02561 Computer Graphics

-

Project: **Worksheet 10**

Thomas Sandfeld Nielsen - s145351
Lennart Staalegaard Windfeld - s144455

December 2018

Introduction

For our project we chose to complete Worksheet 10, Environment mapping & bump mapping, which consists of four parts.

- Part 1 is about texturing a sphere with a cubemap.
- Part 2 is about drawing a textured quad in the background whilst also rendering the sphere from part 1.
- Part 3 is about making the sphere reflective, but not the background.
- Part 4 is about using bump mapping on the sphere, but using a simple image as a texture.

We managed to solve all parts, and the solutions are shown in the next section.

Solution

Part 1

To load the images we use a for loop and in this we have assigned an onload-method to each of the images in the loop.

The method which is triggered onload is where we create the tetrahedron, similar to Worksheet 6, part 3. Using a predefined subdivisionlevel of 5, we initialize the triangles of the tetrahedron. We use a set of three-dimensional vectors as the vertices. Then recursively we create each of the triangles, by push the vertices of the triangles to our `pointsArray`.

After initializing the buffers, we set up the texture. For each of the six textures we bind the texture to the image which corresponds to that texture.

In the fragment shader, we can then use the the normals to look up the correct texture, and thus color the sphere correctly according to the texture images.

Part 2

To include a background which is close to the far plane of the view spectrum, we draw a screen-filling quad using clip-coordinates.

We have created a uniform `mat4` matrix, `mTexture`. In the vertex shader we now calculate the global texture coordinates by multiplying the `mTexture` with the `vPosition`. In this way we transform the vertex positions into texture coordinates. In the fragment shader we can then utilize the texture coordinates instead of the normals, which we used in part 1.

`mTexture` is an identity matrix for the sphere, and thus in order to create the `mTexture` the background we need to do a transformation from the clip space coordinates, which the background is defined by, to world space directions, which is what the sphere is defined by.

The vertices of the background quad is in clip space, which means that all vertex coordinates is in the range -1 to +1, inclusively. However, we are drawing the sphere and texture in the global space, so this is why we need to transform the clip coordinates. A thing to consider is that WebGL always converts coordinate systems into clip space in the vertex shaders.[1] So in a sense we are actually converting the background quad's vertex coordinates from clip space to world space and then to clip space again!

Part 3

For this part we want to make the sphere reflective, but not the quad, so we have created a boolean, `isReflective`, in the fragment shader to distinguish this.

If `isReflective` is equal to true, then we calculate the direction of the incidence by subtracting the eye vector from the global texture coordinates, because the incident "ray" goes from the eye to the object's surface. Then we calculate the reflection using GSLS ES's built-in method `reflect(vec3 incident, vec3 normal)`, where we pass the incident vector we just calculated and the global texture coordinates. Finally the `fragColor` is set to the `texturecube` function with `texture` and `reflected` as parameters.

If `isReflective` is **not** equal to true, we simply set the `fragColor` the `texturecube` function with `texture` and global texture coordinates as parameters.

In our code we set the `isReflective` boolean in the render function by using the `unform1i` function. We set it to false before drawing the background quad, and to true when drawing the sphere.

Part 4

Part 4 is about bump mapping, and for this we use the image `normalmap.png`. As in part 1 we load the images one by one and use the onload method to keep track of when all images are loaded. Here we simply append the path of the normalmap image to the other image paths.

In our JavaScript code we first use `gl.activeTexture(gl.TEXTURE0)` and initialize our sphere texture. In the same manner we initialize the normalmap texture by using `gl.activeTexture(gl.TEXTURE1)`, and configuring the texture.

The biggest change to our code is in the fragment shader. Like in part 3 we have our conditional if-statement in order to compute reflection for the elements we want.

As in exercise 6.3 we start by calculating `u` and `v` for the sphere.

As stated in the Worksheet description, the normal we retrieve is in tangent space, but we need it in world coordinates in order to use it in place of the sphere and calculating reflection direction. This is done in by using the given

[1] https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API/WebGL_model_view_projection

```
if (isReflective) {
    float u = 0.5 + atan2(normals.z, - normals.x) / (2.0 * pi);
    float v = 0.5 - (asin(normals.y) / pi);

    vec3 tangent = texture2D(vNormals, vec2(u,v)).xyz * 2.0 - 1.0;
    normals = rotate_to_normal(normals, tangent);

    vec3 incident = worldPos - vEye;
    normals = reflect(incident, normals);

    gl_FragColor = textureCube(texture, normals);
} else {
    gl_FragColor = textureCube(texture, fTexCoords);
}
```

Figure 1: Snippet from the `main()` function in our fragment shader

`rotate_to_normal()` function after retrieving the tangent space vector. This can be seen in figure 1.

Afterwards we calculated the incidence and reflection just like in part 3.

Results

Figure 2, 3, 4 and 5 shows our results for the four parts of worksheet 10. We believe that we have solved the parts correctly.

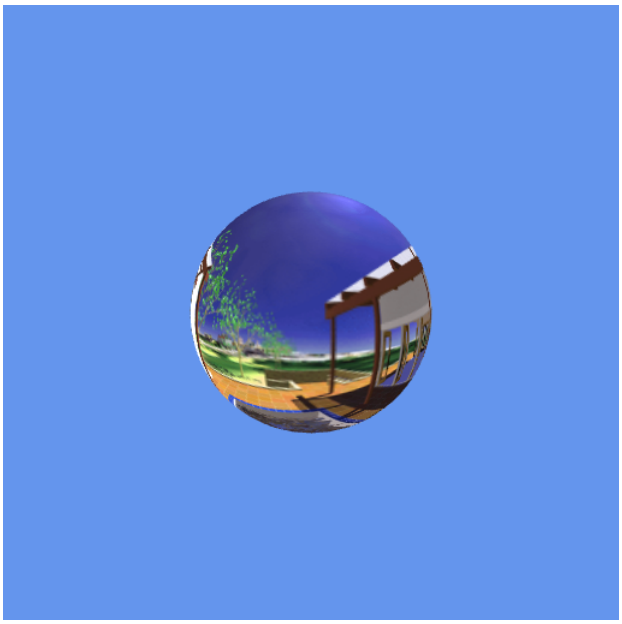

Figure 2: Our result for part 1

Figure 3: Our result for part 2

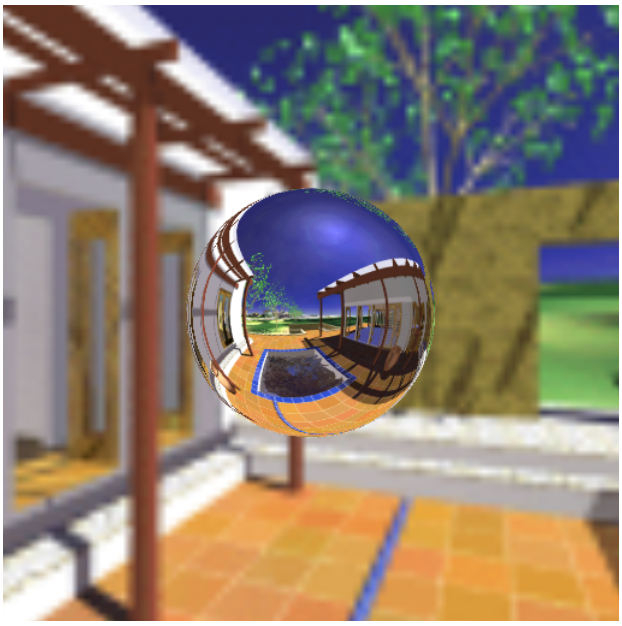

Figure 4: Our result for part 3

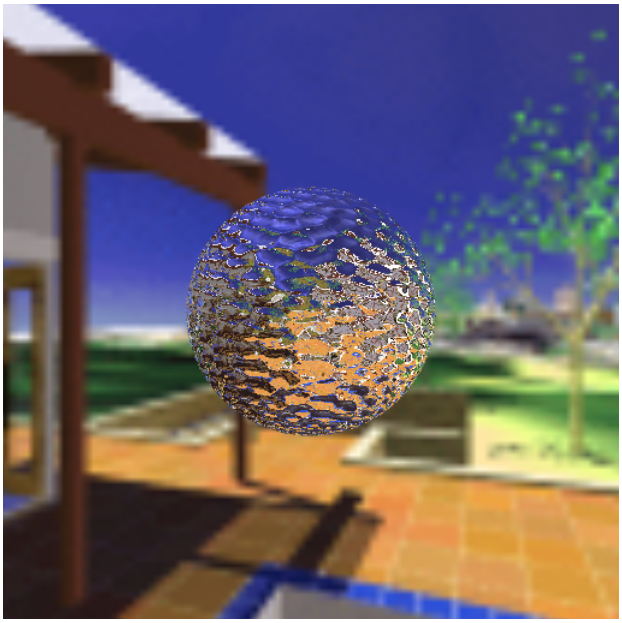

Figure 5: Our result for part 4