IV 3

Practical Gather-based Bokeh Depth of Field

Wojciech Sterna

3.1 Introduction

Depth of field is one of the most common post-process algorithms used in today's games. As it turns out, it might often end up being one of the most expensive as was the case of the initial implementation in *Killzone: Shadow Fall* [Valient 2013]. Some implementations rely on easy-to-implement, good-looking, but costly, scatter-based approaches, as in [Pettineo 2011]. In this chapter, we describe a fast (a little over 1.5 ms in 1080p on GeForce 660 GTX which is our test hardware of choice) and easy to implement gather-based solution, strongly inspired by [Sousa 2013]. Figure 3.1 shows the results. Note that the ornaments to the left and in the center are in focus.

3.2 Problem Formulation

The idea behind implementing fast real-time depth-of-field effects is rather simple. We need to blur the out-of-focus areas of the image. The area that is in focus remains sharp. The area in between the out-of-focus and in-focus areas should be blended between the two to achieve a smooth effect.

There are two out-of-focus areas. One is right in front of the camera and the other is the background. The former we will refer to as the near field and the latter is called the far field. There is also a focal plane perpendicular to the frustum's near plane, whose distance to the plane is the distance at which the image is completely sharp. Usually, depth-of-field effects allow pixels to be sharp only on the focal plane. In our implementation, we allow the size of the sharp area to be customized (see Figure 3.2).

How much a pixel is to be blurred depends on its circle-of-confusion (CoC) value. This value is computed based on the pixel's distance from the camera, or actually, to be more precise, on nb, ne, fb, and fe values. Pixels between nb and

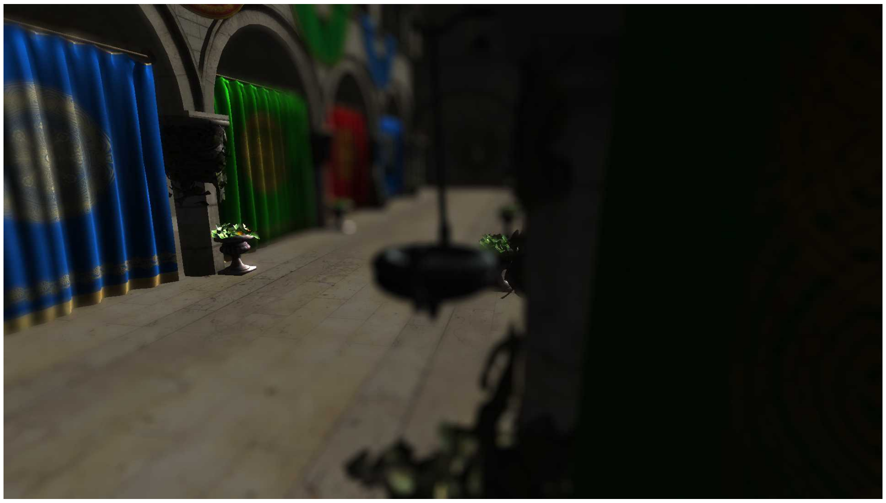

Figure 3.1. Depth-of-field implementation.

ne as well as fb and fe can be blended linearly or in any other fashion we want. In the demo accompanying this chapter, we use linear blend although [Sousa 2013] suggests using different blending functions. It is worth noting that we are using an empirical CoC compuation model; [Sousa 2013] explains how to use a physical one.

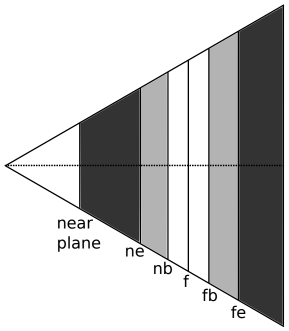

Figure 3.2. The dark grey areas represent full out-of-focus areas called the near and far fields. The light gray represents the areas where blending of out-of-focus areas towards sharp areas occur. The white areas in the middle are completely sharp. The value f is the distance from the camera to the focal plane; nb (near-blend begin) and ne (near-blend end) describe the distances at which in-focus and near field out-of-focus regions blend. Similarly, fb (far-blend begin) and fe (far-blend end) describe the distances at which in-focus and far field out-of-focus regions blend.

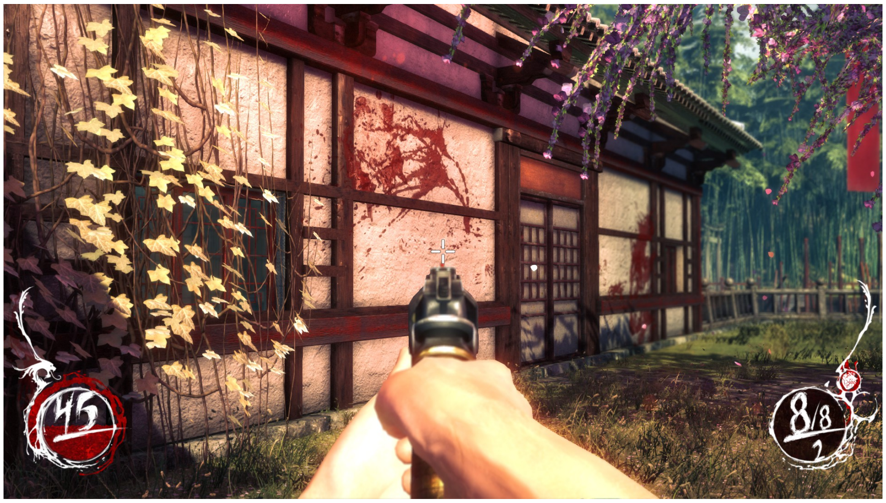

Figure 3.3. The gun in the center of the screen should bleed onto the sharp background but instead the blur is limited to the object's screen boundaries. This might make an impression that the object uses a low-res texture rather than that it is out-of-focus (i.e., blurred).

3.2.1 Fundamental Problem of Near Field

As you already know or will see later in this chapter, generating the far field and blending it with the in-focus area is much easier than doing this for the near field. One of the first well-known implementations of real-time depth-of-field is [Scheuermann and Tatarchuk 2003]. A similar implementation can be found in the *Shadow Warrior* game. It handles far field nicely but struggles with making near field look believable as is shown in Figure 3.3.

The proper solution is to make the near-field pixels bleed onto the sharp in-focus background (an effect called occlusion). However, in real life, this effect also makes an out-of-focus foreground object become more semi-transparent, as can be seen in Figure 3.4.

This problem cannot be solved properly in our case because we don't have pixels that are behind. One obvious solution would be to store layers of pixels (this way also solving, to some extent, the general transparency problem), but that solution would be too expensive for real-time rendering purposes.

There are ways to handle this situation without layering. One approach is simply to not allow any semi-transparency on a foreground near-field object. This might, however, create an impression that the near-field object is a little bigger (caused by pixels that bleed onto the background) when it is in an out-of-focus near-field foreground area. Also, we might end up with blur that has effectively twice as small a radius than required (see Figure 3.5 (a)).

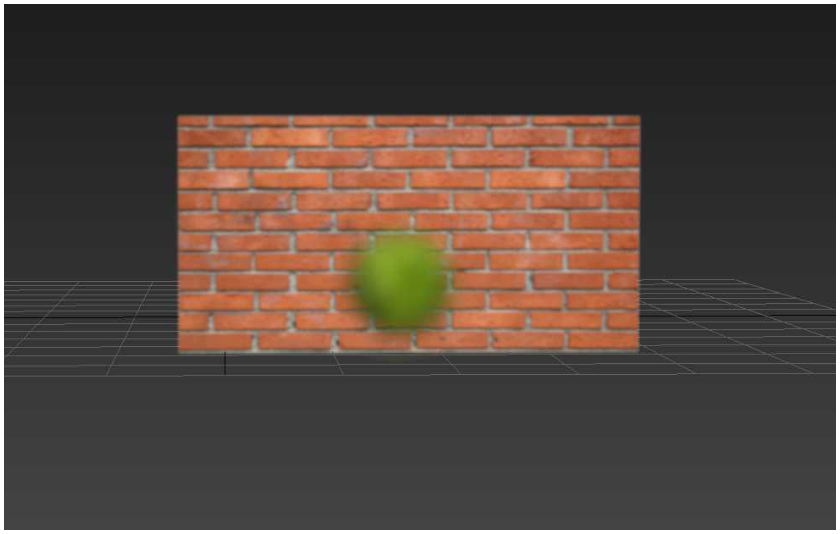

Figure 3.4. The greenish ball is on the near field (in the foreground). It is not only blurred but also transparent on the edges. This indicates strong out-of-focus effect.

A handful of practical solutions is presented in [Jimenez 2014], where they "extend" the background to create a sort of a second layer so that the out-of-focus foreground can become transparent.

The solution proposed in this chapter is to blur the out-of-focus near field along with the sharp background, dragging in background pixels outside of an

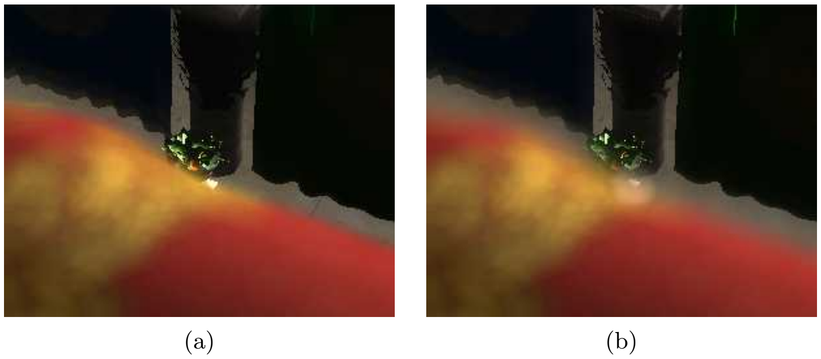

Figure 3.5. (a) pixels only bleed out onto the background and prevent any background-to-foreground bleeding. As a result the blur is less pronounced than it is in (b), which shows the solution proposed in this article. Note, however, how the pad of the ornament in the background renders partially blurred (the pad corner's circle in the middle of image (b)).

object's silhouette onto the foreground region of the image. This will create an inaccurate effect that the sharp background is actually blurred near the silhouettes of near-field objects. However, this artifact is not really that distracting, and the solution itself is very easy to implement (see Figure 3.5 (b)).

3.3 Algorithm

We assume that the input color buffer that we will be using is in R11G11B10F (float) format, which is one of the most popular formats used in today's games for storing color values that go into post process.

One of the basic ideas allowing fast depth-of-field implementation is to render to a lower-resolution render target, as is done with many other post-process effects. Here we render the near and far fields, separately, into quarter-resolution buffers and then up-sample the results, blending them with the original input color buffer.

Figure 3.6 outlines the algorithm with passes specified along with various buffers that are used.

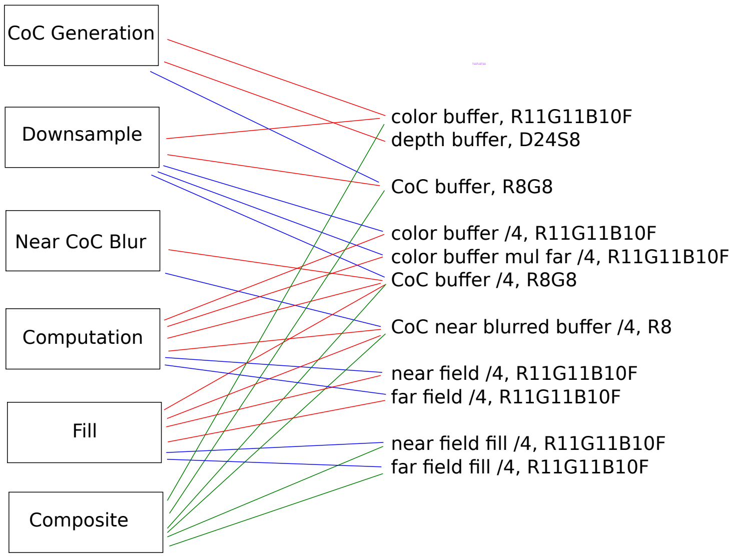
CoC Generation
Downsample
Near CoC Blur
Computation
Fill
Composite
color buffer, R11G11B10F
depth buffer, D24S8
CoC buffer, R8G8
color buffer /4, R11G11B10F
color buffer mul far /4, R11G11B10F
CoC buffer /4, R8G8
CoC near blurred buffer /4, R8
near field /4, R11G11B10F
far field /4, R11G11B10F
near field fill /4, R11G11B10F
far field fill /4, R11G11B10F

Figure 3.6. Consecutive passes of the algorithm on the left and input/output buffers on the right. A red line indicates that a buffer is an input to a pass. A blue line indicates that a buffer is generated in a pass. Green lines indicate input buffers to the composite pass specifically (to avoid confusion). The term "/4" indicates that a buffer is of quarter resolution. At the end of the description of each buffer is its format.

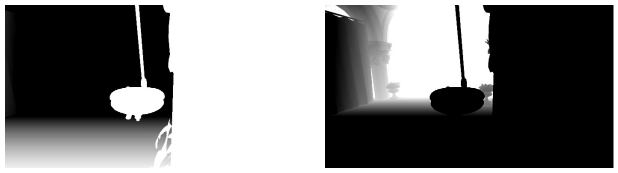

Figure 3.7. Near (left image) and far (right image) circle of confusion values.

3.3.1 Circle of Confusion Generation

In the first pass we calculate the circle of confusion and render it to a full-screen R8G8 (two single-byte components) buffer. The R component holds the CoC value for the near field whereas the G component holds the CoC value for the far field. Those values are calculated based on variables introduced in Section 3.2. However, in the demo application we don't actually expose these to the user to control. Instead, we expose the focal plane distance and range in which the interpolation between the in-focus and out-of-focus near and far fields occur. These values are converted to nb, ne, fb, and fe and fed to the shader. Figure 3.7 shows near and far CoC values.

3.3.2 Downsample

The next step is to downsample two buffers—the input color and the CoC buffer.

There are two reasons we need a downsampled CoC buffer. The first reason is to have a lower bandwidth in the actual depth-of-field calculation pass. The other reason is to be able to perform edge-aware upsampling of the quarter-resolution far field in the final composite pass. The CoC buffer is downsampled simply by using point filtering, taking a pixel from the upper-left corner in each 2×2 input CoC texture quad.

The color buffer can be downsampled in various ways—with point, linear, or some other kind of filtering. Point filtering (that is, taking only one out of four pixels) causes aliasing, which manifests as unpleasant flickering in the final effect. Linear filtering, on the other hand, eliminates flickering at the expense of introducing occasional haloing (between sharp in-focus foreground and blurred out-of-focus background). A good middle-ground is to use edge-aware bilateral downsampling; i.e., performing linear interpolation of all pixels in each 2×2 quad that have depth-buffer values close to, say, the upper-left pixel in the quad. That solves the problem of flickering and eliminates haloing (which is caused by color bleeding). This is exactly what the algorithm does with one exception; it is not the depth buffer that is used to compare samples but rather the far CoC values. The reason for this is that flickering only appears on the far field and it would be a waste of bandwidth to sample the depth buffer given that CoC values are already available.

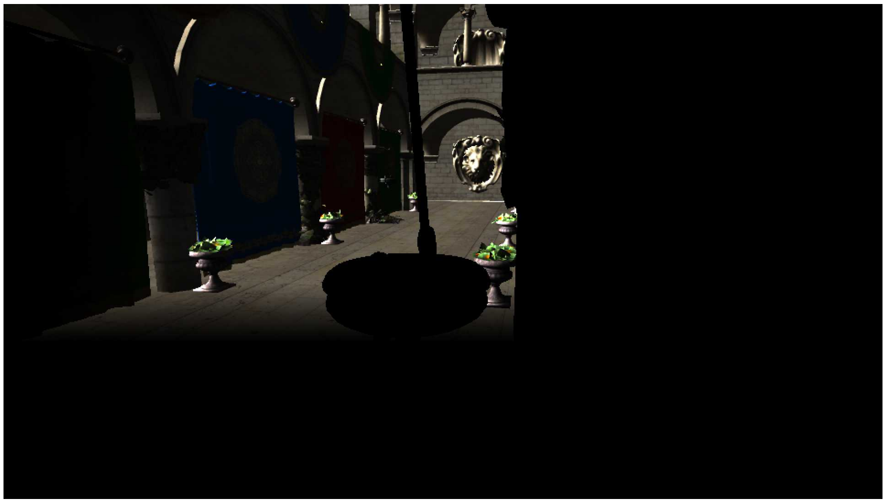

Figure 3.8. Color buffer multiplied by far circle of confusion. Note that there is no color bleeding on the edges of the "black objects."

To sum up, the downsampled color buffer is output twice, once in unaltered form and the second time multiplied by the downsampled far CoC value. Why this is done will be clear later.

Please note here that this pass renders to three render targets: two downsampled color buffers (one multiplied by far CoC, as depicted in Figure 3.8) and to the downsampled CoC buffer.

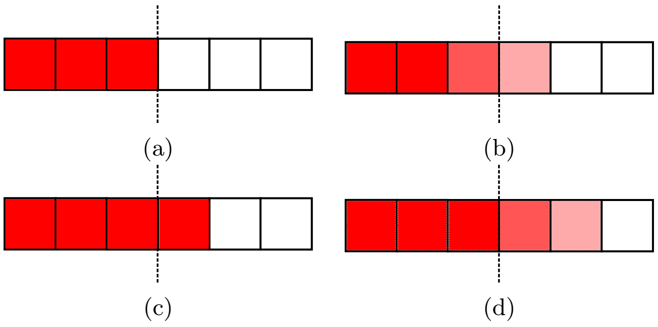

Figure 3.9. Image (a) is the input image; (b) shows the result of applying a three-pixel-wide blur filter to (a). In (c), a three-pixel-wide max filter is applied to (a). Finally, in (d), blur is applied to (c). Performing an n-width max filter followed by an n-width blur will have the effect of blurring outwards. For blurring inwards, min filter should be used.

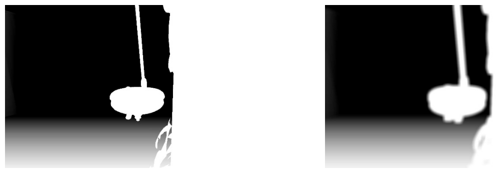

Figure 3.10. Near circle of confusion buffer (left) and its outwardly blurred variant (right).

3.3.3 Near CoC Blur

To bleed the near field onto the background, we are going to need a blurred version of the near CoC buffer. However, our blur needs to go outwards only (why this is the case will be clear later). To compute an outward blur we need to first apply a max filter to the image and then perform blur (see Figure 3.9).

Figure 3.10 shows the original near CoC buffer as well as the blurred variant of the near CoC buffer. Note that the blur indeed goes outwards.

Max (and min) filters are separable, so we can find a block's max value using two separable (horizontal and vertical) passes. Also, we perform blurring using two separable passes. All in all near CoC blur computation requires four passes.

3.3.4 Computation

The crucial part of the algorithm is the actual depth of field computation. In this pass we compute both the near field and far field and output them to two separate render targets. Both fields are computed by taking a bunch of samples in a circular pattern and computing their average.

The circular pattern used is shown in Figure 3.11. We take 49 samples using three circles. The distance between two consecutive circles (the red line)

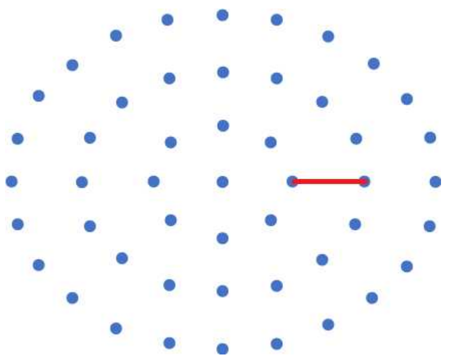

Figure 3.11. Circular sampling pattern for bokeh depth of field effect.

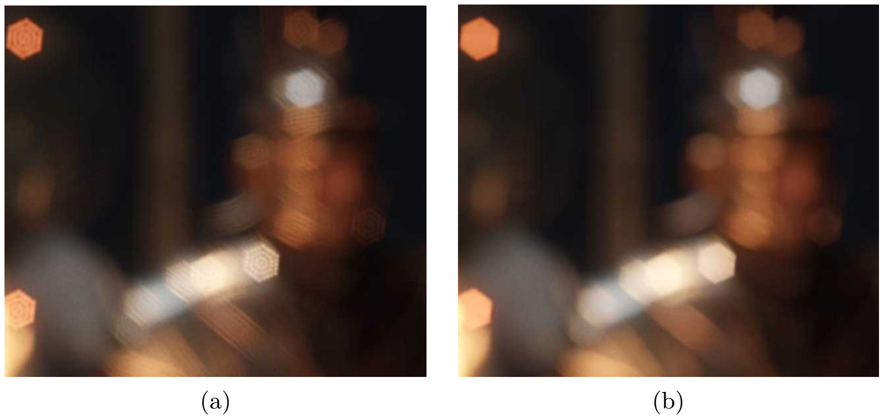

Figure 3.12. (a) Without and (b) with filling. (*Courtesy of* [Sousa 2013].)

is actually two pixels, not one. This way, the radius of the filter is larger at the cost of undersampling, as shown in Figure 3.12(a) (how to fix this will be covered in the next section). We can achieve different bokeh effects by generating other than circular sets of samples, for instance hexagonal. [Sousa 2013] presents formulas for computing various n-gons.

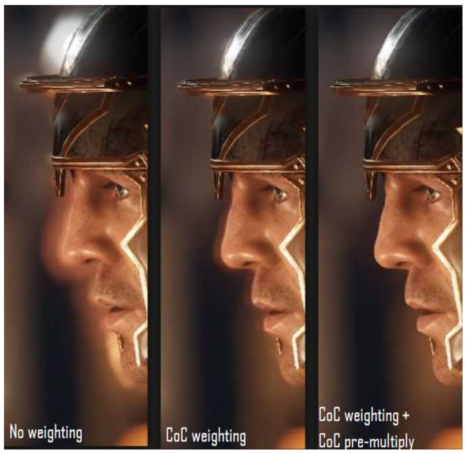

Figure 3.13. The left figure does not perform any far CoC weighting; thus, the in-focus sharp pixels bleed onto the out-of-focus background. In the middle figure, each bilinearly sampled color buffer's sample is weighted by a bilinearly sampled far CoC. By premultiplying the bilaterally filtered color buffer with the far CoC in the downsample pass all bleeding is gone in the right figure. (*Courtesy of* [Sousa 2013].)

When generating the far field each sample is weighted additionally by the far CoC value. This weighting could either be done in the shader that is computing the depth-of-field effect (using the original color and CoC buffers), or we can use the color buffer that is already premultiplied by the far CoC value. We choose the latter option. The reason that this is better is that we want to make use of bilinear filtering when sampling the color buffer, but we also want to avoid color bleeding. That is why we downsampled the color buffer in the downsample pass using bilinear filtering weighted by far CoC values. Figure 3.13 shows different variants.

In our algorithm, since we don't care that the near field will also blur the sharp background (as shown in Figure 3.5), we don't weight by the near CoC. We simply perform vanilla blurring of pixels (in circular pattern).

3.3.5 Fill

As was shown in Figure 3.12, the kernel we use leaves holes in the bokeh pattern. There are a few options to fill them. We could perform an additional 3×3 blur on both the near and far fields once they have been processed in the computation pass. However, [Sousa 2013] proposes to use the max filter instead (also 3×3 kernel). This makes the image violate energy conservation but produces a more appealing bokeh effect.

Taking 49 samples and undersampling in the calculation pass and filling thereafter with nine samples to compensate for this undersampling is much faster (almost three times) than doing only the computation pass, without undersampling, as this requires 144 samples to have the same kernel width.

Computation together with fill passes generate, in our case, a bokeh effect with a circular pattern of 13 pixels in diameter—three circles on each side, separated by two pixels that sum to six pixels on the side. This is in quarter-resolution, so eventually we end up with 12 pixels on the side in full resolution.

3.3.6 Composite

Finally, once we have generated the CoC buffers, near field, and far field and filled the latter two, it's time to composite these with the scene.

First, we sample the input color buffer without depth-of-field effect. Then, we blend the far field on top of it. Note that the far field is in quarter resolution, whereas the input color buffer is in full resolution. As such, blending the far field is a bit tricky because we have to avoid bleeding— that means that vanilla bilinear upsampling will not suffice. Bilateral filtering comes to the rescue, in a similar fashion as in Section 3.3.2, with the exception that there we used bilateral filtering to downsample and now we are upsampling. We could use the depth buffer to compare pixels' depths, but since we have the full-resolution far CoC value around and quarter-resolution buffer far CoC values we can use them. By comparing the full-resolution CoC value with each quarter-resolution CoC, we know which ones to use in the upsampling process. We then use the full-resolution far CoC value to blend between the color buffer and the upsampled far field.

Contrary to the far field, the near field can bleed as much as possible. We upsample the near field bilinearly and blend it with the color buffer (already blended with the far field) using the blurred near CoC buffer (from Section 3.3.3).

3.4 Implementation Details

We now discuss some selected implementation details.

3.4.1 Circle of Confusion Generation

The circle of confusion pass, based on the depth buffer, generates a R8G8 buffer with near and far CoCs. The depth buffer stores depth values in NDC (normalized device coordinates)in the $[0, 1]$ interval in Direct3D and $[-1, 1]$ interval in OpenGL. We convert this value from NDC coordinates back to view space and from this we compute CoC values. Listing 3.1 shows how to do the conversion (a derivation can be found in [Sterna 2013]).

```
float DepthNDCToView(float depth_ndc)
{
    return -projParams.y / (depth_ndc + projParams.x);
}
```

Listing 3.1. NDC-depth to view-space depth conversion; `projParams.x` is the projection matrix's $(3, 3)$th entry, whereas `projParams.y` is the $(4, 3)$th entry (row-major ordering).

In the demo application, a right-handed coordinate system is used; thus, the view-space depth value that is returned by `DepthNDCToView` is negative (because `projParams` values passed to the shader are based on a right-handed perspective matrix). We negate it to obtain a positive number. The CoC is then computed using linear interpolation.

3.4.2 Downsample

In this pass we downsample a couple of buffers from their full resolution, outputting to quarter-resolution buffers. For each output pixel, there are four input pixels in a 2×2 quad (see Figure 3.14). We compute their coordinates as in Listing 3.2.

Downsampling of the color buffer and CoC buffer is shown in Listing 3.3. The color buffer is downsampled using bilinear filtering. We can do so because this buffer will be used for the near field blurring, whose bleeding is desirable for us. The CoC buffer, on the other hand, is sampled using the top-left sample

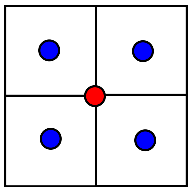

Figure 3.14. The red dot is the middle of the quarter-resolution buffer's pixel. The blue dots are centers of corresponding full-resolution buffer's pixels. If we want to sample full-resolution pixels, given the quarter-resolution pixel's coordinates), we need to shift by half of the full-resolution pixel's size in each UV axis. Or, equivalently, by 0.25 of the quarter-resolution pixel's size.

```
float2 texCoord00 = input.texCoord + float2(-0.25f, -0.25f)*pixelSize;
float2 texCoord10 = input.texCoord + float2( 0.25f, -0.25f)*pixelSize;
float2 texCoord01 = input.texCoord + float2(-0.25f,  0.25f)*pixelSize;
float2 texCoord11 = input.texCoord + float2( 0.25f,  0.25f)*pixelSize;
```

Listing 3.2. Since `input.texCoord` points in the middle of the quarter resolution buffer's pixel, we need to offset that by half the size of the full resolution pixel size, which is 0.25 of quarter resolution pixel's size. `pixelSize` is the size of the quarter resolution buffer's pixel. Figure 3.14 shows more details.

of the full-resolution buffer. In this case, we can't bilinearly filter because of the nature of the CoC data. This data is "edge sensitive" so we will need accurate data, not in-between values, to perform upsampling later on.

```
float4 color = colorTexture.SampleLevel(linearClampSampler, input.texCoord, 0);
float4 coc = cocTexture.SampleLevel(pointClampSampler, texCoord00, 0);
```

Listing 3.3. Downsampling of color and CoC buffers.

Downsampling of the color buffer for the far field requires more attention. To prevent flickering we need a bilinear filter. On the other hand, to prevent haloing (Figure 3.13) we can't do this recklessly. We can bilinearly filter, but only values that are close enough to each other. This closeness can be defined, for instance, in terms of differences in depths in the depth buffer, but since we have far CoC values lying around, which are based on the depth buffer, we can use those (see Listing 3.4).

```
float cocFar00 = cocTexture.SampleLevel(pointClampSampler, texCoord00, 0).y;
float cocFar10 = cocTexture.SampleLevel(pointClampSampler, texCoord10, 0).y;
float cocFar01 = cocTexture.SampleLevel(pointClampSampler, texCoord01, 0).y;
float cocFar11 = cocTexture.SampleLevel(pointClampSampler, texCoord11, 0).y;

float weight00 = 1000.0f;
```

```
float4 colorMulCOCFar = weight00 * colorTexture.SampleLevel(pointClampSampler,
        texCoord00, 0);
float weightsSum = weight00;

float weight10 = 1.0f / (abs(cocFar00 - cocFar10) + 0.001f);
colorMulCOCFar += weight10 * colorTexture.SampleLevel(pointClampSampler, texCoord10,
        0);
weightsSum += weight10;

float weight01 = 1.0f / (abs(cocFar00 - cocFar01) + 0.001f);
colorMulCOCFar += weight01 * colorTexture.SampleLevel(pointClampSampler, texCoord01,
        0);
weightsSum += weight01;

float weight11 = 1.0f / (abs(cocFar00 - cocFar11) + 0.001f);
colorMulCOCFar += weight11 * colorTexture.SampleLevel(pointClampSampler, texCoord11,
        0);
weightsSum += weight11;

colorMulCOCFar /= weightsSum;
colorMulCOCFar *= coc.y;
```

Listing 3.4. Downsampling of the color buffer for far field.

We take all four full-resolution buffer color samples and their CoC values. We assume the top-left sample is sort of a reference, and we will use that always. For the remaining three samples, we compute a weight factor based on the difference of their far CoC values and the reference sample. The closer the samples are to each other, the larger their contribution. Finally, the computed value is normalized (divided by `weightsSum`) and multiplied by the far CoC.

3.4.3 Near CoC Blur

Near CoC blur (the quarter-resolution, downsampled one) is required so that we can blend the blurred near field with the rest of the scene. We know from Section 3.3.4 that our effective maximum kernel radius is six pixels. This means that it would be optimal to blur, outwardly, by six pixels to each side. Technically, we are not obliged to use exactly six-pixels-wide blur. We have the background, we have the blurred near field, we can blend them the way that meets our expectations. We chose to use 12-pixels-wide blur, because the near field looks smooth enough and the artifacts from Figure 3.5 are not yet that objectionable.

To perform the (outward) blur, we first need to apply the max filter. We do this in two passes, first horizontally and then vertically, for performance reasons. The filter's width is 13 (six pixels to each side). Once we've maxed the near-CoC buffer, we blur it using a box filter (We've tried Gauss but found the difference is meaningless). Here again a 13-pixels-wide kernel is used. As a result we end up with a 12-pixels-wide (to the side) outward blur.

This part of the algorithm requires four passes—two for max and two for blur. All those passes sample only the x-component of the input buffer. On tested hardware, this is cheaper than sampling all four channels (by calling shader sample functions and not specifying which components we want to sample), even though the remaining three channels lie next to the x-channel in memory (or actually the remaining one channel because our CoC buffer has two channels).

3.4.4 Computation

To blur the near and far fields we use a circular samples pattern. The generated samples can be found in the depth-of-field computation shader `dof.hlsl`.

Blur of the (downsampled) color buffer to generate the near field is nothing fancy. We just take all samples in the circular pattern and compute the average.

Generating the far field is a bit trickier. Here we don't sample the unaltered, downsampled color buffer, but it is multiplied by the far CoC version. Code is shown in Listing 3.5.

```
float4 Far(float2 texCoord)
{
    float4 result = colorMulCOCFarTexture.SampleLevel(pointClampSampler, texCoord, 0);
    float weightsSum = cocTexture.SampleLevel(pointClampSampler, texCoord, 0).y;

    for (int i = 0; i < 48; i++)
    {
        float2 offset = kernelScale * offsets[i] * pixelSize;

        float cocSample = cocTexture.SampleLevel(linearClampSampler, texCoord + offset
          , 0).y;
        float4 sample = colorMulCOCFarTexture.SampleLevel(linearClampSampler, texCoord
          + offset, 0);

        result += sample; // the texture is pre-multiplied so don't need to multiply
          here by weight
        weightsSum += cocSample;
    }

    return result / weightsSum;
}
```

Listing 3.5. Computation of the far field.

Since the texture is already multiplied by the far CoC, the result will get darker as we move away from fully blurred areas, as shown in Figure 3.8. Combining that with the full resolution color buffer in the composite pass would result in unpleasant dark haloes around objects. A much better solution is to renormalize brightness while still in the computation pass. That's what we're doing here by dividing by `weightsSum`.

An insightful reader will quickly notice that `weightsSum` can actually be zero. This will happen for pixels that, along with their surroundings, are black. However, this is not going to be a problem as `Far` function is only called for pixels that have a far CoC larger than 0. This also, or actually most importantly, acts as a significant early-out optimization, which in this case makes great use of coherence (near field's pixels lie next to each other as well as the far field's do)—see Listing 3.6.

```
PS_OUTPUT PSMain(PS_INPUT input)
{
    PS_OUTPUT output;

    float cocNearBlurred = cocNearBlurredTexture.SampleLevel(pointClampSampler, input.
     texCoord, 0).x;
    float cocFar = cocTexture.SampleLevel(pointClampSampler, input.texCoord, 0).y;
    float4 color = colorTexture.SampleLevel(pointClampSampler, input.texCoord, 0);

    if (cocNearBlurred > 0.0f)
        output.near = Near(input.texCoord);
    else
        output.near = color;

    if (cocFar > 0.0f)
        output.far = Far(input.texCoord);
    else
        output.far = 0.0f;

    return output;
}
```

Listing 3.6. Main function of the computation pass.

We see in the listing that the early-out is applied not only to the far field but also to the near field. In the case of the far field, when depth-of-field is not computed, we output black color. We need to do this to prevent bleeding of non-far-field pixels onto the far field, which would happen in the next pass of the algorithm—fill—if this care was not taken.

In the case of the near field, to the contrary, we like bleeding and we need to output the color buffer's value. Otherwise, due to bilinear filtering, a slight black halo would appear (if we output black color) when blending the near field with the full-resolution color buffer.

As can be seen, the computation pass writes to two output buffers, near field to the first one and the far field to the other.

Since the input color buffer is a HDR floating-point buffer, it might contain large values. Usually this is not a problem unless those large values come from undersampled data, like high frequency specular. This will result in flickering bright pixels (called fireflies) in the depth-of-field effect when this effect is computed before tone mapping. This problem can be alleviated for instance by applying a Karis average, as explained in [Karis 2013]. After increasing the scene's contrast by producing larger lighting values, we found that using this trick indeed resulted in a more appealing rendered image, although a darker one.

3.4.5 Fill

In order to compensate for undersampling introduced in the computation pass, we use max filter to spread bigger values onto smaller values. The effect can be seen in Figure 3.12. We do this for both the near and far fields.

```
if (cocNearBlurred > 0.0f)
{
    for (int i = -1; i <= 1; i++)
    {
        for (int j = -1; j <= 1; j++)
        {
            float2 sampleTexCoord = input.texCoord + float2(i, j)*pixelSize;
            float4 sample = dofNearTexture.SampleLevel(pointClampSampler,
     sampleTexCoord, 0);
            output.nearFill = max(output.nearFill, sample);
        }
    }
}
```

Listing 3.7. Fill of the near field.

The kernel we need to use is barely 3 × 3, so this pass is very fast. This kernel is sufficient as the holes that are present in our bokeh pattern are one-pixel-width. Code for performing this operation for the near field is given in Listing 3.7. It is analogous for the far field.

Here we're just iterating through all samples in the 3×3 kernel and outputting the max value found. Also, to speed-up that process, we make use of early-out.

3.4.6 Composite

Finally, once we have generated the near and the far fields, we can blend them with the color buffer. First, the far field has to be blended in, and, once we have done that, we can blend in the near field. This order has to be maintained.

The shader starts by sampling the color buffer as in Listing 3.8 and stores that result in a temporary variable `result`.

```
float4 result = colorTexture.SampleLevel(pointClampSampler, input.texCoord, 0);
```

Listing 3.8. Color buffer sample.

Later on, we will want to blend the far field with the color buffer, but first we have to upsample it to full resolution. Listing 3.9 shows part of this process which is sampling all necessary textures.

First, we compute texture coordinates that will be used to sample the quarter-resolution far field. Note here that `pixelSize` used is the full-resolution pixel size. It means that when sampling the quarter resolution far field with those texture coordinates, one of the four pixels in each 2 × 2 quad will sample the same pixel in the quarter-resolution far field, two of the four will sample two different pixels, and only one will sample four pixels. This is desired since, for some full-resolution pixels, we want to have values exactly from the quarter-resolution far field and every second column and row a blend of two or four quarter-resolution far-field pixels.

```
float2 texCoord00 = input.texCoord;
float2 texCoord10 = input.texCoord + float2(pixelSize.x, 0.0f);
float2 texCoord01 = input.texCoord + float2(0.0f, pixelSize.y);
float2 texCoord11 = input.texCoord + float2(pixelSize.x, pixelSize.y);

float cocFar = cocTexture.SampleLevel(pointClampSampler, input.texCoord, 0).y;
float4 cocsFar_x4 = cocTexture_x4.GatherGreen(pointClampSampler, texCoord00).wzxy;
float4 cocsFarDiffs = abs(cocFar.xxxx - cocsFar_x4);

float4 dofFar00 = dofFarTexture_x4.SampleLevel(pointClampSampler, texCoord00, 0);
float4 dofFar10 = dofFarTexture_x4.SampleLevel(pointClampSampler, texCoord10, 0);
float4 dofFar01 = dofFarTexture_x4.SampleLevel(pointClampSampler, texCoord01, 0);
float4 dofFar11 = dofFarTexture_x4.SampleLevel(pointClampSampler, texCoord11, 0);
```

Listing 3.9. Sampling of textures needed for far field upsampling.

Later in the listing we sample the full-resolution CoC, quarter-resolution CoCs, and compute their differences which we will use as bilateral upsample weights. We use a gather instruction to sample all four CoCs at once, and this decreased the speed of the whole composite pass by 10%. Finally, we sample the far field in four different coordinates.

We have the far field sampled in correct coordinates and `cocsFarDiffs` that will act as weights for upsampling. We will now calculate vanilla bilinear filtering weights that will be used to blend all four far-field samples. Code for this is presented in Listing 3.10 and is based on [Wikipedia 2017].

```
float2 imageCoord = input.texCoord / pixelSize;
float2 fractional = frac(imageCoord);
float a = (1.0f - fractional.x) * (1.0f - fractional.y);
float b = fractional.x * (1.0f - fractional.y);
float c = (1.0f - fractional.x) * fractional.y;
float d = fractional.x * fractional.y;
```

Listing 3.10. Bilinear filtering weights.

We now have weights for bilinear filtering. From previous code, we have CoC-aware weights. We can now combine the two to compute the upsampled bilateral far-CoC value, as shown in Listing 3.11.

```
float4 dofFar = 0.0f;
float weightsSum = 0.0f;

float weight00 = a / (cocsFarDiffs.x + 0.001f);
dofFar += weight00 * dofFar00;
weightsSum += weight00;

float weight10 = b / (cocsFarDiffs.y + 0.001f);
dofFar += weight10 * dofFar10;
weightsSum += weight10;

float weight01 = c / (cocsFarDiffs.z + 0.001f);
dofFar += weight01 * dofFar01;
weightsSum += weight01;

```

```
float weight11 = d / (cocsFarDiffs.w + 0.001f);
dofFar += weight11 * dofFar11;
weightsSum += weight11;

dofFar /= weightsSum;

result = lerp(result, dofFar, blend * cocFar);
```

Listing 3.11. Far field upsampled value computed.

The final `dofFar` value is composed of the four samples using the same formula we saw during downsampling so there is really nothing new here. At the end of the code, we blend (lerp) the color buffer (stored in `result`) with `dofFar` using the full-resolution far CoC, multiplied by `blend` which is a user-specified constant passed to the shader from the application.

The hard part of composition, the far field, is done. Blending in the near field is way easier, as shown in Listing 3.12.

```
float cocNear = cocBlurredTexture_x4.SampleLevel(linearClampSampler, input.texCoord,
     0).x;
float4 dofNear = dofNearTexture_x4.SampleLevel(linearClampSampler, input.texCoord, 0);

result = lerp(result, dofNear, blend * cocNear);
```

Listing 3.12. Blending in the near field.

We just upsample the near field and blend it with the current result (color buffer blended with the far field), using plain hardware bilinear filtering. It is advisable [Sousa 2013] to use custom bicubic filtering here for better quality. The `blend` parameter will be explained in the next subsection.

3.4.7 Application

Up till now, we have described mostly what takes place on the shader side of the effect. However, the discussion would not be complete without mentioning some subtleties that take place on the application side.

The header of the function that performs the whole effect is shown in Listing 3.13.

```
void DOF(float focalPlaneDistance, float focusTransitionRange, float strength, const
     Matrix& projTransform)
```

Listing 3.13. Header of the DoF function in the application.

The function takes the description of the focal plane. Those values are used to compute variables from Section 3.2 which are later passed to the CoC generation shader.

The value `projTransform`, which is passed as the last argument, is just the perspective projection transform. It is used to properly convert NDC depth values to view space values.

The parameter `strength` says how strong the DoF effect should be. Actually, even though it may not seem so at first, it's not that obvious how that should work. We will discuss that now.

At the beginning of the `DOF` function there are two variables—`kernelScale` and `compositeBlend`. The variable `kernelScale` is used to scale the circular kernel used in the computation pass. We can use it to control the radius of the effect; `compositeBlend`, on the other hand, is used to multiply the CoCs in the composite pass (but only during blending with the color buffer). While it is obvious what the former does, the latter usage might be a bit vague. The kernel scale is useful if we want the application to fade-in or fade-out of the DoF effect. The problem is, however, that if we set it to 0 (or, in general, some small value) the quarter-resolution nature of the fields will become apparent. When the kernel scale is 0, we would like to completely fade out the fields, displaying the original color buffer. This can be achieved with `compositeBlend`. We choose that, if the DoF's strength falls below 0.25, the kernel scale is clamped to 0.25 and the composite blend is changed so that it fades from 1 to 0 (leaving only the original color buffer). Code that does just that is shown in Listing 3.14. Also, code that uses `compositeBlend` is in the last lines of Listings 3.11 and 3.12.

```
float kernelScale = 1.0;
float compositeBlend = 1.0;

if (strength >= 0.25f)
{
    kernelScale = strength;
    compositeBlend = 1.0f;
}
else
{
    kernelScale = 0.25f;
    compositeBlend = 4.0f * strength;
}
```

Listing 3.14. Calculation of kernel scale and composite blend based on DoF strength.

3.5 Per-pixel Kernel Scale

You might have noticed that in the DoF computation pass we don't scale the filter kernel by the pixel's CoC value. That would actually be more natural, and some depth-of-field implementations do that. We went on to implement that modification and found that even though it produces a slightly more realistic DoF, it also produces a kind of undesirable near field, related to the core problem of our algorithm depicted in Figure 3.5. The far field does not have that problem, and it might be visually beneficial to apply per-pixel kernel scaling for it.

To change the current implementation to accommodate for per-pixel kernel scale, you need to add multiplication by pixel's CoC in the sample texture coordinates offset calculation. Also, the filling pass has to be changed slightly. In that pass, we didn't care about depth discontinuities as they were not problematic for us. When per-pixel CoC kernel scaling is applied, we must not use neighboring samples whose CoC is much different than that of the pixel that is being processed.

Once these changes have been applied, we can alter blending factors in the composition pass. To better show off the effect of scaling, we could output the far field without blending with the sharp color buffer for a broader specturm of far CoC values. One just has to remember that the far field is actually of quarter resolution, and once the kernel scaling goes down, this fact will be revealed more.

All in all, given the problems related to per-pixel kernel scaling we decided to not use it at all in the demo application.

3.6 Demo Application

There is an accompanying demo application to this chapter presenting the algorithm in action.

Configuration of the demo can be changed in `config.txt` file located in the folder where the binary is located. The key configuration is

- WSAD + mouse—camera movement,
- Shift— speeding up,
- Insert / Delete—shift focal plane,
- Home / End—increase/decrease in-focus region,
- Page Up / Page Down—increase/decrease DoF effect,
- ESC—exit.

Pass	**Time**
All	1.506 ms
CoC generation	0.13 ms
Downsample	0.145 ms
Near CoC Max X	0.08 ms
Near CoC Max Y	0.08 ms
Near CoC Blur X	0.08 ms
Near CoC Blur Y	0.08 ms
Computation	0.455 ms
Fill	0.122 ms
Composite	0.334 ms

Table 3.1. Performance on GeForce 660 GTX in 1080p.

In the folder where the binary is located, a file called `profiler.txt` will be generated upon exit, where average performance of each consecutive batch of 100 frames is dumped. Table 3.1 shows performance on a GeForce 660 GTX in 1080p.

3.7 Conclusions

In this chapter we presented an efficient, production-quality algorithm for generating depth-of-field effect. It produces stable, convincing results at very good frame rates.

3.8 Acknowledgments

I would like to thank Krzysztof Narkowicz of Flying Wild Hog for taking the time to read the chapter and to provide feedback. I also thank Wessam Bessam, the editor of this section, for smooth cooperation and, of course, Wolfgang Engel, the editor of the entire book, for his editorial work starting with the early *ShaderX* books up to this one :).

Bibliography

Jimenez, J., 2014. Next generation post processing in *Call of Duty: Advanced Warfare*. URL: http://www.slideshare.net/guerrillagames/killzone-shadow-fall-demo-postmortem.

Karis, B., 2013. Tone mapping. URL: http://graphicrants.blogspot.com/2013/12/tone-mapping.html.

Pettineo, M., 2011. How to fake bokeh. URL: https://mynameismjp.wordpress.com/2011/02/28/bokeh/.

Scheuermann, T., and Tatarchuk, N. 2003. Improved depth of field rendering. In *ShaderX3: Advanced Rendering with DirectX and OpenGL*. Charles River Media, Boston, MA, pages 363–377.

Sousa, T. 2013. CryEngine 3 graphics gems. In *Advances in Real-Time Rendering SIGGRAPH 2013 course*, ACM, New York, pages 22–44. URL: http://www.crytek.com/download/Sousa_Graphics_Gems_CryENGINE3.pdf.

Sterna, W., 2013. Reconstructing camera space position from depth. URL: http://wojtsterna.blogspot.com/2013/11/recovering-camera-position-from-depth.html.

Valient, M., 2013. Killzone Shadow Fall postmortem. pages 77-83. URL: http://www.slideshare.net/guerrillagames/killzone-shadow-fall-demo-postmortem.

Wikipedia, 2017. Bilinear filtering. URL: https://en.wikipedia.org/wiki/Bilinear_filtering.