Computer Graphics II
– SSAO (Screen-space Ambient Occlusion)

Kai Lawonn

Introduction

- In the basic lighting lecture: ambient lighting
- Ambient lighting is a fixed light constant added to the overall lighting of a scene to simulate the scattering of light
- In reality, light scatters in all directions with varying intensities → indirectly lit parts should also have varying intensities (instead of a constant)
- Ambient occlusion tries to approximate indirect lighting by darkening creases, holes and surfaces that are close to each other
- These areas are largely occluded by surrounding geometry and thus light rays have less places to escape → appear darker

Introduction

- Example image of a scene with and without screen-space ambient occlusion (SSAO)
- Notice how especially between the creases the (ambient) light is more occluded

Introduction

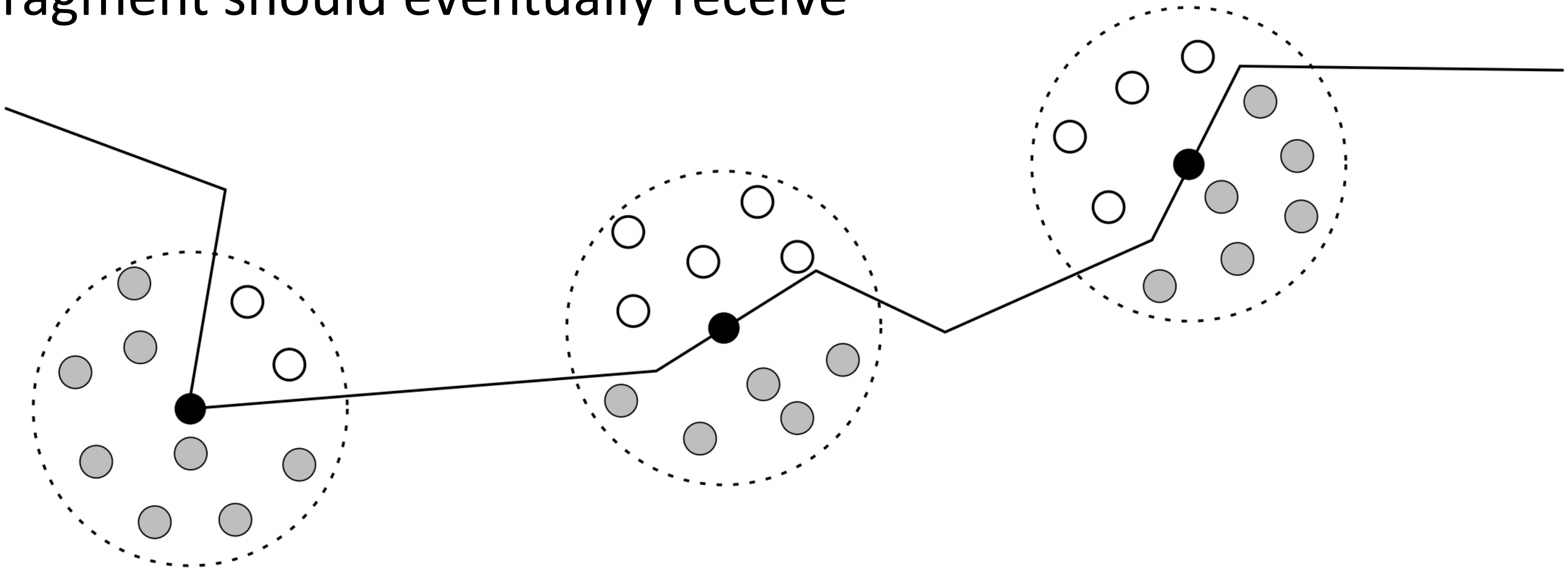
- Ambient occlusion techniques expensive (take surrounding geometry into account)
- Could shoot a large number of rays for each point in space to determine its amount of occlusion → computationally infeasible for real-time solutions
- 2007 Crytek published a technique called screen-space ambient occlusion (SSAO) for use in their title Crysis
- SSAO uses a scene’s depth in screen-space to determine the amount of occlusion instead of real geometrical data
- It is fast compared to real ambient occlusion and gives plausible results

Introduction

- Basics are simple: for each fragment on a screen-filled quad, calculate an occlusion factor based on the fragment's surrounding depth values
- Occlusion factor is used to reduce or nullify the fragment's ambient lighting component
- Occlusion factor obtained by taking multiple depth samples in a sphere (sample kernel surrounding the fragment position and compare samples with the current fragment's depth value)
- Number of samples that have a higher depth value than the fragment's depth represents the occlusion factor

Introduction

- Gray depth samples are inside geometry contribute to the total occlusion factor
- The more samples we find inside geometry, the less ambient lighting the fragment should eventually receive



Introduction

- Quality and precision relates to the number of surrounding samples
- If sample count too low the precision reduces (artifact ‘banding’)
- If too high lose performance
- Reduce amount of samples by some randomness into the sample kernel
- Randomly rotate kernel each fragment
  → high quality results with smaller samples
- Introduces a noticeable noise pattern (fix by blurring the results)

low sample 'banding' random rotation = noise + blur = acceptable

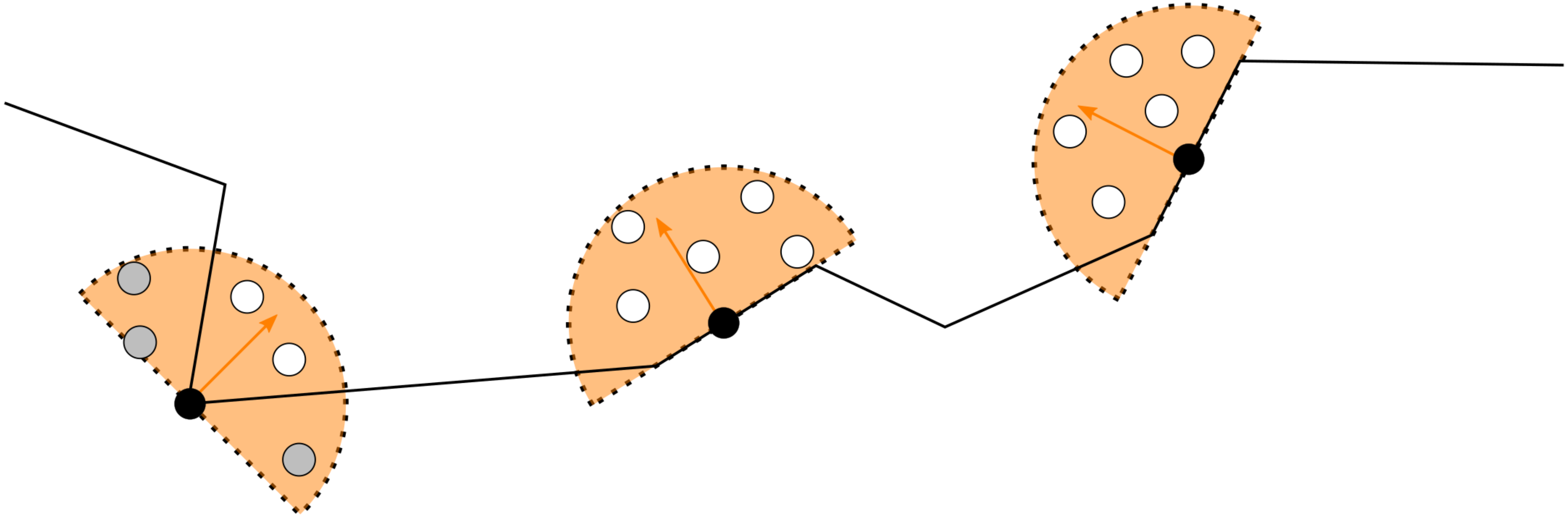
Images by John Chapman

Introduction

- SSAO developed by Crytek had a certain visual style
- Because the sample kernel used was a sphere, it caused flat walls to look gray as half of the kernel samples end up being in the surrounding geometry

Introduction

- For that, will not use a sphere sample kernel, but rather a hemisphere sample kernel oriented along a surface's normal vector:



Sample Buffers

Sample Buffers

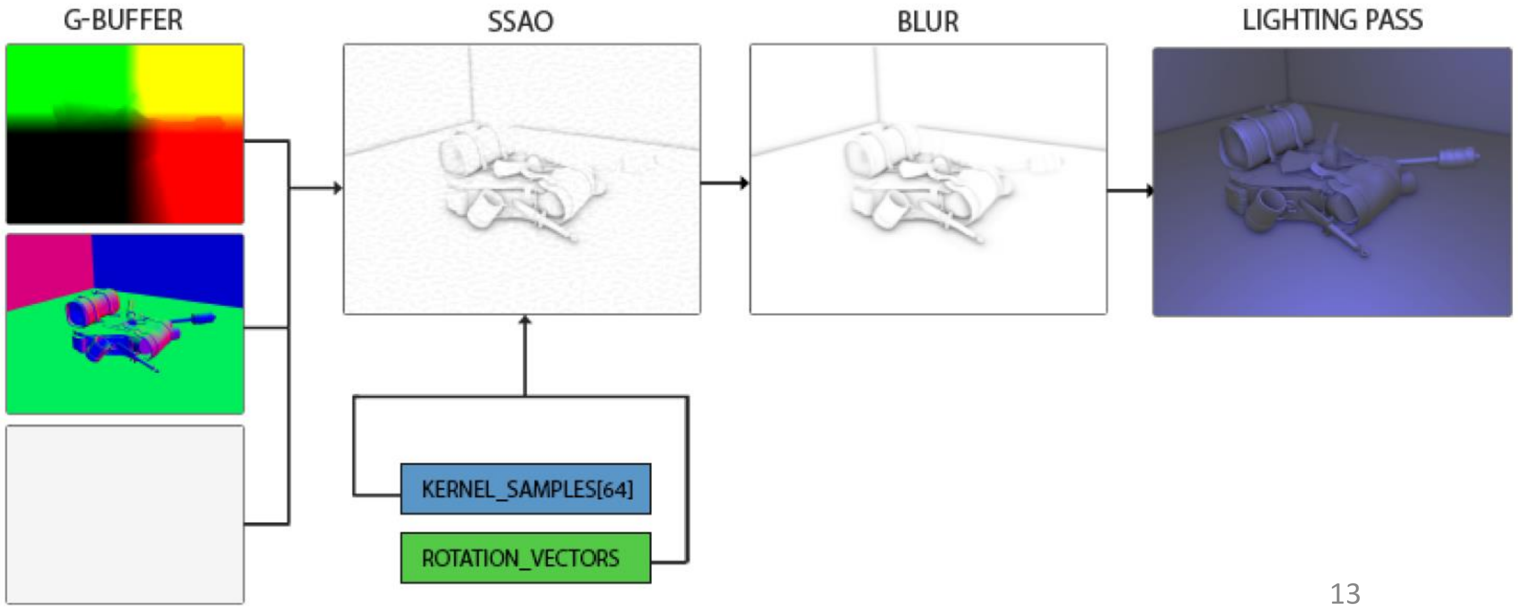
- SSAO requires geometrical info to determine the occlusion factor:
  - A per-fragment position vector
  - A per-fragment normal vector
  - A per-fragment albedo color
  - A sample kernel
  - A per-fragment random rotation vector used to rotate the sample kernel

Sample Buffers

- Per-fragment view-space position to orient a sample hemisphere kernel around the fragment’s view-space surface normal → use to sample position buffer texture at varying offsets
- Per-fragment kernel sample to compare their depth with the original fragment’s depth to determine the amount of occlusion
- Resulting occlusion factor is then used to limit the final ambient lighting component
- Including a per-fragment rotation vector to significantly reduce the number of samples

Sample Buffers

- SSAO screen-space technique, calculate on fragment on a screen-filled 2D quad, have no geometrical information of the scene
- Render geometrical per-fragment data into screen-space textures
- Similar to deferred rendering and for that reason → SSAO is perfectly suited in combination with deferred rendering (already have position and normal vectors in the G-buffer)



Implement SSAO on top of a slightly simplified version of the deferred renderer from the previous lecture

Sample Buffers

- Already have per-fragment position and normal data available from the G-buffer, fragment shader of the geometry stage is simple:

```
#version 330 core
layout (location = 0) out vec4 gPosition;
layout (location = 1) out vec3 gNormal;
layout (location = 2) out vec4 gAlbedoSpec;
in vec2 TexCoords;
in vec3 FragPos;
in vec3 Normal;
void main()
{
   // store the fragment position vector in the first gbuffer texture
   gPosition = FragPos;
   // also store the per-fragment normals into the gbuffer
   gNormal = normalize(Normal);
   // and the diffuse per-fragment color, ignore specular
   gAlbedoSpec.rgb = vec3(0.95);
}
```

Sample Buffers

- SSAO is a screen-space technique where occlusion is calculated based on the visible view → implement the algorithm in view-space
- Thus, FragPos (supplied by geometry stage’s vertex shader) transformed to view space
- Further calculations in view-space → make sure the G-buffer’s positions and normals are in view-space (multiplied by the view matrix as well)

Possible to reconstruct the actual position vectors from depth values (see blog by Matt Pettineo)

Requires extra calculations in the shaders, but saves to store position data in the G-buffer which costs a lot of memory

https://mynameismjp.wordpress.com/2010/09/05/position-from-depth-3/

Sample Buffers

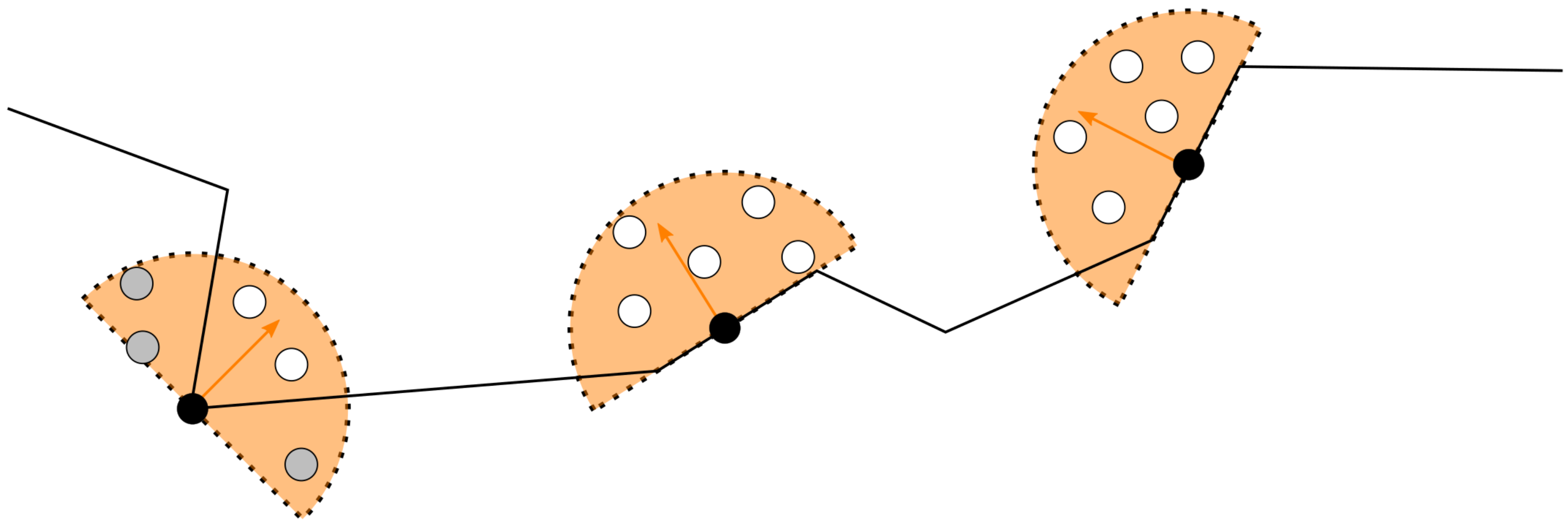
- The gPosition colorbuffer texture is configured as follows:

```
glGenTextures(1, &gPosition);
glBindTexture(GL_TEXTURE_2D, gPosition);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA16F, SCR_WIDTH, SCR_HEIGHT, 0,
        GL_RGBA, GL_FLOAT, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

Normal-Oriented Hemisphere

Normal-Oriented Hemisphere

- Need to generate samples oriented along the normal of a surface → form a hemisphere
- Difficult to generate a sample kernel for each surface normal direction → generate sample kernel in tangent space (normal vector pointing in the positive z direction)



Normal-Oriented Hemisphere

- Assuming having a unit hemisphere, obtain a sample kernel with a maximum of 64 sample values as follows:

```
std::uniform_real_distribution<float> randomFloats(0.0, 1.0);
std::default_random_engine generator;
std::vector<glm::vec3> ssaoKernel;
for (unsigned int i = 0; i < 64; ++i)
{
   glm::vec3 sample(randomFloats(generator) * 2.0 - 1.0,
                    randomFloats(generator) * 2.0 - 1.0,
                    randomFloats(generator));
   sample = glm::normalize(sample);
   sample *= randomFloats(generator);
   ssaoKernel.push_back(sample);
   …
```

Normal-Oriented Hemisphere

- Currently, all samples are randomly distributed in the sample kernel, but better place a larger weight on occlusions close to the actual fragment as to distribute the kernel samples closer to the origin:

```
std::uniform_real_distribution<float> randomFloats(0.0, 1.0);
std::default_random_engine generator;
std::vector<glm::vec3> ssaoKernel;
for (unsigned int i = 0; i < 64; ++i)
{
   glm::vec3 sample(randomFloats(generator) * 2.0 - 1.0,
                    randomFloats(generator) * 2.0 - 1.0,
                    randomFloats(generator));
   sample = glm::normalize(sample);
   sample *= randomFloats(generator);
   ssaoKernel.push_back(sample);
     …
     float scale = float(i) / 64.0;

     // scale samples s.t. they're more aligned to center of kernel
     scale = lerp(0.1f, 1.0f, scale * scale);
     sample *= scale;
     ssaoKernel.push_back(sample);
}
```

Normal-Oriented Hemisphere

- Currently, all samples are randomly distributed in the sample kernel, but better place a larger weight on occlusions close to the actual fragment as to distribute the kernel samples closer to the origin:

```
std::uniform_real_distribution<float> randomFloats(0.0, 1.0);
std::default_random_engine generator;
std::vector<glm::vec3> ssaoKernel;
for (unsigned int i = 0; i < 64; ++i)
{
   glm::vec3 sample(randomFloats(generator) * 2.0 - 1.0,
                    randomFloats(generator) * 2.0 - 1.0,
                    randomFloats(generator));
   sample = glm::normalize(sample);
   sample *= randomFloats(generator);
   ssaoKernel.push_back(sample);
    …
    float scale = float(i) / 64.0;

    // scale samples s.t. they're more aligned to center of kernel
    scale = lerp(0.1f, 1.0f, scale * scale);
    sample *= scale;
    ssaoKernel.push_back(sample);
}
```

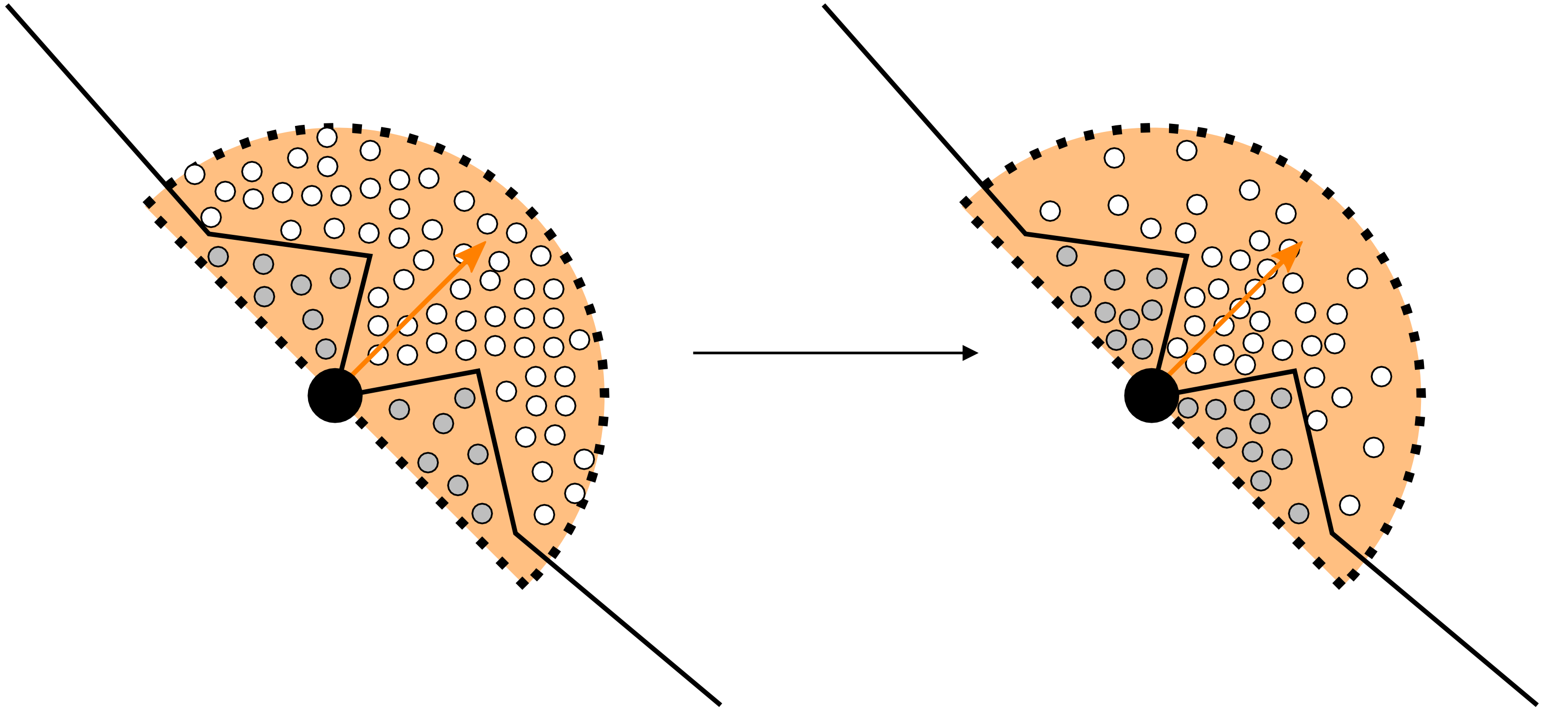
Normal-Oriented Hemisphere

- Where lerp is defined as:

```
float lerp(float a, float b, float f)
{
    return a + f * (b - a);
}
```

Normal-Oriented Hemisphere

- Gives kernel distribution that places most samples closer to its origin



Normal–Oriented Hemisphere

- Each of the kernel samples will be used to offset the view-space fragment position to sample surrounding geometry
- Need quite a lot of samples in view-space in order to get realistic results, which might be too heavy on performance
- However, can introduce some semi-random rotation/noise on a per-fragment basis we can significantly reduce the number of samples required

Random Kernel Rotations

Random Kernel Rotations

- By introducing some randomness onto the sample kernels → reduce the number of samples necessary to get good results
- Could create a random rotation vector for each fragment of a scene → memory-consuming
- Better to create a small texture of random rotation vectors that tile over the screen

Random Kernel Rotations

- Create a 4x4 array of random rotation vectors oriented around the tangent-space surface normal:

```
std::vector<glm::vec3> ssaoNoise;
for (unsigned int i = 0; i < 16; i++)
{
   glm::vec3 noise(randomFloats(generator) * 2.0 - 1.0,
                   randomFloats(generator) * 2.0 - 1.0, 0.0f);
   ssaoNoise.push_back(noise);
}
```

Random Kernel Rotations

- Then create a 4x4 texture that holds the random rotation vectors (make sure to set wrapping to GL_REPEAT → properly tiles over the screen)

```
unsigned int noiseTexture; glGenTextures(1, &noiseTexture);
glBindTexture(GL_TEXTURE_2D, noiseTexture);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F, 4, 4, 0, GL_RGB, GL_FLOAT,
        &ssaoNoise[0]);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

The SSAO Shader

The SSAO Shader

- SSAO shader runs on a 2D screen-filled quad that calculates the occlusion value for fragments (for use in the final lighting shader)
- To store the result of the SSAO stage, create another FBO (red value):

```
unsigned int ssaoFBO;
glGenFramebuffers(1, &ssaoFBO);
glBindFramebuffer(GL_FRAMEBUFFER, ssaoFBO);
unsigned int ssaoColorBuffer;
glGenTextures(1, &ssaoColorBuffer);
glBindTexture(GL_TEXTURE_2D, ssaoColorBuffer);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RED, SCR_WIDTH, SCR_HEIGHT, 0, GL_RED,
            GL_FLOAT, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D,
ssaoColorBuffer, 0);
```

The SSAO Shader

- The complete process for rendering SSAO then looks a bit like this:

```
// geometry pass: render stuff into G-buffer
glBindFramebuffer(GL_FRAMEBUFFER, gBuffer);
…
glBindFramebuffer(GL_FRAMEBUFFER, 0);
// use G-buffer to render SSAO texture
glBindFramebuffer(GL_FRAMEBUFFER, ssaoFBO);
glClear(GL_COLOR_BUFFER_BIT);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, gPosition);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, gNormal);
glActiveTexture(GL_TEXTURE2);
glBindTexture(GL_TEXTURE_2D, noiseTexture);
…
```

```
…
shaderSSAO.use();
SendKernelSamplesToShader();
shaderSSAO.setMat4("projection", projection);
RenderQuad();
glBindFramebuffer(GL_FRAMEBUFFER, 0);
// lighting pass: render scene lighting
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
shaderLightingPass.use();
…
glActiveTexture(GL_TEXTURE3);
glBindTexture(GL_TEXTURE_2D, ssaoColorBuffer);
…
RenderQuad();
```

The SSAO Shader

- ShaderSSAO takes as input the relevant G-buffer textures, the noise texture and the normal-oriented hemisphere kernel samples:

```
#version 330 core
out float FragColor;
in vec2 TexCoords;

uniform sampler2D gPosition;
uniform sampler2D gNormal;
uniform sampler2D texNoise;
uniform vec3 samples[64];
uniform mat4 projection;
// tile noise texture over screen, based on screen dimensions / noise size
const vec2 noiseScale = vec2(800.0/4.0, 600.0/4.0); // screen = 800x600

void main()
{
 …
}
```

The SSAO Shader

- noiseScale: want to tile noise texture all over the screen, but TexCoords vary between [0,1], texNoise texture will not tile at all

```
#version 330 core
out float FragColor;
in vec2 TexCoords;

uniform sampler2D gPosition;
uniform sampler2D gNormal;
uniform sampler2D texNoise;
uniform vec3 samples[64];
uniform mat4 projection;
// tile noise texture over screen, based on screen dimensions / noise size
const vec2 noiseScale = vec2(800.0/4.0, 600.0/4.0); // screen = 800x600

void main()
{
 …
}
```

The SSAO Shader

- Calculate by how much have to scale the TexCoords coordinates by dividing the screen’s dimensions by the noise texture size:

```
vec3 fragPos = texture(gPosition, TexCoords).xyz;
vec3 normal = texture(gNormal, TexCoords).rgb;
vec3 randomVec = texture(texNoise, TexCoords * noiseScale).xyz;
```

The SSAO Shader

- Set tiling parameters of texNoise to GL_REPEAT → random values will be repeated all over the screen
- Together with fragPos and normal vector, have enough data to create a TBN matrix to transform any vector from tangent-space to view-space:

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

The SSAO Shader

- Using the Gramm-Schmidt process to create an orthogonal basis, each time slightly tilted based on the value of randomVec
- Note, because using a random vector for constructing the tangent vector, there is no need to have the TBN matrix exactly aligned to the geometry's surface(no need for per-vertex tangent and bitangent vectors)

The SSAO Shader

- Next, iterate over each kernel samples
- Transform the samples from tangent to view-space
- Add them to the current fragment position and compare the fragment position’s depth with the sample depth stored in the view-space position buffer:

```
float occlusion = 0.0;
for(int i = 0; i < kernelSize; ++i)
{
   // get sample position
   vec3 sample = TBN * samples[i]; // from tangent to view-space
   sample = fragPos + sample * radius;
   …
}
```

The SSAO Shader

- Next, transform sample to screen-space to sample position/depth value as if we were rendering its position directly to the screen
- As the vector is currently in view-space, transform it to clip-space first using the projection matrix uniform:

```
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]
```

The SSAO Shader

- Use them to sample the position texture:

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

The SSAO Shader

- Then check if the sample's current depth value is larger than the stored depth value and if so, add to the final contribution factor:

```
occlusion += (sampleDepth >= sample.z + bias ? 1.0 : 0.0);
```

- Add a small bias to the original fragment's depth value (set to 0.025 in the example)
- A bias is not always necessary, but helps visually tweak the SSAO effect and solves acne effects that might occur based on the scene's complexity

The SSAO Shader

- Not finished yet, still an issue
- If a tested fragment is aligned close to the edge of a surface, it will also consider depth values of surfaces behind the test surface → these values will (incorrectly) contribute to the occlusion factor
- Solve by a range check:

without range check

with range check

Images by John Chapman

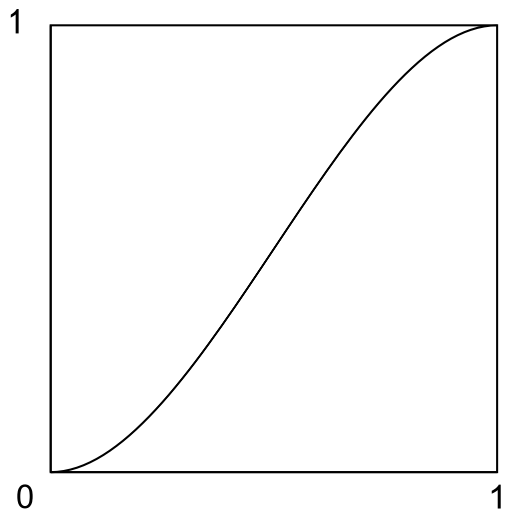
The SSAO Shader

- Range check ensures a fragment contributes to the occlusion factor if its depth values is within the sample's radius
- We change the last line to:

```
float rangeCheck = smoothstep(0.0, 1.0, radius / abs(fragPos.z - sampleDepth));
occlusion+= (sampleDepth >= sample.z + bias ? 1.0 : 0.0) * rangeCheck;
```

- Smoothstep:

```
float smoothstep(float edge0, float edge1, float x)
{
   t = clamp((x - edge0) / (edge1 - edge0), 0.0, 1.0);
   return t * t * (3.0 - 2.0 * t);
}
```



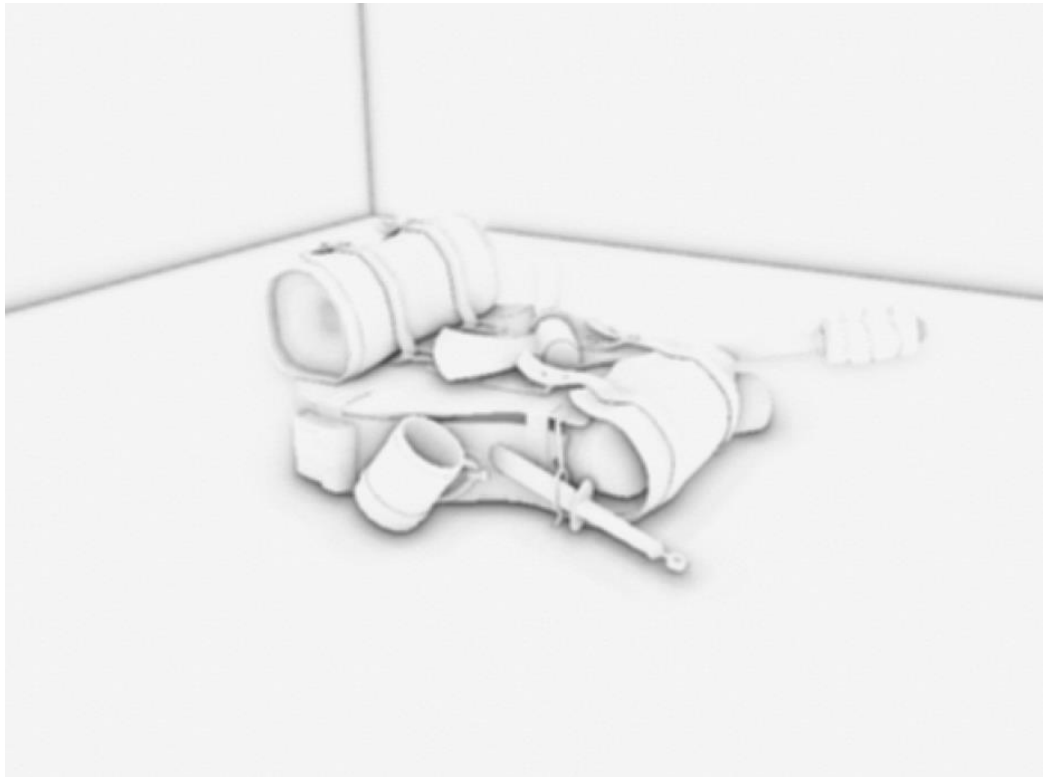
The SSAO Shader

- Final step, normalize the occlusion contribution by the size of the kernel and output the results
- Note, we subtract the occlusion factor from 1.0 → directly use the occlusion factor to scale the ambient lighting component

```
}
occlusion = 1.0 - (occlusion / kernelSize);
FragColor = occlusion;
```

The SSAO Shader

- Ambient occlusion shader produces the following texture:



Ambient Occlusion Blur

Ambient Occlusion Blur

- Between the SSAO pass and the lighting pass, must blur the SSAO texture → create another FBO for storing the blur result:

```
unsigned int ssaoBlurFBO, ssaoColorBufferBlur;
glGenFramebuffers(1, &ssaoBlurFBO);
glBindFramebuffer(GL_FRAMEBUFFER, ssaoBlurFBO);
glGenTextures(1, &ssaoColorBufferBlur);
glBindTexture(GL_TEXTURE_2D, ssaoColorBufferBlur);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RED, SCR_WIDTH, SCR_HEIGHT, 0, GL_RED,
       GL_FLOAT, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D,
       ssaoColorBufferBlur, 0);
```

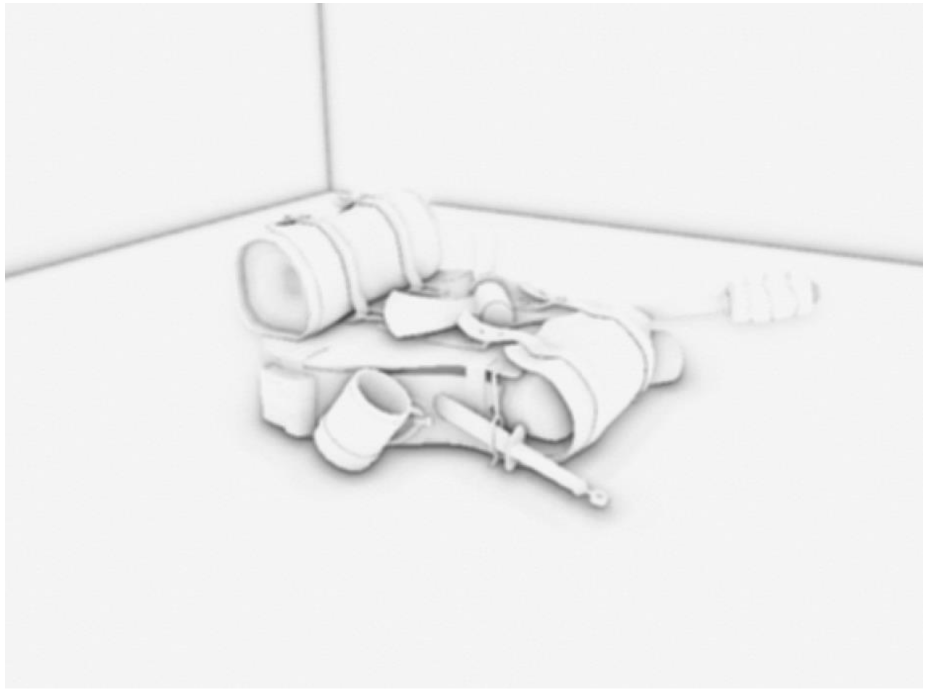
Ambient Occlusion Blur

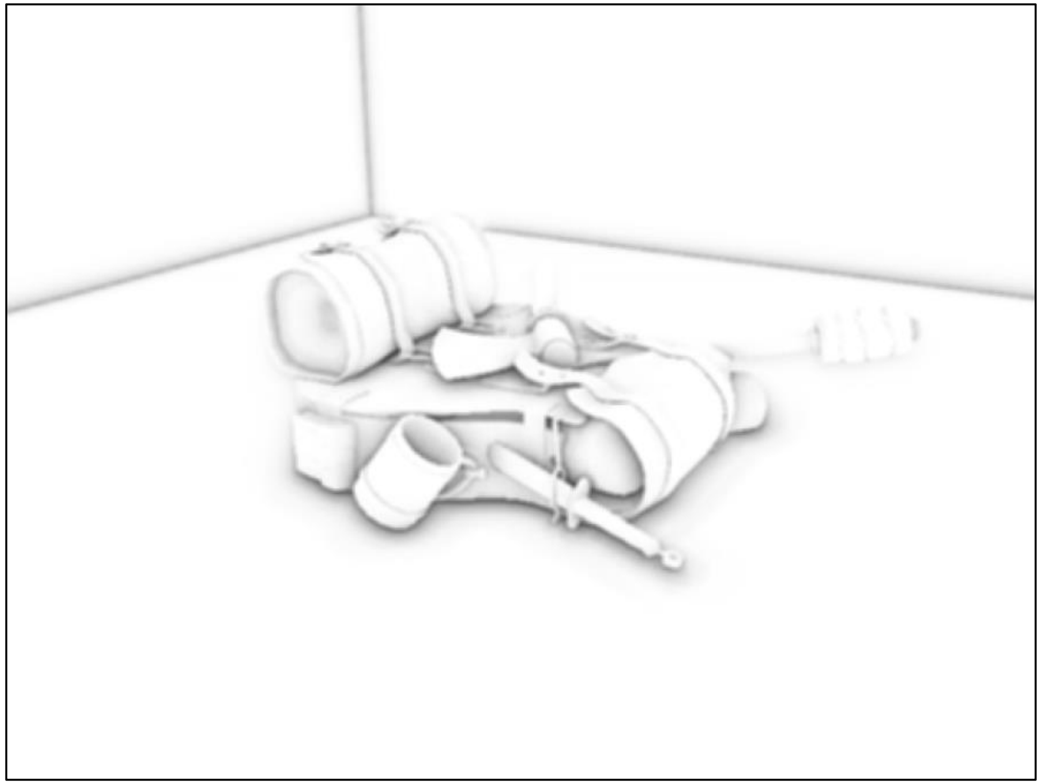
- Tiled random vector texture gives us a consistent randomness, use this property as an advantage to create a very simple blur shader:

```
#version 330 core
out float FragColor;

in vec2 TexCoords;

uniform sampler2D ssaoInput;

void main()
{
    vec2 texelSize = 1.0 / vec2(textureSize(ssaoInput, 0));
    float result = 0.0;
    for (int x = -2; x < 2; ++x)
    {
        for (int y = -2; y < 2; ++y)
        {
            vec2 offset = vec2(float(x), float(y)) * texelSize;
            result += texture(ssaoInput, TexCoords + offset).r;
        }
    }
    FragColor = result / (4.0 * 4.0);
}
```

The SSAO Shader

- Results in a simple, but effective blur:





Applying Ambient Occlusion

Applying Ambient Occlusion

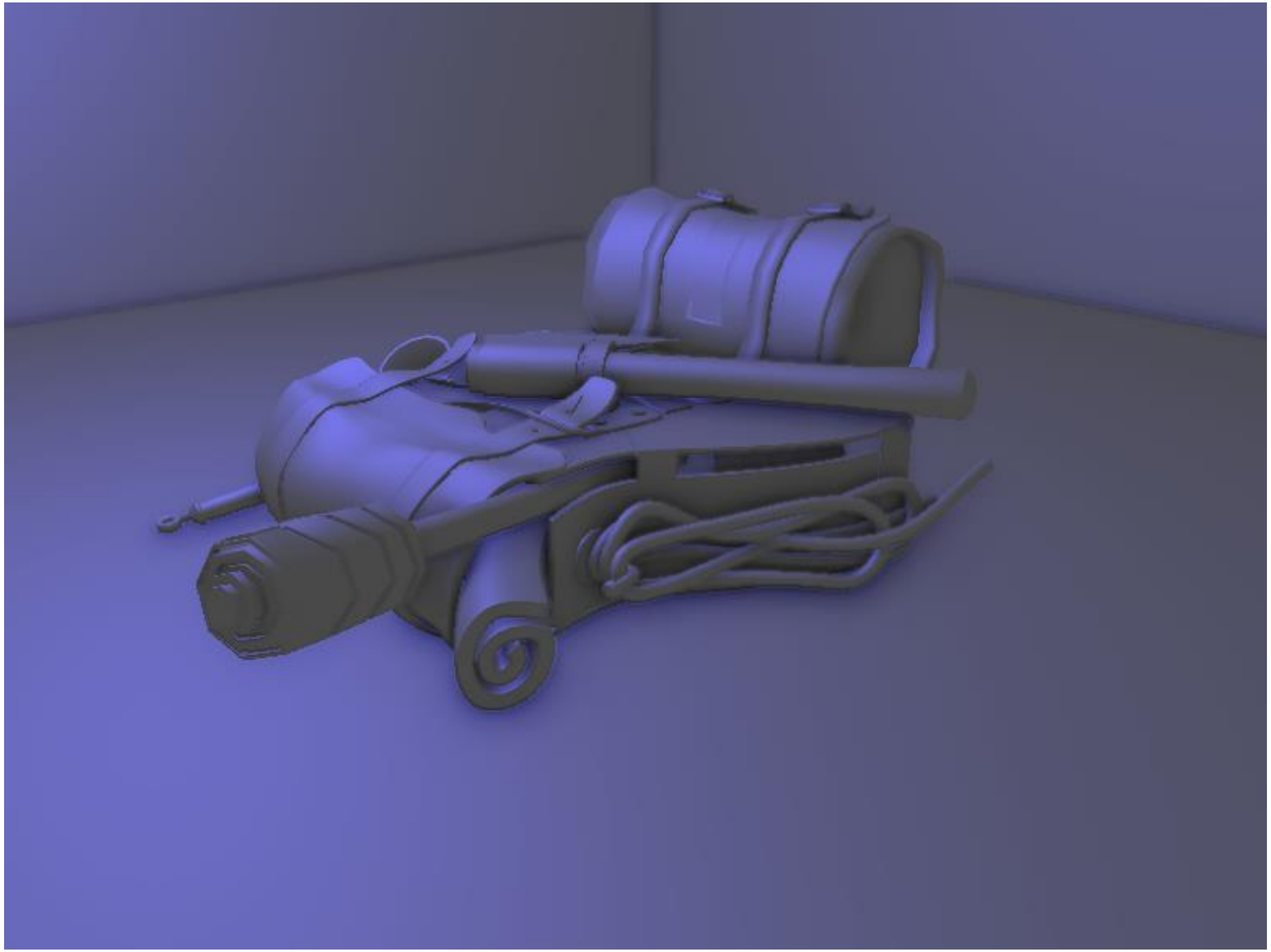
- Occlusion factors to the lighting equation: multiply the per-fragment ambient occlusion factor to lighting's ambient component:

```
#version 330 core
out vec4 FragColor;
in vec2 TexCoords;
uniform sampler2D gPosition;
uniform sampler2D gNormal;
uniform sampler2D gAlbedo;
uniform sampler2D ssao;
struct Light {
    vec3 Position;
    vec3 Color;
    float Linear;
    float Quadratic;
};
uniform Light light;
void main()
{
// retrieve data from gbuffer
vec3 FragPos = texture(gPosition, TexCoords).rgb;
vec3 Normal = texture(gNormal, TexCoords).rgb;
vec3 Diffuse = texture(gAlbedo, TexCoords).rgb;
float AmbientOcclusion = texture(ssao, TexCoords).r;
```

```
// then calculate lighting as usual
vec3 ambient = vec3(0.3 * Diffuse * AmbientOcclusion);
vec3 lighting  = ambient;
vec3 viewDir  = normalize(-FragPos); // viewpos is (0.0.0)

// diffuse
vec3 lightDir = normalize(light.Position - FragPos);
vec3 diffuse = max(dot(Normal, lightDir), 0.0) * Diffuse * light.Color;
// specular
vec3 halfwayDir = normalize(lightDir + viewDir);
float spec = pow(max(dot(Normal, halfwayDir), 0.0), 8.0);
vec3 specular = light.Color * spec;

// attenuation
float distance = length(light.Position - FragPos);
float attenuation = 1.0 / (1.0 + light.Linear * distance + light.Quadratic
* distance * distance);
diffuse *= attenuation;
specular *= attenuation;
lighting += diffuse + specular;
FragColor = vec4(lighting, 1.0);}
```

F5…

- … very nice!



Note

- SSAO is a highly customizable effect that relies heavily on tweaking its parameters based on the type of scene
- There is no perfect combination of parameters for every type of scene
- Some scenes only work with a small radius, while some scenes require a larger radius and a larger sample count for it to look realistic
- The current demo uses 64 samples which is a bit much, play around with a smaller kernel size and try to get good results

Note

- Some parameters: using uniforms: kernel size, radius, bias and/or the size of the noise kernel
- Final occlusion value to a user-defined power to increase its strength:

```
occlusion = 1.0 - (occlusion / kernelSize);
FragColor = pow(occlusion, power);
```

- Try different scenes and parameters for SSAO
- SSAO is a subtle effect that is not too clearly noticeable → adds a great deal of realism to properly lighted scenes

Questions???