SSAO and DOF Implementation in OpenGL

Elmer Dellson* Erik Rolander†

Lund University
Sweden

Abstract

We implemented Screen Space Ambient Occlusion (SSAO) and Depth of Field (DOF) into a small 3D scene using C++ and OpenGL (GLSL) by writing shaders running on screen-covering triangles.

The SSAO effect is achieved by the fragment shader taking samples around each fragment, and checking if they are inside the surrounding geometry or not. This gives us an idea of approximately how much ambient shadows would darken that fragment. The ambient occlusion values of all fragments on screen are rendered to a texture, which is then sampled in the final light-resolve-pass.

The Depth of Field effect is achieved by calculating the circle of confusion for every fragment, and blurring them accordingly. The plane of focus of the scene will always be in the center of the screen, where the camera is looking. Both techniques rely on depth sampling.

The addition of SSAO and DOF into our program resulted in a scene with higher fidelity and more natural lighting. However, the computations required for our implementation proved to be quite heavy, and some further optimization would greatly benefit the performance.

1 Introduction

Screen Space Ambient Occlusion (SSAO) is a method of approximating the natural phenomenon of shadows forming in narrow and tight spaces, such as corners, cracks or behind objects close to a wall [de Vires]. The real world phenomenon is caused by the rays of light statistically having a lower chance of reflecting out of narrow spaces, meaning that less light can reach our eyes from that point. But since this would be too heavy to realistically implement, we instead use the SSAO algorithm to approximate this effect. The algorithm works by taking depth samples from random points within a hemisphere of the current fragment, and determines an occlusion factor based on the ratio between samples with greater depth than the scene and samples with lesser depth than the scene. Using SSAO will give the scene a greater sense of depth and a more realistic-looking lighting.

Depth of Field (DOF) is a natural effect caused by lenses where the the foreground and background (the areas not currently in focus) become blurry [Nvidia]. Since our eyes have adjustable lenses, we would also like to recreate this effect in our shaders, to give the scene a more natural and realistic look. While in reality the effect is caused by the properties of lenses, we will have to artificially recreate the effect trough an algorithm.

2 Algorithms

2.1 Screen Space Ambient Occlusion

The SSAO algorithm is implemented in a separate fragment shader, that runs on a single screen-covering triangle [de Vires]. We begin by letting the main application generate a number (64, in our case) of random tangent-space vectors within a hemisphere of a set radius. This is what we will call the *sample kernel*. We also generate a a small texture (4x4 texels) that contains 16 random vectors in the xy-plane in tangent-space. We call this texture the *noise texture*. In the fragment shader, we use the vectors in the sample kernel, rotated using the vectors in the noise texture, to sample 64 other fragments around each fragment. This is done in view-space. For every one of these samples, we use the depth buffer to determine whether or not the sample is *inside* the surrounding geometry or not (i.e. if there is something between the sample and the camera). See Figure 1.

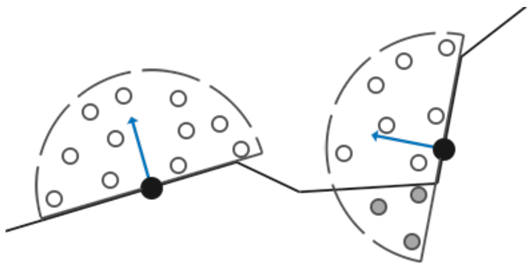


Figure 1: SSAO sampling in a hemisphere [de Vires].

We then divide the number of samples that pass (are not inside geometry) by the total number of samples, and invert this value (take it away from 1.0). This gives us the total ambient occlusion for that fragment, which we save to a buffer. Finally, we blur the values in the buffer to eliminate any visible noise that might appear due to the use of the randomized rotation vectors in the noise texture, and multiply them with the ambient-term of the Phong-shading.

2.2 Depth Of Field

Like the SSAO, the DOF is in a separate shader run on a screen covering triangle. It will blur parts of the screen depending on how far they are from the plane in focus [Nvidia]. The plane in focus is determined by sampling the depth value from the depth buffer at the center of the screen, since that is where the camera is ”looking”. It will then blur the image according to the *circle of confusion*.

The circle of confusion (CoC) is the circular area that the light from a single point will project to on a plane as it passes through the lens. A point in perfect focus will project to a perfect point on the plane, whereas points out of focus will project to a circle, resulting in a blurry image. The further away from the *plane of focus* the point is, the large the diameter of the CoC will be.

To blur areas that are out of focus, we calculate the CoC for each fragment, and then sample the surrounding fragments in a square pattern. The larger the CoC, the more samples we take, meaning that fragments with a large CoC will sample a larger area around itself, and therefore get more blurred than fragments with a small CoC. Finally, we average the color of the samples, and use that as final color for the current fragment. To avoid taking ridiculously large numbers of samples for fragments very far out of focus, we bounded the CoC to a reasonable value (20 in our case).

*e-mail: el0860de-s@student.lu.se
†e-mail: er1748ro-s@student.lu.se

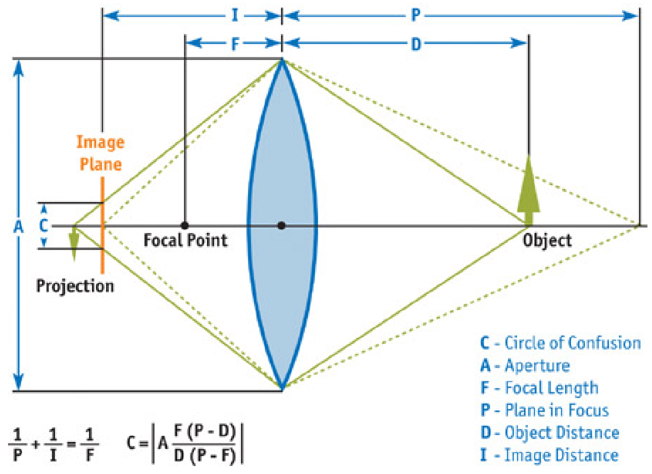


Figure 2: A model of a lens, with the equation for the circle of confusion (C) [Nvidia].

3 Results

The results can be seen in figures 3 - 8 (see appendix). Running on an *Nvidia GTX 970M* GPU in full HD resolution (1920 x 1080 pixels), we generally get around 30 FPS (Frames Per Second) with both SSAO and DOF on. However, the strength of the DOF effect greatly impacts the performance: moving the camera really close to a foreground object so that a lot of the background is blurry, like in figure 4, results in significant framerate dips, sometimes as low as 7 FPS(!). Using more powerful hardware can of course mitigate the issue and give ”playable” framerates even in the more extreme scenarios (our demo video was recorded on an *Nvidia RTX 2070 SUPER* GPU).

4 Discussion

4.1 Evaluation of Results

All in all we are very pleased with the results. The SSAO adds a great sense of depth by (relatively) cheaply simulating a lot of the shadows you would see in the real world, which are sorely needed in a rasterized render. The preferable solution to get these shadows would of course be proper ray-tracing, but this technique is still out of reach for most today (even though some games actually do support it already).

The DOF is a really visually pleasing effect, and can be used to simulate a camera or the human eye depending on the values of the ”virtual camera”'s parameters. Smaller apertures give the scene a more cinematic look, while larger ones add a nice subtle effect. The blur effect could be improved by picking the samples around each fragment in a circular pattern (rather than the square one we are using in this project). This would result in a *bokeh*-effect, where out-of-focus bright-spots (such as point-lights) will look like soft circles, further mimicking a real camera.

4.2 Implementation Issues

The implementation of these effects led to some issues and artefacts. Some of these we managed to solve, while others remain.

4.2.1 Depth Value Comparisons

Our main struggle during implementation was the comparisons of depth values. The project we used as the foundation for this demo (Assignment 2 in the course EDAN35 at LTH) stores the depth buffer values in normalized screen-space. The two techniques implemented in this paper both rely heavily on depth comparisons, but we could not get depth comparisons working if we compared in normalized screen-space. Our naïve solution to this was to always transform all depth values into view-space before comparisons, which leads to a lot of computation-heavy matrix multiplications and is likely the reason for the poor performance presented above. We intended to fix this, but ran out of time. Our proposed solution was to generate a new depth buffer (when filling the geometry buffer) in which the depth values would be stored linearly, allowing easy comparison after just one texture read.

4.2.2 False Occlusion Contribution

When calculating the occlusion factor for a fragment that is close to the edge of an object in the foreground, some of the randomized sample points may end up behind the object and therefore be compared to the depth value of the object. This will incorrectly contribute to the occlusion factor, resulting in shadows along the silhouettes of objects in the foreground. To prevent this we implemented a range check that would disregard samples if the difference in depth values were greater than the radius of the sample kernel hemisphere.

4.2.3 SSAO Sampling Out Of Bounds

This is an artefact where ambient occlusion shadows disappear when close to the edge of the screen. The cause of this is that fragments close to the edge of the screen will try to sample outside of the full-screen depth buffer. This gives us incorrect depth values, which causes fragments that should be occluded to remain fully lit.

4.2.4 Color Bleed

One problem we *did* manage to solve was that of *color bleeding*. This is an artefact that appears when a foreground object is in focus, and makes it look like the color from the edges of the foreground object ”bleed” into the background.

This is caused by the fragments in the background sampling fragments in the foreground, and we solved it with yet another depth comparison: if the distance between a sample fragment and the plane of focus is larger than a set value (we used 20.0), don't add its color to the average for the current fragment. Instead (but rather than just moving on to the next sample) add the color of the fragment *on the opposite side of the current fragment*. This ensures that no color bleeds, but that the level of ”blurriness” is the same all the way up to the edge of the object in focus.

4.2.5 Sharp Foreground Edges

Another artefact appears when the background is in focus, and there is a blurry object visible in the foreground. In this case, the edge of the blurry object will be razor sharp, even though it should be just as blurry as the object itself is. The immediate solution to this would be to have the fragments in the background closest to the edge of the foreground object sample some of its color. Unfortunately we could not figure out how to get this working in time though, so the artefact remains.

5 Conclusion

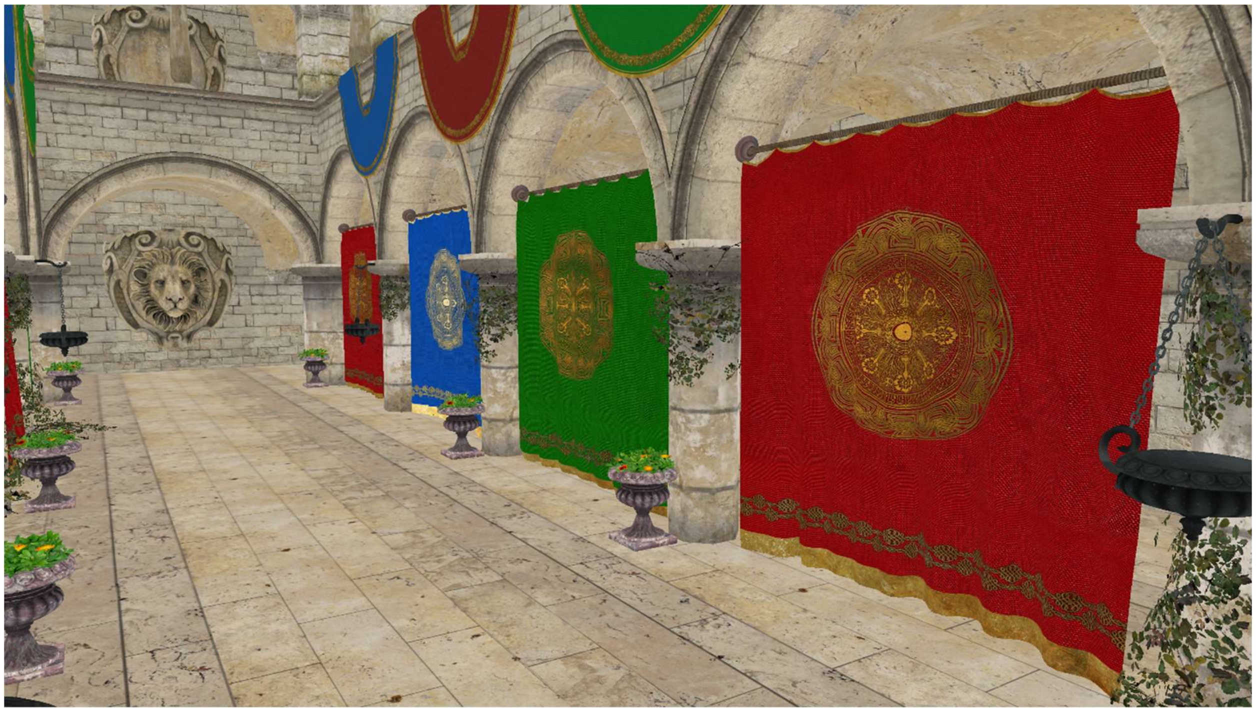
The addition of effects like SSAO and DOF greatly improve the visual fidelity of a scene, even if they are quite subtle. If the hardware can handle it, it is well worth including in any rasterized rendering. However, performance can be greatly impacted if the implementations are not clever, and it might not be worth sacrificing the framerate too much for improved looks. A good balance between looks and performance is desirable, and neat solutions to artefacts and performance issues can often be found.

References

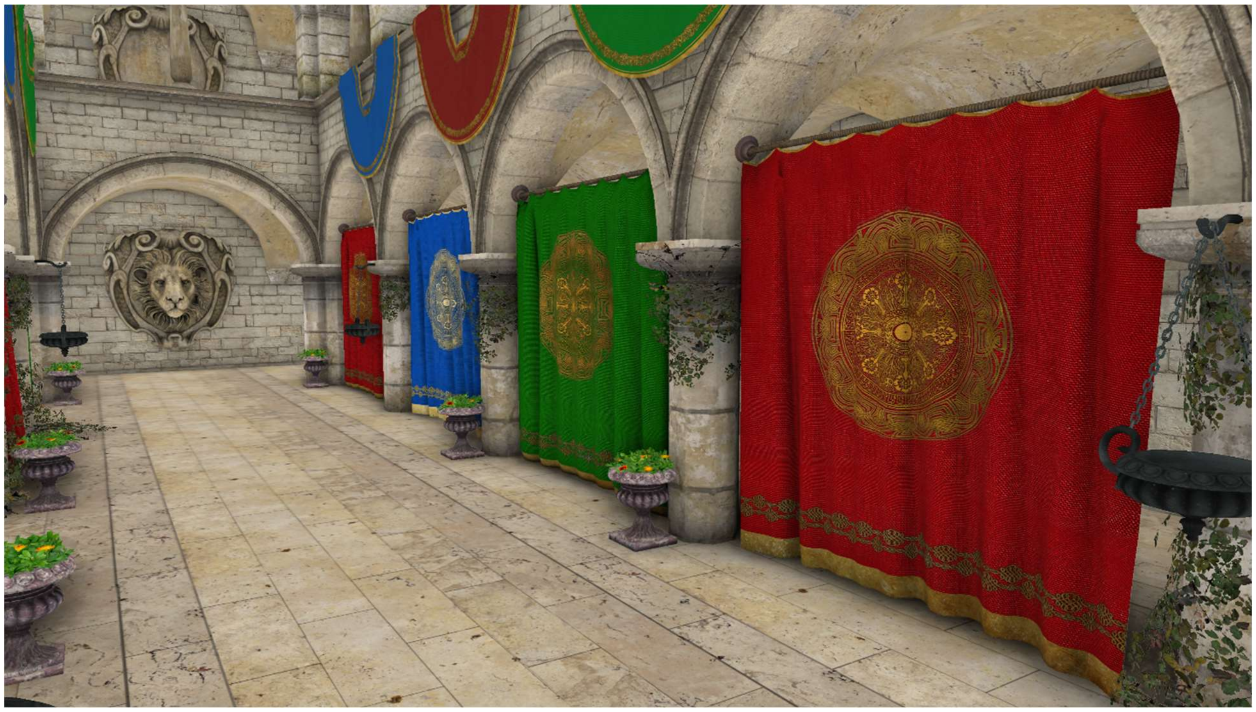
DE VIRES, J. Ssao. `https://learnopengl.com/Advanced-Lighting/SSAO`. Accessed: 2020-12-18.

NVIDIA. Depth of field: A survey of techniques. `https://developer.nvidia.com/gpugems/gpugems/part-iv-image-processing/chapter-23-depth-field-survey-techniques`. Accessed: 2020-12-18.

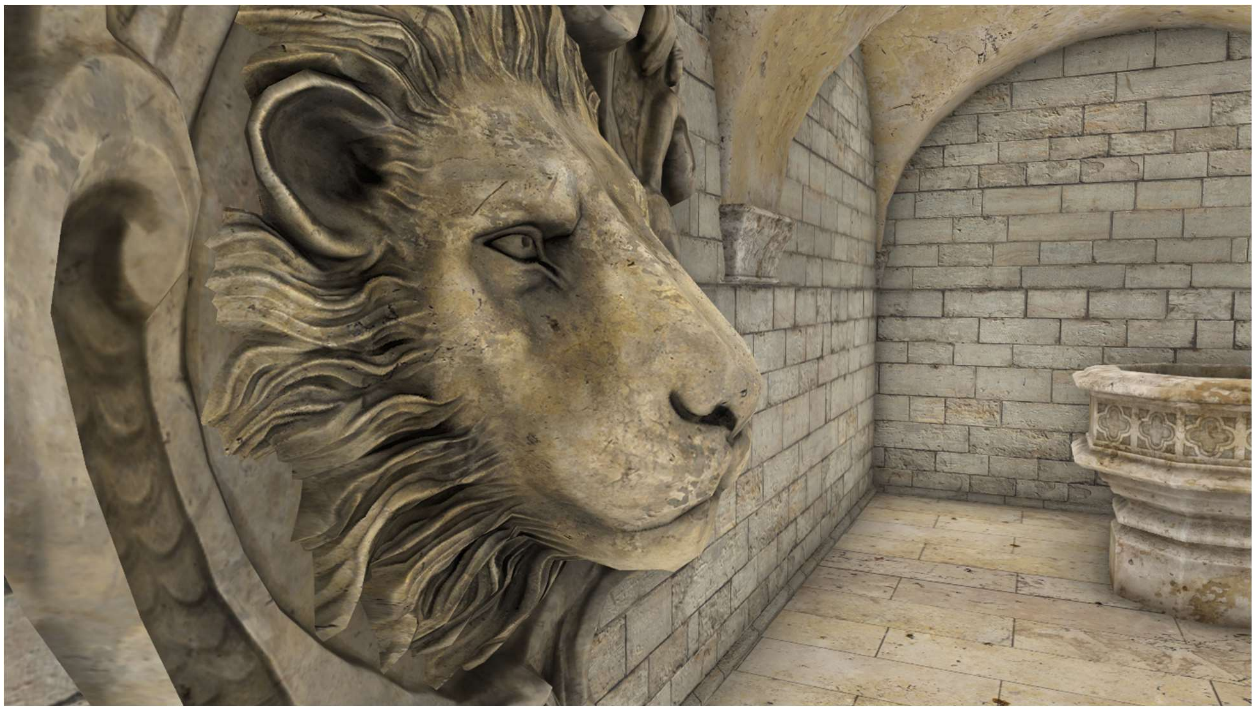
Appendix: Screenshots



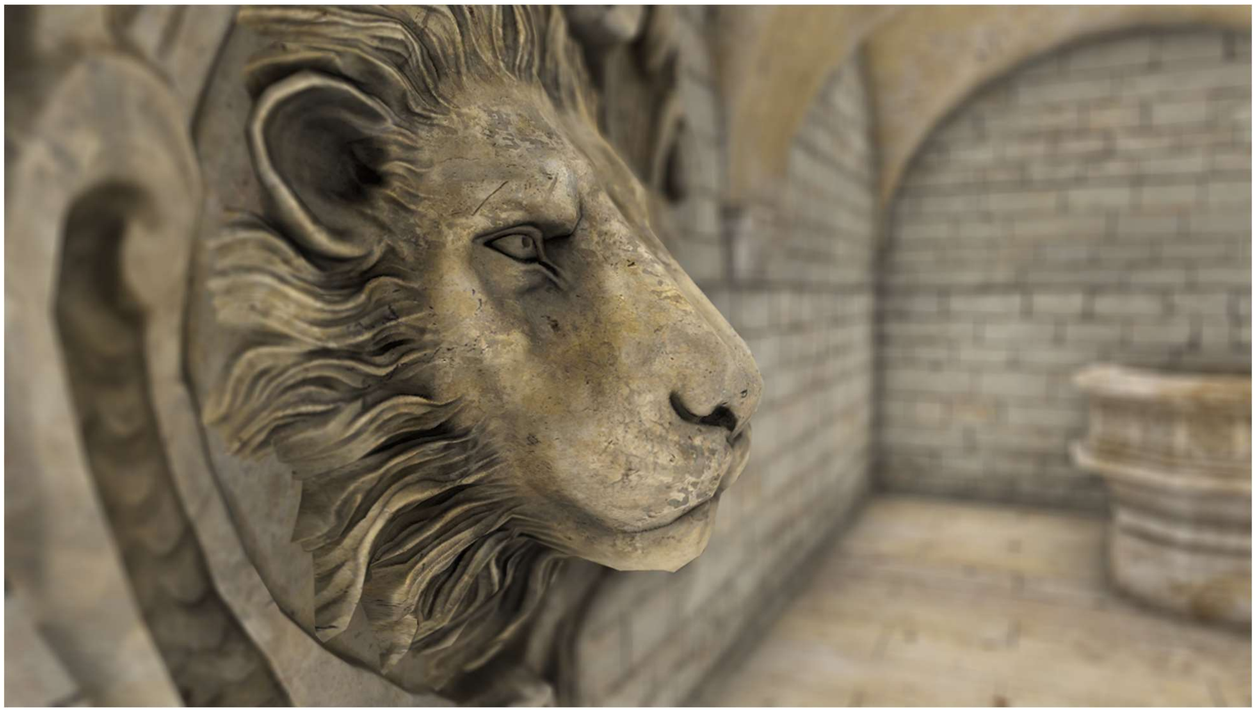
3: SSAO off



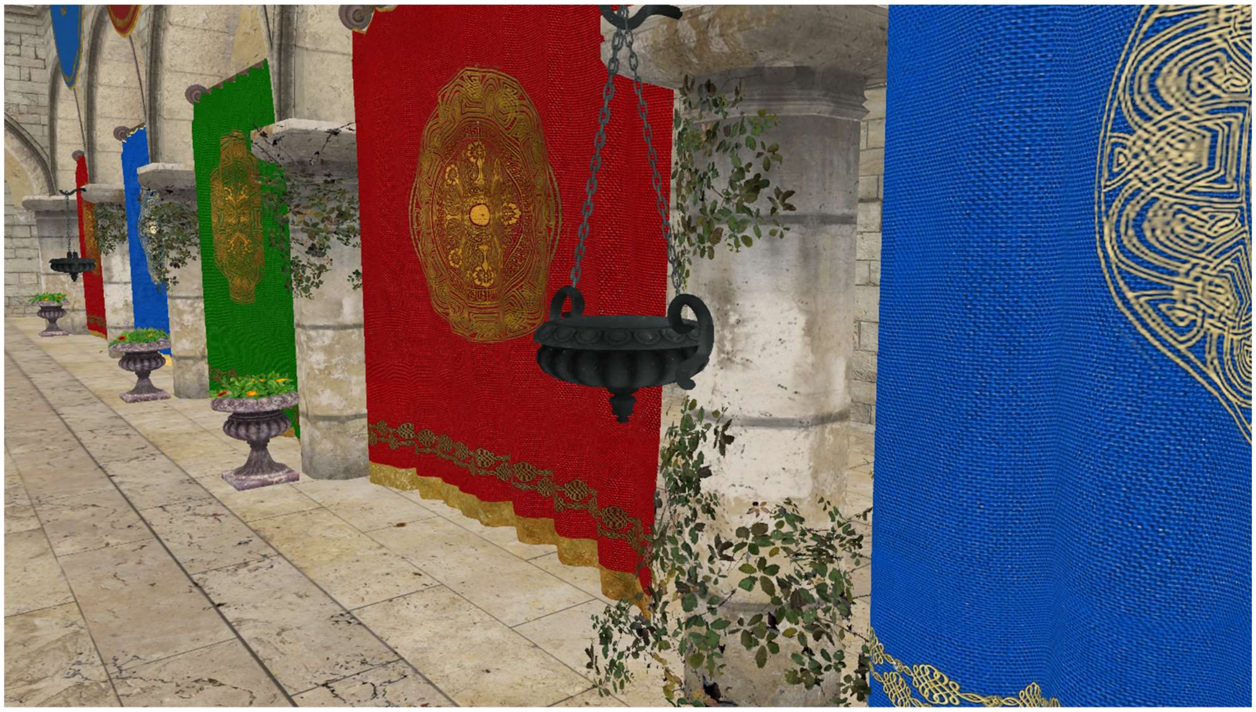
4: SSAO on



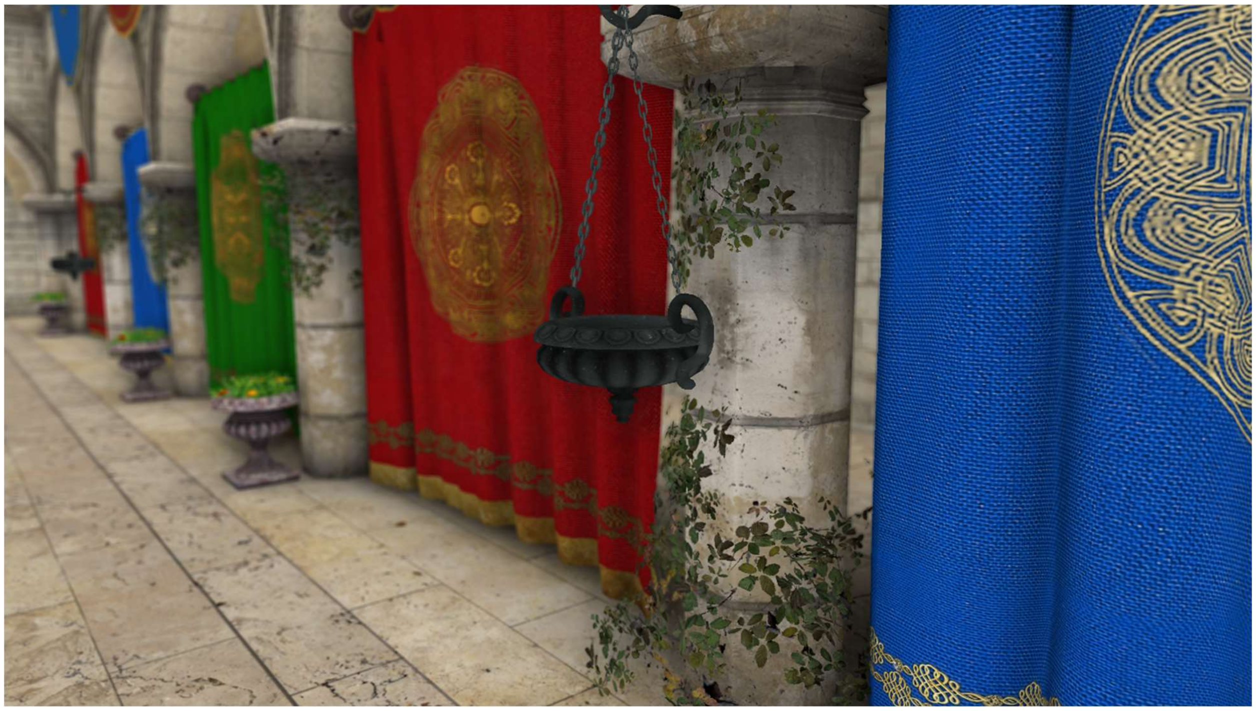
5: DOF off (SSAO on)



6: DOF on (SSAO on)



7: SSAO and DOF off



8: SSAO and DOF on