Depth of Field with Bokeh Rendering

15

Charles de Rousiers and Matt Pettineo

15.1 Introduction

In order to increase realism and immersion, current games make frequent use of depth of field to simulate lenticular phenomena. Typical implementations use screen-space filtering techniques to roughly approximate a camera's circle of confusion for out-of-focus portions of a scene. While such approaches can provide pleasing results with minimal performance impact, crucial features present in real-life photography are still missing. In particular, lens-based cameras produce a phenomenon known as *bokeh* (blur in Japanese). Bokeh manifests as distinctive geometric shapes that are most visible in out-of-focus portions of an image with high local contrast (see Figure 15.1). The actual shape itself depends on the shape of the camera's aperture, which is typically circular, octagonal, hexagonal, or pentagonal.

Current and upcoming Direct3D 11 engines, e.g., CryENGINE, Unreal Engine 3, Lost Planet 2 Engine, have recently demonstrated new techniques for simulating bokeh depth of field, which reflects a rising interest in reproducing such effects in real time. However, these techniques have performance requirements that can potentially relegate them to high-end GPUs. The precise implementation details of these techniques also aren't publicly available, making it difficult to integrate these techniques into existing engines. Consequently, it remains an active area of research, as there is still a need for implementations that are suitable for a wider range of hardware.

A naive approach would be to explicitly render a quad for each pixel, with each quad using a texture containing the aperture shape. While this can produce excellent

Figure 15.1. Comparison between a simple blur-based depth of field (left) and a depth of field with bokeh rendering (right).

results [Sousa 11, Furturemark 11, Mittring and Dudash 11], it is also extremely inefficient due to the heavy fill rate and bandwidth requirements. Instead, we propose a hybrid method that mixes previous filtering-based approaches with quad rendering. Our method selects pixels with high local contrast and renders a single textured quad for each such pixel. The texture used for the quad contains the camera's aperture shape, which allows the quads to approximate bokeh effects. In order to achieve high performance, we use atomic counters in conjunction with an image texture for random memory access. An indirect draw command is also used, which avoids the need for expensive CPU-GPU synchronization. This efficient OpenGL 4.2 implementation allows rendering of thousands of aperture-shaped quads at high frame rates, and also ensures the temporal coherency of the rendered bokeh.

15.2 Depth of Field Phenomemon

Depth of field is an important effect for conveying a realistic sense of depth and scale, particularly in open scenes with a large viewing distance. Traditional real-time applications use a pinhole camera model [Pharr and Humphreys 10] for rasterization, which results in an infinite depth of field. However, real cameras use a thin lens, which introduces a limited depth of field based on aperture size and focal distance. Objects outside this region appear blurred on the final image, while objects inside it remain sharp (see Figure 15.2).

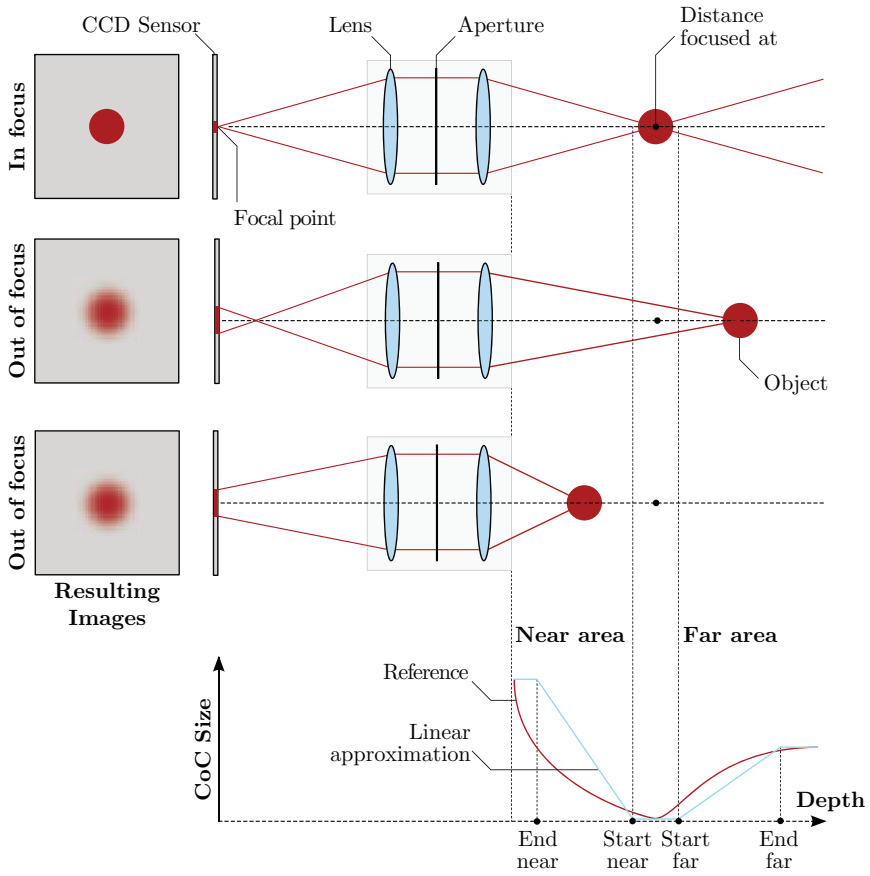

Figure 15.2. Depth of field phenomenon, where a thin lens introduces a limited depth of field. In-focus objects appear sharp, while out-of-focus objects appear blurred. The size of the circle of confusion depends on the distance between object and the point at which the camera is focused. We use a linear approximation in order to simplify parameters as well as run-time computations.

The "blurriness" of an object is defined by its *circle of confusion* (CoC). The size of this CoC depends on the distance between the object and the area on which the camera is focused. The further an object is from the focused area, the blurrier it appears. The size of the CoC does not increase linearly based on this distance. The size actually increases faster in the out-of-focus foreground area than it does in the out-of-focus background area (see Figure 15.2). Since the CoC size ultimately depends on focal distance, lens size, and aperture shape, setting up the simulation parameters may not be intuitive to someone inexperienced with photography. This

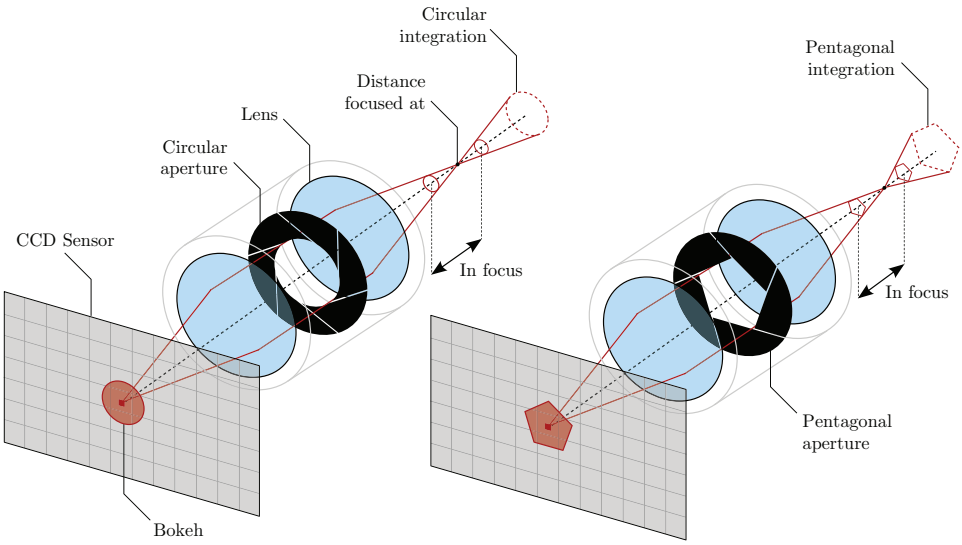

Figure 15.3. Aperture shape of a camera. Aperture blocks a portion of the incoming light. Its shape modifies the pixel integration and, hence, changes the bokeh shape.

is why we use a simple linear approximation as proposed by [Earl Hammon 07] (see Figure 15.2).

The *aperture* of a camera is responsible for allowing light to pass through the lens and strike the sensor (or film).[1] The shape of this aperture directly impacts the formation of the image since each out-of-focus point is convolved with the aperture shape (see Figure 15.3).

While it is often difficult to see distinct bokeh patterns in areas with low contrast, bokeh is clearly visible in areas that are significantly brighter than their surroundings. We use this observation as a heuristic to determine where bokeh quads need to be drawn in order to provide a plausible approximation.

15.3 Related Work

Several methods have been proposed during the last decade for efficiently approximating a depth-of-field effect. However, those methods use a Gaussian blur or heat diffusion to simulate out-of-focus areas [Earl Hammon 07, Lee et al. 09, Kosloff and Barsky 07] and are therefore unable to reproduce bokeh effects.

[1] If an object or the camera moves while the aperture is open, the objects will appear blurred. This is known as motion blur.

An earlier approach from Krivanek [Krivanek et al. 03] uses sprite splatting as a means for implementing depth of field rather than a filtering approach. While this brute-force method does produce bokeh shapes, it is quite inefficient due to excessive overdraw and bandwidth consumption.

The video game industry has shown a recent interest in bokeh effects [Capcom 07, Sousa 11, Furturemark 11, Mittring and Dudash 11]. While complete implementation details are not available, the methods used by video game developers largely take a similar approach to Krivanek where sprites are rendered for each pixel. Consequently, these techniques make use of complex optimizations, e.g., hierarchical rasterization, multiple downscaling passes, in order to improve performance.

A recent approach proposed by White [White and Brisebois 11] reproduces hexagonal bokeh using several directional blur passes. While efficient, this method does not support arbitrary aperture shapes.

15.4 Algorithm

We observe that only points with a high local contrast will produce distinct bokeh shapes. We use this heuristic to detect bokeh positions in screen space [Pettineo 11] and then splat textured quads at those locations. The remaining pixels use a blur-based approach to simulate a circle of confusion.

15.4.1 Overview

Our approach is divided into four passes (see Figure 15.4). The first pass computes the CoC size for each pixel based on its depth value, and then outputs a linear depth

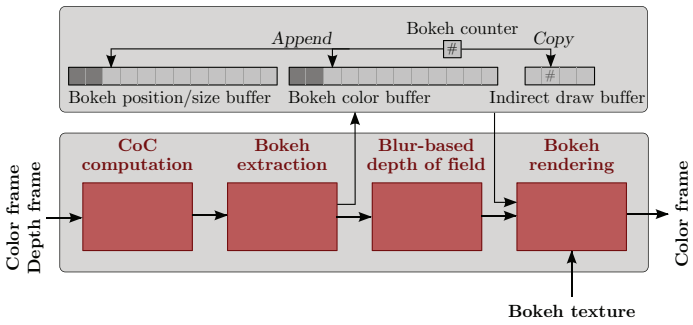

Figure 15.4. Overview of the pipeline. It is composed of four passes and takes as input the color and depth buffers of the current frame. It outputs a final color image with depth of field and bokeh effects.

value to a framebuffer attachment.[2] Then, the second pass computes the contrast of the current pixel by comparing the pixel's brightness with the brightness of the 5×5 neighboring pixels. If this contrast is above a predefined threshold, its position, CoC size, and average color are appended to a buffer. During the third pass, a blur-based depth of field is computed with one of the previous methods, e.g., Gaussian blur. Finally, a fourth pass splats textured quads at the bokeh positions that were appended to the buffer in the second pass.

In order to maintain high performance, it is crucial to avoid CPU/GPU synchronization. We ensure this by making use of an indirect draw command [Bolz et al. 09], which renders a number of quads based on the count stored in the append buffer. This way, the number of bokeh points detected by the GPU is never read back by the CPU.

15.4.2 Circle of Confusion Computation

Setting up depth of field using physical camera parameters, such as focal length and aperture size, can be nonintuitive for those not familiar with optics or photography. Instead, we define two areas where geometry is out of focus: a near/foreground area and a far/background area. Both areas are delimited with a near and far depth value (see Figure 15.2). In both regions, the blur amount is linearly interpolated between the two bounds. This allows a simple and intuitive means of describing the depth of field for a given scene.

$$\mathrm{CoC} = \frac{Z_{\mathrm{pixel}} - Z_{\mathrm{start}}}{Z_{\mathrm{end}} - Z_{\mathrm{start}}}.$$

The resulting CoC size from this equation is normalized to [0,1]. An extra parameter `MaxRadius` determines the final size of the blur in screen space a posteriori. This setting can be tweaked by artists in order to achieve the desired appearance and provides a means of balancing performance: smaller values of `MaxRadius` result in greater performance.

15.4.3 Bokeh Detection

The detection pass aims to detect pixels from which we will generate bokeh shapes. To detect such pixels, we use the following heuristic: *a pixel with high contrast in a given neighborhood will generate a bokeh shape*. We compare the current pixel luminance L_{pixel} to its neighborhood luminance L_{neigh}. If the difference $L_{\mathrm{pixel}} - L_{\mathrm{neigh}}$ is greater than the threshold, `LumThreshold`, then the current pixel is registered as a bokeh point.[3] Pixels detected as bokeh are sparse, which means writing them into a framebuffer attachment would be wasteful in terms of both memory usage

[2] If a linear depth buffer is available as an input, the first two passes can be merged together.

[3] We also use the threshold `CoCThreshold` to discard bokeh with a small radius.

and bandwidth.[4] To address this problem, we use the OpenGL `ImageBuffer` [Bolz et al. 11] in combination with an "atomic counter" [Licea-Kane et al. 11]. This allows us to build a vector in which we append parameters for detected bokeh points. `ImageBuffers` have to be preallocated with a given size, i.e., the maximum number of bokeh sprites that can be displayed on screen. The atomic counter `BokehCounter` stores the number of appended bokeh points. Its current value indicates the next free cell in the `ImageBuffer` vector. Two `ImageBuffer` variables, `BokehPosition` and `BokehColor`, are used to store the CoC size, position, and color of the detected bokeh points. See Listings 15.1 and 15.2 and Figure 15.5.

```
// Create indirect buffer
GLuint indirectBufferID;
glGenBuffers(1, &indirectBufferID);
glBindBuffer(GL_DRAW_INDIRECT_BUFFER, indirectBufferID);
DrawArraysIndirectCommand indirectCmd;
indirectCmd.count = 1;
indirectCmd.primCount = 0;
indirectCmd.first = 0;
indirectCmd.reservedMustBeZero = 0;
glBufferData(GL_DRAW_INDIRECT_BUFFER, sizeof(DrawArraysIndirectCommand), &↩
    indirecCmd, GL_DYNAMIC_DRAW);

// Create a texture proxy for the indirect buffer
// (used during bokeh count synch.)
glGenTextures(1, &bokehCountTexID);
glBindTexture(GL_TEXTURE_BUFFER, bokehCountTexID);
glTexBuffer(GL_TEXTURE_BUFFER, GL_R32UI, indirectBufferID);

// Create an atomic counter
glGenBuffers(1, &bokehCounterID);
glBindBuffer(GL_ATOMIC_COUNTER_BUFFER, bokehCounterID);
glBufferData(GL_ATOMIC_COUNTER_BUFFER, sizeof(unsigned int), 0, GL_DYNAMIC_DRAW);

// Create position and color textures with a GL_RGBA32F inner format
...

// Bind atomic counter
glBindBufferBase(GL_ATOMIC_COUNTER_BUFFER, 0, bokehCounterID);

// Bind position image buffer
glActiveTexture(GL_TEXTURE0 + bokehPosionTexUnit);
glBindImageTexture(bokehPostionTexUnit, bokehPositionTexID, 0, false, 0, ↩
    GL_WRITE_ONLY, GL_RGBA32F);

// Bind color image buffer
glActiveTexture(GL_TEXTURE0 + bokehColorTexUnit);
glBindImageTexture(bokehColorTexUnit, bokehColorTexID, 0, false, 0, GL_WRITE_ONLY,↩
    GL_RGBA32F);

DrawSceenTriangle();
```

Listing 15.1. Host application for extracting bokehs *(Pass 2)*.

[4] During our tests, less than 1% of pixels are detected as bokeh at 720p.

```
#version 420
// Bokeh counter, position (x,y,z,size), and color
layout(binding = 0, offset = 0) uniform atomic_uint BokehCounter;
layout(size4x32) writeonly uniform image1D BokehPositionTex;
layout(size4x32) writeonly uniform image1D BokehColorTex;

// Constrast and CoC thresholds
uniform float LumThreshold;
uniform float CoCThreshold;
...

float cocCenter;  // Current CoC size
vec3 colorCenter; // Current pixel color
vec3 colorNeighs; // Average color of the neighborhood

// Append pixel whose constrast is greater than the user's threshold
float lumNeighs = dot(colorNeighs, vec3(0.299f, 0.587f, 0.114f));
float lumCenter = dot(colorCenter, vec3(0.299f, 0.587f, 0.114f));
if((lumCenter - lumNeighs) > LumThreshold && cocCenter > CoCThreshold)
{
  int current = int(atomicCounterIncrement(BokehCounter));
  imageStore(BokehPositionTex, current,vec4(gl_FragCoord.x, gl_FragCoord.y, depth, ↩
      cocCenter));
  imageStore(BokehColorTex, current, vec4(colorCenter, 1));
}
```

Listing 15.2. Fragment shader for extracting bokehs *(Pass 2)*.

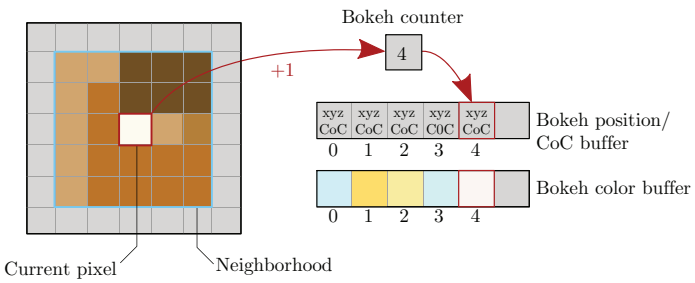

Figure 15.5. Bokeh detection. The luminance of the current pixel is compared to its neighborhood. If the difference is greater than `LumThreshold`, bokeh parameters, i.e., position, color, and CoC, are appended into `BokehPosition` and `BokehColor` image buffers. The atomic counter `BokehCounter` is also incremented.

15.4.4 Blur-Based Depth of Field

Several approaches are possible for this pass. We refer readers to previous work for this step. Nevertheless, here is a short summary of popular approaches:

- Perform a Gaussian blur with fixed kernel width at various resolutions, and apply a linear interpolation to blend them according to CoC size.

- Perform a Poisson disc sampling in screen space, with radius determined by pixel CoC size.[5]
- Apply a large-width bilateral filter where invalid pixels are rejected based on depth.[6]

The Hammon's approach [Earl Hammon 07] can be used for processing the foreground out-of-focused area. This approach is compatible with the bokeh rendering technique presented here.

15.4.5 Bokeh Rendering

In order to avoid CPU/GPU synchronization, we use the indirect drawing command `glDrawArraysIndirect`. This command draws instances where the count is read from a buffer located in GPU memory. This buffer can be updated from either the CPU or the GPU. In order to allow the GPU to operate independently of the CPU, we update this buffer from the GPU before the last pass. We bind this indirect buffer as an `ImageTexture` and copy the value of the atomic counter into it. Thus, the number of instances drawn is equal to the number of detected bokeh points (see Listing 15.3).

We use this command in combination with a vertex array object (VAO), describing a single vertex to render a point primitive. The instanced points are translated by the vertex shader so that they are located at the screen-space bokeh position. This position is read from the `BokehPosition` array buffer, which is indexed using

```
#version 420
layout(binding = 0, offset = 0) uniform atomic_uint BokehCounter;
layout(size1x32) writeonly uniform uimage1D IndirectBufferTex;
out vec4 FragColor;

void main()
{
  imageStore(IndirectBufferTex, 1, uvec4(atomicCounter(BokehCounter), 0, 0, 0));
  FragColor = vec4(0);
}
```

Listing 15.3. Synchronization of the indirect buffer with the atomic counter *(Pass 3/4)*. The function `glMemoryBarrier` has to be call before this shader in order to ensure that all bokeh data have been written.

[5] A random rotation can be applied to a Poisson sampling pattern for transforming aliasing into noise.

[6] For implementation details, we refer the reader to the code sample. This approach offers a good compromise between quality and performance. However, larger filter kernels require a large sampling radius. An OpenCL implementation would allow for better performance since shared memory can be used to cache texture fetches.

```
#version 420
uniform mat4 Transformation;
uniform vec2 PixelScale;
in float vRadius[1];
in vec4 vColor[1];
out vec4 gColor;
out vec2 gTexCoord;
layout(points) in;
layout(triangle_strip, max_vertices = 6) out;

void main()
{
  gl_Layer = 0;
  vec4 offsetx = vec4(PixelScale.x * Radius[0], 0, 0, 0);
  vec4 offsety = vec4(0, PixelScale.y * Radius[0], 0, 0);
  gColor = vColor[0];
  gl_Position  = Transformation * (gl_in[0].gl_Position - offsetx - offsety);
  gTexCoord = vec2(0,0);
  EmitVertex();
  gl_Position = Transformation * (gl_in[0].gl_Position + offsetx - offsety);
  gTexCoord = vec2(1,0);
  EmitVertex();
  gl_Position = Transformation * (gl_in[0].gl_Position - offsetx + offsety);
  gTexCoord = vec2(0,1);
  EmitVertex();
  gl_Position = Transformation * (gl_in[0].gl_Position + offsetx + offsety);
  gTexCoord = vec2(1,1);
  EmitVertex();
  EndPrimitive();
}
```

Listing 15.4. Geometry shader for rendering bokeh *(Pass 4)*.

the built-in `gl_InstanceID` input variable. After being transformed in the vertex shader, each point is expanded into a quad in the geometry shader. The size of this quad is determined by the bokeh size, which is also read from the `BokehPosition` array buffer. Finally, the fragment shader applies the alpha texture bokeh onto the quad and multiplies it by the bokeh color, which is read from the `BokehColor` array buffer (see Listing 15.4).

15.5 Results

Figures 15.1, 15.6, and 15.7 show the rendering of a tank using our method. Since the final bokeh shape is texture-driven, we can apply arbitrary shapes (see Figure 15.7).

Figure 15.8 details the rendering times of each pass as well as the number of detected bokeh points. We can see that the blur-based depth-of-field pass is the most expensive, indicating that a more optimal approach might be more suitable. Unlike the blur and detection passes, the rendering pass is strongly dependent on the number of detected bokeh points and is fill-rate bound. When the scene is entirely out of focus, our algorithm detects around 5,000 bokeh points in the tank scene. In this case, the cost of the rendering pass is less than 2 ms.

Figure 15.6. Rendering of a tank with a small depth of field. Bokeh shapes are clearly visible on the more reflective surfaces of the tank.

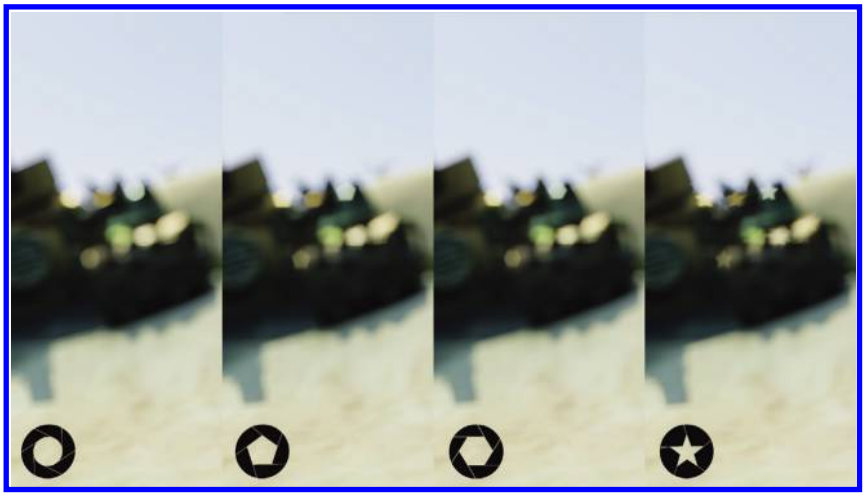

Figure 15.7. Rendering of the same scene with different aperture shapes. Bokeh textures are 32×32 pixel grayscale bitmap. From left to right: a circle aperture, a pentagonal aperture, an hexagonal aperture, and a star aperture.

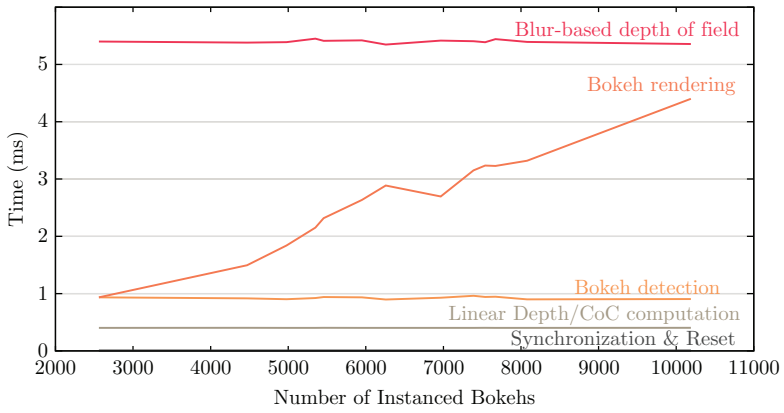

Figure 15.8. Timings of the different passes for varying numbers of detected bokeh points. Those timings have been recorded on an NVIDIA GeForce GTX 580 at 1280 × 720.

15.6 Discussion

Temporal coherence is a natural concern for this approach. Like other methods, we base our approach on the final color buffer. If subpixel aliasing is addressed by previous rendering steps, our approach is stable, and bokeh shapes are coherent from frame to frame. In the case of subpixel aliasing, our method exhibits the same limitations as all previous methods, and the resulting bokeh shapes may flicker.

Also, our method requires preallocated buffers for storing the bokeh position and color. Consequently, a maximum number of bokeh points has to be specified. If this number is too low, bokeh points may pop and flicker from frame to frame with little coherency. If this number is too large, then GPU memory is potentially wasted. Thus, the maximum number of sprites must be carefully chosen to suit the type of scene being displayed.

15.7 Conclusion

We have presented an efficient implementation for rendering a depth-of-field effect with bokeh. This method allows us to combine an efficient blur-based approach with plausible bokeh reproduction. We use a heuristic to identify pixels that produce distinct bokeh shapes and then render those shapes as textured quads. This implementation avoids costly CPU/GPU synchronization through the use of indirect draw commands. These commands allow the GPU to directly read the number of instances without the need for CPU readback.

While this approach provides good visual results, several optimizations can be made in order to improve performance. In particular, large CoC sizes require rasterization of quads that cover a significant portion of the screen. Using hierarchical rasterization,[7] as proposed in [Furturemark 11], could improve performance by reducing the number of pixels that need to be shaded and blended.

Bibliography

[Bolz et al. 09] Jeff Bolz, Pat Brown, Barthold Lichtenbelt, Bill Licea-Kane, Merry Bruce, Sellers Graham, Roth Greg, Haemel Nick, Boudier Pierre, and Piers Daniell. "ARB_draw_indirect." OpenGL extension, 2009.

[Bolz et al. 11] Jeff Bolz, Pat Brown, Barthold Lichtenbelt, Bill Licea-Kane, Eric Werness, Graham Sellers, Greg Roth, Nick Haemel, Pierre Boudier, and Piers Daniell. "ARB_shader_image_load_store." OpenGL extension, 2011.

[Capcom 07] Capcom. "Lost Planet 2 DX10 Engine." 2007.

[Earl Hammon 07] Earl Hammon Jr. "Blur Practical Post-Process Depth of Field." In *GPU Gems 3: Infinity Ward*. Reading, MA: Addison Wesley, 2007.

[Furturemark 11] Furturemark. "3DMark11 Whitepaper." 2011.

[Kosloff and Barsky 07] Todd Jerome Kosloff and Brian A. Barsky. "An Algorithm for Rendering Generalized Depth of Field Effects Based on Simulated Heat Diffusion." Technical report, University of California, Berkeley, 2007.

[Krivanek et al. 03] Jaroslav Krivanek, Jiri Zara, and Kadi Bouatouch. "Fast Depth of Field Rendering with Surface Splatting." *Proceedings of Computer Graphics International*. 2003.

[Lee et al. 09] Sungkil Lee, Elmar Eisemann, and Hans-Peter Seidel. "Depth-of-Field Rendering with Multiview Synthesis." *SIGGRAPH Asia '09*, pp. 134:1–134:6, 2009.

[Licea-Kane et al. 11] Bill Licea-Kane, Barthold Lichtenbelt, Chris Dodd, Eric Werness, Graham Sellers, Greg Roth, Jeff Bolz, Nick Haemel, Pat Brown, Pierre Boudier, and Piers Daniell. "ARB_shader_atomic_counters." OpenGL extension, 2011.

[Mittring and Dudash 11] Martin Mittring and Bryan Dudash. "The Technology Behind the DirectX 11 Unreal Engine "Samaritan" Demo." GDC. Epics Games, 2011.

[Pettineo 11] Matt Pettineo. "How to Fake Bokeh." Ready At Dawn Studios, 2011.

[Pharr and Humphreys 10] Matt Pharr and Greg Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation*, Second edition. San Francisco, CA: Morgan Kaufmann Publishers Inc., 2010.

[7] Quads are rasterized into different viewports according to their size: full resolution, half resolution, quarter resolution, etc. The bigger a quad is, less the viewport resolution is.

[Sousa 11] Tiago Sousa. "Crysis 2 DX11 Ultra Upgrade." Crytek, 2011.

[White and Brisebois 11] John White and Colin Barre Brisebois. "More Performance Five Rendering Ideas from Battlefield 3 and Need for Speed: The Run." Siggraph talk. Black Box and Dice, 2011.