Abstract

Post-processing techniques are used to change a rendered image as a last step before presentation and include, but is not limited to, operations such as change of saturation or contrast, and also more advanced effects like depth-of-field and tone mapping.

Depth-of-field effects are created by changing the focus in an image; the parts close to the focus point are perfectly sharp while the rest of the image has a variable amount of blurriness. The effect is widely used in photography and movies as a depth cue but has in the latest years also been introduced into computer games.

Today's graphics hardware gives new possibilities when it comes to computation capacity. Shaders and GPGPU languages can be used to do massive parallel operations on graphics hardware and are well suited for game developers.

This thesis presents the theoretical background of some of the recent and most valuable depth-of-field algorithms and describes the implementation of various solutions in the shader domain but also using GPGPU techniques. The main objective is to analyze various depth-of-field approaches and look at their visual quality and how the methods scale performance wise when using different techniques.

Acknowledgments

We would like to give huge thanks to Torbjörn Söderman and Johan Andersson, our supervisors at DICE, for their feedback and invaluable discussions. Thanks to the tools and technology team for the great amount of inspiration and support. Thanks to our academic supervisor Matt Cooper and our opponents for their opinions and valuable feedback on the report. Great appreciations to Mark Harris at NVIDIA for showing interest and giving advice and pointers regarding graphics hardware. Finally, many thanks to our families and friends for always being there.

Contents

List of Figures

List of Tables

List of Abbreviations

2D	Two Dimensions
3D	Three Dimensions
ADI	Alternating Directions Implicit
ALU	Arithmetic Logic Unit
API	Application Programming Interface
CAL	Compute Abstraction Layer
CDPPL	CUDA Data Parallel Primitives Library
COC	Circle Of Confusion
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DOF	Depth Of Field
DX10	DirectX 10
DICE	Electronic Arts Digital Illusions CE AB
GPU	Graphics Processing Unit
GPGPU	General-Purpose computation on GPUs
HLSL	High Level Shading Language
MPAD	Multi-Passed Anisotropic Diffusion
SAT	Summed-Area Table
SDK	Software Development Kit
SSD	Separable Simulated Diffusion

Chapter 1

Introduction

The purpose of this first chapter is to introduce the reader to the thesis. It will provide a heads up start about what is coming. First the problem is described and stated. Secondly we summarize the major objectives of the thesis, what we strive to achieve. This is followed by the report outline which explains how the report is structured. The last part presents recommended prerequisites that we believe are necessary to fully understand the content of the thesis.

1.1 Problem Description

Today's games deliver outstanding visual experiences to the player and developers like Electronic Arts Digital Illusions CE AB (DICE) always strive to push the limits of the hardware. New visual effects, more game content and fresh challenges are all among the things that gamers of today expect when playing new games.

The game industry has always been one of the fields that have pushed the computer graphics visuals to the next level, which is almost always done with the goal to mimic reality of the world and to produce movie-like results. Something that has been missing for a long time is certain visual cues that traditionally have been too complex and time consuming to accurately simulate in real time. Because of this, effects like Depth-Of-Field (DOF), that imitates a camera lens and presents the reality in a human-like perspective, have so far been limited to the movie industry. In the case of DOF this is usually a matter of making objects at certain depths appear blurry while keeping sharpness where the focus is.

The blurriness could clearly be seen as a demolisher of other special effects but it also provides depth cues that are important to the viewer and can guide the player to pay attention to certain areas. This property is also something that is widely used and accepted in the field of photography.

With proper post-processing, DOF effects and many others can be achieved with simpler methods than the ones being physically correct. Unfortunately, even with simpler models, post-processing can be both complex and computationally costly. Therefore, more advanced post-processing effects such as DOF and bloom

filters are missing in most games. However, with higher demands and more computing power this is becoming closer to reality.

During recent years Graphics Processing Units (GPU) have been used more and more to meet these high demands. The GPU can outrace the Central Processing Unit (CPU) when it comes to computing power and is much more suited for most kinds of parallel computations. However, using the standard rendering methods for post-processing does not utilize the full potential of the available power and feature set that modern GPUs have. General-Purpose computation on GPU (GPGPU) techniques using custom GPGPU languages such as NVIDIA's Compute Unified Device Architecture (CUDA) or ATI's Compute Abstraction Layer (CAL) have a much better mapping of what the hardware is actually capable of. GPGPU languages therefore provide possibilities to create much more advanced post-processing effects than it would be possible to do, using the standard high level shading languages.

1.2 Thesis Objectives

The scope of this thesis is to implement advanced post-processing filters such as DOF using High Level Shading Language (HLSL), CUDA and possibly other GPGPU languages. The theory part involves in-depth studies of post-processing algorithms and how they can be applied to the GPGPU space. Furthermore, performance between different algorithms and with different techniques is to be examined.

In addition, our objective is to implement and investigate how these new techniques can be used to improve post-processing in the computer games industry. Problems like what methods are suitable, and the tradeoff between quality and performance will be further evaluated.

1.3 Outline of Report

Following this introductory chapter will be chapter 2, which describes the background, i.e., gives an introduction to the image processing field, and the use of post-processing in computer graphics. In addition, we introduce the GPU and give some examples of previous work. Chapter 3 goes into more details about camera models and DOF, how it is perceived and methods for simulating the effect in computer graphics. Discussed in chapter 4 is the theory of CUDA and in chapter 5 we explain our implementations and how we use the theory from chapter 3 and 4 to produce different DOF simulations. Finally, our results are presented in chapter 6 and chapter 7 adds discussion and conclusions. The last chapter also brings forth ideas about future improvements and possibilities.

1.4 Reader Prerequisites

To fully understand and appreciate the content of this thesis, good knowledge and understanding of image processing, computer graphics, and fundamental photography is highly recommended. It is also beneficial to have an understanding of computer games.

Chapter 2

Background

In this chapter we present the area of image processing and techniques related to post-processing. The first section gives a brief introduction to digital images and what processes can be applied. Secondly we discuss how post-processing is done and its role in computer graphics. Next we give a concise presentation of the GPU and continue with how it can be used for both visual effects and more general purposes. We conclude with various previous works in the DOF area, that are important for the thesis.

2.1 Digital Image Processing

Digital image processing refers to manipulation and processing of digital images using a computer. A digital image is composed of a finite number of image elements, also called *pixels*, which describe the intensity and color of the image. A digital image can come from many sources. It could be a picture taken with a digital camera, or a photograph or painting that has been scanned into digital form, or in our case images from a rendered three dimensional (3D) scene.

Dimensions of an image are measured in pixels that correspond to the width and height of an image. For color images these pixels contains three or four channels. One channel for each of the colors red, green and blue. If a fourth channel is present, it typically contains alpha/transparency values.

When talking about image processing there are a number of processes that can be pointed out to give basic understanding of the field [4].

- *Image acquisition* is the first process that we have to consider and deals with the origin of a digital image. Generally, the acquisition of an image also involves some pre-processing, for example scaling or rotation.

- *Image enhancement* is probably the most used and appealing area. The idea here is to highlight important details to enhance the visual experience of an image. An example of this is adjustment of contrast or brightness. This stage is naturally very subjective.

- *Image restoration* is used to rescue and restore damaged images. This also deals with image enhancement, but is not subjective in the same matter. Instead, restoration deals with minimizing the amount of degradation by relying on mathematical or probabilistic models as a measurement.
- *Color image processing* has been gaining importance as the use of digital photography and Internet has increased. Methods of how to tamper with color images, to enhance and extract features, is of great interest since almost all images now appear in color.

There are of course many more examples of how to alter images, such as morphing, compression, segmentation and recognition. However, for the purpose of this thesis those processes are of lesser importance.

2.2 Post-processing

Post-processing is a type of image processing with the difference that there is in most cases additional information available, for example scene depth values. Adjusting and changing an image can highly affect how the output is percepted and is therefore important in the fields of computer graphics.

The post-processing step takes place after an image is acquired either from a 3D rendering or from another source. The step includes but is not limited to simple special effects such as grayscale, simple blur, saturation, contrast, and image morphing but also more advanced effects like DOF, tone mapping, motion blur, and bloom filters.

When talking about post-process rendering, we usually do not have the same resources as during the normal rendering pass. Typically there is no sense of objects or lights, instead we have plain images that describe the scene. The post-processing step does generally not intervene with the ordinary rendering pipeline, rather is it executed as a separate image processing step.

For more simple filters in the post-processing pass this is usually not a problem, since they are mostly concerned about the color image that is produced after the 3D-scene is rendered. For the more advanced filters this pass can require additional information about the scene, for example the depth. A rendered example with both color and depth information can be seen in figure 2.1. The additional information can be output in different buffers during the rendering pass for later use when the post-process takes place.

2.3 Graphics Processing Unit

To meet the high demands of today's computer graphics the GPU plays an important role. This special processor is built and optimized for the parallel data operations that computer graphics makes use of [2].

In computer graphics many of the visual effects are nowadays calculated on the GPU. This is usually suitable since the GPU works very well with parallel computing and is especially fast if the work area is small.

Figure 2.1. Depth and color. Linear grayscale depth map on left (white is far away), color image for the same scene on the right.

For some time the graphics pipeline has been programmable to the developer, which gives additional processing power and allows for more advanced rendering effects. This ability to control the details of the graphics pipeline is called "the programmable graphics pipeline", in contrary to the traditional "fixed-function pipeline" that did not really offer any control to the programmer.

2.3.1 The Programmable Graphics Pipeline

In order to control the graphics pipeline the developer needs to write so-called shader programs that are uploaded to the GPU memory. Usually these shader programs are written in a special shader language. The three most important ones are Microsoft's HLSL, NVIDIA's Cg and GLSL that comes from the open source community that also created OpenGL.

Despite the fact that all of them are C-like and offers essentially the same functionality they support different levels of programmability and they are also used with different graphics software libraries. HLSL is used with Microsoft's own DirectX library whereas the two latter are utilized with the OpenGL, an open graphics library.

Usually there are two types of programs: vertex shaders and fragment shaders. The vertex shader handles geometry data, and has the ability to modify positions, normal vectors, and other vertex properties. The other type is the fragment shader which works per-pixel and can adjust values such as color and opacity.

Nowadays, a third type of shader is also occurring called geometry shader. The aim of a geometry shader is to allow for more control than simply modifying the position of vertices coming from the input data. In this new shader it is therefore also possible to add and remove vertices.

2.3.2 Vertex Shader

The vertex shader handles the geometry information in the scene by doing an operation on each vertex that is fed to the graphics processor. A vertex is a

description of a position in 3D-space, but can also have other properties connected, e.g., color and texture coordinates. Vertices are then used to define polygons that describe objects in a 3D-scene.

The shader cannot create any new vertices but it can modify properties such as position, normal, colors and texture coordinates. The intention is to transform 3D positions into screen space, however there is obviously room for additional actions as well. In general the vertex shader also calculates a depth value for each vertex in a scene, which can be used for other operations later on in the pipeline.

2.3.3 Fragment Shader

The fragment shader is also known as pixel shader in DirectX. Input comes from the rasterization step where fragments are made to fill the surfaces created by the vertices with color. Here the fragment shader can perform per-pixel operations, typically to simulate lighting effects, and change the color of each fragment. In order to make the rendering look as realistic as possible, the fragment shader has access to light sources, normal vectors, textures and possibly other resources passed in by the developer.

2.4 General Purpose Processing

Since the GPU has become so powerful during the recent years, the latest trend is to use it for more than plain graphics applications. Also the programmable shaders and the efficient parallel architecture are making the GPU very attractive as a speed up for many other computing problems.

In the latest shader models, introduced by Microsoft with DirectX 10 (DX10), the specification unifies the different shaders into a new single computation pool of programmable resources. This pool can be used as the developer pleases for vertex shaders, fragment shaders and the new geometry shader. In addition, the unified shader hardware balances better depending on what shader requires extra computing resources.

The parallel computing fashion that the GPU comes with, operates with vectors. This means that a vector operation, such as dot product or cross product, can be performed with a single operation on the GPU, however it would take the same number of operations as the number of components in the vector, on the CPU side. In computer graphics this has been obvious for a longer time, but is clearly useful for other applications as well. General purpuse computing is trying to take advantage of this feature and is often referred to as GPGPU.

One former problem has been that it was hard to make use of the GPU with more general data, mainly because both the hardware and the programmable shaders were intended for graphics. However in the last years better tools have been developed. One of these tools is NVIDIA's CUDA, a Software Development Kit (SDK) and an Application Programming Interface (API) that allows the programmer to an extension to C, instead of one of the ordinary shader languages. Usually this is much easier because it is not restricted in the same way as shader

programs are when it comes to memory access and control over the execution process. ATI/AMD offers similar technology called CAL. Tools like these makes it easier to create programs that can utilize the power of the GPU for more general purposes.

2.5 Previous Work

Work has been done regarding DOF for quite some time, which means that different solutions have come forth in the process. Methods have been developed for both real-time and offline applications, supplying a flora of ideas and techniques that can be implemented. It is important to understand that when real-time performance is of the essence, more simplified and computationally light solutions are necessary and they cannot compete with the more computationally heavy solutions when it comes to realism.

Almost all methods have in common that they achieve variable blurs according to a simulated lens, further discussed in section 3.2.2, and tries to mimic the human perception. In [18] and further improved in [20], Tatarchuk et.al. makes use of a thin lens model, Poisson disc distribution and downsampled images to achieve DOF effects for game cinematics. The results are good but only small blurs are supported and artifacts arise around objects where fast changes of sharpness occur. A similar approach is further discussed in [5] with the same rendering problems for a real-time in-game solution.

When real-time performance is not the main goal other methods has been tested. For interactive frame rates, [1] and [11] use heat diffusion to create larger blurs by solving differential equations on graphics hardware. The first one stated runs in several passes to blur the image more and more for each pass, and the latter solves a tridiagonal matrix, further discussed in section 3.5.2, to achieve blurs with no blur size limit whatsoever. Another method, based on bilinear interpolation, has been discussed in [14] that is efficiently coupled to new graphics hardware and supports variable blur sizes when run in multiple passes.

Slower methods have been made as well, for areas where rendering time is not utterly important. In [13], the authors propose an image-based algorithm where sets of output images are calculated depending on depth values. The images are then recomposed to achieve the final result that is natural and fairly realistic. To get perfectly realistic simulations, ray traced solutions has to be made. Unfortunately these are too slow to be run for anything except still images, but can effectively be used for reference rendering or when the rendering time is not a problem. In addition, ray traced methods cannot be applied when the overall pipeline does not support it. In most cases information about the geometry is not available in the post-process step, and that is where the DOF effects are most effectively implemented.

There has been discussions about whether or not DOF effects in computer games affect the player when it comes to in-game performance. In [10] a study was made regarding this topic and after testing it was found that players do not perform worse when DOF is turned on, and was at the same time appreciated by

the testers.

Chapter 3

Depth of Field

This chapter presents the fundamentals of DOF and the physics behind the effect. The first part talks about the human perception and why this effect occurs. The second section discusses different camera models and how lens parameters can be used to affect the outcome of an image. Furthermore, different blurring approaches are described followed by a section focusing on depth calculation. Lastly we describe a more advanced method for creating and calculating larger and better blurs for DOF. The last section is additionally the most theoretically heavy but also necessary for the upcoming implementation chapter.

3.1 Human Perception

To understand the human visual judgement and stimuli, perception is always of importance when dealing with image generation [4]. Changes in illumination and focus can highly affect the perception of an image. Focus and blurriness are used in the human brain as depth cues, and are therefore of great importance when it comes to three dimensional perception.

3.1.1 Focus in the Human Eye

Since the human eye can only focus on a small part of the visual area at a time, it has to change its properties depending on the focus distance. The lens changes its refractive power to adjust the focal length (distance from lens to retina), resulting in sharp and blurry areas. The DOF is the area that appears to be sharp in front of and beyond the focus point as seen in figure 3.1.

3.2 Camera Models

When capturing a scene, may it be in reality (photography) or from a computer generated world (rendering), different images are being generated depending on the camera setup. Lenses and camera parameters may bend, distort, or blur the

Figure 3.1. Depth of field photo example. Focus on the backside of the frame with short DOF gives blurry foreground and background.

output. How we choose the camera model will therefore be of great significance for the resulting image. The two most common models are described in this section, but keep in mind that there are also other models available such as fisheye lenses (wide-angle lenses).

3.2.1 Pinhole Camera Model

A pinhole camera model can be used to project a 3D-scene onto a screen. In most real-time computer graphics applications this is the standard way of rendering. Light scattered from the environment is sent though an infinitely small hole before hitting the image-plane, as shown in figure 3.2. This results in a perfectly sharp and focused image [20]. Only one ray is allowed through the hole for each pixel in the image, hence no opportunity for distortion or blurring is given. However, this method is fast and easily implemented and has therefore been the most used one in the past.

3.2.2 Thin Lens Camera Model

Real-world cameras have a thin lens that bundles of light can pass through. From each point in the scene a cluster of light rays is emitted through the lens. They are being refracted by the lens and refocused at a point close to, or on the image plane. If the bundle converges exactly on the image plane the pixel will be in full focus. If this is not the case the bundle will go through the image-plane without converging on it, giving way to an intersecting, circular area called the Circle Of Confusion (COC) [5]. How this camera model works is illustrated in figure 3.3.

The parameters used in a thin-lens model are the following:

- f - focal length
- a (f-stop) - aperture number
- d_{focus} - focal distance

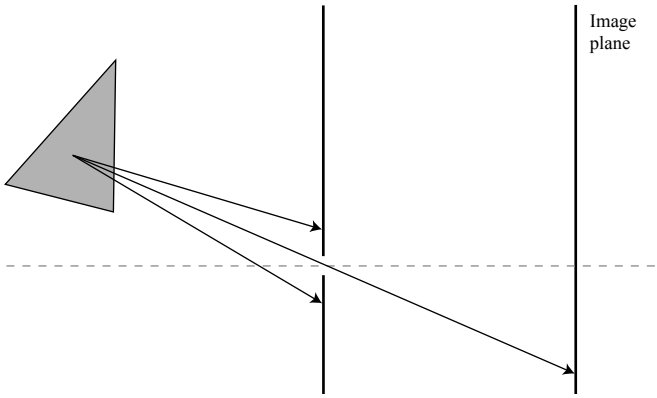

Figure 3.2. Pinhole camera model, where a single ray from a fragment hits the image plane and results in a perfecly sharp image.

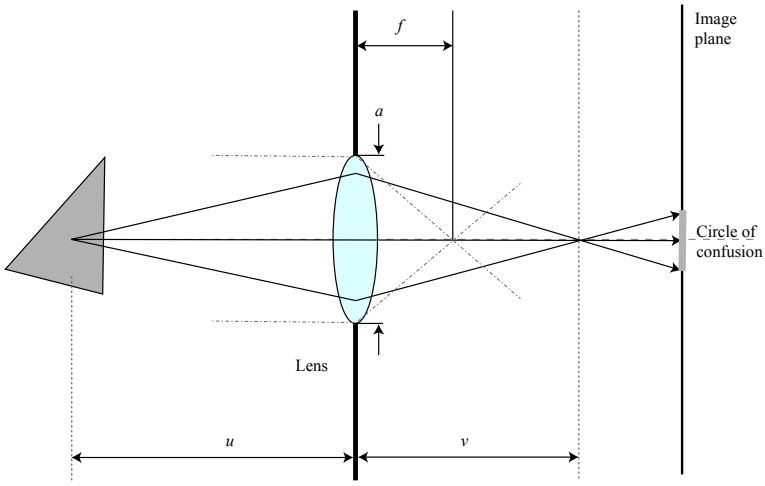

Figure 3.3. Using the thin lens model a boundle of rays passes through the lens and creates a COC.

- u - distance between a point and the camera lens
- v - distance behind the lens where image is in focus

The thin-lens equation states the relationship between f, u and v as:

$$\frac{1}{u} + \frac{1}{v} = \frac{1}{f} \tag{3.1}$$

The COC diameter for a point at distance d from the camera can be computed as:

$$coc = \left| \frac{f\, d}{d - f} - \frac{f\, d_{focus}}{d_{focus} - f} \right| \left(\frac{d - f}{a\, d} \right) \tag{3.2}$$

3.3 Blurring Methods

All spatial-smoothing (blurring) techniques are using the same basic fundamentals. For every pixel in the image, a new color is calculated based on the pixel's surroundings. How the samples are chosen depends on the method and could be either symmetric or non-symmetric to give the desired result. In this section we will discuss bilinear interpolation and Gaussian blur since they can be seen as the building blocks for more advanced methods.

3.3.1 Bilinear Interpolation

When talking in mathematical terms, bilinear interpolation is the extended version of linear interpolation, which is used to find the value of an unknown function for a particular point on a line. If the four corner-values on a rectangular grid are known, the value of point P inside the grid can be estimated by first doing linear interpolation on the horizontal lines, followed by an interpolation on the resulting vertical line, as illustrated in figure 3.4:

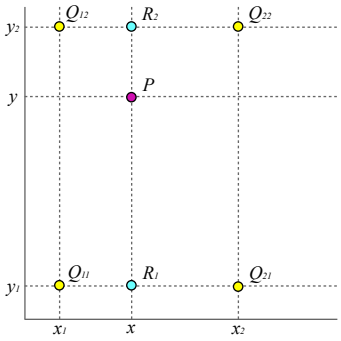

Figure 3.4. Bilinear interpolation illustration where the value at P is calculated from the corner values at positions Q_{11}, Q_{12}, Q_{21}, and Q_{22}.

Using the notation from figure 3.4 we can define the value at point P as

$$f(P) \approx \frac{y_2 - y}{y_2 - y_1} f(R_1) + \frac{y - y_1}{y_2 - y_1} f(R_2), \tag{3.3}$$

where $f(R_1)$ and $f(R_2)$ are the horizontal interpolations:

$$f(R_1) \approx \frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21}), \tag{3.4}$$

where $R_1 = (x, y_1)$ and

$$f(R_2) \approx \frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22}), \tag{3.5}$$

where $R_2 = (x, y_2)$.

Using the basic functionality of the bilinear interpolation on graphics hardware, fast blurring and effective down sampling becomes available [15]. For example a small blur can be achieved by, for every pixel, do a bilinear sampling at the lower-right corner of the pixel, resulting in an average value for those four adjacent pixels [13].

3.3.2 Gaussian Blur

The image-blurring filter known as Gaussian blur uses a normal distribution for deciding the transformation that should be applied to an image's pixels. The distribution function in two dimensions (2D) can be stated as:

$$G(u, v) = \frac{1}{2\pi\sigma^2} e^{-(u^2+v^2)/(2\sigma^2)} \tag{3.6}$$

Where σ^2 is the standard deviation and r ($r^2 = u^2 + v^2$) is the blur radius of the distribution. Values from this distribution can be used to construct kernels (convolution matrices), that can be applied to an image in 2D. Every pixel gets a new value based on the values of its neighborhood and the Gaussian kernel. Each tap in the kernel contains a weight saying how much the pixel in that spot should affect the outcome of the new computed pixel value. An example of a 3x3 approximated Gaussian discrete kernel can be constructed as in figure 3.5 [18].

1	2	1
2	4	2
1	2	1

Figure 3.5. Appoximated Gaussian weights for a kernel of size 3x3, the center pixel is aligned with the pixel being calculated.

To be more efficient when convolving, symmetrical filter kernels can be separated into two identical smaller kernels. Filtering an image with these two kernels (one of them transposed) will result in the same output as if the image was being filtered with the former one. The separable kernels extracted from figure 3.5 can be viewed in figure 3.6:

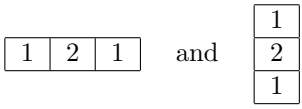

Figure 3.6. These two separate kernels can be used in two passes instead of the single kernel in figure 3.5 to allow for faster filtering.

3.3.3 Poisson Disc Filter

Another way of blurring an image is to apply a Poisson disc filter for each pixel in the image. Stochastic methods according to a Poisson disc distribution is used for determination of the tap positions on the filter kernel [18].

The center tap is aligned with the pixel being filtered, and the outer taps are being sampled from the pixels in the neighborhood. A Poisson disc with 12 taps for image blurring could look like figure 3.7.

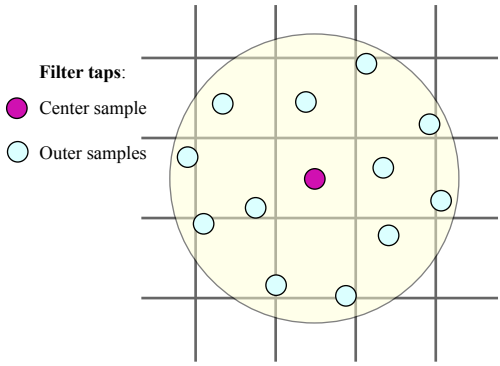

Figure 3.7. Example of a Poisson disc distribution with 12 outer taps where the position of the taps are used to sample pixel values from an image.

3.3.4 Summed-Area Tables

Variable width filters in constant time per pixel can be achieved using a Summed-Area Table (SAT) as in [6, 7, 9]. Every entry in the table stores the sum of all values between the entry point and the lower left corner in the image, resulting in small values in the lower left, and high values in the upper right (depending on input). When computing the table for a 2D-image, one can efficiently do a sum scan first for all the rows followed by all the columns of the image.

When creating a box filter using a SAT only four texture lookups are needed per pixel, one for every corner in the blurring area surrounding the computed pixel. How far away the samples are made depends on the size wanted for the filter region. The result of the filter can be written as:

$$s_{filter} = \frac{s_{ur} - s_{ul} - s_{lr} + s_{ll}}{w \times h} \tag{3.7}$$

where s_{ur} is the upper right fetch, s_{ul} is the upper left, and so on. w and h are the width and height of the filter kernel.

3.4 Depth Calculation

A pixel on the image-plane has a color sampled from the 3D-scene. In a similar fashion every pixel on the image-plane could have a depth value depending on the 3D-representation. This value can be calculated as the distance from the pixel in the image-plane to its corresponding point in 3D-space.

The DOF filter is totally dependent on these values, hence they are needed for the fundamental computations. There are different methods available for resolving these values and the approach depends on the rendering API, platform hardware and programmer's choice.

3.4.1 Extracting Depth Values

The depth value could be calculated in the vertex shader and stored in the alpha channel in the framebuffer, or saved to an external texture for later use. This is not optimal though, since the alpha channel might be used to store material opacity values and using an extra texture requires time and memory that might be indispensable. Therefore, other methods have been developed, allowing the programmer to efficiently make a copy of the depth buffer and use it as a texture at a later stage. This approach and similar has been the best way to go before the release of DX10. On state-of-the-art hardware you can now directly access the depth buffer as a shader resource (texture) and use it in your shaders with no extra expense [3].

3.4.2 Precision of the Depth Buffer

The standard hardware depth buffer, or Z-buffer as it is also called, is calculated with the assumption that it is going to be used to decide visibility. This means that for every pixel that is to be drawn, if there are objects that must be rendered to the same pixel, a comparison is made in the hardware. The object with the depth closest to the observer is drawn to the screen. Usually objects closer to the screen are more important and cover a larger area of the screen than distant ones. Therefore, the depth buffer has better precision near the observer and is not stored linearly throughout the depth. Another reason is that optimization makes it possible to interpolate the depth calculation for each pixel by using the vertex depth. This takes fewer computations and, even though this is essentially done anyways for texture coordinates, it is originally a speed up that has stayed. Additionally, linear depth is not something that is useful for all applications. Unfortunately, for DOF, linear depth is of utter importance. For these types of situations there is an alternative to the standard Z-buffer called W-buffer that has linear depth. Sadly, this one has not too good support in hardware at the current time.

3.5 Simulated Diffusion Method

From a performance perspective, acquiring large variable width blur kernels are a major complication. Gaussian blur kernels can be made efficient but becomes slow when the kernel size is too large. Larger kernels will also cause troubles for complex boundaries of in-focus and out-of-focus objects. Simulated diffusion is a way to deal with this problem by applying knowledge from the field of thermal heat [1]. This method will respect the boundaries of objects that are in different focus areas as well as dealing with spatial varying blurs [11]. How this is done is further explained in next section 3.5.1.

3.5.1 Heat Diffusion

Simulated heat diffusion comes from the idea that the COC circles are treated as heat transfusion variations. This allows every pixel to diffuse onto their neighborhood as if they where temperature samples on a surface. Using this formulation, heat conductivity will vary depending on the circle of confusion. In-focus objects will therefore carry conductivity of zero, keeping the object in-focus and at the same time maintain its boundaries [11].

If we consider an image $x(u, v)$ that we want to diffuse using the heat equation, our result image $y(u, v)$ can be derived from equation 3.8.

$$\gamma(u, v)\frac{\partial y}{\partial t} = \nabla \cdot (\beta(u, v)\nabla y) \tag{3.8}$$

Here γ is the specific heat for the medium and β is the heat conductivity. By using the input image as the initial heat distribution we can integrate the heat equation with respect to the time and get a blurred result image.

We need to use a numerical method to solve this equation, and there are a number to choose from. The most intuitive method is known as forward Euler's method but will not be sufficient enough for our performance demands. In [12], the authors propose another method called Alternating Directions Implicit (ADI) where they are using this to simulate shallow water, independent of wave speed. This method is furthermore used in [11] where the idea is to split the heat equation into two substeps where each axis is made separately, much in the same way as a 2D Gaussian kernel can be separated into two one dimensional Gaussian kernels as explained earlier in section 3.3.2.

Using this method, each substep must solve equation 3.9.

$$\gamma\frac{\partial y}{\partial t} = \frac{\partial}{\partial u}\beta(u)\frac{\partial y}{\partial u} \tag{3.9}$$

To make this as simple as possible and since we are dealing with pixels with unit distance we can, by selecting a time step equal to 1, discretize this equation over space as in equation 3.10:

$$\gamma_i(y_i - x_i) = \beta_i(y_{i+1} - y_i) - \beta_{i-1}(y_i - y_{i-1}), \tag{3.10}$$

where $\beta_0 = \beta_n = 0$ (n is the last value of i), to force the boundary of the image to be surrounded by heat insulators and will therefore be treated as in-focus.

Equation 3.10 can usefully be written in a tridiagonal matrix form, how this benefits us is further explained in section 3.5.2.

3.5.2 Tridiagonal Systems

A tridiagonal system can be written in the form $Ty = x$, where x is the input and y is the output and T is the quadratic tridiagonal matrix with nonzero entries only in the diagonal positions and their closest neighbours as:

$$\begin{pmatrix} b_1 & c_1 & & & 0 \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & c_3 & \\ & & \ddots & \ddots & \ddots \\ 0 & & & a_n & b_n \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} \tag{3.11}$$

Tridiagonal systems can for example be of use when simulating water [12] or, as in our case, when calculating DOF [11]. Tridiagonal systems can be solved very efficiently in constant time per sample as opposed to other systems with matrices on other forms. For solving the tridiagonal system one can make use of the traditional lower/upper triangular decomposition and forward/backward substitution method [8], which will guarantee a correct solution in constant time regardless of filter kernel size. In our situation however, we are interested in a method that makes good use of the parallelism of the GPU, and another approach known as cyclic reduction is more feasible for our needs when it comes to efficiency and speed.

3.5.3 Cyclic Reduction

The recursive method known as cyclic reduction, reduces the size of the matrix T in phases (cycles) until only one row remains (T must be of size $N = 2^n - 1$). This last row is then propagated back into the partial matrices until the whole system is solved. The computations needed for solving the system in this way can be divided into smaller parts and distributed to multiple computing threads [7], hence giving way to good parallelism that is well suited for the GPU.

Since this is a key method for our work it is of importance to discuss it further, so consider a tridiagonal matrix of phase j as M^j and the vector as y^j. The values in each row of M^j is stated a_i^j, b_i^j and c_i^j similar to equation 3.11. The operations that are needed to work out the values in row i in the matrix are [19]:

$$\alpha_i^j = -\frac{a_i^{j-1}}{b_{i-2^{j-1}}^{j-1}}, \quad \gamma_i^j = -\frac{c_i^{j-1}}{b_{i+2^{j-1}}^{j-1}}, \tag{3.12}$$

$$a_i^j = \alpha_i^j a_{i-2^{j-1}}^{j-1}, \quad c_i^j = \gamma_i^j c_{i+2^{j-1}}^{j-1}, \tag{3.13}$$

$$b_i^j = b_i^{j-1} + \alpha_i^j c_{i-2^{j-1}}^{j-1} + \gamma_i^j a_{i+2^{j-1}}^{j-1}, \tag{3.14}$$

$$y_i^j = y_i^{j-1} + \alpha_i^j y_{i-2^{j-1}}^{j-1} + \gamma_i^j y_{i+2^{j-1}}^{j-1} \tag{3.15}$$

It is clearly seen in equation 3.15 that the system is dependent on the previous phase. The row dependency for the different phases can be viewed in figure 3.8 where the top of the tree is the last and final level and is obtained by computing equation 3.15.

The back substitution is started when the final level is reached and the values of y_i can be determined recursively. The first operation is

$$y_0^{n-1} = \frac{y_0^{n-1}}{b_0^{n-1}} \tag{3.16}$$

and the other values of y_i can be computed as:

$$y_i = \frac{y_i^{j-1} - a_i^{j-1} y_{i-2^{j-1}} - c_i^{j-1} y_{i+2^{j-1}}}{b_i^{j-1}} \tag{3.17}$$

where j is the last phase in the reduction step.

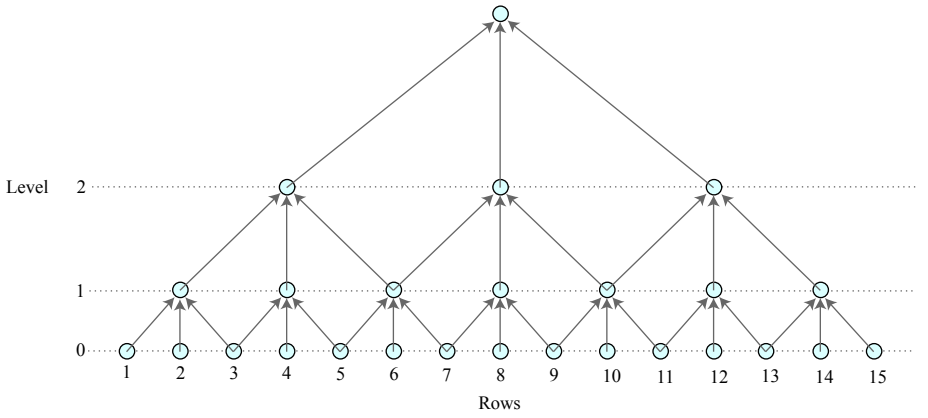

Figure 3.8. Row dependency for odd-even cyclic reduction for N = 15. The tree illustrates the connectivity of three downsampling passes in the cyclic reduction algorithm. Data from level 0 is used in the calculation of level 1 and so on until a level with only one value remains.

Chapter 4

CUDA

We chose CUDA as our GPGPU programming language mainly because of its ease of use but also because of its fairly large community which offered great support.

This chapter explains the essential knowledge needed to use NVIDIA's programming API CUDA. We start off by explaining what this type of programming can be used for and what is needed from the programmer to utilize it. Next we introduce two more technical sections which describe available computational resources and how to best take advantage of these, as discussed in [17]. This chapter is important for the second half of the following implementation chapter.

4.1 General

CUDA is a C-like language provided by NVIDIA for programming more general computations on the GPU. CUDA is supported on NVIDIA graphics cards from the G80-series and later. The advantage of this compared to using a standard graphics API is that CUDA do not limit the programmer to a fixed pipeline nor demands learning a specific language but is purely an extension to C. In practice, this gives a pretty flat learning curve for most programmers. In addition, it does not require that general data is mapped to fit a graphics application as before.

As mentioned in section 2.4 the GPU has traditionally been reserved for graphics computations and the GPU was developed to fit the needs for these types of applications; usually highly parallel computations with large data sets, such as arrays with pixels and vertices. The hardware is therefore designed in a way that fewer transistors are used for data caching and flow control and more for processing. A comparison scheme can be seen in figure 4.1.

This makes the GPU very well suited for problems that can be executed for many elements at the same time and does not have too much data dependency. Naturally this can be taken advantage of in applications outside computer graphics.

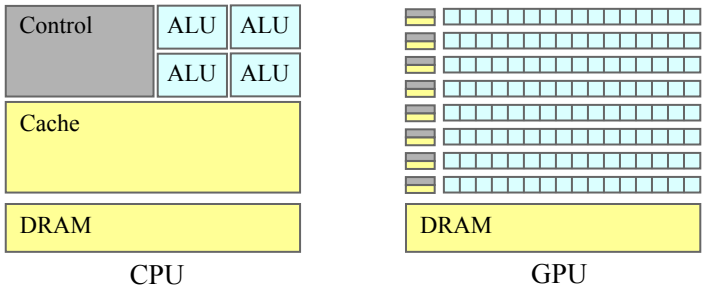

Figure 4.1. Comparison scheme between CPU and GPU transistor setup. The GPU consists of more transistors, giving way to fast parallel computation but less caching and flow control.

4.2 Memory Usage

CUDA also offers better memory addressing, with both gather and scatter capability, which is shown in figure 4.2 and basically translates to both read and write access at any location. The lack of this has been one of the major difficulties with standard shaders and a main reason that "render-to-texture" techniques implemented in section 5.3.1 are being used.

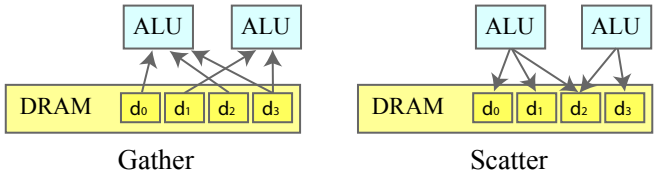

Figure 4.2. The left side shows a memory gathering instruction where values from various DRAM positions are read by a computing unit. The right side shows scattering of values into memory.

4.2.1 Device Memory

The first memory type on the GPU is Device Random Access Memory, which corresponds to the device memory. This is the counterpart to Random Access Memory on the CPU side. Device memory holds the largest capacity and can be both read and written to by the host CPU. When using CUDA, we divide device memory into three different memory spaces: global memory, texture memory, and constant memory. The reason is that they are optimized for different usages.

Global memory is a non-cached and straight forward memory, but also the most time-costly memory type. Access to global memory should therefore be minimized as much as possible. Texture memory is read-only but does also offer a 2D spatial local cache, so threads that read texture coordinates that are close together achieve best performance.

Constant memory is also read-only and contains a caching function but is not spatial dependent but more so fully used when all threads read from the same address.

4.2.2 Shared Memory

CUDA features an on-chip shared memory with fast read and write access. As shared memory indicates it can be used to share data between threads, but is limited to do so within a block; blocks will be further discussed in section 4.3. When utilized accurately, the shared memory can minimize fetching from global memory and thus making the implementation depend less on memory bandwidth. Put simple, shared memory lies closer to the computational units and therefore gives faster access with minimal latency. A scheme of this can be seen in figure 4.3.

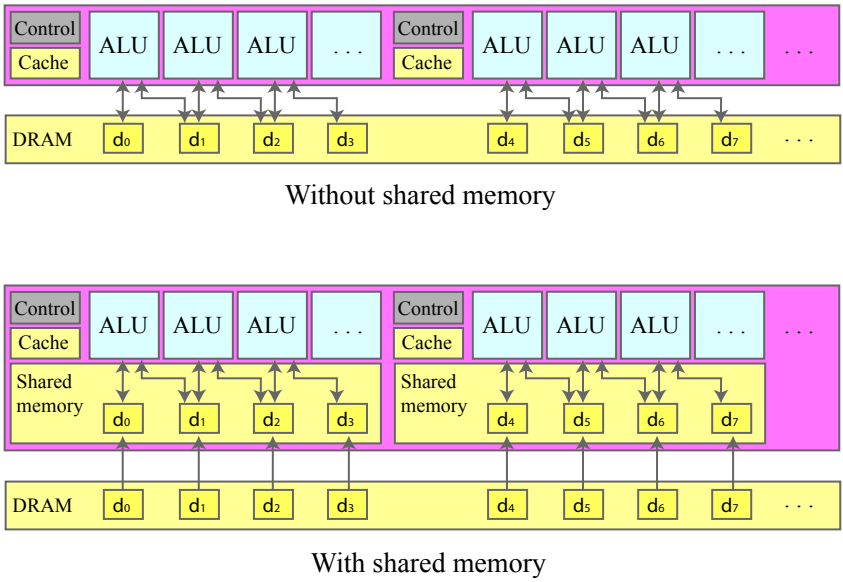

Figure 4.3. Shared memory reduces memory latency by shortening the distance between computing unit and memory. The upper figure shows a setup without shared memory while the lower illustrates how shared memory can allow for faster access.

Clearly, shared memory brings the preeminent performance gain when numerous memory reads from device memory occurs. The programming pattern is therefore typically to stage device memory data into shared memory, process the data when in shared state, synchronize threads, and finally write back to device memory.

Shared memory can be as fast as register accessing provided that there are no bank conflicts between threads. Banks are memory modules that can be accessed concurrently, so reads and writes in memory can be dealt with at the same time as long as they fall in unique memory banks. When two memory addresses fall within the same bank, a bank conflict arises [6]. To resolve this a serialization is

done and the hardware splits the memory bank into as many separate parts as necessary until the bank conflict is resolved.

In order to reach as good performance as possible with CUDA it is especially important to understand how memory banks and memory addressing works so bank conflicts can be minimized and memory allocation made efficient.

Since shared memory can only be accessed by threads within a block, next section, 4.3, will discuss the meaning and function of these two.

4.3 Programming Model

When using CUDA, the GPU can be seen as an extra compute device that can off-load the CPU for portions of the application. The most advantageous types of sections are those that need to be executed repeatedly but are independent of different data. These sections are compiled into so-called kernels, which are downloaded to the device.

4.3.1 Threads

The GPU works best with a very high amount of threads that runs at the same time. Threads are run in different batches called thread blocks. Within these blocks they share data through the shared memory described in section 4.2.2.

4.3.2 Grids

Naturally there is a limit of how many threads that a block can contain. To increase the total number of threads even more blocks are grouped into grids of blocks. Several blocks can be run simultaneously but cannot communicate between, nor be synchronized between different blocks. The execution order of different blocks in a grid is unfortunately undefined, so there is no safe way for threads to communicate with each other even if they belong to the same grid.

4.3.3 Execution

Depending on the hardware different amounts of blocks can be processed at the same time. It is also dependent on how much shared memory each block requires since the amount of shared memory is split among the threads in each block. Each processed block is then split into so-called warps.

The notion of warps is very important as shared memory contains banks, mentioned in section 4.2.2, which have the same size as a half-warp. This means that a shared memory request for a warp is split into two parts, one for each half. Hence, a thread that belongs to the first half of a warp cannot generate any bank conflicts with a thread belonging to the other half of the same warp. This is valuable when trying to optimize CUDA kernels.

Chapter 5

Implementation

This chapter explains practical details of the implementations that were made. The first part briefly describes the application environment, followed by the camera and the DirectX section were we discuss the algorithms implemented using HLSL. The last section goes through the CUDA implementation and how the algorithms were transferred to fit the multi-threaded domain.

5.1 Application Environment

For the implementations Microsoft Visual Studio 2005 was used as development environment. As programming language we use C++ with DX10 extended with HLSL as the high level shading language. All these choices are industry standard and are a natural part of the development at DICE. Especially DX10, which is available with Microsoft Windows Vista, was chosen for its improved depth buffer capabilities. It is also the expected next generation gaming platform.

5.2 Camera

We simulate a thin lens to be able to create variable blurs in our implementations. Basically our lens model takes the depth map as input and converts it into a blur amount map, taking lens parameters, maximum COC and near/far plane into consideration.

The standard depth map in DirectX is non-linear to allow for more accurate near camera Z-buffering. In our case though, we need a linear depth map if the DOF effect is to make any sense. The solution to the linearity problem is to divide all vertex Z values, passed through the vertex shader, with the far plane distance, scaling the range of it between 0.0 and 1.0. This will result in a linear output that will not change during perspective division. To further optimize this computation we can instead directly do the scaling on the Z column in the projection matrix to obtain the linear depth map.

When we have a linear depth map we calculate the COC for every pixel using equation 3.1 from section 3.2.2. The parameters: aperture, focal length and focal distance, can be changed with sliders during runtime to change the DOF effect.

5.3 DirectX

DirectX is as OpenGL a proprietary hardware graphics acceleration platform for Microsoft Windows. At DICE this is the standard development environment and therefore a natural choice for our implementation.

As of version 10 of DirectX a number of new features have been presented to supported hardware. As mentioned in section 2.4 the new application pool offers new ways to allocate resources for where they are best needed and the geometry shader delivers totally new ways to handle vertices by allowing both creation and deletion.

Another improvement is the use and handling of shader resources. This has been greatly improved with added functionality for reading the depth buffer as a shader resource (a texture). This was not possible in DirectX 9 where an extra pass of texture creation and copy process was needed. For our purpose this is a clear upgrade since our application depends heavily on the depth buffer.

5.3.1 Standard DirectX 10 Pipeline

Since the standard rendering pipeline is somewhat fixed in its way of handling data and computations there is a need for arranging the pipeline in a stepwise manner. For our post-processing operations this refers to multi-pass rendering where we are passing textures back and forth for GPU procedures.

The technique when data is stored as a texture, processed on the GPU, and saved as a texture is called "render-to-texture". When done in multiple passes it is sometimes called "ping-ponging" because usually the data is transferred back and forth between textures. This, of course, gives an overhead of data transfer time between CPU and GPU. But the trade off for not using the GPUs parallelism is much worse, so in the end there is still a huge gain when using this method.

Figure 5.1 shows a simple scheme of how the data in form of textures are bound and sent to the GPU for processing, saved in texture again, and sent back for further processing.

The methods and implementations described in following sections 5.3.2 - 5.3.5 are using this scheme for passing textures as data for GPU processing.

5.3.2 Gaussian and Pyramid Blurs

To get a good understanding of blurring methods on the GPU, we have implemented simple Gaussian and pyramid blurs that can be used as reference and for speed-tests. Separable Gaussian kernels can be created with size 3, 5, 7 or larger and run through the shader in two passes. The kernels are made according to the theory in section 3.3.2 and for each size we make use of an offset vector and a weight vector, giving way to an efficient data structure. When calculating the new

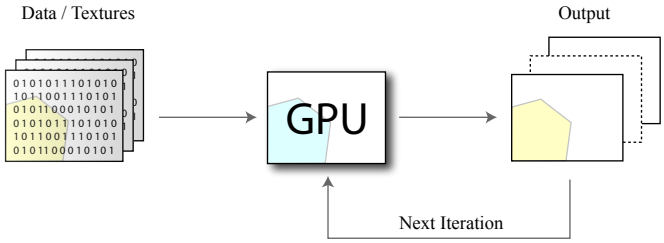

Figure 5.1. Schematic over the render-to-texture pipeline, where data from previous rendering is used in the upcoming iteration step.

color for a pixel on the screen, texture lookups are made in the input image according to the offset vector. The output values are weighted correctly (multiplied with the value in the weight vector) and fed back to the pixel. This is done first as a horizontal pass followed by the vertical one to achieve a 2D-symmetric filter solution.

This simple method can be of use when we construct the more advanced filters later on. We also found it valuable to try bigger blurs for speed-testing, and we therefore constructed a pyramid kernel that can take any user defined value. The difference between this one and the separable Gauss is the underlying function that sets the weight vector. The middle tap in the pyramid blur with odd size N has the weight of $(N+1)/2$ and the other weights get their value as ($middle\ tap\ weight\ -\ number\ of\ steps\ to\ middle\ tap$). This results in a separable kernel as the one seen in figure 5.2.

1	2	3	4	5	6	7	6	5	4	3	2	1

Figure 5.2. Pyramid kernel of size 15. The middle tap (7) is aligned to the position with the pixel currently being filtered.

To make use of the highly efficient texture lookups that are available on recent hardware, we have also created a filter system based on box filters using interpolation on the GPU as discussed in section 3.3.1. We have a function that does one bilinear texture lookup only, at the lower corner of a pixel, resulting in values from the four adjacent pixels that can then be weighted (a 2x2 box filter) as in equation 5.1.

$$h_{box}^{2\times 2} = \frac{1}{4}\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \tag{5.1}$$

When an image is passed through this filter a blurred version will be the result. As discussed in [14], the method can be used to create bigger blurs by passing the image through several times, each time giving way to more blur, the same way as larger Gaussian blur kernels gives more blur. The drawback with this method is that the data will have to be transported back and forth in the GPU, once for every small 2x2 blur resulting in a fairly slow solution, when many passes are needed.

5.3.3 Poisson Disc Solution

Our Poisson disc solution is based on the theory in section 3.3.3 and supports small but variable blurs. Solutions similar to the one we present here are used for games that are currently on the market [20], as it is known to be fast and "accurate enough". We start off by calculating the COC with a thin lens model as described in section 3.2.2. After that, we construct a Poisson disc of the wanted size depending on the COC. The distributions we use have 8 taps, which are scattered across the circle to get an even sample area. Each tap has an offset vector according to its distance from the middle tap so the texture lookups can be done easily. We obviously want to have blurrier areas where the COC is big, and we therefore create a smaller version of the input image, called the downsampled image. This image is 1/16 the size of the input image, hence does not contain as much details as the full scale version.

For each pixel, for every tap in the Poisson disc, we do a texture lookup in both the full scale and the down sampled image. We already have the COC value for the Poisson disc center, and can therefore linearly interpolate (blend between two values) between the two different resolutions and save a new color value depending on the COC value. Once this is done for all taps on the disc, a final value is accumulated and scaled to fit the output.

The results of the method gives the desired output, however there are drawbacks as well. When there is a fast change in COC between pixels, artifacts will arise creating halos around objects in focus. To compensate for these there are two choices worth considering. We can either scale the taps' contribution to the disc, depending on if they have COC values larger or smaller than the middle tap, or do another pass with the disc for every tap in the image. The latter gives the best result when tweaked properly, but is very slow and scales the computations needed with the number of taps. The first stated fix will help a little and does not scale the computation noticeable. There is no way to get rid of the halos totally though and these artifacts and the small blur radius are considered to be the disadvantages with the Poisson disc based DOF.

5.3.4 Multi-Passed Anisotropic Diffusion

The previous methods discussed in section 5.3.2 and 5.3.3 are good for simple purposes, but performs badly visually when it comes to more advanced implementations. Because of this we wanted to implement an approach based on heat diffusion explained in section 3.5.

Our implementation is based on [1] in which they also introduce anisotropy in the heat equation. This is done by adding a weighting function which helps to preserve edges. The weight function varies with COC and is then implicit dependent on the depth buffer. Equation 5.2 shows the heat equation for image I combined with the weighting function g.

$$\frac{\partial I}{\partial t}(x, y, t) = \nabla \cdot (g(x, y, t)\nabla I(x, y, t)) \tag{5.2}$$

Since g is not varying with time, but with the depth value, it can be concidered as a constant. This insight will simplify the equation to equation 5.3.

$$\frac{\partial I}{\partial t}(x,y,t) = g\nabla \cdot (\nabla I(x,y,t)) = g\Delta(I(x,y,t)) \tag{5.3}$$

Using this equation we will obtain the same results as a local convolution with a Gaussian kernel of same width as COC in this point. Notice that COC still varies depending on what point we are considering, so there is not only one Gaussian kernel. To solve this we use an explicit numerical scheme stated in equation 5.4.

$$I_{i,j}^{n+1} = I_{i,j}^{n} + 0.25\nabla^{-} \cdot (g_{i,j}\nabla^{+} I_{i,j}^{n}) \tag{5.4}$$

where ∇^{-} and ∇^{+} are the neighbouring pixels gradient values.

For our implementation we first render a pass for saving a texture with weighting values. After that we iterate the original image with equation 5.4. The iteration step will continue until sufficient blur is achieved (further details in appendix B). If the iteration step continues for N steps, it can be compared with Gaussian smoothing of width σ according to equation 5.5.

$$N = \frac{\sigma^2}{2\sqrt{2}} \tag{5.5}$$

The iteration step is also the methods largest drawback. For bigger blurs a extensive number of iterations are needed which obviously will affect performance. For example, using 20 iterations is still just above a Gauss kernel of size 7, which can be seen in figure 5.3. That is usually not enough for large DOF nor film-like camera parameters and is still too computational heavy for real-time applications.

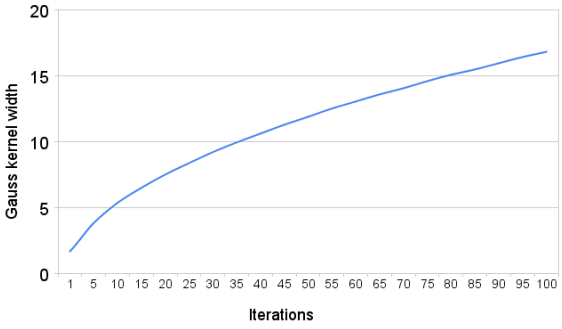

Figure 5.3. Relation between iterations and comparable Gaussian kernel width.

5.3.5 Separable Simulated Diffusion

As discussed in section 3.5 huge blurs can be achieved with advanced methods. We have implemented a solution using tridiagonal systems according to the theory in section 3.5.2.

To make use of the cyclic reduction algorithm we have to create additional resources to deal with the new data that the algorithm outputs. We therefore create new textures that can be used for temporary storage with the sizes necessary for the accurate down sampling. We will need a double setup of these textures, one for the output values in each phase, and one for the algorithms semantic values. For an image with width $(2^N - 1)$ the temporary textures needed gets their width in an iterative manner as $\lfloor (2^M - 1) \rfloor$ where M is the phase of the texture, resulting in the smallest texture being 1 in width, the next 3, then 7 and so on, until we reach the width of the input image. Since the algorithm is done in two passes, we first do the horizontal pass, and then a vertical pass with the difference that the texture sizes in the vertical pass are changed according to height instead of width. All the information that is to be stored could be packed into one larger texture, but we found it much easier and more controllable to use multiple resources.

The algorithm itself is run in three steps that are all done in two passes once the resources are created:

1. Calculate the heat conductivity β for the current pixel and the previous pixel by sampling from the depth map, and save these values as a data triple in the matrix texture at the same position as the current pixel. The data we store at this step is: $-\beta_{prev}, 1 + \beta_{prev} + \beta, -\beta$ and corresponds to the a, b and c values in the tridiagonal matrix.

2. Level down the image/matrix texture in phases and calculate partial solutions for the tridiagonal matrices stored for each row in the image until only one value remains. This is done according to the cyclic reduction method described in section 3.5.3.

3. Level up the image in phases and propagate the solution keys back into the partial solutions until the whole system is solved.

How the tridiagonal matrix values are saved to the matrix texture can be seen in figure 5.4. The pseudo code for step 2 and 3 looks as follows (for more details see appendix B):

```
- Level Down
1:  FOR L = 1...log2(N+1) - 1 {
2:      FOR all pixels {
3:          # Compute alpha and gamma
4:          # Compute a, b, c
5:          # Set the output value
6:      }
7:  }
8:  #Normalize the output of the last level with b

- Level Up
9:  FOR L = log2(N+1) - 2...0 {
10:     FOR all pixels {
11:         IF pixel position is [odd] THEN
12:             # Set output from last level
13:         ELSE
14:             # Set output based on a,b,c and old value
15:     }
16: }
```

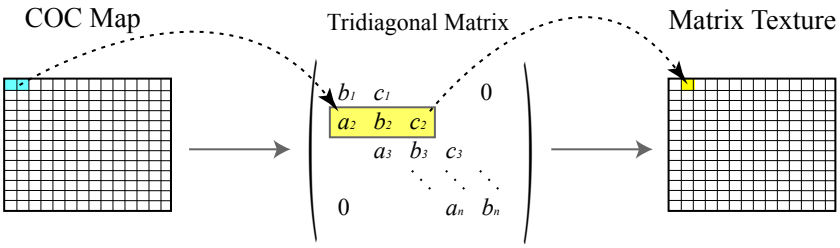

Figure 5.4. Tridiagonal matrix saved to texture. Values from the depth map are used to calculate parameters in the tridiagonal matrix and are stored as a texture (as RGB). One tridiagonal matrix corresponds to one row in the matrix texture.

5.4 CUDA

The multi-threaded, GPU based approach of CUDA can effectively optimize CPU-driven algorithms and is proven to be fast when it comes to memory reads and scatter/gather operations. We decided to implement our algorithms from section 5.3 using CUDA to get an understanding of the powers/weaknesses of the framework. Furthermore we implemented a SAT based DOF using the CUDA Data Parallel Primitives Library (CDPPL). Also of interest was to investigate whether or not CUDA allows for more advanced post-processing implementations.

5.4.1 Standard CUDA Pipeline

Compared to the DirectX 10 pipeline the way of computing data using CUDA is bit different. Even though there is a sense of textures there is no real fixed pipeline that we need to follow. Sometimes because of data dependencies and memory limitations we are however required to do multiple passes with our data. This should not be confused with "render-to-texture" but is rather a memory management method. When using texture memory, the routine looks similar. This is because of texture write limitations, which is not allowed yet, but could still be switched to device or shared memory.

Since version 2.0 of CUDA was still in beta and did not support direct access to GPU data we first download the needed data to CPU and then give our CUDA kernels access. The same goes for uploading the result to the GPU. This gives a pretty expensive overhead that hopefully disappears with the final version 2.0 and later, but during our implementations this was still a drawback. On the other hand, the performance drop is not included in our timings.

In the CUDA kernel itself blocks, threads and memory are setup and allocated with respect to rules and limitations mentioned in chapter 4. This is highly dependent on the application and will therefore be described individually for each implementation.

5.4.2 Gaussian Blurs in CUDA

As discussed in section 3.3.2, information about a pixel's neighborhood is needed to obtain a Gaussian blur. The big difference between our DirectX and CUDA implementation involves blocks, threads, and memory accesses. Since we wanted to try out different solutions we decided to implement three unique solutions, each building on a different memory structure. One uses device memory access, one uses texture reads and one uses shared memory write/reads. The two former methods are pretty straightforward and have much in common with our implementation from section 5.3.2.

For all solutions the first two steps are the same, we start off by copying the input data from the host computer to the device domain, making sure the correct amount of data is allocated and transferred. After that we make use of the constant device memory, where we store the needed filter kernel, which is fairly small memory wise and will not change during the lifetime of the CUDA kernel. When the setup is done we run the CUDA kernels, one for the horizontal blur, and one for the vertical. The block setup for the two first approaches is set to 16x16, and the number of blocks depends on the width and height of the image. Keep in mind that when using device memory or textures, we are not bound to do reads only inside the own block.

In the device memory only approach, we simply do data reads at the positions needed for each thread, and normalize and store the output value. For the texture read approach we first copy the device allocated data to a *cudaArray*, which can be bound to a texture (before kernel execution). Then data from the texture can be read from inside the kernel using *tex2D* reads, and the blur can be constructed as in the first approach.

When using the shared memory, the approach is a bit different. We start of with the horizontal pass where the blocks are constructed with a height of 1 and a width depending on the kernel radius. As discussed in section 4.3 threads are executed in chunks of half-warp size and should be aligned to fit the memory addresses so no shared memory bank conflicts occur. To meet these recommendations the width of a block is chosen as in figure 5.5 where *blockDimX* is set to 256 and *kernelRadiusAlign* is chosen so we can make sure the memory banks are aligned (preferable 16 for smaller filters).

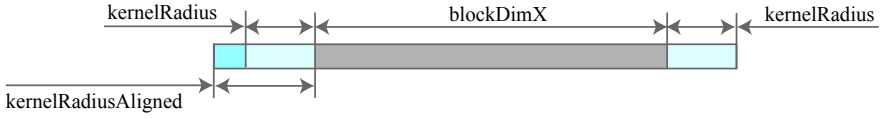

Figure 5.5. Thread block for row filtering pass. Block width is chosen as a sum of currently processed pixel area width, the kernel radius and a aligned kernel radius. Aligning the memory is not necessary but can increase performance.

We use one thread for each position in the block and apron and makes a copy from device memory to shared memory. This is done block-wise and not for every entry since the whole block may access the same shared memory throughout the

kernel lifespan. After the copy all threads within the block calculate the blur as in the other methods; the difference is that all data gathered comes from the shared memory this time.

The vertical pass has the same semantics as the horizontal one, but the blocks have to be changed to fit both the vertical filter layout and the memory banks. The blocks we use looks as in figure 5.6 where the *blockDimY* is set to 16. Preferably we would like to have blocks with bigger height to maximize the block size, but since we use floating point variables with four channels the shared memory size would not be enough. *blockDimX* is set to 16 to fit the wanted half-warp execution.

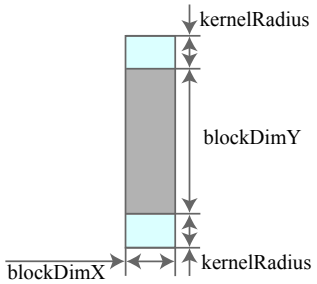

Figure 5.6. Thread block for column filtering pass. Block width dimension is kept as 16 to fit the half-warp execution while block height is the sum of the pixel area height with kernel radius as an apron buffer on both sides.

5.4.3 Multi-Passed Anisotropic Diffusion in CUDA

Our CUDA based solution using diffusion in multiple passes is very similar to the implementation in section 5.3.4 when it comes to the mathematic algorithm. Hence, parts of it will not be repeated again. There are still new things to keep in mind when using CUDA though, and to get information about the speed of the different memories we did four implementation using global memory, texture memory, shared memory and global/shared memory. For the global memory solution we simply copy the data to the device, run the kernel and go through the computations and save the output, which is passed into the kernel once again depending on the number of iterations.

The texture based approach is fairly similar but in the kernel data is read from a texture while the results are still stored in device memory. The cudaArray, which is bound to the input texture, needs to be filled after every CUDA kernel execution to make sure the next pass gets up-to-date data. The block sizes used here is 16x16.

For the shared memory implementation we use a block that is of size 32x8 with an extra apron of 1 at each border, since the algorithm needs to access data one unit around the computed pixel. We start the kernel by allocating shared memory and every thread fills it with one value from the global memory at the thread's position. After that, all reads are done from shared memory when computing the

result, which is sent back into the kernel for further passes.

After creating the shared memory based solution, we decided to make a hybrid approach, which uses shared memory for the horizontal fetches and device memory for the vertical fetches. This gives way to blocks of size 256 x 1 plus an apron of 1 at each side, and lets us align the reads to the shared memory banks.

5.4.4 Seperable Simulated Diffusion in CUDA

As already discussed in section 5.3.5 the algorithm based on matrix reduction and simulated diffusion requires extra resources to hold the information needed to solve the system. Therefore we start with allocation, device memory for the input image data, temporary image data for between the passes, output image data, input and output depth data and finally memory for the ABC matrix for the horizontal and vertical pass. We use blocks of size 256 x 1. Since the huge amount of data needs to be transferred in and out of the CUDA kernels we decided to use cudaArrays and texture binds for the final implementation because the total amount of shared memory on the device is not sufficient. Furthermore the maximum number of registers available is not enough for a shared memory only based approach.

To handle the different input/output sizes that every pass is coupled with we make use of a standard C-struct to keep all information in good order and easily accessable. As described in section 3.5.1 the solver can be divided into two parts, one horizontal and one vertical, where the output image of the horizontal pass is the input image of the vertical pass. This is the case here and for both passes we first run the down-sampling, and then the up-sampling. Between the iterations in the passes we copy the device data from the output image and ABC matrix to the corresponding cudaArrays as discussed in section 5.4.3.

5.4.5 Summed-Area Table Blur in CUDA

Our SAT approach is divided into two parts, the first is to construct the SAT, and the second is to compute the filter value for each pixel in the image. For the first part of the solution we make use of NVIDIA's CDPPL, which includes an optimized scan algorithm, exactly the one we are looking for. Input to the algorithm is a 2D data structure and we therefore separate the color layers in our input image and run the scan once for every layer. Since we need to create the sum of both the rows and columns, we run it twice per layer with a transpose in between and afterwards to get a horizontal and a vertical run.

Once the last transpose is made we reconstruct our image to its original composition and execute the blur CUDA kernel. We use a texture to sample the SAT and calculate the new output value with the algorithm described in section 3.3.4.

Chapter 6

Results

This chapter presents the results from our applications. Firstly, we discuss the test environment and the systems running the tests. The second part presents timing performance of the different implementations in HLSL and CUDA. The last section shows the visual quality of the different approaches and artifacts that has come forth.

6.1 Test Environment

The implementation part of this thesis was built upon DX10, using HLSL and CUDA for post-processing. For all solutions we start the post-processing with a rendered image and a depth map. The objectives of our tests are to evaluate performance in HLSL and CUDA for the different algorithms, and furthermore look at visual quality and arising artifacts.

The Gaussian blur filter is a pure evaluator of the different memory types available in CUDA and how they scale performance wise versus the textures of HLSL. The Multi-Passed Anisotropic Diffusion (MPAD) gives a good picture of how multiple kernel and shader runs scale in performance when the amount of iterations is being pushed high. The Separable Simulated Diffusion (SSD) is the most advanced algorithm and deals with memory capacity and usage.

The performance runs have been commenced on two separate machines with the following specifications:

- Windows Vista
- Intel(R) Xeon(R) CPU E5450 @ 3.00 Ghz (4 CPUs)
- GeForce 8800 GTS 512 / GeForce 8800 Ultra 768

Calculated time in all tests is displayed in milliseconds (ms) for both graphics devices.

6.2 Performance

Measuring performance of our different implementations gives an analysis of which algorithms are "fast enough" and have potential for future game development, and lets us setup a clear picture of how CUDA scales versus HLSL when it comes to post-processing.

The performance tests are being run with HLSL and CUDA and we capture the GPU time usage for the different methods. To achieve trustworthy and solid data we run every algorithm 100 times and calculate the mean value of those executions.

6.2.1 Gaussian Blur

Gaussian blur is a fairly simple calculation but for larger filters it gets very heavy on the pixel shader. As seen in table 6.1 the standard HLSL technique with two passes on the shader is clearly the fastest for the smaller kernel of size 7. This is not particularly surprising since the pixel shader is optimized for picking pixels in a small neighborhood. Already at this state there are major differences between the three different CUDA implementations and the importance of correct usage of memory shows.

As we progress to larger kernels the impact of what memory is used in the CUDA implementation become more obvious. Shared memory and better use of threads out-performs the two other more naive solutions. For the last kernel size this implementation even out-performs the otherwise faster shader for larger image sizes, while the two other CUDA applications fall far behind. This is most likely because the shader is getting limited by texture cache, which does not extend to the full kernel size. With CUDA we can control what needs to be cached and in this way use fast shared memory to our advantage.

		HLSL	CUDA		
Kernel Size	Image Size	Texture	Shared	Texture	Device
7×7	256^2	0.13	0.73	1.06	1.60
7×7	512^2	0.47	1.40	3.54	5.72
7×7	1024^2	1.86	4.79	13.66	22.00
15×15	256^2	0.24	0.75	1.82	3.14
15×15	512^2	0.87	1.54	6.52	11.80
15×15	1024^2	3.42	5.31	25.63	46.00
33×33	256^2	0.91	1.01	3.51	6.60
33×33	512^2	3.96	3.24	13.29	25.40
33×33	1024^2	17.20	12.07	52.45	99.80

Table 6.1. Timings for 2D Gaussian separable blur on GeForce 8800 GTS 512.

6.2.2 Poisson Disc

Our Poisson disc based DOF is the fastest method and as we can see in table 6.2, 3 ms for a 1K (1024^2) image is not much. This method was only implemented on the shader, since we are confident that the algorithm will scale pretty similar as the 7x7 Gaussian blur from table 6.1 when constructed in CUDA, because of the small sampling neighborhood. Furthermore this is the method currently used by most developers and is already fast enough for usage when run in HLSL.

Size	8800 Ultra	8800 GTS
256^2	0.21	0.21
512^2	0.77	0.78
1024^2	3.00	3.12

Table 6.2. Timings for Poisson disc based DOF using HLSL.

6.2.3 Summed-Area Table

Constructing and sampling from the SAT is slower then doing the Poisson disc approach, which can be seen when looking at table 6.3 and 6.2 respectively. When the input image is becoming bigger, the SAT algorithm does not scale as bad as the Poisson disc when it comes to performance, but on the other hand the time needed is fairly long already at the 256×256 image and the 1K image takes 11.90 ms on the 8800 Ultra which is high.

Size	8800 Ultra	8800 GTS
256^2	4.65	6.27
512^2	6.12	8.33
1024^2	11.90	17.50

Table 6.3. Timings for summed-area table based DOF using CUDA.

6.2.4 Multi-passed Anisotropic Diffusion

The approach taken by the MPAD method offers very realistic results with rather uncomplicated calculations and implementation. On the other hand, the character of it being a multi-passed method gives overheads that are hard to work around. In table 6.4 we can see the obvious impact that more passes have, especially for larger image sizes which forces the GPU to transfer quite large amounts of data in memory. The broader bandwidth of 8800 Ultra compared to 8800 GTS shows to be quite important when shoveling pixels [16].

The result is nevertheless one of the best of all the methods, and this without too much trouble on the way. Color bleeding is in principal not visible at all and

the borders are preserved clean. This good result also works throughout a very high amount of passes. More about the visual result in section 6.3.

For CUDA the overhead of texture transportations are a major part of the timings. It should be taken into consideration that the memory copying instructions used are not fully optimized in the beta version that we were running. Consequently, CUDA seems to suffer from far worse overhead than the shader implementation but is still something that we cannot assume to disappear fully even in a final version. However, with CUDA's shared memory it should be possible to work out a few passes within the same kernel and cut down on copy instructions.

With the support from the graph in figure 5.3 we can see that we need several passes to achieve satisfactory blur on the image. A closer look in table 6.4 shows that each pass for a 1K image takes around 0.8 ms and is far too expensive for real-time performance, with CUDA this extends to the double. Since texture look-ups stay close together for all passes, which benefits the shader, this is then again not too surprising. With faster CUDA copy operations the difference would probably be less significant.

		HLSL		CUDA	
Iterations	Image Size	8800 Ultra	8800 GTS	8800 Ultra	8800 GTS
5	256^2	0.28	0.41	1.49	1.86
5	512^2	0.97	1.61	2.75	3.53
5	1024^2	3.83	6.70	7.81	10.23
15	256^2	0.79	1.20	3.49	4.03
15	512^2	2.79	4.70	6.94	8.68
15	1024^2	11.17	19.51	20.88	27.52
25	256^2	1.30	1.98	5.52	6.18
25	512^2	4.66	7.77	11.02	13.83
25	1024^2	18.32	32.31	34.03	44.64
35	256^2	1.81	2.77	7.23	8.34
35	512^2	6.42	10.92	15.15	18.98
35	1024^2	24.82	45.11	47.17	61.81

Table 6.4. Timings for multi-passed anisotropic diffusion based DOF.

6.2.5 Separable Simulated Diffusion

The reduction needed for the matrix solving in this method presented somewhat difficult implementation problems for us, both with CUDA but even more so with shaders. Structuring and storing is a big part of the algorithm and it was not always straightforward to get it efficient.

However, it seems like we succeeded a lot better with the shader model than the CUDA model. CUDA gets big overheads due to memory copying which is not

present in the shader. Still, we do not believe that the CUDA timings in table 6.5 reflect the situation perfectly. There is probably much more to do when it comes to optimization and memory setup and usage in the CUDA implementation.

Nevertheless, there are some natural causes why CUDA does not perform that well here. First, the strength in CUDA is when there are reads that can be saved in shared memory and reused; this algorithm does not really contain too much where this can be practiced. Secondly, shared memory size, which at this point is only 16 kilobytes, grid size limits, and amount of registers limits us to very small resolutions. As a result, optimization is very much limited compared to the other algorithms if larger image sizes are to work.

One interesting thing that we saw in this algorithm is that the difference between the more high-end card, 8800 Ultra, and the one aimed more towards the middle-range consumer, 8800 GTS 512, is not that big at all. Only in the shader implementation at higher resolutions is the difference in computing power truly noticeable. This is most likely because the algorithm does a very small amount of calculations but still several iterations of memory copying. The later is more probable to take the same time for both cards, thus putting them pretty much side to side.

	HLSL		CUDA	
Image Size	8800 Ultra	8800 GTS	8800 Ultra	8800 GTS
256^2	1.71	1.86	11.23	10.06
512^2	2.90	3.62	17.86	15.45
1024^2	7.31	11.85	29.49	30.45

Table 6.5. Timings for separable simulated diffusion based DOF.

6.3 Visual Quality

There are different things to consider when examining the output images from the various implementations. Filter impact, in terms of allowed blurring size and the nature of the gather method used, play a big part in deciding the good and bad properties of a DOF filter. Different artifacts arise when filtering and these must also be in the calculation when choosing method. Some artifacts can be overlooked when traded for better performance, but in other cases the distortions might be unacceptable. A side by side comparison can be seen in figure 6.1.

6.4 Filter Impact

The maximal blur amount achievable is different for the various methods. SSD can achieve blurs of any size in constant time because of the nature of the algorithm. Endless blur can also be achieved using the MPAD, but only with an immense number or iterations, which is not considerable for real-time applications. The

SAT DOF does also support huge blurs in constant time but for the Poisson disc method there are limitations since the sampling area for each pixel is fairly small and the down sampled version of the image defines the maximum attainable blurriness.

Figure 6.1. Top left: Original image. Top right: Result with Poisson disc method. Bottom left: Result with MPAD method. Bottom right: Result using SSD approach. Larger images can be seen in Appendix A.

6.5 Artifacts

Since we are working in screen space with only color and depth information we are bound to get artifacts. The Poisson disc approach offers great performance but suffers from color leaking, seen in figure 6.2 around sharp edges. This mainly comes from the fact that we are sampling from a blurred image that by its nature have neighboring pixel values. The artifact gives an appearance similar to a halo or an aura around the object. The filter can be improved with additional samples in the Poisson disc but will unsurprisingly have a negative impact on performance and there will still be some color leaking left. Color leaking shows up in the MPAD method too, but is far less apparent.

Another artifact can be seen in figure 6.3, where an edge of an out of focus object gets sharp boundaries to an in-focus object. This is caused since the COC is very small as the object is in focus. The only way to solve this is to use different passes where foreground, midground and background are treated separatly. Knowing that we do not treat separate depth ranges differently this is an artifact that all our algorithms suffer from.

The SSD method suffers from a blocking artifact shown in figure 6.4. This happens since in-focus objects act as heat insulators in the heat equation and they will therefore result in an impenetrable threshold for nearby blurs to pass. The

Figure 6.2. Around sharp edges, colors from the blurred image leak outside the in-focus line.

Figure 6.3. A foreground area that is off-focus gets sharp edges to an in-focus background.

solution comes from separating image layers depending on depth range and matte out the in-focus object so that the background can be blurred correctly.

Figure 6.4. Blocking artifact where foreground and background meet.

The summed-area table approach suffers from huge amount of leakage artifacts, especially when larger blurring areas are wanted and these cannot be overlooked. Since the SAT method is both slower and does have far worse leakage than the Poisson method we consider it a bad approach for DOF.

Chapter 7

Discussion

This chapter will conclude this thesis starting off by going through the implemented methods and the used programming techniques. Furthermore it will suggest future work and propose further improvements to the developed applications, also discussing the eventual future break-through of GPGPU programming.

7.1 Conclusions

We were excited and hopeful to find plausible and fast methods for post-processing when starting working on this project. Now when it is done we are pleased with what we discovered but at the same time slightly disappointed that the more advanced presented algorithms are still not swift enough for the games industry.

DOF is one of the hardest post-processing effects to achieve, mostly because of its scattering process. Using GPGPU techniques to solve this is probably the right way to go since random lookups is not preferred in ordinary shaders. However, when using the latest technology there was also numerous implementation issues that arose. Driver problems and beta versions in general sometimes did not offer the full functionality or the full performance. Such things are most likely to be fixed in a close future, but are still troublesome to work with. Additionally, such versions rarely come with good documentation. On the other hand, using first-hand versions and being part of new software is also very rewarding when giving feedback to the developer.

Time consumption has been a major part of the project. The new GPU programming techniques shows big potential in both ease of use for the programmer and performance. Still, the time demands for an already optimized game engine are so high that our implementations are not expected to be seen in a game anytime soon. However, we believe that our results can be seen as an excellent base for decision making when wanting to move on to more advanced effects maybe together with new GPU techniques.

7.1.1 Implemented Methods

Our version of the SSD method still has a bit to go if we are to achieve perfect visual quality. The performance of the method is too slow to be run in real-time, but in a couple of years this is likely to change. Solutions for the arisen artifacts has been presented in [11] and could be implemented in our system if calculation time was not of such importance. Already with half the work done, the method is too time consuming for what we were searching for. Furthermore, solving the matrix system in the SSD method only works for special resolutions, which might cause problems when put into a game engine. Doing an efficient version using CUDA's shared memory is limited by the memory size and amount of registers and these problems cannot be ignored when choosing method. However, with hardware improvements and sufficient optimization this can definitely be an algorithm to count on later on, much thanks to its endless blur capability.

Using a method such as MPAD works very well and could be utilized if there is more computing power available. We must keep in mind that using higher resolutions will require more passes if we are to obtain a convincing DOF effect.

The Poisson disc approach will most likely be the one used in the near future and there are improvements to consider here as well if more GPU clock cycles are available. The linear interpolation between the real image and a down sampled version is effective in speed, but the result is not truly realistic. To totally abandon the scaled image, and instead sample all values used from the full scale image, taking more samples and averaging their values dependent of the depth-map would most likely reduce the halo artifacts with the tradeoff of performance. However, if a tradeoff of greater magnitude is needed, perhaps using a diffusion based method should be considered instead.

7.2 Future Work

The implemented and presented DOF algorithms in this thesis do give room for further improvements. We must simply acknowledge the fact that bleeding artifacts is the main problem for all DOF algorithms, and there is no easy solution to this matter when real-time performance is essential, and furthermore, when real-time performance in an already time-optimized game engine is needed.

This section will discuss future work and things that are important to consider when it comes to implementing DOF effects in the coming years. The first part goes through our applications and how they can be further improved and the second part discusses the future of GPGPU programming.

7.2.1 Depth-of-Field Algorithms

For our implementations we had only the output image and the depth-map to our disposal, limiting the work that could be done. If we instead could render parts of the scene, and separate the background, foreground, and midrange objects in the view, we could do multiple passes and blur the different parts independent of each other and achieve even more realistic results, minimizing the leaking from

in focus objects onto the background. These types of solutions would most likely improve the visual quality, but would interfere with the other parts of the rendering pipeline, being hard to integrate into current game engines. On the same time these solutions would have problems in the mid-range area; i.e., objects belonging to both foreground and midrange will be hard to filter when the COC change significantly over a single geometry.

When it comes to future implementations for us, we do consider working more on the diffusion based approaches, as they have the ability to create huge blurs and make effects based on thicker lenses as seen in movies, minimizing bleeding with the insulation criteria. On the same time creating smaller DOF effects using the Poisson disc approach or similar is of greater interest in the near future because of their speed and improving these is therefore also of importance for us.

7.2.2 General-Purpose Computation

Computer graphics has for a longer time been favored with the power of the GPU. But as many other fields now have discovered the potential to use this additional processor for other computations, it is likely that new tools for general purpose computations are going to be added.

Tools like CUDA and CAL are already gaining popularity and with this there will be new demands and possibilities of what applications that can be created. The problem here is that they are vendor specific and will therefore not push for an industrial type of usage. Computer games companies like DICE need to make effects that work for a broad collection of hardware, thus programming effects for specific hardware is not too accepted. Conversely, even today's game consoles require somewhat specific code solutions and a console's exclusive abilities are used as sales arguments. However, on the desktop side of gaming, such hardware variations are more substantial and much harder to adapt to.

As with the new geometry shader there is also not unlikely that supplementary shaders will be introduced later on, i.e., shaders that are more general between different hardware and for a general purpose that requires more general computations.

What really speaks against doing post-processing with a GPGPU technique instead of a shader is mainly that if the algorithm remains an image processing problem, rather than a general calculation, the pixel shader will normally work more than well.

A wish from our side would be for a numerous of different shader types that can work side by side, and if possible, with shared data. This would keep the strict but efficient pipe for graphics applications but at the same time give the programmer freedom to utilize the GPU as a coprocessor for a variety of parallel computations.

Bibliography

[1] M. Bertalmío, P. Fort, and D. Sánchez-Crespo. Real-time, accurate depth of field using anisotropic diffusion and programmable graphics cards. *3D Data Processing, Visualization and Transmission 2004 (3DPVT 2004)*, 2004.

[2] D. Ebert, K. Musgrave, D. Peachey, K. Perlin, and S. Worley. *Texturing & Modeling, A procedural Approach*, pages 97–132, 2003. Morgan Kaufmann. 2003. ISBN 1-55860-848-6.

[3] D. Gillham and C. Brighton. Real-time depth-of-field implemented with a postprocessing-only technique. *ShaderX5: Advanced Rendering Techniques*, pages 163–175, 2006. Charles River Media. 2007. ISBN 1-58450-499-4.

[4] R. Gonzalez and R. Woods. Digital image processing. pages 164–165, 2002. Prentice-Hall, Inc. 2002. ISBN 0-201-18075-8.

[5] E. Hammon Jr. Practical post-process depth of field. *GPU Gems 3*, pages 583–605, 2007. Addison-Wesley. 2007. ISBN 978-0-321-51526-1.

[6] M. Harris, S. Sengupta, and J. D. Owens. Parallel prefix sum. *GPU Gems 3*, pages 851–876. Addison-Wesley. 2007. ISBN 978-0-321-51526-1.

[7] M. Harris, Y. Zhang, and J. Owens. Scan primitives for gpu computing. *Graphics Hardware 2007*, 2007.

[8] D. Heller. A survey of parallel algorithms in numerical linear algebra. *SIAM Review, Vol. 20, No. 4*, pages 740–777, 1978.

[9] J. Hensley, T. Scheuermann, G. Coombe, M. Singh, and A. Lastra. Fast summed-area table generation and its applications. *Eurographics 2005 (EG'05)*, 2005.

[10] S. Hillaire, A. Lécuyer, R. Cozot, and G. Casiez. Depth-of-field blur effects for first-person navigation in virtual environments. *Proceedings of the 2007 ACM symposium on Virtual reality software and technology*, 2007.

[11] M. Kass, L. Lefohn, and J. Owens. Interactive depth of field using simulated diffusion on a gpu. *Pixar Technical Memo 06-01, Pixar Animation Studios*, 2006.

[12] M. Kass and G. Miller. Rapid, stable fluid dynamics for computer graphics. *Computer Graphics (Proceedings of SIGGRAPH 90)*, pages 49–57, 1990.

[13] M. Kraus and M. Strengert. Depth-of-field rendering by pyramidal image processing. *Eurographics 2007 (EG'07)*, 2007.

[14] M. Kraus and M. Strengert. Pyramid filters based on bilinear interpolation. *GRAPP 2007*, 2007.

[15] M. Kraus, M. Strengert, and T. Ertl. Pyramid methods in gpu-based image processing. *Proceedings Vision, Modeling, and Visualization 2006*, pages 169–176, 2006.

[16] NVIDIA. Geforce 8800 product information. Website, 2008. *http://www.nvidia.com/page/geforce_8800.html*, Last accessed: 2008-05-21.

[17] NVIDIA. Nvidia cuda programming guide 2.0 beta. 2008.

[18] G. Riguer, N. Tatarchuk, and J. Isidoro. Real-time depth of field simulation. *ShaderX2: Shader Programming Tips and Tricks with DirectX 9*, pages 529–556, 2003. Wordware Publishing, Inc. 2004. ISBN 1-55622-988-7.

[19] E. Santos. Optimal and efficient parallel tridiagonal solvers using direct methods. *The Journal of Supercomputing*, pages 97–115, 2004. Kluwer Academic Publishers.

[20] T. Scheurmann and N. Tatarchuk. Improved depth-of-field rendering. *ShaderX3: Advanced Rendering with DirectX and OpenGL*, pages 363–377, 2004. Charles River Media. 2005. ISBN 1-58450-357-2.

Appendix A

Result Images

Figure A.1. Test scene 1: Original color image and depth map.

Figure A.2. Test scene 1: Comparison between different DOF methods. Top: Result with Poisson disc method. Middle: Result with MPAD method. Bottom: Result using SSD approach.

Figure A.3. Test scene 2: Original color image and depth map.

Figure A.4. Test scene 2: Comparison between different DOF methods. Top: Result with Poisson disc method. Middle: Result with MPAD method. Bottom: Result using SSD approach.

Figure A.5. Test scene 3: Setup with different DOF settings computed with MPAD. Top: Car in focus. Middle: Soldier in focus. Bottom: Weapon in focus.

Appendix B

Pseudo Code

This is the simplified HLSL code for the MPAD algorithm, where the COC values for the pixel and two neighbours (left and upper) are already sampled into $txNeighbourDepth$ as r,g and b, and the rendered image is stored in $txColorBuffer$. Here we also have integer steps in texture coordinates where $texCoord$ is the current position.

```
- Fetch color data
1:  I  = txColorBuffer.Sample(SamplePoint, texCoord);
2:  IR = txColorBuffer.Sample(SamplePoint, texCoord + (1,0));
3:  IL = txColorBuffer.Sample(SamplePoint, texCoord + (-1,0));
4:  IT = txColorBuffer.Sample(SamplePoint, texCoord + (0,-1));
5:  IB = txColorBuffer.Sample(SamplePoint, texCoord + (0,1));

- Calculate derivatives
6:  gradPlusPlusU  = IR - I;
7:  gradPlusPlusV  = IB - I;
8:  gradPlusMinusU = I  - IL;
9:  gradPlusMinusV = I  - IT;

- Sample COC values
10: g = txNeighbourDepth.Sample( SamplePoint, texCoord);

- Do the heat diffusion and return
11: gradMinus = g.r*gradPlusPlusU - g.g*gradPlusMinusU
              + g.r*gradPlusPlusV - g.b*gradPlusMinusV;
12: return (I + 0.25 * gradMinus);
```

This is the simplyfied CUDA code for the SSD algorithm's row pass, where we begin with the down pass, followed by the up pass. *indexX* and *indexY* are the current coordinates on a 2D-grid. The tridiagonal matrices are already stored in the rows of the abc texture.

```
- Down pass
- Set data coordinates
1:  i_2j    = indexX*2;
2:  i_2j_p1 = i_2j + 1;
3:  i_2j_p2 = i_2j + 2;

- Sample abc values
4:  abc_2j    = tex2D(tex_abc, i_2j, indexY);
5:  abc_2j_p1 = tex2D(tex_abc, i_2j_p1, indexY);
6:  abc_2j_p2 = tex2D(tex_abc, i_2j_p2, indexY);

- Sample color values
7:  y_2j    = tex2D(tex_color, i_2j, indexY);
8:  y_2j_p1 = tex2D(tex_color, i_2j_p1, indexY);
9:  y_2j_p2 = tex2D(tex_color, i_2j_p2, indexY);

- Calculate alpha and gamma
10: alpha = abc_2j_p1.x / abc_2j.y;
11: gamma = abc_2j_p1.z / abc_2j_p2.y;

- Store abc values
12: d_odata_abc[index].x = -alpha*abc_2j.x;
13: d_odata_abc[index].y = abc_2j_p1.y - (alpha*abc_2j.z + gamma*abc_2j_p2.x);
14: d_odata_abc[index].z = -gamma*abc_2j_p2.z;
15: d_odata_abc[index].w = 1.0;

- Store color value
16: d_odata_color[index] = y_2j_p1 - (alpha * y_2j + gamma * y_2j_p2);

- Normalize if last row
17: if (row_length == 2) d_odata_color[index] = d_odata_color[index] / d_odata_abc[index].y;



- Up pass
- Set small coordinate
1:  indexX_small = indexX / 2;

- Sample abc and color values
2:  abc_j   = tex2D(tex_abc, indexX, indexY);
3:  y_j     = tex2D(tex_color, indexX, indexY);
4:  y_jp    = tex2D(tex_color_small, indexX_small, indexY);
5:  y_jp_m1 = tex2D(tex_color_small, indexX_small - 1, indexY);

- Check if odd index and save color value
6:  if (indexX % 2  == 1)
7:    d_odata_color[index] = y_jp;
8:  else
9:  {
10:   d_odata_color[index] = y_j - abc_j.z * y_jp - abc_j.x * y_jp_m1;
11:   d_odata_color[index] = d_odata_color[index] / abc_j.y;
12: }
```