Combining Screen-Space Ambient Occlusion and Cartoon Rendering on Graphics Hardware

Brett Lajzer
Dan Nottingham

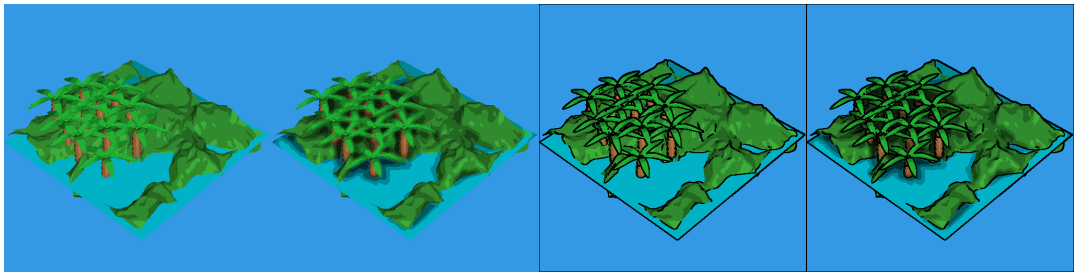


Figure 1: *Four visualizations of the same scene: a) no SSAO or outlining, b) SSAO, no outlines, c) no SSAO, outlines, d) SSAO and outlines*

1. Motivation

Methods for non-photorealistic rendering of 3D scenes have become more popular in recent years for computer animation and games. We were interested in combining two particular NPR techniques: ambient occlusion and cartoon shading. Ambient occlusion is an approach to global lighting that assumes that a point on the surface of an object receives less ambient light if there are many other objects occupying the space nearby in the hemisphere around that point. Screen-space ambient occlusion approximates this on the GPU using the depth buffer to test occlusion of sample points. We combine this with cartoon shading, which draws dark outlines on objects based on depth and normal discontinuities, and thresholds lighting intensity to several discreet values. We wanted our results to be applicable to games and thus had to be implemented as shader programs on the GPU to achieve interactive frame rates.

2. Related Work

Ambient Occlusion

Ambient Occlusion is an approximation to global illumination that assumes that, for any given point on a surface, the amount of ambient light reaching that point will depend on how much of the hemisphere surrounding that point is occupied by occluding objects. In theory, this involves integrating a visibility function over all the angles in the hemisphere. In practice, this integral is usually evaluated using some form of Monte Carlo ray casting.

Screen-Space Ambient Occlusion

Screen-space ambient occlusion (SSAO) is a further approximation of this technique, which was developed by CryTek for their game Crysis and its engine. This version computes ambient occlusion for each pixel visible on the screen, by generating random points in the hemisphere around that pixel, and determining occlusion for each point by comparing its depth to a depth map of the scene. The sample is considered occluded if it is further from the camera than the depth of the nearest visible object at that point, unless the difference in depth is greater than the sample radius. The advantage of this method is that it can be implemented on graphics hardware and run in real time, making it more suited to dynamic, interactive scenes due to its dependence only upon screen resolution rather than scene complexity. The main disadvantage is that it considers only information for the front-most geometry visible to the camera. For instance, a very long object will not occlude surfaces behind it, since the samples are only compared against the depth of the face closest to the camera.

3. Implementation

We implemented our techniques in OpenGL and GLSL. Most of our work is done in the fragment (pixel) shader stage of the pipeline, and not at the vertex level. The scene is rendered in several passes, rendering intermediate results to textures: first we acquire depth and normal information, next we compute the ambient occlusion, then we blur the SSAO results, then calculate lighting, and finally we draw outlines.

Pass 1: Depth and normal

In this step, we render the entire scene, outputting a color based on the normal at each pixel. Normal vectors are specified per vertex and interpolated across the triangle. At each pixel, we output the x, y and z components of the normal as the r, g and b components of the color, which is rendered to a texture. This step fills up the depth buffer, which we also store as a texture.

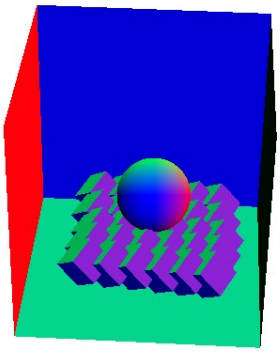


Figure 2: *The eye-space normals of the scene.*

Pass 2: Screen-space ambient occlusion

Here we compute the ambient occlusion at each pixel. We render a single quad that fills the screen, textured with the depth map from the previous pass and a randomized set of vectors given as a texture. The fragment shader generates eight random sample points within a radius by taking eight uniform points and reflecting them using a sample from the vector map. The depth map is sampled at each of these points, and if the depth at that point is greater than the depth at the pixel being calculated, we increase occlusion at that point by 1. The occlusion is then mapped from 0 to 1 and we output 1 – occlusion as the color.

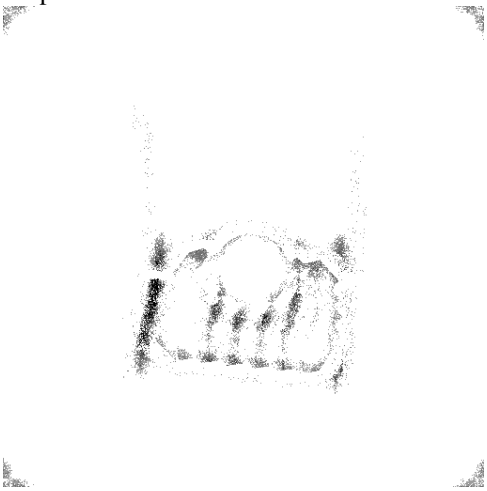


Figure 3: *The raw output of the SSAO shader.*

Pass 3: Blurring the SSAO

Because of the random sampling, the output of the ambient occlusion step is very noisy. We compensate by blurring the image using a 3x3 Gaussian filter. This is done by again mapping the texture to a quad and rendering it, and in the fragment shader, outputting the color as a weighted average of that pixel and the eight pixels surrounding it. This is applied several times, and smooths out the noise in the ambient occlusion considerably.

Figure 4: *The blurred SSAO.*

Pass 4: Lighting and color

The entire scene is rendered again in order to apply shading based on lighting and color. Standard per-pixel Phong lighting is used to calculate the intensity, but we also apply the brightness sampled from the ambient occlusion map. While theoretically this value would be used as the ambient term, we found that for our purposes, the best look was achieved by multiplying the diffuse and ambient terms by the value, then also adding some extra ambient. Specular highlights are not affected. The intensity is then clamped to a small number of threshold values to produce cartoon-like shading, and multiplied by the object color to produce the result.

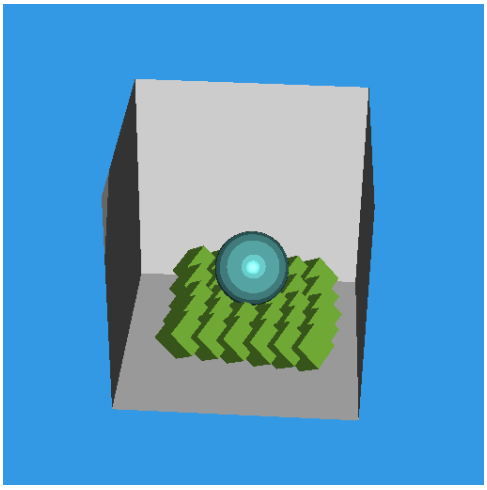


Figure 5: *The Phong interpolated lighting for the scene.*

Pass 5: Outlining

The final pass draws cartoon-like outlines on top of the image produced by the previous result. The shader is given the stored depth and normal textures, and examines them for discontinuities. At each pixel, we sample the depth and normal at that point and the eight points surrounding it in a square. We compare each sample to the center point, looking at the dot product of the normals and the difference in depth. If either violates a certain threshold for any sample, we consider that a discontinuity, and color that pixel black to create an outline. The depth discontinuities do a good job of finding the outer edges and silhouette of an object, but fail if the difference in depth between two objects is too small. Normal discontinuities find the sharp ridges on the surface of an object, and also tend to find outer edges if the normals between an object and the one behind it are fairly different. By combining the two methods, we usually find all the significant features that should have an outline drawn for them.

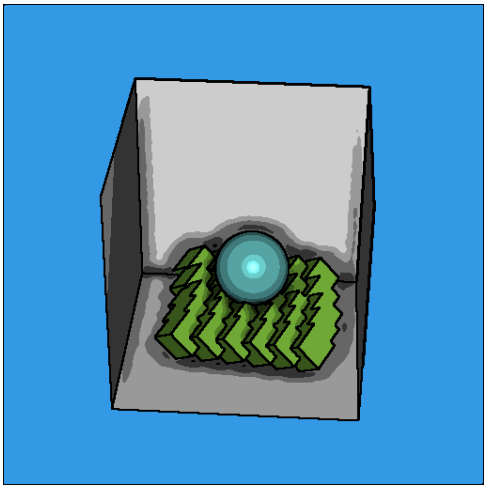


Figure 6: *The final composited scene with outlines.*

4. Results

As can be seen from the images in Appendix B, the application of Screen-Space Ambient Occlusion to cartoon shading has remarkable results. From figure 1, one can see that even without the thick outlines, SSAO can provide visual cues that help separate foreground and background. Adding outlines gives a distinct visual style that further enhances the details of the scene. This technique brings out topological details that are missed by both the Phong shading and the thick, black outlines, and it provides visual cues in the form of soft shadows/halos around objects. We also feel that the effect is quite pleasing and certainly adds more visual depth to a scene.

It also runs at interactive speeds, although with newer hardware, achieving realtime speeds would not be a problem.

5. Conclusions

SSAO Implementation Issues

Our implementation of SSAO makes a number of approximations. First of all, a sample is occluded even if the difference in depth is very large. This leads to some amount of over-occluding, but the resulting silhouettes actually work fairly well with our non-photorealistic style. Also, we sample within a sphere around each point, rather than a hemisphere based on the direction of the normal. This leads to some self-occlusion on flat surfaces, as some sample points will fall behind the surface itself and count as occlusions. However, we found that the number of samples for which this occurs is low enough that our value mapping technique eliminates this. Finally, we don't actually use the z-value of our random sample points, but rather the depth of the pixel being calculated. Essentially, this means that our sample points all fall on a plane and have the same screen-space z position, rather than being distributed throughout a sphere. We don't believe that this makes a very significant impact on our results, however.

Most of these inaccuracies in our implementation are due mainly to the limitations on the size of shader programs for our target hardware. The device we programmed for (ATI FireGL V3200) only supports up to Pixel Shader 2.0, which limits it to only 768 instructions per shader (as opposed to 65536 for 3.0). Since our SSAO shader has to iterate over eight samples, and since 'for' loops in GLSL are unrolled due to the lack of branching, we found ourselves very close to the instruction limit. Even a few small steps added inside the loop would put us over this limit, causing the shader to either crash or run in software at unacceptably slow speeds. Since we found that these details were not essential for fairly good-looking SSAO, we decided not to implement them to cut down on the instruction count.

Potential for Use

We feel that this technique has great potential for use as more companies look toward NPR shading techniques for use in games and hardware becomes increasingly more powerful. With more powerful hardware, proper SSAO can be implemented, which will achieve much smoother and more accurate results. Adding shadows and textures to this technique would fully complement and complete the visual style.

6. Bibliography

[1] *Finding Next Gen – CryEngine 2*, Mittring (2007)

[2] *Non-Photorealistic Rendering with Pixel and Vertex Shaders*, Card, Mitchell

Appendix A: Code Listings

Listing 1: Screen Space Ambient Occlusion Shader

```
// ssao.fs
//
// Screen Space Ambient Occlusion

uniform vec3 sample_points[8];

uniform sampler2D s_depth;
uniform sampler2D s_norm;

#define dp 1.0/512

const float radius = 16 * dp;

void main(void)
{
    //get the depth
    vec4 f_depth = texture2D(s_depth, gl_TexCoord[0].st);
    vec3 f_norm = normalize(texture2D(s_norm, gl_TexCoord[1].st).xyz);

    float occlusion = 0.0;

    float depth_sample;
    vec2 sp;
    for(int i=0; i < 8; i++)
    {
        sp = radius * (reflect(sample_points[i], f_norm).xy) + gl_TexCoord[0].st;
        depth_sample = f_depth.r - texture2D(s_depth, sp).r;

        //if the sample is closer than the current pixel -> occlude
        if(depth_sample > 0)// && depth_sample < radius * 0.05)
        {
            occlusion += 1.0;
        }
    }

    if(occlusion < 5.0){
        occlusion = 0;
    }else{
    //quadratic mapping
    occlusion = pow(occlusion, 2.0) / 64.0;
    }

    //output shade
    gl_FragColor.rgb = vec3(1.0-occlusion);
}
```

Listing 2: Per-Pixel Specular Toon Lighting Shader

```
// specular.fs
//
// per-pixel specular lighting

varying vec3 N, L;
varying float specularExp;

uniform sampler2D sampler0;

void main(void)
{
                //get SSAO value
                float ao = texture2D(sampler0, gl_FragCoord.xy/512.0).r;

    vec3 NN = normalize(N);
    vec3 NL = normalize(L);
    vec3 NH = normalize(NL + vec3(0.0, 0.0, 1.0));

    // calculate diffuse lighting
    float intensity = (max(0.0, dot(NN, NL)) + 0.1) * ao + 0.2;

    vec3 diffuse = gl_Color.rgb * intensity;

    // calculate specular lighting
    vec3 specular = vec3(0.0);
    if (intensity >= 0.0 && specularExp > 1.0)
    {
        intensity += pow(max(0.0, dot(NN, NH)), specularExp);
    }

                intensity = floor(intensity * 4.0) / 4.0;

    // sum the diffuse and specular components
    gl_FragColor.rgb = gl_Color.rgb * intensity;
    gl_FragColor.a = gl_Color.a;
}
```

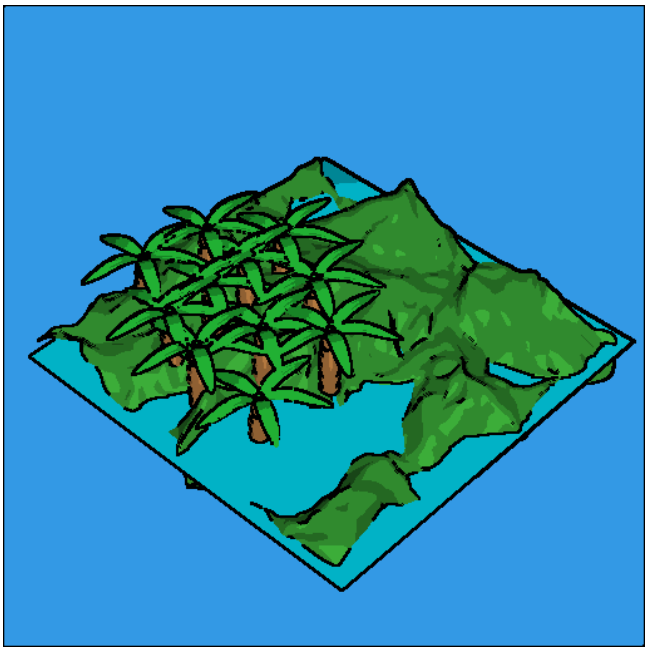
Listing 3: Toon Outline Shader

```
// toon_outlining.fs
//
// Depth discontinuities for toon outlining

vec2 offsets [8];

uniform sampler2D s_depth;
uniform sampler2D s_norm;

void main(void)
{
    vec4 depth = texture2D(s_depth, gl_TexCoord[0].st);
    vec4 norm = texture2D(s_norm, gl_TexCoord[0].st);
    float dp = 1.0 / 512.0;

    offsets[0] = vec2(-dp,-dp);
    offsets[1] = vec2(-dp,0);
    offsets[2] = vec2(-dp,dp);

    offsets[3] = vec2(0,-dp);
    offsets[4] = vec2(0,dp);

    offsets[5] = vec2(dp,-dp);
    offsets[6] = vec2(dp,0);
    offsets[7] = vec2(dp,dp);

    float darkness_depth = 0.0;
    float darkness_norm = 0.0;

    float base_depth = depth.r;
    vec3 base_norm = normalize(norm.xyz);

    float threshhold_depth = 0.05 * (1.0 - depth.r);
    float threshhold_norm = 0.95;

    for(int i = 0; i < 8; i++)
    {
        depth = texture2D(s_depth, gl_TexCoord[0].st + offsets[i]);
        norm.xyz = normalize(texture2D(s_norm, gl_TexCoord[0].st + offsets[i]).xyz);

        if (abs(depth.r - base_depth) > threshhold_depth) darkness_depth += 1.0;
        if (dot(norm.xyz, base_norm) < threshhold_norm) darkness_norm += 1.0;
    }

    float illum = 1.0 - (darkness_depth + darkness_norm);

    gl_FragColor.rgb = vec3(illum, illum, illum);
    gl_FragColor.a = 1.0;
}
```

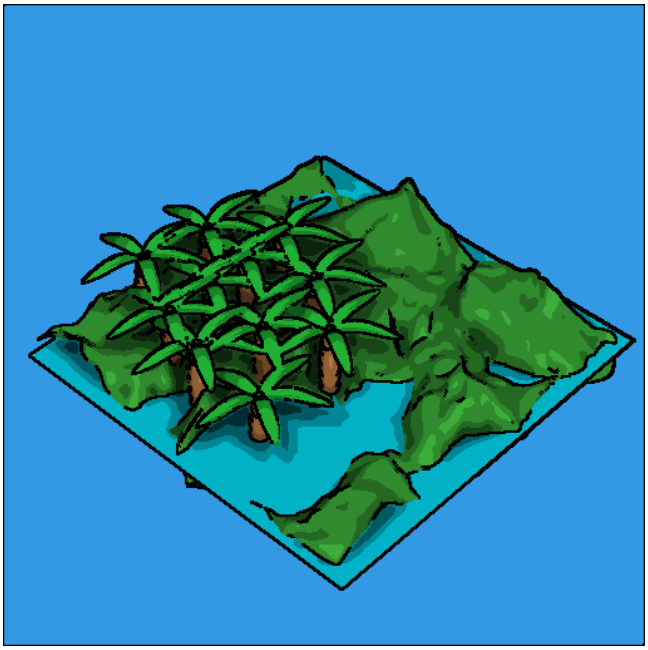
Appendix B: Sample Images

Sample 1: terrain.obj

Scene without SSAO

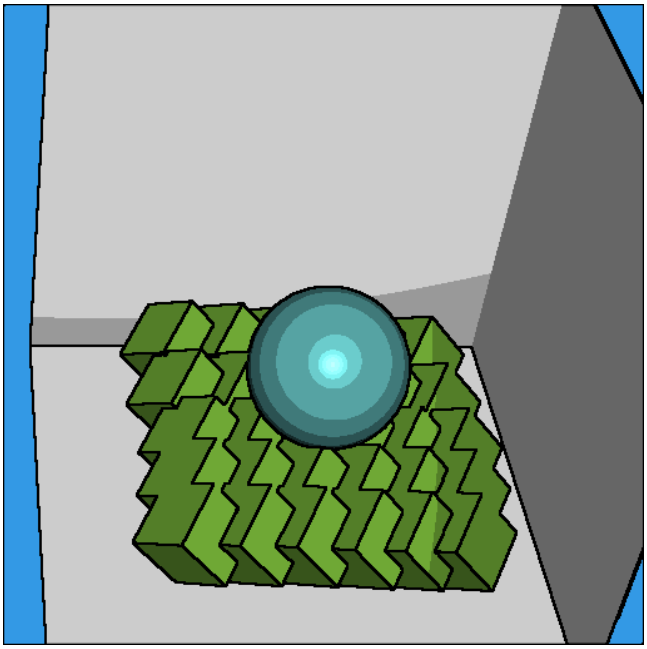


Scene with SSAO

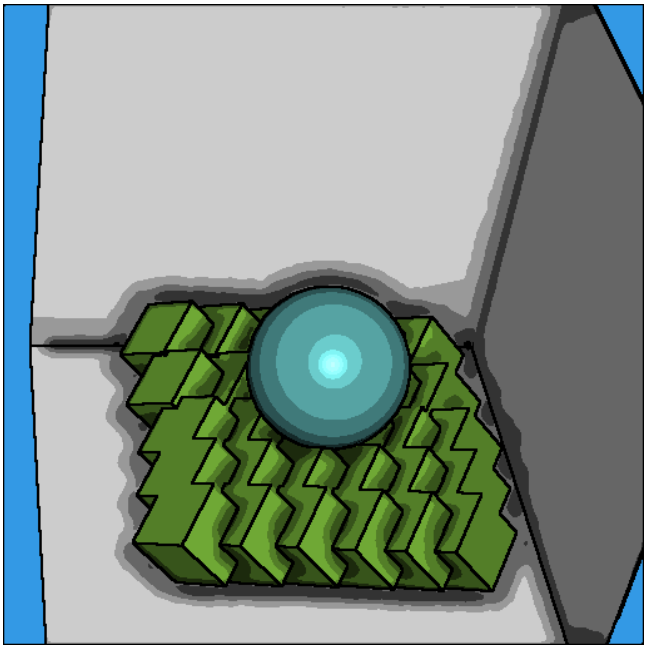


Sample 2: room.obj

Scene without SSAO

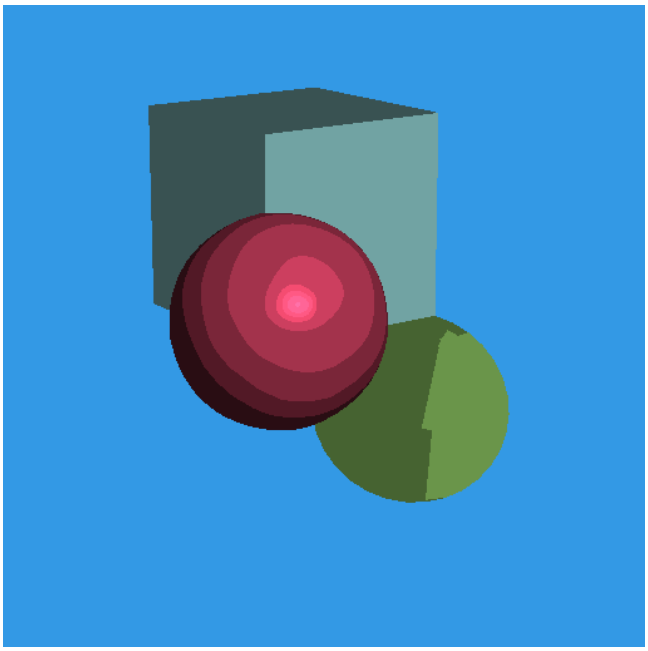


Scene with SSAO

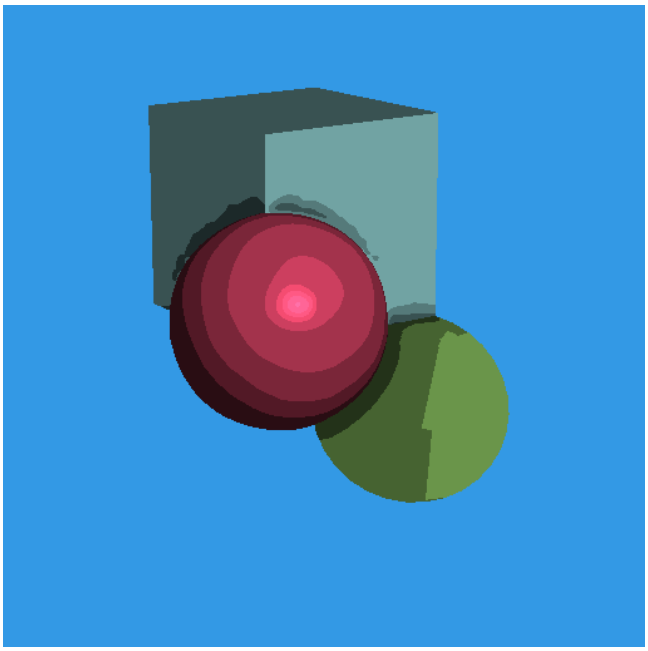


Sample 3: solids.obj

Scene without SSAO



Scene with SSAO



Sample 4: pink.obj

Scene without SSAO

Scene with SSAO