# Screen Space Ambient Occlusion

Visualização e Iluminação I

MIEI

António Silva
A73827
a73827@alunos.uminho.pt

Marcelo Queirós Pinto
Pg35396
mqspinto@gmail.com

January 31, 2018

# Contents

# Chapter 1

# Introduction

Lighting is perhaps the most important factor that contributes to a virtual scene's realism. Traditional lighting methods have evolved to offer increasingly more precise and convincing virtual environments, but such improvements come with a performance cost. Such a cost might not matter when associated with pre-rendered scenes, such as images or videos, but when we aim to offer interactive programs such as games, those costs must be kept to a minimum.

In this essay, we'll present an alternative way to improve the quality of a real-time 3D scene without resorting to more expensive techniques called Ambient Occlusion.

# Chapter 2

# Ambient Occlusion

In Computer Graphics, ambient lighting is often times a constant added to the global illumination, simulating light scattering. In real life, however, this scattering occurs with different intensities, depending on the environment's topology.

In order to produce a more realistic result, the occlusion to which an area is subjected as a result of proximity to surrounding geometry should be taken into account. Consequentially, creases, wholes and corners should be darker, since they represent areas where close geometry exists.

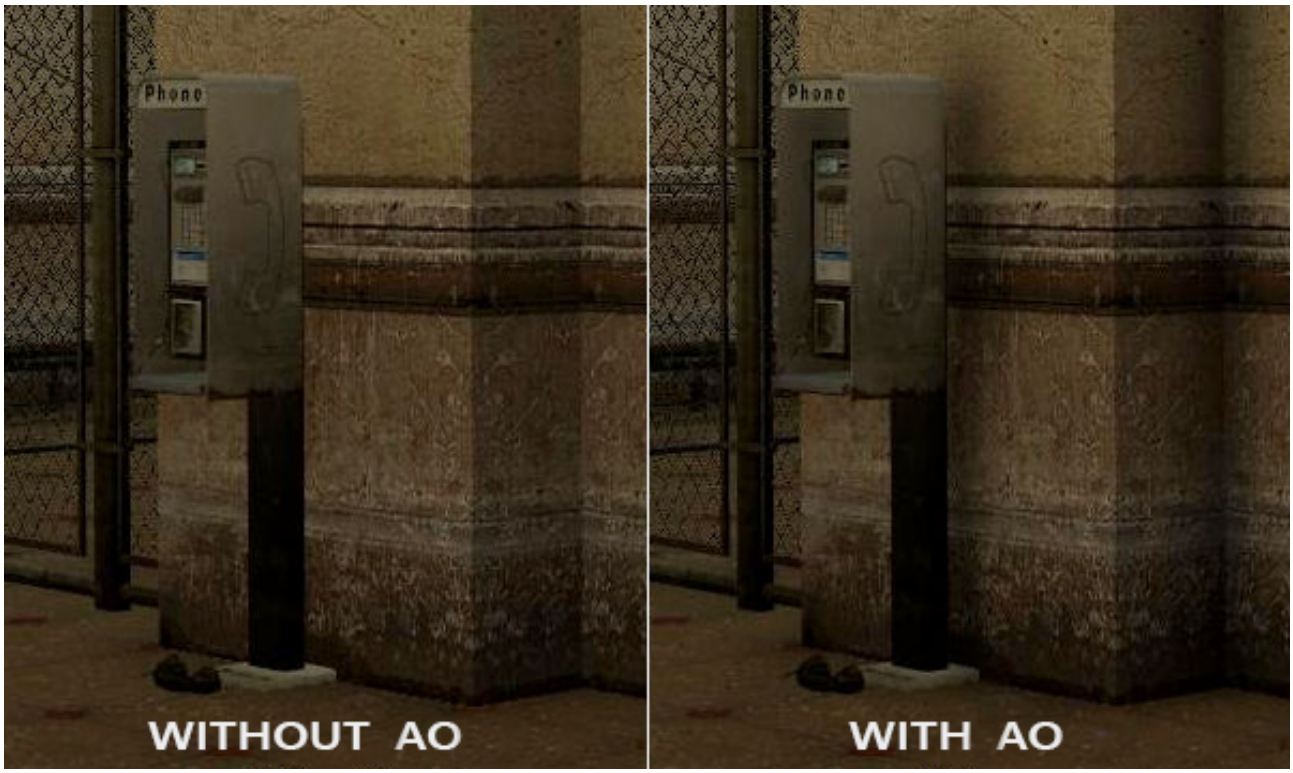


Figure 1: Results of applying AO to a scene.

Traditional implementations of Ambient Occlusion work by analyzing geometry surrounding the position from which we want to obtain the occlusion. This is computationally expensive and this cost increases exponentially the more complex a scene is, rendering it (*no pun intended*) useless for real-time use scenarios such as games.

# Chapter 3

# Screen Space Ambient Occlusion

## 3.1 Crytek's implementation

In 2007, Crytek published a solution to this problem: Screen Space Ambient Occlusion (SSAO) which, as the name indicates, uses a scene's depth in screen-space to determine the amount of occlusion instead of real geometrical data. The outcome has negligible visual differences to standard AO but, since it's independent from scene complexity, it ends up being much faster and hence usable in real time rendering.

The technique works by calculating an occlusion factor based on depth samples taken in a sphere kernel around the fragment's position. The number of samples that have a higher depth value than the fragment's depth represents the occlusion factor.

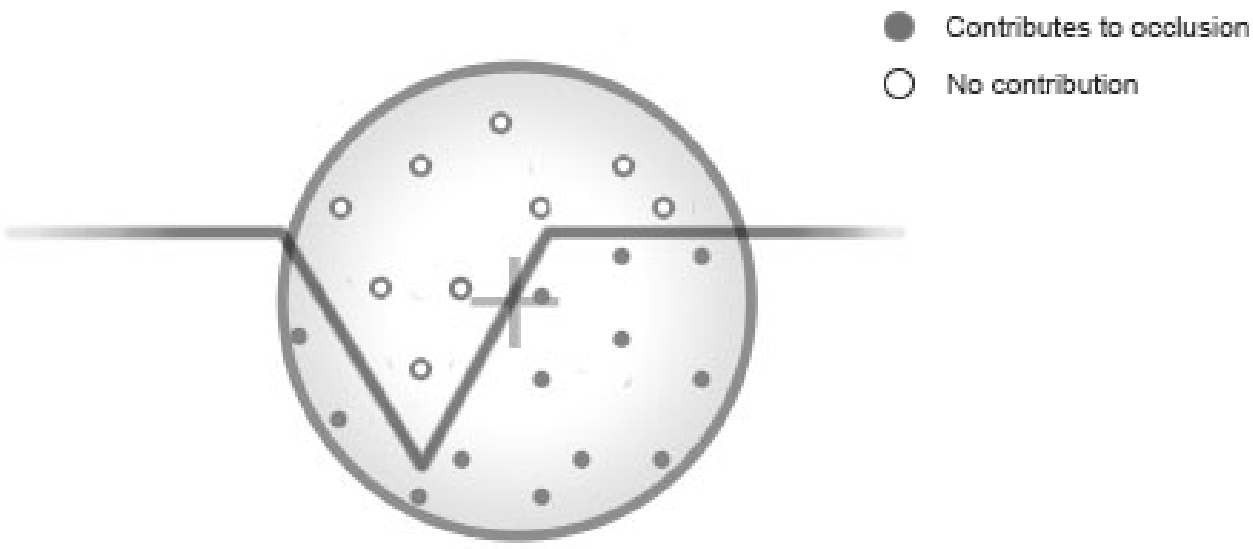


Figure 2: Kernel Sphere.

Unsurprisingly, the quality of the result is proportional to the number of samples, which needs to be minimized in order to achieve decent performance. On the other hand, if the sample count is too low, we get an artifact called banding. This can, however, be improved by randomly rotating the kernel. This generates a noticeable noise pattern that can also be fixed by adding an extra blurring step.

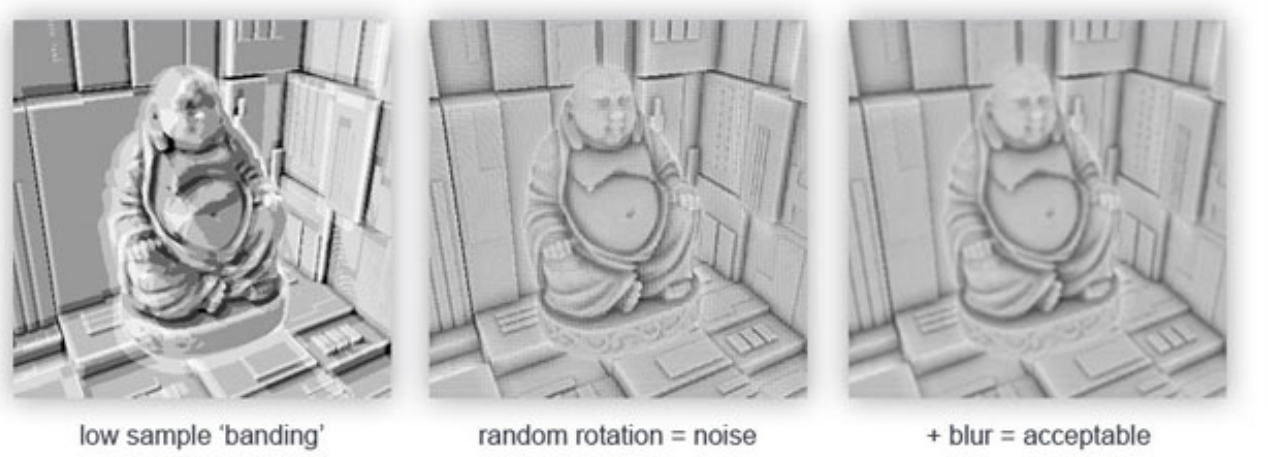


Figure 3: Banding noise.

### 3.1.1 Problems with Crytek's implementation

On a flat surface, due to the spherical nature of the kernel, approximately half the samples end up contributing to the occlusion, when no occlusion should occur. This means surfaces which in theory should have no occlusion end up looking darker.

## 3.2 Improving Crytek's approach: Normal-oriented hemisphere

By sampling around a normal-oriented hemisphere, we do not consider the fragment's underlying geometry as contribution to the occlusion factor:

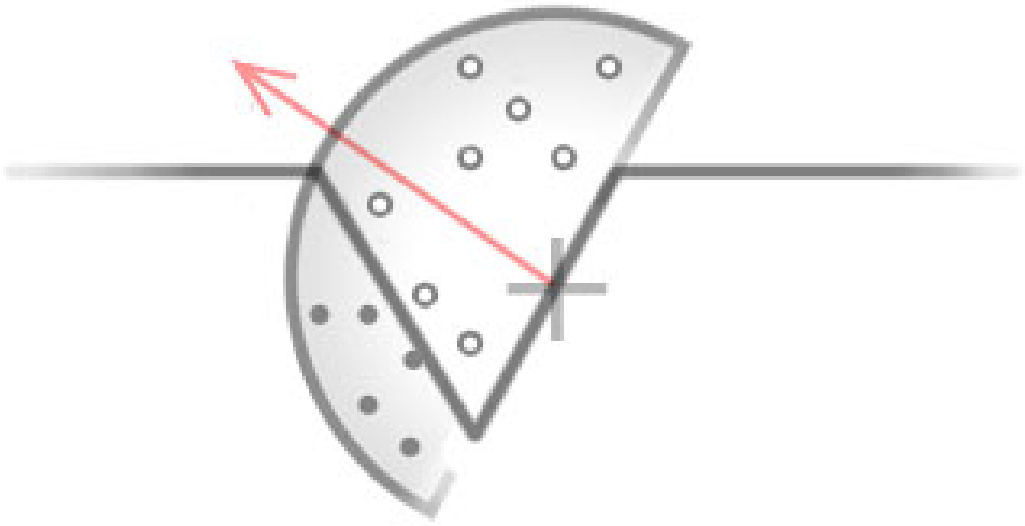

Figure 4: Normal oriented kernel hemisphere.

# Chapter 4

# Improved Version Implementation

## 4.1 Deferred Shading: Intermediate Pass

This particular version of SSAO benefits from a combination with deferred rendering, since the position and normal vectors that are needed for calculations are already gathered beforehand. The first pass for the program will, therefore, be based on simple vertex and fragment shaders that only fetch position and normal maps.

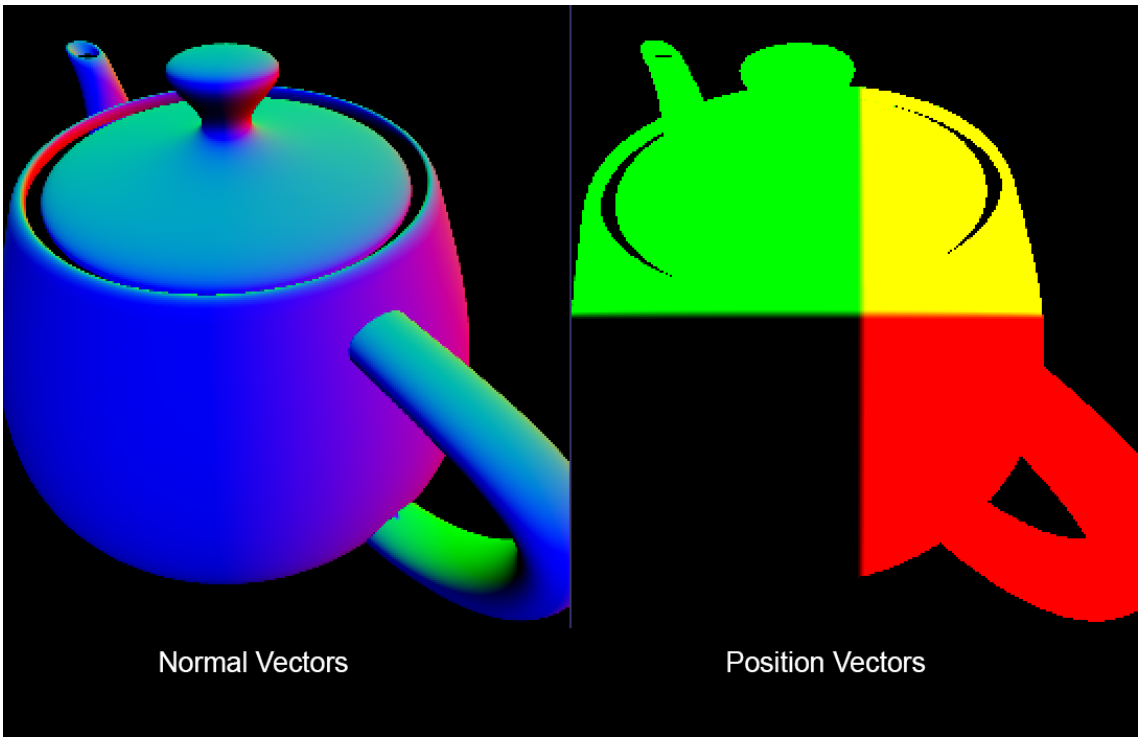


Figure 5: Fetched teapot normal and position maps.

## 4.2 SSAO Pass

### 4.2.1 Vertex Shader

The vertex shader for this phase is simple, since all information has been gathered in the previous pass.

### 4.2.2 Fragment Shader

#### Random Kernel generation

In order to create well distributed samples around the origin, we can generate random 3D vectors and multiply them by a random value between 0 and 1. It is, however, a good idea to increase the sample count closer to the origin, placing a larger weight on the nearest occlusions.

```
for (int i = 0; i < MAX_KERNEL_SIZE; ++i) {
    vec2 seed=vec2(i,MAX_KERNEL_SIZE);

    random_kernel[i] = vec3(
    rand(seed)*2.0-1.0, //varies between -1 and 1
    rand(seed)*2.0-1.0, //varies between -1 and 1
    (rand(seed))); //varies between 0 and 1, otherwise if we had between -1 and 1, we'd have a sphere kernel

    random_kernel[i]=normalize(random_kernel[i]);

    // distribute within hemisphere
    random_kernel[i] *= rand(seed);

    //accelerating interpolation applied to distribution
    float scale = float(i) / float(MAX_KERNEL_SIZE);
    scale = mix(0.1f, 1.0f, scale * scale);
    random_kernel[i] *= scale;
}
```

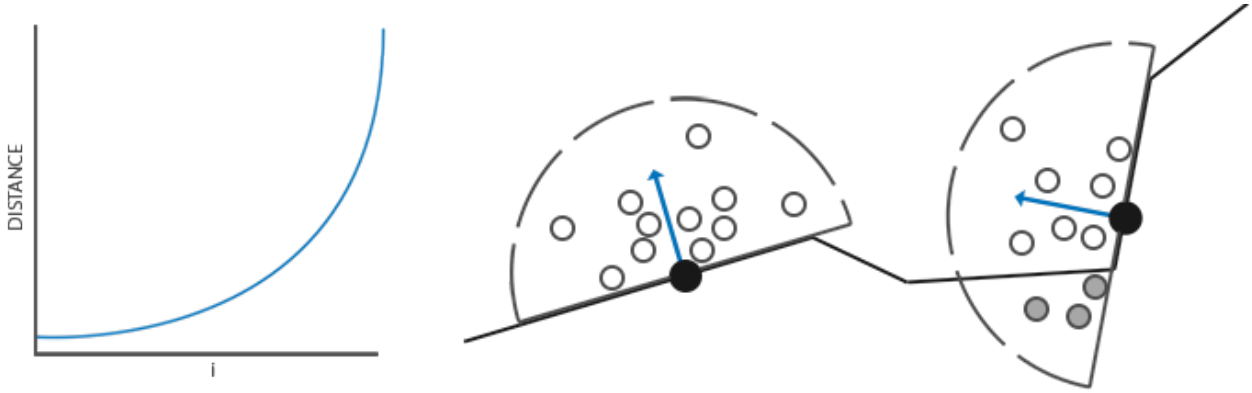


Figure 6: Accelerating interpolation function.

#### Change-of-Basis Matrix

The previous vectors are in tangent space and, in order to sample the depth at the generated position from the depth texture, we need them in screen space. To accomplish this, we first need to convert them from tangent to view space using a method called ***Gram-Schmidt process***, while also incorporating the random rotation through the use of a vector extracted from a noise texture. The result is a *TBN* matrix, which will be multiplied by the vectors.

```
//random rotation vector
//noise texture is tiled all over the screen by dividing screen size by texture size
vec3 rvec = texture(noise_texture, texPos.xy*(resolution.x/64,resolution.y/64)).xyz;

// gram-schmidt
vec3 tangent = normalize(rvec - normal * dot(rvec, normal));
vec3 bitangent = cross(normal, tangent);
mat3 tbn = mat3(tangent, bitangent, normal);
```

### Main cycle and occlusion calculation

In this last phase, we iterate through the generated kernel, converting the vectors from tangent to view space using the *TBN* matrix and then from view to screen space multiplying them by the projection matrix, performing a w-divide (or perspective divide) and finally distributing the values through 0 to 1.

```
for (int i=0; i < MAX_KERNEL_SIZE; ++i){

    // from tangent to view space position
    sample = (tbn * random_kernel[i]) * kernel_radius;
    sample = sample + pos.xyz;
    float generatedDepth=sample.z;

    // From view space to screen space
    offset = vec4(sample, 1.0);
    offset = CamP * offset; // In the range -w, w
    offset.xyz /= offset.w; // in the range -1, 1
    offset.xyz = offset.xyz * 0.5 + 0.5;

    [...]

}
```

Having the sample in screen space, we can now extract the depth from the depth map. After that, the last step is unprojecting the full vector back into view space so we can compare its depth with the randomly generated one (we could also do the opposite and project the generated vector into screen space). A range check is also needed in order to exclude occlusions that fall outside of the kernel radius.

```
// depth at sample's position
sample_depth = texture(depth_texture, offset.xy).r;

//back to view space
vec4 VSsample = vec4(offset.xy, sample_depth, 1.0); // range 0, 1
VSsample.xyz = VSsample.xyz * 2.0 - 1.0;
VSsample = CamPI * VSsample;
VSsample /= VSsample.w;
float actualDepth=VSsample.z;

float rangeCheck= abs(pos.z - actualDepth) < kernel_radius ? 1.0 : 0.0;
occlusion += (actualDepth >= generatedDepth ? 1.0 : 0.0)* rangeCheck;
```

The previous occlusion factor has to be normalized and inverted, so we produce a value that can be stored in an occlusion map. We can also control how aggressive the effect is by introducing an *ssao power* variable:

```
occlusion = 1-(occlusion / float(MAX_KERNEL_SIZE));
occlusion = pow(occlusion, ssao_power);
occMap = vec4(occlusion);
```

## 4.3 Results

With these shader passes implemented, we get the following result with a kernel size of 64 samples applied to the teapot scene:

Figure 7: Occlusion map on teapot.

As we expected, areas where close geometry exists like the teapot handle get a localized "shadow effect" which is caused by occlusion.

# Chapter 5

# SSAO without projection

An alternative to the previous implementation that does not rely on projections exists [3], doing all calculations in 2D. It also works best on top of a deferred renderer, since it takes the position and normal buffers and generates a one-component-per-pixel occlusion buffer.

Instead of gathering samples around the origin on an hemisphere, it instead treats all neighboring pixels as small spheres, adding together their contributions. The occlusion contribution will depend on the distance to the occludee and the angle that the sphere makes with the occludee's normal. The occlusion will, therefore, increase linearly with decreasing sample distance and angle in relation to the origin's normal.

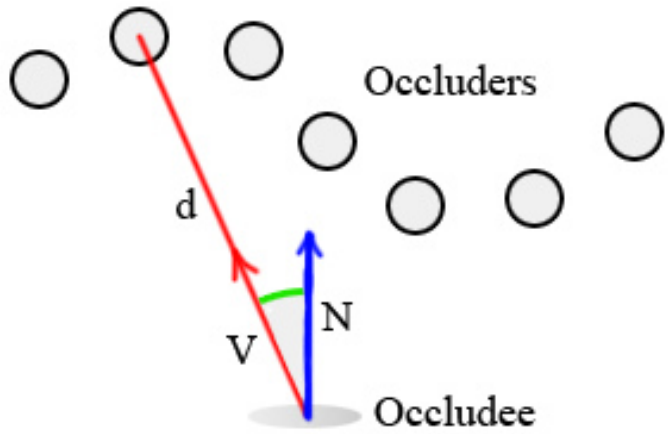


Figure 8: Origin-to-sample ray in relation to origin's normal.

This version also manages to solve the self occlusion problems of Crytek's variant, since the samples are limited to positions that obey to a certain angle amplitude in relation to the normal.

# Chapter 6

# Performance Measuring

In this chapter we'll measure the impact that adding SSAO to a scene has. The scenes with SSAO are the same as the original scene, but with an extra pass for occlusion calculation which is multiplied by the scene's Ambient lighting. Since the computationally heavier part of the program is located in the last loop, the main test parameter will be kernel size (or number of samples).

| Kernel Size | Average FPS | Performance loss |
|---|---|---|
| 0 (No SSAO) | 2946 | – |
| 16 | 2375 | 19% |
| 32 | 2240 | 23% |
| 64 | 129 | 95% |

As we can observe, the inclusion of SSAO in the scene imposes a significant (but acceptable) toll on the scene's performance with around 16 to 32 samples. Increasing the number of samples to 64 yields a major performance decrease of 95%, which is unusable in real-time. Already measured results with 16 samples for the SSAO version without projections [3] point to a toll of around 16% in performance compared to the original scene, which is slightly better but in line with the results achieved here.

Figure 9: Scene without SSAO on the left with 175fps and scene with SSAO on the right with 110 fps.

# Chapter 7

# Disadvantages of SSAO

So far, SSAO has shown solid results and implementation flexibility, however there are some disadvantages comparatively to geometry dependent AO that are common to both versions presented.

- The raw output is noisy due to the random distribution of the samples and both solutions to this issue (increasing samples or adding an extra blur pass) incur in performance loss. This performance hit is less significant with the blur pass, but also adds another artifact: areas of occlusion/non-occlusion will tend to leak, most noticeably where there are sharp discontinuities in the depth buffer such as object edges.
- Performance is very dependent on sampling radius and distance to the camera, since objects near the front plane of the frustum will use bigger radiuses than those far away.
- Does not take into account hidden geometry (especially geometry outside the frustum).

# Chapter 8

# Conclusion

With all the information gathered here, we can come to the conclusion that we can achieve very good results visually when compared to traditional geometry dependent solutions for Ambient Occlusions, with acceptable performance downgrades. Nowadays, most real-time programs offer the option to add shading techniques such as SSAO, which proves the usefulness of improving visual quality without imposing limits on geometry.

# Bibliography

[1] John Chapman's blog
john-chapman-graphics.blogspot.pt/2013/01/ssao-tutorial.html

[2] Learn OpenGL
learnopengl.com/Advanced-Lighting/SSAO

[3] Gamedev.net: 2D calculations only SSAO
gamedev.net/articles/programming/graphics/a-simple-and-practical-approach-to-ssao-r2753

[4] Article about improving SSAO specific artifacts
mtnphil.wordpress.com/2013/06/26/know-your-ssao-artifacts