Laboratory work 9

1 Objectives

The objective of this laboratory is to briefly describe how to add basic shadows to your OpenGL 4 application using the **Shadow Mapping** technique.

2 Shadow Mapping

The lighting techniques discussed in previous laboratories work great for illuminating objects. However, the lighting effect obtained takes no account of occlusions. In the real world, when an object is not directly hit by rays emitted by a light source due to obstructions (other objects), then that object is in shadow. Thus, shadows add a very important degree of realism to our OpenGL scenes, also allowing for better depth perception (Figure 1 and Figure 2).

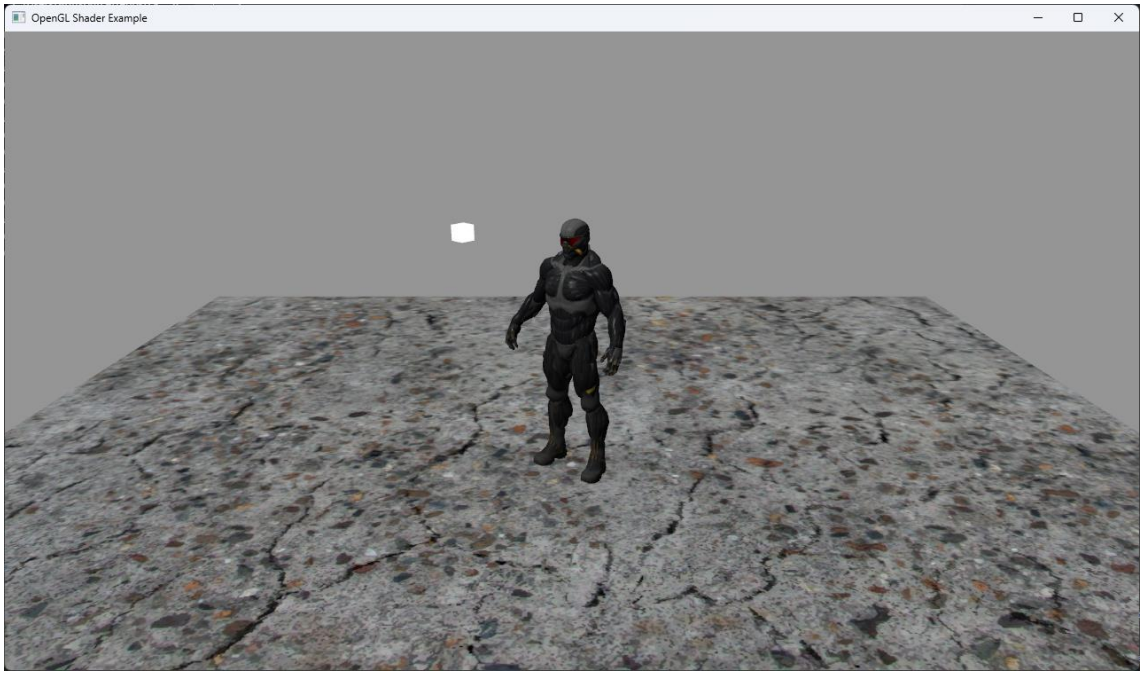

Figure 1 – Rendering without shadows

There is a plethora of modern techniques that can be employed to produce shadows (shadow computation) and all of them can be implemented in OpenGL. However, there are no perfect real-time algorithms for this, each having each its own, advantages and disadvantages. For this laboratory, we will cover a technique that offers decent results and is easy to implement, namely shadow mapping.

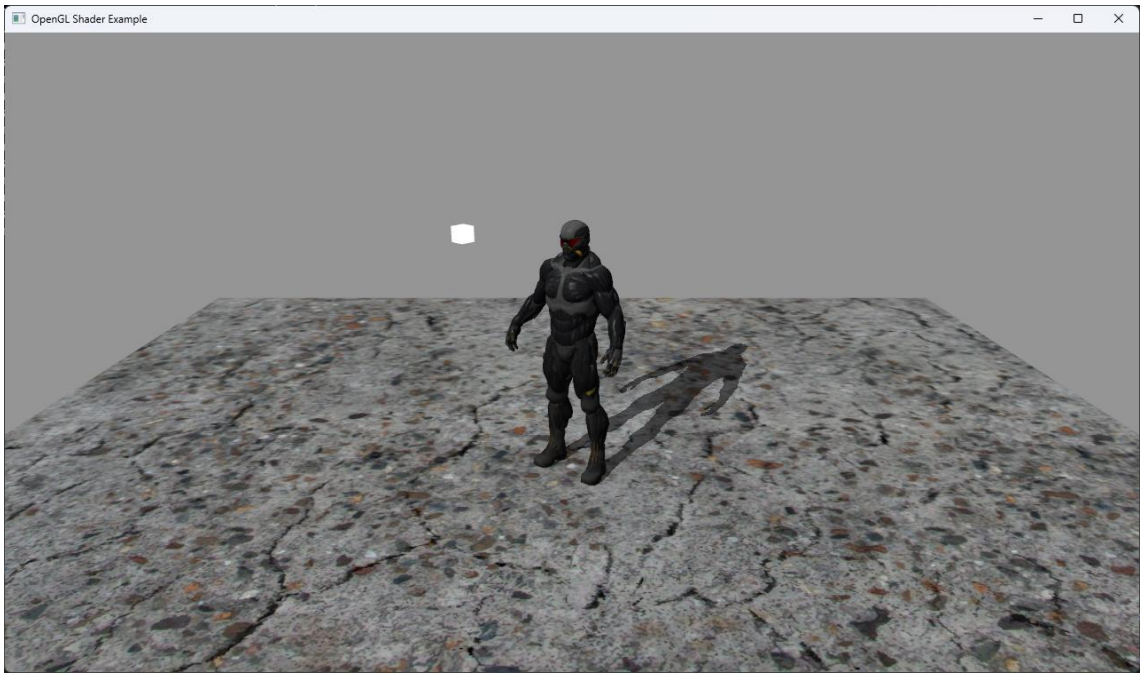

Figure 2 – Rendering with shadows

Shadow mapping is a multi-pass technique that uses depth textures to decide whether a point lies in shadow or not. The key is to observer the scene from the light source's point of view instead of the final viewing location. Any part of the scene that is not directly observable from the light's perspective will be in shadow.

The main passes of the algorithm are described below:

1. Render the scene from the light's point of view. It does not matter how the scene looks like (color information); the only relevant information at this point are the depth values. These values are stored in a *shadow map* (or *depth map*) and can be obtained by creating a depth texture, attaching it to a framebuffer object and rendering the entire scene (as viewed by the light) into this object. This way, the depth texture is directly filled with the relevant depth values.
2. Render the scene from the final point of view (the camera's position) and compare the depth of each visible fragment (projected into the light's reference frame) with depth values in the shadow map. Fragments that have a depth greater than what was previously stored in the depth map are not directly visible from the light's point of view and are, thus, in shadow.

2.1 Generating the depth map

The first pass of the shadow mapping algorithm generates a depth map of the entire scene viewed from the light's perspective. This map can be saved directly into a texture by attaching the texture to a framebuffer object and rendering directly into it.

OpenGL does all its rendering (color, depth and stencil information) into a framebuffer. The colors stored into the framebuffer are used by the screen when displaying the visual content our applications generate. Framebuffer objects allow us to create our own framebuffers and instead of rendering into

special areas of GPU memory, we can write information (color, depth and stencil information) into textures. These textures are attached as the color, depth and stencil buffers for our framebuffer objects.

2.1.1 Creating a framebuffer object

A framebuffer object is created as shown below:

```
GLuint shadowMapFBO;
//generate FBO ID
glGenFramebuffers(1, &shadowMapFBO);
```

To attach a texture as a depth buffer for our framebuffer object, we should first create the texture:

```
GLuint depthMapTexture;
//create depth texture for FBO
glGenTextures(1, &depthMapTexture);
glBindTexture(GL_TEXTURE_2D, depthMapTexture);
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT,
        SHADOW_WIDTH, SHADOW_HEIGHT, 0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
float borderColor[] = { 1.0f, 1.0f, 1.0f, 1.0f };
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
```

The depth texture is created using GL_DEPTH_COMPONENT as the texture type. Also, to avoid unwanted visual artefacts when generating shadows, the texture wrap mode is set to GL_CLAMP_TO_BORDER. SHADOW_WIDTH and SHADOW_HEIGTH are constants and represent the depth map's resolution, which should be at least the size of the OpenGL window's framebuffer. Too low resolutions tend to lead to poorer results and too high resolutions tend to waste memory.

Once the depth texture is created, it should be attached as the framebuffer object's depth buffer:

```
//attach texture to FBO
glBindFramebuffer(GL_FRAMEBUFFER, shadowMapFBO);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, depthMapTexture,
0);
```

Since the first pass of the shadow mapping algorithm does not require a color or stencil attachment but a framebuffer object would not be complete without them, we can explicitly bind nothing to these attachment points:

```
glDrawBuffer(GL_NONE);
glReadBuffer(GL_NONE);
```

Once a framebuffer object is complete, we should unbind it until we are ready to use it:

```
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

With a properly configured framebuffer object, the complete rendering stage, using the depth map, is as follows:

```
// render to depth map
Change the viewport to accommodate the depth texture size;
Bind the framebuffer object;
        Clear the depth buffer;
        Configure shader (for filling the depth texture) and transformation matrices (for the light space);
        Render the scene;
Unbind the framebuffer object;
// render to the main framebuffer
Change the viewport to accommodate the OpenGL window's framebuffer;
        Clear the color and depth buffer;
        Configure shader (for rendering with shadows) and transformation matrices;
        Bind the depth texture;
        Render the scene;
```

2.1.2 Light space transforms

The first pass should render the scene from the light's point of view. This means that the coordinates of all objects in the scene must be transformed to be relative to the light instead of the camera. Thus, we can use the same function as for the Eye space transformation, **lookAt**, but with the light's position as the first parameter. For this laboratory, we shall use a directional light. Since directional lights have no position (they are placed at infinity), we can use as a position any point along the light's direction, including the direction itself (interpreted as a point):

```
glm::mat4 lightView = glm::lookAt(lightDir, glm::vec3(0.0f), glm::vec3(0.0f, 1.0f, 0.0f));
```

Because all rays coming from a directional light are parallel, we shall use an orthographic projection to avoid any perspective deforming:

```
const GLfloat near_plane = 0.1f, far_plane = 6.0f;
glm::mat4 lightProjection = glm::ortho(-1.0f, 1.0f, -1.0f, 1.0f, near_plane, far_plane);
```

The final transformation matrix for the light's space is:

```
glm::mat4 lightSpaceTrMatrix = lightProjection * lightView;
```

2.1.3 Rendering to the depth map

The vertex shader that transforms all vertices into the light's space is presented below:

```
#version 410 core

layout(location=0) in vec3 vPosition;

uniform mat4 lightSpaceTrMatrix;
uniform mat4 model;

void main()
{
    gl_Position = lightSpaceTrMatrix * model * vec4(vPosition, 1.0f);
```

```
}
```

In OpenGL 4, it is legal to have no fragment shader only when rasterization is turned off. Thus, a simple fragment shader that writes dummy output should be implemented[1]:

```
#version 410 core

out vec4 fColor;

void main()
{
	fColor = vec4(1.0f);
}
```

Rendering into the depth map now becomes:

```
//render the scene to the depth buffer

depthMapShader.useShaderProgram();

glUniformMatrix4fv(glGetUniformLocation(depthMapShader.shaderProgram, "lightSpaceTrMatrix"),
	1,
	GL_FALSE,
	glm::value_ptr(computeLightSpaceTrMatrix()));

glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
glBindFramebuffer(GL_FRAMEBUFFER, shadowMapFBO);
glClear(GL_DEPTH_BUFFER_BIT);

RenderTheScene();

glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

If we were to visualize the depth map at this point, it would look like in Figure 3.

2.2 Rendering with shadows

Once the depth map has been created, it can be used to render the scene with shadows. When comparing the depth of the current fragment with the depth existent in the depth map, both must be in the light's reference system. Since the first one is not, it must be transformed using the same matrix as when generating the map. Transformations should be performed in the vertex shader:

```
…
out vec4 fragPosLightSpace;
…
uniform mat4 lightSpaceTrMatrix;
…
fragPosLightSpace = lightSpaceTrMatrix * model * vec4(vPosition, 1.0f);
…
```

[1] We could also have an empty fragment shader since we only care about depth values, which are generated behind the scenes by OpenGL anyway.

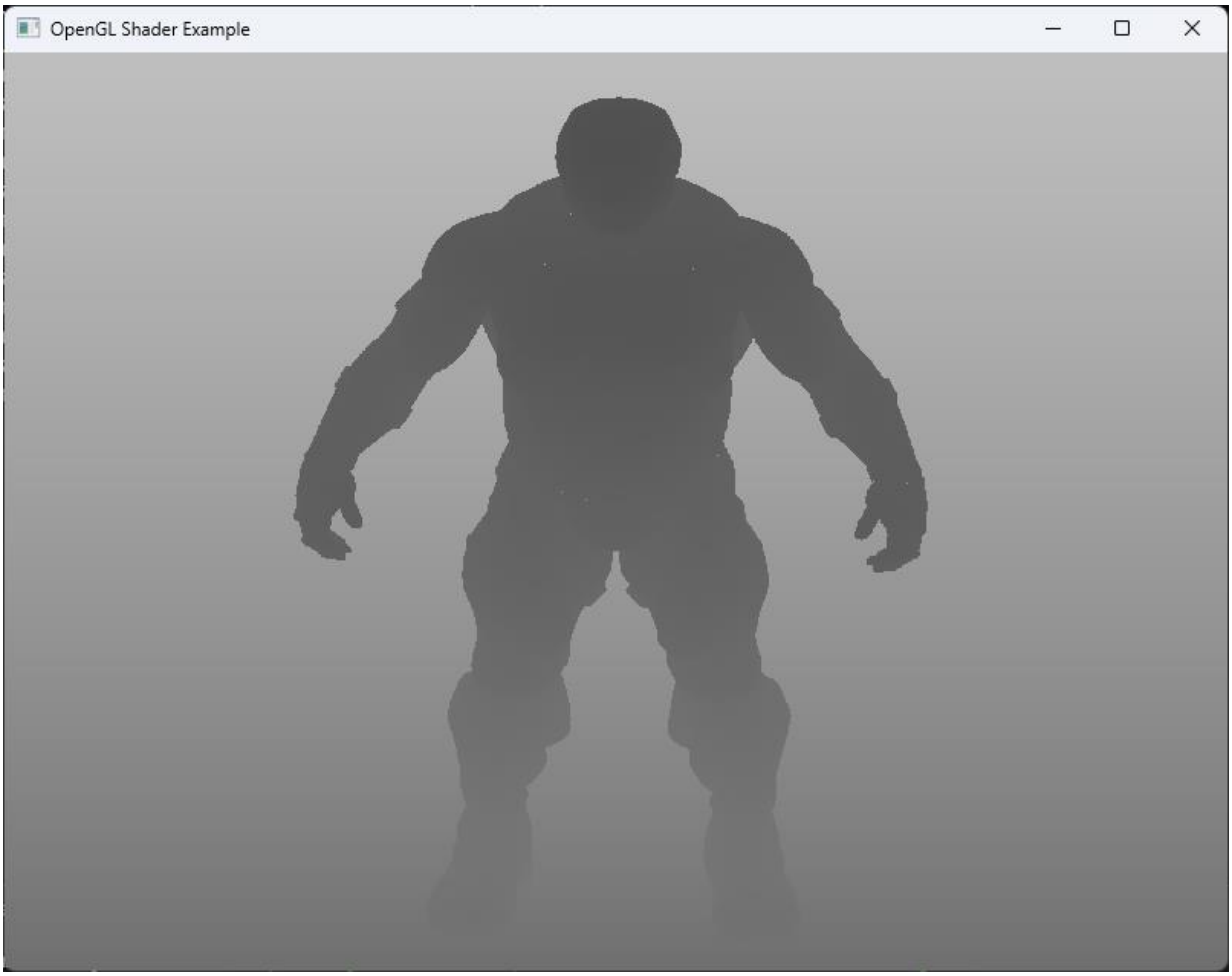

Figure 3 – Depth map example

The actual shadow computation will take place in the fragment shader. The final color of each fragment (for the diffuse and specular components) will be modulated with the computed shadow value (either 1.0 or 0.0)

```
…
float computeShadow()
{
…
}
…
//modulate with shadow
shadow = computeShadow();
vec3 color = min((ambient + (1.0f - shadow)*diffuse) + (1.0f - shadow)*specular, 1.0f);
```

To perform the actual computation of the shadow, the first thing we should do is to transform the fragment's position into the light's clip space normalized coordinates. Since the transformation to the light's clip space was done in the vertex shader, what remains is to apply perspective division to get the normalized coordinates:

```
in vec4 fragPosLightSpace;

…
float computeShadow()
{
…
// perform perspective divide
vec3 normalizedCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
…
}
…
```

This returns the current fragment's position in the [-1,1] range. Since fragment depth values are in the [0,1] range, we should transform the normalized coordinates accordingly:

```
…
float computeShadow()
{
…
// Transform to [0,1] range
normalizedCoords = normalizedCoords * 0.5 + 0.5;
…
}
…
```

We can now sample the existing depth in the depth map using these coordinates since they now correspond with the NDC coordinates from the first pass:

```
…
uniform sampler2D shadowMap;
…
float computeShadow()
{
…
// Get closest depth value from light's perspective
float closestDepth = texture(shadowMap, normalizedCoords.xy).r;
…
}
…
```

This gives us the closest depth value from the light's perspective. Next, we take the depth (in the light's reference system) of the current fragment:

```
…
float computeShadow()
{
…
```

```
// Get depth of current fragment from light's perspective
float currentDepth = normalizedCoords.z;
…
}
…
```

We can now compare the two values. If the current fragment's depth is greater than the value in the depth map, the current fragment is in shadow. Else, it is illuminated:

```
…
float computeShadow()
{
…
// Check whether current frag pos is in shadow
float shadow = currentDepth > closestDepth  ? 1.0 : 0.0;
…
}
…
```

When using this shader, with the properly bound depth texture, the output should be similar to the one in Figure 4.

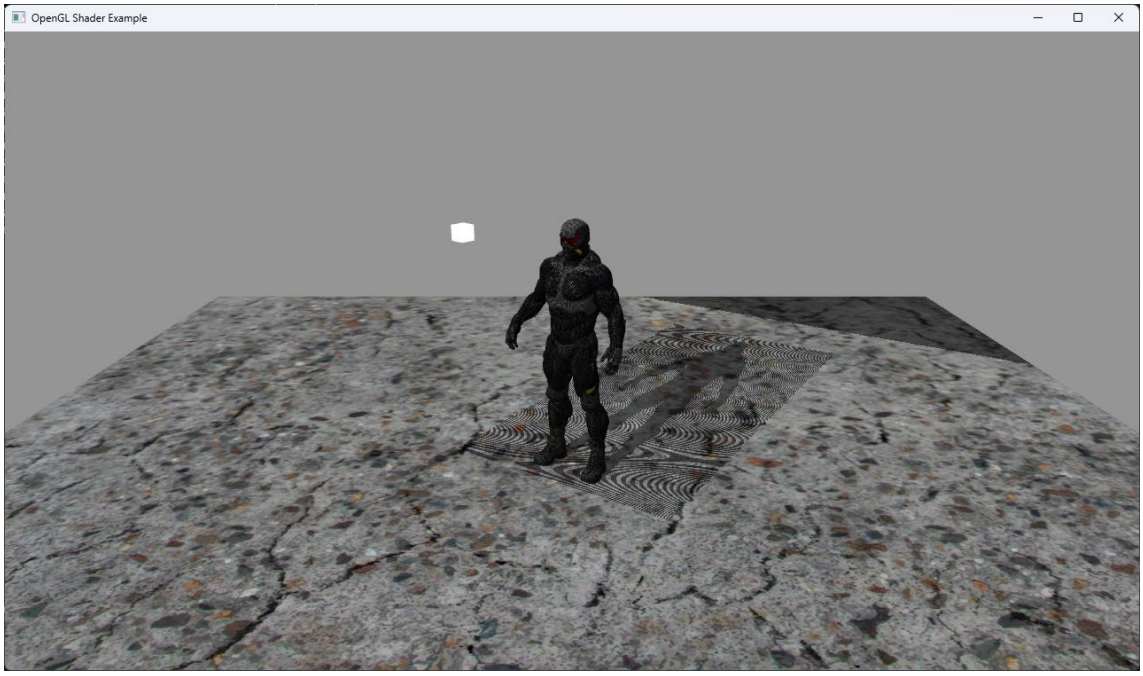

Figure 4 – Rendering with shadows

2.3 Improving quality

2.3.1 Shadow acne

On a closer look, we can spot an unwanted pattern as illustrated in Figure 5.

Figure 5 – Shadow acne

This happens due to the limited resolution of the depth map and it occurs when multiple fragments sample the same value from the depth map. We can usually solve this using a small bias to the fragment's depth, either a fixed value or a value based on the angle the surface makes with the direction of the light (this solves issues for surfaces that have a steep angle to the light):

```
…
float computeShadow()
{
…
// Check whether current frag pos is in shadow
float bias = 0.005f;
float shadow = currentDepth - bias > closestDepth  ? 1.0f : 0.0f;
…
}
…
```

or

```
…
float computeShadow()
{
…
// Check whether current frag pos is in shadow
float bias = max(0.05f * (1.0f - dot(normal, lightDir)), 0.005f);
float shadow = currentDepth - bias > closestDepth  ? 1.0f : 0.0f;
…
}
…
```

The visual output should now become similar to the one in Figure 6.

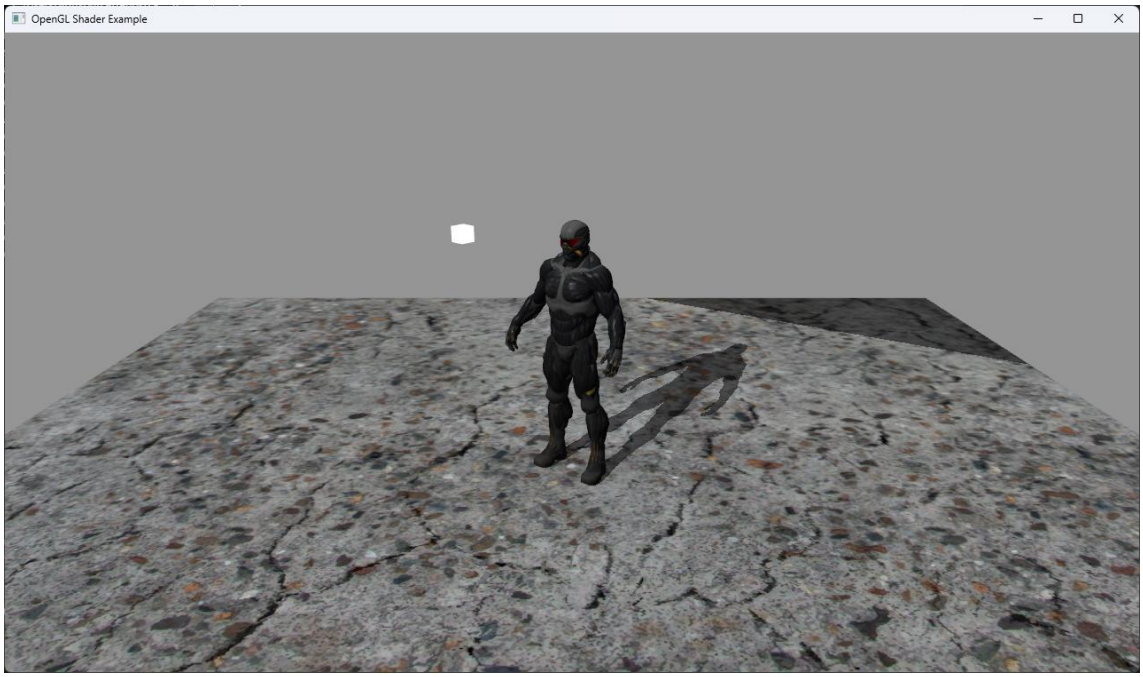

Figure 6 – No shadow acne

2.3.2 Over sampling

Notice that there are areas in shadow which actually should be illuminated (the far-right corner in Figure 6). This happens because fragments outside the depth map's frustum have a depth value greater than 1.0. This can be easily rectified in the fragment shader:

```
…
float computeShadow()
{
…
if (normalizedCoords.z > 1.0f)
        return 0.0f;
…
}
…
```

The visual output should now become similar to the one in Figure 7.

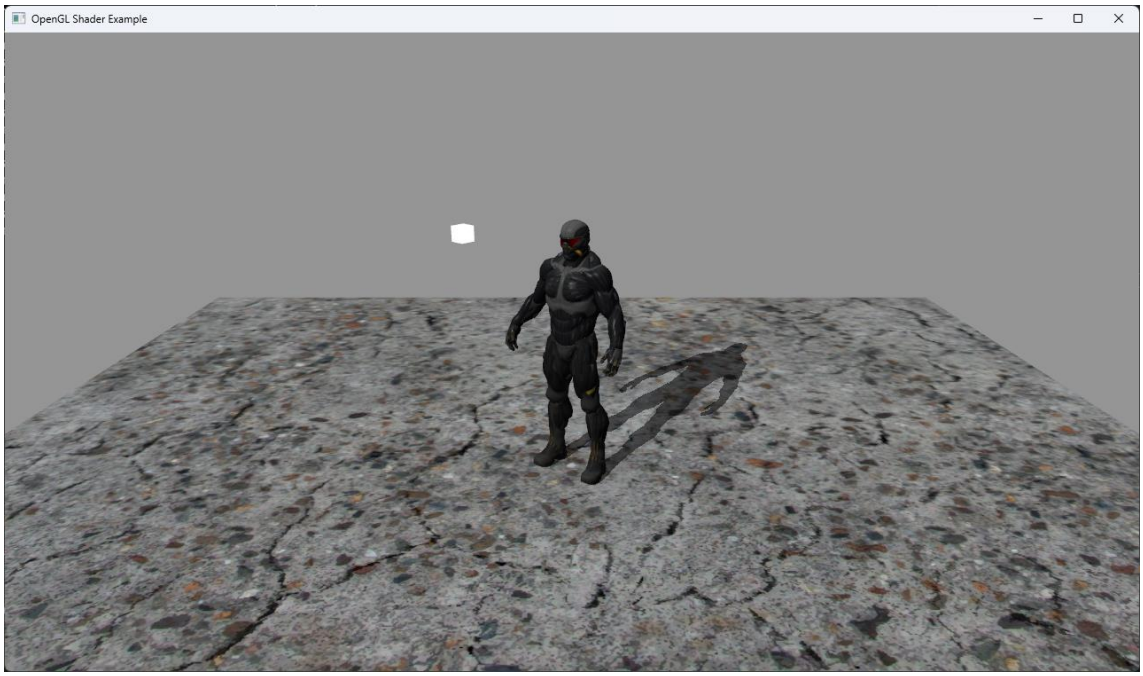

Figure 7 – No over sampling

3 Further reading

- OpenGL Programming Guide 8th Edition – Chapter 7 (Shadow Mapping)
- OpenGL tutorials – Advanced OpenGL (Depth testing, Face culling, Framebuffers), Advanced lighting (Shadows) – http://www.learnopengl.com/

4 Assignment

1. Download the resources from Laboratory work 9
2. Add the ground object to your scene
3. Add a framebuffer object to the existing application
4. Add a depth texture to the framebuffer object
5. Add the necessary code and shaders to render the light's depth map into the depth texture
6. Add the necessary code to render the scene with shadows
7. Make the necessary adjustments to solve the various visual issues encountered (shadow acne, over sampling)