Depth of Field and Motion Blur in Realtime Computer Graphics

Advanced Seminar by Prof. Dr. Alfred Nischwitz
University of Applied Sciences Munich

Thomas Post

Summer Term 2013
Abstract

Recent video games increasingly try to produce a more cinematic atmosphere. An important part of cinematography is depth of field and motion blur. With depth of field a director can focus a viewer’s attention to a certain region of the scene. It can be seen as an artistic tool for motion pictures. To bring video games more closer to motion pictures this artistic tool is also widely used in recent video games. Although motion blur is more of a by product relating to the physical limitations of photo and video cameras, it can also be used as an artistic tool to visually enable a better perception of movement and actions in motion pictures and video games.

This report takes a look at some state of the art techniques for motion blur and depth of field rendering on modern graphics pipelines in realtime. Depth of field is covered with an older approach that is often used and is extended with the effect of a fast aperture bokeh. Motion blur is covered with a rather new approach that can also be used for depth of field. This approach is intended for future generations of graphics hardware and is not production ready today.
## Contents

1. **Introduction** .......................................................... 4  
   1.1. Overview ....................................................... 4  
   1.2. Research Papers ............................................. 4  
   1.3. Document Structure ......................................... 5  

2. **Depth of Field** ................................................... 6  
   2.1. Outline ....................................................... 6  
   2.2. Basics of Camera Lenses ................................... 6  
   2.3. Bokeh ........................................................ 6  
   2.4. Related Work ................................................ 7  
   2.5. Screen-Space Depth of Field ............................... 8  
   2.6. Screen-Space Bokeh ........................................ 9  
      2.6.1. Separable Filters .................................... 10  
      2.6.2. Combining Filters ................................... 10  
      2.6.3. Artefacts .............................................. 10  
      2.6.4. Implementation ....................................... 10  
      2.6.5. Performance & Results ............................... 12  

3. **Motion Blur** ...................................................... 13  
   3.1. Outline ....................................................... 13  
   3.2. Shutter and Image Sensor .................................. 13  
   3.3. Camera - and Object Motion Blur ......................... 13  
   3.4. Related Work ................................................ 14  
   3.5. Multi Sample Anti Aliasing ............................... 14  
   3.6. Decoupled Sampling ....................................... 15  
      3.6.1. Implementation ....................................... 16  
      3.6.2. Transparency ......................................... 19  
      3.6.3. Performance and Results ............................ 19  
   3.7. Conclusion .................................................. 20  

A. **Code Listings** .................................................. 26  

B. **Data CD** ......................................................... 29  

C. **Erklärung** .......................................................... 30
1. Introduction

This work emerged from an advanced seminar with topics from computer graphics and digital image processing. It is part of the computer science masters degree course at the Munich University of Applied Sciences. The Seminar was held by Prof. Dr. Alfred Nischwitz and focused on the current state of real-time computer graphics.

This work covers the topics of Depth of Field and Motion Blur. Two state of the art research papers are evaluated and discussed. Where it suits the evaluation or discussion, implementations are either made or reviewed. The first paper is based on a widely used approach and extends to add a bokeh effect to the depth of field. The second paper introduces a new approach on how to render depth of field and motion blur.

1.1. Overview

In real-time computer graphics a pin-hole camera model is used. While this model is classed as mathematically ideal, in the real world no camera is a perfect pin-hole. The aperture nor the shutter speed can be infinite small or short and there are lenses involved that bring distortion and refraction with them. Since we are used to the effects produced by these non ideal pin-hole cameras, we want them in computer graphics as well. They produce a cinematographic effect which makes it look more realistic and are used as an artistic tool. In motion pictures DOF is used to attract the viewers attention to a certain region in an image or video. The same can be done in video games e.g. in cut scene.

The work here focuses on Depth of Field (DOF), Motion Blur and Bokeh which in itself is an addition to DOF. DOF is affected by the curvature of the lens and the not infinite small aperture. Motion Blur occurs because the shutter speed is not infinite small. So the images start looking blurry on fast movements of the camera or parts of the scene.

1.2. Research Papers

The Motion Blur Rendering: State of the Art [Navarro et al., 2011] research is a good starting point for motion blur rendering. All kinds of techniques are reviewed and compared. According to this paper one of the most recent and most promising motion blur techniques seems to be Decoupled sampling for graphics pipelines [Ragan-Kelley et al., 2011]. It introduces a new approach to decouple shading from visibility sampling. Despite the fact that this is a very new paper, there is already some research that takes this approach further. Since deferred shading [Hargreaves, 2004] becomes more and more popular in recent game engines, combining it with decoupled sampling seems popular, as seen in [Liktor and Dachsbacher, 2012] and [Petrik et al., 2013].

1 “Deferred shading is a screen-space shading technique. It is called deferred because no shading is actually performed in the first pass of the vertex and pixel shaders: instead shading is deferred until a second pass.” [Wikipedia, 2013b]
Latest research in stylized depth of field is Efficiently Simulating the Bokeh of Polygonal Apertures in a Post-Process Depth of Field Shader [McIntosh et al., 2012]. Here a rather simple and fast screen-space approach is taken to simulate polygonal apertures and simulate the so called Bokeh effect. It is based on a very popular technique for DOF rendering first introduced by [Scheuermann and Tatarchuk, 2004] and [Riguer et al., 2004].

1.3. Document Structure

This introduction gave a general overview of the covered topics and evaluated research papers. The following chapters cover these research papers in more detail. Chapter 2 Depth of Field covers the Efficiently Simulating the Bokeh of Polygonal Apertures in a Post-Process Depth of Field Shader paper. First a look at how DOF with a conventional photo or video camera works and how bokeh comes into existence. It gives an introduction of the general problematics of the screen space approach and how they can be solved. An implementation of the paper was made and results of it are presented.

The next chapter, 3 Motion Blur, covers the Decoupled sampling for graphics pipelines paper. This is a relatively new approach to simulate stochastic rendering of Motion Blur and DOF. It is based on the idea of Multi Sample Anti Aliasing (MSAA). In addition to this, a more detailed look at two suggested implementations from [Liktor and Dachsbacher, 2012] is taken. It shows a more practical implementation and extends decoupled sampling with deferred shading. Results from both [Ragan-Kelley et al., 2011] and [Liktor and Dachsbacher, 2012] are compared and evaluated.

Finally the reviewed research is summed up and regarded in matter of progress in these fields of research.
2. Depth of Field

2.1. Outline

This chapter covers depth of field rendering with bokeh in a screen-space technique. Based on the research in [McIntosh et al., 2012].

First we take a look at the physical model of cameras and lenses and show how depth of field and bokeh comes into existence. Sections 2.2 Basics of Camera Lenses and 2.3 Bokeh cover the physical aspects.

Section 2.4 Related Work takes a look at the related work of [McIntosh et al., 2012]. From this, the idea of screen-space is demonstrated and how DOF is achieved with this technique. Section 2.5 Screen-Space Depth of Field discusses the implementation and shows the problems and inaccuracies compared to the physical model.

In the last section, 2.6 Screen-Space Bokeh, the bokeh technique of [McIntosh et al., 2012] is reviewed. A complete implementation was achieved with the rendering engine Irrlicht. A closer look at this implementation and the problems of this techniques are exposed.

2.2. Basics of Camera Lenses

When a real camera with a lens takes a picture the light rays are refracted by the lens and, depending on the distance between the lens and the object, they end up at different locations on the image plane. Figure 2.1 shows that objects at different distances \((D_N, S, D_F)\) end up at different locations on the image plane \((V_F, V, V_N)\). The blue object becomes sharp on the image plane, but the red and green objects become blurred either because they’re too far away or too close. The size of the aperture \(c\) also changes which decides how intensively blurred the object becomes.

Since realtime rendering pipelines do not use a pinhole camera model they do not have a lens that would generate DOF. This effect therefore has to be applied additionally. In non realtime rendering with ray-tracing a lens can be simulated relatively easily. Multiple rays from slightly different positions can simply be shot through a focal point. For more on this see [Cook et al., 1984].

2.3. Bokeh

The word Bokeh origins from japanese (ぼかし) and is blur or the aesthetic quality of

---

1Rushing Mania / CC-BY-NC-2.0
the blur, in out-of-focus areas of an image [Wikipedia, 2013a]. Its origin lies in the shape of the aperture and lens.

This means that the size and geometry of \( d \) in figure 2.1 effects how the out-of-focus various areas \( (V_F, V_N) \) look like. In figure 2.2 a seven edge polygon shape is used as an aperture. This results in seven edge shaped fragments in the blurred areas.

2.4. Related Work

In [McIntosh et al., 2012] five different categories of depth of field techniques are listed:

- Distributed traced rays across the surface of a lens. This technique delivers the best quality but is for ray-tracing and is not very suitable for modern graphics pipelines and real-time computer graphics.

- Rendering the scene from multiple cameras (accumulation-buffer). This also delivers qualitative good results but is too slow for real-time computer graphics.

- Rendering and compositing multiple layers. This technique is becoming more and more popular and is possible to do in real time.

- Forward mapped z-buffer. The influence of each pixel on other pixels is computed. This his not very suitable for modern graphic pipelines.

- Reverse mapped z-buffer. The idea is the same as in forward mapped z-buffer but for any one pixel all pixels influencing this pixel are considered. This is a lot more suitable for modern graphic pipelines. Therefore this approach is chosen here.

A standard screen space depth of field approach that is described in [Riguer et al., 2004] and [Scheuermann and Tatarchuk, 2004] is used. These are conventional-reverse mapped z-buffer techniques. These techniques do not consider bokeh, they simply blur depending on the depth value in

---

\(^2\)Jeff Conrad / CC-BY-SA-3.0
the z-buffer.

For the blur a detachable low-pass filter is used. This is a very standard approach that is widely used in modern computer graphics for all kinds of blurs and other effects. Often gauss or mean filters are used.

2.5. Screen-Space Depth of Field

Screen Space: in screen space techniques usually the entire scene is rendered into a texture instead of rendering it directly to the screen. In a second pass some algorithms are applied to this texture. Together with additional information, such as a depth map or a normal map, a wide variety of effects can be applied. The advantage of this effect is that it is completely independent from the scenes complexity. It is called screen space since it operates on the 2D image or screen space, but not in the 3D space.

Circle of Confusion: the reverse mapped z-buffer technique uses variable filter sizes that is related to the depth value of each pixel. To determine the filter size, the (circle of confusion, CoC) formula 2.1 is used. The result of this formula is then stored into the alpha channel of the image. No extra render pass for the CoC map is needed.

\[ c = A \cdot \frac{|S_2| - |S_1|}{S_2} \cdot \frac{f}{S_1 - f} \tag{2.1} \]

\[ c \] Circle of Confusion size
\[ A \] The aperture’s diameter
\[ f \] focal length
\[ S_1 \] focal distance
\[ S_2 \] The pixels depth value from the z-Buffer

This formula is derived from the well known thin-lense-model. Figure 2.3 shows a rendering with this simple depth of field approach. On the upper left the red square shows the depth map of the rendered scene. But this method comes with a few downsides that are addressed only partially here:

- **Intensity leakage or pixel bleeding:** focused objects leak onto blurred objects in the background. Since the object in the background does have a large CoC it also includes the object on the foreground. [Riguer et al., 2004] and [Scheuermann and Tatarchuk, 2004] address this problem by adding weight to all samples according to their depth value in comparison to the average depth.

- **Lack of partial occlusion:** on a photo taken by a camera blurry objects on the foreground spread out smoothly over focused areas on their edges. In post processing depth of field techniques blurry foreground objects stay opaque. Figure 2.4 shows an example of this problem. A pillar in front
covers a wall that is in focus on the background. The edges of the pillar are opaque.

Figure 2.4.: Lack of Partial Occlusion and Sharp Silhouettes

In [McIntosh et al., 2012] this problem is unsolved, as it remains in the implementation for this seminar. But new research [Schedl and Wimmer, 2012] applies a layered approach to the screen space DOF to solve this problem.

**Sharp silhouettes:** is another artifact that can be seen in figure 2.4 too. The edges are not only opaque, they are also sharp. This problem can be reduced by filtering the cod map. The difference between the CoC sizes of two neighbor pixels in different focal areas are not so large anymore. Details about this solution can be found in [Hammon, 2007].

**No realistic bokeh:** since the shape of the aperture is normally a square in these approaches, there is no aperture bokeh. This is where [McIntosh et al., 2012] comes in action. It adds a bokeh effect by adjusting the shape of the filter kernel.

### 2.6. Screen-Space Bokeh

A naive approach to achieve bokeh would be to use a kernel that is shaped like the desired aperture shape. This would result in good bokeh and has the advantage that every desired shape is possible. However, its performance would be very poor. If we want a polygonal kernel like in figure 2.5 up to a 120 samples would be required. Since current low end graphics hardware is not even able to perform 32 texel fetches per fragment, this solution is not practicable.

Figure 2.5.: Naive Filter

This is where [McIntosh et al., 2012] comes up with a separable approach for polygonal shaped filters. It begins with a modified ‘box-blur’ filter, which can simulate the effect of any parallelogram-shaped aperture. By combining two or more of these parallelogram-shaped filters many kinds of shapes can be generated.
2.6. SCREEN-SPACE BOKEH

2.6.1. Separable Filters

A fixed number of samples are used. These samples are spread according to the CoC of the currently processed pixel when filtering the image. In a second pass the same samples are used to filter the output of the first pass. This time, however, the samples are rotated by some angle. With this any parallelogram shaped filter can be generated. Figure 2.6 shows a filter kernel that is rotated by $\frac{\pi}{4}$.

Figure 2.6.: Parallelogram Shaped Filter

2.6.2. Combining Filters

Now to achieve a more complex shape, multiple images filtered with different shapes can be combined with boolean operations. This technique is known from constructive solid geometry. Figure 2.7 shows two examples of such combinations.

Figure 2.7.: Combining Simple Shapes to Generate More Complex Shapes

The problem is however that we only have pre-filtered images. In these images every pixel has its own polygon and overlaps the polygons of all surrounding pixels. Therefore a proper boolean intersection or union is not possible. However there are the $\text{Min}(x, y)$ and $\text{Max}(x, y)$ functions which will accomplish a similar effect.

With the $\text{Min}(x, y)$ function the least bright of $x$ and $y$ is returned, so a boolean intersection by preserving bright pixels only can be achieved. The boolean union can therefore be approximated with the $\text{Max}(x, y)$ function. It returns the brighter pixel of $x$ and $y$. [McIntosh et al., 2012] suggests to use the $\text{Max}(x, y)$ function since the results show that it produces less artifacts.

2.6.3. Artefacts

This technique produces good results, but there are still some flaws. If the CoC of variably bright points start to overlap, bright defocused highlights with the wrong shape appear. Also the $\text{Min}(x, y)$ function decreases the images intensity in the defocused areas. The $\text{Max}(x, y)$ function on the other hand causes problems on overlapping defocused highlights. If they do not overlap in the input image but should in the combined image. $\text{Max}(x, y)$ also produces slightly increased brightness of the image.

The major artifacts appear to do the sampling of the filter kernel. With a fixed size of samples the larger the filters are, the more artifacts arise. With more samples on larger filters this could be reduced while still maintaining good performance on small filters.

2.6.4. Implementation

The implementation developed for this seminar is based on the rendering engine Irrlicht. Irrlicht is a platform independent engine written in C++ and has all the required features to implement this technique. The
2.6. SCREEN-SPACE BOKEH

(a) Result of a default quadratic filter

(b) Result of the First Filter Pass (angle $\frac{\pi}{4}$)

(c) Result of the Second Filter Pass (angle $\frac{\pi}{4}$)

(d) Result of the Max($x, y$) Function

(e) Result of the Min($x, y$) Function

Figure 2.8.: Bokeh Rendering
prototype is organized in four steps.

1. Render the scene into a texture. Calculate the CoC for every fragment and store it in the alpha channel.

2. Apply the first separable filter in two render passes with the given rotation angles.

3. Apply the second separable filter in two render passes with the given rotation angles. If the first pass of the previous step uses the same filter the output of it could be reused for this step.

4. Finally combine the results form step 2 and 3 with the $\text{Min}(x, y)$ or $\text{Max}(x, y)$ function.

Listing A.2 in Appendix A shows the fragment shader used for step 2 and 3. It is a simple implementation with an average filter.

A look at the implementation makes it also obvious that the entire process is performed on the GPU. Only mapping render targets and textures is initiated by the CPU. Everything else takes part in three shaders. The CoC shader, the filter shader and the final combination shader. In the actual prototype an additional render pass is done to use the built in shaders for rendering the light map scene. However, this is just an Irrlicht implementation detail and should not be of any concern here.

### 2.6.5. Performance & Results

In [McIntosh et al., 2012] the performance of the separable approach is first compared to the naive approach. The abnormality there is that the naive approach with 256 and more samples does not compile anymore on a GeForce 8600 GT 256MB video card.

With regards to the naive approach, the performance is estimated according to the number of samples taken. For the separable approach the performances can be estimated with the number of samples per pass squared. Since the images are also combined afterwards this is not totally accurate, but enough for an approximation.

The computation power required for a naive filter with 64 samples, a $16 \times 16$ filter with a combination of two images could be used instead. To achieve that however with a naive filter more than 256 samples would be needed. One can clearly see that the amount of samples taken can be dramatically reduced by this separable approach.

We see that this separable approach is much faster as a naive approach while still delivering acceptable results. A comparison with other DOF techniques is missing from [McIntosh et al., 2012] but is suggested in the future work section. Using this technique as a fallback for lower-end hardware is also proposed, if one wants to use a better but also more computationally intensive bokeh effect.
3. Motion Blur

This Chapter covers Motion Blur and DOF rendering with Decoupled Sampling for Graphics Pipelines [Ragan-Kelley et al., 2011]. It is inspired by the idea of Multi Sample Anti Aliasing (MSAA) and Render-Man’s Reyes architecture [Cook et al., 1987]. The paper covers Motion Blur and DOF but the focus in this chapter is on Motion Blur since the general part of DOF is already covered in the previous chapter.

3.1. Outline

The first part of this chapter covers general motion blur. It shows how motion blur is generated on conventional video cameras and what different kinds of motion blur can occur. Sections 3.2 Shutter and Image Sensor and 3.3 Camera - and Object Motion Blur cover this.

In section 3.4 Related Work a look is taken at the work upon which [Ragan-Kelley et al., 2011] is based. Also some work that further extended [Ragan-Kelley et al., 2011] is covered there. Then the details of the decoupled sampling approach is covered in section 3.6 Decoupled Sampling.

The final chapter 3.7 Conclusion is a closing part that sums up the work.

3.2. Shutter and Image Sensor

When a camera takes an image the image sensor has to be exposed to the light. Therefore the shutter is opened for some time. Now if something changes in the scene during this time multiple photo diodes are exposed from the same object and the object will appear blurry on the final image. In order to have an image that does not have any motion blur, the exposure time has to be an infinitely short time. This is the case on a modern graphics pipeline but can never be achieved on a real-world camera. To achieve a realistic looking rendering this effect has to be simulated with a modern graphics pipeline.

3.3. Camera - and Object Motion Blur

Often motion blur is categorized in two categories: Camera Motion Blur (CMB) and Object Motion Blur (OMB). CMB origins from the movement of the camera during the exposure and CMB origins from objects in the scene being moved during exposure.

In recent video games CMB is often approximated with a very simple technique that delivers appealing results. It is a very similar technique to the DOF technique used in the previous chapter 2 Depth of Field. The research of Valve in [Vlachos, 2008] and Crytek in [Sousa, 2008] cover these approaches for CMB in detail.

For OMB a lot more work has to be done, since it only applies to some objects in the scene. For most techniques additional motion information is stored or computed for moving objects. These objects are then blurred according to their motion information. The decoupled sampling can produce accurate results for both kinds of motion blur. It also takes into account that when
an object is exposed to multiple pixels it does not have to be shaded multiple times.

Let’s use an example of a rocket flying through a scene in front of the eyes of a user or game-player. Motion vectors for this rocket have to be calculated and the rocket is then blurred and or deformed according to these vectors. To simulate CMB with this technique every object in the scene become motion vectors.

### 3.4. Related Work

To render high quality antialiasing, motion, and defocus blur an accumulation buffer [Haeberli and Akeley, 1990] or stochastic rasterization [Cook, 1986] [Akenine-Möller et al., 2007] can be used. These techniques deliver very high quality results but are still too expensive for real-time computer graphics today.

As mentioned before another technique is the aforementioned used in the previous chapter 2 Depth of Field. This could also be used for motion blur but it would suffer from a lot of approximation errors. This is where decoupled sampling comes in since it does not suffer from the same approximation problems as screen space techniques.

Decoupled sampling is inspired by Reye’s separation of shading and visibility rates [Cook et al., 1987]. Reye uses so called micro polygons: geometry primitives are tessellated in small quads. Then these micro polygons are shaded. This has the disadvantage over shading since shading is done before the visibility test and the splitting into these micro polygons is expensive. Reye decouples shading from visibility but couples the shading rate to the geometry sampling rate. [Fatahalian et al., 2009]

[Ragan-Kelley et al., 2011] decouples shading from both, visibility and geometry sampling with motion- and defocus blur. This is achieved by rendering in a way like MSAA is performed on modern graphics hardware. With MSAA the shading is decoupled from visibility sampling. But the relationship between shading samples and visibility samples is always one-to-n. Visibility samples in a certain area always map to one visibility sample. In decoupled sampling the relationship is inverted to a many-to-one relationship. This is done with a shading cache that can be reused for visibility sampling. The shading rate on blurred areas of the scene is therefore reduced.

This technique is extended by [Liktor and Dachsbacher, 2012] and [Petrik et al., 2013] with deferred shading. This combination is, according to [Liktor and Dachsbacher, 2012], more “interactive” than “real time” however. [Ragan-Kelley et al., 2011] cannot make a statement about real performance since it was only implemented in a simulator.

### 3.5. Multi Sample Anti Aliasing

With MSAA shading is performed in pixel resolution and visibility is processed at a supersampled resolution. The shading is then blended with the frame buffer at the supersampled resolution. In the final step the blended supersampled buffer is down sampled to pixel resolution. Figure 3.1 gives a good overview of that process.

This technique only anti aliases the edges of polygons. It does no anti aliasing on the textures inside the polygons. For texture anti aliasing other techniques such as mip-mapping can be used. The memory consumption of this technic is also critical. Since 2× MSAA already uses twice the amount of memory as without MSAA.
3.6. Decoupled Sampling

Now let's first take a look at the pseudo code of the decoupled sampling algorithm in listing 3.1:

**Listing 3.1: Decoupled Sampling**

```plaintext
for all primitives do
    setup, compute edge equations
    for all visibility samples do
        skip if not visible
        map to shading sample
        if not in cache then
            shade and cache
        else
            use cached value
        end if
    end for
end for
```

In the first step a primitive is rasterized against visibility samples. Here extra data per vertex might be included, such as a $t$ for the time dimension (motion blur) or $u, v$ for the lens (DOF).

Next step maps the visibility sample to a shading sample. E.g, a vertex is moved over time and results in multiple visibility samples. The shading sample, however, is always the same since it is assumed that the shading will not alter over time. The visibility sampling is therefore decoupled from the shading and results in a many-to-one relationship.

A cache is used for the shading samples. For every visibility sample a lookup is done and the cache value is used if the shading was already computed for this visibility sample. Otherwise the shading sample is

---

1From [Ragan-Kelley et al., 2011]
computed and stored in the cache for later use. [Ragan-Kelley et al., 2011] uses timestamps in the cache. On every access of the cache the timestamp is set to the current time. When the cache is full and a new shading sample has to be added, the entry with the oldest timestamp is replaced. This is a so called LRU (least recently used) strategy. In [Liktor and Dachsbacher, 2012] a bucket hash array is used as a cache, and the shading samples are replaced on hash collisions. This might result in some unnecessary re-shading but can be implemented faster than the LUR strategy.

In the last step the resulting color is blended into the frame buffer at full super-sampled resolution. This is equivalent to the MSAA with the exception that it is not a strict one-to-many relationship.

### 3.6.1. Implementation

To implement this algorithm directly on a modern graphics pipeline a few problems have to be solved first. The main problem is that the GPU runs fragment shaders in parallel. A cache of already computed fragments is therefore critical since synchronization is necessary. The following sections cover two alternative implementations. They are both from [Liktor and Dachsbacher, 2012] and [Liktor and Dachsbacher, 2013], where the second is a GPU Pro4 article based on the first one. The implementations from [Ragan-Kelley et al., 2011] are not discussed any further here since the one intended for implementation on conventional GPUs is more theoretical and the other suggested implementation is for the Larrabee architecture which is still only theoretical. Figure 3.2 gives a good overview of how all these implementations work in general.

#### Global Shading Cache

First the range of the Shading Sample ID’s (ssID) for every primitive is generated. This can be done with a geometry shader and an atomic counter. Fragment shaders can then map visibility samples to shadings samples and eliminate redundancy from the shading data. Listing 3.2 shows pseudo code for such a geometry shader. To implement this at least OpenGL 4.2 is required.

```cpp
void main()
{
    // project screen position to the shading grid
    vec2 pos0 = scrPos[0] * shadingRate;
    [...]
    vec2 minCorner = min(pos0, min(pos1, pos2));
    vec2 maxCorner = max(pos0, max(pos1, pos2));
    // shading grid: xy-top left corner, zw-grid size
    domain.x = int(minCorner.x) - 1;
    domain.y = int(minCorner.y) - 1;
    domain.z = int((maxCorner.x)) - domain.x + 1;
    domain.w = int((maxCorner.y)) - domain.y + 1;
    // we allocate the ssID range with an atomic counter
    uint reserved = uint((domain.z) * (domain.w));
    startID = imageAtomicAdd(
        uCtrSSID, 0,
        reserved
    );
}
```

To improve performance of the cache lookup the fact that only recently stored shading values are “interesting” is used. This originates from the streaming nature of GPUs where fragments that are close to each other are also processed close to each other in time. Instead of searching the entire buffer only the recently used ssID’s are searched. [Liktor and Dachsbacher, 2012] uses a bucket hash array with a simple modulo as
the hash function $h()$. To ensure that an existing cache value does not originate from a hash collision, the actual ssID is stored along with the shading value. These tuples are then stored in a FIFO queue per bucket. To determine if a shading value is already computed or not, $h(ssID)$ is computed and the according value is fetched. To ensure that not multiple threads insert the same ssID into the cache synchronization has to be introduced. Since the same ssID always points to the same bucket a binary semaphore per bucket is used. Listing 3.3 shows a pseudo code implementation of such a synchronized cache. In this code instead of a direct shading sample only an address to the storage of the shading data is used. This is due to the deferred shading. Only the address to the deferred shading data is stored in the cache instead of the direct shading sample.

Listing 3.3: Global Shading Cache

```c
layout(rgba32ui) uniform uimageBuffer uShaderCache;
layout(r32ui) uniform volatile uimageBuffer uBucketLocks;

int getCachedAddress(
inout bool needStore

int hash = hashSSID(ssID);
vec4 bucket = imageLoad(uShaderCache, hash);
in address = searchBucket(ssID, bucket);

// cache miss
while(
  address < 0 &&
  iAttempt++ < MAX_ATTEMPTS
){
  // this thread is competing for storing a sample
  uint lock = imageAtomicCompSwap(
    uBucketLocks,
    hash,
    FREE,
    LOCKED
  );
  if(lock == FREE){
    address = int(atomicCounterIncrement(
      bufferTail));
    // update the cache
    bucket = storeBucket(ssID, hash, bucket);
    imageStore(uShaderCache, hash, bucket);
    needStore = true;
    memoryBarrier(); // release the lock
  } else if(lock == LOCKED){
    while(
      lock == LOCKED &&
      lockAttempt++ < MAX_LOCK_ATTEMPTS
    ) {
      lock = imageLoad(uBucketLocks, hash).x;
      // now try to get the address again
      bucket = imageLoad(uShaderCache, hash);
    address = searchBucket(ssID, bucket);
    // if everything failed, store the data redundantly
    if(address < 0){
      address = int(atomicCounterIncrement(
        bufferTail));
      needStore = true;
    }
  }
}
```

Figure 3.2.: Decoupled Sampling Piepeline
Per-Tile Shading Cache The above algorithm uses global memory atomics. This becomes the main bottleneck of the entire process. So the following implementation takes into account that a shading sample is normally only reused in a certain area on the screen and not at completely arbitrary locations. Therefore the image is split into uniform tiles. The cache is then only valid inside such a tile and it is entirely processed in one thread. So no synchronization between threads is necessary. This has the disadvantage that some samples do not end up in the same tile and have to be shaded multiple times. So the tile size selection is important and might vary according to the amount of blur used. The per-tile shading has three steps:

- In the first step a depth map and all the ssID’s for the entire scene are generated. Instead of directly mapping the samples in the fragment shader (as in the previous approach). The ssID’s are written out and used in a second pass.

- The second pass then tiles the data and processes every tile in a single thread. There the ssID’s are mapped to shading data addresses (again because of the deferred shading). This pass can be implemented in a fully computational pass to control the thread execution. In [Liktor and Dachs, 2012] OpenCL is suggested for this step.

The output of this pass is an address for each visibility sample where the shading data can be stored.

Now in the last step the shading data is physically stored. The depth buffer from the first pass is here used to ensure that only visible fragments are executed here. This avoids that false data is written, since we do not have any synchronization here.

The main advantage of this algorithm over the first one is that the local per tile cache is a lot faster than the global cache.
And with a good tile size selection the problem of over shading can be minimized. Figure 3.3 shows this process. The tiling here causes the corner of the yellow triangle to be shaded twice since it does not end up entirely in the same tile. For simplicity the triangles are shaded flat in this example.

[Petrik et al., 2013] uses such a tile based approach and extends with adaptive anisotropic shading (AAS) [Vaidyanathan et al., 2012]. Figure 3.4 shows the over shading at tile edges and that it is further reduced by AAS.

### 3.6.2. Transparency

With motion blur and depth of field also comes the requirement of transparency rendering. In [Liktor and Dachsbacher, 2012] it is suggested that [Enderton et al., 2010] or [Yang et al., 2010] is used for order independent transparency.

The first one uses a stochastic approach that works well with MSAA but adds noise. The second technique uses linked lists to generate a so called A-buffer. An A-buffer is a depth buffer that not only stores the nearest fragments position but rather a list with all fragments at that position. So the fragments can be evaluated in their correct order to produce a final shading value.

[Ragan-Kelley et al., 2011] does not explicitly point at techniques for solving the transparency problem. Although it is mentioned that RenderMan uses an A-buffer but since it needs a complex data structure a stochastic approach would fit better.

### 3.6.3. Performance and Results

Figure 3.5 is from [Liktor and Dachsbacher, 2012] and shows their results. The first row is a reference implementation using OptiX.

The second row is the result of the decoupled deferred rendering. In the right column the amount of shading samples per pixel (sspp) is visualized. One can see that at the areas of high motion blur the shading rate is dramatically reduced.

In terms of speed and memory consumption this approach is still very expensive. As in figure 3.5 the gargoyles scene with motion blur needs about 400ms to render. It is still not therefore capable in real-time. Further results show the rendering of the sponza scene blurred with 251ms and sharp with 357ms. There the same effect can be seen that blurred areas need less computational time since the caching is more effective. Memory consumption in this approach is also very high. Due to the deferred-rendering approach the sponza scene at 1280 × 720 needs around 250MB of graphics memory at 32× supersampling. And this is already using an optimized version of the regular G-buffer used in deferred rendering called CG-buffer for compact geometry buffer. Only the depth and reference buffer is stored at supersampled resolution. Normal, diffuse and specular are only stored at the normal sampling rate. A fully supersampled deferred rendering at 32× supersampling would use up to 350MB of graphics memory.

[Ragan-Kelley et al., 2011] does a very detailed performance and quality analysis. Although most of the analysis goes into the comparison of the sort-last (GPU-style) and the sort-middle (Larrabee) implementation. The tests are performed with a “Direct3D 9 functional simulator”. But since Larrabee is only a theoretical concept these results are not worth a lot. So in the paper itself it is stated:

---

4From [Liktor and Dachsbacher, 2012]

3.7. Conclusion

...it is challenging to directly predict the absolute performance of a hardware architecture that has not been built, ...

More interesting is the comparison with an accumulation buffer and a stochastic supersampling implementation. As seen in figure 3.6 it shows similar results as [Liktor and Dachs, 2012] in case of the shading samples per pixel (Shading rate). The difference of the shading rate between 8 and 64 visibility samples per pixel is only around 0.7 to 1.0. The main goal of decoupled sampling, keeping the shading rate low at a high visibility sampling rate, is therefore extremely successful.

3.7. Conclusion

The underlying technique used for the approach in chapter 2 is nearly ten years old. A lot a research has gone into this approach and is still ongoing, as we can see by [McIn-
3.7. CONCLUSION

 CHAPTER 3. MOTION BLUR

Figure 3.6: Results from [Ragan-Kelley et al., 2011]. Accumulation buffer rendering and stochastic supersampling are compared to decoupled sampling.

tosh et al., 2012] and [Schedl and Wimmer, 2012]. Due to improvements in graphics hardware over the last few years this technique can now be used with expensive and complex filter kernels and, with the scalability of filter kernels, this technique can also be a good replacement for more sophisticated DOF on low-end hardware. The technique produces good results with the suggested improvements.

The second approach from chapter 3, however, is a completely new approach on this topic. It is relatively new and needs top-end graphics hardware to run at least at an interactive frame rate. It introduces a new approach for stochastic sampling performed in a conventional forward-rendering graphics pipeline. There is still more new research going into this technique as we can see by the research paper [Petrik et al., 2013] that will be presented at SIGGRAPH 2013, specifically the fact that it reduces shading costs on blurred areas, as this scales with the visual impression a user gets from the rendering.

The screen space approach on the other hand (chapter 2) does exactly the opposite. It is more costly on blurred areas than on focused areas. If the cache mechanism gets better hardware support in the future, decoupled sampling might be able to be performed in real-time and become faster even than the filter approach for some tasks.

Looking to the future, [Liktor and Dachsbacher, 2012] suggests investigating better caching strategies for the tile-based
approach. The generation of the ssID could also be made temporarily coherent to reuse shading samples across multiple frames.


List of Figures

2.2. Bokeh Photography ............................................. 6
2.1. DOF with Lense .............................................. 7
2.3. Simple Depth of Field Rendering ................................. 8
2.4. Lack of Partial Occlusion and Sharp Silhouettes ................. 9
2.5. Naive Filter .................................................. 9
2.6. Parallelogram Shaped Filter .................................... 10
2.7. Combining Simple Shapes to Generate More Complex Shapes ... 10
2.8. Bokeh Rendering ............................................. 11

3.1. Multi Sample Anti Aliasing ................................... 15
3.2. Decoupled Sampling Piepeline ................................ 17
3.3. Per-Tile Shading .............................................. 18
3.4. Results from [Vaidyanathan et al., 2012] show the over shading at the tile edges and how it is reduced with AAS ....................... 20
3.5. Results from [Liktor and Dachsbacher, 2012] (bottom row) compared to a reference implementation with OptiX (top row) ............... 20
3.6. Results from [Ragan-Kelley et al., 2011]. Accumulation buffer rendering and stochastic supersampling are compared to decoupled sampling .... 21
A. Code Listings

Listing A.1: Filter Vertex Shader

```glsl
uniform mat4 mWorldViewProj;
uniform float uFilterAngle;

varying vec2 vSamples[6];

void main(void)
{
    gl_Position = mWorldViewProj * gl_Vertex;
    gl_TexCoord[0] = gl_MultiTexCoord0;

    float aspectRatio = 640.0/480.0;
    float radius = 0.1;
    float angle = uFilterAngle;
    int numbSamples = 6;

    // rotate sample point
    vec2 point = vec2(radius * cos(angle), radius * sin(angle));

    point.x /= aspectRatio;

    // create sample positions in vertex shader so they are interpolated and dependent texture reads are avoided
    for (int i = 0; i < numbSamples; i++) {
        float t = float(i) / (float(numbSamples) - 1.0);
        vSamples[i] = mix(-point, point, t);
    }
}
```

Listing A.2: Filter Fragment Shader

```glsl
uniform sampler2D uColorTexture0;
uniform sampler2D uCoCTexture0;
uniform sampler2D uDepthTexture0;
```
varying vec2 vSamples[6];

void main ( void )
{
    float bleedingMult = 0.5;
    float bleedingBias = 0.01;

    vec2 textureCoord = vec2(gl_TexCoord[0]);
    textureCoord = vec2(textureCoord.x, 1.0 - textureCoord.y);

    //read color, coc and depth at fragment position
    vec4 color = texture2D(uColorTexture0, textureCoord);
    float CoC = texture2D(uCoCTexture0, textureCoord).a;
    float depth = texture2D(uDepthTexture0, textureCoord).r;

    int numbSamples = 6;
    vec4 outputColor = vec4(0.0);

    //iterate over every sample
    for ( int i = 0; i < numbSamples; i++) {
        vec2 offset = vSamples[i];
        vec2 sampleCoords = textureCoord + offset * CoC;

        //read color, CoC and depth at sample location
        vec4 sampleColor = texture2D(uColorTexture0, sampleCoords);
        float sampleCoC = texture2D(uCoCTexture0, sampleCoords).a;
        float sampleDepth = texture2D(uDepthTexture0, sampleCoords).r;

        //avoid light bleeding from focused objects to background
        float weight = sampleDepth < depth ? sampleCoC * bleedingMult : 1.0;
        weight = (CoC > sampleCoC + bleedingBias) ? weight : 1.0;
        weight = clamp(weight, 0.0, 1.0);

        //add sample to total color
        outputColor.rgb += sampleColor.rgb * weight;
        outputColor.a += weight;
    }

    //average over all samples
    outputColor /= outputColor.a;
}
\begin{verbatim}
gl_FragColor = vec4(outputColor.rgb, 1.0);
\end{verbatim}
B. Data CD

CD
  | Documentation
  | __Seminar.pdf ......................... A digital version of this document
  | Presentations
  | __Konzept.pdf ....................... Concept presentation from April 4, 2013
  | __Zwischenbericht-I.pdf .............. Status report from Mai 2, 2013
  | __Zwischenbericht-II.pdf .............. Status report from June 6, 2013
  | __FinalPresentation.pdf .............. Final seminar presentation June 27, 2013
  | Code
  | __Bokeh ........ Xcode project with bokeh DOF implementation with irrlicht
C. Erklärung

Thomas Post München, 26.06.2013

IG/SS 2013

Erklärung

Gemäß §40 Abs. 1 i. V. m. §31 Abs. 7 RaPO

Hiermit erkläre ich, dass ich die Seminararbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benützt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Thomas Post