Screen Space Ambient Occlusion

TSBK03: Advanced Game Programming

August Nam-Ki Ek, Oscar Johnson and Ramin Assadi

March 5, 2015

This project report discusses our approach of implementing Screen Space Ambient Occlusion (SSAO) with deferred rendering. The report will bring forward the theory behind SSAO but also some details from our implementation. Due to the nature of SSAO, deferred rendering is well suited as a preprocessing step to SSAO and our implementation uses this. The report will also describe deferred rendering in theory and how it is implemented using OpenGL. The report will also discuss the visual result of the implementation and how tweaking different parameters affects the result.

Contents

1. Introduction

Ambient occlusion is a method to manipulate the otherwise constant ambient factor in light models for computer graphics. It creates a subtle but often very realistic look with smooth shadows. The look can be compared to lightning conditions on a cloudy day. Ambient occlusion is a global rendering model and therefore quite computationally heavy. Ambient occlusion is compared to radiosity lower cost in computation and gives a better look than plain Phong shading. The basic principle in ambient occlusion is that objects nearby a surface point can contribute to the occlusion of a point depending on the occluders orientation. For example a surface point in a concave area of an object will be occluded, but not a surface point in a convex area.

There are approximations which makes ambient occlusion work faster, Screen Space Ambient Occlusion (SSAO) being one of those. The game Crysis was the first to use SSAO in 2007, it was developed by Vladimir Kajalin working at Crytek. SSAO is the technique which will be discussed in this article. Screen space ambient occlusion makes use of the view space, which optimizes performance due to not taking into account areas not seen by the viewer which makes it fast and well suited to real-time rendering (games etc). Note that today there is also implementations which can compute more accurate ambient occlusion in real time (Horizon Based AO, High Definition AO).

2. Screen Space Ambient Occlusion

Ambient Occlusion is how much each point on a surface is occluded by the surrounding geometry. Depending on how much each point gets occluded, the amount of incoming light will increase or decrease. This means that more occlusion leads to less light and less occlusion leads to more light. This will create the shadow effect in corners and between narrow spaces in objects.

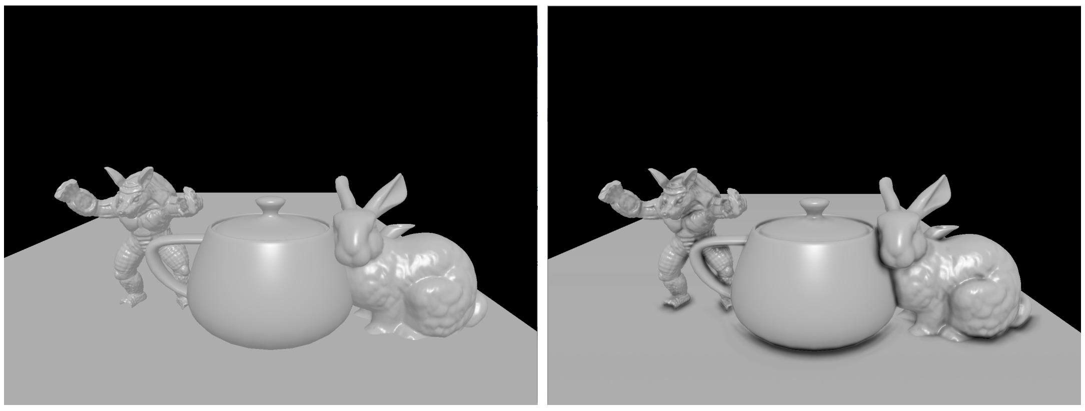

Figure 1: Left image is with Ambient Occlusion off and the right is with the effect on.

If we look at Figure 1 we can see the Ambient Occlusion effect shadows the object in the narrow spaces. Now this effect needs to be simulated in real time. As mentioned before the Ambient Occlusion is a global rendering model and therefore approximations needs to be done to make this effect in real time. One of these approximations is Screen Space Ambient Occlusion (SSAO).

2.1. Implementation of SSAO

To implement the SSAO there are a couple of steps that needs to be done. First of, we need to generate a sample kernel to store different sample points in. These sample points are used to compare if there are some geometry nearby the surface point. Next we need to generate a noise texture so we can rotate the sample kernel in different ways to minimize the artifacts of the effect. Now we can start on working with the SSAO shader and do the necessary steps to make the Ambient Occlusion effect. The last step is to add a blur shader to the SSAO component so the effect becomes more smoother and better looking.

2.1.1. Sample Kernel

First of we need to create our sample kernel. The sample kernel is a hemisphere oriented along the normal of the surface. In this step we need to create a couple of sample points that are randomly placed within this hemisphere. We also need to generate more points closer to the origin of the hemisphere because samples close to the origin contribute more to the occlusion.

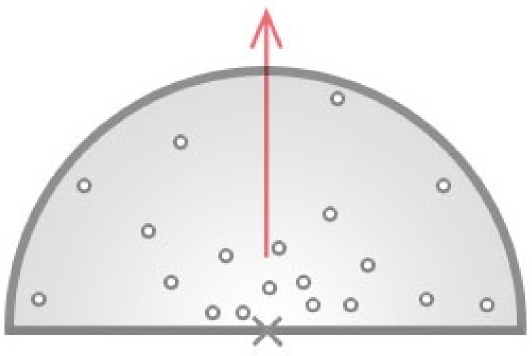

Figure 2: The Sample Kernel, image from Chapman [1]

The distribution of samples should be as seen in Figure 2. We can see that the sample points are clustered near the origin of the hemisphere and less points further away.

2.1.2. Noise Texture

To remove the banding artifacts caused by the hemisphere samples being constant for each surface point, we introduce noise. After creating the sample kernel we need to generate a texture so we can randomly rotate the sample kernel around the surface normal. This will create more randomly distributed sample points and get rid of the artifacts when only using one sample kernel. This will make the result look much better but it will also generate noise in the output. By using a blur shader later it is possible to average this noise. In our case we use a 64x64 texture image and tile it over the screen to rotate the sample kernel and create more arbitrarily located sample points to get rid of banding artifacts in an efficient way.

2.1.3. SSAO Shader

Now it is time to start with the SSAO shader. In our project we have implemented a deferred renderer. This means that the positions and normals for the geometry is accessible in corresponding textures. The first step is to get the position and normal of the current position in the texture. Now we need to take our sample kernel and position it on the position we obtained and then reorient it along the normal.

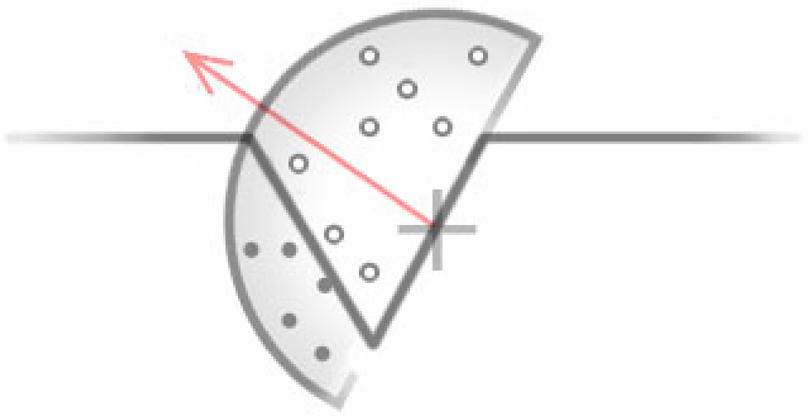

Figure 3: The reoriented Sample Kernel, image from Chapman [1]

In Figure 3 we can see how the sample kernel is positioned on the surface and reoriented along the surface normal.
The next step is to check if the sample points are occluding or not. To do this we need to project each sample point onto the view plane and get the corresponding surface position in the position texture. Now a simple depth check is done to see if the sample position is further away from the surface position or not. If the sample position is further away than the surface position then the sample point will be added to the occlusion. If we look at Figure 3 we can see that the sample points that are behind the surface are colored black to symbolize that they are occluded. When all sample points have been calculated

the mean value of how much the origin point is occluding is calculated.

Now that we have figured out how much each point is occluding we need to have a range check to see if the occluding geometry is near or not. This is done by looking if the distance between the sample point and the surface point is greater then a fixed value. If the distance is greater then the fixed value it is to far away and it will not be occluded.

The last thing we need to do is to normalize and invert the occluding value so we get an value between 0 and 1. This makes it possible to affect the ambient factor so that if there is no occlusion the ambient factor will be multiplied with 1. If it is fully occluded the ambient factor gets multiplied with 0. This will give us the Ambient Occlusion effect.

2.1.4. Blur Shader

Before applying the Ambient Occlusion we need to put a blur effect on the result. This is because the end result looks like dirt and not shadow. If we blur the effect we will get a smoother transition and it will look like shadow instead of dirt. In our project we used a 5x5 box blur to blur the effect.

3. Deferred Shading

Deferred shading is a multiple-pass rendering method that in contrast to the more traditional *forward shading* decouples rendering of geometry from shading calculations. The first pass is used to gather data important for shading calculations such as positions and normals and then store this data in the G-buffer (Geometry-buffer). This buffer contains several textures with per pixel information about the geometry. It is possible to generate these textures with the help of a fragment shader, framebuffer objects and by using multiple rendering targets.

Three different textures are generated and then used to compose the final image. The first two of these contain information about the positions and normals in view space and are generated in the deferred pass. The third contain information about the occlusion and is generated in the SSAO pass. The three different textures and the resulting image can be seen in Figure 9 in A.

Deferred shading was used in this project because of convenience and performance reasons. The occlusion term should be calculated in screen space which make deferred shading a good candidate since the information needed to calculate the occlusion easily can be obtained by texture lookups. The calculation is also done only once for each pixel which gives a performance boost since a forward rendering approach would require the calculation to be done on non-visible fragments which with deferred shading is discarded by depth testing. It is also easy to introduce image post-processing steps into the GPU pipeline since the scene is rendered to a full-screen quad which then can be processed as a normal image.

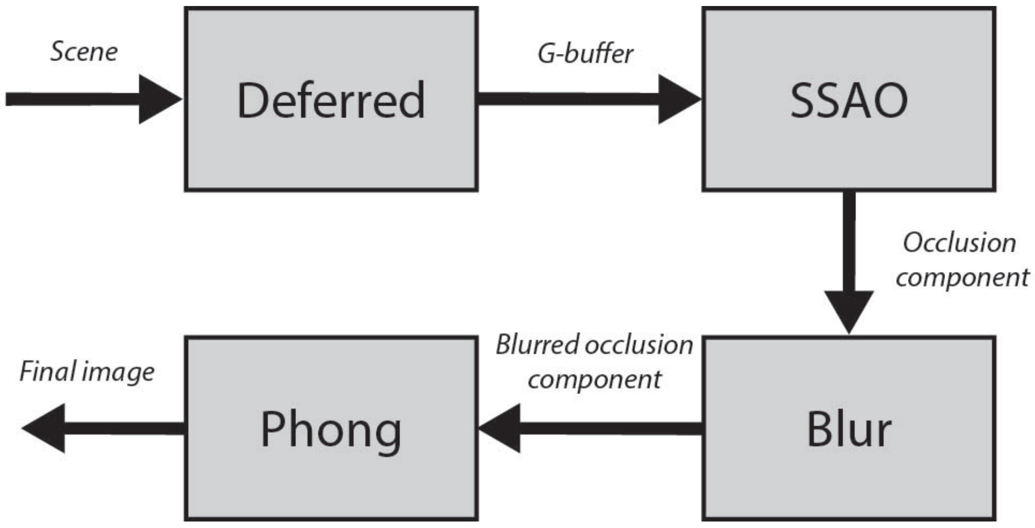

Figure 4: An illustration of the pipeline and its four passes. The scene geometry enters the deferred shader which produces a G-buffer. The G-buffer is then used in a SSAO shader to calculate per pixel occlusion contribution. This term is then blurred by an box-blur shader to average out regularities generated by noise (for reasons why noise has to be used, please see section 2). The resulting term is then used in a Phong shader which shades the pixel. The final image can then be displayed on the screen.

Note that the G-buffer still is available for the Phong shader since the normals and positions stored in the G-buffer is necessary for lightning s.

A simplified figure of the deferred pipeline used in this project can be seen in Figure 4.

3.1. Deferred pass

The inputs of this pass are the vertex positions and normals of the objects in the scene. The normals and positions are transformed to view space and then saved to two render targets. Writing to multiple render targets is possible in OpenGL with framebuffer objects. A framebuffer object (or FBO) is in this case generated with two textures attached meaning that the fragment shader will output its results to these textures instead of on the screen.

Storing textures
It is very important to use correct formats when storing information about positions and normals in textures. Consider the following OpenGL snippet where GL_RGB is used as internal format and GL_UNSIGNED_BYTE is used as type:

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, kWidth, kHeight, 0,
    GL_RGB, GL_UNSIGNED_BYTE, nullptr);
```

This would yield incorrect results since the data would be stored as unsigned bytes and become normalized with a minimum value of 0.0 and maximum of 1.0[5], clamping values below zero to zero. Using the following configuration will store the data as 32-bit floats thus avoiding incorrect values:

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB32F, kWidth, kHeight, 0,
    GL_RGB, GL_FLOAT, nullptr);
```

This format requires more bandwidth but is the format used to store both positions and normals since the performance is good enough.

3.2. Ambient occlusion pass

The position- and normal textures are then used in the next pass where the geometry simply consists of a full-screen quad. The data can be obtained by regular texture lookups and then be used to calculate the occlusion term for each pixel. The calculated occlusion term is then rendered to another texture attached to another FBO. This texture does however only store one value per pixel and can therefore use a more bandwidth friendly format:

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_R8, kWidth, kHeight, 0,
    GL_RED, GL_UNSIGNED_BYTE, nullptr);
```

A thorough description on how the actual occlusion calculation is performed can be found in section 2.

3.3. Blur pass

It is as discussed in section 2 necessary to use noise to rotate the sample kernel. The noise used for this calculation consists in this case of a 64×64 pre-rendered normal texture that is tiled across the screen. The tiling results in noise which can be reduced to a barely visible level by averaging neighboring pixels with a box blur. Please see Figure 5 for a comparison. The blurred occlusion component can then be stored in another texture attached to another FBO since it is undefined behavior to write and read from the same texture. The blurred occlusion component is then ready to be used in the final Phong shading pass.

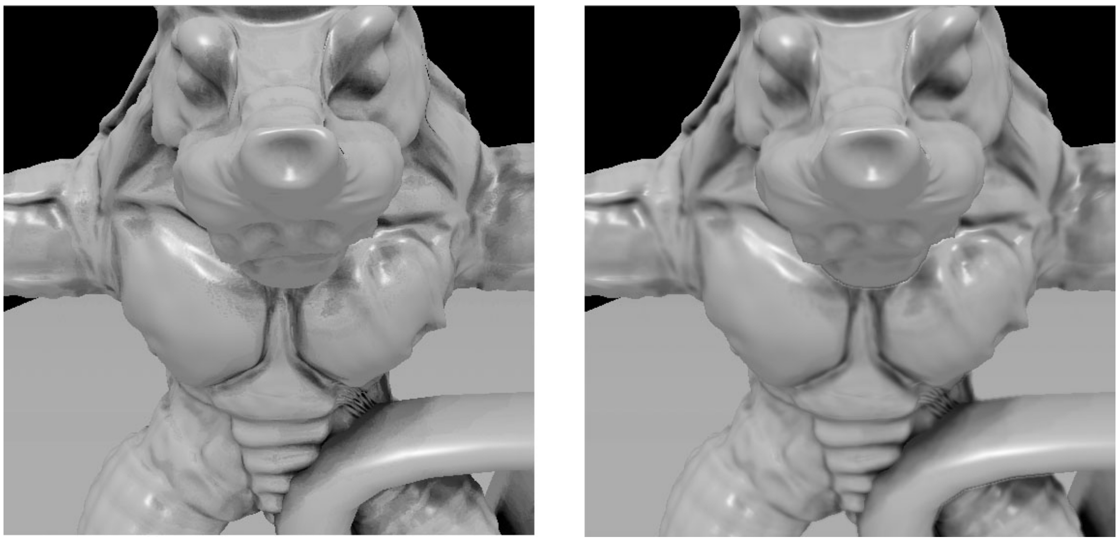

Figure 5: To the left: an image without any blur applied. Notice that the occlusion looks more like dirt than shadow. To the right: 5×5 box blur applied to the same image resulting in a smoother, more shadow-like look.

3.4. Phong shading pass

The occlusion component can be used in any way that the user think looks good in the phong shader. It should however be used to manipulate the ambient term in the phong reflection model to give some kind of realistic result. The implementation used for this project was to directly weight the ambient term with the occlusion term:

$$ambient = occlusion \times i_a k_a, \tag{1}$$

where i_a is the ambient lighting value, k_a is the ambient reflection constant and *occlusion* is the occlusion factor. *occlusion* is calculated by:

$$occlusion = max(1.0 - occlusion, 0.0), \tag{2}$$

where *occlusion* previously has been calculated to be higher the more occludes the specific pixel has. So the pixel will simply not get any ambient light at all if the occlusion term is zero.

4. Result

The result in Figure 6 has the following parameters: a 5x5 boxblur, 60 hemisphere samples, a hemisphere radius of 3.0 and a scaling of the ssao component of 5.0.

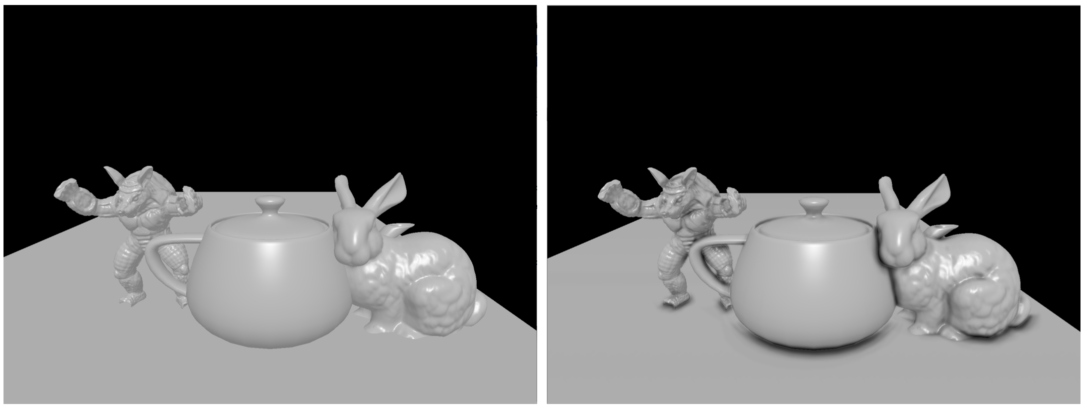

Figure 6: The left image shows the scene with phong shading and no SSAO contribution. In the right image is the SSAO component added.

The figures below will show variations in one parameter while the others are kept constant according to the example above.

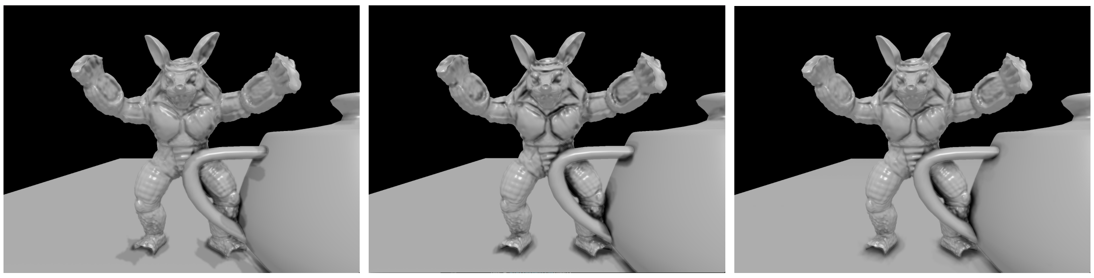

Figure 7: The images above show the effect of the sampling size in the hemisphere, in increasing order from left to right the sample sizes are: 10, 20 and 60.

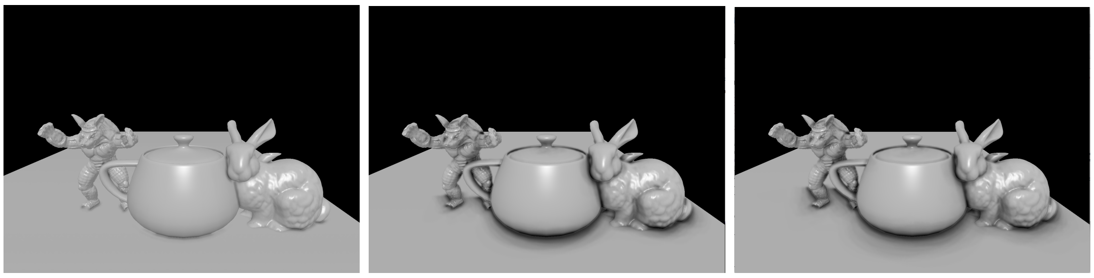

Figure 8: The images above show how the radius impacts appearance. Radius size from left to right: 1, 8 and 13.

5. Discussion

5.1. Parameters

There are a few parameters in SSAO that affects the appearance of the SSAO. In this section will revolve around the contribution of each parameter to the visual result and computation weight. Below is a list of all parameters:

- Amount of samples in the hemisphere
- Radius of hemisphere
- Blur kernel size
- scaling factor of ssao component.

5.1.1. Hemisphere samples size

In the results section a collection of images can be seen with different values for the hemisphere sample size. The sample size can have a very unpredictable effect on the visual result. In figure 7 with a sample size 20 the result is quite dark, not noticeably darker in narrower areas as should be, the darker areas are were arbitrarily placed. The sampling size is too small and the algorithm is not able to find a distinction between narrow area and open surfaces, the shadows also become very hard and dark. In the image with 60 samples the shadows are softer and in more likely areas. The conclusion is that higher sample size will always yield a more accurate result. However a greater sampling size comes with a cost which increases linearly.

5.1.2. Hemisphere radius

The radius as can be seen in figure 8 affects the visual result through a shadow that ranges further. This can be explained by the samples being placed a greater distance from the occlude point which increases the likelihood of hitting an area which contributes to occlusion. This parameters correctness is merely aesthetic, and a good value can only be found by comparing visual results. Because the computation is the same regardless of the value of the radius, this parameter should be changeable without any impact on performance.

5.1.3. Blur kernel size

The blur shader makes a box blur on the SSAO texture. The blur will make the ssao impact have a smooth transition giving soft shadows. The result will be more visually appealing and eliminate the high frequency noise generated by the randomly rotated hemisphere. The size of the kernel will affect the appearance and will look better with a larger kernel to a certain degree. If the kernel is too large the blur will make the effect of the SSAO unnoticeable which is undesired. Also this factor has no correct value and will be found empirically comparing visual results. The performance impact is quite heavy because the blur kernel is quadratic and increases in blur kernel size will increase the computations per pixel quadratically

5.1.4. Scaling factor

The scaling factor is a weight on how much the SSAO should affect the final result. This value is also arbitrary and can be determined depending on visual result. Thus this only being a scalar multiplication the effect on performance is unchanged considering any value.

5.2. Reconstructing position from depth

It is possible to reconstruct positions directly from the depth value stored in the Z-buffer instead of naively storing all of the positions in a texture. This would give an increase of performance since we can skip the 4-byte per pixel texture for storing the position data, thus decreasing the size of the G-buffer. There are different ways to reconstruct the position from the Z-buffer and some of these methods are discussed in Pettineo [2].

5.3. Advantages and disadvantages

The SSAO approach requires no precomputation (except for the deferred rendering) and can easily be implemented in an existing pipeline. The computations are not dependent on the polygon count, but on the number of pixels in the ND-buffer, which is a big advantage compared to other methods, the only limitation factor then becomes the memory bandwidth, which for most modern graphics cards are probable to be of sufficient size. One of the disadvantages with the SSAO method is that even though the calculations is not dependent on the polygons, the visual result gets less artifacts with a higher polygon count in the geometry.

6. Conclusion

The implementation done during this project shows that SSAO is a method that easily can be implemented in a deferred rendering pipeline and give results that look better than just Phong shading alone.

A. Images

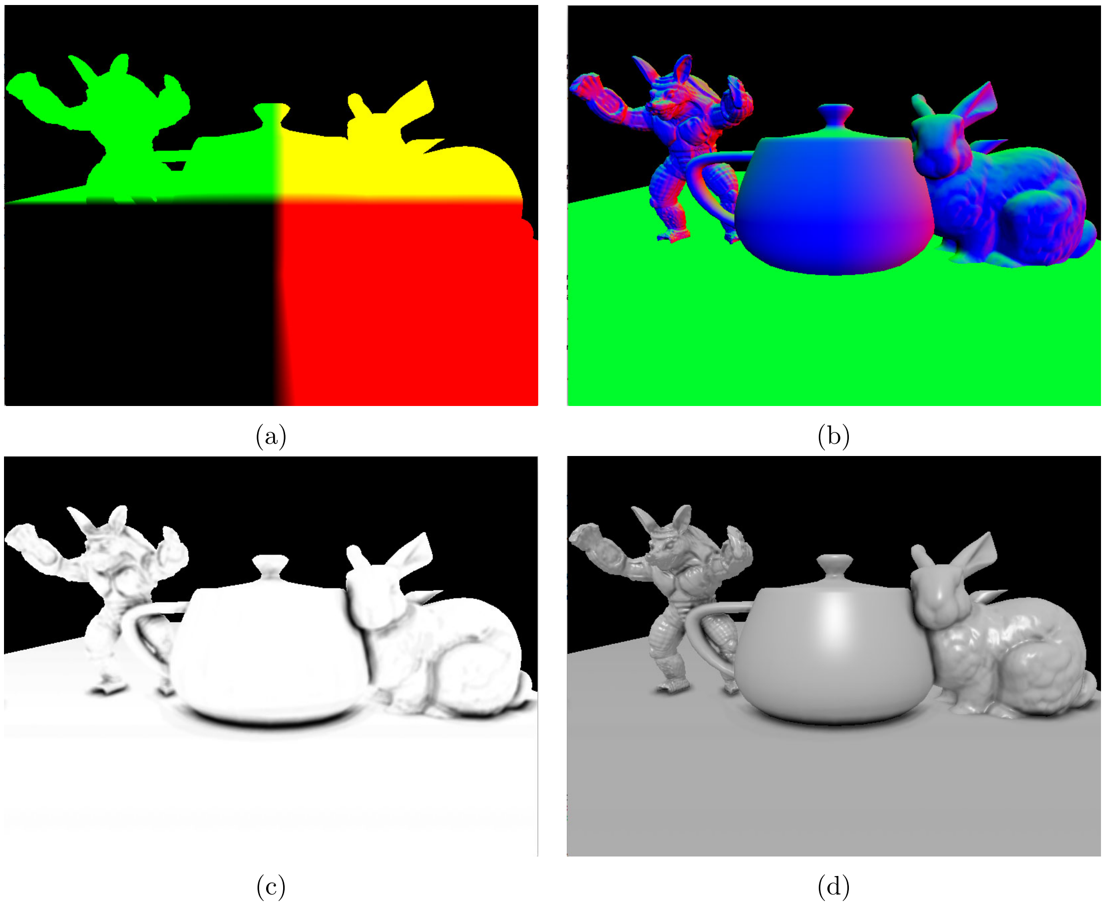

Figure 9: (a): The positions of the objects in view-space. (b): The normals of the objects in view-space. (c): The occlusion contribution of the scene. (d): The final image which is composed of the textures in (a), (b) and (c).

References

[1] John Chapman. Ssao tutorial, January 2013.

[2] Matt Pettineo. Position from depth 3: Back in the habit, March 2009.

[3] Ingemar Ragnemalm. So how can you make them scream?, 2008.

[4] Perumaal Shanmugam and Okan Arikan. Hardware accelerated ambient occlusion techniques on gpus. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, I3D '07, pages 73–80, New York, NY, USA, 2007. ACM.

[5] Dave Shreiner, Graham Sellers, John M. Kessenich, and Bill M. Licea-Kane. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*. Addison-Wesley Professional, 8th edition, 2013.