DOI: 10.1111/j.1467-8659.2012.02097.x

COMPUTER GRAPHICS *forum*
Volume 0 (2012), number 0 pp. 1–11

Efficiently Simulating the Bokeh of Polygonal Apertures in a Post-Process Depth of Field Shader

L. McIntosh, B. E. Riecke and S. DiPaola

School of Interactive Arts and Technology, Simon Fraser University, Vancouver, Canada
{lmcintos, ber1, sdipaola}@sfu.ca

Abstract

The effect of aperture shape on an image, known in photography as 'bokeh', is an important characteristic of depth of field in real-world cameras. However, most real-time depth of field techniques produce Gaussian bokeh rather than the circular or polygonal bokeh that is almost universal in real-world cameras. 'Scattering' (i.e. point-splatting) techniques provide a flexible way to model any aperture shape, but tend to have prohibitively slow performance, and require geometry-shaders or significant engine changes to implement. This paper shows that simple post-process 'gathering' depth of field shaders can be easily extended to simulate certain bokeh effects. Specifically we show that it is possible to efficiently model the bokeh effects of square, hexagonal and octagonal apertures using a novel separable filtering approach. Performance data from a video game engine test demonstrates that our shaders attain much better frame rates than a naive non-separable approach.

Keywords: computer graphics, rendering, real-time, shaders, depth of field

ACM CCS: I.3.7[Computer Graphics]: Three-Dimensional Graphics and Realism: Colour; Shading; Shadowing and Texture

1. Introduction

In real-world cameras, precise focus can only be achieved for objects at a specific distance from the lens. Objects nearer to or farther from the lens than this distance are defocused and produce progressively larger blurred spots known as Circles of Confusion (CoC), on the camera's image sensor. This effect, called 'depth of field', provides cues about scene depth to viewers, and has become an important tool in cinematography where it is often used to draw attention to particular parts of a scene [Mon00].

As video games increase in narrative and graphical sophistication, many are attempting to emulate this cinematic effect in real time. Unfortunately, most real-time depth of field implementations fail to fully consider the effect of 'bokeh' on the image (see Figure 1). Bokeh refers to the appearance of the CoC in the defocued areas of an image, and is most clearly revealed by small bright defocused points of light. Many aspects of a camera's design contribute to bokeh, but the most significant aspect is usually the shape of the aperture [Mer97]. Lens effects like spherical and chromatic aberration also contribute significantly to bokeh, but in this paper we concern ourselves solely with the effect of aperture shape.

This paper presents a novel approach that applies separable filtering to the problem of efficiently simulating the bokeh of various aperture shapes. Section 2 reviews some previously published depth of field techniques, Section 3 presents a 'naive' non-separable approach to bokeh as motivation, Section 4 presents our separable bokeh technique and Section 5 presents the results of a performance test that compares the efficiency of our approach to the naive one.

2. Background and Related Work

Beginning with the work of Potmesil and Chakravarty [PC81], there has been substantial research into various techniques for adding depth of field effects to computer graphics. Demers [Dem04], organizes depth of field techniques into five categories:

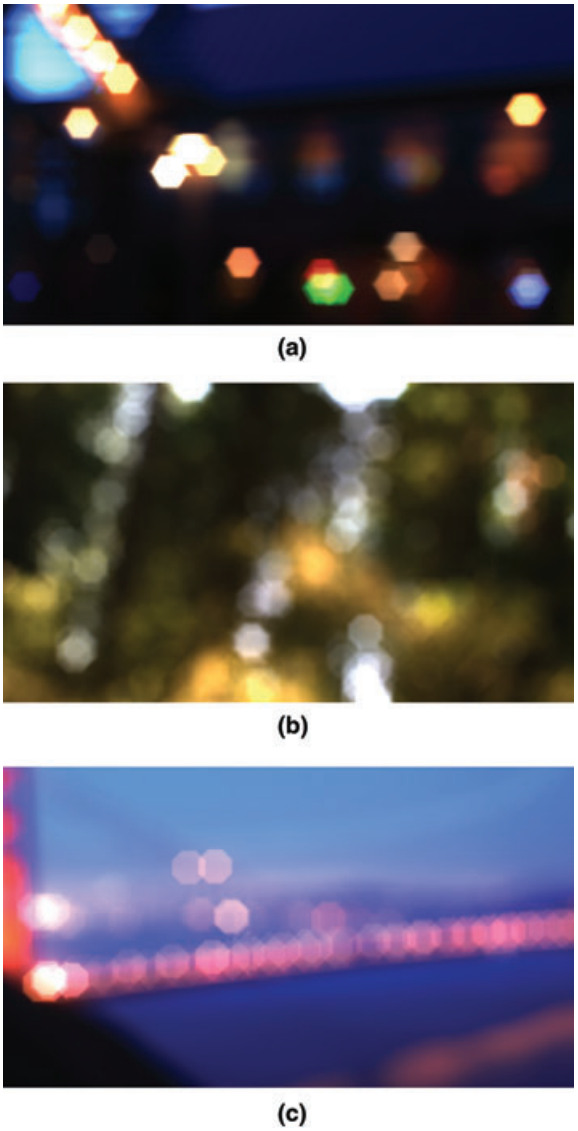

Figure 1: *These high dynamic range (HDR) photographs have been processed by our separable bokeh technique to simulate depth of field (a constant scene depth was assumed). (a) Demonstrates the bokeh of a hexagonal aperture, (b) and (c) simulate the bokeh of octagonal apertures. (HDR photos courtesy of Mark Fairchild: http://www.cis.rit.edu/fairchild/).*

- Distributing traced rays across the surface of a lens
- Rendering from multiple cameras (accumulation-buffer)
- Rendering and compositing multiple layers
- Forward-mapped (scattering) z-buffer techniques
- Reverse-mapped (gathering) z-buffer techniques

In general, ray tracing and accumulation-buffer techniques remain too computationally expensive to be executed with acceptable results in real time. Compositing techniques are feasible for real-time applications and are becoming increasingly popular in the computer graphics literature [KLO06, KS07, LKC08, LES09, LES10]. These approaches can produce quite realistic results, with proper partial occlusion and other difficult issues addressed. The layered rendering required by these techniques can require significant changes to existing graphics engines to implement, however, and for lower end hardware they remain computationally expensive. For these reasons, depth of field techniques for real-time applications remain largely centred around z-buffer (post-process) techniques.

Forward-mapped (scattering) z-buffer techniques—sometimes called 'point splatting' techniques—work by distributing each pixel's colour to neighbouring pixels. A colour map and a depth map (i.e. z-buffer) are produced for the scene, and the diameter of the CoC is calculated for each pixel based on the depth map and some focus parameters. Finally, each pixel is blended into the frame buffer as a circle (or any desired aperture shape) with a diameter equal to the CoC. Typically this is accomplished by rendering textured sprites, centred at the location of each original pixel. Using a textured sprite has the advantage that any desired aperture shape can be modelled by simply changing the texture used. Depth-sorting can be used to ensure each sprite only affects pixels that are farther from the camera than itself (to prevent blurry backgrounds from affecting focused foregrounds). Unfortunately, such forward-mapped techniques are not very amenable to execution in real time as they require a large number of sprites to be rendered every frame. In addition, they require the costly depth-sorting of these sprites [LKC08].

Reverse-mapped (gathering) z-buffer techniques are similar in concept to forward-mapped techniques, but with one significant difference. Rather than each pixel spreading its colour value to neighbouring pixels (by way of rendering a sprite), each pixel explicitly samples the colour values of its neighbours to determine its own value. This category of techniques is generally much better suited to execution on current graphics hardware, as it avoids the costly processing of millions of sprites and takes advantage of fast hardware texture lookups. For this reason, many modern video games implementing depth of field use a technique from this category (for example, Call of Duty 4: Modern Warfare (Infinity Ward, 2007) uses one [Ham07]). In general, realistic bokeh effects (like circular or polygonal apertures) are not efficiently modelled with reverse-mapped z-buffer techniques. Our approach, and a similar one developed concurrently at DICE [WB11] (published while this paper was in review), appear to be the first attempts to do so.

Riguer *et al.* [RTI03] and Scheuermann [Sch04] provide good introductions to the conventional reverse-mapped

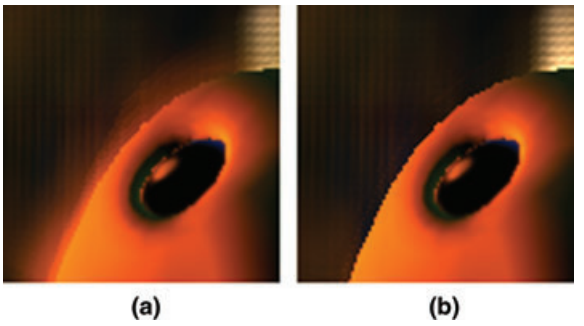

Figure 2: *In this comparison, intensity leakage or 'pixel bleeding' is visible in (a) around the focused foreground object. However, in (b) where an intensity leakage solution [RTI03] has been applied, the effect is greatly reduced.*

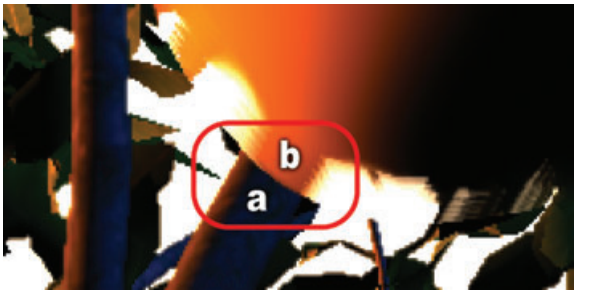

Figure 3: *In this image, produced by our separable bokeh technique, the defocused foreground object's silhouette is too sharp against the focused tree branch behind it. Some fading occurs inside the silhouette (b), but the silhouette will not realistically spread out over its borders onto the tree branch (a).*

z-buffer techniques that do not consider bokeh. Riguer *et al.* describe two separate reverse-mapped z-buffer techniques, and Scheuermann essentially combines them into a hybrid technique that overcomes some limitations of both. Using Scheuermann's technique, one starts by rendering a CoC map into the alpha channel of the colour map. This colour map is then downsampled to 1/4th of its original dimensions (1/16th the number of pixels), and a two-pass separable Gaussian blur is applied to it. In a final pass, the desired CoC is read from the alpha channel of the original colour map, and a filter with an 8-sample stochastic kernel (of the CoC size) is used to sample from both the original and the downsampled colour maps. These two samples are then linearly interpolated based on the CoC size. The result of this interpolation is that samples with a large CoC effectively come from the blurred colour map, although samples with a small CoC effectively come from the original colour map. The interpolated samples are then combined in a weighted average to determine the final value of each pixel.

2.1. Common post-processing artefacts

Scheuermann's depth of field technique, along with many similar reverse-mapped z-buffer techniques, suffers from a number of graphical artefacts that reduce the realism of the final image. Our technique specifically addresses one of these—lack of realistic bokeh—but we briefly discuss them all as our technique exhibits many of them.

The first artefact—and one that Riguer *et al.* [RTI03] specifically attempt to solve in their technique—is called **intensity leakage** (or pixel bleeding). Focused foreground objects appear to 'leak' onto blurry backgrounds (see Figure2), because the blurry pixels in the background have large CoC diameters and sample some part of their value from pixels of the focused foreground object. Riguer *et al.* and Scheuermann address this problem by carefully weighting each of the samples in the average. Samples that have a significantly smaller CoC (those that are more focused) and are closer to the camera than the centre sample are weighted much lower than normal. Our technique includes this solution.

Another artefact inherent to post-processing depth of field techniques is a **lack of partial occlusion**. In images produced by real cameras, the edges of blurry foreground objects spread out smoothly and appear semi-transparent over top of focused objects behind them. In the images produced by most post-process gathering depth of field techniques however, blurry foreground objects remain opaque, revealing no details behind them. Solutions to this problem typically involve rendering multiple layers. In general, post-process gathering techniques, including ours, do not attempt to solve this problem. In most applications the issue can be side-stepped by simply preventing foreground objects from coming too near the camera (and thus from becoming excessively blurry, revealing the artefact).

Another artefact is the appearance of **sharp silhouettes** on defocused foreground objects against focused backgrounds (see Figure3). This occurs because pixels from the focused background have small CoC and will not sample from the defocused foreground, and thus the defocused foreground will not appear to spread out over the focused background. Hammon [Ham07] proposes a simple solution to this which involves blurring the CoC map itself in a specific way. Our technique does not specifically address this problem, though Hammon's solution could easily be integrated.

The last common artefact, and the one we specifically attempt to address, is a **lack of realistic bokeh**—especially in high dynamic range (HDR) rendering where small bright defocused points of light tend to reveal the bokeh of the simulated camera. In real-world cameras, circular and polygonal apertures are nearly universal. However, in reverse-mapped

Figure 4: *(a) Shows a 2D 127-sample kernel that could be used to produce a hexagonal bokeh effect. (b) Shows a 2D 120-sample kernel that could be used to produce an octagonal bokeh effect. As we demonstrate, this approach suffers from poor performance because of the large number of samples necessary.*

z-buffer techniques the bokeh is most often Gaussian (as in [KLO06], [KS07] and [ZCP07]). In contrast, our separable technique efficiently approximates the bokeh of simple polygonal apertures—specifically, we demonstrate square, hexagonal and octagonal apertures, though many others are possible. Note that we do not attempt to model complex lens effects such as chromatic or spherical aberrations. For discussion of a ray tracing technique that can model these sort of bokeh effects see Wu *et al.*[WZH*10].

3. A Naive Approach

Before we describe our separable bokeh technique, we first describe a naive approach as motivation. The most obvious approach to creating bokeh effects in a gathering depth of field shader might be to filter the colour map with a carefully designed variable-size 2D kernel, similar to the 8-sample stochastic filter used by Scheuermann. Using many samples, and carefully arranging their positions in the kernel, various aperture shapes could be modelled. For example, to approximate the bokeh of hexagonal and octagonal apertures, 2D kernels like those in Figure4 could be used. This approach produces the desired bokeh effect, and has the advantage that any conceivable aperture shape can be approximated. Indeed, it would appear to be a perfect solution. As we will show, however, the performance of such a shader is poor, and attempting to increase performance using fewer samples leads inevitably to sampling artefacts.

4. A Separable Approach

Our technique leverages the efficiency of separable filters to achieve its performance gains. In a nutshell, we begin with a modified 'box-blur' filter, which can simulate the effect of any parallelogram-shaped aperture (for instance, a square aperture). To simulate more complex aperture shapes like hexagons and octagons, we then combine two or more images produced by different parallelogram-shaped apertures with an intersection-like (or union-like) operation. For instance, an octagonal aperture is simulated by combining the results of two square apertures, one aperture rotated 45 degrees from the other. A more thorough explanation of the technique follows. Please refer to the Appendix for a complete code listing.

4.1. Rendering the CoC map

Similar to the work by Scheuermann [Sch04], our separable bokeh technique starts by rendering a CoC map into the alpha channel of the colour map. Storing the CoC map in the alpha channel of the colour map allows for blurring the CoC map itself alongside the colour map (we explain the need for this in Section 4.4). A 16-bit render target is used to hold HDR lighting information (allowing for bright defocused highlights) in the RGB channels, as well as precise CoC data in the alpha channel. The CoC for each pixel is calculated by the following formula (from the well known thin-lens model), where c is the diameter of the CoC, A is the diameter of the aperture, f is the focal length, S_1 is the focal distance and S_2 is the given pixel's scene depth, (as read from a depth map):

$$c = A \cdot \frac{|S_2 - S_1|}{S_2} \cdot \frac{f}{S_1 - f}.$$

For storage in the alpha channel, the CoC diameter is then divided by the height of the virtual 'image sensor'. Any value can be used, but for realism we chose 0.024 m (24 mm) in our shaders—a value taken from the real-world 35 mm full frame format. We consider this length to represent the maximum CoC diameter that could reasonably be captured by our simulated camera. This division ensures more effective usage of our full 16-bit render target, and makes the calculations in later passes more efficient as the CoC diameter is now expressed as a percentage of the image size (the native format for texture coordinates in Direct3D and OpenGL). Finally, the return value is clamped between zero and an artificially imposed maximum percentage. This is done to prevent excessively large CoC from causing notable sampling artefacts.

4.2. Parallelogram-shaped apertures

Because all our aperture shapes are based on parallelograms, we begin with a discussion of them.

Simulating a parallelogram-shaped aperture is mostly a straightforward application of a 'box-blur' filter. A 1D uniform kernel is applied in each pass, between passes the kernel is rotated, and the output of the first pass is provided as input to the second pass to create a cumulative effect. The major differences from a separable box-blur filter are:

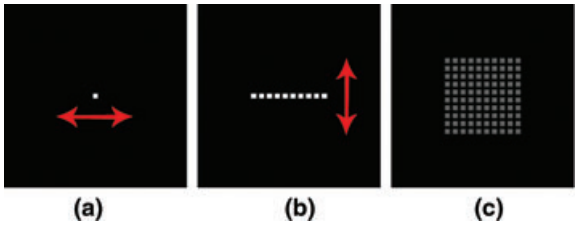

Figure 5: *In this series, a single bright test point is spread into a square over two passes. It shows, (a) the original image, (b) the result of the first pass and (c) the final result of the second pass. Arrows indicate the direction of the linear kernel applied in each pass to produce the next image.*

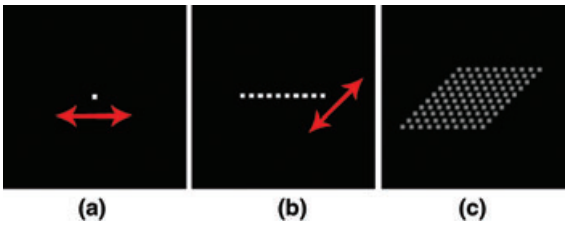

Figure 6: *In this series, a single bright test point is spread into a diamond over two passes. It shows, (a) the original image, (b) the result of the first pass and (c) the final result of the second pass.*

- We choose the rotation of the kernel in each pass to aid the construction of the desired polygonal shape.
- We use a uniform kernel (i.e. the samples are weighted evenly) but we ignore those samples that would contribute to intensity leakage.
- We vary the kernel width per-pixel based on the CoC map read from the alpha channel.
- We make no effort to align sample coordinates to pixel boundaries or centres.

4.3. Creating sample offsets

To create the sampling offsets, the desired number of samples per pass must first be decided. Using too few samples may result in sampling artefacts—especially in areas of the image with large CoC. Using too many samples will adversely affect performance. Through experimentation we found that a nice compromise is about 13 samples per pass, resulting in an effective 169 samples (13×13) after the second pass. Any number may be used, however.

The offsets are evenly spaced along a straight line 1 unit in length, and are centred about the origin (0,0). By choosing different orientations for the line, various parallelograms can be achieved. For instance, a square is made by using an angle of 0 degrees on the first pass, and 90 degrees on the second pass (see Figure 5). A 'diamond' with acute angles of 45 degrees is made by using an angle of 0 degrees on the first pass, and 45 degrees on the second pass (see Figure 6).

To increase shader performance, the sampling offsets for each filtering pass are pre-calculated once in the application code. The offsets are then passed into the shader via registers upon each pass. Before using these offsets, they are multiplied in the shader by the desired CoC diameter at the given pixel to achieve the correct kernel width for that pixel.

4.4. Multi-pass filtering

Each filtering pass is started by reading the colour map at the current pixel (including the alpha channel storing the CoC map). The depth map is also read at the current pixel.

Next, the sampling is done in a loop. For each sample, the texture coordinates are calculated by multiplying the sample offset (calculated earlier) with the desired CoC diameter stored in the alpha channel of the colour map. This offset is then added to the current pixel coordinates to get the final sample coordinates. The colour map and the depth map are both read at these coordinates, and the results are stored in local variables.

Still inside the loop, we apply the intensity leakage solution. The sample depth is compared to the current pixel depth, and the sample CoC is compared to the current pixel CoC. If the sample depth is less than the current pixel depth (i.e. it is closer to the camera) and the sample pixel CoC is less than the current pixel CoC (i.e. it is more focused), then using the sample would contribute to intensity leakage and the sample is ignored.

For valid samples, their colour and alpha values are added to a running total, and a counter is incremented to accumulate the number of valid samples found.

When all the samples have been made, the loop exits and the samples are averaged by dividing the running total by the number of valid samples found. This average is then returned by the shader.

One should note that we blur both the colour map, and the CoC map (stored in the alpha channel of the colour map) during each filtering pass. Though not based in any real physical process, we have found that blurring the CoC map in this way helps to reduce sharp edges that otherwise sometimes occur in defocused areas at depth-discontinuities (see Figure 7). We believe this acts as a less rigorous version of the more sophisticated CoC map blurring that Hammon [Ham07] performs.

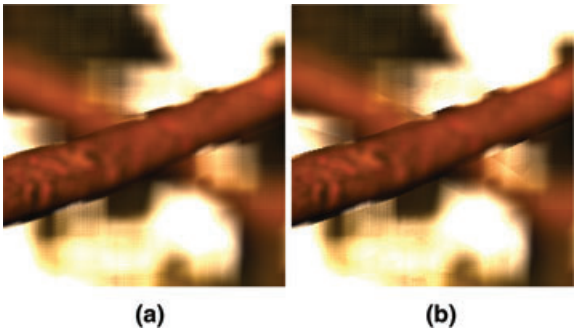

Figure 7: *This figure compares the effect of blurring (a) and not blurring (b) the CoC map alongside the colour map in each filtering pass. Note the sharp edges in (b) along the edges of the tree branches.*

Unlike the work by Riguer *et al*. [RTI03] and Scheuermann [Sch04], we do not create a downsampled copy of our colour map. Instead, all filtering operations are performed on the full-size colour map. This reduces artefacts, reduces the total number of passes required (along with the associated performance overhead) and simplifies the implementation.

Throughout the multi-pass filtering, we use **linear texture filtering** and **clamped texture addressing**. Linear filtering (although technically incorrect at depth discontinuities) provides smoother, less pixelated results than nearest-neighbour filtering. Clamped addressing provides reasonable sample extrapolation for pixels near the edges of the image.

4.5. Complex aperture shapes

To simulate more complex aperture shapes, (like hexagons and octagons), we combine two or more images produced by different parallelogram-shaped apertures. Conceptually, the combination can be thought of as a boolean operation, as in constructive solid geometry. For example, a hexagon can be created by taking the boolean intersection of two carefully chosen parallelograms (see Figure 8a). The area common to both shapes is kept, although other areas are discarded. Similarly, an octagon can be created by taking the boolean intersection of two squares (see Figure 8b). A star polygon can be created by taking the boolean union of two squares (see Figure 8c). With a little creativity a variety of shapes can be realized as the boolean intersections and unions of parallelograms, but for the sake of simplicity, we have chosen to limit our discussion to just hexagons and octagons.

Unfortunately for our purposes, performing a proper boolean intersection or union is out of the question. We are, after all, not actually dealing with individual polygons, but rather thousands of pixels representing thousands of overlapping CoC. It is thus impossible for us to consider any

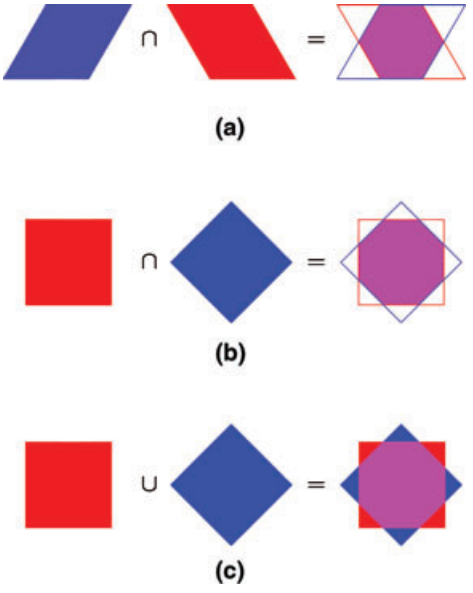

Figure 8: *Hexagons (a), octagons (b) and star polygons (c) can all be created by taking the boolean intersections or unions of various parallelograms. With a little creativity, a great variety of shapes can be realized this way.*

two polygons in isolation. There are however, two functions which when supplied with our images as arguments, will accomplish sufficiently similar effects: they are Min(x, y) and Max(x, y).

The Min(x, y) function returns the lesser (i.e. least bright) of x and y at every pixel, and thus approximates a boolean intersection by preserving bright pixels only in the areas where bright CoC coincide in both images. The Max(x, y) function returns the greater (i.e. most bright) of x and y at every pixel, and thus approximates a boolean union by preserving bright pixels wherever they exist in either image (see Figure 9).

The complete filtering workflow for our separable hexagon and octagon implementations is thus as shown in Figure 10. Note that we have eliminated a pass from the hexagon implementation by reusing the results of the first pass to produce both parallelograms. This optimization can be used any time two parallelograms share a side in common. Another optimization (not shown for clarity), is that we perform the final Min(x, y) or Max(x, y) operation as an extra step in the last filtering pass, rather than make a dedicated pass for it. This saves the overhead of setting up another pass for such a trivial operation.

The bokeh effects achieved though this simple technique can be surprisingly good. For instance, see Figure 1 where hexagonal and octagonal bokeh are clearly visible. Also see

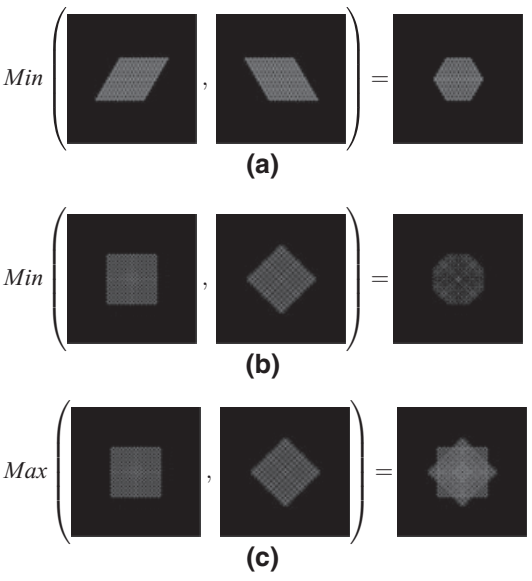

Figure 9: *The $Min(x, y)$ function approximates a boolean intersection. We use it to model the effect of hexagonal (a) and octagonal (b) apertures. The $Max(x, y)$ function approximates a boolean union. It could be used to model the effect of a star-shaped aperture (c).*

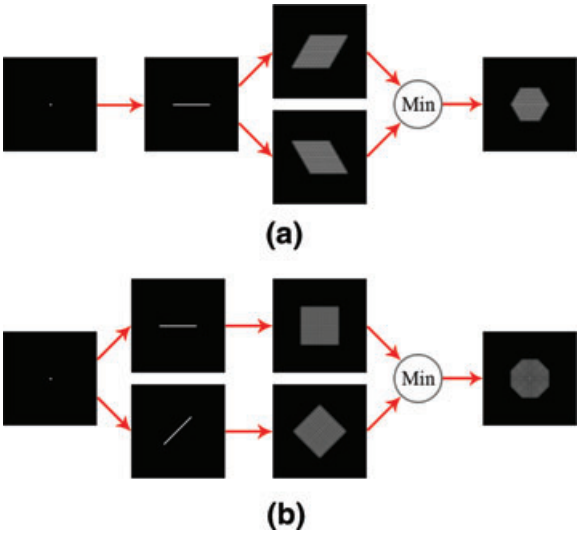

Figure 10: *The filtering workflow for our separable hexagon (a) and octagon (b) implementations.*

Figure 11: *A hexagonal bokeh effect, produced by our separable depth of field technique, is visible in the tree canopy of this 3D HDR scene.*

Figure 11 that demonstrates our hexagonal technique on a full 3D scene, with a wide range of CoC sizes.

Regarding the design of these separable filters for arbitrary aperture shapes, we have not been able to discern any particularly helpful rules that would make the construction process more formulaic, and this remains a potential area for future research. We can say that both convex and concave shapes are possible though (see Figure 9). We also note that the technique seems to lend itself best to polygons with fewer sides (as more sides inevitably require more parallelograms to make), and to polygons with even numbers of sides (though odd-sided shapes are certainly possible). Finally, we note that the artefacts produced by $\mathrm{Max}(x, y)$ (see Figure 13) tend to be less notable in practice than those of $\mathrm{Min}(x, y)$ (see Figure 12), and it may therefore be best to construct shapes as boolean unions when possible.

4.6. Artefacts

Our separable bokeh technique is not without its flaws. $\mathrm{Min}(x, y)$ causes artefacts where an area that was intended to be discarded (such as the corner of a square in our octagon shader) in one image, happens to coincide with another such area in the other image. The result is that seemingly 'from nowhere', bright defocused highlights of the wrong shape appear (see Figure 12). In addition, $\mathrm{Min}(x, y)$ generally causes a small decrease in image intensity (brightness) in the defocused areas of an image. $\mathrm{Max}(x, y)$ causes artefacts where two bright defocused highlights should overlap in the final image, but do not overlap in either input image. The result is that the highlights, although appearing

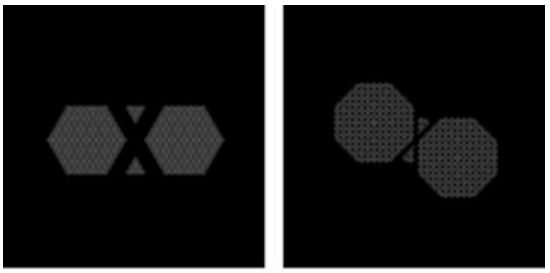

Figure 12: *In this test, two bright points are carefully placed to expose the bokeh artefacts caused by our usage of $Min(x, y)$. The effect is more exaggerated in our hexagonal implementation (a) than in our octagonal implementation (b) because of the more pronounced corners on the parallelograms used.*

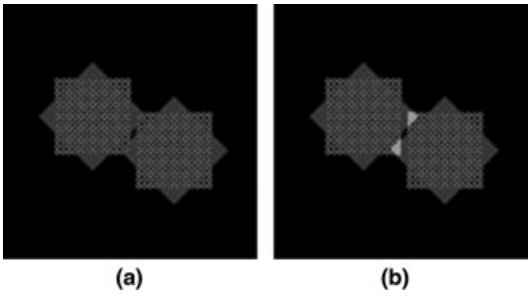

Figure 13: *In this test, two bright points are carefully placed to expose the bokeh artefacts caused by our usage of $Max(x, y)$. (a) Shows the result of our separable bokeh technique. For comparison, (b) shows the correct result where intensities are added in the overlapping areas.*

to overlap, do not properly add their intensities in overlapping areas (see Figure 13). In addition, $\mathrm{Max}(x, y)$ generally causes a small increase in image intensity (brightness) in the defocused areas of an image. Finally, like all gathering depth of field shaders, our approach naturally suffers from sampling artefacts in image areas with large CoC. Our technique tends to produce even, higher frequency noise (see Figures 12 and 13) though, which is generally less perceptible to human viewers. Dynamically adjusting the number of samples used based on the CoC diameter of each pixel might eliminate sampling artefacts for large diameters, although remaining efficient for small ones. Modern GPUs now support the dynamic branching necessary to implement this, and this could be an interesting direction for future improvement.

Our test images have been carefully designed to expose these artefacts. In practice though, we think these artefacts are slight enough as to go largely unnoticed in most scenes. This is especially true given that these artefacts are all absent from the focused areas of images where the attention of viewers will naturally be drawn. See Figure 1(c) for an example of the artefacts caused by overlapping parallelogram corners in practice (look for the small imperfections in the octagonal bokeh produced by the tightly spaced lights along the bridge deck).

Table 1: *The FPS attained by each shader technique at different numbers of effective samples. The naive shader fails to compile at 256, 400 and 576 samples, so these measurements could not be performed.*

Effective samples	16	64	144	256	400	576
Separable hexagon	105	80	62	52	46	40
Separable octagon	93	65	51	41	35	30
Naive	106	47	24			

5. Performance Tests

To determine the relative performance of our separable technique, we implemented the aforementioned hexagon and octagon versions of it in High Level Shader Language and integrated them into the Torque Game Engine Advanced (TGEA) 1.8.0 video game engine (see //www.garagegames.com/). In addition, we implemented the naive non-separable technique in the same manner. We then proceeded to measure the performance in frames per second (FPS) of each technique at several different 'effective sample counts' (16, 64, 144, 256, 400 and 576). In the case of the naive technique, the number of effective samples is simply equal to the number of actual sample operations performed. In the case of our separable techniques, we take the number of effective samples to be the number of samples performed per pass squared. This would be exactly correct in the case of a single parallelogram. In the case of our hexagons and octagons the situation is less clear because of the combination of multiple parallelograms using the $\mathrm{Min}(x, y)$ operator. We feel this offers a good approximation to the effective number of samples in our technique however.

A static test scene (of about 8000 polygons) was rendered at 800×800 with no multi-sample anti-aliasing. FPS was averaged automatically over a 10 s period using the FRAPS benchmarking tool (see //www.fraps.com/). All tests were performed on a 2.66GHz Intel Core 2 Duo CPU, with 2GB RAM and a GeForce 8600 GT 256MB video card, running Windows 7 (64-bit). Vertex and pixel shaders were compiled against Shader Model 3.0 profiles, (vs_3_0 and ps_3_0, respectively).

5.1. Results and discussion

As expected, our performance tests in TGEA (see Table 1) reveal that our separable hexagon shader performs the fastest, followed by our separable octagon shader and finally the naive shader, across virtually the whole range of effective

samples. One will note that the naive shader fails to compile for larger numbers of samples due to the size restrictions of Shader Model 3.0.

Our performance tests in TGEA demonstrate that our separable bokeh technique has a clear performance advantage over the naive approach. This is not surprising, considering the difference in the number of sample operations performed. For example, to achieve 144 effective samples, our separable hexagon shader performs just 36 sample operations per pixel (12 per pass $\times$ 3 passes). Similarly, our octagon shader performs 48 sample operations per pixel (12 per pass $\times$ 4 passes). The naive approach, however, must perform the full 144 sample operations per pixel to achieve a similar quality and its performance therefore suffers immensely. By leveraging the efficiency of separable filters, our approach attains much better frame rates at similar quality.

6. Conclusion

In this paper, we presented a novel 'gathering' depth of field shader technique for real-time applications. By leveraging separable filters, our technique can efficiently simulate the bokeh effects of polygonal apertures like squares, hexagons and octagons, among others. The approach introduces some minor bokeh artefacts, but they are confined to the defocused areas of an image, and we feel they would largely go unnoticed by casual viewers. Performance data from a video game engine test indicates that our separable bokeh technique attains much better frame rates than a naive non-separable approach, at comparable image quality.

6.1. Future work

It would be interesting to compare the performance and image quality of other depth of field techniques that consider bokeh—specifically point-splatting techniques—to our separable bokeh technique. We foresee that perhaps applications could use point-splatting for better quality on higher end hardware, with our separable bokeh functioning as a competent fall back technique on lower end hardware. In addition, it would be interesting to investigate the possibility of approximating lens effects like chromatic aberration in our approach.

References

[Dem04] Demers J.: Depth of field: A survey of techniques. In *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. R. Fernando (Ed.). Addison-Wesley Professional, Reading, MA (2004), pp. 375–390.

[Ham07] Hammon E.: Practical post-process depth of field. In *GPU Gems 3*. H. Nguyen (Ed.). Addison-Wesley Professional, Reading, MA (2007), pp. 583–605.

[KLO06] Kass M., Lefohn A., Owens J.: Interactive depth of field using simulated diffusion on a GPU. //graphics.pixar.com/library/DepthOfField/paper.pdf. Accessed 1 November 2011.

[KS07] Kraus M., Strengert M.: Depth-of-field rendering by pyramidal image processing. *Computer Graphics Forum 26*, 3 (September 2007), 645–654.

[LES09] Lee S., Eisemann E., Seidel H.: Depth-of-field rendering with multiview synthesis. *ACM Transactions on Graphics 28*, 5 (December 2009), 6 pages.

[LES10] Lee S., Eisemann E., Seidel H.: Real-time lens blur effects and focus control. *ACM Transactions on Graphics 29*, 4 (July 2010), 7 pages.

[LKC08] Lee S., Kim G. J., Choi S.: Real-time depth-of-field rendering using point splatting on per-pixel layers. *Computer Graphics Forum 27*, 7 (2008), 1955–1962.

[Mer97] Merklinger H. M.: A Technical View of Bokeh. Photo Techniques (1997). //www.trenholm.org/hmmerk/ATVB.pdf. Accessed 1 November 2011.

[Mon00] Monaco J.: *How to Read a Film: The World of Movies, Media, Multimedia: Language, History, Theory* (3rd edition). Oxford University Press, New York, USA, 2000.

[PC81] Potmesil M., Chakravarty I.: A lens and aperture camera model for synthetic image generation. In *Proceedings of the 8th Annual Conference on Computer Graphics and Interactive Techniques* (Dallas, TX, USA, 1981), ACM, pp. 297–305.

[RTI03] Riguer G., Tatarchuk N., Isidoro J.: Real-time depth of field simulation. In *ShaderX2: Shader Programming Tips and Tricks with DirectX 9.0*. Wordware Publishing, Inc., Plano, TX, October 2003. //tog.acm.org/resources/shaderx/Tips_and_Tricks_with_DirectX_9.pdf. Accessed 1 November 2011.

[Sch04] Scheuermann T.: Advanced Depth of Field, 2004. //developer.amd.com/media/gpu_assets/Scheuermann_DepthOfField.pdf. Accessed 1 November 2011.

[WB11] White J., Barré-brisebois C.: More performance! five rendering ideas from Battlefield 3 and Need For Speed: The Run, August 2011. //publications.dice.se/. Accessed 20 January 2012.

[WZH*10] Wu J., Zheng C., Hu X., Wang Y., Zhang L.: Realistic rendering of bokeh effect based on optical aberrations. *The Visual Computer 26*, 6-8 (April 2010), 555–563.

[ZCP07] Zhou T., Chen J. X., Pullen M.: Accurate depth of field simulation in real time. *Computer Graphics Forum 26*, 1 (2007), 15–23.

Appendix: Code Listing

```
//------------------------------------------------------------------------
// CoCMap_PS
// Lorne McIntosh, 2011
//------------------------------------------------------------------------
//      Computes the Circle of Confusion diameter at every pixel. This is
//      then converted into a % of the total image size, by assuming an image
//      sensor (or film) 24mm in height (35mm film standard). To avoid artifacts,
//      this % is then artificially clamped to MaxCoC. The result should be stored
//      in the alpha channel of the color map to be read by subsequent passes.
//------------------------------------------------------------------------
half4 CoCMap_PS(QuadVertexOutput IN,
        uniform sampler2D DepthSamp,
        uniform half A,         //aperture
        uniform half f,         //focal length
        uniform half S1,        //focal distance
        uniform half Far,       //far clipping plane
        uniform half MaxCoC     //max CoC diameter
) : COLOR
{
        //reconstruct scene depth at this pixel
        const half S2 = tex2D(DepthSamp, IN.UV).x * Far;

        //calculate circle of confusion diameter
        //(from http://en.wikipedia.org/wiki/Circle_of_confusion)
        const half c = A * (abs(S2 - S1) / S2) * (f / (S1 - f));

        //define height of camera's image sensor (width is assumed from
        //aspect ratio). (35mm "full-frame" film format is 36mm x 24mm)
        const half sensorHeight = 0.024f;       //24mm

        //put CoC into a % of the image sensor height
        const half percentOfSensor = c / sensorHeight;

        //artificially clamp % between 0 and MaxCoC
        const half blurFactor = clamp(percentOfSensor, 0.0f, MaxCoC);

        return blurFactor;
}
```

Listing 1: *CoC Map Pixel Shader (HLSL)*

```
//--------------------------------------------------------------------------
// makeOffsets
// Lorne McIntosh, 2011
//--------------------------------------------------------------------------
//      Creates linear sample offsets given an angle in radians
//      This would normally be done in the application once & stored. The values
//      would then be passed to the shader via constant registers.
//--------------------------------------------------------------------------
OffsetData makeOffsets(half angle)
{
        OffsetData output;

        const half aspectRatio = ViewportSize.x / ViewportSize.y;

        half radius = 0.5f;

        //convert from polar to cartesian
        half2 pt = half2(radius * cos(angle), radius * sin(angle));

        //account for aspect ratio (to avoid stretching highlights)
        pt.x /= aspectRatio;

        //create the interpolations
        for(int i = 0; i < numSamples; i++)
        {
                half t = i / (numSamples - 1.0f); //0 to 1
                output.offsets[i] = lerp(-pt, pt, t);
        }

        return output;
}
```

Listing 2: *Creating the sample offsets (HLSL)*

```
//--------------------------------------------------------------------------
// DepthOfField_PS
// Lorne McIntosh, 2011
//--------------------------------------------------------------------------
//      A bokeh-producing depth of field pixel shader. This performs one pass of a
//      seperable filter and must be used twice (with different offsets) to create
//      a cumulative effect. The alpha channel of ColorSamp must hold a CoC
//      (i.e. bluriness) map. See the paper for more info.
//--------------------------------------------------------------------------
half4 DepthOfField_PS(QuadVertexOutput IN,
        uniform sampler2D ColorSamp,
        uniform sampler2D DepthSamp,
        uniform OffsetData offsetData
) : COLOR
{
        //these are used to tune the "pixel-bleeding" fix
        const half bleedingBias = 0.02f;
        const half bleedingMult = 30.0f;

        //get the center samples for later reference
        half4 centerPixel = tex2D(ColorSamp, IN.UV);
        half centerDepth = tex2D(DepthSamp, IN.UV);

        //for finding the weighted average
        half4 color = 0.0f;
        half totalWeight = 0.0f;

        //for each sample
        for(int t = 0; t < numSamples; t++)
        {
```

```
            half2 offset = offsetData.offsets[t];

            //calculate the coordinates for this sample
            half2 sampleCoords = IN.UV + offset * centerPixel.a;

            //do the texture sampling for this sample
            half4 samplePixel = tex2D(ColorSamp, sampleCoords);
            half sampleDepth = tex2D(DepthSamp, sampleCoords);

            //-------------------------------------------------------------------------
            //Prevent focused foreground objects from bleeding onto blurry backgrounds
            //but allow focused background objects to bleed onto blurry foregrounds
            //-------------------------------------------------------------------------
            half weight = sampleDepth < centerDepth ? samplePixel.a * bleedingMult : 1.0f;
            weight = (centerPixel.a > samplePixel.a + bleedingBias) ? weight : 1.0f;
            weight = saturate(weight);
            //-------------------------------------------------------------------------

            //add this sample to the weighted average
            color += samplePixel * weight;
            totalWeight += weight;
        }

        //return the weighted average
        return color / totalWeight;
}
```

Listing 3: *Depth of Field Pixel Shader (HLSL)*

```
//-------------------------------------------------------------------------
// DepthOfFieldFinal_PS
// Lorne McIntosh, 2011
//-------------------------------------------------------------------------
//      A bokeh-producing depth of field pixel shader for use on the final pass
//      of the separable technique. It performs a regular DepthOfField_PS, and
//      then returns the min() of that result and the supplied ColorSampB to
//      complete the desired bokeh shape. max() could also be used. See the
//      paper for more info.
//-------------------------------------------------------------------------
half4 DepthOfFieldFinal_PS(QuadVertexOutput IN,
        uniform sampler2D ColorSampA,
        uniform sampler2D DepthSamp,
        uniform OffsetData offsetData,
        uniform sampler2D ColorSampB
) : COLOR
{
        half4 colorMapA = DepthOfField_PS(IN, ColorSampA, DepthSamp, offsetData);

        //now choose the minimum value to complete the shape
        return min(colorMapA, tex2D(ColorSampB, IN.UV));
}
```

Listing 4: *Depth of Field Final Pixel Shader (HLSL)*