractive. This has the consequence that new graftals are not yet entirely visible in their first few frames but that they grow into the scene over time.

An other new step was to introduce compound graftals, or tufts. Tufts have several levels of detail, all of which can contain a different number of graftals. In the lowest level of detail, the tuft may consist of only a single graftal, while it can contain a lot of graftals on higher levels of detail. In this sense it comes closer to the original definition of graftals. In combination with gradual transitions, it is possible to look at a surface defined by a tuft from far and then zoom in to a very detailed look, while no significant visual distractions will occur. As an additional advantage, tufts also allow for more efficiency due to their multi-resolution nature. When the tuft is to be drawn in a low level of detail, there is no need to visit the individual graftals needed in higher levels.

3.3 Geograftals

Another approach is taken by Kaplan et al. [3]. They introduce the concept of geograftals. A geograftal differs from a graftal in a sense that each geograftal is statically placed on the model's surface while graftals are placed at run-time. This placing can be done manually by selecting the specific points on the surface or randomly if the user sets the density, see Figure 4. In this way information like position and color can be precomputed as much as possible. In this section we describe some properties of geograftals.

Fig. 4. A blue leaf/fur geograftal on a surface[4]. The white dot illustrates the location point.

The goal of geograftals is, just like with graftals, to imitate hand-drawn images. For example, trees could be rendered with this technique. The effects needed are most evident near silhouette edges and therefore, geograftals near silhouette edges are drawn as large as possible. To achieve frame-to-frame coherence all geograftals are drawn each frame. This prevents distracting popping effects. This popping normally occurs when the surface where the graftal is placed at is becoming visible. By drawing all the geograftals this problem is avoided.

Another property of geograftals is the scaling factor. This scaling factor creates large geograftals near silhouettes and tiny geograftals in the interior. The problem with these tiny geograftals is that the black edges disrupt the effect of displaying information near silhouette edges. Kaplan et al. solve this problem by introducing two extra scaling functions. The first one scales the edge width, the second one scales the color of the edges so that it matches more with the color of the interior. An example can be seen in Figure 5.

Fig. 5. Leafs on a sphere which show scaling of the edges of the geograftals in the interior[4].

Kowalski et al. faced the problem that as distance varies, graftals were appearing and disappearing. Instead of removing and adding geograftals to a scene, Kaplan et al. solve this problem by using a scaling factor with distance. Only a few geograftals are drawn bigger to achieve the effect of artist who only draw a few graftals if the object is at a far distance.

In this section we have seen challenges and different solutions for this challenges when rendering art-based scenes with graftals. The next section compares the different solutions.

[4] Images from [3]

4 Results and discussion

We discussed two different ways of rendering graftals, being the graftal implementation first introduced by Kowalski et al. [1] and later improved by Markosian et al. [2], and the geograftal implementation by Kaplan et al. [3].

Because the geograftals are placed statically on a surface, no new graftals will appear when zooming in past a certain level. This property moves the graftals even further away from their original definition, which had the property that a surface looked similar independent of the zoom level. But, as mentioned in the previous section, placing graftals statically does give a significant performance boost because much information can be precomputed. The graftal implementation can generate new graftals when zooming in, for example with tufts, but does so at a higher performance cost.

A problem with both implementation methods is the appearing graftals on surfaces just behind a silhouette. With the geograftals implementation, this can be solved by just drawing all graftals, including the ones on hidden surfaces. Of course, the graftals that should not be visible at all will be drawn as well, but because they are part of the scene geometry they won't be visible in the final picture. While this solves the problem of graftals on hidden surfaces that should be partially visible, it also slows down the rendering process significantly. Drawing hidden graftals with the first graftal implementation will require some modification of the algorithm and will probably slow the rendering process down even more, but with the gradual appearance and disappearance it may not be as needed either.

The geograftal implementation uses fading and scaling for smooth appearance and disappearance, while the other graftal implementation varies the detail of the graftals, introduces or removes graftals (with tufts) and uses scaling as well. Opinions may differ, but the geograftal method will usually give less realistic and visually less pleasing results. For example, when a graftal should be drawn quite small between graftals of a normal size, drawing it with reduced detail will probably look more natural than just scaling and fading the full graftal. Also, when zoomed in a lot on a graftal, increased detail or even newly introduced graftals will probably look nicer than just a bloated version of a graftal with normal detail.

In short, while the first graftal implementation generally gives a bit more pleasing results, the geograftal implementation has better overall performance. On modern hardware, performance may not be as much an issue as it was when the methods were introduced, but demands and expectations tend to grow along with the possibilities, so performance will not lose its importance any time soon. However, increased hardware performance does make computationally more intensive methods more feasible.

In the end, we may not even have to choose between visual quality and performance, because it may be possible to combine the methods. It could very well be possible to combine statically placed tufts and gradually changing levels of detail with the concepts of geograftals, to get the best of both. It will not be trivial because the precomputation will definitely be affected, but it should be possible to combine both concepts in a useful way.

5 Future Work

We do think that graftals can be very useful in art-based rendering. It has less performance costs while the scenes produced are comparable with hand-drawn images. But there are a few disadvantages in rendering graftals. The most important one is the appearance/disappearance when navigating through a scene. Even though this has been improved over time, we still believe some research in this area can be useful.

Also, the current application areas for graftals are rather limited. In our paper we only discussed graftals for rendering trees, grass and fur. We think graftals can be useful in other application areas as well. One could study work of artists to see where graftals also can be used. An other option is to explore the possibilities of extending the styles of graftals. They now have more or less the same shape, but why shouldn't it be possible to extend those styles to render different objects? As an example we are thinking of flames like those in Figure 6. We believe it is possible to extend the styles of graftals in a way that flames can be rendered.

Fig. 6. An art-based rendered image of a fire[5]

The last proposal for future work we would like to make is combining the two methods we described in this paper. Tufts as used in [2] are useful for changing the level of detail and geograftals have the property that they avoid popping of graftals as one navigates through the scene. We think that exploring the possibilities of combining those two methods can yield remarkable results.

References

[1] M. A. Kowalski, L. Markosian, J. D. Northrup, L. Bourdev, R. Barzel, L. S. Holden and J. F. Hughes: Art-Based Rendering of Fur, Grass and Trees. Proceedings of SIGGRAPH 99 (1999) 433–438
[2] L. Markosian, B. J. Meier, M. A. Kowalski, L. S. Holden, J. D. Northrup and J. F. Hughes: Art-based Rendering with Continuous Levels of Detail Proceedings of NPAR 2000 (2000) 59–66
[3] M. Kaplan, B. Gooch and E. Cohen: Interactive Artistic Rendering Proceedings of NPAR 2000 (2000) 67–74
[4] A. R. Smith: Plants, fractals and formal languages. Proceedings of SIGGRAPH 84 (1984) 1–10
[5] A. Lindenmayer: Mathematical Models for Cellular Interactions in Development, Parts I and II Journal of Theoretical Biology 18 (1968) 280–315
[6] N. I. Badler and A. S. Glassner: 3D object modeling SIGGRAPH 97 Introduction to Computer Graphics Course Notes (1997)
[7] M. P. Salisbury, M. T. Wong, J. F. Hughes and D. H. Salesin: Orientable textures for image-based pen-and-ink illustration Proceedings of SIGGRAPH 97 (1997) 401–406

[5] Image from www.openclipart.org

High Quality Printing Of Pen-and-Ink Rendering Methods

Hedde Bosman[1] and Imco Veenstra[1]

University of Groningen, Postbus 407 9700 AK Groningen, The Netherlands

Abstract. This paper discusses the printing quality of non-photorealistic rendering methods called pen-and-ink rendering methods. The general aspects of printing quality are discussed and four different pen-and-ink methods will be reviewed with emphasis on these printing quality aspects. The focus here is on scalability to higher resolutions. Based on our findings, we present a unifying model for pen-and-ink rendering methods.

1 Introduction

Since the conception of computer graphics, the goal of this field has been to produce ever more photorealistic images.

For certain applications it turned out that photorealistic rendering is not the optimal way of depicting things. Photorealistic images are not ideal if they need to portray structure or domain-specific information. It is, for example, very hard for a layman to identify the main artery when looking at a photorealistic rendition of a hart.

Photorealism is only one aspect of an image. Realising that an image can have more aspects than just photorealism, such as structure, texture and domain-specific information, gives rise to the field of non-photorealistic rendering (NPR).

The field of non-photorealistic rendering investigates algorithms that produce non-photorealistic images. These can be purely artistic and illustrative renderings with hand-drawing quality, or any other method that is not geared to photorealism. The latter can be specialized methods which are used in medical illustration, scientific illustration and technical illustration.

This paper focuses on a specific area of NPR called pen-and-ink rendering. This area of NPR concentrates on researching and developing methods that render objects using only line and dot primitives. This research depends heavily on the traditional artistic line and dot drawing techniques from scientific and technical illustration.

Almost every computer is equipped with raster display devices nowadays, which makes it more natural to use raster representations when working with these computers. As a consequence, a huge gap exists between NPR methods and the requirements for printing, especially concerning the quality of the generated images when printed.

Therefore, we discuss the printing quality aspect of pen-and-ink rendering. First the general issues concerning printing quality will be discussed in Section 2. In Sections 3, 4, 5 and 6 four different pen-and-ink rendering styles are discussed with special attention to the printing quality of these methods. We present a unifying model for pen-and-ink rendering methods in Section 7, which should make these methods more suitable for high quality printing. Section 8 concludes this paper with suggestions for future work.

2 Hiqh Quality Output

The term high quality output has a different meaning depending on the output device. It is also a matter of taste to some degree as to what constitutes high quality. For the purpose of this paper we distinguish between methods for generating hatch lines and stippling, methods for storing these lines and dots, and methods used to render them. This distinction indicates that these three elements of a pen-and-ink rendering method also have their own levels of quality. It is, thus, not trivial to define what is meant with high quality in this context. We start with looking at the different output media involved in the process.

2.1 Output Media

Displays - CRT, LCD and projectors are the standard devices for visual feedback of a computer these days. Projectors are a special case which is ignored in this paper. Aforementioned output devices consist of a raster of pixels which output a color. Colors are defined as RGB values and can be anything from black to red, green, blue to white, in steps defined by the

video card. Pixel resolution of modern displays range from 72 to 130 Pixels Per Inch (PPI), which is quite low in contrast with 3600 Dots Per Inch (DPI) used in high end laser printers.

For a raster display device, aliasing can occur due to the rasterization process. Aliasing gives a jagged appearance to lines and borders, which degrades the visual quality of an image. The methods to counter this aliasing effect are called anti-aliasing methods. Anti-aliasing reduces jagged edges in the displayed image by using gray pixels to create visually smooth lines.

Printers - A printer is a device that produces hard copies of documents that are stored electronically.

Plotters are printers which use vector graphics for input. A pen is moved over the surface of the paper to create solid smooth lines. Printing detail is related to the pen size and the motor precision.

Most modern printers however, use rasterized methods. These include Inkjet, Laser and LED printers. The source material may be encoded in any number of special page description languages such as Adobe PostScript (PS) or HP Printer Command Language (PCL), as well as unformatted text-only data. A Raster Image Processor uses the page description language to generate a bitmap of the final page in the raster memory. Once the entire page has been rendered in raster memory, the printer is ready to begin the process of sending the rasterized stream of dots to the paper in a continuous stream. These dots can be spaced in a range from 300 to as little as 3600 DPI.

These printer types use four primary printing colors: cyan, magenta, yellow and black (CMYK). These colors are sometimes called *process colors* when the printer mixes these four colors to get the desired output color.

This means that all colors except cyan, magenta, yellow and black have to be displayed as a mix of dots with some (white) spacing, the amount of dots conveying the color tone. This technique is called half toning.

Another way for printers to apply colors is *spot colors*. Spot colors are specially prepared colors that are not mixed by the printer itself. A printer can apply one spot color in a single run. To apply three spot colors, the printer needs to do three separate print runs to apply the three colors to the medium.

2.2 Quality Of Method

When assessing the printing quality of pen-and-ink rendering methods, it is also necessary to look at the quality of the method itself. The printing quality can never be better than the quality of the generated images, so when a specific pen-and-ink method generates poor quality images, printing these images will also result in a poor quality output.

The way a pen-and-ink method generates hatches and stipples can be divided into two types: object-space and image-space. This is of influence on how such a method stores the images. Hatched and stippled renderings can be stored in vector or raster formats. The former needs to be rasterized at some point to be displayed on screen or in print, unless a plotter is used.

Image-space methods usually make use of the traditional 3D rendering pipeline and perform operations on buffers containing pixels. The results of these methods are also stored as pixel images. Object-space methods, on the contrary, use the three-dimensional descriptions of objects to generate line or stipple information. These types of methods lend themselves more to vector representations.

There are, of course, exceptions to these rules. Object-space methods could generate vector information while processing a buffer of pixels, but this is not supported on graphics hardware and is done on the CPU[1].

2.3 General Quality

Pixel images have some inherent problems. Every picture element is stored as an explicit color and an implicit coordinate, usually in a two-dimensional array. This means that the size of a pixel image will increase when the desired resolution of the image increases. The image size is *output sensitive*, thus, the size increases with the desired resolution and does not depends on what is being depicted [2].

When such an image is used with another size or resolution it must be resampled or scaled. Scaling and resampling are the same thing in this context. Scaling gives unwanted aliasing effects because of the pixel replication or deletion.

Zooming is another problem with pixel images. The discrete nature of a pixel image becomes apparent very quickly when zooming in. This is not desired

[1] With the new Shadermodel 4 graphics hardware this could become possible.

[2] Some pixel image formats allow for indexed color palettes. This means that an image with two colors will be smaller than an image with 256 colors.

behavior when we want to print an image with part of that image zoomed in for greater detail.

A solution to these problems is storing an image for a series of resolutions, a technique called mipmapping. This obviously costs even more in terms of storage. And there is another problem.

Depending on the output resolution, type of printer used and various other factors, printers sometimes need to resample images internally to be able to obtain the desired output image. These effects should be accounted for when maintaining various resolutions per image, which is extremely hard because this means a pen-and-ink rendering method should consider every printer that could reasonably be used for printing its images.

Vector graphics do not have these problems. Vector representations can be sent to the printer, where the printer driver will convert the image to whatever format is optimal for the print job at hand.

Other advantages of vector graphics are the fact that scaling does not cause unwanted artifacts like aliasing. This means that one vector representation can be used for every resolution imaginable.

Zooming is also no problem for vector images, as the analytic nature of vector representations makes it possible to show every detail present in an image, even when zoomed in heavily, without loosing any resolution.

The image size of vector images is dependent on the number of objects represented. This means that vector graphics can be bigger for smaller images sizes in comparison with pixel images, but they tend to become smaller in comparison with pixel images when the resolution is bigger. This turn around point differs per image, but for resolutions used in high quality printing vector graphics are usually smaller than pixel images.

There are some problems with scaling vector images, as mentioned by Salisbury et al. [SA96]. When simple scaling is used, the same amount of lines or dots will be used to shade the same relative area of an image, which will result in perceiving a lighter tone overall. This means that more lines or dots must be added when the image is scaled to obtain the same perceived tone as the original image.

A similar problem is the fact that line and dot placing influence the perceived tone. When these lines or dots are generated in object-space, curved areas will have different perceived tones due to perspective mapping.

Salisbury et al. [SA96] present an image based solution to these problems by maintaining a gray-scale image per used texture and per used texture a link to a stroke texture [WS94]. These stroke textures are used together with the gray-scale images to recreate the image at the desired resolution in a process called *blasting*. This means that the strokes from the stroke textures are applied to the gray-scale images to get the same tone as the gray-scale image. This solution is a hybrid between pixel based images and vector based stroke textures.

3 Stroke Textures

A pen-and-ink illustration creates the appearance of surface texture by applying strokes in a certain pattern to the surface. Cross-hatches can express different tones as well as textures depending on their use. Winkenbach and Salesin [WS94] have devised a way to convey these patterns in so called *stroke textures*.

Stroke textures are collections of strokes described in an analytical way. In a sense it could be called a vector representation of a texture containing stroke information. The strokes in these collections are prioritized to be able to express texture. Starting with the highest priority, the strokes are drawn until a proper tone is achieved. This prioritization is different for every stroke texture. The path of a stroke is modified with a waviness function, to give character to the stroke as well as indicating material properties. Wood, for instance, is wavier than glass.

Outlines do not only indicate the boundary of objects, but they can also express texture and even shadows. To that extent each stroke texture has a boundary outline texture, which is applied when the outline of a polygon is rendered, but only if the shades of different regions are not sufficiently different. This minimizes outlines.

The advantage of using stroke textures is that the system scales the stroke textures automatically and chooses the right prioritized stroke from a stroke texture depending on the scale and resolution of the output needed. This prevents aliasing artifacts and also brightness change since the amount of strokes can be increased with increased scale. In other words, this method scales well to different resolutions for printing.

This method however is tailored for use with architectural models only; it can not be directly applied to other types of objects such as plants and animals.

Another major disadvantage is that only a certain amount of stroke textures can be devised beforehand expressing different materials. For materials that have no stroke texture yet, the user must define his own.

4 Illustrating smooth surfaces

Hertzmann and Zorin [HZ00] introduce an object-space method for generating line drawings from triangle meshes that represent smooth surfaces. Their algorithm consists of three parts: compute a direction field in object space on the object surface, generate silhouette curves and apply hatch lines.

Smooth surfaces are most notably curved. This curvature can be described by the two *principal curvatures* at a point $\mathbf{p}$ that are the minimal and maximal curvatures of all the curves on the surface going through that point. From these principal curvatures principal directions can be obtained, being tangent to the curves which belong to the principal curvatures.

The principal directions are always orthogonal and lie in the tangent plane to the surface. Principal curvatures and principal curvature directions locally define the best approximating quadratic surface, so these are used to generate outlines for silhouettes and to generate direction fields used for hatching.

The direction fields are calculated by first smoothing the object if necessary. Then the field is initialized at points where the surface is sufficiently curved using principal curvature directions. And finally the field is calculated on the points on the surface with low curvature using the information from the second step.

Curvature information is also used to compute the outlines of the smooth surfaces. These outlines are the points of a surface where the normal of the surface is orthogonal to the viewing direction, $(\mathbf{n}(\mathbf{p}) \cdot (\mathbf{p} - \mathbf{c})) = 0$. Here $\mathbf{n}()$ is the normal, $\mathbf{p}$ the point on the surface and $\mathbf{c}$ the camera position.

Boundary outlines pose no problems, but interior outlines may contain singularities. Hertzmann and Zorin [HZ00] name these outline singularities *cusps*, see Figure 1.

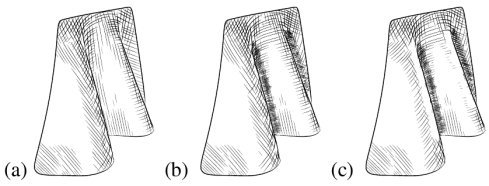


FIGURE 1: (a) Cusps (b) With undercuts (c) With Mach bands. [HZ00]

These cusps are detected and used to determine the proper outlines including the cusps.

Finally the hatch lines are generated. Stepping along boundary and silhouette outlines Mach bands and undercuts are detected by a ray test near each curve point. Both areas are marked in a 2D grid. Cells in which a Mach band occurs are not to be hatched, and undercuts are to be hatched extra dense. Then, all regions designated for single and cross hatches are cross hatched, after which in the single hatch regions cross hatches are removed.

The 2D grid suggests rasterized processing of the hatches, which would probably show in high resolution images. The algorithms, however, are suited for vector processing, allowing for rasterizing at different scales. The use of Mach bands and undercuts increase contrast where surfaces overlap, aiding in the interpretation of the surface.

The use of direction fields and their principal curvatures is a good way to generate hatches in smooth corners and other non-linear surfaces, as opposed to the stroke texture mapping of Winkenbach et al. [WS94]

5 Streaming through vector fields

Similar to the method of Hertzmann and Zorin [HZ00], Zander et al. [ZA04] also start off with calculating principal direction fields. However, this direction field is processed further to enhance its quality. Cross hatching can be achieved by rotating the resulting lines. This allows not only orthogonal hatching but hatching arbitrary angles.

Streamlines can be computed by integrating the direction vector field, obtained from a smooth direction field, on the model surface. To prevent two streamlines from crossing, a space around a streamline is preserved in the form of a tube that no other streamline may penetrate. However, the faces of these tubes are flat and end at the streamline end. This allows other streamlines to come closer to the endpoint and reduce gaps between two line ends.

Hatching lines can be generated from the streamlines in 3D space. To accomplish this, a Hidden Line Removal (HLR) algorithm is used from Isenberg et al. [IS02] together with removing all strokes on back faces. This ensures that occluded lines are not visible. Next, a NPR line shader is used to produce smooth transitions from fully drawn to transparency according to the lighting. Using byte code and a virtual machine, the shader model can be adapted by the user. This al-

lows one not only to adjust parameters but also create other lighting effects.

Shaded lines conflict with the goal of monochrome lines, thus a form of one-dimensional stippling is used. This stippling is used to change the tone of a line and does have the nice property of being able to generate a cross-over between hatching and stippling.

All computations thus far have been done in object-space and the resulting hatch strokes have to be projected to image-space. This introduces a drawback of this method because tone and shading are not only conveyed by the amount of strokes in an area, but also by the stroke density. Strokes on surfaces not orthogonal to the viewing direction appear closer together from the viewing direction after projection. To counter this effect the line width is adapted using a correction factor.

This method being an object-space method generating streamlines as vectors makes it very suitable for high quality printing.

6 Weighted Voronoi stippling

Stippling is the technique of placing small dots of ink onto paper such that their density gives the impression of tone. One of the features of a good stipple drawing according to Secord [SE02] is that the stipples are well-spaced, that is, the stipples do not clump together, leave uneven voids or form unwanted patterns. The artist achieves this by carefully placing each stipple onto the page, explaining why stipple drawings often take weeks to create by hand.

Secord [SE02] starts off with placing stipples roughly using a dithering algorithm. Using the concept of Centroidal Voronoi Diagrams the generated stipples are relaxed to form an evenly distributed set of stipples. In a Centroidal Voronoi Diagram each generating point lies exactly on the centroid of its Voronoi region. The Voronoi diagrams are calculated using graphics hardware by placing cones on the generating points and looking at the cones from above. Using the z-buffer tests the creation of Voronoi regions becomes quite fast.

However, using this method implies using a rasterized representation. This is noted by Secord [SE02], also stating that the relative error of the calculated centroid increases as the number of pixels per Voronoi region decreases. With very low resolution two generating points might even overlap. The solution is to compute the Voronoi diagram in tiles and 'stitch' these together to create a high resolution diagram.

This gives good results as can be seen in Figure 2. However, for sufficient printing quality, the image must be precomputed in very high resolution. Hardware / memory limitations can limit this resolution substantially.

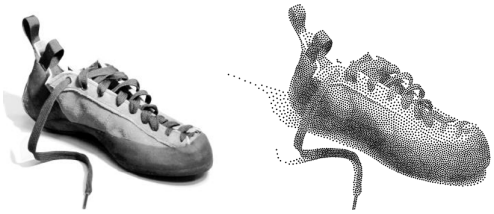


FIGURE 2: Variations in line width and tone. [SE02]

Another suggested way to render stippled drawings is to precompute stippling textures and map these in an image according to the needed tone. This allows fast real time rendering of stippled drawings but it has poor quality due to artifacts occurring because different textures do not tile together well. This introduces very visible voids, clumps of stipples and patterns. Thus this method can only be used in previewing.

7 Towards a unifying model

The methods discussed above all aim at producing pen-and-ink illustrations. Each paper addresses some aspect of high quality printing. To this end, a good standard model might improve overall quality.

A unifying NPR model would be suited to output images in different viewing formats. We suggest a model based on the Model-View-Controller concept [KP88]. The underlying representation of the model needs to contain all information to (re)create the image in different scales without the loss of features or changes in tone, or other errors that might occur with scaling. The different views might be printers, plotters or computer screens (with for instance anti-aliasing).

Strokes should have a set of properties that describe how the output is generated. These properties might include

Priority like the stroke priority mentioned in the paper of Winkenbach et al. [WS94]. Rendering higher scales might then include extra information (strokes) with lower priority.

Line Style might indicate thickness and waviness as mention by Winkenbach et al. [WS94].

Line End lets one describe the shape of the end of a line, to be rounded or taggered as suggested by Zander et al. [ZA04]

General direction to be used, for instance, with stippling to indicate a general direction in which the dots are placed. Non-uniform dots can indicate a certain pen direction or marks left by the pen imprint to add realism. This direction can be inferred by direction fields as described by [HZ00]

Tone/color to introduce the stippling of a stroke path when the tone of a line is not just binary (black or white) like described by Zander et al. [ZA04]

These are only a few of the possible properties, and thus the representation of the model should allow for extension.

Existing pixel-based pen-and-ink methods need a custom post-processing step to generate output for our suggested model, one of the requirements being that the strokes or stipples should be stored as vectors. Salisbury et al. [SA96] describe a reconstruction method that magnifies a low-resolution image that keeps the resulting image sharp along discontinuities. A technique like this can be used to describe the output of existing pixel-based methods in a vector representation.

New techniques can output directly to a format for the model. Output format and the level-of-detail contained in the model determine the printing quality in our proposed model.

Today, graphics hardware can play a useful part not only in the generation of the data for the model, but also in the model itself. The latest GPUs have Vertex and Geometry shaders, which present a lot of possibilities to manipulate vectors and vector graphics in hardware.

8 Conclusion / Future work

The field of NPR, with pen-and-ink rendering in particular, is a rich field with widely varying approaches. We have discussed several methods that add innovative ideas to increase quality. Be it stroke textures, the generation of hatches with the help of principal direction fields or streamlines generated in the principal direction fields, placing evenly spaced dots or increasing contrast around surface features like cusps, all these methods are eventually outputted on a device using a raster. We suggest a model in which all of these methods of hatching and stippling might fit. This model in turn takes care of the different aspects encountered with the output on different media, especially making all the different pen-and-ink methods suitable for high quality printing.

Although this model is a step toward unifying the creation of realistic pen-and-ink methods, a lot more can be done, as a study of Isenberg et al. [IS06] shows. A good assessment of the use and results of NPR with end users can also increase quality by targeting aspects that are of importance to the targeted audience.

References

WS94. Winkenbach, G. A., Salesin, D. H.: Computer-Generated Pen-and-Ink Illustration.
Proceedings of ACM SIGGRAPH 94 pages 91–100, New York, 1994. ACM Press

HZ00. Hertzmann, A., Zorin, D.: Illustrating Smooth Surfaces.
Proceedings of ACM SIGGRAPH 2000 pages 517-526, New York, 2000. ACM Press.

SE02. Secord, A.: Weighted Voronoi Stippling.
Proceedings of the Second International Symposium on Non-Photorealistic Animation and Rendering pages 37–44, New York, 2002. ACM Press.

ZA04. Zander, J., Isenberg, T., Schlechtweg, S., Strothotte, T.: High Quality Hatching.
Computer Graphics Forum, 23(3) 421–430, September 2004.

SA96. Salisbury, M., Anderson, C., Lischinski, D., Salesin, D.H.: Scale-Dependent Reproduction of Pen-and-Ink Illustrations
SIGGRAPH 96 Conference Proceedings pages 461–468, 1996. ACM Press.

IS02. Isenberg, T., Halper, N., Strothotte, T.: Stylizing SIlhouettes at Interactive Rates: From Silhouette Edges to Silhouette Strokes.
Computer Graphics Form 21(3) 249–258, September 2002.

IS06. Isenberg, T., Neumann, P., Carpendale, S., Sousa, M.C., Jorge, J.A.: Non-Photorealistic Rendering in Context: An Observational Study
Proceedings of the Fourth International Symposium on Non-Photorealistic Animation and Rendering pages 115–126, 2006. ACM Press.

KP88. Krasner, G.E., Pope, S.T.: A cookbook for using the model view controller user interface paradigm in Smalltalk-80.
Journal of Object-Oriented Programming pages 26–49, 1988.

Non-Photorealistic Expressive Modeling and Animation

Jaap Bresser (1567160)
Nico de Poel (1277219)

Abstract

While most non-photorealistic computer rendering (NPR) techniques succeed in emulating different artistic and illustrative drawing techniques, they generally fail to infuse their depictions with the same character and expressiveness that are so typical for real man-made drawings. We look into the reason why traditional NPR techniques are lacking in this respect and why modeling and animation play a part in the solution. We also discuss several solutions involving expressive modeling and animation that have already been tried and tested.

1 Introduction

For a long time, computer graphics research has been largely aimed at producing images that are increasingly close to photorealism. A different branch of research is involved in developing techniques that actually go in the other direction, creating computer renderings that are ever farther away from reality and more closely resemble artistic and illustrative drawings. This type of computer rendering is called non-photorealistic rendering (NPR).

One specific research topic in the field of NPR involves giving computer generated imagery the same expressiveness and character that can be found in cartoon drawings made by human artists. With the large amount of computer animated cartoons released today, this is an issue that is faced regularly by many computer graphics artists.

Most non-photorealistic graphics research tends to focus on different ways in which an object can be drawn, and it is indeed possible to add a layer of expressiveness to a computer rendering by using specialized techniques. The loose line drawing technique described by Sousa and Prusinkiewicz [Sou03] is a good example of this. Such techniques, however, are still inherently limited by the amount of geometric information that is available in a 3D model. It makes sense to go further than just applying rendering techniques and to involve modeling and animation techniques as well in trying to bring a character alive.

In this paper, we look at the problems that are involved in giving computer graphics the same expressiveness as drawings made by illustrators, and we discuss several methods that have been employed to solve these problems.

2 Methods

A study that observed the human perception of non-photorealistic images [Isen06] has shown that the impression that an object was drawn by an illustrator does not only come from the way in which it is rendered. One of the major factors that separates man-made drawings from computer renderings is the subtle deviations from the original shape that man-made drawings have. This difference is clearly visible in Figure 1, showing a pen-and-ink drawing of a tropical pitcher plant,

with the drawing from an illustrator on the left and the computer rendering on the right. The professional illustrator used his knowledge of the object in question to add details that were not present in the original model. It is these deviations from reality that add a human touch to these drawings and give them their expressive nature.

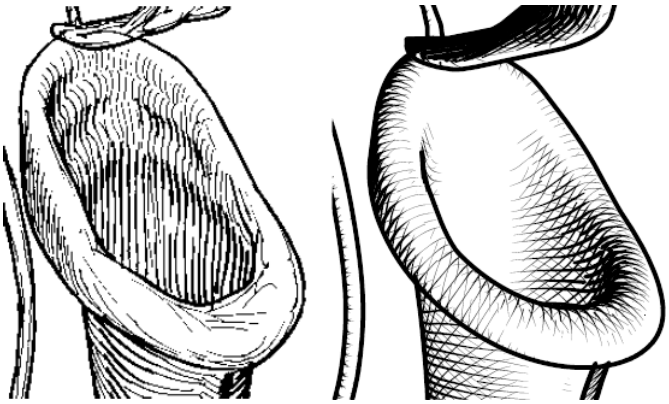


Figure 1: Man-made drawing vs. a computer rendering [Isen06]

An important conclusion drawn by this study [Isen06] is that computer-generated images lack character and expressiveness because they follow the original model too closely. In order to add expressiveness to an image, computerized image generators will have to be able to diverge from the rigid shape of a 3D model. In other words, NPR research should be concerned with modeling and animation aspects as well, instead of just focusing on rendering techniques.

These concepts are reflected in the different methods that are described below. They all center around the deformation of a model in such a way that it receives an extra layer of expressiveness not found in the original geometry, but they strongly differ in their approach. One of the key differences that we focus on is the amount of freedom these methods offer to artists for adding their own knowledge and judgement to influence the results.

2.1 Silhouette shaping and articulation

One area where non-photorealistic rendering, animation and modeling techniques are actually used in practice is at movie animation studios such as Pixar. Because many animation movies tend to focus on unorthodox situations with unlikely heroes, animators are often faced with the challenge of making an otherwise unlikeable character appealing to audiences.

One example is Pixar's recent movie Ratatouille, in which a rat plays the part of the hero and therefore needs to win the audiences' hearts, despite the fact that rats are usually portrayed negatively in our society. In one of their articles, Konishi and Venturini describe the steps they have taken to give animated rats appeal [Kon07].

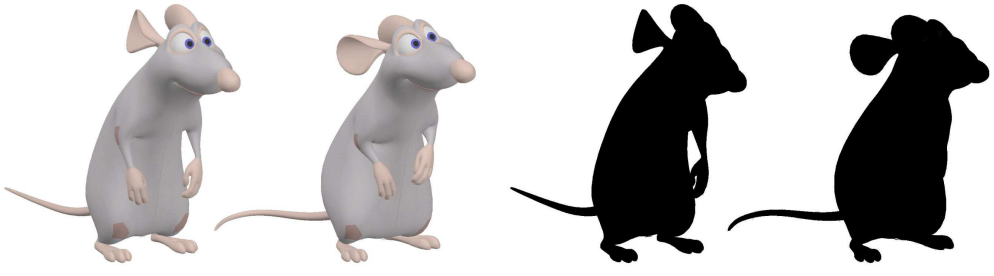


Figure 2: Complex vs. simple silhouette of a rat [Kon07]

Figure 2 demonstrates the effect of a silhouette on the appeal of a model. The left silhouette is more realistic, but it looks rough and clumsy. By reducing the silhouette back to a simple teardrop shape, as shown by the right silhouette, a soft appeal is created that is easier on the eyes.

Konishi and Venturini go on to describe more subtle effects for adding appeal. Lip movements influence the appearance of the eyes through deformation of the cheeks, mimicking the complex muscle structure of a human face and thus making a rat face easier to relate to. Teeth are kept framed within the lips to avoid exposing a gum-line, which would otherwise give the character an aggressive appearance. A rat's ears and the way they are positioned are also helpful tools in giving a character expression. The latter is visible in Figure 2, where the lower position of the ears in the right silhouette gives the character a friendlier appearance.

2.2 Caricaturization

One way of adding expressiveness images is to exaggerate certain trademark features of the subject being portrayed. Caricaturization is an example of where this technique is used. In caricatures, distinguishing features are exaggerated, while irrelevant features are thrown out to create an instantly recognizable image of a person or object. One of the challenges with the creation of caricatures is identifying which features are distinguishing and which are irrelevant.

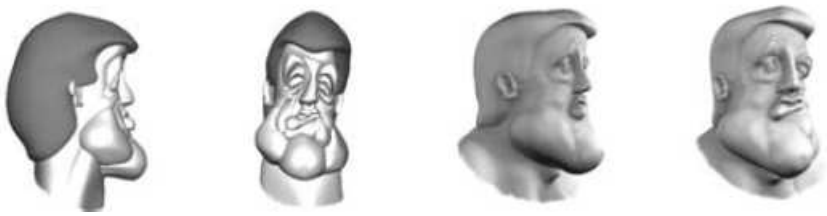


Figure 3: Caricaturized model of Sylvester Stallone [Akle04]

This technique of feature exaggeration can be applied to 3D models. Akleman and Reisch have described a step-by-step methodology for the creation of recognizable caricatures in the form of 3D models [Akle04], which they have applied for a course in 3D computer art and design. By employing an iterative method, students would identify a person's unique features, exaggerate these features one at a time, and verify with each step whether or not this improved the likeness of the model. This has resulted in models such as shown in Figure 3, which is a clearly recognizable caricature of Sylvester Stallone.

Much as the silhouette shaping and articulation techniques used by Pixar, the caricaturization technique developed by Akleman and Reisch is a purely artistic one. As such, it does not depend on any specific tools or algorithmic support. Feature identification and evaluation of the resulting caricature are all done by hand and eye, which means that a lot of input is required from the artist working on the models.

2.3 Collage assembly

A completely different approach in adding expressiveness to an object is creating compound representations of a model (also known as a collage), consisting of smaller elements or shapes. These smaller elements that make up the collage are typically related to the object the collage represents. For example, a collage of a seahorse could be all made up out of seashells. The key challenge in creating a collage is that both the original shape and the smaller parts should still be recognizable in the end result.

Collages have been used for centuries as an artistic device for adding a layer of expressiveness to drawings or paintings. One famous example is the painter Giuseppe Arcimboldo (1527-1593), whose surrealistic paintings of human faces made up of vegetables and fruits showed that people are capable of perceiving both shape and contents separately.

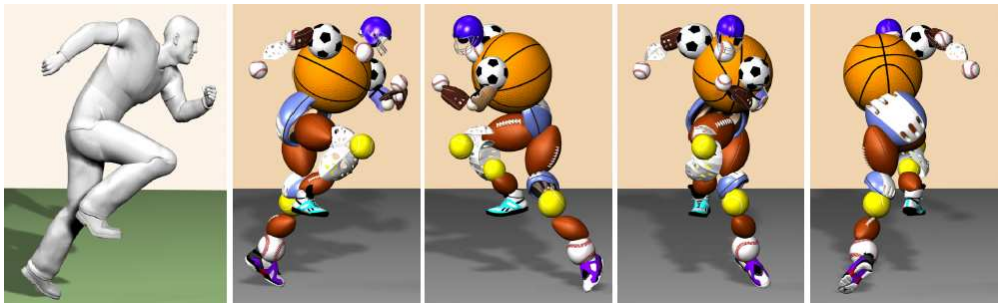


Figure 4: Collage of a running sportsman [Gal07]

Ran Gal et al. have researched a method [Gal07] to make collages of 3D models by approximating them using a collection of 3D shapes that can be selected from a database. Their research has resulted in a collage assembly framework that has been used to generate images such as Figure 4. It employs local fitting and partial shape matching algorithms to calculate the best fit for a model, using a predefined set of parameters.

After specifying the set of elements to use in collage construction, the collage assembly framework is capable of fully automatically constructing complete 3D collages of a given target shape. However, as demonstrated by the various examples described in their article [Gal07], the initial results produced by the assembly framework are rarely satisfactory. Although the tool is able to construct complex compound models that would be practically impossible to create manually, it is incapable of identifying key features in the original model, and consequently chooses elements without paying respect to these features. This results in an overall loss of recognizability of the target shape. To compensate for this shortcoming, the artist working with the assembly framework is able to provide feedback and make manual changes to the various stages of the collage assembly.

2.4 Implying and stylizing motion

The above methods have all been concerned with changing the static appearance of a 3D model. However, expressiveness can also be influenced by the way an object animates and how that can affect its look. Paul Noble and Wen Tang have taken this approach in their article [Noble07]. They have investigated the techniques that real-life cartoonists use to imply motion in still images, such as speedlines, after-images and subtle bending of otherwise rigid objects.

The modeling technique described by Noble and Tang's paper involves using animation data to bend an object according to its movement. The strength of the bend depends on the velocity with which that object moves. This effect plays into the way the human eye reacts to motion. One well known example is the "rubber pencil illusion", where a rigid pencil will appear to bend when it is quickly moved up and down. Figure 5 demonstrates the effect in a cartoon image. By bending the tennis racket in the opposite direction of its movement, it is easy to imagine that the racket moves at a high velocity, despite the fact that the image is static.

To achieve this effect, Noble and Tang use the animation data from the 3D model's bone structure to determine in what direction a limb moves and how fast. This information is applied to a combination of lattice and non-linear bend deformers to bend the limb in the opposite direction of its movement. The magnitude of the bend is directly influenced by the velocity of the movement. Smoothing of the animation over a longer time frame is applied to remove any 'popping' of the

Figure 5: Different visual cues for implied motion [Noble07]

bones caused by transitions in animation.

This research [Noble07] has resulted in a tool for Autodesk Maya that performs these deformations, allowing artists to specify a set of parameters that influence the strength of the bending effect. It is a relatively simple yet elegant solution for adding cartoon-like expressiveness to computer generated images. As Figure 5 shows, there are more types of visual cues that help to imply motion, and the motion data extracted for Noble and Tang's technique could very well be reused to implement additional effects. Combining the bending effect with a speedline rendering technique such as described by Masuch et al. [Mas99] would result in even more convincing implied motion.

3 Conclusion

As we have seen, there are many different approaches to the common problem of adding character and expressiveness to computer generated images. Solutions range from stylized shape deformations [Akle04], [Kon07] to creating surrealistic approximations of objects [Gal07], to employing techniques used by cartoonists [Noble07]. Some techniques consist merely of a set of artists' guidelines, while other techniques have been implemented into fully automatized computer applications.

A common factor in these solutions, however, is that they all show that adding expressiveness to non-photorealistic images goes further than simply drawing it in a different style. Much like the evaluation of NPR techniques has shown [Isen06], non-photorealistic changes in the shape of a model can play an important part in moving the feel of computer generated imagery closer to that of an artist's impression.

While the different solutions require different levels of user interaction, they all depend one way or another on the influence of an artist to improve the quality of their results. Nevertheless, the methods described in this article only scratch the surface of what is possible in the field of non-photorealistic modeling and animation. It might therefore be possible that researchers manage to develop modeling and animation techniques that can automatically create expressiveness without the intervention of an artist. For now though, the conclusion is that the intuition and expertise of an artist remain just as important as the tools that will aid them.

References

[Akle04] Ergun Akleman, Jon Reisch. *Modeling Expressive 3D Caricatures*. In ACM SIGGRAPH 2004 Conference Abstracts and Applications, New York, 2004. ACM Press.

[Gal07] Ran Gal, Olga Sorkine, Tiberiu Popa, Alla Sheffer, and Daniel Cohen-Or. *3D Collage: Expressive Non-Realistic Modeling*. Proceedings of the Fifth International Symposium on Non-Photorealistic Animation and Rendering (NPAR 2007, San Diego, California, USA, August 4-5, 2007), pages 7–14, New York, 2007. ACM Press.

[Isen06] Tobias Isenberg, Petra Neumann, Sheelagh Carpendale, Mario Costa Sousa, and Joaquim A. Jorge. *Non-Photorealistic Rendering in Context: An Observational Study*. Proceedings of the Fourth International Symposium on Non-Photorealistic Animation and Rendering (NPAR 2006, Annecy, France, June 5-7, 2006), pages 115–126, New York, 2006. ACM Press.

[Kon07] Sonoko Konishi, Michael Venturini. *Articulating the Appeal*. Pixar Technical Memo #07-12. Images are ©Disney / Pixar. All rights reserved.

[Mas99] Masuch, M., Schlechtweg, S., and Schulz, R. *Speedlines, depicting motion in motionless pictures*. In: SIGGRAPH'99 Conference Abstracts and Applications. S. 277. ACM, New York, 1999.

[Noble07] Paul Noble, Wen Tang. *Automatic Expressive Deformations for Implying and Stylizing Motion*. The Visual Computer, 23(7):523–533, July 2007.

[Sou03] M. C. Sousa and P. Prusinkiewicz *A few good lines: Suggestive drawing of 3D models*. Computer Graphics Forum, 22(3), pages 381–390, 2003.

The User-friendliness of NPR Interfaces

Tim Havinga s1457489 t.s.havinga@student.rug.nl
Jasper Hafkenscheid s1650173 j.m.hafkenscheid@student.rug.nl

University of Groningen

Abstract. Several methods have been developed to improve on the current practice of applying artistic or illustrative effects to images. These methods are called *non-photorealistic rendering* (NPR) methods. They apply NPR to images, or create NPR images from the ground up, using a more advanced and user-interactive approach. Most of them have new and innovative ways to influence the rendering process, and use non-standard input devices.

Our research extends to some of the most frequently used NPR algorithms available today, from RenderBots swarming the image to Surface Drawing in 3D. We consider these from the viewpoint of ordinary users, who want to adapt the image, and not tweak lots of parameters and re-render countless times. In other words, we discuss what method gives the best results with the least amount of effort.

Methods are discussed and judged based on their usability, interactivity, and the obtained results. Therefore, user interaction, hardware requirements, and user feedback incorporation are compared. The future of NPR and the interfaces used in that field of work are also examined, to see what user interfaces perform best for working with NPR. The user interface should be easy to learn, and intuitive, easy to use. Furthermore, as keyboard and mouse are not adequate for capturing painting movements, alternative methods of interaction are investigated.

Keywords: Non-photorealistic rendering, interactive, interface, user-friendliness.

1 Introduction

In this paper, four approaches are discussed and evaluated that apply a *non-photorealistic rendering* (NPR) style to an image, or render NPR images. This means that images do not need to look realistic, but rather artistic. In our research we looked at the methods that are used to apply such algorithms. They are mostly targeted at ordinary people, who like a simple user interface.
For artistic uses, parameter tweaking is not that important, but for a photo or a 3D-model it is. Art is more about the style of drawing and the freedom to create complicated shapes quickly. These 'artists' want to just start drawing, without having to know every setting in advance.

We now give an introduction to each method, to illustrate the general idea behind them.

Modeling with Rendering Primitives: An Interactive Non-Photorealistic Canvas [1, 2] is about drawing primitives on a large display with a touch screen. The concept is that all primitives are stored individually: this enables the user to change them, in contrast to most

programs that allow no interaction with the strokes as soon as they are drawn. The interface is based on a canvas with a palette to select tools and colours.
Figure 1 shows a 'painting' made with the Rendering Primitives implementation, featuring multiple layers of strokes. The colour is defined by a picture of a sunset.

Fig. 1. A drawing of a sunset with Rendering Primitives that clearly shows the strokes. Taken from [1].

Surface Drawing: Creating Organic 3D Shapes with the Hand and Tangible Tools [3] is about creating 3D models with hand motions. The interface used is the *Responsive Workbench*, that can capture the movement of the hand, and converts it to a 3D location. These locations are combined with the hand posture to generate the 3D strokes. With tools the user can further adjust the shape.
Figure 2 shows some examples of drawings created with the Surface Drawing technique. (Note that these shapes are actually 3D.)

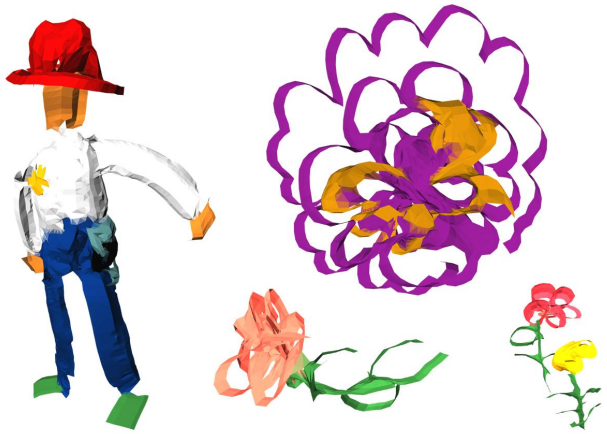


Fig. 2. Several sample drawings made with the Surface Drawing technique. Taken from [3].

RenderBots: Multi-Agent Systems for Direct Image Generation [4] generates NPR images or styles, that resemble much of the methods available today for rendering of artistic images or styles. The difference is in the way these styles are applied: RenderBots can be seen as robots, or agents, that are placed by hand and then swarm the image. There are a number of different types of RenderBots. Some draw lines or strokes, others are points or mosaic tiles. Every RenderBot can multiply itself, draw and delete himself.
Figure 3 shows an example image, created with the RenderBots method. It contains different drawing styles: stippling, hatching, painting and mosaics.

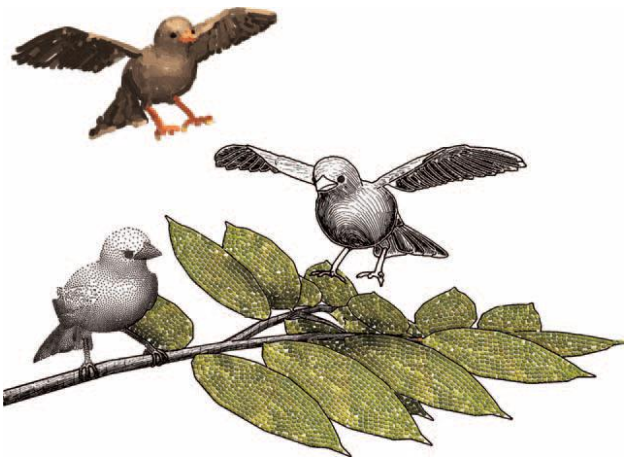


Fig. 3. A bird scene created with RenderBots, featuring different drawing styles. Taken from [4].

WYSIWYG[1] *NPR: Drawing strokes Directly on 3D Models* [5] draws lines on 3D models to create a 2D scene. The program offers a lot of possibilities to change line styles and types, and also the background. These strokes are

[1] WYSIWYG is a common abbreviation for "What You See Is What You Get".

replicated across the line.
There is also a possibility to draw hatching strokes (to mimic shade), and decal strokes. These strokes can be painted individually.
Figure 4 shows a rendered image of a simple landscape with two cones (the trees), three spheres (the snowman) and a flower. There are creases at the base of the snowman, the snowman itself features hatching strokes and decal strokes for the face.

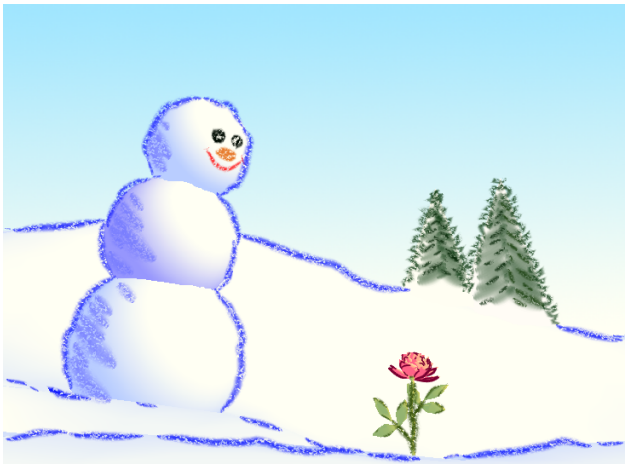


Fig. 4. A snow scene created with the WYSIWYG NPR program, an example of a simple model that gives nice results. Taken from [5].

2 Discussion

This section gives more detail about the four methods that we reviewed. They are discussed, especially as regards user interaction, hardware requirements and user feedback.

2.1 Rendering primitives

The concept The program of this method is meant to run using the canvas concept. This means that the only interaction method is by hand, and the application is full screen. The tools and colours can be chosen by using the palette. It allows both hands to be used (at the same time), one for drawing and the other for selecting tools and colours.
In later versions of the implementation the Wiimote (remote control of the Nintendo Wii gaming console) can be used, together with a set of pens. The screen can detect the size of the object used to touch the screen, i.e. a pen, one or multiple fingers, a fist or a flat hand. The palette, which was meant to make the interaction feel like painting, is replaced by a more modern, stylish menu.

User interaction The user-interface is user-friendly after a short familiarization with its meaning. It uses symbols that represent the settings that can be altered. These symbols are relatively intuitive, and easy to interpret.

Feedback The developers used colleagues and four professional artists to evaluate the implementation of their concept. [1] The feedback of the reviewers was that although the interface is made to resemble a canvas, it did not feel like painting. However, they were pleased with the effects that can be created with it. The artists missed the option to create their own primitives. This was added to a newer version of the implementation. They needed some time to get used to the concept, but were able to quickly create images with it.
During the process of development the feedback from the users was directly used to adapt the implementation.

Hardware requirements The software behind this interactive canvas require a lot of memory, because all primitives, colours, sizes and directions need to be stored. On a standard monitor this is not a problem with up-to-date hardware. For large resolutions however, a high-end videocard and a reasonable amount of memory is required.

Judgement We think the user interaction can be improved. The programmers have already adapted their interface and interaction

with user comments, but it could be even more intuitive.
Because the screen can detect multiple fingers, you could let the distance between them indicate the amount of strokes to be drawn or the stroke width.
Also, the placing of the strokes is a peculiar process, obligating the user to fill the canvas themselves. For example, when using a picture for colours, a blurred version of this picture could be used as background.

2.2 Surface drawing

The concept The main tool to create shapes is a glove that can be tracked, with a thumb-switch attached to it. Drawing begins when the thumb is pressed against the hand.
Other tangible tools have been added to modify the created shapes. They can be tracked on their own, or have to be used with the glove. The eraser enables removing parts of a shape, creating holes or removing it entirely. The magnet deforms the shape, stretching it into the direction of the hand. The last tool is a pair of kitchen-tongs, that allow the user to reposition, rotate and resize the model. The strokes that are drawn are always the size of the hand or a finger: small details can be added by first enlarging the object. For colour selection a colour-wheel is added to the workbench.

Feedback The evaluation of the product has been done by on-site demos, exhibitions, collaborations with artists and industrial designers, and a small user study. [3] The development has been going on for a while and the developers have used feedback from the users to improve their product. The first version did not have props (the eraser, magnet and tongs), hand postures changed the function. Computer scientists were not all comfortable with the interface: they asked for control points or other shape handles. Artists and general users, however, were very pleased.

Hardware requirements The method was developed to work with the *Responsive Workbench*, which is an expensive piece of equipment to show 3D images and track movements. It is a large machine that is not that mobile, making it hard to demonstrate it somewhere else, in contrast to other NPR ideas, which are not dependent on specific equipment.

Judgement An improvement to this method (as also mentioned by the authors) could be adding the notion of interfering with existing objects. Because the 3D objects cannot be sensed by a physical blockage, it is hard to attach different strokes to each other. The user could be made attentive of colliding with earlier drawn strokes by means of vibrations.
We do think the improvements that the programmers made (adding additional props for stretching, erasing, etc.) according to user feedback are good ones, that make the system more manageable.

2.3 RenderBots

The concept This technique was implemented by creating several example bots. These bots all create a different effect. Examples are the StippleBots, HatchingBots, LineBots, MosaicBots, PaintingBots, and VectorFieldBots. The last category shows the direction of the normal in the image, for more scientific purposes. It is also possible to create new bots, to implement a drawing style that is not yet present.
For each RenderBot type, the weight, maximum speed and accelerations can be set, so the bots behave to physics rules. This influences their paths, giving different results.

User interaction The bots are placed on the image by hand, and it is possible to influence the rendering by deleting bots, attracting the bots to the mouse or adding extra bots. The end of the rendering process is decided by the user, because the program is (naturally) not

capable of deciding when the rendering process is finished.
The implementation uses a more traditional user interface, with sliders and input fields to change the parameters. Sadly there are a lot of parameters: each different type of RenderBot has about 20 to 30 parameters that can be set to a preferred value. Without any knowledge of their meanings, it was unclear what the effect of each parameter was, making it hard to set them to the right value.
More natural interaction could be achieved by using a tablet, but this is not a requirement.

Hardware requirements Requirements for this program can be high: they depend on the number of bots. For example, one image can contain a lot of StippleBots, which slows down the program significantly.

Feedback This paper did not discuss any form of evaluation or implementation, they focus more on the technical details of their algorithm. We tried the demo-program and mention our own experiences here.

Judgement Some values of sliders or input fields just had to be tested to see the result.
A major flaw is that we were not able to produce any output without the RenderBot markings. The obtained output was a very unknown (but easy to make) file format.
The creators of the program were able to produce some nice results (see Figure 3).
Improvements to this software could be obtained by making the user interface more intuitive. Some parameters are guessable, but most are not.

2.4 WYSIWYG NPR

The concept The general idea is that, given a 3D model, the silhouette strokes and different types of creases are automatically found. Creases are strokes for an edge that is not part of the silhouette. Each line type belonging to a different object can then be set to adjust the stroke colour, thickness, texture and (repeated) shape. Different settings of these values can be seen in Figure 4. Once the model is loaded, it is viewable from all sides as a 2D NPR scene. With the help of an exterior program, it is even possible to create animations with it.

User interaction The program is started by loading a model, using a command prompt. Clicking on the 3D model turns it into a 2D scene. Now, any stroke type can be clicked to change its appearance.
There are many ways in which a line can be changed. The right half of the screen is completely filled with parameter sliders, and even more (collapsible) menus. There are buttons at the top which allow browsing though the different windows.
This program was designed to work with a tablet: all interaction with the image (zooming, rotating, moving) is done with the pointing device. Clicking right or left on different parts of the preview has different effects: it was not always clear what action would be performed with a certain mouse action.

Hardware requirements Generally, the implementation runs just fine. Adding too much creases can slow it down significantly, but having a lot of silhouette strokes does not seem to greatly affect the performance. [5]

Feedback This paper focuses on the technical details of this algorithm. We tried the demo-program and mention our own experiences here.

Judgement As mentioned before, the WYSIWYG NPR program allows every parameter to be set – but it needs a lot of sliders for this. The most frustrating is that even the colour selection is done with a slider, while this could

be easily done with a colour dropping tool.
Without the manual, the program would be hard to understand, especially because it also uses keyboard shortcuts to open other menus or accept changes.
However, the algorithm does allow creating very nice pictures, and one can quickly create their own (simple) drawing of a model. The program produces very nice results with a minimum of models (see Figure 4), which gives the probability to create some interactive landscape with it.
This program has way too much options. More options (features) are not always a good thing. Adding some sort of menu to open models, instead of the command prompt, is also a good idea.

3 Conclusion

After viewing different approaches that are some possible futures of NPR, we must agree that the user-friendliness of these methods can be improved. But these methods take a step in the right direction: bringing more influenceable artistic images a step closer to the home PC.

We saw Modeling with Rendering Primitives, that pretends to act a lot like drawing on a real canvas, but is different due to the fixed shape of strokes, and their mobility. It could be a nice idea to really implement the painting on a canvas with their tools, by mixing colours as variable-length strokes are drawn.
Surface Drawing is a nice and new way to model things in 3D, but not really feasible for the average user. Also, their input methods could be improved, for example, by allowing more colours to be drawn.
RenderBots yield some very nice images, but these are yet to be exported out of the program in an easy way. The parameters given here to adjust the different RenderBot types are not obvious in their effects.
The WYSIWYG NPR program features a lot of options, but this comes at the cost of a good user interface. New users will probably soon give up because they cannot find their way. Nevertheless, the program allows creating very nice images and even animations with a minimum of models and not much work.

To summarize, these new methods for applying new, innovative and interesting non-photorealistic rendering techniques to an image are great, only the developers have to think more about the users and thus the usability of the system or software that comes with it, and less about the possibilities that the software has.

References

1. Martin Schwarz, Tobias Isenberg, Katherine Mason, and Sheelagh Carpendale. Modeling with rendering primitives: an interactive non-photorealistic canvas. In *NPAR '07: Proceedings of the 5th international symposium on Non-photorealistic animation and rendering*, pages 15–22, New York, NY, USA, 2007. ACM.
2. Jens Grubert, Mark Hanckock, Sheelagh Carpendale, Edward Tse, and Tobias Isenberg. Interacting with stroke-based rendering on a wall display. Technical report, University of Calgary and University of Groningen, 2007.
3. Steven Schkolne, Michael Pruett, and Peter Schröder. Surface drawing: creating organic 3d shapes with the hand and tangible tools. In *CHI '01: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 261–268, New York, NY, USA, 2001. ACM.
4. Stefan Schlechtweg, Tobias Germer, and Thomas Strothotte. RenderBots—Multi Agent Systems for Direct Image Generation. *Computer Graphics Forum*, 24(2):137–148, June 2005.
5. Robert D. Kalnins, Lee Markosian, Barbara J. Meier, Michael A. Kowalski, Joseph C. Lee, Philip L. Davidson, Matthew Webb, John F. Hughes, and Adam Finkelstein. Wysiwyg npr: drawing strokes directly on 3d models. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 755–762, New York, NY, USA, 2002. ACM.

Architecture documentation on design rationale and decision

Bart van Teeseling and Arnaud van Gelder

Rijksuniversiteit Groningen, The Netherlands

Abstract. Software architectures usually don't document the rationale behind design decisions and design decisions themselves very well. We will present a couple of methods to identify the decisions and rationale in a structured way and document them so every stakeholder knows exactly why design decisions are made and what influence they have on the system design in a whole. We'll compare the methods and show the similarities and differences between them.

1 Introduction

Large software systems have a complex structure and designing such a system involves much thinking about the requirements and how to turn those requirements eventually into a real working system. In this process many decisions have to be made. The architecture which describes the system often doesn't show what the decisions were, and yet as important, why these decisions have been taken and how they influence the rest of the system. Software architecture is not only important for developing the system, people have focused mainly on the result for many years, but it should also be the basic assumption for continuous maintenance, enhancements and integration. Architects and designers who are not the original developers often have to maintain and control the quality of the system. These people require a good understanding of the architecture and the system. They have to understand why decisions have been taken and how they affect parts of the system. Generally design rationale (the fundamental considerations and reasons behind decisions) cannot be obtained from design specifications because there is no systematic practice to capture them. We will compare several ways of documenting the design rationale and explain how they can be used in the right manner.

2 Problem area

Design rationale capture the reasons behind design decisions. They show why certain design choises are selected over other ones, how the design satisfies the requirements and environmental conditions influence the architecture. Often decisions are made and justified, but the justifications are often not documented and are lost over time. When designers are asked why they have made these decisions they often have to reconstruct the rationale from the design. Often they

have forgotten what the idea behind the decision was and this might result in an inconsistent design or violations of design constraints. The impact can be really serious because architecture design is fundamental to a system and rectification of errors might be very costly.
Software architecture design often involves many implicit assumptions; for example how one component influences another. A change in one part of the design might affect many other parts of the system. This change could have been a small one, but the influence in the other parts might be huge and such change-impacts can not be easily identified. Without traceable design rationale, the implicit relationshpis between the design objects can be lost, creating many problems. Examples of problems that might arise are an expensive reconstruction of the design rationale through analysis; unclear design criteria and environmental factors that influence the system; violation of design integrity when intricatley related assumptions and constraints are omitted; misunderstanding or lacking of the tradeoffs in decisions or a wrong assessment of the impact of changing requirements and evironmental factors. Therefore it is of the most importance that design decisions and rationale are well captured.

3 The stated solutions

The solutions for documenting design decisions we have considered are described in detail in [1],[2], and [3]. We will sumarize the essence of these solutions in this section.

3.1 Solution 1 - Tyree's Template

The solution described in [1] is derived from REMAP (Representation and Maintenance of Process Knowledge) and DRL (Decision Representation Language). First the criteria leading to a decision are stated in a table, together with the alternatives for the decision, like in the table below:

	Alternative 1	**Alternative 2**	**Alternative 3**
Selection criterion 1	yes	no	yes
Selection criterion 2	yes	yes	no
...	...	...	...
Selection criterion N	yes	no	no

Table 1: Check which criteria are satisfied by the design decision alternatives.

Yes or no stands for whether or not the criterion is met by the specific alternative. Using this table helps to show why a certain alternative is finally chosen in the design decision. This table can also help to trace back the decision to the requirements, since the selection criteria often relate to requirements or even are requirements themselves. When the decision is actually made its properties are displayed in a list or table:

Decision ID: Title	
Issue	
Decision	
Status	
Grouping	
Assumptions	
Constraints	
Positions	
Argument	
Implications	
Related decisions	
Related requirements	
Related artefacts	
Related principles	
Notes	

Table 2: Documenting a design decision and its properties.

Finally, [1] gives an example in which new design decisions are identified that have to be taken as a result of the initial decision just taken, and to display these in a tree, using UML notation. This way relations between different design decision can be shown in a clear way, and architects can analyze whether a new decision results in new problems. If so, the decision, and the decisions that might depend on it, might have to be reconsidered.
The authors of [1] claim that their solution conveys architectural changes, and conveys implications and rationales of design decisions, and that it provides for good traceability between decisions and requirements. It should support good agile documentation.

3.2 Solution 2 - Kruchten's Ontology

The second solution we've considered proposes to create an ontology of design decisions, in which a clear classification of decisions exists, and in which each decision has a set of relevant attributes. Also a set of relationships between decisions are defined. In [2] there are three major classes of design decisions defined:

1. **Existence decisions:** These decisions state that some element or artifact will exist in the design or implementation. This type of decisions can be further divided into structural and behavioral decisions. Also non-existence decisions can be defined, stating that some element will not appear in the design or implementation.
2. **Property decisions:** These decisions state an enduring, overall property of the system. These could be design guidelines or constraints that apply to the design as a whole. This should be a separate class of decisions because they are hard to trace.

3. **Executive decisions:** These are business driven decisions that affect the development process, the people, the organization, and the choices of technology and tools.

Attributes of a decision are Epitome (the decision itself in a textual description), a rationale, the scope of the decision, the decision state (approved, rejected, et cetera), the author, time-stamp, the decision's history, possible categories the decision may belong to, the decision cost, and the decision risk. The relationships

Name	*Type*
Epitome	Text
Rational	Text or Pointer
Scope	Text
State	Enumeration
History	List of (time stamp+author+change)
Cost	Value
Risk	Exposure level

Fig. 1. Attributes of a design decision in solution 2, picture taken from [2].

defined are mostly binary relationships that can exist between decisions, like decision A depends on decision B, A conflicts with B, et cetera.

1	Constrains
2	Forbids
3	Enables
4	Subsumes
5	Conflicts with
6	Overrides
7	Comprises (is made of)
8	Is bound to
9	Is an alternative to
10	Is related too
11	Traces to
12	Does not comply with

Fig. 2. Possible relationships between design decision in solution 2, image from [2]

3.3 Solution 3 - The AREL Model

The solution described in [3] is called the AREL (Architecture Rationale and Element Linkage) model. This model can be used to capture the relationships

between the entities *Architecture Rationale and Architecture Elements.*

First a conceptual model is presented in which a *motivational reason* (for instance a requirement) acts as input for a *design decision.* An *architecture rationale* is created by a design decision, and justifies this decision. Finally, the *decision outcome* is the result of the design decision in this model. The AREL

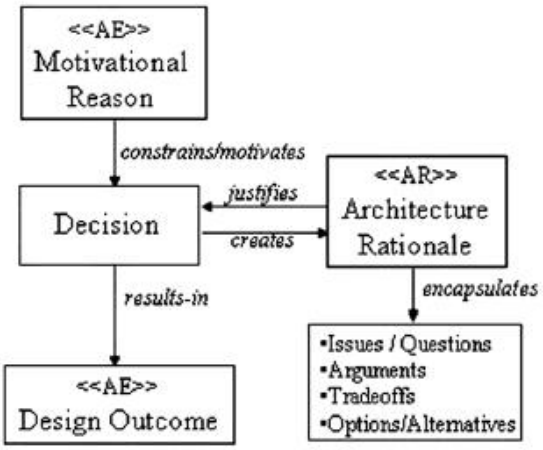


Fig. 3. Conceptual model that serves as a basis for the AREL model.

model is an implementation of the conceptual model described above using UML notation. In AREL motivational reasons, design decisions, and design outcomes are called architectural elements (AE). An architectural rationale (AR) describes related issues, options, and arguments of a design decision.

By displaying the AR's and AE's and their relationships in UML notation decisions should be made traceable according to this solution.

4 Similarities and differences between solutions

What are big differences and similarities between the three solutions described? By considering the similarities and differences between the solutions we try to find properties that the any solution for documenting design decisions should have.

4.1 Similarities

What all solutions state is the need for treating a design decision as an entity in itself. Design decisions shouldn't implicitly emerge from text and diagrams in architecture and design documents. Design decisions should be explicitly documented, so that architects can clearly understand why an architecture

has emerged to what it is, and so that they don't take any decisions in the future that conflict with important design decisions made in the past. Also, all three solutions propose to have a set of properties belonging to the decisions documented, although the properties differ in some aspects. They also all propose a UML-like way for documenting relationships between decisions. In The AREL Model this is mandatory, and in Tyree's template and Kruchten's Ontology this is optional.

4.2 Differences

The biggest difference can be found in the way decisions are displayed. While Tyree's Template proposes a rather loose format, in which an architect can describe a predefined set of attributes in his own words using text, and optionally some diagrams, Kruchten's Ontology and The AREL Model propose to use a much more strict format for describing a decision. Kruchten's Ontology defines a strict classification and set of properties, but does not explicitly propose a way displaying the decision. The AREL Model clearly states the use of UML notation for documenting the decisions.

5 Discussion

Software architectures are hard to develop and it isn't always easy to develop an architecture in which everything is crystal clear. However, it is important, especially in large software systems, that every decision is documented in a well-formed manner. Maintainers, quality managers and many other stakeholders need to understand why the system is build in this specific way. Therefore the architecture has to document all the decisions, rationale and the influence the decisions have on some parts and components of a system. [1] propose a system in which key architecture decisions are documented in table form. For every decision the properties and traces are filled in and this way people can obtain the rationale behind the decisions in a structured way. [2] proposes a system in which a decision is an entity, which has several attributes and relationships. They model this in UML which is already a standard in the software architecture community and therefore easy to adapt and to understand. [3] propose the AREL model, in which rationales are strictly coupled to decisions. This way there is a very good understanding of the traceability and the influence a change has on the rest of the system.

References

1. Tyree, J., Akerman, A.: Architecture Decisions: Demystifying Archtitecture. Capital One Financiel (2005)
2. Kruchten P.: An Ontology of Architectural Design Decisions in Software-Intensive Systems. (2004)
3. Tang, A. *et al*: A rationale-based architectural model for design traceability and reasoning. Elsevier (2006)

Creation and utilization of Pattern Languages

Dyon Keupink (s1665790) and Martijn de Groote (s1650130)

Rijksuniversiteit Groningen

Abstract. *Today, using Software Patterns to setup the architecture of a software project is becoming more and more common. In this paper we will explain what a "software pattern" is, and how one can create a pattern language from it. We will also explain how one can make use of a pattern language. How classification of architectural patterns in this context is done will also be talked about. This classification can only be done when you have a certain architectural view. What architectural views might be relevant to classify what patterns also is something we will talk about in the paper. Furthermore we will of course give examples of architectural views and the patterns that can be classified under that view. We will discuss the use of MetaPatterns; a pattern language for the actual writing of patterns. Furthermore, we will give an example of a pattern language for "Distributed computing". When addressing this specific pattern language we will mainly talk about the distribution and application infrastructure of the language and concurrency. We expect to find some general new way to describe and look at the design of software and / or it's architecture.*

1 Introduction

Christopher Alexander, an architect and author, coined the term pattern language. He used it to refer to common problems of civil and architectural design, from how cities should be laid out to where windows should be placed in a room. The idea was initially popularized in his book "A Pattern Language". Alexander's book "The Timeless Way of Building" describes what he means by pattern language and how it applies to the design and construction of buildings and towns. However, it is also applicable to any field of designing computer programs.

During a design process, the designer must make many decisions about how to solve problems. A single problem, documented with its best solution, is a single design pattern. Each pattern has a name, a descriptive entry, and some cross-references, much like a dictionary entry. A documented pattern must also explain why that solution is considered the best one for that problem in the given situation.

Just as words must have grammatical and semantic relationships to each other in order to make a spoken language useful, design patterns must be related to each other in order to form a pattern language. Implicit in Alexander's work is the idea that the patterns should be organized in such a way that it makes the most intuitive sense to

the designer. The actual structure (hierarchical, iterative, etc.) may vary, depending on the topic. Each pattern should indicate its relationship to other patterns and to the language as a whole. This gives the designer some guidance about the order in which problems should be solved. In the paper Architectural Patterns Revisited - A Pattern Language [1] by Paris Avgeriou and Uwe Zdun a common way of using and documenting Patterns was introduced by proposing a Pattern Language which acts as a superset of the existing architectural pattern collections and categorizations.
This language is particularly focused on establishing the relationships between the patterns and performs a categorization based on the concept of "architectural views".

2 Creating a pattern language from Architectural Patterns

Architectural patterns are a key concept in the field of software architecture because they offer well-established solutions to architectural problems, help to document design decisions and describe the quality attributes of a software system as forces. Those forces are working against eachother and the designer has to make choices in order to bring all forces in balance. As far as the granularity of architectural patterns is concerned, the boundaries between design patterns and architectural patterns (or potentially other classifications) are unclear. In general it is hard to draw the line between architectural patterns and design patterns. In fact, it depends heavily on the viewpoint of the designer or architect whether a specific pattern is categorized as an architectural pattern or a design pattern. Unfortunately, there is no single catalog of architectural patterns for software architects to use. Instead there is a voluminous and heterogeneous literature about patterns, where the various patterns differ in their scope, context, way of description and they are often not related in the context of a pattern language. To make things worse, many architectural pattern languages have been developed since the earlier software patterns literature was documented. And the former are not clearly related to the latter. Of course, there have been attempts to classify architectural patterns, but again there is no consensus on these classifications that could possibly lead to a single scheme.

As stated in the introduction, in the paper Architectural Patterns Revisited - A Pattern Language [1] by Paris Avgeriou and Uwe Zdun the classification scheme for architectural patterns that was proposed is based on the concept of architectural views. An architectural view is a representation of a system from the perspective of a related set of concerns (e.g. a concern in a distributed system is how the software components are allocated to network nodes). This representation is comprised of a set of system elements and the relationships associated with them. An Architectural Pattern, on the other hand, defines types of elements and relationships that work together in order to solve a particular problem from some perspective.

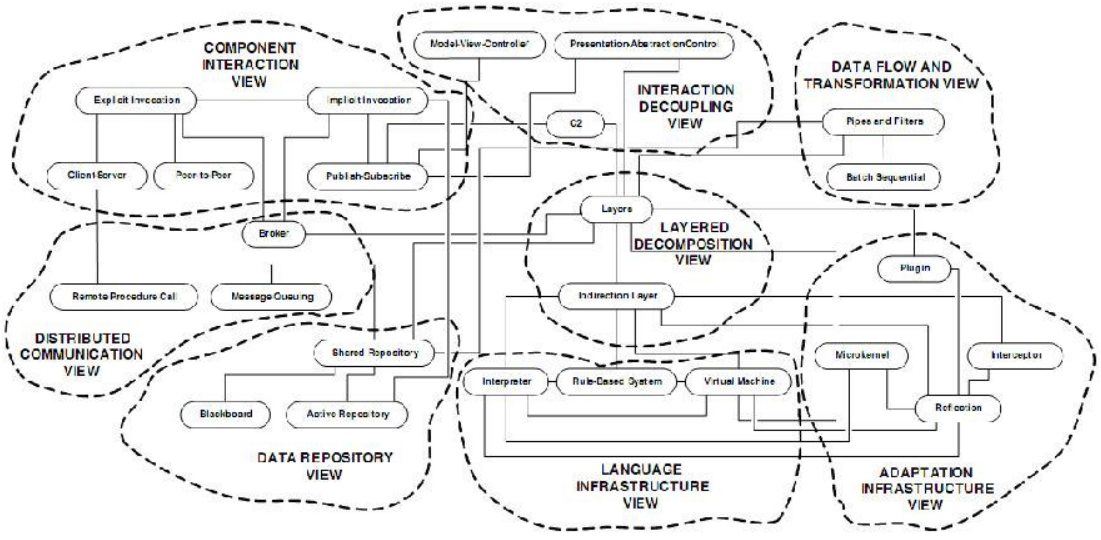


Fig. 1. The patterns of the different views and the most significant pattern relationships from [1]

Now we will summarize all the patterns Paris Avgeriou and Uwe Zdun used and show the categorization used and the relationships between the patterns. By doing this, a pattern language was created. The emphasis of this language is not on describing the individual patterns; they have already been elaborately described in other works. Instead, emphasis is given only on the related pattern sections that analytically describe the relationships between the patterns. So, the language, as a whole, is greater than the sum of its parts because it particularly focuses on establishing the relationships between the patterns in order to present the "big picture".

- *Layered decomposition view*

Each layer offers a dedicated explicit interface to the higher-level layers, which remains stable, whereas internal implementation details can change. This way the layers pattern allows the work to be sub-divided along clear boundaries. Two adjacent layers can be considered as a client-server pair, the higher layer being the client and the lower layer being the server. Also, the logic behind layers is especially obvious in the indirection layer where a special layer “hides” the details of a component or subsystem and provides access to its services. The patterns pipes and filters and shared repository may use the layers pattern for structuring the internal architecture of individual architecture elements. A microkernel is a layered architecture with three layers: external servers, the micro kernel, and internal servers. Similarly the presentation-abstraction-control pattern also enforces layers: a top layer with one agent, several intermediate layers with numerous agents, and one bottom layer which contains the “leaves” agents of the tree-like hierarchy. The indirection layer pattern allows implementing the reflection, virtual machine and interceptor patterns.

- *Data flow and transformation view*

In batch sequential there is no explicit abstraction for connectors. In the patterns "pipes" and "filters" though, the pattern considers the pipe connector to be of paramount importance for the transfer of data streams. The keyword in pipes and filters is flexibility in connecting filters through pipes in order to assemble custom configurations that solve specific problems. Also in pipes and filters there is a constant flow of data streams between the filters, while in batch sequential, the processing steps are discrete in the sense that each step finishes before the next step may commence. Pure pipes and filters is an alternative to layers and shared repositories if data sharing between nonadjacent processing tasks is not needed. More relaxed forms of the pipes and filters pattern can be combined with data repository architectures like shared repository, active repository, or blackboard to allow for data-sharing between filters. Pipes and filters can also be used for communication between layers if data flows through layers are needed.

- *Data repository view*

A shared repository offers an alternative to sequential architectures for structuring software components, such as layers and pipes and filters when data sharing or other interaction between non-adjacent components is needed. Shared repositories can be used in a pipes and filters architecture to allow for data sharing between

filters. A shared repository, where all its clients are independent components, can be considered as client-server, with the data store playing the server part. Similarly it can be considered as a system of two layers where the higher level of clients uses the services of the lower level of the shared repository. A variant of the shared repository pattern is the active repository pattern in which the notification mechanism can be realized using ordinary explicit invocations, but in most cases implicit invocations, such as publish-subcribe are more appropriate. Another variant of the shared repository pattern is the blackboard pattern which is appropriate when a shared repository is used in an immature domain in which no deterministic approach to a solution is known or feasible.

- *Adaption Infrastructure View*

Microkernels are usually structured in layers: the lowest layer implements an abstraction of the system platform, the next layer implements the services (of the internal servers), the following layer implements the functionality shared by all application versions, and the highest layer glues the external and internal servers together. Apparently, the lowest layer is an indirection layer hiding the low-level system details from the application logic. The reflection pattern is organized into layers: the meta-level contains the meta-objects which encapsulate the varying structure and behavior; and the base level contains the application logic components that depend on the meta-objects. Plugins often use reflection because reflection enables connecting components with out compile-time dependencies on them. The Reflection facility simply looks up the Plugin class at runtime and connects it.

Interceptor can use a reflection mechanism in order to query the framework and retrieve the necessary information to process incoming events.The interceptor pattern can be realized using an indirection layer or one of its variants, such as interpreter or virtual machine.

- *Language Infrastructure View*

Some interpreters use optimizations like on-the-fly byte-code compilers. Internally they realize elements of a Virtual Machine. Note that an interpreter is different to a virtual machine because it allows for runtime interpretation of scripts, whereas the virtual machine architecture depends on compilation before runtime. An alternative to interpreters and virtual machines, when rule-based or logical languages are needed, is a rule-based system. Indirection layer is the architectural foundation for interpreter, virtual machine, and rule-based system, since either the instructions of the language or the byte-code are re-directed dynamically (at runtime).

- *Interaction Decoupling View*

The notification mechanism that updates all Views and Controllers in the Model-View-Controller pattern can be based on Publish-Subscribe. All Controllers and Views subscribe to the Model, which in turn publishes the notifications. The Presentation-Abstraction-Control pattern is in essence based on MVC, in the sense that every agent is designed according to MVC: the Abstraction matches the MVC Model, while the presentation matches the MVC View and Controller. On a more macroscopic level, the PAC pattern is structured according to layers: the top layer contains the chief agent that controls the entire application; the middle layer contains agents with coarse-grained functionality while the lower layer is comprised of fine-grained agents that handle specific services which users interact with. The C2 pattern provides substrate independence, isolating a component from the components underneath it, the layer where a component is placed is in essence an indirection layer. The interaction between the C2 components takes place through asynchronous message exchange, thus utilizing an implicit invocation mechanism, and specifically callbacks, e.g. publish-subscribe.

- *Component Interaction View*

During the Explicit invocation the identification of the service supplier can be realized, for instance by using the pattern Object ID. The client also knows the location of the service supplier, and furthermore, in some systems, the service supplier needs to know about the location of the client, so that the result can be sent back. This can be achieved by Object IDs enriched with location information, as mandated by the pattern Absolute Object Reference. Explicit Invocation usually uses a Broker to hide the details of network communication and allow the components to contain only their application logic. This also applies to the Implicit Invocation Pattern, a general alternative to Explicit Invocations, even though they can be used together in a single system. An example of implicit invocation is the synchronization

between Model, View, and Controller in the MVC pattern. If Implicit Invocation is used asynchronous the Poll Object and Result Callback patterns are used. Implicit Invocation is used for looking up the initial reference in a Peer-to-Peer system. In the Client-server pattern, Both client and server must implement collective tasks, such as security, transaction, and systems management, something that is more complex in a Client-Server architecture than in simple Explicit Invocations. Sophisticated, distributed Client-Server architectures usually rely on the Broker pattern to make the complexity of the distributed communication manageable. The same is true for the Peer-to-Peer pattern: once an initial reference of the Peer-to-Peer network is found, we need to find other peers in the network. For this purpose, each peer (or each dedicated peer) realizes the Lookup pattern. Using Lookup peers can be found based on their names or their properties. Peer-to-Peer can be realized internally using Client-Server, or other patterns. As stated before, it usually also uses a Broker architecture. Whereas Client-Server and Peer-to-Peer concentrate on Explicit Invocations, Publish-Subscribe is an interaction pattern that is heavily based on Implicit Invocations. In the local context the Publish-Subscribe can be based on the Observer pattern, where the Publish-Subscribe mechanism is implemented as part of the `subject' (i.e. the event producer). In the remote context Publish-Subscribe is used in Message Queuing implementations or as a pattern implementation of its own. The Publish-Subscribe pattern can be used in the context of the Active Repository pattern, so that accessors of data subscribe to the repository, which in turn notifies them when the data is updated. Publish-Subscribe is sometimes used to realize Client-Server and Peer-to-Peer: for instance, in distributed implementations of Client-Server and Peer-to-Peer it is necessary to bridge between the asynchronous network events and the synchronous processing model of the server. This can be done using a local Publish-Subscribe model, where event handlers subscribe for the network events.

- *Distributed Communication View*

The Broker is a compound pattern that is realized using a number of remoting patterns. The most fundamental remoting patterns in a Broker architecture are Requestor, Invoker, and Marshaller. There are many others. Some important examples are a Client Proxy, which represents the remote object in the client process. This proxy has the same interface as the remote object it represents. An Interface Description is used to make the remote object's interface known to the clients. Lookup allows clients to discover remote objects. The Broker uses a layers architecture. Many well-known Broker realizations are based on the Client-Server pattern. However, the other patterns for component interactions, such as Explicit Invocation, Peer-to-Peer, Message Queuing, and Publish-Subscribe, can also use a Broker to isolate communication-related concerns, when used in a distributed setting. Remote Procedure Calls is a variant of Client-server. They usually operate in a distributed setting and are mutual alternatives. They usually employ a Broker architecture internally. Remote Procedure Calls leverage the Client-server pattern of interaction: a client invokes operations, and a server provides a well-defined set of operations that the client can invoke. Message Queuing realizes Client-Server interactions and implements Implicit Invocation as the primary invocation pattern.

In the next chapter we will discuss the use of MetaPatterns; a pattern language for the actual writing of patterns. We will learn that there is no single correct way to write a pattern language. This is because creative individuals try new ways to communicate and organize their thoughts.

3 Metapatterns

In the article Metapatterns [2], the authors Meszaros and Doble came to the conclusion that there was a "Pattern" behind what they liked about Patterns and Pattern languages of different kinds. This has lead to the development of a pattern language for writing patterns which they called "Metapatterns". Metapatterns describes the characteristics what the authors and reviewers in their review groups found most interesting about the patterns and pattern languages they reviewed. The authors describe these characteristics using patterns themselves. Patterns in Metapatterns fall into two broad categories: patterns which are applicable to all patterns, and patterns which are only applicable to patterns within a pattern language. To be exact the patterns in Metapatterns can be divided into the following categories:

- *Context-Setting Patterns*, this category introduces the concept of a Pattern (which is a solution to a problem in a certain context) and a pattern language (patterns grouped into collections which are related to each other because they solve the same problems or they are parts of a solution to a larger partitioned problem) so that they may be used throughout Metapatterns;
- *Pattern Structuring Patterns*, this category consists of patterns which describe the intended content and structure of individual patterns, whether part of a larger Pattern language or not;
- *Pattern Naming and Referencing Patterns*, this category consists of patterns which specify techniques for naming patterns and patterns which describe how including to references or other patterns should be done.
- *Patterns for making Patterns Understandable*, this category consists of patterns which embody techniques for making patterns and pattern languages easier to read, understand and apply to certain problems;
- *Pattern Language Structuring Patterns*, this category consists of patterns which specify the intended content and structure of pattern languages.

Techniques and approaches for writing patterns and pattern languages are constantly being improved because creative individuals try new ways to communicate and organize their thoughts. Also there is no single correct way to write a pattern language. This makes that the patterns in the Metapatterns language should be treated as suggestions to be tried and adopted where they help. One of the characteristics of Metapatterns is that it has been designed to give examples of the patterns it consists of. The idea is that this pattern language will help authors of patterns to organize and communicate their thoughts.

Every pattern in Metapatterns has been written with sections that can be skipped depending on the intentions of the reader. A reader who is trying to get a sense of the language can focus on the sections (in *Meszaros and Doble [2]*): Problem, Context and Solutions. In the event that a reader finds a certain pattern interesting he/she can look at a section called Forces for guidance on the determination on whether the pattern is applicable to the situation of the reader. The authors indicate that the pattern language Metapatterns will continue to evolve as long as the art of pattern writing continues to evolve. Finally, the authors of [2] make the remark that there are several areas that Metapatterns does not cover (or even attempted to cover).

4 Distributed computing

One important aspect of modern computing nowadays is distributed computing. Distributed computing enables the connection of resources and users in a transparant and scalable way. Furthermore distributed systems are open in the sense that each subsystem is always open to interaction with other systems. This kind of computing however poses new challenges, especially when distributed applications use concurrency and event handling. Patterns and pattern-languages could address at least some of these challenges.
The article of Buschmann and Henney [3] is about parts of a pattern language for distributed computing. Patterns from the Pattern-Oriented Software Architecture series [POSA1] [POSA2], the Gang-of-Four [GoF95], and some other authors of this specific topic are weaved together. This pattern language is not intended as a replacement for the original descriptions of the patterns, but instead complement them by looking at them from a different view. For implementation-details the article simply refers to the original source where the authors got the pattern from.

Distribution infrastructure

Many distributed systems are exposed to the following design challenges:

- *Location-independece of components*, in the perfect situation client components in a distributed system should be able to communicate with remote service components as if they where located in the same address space. To reach this goal the service components and clients should not include code that deals with remote communication or with location-specific details (e.g. IP-addresses and port numbers) of other components;
- *Flexible component deployment*, redeployment of a distributed system should be possible. In the best case it should be able to without shutting the entire system down and changing the system's code;
- *Integration of legacy code*, because most distributed systems are constructed from existing software components integration with the distributed system might become challenging. Especially if the source-code of the existing legacy component is no longer availlable;

- *Heterogeneous components*, a distributed system should be able to be comprised of components that are written in different programming languages.

To overcome these challenges distributed systems should be equipped with dedicated middleware and distribution infrastructures that handle the above issues in order to let the application code focus only on its primary responsibility: the implementation of the required domain-specific functionality. Technologies developed by Sun Microsystems, Microsoft and consortia like the Object Management Group (OMG) can handle these challenges. These technologies share a common architectural vision, which is reflected in the entry point of the distributed computing pattern language:

The *Broker* architectural pattern manages the interaction with remote services. Specifically the broker component is responsible for coordinating communication. One can think of types of communication like forwarding requests, transmitting results and transmitting exceptions.
One of the design-patterns that helps to realize the Broker-pattern is a distribution infrastructure pattern called "Client-Dispatcher-Server". Application of this pattern means that an extra layer will be introduced to the client-server model. This layer is called "the dispatcher component" and hides the details of the establishment of the communication between clients and servers. Furthermore, a name service provides location transparency (which means that the client-code does not necessarily needs to now the IP-address of the server it is talking to and vice versa).
Both the Broker- and the "Client-Dispatcher-Server" pattern are integrated in the pattern language of the authors as displayed below:

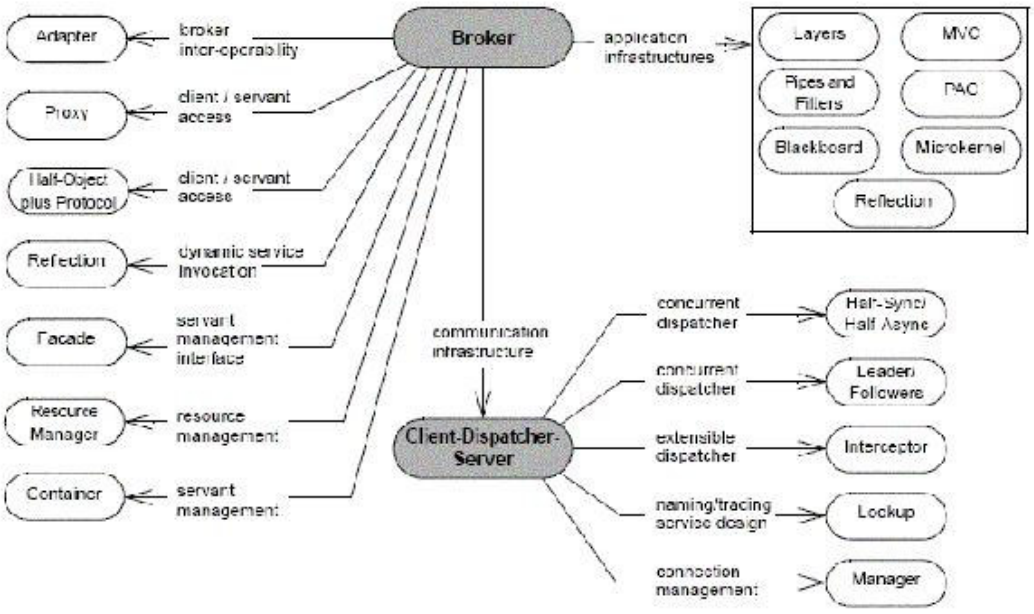


Fig. 2. Integration of the Broker- and the "Client-Dispatcher-Server"-pattern from [3]

The authors do not view Pipes, Filters and Microkernel patterns as being distribution infrastructures. Instead of this they are viewed as patterns that help to provide structure for distributed applications. The authors of the article are also aware of the fact that their distribution infrastructure patterns are not complete. As an example of this they talk about peer-to-peer computing which requires more than just Broker-based middleware.

Application infrastructure

The following questions and challenges needed to be answered in order for the authors to gain a application architecture (basically a component and subsystem decomposition that expresses the system's functionality):

- *How is application processing organized?*
- *How does the application interact with its environment?*
- *What is the life expectancy of the application?*

The distributed computing pattern language of the article included seven strategic patterns that helped to answer those questions:

- *Layers*-architectural pattern. This pattern provides structure to applications that are basically a composition of subtasks into groups in which each group of subtasks is at a certain abstraction-level;
- *Pipes and Filters*-architectural pattern provides structure for systems whose purpose is to process a stream of data;
- *BlackBoard*-architectural pattern where some subsystems each having a certain specialization assemble their knowledge to build a possibly partial or approximate solution to a problem;
- *Model-View-Controller*-architectural pattern that divides an interactive application into three components (a model, a view and a controller);
- *Presentation-Abstraction-Control*-architectural pattern is meant for interactive systems. This pattern defines a hierarchy consisting of cooperating agents, where each agent is responsible for a specific task of the application;
- *Reflection*-architectural pattern supports fundamental aspects of software systems to be adapted dynamically. When thinking about these aspects one can think of things such as function call mechanisms and structures;
- *Microkernel*-architectural pattern is used for systems whose requirements can change over time. This pattern separates a core with minimal functionality from extended functionality and parts that are specific to the customer.

These seven patterns comprise all the patterns from A Systems Of Patterns (POSA) that is classified as user interaction and system adaptation-patterns.

Concurrency

When developing distributed systems, concurrency is often used as well. Concurrency-programming has the following challenges:

- *Application diversity*, different types of components expose the application to different structural and behavioural characteristics;
- *Multi-threading-costs*. Thread-management, context switches, synchronization and data-movement influence how the application will behave (e.g. overhead);
- *Multi-threading hazards*, incorrect use of synchronization mechanisms can not only lead to extra overhead but also to application misbehavior due to deadlocks and race conditions;
- *Portability*, different hardware and software platforms that the distributed application must be able to work on/with complicate the development of concurrent applications. This accidental complexity arises from limitations with existing development methods, tools and operating systems.

The above challenges has the consequence that a concurrent architecture must always be specified early in the development-cycle of a software system. To cope with the above challenges the authors included the following patterns to their pattern language:

- *Half-Sync/Half-Async* architectural pattern introduces two intercommunicating layers, one for asynchronous and one for synchronous processing of a service;
- *Leader/Followers*-architectural pattern. This pattern introduces a model where multiple threads take turn sharing a set of event sources. This is done to detect, demultiplex, dispatch and process requests that occur on the event sources;
- *Active Object* design pattern. This pattern enhances concurrency and makes it easier for synchronized access to objects that reside in their own thread of control;
- *Monitor Object* design pattern. This pattern ensures that only one method at a certain time is running in a object. To do this it synchronizes concurrent method execution;
- *Guarded Suspension* design pattern. Objects within this pattern can only execute when certain conditions hold.

The authors of the article have deliberately left the Thread-Specific-Storage and the Scheduler-pattern out of their pattern language. Unfortunately they did not give an reason to why they did this.

Event handling

The following patterns have been included to enable Event handling:

- *Reactor* architectural pattern. This pattern allows event-driven applications to dispatch and demultiplex service requests which are being delivered to an application from a client or several clients;

- *Proactor* architectural pattern. This pattern allows efficient dispatch and demultiplex service requests which are triggered by completion of asynchrous operations;
- *Acceptator-Connector* design pattern. The processing performed by the peer services after they are connected and initialized are decoupled and initialized in a networked system;
- *Asynchronous Completion Token* design pattern. This pattern allows to demultiplex and process efficiently the responses of asynchronous operations which are invoked on the services.

Conclusion

In the first part of the paper we showed how different Architectural Patterns can be combined to create a Pattern Language. It actually formed a whole architecture which can be used in software development projects. This architecture provides a sole base for architect builders who want to create good software. In larger projects, you can be sure that most of the patterns mentioned above will be implemented in some way. Our advise to software developers is: before coding, think of building your architecture and use patterns to create it! They provide in common tasks to get your software to work.

The authors Meszaros and Doble [2] came to the conclusion that there was a “Pattern” behind what they liked about Patterns and Pattern languages of different kinds. This lead to the development of a pattern language for writing patterns called “Metapatterns”. Metapatterns describes using patterns the characteristics which the authors and other reviewers in their review groups found most interesting about the patterns and pattern languages that they reviewed. In the event that a reader finds a certain pattern interesting he/she can look at a section called Forces for guidance on the determination on wheter the pattern is applicable to the situation of the reader.

Finally we believe that when it comes to distributed systems there is made a good start in creating a pattern language for this type of computing. Specifically things like the handling of events, concurrency and also the application and distribution interface have been handled to some extend already. We believe that in the future a pattern language like the one discribed in [3] will make it very easy to take important decissions when it comes to the engineering of distributed software.

References

[1] Paris Avgeriou and Uwe Zdun, Architectural Patterns Revisited - A Pattern Language

[2] Gerard Meszaros and Jim Doble, MetaPatterns: A Pattern Language for Pattern Writing

[3] Frank Buschmann and Kevlin Henney, A Distributed Computing Pattern Language

Links

http://en.wikipedia.org/wiki/Pattern_language
http://en.wikipedia.org/wiki/Distributed_computing

Using Model Checking to Prevent Data Loss by File System Errors.

Simon P. Takens (1675478), Dennis Kanon (1673491)

Department of Computer Science, University of Groningen
s.t.takens@student.rug.nl, d.kanon@student.rug.nl

Abstract: File System crashes and errors can cause a lot of “headaches” to users and administrators everywhere. We would like to do research in the possibilities to detect these errors before they actually happen. Using earlier Research done in this field we would like to think of ways to try and warn before such errors happen. The reason why this is interesting is because if you can detect a “dangerous” and/or “vulnerable” area or write action, you would be able to give a warning in time to the user to make a backup of his data before proceeding further. We would like to use the possibilities found by earlier research in the field of using model checking to find serious file system errors to identify the problems before they might happen. Model checking uses techniques to reduce states so that a huge amount of states that are actually important can be checked in an efficient manner. So by combining this technology with the findings made in previous research we hope to achieve a way to detect errors before they happen.

1 introduction

Using model checking to prevent data loss by file system errors is an ambitious research that can be very useful. That is why we have chosen to research this and make it our main focus.

This is interesting for use in Operating Systems already on the market and even as a separate product. Often people, especially consumer and users of workstations without a IT background, don't realise the impact of a File System error and that rebooting and/or using the system might only make the problem worse.

First we give an introduction to model checking. Secondly we talk about the main part of our research namely using model checking to detect hard disk errors. This consisting of the advantages of using model checking, detecting error/crashes before they happen and implementation. Finally we give our conclusion.

2 Intro to model checking

2.1 What is model checking

Model checking is a process that checks if a model adheres to rules of existing logics. Model checking is a general concept that applies to all kinds of logics and structures suitable.

2.2 Where is it used

Model checking is mostly used in the design of hardware, in software design however it isn't used as much as it is with hardware. This is caused by different points of view in software and model design in this field. Often model checking proofs or disproof's something that is actually the opposite.

2.3 What are we going to use it for

We want to use the logical formulas to check and predict hard disk errors before they actually happen. Now with the help of model checking this process should be able to speed up. Hoping we are able to get it to a level so that it can run as a constant resident program, without interfering too much with the other processes on the system.

This would be a very handy tool for servers to backup really critical data before actual crashes and/or errors happen. Also, if the process gets speed up to a level that it doesn't interfere in a notable way with the rest of the computer, it can be a resident program for everyday consumer use as well.

3 Using model checking to detect hard disk errors

3.1 Advantages of using model checking

The first advantage that can be mentioned is that if you are going to use model checking for security than you wouldn't need RAID 1. Although model checking is cheaper than RAID 1 it is not as good and can still fail and it's also a strain on the system. But this strain is relative because model checking uses techniques to reduce states so that a huge amount of states that are actually important can be checked in an efficient manner.

Recent work has developed implementation-level model checkers that check implementation code directly without requiring an abstract specification [2, 3, 4]. [1] has leveraged this approach to create a model checking infrastructure, the File System Checker (FiSC), which lets implementers model-check real, unmodified file systems with relatively little model checking knowledge.

Model checking works best at ferreting out complex interactions of a small number of nouns (files, directories, blocks, threads, etc) since this small number allows caching techniques to give the most leverage. Also some bugs are hard to find without a model checker.

Model checking has some nice properties. First, it makes it trivial to verify that the original error is fixed. Second, it allows more comprehensive testing of patches than appears to be done in commercial software houses. Third, it finds the corner-case implications of seemingly local changes in seconds and demonstrates that they violate important consistency invariants.

Model checking also has multi-threading support. The model checker is single-threaded both above and below the system call interface. Above, because only a single user process does file system operations. Below, because each state transition runs automatically to completion. This means many interfering state modifications never occur in the checked system. In particular, in terms of high-level errors, file system operations never interleave and, consequently, neither do partially completed transactions (either in memory or on disk). [1] expects both to be a fruitful source of bugs that is why we have to check them to.

3.2 Detect errors/crashes before they happen

3.2.1 Checking the file system

This section will talk more in-depth about how we are planning to check our File System and the writes to our File system. We are not considering the reads because if those would cause errors/crashes the system and the disk where already unstable and a mayor hardware failure is imminent.

There is only a slight difference between checking an empty just formatted disk, or a disk that is already in use. We split them up in two separate sections and just mention the extra consideration the second one needs. We talk a bit more in detail about each possible option the software has as an initial state.

On an empty disk

We check our empty disk by adding a layer for writing files, this layer basically adds a virtual file system on which we can test every possible way to write data to this file system. By using model checking we create a model of this disk with each write, seeing if the write causes any problems before continuing.

With this we will be able to give a warning and cancel the write in case it would be harmful to the system and thus let the system prevent possible bad writes or choose the safest one that won't cause a File System Error and keep the data on this disk safe.

If there is a write to a non-system disk, and it causes a major problem in at least one test in every possible way it can be written, then the user gets a warning not to write this data and is requested to try this again on a later time. If the error persists the users is prompted to write the data to another disk or data-carrier.

On an already in use disk
On a disk that is already in use there are more variables that need to be kept in mind, one cannot just assume: "This disk was working "OK" till now so lets assume we can start with an empty slate as if it where a disk that was just formatted." For instance many users at home use the same disk that also contains their OS or other data that they got before they started using this system.

So before we do a model checking per write, we make a map of the whole disk as to provide a model of the data already there (metadata). To be absolutely sure this mapping won't cause errors the map will first be written to a virtual disk created in the computers RAM. Although the mapping of a hard drive is pretty compact it may be done in blocks if the mapping turns out to exceed the RAM size.

After this check is done, the system runs the checks (described earlier) on the populated disk. First to check if it would be safe and what the best way is to write this model to the disk in use. Secondly we write the actual model to the disk. Afterwards any disk write made by the OS or user can be checked.

However since, as stated above, this disk can be the main OS disk, the OS itself without any interaction (of the user per say) does a lot of these disk-writes. To make a system that can communicate with the OS at that level, the program has to run at driver level as if becoming total separate driver that pipes the date towards the disk, thus adding a layer to the communication between OS and the Disks involved.

As described above, if the user makes his own writes to the disk, say save a document. He would just get a warning if the write would be harmful and is asked to try again or doing the disk write to another disk or data carrier.

One might argue if you stop the OS from making writes, the OS will cease its functions and wont work properly. We just want to add the possibility of backing up the data, even if the OS is corrupted and the OS would ruin the data on the disk if used. This can involve a separate OS booting up to create the backup of the essential data. So even if the OS is corrupted and needs to be repaired or reinstalled the data is safe to be backed up first.

3.2.2 Description of the different checks available

Doing different checks is essential to our research. That is why we have looked at a lot of the different checks that were well defined, and used the ones that are relevant to the tasks we want them to fulfil.

Here are the descriptions of the checks that we used, which are preformed by the File System Checker (FiSC). We used FiSC because it inspects the actual state of the system and can thus catch errors that are difficult or impossible to diagnose with static analysis. It is capable of doing a set of general checks that could apply to any code run in the kernel:

· **Deadlock** Checking for circular waits is important to ensure to integrity of the files that are being used;
· **NULL** FiSC reports an error whenever a NULL pointer accurse;
· **Paired functions** There are some kernel functions, like iget, iput for inode allocation and dget, dput for directory cache entries, which should always be called in pairs. FiSC instruments these functions in the kernel and then check that they are always called in pairs while running the model checker;
· **Memory leak** Memory allocation and deallocation functions need to be instrumented so FiSC can track currently used memory. The system-wide freelist needs to be altered to prevent memory consumers from allocating objects without the model checker's knowledge.

These were generic checks. FiSC checks the following consistency properties (descriptions of these checks can be found in [1]):

· **System calls map to actions**
· **Changed buffers are marked dirty**
· **Buffer consistency**
· **Double fsck**

3.2.3 Scaling

For some systems scaling may not be realistic but because it can be useful we are thinking of implementing it but are not yet at that stage. Because there is a lot of residual research still to be done on this subject and we haven't had time to find the parts that are relevant to our research. Some possible fields of interest would be to run the checks in parallel.

A good area for further research would be the recent developments in Video Cards. These GPU (Graphical Processing Unit) can run a lot of calculations in parallel without breaking a sweat. If optimised for these new universal shaders/stream processors, we think we can reap enormous benefits of this development and offer a rather cheap solution as an add-on card for servers.

3.3 Implementation

In this section we try to better explain how we plan to implement this system. This section tells in more detail about what we exactly researched in earlier articles that use model checking to prevent data loss in different ways.

3.3.1 Driver Level

Because we want to monitor and test actual reads and writes before they happen, we want to run our virtual disk at a driver level. Thus providing one extra layer between the actual OS and the Disk. The reason for this is that the OS can't write data to disks without interception of our system.

We planned to implement this by sitting in-between the two, to an OS the program would identify itself as a HDD but to the HDD it would be looking like it was an OS making a write.

3.3.2 Optimisation

Because model checking can be very intensive, model checking big HDDs (that are standard these days) can take very long so we need some form of optimisation.

Cluster-based model checking

With the current trend of multi core CPU's it would be a shame not to use them for this process. Like most model checking, can easily be converted to run in parallel. This has an advantage that it speeds up the whole check and the model making of an already in used disk considerably.

In the future 4 core systems will be sold a lot more and 8 core systems are already on the horizon. Making this process parallel would not only be a smart choice with this in mind, but almost a necessity.

Splitting up large models

Because a computers memory is a lot smaller then its HDD capacity there is a good change that a model of this HDD will be bigger then the memory available. The solution to this would be to split up the model in separate chunks, because you don't want to swap, as that would give you more data to check before writing. One could first check a chunk of model before checking the next one required and so on.

These chunks would have to be considerably smaller then the memory because the system also needs to remember the states it wend trough during the check.

4 Conclusions

Although our source documents had a lot of problems providing a good system for normal sized disks, we think it is still a feasible solution to provide checks before the

actual data is written. We think that with the coming of multi core CPU's and faster bandwidth in memory this thing would actually be very well possible.

Also by creating certain optimisation and not fixing the disk only to warn users that data is being corrupted one can optimise to an extend not possible with checking. By cutting out the more strenuous tests and checks. We hope to provide a better performing alternative to the ones used. One might be able to further evolve and improve on the existing system. The goal is an almost unnoticeable effect on resource usage so that a server can run this system in the background.

References

[1] Junfeng Yang, Paul Twohey, Dawson Engler and Madanlal Musuvathi. Using Model Checking to Find Serious File System Errors.

[2] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In Proceedings of the 24th ACM Symposium on Principles of Programming Languages, 1997.

[3] M. Musuvathi and D. R. Engler. Model checking large network protocol implementations. In Proceedings of the First Symposium on Networked Systems Design and Implementation, 2004.

[4] M. Musuvathi, D. Y. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In Proceedings of the Fifth Symposium on Operating Systems Design and Implementation, 2002.