SSAO Assignment Report

Luis Carranza
luis.carranza@estudiantat.upc.edu
Universitat Politècnica de Catalunya
Barcelona, Spain

Figure 1. Left: Phong Shading. Middle: Alchemy SSAO with 12 samples per pixel. Right: Both methods combined.

Abstract

The project challenges students to learn the techniques used in Screen-Space Ambient Oclussion (SSAO) and implement one SSAO method based on a paper. In this project, The Alchemy Screen-Space Ambient Obscurance Algorithm was chosen to be implemented on top of the lightning rendering techniques developed in the previous assignment.

Keywords: rendering techniques, ambient occlusion, ambient obscurance, screen-space ambient occlusion

1 Introduction

With respect to the rasterization rendering method, ambient occlusion adds more information about the surface of an abject such as darkened corners, cracks, and wrinkles; proximity darkening; and contact shadows [McGuire et al. 2011]. In order to achieve this, the approach used is Screen-Space Ambient Occlusion since its wide adoption and good performance. Through the assignment, we focused on The Alchemy Screen-Space Ambient Obscurance Algorithm from Mcguire et al. in 2011. Due to the nature of the assignment, the project is divided into three exercises: Two-step renderer, Basic SSAO Implementation, and Improving SSAO.

For the first exercise, it is asked to implement a two-step renderer. By this, it will be possible to obtain information of the rendered image that will later be used. In the first render, information about normal, depth and albedo will be stored in textures.

The second exercise consists of a basic SSAO. In this exercise, the Ambient Obscurance (AO) formula from the paper will be applied with basic samples methods. To achieve this goal, several concepts will be used such as the ambient obscurance approximation and tuning parameters from the paper, uniform random sampling, kernel sampling, and recovering depth position between different coordinate systems.

In the last exercise, several improvements will be tried to the basic SSAO in order to overcome the problems it has until that step. Because of the nature of the AO equation from the chosen paper, most of these improvements will be focused on how to sample since other effects can be tuned using the parameters.

2 Setup

The project has been developed in Windows OS and the software pre-installed to run it is the following:

- Microsoft Visual C++ Compiler 17.1.32210.238 (amd64)
- QT 5.15.2 MSVC2019 64bits

If the project is build using a x86 architecture instead of x64, it is necessary to uncomment the lines in the .pro file located in the base directory. Once the configuration is done, the initial run should show Figure 3.

```
ViewerPBS.pro
25
26 # Using x64 architecture, comment if necessary
27 LIBS += -L'$$PWD/dependencies/glew/lib/Release/x64/' -lglew32 -lglu32 -lopengl32
28 LIBS += -L'$$PWD/dependencies/glew/bin/Release/x64/' -lglew32
29 # Using x86 architecture, uncomment if necessary
30 #LIBS += -L'$$PWD/dependencies/glew/lib/Release/Win32/' -lglew32 -lglu32 -lopengl32
31 #LIBS += -L'$$PWD/dependencies/glew/bin/Release/Win32/' -lglew32
```

Figure 2. Lines to configure depending on the compiler architecture in ViewerPBS.pro.

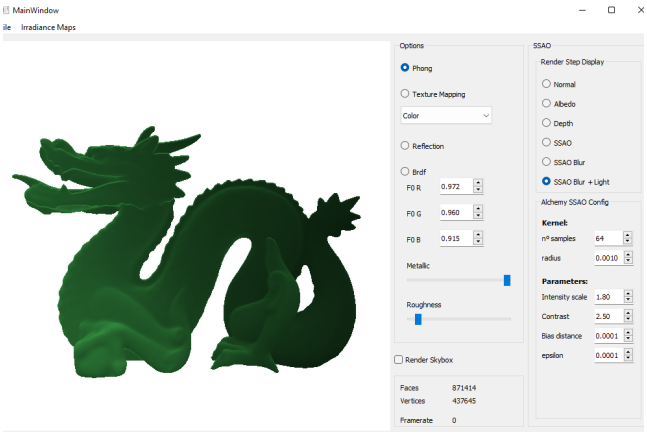

Figure 3. Initial status of the program when run.

3 The Alchemy Screen-Space Ambient Obscurance Algorithm

Their approach with respect to the others papers proposed is the extension of the Ambient Occlusion to an Ambient Obscurance equation that considers the distance by using a fallout function from Filion and McNaughton [2008] used in Starcraft II. Furthermore, the contribution of this paper is the utilization of comprehensive parameters for artists to tune the SSAO results. These parameters are σ, k, and β which resembles the Intensity scale, Contrast, and Bias distance correspondingly. As for the Ambient Obscurance algorithm, the explanation is as follows.

Given a point C in space, Γ as the set of all its near-field occluders projected onto an hemisphere Ω at a distance $0 < t(C, \hat{\omega}) < r$, and a fallout function $g(x)$, the Ambient Obscurance equation is the following:

$$A = 1 - \int_{\Gamma} g(t(C, \hat{\omega}))\hat{\omega} \cdot \hat{n} d\hat{\omega}$$

where $\hat{n}$ is the normal of point C.

In the paper, the fallout function chosen is $g(t) = u \cdot t \cdot max(u, t)^{-2}$, where u is a parameter that depends on r. What's more, since $\hat{\omega}$ is the unit direction of a ray from C, the expression $t(C, \hat{\omega})$ can be rewritten as $||\vec{v}(\hat{\omega})||$, and $\hat{\omega} = \vec{v}(\hat{\omega})/||\vec{v}(\hat{\omega})||$. That is:

$$A = 1 - u \int_{\Gamma} \frac{\vec{v}(\hat{\omega}) \cdot \hat{n}}{max(u^2, \vec{v}(\hat{\omega}) \cdot \vec{v}(\hat{\omega}))} d\hat{\omega}$$

Discretizing the integral using Riemman sum with a Heaviside step function $H(x)$:

$$A \approx 1 - \frac{2\pi u}{s} \sum_{i=1}^{s} \frac{max(0, \vec{v}_i \cdot \hat{n}) \cdot H(r - ||\vec{v}_i||)}{max(u^2, \vec{v}_i(\hat{\omega}) \cdot \vec{v}_i(\hat{\omega}))}$$

Finally, they replace ω for a σ parameter representing the Intensity scale, power the hole expression to k representing contrast, add a bias distance $z_C * \beta$ similar to the shadow map bias, and an ϵ to avoid division by zero.

$$A \approx max(0, 1 - \frac{2\sigma}{s} \sum_{i=1}^{s} \frac{max(0, \vec{v}_i \cdot \hat{n} + z_C\beta)}{\vec{v}_i \cdot \vec{v}_i + \epsilon})^k$$

4 Two-step renderer Exercise

For this exercise, it is needed to declare frame and render buffers as a previous step to obtain information to be used in the final render on the screen. In order to use them , these information buffers will be stored in textures. As for the exercise, it is asked to compute the Normal, Albedo and Depth. As for the implementation details, these will be split in two parts, First-step renderer and Second-step renderer.

4.1 First-step renderer

During the first-step, textures for albedo, normal and depth are created with Null data as seen in Figure 4, then a framebuffer is created for normal and albedo, while a renderbuffer is for depth. The creation and attachment implementation is shown in Figure 5.

```
832   // create a normal attachment texture
833   glGenTextures(1, &tex_ssao_map_normal_);
834   glBindTexture(GL_TEXTURE_2D, tex_ssao_map_normal_);
835   glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, scrWidth, scrHeight,
836                0, GL_RGB, GL_UNSIGNED_BYTE, NULL);
837   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
838   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
839   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
840   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
841
842   // create a depth attachment texture
843   glGenTextures(1, &tex_ssao_map_depth_);
844   glBindTexture(GL_TEXTURE_2D, tex_ssao_map_depth_);
845   glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT32, scrWidth, scrHeight,
846                0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
847   glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
848   glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
```

Figure 4. Create textures in InitializeGL.

```
851   // generate the FrameBuffers and attach the textures to them
852   // generate the RenderBuffers and attach the textures to them
853   glGenFramebuffers(1, &ssao_normal_FBO);
854   glGenRenderbuffers(1, &ssao_depth_RBO);
855
856   glBindFramebuffer(GL_FRAMEBUFFER, ssao_normal_FBO);
857   glBindRenderbuffer(GL_RENDERBUFFER, ssao_depth_RBO);
858   glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT, scrWidth, scrHeight); // use a single renderbuffer
859   glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER, ssao_depth_RBO);
860   glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, tex_ssao_map_normal_, 0);
861   glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, tex_ssao_map_depth_, 0);
862
863   // define the array of color buffers where the fragment shader will draw into
864   glDrawBuffer(GL_COLOR_ATTACHMENT0);
```

Figure 5. Create buffers in InitializeGL.

Once the buffers and textures are correctly declared, these can now be called in paintGL for a first-rendering step. In

order to paint in the framebuffer, the declaration as shown in in Figure 6 is necessary before drawing the objects. Finally, in the fragment shader the out variables are the ones to fill the textures in the way those were configured, that is depending on the drawBuffers declaration and color_attatchment corresponding to each texture. Since the albedo was computed in the previous assignment, step_one.frag only sends the normal, while the lighting fragments send both the lighting and albedo textures. These implementations can be seen in Figure 7 and Figure 8.

```
1151    /*************** First render step ***************/
1152    // bind the corresponding frame buffer.
1153    glBindFramebuffer(GL_FRAMEBUFFER, ssao_normal_FBO);
1154    //clear the buffers (at least color and depth), send the uniforms, and draw the geometry.
1155    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
1156    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Figure 6. Call buffers in PaintGL.

```
14  //layout (location = 0) out vec4 frag_color;
15  layout (location = 0) out vec4 frag_normal;
16
17  void main (void) {
18   vec3 N = normalize(eye_normal);
19
20   //vec3 albedo = vec3(0.3125f, 0.859375f, 0.390625f);
21
22   // write Total Color:
23   //frag_color = vec4(albedo, 1.0);
24
25   frag_normal = vec4(N, 1.0);
```

Figure 7. Get Normal and Depth in step_one.frag.

```
16  layout (location = 0) out vec4 frag_lightning;
17  layout (location = 1) out vec4 frag_color;
18
19  void main (void) {
40   // write Total Light:
41   frag_lightning = vec4(Iamb + Idiff + Ispec, 1.0);
42
43   vec3 albedo = vec3(0.3125f, 0.859375f, 0.390625f);
44
45   // write Total Color:
46   frag_color = vec4(albedo, 1.0);
47  }
```

Figure 8. Get Color and Lighting in phong.frag.

4.2 Second-step renderer

Since we already got the information as textures, painting those in a plane that exactly fits in the screen is enough to render the final results. In order to achieve that, a quad model is necessary to be called as seen in Figure 9. Once in the fragment, it is as simple as compute the texture with the TexCoords of the quad as seen in Figure 10.

Figure 11 shows the results of normal, albedo and depth of the space rendered in the quad.

```
1492        glBindFramebuffer(GL_FRAMEBUFFER, 0);
1493        glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
1494        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
1505        glUniform1i(texture_ssao_albedo_location, 9);
1506        glUniform1i(texture_ssao_normal_location, 10);
1507        glUniform1i(texture_ssao_depth_location, 11);
1508        glUniform1i(texture_ssao_ssao_location, 13);
1509        glUniform1i(texture_ssao_ssao_blur_location, 14);
1510        glUniform1i(texture_ssao_lightning_location, 15);
1511        glUniform1i(ssao_render_mode_location,ssao_render_mode_);
1512
1513        glBindVertexArray(quad_VAO);
1514        glDrawArrays(GL_TRIANGLES, 0, 6);
```

Figure 9. Call default screen, call textures and paint quad in PaintGL.

```
15  void main()
16  {
17
18      vec3 col = texture(texture_ssao_albedo, TexCoords).rgb;
19      vec3 normal = texture(texture_ssao_normal, TexCoords).rgb;
20      float depth = texture(texture_ssao_depth, TexCoords).r;
21      float ssao = texture(texture_ssao_ssao, TexCoords).r;
22      float ssao_blur = texture(texture_ssao_ssao_blur, TexCoords).r;
23      vec3 lighting = texture(texture_ssao_lightning, TexCoords).rgb;
24
25      if(ssao_render_mode == 0){ // Normal
26          frag_color = vec4(normal, 1.0);
27      } else if(ssao_render_mode == 1){ // Albedo
28          frag_color = vec4(col, 1.0);
29      }
30      else if(ssao_render_mode == 2){ // Depth
31          frag_color = vec4(depth,depth,depth, 1.0);
32      }
33      else if(ssao_render_mode == 3){ // SSAO
34          frag_color = vec4(ssao,ssao,ssao, 1.0);
35      }
36      else if(ssao_render_mode == 4){ // SSAO Blur
37          frag_color = vec4(ssao_blur,ssao_blur,ssao_blur, 1.0);
38      }
39      else if(ssao_render_mode == 5){ // SSAO Blur + lighting
40          frag_color = vec4(col, 1.0)*vec4(lighting, 1.0)*vec4(ssao_blur,ssao_blur,ssao_blur, 1.0);
41      }
```

Figure 10. Paint quad with texture based on render mode in step_four.frag.

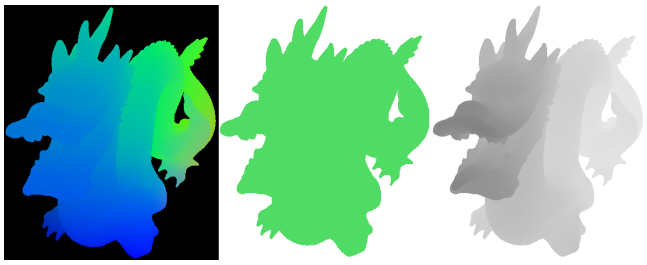

Figure 11. Left: Normal. Middle: Albedo. Right: Depth.

5 Basic SSAO Implementation Exercise

In this exercise, a first SSAO algorithm will be implemented based on the chosen paper. The first step will implement a random sampling method to analyze the occlusion of neighbor pixels in the fragment. After this kernel should be sent to the fragment shader with the normal and depth to compute the Alchemy SSAO algorithm explained before.

5.1 Sampling Method: Uniform Random Sampling

According on the chosen paper, a uniform random sampling should be implemented to compute the AO. Based on the Non-Uniform Random Variate Generation book by Luc Devroye published in 1986, shown in https://blogs.sas.com/content/iml/2016/03/30/generate-uniform-2d-ball.html, to

get random samples uniformly, the radius sampling should be over the square root of a uniform random distribution. The implementation of this kernel is shown in Figure 12.

```
963     //Kernel sampling
964     float twopi = 2 * M_PI;
965     for (unsigned int i = 0; i < 64; ++i)
966     {
967         float theta = twopi * randomFloats(*QRandomGenerator::global());        /* an
968         float r = /*ssao_radius_ **/ sqrt(randomFloats(*QRandomGenerator::global()));
969         glm::vec2 sample(
970             r*cos(theta),
971             r*sin(theta)
972         );
973         ssao_kernel.push_back(sample);
974     }
```

Figure 12. Generate uniform random sampling in initializeGL.

5.2 Compute the Alchemy SSAO

I order to get the correct z component from the depth component, it is necessary to transform its value from the Perspective frustrum to the Normalized Device Components as explained in the Guideline 2. Once in the NDC space, a final transformation to the Eye space is computed to be in the same coordinate system as X_eye and Y_eye. Its implementation can be seen in lines 34-36 in Figure 13. As for the Alchemy SSAO algorithm, its implementation is shown in the following lines of the same Figure. The results can be seen in Figure 14 and in the video attached, where the parameters shown a great contribution in the SSAO results.

```
40  float sample_obscurance_sum = 0;
41  for(int i=0;i<ssao_n_samples;i++){
42      float ssao_sample_x = ssao_samples[i].x * cos(twopi*random) - ssao_samples[i].y * sin(twopi*random);
43      float ssao_sample_y = ssao_samples[i].x * sin(twopi*random) - ssao_samples[i].y * cos(twopi*random);
44      vec2 ssao_sample = normalize(vec2(ssao_sample_x,ssao_sample_y))*ssao_radius;
45      ssao_sample_x = ssao_sample.x;
46      ssao_sample_y = ssao_sample.y;
47
48      /*float ssao_sample_x  = ssao_samples[i].x * ssao_radius;
49      float ssao_sample_y  = ssao_samples[i].y * ssao_radius;*/
50
51      float EFx = ssao_sample_x * z_eye / z_near;
52      float EFy = ssao_sample_y * z_eye / z_near;
53
54      float Fx = Vertex.x + EFx;
55      float Fy = Vertex.y + EFy;
56      if(Fx > 1.f || Fx < 0.f || Fy > 1.f || Fy < 0.f)
57          continue;
58      float depth_2 = texture(texture_ssao_depth, vec2(Fx,Fy)).r;
59      float Fz = 2.0 * depth_2 - 1.0;
60      Fz = 2.0 * n * f / (f + n - Fz * (f - n));
61
62      vec3 sample_vector = vec3(EFx, EFy, -(Fz-z_eye));
63      sample_obscurance_sum += max(0, dot(sample_vector,normal) - z_eye+ssao_beta)/(dot(sample_vector,sample_vector)+ssao_epsilon);
64  }
65
66  float sigma = ssao_sigma * ssao_radius+twopi;
67  sample_obscurance_sum = pow(max(0,1-(1/sigma/ssao_n_samples)*sample_obscurance_sum),ssao_k);
68
69  frag_color = vec4(sample_obscurance_sum,sample_obscurance_sum,sample_obscurance_sum, 1.0);
```

Figure 13. Compute Ambient Obscurance in step_two.frag.

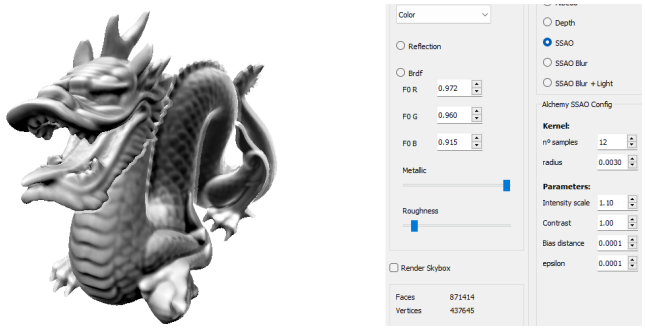

Figure 14. Basic SSAO render.

6 Improving SSAO Exercise

As for the last exercise, some techniques recommended in the chosen paper will be implemented to enhance results, while others will be analyzed to understand why these were not proposed in the paper.

6.1 Random texture for kernel sampling

Random texture avoid the banding effect when computing the SSAO with multiple sample of high radius, with the cost of adding noise. This texture acts as an angle to rotate the kernel sample location using its alpha channel as seen in line 42-43 of Figure 13. Three random textures have been used to see how different configuration of random textures affects the final result. The results of these experiments leads to not to choose a random texture and use low radius to avoid the banding effect while tuning the parameters for desired results. This is because the noise effect can create a cloud or noisy filter depending on the random method chosen that is difficult to remove with basic blur methods.

6.2 SSAO Blur

As for the Blur method, the paper proposes applying Gaussian blur to maintain shape. In comparison to a basic mean blur, Gaussian blur outperforms because of minimizing the effect of blurry images. This effect and comparison can be seen both in Figure 17 and in the video. As for the implementation details, the Gaussian blur is seen is Figure 16 and basic mean blur in Figure 15.

```
49      vec2 texelSize = 1.0 / vec2(textureSize(texture_ssao_ssao, 0));
50      float result = 0.0;
51      for (int x = -2; x < 2; ++x)
52      {
53          for (int y = -2; y < 2; ++y)
54          {
55              vec2 offset = vec2(float(x), float(y)) * texelSize;
56              result += texture(texture_ssao_ssao, TexCoords + offset).r;
57          }
58      }
59      result = result / (4.0 * 4.0);
60
61
62      frag_color = vec4(result,result,result, 1.0);
```

Figure 15. Blur SSAO fragment.

```
23  void main()
24  {
25      int diameter = 3;
26      float iFiltered = 0;
27      float wP = 0;
28      float neighbor_x = 0;
29      float neighbor_y = 0;
30      int halff = diameter / 2;
31      vec2 texelSize = 1.0 / vec2(textureSize(texture_ssao_ssao, 0));
32
33      for(int i = 0; i < diameter; i++) {
34          for(int j = 0; j < diameter; j++) {
35              neighbor_x = TexCoords.x - (halff - i)*texelSize.x;
36              neighbor_y = TexCoords.y - (halff - j)*texelSize.y;
37              float gi = gaussian(texture(texture_ssao_ssao, vec2(neighbor_x,neighbor_y)).r - texture(texture_ssao_ssao, TexCoords).r, 0.12f);
38              float gs = gaussian(distance(TexCoords.x, TexCoords.y, neighbor_x, neighbor_y), 16.f);
39              float w = gi * gs;
40              iFiltered = iFiltered + texture(texture_ssao_ssao, vec2(neighbor_x,neighbor_y)).r * w;
41              wP = wP + w;
42          }
43      }
44      iFiltered = iFiltered / wP;
45      frag_color = vec4(iFiltered,iFiltered,iFiltered, 1.0);
```

Figure 16. Gaussian Blur SSAO fragment.

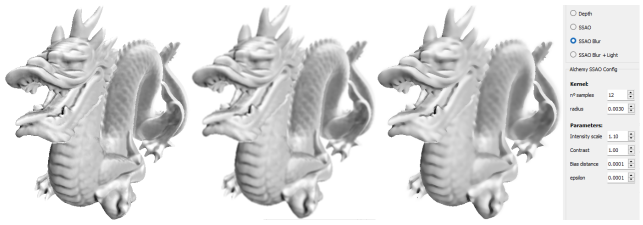

Figure 17. Left: Adding random texture. Mid: Adding mean blur. Right: Adding Gaussian blur.

6.3 Light + SSAO Blur

Finally, we combine the Light techniques from the previous assignment with the enhanced SSAO giving results as seen in Figure 18. The final values are the result of multiplying the singular values of each technique as seen in Figure 10.

Figure 18. Left: Phong Shading + Improved SSAO. Mid: Phong Shading + Texture Mapping + Improved SSAO. Right:Phong Shading + Improved SSAO + Skybox.