Screen-Space Ambient Occlusion by Unsharp Masking of the Depth Buffer

Jay Chamberlin*
Rensselaer Polytechnic Institute

Evan Sullivan†
Rensselaer Polytechnic Institute

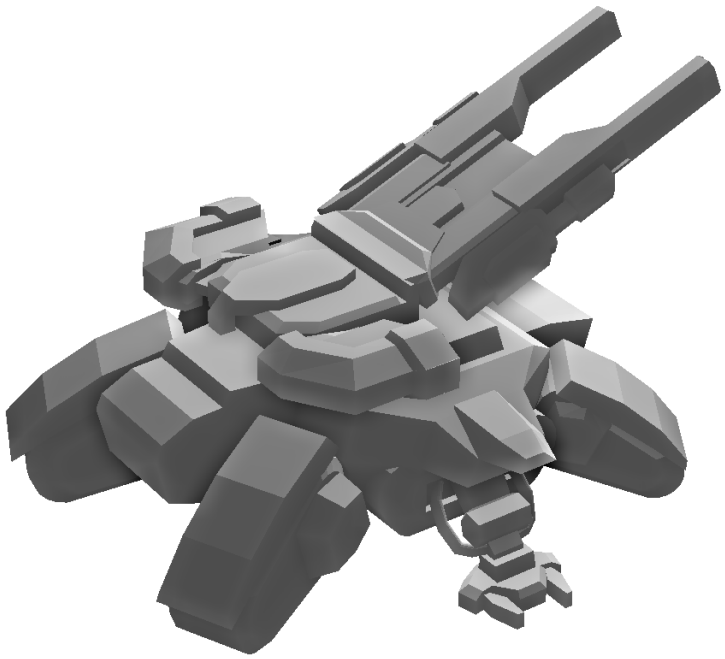

Figure 1: *Our SSAO implementation applied to a Siege Tank*

Abstract

With the advent of the programmable graphics pipeline, it has become possible to move expensive rendering processes to the GPU for real-time computation. The demand for realistic real-time rendering in games has led to the creation of GPU shaders that approximate global illumination. Our project is an implementation of one such approximation known as screen-space ambient occlusion (SSAO).

Keywords: screen-space ambient occlusion, unsharp mask, depth buffer

1 Introduction

One approximation to global illumination used commonly in production-quality renderings is ambient occlusion. Ambient occlusion uses the assumption that areas surrounded by a lot of other geometry will receive less light, making them darker. Normally, this process is accomplished by checking for occlusions within some hemisphere around a surface. SSAO approximates the process by taking advantage of the depth buffer.

We have written an SSAO shader using a technique known as unsharp masking on the depth buffer. The process results in a cheap and visually pleasing approximation of global illumination that is invariant to scene complexity.

*e-mail: chambj2@rpi.edu
†e-mail: sullie@rpi.edu

2 Related Work

In 2007, CryTek released a paper explaining several real-time rendering algorithms they were considering for their next-gen engine, CryEngine 2. One of these has come to be known as screen-space ambient occlusion [Mittring 2007]. Their implementation of SSAO on the 2007 video game Crysis was the first. Subsequently, the effect was adopted by many other game developers and used in big-budget games like Battlefield: Bad Company 2 (2010) and StarCraft 2 (2010). 2007 was also the year Shanmugam et al. described GPU-accelerated ambient occlusion in their paper Hardware Accelerated Ambient Occlusion Techniques on GPUs [Shanmugam and Arikan 2007].

Our implementation follows the algorithm outlined in Luft et al.s 2006 paper, Image Enhancement by Unsharp Masking the Depth Buffer [Luft et al. 2006]. Luft et al. describe a method by which 3D scenes can be enhanced with an approximation to ambient occlusion just by using the depth buffer. Our implementation differs from Luft et al.s slightly in that we implement the whole algorithm on a shader whereas they primarily worked on the CPU.

Since the introduction of these algorithms from 2006-2007, even more advanced rendering techniques have been implemented in screen space using the GPU. Of particular note is Approximating Dynamic Global Illumination in Image Space by Ritschel et al. in 2009 [Ritschel et al. 2009]. This paper outlines methods for directional shadows and indirect color bleeding, as well as ambient occlusion.

3 Implementation

All of our test scenes were rendered at a resolution of 1024x1024. Our implementation of SSAO utilizes a two-pass technique to first collect depth information about the scene and then use that information to calculate the unshap mask to apply to the scene. Gathering depth information from the depth buffer and storing it in a texture can be done with existing OpenGL functionality. However, processing that depth information to acquire an unsharp mask and overlaying that mask on the scene required a customized shader.

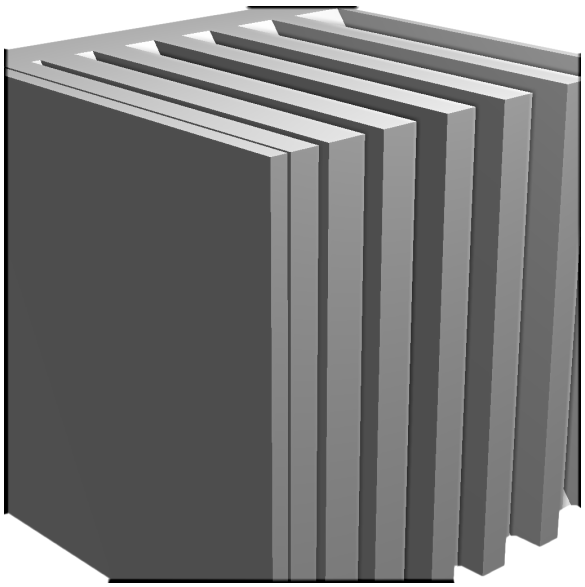

Figure 2: *Demonstration of darkening at the side clipping planes.*

Our fragment shader, which can be seen in the Fragment Shader section at the end of this document, is applied on the second pass of our render. The depth texture that was rendered in the first pass is sampled and averaged in a square with dimensions kernelDimension centered at the current fragment location. The larger the kernelDimension used, the better the SSAO effect will appear due to the averaging of more pixels depth values (Figure 3). The trade-off here is performance. The larger the kernelDimension the longer the shader will take to run due to the O(N2) performance of the averaging procedure. There is also a drastic difference depending on the hardware the shader is running on. With a kernelDimension of 15, a frame rate of 1-2 fps can be achieved on an Nvidia Qadro NVS 140M. On an Nvidia GTX 460 with the same kernelDimension, the frame rate was limited at 60 fps. Increasing the kernelDimension to 40 increased the render time to 5 seconds on the 140M, while the 460 was still being limited at 60 fps. On other machines we tested, a kernelDimension of 40 would crash the display driver entirely. Before we can use the depth values stored in the depth texture they must be converted from the logarithmic scale, in which they are originally calculated, to a linear scale:

$$D_L = \frac{(2.0 * N)}{F + N - Z * (F - N))} \tag{1}$$

Where D_L is the figurelinearized depth value, N and F are the near and far clipping planes respectively, and Z is the logarithmic depth value. Using this linearized value will result in a more consistent effect regardless of the models position in the scene with respect to the near and far clipping planes.

There are two checks which must be made before a neighboring pixels depth is added to our average from the depth texture. First, we must ensure that the neighbor we are trying to sample is actually on in our viewing window, i.e. in the depth texture. Failure to make this check will result in artifacts appearing when scene geometry is moved to the edge of the viewing window (Figure 2). The second condition that must be met before a neighboring pixels depth value is included in our average is a simple locality check. If the difference between the depth value of the neighbor and that of the current fragment is greater than the specified threshold value, that neighbors depth value is not incorporated into the average. Without this check in place, edges of the mesh that do not have scene geometry close behind them show the effects of SSAO in undesirable ways, either washing out the edges of the mesh, or highlighting edges that shouldnt be. Figures 4 and 5 show examples of these situations. In Figure 4 when pixels on the edge of the claw, which is not anywhere near the far clipping plane, are averaged with the other pixel depths on the claw, everything is fine. However, when pixel depths just off the claw at the far clipping plane, where the depth value is at its maximum, the average will come out to be much larger than it should. Subtracting this large average from the original depth on the claw can result in a negative value on the unsharp mask, which, when multiplied by 20, yields an even larger negative shade value (quite likely smaller than negative 1). Since shade is subtracted from the original pixel color, this operation effectively turns into an addition of a very large value to the fragments color which gets clamped to 1, a blown-out white that produces the results in the second image of Figure 4.

Incorporating a depth difference threshold eliminates the issue of averaging over the far clipping plane, and allows for averaging with pixels that have a more modest depth difference. the third image shows the threshold set to 0.1, allowing for the same SSAO effect around the mouth as seen in the second image, but none of the blown-out white around the edges that the second image suffered from.

Figure 5 shows another situation to be wary of with this fix. The claw in the foreground is far enough from the body that the body should not experience any noticeable occlusion due to the claw. This achieves the opposite effect we were going for; such occlusion makes it seem as though the claw is very close to the body, rather than helping the viewer differentiate the depths of objects. This sort of artifact is an unavoidable side-effect of this naive SSAO method; you have to choose a range of valid neighboring depths and you might encounter situations in which that range does not work. In the situation in Figure 5, reducing the depth threshold value to .02 eliminated the undesirable shadowing of the body under the claw.

One restriction that is a result of this check is that scene geometry must be kept in the front (100-threshold*100)% of the scene, otherwise the white-out effect will be observed. As the threshold value is typicaly quite small, though, this is a minor limitation. We never had a threshold value greater than 0.1 which restricted us to the front 90% of the view frustum (hardly a restriction at all).

4 Results

Our algorithm yields pleasant results on all of the models we tested, given that a few user-tewakable parameters have been correctly set. Figure 3 below shows the difference between using a 15x15 kernel (top) and using a 40x40 kernel (bottom). Both kernels accentuate edges well enough that the distinction between close and far geometry becomes more clear, but the size 40 kernel gives a subjectively better result.

Looking under the chin of the bunny in the original images, there is no distinguishable color change from the mouth to the chest. Us-

ing the size 15 kernel, it becomes easier to distinguish this depth difference and easier still using a size 40 kernel. This effect is also present at the transition from the top of the head to the right ear (our left). The transition is not difficult to see in the original image, but with SSAO applied, it becomes more distinct.

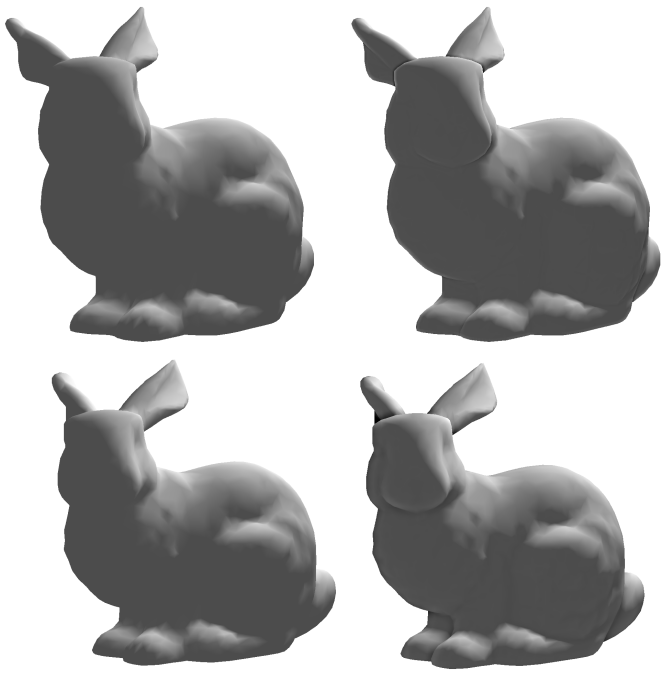

Figure 3: *Left Side: Stanford Bunny with Gouraud shading, Top Right: SSAO with kernelDimension = 15, Bottom Right: SSAO with kernelDimension = 40.*

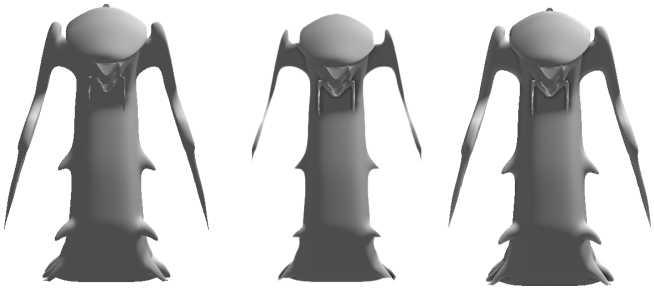

Figure 4: *Demonstration of a range of depth checking.*

The Siege Tank above is a good demonstration of what our SSAO effect looks like on complex geometry. The rear of the tank is uniformly shaded without SSAO, but has good distinction between the treads and body with SSAO. Some subtle points of interest are the rear of the turret, the stabilization mechanism, and the main gun.

5 Work Distribution

Our initial goals turned out to be much more ambitious than we had time for. This simple method of unsharp masking of the depth buffer was meant to be quickly completed by Jay to get a feel for SSAO so we could move on to more advanced methods of SSAO such as those described in [Shanmugam and Arikan 2007] and [Ritschel et al. 2009]. Due to the difficulties he encountered early on in the implementation of this method, after Evan created the test scenes he decided to skip beginning the implementation of described by Shanmugam et al. and assist Jay. We worked our way through several major revisions of our code to get the depth texture into the shader. The initial shader code was written by Jay, but Evan found several critical errors including the need to linearize the depth buffer values. We also experienced the joys of working on different hardware and attempting to debug problems on one machine that didnt exist on the other.

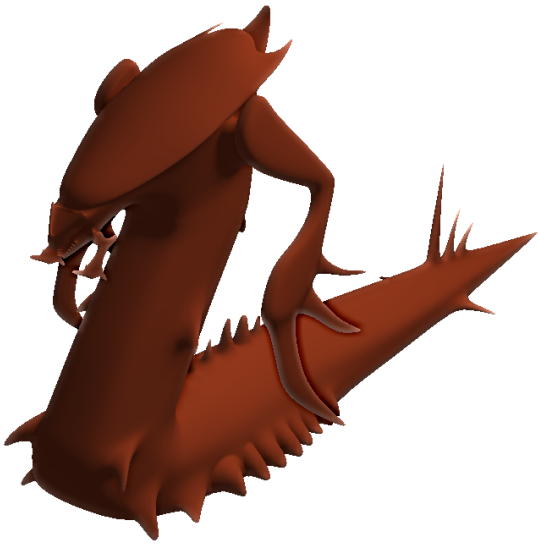

Figure 5: *Demonstration of insufficient depth checking (see claw).*

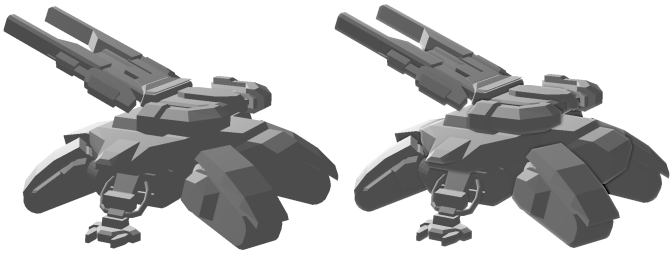

Figure 6: *Siege Tank without SSAO (left) and with SSAO (right).*

6 Conclusion

The performance we saw varied on different graphics cards as well as based on the size of kernelDimension. Performance is also affected by the resolution of the scene being rendered, athough we did not test any resolutions other than 1024x1024. Scene complexity did not play a significant role in the time it took our shader to run. The Hydralisk (Figure 5) has just over 58 thousand polygons and the time it took to apply our shader to it was similar to the time it took to apply our shader to a one thousand polygon Stanford Bunny. This is due to the nature of algorithms that operate in screen-space. The algorithms run-time is linked with the number of pixels on the screen.

It is feasible to run a simple screen-space ambient occlusion algorithm by doing unsharp masking of the depth buffer on a graphics card. The results attained through the use of our shader greatly improves the shadowing effect along the edges of our test scenes.

There were more enhancements to this algorithm as described in Luft et al. that we would like to implement in the future to improve the ambient occlusion. They describe a way to remove the white highlight along the edges where ambient occlusion occurs, which is undesirable in some situations. In most of our test scenes we felt the white highlight helped accentuate the effect, but there are certainly some cases where it would be distracting.

Acknowledgements

We thank Michael Snyder for helping us debug a few problems with our FBO code and SSAO shader.

Fragment Shader

```
varying vec3 normal;
varying vec3 position_eyespace;

uniform sampler2D depthValues;

const float kernelDimension = 15.0;
const float screenDimension = 1024.0;

float LinearizeDepth(vec2 uv)
{
  float n = 2.0; // camera z near
  float f = 15.0; // camera z far
  float z = texture2D(depthValues, uv).x;
  return (2.0 * n) / (f + n - z * (f - n));
}

void main()
{
  float sum = 0;
  int i = int(gl_FragCoord.x);
  int j = int(gl_FragCoord.y);
  int maxX = i + int(floor(kernelDimension/2.0));
  int maxY = j + int(floor(kernelDimension/2.0));
  float sampX;
  float sampY;
  float neighborCount = 0;

  for (int x = i - int(floor(kernelDimension/2.0)); x < maxX; x++) {
    for (int y = j - int(floor(kernelDimension/2.0)); y < maxY; y++) {
      sampX = float(x) / screenDimension;
      sampY = float(y) / screenDimension;
      if (sampX >= 0.0 && sampX <= 1.0 && sampY >= 0.0 && sampY <= 1.0 &&
        abs(LinearizeDepth(gl_FragCoord.xy/screenDimension) -
            LinearizeDepth(vec2(sampX, sampY))) < 0.02) {
        sum += LinearizeDepth( vec2(sampX, sampY) );
        neighborCount++;
      }
    }
  }

  vec3 color = vec3(1.0,1.0,1.0);
  vec3 light = normalize(gl_LightSource[1].position.xyz - position_eyespace);
  float ambient = 0.3;
  float diffuse = 0.7*max(dot(normal,light), 0.0);
  color = ambient * color + diffuse * color;

  sum = sum / neighborCount;
  float shade = 20 * (LinearizeDepth(gl_FragCoord.xy / screenDimension) - max(0.0, sum));
  gl_FragColor = vec4(color,1.0) - vec4(shade, shade, shade, 1.0);
}
```

References

LUFT, T., COLDITZ, C., AND DEUSSEN, O. 2006. Image enhancement by unsharp masking the depth buffer. *ACM Transactions on Graphics 25*, 3 (jul), 1206–1213.

MITTRING, M. 2007. Finding next gen: Cryengine 2. In *ACM SIGGRAPH 2007 courses*, ACM, New York, NY, USA, SIGGRAPH '07, 97–121.

RITSCHEL, T., GROSCH, T., AND SEIDEL, H.-P. 2009. Approximating Dynamic Global Illumination in Screen Space. In *Proceedings ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*.

SHANMUGAM, P., AND ARIKAN, O. 2007. Hardware accelerated ambient occlusion techniques on gpus. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, ACM, New York, NY, USA, I3D '07, 73–80.