Hardware Architecture for Fast Camera Effects

Gábor Szijártó
Budapest University of Technology and Economics
Magyar Tudósok Krt. 2
H-1117, Budapest, Hungary
szijarto.gabor@freemail.hu

ABSTRACT

Current commercial graphics cards do not offer fast enough special hardware to render camera effects like motion blur or depth of field. Even the pixel shader, which is useful in many rendering algorithms, is unable to support these effects. This paper presents a hardware structure, which provides fast rendering capability for realizing filtering effects, including also depth of field and motion blur.

Keywords

Keywords: camera effects, depth of field, motion blur, image processing, fast hardware filtering, edge detection, texture filtering, human eyes, real-time rendering, graphics render pipeline.

1. INTRODUCTION

Current graphics cards do not offer good support for rendering camera effects, such as the motion blur. On the other hand, the images are results of two fundamentally different techniques: triangle rendering and texturing. Triangle rendering is accurate, but texture filtering is not. This duality appears unnatural for the human eye [Szi02a], thus everybody can recognize a real-time rendered image easily. The result of triangle rendering is a sharp image, especially without *full scene anti-aliasing* (FSAA), thus all objects seem to be in focus. Mipmap based texture filtering, on the other hand, is based on averaging four texture pixels, whose size is proportional to the mipmap level. Thus image portions calculated this way can be neither sharp nor accurate. Moreover, texture anisotropic filtering is not accurate enough on current graphic cards, thus the texture is more blurred in perspective field than in other perspectively not distorted areas (see Fig 1). Therefore textured pixels look as if they were out of focus. In still images this unnatural blurring results in incorrect perception of object position. However, when the picture is in motion, the human eye can correct this problem. Another problem is posed by fast moving objects. The human brain cannot recognize the full process of the motion, thus it seems that fast moving objects disappear and appear at another place.

WSCG SHORT PAPERS proceedings
WSCG'2003, February 3-7, 2003, Plzen, Czech Republic.
Copyright UNION Agency – Science Press

Depth of field (DOF) helps to improve the blurring in pictures, and motion blur helps to recognize fast object motion. Thus these camera effects are important to create more realistic real-time rendered images. However, these effects decrease information of images in some aspects. For example, first person shooter (FPS) gamers like to see the image sharply to quickly recognize the enemy. Thus many FPS gamers turn off even the textures. This is an extreme case, in most of the cases camera effects are useful and improve visual quality.

Figure 1. Example of blur chaos (screenshot from an FPS game). Soldier and tank looks like in the air due to poor texture anisotropic filter and sharp triangle edges.

In the next section, the camera effects are described from the aspects of physics. In section 3 a new

hardware solution is described. In the section 4 this hardware solution is analyzed, and some theoretical performance calculations are presented. In section 5 depth of field is described and previous work is presented in this field. Finally, a depth of field algorithm is implemented for the proposed hardware architecture. Section 6 summarizes the results.

2. CAMERA EFFECTS

A camera is very similar to the human eye. Both have a lens and an image plane (retina). The lens has finite area in order to collect light rays from the objects. A small pinhole cannot collect enough rays to be sensed by retina of the eye or CCD or film of the camera. An ideal convex lens has a specified focal distance, and is able to project sharply only those objects that are at this distance, thus there are many blurred objects in a photograph (see Fig. 2). If an object is nearer than the focal point, the sharp projected image will be behind of the retina or the image plane. To move an object into the focus, we should either change the focal distance of the lens or modify the distance between the image plane and the lens. Ordinary camera has fix glass lens, thus it can use only the latter solution. On the other hand a human eye has theoretically fix distance between retina and lens, but lens is elastic, thus muscles stretch it to find the right focus.

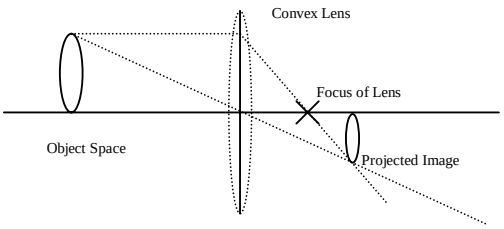

Figure 2. Example of Focus.

Pinhole Camera Model

Current real-time computer generated images are rendered with the assumption of an infinitely small "pinhole" lens. All objects scatter light rays in every direction in the world space. Some rays will pass through the pinhole if the source object is not occluded by another object. There is no lens refraction, thus the focus of lens is meaningless. The image plane can be anywhere, and the projected image on the image plane will be sharp (see Fig 3).

Thin Lens Model

Obviously, real lenses have a finite area, and rays refract on lenses. Thus the projected image of a source point is sharp only at a single image plane position, which depends on the focus of lens. The projections of most of the source objects are blurred, because these are behind or in front of focus, where rays are not concentrated in one point.

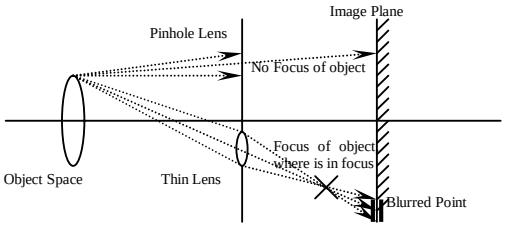

Figure 3. Infinitely small pinhole and thin lens.

Object Motion

The sensation of human eye receptors is not infinitely fast, thus a receptor still senses rays for a while, after its source is turned off. This makes images of fast moving objects blurred. This effect can also be observed in cameras since in order to collect enough light for the film, the shutter of the camera is opened for a given time. This effect is called *motion blur*. This effect is difficult to simulate in computer graphics, because it requires the calculation of object motion between two frames. One possible method is rendering several static images between two frames and blending them. The speed of this solution is proportional to the number of the blended images.

3. INVERSE POST FILTER

Previews

Current real-time hardware has many programmable units to implement a lot of visual effects. Advanced commercial graphics cards have two programmable units: a pixel shader and a vertex shader. A vertex shader can modify the geometry of the objects, thus this unit is not suitable for image filtering. Pixel shader, on the other hand, is responsible for the programmable rasterization in two dimensions. Therefore it is a primary candidate to implement an image filter algorithm. Pixel shader fills a given two dimensional triangle while executing a preloaded program code. This program can read almost any texture pixel and the most advanced cards even implement the flow control [Mic02a] (version 3.0). Thus an image filtering algorithm can be implemented with a pixel shader program. The screen can be covered with two triangles. Thus after rendering the required image it should load the pixel shader code, and render two triangles, which cover the whole screen, carrying out the filtering operation. The main disadvantage of his method is that this architecture is not designed for image filtering, thus cache hitting is not optimal. Increasing cache size is a possible solution to solve this problem, but it is usually prohibitively expensive.

Traditional Algorithm

To create camera effects, the easiest way is to apply filtering with an appropriate mask. The size of such masks is $n \times n$, and their center is in the filtered computed pixel. The result is computed from the elements of this mask and from the neighborhood of the actual pixel. In the simplest approach only addition and subtraction instructions are used to calculate the required pixel. If we can use more instructions, the filter will be able to realize more advanced effects. When the filter uses all n^2 elements, the processing time will be slow. If more execution units work in parallel, the rendering time can be reduced proportionally with the number parallel execution units.

The other main drawback of image filtering is memory accessing. Filter algorithms need many memory reads. Current arithmetic processors can process data significantly faster than fetching data from the main memory. For realizing a filter algorithm, we have to read n^2 pixels for one result. Memory bottleneck can be attacked by cache memories. For the best performance, the required cache size is

(min(image *width* , *image height*) + *mask size* - 1) *
(*mask size* - 1)*(*pixel data size*) .

Unfortunately, this number can be rather high.

Algorithm

Filtering performance can be improved by better resource distribution. This solution is similar to pipelining. To reach this goal, one pixel calculation should be decomposed into many small stages. A possible solution is the *inverse calculation*, when the effect of a pixel is simultaneously distributed to all pixels that might be affected, instead of gathering the effects of pixels that might affect a certain pixel [Had01]. In the distribution approach, the effect of the pixel of the mask center is calculated onto other elements of the mask. Thus one calculation cycle is reduced from n^2 to 1. To increase processing speed, each element of the mask must be a processing unit. To implement this algorithm, we should store the temporary results and the final value is computed from these temporary values. The temporary pixel information can be stored in a pipe, which can be realized by a configurable shift register. A shift register can be implemented in hardware much more easily than a full access memory. Full access memories have many wire crossings and wires are very long. Shift registers do not have many wire crossings and the average wire length is short.

Architecture

To take these facts into consideration, we have designed a new hardware architecture, which is called the *Inverse Post Filter*. It has basically four different parts: an *input processing unit*, an *output processing unit*, an *execution mask unit* and a *shift register array*. Practically all units are parts of a long data pipe. The beginning of the pipe is the input processing unit, which reads and processes the raw pixel data from the video memory, including the z-buffer value and the pixel color information. The shift register array only stores data and shifts this data at each clock cycle. The execution unit calculates temporary results. At the end the output processing unit calculates the final result from the temporary data.

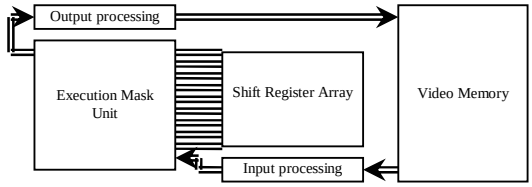

Figure 4. Inverse Post Filter architecture.

In the following subsections each unit is described in detail, and an example is presented, in which the execution mask size is only 3x3.

3.1.1 Input/Output Processing Unit

These two units are very similar, but their functions are basically different. Input processing unit fetches the required data from the video memory. Moreover, it sets initial values of the temporary data. For example, this data can be a pixel color or the radius of the depth of field. At the end of the pipe the output processing unit calculates the final result from the calculated data.

3.1.2 Shift Register Array

The main function of the shift register array is storing data, which is used in one of the following clock steps. The number of register pipes in the shift register is one less than the number of rows in the filter mask. Only the first element of the pipe can be written and only the last element can be read at each clock cycle. The rendering target can be large, thus we must use a long configurable shift register, in order to avoid wasted step cycles. For example, if the inverse post filter has a 1600 stage long fixed shift registers and the rendering target is only 320, then 1280 cycles will be wasted, which is a great deal of unnecessary wasted time. This problem can be solved by a configurable length shift register.

The main idea to construct a configurable length line is based on the fact that all integer numbers can be expressed as the sum of exponents of two. Thus the shift register is made from exponents of two length stages (1, 2, 4, 8 … length stages). Each stage can be disconnected (see Fig. 5); in this case the data does not enter into that stage. When the stage is connected the total length is increased by the length of the stage. Thus a shift register pipe can be configured with only logarithm of the maximum line length number of switches.

$$N_{switches} = \lceil \log_2 (LengthOfPipe) \rceil$$

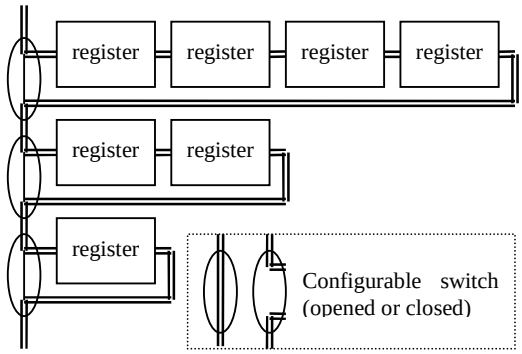

Figure 5. Configurable shift register.

3.1.3 Execution Mask Unit

The execution mask unit is an image processing mask (see Fig. 6). At each mask position there is an execution processor. The center position is special since it affects others in inverse processing. Center position data can be read by others and use this information to calculate their temporary data.

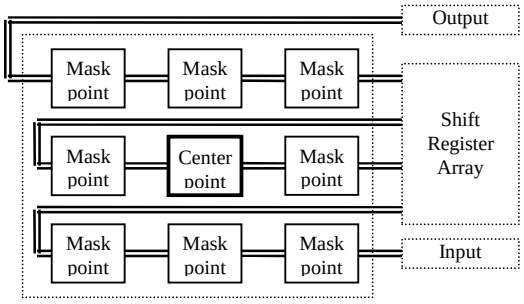

Figure 6. Execution unit mask.

One execution unit is similar to a pixel shader unit. It has an arithmetic processor, a temporary memory, a constant memory, center pipe data and own pipe data. The processor has full access to its own pipe data, but the data of the center position pipe is read only to access all mask positions. The processor calculates the filtered value and writes it into its own pipe data. Each processor has a unique temporary memory to create some intermediate calculation. Thus this memory is readable and writeable. The processor can be only a simple ALU processor unit or a complete processor with flow control. Each processor has the same program code with the difference of constant values that determine the position of the mask or the distance from the center position.

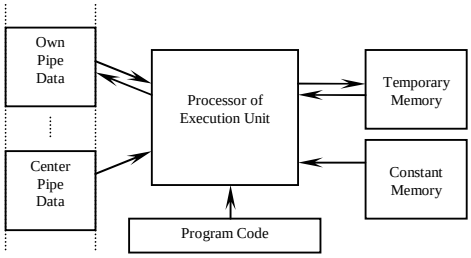

Figure 7. One execution unit.

4. ADDITIONAL CALCULATIONS

Performance Calculations

The performance of the inverse post filter is difficult to analyze since it depends on the complexity of the algorithm in the execution mask unit, processor speed and memory speed. Each of these factors can be the bottleneck of system. Thus we can provide only a theoretical calculation.

At first suppose that the video memory is the bottleneck. It means that the execution unit can process data in one clock cycle. Current video cards have about 270-320 MHz DDR RAM that has 128bit or 256 bit memory interface. Thus the peak bandwidth is 20 Gbyte per second. Moreover, suppose that the rendering target dimension is 1600x1200 and it uses 32bit color and 32bit z-buffer information. In this case each rendering surface must be read, and only color information must be written. Thus the accessed data size is approximately 22 Mbyte. It results that the total calculation time is about 1ms (1000fps).

In fact if the execution algorithm is complex, the video processor cannot provide fast enough calculation speed. In this case the processing time is directly proportional to the execution time of the algorithm. At the previous example the speed will be 1ms*(*algorithm time*) if the core and memory clocks are the same. Our aim is at least 20fps, which corresponds to 50 ms calculation time. Thus a realistic time for post filter calculation is about 20-30 ms, which means 20-30 clock cycles. This time would be enough even for 80-120 instructions if the resolution was only 800x600. With this instruction number, quite sophisticated algorithms can be implemented.

Further Architecture Solution

The bottleneck of a complex post filtering algorithm is the processor, which means that the memory is not used in many clock cycles. Thus rendering and post filtering are worth executing in pipeline. In this case the frame rate is raised, but the rendering latency can still be high. For the human observers, 20-25 fps speed is the minimum requirement. However, long latency can result in large reactions times, up to 20-100 ms, which corresponds to 50-10 fps.

The mask size is important for many visual effects. If the mask size is small, effects will be less spectacular, because the maximum distance between two pixels, which are affected together, is small. On the other hand, too large mask size requires a huge shift register and many processor units.

To reduce the number of execution units, one unit can execute data at more than one mask position.

To reduce the shift register length, the length of the last stage cannot be an exponential of two, thus the last stage length is calculated from the following formula.

$$size = \min(width, height) + masksize - 1$$

$$lastloopsize = size - 2^{\lfloor \log_2(size) \rfloor} + 1$$

where *width* and *height* are the width and the height of the maximum render target. Moreover, *lastloopsize* is the length of last stage. Thus if the maximum render target size is 1600x1200 and *masksize* = 17, then *size* = 1216 and *lastloopsize* = 1216-1024+1 = 193. Thus last stage size is reduced from 1024 to 193.

Another possibility to reduce the shift register size is decreasing maximum length of the shift register. If the whole render surface is rendered in four parts, then the maximum shift register length is a quarter of maximum length plus the mask size and minus one. However, in this case the execution has an overhead, which is equal to the mask size minus one at each row. Using the previous example, the shift register's maximum size is 416 and the overhead is

(2*(17-1))*(4-1)*1600 = 153600

stages. Note that this is only 7,88% overhead, but this solution needs about a quarter of the shift register memory.

Filter for Image Processing

Edge detection, blur filters and many other filters can be realized with the mask filter. Each mask position is a number, which determines the weight of the position. For example, a simple edge detection mask is shown in Figure 8.

0	-1	0
-1	4	-1
0	-1	0

Figure 8. Example of edge detections.

To realize this algorithm on the inverse post filter hardware, at first the input processing unit should be activated. It loads pixel color from render buffer into its pipe stage, allocates *tmpColor* and *tmpWeight*, and fills up these values from the video and constant memories. The output processing unit multiplies *tmpColor* with *tmpWeigth* to get the weighted color. The programming code of this algorithm is very short; it is presented in Figure 9. The *positionWeigth* is preloaded into the constant memory, and it contains the mask weight.

```
tmpColor += centerColor * positionWeight;
tmpWeight += positionWeigh
```

Figure 9. Execution unit program code of mask filtering.

5. DEPTH OF FIELD

There are many solutions for the calculation of the depth of field (DOF), but the most of them are too slow to use in real-time computer graphics. In real time rendering accuracy is not so important if there are no objectionable artifacts, and the speed is the primary goal.

Potmesil and Chakravarty [Pot81a] were the first to introduce a DOF algorithm in computer graphics. This algorithm is a linear postfiltering one, and uses RGB and Z depth for to compute the final image. The main disadvantage of this approach is that filtering does not recognize objects partially blocking the CoC's ("circle of confusion") effect. This partial occlusion can cause blurry backgrounds to affect sharp foreground objects.

Potmesil and Chakravarty's method is not accurate in many cases. Thus there were many attempts to create a fast and accurate DOF method. Cook, Porter, and Carpenter [Coo84a] implemented a distributed ray tracing algorithm. Haeberli and Akeley [Hae90a] created an accumulation buffer to provide hardware anti-aliasing, and it can also be used for DOF effects. Shinya [Shi94a] adapted the distributed ray tracing idea and converted it into a postfiltering process. Fearing [Fea96a] introduced an importance ordering algorithm to recalculate the DOF effect only at the relevant part of the image.

DOF with Inverse Post Filter

The aim of the proposed algorithm is to realize a fast depth of field algorithm without visual artifacts. The

following algorithm executed by the inverse post filter hardware meets these requirements.

The input processing unit calculates the depth of field radius (CoC) and the weight of CoC, which is reciprocal of area of the circle. Moreover it clears the weight of the pixel and register of the final color value.

Name	Size	IPU op.
Color	48bit	clear
ColorWeigth	8bit	clear
AreaWeigth	8bit	calculate
Radius	8bit	calculate
OriginalColor	24bit	copy
Z-Value	32bit	copy

Figure 10. One stage of Pipe data. Name, size in bit and input processing unit operation.

The code has to check two main conditions. The first one corresponds to the classical solution. If the center point is nearer than the actual mask point, then the center point affects the actual point for sure. If only this condition was used, artifacts would occur at the borders of those objects which have different CoCs size. In this case the blur of objects will not be continuous, because at one part of the border the foreground and the background objects' weights are about 50-50%, and at the other side these weights are 100%-0%. This creates a strongly noticeable visual artifact for the human eye.

```
if ( centerZ <= positionZ) {
   if (curDistance <= centerCoCRadius)) {
      tmpColor += centerColor * areaCenterWeight;
      tmpWeight += areaCenterWeight;
   }
} else if (positionCoCRadius < curDistance) {
   tmpColor += centerColor * areaPositionWeight;
   tmpWeight += areaPositionWeight;
}
```

Figure 11. Execution unit program code.

In the programming code (see Fig. 11) the second conditions solves this problem. Thus the background object affects the foreground object with CoCs of rendered point. This method is still not fully accurate, but do not create noticeable artifacts.

The *centerZ, centerColor, centerCoCRadius and areaCenterWeight* are data of mask center position. The *centerZ* is the Z-buffer value, *Color* is the color value, *centerCoCRadius* is the CoCs value, *areaCenterWeight* is a weight, which is calculated from area of CoCs. *Position* prefix values are the data of calculated position. More over their meaning are similar to *center* prefix values. The *centerZ, positionZ* and *centerColor* are data from the pipe, and these data came from the rendering target. The *positionCoCRadius, centerCoCRadius, areaCenterWeight* and *areaPositionWeight* are provided by the input processing unit. *curDistance* is the mask position depended constant. The *tmpColor* and *tmpWeight* are calculated for the output processing unit. At the end, the output processing unit computes the final pixel color from *tmpColor* and *tmpWeight.*

Aliased CoC causes another visual artifact, because there is a sharpness line where CoC size changes. This effect is similar to the artifacts of bilinear filtering. Therefore we used a dithered radius in input processing. Another possible solution can be the consideration of the border of the circle.

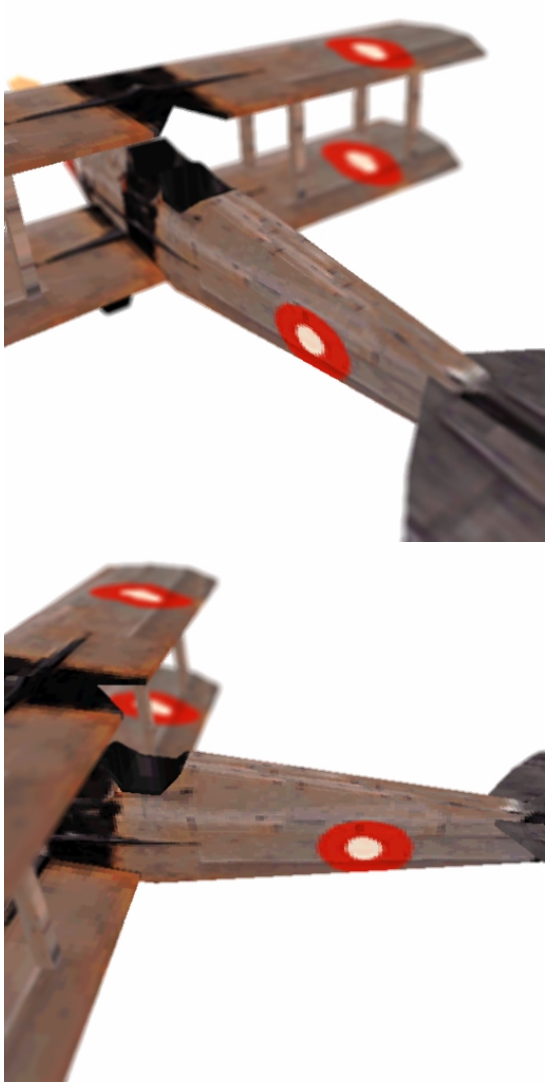

Figure 12. Two examples of Depth of Field algorithm with dithered radius, fuselage is in focus. Texture filtering is only nearest position to stress the DOF effect.

This algorithm consists of at most four instructions, and if it can be executed in five clock cycle, then the example of section 5 (1600x1200x32, 310MHz DDR RAM) can be processed in at most 5ms (200fps).

Figure 13. Examples of Depth of Field algorithm. Original, CoCs size and final images.

6. CONCLUSION

This paper presented a hardware architecture, which can execute image filtering and camera effect algorithms. The hardware is called the inverse post filter, and uses shift registers to cache the temporary information. We also proposed a method to dynamically configure the shift register line length.

Figure 14. Examples of Depth of Field algorithm. Original, CoCs size and final images.

This paper also proposed a fast depth of field algorithm, which do not have objectionable visual artifacts. This hardware also is capable to compute motion blur effects. It is more complex than the depth of field, because it needs pre-calculation at each pixel to determine the motion information of that point. The inverse post filter programming code is also longer. If depth of field and motion blur are used together, it can be realized by a common inverse post filter program.

7. ACKNOWLEDGMENTS

This project has been supported by the Slovene-Hungarian Action Fund (SLO7/01) and by OTKA (T029135).

8. REFERENCES

[Bur81a] Burt, P.J. Fast filter transforms for image processing. Computer Graphics, Image Processing, 6:20-51, 1981.

[Coo84a] Cook, R., Porter, T., and Carpenter, L. Distributed Ray Tracing. Computer Graphics (SIGGRAPH), 18(3):137-145, 1984.

[Fea96a] Fearing, P. Importance ordering for real-time depth of field. In Proceedings of the Third International Conference on Computer Science, pages 372-380, 1996.

[Had01] Markus Hadwiger, Thomas Theußl, Helwig Hauser, M. Eduard Gröller, "Hardware-Accelerated High-Quality Filtering on PC Graphics Hardware". In Proceedings of Vision, Modeling, and Visualization 2001, November 2001, June 2001, Stuttgart, Germany, pp. 105-112

[Hae90a] Haeberli, P. and Kurt, A. The Accumulation Buffer: Hardware Support for High Quality Rendering. Computer Graphics (SIGGRAPH), 24(4):309-317, 1990.

[Mic02a] Microsoft DirectX help, 2002. www.microdoft.com/directx

[Mul00a] Mulder, J. D., and Liere, R. Fast Perception-Based Depth of Field Rendering, 2000.

[Pot81a] Potmesil, M. and Chakravarty, I. A Lens and Aperture Camera Model for Synthetic Image Generation. Computer Graphics (SIGGRAPH), 15(3):297-305, 1981.

[Szi02a] Szijarto, G. Texture Filtering with Summed Area Table in Hardware Rendering, First Hungarian Computer Graphics and Geometrics Conference, 2002.

[Szi95] Szirmay-Kalos, L. Theory of Three-Dimensional Computer Graphics, Publishing House of the Hungarian Academy of Sciences, 1995.

[Shi94a] Shinya, M. Post-filtering for Depth of Field Simulation with Ray Distribution Buffer. In GI, pages 59-66. Canadian Information Processing Society, 1994.