Approximating Dynamic Global Illumination in Image Space: Project Report

Putian YUAN
putian.yuan@ip-paris.fr

1 Paper Summary

In order to implement this project, I have studied two technique: real-time Screen-Space Ambient Occlusion($SSAO^{[3]}$) and real-time Screen-Space Directional Occlusion($SSDO^{[1]}$).

1.1 SSAO

SSAO proposes a simple-fast way to compute ambient occlusion for each pixel by using its neighbor pixels in screen-space, then using this occlusion to compute final color:

$$fColor = (1 - occlusion) * colorResponse \tag{1}$$

where $fColor$ is pixel color output to screen and $colorResponse$ is approximated radiant at this pixel. As figure 1 shows, considering now we have a 3D position $\boldsymbol{P}$ on a plane, all incoming lights(yellow line in figure 1) over the hemisphere are contributing some radiant to $\boldsymbol{P}$, while sphere $\boldsymbol{C}$ occludes part of them. SSAO refers to sphere $\boldsymbol{C}$ as an occluder, and the occlusion of $\boldsymbol{P}$ is the sphere cap area or solid angle quantity (gray part in figure 1) in unit hemisphere $\boldsymbol{P}$. The formula for this part is

$$A_{\Psi}(\boldsymbol{C}, r, \boldsymbol{P}, \vec{n}) = S_{\Omega}(\boldsymbol{P}, \boldsymbol{C}, r) * max(\vec{n}, \vec{\boldsymbol{PC}}, 0)$$

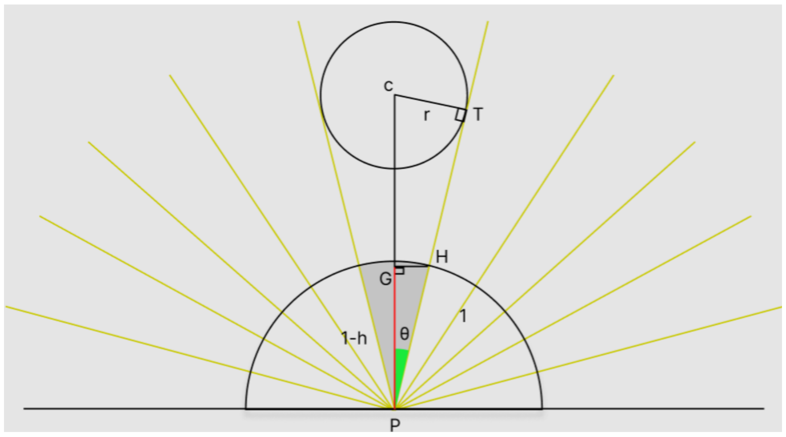


Figure 1: Ambient occlusion due to a single sphere at a point on a plane.

where A_Ψ is ambient occlusion equation, and S_Ω represents the surface area of spherical cap subtended by the sphere $\boldsymbol{C}$. And,

$$S_\Omega = 2 * \pi * (1 - cos(sin^{-1}(\frac{r}{|\boldsymbol{PC}|})))$$

Following this idea, SSAO samples a lot of neighbor pixels at $\boldsymbol{P}$, and marks those points

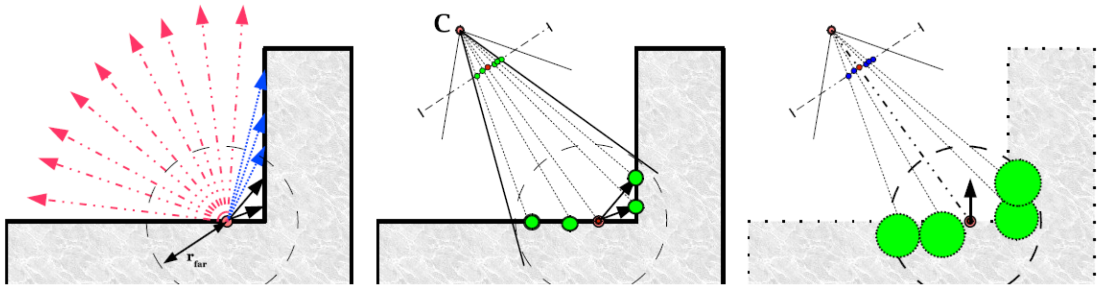


Figure 2: SSAO - Finding occluders

inside sphere with $radius = r_{far}$(user-define) as occluders, shown in figure 2(left). However, this method will find some occluders on flat plane, which are impossible to occldue point $\boldsymbol{P}$, shown in 2(middle, two green points on left side of $\boldsymbol{P}$). SSAO uses a trick, which is to move all occluders along with view direction(from viewpoint to occluder) a little bit, to avoid this issue, shown in figure 2(right).

For each pixel $\boldsymbol{P}$, SSAO uses the projected radius $r_{projected}$ to find its neighbor pixels randomly, with condition $d <= r_{projected}$ in image space, where d is difference of pixels' depth in image space, as figure 3 shown. After found these occluders, SSAO uses the equations mentioned above to compute $occlusion$ and $fColor$.

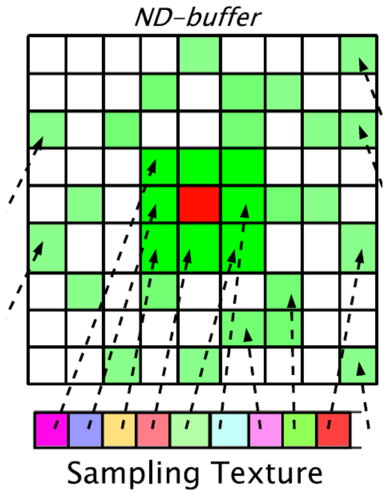


Figure 3: Sampling in Screen-Space

1.2 SSDO

SSAO is simple and efficient, which provides good enough visual result. However, it decouples visibility(opposite to occlusion) and illumination, ignores all *directional* information of the incoming light. In order to take into account these *directional* information, SSDO extends the implementation based on SSAO, to approximate two significant effects: *directional occlusion*(DO) and *indirect bounces* in real-time. Moreover, DO can be used to correct *Peter Panning* issue while applying *Shadow Mapping* with *bias*.

1.2.1 Direct Lighting using DO

As figure 4(left) shown, SSDO samples points randomly located in a hemisphere $\boldsymbol{P}$ with radius r_{max}, to compute the approximated radiant at $\boldsymbol{P}$ from all incoming directions: for every pixel at 3D position $\boldsymbol{P}$ with normal $\boldsymbol{n}$, the direct radiance L_{dir} is computed from N sampling directions ω, uniformly distributed over the hemisphere, each covering a solid angle of $\triangle\omega = 2\pi/N$:

$$L_{dir}(P) = \sum_{i=1}^{N} \frac{\rho}{\pi} L_{in}(\omega_i) V(\omega_i) \cos\theta_i \triangle\omega. \quad (2)$$

where Lin can be computed from point lights or an environment map, $V(\omega_i)$ is visibility which determine where ω_i direction is visible. If it is not visible, then we could say it contributes an occlusion to $\boldsymbol{P}$. θ_i is the angle between $\boldsymbol{n}$ and ω_i. For each sampling direction ω_i, SSDO takes a step of random length $\lambda_i \in [0...r_{max}]$ from $\boldsymbol{P}$ in direction ω_i, where r_{max} is a user-define radius. This results in a set of sampling points $\boldsymbol{P} + \lambda_i\omega_i$, located in hemisphere $\boldsymbol{P}$. Next,

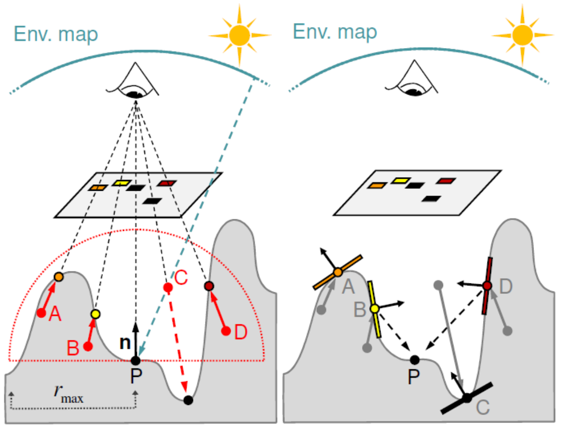


Figure 4: SSDO - Finding Occluders

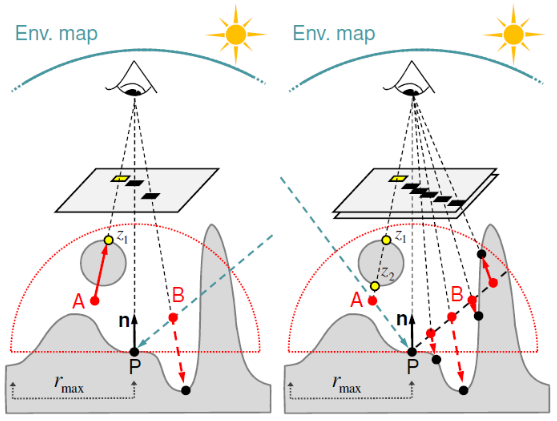


Figure 5: SSDO - Misclassification and Solution

SSDO projects these sampling points into screen-space, which I refers to these projected points as *surface* points or *front* points. If a sampling point is below the corresponding surface point in view direction, which means the *depth* of a sampling point is ***deeper*** than its surface point, SSDO will refer to this sampling point as an *occluder*, and the corresponding ω_i becomes invisible. For example, in figure 4(left), ***A*** is the sampling point and it is below the surface. Therefore, A becomes an occluder and the sampling direction $\vec{PA}$ is invisible, while ***C*** is above surface and classified as *visible*.

1.2.2 Depth Peeling and Multiple Sampling

However, not all sampling points have been classified correctly. As figure 5(left), by using the method described above, ***A*** is classified as an occluder but it is actually not, and ***B*** is classified

as not an occluder but its corresponding direction ω_i is actually invisible. SSDO proposes *depth peeling* solution(figure 5 right) to fix the issue on $\boldsymbol{A}$, which uses the extra depth from back-face. $\boldsymbol{A}$ will be classified as an occluder if and only if its depth is between the depth range of front-face and back-face. Sampling more points along with ω_i are proposed to fix the issue on $\boldsymbol{B}$(figure 5 right).
Those method are based on the surface points of sampling points. However, not all sampling points have its corresponding surface points due to sampling point can be outside of view frustum. The *additional camera* method proposed by SSDO, which sampling points on different view position/direction, can be used to fix this issue, but it is not implemented in my codes.

1.2.3 Indirect Bounces

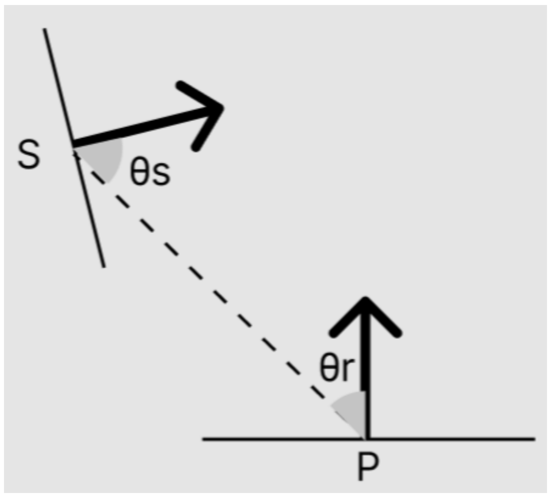
Considering point $\boldsymbol{P}$ are receiving directional light from $\boldsymbol{N}$ sampling direction ω_i, visible ω_i contributes directional light radiant, while invisible ω_i contributes indirect light(bounce light). Those invisible ω_i can be used to find surface points of point $\boldsymbol{P}$. SSDO uses the pixel color of surface point to compute bounce light lighting on $\boldsymbol{P}$, as figure 4(right) shown. The indirect radiant at $\boldsymbol{P}$ can be approximated as:

$$L_{ind}(\boldsymbol{P}) = \sum_{i=1}^{N} \frac{\rho}{\pi} L_{pixel}(1 \ - \ V(\omega_i)) \frac{A_s \cos\theta_{s_i} \cos\theta_{r_i}}{d_i^2} \tag{3}$$

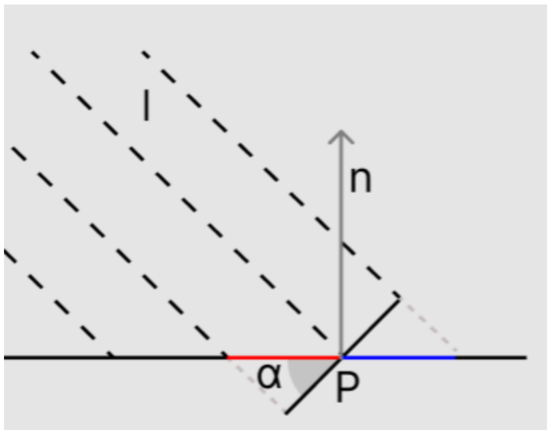
where L_{pixel} is the L_{dir} of a surface point in invisible ω_i direction, and $V(\omega_i)$ is visibility which determine where ω_i direction can be visible. A_s can be a constant value changed by user. d_i is the distance from surface point to $\boldsymbol{P}$, clamped in $[0,\ 1]$. θ_s and θ_r are shown in figure 6(a), $\boldsymbol{S}$ is a surface point as sender sending bounce light to a receiver $\boldsymbol{P}$.

1.2.4 Peter Panning Correction using DO and Multiple sampling

When applying *Shadow Mapping* in our scene, it will bring *Shadow Acne* issue due to angle α between light and plane. As figure 6(b) shown, when incoming light l hits on point $\boldsymbol{P}$ with normal n, those pixels on *blue* line part will be classified in shadow while pixels on *red* line will be not. While using bias to solve *Shadow Acne* issue, it brings another issue: *Peter Panning* as figure 7 shown, those shadows near to object will disappear. SSDO proposes to use DO for shadow correction, as figure 8 shown: sampling $\boldsymbol{K}$ points($\boldsymbol{K}$-sampling) between $\boldsymbol{P}$ and $\boldsymbol{P}\ +\ b \cdot\ l$ uniformly, where b is bias. Next, use the same occlusion test method as described above to evaluate the visibility of sampling point. If one occluder found during the $\boldsymbol{K}$-sampling iteration, $\boldsymbol{P}$ can be considered in shadow.



(a) SSDO - Sender and Receiver



(b) Sampling in Light Space

Figure 6

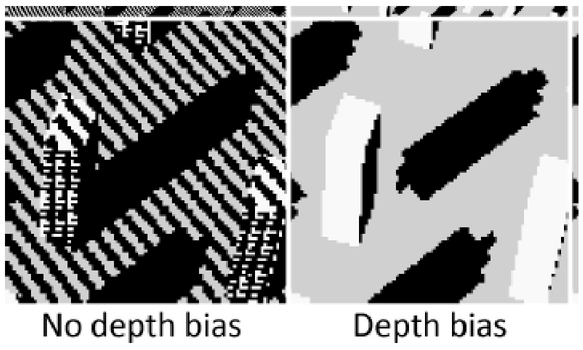


Figure 7: left: Shadow Acne, right: Peter Panning

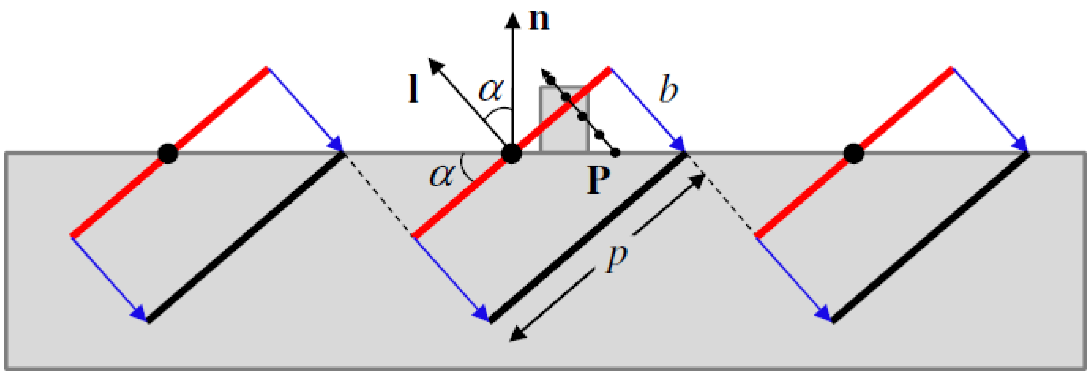


Figure 8: Shadow Correction using DO

2 Implementation

This project is based on the code-base from the first assignment, which has been improved after each assignment session.

2.1 Main Procedure

In my project, there are **11** pass and **1** precomputation:

1. G-buffer pass

   All objects are rendered in scene once, and store their *position*, *normal* and *material* information into different textures for later use.

2. DP pass

   All objects are rendered in scene again, but with culling front face setting, to store their back-face *depth* information into depth texture for later use in *Depth Peeling* method.

3. Random Sampling Direction Computation

   A $Texture2D$ array is created, where $width$ and $height$ are equal to $Window\ Size$, $layer$ is equal to user-define sampling number, and get filled with $\lambda_i \omega_i$ as described above.

4. DO pass

   L_{dir} will be computed by using the equation 2. *Directional occlusion* will also be computed for verification in test rendering.

5. Blur DO pass

   L_{dir} and DO from last pass will be blurred by Gaussian Filter.

6. Lighting pass

   Specular term $L_{specular}$ will be computed based on micro-facet model. Also, PBR color will be computed for final comparison in test rendering.

7. Bounce pass

   L_{lind} will be computed by using the equation 3.

8. Blur Bounce pass

   L_{lind} will be blurred by using the same filter as described in Blur DO Pass.

9. Shadow Mapping pass

   Shadow Map will be created according the active light in scene.

10. Shadow Result pass

    At each pixel, shadow result from each active light will be computed by using *Shadow Map* from previous pass. *Depth bias*, *DO* and *Multiple Sampling* will be used together to correct *Shadow Acne* and *Peter Panning* issues.

11. Blur Shadow Result pass

    Shadow Result will be blurred by using the same filter.

12. Accumulate Result pass

    Color response $fColor$ at pixel $\boldsymbol{P}$ will be computed by using:

$$fColor \ = \ L_{ind} \ + \ max(0, \ min(1, \ 1 \ - \ shadow)) \ * \ (L_{dir} \ + \ L_{specular}) \qquad (4)$$

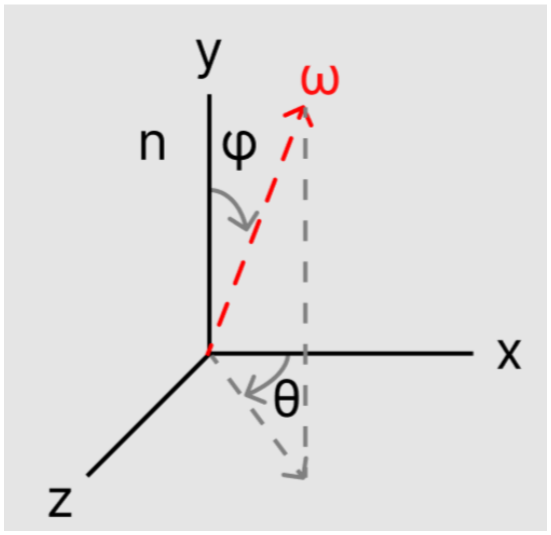
2.2 Implementation Details

1. The alpha value of *Albedo* texture is used to determine whether the pixel contains information of *position* in fragment shader.

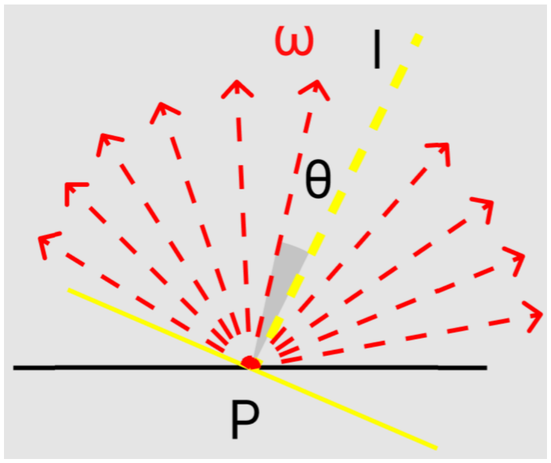
2. As figure 9(a) sampling directions are computed following this way: For each point P with normal n, ϕ is from 0 to $\frac{\pi}{2}$, θ is from 0 to 2π, and y axis is *normal*. There are two user-defined variable *samplingPhi* and *samplingTheta* in code determine final sampling number N. Each sampling direction will be ***rotated*** according to *normal* before using it to sample neighbor points at P.

3. When sampling points around point P, there will be some points outside of view frustum. In this case, these points will be treated as visible to point P.

4. In L_{dir} computation, L_i takes into account distance between light and point P, it also considers *angle* between sampling direction ω_i and light direction l. As figure 9(b) shown, when θ between ω_i and l becomes 0, point P receives the most radiant. An absorb coefficient, where $absorb \ = \ \max(0, \omega_i \ \cdot \ l)$, is applied in L_{dir} computation.



(a) Sampling Direction ω



(b) Sampling Direction and Light Direction

Figure 9

5. In *Shadow Result* computation, a *bias* is computed following $bias = \max(bias_{min}, patchSize * 0.5 * \tan\alpha)$, and $patchSize = \max(\frac{1}{texSize.x}, \frac{1}{texSize.y})$, where *texSize* is the size of *ShadowMap* texture, α is the angle between light and normal at point P, and $bias_{min}$ is user-defined.

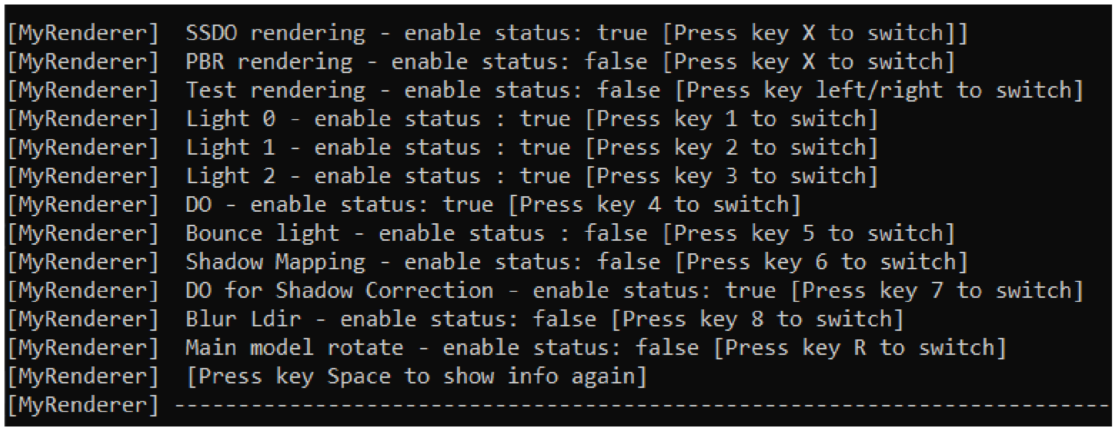
2.3 Control Screen Output

There are several key available for control screen output, in order to compare and check the output of each pass. As figure 10 shown, each time a key get pressed, console will output the corresponding state information, which is helpful to locate current rendering output. The explanation is shown as following:

1. SSDO rendering: the program is rendering by using SSDO method.

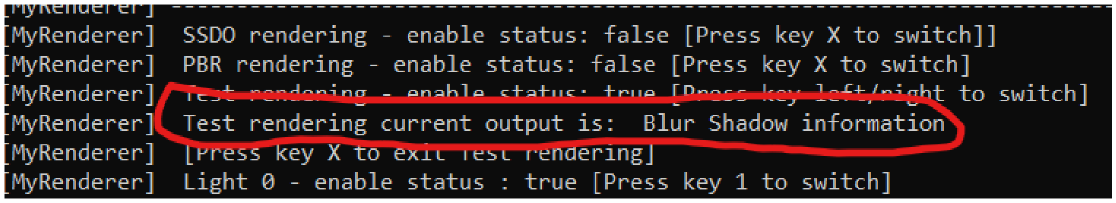
2. PBR rendering: the program is rendering by using PBR method.

3. Test rendering: the program is rendering the output from one specific pass. Key left/right can enter/exit this Test rendering. Each time to press Key left will decrease *outputType* while press Key right will increase it. *outputType* will be ping-pong inside range [0, 13]. Each value determine one the output from specific pass. More details could be found in *TestFragmentShader.glsl*. Information of which type of output will be displayed in console, as figure 11 shown.



```
[MyRenderer]  SSDO rendering - enable status: true [Press key X to switch]]
[MyRenderer]  PBR rendering - enable status: false [Press key X to switch]
[MyRenderer]  Test rendering - enable status: false [Press key left/right to switch]
[MyRenderer]  Light 0 - enable status : true [Press key 1 to switch]
[MyRenderer]  Light 1 - enable status : true [Press key 2 to switch]
[MyRenderer]  Light 2 - enable status : true [Press key 3 to switch]
[MyRenderer]  DO - enable status: true [Press key 4 to switch]
[MyRenderer]  Bounce light - enable status : false [Press key 5 to switch]
[MyRenderer]  Shadow Mapping - enable status: false [Press key 6 to switch]
[MyRenderer]  DO for Shadow Correction - enable status: true [Press key 7 to switch]
[MyRenderer]  Blur Ldir - enable status: false [Press key 8 to switch]
[MyRenderer]  Main model rotate - enable status: false [Press key R to switch]
[MyRenderer]  [Press key Space to show info again]
[MyRenderer] -----------------------------------------------------------------------
```

Figure 10: Console Output



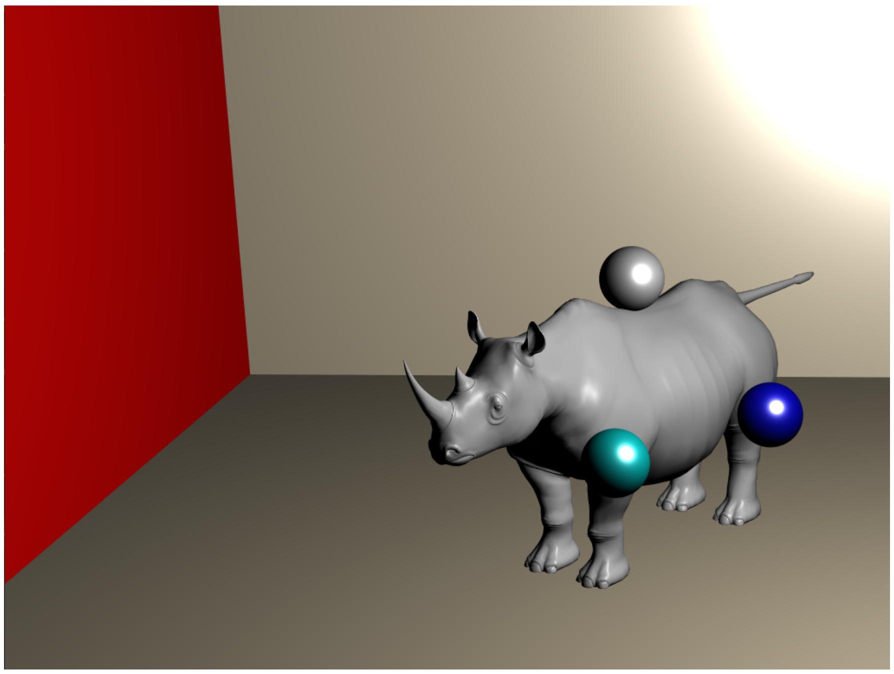
```
[MyRenderer] -----------------------------------------------------------------------
[MyRenderer]  SSDO rendering - enable status: false [Press key X to switch]]
[MyRenderer]  PBR rendering - enable status: false [Press key X to switch]
[MyRenderer]  Test rendering - enable status: true [Press key left/right to switch]
[MyRenderer]  Test rendering current output is:  Blur Shadow information
[MyRenderer]  [Press key X to exit Test rendering]
[MyRenderer]  Light 0 - enable status : true [Press key 1 to switch]
```

Figure 11: Console Output

4. Key Function:

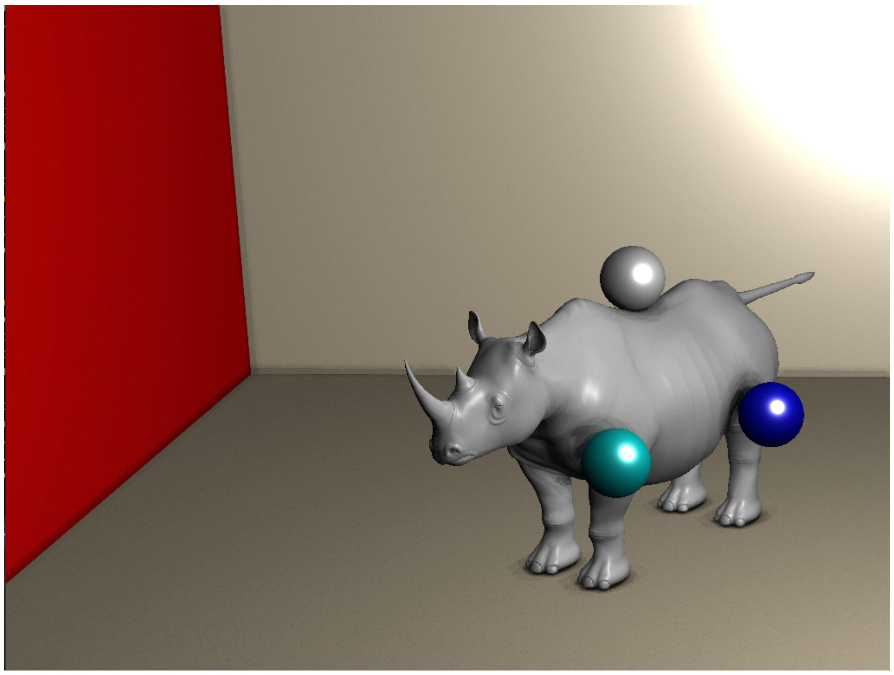
- Key x to switch to SSDO rendering or PBR rendering.
- Key left/right to enter/exit Test rendering. If it is already in Test rendering, then it will change different output from specific pass.
- Key r to rotate the main mesh.
- Key from number1 to number3 are controlling the activity of lights.
- Key number4 to select output $L_{dir} + L_{specualr}$ or L_{pbr}.
- Key number5 to select output L_{ind} or not.
- Key number6 to select output $Shadow$ or not.
- Key number7 to select using DO and Multiple Sampling for shadow correction or not.
- Key number8 to select blurred L_{dir} or not
- Key space to print current state.

3 Results



(a) PBR only

Figure 12


(a) PBR + blur DO

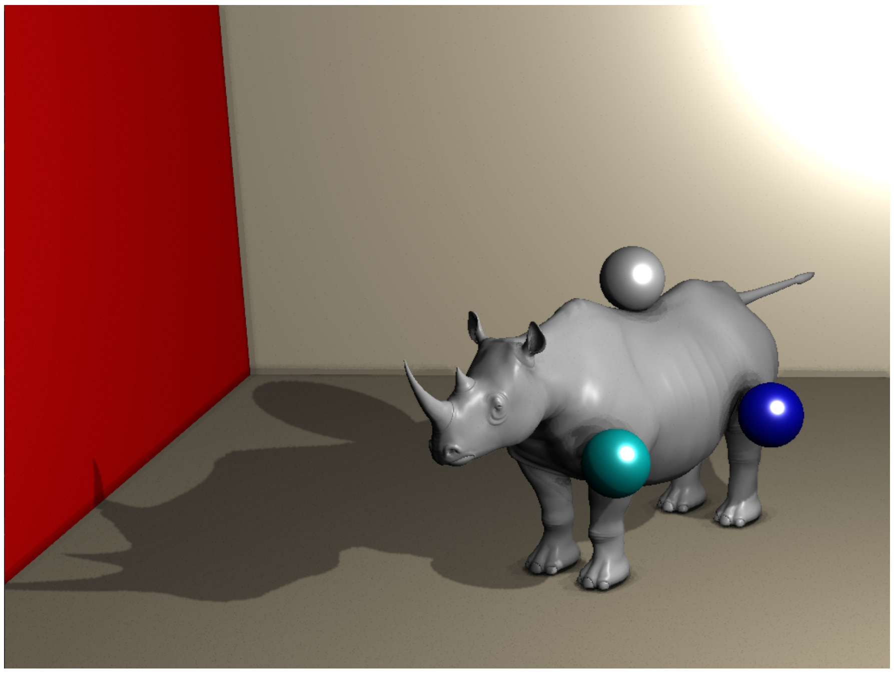

(b) PBR + blur DO+ shadow

Figure 13

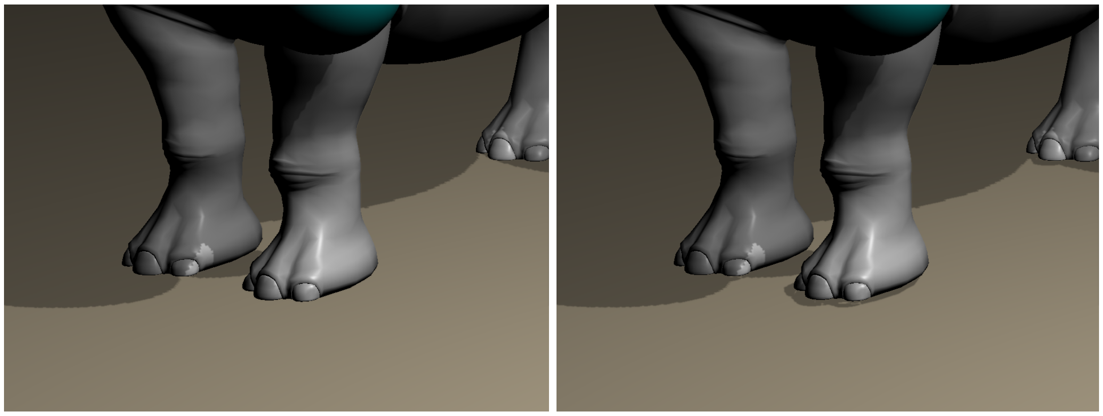

(a) Shadow Map + bias (b) Shadow Map + bias + DO correction

Figure 14

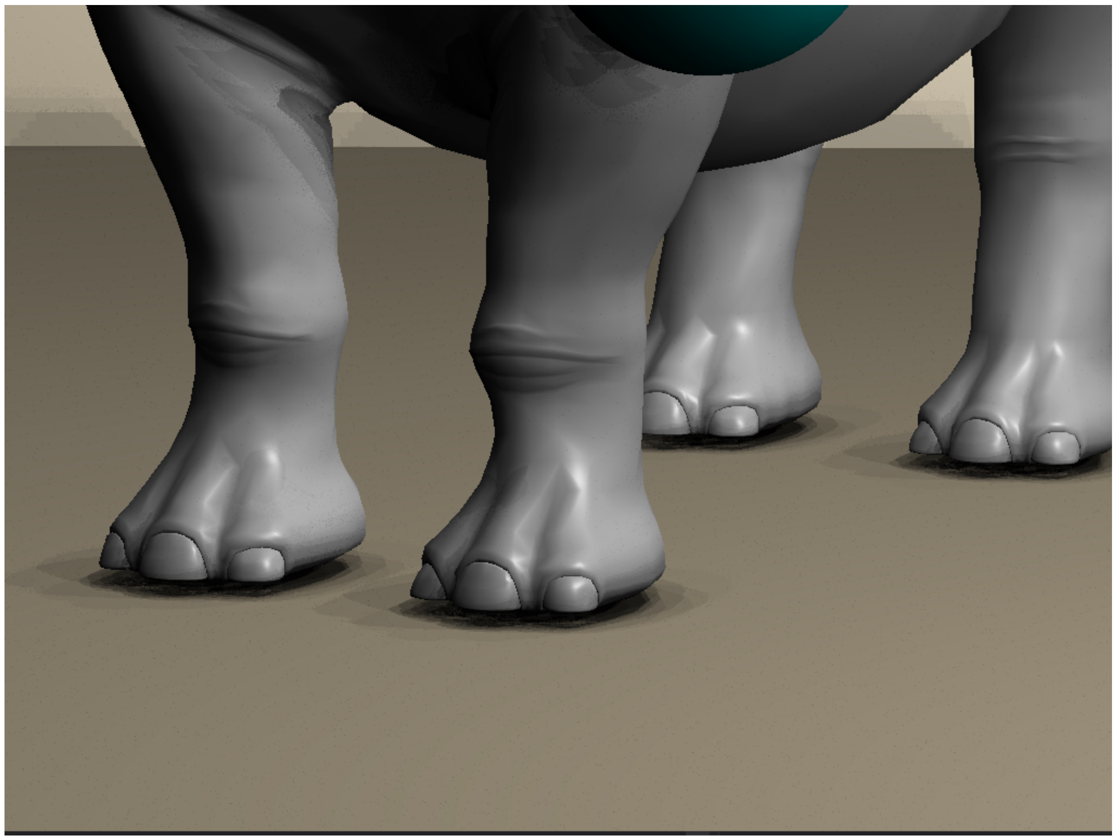
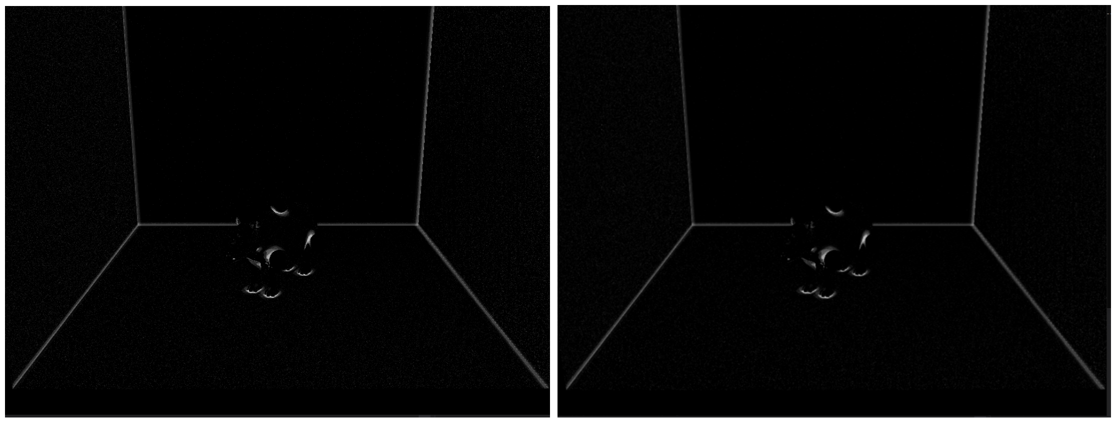


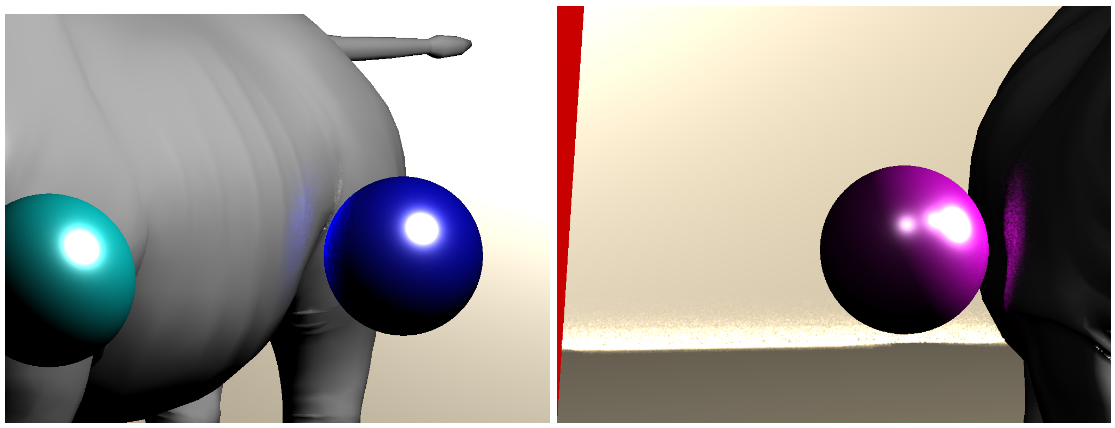
Figure 15: DO only



(a) DO

(b) Blurred DO

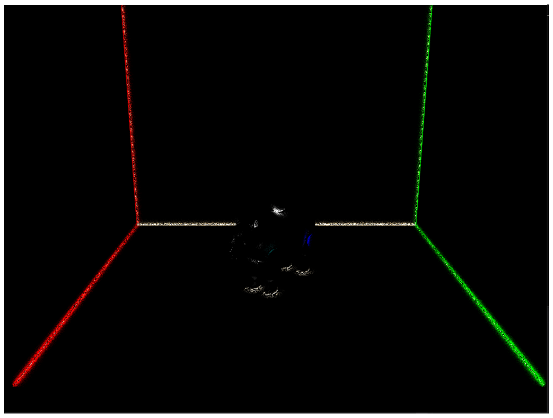
Figure 16



(a) Bounce 1

(b) Bounce 2

Figure 17


(a) Bounce

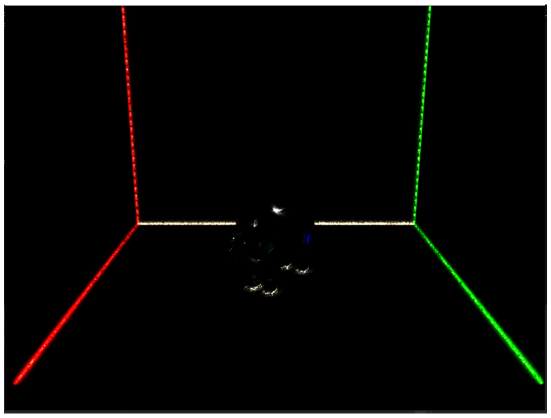

(b) Blurred Bounce

Figure 18

4 Conclusion

In summary, SSDO takes advantage of information that is already computed during the SSAO process to approximate two significant effects which contribute to the realism of the results: *Directional Occlusion* and *Indirect Bounces*. During the period when I implemented SSDO, I tried using SSDO technique to compute average Directional Occlusion, then blurred it and used it in equation 1, and it gave me a good enough visual result. However, I choose to follow equation 2 to compute the diffuse color in my final implementation.

Due to limit time, there are several deficiencies in my implementation, but I propose some possible improvement solutions:

- After program starts, the bottom floor will be black. It will become normal after move or rotate camera.
- In my implementation, I store $\lambda\omega_i$ in a Texture2D array for DO and Bounce pass to use. DO and Bounce do the same sampling procedure which is redundant. It can be improved as SSDO proposed, saving the sampling result in DO pass and reuse it in Bounc pass.
- When activating multiple lights, the shadows are not well being blurred and handled. Some shadow artifacts will occur on mesh surface.
- Apply L_{dir} into final color computation will bring noise but keep the distinct details of mesh, while using blurred L_{dir} will reduce noise but losing these details, because

I uses Gaussian Blur method which is not geometry-sensitive blur method. Due to limit time, I have not read the paper[2] proposed a geometry-sensitive blur method. However, another possible solution is to use DO directly to compute color like SSAO does, as mentioned in the beginning of this section, but it seems to degenerate SSDO to traditional SSAO.

- L_{ind} does not give smooth enough result even if blur it. One possible solution is to find a better filter, and do more sampling. Moreover, if we only choose specific object to bounce light, it will bring more pleasing visual result. In my scene, colored spheres and colored walls should be chosen to bounce light while others are not.
- This program runs in a very low FPS, which can be improved by reduce amount of pass, reduce loop in shader and code optimization.

References

[1] T. Ritschel, T. Grosch, and H.-P. Seidel. Approximating dynamic global illumination in image space. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, I3D '09, page 75–82, New York, NY, USA, 2009. Association for Computing Machinery.

[2] B. Segovia, J.-C. Iehl, and B. Péroche. Non-interleaved Deferred Shading of Interleaved Sample Patterns. In *Eurographics/SIGGRAPH Workshop on Graphics Hardware '06*, pages 53–60, nicosie, Cyprus, June 2006.

[3] P. Shanmugam and O. Arikan. Hardware accelerated ambient occlusion techniques on gpus. *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, 2007.