MASTER'S THESIS | LUND UNIVERSITY 2015

Multi-coefficient Parallel Adaptive Wavelet Rendering

Robin Somers

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2015-05

Lund University

Master Thesis

Multi-coefficient Parallel Adaptive Wavelet Rendering

Author:
Robin Somers

Supervisor:
Dr. Michael Doggett

Examiner:
Prof. Tomas Akenine-Möller

Department of Computer Science

March 2015

Lund University

LUND UNIVERSITY

Abstract

Faculty of Engineering, LTH
Department of Computer Science

Master of Science in Engineering

Multi-coefficient Parallel Adaptive Wavelet Rendering

by Robin Somers

Adaptive Wavelet Rendering is a sampling method used for ray tracing in order to render photorealistic images. The concept of wavelets and the so-called discrete wavelet transform is used to create a multi-scale view of the image when sampling. This allows the method to identify image variance on different levels and therefore to differentiate and appropriately handle variance resulting from sharp edges or blurred regions, thus creating visually appealing images with minimal work even for complex scenes.

This thesis investigates the algorithm and specifically how it can be improved through multi-core concurrency. To this end an alternative version is proposed which works on multiple regions simultaneously. Parallelism is considered for both the original and the alternative version. Furthermore, they are compared both based on the qualitative difference between their results and their respective performance gains through concurrency.

It is shown that although the structure of the algorithm limits the potential for concurrency, some improvements can be made, especially for the alternative multi-coefficient version with results maintaining high quality, thus making it better suited to todays highly parallel compute systems. Finally some future directions are considered based on the detailed analysis of how concurrency affects the major components of the algorithm.

Acknowledgements

The thesis would not have been possible without the previous work by Ryan S. Overbeck, Craig Donner and Ravi Ramamoorthi.

I would also like to thank Michael Doggett for the support and supervision throughout the entire process, from start to finish.

Contents

Abstract i

Acknowledgements ii

Contents iii

List of Figures v

List of Tables vi

1 Introduction 1

2 Background and Theory 4
- 2.1 Ray Tracing . . . 4
- 2.2 Sampling . . . 5
  - 2.2.1 Random sampling . . . 6
  - 2.2.2 Jittered sampling . . . 7
  - 2.2.3 Adaptive sampling . . . 7
- 2.3 Previous work . . . 7
  - 2.3.1 Multidimensional adaptive sampling . . . 8
  - 2.3.2 Adaptive wavelet sampling . . . 8
- 2.4 Wavelets . . . 8
  - 2.4.1 Example: The Haar wavelet . . . 8
  - 2.4.2 Uses for images and computer graphics . . . 9
  - 2.4.3 Wavelet basis functions . . . 11
  - 2.4.4 Two-dimensional transforms . . . 13
  - 2.4.5 The discrete wavelet transform . . . 14
  - 2.4.6 Boundary handling . . . 16
  - 2.4.7 Types of wavelets . . . 18
  - 2.4.8 Cohen-Daubechies-Feauveau 9/7 . . . 20
- 2.5 Parallel computation . . . 21
  - 2.5.1 Multithreading . . . 21
  - 2.5.2 Concurrency hazards . . . 22

3 Adaptive Wavelet Rendering 25
- 3.1 Definitions and conventions . . . 26
- 3.2 Sampling stage . . . 27

3.3 Reconstruction stage 32

4 Parallel Adaptive Wavelet Rendering 35
4.1 Concurrency in AWR 35
4.1.1 Initial sampling 36
4.1.2 Adaptive sampling 36
4.1.3 Wavelet transforms 36
4.1.4 Priority computations 37
4.1.5 Other components 38
4.2 Modifying the algorithm 38
4.2.1 Multiple coefficients per iteration 38
4.2.2 Performance consequences 39
4.2.3 Qualitative consequences 40

5 Results 42
5.1 Breakdowns 43
5.2 Total runtimes 44
5.3 Sampling stages 45
5.4 Wavelet transforms 46
5.5 Priority computations 47
5.6 Varying the sampling distribution function 48

6 Discussion 50
6.1 Conclusions 50
6.2 Future work 54

A Additional runtime breakdowns 56
A.1 Spheres scene 57
A.2 Buddha scene 58
A.3 Creature scene 59
A.4 Strands scene 60

Bibliography 61

List of Figures

2.1 Ray tracing 5
2.2 Simple sample techniques 6
2.3 Transformed image example 11
2.4 Haar Wavelet 19
2.5 Daubechies 4 Wavelet 19
2.6 CDF 9/7 Wavelet 20
2.7 Multithreading 21

3.1 Spheres scene: Example AWR image 26
3.2 Wavelet magnitude example 30
3.3 2D sampling distribution 30
3.4 Buddha scene: AWR image without blurry effects 31
3.5 Wavelet reconstruction 33
3.6 Sampling comparison: jittered and AWR 34

4.1 Quality comparison: original and modified AWR 41

5.1 Creature scene 43
5.2 Strands scene 45

List of Tables

2.1 CDF 9/7 filter bank 20

5.1 Runtime breakdown for creature scene, $S(k) = 32 \cdot 4^k$ 44
5.2 Total runtimes, $S(k) = 32 \cdot 4^k$ 44
5.3 Initial sampling results 45
5.4 Adaptive sampling results 46
5.5 Sequential vs. parallel local transforms 46
5.6 Local transform results 47
5.7 Sequential vs. parallel priority computations 47
5.8 Priority computation results 47
5.9 Runtime breakdown for creature scene, $S(k) = 45 \cdot 4^k$ 48
5.10 Total runtimes, $S(k) = 45 \cdot 4^k$ 48
5.11 Runtime breakdown for creature scene, $S(k) = 64 \cdot 2^k$ 49
5.12 Total runtimes, $S(k) = 64 \cdot 2^k$ 49

Note: the appendix contains additional runtime breakdown tables.

Chapter 1

Introduction

Photorealistic images, often used for promotional material or for special effects in for example movies, are typically rendered using a technique called ray tracing. Scenes are represented by data and relations that specify the appearances of objects. Rays are figuratively shot from cameras in order to sample the scene and create an image. Typically several samples per pixel are needed even for the most basic scenes in order to guarantee an accurate image. When complex effects are added, such as depth of field, area lighting and motion blur, the amount of samples needed increases significantly in order to decrease the extra variance that comes with each effect.

The time required to render an image using ray tracing is directly proportional to the number of rays that are traced through the scene. Thus, if accuracy is of importance, adding complex effects to scene will increase the running time considerably as the amount of samples per pixel will need to be increased to keep variance at tolerable levels. However, many effects will only affect parts of the image resulting in many pixels having an unnecessarily high amount of samples. Therefore many methods have been proposed to sample different image regions with different amounts of samples. By adaptively sampling the scene samples can be focused to the areas where they are needed the most. This requires an algorithm that keeps track of the variance in the image and uses that information to place samples where they are needed the most. By doing this, sample counts for whole images can generally be kept fairly low while the most sampled individual pixels or regions can get the amount they need.

Constructing an algorithm that does not introduce too much overhead while simultaneously making smart choices is nontrivial. Most such algorithms perform well for certain effects, such as variations in the image plane, but work less efficiently for smooth regions depending on other dimensions such as time for example. Others have the opposite

strengths and weaknesses and some might perform well in both cases but gets exponentially slower with the number of added dimensions concerning each effect in the scene. It is also of importance that the algorithm can target edges as well as smooth regions in order to produce accurate images.

In order to handle all of the cases and issues mentioned above an algorithm was proposed by Overbeck et al. in 2009[1]. The algorithm uses the concept of wavelets in order to generalize the notion of pixels into a hierarchical structure. Sharp variations such as edges are captured in the analysis of the lower levels of the hierarchy while the variance from smooth regions become apparent in the higher levels. Furthermore, variance from all dimensions, not only the image plane, are naturally captured by the wavelet analysis and are thus taken into account when deciding where samples needs to be placed. Consequently, since the added dimensions do not explicitly enter into the calculations, the time complexity is kept reasonable with fairly low overhead overall.

The algorithm has two parts. The first and main part is where the the scene is adaptively sampled through analysing and updating the wavelet structure. After the first part the image is mostly complete but some noise remains, mostly around the edges of smooth regions. The second part utilizes the information contained within the wavelet structure to smoothen the image and remove the remaining noise. Together they produce an accurate and visually pleasing image.

The aim of this thesis is to use a version of the algorithm described above as a foundation and improve upon it by introducing concurrency in its computations. Modern computers often have processors with multiple cores or other hardware traits which allow for parallel computations. Used well, a program can be sped up significantly by performing tasks concurrently. However, parallelism creates difficulties and pitfalls that are not present in sequential programs, which means that it is not suitable for many applications. The adaptive wavelet rendering algorithm is one case where its structure forces some sequential computations while seemingly allowing for parallelism in other parts. It is thus unclear whether introducing concurrency will benefit the algorithm as a whole or if it will only introduce extra complexity which will negatively affect the performance.

Ray tracing is an accurate but slow process. Much research has been carried out in an effort to speed it up which, among other things, has resulted in adaptive sampling algorithms. To further increase efficiency parallelism becomes a natural next step. In this thesis the adaptive wavelet rendering algorithm is investigated to get insight as to whether or not its structure allows for improvements through parallelism. To this end the original version will be considered as well as a slightly altered version which sacrifices some adaptiveness in order to increase the workload that can be done in parallel. This

is done by considering multiple *scale coefficients* simultaneously, where each refers to an image region, rather than just one at a time as in the original method.

The thesis is structured as follows: Chapter 2 presents the necessary background and theory concerning Ray Tracing, sampling, wavelets and concurrency. Chapter 3 presents the original adaptive wavelet algorithm and goes through its steps in detail. Following that the potential improvements through parallelism are presented in Chapter 4 with the results presented in Chapter 5. Finally in Chapter 6 the results are discussed and possible future work is considered.

Chapter 2

Background and Theory

This chapter includes a brief introduction to ray tracing and sampling techniques. It goes on to cover some previous work in the area of adaptive sampling, followed by theory about wavelets relevant to this thesis. Finally section 2.5 gives an introduction to parallel computing; its uses and hazards.

2.1 Ray Tracing

For Computer Graphics there are two main rendering techniques. Rasterization, which the graphics processing units (GPU) were originally designed for, is mainly used for real-time rendering due to the massive parallelism offered by the GPU:s. It is not relevant for this thesis and is mentioned mostly for reference. Ray tracing is fundamentally different and harder to perform in parallel compared to rasterization, although work is being done to create efficient real-time ray tracers on the GPU as well. What ray tracing offers is a simpler concept that is more physically intuitive. Reflections, refractions and shadows all emerge naturally and behave as we would expect. It is also easier to include effects such as global illumination and the resulting softer shadows. Consequently, it is therefore a far superior algorithm when photorealism is the goal and fast rendering is not a necessity.

A scene is represented by data and relations that together specify how the different surfaces will appear, when viewed from a given location, with regards to the light sources and surroundings. A camera (or eye) is figuratively placed somewhere in the scene. Rays are then traced in different angles from the camera out into the scene. The rays typically behave as rays of light, or photons, as we normally picture them physically. This means that they bounce off reflective surfaces, bend when entering a refractive material and stop when hitting a diffuse surface or possibly spread randomly for global illumination.

Each ray gets an initial direction when traced. This direction determines its path through the scene based on the rules and relations that determine the physics behind it. When the ray stops on some surface the resulting color is recorded. The direction also specifies which pixel the ray belongs to as shown in Figure 2.1. Typically each pixel will have several rays contributing to it with the final pixel color being the arithmetic mean of all the colors recorded by the rays.

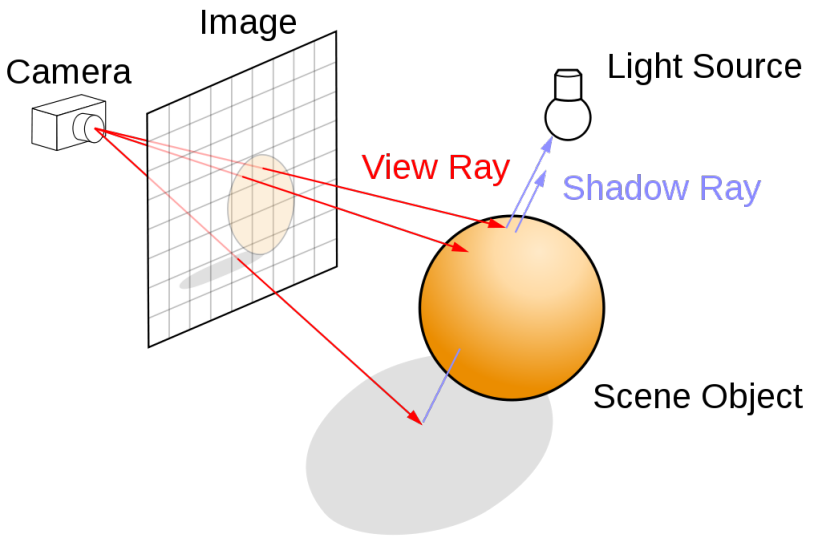


FIGURE 2.1: Rays belong to different pixels when sampling the scene.
Source: Wikipedia page for ray tracing by user Henrik

Only the basic concepts of ray tracing are described above. There is more than one way to construct a tracer, all requiring careful considerations. Each kind has different strengths and flaws and most can be extended in different ways to capture complex effects. For the purpose of this thesis a *Whitted* tracer is used for its simplicity. It recursively traces rays when hitting reflective or refractive surfaces and generally works quite fast. It does not, however, naturally take global illumination into account and it does not create soft shadows. Therefore, in order to create more complex scenes for testing, it has been extended to support depth of field and motion blur, thus simulating the use of a more physically realistic camera able to take pictures of objects in motion.

2.2 Sampling

Deciding where and how to trace rays is an important part of any ray tracing algorithm since the accuracy of the produced image will improve with the number of samples. However, the running time increases linearly with the sample count which means that it should be kept as low as possible. Creating a smart sampling algorithm is an optimization problem. The accuracy of the image should be as high as possible with the lowest

possible amount of samples. Furthermore, complex effects in a scene might introduce other variables that need to be sampled. Depth of field requires each ray to be sent from a specific lens position and motion blur requires sampling the scene in time. While taking everything into account the extra overhead of the algorithm must be kept low for it to have an overall positive effect.

It is extremely hard, if not impossible, to find a scheme that performs optimally in every situation but many good algorithms exist. Three simple strategies are explained below; first a naive one and then another that is just barely harder to implement while giving a considerable performance improvement. These are illustrated in Figure 2.2 as well. The third is more complicated and performs better but still has clear flaws. The next section gives brief descriptions of two more advanced sampling algorithms including the one of particular interest for this thesis, the adaptive wavelet sampling scheme. In Chapter 3 the algorithm is described in detail.

2.2.1 Random sampling

The brute force method to sample a scene is to generate the sample locations randomly. This strategy is not really feasible since the sample count required for an image to converge will be very high, consequently giving long running times. Therefore it is not done in practice. The simplest scheme that is feasible is to decide a specific amount of samples per pixel and randomly distribute them within each pixel. This guarantees that each pixel receives an adequate amount of samples but it is still likely that some pixels will have clustered distributions which might not capture the proper color of those pixels even with many samples.

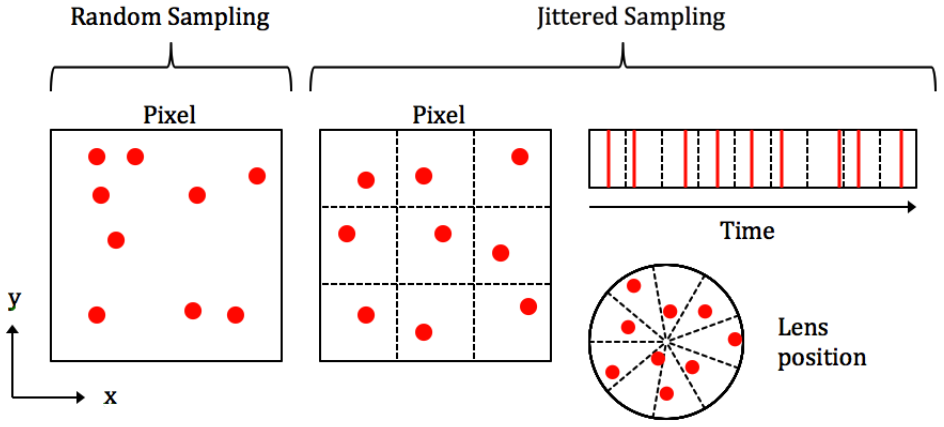


FIGURE 2.2: Simple sample techniques. Red dots or lines represent samples

2.2.2 Jittered sampling

If the sample count per pixel is chosen as a square of an integer the pixel can be split up into uniform regions where one sample can be placed randomly in each such region within each pixel. This solves the problem of clustering as the samples are forced apart to cover different parts of the pixel. Jittered sampling can easily be applied to time and lens sampling as well as shown in Figure 2.2. The main problem that remains is that each pixel still receives the same amount of samples. Most often the amount of samples needed in different pixels will vary considerably in a scene. To guarantee adequate convergence everywhere, each pixel will have to receive the same amount of samples as the pixel that needs the most.

2.2.3 Adaptive sampling

There are two ways to enable sampling with different amounts of samples per pixel. The first is to have the user specify the sample count for each pixel which immediately becomes infeasible for any reasonable image size. The second is to let the ray tracer itself measure the variance in the scene and sample where the variance is high. This is what is called adaptive sampling and it can been done in many different ways. A simple example is to place a few samples in the corners of each pixel. The pixels for which sample colors vary greatly are split into smaller regions and the procedure is repeated for each such region. When the color varies little for a given region it is considered to be sampled enough. This algorithm, while being able to reduce sample counts and focus attention where needed, clearly still has a risk of missing thin variations within a pixel and might also perform badly in regions where time and lens sampling are of importance.

2.3 Previous work

Both of the works described in this section are of importance to this thesis. The first is a well known algorithm for adaptive sampling with a particular weakness that the second is specifically designed not to have. The second work is the foundation of this thesis.

2.3.1 Multidimensional adaptive sampling

The algorithm, created by Hachisuka et al. in 2008[2], commonly referred to as MDAS, manages to produce good results both in image space dimensions and in extra dimensions that results in blurry smooth regions. It does this by operating on the full multidimensional space, focusing on regions where the local contrast between samples is high. However, while the algorithm produces good results for indivudual or a limited number of effects it scales poorly as the dimensionality increases. This is what Overbeck refers to as the curse of dimensionality in his paper and is a direct consequence of Bahvalov's theorems which state that the best case performance of any numerical integration algorithm over standard Monte Carlo decreases exponentially with the number of dimensions[1].

2.3.2 Adaptive wavelet sampling

In order to not fall victim to the curse of dimensionality the adaptive wavelet sampling algorithm was created by Overbeck et al. in 2009[1]. It avoids the issue by using the properties of the wavelet transform to distinguish image space discontinuities from smooth variation caused by added dimensions. Numerical integration thus only occurs in the image space dimensions. The different parts of the algorithm is described in detail in chapter 3. However, to fully understand what it does, knowledge about wavelets and the discrete wavelet transform is needed. Therefore the next section contains the wavelet theory relevant to the algorithm.

2.4 Wavelets

This section is split to cover each important concept individually, all relevant to the wavelet algorithm with the exception of the first two subsections which can easily be skipped but serve as an instructive introduction to wavelets and their uses for images and computer graphics.

2.4.1 Example: The Haar wavelet

Consider the following vector:

$$\boldsymbol{c}_0 = (c_0^{(0)}, c_0^{(1)}, c_0^{(2)}, c_0^{(3)}, c_0^{(4)}, c_0^{(5)}, c_0^{(6)}, c_0^{(7)}) = (3, 7, 8, 6, 2, 4, 9, 5)$$

Suppose that we want to know the averages of the first and second values and the third and fourth and so on. This gives four values in total. However, we also want to keep the

information about the original values while not storing more than eight values in total. This can be done by transforming the original vector as follows:

$$c_{k+1}^{i} = \frac{c_k^{2i} + c_k^{2i+1}}{2} \tag{2.1}$$

$$d_{k+1}^{i} = \frac{c_k^{2i} - c_k^{2i+1}}{2} \tag{2.2}$$

where the new vector becomes:

$$\boldsymbol{c}_1 = (c_1^{(0)}, c_1^{(1)}, c_1^{(2)}, c_1^{(3)}, d_1^{(0)}, d_1^{(1)}, d_1^{(2)}, d_1^{(3)}) = (5, 7, 3, 7, -2, 1, -1, 2)$$

The first four values, given by equation 2.1 is simply the mean of each pair of values in the previous vector. Equation 2.2 specifies the deviation from the mean each of the values in the pair has. To reconstruct the original values each deviation is added and subtracted to the corresponding mean to get the two values of the pair respectively. The transform can then be applied again to further separate core features (means) from details (deviations). In this example we get the following sequence (Note that the transform is only applied to the c_k values each time):

$$\boldsymbol{c}_0 = (c_0^{(0)}, c_0^{(1)}, c_0^{(2)}, c_0^{(3)}, c_0^{(4)}, c_0^{(5)}, c_0^{(6)}, c_0^{(7)}) = (3, 7, 8, 6, 2, 4, 9, 5)$$
$$\boldsymbol{c}_1 = (c_1^{(0)}, c_1^{(1)}, c_1^{(2)}, c_1^{(3)}, d_1^{(0)}, d_1^{(1)}, d_1^{(2)}, d_1^{(3)}) = (5, 7, 3, 7, -2, 1, -1, 2)$$
$$\boldsymbol{c}_2 = (c_2^{(0)}, c_2^{(1)}, d_2^{(0)}, d_2^{(1)}, d_1^{(0)}, d_1^{(1)}, d_1^{(2)}, d_1^{(3)}) = (6, 5, -1, -2, -2, 1, -1, 2)$$
$$\boldsymbol{c}_3 = (c_3^{(0)}, d_3^{(0)}, d_2^{(0)}, d_2^{(1)}, d_1^{(0)}, d_1^{(1)}, d_1^{(2)}, d_1^{(3)}) = (5.5, 0.5, -1, -2, -2, 1, -1, 2)$$

The first value in the last vector is effectively the mean value for the entire starting vector. That value in combination with the values specifying the details are enough to reconstruct each of the previous vectors. The number of c_k values specifies the resolution of the vector and the d_k values are kept only for reconstruction purposes. Without them the means would not give enough information to uniquely determine the original values. This transform is known as the Haar transform and is the simplest possible wavelet transform[3, p. 9-11]. The characteristic of splitting a number of values into a more rough representation with corresponding values specifying the details is shared between all wavelet transforms and is what makes them useful. The difference is that more complex wavelets have more complicated summations.

2.4.2 Uses for images and computer graphics

The properties of the wavelet transform means that it has many uses in applications where separating, analysing, editing or removing fine details are of importance. Three

examples of such applications in image editing and computer graphics are presented below. However, there are many more and for the interested the book "Wavelets for Computer Graphics : Theory and applications" is recommended[3].

Image compression

The wavelet transform can easily be extended to more dimensions by just applying the transform on every dimension independently which is described in more detail in sections 2.4.3 and 2.4.4. It is therefore possible to apply it to the pixels of images by performing the transform row and column wise as is done in Figure 2.3. Thus, instead of specifying the color of each pixel exactly, it is possible to have, at the extreme, one single pixel which contains the mean color of the entire image where every other pixel contains information about the deviation as in the previous example. The image can then be reconstructed when necessary[3, p. 18-31].

The question is what benefits such a representation gives since the same amount of numbers will need to be stored as in the original image. There are two main reasons. Firstly the magnitudes of the deviations will in general be much smaller than the original values. This means that it will require less bits to represent the values and consequently, less memory to store the exact same image, but with the inconvenience of having to transform and reconstruct it. The operations required for the transform are cheap, however, so the memory savings might well be worth it. Secondly, if a perfect reconstruction is not necessary, in other words if the compression is allowed to be lossy, then by removing the detail coefficients in order of the smallest magnitudes first it is guaranteed that the image will be affected as little as possible. Each value removed will decrease the amount of memory necessary to store the image so detail coefficients can be removed until sufficient compression is achieved. Since images typically have large regions with small variations, there is often a high potential for good compression without significant changes in appearance. This is one of the main uses for wavelets overall and is used for example in JPEG 2000[4].

Curve editing

Wavelets are very well suited for editing curves. With a set of control points the wavelet transform is able to separate features such as the sweep and the character of a curve which makes it possible to change one without affecting the other. This can be used in any application where such curve manipulation might be desirable such as a drawing tool or for easy editing of an interpolation curve[3, p. 109-120].

Surface applications

Surfaces can be represented by wavelets as well by generalizing the transform to two dimensions. However, unlike for image applications, it is not sufficient to apply the transform for each dimension independently. The entire theory must be reformulated to take multidimensional considerations into account[3, p. 141-159]. Fortunately this is fairly straightforward and can easily be extended to three or more dimensions as well. Surfaces can then be manipulated in ways similar to curve editing or compressed into simpler shapes by using the same strategy as for image compression[3, p. 141-159].

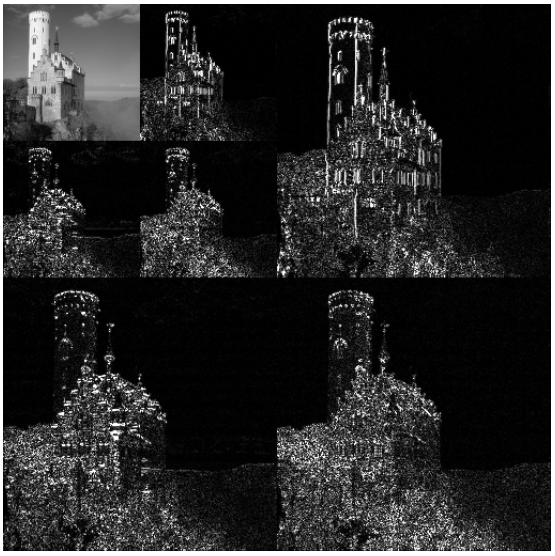


FIGURE 2.3: The wavelet transform has been applied twice using the nonstandard decomposition which yields a coarse approximation of the original image in the top left corner while containing detail information in the other parts of the image. The transform is the same as used in JPEG 2000. Source: Wikipedia page for Discrete wavelet transform by user Alessio Damato

2.4.3 Wavelet basis functions

A one-dimensional wavelet basis is defined by the translations and dilations of a scale function ϕ and a wavelet function ψ as follows[1]:

$$\phi(x)_{k,i} = \phi(2^{-k}x - i) \tag{2.3}$$

$$\psi(x)_{k,i} = \psi(2^{-k}x - i) \tag{2.4}$$

The parameter k is the level of the transform. When $k = n$ for some n it means that it should be used for an n times transformed function. Thus $k = 1$ is the finest wavelet level and $k = 0$ refers to the original input function. The translation is given by i.

Generalizing the definitions above to two dimensions (for flat surfaces such as images) requires taking the tensor products of the one-dimensional functions giving one scale function and three wavelet functions: $\Phi = \phi \otimes \phi, \Psi^1 = \phi \otimes \psi, \Psi^2 = \psi \otimes \phi, \Psi^3 = \psi \otimes \psi$

The two-dimensional basis becomes (for $1 \leq \alpha \leq 3$ and j being the translation in the y-direction)[1]:

$$\Phi_{k,ij}(x,y) = \Phi(2^{-k}x - i, 2^{-k}y - j) \tag{2.5}$$

$$\Psi^{\alpha}_{k,ij}(x,y) = \Psi^{\alpha}(2^{-k}x - i, 2^{-k}y - j) \tag{2.6}$$

The process of computing the transform for some input function $f(x,y)$ is called analysis and the inverse computation is called synthesis. The general form of wavelet analysis is computed using the inner product[1]:

$$c_{k,ij} = \langle f, \Phi_{ij,k} \rangle = \iint f \cdot \Phi_{ij,k}\, dxdy \tag{2.7}$$

$$d^{\alpha}_{k,ij} = \langle f, \Psi^{\alpha}_{ij,k} \rangle = \iint f \cdot \Psi^{\alpha}_{ij,k}\, dxdy \tag{2.8}$$

Here $c_{k,ij}$ are the so called scale coefficients and $d^{\alpha}_{k,ij}$ are the respective wavelet coefficients. Different inner products might be appropriate depending on the application. Note that in the Haar example (section 2.4.1) the mean and deviation values are the scale and wavelet coefficients respectively. Being a one-dimensional example there is only one set of wavelet coefficients.

Mathematically the scale functions (1D: 2.3 2D: 2.5) for a given level k define a basis for the vector space V^k containing every possible element that can be described on that level[3, p. 11-16]. For example, if we consider images, where $f(x,y)$ can be thought of as a piecewise constant function that describes the colors of every pixel, of size 512×512 then V^0 contains every possible image of that size, V^1 contains every possible image of size 256×256 and more generally V^k contains every image of size $(512 \cdot 2^{-k}) \times (512 \cdot 2^{-k})$.

The wavelet functions (1D: 2.4 2D: 2.6) for a given level k defines another basis, namely for the vector space W^k which has the following properties[3, p. 11-16]:

1. W^k is the orthogonal complement to V^k in V^{k-1} which, in other words, means that the basis functions of W^k together with the basis functions of V^k form a basis for V^{k-1}.

2. Every basis function of W^k is orthogonal to every basis function of V^k under the chosen inner product.

For the image example this means that V^1 combined with W^1 will contain every possible image of size 512×512. Sometimes a better suited wavelet basis, for certain applications, might be constructed by relaxing the rules above or by making them more strict. Different types of wavelets are presented in section 2.4.7.

2.4.4 Two-dimensional transforms

Transforming a two-dimensional flat surface such as an image only requires transforming rows and columns independently. However there are many different orders possible for the wavelet analysis. Below the two most common are presented. Note that deciding the analysis order immediately decides the synthesis order as well. When doing the reverse transform the order should be the complete inverse compared to the analysis.

Standard decomposition

The standard decomposition of level k is obtained by applying the one-dimensional transform on each row k times followed by applying the same transform on each column k times. It is appealing since it is very easy to implement for a computation. The number of assignment operations required for an $n \times n$ image is $4(n^2 - n)$[3, p. 21-27].

Nonstandard decomposition

For the nonstandard decomposition a full transformation involves computing the transform first once for each row and then once for each column. This will effectively split the image up into four quadrants, each corresponding to one of the four basis functions defined by 2.5 and 2.6. Each successive round (which each increases the transform level by one) will recursively apply the full transform only to the quadrant corresponding to the scale functions[3, p. 21-27]. Figure 2.3 is a two-level nonstandard decomposition.

The nonstandard decomposition requires slightly more work to implement but offers the advantage of a clear splitting corresponding to the basis functions. Furthermore the quadrant corresponding to the scale functions is effectively a lower resolution version of the original image. These properties are crucial for the adaptive wavelet rendering algorithm which means that the nonstandard decomposition is chosen by default for the transforms. In addition the number of assignment operations is less than for the standard decomposition. For an $n \times n$ image the amount is only $\frac{8}{3}(n^2 - 1)$[3, p. 21-27].

2.4.5 The discrete wavelet transform

So far the formulas and strategies have not been dependent on whether the problem is continuous (real-valued) or discrete. Most applications, including image handling, has a discrete set of data and thus the transform will have to reflect this. There are several variants that do this but the one used for the algorithm is the so called discrete wavelet transform, or DWT. The first thing to note is that numerical approximations of the inner products (equations 2.7 and 2.8) are costly operations and should be avoided if possible. In practice other methods are used such as a process called lifting or by using filter banks. For the purpose of this thesis the filter bank method will be explained as it is the one used in the algorithm.

The method needs only to be explained for a one-dimensional discrete set of data, which will be referred to as a signal from now on, since it can then easily be extended to work on a two-dimensional set by using one of the strategies presented in section 2.4.4. Firstly, the wavelet basis needs to be chosen. That choice determines which filter bank to use. The common wavelet choices have filter banks that are well documented in books or on the internet. If not, the filter bank will have to be computed which is well beyond the scope of this thesis. Each filter bank consists of four sets of coefficients:

Analysis Low-Pass: denoted as a
Analysis High-Pass: denoted as b
Synthesis Low-Pass: denoted as p
Synthesis High-Pass: denoted as q

Each of the filters typically have, at index 0 (for example $a[0]$), the most central coefficient if the number of coefficients is uneven or one of the two most central ones if the number is even. Note that negative indices are allowed. The number of coefficients for different wavelets can vary considerably. The numbers signify how many parts of the signal that will influence every resulting scale or wavelet coefficient. A large number means a large region of influence which may or may not be appealing depending on the application.

Some additional definitions and considerations are needed before presenting the formulas. The discrete signal is denoted by $c_k[n]$ where k is the level of the transform and n is the index. $c_{k+1}[n]$ will effectively be the scale coefficients for the next level and should have half the number of elements compared to $c_k[n]$. In other words, if for level k there are N elements, indexed from 0 to $N-1$, then $c_{k+1}[n]$ should only be computed up to, but not including, index $N/2$. The wavelet coefficients $d_{k+1}[n]$ should be computed up to the same index as well (each scale coefficient has a corresponding wavelet coefficient with the same index). Successive transforms should only be applied to the scale coefficients

which effectively halves the number of scale coefficients on each level. With this the analysis equations, for a given index n, becomes[5, p. 156-157]:

$$c_{k+1}[n] = \sum_m a[m]c_k[2n-m] \tag{2.9}$$

$$d_{k+1}[n] = \sum_m b[m]c_k[2n-m] \tag{2.10}$$

In order to reconstruct the original signal the inverse transform is required. Note that in this case the coefficients $c_k[n]$ should be computed from index 0 to $N-1$. The synthesis equation for index n is[5, p. 156-157]:

$$c_k[n] = \sum_m p[n-2m]c_{k+1}[m] + q[n-2m]d_{k+1}[m] \tag{2.11}$$

In every one of the equations above the summation index m should be summed over all values for which the filters (a, b, p, q) are defined. They can be thought of as containing the value 0 for every index outside their defined bounds. However, such a summation will in general allow values for which the signal is not defined. The reason for this is that the wavelet transform, as described so far, is defined over an infinite space[6]. Thus boundaries are not taken into account in the descriptions and equations above. This clearly becomes a problem since the transform is applied to finite regions in most applications, such as for an image. Therefore, there needs to be a strategy for handling boundaries. The next section gives several examples of such strategies and presents a redefinition of the equations above for the strategy used for this thesis.

Note for the interested: The calculations from the Haar wavelet example in section 2.4.1 can be represented by equations 2.9 and 2.10 without boundary issues. By defining the filters so that $a[0] = a[-1] = b[0] = 1/2$ and $b[-1] = -1/2$ and by doing the summations over m from -1 to 0 the transform becomes as defined by 2.1 and 2.2. Clearly this representation, while correct, is not ideal for describing the Haar transform. The equations can easily be re-indexed to give a more intuitive summation and indexing for the Haar filter banks and other similar wavelets. However, the given representation works well for symmetrical wavelets and symmetric boundary handling, both of which are used in this thesis.

2.4.6 Boundary handling

Below are four examples of how the boundaries can be handled but note that there are other ways[6].

Edge wavelets

The most proper method for handling edges is to redefine the wavelets on a bounded interval[6]. Practically, for the filter bank method, this means having special filter bank coefficients around the boundaries to reflect the lack of information on the other side. However, recomputing the coefficients in this way for a given wavelet is mathematically complicated and documented computations are in general not as easy to find. When complete accuracy is required this is the superior method. For most applications, however, simpler methods work well enough since there often are other aspects that affects the quality of the result to a higher degree than a slight border error.

Zero padding

The easiest method for dealing with edges is to assume that any part of the signal outside its defined region is zero. This artificially creates a sharp discontinuity along the edges which can affect the values of the scale and wavelet coefficients significantly[6]. Furthermore the reconstruction (synthesis) might not be perfect around the edges. There are applications where such boundary handling is sufficient but while edge wavelets might be unnecessarily complicated, zero padding risks introducing too much error in general.

Periodic extension

Assuming that the signal is periodic is a fairly common way of handling boundaries[6]. If a signal c contains N elements indexed from 0 to $N-1$, then $c[N] = c[0], c[N+2] = c[2], c[-3] = c[N-3]$ and so on. Since the values on the opposite boundaries need not be related this method also introduces artificial discontinuities. However, if there is no loss of information, for example by a compression strategy, synthesis will produce a perfect reconstruction. Thus periodic extension is better than zero padding while also being almost as easy to implement.

Symmetric extension

Assuming that the the signal is symmetric around the borders is yet another method[6]. However, that assumption alone leads to a problem. Assume that we have a low-pass filter a with three coefficients indexed from -1 to 1. We get, from equation 2.9:

$$c_{k+1}[1] = a[-1]c_k[3] + a[0]c_k[2] + a[1]c_k[1]$$
$$c_{k+1}[-1] = a[-1]c_k[-1] + a[0]c_k[-2] + a[1]c_k[-3] = a[-1]c_k[1] + a[0]c_k[2] + a[1]c_k[3]$$

It immediately becomes clear that $c_{k+1}[1]$ and $c_{k+1}[-1]$ are not necessarily equal. If successive transforms or the inverse transform should be applied it requires storing additional values outside the boundaries which is clearly not appealing. However, if the filter is symmetric around zero as well the symmetry is preserved and only one of the values needs to be stored. Therefore symmetric extension is only a good alternative for symmetric filters.

This method, with symmetric filters to give a nonexpansive output, enables perfect reconstruction and will avoid large artificial discontinuities at the borders. The assumption can still skew the results but without large discontinuities this effect is very small. Symmetric extension will, in general, produce very good results and is frequently used[7]. Because of this many of the common wavelets were constructed to be symmetric. It is also the method used for this thesis and will therefore be described in more detail since additional considerations need to be made.

First note that wavelets can be constructed to have symmetric filters with both even and odd number of coefficients. These need to be treated differently with the even length filters being the more complex ones[5, p. 159-161]. Thus, most filters are not only constructed to be symmetric, but also to have an odd number of coefficients. Below only odd length filters are considered as the wavelet used in the thesis has an odd number of coefficients in all four filters.

Knowing that the filters has odd length and defining L_a and L_b as the length of a and b respectively, the new analysis equations becomes[5, p. 159-164]:

$$c_{k+1}[n] = \sum_{m=-(L_a-1)/2}^{(L_a-1)/2} a[m]c_k[2n-m] \tag{2.12}$$

$$d_{k+1}[n] = \sum_{m=-(L_b-1)/2}^{(L_b-1)/2} b[m]c_k[2n-m+1] \tag{2.13}$$

The shift in the second equation is necessary for the symmetric calculations to work properly. For the equations above the signal is treated as exhibiting whole-sample symmetry which means that it is symmetric around the end points 0 and $N-1$ (where N is the signal length)[5, p. 159-164]. Thus $c[-1] = c[1], c[-3] = c[3], c[N] = c[N-2]$ and $c[N+5] = c[N-7]$ and so on. For the synthesis equation it is not quite that simple. With L_p and L_q being the lengths of p and q we get the new synthesis equation[5, p. 159-164]:

$$c_k[n] = \sum_m p[n-2m]c_{k+1}[m] + q[n-2m-1]d_{k+1}[m] \tag{2.14}$$

where m is every integer that fulfills $-\frac{\max(L_p,L_q)-1}{2} + n - 1 \leq 2m \leq \frac{\max(L_p,L_q)-1}{2} + n$

Firstly note the shift for the wavelet coefficient. Furthermore, for correct symmetry handling, the left boundary (at index 0) of c_k and the right boundary (at index $N-1$) of d_k in equation 2.14 should be treated as having whole-sample symmetry just as before. However, the right boundary of c_k and the left boundary of d_k should be treated as exhibiting half-sample symmetry which means that the signal is symmetric around $-\frac{1}{2}$ and $N - \frac{1}{2}$ respectively[5, p. 159-164]. Effectively this means that $c[-1] = c[0], c[-3] = c[2], c[N] = c[N-1]$ and $c[N+5] = c[N-6]$ and so on. Equations 2.12, 2.13 and 2.14, together with these considerations completely describe a full DWT with boundary handling through symmetric extension.

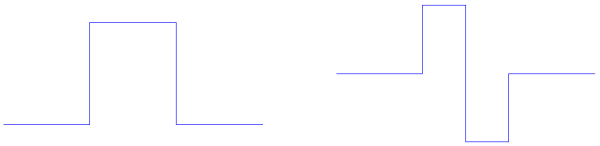
2.4.7 Types of wavelets

In this section different types of wavelets are discussed. Their different properties means that they are useful for different kinds of applications.

Orthogonal wavelets

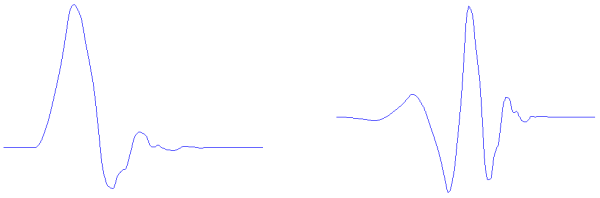
Orthogonal wavelets are those for which the scale functions are orthogonal to each other, the wavelet functions are orthogonal to each other and each of the wavelet functions is orthogonal to every coarser (higher level) scale function[3, p. 85-87]. One consequence of this is that the synthesis filters are immediately given by the analysis filters by the relations $p = a$ and $q = b$. Examples of orthogonal wavelets include the Haar wavelet and the Daubechies wavelets which is a whole family of wavelets with different filter sizes[3, p. 87-89].

Orthogonality is desirable since it generally gives simple calculations and constructions of wavelets. However it severely restricts the constructions and makes it impossible to achieve certain other desirable traits. For example it is impossible to construct



(A) Scale function (B) Wavelet function

FIGURE 2.4: Shape of the Haar Wavelet



(A) Scale function (B) Wavelet function

FIGURE 2.5: Shape of the Daubechies 4 Wavelet

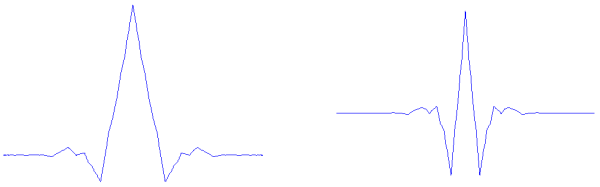
orthogonal, compactly supported and symmetric wavelets (other than the Haar basis)[3, p. 89-91]. Therefore, sacrificing orthogonality is necessary to create wavelets with more varying features.

Semiorthogonal wavelets

By solely requiring that wavelet functions should be orthogonal to all coarser scale functions the result is semiorthogonal wavelets[3, p. 89-91]. The relaxed restrictions are equivalent to the rules given in section 2.4.3 and allows for wavelets with other kinds of properties such as symmetry[3, p. 89-91]. However, without as much guaranteed orthogonality, they become more difficult to construct. Moreover the resulting analysis filters may have so many elements that computations can get inefficient in time[3, p. 97]. One family of semiorthogonal wavelets that manages to avoid inefficiency are spline wavelets, which are built from B-splines and are useful for example for curve-editing[3, p. 91-97].

Biorthogonal wavelets

In order to gain even more control over the construction and resulting filter banks the final orthogonality requirement can be removed as well. Fortunately, it is possible to create such wavelets that still have many of the important properties of semiorthogonal



(A) Scale function (B) Wavelet function

FIGURE 2.6: Shape of the CDF 9/7 Wavelet

wavelets. These are called biorthogonal wavelets with the only restriction that both the transform and the inverse transform exists, in other words that a, b, p and q are well-defined[3, p. 97-108]. While being even more complex to construct properly, they allow for many different variations with properties suited for different applications. The CDF (Cohen-Daubechies-Feauveau) family of wavelets are all biorthogonal, with CDF 9/7 being of special interest as it is the wavelet used in the algorithm for this thesis[8].

2.4.8 Cohen-Daubechies-Feauveau 9/7

The CDF 9/7 wavelet, shown in Figure 2.6, is a biorthogonal symmetric wavelet[8]. It is used for high-quality lossy encoding in the JPEG 2000 because it is very smooth[1]. The analysis low-pass and synthesis high-pass filters both have 9 coefficients while the analysis high-pass and synthesis low-pass has 7 each, both odd numbers, and they are what 9/7 refers to. Other wavelets in the CDF family have other filter sizes which makes it a suitable identifier. However there are other naming conventions as well. The CDF 9/7 filter is sometimes referred to as CDF 4.4 or biorthogonal 4.4 where the number refers to the fact that the filters have four vanishing moments[5, p. 162]. This means that they can exactly fit polynomials of degree 3 or lower[1]. The filter bank coefficients for CDF 9/7 are given in Table 2.1.

Index	a	b	p	q
0	0.85269867901	0.78848561641	0.78848561641	0.85269867901
-1, 1	0.37740285561	-0.41809227322	0.41809227322	-0.37740285561
-2, 2	-0.11062440442	-0.040689417609	-0.040689417609	-0.11062440442
-3, 3	-0.02384946502	0.064538882629	-0.064538882629	0.02384946502
-4, 4	0.03782845551	—	—	0.03782845551

TABLE 2.1: Filter bank coefficients for CDF 9/7 wavelet. a and b are analysis low- and high-pass filters respectively and p and q are synthesis low- and high-pass filters.

2.5 Parallel computation

Parallelism, or concurrency, is about two or more activities happening simultaneously. For computers this refers to a single system performing several independent tasks in parallel, rather than one after the other (sequentially). Although parallelism can be simulated in software through context switching, which is explained below, truly parallel computations can only be achieved if the hardware itself supports it. Today, hardware concurrency can be achieved in multiple ways. Graphics Processing Units (GPU) are designed to be massively parallel in their computations. Some processors make use of SIMD extensions (Single Instruction, Multiple Data) which allows for instructions on sets of data rather than element-wise. For this thesis, however, the Central Processing Unit (CPU) is used with parallelism achieved by multithreading.

2.5.1 Multithreading

Modern CPUs typically have more than one core. This allows for true concurrency as each core can run its own separate tasks. The most common approach to make use of the concurrency in mainstream languages is to split a single process into multiple threads, each running independently and perhaps with different sets of instructions. However, since they are still part of the same process, they share the same address space and thus share most of the data. In essence global variables are shared while pointers and references can be passed between threads to access anything else[9, p. 2-6].

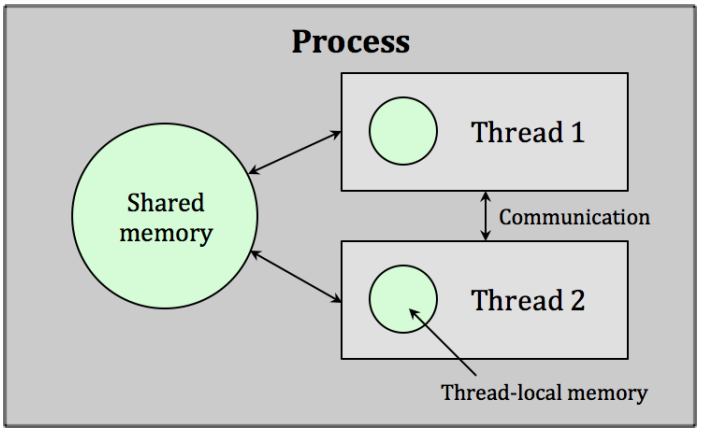


FIGURE 2.7: A process running two threads with both shared and thread-local memory. The threads are run in parallel but can still communicate and access each others memory if necessary through the use of references and pointers.

The number of threads that can be run concurrently is typically equal to the number of cores in the processor. Nevertheless, there is nothing that prevents launching additional threads which is not necessarily a bad thing. On the contrary it is often useful to have many threads running. Context switching allows the processor cores to switch between all of the different threads, or tasks, many times per second. This creates an illusion of concurrency which, for example, allows computer users to have several programs running at the same time[9, p. 2-4]. Furthermore, while the number of concurrent threads is related to the number of cores, it is not always exactly the same. Modern Intel processors, for example, has two logical processors per core which share execution resources[10]. This enables two threads per core that are not truly concurrent but, with effective context switching, are able to perform slightly more computations compared to the usage of a single thread. This is called hyperthreading[10].

Concurrency is used mainly for two reasons. It separates concerns and it can drastically improve performance. By using parallelism a process can be completed much more efficiently if the work can be divided evenly between threads. Sometimes this means dividing up a single task into equal parts and letting each thread do the same type of work. Other times threads have different responsibilities which, for example, can allow a background tasks to be completed while other threads work on the main problem. However, while many processes can be improved by concurrency it is not always the case. The benefits from multithreading comes with a number of issues as well which are not present in regular sequential programming.

2.5.2 Concurrency hazards

Different threads in a single process share the same address space. This is largely what makes multithreading useful but it is also what makes concurrency programming complicated. To begin with many operations and updates to data structures are not instant. This leads to so called race conditions where the outcome depends on the relative ordering of execution of operations between two or more threads[9, p. 34-36]. Some race conditions are benign such as two threads adding numbers to a global value in which case the order does not matter. Other times race conditions can cause the entire program to crash such as when several threads adds and removes items from a linked list simultaneously. The pointers might very well end up pointing to faulty locations, thus corrupting the data structure. Furthermore, race conditions might lead to indeterministic pieces of code. For example if one thread reads from the same memory location that another thread writes to it becomes unclear whether the reading thread receives the old or the new value. This may not be a problem depending on the application but often it is.

There is more than one way to deal with race conditions. A common one is the use of mutexes. A mutex (mutual exclusion) is associated with some shared data and whenever a thread needs access to that data it has to acquire and lock the mutex and later unlock it when the work is finished[9, p. 37]. As long as the mutex is locked it cannot be acquired by another thread which means they either have to wait until their turn or do some other task in the meantime if possible. With the protection that mutexes offer race conditions can be avoided. However, if multiple mutexes are used simultaneously it can lead to deadlocks where threads wait for each other to finish, each one holding a mutex that some other thread wants, and progress stops completely[9, p. 47-49].

Deadlocks can often be prevented by carefully considering orderings of locks and unlocks and is therefore not one of the bigger issues[9, p. 47-50]. More problematic is the fact that in any program with a high number of race condition much waiting will generally occur due to mutexes or other methods of handling race conditions. Waiting effectively means loss of concurrency and since the launching and stopping of threads, locking and unlocking of mutexes all come with some inherent overhead, the overall running time might be longer for the multithreaded version of such a program compared to the single-threaded version[9, p. 8-9]. Moreover, any program that frequently needs to make certain that threads are properly synchronized in their operations or regularly requires threads to stop and launch again, due to there being some operations in between that cannot effectively be divided between threads, also risks adding more runtime through multithreading than it removes.

Sharing the same address space comes with another type of issue as well. Every processor core generally has its own cache memory where it stores data that was recently accessed or modified due to it being likely that the same data will be accessed soon again. Whenever data is shared and one core modifies it, the cache memory in the other cores become invalidated which means that they will have read the data from the main memory instead[9, p. 235-237]. Main memory reads take considerably longer than cache memory reads which thus adds running time. This becomes especially noticeable if the shared data is accessed very often and is sometimes called cache ping-pong[9, p. 235-237]. When data needs to be shared, this unfortunately can not be avoided. However, when possible, it is generally better if each thread handles its own data with a merge between threads at the end. This, of course, requires the merging to be effective for it to have a net gain in time.

Finally there is a more subtle and sinister version of the cache ping-pong problem called false sharing[9, p. 237-238]. Even if two different threads on two different cores are accessing two different variables and at least one is writing it might result in cache invalidation for the other. They may not be sharing the same data but they are,

unintentionally, sharing the same cache line, or rather their two individual pieces of data are sharing the cache line. When any data in a cache line is invalidated the whole line becomes invalidated as well[9, p. 237-238]. Consequently data local to different threads should generally be kept apart in the memory. This is more easily said than done as it is not always clear how exactly data is ordered. However there are guidelines for separating data and methods that decrease the risk for false sharing, sometimes by sacrificing some extra memory which is a typical tradeoff for efficient multithreaded programming[9, p. 237-238][11].

Delving deeper into the problems and their solutions is beyond the scope of this thesis. Note, however, that care has been taken with regards to these issues when implementing the parallel components used in this thesis. When an issue is unavoidable and suspected to affect the results it will be mentioned.

Chapter 3

Adaptive Wavelet Rendering

This chapter explains the adaptive wavelet rendering algorithm in detail. It consists of two separate parts that work together to create a final image. The first is the most crucial part where the majority of the time is spent. It is the part where the samples are distributed in the scene and will thus be called the sampling stage. The following applies to the first stage[12]:

- The fine scale wavelets will handle edge regions accurately.
- The coarse scale wavelets will handle smooth regions accurately.
- The fine scale wavelets will *not* handle smooth regions accurately. The result is that boundaries of smooth regions will be noisy.

The image produced after the first step will thus contain some unwanted noise. Therefore the second step is needed which is referred to as the reconstruction stage. Here, the information from the first stage will be used to smooth the image and effectively remove the unwanted noise.

Important note: The algorithm described in this chapter was created by Ryan S. Overbeck and Craig Donner with prof. Ravi Ramamoorthi as suporvisor[1]. Every step, formula and heuristic should be accredited to them unless otherwise noted. However, the implementation for the thesis might differ in some ways compared to theirs where information concerning implementation details were sparse or omitted from the paper. Thus, it is not guaranteed that every step described below is exactly as for the original implementation.

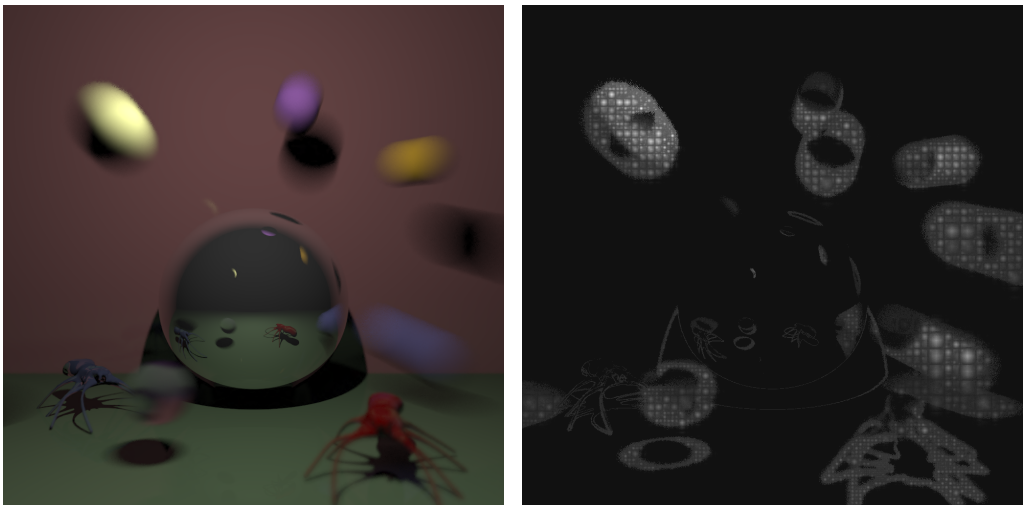


FIGURE 3.1: The image to the left, which contains both depth of field and motion blur effects, was rendered using the adaptive wavelet rendering algorithm using on average 36 samples per pixel. The image to the right shows the corresponding sample distribution.

3.1 Definitions and conventions

Before describing the algorithm some definitions and choice of methods need to be made. To begin with, the discrete wavelet transform (DWT) is used as described in section 2.4. In short this means transforming the images using the nonstandard decomposition with boundaries handled using symmetric extension with CDF 9/7 as the choice of wavelet basis. When transformed, the image is decomposed five times to create representations for levels 0-5 where level 0 refers to the original pixels and level 5 is the five times transformed image. Every color channel is transformed independently and, when transformed, every single scale and wavelet coefficient is saved (and updated when needed) throughout the rest of the program.

To render a single pixel means to evaluate the integral:

$$B(x,y) = \int_{y}^{y+1} \int_{x}^{x+1} \int_{\mathbf{s}} F(u,v,\mathbf{s}) d\mathbf{s} du dv \tag{3.1}$$

which is a compact version of the rendering equation where x and y are pixel positions, B is the pixel color and $\mathbf{s}$ compactly denotes all extra-dimensional effects. Evaluating the function F is the responsibility of the ray tracer and is usually done using Monte Carlo sampling in some way. The purpose of the sampling algorithm is to decrease the number of such evaluations that are needed. For this thesis a simple Whitted tracer is used to evaluate the equation above.

Theoretically the scale and wavelet coefficients are computed using the inner product with the scale and wavelet functions respectively as defined by equation 2.7 and 2.8. For notational convenience the inner product form will be used to denote the transform whereas, in practice, the DWT is performed. Thus, with $\tilde{B}$ being the pixel mean (color), the wavelet transform, for $1 \leq \alpha \leq 3$, is denoted:

$$c_{k,ij} = \langle \tilde{B}, \Phi_{k,ij} \rangle = \left\langle \int F d\mathbf{s}, \Phi_{k,ij} \right\rangle = \iiint F \cdot \Phi_{k,ij} d\mathbf{s} dx dy \tag{3.2}$$

$$d^{\alpha}_{k,ij} = \langle \tilde{B}, \Psi^{\alpha}_{k,ij} \rangle = \left\langle \int F d\mathbf{s}, \Psi^{\alpha}_{k,ij} \right\rangle = \iiint F \cdot \Psi^{\alpha}_{k,ij} d\mathbf{s} dx dy \tag{3.3}$$

Finally, there are parts in the algorithm where a color needs to be transformed into a single luminance value. There are many ways to do this and any reasonable weighting formula based on color perception should work fine. For this thesis the following is used:

$$L(B) = 0.2989 \cdot B_r + 0.5871 \cdot B_g + 0.1140 \cdot B_b \tag{3.4}$$

where $L(B)$ is the luminance value and r, g and b refers to the red, green and blue parts of the color B. The weights reflect the fact that humans tend to notice intensity variations in green considerably more than blue with red somewhere in between.

3.2 Sampling stage

The strategy is to determine the scale coefficient that should receive more samples at each iteration. If a level 0 coefficient is chosen it means sampling a specific pixel further. This should occur at sharp edges in order to capture the sharp variation in color. If, on the other hand, a level 5 coefficient is chosen samples will be spread out in a much larger area covering several pixels. This is preferable for smooth regions. To be able to compare scale coefficients, their variance will need to be estimated in some way. To this end we first define, for each pixel, the variance of the samples:

$$\sigma^2(F) = \frac{L(I_{max} - I_{min})^2}{L(I_{max} + I_{min})^r} \tag{3.5}$$

where $\sigma^2(F)$ is the variance, I_{max} and I_{min} are the sample colors with the maximum and minimum luminance values respectively, for that pixel, as defined by equation 3.4. L refers to equation 3.4 as well which effectively means that the sample variance will consist of one value per pixel. Finally r is a parameter which in the original paper by Overbeck et al. is set to 2. This makes the equation equivalent to the squared contrast metric used by Mitchell[13]. The idea behind the equation is that the numerator provides an upper bound estimate of the pixel variance, and the denominator weights the adaptive

sampler towards the darker image regions, where the human eye is more sensitive to error[1]. However, in scenes with local very bright and complex areas this can give a misleadingly low variance value. Therefore, for such scenes, the parameter r can be tuned to give a better representation. Having $r = 2.0$ is a good starting point and, if bright areas become too noisy, the value can be lowered but should most likely stay in the range 1.0 - 2.0 for almost every scene as it comes at the expense of darker regions instead.

With the variance of the samples defined, the estimator variance of the pixel mean becomes:

$$\sigma^2(\tilde{B}) = N^{-1}\sigma^2(F) \tag{3.6}$$

where N is the sample count for the relevant pixel. Thus, the variance of each pixel or, equivalently, each scale coefficient at level 0 can be computed. To get an estimate for higher level scale coefficients a low-pass DWT is performed on the level 0 variance values:

$$\sigma^2(c_{k,ij}) = \langle \sigma^2(\tilde{B}), \Phi^2_{k,ij} \rangle \tag{3.7}$$

Here $\Phi^2_{k,ij}$ means that the filter coefficients used are actually the squared filter coefficients of the normal transform. In the appendix of the Adaptive Wavelet Rendering paper this relation is motivated through a derivation[1]. Furthermore, through empirical testing, they found that the algorithm works best if the squared low-pass analysis coefficients (a filter) are renormalized to sum to 1.05 for the variance computations. Equation 3.7 and the renormalization both introduces bias into the computations. However, this only affects the priority computations and not the final image so it works well in practice. The renormalization value is another parameter that can be tuned. A higher value will favor higher level coefficients in the priority computations and vice versa. For certain scenes there might be reasons to favor one or the other to a higher degree. Finally note that there is no need to compute the corresponding wavelet coefficients since no synthesis is done for the variance.

With the variance computations explained the different parts of the sampling stage can now be detailed. Note that the algorithm works with a fixed sample count that needs to be decided beforehand.

Initialization

The scene is initially sampled with a low sample count per pixel (4 or 9 for example). This can be done for example by standard random or jittered sampling (see section 2.2). The mean color of each pixel is used to compute the full DWT (equations 3.2 and 3.3)

and the scale coefficients are put in a priority queue which is explained further below. Because the transform is applied to each color channel separately, the resulting scale and wavelet coefficients should be treated and stored as colors as well with three separate values. Finally the variance for the scale coefficients are computed and stored using equations 3.5 through 3.7.

Priority computation

A priority queue is used to decide which coefficient that needs more samples. Therefore it needs a priority function that correctly compares and chooses among many scale coefficients across six different levels. The function, $P(c_{k,ij})$ is given by:

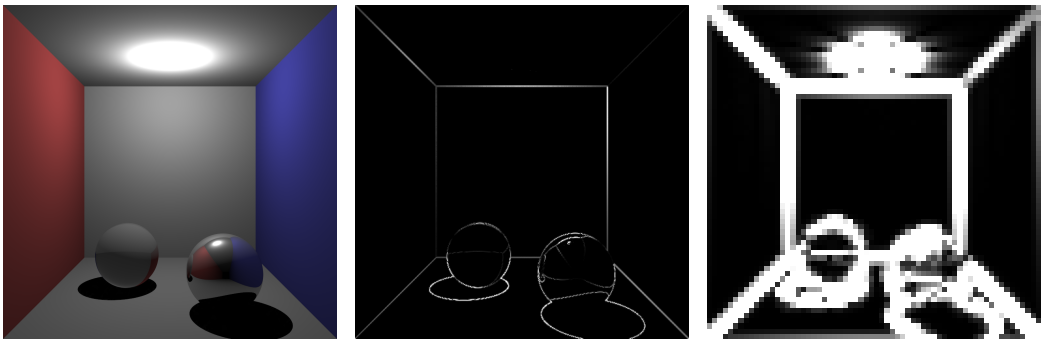
$$P(c_{k,ij}) = \sigma^2(c_{k,ij}) - D^2_{k,ij} \tag{3.8}$$

where $D^2_{k,ij}$ is the squared wavelet magnitude given by the squared mean of the intensity values for the three different wavelet coefficients:

$$D^2_{k,ij} = \frac{1}{3}\sum_{\alpha=1}^{3} L(d^{\alpha}_{k,ij})^2 \tag{3.9}$$

The motivation for the priority function is the following: Fine features and sharp edges in a scene will give high variance in pixels or small regions which will be apparent in the low-level scale variance coefficients. Similarly smooth regions will have high variance when observed over a larger area which will thus be captured by high-level scale variance coefficients. Both of these cases are therefore covered by the $\sigma^2(c_{k,ij})$ part. However, with the scale variance alone the fine scale features also heavily affect the course scale priorities which we do not want. Therefore, because wavelet coefficients are in a sense exactly the error (or detail) due to edges or other fine features, the wavelet magnitude can be subtracted from the variance to reduce the impact of sharp features on the priority computations for high-level coefficients. Figure 3.2 indeed shows that the wavelet magnitude increases the most along edges as the level increases.

Note that the function and motivation says nothing specific about the handling of smooth regions coupled with low-level coefficients which is mostly the case at the boundaries of smooth regions. The noise this results in is handled by the reconstruction stage (section 3.3).



(A) Rendered image (B) Level 1 (C) Level 4

FIGURE 3.2: The middle and rightmost images show the intensity of the wavelet magnitudes for the rendered image to the left. It is clear that edges result in greater values for the wavelet coefficients. It is also apparent that the magnitude increases along the edges as the hierarchy level increases. Note that the differences among the edges is a result of the luminance formula used (equation 3.4).

Sampling

At each iteration the highest priority scale coefficient is taken from the priority queue. Because the coefficients cover regions rather than pixels (aside from level 0 coefficients) several samples are allocated at each iteration. In Overbecks paper the amount per iteration is $64 \cdot 2^k$ samples where k is the level of the coefficient taken from the queue. Tuning the values offers trade-offs between speed and quality of adaptation.

The sampling in the image plane is done by importance sampling in the shape of the scale function (Figure 2.6a). This can be achieved by precomputing a partial distribution function (PDF) which is the norm of the scale function normalized to 1. The PDF is then used to compute the cumulative distribution function (CDF). Then, by uniformly sampling between 0 and 1, the CDF gives the corresponding sample locations to create the correct shape. Note that the x- and y-axis is importance sampled separately so only the one-dimensional CDF is needed. The resulting sample distribution is shown in Figure 3.3.

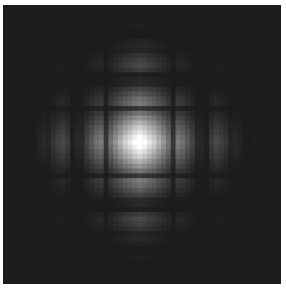


FIGURE 3.3: 2D sampling distribution using CDF 9/7

The priority queue gives the highest priority scale coefficient and, consequently, its level k and the location in the image that corresponds to the center of the coefficient. The importance sampling is then done over the pixel area covering the 9×9 coefficients of level $(k-1)$ that contributes in the calculation of the current coefficient. The reasoning behind this is that the amount of samples given to each area corresponding to a level $(k-1)$ coefficient

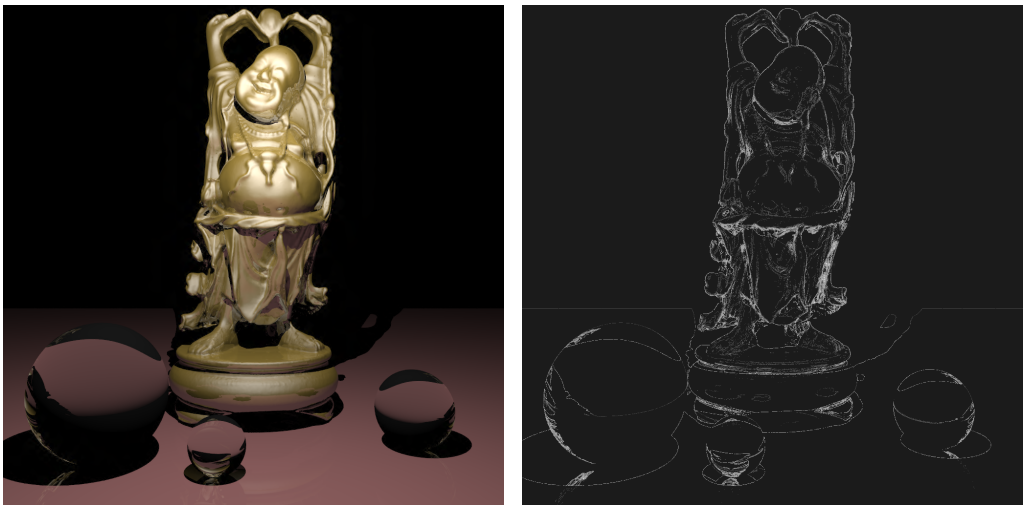


FIGURE 3.4: A scene without complex effects rendered with, on average, 9 samples per pixel. Without blurry regions the adaptive tracer can focus on edges and other small details as is evident in the sample distribution image.

is proportional to its contribution in the calculation of the current level k coefficient. This should thus work well to lower the variance of the coefficient and, therefore, the priority as well. After a round of sampling the coefficient is reinserted into the priority queue in case it needs more samples later. Note that level 0 coefficients do not need to be importance sampled. If such a coefficient is chosen it means that an individual pixel needs more samples which can be achieved by standard random or jittered sampling. Lastly, for dimensions other than in the image plane the sample locations/values are chosen via random sampling.

Iteration

At the end of the initialization, the scale coefficients are placed in the priority queue which marks the start of the first iteration. Each iteration then follows these steps:

1. The highest priority scale coefficient is taken from the priority queue and the scene is importance sampled.
2. The wavelet transforms for the image and variance are updated to reflect the information given by the new samples.
3. The scale coefficient is reinserted into the priority queue and the priority values for the coefficients affected by the new samples are updated.
4. If there are more samples left to distribute a new iteration begins by returning to step 1. Otherwise the sampling stage is completed and the algorithm moves on to the reconstruction stage.

Steps 2 and 3 deserve some additional clarification. In order to make the DWT updates efficient it is important that they are performed only locally over the affected image area after a round of sampling is complete. This requires storing additional information that is usually not kept when performing a DWT. In addition to storing every scale and wavelet coefficient we must keep track of every intermediate coefficient as well, that results from applying the transform in one dimension before applying it to the other. This applies to the variance updates as well with the difference that the final wavelet coefficients are not needed. Naturally, this requires some extra memory compared to performing a full transform each time. However, it also significantly improves the running time to the extent that it is not only an improvement but a necessity for the algorithm to be practical.

After each round of sampling the priority values need to be updated as well. For efficiency it is preferable if this is done only for the affected coefficients which means that a standard priority queue is not good enough. Instead an updatable priority queue is needed for which the priority values can efficiently be updated for any element in the queue while preserving correct ordering. There is more than one way to construct such a queue and any one is fine as long as it is fast.

3.3 Reconstruction stage

After the sampling stage the variance that remains in the image is mostly noise which should be removed. The noise will appear as fine-grained jump discontinuities in the image and should thus be captured by the wavelet coefficients. This fact is the key of the reconstruction stage and the strategy becomes to reduce the magnitude of such coefficients. To do this the standard deviations of the wavelet coefficients are needed (for every α) and computed similarly to the variance:

$$\sigma(d^{\alpha}_{k,ij}) = \sqrt{\langle \sigma^2(\tilde{B}), (\Psi^{\alpha}_{k,ij})^2 \rangle} \tag{3.10}$$

Before, the scale variance was of interest (equation 3.7) whereas now we want the wavelet variance instead. However, this still involves computing the former since the scale coefficients are needed to calculate the wavelet coefficients at each level. Effectively a full DWT is performed on the pixel variance with squared wavelet filters. The pixel variance itself differs as well in two ways where the first is that each color channel is treated separately this time. The second difference is that it should no longer be weighted specifically towards darker regions. The equation for the sample variance, per color channel, becomes:

$$\sigma^2(F) = (I_{max} - I_{min})^2 \tag{3.11}$$

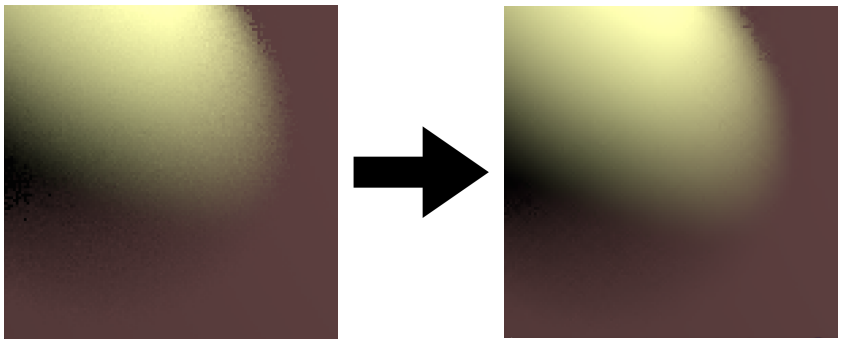


FIGURE 3.5: The reconstruction stage removes much of the noise that remains after the sampling stage and smoothens the image.

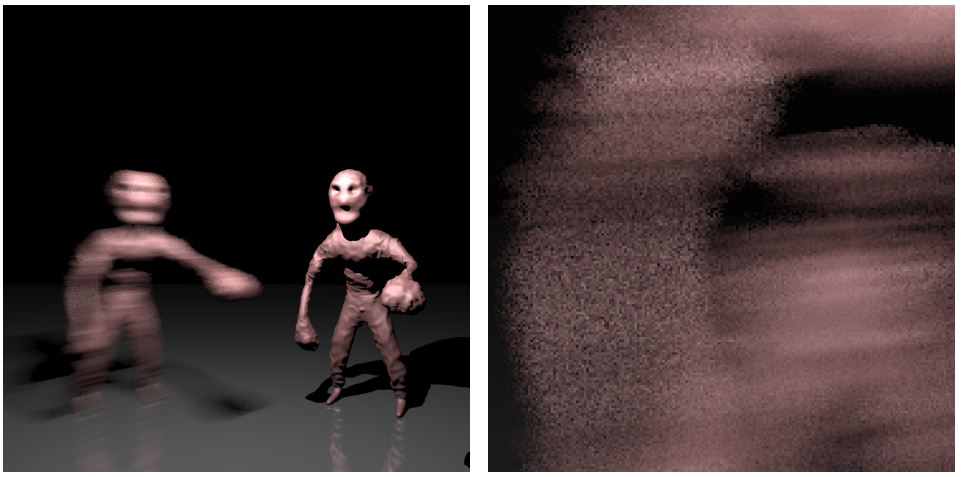
The estimator variance of the pixel mean is then calculated as before (equation 3.6) but for each channel as well. The standard deviation provides a range of statistically valid values for the wavelet coefficients. The wavelet coefficients are the expected values in the middle of the range and the strategy becomes to decrease their magnitude so that they instead take the smallest but still valid values. This is equivalent to choosing the smoothest image which fits the chosen rendering samples.

To achieve this, the standard deviations are subtracted from the wavelet coefficients giving the new coefficients $\tilde{d}^{\alpha}_{k,ij}$:

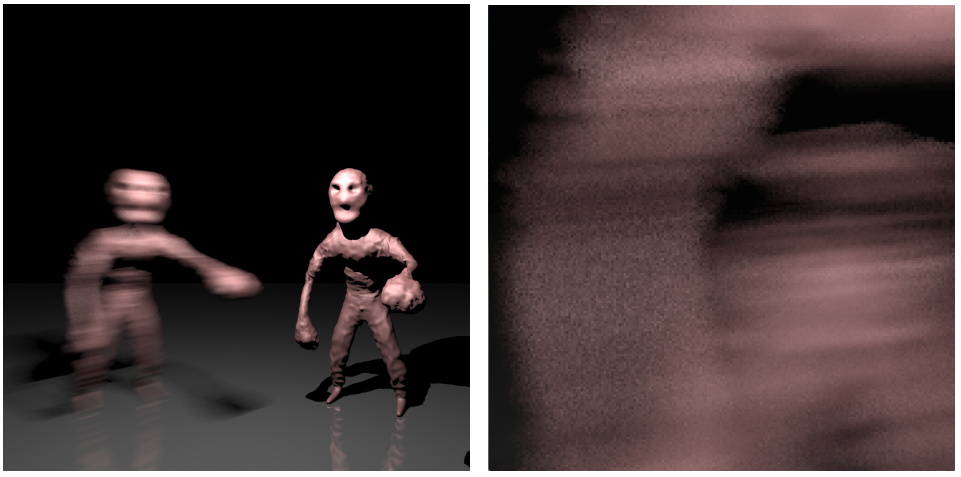
$$\tilde{d}^{\alpha}_{k,ij} = \text{sign}(d^{\alpha}_{k,ij}) \cdot \max\left(0, |d^{\alpha}_{k,ij}| - c_s \cdot \sigma(d^{\alpha}_{k,ij})\right) \tag{3.12}$$

where c_s is a constant parameter that provides a trade-off between noise and wavelet artifacts. Increasing the value creates a smoother image but risks introducing ringing around edges or blocky reconstruction. In Overbecks paper they found that setting it to 1.0 works well in most cases. In addition they also found that the filters for equation 3.10 should be renormalized to a value less than one, with $2^{-1/2}$ working well. This lessens the smoothing effect considerably as the level in the wavelet hierarchy increases which is what we want. The reason is that the noise will be apparent in the low-level coefficients meaning that it is those that should be targeted. Without the renormalization unwanted smoothing artefacts appear in blurry regions corresponding to high-level coefficients.

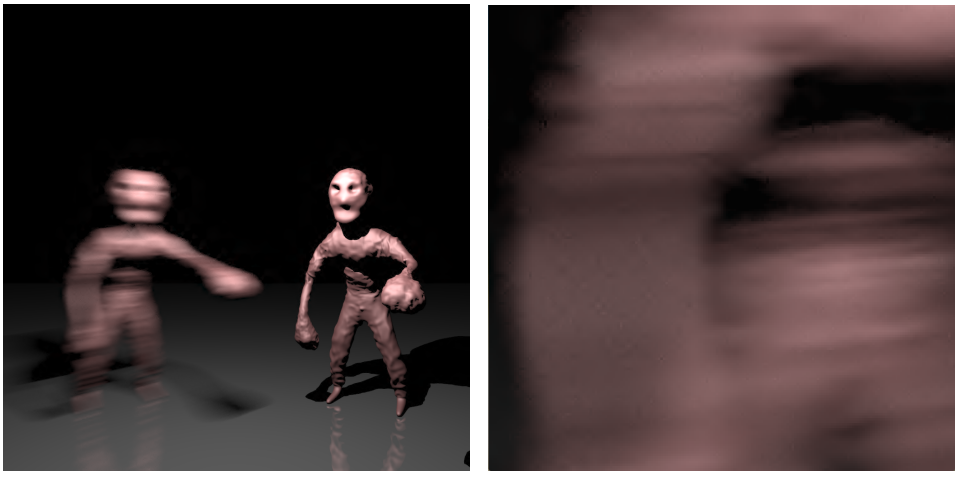
Finally, using the scale coefficients and the newly calculated wavelet coefficients, a full wavelet synthesis is performed. This creates the final image and thus ends the algorithm. The final result has very little noise and is well-sampled where needed in every relevant dimension, even with limited sample counts.



Jittered sampling: Samples per pixel: 25 Total runtime: 907 s



Jittered sampling: Samples per pixel: 100 Total runtime: 3632 s



Adaptive wavelet sampling: Samples per pixel on average: 25 Total runtime: 2945 s

FIGURE 3.6: Comparison between jittered sampling and adaptive wavelet sampling. The scene consists of two creatures, one still and one in motion but otherwise exactly the same. The left side images shows the rendered scene while the corresponging images to the right are zoomed in to show detail in the motion blur. The adaptive sampler clearly produces less noise even with a considerably lower sample count.

Chapter 4

Parallel Adaptive Wavelet Rendering

The adaptive wavelet rendering algorithm works well when implemented using efficient data structures and local updating. With the use of multithreading it can potentially become even better. However, the structure of the algorithm limits the potential. Wavelet transforms, priority updating and even the sample handling requires frequent synchronization and some sequential operations. As will become clear the major parts of the algorithm can all be done concurrently in some fashion. The question is if the limited amounts of operations and data handled in each thread sequence will be worth the added overhead, since threads will need to be stopped and launched again multiple times in each iteration.

In this chapter it is discussed how concurrency by multithreading can be used to potentially improve the algorithm. Firstly each individual part of the original algorithm is considered, if improvements seem likely and what potential problems there are. Following that, the alternative multi-coefficient version of the algorithm is presented which is slightly modified to allow for more concurrency at the price of some adaptiveness. Note that this chapter presents the ideas whereas the quantitative results of the changes are given in chapter 5. However, the qualitative differences between the original and modified algorithm are considered and compared in this chapter.

4.1 Concurrency in AWR

The suggested changes in this section does not affect the resulting images. The sample distributions and images produced will be qualitatively unchanged.

4.1.1 Initial sampling

The sampling done in the initialization stage is standard random sampling or jittered sampling, both of which works well to do in parallel. The only requirement is that the ray tracer used supports casting rays into the scene in parallel. The pixels are equally divided between the threads and each one processes its share of pixels. Since the samples sent out during the initialization stage typically accounts for at least ten percent of the total samples and sometimes considerably more, it will likely have a positive and noticeable impact on performance.

4.1.2 Adaptive sampling

As with the initial sampling the work can easily be divided between threads during the adaptive sampling phase. Instead of dividing the pixels, however, the samples are all sent into the same area as specified by the highest priority coefficient. Each thread calculates a position for each of its samples according to the importance sampling pattern. While this should work fine there is one concern. The sampling threads will have to be launched and stopped once per iteration and the only samples that can be distributed are those for the current round. Furthermore, for almost every scene, it will mostly be the lower level coefficients that are picked. With the standard parameters this means that in most rounds 64 or perhaps 128 samples will be distributed. Utilizing a quad-core processor fully, this means 16 or 32 samples per thread (8 or 16 with hyperthreading, see section 2.5.1). Is that enough samples to warrant the overhead of dividing the work and launching the threads?

4.1.3 Wavelet transforms

The natural method of performing wavelet transforms in parallel is to divide the rows and column at each level between the threads since each row of a wavelet transform can be computed independently of the others and the same applies to each column. However, every row transform must be completed before any respective column transform can start. Furthermore each level needs to be completely done before work can begin to compute the next. Otherwise the result will likely contain errors which will skew both the priority computations and the final synthesized image. This means, for a five level transform, that there are ten ”synchronization points” where the threads have to wait for each other before moving on. The wait will likely be short if work is divided evenly but ensuring synchronization comes with some inherent overhead.

The total amount of full wavelet transforms in one run of the algorithm is only four: One full analysis and one low-pass analysis (for the variance) at the end of the initialization stage and one full analysis for the variance and synthesis in the reconstruction stage. They will thus only account for a very small fraction of the runtime. Even if concurrency makes them considerably faster it will likely have negligible impact on the running time of the algorithm as a whole. Instead, it is the local transforms that are used in the adaptive sampling stage which are of interest. Being local, they recompute only the coefficients that could have changed in the current iteration. This is of course an important optimization but, unfortunately, it severely lessens the potential gain from concurrency. For example, in the extreme (but also most common) case where a level 0 coefficient or, in other words, a lone pixel is sampled, the local transform for the first level consists of part of one row and parts of nine columns.

Considering the limited data handled in one instance of the local transform, the synchronization required and the fact that the operations performed are simple, it is unlikely that concurrency will have a positive impact on the transforms.

4.1.4 Priority computations

At the end of the initialization stage the priority queue is filled with information about each coefficient and is later updated in each iteration of the adaptive phase. Just as for the local transforms only the affected coefficients are updated. While the priority calculations (eq. 3.8 - 3.9) can easily be done in parallel for the updates, the retrievals and updates of the priority queue accounts for much of the running time. Well-optimized concurrent priority queues are easily available through for example the Intel Threading Building Blocks library[14]. However, for it to be useful it has to be updatable as well which standard priority queues generally are not.

Being updatable and concurrent is a tricky combination but it can be achieved by, for example, combining a standard concurrent priority queue with another data structure that keeps track of the most recent values. A problem is that concurrent priority queues generally have to lock at least parts of the structure to preserve correctness when performing operations. If many threads tries to perform several operations rapidly the locking can force the operations to effectively become almost sequential. An alternative solution is to have one updatable priority queue per thread. This removes the need to lock the queues but instead requires either synchronization between the queues or some method of correctly handling duplicates with outdated priority values. Additionally retrieval of the highest priority coefficient requires some extra work since the top value

of each queue needs to be compared. Both solutions are tested in this thesis where the latter handles duplicates rather than fully synchronizing the priority queues.

4.1.5 Other components

The components of the algorithm that are not mentioned above each typically contributes to less than one percent of the total runtime. This applies to the pre-sampling setups, pixel color and variance updating and finally the reconstruction stage. While some of them might permit some concurrency they have not been investigated much in this thesis since the gain will be close to insignificant at best. Instead, focus has been placed on the major components. Note that a breakdown of the running time for the different components is included in the next chapter (table 5.1).

4.2 Modifying the algorithm

There exists some potential for parallelism in the adaptive wavelet rendering algorithm but it is limited as described in the previous section. By altering the algorithm slightly the potential can be significantly increased. However, with changes comes consequences, both for performance and image quality. This section presents one modification that is tested in the thesis and discusses its potential impacts.

4.2.1 Multiple coefficients per iteration

The modification is quite simple. Normally only one coefficient, the highest priority one, is handled per iteration. By instead handling the n highest priority ones at each iteration, where n is some appropriate integer, concurrency becomes more natural. The higher n is, the less adaptive the algorithm becomes. Samples will in general still be placed where they are needed but not necessarily where they are needed the most. On the other hand, threads are able to complete more work before being stopped and as a result the inherent overhead decreases. It thus becomes a trade-off between performance and quality. The question is if the net effect is positive which likely depends on the value of n. The properties of the algorithm using each possible value of n can be tested and analysed indefinitely. For the purpose of this thesis the testing is limited to $n = 1$, the original algorithm, and $n = 4$, to work nicely with quad-core processors.

4.2.2 Performance consequences

For the sampling stages the only difference when using multiple coefficients is the average number of samples sent in each iteration. With more samples per round the overhead gets smaller. Note that it is still preferable to split the sample count evenly for each thread rather than dividing up the coefficients per threads. This is because coefficients require different amounts of samples depending on their level which might result in some threads having more work to complete than others. Of course, the samples should still be distributed in the correct areas with the correct amount of samples in each.

When it comes to the wavelet transforms the concurrency potential is greater with multiple coefficients. By launching one thread per coefficient every local transform can be completed concurrently. However, with multiple coefficients the transform areas may overlap especially for the higher-level transforms. This can be handled in two ways:

1. It can be completely ignored. As long as the threads are synchronized, the overlap can only result in that some work might be repeated. Since the computations are simple it will likely not be very costly.

2. The coefficients that need to be updated can be pre-computed and divided evenly between the threads. This means that repeated work can be avoided. However, the synchronization requirement remains. Furthermore, pre-computing and excluding regions of overlap is quite complex and will need to be done for every level of the transform. Unfortunately, the pre-computations will most likely take longer than the entire local transform.

Based on the considerations above option 1 was chosen for this thesis. Some repeated work will almost certainly take less time than area overlap computations. Lastly the priority updating remains largely the same as in the single-coefficient case. Using multiple coefficients does not create new methods of handling the computations. The difference is that there will be more updates per iteration and fewer iterations in total which should decrease overhead.

However, there is one implementation detail worth noting: If the coefficients that should be updated are somehow marked during the local transformation stage and if overlaps are allowed for the transforms, then it is generally good to ensure that priority updates are not repeated as well. This does require extra work but unlike the area overlap computations it will likely save more time than it adds. Depending on how the extra work is done it might even be possible to do it concurrently which should reduce the extra time it adds to the runtime. Furthermore some of the priority calculations (before

insertions into the queue) can be completed together with the extra work which can further reduce its negative impact. These facts are all used in the implementation for this thesis.

4.2.3 Qualitative consequences

Picking multiple coefficients per iteration means fewer iterations and a broader targeting of coefficients. Figure 4.1 shows what effect this has on the sampling pattern. It gets less structured and high-level coefficients do not seem to get as many samples as they need which results in lower image quality for complex regions. However, this can be adjusted by changing some parameters. When sampling a specific scale coefficient the number of samples used, where k is the level of the coefficient and A and B are integers, is given by:

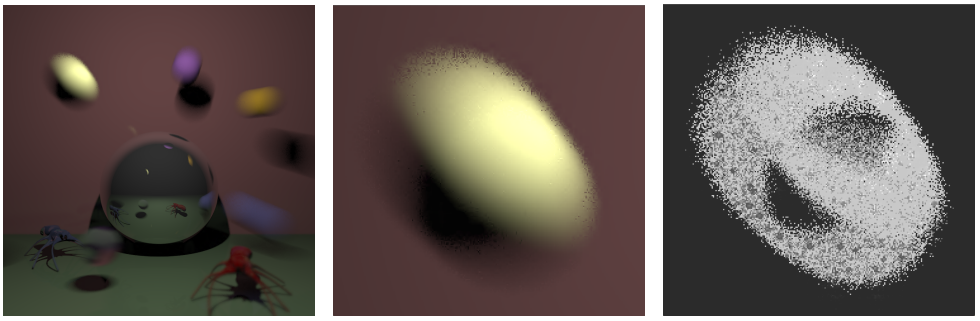
$$S(k) = A \cdot B^k \tag{4.1}$$

In the original paper $A = 64$ and $B = 2$[1]. The following applies to $S(k)$:

- Increasing either A or B leads to a shorter running time but less adaptive sampling as more samples will be sent out at each round, thus decreasing the number of iterations. Naturally the opposite applies when decreasing either A or B.
- Decreasing A is costly time-wise but will generally favor high-level coefficients without necessarily affecting low-level ones that much. This is because, for example, many level 0 coefficients (individual pixels) will not need 64 extra samples. Half of that can often be enough and since they are the most common coefficients many samples will be freed up for use by higher-level coefficients instead.
- Increasing B is naturally very beneficial for high-level coefficients. Just by going from 2 to 4, the level 5 coefficients receive a massive increase in samples while level 0 coefficients will not be affected at all.

Based on the above it should be possible to decrease A and increase B in order to counteract the trend of high-level coefficients not receiving enough samples while simultaneously ensuring that the running time is not affected too much. As shown in figure 4.1 it is indeed possible and the images produced are almost as good as when using one coefficient per iteration.

One coefficient per iteration. $S(k) = 64 \cdot 2^k$



Four coefficients per iteration. $S(k) = 64 \cdot 2^k$

Four coefficients per iteration. $S(k) = 45 \cdot 4^k$

Four coefficients per iteration. $S(k) = 32 \cdot 4^k$

FIGURE 4.1: Comparison between the original and modified algorithm. Four coefficients per iteration leads to less well-targeted sampling which becomes especially apparent in complex regions. However, with the right changes to $S(k)$ the difference gets smaller and the characteristic sampling grid reappears for complex blurry areas. Note that when the rendered image is viewed in full size there is a noticable quality difference between the top two versions whereas the first and fourth are almost indistinguishable without zooming. The third is almost as good but a difference can be seen.

Chapter 5

Results

This chapter presents the results relevant to understand how the concurrency changes affects performance for the adaptive wavelet algorithm. Parameters can be changed and scenes can be varied in infinitely many ways but the ones presented here were chosen to try to give the best representation possible. To begin with, the first section contains runtime breakdowns of a few different versions of the algorithm. Next, total runtimes are presented for four different scenes using different versions of the algorithm. Following that, further results are presented for the major components: sampling, wavelet transforms and priority updating. Lastly, the sampling distribution function $S(k)$ is varied to show how it affects the results.

All of the results were generated on a laptop with a 2.3 GHz Intel Core i7 processor which has four cores supporting hyperthreading. Every image was rendered at 1024×1024 and unless otherwise noted, $S(k) = 32 \cdot 4^k$. Note that the ray tracer that was used in conjunction with the sampling algorithm was a Whitted tracer, not heavily optimized, with fairly rudimentary motion blur and depth of field implementations. Thus, the tracing times are not particularly fast and, consequently, the same applies to the total rendering time. However, it is the relative differences between the AWR versions that are of importance, not the absolute values. Finally, every test was repeated three times and the results presented are the arithmetic means from those runs.

Many of the tables in this chapter contain, aside from runtimes, ratios that show the speedup gained when comparing two AWR versions. For testing the following four scenes were used:

Spheres scene: Shown in figure 3.1 on page 26, featuring both motion blur and depth of field effects.

Buddha scene: Shown in figure 3.4 on page 31, featuring no special effects.

Creature scene: Shown in figure 5.1 on this page, featuring motion blur.

Strands scene: Shown in figure 5.2 on page 45, featuring depth of field.

5.1 Breakdowns

Table 5.1 shows breakdowns of the running times for three different AWR versions, two of which makes use of concurrency. It is only intended as an example, using the creature scene, to show the relative contributions from the different components. These are the best implementations found in each case during the research. This means that not all components are necessarily concurrent in the parallel cases which explains why some running times are similar. The sections that follow more closely examines each major component.

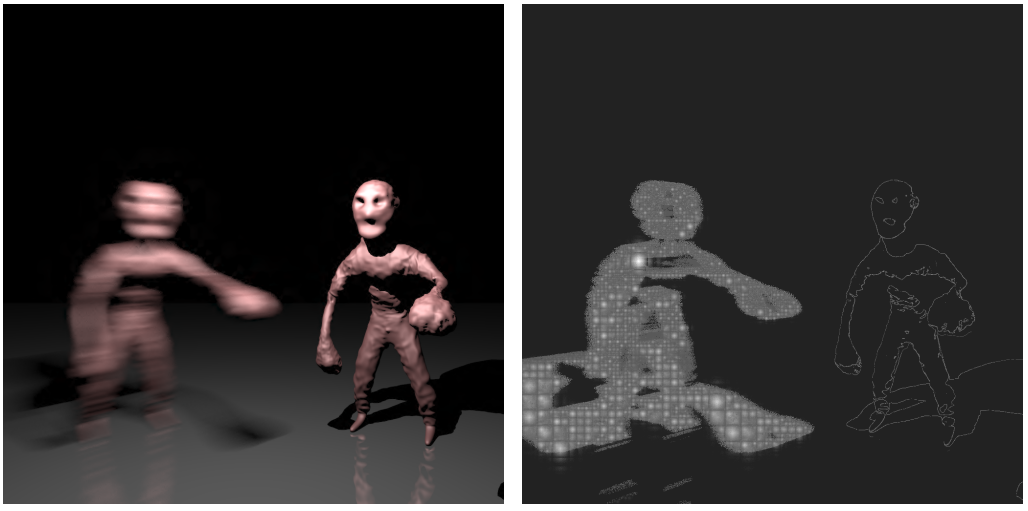


FIGURE 5.1: The scene, in which the creature to the left is in motion, was rendered using on average 25 samples per pixel. The sample distribution to the right nicely shows the difference in how the adaptive sampler handles sharp and blurry regions.

Component	Sequential	Parallel 1	Parallel 4
Initialization phase			
Sampling	316	160	160
Wavelet transforms	0.46	0.44	0.46
Priority computations	< 0.1	< 0.1	< 0.1
Adaptive phase			
Sampling setup	1.7	1.9	1.5
Sampling	2570	742	725
Data updating	0.15	0.85	0.66
Local transforms	43	43	26
Priority updating	8.3	8.7	9.2
///////////////			
Reconstruction	6.0	6.0	5.9
Total	**2945**	**964**	**929**

TABLE 5.1: Runtime breakdown in seconds for the creature scene using the best implementations of the original AWR algorithm, the single-coefficient parallel algorithm and the four-coefficient equivalent. The parameters used in every instance were exactly the same. For the adaptive phase entries, every value is the cumulative time after all iterations. Finally, note that the time spent to load object files, build the scene and save different images is not part of any of the categories, including **Total**, since they are not part of the actual ray tracing or the sampling algorithm.

5.2 Total runtimes

Table 5.2 contains the runtimes of the complete runs using the different AWR versions on the four different scenes. The table also shows the speedup ratios between the different algorithm versions. For full runtime breakdowns for every major test as in Table 5.1, see the appendix.

	Scenes			
AWR version	**Spheres**	**Buddha**	**Creature**	**Strands**
Sequential (s)	2521	1084	2945	2284
Parallel 1 (s)	819	345	964	963
Parallel 4 (s)	811	292	929	760
Speedup ratio Sequential / Parallel 1	3.08	3.14	3.05	2.37
Speedup ratio Sequential / Parallel 4	3.11	3.71	3.17	3.01
Speedup ratio Parallel 1 / Parallel 4	1.01	1.18	1.04	1.27

TABLE 5.2: Total runtime in seconds for the different scenes. Note that the time spent to load object files, build the scene and save different images is not included since they are not part of the actual ray tracing or the sampling algorithm. The last three rows shows speedup ratios between the different versions.

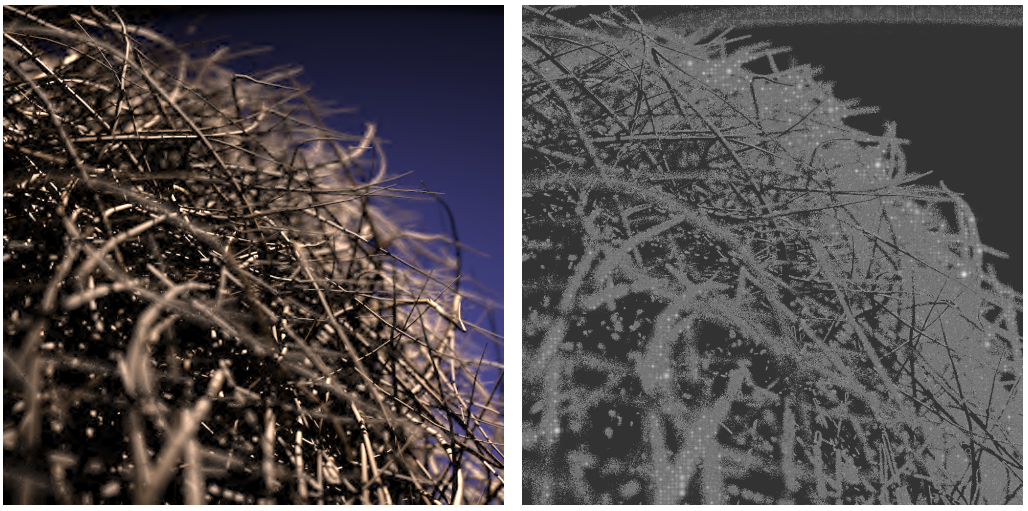


FIGURE 5.2: The scene contains a fairly complex object with a very noticable depth of field effect which also shows in the sample distribution. Around the focus plane the sampling is relatively sharp while it becomes gradually more diffuse closer or further from the camera in order to accurately capture the depth of field blurriness. The scene was sampled using on average 40 samples per pixel.

5.3 Sampling stages

Sampling occurs in the initialization stage, as jittered sampling, and as part of the adaptive iterations. Concurrent sampling is fairly straightforward and the running times, for the four different scenes, are shown in the tables below. Tables 5.3 and 5.4 contains the results for the initial and adaptive sampling respectively and also shows the speedup ratios. Note that while the number of coefficients used per iteration affects the adaptive sampling, it has no effect on the initialization stage. Therefore, the first table does not have individual results for different parallel versions.

	Scenes			
AWR version	**Spheres**	**Buddha**	**Creature**	**Strands**
Sequential (s)	527	344	316	516
Parallel (s)	153	86	160	127
Speedup ratio Sequential / Parallel	3.44	4.00	1.98	4.06

TABLE 5.3: Running time in seconds of the initial sampling for the different scenes. The final row shows the speedup ratio by using the parallel version compared to the sequential.

	Scenes			
AWR version	**Spheres**	**Buddha**	**Creature**	**Strands**
Sequential (s)	1896	690	2570	1554
Parallel 1 (s)	565	209	742	618
Parallel 4 (s)	559	175	725	512
Speedup ratio Sequential / Parallel 1	3.36	3.30	3.46	2.51
Speedup ratio Sequential / Parallel 4	3.39	3.94	3.54	3.04
Speedup ratio Parallel 1 / Parallel 4	1.01	1.19	1.02	1.21

TABLE 5.4: Cumulative running time in seconds of the adaptive sampling for the different scenes. The last three rows shows speedup ratios between the different versions.

5.4 Wavelet transforms

The wavelet transforms that are considered here are only the local transforms during the adaptive stage. The full transforms might gain more by using parallelism but since there are so few of them in a program run, they become unimportant in comparison. Furthermore, note that the runtimes presented are the cumulative runtimes for both the image transforms and the variance transforms added together. The reason is that, while the low-pass variance transform contains less calculations in total, every calculation that is done is exactly as for the image transform. Therefore, the changes for concurrency are exactly the same as well and there is no reason to compare them separately.

The first question is whether the local transform itself can be done efficiently in parallel. Table 5.5 shows a comparison between the sequential version and two parallel versions. The first one attempts to divide the work evenly between threads whereas the second uses the C++11 thread library *futures* and *async* functions to let the compiler itself decide which rows should be done in parallel and which should not.

	Scenes			
AWR version	**Spheres**	**Buddha**	**Creature**	**Strands**
Sequential (s)	73	36	43	173
Parallel thread (s)	147	80	99	294
Parallel async (s)	747	411	497	2098

TABLE 5.5: Comparison between sequential and two different parallel versions of the local wavelet transforms. The values are the cumulative runtimes after all iterations and only one coefficient was used per iteration for every test.

It is evident from Table 5.5 that the local transform performance is affected negatively by introducing concurrency. Therefore, the second question becomes if using multiple coefficients per iteration allows for more effective parallelization. Table 5.6 shows the results when using four coefficients and compares them to their sequential counterparts.

	Scenes			
AWR version	**Spheres**	**Buddha**	**Creature**	**Strands**
Sequential (s)	73	36	43	173
Parallel 4 (s)	67	17	26	83
Speedup ratio Sequential / Parallel 4	1.09	2.12	1.65	2.08

TABLE 5.6: Cumulative running time in seconds of the local wavelet transforms for the different scenes. The final row shows the speedup ratios between the sequential version and the four-coefficient parallel version.

5.5 Priority computations

The main contributor to the priority calculation runtime is the actual priority queue. Table 5.7 shows the results for the sequential version as well as for two methods for parallel updatable priority queues. The first method uses a single concurrent priority queue while the second method uses multiple standard ones functioning together as one. Note that both methods use some additional data structures to make them updatable.

	Scenes			
AWR version	**Spheres**	**Buddha**	**Creature**	**Strands**
Sequential (s)	14	6.3	8.3	33
Concurrent queue (s)	56	29	37	137
Multiple queues (s)	62	34	40	164

TABLE 5.7: Comparison between sequential and two different parallel methods of handling the priority computations. The values are the cumulative runtimes after all iterations and only one coefficient was used per iteration for every test.

Just as for the transforms, the parallel versions are inferior to the sequential version. Therefore, sequential priority handling is still used for the parallel algorithms. Furthermore, when using multiple coefficients, extra work needs to be done to avoid duplicate elements in the queue. Table 5.8 shows how this affects the results.

	Scenes			
AWR version	**Spheres**	**Buddha**	**Creature**	**Strands**
Sequential (s)	14	6.3	8.3	33
Parallel 1 (s)	15	6.5	8.7	35
Parallel 4 (s)	22	5.3	9.2	29
Speedup ratio Sequential / Parallel 1	0.93	0.97	0.95	0.94
Speedup ratio Sequential / Parallel 4	0.64	1.19	0.90	1.14
Speedup ratio Parallel 1 / Parallel 4	0.68	1.23	0.95	1.21

TABLE 5.8: Cumulative running time in seconds of the priority computations, including queue handling, for the different scenes. The last three rows shows speedup ratios between the different versions.

5.6 Varying the sampling distribution function

Every result presented so far belongs to runs with the same sampling distribution function, $S(k) = 32 \cdot 4^k$. It was chosen because it produces high quality images even when sampling four coefficients per iteration. However, if a slight decrease in quality is allowed in order to speed up the algorithm further while using the same total sample count, $S(k) = 45 \cdot 4^k$ can be used instead. Table 5.9 shows the breakdowns for the creature scene and Table 5.10 shows the total runtimes for the different scenes. These tables are equivalent to Table 5.1 and 5.2 with the only exception being the sampling distribution function that was used.

Finally, Table 5.11 and 5.12 shows the example breakdown and total runtimes respectively using $S(k) = 64 \cdot 2^k$. This is the sample distribution function used in the original paper by Overbeck and is primarily included for reference[1]. Note that using this function in combination with four coefficients per iteration produces results with more significant quality hits.

Component	**Sequential**	**Parallel 1**	**Parallel 4**
Initialization phase			
Sampling	317	160	160
Wavelet transforms	0.47	0.46	0.46
Priority computations	< 0.1	< 0.1	< 0.1
Adaptive phase			
Sampling setup	1.3	1.6	1.1
Sampling	2578	718	671
Data updating	< 0.1	0.71	0.56
Local transforms	38	38	22
Priority updating	7.2	7.4	7.7
//////////////			
Reconstruction	6.0	6.0	5.9
Total	**2948**	**932**	**870**

TABLE 5.9: Runtime breakdown in seconds for the creature scene with $S(k) = 45 \cdot 4^k$.

	Scenes			
AWR version	**Spheres**	**Buddha**	**Creature**	**Strands**
Sequential (s)	2628	1088	2948	2326
Parallel 1 (s)	781	306	932	817
Parallel 4 (s)	764	265	870	672
Speedup ratio Sequential / Parallel 1	3.36	3.56	3.16	2.85
Speedup ratio Sequential / Parallel 4	3.44	4.11	3.39	3.46
Speedup ratio Parallel 1 / Parallel 4	1.02	1.15	1.07	1.22

TABLE 5.10: Total runtime in seconds for the different scenes with $S(k) = 45 \cdot 4^k$.

Component	Sequential	Parallel 1	Parallel 4
Initialization phase			
Sampling	317	160	160
Wavelet transforms	0.46	0.45	0.45
Priority computations	< 0.1	< 0.1	< 0.1
Adaptive phase			
Sampling setup	1.1	1.2	1.0
Sampling	2567	747	743
Data updating	0.64	1.4	0.87
Local transforms	51	50	27
Priority updating	12	12	9.9
//////////////			
Reconstruction	6.0	6.0	6.0
Total	**2955**	**979**	**949**

TABLE 5.11: Runtime breakdown in seconds for the creature scene with $S(k) = 64 \cdot 2^k$.

	Scenes			
AWR version	**Spheres**	**Buddha**	**Creature**	**Strands**
Sequential (s)	2539	1085	2955	2351
Parallel 1 (s)	840	303	979	830
Parallel 4 (s)	814	277	949	700
Speedup ratio Sequential / Parallel 1	3.02	3.58	3.02	2.83
Speedup ratio Sequential / Parallel 4	3.12	3.92	3.11	3.36
Speedup ratio Parallel 1 / Parallel 4	1.03	1.09	1.03	1.19

TABLE 5.12: Total runtime in seconds for the different scenes with $S(k) = 64 \cdot 2^k$.

Note: For full breakdowns of every scene with every sampling distribution used in this thesis, see the appendix.

Chapter 6

Discussion

The previous chapters described the necessary background theory, the AWR algorithm and potential methods to improve it with the use of multi-core concurrency, followed by the results of those changes. Finally, this chapter will analyse those results and attempt to draw some conclusions, which is done in the first section. The second section then discusses possible future work to further enhance the algorithm.

Firstly it should be noted that there are many small implementation details that can affect the results, such as how to handle repeated calculations and duplicate markings when using multiple coefficients per iteration (see section 4.2.2). Alternative implementations are possible and they will possibly change the overall behaviour to some extent. However, the presented results in chapter 5 and the conclusions that will be drawn from them are such that it is unlikely that they would be affected significantly by minor method changes. Nevertheless, in many such cases different options were tested and the ones that generally seemed to work best were used when generating the results for the report.

6.1 Conclusions

To begin with, the total runtimes in Table 5.2, which in the end are the most interesting results, shows varying behaviour depending on the scene. In every case there is a significant improvement going from the sequential to the parallel versions, with the strands and buddha scenes seeing the worst and best improvements respectively. However, the differences between the parallel versions are more interesting. The spheres scene is barely improved at all by using four coefficients per iteration whereas the strands scene almost gets 30 % faster. To understand the differences the different components need

to be examined separately but before that there is an important fact to note: No test during the thesis work, whether part of the presented results or not, ever resulted in the multi-coefficient version taking more time than the single-coefficient counterpart. This might seem intuitive but it is not necessarily true. As is evident from the results there are components that can get a negative impact by using more coefficients. It is therefore encouraging that the results strongly seems to indicate that, as a whole, negative impact is rare if it exists at all, at least for the types of scenes used in the thesis with reasonable parameter choices.

The initial sampling, which is just jittered sampling with the image divided up into equal sections when performed in parallel, behaves as expected (Table 5.3). The spheres scene and especially the creature scene contains localized motion blur which requires extra computations when ray tracing. Thus, some sections require more work than others which means that some threads will have work left when others have finished. The other two scenes have more evenly distributed complexity and thus see a greater improvement. These issues can easily be fixed by improving the motion blur computations, by dividing up work differently or by letting threads help each other when they are finished. Since the initial sampling was not the main interest of the thesis, those optimizations had low priority and were left out in favor of other work. However, the sampling during the adaptive phase does handle the issue by ensuring that the work is split evenly between threads, even if the scene contains localized high complexity regions and multiple coefficients are sampled per iteration.

In section 4.1.2 a concern was raised whether enough samples are sent out at each iteration of the adaptive phase to warrant the overhead of dividing the work and launching the threads. The results in Table 5.4 shows that parallel sampling indeed works well even though the number of samples per thread in some iterations can be as few as four. Nevertheless, the results also seem to indicate that the overhead has some negative impact. Using four coefficients per iteration further speeds up the sampling but since the total amount of samples are the same, the only possible explanation for the speedup is that more samples are sent out per iteration, thus decreasing the amount of times work needs to be divided and threads needs to be launched. Furthermore, the scenes with motion blur prioritizes higher level coefficients to a much greater extent than the others. For higher level coefficients many samples are sent out even in the single-coefficient case (especially when the exponential part of the sampling distribution function has base 4). This should mean that the gain from using the multi-coefficient version should be less for such scenes which is exactly what the results show.

The next important component is the (local) wavelet transform. Firstly Table 5.5 shows the effect of parallel computations within the local transform for single coefficients. As

expected, the results are discouraging, even ridiculous in the async case. The reason is likely threefold:

1. The amount of work per thread is in general very small.
2. The work itself consists of calculations that are very simple for a computer and are thus completed very fast.
3. Threads have to wait for each other several times which introduces some overhead (see section 4.1.3).

The overhead introduced, in other words the division of work, the launching of threads or futures and the synchronization points, is greater than the time saved by doing the work in parallel meaning that the local transform, which is already an effective optimization compared to the full transform, is better left sequential.

When using a quad-core processor and four coefficients per iteration there is more work in total per iteration and it requires very little overhead to assign each coefficient to a processor core. Table 5.6 shows that it does give runtime improvements as opposed to the previous cases. The improvements are still not great but they are consistently better than their sequential counterparts. It is hard to pinpoint the exact reasons for the differences between the scenes but one aspect that likely has effect is how often the four coefficients belong to similar levels in the wavelet hierarchy. Work is not evenly spread between the threads if one coefficient covers a much larger area than the others. If this happens relatively often during a program run the runtime gains will be less than if they are often similar in size. Again, the coefficient prioritization in the motion blur scenes is to blame. With a higher probability of picking high-level coefficients, significant covering differences between the coefficients is more likely to occur as well. It might seem appealing to attempt to divide the work more evenly. However, that will almost certainly introduce overhead and as discussed above, overhead is very costly for the local transform. No reliable method to achieve better distribution was found in the course of the thesis work.

The final component of interest is the priority computation. Table 5.7 shows the results from tests using parallel updatable priority queues and it is clear that the parallel versions are inferior to the sequential version. The concurrent priority queue version most likely suffers from too many threads trying to insert elements at the same time. This requires the queue to constantly lock and unlock itself, effectively making the insertions sequential with the added overhead required to maintain a concurrent priority queue. In the multiple queue version it seems that the overhead of splitting the work when inserting and updating, as well as comparing elements from each queue when retrieving and correctly handling possible duplicate entries, just adds more time than it saves.

Table 5.8 shows the priority computation results for the finalized versions. Firstly, the single-coefficient parallel version is almost exactly the same as the sequential version and yet it is consistently a little bit slower. This is only an unwanted side effect of the slightly altered structure of the implementation to allow for parallelism in other areas. Since the relative difference is small it barely affects the run as a whole and is thus a minor issue. The four-coefficient version is more interesting showing both improved and worsened runtimes depending on the scene.

In the implementation for this thesis, coefficients are marked during the local transform with the information then sent to the priority handler in order to update the priority values for the marked coefficients. When using multiple coefficients, overlaps are allowed during the local transform which can result in repeated information sent to the priority handler. Since repeated priority calculations and insertions into the priority queue are highly unwanted, both time-wise and memory-wise, it is preferable to first remove repeated information from the "list" of updates. However, while this is beneficial for the runtime compared to letting duplicates remain, it does add some time itself meaning that, as a whole, the priority computation should still take somewhat longer when using the multi-coefficient version of the algorithm.

The worsened runtimes for the four-coefficient version thus have an explanation but what about the improved ones? Well, the scenes that get improved are those where sample regions are more spread out and not too focused on specific areas. This means less overlap and thus much fewer repeated marks. Furthermore, multiple coefficients means fewer iterations and instead more updates per iterations. This is preferable because it allows for more cohesive iteration without as much constructing and destructing of the data structures holding the information between rounds. In addition, while iterating to remove elements, the priority calculations (before insertion into the queue) can be done in parallel. This reduces the runtime impact of the duplicate removal task and works best when there are few duplicate entries to remove. Unfortunately, in the end, the performance change for the priority computation seems very scene dependent. However, the potential performance hit is relatively small compared to the gains for other components.

Lastly, there are many parameters that can be changed for the algorithm. However, the sampling distribution function is possibly the most interesting since it has such a direct impact on the multi-coefficient performance. The results in section 5.6 and in the appendix show that the trends discussed so far seems to hold even when varying the sampling distribution function. Note that the four-coefficient version with $S(k) = 64 \cdot 2^k$ has relatively poor performance quality-wise so deviations in those results are of little importance.

Depending on the application, the slight quality hit when using $S(k) = 45 \cdot 4^k$ might very well be worth it. Comparing Tables 5.2 and 5.10 shows that it is considerably faster than $S(k) = 32 \cdot 4^k$ for every scene for the parallel AWR versions.

In summary, it is not straightforward to make the adaptive wavelet algorithm concurrent. Its structure, with many iterations consisting of many small parts, limits the potential for parallelism. The main culprit, that prevents work to be cleanly divided as with for example jittered sampling, is the priority queue. Every iteration requires all coefficients to be compared and ordered based on their priority. This prevents coefficients to be split into groups where each group can be handled independently. In the end, for the original algorithm, only the actual ray tracing can be done effectively in parallel. In one way this speaks well of the algorithm as it shows that the different parts are constructed efficiently and fast enough to make even concurrency overhead problematic. On the other hand it means that substantial improvements are unlikely without fundamental modifications.

One such modification is the use of multiple coefficients per iteration. Not only does it make the actual sampling faster. It also enables parallel wavelet transforms and semi-parallel priority computations. The improvements are not great but they are significant enough to be of interest. Furthermore, the breakdown tables also show that the sampling setup and the data updating gets faster as well. This is not due to concurrency but rather because they are done fewer times. Their contribution to the runtime is minimal but this effect is still worth noting. Finally, while the multi-coefficient version is more sensitive to the choice of parameters, the quality can be kept close enough to the original single-coefficient version to render virtually indistinguishable images. These facts combined leads to the conclusion that the proposed multi-coefficient AWR algorithm is a worthwhile modification relevant for use and for further research.

6.2 Future work

In this thesis focus has been placed on the major parts of the algorithm but it is likely that the smaller parts can be made at least somewhat faster through concurrency as well. The sampling preparation and data updating should work well concurrently in the multi-coefficient case. While the local wavelet transforms worked poorly in parallel, the same is not necessarily true of the full transforms since there is much more data per thread to work with. The reconstruction stage would benefit from parallel full transforms and it might be possible to do some of the other computations there concurrently as well. However, improving these components can only yield very small improvements for

the algorithm as a whole, shaving off a few more seconds at best. If further significant improvements are possible, they must come through other strategies.

One possibility is to use other methods than multi-core programming for concurrency. There is for example SIMD processing which could work well for the wavelet transforms but does not seem particularly useful otherwise. More interesting is the potential use of a GPU for the computations. The priority queue will likely be problematic to handle but a well-implemented combined effort using the CPU and GPU would be an interesting development for the algorithm.

Another possibility would be to further modify the algorithm. Using multiple coefficients proved successful but one could imagine other changes. The results show that the sampling stages still accounts for most of the runtime. Therefore, any change that makes the algorithm smarter and thus decreasing the amount of samples needed is a good one. For example, the multi-coefficient version fully allows picking four coefficients where the smaller ones are successively contained in the larger ones. Granted, this might mean that the region truly needs many samples but it could also result in an unnecessary amount. Thus, it could be interesting to see how the image quality and the performance would be affected if that is actively prevented from happening. A perhaps even better modification would be if a sampling distribution function can be found that actually takes the variance for the relevant coefficient into account and adjusts the amount of samples accordingly.

The adaptive wavelet rendering algorithm is not structured to easily allow for parallelism. However, whether through other types of hardware concurrency or through further modification, there is still potential for further improvement. By making the algorithm better suited to todays highly parallel compute systems it will, in turn, remain relevant for the future of ray tracing.

Appendix A

Additional runtime breakdowns

The pages that follow contain tables showing the runtime breakdowns for every major test in this thesis. Note the following:

- The different tables differ only in scenes and sampling distribution functions where k refers to the wavelet hierarchy level. The different scenes has some parameter differences but parameters were otherwise kept constant.
- Each table contains the results using three different AWR versions: **sequential**, **parallel 1** and **parallel 4** where the number in the parallel cases shows the number of coefficients per iteration that was used. These are the best implementations found in each case during the research. This means that not all components are necessarily concurrent in the parallel cases which explains why some running times are similar.
- Each run was repeated three times and the arithmetic means are presented.
- For the adaptive phase entries, every value is the cumulative time after all iterations.
- The time spent to load object files, build the scene and save different images is not part of any of the categories, including **Total**, since they are not part of the actual ray tracing or the sampling algorithm.
- Breakdowns for the non-sequential runs in table 5.5 and 5.7 are not included since they do not differ from their sequential counterparts in any way other than what is shown in those tables.

A.1 Spheres scene

Shown in figure 3.1 on page 26, featuring both motion blur and depth of field effects.

Component	Sequential	Parallel 1	Parallel 4
Initialization phase			
Sampling	527	153	153
Wavelet transforms	0.47	0.46	0.46
Priority computations	< 0.1	< 0.1	< 0.1
Adaptive phase			
Sampling setup	3.2	3.5	1.2
Sampling	1896	565	559
Data updating	0.55	1.7	1.1
Local transforms	73	73	67
Priority updating	14	15	22
//////////////			
Reconstruction	6.1	6.0	6.1
Total	**2521**	**819**	**811**

$$S(k) = 32 \cdot 4^k$$

Component	Sequential	Parallel 1	Parallel 4
Initialization phase			
Sampling	526	151	149
Wavelet transforms	0.46	0.46	0.46
Priority computations	< 0.1	< 0.1	< 0.1
Adaptive phase			
Sampling setup	2.6	2.9	1.5
Sampling	2020	544	536
Data updating	0.35	1.4	0.90
Local transforms	62	63	53
Priority updating	11	12	17
//////////////			
Reconstruction	6.0	6.0	5.9
Total	**2628**	**781**	**764**

$$S(k) = 45 \cdot 4^k$$

Component	Sequential	Parallel 1	Parallel 4
Initialization phase			
Sampling	523	154	152
Wavelet transforms	0.46	0.47	0.46
Priority computations	< 0.1	< 0.1	< 0.1
Adaptive phase			
Sampling setup	2.9	3.2	1.5
Sampling	1899	568	594
Data updating	2.5	3.8	1.0
Local transforms	86	84	45
Priority updating	19	20	15
//////////////			
Reconstruction	6.0	6.0	6.0
Total	**2539**	**840**	**814**

$$S(k) = 64 \cdot 2^k$$

A.2 Buddha scene

Shown in figure 3.4 on page 31, featuring no special effects.

Component	Sequential	Parallel 1	Parallel 4
Initialization phase			
Sampling	344	86	87
Wavelet transforms	0.46	0.45	0.45
Priority computations	< 0.1	< 0.1	< 0.1
Adaptive phase			
Sampling setup	0.72	0.80	0.36
Sampling	690	209	175
Data updating	< 0.1	0.32	0.17
Local transforms	36	36	17
Priority updating	6.3	6.5	5.3
//////////////			
Reconstruction	6.0	5.9	6.0
Total	**1084**	**345**	**292**

$$S(k) = 32 \cdot 4^k$$

Component	Sequential	Parallel 1	Parallel 4
Initialization phase			
Sampling	345	89	88
Wavelet transforms	0.46	0.46	0.46
Priority computations	< 0.1	< 0.1	< 0.1
Adaptive phase			
Sampling setup	0.53	0.58	0.28
Sampling	706	179	154
Data updating	< 0.1	0.25	0.15
Local transforms	27	26	12
Priority updating	4.6	4.7	3.8
//////////////			
Reconstruction	6.0	6.1	5.1
Total	**1088**	**306**	**265**

$$S(k) = 45 \cdot 4^k$$

Component	Sequential	Parallel 1	Parallel 4
Initialization phase			
Sampling	346	86	87
Wavelet transforms	0.46	0.44	0.45
Priority computations	< 0.1	< 0.1	< 0.1
Adaptive phase			
Sampling setup	0.44	0.48	0.23
Sampling	710	186	172
Data updating	< 0.1	0.23	0.17
Local transforms	19	19	8.8
Priority updating	3.3	3.4	2.6
//////////////			
Reconstruction	6.0	6.0	5.9
Total	**1085**	**303**	**277**

$$S(k) = 64 \cdot 2^k$$

A.3 Creature scene

Shown in figure 5.1 on page 43, featuring motion blur. Note: These are the same tables as presented in chapter 5.

Component	Sequential	Parallel 1	Parallel 4
Initialization phase			
Sampling	316	160	160
Wavelet transforms	0.46	0.44	0.46
Priority computations	< 0.1	< 0.1	< 0.1
Adaptive phase			
Sampling setup	1.7	1.9	1.5
Sampling	2570	742	725
Data updating	0.15	0.85	0.66
Local transforms	43	43	26
Priority updating	8.3	8.7	9.2
//////////////			
Reconstruction	6.0	6.0	5.9
Total	**2945**	**964**	**929**

$$S(k) = 32 \cdot 4^k$$

Component	Sequential	Parallel 1	Parallel 4
Initialization phase			
Sampling	317	160	160
Wavelet transforms	0.47	0.46	0.46
Priority computations	< 0.1	< 0.1	< 0.1
Adaptive phase			
Sampling setup	1.3	1.6	1.1
Sampling	2578	718	671
Data updating	< 0.1	0.71	0.56
Local transforms	38	38	22
Priority updating	7.2	7.4	7.7
//////////////			
Reconstruction	6.0	6.0	5.9
Total	**2948**	**932**	**870**

$$S(k) = 45 \cdot 4^k$$

Component	Sequential	Parallel 1	Parallel 4
Initialization phase			
Sampling	317	160	160
Wavelet transforms	0.46	0.45	0.45
Priority computations	< 0.1	< 0.1	< 0.1
Adaptive phase			
Sampling setup	1.1	1.2	1.0
Sampling	2567	747	743
Data updating	0.64	1.4	0.87
Local transforms	51	50	27
Priority updating	12	12	9.9
//////////////			
Reconstruction	6.0	6.0	6.0
Total	**2955**	**979**	**949**

$$S(k) = 64 \cdot 2^k$$

A.4 Strands scene

Shown in figure 5.2 on page 45, featuring depth of field.

Component	Sequential	Parallel 1	Parallel 4
Initialization phase			
Sampling	516	127	127
Wavelet transforms	0.46	0.46	0.46
Priority computations	< 0.1	< 0.1	< 0.1
Adaptive phase			
Sampling setup	0.66	0.87	0.87
Sampling	1554	618	512
Data updating	0.11	1.6	1.1
Local transforms	173	173	83
Priority updating	33	35	29
//////////////			
Reconstruction	6.0	6.0	6.0
Total	**2284**	**963**	**760**

$$S(k) = 32 \cdot 4^k$$

Component	Sequential	Parallel 1	Parallel 4
Initialization phase			
Sampling	516	127	126
Wavelet transforms	0.45	0.46	0.46
Priority computations	< 0.1	< 0.1	< 0.1
Adaptive phase			
Sampling setup	0.55	0.67	0.67
Sampling	1658	535	461
Data updating	< 0.1	1.3	0.99
Local transforms	122	122	56
Priority updating	23	24	20
//////////////			
Reconstruction	6.0	6.0	5.9
Total	**2326**	**817**	**672**

$$S(k) = 45 \cdot 4^k$$

Component	Sequential	Parallel 1	Parallel 4
Initialization phase			
Sampling	518	126	126
Wavelet transforms	0.43	0.53	0.51
Priority computations	< 0.1	< 0.1	< 0.1
Adaptive phase			
Sampling setup	0.43	0.53	0.51
Sampling	1705	573	499
Data updating	< 0.1	1.3	1.1
Local transforms	102	102	49
Priority updating	19	20	17
//////////////			
Reconstruction	6.0	6.0	6.0
Total	**2351**	**830**	**700**

$$S(k) = 64 \cdot 2^k$$

Bibliography

[1] Ryan S. Overbeck, Craig Donner, and Ravi Ramamoorthi. Adaptive Wavelet Rendering. *ACM Transactions on Graphics (SIGGRAPH Asia 09)*, 28(5):1–12, 2009. URL `http://www1.cs.columbia.edu/~rso2102/AWR/`.

[2] Toshiya Hachisuka, Wojciech Jarosz, Richard Peter Weistroffer, Kevin Dale, Greg Humphreys, Matthias Zwicker, and Henrik Wann Jensen. Multidimensional adaptive sampling and reconstruction for ray tracing. *ACM Transactions on Graphics (SIGGRAPH 2008)*, pages 1–10, 2008. URL `http://www.ci.i.u-tokyo.ac.jp/~hachisuka/mdas.pdf`.

[3] Eric J. Stollnitz, Tony D. Derose, and David H. Salesin. *Wavelets for Computer Graphics : Theory and applications*. Morgan Kaufmann, San Francisco, 1996.

[4] Jpeg 2000 specification. `http://www.jpeg.org/jpeg2000/index.html`. Accessed: 2014-11-03.

[5] E. Wes Bethel, Hank Childs, and Charles Hansen. *High Performance Visualization: Enabling Extreme-Scale Scientific Insight*. Chapman and Hall/CRC, November 2012. This book is a collection of some of the most seminal work in the field of Visualization and Analysis tools.

[6] D.Černá, V.Finěk, M.Gottfried, P.Hübnerová, S. Paulusová, J. Róža, and L. Viščur. Boundary artifact reduction in wavelet image compression. *Numerical analysis and applied mathematics, AIP Conference Proceedings*, 1281:1–11, 2010. URL `http://dsp.vscht.cz/konference_matlab/MATLAB09/prispevky/019_cerna.pdf`.

[7] Matlab wavelet framework: Border effects. `http://www.mathworks.se/help/wavelet/ug/dealing-with-border-distortion.html`. Accessed: 2014-11-06.

[8] A. Cohen, I. Daubechies, and J. C. Feaveau. Biorthogonal bases of compactly supported wavelets. *Communications on Pure and Applied Mathematics*, 45(5): 485–500, June 1992. URL `http://www.math.duke.edu/~ingrid/publications/CPAM_1992_p485.pdf`.

[9] Anthony Williams. *C++ Concurrency in Action*. Manning Publications, 2012.

[10] Michael E. Thomadakis. The architecture of the nehalem processor and nehalem-ep smp platforms. Technical report, Texas A&M University, March 2011. URL `http://sc.tamu.edu/systems/eos/nehalem.pdf`.

[11] Concurrency hazards: False sharing. `http://www.codeproject.com/Articles/51553/Concurrency-Hazards-False-Sharing`. Accessed: 2015-01-22.

[12] Ryan S. Overbeck, Craig Donner, and Ravi Ramamoorthi. Adaptive wavelet rendering. Presentation, 2009. URL `http://www1.cs.columbia.edu/~rso2102/AWR/`.

[13] Don P. Mitchell. Generating antialiased images at low sampling densities. *ACM SIGGRAPH 87*, 21(4):65–72, July 1987.

[14] Intel threading building blocks. `https://www.threadingbuildingblocks.org`. Accessed: 2015-01-28.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTATIONSDAG 2015-03-20

EXAMENSARBETE Multi-coefficient Parallel Adaptive Wavelet Rendering
STUDENT Robin Somers
HANDLEDARE Michael Doggett (LTH)
EXAMINATOR Tomas Akenine-Möller (LTH)

Photorealistic rendering using signal processing techniques

POPULÄRVETENSKAPLIG SAMMANFATTNING **Robin Somers**

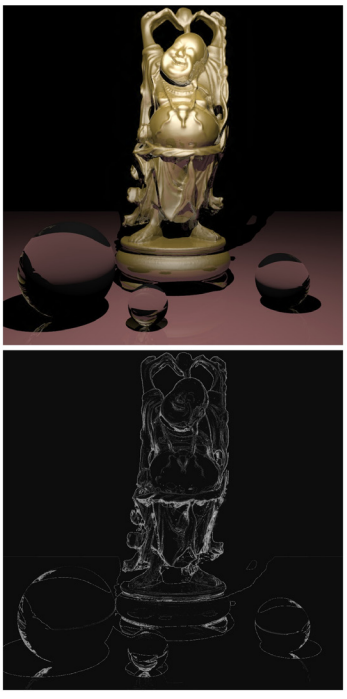
It is a notoriously slow process to create complex photorealistic images. However, by using the power of multiple cores in modern processors, one already efficient method has now been altered to become even faster.

Photorealistic rendering is used today for many different applications, such as for movies, art, advertising and games. It differs from real-time rendering, used for interactive graphics, in that it gives more physically accurate results. In other words it looks more real but it comes at the cost of time. In fact, one such image can take minutes, even hours, to render depending on the complexity of the scene it shows. Consider what this means for a movie containing many visual effects. Every second typically contains at least 24 still images. If each image takes roughly 20 minutes to render and there are 60 minutes containing visual effects in the movie, this means that the total rendering time will be close to 30 000 hours! In practice, to make the time reasonable, giant clusters of computers are used to render several images simultaneously. Nevertheless, any decrease in rendering time can have a profound effect, shaving off thousands of hours. It is no big surprise that much research has gone into doing just that.

The so-called Adaptive Wavelet Rendering method was proposed a couple of years ago and works relatively well. However, in the thesis it is shown how it can be improved by letting the computer perform multiple tasks in parallel rather than just one at a time. This is achieved by using multi-core processing where each processor core can work independently but still in collaboration with the others. The result is that the method works two to three times faster! Not only that, but by slightly modifying the method to work on several regions of an image at the same time it can become up to 30 % faster still! In fact, with the modification, the image below is rendered in only a quarter of the time compared to when using a single processor core. Not every scene gets such a substantial improvement but it never gets worse either. The Adaptive Wavelet Rendering method is just one among many but by parallelizing it, it becomes faster and better suited to todays highly parallel compute systems.

Adaptive Wavelet Rendering

The method uses the concept of wavelets, used in signal processing, to redistribute the pixel information in a not yet completed image. This is done multiple times where each creates a new layer of information. The trick is that these layers are constructed to easily show which parts of an image that need more work and which that are good as they are. Therefore areas with greater complexity can be identified and worked on to a greater extent whereas unnecessary work is avoided for simpler parts of a scene. By being able to target where work is needed much time is saved. The top image to the right shows a rendered image and the bottom image shows how the method has interpreted the complexity, brighter meaning more complex.



Rendering example