REAL-TIME APPROXIMATED GLOBAL ILLUMINATION FROM SSAO TO SSDO

INF584 Images de synthèse : théorie et pratique

18 février 2016

Chia-Man Hung

1 Introduction

The demand for realistic real-time rendering in games has led to the creation of GPU shaders that approximate global illumination. One such example is Screen-Space Ambient Occlusion (SSAO), an approximation of the well-known method Ambient Occlusion (AO) in screen-space. This project is an implementation of another approximation known as Screen-Space Directional Occlusion (SSDO), based on *Approximating Dynamic Global Illumination in Image Space*, Tobias Ritschel et al. [2009]. SSDO solves the problem of AO or SSAO decoupling visibility and illumination information by accounting for the direction of the incoming light. Results are mainly showed by pictures.

2 Methods

2.1 Screen-space ambient occlusion

Ambient occlusion describes the shadow which is not created by direct lighting, but by a virtual diffused surrounding light emitter. One of the main effect of ambiant occlusion is that volumes, concavities and contact areas are far better perceived using it than with direct lighting only. Ambient occlusion techniques are expensive as they have to take surrounding geometry into account. One could shoot a large number of rays for each point in space to determine its amount of occlusion, but that quickly becomes computationally infeasible for real-time solutions.

In 2007 Crytek published a technique called screen-space ambient occlusion (SSAO) for use in their title Crysis. The technique uses a scene's depth in screen-space and generates samples in a normal-oriented hemisphere (Fig. 1) to determine the amount of occlusion instead of real geometrical data. This approach is incredibly fast compared to real ambient occlusion and gives plausible results, making it the de-facto standard for approximating real-time ambient occlusion.

More precisely, when loading the 3D model considered, it computes a g-buffer to store its position, depth and normal values in screen-space. Then, for every pixel of the screen, it gets its position, depth and normal values. By knowing the position and the normal of the considered pixel, we can randomly generate a certain number of samples in the corresponding normal-oriented

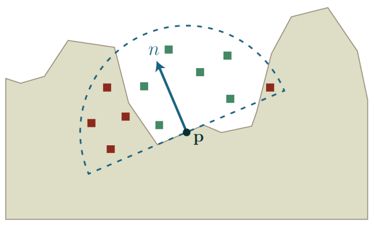

Figure 1 – Approximation of AO [Filion and McNaughton 2008]. The green and red rectangles are the visible and occluded samples, respectively.

hemisphere and count the number of occluded samples to evaluate the ambient occlusion value. We may add a step to blur the ambient occlusion value here to attenuate the effect of random samples. At last, we multiply every pixel color computed beforehand by its ambient occlusion value.

Compared to other the original ambient occlusion, SSAO has the following advantages : (from wikipedia)

- Independent from scene complexity.
- No data pre-processing needed, no loading time and no memory allocations in system memory.
- Works with dynamic scenes.
- Works in the same consistent way for every pixel on the screen.
- No CPU usage – it can be executed completely on the GPU.
- May be easily integrated into any modern graphics pipeline.

Of course it has its disadvantages as well :

- Rather local and in many cases view-dependent, as it is dependent on adjacent texel depths which may be generated by any geometry whatsoever.
- Hard to correctly smooth/blur out the noise without interfering with depth discontinuities, such as object edges (the occlusion should not "bleed" onto objects).

Just as AO, SSAO decouples visibility and illumination information by simply multiplying the previous illumination value by its ambient occlusion value.

2.2 Screen-space directional occlusion

In 2009, Tobias Ritschel published another technique called screen-space directional occlusion (SSDO), which is a derivative of screen-space ambient occlusion method, introduced to improve its visual quality. It works by calculating occlusion and environmental lighting at the same time.

Similar to SSAO, it starts by computing a g-buffer. For each pixel of the screen, it generates a certain number of samples randomly in the corresponding normal-oriented hemisphere. When a sample is not occluded, we accumulate the directonal lighting of the environment in memory. A one-boune indirect lighting is also approximated in screen-space.

In Fig. 2, among the four samples, A, B and D are classified as occluders. We add the color of the environment map from C to P to the directional lighting memory as shown on the left side. On the right side, to compute one indirect bounce of light, for each occluder, we check if P is in its positive half-space.

Compared to SSAO, SSDO has the advantage of seperating visibility and illumination information. Two blur passes are required to remove the noise : one for the directional lighting and the other for indirect lighting. Thus, it costs more computational time and memories.

Since this method is applied in screen-space, not every source of indirect light is taken into accout. Tobias Ritschel proposed *Depth Peeling* and *Additional Cameras* to overcome the limitations. However, this part is not included in my implementation.

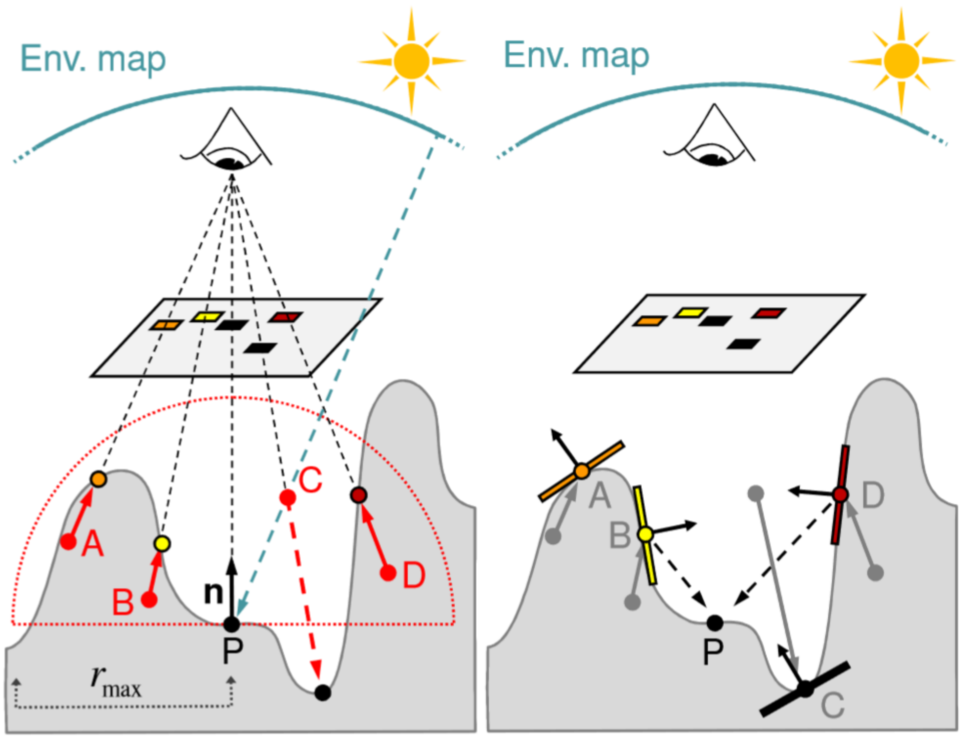

FIGURE 2 – Left : direct lighting with directional occlusion/ Right : one-bounce indirect lighting.

3 IMPLEMENTATION

3.1 ENVIRONMENT

The main codes are written in C++ and the shaders in OpenGL Shading Language (GLSL). The codes are based on the OpenGL tutorial site LearnOpenGL and part of its codes are includes, espcially for loading the model, reading files and creating the camera. In addition, the libraries OpenGL Extension Wrangler Library (GLEW), GL Frame Work (GLFW), GL Mathematics (GLM) and Simple OpenGL Image Loader (SOIL) are used.

3.2 MAIN FUNCTION

In the main codes, constants are defined. Our scene is lit by a point light and an environment cube map (skybox). The window is set up and displayed. The memories are properly linked.

Samples are generated. Then, we pass from the first pass to the last one. Outputs are stored correctly in their corresponding memories and sometimes used by later passes. The output of the last pass is displayed on the window.

3.3 Different passses

In total, eight passes are used and will be listed in the following.

1. Geometry pass

This pass loads the model and stores a per-fragment position texture, a linear depth texture, a per-fragment normal texture and a constant albedo texture in a g-buffer.

2. SSDO pass

This pass takes the position, the depth and the normal texture of our g-buffer and the environment cube map texture as input and computes the directional occlusion.

3. SSDO blur pass

This pass takes the output of SSDO pass as input and blurs every per-fragment value with its neighbors. The output is stored apart.

4. SSDO lighting pass

This pass takes all the textures of our g-buffer and the output of SSDO blur pass as input and computes the sum of the global illumimation (diffuse : albedo, specular : Blinn-Phong) of a point light and the directional light generated by the environment cube map.

5. SSDO indirect lighting pass

This pass takes the position, the depth and the normal texture of our g-buffer as input and computes a indirect one-bounce lighting.

6. SSDO indirect lighting blur pass

Just as what SSDO blur pass does, but instead of taking the output of SSDO pass as input, this pass takes that of SSDO indirect lighting pass.

7. Accumulate lighting pass

This pass takes the position, the depth and the normal texture of our g-buffer as well as the outputs from pass 2 to 6 as input and computes the sum of different effects. Based on which of the 1 - 9, 0, P, O keys we pressed (draw mode 1 - 12), specific buffer values are showed.

8. Sybox pass

This pass takes the output of Accumlate lighting pass and the environment cube map texture as input. For draw mode 1 - 5, if the depth value is inferior to 0.009 (this value is superior to the default float value stored in a buffer and inferior to the minimum depth value really put in the g-buffer), the color environment cube map is set as output. Otherwise, it is the output of the prvious pass.

3.4 Possible improvements

Our scene is lit by an environment cube map and a point light. However, only the point light contributes a specular term to the global illumination. We could have added a specular term from the environment cube map by adding another two passes after SSDO blur pass. It would require generating samples to act as incoming light in the rendering equation. The previous computation would be done in the first added pass and as usual, we would need to blur the texture to remove the noise. This would slow down our program a little. Since the environment cube map is rather homogeneous, the specular term might not have great effect. Above are the reasons why we chose not to do so.

The illumination of the point light does not take any ambient occlusion method into consideration (we could have added one although it is not realistic). Neither does it have a shadow. We could have computed a shadow map for the point light and check on every pixel to see if it is in the shadow.

In the last pass, experiments show that the default value of a float stored in a buffer is between 0.098 and 0.099. The condition of checking whether we should display the color of the environment cube map is "if depth $<$ 0.099". If we can set default value at 0 or infinity, we may change this condition to "if depth == 0" of "if depth == infinity". Note that this does not influence our results since the depth values stored in the g-buffer are reranged from 0.1 to 50. We can even change our condition to "if depth $<$ 0.1". (The default value of the last coordinate of a vec4 color buffer is 1. That is why we created another float buffer to store the depth.)

We may change the specular term from Blinn-Phong to other models, such as GGX or Beckmann learned in lectures. We may try other models, other environment cube maps or other position and color of point lights.

4 RESULTS

4.1 Main results

We successfully obtained some nice results of a nanosuit lying on the floor. Below are some pictures of different draw modes. (Fig. 3, 4, 5, 6.) Draw mode 5 is our final result (Fig. 4 Middle).

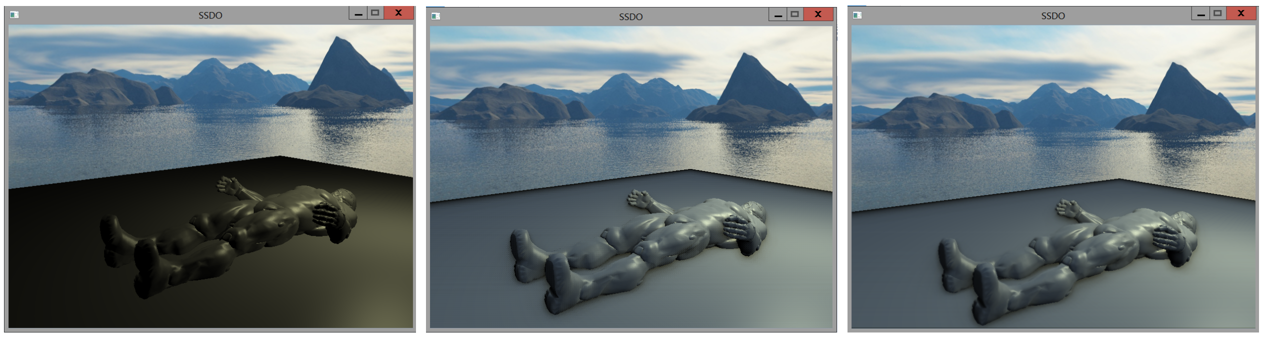

Figure 3 – Left : direct lighting of a point light/ Middle : point light + unblurred direct directional occlusion of the environment cube map/ Right : point light + blurred direct directional occlusion.

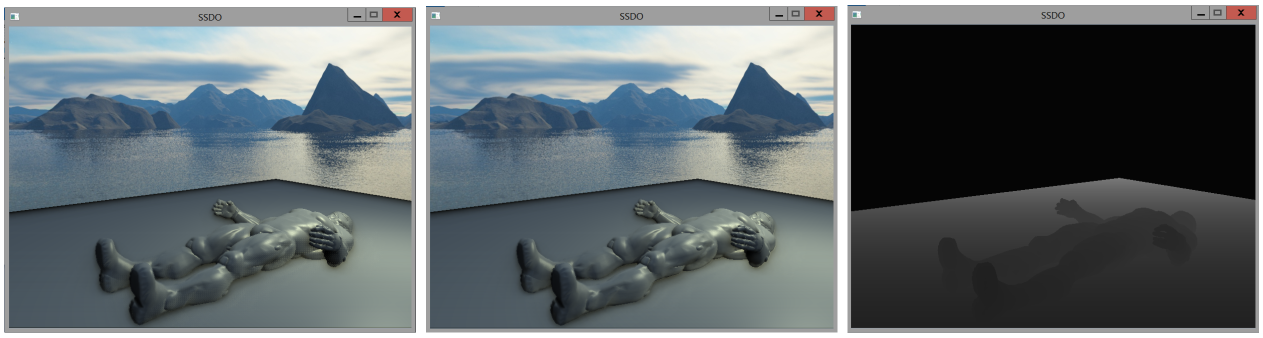

Figure 4 – Left : point light + blurred SSDO + unblurred indirect lighting/ Middle : point light + blurred SSDO + blurred indirect lighting/ Right : depth texture.

4.2 Deficiency

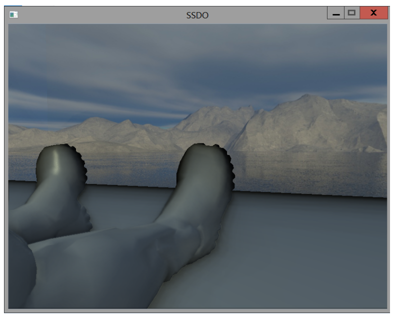

Figure 7 – The shoe borders neighboring the cube map are darker.

Borders of the floor block as well as the pixels of the nanosuit neighboring the environment cube map are darker. Due to the near-zero default value of a float buffer, the cube map is considered closer to the camera than the model. Thus when computing the SSDO, samples on the cube map is classified as occluders. In Fig. 7, the upper borders of the shoes are much darker than the lower part.

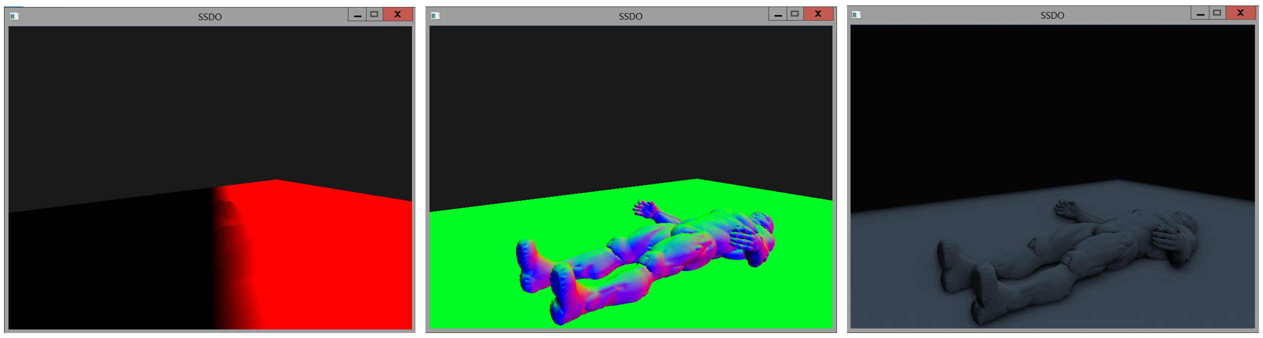

Figure 5 – Left : position texture/ Middle : normal texture/ Right : unblurred SSDO.

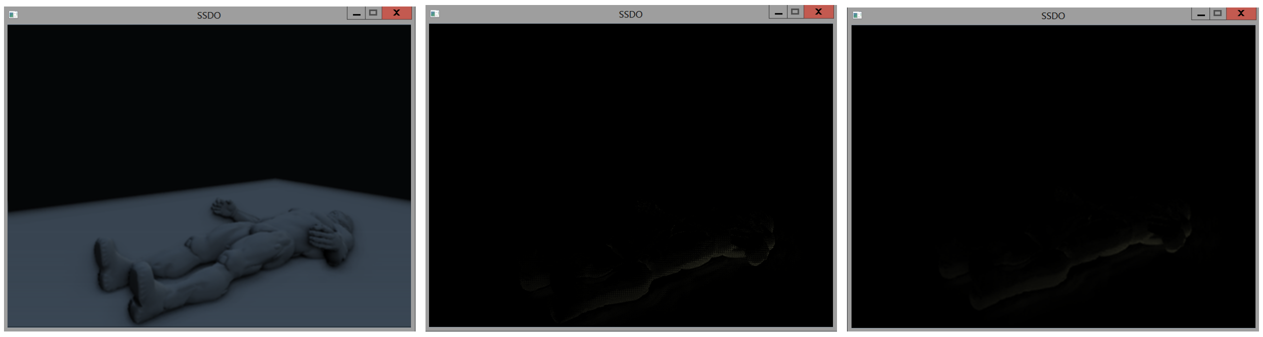

Figure 6 – Left : blurred SSDO/ Middle : unblurred indirect lighting/ Right : blurred indirect lighting.

5 CONCLUSION

We implemented two generalizations of screen-space ambient occlusion that add directional occlusion and diffuse indirect bounces by using eight shader passes, which is three more (pass 5, 6, 7) compared to screen-space ambient occlusion. The specular term of the environment cube map could be added but this costs more computation time, as discussed in Section 3.4. The problem of setting the default float value of a buffer to what we desire is to overcome. Besides these aspects, results seem to be satisfying.

6 REFERENCES

T Ritschel, T Grosch, HP Seidel. 2009. Approximating dynamic global illumination in image space. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, 75-82.