Report: SSAO project

Alberto Jovane

December 4, 2018

1 First step: postions and normals

The first step is the rendering of the given model
For the ambient occlusion technique we have chosen, we are gonna store two informations: the normals and one the positions of our model, in two different texture (gNormals and gPosition).

1.1 Vertex shader

In the vertex shader we compute the transfomration for our mdoel, and then we send the results to the fragment shader.
As uniform input we get our projection, model, view matrices (4x4), and the normal matrix (3x3) to modify the position of the normals.
From this inputs, combined with the layout input: vert and normals (taken from the model), we compute our results.
So in output we send the positions trasformed by modelviewprojection, and the associated oriented normals.

```
#version 330 core
layout (location = 0) in vec3 vert;
layout (location = 1) in vec3 normal;

uniform mat4 u_projection;
uniform mat4 u_view;
uniform mat4 u_model;
uniform mat3 u_normal_matrix;

smooth out vec3 N;
smooth out vec3 V;

void main()
{
    vec4 view_vertex = u_view * u_model * vec4(vert, 1);
    V = view_vertex.xyz;
    N = normalize(u_normal_matrix * normal);

    gl_Position = u_projection * view_vertex;

}
```

1.2 Fragment shader

In the fragment shader we get the two inputs from the vertex shader V (positions) and N (normals). And we defined our output that is gonna be saved in the two textures we prepared:
-The position ,as it is.
-For the normal we add one and divide by 2 to obtain something in the range [0, +1] to fit the texture's range of values.

```
#version 330 core
layout (location = 0) out vec3 gNormal;
layout (location = 1) out vec3 gPosition;

smooth in vec3 N;
smooth in vec3 V;

uniform sampler2D tex;

out vec4 frag_color;

void main(void)
{
      gPosition=V;
      gNormal = (normalize(N)+1.0)/2;

}
```

2 Second step: computation of ssao

In this second step we are gonna compute the actual occlusion of each step, starting from the information that we stored in the previous step.

2.1 Vertex shader

In the vertex shader we pre-compute the informations of the coordinates of the quad, that is gonna be sent in output as TexCoords for the fragment shader.
So in output we are gonna define the position of the quad and, for the texCoords, we are gonna consider only the x and y coordinate.
In a similar way as before, we are gonna normalize our positionIn value, to make it fit the texture space.

```
#version 330 core
layout (location = 0) in vec3 vert;

out vec2 TexCoords;

void main(void)
{
    vec2 positionIn = vec2(vert.x,vert.y);
    TexCoords = (positionIn * vec2(0.5)) + vec2(0.5);
    gl_Position = vec4(vert,1.0);
}
```

2.2 Fragment shader

The occlusion of each fragment is computed in the fragment shader.

The inputs are:
-3 textures: the normal, the positon (from the prevoius step), and also one noise texture, that is gonna be useful to introduce some randomoness in our samples.
-Associated with this textures we have the TexCoords that we computed in vertex shader.
-We also need the projection matrix, in order to move our offest from to clip-space.
-In addition to those input we also need the actual samples that we are gonna use to compute the actual occlusion of the fragment.
-And the sampleN which define how many of the 64 samle are gonna be used.
(Those samples are randomly generated in a emisphere, we define 64 randomly generated samples in our code, we put a bigger number of sample closeer to the origin of the actual fragment. The random texture is then used to add rotation to our sample (around z), in order to have much variability and better results.)
-We also add the radius parameter, that define the actual size of the emisphere, to give more customability to our shader.
-Finaly we also get as input the width and the heigth of our screen to adapt the noise texture on the screen dimension.

Computation:
The first thing we do is extracting the informations from the textures we have.
(Notice that the normal is multiplied by 2 and subtracted by 1, to restorethe original value.)

After that wecompute the TBN matrix to transform any vector from tangent space to view space. Here we are gonna use the normal and the random value (that we extract from the nosie texture). We need to change because when we generate the samples we consider a tangent space, that means that we took as z coordinate the normal vector.

The next step is teh actual loop in which we compute the occlusion factor. In order to set how many sample to use, our loop is based on the sampleN we set as input.
Inside the loop we:
-transform each sample from tangent to view space.
-then find the acutal position of the sample around the fragment (weighted by the radius).
-then from this we compute the offset that we are gonna use to extract the sampleDepth from the gPosition texture.
-after we retrieve this depth value (from view space) we compare it to our fragment depth, and if it is lower then we increase the occlusion of our fragment.

At the end we normalize our occlusion value, and we color our otuput buffer and also in this case is going to be printed into a texture.

```
#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[64];
uniform mat4 projection;

uniform float radius;

uniform float width;
uniform float height;

//tile noise texture over screen based on screen dimensions divided by noise size
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)
    {
        // get sample position
        vec3 sample = TBN * samples[i]; // From tangent to view-space

        sample = fragPos + sample * radius;

        vec4 offset = vec4(sample, 1.0);
        offset      = projection * offset;    // from view to clip-space
        offset.xyz /= offset.w;               // perspective divide
        offset.xyz  = offset.xyz * 0.5 + 0.5; // transform to range 0.0 - 1.0


        float sampleDepth = texture(gPosition, offset.xy).z;

        if (fragPos.z < sampleDepth && abs(fragPos.z - sampleDepth) < radius) {
            occlusion+=1;
        }


    }

    occlusion = 1.0 - (occlusion / sampleN);
    frag_color = vec4(occlusion,occlusion,occlusion,1.0);
}
```

3 Alternative second step: separable version

The separable version of the SSAO works more or less in a similar way to the original one. The main different is that the ambient occlusion is approximated by the x-term and y-term, each one in 1 dimension.
The only requirement was for us to change the computation of the samples,in the separable version we don't use anymore a emisphere but we take the samples from two planes.
This semlification mixed with randomoness can lead to good results, actually with less definition of the previous techinque, but still acceptable after the blur.

3.1 Vertex shader

The vertex shader works in a similar way of the previous ones.
Again we send the position and the coordinates of our quad.

```
#version 330 core
layout (location = 0) in vec3 vert;

out vec2 TexCoords;

void main(void)
{
    vec2 positionIn = vec2(vert.x,vert.y);
    TexCoords = (positionIn * vec2(0.5)) + vec2(0.5);
    gl_Position = vec4(vert,1.0);
}
```

3.2 Fragment shader

The fragment shader is mostly similar to the noraml version.
But this time we get two arrays of 3d samples, one for the x direction and one for the y direction.
As before we esxtract the informations from the different textures, and we compute the TBN matrix.
Then the things are slightly different, we have two loops:
The first one is on the x axis, and the second one is around the y axes.
So we will generate our sampleNxsampleN texel aroud the fragment, and compute the occlusion base on this samples.
The occlusion is then computed in the same way as before in the two loops, and at the end is normalized based on the two loops.

```
#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];
uniform vec3 samples_y[64];
uniform mat4 projection;

uniform float radius;

uniform float width;
```

```
uniform float height;

//tile noise texture over screen based on screen dimensions divided by noise size
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 < 9; ++i)
    {
        // get sample position
        vec3 sample = TBN * samples_x[i]; // From tangent to view-space

        sample = fragPos + sample * radius;

        vec4 offset = vec4(sample, 1.0);
        offset      = projection * offset;    // from view to clip-space
        offset.xyz /= offset.w;               // perspective divide
        offset.xyz  = offset.xyz * 0.5 + 0.5; // transform to range 0.0 - 1.0


        float sampleDepth = texture(gPosition, offset.xy).z;

        if (fragPos.z < sampleDepth && abs(fragPos.z - sampleDepth) < radius) {
            occlusion+=1;//distance(normal, s_normal);
        }
    }
    for(int i = 0; i < sampleN; ++i)
    {
        // get sample position
        vec3 sample = TBN * samples_y[i]; // From tangent to view-space

        sample = fragPos + sample * radius;

        vec4 offset = vec4(sample, 1.0);
        offset      = projection * offset;    // from view to clip-space
        offset.xyz /= offset.w;               // perspective divide
        offset.xyz  = offset.xyz * 0.5 + 0.5; // transform to range 0.0 - 1.0

        float sampleDepth = texture(gPosition, offset.xy).z;

        if (fragPos.z < sampleDepth && abs(fragPos.z - sampleDepth) < radius) {
            occlusion+=1;//distance(normal, s_normal);
        }
    }
    occlusion = 1.0 - (occlusion / (2*float(sampleN)));
    frag_color = vec4(occlusion,occlusion,occlusion,1.0);
}
```

4 Third step: blurring

The blurring part is important to get rid of the artifacts that we get from the previous computations. The problem with the general ambient occlusion is that, in order to reduce the number of samples, we introduce noise, and this noise generate a noisy result. That is why a blurring step is needed. We decided to implement a gaussian blur.

4.1 Vertex shader

The vertex shader works in a similar way of the previous ones.

```
#version 330 core
layout (location = 0) in vec3 vert;
layout (location = 1) in vec2 texCoords;

uniform mat4 q_projection;
uniform mat4 q_view;

out vec2 TexCoords ;

void main(void)  {

    TexCoords = texCoords;

    vec4 view_vertex = q_view * vec4(vert, 1);

    gl_Position = q_projection * view_vertex;
}
```

4.2 Fragment shader

In the fragment shader we get as input the texture where all the occlusion information are stored. Then we get also some parameter to tune the blur (sigma and blur radius)
And finally as input we also have a boolean $blur_flag$, to determine if the blur is active or not.

The gaussian blur give higher weigths to the samples closer to our fragment.
So the main idea is having a double loop, one inside the other, to get the blur in both directions x and y.
We so for each iteration we generate the cordinate (offset) for extracting the value from the ssao texture, and then base on the weigth we modify the fragment color.
At the end we normalize the results and we send them in output, to finally print them on the quad.
We compute our weights based on the x and y values of our loop, in order to obtain biger values when we get closer to the fragment.
The x and y values, are also the one that define from where we are gonna extract the sample (the distance is based on the radius).
Having a loop inside a loop allo us to explore the area in the plane around our fragment without using two separate loops.

```
#version 330 core
out vec4 frag_color;

in vec2 TexCoords;

uniform sampler2D ssaoInput;

uniform float blurRadius;
uniform float sigma;
uniform float width;
uniform float height;
```

```
uniform bool blur_flag;

void main()
{
    vec2 texelSize = 1.0 / vec2(textureSize(ssaoInput, 0));
    float nColor = 0.0;
    float weight_total = 0.0;
    if (blur_flag){

        for (int x = -2; x <= 2; ++x)
        {
            for (int y = -2; y <= 2; ++y)
            {
                float weight=exp(-((x*x + y*y)/sigma));
                vec2 offset = vec2(float(x) * blurRadius, float(y) * blurRadius) * te
                nColor += texture(ssaoInput, TexCoords + offset).r * weight;
                weight_total+=weight;

            }
        }
        float val = nColor / weight_total;
        frag_color = vec4(val, val, val, 1.0);
    }
    else{
        vec4 color = texture(ssaoInput,TexCoords);
        frag_color = color;
        }
}
```