Abstract

This seminar thesis focuses on the emulation of various effects and imperfections of a physical camera with OpenGL. In the filed of computer graphics the camera only exists in a very abstracted concept used to project the virtual environment on a two-dimensional plane, usually the screen. Therefore, none of the imperfections, which all real cameras are prone to, affect them. This especially applies to traditional rasterizers. Depth of field, chromatic aberrations and vignetting, seen in movies or photography, are very familiar effects and can play a significant part in a realistic depiction of a virtual environment. An image without these imperfections might appear sterile to the viewer. Several methods and techniques are presented, partially implemented, and compared in terms of quality and performance.

Contents

Acronyms

1 Introduction

1.1 Goal

Cameras are complex devices with many variables and moving parts. Not every detail about them or the physics behind them can be explained. Instead, only a select few properties are observed more closely. The example implementation provided alongside this thesis is meant to demonstrate how the effects and imperfections of a physical camera can be emulated and applied in a minimal set-up. The example set-up is the main focus of this thesis. It is not meant to be robust or portable, since these requirements would increase its complexity beyond the scope of this thesis.

The render-pipeline does not support High Dynamic Range (HDR). Effects like tone-mapping, bloom or lens-flares, which also fall under the category of camera-effects, are not part of the example implementation or thesis. Motion-blur, a temporal camera-effect, is also omitted, due to its complexity. The following effects are discussed in detail:

- Depth of Field (DoF)
  - Basic implementation
  - Advanced implementation
  - Bokeh-Shapes
- Barrel distortion
- Chromatic aberration
- Vignetting

They are not implemented with physical accuracy in mind. Since real-time environments employ a series of restrictions onto the applications, computationally expensive effects like DoF can only be approximated. The aim is to implement a post-processing pipeline, which creates plausible and visually appealing results.

2 Depth of Field

The depth of field effect causes only objects at a certain distance from the camera to be in focus. That distance is also referred to as the focus distance or focus plane. The farther an object is from the focus distance, the stronger it will be blurred. The effect "[...] is frequently used in photography and cinematography to direct the viewer's attention within the scene, and to give a better sense of depth within a scene." [Dem] Depth of field can be observed when light is passing through an optical system such as a camera lens. "In order for that light to converge to a single point on the film [...], its source needs to be at a certain distance from the lens." [Dem]

2.1 Circle of Confusion

Light rays originating from either side of the focus plane will converge on the film in an area referred to as the circle of confusion. The further an object is away from the focus distance, the larger the circle of confusion will be (Figure 2.1). However, there are more parameters affecting its size like the focal length and aperture of the lens. The Circle of Confusion (CoC) can be calculated with the following equation 2.1 [Dem]:

$$CoC = \left| \frac{a * (d * p)}{d * (p - f)} \right| \tag{2.1}$$

a denotes the aperture diameter, d the distance of the object in question, p the focus distance and f the focal length. These parameters can be provided by the virtual camera and the depth buffer. With the distance from the camera for each pixel, the CoC can be computed and used to blur the original image accordingly.

2.2 Implementation

2.2.1 Accumulation DoF

Unlike the other discussed approaches, accumulation DoF is not a post processing effect. In order to achieve depth of field, the camera itself is moved around the focus point based on the aperture size and an image is rendered. This process is repeated several times, accumulating every image in a frame buffer, as the name already implies. This approach has access to information about obstructed objects, which is usually not the case for post-processing effects. In a way it is similar to properly ray tracing depth of field and therefore, given enough iterations, the most accurate way of calculating the effect. However, there are several downsides to this process. In order to achieve a high quality result without any noticeable artifacts, many iterations are necessary. The amount of iterations needed also increases with the desired blur/aperture size. Considering the entire scene has to be rendered sometimes up to a hundred times for a single frame, this approach is not feasible for real-time applications. While the implementation is not particularly difficult, it is incompatible with most modern rendering pipelines based on deferred shading. The frame-buffers containing normals, positions or other pieces of information about the scene for deferred shading, would not only increase the accumulation workload, but are also unfit for accumulation.

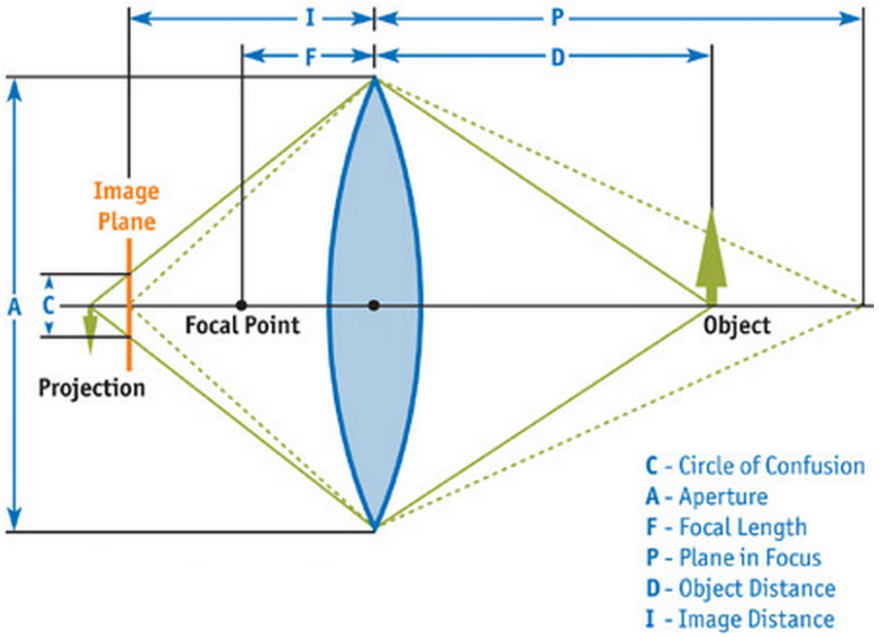

Figure 2.1: Circle of Confusion visualized for a thin lens. [Dem]

2.2.2 Scatter DoF

The scatter method is the most intuitive implementation of post-processing depth of field. For each pixel the circle of confusion will be used to determine the radius of the blur effect. The color of the pixel is then scattered across all the pixels within said radius. This implementation is unsuitable for usage in a fragment shader on a Graphics Processing Unit (GPU). Shaders make use of heavy parallelization and cannot have any side effects, meaning no pixels can be altered, but the one the shader is tasked to calculate at a given time. Compute shaders on the other hand might be suitable for a scatter DoF implementation, but require a OpenGL version of 4 or above and were not explored in the provided implementation. Scatter based methods are usually used for non-real-time applications like 3D-software or compositing-software. A simple Central Processing Unit (CPU) based implementation is used as a reference for other methods later on.

2.2.3 Gather DoF

Gather methods are the most commonly used implementations of a depth of field effect in real-time graphics applications. The process used for scatter DoF is essentially reversed due to the aforementioned restrictions of fragment shaders. In order to calculate the color of a given pixel in a shader, every surrounding pixel within a predefined maximum blur-size has to be taken into account. This is computationally intensive, since each pixel within the search radius might contribute to the resulting color based on its own CoC. [Gus]

The fundamental set-up to use any of the post-processing effects requires the ability to render the original scene into a texture, which is accessible from a fragment shader. In addition to the fully shaded image of the scene, depth information is needed in order to calculate the CoC. The example implementation uses a deferred rendering pipeline, which conveniently provides a texture containing positional data for each pixel. This can be used to extract the z-component in linear space. For non-deferred renderers, the depth information can be obtained by copying the device depth-buffer into a texture and transforming it to

linear space.

Basic gather DoF The basic gather DoF in this thesis refers to a very naive implementation and does not employ any techniques to mitigate artifacts caused by a gather-based approach. It accumulates the colors of a random pixel within the search radius, directly derived from its own circle of confusion. This works surprisingly well as long as no out-of-focus object is close to the camera. A random sampling strategy causes the resulting blur to be noisy at low sample counts.

```
out vec3 color;
in vec2 TexCoords;
uniform sampler2D ColorPass;

float getBlurSize(float depth) {
   // Calculate the blur based on equation 2.1
}

vec2 rand(vec2 seed) {
   // Random vector based on IEEE floating point rounding
}

void main() {
    float centerDepth = texture(linearDistance, TexCoords).r;
    float centerBlur = getBlurSize(centerDepth);
    vec3 color = texture(ColorPass, TexCoords).rgb;
    for (int i = 0; i < iterations; i++) {
        vec2 offset = rand2(TexCoords + i) * centerBlur;
        vec2 uv = TexCoords + offset * pixelSize;
        color += texture(ColorPass, uv).rgb;
    }
    color /= float(iterations);
    FragColor = color;
}
```

Advanced gather DoF The implementation is directly based of Dennis Gustafssons work [Gus], known for his work on interactive destruction mechanics in Smash Hit and Teardown. It addresses several issues the basic gather method suffers from, while maintaining a similar level of complexity. Instead of sampling random surrounding pixels, it starts from the center pixel and works its way outwards in a spiral pattern until it reaches a predefined maximum blur size. It considers the depth and circle of confusion for each of the sample pixels, which introduces additional texture look-ups. The CoC is used to determine the influence the sample has on the center pixel.

```
void main() {
  float centerDepth = texture(linearDistance, uv).r;
  float centerBlur = getBlurSize(centerDepth);
  vec3 color = texture(shadedPass, uv).rgb;
  float steps = 1.0;

  for (float ang = 0.0; radius < MAX_BLUR_SIZE; ang += GOLDEN_ANGLE) {
    // Convert the polar coordinates and consider the pixel size
    vec2 offset = vec2(cos(ang), sin(ang)) * pixelSize * radius;

    // Gather the color, depth on coc of the sample pixel
    vec3 sampleColor = texture(shadedPass, uv + offset);
    float sampleDepth = texture(linearDistance, uv + offset).r;
    float sampleBlur = getBlurSize(sampleDepth);

    // If the sampled pixel is behind the center pixel
    // clamp the blurSize to the center blur size
    // This prevents the background form blurring over
    // the foreground
    if (sampleDepth > centerDepth) {

      sampleBlur = clamp(sampleBlur, 0.0, centerBlur * 2.0);
    }
    // Scale the blur size to 0.0-1.0 based on the current radius
    float m = smoothstep(radius - 0.5, radius + 0.5, sampleBlur);

    // Mix the current average color with the current sample
    color += mix(color / steps, sampleColor, m);
    steps += 1.0;

    // Increase the radius, progressively slowing down
    radius += RAD_SCALE / radius;
  }
  FragColor = color / steps;
}
```

Multi pass A extension to the gather DoF method divides the image into multiple passes.

- The near plane contains all pixels, which are closer to the camera than the focus distance.
- The far plane contains all pixels farther away than the focus distance.

The contents of both textures will be out of focus and therefore blurred to a certain degree. Since these passes are both separate from the main pass and will mostly contain blurred, low frequency detail, they can be down-scaled. This reduces the work required to blur the contents of the image significantly and areas, which are neither on the foreground or background plane can be omitted. This offers a important performance benefit over single-pass gather methods, since they cannot take advantage of any early-out optimizations.

2.3 Bokeh

Bokeh describes the visual quality of the blur and how it behaves in certain situations.

2.3.1 Aperture blades

While the presented DoF methods use different sampling strategies, they all sample within a circular pattern or disk. The aperture of a real lens consists out of multiple blades, which allow the size of the aperture to change. If the lens has fewer blades, the shape of the aperture isn't perfectly round. For example this is visible when closely examining the blur shape of a bright light source.

$$r = \frac{\cos\left(\frac{\pi}{n}\right)}{\cos\left(\left(\theta \mod \frac{2\pi}{n}\right) - \frac{\pi}{n}\right)} \tag{2.2}$$

The formula 2.2 [Cou] can be used to map the polar coordinates of a circle to a circular shape with n edges. The advanced gather shader can simply be extended to include this functionality by swapping out line 9 with the following snippet.

```
1 float r = radius * cos(PI / n);
2 r /= cos(ang - (2.0f * PI / n) * floor((n * ang + PI) / 2.0f / PI));
3 vec2 offset = vec2(cos(ang), sin(ang)) * pixelSize * r;
```

The radius is now altered based on the current angle, which is then converted from a polar coordinate system back into the Cartesian coordinate system where the offset can be used. The left image in figure 2.2 shows the altered sample distribution for a bokeh-shape with five edges.

2.3.2 Cats-eye Bokeh

This term refers to the bokeh shape at the edge of an image. The effect is very apparent in the real life example in figure 2.2 (right). The center bokeh shape is mostly round, while the other light-sources closer to the edge get distorted. In order to achieve this effect, the sampling shape has to be scaled around an axis pointing towards the center of the image. This is done using a matrix which

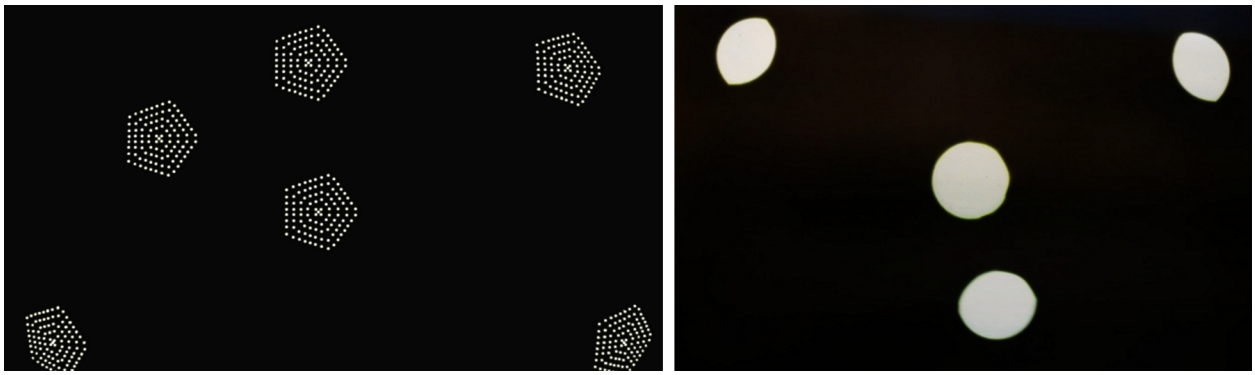

Figure 2.2: Changing blur shapes towards the edges of the image. Sample distribution based on screen position with five aperture blades (left) and a real world example using a 60 mm $f/2.0$ lens with seven aperture blades(right).

- Rotates the sample point, so it is aligned to the axis it has to be scaled to.
- Scales the sample point.
- Rotates it back to its initial position.

```
// get the angle for the vector to scale along
float angle = acos(dot(vec2(1.0, 0.0), normalize(fromCenter)));
// flip the lower half so it scales symmetrically
if (fromCenter.y > 0.0) { angle = -angle; }
// 3. rotate it back in place
mat2 squeeze = rotate2d(angle);
// 2. scale the shape on the x-axis
squeeze *=  mat2(1.0 + squeezeStrength, 0.0, 0.0, 1.0);
// 1. rotate to align to the x-axis
squeeze *= rotate2d(-angle);
```

This matrix is applied to each of the sample points, resulting in the sample distribution seen in figure 2.2 (left). The implementation using this approach is referred to as bokeh gather in this thesis and is not based on Dennis Gustafssons work (advanced gather DoF) as it differs in some other aspects as well. The result of this effect can be seen in figure 2.3.

2.4 Comparisons

2.4.1 Performance

Depth of field is an effect, which contributes significantly to the overall visual quality of a computer generated image. However, there are many more steps in a graphics pipeline before the image can be presented on a screen. Many of them more important than the DoF effect. To ensure only a reasonable amount of time is spent for this effect, several tests were conducted. The table 2.1 lists a comparison between different approaches and sample counts. The frame-time shows the average over a period of 20 seconds on an AMD RX Vega 56 GPU at a resolution of 1280 by 720 pixels. Without post processing depth of field the example set-up runs at roughly 1020 Frames per Second (FPS), which translates

Figure 2.3: Example of the bokeh gather blur and the cats eye effect.

to a frame-time of 0.98 ms. Comparing that number to either of the methods in table 2.1, the impact on performance becomes apparent. Even the very basic gather method cuts the FPS in half. This is not due to any expensive mathematical calculations, but rather due to texture look-ups. At 64 samples there are already 66 look-ups. One to get the color of the pixel at the center, one to get the depth of the pixel and one for each of the samples gathering colors from surrounding pixels. The other methods also take the depth of the samples into account, which almost doubles the amount of total texture look-ups.

In order to reduce these, rendering the depth of field to a lower resolution texture would be an option. This is why multi-pass DoF methods are used in most game-engines, even though they increase the complexity of the rendering pipeline by adding two additional textures.

Method	Frame-time at 64 Samples	Frame-time at 512 Samples
Basic gather	1.92 ms	8.13 ms
Advanced gather	2.01 ms	8.85 ms
Bokeh gather	2.13 ms	7.70 ms

Table 2.1: Performance comparison between the three implemented DoF methods.

2.4.2 Visual quality

All three compared methods are similar in performance, but the resulting visual quality greatly differs. The difference becomes apparent at lower sample counts, since different sampling strategies were used. The basic gather method produces a noisy image, since it uses random sample points to gather color information from the surrounding pixels. The advanced gather uses a spiral sampling pattern starting from the center, resulting in banding artifacts when using few samples per pixel. Bokeh gather behaves in a similar way but also introduces a noticeable noise pattern based on the bokeh shape.

Even with high amounts of samples none of these approaches are truly free of any artifacts. There are two major challenges for a post-processing depth of field effect.

Blurring partially obstructed objects: Pixels with a large circle of confusion may sample surrounding pixels, which are closer to the camera than itself. If the surrounding pixels happen to be in focus, color bleeding onto the center pixel will occur. The basic gather method suffers from this artifact as seen in Figure 2.4 where the yellow post bleeds onto the darker background. To prevent this, the shader could reject samples, which are close to the camera than itself. This is not physically correct but usually indistinguishable from the ground truth. The reference scatter and advanced gather method (Figure 2.4 Top middle and bottom middle) use this approach to some extent. The bokeh gather method doesn't bleed color onto the background but suffers from a halo effect around the edges. As the background pixels get closer to the yellow post, the blur size gradually decreases. The advanced gather shader is visually identical to the reference implementation and maintains pixels perfect accuracy around the edges of the object.

Blurring objects closer to the camera than the focus distance: Achieving a smooth transition between the blurred foreground and the focused background poses an even more difficult challenge than blurring partially obstructed objects. All three compared methods in Figure 2.5 show visible artifacts around the edges. The basic gather method (Figure 2.5 left) does not attempt blur the yellow post over some of the background because it only takes its own circle of confusion into account and blurs itself accordingly. The advanced gather method blurs the object beyond its bounds and creates a very good result. There are only small areas of the original unblurred object visible, similar to the bokeh gather method, which suffers far more from this artifact. Hiding the original object requires the blurred version to occupy a greater area, which is not physically accurate. However, this is the only option in a post processing effect, which is unaware of the obstructed object.

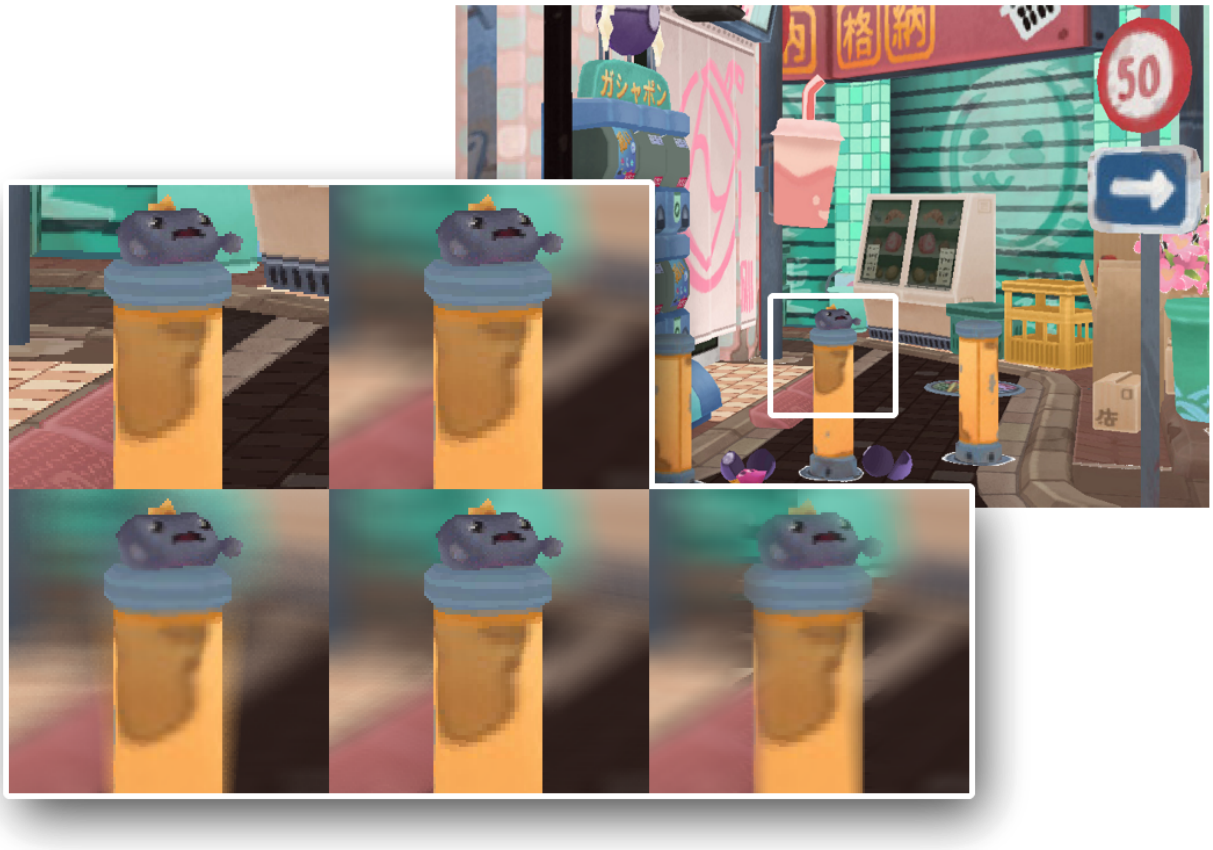

Figure 2.4: Visual Quality of different methods at 512 samples side by side.
From top left to bottom right: Original, Reference scatter method, Basic gather, Advanced gather and Bokeh gather

Figure 2.5: Bleeding from out-of-focus foreground onto the background.
Basic gather (left), Advanced gather (middle) and Bokeh gather (right)

3 Distortion

A camera lens consists out of circular concave and convex lens elements. Depending on the focal length and corrections done inside the lens, there will be some distortions to the image projected on the sensor.

3.1 Barrel Distortion

The most noticeable type of barrel distortion is produced by a very short focal length. Fish-eye lenses for example bend objects (Figure 3.1), the closer they get to the edges of the sensor. This effect can be very easily achieved in a post processing fragment shader by creating a vector from the center of the image to the current pixel. The longer the vector is, the stronger the texture coordinates will be offset in that direction. The vector then needs to be adjusted for the aspect ratio of the screen in order to achieve the barrel distortion of a circular lens element. Unfortunately this method reveals the border of the rendered image. In order to fill the screen, the resulting image needs to be cropped resulting in a lower resolution. To mitigate this, the example implementation uses a global resolution multiplier, which can be changed at run-time. This is easy to implement in render-pipelines based on deferred shading, since only the resolution of the textures the scene is rendered into need to be changed. Many modern video-games offer this functionality and expose it to players in the graphics options as a resolution multiplier slider.

```
out vec3 color;
in vec2 TexCoords;
uniform sampler2D ColorPass;

void main() {
    vec2 uv = TexCoords;
    // Sensor crop
    uv = uv * (1.0 - crop) + crop * 0.5;
    // Vector pointer away from the center
    vec2 fromCenter = (vec2(0.5, 0.5) - uv) * vec2(aspectRatio, 1.0);
    float fromCenterLength = length(fromCenter);
    uv += fromCenter * strength;

    if (uv.x > 1.0 || uv.y > 1.0 || uv.x < 0.0 || uv.y < 0.0 ) {
        color  = vec3(0); // black borders
    } else {
        color = texture(ColorPass, uv).rgb;
    }
}
```

Figure 3.1: Example of strong barrel distortion paired with a wide angle lens.

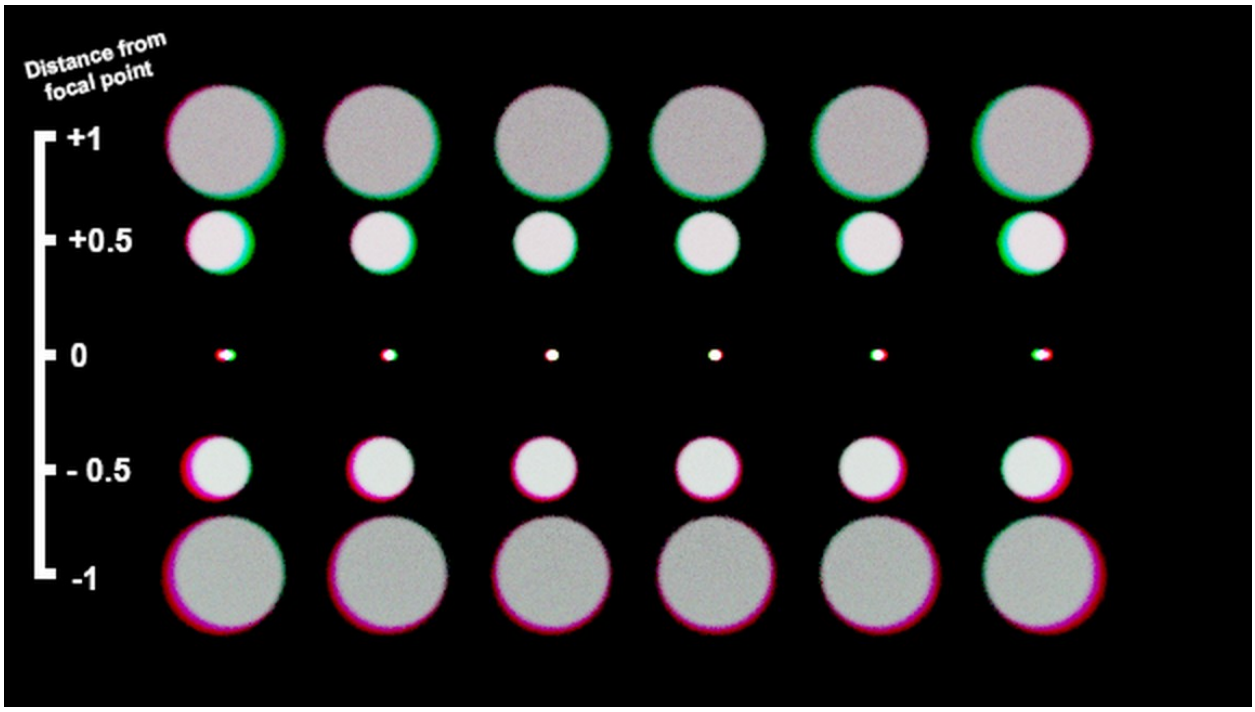

Figure 3.2: Chromatic aberration simulated using a raytracer (Blender Cycles) [ble]

3.2 Chromatic aberration

When light leaves a medium and enters another one, the Index of Refraction (IOR) of both affect how the light ray refracts. Generally the refractive index varies for different wave-lengths. Even if an object is perfectly focused, light rays of different wave-length might not converge at the same point on the camera sensor. Like barrel distortion, this effect becomes more apparent at the edges of the image. The implementation is very similar as well. Instead of distorting the whole image equally, the different color channels are distorted at slightly different intensities and combined back together afterwards.

```
1 void main() {
2     [...]
3     vec2 disp = fromCenter * fromCenterLength;
4     color.r += texture(ColorPass, uv + (disp * -0.003)).r;
5     color.g += texture(ColorPass, uv + (disp *  0.001)).g;
6     color.b += texture(ColorPass, uv + (disp *  0.003)).b;
7 }
```

It is also possible to split up the colors differently to get an effect, which represents the chromatic aberration present in real camera lenses more closely as seen in figure 3.2.

3.3 Vignette

A vignette causes the image to gradually decrease in brightness towards the edges. This effect is especially strong when using a lens with a wide aperture and therefore strong depth of field. Some light-rays at the edge of the sensor might not reach it since the circle of confusion extends beyond the area of the sensor. These light-rays don't contribute to the image brightness. This means vignetting isn't a distortion effect, but the implementation closely relates to the previous distortion effects and some variable can be shared with them. The length of the vector pointing outwards from the image center can be used to calculate the vignette. To shape the strength of the vignette towards the edges, an exponential function can be used. The effect looks more natural if the strength is not linear.

```
void main() {
   [...]
  // shape the falloff
  float vignette = pow(fromCenterLength, vignetteFalloff);
  // invert and multiply it onto the image
  color *= 1.0 - (vignette * vignetteStrength);
  // Desaturate the edges
  color = mix(color, vec3(length(color)), vignette * vignetteDesaturation);
}
```

4 Conclusion

Trying to emulate behaviors and imperfections of the real world is a fascinating task within the field of computer graphics. Real-time light transport simulation is becoming reality and many other techniques are being worked on to further improve the realism of virtual environments and how they can be presented to the viewer. While this means a computer generated images can be closer to the ground truth that is reality, there still are a lot of things, which are not worth simulating properly, even if the computational resources available, would allow it. A simple vignette effect is applied easily as a post processing effect without the need to simulate a camera lens and its internal workings. The rise of virtual reality on the other hand might render the emulation of camera effects obsolete, since they are very likely to break the immersion or distract the player. The demand for a cinematic experience, where physical accuracy and presence take the backseat, will still stay relevant for the foreseeable future.

List of Figures

Bibliography

[ble] *Advanced Optical Experiments!* https://web.archive.org/web/20200730130451/https://blenderartists.org/t/advanced-optical-experiments/695835/2. – Accessed on 30.07.2020

[Cou] Courrèges, Adrian: *UE4 Optimized Post-Effects*. http://www.adriancourreges.com/blog/2018/12/02/ue4-optimized-post-effects/. – Accessed on 30.07.2020

[Dem] Demers, Joe: *Chapter 23. Depth of Field: A Survey of Techniques NVIDIA Developer*. https://web.archive.org/web/20200726214114/https://developer.nvidia.com/gpugems/gpugems/part-iv-image-processing/chapter-23-depth-field-survey-techniques. – Accessed on 26.07.2020

[Gus] Gustafsson, Dennis: *Bokeh depth of field in a single pass*. https://web.archive.org/web/20200730130530/https://tuxedolabs.blogspot.com/2018/05/bokeh-depth-of-field-in-single-pass.html. – Accessed on 30.07.2020