Post Processing Effects

Bernhard Steiner[*]
Vienna University of Technology
05.08.2011

Abstract

Post processing effects, photo realistic ones as well as non photo realistic ones, are very important to modern computer games. This paper tries to give an overview of how post processing works and which methods can be used to perform them in real-time, giving hints how these special effects can be implemented on modern graphic hardware. Additionally, this paper gives an overview of algorithms used for special post processing effects like depth of field or motion blur, making them comparable to other implementations.

Keywords: post processing, bloom, glow, depth of field, motion blur, cel shading

1 Introduction

Since the 1972 when ATARI released the first widely used, commercial computer game ”Pong”, the world of video games has changed a lot. In early years it was enough to have a computer and a team of one or two people, now hundred or more people work on modern triple A productions. Also the appearance of the worlds and levels have changed a lot. In early years the levels looked sterile and unrealistic, today open world games render images that are very close to the ones taken by cameras in the real world. Modern graphic hardware allows the programmer to do stuff not even imaginable in the beginnings of video games. In the last few years the architecture of graphic cards made some further steps as allowing the game developers to execute their own shader code directly in hardware, making them able to create effects that increase the scenes realism dramatically.

One major benefit of modern graphic cards is the possibility of multi pass rendering, allowing us to perform so called ”Post Processing Effects”. Post processing describes the fact that these effects do not work with scene geometry, but on a previously rendered image of the scene. In that way post processing effects are what is often know as image manipulation or image processing equal to what filters in Photoshop or other image manipulation programs do. These types of effects allow designers to generate more realistic images due to the fact that real world camera models can be approximated. Another group of effects, the non photo realistic ones, make it possible to create images that look more artistic, for example like hand drawn comics. More on this can be found in section 6.

In general post processing effects work in the two dimensional image space, performing operations depending only on the information in this image, or some additional information that is present in the same space, as depth information or normal images. Examples for effects working only with image data are Glow (section 5) and Bloom (section 5.2). From a mathematical point of view a post processing effect is a transformation function $f(x,y)$ from a $n \times m \times l$ dimensional input matrix to a output matrix (O). The combination of a color input (three $n \times m$ matrix) and a depth image (one $n \times m$ image) can be seen as one input set of dimension $n \times m \times 4$ where n,m describe the resolution of the images:

$$I^{n\times m\times l} \xrightarrow{\text{post processing effect}} O \quad (1)$$

This paper will first give an overview of some techniques used for post processing (section 2) and then give a historical overview on some post processing effects like Depth of field (section 3) or Motion Blur (section 4) that improves the images realism. In the last section a short overview of a non photo realistic method, toon shading is presented. This paper aims on making different algorithms for the same effect comparable to each other, so there are also some effects described that are a little out of date. This is needed to show why and how modern implementations are better (or worse) then previous ones.

2 GPU implementation details

2.1 Per Pixel Filtering

Most post processing effects are defined by a function $f(x,y)$, that describes for every point (x,y) in an image, how it has to be changed for the output image. Applying this function f to the whole image can be performed very fast on the graphic hardware. To perform this, the original image must be rendered to a texture (in OpenGL this is done by using frame buffers), then this texture and, if needed, some other ones are bound to texture units. The drawing is done by rendering a full screen quad with texture coordinates in the range from 0 to 1, (figure 1(a)). Performing the described steps would copy the whole input image to a new output render target. The post processing effect f can now be applied in the fragment shader (or pixel shader in DirectX), because this program is executed for every pixel in the input image.

Reading texels in the neighbourhood of (x,y) can be done in two ways. The first is, that the size of the input texture is fixed, then the fragment shader program can read with a fixed offset. If this is not possible, the original texture can be bound to multiple texture units, each with another offset, so that all information for a pixel can be found at the same location on the different units (figure 1(b)).

2.2 Separable Convolution

Another task, which screen space post processing effects often need to perform, is filtering of an input image with a given filter kernel. This is, seen from a mathematical point of view, a convolution in spatial domain of the image matrix with the filter kernel matrix, or a multiplication if the image is transformed to frequency domain. As a convolution needs many texture lookups in order to read the

[*]e-mail: bernhard.steiner@tuwien.ac.at

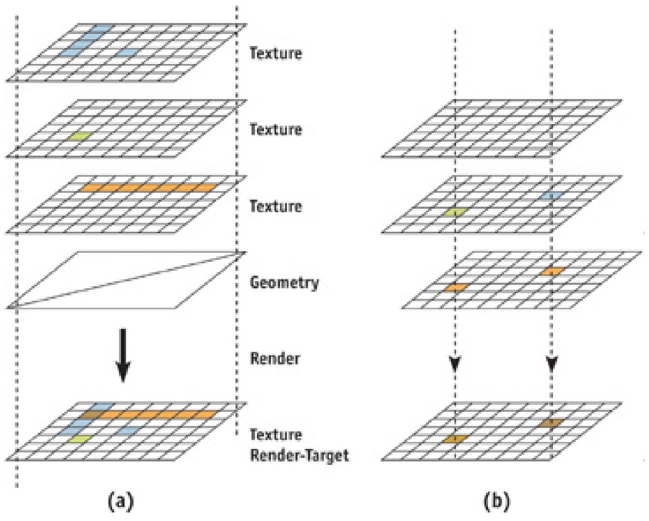

Figure 1: Per pixel post processing. (a) using full screen quads to perform per pixel post processing, (b) textures with different offsets, so that all information for a pixel can be found at one location (image from [Nguyen 2007])

neighbourhood of a given point, it is very expansive to calculate. For example an image of $n \times m$ convolved with a $k \times l$ filter kernel needs to perform $k \times l \times n \times m$ lookups.

[Nguyen 2007] shows that for some filter kernels (e.g. the Gaussian filter kernel) it is possible to decompose this two dimensional kernel into two one dimensional kernels. In general it is not necessary that the original kernel is mathematical decomposable, the result of the two kernels only needs to look correct. The convolution with two kernels is then performed by filtering first with one kernel and then filtering the result with the second one. As can be seen, the above example needs only $(k+l) \times (n \times m)$ lookups. Figure 2 shows the filter kernels and the resulting image of a decomposed Gaussian convolution.

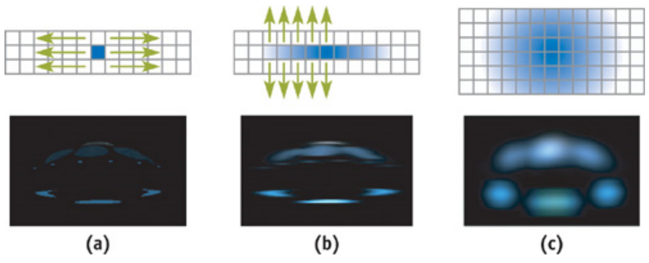

Figure 2: The Two-Step Separable Approach for Creating Blurs Efficiently (image and caption from [Nguyen 2007])

3 Depth of Field

A major problem, that is caused by the limitations of using a pinhole camera for rendering 3D scenes, is the fact that every object, regardless to the distance of the camera, is rendered in perfect focus. Real world cameras, including the human eye, always have a limited depth of field, due to the lens system they use. In order to render more realistic images, or to give the artists the opportunity

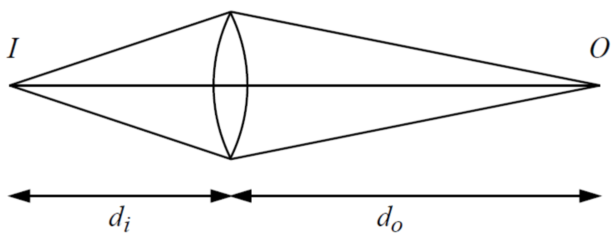

Figure 3: Thin lens system (image from [Mulder and van Liere 2000])

to control the focus of the viewers, it is necessary to add depth of field blurring.

3.1 General

When a pinhole camera is used, every point in the 3D scene is mapped to one point in the image plane. This causes all objects to have sharp edges. This is not desirable because a lens system would render sharp edges only in short range around the focus distance. The further points are away from this distance, the more they should become blurred. The human eye uses this fact to create a depth cue, helping the human to understand the order of objects in the scene. [Zhou et al. 2007]

3.1.1 Camera Models

To understand how this blurring arises, we will take a short look to a camera model which is a little more complex , the thin lens camera. In this model the lens system of real cameras is approximated by one lens (Figure 3). An object point O is refracted by the lens through the image point I.

As described by [Mulder and van Liere 2000], there is a relation between the power of the lens P, the distance to the object d_O and the distance to the image point d_I:

$$\frac{1}{P} = \frac{1}{d_O} + \frac{1}{d_I} \tag{2}$$

You can see that the refraction of an object point depends on the power of the lens. The human eye can adapt its power to focus on the point of interest. Points further away or nearer to the lens than this point of interest are out of focus and will be blurred. The area to which a point gets blurred is called circle of confusion (CoC).

[Bertalmío et al. 2004] describes in his paper (related to the work of [Potmesil and Chakravarty 1981]) why there is a CoC. When there is an object point A, that is in the perfect focal distance, its reflection lines cross exactly at the the image plane. Another point's (B) reflection lines cross each other behind the image plane (or before if B is between the lens and A). The diameter of the CoC is given as the difference between the two lines at the image plane [$CoC(B')$ in Figure 4].

[Mulder and van Liere 2000] describes, based on the work of [Potmesil and Chakravarty 1981], how to calculate the size of the CoC. According to these papers the diameter of the CoC on the retina (see Figure 5) is

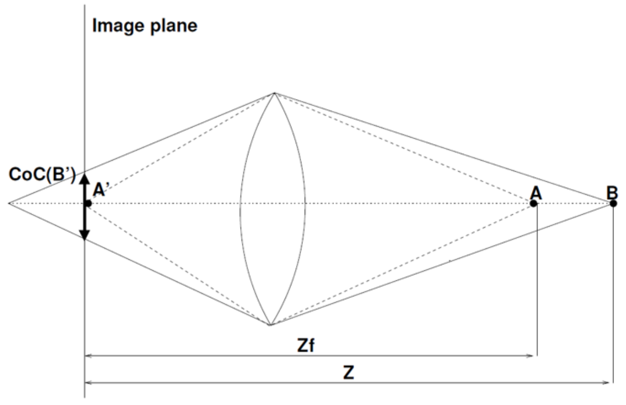

Figure 4: DOF model: A projects to A', B projects to a circle $CoC(B')$ (image from [Bertalmío et al. 2004])

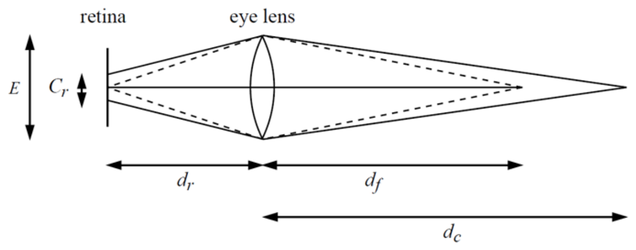

Figure 5: Calculation of the CoC (image from [Mulder and van Liere 2000])

$$C_r = |V_d - V_f|(E - V_d) \tag{3}$$

where

$$\begin{aligned} V_d &= \frac{Pd}{d-P} && d > P \\ V_f &= \frac{Pd_f}{d_f - P} && d_f > P \\ E &= lensdiameter \end{aligned}$$

in which

$$\begin{aligned} d &= \text{distance to unfocused object} \\ d_f &= \text{focus distance} \\ d_r &= \text{distance from lens to retina} \\ P &= \frac{1}{\frac{1}{d_f} + \frac{1}{d_r}} \end{aligned}$$

With this formulas the size of the CoC on the display screen can be calculated

$$C_s = \frac{d_s}{d_r} C_r \tag{4}$$

where d_s is the distance from the lens to the display screen

3.2 Approaches

Many authors like [Barsky and Kosloff 2008] categorize two major types of depth of field approaches. The first one are geometric approaches, in which the calculation is done in the 3D object space. The DoF calculation is done directly in the rendering pipeline. In general these methods give a better result with less artefacts, a reason for this is that this calculations are more similar to what happens in real lens systems. The problem with geometric approaches is that they are far away from calculation images in real time. The other category of depth of field algorithms are post processing approaches, which are done in image space. These methods are much faster and can be done in real time, which is necessary for using them in computer games. Some of these approaches can be calculated on the graphic hardware, using information that is available in the OpenGL or DirectX rendering pipeline.

3.3 Quality criteria for image space approaches

[Kosloff et al. 2008], [Randima 2004] and [Barsky and Kosloff 2008] identify in their works some goals for image based depth of field algorithms. These can be used to compare different algorithms against each other.

Per-Pixel Blur Level Control

Describes if it is possible to calculate the amount blur for each pixel independently of the other pixels. As objects can have complex shapes, the depth value of neighbour pixel can differ. Some approaches, as mentioned in [Barsky and Kosloff 2008], use pyramids or fourier transformations to calculate the blurring over the whole image. The perfect depth of field algorithm should have full control over the pixel blur level.

Choice of point spread function

We have described a simple camera model in section 3.1.1, but there are many different and more complex models. The point spread function (PSF) describes how light points get blurred by a specific lens system. Better DoF algorithms allow a width variety of PSFs.

Lack of intensity leakage

As described above the human eye builds a visual cue for all objects visible to it. One fact, that depth cue depends on, is that unfocused foreground objects will never blur over objects further away. Some of the simpler implementations like linear filtering suffer from intensity leakage artefacts.

Lack of depth discontinuity artefacts

Blurred foreground objects will always have soft edges in real world situations, linear filtering and some other algorithms have problems with these and calculate sharp edges even if the object is not in focus. This artefact is visualized in figure 7.

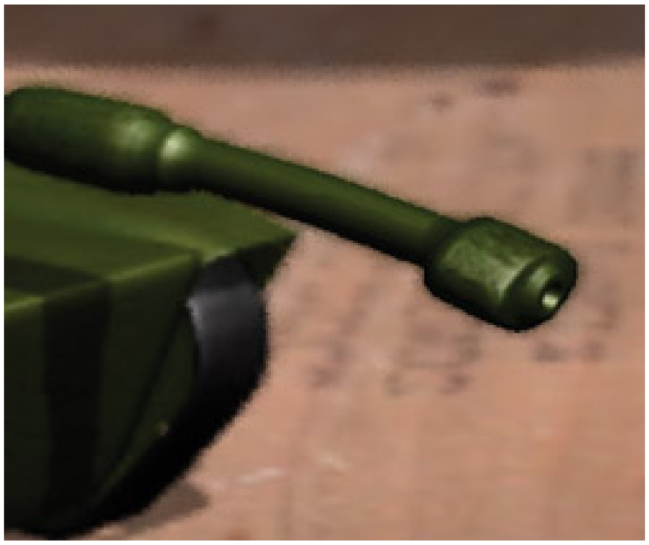

Figure 6: Intensity leakage artefacts, (image from [Randima 2004])

Proper simulation of partial occlusion

Partial occlusion means that unfocused foreground objects with blurred edges may show something of the background. The name partial occlusion comes from the fact that background objects are partially blocked by the foreground. You can see this in figure 8.

This effect is very hard to generate in post processing approaches because the algorithm has only the pinhole camera information at the beginning.

Performance

The time an algorithm needs to calculate a result image. Some methods take several minutes, but the faster ones can be done in interactive, or in real time.

These criteria should give a short overview of features that can be compared over approaches for depth of field calculation. There is no perfect solution that satisfies all of this goals now, this is still an open problem.

3.4 Linear Filtering

[Potmesil and Chakravarty 1981] describes in his work from 1981 the first depth of field algorithm using linear filtering. He does his calculation in spatial domain so the cost of this filter depends only on the size of the filter kernel used. In his work he describes a spatial variant linear filter using a depth dependent PSF. [Barsky and Kosloff 2008] explain this formulae in their paper:

$$B(x,y) = \sum_i \sum_j psf(x,y,i,j) S(i,j) \tag{5}$$

where B is the resulting image, (x, y) are coordinates in this image, (i, j) are coordinates in the input image (share image, S). Some of the results from Potmesil and Chakravarty's paper are shown in figure 9.

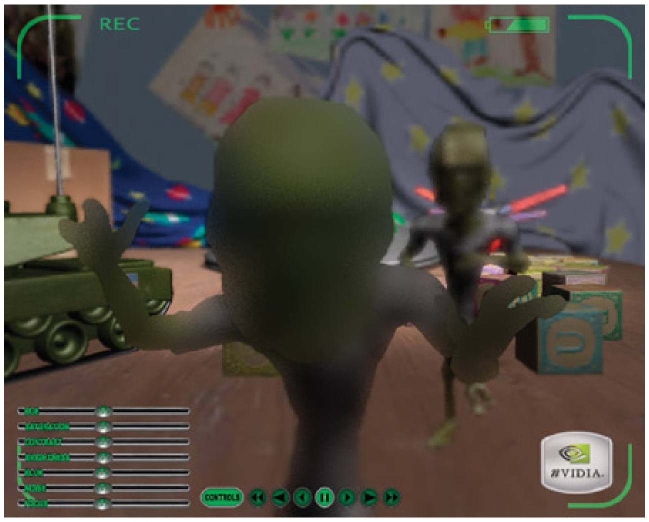

Figure 7: Depth discontinuity artefacts, (image from [Randima 2004])

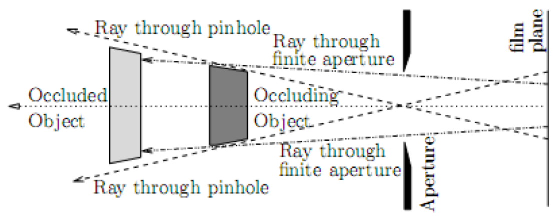

Figure 8: Partial Occlusion, (image from [Barsky et al. 2003])

This simple algorithm has some problems with depth discontinuity and intensity leakage.

3.5 Ray distribution buffer

In 1994 Shinay described a method that should replace linear filtering and handles visibility problems. (see [Shinya 1994]) This creates a ray distribution buffer (RDB) for each pixel, in which depth values are stored for each location from where light comes to this point on the image plane. This allows you to handle complex visibility problems by using the z-buffer. When the RDB creation is finished, the color for each pixel is averaged to get the blurred result. RDBs perform much better in the way of visibility problems, but they do this in cost of memory and time.

3.6 Layered Depth of Field

Some approaches try to separate the scene into different layers of depth, using one blur factor for each of them. [Scofield 1994] describes one of these methods and uses fast fourier transforms (FFT) to perform the filtering. Because of this, his methods efficiency is independent of the size of the PSF. After the blurring process the layers are combined using alpha blending. This technique eliminates intensity leakage completely and calculates a good approximation of partial occlusion. The biggest disadvantage is that only one blurring factor can be applied per layer, so depth changes within

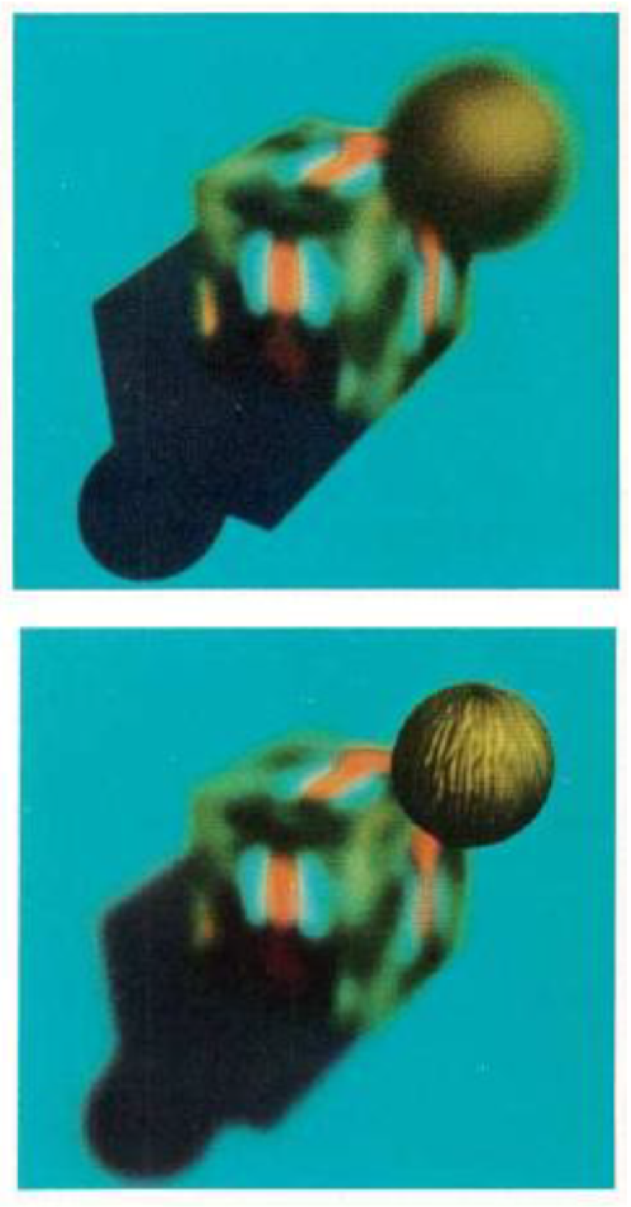

Figure 9: DoF using linear filter with different focus depths, (image from [Potmesil and Chakravarty 1981])

a layer will not be visible. Result of layered depth of field can be seen in figure 10.

3.7 Perceptual Hybrid Methods

Looking at computer generated images in games or virtual reality application, you will notice that the center of attention is (in general) near to the center of the image (see figure 11). [Mulder and van Liere 2000] uses this and filters the input image with two different techniques. The first one is a very fast algorithm, that is used for the peripheral viewing area, and is less accurate. The other one is a high quality method for the objects in the center of attention volume.

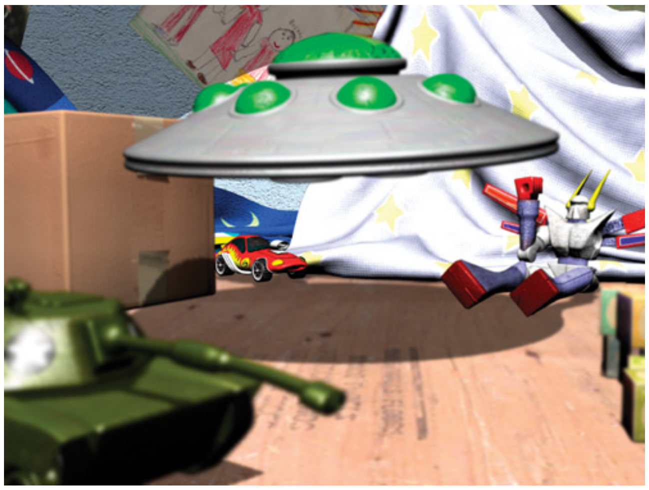

Figure 10: Layered DOF, (image from [Randima 2004])

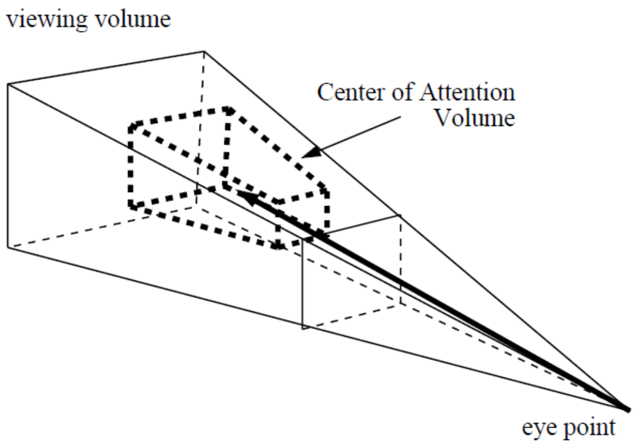

Figure 11: Center of attention volume, (image from [Mulder and van Liere 2000])

3.7.1 High quality algorithm

[Mulder and van Liere 2000] uses a number of discretized CoCs to perform a two pass rendering (you can see the discretized CoCs in figure 12). In the first path the CoC borders for each pixel are drawn starting by the largest on. Only the CoCs for those pixel were drawn that have a circle of confusion larger or equal to the current CoC. In the second pass the CoCs are rendered from the smallest to the largest ones. This time only the CoCs that contains that pixel are rendered. The advantage of this algorithm is that it can easily be implemented in graphic hardware by using a texture of the same size of the input image, containing the depth values in the alpha channel.

3.7.2 Low quality algorithm

For the fast algorithm, Gaussian pyramids (see [Burt 1981]) are used. Two pyramids are constructed, one only containing the pixels in front of the focus plane, the second one the pixels behind the focus plane. The resulting image is rendered by first drawing the levels of the front pyramid from front to back. Then the images on the focus plane are blended. After this the back pyramid is rendered

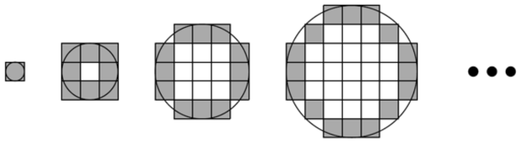

Figure 12: Discretized CoC sizes and their borders, (image from [Mulder and van Liere 2000])

from back to front. All this is done with depth testing enabled, so pixels will only be drawn when there is no other object in front of them.

The advantage of perceptual methods is, that they are faster than only using the high quality method, while users may not see a difference, because the low resolution part of the image will only be in his peripheral viewing area. One problem is, that it is possible that objects of interest are near to the image borders and not in the high quality area.

Results of [Mulder and van Liere 2000] can be seen in figure 13.

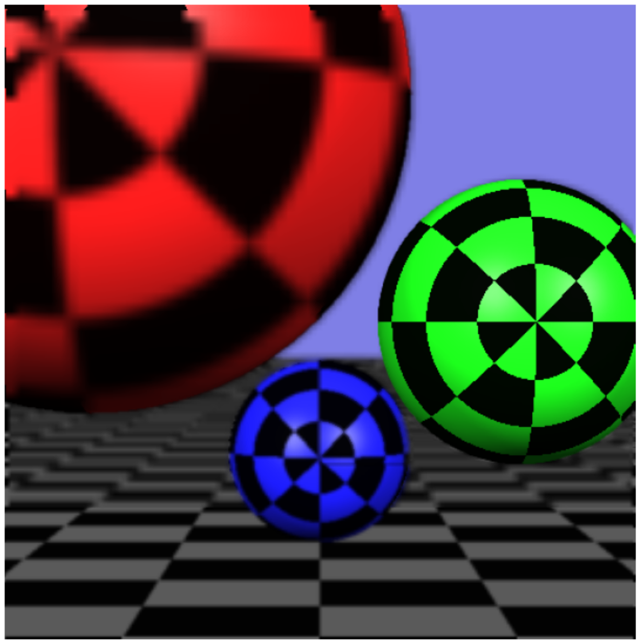

Figure 13: Perceptual DoF, (image from [Mulder and van Liere 2000])

3.8 Depth of Field on the GPU

In the last few years many methods, running on graphic hardware, where developed. One of the first methods was described by [Arce and Wloka 2002] but suffers from intensity leakage. Additional work on this was published by [Demers 2003] and [Riguer et al. 2003]. The problem with some of these methods is, that they need special capabilities of the graphic card or additional steps to create the information needed. In 2007, [Zhou et al. 2007] presented a possibility to calculate depth of field using only information that is a by-product of existing rendering technologies. One example for this is, that they avoid using a special render pass to calculate the depth information and make use of the z-buffer.

3.8.1 Use of non linear z-buffer

A common problem when working with data from a graphic cards z-buffer is the fact that the values are non linear and projection dependent. The simplest method to recover the original scenes depth information is to multiply z-buffer values with the inverted projection matrix. A rendered point p' can be described as the transformation of a point p to a canonical viewing space with the matrix P.

$$p' = p \times P \tag{6}$$

[Zhou et al. 2007] describes why it is not necessary to calculate the inverse matrix because only z information is needed. A perspective projection matrix P is in common given as

$$P = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix} \tag{7}$$

where r, l, b, t are the edges of the view volume and n, f are the near and far clipping plane.

Assuming that

$$p = x, y, z, 1 \tag{8}$$

and the transformed point in the canonical viewing volume

$$p' = \{\frac{x'}{w'}, \frac{y'}{w'}, \frac{z'}{w'}\} \tag{9}$$

we see corresponding to formulae 6 and 7 that

$$z' = P_{33} \times z + P_{34} \tag{10}$$

and

$$w' = -z \tag{11}$$

The resulting depth value z is then given as

$$z = \frac{-P_{34}}{\frac{z'}{w'} + P_{33}} \tag{12}$$

As P_{34} and P_{34} only refer to n and f, the clipping planes, the values can be precomputed when the projection is defined. There are also possibilities to use the vector arithmetic of the graphic card to compute four points parallel.

3.8.2 Zhou, Chen and Pullens algorithm

This algorithm is designed as a two pass filter, filtering in the first pass in horizontal direction and in the second pass in vertical direction, using the result of the first pass as input. After the second pass the results of the two passes get blended and normalized to create the output image. This approach is similar to the one used for separable Gaussian filters. To control the amount of blur they developed an adaptive filter kernel, that dynamically adjusts the CoC size at every pixel, using three factors to specify the weight applied to each pixel in the filter.

The first factor is the light intensity function, that is approximated as described by [Chen 1987] in order to make it applicable in the pixel shader. This formulae describes a light intensity falloff proportional to the reciprocal of the squared radius of the CoC.

$$I(r_p) = \frac{1}{r_p^2} \tag{13}$$

The second factor is the intensity leakage control function $L(z_p)$, that should help to prevent intensity leakage artefacts as they occur in most post processing filter methods. [Zhou et al. 2007] describes the use of this function as follows: "If the sampled pixel P, is farther from the camera than the focal plane, its weight is adjusted by a factor based on the size of the circle of confusion of the center pixel (up to a maximum value of 1), otherwise its weight is unchanged.".

$$L(z_P) = \begin{cases} \alpha r_c, & z_P > z_f \\ 1, & z_P \geq z_f \end{cases} \tag{14}$$

in which

$$\begin{aligned} r_c &= \text{radius of the circle of confusion, measured in pixel widths} \\ z_P &= \text{scene depth of the sampled pixel} \\ z_f &= \text{scene depth of the focal plane} \end{aligned}$$

This formulae reduces the contribution to the amount of blur of a foreground pixel in a way that it will be zero if the pixel is in focus. When the focal plane is behind the pixel it stays one.

The third factor is the overlap function ($O(r_p)$), that describes how much of pixels area is overlapped by the CoC. This can be: not overlapped, which leads to no contribution, partial overlapped (contribution factor in the range from zero to one) and totally overlapped with a factor of one. The three cases are shown in figure 14.

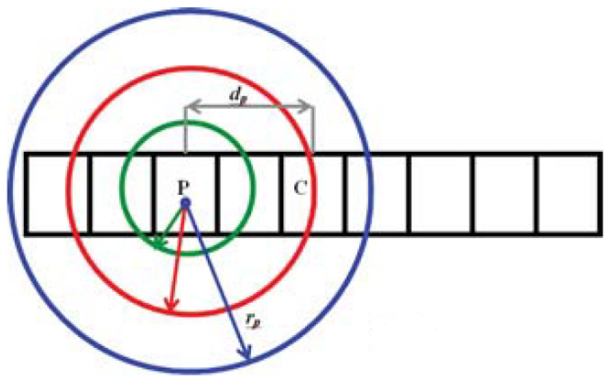

Figure 14: circle of confusion of pixel P, (image from [Zhou et al. 2007])

$$O(r_P) = \begin{cases} 0, & r_P \leq d_P \\ r_P - d_P, & d_P \leq r_P \leq d_{P+1} \\ 1, & r_P \geq d_{P+1} \end{cases} \tag{15}$$

where

$$\begin{aligned} r_P &= \text{radius of the circle of confusion of pixel P} \\ d_P &= \text{distance of pixel P to center pixel C} \\ d_{P+1} &= \text{distance of the next pixel outside the kernel} \end{aligned}$$

The final weight for a pixel is the product of these three factors

$$W(P) = I(r_P) \times L(z_P) \times O(r_P) \tag{16}$$

For the center pixel only the light intensity function is used

$$W(C) = I(r_C) \tag{17}$$

The big advantage of the [Zhou et al. 2007] method is, that the design as a two pass filter approach allows it to use much more sample points than it would be possible with simple filtering. Results of the algorithm can be seen in figure 15 and 16.

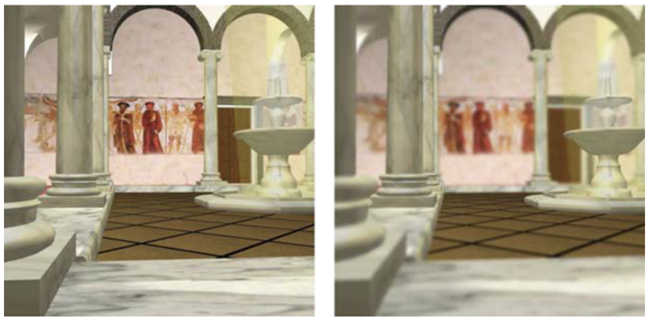

Figure 15: Cloisters: Left image, no depth of field. Right image, depth of field with focus on the fountain., (image from [Zhou et al. 2007])

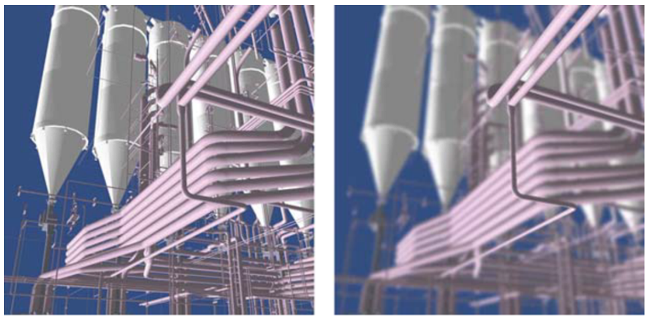

Figure 16: Factory: Left image, no depth of field. Right image, depth of field, with focus on foreground pipes., (image from [Zhou et al. 2007])

3.9 Summed Area Tables

Summed area tables can, in the context of DoF, be used as an alternative for the sampling of the CoC. In addition to this it is not necessary to calculate a down sampled image or to distribute point samples. Summed area tables were first described by [Crow 1984] for texture mapping problems. [Hensley et al. 2005] shows how they can be calculated fast enough for other computer graphic issues like depth of field. The advantage of SAT is that the creation of them takes a fixed time, independent of the size of the filter. The SAT creation can also be fastened up by using a doubling technique on the GPU. In general depth of field methods using summed area tables have problems with intensity leakage and depth discontinuity.

3.10 Pyramidal Methods

Pyramids, as described in section 3.7.2, can be used to simulate the blurring effect, which will always occur when objects are out of focus. [Mulder and van Liere 2000] used Gaussian pyramids only in the regions that are not important to the user, because they used a very simple approach and got block artefacts in their images.

That depth of field can be performed for regions in the center of attention, is shown by [Kraus and Strengert 2007]. They developed a layered method, but, in difference to the one in 3.6, a point is not mapped to the nearest plane. In order to avoid depth discontinuity artefacts on layer edges, it distributes a point to several adjacent layers. In the next step the layers get blurred by first down sampling it and then sampling it up. In this step it is necessary to use a weight function to avoid block artefacts as they would occur with simpler pyramid types. To create the result image the layers get composed using alpha blending.

This algorithm avoids depth discontinuity and has the possibility to control intensity leakage and is in general much faster than other layered methods. The problem with pyramidal approaches is the limitation of the PSF to the ones that can be applied to pyramids. A result from [Kraus and Strengert 2007] can be seen in figure 17.

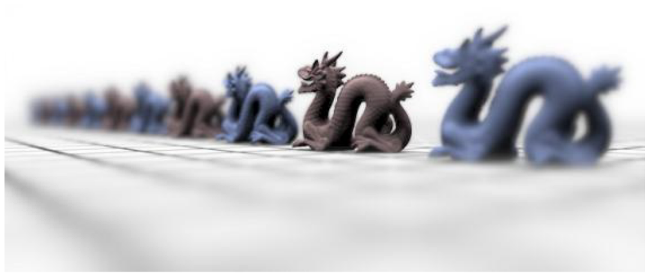

Figure 17: Factory: Left image, no depth of field. Right image, depth of field, with focus on foreground pipes., (image from [Kraus and Strengert 2007])

3.11 Approaches based on head diffusion

Working with Partial Differential Equations (PDE), as the heat diffuse is one, has a long history in computer graphics. Since the work of [Koenderink 1981] and [Witkin 1987] in the early eighties, many different papers where published like the work of [Perona and Malik 1990], in which they describe a method to replace Gaussian smoothing by a head diffusion.

$$\frac{\partial I}{\partial t}(x,y,t) = \triangle I(x,y,t) = \nabla \cdot (\nabla I(x,y,t)) \tag{18}$$

The heat equation (given in formula 18) is a second order PDE, which can be given in numerical form for the n-th step

$$I^{n+1}(i,j) = I^n(i,j) + \frac{h^2}{8} \triangle I^n(i,j) \tag{19}$$

where $\frac{h^2}{8}$ is the time step (It can be shown, that the greatest value for $\frac{h^2}{8}$ that ensures stability, is 0.25, so this would be a good choice).

Applying one step of the heat diffusion equals to a convolution of the image with a Gaussian filter kernel of the width h

$$I^n = I^{n-1} \star G_h \tag{20}$$

According to this running N steps equals a convolution with filter width of$\sqrt{2Nh}$

$$I^n = I^0 \star G_{\sqrt{2Nh}} \tag{21}$$

Assuming that the step width is 0.25, we get a total number of steps N, that must be performed to get an equal result to the convolution of the image I with a Gaussian filter with a width of σ:

$$N = \frac{\sigma^2}{2\sqrt{2}} \tag{22}$$

[Bertalmío et al. 2004] shows, that a more generalized version of the head diffusion equation 18 can be used to simulate depth of field:

$$\frac{\partial I}{\partial t}(x,y,t) = g\nabla(g(x,y,t)\nabla I(x,y,t)) \tag{23}$$

In equation 23, g has to control how the depth of the corresponding pixel influences the diffusion process. To understand how this is possible, we need to take a closer look on how the diameter of the CoC, given in equation 3, can be represented. Formula 24 gives us an alternative version of the former one:

$$c = \alpha \frac{|Z - Z_f|}{Z} \tag{24}$$

where Z_f is the focal distance and Z is the distance to the object. α is a constant value that depends on the focal length of the lens (F) and a aperture number (n).

$$\alpha = \frac{F^2}{n \cdot (Z_f - F)} \tag{25}$$

According to this g can be formulated as

$$g(x,y) = \alpha \cdot \frac{|Z(x,y) - Z_f|}{Z(x,y} \tag{26}$$

The result of the diffusion process in 23 and 26 gives the same output image as a convolution with a Gaussian filter kernel of the Center of Confusions width. As described by [Bertalmío et al. 2004] (linking to the work of [Ladyzhenskaya 1985]), it can be shown, that these equations are well formed, so there will always be a unique solution.

The use of the heat equation for DoF has one major benefit: It does not cause intensity leakage artefacts. This is caused by the fact that it uses the gradient of I (∇I) which is not the same as g multiplied with the Laplacian $\triangle I$.

$$\frac{\partial I}{\partial t} = \nabla \cdot (g\nabla I) = g\triangle I + \nabla \cdot g\nabla I \tag{27}$$

The second part ($\nabla \cdot g\nabla I$) prevents the intensity leakage. For a more detailed explanation of this see [Bertalmío et al. 2004]. Image 18 shows a comparison of the results from formula 23, with a version without the leakage prevention term.

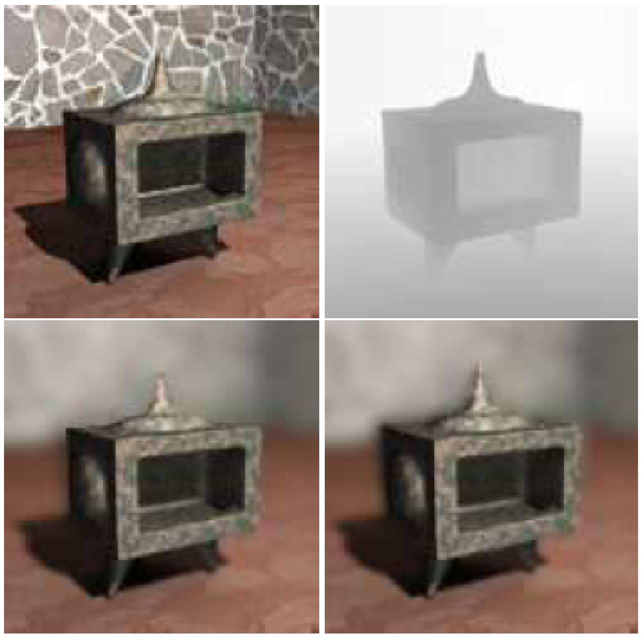

Figure 18: Top: original image (left) and Z-buffer (right.) Bottom: Heat diffusion with (left) and without(right) leakage prevention, (image from [Bertalmío et al. 2004])

4 Motion Blur

Another image artefact, that is caused by the mechanical apparatus of real world cameras, is motion blur. This effect is the best way in computer games to simulate speed and can be, especially for racing games or flight simulators, the most important effect because images with motion blur look much more realistic. [Nguyen 2007]

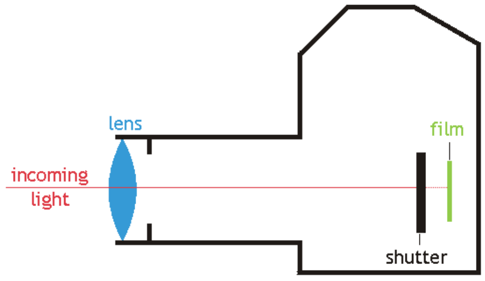

Figure 19: Camera apparatus including lens and shutter

When an image is taken by a camera, the shutter opens for a short time so light can enter the camera and reach the film, that accumulates the incoming light. This is shown in figure 19. The longer the shutter is opened the brighter the image will be. The time the shutter stays opened (according to [Potmesil and Chakravarty 1983] this time is called $exposuretime$), causes motion blur, which is the effect that moving objects in a scene get blurred in the direction of their movement. There are two possibilities why this happens:

- The most common reason for blurred images is, that moving objects change their position in the exposure time, causing the film (or sensor in digital cameras) to accumulate the objects reflection on slightly different positions.
- The other fact is that the shutter takes a short time to change its position from totally closed to opened (and vice versa). This effect is in general not visible to the human eye and we will not talk about this later on.

In computer graphic cameras this effect will not appear due to the fact that they will always render on discrete position in time, what would be the same as an infinite short opening time of the cameras shutter. There are several approaches to create motion blur on rendered images that are used by many modern games (see image 20). Another similar area of research is the removing of motion blur artefacts from photographs, witch can be seen as the inverse problem to motion blur rendering. So many methods and algorithms can be used for both.

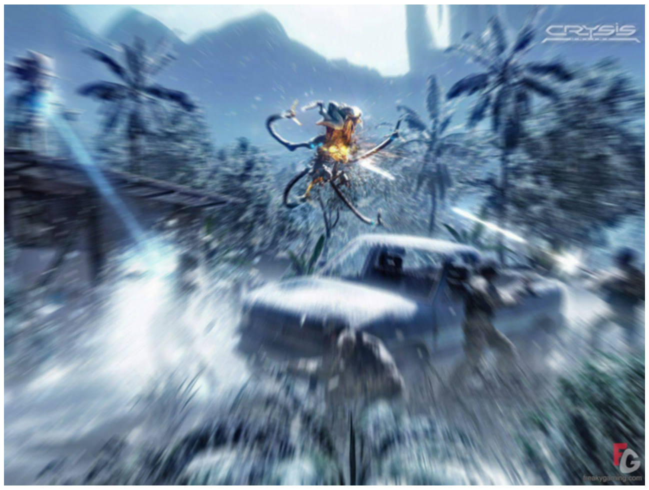

Figure 20: Motion blur in Crysis

4.1 Approaches

Early approaches of [Potmesil and Chakravarty 1983] describe methods of image processing, like transformations to frequency domain, multiplying with a uniform blurring function and than perform an inverse Fourier transformation. This was a good approach when motion in every part of the image is equal, but in modern game scenarios this assumption can not be given.

4.2 Max and Lerner: 2 and a half D method

In 1985 an extension to [Potmesil and Chakravarty 1983] method was described by [Max and Lerner 1985], where the whole scene gets split-up in layers, where each layer includes color and opacity information. In addition to this the motion on every layer has to be the same. All of these layers get blurred by a very fast raster blurring algorithm. Blurring an image is very easy along the axis (in x or y direction) so this algorithm skews the original image in a way that blurring is only needed along the largest blur vector component. The whole blurring is constructed as a three pass post processing algorithm. The three steps are:

Skewing

As described above the original image raster has to be skewed by a transform function that shifts images to the needed direction. An example for this, that is given in there paper, is: "Suppose a raster is to be blurred twice as much in the x-direction as in the y-direction. Then the raster is skewed by shifting downwards by

$$\triangle y = x \cdot \frac{dy}{dx} \tag{28}$$

, which is x^2 in our example" [Max and Lerner 1985]. In general the shifting factor, $\triangle y$ in the example, will not be an integer number, so it is needed to calculate a weighted sum from the two nearby pixels for every shifted one. The result of the skewing described in the example is shown in image 21 (top right).

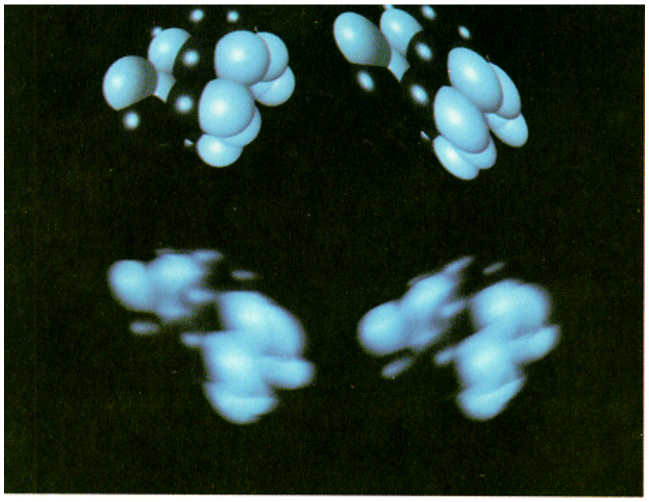

Figure 21: Raster Blur Algorithm
top-left: original image
top-right: skewed raster
bottom-left: blurred, skewed raster
bottom-right: unskewed raster
(image from [Max and Lerner 1985])

Blurring

This step has to be performed along the largest blur vectors component (and because of this always along the other axis than the skewing step). As an assumption the blurring distance dx (or dy if $dy > dx$, from now on we will not mention the other direction every time it is possible) should be smaller than the size of the skewed raster.

Simply blurring the image in the given direction would be the easiest option, but this would only clarify along which axis a object is moving, but not in which direction. This is because a constant coloured object would be of constant intensity in the center and decrease its intensity along the axis of the movement. The brightness (B) of a pixel x, in this unweighted blurring, is calculated as the sum of the dx last pixels in this row of the raster S.

$$B(x) = \sum k = 0^d x S(x-k) \tag{29}$$

In practice this summation can be done very fast by computing it incrementally

$$B(x) = B(x-1) + S(x) - S(x - dx - 1) \tag{30}$$

As mentioned above blurring with this method does not indicate the direction of movement. So a function is needed that produces long fading streaks and indicates the direction. This can be done by adding a weighting factor to formula 29.

$$B'(x) = \sum k = 0^d x (dx-k) S(x-k) \tag{31}$$

$(dx-k)$ is a linear weighting factor over the time, that is zero for the oldest position and one for the actual position. To conserve the overall intensity of the image, a weighting factor N can be used. B' can also be calculated incrementally:

$$B'(x) = B'(x-1) + (dx+1)S(x) - B(x) \tag{32}$$

The composition of the result image of this blurring step can be calculated using formulae 33, where a and b are free variables for the weighting. Energy conservation is given as long as $a+b=1$. It is possible to adjust them in a way that $a+b>1$ for fast moving objects because they might get too dark. Some combinations of a and b are shown in image 22, in the top left photo, constant sum blur ($a = 1$, $b = 0$) is applied. On the top right, linear sum blur is shown ($a = 0$, $b = 1$), in the bottom left image $a = b = 0.5$ and in the bottom right image $a = 0.1$ and $b = 0.9$. The last case is also shown in figure 21 in the bottom line.

$$B''(x) = a \cdot M \cdot B(x) + b \cdot N \cdot B'(x) \tag{33}$$

Unskewing and drawing

After blurring the image, the inverse transformation, to the one described in 4.2 in the Skewing paragraph, is applied to the skewed raster to create an unskewed raster ($I(x)$). The resulting raster is then drawn to the screen, combining it with the previously drawn by using the opacity mask. For this algorithm it is necessary that the layers get drawn from back to front.

4.3 Motion Blur for Stop Motion Animation

There are two big problems with the algorithm described in 4.2. At first there is the problem of separating the whole scene in layers with constant motion, that can be very time consuming. The second problem, related to the first one, is that motion for every point on the raster has to be known.

One of the first works, dealing with motion blur on stop motion animations, were developed by [Brostow and Essa 2001]. Stop motion animation describes the technique where still images are taken of a

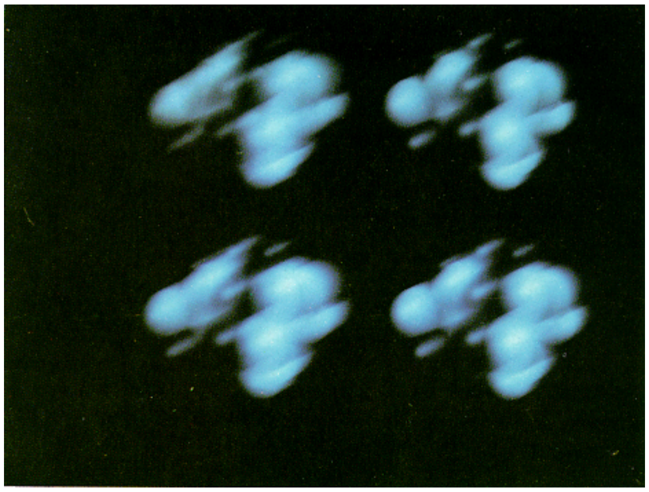

Figure 22: Four different blur weights
top-left: constant sum blur
top-right: linear sum blur
bottom-left: $a = b = 0.5$
bottom-right: $a = 0.1, b = 0.9$
(image from [Max and Lerner 1985])

scene at discrete time steps and than playing them with a frame rate high enough to simulate continuous motion.

[Brostow and Essa 2001] calculate the motion of a point by using a blob motion detection algorithm. For this, they start by using a frame differencing technique to identify points that are in motion. First they calculate a median image (I_b) off all subsequent frames by calculating the median intensity value for each pixel. This image describes the static background of the scene. The next step is to calculate an image that only contains points that are different from the background in a frame (I_f). After this a threshold is applied to the image in a way that:

$$I_m(x,y) = \begin{cases} I_f(x,y) & |I_f(x,y) - I_b(x,y)| > threshold \\ 0 & otherwise \end{cases} \tag{34}$$

The blob generation is than performed on the threshold image (I_m) by grouping continuous regions. The motion calculation used by [Brostow and Essa 2001] is based on the work of [Bergen et al. 1990] and [Tweed and Calway 2000], using the previously generated information. The result is a vector map V_i, that is visualized in figure 23 describing the motion of each point in I_m. It is possible to improve the results by performing a flow correction algorithm on this result, similar to [Black and Anandan 1996].

Rendering the blur

Now the way each pixel moves from frame t_i to t_{i+1} can be given by a motion function $L_i(x,y,t)$, describing the interpolated offset of a pixel (x,y) at moment t. For calculating the color of a pixel there are two other parameters needed: τ, the time between two frames in seconds and s, the exposure time. To simulate this exposure time it is needed to integrate L_i for a total time of s, half before and half after t_i. The image describing the result from t_i to $t_i + \frac{s}{2}$ is called I_{Aft}. Each pixel in this image can independently be created by the

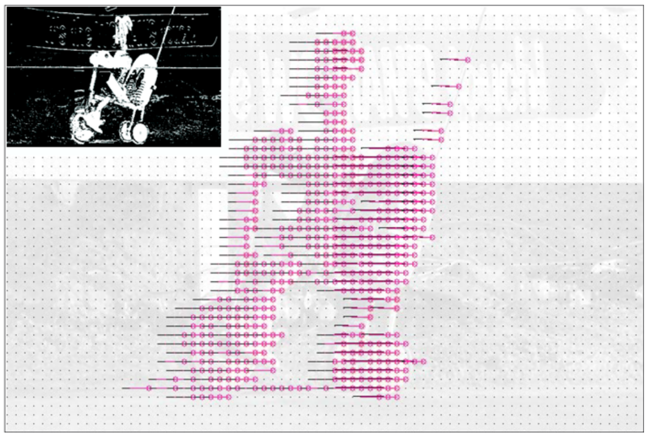

Figure 23: Vector map V_i for a frame from the film chicken run (image from [Brostow and Essa 2001])

integral over all pixels lying in the line of L_i. The fraction of a path over a pixel with borders a and b can be expressed as following:

$$w(x,y) = \frac{\int_a^b (\sqrt{(\frac{dx}{dt})^2 + (\frac{dy}{dt})^2})dt}{s/2} \tag{35}$$

For better understanding the movement function L_i mapped to a pixel grid is shown in figure 24:

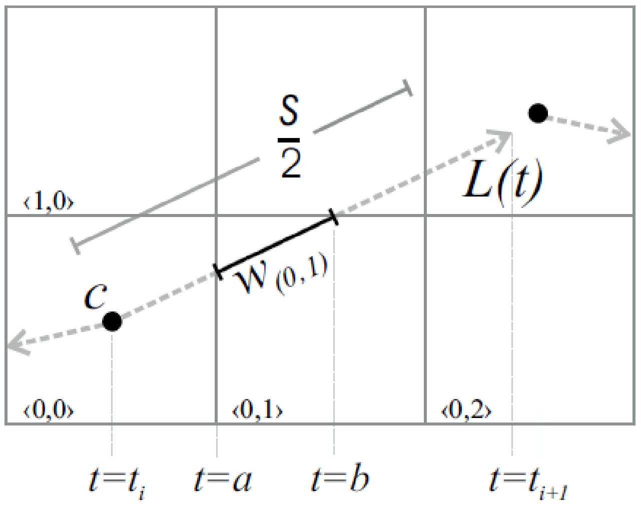

Figure 24: Color pixel c moved according to the dotted path L. After choosing a desired shutter speed s, cs RGB values are redistributed according to the time it spent moving through each pixel. (image from [Brostow and Essa 2001])

It is also possible to weight the pixels in a non linear way, for example by using higher weights at the end or beginning of the motion vector. The resulting color for a destination pixel (x,y) is then calculated as the color c of this pixel multiplied with the weight w $(c \cdot w(x,y))$. The other half shutter time image before t_i is calculated similar to I_{Aft} and will be referred to as I_{Bef}.

The total result of this blurring algorithm can then be calculated as the average of I_{Aft} and I_{Bef}, because each of these two images describe the movement for half of the exposure time. Results of this method can be seen in image 25.

Some authors describe a very similar method to the algorithm from [Brostow and Essa 2001] by assuming that the total time of s is before t_i and only calculating I_{Bef}. This can be very useful in computer game environments where the image of the next frame I_{t+1} is not known at the render time t_i.

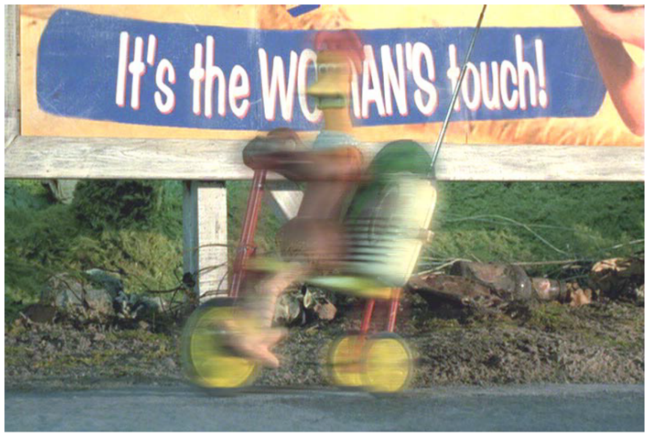

Figure 25: Result from [Brostow and Essa 2001] Frame from chicken run (Rocky rolling backwards on the tricycle)

4.4 Accumulation Buffer

The previously described methods all try to integrate over the reflected light while the shutter is opened. Some of them solve this integral explicit, others use it implicit to blur along the moving direction. The use of graphic hardware to solve this integral is not easy to handle, as numerical integration may not be fast enough to perform in real time applications. One of the first hardware supported approaches of motion blur was described by [Haeberli and Akeley 1990] in his work on accumulation buffers.

The accumulation buffer is a special hardware or software buffer which calculates the average of multiple images by summing the images and then dividing them through the number of images. In modern graphic cards (since 2002, ATI Radeon 9700) the accumulation buffer is always implemented in hardware.

Motion blur with accumulation buffer can be performed by accumulating for the frame F_t over the last N frames, that where rendered before t. Higher quality can be achieved by calculating subframes between the presented frames, what is needed for fast moving objects to prevent from ghosting artefacts. Ghosting is when a moving object moves too much between frame F_t and F_{t-1}, so that there are holes in the image. Ghosting can be seen in figure 26.

The quality of final results of accumulation buffer algorithms depend mostly on the number of subframes calculated between two frames, because rotational motion is then better visible. Result images from [Haeberli and Akeley 1990] can be found in image 27.

4.5 Velocity Maps with Multiple Render Targets

Rendering many frames per second to accumulate them may be a problem for modern games. Another very fast and hardware supported algorithm is the generation of motion vector maps as a by-

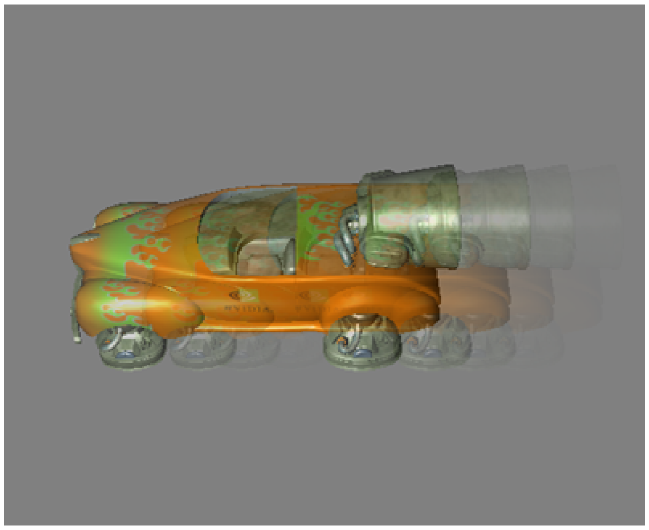

Figure 26: Ghosting or temporal aliasing effect when the motion blur of a fast moving object is generated using accumulation buffer but large time interval between multiple renderings. (image and caption from [Shimizu et al. 2003])

product of the rendering pipeline, using multiple render targets. One of these render targets will than be the vector map. Motion vectors from a frame to the next can easily be calculated with the matrices of the previous and current frame, calculating the distance between the points transformed with them. [Nguyen 2007]

The final motion blur is then performed by sampling points for each pixel along the corresponding motion vector and accumulating them.

4.6 Offset Warping

Another method that uses multiple render targets is the algorithm described by [Shimizu et al. 2003]. This algorithm is a two step approach, where the second step is an iteration method which performs better with each iteration. The approach is visualized in figure 28.

In the first step a per vertex displacement vector field is calculated that defines how much warping will be performed on this vertex in a three dimensional vector. The offset vector depends on the normal of the vertex, the direction of movement and the movement speed and can be calculated using formulae 36:

$$W = (n \cdot v)V \tag{36}$$

where n is the normal, v is the normalized motion vector and V is the real motion vector.

The second step performs the per vertex warping, starting with the maximum warping factor and moving towards the final objects position. This resulting warped image is then alpha blended with the input image. This is performed one time in the vector fields direction and one time in the opposite direction. The warping in the opposite direction is done with a lower scaling factor because trailing edges should be blurred more than leading edges. The alpha value for blending is inverse to the warping factor, which allows warping, that is nearer to the original position, is more visible in

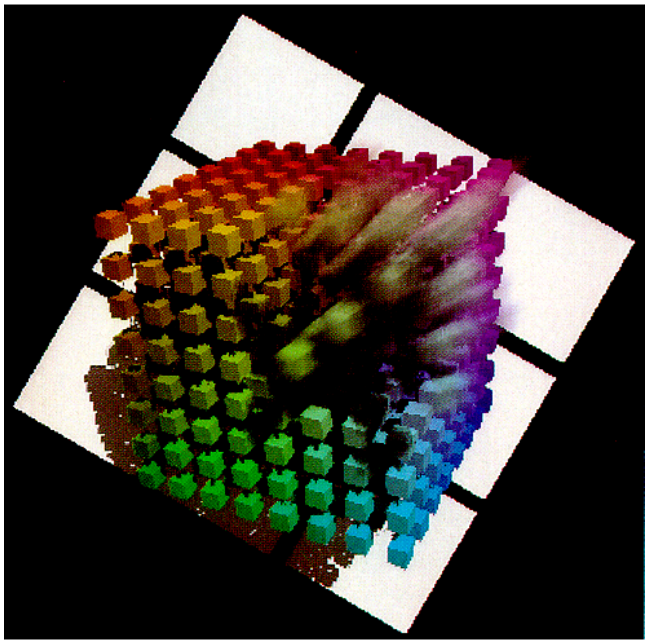

Figure 27: Motion blur as an accumulation of 23 frames (image and caption from [Haeberli and Akeley 1990])

the final image. This iterative step can also be seen as a line integral convolution along the motion.

Image 29 shows the result of this algorithm

4.7 Velocity Maps from Depth Buffer

The method in 4.5 is a very fast and good method for velocity map creation, but there are some drawbacks. The first is that it is very hard to integrate an additional render target in an existing game engine, because all shaders need to be rewritten in order to output velocity information. The second problem has to do with the limited render target space on game consoles, which may not allow to maintain an additional render target.

These facts leed to a, a little bit more calculation intensive but more storage friendly, method described in [Nguyen 2007]. A point in screen space is given by its x and y component in the frame buffer and its z coordinate in the z-buffer. These three values can be transformed back to scene space, using the inverse projection matrix and the methods for non linear z-buffers described in section 3.8.1. The world space position now allows to calculate the motion vector by transforming it with the current and the previous projection matrix.

4.8 Split/Second Motion Blur

A method to improve performance for motion blur in video games is given by [Ritchie et al. 2010]. He describes an algorithm that uses motion coherence in a scene as it may be given in many games, especially in racing games. As the motion is constant in big parts of the scene, all objects that have the same movement get a shared motion vector ID, using 4 bits, there can be 16 different of them. To get the image motion vectors, the scene motion vector corresponding to a motion vector ID can be inverse projected.

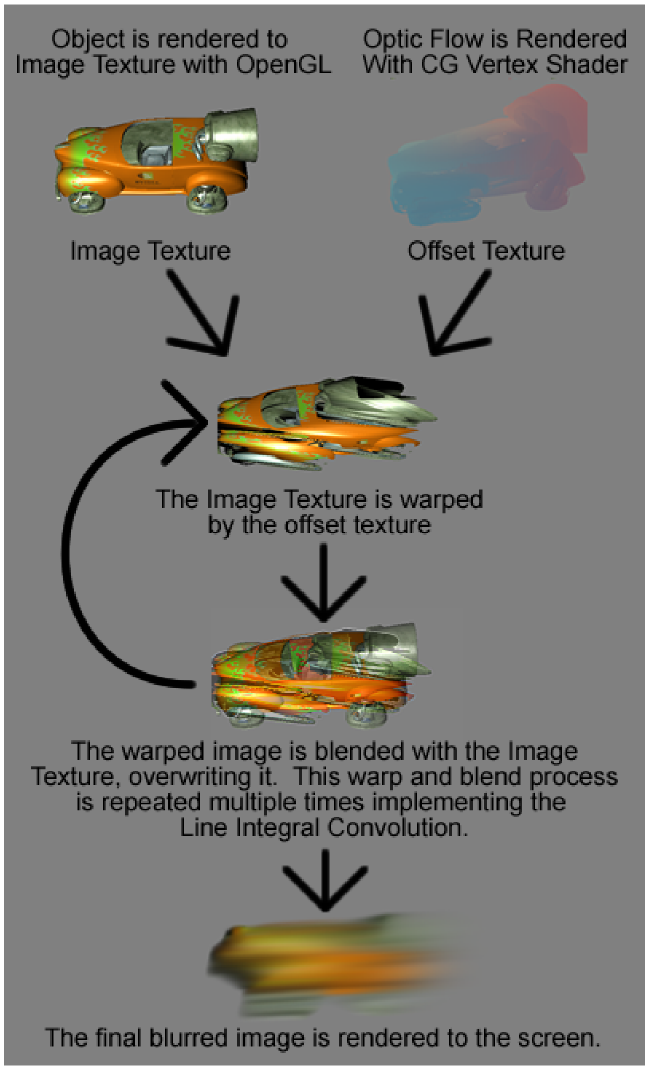
Object is rendered to Image Texture with OpenGL
Optic Flow is Rendered With CG Vertex Shader
Image Texture
Offset Texture
The Image Texture is warped by the offset texture
The warped image is blended with the Image Texture, overwriting it. This warp and blend process is repeated multiple times implementing the Line Integral Convolution.
The final blurred image is rendered to the screen.

Figure 28: Overview of the algorithm (image from [Shimizu et al. 2003])

[Ritchie et al. 2010] advices also to use a fixed number of shaders with different filter kernel width for the smoothing process. Results of the motion vector generation can then be used to determine which shader has to be used.

To achieve better results, in fact the above described method will not look very good, the use of a texture space motion blur as described by [Bala and Dutr 2010] will be needed.

This algorithm is used in the game Split/Second developed by the Disney Interactive Studios and released in 2010.The big advantage of this method is that it combines image space motion blur and texture space motion blur with a very low number of different motion vectors, which allows to use few samples while producing an optical good result. Screen shots from Split/Second can be found in image 30.

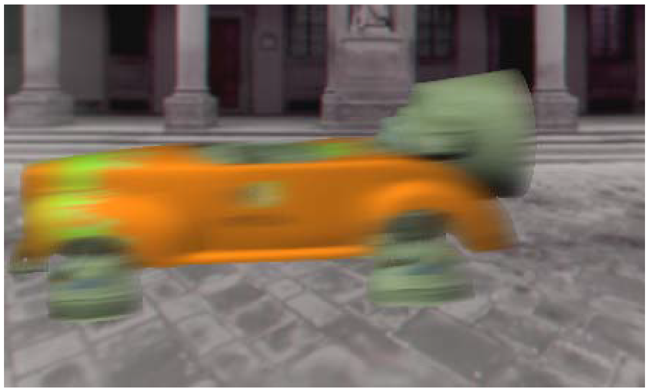

Figure 29: A blurred car in front of a static background (image from [Shimizu et al. 2003])

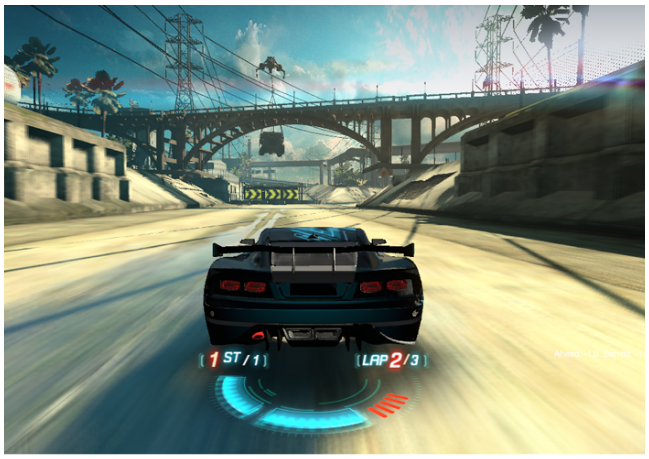

Figure 30: Motion blur system in Split/Second from top to bottom; image space adaptive at high speed, augmented with texture space blur (image and caption from [Ritchie et al. 2010])

5 Glow and Bloom

The previously described effects, depth of field and motion blur, solve problems that come from the limitation of the camera model in real time graphics. Another one of these effects are Glow and Bloom.

Glows or halos in the real world come from the fact, that light is scattered in the atmosphere or in the human eye. When playing computer games or watching films there is a limited amount of light, that can be transported to the humans eye. The glow effects calculate halos around bright objects, fooling the human eye in a way that even objects with a very small halo look much brighter than the same object without halo. In image 31 the same scene is rendered without glow (on the top) and with glow (bottom).

5.1 Tron 2.0

[Nguyen 2007] presents a glow algorithm in his work, written for the computer game Tron 2.0, which is designed to produce large glow areas over the whole screen and allowing the artists easily to define the glow factor of their game assets. This technique is fast

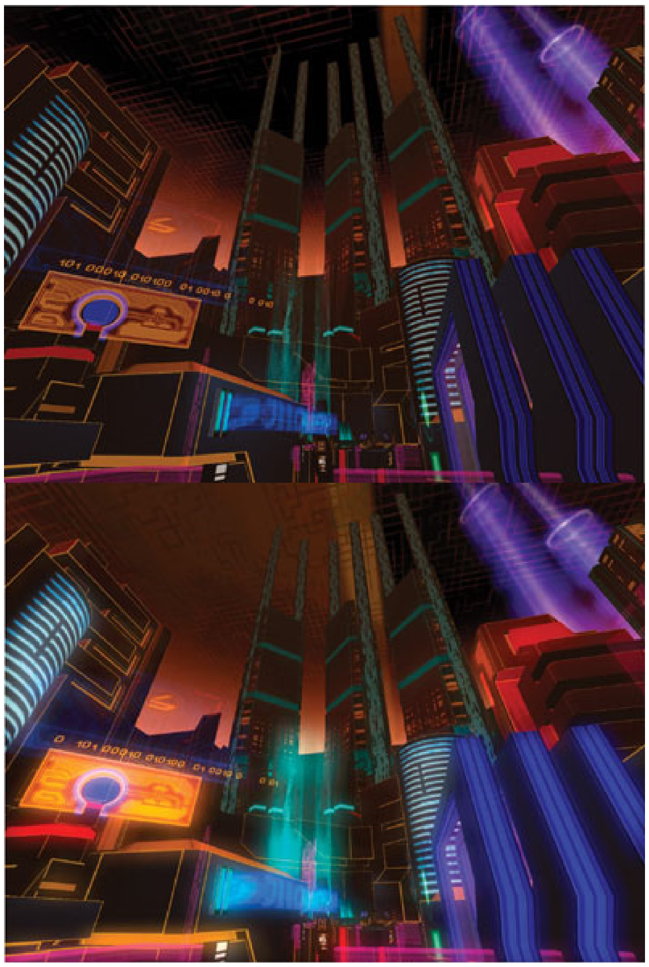

Figure 31: Same scene with (bottom) and without glow(top). (image from [Nguyen 2007])

enough to be performed for shooter games with a minimum of 60 frames per second.

There are four steps that have to be performed in order to render the glow effects, that are shown in figure 32. First the scene is rendered to a texture (a). Then the glow source texture is created by rendering the same scene, but outputting the RGB color from step one multiplied with the glow factor of the object or object part(b). This can also be combined with the first step using multiple render targets. Then the glow source texture gets blurred by using the separable Gaussian filter described in section 2.2 (c). It is also possible to perform this step at a lower image size, as applying a small filter kernel to a small image may look very similar to applying a large filter kernel to a large image. The down sampling can be done using a lower MipMap level. The final result is then rendered by combining the original scene image from step one and blending it with the blurred glow source texture, using additive alpha blending (d).

Glow factor

As described above, a factor, defining how glowing an object is, is needed to create the glow source texture. This can be done by defining a glow factor for each object. A better approach also used in Tron 2.0 is the usage of a glow factor texture, in order to allow different parts of an object to have different glow. [Nguyen 2007]

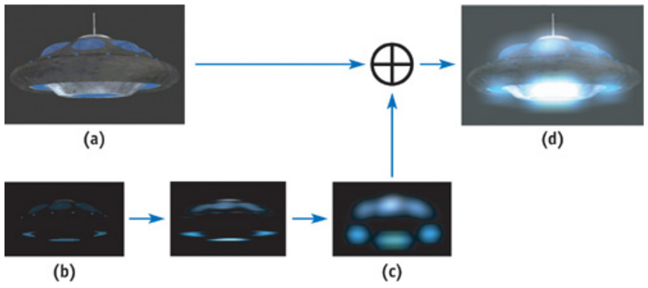

Figure 32: Rendering steps for glow effects. (image from [Nguyen 2007])

also describes, that it would be good to store the glow information in the alpha channel of the rgb texture if not needed, because the alpha channel can be read with the same texture lookup as the color value. Figure 33 shows such a texture composition and the resulting glow source texture.

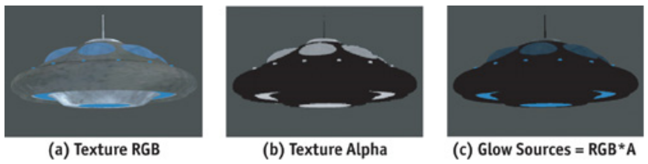

Figure 33: Glow factor in the color textures alpha channel. (image from [Nguyen 2007])

Results from the computer game Tron 2.0 can be seen in image 34.

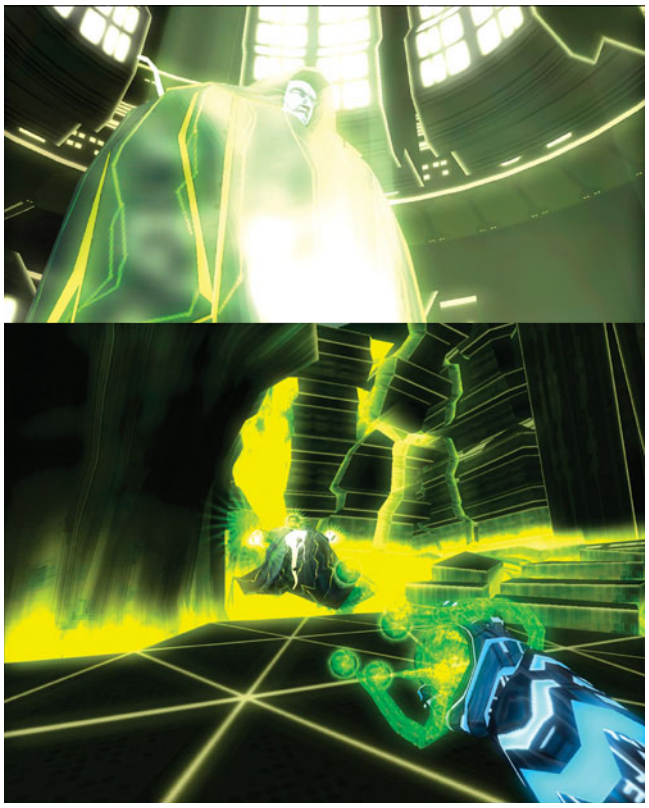

Figure 34: Glow effect in the computer game Tron 2.0. (image from [Nguyen 2007])

5.2 Bloom

Bloom is an effect that's implementation is very similar to the one of glow. The bloom effect lets bright areas bleed into darker areas, producing also the feeling of a higher brightness. The only difference to the glow effect is the creation of the glow source texture.

The glow source texture is created by rendering the original scene image (a) applying a threshold value, in order to render only points that are brighter than the threshold.

In the result image (figure 35), the bloom effect can be seen very good at the edges of the front man's head.

6 Toon/Cel Shading

In the width area of post processed special effects there are in general two categories, the photo realistic methods and the non photo realistic ones. Photo realistic rendering tries to produce images that are as similar as possible to a real photograph. All previously described effects, as motion blur or depth of field, are from these type, trying to cope with the limitations of real time rendering camera models.

The other class, non photo realistic rendering, comes from a different point of view. These methods try to produce more stylized and suggestive results. One of these methods, the cel shading or cartoon shading, tries to render pictures that look like good old hand-drawn comics. An example for this is given in figure 36, showing a teapot rendered with a photo realistic renderer (top) and with cartoon shading (bottom). [Decaudin 2006]

Most algorithms for cel shading consists of two parts. First the object is shaded in a way that all points illuminated by a light source get the same color, or a limited number of colors depending on the brightness. This gives the objects the typical comic look. The second part is the calculation of dark outlines, or if wanted of hard edges in the object. There are many different approaches to perform these two tasks, some of them will be explained in the next sections.

Some cartoon shaded renderers support also the rendering of some other nice features. One of them is the keeping of highlights (phong highlights), that give a better feeling of the consistency of the objects material. Another feature are shadows, especially shadows that are not exact, but look more comic like (see [Gulbrandsen 2010]).

6.1 Algorithm

Most toon shading techniques calculate the first step, the reduction of the shading values, as a direct output from the 3D scene rendering. One method for this is the one described by [Decaudin 2006], calculating the intensity of an object by using a slightly modified version of the Phong illumination model:

$$\begin{aligned} color =& diffuse_{source} \times diffuse_{material} + \\ & max(v \cdot n, 0) \times specular_{source} \times specular_{material} \end{aligned} \tag{37}$$

The images for each light get then combined to calculate the result image. This algorithm calculates an image with constant color over an object. This is not the best looking possibility for cel shading,

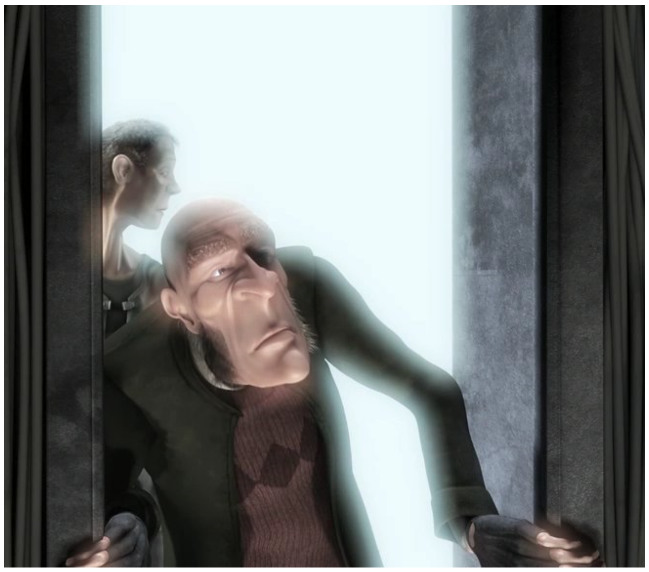

Figure 35: Bloomed image from the film "Elephants Dream".

but allows the usage of fixed function pipeline graphic hardware and keeps the highlights, giving the user a better immersion of the materials appearance.

The major problem, the lack of back face shadows, can be handled in two ways. The first one is the usage of shadow mapping for shadows, which calculates back face shadows as a by-product. Another possibility would be to add a simple term to formulae 37:

$$color = max(sgn(l \cdot n),0) \times diffuse_{source} \times diffuse_{material} + max(s \cdot n,0) \times specular_{source} \times specular_{material} \tag{38}$$

This extension calculates colors only if the dot product between the normal n and the lightvector l is positive. Figure 36 (bottom) shows an image, calculated with the shadow mapping option. You can see that all lightened parts of the teapot have the same color.

Another algorithm (described in [Horsch 2007]) is, that in the fragment shader the shading value is calculated using Phong's illumination model. This smooth shading values get then mapped to discrete values, for example four different ones. In a high level shading language like GLSL, this step can be performed very fast and easy by rounding each smooth value up to the next value. For example the value in the range of $[0, 0.25]$ gets mapped to 0.25. Figure 37 shows the difference to the previous method. Here the object is not only coloured with one color, but with four different colors, depending on the light intensity.

It is also possible to perform the rounding in a post processing stage. This can be very useful if the comic drawing effect has to be added to an existing rendering engine, because it can be hard to change all shaders to perform the rounding.

A speed up can, according to [Barla et al. 2006], be achieved when the brightness values are stored in a 1D texture from dark to bright. When a value, that is according to the dot product in the Lambertian shading model always between 0 and 1, should be mapped to one of these discrete values, the texture can be queried with nearest neighbour filtering, allowing to map the value to the nearest one in the texture. The only difference to the previously described calculation is, that values get not only rounded up, but get mapped to the nearest one.

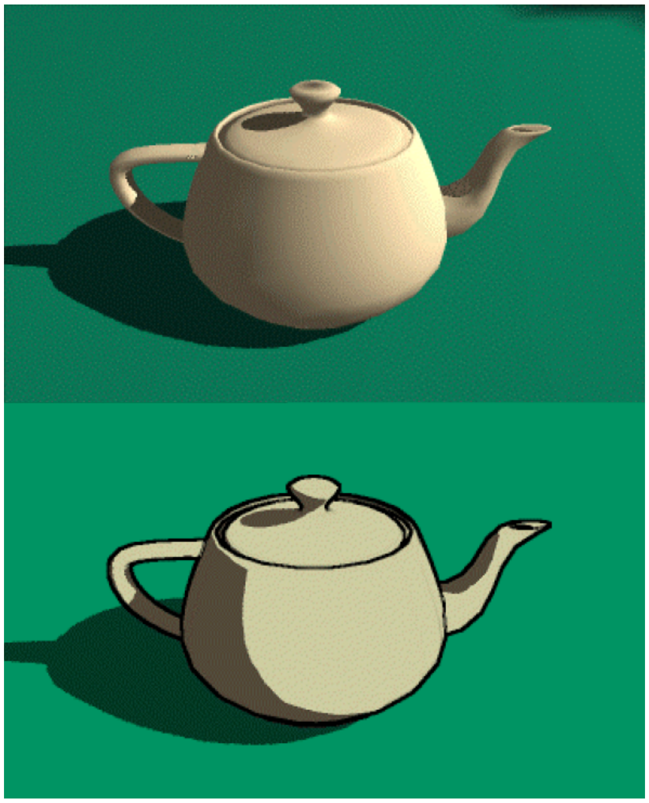

Figure 36: A teapot rendered with a photo realistic method (top) and with cartoon shading (bottom). (image from [Decaudin 2006])

6.2 Line Drawing

First implementations of line drawing algorithms for cel shading tried to create the outline edges of objects only. This was done by inverting the back face culling and drawing the scene only in black. To produce thick lines this back face drawing has to be performed multiple times with different offsets. There are two major problems with this method. First is that lines can only be drawn outside of an object, but not in an object. As [Decaudin 2006] describes in his work, dark lines in a comic strip correspond to locations in the scene, where the view vector is a tangent to the objects surface. The other problem is the performance aspect, as the back faces have to be drawn multiple times.

[Decaudin 2006] and [Horsch 2007] show in their works two approaches working with post processing. [Decaudin 2006] uses the z-buffer of a rendered scene, because it is calculated as a by-product of the coloring step. On this depth information he performs 3x3 differential operator g of order one.

$$g = \frac{1}{8}(|A-x| + 2|B-x| + |C-x| + 2|D-x| + 2|E-x| + |F-x| + 2|G-x| + |H-x|) \tag{39}$$

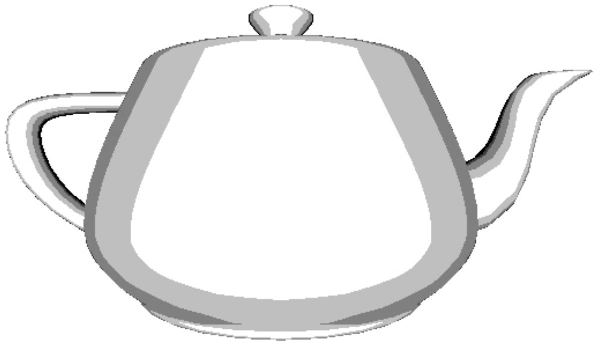

Figure 37: A teapot rendered with [Horsch 2007]'s algorithm. (image from [Horsch 2007])

where A...H correspond to the positions (x) neighbourhood as shown in figure 38. This operator is then used to identify gradients in the z-buffer image, using g with a threshold gives p, that identifies outline edges in the image.

A	B	C
D	x	E
F	G	H

Figure 38: Neighbourhood of point x for the differential operator. (image from [Decaudin 2006])

$$p = min\left\{\left(\frac{g_{max}-g_{min}}{k_p}\right)^2, 1\right\} \tag{40}$$

where g_{min}, g_{max} are the minimum and maximum gradients in the 3x3 neighbourhood and k_p is the threshold. When k_p is high, there will be less edges than with a small k_p.

The calculation of edges in an object can be performed with a higher order differential operator on the z-buffer, but this is numerical not stable. [Decaudin 2006] suggest to use a first order differential operator to an image with normal information of the scene, which can be calculated using multiple render targets. The operator can be the same as for the objects outline, with respect to the fact that normals are three dimensional.

The method described by [Horsch 2007] uses an edge detection filter, as the Laplace filter ([Jaehne 2005], on the color image. This operation can be done very fast on the graphic hardware.

Results from [Horsch 2007] and [Decaudin 2006] can be seen in images 39 and 40.

6.3 Comic Style Shadows

As described above, [Decaudin 2006] as well as [Horsch 2007] can calculate back face shadows as booth work with Phong's illumination model. But in comic styled rendering it is often not wanted to

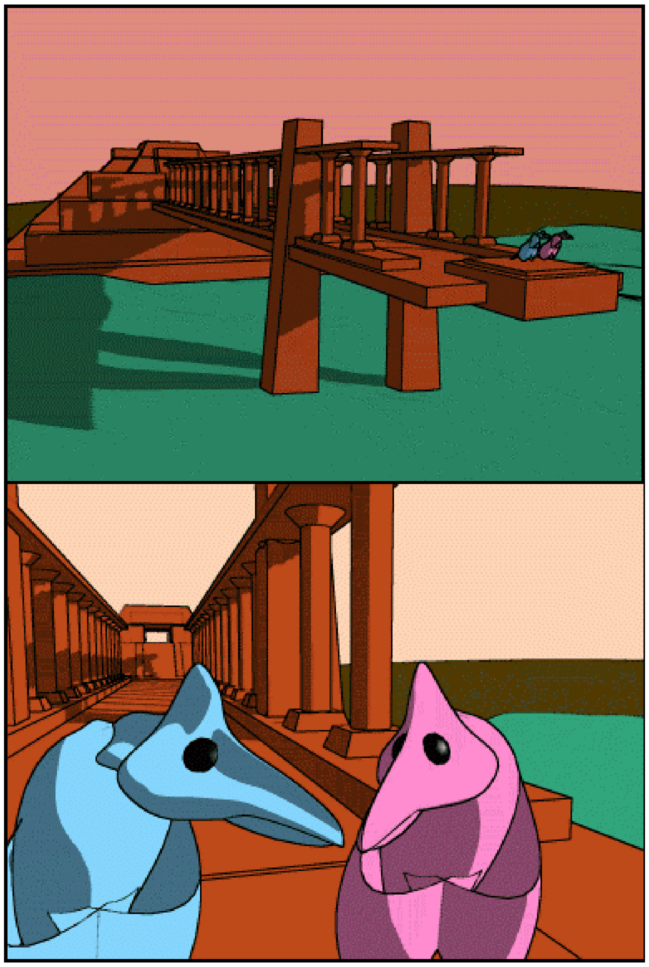

Figure 39: Results of [Horsch 2007]'s algorithm. A scene from (image from the movie Rendez-vous)

separate the lightened areas exactly from the ones in shadows, as shown in image 42 (left). [Gulbrandsen 2010] proposes not to use the exact normals of the geometry, but to use a simple hull geometry, that's normals get stored in a normal map. Such a hull for the model shown in image 42 can be seen in 41.

In the fragment shader the normal of the geometry and the normal of the normal map gets interpolated using a factor β which allows the user to control the amount of influence of the hulls normal. This algorithm produces smoother shadows, masking unwanted details. Such shadows can often be found in mangas, the Japanese comics. Results from [Gulbrandsen 2010] can be seen in figure 42 (right), especially in the person's face.

6.4 Attention based details

In hand drawn comics the unimportant parts of a scene, for example objects in the background, are drawn with less details as the important ones in the foreground. [Barla et al. 2006] refers to that as Level of Attention (LOA) and suggest to use two dimensional textures for the brightness lookup.

These textures can be seen as a stack of one dimensional cel shading brightness textures (along the x-axis) with different level of detail (along the y-axis, 0 is lowest detail level). An example texture is

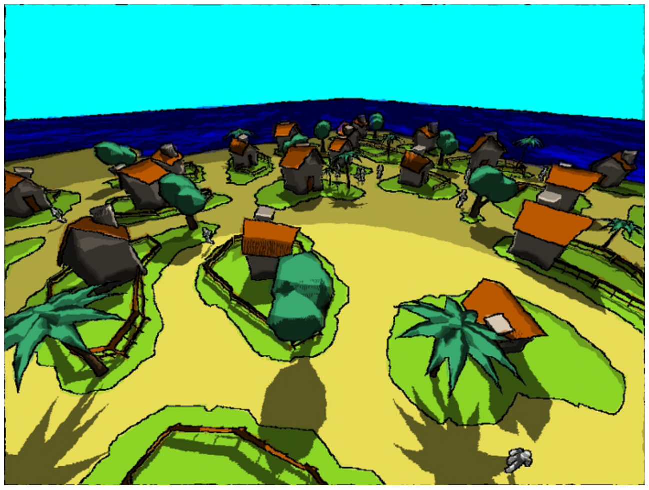

Figure 40: Results of [Horsch 2007]'s algorithm. (image from [Horsch 2007])

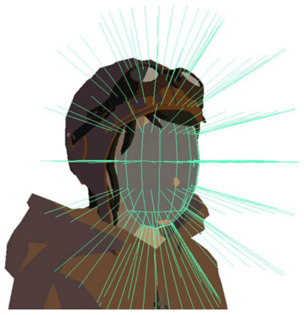

Figure 41: Hull with normals for the model shown in 42. (image from [Gulbrandsen 2010])

given in image 43, where D is the detail level and $n \cdot l$ refers to the Lambertian shading model.

There are two methods of choosing the detail level, described by [Barla et al. 2006], for a given pixel. The first one defines the detail level only in addiction to the depth in the scene. The formulae for this is:

$$D = 1 - \frac{log(\frac{z}{z_{min}})}{log(\frac{z_{min}}{z_{max}})} \tag{41}$$

where z_{min} is the depth before the highest detail level is used and z_{max} the one of the lowest detail level. z_{max} can also be written as $z_{max} = r \times z_{min}$ with a scaling factor r. This leads to a simpler version of formulae 41:

$$D = 1 - log_r(\frac{z}{z_{min}}) \tag{42}$$

This formulae depends only on the parameters of the minimal and maximal details depth, but does not allow to draw the highest detail at a given focal distance (z_c), similar to depth of field. For this another formulae for D can be used:

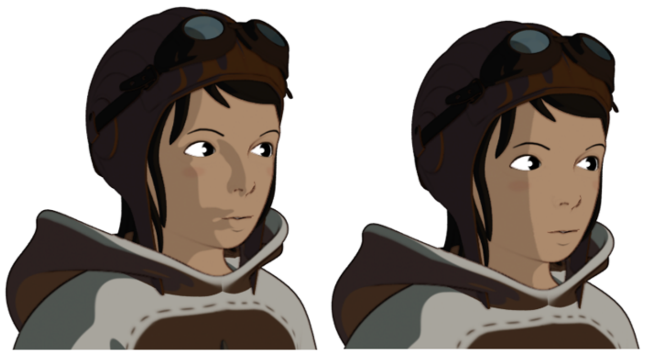

Figure 42: Shadows in comic styled rendering. Left: Shading with geometry normals only, Right: Shading with a combination of geometry normals and hull normals (image from [Gulbrandsen 2010])

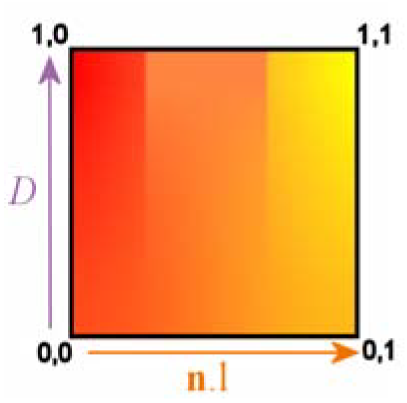

Figure 43: Example for a 2D detail brightness map (image from [])

$$D = \begin{cases} 1 - log(\frac{z}{z^-_{min}})/log(\frac{z^-_{max}}{z^-_{min}}), & z < z_c \\ log(\frac{z}{z^+_{min}})/log(\frac{z^+_{max}}{z^+_{min}}), & z \geq z_c \end{cases} \tag{43}$$

with $z^{\pm}_{min} = z_c \pm z_{min}$ and $z^{\pm}_{max} = z_c \pm r z_{min}$

6.5 Free hand sketches

Hand drawn sketches are in general drawn by the artist only using lines in one color. This can be achieved very easy by only using the techniques described for line drawing in section 6.2. But only very few artists have the ability to draw exact lines, so it is, according to [Horsch 2007] needed to deform the image. To make hand sketches more realistic it is needed to draw the lines a few times, every time with a different warping factor. It can also look good when shadows are also drawn in the image. Results for this can be seen in figure 44.

7 Conclusion

Post processing effects are a very important part of today's computer games. Nearly every modern game has some type of post processing in its rendering pipeline, trying to render either more realistic images or create fancy effects on the screen. For some of the

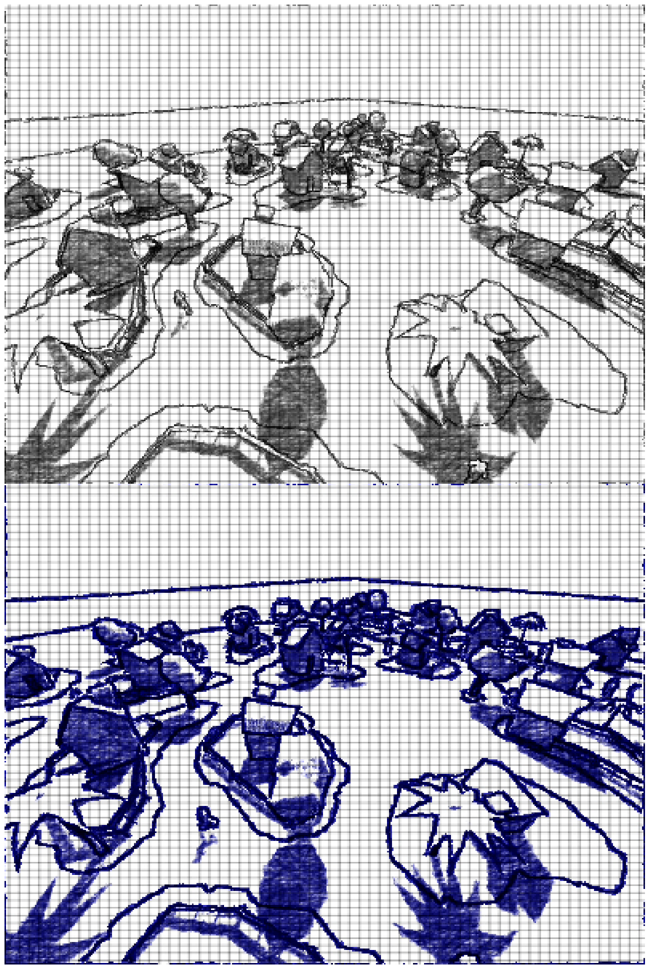
Figure 44: Pencil drawing (top) and ink drawing (bottom) (image from [Horsch 2007])

effects mentioned above like Glow or Bloom there is an agreement about how these can be produced, other ones like motion blur are an ongoing field of research producing new and better or faster ways of rendering them every year. In the last few years there was even a massive development of new effects like screen space ambient occlusion and others.

Beside computer games and film rendering there is another big research area, the virtual reality, in which post processing effects play an important role, as they can simulate effects depending on the used camera system. In this area a lot of work is done in improving post processing effects so that they can produce more and more realistic results in real time.

From today's point of view the usage of post processing effects will increase in the next generation of video games, as the development of faster hardware allows programmers to create complexer effects and use more of them for every frame.

References

ARCE, T., AND WLOKA, M., 2002. In-game special effects and lighting. http://www.nvidia.com/object/gdc_in_game_special_effects.html.

BALA, K., AND DUTR, P., 2010. Motion blur for textures by means of anisotropic filtering.

BARLA, P., THOLLOT, J., AND MARKOSIAN, L. 2006. X-toon: an extended toon shader. In *NPAR '06 Proceedings of the 4th international symposium on Non-photorealistic animation and rendering*.

BARSKY, B. A., AND KOSLOFF, T. J. 2008. Algorithms for rendering depth of field effects in computer graphics. In *ICCOMP'08 Proceedings of the 12th WSEAS international conference on Computers*, World Scientific and Engineering Academy and Society (WSEAS) Stevens Point, 999–1010.

BARSKY, B., HORN, D., KLEIN, S., PANG, J., AND YU, M. 2003. Camera models and optical systems used in computer graphics: Part ii, image-based techniques. In *Computational Science and Its Applications ICCSA 2003*, vol. 2669 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 983–983.

BERGEN, J., BURT, P., HINGORANI, R., AND PELEG, S. 1990. Computing two motions from three frames. In *In Proceedings of International Conference on Computer Vision 1990*, IEEE, 27–32.

BERTALMÍO, M., FORT, P., AND SÁNCHEZ-CRESPO, D. 2004. Real-time, accurate depth of field using anisotropic diffusion and programmable graphics cards. In *3D Data Processing, Visualization and Transmission, 2004. 3DPVT 2004. Proceedings. 2nd International Symposium on 3D Data Processing, Visualization, and Transmission*.

BLACK, M. J., AND ANANDAN, P. 1996. The robust estimation of multiple motions: parametric and piecewise-smooth flow fields. *Computer Vision and Image Understanding 63*, 75–104.

BROSTOW, G. J., AND ESSA, I. 2001. Image-based motion blur for stop motion animation. In *SIGGRAPH '01 Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, ACM New York, 561–566.

BURT, P. J. 1981. Fast filter transform for image processing. *Computer Graphics and Image Processing 16*, 1, 20–51.

CHEN, Y. C. 1987. Lens effect on synthetic image generation based on light particle theory. *The Visual Computer 3*, 3, 125–136.

CROW, F. C. 1984. Summed-area tables for texture mapping. In *SIGGRAPH '84 Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, ACM New York, 207–212.

DECAUDIN, P. 2006. Cartoon-looking rendering of 3d-scenes.

DEMERS, J., 2003. Depth of field in the 'toys' demo. Ogres and Fairies: Secrets of the NVIDIA Demo Team presened at GDC2003.

GULBRANDSEN, O. 2010. Controlling the dark side in toon shading. In *SIGGRAPH '10 ACM SIGGRAPH 2010 Posters*, ACM New York.

HAEBERLI, P., AND AKELEY, K. 1990. The accumulation buffer: hardware support for high-quality rendering. In *SIGGRAPH '90 Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, ACM New York, 309–318.

HENSLEY, J., SCHEUERMANN, T., COOMBE, G., SINGH, M., AND LASTRA1, A. 2005. Fast summed-area table generation and its applications. *Computer Graphics Forum 24*, 3, 547555.

HORSCH, M. 2007. Nicht-photorealistisches rendering mit programmierbarer grafik-hardware.

J. KRIVANEK, ZARA, J., AND BOUATOUCH, K. 2003. Fast depth of field rendering with surface splatting. In *Computer Graphics International, 2003. Proceedings*, IEEE, 196–201.

JAEHNE, B. 2005. *Digitale Bildverarbeitung, 6. Auflage*. Springer-Verlag Berlin Heidelberg.

KOENDERINK, J. J. 1981. The structure of images. *Biological Cybernetics 50*, 5, 363–370.

KOSLOFF, T. J., TAO, M., AND BARSKY, B. A., 2008. Depth of field postprocessing for layered scenes using constant-time rectangle spreading.

KRAUS, M., AND STRENGERT, M. 2007. Depth-of-field rendering by pyramidal image processing. *Computer Graphics Forum 26*, 3, 645–654.

LADYZHENSKAYA, O. A. 1985. *The Boundary Value Problems of Mathematical Physics*. Springer Berlin / Heidelberg.

MAX, N. L., AND LERNER, D. M. 1985. A two-and-a-half-d motion-blur algorithm. In *SIGGRAPH '85 Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, ACM New York, 85–93.

MULDER, J. D., AND VAN LIERE, R. 2000. Fast perception-based depth of field rendering. In *VRST '00 Proceedings of the ACM symposium on Virtual reality software and technology 2000*, ACM New York, 129–133.

NGUYEN, H. 2007. *GPU Gems 3*. Pearson Education, Inc.

PERONA, P., AND MALIK, J. 1990. Scale-space and edge detection using anisotropic diffusion. *IEEE Transactions on Pattern Analysis and Machine Intelligence 12*, 7, 629–639.

POTMESIL, M., AND CHAKRAVARTY, I. 1981. A lens and aperture camera model for synthetic image generation. In *SIGGRAPH '81 Proceedings of the 8th annual conference on Computer graphics and interactive techniques*, ACM New York, 297–305.

POTMESIL, M., AND CHAKRAVARTY, I. 1983. Modeling motion blur in computer-generated images. 389–399.

RANDIMA, F. 2004. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Education, Inc.

RIGUER, G., TATARCHUK, N., AND ISIDORO, J. 2003. Real-time depth of field simulation. In *ShaderX2 Shader Programming Tips and Tricks with DirectX 9*. Wordware Publishing Inc.

RITCHIE, M., MODERN, G., AND MITCHELL, K. 2010. Split second motion blur. ACM New York.

SCOFIELD, C. 1994. *Graphics Gems III*. Academic Press Professional, Inc. San Diego.

SHIMIZU, C., SHESH, A., AND CHEN, B. 2003. Hardware accelerated motion blur generation. *EUROGRAPHICS 22*.

SHINYA, M. 1994. Post-filtering for depth of field simulation with ray distribution buffer. In *Proceedings of Graphics Interface 94*, Canadian Information Processing Society, 59–66.

TWEED, D., AND CALWAY, A. 2000. Motion segmentation based on integrated region layering and motion assignment. In *Proceedings of Asian Conference on Computer Vision 2000*, 1002–1007.

WITKIN, A. P. 1987. Scale-space filtering. In *Readings in Computer Vision*. Morgan Kaufmann Publisher Inc.

ZHOU, T., CHEN, J. X., AND PULLE, M. 2007. Accurate depth of field simulation in real time. *Computer Graphics Forum 26*, 1, 15–23.