A Layered Depth-of-Field Method for Solving Partial Occlusion

David C. Schedl[*]
david.schedl@cg.tuwien.ac.at

Michael Wimmer[*]
wimmer@cg.tuwien.ac.at

[*]Vienna University of Technology, Austria

ABSTRACT

Depth of field (DoF) represents a distance range around a focal plane, where objects on an image are crisp. DoF is one of the effects which significantly contributes to the photorealism of images and therefore is often simulated in rendered images. Various methods for simulating DoF have been proposed so far, but little tackle the issue of partial occlusion: Blurry objects near the camera are semi-transparent and result in partially visible background objects. This effect is strongly apparent in miniature and macro photography. In this work a DoF method is presented which simulates partial occlusion. The contribution of this work is a layered method where the scene is rendered into layers. Blurring is done efficiently with recursive Gaussian filters. Due to the usage of Gaussian filters big artifact-free blurring radii can be simulated at reasonable costs.

Keywords:
depth of field, rendering, real-time, layers, post-processing

1 INTRODUCTION

DoF represents a distance range around a focal plane in optic systems, such as camera lenses. Objects out of this range appear to be blurred compared to sharp objects in focus. This effect emphasizes objects in focus and therefore is an important artistic tool in pictures and videos.

People in the field of computer graphics aim for the ambitious goal of generating photo-realistic renderings. Depth of Field is one effect which significantly contributes to the photorealism of images because it is an effect that occurs in most optical systems. In computer renderings, the pinhole-camera model, which relies upon the assumption that all light-rays travel through one point before hitting the image plane, is used. Therefore, there is no focus range and no smearing occurs, resulting in a crisp image. However, in real-life optical systems—such as the eye or photographic cameras—sharp images are only produced if the viewed object is within a certain depth range: the depth of field.

DoF can be simulated very accurately by ray tracing, but the rendering of accurate DoF effects is far from interactive frame rates. For interactive applications, the effect has to be approximated in real time. Therefore, most approaches use fast post-processing techniques and sacrifice visual quality, causing artifacts. Common techniques to produce the DoF effect use an approach where pixels get smeared according to their circle of confusion (CoC) [25]. The CoC depends on the distance of objects and the lens parameters. One artifact in post-processing approaches is *partial occlusion*: An object in-focus occluded by an out-of-focus object should be partly visible at the blurred object borders of the front object. In computer graphics, the used pinhole camera model in combination with depth testing leads to a dismissing of background pixels. Real optical systems use a finite aperture camera model where light rays from occluded objects can hit the image sensor. Figure 1 shows this effect next to a pinhole rendering.

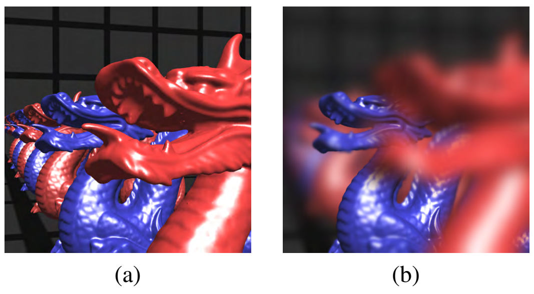
(a) (b)

Figure 1: A pinhole rendering of the scene *Dragons* resulting in a crisp image (a). Simulating shallow depth-of-field with the proposed method partly reveals occluded scene content (b). Note how the tongue of the dragon almost vanishes.

In this paper, we present an approach to tackling the partial occlusion problem. By rendering the scene with depth peeling [11], occluded pixels can be retrieved (Section 3.1). This occluded scene information is used

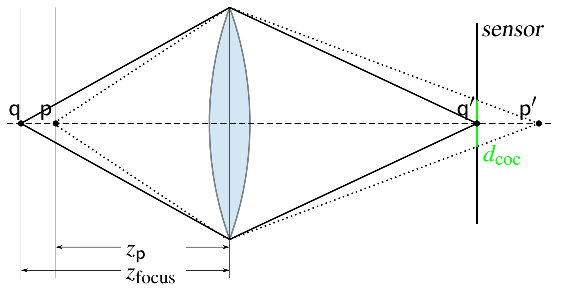

Figure 2: The model of a thin lens and how the points q (in-focus) and p (out-of-focus) are projected onto the image sensor (inspired by [25]).

to overcome the problem of partial occlusion. Rendered fragments are weighted, based on their depth, into layers (section 3.2), where each layer is blurred uniformly (section 3.3). Previous such layered DoF methods produced artifacts due to the layer splitting. We avoid most of these artifacts by smoothly decomposing layers and additional scene information. After blurring, the layers are composed by blending, thus producing renderings with convincing partial occlusion effects.

2 PREVIOUS WORK

DoF is an effect caused by the fact that optical lenses in camera systems refract light rays onto the image sensor, but fail to produce crisp projections for all rays. Figure 2 shows a schematics of a thin lens model and how rays are refracted. Although modern optical systems use a set of lenses, for the purpose of explaining DoF, a single lens is sufficient. Hypothetically, a sharp image point will only appear on the image plane from an object exactly in focus, located at z_{focus} (see figure 2). In practice, because of limitations of the human eye, objects within an acceptable sharpness are recognized as sharp. Objects out of the DoF range are projected as circles on the image plane. The diameter of this so-called circle of confusion (CoC) can be calculated as

$$d_{\text{coc}}(z, f, N, z_{\text{focus}}) = \left| \frac{f^2 \, (z - z_{\text{focus}})}{zN \, (z_{\text{focus}} - f)} \right|, \qquad (1)$$

where z is the distance to the object in front of the lens, f is the focal length of the lens, N is the f-stop number, and z_{focus} is the focus distance [25].

While the blurring is produced as an imperfection in optical systems, computer renderings usually produce a crisp image. Therefore DoF has to be simulated by specific rendering methods [1, 2, 10, 3].

Methods operating in object space simulate rays that do not go through the center of the lens. These methods include distributed ray tracing [8] and the accumulation buffer method [12], which both produce high-quality results but fail to deliver real-time frame rates.

Faster methods are based on the idea of rendering the scene with a pinhole camera model and simulating the DoF effect via post processing, leading to few or no changes in the rendering pipeline. The first method discussed by Potmesil and Chakravarty in 1981 presented equation 1, the formula for calculating the CoC [25]. Most modern methods (including this one) are based on this work.

Methods using graphic cards for acceleration use pyramid methods, Poisson sampling or a combination of both [24, 26, 27, 13, 22]. Isotropic filters lead to intensity leaking artifacts, where colors from in-focus foreground pixel bleed on the out-of-focus background. Cross-bilateral filters or heat diffusion ([6, 14]) can be used to overcome this artifact, but this introduces other issues like discontinuity artifacts: Out-of-focus objects have sharp boundaries although the object itself is blurred.

The partial occlusion artifact is apparent in all previously mentioned methods. Rasterization techniques do not store occluded fragments, therefore it is not possible to accurately simulate transparency caused by out-of-focus smearing. To fully simulate this effect, occluded information has to be either stored or interpolated in layers. Layered DoF methods compose scene fragments into layers depending on fragment depth. With this representation it is possible to store or interpolate occluded scene content. Furthermore it is possible to uniformly blur each layer. One prominent artifact in layered DoF methods are discretization artifacts: Layers get blurred and therefore object borders are smeared out. When this smeared-out layer is blended with the other layers, the smeared border region appears as a ringing artifact at object borders due to the reduced opacity. In [4, 5], the authors investigate such artifacts. One way to avoid these ringing artifacts is presented in [18], where occluded scene information is interpolated before layers are created. Blurring and interpolation is done by a pyramidal method, which approximates a Gaussian filter. The author presents a variation of this method in [17], where the costly interpolation steps are left out and different filters are used. However, these methods fail to correctly solve the partial occlusion problem, because hidden scene information is only interpolated and does not represent any actual scene content.

The DoF methods [21, 15] are able to solve partial occlusion by generating CoC-sized splats for each pixel. However, these methods come with additional costs for sorting fragments, making them impractical for complex scenes.

In [19], layered rendering is used to generate a layered representation. The DoF effect is then generated by ray-traversing these layers. Rays are scattered across the aperture of a virtual lens, thus avoiding the previously mentioned discretization artifacts. An improvement is discussed in [20], where the layers are generated by depth peeling and ray-tracing is done differently. Furthermore various lens effects (e.g., chromatic aberration and lens distortion) can be simulated. However, the method needs preprocessing previously to ray intersecting and needs back faces to be rendered. If the number of rays is not sufficient both methods produces noise and aliasing artifacts. Especially for strong blurs many rays have to be used, resulting in non-interactive rates.

For a solid approximation of partial occlusion, a layered scene representation, storing occluded fragments, has to be used. The approach presented in the following section is a layered DoF method which produces convincing partial occlusion effects while vastly avoiding the discussed issues.

3 METHOD

The method proposed in this paper decomposes the scene into *depth layers*, where each layer contains pixels of a certain depth range. The resulting layers are then blurred with a filter that is sized according to the distance from the focal point, and then composited. This approach handles partial occlusion, because hidden objects are represented in more distant layers and contribute to the compositing.

One way to generate the K layers would be to render the scene K times, with near- and far planes adjusted to cover the desired depth range of the layer. However, this leads to two problems: first, rendering the scene K times is too expensive for interactive applications, and second, discretization artifacts would appear due to the hard layer borders. In this paper, we solve both problems:

In order to avoid rendering the scene K times, we use depth peeling to generate a number M of *occlusion layers* (also named *buffers* in the following), where $M < K$. Note that each occlusion layer can contain fragments from the full depth range of the scene, while a depth layer is bound by its associated depth range. We then generate the depth layers by decomposing the occlusion layers into the depth ranges, which is much faster than rendering each depth layer separately.

To avoid discretization artifacts, we do not use hard boundaries for each depth layer, but a smooth transition between the layers, given by *matting functions*.

Furthermore, we also propose a method for efficiently computing both the blur and the layer composition in one step.

Our method consists of the following steps:

1. Render the scene into M buffers, where I_0 and Z_0 contain the color and depth from an initial pinhole rendering. The buffers $I_1 \dots I_{M-1}$ and $Z_1 \dots Z_{M-1}$ store peeled fragments from front to back.

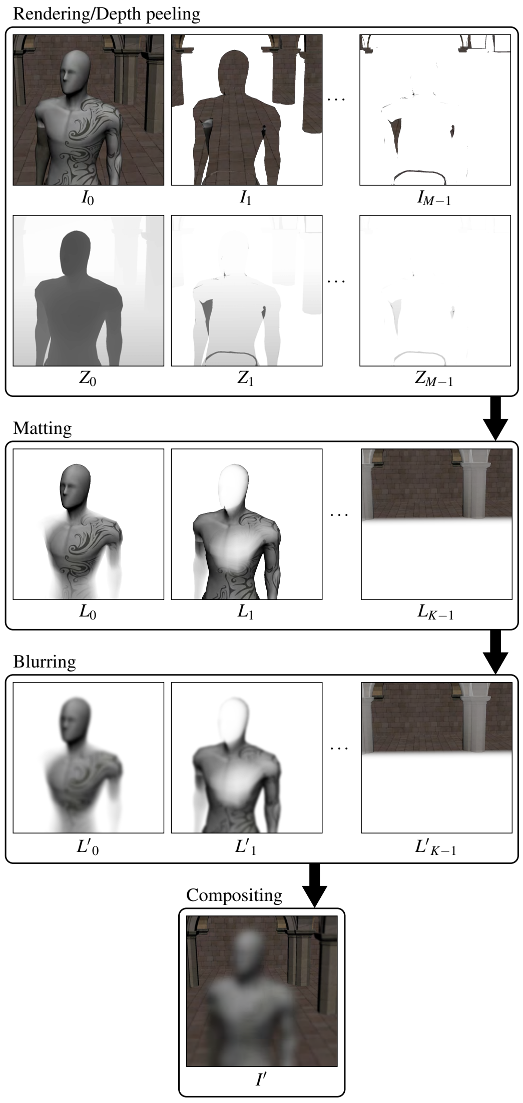

Figure 3: A overview of the proposed method in this work: The scene is rendered into color buffers I_0 to I_{M-1} and depth buffers Z_0 to Z_{M-1} by depth peeling . The color buffers are decomposed into K layers L_0 to L_{K-1} by a depth-dependent matting function. The decomposed layers get blurred by their CoC and composed. Note that the final algorithm combines the blurring and composing step, which is simplified in this figure.

2. Decompose the fragments of the input buffers into K depth layers L_0 to L_{K-1}, based on a matting function and the fragments' depth.

3. Blend and blur each layer L_k onto the buffers I'_{front}, I'_{focus} or I'_{back}, which are composed back-to-front afterwards.

Figure 3 outlines the above described algorithm.

3.1 Rendering

Rendering is done by depth peeling [11]. For depth peeling, first a 3D scene is rendered into a buffer storing the color I_0 and the depth Z_0 of a rendering, shown in figure 3. Then the scene is rendered a second time into new buffers I_m and Z_m while projecting the previous depth buffer Z_{m-1} onto the scene. A fragment p gets rejected if its depth z_{p} has the same or smaller depth than the previously rendered fragment, stored in I_{m-1} and Z_{m-1}. This means that only previously occluded fragments are stored in I_m and Z_m. This is done iteratively until M layers are retrieved. If a fragment is rejected, it is “peeled away,” revealing objects behind the first layer. Although there are faster peeling methods (e.g., [23]), we rely on [11], because peeling can be done iteratively from front to back.

3.2 Scene decomposition

The input images $I_0 \dots I_{M-1}$ are decomposed into K layers $L_0 \dots L_{(K-1)}$ by matting functions $\omega(z)$ and $\dot{\omega}$:

$$L_k = \Big(I_0 \cdot \omega_k(Z_0)\Big) \oplus \Big(I_1 \cdot \dot{\omega}_k(Z_1)\Big) \dots \oplus \Big(I_{M-1} \cdot \dot{\omega}_k(Z_{M-1})\Big). \quad (2)$$

The functions $\omega_k(z)$ and $\dot{\omega}_k(z)$ denote the matting function for the layer L_k and $A \oplus B$ denotes alpha-blending A over B.

3.2.1 Matting functions

The matting function ω_k was introduced in [18] and guarantees a smooth transition of objects between layers, while $\dot{\omega}_k$ retains a hard cut at the back layer boundaries to avoid situations where background fragments would be blended over foreground layers. The formulas are

$$\dot{\omega}_k(z) = \begin{cases} \frac{z - z_{k-2}}{z_{k-1} - z_{k-2}} & \text{for} \quad z_{k-2} < z < z_{k-1}, \\ 1 & \text{for} \quad z_{k-1} \le z \le z_k, \\ 0 & \text{otherwise,} \end{cases} \quad (3)$$

and

$$\omega_k(z) = \begin{cases} \frac{z_k - z}{z_k - z_{k+1}} & \text{for} \quad z_k < z < z_{k+1}, \\ \dot{\omega}_k(z) & \text{otherwise,} \end{cases} \quad (4)$$

where z_{k-2} to z_{k+1} defines anchor points for the layer boundaries. A plot of the functions is shown in figure 4. Special care has to be taken when matting the front L_0 and back L_{K-1} layer, where the boundaries are set to $z_{-2} = z_{-1} = z_0 = -\infty$ and $z_{K-1} = z_K = \infty$, respectively.

3.2.2 Layer boundaries

The layer matting relies on anchor points. Similarly to [18], the boundaries are spaced according to the filter size of the blurring method (further explained in section 3.3). Potmesil’s formula for calculating the CoC (equation 1) can be rearranged to calculate a depth z based on a given CoC d. Since d_{coc} is non-injective, there are two possible results of this inversion:

$$d_{\mathrm{coc}}^{-1}(d) = \big(D_1(d), D_2(d)\big) \quad (5)$$

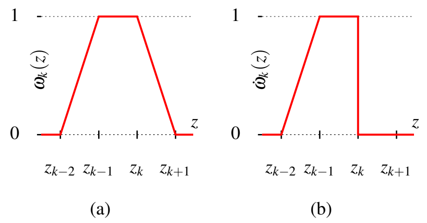

Figure 4: The matting functions ω_k (a) and $\dot{\omega}_k$ (b) with exemplary depth coordinates z_{k-2} to z_{k+1}.

with

$$D_1(d) = \frac{z_{\mathrm{focus}} \cdot f^2}{f^2 + d \cdot N \cdot (z_{\mathrm{focus}} - f)}, \quad (6)$$

$$D_2(d) = \frac{z_{\mathrm{focus}} \cdot f^2}{f^2 - d \cdot N \cdot (z_{\mathrm{focus}} - f)}. \quad (7)$$

With equation 5, the depth of anchor points can be calculated by using d_{coc}as input parameter, calculated by the filter size of the blurring method. Note that $D_2(d)$, $d \in \mathbb{R}^+$ is only applicable as long as

$$d < \frac{f^2}{N \cdot (z_{\mathrm{focus}} - f)}. \quad (8)$$

The anchor point furthest away from the camera, z_{K-1}, is limited by this constraint. An anchor pointz_k is placed at the average CoC of the layers L_k and L_{k+1}. Thus

$$z_k = \begin{cases} D_1\left(\frac{d_k + d_{k+1}}{2}\right) & \text{for} \quad k < k_{\mathrm{focus}}, \\ D_2\left(\frac{d_k + d_{k+1}}{2}\right) & \text{for} \quad k \ge k_{\mathrm{focus}}, \end{cases} \quad (9)$$

where k_{focus} is the index of the layer in focus and d_k and d_{k+1} are the CoCs of the layers L_k and L_{k+1} respectively. The layer’s CoC d_k is given by the blur radius for a discrete layer L_k, determined by the blurring method (see section 3.3).

3.2.3 Determining the number of layers

The depth of rendered fragments in the scene should lie within the depth range of the closest and furthest anchor points (z_{K-1} and z_0). Therefore enough anchor points to cover the scene have to be generated. This can be done manually or automatically. One naive automatic approach would be to use the near and far clipping planes, resulting in the highest possible depth range, which usually is not present in a scene. A better approach is to use hierarchical N-Buffers for determining the minimum and maximum depth values within the view frustum [9].

3.3 Blurring and Composition

We use Gaussian filters for blurring, because they can be separated, recursively applied and produce smooth results. The mapping from CoC to the standard deviation σ of a Gaussian kernel is chosen empirically as

$$d_{\mathrm{pix}} = 4\sigma. \quad (10)$$

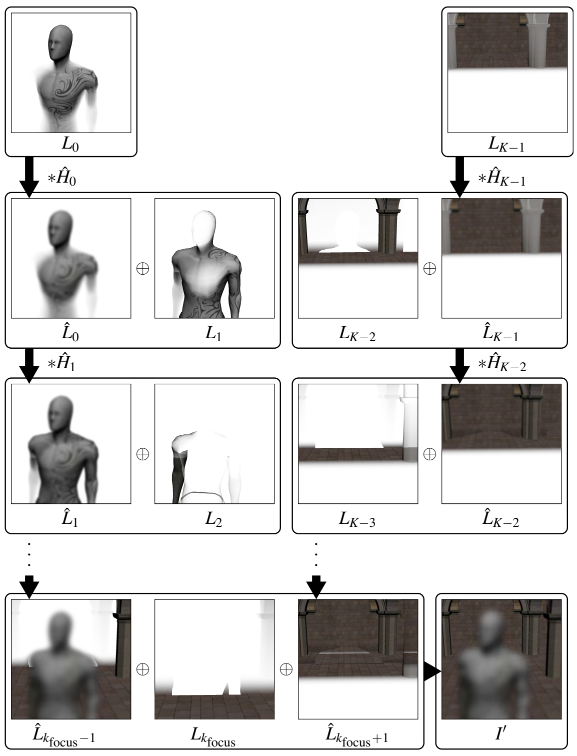

Figure 5: Overview of the combined blurring and composition steps: The layers L_0 to L_{K-1} are blended iteratively. Between each blending step the composition is blurred. Layers in front of the focus layer $L_{k_{\text{focus}}}$ and layers behind the focus layer are composed separately. Those are combined in a final step into the result I'.

Note that d_{pix} is the d_{coc} in screen coordinates and has to be transformed into the world coordinate system. Each layer is blurred by a Gaussian kernel H_k with the standard deviation σ_k as

$$L'_k = L_k * H_k, \tag{11}$$

where $*$ denotes a convolution.

Instead of convolving each layer separately (shown in figure 3 for illustrative reasons), a cascaded approach is chosen. Between each blur, one layer is blended onto the composition.

Layers in front and behind the in-focus layer have to be composed and blurred separately. Otherwise it is not possible to keep the correct depth ordering of the layers. The composition of the front layer starts by taking the layer closest to the camera (i.e., L_0) and blurring it with the filter kernel $\hat{H}_0$. In the next step this blurred layer is blended over the next closest layer (i.e., L_1) and afterwards blurred with $\hat{H}_1$. A schematic of the composition steps is shown in figure 5. Since a blurred layer L_k is blended over L_{k+1} and then blurred again, the effect of this method is that L_k is blurred by $\hat{H}_k$ and by $\hat{H}_{k+1}$. The iteration continues until the layer in-focus $L_{k_{\text{focus}}}$ is reached. In general, such recursive Gaussian filters produce the same result as blurring with one big Gaussian. The resulting filter sizes can be calculated by the Euclidean distance [7, chapter 8]. However, in our application the results differ due to occlusions within the layers.

Back layers are blurred similarly, starting with L_{K-1}. To keep the correct layer ordering, the layer closer to the camera (i.e., L_{K-2}) has to be blended over the previously blurred layer. The iteration is again continued until the layer in-focus is reached.

The number of blurring iterations for a layer L_k is given by $|k - k_{\text{focus}}|$. Calculating the final composition I' is done by

$$I' = \hat{L}_{k_{\text{focus}}-1} \oplus (L_{k_{\text{focus}}} \oplus \hat{L}_{k_{\text{focus}}+1}), \tag{12}$$

where

$$\hat{L}_k = \begin{cases} L_k & \text{for} \quad k = k_{\text{focus}} \\ L_k * \hat{H}_k & \text{for} \quad k = 0 \text{ and } k = K-1 \\ (\hat{L}_{k-1} \oplus L_k) * \hat{H}_k & \text{for} \quad k < k_{\text{focus}} \\ (L_k \oplus \hat{L}_{k+1}) * \hat{H}_k & \text{for} \quad k > k_{\text{focus}} \end{cases} \tag{13}$$

Results in section 4 are produced with a Gaussian filter kernel $\hat{H}_k$ with a standard deviation of $\hat{\sigma}_k$:

$$\hat{\sigma}_k = |k - k_{\text{focus}}|. \tag{14}$$

Various methods for calculating the filter size can be used. For Gaussians, the adequate (non-recursive) σ_k can be calculated by

$$\sigma_k = \begin{cases} 0 & \text{for} \quad k = k_{\text{focus}}, \\ \sqrt{\hat{\sigma}_k^2 + \sigma_{k+1}^2} & \text{for} \quad k < k_{\text{focus}}, \\ \sqrt{\hat{\sigma}_k^2 + \sigma_{k-1}^2} & \text{for} \quad k > k_{\text{focus}}, \end{cases} \tag{15}$$

where k is in the interval $[0, K-1]$.

3.3.1 *Normalization*

Due to the usage of matting functions ω and $\dot{\omega}$, resulting in expanded depth layers, and the usage of depth peeling, discretization artifacts as discussed in [5, 4] are mostly avoided. However, in some circumstances (e.g., almost perpendicular planes) such artifacts may still appear. We use premultiplied color values while matting and filtering. Therefore the composition can be normalized (divided by alpha), thus further minimizing discretization artifacts.

4 RESULTS

The proposed method was implemented in OpenGL and the shading language GLSL. Performance benchmarks are done on an Intel Core i7 920 CPU with a Geforce GTX 480 graphics card. Depth peeling uses a 32bit

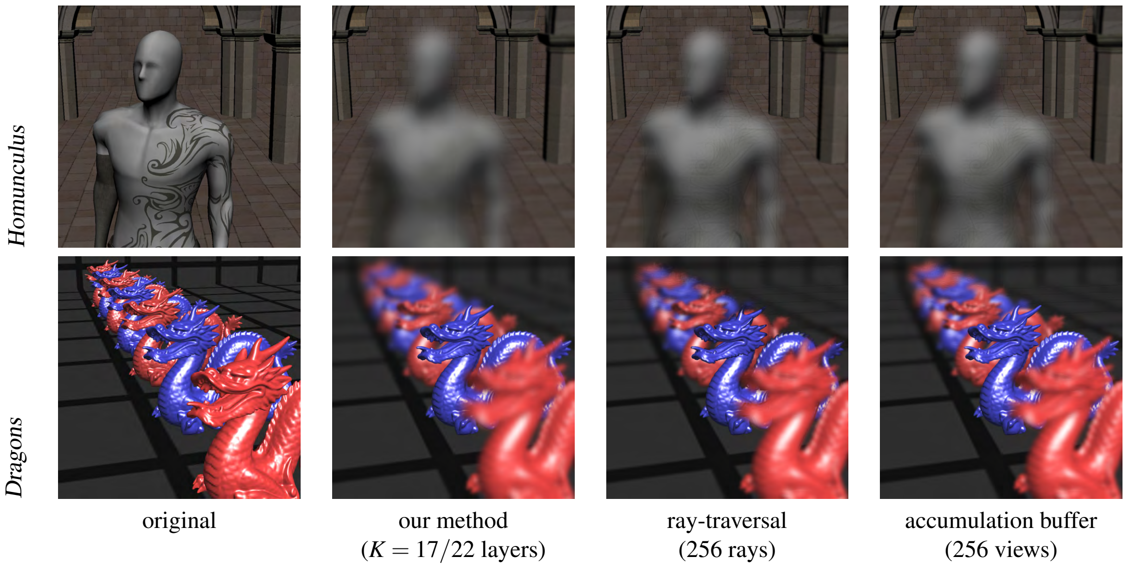

Figure 6: DoF effects produced with our method, the ray-traversal method ([20]) and the accumulation-buffer method. The first row shows the scene *Homunculus* (74k faces), and the second shows *Dragons* (610k faces). Renderings have the resolution 1024×1024. Note that there are sampling artifacts on the chest of the *Homunculus* scene in the accumulation and ray-traversal method, although there are 256 rays/views used. Our method avoids such artifacts by cascaded Gaussian filtering. Also note the partial-occlusion artifacts (e.g., in the second red dragon) in the ray-traversal method. The lens settings are $f = 0.1$, $N = 1.4$ and is focused at the stone wall in the back ($z_{\text{focus}} = 18.5$) for the *Homunculus* and at the foremost blue dragon ($z_{\text{focus}} = 3$) for the *Dragon* scene.

z-buffer to avoid any bias values due to z-buffer imprecisions.

We compare our method to the accumulation buffer-technique [12] and to a ray-traversal technique[20], because the first simulates high-quality DoF effects if enough views are sampled, while the latter method is a state-of-the-art method which handles partial occlusion correctly. The accumulation buffer technique is implemented in OpenGL, but does not use the accumulation buffer, because of precision issues when accumulating a high number of views. Instead, a 32bit-per-channel float texture is used for accumulation. The ray-traversal method was implemented in OpenGL and GLSL. Although there are some tweakable parameters, the authors give little information about their configuration. So for the intersection tests of rays, we use 100 steps for the linear search and 5 for the binary search in normalized device coordinates. A hierarchical N-Buffer is used to decrease the depth range for each rays. In our implementation, we decrease the depth range for each ray individually—while the original authors packed rays—and we use 4 regular depth-peeling layers, without any optimizations. Additionally, the ray-traversal method requires closed scene objects and back-face culling to be turned off, for reliable intersection testing. This introduces additional rendering costs and decreases the usable peeling layers to only 2. The authors propose ways to overcome the latter limitation. However, in our implementation we use the simplified version containing only two peeling layers and their backface counterparts. For both reference methods we use a Gaussian distribution for lens samples positioning.

The methods are applied to the scenes *Homunculus* (74k triangles) and *Dragons* (610k triangles), shown in figure 6. Performance comparisons are shown in table 1. Rendering costs for the accumulation-buffer method are basically the costs for one scene rendering multiplied by the number of views. Our method is, apart from depth peeling, independent of the scene complexity, and faster than the ray-traversal method, even when that method uses only 32 rays, resulting in sampling artifacts. Our method is faster at processing the *Dragons* scene, although the scene *Homunculus* has fewer triangles. This can be explained by the distribution of the depth layers and the resulting amount of blur. In the *Homunculus* scene there are more highly blurred foreground layers, resulting in overall more rendering costs than in *Dragons*, where the layers are more, but evenly spread. Note that although scene *Dragons* has more triangles than *Homunculus*, it is rendered faster due to shading without textures and the use of vertex-buffer objects.

We currently store all sub-images on the graphics card—for convenience and debug reasons—resulting in heavy memory usage. However, additional depth layers ($\hat{L}_0$ to $\hat{L}_{K-1}$) can be avoided by applying the process-queue (matting, blurring, composing) in one buffer, which would decrease memory consumption.

	our (DP/matting/blur) Total		(DP/ray-traversal) Total			accum.
	cascaded	non-cascaded	256	128	32 rays	256 views
Homunculus (74k tri.)	(46/5/51)102	(46/5/95)146	(58/1290)1348	(48/643)691	(48/140)188	4809
Dragons (610k tri.)	(40/7/51)98	(40/8/85)133	(69/1374)1443	(69/685)754	(59/152)211	4163

Table 1: Performance comparisons, in ms, of our method (cascaded and non-cascaded blurring) with the ray-traversal method ([20]) and the accumulation-buffer method for the scenes *Homunculus* and *Dragons*. Renderings have the resolution 1024×1024 and 4 Depth-peeling iterations (DP) have been used.

5 CONCLUSION AND FUTURE WORK

We have presented a depth-of-field post-processing method with the aim of overcoming the partial occlusion artifact. The contribution of this work is a simple, efficient and GPU-friendly method. We combine depth-peeling with improved matting functions to avoid the overhead of rendering to a high number of depth layers. Furthermore we use high-quality Gaussian filters in a recursive way, which has not been done—to our knowledge—in DoF methods before. With the usage of Gaussian filters, high blur radii can be simulated, where even the reference methods start to produce sampling artifacts. We have shown that those DoF effects can be produced at frame rates that are significantly higher than previous methods, making high-quality DoF available for interactive applications.

One important step for the correct handling of partial occlusion is depth peeling, which is frequently used to resolve transparency issues, thus making the method hardly usable for interactive applications like games.

Currently we use Gaussian filters, which are separable and can be computed efficiently while delivering artifacts-free images. The usage of cascaded filters while composing the DoF effect slightly alters the produced image, but results in better performance. If higher frame rates are needed and visual quality can be sacrificed faster blurring methods (e.g., box, pyramid filters) can be used.

The composition by alpha-blending is simple and efficient, thus leading to faster results when compared to current methods like [20]. Layering discretization artifacts known from other methods are mostly avoided by matting, depth peeling and normalization.

Wide-spread anti-aliasing methods (i.e., MSAA) cannot be easily enabled for our method. However, image-based anti-aliasing methods (such as MLAA or FXAA)—which are becoming more popular due to the wide usage of deferred shading—can be applied.

Currently, layers are split based on the d_{coc} of fragments and on the chosen blurring method. This might result in empty layers. Decomposition could be optimized by using clustering methods, such as k-means clustering, as proposed in [21, 16]. With the use of clustering, layer borders could be tailored to the pixel density in scenes and empty layers could be avoided. However, clustering is a costly process and therefore only applicable for off-line rendering.

One further field of investigation would be the impact of correct partial occlusion rendering on human perception. We think that a correct handling of partial occlusion in combination with gaze-dependent focusing (e.g., with an eye-tracker) would result in deeper immersion of the user.

6 ACKNOWLEDGMENTS

Thanks to Juergen Koller for providing the Homunculus model. The used Dragon and Sponza models are courtesy of Stanford Computer Graphics Laboratory and Marko Dabrovic.

7 REFERENCES

[1] Brian A. Barsky, Daniel R. Horn, Stanley A. Klein, Jeffrey A. Pang, and Meng Yu. Camera models and optical systems used in computer graphics: Part I, Object based techniques. Technical report, University of Berkeley, California, USA, 2003.

[2] Brian A. Barsky, Daniel R. Horn, Stanley A. Klein, Jeffrey A. Pang, and Meng Yu. Camera models and optical systems used in computer graphics: Part II, Image-based techniques. Technical report, University of Berkeley, California, USA, 2003.

[3] Brian A. Barsky and Todd J. Kosloff. Algorithms for rendering depth of field effects in computer graphics. In *Proceedings of the 12th WSEAS international conference on Computers*, pages 999–1010, Stevens Point, Wisconsin, USA, 2008. World Scientific and Engineering Academy and Society (WSEAS).

[4] Brian A. Barsky, Daniel R. Tobias, Michael J.and Horn, and Derrick P. Chu. Investigating occlusion and discretization problems in image space blurring techniques. In *First International Conference on Vision, Video and Graphics*, pages 97–102, University of Bath, UK, July 2003.

[5] Brian A. Barsky, Michael J. Tobias, Derrick P. Chu, and Daniel R. Horn. Elimination of artifacts due to occlusion and discretization problems in image space blurring techniques. *Graphics Models*, 67(6):584–599, November 2005.

[6] Marcelo Bertalmio, Pere Fort, and Daniel Sanchez-Crespo. Real-time accurate depth of field using anisotropic diffusion and programmable graphics cards. In *Proceedings of the 3D Data Processing, Visualization, and Transmission, 2nd International Symposium*, 3DPVT '04, pages 767–773, Washington, DC, USA, 2004.

[7] Wilhelm Burger and Mark J. Burge. Principles of Digital Image Processing: Advanced Techniques. To appear, 2011.

[8] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '84, pages 137–145, New York, NY, USA, 1984. ACM.

[9] Xavier Décoret. N-buffers for efficient depth map query. *Computer Graphics Forum*, 24(3), 2005.

[10] J. Demers. Depth of field: A survey of techniques. In Fernand Randima, editor, *GPU Gems*, chapter 23, pages 375–390. Pearson Education, 2004.

[11] C. Everitt. Interactive order-independent transparency. Technical report, NVIDIA, 2001.

[12] Paul Haeberli and Kurt Akeley. The accumulation buffer: hardware support for high-quality rendering. *SIGGRAPH Computer Graphics*, 24:309–318, September 1990.

[13] Earl J. Hammon. Practical post-process depth of field. In Hubert Nguyen, editor, *GPU Gems 3: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter 28, pages 583–606. Addison-Wesley, 2007.

[14] Michael Kass, Lefohn Aaron, and John Owens. Interactive depth of field using simulated diffusion on a GPU. Technical report, Pixar Animation Studios, 2006.

[15] Todd J. Kosloff, Michael W. Tao, and Brian A. Barsky. Depth of field postprocessing for layered scenes using constant-time rectangle spreading. In *Proceedings of Graphics Interface 2009*, pages 39–46, Toronto, Canada, 2009.

[16] Todd Jerome Kosloff. *Fast Image Filters for Depth of Field Post-Processing*. PhD thesis, EECS Department, University of California, Berkeley, May 2010.

[17] Martin Kraus. Using Opaque Image Blur for Real-Time Depth-of-Field Rendering. In *Proceedings of the International Conference on Computer Graphics Theory and Applications : GRAPP 2011*, pages 153–159, Portugal, 2011. Institute for Systems and Technologies of Information, Control and Communication.

[18] Martin Kraus and Magnus Strengert. Depth-of-field rendering by pyramidal image processing. *Computer Graphics Forum*, 26(3):645–654, 2007.

[19] Sungkil Lee, Elmar Eisemann, and Hans-Peter Seidel. Depth-of-field rendering with multiview synthesis. *ACM Transactions on Graphics (TOG)*, 28(5):1–6, 2009.

[20] Sungkil Lee, Elmar Eisemann, and Hans-Peter Seidel. Real-time lens blur effects and focus control. *ACM Transactions on Graphics (TOG)*, 29(4):65:1–65:7, July 2010.

[21] Sungkil Lee, Gerard Jounghyun Kim, and Seungmoon Choi. Real-time depth-of-field rendering using splatting on per-pixel layers. *Computer Graphics Forum (Proc. Pacific Graphics'08)*, 27(7):1955–1962, 2008.

[22] Sungkil Lee, Gerard Jounghyun Kim, and Seungmoon Choi. Real-time depth-of-field rendering using anisotropically filtered mipmap interpolation. *IEEE Transactions on Visualization and Computer Graphics*, 15(3):453–464, 2009.

[23] Fang Liu, Meng-Cheng Huang, Xue-Hui Liu, and En-Hua Wu. Single pass depth peeling via cuda rasterizer. In *SIGGRAPH 2009: Talks*, SIGGRAPH '09, New York, NY, USA, 2009. ACM.

[24] Jurriaan D. Mulder and Robert van Liere. Fast perception-based depth of field rendering. In *Proceedings of the ACM symposium on Virtual Reality Software and Technology*, VRST '00, pages 129–133, Seoul, Korea, October 2000. ACM.

[25] Michael Potmesil and Indranil Chakravarty. A lens and aperture camera model for synthetic image generation. In *Proceedings of the 8th Aannual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '81, pages 297–305, Dallas, Texas, USA, 1981. ACM.

[26] Guennadi Riguer. Real-time depth of field simulation. In Wolfgang F. Engel, editor, *ShaderX^2: Shader Programming Tips and Tricks with DirectX 9*, pages 529–556. Wordware Publishing, October 2003.

[27] Thorsten Scheuermann and Natalya Tatarchuk. Improved depth of field rendering. In Wolfgang Engel, editor, *ShaderX^3: Advanced Rendering Techniques in DirectX and OpenGL*, pages 363–378. Charles River Media, 2004.