
Linköpings universitet

# Depth Of Field by Simulated Heat Diffusion

Johan Beck-Norén, johbe559
Andreas Valter, andva287
Axel Kinner, axeki412

TSBK03 - Techniques for Advance Computer Games

2012-12-21

## Abstract

The methods that are presented in this report describes the theory and implementation for a interactive depth of field based on heat diffusion.

The technique uses heat diffusion to decide how the color gets distributed in each frame. A so called circle of confusion is used to calculate how far from the camera focus point each fragment is. Fragments that are in focus acts as thermal insulators that does not transmit or receive any influences from any other pixels. When the circle of confusion increases that fragment is a better conductor.

To solve the heat equation, the ADI method is used which is based on a division in a horizontal part and a vertical part and uses an implicit method to be able to take an arbitrary time step and still have a stable solution. This results in the need to solve an inverse of a tridiagonal matrix which is done using the Jacobi method.

The implementation is made in C++ using the OpenGL API to take advantage of rendering, as well as shading techniques for post processing and general computing. Ping-ponging between frame buffer objects is used for the Jacobi solver to converge its solution toward the true matrix inverse.

# Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Background

Depth of field is a phenomenon known by most from the field of photography. The distance between the nearest and farthest objects in a scene that, to the viewer, appears sharp and in-focus, are said to be within the scene's depth of field. The technique is widely used in cinematography and photography to draw the viewers focus to certain objects in a scene, or to give a sense of depth. When producing high quality computer graphics, depth of field is essential to get one step closer to a photo-realistic result, and enables game creators to move towards films in their storytelling and presentation.

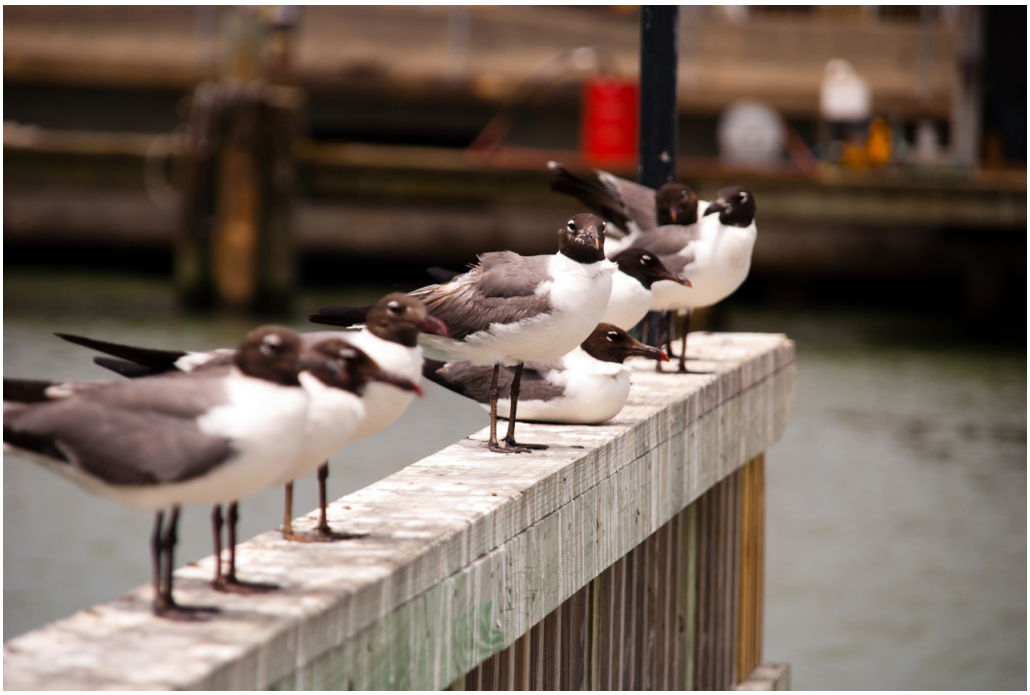


Figure 1.1: Depth of field in a photograph.

The out-of-focus parts of an image with a short depth of field comes as a result from a camera's aperture size. A point that is out of focus will occupy a larger area on the camera's sensor,

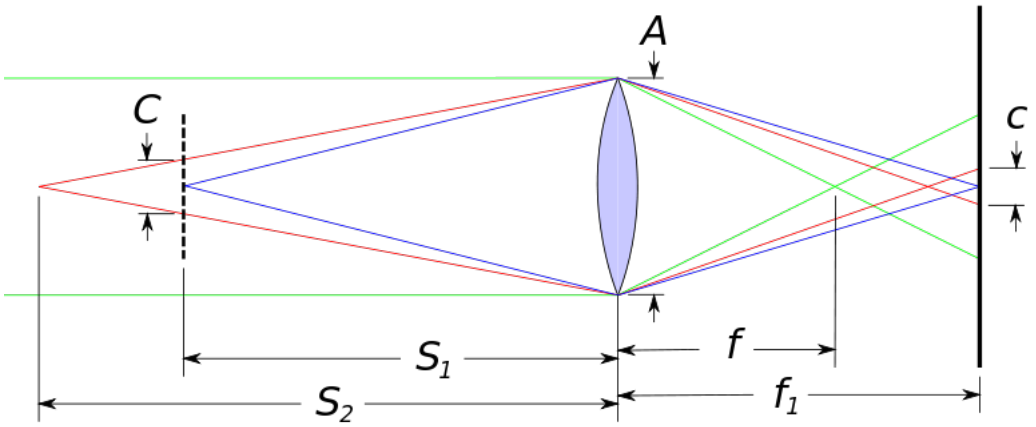



Figure 1.2: Relationship between camera parameters and circle of confusion.

instead of just one focused point as an in-focus point would. The area of the sensor which an out-of-focus point will occupy is described by that point's Circle of Confusion (CoC).

The depth of field for a point is proportional to the size of that point's CoC, as it describes how much out-of-focus an arbitrary point is. As seen in figure 1.2, the CoC for a point is calculated from a camera's lens diameter, focal length and focal point, as well as the distance from the lens to the point for which we seek its CoC. From these parameters, the CoC $c$ for a point can be calculated, as seen in equation (1.1).

$$c = A\frac{|S_2 - S_1|}{S_2}\frac{f}{S_1 - f} \tag{1.1}$$

## 1.2 Tools

### 1.2.1 OpenGL and GLSL

OpenGL is a cross-platform API supported by Khronos group to develop computer graphics. With it, it is possible to convert a data representation to a 3D image that is displayed on the screen and to control the graphics pipeline to some extent. It is used in various applications, both in visualisation, CAD and video games.

GLSL is the shading language that is connected to OpenGL. It is used to control the graphics pipeline without having to use hardware-specific languages. Depending on the version of OpenGL, GLSL allows you to control different parts of this pipeline. The two parts that have been used in this application is the vertex shader and the fragment shader.

The vertex shader runs once for each vertex and is responsible for transforming the vertices from world-space coordinates to screen-space coordinates. It is possible to manipulate vertex positions and normals, but not to create any new vertices. This information will later be passed to the fragment shader.

The fragment shader is responsible for computing the output color of each fragment. This is where the lighting calculations are performed. The fragment shader can also be used for post-processing effects where an image is loaded as a texture and fragment transformations are done. The fragment shader can also be used as a general computing unit. If the data is stored in textures, it is possible to do calculations on it in parallel on the GPU and benefit from the high compute capability of a GPU. For data that is easily mapped to a image with 4 channels, it is a good alternative.

## 1.3 Previous work

There are many implementations of depth of field in computer graphics. A ray-tracing method would result in the most physically correct result. As a post-process method, depth of field behaves much like a scattering method. Scattering produces a high quality result, but it requires a sorting of pixels by depth. It is also difficult to preserve image intensity and a high precision is needed when blending. Another offline approach is to use a variant of an accumulation buffer. This simply means moving the computer camera within the simulated camera's aperture opening and performing several render passes, usually 40 to 60 passes, and then blending the results. None of these methods are suitable for real-time applications such as computer games.

For real time performance, one option is to use gather techniques. This method has great speed performance, but has problems with out-of-focus areas bleeding its color values over areas in sharp focus. This technique also runs into problems when using large filter kernels for large blurs. A few documented gather methods used in depth of field implementations are Poisson Disk Blur and Summed Area Tables, Unger et al. [2].

The implementation that was chosen to focus on in this report is using heat diffusion to simulate depth of field, as described in Kass et al. [1]. Depending on the implementation, this method eliminates the problems associated with gather techniques as it allows for sharp edges between in-focus and out-of-focus objects, as well as arbitrarily large filter kernels in constant time.

# Chapter 2

# Diffusion Depth of Field

## 2.1 Heat diffusion

The heat equation (2.1) describes the distribution of heat over time for a specific point. The method described in this report takes advantage of this equation to distribute fragment color values according to each fragment's CoC-value. Fragments that are fully in focus are considered thermal insulators. This means that they neither contribute to other fragment's values or receive any contributions from other fragments. Fragments with a non-zero CoC will allow for a greater heat transfer, and will therefore give and receive more fragment values, resulting in a larger blur as the CoC increases. This is a way to be sure that there is no edge bleeding over objects that are in focus from objects that are behind by taking advantage of the varying filter length that this technique enables.

$$\gamma(u,v)\frac{\partial y}{\partial t} = \nabla \cdot (\beta(u,v)\nabla y)) \tag{2.1}$$

There are several possibilities when solving this equation numerically where the simplest method is to evaluate the equation at time zero and iterate using forward Euler steps (2.2). This will however limit the quality of the solution, depending on the length of each time-step. However, this solution requires $n^2$ iterations to produce a filter width proportional to $n$. Because of the time complexity for this, $O(n^2)$, there is a need to look to other methods for a solution to this problem.

$$y(\Delta t) = y(0) + \Delta t(\partial y/\partial t) \tag{2.2}$$

$$\gamma\frac{\partial y}{\partial t} = \frac{\partial}{\partial u}\beta(u)\frac{\partial y}{\partial u} \tag{2.3}$$

The Alternate Direction Implicit (ADI) method instead creates separable IIR filters that are independent of the timestep. Therefore, it is possible to find an approximate solution to the heat equation, resulting in a spatially varying filter width without having to iterate. The ADI method splits the equation in two parts, solving the heat equation in the $u$ dimension and $v$

dimension separately. Equation (2.3) describes diffusion along the $u$-axis. Each sub-step is calculated using an implicit method. The method used in this report is the backwards Euler method because of its simplicity while still providing a solution that is stable for all time-steps.

As described in Kass et al. [1] it is possible to map the heat equation (2.1) to the DoF problem. This is done by first assuming that $\beta$ is uniform and that $\gamma$ is unit over each time-step which results in equation (2.5). It is then possible to calculate the frequency response, which is the same as the frequency response of a Butterworth low-pass filter (2.4). This shows that $\beta = d^2$ where $d$ equals the radius of a fragment's CoC. To ensure that no energy dissipates over edges of the image and that the border fragments are not affected by outside fragments, $\beta_0 = \beta_n = 0$ which states that the borders of the CoC act as thermal insulators.

$$\tilde{y} = \frac{1}{1 + (\omega/\omega_c)^2}\tilde{x} \tag{2.4}$$

$$\gamma_i \frac{\partial y}{\partial t} \approx \beta_i(y_{i+1} - y_i) - \beta_{i-1}(y_i - y_{i-1}) \rightarrow y_i - x_i = \beta(y_{i+1} - 2y_i + y_{i-1}) \tag{2.5}$$

$$\frac{-\beta_{i-1}}{\gamma_i} y_{i+1} + \frac{\gamma_i + \beta_i + \beta_{i-1}}{\gamma_i} y_i + \frac{-\beta}{\gamma_i} y_{i+1} = x_i \tag{2.6}$$

After re-factoring this equation, a tridiagonal equation system is created (2.7) where the elements in the matrix are described in equations (2.7b) - (2.7e). The solution to the problem is therefore reduced to solving the inverse of the matrix for each row and column of the image. This method is now only dependent on the time-frame to solve the matrix inverse. And if a fast way to solve this is used, a high speed-up is achieved, resulting in a solution where the filter size can vary without affecting performance. Due to the fact that the ADI method separates the layers, the resulting image may suffer from anisotropic artefacts.

$$\begin{pmatrix} b_1 & c_1 & & & 0 \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & c_3 & \\ & & \ddots & \ddots & \ddots \\ 0 & & & a_n & b_n \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} \rightarrow AY = X \tag{2.7a}$$

$$a_i = \frac{-\beta_{i-1}}{\gamma_i} \tag{2.7b}$$

$$b_i = 1 + \frac{\beta_{i-1} + \beta_i}{\gamma_i} \tag{2.7c}$$

$$c_i = \frac{-\beta_i}{\gamma_i} \tag{2.7d}$$

$$c_i = a_{i+1} \tag{2.7e}$$

## 2.2 Inverse matrix solver

Equation (2.7a) shows that the core of the ADI method solution of heat diffusion is to obtain $y$ by calculating the inverse of the tridiagonal matrix $A$.

There are a variety of solvers for tridiagonal matrices, one common method is LU decomposition. However, since our solution shall be in real time with a number of frames per second, it requires that the calculation of the inverse is done on the graphics card for performance reasons. LU decomposition is a very fast method for sequential programs but parallelizes poorly and is therefore not a good solution to this problem. A method recommended by Kass et al. [1] is cyclic reduction which is a divide and conquer method. This method parallelizes well and is therefore a good solution to the problem. However, the method requires a pyramid structure of textures, resulting in high usage of video card memory. Zhang et al. [3] shows a hybrid solver of cyclic reduction and parallel cyclic reduction to be even faster. But the method that was chosen was the Jacobi method, Black et al. [4]. It solves the equation approximately by iterating toward the correct solution. The advantages of this method is that it is easy to parallelize and implement using shaders, but has the disadvantage that it converges slowly. It is quite fast anyway since the calculations are done on the GPU. The Jacobi method also requires that it meets the requirement of equation (2.9), to ensure that the solution converges.

The Jacobi method is not only a solver for tridiagonal matrices, but a solver for arbitrary matrices. Its definitions are described in equations (2.8) (2.9), where $y_i^k$ is solution $k$ of $y_i$, $b_i$ is an arbitrary value used as an initial value and $A$ is the tridiagonal matrix. Equation (2.8) can be simplified to equation (2.10) because in this case it is only used to compute a tridiagonal matrix. $A_{ii-1}$, $A_{ii}$, and $A_{ii+1}$ are equal to $a$, $b$ and $c$ in equation (2.7). Using these values results in equation (2.11). This shows that the equation always converges, leading to equations (2.12).

$$y_k^i = \frac{b_i - \sum_{j \neq i} A_{ij} y_j^{k-1}}{A_{ii}} \tag{2.8}$$

$$|A_{ii}| > \sum_{j \neq i} |A_{ij}| \tag{2.9}$$

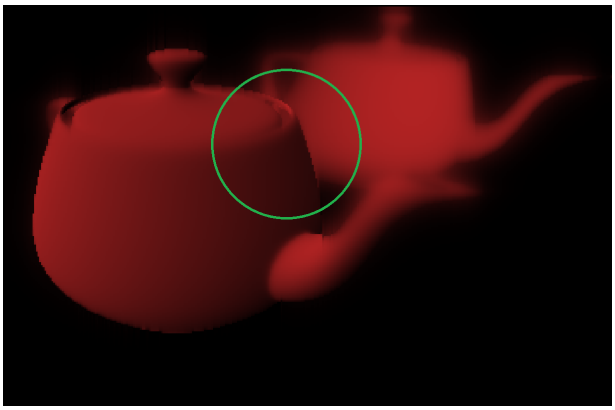
$$y_k^i = \frac{b_i - (A_{ii-1} y_{i-1}^{k-1} + A_{ii+1} y_{i+1}^{k-1})}{A_{ii}} \tag{2.10}$$

$$y_k^i = \frac{b_i - (\frac{-\beta_{i-1}}{\gamma_i} y_{i-1}^{k-1} + \frac{-\beta_i}{\gamma_i} y_{i+1}^{k-1})}{1 + \frac{\beta_{i-1} + \beta_i}{\gamma_i}} \tag{2.11}$$

$$|1 + \frac{\beta_{i-1} + \beta_i}{\gamma_i}| > |\frac{-\beta_{i-1}}{\gamma_i}| + |\frac{-\beta_i}{\gamma_i}| \tag{2.12a}$$

$$\beta \geq 0 \tag{2.12b}$$

$$\gamma > 0 \tag{2.12c}$$



(a) The figure shows the midground to background insulator problem.

(b) The figure shows the foreground to midground insulator problem.

Figure 2.1: The figures shows the Heat insulator problems that comes with Heat diffusion depth of field. Both problems is centered by a circle.

## 2.3 Separable layers

The figures 2.1a and 2.1b shows that the implementation has some limitations. Both of these problems are due to the fact that in-focus objects acts as insulators which will stop the blur from being spread over or behind the in-focus object.

Kass et al. [1] describes two methods for improving the quality of the implementation, but at the expense of performance. The methods describe how it is possible to eliminate these problems by using $\gamma$, and to make more diffusion calculations, e.g. on the CoC texture.

## 2.4 Implementation

For an image of size $n \cdot m$ pixels, a total number of $n \cdot (m \cdot m)$ tridiagonal matrices for the horizontal equations needs to be used, while the vertical equations needs $m \cdot (n \cdot n)$ number of tridiagonal matrices. Saving all of these is both a waste of memory and computing power. It can be seen from equation (2.7) that it is possible to calculate these values directly from the CoC texture. See figure 2.2

Since values $A_{ii-1}$, $A_{ii}$ and $A_{ii+1}$ from the Jacobi equation are known, it is only $b_i$ from the same equation that needs to be determined. Since the image is based on an original image to be blurred, it is logical to use the original image fragment values as a starting point (2.13).

$$y_k^i = \frac{x_i - (\frac{-\beta_{i-1}}{\gamma_i} y_{i-1}^{k-1} + \frac{-\beta_i}{\gamma_i} y_{i+1}^{k-1})}{1 + \frac{\beta_{i-1}+\beta_i}{\gamma_i}} \tag{2.13}$$

This means that $y_i^0$ is $x_i$. For the main implementation $\gamma_i$ will be unit as stated in Kass et al. [1].

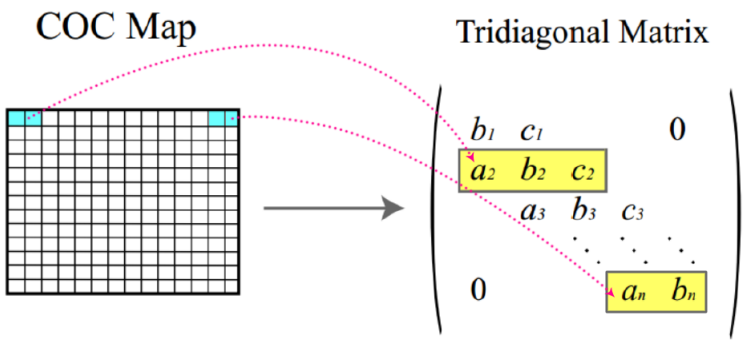



Figure 2.2: The figure shows how the CoC texture is mapped to one tridiagonal matrix in the horizontal part of the solution. Image courtesy of [2].

### 2.4.1 OpenGL and GLSL shaders

It is now possible to solve the equation (2.7) with equation (2.13), by sending in three different textures to the shader; a texture that contains the original image, a texture containing CoC values, and a texture the solution from a previous iteration. Figure 2.3 illustrates this.

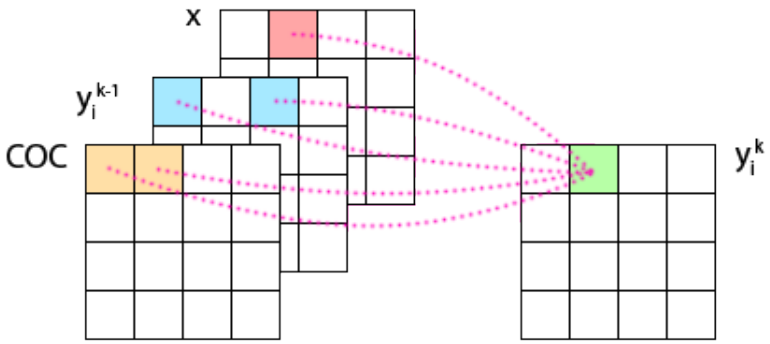



Figure 2.3: An illustration of how the textures interacts with the shader in the horizontal part of the solution. Where $x$ is the original image, $y_i^{k-1}$ is the previous result, $COC$ is the CoC texture and $y_i^k$ is the result from the shader of all these.

GLSL code for the Jacobi shader:

```
void main(void)
{
    vec2 offset2d;
    if(vertical)
        offset2d = vec2(1.0f/width, 0.0f);
    else
        offset2d = vec2(0.0f, 1.0f/height);

    vec4 bethaPrev = texture(coc, texCoord - offset2d);
```

```
    vec4 betha = texture(coc, texCoord);

    vec4 a = -bethaPrev;
    vec4 b = 1 + bethaPrev + betha;
    vec4 c = -betha;

    vec4 yr = texture(oldY, texCoord + offset2d);
    vec4 yl = texture(oldY, texCoord - offset2d);
    vec4 y0 = texture(original, texCoord);

    out_Color = (y0 - (a * yl + c * yr))/b;
}
```

### 2.4.2 Separated layers

In section [2.3] a few artefacts were described as a result of viewing the Depth of Field problem as a heat equation. It also refers to Kass et al. [1] which had a solution for these problems. To eliminate the problems with out-of-focus objects in the foreground, some ideas from their method is used. The first step is to render the scene twice, once when rendering foreground objects and a second time when the rest of the scene is rendered. The CoC for each component are then used to perform diffusion calculations on both color and alpha values. Finally, the results are blended together to form a final image by drawing out the foreground and background weighted by the foreground alpha values.

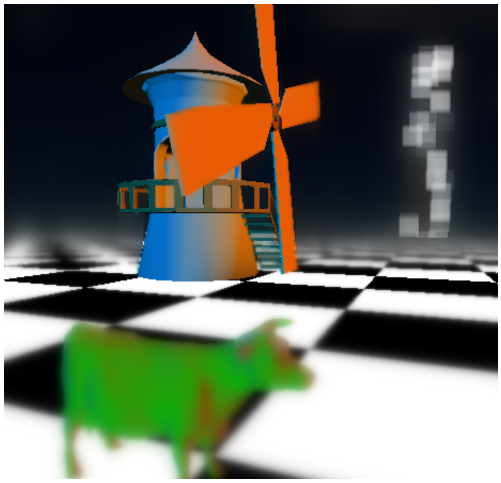
Attempts to implement the full algorithm from Kass et al. [1] have been made in MATLAB as follows:

1. Render the midground and background part of the scene, or the whole scene if the foreground scene is not calculated separately.
2. Calculate the CoC texture from the rendered image.
3. Do a diffusion computation on the image using the CoC texture.
4. Divide the image in a midground and a background part with a threshold. This is done by calculateing gamma for each pixel, gamma is one for pixels with a CoC bigger than the threshhold value (bakground) and ramps down to zero for pixels in full focus. But because it is not suitable to divide with zero, a value close to zero has been used instead.
5. Do a diffusion computation on the CoC texture using the threshold value as CoC value.
6. Use the diffused CoC texture for compute the diffusion on the background.
7. Blend the result in 3. and 6. using the alpha values taken from gamma in 4.

### 2.4.3 Bokeh

In photography, bokeh refers to the effect produced when a light source is out of focus. The result is a light-splatter the same color as the light source. The shape of a bokeh effect is

(a) Bokeh effects seen in out-of-focus background.



(b) Square shaped bokeh effects seen in out-of-focus background.

Figure 2.4: The figures show bokeh in a real photograph and an implementation using gather.

determined by the shape of the camera's aperture, usually different degrees of $n$-gonal shapes. Through different manipulations of the camera's lens, the bokeh effect can assume arbitrary shapes.

In a computer graphics context, this translates to a fragment with an intensity value greater than a set threshold and with a CoC greater than a set threshold should have a bokeh effect applied to it.

The easiest implementation of this would be to use a scatter technique, but using the graphics pipeline in the way that this report describes, scatter is not a practical method. One implemented approach was to use a Butterworth filter on the affected fragments, but the drop in performance with this technique was unacceptable. Using billboards, quads with textures and alpha blending, would have been the best approach for this project. Unfortunately the time constraints for the project did not allow us to implement this. Instead, a simple gather was implemented which resulted in bokehs with square shape with correct color according to the originating light source and correct intensity according to the fragment's CoC (2.4b).

# Chapter 3

# Results

## 3.1 Results

### 3.1.1 Diffusion depth of field

Figure 1 in appendix A shows the original input image at the top left corner, the COC map at the top right corner, the resulting depth of field at the bottom left and depth of field with bokeh effect at the bottom right. The COC map describes which fragments are in focus. A value of zero means that the fragment is totally in focus. The diffused images at the bottom half are rendered using 400 Jacobi iterations and camera parameters that correspond to a fairly short depth of field.

As seen in the COC map in figure 1 in appendix A the windmill is in focus, with the cow in the foreground slightly out of focus and the skyscraper in the background completely out of focus. From the diffused image it is clear that the area of focus is the windmill, and a sense of depth is added to the frame. Small anisotropic artefacts can be seen in the border between the cow's rump and checkerboard ground plane, as well as near the tip of the top-most rotor on the windmill. This occurs near the borders between areas with large differences in COC size. This is most likely because of the nature of the ADI solution used, and corresponds to the overshoot of an IIR filter.

### 3.1.2 Jacobi method performance

The Jacobi method is an approximate method that iterates toward an analytically correct solution. The result from the Jacobi method is greatly dependent on the number of Jacobi iterations, and the number of iterations in turn affects the framerate of the application. Figure 2 in appendix A shows how the Jacobi solution differs from the analytically correct solution of a 1024 by 1024 matrix by mean square error and maximum error. From this figure we see that the mean square error decreases by $2/3$ for every Jacobi iteration. Figure 3 in appendix A shows the difference between the Jacobi solution and the analytically correct solution when using ten iterations and varying the matrix size. The maximum error increases approximately squared for every increase in matrix dimensions by power of two, while the mean error increases and then converges toward a fixed limit.

While the Jacobi method was easy to parallelize and run on the GPU using GLSL shaders, it required a large number of iterations to reach convergence for very large filter kernels when using a short depth of field. We found that in order to get smooth blurring with minimal artefacts, around 400 iterations was necessary, and as a consequence the framerate suffered.

The data in table 3.1 was run a system consisting of an Intel Core i7-3770k CPU and an NVIDIA GeForce GTX 660 graphics card.

| No. of iterations | Time per frame (s) |
|---|---|
| 50 | 0.028 |
| 100 | 0.055 |
| 200 | 0.221 |
| 1000 | 0.600 |

Table 3.1: Jacobi iterations versus time per frame.

### 3.1.3 Separated layers

Figure 3.1: The figure shows the result of our fix of the foreground heat insulator problem. It can be seen that the method need a lot more work before it can be used.

In figure 3.2, we can see that the heat insulator problem for the cow has disappeared, but it can also be seen that the blending between the foreground and the midground/background layer is not entirely correct.

The result of the background separation in figure 2.1a shows that this algorithm also has some severe artefacts.

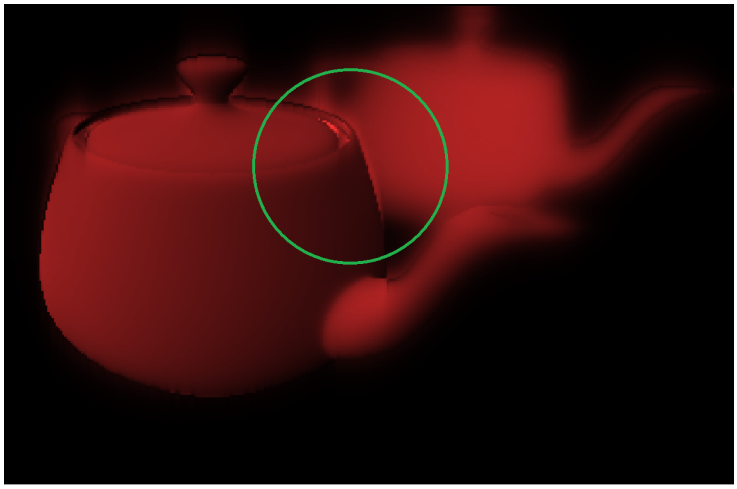


Figure 3.2: The result of the fix of the background heat insulator problem. The interesting part is inside the green circle. It can be seen that the result is worse than 2.1a

## 3.2 Discussion

The method that has been presented in this report solves a lot of the problems that previous work showed. However there is still a lot of problems that have to be solved to be able to use this method in an interactive environment.

It is possible to see anisotropic artefacts in the images that are produced which are a result from the fact that the ADI method solves the equation in one horizontal and one vertical part. A good subject for future work would be to see if there is an improvement if a more advanced method than backwards Euler is used. For the matrix solver that was presented in Kass et al. [1] this would have been a problem, but because our matrix solver does not require a tridiagonal matrix, it is possible.

Effects like the one that is presented in this report are low prioritized in a gaming environment and must therefore be fast. This might be a reason why we have not found any later reports that uses the diffusion method with separated layers. Due to the fact that it takes time and resources to render scenes, it is a problem when a method that is supposed to be a post processing technique requires one extra rendering pass and three or more extra diffusion computations. This is why separated layers may not be suitable for video games yet. The DoF effect is mostly used in video games with scenes with lots of movement in it, so the viewer will probably not notice artefacts like color bleeding that much. This combined with the fact that gather methods are much faster (Unger et al. [2]) is probably a reason why gather methods are more common in today's games.

The implementation of separated layers in this report was not completed because of several reasons. The main reason is that many attempts have been made in MATLAB without real success. One reason for that could be that we never succeeded to mimic Kass et al. [1] algorithm of the separation of layers. The second reason is that is was not prioritized because it was only a technique to improve the already working result. The foreground implementation in OpenGL in this report is also just a very early version and is not finished, a lot more work is need for this implementation to reach a good result.

## 3.3 Future work

The inverse matrix solver used in this report is an approximative solver that iterates toward the correct solution. If the cyclic reduction method was implemented as a solver instead, a stable and analytically correct solution would be obtained in constant time.

A problem that would be great to solve in the future is the separation of layers. Even though this technique is not yet suitable for video games, it is still very interesting. It would solve these problems that comes with the heat diffusion depth of field and could therefore find itself into games in the future.

For a better visual quality of the bokeh effects with arbitrary shapes, using geometry for the effects instead of post-processing techniques would have been favourable. The only viable post processing technique when using shaders as in this project is to perform a gather on affected pixels. In hindsight, using geometry in the form of billboards would have been a better alternative. This would include drawing quads in the position of pixels that should have bokeh effects applied. These quads would always be turned towards the camera and have a texture with alpha blending to allow for arbitrary colors and bokeh shapes.

# Appendix A

# Images

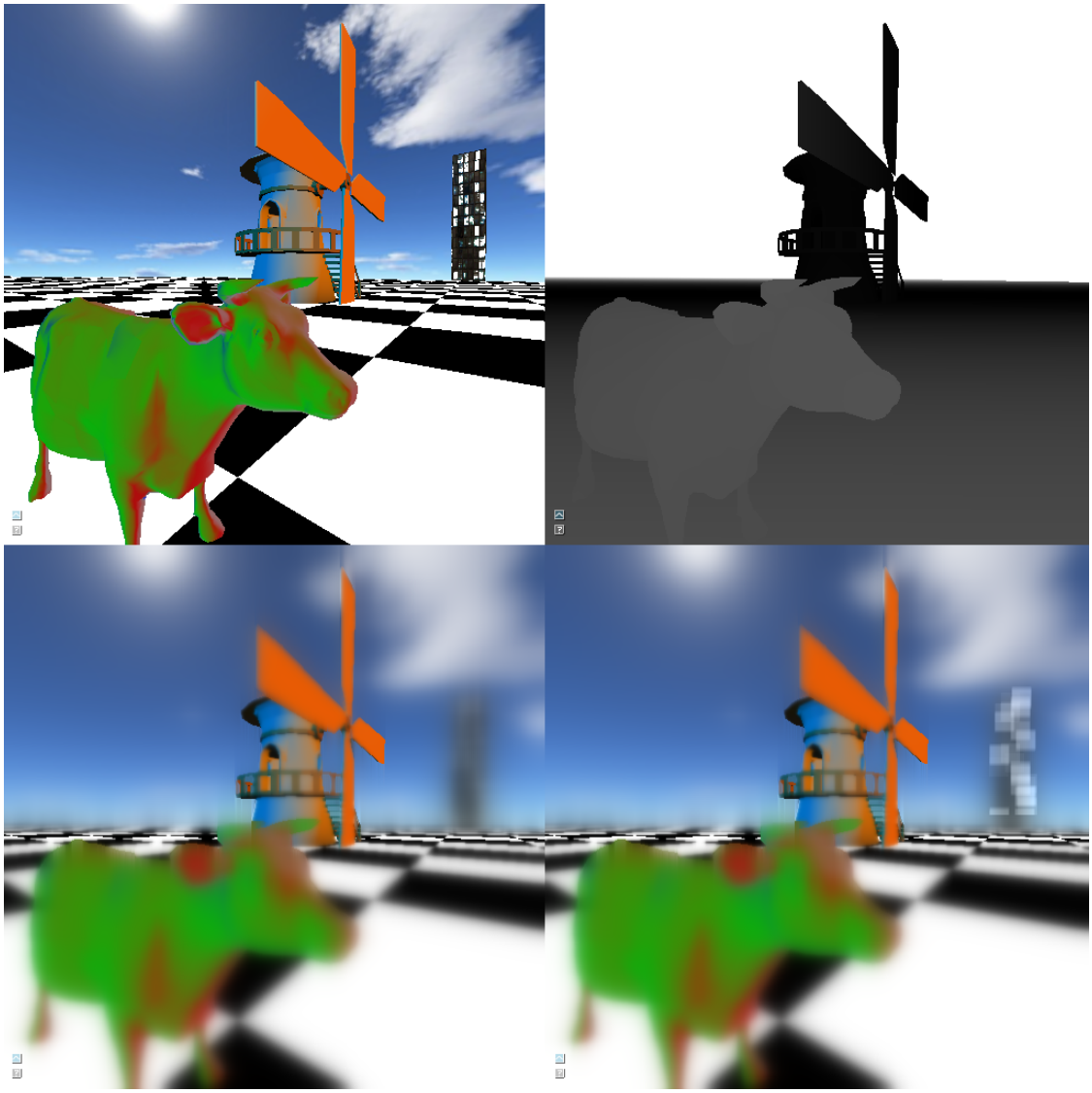


Figure 1: Input image top left, COC map top right, depth of field bottom left and depth of field with bokeh effects bottom right.

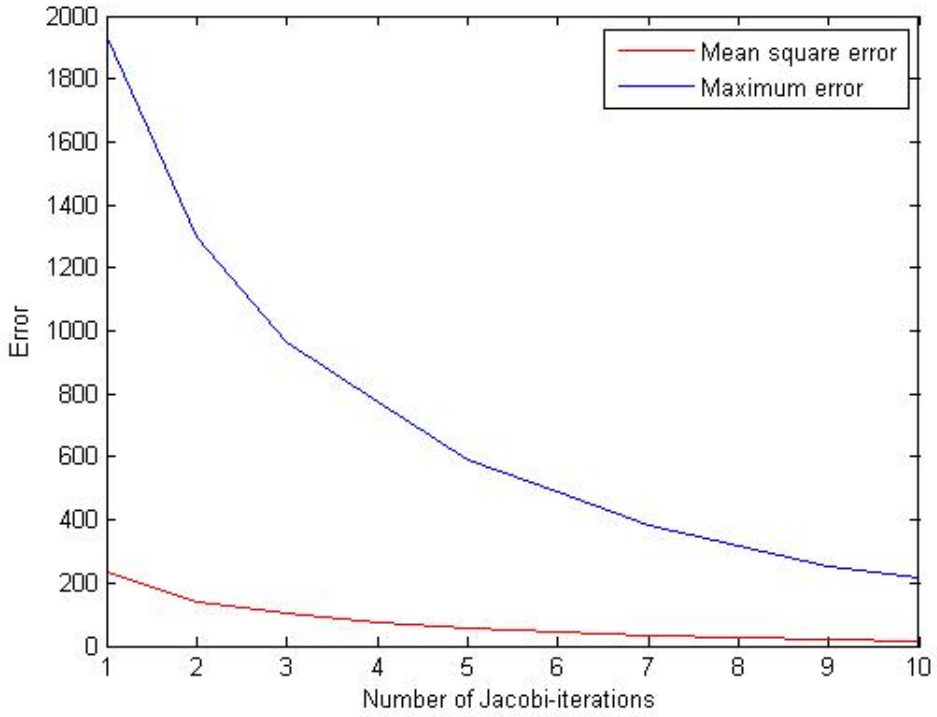



Figure 2: Error for a 1024x1024 matrix with varying number of Jacobi iterations.

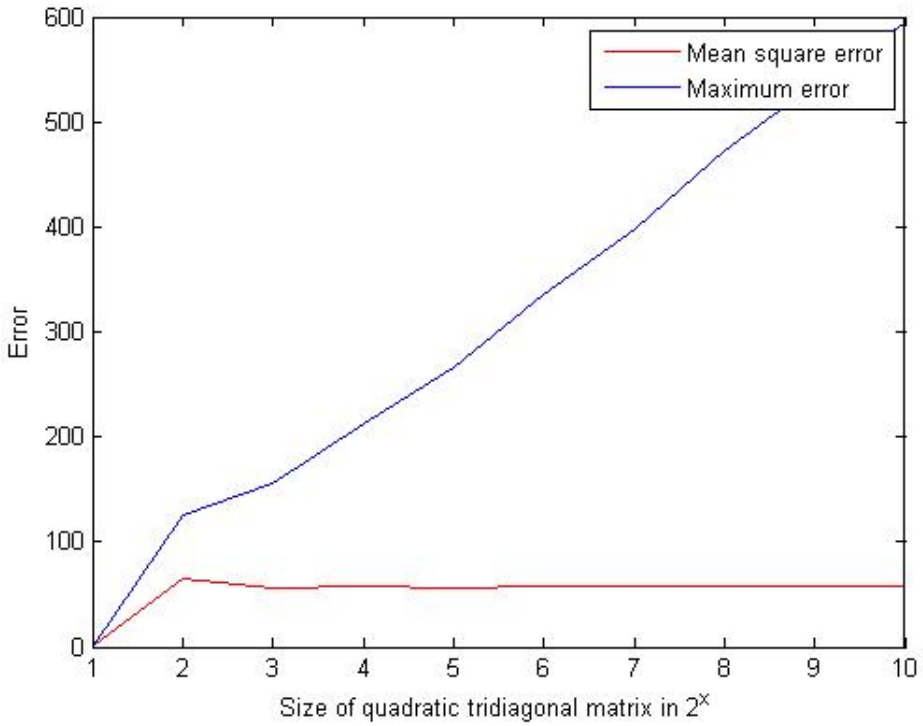



Figure 3: Error for a 10 iteration Jacobi solution with varying matrix dimensions.

# Bibliography

[1] *Interactive Depth of Field Using Simulated Diffusion on a GPU*. Kass, Lefohn, Owens, Pixar Animation Studios, 2006.

[2] *Advanced Real-time Post-Processing using GPGPU techniques*. Lönroth, Unger, DICE, 2009.

[3] *Fast Tridiagonal Solvers on the GPU*. Zhang, Cohen, Owens, NVIDIA, 2010.

[4] *Jacobi Method*. Black, Noel; Moore, Shirley; and Weisstein, Eric W. MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/JacobiMethod.html