Accurate Depth of Field Simulation in Real Time

Tianshu Zhou, Jim X. Chen and Mark Pullen

George Mason University, Fairfax, Virginia, USA
tzhou@gmu.edu

Abstract

We present a new post processing method of simulating depth of field based on accurate calculations of circles of confusion. Compared to previous work, our method derives actual scene depth information directly from the existing depth buffer, requires no specialized rendering passes, and allows easy integration into existing rendering applications. Our implementation uses an adaptive, two-pass filter, producing a high quality depth of field effect that can be executed entirely on the GPU, taking advantage of the parallelism of modern graphics cards and permitting real time performance when applied to large numbers of pixels.

Keywords: depth of field, post-processing, depth buffer, GPU, leakage

ACM CCS: I.3.3 Computer Graphics: *Picture/Image Generation*: *Display algorithms* I.3.7 Computer Graphics: *Three-Dimensional Graphics and Realism*: *Virtual reality* I.4.3 *Image Processing and Computer Vision*: *Enhancement*: *Filtering*

1. Introduction

Simulation of real lens systems is becoming more important in many 3D rendering applications, from games to the combination of real and synthetic scenes in augmented reality. Generating depth of field is a key part of the simulation of real lens systems, increasing realism and improving perception and user comprehension of the rendered scene.

Most depth of field implementations use either post processing, or a form of super-sampling, to achieve the desired effects. Post-processing methods generally simulate the results of a lens system by blurring the rendered image using a variety of filter techniques, depending on scene depth information. This produces a depth of field effect with a computational cost that is independent of the scene complexity, but these methods can produce noticeable artifacts that may adversely affect the image quality. Super-sampling methods such as image accumulation and distributed ray tracing provide more accurate results by attempting to model the physical lens. This avoids many of the artifacts of the post-processing methods, but comes with high computational cost. Additionally, most depth of field implementations require significant customization of the application and rendering engine, which presents an additional barrier to adoption.

While super-sampling methods are generally still far outside the capabilities of current graphics systems, recent developments in consumer graphics hardware and its accessibility via the latest generation of Application Program Interfaces, arenow bringing post-processing depth of field techniques within the range of modern PC systems for interactive applications.

In this paper, we present a new post-processing method for accurately simulating depth of field effects at interactive frame rates. Our method is an improvement over other post-processing methods in that it simulates depth of field by efficiently applying the depth buffer directly to compute the original scene depth. This eliminates the need for specialized rendering passes to compute custom depth information, and simplifies the implementation, making it easy to integrate into existing rendering applications. Additionally, our two-pass filtering technique provides a high-quality depth of field effect by sampling a large number of pixels, while eliminating intensity leakageartifacts common to many post-processing techniques.

© 2007 The Authors
Journal compilation © 2007 The Eurographics Association and Blackwell Publishing Ltd. Published by Blackwell Publishing, 9600 Garsington Road, Oxford OX4 2DQ, UK and 350 Main Street, Malden, MA 02148, USA.

Submitted September 2004
Revised July 2005
Accepted June 2006

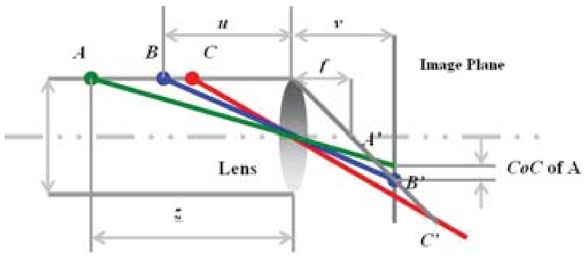

Figure 1: *The thin lens model.*

In the following sections, we first review camera models, the existing approaches for simulating depth of field, a mathematical analysis of computing scene depth, followed by our algorithm, implementation, results and conclusions.

2. Camera Models and Depth of Field

Real-world optical imaging systems such as the human eye, have a finite depth of field. Objects at a certain distance (the focal plane) appear sharp, while objects closer to, or further away, become progressively more blurred. However, these subtle visual cues are missing from images generated by conventional 3D rendering methods that use a pinhole camera model. The pinhole camera model produces images with perfect sharpness, regardless of the distance of the scene objects from the camera. This perfection can affect the perception of the human observer, reducing the realism of the scene.

The thin lens model provides a more realistic model for optic systems. For a lens with a focal length f, a sharp image is projected onto the image plane at distance v, given the object distance u, if the following relationship is fulfilled

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

Otherwise, light rays from the object pass through the lens and intersect the image plane to form an approximate circle, called the *'Circle of Confusion'*. The depth of field is characterized by a circle of confusion that is smaller than the resolution of the imaging system. The diameter of the circle of confusion on the image plane can be calculated by

$$CoC = \left| D \cdot f \frac{(u - z)}{z \cdot (u - f)} \right|, \tag{2}$$

where D is the diameter of the lens, and z is the distance of the object from the camera. Figure 1 shows the basic principles of the thin lens model. Rays from point A produce a circle of confusion on the image plane that is farther away from the focal point A', while rays from point C produce a circle of confusion on the image plane that is closer than the focal point C'. Rays from point B converge to the focal plane and satisfy Equation (1).

More sophisticated lens models can account for effects such as chromatic aberration, the shape of the aperture, defects in the lens, etc. However, these models are computationally expensive to simulate and since depth of field is usually the most prominent aspect of optical systems, most visualization methods, including ours, make use of the thin lens model.

3. Previous Work

A number of approaches have been developed to generate depth of field effects, with varying degrees of success. The following is a brief summary of some common methods.

3.1. Distributed ray tracing

These methods use a multisampling approach to ray tracing. Several camera rays are traced through each image pixel on the projection plane from slightly different projection reference points and the final pixel color is calculated as the average of the intensity of the distributed rays [Che87–CPC84]. This method yields some of the most realistic depth of field images because it more accurately models the actual light transport properties, but is unlikely to become a real-time solution in the near future.

3.2. Image accumulation

These methods model the depth of field from a finite aperture size by rendering the scene multiple times using a pinhole camera model from slightly different camera positions, effectively sampling across the lens [HA90]. The resulting images are blended together in an accumulation buffer. This approach solves problems such as partial occlusion, but may require the scene to be rendered too many times to achieve a high-quality result at interactive frame rates. This is particularly problematic with large circles of confusion, since these require progressively more samples in order to avoid artifacts such as ghosting, and this may in turn cause problems with the precision of the accumulation buffer.

3.3. Layered depth

This method divides a scene into many layers in which the pixels share the same, or similar, distance to the camera [Sco92]. Each layer is then rendered and blurred separately to generate a defocusing effect, and then combined together to produce the final image. This method has problems when a scene contains objects that span a large depth range, such as walls or floors. These objects can be broken up across multiple depth layers, but may then exhibit discontinuities at the layer boundaries.

3.4. Post processing

Most post-processing methods are based on the early work of Potmesil and Chakravarty [PC82]. The main advantage of the post-processing approach is that it is relatively simple to implement, with a cost that is usually independent of scene complexity. Although there are many variants, they all follow a common procedure. Firstly, the scene is rendered into the frame buffer. The scene image is then captured and reprocessed using a filter, and the resulting image is then sent to the display. Most of the variations occur in the filtering stage, where differing algorithms are used to sample and weight the pixel intensities based on depth, or depth-associated information calculated during the initial rendering. Each of the variants has their own advantages and disadvantages. The method proposed by Arce [AW02] suffers from intensity leakage, or pixel bleeding, while Demers resorts to a more costly layered depth to resolve leakage [Dem03]. Riguer *et al.*, provide one of the best implementations to date, and solve the leakage problem, but their filtering method only samples a subset of neighboring pixels, and bases neighboring pixel contributions on the circle of confusion of the center pixel, rather than the circle of confusion of the neighbor [RTI02]. An additional problem inherent in post-processing techniques is that of partial occlusion, where fundamental information regarding the visibility of an object across a finite lens aperture is not available. Solutions to this problem, including Shinya [Shi94], lack hardware support, add significant computation and memory costs and are too slow. In general, post-processing techniques, including ours, do not attempt to solve this problem.

Current work tends to focus on the post-processing methods and the use of graphics hardware in implementing depth of field [AW02–RTI02, BFS04, KZ03], since these are the implementations most likely to achieve real-time frame rates. However, very little performance data are available in previous work, and with the exception of Bertalmio, where it is available; it is usually of the order of a few frames per second [BFS04]. Additionally, existing work devotes little or no time at all to the problem of generating the depth information, preferring instead to focus on the filtering techniques. This creates a number of problems. If the computed depth information is not accurate, it reduces the value of filtering based on accurate circle of confusion computations. On the other hand, in an effort to achieve accuracy, many techniques require special capabilities of the GPU or graphics API that make it difficult to integrate the technique into existing rendering applications. Our approach resolves these issues.

4. Technical Details

4.1. Depth computation from a nonlinear depth buffer

Since one of the goals of our method is to make it easy to integrate into existing rendering applications, we avoid the traditional technique of custom shaders and special render targets to generate depth information during scene rendering, and instead make use of the depth information stored in the depth buffer as part of the normal rendering process. However, the depth values stored in the depth buffer are nonlinear, and need to be mapped back to linear, scene-based depth values. A naive technique for this is to invert the projection matrix, and use this to compute the full 3D position of the fragment (relative to the camera). Since we require only the depth value z, not the x and y values, this is unnecessarily expensive, and we present a faster technique.

Given a point p in world coordinates, the mapping to canonical viewing volume coordinates p', is done by the projection matrix P.

$$p' = P \times p. \tag{3}$$

For a perspective projection,

$$P = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f+n}{n-f} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{4}$$

where l, r, t and b are the left, right, top and bottom edges of the view volume, and n and f are the near and far clipping planes, respectively.

Given a point (in homogeneous coordinates)

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

and the corresponding transformed point

$$p = \left\{x', y', z', 1\right\},$$

the transformed 3D point in canonical viewing volume is $(\frac{x'}{w'}, \frac{y'}{w'}, \frac{z'}{w'})$, and the corresponding value saved in the depth buffer is $\frac{z'}{w'}$. From Equations (3) and (4) we get

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

and

$$w' = -z,$$

where

$$P_{33} = \frac{f+n}{n-f}, \tag{5}$$

$$P_{34} = \frac{-2f \times n}{f-n}. \tag{6}$$

Therefore, the depth value z can be represented as

$$z = \frac{-P_{34}}{\frac{z'}{w'} + P_{33}}, \tag{7}$$

Given that we can obtain $\frac{z'}{w'}$, from the depth buffer, and both P_{33} and P_{34} can be computed based on knowledge of the camera properties, we can solve this equation to convert a value from the depth buffer to an actual distance from the camera. Furthermore, by reducing the problem to a scalar computation, we can take advantage of the vector maths capabilities of the GPU to compute the depth values of up to four pixels simultaneously.

4.2. Depth of field algorithm

To produce our depth of field effect, we use a two-pass filter, similar to the application of a separable Gaussian filter. On the first pass, a vertical filter is applied to each pixel, sampling and weighting neighboring pixels according to criteria discussed below. The output of this pass is then used as the input to the second pass, which performs a horizontal sampling, blending the result and then renormalizing, to produce the final output image.

However, our filter is adaptive, with the weights dynamically adjusted for each pixel depending on calculated circle of confusion sizes. Performing a two-stage filter like this gives us the ability to sample more pixels than are typically possible with a single-pass filter, which usually sample only ten or twelve neighbors and are more likely to suffer from aliasing artifacts. Additionally, it eliminates the need to perform down-sampling of the frame-buffer image, either as a prerequisite to the post-processing, or during pixel lookups, which further reduces the bandwidth requirements.

Our method uses three factors to control the weights applied to each pixel in the filter. The first factor affecting the filter weight is the overlap function, $O(r_p)$. If the circle of confusion of a sample pixel P, does not overlap the center pixel C, then P makes no contribution to C. As the circle of confusion grows, it will make a partial contribution, based on the degree of overlap, until the circle of confusion totally overlaps C, at which point it achieves maximum contribution. Thus, $O(r_p)$ is defined as follows:

$$O(r_p) = \begin{cases} 0, & r_p \leq d_p \\ r_p - d_p, & d_p \leq r_p < d_{p+1} \\ 1, & r_p \geq d_{p+1} \end{cases},$$

where r_p is the radius of the circle of confusion of pixel P, d_p is the distance of pixel P from center pixel C and d_{p+1} is distance of the next pixel in the filter outside P, both measured in pixel widths. This avoids discretizing the circle of confusion to a fixed number of pixels, which helps reduce visual artifacts such as popping. Figure 2 demonstrates the contribution of a sample pixel P, to the filter's center pixel C, given different circles of confusion.

The second factor affecting the filter weight is the light intensity function $I(r_p)$. In reality, the light intensity function is too complex to make its calculation in a pixel shader practical, and based on [Che87], we use a simple uniform spread of intensity across the circle of confusion, falling off simply in proportion to the reciprocal of the square of the radius

$$I(r_p) \propto \frac{1}{r_p^2}.$$

The third factor affecting the filter weight is the intensity leakage control function $L(z_p)$. As with most post-processing techniques, a filter as used above will suffer from intensity leakage. Leakage occurs when a background object, which is out of focus, is blurred over a foreground object that is in focus. This is an undesirable effect, particularly since it is usually the in-focus foreground object that is the subject of the user's attention. We introduce a factor into the filter sample weights 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. Thus, $L(z_p)$ is defined as follows:

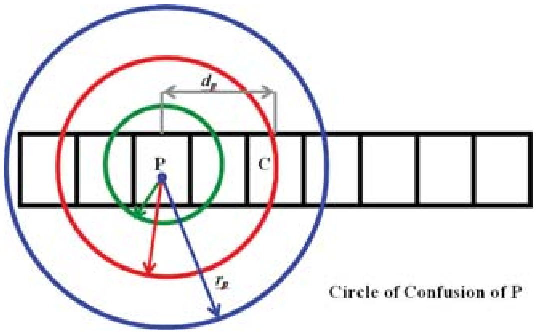

Figure 2: *The green CoC does not overlap the center pixel, while the red CoC has a partial overlap, and the blue CoC has a total overlap.*

$$L(z_p) = \begin{cases} \propto r_c, & z_p > z_f \\ 1 & z_p \leq z_f \end{cases},$$

where z_p is the scene depth of the sample pixel, z_f is the scene depth of the focal plane and r_c is the size of the circle of confusion of the center pixel, measured in pixel widths. This means that a background pixel's contribution to the blurring of a foreground pixel is reduce to zero as the foreground pixel comes into focus. In the reverse situation, a foreground pixel will continue to contribute to a background pixel that is in focus.

The final weight assigned to a sample pixel is the product of these three functions

$$W(P) = O(r_p) \times I(r_p) \times L(z_p).$$

The weight assigned to the center pixel is simply the light intensity function

$$W(C) = I(r_c).$$

The filter blends the pixel colors according to their respective weights, then renormalizes the result back into the range expected by the hardware and outputs the resulting color to the frame buffer.

5. Implementation

The basis for our implementation was a custom hierarchical scene management and OpenGL-based rendering engine developed by the authors, with a simple GLUT application framework. Since our method only requires the frame buffer and depth buffer as input, absolutely no changes were made to the rendering engine or scene manager. Instead, the post-processing implementation was integrated by inserting a single function call immediately prior to the *glutSwapBuffers* call in the application framework.

The first step is to render the scene into the frame buffer, which is done by the application as before. Once the scene rendering is complete, the application calls our function, and our post-processing code is executed. The frame buffer and depth buffer are then copied to textures using OpenGL's copy-to-texture functionality. Both of these textures were configured with no filtering enabled, since our algorithm blends the frame buffer image, and we require accurate depth values from the depth texture. Additionally, 'clamp to edge' was used on both textures so that the border pixels would be blended with themselves.

Next, the depth conversion parameters P_{33} and P_{34} are computed based on the camera properties. The depth of field processing is then initiated by drawing a screen-aligned polygon, textured with the frame and depth textures, and using a custom fragment shader written in the OpenGL Shading Language. The depth conversion parameters are passed to the fragment shader as uniform variables, along with the desired lens properties. The fragment shader performs depth and frame texture lookups on the pixel elements in a vertical filter, computing the appropriate pixel weights based on the computed circles of confusion, and then writes the blended color fragment to the frame buffer. The pseudo code for the fragment shader is as follows;

```
get center pixel depth
compute center pixel CoC
compute center pixel intensity weight
get center pixel color
for each sample pixel
{
    get sample pixel depth
    compute sample pixel CoC
    get sample pixel color
    compute sample pixel overlap
    weight O(r)
    compute sample pixel intensity
    weight I(r)
    compute sample pixel leakage
    weight L(z)
    compute sample pixel final weight
    add weighted sample pixel color
    to center pixel
}
re-normalize center pixel color
output center pixel color
```

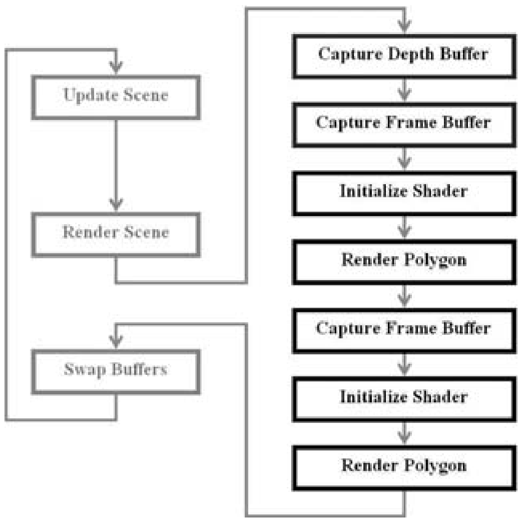

Figure 3: *Integration into rendering pipeline.*

Table 1: *Performance of Our Algorithm for Cloisters Scene (time in milliseconds)*

Resolution	Frame time (no DOF)	Frame time (DOF)	DOF cost
512 × 512	2.3	8.8	6.5
1024 × 512	2.7	15.8	13.1
1024 × 1024	3.1	28.5	25.5

Table 2: *Performance of Our Algorithm for Playroom Scene (time in milliseconds)*

Resolution	Frame time (no DOF)	Frame time (DOF)	DOF cost
512 × 512	4.5	10.9	6.4
1024 × 512	4.7	17.7	13.0
1024 × 1024	5	30.1	25.1

The frame buffer is then copied back to the texture, and a second screen-aligned polygon is drawn to initiate the second post-processing pass. This time, the fragment shader performs the horizontal filtering, computing the appropriate pixel weights, before writing the final result into the frame buffer. The post-processing code then exits, and the *glutSwapBuffers* function is called by the application to display the final image onto the screen. Figure 3 shows the steps in our post-processing implementation, and how it integrates into the rendering application.

Table 3: *Performance of Our Algorithm for Factory Scene (time in milliseconds)*

Resolution	Frame time (no DOF)	Frame time (DOF)	DOF cost
512 × 512	11.1	17.3	6.2
1024 × 512	11.9	24.6	12.7
1024 × 1024	12.7	37.9	25.2

Some effort was expended to optimize the fragment shader to achieve the frame rates needed for interactive applications. Initial test on the graphics hardware indicated that the shader performance was negatively affected by the size and complexity of the fragment shader code, and in particular, the branching statements needed to control the pixel weights. A number of steps were taken to solve this problem.

- Any per-frame computations was moved out of the shader and onto the CPU, the results being passed through to the shader via uniform variables. At the same time, the circle of confusion and depth value computations were combined and scaled to pixels, significantly reducing the per pixel computations.
- Conditional code was replaced with GPU functions such as ‘step’, to zero out the weights of pixels that should not contribute, and ‘smooth-step’ to calculate a circle of confusion overlap. This meant that although the shader would end up performing redundant texture lookups and blending, this was still significantly faster than using conditional branches to avoid performing the work at all.
- Advantage was taken of the vector-processing capabilities of the GPU, allowing the circle of confusion and weighting values for up to four pixels to be computed in parallel.
- Texture coordinate computations were paired up, so that a single texture coordinate vector, consisting of four floating point values (x, y, z, w), was used to store a symmetric pair of (x, y) texture coordinates. Additionally, these were moved into a vertex shader, thereby offloading the interpolation of these values onto the fixed functionality pipeline—the GPU equivalent of a free lunch.

Initial attempts to use an alpha channel to store depth information were unsuccessful. We had planned to output depth information computed on the first filter pass into the alpha channel, thus avoiding depth texture lookups on the second filter pass. However, testing showed that an 8-bit alpha channel could not provide sufficient resolution for our needs, resulting in noticeable popping, or stepping, as the focal plane was moved. A 16-bit alpha buffer would probably have resolved this issue, but we did not wish to place any restriction on how the initial render context was configured, so this idea was not pursued further.

6. Results

The algorithms in this paper have been implemented on a 2.8 GHz Pentium IV platform, with 1 GB RAM and an nVIDIA 6800 graphics card, running Linux.

Three test scenes were chosen, representing indoor and outdoor scenes of varying complexity. The tests were executed at three different resolutions, 512 × 512 pixels,1024 × 512 pixels and 1024 × 1024 pixels. Our results are summarized in tables 1–3.

Initial tests with our chosen test scenes showed that our method produced a high-quality depth of field effect, with no leakage or obvious aliasing artifacts. The first scene, Figure 4, ‘Cloisters’ consisted of 81,000 triangles, heavily textured. All times are averaged over 100 frames and measured in milliseconds. The second scene, Figure 5, ‘Playroom’ consisted of 120,000 triangles, with textures limited to the floor and jack-in-the-box. The third scene, Figure 6, ‘Factory’ consisted of 419,000 triangles, no textures. Figure 7 shows a small portion of the playroom scene that has been enlarged. The first picture has no depth of field, while the second picture has depth of field applied, with the focus on the handlebar of the tricycle. This shows that in-focus objects remain sharp, and do not leak onto out-of-focus background objects.

As can be seen from the results, the post-processing overhead is independent of scene complexity, and scales with the resolution. Tests indicated that in all cases, the texture lookups account for approximately 80% of the total depth of field cost.

Further tests were conducted with lens properties chosen to increase the filter width. However, significant performance degradation was observed when the filter width exceeded nine pixels. At 11 pixels, the depth of field cost increases by 45% and at 13 pixels, the cost increased by 70% over a nine-pixel filter width. For a 1024 × 1024 image size, the depth of field cost was approximately 42.5 milliseconds for a 13-pixel filter width. Whether or not this will be a problem depends on a number of factors, including the simulated lens diameter and focal length, and the screen resolution. Ultimately, a trade off between filter size and performance may be necessary.

One possible problem with our algorithm is related to the fact that our filter is not truly separable due to the dynamically adjusted weights. This can cause a reduction in blurring when multiple depth discontinuities are encountered in close proximity, and in a pathological case, could cause a pixel not to be blurred at all. In practice, this turned out not to be a problem since the conditions are unlikely to occur, and when they do occur, the color and depth discontinuities make it difficult for the human eye to discern a lack of blending over such small regions.

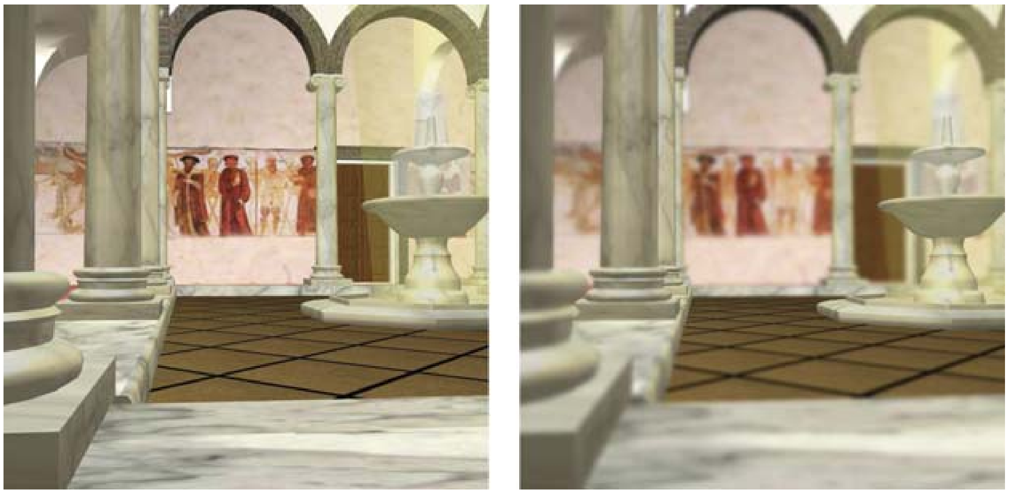
Figure 4: *Cloisters: Left image, no depth of field. Right image, depth of field with focus on the fountain.*

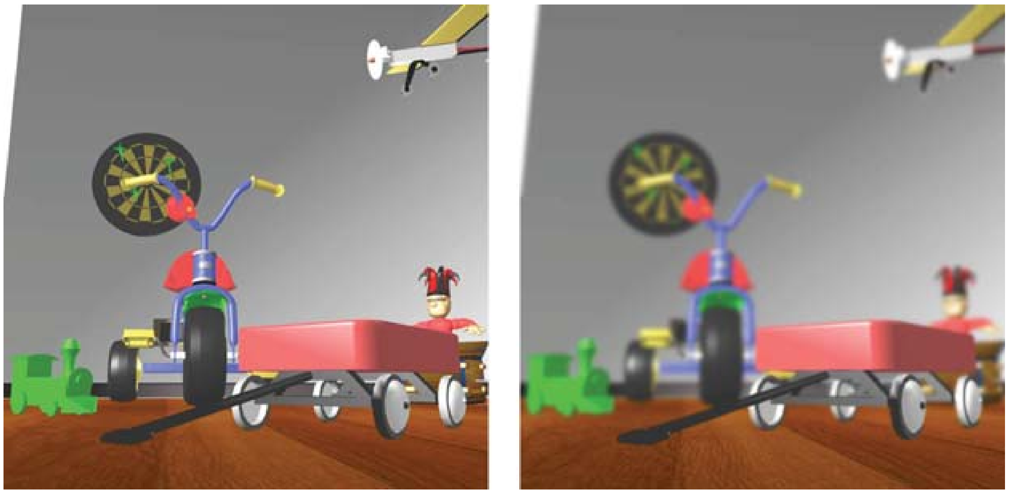
Figure 5: *Playroom: Left image, no depth of field. Right image, depth of field with focus on foreground.*

7. Conclusion and Future Work

In this paper we presented a new post-processing method for producing a realistic depth of field effect. Our contribution has the following attributes:

- It is based on accurate computation of circles of confusion using existing depth buffer information.
- A filtering algorithm ideally suited to a GPU implementation, achieving real-time frame rates.
- It requires no special rendering passes, making it extremely simple to integrate into existing rendering applications.

Our implementation currently has some limitations, in particular, the performance constraints based on the filter size. This generally only becomes a problem for scenes with a large scene depth, for example outdoor scenes, where the accuracy of the depth of field simulation has to be traded off for performance. It is expected that as the performance of modern graphics cards continues to improve, this will become less of a problem.

In future work, we plan to look at problems related to transparency, as well as some of the more advanced hardware features that are being standardized, such as render to texture, which may afford further performance benefits.

© 2007 The Authors
Journal compilation © 2007 The Eurographics Association and Blackwell Publishing Ltd.

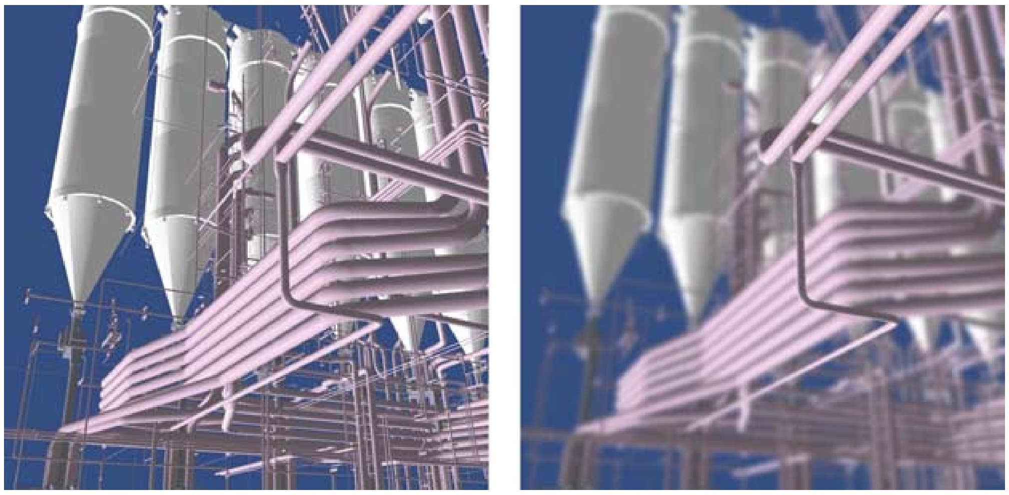

Figure 6: *Factory: Left image, no depth of field. Right image, depth of field, with focus on foreground pipes.*

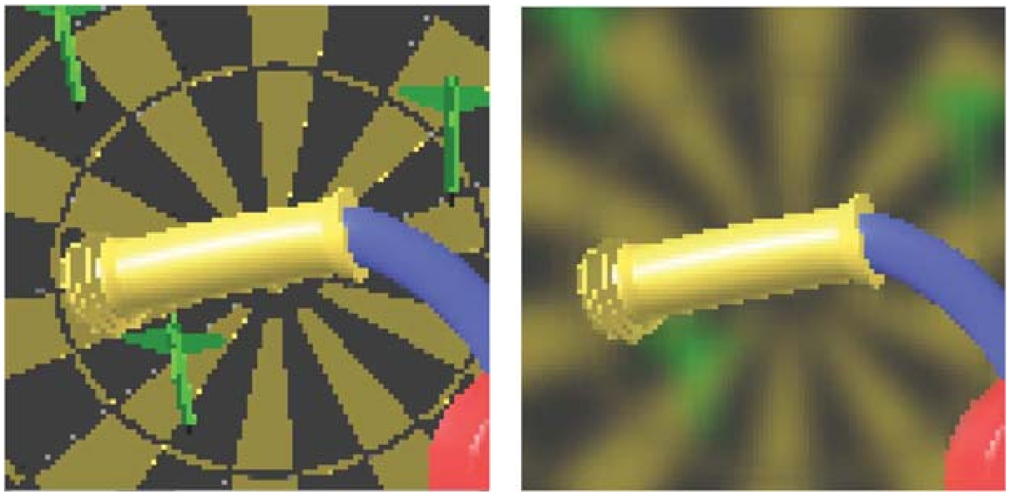

Figure 7: *Intensity leakage: Left image, no depth of field. Right image, depth of field, no intensity leakage.*

References

[AW02] ARCE T., WOLKA M.: *In-Game Special Effects and Lighting*, 2002. http://www.nvidia.com/object/gdc_in_game_special_effects.html.

[BFS04] BERTALMIO M., FORT P., SANCHEZ-CRESPO D.: Real-time, accurate depth of field using anisotropic diffusion and programmable graphics cards. In *Proceedings of Second International Symposium on 3D data Processing, Visualization and Transmission 2004*, pp. 767–773, 2004.

[Che03] CHEN J. X.: *Guide to Graphics Software Tools*. Springer Verlag, 2003.

[Che87] CHEN Y. C.: Lens effect on synthetic image generation based on light particle theory. In *CG International '87 on Computer graphics 1987*, pp. 347–366, 1987.

[Coo86] COOK R.: Stochastic sampling in computer graphics. *ACM Transactions Graphics* 5(1): 51–72, 1986.

[CPC84] COOK R., PORTER T., CARPENTER L.: Distributed ray tracing. *ACM SIGGRAPH Computer Graphics* 18(3): 137–145, 1984.

[Dem03] DEMERS J.: *Depth of Field in the 'Toys' Demo*, 2003. http://developer.nvidia.com/docs/IO/8230/GDC2003_Demos.pdf.

[Fea96] FEARING P.: Importance ordering for real-time depth of field. In *Proceedings of the Third International Conference on Computer Science*, pp. 372–380, 1996.

[HA90] HAEBERLI P., AKELEY K.: The accumulation buffer: hardware support for high-quality rendering. *Computer Graphics* 24(4): 309–318, 1990.

[HSS97] HEIDRICH W., SLUSALLEK P., SEIDEL H.: An image-based model for realistic lens systems in interactive computer graphics. In *Proceedings of Graphics Interface*, pp. 68–75, 1997.

[KMH01] KOSARA R., MIKSCH S., HAUSER H.: Semantic depth of field. In *Proceedings of the 2001 IEEE Symposium on Information Visualization(InfoVis) 2001*, pp. 97–104, 2001.

[KMH95] KOLB C., MITCHELL D., HANRAHAN P.: A realistic camera model for computer graphics. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques 1995*, pp. 317–324, 1995.

[KZ03] KRIVANEK J., ZARA J.: Fast depth-of-field rendering with surface splatting. In *Proceedings of Computer Graphics International*, pp. 196–201, 2003.

[ML00] MULDER J., LIERE R.: Fast perception-based depth of field rendering. In *Proceedings of the ACM symposim on Virtual Reality software and technology*, pp. 129–133, 2000.

[PC82] POTMESIL M., CHAKRAVARTY L.: Synthetic image generation with a lens and aperture camera model. *ACM Transactions on Graphics* 1(2): 85–108, 1982.

[Rok96] ROKITA P.: Generating depth-of-field effects in virtual reality applications. *IEEE Computer Graphics and Applications* 16(2): 18–21, 1996.

[RTI02] RIGUER G., TATARCHUK N., ISIDORO J.: Real-time depth of field simulation, 2002. http://www.ati.com/developer/shaderx/ShaderX2_Real-TimeDepthOfFieldSimulation.pdf.

[Sco92] SCOfiELD C.: $2\frac{1}{2}$d depth of field simulation for computer animantion. In D. Kirk, Editor, *Graphics Gems III*, Morgan Kaufmann, pp. 36–38, 1992.

[Shi94] SHINYA M.: Post-filtering for depth of field simulation with ray distribution buffer. In *Proceedings Graphics Interface'94*, pp. 59–66, 1994.

[WND97] WOO M., NEIDER J., DAVIS T.: *OpenGL Programming Guide* (*Third Edition*). Addison Wesley, Boston, Massachusetts, 1997.

[Woo05] WOOLLEY C.: GPU Program Optimization. In M. Pharr, Editor, *GPU Gems 2*. Addison-Wesley Professional, pp. 557–571, 2005.

© 2007 The Authors
Journal compilation © 2007 The Eurographics Association and Blackwell Publishing Ltd.