INF584 - Screen Space Directional Occlusion

Matthias Hasler, final project
Approximating Dynamic Global Illumination in Image Space [T. Ritschel, 2009][1]

Introduction

Up to this day, real-time graphics have been dominated by rasterization, where a lot of approximations have to be made. These approximations capture some of the complex processes occurring in nature, but make rasterized renderings seem “off”.

Ambient illumination is key for creating photorealistic renderings. It consists in computing the lighting that surface receives not only as any direct light it might be subject to, but also as the rays of lights that have bounced and re-bounced on other surfaces in the scene, creating indirect illumination. As they lose energy at each bounce, less light can get to corners and other occluded surfaces. Ambient occlusion computes the shadows that would be found in a scene with ambient light: it helps create realistic-looking indirect shadows on surfaces where nearby geometry blocks some of the incoming light. With SSAO, the most common technique at that time, shadows looked dull. The occlusion factor was computed from the geometry, neglecting directional information about the light source.

The proposed technique couples the sampling of the light with occlusion tests. This is achieved with low overhead over SSAO, by making computations in screen space.

Paper review

An earlier method SSAO (Screen Space Ambient Occlusion) moderately achieved the desired effect, by computing at each pixel an occlusion factor to dim it. The term “screen space” refers to the fact that this technique only needs to look at pixel values. This is very practical as it allows us to do all the computations on the GPU - a major advantage of this technique.

The method begins by taking the depth and the normals at each pixel, and using them to sample points on a hemisphere tangent to the surface appearing on that pixel. Points of this hemisphere that appear closer in screen space than the sample point contribute to computing the occlusion factor.

The major shortcoming of SSAO is that there is no directional information about occlusion, so in practice people would use it in conjunction with static illumination techniques like baking lights into textures. The innovation of SSDO is the coupling of the sampling of light with the occlusion, while still making all computations in screen space to keep computing costs low.

[1] https://people.mpi-inf.mpg.de/~ritschel/Papers/SSDO.pdf

The formula used in SSDO closely resembles the discrete version of the rendering equation, more common in raytraced renderers. It is a weighted convolution of the visibility function V and the incoming light L_{in} over the sampling directions w_i.

$$L_{\mathrm{dir}}(\mathbf{P}) = \sum_{i=1}^{N} \frac{\rho}{\pi} L_{\mathrm{in}}(\omega_i) V(\omega_i) \cos\theta_i \Delta\omega.$$

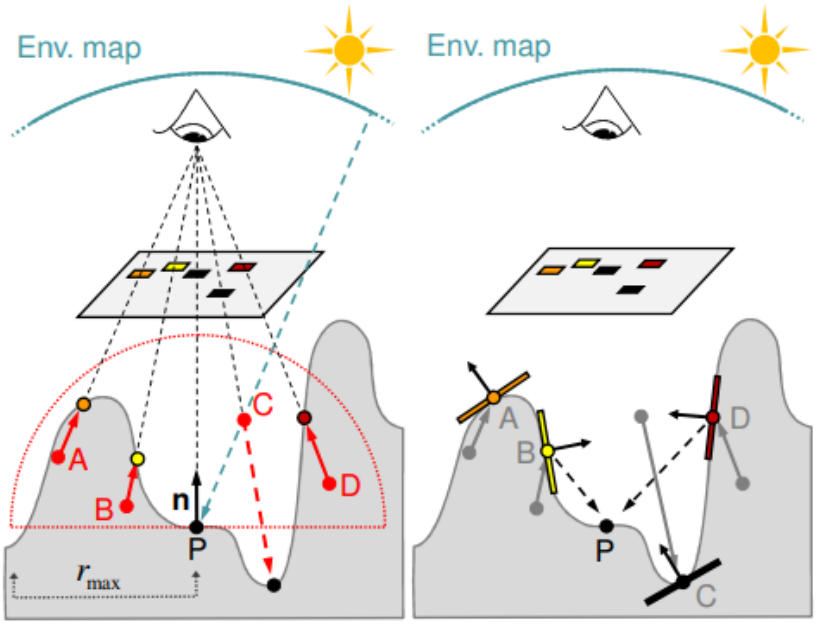

Figure 1: We want to compute the light coming from point P. We consider the sample points A,B,C,D in a hemisphere centered at P and oriented by the normal. Here, only C is not occluded, so we add the contribution of the direct light coming from the direction PC. Among the occluded samples, once they are projected to the nearest surface in screen space, B and D are facing towards P, and so their color contributes this pixel's value.

This barebone version of SSDO is likely to make some mistakes:
A point can be wrongly classified as occluder if something stands between the camera and the sample. And similarly, the occluder in a direction might be missed if the sample point we take is not at the right distance from the fragment position.
Moreover, as we sample in screen space, we are not necessarily finding the closest occluder and this can cause surfaces to bounce off light only from a certain range of angles.

I did not implement what follows, but it is quite interesting. The following techniques would make more sense in a more complex scene, where we can see the shortcomings of the default technique, and work with shadows casted from objects further away.

To address some issues of the presented method, the authors suggest depth peeling, a process in which we compute not only the depth of the closest geometry, but also the subsequent layers.

Then it checks if a sample point is occluded; assuming we are dealing with closed 2D surfaces, we simply check the evenness of the number of layers above it.
The original implementation uses 2 depth buffers for peeling[2].

Another suggested technique consists in taking multiple cameras and checking for occlusion in each of their screen spaces. We compute position and color buffers for each point of view. Then, for each point in screen space, we perform a transform to find the corresponding one in the other view space and check its occlusion there. This technique allows for surfaces hidden from the main camera to cast a colored shadow.

Finally, they discussed integrations in global illumination:

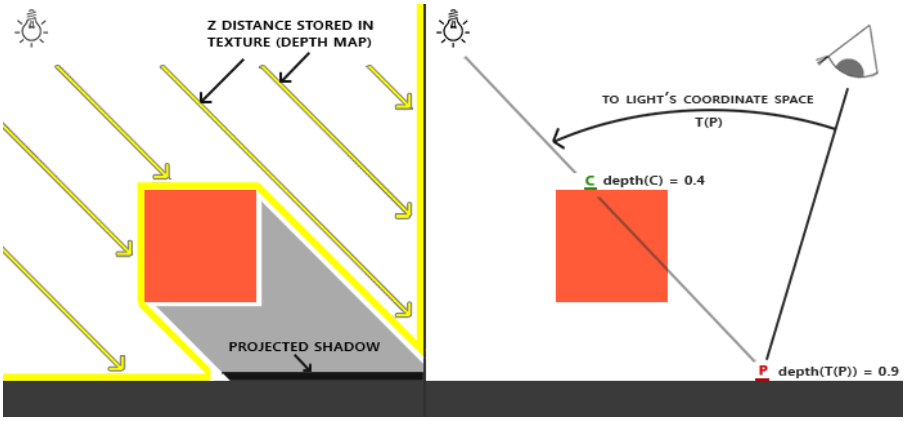

Figure 3: shadow mapping principle

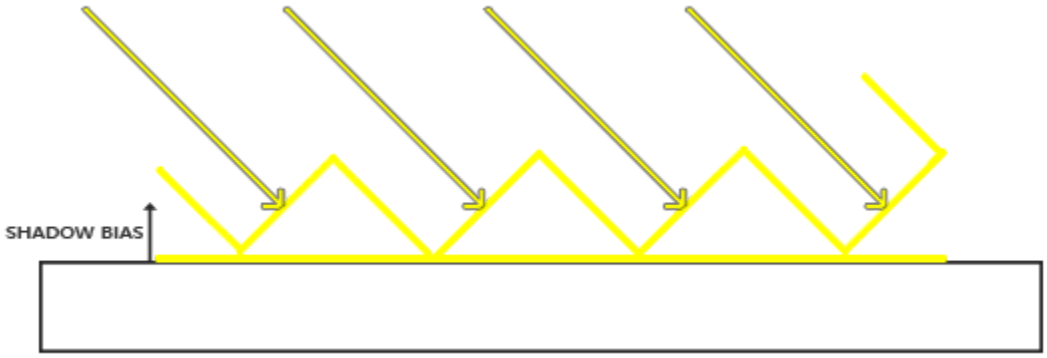

Figure 4: A shadow bias is necessary to avoid intersections.

Shadow mapping principle: From the sun's point of view, render a depth map. Then, to render in screen space, convert the sampling coordinates into the light's coordinate space and sample from the texture to find out if the point is on the first surface hit by the light ray.

Fine shadow maps require a high resolution, which is costly, and cannot be exact because of the finite resolution of the buffer. Also, to avoid fringing effects, a bias is added to make sure the

[2] http://developer.download.nvidia.com/assets/gamedev/docs/OrderIndependentTransparency.pdf

entire shadow is behind the surface. This makes it impossible for small objects to have a shadow and causes the borders of shadows to be approximated.
In the paper, the authors present a method of refining these coarse shadowmaps, in screen space, by sampling in the direction of the lightsource over a short distance. This is useful for contact shadows.

Implementation

The implementation is based on the course's BaseGL codebase
and is inspired by LearnOpenGL articles on SSAO[3] and Cubemaps[4].
It includes the STB library[5] for loading images, and the OpenGL libraries glad, GLFW and GLM.

Each pass is rendered to a framebuffer. The only pass dealing with the geometry of the scene is the geometry pass. Others simply reuse outputs of the buffers of previous passes.
In the end, they are blended by summing the contribution of different shaders and outputted to the default framebuffer for rendering. To choose between rendering the skybox and the model, we check the depth buffer value.

The program uses the following 8 shaders:

- Geometry pass: compute positions, normal and depth of the displayed fragments
- Lighting pass: Phong shading using a point light source
- SSDO Direct: compute ambient light from the skybox with the SSDO algorithm
- SSDO Direct Blur: smoothing kernel on the previous buffer
- SSDO Indirect: compute ambient light bouncing off occluding surfaces
- SSDO Indirect Blur
- Skybox: rendering of the inside of a cube centered at the origin, using the view matrix without its translation component.
- Mixer: check the depth buffer to pick between skybox and geometry, and blend the appropriate buffers (additive mixing).

Both blur shaders are running the same code and simply convolve the previous image with a 4 by 4 kernel. The other shaders use the output of the geometry pass. The indirect lighting also takes in the Lighting pass buffer to consider the color of occluding surfaces. The SSDO direct and the skybox shaders use cubemaps to compute coordinates of the skybox texture.

We added a *mode* enum to the mixer shader to pick between different buffers to render.
The OpenGL business logic is partly handled by the Mesh and ShaderProgram class, but the buffer manipulations were done using the OpenGL API, since apart from the initialization, which is a bit verbose, there is not too much to abstract in the main loop.

---

[3] https://learnopengl.com/Advanced-Lighting/SSAO
[4] https://learnopengl.com/Advanced-OpenGL/Cubemaps
[5] https://github.com/nothings/stb/blob/master/stb_image.h

Results

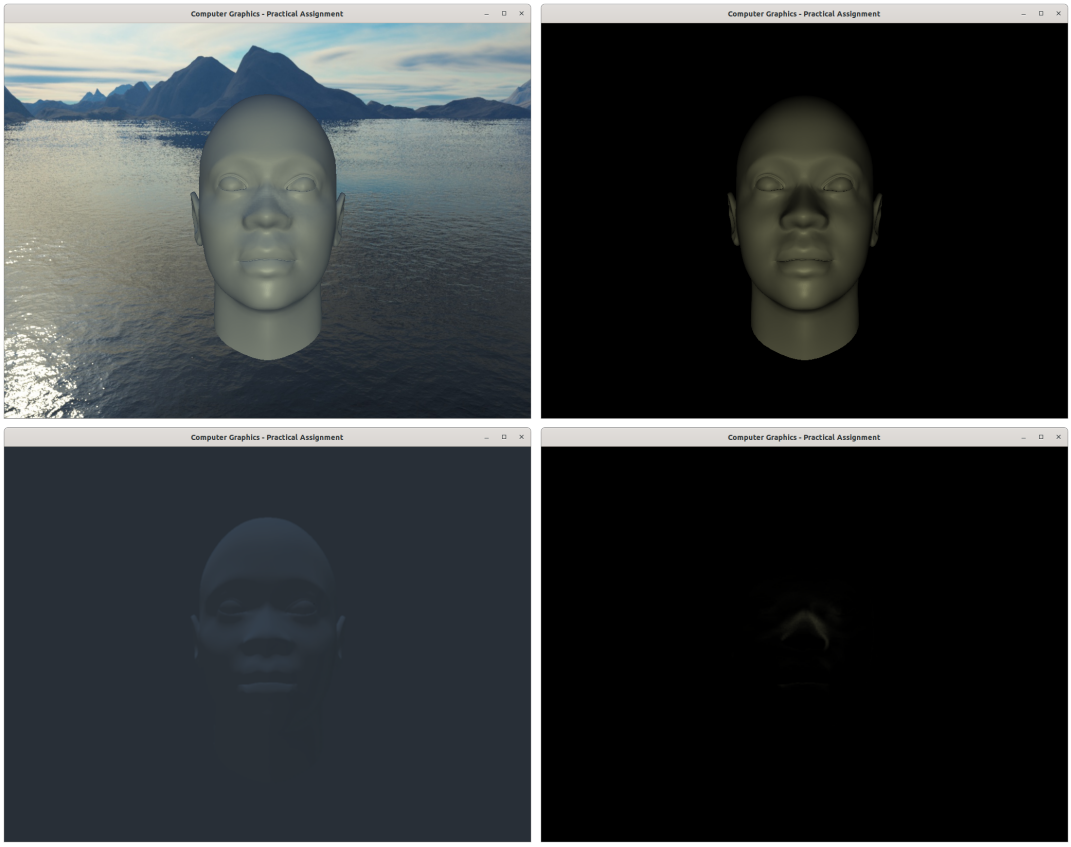

Figure 4: The screenshots here show respectively, the final image, the lighting pass, the direct SSDO and the indirect SSDO. The lighting pass shows features characteristic of the phong model, we notice the specular light on the chin and the tip of the nose. The direct SSDO lighting takes contribution from the skybox, where the sky is brighter than the sea. The indirect SSDO highlights a part of the nose, which reflects the light hitting the eyes of the face mesh.

I didn't perform extensive testing, but the performance seems good, since the program runs on my laptop, with no GPU, at a decent framerate.

Conclusion

We have seen how SSDO couples the occlusion with the light sources, to create more realistic shadows, and a few techniques that help overcome some of the shortcoming of the simplest implementation. All this is achieved with a low overhead, by working in screen space.
This paper touched many techniques used in the modern graphics pipeline. It showed that we can build on the elegant ideas behind screen space processing and shadow mapping, and how these techniques can be combined to achieve more detailed and realistic shadows.

For me, this project was a good opportunity to learn about and practice with FBOs, and better understand the inner workings OpenGL.