Department of Computer Science,
Faculty of Engineering, LTH

Screen Space Ambient Occlusion & Colour Bleeding

Project Report in High Performance Computer Graphics, EDAN35

Christian Colliander & Elin Törnquist

December 2015

Abstract

This report describes the implementations of Screen Space Ambient Occlusion and Colour Bleeding intended to improve photorealism in a graphics model of the Sponza atrium.

Algorithms as described by John Chapman in [1] and learnopengl.com in [4] were implemented to add Screen Space Ambient Occlusion and algorithms as described by Ritschel et al. in [5] were implemented to add Colour Bleeding. Everything was done in the Bonobo framework in Visual Studio. All additions to the original code provided via the home page of the course (found at [3]) has been tagged to enable easy overview.

Introduction

In this project **screen space ambient occlusion** (**SSAO**) and **colour bleeding** were implemented and used on the scene shown in Figure 1. SSAO lends increased photorealism to a scene by adding *proximity shadows*, i.e. shadows created when nearby geometries occlude scattered (ambient) light. This was done by implementing an algorithm that approximates the amount of ambient light that reaches a given point in the scene, based on nearby geometry. Colour bleeding increases the photorealism of a scene by adding colour to the shadows based on the colour of the geometry causing the occlusion. As is made obvious by comparing the versions of the Sponza detail in Figures 7-14 the SSAO and colour bleeding has a subtle but noticeable impact on making a scene look realistic.

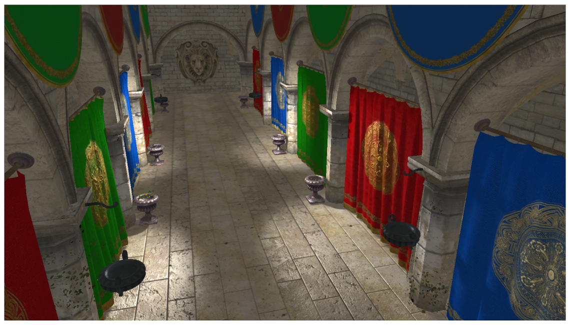

Figure 1: The Crytek Sponza atrium scene rendered using deferred shading in the Bonobo framework, in the second laboratory exercise of the course (see the pdf at [3]).

SSAO

In this project a variant of the **normal oriented hemisphere method**, described in [1, 4], has been used for implementing SSAO. The method is a post-processing addition to a deferred rendering pipeline. It uses per-fragment depth information to recreate the fragment's position in *world space* and checks if there is any nearby geometry which occludes the sample. Except for some precalculations done on the CPU the whole process can be performed on the GPU using data stored in frame buffers and textures, which makes the method work in realtime and independent of the scene's complexity.

After the fragment-to-be-tested has been projected to world space nearby points in world space are selected using a set of points within a hemisphere, oriented according the fragment's world space normal and scaled depending on the scale of the scene (see Figure 2). These sample points are then projected back to screen space and the depth values in their corresponding fragments are compared to the depth of the projected points. If the depth of a sample point's corresponding fragment, i.e. the corresponding geometry, is less than the depth of the projected point, then the sample point is behind some geometry occluding the original fragment. The greater the number of sample points with larger depth values than their corresponding fragments' depths, the more nearby geometry occludes the original fragment. The amount of occluded samples is then finally converted into a value between 0 and 1 that can be used to scale the amount of ambient light that reaches the tested fragment.

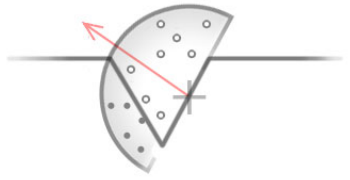

Figure 2: Hemisphere kernel, oriented with the fragment normal, with sample points within the hemisphere [1].

If, instead of a hemisphere, a sphere had been used about 50% of the sample points corresponding to fragments at flat surfaces would be located inside the surface, contributing to occluding the fragment and hence rendering the surface darker than what would be expected. A spherical kernel would also make convex corners lighter than what would be expected as fewer sample points would be inside the geometry. A hemispherical kernel hence creates a more photorealistic SSAO effect as doesn't generate any unnecessary and unrealistic sample points behind the tested point.

The amount of sample points for each tested fragment can be tweaked to either increase performance or quality - more sample points give better results but requires more computation.

For performance optimization the number of sample points needs to be low but this can lead to banding artifacts. To combat this problem the sample kernel is rotated at each pixel to (semi-)randomize the position of the sample points. This leads to less banding but instead introduces high frequency noise which is removed through blurring in the final pass.

To implement the SSAO algorithm the following data is needed for each fragment[4]:

- A per-fragment **position** vector
- A per-fragment **normal** vector
- A linear **depth texture**
- A **sample kernel**
- A per-fragment **random rotation** vector used to rotate the sample kernel

Figure 3 shows the implementation steps, from sampling the G-buffer in order to attain the per-fragment position and normal data to the lighting pass of the rendering.

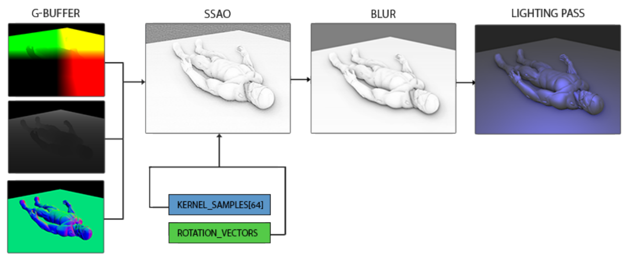

Figure 3: Overview of the steps needed to implement the SSAO algorithm [4].

Colour Bleeding

As with SSAO, colour bleeding, as per the method described in [5], is implemented in screen space and is hence both independent of the complexity of the scene and easy to implement in deferred rendering alongside SSAO as most of the needed parameters are already at hand.

The method described in [5] uses the same hemisphere and sample points that are used for the regular SSAO. Corresponding screen-space fragments are identified for each sample point in the same way as for SSAO, but instead of only looking at depth values also fragment normals and colours are considered when modeling how nearby surfaces affect the light reaching the original fragment. If a sampled fragment has a normal that is oriented towards the original fragment's surface (see Figure 4) light hitting the sampled fragment will reflect ("bleed") onto the other fragment, affecting its final colour. This indirect lighting is often very subtle but can help make a scene more believable and realistic, especially when shiny materials, like marble, are involved.

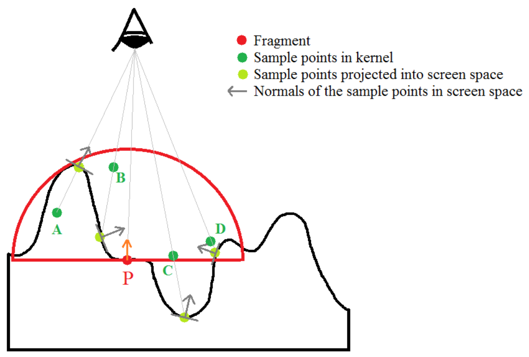
● Fragment
● Sample points in kernel
● Sample points projected into screen space
← Normals of the sample points in screen space
A
B
C
D
P

Figure 4: Original fragment (red) with sample points (dark green), projected sample positions (light green) and their corresponding normal vectors. The normal vectors of sample fragments B and D are oriented towards the original fragment's surface and hence the light hitting those sample fragments will bleed onto the original fragment.

Algorithms

The SSAO and colour bleed was implemented in the Bonobo framework, using the program from Assignment 2 as a basis. The same scene (i.e. the updated Crytek Sponza) was used and the general deferred shading pipeline was also kept, with SSAO + screen space colour bleed added at the end as a post-processing effect. Below follows, in broad strokes, all the changes and additions to the code that was necessary in order to get SSAO and colour bleed to work. First. two additional frame buffer objects were set up:

- The *SSAOFbo* buffer, to which a screen-size texture was attached that was used for storing the colour bleed in the rgb channels and the occlusion factor in the alpha channel.
- The *allLightFbo*, to which a screen-size texture was attached that was used for storing all diffuse and specular light from the light sources in the scene. This was essentially the same image that the deferredResolveShader output to the screen in the final pass in the Assignment 2 program (minus the ambient light).

Two textures were also set up and the values stored in their texels were precomputed in the main program, before the main loop:

- The *rtNoiseTexture*, a 2D texture with 4x4 texels, for storing the noise used for rotating the sample kernel on a per-fragment-basis.
- The *rtKernelTexture*, a 1D texture with the same width, in texels, as the number of sample points in the kernel, used for storing those sample points.

Both these textures were set to "nearest" texture filtering; interpolation between the texels was not desired, only the exact values in each texel was of interest.

The noise texture's wrapping mode was set to "repeat" so that it could be tiled across the entire screen. For the tiling to work correctly a *noise scale* was also set up as $\frac{Screenresolution}{Noisetextureresolution}$ and stored in a 2D vector (containing the scale both in the x- and the y-direction).

To be able to convert differences in depth in the depth buffer into differences in distance in world space, and vice versa, a *depth span* was set up as $(farplane - nearplane)$. This value was the distance in world space that was mapped onto the [0, 1] scale in the depth buffer.

The Kernel Texture

Each texel in the kernel texture was generated from a 3D vector with uniform random values on [-1, 1] in the x- and y- dimensions and uniform random values on [0.05, 1] in the z-dimension. The minimum value in the z-dimension was set to avoid samples near or in the surface containing the fragment as this could cause artifacts due to the quantized depth buffer [2].

Normalizing this vector yielded a vector pointing to some point on the surface of a unit hemisphere oriented along the z-axis. A quadratic function was used to bias the placement of the sample points towards the center of the kernel. This biasing would have the effect of indirectly making geometry close to the origin of the kernel have a greater impact on occlusion, since (generally) more sample points would be there to detect that geometry, compared to geometry further away from the kernel's origin where there are fewer sample points. Finally, the sample vectors were stored as RGB-values in the kernel texture.

The Noise Texture

Each texel in the noise texture was generated from a 3D vector with uniform random numbers (on [-1, 1]) as the x- and y-components (corresponding to the tangent and bitangent in normal space) and 0 as the z-component (corresponding to the normal). This vector was then normalized (thus mapped to the edge of the unit circle) and finally stored as RGB-values in the noise texture.

Once all these things had been set up the main loop was started and proceeded just as in assignment 2 until all the direct light sources (i.e. diffuse and specular light) had been rendered (i.e. pass 2), after which the direct light was accumulated in the buffer *allLightFbo*. The data in that buffer was then used both in the *SSAO* and in the *resolve_deferred* shaders.

SSAO Pass

In addition to some uniform projection matrices and constants, the following textures/buffers were bound to the *SSAO* shader:

- Depth buffer
- Normal buffer
- *allLightFbo*
- Noise texture
- Kernel texture

Each fragment the SSAO shader runs on is projected into world space. There, a temporary tangent space is set up by using the cross product between the fragment's world space normal and an arbitrary vector as the *tangent* and the cross product between this tangent and the known normal as the *bitangent*. Note that the same vector is used for creating the tangent in all fragments, except when a fragment's normal happens to be parallel to that vector where another "fallback" vector is used. A TBN matrix is created from these 3 vectors and multiplied with the *noise texture vector* corresponding to the source fragment's position in screen space, thus creating a "noise vector" that is a quasi-random linear combination of the tangent and bitangent.

A new tangent space is then constructed by using the Graham-Schmidt process on the fragment's world space normal and the noise vector. The resulting tangent space vectors are stored in a TBN matrix.

The end result of this is a quasi-random rotation of the tangent space on a fragment-by-fragment basis, with the same orientation being repeated every fourth pixel on surfaces with the same world space normal, as seen in Figure 5.

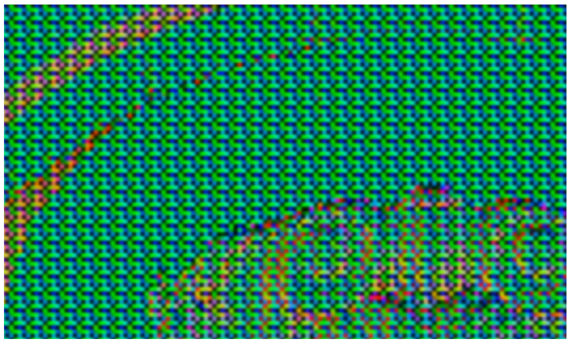

Figure 5: A closeup of the tangent vectors in the scene, presented as RGB values, without the blur filter. Notice that the same tangent direction(/colour) is repeated every fourth pixel on surfaces with the same normal.

Each vector stored in the *kernel texture* is then, in turn, multiplied with the rotated TBN matrix, thus moving the sample points into a tangent space defined in world space coordinates (compare this to how surface normals are permuted in bump mapping). The radius of the kernel hemisphere is then scaled using a scene dependent factor [1], and the kernel is moved to the fragment's position by adding the fragment's world space coordinates.

The result is a set of "sample points" within a hemisphere centered on the fragment's world space position, pointing along the fragment's surface normal.

Each *sample point* is then projected back into screen space and the depth of the corresponding *sample fragment* is extracted from the depth buffer. This depth is then both compared to the depth of the *original fragment* and the depth of the sample point.

The first check (the "range check") is there to prevent occlusion from very large depth discontinuities. Without this check objects in front of distant geometry would appear to have a black aura (see figures about the range check in [1]). If the sample fragment is very far from the original fragment then AO is not applied for that sample point. The acceptable depth difference between the two fragments decreases with the square of the depth of the original fragment - this is discussed further in the "discussion" section.

The second check (the "occlusion check") tests if the sample point's corresponding fragment is in front of the the sample point. If so then the sample point is assumed to be inside geometry.

If both the occlusion check and the range check passes then it is assumed that some geometry is near (and in front of) the original fragment *and* that that geometry is close enough to realistically occlude the original fragment. If both these checks passes then that sample point is counted as an occluder and an *occlusion factor* is increased by 1.

These checks are performed for all sample points in the kernel and the final occlusion factor is normalized to [0, 1] (i.e. divided by the total number of sample points) and inverted, so that many occluders result in a small factor and few occluders result in a large factor. The occlusion factor is stored in the alpha channel of the *SSAOFbo*.

Calculating Colour Bleed

Using information already stored in different buffers and reusing the sample points set up for SSAO it's possible to add colour bleeding as a post processing effect. Colour bleed is calculated in the SSAO shader, after AO has been computed.

For each sample point that contributes to occlusion some extra calculations are performed, the intent of which is to see how much, if any, light should bounce from the (world space) *occluding fragment* onto the original fragment.

The amount of light that bounces is determined using Equation (1), which is a somewhat simplified version of the second equation in [5], where d_i is the distance between the original fragment P and the occluding fragment i, A_f is a factor loosely representing the size of the occluding fragment, Θ_{oc_i} is the angle between the normal of the occluding fragment and the trans-

[1]The size of the hemisphere determines the distance at which occluders will be detected. Different scenes generally require different radii and it's often up to the artist/programmer to determine what value to go with [4].

mittance vector (see Figure 6) and Θ_{or_i} is the angle between the normal of the original fragment and the transmittance vector.

$$L_{dir}(\mathbf{P}) = \sum_{i=1}^{N} L_{pixel} \frac{A_f cos\Theta_{oc_i} cos\theta_{or_i}}{d_i^2} \tag{1}$$

If the occluding fragment is oriented towards the original fragment, light will, to a degree, reflect from it onto the original fragment. Therefore, if the cosine factors meet the demand of being greater than 0, colour is sampled from the *allLightFbo* at the occluding fragment's screen space coordinates, scaled using the result of Equation (1) and stored in the RGB-channels of the *SSAOFbo*. Note this value only was the light transmitted *from* the occluding fragment to the original fragment and that the interaction with the original fragment's surface was calculated later, in the blur shader.

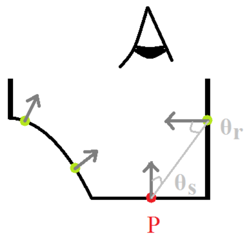

Figure 6: If the original fragment and the occluding fragment are facing each other then light hitting the occluding fragment will bleed onto the original fragment. Whether the two fragments are facing each other is determined by comparing the angles between the respective fragment's normal and a *transmittance vector*, going from one fragment to the other.

Blur Pass

The blur pass required the *SSAOFbo*, the *allLightFbo* and the diffuse texture buffer from pass 1, in addition to the resolution of the noise texture and the inverse of the screen resolution. First, the direct light and diffuse colour for the current fragment was extracted from the *allLightFbo* and the diffuse texture, respectively. Ambient light was calculated from the diffuse colour and a scene dependent ambient light factor. Then, the occlusion factor and *incoming* indirect light (colour bleed) was calculated for the current fragment by sampling the SSAO buffer in a grid with the same size as the noise texture (4x4 by default) and averaging the result. The final ambient light was then simply calculated by multiplying the (averaged) occlusion factor with the ambient light. The final indirect light was calculated by first multiplying the incoming indirect light with the original fragment's diffuse colour (material interaction) and then applying a filter that dampened colour bleed on surfaces with little incoming direct light (this filter prevented unrealistic highlights on dark surfaces). The final colour of the fragment was computed as the sum of the direct (diffuse + specular), the ambient light and the indirect light ("colour bleed").

Results

By comparing Figures 7 and 8 it is noticeable that proximity shadows as well as colour bleeding has been added to the scene. The differences between the different effects are made more obvious in Figures 7-14.

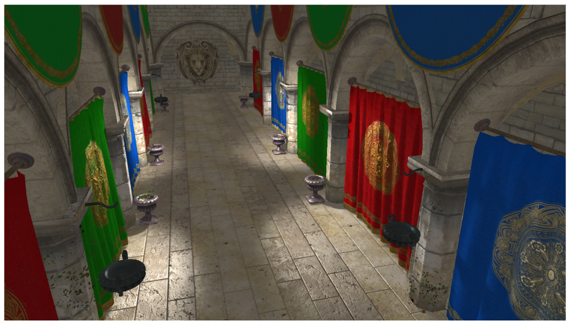

Figure 7: Overview of the original scene, where deferred shading has been used to render the Sponza palace.

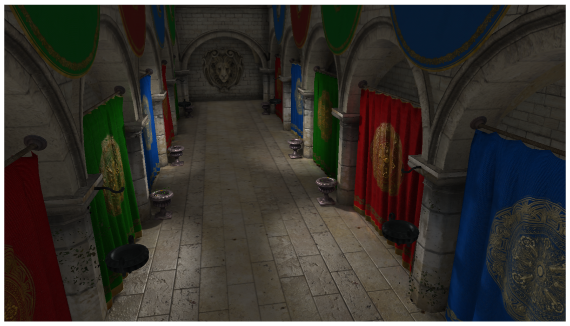

Figure 8: Overview of the Sponza palace scene where SSAO and colour bleeding has been added to the deferred rendering.

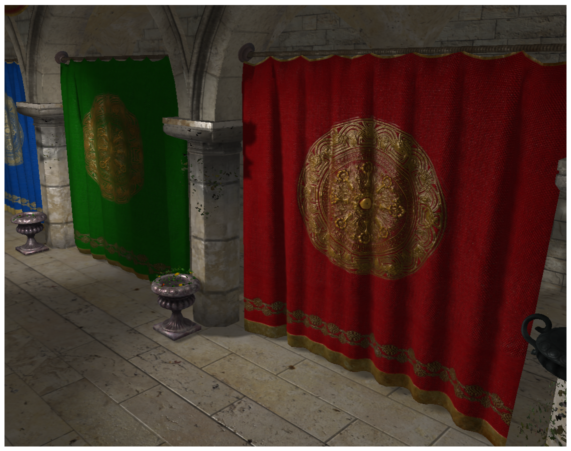

Figure 9: Detail of the Sponza scene, with both SSAO and colour bleeding disabled.

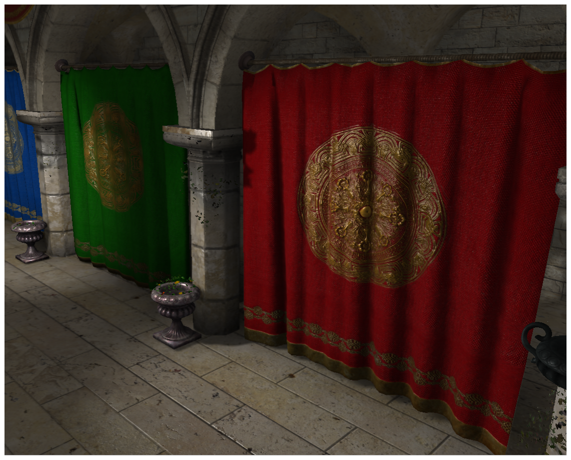

Figure 10: Detail of the Sponza scene, with added SSAO.

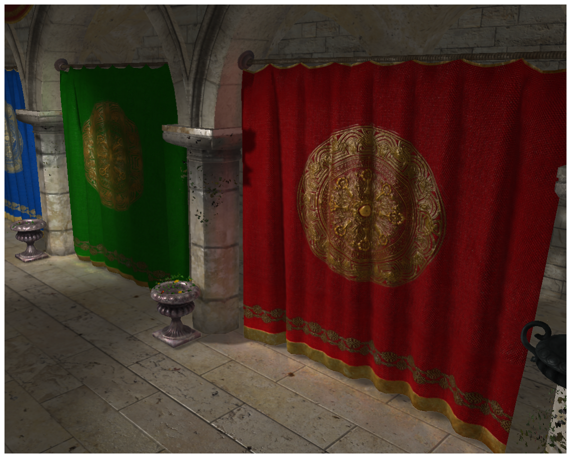

Figure 11: Detail of the Sponza scene, with added colour bleeding.

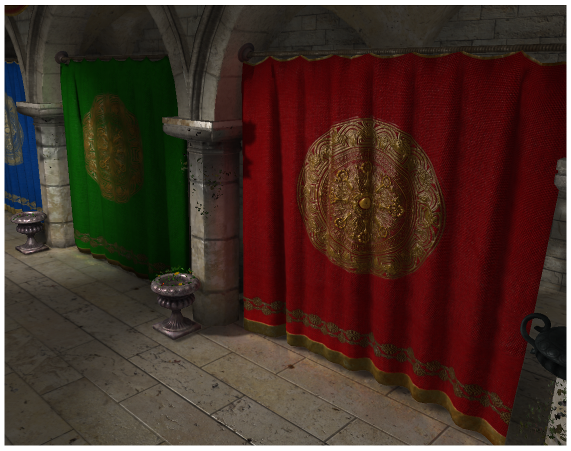

Figure 12: Detail of the Sponza scene, with both SSAO and colour bleeding added.

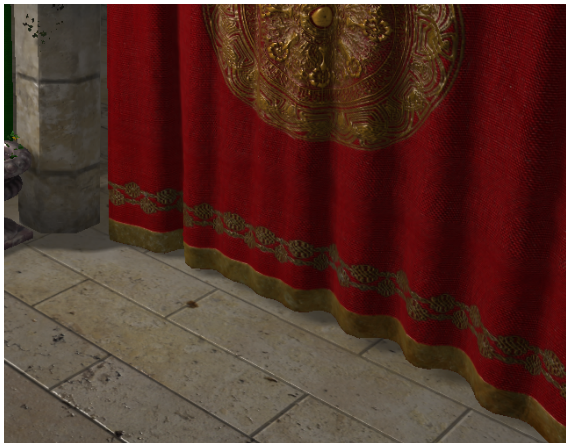

Figure 13: Detail showing the original state, with both SSAO and colour bleed disabled.

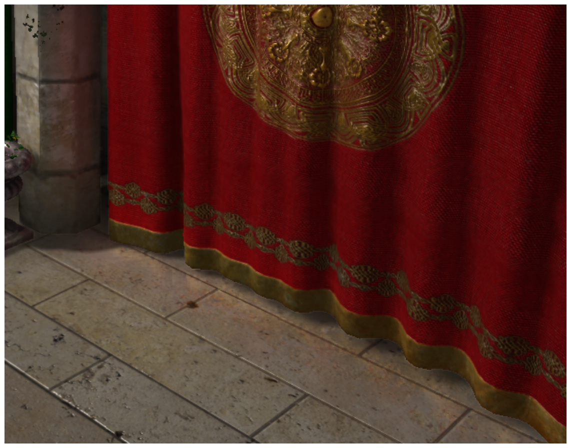

Figure 14: Detail showing the final state, with both SSAO and colour bleed enabled. The SSAO is most visible on the side of the pillar to the left, and colour bleed is most obvious on the floor bellow the tapestry.

Discussion

Much information about how to implement the algorithms for both SSAO and colour bleeding could be found online. Online guides did a good job at outlining the general steps needed to get SSAO up and running, as well as some tips for how to tweak certain parameters. Even so, every step along the way had to be manually adapted to the code from Assignment 2, something which provided ample opportunity to get into the code and really understand what every line of code did.

Finding variables (hemisphere sphere size, occluder fragment size for colour bleed, ambient light strength, kernel size etc.) that worked well required a lot of trial and error. In addition to directly tweaking variables there were lots of other things that could be done in the code to affect the final look of the image, like what kind of function to use for scaling the acceptable depth distance during the *range check*.

We were satisfied with the general implementation of the **AO**, but felt that some of the variables and expressions (like kernel radius, the expression for the range check radius and how much the occlusion factor should be increased for each occluder) could use some more tweaking to improve the look of the AO in the scene.

As an example, no range check caused good looking self shading but unacceptable SSAO auras, a constant radius size caused auras for objects far away and no occlusion for objects close to the camera, a linearly decreasing value made had the same effect as constant values, but to a lesser degree and a quadratic expression almost removed the phenomenon (granted, it still wasn't perfect and there was certainly room left for improvement[2]).

[2]Shortly after handing in the project we found an even better equation for the range check in [4]. This equation was a smoothstep from 0 to 1 with the quotient between the AO radius and the depth difference between the original and sample fragments as the source value.

We were quite satisfied with the **colour bleed** too but think that there was a lot of room left for improving the effect.
For example, sometimes the bleed effect looked a bit jagged/pixelated. Averaging the colour bleed over a larger grid could probably help with that, or even using a more advanced blurring algorithm. That the colour bleed was only calculated in image space could be seen by looking at the pillars near the drapes: colour bled to the parts of the pillars that were in front of the drapes, but the parts of the pillars that were behind the drapes received no colour bleed (because the normal of the visible parts of the drapes were pointing away from those surfaces), as seen in Figure 15. In [5] the authors used a combination of multiple cameras and depth peeling to solve this, a similar solution is possible for our implementation but it is questionable whether the performance cost (especially for multiple cameras) is worth it. Using a smoothstep function to scale the range check radius instead of a quadratic function made the effect less obvious, as seen in Figure 16, so it can be said that that was a partial solution to the problem.

Further, ambient light doesn't cause any colour bleed in our implementation because we don't add ambient light until the final pass. This is tricky to solve because we don't know exactly how much ambient light will be at any given point until we have completed the SSAO pass. We haven't figured out any perfect solution, but a decent solution could be to add a small amount of ambient light before the SSAO pass, calculate SSAO + colour bleed and, in the blur pass, subtract the ambient light that was added from the *allLightFbo* and add new ambient light that is scaled according to the occlusion. This would add some colour bleed from ambient light, but it would not properly scale with occlusion.

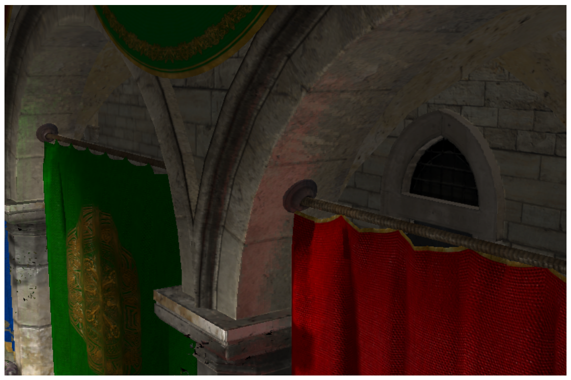

Figure 15: A closeup of colour bleed near the drapes in the scene, with a quadratic function used for the range check. Colour bleed strength was scaled to make it more apparent for the screen capture. Note the sharp cutoff of the bleed for fragments "behind" the drape.

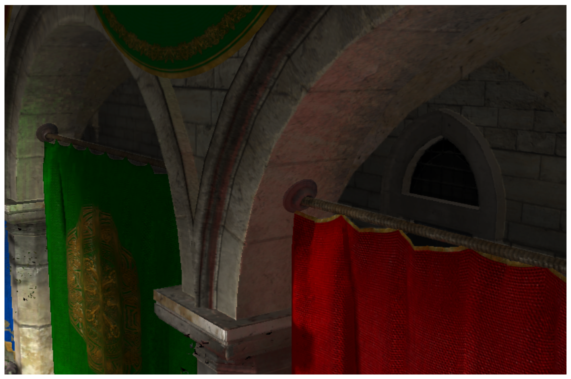

Figure 16: A closeup of colour bleed near the drapes in the scene, with a smoothstep function used for the range check. Colour bleed strength was scaled to make it more apparent for the screen capture. Note the smooth cutoff of the bleed for fragments "behind" the drape.

Performance

Performance is critical within real-time rendering and overhead should be kept at a minimum. When implementing SSAO there are a lot of factors that impact performance and many tweaks that can be done to enhance it, often without compromising visual fidelity to any major extent.

A great example of these kind of optimizations is the use of a pre-calculated sample kernel + noise texture and a blur filter, as opposed to calculating multiple random sample points for each fragment in the SSAO shader. Sampling a couple of times from 2 very small textures (using only "nearest" filtering) and blurring the result is cheaper than generating multiple random values and converting them to points with the desired distribution throughout a hemisphere in each instance of the SSAO shader. While using completely random sample points in each fragment would give a better result (e.g. absolutely no banding artifacts given a reasonable amount of samples), the cost associated with doing so is too high for real-time rendering.

The variable that is the most intuitive "quality-vs-performance" dial is the number of sample points in the kernel: more points give better and more consistent detection of occluders, but also has a negative impact on performance since more sample points will have to be looped through in each fragment. We found that 16 samples was a pretty good "average" value, with half of that (8) introducing noticeable banding and that twice of that (32) providing noticeably better looking results. Anything above 64 provided little, if any, improvement in visual fidelity, making a kernel size of 64 a reasonable upper cutoff because of the diminishing returns thus associated with higher sample counts.

Average ms per frame (over 10 seconds) for different numbers of sample points in the kernel are shown in Table 1. Also shown is the average time per frame with no AA (measured in the program from assignment 2). The results were measured on a laptop with an Nvidia GeForce 460M, an Intel i7-2630QM and 12 GB of RAM.

Sample points	Average ms/frame	% increase
No AA	53.48	0.00
8	68.03	27.21
16	68.03	27.21
32	84.03	57.12
64	149.25	179.08

Table 1: Average time per frame (in milliseconds) measured over 10 seconds and the % increase in time needed per frame from the lowest value for different amounts of sample points in the kernel. Also listed is the same values for the program from Assignment 2 (i.e. no SSAO at all).

Outside of tweaking the number of sample points there are multiple other things that can be done to increase performance, like running the SSAO shader at half the window resolution, which allegedly provides great performance improvements in exchange for a small reduction in visual fidelity, partly because the final result is blurred [6]. This wasn't something that we implemented in our project, but it would probably be the first thing we would implement if we were to take it further.

Measuring how much more expensive SSAO with screen space colour bleed is compared to just SSAO would be interesting, but it would require rewriting a lot of code (mainly cutting out the early light accumulation pass before SSAO and doing that in the blur pass and removing the *allLightFbo*). It's possible to get an idea of the cost by commenting out everything that pertains to colour bleed in the SSAO and blur shaders: using a kernel size of 64 this results in an average time (over 10 seconds) per frame of 117.65 ms, a reduction of 21.17 % compared to the same number of samples with colour bleed. Commenting away those lines of code removed a lot of texture samples (1 from the normal buffer and 0-1 from the direct light buffer per occluder in the SSAO shader) and several lines of calculation (dot products, normalization, projections along with lots of arithmetic calculations).

This implies that screen space color bleed and SSAO combined isn't much more expensive than SSAO alone, at least for our implementation.

References

[1] John Chapman. *SSAO Tutorial.* 2011. URL: `http://john-chapman-graphics.blogspot.se/2013/01/ssao-tutorial.html?m=0` (visited on 12/03/2015).

[2] Phil 'mtnphil' Fortier. *Know your SSAO artifacts.* 2013. URL: `https://mtnphil.wordpress.com/2013/06/26/know-your-ssao-artifacts/` (visited on 12/08/2015).

[3] EDAN35 High Performance Computer Graphics. *Assignment 2 : Deferred Shading and Shadow Maps.* 2015. URL: `http://cs.lth.se/edan35/assignment-2/` (visited on 12/14/2015).

[4] learnopengl.com. *SSAO.* URL: `http://www.learnopengl.com/#!Advanced-Lighting/SSAO` (visited on 12/14/2015).

[5] T. Ritschel, T. Grosch, and H-P. Seidel. "Approximating Dynamic Global Illumination in Image Space". In: *Proceedings of the 2009 symposium on Interactive 3D graphics and games* (2009), pp. 75–82.

[6] Peter Wester. "Tech Feature: SSAO and Temporal Blur". In: (2014). URL: `http://frictionalgames.blogspot.se/2014/01/tech-feature-ssao-and-temporal-blur.html` (visited on 12/14/2015).