Informatik
Hauptcampus

HOCH
SCHULE
TRIER

State of the Art in Image-Based Post-Processing Effects

State of the Art der bildbasierten Post-Processing Effekte

Tim Dörries

Bachelor's Thesis

Advisor: Prof. Dr. Christof Rezk-Salama

Trier, November 15, 2018

Abstract

Image-based post-processing effects are commonly used in real-time rendering applications to simulate optical phenomena in real cameras and other lighting effects. Recent advances in this field have brought forth a wide variety of techniques, producing high quality effects. This thesis aims to give an overview of and a comparison between modern approaches to post-processing, highlighting concepts that might prove important in future research. Focus is placed on motion blur, depth of field and screen space ambient occlusion (SSAO). In particular, scatter-as-gather algorithms for both motion blur and depth of field are discussed. Scatter-as-gather depth of field is contrasted by sprite-based depth of field, a technique promising high quality visuals on powerful hardware. Discussed screen space ambient occlusion techniques include the original SSAO algorithm and modern horizon-based approaches like Horizon-Based Ambient Occlusion (HBAO) and Ground Truth Ambient Occlusion (GTAO). It is concluded that modern image-based post-processing effects produce high quality results without compromising performance. This is made possible by exploiting temporal coherency and spreading computational work over multiple frames. In addition, cache aware texture sampling strategies, such as tile-based scatter-as-gather motion blur, save memory bandwidth, enabling high performance.

Optische Phänomene realer Kameras, sowie andere Beleuchtungseffekte werden in Real-Time Rendering Anwendungen häufig durch bildbasierte Post-Processing Effekte simuliert. Neue Fortschritte in diesem Gebiet haben eine Vielzahl verschiedener Techniken hervorgebracht. Diese Arbeit stellt eine Übersicht moderner Post-Processing Lösungen an und vergleicht diese mit dem Ziel, vielversprechende Konzepte für zukünftige Forschung hervorzuheben. Der Fokus liegt hierbei auf Motion Blur, Depth of Field und Screen Space Ambient Occlusion (SSAO). Insbesondere werden Scatter-as-Gather Algorithmen für sowohl Motion Blur, als auch Depth of Field behandelt. Scatter-as-Gather Depth of Field wird mit spritebasiertem Depth of Field kontrastiert, eine Technik, die hohe Qualität auf leistungsfähiger Hardware verspricht. Im Rahmen von Screen Space Ambi-

ent Occlusion werden der orginale SSAO Algorithmus, sowie moderne horizontbasierte Ansätze, wie Horizon-Based Ambient Occlusion (HBAO) und Ground Truth Ambient Occlusion (GTAO) thematisiert. Es stellt sich heraus, dass moderne bildbasierte Post-Processing Effekte fähig sind, qualitativ hohe Ergebnisse zu liefern, ohne dabei die Performanz negativ zu beeinträchtigen. Dies wird ermöglicht durch das Nutzen zeitlicher Kohärenzen, sodass Berechnungen über mehrere Bilder verteilt werden können. Zudem schonen cache-bewusste Speicherzugriffsstrategien, wie kachelbasiertes Scatter-as-Gather Motion Blur, die Speicherbandbreite, was eine hohe Performanz erlaubt.

Contents

1 Introduction 1
1.1 Image-Based Post-Processing Effects 1
1.2 Implementation Framework 2

2 Related Work 3
2.1 Motion Blur 3
2.2 Depth of Field 4
2.3 Screen Space Ambient Occlusion 5

3 Method 6
3.1 Motion Blur 6
3.1.1 Physical Basis and Motivation 6
3.1.2 Velocity Buffer Generation 6
3.1.3 Simple Motion Blur 9
3.1.4 Single-Direction Scatter-as-Gather 11
3.1.5 Multi-Direction Scatter-as-Gather 16
3.2 Depth of Field 21
3.2.1 Physical Basis and Motivation 21
3.2.2 Implementing a Virtual Camera 21
3.2.3 Filter Kernels 23
3.2.4 Simple Depth of Field 24
3.2.5 Sprite-Based Depth of Field 25
3.2.6 Scatter-as-Gather Depth of Field 28
3.2.7 Depth of Field Implementation Considerations 32
3.3 Screen Space Ambient Occlusion 32
3.3.1 Physical Basis and Motivation 32
3.3.2 Original Algorithm 35
3.3.3 Hemisphere Kernel 39
3.3.4 Horizon-Based Ambient Occlusion 40
3.3.5 Ground-Truth Ambient Occlusion 44
3.3.6 Screen Space Ambient Occlusion Implementation Considerations 49
3.4 Measurement Method 50
3.4.1 Motion Blur Test Setup 52

3.4.2 Depth of Field Test Setup . . . 52
3.4.3 Screen Space Ambient Occlusion Test Setup . . . 52

4 Results and Discussion . . . 54
4.1 Motion Blur . . . 54
4.2 Depth of Field . . . 56
4.3 Screen Space Ambient Occlusion . . . 60

5 Conclusion . . . 64
5.1 Conclusion . . . 64
5.2 Future Work . . . 64

References . . . 65

Erklärung der Kandidatin / des Kandidaten . . . 67

List of Figures

1.1 Multi-Direction Scatter-as-Gather Motion Blur, Scatter-as-Gather Depth of Field and Ground Truth Ambient Occlusion applied to the Crytek Sponza scene. 2

3.1 Visualization of a velocity buffer featuring both object and camera motion. 7
3.2 Simple motion blur applied to an image of a moving car. 10
3.3 Single-direction scatter-as-gather motion blur applied to an image of a moving car. 12
3.4 The velocity tile texture holds the velocity with the maximum magnitude of each tile. 13
3.5 Strong differences between tiles can lead to motion blur artifacts. . . 16
3.6 The improved sampling scheme of multi-direction scatter-as-gather motion blur almost completely removes the artifacts present in single-direction scatter-as-gather motion blur. 17
3.7 The 49 sample filter kernel used for depth of field. 24
3.8 A simple bokeh depth of field implementation. 24
3.9 A depth of field effect can be achieved by drawing textured sprites for every pixel. 26
3.10 The near field texture is stored in the left half and the far field texture is stored in the right half of the texture atlas. 27
3.11 Scatter-as-gather in two dimensions. Source: [Jim14], modified. 29
3.12 Scatter-as-gather depth of field applied to the scene. 29
3.13 The circle of confusion texture (left) is used to generate a tile texture (right), holding the maximum circle of confusion of every tile. 30
3.14 Direct illumination (left) and indirect illumination (right). 33
3.15 Without ambient occlusion, objects do not look connected (top left). Adding ambient occlusion gives visual cues about spatial relations between objects (top right). Visualization of the ambient occlusion term (bottom). 33
3.16 SSAO computed for a scene of mixed complexity. 36
3.17 The four-by-four SSAO noise pattern is clearly visible. 37
3.18 Visualization of the SSAO kernel on different surfaces. 38

3.19 Visualization of a hemispherical SSAO kernel on different surfaces. . 39
3.20 The ambient occlusion term calculated with a hemispherical kernel. 40
3.21 Information about unoccluded directions (green) cannot be extracted from a height field. ω_o is the view vector. Source: [JWPJ16b], modified. 41
3.22 The ambient occlusion term calculated with HBAO. 42
3.23 Horizon search with four steps (left to right and top to bottom). P is the point to be shaded and S_i are the sample locations. Every found horizon is highlighted in green. Source: [BS08], modified. 44
3.24 GTAO reference frame. ω_o is the view vector, θ_1 and θ_2 are the horizon angles and φ is the angle around the integration axis. Source: [JWPJ16b], modified. 45
3.25 The ambient occlusion term calculated with GTAO. 46
3.26 The normal n does not lie in the plane and must be projected onto it, yielding the projected normal n_p. θ is the horizon. Source: [JWPJ16b], modified. 47
3.27 Without the spatial noise reduction, the noise pattern is clearly visible (left). The result of this operation (right) still needs to be filtered temporally. 49
3.28 False occlusion is detected in horizon based approaches when using interpolated normals. Source: [BS08], modified. 50
3.29 Using a constant velocity ensures reproducibility. 52

4.1 Simple motion blur (top left), single-direction motion blur (top right) and multi-direction motion blur (bottom) applied to the same scene. 54
4.2 Simple depth of field (top left), sprite based depth of field (top right) and scatter-as-gather depth of field (bottom) applied to the first test scene. 57
4.3 Simple depth of field (top left), sprite based depth of field (top right) and scatter-as-gather depth of field (bottom) applied to the second test scene. 57
4.4 SSAO (top left), SSAO (Hemisphere) (top right), HBAO (bottom left) and GTAO (bottom right) applied to the first test scene. 61
4.5 SSAO (top left), SSAO (Hemisphere) (top right), HBAO (bottom left) and GTAO (bottom right) applied to the second test scene. . . . 61

List of Tables

3.1 Hardware specifications of the two test systems. 51
3.2 Overview of techniques measured at each benchmark pass. 51
3.3 SSAO (Hemisphere) parameters used during the benchmarks. 53
3.4 HBAO parameters used during the benchmarks. 53
3.5 GTAO parameters used during the benchmarks. 53

4.1 Motion blur timings on the desktop test system. All timings are in milliseconds. 55
4.2 Motion blur timings on the laptop test system. All timings are in milliseconds. 55
4.3 Depth of field timings of the first test scene on the desktop test system. Time is measured in milliseconds. 58
4.4 Depth of field timings of the first test scene on the laptop test system. Time is measured in milliseconds. 58
4.5 Depth of field timings of the second test scene on the desktop test system. Time is measured in milliseconds. 59
4.6 Depth of field timings of the second test scene on the laptop test system. Time is measured in milliseconds. 59
4.7 Screen space ambient occlusion timings of the first test scene on the desktop test system. Time is measured in milliseconds. 62
4.8 Screen space ambient occlusion timings of the first test scene on the laptop test system. Time is measured in milliseconds. 62
4.9 Screen space ambient occlusion timings of the second test scene on the desktop test system. Time is measured in milliseconds. 62
4.10 Screen space ambient occlusion timings of the second test scene on the laptop test system. Time is measured in milliseconds. 63

1

Introduction

1.1 Image-Based Post-Processing Effects

The field of computer graphics strives to create images indistinguishable from those taken with a real camera. Offline rendering techniques already deliver results very close to this goal. On the contrary, real-time rendering applications require solutions capable of mimicking the results produced by offline techniques, running in a fraction of the time. Real-time rendering applications such as games often use hardware acceleration in the form of graphics chips. Modern graphics hardware creates images by rasterizing primitives, usually triangles. Although significant progress has been made in real-time lighting algorithms, images synthesized through rasterization usually lack certain effects commonly found in photography and film. Many of these effects, like chromatic aberration, film grain or depth of field are caused by the way cameras create images. Since they are ubiquitous in movies and other media, simulating these effects can significantly contribute to the perceived realism of computer generated images. In real-time rendering this is usually done using image-based post-processing techniques. Post-processing effects are applied after an image has been rendered. While offline techniques have access to a complete description of the scene, real-time approaches often do not. For this reason, post-processing in real-time typically generates the finished image based on information taken solely from previously rendered images. Due to this limitation and the requirement of running in real-time, many effects heavily approximate real phenomena. Graphics hardware has developed rapidly in recent years, allowing new techniques to gradually abandon certain approximations and model reality ever closer. With an abundance of image-based post-processing effects available, this thesis aims to give an overview of and a comparison between promising state of the art techniques. In particular, several motion blur, depth of field and screen space ambient occlusion algorithms will be discussed. Although screen space ambient occlusion is not exactly a post-processing effect, it is often placed into this category due to its image-based nature. All techniques will be compared with respect to their performance, ease of implementation and visual accuracy. Figure 1.1 demonstrates motion blur, depth of field and screen space ambient occlusion.

Fig. 1.1. Multi-Direction Scatter-as-Gather Motion Blur, Scatter-as-Gather Depth of Field and Ground Truth Ambient Occlusion applied to the Crytek Sponza scene.

1.2 Implementation Framework

All effects and techniques discussed in this thesis have been implemented using a custom framework based on C++ and OpenGL 4.5. The framework features physically based, deferred rendering. Having access to a geometry buffer opens additional options when implementing image-based post-processing effects. Since the renderer uses OpenGL, all listings in this thesis use the OpenGL Shading Language (GLSL).

2

Related Work

2.1 Motion Blur

Motion blur can be separated into camera and object motion blur. The former is trivially computable from the view projection matrix. Object motion blur is considerably more difficult to achieve and is prone to visual artifacts. Green [Gre03] proposes combating harsh silhouettes around motion blurred objects by stretching object geometry along its screen space velocity direction. Motion blur in the video game *Portal* is restricted to camera motion. Vlachos et al. [Vla08] decompose the effect into vertical, horizontal, roll and a forward blur direction. These components are then weighted and combined to produce the finished effect.

For the video game *Crysis*, Sousa et al. [Sou08] treat camera and object motion blur separately. Camera motion blur is achieved by rendering a sphere around the camera. Velocities are computed directly as the difference of the sphere vertex positions in screen space. Blurring is then done in the same pass. Nearby geometry is masked out based on depth. Iterative application of the same filter pass is used to increase blur quality. Object motion blur is achieved with a velocity buffer. The velocities are dilated in the blur pass, eliminating sharp edges. Camera and object motion blur are combined, producing the finished image.

McGuire et al. [MHBO12] reason about scatter-as-gather approaches to motion blur and use a secondary texture to determine the blur direction of a local pixel neighborhood. The technique exclusively samples along this direction, producing accurate blur around object edges. Sousa et al. [Sou13] improve upon this technique by leveraging modern hardware's SIMD (Single Instruction Multiple Data) capabilities. Guertin et al. [GMN13] introduce further improvements to this technique. They add a second sampling direction, greatly reducing previously present artifacts. Jimenez et al. [Jim14] extend this approach with more accurate sample weighting.

This thesis will focus on recent scatter-as-gather approaches, highlighting their differences and implementation details.

2.2 Depth of Field

Convincing depth of field has been a difficult effect to achieve for a long time. Scheuermann [Sch04] uses a variable size filter kernel to blur the downsampled source image. The circle of confusion filter kernel uses a poisson distribution to place the samples. Color bleeding from sharp foreground objects onto blurry background objects is reduced by taking the depth of samples in account.
Demers [Dem07] compares several different depth of field techniques, among them accumulation-buffer depth of field, which renders the scene multiple times from different angles and accumulates the result in a buffer. He notes that this technique requires a large number of additional render passes to produce acceptable results, making it impractical in most real-time applications. Demers further distinguishes forward-mapped and reverse-mapped depth buffer depth of field. Forward-mapped depth of field renders a sprite for every pixel. The sprite is scaled by the pixel's circle of confusion and blended additively into the scene. Contrary to this, reverse-mapped depth of field tries to determine at each pixel, which other pixels blur over it. Although forward-mapped depth of field can produce convincing results, reverse-mapped depth of field seems to be the more promising technique, as it maps better to graphics hardware.
Hammon [Ham08] builds on reverse-mapped depth of field (which is referred to as scatter-as-gather depth of field) and introduces a number of changes to combat sharp discontinuities on blurry foreground object silhouettes. He generates a full resolution texture holding the circle of confusion for each pixel, downsamples it to a lower resolution and applies a gaussian blur to pixels belonging to the near field. Although not physically based, this procedure efficiently removes unpleasant looking discontinuities.
Kasyan et al. [KSS11] batch motion blur and depth of field together and use taken texture samples for both techniques, improving the performance. Similar to Mittring et al. [MD11] and Valient et al. [Val13], they also use forward-mapped depth of field to produce high quality bokeh depth of field.
McIntosh et al. [MRD12] simulate the bokeh of polygonal apertures using separable filters. They approximate boolean unions and intersections using the *min()* and *max()* functions to composite basic shapes, producing polygonal bokeh shapes.
Based on the scatter-as-gather motion blur technique developed by McGuire et al. [MHBO12], Sousa et al. [Sou13] use a tile texture indicating the maximum circle of confusion to determine the blur kernel radius. Jimenez et al. [Jim14] improve upon Sousa et al.'s approach by reducing the required texture memory. They further employ filtering techniques before and after the blur to reduce undersampling artifacts. Similar to motion blur, scatter-as-gather techniques seem to be popular in recent research, which is why this approach will be discussed in detail. In addition, forward-mapped depth of field will be implemented and contrasted with other techniques.

2.3 Screen Space Ambient Occlusion

Screen Space Ambient Occlusion (SSAO) was originally introduced with the video game *Crysis*, released in 2007. Kajalin et al. [Kaj09] approximate ambient occlusion as the ratio of visible and occluded samples in a spherical kernel applied in screen space on the depth buffer.
Bavoil et al. [BS08][BSD08] calculate ambient occlusion analytically as the angle between the view-space tangent and the maximum horizon as seen from the pixel to be shaded. They call this technique Horizon-Based Ambient Occlusion (HBAO).
Bavoil et al. [BS09] improve upon general SSAO artifacts that happen due to missing information. They use depth peeling and enlarged input images featuring a guard band to restore information previously unavailable during ambient occlusion calculation.
Mattausch et al. [MSW10] exploit temporal coherence between frames to distribute ambient occlusion calculation over time. They store the number of frames contributing to a pixel and dynamically reduce the number of samples taken, when the image is in a converged state.
Hoang et al. [HL11] improve upon the quality of previous SSAO techniques by combining ambient occlusion results for images of various resolutions, producing both small scale detail and a large scale effect.
McGuire et al. [MML12] introduced Scalable Ambient Obscurance (SAO), which prefilters the depth buffer, improving memory efficiency and allowing large kernel radii.
Temporally filtered SSAO often has visual artifacts in the form of moving objects leaving trails. Bavoil et al. [BA12] developed selective temporal filtering, a technique that classifies pixels as stable (potentially leaving trails) or unstable (potentially flickering). Temporal filtering is then disabled for stable pixels, resulting in a mostly flicker and trailing free image.
Timonen [Tim13a] [Tim13b] recognized that in horizon-based techniques adjacant pixels share the same texture lookups and often have similar horizons. He leverages this by building a data structure allowing efficient lookup of maximum horizons in constant time.
Since SSAO is a bandwidth intensive effect, it is often done in half resolution. Bavoil et al. [BJ13] note that the usage of tiled noise in the sampling kernel is detrimental to performance, as it leads to texture cache trashing. They propose deinterleaving the source image into several smaller images, each having its own constant noise value. Ambient occlusion is then calculated for each smaller image. In a reinterleaving pass the final image is constructed by interleaving the results of the small images.
Jimenez et al. [JWPJ16a] [JWPJ16b] introduced Ground Truth Ambient Occlusion (GTAO), a novel technique aimed at achieving results close to ambient occlusion ground truth. GTAO is a horizon-based technique, heavily relying on temporal filtering to achieve the performance target of 0.5 ms on consoles. Horizon-based techniques like GTAO and HBAO are state of the art, which is why they will be discussed extensively in this thesis.

3

Method

3.1 Motion Blur

3.1.1 Physical Basis and Motivation

Motion blur is an effect that happens in photography where moving objects appear to streak across the image. This effect happens both in digital and in analog photography. When a camera creates an image, it exposes its sensor to the incoming light. However, this exposure is not instantaneous but happens over a period of time, called exposure time or shutter speed. While the sensor is exposed to the incoming light, the scene may change for various reasons, such as movement of objects or the camera itself. This movement then manifests itself as streaking in the image along the movement direction. For this reason the strength of the blur effect depends on exposure time and object movement, where a long exposure time and fast movement result in strong motion blur. Although motion blur is not noticeable in human vision, it is often used in video games due to the ubiquity of movies and images featuring it. Motion blur can be used to convey the impression of very fast motion. Another frequently used application of motion blur is masking of low frame rates.

3.1.2 Velocity Buffer Generation

All motion blur algorithms presented in this thesis rely on per-pixel motion vectors stored in a velocity buffer. These vectors represent the difference between a pixels current and previous position in image space and are used to determine the blur direction. This difference correlates to the relative movement that happened between two frames. Pixels can move because of camera movement as well as because of object movement. Figure 3.1 shows a velocity buffer featuring both object and camera motion. The car is moving to the lower left corner, while the camera is rotating clockwise around the y-axis.

Camera Motion

Computation of camera motion vectors is simple, as it requires only one matrix multiplication and one depth buffer sample. Specifically a reprojection matrix

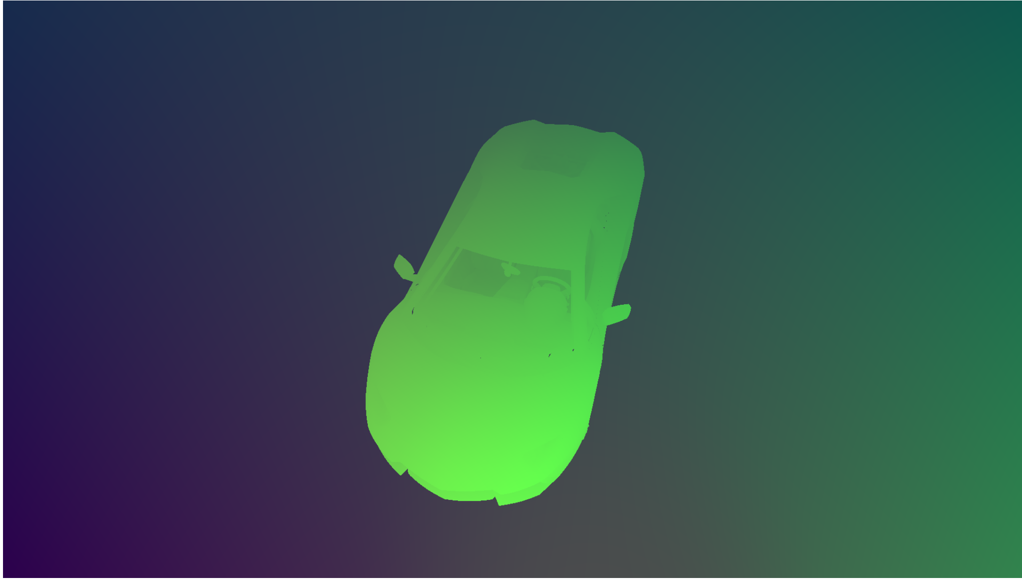

Fig. 3.1. Visualization of a velocity buffer featuring both object and camera motion.

$reproj$ needs to be constructed. This matrix consists of the product of the inverse view projection matrix of the current frame $viewProj_{cur}^{-1}$ and the view projection matrix of the previous frame $viewProj_{prev}$, shown in equation 3.1.

$$reproj = viewProj_{prev} * viewProj_{cur}^{-1} \tag{3.1}$$

To reproject a pixel, its position in normalized device coordinates $position_{NDC}$ must be determined. This can be done by mapping the pixel's texture space coordinate from the range [0, 1] to the range [-1, 1]. The texture space coordinate is three-dimensional. Its first two components $positionX_{UV}$ and $positionY_{UV}$ are given by the two-dimensional pixel position in texture space, whereas the third component $depth_{UV}$ needs to be sampled from the depth buffer. The construction of this new coordinate is shown in equation 3.2.

$$position_{NDC} = (positionX_{UV}, positionY_{UV}, depth_{UV})^T * 2 - 1 \tag{3.2}$$

The actual reprojection is done by multiplying $position_{NDC}$ with $reproj$. Since $reproj$ is a 4x4 matrix, $position_{NDC}$ needs to be expanded by a fourth component with the value 1. Equation 3.3 shows how a pixel is reprojected using its position in normalized device coordinates and the previously constructed reprojection matrix.

$$position_{prev} = reproj * (position_{NDC}, 1)^T \tag{3.3}$$

After the multiplication a perspective division must be performed. This is done by dividing the result vector by its fourth component. The first two components must now be mapped from normalized device space to texture space. The result of

this operation is the coordinate of the pixel in the previous frame corresponding to the current pixel. These operations are summarized in listing 3.1.

```
1  vec2 reproject(vec2 texCoord, mat4 reproj, sampler2D depthBuffer)
2  {
3     float depth = texture(depthBuffer, texCoord).x;
4     vec3 positionNDC = vec3(texCoord, depth) * 2.0 - 1.0;
5     vec4 positionReproj = reproj * vec4(positionNDC, 1.0);
6     return (positionReproj.xy / positionReproj.w) * 0.5 + 0.5;
7  }
```

Listing 3.1. Reprojection of a screen space coordinate into the previous frame using a uniform reprojection matrix.

It should be noted that the reprojection matrix is uniform across the whole image, since it only depends on the previous and current view projection matrices. As such it can be calculated once and sent as a uniform value to the shader. Since this technique requires only a single matrix multiplication and otherwise only depends on a depth buffer sample, it can be implemented ad-hoc in the same render pass that does the motion blur. This saves memory and bandwidth as the resulting velocity vector does not need to be stored in a buffer. The obvious drawback of this technique is that it only accounts for camera motion and ignores object motion.

Object Motion

To compute per-object motion vectors, one needs to account for changes in the model matrix of objects. This is best done while drawing them. Every object needs to save its previous frame's model matrix. When drawing the object, both the current and the previous model view projection matrices are sent to the vertex shader. Every vertex must then be transformed by both matrices. Afterwards a perspective division must be performed on both coordinates. The per-object motion vector is now given by mapping both coordinates from normalized device space to texture space and taking their difference. Listing 3.2 summarizes this process.

```
1  vec2 reproject(vec3 position, mat4 prevTransformation,
2                                mat4 curTransformation)
3  {
4     vec4 prevPosition = prevTransformation * vec4(position, 1.0);
5     vec4 curPosition = curTransformation * vec4(position, 1.0);
6     vec2 prev = (prevPosition.xy / prevPosition.w) * 0.5 + 0.5;
7     vec2 cur = (curPosition.xy / curPosition.w) * 0.5 + 0.5;
8     return cur - prev;
9  }
```

Listing 3.2. Reprojection of a screen space coordinate into the previous frame using a the previous object transformation.

The resulting per-object velocity vector should be written to a floating point texture of at least 16 bits per component as otherwise precision errors might occur. Alternatively the vector can be encoded and stored in smaller formats, however it may be desired for other effects relying on a velocity buffer (such as temporal

anti-aliasing or GTAO) to store the velocity vectors as precise as possible. It should be noted that the per-object velocity vectors account for both camera and object motion, because the previous frames transformation matrix used in the computation also contains the previous frames view projection matrix.

Although precise velocity vectors may be required for other effects, sometimes it is necessary to modify the velocities to achieve a more visually pleasing motion blur. One issue that might arise when using precise velocities is that motion blur is inversely dependent on frame rate. This is because the frame time is effectively used as exposure time when creating the velocity buffer. This effect can be corrected by scaling the velocities by a frame time dependent factor as seen in equation 3.4.

$$scaleFactor = TARGET_FRAME_TIME / frameTime \tag{3.4}$$

Vlachos et al. [Vla08] suggest reducing the influence of camera motion to about 15 %. This is easily done by linear interpolation between the previous and the current view projection matrices, shown in equation 3.5.

$$viewProj_{prevcorrected} = 0.15 * viewProj_{prev} + (1 - 0.15) * viewProj_{cur} \tag{3.5}$$

Modifying the velocity buffer as described might lead to artifacts in other effects also relying on it. Since motion blur should be applied immediately before tonemapping, other effects requiring velocities are usually already done at this point. This allows using precise velocities for most of the pipeline and only modifying the velocity buffer before using it for motion blur. The modification itself can be efficiently done by binding the velocity buffer as a read/write texture. A single pass of a compute shader can then read each velocity, modify it and write it to the same location, avoiding the additional texture required by ping-pong techniques.

3.1.3 Simple Motion Blur

Overview

The basic idea of most motion blur algorithms is to blur the image along certain directions, which are in most cases given per pixel [Gre03][Sou08]. The blur uses uniform weights and an arbitrary number of samples. Blur quality can be improved by raising the number of samples. Samples can be placed uniformly or jittered per pixel to trade banding for noise [Gre03]. Figure 3.2 illustrates what this effect looks like.

Fig. 3.2. Simple motion blur applied to an image of a moving car.

Implementation

A very simple implementation of this basic motion blur can be achieved by placing a fixed number of samples along the current pixel's velocity direction and averaging the sum of all samples. Motion blur should be done before tonemapping and can in fact be done in the same shader pass that performs the tonemapping. An implementation of this algorithm is given in listing 3.3.

```
1  vec3 motionBlur(vec2 texCoord, sampler2D colorTexture,
2                               sampler2D velocityTexture)
3  {
4      vec2 velocity = texture(velocityTexture, texCoord).xy;
5      vec3 sum = vec3(0.0, 0.0, 0.0);
6
7      const float SAMPLE_COUNT = 25.0;
8
9      for (float i = 0.0; i < SAMPLE_COUNT; ++i)
10     {
11         float t = mix(-1.0, 1.0, i / (sampleCount - 1.0));
12         vec2 offset = velocity * t;
13         sum += texture(colorTexture, texCoord + offset).rgb;
14     }
15
16     return sum / SAMPLE_COUNT;
17 }
```

Listing 3.3. A simple motion blur implementation.

This algorithm can additionally be improved by limiting the number of samples to a maximum value and skipping the blurring if the velocity is too small. Care should also be taken when performing motion blur on a still picture, as all velocity vectors will have a length of zero. This results in the loop exiting before the first iteration, causing *sum* to remain zero, which in turn makes the whole image black. For this reason it is necessary to check if the velocity is sufficiently large. If this

is the case, motion blur can be performed as usual. Otherwise the blur needs to be skipped and the pixel's color must be sampled from the input image. These improvements have been implemented in listing 3.4.

```
1  vec3 motionBlur(vec3 color, vec2 texCoord,
2                  sampler2D colorTexture, sampler2D velocityTexture)
3  {
4     vec2 velocity = texture(velocityTexture, texCoord).xy;
5
6     const float MAX_SAMPLES = 25.0;
7     float velocityLength = length(velocity * textureSize(velocityTexture, 0));
8     float sampleCount = min(floor(velocityLength), MAX_SAMPLES);
9     vec3 result = color;
10
11    if(sampleCount >= 1.0)
12    {
13       vec3 sum = vec3(0.0);
14
15       for (float i = 0; i < sampleCount; ++i)
16       {
17          float t = mix(-1.0, 1.0, i / (sampleCount - 1.0));
18          vec2 offset = velocity * t;
19          sum += texture(colorTexture, texCoord + offset, 0).rgb;
20       }
21
22       result = sum / sampleCount;
23    }
24
25    return result;
26 }
```

Listing 3.4. Improved implementation of a simple motion blur effect.

This simple motion blur algorithm has several shortcomings which need to be addressed in order to create plausible motion blur. The most prominent of these shortcomings is the fact that for each pixel only its own velocity vector is examined. This results in abrupt borders on the silhouette of moving objects as motion blur is only performed on pixels which are inside the object. The blur does not extend over static pixels around the moving object because these static pixels have a velocity of zero and as such have no motion blur. This is especially visible in figure 3.2 on the silhouette of the car. There have been many attempts at solving this problem, including dilating the velocities [Sou08] or stretching the geometry along its motion vector using a geometry shader [Gre03]. A popular modern approach to this problem seem to be tile-based scatter-as-gather algorithms.

3.1.4 Single-Direction Scatter-as-Gather

Overview

The motion blur algorithm developed by McGuire et al. [MHBO12] makes use of scatter-as-gather to combat sharp edges on object silhouettes. To understand scatter-as-gather, it is important to realize that what motion blur tries to accomplish is to scatter a pixel's color along a certain direction. However, scattering does not map very well to modern graphics hardware. For this reason it is beneficial to

invert the problem to make use of gathering. Specifically, each pixel must check if any other pixel scatters over it and if it does, gather this pixel and average it with all other pixels scattering over the current one. This formulation of the problem poses a significant difficulty when trying to implement it. To properly determine the pixels scattering over it, every pixel would have to sample the velocity of every other pixel in the image. This is infeasible in real-time and therefore an approximation must be made. McGuire et al. [MHBO12] propose dividing the image into tiles and for each tile determining the velocity with the maximum magnitude. Afterwards the same procedure needs to be applied to the 3x3 neighborhood of each tile. This ensures that each tile holds the largest velocity that might scatter over a pixel inside this tile. The texture holding the tiles can now be used when performing motion blur. Each pixel gathers samples along its tile's dominant velocity direction and determines for each sample, using the sampled velocity, if the sample scatters over it. This process eliminates the abrupt borders around moving objects because pixels outside the object can access information about the movement of pixels inside the object by sampling the tile texture. Figure 3.3 demonstrates how this algorithms improves upon the simple motion blur effect discussed in subsection 3.1.3. Note how there are no abrupt discontinuities on the silhouette of the car.

Fig. 3.3. Single-direction scatter-as-gather motion blur applied to an image of a moving car.

Implementation

The algorithm is divided in three passes. The first pass calculates the maximum velocity per tile, the second pass does the same for the local tile neighborhood and the final pass does the actual filtering. Calculating the maximum tile velocity is

a separable problem and as such can be accelerated by separating it into a horizontal and a subsequent vertical pass, effectively reducing the complexity from $O(m * n)$ to $O(m + n)$. Figure 3.4 shows the resulting tile texture when applying this operation to the velocity buffer in figure 3.1.

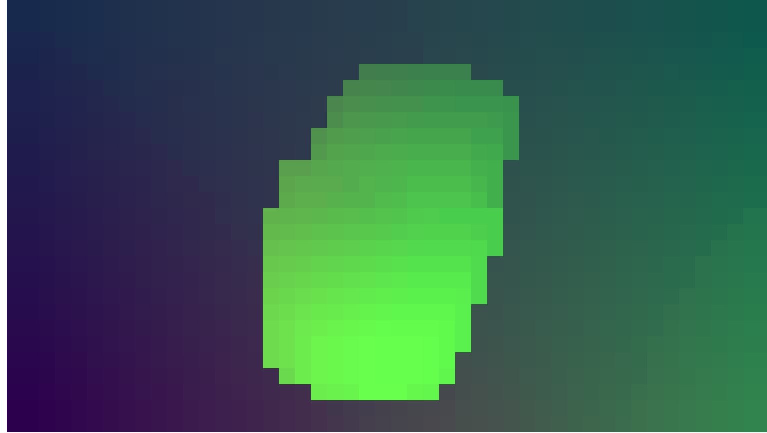

Fig. 3.4. The velocity tile texture holds the velocity with the maximum magnitude of each tile.

For the filtering pass McGuire et al. [MHBO12] distinguish three cases when calculating the weight of a sample:

1. The sample is in front of the center pixel and blurs over it.
2. Any sample behind a blurry center pixel.
3. Simultaneously blurry sample and center pixel.

The first case is the most obvious one, where another pixel is in front of the center pixel and blurs over it. In the second case the center pixel is blurry and the background behind it must be estimated by using the color of the sample. This introduces error but is necessary because the information about the scene behind the center pixel is lost at this point of the pipeline. In the last case both the sample and the center pixel are blurry and need to blur over each other.

The three cases presented above already hint that sample weighting is dependent on depth. In fact, neglecting depth leads to visual artifacts where fast moving background objects blur over static foreground objects, e.g. a car passing behind a street lamp. Note that this does not happen in the algorithm discussed in subsection 3.1.3 because it has the additional error that motion blur is only applied to pixels that are themselves moving. However when extending the blur over the object boundary using scatter-as-gather, depth information must be considered. This is done by a continuous classification in relative background and foreground.

McGuire et al. [MHBO12] use camera space depth values and assume a right handed coordinate system, so the depth values are linear and in the range of the

negative near plane and the negative far plane. They use the *cone()* function for the first two cases, the *cylinder()* function for the last case, where both sample and center pixel are blurry. The *softDepthCompare()* function is used to classify the sample as fore- or background. All functions are shown in listing 3.5.

```
1  float cone(float dist, float velocityMag)
2  {
3     return velocityMag > 0.0 ?
4     clamp(1.0 - dist / velocityMag, 0.0, 1.0) : 0.0;
5  }
6
7  float cylinder(float dist, float velocityMag)
8  {
9     return 1.0 - smoothstep(0.95 * velocityMag, 1.05 * velocityMag, dist);
10 }
11
12 float softDepthCompare(float a, float b)
13 {
14    return clamp(1.0 - (a - b) / SOFT_Z_EXTENT, 0.0, 1.0);
15 }
```

Listing 3.5. Utility functions for single-direction scatter-as-gather motion blur [MHBO12].

The full algorithm is shown in listing 3.6.

```
1  vec3 singleDirectionMotionBlur(vec2 texCoord, vec3 centerColor,
2              sampler2D colorTexture, sampler2D depthTexture,
3              sampler2D velocityTexture, sampler2D tileTexture)
4  {
5      vec2 vN = texture(tileTexture, texCoord).xy;
6
7      vec3 result = centerColor;
8
9      if(dot(vN, vN) <= dot(0.5, 0.5))
10     {
11         return centerColor;
12     }
13
14     vec2 vC = texture(velocityTexture, texCoord).xy;
15     float dC = texture(depthTexture, texCoord).x;
16
17     float vCLen = max(length(vC), 0.5);
18     float vNLen = length(vN);
19
20     float weight = 1.0 / vCLen;
21     vec3 sum = color * weight;
22
23     const int SAMPLE_COUNT = 25;
24
25     float j = random(-0.5, 0.5);
26
27     for (int i = 0; i < SAMPLE_COUNT; ++i)
28     {
29         if (i == SAMPLE_COUNT / 2)
30         {
31             continue;
32         }
33
34         float t = mix(-1.0, 1.0, (i + j + 1.0) / (SAMPLE_COUNT + 1.0));
35         vec2 S = (floor(texCoord * texSize) + floor(vN * t + 0.5)) / texSize;
36
37         float dS = texture(depthTexture, S).x;
38
39         float f = softDepthCompare(dC, dS);
40         float b = softDepthCompare(dS, dC);
41
42         vec2 vS = texture(velocityTexture, S).xy;
43
44         float vSLen = max(length(vS), 0.5);
45
46         float dist = abs(t) * vNLen;
47
48         float alpha = f * cone(dist, vSLen)
49                       + b * cone(dist, vCLen)
50                       + cylinder(dist, vSLen)
51                       * cylinder(dist, vCLen) * 2.0;
52
53         weight += alpha;
54         sum += alpha * centerColor;
55     }
56
57     return sum / weight;
58 }
```

Listing 3.6. Implementation of single-direction scatter-as-gather motion blur.

It should be noted that the algorithm in listing 3.6 trades banding for noise by using a random offset j. This can be used to disguise low sample counts. Another interesting detail is the fact that the center sample, which in most cases should be the center pixel, is skipped in the blur loop, as it is already accounted for in the

sum before entering the loop. Since the filtering pass has access to the tile texture, motion blur processing can be skipped altogether if the maximum velocity has a magnitude equal to or less than half a pixel. The presented algorithm creates plausible motion blur, respecting depth discontinuities, reconstructing occluded backgrounds and properly accounting for the reach of pixel scattering.

3.1.5 Multi-Direction Scatter-as-Gather

Overview

The motion blur algorithm presented in subsection 3.1.4, although in most scenes very convincing, introduces major error by only considering the maximum velocity of each tile. The visibility of this error depends on the scene but is maximized when many small objects are moving in different directions. This is because neighboring tiles have drastically different velocity directions. The problem also arises when a very small object is moving very fast in a different direction than most other objects in a tile. The high velocity of the small object masks all other velocities. McGuire et al. [MHBO12] place all samples exclusively along the tile maximum velocity direction. Samples are weighted based only on their velocity magnitude and not their direction. The error introduced by this leads to motion blur discontinuities between tile boundaries. Figure 3.5 demonstrates this visual artifact around the rear end of the car.

Fig. 3.5. Strong differences between tiles can lead to motion blur artifacts.

For this reason Guertin et al. [GMN13] propose a number of improvements to the motion blur algorithm discussed in subsection 3.1.4. The major contribution is that samples are no longer placed only along the tile maximum velocity direction but also along an additional direction to capture object movement that may have

been masked by other large velocities. Guertin et al. [GMN13] discuss several options for the secondary sampling direction and how many samples to place along it. They reason that if a pixel's velocity differs significantly from the dominant direction, then it should also be considered during sampling. Additionally, they modify the sample weighting to account for velocity directions.

Fig. 3.6. The improved sampling scheme of multi-direction scatter-as-gather motion blur almost completely removes the artifacts present in single-direction scatter-as-gather motion blur.

Implementation

Minimizing tile boundary discontinuities starts with selecting the proper dominant velocity. Guertin et al. [GMN13] suggest a tight culling of dominant velocities during the maximum neighbor velocity tile pass. In particular, they observed that neighboring velocity tiles can be ignored if their direction does not blur over the center tile. Although this reduces the error introduced by irrelevant dominant directions masking more important directions, this optimization was not implemented due to time constraints. When sampling the dominant direction from the tile texture during the motion blur pass, Guertin et al. [GMN13] propose stochastically offsetting the texture lookup for pixels near a tile edge, using a linear falloff. The probability of sampling a neighboring tile decreases with increasing distance to the tile edge.

Using the center pixel's velocity direction wastes samples when the velocity is very small. Guertin et al. [GMN13] propose linearly interpolating between the center pixel velocity direction and the direction perpendicular to the dominant one, based on the center pixel velocity magnitude. This is shown in equation 3.6, where $w_c(p)$

is the resulting secondary sampling direction, $v(p)$ is the velocity at point p, $v^{\perp}_{max}(t)$ is the direction perpendicular to the dominant direction in tile t and γ is a user defined threshold.

$$w_c(p) = lerp(v(p), v^{\perp}_{max}(t), (||v(p)|| - 0.5)/\gamma) \tag{3.6}$$

Guertin et al. [GMN13] discuss dynamically splitting the samples between the two sampling directions but ultimately decide that placing half of the samples in each direction is sufficient and that samples wasted on the less important direction do not cause objectionable artifacts under motion. Sampling directions are alternated each sample.

Compared to the motion blur algorithm developed by McGuire et al. [MHBO12], the sample weighting has been modified slightly. However, the basic cases remain the same as before:

1. The sample is in front of the center pixel and blurs over it.
2. Any sample behind a blurry center pixel.
3. Simultaneously blurry sample and center pixel.

The first case, where the sample blurs over the center pixel, is additionally weighted by the dot product of the sample's velocity direction and the sampling direction. This term restricts samples from contributing if their direction does not warrant blurring over the center pixel.
In the second case the center pixel's blur onto its surrounding causes a transparency. This is additionally weighted by the dot product of the direction computed using equation 3.6 and the dominant direction.
In the last case, both the center pixel and the sample blur over one another. The weight of this case is multiplied by the maximum of the two dot products of the first two cases.
Note that the secondary sampling direction is always the center pixel's direction. $w_c(p)$ is used in the dot product for the second (and third) case only. This may be a misinterpretation of the authors' intentions but it produces convincing results and is true to the algorithm's pseudo code listed in the paper [GMN13].

Guertin et al. [GMN13] propose to jitter the sampling using a halton sequence. The halton values are sampled from a lookup texture. Introducing jitter into the image trades banding for noise, masking low numbers of samples. Instead of a halton sequence, the implementation for this thesis uses interleaved gradient noise, published by Jimenez et al. [Jim14]. A function to compute this noise is given in listing 3.7.

```
1 float interleavedGradientNoise(vec2 v)
2 {
3     vec3 magic = vec3(0.06711056, 0.00583715, 52.9829189);
4     return fract(magic.z * dot(v, magic.xy));
5 }
```

Listing 3.7. A function to compute interleaved gradient noise [Jim14].

Interleaved gradient noise features a mix of pseudo-random and regular qualities, originally developed to be used for a shadow mapping filter kernel. The noise does not need to be precomputed and sampled from a texture. Instead it can be computed directly inside the motion blur shader, making the implementation simpler and faster.

The complete algorithm is shown in listing 3.8.

```
vec3 multiDirectionMotionBlur(vec2 texCoord, vec3 centerColor,
                sampler2D colorTexture, sampler2D depthTexture,
                sampler2D velocityTexture, sampler2D tileTexture)
{
    vec2 texSize = textureSize(colorTexture, 0);

    float j = interleavedGradientNoise(texCoord * texSize);
    vec2 vmax = texture(tileTexture, texCoord + jitterTile(texCoord)).rg;
    vmax *= texSize;
    float vmaxLength = length(vmax);

    if (vmaxLength <= 0.5)
    {
        return centerColor;
    }

    vec2 wN = vmax / vmaxLength;
    vec2 vC = texelFetch(velocityTexture, ivec2(gl_FragCoord.xy), 0).rg;
    vC *= texSize;
    vec2 wP = vec2(-wN.y, wN.x);
    wP = (dot(wP, vC) < 0.0) ? -wP : wP;

    float vCLength = max(length(vC), 0.5);
    const float gamma = 1.5;
    vec2 wC = normalize(mix(wP, vC / vCLength, (vCLength - 0.5) / gamma));

    const float N = 25;

    float totalWeight = N / (TILE_SIZE * vCLength);
    vec3 result = centerColor * totalWeight;

    float depthC = texelFetch(depthTexture, ivec2(gl_FragCoord.xy), 0).x;
    depthC = -linearDepth(depthC);

    vec2 dN[2] = { wN, vC / vCLength };

    for (int i = 0; i < N; ++i)
    {
        float t = mix(-1.0, 1.0, (i + j * 0.95 + 1.0) / (N + 1.0));

        vec2 d = bool(i & 1) ? vC : vmax;
        float T = abs(t) * vmaxLength;
        ivec2 S = ivec2(gl_FragCoord.xy) + ivec2(t * d);

        float depthS = -linearDepth(texelFetch(depthTexture, S, 0).x);

        float f = softDepthCompare(depthC, depthS);
        float b = softDepthCompare(depthS, depthC);

        vec2 vS = texelFetch(velocityTexture, S, 0).rg * texSize;

        float vSLength = max(length(vS), 0.5);

        int index = i & 1;
        float wA = abs(dot(wC, dN[index]));
        float wB = abs(dot(vS / vSLength, dN[index]));

        float weight = 0.0;

        weight += f * cone(T, vSLength) * wB;
        weight += b * cone(T, vCLength) * wA;

        weight += cylinder(T, min(vSLength, vCLength)) * max(wA, wB) * 2.0;

        totalWeight += weight;
        result += centerColor * weight;
    }

    return result / totalWeight;
}
```

Listing 3.8. Implementation of multi-direction scatter-as-gather motion blur

3.2 Depth of Field

3.2.1 Physical Basis and Motivation

Depth of field describes the distance to the plane of focus at which objects appear acceptably sharp in an image. An optical camera can perfectly focus only on objects at a certain distance. Everything at this distance is projected as a single dot onto the camera sensor. Objects in front or behind the plane of focus are no longer projected as a dot but as a circle where the radius depends on the distance to the plane of focus. This is called the circle of confusion. Since sensors used in cameras have a finite resolution, the circle manifests itself as blur only once it reaches a certain radius. The size of the circle of confusion scales with the aperture size. The quality of the blur in the out of focus region is commonly referred to as bokeh. The shape of the bokeh highlights depends on the shape of the aperture. In photography circular bokeh is preferred whereas in video games up until recently polygonal bokeh was used more commonly. Depth of field is an effect that occurs not only on film but also naturally in our eyes. As such it can add to the realism of synthetic images. Contrary to real cameras, images created by rasterization are perfectly sharp. Conceptually these images are created with a camera with an infinitesimal small aperture, called a pinhole lens. Recalling that the circle of confusion scales with aperture size, this means that rasterized images do not exhibit any blur. For this reason depth of field must be artificially added to the rendered image. It should be noted that convincing depth of field cannot be achieved with a gaussian blur, because the characteristic bokeh shapes only appear when using flat weighting.

3.2.2 Implementing a Virtual Camera

Achieving convincing depth of field strongly depends on the parameters used for the effect. One way to provide good parameters is to implement a virtual camera, mimicking a real optical camera. Such a virtual camera can be fed the same parameters as a real camera. Done correctly, the depth of field effect should be similar to the actual depth of field in photography. Among others, real cameras feature the following parameters:

- Focal Length
- Focal Plane
- Aperture Diameter
- Focal Ratio
- Shutter Speed
- Sensor Sensitivity (ISO)
- Film Width

Focal length F is the distance from the lens to the point where the incoming rays are brought to a focus. Objects inside the focal plane P are projected perfectly

sharp onto the sensor. Focal ratio, also known as f-stops or f-number, can be computed as the ratio between focal length and aperture diameter. Aperture diameter is usually set indirectly via manipulation of the focal ratio parameter. Shutter speed or exposure time is the period of time during which the sensor is exposed to light. ISO is the sensitivity of the sensor. Film width is the width of the sensor onto which the light is projected.

Shutter speed and ISO can be disregarded for depth of field computation as they have no influence on the circle of confusion. In photography, the field of view depends on film width and focal length. However, in real-time applications it is usually desired to change the field of view directly. Thus, focal length can be either derived from the field of view or set separately. Making field of view and focal length independent of one another has the advantage of producing a consistent effect. The obvious drawback is that the effect is no longer physically based, which may or may not be noticeable to the viewer. For the implementation for this thesis it was decided to couple field of view and focal length.

A virtual camera can then be implemented relying on only a few parameters:

- Focal Plane
- Focal Ratio
- Field of View
- Film Width

The focal plane can be determined automatically by taking a few samples from the center of the depth buffer and smoothing them temporally. This ensures that the center of the image is always in focus and that the focus does not change abruptly. Alternatively, the focal plane can be set manually. Usual focal ratio values lie in the range [1.4, 16]. Field of view is a parameter that is already present in real-time rendering applications. A commonly used film width is 35 mm. Note that this refers to the width of the complete film, including areas where no light is projected, like the perforated sides or the audio track. The width of the actual area onto which light is projected differs from format to format but can be easily looked up.

Based on these parameters, a physically based circle of confusion can be computed. The circle of confusion CoC depends on focal length F, aperture diameter A, the focal plane P and the distance D to the object that is being viewed. It can be calculated using equation 3.7.

$$CoC = |(\frac{F * D}{D - F}) - (\frac{F * P}{P - F})| * (\frac{D - F}{(F/A) * D}) \tag{3.7}$$

This simplifies to equation 3.8.

$$CoC = |A * (\frac{F * (P - D)}{D * (P - F)})| \tag{3.8}$$

Focal length can be derived from the field of view using equation 3.9.

$$F = \frac{0.5 * filmWidth}{\tan(fieldOfView/2)} \tag{3.9}$$

The aperture diameter can be computed from the focal length and the focal ratio using equation 3.10.

$$A = F/focalRatio \tag{3.10}$$

The distance D to the object being viewed is usually taken from the depth buffer. Care should be taken when implementing these equations to ensure all values using the same unit. The resulting circle of confusion computed using equation 3.8 must then be converted to pixels. This is easily done with equation 3.11.

$$CoC_{pixels} = CoC/(filmWidth/imageWidth) \tag{3.11}$$

The circle of confusion should be clamped to a maximum value to avoid cache trashing and depending on the technique, undersampling or massive overdraw. Achieving a desired look by manipulating these parameters is not always intuitive, which is why many implementations seem to use a more simple, not physically based function to compute the circle of confusion. The implementation for this thesis uses the physically based method presented here.

3.2.3 Filter Kernels

Most depth of field techniques, including two of the techniques discussed in this thesis, rely on a filter kernel to achieve the desired blur. Since circular bokeh seems to be the preferred shape, a circular kernel was used for this thesis. Sousa et al. [Sou13] use an algorithm proposed by Shirley et al. [SC97] to map unit square coordinates to a circle. It is possible to morph the resulting coordinates to form arbitrary n-gons, effectively simulating different apertures [Sou13]. As for the number of samples, both Sousa et al. [Sou13] and Jimenez et al. [Jim14] use kernels with 49 samples, which seems to be a reasonable number on modern graphics hardware. A visualization of the filter kernel is shown in figure 3.7.

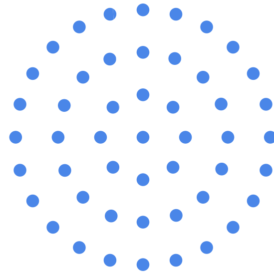

Fig. 3.7. The 49 sample filter kernel used for depth of field.

3.2.4 Simple Depth of Field

Overview

A very simple depth of field algorithm, similar to the motion blur algorithm discussed in subsection 3.1.3 can be created by scaling a 2D filter kernel by the current pixel's circle of confusion. However, similar to the motion blur, this technique leads to discontinuities in the blur when blurry foreground objects are in front of sharp background objects. The blur does not extend past the foreground object but instead ends abruptly at the silhouette. A very simple way to combat this artifact is to blur the circle of confusion with a weak gaussian blur. Although this introduces a number of other artifacts, it subjectively looks more pleasant. Figure 3.8 illustrates this simple depth of field effect.

Fig. 3.8. A simple bokeh depth of field implementation.

Implementation

The algorithm is separated into two passes. The first pass takes the blurred circle of confusion texture and the scene color texture as inputs and applies the filter kernel twice, once for the near field and once for the far field. The kernel is scaled by the near field and the far field circle of confusion respectively. Blurring should be done in half resolution to improve performance and widen the blur radius. This first pass outputs two textures containing the blurred near field and the blurred far field. An implementation of this pass is given in listing 3.9.

```
1  void simpleDepthOfFieldBlur(vec2 texCoord, vec2 texelSize, float nearCoc,
2        float farCoc, sampler2D nearColorTexture, sampler2D farColorTexture,
3        out vec3 nearColor, out vec3 farColor)
4  {
5     nearColor = vec3(0.0);
6     farColor = vec3(0.0);
7
8    for (int i = 0; i < KERNEL_SIZE; ++i)
9    {
10       vec2 kernelSampleCoord = SAMPLE_COORDS[i];
11
12      vec2 sampleCoordNear = nearCoc * texelSize * kernelSampleCoord;
13      nearColor += texture(uColorTexture, texCoord + sampleCoordNear).rgb;
14
15       vec2 sampleCoordFar = farCoc * texelSize * kernelSampleCoord;
16      farColor += texture(uColorTexture, texCoord + sampleCoordFar).rgb;
17    }
18
19     nearColor *= (1.0 / 49.0);
20     farColor *= (1.0 / 49.0);
21 }
```

Listing 3.9. Implementation of the blur pass of simple depth of field.

The second pass composites the full resolution source image with the blurred near and far field images. The images are composited back-to-front. The far field image and the full resolution image are interpolated based on the full resolution circle of confusion. The same is then done for the near field image and the result of the previous operation. The implementation for this is given in listing 3.10.

```
1  vec3 simpleDepthOfFieldComposite(vec2 texCoord, float nearCoc, float farCoc,
2        vec3 fullResolutionColor, vec3 nearColor, vec3 farColor)
3  {
4     vec3 result = mix(fullResolutionColor, farColor, clamp(farCoc, 0.0, 1.0));
5     return mix(result, nearColor, clamp(nearCoc, 0.0, 1.0));
6  }
```

Listing 3.10. Full resolution source image and blurred half resolution image are composited.

3.2.5 Sprite-Based Depth of Field

Overview

In stark contrast to scatter-as-gather based algorithms, a depth of field effect can also be achieved by using sprites [Dem07] [MD11] [KSS11] [Val13]. This sprite-based technique renders a sprite in the shape of a triangle or a quad for every

pixel in the sharp image. The size of the sprite is scaled by the circle of confusion of the corresponding pixel. The sprite can be textured with an artist defined texture, allowing for interesting effects. Similar to other depth of field algorithms, this one also splits the image in near and far field, rendering the sprite in either one of them, depending on the circle of confusion [KSS11]. The actual scattering effect is done by applying a texture to the sprite and tinting it with the color of the pixel for which the sprite was drawn. Figure 3.9 shows this depth of field technique.

Fig. 3.9. A depth of field effect can be achieved by drawing textured sprites for every pixel.

Implementation

Instancing is used to draw a number of quads equal to the number of pixels in the image. The position of the pixel for which the quad is drawn is reconstructed from the gl_InstanceID value. The quad is scaled by the radius of the circle of confusion of the source pixel and moved to the pixel's position. The pixel color and the inverse size of the quad are passed to the fragment shader. Sprite based depth of field uses a near field and a far field texture. These textures could be created by using multiple render targets or drawing all quads twice. A texture atlas is used instead. The near field is placed in the left half of the target texture and the far field in the right half. Quads are positioned in either half, depending on their circle of confusion. Bleeding between the images is avoided by sending a flat value to the fragment shader, indicating to which side of the atlas the quad belongs. The fragment shader uses this information to clip the quad at the border. Figure 3.10 illustrates the atlas containing both textures.

Fig. 3.10. The near field texture is stored in the left half and the far field texture is stored in the right half of the texture atlas.

Quads with a size of less than a pixel are moved outside the viewport, stopping further processing. This is a performance optimization and needs to be accounted for during composition. Vertex shader code for this stage is given in listing 3.11.

```
1  vec4 spriteDepthOfFieldVertexShader(vec2 position, vec2 texCoord,
2          int instanceID, int width, int height, float coc,
3          sampler2D colorTexture, out float near, out vec4 color)
4  {
5      vec2 pixelPos;
6      pixelPos.x = instanceID \% width;
7      pixelPos.y = instanceID / height;
8
9      color = vec4(texture(colorTexture, texCoord).rgb, 1.0 / (coc * coc));
10     near = coc > 0.0 ? 1.0 : 0.0;
11
12     // scale by coc
13     position *= coc;
14
15     // move to pixel position
16     position += pixelPos;
17
18     // scale to pixel size
19     position *= (1.0 / vec2(width, height));
20
21     // position on atlas
22     position = position * vec2(1.0, 2.0) - vec2(near, 1.0);
23
24     // kill primitive if coc is too small
25     return vec4(position, coc < 1.0 ? -2.0 : 0.0, 1.0);
26 }
```

Listing 3.11. Vertex shader code for sprite-based depth of field.

The fragment shader samples the bokeh texture and multiplies the passed in pixel color and inverse sprite area with the color of the texture sample. This tints the color by the supplied texture and allows for arbitrary bokeh shapes using the alpha channel as mask. If the quad is leaking onto the other image inside the atlas, the alpha value is set to zero, effectively clipping the quad at the border. Color and alpha channel are written additively into the framebuffer. Fragment shader code for this stage is given in listing 3.12.

```
1  vec4 spriteDepthOfFieldFragmentShader(vec2 texCoord,
2       sampler2D spriteTexture, float near, vec4 color)
3  {
4     vec4 spriteColor = texture(spriteTexture, texCoord).rgba;
5     vec4 result = vec4(color * spriteColor.rgb, spriteColor.a) * color.a;
6
7     float leftSide = texCoord < 0.5 ? 1.0 : 0.0;
8     result *= vNear == leftSide = 1.0 : 0.0;
9
10    return result;
11 }
```

Listing 3.12. Fragment shader code of sprite-based depth of field.

Compositing is done similar to the depth of field algorithm presented in subsection 3.2.4, except that the color channels of the near and far textures are divided by their alpha channel before doing the linear interpolations. The color value represents a weighted sum, where the alpha value is the total weight. Dividing be alpha normalizes the sum. At this stage it becomes apparent why alpha is weighted by the sprite area. Doing this is a means of preserving energy when scattering the color of a pixel over an area. Neglecting to account for the sprite area causes harsh discontinuities in the blur and introduces error into the final image by adding energy. In the vertex shader, quads with a very small circle of confusion were skipped. This must be accounted for during compositing. This can be done by downsampling the full resolution source texture and using the downsampled color where required. An implementation of this is given in in listing 3.13.

```
1  vec3 spriteDepthOfFieldComposite(vec4 nearField, vec4 farField,
2       vec3 fullResColor, vec3 halfResColor, vec2 coc)
3  {
4     farPlane.rgb = farPlane.a > 0.0 ?
5     farPlane.rgb / farPlane.a : farPlane.rgb;
6     nearPlane.rgb = nearPlane.a > 0.0 ?
7     nearPlane.rgb / nearPlane.a : nearPlane.rgb;
8
9     vec3 color = mix(fullResColor, halfResColor,
10    clamp(max(coc.x, coc.y), 0.0, 1.0));
11    color = mix(color, farField.rgb, clamp(farField.a, 0.0, 1.0));
12    color = mix(color, nearField.rgb, clamp(nearField.a, 0.0, 1.0));
13    return color;
14 }
```

Listing 3.13. The full resolution source texture and the blurred half resolution textures are composited.

3.2.6 Scatter-as-Gather Depth of Field

Overview

The rationale behind the motion blur algorithms discussed in subsection 3.1.4 and subsection 3.1.5 also applies to depth of field. While motion blur scatters a pixel only along a single direction, depth of field blurs pixels in all directions. Thus it is necessary to expand the scatter-as-gather approach to two dimensions. Similar to motion blur, the image is divided into tiles, where each tile holds the maximum circle of confusion that may overlap pixels inside the tile. Again, the image is split

into near field and far field, based on the circle of confusion. Both images are then processed together. Processing can be skipped if the tile texture indicates that no other pixel blurs over the center pixel. A filter kernel is then used to sample pixels from the local neighborhood. The contribution of each sample is determined based on its circle of confusion. Figure 3.11 shows the scatter-as-gather principle in two dimensions. Samples contribute only if their circle of confusion reaches over the center pixel. A second pass is employed to fill holes inside the bokeh shapes produced by the first pass. In a final pass, the near and far field images are composited with the full screen image [Sou13]. Figure 3.12 illustrates the effect of this depth of field technique.

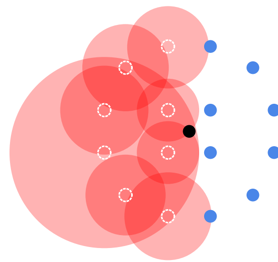

Fig. 3.11. Scatter-as-gather in two dimensions. Source: [Jim14], modified.

Fig. 3.12. Scatter-as-gather depth of field applied to the scene.

Implementation

Generation of the tile texture is done similar to the motion blur techniques discussed in subsection 3.1.4 and subsection 3.1.5. The input circle of confusion tex-

ture and the resulting tile texture are shown in figure 3.13.

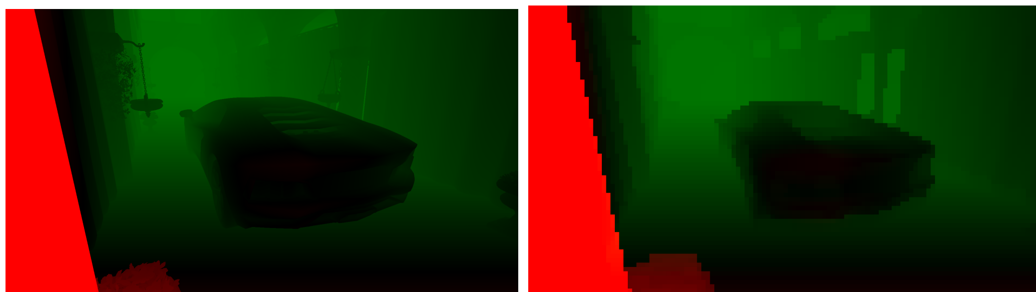

Fig. 3.13. The circle of confusion texture (left) is used to generate a tile texture (right), holding the maximum circle of confusion of every tile.

To reduce bandwidth and improve the smoothness of the blur, the source color texture is downsampled to half resolution using a bilateral filter. The image is split into near and far plane, depending on the sign of the circle of confusion of each pixel. When creating the near plane image, full screen pixels belonging to the far plane are ignored. The final circle of confusion value associated with the resulting half resolution pixel is the maximum value of the four source pixels. The downsampled images are premultiplied by their circle of confusion. The CoC is then stored in the alpha channel. These measures significantly reduce leaking of sharp foreground objects. Since the circle of confusion is a unit of full resolution pixels and the effect is done in half resolution, it must be divided by two to preserve the radius. The effect is implemented using floating point textures with four channels and 16 bits per channel. This ensures enough precision for the source image which likely contains large values.

The main blur pass processes both images together. Processing can be skipped if the tile texture indicates that the maximum circle of confusion in the tile is zero. Using precomputed sampling locations (see subsection 3.2.3), samples are taken from the downsampled images and weighted according to their circle of confusion. Near and far field are weighted differently. Sousa et al. [Sou13] suggest using the scatter-as-gather approximation only on the near field. Near field sample weighting compares the distance between the center pixel and sample with the circle of confusion of the sample. Jimenez et al. [Jim14] propose using the number of rings in the kernel instead of pixels as unit during weighting. This allows implementing a smooth falloff, avoiding harsh discontinuities. It is important to account for energy conservation, especially in the near field. Taking energy conservation into account is as simple as dividing the sample weight by the area of its circle of confusion. The area can be computed using equation 3.12.

$$area_{coc} = \pi * CoC^2 \tag{3.12}$$

Failing to consider energy conservation results in unrealistic looking blobs of color in the near field. This is very noticeable for blurry foreground objects in front of sharp objects in the background. Near field samples are weighting using the function shown in listing 3.14.

```
1  float weightNearField(float sampleDistance, float sampleCoc,
2        float pixelToRingScale)
3  {
4     float sampleWeight = clamp(sampleCoc * pixelToRingScale -
5     sampleDistance * pixelToRingScale + 1.0, 0.0, 1.0);
6
7     sampleWeight *= 1.0 / (PI * sampleCoc * sampleCoc);
8
9     return sampleWeight;
10 }
```

Listing 3.14. Near field weighting function.

True to the technique proposed by Sousa et al. [Sou13], far field weighting does not use scatter-as-gather. Instead the center pixel's circle of confusion is used to scale the kernel radius. Samples are weighting using the function given in listing 3.15.

```
1  float weightFarField(float centerCoc, float sampleCoc)
2  {
3     return (sampleCoc >= centerCoc) ? 1.0 : clamp(sampleCoc, 0.0, 1.0);
4  }
```

Listing 3.15. Far field weighting function.

Although not as correct as the scatter-as-gather approximation, this still produces plausible results, presumably because errors in the far field are less noticeable. These weights are used to produce a weighted sum, which is normalized at the end of the pass.

Large kernel radii result in undersampling, which manifests itself as visible rings in the bokeh shape. This artifact can be combated with a second pass in which the *max()* function is used on the local pixel neighborhood. The kernel for this pass is a circular 3x3 kernel, scaled by the pixel size. Jimenez et al. [Jim14] suggest using a median filter instead. However, this is not as fast and was not implemented due to time constraints.

The compositing pass is run in full resolution and combines the near field, the far field and the full resolution source image. Near and far field images are divided by their circle of confusion, undoing the premultiplication done in the downsampling pass. Compositing is then done in two steps. First, the far field image is composited with the full resolution image using linear interpolation and the full resolution circle of confusion as weight. The weight is clamped to the range [0, 1]. The near field is composited on top of the result of the previous operation. Unlike the far field, the near field uses the circle of confusion sourced from the near field texture as weight. This weight is also clamped to the interval [0, 1]. Using the circle of confusion from

the near field ensures that blurry foreground objects bleed over sharp background objects. Sousa et al. [Sou13] note the importance of using bicubic upsampling to improve the image quality.

3.2.7 Depth of Field Implementation Considerations

Specular aliasing is an artifact commonly found in real-time rendering, especially when using physically based rendering. Specular aliasing happens because the pixel raster undersamples the specular response of distant objects [Kar14]. This leads to pixels rapidly changing their brightness from frame to frame, causing the impression of flickering. This flickering is highlighted by depth of field effects, making the artifact even more objectionable. There are many ways to combat specular aliasing, one of which is to use post-processing anti-aliasing before the depth of field effect [Kar14]. The implementation for this thesis uses Subpixel Morphological Anti-Aliasing (SMAA), developed by Jimenez et al. [JESG12]. SMAA detects edges and smoothes them. SMAA features an optional temporal filter, using a jittered projection matrix and temporal coherency to reconstruct subpixel details. Similar to other post-processing anti-aliasing techniques, SMAA expects tonemapped values in gamma space as input. Thus, the image is tonemapped and converted to gamma space using the tonemapping operator suggested by Karis [Kar14]. After the anti-aliasing pass, the image is then converted back to linear color space and the tonemapping operator is inverted, producing high dynamic range values. Theoretically using anti-aliasing in such a way introduces error because the color values no longer correspond to the depth values. However in practice, this is only visible when using temporal filtering, as the jittering produces different depth values each frame even for a still image.

3.3 Screen Space Ambient Occlusion

3.3.1 Physical Basis and Motivation

Ambient occlusion is the name of a set of techniques aimed at approximating global illumination. When a surface is lit by a light source, part of the light is absorbed and part of it bounces off of it. Some of these light rays find their way directly to the observer. This is called direct illumination. Other rays bounce onto different surfaces which again absorb and reflect a certain ratio of the incoming light. After a number of bounces, a fraction of the light might reach the observer. This is called indirect illumination. Figure 3.14 illustrates this concept.

Algorithms and techniques that compute or approximate indirect illumination are commonly referred to as global illumination (GI). Calculating GI in real-time is difficult and remains a problem of active research. A number of algorithms have been developed that approximate select aspects of GI in real-time, ambient occlusion being one of the most well known. Ambient occlusion techniques compute the

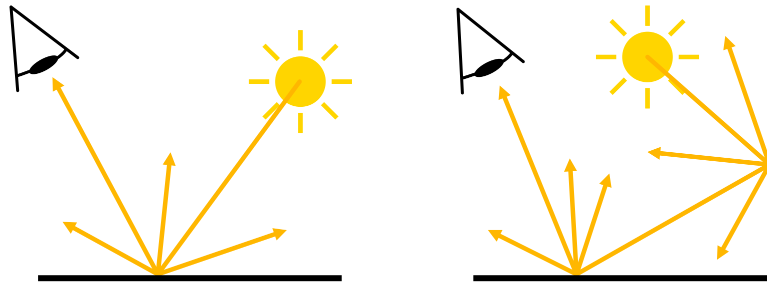

Fig. 3.14. Direct illumination (left) and indirect illumination (right).

probability of point P to be hit by indirect light rays originating from other surfaces. Properly applying ambient occlusion to a scene gives the viewer an increased perception of depth and gives cues on the relation of objects to one another. Essentially it makes objects "stick to the ground". Figure 3.15 illustrates this effect.

Fig. 3.15. Without ambient occlusion, objects do not look connected (top left). Adding ambient occlusion gives visual cues about spatial relations between objects (top right). Visualization of the ambient occlusion term (bottom).

The definition of ambient occlusion can be derived from the rendering equation, a simplified version of which is shown in equation 3.13, where $L_o(\omega_o)$ is the light exiting from a (implicit) point P towards direction ω_o. This light is the sum of the emitted radiance $L_e(\omega_o)$ and the integral of incoming radiance $L_i(\omega_i)$ from direction ω_i at P over the hemisphere. The incoming radiance is weighted by the bidirectional reflectance distribution function (BRDF) $f_r(\omega_i, \omega_o)$. $(n \cdot \omega_i)$ is a weak-

ening factor based on incident angle.

$$L_o(\omega_o) = L_e(\omega_o) + \int_{\Omega} f_r(\omega_i, \omega_o) L_i(\omega_i)(n \cdot \omega_i) d\omega_i \tag{3.13}$$

Ambient occlusion techniques assume a lambertian BRDF ($\frac{\rho d}{\pi}$) and that there is no emission, resulting in equation 3.14.

$$L_o(\omega_o) = 0 + \int_{\Omega} \frac{\rho d}{\pi} L_i(\omega_i)(n \cdot \omega_i) d\omega_i \tag{3.14}$$

Further assuming that the scene is illuminated by a constant white dome and that there is only one bounce of light results in equation 3.15. $L_i(\omega_i)$ has been replaced with a constant factor, simulating a constant hemispherical light. A new term $V(\omega_i)$ has been added, which determines the visibility for a given direction. $V(\omega_i)$ is defined in equation 3.16.

$$L_o(\omega_o) = \int_{\Omega} \frac{\rho d}{\pi} V(\omega_i) 1 (n \cdot \omega_i) d\omega_i \tag{3.15}$$

$$V(\omega_i) = \begin{cases} 1, & \text{if } \omega_i \text{ hits the sky} \\ 0, & \text{otherwise} \end{cases} \tag{3.16}$$

Rearranging the terms yields equation 3.17, the definition of ambient occlusion.

$$L_o(\omega_o) = \rho d \frac{1}{\pi} \int_{\Omega} V(\omega_i)(n \cdot \omega_i) d\omega_i = \rho d V_d \tag{3.17}$$

According to this definition, ambient occlusion algorithms integrate the cosine weighted visibility $V(\omega_i)$ of each direction ω_i inside the hemisphere oriented around the normal n at P.

The result of the ambient occlusion computation is a probability in the range [0, 1], which contrary to the name actually models visibility, not occlusion. Visibility and occlusion can be trivially converted into one another as shown in equation 3.18 and 3.19. This is of relevance because some techniques discussed in this thesis initially compute occlusion and convert it to visibility in a later step.

$$visibility = 1 - occlusion \tag{3.18}$$

$$occlusion = 1 - visibility \tag{3.19}$$

Recalling the assumptions made above, albedo (ρd) multiplied by ambient occlusion (V_d) is the ground truth lighting for the case of:

1. A lambertian surface.
2. No emission.
3. A constant white dome illuminating the scene.
4. A single bounce of light.

This is almost always not the case in real scenes but is a sufficient approximation. The visibility value computed by ambient occlusion techniques should be applied to the indirect diffuse lighting term, since a lambertian surface was assumed. The indirect specular term should be modulated by a specular occlusion term. Applying ambient occlusion only to the indirect diffuse term implies that it cannot be applied as a pure post-processing effect, since it must have been already computed when lighting the scene. However, this is sometimes disregarded due to artistic reasons and ambient occlusion is applied to all light terms, including direct illumination.

The question then remains how to efficiently evaluate and integrate the visibility term $V(\omega_i)$ in real-time. This is difficult as it requires a description of the scene and a way to integrate over the hemisphere. The desire for good looking ambient occlusion in real-time has spawned a number of algorithms working in screen space, called Screen Space Ambient Occlusion (SSAO).

3.3.2 Original Algorithm

Overview

The first game to feature SSAO was *Crysis* made by *Crytek* in 2007 [Kaj09]. Ambient occlusion is usually calculated by tracing rays in random directions inside a hemisphere centered around the normal of a point P, for which the occlusion is to be computed. If a ray hits geometry, P gains occlusion from that direction. However this process is not suitable for real-time computation on modern graphics hardware. This is primarily because a complete scene description is needed against which to test the rays, which does not map well to current graphics chips. Additionally the large number of rays to trace for a convincing effect poses another problem. SSAO solves both problems using approximations. The important contribution is that ambient occlusion is computed using only screen space information sourced from the depth buffer. The depth buffer holds a partial scene description and can be easily used as a texture in a shader. The second major problem, the large number of rays, is solved by not tracing rays at all. Instead, occlusion is computed as the ratio of empty space to geometry around P. The algorithm devised by Crytek also does not rely on normals and instead sources all information exclusively from the depth buffer. This has the benefit of avoiding taking additional texture samples from a second texture. This also allows to employ SSAO

in an environment where no screen space normals are available, e.g. in a purely forward renderer. The texture produced by SSAO holds values in the range [0, 1], where higher values mean less occlusion. During lighting calculation the texture is sampled and the resulting value is applied to the indirect diffuse lighting term. Figure 3.16 shows the SSAO term.

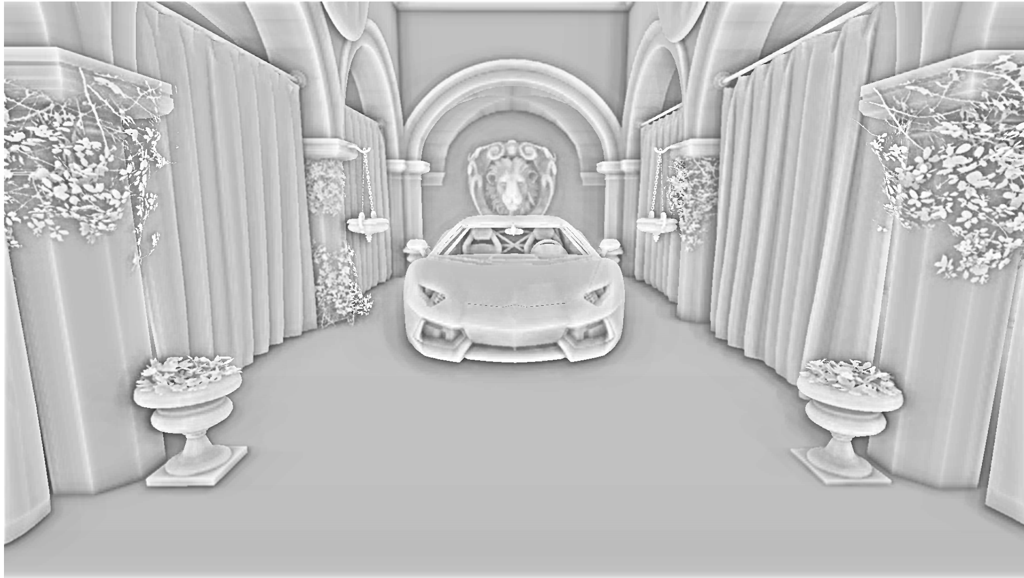

Fig. 3.16. SSAO computed for a scene of mixed complexity.

Implementation

At the core of SSAO lies the aforementioned occlusion computation based on the ratio of empty space to geometry. This is done using a spherical kernel centered around point P. The kernel places samples randomly inside a sphere, where most samples are concentrated near the center. This effectively simulates distance attenuation by reducing the weight of occluded samples further away. Ambient occlusion is then calculated by taking the ratio of samples inside geometry to samples outside geometry. Specifically, samples are evaluated in a loop and their results averaged, producing the ambient occlusion at P. In a correct simulation only the hemisphere centered around the normal of P would be considered. However due to using depth buffer information only, the normal is not available and as such the technique places samples inside a sphere instead of a hemisphere. Alternatively, a normal could be computed using the finite difference method, albeit doing so requires additional samples, further slowing down the already bandwidth bound algorithm. The spherical kernel is not scaled dependent on scene depth or similar parameters. Instead it is applied purely in screen space. The consequence of this is that ambient occlusion is calculated for small nearby foreground features and large distant background features alike. The sample positions of the kernel are com-

puted on the fly in the shader. The original implementation for *Crysis* used only 16 samples. When using such a low number of samples without further measures, the resulting image shows unpleasant looking banding. A commonly used trick to combat banding is the usage of noise, which in the case of SSAO is introduced as random rotations applied to the kernel. These random rotations are sourced from a precomputed four-by-four texture holding random vectors with three elements. The texture is tiled over the image in such a way, that every four pixels the same rotation is used. The resulting image now displays noise instead of banding, which is preferable but still not optimal. The repeating pattern of the noise is clearly visible in figure 3.17.

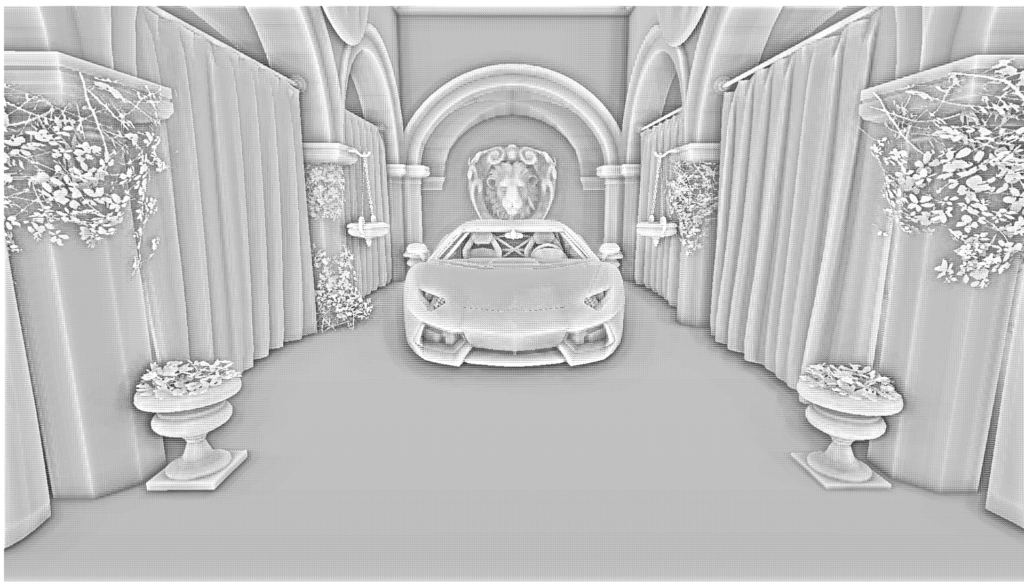

Fig. 3.17. The four-by-four SSAO noise pattern is clearly visible.

In order to remove the high frequency noise in the ambient occlusion texture produced by the first pass, a second pass is run that blurs the texture using a four-by-four blur kernel. Care must be taken when blurring to preserve edges. For this reason the blur considers the depth of all samples and weights samples according to their depth relative to the center pixel. The implementation for this thesis uses the weighting function 3.16.

```
1 float bilateralBlurWeight(float centerDepth, float sampleDepth)
2 {
3     float weight = abs(sampleDepth - centerDepth) / (centerDepth * 0.1);
4     return clamp(1.0 - weight, 0.0, 1.0);
5 }
```

Listing 3.16. Bilateral weighting function for SSAO blur.

This function allows samples with a maximum depth difference of 10 % to the center pixel depth, which is useful when looking at a plane at an angle, e.g. looking

down a long corridor. If the weighting is too restrictive, the noise is not properly eliminated and if it is too lenient, ambient occlusion will bleed over edges. Applying the technique as described will yield ambient occlusion halos around foreground objects. The reason for this is that the depth buffer is not an accurate description of the scene as information about objects behind the frontmost object is lost. When samples of kernels belonging to pixels in the background land on foreground objects, the sample is always occluded, even if there is a large depth difference between the background pixel and the sample on the foreground object. In the original SSAO implementation for *Crysis*, this artifact was avoided by weighting each sample by the function in listing 3.17.

```
1 float sampleWeight(float accessibility, float centerDepth, float sampleDepth)
2 {
3     float rangeIsInvalid = (centerDepth - sampleDepth) / sampleDepth;
4     return mix(accessibility, 0.5, clamp(rangeIsInvalid, 0.0, 1.0));
5 }
```

Listing 3.17. Sample weighting function for SSAO.

This weighting defaults to a value of 0.5 if the sample is deemed to far away. This can be explained by considering that the sampling kernel used is spherical. When applied to a flat surface like a wall, half the samples are inside and the other half outside of geometry. This means that an ambient occlusion value of 0.5 corresponds to the maximum visibility value achievable using a proper hemispherical kernel. However, due to using a spherical kernel there are instances where the visibility value exceeds 0.5. Figure 3.18 shows the sampling kernel applied to different surfaces and illustrates the problem of using a spherical kernel. Half the samples of the kernel at position P_1 are occluded, resulting in an occlusion value of 0.5 on a flat surface. Most samples at P_2 are occluded, causing a darkening effect in corners. At position P_3 fewer samples than at P_1 are occluded. This has the effect that compared to the rest of the object, corners appear to be brighter. Although this is an artifact of the approximation done by using a spherical kernel, it can be considered desirable as it makes the three-dimensionality of the scene more apparent [Kaj09].

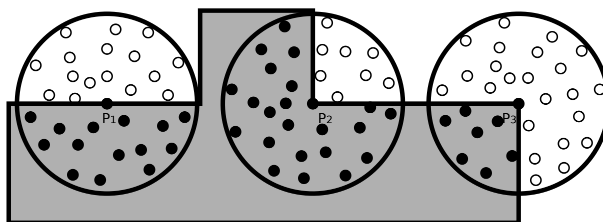

Fig. 3.18. Visualization of the SSAO kernel on different surfaces.

Since SSAO produces only one value in the range [0, 1] per pixel, a texture with one channel and 8 bit per texel is sufficient to hold the results. Of course when using ping-ponging for the subsequent blur, a second texture is required. However, the

implementation for this thesis uses textures with two channels where the second channel stores the depth. Although depth can also be sourced from the depth buffer, storing depth with the ambient occlusion value, reduces the texture fetches in the blur pass by half.

3.3.3 Hemisphere Kernel

Overview

A common modification of the SSAO technique discussed in subsection 3.3.2 is the usage of a hemispherical kernel instead of a spherical one. Since the hemisphere needs to be oriented around the normal, either screen space normals must be available as texture or additional depth samples must be taken to compute a normal using the finite difference method. The hemispherical sampling kernel is shown in figure 3.19. Since ambient occlusion is defined over a hemisphere, using a hemispherical kernel makes the approximation more accurate. Figure 3.20 shows the resulting ambient occlusion term.

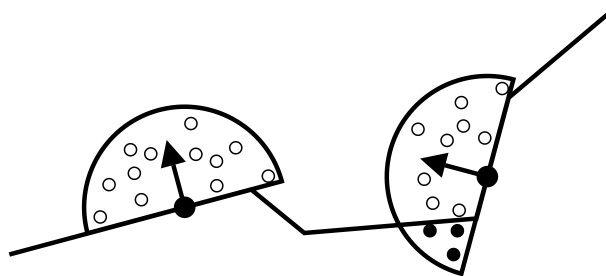

Fig. 3.19. Visualization of a hemispherical SSAO kernel on different surfaces.

Implementation

The implementation for this thesis uses a deferred environment in which screen space normals are readily available as part of the geometry buffer. For this reason the modified SSAO technique sources its normals from the geometry buffer. Contrary to the original algorithm, the sample positions of the kernel are precomputed using a pseudo-random number generator and stored in uniform values. Similar to the original algorithm most samples are still placed near the center of the kernel, attenuating the weight of further away occluders. Noise is introduced a second time when the kernel is rotated. This is done by calculating a tangent basis using a random vector supplied by a four-by-four texture. The precomputed kernel samples are then transformed from tangent space to view space. The view space position of the current pixel is offset using the transformed sample position. The position computed this way is then projected into screen space, where its coordinate is used as final sample coordinate. The major consequence of this process

Fig. 3.20. The ambient occlusion term calculated with a hemispherical kernel.

is that the kernel is no longer placed purely in screen space but in view space, making it cover the same radius in world space regardless if the pixel is close or far away from the camera. Compared to the original technique, this is closer to the ground truth. Another minor difference is the depth weighting of samples. Samples are only valid if the geometry they land on is inside the world space kernel radius. The function is listed in listing 3.18.

```
1  float sampleWeight(float occlusion, float centerViewSpaceDepth,
2                                 float sampleViewSpaceDepth, float kernelRadius)
3  {
4      float rangeIsValid = smoothstep(0.0, 1.0,
5      kernelRadius / abs(centerViewSpaceDepth - sampleViewSpaceDepth));
6      return occlusion * rangeIsValid;
7  }
```

Listing 3.18. Sample weighting function for SSAO (Hemisphere).

The noise texture is tiled across the image in the same way as in the original algorithm. The subsequent blur pass is also identical.

3.3.4 Horizon-Based Ambient Occlusion

Overview

Horizon-Based Ambient Occlusion (HBAO) was first introduced 2008 by Bavoil et al. [BS08] [BSD08]. It approaches the problem of screen space ambient occlusion radically different than the two previous techniques discussed in subsection 3.3.2 and subsection 3.3.3. Bavoil et al. [BS08] realized that the ambient occlusion integral over the hemisphere can be split into a double integral of which the inner integral can by solved analytically. Said double integral can be derived from the

ambient occlusion definition in equation 3.17 as shown in equation 3.20. The direction ω_i is now expressed in polar coordinates.

$$V_d = \frac{1}{\pi}\int_{\Omega} V(\omega_i)(n \cdot \omega_i)d\omega_i = \frac{1}{\pi}\int_0^{2\pi}\int_0^{\frac{\pi}{2}} V(\theta,\Phi)|sin(\theta)|d\theta d\Phi \tag{3.20}$$

The inner integral integrates the occlusion for one direction inside the circle. The outer integral rotates this slice around, covering the complete hemisphere. HBAO aligns the hemisphere with the view direction, resulting in a circular projection in screen space. The inner integral can be computed analytically by calculating the angle between the view space tangent and the maximum horizon as seen from the point for which to compute ambient occlusion. Approximating visibility this way is sufficient as the depth buffer represents a height field, for which it is impossible to determine all unoccluded directions. Therefore determining the angle described above is possibly the best way to extract visibility information from a depth buffer for a certain direction. Figure 3.21 illustrates this reasoning. Using a height field, it is impossible to know that the directions shown in green are unoccluded.

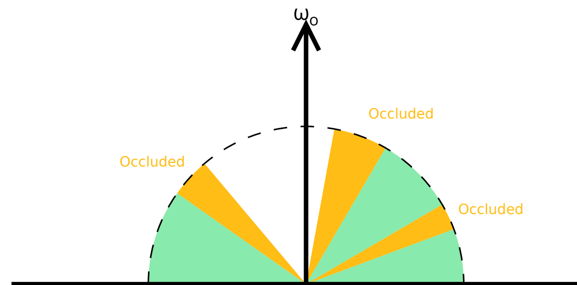

Fig. 3.21. Information about unoccluded directions (green) cannot be extracted from a height field. ω_o is the view vector. Source: [JWPJ16b], modified.

The outer integral is calculated numerically by averaging the occlusion of a fixed number of directions. The last step is the conversion of occlusion to visibility as shown in equation 3.18. The technique can be decomposed into the following steps:

1. For each direction find the maximum horizon.
2. Compute the occlusion based on the angle between tangent and horizon.
3. Sum the occlusion of all directions and average the result.

Figure 3.22 illustrates the ambient occlusion term calculated with HBAO.

Implementation

Since HBAO calculates ambient occlusion for a hemisphere in view space, its screen space projection must be determined first. Due to being aligned with the Z-axis in

Fig. 3.22. The ambient occlusion term calculated with HBAO.

view space, the hemisphere can be approximated as a disk. The function in listing 3.19 demonstrates how to calculate the disk radius in pixels. Note that the HBAO implementation for this thesis is based on NVIDIA's Direct3D SDK 11 implementation of HBAO and therefore uses a left-handed coordinate system, which means that the sign of the z-component is flipped.

```
1  float projectHemisphereToDisk(float radius, float viewSpaceDepth,
2                     float width, float height, float fovy)
3  {
4     float aspectRatio = height / width;
5     float focalLength = 1.0 / tan(fovy * 0.5) * aspectRatio;
6     float radiusUV = 0.5 * radius * focalLength / viewSpaceDepth;
7     float radiusPix = radiusUV * vec2(width, height);
8     return radiusPix;
9  }
```

Listing 3.19. Projecting the hemispherical kernel from view space to screen space.

At this point further processing can be canceled if the computed radius is less than a pixel. Conversely, if the radius is large enough, the step size and number of samples is computed. The maximum kernel radius should be limited as otherwise geometry very close to the camera would warrant kernel radii spanning the whole image, severely reducing performance due to texture cache thrashing. Further optimization can be done here by clamping the number of samples to the radius length in pixels. Doing this avoids sampling the same pixels multiple times. Another item of interest is that the per pixel randomization that is usually employed by SSAO algorithms is introduced at this point. Similar to previous algorithms, random values are supplied by a four-by-four texture that is tiled over the image. If the calculated number of steps is larger than the maximum number of steps, a random value is used to introduce jittering into the kernel.

HBAO requires per-pixel normals. However, unlike the SSAO implementation discussed in subsection 3.3.3, HBAO sources its normals from the depth buffer using the finite difference method. More precisely, it calculates the tangent basis vectors. The choice of whether to use screen space normals from the geometry buffer or to reconstruct them from the depth buffer is primarily an artistic one and will be discussed later.

Having computed the kernel radius, the sample placements and the tangent basis vectors, the actual ambient occlusion calculation can begin. Solving the double integral can be naturally expressed using a loop, evaluating a different direction at each iteration and summing its occlusion. Determining the direction is easily done by computing it on the unit circle using the cosine and sine functions. The direction obtained this way is then rotated by a random angle. The sine and cosine values of the matrix expressing this rotation are precomputed and passed as part of the random texture to the shader.

The inner integral is evaluated for each direction. The view space tangent vector of each direction is computed using the previously calculated tangent basis vectors. Searching for the maximum horizon is then done using the angle of the tangent vector from the view space axis as a starting point. For each sample, its view space position is calculated. If it lies within the kernel radius and represents a higher horizon than the previous maximum horizon, the occlusion cast by this horizon is added to the total ambient occlusion. Horizon search is illustrated in figure 3.23. The black line represents the one-dimensional height field and points S_0, S_1, S_2 and S_3 the sample locations. Note that S_1 lies below the previous maximum horizon and therefore does not cast any occlusion.

Occlusion is calculated as shown in listing 3.20.

```
1  float calculateAo(float sinOfNewHorizon, float sinOfOldHorizon)
2  {
3      return sinOfNewHorizon - sinOfOldHorizon;
4  }
```

Listing 3.20. Analytical calculation of ambient occlusion.

The occlusion value calculated this way is additionally attenuated based on the distance of the sample to the center point. As the function parameters in listing 3.20 imply, the maximum horizon value is updated with the new horizon and used in the next iteration as previous horizon.

The last step of the ambient occlusion calculation is to average the previously computed occlusion and invert it to obtain the visibility value that is later used during lighting. Optionally the visibility value can be raised to a power to increase the effect's visual impact. Similar to the previous two discussed SSAO techniques, the randomness introduced by the tiled four-by-four random textures results in

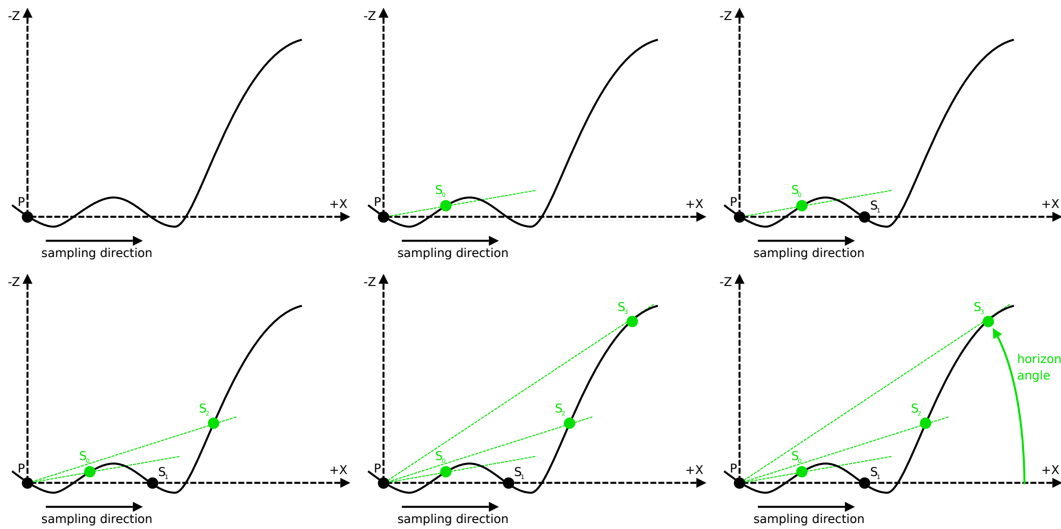

Fig. 3.23. Horizon search with four steps (left to right and top to bottom). P is the point to be shaded and S_i are the sample locations. Every found horizon is highlighted in green. Source: [BS08], modified.

a visible noise pattern. This is solved using the same blur pass as described in subsection 3.3.2.

3.3.5 Ground-Truth Ambient Occlusion

Overview

The development of Ground-Truth Ambient Occlusion (GTAO) was motivated by the desire to compute accurate screen space ambient occlusion, closely matching the ground truth and fast enough to run in about 0.5 milliseconds on consoles [JWPJ16a] [JWPJ16b]. Fundamentally it builds on Horizon-Based Ambient Occlusion, where visibility is computed analytically, based on a horizon search. Contrary to HBAO, horizon search is done on a 180° slice, as opposed to a 90° slice. For each slice, the maximum horizon is searched for in in both directions. The view vector serves as the spherical integral axis, which is why horizons can be in the negative view hemisphere. Therefore horizon search must be done on the whole sphere. The maximum horizons are later clamped to the normal aligned hemisphere. Unlike some other SSAO algorithms such as HBAO, GTAO does not do any per-sample attenuation. This is because it is not possible to match ground truth renderings with per-sample attenuation. Not using attenuation also allows to integrate the inner integral with respect to the view vector rather than the view-space tangent plane. This in turn halves the number of integrals, reducing computations since calculations can be shared. Integrating from view vector to horizon means that GTAO computes visibility instead of occlusion. Figure 3.24 shows the reference frame for GTAO. The horizons θ_1 and θ_2 are searched for in both directions.

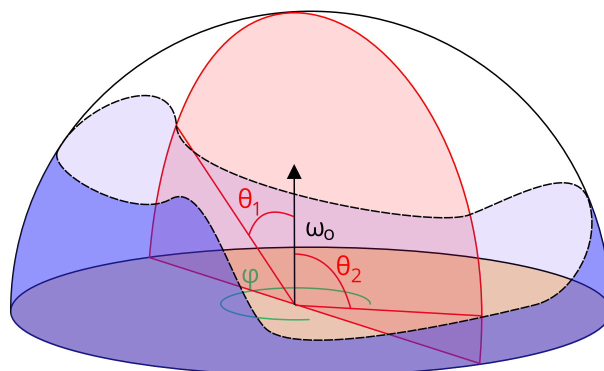

Fig. 3.24. GTAO reference frame. ω_o is the view vector, θ_1 and θ_2 are the horizon angles and φ is the angle around the integration axis. Source: [JWPJ16b], modified.

The original HBAO technique employs uniform weighting of visibility, disregarding the cosine term in the ambient occlusion definition in equation 3.17. Since GTAO is concerned with producing accurate values, it is necessary to account for this. The double integral of visibility as computed by GTAO is shown in equation 3.21. θ and Φ are ω_i expressed in polar coordinates and γ is the angle between the integral axis (the view vector) and the normal at P projected onto the slice.

$$V_d = \frac{1}{\pi} \int_0^{pi} \int_{\frac{-\pi}{2}}^{\frac{\pi}{2}} V(\theta, \Phi) cos(\theta - \gamma) |sin(\theta)| d\theta d\Phi \tag{3.21}$$

Even though GTAO already reduced the number of computations by sharing calculations and integrating only once per slice, integrating the outer integral numerically is still too expensive. GTAO solves this by heavily relying on temporal coherence, spreading the numerical computation of the outer integral over multiple frames. Figure 3.25 illustrates the GTAO ambient occlusion term.

Implementation

Similar to HBAO, GTAO also aligns the kernel hemisphere with the view vector. This means that the disk projection of the hemisphere in screen space can be computed the same way as in HBAO. The implementation for this thesis sources its normals from depth buffer. Noise is used to jitter the sampling stride and rotate the directions. This noise can be passed in as uniform values or computed directly in the shader. The noise is not pseudo-random like the noise used for the previously discussed SSAO algorithms. As has been already hinted at, GTAO heavily relies on temporal coherency. Therefore the noise has a spatial component and a temporal component. Listing 3.21 shows how the noise is computed.

Fig. 3.25. The ambient occlusion term calculated with GTAO.

```
1  // x = spatial direction / y = temporal direction
2  // z = spatial offset / w = temporal offset
3  vec4 getNoise(int frame, ivec2 coord)
4  {
5      vec4 noise;
6
7      noise.x = (1.0 / 16.0) * ((((coord.x + coord.y) & 0x3 ) << 2)
8      + (coord.x & 0x3));
9      noise.z = (1.0 / 4.0) * ((coord.y - coord.x) & 0x3);
10
11     float rotations[] = { 60.0, 300.0, 180.0, 240.0, 120.0, 0.0 };
12     noise.y = rotations[coord \% 6] * (1.0 / 360.0);
13
14     float offsets[] = { 0.0, 0.5, 0.25, 0.75 };
15     noise.w = offsets[(coord / 6 ) \% 4];
16
17     return noise;
18 }
```

Listing 3.21. A function to calculate noise values depending on time and pixel position [JWPJ16b].

GTAO computes ambient occlusion only for one slice per frame, which reduces the core of the algorithm to a single loop searching for the maximum horizons in both directions of the slice. Horizons can be computed trivially using the function in listing 3.22.

```
1  float computeHorizon(vec3 centerViewSpacePos, vec3 sampleViewSpacePos)
2  {
3      vec3 D = normalize(sampleViewSpacePos - centerViewSpacePos);
4      vec3 V = -normalize(centerViewSpacePosP);
5      return dot(V, D);
6  }
```

Listing 3.22. Calculation of potential new horizons during horizon search.

Although no per-sample attenuation is used, the horizons of further away samples can be attenuated by a distance based factor, ensuring ground-truth results on close geometry and attenuating on far away geometry. Attenuation is done while computing the horizon and before comparing it with the current maximum. After completion of the horizon search loop, the cosine of the angle that was obtained by using the *dot()* function must be inverted by applying the *acos()* function on the maximum horizon angles h_1 and h_2. Having found the maximum horizons, the integral in equation 3.22 can be calculated. Jimenez et al. [JWPJ16a] give an analytic solution, listed in equation 3.23. γ is the angle between the integral axis and the normal.

$$AO_{slice} = \int_0^{h_1} cos(\theta - \gamma)|sin(\theta)|d\theta + \int_0^{h_2} cos(\theta - \gamma)|sin(\theta)|d\theta \tag{3.22}$$

$$AO_{slice} = \frac{1}{4}(-cos(2h_1 - \gamma) + cos(\gamma) + 2h_1 sin(\gamma)) + \frac{1}{4}(-cos(2h_2 - \gamma) + cos(\gamma) + 2h_2 sin(\gamma)) \tag{3.23}$$

Equation 3.23 assumes that the normal at P lies within the slice, which in general is not the case. Timonen [Tim13a] showed that the normal can be substituted by the normal projected onto the slice. This projection is shown in figure 3.26.

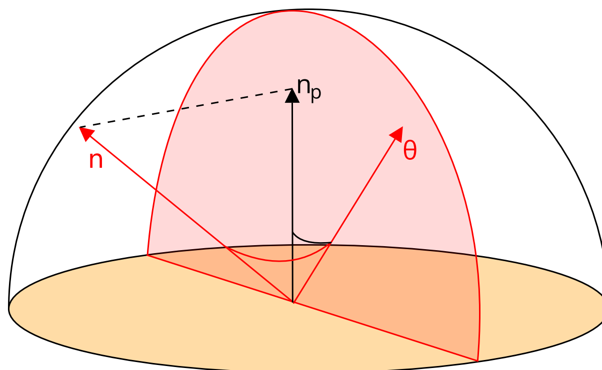

Fig. 3.26. The normal n does not lie in the plane and must be projected onto it, yielding the projected normal n_p. θ is the horizon. Source: [JWPJ16b], modified.

The integral must then be corrected by the length of the projected normal to be equal to the original assumption of the normal lying withing the plane. The implementation of GTAO for this thesis derives gamma by applying the *acos()* function to the dot product of the integral axis and the (projected) normal. A pitfall that was encountered here was that the sign of the angle was lost this way. The

solution was to reconstruct the sign from the sign of the dot product of the projected normal and the tangent of the slice. Calculation of γ is shown in listing 3.23.

```
float computeGamma(vec3 samplingDir, vec3 viewDir, vec3 normal)
{
    // project normal onto slice plane
    vec3 planeN = normalize(cross(samplingDir, viewDir));
    vec3 projectedN = normal - dot(normal, planeN) * planeN;

    projectedN = normalize(projectedN);

    // calculate gamma
    vec3 tangent = cross(viewDir, planeN);
    float cosGamma = dot(projectedN, viewDir);

    // reconstruct sign of gamma
    float gamma = acos(cosGamma) * sign(-dot(projectedN, tangent));

    return gamma;
}
```

Listing 3.23. Calculation of γ.

The introduced noise is removed from the image using a depth aware bilateral blur similar to the one used for SSAO and HBAO. Since the integration of the outer integral is spread over multiple frames, a temporal filter must be applied after the spatial filter to combine the results of previous frames with the current one. The filtered ambient occlusion of the previous frame is found by reprojecting the current pixel using screen space motion vectors. Calculation of these vectors is discussed in subsection 3.1.2. Temporally filtered ambient occlusion is then computed from the previous frame's temporally filtered ambient occlusion and the current frame's spatially filtered ambient occlusion. This is shown in equation 3.24, where α is the blend factor in the linear interpolation function *lerp()*.

$$filteredAO_{n+1} = lerp(filteredAO_n, AO_{n+1}, \alpha) \tag{3.24}$$

The reprojected ambient occlusion of the last frame does not always correspond to the same position in world space as the ambient occlusion of the current frame. This can happen due to camera or object movement and needs to be detected in order to avoid accumulating incorrect values. Previous values can be discarded if their coordinate lies outside the frame. When geometry is visible in the current frame that did not exist in the previous frame, a disocclusion happened. Detecting disocclusions requires the previous frame's depth buffer to correctly determine world space positions. As explained when discussing the spatial filter in subsection 3.3.2, depth is saved in a second channel alongside the ambient occlusion value. Jimenez et al. [JWPJ16b] suggest using the depth difference between the current and the previous pixel and the magnitude of the motion vector as a heuristic for the likeliness of both pixels referring to the same position. The implementation for this thesis computes world space positions and compares these instead of comparing just depth [BA12] [MSW10]. Even though this requires some additional matrix multiplications, it seems to be a more robust approach to the problem,

which makes it worth it. Another problem, specific to ambient occlusion, arises when using temporal filters. Ambient occlusion depends on the surrounding environment. It is therefore necessary to test if the neighborhood of a pixel still casts occlusion onto the pixel. Disregarding this leads to trails behind moving objects because previous ambient occlusion values are not rejected even though the moving object no longer casts occlusion onto the pixel. Jimenez et al. [JWPJ16a] [Jim16] propose using filtered neighborhood clamping, a technique primarily used in temporal anti-aliasing techniques. Neighborhood clamping examines the local neighborhood of a pixel and clamps the pixel to the minimum and maximum values. When using neighborhood clamping with GTAO, a low pass filter is applied to the neighborhood. This is easily achieved by placing the sample positions between pixels and using a linear sampler. Jimenez et al. [JWPJ16a] further suggest employing a dynamic convergence time, dependent on frame time and converging faster for fast moving objects. This was disregarded for the implementation of this thesis because it introduced flickering into the image. A constant α value is used instead. Of course this value is multiplied with the weighting factors described above. As a final step, both filtered ambient occlusion and depth are stored in the result texture, which is used for shading the current frame and as input for the temporal filter in the next frame. Figure 3.27 shows the ambient occlusion term before spatial and temporal denoising.

Fig. 3.27. Without the spatial noise reduction, the noise pattern is clearly visible (left). The result of this operation (right) still needs to be filtered temporally.

3.3.6 Screen Space Ambient Occlusion Implementation Considerations

Except for the original SSAO, all other SSAO techniques discussed in this thesis rely on screen space normals. Sourcing these normals from the geometry buffer in a deferred environment produces incorrect values in most cases. This is due to the fact that normals are usually interpolated across each triangle. Figure 3.28 illustrates the problem. The interpolated normal (gray) differs from the actual normal (green), resulting in false occlusion as the tangent does not match with the actual geometry. Using normal mapping adds further deviation to the normals. Normals produced this way do not correspond to the geometry as defined by the depth

buffer.

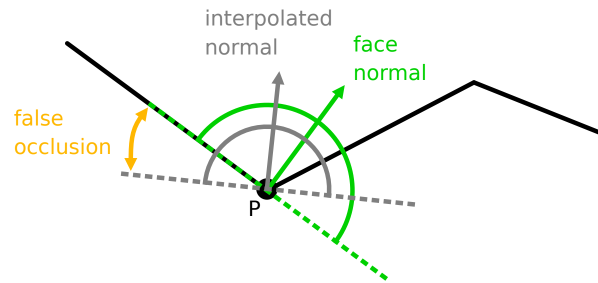

Fig. 3.28. False occlusion is detected in horizon based approaches when using interpolated normals. Source: [BS08], modified.

Reconstructing normals from depth buffer yields almost always values that correspond to the actual geometry. The drawback of this approach however is that ambient occlusion might accentuate the faceted nature of certain meshes.
Deciding for one or the other is an artistic choice. Since precomputed small scale ambient occlusion can be supplied by a texture, the implementation for this thesis almost always uses normals reconstructed from the depth buffer.

SSAO techniques are usually bandwidth limited and can profit from complementary cache-aware techniques such as Line Sweep Ambient Occlusion (LSAO) [Tim13a], deinterleaved rendering [BJ13] and performing the effect in half resolution. LSAO computes a data structure allowing efficient horizon search for a given direction. Deinterleaved Rendering deinterleaves the frame according to the four-by-four noise pattern commonly used with SSAO. The effect is then computed seperately on each deinterleaved quarter resolution slice. Since all texture fetches inside a slice share the same access patterns and cover a greater distance in full resolution screen space than in the quarter resolution screen space, the texture cache is used much more efficiently.
Downsampling the image and performing SSAO in half resolution is a commonly done optimization. SSAO is a low frequency effect, which lends itself to a lower resolution. Still, care must be taken when downsampling the depth buffer and upsampling the SSAO result in order to avoid bleeding the effect over edges. Downsampling depth is problematic because averaging the values results in depth values that do not correspond to any surface in the source image.
Due to time constraints these techniques have not been implemented for this thesis.

3.4 Measurement Method

The framework used to implement the techniques discussed in this thesis encapsulates all graphics pipeline state setup and draw calls specific to a shader into *RenderPass* and *ComputePass* objects. An asynchronous OpenGL timer query is

used to measure the elapsed GPU time of a pass. This includes all API calls necessary for the draw call(s), including binding of textures, framebuffers and changes of the viewport or per-fragment operations. Measuring this way ensures that all additional performance cost associated with an effect is accounted for.
The techniques have been tested on two different systems, a desktop computer and a laptop. The hardware specifications of both systems are listed in table 3.1.

	Desktop	Laptop
CPU	Intel i5 4690K @4.6 GHz	Intel i5 7300HQ @3.5 GHz
GPU	AMD R9 390 8 GB	NVIDIA GeForce GTX 1050 2 GB
RAM	16 GB	8 GB
OS	Windows 7	Windows 10

Table 3.1. Hardware specifications of the two test systems.

Automatic benchmarking functionality has been implemented to increase the reproducibility of the measurements. There are seven predefined camera views (scenes). Upon starting the benchmark, camera controls are locked to prevent the user from invalidating the measurements by moving the camera. The benchmark runs in several passes, measuring the performance of different techniques every pass. Each pass collects data from 200 frames. When a benchmark pass is completed, its average timings are automatically written to a file. Time is measured in milliseconds. All techniques discussed in this thesis consist of multiple intermediary steps, each of which is encapsulated in either a *RenderPass* or a *ComputePass* object. The performance of every step is measured separately. The total run time of a technique is calculated as the sum of all its steps. An overview of the techniques measured at each benchmark pass is shown in table 3.2.

Pass	Motion Blur	Depth of Field	Screen Space Ambient Occlusion
0	Simple	Simple	SSAO
1	Single-Direction	Sprite Based	SSAO (Hemisphere)
2	Multi-Direction	Tile Based	HBAO
3	Multi-Direction	Tile Based	GTAO

Table 3.2. Overview of techniques measured at each benchmark pass.

Although measurements have been made for all seven scenes, only two representative scenes per effect have been picked to discuss in this thesis. All measurements were made at a resolution of 1920x1080.

3.4.1 Motion Blur Test Setup

The cost of motion blur can be folded with other fullscreen passes, such as the tonemapping pass. Motion blur is executed in the same shader as, and immediately before tonemapping and gamma correction. Unfortunately, this setup prohibits taking accurate measurements of motion blur performance. The performance impact of motion blur usually depends on magnitude and direction of the per-pixel velocities. In order to create reproducible results, a constant velocity is written to the velocity buffer. Therefore motion blur has been moved to a separate pass for the purpose of measuring. Figure 3.29 shows the constant velocity used while benchmarking.

Fig. 3.29. Using a constant velocity ensures reproducibility.

Although this does not reflect real-world performance, it allows for a fair comparison between different techniques. Having constant velocities makes motion blur independent of scene complexity. For this reason motion blur results are discussed for only one scene.

3.4.2 Depth of Field Test Setup

Depth of field is implemented with an instantaneous auto focus. The distance of the focal plane is calculated as the linear depth of the pixel in the center of the image. The radius of the circle of confusion increases in both near and far field when the focal plane is close. As an increased circle of confusion radius results in an increased blur radius and thus a larger performance hit, one of the representative scenes has been selected to serve as a worst case scenario. The other scene has been selected for its average depth of field strength, resulting in an average performance hit. The focal ratio is available as a parameter in the graphical user interface. Since the circle of confusion radius directly depends on the focal ratio, care was taken to take all measurements with a constant focal ratio of 1.4.

3.4.3 Screen Space Ambient Occlusion Test Setup

Most screen space ambient occlusion techniques discussed in this thesis dynamically scale the sampling kernel to be uniform in view space. This means that

the screen space distance covered by the kernel increases with decreasing distance to the camera. Similar to depth of field and other techniques relying on a sampling kernel, a larger kernel size results in worse performance due to texture cache misses. One of the two chosen representative scenes contains a balanced mix of close and further away geometry, while the other contains mainly close geometry. These criteria should allow to reason about both the typical and the worst case performance impact. The parameters used during the benchmark are shown in table 3.3 for SSAO (Hemisphere), table 3.4 for HBAO and table 3.5 for GTAO. The parameters were chosen with the goal of producing similar visual results. Note that SSAO (Hemisphere) and HBAO use the same number of samples. Since GTAO is temporally filtered, it uses less samples then the other techniques.

Parameter	Value
Kernel Size	64
Kernel Radius	2.0

Table 3.3. SSAO (Hemisphere) parameters used during the benchmarks.

Parameter	Value
Directions	8
Steps	8
Radius	1.0
Maximum Pixel Radius	256

Table 3.4. HBAO parameters used during the benchmarks.

Parameter	Value
Steps	8
Radius	2.0
Maximum Pixel Radius	256

Table 3.5. GTAO parameters used during the benchmarks.

4

Results and Discussion

4.1 Motion Blur

Figure 4.1 gives a visual overview over the three motion blur techniques discussed in this thesis. When comparing the different techniques with respect to their visual quality, the simple motion blur algorithm looks least convincing due to the edge discontinuities. Single-direction scatter-as-gather looks slightly better than multi-direction scatter-as-gather, because the latter spreads all samples evenly along two directions. In this image there is only one dominant velocity direction, causing the multi-direction motion blur technique to waste samples on a secondary direction and sampling the main direction with only half the samples. However, this is not very noticeable under motion. Conversely, the tile discontinuity artifacts present in single-direction motion blur are not very noticeable under motion either. Whether situational undersampling is worse than visual artifacts is subjective.

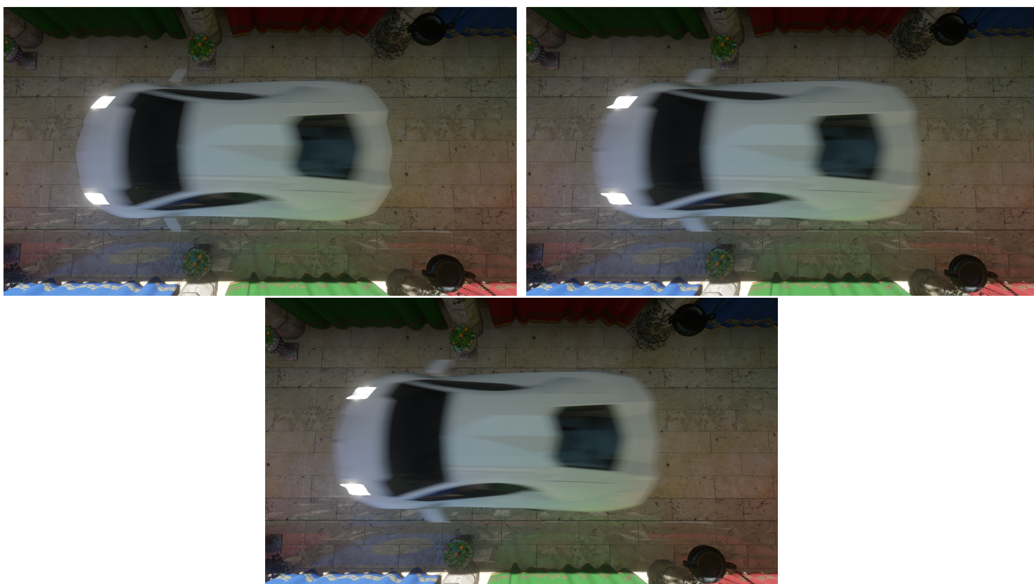

Fig. 4.1. Simple motion blur (top left), single-direction motion blur (top right) and multi-direction motion blur (bottom) applied to the same scene.

Table 4.1 and table 4.2 show the timings of each motion blur technique on the desktop test system and the laptop test system respectively. The velocity correction pass is a constant cost in all techniques. The two tiling passes necessary for the scatter-as-gather motion blur techniques make up only a fraction of the total time spent on the algorithm. The run time of all three techniques is dominated by the blur pass, presumably due to the large number of texture fetches done in full resolution. Both scatter-as-gather algorithms spend considerably more time in the blur pass then the simple motion blur algorithm. Since all three techniques take the same maximum number of samples and have a similar memory access pattern, this might hint at possible performance improvements achievable through optimization. Multi-direction scatter-as-gather motion blur is slightly faster in the blur pass than single-direction scatter-as-gather. This is unexpected because both algorithms are very similar. Although outside the scope of this thesis, extensive profiling and examination of shader code disassembly might be helpful in finding an answer to this phenomenon.

	Simple	Single-Direction	Multi-Direction
Velocity Correction	0.0701084	0.0698048	0.0699729
Velocity Tile Max	n/a	0.0756721	0.0759832
Velocity Neighborhood Tile Max	n/a	0.00705657	0.00704838
Blur	0.65022	2.17453	2.05213
Total	0.709724	2.32309	2.19394

Table 4.1. Motion blur timings on the desktop test system. All timings are in milliseconds.

	Simple	Single-Direction	Multi-Direction
Velocity Correction	0.354058	0.353593	0.354268
Velocity Tile Max	n/a	0.195535	0.195888
Velocity Neighborhood Tile Max	n/a	0.00498672	0.0053576
Blur	0.995369	3.55847	3.40387
Total	1.33069	4.10491	3.93877

Table 4.2. Motion blur timings on the laptop test system. All timings are in milliseconds.

Compared to one another, all techniques have the same relative cost, regardless of the test system. This indicates that the relative performance of all techniques is independent of hardware. When deciding for a technique, implementation effort should also be considered. The simple motion blur algorithm produces acceptable results and is trivial to implement. Especially in light of its quick run time, this technique might be useful on weaker hardware or in a time constrained environment. While offering a significantly higher quality, both scatter-as-gather

algorithms incur a greater performance impact and are more difficult to implement. Since both techniques are visually almost equal, choosing between them is subjective. However, the implementation of multi-direction scatter-as-gather is measurably faster and behaves more stable when taking a still image. Furthermore, multi-direction scatter-as-gather will most likely be the preferred technique going forward, as increasing hardware performance will make spending samples on multiple directions more feasible.

4.2 Depth of Field

Figure 4.2 and figure 4.3 show the three different depth of field techniques discussed in this thesis. Visually, all three techniques look very similar, which is due to the fact that great care was taken to implement a consistent effect. Still, there are subtle visual differences between the different approaches. Looking closely at the bokeh shape of the simple depth of field implementation reveals undersampling artifacts in the form of visible rings. These rings only manifest themselves when the circle of confusion reaches the maximum radius. Scatter-as-gather depth of field reduces undersampling artifacts by employing a seperate fill pass. Unfortunately, this fill pass grows the bokeh shape slightly, making the blur a little stronger than in the other techniques. However, this is a matter of preference and can be solved by using a median filter instead of a maximum filter. Another unpleasant artifact visible in the simple depth of field technique are halos around near field objects, such as the leaves of the plant in figure 4.2. This can be attributed to a lack of leak reduction measures and the indiscriminate blurring of the circle of confusion texture.
Sprite based depth of field naturally does not suffer of undersampling. Correctly implemented, it is potentially one of the most realistic ways to simulate bokeh depth of field, as it models the blur conceptually true to reality. However, correctly implementing sprite based depth of field is difficult because sprites approaching a radius of less then a pixel are problematic to rasterize and thus need to be handled separately. The implementation of sprite based depth of field suffers from slight artifacts at the transition from rasterized sprites to downsampled color. Investing more time into this technique would most likely solve this problem.
Scatter-as-gather depth of field is the most advanced of all three depth of field techniques presented here. It displays almost no leaks, thanks to careful downsampling and circle of confusion premultiplication. Since the circle of confusion is clamped to a maximum radius, specially tweaked for this technique, undersampling artifacts are not visible. Visually, this algorithm is the most impressive of the three.
Table 4.3 and table 4.4 show measurements for the first depth of field test scene, table 4.5 and table 4.6 for the second. Similar to motion blur, all depth of field techniques share a common pass. The compute pass calculating the circle of confusion texture incurs a fixed performance impact, regardless of technique. Overall, the simple depth of field technique is the fastest, performing slightly faster than

Fig. 4.2. Simple depth of field (top left), sprite based depth of field (top right) and scatter-as-gather depth of field (bottom) applied to the first test scene.

Fig. 4.3. Simple depth of field (top left), sprite based depth of field (top right) and scatter-as-gather depth of field (bottom) applied to the second test scene.

scatter-as-gather depth of field. Although that was expected, it is remarkable how close scatter-as-gather depth of field is to the simple technique. Comparing the timings of the two algorithms shows that the blur pass of scatter-as-gather depth of field is considerably and consistently faster than the simple depth of field blur pass. This performance advantage can be explained by the fact that scatter-as-gather depth of field leverages its tile texture to dynamically skip the blur, saving memory bandwidth and time. Performance seems to be largely independent of the circle of confusion radius, provided that it is clamped to a sensible maximum. This also holds true for the simple depth of field technique. Contrary, sprite based depth of field heavily depends on the circle of confusion radius and thus on scene complexity, making it difficult to anticipate the performance impact. This becomes especially apparent when comparing the timings of table 4.3 and table 4.5. Furthermore, sprite-based depth of field runs slower than the other techniques. Its performance is dominated by the sprite render cost. Performance deteriorates with increasing circle of confusion, presumably due to the massive overdraw.

	Simple	Sprite Based	Scatter-as-Gather
CoC Compute	0.0903024	0.090318	0.090318
CoC Blur	0.0701862	n/a	n/a
CoC Tile Max	n/a	n/a	0.0812329
CoC Neighborhood Tile Max	n/a	n/a	0.00690655
Color Downsample	n/a	n/a	0.115736
Blur	0.944527	3.72753	0.784767
Fill	n/a	n/a	0.170156
Composite	0.1615	0.227597	0.290051
Total	1.45175	4.03517	1.53126

Table 4.3. Depth of field timings of the first test scene on the desktop test system. Time is measured in milliseconds.

	Simple	Sprite Based	Scatter-as-Gather
CoC Compute	0.201104	0.201014	0.200751
CoC Blur	0.191823	n/a	n/a
CoC Tile Max	n/a	n/a	0.137517
CoC Neighborhood Tile Max	n/a	n/a	0.00724128
Color Downsample	n/a	n/a	0.319107
Blur	1.1224	8.4519	1.0932
Fill	n/a	n/a	0.232149
Composite	0.552918	0.521314	0.574464
Total	2.32298	9.16142	2.554464

Table 4.4. Depth of field timings of the first test scene on the laptop test system. Time is measured in milliseconds.

	Simple	Sprite Based	Scatter-as-Gather
CoC Compute	0.0901254	0.0903425	0.0903269
CoC Blur	0.0701788	n/a	n/a
CoC Tile Max	n/a	n/a	0.0812195
CoC Neighborhood Tile Max	n/a	n/a	0.00691027
Color Downsample	n/a	n/a	0.115768
Blur	0.938762	1.66884	0.767834
Fill	n/a	n/a	0.166112
Composite	0.161367	0.233008	0.289897
Total	1.4442	1.97278	1.5105

Table 4.5. Depth of field timings of the second test scene on the desktop test system. Time is measured in milliseconds.

	Simple	Sprite Based	Scatter-as-Gather
CoC Compute	0.20265	0.203267	0.203552
CoC Blur	0.191927	n/a	n/a
CoC Tile Max	n/a	n/a	0.139423
CoC Neighborhood Tile Max	n/a	n/a	0.00726528
Color Downsample	n/a	n/a	0.323774
Blur	1.12628	2.0308	1.04904
Fill	n/a	n/a	0.23133
Composite	0.557372	0.486538	0.57232
Total	2.33176	2.67706	2.51695

Table 4.6. Depth of field timings of the second test scene on the laptop test system. Time is measured in milliseconds.

Overall, the scatter-as-gather technique seems to be the most viable solution to fast, consistent and good looking depth of field. Its marginally worse performance compared to simple depth of field is negligible in light of its high visual quality. Sprite based depth of field might be useful when aiming for extremely high quality on good hardware. Another unique aspect of sprite based depth of field is the ability to use arbitrary textures for the bokeh shape, allowing interesting artistic effects. The simple depth of field technique is least suited to be used, even for weak hardware, as its minimal performance advantage does not outweigh its comparatively bad quality. It should be noted that individual parts of the techniques leave room to optimize for performance or visual quality. For instance, the simple depth of field technique could be enhanced with an additional fill pass similar to the one used in the scatter-as-gather approach. Additionally, changes to the sampling kernel and sample weighting functions can have a drastic effect on performance and visual fidelity.

Generally, scatter-as-gather techniques seem to be a promising approach to the scatter problem, both in depth of field and motion blur. They provide a higher vi-

sual quality and potentially save performance when computations can be skipped dynamically.

4.3 Screen Space Ambient Occlusion

Figure 4.4 and figure 4.5 illustrate the four different implemented screen space ambient occlusion techniques applied on the two test scenes. Since GTAO is by design as close as possible to a ray traced ground truth and because no actual ray traced solution is available, GTAO will be treated as the ground truth for the purpose of this comparison. The original SSAO features only very small scale occlusion, leaving the majority of the image at a default occlusion value of 0.5. Although this is due to kernel scale, which can be tweaked, restricting the algorithm to 16 samples does not allow for a much larger scale. Modern hardware can handle more samples easier then hardware from 2007 when SSAO was developed. Other unique aspects of SSAO are its edge highlighting and independence of view space depth, causing the same effect on both close by and far away geometry. These details seem to be considered undesirable, as is evident by their absence in other techniques. A more modern approach to the original technique is the modified SSAO technique featuring a higher sample count and a hemispherical view space kernel. The original SSAO was included into this comparison not as a serious competitor but for historical reasons and to illustrate the progress screen space ambient occlusion has made since then. Compared to the original algorithm, modern SSAO looks much closer to GTAO. However, in some areas it does not darken the image enough. This is especially visible under the car and behind the plant pots seen in the first test scene. HBAO looks even more similar to GTAO. This is presumably because they are both horizon based techniques. Although HBAO features occlusion of a larger scale than modern SSAO, it also does not darken certain areas enough. In the second test scene it casts occlusion too far away, resulting in unpleasant looking shadowing around the teapots. In practice HBAO can be artificially darkened to create the desired appearance. Decreasing the radius is most likely also both beneficial for performance and small scale ambient occlusion. GTAO looks almost always pleasing, independent of scene complexity. The only problematic situation observed while implementing the technique is thin geometry such as the plant leaves on the wall in the first test scene. GTAO darkens these thin features too much. The original authors of the technique suggest a heuristic to attenuate horizons for thin features, reducing the darkening effect [JWPJ16a]. Apart from this minor inaccuracy, quality might decrease around rapidly moving objects as the temporal filter rejects previous frames due to invalidation. Fortunately, most of the time this is not noticeable due to the rapid movement itself.
Measurements of the performance of all algorithms for both desktop and laptop are shown in table 4.7 and table 4.8 for the first scene. Table 4.9 and table 4.10 show the results for the second scene. The results show that the original SSAO algorithm is independent of scene complexity and runs in a consistent short amount of time. This can be explained with the kernel being applied purely in screen space.

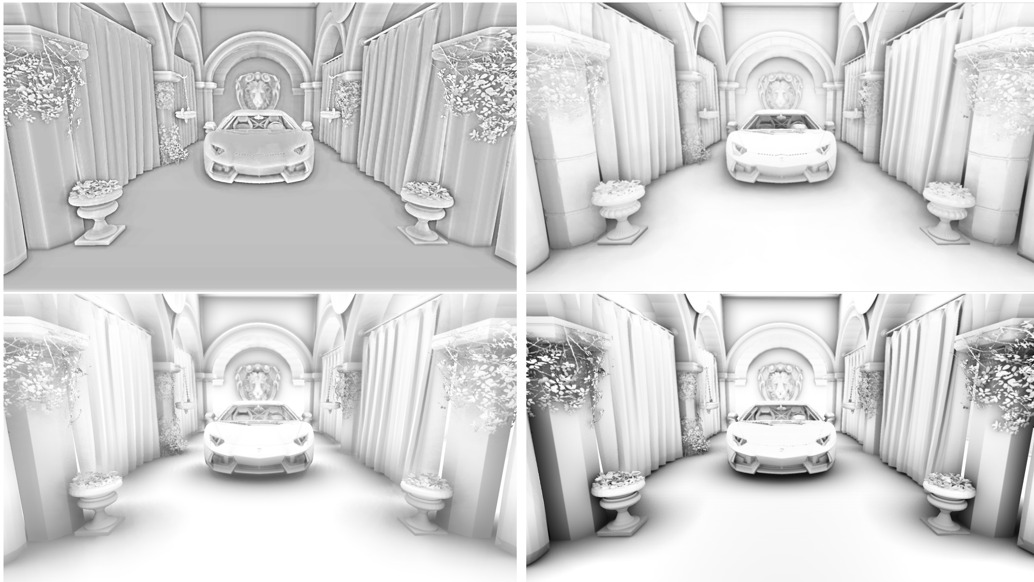

Fig. 4.4. SSAO (top left), SSAO (Hemisphere) (top right), HBAO (bottom left) and GTAO (bottom right) applied to the first test scene.

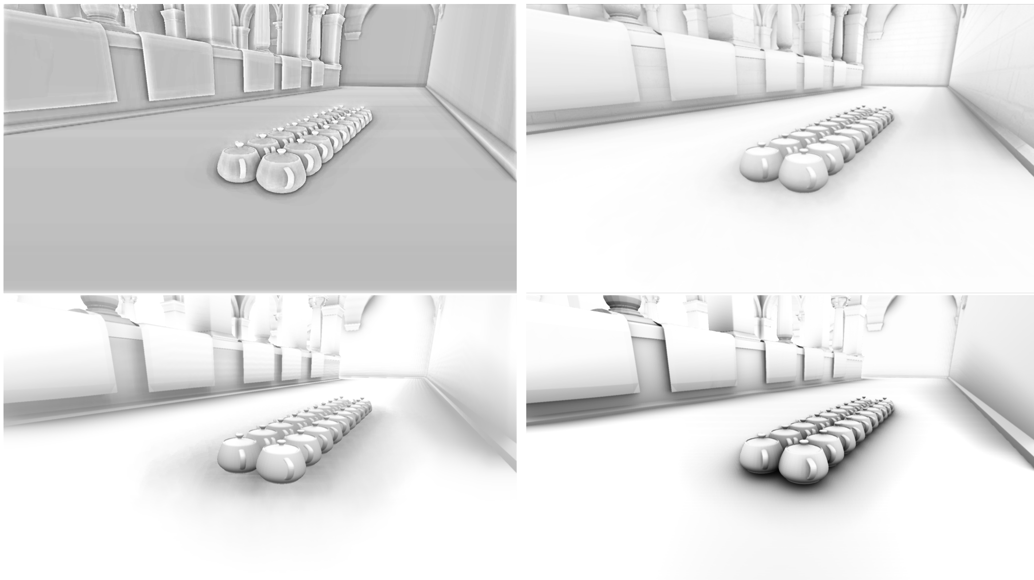

Fig. 4.5. SSAO (top left), SSAO (Hemisphere) (top right), HBAO (bottom left) and GTAO (bottom right) applied to the second test scene.

All other techniques scale their kernel to be constant in view space, making performance dependent on the scene. Since the camera in the second scene is closer to the geometry, the techniques were expected to exhibit worse performance then in the first test scene. Surprisingly this was not the case. GTAO remains consistent, modern SSAO performs much worse, while HBAO performs slightly better. GTAO running in a similar time could be due to its heavy reliance on temporal filtering, taking only few samples each frame and thus making it less memory dependent then the other algorithms. Modern SSAO behaves as expected, leaving not much to interpret. Contrary to modern SSAO, HBAO behaves unexpected, warranting further investigation in the future.
Overall, all screen space ambient occlusion techniques are bandwidth limited. Using one of the optimization techniques discussed in 3.3.6 is likely to improve performance considerably.

	SSAO	SSAO (Hemisphere)	HBAO	GTAO
Ambient Occlusion Compute	0.77788	5.42482	6.83595	2.01105
Spatial Denoise	0.287762	0.287831	0.288115	0.290643
Temporal Denoise	n/a	n/a	n/a	0.131394
Total	1.06032	5.6725	7.08665	2.42091

Table 4.7. Screen space ambient occlusion timings of the first test scene on the desktop test system. Time is measured in milliseconds.

	SSAO	SSAO (Hemisphere)	HBAO	GTAO
Ambient Occlusion Compute	2.18063	18.3955	18.571	6.53066
Spatial Denoise	0.490155	0.489214	0.488914	0.491941
Temporal Denoise	n/a	n/a	n/a	0.238019
Total	2.65761	18.7552	18.9623	7.2247

Table 4.8. Screen space ambient occlusion timings of the first test scene on the laptop test system. Time is measured in milliseconds.

	SSAO	SSAO (Hemisphere)	HBAO	GTAO
Ambient Occlusion Compute	0.776153	9.71725	5.61758	2.05497
Spatial Denoise	0.287617	0.287814	0.287734	0.290846
Temporal Denoise	n/a	n/a	n/a	0.131391
Total	1.05846	9.9764	5.8698	2.4649

Table 4.9. Screen space ambient occlusion timings of the second test scene on the desktop test system. Time is measured in milliseconds.

Since screen space ambient occlusion techniques require many samples to produce accurate results, the modern approach seems to be to employ temporal filtering

	SSAO	SSAO (Hemisphere)	HBAO	GTAO
Ambient Occlusion Compute	2.07547	25.3305	11.7404	5.69297
Spatial Denoise	0.488034	0.48694	0.486443	0.490677
Temporal Denoise	n/a	n/a	n/a	0.229724
Total	2.55017	25.7228	12.1319	6.38156

Table 4.10. Screen space ambient occlusion timings of the second test scene on the laptop test system. Time is measured in milliseconds.

to spread the computation above multiple frames and exploit temporal coherency. This is why GTAO is (except for the original SSAO) the fastest and most stable algorithm. When choosing a screen space ambient occlusion solution, GTAO seems to be the obvious choice. It is not only relatively efficient but also produces accurate results. Generally, horizon-based approaches and temporal filtering seem to be the most promising techniques. The downside to GTAO and other temporal SSAO techniques is that the temporal filter is difficult to implement. Dynamic objects leaving trails is a commonly found artifact.

5

Conclusion

5.1 Conclusion

Having implemented a variety of different motion blur, depth of field and screen space ambient occlusion techniques, a comparison with respect to visual quality, performance and ease of implementation was made. It was shown that modern approaches to these problems eschew approximations and strive to come as close as possible to the ground truth. In particular, motion blur and depth of field profit from scatter-as-gather approaches correctly determining the scattering range and accurately weighting the contribution of other pixels. Nonetheless, some older techniques are still viable for weak hardware. Modern techniques do not simply profit off of faster hardware. Instead, new strategies have been developed, exploiting spatial and temporal coherence to improve performance. This has been demonstrated with screen space ambient occlusion, where a novel technique (GTAO) produces remarkably accurate results in a fraction of the time of previous established techniques, like HBAO. In conclusion, both visual quality and speed of image-based post-processing effects are improving rapidly.

5.2 Future Work

In future work, the possibility of exploiting temporal coherence in other effects might prove important. In addition, reevaluating the feasibility of accurate solutions for real-time rendering applications is an interesting topic for future research.

References

BA12. Bavoil, Louis and Johan Andersson: *Stable SSAO in Battlefield 3 with Selective Temporal Filtering*, 2012.

BJ13. Bavoil, Louis and Jon Jansen: *Particle Shadows & Cache-Efficient Post-Processing*, 2013.

BS08. Bavoil, Louis and Miguel Sainz: *Image-Space Horizon-Based Ambient Occlusion Siggraph 2008*, 2008.

BS09. Bavoil, Louis and Miguel Sainz: *Multi-Layer Dual-Resolution Screen-Space Ambient Occlusion*, 2009.

BSD08. Bavoil, Louis, Miguel Sainz and Rouslan Dimitrov: *Image-Space Horizon-Based Ambient Occlusion*, 2008.

Dem07. Demers, Joe: *Depth of Field: A Survey of Techniques.* In Fernando, Randima (editor): *Programming techniques, tips, and tricks for real-time graphics*, GPU gems. Addison-Wesley, Boston, Mass., 2007.

GMN13. Guertin, Jean-Philippe, Morgan McGuire and Derek Nowrouzezahrai: *A Fast and Stable Feature-Aware Motion Blur Filter*, 2013.

Gre03. Green, Simon: *Stupid OpenGL Shader Tricks*, 2003.

Ham08. Hammon, Earl, Jr.: *Practical Post-Process Depth of Field.* In Nguyen, Hubert (editor): *GPU gems 3*, Safari Books Online. Addison-Wesley, Upper Saddle River, N.J, 2008.

HL11. Hoang, Thai-Duong and Kok-Lim Low: *Multi-Resolution Screen-Space Ambient Occlusion*, 2011.

JESG12. Jimenez, Jorge, Jose Echevarria, Tiago Sousa and Diego Gutierrez: *SMAA: Enhanced Subpixel Morphological Antialiasing.* Eurographics 2012, 2012.

Jim14. Jimenez, Jorge: *Next Generation Post Processing in Call of Duty Advanced Warfare*, 2014.

Jim16. Jimenez, Jorge: *Filmic SMAA*, 2016.

JWPJ16a. Jimenez, Jorge, Xian-Chun Wu, Angelo Pesce and Adrian Jarabo: *Practical Realtime Strategies for Accurate Indirect Occlusion*, 2016.

JWPJ16b. Jimenez, Jorge, Xianchun Wu, Angelo Pesce and Adrian Jarabo: *Practical Realtime Strategies for Accurate Indirect Occlusion Siggraph 2016*, 2016.

Kaj09. Kajalin, Vladimir: *Screen-Space Ambient Occlusion.* In Engel, Wolfgang (editor): *ShaderX 7*, ShaderX series. Course Technology, Boston, Mass., 2009.

Kar14. Karis, Brian: *High Quality Temporal Supersampling*, 2014.

KSS11. Kasyan, Nickolay, Nicolas Schulz and Tiago Sousa: *Secrets of CryENGINE 3 Graphics Technology*, 2011.

MD11. Mittring, Martin and Bryan Dudash: *The Technology Behind the DirectX 11 Unreal Engine "Samaritan" Demo*, 2011.

MHBO12. McGuire, Morgan, Padraic Hennessy, Michael Bukowski and Brian Osman: *A Reconstruction Filter for Plausible Motion Blur*, 2012.

MML12. McGuire, Morgan, Michael Mara and David Luebke: *Scalable Ambient Obscurance.* 2012.

MRD12. McIntosh, Lorne, Bernhard Riecke and Steve DiPaola: *Efficiently Simulating the Bokeh of Polygonal Apertures in a Post-Process Depth of Field Shader*, 2012.

MSW10. Mattausch, Oliver, Daniel Scherzer and Michael Wimmer: *High–Quality Screen–Space Ambient Occlusion using Temporal Coherence*, 2010.

SC97. Shirley, Peter and Kenneth Chiu: *A Low Distortion Map Between Disk and Square*, 1997.

Sch04. Scheuermann, Thorsten: *Advanced Depth of Field*, 2004.

Sou08. Sousa, Tiago: *Crysis Next Gen Effects*, 2008.

Sou13. Sousa, Tiago: *CryENGINE 3 Graphics Gems*, 2013.

Tim13a. Timonen, Ville: *Line-Sweep Ambient Obscurance*, 2013.

Tim13b. Timonen, Ville: *Screen-Space Far-Field Ambient Obscurance.* Proceedings of the 5th High-Performance Graphics Conference, pages 33–43, 2013.

Val13. Valient, Michal: *Killzone Shadow Fall Demo Post Mortem*, 2013.

Vla08. Vlachos, Alex: *Post Processing in The Orange Box*, 2008.

A

Erklärung der Kandidatin / des Kandidaten

☐ Die Arbeit habe ich selbstständig verfasst und keine anderen als die angegebenen Quellen- und Hilfsmittel verwendet.

☐ Die Arbeit wurde als Gruppenarbeit angefertigt. Meine eigene Leistung ist ...

Diesen Teil habe ich selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Namen der Mitverfasser: ...

Datum

Unterschrift der Kandidatin / des Kandidaten