Computer Graphics Lab 4

Name: James Davies
Student ID: 200682354

October 2023

Contents

1 Exercise A

1.1 A1 - Texture setup and rendering

A1 is primarily concerned with the set up of the scene and the object we will be rendering, 'spot'. The task also deals with the texture mapping onto the model, which is achieved via using internal addresses within the shader and 'pointing' towards them using the uniform1() method. The method is used to assign values to uniform values, but in the line given in the brief we can use it to bind the shader texture to the gl.TEXTURE0 address after it's created further on in the program.

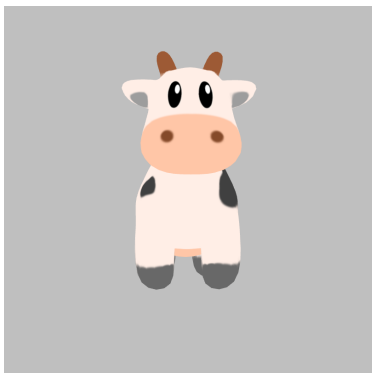

Figure 1: A1i

Figure 2: A1ii

To implement the actual mapping itself, we need to use a function in the fragment shader called texture2D(), which takes the sampler2D object (texture) and the vec2 coordinates (map) as argument. What is returned from this function is called a 'texel'; a vec4 colour value for the given coordinates. In A1i, we directly assign the output of this function call to gl_FragColor, losing all lighting effects in the process. However, we circumvent this by instead creating a new vec4 variable **material_colour** of which we pass to the various ambient/diffuse calculations.

1.2 A2 - Texture coordinates

A2 revolves around a colour-based visualisation of the texture coordinates in the vec2 map (s and t). This is implemented via taking the vec4 object used for passing rbg + alpha to gl_FragColor in the fragment shader, and instead passing the s coordinate (map.s) as red, and the t coordinate (map.t) as green.

Figure 3: A2

The resulting output shows that each segment of the object is treated as independent when it comes to mapping the texture coordinates; there are clear dividing lines between the two sides of spot, and the legs are different colours. The darker head could be because of the specific nature of the coordinate data (i.e. s and t) relative to the actual texture mapping. Given more time I would have liked to delve into this further.

1.3 A3 - Texture filtering

In A3, we now visualise a striped pattern as texture, via use of a new model (part A3i) and manipulation of the texture filters within the JS file (A3ii).

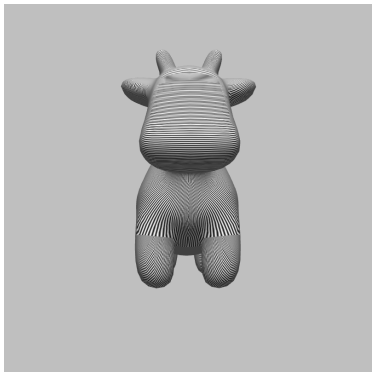

Figure 4: A3i

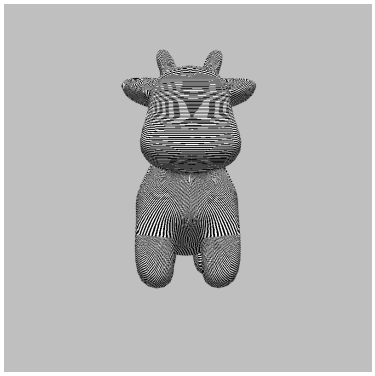

Figure 5: A3ii

Between A3i and A3ii, we change (via the given code in the brief) the texture filters by altering TEXTURE_MIN_FILTER and TEXTURE_MAG_FILTER. In A3i, both of these settings are set to mipmap and bilinear respectively; in A3ii, we change both of these to a more basic rendering (nearest neighbour). Mipmap is concerned with reducing computational load and avoiding expensive operations, and is a standard technique in filtering. Bilinear filtering uses a weighted average of the surrounding texels, which gives off a smooth gradient from texel to texel. By comparison, nearest neighbour is a limited filtering methodology; fast, but results in aliasing and other artifacts during rendering.

We can see a highlighted example of this type of aliasing on the back of the head of our model when we switch over to neighest neighbours. This does not occur in A3i because of the benefits of mipmap/bilinear filtering methods regarding computation/smooth gradient change respectively.

1.4 A4 - Texture editing

Task A4 revolves around taking a different model (banana), copying the original texture for that model, and then using an image editing program to alter one of the surfaces within the texture png. When it is now mapped onto the surface of the model, we will be able to visualise the alterations as well on whichever surface we have chosen.

Figure 6: A4i

Figure 7: A4ii

This demonstrates how texture mapping actually works in regards to how a png file is mapped onto a model. Each of the 'sides' of the banana model have respective parts of the png file; I picked the bottom right 2D surface within the png, and that was mapped to a particular side of the model rendering.

2 Exercise B

2.1 B1 - Skyboxes

The B1 task is concerned with demonstrating the functionality of gl.DEPTH_TEST; with B1i showing the skybox rendering itself, and B1ii demonstrating how the spot object is able to intersect with the skybox due to gl.DEPTH_TEST being enabled via the function call .enable().

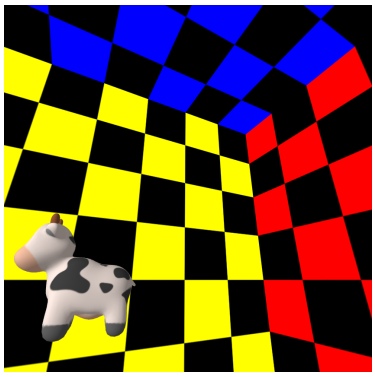

Figure 8: B1i

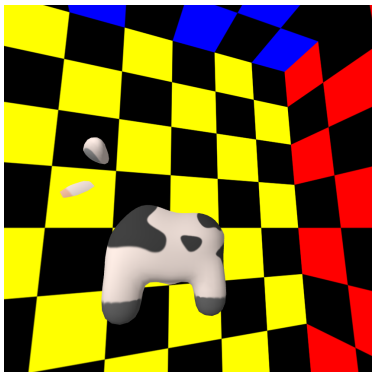

Figure 9: B1ii

Depth testing makes use of the fact that depth (z) values of fragments are stored in a buffer called the depth-buffer. This maintains all of the fragment depths within the scene, and in this case (and many others) it is desirable behaviour due to the fact it would be computationally inefficient to render fragments that are behind the cubemap panorama. The DEPTH_TEST setting is disabled by default, and so we need to enable it before we render the main model again.

2.2 B2 - Reflection mapping

Continuing with the skybox/spot rendering, task B2 now considers the inclusion of reflection mapping to create a more realistic model. We do this by augmenting the colour assignment to gl_FragColor in the fragment shader; with B2i concerned with demonstrating reflection mapping on the model in a more natural way, and B2ii showing 'pure reflection mapping'.

Figure 10: B2i

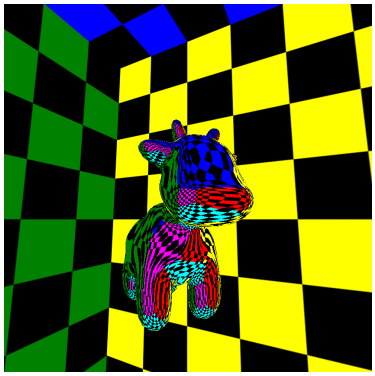

Figure 11: B2ii

FragColor is taken as the sum of the ambient, diffuse, specular and reflection terms (Fig. 12). The others is calculated in isolation outside of the reflection, instead using lighting and the texture (in this case) for defining the terms. The reflection however (Fig. 13), uses the cubemap and the coordinates of the reflection vector (r) to determine what should be reflected at what point on the model.

```
// combined colour
if(render_texture) {
    gl_FragColor = vec4((0.5 * ambient +
                         0.5 * diffuse +
                         0.01 * specular +
                         0.1 * reflection_colour).rgb, 1.0); // B2

}
```

Figure 12: FragColor declaration for pure reflection mapping

The pure reflection mapping occurs when in the else statement (from Fig. 12 if statement) states that gl_FragColor should **only** be the reflection mapping term from Fig. 13.

```
// reflected background colour
vec4 reflection_colour = textureCube(cubemap,vec3(-r.x,r.y,r.z));
```

Figure 13: FragColor declaration for pure reflection mapping

3 Exercise C

3.1 C1 - RGB transformations

C1 performs a simple arithmetic operation on the RGBA components, the result of which renders RGB black to white and vice versa, with the other component (alpha) left unchanged by the operation.

Figure 14: C1

We first start by taking the original declaration for gl.FragColor and then defining it outside of the if statement, and assigning it to a vec4 object. We then take the rgb values and subtract them from a vector (1.0, 1.0, 1.0) which gives us the inverted colour scheme regarding black/white (Fig. 15). This can then be used in the FragColor definition, alongside the alpha value 1.0.

```
// C1 variables
vec4 original = vec4((0.5 * ambient + 0.5 * diffuse + 0.01 * specular + 0.1 * reflection_colour).rgb, 1.0);
vec3 inverted = vec3(1.0) - original.rgb;
```

Figure 15: C1 variable declarations

3.2 C2 - Alpha transformations

C2 now aims to alter the other component in the RGBA vec4 object, the alpha component. This is demonstrated via rendering the black parts of the texture as transparent (0.0) instead of opaque (1.0).

Figure 16: C2

This is achieved via augmenting the gl.Frag_Color declaration, in which we first declare a new vec4 object. Within this vec4 declaration we use the original vec3 defined for C1, and then take the length (length() function) of the original added to 0.0. This float plus the output of the length() function serves as our new alpha value in the RGBA, in which the darker values (i.e. 'shorter' vectors) have a smaller alpha value and therefore will be transparent. Black values will be fully transparent as the length of the vector will be zero. The if statement is necessary because the fragments to the rear of the model won't be rendered in the same way, and therefore will be opaque and visible through the model.

```
// ------------------------------------------------------------------------------ //
// C2
// ------------------------------------------------------------------------------ //

gl_FragColor = vec4(vec3(original), 0.0 + length(vec3(original.rgb))); // C2
```

Figure 17: C2 FragColor

3.3 C3 - Gamma transformations

We then utilise one of the functions we have studied in the lectures (gamma_transform) in the fragment shader. This function serves to demonstrate how we can improve realism of the model by taking the RGB components of a vec4 object and taking them to a given power. The powers that we use in the task are 0.5 for C3i and 2.0 for C3ii.

Figure 18: C3i

Figure 19: C3ii

The process of gamma transformations relies on the fact that our display devices (amongst others) do not have a linear-based response, and neither do our eyes. We notice changes in dark tones more easily than if we were to perform the same change in a light tone. The gamma transformations serve as a methodology to reflect this nonlinear characteristic of our eyesight and offset the nature of the screen. The resulting output is a far more natural image/rendering regarding black and white values.

3.4 C4 - Vignetting

In task C4 we implement a vignette effect, in which the edges/corners of the scene are darkened. This is achieved by creating a function of our own within the fragment shader, in which we take the fragment coordinates as argument (vec4 frag_coord) and a float value is returned.

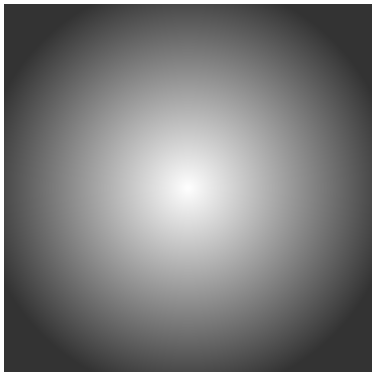

Figure 20: C4i

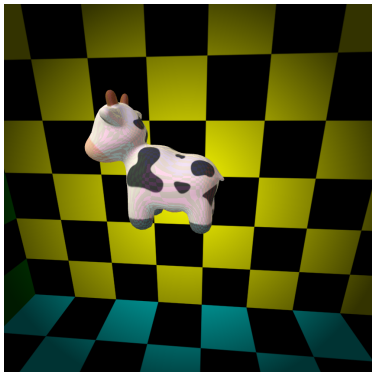

Figure 21: C4ii

To implement the vignette function, we first specify the radius of the vignette itself; 1200.0 as a float value was slightly arbitrary and came via experimentation for the correct effect.

```
// vignette function
float vignette(vec4 frag_coord) {

    float ret_num;
    float radius = 1200.0/2.0;
    float centre_x = 850.0/2.0;
    float centre_y = 850.0/2.0;
    float x = gl_FragCoord.x - centre_x;
    float y = gl_FragCoord.y - centre_y;

    ret_num = 1.0 - sqrt(pow(x,2.0) + pow(y,2.0)) / radius;

    if(ret_num < 0.2){
        return 0.2;
    } else {
        return ret_num;
    }
```

Figure 22: Vignette Function

We then need to specify the centre of the image, using the hard-coded value (850x850) of the scene size divided by 2. The x and y coordinates of each fragment is then taken as gl_FragCoord.x/y - the centre values. Finally we

utilise the mathematical equation from the lecture and apply it to our values from within the function to get:

$$1.0 - \sqrt{\frac{x^2 + y^2}{radius}}$$

This allows us to specify that the further the fragment is from the centre of the vignette, the darker the fragment has to be. The value is 1.0 in the centre, and 0.0 on the outer ring of the vignette.

4 Exercise D

4.1 D1 - Real Images

In D1 we now move to a more realistic scene, utilising the square model instead of the chessboard. We also switch from the spot model to the sphere, which will prove useful for the D2 exercise. We then demonstrate how the model is actually laid out (still the same cube) by switching one of the sides to the chessboard model (chessboard-front).

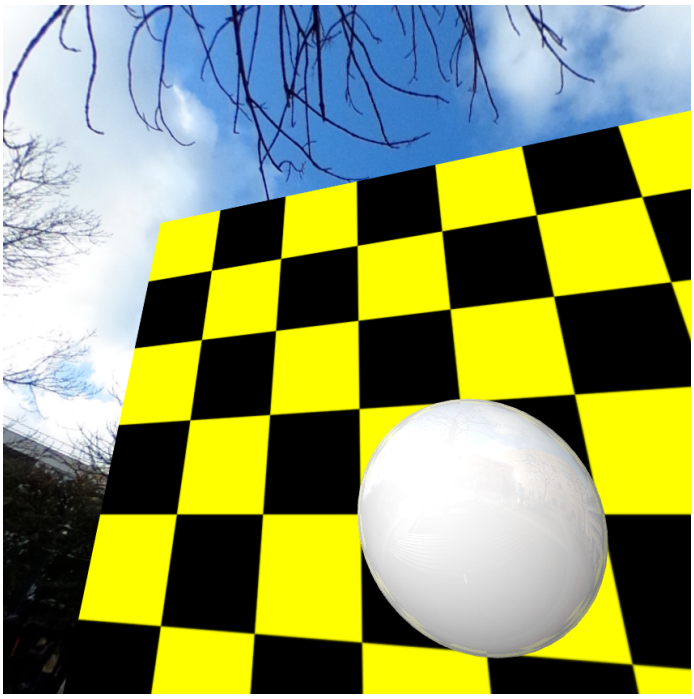

Figure 23: D1

The cubemap panorama is made up of six sides defined within environment.html: right, left, front, back, over and under.

4.2 D2 - Reflection mapping revisited

Exercise D2 aims to visualise how we can use gamma transformations (via our predefined function in the fragment shader) to enhance realism. In this case we will be using the sphere model and pure reflection mapping to 'replace' the silver sphere in the square model. D2i serves as the default model, with D2ii demonstrating the actual placement of the sphere model over the original sphere.

Figure 24: D2i

Figure 25: D2ii

After closer comparison of the reflections of the photographic and synthetic spheres, it seems as if two of the sides of the cube map panorama have been mixed up. As in the photographic image, there is no tree/benches in the reflection; in the synthetic model there is a large tree/bench clearly reflected.

```
} else {
    // reflection only
    gl_FragColor = gamma_transform(reflection_colour, 1.2); // D2
}
```

Figure 26: D2 Gamma Transformation

In regards to the gamma transformation, the value used was 1.2 (Fig. 21). Although not perfect, this value was a compromise between realism and the dominant colours involved in the pure reflection (blue sky) from becoming too dark. The silver nature of the sphere would have been more realistic had the gamma value been higher; the final result is a trade off.

4.3 D3 - Combined rendering

D3 ties everything together that we have studied throughout the labs, in which I have implemented the spot model with the cubemap panorama 'leadenhall-1'. We also utilise the vignette function we defined earlier in the lab, also using the gamma_transform() function to create a better sense of realism. This is demonstrated via the D3i rendering, with D3ii serving as representation of the underlying wireframe/mesh of the model.

Figure 27: D3i

Figure 28: D3ii

I did not choose to create a specific function in order to change the rendering mode, instead choosing to hard-code the values during rendering. Given more time I would have implemented this functionality.