Fast Realistic Rendering: SSAO

Albert Gil Saavedra

May 2019

1 SSAO Program

1.1 Contents:

The SSAO program has been developed using Visual Studio in Windows. The main folder contains the Visual Studio solution along other folders containing the files needed for the program. The main folders and files are the following:

- Dependencies (folder): Contains all the dependencies of our program. Includes assimp, GLEW and GLFW.
- SSAO(folder): Contains all the codes files and the resources files.
- Res(folder): Contains all the resources needed for the program. The main folders are shaders, textures, cat. The shaders folder contains all the shaders used, and the cat folder contains the model used, which in this case is a cat.
- Src(folder): Contains all the code files and the vendor folder. In the vendor folder you can find the libraries used as glm, imgui and stb_image.
- App.cpp: Main code file. It contains all the SSAO implementation code.

In order to guarantee the correct behavior of the SSAO program all theses files must remain in the same folder that they are.

1.2 User Instructions:

The SSAAO is implemented in Visual Studio, and configured in 32 bytes. It can be executed in release or debug mode. In order to compile the main program you need to:

1. Open the solution file "SSAO.sln" with Visual Studio.
2. Select the mode "Debug/Release" and 32 bytes.
3. Execute the App.cpp file. Or Just click the run button also named "Local Windows Debuger".

The main program will start with the default mode(SSAO Lighting). The program shows the interface in the left corner of the scene. The interface is used using the mouse, the camera can be rotated also using the mouse and moved with the keyboard. The main actions are the following:

- Mouse Movement: Rotate the camera.
- Mouse wheel movement: Zoom in or zoom out.
- Press Left Shift : Make the mouse visible and allow the user to modify the interface. While pressing it the camera will not move. The shift must be not released until you wanted to return to the regular application.

The interface shows the performance of the application and the current mode. It allows to change mode, to show one mode only the current mode has to be selected, otherwise it shows the default mode. Also includes sliders for different variables which will allow the user to see the effect of each one without changing the code. In the *Fig 1* it is represented the initial scene.

The applications is configured with a 1280x720 resolution. If you need to change the resolution it can be done by changing the variables in the App.cpp. They are declared at the start of the code file. It is important to modify the shader_ssao noisScale calculation that includes the size of the windows application.

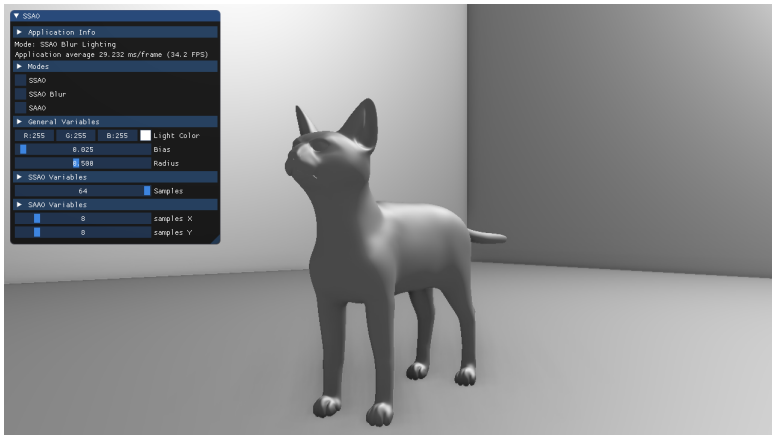

Figure 1: Application default visualization.

2 Implementation

2.1 General Features:

Our objective was to implement SSAO and the Separable Aproximation of AO, and compare the performance between them.

As we are interested in seeing the results of the SSAO textures and the effect of the different variables implied in the calculation we added the SSAO and SSAO Blur modes which show the only the textures obtained which each effect. Using this modes is easier to see the effect that have each variable.

As it is said before all the implementation code is located in the App.cpp code file. The main code follows the next structure:

1. Settings and Variables declaration.
2. Main Loop.
   (a) Window Initialization and configuration.
   (b) Data arrays.
   (c) Initialization of VAO, VBO, FBO, Textures, shaders and model.
   (d) Rendering Loop.
3. Utility functions.

2.2 Screen Space Ambient Occlusion:

The SSAO technique that we have used have four main stages: Geometry draw, Generate SSAO texture, Blur SSAO texture and Lighting. In this section we will explain the implementation and the shaders used in each stage. [1]

2.2.1 Geometry draw:

The main purpose of the Geometry draw stage is to save the geometrical information into the G-Buffer. At the contrary of the other three stages in this one we draw the actual scene, meaning that we draw the actual objects not a quad as we are going to do in the other three. The steps of the stage are the following:

1. Bind the G-Buffer.
2. Bind the shader which will take care of the geometry stage, and send the necessary uniforms.
3. Draw objects.
4. Unbind the G-Buffer

The G-Buffer is a framebuffer created previously, and it contains three main textures: Position, Normal and Albedo and a depth renderbuffer object. The data is filled in the geometry stage using the fragment shader.

Geometry shader:

[1] The implementation of the SSAO used in our program is based on the implementation used in `https://learnopengl.com/Advanced-Lighting/SSAO`.

The shader_geometry.shader is the shader file used in the geometry stage and it contains both of the vertex shader and the fragment shader. The vertex shader has two main purposes, the first is to send the positions and normals in the View Space to the fragment shader, and the second is to draw the positions using the MVP matrix.

The fragment shader recieves the positions and normals from the vertex shaders and send them to the G-Buffer with the albedo that is the diffuse color vector. Our positions and normals are in the View Space because all the occlusion algorithm is based in the View Space, so we need to transform it only at the beginning.

2.2.2 SSAO Texture Generation:

The main purpose of this stage is to calculate the occlusion and generate the texture that holds it. As we have said before in these stage we are only going to draw a simple quad covering all the window, and using the data from the G-Buffer we will create the SSAO texture. The steps of the stage are the following:

1. Bind the SSAO framebuffer.
2. Bind the shader which will take care of the SSAO stage, and send the necessary uniforms.
3. Bind the Position, Normal and noise textures.
4. Draw the quad.
5. Unbind the SSAO framebuffer.

In this stage we calculate the occlusion using the shader, but before we can talk about it we have to talk about the kernel created and the noise texture. These stage needs the creation of one new framebuffer dedicated tot the SSAO which will hold the SSAO texture, the creation of two kernels made of 3D vectors and one texture to hold one of the kernel's texture.

The occlusion factor is calculated by taking multiple depth samples in a semi-sphere sample kernel surrounding the fragment position and compare each of the sample with the current fragment's depth value. For this reason we have created a sample kernel which all the samples are randomly distributed, which is generated in the tangent space with the normal vector pointing to the positive z-direction. Also we have added some randomness by creating a small random rotations vectors and the texture that holds it. This texture is one of the three textures we have to attach in these stage.

SSAO shader:

The main purpose of the vertex shader is to draw the vertex of the quad and send the texture coordinates to the fragment shader.

The fragment shader takes as input the three textures: position, normal and noise, and the semi-sphere sample kernel. The first thing we have to do is to tile the noise texture as its coordinates do not match the texture coordinates of

the quad. Once we have done it, we sample the corresponding texture for the three textures using the texture function, and create a matrix to transform the coordinates from the tangent space to the view space where we are going to do all the calculations. Now we start the occlusion calculation.

In order to calculate we have to loop over all the samples of the sample kernel. In each iteration we will do the following steps:

1. Transform the vector of the sample to the view space and add to the current fragment position.
2. Transform the sample to clip space so we can sample the position/depth value as if we were rendering its position directly to the screen.
3. Get depth value of the kernel.
4. Introduce the rangeCheck correction.
5. Compare the sample depth with the fragment depth and if the fragment depth is bigger add $1.0 * rangeCheck$ to the occlusion sum, otherwise sum 0.

Finally the occlusion calculated will be 1 minus the occlusion added through the loop divided by the kernel size.

2.2.3 SSAO Blur:

The main purpose of these stage is to blur the SSAO texture we have created previously. In order to do that we need to create another framebuffer with a texture attached as we have done it before. The steps of the stage are the following:

1. Bind the SSAO blur framebuffer.
2. Bind the shader which will take care of the SSAO blur stage, and send the necessary uniforms.
3. Bind the SSAO texture.
4. Draw the quad.
5. Unbind the SSAO blur framebuffer.

SSAO Blur shader:

The vertex shader is the same used in the previous stage, and the fragment shader is a simple shader that blurs the SSAO texture, which it is used as an input.

To create the blur effect we have to loop the surrounding SSAO texels between -2 and 2 sampling the SSAO texture an amount of times exactly as the noise texture's dimension. Then each iteration offset each texture coordinate by the exact size of a single texel. And finally when we have finished the loop we have to average the result using again the size of the noise texture's.

2.2.4 Lighting:

The lighting stage is the last, and its main purpose is to apply light to the scene. As said before we have applied a simplified version of deferred renderer as we already have the position and normal vectors stored in the G-Buffer. [2] Also we apply the ambient occlusion factor in the lighting calculation using the fragment shader.

Lighting shader:

The vertex shader is the same as we have used previously to draw the quad, and the fragment shader is the one which perform all the light calculations.

The fragment shader takes as input four textures(position, normal, albedo and the SSAO texture), and the light parameters(color, position and the linear and quadratic terms for the attenuation). Using this it is easy to calculate the ambient, diffuse and specular components of the light using a Blinn-Phong shading Model.

For the SSAO texture it is really easy to introduce it in the ambient component. We only need to multiply this component for the AO red component and the SSAO effect will be introduced.

2.3 Separable Approximation of Ambient Occlusion:

The SAAO method structure is the same as the SSAO method, but now we will calculate the ambient occlusion separating it by two components: x and y. Also we have applied a Gaussian blur instead of the blur technique used in the SSAO method.

As we are calculating the ambient occlusion divided in two components we will need two framebuffers, with one texture associated each one for the SAAO texture generation. In the first one it calculates the x component and the second one calculates the y component, and using the first one as an input saves the total ambient occlusion in the texture. This avoids the necessity of an extra step to add the two components.

SAAO shader:

It's composed by two shaders one for x and another for y. Both of them are similar to the SSAO shader where the occlusion is calculated. The difference is that we use only the x or y component of the random sample vector. We also have to add an operation that check if the direction of the vector used to calculate the ambient occlusion does not go inside the surface.

In the blur phase we use a Gaussian blur.[3] This Gaussian blur is also divided by x and y components. As the performance is effected by the number of times that we blur the texture we only blur on time horizontally and one vertically. We also need two framebuffers in this phase, but now we are not going to merge them in one texture, as we use both textures as input in the lighting phase.

[2] The implementation of the deferred shading used in our program is based on the implementation used in `https://learnopengl.com/Advanced-Lighting/Deferred-Shading`.

[3] The implementation of the Gaussian used in our program is based on the implementation used in `https://learnopengl.com/Advanced-Lighting/Bloom`.

SAAO Gaussian Blur shader:

It's composed by two shaders one for x and another for y. Both of them have the same functionality, but the x component blurs the texture horizontally and the y component blurs the texture vertically. It works as the blur shader, but now we have to add the Gaussian weights and keep in mind that we are blurring in just one direction.

The lighting phase is the same as before, changing that now we will have two input textures from the Gaussian blur phase.

3 Results and Comparison:

Here we are going to compare the two methods evaluating the visual appearance and performance. All the results have been obtained using a windows laptop with a NVIDIA GTX850M. The scene used for the tests is a model(cat) inside a cube.

Visual Appearance:

In the next three figures you can see the comparison between the two methods and the SSAO blurred texture:

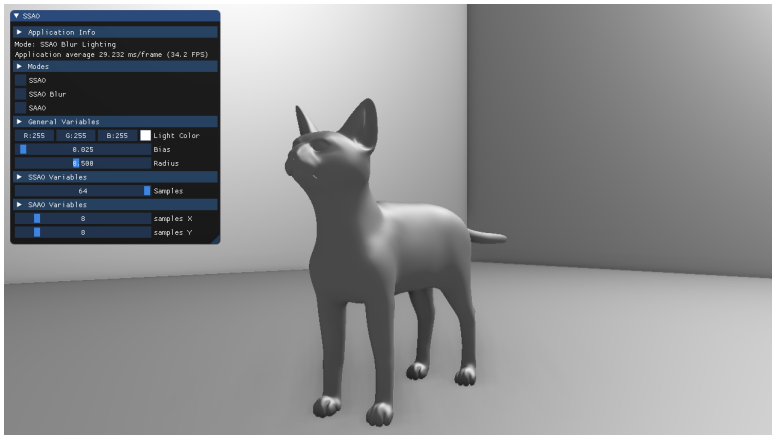

Figure 2: SSAO method.(34FPS)

As it can be seen the results of both methods are similar, but they are not exactly the same. It can be seen a darker effect in the legs of the cat in the SSAO and the corner of the cube, as well and the edges. In some places the SAAO effect is more noticeable as inside right ear or the left eye.

In the *Fig 3* we can see clearly the area of effect of the ambient occlusion. As said before in the scene it is clearly noticeable in the legs of the cat and the edges and border of the exterior cube.

Performance evaluation:

The performance of each technique mostly depends on the samples used for the ambient occlusion calculations. This parameter can be modified on-the-fly

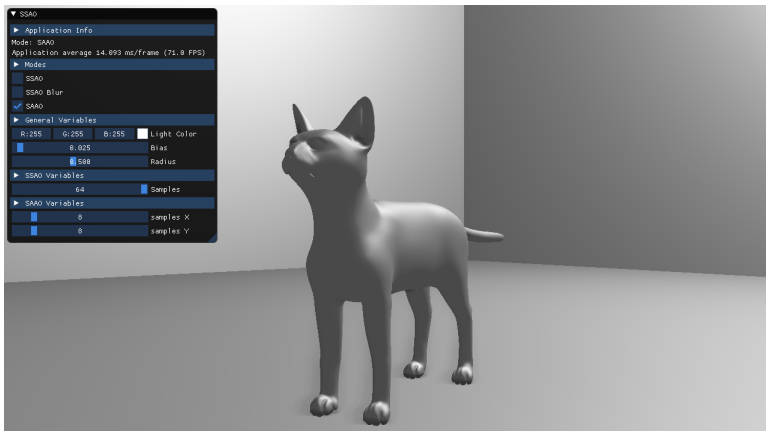

Figure 3: Separable Approximation of Ambient Occlusion method.(71FPS)

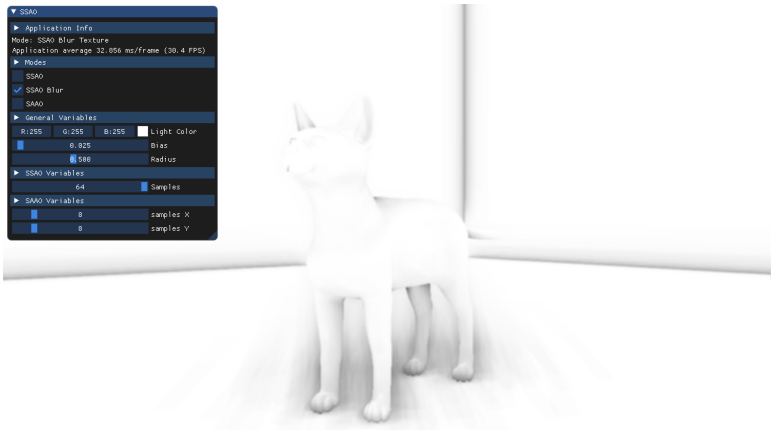

Figure 4: SSAO blurred texture.

in order to see the effect it has on the FPS. For the SSAO the we have configured a maximum number of samples of 64, while for the SAAO the maximum number of samples we have configured is 32 for each component.

In this section we are going to show some of the performance for different number of samples:

Samples	FPS
64	30
32	48
16	68

Table 1: Performance of the SSAO algorithm using different number of samples.

Samples	FPS
4x4	87
8x8	71
16x16	53

Table 2: Performance of the SAAO algorithm using different number of samples.

As we can see in the tables as more samples we use the more we reduce the performance, but it improves the quality of the visual result.

Using a few samples in the SAAO implementation we can get a similar result of the SSAO and get an improvement of the performance. As it can bee seen in the *Figure 3* and *Figure 4* the SSAA and SAAO visual results are really similar but we have an improvement of 2x for the SAAO performannce.

4 Conclusions

To conclude we can say that both methods give us a good approximation of the ambient occlusion effect. The SSAO method give us better visual results and a still a good performance. While in the SAAO method the visual results are not as good as the SSAO, but it is a really good approximation, and by using a few samples we can get high performance of the method and still good visual results.

For future work we think that for a better and conclusive results we will need to test both methods on a more complex scenes and compare the visual results and the performance. We think that in this kind of scenes the usage of SAAO will be better suited in terms of performance.