Fast Realistic Rendering

LAB 1

Emeline GOT

---

1 Screen-space ambient occlusion with blur

The SSAO effect is applied on each fragment. To be able to compute SSAO, we first need to render the geometry of the scene (positions, normals and depth) inside textures that we will then use for the computation of the occlusion factor. After this, we will use the render of SSAO to compute a Gaussian Blur. We also need to render SSAO into a texture.The global algorithm will thus have 3 passes : the geometry pass, the SSAO pass and the blur pass.

In the following parts, I will document the shaders used at each pass.

1.1 Geometry pass

The main purpose of the fragment shader is to get the vertices and the normals and to apply transformations to go from 3D to 2D. It means that we are going to use the matrices of transformations (projection, view, model and normal matrix) in order to be able to see the objects with the perspective applied. To do so, we get the vertices and the normals as vertex attributes using the keyword *layout*. Those variables are linked to the VBOs by the function *bindAttributeLocation*. We also get the matrices of transformations as global variables with the keyword *uniform*. We compute the multiplications between the matrices and the normals / vertices in order to get them in 2D perspective and we send those projected coordinates to the fragment shader. We define *gl_Position* as the clip-space output position vector of the vertex shader.

ssao_geom.vert

```
#version 330 core

//Vertex attributes
layout (location = 0) in vec3 vert;    //Vertice inputs
layout (location = 1) in vec3 normal;    //Normal inputs

// Get matrices as uniforms
uniform mat4 u_projection;
uniform mat4 u_view;
uniform mat4 u_model;
uniform mat3 u_normal_matrix;

// Send normals & vertices transformed in perspective to fragment shader
out vec3 N;
out vec3 V;



void main()
{

    vec4 view_vertex = u_view * u_model * vec4(vert, 1);    //Transform
        vertices to perspective
    V = view_vertex.xyz;
    N = normalize(u_normal_matrix * normal);    // Transform normals to
        perspective

    gl_Position = u_projection * view_vertex;    // Value of the clip-
        space output position vector

}
```

In the fragment shader, we receive the outputs N and V of the vertex shader. The *main* function will attribute the 2D coordinates of N and V to the *gPosition* and *gNormal* textures that we defined as Position and Normal textures in the FrameBuffer. To use the normal coordinates (between -1 and 1), we realise a little transformation that allows us to obtain normal texture coordinates between 0 and 1, which are the min and max coordinates of a texture. We

send the two textures as output in order to use them in the second pass.

ssao_geom.frag

```
#version 330 core

// Send textures as output
layout (location = 0) out vec3 gNormal;
layout (location = 1) out vec3 gPosition;

// Receive vertex shader outputs as inputs
in vec3 N;
in vec3 V;

void main(void)
{
     gPosition=V;  // Store the fragment positions in the texture gPosition
     gNormal = (normalize(N)+1.0)/2;   //Store the per-fragment normals in the texture gNormal

}
```

1.2 SSAO pass

To carry out with the SSAO pass, we will compute the occlusion value for each of the fragments of a 2D screen-filled quad. To do so, we will get the coordinates of the fragments displayed on the quad that is rendering at render time by the SSAO pass. The fragment positions are vec2 because the quad is in 2D. We transform them to make sure they are between 0 and 1 as all texture coordinates (eliminate negative values). We also define *gl_Position* to have the final position of the vertices on the screen.

ssao.vert

```
#version 330 core

layout (location = 0) in vec3 vert;  //Coordinates of the displayed quad

out vec2 TexCoords;  // Texture coordinates (U,V) as output to fragment shader


void main(void)  {

    vec2 FragPos = vec2(vert.x,vert.y);   // Fragment position defined by the vertices (x,y) of the quad
    TexCoords = (FragPos * vec2(0.5)) + vec2(0.5);   // Transform fragment position into texture coordinates
    gl_Position = vec4(vert,1.0);   // Value of the clip-space output position vector


}
```

In the fragment shader, we have the code of the SSAO algorithm. The idea is that we consider each fragment and for each of this fragment, we consider samples that are in a hemisphere around them. We will make a comparison between the fragment's depth and the sample's. The occlusion factor will be the number of samples that have a higher depth value than the fragment's.

For each fragment, we get its position and its normal thanks to the textures that we obtained in the geometry pass. We also consider a random value that allows us to have hemispheres created with some randomness and to be able to have a good result with less samples. This random value is stored in a small 4x4 texture. We compute a *noiseScale* that will be used to repeat this texture on the whole screen.

The samples are generated in tangent space and we define the othonormal basis that allows us to do the change from tangent to view space. In view space we will be able to consider samples around the position of the considered fragment. We do a depth test in screen space by using the *gPosition* texture and we compute the occlusion value for each fragments. Using the occlusion value, we can determine the fragment color by taking the corresponding lighting value (1-occlusion).

ssao.frag

```
#version 330 core

in vec2 TexCoords;  // Get texture coordinates from vertex shader

out vec4 frag_color;   // Give the color of the fragment as output to the shader

// Get the textures as uniform inputs
uniform sampler2D gNormal;
uniform sampler2D gPosition;
uniform sampler2D texNoise;

uniform int sampleN;  // Number of samples to use
uniform vec3 samples[64]; // Array of samples created at initialisation
uniform mat4 projection; // Projection matrix used to be in screen space
uniform float radius;  // Radius of the hemisphere in which we take the samples
uniform float width;  // Width of the screen
uniform float height;  // Height of the screen

// Find the scaling necessary to scale TexCoords between 0 & 1 and tile the screen with the noise texture
vec2 noiseScale = vec2(width/4.0, height/4.0);

void main(void)
{
    vec3 fragPos = texture(gPosition,TexCoords).xyz;   // Fragment position taken from gPosition texture
    vec3 normal = normalize(texture(gNormal, TexCoords).rgb *2 -1);    // Per-fragment normal taken from gNormal texture and transformed back to the initial value by doing the inverse computation
    vec3 randomVec = texture(texNoise, TexCoords * noiseScale).xyz;  // Random value computed from the texNoise texture using the noiseScale
```

```
27
28          //Creation of an orthonormal basis using randomness for the
                tangent vector
29      vec3 tangent   = normalize(randomVec - normal * dot(randomVec, normal
            ));     // Tangent vector
30      vec3 bitangent = cross(normal, tangent);  // Bitangent vector (3rd
            vector of the basis)
31      mat3 TBN       = mat3(tangent, bitangent, normal);  //Orthogonal
            matrix of change of coordinates
32
33      float occlusion = 0.0;  // Initialise occlusion
34
35      for(int i = 0; i < sampleN; ++i)  // Loop over all the samples
36      {
37
38          vec3 sample = TBN * samples[i]; // Transform samples from tangent
                into view-space
39
40          sample = fragPos + sample * radius;  // Add them to the current
                fragment position to choose points inside the hemisphere (with
                value of radius)
41
42          vec4 offset = vec4(sample, 1.0);  // Add 4th coordinate to apply
                transformations
43          offset      = projection * offset;    // Transform offset from
                view-space to clip-space (prepare for rasterisation)
44          offset.xyz /= offset.w;               // Divide by w to get to
                screen-space between -1 and 1
45          offset.xyz  = offset.xyz * 0.5 + 0.5; // Transform to range 0.0 -
                1.0
46
47
48          float sampleDepth = texture(gPosition, offset.xy).z;  // Sample
                the position texture & get depth of the sample position from
                viewer's perspective
49
50          if (fragPos.z < sampleDepth && abs(fragPos.z - sampleDepth) <
                radius) {   // Test if the sample's depth value is larger than
                the depth value of the fragment and apply a range check
51              occlusion+=1;  // Add 1 if the sample contributes to
                    occlusion
52          }
53
54
55      }
56
57      occlusion = 1.0 - (occlusion / sampleN);   // Normalize occlusion
            contribution by kernel size & substract from 1.0 to use as light
            component
58      frag_color = vec4(occlusion,occlusion,occlusion,1.0);  // Use
            occlusion as light component
59
60
61
62  }
```

1.3 Blur

In the blur vertex shader, we need to retrieve the vertices of the quad as we will still compute the blur on each fragments of a 2D screen-filled quad. We also have the texture coordinates from the SSAO texture that we can use without modifications since they were made previously in the SSAO shader. As before in vertex shaders, we compute the final position of the vertices on the screen. Here we do not need any model matrix since the quad fills the screen.

blur.vert

```
#version 330 core

layout (location = 0) in vec3 vert;    // Get the quad vertices
layout (location = 1) in vec2 texCoords;    // Get the corresponding
    texture coordinates

// Transformation matrices for the quad as uniforms
uniform mat4 q_projection;
uniform mat4 q_view;

// Send texture coordinates to fragment shader
out vec2 TexCoords ;

void main(void)  {

    TexCoords = texCoords;  // The texture coordinates are already
        fitting

    vec4 view_vertex = q_view * vec4(vert, 1);  //  Transform vertices to
         perspective
    gl_Position = q_projection * view_vertex;    // Value of the clip-
        space output position vector

}
```

In the fragment shader, we perform a Gaussian blur on both directions at the same time. We consider a tessellation of the whole screen into even texels and for each fragment, we consider as color the gaussian-weighted mean of the colors of the surrounding points. The surrounding points are obtained by checking the 24 texels around the considered fragment. They are also affected by a blurRadius that allows us to perform a more or less effective blur.

blur.frag

```
#version 330 core

out vec4 frag_color;  // Fragment's color is the output

in vec2 TexCoords;  // Texture coordinates as input from vertex shader

uniform sampler2D ssaoInput;  // SSAO texture as uniform input

// Other parameters for the blur as inputs
uniform float blurRadius;
```

```
11 uniform float sigma;
12 uniform float width;
13 uniform float height;
14
15 // Bool to activate or desactivate the blur from the viewer
16 uniform bool blur_flag;
17
18 void main()
19 {
20
21     vec2 texelSize = 1.0 / vec2(textureSize(ssaoInput, 0));  // Size of a
       texel ( 1 / textureSize)
22     // Accumulators
23     float nColor = 0.0;
24     float weight_total = 0.0;
25
26     if (blur_flag){
27
28
29         for (int x = -2; x <= 2; ++x)  // 2 Loops on horizontal &
           vertical direction
30         {
31             for (int y = -2; y <= 2; ++y)
32             {
33
34                 float weight=exp(-((x*x + y*y)/sigma));  // Compute a
                   gaussian weight
35                 vec2 offset = vec2(float(x) * blurRadius, float(y) *
                   blurRadius) * texelSize;  // Compute the offset around
                    the considered fragment within a texel affected by
                    the blurRadius
36                 nColor += texture(ssaoInput, TexCoords + offset).r *
                   weight;  // Add the weighted color corresponding of
                   the surrondings points
37                 weight_total+=weight;   // Compute the total weight
38
39             }
40         }
41
42         float val = nColor / weight_total;  // Normalise by the total
           weight
43         frag_color = vec4(val, val, val, 1.0);  // Affect it to the
           fragment's color
44     }
45
46     else{
47         // If blur is not activated
48         vec4 color = texture(ssaoInput,TexCoords);  // Take the standard
           color from SSAO
49         frag_color = color;
50
51         }
52 }
```

2 Separable SSAO

For the separable version of SSAO, the main idea is to apply the same thing as for SSAO but instead of doing the sampling inside a hemisphere, we do it only on two directions and one direction at a time. So the main difference will first come from how we generate the samples at the initialisation. Then, the second main difference will be in the fragment shader where we will use two separate loops instead of one inside the other. The vertex shader is exactly the same as for the non-separable SSAO. We compute the occlusion factors in each direction in the fragment shader the same way we did for non-separable SSAO and get the mean between both values at the end.

I will only comment the differences between the new separable fragment shader and the old SSAO fragment shader on the following code.

separable.frag

```
#version 330 core

in vec2 TexCoords;

out vec4 frag_color;

uniform sampler2D gNormal;
uniform sampler2D gPosition;
uniform sampler2D texNoise;

uniform int sampleN;
uniform vec3 samples_x[64];  // Samples for the first direction
uniform vec3 samples_y[64];  // Samples for the second direction

uniform mat4 projection;
uniform float radius;
uniform float width;
uniform float height;

vec2 noiseScale = vec2(width/4.0, height/4.0);

void main(void)
{
    vec3 fragPos = texture(gPosition,TexCoords).xyz;
    vec3 normal = normalize(texture(gNormal, TexCoords).rgb *2 -1);
    vec3 randomVec = texture(texNoise, TexCoords * noiseScale).xyz;

    vec3 tangent   = normalize(randomVec - normal * dot(randomVec, normal
        ));
    vec3 bitangent = cross(normal, tangent);
    mat3 TBN       = mat3(tangent, bitangent, normal);

    float occlusion = 0.0;

    for(int i = 0; i < sampleN; ++i)  // First loop on all the samples
    {

        vec3 sample = TBN * samples_x[i]; // Get the samples in the first
            direction
```

```
38
39          sample = fragPos + sample * radius;
40
41          vec4 offset = vec4(sample, 1.0);
42          offset      = projection * offset;
43          offset.xyz /= offset.w;
44          offset.xyz  = offset.xyz * 0.5 + 0.5;
45
46          float sampleDepth = texture(gPosition, offset.xy).z;
47
48          if (fragPos.z < sampleDepth && abs(fragPos.z - sampleDepth) <
               radius) {
49              occlusion+=1;  // Compute occlusion value in 1 direction
50          }
51
52
53      }
54
55      for(int i = 0; i < sampleN; ++i)  // Second loop on all the samples
56      {
57
58          vec3 sample = TBN * samples_y[i];   // Get the samples in the
               second direction
59
60          sample = fragPos + sample * radius;
61
62          vec4 offset = vec4(sample, 1.0);
63          offset      = projection * offset;
64          offset.xyz /= offset.w;
65          offset.xyz  = offset.xyz * 0.5 + 0.5;
66
67
68          float sampleDepth = texture(gPosition, offset.xy).z;
69
70          if (fragPos.z < sampleDepth && abs(fragPos.z - sampleDepth) <
               radius) {
71              occlusion+=1;   // Compute occlusion value in 1 direction
72          }
73
74
75      }
76
77
78
79      occlusion = 1.0 - (occlusion / (2*float(sampleN)));  // Mean of the 2
           occlusion values
80      frag_color = vec4(occlusion,occlusion,occlusion,1.0);
81
82
83
84  }
```

3 Results

Here are some examples of the results obtained with all the different techniques. By displaying the fps in the viewer, we can also observe than for the same number of samples n, the SSAO and the separable one have similar rates. However, when we increase the number of samples, the separable version is quicker than the non-separable one. Those results are logical considering that the separable version is in $\mathcal{O}(n)$ and the non-separable one in $\mathcal{O}(n^2)$.

FIGURE 1 – Non-separable Version - 8x8 samples

FIGURE 2 – Non-separable Version - 8x8 samples - Blurred

FIGURE 3 – Non-separable Version - 3x3 samples

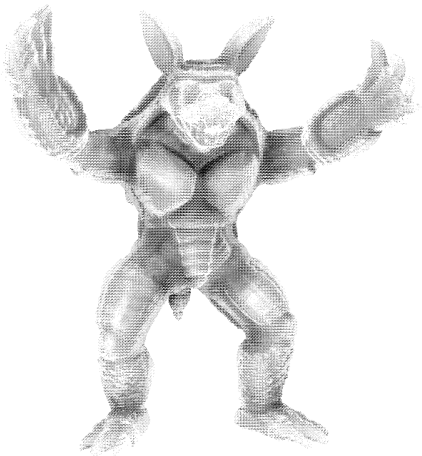

FIGURE 4 – Separable Version - 3+3 samples

FIGURE 5 – Non-separable Version - 6x6 samples

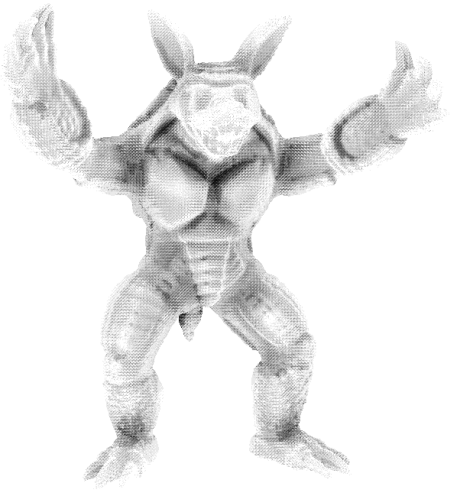

FIGURE 6 – Separable Version - 6+6 samples

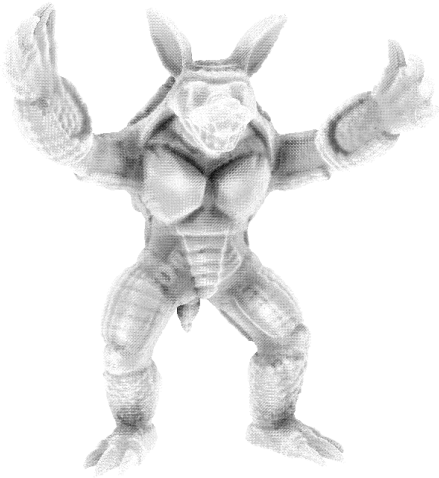

FIGURE 7 – Separable Version - 8+8 samples