Implementing realistic depth of field in OpenGL

Filip Nilsson* Philip Ljungkvist†

Lund University
Sweden

Abstract

In recent years the post-processing effects in games and simulations have increased by a lot, both in number of effects and the quality of them. This is due to the increased power of the graphic cards and the never-ending demand from the public for cinematic-looking games. One of these effects is called depth of field, or DOF.

In most photos theres a focus point, something the photographer wants the viewer to look at. This is often achieved by making that point or object sharp, and everything out of focus unsharp. In a camera this is caused by a combination of a number of factors, including the lens, aperture and distance to the focus point. The region that is sharp is called the depth of field. The depth of field effect can also be seen by the human eye, so even if you do not want depth of field for a cinematic look in your game you most likely want it for the realistic look.

One phenomena that you will see with both your eyes and with a camera is the Bokeh effect. This effect is as most visible when looking at an unsharp light point and it will appear as a certain shape. In a camera this is a side effect from the cameras aperture where the shape of the aperture will determine the shape of unsharp points of lights in the image, but is now a popular part of photography. Since the shape of the Bokeh effect is based on the aperture, for the human eye light spots will appear as circles, while a camera will make it look like a polygon.

In computer graphics there are some key problems when implementing DOF. The first one is that by default colour information is encoded in a nonlinear format. The result of this is that blurred parts dont retain highlights. Another problem is that in-focus foreground objects colours bleed into the background. Finally the last problem occurs when there is a blurred object in front of a sharp background but the edge between these two is sharp instead of blurred out.

Using our solution we have implemented steps to either solve or minimize these problems.

1 Introduction

The gaming industry today wants to create as cinematic-looking games as possible. Weve seen a rise in games where youre practically going through a movie but controlling the main character. One important aspect of the cinematic look is depth of field. Already a well-documented phenomena in photography, graphic programmers have been able to create some very realistic effects in games and animated movies.

Depth of field is not a new phenomena in games thought, even if its more prominent these days. One of the first games to use the effect is Outcast, released in 1999, but it was still a rare treat for another half a decade. With the release of a new generation of game consoles and the rise of new game engines like Unreal Engine 3 depth of field is now a common effect featured in a lot of game.

Our aim with this project is to create a realistic depth of field that will work when unsharp objects are both in front and behind the plane in focus. We want a prominent Bokeh effect to further add to the realistic look and make the light points stand out even when blurred.

*e-mail: ada09fni@student.lu.se
†et07pl3@student.lth.se

Figure 1: Crysis 2 (2011) showcasing heavy DOF. Also notice the circular Bokeh effect of the sparks and lights.

2 Algorithms or Application

The amount of blur a certain point, or pixel, will have in a game is just like in photography depending on a number of variables. We have chosen to implement a function using real optical laws to create an as realistic effect as possible. The function returns a value representing the Circle of Confusion, or CoC. The value will later be used to determine within what radius around the pixel we want to sample points to determine the color of the pixel. In other words, a large CoC means that the point will be more blurred. The function is as following [Demers 2004]:

$$CoC(D) = |A * \frac{F * (P - D)}{D * (P - F)}| \quad (1)$$

Variable	Name	Our value
A	Aperture	20
F	Focal length	??
P	Plane in focus	
D	Object distance	
I	Image distance	??

The focal length is calculated based on the plane in focus and image distance with the following equation:

$$\frac{1}{P} + \frac{1}{I} = \frac{1}{F} \quad (2)$$

When looking at a graph of the function we see that points will blur differently depending on if they are in front of the plane in focus or behind.

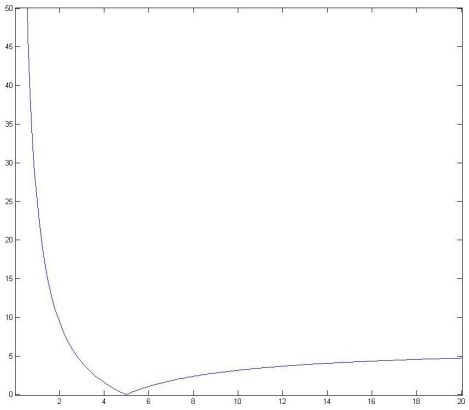

Figure 2: The CoC function plotted in Matlab with a fixed Plane in focus

2.1 Aperture

Since the camera in a computer generated scene dont have a physical aperture this is a value we chose our self. A lower value would mean a wider depth of field or range of focus. We chose a quite high value to better show the effect of depth of field. Object distance The object distance is the distance to the point or pixel the fragment shader will colour. To find out this value we look at the depth buffer already available from when creating the geometry.

2.2 Plane in focus

This is the distance between the camera to where the scene should be the sharpest. We implemented a dynamic system to automatically set the distance to the surface the mouse cursor pointed at. In most games with DOF you either have a constant value, or let a certain point on the screen always be the focus, like in the middle where your aim is. To find the distance to the surface we again look at the depth buffer.

2.3 Image distance

This is the distance between the lens and the film/sensor in a real camera. This value depends on the scale of the scene. We have set it to a value of 1.0.

2.4 Focal length

In a real camera the focal length is a measurement of how strongly light converges or diverges. It depends on the distance to the plane in focus and image distance.

2.5 Sampling

With a radius around the pixel mapped out by the previous function, we can now start looking at the colours of the pixels around it. There are a number of ways to do this, and we will mention three. They all use two for loops, one depending on rings and one depending on segments. For each ring starting from either the center pixel or near it ranging out to the outer radius of the CoC, the shader will sample the number of segments specified.

The first method will sample in a square around the pixel. This is the easiest method to implement, but also gives the worst results in terms of looks.

The second method will first only blur vertically, and then only blur horizontally. This will make the program run a lot faster since the sampling is now done in linear time instead of exponential. However, the looks will take a hit. This method is still used in a lot of games.

The last method is picking points in a number of circles around the pixel. This will require more computational power, but will overall give a better look. This is the method we used. No matter what method is used, once all the samples have been added the shader will have to divide with the number of samples to get the average colour.

The sampling will have to be run for both the light buffer and the texture buffer. Luckily, this can and should be done in the same for-loops.

This is the basics of a DOF implementation and will create an image where you have a region in focus and everything behind and in front blurred out. However, this implementation will suffer from a number of artifacts and problems.

2.6 Artifacts

2.6.1 Circle of confusion

Starting with the circle of confusion, if we take a look at the graph we will see that the size of it for objects close to the camera will increase towards infinity if we look at something far away. Likewise, even though the functions growth starts to diminish the further from the focus point the point is, we will get unpractical sizes of the CoC. To fix this without changing the results for all other cases, we can let the shader clamp the value of CoC to a maximum before using it in further computations. We chose a value of 20.

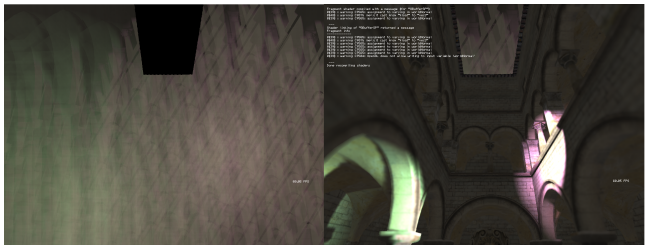

Figure 3: Two cases where the circle of confusion is too big

2.6.2 Bleeding artifacts

One other artifact is called the bleeding or halo artifact. This problem will appear when a sharp object is in front of an unsharp background, and is as most prominent when the relative depth difference is bigger, often in close-ups. The background pixels close to the edge between the background and foreground will have a big CoC and therefore sample points from not only the background but also the foreground. This will cause the colours of the sharp foreground object to bleed out into the background. Once again there are a number of ways to counter this problem. One could first only render the background, blur it and then add the sharp foreground on top of it. This would produce the most accurate results but would be very computation heavy and unfeasible for most real-time applications. The two other methods are very similar and only differs in one detail. If we compare the depth between the sample point and center point we can detect if theres a big difference in depth and not add that sample point to the average value of the pixel. This is one of the options, but will cause the average to be shifted more then necessary away from the edge between the background and foreground since most samples will be on that side of the center pixel. A better solution is to add the center pixels sample instead if a sample point is in the foreground. This will still cause shift, but much smaller. Getting rid of the shift completely would require us to know the color of the points obscured behind the foreground object which brings us back to the first, impractical, solution.

2.6.3 Sharp unsharpness

If we instead look at a similar case, where the background is sharp and foreground unsharp we will see another artifact. The blurred out foreground will have a clear and distinct edge making the DOF

Figure 4: A scene before and after we've fixed the bleeding artifact

effect look very unrealistic and cheap. In reality the edge should be blurred out the same amount as the object in the foreground. The cause of this artifact is similar to the bleeding artifact, but is instead caused by the pixels in the background near the edge not sampling the pixels of the foreground object. This problem is solved by blurring all CoC values before using them to blur the light and textures.

Figure 5: A scene before and after we've fixed the sharp unsharpness artifact

2.6.4 Light intensity

Normally images one a computer are stored in a nonlinear colour format known as gamma-encoding. With this encoding dark colours are encoded using more bits than bright colours. Since the human eye is more sensitive to changes in darker colours this means that a higher quality image can be stored using fewer bits per pixel than if linear colour was used. When averaging the samples from the CoC this encoding is a problem. Because of the nonlinear encoding darker colours have more influence than highlights on the result of the averaging. This problem is solved by storing the result of the light calculations in linear floating-point colour and gamma encoding the colours in the last step of rendering where the pixels are written to the framebuffer.

3 Results

When we first implemented the naive blur that only blurred based on the pixels depth and nothing else we did not see any performance hit. Once we fixed the artifacts by taking the steps described in the section above, the frame rate went down by a lot, from 60 to as low as 8. This was mainly due to a lot of texture look-ups in the depth map when we compared each samples depth with the center pixels depth, but also some other factors. The extra texture look-ups cant be optimized without using a completely different method as far as we know, but we did manage to get the frame rate to what we think are acceptable levels for this application by other optimisations. Here are the biggest ones:

Using less samples for pixels with a smaller CoC value. Instead of setting an constant value of samples for each ring independent of the size of the ring we let it scale with the radius of the circle of confusion. This made the sampling a bit more dynamic and gave us a higher frame rate in most cases, especially in images with less or moderate blur where we could gain as much as 10 frames per second. This change also makes sure the Bokeh shape is always solid. Before big Bokeh shapes would be made up out of a number of spread out dots instead because the sample points just werent enough. It does give us a lower frame rate in some extreme cases, where most of the screen is very blurry since most pixels will have a very high CoC and therefore a lot of sample points, but as well explain later we think thats acceptable.

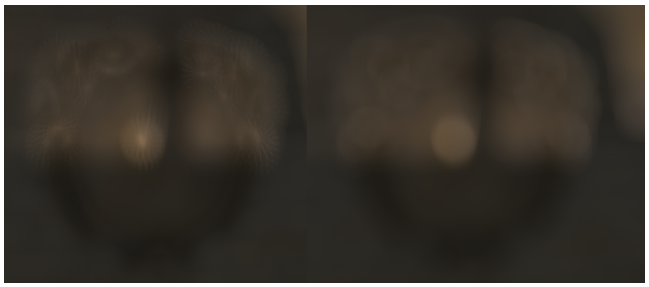

Figure 6: A scene with constant versus dynamic number of samples per ring.

4 Discussion

As we wrote in the results section the performance got worse in some extreme cases by adding dynamic sampling. These cases are what photographers would call macro perspectives, extreme close-ups, with extreme amount of blur. We might be able to remedy this by adding more checks and making the implementation even more dynamic, but this might at the same time worsen the performance for non-macro perspectives. Since this implementation is meant for real-time applications such as games the macro perspectives would be very rare by default, and could be avoided completely by a number of solutions. The easiest being clamping the CoC function at a lower bound then before making the blur look more unrealistic, or clamping the amount of samples one ring can have, but then you might lose the solid Bokeh shape, which is why we opted not to do it for this demo.

In the so called macro perspectives the bleeding effect will sometimes appear again, despite the steps we took to minimize that artifact. This is because even the smallest depth difference will be relatively big in the depth buffer, so if the focus isnt on the very outer edge of the foreground object that point will also be out of focus and therefore should be blurred according to the normal rules. Again, this problem is only for those very rare macro perspectives and so we chose not to make it more dynamic as it would affect the performance of the other cases. Even still, getting the dynamic system to work and counter all the artifacts, for all different angles and perspectives but the macro, was easily the most challenging part of this project.

One of the things that worked well in this project was all the different buffers. It was easy to get the information needed for each pixel, and writing the sampling code was very straight forward. Overall, the code for each part of the solution was easy to implement once we had an clear plan of what we wanted to do to solve the artifacts.

References

DEMERS, J., 2004. Depth of Field: A Survey of Techniques. http://http.developer.nvidia.com/GPUGems/gpugems_ch23.html. [Online; accessed 12-Dec-2012].